DIWAKAR EDUCATION HUB

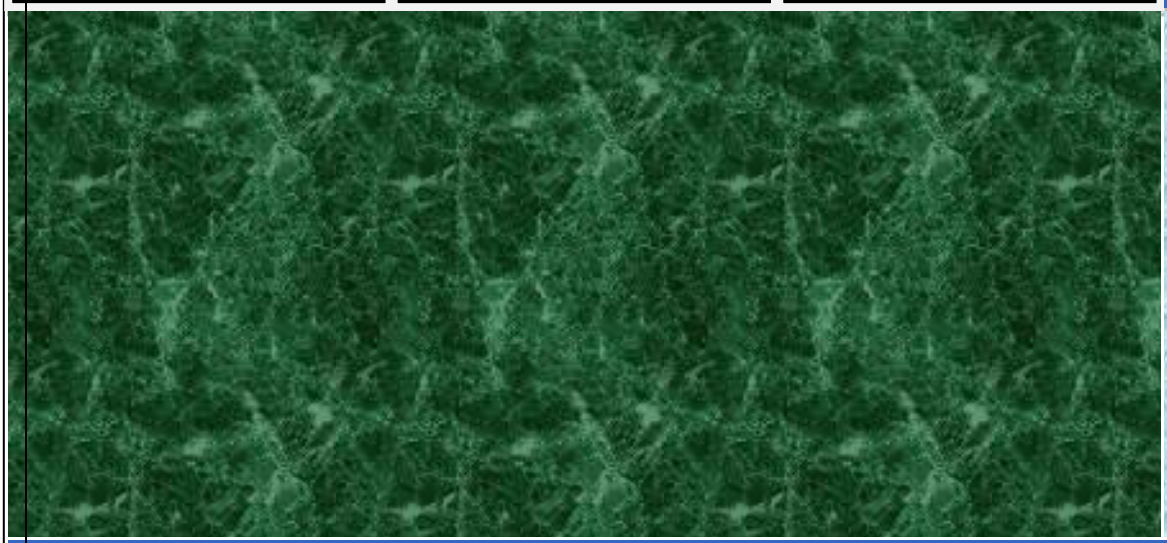
Data Structures and Algorithms
Unit – 7

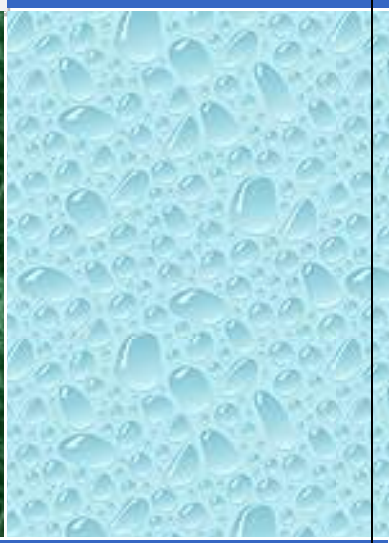
As per updated syllabus

DIWAKAR EDUCATION HUB



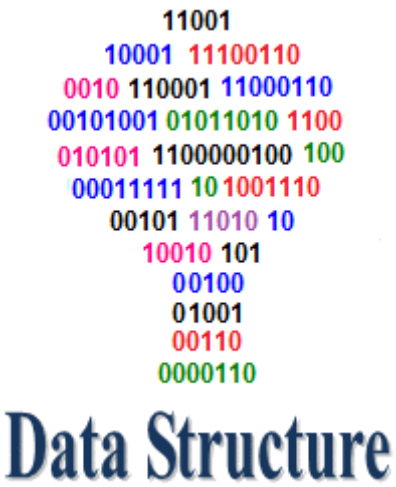
2020





THE LEARN WITH EXPERTIES

Data Structures



Data Structure is a way to store and organize data so that it can be used efficiently.

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organising data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artifical intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vitle role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible

Basic Terminology

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned

Data: Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.

Group Items: Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

Record: Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

File: A File is a collection of various records of one type of entity, for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

Attribute and Entity: An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

Field: Field is a single elementary unit of information representing the attribute of an entity.

Need of Data Structures

As applications are getting complexed and amount of data is increasing day by day, there may arrise the following problems:

Processor speed: To handle very large amout of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

Data Search: Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

Multiple requests: If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process

in order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

Advantages of Data Structures

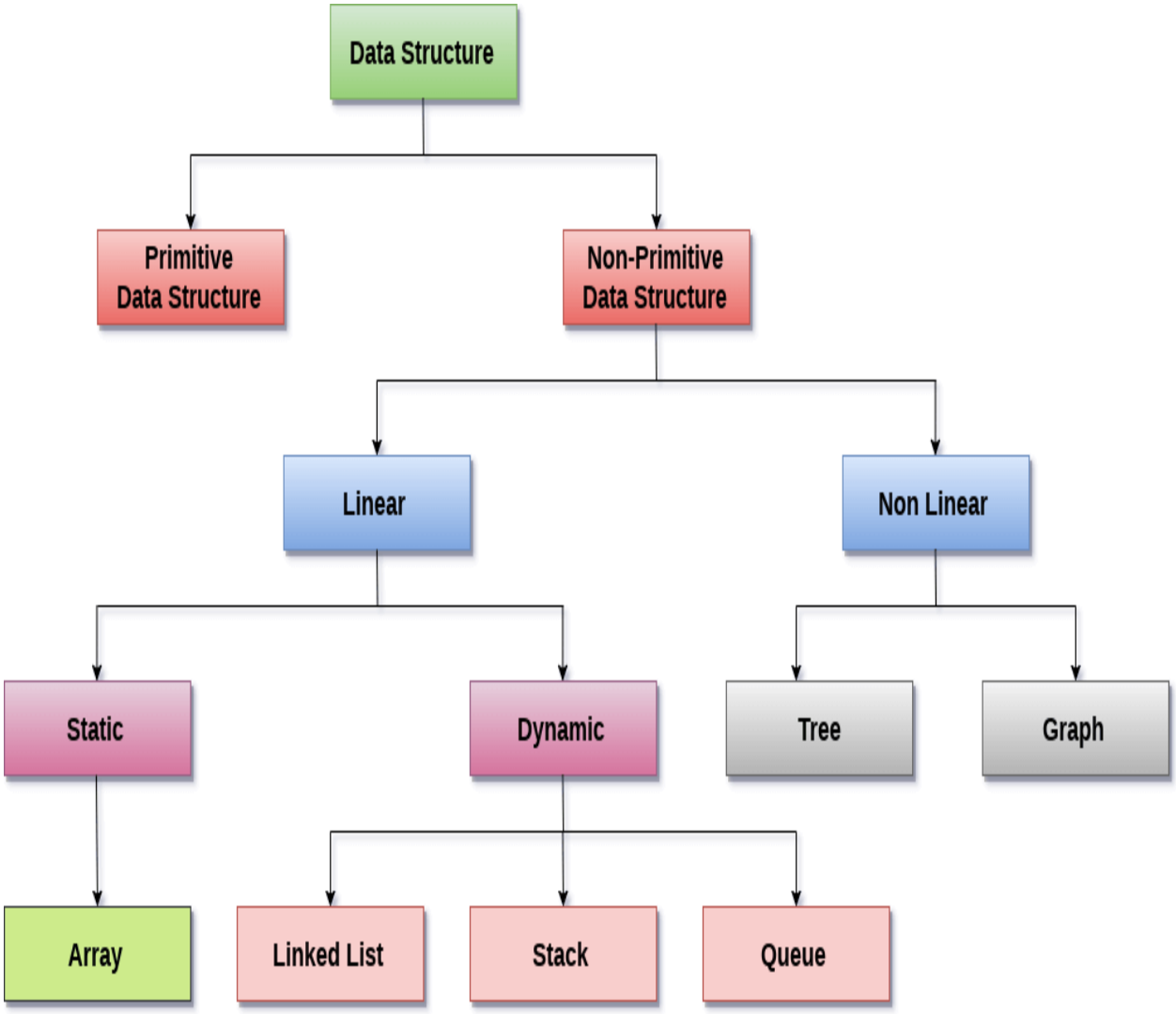
Efficiency: Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a perticular record. In that case, if we organize our data in an array, we will have

to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

Reusability: Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

Abstraction: Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

Data Structure Classification



Linear Data Structures: A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

Arrays: An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are:

age[0], age[1], age[2], age[3],..... age[98], age[99].

Linked List: Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

Stack: Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

Queue: Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.

It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

Non Linear Data Structures: This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non Linear Data Structures are given below:

Trees: Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the herierchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node. Trees can be classfied into many categories which will be discussed later in this tutorial.

Graphs: Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree can not have the one.

Operations on data structure

1) **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will devide that sum by the number of subjects i.e. 6, in order to find the average.

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:**The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure
then **underflow** occurs.

4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching,

Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

5) **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

Algorithm

An algorithm is a procedure having well defined steps for solving a particular problem. Algorithm is finite set of logic or instructions, written in order for accomplish the certain predefined task. It is not the complete program or code, it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudo code.

The major categories of algorithms are given below:

- **Sort:** Algorithm developed for sorting the items in certain order.
- **Search:** Algorithm developed for searching the items inside a data structure.
- **Delete:** Algorithm developed for deleting the existing element from the data structure.
- **Insert:** Algorithm developed for inserting an item inside a data structure.
- **Update:** Algorithm developed for updating the existing element inside a data structure.

The performance of algorithm is measured on the basis of following properties:

- **Time complexity:** It is a way of representing the amount of time needed by a program to run to the completion.
- **Space complexity:** It is the amount of memory space required by an algorithm, during a course of its execution. Space complexity is required in situations when limited memory is available and for the multi user system.

Each algorithm must have:

- **Specification:** Description of the computational procedure.
- **Pre-conditions:** The condition(s) on input.
- **Body of the Algorithm:** A sequence of clear and unambiguous instructions.
- **Post-conditions:** The condition(s) on output.

Example: Design an algorithm to multiply the two numbers x and y and display the result in z.

- Step 1 START
- Step 2 declare three integers x, y & z
- Step 3 define values of x & y
- Step 4 multiply values of x & y
- Step 5 store the output of step 4 in z
- Step 6 print z
- Step 7 STOP

. Alternatively the algorithm can be written as ?

- Step 1 START MULTIPLY
- Step 2 get values of x & y

- Step 3 z← x * y
- Step 4 display z
- Step 5 STOP

Characteristics of an Algorithm

An algorithm must follow the mentioned below characteristics:

- **Input:** An algorithm must have 0 or well defined inputs.
- **Output:** An algorithm must have 1 or well defined outputs, and should match with the desired output.
- **Feasibility:** An algorithm must be terminated after the finite number of steps.
- **Independent:** An algorithm must have step-by-step directions which is independent of any programming code.
- **Unambiguous:** An algorithm must be unambiguous and clear. Each of their steps and input/outputs must be clear and lead to only one meaning.

Array

- Arrays are defined as the collection of similar type of data items stored at contiguous memory locations.
- Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc.
- Array is the simplest data structure where each data element can be randomly accessed by using its index number.
- For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variable for the marks in different

subject. instead of that, we can define an array which can store the marks in each subject at a the contiguous memory locations.

The array **marks[10]** defines the marks of the student in 10 different subjects where each subject marks are located at a particular subscript in the array i.e. **marks[0]** denotes the marks in first subject, **marks[1]** denotes the marks in 2nd subject and so on.

Properties of the Array

1. Each element is of same data type and carries a same size i.e. int = 4 bytes.
2. Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
3. Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of data element.

for example, in C language, the syntax of declaring an array is like following:

1. **int** arr[10]; **char** arr[10]; **float** arr[5]

Need of using Array

In computer programming, the most of the cases requires to store the large number of data of similar type. To store such amount of data, we need to define a large number of variables. It would be very difficult to remember names of all the variables while writing the programs. Instead of naming all the variables with a different name, it is better to define an array and store all the elements into it.

Following example illustrates, how array can be useful in writing code for a particular problem.

In the following example, we have marks of a student in six different subjects. The problem intends to calculate the average of all the marks of the student.

In order to illustrate the importance of array, we have created two programs, one is without using array and other involves the use of array to store marks.

Program without array:

```
1. #include <stdio.h>
2. void main ()
3. {
4.     int marks_1 = 56, marks_2 = 78, marks_3 = 88, marks_4 = 76, marks_5 = 56, marks_6 = 89;
5.     float avg = (marks_1 + marks_2 + marks_3 + marks_4 + marks_5 +marks_6) / 6 ;
6.     printf(avg);
7. }
```

Program by using array:

```
1. #include <stdio.h>
2. void main ()
3. {
4.     int marks[6] = {56,78,88,76,56,89);
5.     int i;
6.     float avg;
7.     for (i=0; i<6; i++ )
8.     {
9.         avg = avg + marks[i];
```

```
10.  }
11.  printf(avg);
12.}
```

Complexity of Array operations

Time and space complexity of various array operations are described in the following table.

Time Complexity

Algorithm	Average Case	Worst Case
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	O(n)	O(n)
Deletion	O(n)	O(n)

Space Complexity

In array, space complexity for worst case is **O(n)**.

Advantages of Array

- Array provides the single name for the group of variables of the same type therefore, it is easy to remember the name of all the elements of an array.

- Traversing an array is a very simple process, we just need to increment the base address of the array in order to visit each element one by one.
- Any element in the array can be directly accessed by using the index.

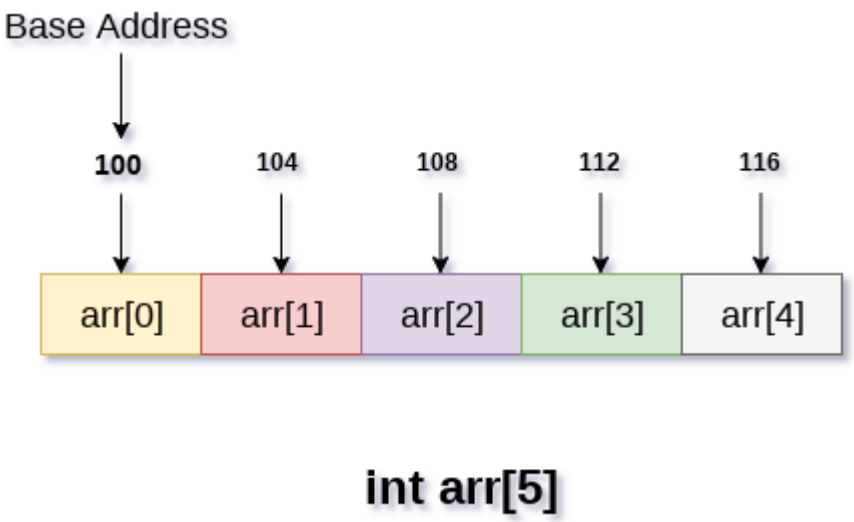
Memory Allocation of the array

As we have mentioned, all the data elements of an array are stored at contiguous locations in the main memory. The name of the array represents the base address or the address of first element in the main memory. Each element of the array is represented by a proper indexing.

The indexing of the array can be defined in three ways.

1. 0 (zero - based indexing) : The first element of the array will be arr[0].
2. 1 (one - based indexing) : The first element of the array will be arr[1].
3. n (n - based indexing) : The first element of the array can reside at any random index number.

In the following image, we have shown the memory allocation of an array arr of size 5. The array follows 0-based indexing approach. The base address of the array is 100th byte. This will be the address of arr[0]. Here, the size of int is 4 bytes therefore each element will take 4 bytes in the memory.



In 0 based indexing, If the size of an array is n then the maximum index number, an element can have is **n-1**. However, it will be n if we use **1** based indexing.

Accessing Elements of an array

To access any random element of an array we need the following information:

1. Base Address of the array.
2. Size of an element in bytes.
3. Which type of indexing, array follows.

Address of any element of a 1D array can be calculated by using the following formula:

1. Byte address of element A[i] = base address + size * (i - first index)

Example :

1. In an array, A[-
   10 ..... +2], Base address (BA) = 999, size of an element = 2 bytes,

2. find the location of A[-1].
3. L(A[-1]) = 999 + [(-1) - (-10)] x 2
4. = 999 + 18
5. = 1017

Passing array to the function :

As we have mentioned earlier that, the name of the array represents the starting address or the address of the first element of the array. All the elements of the array can be traversed by using the base address.

The following example illustrate, how the array can be passed to a function.

Example:

```
1. #include <stdio.h>
2. int summation(int[]);
3. void main ()
4. {
5.     int arr[5] = {0,1,2,3,4};
6.     int sum = summation(arr);
7.     printf("%d",sum);
8. }
9.
10.int summation (int arr[])
11.{
12.    int sum=0,i;
```

```
13.  for (i = 0; i<5; i++)
14.  {
15.      sum = sum + arr[i];
16.  }
17.  return sum;
18.}
```

The above program defines a function named as summation which accepts an array as an argument. The function calculates the sum of all the elements of the array and returns it.

2D Array

2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.

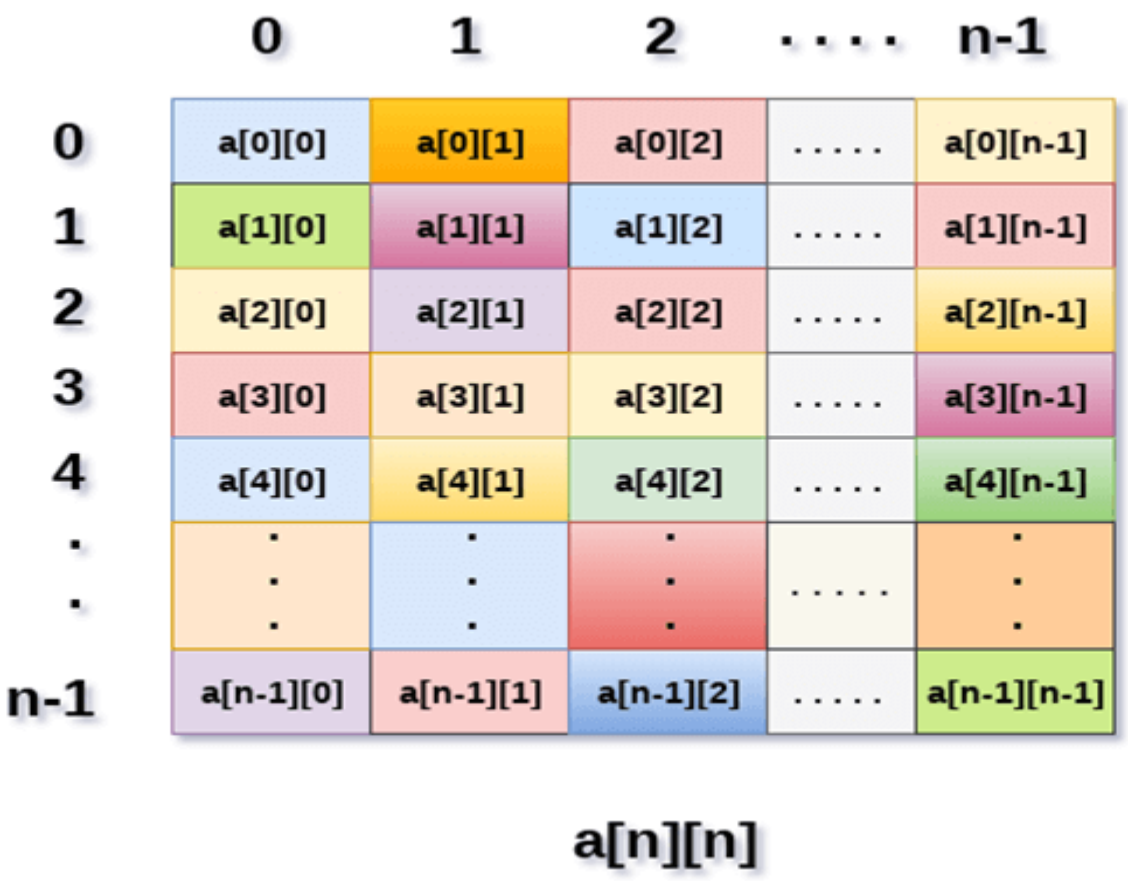
However, 2D arrays are created to implement a relational database look alike data structure. It provides ease of holding bulk of data at once which can be passed to any number of functions wherever required.

How to declare 2D Array

The syntax of declaring two dimensional array is very much similar to that of a one dimensional array, given as follows.

```
1.  int arr[max_rows][max_columns];
```

however, It produces the data structure which looks like following.



Above image shows the two dimensional array, the elements are organized in the form of rows and columns. First element of the first row is represented by a[0][0] where the number shown in the first index is the number of that row while the number shown in the second index is the number of the column.

How do we access data in a 2D array

Due to the fact that the elements of 2D arrays can be random accessed. Similar to one dimensional arrays, we can access the individual cells in a 2D array by using the indices of the cells. There are two indices attached to a particular cell, one is its row number while the other is its column number.

However, we can store the value stored in any particular cell of a 2D array to some variable x by using the following syntax.

```
1. int x = a[i][j];
```

where i and j is the row and column number of the cell respectively.

We can assign each cell of a 2D array to 0 by using the following code:

```
1. for ( int i=0; i<n ;i++)
2. {
3.     for (int j=0; j<n; j++)
4.     {
5.         a[i][j] = 0;
6.     }
7. }
```

Initializing 2D Arrays

We know that, when we declare and initialize one dimensional array in C programming simultaneously, we don't need to specify the size of the array. However this will not work with 2D arrays. We will have to define at least the second dimension of the array.

The syntax to declare and initialize the 2D array is given as follows.

```
1. int arr[2][2] = {0,1,2,3};
```

The number of elements that can be present in a 2D array will always be equal to (**number of rows * number of columns**).

Example : Storing User's data into a 2D array and printing it.

C Example :

```
1. #include <stdio.h>
2. void main ()
3. {
4.     int arr[3][3],i,j;
5.     for (i=0;i<3;i++)
6.     {
7.         for (j=0;j<3;j++)
8.         {
9.             printf("Enter a[%d][%d]: ",i,j);
10.            scanf("%d",&arr[i][j]);
11.        }
12.    }
13.    printf("\n printing the elements ....\n");
14.    for(i=0;i<3;i++)
15.    {
16.        printf("\n");
17.        for (j=0;j<3;j++)
18.        {
19.            printf("%d\t",arr[i][j]);
20.        }
21.    }
```

```
22.}
```

Java Example

```
1.  import java.util.Scanner;
2.  publicclass TwoDArray {
3.  publicstaticvoid main(String[] args) {
4.      int[][] arr = newint[3][3];
5.      Scanner sc = new Scanner(System.in);
6.      for (inti =0;i<3;i++)
7.      {
8.          for(intj=0;j<3;j++)
9.          {
10.             System.out.print("Enter Element");
11.             arr[i][j]=sc.nextInt();
12.             System.out.println();
13.         }
14.     }
15.     System.out.println("Printing Elements...");
16.     for(inti=0;i<3;i++)
17.     {
18.         System.out.println();
19.         for(intj=0;j<3;j++)
```

```
20.      {
21.          System.out.print(arr[i][j]+"\t");
22.      }
23.   }
24.}
25.}
```

C# Example

```
1.  using System;
2.
3.  public class Program
4.  {
5.      public static void Main()
6.      {
7.          int[,] arr = new int[3,3];
8.          for (int i=0;i<3;i++)
9.          {
10.             for (int j=0;j<3;j++)
11.             {
12.                 Console.WriteLine("Enter Element");
13.                 arr[i,j]= Convert.ToInt32(Console.ReadLine());
14.             }
```

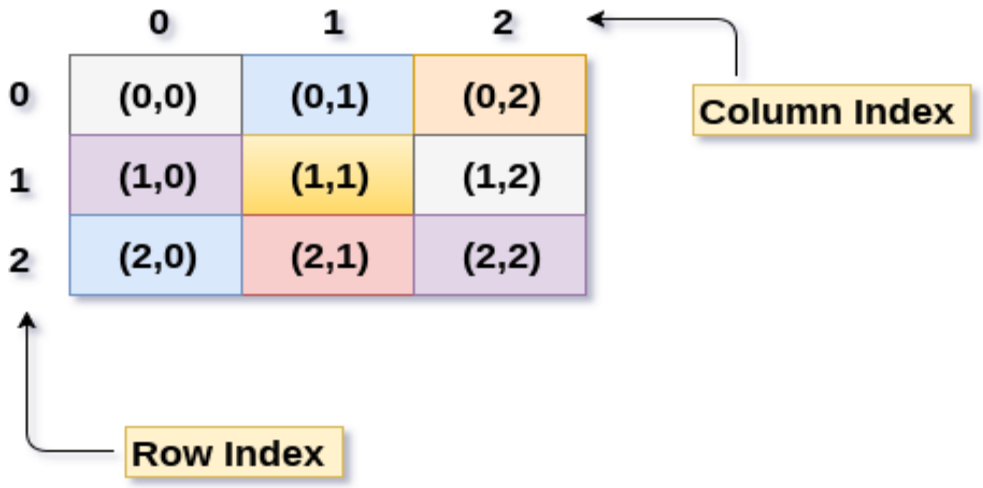
```
15.       }
16.       Console.WriteLine("Printing Elements...");
17.       for (int i=0;i<3;i++)
18.       {
19.          Console.WriteLine();
20.          for (int j=0;j<3;j++)
21.          {
22.             Console.Write(arr[i,j]+" ");
23.          }
24.       }
25.    }
26.}
```

Mapping 2D array to 1D array

When it comes to map a 2 dimensional array, most of us might think that why this mapping is required. However, 2 D arrays exists from the user point of view. 2D arrays are created to implement a relational database table lookalike data structure, in computer memory, the storage technique for 2D array is similar to that of an one dimensional array.

The size of a two dimensional array is equal to the multiplication of number of rows and the number of columns present in the array. We do need to map two dimensional array to the one dimensional array in order to store them in the memory.

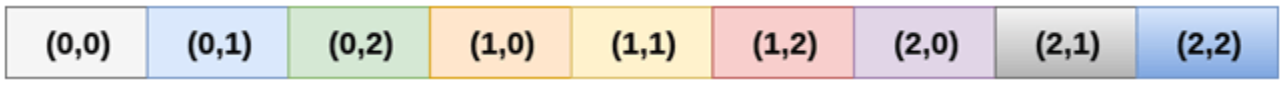
A 3 X 3 two dimensional array is shown in the following image. However, this array needs to be mapped to a one dimensional array in order to store it into the memory.



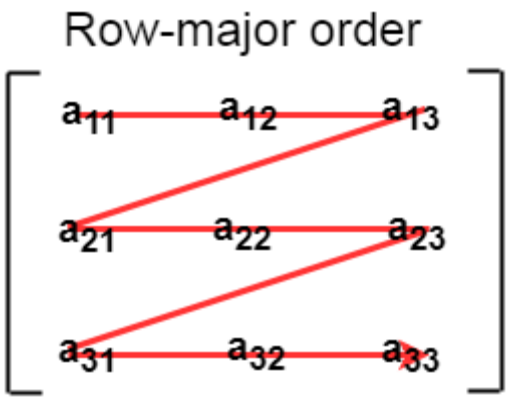
There are two main techniques of storing 2D array elements into memory

1. Row Major ordering

In row major ordering, all the rows of the 2D array are stored into the memory contiguously. Considering the array shown in the above image, its memory allocation according to row major order is shown as follows.

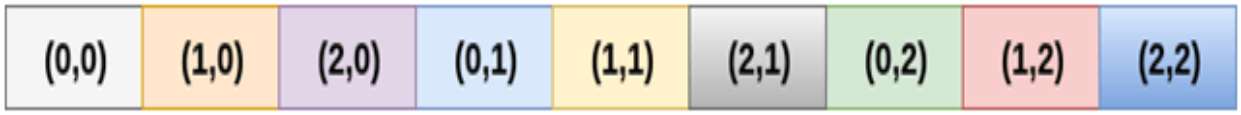


first, the 1st row of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last row.

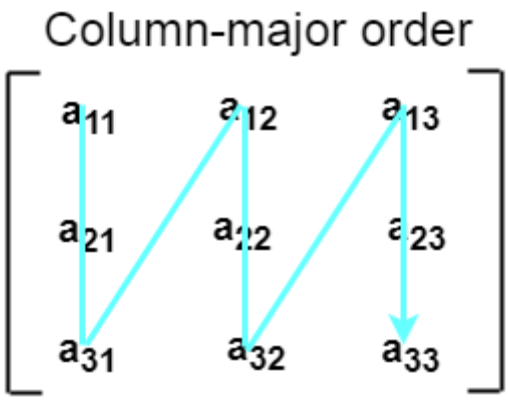


2. Column Major ordering

According to the column major ordering, all the columns of the 2D array are stored into the memory contiguously. The memory allocation of the array which is shown in in the above image is given as follows.



first, the 1st column of the array is stored into the memory completely, then the 2nd row of the array is stored into the memory completely and so on till the last column of the array.



Calculating the Address of the random element of a 2D array

Due to the fact that, there are two different techniques of storing the two dimensional array into the memory, there are two different formulas to calculate the address of a random element of the 2D array.

By Row Major Order

If array is declared by a[m][n] where m is the number of rows while n is the number of columns, then address of an element a[i][j] of the array stored in row major order is calculated as,

1. Address(a[i][j]) = B. A. + (i * n + j) * size

where, B. A. is the base address or the address of the first element of the array a[0][0] .

Example :

a[10...30, 55...75], base address of the array (BA) = 0, size of an element = 4 by tes .

Find the location of a[15][68].

Address(a[15][68]) = 0 +

((15 - 10) x (68 - 55 + 1) + (68 - 55)) x 4

= (5 x 14 + 13) x 4

= 83 x 4

= 332 answer

By Column major order

If array is declared by a[m][n] where m is the number of rows while n is the number of columns, then address of an element a[i][j] of the array stored in row major order is calculated as,

1. Address(a[i][j]) = ((j*m)+i)*Size + BA

where BA is the base address of the array.

Example:

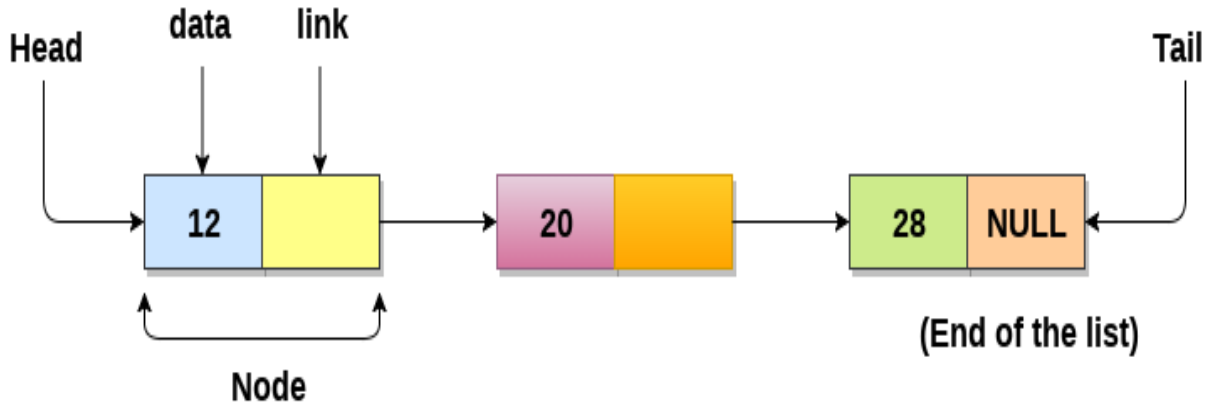
A [-
5 ... +20][20 ... 70], BA = 1020, Size of element = 8 bytes. Find the location of a[0][30].

Address [A[0][30]) = ((30-
20) x 24 + 5) x 8 + 1020 = 245 x 8 + 1020 = 2980 bytes

Linked List

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.

- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.



Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
- list size is limited to the memory size and doesn't need to be declared in advance.
- Empty node can not be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.

Array contains following limitations:

1. The size of array must be known in advance before using it in the program.

2. Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.

3. All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

1. It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.

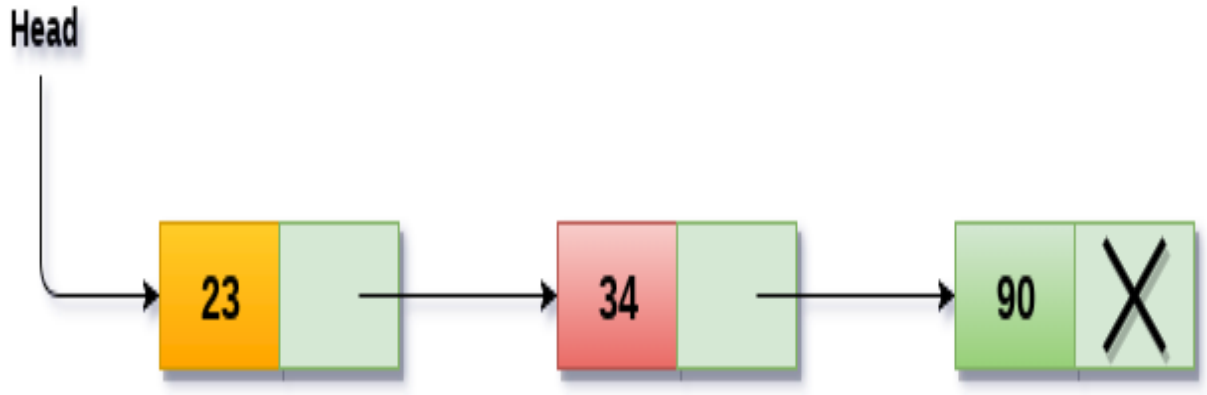
2. Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

Singly linked list or One way chain

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.



In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Complexity

Data Structure	**Time Complexity**								**Space Compleity**
	Average				**Worst**				**Worst**
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	

Singly Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

Node Creation

```
1. struct node
2. {
3.     int data;
4.     struct node *next;
5. };
6. struct node *head, *ptr;
7. ptr = (struct node *)malloc(sizeof(struct node *));
```

Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as

		the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.

2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .

Linked List in C: Menu Driven Program

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. struct node
4. {
5.     int data;
```

```
6.      struct node *next;
7.  };
8.  struct node *head;
9.
10.void beginsert ();
11.void lastinsert ();
12.void randominsert();
13.void begin_delete();
14.void last_delete();
15.void random_delete();
16.void display();
17.void search();
18.void main ()
19.{
20.    int choice =0;
21.    while(choice != 9)
22.    {
23.        printf("\n\n*********Main Menu*********\n");
24.        printf("\nChoose one option from the following list ...\n");
25.        printf("\n===============================================\n")
    ;
```

```
26.     printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random l
        ocation\n4.Delete from Beginning\n
27.     5.Delete from last\n6.Delete node after specified location\n7.Search fo
        r an element\n8.Show\n9.Exit\n");
28.     printf("\nEnter your choice?\n");
29.     scanf("\n%d",&choice);
30.     switch(choice)
31.     {
32.        case 1:
33.        beginsert();
34.        break;
35.        case 2:
36.        lastinsert();
37.        break;
38.        case 3:
39.        randominsert();
40.        break;
41.        case 4:
42.        begin_delete();
43.        break;
44.        case 5:
45.        last_delete();
```

```
46.         break;
47.         case 6:
48.         random_delete();
49.         break;
50.         case 7:
51.         search();
52.         break;
53.         case 8:
54.         display();
55.         break;
56.         case 9:
57.         exit(0);
58.         break;
59.         default:
60.         printf("Please enter valid choice..");
61.       }
62.   }
63.}
64.void beginsert()
65.{
66.   struct node *ptr;
```

```
67.  int item;
68.  ptr = (struct node *) malloc(sizeof(struct node *));
69.  if(ptr == NULL)
70.  {
71.      printf("\nOVERFLOW");
72.  }
73.  else
74.  {
75.      printf("\nEnter value\n");
76.      scanf("%d",&item);
77.      ptr->data = item;
78.      ptr->next = head;
79.      head = ptr;
80.      printf("\nNode inserted");
81.  }
82.
83.}
84.void lastinsert()
85.{
86.  struct node *ptr,*temp;
87.  int item;
```

```
88.  ptr = (struct node*)malloc(sizeof(struct node));
89.  if(ptr == NULL)
90.  {
91.      printf("\nOVERFLOW");
92.  }
93.  else
94.  {
95.      printf("\nEnter value?\n");
96.      scanf("%d",&item);
97.      ptr->data = item;
98.      if(head == NULL)
99.      {
100.                 ptr -> next = NULL;
101.                 head = ptr;
102.                 printf("\nNode inserted");
103.             }
104.             else
105.             {
106.                 temp = head;
107.                 while (temp -> next != NULL)
108.                 {
```

```
109.                 temp = temp -> next;
110.             }
111.             temp->next = ptr;
112.             ptr->next = NULL;
113.             printf("\nNode inserted");
114.
115.         }
116.     }
117. }
118. void randominsert()
119. {
120.     int i,loc,item;
121.     struct node *ptr, *temp;
122.     ptr = (struct node *) malloc (sizeof(struct node));
123.     if(ptr == NULL)
124.     {
125.         printf("\nOVERFLOW");
126.     }
127.     else
128.     {
129.         printf("\nEnter element value");
```

```
130.            scanf("%d",&item);
131.            ptr->data = item;
132.            printf("\nEnter the location after which you want to insert ");
133.            scanf("\n%d",&loc);
134.            temp=head;
135.            for(i=0;i<loc;i++)
136.            {
137.               temp = temp->next;
138.               if(temp == NULL)
139.               {
140.                  printf("\ncan't insert\n");
141.                  return;
142.               }
143.
144.            }
145.            ptr ->next = temp ->next;
146.            temp ->next = ptr;
147.            printf("\nNode inserted");
148.         }
149.      }
150.      void begin_delete()
```

```
151.    {
152.        struct node *ptr;
153.        if(head == NULL)
154.        {
155.            printf("\nList is empty\n");
156.        }
157.        else
158.        {
159.            ptr = head;
160.            head = ptr->next;
161.            free(ptr);
162.            printf("\nNode deleted from the begining ...\n");
163.        }
164.    }
165.    void last_delete()
166.    {
167.        struct node *ptr,*ptr1;
168.        if(head == NULL)
169.        {
170.            printf("\nlist is empty");
171.        }
```

```
172.        else if(head -> next == NULL)
173.        {
174.           head = NULL;
175.           free(head);
176.           printf("\nOnly node of the list deleted ...\n");
177.        }
178.
179.        else
180.        {
181.           ptr = head;
182.           while(ptr->next != NULL)
183.           {
184.              ptr1 = ptr;
185.              ptr = ptr ->next;
186.           }
187.           ptr1->next = NULL;
188.           free(ptr);
189.           printf("\nDeleted Node from the last ...\n");
190.        }
191.     }
192.     void random_delete()
```

```
193.    {
194.        struct node *ptr,*ptr1;
195.        int loc,i;
196.        printf("\n Enter the location of the node after which you want to p
        erform deletion \n");
197.        scanf("%d",&loc);
198.        ptr=head;
199.        for(i=0;i<loc;i++)
200.        {
201.          ptr1 = ptr;
202.          ptr = ptr->next;
203.
204.          if(ptr == NULL)
205.          {
206.             printf("\nCan't delete");
207.             return;
208.          }
209.        }
210.        ptr1 ->next = ptr ->next;
211.        free(ptr);
212.        printf("\nDeleted node %d ",loc+1);
```

```
213.    }
214.    void search()
215.    {
216.       struct node *ptr;
217.       int item,i=0,flag;
218.       ptr = head;
219.       if(ptr == NULL)
220.       {
221.          printf("\nEmpty List\n");
222.       }
223.       else
224.       {
225.          printf("\nEnter item which you want to search?\n");
226.          scanf("%d",&item);
227.          while (ptr!=NULL)
228.          {
229.             if(ptr->data == item)
230.             {
231.                printf("item found at location %d ",i+1);
232.                flag=0;
233.             }
```

```
234.                else
235.                {
236.                   flag=1;
237.                }
238.                i++;
239.                ptr = ptr -> next;
240.             }
241.             if(flag==1)
242.             {
243.                printf("Item not found\n");
244.             }
245.          }
246.
247.       }
248.
249.       void display()
250.       {
251.          struct node *ptr;
252.          ptr = head;
253.          if(ptr == NULL)
254.          {
```

```
255.            printf("Nothing to print");
256.        }
257.        else
258.        {
259.            printf("\nprinting values . . . . .\n");
260.            while (ptr!=NULL)
261.            {
262.                printf("\n%d",ptr->data);
263.                ptr = ptr -> next;
264.            }
265.        }
266.    }
267.
```

Output:

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

1

Enter value

1

Node inserted

*********Main Menu*********

Choose one option from the following list ..

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

2

Enter value?

2

Node inserted

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

3

Enter element value1

Enter the location after which you want to insert 1

Node inserted

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

8

printing values . . . . .

1

2

1

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

2

Enter value?

123

Node inserted

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

1

Enter value

1234

Node inserted

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

4

Node deleted from the begining ...

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

5

Deleted Node from the last ...

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

6

Enter the location of the node after which you want to perform deletion

1

Deleted node 2

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

8

printing values . . . . .

1

1

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

7

Enter item which you want to search?

1

item found at location 1

item found at location 2

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

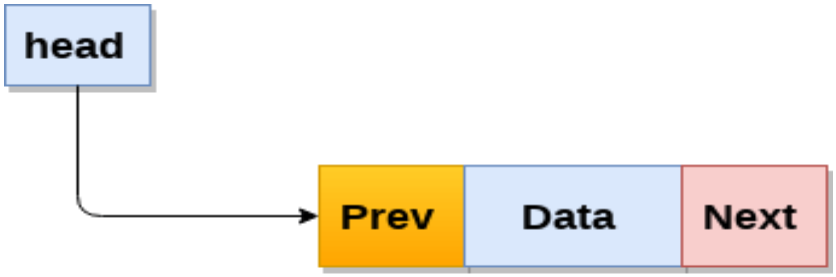
9.Exit

Enter your choice?

9

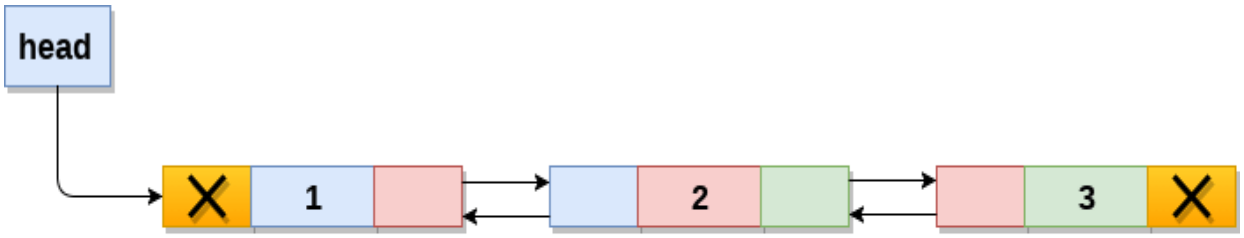
Doubly linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



Node

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



Doubly Linked List

In C, structure of a node in doubly linked list can be given as :

```
1. struct node
2. {
3.     struct node *prev;
4.     int data;
5.     struct node *next;
6. }
```

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

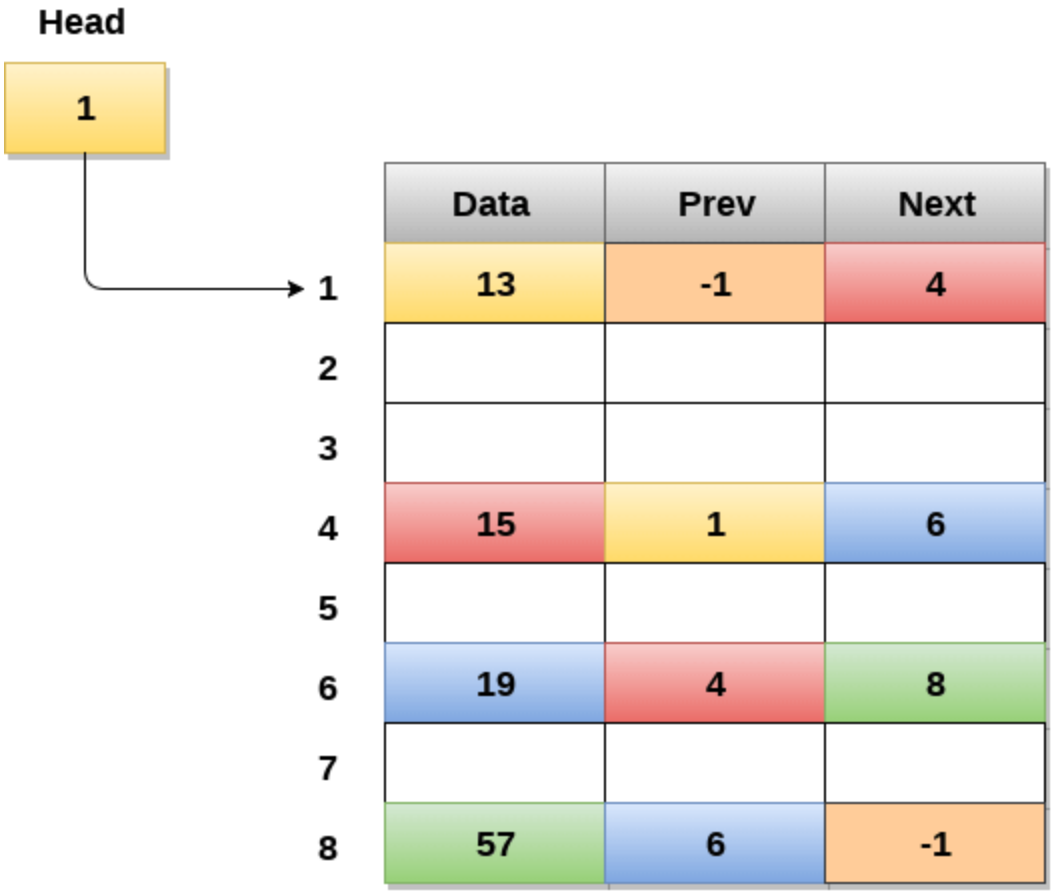
In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.



Memory Representation of a Doubly linked list

Operations on doubly linked list

Node Creation

```
1. struct node
2. {
3.     struct node *prev;
```

```
4.  int data;
5.  struct node *next;
6. };
7. struct node *head;
```

All the remaining operations regarding doubly linked list are described in the following table.

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.

6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Menu Driven Program in C to implement all the operations of doubly linked list

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. struct node
4. {
5.     struct node *prev;
6.     struct node *next;
7.     int data;
8. };
9. struct node *head;
10.void insertion_beginning();
11.void insertion_last();
```

```
12.void insertion_specified();
13.void deletion_beginning();
14.void deletion_last();
15.void deletion_specified();
16.void display();
17.void search();
18.void main ()
19.{
20.int choice =0;
21.  while(choice != 9)
22.  {
23.      printf("\n*********Main Menu*********\n");
24.      printf("\nChoose one option from the following list ...\n");
25.      printf("\n===============================================\n")
    ;
26.      printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random l
    ocation\n4.Delete from Beginning\n
27.      5.Delete from last\n6.Delete the node after the given data\n7.Search\
    n8.Show\n9.Exit\n");
28.      printf("\nEnter your choice?\n");
29.      scanf("\n%d",&choice);
30.      switch(choice)
```

```
31.     {
32.         case 1:
33.         insertion_beginning();
34.         break;
35.         case 2:
36.             insertion_last();
37.         break;
38.         case 3:
39.         insertion_specified();
40.         break;
41.         case 4:
42.         deletion_beginning();
43.         break;
44.         case 5:
45.         deletion_last();
46.         break;
47.         case 6:
48.         deletion_specified();
49.         break;
50.         case 7:
51.         search();
```

```
52.          break;
53.          case 8:
54.          display();
55.          break;
56.          case 9:
57.          exit(0);
58.          break;
59.          default:
60.          printf("Please enter valid choice..");
61.       }
62.   }
63.}
64.void insertion_beginning()
65.{
66.   struct node *ptr;
67.   int item;
68.   ptr = (struct node *)malloc(sizeof(struct node));
69.   if(ptr == NULL)
70.   {
71.      printf("\nOVERFLOW");
72.   }
```

```
73. else
74. {
75.  printf("\nEnter Item value");
76.  scanf("%d",&item);
77.
78. if(head==NULL)
79. {
80.     ptr->next = NULL;
81.     ptr->prev=NULL;
82.     ptr->data=item;
83.     head=ptr;
84. }
85. else
86. {
87.     ptr->data=item;
88.     ptr->prev=NULL;
89.     ptr->next = head;
90.     head->prev=ptr;
91.     head=ptr;
92. }
93. printf("\nNode inserted\n");
```

```
94.}
95.
96.}
97.void insertion_last()
98.{
99.   struct node *ptr,*temp;
100.        int item;
101.        ptr = (struct node *) malloc(sizeof(struct node));
102.        if(ptr == NULL)
103.        {
104.           printf("\nOVERFLOW");
105.        }
106.        else
107.        {
108.           printf("\nEnter value");
109.           scanf("%d",&item);
110.            ptr->data=item;
111.           if(head == NULL)
112.           {
113.              ptr->next = NULL;
114.              ptr->prev = NULL;
```

```
115.              head = ptr;
116.            }
117.            else
118.            {
119.              temp = head;
120.              while(temp->next!=NULL)
121.              {
122.                 temp = temp->next;
123.              }
124.              temp->next = ptr;
125.              ptr ->prev=temp;
126.              ptr->next = NULL;
127.              }
128.
129.            }
130.          printf("\nnode inserted\n");
131.         }
132.       void insertion_specified()
133.       {
134.          struct node *ptr,*temp;
135.          int item,loc,i;
```

```
136.        ptr = (struct node *)malloc(sizeof(struct node));
137.        if(ptr == NULL)
138.        {
139.           printf("\n OVERFLOW");
140.        }
141.        else
142.        {
143.           temp=head;
144.           printf("Enter the location");
145.           scanf("%d",&loc);
146.           for(i=0;i<loc;i++)
147.           {
148.              temp = temp->next;
149.              if(temp == NULL)
150.              {
151.                 printf("\n There are less than %d elements", loc);
152.                 return;
153.              }
154.           }
155.           printf("Enter value");
156.           scanf("%d",&item);
```

```
157.            ptr->data = item;
158.            ptr->next = temp->next;
159.            ptr -> prev = temp;
160.            temp->next = ptr;
161.            temp->next->prev=ptr;
162.            printf("\nnode inserted\n");
163.          }
164.       }
165.       void deletion_beginning()
166.       {
167.          struct node *ptr;
168.          if(head == NULL)
169.          {
170.             printf("\n UNDERFLOW");
171.          }
172.          else if(head->next == NULL)
173.          {
174.             head = NULL;
175.             free(head);
176.             printf("\nnode deleted\n");
177.          }
```

```
178.        else
179.        {
180.            ptr = head;
181.            head = head -> next;
182.            head -> prev = NULL;
183.            free(ptr);
184.            printf("\nnode deleted\n");
185.        }
186.
187.    }
188.    void deletion_last()
189.    {
190.        struct node *ptr;
191.        if(head == NULL)
192.        {
193.            printf("\n UNDERFLOW");
194.        }
195.        else if(head->next == NULL)
196.        {
197.            head = NULL;
198.            free(head);
```

```
199.            printf("\nnode deleted\n");
200.        }
201.        else
202.        {
203.            ptr = head;
204.            if(ptr->next != NULL)
205.            {
206.                ptr = ptr -> next;
207.            }
208.            ptr -> prev -> next = NULL;
209.            free(ptr);
210.            printf("\nnode deleted\n");
211.        }
212.    }
213.    void deletion_specified()
214.    {
215.        struct node *ptr, *temp;
216.        int val;
217.        printf("\n Enter the data after which the node is to be deleted : ");
218.        scanf("%d", &val);
219.        ptr = head;
```

```
220.        while(ptr -> data != val)
221.        ptr = ptr -> next;
222.        if(ptr -> next == NULL)
223.        {
224.           printf("\nCan't delete\n");
225.        }
226.        else if(ptr -> next -> next == NULL)
227.        {
228.           ptr ->next = NULL;
229.        }
230.        else
231.        {
232.           temp = ptr -> next;
233.           ptr -> next = temp -> next;
234.           temp -> next -> prev = ptr;
235.           free(temp);
236.           printf("\nnode deleted\n");
237.        }
238.     }
239.     void display()
240.     {
```

```
241.        struct node *ptr;
242.        printf("\n printing values...\n");
243.        ptr = head;
244.        while(ptr != NULL)
245.        {
246.           printf("%d\n",ptr->data);
247.           ptr=ptr->next;
248.        }
249.     }
250.     void search()
251.     {
252.        struct node *ptr;
253.        int item,i=0,flag;
254.        ptr = head;
255.        if(ptr == NULL)
256.        {
257.           printf("\nEmpty List\n");
258.        }
259.        else
260.        {
261.           printf("\nEnter item which you want to search?\n");
```

```
262.            scanf("%d",&item);
263.            while (ptr!=NULL)
264.            {
265.               if(ptr->data == item)
266.               {
267.                  printf("\nitem found at location %d ",i+1);
268.                  flag=0;
269.                  break;
270.               }
271.               else
272.               {
273.                  flag=1;
274.               }
275.               i++;
276.               ptr = ptr -> next;
277.            }
278.            if(flag==1)
279.            {
280.               printf("\nItem not found\n");
281.            }
282.         }
```

283.

284. }

Output

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

8

printing values...

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

1

Enter Item value12

Node inserted

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

1

Enter Item value123

Node inserted

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

1

Enter Item value1234

Node inserted

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

8

printing values...

1234

123

12

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

2

Enter value89

node inserted

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

3

Enter the location1

Enter value12345

node inserted

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

8

printing values...

1234

123

12345

12

89

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

4

node deleted

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

5

node deleted

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

8

printing values...

123

12345

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

6

Enter the data after which the node is to be deleted : 123

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

8

printing values...

123

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

7

Enter item which you want to search?

123

item found at location 1

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

6

Enter the data after which the node is to be deleted : 123

Can't delete

*********Main Menu*********

Choose one option from the following list ...

==========

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

9

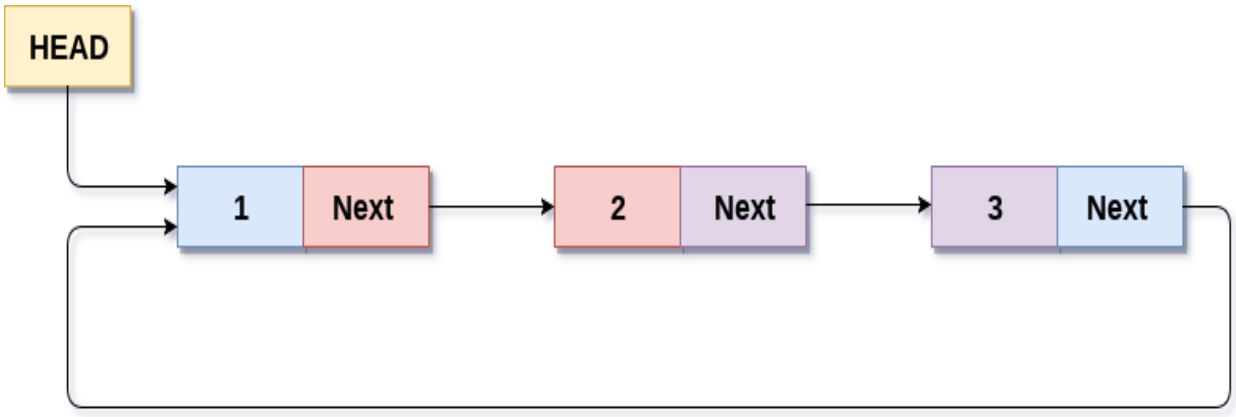
Exited..

Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



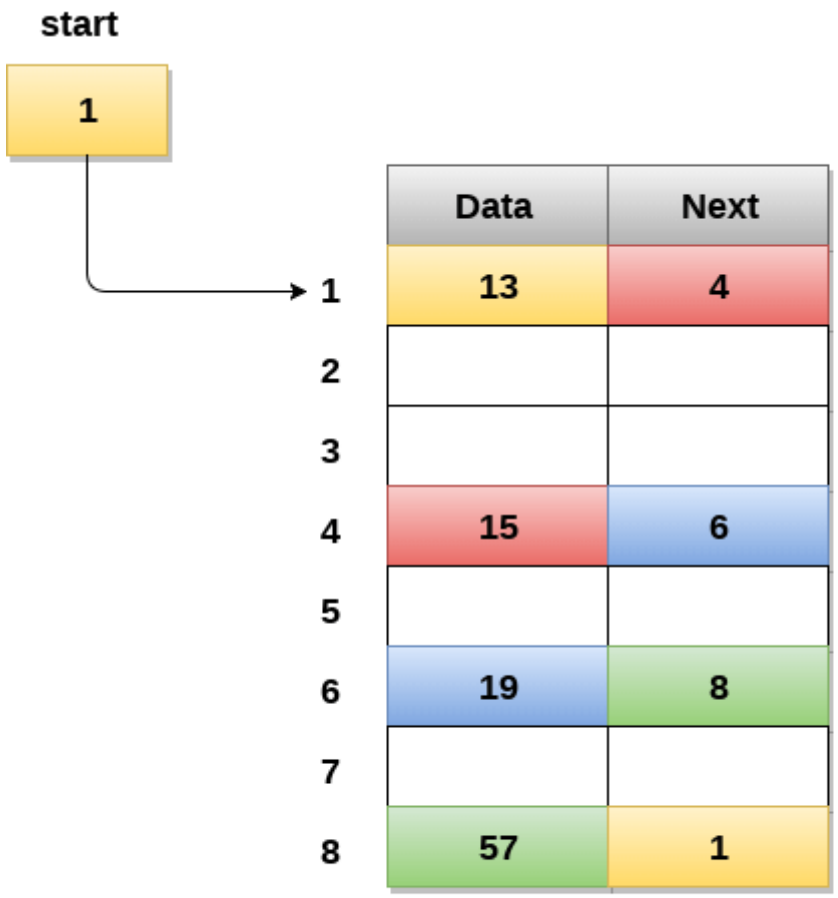
Circular Singly Linked List

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Memory Representation of circular linked list:

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.

However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.



Memory Representation of a circular linked list

We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can find out the number of iterations which need to be performed while traversing the list.

Operations on Circular Singly linked list:

Insertion

SN	Operation	Description
1	Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2	Insertion at the end	Adding a node into circular singly linked list at the end.

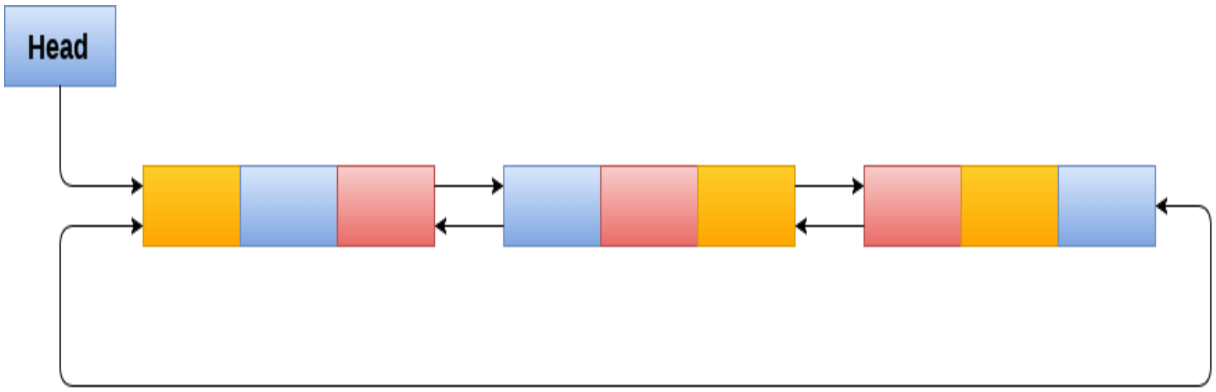
Deletion & Traversing

SN	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list at the beginning.
2	Deletion at the end	Removing the node from circular singly linked list at the end.
3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

Circular Doubly Linked List

Circular doubly linked list is a more complexed type of data structure in which a node contain pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the node. The last node of the list contains the address of the first node of the list. The first node of the list also contain address of the last node in its previous pointer.

A circular doubly linked list is shown in the following figure.



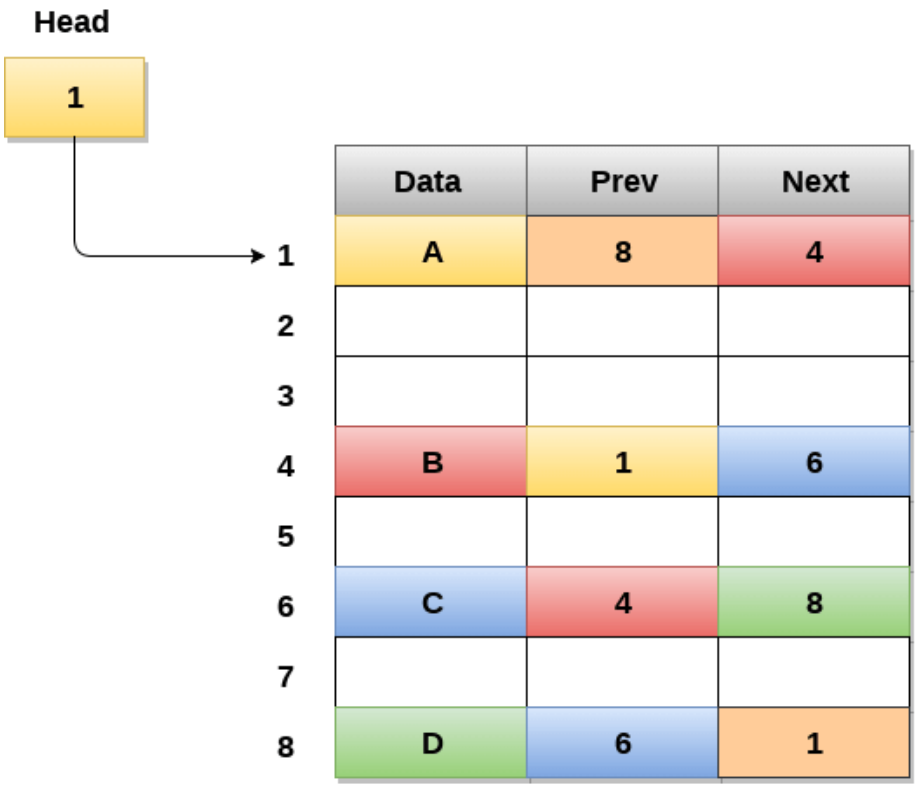
Circular Doubly Linked List

Due to the fact that a circular doubly linked list contains three parts in its structure therefore, it demands more space per node and more expensive basic operations. However, a circular doubly linked list provides easy manipulation of the pointers and the searching becomes twice as efficient.

Memory Management of Circular Doubly linked list

The following figure shows the way in which the memory is allocated for a circular doubly linked list. The variable head contains the address of the first element of the list i.e. 1 hence the starting node of the list contains data A is stored at address 1. Since, each node of the list is supposed to have three parts therefore, the starting node of the list contains address of the last node i.e. 8 and the next node i.e. 4. The last node of the list that is stored at address 8 and containing data as 6, contains address of the first node of the list as shown in the image i.e. 1. In

circular doubly linked list, the last node is identified by the address of the first node which is stored in the next part of the last node therefore the node which contains the address of the first node, is actually the last node of the list.



Memory Representation of a Circular Doubly linked list

Operations on circular doubly linked list :

There are various operations which can be performed on circular doubly linked list. The node structure of a circular doubly linked list is similar to doubly linked list. However, the operations on circular doubly linked list is described in the following table.

SN	Operation	Description
1	Insertion at beginning	Adding a node in circular doubly linked list at the beginning.
2	Insertion at end	Adding a node in circular doubly linked list at the end.
3	Deletion at beginning	Removing a node in circular doubly linked list from beginning.
4	Deletion at end	Removing a node in circular doubly linked list at the end.

Traversing and searching in circular doubly linked list is similar to that in the circular singly linked list.

Stack

1. Stack is an ordered list in which, insertion and deletion can be performed only at one end that is called **top**.
2. Stack is a recursive data structure having pointer to its top element.
3. Stacks are sometimes called as Last-In-First-Out (LIFO) lists i.e. the element which is inserted first in the stack, will be deleted last from the stack.

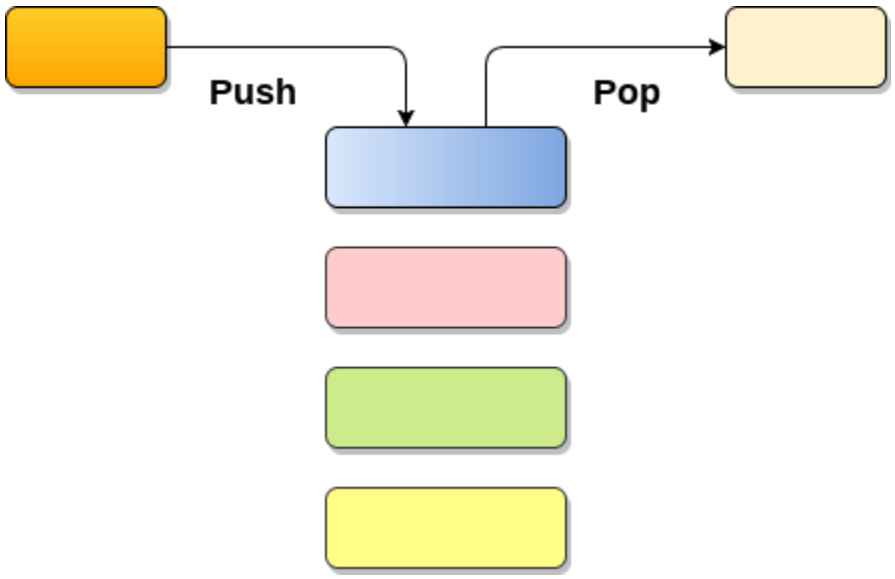
Applications of Stack

1. Recursion
2. Expression evaluations and conversions
3. Parsing
4. Browsers

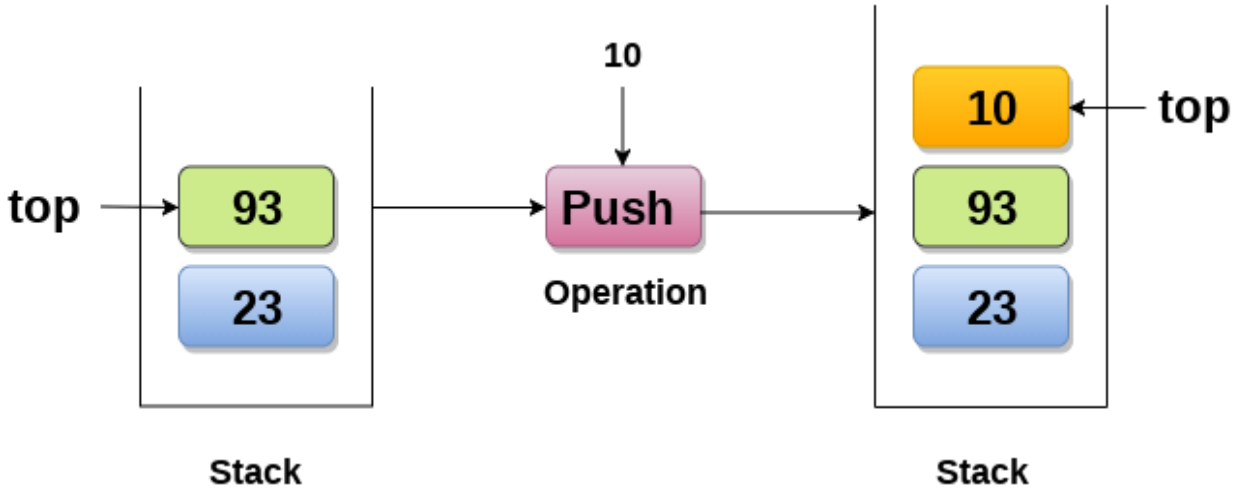
5. Editors
6. Tree Traversals

Operations on Stack

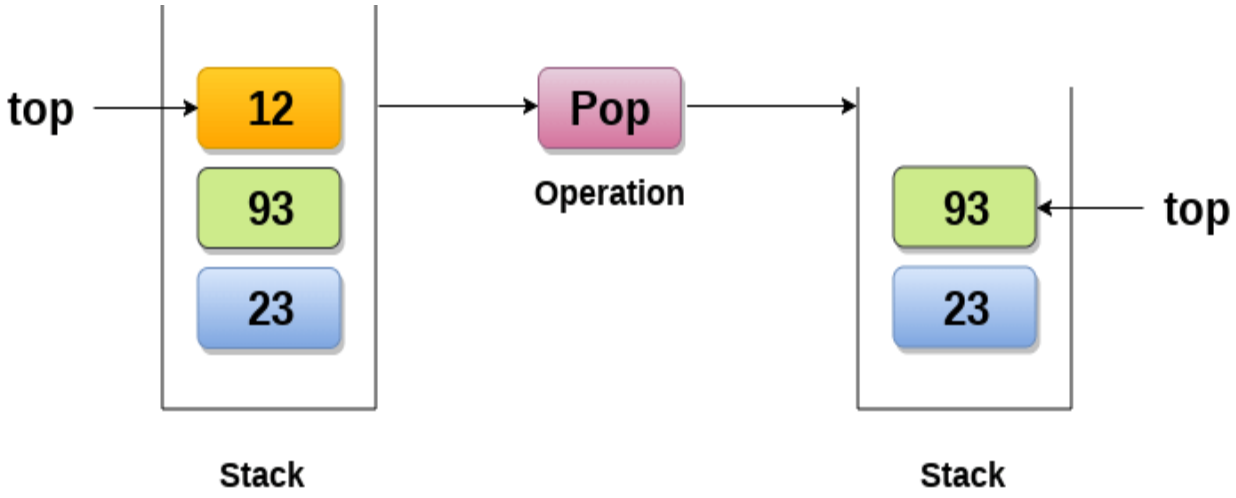
There are various operations which can be performed on stack.



1. Push : Adding an element onto the stack



2. Pop : Removing an element from the stack

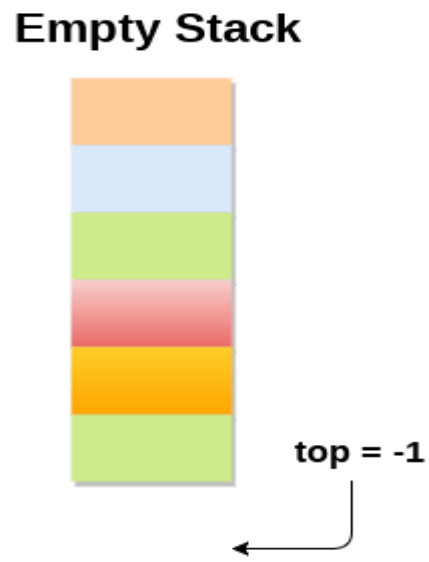


3. Peek : Look all the elements of stack without removing them.

How the stack grows?

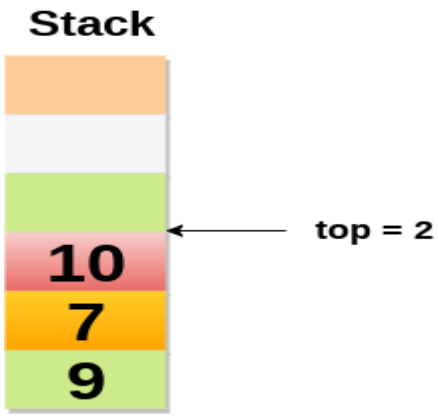
Scenario 1 : Stack is empty

The stack is called empty if it doesn't contain any element inside it. At this stage, the value of variable top is -1.



Scenario 2 : Stack is not empty

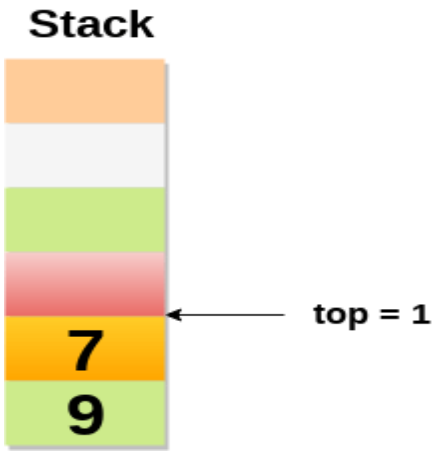
Value of top will get increased by 1 every time when we add any element to the stack. In the following stack, After adding first element, top = 2.



Scenario 3 : Deletion of an element

Value of top will get decreased by 1 whenever an element is deleted from the stack.

In the following stack, after deleting 10 from the stack, top = 1.



Top and its value :

Top position	Status of stack
-1	Empty
0	Only one element in the stack
N-1	Stack is full
N	Overflow

Array implementation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refere to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm:

1. begin
2. **if** top = n then stack full
3. top = top + 1
4. stack (top) : = item;
5. end

Time Complexity : o(1)

implementation of push algorithm in C language

```
1. void push (int val,int n) //n is size of the stack
2. {
3.     if (top == n )
```

```
4.  printf("\n Overflow");
5.  else
6.  {
7.  top = top +1;
8.  stack[top] = val;
9.  }
10.}
```

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack. The top most element of the stack is stored in an another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.

The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm :

```
1. begin
2.  if top = 0 then stack empty;
3.  item := stack(top);
4.  top = top - 1;
5. end;
```

Time Complexity : o(1)

Implementation of POP algorithm using C language

```
1. int pop ()
2. {
3.     if(top == -1)
4.     {
5.         printf("Underflow");
6.         return 0;
7.     }
8.     else
9.     {
10.        return stack[top - - ];
11.    }
12.}
```

Visiting each element of the stack (Peek operation)

Peek operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an already empty stack.

Algorithm :

PEEK (STACK, TOP)

```
1. Begin
2.     if top = -1 then stack empty
3.     item = stack[top]
4.     return item
```

5. End

Time complexity: o(n)

Implementation of Peek algorithm in C language

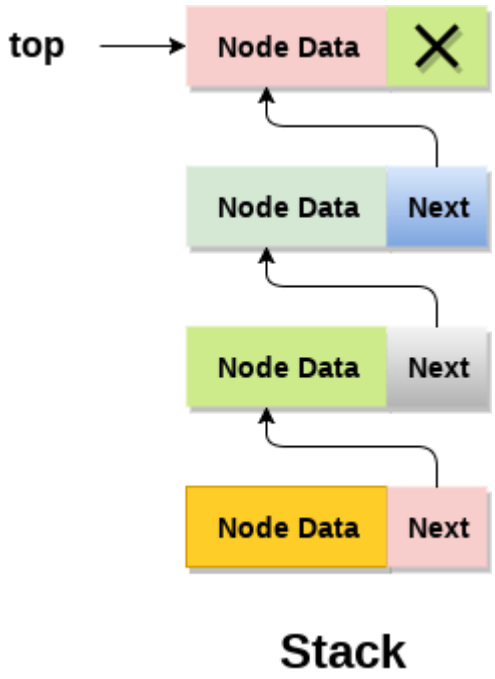
```
1. int peek()
2. {
3.     if (top == -1)
4.     {
5.         printf("Underflow");
6.         return 0;
7.     }
8.     else
9.     {
10.        return stack [top];
11.    }
12.}
```

Linked list implementation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in

the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



Stack

The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

Adding a node to the stack (Push operation)

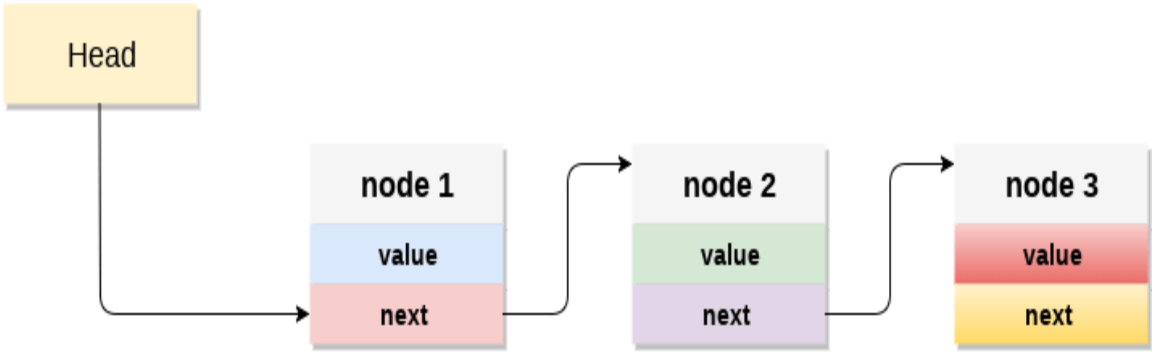
Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

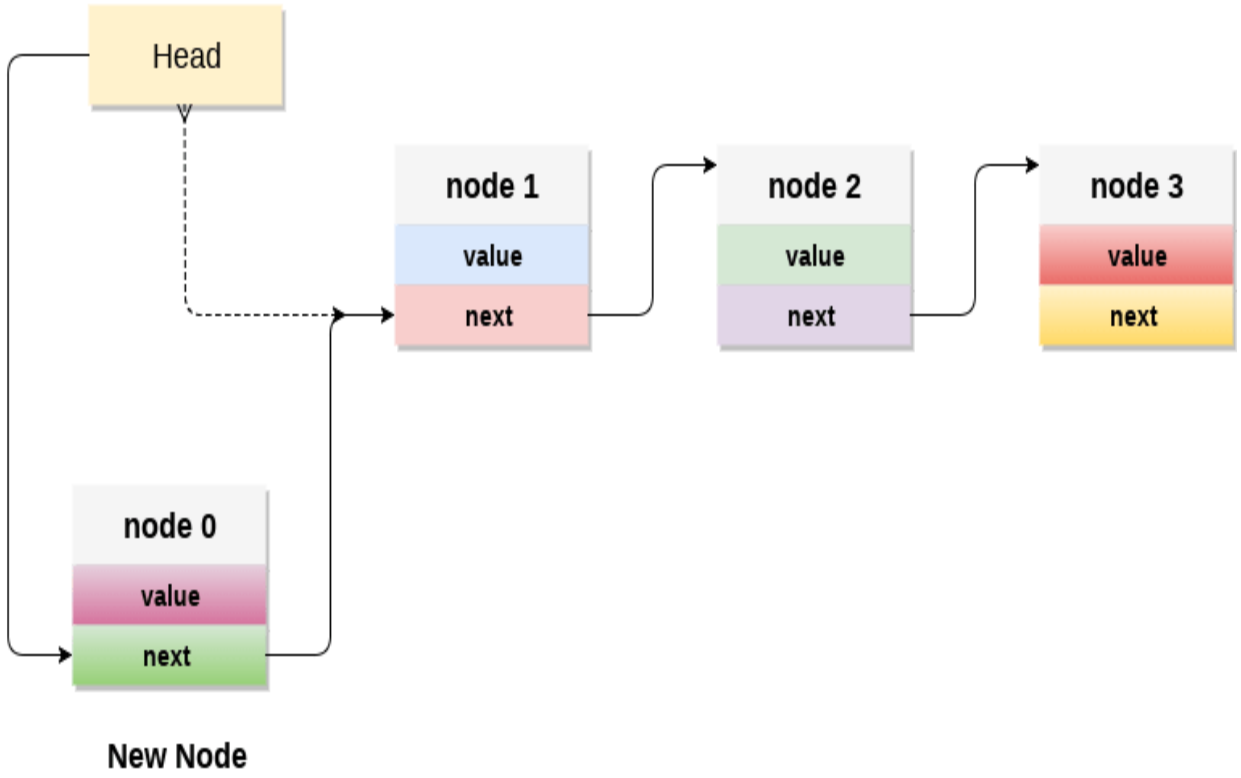
1. Create a node first and allocate memory to it.

2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.

3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Time Complexity : o(1)





C implementation :

```
1. void push ()
2. {
```

```
3.  int val;
4.  struct node *ptr =(struct node*)malloc(sizeof(struct node));
5.  if(ptr == NULL)
6.  {
7.     printf("not able to push the element");
8.  }
9.  else
10. {
11.    printf("Enter the value");
12.    scanf("%d",&val);
13.    if(head==NULL)
14.    {
15.       ptr->val = val;
16.       ptr -> next = NULL;
17.       head=ptr;
18.    }
19.    else
20.    {
21.       ptr->val = val;
22.       ptr->next = head;
23.       head=ptr;
```

```
24.
25.       }
26.       printf("Item pushed");
27.
28.    }
29.}
```

Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

30. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
31. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity : o(n)

C implementation

```
1. void pop()
2. {
3.     int item;
4.     struct node *ptr;
```

```
5.  if (head == NULL)
6.  {
7.      printf("Underflow");
8.  }
9.  else
10. {
11.     item = head->val;
12.     ptr = head;
13.     head = head->next;
14.     free(ptr);
15.     printf("Item popped");
16.
17. }
18.}
```

Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

19. Copy the head pointer into a temporary pointer.
20. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Time Complexity : o(n)

C Implementation

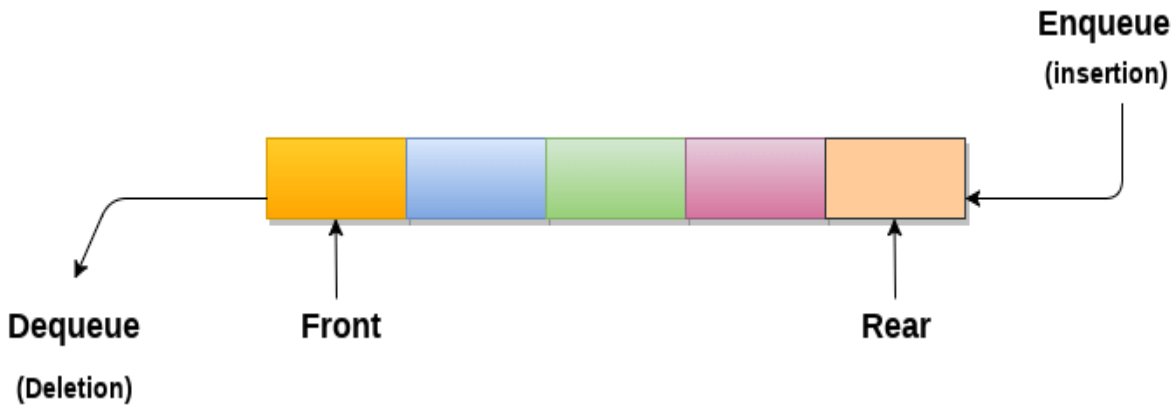
```
1.  void display()
2.  {
3.      int i;
4.      struct node *ptr;
5.      ptr=head;
6.      if(ptr == NULL)
7.      {
8.          printf("Stack is empty\n");
9.      }
10.     else
11.     {
12.         printf("Printing Stack elements \n");
13.         while(ptr!=NULL)
14.         {
15.             printf("%d\n",ptr->val);
16.             ptr = ptr->next;
17.         }
18.     }
19.}
```

Queue

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

2. Queue is referred to be as First In First Out list.

3. For example, people waiting in line for a rail ticket form a queue.



Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.

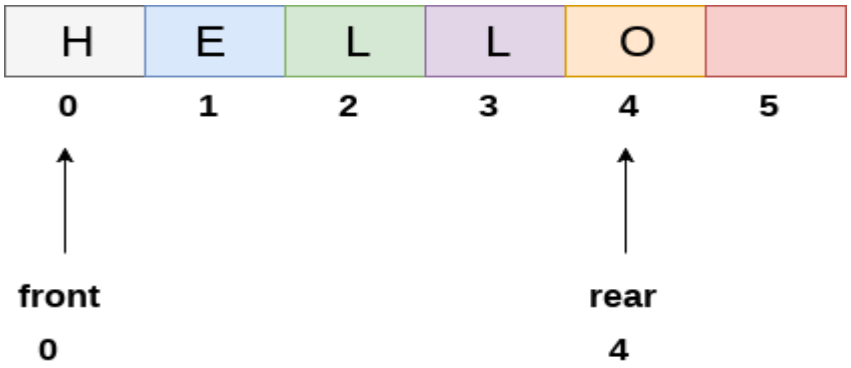
Queues are used in operating systems for handling interrupts.

Complexity

Data Structure	**Time Complexity**								**Space Complexity**
	Average				**Worst**				**Worst**
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Queue	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

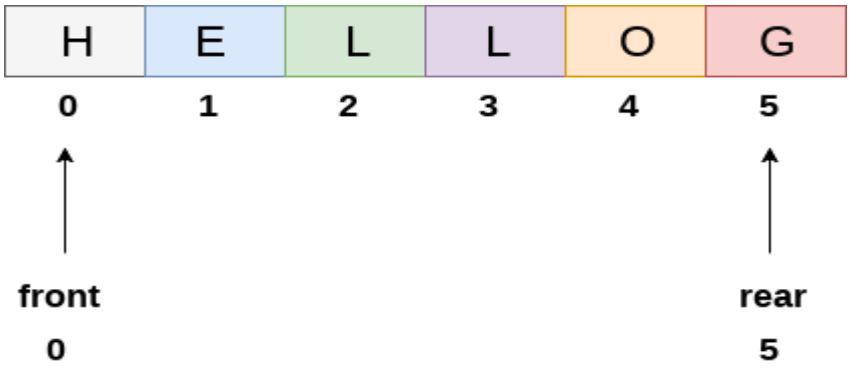
Array representation of Queue

We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



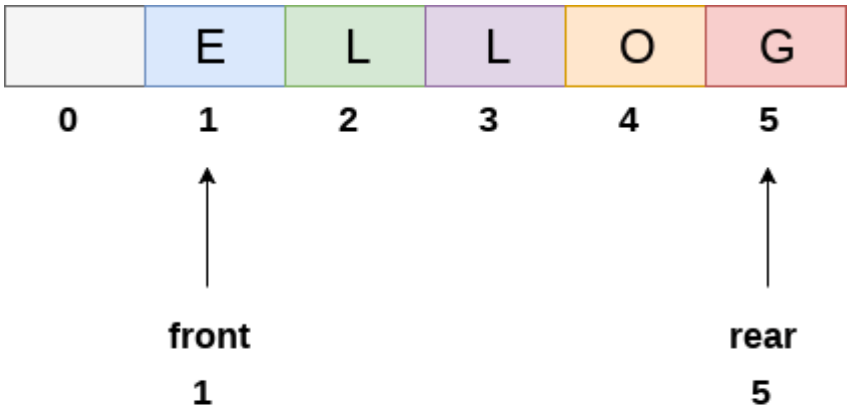
Queue

The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

- **Step 1:** IF REAR = MAX - 1
  Write OVERFLOW
  Go to step
  [END OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE

SET REAR = REAR + 1
[END OF IF]

- **Step 3:** Set QUEUE[REAR] = NUM
- **Step 4:** EXIT

C Function

```
1.  void insert (int queue[], int max, int front, int rear, int item)
2.  {
3.      if (rear + 1 == max)
4.      {
5.          printf("overflow");
6.      }
7.      else
8.      {
9.          if(front == -1 && rear == -1)
10.         {
11.             front = 0;
12.             rear = 0;
13.         }
14.         else
15.         {
16.             rear = rear + 1;
```

```
17.      }
18.      queue[rear]=item;
19.   }
20.}
```

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

- **Step 1:** IF FRONT = -1 or FRONT > REAR
  Write UNDERFLOW
  ELSE
  SET VAL = QUEUE[FRONT]
  SET FRONT = FRONT + 1
  [END OF IF]
- **Step 2:** EXIT

C Function

```
1. int delete (int queue[], int max, int front, int rear)
2. {
3.    int y;
4.    if (front == -1 || front > rear)
5.
```

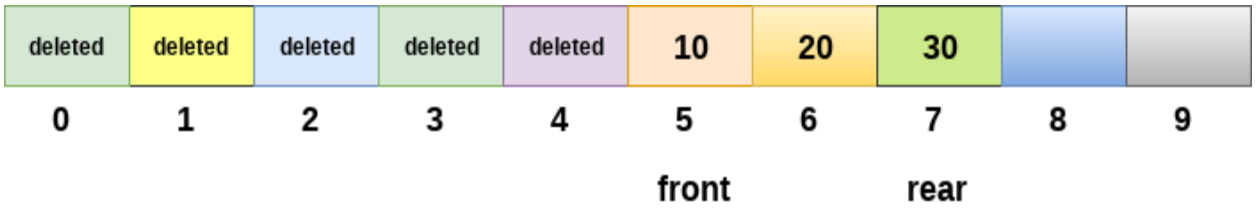
```
6.    {
7.        printf("underflow");
8.    }
9.    else
10.   {
11.       y = queue[front];
12.       if(front == rear)
13.       {
14.           front = rear = -1;
15.           else
16.           front = front + 1;
17.
18.       }
19.       return y;
20.   }
21.}
```

Drawback of array implementation

Although, the technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.

- **Memory wastage :** The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because

the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.



limitation of array representation of queue

The above figure shows how the memory space is wasted in the array representation of queue. In the above figure, a queue of size 10 having 3 elements, is shown. The value of the front variable is 5, therefore, we can not reinsert the values in the place of already deleted element before the position of front. That much space of the array is wasted and can not be used in the future (for this queue).

- **Deciding the array size**

On of the most common problem with array implementation is the size of the array which requires to be declared in advance. Due to the fact that, the queue can be extended at runtime depending upon the problem, the extension in the array size is a time taking process and almost impossible to be performed at runtime since a lot of reallocations take place. Due to this reason, we can declare the array large enough so that we can store queue elements as enough as possible but the main problem with this declaration is that, most of the array slots (nearly half) can never be reused. It will again lead to memory wastage.

Linked List implementation of Queue

Due to the drawbacks discussed in the previous section of this tutorial, the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

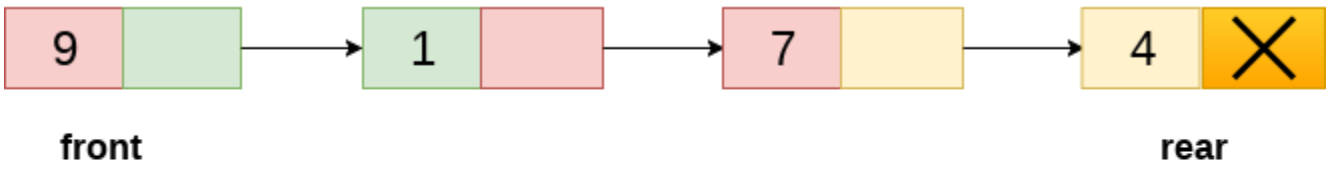
The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

Insert operation

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

Firstly, allocate the memory for the new node ptr by using the following statement.

1. Ptr = (struct node *) malloc (sizeof(struct node));

There can be the two scenario of inserting this new node ptr into the linked queue.

In the first scenario, we insert element into an empty queue. In this case, the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

```
1. ptr -> data = item;
2.      if(front == NULL)
3.      {
4.          front = ptr;
5.          rear = ptr;
6.          front -> next = NULL;
7.          rear -> next = NULL;
8.      }
```

In the second case, the queue contains more than one element. The condition front = NULL becomes false. In this scenario, we need to update the end pointer

rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also need to make the next pointer of rear point to NULL.

1. rear -> next = ptr;
2. rear = ptr;
3. rear->next = NULL;

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

Algorithm

- **Step 1:** Allocate the space for the new node PTR
- **Step 2:** SET PTR -> DATA = VAL
- **Step 3:** IF FRONT = NULL
  SET FRONT = REAR = PTR
  SET FRONT -> NEXT = REAR -> NEXT = NULL
  ELSE
  SET REAR -> NEXT = PTR
  SET REAR = PTR
  SET REAR -> NEXT = NULL
  [END OF IF]
- **Step 4:** END

C Function

1. **void** insert(struct node *ptr, **int** item;)
2. {
3.

```
4.
5.    ptr = (struct node *) malloc (sizeof(struct node));
6.    if(ptr == NULL)
7.    {
8.       printf("\nOVERFLOW\n");
9.       return;
10.   }
11.   else
12.   {
13.      ptr -> data = item;
14.      if(front == NULL)
15.      {
16.         front = ptr;
17.         rear = ptr;
18.         front -> next = NULL;
19.         rear -> next = NULL;
20.      }
21.      else
22.      {
23.         rear -> next = ptr;
24.         rear = ptr;
```

```
25.          rear->next = NULL;
26.       }
27.    }
28.}
```

Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

```
1. ptr = front;
2.          front = front -> next;
3.          free(ptr);
```

The algorithm and C function is given as follows.

Algorithm

- **Step 1:** IF FRONT = NULL
  Write " Underflow "
  Go to Step 5
  [END OF IF]
- **Step 2:** SET PTR = FRONT
- **Step 3:** SET FRONT = FRONT -> NEXT

- **Step 4:** FREE PTR
- **Step 5:** END

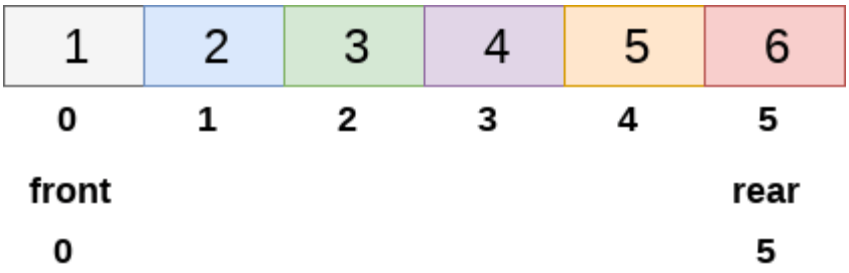
C Function

```
1. void delete (struct node *ptr)
2. {
3.     if(front == NULL)
4.     {
5.         printf("\nUNDERFLOW\n");
6.         return;
7.     }
8.     else
9.     {
10.        ptr = front;
11.        front = front -> next;
12.        free(ptr);
13.    }
14.}
```

Circular Queue

Deletions and insertions can only be performed at front and rear end respectively, as far as linear queue is concerned.

Consider the queue shown in the following figure.

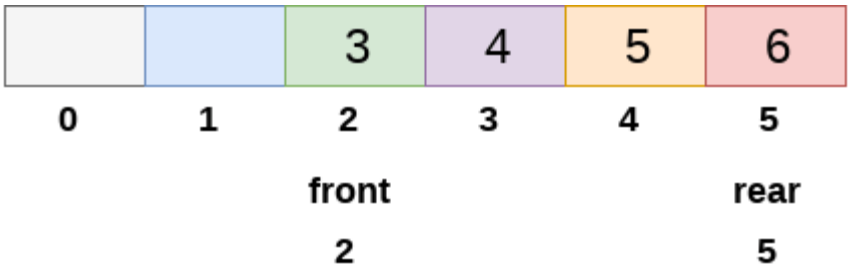


Queue

The Queue shown in above figure is completely filled and there can't be inserted any more element due to the condition **rear == max - 1 becomes true**.

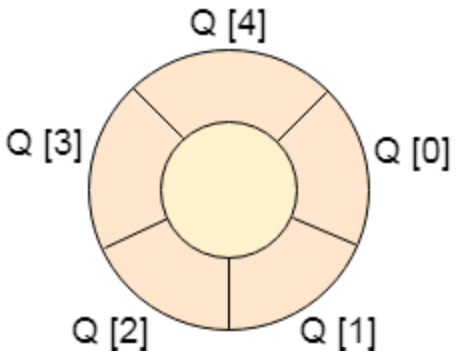
However, if we delete 2 elements at the front end of the queue, we still can not insert any element since the condition **rear = max -1 still holds**.

This is the main problem with the linear queue, although we have space available in the array, but we can not insert any more element in the queue. This is simply the memory wastage and we need to overcome this problem.



Queue after deletion of first 2 elements

One of the solution of this problem is circular queue. In the circular queue, the first index comes right after the last index. You can think of a circular queue as shown in the following figure.



Circular queue will be full when **front = -1** and **rear = max-1**. Implementation of circular queue is similar to that of a linear queue. Only the logic part that is implemented in the case of insertion and deletion is different from that in a linear queue.

Complexity

Time Complexity

Front	O(1)
Rear	O(1)
enQueue()	O(1)
deQueue()	O(1)

Insertion in Circular queue

There are three scenario of inserting an element in a queue.

1. **If (rear + 1)%maxsize = front**, the queue is full. In that case, overflow occurs and therefore, insertion can not be performed in the queue.
2. **If rear != max - 1**, then rear will be incremented to the **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
3. **If front != 0 and rear = max - 1**, then it means that queue is not full therefore, set the value of rear to 0 and insert the new element there.

Algorithm to insert an element in circular queue

- **Step 1:** IF (REAR+1)%MAX = FRONT
  Write " OVERFLOW "
  Goto step 4
  [End OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE IF REAR = MAX - 1 and FRONT ! = 0
  SET REAR = 0
  ELSE
  SET REAR = (REAR + 1) % MAX
  [END OF IF]
- **Step 3:** SET QUEUE[REAR] = VAL
- **Step 4:** EXIT

C Function

```
1. void insert(int item, int queue[])
2. {
```

```
3.  if((rear+1)%maxsize == front)
4.  {
5.      printf("\nOVERFLOW");
6.      return;
7.  }
8.  else if(front == -1 && rear == -1)
9.  {
10.     front = 0;
11.     rear = 0;
12. }
13. else if(rear == maxsize -1 && front != 0)
14. {
15.     rear = 0;
16. }
17. else
18. {
19.     rear = (rear+1)%maxsize;
20. }
21. queue[rear] = item;
22.}
```

Algorithm to delete an element from a circular queue

To delete an element from the circular queue, we must check for the three following conditions.

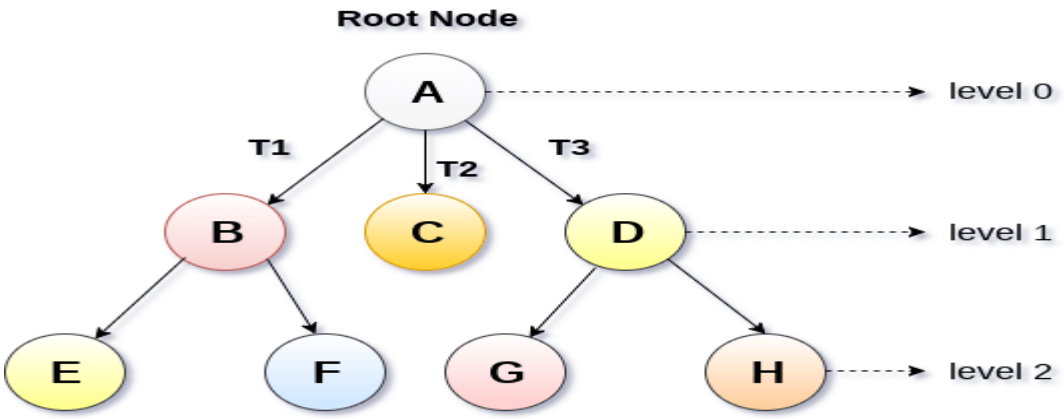
1. If front = -1, then there are no elements in the queue and therefore this will be the case of an underflow condition.
2. If there is only one element in the queue, in this case, the condition rear = front holds and therefore, both are set to -1 and the queue is deleted completely.
3. If front = max -1 then, the value is deleted from the front end the value of front is set to 0.
4. Otherwise, the value of front is incremented by 1 and then delete the element at the front end.

Algorithm

- **Step 1:** IF FRONT = -1
  Write " UNDERFLOW "
  Goto Step 4
  [END of IF]
- **Step 2:** SET VAL = QUEUE[FRONT]
- **Step 3:** IF FRONT = REAR
  SET FRONT = REAR = -1
  ELSE
  IF FRONT = MAX -1
  SET FRONT = 0
  ELSE
  SET FRONT = FRONT + 1
  [END of IF]
  [END OF IF]
- **Step 4:** EXIT

Tree

- A Tree is a recursive data structure containing the set of one or more data nodes where one node is designated as the root of the tree while the remaining nodes are called as the children of the root.
- The nodes other than the root node are partitioned into the non empty sets where each one of them is to be called sub-tree.
- Nodes of a tree either maintain a parent-child relationship between them or they are sister nodes.
- In a general tree, A node can have any number of children nodes but it can have only a single parent.
- The following image shows a tree, where the node A is the root node of the tree while the other nodes can be seen as the children of A.



Tree

Basic terminology

- **Root Node** :- The root node is the topmost node in the tree hierarchy. In other words, the root node is the one which doesn't have any parent.
- **Sub Tree** :- If the root node is not null, the tree T1, T2 and T3 is called sub-trees of the root node.
- **Leaf Node** :- The node of tree, which doesn't have any child node, is called leaf node. Leaf node is the bottom most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Path** :- The sequence of consecutive edges is called path. In the tree shown in the above image, path to the node E is A→ B → E.
- **Ancestor node** :- An ancestor of a node is any predecessor node on a path from root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, the node F have the ancestors, B and A.
- **Degree** :- Degree of a node is equal to number of children, a node have. In the tree shown in the above image, the degree of node B is 2. Degree of a leaf node is always 0 while in a complete binary tree, degree of each node is equal to 2.
- **Level Number** :- Each node of the tree is assigned a level number in such a way that each node is present at one level higher than its parent. Root node of the tree is always present at level 0.

Static representation of tree

```
1. #define MAXNODE 500
2. struct treenode {
3.     int root;
4.     int father;
```

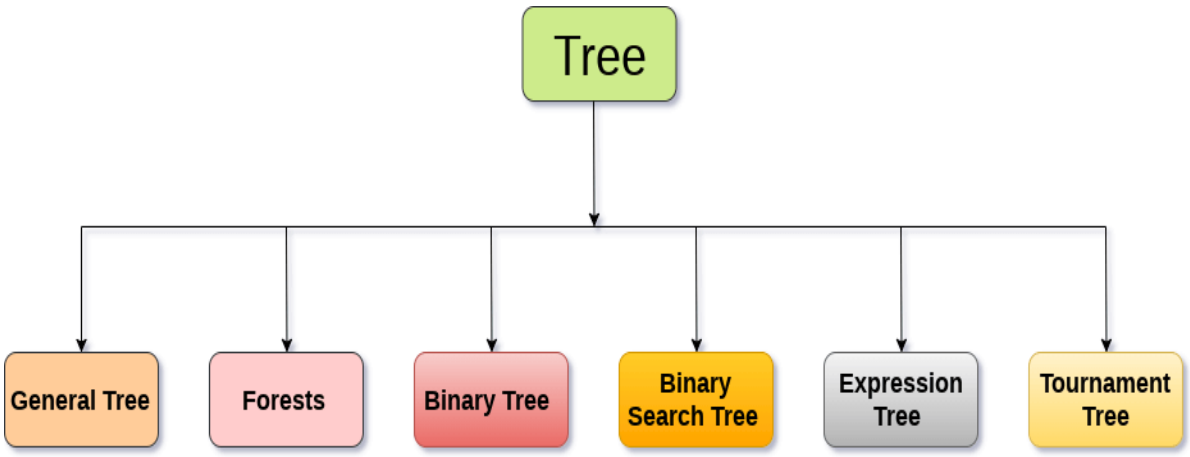
```
5.    int son;
6.    int next;
7.  }
```

Dynamic representation of tree

```
1.  struct treenode
2.  {
3.     int root;
4.     struct treenode *father;
5.     struct treenode *son
6.     struct treenode *next;
7.  }
```

Types of Tree

The tree data structure can be classified into six different categories.

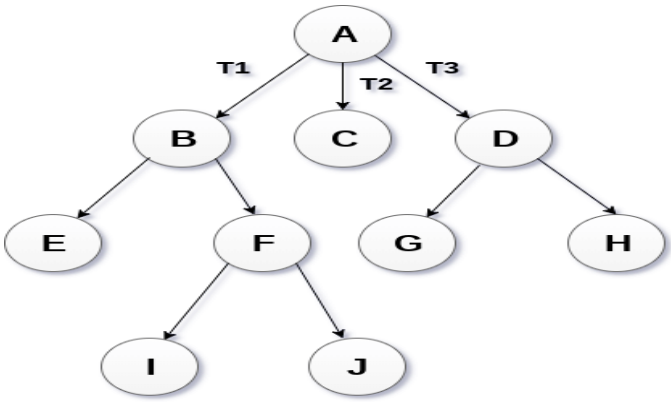


General Tree

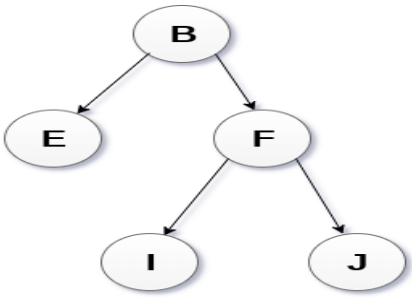
General Tree stores the elements in a hierarchical order in which the top level element is always present at level 0 as the root element. All the nodes except the root node are present at number of levels. The nodes which are present on the same level are called siblings while the nodes which are present on the different levels exhibit the parent-child relationship among them. A node may contain any number of sub-trees. The tree in which each node contain 3 sub-tree, is called ternary tree.

Forests

Forest can be defined as the set of disjoint trees which can be obtained by deleting the root node and the edges which connects root node to the first level node.



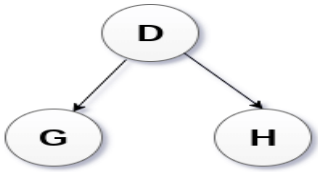
General Tree



Forest 1



Forest 2



Forest 3

Binary Tree

Binary tree is a data structure in which each node can have at most 2 children. The node present at the top most level is called the root node. A node with the 0

children is called leaf node. Binary Trees are used in the applications like expression evaluation and many more. We will discuss binary tree in detail, later in this tutorial.

Binary Search Tree

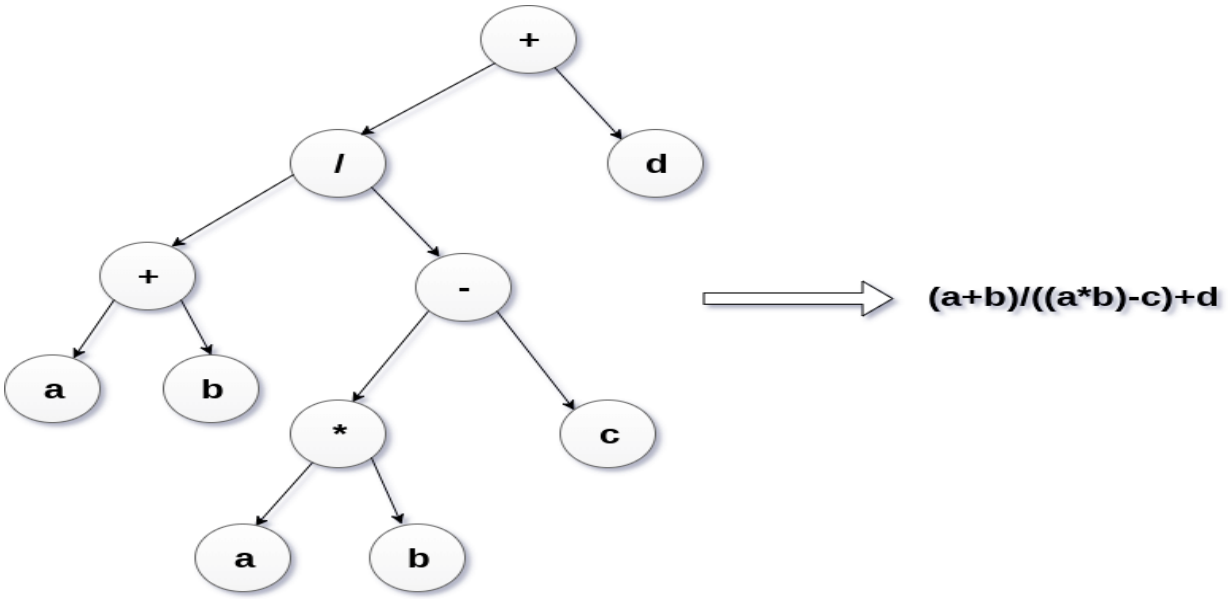
Binary search tree is an ordered binary tree. All the elements in the left sub-tree are less than the root while elements present in the right sub-tree are greater than or equal to the root node element. Binary search trees are used in most of the applications of computer science domain like searching, sorting, etc.

Expression Tree

Expression trees are used to evaluate the simple arithmetic expressions. Expression tree is basically a binary tree where internal nodes are represented by operators while the leaf nodes are represented by operands. Expression trees are widely used to solve algebraic expressions like (a+b)*(a-b). Consider the following example.

Q. Construct an expression tree by using the following algebraic expression.

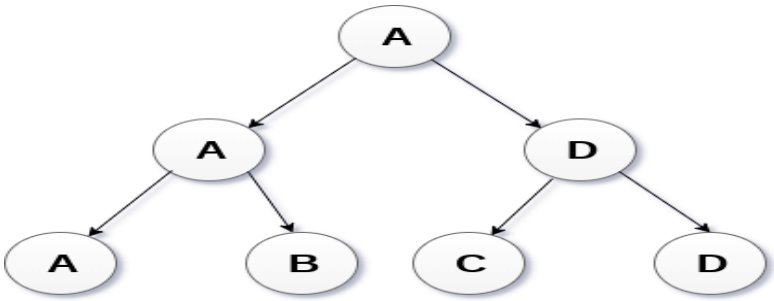
(a + b) / (a*b - c) + d



Tournament Tree

Tournament tree are used to record the winner of the match in each round being played between two players. Tournament tree can also be called as selection tree or winner tree. External nodes represent the players among which a match is being played while the internal nodes represent the winner of the match played. At the top most level, the winner of the tournament is present as the root node of the tree.

For example, tree .of a chess tournament being played among 4 players is shown as follows. However, the winner in the left sub-tree will play against the winner of right sub-tree.

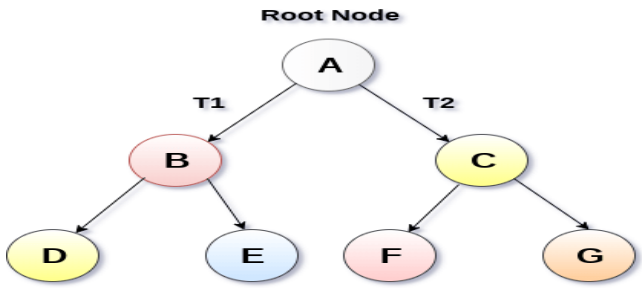


Binary Tree

Binary Tree is a special type of generic tree in which, each node can have at most two children. Binary tree is generally partitioned into three disjoint subsets.

1. Root of the node
2. left sub-tree which is also a binary tree.
3. Right binary sub-tree

A binary Tree is shown in the following image.



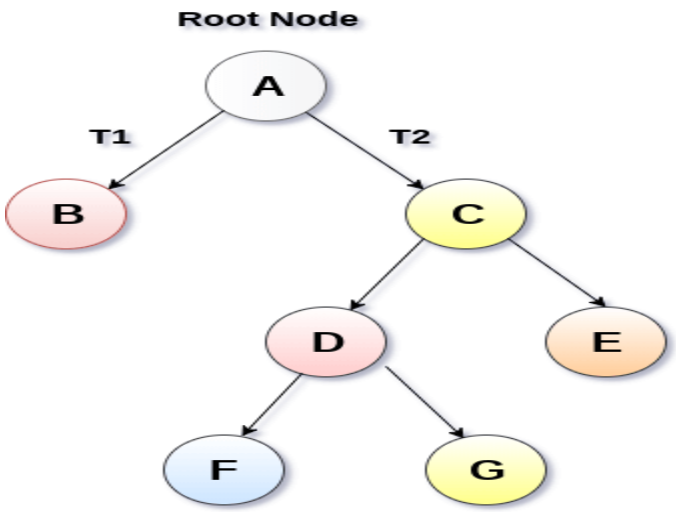
Binary Tree

Types of Binary Tree

1. Strictly Binary Tree

In Strictly Binary Tree, every non-leaf node contain non-empty left and right sub-trees. In other words, the degree of every non-leaf node will always be 2. A strictly binary tree with n leaves, will have (2n - 1) nodes.

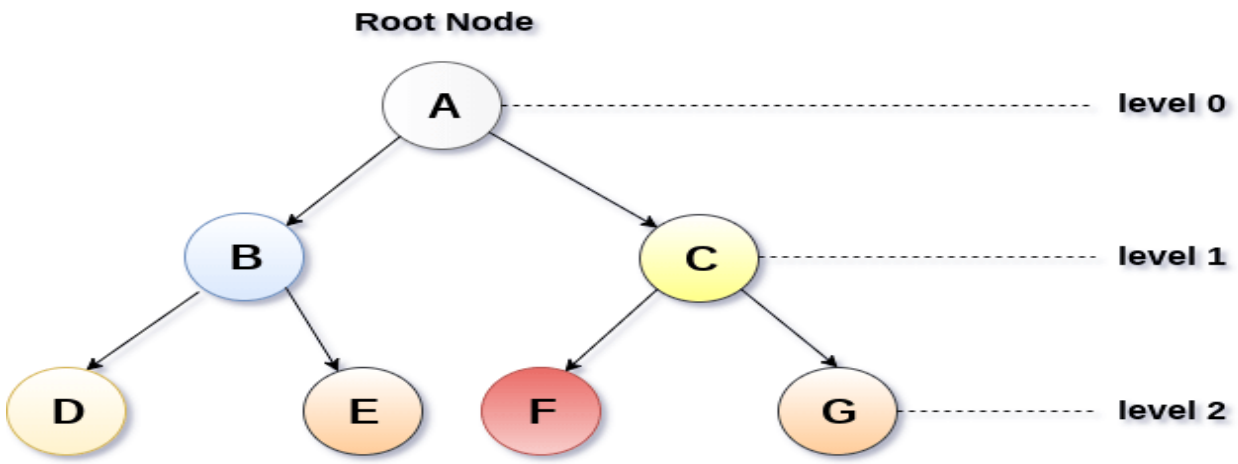
A strictly binary tree is shown in the following figure.



Strictly Binary Tree

2. Complete Binary Tree

A Binary Tree is said to be a complete binary tree if all of the leaves are located at the same level d. A complete binary tree is a binary tree that contains exactly 2^l nodes at each level between level 0 and d. The total number of nodes in a complete binary tree with depth d is 2^{d+1}-1 where leaf nodes are 2^d while non-leaf nodes are 2^d-1.



Complete Binary Tree

Binary Tree Traversal

SN	Traversal	Description
1	Pre-order Traversal	Traverse the root first then traverse into the left sub-tree and right sub-tree respectively. This procedure will be applied to each sub-tree of the tree recursively.
2	In-order Traversal	Traverse the left sub-tree first, and then traverse the root and the right sub-tree respectively. This procedure will be applied to each sub-tree of the tree recursively.

3	Post-order Traversal	Traverse the left sub-tree and then traverse the right sub-tree and root respectively. This procedure will be applied to each sub-tree of the tree recursively.

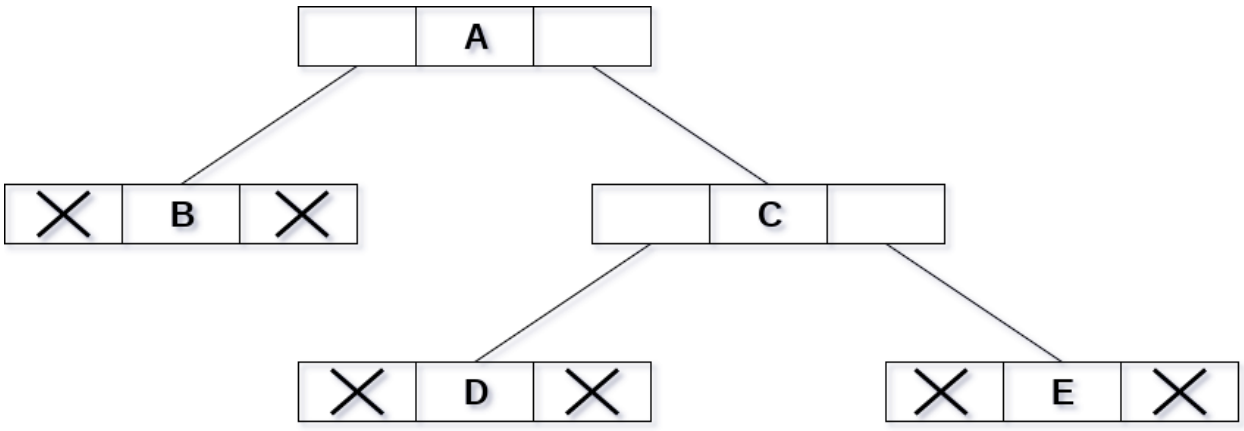
Binary Tree representation

There are two types of representation of a binary tree:

1. Linked Representation

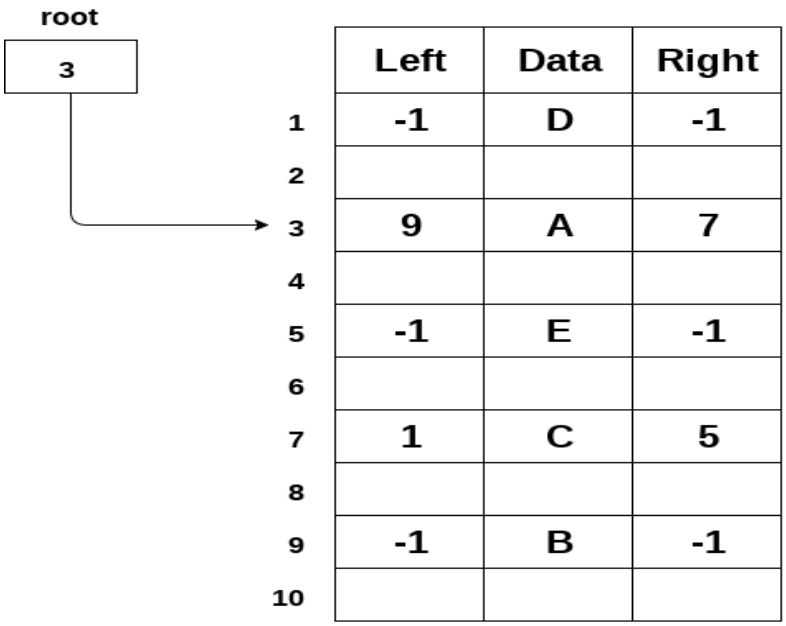
In this representation, the binary tree is stored in the memory, in the form of a linked list where the number of nodes are stored at non-contiguous memory locations and linked together by inheriting parent child relationship like a tree. every node contains three parts : pointer to the left node, data element and pointer to the right node. Each binary tree has a root pointer which points to the root node of the binary tree. In an empty binary tree, the root pointer will point to null.

Consider the binary tree given in the figure below.



In the above figure, a tree is seen as the collection of nodes where each node contains three parts : left pointer, data element and right pointer. Left pointer stores the address of the left child while the right pointer stores the address of the right child. The leaf node contains **null** in its left and right pointers.

The following image shows about how the memory will be allocated for the binary tree by using linked representation. There is a special pointer maintained in the memory which points to the root node of the tree. Every node in the tree contains the address of its left and right child. Leaf node contains null in its left and right pointers.



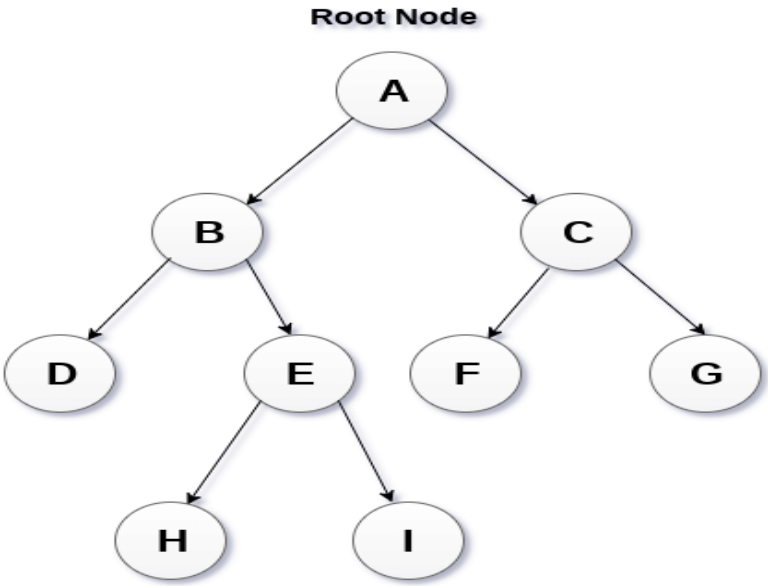
	Left	Data	Right
1	-1	D	-1
2			
3	9	A	7
4			
5	-1	E	-1
6			
7	1	C	5
8			
9	-1	B	-1
10			

Memory Allocation of Binary Tree using linked Representation

2. Sequential Representation

This is the simplest memory allocation technique to store the tree elements but it is an inefficient technique since it requires a lot of space to store the tree

elements. A binary tree is shown in the following figure along with its memory allocation.



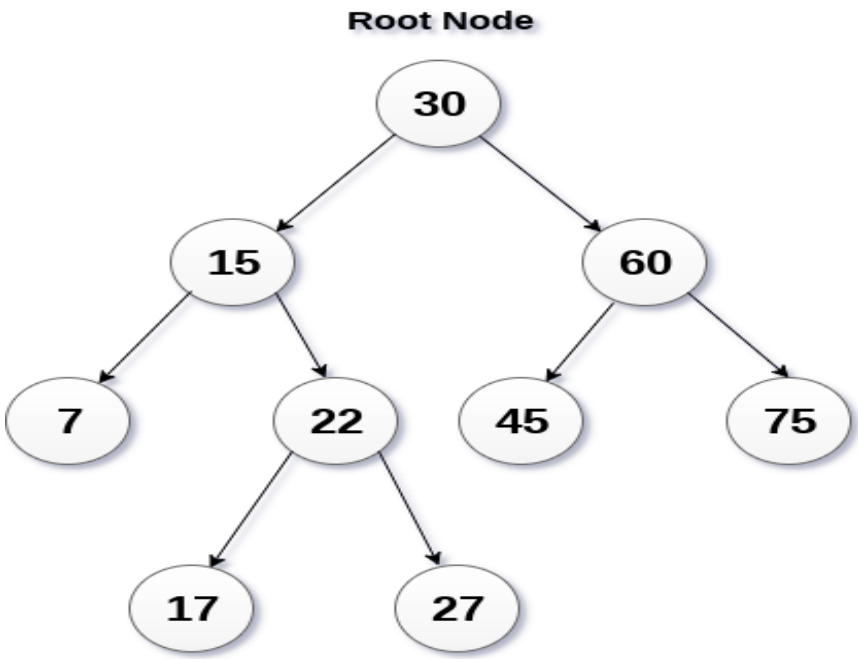
A	B	C	D	E	F	G			H	I
1	2	3	4	5	6	7	8	9	10	11

Sequential Representation of Binary Tree

In this representation, an array is used to store the tree elements. Size of the array will be equal to the number of nodes present in the tree. The root node of the tree will be present at the 1st index of the array. If a node is stored at ith index then its left and right children will be stored at 2i and 2i+1 location. If the 1st index of the array i.e. tree[1] is 0, it means that the tree is empty.

Binary Search Tree

1. Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree.
2. In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
3. Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.
4. This rule will be recursively applied to all the left and right sub-trees of the root.



Binary Search Tree

A Binary search tree is shown in the above figure. As the constraint applied on the BST, we can see that the root node 30 doesn't contain any value greater than or

equal to 30 in its left sub-tree and it also doesn't contain any value less than 30 in its right sub-tree.

Advantages of using binary search tree

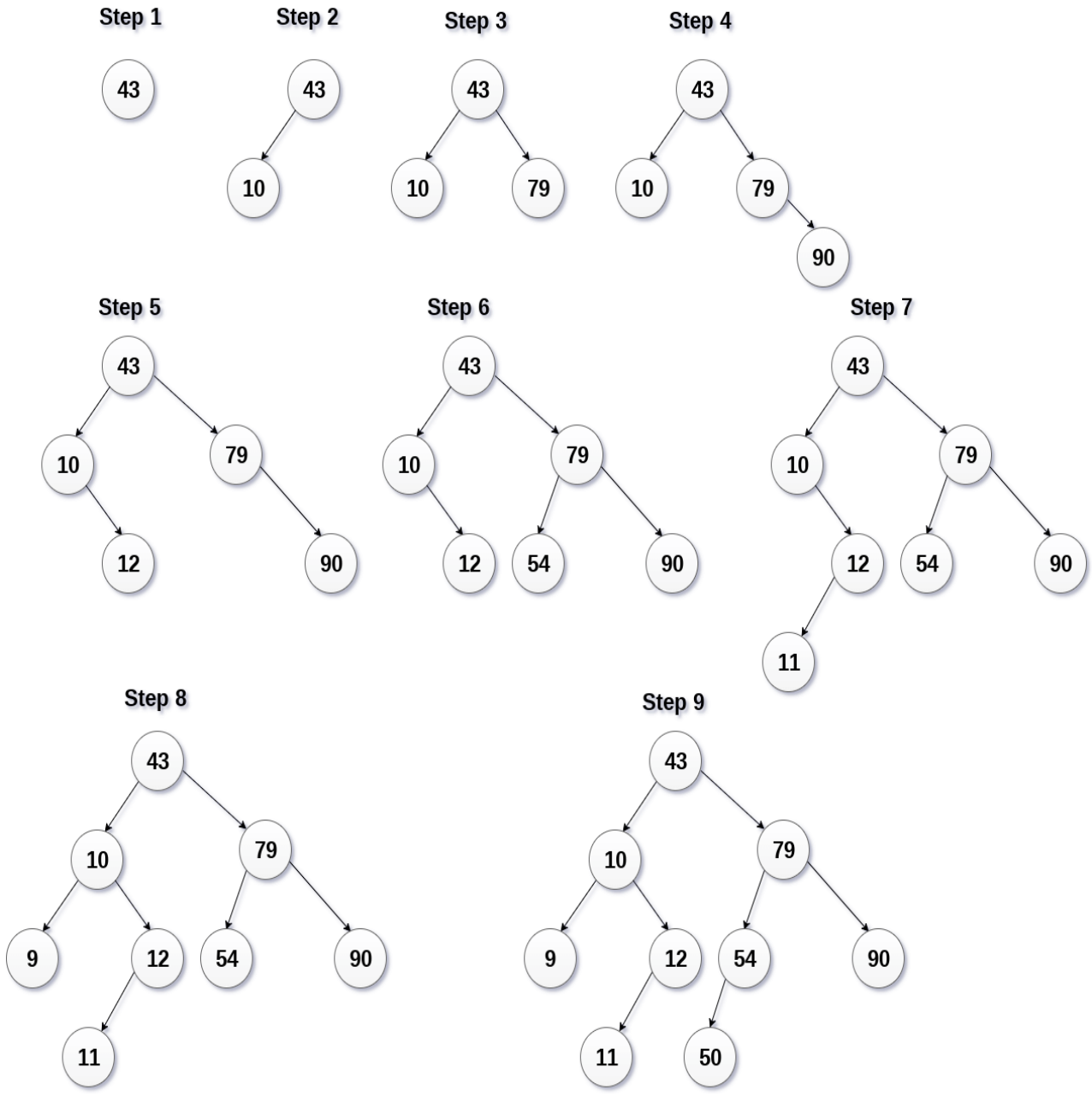
1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes o($\log_2 n$) time. In worst case, the time it takes to search an element is 0(n).
3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

Q. Create the binary search tree using the following data elements.

43, 10, 79, 90, 12, 54, 11, 9, 50

1. Insert 43 into the tree as the root of the tree.
2. Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right sub-tree.

The process of creating BST by using the given elements, is shown in the image below.



Binary search Tree Creation

Operations on Binary Search Tree

There are many operations which can be performed on a binary search tree.

SN	Operation	Description
1	Searching in BST	Finding the location of some specific element in a binary search tree.
2	Insertion in BST	Adding a new element to the binary search tree at the appropriate location so that the property of BST do not violate.
3	Deletion in BST	Deleting some specific node from a binary search tree. However, there can be various cases in deletion depending upon the number of children, the node have.

AVL Tree

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

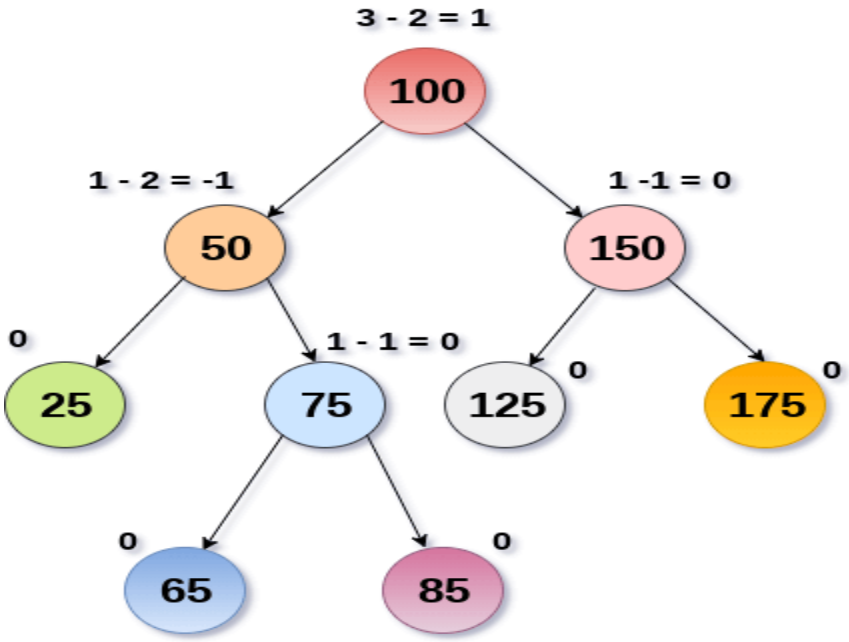
Balance Factor (k) = height (left(k)) - height (right(k))

If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



AVL Tree

Complexity

Algorithm	Average case	Worst case
Space	o(n)	o(n)
Search	o(log n)	o(log n)
Insert	o(log n)	o(log n)
Delete	o(log n)	o(log n)

Operations on AVL tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.

2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

Why AVL Tree ?

AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is **O(h)**. However, it can be extended to **O(n)** if the BST becomes skewed (i.e. worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be **O(log n)** where n is the number of nodes.

B Tree

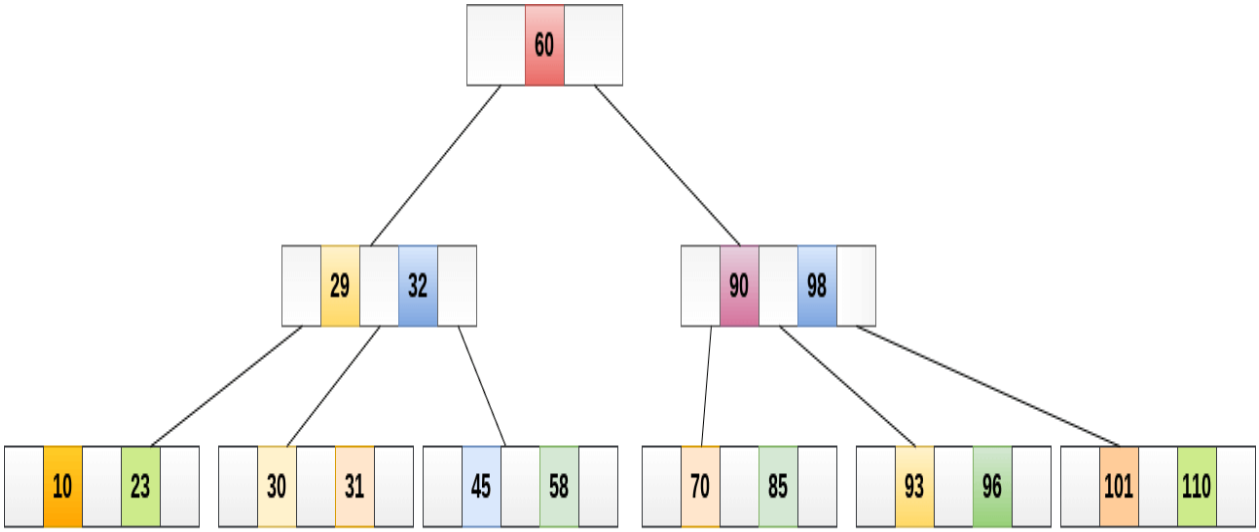
B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

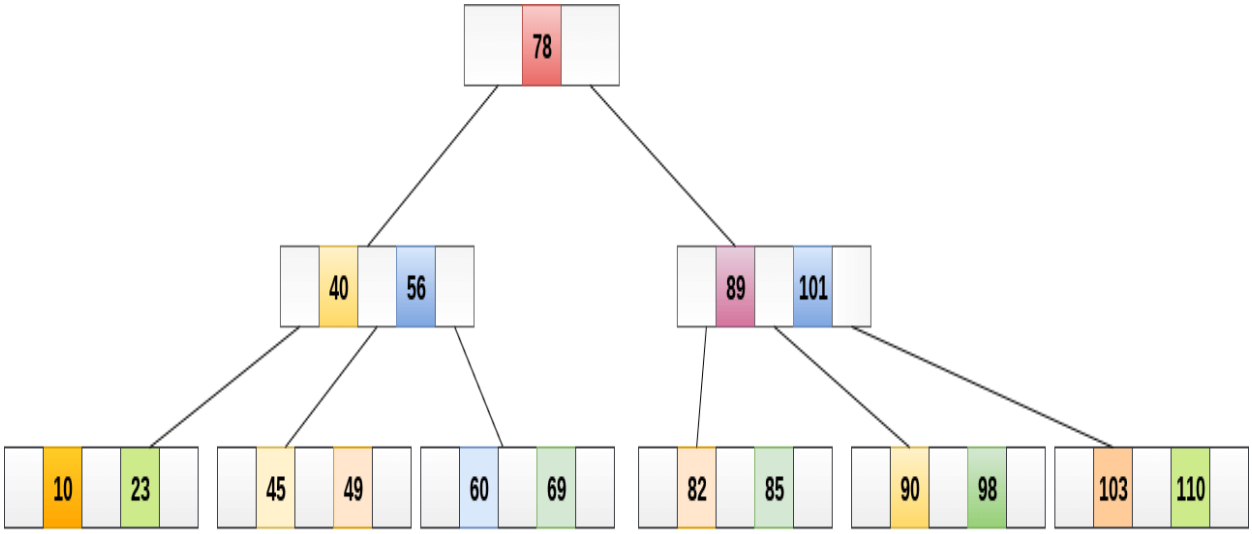
Operations

Searching :

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

1. Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
2. Since, 40<49<56, traverse right sub-tree of 40.
3. 49>45, move to right. Compare 49.
4. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes O(log n) time to search any element in a B tree.



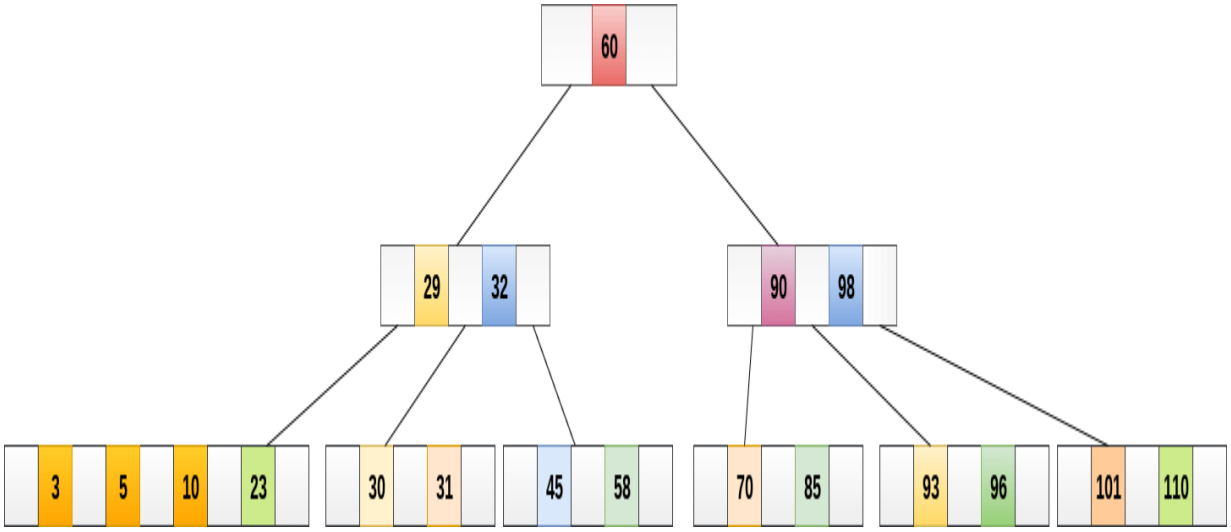
Inserting

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

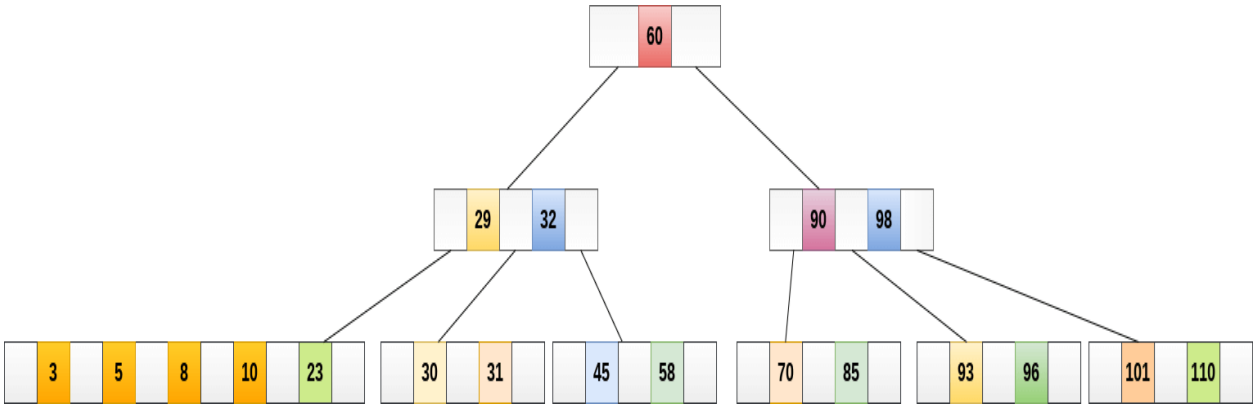
1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
    - Insert the new element in the increasing order of elements.
    - Split the node into the two nodes at the median.
    - Push the median element upto its parent node.
    - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

Example:

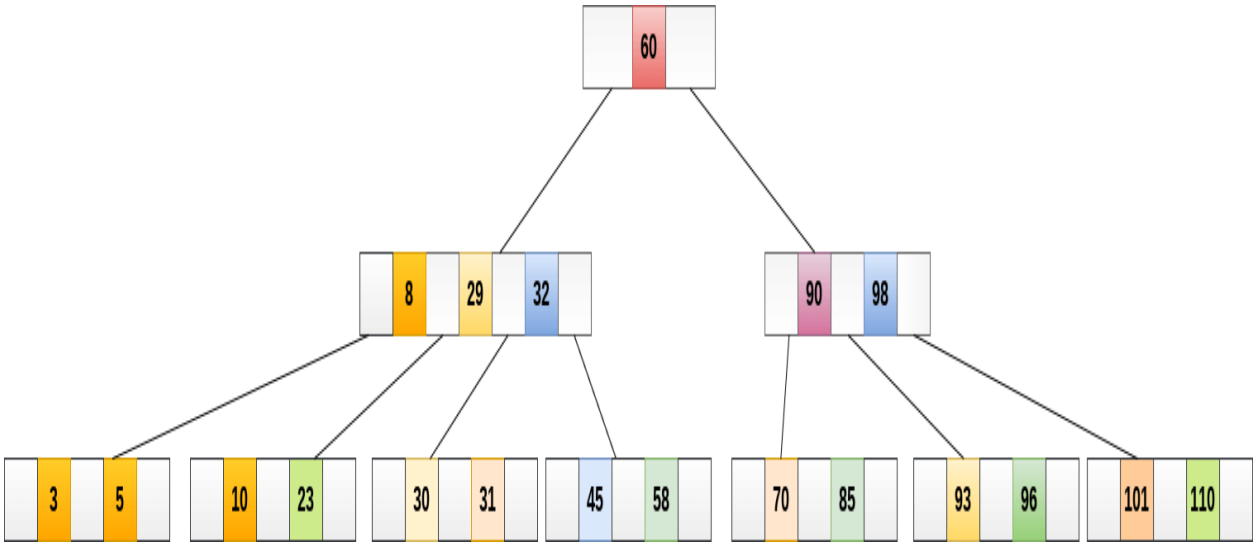
Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5, therefore insert 8.



The node, now contain 5 keys which is greater than (5 -1 = 4) keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



Deletion

Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

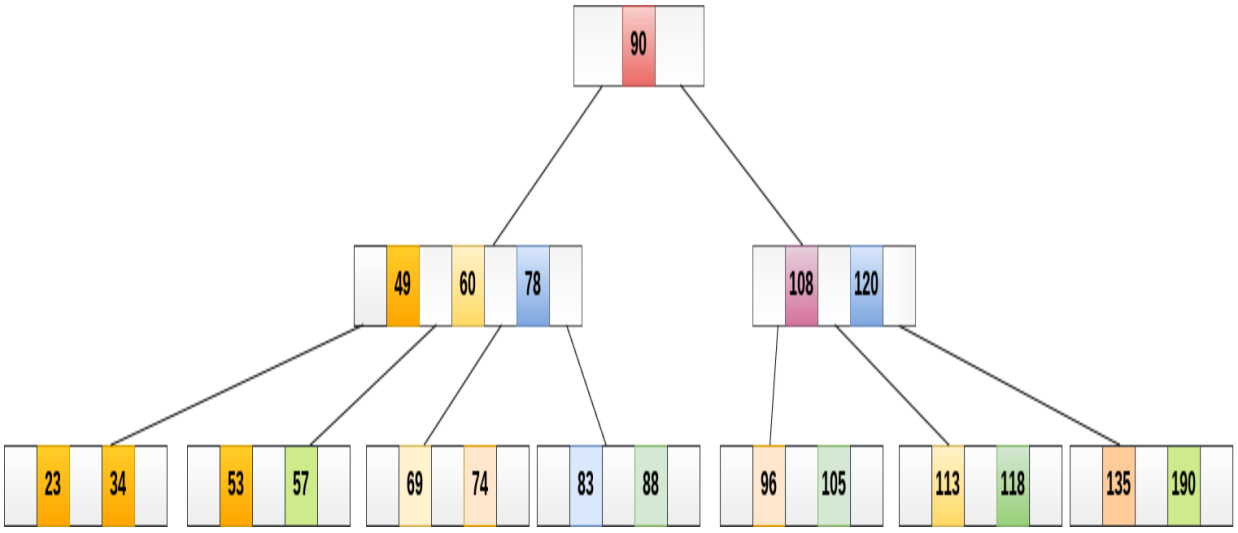
1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
    - If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
    - If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.

4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than m/2 nodes then, apply the above process on the parent too.

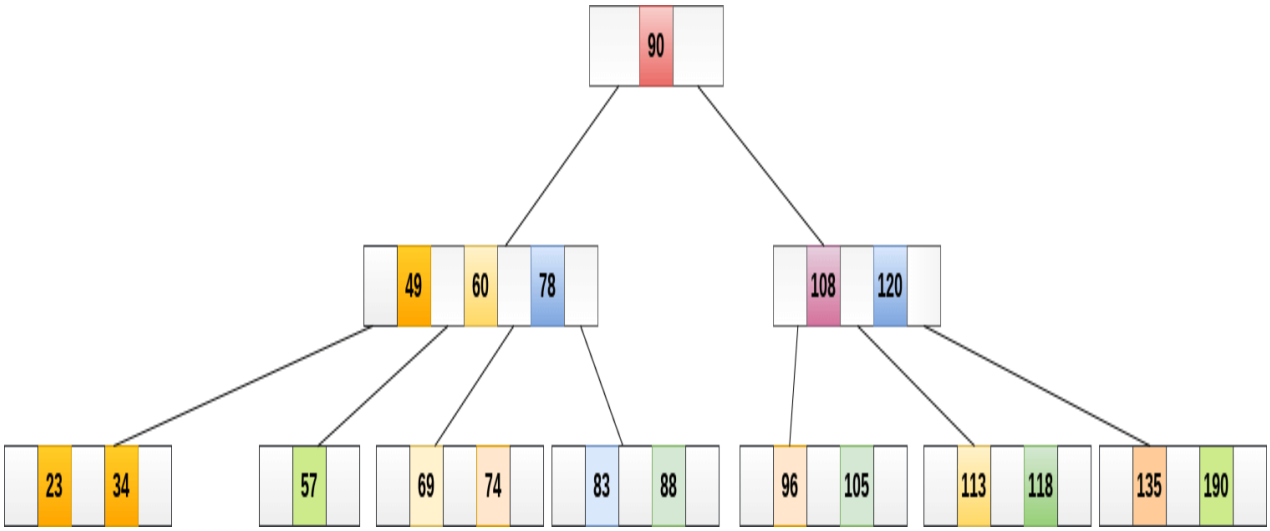
If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example 1

Delete the node 53 from the B Tree of order 5 shown in the following figure.

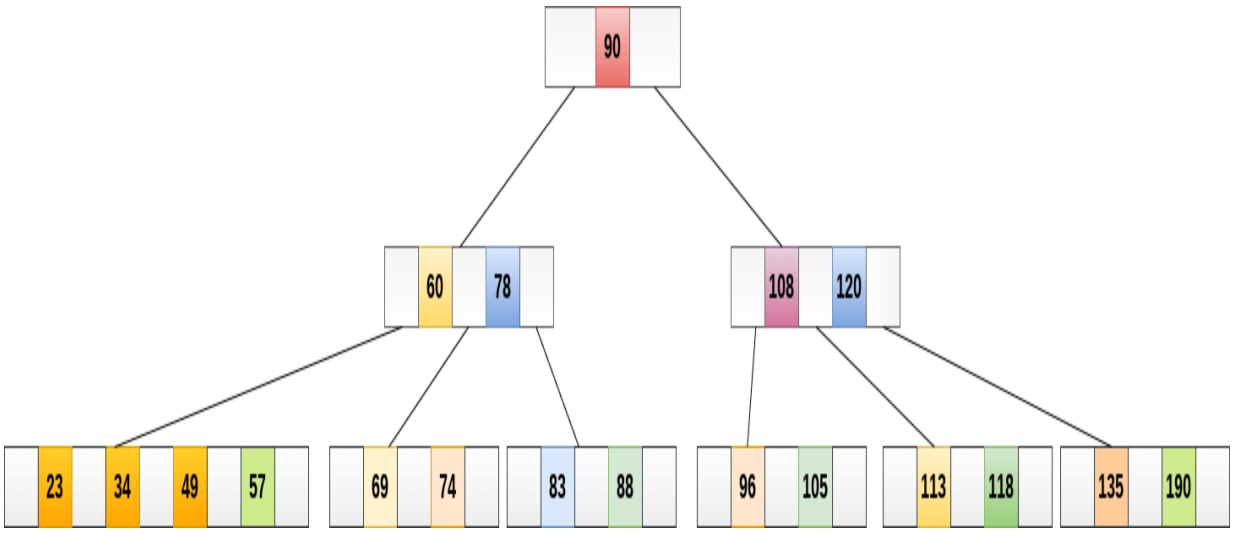


53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.



Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case.

B+ Tree

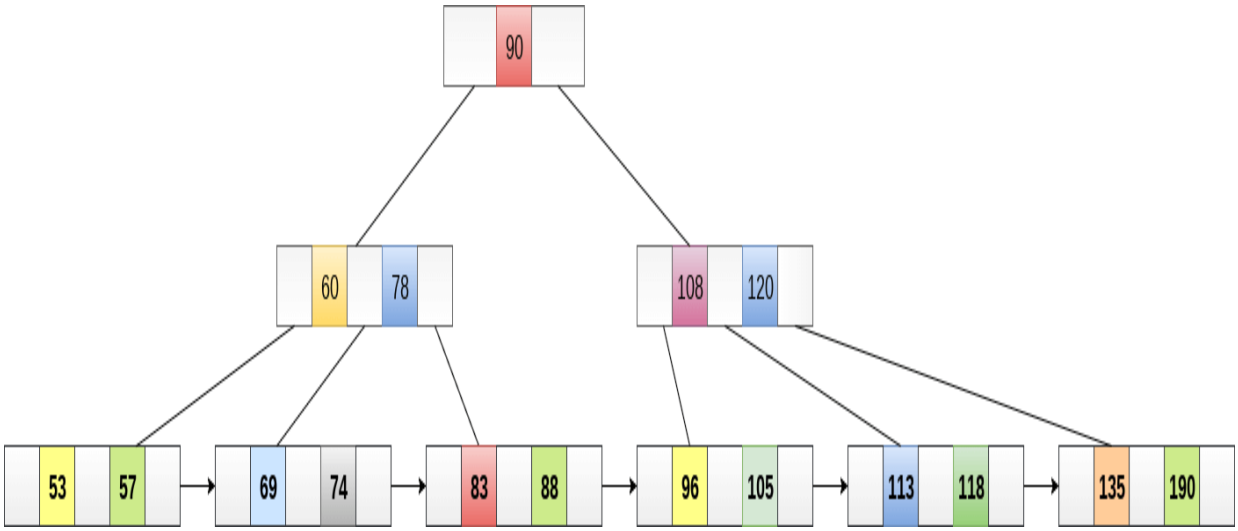
B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.

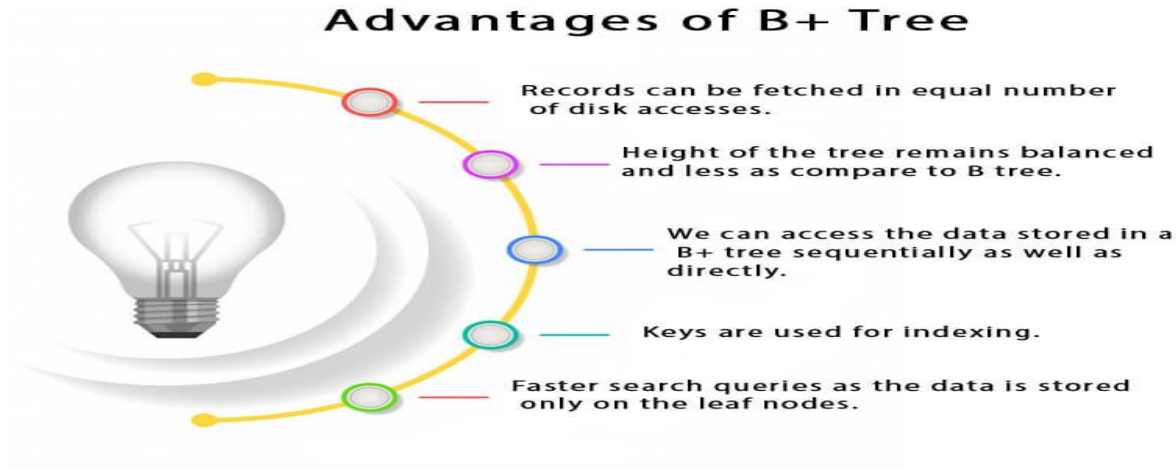
B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure.



Advantages of B+ Tree

1. Records can be fetched in equal number of disk accesses.
2. Height of the tree remains balanced and less as compare to B tree.
3. We can access the data stored in a B+ tree sequentially as well as directly.
4. Keys are used for indexing.
5. Faster search queries as the data is stored only on the leaf nodes.



B Tree VS B+ Tree

SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

Insertion in B+ Tree

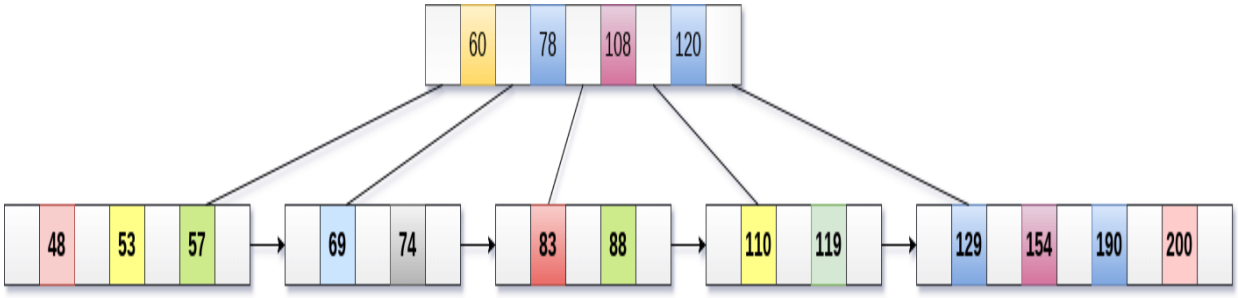
Step 1: Insert the new node as a leaf node

Step 2: If the leaf doesn't have required space, split the node and copy the middle node to the next index node.

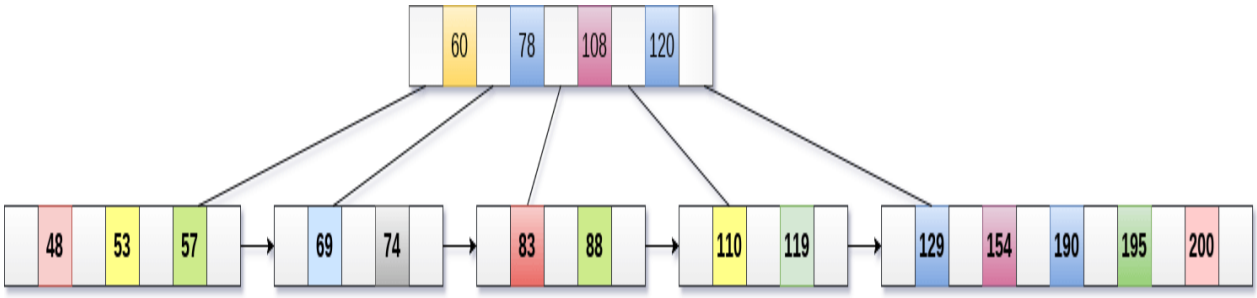
Step 3: If the index node doesn't have required space, split the node and copy the middle element to the next index page.

Example :

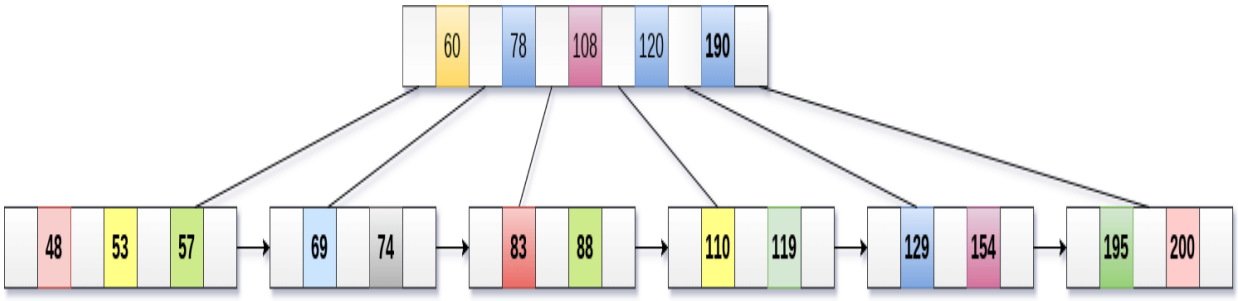
Insert the value 195 into the B+ tree of order 5 shown in the following figure.



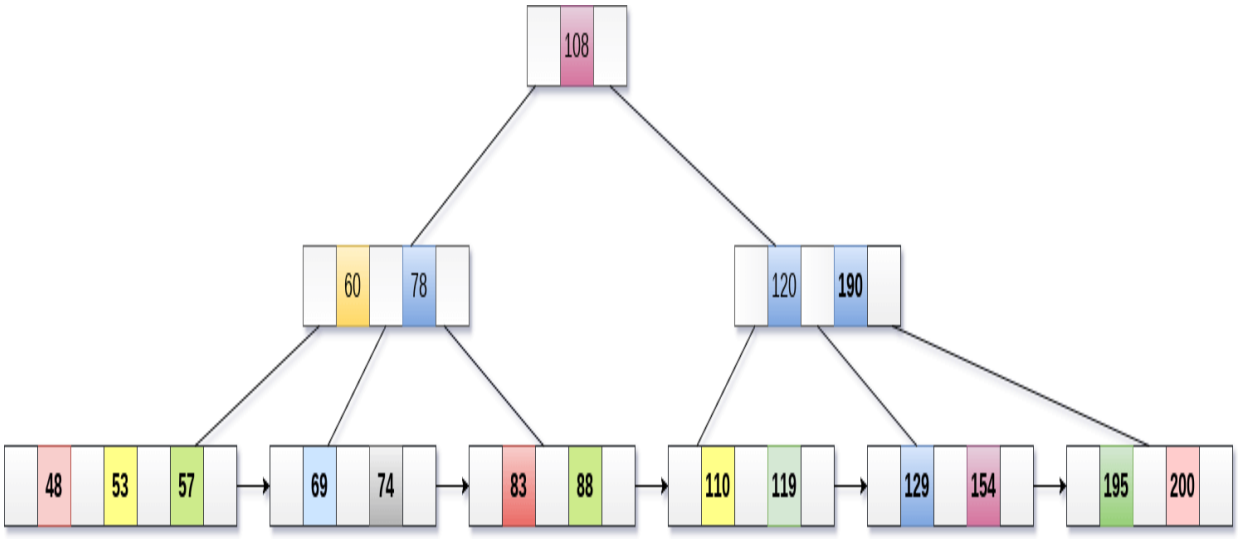
195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position.



The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the median node up to the parent.



Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown as follows.



Deletion in B+ Tree

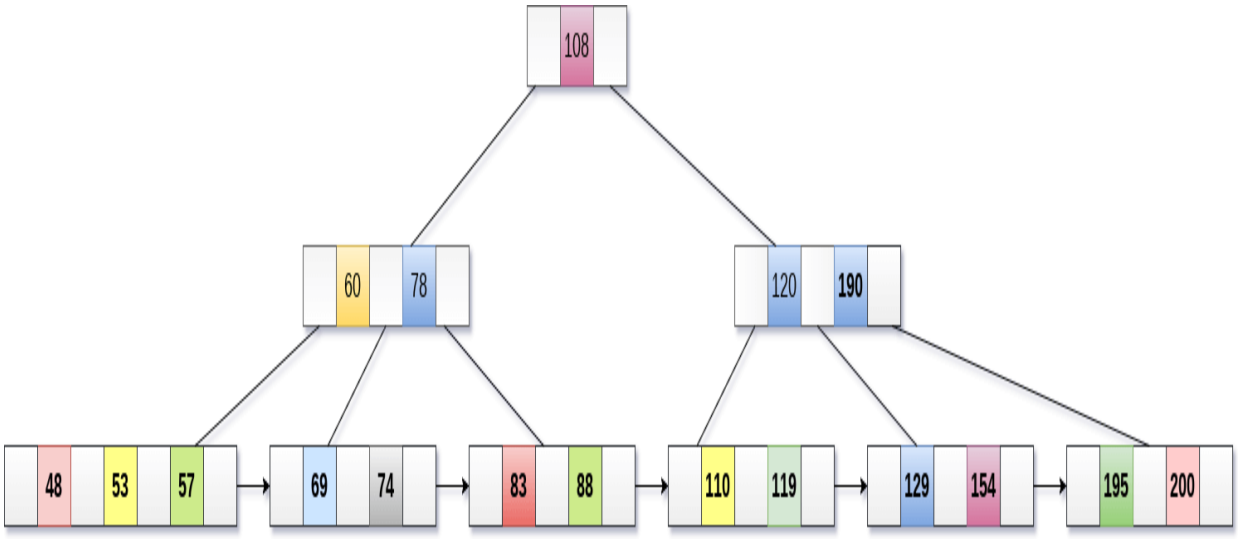
Step 1: Delete the key and data from the leaves.

Step 2: if the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.

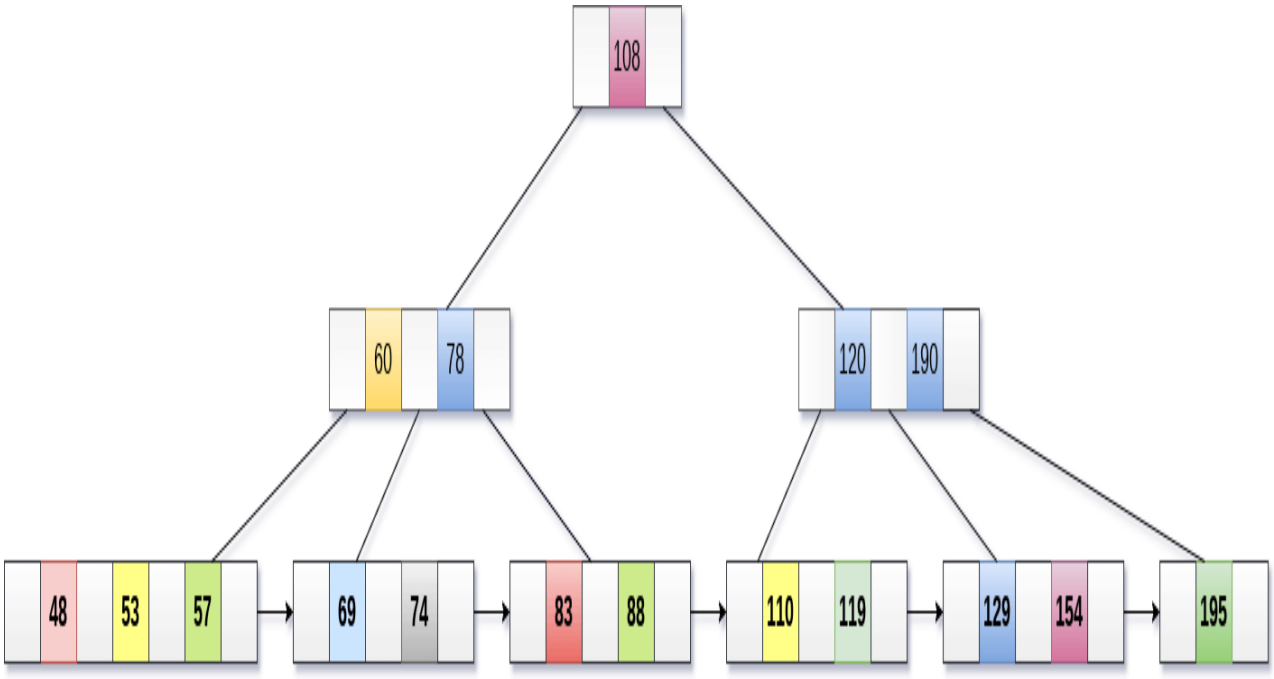
Step 3: if the index node contains less than minimum number of elements, merge the node with the sibling and move down the key in between them.

Example

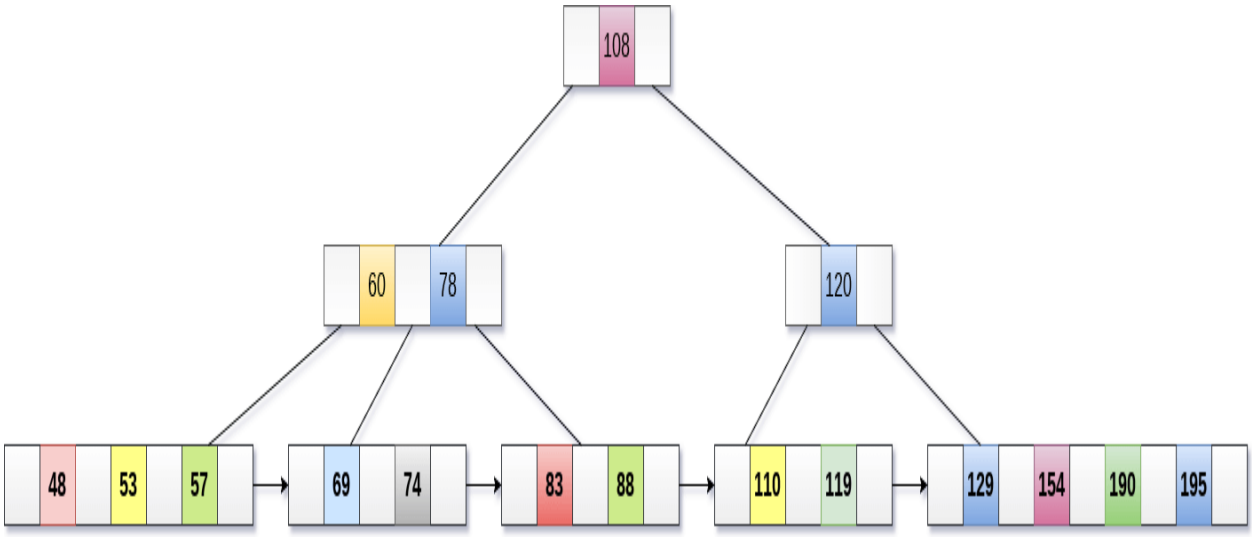
Delete the key 200 from the B+ Tree shown in the following figure.



200 is present in the right sub-tree of 190, after 195. delete it.

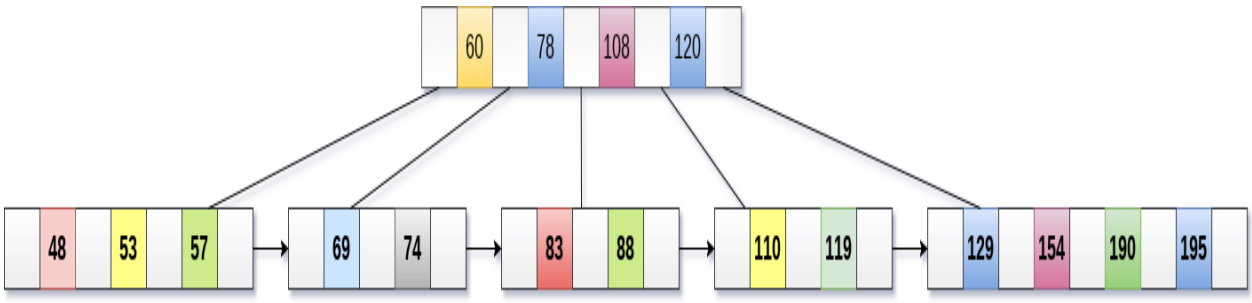


Merge the two nodes by using 195, 190, 154 and 129.



Now, element 120 is the single element present in the node which is violating the B+ Tree properties. Therefore, we need to merge it by using 60, 78, 108 and 120.

Now, the height of B+ tree will be decreased by 1.



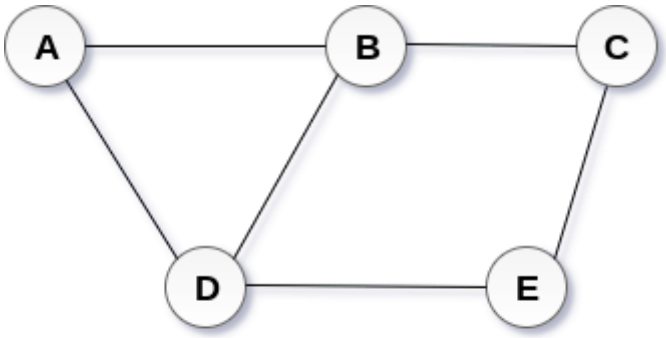
Graph

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

Definition

A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



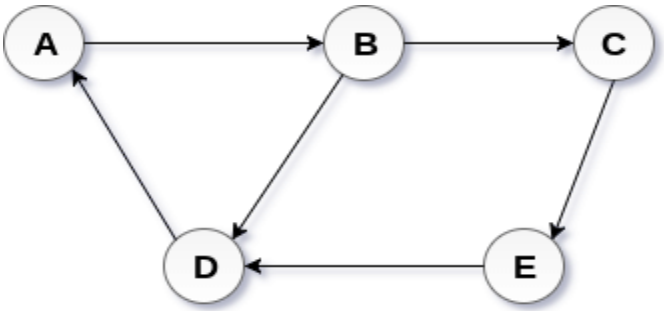
Undirected Graph

Directed and Undirected Graph

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.

A directed graph is shown in the following figure.



Directed Graph

Graph Terminology

Path

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Closed Path

A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0=V_N$.

Simple Path

If all the nodes of the graph are distinct with an exception $V_0=V_N$, then such path P is called as closed simple path.

Cycle

A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Connected Graph

A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.

Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

Loop

An edge that is associated with the similar end points can be called as Loop.

Adjacent Nodes

If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

Degree of the Node

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

Graph Representation

By Graph representation, we simply mean the technique which is to be used in order to store some graph into the computer's memory.

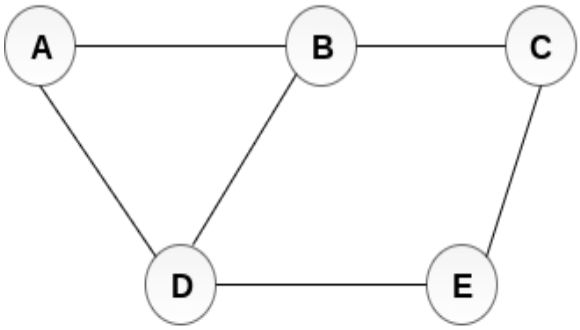
There are two ways to store Graph into the computer's memory. In this part of this tutorial, we discuss each one of them in detail.

1. Sequential Representation

In sequential representation, we use adjacency matrix to store the mapping represented by vertices and edges. In adjacency matrix, the rows and columns are represented by the graph vertices. A graph having n vertices, will have a dimension **n x n**.

An entry M_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if there exists an edge between V_i and V_j.

An undirected graph and its adjacency matrix representation is shown in the following figure.



Undirected Graph

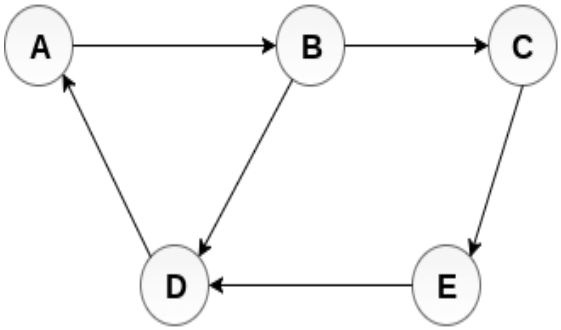
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

in the above figure, we can see the mapping among the vertices (A, B, C, D, E) is represented by using the adjacency matrix which is also shown in the figure.

There exists different adjacency matrices for the directed and undirected graph. In directed graph, an entry A_{ij} will be 1 only when there is an edge directed from V_i to V_j.

A directed graph and its adjacency matrix representation is shown in the following figure.



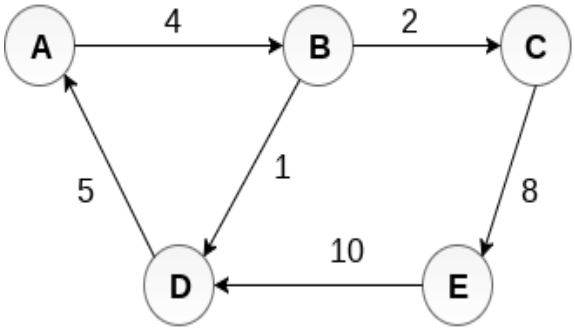
Directed Graph

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

Representation of weighted directed graph is different. Instead of filling the entry by 1, the Non- zero entries of the adjacency matrix are represented by the weight of respective edges.

The weighted directed graph along with the adjacency matrix representation is shown in the following figure.



Weighted Directed Graph

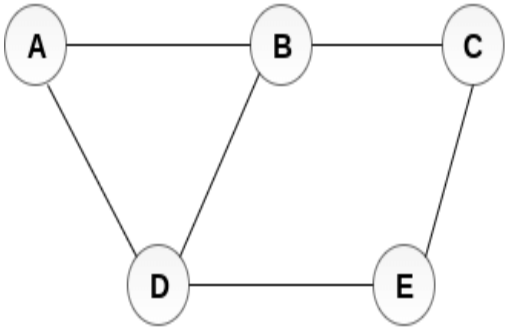
	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

Adjacency Matrix

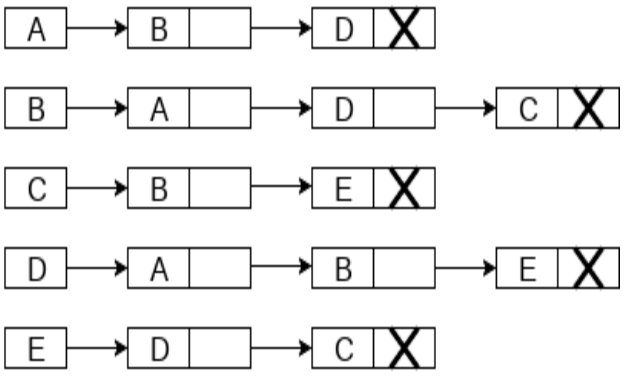
Linked Representation

In the linked representation, an adjacency list is used to store the Graph into the computer's memory.

Consider the undirected graph shown in the following figure and check the adjacency list representation.



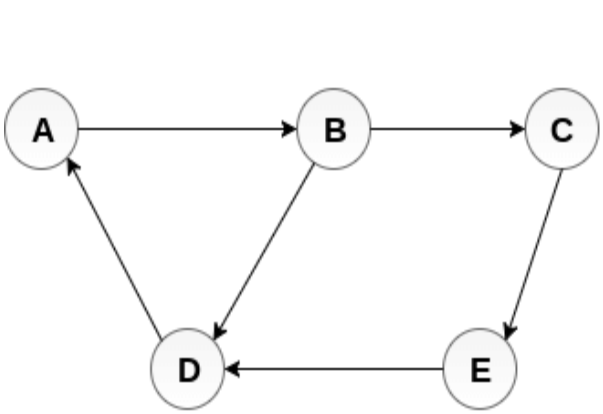
Undirected Graph



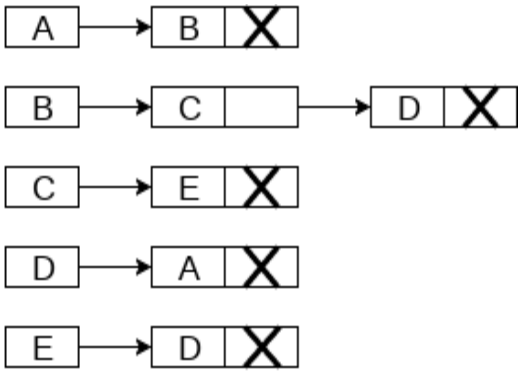
Adjacency List

An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list. The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.

Consider the directed graph shown in the following figure and check the adjacency list representation of the graph.



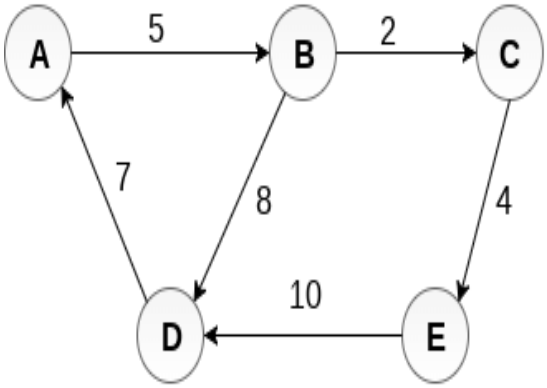
Directed Graph



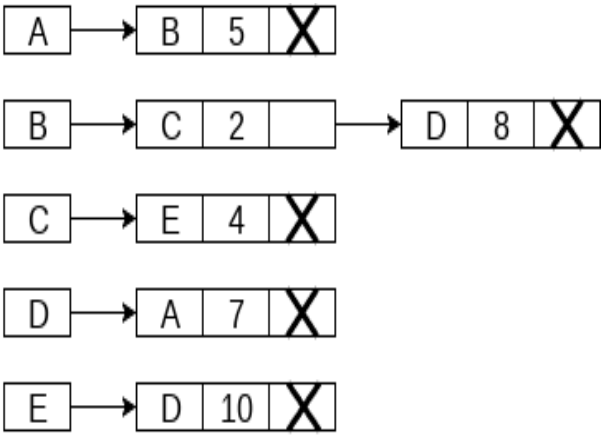
Adjacency List

In a directed graph, the sum of lengths of all the adjacency lists is equal to the number of edges present in the graph.

In the case of weighted directed graph, each node contains an extra field that is called the weight of the node. The adjacency list representation of a directed graph is shown in the following figure.



Weighted Directed Graph



Adjacency List

Graph Traversal Algorithm

In this part of the tutorial we will discuss the techniques by using which, we can traverse all the vertices of the graph.

Traversing the graph means examining all the nodes and vertices of the graph. There are two standard methods by using which, we can traverse the graphs. Lets discuss each one of them in detail.

- Breadth First Search
- Depth First Search

Breadth First Search (BFS) Algorithm

Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

The algorithm of breadth first search is given below. The algorithm starts with examining the node A and all of its neighbours. In the next step, the neighbours of

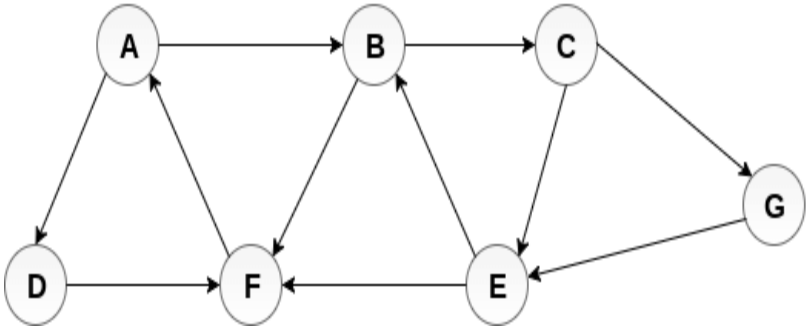
the nearest node of A are explored and process continues in the further steps. The algorithm explores all neighbours of all the nodes and ensures that each node is visited exactly once and no node is visited twice.

Algorithm

- **Step 1:** SET STATUS = 1 (ready state)
  for each node in G
- **Step 2:** Enqueue the starting node A
  and set its STATUS = 2
  (waiting state)
- **Step 3:** Repeat Steps 4 and 5 until
  QUEUE is empty
- **Step 4:** Dequeue a node N. Process it
  and set its STATUS = 3
  (processed state).
- **Step 5:** Enqueue all the neighbours of
  N that are in the ready state
  (whose STATUS = 1) and set
  their STATUS = 2
  (waiting state)
  [END OF LOOP]
- **Step 6:** EXIT

Example

Consider the graph G shown in the following image, calculate the minimum path p from node A to node E. Given that each edge has a length of 1.



Adjacency Lists

A : B, D
B : C, F
C : E, G
G : E
E : B, F
F : A
D : F

Solution:

Minimum Path P can be found by applying breadth first search algorithm that will begin at node A and will end at E. the algorithm uses two queues, namely **QUEUE1** and **QUEUE2**. **QUEUE1** holds all the nodes that are to be processed while **QUEUE2** holds all the nodes that are processed and deleted from **QUEUE1**.

Lets start examining the graph from Node A.

1. Add A to QUEUE1 and NULL to QUEUE2.

    1. QUEUE1 = {A}
    2. QUEUE2 = {NULL}

2. Delete the Node A from QUEUE1 and insert all its neighbours. Insert Node A into QUEUE2

    1. QUEUE1 = {B, D}
    2. QUEUE2 = {A}

3. Delete the node B from QUEUE1 and insert all its neighbours. Insert node B into QUEUE2.

1. QUEUE1 = {D, C, F}
2. QUEUE2 = {A, B}

4. Delete the node D from QUEUE1 and insert all its neighbours. Since F is the only neighbour of it which has been inserted, we will not insert it again. Insert node D into QUEUE2.

1. QUEUE1 = {C, F}
2. QUEUE2 = { A, B, D}

5. Delete the node C from QUEUE1 and insert all its neighbours. Add node C to QUEUE2.

1. QUEUE1 = {F, E, G}
2. QUEUE2 = {A, B, D, C}

6. Remove F from QUEUE1 and add all its neighbours. Since all of its neighbours has already been added, we will not add them again. Add node F to QUEUE2.

1. QUEUE1 = {E, G}
2. QUEUE2 = {A, B, D, C, F}

7. Remove E from QUEUE1, all of E's neighbours has already been added to QUEUE1 therefore we will not add them again. All the nodes are visited and the target node i.e. E is encountered into QUEUE2.

1. QUEUE1 = {G}
2. QUEUE2 = {A, B, D, C, F, E}

Now, backtrack from E to A, using the nodes available in QUEUE2.

The minimum path will be **A → B → C → E**.

Depth First Search (DFS) Algorithm

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

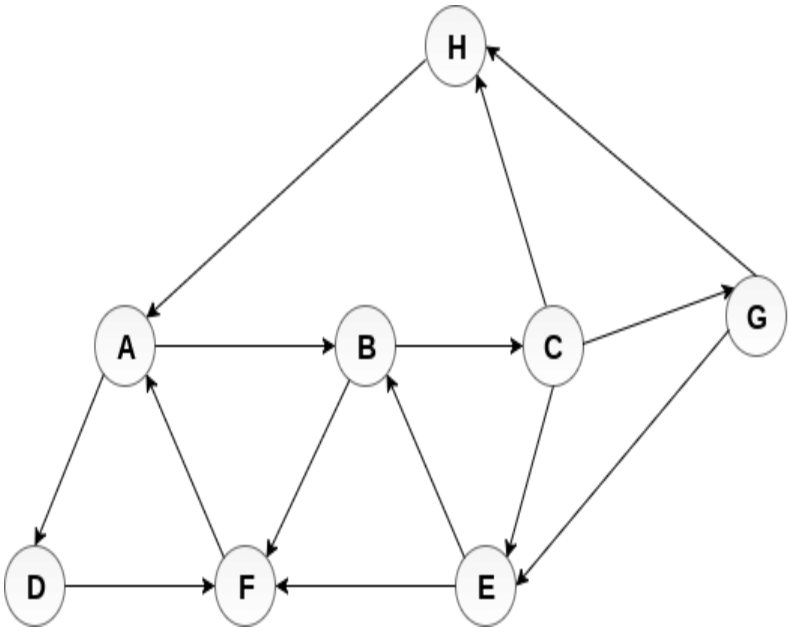
The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

Algorithm

- **Step 1:** SET STATUS = 1 (ready state) for each node in G
- **Step 2:** Push the starting node A on the stack and set its STATUS = 2 (waiting state)
- **Step 3:** Repeat Steps 4 and 5 until STACK is empty
- **Step 4:** Pop the top node N. Process it and set its STATUS = 3 (processed state)
- **Step 5:** Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their
  STATUS = 2 (waiting state)
  [END OF LOOP]
- **Step 6:** EXIT

Example :

Consider the graph G along with its adjacency list, given in the figure below. Calculate the order to print all the nodes of the graph starting from node H, by using depth first search (DFS) algorithm.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Solution :

Push H onto the stack

1. STACK : H

POP the top element of the stack i.e. H, print it and push all the neighbours of H onto the stack that are is ready state.

1. Print H
2. STACK : A

Pop the top element of the stack i.e. A, print it and push all the neighbours of A onto the stack that are in ready state.

1. Print A
2. Stack : B, D

Pop the top element of the stack i.e. D, print it and push all the neighbours of D onto the stack that are in ready state.

1. Print D
2. Stack : B, F

Pop the top element of the stack i.e. F, print it and push all the neighbours of F onto the stack that are in ready state.

1. Print F
2. Stack : B

Pop the top of the stack i.e. B and push all the neighbours

1. Print B
2. Stack : C

Pop the top of the stack i.e. C and push all the neighbours.

1. Print C
2. Stack : E, G

Pop the top of the stack i.e. G and push all its neighbours.

1. Print G
2. Stack : E

Pop the top of the stack i.e. E and push all its neighbours.

1. Print E
2. Stack :

Hence, the stack now becomes empty and all the nodes of the graph have been traversed.

The printing sequence of the graph will be :

1. H → A → D → F → B → C → G → E

Spanning Tree

Spanning tree can be defined as a sub-graph of connected, undirected graph G that is a tree produced by removing the desired number of edges from a graph. In other words, Spanning tree is a non-cyclic sub-graph of a connected and undirected graph G that connects all the vertices together. A graph G can have multiple spanning trees.

Minimum Spanning Tree

There can be weights assigned to every edge in a weighted graph. However, A minimum spanning tree is a spanning tree which has minimal total weight. In other words, minimum spanning tree is the one which contains the least weight among all other spanning tree of some particular graph.

Shortest path algorithms

In this section, algorithms will be discussed to calculate the shortest path between two nodes in a graph.

There are two algorithms which are being used for this purpose.

Prim's Algorithm

Prim's Algorithm is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explore all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

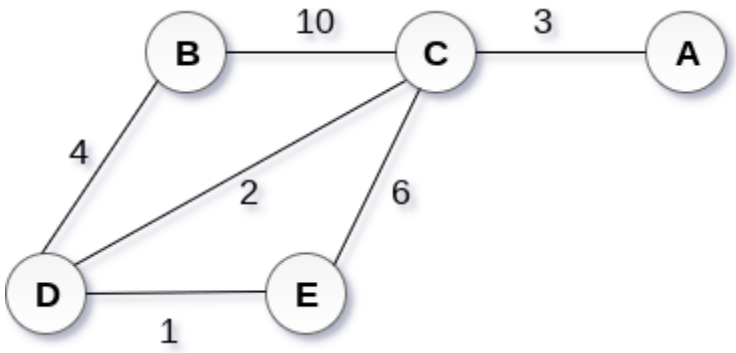
The algorithm is given as follows.

Algorithm

- **Step 1:** Select a starting vertex
- **Step 2:** Repeat Steps 3 and 4 until there are fringe vertices
- **Step 3:** Select an edge e connecting the tree vertex and fringe vertex that has minimum weight
- **Step 4:** Add the selected edge and the vertex to the minimum spanning tree T
  [END OF LOOP]
- **Step 5:** EXIT

Example :

Construct a minimum spanning tree of the graph given in the following figure by using prim's algorithm.



Solution

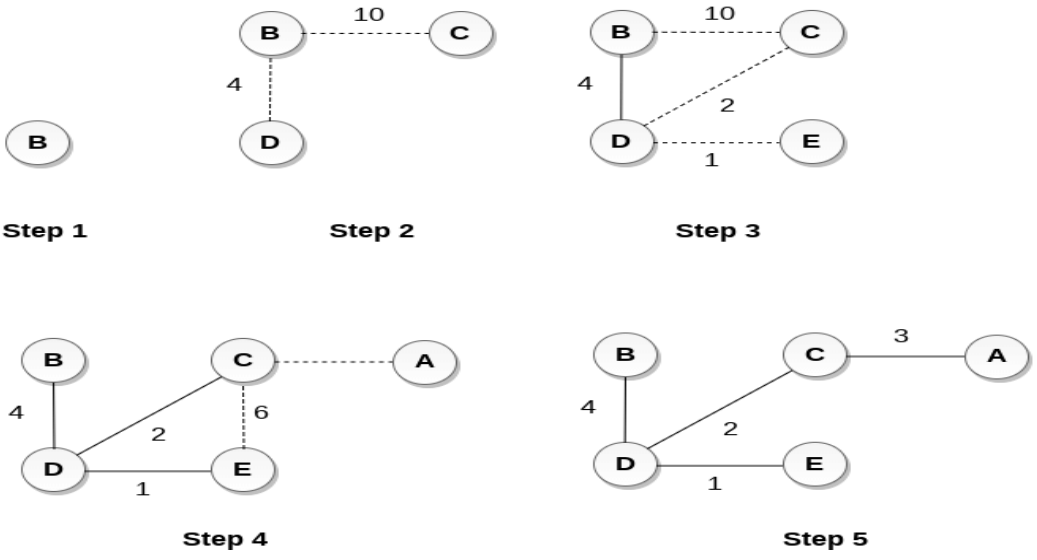
- **Step 1 :** Choose a starting vertex B.
- **Step 2:** Add the vertices that are adjacent to A. the edges that connecting the vertices are shown by dotted lines.

- **Step 3:** Choose the edge with the minimum weight among all. i.e. BD and add it to MST. Add the adjacent vertices of D i.e. C and E.
- **Step 3:** Choose the edge with the minimum weight among all. In this case, the edges DE and CD are such edges. Add them to MST and explore the adjacent of C i.e. E and A.
- **Step 4:** Choose the edge with the minimum weight i.e. CA. We can't choose CE as it would cause cycle in the graph.

  The graph produces in the step 4 is the minimum spanning tree of the graph shown in the above figure.

  The cost of MST will be calculated as;

  cost(MST) = 4 + 2 + 1 + 3 = 10 units.



Kruskal's Algorithm

Kruskal's Algorithm is used to find the minimum spanning tree for a connected weighted graph. The main target of the algorithm is to find the subset of edges by using which, we can traverse every vertex of the graph.

Kruskal's algorithm follows greedy approach which finds an optimum solution at every stage instead of focusing on a global optimum.

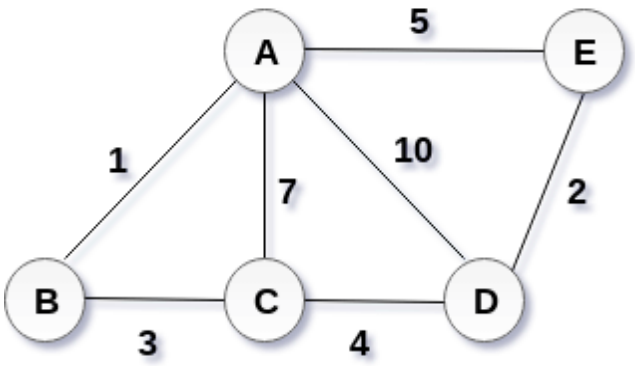
The Kruskal's algorithm is given as follows.

Algorithm

- **Step 1:** Create a forest in such a way that each graph is a separate tree.
- **Step 2:** Create a priority queue Q that contains all the edges of the graph.
- **Step 3:** Repeat Steps 4 and 5 while Q is NOT EMPTY
- **Step 4:** Remove an edge from Q
- **Step 5:** IF the edge obtained in Step 4 connects two different trees, then Add it to the forest (for combining two trees into one tree).
  ELSE
  Discard the edge
- **Step 6:** END

Example :

Apply the Kruskal's algorithm on the graph given as follows.



Solution:

the weight of the edges given as :

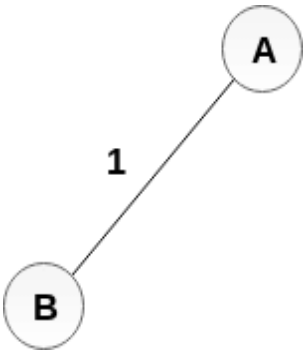
Edge	AE	AD	AC	AB	BC	CD	DE
Weight	5	10	7	1	3	4	2

Sort the edges according to their weights.

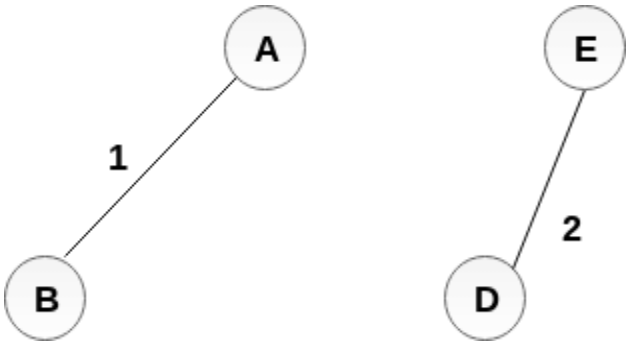
Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

Start constructing the tree;

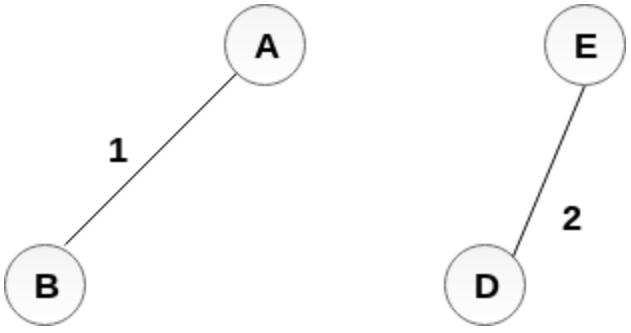
Add AB to the MST;

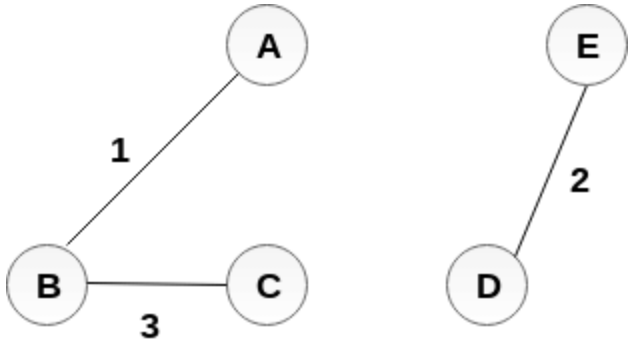


Add DE to the MST;



Add BC to the MST;





The next step is to add AE, but we can't add that as it will cause a cycle.

The next edge to be added is AC, but it can't be added as it will cause a cycle.

The next edge to be added is AD, but it can't be added as it will contain a cycle.

Hence, the final MST is the one which is shown in the step 4.

the cost of MST = 1 + 2 + 3 + 4 = 10.

Sorting Algorithms

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios −

- **Telephone Directory** − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary** − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

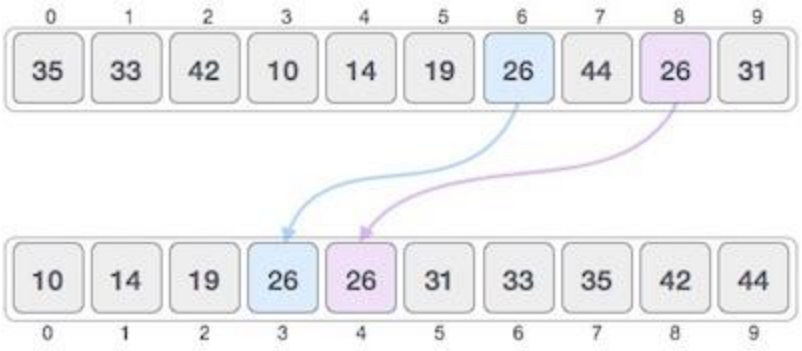
In-place Sorting and Not-in-place Sorting

Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called **in-place sorting**. Bubble sort is an example of in-place sorting.

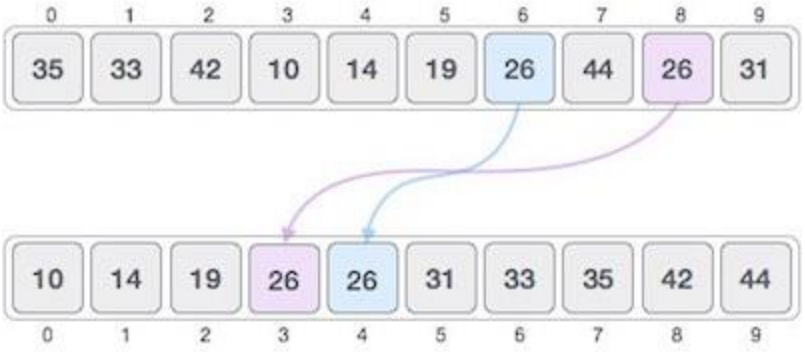
However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called **not-in-place sorting**. Merge-sort is an example of not-in-place sorting.

Stable and Not Stable Sorting

If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called **stable sorting**.



If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.



Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.

Adaptive and Non-Adaptive Sorting Algorithm

A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them.

A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sortedness.

Important Terms

Some terms are generally coined while discussing sorting techniques, here is a brief introduction to them −

Increasing Order

A sequence of values is said to be in **increasing order**, if the successive element is greater than the previous one. For example, 1, 3, 4, 6, 8, 9 are in increasing order, as every next element is greater than the previous element.

Decreasing Order

A sequence of values is said to be in **decreasing order**, if the successive element is less than the current one. For example, 9, 8, 6, 4, 3, 1 are in decreasing order, as every next element is less than the previous element.

Non-Increasing Order

A sequence of values is said to be in **non-increasing order**, if the successive element is less than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 9, 8, 6, 3, 3, 1 are in non-increasing order, as every next element is less than or equal to (in case of 3) but not greater than any previous element.

Non-Decreasing Order

A sequence of values is said to be in **non-decreasing order**, if the successive element is greater than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 1, 3, 3, 6, 8, 9 are in non-decreasing order, as every next element is greater than or equal to (in case of 3) but not less than the previous one.

Bubble sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.



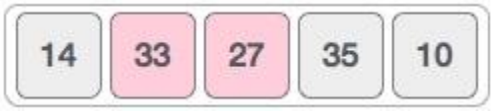
Bubble sort starts with very first two elements, comparing them to check which one is greater.



In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this –



Next we compare 33 and 35. We find that both are in already sorted positions.



Then we move to the next two values, 35 and 10.



We know then that 10 is smaller 35. Hence they are not sorted.



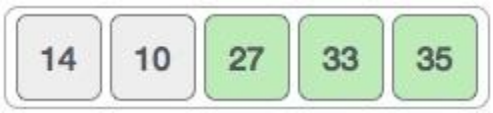
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −



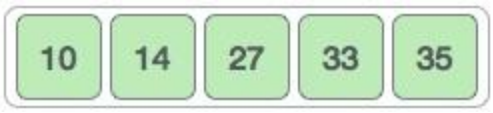
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
begin BubbleSort(list)

  for all elements of list

    if list[i] > list[i+1]

      swap(list[i], list[i+1])
```

```
    end if
  end for
  return list
end BubbleSort
```

Pseudocode

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Pseudocode of BubbleSort algorithm can be written as follows –

```
procedure bubbleSort( list : array of items )
  loop = list.count;
  for i = 0 to loop-1 do:
    swapped = false
    for j = 0 to loop-1 do:
      /* compare the adjacent elements */
      if list[j] > list[j+1] then
        /* swap them */
        swap( list[j], list[j+1] )
        swapped = true
```

```
    end if
  end for

  /*if no number was swapped that means
  array is sorted now, break the loop.*/
  if(not swapped) then
```

```
    break
  end if
 end for
end procedure return list
```

Implementation

One more issue we did not address in our original algorithm and its improvised pseudocode, is that, after every iteration the highest values settles down at the end of the array. Hence, the next iteration need not include already sorted elements. For this purpose, in our implementation, we restrict the inner loop to avoid already sorted values.

Quick Sort

Quick sort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting of an array of n elements. This algorithm follows divide and conquer approach. The algorithm processes the array in the following way.

1. Set the first index of the array to left and loc variable. Set the last index of the array to right variable. i.e. left = 0, loc = 0, en d = n - 1, where n is the length of the array.

2. Start from the right of the array and scan the complete array from right to beginning comparing each element of the array with the element pointed by loc.

Ensure that, a[loc] is less than a[right].

1. If this is the case, then continue with the comparison until right becomes equal to the loc.
2. If a[loc] > a[right], then swap the two values. And go to step 3.
3. Set, loc = right

1. start from element pointed by left and compare each element in its way with the element pointed by the variable loc. Ensure that a[loc] > a[left]
    1. if this is the case, then continue with the comparison until loc becomes equal to left.
    2. [loc] < a[right], then swap the two values and go to step 2.
    3. Set, loc = left.

Complexity

Complexity	**Best Case**	**Average Case**	**Worst Case**
Time Complexity	O(n) for 3 way partition or O(n log n) simple partition	O(n log n)	$O(n^2)$
Space Complexity			O(log n)

Algorithm

PARTITION (ARR, BEG, END, LOC)

- **Step 1**: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG =
- **Step 2**: Repeat Steps 3 to 6 while FLAG =
- **Step 3**: Repeat while ARR[LOC] <=ARR[RIGHT]
  AND LOC != RIGHT
  SET RIGHT = RIGHT - 1
  [END OF LOOP]
- **Step 4**: IF LOC = RIGHT
  SET FLAG = 1
  ELSE IF ARR[LOC] > ARR[RIGHT]
  SWAP ARR[LOC] with ARR[RIGHT]
  SET LOC = RIGHT
  [END OF IF]
- **Step 5**: IF FLAG = 0
  Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
  SET LEFT = LEFT + 1
  [END OF LOOP]
- **Step 6**:IF LOC = LEFT
  SET FLAG = 1
  ELSE IF ARR[LOC] < ARR[LEFT]
  SWAP ARR[LOC] with ARR[LEFT]
  SET LOC = LEFT
  [END OF IF]
  [END OF IF]
- **Step 7**: [END OF LOOP]
- **Step 8**: END

QUICK_SORT (ARR, BEG, END)

Quick Sort

Quick sort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting of an array of n elements. This algorithm follows divide and conquer approach. The algorithm processes the array in the following way.

1. Set the first index of the array to left and loc variable. Set the last index of the array to right variable. i.e. left = 0, loc = 0, en d = n - 1, where n is the length of the array.
2. Start from the right of the array and scan the complete array from right to beginning comparing each element of the array with the element pointed by loc.

Ensure that, a[loc] is less than a[right].

    1. If this is the case, then continue with the comparison until right becomes equal to the loc.
    2. If a[loc] > a[right], then swap the two values. And go to step 3.
    3. Set, loc = right

1. start from element pointed by left and compare each element in its way with the element pointed by the variable loc. Ensure that a[loc] > a[left]

    1. if this is the case, then continue with the comparison until loc becomes equal to left.
    2. [loc] < a[right], then swap the two values and go to step 2.
    3. Set, loc = left.

Complexity

Complexity	Best Case
Time Complexity	O(n) for 3 way partition or O(n log n) simple partition
Space Complexity	

Algorithm

PARTITION (ARR, BEG, END, LOC)

- **Step 1**: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG =
- **Step 2**: Repeat Steps 3 to 6 while FLAG =
- **Step 3**: Repeat while ARR[LOC] <=ARR[RIGHT]
  AND LOC != RIGHT
  SET RIGHT = RIGHT - 1
  [END OF LOOP]
- **Step 4**: IF LOC = RIGHT
  SET FLAG = 1
  ELSE IF ARR[LOC] > ARR[RIGHT]
  SWAP ARR[LOC] with ARR[RIGHT]
  SET LOC = RIGHT
  [END OF IF]
- **Step 5**: IF FLAG = 0
  Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT
  SET LEFT = LEFT + 1
  [END OF LOOP]

- **Step 6**:IF LOC = LEFT
  SET FLAG = 1
  ELSE IF ARR[LOC] < ARR[LEFT]
  SWAP ARR[LOC] with ARR[LEFT]
  SET LOC = LEFT
  [END OF IF]
  [END OF IF]
- **Step 7**: [END OF LOOP]
- **Step 8**: END

QUICK_SORT (ARR, BEG, END)

Heap Sort

Heap sort processes the elements by creating the min heap or max heap using the elements of the given array. Min heap or max heap represents the ordering of the array in which root element represents the minimum or maximum element of the array. At each step, the root element of the heap gets deleted and stored into the sorted array and the heap will again be heapified.

The heap sort basically recursively performs two main operations.

- Build a heap H, using the elements of ARR.
- Repeatedly delete the root element of the heap formed in phase 1.

Complexity

Complexity	Best Case	Average Case	Worst case
Time Complexity	Ω(n log (n))	θ(n log (n))	O(n log (n))

Space Complexity			O(1)

Algorithm

HEAP_SORT(ARR, N)

- **Step 1**: [Build Heap H]
  Repeat for i=0 to N-1
  CALL INSERT_HEAP(ARR, N, ARR[i])
  [END OF LOOP]
- **Step 2**: Repeatedly Delete the root element
  Repeat while N > 0
  CALL Delete_Heap(ARR,N,VAL)
  SET N = N+1
  [END OF LOOP]
- **Step 3**: END

Radix Sort

Radix sort processes the elements the same way in which the names of the students are sorted according to their alphabetical order. There are 26 radix in that case due to the fact that, there are 26 alphabets in English. In the first pass, the names are grouped according to the ascending order of the first letter of names.

In the second pass, the names are grouped according to the ascending order of the second letter. The same process continues until we find the sorted list of names. The bucket are used to store the names produced in each pass. The number of passes depends upon the length of the name with the maximum letter.

In the case of integers, radix sort sorts the numbers according to their digits. The comparisons are made among the digits of the number from LSB to MSB. The

number of passes depend upon the length of the number with the most number of digits.

Complexity

Complexity	Best Case	Average Case	Worst Case
Time Complexity	Ω(n+k)	θ(nk)	O(nk)
Space Complexity			O(n+k)

Example

Consider the array of length 6 given below. Sort the array by using Radix sort.

A = {10, 2, 901, 803, 1024}

Pass 1: (Sort the list according to the digits at 0's place)

10, 901, 2, 803, 1024.

Pass 2: (Sort the list according to the digits at 10's place)

02, 10, 901, 803, 1024

Pass 3: (Sort the list according to the digits at 100's place)

02, 10, 1024, 803, 901.

Pass 4: (Sort the list according to the digits at 1000's place)

02, 10, 803, 901, 1024

Therefore, the list generated in the step 4 is the sorted list, arranged from radix sort.

Algorithm

- Step 1:Find the largest number in ARR as LARGE
- Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
- Step 3: SET PASS =0
- Step 4: Repeat Step 5 while PASS <= NOP-1
- Step 5: SET I = 0 and INITIALIZE buckets
- Step 6:Repeat Steps 7 to 9 while I<n-1< li=""></n-1<>
- Step 7: SET DIGIT = digit at PASSth place in A[I]
- Step 8: Add A[I] to the bucket numbered DIGIT
- Step 9: INCREMENT bucket count for bucket numbered DIGIT
  [END OF LOOP]
- Step 10: Collect the numbers in the bucket
  [END OF LOOP]
- Step 11: END

Insertion Sort Algorithm

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).

Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Step by Step Process

The insertion sort algorithm is performed using the following steps...

- Step 1 - Assume that first element in the list is in sorted portion and all the remaining elements are in unsorted portion.
- Step 2: Take first element from the unsorted portion and insert that element into the sorted portion in the order specified.
- Step 3: Repeat the above process until all the elements from the unsorted portion are moved into the sorted portion.

Following is the sample code for insertion sort...

Insertion Sort Logic

```
//Insertion sort logic
  for i = 1 to size-1 {
    temp = list[i];
    j = i-1;
    while ((temp < list[j]) && (j > 0)) {
      list[j] = list[j-1];
      j = j - 1;
    }
    list[j] = temp;
  }
```

Example

Consider the following unsorted list of elements...



Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...



Move the first element 15 from unsorted portion to sorted portion of the list.



To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position



To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.



To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.



To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.



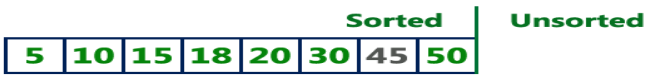
To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.



To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.



To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.



Unsorted portion of the list has became empty. So we stop the process. And the final sorted list of elements is as follows...



Complexity of the Insertion Sort Algorithm

To sort an unsorted list with 'n' number of elements, we need to make (1+2+3+.....+n-1) = (n (n-1))/2 number of comparisions in the worst case. If the list is already sorted then it requires 'n' number of comparisions.

Worst Case : $O(n^2)$
Best Case : $\Omega(n)$
Average Case : $\Theta(n^2)$

Implementaion of Insertion Sort Algorithm using C Programming Language

```
#include<stdio.h>

#include<conio.h>


void main(){


  int size, i, j, temp, list[100];


  printf("Enter the size of the list: ");

  scanf("%d", &size);


  printf("Enter %d integer values: ", size);

  for (i = 0; i < size; i++)

    scanf("%d", &list[i]);

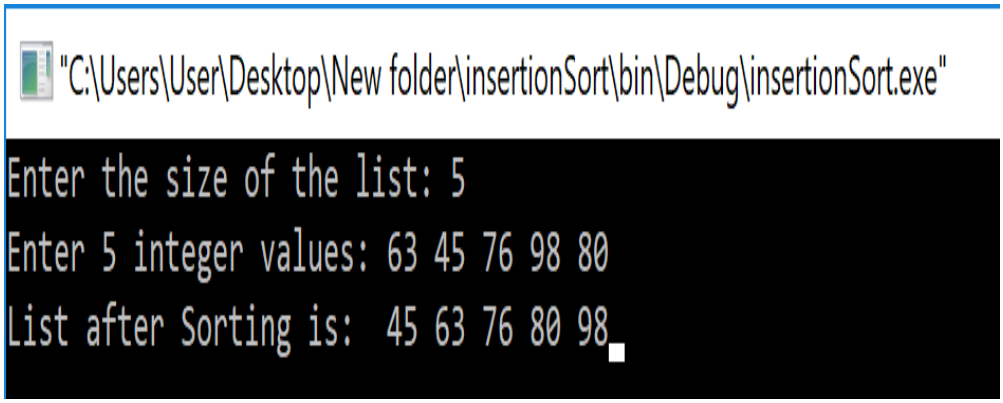

  //Insertion sort logic
```

```
  for (i = 1; i < size; i++) {
     temp = list[i];
     j = i - 1;
     while ((temp < list[j]) && (j >= 0)) {
        list[j + 1] = list[j];
        j = j - 1;
     }
     list[j + 1] = temp;
  }

  printf("List after Sorting is: ");
  for (i = 0; i < size; i++)
     printf(" %d", list[i]);

 getch();
}
```

Output



Selection Sort Algorithm

Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with all the remaining elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped so that first position is filled with the smallest element in the sorted order. Next, we select the element at a second position in the list and it is compared with all the remaining elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated until the entire list is sorted.

Step by Step Process

The selection sort algorithm is performed using the following steps...

- Step 1 - Select the first element of the list (i.e., Element at first position in the list).
- Step 2: Compare the selected element with all the other elements in the list.
- Step 3: In every comparision, if any element is found smaller than the selected element (for Ascending order), then both are swapped.
- Step 4: Repeat the same procedure with element in the next position in the list till the entire list is sorted.

Following is the sample code for selection sort...

Selection Sort Logic

```
//Selection sort logic


 for(i=0; i<size; i++){
    for(j=i+1; j<size; j++){
        if(list[i] > list[j])
                    {
           temp=list[i];
           list[i]=list[j];
           list[j]=temp;
        }
    }
 }
```

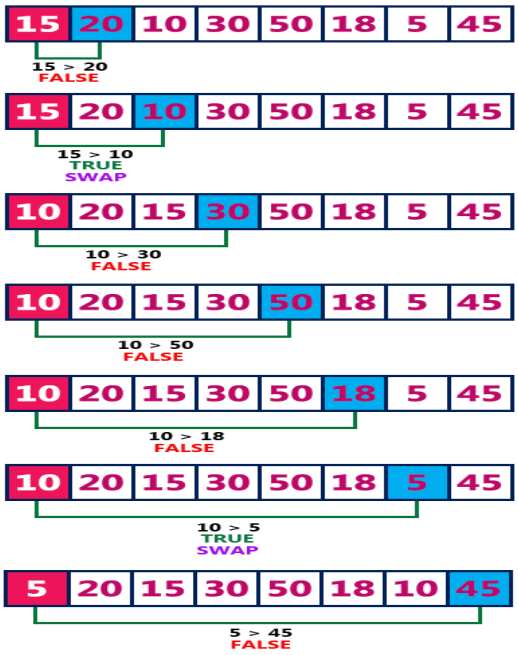
Example

Consider the following unsorted list of elements...



Iteration #1

Select the first position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.



List after 1st iteration



Iteration #2

Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 2nd iteration



Iteration #3

Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 3rd iteration



Iteration #4

Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 4th iteration



Iteration #5

Select the fifth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 5th iteration



Iteration #6

Select the sixth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 6th iteration



Iteration #7

Select the seventh position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 7th iteration



Complexity of the Selection Sort Algorithm

To sort an unsorted list with 'n' number of elements, we need to make ((n-1)+(n-2)+(n-3)+.....+1) = (n (n-1))/2 number of comparisions in the worst case. If the list is already sorted then it requires 'n' number of comparisions.

Worst Case : $O(n^2)$
Best Case : $\Omega(n^2)$
Average Case : $\Theta(n^2)$

Implementaion of Selection Sort Algorithm using C Programming Language

```
#include<stdio.h>

#include<conio.h>


void main(){


 int size,i,j,temp,list[100];

 clrscr();


 printf("Enter the size of the List: ");

 scanf("%d",&size);


 printf("Enter %d integer values: ",size);

 for(i=0; i<size; i++)

    scanf("%d",&list[i]);
```

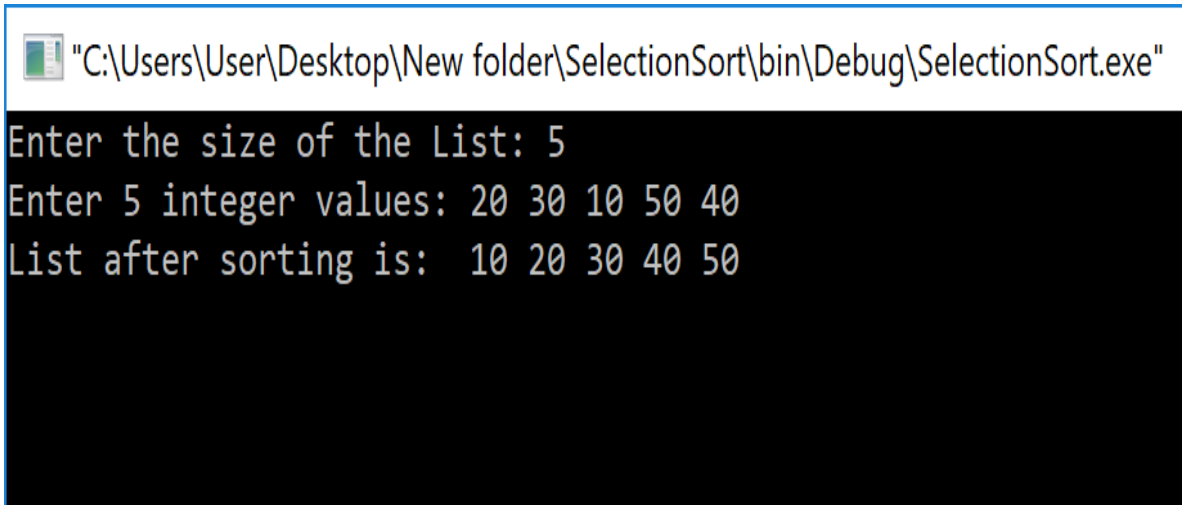
```
//Selection sort logic


for(i=0; i<size; i++){
   for(j=i+1; j<size; j++){
       if(list[i] > list[j])
                   {
          temp=list[i];
          list[i]=list[j];
          list[j]=temp;
       }
   }
}

printf("List after sorting is: ");
for(i=0; i<size; i++)
   printf(" %d",list[i]);

getch();
}
```

Output



Divide and Conquer

If we can break a single big problem into smaller sub-problems, solve the smaller sub-problems and combine their solutions to find the solution for the original big problem, it becomes easier to solve the whole problem.

an example, **Divide and Rule**.

When Britishers came to India, they saw a country with different religions living in harmony, hard working but naive citizens, unity in diversity, and found it difficult to establish their empire. So, they adopted the policy of **Divide and Rule**. Where the population of India was collectively a one big problem for them, they divided the problem into smaller problems, by instigating rivalries between local kings, making them stand against each other, and this worked very well for them.

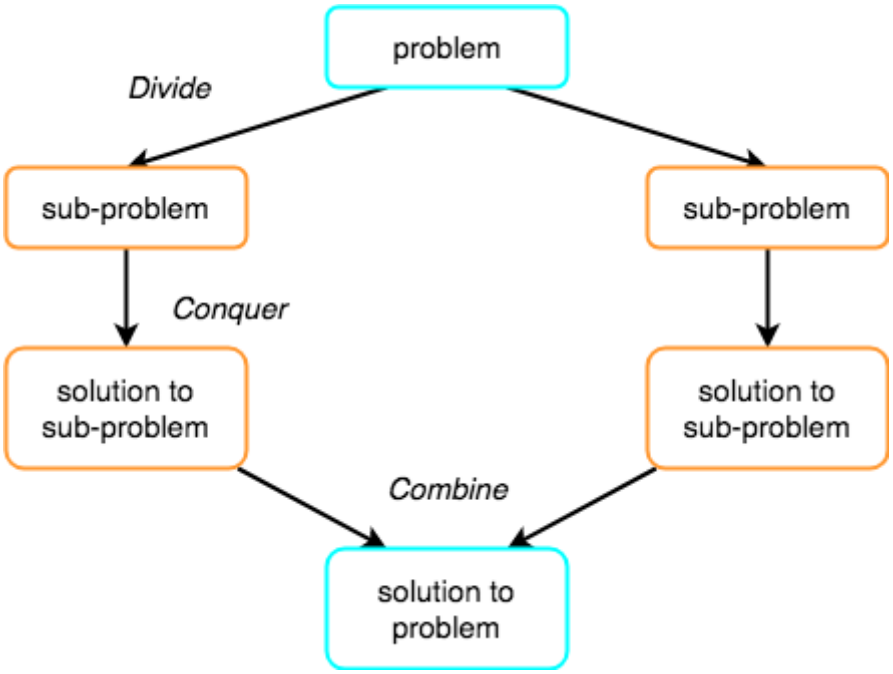
Well that was history, and a socio-political policy (**Divide and Rule**), but the idea here is, if we can somehow divide a problem into smaller sub-problems, it becomes easier to eventually solve the whole problem.

In **Merge Sort**, the given unsorted array with n elements, is divided into n subarrays, each having **one** element, because a single element is always

sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.

The concept of Divide and Conquer involves three steps:

1. **Divide** the problem into multiple small problems.
2. **Conquer** the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
3. **Combine** the solutions of the subproblems to find the solution of the actual problem.



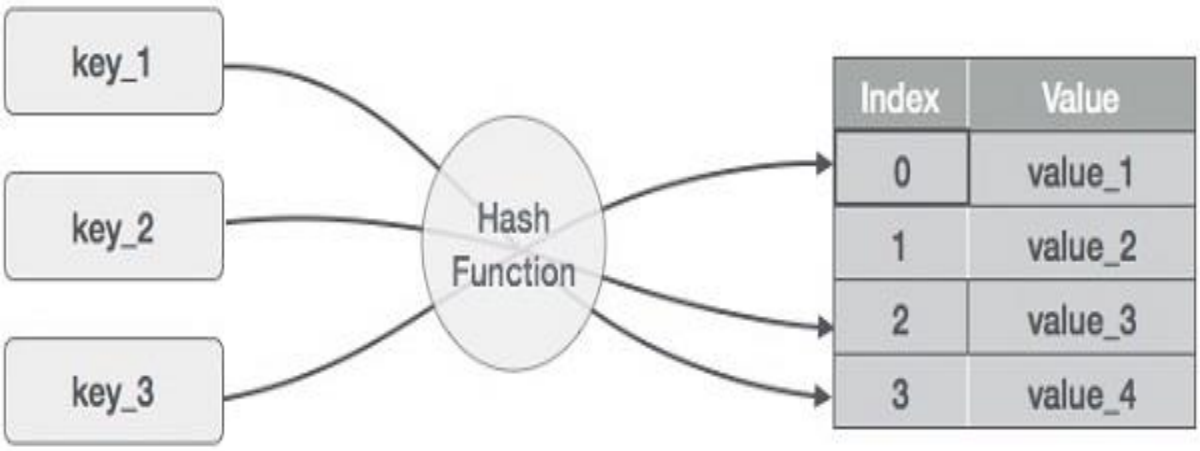
Hashing

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)

- (37,98)

Sr.No.	Key	Hash
1	1	1 % 20 = 1
2	2	2 % 20 = 2
3	42	42 % 20 = 2
4	4	4 % 20 = 4
5	12	12 % 20 = 12
6	14	14 % 20 = 14
7	17	17 % 20 = 17
8	13	13 % 20 = 13
9	37	37 % 20 = 17

Linear Probing

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Basic Operations

Following are the basic primary operations of a hash table.

- **Search** − Searches an element in a hash table.
- **Insert** − inserts an element in a hash table.

- **delete** − Deletes an element from a hash table.

DataItem

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

```
struct DataItem {
  int data;
  int key;
};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){

  return key % SIZE;

}
```

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

Example

```
struct DataItem *search(int key) {
  //get the hash
  int hashIndex = hashCode(key);
  //move in array until an empty
```

```
  while(hashArray[hashIndex] != NULL) {
    if(hashArray[hashIndex]->key == key)
      return hashArray[hashIndex];
    //go to next cell
    ++hashIndex;
    //wrap around the table
    hashIndex %= SIZE;
  }
  return NULL;
}
```

Insert Operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

Example

```
void insert(int key,int data) {
  struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
  item->data = data;
  item->key = key;
  //get the hash
  int hashIndex = hashCode(key);
  //move in array until an empty or deleted cell
  while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {
    //go to next cell
    ++hashIndex;
    //wrap around the table
    hashIndex %= SIZE;
  }
  hashArray[hashIndex] = item;
}
```

Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

Example

```
struct DataItem* delete(struct DataItem* item) {
  int key = item->key;
  //get the hash
  int hashIndex = hashCode(key)
  //move in array until an empty
  while(hashArray[hashIndex] !=NULL) {
    if(hashArray[hashIndex]->key == key) {
      struct DataItem* temp = hashArray[hashIndex];
      //assign a dummy item at deleted position
      hashArray[hashIndex] = dummyItem;
      return temp;
    }
    //go to next cell
    ++hashIndex;
    //wrap around the table
    hashIndex %= SIZE;
  }
  return NULL;
}
```

Performance Analysis

What is Performance Analysis of an algorithm?

If we want to go from city "A" to city "B", there can be many ways of doing this. We can go by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science, there are multiple algorithms to solve a problem. When we

have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.

When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements. The formal definition is as follows...

Performance of an algorithm is a process of making evaluative judgement about algorithms.

It can also be defined as follows...

Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.

That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem.

We compare algorithms with each other which are solving the same problem, to select the best algorithm. To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, the execution speed of that algorithm, easy to understand, easy to implement, etc.,

Generally, the performance of an algorithm depends on the following elements...

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.,

When we want to analyse an algorithm, we consider only the space and time required by that particular algorithm and we ignore all the remaining elements.

Based on this information, performance analysis of an algorithm can also be defined as follows...

Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.

Performance analysis of an algorithm is performed by using the following measures...

1. Space required to complete the task of that algorithm (**Space Complexity**). It includes program space and data space
2. Time required to complete the task of that algorithm (**Time Complexity**)

Recurrences

The recursive nature of D&C leads to recurrences, or functions defined in terms of:

- one or more base cases, and
- itself, with smaller arguments.

Solving Recurrence Equations

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. Recurrences are generally used in divide-and-conquer paradigm.

consider **T(n)** to be the running time on a problem of size **n**.

If the problem size is small enough, say **n < c** where **c** is a constant, the straightforward solution takes constant time, which is written as **θ(1)**. If the division of the problem yields a number of sub-problems with size nbnb.

To solve the problem, the required time is **a.T(n/b)**. If we consider the time required for division is **D(n)** and the time required for combining the results of sub-problems is **C(n)**, the recurrence relation can be represented as −

T(n)={θ(1)aT(nb)+D(n)+C(n)ifn⩽cotherwiseT(n)={θ(1)ifn⩽caT(nb)+D(n)+C(n)otherwise

A recurrence relation can be solved using the following methods −

- **Substitution Method** − In this method, we guess a bound and using mathematical induction we prove that our assumption was correct.
- **Recursion Tree Method** − In this method, a recurrence tree is formed where each node represents the cost.
- **Master's Theorem** − This is another important technique to find the complexity of a recurrence relation.

Time Complexity of Algorithms

For any defined problem, there can be N number of solution. This is true in general. If I have a problem and I discuss about the problem with all of my friends, they will all suggest me different solutions. And I am the one who has to decide which solution is the best based on the circumstances.

Similarly for any problem which must be solved using a program, there can be infinite number of solutions. Let's take a simple example to understand this. Below we have two different algorithms to find square of a number(for some time, forget that square of any number n is n*n):

One solution to this problem can be, running a loop for n times, starting with the number n and adding n to it, every time.

```
/*
   we have to calculate the square of n
*/
for i=1 to n
   do n = n + n
// when the loop ends n will hold its square
return n
```

Or, we can simply use a mathematical operator * to find the square.

```
/*
   we have to calculate the square of n
*/
return n*n
```

In the above two simple algorithms, you saw how a single problem can have many solutions. While the first solution required a loop which will execute for n number of times, the second solution used a mathematical operator * to return the result in one line. So which one is the better approach, of course the second one.

What is Time Complexity?

Time complexity of an algorithm signifies the total time required by the program to run till its completion.

The time complexity of algorithms is most commonly expressed using the **big O notation**. It's an asymptotic notation to represent the time complexity. We will study about it in detail in the next tutorial.

Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution. Like in the example above, for the first code the loop will run n number of times, so the time complexity will be n atleast and as the value of n will increase the time taken will also increase. While for the second code, time complexity is constant, because it will never be dependent on the value of n, it will always give the result in 1 step.

And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

Calculating Time Complexity

Now lets tap onto the next big topic related to Time complexity, which is How to Calculate Time Complexity. It becomes very confusing some times, but we will try to explain it in the simplest way.

Now the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity. In general you can think of it like this :

statement;

Above we have a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to N.

```
for(i=0; i < N; i++)
{
   statement;
}
```

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for(i=0; i < N; i++)
{
   for(j=0; j < N;j++)
   {
   statement;
   }
}
```

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

```
while(low <= high)
{
   mid = (low + high) / 2;
   if (target < list[mid])
      high = mid - 1;
   else if (target > list[mid])
      low = mid + 1;
   else break;
}
```

This is an algorithm to break a set of numbers into halves, to search a particular field(we will study this in detail later). Now, this algorithm will have a **Logarithmic** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here). This is because the algorithm divides the working area in half with each iteration.

```
void quicksort(int list[], int left, int right)
{
   int pivot = partition(list, left, right);
   quicksort(list, left, pivot - 1);
   quicksort(list, pivot + 1, right);
}
```

Taking the previous algorithm forward, above we have a small logic of Quick Sort(we will study this in detail later). Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times(where N is the size of list). Hence time complexity will be **N*log(N)**. The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

NOTE: In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic.

Types of Notations for Time Complexity

Now we will discuss and understand the various notations used for Time Complexity.

1. **Big Oh** denotes "fewer than or the same as" <expression> iterations.
2. **Big Omega** denotes "more than or the same as" <expression> iterations.
3. **Big Theta** denotes "the same as" <expression> iterations.
4. **Little Oh** denotes "fewer than" <expression> iterations.
5. **Little Omega** denotes "more than" <expression> iterations.

Understanding Notations of Time Complexity with Example

O(expression) is the set of functions that grow slower than or at the same rate as expression. It indicates the maximum required by an algorithm for all input values. It represents the worst case of an algorithm's time complexity.

Omega(expression) is the set of functions that grow faster than or at the same rate as expression. It indicates the minimum time required by an algorithm for all input values. It represents the best case of an algorithm's time complexity.

Theta(expression) consist of all the functions that lie in both O(expression) and Omega(expression). It indicates the average bound of an algorithm. It represents the average case of an algorithm's time complexity.

Suppose you've calculated that an algorithm takes f(n) operations, where,

```
f(n) = 3*n^2 + 2*n + 4.   // n^2 means square of n
```

Since this polynomial grows at the same rate as $\mathbf{n^2}$, then you could say that the function **f** lies in the set **Theta(n^2)**. (It also lies in the sets **O(n^2)** and **Omega(n^2)** for the same reason.)

The simplest explanation is, because **Theta** denotes the same as the expression. Hence, as **f(n)** grows by a factor of $\mathbf{n^2}$, the time complexity can be best represented as **Theta(n^2)**.

Space Complexity of Algorithms

Whenever a solution to a problem is written some memory is required to complete. For any algorithm memory may be used for the following:

1. Variables (This include the constant values, temporary values)
2. Program Instruction
3. Execution

Space complexity is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.

Sometime **Auxiliary Space** is confused with Space Complexity. But Auxiliary Space is the extra space or the temporary space used by the algorithm during it's execution.

Space Complexity = **Auxiliary Space + Input space**

Memory Usage while Execution

While executing, algorithm uses memory space for three reasons:

1. **Instruction Space**

It's the amount of memory used to save the compiled version of instructions.

2. **Environmental Stack**

Sometimes an algorithm(function) may be called inside another algorithm(function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm(function) is made.

For example, If a function A() calls function B() inside it, then all th variables of the function A() will get stored on the system stack temporarily, while the function B() is called and executed inside the funciton A().

3. **Data Space**

Amount of space used by the variables and constants.

But while calculating the **Space Complexity** of any algorithm, we usually consider only **Data Space** and we neglect the **Instruction Space** and **Environmental Stack**.

Calculating the Space Complexity

For calculating the space complexity, we need to know the value of memory used by different type of datatype variables, which generally varies for different operating systems, but the method for calculating the space complexity remains the same.

Type	**Size**
bool, char, unsigned char, signed char, __int8	1 byte
__int16, short, unsigned short, wchar_t, __wchar_t	2 bytes
float, __int32, int, unsigned int, long, unsigned long	4 bytes
double, __int64, long double, long long	8 bytes

how to compute space complexity by taking a few examples:

```
{
   int z = a + b + c;
   return(z);
}
```

In the above expression, variables a, b, c and z are all integer types, hence they will take up 4 bytes each, so total memory requirement will be (4(4) + 4) = 20 bytes, this additional 4 bytes is for **return value**. And because this space requirement is fixed for the above example, hence it is called **Constant Space Complexity**.

another example, this time a bit complex one,

```
// n is the length of array a[]
int sum(int a[], int n)
{
        int x = 0;              // 4 bytes for x
        for(int i = 0; i < n; i++)      // 4 bytes for i
        {
           x  = x + a[i];
        }
        return(x);
}
```

- In the above code, 4*n bytes of space is required for the array a[] elements.
- 4 bytes each for x, n, i and the return value.

Hence the total memory requirement will be (4n + 12), which is increasing linearly with the increase in the input value n, hence it is called as **Linear Space Complexity.**

Similarly, we can have quadratic and other complex space complexity as well, as the complexity of an algorithm increases.

But we should always focus on writing algorithm code in such a way that we keep the space complexity **minimum**.

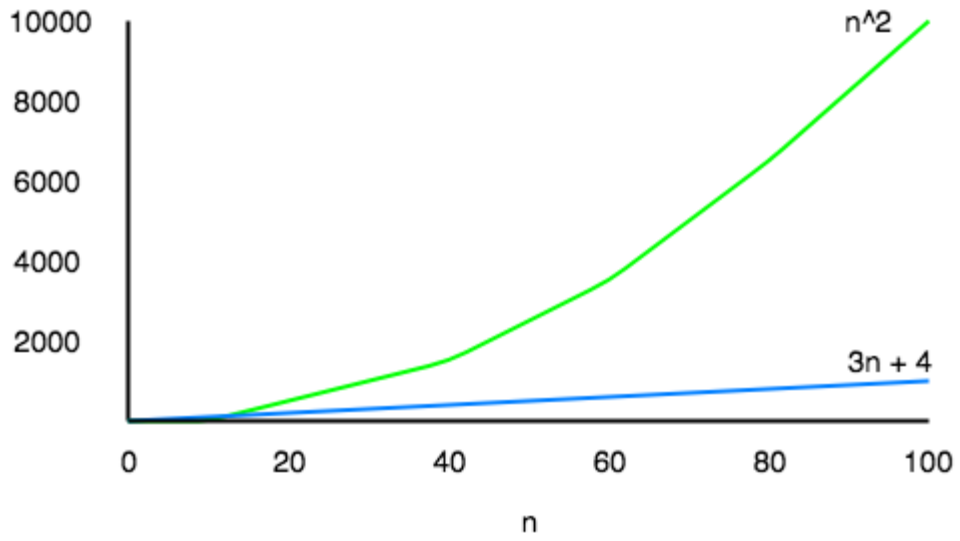
Asymptotic Notations

When it comes to analysing the complexity of any algorithm in terms of time and space, we can never provide an exact number to define the time required and the

space required by the algorithm, instead we express it using some standard notations, also known as **Asymptotic Notations**.

When we analyse any algorithm, we generally get a formula to represent the amount of time required for execution or the time required by the computer to run the lines of code of the algorithm, number of memory accesses, number of comparisons, temporary variables occupying memory space etc. This formula often contains unimportant details that don't really tell us anything about the running time.

an example, if some algorithm has a time complexity of $T(n) = (n^2 + 3n + 4)$, which is a quadratic equation. For large values of n, the 3n + 4 part will become insignificant compared to the n^2 part.



For n = 1000, n^2 will be 1000000 while 3n + 4 will be 3004.

Also, When we compare the execution times of two algorithms the constant coefficients of higher order terms are also neglected.

An algorithm that takes a time of $200n^2$ will be faster than some other algorithm that takes n^3 time, for any value of n larger than 200. Since we're only interested in the asymptotic behavior of the growth of the function, the constant factor can be ignored too.

What is Asymptotic Behaviour

The word **Asymptotic** means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken).

Remember studying about **Limits** in High School, this is the same.

The only difference being, here we do not have to find the value of any expression where n is approaching any finite number or infinity, but in case of Asymptotic notations, we use the same model to ignore the constant factors and insignificant parts of an expression, to device a better way of representing complexities of algorithms, in a single coefficient, so that comparison between algorithms can be done easily.

an example to understand this:

If we have two algorithms with the following expressions representing the time required by them for execution, then:

Expression 1: $(20n^2 + 3n - 4)$

Expression 2: $(n^3 + 100n - 2)$

Now, as per asymptotic notations, we should just worry about how the function will grow as the value of n(input) will grow, and that will entirely depend on n^2 for the Expression 1, and on n^3 for Expression 2. Hence, we can clearly say that the algorithm for which running time is represented by the Expression 2, will grow faster than the other one, simply by analysing the highest power coeeficient and ignoring the other constants(20 in $20n^2$) and insignificant parts of the expression(3n - 4 and 100n - 2).

The main idea behind casting aside the less important part is to make things **manageable**.

All we need to do is, first analyse the algorithm to find out an expression to define it's time requirements and then analyse how that expression will grow as the input(n) will grow.

Types of Asymptotic Notations

We use three types of asymptotic notations to represent the growth of any algorithm, as input increases:

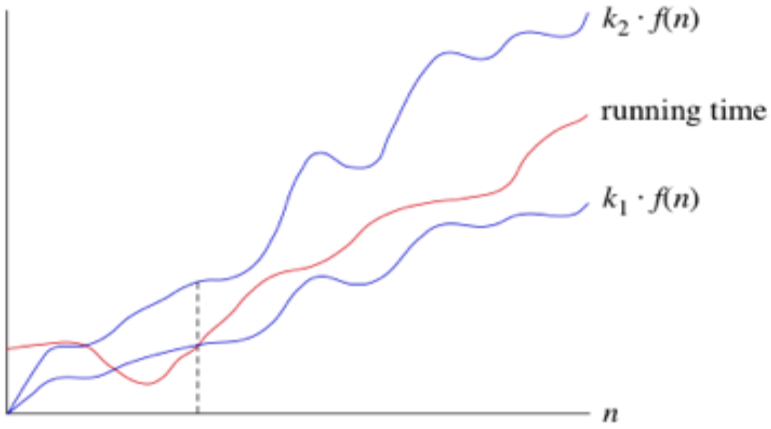
1. Big Theta (Θ)
2. Big Oh(O)
3. Big Omega (Ω)

Tight Bounds: Theta

When we say tight bounds, we mean that the time compexity represented by the Big-Θ notation is like the average value or range within which the actual time of execution of the algorithm will be.

For example, if for some algorithm the time complexity is represented by the expression $3n^2 + 5n$, and we use the Big-Θ notation to represent this, then the time complexity would be $\Theta(n^2)$, ignoring the constant coefficient and removing the insignificant part, which is 5n.

Here, in the example above, complexity of $\Theta(n^2)$ means, that the avaerage time for any input n will remain in between, $k1 * n^2$ and $k2 * n^2$, where k1, k2 are two constants, therby tightly binding the expression rpresenting the growth of the algorithm.



Upper Bounds: Big-O

This notation is known as the upper bound of the algorithm, or a Worst Case of an algorithm.

It tells us that a certain function will never exceed a specified time for any value of input n.

The question is why we need this representation when we already have the big-Θ notation, which represents the tightly bound running time for any algorithm. Let's take a small example to understand this.

Consider Linear Search algorithm, in which we traverse an array elements, one by one to search a given number.

In **Worst case**, starting from the front of the array, we find the element or number we are searching for at the end, which will lead to a time complexity of n, where n represents the number of total elements.

But it can happen, that the element that we are searching for is the first element of the array, in which case the time complexity will be 1.

Now in this case, saying that the big-Θ or tight bound time complexity for Linear search is Θ(n), will mean that the time required will always be related to n, as this is the right way to represent the average time complexity, but when we use the big-O notation, we mean to say that the time complexity is O(n), which means

that the time complexity will never exceed n, defining the upper bound, hence saying that it can be less than or equal to n, which is the correct representation.

This is the reason, most of the time you will see Big-O notation being used to represent the time complexity of any algorithm, because it makes more sense.

Lower Bounds: Omega

Big Omega notation is used to define the **lower bound** of any algorithm or we can say **the best case** of any algorithm.

This always indicates the minimum time required for any algorithm for all input values, therefore the best case of any algorithm.

In simple words, when we represent a time complexity for any algorithm in the form of big-Ω, we mean that the algorithm will take atleast this much time to cmplete it's execution. It can definitely take more time than this too.

Different types of recurrence relations and their solutions

Type 1: Divide and conquer recurrence relations –
Following are some of the examples of recurrence relations based on divide and conquer.

$T(n) = 2T(n/2) + cn$

$T(n) = 2T(n/2) + \sqrt{n}$

These types of recurrence relations can be easily solved using **Master Method**. For recurrence relation $T(n) = 2T(n/2) + cn$, the values of a = 2, b = 2 and k =1. Here $\log_b(a) = \log_2(2) = 1 = k$. Therefore, the complexity will be $\Theta(n\log_2(n))$. Similarly for recurrence relation $T(n) = 2T(n/2) + \sqrt{n}$, the values of a = 2, b = 2 and k =1/2. Here $\log_b(a) = \log_2(2) = 1 > k$. Therefore, the complexity will be $\Theta(n)$.

Type 2: Linear recurrence relations –
Following are some of the examples of recurrence relations based on linear recurrence relation.

T(n) = T(n-1) + n for n>0 and T(0) = 1

These types of recurrence relations can be easily solved using substitution method.
For example,

T(n) = T(n-1) + n

= T(n-2) + (n-1) + n

= T(n-k) + (n-(k-1))….. (n-1) + n

Substituting k = n, we get

T(n) = T(0) + 1 + 2+….. +n = n(n+1)/2 = O(n^2)

Type 3: Value substitution before solving –
Sometimes, recurrence relations can't be directly solved using techniques like substitution, recurrence tree or master method. Therefore, we need to convert the recurrence relation into appropriate form before solving. For example,

T(n) = T(√n) + 1

To solve this type of recurrence, substitute n = 2^m as:

T(2^m) = T(2^m /2) + 1

Let T(2^m) = S(m),

S(m) = S(m/2) + 1

Solving by master method, we get

S(m) = Θ(logm)

As n = 2^m or m = log2(n),

T(n) = T(2^m) = S(m) = Θ(logm) = Θ(loglogn)

some questions based on the approaches discussed.

Que – 1. What is the time complexity of Tower of Hanoi problem?
(A) T(n) = O(sqrt(n))
(D) T(n) = O(n^2)
(C) T(n) = O(2^n)
(D) None

Solution: For Tower of Hanoi, T(n) = 2T(n-1) + c for n>1 and T(1) = 1. Solving this,

T(n) = 2T(n-1) + c

= 2(2T(n-2)+ c) + c = 2^2*T(n-2) + (c + 2c)

= 2^k*T(n-k) + (c + 2c + .. kc)

Substituting k = (n-1), we get

T(n) = 2^(n-1)*T(1) + (c + 2c + (n-1)c) = O(2^n)

Que – 2. Consider the following recurrence:
T(n) = 2 * T(ceil (sqrt(n))) + 1, T(1) = 1
Which one of the following is true?
(A) T(n) = (loglogn)
(B) T(n) = (logn)
(C) T(n) = (sqrt(n))
(D) T(n) = (n)

Solution: To solve this type of recurrence, substitute n = 2^m as:

T(2^m) = 2T(2^m /2) + 1

Let T(2^m) = S(m),

S(m) = 2S(m/2) + 1

Solving by master method, we get

S(m) = Θ(m)

As n = 2^m or m = log2n,

T(n) = T(2^m) = S(m) = Θ(m) = Θ(logn)

Algorithm Design Techniques

The following is a list of several popular design approaches:

1. Divide and Conquer Approach: It is a top-down approach. The algorithms which follow the divide & conquer techniques involve three steps:

- Divide the original problem into a set of subproblems.
- Solve every subproblem individually, recursively.
- Combine the solution of the subproblems (top level) into a solution of the whole original problem.

2. Greedy Technique: Greedy method is used to solve the optimization problem. An optimization problem is one in which we are given a set of input values, which are required either to be maximized or minimized (known as objective), i.e. some constraints or conditions.

- Greedy Algorithm always makes the choice (greedy criteria) looks best at the moment, to optimize a given objective.
- The greedy algorithm doesn't always guarantee the optimal solution however it generally produces a solution that is very close in value to the optimal.

3. Dynamic Programming: Dynamic Programming is a bottom-up approach we solve all possible small problems and then combine them to obtain solutions for bigger problems.

This is particularly helpful when the number of copying subproblems is exponentially large. Dynamic Programming is frequently related to **Optimization Problems**.

4. Branch and Bound: In Branch & Bound algorithm a given subproblem, which cannot be bounded, has to be divided into at least two new restricted subproblems. Branch and Bound algorithm are methods for global optimization in non-convex problems. Branch and Bound algorithms can be slow, however in the worst case they require effort that grows exponentially with problem size, but in some cases we are lucky, and the method coverage with much less effort.

5. Randomized Algorithms: A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation.

6. Backtracking Algorithm: Backtracking Algorithm tries each possibility until they find the right one. It is a depth-first search of the set of possible solution. During the search, if an alternative doesn't work, then backtrack to the choice point, the place which presented different alternatives, and tries the next alternative.

7. Randomized Algorithm: A randomized algorithm uses a random number at least once during the computation make a decision.

Example 1: In Quick Sort, using a random number to choose a pivot.

Example 2: Trying to factor a large number by choosing a random number as possible divisors.

Loop invariants

This is a justification technique. We use loop invariant that helps us to understand why an algorithm is correct. To prove statement S about a loop is correct, define S concerning series of smaller statement S_0 S_1....S_k where,

- The initial claim so is true before the loop begins.
- If S_{i-1} is true before iteration i begin, then one can show that S_i will be true after iteration i is over.

- The final statement S_k implies the statement S that we wish to justify as being true.

Divide and Conquer Approach

Many algorithms are recursive in nature to solve a given problem recursively dealing with sub-problems.

In **divide and conquer approach**, a problem is divided into smaller problems, then the smaller problems are solved independently, and finally the solutions of smaller problems are combined into a solution for the large problem.

Generally, divide-and-conquer algorithms have three parts −

- **Divide the problem** into a number of sub-problems that are smaller instances of the same problem.
- **Conquer the sub-problems** by solving them recursively. If they are small enough, solve the sub-problems as base cases.
- **Combine the solutions** to the sub-problems into the solution for the original problem.

Pros and cons of Divide and Conquer Approach

Divide and conquer approach supports parallelism as sub-problems are independent. Hence, an algorithm, which is designed using this technique, can run on the multiprocessor system or in different machines simultaneously.

In this approach, most of the algorithms are designed using recursion, hence memory management is very high. For recursive function stack is used, where function state needs to be stored.

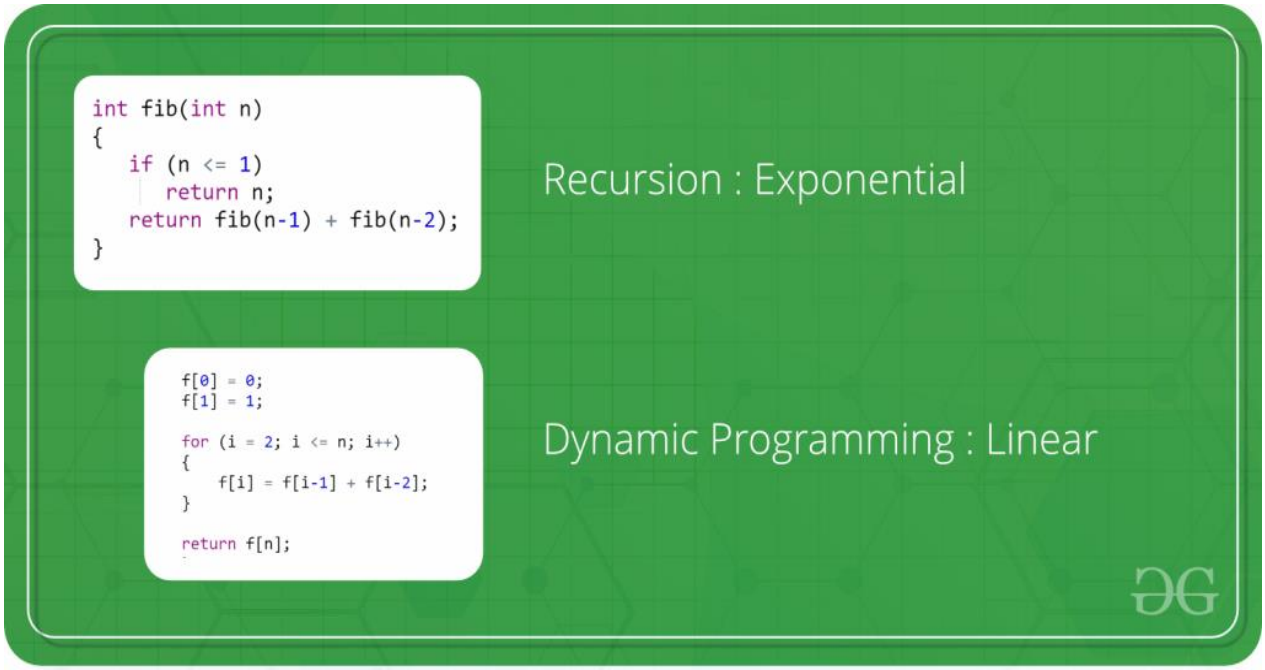
Application of Divide and Conquer Approach

Following are some problems, which are solved using divide and conquer approach.

- Finding the maximum and minimum of a sequence of numbers
- Strassen's matrix multiplication
- Merge sort
- Binary search

Dynamic Programming

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial. For example, if we write simple recursive solution for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.



Greedy Algorithms

An algorithm is designed to achieve optimum solution for a given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.

Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

Counting Coins

This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of ₹ 1, 2, 5 and 10 and we are asked to count ₹ 18 then the greedy procedure will be −

- **1** − Select one ₹ 10 coin, the remaining count is 8
- **2** − Then select one ₹ 5 coin, the remaining count is 3
- **3** − Then select one ₹ 2 coin, the remaining count is 1
- **4** − And finally, the selection of one ₹ 1 coins solves the problem

Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.

For the currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins than necessary. For example, the greedy approach will use 10 + 1 + 1 + 1 + 1 + 1, total 6 coins. Whereas the same problem could be solved by using only 3 coins (7 + 7 + 1)

Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

Examples

Most networking algorithms use the greedy approach. Here is a list of few of them –

- Travelling Salesman Problem
- Prim's Minimal Spanning Tree Algorithm
- Kruskal's Minimal Spanning Tree Algorithm
- Dijkstra's Minimal Spanning Tree Algorithm
- Graph - Map Coloring
- Graph - Vertex Cover
- Knapsack Problem
- Job Scheduling Problem

There are lots of similar problems that uses the greedy approach to find an optimum solution.

History of Greedy Algorithms

Here is an important landmark of greedy algorithms:

- Greedy algorithms were conceptualized for many graph walk algorithms in the 1950s.
- Esdger Djikstra conceptualized the algorithm to generate minimal spanning trees. He aimed to shorten the span of routes within the Dutch capital, Amsterdam.

- In the same decade, Prim and Kruskal achieved optimization strategies that were based on minimizing path costs along weighed routes.
- In the '70s, American researchers, Cormen, Rivest, and Stein proposed a recursive substructuring of greedy solutions in their classical introduction to algorithms book.
- The greedy paradigm was registered as a different type of optimization strategy in the NIST records in 2005.
- Till date, protocols that run the web, such as the open-shortest-path-first (OSPF) and many other network packet switching protocols use the greedy strategy to minimize time spent on a network.

Greedy Strategies and Decisions

Logic in its easiest form was boiled down to "greedy" or "not greedy". These statements were defined by the approach taken to advance in each algorithm stage.

For example, Djikstra's algorithm utilized a stepwise greedy strategy identifying hosts on the Internet by calculating a cost function. The value returned by the cost function determined whether the next path is "greedy" or "non-greedy".

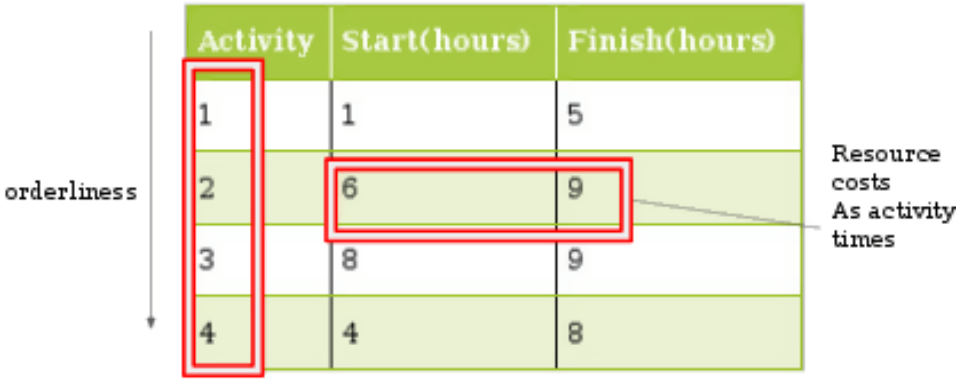
In short, an algorithm ceases to be greedy if at any stage it takes a step that is not locally greedy. The problem halts with no further scope of greed.

Characteristics of the Greedy Approach

The important characteristics of a greedy method are:

- There is an ordered list of resources, with costs or value attributions. These quantify constraints on a system.
- You will take the maximum quantity of resources in the time a constraint applies.

- For example, in an activity scheduling problem, the resource costs are in hours, and the activities need to be performed in serial order.



Why use the Greedy Approach?

Here are the reasons for using the greedy approach:

- The greedy approach has a few tradeoffs, which may make it suitable for optimization.
- One prominent reason is to achieve the most feasible solution immediately. In the activity selection problem (Explained below), if more activities can be done before finishing the current activity, these activities can be performed within the same time.
- Another reason is to divide a problem recursively based on a condition, with no need to combine all the solutions.
- In the activity selection problem, the "recursive division" step is achieved by scanning a list of items **only once** and considering certain activities.

How to Solve the activity selection problem

In the activity scheduling example, there is a "start" and "finish" time for every activity. Each Activity is indexed by a number for reference. There are two activity categories.

1. **considered activity**: is the Activity, which is the reference from which the ability to do more than one remaining Activity is analyzed.
2. **remaining activities:** activities at one or more indexes ahead of the considered activity.

The total duration gives the cost of performing the activity. That is (finish - start) gives us the durational as the **cost** of an activity.

You will learn that the **greedy extent** is the number of remaining activities you can perform in the time of a considered activity.

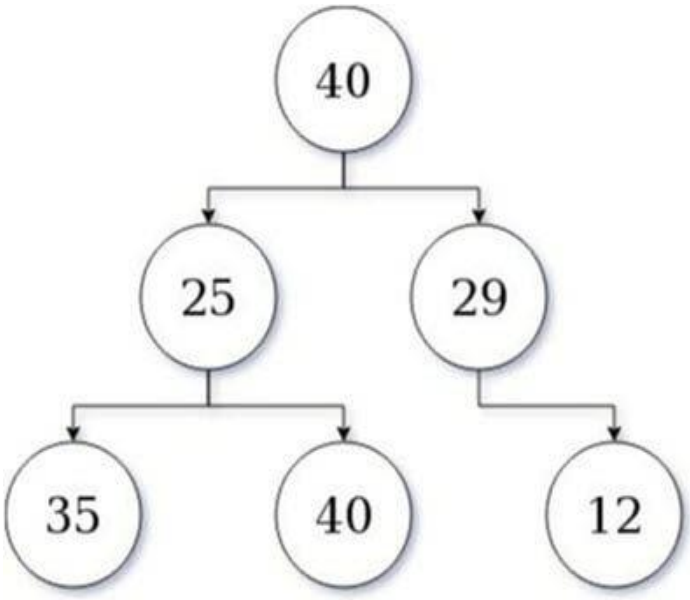
Disadvantages of Greedy Algorithms

It is not suitable for problems where a solution is required for every subproblem like sorting.

In such problems, the greedy strategy can be wrong; in the worst case even lead to a non-optimal solution.

Therefore the disadvantage of greedy algorithms is using not knowing what lies ahead of the current greedy state.

Below is a depiction of the disadvantage of the greedy approach.



In the greedy scan shown here as a tree (higher value higher greed), an algorithm state at value: 40, is likely to take 29 as the next value. Further, its quest ends at 12. This amounts to a value of 41.

However, if the algorithm took a sub-optimal path or adopted a conquering strategy. then 25 would be followed by 40, and the overall cost improvement would be 65, which is valued 24 points higher as a suboptimal decision.

Backtracking

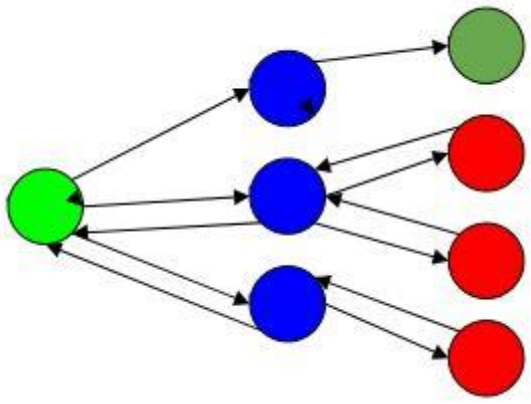
Backtracking is a technique based on algorithm to solve problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that doesn't give rise to the solution of the problem based on the constraints given to solve the problem.

Backtracking algorithm is applied to some specific types of problems,

- Decision problem used to find a feasible solution of the problem.
- Optimisation problem used to find the best solution that can be applied.
- Enumeration problem used to find the set of all feasible solutions of the problem.

In backtracking problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints from where the problem can backtrack if no feasible solution is found for the problem.

Example,



Here,

Green is the start point, blue is the intermediate point, red are points with no feasible solution, dark green is end solution.

Here, when the algorithm propagates to an end to check if it is a solution or not, if it is then returns the solution otherwise backtracks to the point one step behind it to find track to the next point to find solution.

Algorithm

Step 1 – if current_position is goal, return success

Step 2 – else,

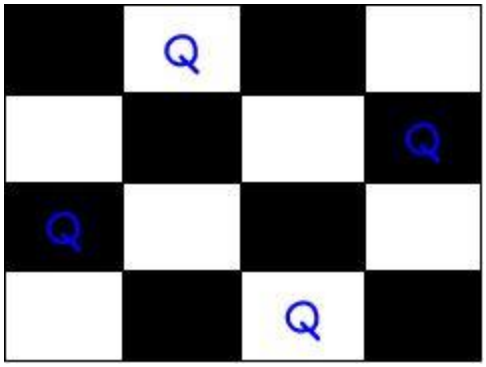
Step 3 – if current_position is an end point, return failed.

Step 4 – else, if current_position is not end point, explore and repeat above steps.

Let's use this backtracking problem to find the solution to **N-Queen Problem**.

In N-Queen problem, we are given an NxN chessboard and we have to place n queens on the board in such a way that no two queens attack each other. A queen will attack another queen if it is placed in horizontal, vertical or diagonal points in its way. Here, we will do 4-Queen problem.

Here, the solution is −



Here, the binary output for n queen problem with 1's as queens to the positions are placed.

{0 , 1 , 0 , 0}

{0 , 0 , 0 , 1}

{1 , 0 , 0 , 0}

{0 , 0 , 1 , 0}

For solving n queens problem, we will try placing queen into different positions of one row. And checks if it clashes with other queens. If current positioning of queens if there are any two queens attacking each other. If they are attacking, we will backtrack to previous location of the queen and change its positions. And check clash of queen again.

Algorithm

Step 1 − Start from 1st position in the array.

Step 2 − Place queens in the board and check. Do,

Step 2.1 − After placing the queen, mark the position as a part of the solution and then recursively check if this will lead to a solution.

Step 2.2 − Now, if placing the queen doesn't lead to a solution and trackback and go to step (a) and place queens to other rows.

Step 2.3 − If placing queen returns a lead to solution return **TRUE.**

Step 3 − If all queens are placed return TRUE.

Step 4 − If all rows are tried and no solution is found, return FALSE.

There are three types of problems in backtracking –

1. Decision Problem – In this, we search for a feasible solution.
2. Optimization Problem – In this, we search for the best solution.
3. Enumeration Problem – In this, we find all feasible solutions.

How to determine if a problem can be solved using Backtracking?

Generally, every constraint satisfaction problem which has clear and well-defined constraints on any objective solution, that incrementally builds candidate to the solution and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution, can be solved by Backtracking. However, most of the problems that are discussed, can be solved using other known algorithms like *Dynamic Programming* or *Greedy Algorithms* in logarithmic, linear, linear-logarithmic time complexity in order of input size, and therefore, outshine the backtracking algorithm in every respect (since backtracking algorithms are generally exponential in both time and space). However, a few problems still remain, that only have backtracking algorithms to solve them until now.

Consider a situation that you have three boxes in front of you and only one of them has a gold coin in it but you do not know which one. So, in order to get the coin, you will have to open all of the boxes one by one. You will first check the

first box, if it does not contain the coin, you will have to close it and check the second box and so on until you find the coin. This is what backtracking is, that is solving all sub-problems one by one in order to reach the best possible solution.

Consider the below example to understand the Backtracking approach more formally,

Given an instance of any computational problem and data corresponding to the instance, all the constraints that need to be satisfied in order to solve the problem are represented by . A backtracking algorithm will then work as follows:

The Algorithm begins to build up a solution, starting with an empty solution set . **S = {}**

1. Add to the first move that is still left (All possible moves are added to one by one). This now creates a new sub-tree in the search tree of the algorithm.

2. Check if satisfies each of the constraints in .
   - If Yes, then the sub-tree is "eligible" to add more "children".
   - Else, the entire sub-tree is useless, so recurs back to step 1 using argument .

3. In the event of "eligibility" of the newly formed sub-tree , recurs back to step 1, using argument .

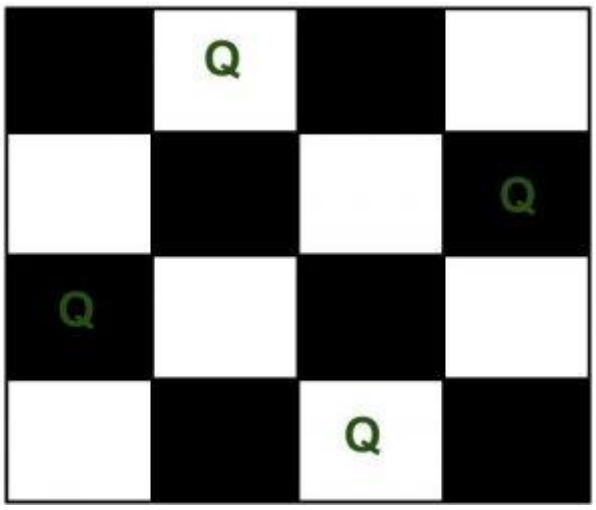
4. If the check for returns that it is a solution for the entire data .
   Output and terminate the program.
   If not, then return that no solution is possible with the current and hence discard it.

Pseudo Code for Backtracking :

```
1. Recursive backtracking solution.
2. void findSolutions(n, other params) :
3.     if (found a solution) :
4.         solutionsFound = solutionsFound + 1;
5.         displaySolution();
6.         if (solutionsFound >= solutionTarget) :
7.             System.exit(0);
8.         return
9.
10.    for (val = first to last) :
11.        if (isValid(val, n)) :
12.            applyValue(val, n);
13.            findSolutions(n+1, other params);
14.            removeValue(val, n);
15.Finding whether a solution exists or not
16.boolean findSolutions(n, other params) :
17.    if (found a solution) :
18.        displaySolution();
19.        return true;
20.
21.    for (val = first to last) :
```

```
22.     if (isValid(val, n)) :
23.         applyValue(val, n);
24.         if (findSolutions(n+1, other params))
25.             return true;
26.         removeValue(val, n);
27.     return false;
```

Let us try to solve a standard Backtracking problem, **N-Queen Problem**.
The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for the above 4 queen solution.

{ 0, 1, 0, 0}

{ 0, 0, 0, 1}

{ 1, 0, 0, 0}

{ 0, 0, 1, 0}

Backtracking Algorithm: The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

1) Start in the leftmost column

2) If all queens are placed

   return true

3) Try all rows in the current column. Do following for every tried row.

   a) If the queen can be placed safely in this row then mark this [row,

      column] as part of the solution and recursively check if placing

      queen here leads to a solution.

   b) If placing the queen in [row, column] leads to a solution then return

      true.

   c) If placing queen doesn't lead to a solution then unmark this [row,

      column] (Backtrack) and go to step (a) to try other rows.

3) If all rows have been tried and nothing worked, return false to trigger

   backtracking.

Branch and Bound Algorithm

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically

exponential in terms of time complexity and may require exploring all possible permutations in worst case. The Branch and Bound Algorithm technique solves these problems relatively quickly.

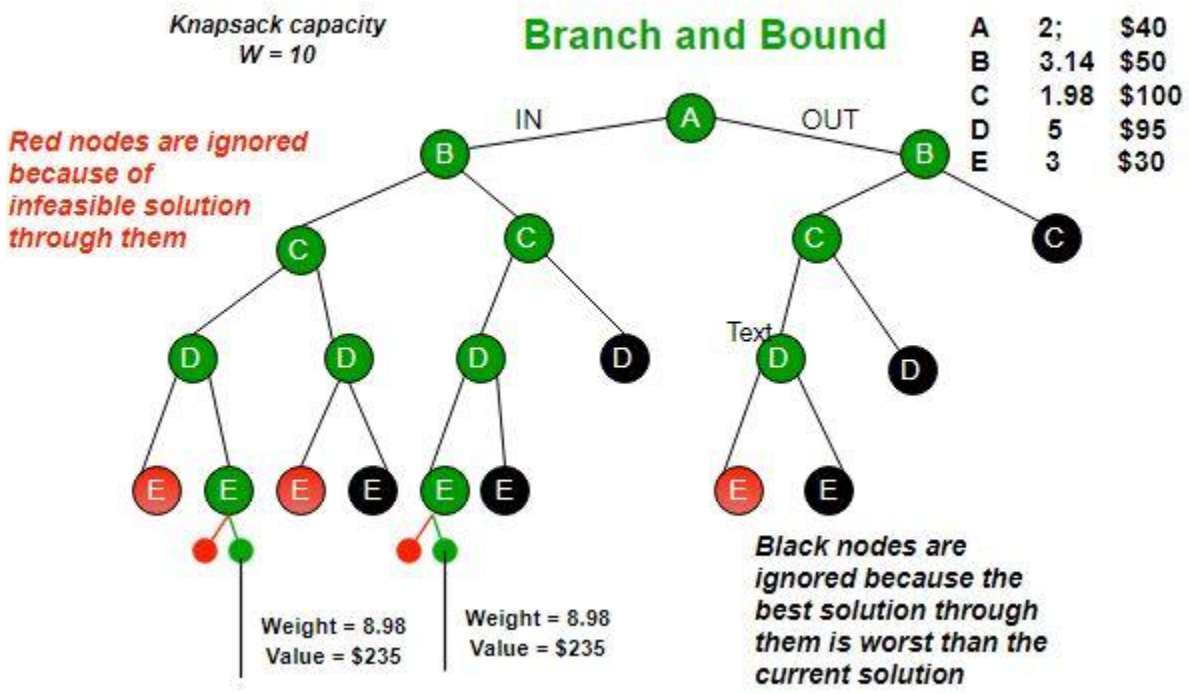
the **0/1 Knapsack problem** to understand Branch and Bound.

There are many algorithms by which the knapsack problem can be solved:

- Greedy Algorithm for Fractional Knapsack
- DP solution for 0/1 Knapsack
- Backtracking Solution for 0/1 Knapsack.

the Branch and Bound Approach to solve the **0/1 Knapsack problem**: The Backtracking Solution can be optimized if we know a bound on best possible solution subtree rooted with every node. If the best in subtree is worse than current best, we can simply ignore this node and its subtrees. So we compute bound (best solution) for every node and compare the bound with current best solution before exploring the node.

Example bounds used in below diagram are, A down can give $315, B down can $275, C down can $225, D down can $125 and E down can $30.



Lower Bound Theory

Lower Bound Theory Concept is based upon the calculation of minimum time that is required to execute an algorithm is known as a lower bound theory or Base Bound Theory.

Lower Bound Theory uses a number of methods/techniques to find out the lower bound.

Concept/Aim: The main aim is to calculate a minimum number of comparisons required to execute an algorithm.

Techniques:

The techniques which are used by lower Bound Theory are:

1. Comparisons Trees.
2. Oracle and adversary argument

3. State Space Method

1. Comparison trees:

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence (a1; a2.....an).

Given a_i,a_j from (a_1, a_2.....a_n)We Perform One of the Comparisons

- $a_i < a_j$ less than
- $a_i \leq a_j$ less than or equal to
- $a_i > a_j$ greater than
- $a_i \geq a_j$ greater than or equal to
- $a_i = a_j$ equal to

To determine their relative order, if we assume all elements are distinct, then we just need to consider $a_i \leq a_j$ '=' is excluded &, ≥,≤,>,< are equivalent.

Consider sorting three numbers a1, a2, and a3. There are 3! = 6 possible combinations:

1. (a1, a2, a3), (a1, a3, a2),
2. (a2, a1, a3), (a2, a3, a1)
3. (a3, a1, a2), (a3, a2, a1)

The Comparison based algorithm defines a decision tree.

Decision Tree: A decision tree is a full binary tree that shows the comparisons between elements that are executed by an appropriate sorting algorithm operating on an input of a given size. Control, data movement, and all other conditions of the algorithm are ignored.

In a decision tree, there will be an array of length n.

So, total leaves will be n! (I.e. total number of comparisons)

If tree height is h, then surely

$n! \leq 2^n$ (tree will be binary)

Taking an Example of comparing a1, a2, and a3.

Left subtree will be true condition i.e. $a_i \leq a_j$

Right subtree will be false condition i.e. $a_i > a_j$

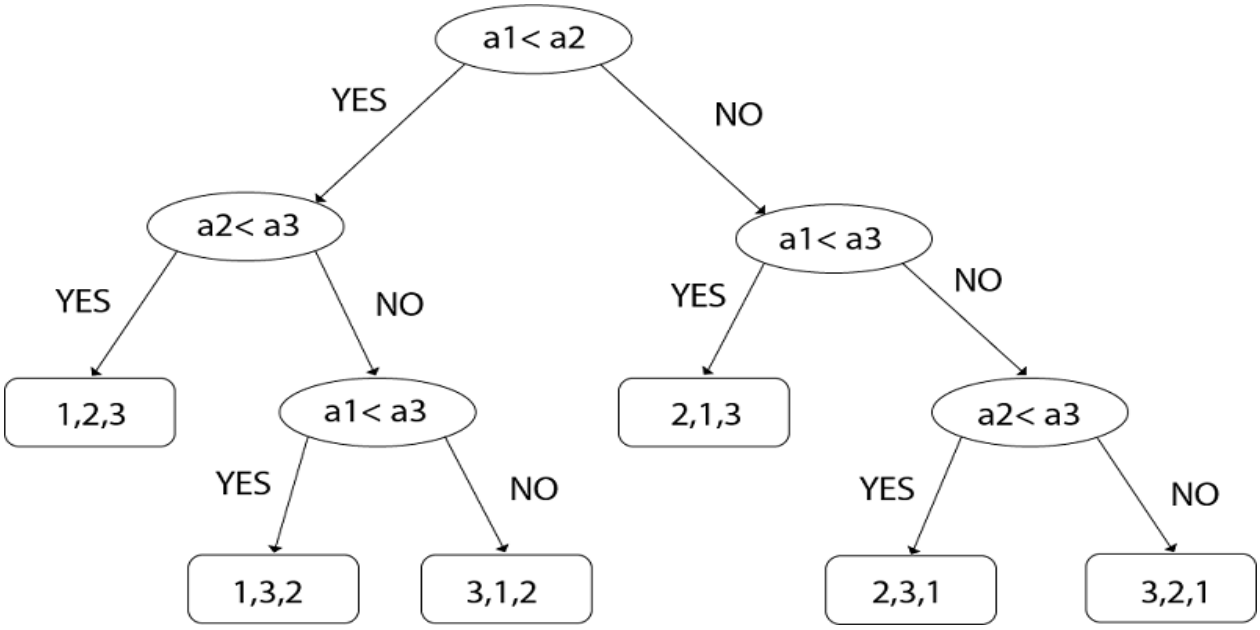


Fig: Decision Tree

So from above, we got

$N! \leq 2^n$

Taking Log both sides

Log n! ≤ h log 2

h log 2≥log n!

h ≥ log2 n!

h≥log 2 [1, 2, 3……n]

$h \geq \log_2 1 + \log_2 2 + \log_2 3 + \cdots \log_2 n$

$h \geq \sum_{i=1}^{n} \log_2 i$

$h \geq \int_i^n \log_2 i - 1 \, d^i$

$h \geq \log_2 i \, . x^o \int_1^n - \int_1^n \frac{1}{i} xi \, di$

$h \geq n\log_2 n - \int_1^n 1 \, di$

$h \geq n\log_2 n - i \int_1^n$

$h \geq n \log_2 n - n + 1$

Ignoring the Constant terms

$h \geq n\log_2 n$

h=π n (log n)

Comparison tree for Binary Search:

Example: Suppose we have a list of items according to the following Position:

1. 1,2,3,4,5,6,7,8,9,10,11,12,13,14

Mid = $\left[\left(\frac{1+14}{2}\right)\right] = \frac{15}{2} = 7.5 = \mathbf{7}$

Note: Choose the greatest integer

1, 2, 3, 4, 5, 6	8, 9, 10, 11, 12, 13, 14
Mid=$\left(\frac{1+6}{2}\right)$=**3**	Mid =$\left(\frac{8+14}{2}\right)$ = **11**

1, 2	4, 5, 6	8, 9, 10	12, 13, 14
Mid=$\left(\frac{1+2}{2}\right)$=**1**	Mid=$\left(\frac{4+6}{2}\right)$=**5**	Mid= $\left(\frac{8+10}{2}\right)$=**9**	Mid=$\left(\frac{12+14}{2}\right)$=**13**

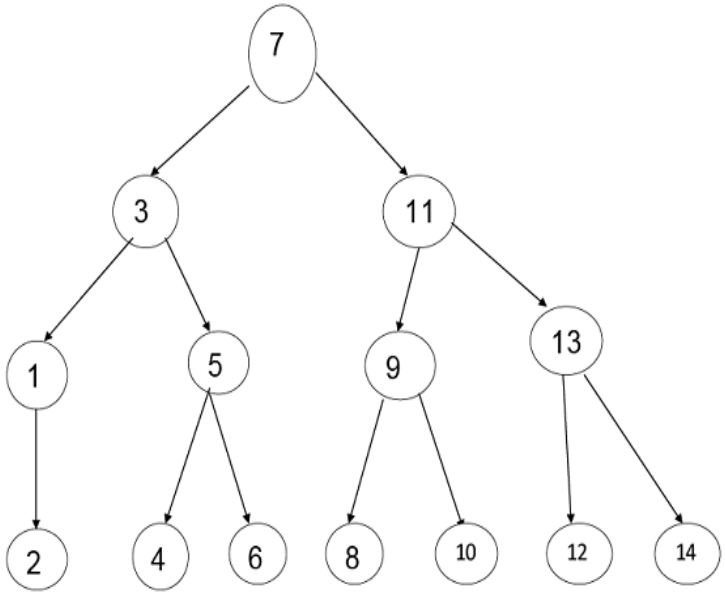
And the last midpoint is:

1. 2, 4, 6, 8, 10, 12, 14

Thus, we will consider all the midpoints and we will make a tree of it by having stepwise midpoints.

The Bold letters are Mid-Points Here

According to Mid-Point, the tree will be:



Step1: Maximum number of nodes up to k level of the internal node is 2^k-1

For Example

2^k-1

2^3-1= 8-1=7

Where k = level=3

Step2: Maximum number of internal nodes in the comparisons tree is n!

Note: Here Internal Nodes are Leaves.

Step3: From Condition1 & Condition 2 we get

N! ≤ 2^k-1

14 < 15

Where N = Nodes

Step4: Now, n+1 ≤ 2^k

Here, Internal Nodes will always be less than 2^k in the Binary Search.

Step5:

$n+1 <= 2^k$

$\text{Log}(n+1) = k \log 2$

$k >= \frac{\log(n+1)}{\log 2}$

$k >= \log_2(n+1)$

Step6:

1. T (n) = k

Step7:

$T(n) >= \log_2(n+1)$

Here, the minimum number of Comparisons to perform a task of the search of n terms using Binary Search

2. Oracle and adversary argument:

Another technique for obtaining lower bounds consists of making use of an "oracle."

Given some model of estimation such as comparison trees, the oracle tells us the outcome of each comparison.

In order to derive a good lower bound, the oracle efforts it's finest to cause the algorithm to work as hard as it might.

It does this by deciding as the outcome of the next analysis, the result which matters the most work to be needed to determine the final answer.

And by keeping step of the work that is finished, a worst-case lower bound for the problem can be derived.

Example: (Merging Problem) given the sets A (1: m) and B (1: n), where the information in A and in B are sorted. Consider lower bounds for algorithms combining these two sets to give an individual sorted set.

Consider that all of the m+n elements are specific and A (1) < A (2) < ....< A (m) and B (1) < B (2) < ....< B (n).

Elementary combinatory tells us that there are C ((m+n), n)) ways that the A's and B's may merge together while still preserving the ordering within A and B.

Thus, if we need comparison trees as our model for combining algorithms, then there will be C ((m+n), n)) external nodes and therefore at least log C ((m+n), m) comparisons are needed by any comparison-based merging algorithm.

If we let MERGE (m, n) be the minimum number of comparisons used to merge m items with n items then we have the inequality

1. Log C ((m+n), m) MERGE (m, n) m+n-1.

The upper bound and lower bound can get promptly far apart as m gets much smaller than n.

3. State Space Method:

1. State Space Method is a set of rules that show the possible states (n-tuples) that an algorithm can assume from a given state of a single comparison.

2. Once the state transitions are given, it is possible to derive lower bounds by arguing that the finished state cannot be reached using any fewer transitions.

3. Given n distinct items, find winner and loser.

4. Aim: When state changed count it that is the aim of State Space Method.

5. In this approach, we will count the number of comparison by counting the number of changes in state.

6. Analysis of the problem to find out the smallest and biggest items by using the state space method.

7. State: It is a collection of attributes.

8. Through this we sort out two types of Problems:

1. Find the largest & smallest element from an array of elements.
2. To find out largest & second largest elements from an array of an element.

9. For the largest item, we need 7 comparisons and what will be the second largest item?

Now we count those teams who lose the match with team A

Teams are: B, D, and E

So the total no of comparisons are: 7

Let n is the total number of items, then

Comparisons = n-1 (to find the biggest item)

No of Comparisons to find out the 2nd biggest item = $\log_2 n-1$

10. In this **no of comparisons** are equal to the **number of changes of states** during the execution of the algorithm.

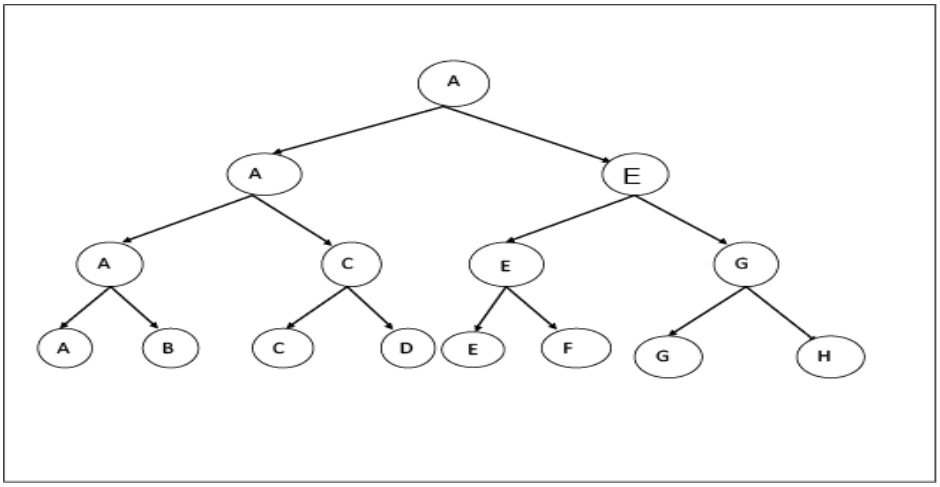
Example: State (A, B, C, D)

A. No of items that can never be compared.

B. No of items that win but never lost (ultimate winner).

C. No of items that lost but never win (ultimate loser)

D. No of items who sometimes lost & sometimes win.



Firstly, A, B compare out or match between them. A wins and in C, D.C wins and so on. We can assume that B wins and so on. We can assume that B wins in place of A it can be anything depending on our self.

Phase-1:

A	B	C	D	
8	0	0	0	No match occurs; there are total 8 external nodes or 8 teams
6	1	1	0	Match between A & B,6 left in A state as only one match is done between A & B.1 add to B and 1 add to C .
4	2	2	0	C & D match out 4 left out 1 for C wins and 1 for D's lost and 0 for there is no team that sometimes wins or lost.
2	3	3	0	Match between E & F
0	4	4	0	Match between E & F and Match between G & H are compare out.

In Phase-1 there are 4 states
If the team is 8 then 4 states
As if n team the n/2 states.

Phase-2

A	B	C	D	
0	4	4	0	Started at point where Phase 1 ends. Initial States for Phase-2
0	3	4	1	A & C both wins from lower level. But now A wins and C lost. But, C wins from lower level so C will be counted as sometimes wins & sometimes lost.
0	2	4	2	E & G are match E wins but G lost but G wins from lower level. It will be counted as sometimes wins & sometimes lost.
0	1	4	3	A & C both wins at Phase-1 but now A wins and C lost. So, C wins before and last now A will come in state- D

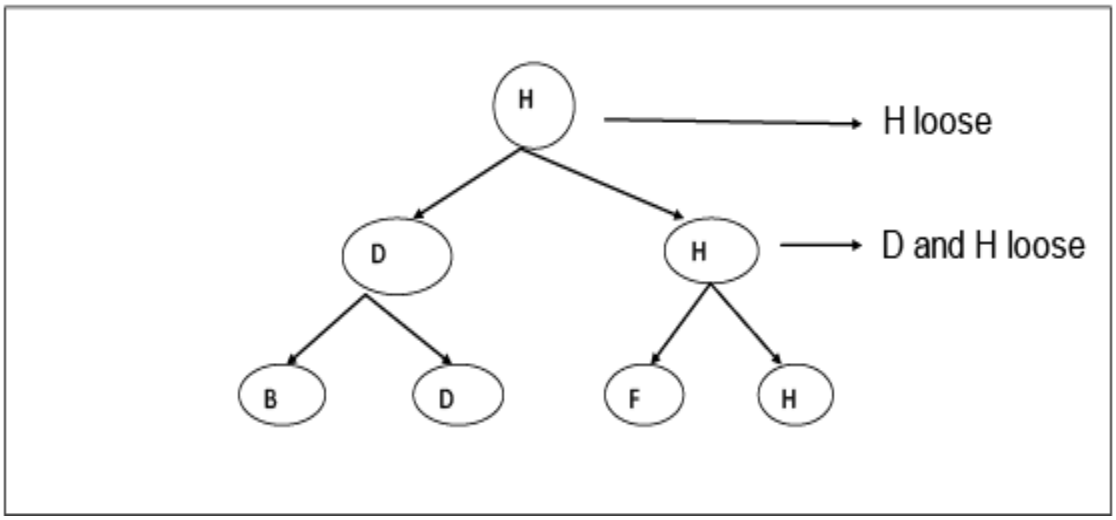
4 is Constant in C-State as B, D, F, H are lost teams that never win.

Thus there are 3 states in Phase-2,
If n is 8 then states are 3

If n teams that are $\left(\frac{n}{2}-1\right)$ states.

Phase-3: This is a Phase in which teams which come under C-State are considered and there will be matches between them to find out the team which is never winning at all.

In this Structure, we are going to move upward for denoting who is not winning after the match.



Here H is the team which is never winning at all. By this, we fulfill our second aim to.

A	B	C	D
0	1	4	3
0	1	3	4
0	1	2	5
0	1	1	6

Here by match out of B & D .We find that B wins and D looses But B at lower level looses∴ It will be counted as sometimes wins and lost and it will be removed from lost team.∴ 4 will turn to 3 from lost team.

After match out F & H, F wins and H looses but F looses at lower level ∴F will be counted as sometimes wins & lost also removed from lost team thus 2 turn to 1 from lost team.

Match between F & H

After match out of D & H.H wins and D lost but H looses from lower level ∴it will be counted as sometimes wins & lost also removed from lost team thus 2 turn to 1 from lost team.

Thus there are 3 states in Phase -3

If n teams that are $\left(\frac{n}{2}-1\right)$ states

Note: the total of all states value is always equal to 'n'.

Thus, by adding all phase's states we will get:-

Phase1 + Phase 2 + Phase 3

$$\frac{n}{2} + \left(\frac{n}{2} - 1\right) + \left(\frac{n}{2}\right) - 1$$

$$\frac{n}{2} + \frac{n}{2} - 1 + \frac{n}{2} - 1$$

$$\frac{n+n-2+n-2=}{2} \quad \frac{3n-4}{2}$$

$$\frac{3n-2}{2}$$

Suppose we have 8 teams then $\left[3\left(\frac{8}{2}\right) - 2\right]=10$ states are there (as minimum) to find out which one is never wins team.

Thus, Equation is:

$$T(n) = \frac{3n-2}{2}$$

Lower bound (L (n)) is a property of the particular issue i.e. the sorting problem, matrix multiplication not of any particular algorithm solving that problem.

Lower bound theory says that no calculation can carry out the activity in less than that of (L (n)) times the units for arbitrary inputs i.e. that for every comparison based sorting algorithm must take at least **L (n)** time in the worst case.

L (n) is the base overall conceivable calculation which is greatest finished.

Trivial lower bounds are utilized to yield the bound best alternative is to count the number of elements in the problems input that must be prepared and the number of output items that need to be produced.

The lower bound theory is the method that has been utilized to establish the given algorithm in the most efficient way which is possible. This is done by discovering a function **g (n)** that is a lower bound on the time that any algorithm

must take to solve the given problem. Now if we have an algorithm whose computing time is the same order as **g (n)** , then we know that asymptotically we cannot do better.

If **f (n)** is the time for some algorithm, then we write **f (n) = Ω (g (n))** to mean that **g (n)** is the **lower bound of f (n)** . This equation can be formally written, if there exists positive constants **c** and **n0** such that **|f (n)| >= c|g (n)|** for all **n > n0**. In addition for developing lower bounds within the constant factor, we are more conscious of the fact to determine more exact bounds whenever this is possible.

Deriving good **lower bounds** is more challenging than arrange efficient algorithms. This happens because a lower bound states a fact about all possible algorithms for solving a problem. Generally, we cannot enumerate and analyze all these algorithms, so lower bound proofs are often hard to obtain.

Lower bounds by reduction

Suppose we already have a lower bound S(n) on some problem A, and that there is a reduction from A to some other problem B. Formally, a reduction is a function f such that A(x) = B(f(x)). Let $T_f(n)$ be an upper bound on the time to compute f(x) when |x| = n, and let $T_B(n)$ be an upper bound on the time to compute B(x) when |x| = n. Then computing A(x) as B(f(x)) with |x| = n takes time at most $T_f(n) + T_B(T_f(n))$. (The odd second term comes from the fact that $T_f(n)$ is an upper bound on the length of the output of f(x); note that we are also assuming here that T_B is nondecreasing.)

An important special case of this is when $T_f = O(n^c)$ and $T_B = O(n^d)$ for some c and d. Then $T_f(n) + T_f(T_B(n)) = O(n^c) + O(n^{cd}) = O(n^{cd})$. In words:

- Any problem that has a polynomial-time reduction to a polynomial time problem can be solved in polynomial time.

How do we get lower bounds out of this? Since $S(n) \leq T_f(n) + T_f(T_B(n))$, we have $T_B(n) \geq T_f^{-1}(S(n) - T_f(n))$. For example:

- Suppose there is a reduction f from sorting to some problem B that takes O(n) time and does not examine the elements of the sorting problem. Then B(f(x)) is a comparison-based sorting algorithm if B uses only comparisions, and $T_B(n) = \Omega(n \log n) - O(n) = \Omega(n \log n)$.
- Suppose there is no polynomial-time algorithm for A but there is a polynomial-time reduction from A to B. Then there is no polynomial-time algorithm for B. This is the central idea behind NP-completeness, discussed in PvsNp.

Shortest Paths

The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.

This problem could be solved easily using **(BFS)** if all edge weights were (1), but here weights can take any value. Three different algorithms are discussed below depending on the use-case.

Bellman Ford's Algorithm:

Bellman Ford's algorithm is used to find the shortest paths from the source vertex to all other vertices in a weighted graph. It depends on the following concept: Shortest path contains at most n−1 edges, because the shortest path couldn't have a cycle.

So why shortest path shouldn't have a cycle ?
There is no need to pass a vertex again, because the shortest path to all other vertices could be found without the need for a second visit for any vertices.

Algorithm Steps:

- The outer loop traverses from 0 : n−1.

- Loop over all edges, check if the next node distance > current node distance + edge weight, in this case update the next node distance to "current node distance + edge weight".

This algorithm depends on the relaxation principle where the shortest distance for all vertices is gradually replaced by more accurate values until eventually reaching the optimum solution. In the beginning all vertices have a distance of "Infinity", but only the distance of the source vertex = 0, then update all the connected vertices with the new distances (source vertex distance + edge weights), then apply the same concept for the new vertices with new distances and so on.

Implementation:

Assume the source node has a number (0):

```
vector <int> v [2000 + 10];
int dis [1000 + 10];


for(int i = 0; i < m + 2; i++){


   v[i].clear();
   dis[i] = 2e9;
}


for(int i = 0; i < m; i++){


   scanf("%d%d%d", &from , &next , &weight);
```

```
    v[i].push_back(from);
    v[i].push_back(next);
    v[i].push_back(weight);
}

 dis[0] = 0;
 for(int i = 0; i < n - 1; i++){
    int j = 0;
    while(v[j].size() != 0){

       if(dis[ v[j][0]  ] + v[j][2] < dis[ v[j][1] ] ){
          dis[ v[j][1] ] = dis[ v[j][0]  ] + v[j][2];
       }
       j++;
    }
 }
```

A very important application of Bellman Ford is to check if there is a negative cycle in the graph,

Time Complexity of Bellman Ford algorithm is relatively high O(V·E), in case E=V2, O(V3).
Let's discuss an optimized algorithm.

Dijkstra's Algorithm

Dijkstra's algorithm has many variants but the most common one is to find the shortest paths from the source vertex to all other vertices in the graph.

Algorithm Steps:

- Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
- Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.
- Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
- Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex
  with the new distance to the priority queue.
- If the popped vertex is visited before, just continue without using it.
- Apply the same algorithm again until the priority queue is empty.

Implementation:

```
Assume the source vertex = 1.

#define SIZE 100000 + 1



vector < pair < int , int > > v [SIZE];   // each vertex has all the connected vertices with the edges weights

int dist [SIZE];
```

```
bool vis [SIZE];


void dijkstra(){
                                    // set the vertices distances as infinity
   memset(vis, false , sizeof vis);          // set all vertex as unvisited
   dist[1] = 0;
   multiset < pair < int , int > > s;        // multiset do the job as a min-priority
queue


   s.insert({0 , 1});                    // insert the source node with distance = 0


   while(!s.empty()){


      pair <int , int> p = *s.begin();       // pop the vertex with the minimum
distance
      s.erase(s.begin());


      int x = p.s; int wei = p.f;
      if( vis[x] ) continue;               // check if the popped vertex is visited before
       vis[x] = true;


      for(int i = 0; i < v[x].size(); i++){
```

```
        int e = v[x][i].f; int w = v[x][i].s;

        if(dist[x] + w < dist[e]  ){          // check if the next vertex distance could be
minimized

            dist[e] = dist[x] + w;

            s.insert({dist[e],  e} );         // insert the next vertex with the updated
distance

        }

      }

   }

}
```

Time Complexity of Dijkstra's Algorithm is O(V2) but with min-priority queue it drops down to O(V+ElogV).

However, if we have to find the shortest path between all pairs of vertices, both of the above methods would be expensive in terms of time. Discussed below is another alogorithm designed for this case.

Floyd\u2013Warshall's Algorithm

Floyd\u2013Warshall's Algorithm is used to find the shortest paths between between all pairs of vertices in a graph, where each edge in the graph has a weight which is positive or negative. The biggest advantage of using this algorithm is that all the shortest distances between any 2 vertices could be calculated in O(V3), where V is the number of vertices in a graph.

The Algorithm Steps:

For a graph with N vertices:

- Initialize the shortest paths between any 2 vertices with Infinity.

- Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1 intermediate vertex and so on.. until using all N vertices as intermediate nodes.
- Minimize the shortest paths between any 2 pairs in the previous operation.
- For any 2 vertices (i,j) , one should actually minimize the distances between this pair using the first K nodes, so the shortest path will be: min(dist[i][k]+dist[k][j],dist[i][j]).

dist[i][k] represents the shortest path that only uses the first K vertices, dist[k][j] represents the shortest path between the pair k,j. As the shortest path will be a concatenation of the shortest path from i to k, then from k to j.

```
for(int k = 1; k <= n; k++){

   for(int i = 1; i <= n; i++){

      for(int j = 1; j <= n; j++){

         dist[i][j] = min( dist[i][j], dist[i][k] + dist[k][j] );

      }

   }

}
```

Time Complexity of Floyd\u2013Warshall's Algorithm is O(V3), where V is the number of vertices in a graph.

Maximum flow

In graph theory, a flow network is defined as a directed graph involving a source(S) and a sink(T) and several other nodes connected with edges. Each edge has an individual capacity which is the maximum limit of flow that edge could allow.

Flow in the network should follow the following conditions:

- For any non-source and non-sink node, the input flow is equal to output flow.
- For any edge(Ei) in the network, 0≤flow(Ei)≤Capacity(Ei).
- Total flow out of the source node is equal total to flow in to the sink node.
- Net flow in the edges follows skew symmetry
  i.e. F(u,v)=−F(v,u) where F(u,v) is flow from node u to node v. This leads to a conclusion where you have to sum up all the flows between two nodes(either directions) to find net flow between the nodes initially.

Maximum Flow:
It is defined as the maximum amount of flow that the network would allow to flow from source to sink. Multiple algorithms exist in solving the maximum flow problem. Two major algorithms to solve these kind of problems are Ford-Fulkerson algorithm and Dinic's Algorithm. They are explained below.

Ford-Fulkerson Algorithm:
It was developed by L. R. Ford, Jr. and D. R. Fulkerson in 1956. A pseudocode for this algorithm is given below,

Inputs required are network graph G, source node S and sink node T.

function: FordFulkerson(Graph G,Node S,Node T):

Initialise flow in all edges to 0

while (there exists an augmenting path(P) between S and T in residual network graph):

Augment flow between S to T along the path P

Update residual network graph

return

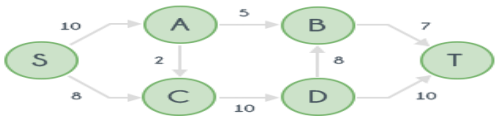
An augmenting path is a simple path from source to sink which do not include any cycles and that pass only through positive weighted edges. A residual network graph indicates how much more flow is allowed in each edge in the network graph. If there are no augmenting paths possible from S to T, then the flow is maximum. The result i.e. the maximum flow will be the total flow out of source node which is also equal to total flow in to the sink node.

A demonstration of working of Ford-Fulkerson algorithm is shown below with the help of diagrams.

Network (G)



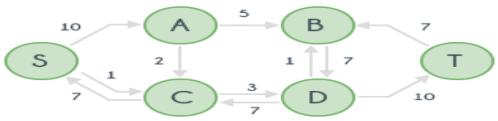
Residual Graph (G_R)



Flow = 0

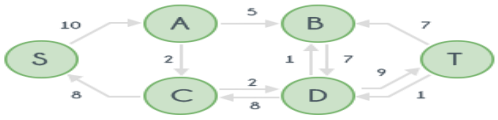
Path 1: S - C - D - B - T ⟶ Flow = Flow + 7





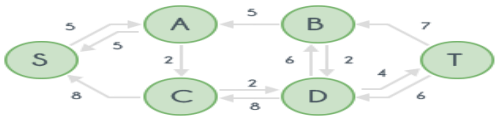
Path 2: S - C - D - T ⟶ Flow = Flow + 1





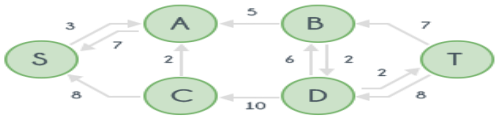
Path 3: S - A - B - T ⟶ Flow = Flow + 5





Path 4: S - A - C - D - T ⟶ Flow = Flow + 2



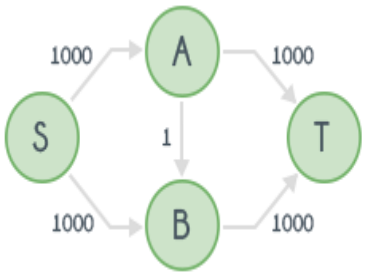


No More Paths Left
Max Flow = 15

Implementation:

- An augmenting path in residual graph can be found using DFS or BFS.
- Updating residual graph includes following steps: (refer the diagrams for better understanding)
    - For every edge in the augmenting path, a value of minimum capacity in the path is subtracted from all the edges of that path.
    - An edge of equal amount is added to edges in reverse direction for every successive nodes in the augmenting path.

The complexity of Ford-Fulkerson algorithm cannot be accurately computed as it all depends on the path from source to sink. For example, considering the network shown below, if each time, the path chosen
are S−A−B−T and S−B−A−T alternatively, then it can take a very long time. Instead, if path chosen are only S−A−T and S−B−T, would also generate the maximum flow.



Dinic's Algorithm

In 1970, **Y. A. Dinitz** developed a faster algorithm for calculating maximum flow over the networks. It includes construction of level graphs and residual graphs and finding of augmenting paths along with blocking flow.

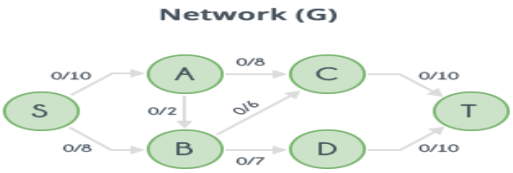
Level graph is one where value of each node is its shortest distance from source. Blocking flow includes finding the new path from the bottleneck node. Residual graph and augmenting paths are previously discussed.

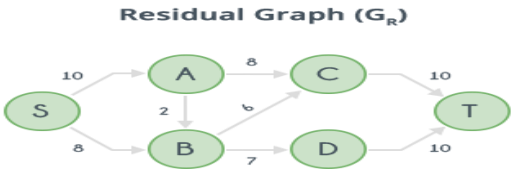
Pseudocode for Dinic's algorithm is given below.

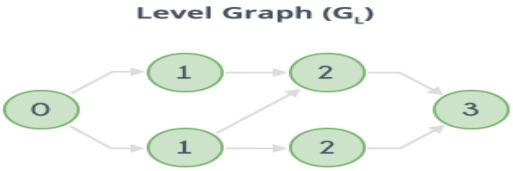
Inputs required are network graph G, source node S and sink node T.

```
function: DinicMaxFlow(Graph G,Node S,Node T):
    Initialize flow in all edges to 0, F = 0
    Construct level graph
    while (there exists an augmenting path in level graph):
        find blocking flow f in level graph
        F = F + f
        Update level graph
    return F
```

Update of level graph includes removal of edges with full capacity. Removal of nodes that are not sink and are dead ends. A demonstration of working of Dinic's algorithm is shown below with the help of diagrams.



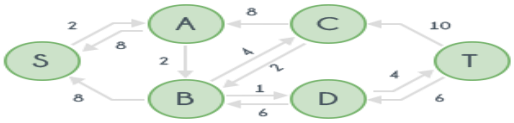


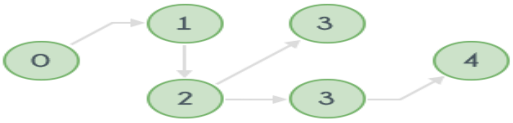


Blocking Flow Paths

S - A - C - T ⟶ Flow = 8
S - B - C - T ⟶ Flow = 2
S - A - D - T ⟶ Flow = 6



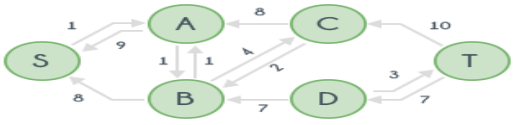


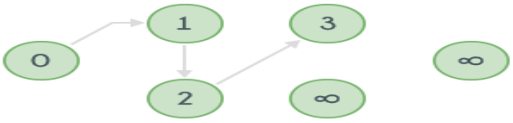


Blocking Flow Paths

S - A - B - D - T ⟶ Flow = 1







No Paths Left
Total Flow = 17

P and NP Class Problems

In Computer Science, many problems are solved where the objective is to maximize or minimize some values, whereas in other problems we try to find whether there is a solution or not. Hence, the problems can be categorized as follows −

Optimization Problem

Optimization problems are those for which the objective is to maximize or minimize some values. For example,

- Finding the minimum number of colors needed to color a given graph.
- Finding the shortest path between two vertices in a graph.

Decision Problem

There are many problems for which the answer is a Yes or a No. These types of problems are known as **decision problems**. For example,

- Whether a given graph can be colored by only 4-colors.
- Finding Hamiltonian cycle in a graph is not a decision problem, whereas checking a graph is Hamiltonian or not is a decision problem.

What is Language?

Every decision problem can have only two answers, yes or no. Hence, a decision problem may belong to a language if it provides an answer 'yes' for a specific input. A language is the totality of inputs for which the answer is Yes. Most of the algorithms discussed in the previous chapters are **polynomial time algorithms**.

For input size ***n***, if worst-case time complexity of an algorithm is $O(n^k)$, where ***k*** is a constant, the algorithm is a polynomial time algorithm.

Algorithms such as Matrix Chain Multiplication, Single Source Shortest Path, All Pair Shortest Path, Minimum Spanning Tree, etc. run in polynomial time. However there are many problems, such as traveling salesperson, optimal graph coloring,

Hamiltonian cycles, finding the longest path in a graph, and satisfying a Boolean formula, for which no polynomial time algorithms is known. These problems belong to an interesting class of problems, called the **NP-Complete** problems, whose status is unknown.

In this context, we can categorize the problems as follows −

P-Class

The class P consists of those problems that are solvable in polynomial time, i.e. these problems can be solved in time $O(n^k)$ in worst-case, where **k** is constant.

These problems are called **tractable**, while others are called **intractable or superpolynomial**.

Formally, an algorithm is polynomial time algorithm, if there exists a polynomial $p(n)$ such that the algorithm can solve any instance of size **n** in a time $O(p(n))$.

Problem requiring $\Omega(n^{50})$ time to solve are essentially intractable for large n. Most known polynomial time algorithm run in time $O(n^k)$ for fairly low value of k.

The advantages in considering the class of polynomial-time algorithms is that all reasonable **deterministic single processor model of computation** can be simulated on each other with at most a polynomial slow-d

NP-Class

The class NP consists of those problems that are verifiable in polynomial time. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information. Hence, we aren't asking for a way to find a solution, but only to verify that an alleged solution really is correct.

Every problem in this class can be solved in exponential time using exhaustive search.

P versus NP

Every decision problem that is solvable by a deterministic polynomial time algorithm is also solvable by a polynomial time non-deterministic algorithm.

All problems in P can be solved with polynomial time algorithms, whereas all problems in *NP - P* are intractable.

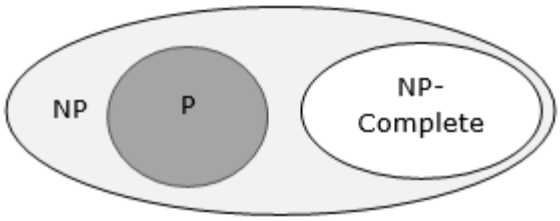
It is not known whether ***P = NP***. However, many problems are known in NP with the property that if they belong to P, then it can be proved that P = NP.

If ***P ≠ NP***, there are problems in NP that are neither in P nor in NP-Complete.

The problem belongs to class **P** if it's easy to find a solution for the problem. The problem belongs to **NP**, if it's easy to check a solution that may have been very tedious to find.

NP-Completeness

A problem is in the class NPC if it is in NP and is as **hard** as any problem in NP. A problem is **NP-hard** if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.



If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called **NP-complete**. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

Definition of NP-Completeness

A language **B** is ***NP-complete*** if it satisfies two conditions

- **B** is in NP
- Every **A** in NP is polynomial time reducible to **B**.

If a language satisfies the second property, but not necessarily the first one, the language **B** is known as **NP-Hard**. Informally, a search problem **B** is **NP-Hard** if there exists some **NP-Complete** problem **A** that Turing reduces to **B**.

The problem in NP-Hard cannot be solved in polynomial time, until **P = NP**. If a problem is proved to be NPC, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation algorithm.

NP-Complete Problems

Following are some NP-Complete problems, for which no polynomial time algorithm is known.

- Determining whether a graph has a Hamiltonian cycle
- Determining whether a Boolean formula is satisfiable, etc.

NP-Hard Problems

The following problems are NP-Hard

- The circuit-satisfiability problem
- Set Cover
- Vertex Cover
- Travelling Salesman Problem

In this context, now we will discuss TSP is NP-Complete

TSP is NP-Complete

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip

Proof

To prove ***TSP is NP-Complete***, first we have to prove that ***TSP belongs to NP***. In TSP, we find a tour and check that the tour contains each vertex once. Then the total cost of the edges of the tour is calculated. Finally, we check if the cost is minimum. This can be completed in polynomial time. Thus ***TSP belongs to NP***.

Secondly, we have to prove that ***TSP is NP-hard***. To prove this, one way is to show that ***Hamiltonian cycle $\leq_p$ TSP*** (as we know that the Hamiltonian cycle problem is NPcomplete).

Assume ***G = (V, E)*** to be an instance of Hamiltonian cycle.

Hence, an instance of TSP is constructed. We create the complete graph ***G′ = (V, E′)***, where

$$E'=\{(i,j):i,j\in V and i\neq j$$

Thus, the cost function is defined as follows −

$$t(i,j)=\begin{cases}0 & if\ (i,j)\in E\\ 1 & otherwise\end{cases}$$

Now, suppose that a Hamiltonian cycle ***h*** exists in ***G***. It is clear that the cost of each edge in ***h*** is **0** in ***G′*** as each edge belongs to ***E***. Therefore, ***h*** has a cost of **0** in ***G′***. Thus, if graph ***G*** has a Hamiltonian cycle, then graph ***G′*** has a tour of **0** cost.

Conversely, we assume that ***G′*** has a tour ***h′*** of cost at most **0**. The cost of edges in ***E′*** are **0** and **1** by definition. Hence, each edge must have a cost of **0** as the cost of ***h′*** is **0**. We therefore conclude that ***h′*** contains only edges in ***E***.

We have thus proven that ***G*** has a Hamiltonian cycle, if and only if ***G***′ has a tour of cost at most **0**. TSP is NP-complete.

Reducibility

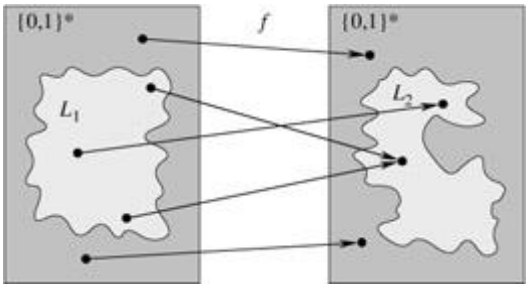
Intuitively, a problem *Q* can be reduced to another problem *Q*′ if any instance of *Q* can be "easily rephrased" as an instance of *Q*′, the solution to which provides a solution to the instance of *Q*. For example, the problem of solving linear equations in an indeterminate *x* reduces to the problem of solving quadratic equations. Given an instance $ax + b = 0$, we transform it to $0x^2 + ax + b = 0$, whose solution provides a solution to $ax + b = 0$. Thus, if a problem *Q* reduces to another problem *Q*′, then *Q* is, in a sense, "no harder to solve" than *Q*′.

Returning to our formal-language framework for decision problems, we say that a language L^1 is ***polynomial-time reducible*** to a language L_2, written $L_1 \leq_P L_2$, if there exists a polynomial-time computable function $f : \{0, 1\}^* \rightarrow \{0,1\}^*$ such that for all $x \{0, 1\}^*$,

(34.1) $x \in L_1$ if and only if $f(x) \in L_2$

We call the function *f* the ***reduction function***, and a polynomial-time algorithm *F* that computes *f* is called a ***reduction algorithm***.

Figure illustrates the idea of a polynomial-time reduction from a language L_1 to another language *L*2. Each language is a subset of {0, 1}*. The reduction function *f* provides a polynomial-time mapping such that if $x \in L_1$, then $f(x) \in L_2$. Moreover, if $x \notin L_1$, then $f(x) \notin L_2$. Thus, the reduction function maps any instance *x* of the decision problem represented by the language L_1 to an instance $f(x)$ of the problem represented by L_2. Providing an answer to whether $f(x) \in L_2$ directly provides the answer to whether $x \in L_1$.



Figure

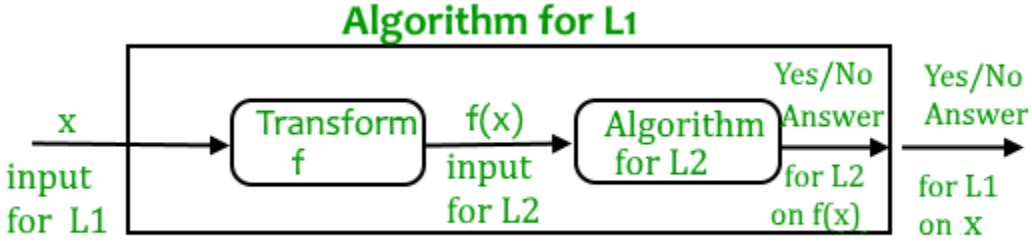
An illustration of a polynomial-time reduction from a language L_1 to a language L_2 via a reduction function f. For any input $x \in \{0, 1\}^*$, the question of whether $x \in L_1$ has the same answer as the question of whether $f(x) \in L_2$.

Polynomial-time reductions give us a powerful tool for proving that various languages belong to P.

What is Reduction?

Let L_1 and L_2 be two decision problems. Suppose algorithm A_2 solves L_2. That is, if y is an input for L_2 then algorithm A_2 will answer Yes or No depending upon whether y belongs to L_2 or not.

The idea is to find a transformation from L_1 to L_2 so that the algorithm A_2 can be part of an algorithm A_1 to solve L_1.



Learning reduction in general is very important. For example, if we have library functions to solve certain problem and if we can reduce a new problem to one of the solved problems, we save a lot of time. Consider the example of a problem where we have to find minimum product path in a given directed graph where

product of path is multiplication of weights of edges along the path. If we have code for Dijkstra's algorithm to find shortest path, we can take log of all weights and use Dijkstra's algorithm to find the minimum product path rather than writing a fresh code for this new problem.

How to prove that a given problem is NP complete?

From the definition of NP-complete, it appears impossible to prove that a problem L is NP-Complete. By definition, it requires us to that show every problem in NP is polynomial time reducible to L. Fortunately, there is an alternate way to prove it. The idea is to take a known NP-Complete problem and reduce it to L. If polynomial time reduction is possible, we can prove that L is NP-Complete by transitivity of reduction (If a NP-Complete problem is reducible to L in polynomial time, then all problems are reducible to L in polynomial time).

What was the first problem proved as NP-Complete?

There must be some first NP-Complete problem proved by definition of NP-Complete problems. SAT (Boolean satisfiability problem) is the first NP-Complete problem proved by Cook.

It is always useful to know about NP-Completeness even for engineers. Suppose you are asked to write an efficient algorithm to solve an extremely important problem for your company. After a lot of thinking, you can only come up exponential time approach which is impractical. If you don't know about NP-Completeness, you can only say that I could not come with an efficient algorithm. If you know about NP-Completeness and prove that the problem as NP-complete, you can proudly say that the polynomial time solution is unlikely to exist. If there is a polynomial time solution possible, then that solution solves a big problem of computer science many scientists have been trying for years.

NUMBER-THEORETIC ALGORITHMS

Number theory was once viewed as a beautiful but largely useless subject in pure mathematics. Today number-theoretic algorithms are used widely, due in part to the invention of cryptographic schemes based on large prime numbers. The

feasibility of these schemes rests on our ability to find large primes easily, while their security rests on our inability to factor the product of large primes. This chapter presents some of the number theory and associated algorithms that underlie such applications.

Section 33.1 introduces basic concepts of number theory, such as divisibility, modular equivalence, and unique factorization. Section 33.2 studies one of the world's oldest algorithms: Euclid's algorithm for computing the greatest common divisor of two integers. Section 33.3 reviews concepts of modular arithmetic. Section 33.4 then studies the set of multiples of a given number a, modulo n, and shows how to find all solutions to the equation $ax \equiv b \pmod{n}$ by using Euclid's algorithm. The Chinese remainder theorem is presented in Section 33.5. Section 33.6 considers powers of a given number a, modulo n, and presents a repeated-squaring algorithm for efficiently computing $a^b \bmod n$, given a, b, and n. This operation is at the heart of efficient primality testing. Section 33.7 then describes the RSA public-key cryptosystem. Section 33.8 describes a randomized primality test that can be used to find large primes efficiently, an essential task in creating keys for the RSA cryptosystem. Finally, Section 33.9 reviews a simple but effective heuristic for factoring small integers. It is a curious fact that factoring is one problem people may wish to be intractable, since the security of RSA depends on the difficulty of factoring large integers.

Size of inputs and cost of arithmetic computations

Because we shall be working with large integers, we need to adjust how we think about the size of an input and about the cost of elementary arithmetic operations.

In this chapter, a "large input" typically means an input containing "large integers" rather than an input containing "many integers" (as for sorting). Thus, we shall measure the size of an input in terms of the *number of bits* required to represent that input, not just the number of integers in the input. An algorithm with integer inputs $a_1, a_2, \ldots, a_k$ is a ***polynomial-time algorithm*** if it runs in time polynomial in

lg a_1, lg a_2, . . . , lg a_k, that is, polynomial in the lengths of its binary-encoded inputs.

In most of this book, we have found it convenient to think of the elementary arithmetic operations (multiplications, divisions, or computing remainders) as primitive operations that take one unit of time. By counting the number of such arithmetic operations an algorithm performs, we have a basis for making a reasonable estimate of the algorithm's actual running time on a computer. Elementary operations can be time-consuming, however, when their inputs are large. It thus becomes convenient to measure how many ***bit operations*** a number-theoretic algorithm requires. In this model, a multiplication of two β-bit integers by the ordinary method uses $\Theta(\beta^2)$ bit operations. Similarly, the operation of dividing a β-bit integer by a shorter integer, or the operation of taking the remainder of a β-bit integer when divided by a shorter integer, can be performed in time $\Theta(\beta^2)$ by simple algorithms. (See Exercise 33.1-11.) Faster methods are known. For example, a simple divide-and-conquer method for multiplying two β-bit integers has a running time of $\Theta(\beta^{\lg 3})$, and the fastest known method has a running time of $\Theta(\beta \lg \beta \lg \lg \beta)$. For practical purposes, however, the $\Theta(\beta^2)$ algorithm is often best, and we shall use this bound as a basis for our analyses.

Elementary number-theoretic notions

This section provides a brief review of notions from elementary number theory concerning the set **Z** = {. . . , -2, -1, 0, 1, 2, . . .} of integers and the set **N** = {0, 1, 2, . . .} of natural numbers.

Divisibility and divisors

The notion of one integer being divisible by another is a central one in the theory of numbers. The notation $d \mid a$ (read "d ***divides*** a") means that $a = kd$ for some integer k. Every integer divides 0. If $a > 0$ and $d \mid a$, then $|d| \leq |a|$. If $d \mid a$, then we also say that a is a ***multiple*** of d. If d does not divide a, we write $d \nmid a$.

If $d \mid a$ and $d \geq 0$, we say that d is a ***divisor*** of a. Note that $d \mid a$ if and only if $-d \mid a$, so that no generality is lost by defining the divisors to be nonnegative, with the understanding that the negative of any divisor of a also divides a. A divisor of an integer a is at least 1 but not greater than $|a|$. For example, the divisors of 24 are 1, 2, 3, 4, 6, 8, 12, and 24.

Every integer a is divisible by the ***trivial divisors*** 1 and a. Nontrivial divisors of a are also called ***factors*** of a. For example, the factors of 20 are 2, 4, 5, and 10.

Prime and composite numbers

An integer $a > 1$ whose only divisors are the trivial divisors 1 and a is said to be a ***prime*** number (or, more simply, a ***prime***). Primes have many special properties and play a critical role in number theory. The small primes, in order, are

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,... .

Exercise 33.1-1 asks you to prove that there are infinitely many primes. An integer $a > 1$ that is not prime is said to be a ***composite*** number (or, more simply, a ***composite***). For example, 39 is composite because $3 \mid 39$. The integer 1 is said to be a ***unit*** and is neither prime nor composite. Similarly, the integer 0 and all negative integers are neither prime nor composite.

The division theorem, remainders, and modular equivalence

Given an integer n, the integers can be partitioned into those that are multiples of n and those that are not multiples of n. Much number theory is based upon a refinement of this partition obtained by classifying the nonmultiples of n according to their remainders when divided by n. The following theorem is the basis for this refinement. The proof of this theorem will not be given here (see, for example, Niven and Zuckerman [151]).

Theorem 1

For any integer a and any positive integer n, there are unique integers q and r such that $0 \leq r < n$ and $a = qn + r$.

The value $q = \lfloor a/n \rfloor$ is the ***quotient*** of the division. The value $r = a \bmod n$ is the ***remainder*** (or ***residue***) of the division. We have that $n \mid a$ if and only if $a \bmod n = 0$. It follows that

$a = \lfloor a/n \rfloor n + (a \bmod n)$ **(1)**

or

$a \bmod n = a - \lfloor a/n \rfloor n$. **(2)**

Given a well-defined notion of the remainder of one integer when divided by another, it is convenient to provide special notation to indicate equality of remainders. If $(a \bmod n) = (b \bmod n)$, we write $a \equiv b \pmod n$ and say that a is ***equivalent*** to b, modulo n. In other words, $a \equiv b \pmod n$ if a and b have the same remainder when divided by n. Equivalently, $a \equiv b \pmod n$ if and only if $n \mid (b - a)$. We write $a \not\equiv b \pmod n$ if a is not equivalent to b, modulo n. For example, $61 \equiv 6 \pmod{11}$. Also, $-13 \equiv 22 \equiv 2 \pmod 5$.

The integers can be divided into n equivalence classes according to their remainders modulo n. The ***equivalence class modulo n*** containing an integer a is

$[a]_n = \{a + kn : k \in \boldsymbol{Z}\}$.

For example, $[3]_7 = \{. . . , -11, -4, 3, 10, 17, . . .\}$; other denotations for this set are $[-4]_7$ and $[10]_7$. Writing $a \in [b]_n$ is the same as writing $a \equiv b \pmod n$. The set of all such equivalence classes is

$\boldsymbol{Z}_n = \{[a]_n : 0 \le a \le n - 1\}$. **(3)**

One often sees the definition

$\boldsymbol{Z}_n = \{0, 1,..., n - 1\}$, **(4)**

which should be read as equivalent to equation (33.3) with the understanding that 0 represents $[0]_n$, 1 represents $[1]_n$, and so on; each class is represented by its least nonnegative element. The underlying equivalence classes must be kept in

mind, however. For example, a reference to - 1 as a member of $\boldsymbol{Z}_n$ is a reference to $[n - 1]_n$, since $-1 \equiv n - 1 \pmod{n}$.

The Fast Fourier Transform

The fast Fourier transform (FFT) is an algorithm which can take the discrete Fourier transform of a array of size $n = 2^N$ in $\boldsymbol{\Theta}(n \ln(n))$ time. This algorithm is generally performed in place and this implementation continues in that tradition. Two implementations are provided:

- The first implementation, FFT.basic.h, emphasizes the algorithm; consequently, to simplify the presentation, it allocates additional memory when dividing the vector into even and odd entries and explicit recursion is used, and
- The second implementation, FFT.array.h, demonstrates how the FFT algorithm can be performed with only $\boldsymbol{O}(1)$ additional memory and without recursive function calls. Reference, Maple 8, Waterloo Maple Inc., Waterloo, Ontario.

Some comments on the code:

The value π is found using double const PI = 4.0*std::atan(1.0); The inverse tangent function of 1 returns π/4 and and multiplication by a power of 2 does not affect the relative error of a floating point number: in this case, multiplication by four simply adds two to the exponent.

The testing function applies the fast Fourier transform to the vector $(2, 2, 0, 0, 0, 0, 0, 0)^T$. It prints the vector, the fast Fourier transform of the vector, and the inverse fast Fourier transform of that result. The output is

(1,0) (1,0) (0,0) (0,0) (0,0) (0,0) (0,0) (0,0)

(2,0) (1.70711,-0.707107) (1,-1) (0.292893,-0.707107) (0,0) (0.292893,0.707107) (1,1) (1.70711,0.707107)

(1,0) (1,0) (0,0) (0,0) (0,0) (0,0) (0,0) (0,0)

You will note that $1/\sqrt{2} \approx 0.707107$, and $1 - 1/\sqrt{2} \approx 0.292893$.

The function printing the complex array will print 0 in place of either component being less than 10^{-15}. This is to make aid in the visualization of the result.

The Discrete Fourier Transform

The discrete Fourier transform maps a vector from the space domain to the complex frequency domain. To describe this, recall that a vector of *n* dimensions represents coordinates in space: the basis vectors are the *n* unit vectors (1, 0, 0, ⋯), (0, 1, 0, ⋯), *etc*. These are shown in Figure 1.

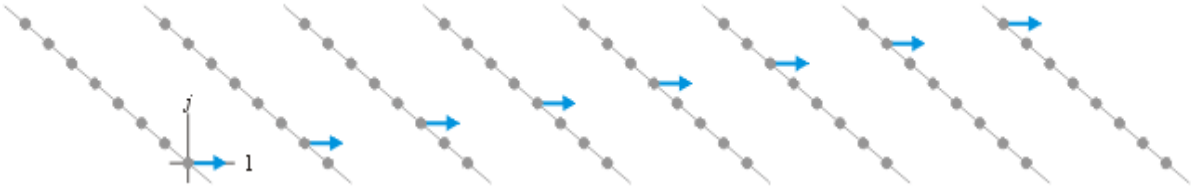


Figure 1. The unit basis vectors for a vector of dimension 8.

While these unit vectors are exceelent for representing the specific location of a vector, they do not suggest any periodicity. The discrete Fourier transforms is a linear transformation from a basis of unit vectors to a basis of vectors which are capture the periodic behaviour of the unit vector: if the copies of the vector were laid end-to-end, what periodicity would there be in the patterns and what are the periods thereof.

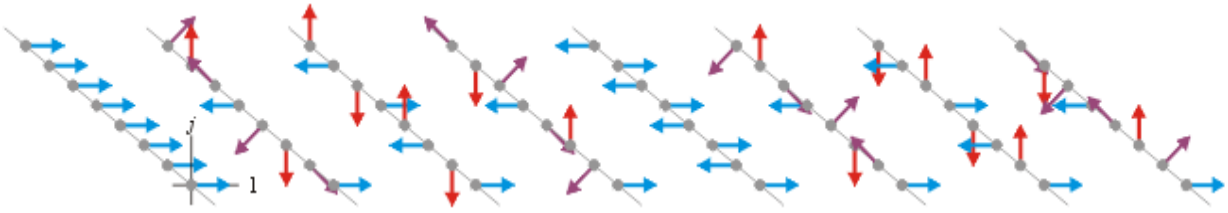


Figure 2. The basis vectors of the frequency space of dimension 8.

template <typename T> class complex<T>

This code uses the STL complex class which has a constructor which takes the real and imaginary parts as arguments. The following functions are friends of the class and for the argument $z = a + jb$:

Function	**Returns**
double abs(z)	$\|z\| = \sqrt{(a^2 + b^2)}$
double arg(z)	$\operatorname{atan2}(\Im(z), \Re(z)) = \operatorname{atan2}(b, a)$
complex conj(z)	$z^* = a - jb$
double imag(z)	$\Im(z) = b$
double norm(z)	$\|z\|^2 = a^2 + b^2$
complex polar(r, θ)	$re^{j\vartheta}$
double real(z)	$\Re(z) = a$
complex exp(z)	e^z
complex log(z)	$\ln(z)$
complex pow(w, z)	w^z
complex sqrt(z)	$\sqrt{z}$
complex sin(z)	$\sin(z)$
complex cos(z)	$\cos(z)$

complex sinh(z)	sinh(*z*)
complex cosh(z)	cosh(*z*)

A complex number *a* + *jb* is printed as the ordered pair (a,b).

Determining Powers of Two

The assertion assert(n > 0 && (n & (~n + 1)) == n); checks to ensure that the argument *n* is both positive and a power of two. The operation n & (~n + 1) selects the least significant one in the integer. If the least significant one is also the most significant one, then the bitwise and will return that value; otherwise, the result will not equal the original value *n*. This is shown in Figure 3 with both a power of two (64) and a number which is not a power of two (84).

```
  00001000000      00001010100
  11110111111      11110101011
  +          1     +          1
& 11111000000    & 11110000100
  00001000000      00000000100
```

Figure 3. Selecting the least significant one (1) bit in a binary number.

String Matching Algorithms

A string is a sequence of characters. In our model we are going to represent a string as a 0-indexed array. So a string S = ”Galois” is indeed an array [‘G’, ’a’, ’l’, ’o’, ’i’, ’s’]. The number of characters of a string is called its length and is denoted by |S|. If we want to reference the character of the string at position i, we will use S[i].

A substring is a sequence of consecutive contiguous elements of a string, we will denote the substring starting at i and ending at j of string S by S[i...j].

A prefix of a string S is a substring that starts at position 0, and a suffix a substring that ends at |S|-1. A proper prefix of a S is a prefix that is different to S. Similarly, a proper suffix of S is a suffix that is different to S. The + operator will represent string concatenation.

Example:

S="Galois"

|S|=6

S[0]='G', S[1]='a', S[2]='l',...,S[5]='s'

S[1...4]="aloi"

W="Evariste"

W+S="EvaristeGalois"

A Needle in the haystack (KMP algorithm)

Given a text T we are interested in calculating all the occurrences of a pattern P.

This simple problem has a lot of applications. For example, the text can be the nucleotide sequence of the human DNA and the pattern a genomic sequence of some genetic disease, or the text can be all the internet, and the pattern a query (the Google problem), etc.

We are going to study the exact string matching problem, that is given two strings T and P we want to find all substrings of T that are equal to P. Formally, we want to calculate all indices i such that T[i+s] = P[s] for each $0 \leq s \leq |P|-1$.

In the following example, you are given a string T and a pattern P, and all the occurrences of P in T are highlighted in red.

P = aat

T = acataaatattttgataacatgaatattaagcagagaattaaaagtgaaatgatatagg

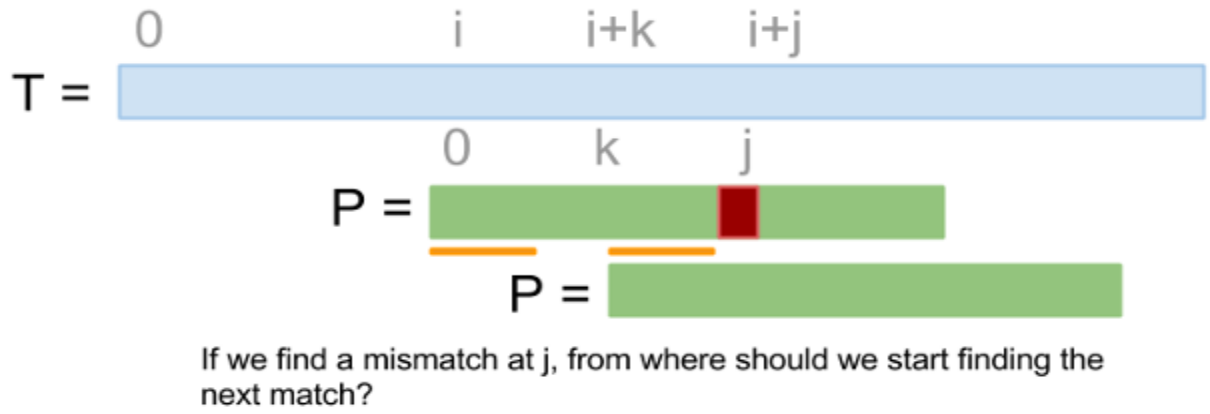
One easy way to solve the problem, is to iterate over all i from 0to |T| - |P|, and check if there is a substring of T that starts at i and matches with P:

```
def find_occurrences(t,p):
  lt,lp=len(t),len(p)
  for i in range(lt-lp+1):
    match=True
    for l in range(lp):
      if t[i+l]!=p[l]:
        match=False
        break
    if match: print i
```

Unfortunately this solution is very slow, its time complexity is O(|T|·|P|), however it gives us some insights.

Suppose that we are finding a match starting at position i on the text, then there are two possibilities:

We don't find a match. Then there exists at least one index in which the text is not equal to the pattern. Let i+j be the smallest of such indices: T[i...i+j-1] = P[0...j-1] and T[i+j] ≠ P[j]



Since there is no match at position i, we should start finding a match from another position, but the question is from where? Our previous algorithm always selects as next position i+1, but if we start from i+1, it is probable that we could end up finding a mismatch in a position even before than i+j-1. At least, we could want to start from a position that guarantees that the strings matches until i+j-1. Therefore, we should start finding for a match starting from the smallest i+k such that T[i+k...i+j-1] matches with some prefix of P (look at the picture above).

Since we already know that T matches with P from i to i+j-1, then T[i+k...i+j-1] is a suffix of P[0...j-1].

That means that if we find a mismatch at position j, we should start from the smallest k such that P[k...j-1] is a prefix of P. k is the smallest, so P[k...j-1] is the largest proper suffix that is also a proper prefix. From now on we will call “border” to the proper prefixes that are also proper suffixes (e.g the string ABCDABCDAB have two borders ABCDAB and AB).

We find a match. Using the same argument, it’s easy to see that we have to start finding for a match from the smallest k such that P[k...j-1]is a proper prefix of P

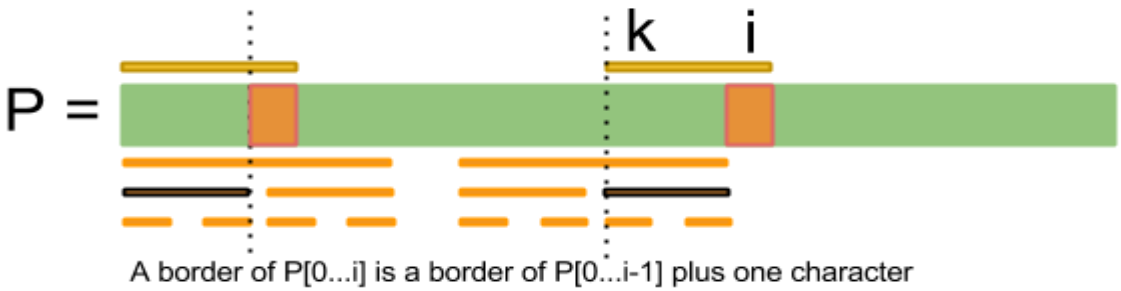
the above two cases with an example:

```
    01234567890123456789012345678
S = HACKHACKHACKHACKITHACKEREARTH
P =     HACKHACKIT
P =         HACKHACKIT...[match!]
P =                     HACKHACKIT
```

In the picture above we are finding occurrences starting from position 4, but there is a mismatch at position 12. Before getting a mismatch we have already matched the string HACKHACK, and the largest proper prefix that is also a proper suffix (border) of HACKHACK is HACK, so we should start finding occurrences again from position 8 and taking into account that all characters from 8 to 11 are already matched. It turns out that there is an occurrence starting at position 8, so all characters of the pattern have matched, since HACKHACKIT does not have any border, then we start finding occurrences again starting from position 18.

According to the previous analysis we need to know for each i, the the largest border of P[0...i]. Let f[i] be the length of the largest border of P[0...i]. Function f is known as failure function, because it says from where start if we find a mismatch.

How can we calculate f[i] efficiently? One approach is to think in an inductive way: suppose that we have already calculated the function f for for all indices less than i. Using that information how can we calculate f[i]?



Note that if P[0...j] is a border of P[0...i], then P[0...j-1] is a border of P[0...i-1] and P[j] = P[i].

The previous argument suggest this algorithm for finding f[i]: Iterate over all borders of P[0...i-1], in decreasing order of length, until find a border P[0...j-1] such that P[j] = P[i]. How can we iterate over all the borders of a string? That can be easily done using this observation: if B is a border of P, then a border of B is also a border of P, that implies that the borders of P[0...i] are prefixes of P that ends at f[i]-1, f[f[i]-1]-1, f[f[...f[i]-1...]-1]-1, ... (at most how many borders can have a string?).

What is the complexity of our algorithm? Let P[j...i-1] be the largest border of P[0...i-1]. If P[k...i] is the largest border of P[0...i], then k ≥ j (Why?). So when we iterate over the borders of P[0...i-1], we are moving the index j to k. Since j is moving always to the right, in the worst case it will touch all the elements of the array. That means that we are calculating function f in O(|P|).

Using function f it is easy to search for the occurrences of a pattern on a text in O(|T|+|P|):

```
def failure_function(p):
  l=len(p)
  f=[0]*l
  j=0
  for i in range(1,l):
    while j>=0 and p[j]!=p[i]:
      if j-1>=0: j=f[j-1]
      else: j=-1
    j+=1
    f[i]=j
  return f
```

```
def find_occurrences(t,p):
  f=failure_function(p)
  lt,lp=len(t),len(p)
  j=0
  for i in range(lt):
    while j>=0 and t[i]!=p[j]:
      if j-1>=0: j=f[j-1]
      else: j=-1
    j+=1
    if j==lp:
      j=f[lp-1]
      print i-lp+1
```

As you can note from the pseudo code (it is python code indeed), find_occurrences is almost equal to failure_function, that is because in some sense failure_function is like matching a string with itself.

The algorithm described above is known as Knut-Morris-Pratt (or KMP for short). Note that with KMP algorithm we don't need to have all the string T in memory, we can read it character by character, and determine all the occurrences of a pattern P in an online way.

The Z function

Given a string S, let z[i] be the longest substring of S that starts at i and is also a prefix of S.

Example:

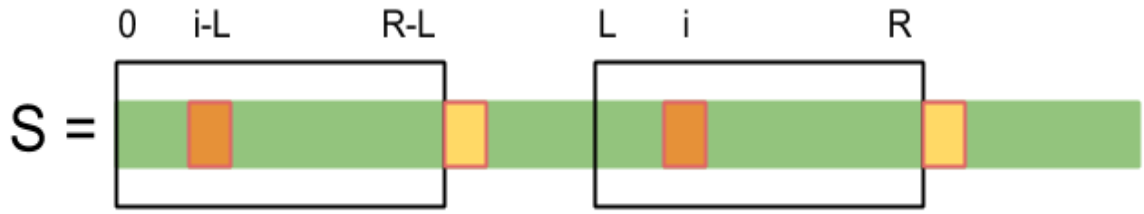
```
S = A B R A B R A C A D A B R A
Z =[0,0,0,4,0,0,1,0,1,0,4,0,0,1]
```

z[3] = 4 because starting from position 3, the largest string that is also a prefix of S is ABRA.

If we can calculate the function Z efficiently, then how can we find all the occurrences of a pattern P on a text T? Since z[i] gives matches with a prefix, a good idea is to observe how the z function behaves when P is concatenated with T. So let's calculate the function z on S = P+T. It turns out that if for certain i, z[i] ≥ |P| then there is an occurrence of P starting at i.

Now only remains to find a way of calculate that powerful z function.

The naive approach for calculate z in every i is to iterate over all j ≥ i until we find a mismatch (z[i...j] ≠ z[0...j-i]), then z[i] = j-i. This algorithm is of quadratic time, so we need a better solution.



The z function applied at position i of a string determines an interval [i, i+z[i]-1] known as z-box, that interval is special, because by the definition of z, S[i...i+z[i]-1] = S[0...z[i]-1].

Now let's think again in a inductive way, and calculate z[i] given that we already know z[0],...,z[i-1].

Let [L,R] be the z-box with largest R that we have seen until now.

If i is inside [L,R], then S[i...R] = S[i-L...R-L] (look at the picture above), so we can use the value of z[i-L] that we have already calculated. If z[i-L] Is less than R-i+1, then z[i] = z[i-L], otherwise since we already know that S[i...R] is a prefix of S, it remains to check if we can expand S[i...R] starting from R+1.

On the other hand If i is outside [L...R], then we can calculate z[i] using the naive approach. This algorithm is linear because the pointer R traverses the array at most once.

```
def zeta(s):
  n=len(s)
  z=[0]*n
  L,R=-1,-1
  for i in range(1,n):
    j,k=0,i
    if L<=i<=R:
      ii = i-L
      j=min(ii+z[ii], R-L+1)-ii
      k=i+j
    while k<n and s[j]==s[k]:
      j+=1
      k+=1
    z[i]=k-i
    if z[i]>0 and i+z[i]-1>R:
```

```
      L,R=i,i+z[i]-1
  return z


def find_occurrences(p,t):
  lp,lt=len(p),len(t)
  z=zeta(p+t)
  for i in range(lp,lp+lt):
    if z[i]>=lp: print i-lp
```

Hashing strikes back (Rabin-Karp algorithm)

Pattern P matches with text T at position i, if and only if there is a substring of T that starts at i and is equal to P. So if we can compare quickly two strings (T[i...i +|P|-1] with P), then we can use our naive algorithm (iterate over all i, and check if there is a match). One way of checking if two strings are not equal is to find a property that is different in those strings. Let h be a black-box that have as input a string, and outputs certain property of the string, that property is defined in such a way that if two strings are different, the value of that property is also different: if S1 ≠ S2 then h(S1) ≠ h(S2). Note that with this definition, we can't assert if two strings are equal, because two different strings can have the same value of the chosen property.

Since numbers are easy to compare it will be nice if h outputs a number. So the question is how to represent a string as a number. Note that strings are like numbers, because we can consider its characters as digits. Since the alphabet have 26 letters, we could say that a string is a number in a numeration system of base 27. The problem is that strings can be very large, and we can represent integers in a very limited range (from 0 to 18446744073709551615 using a 64 bit unsigned integer), in order to keep the output of h in a small range, let's apply the

modulo operation (is because of this last necessary operation that two different strings can map to the same integer i.e there is a hash collision).

Example: if S="hack", then h(S) = (8 · 283 + 1 · 282 + 3 · 281 + 11 · 280) %M.

Note that we are considering the value of the digit "a" as 1 instead of 0, that is to prevent cases with leading zeroes (e.g "aab" and "b" are different, but if a = 0 they are equal).

A function like h, that converts elements from a very large range (like strings) to elements in a small range (64 bit integers) are called hash functions The value of the hash function applied to certain element is called it's hash.

In order to reduce hash collisions, M is usually chosen as a large prime. However if we use an unsigned 32 bits integer we could just let the value overflow (in this case the modulo is 232).

It remains to solve this problem: if we know the hash of the substring (of length |P|) that starts at i, how to calculate the hash of the substring that starts at i+1? (see figure below)



Let h be the hash of the substring of length |P| that starts at i. We can calculate the hash of the substring that starts at i+1 in two steps:

Remove the character at position i:

$$h = h - 26^{|P|-1} * T[i]$$

Add the character at position i+|P|:

$$h = h * 26 + T[i + |P|]$$

The idea of calculate the hash that starts at position i+1 using the hash at i is called rolling hash.

```
def val(ch): #maps a character to a digit e.g a = 1, b = 2,...
  return ord(ch)-ord("a")+1


def find_occurrences(p,t):
  lp,lt=len(p),len(t)
  m=10**9+7 #modulo (a "big" prime)
  b=30      #numeration system base
  hp=0      #hash of the pattern
  ht=0      #hash of a substring of length |P| of the text
  for i in range(lp):
    hp=(hp*b+val(p[i]))%m
    ht=(ht*b+val(t[i]))%m
  pwr=1
  for i in range(lp-1):
    pwr=pwr*b%m


  if hp==ht: print 0
  for i in range(1,lt-lp+1):
```

```
#rolling hash

#remove character i-1:

ht=(ht-val(t[i-1])*pwr)%m

ht=(ht+m)%m

#add character i+|P|-1
```

```
ht=(ht*b+val(t[i+lp-1]))%m

if ht==hp: print i
```

Note that in the code above, we are finding matches with high probability (because of hash collision). It is possible to increase the probability using two modulos, but in programming contests usually one modulo is enough (given a modulo how to generate two different strings with the same hash?).

Parallel Algorithm

An **algorithm** is a sequence of steps that take inputs from the user and after some computation, produces an output. A **parallel algorithm** is an algorithm that can execute several instructions simultaneously on different processing devices and then combine all the individual outputs to produce the final result.

Concurrent Processing

The easy availability of computers along with the growth of Internet has changed the way we store and process data. We are living in a day and age where data is available in abundance. Every day we deal with huge volumes of data that require complex computing and that too, in quick time. Sometimes, we need to fetch data from similar or interrelated events that occur simultaneously. This is where we require **concurrent processing** that can divide a complex task and process it multiple systems to produce the output in quick time.

Concurrent processing is essential where the task involves processing a huge bulk of complex data. Examples include − accessing large databases, aircraft testing, astronomical calculations, atomic and nuclear physics, biomedical analysis, economic planning, image processing, robotics, weather forecasting, web-based services, etc.

What is Parallelism?

Parallelism is the process of processing several set of instructions simultaneously. It reduces the total computational time. Parallelism can be implemented by using **parallel computers,** i.e. a computer with many processors. Parallel computers require parallel algorithm, programming languages, compilers and operating system that support multitasking.

In this tutorial, we will discuss only about **parallel algorithms**. Before moving further, let us first discuss about algorithms and their types.

What is an Algorithm?

An **algorithm** is a sequence of instructions followed to solve a problem. While designing an algorithm, we should consider the architecture of computer on which the algorithm will be executed. As per the architecture, there are two types of computers −

- Sequential Computer
- Parallel Computer

Depending on the architecture of computers, we have two types of algorithms −

- **Sequential Algorithm** − An algorithm in which some consecutive steps of instructions are executed in a chronological order to solve a problem.
- **Parallel Algorithm** − The problem is divided into sub-problems and are executed in parallel to get individual outputs. Later on, these individual outputs are combined together to get the final desired output.

It is not easy to divide a large problem into **sub-problems**. Sub-problems may have data dependency among them. Therefore, the processors have to communicate with each other to solve the problem.

It has been found that the time needed by the processors in communicating with each other is more than the actual processing time. So, while designing a parallel algorithm, proper CPU utilization should be considered to get an efficient algorithm.

To design an algorithm properly, we must have a clear idea of the basic **model of computation** in a parallel computer.

Model of Computation

Both sequential and parallel computers operate on a set (stream) of instructions called algorithms. These set of instructions (algorithm) instruct the computer about what it has to do in each step.

Depending on the instruction stream and data stream, computers can be classified into four categories −

- Single Instruction stream, Single Data stream (SISD) computers
- Single Instruction stream, Multiple Data stream (SIMD) computers
- Multiple Instruction stream, Single Data stream (MISD) computers
- Multiple Instruction stream, Multiple Data stream (MIMD) computers

SISD Computers

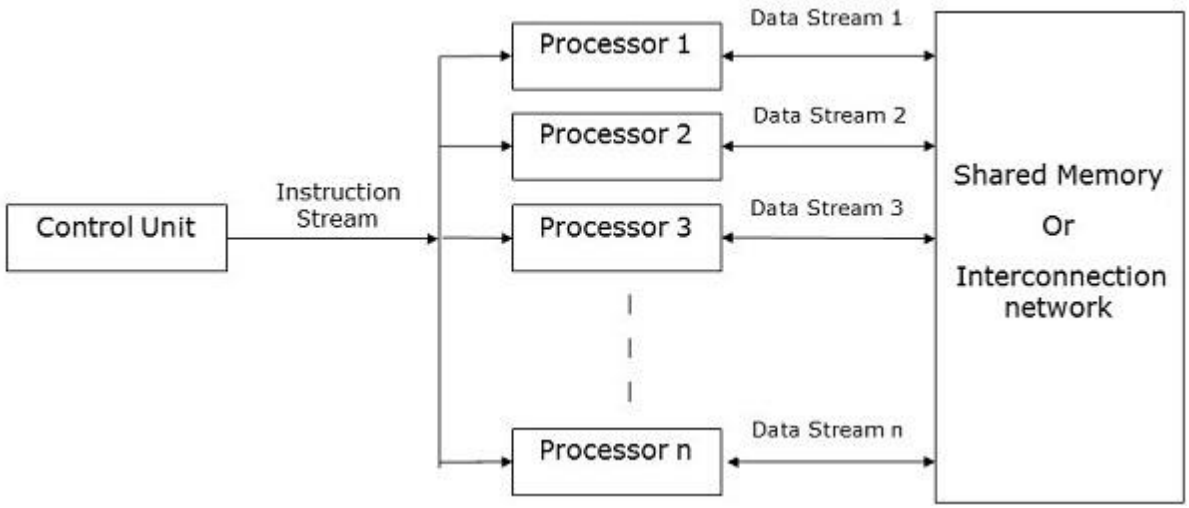
SISD computers contain **one control unit, one processing unit,** and **one memory unit**.



In this type of computers, the processor receives a single stream of instructions from the control unit and operates on a single stream of data from the memory unit. During computation, at each step, the processor receives one instruction from the control unit and operates on a single data received from the memory unit.

SIMD Computers

SIMD computers contain **one control unit, multiple processing units,** and **shared memory or interconnection network**.



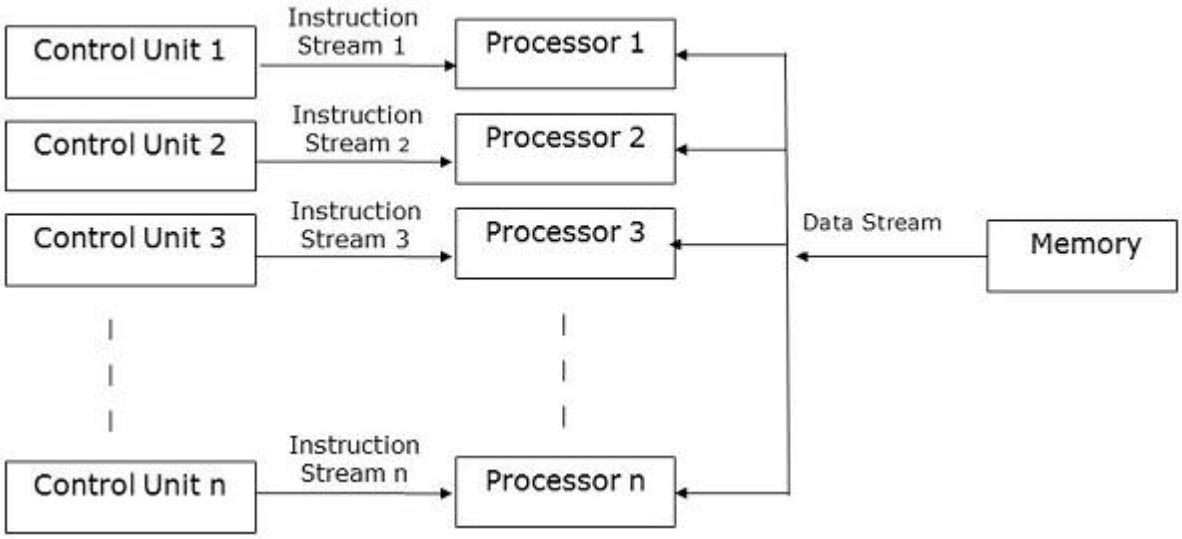
Here, one single control unit sends instructions to all processing units. During computation, at each step, all the processors receive a single set of instructions from the control unit and operate on different set of data from the memory unit.

Each of the processing units has its own local memory unit to store both data and instructions. In SIMD computers, processors need to communicate among themselves. This is done by **shared memory** or by **interconnection network**.

While some of the processors execute a set of instructions, the remaining processors wait for their next set of instructions. Instructions from the control unit decides which processor will be **active** (execute instructions) or **inactive** (wait for next instruction).

MISD Computers

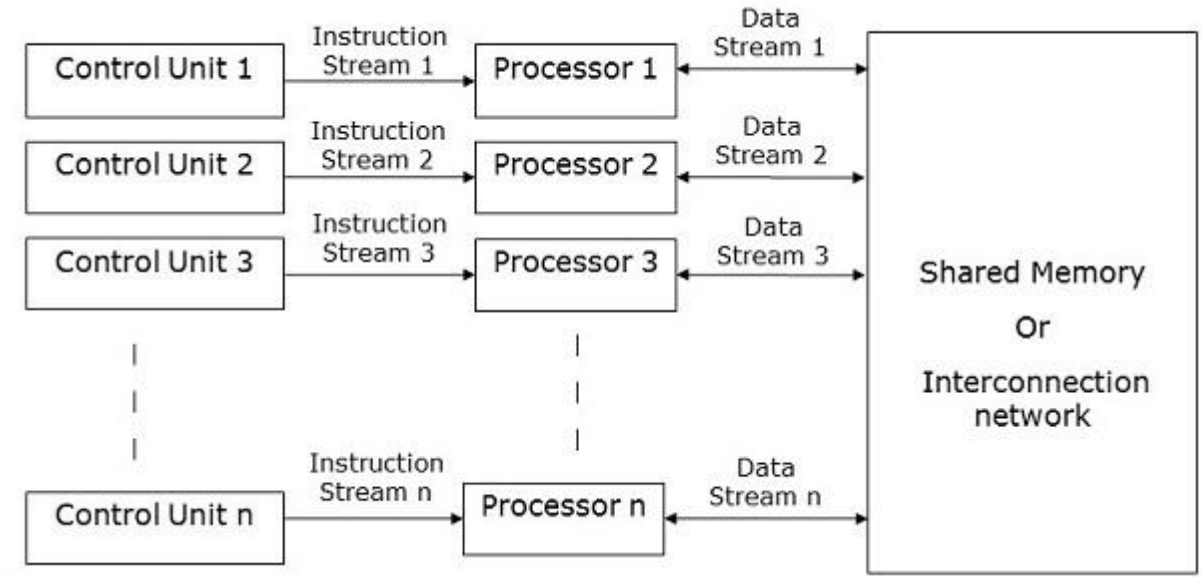
As the name suggests, MISD computers contain **multiple control units, multiple processing units,** and **one common memory unit**.



Here, each processor has its own control unit and they share a common memory unit. All the processors get instructions individually from their own control unit and they operate on a single stream of data as per the instructions they have received from their respective control units. This processor operates simultaneously.

MIMD Computers

MIMD computers have **multiple control units, multiple processing units,** and a **shared memory** or **interconnection network**.



Here, each processor has its own control unit, local memory unit, and arithmetic and logic unit. They receive different sets of instructions from their respective control units and operate on different sets of data.

Note

- An MIMD computer that shares a common memory is known as **multiprocessors,** while those that uses an interconnection network is known as **multicomputers**.
- Based on the physical distance of the processors, multicomputers are of two types −
    - **Multicomputer** − When all the processors are very close to one another (e.g., in the same room).
    - **Distributed system** − When all the processors are far away from one another (e.g.- in the different cities)

Parallel Algorithm - Sorting

Sorting is a process of arranging elements in a group in a particular order, i.e., ascending order, descending order, alphabetic order, etc. Here we will discuss the following −

- Enumeration Sort
- Odd-Even Transposition Sort
- Parallel Merge Sort
- Hyper Quick Sort

Sorting a list of elements is a very common operation. A sequential sorting algorithm may not be efficient enough when we have to sort a huge volume of data. Therefore, parallel algorithms are used in sorting.

Enumeration Sort

Enumeration sort is a method of arranging all the elements in a list by finding the final position of each element in a sorted list. It is done by comparing each element with all other elements and finding the number of elements having smaller value.

Therefore, for any two elements, a_i and a_j any one of the following cases must be true −

- $a_i < a_j$
- $a_i > a_j$
- $a_i = a_j$

Algorithm

```
procedure ENUM_SORTING (n)


begin
```

```
for each process P1,j do
  C[j] := 0;

for each process Pi, j do

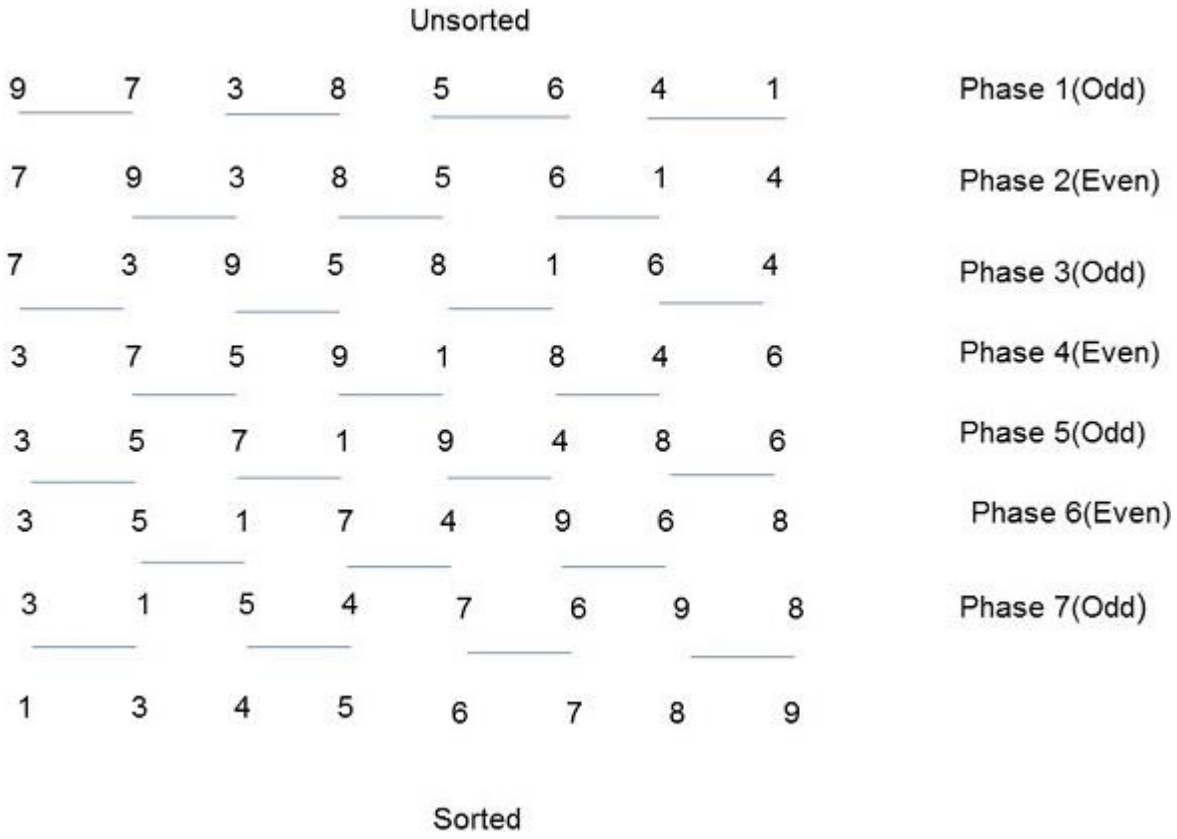
  if (A[i] < A[j]) or A[i] = A[j] and i < j) then
    C[j] := 1;
  else
    C[j] := 0;

for each process P1, j do
  A[C[j]] := A[j];

end ENUM_SORTING
```

Odd-Even Transposition Sort

Odd-Even Transposition Sort is based on the Bubble Sort technique. It compares two adjacent numbers and switches them, if the first number is greater than the second number to get an ascending order list. The opposite case applies for a descending order series. Odd-Even transposition sort operates in two phases – **odd phase** and **even phase**. In both the phases, processes exchange numbers with their adjacent number in the right.



Algorithm

```
procedure ODD-EVEN_PAR (n)


begin
  id := process's label
  for i := 1 to n do
  begin


    if i is odd and id is odd then
      compare-exchange_min(id + 1);
```

```
    else
      compare-exchange_max(id - 1);

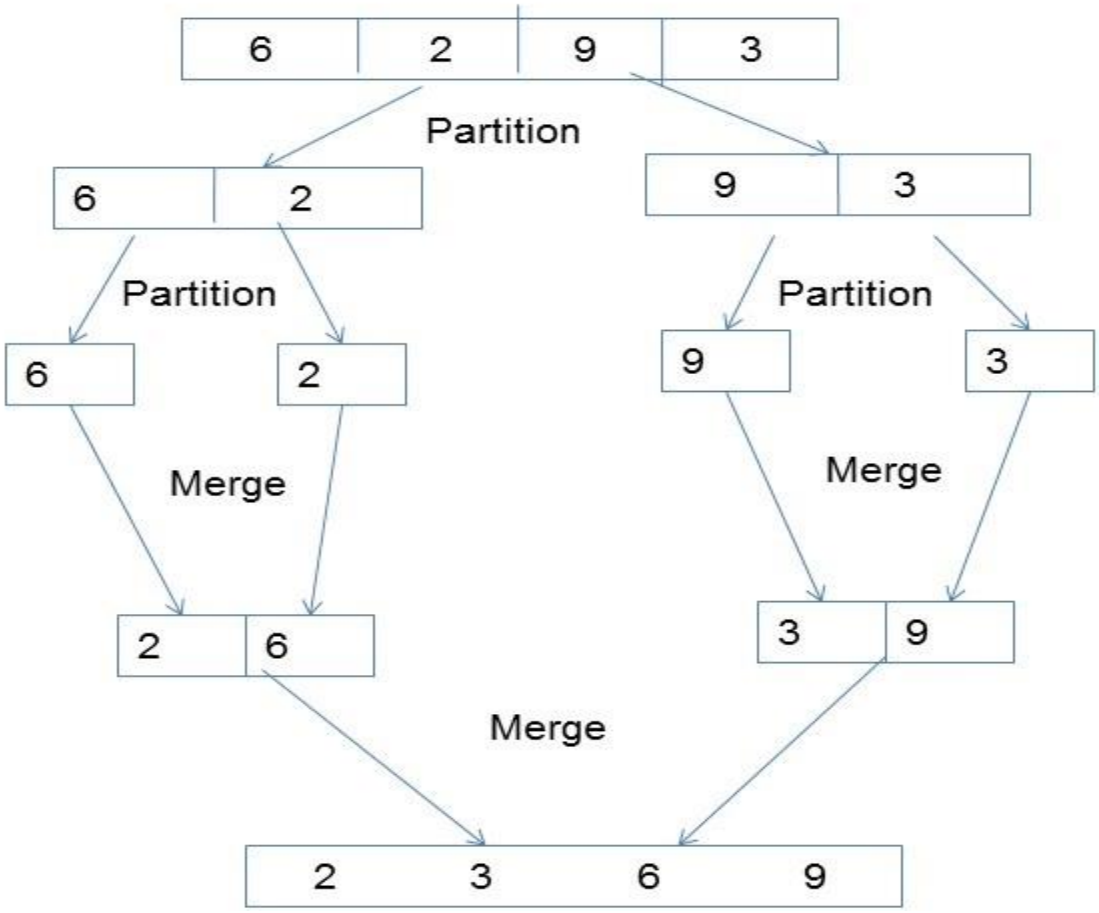
    if i is even and id is even then
      compare-exchange_min(id + 1);
    else
      compare-exchange_max(id - 1);

  end for

end ODD-EVEN_PAR
```

Parallel Merge Sort

Merge sort first divides the unsorted list into smallest possible sub-lists, compares it with the adjacent list, and merges it in a sorted order. It implements parallelism very nicely by following the divide and conquer algorithm.



Algorithm

```
procedureparallelmergesort(id, n, data, newdata)


begin
  data = sequentialmergesort(data)


    for dim = 1 to n
      data = parallelmerge(id, dim, data)
    endfor
```

```
  newdata = data
end
```

Hyper Quick Sort

Hyper quick sort is an implementation of quick sort on hypercube. Its steps are as follows −

- Divide the unsorted list among each node.
- Sort each node locally.
- From node 0, broadcast the median value.
- Split each list locally, then exchange the halves across the highest dimension.
- Repeat steps 3 and 4 in parallel until the dimension reaches 0.

Algorithm

procedure HYPERQUICKSORT (B, n)

begin

```
  id := process's label;
  for i := 1 to d do
    begin
    x := pivot;
    partition B into B1 and B2 such that B1 ≤ x < B2;
    if ith bit is 0 then
```

```
    begin
      send B2 to the process along the ith communication link;
      C := subsequence received along the ith communication link;
      B := B1 U C;
    endif

    else
      send B1 to the process along the ith communication link;
      C := subsequence received along the ith communication link;
      B := B2 U C;
      end else
    end for

  sort B using sequential quicksort;
end HYPERQUICKSORT
```

Approximate Algorithms

An Approximate Algorithm is a way of approach **NP-COMPLETENESS** for the optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time. Such algorithms are called approximation algorithm or heuristic algorithm.

- For the traveling salesperson problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find a short cycle.
- For the vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices, and the approximation problem is to find the vertex cover with few vertices.

Performance Ratios

Suppose we work on an optimization problem where every solution carries a cost. An Approximate Algorithm returns a legal solution, but the cost of that legal solution may not be optimal.

For Example, suppose we are considering for a **minimum size vertex-cover (VC)**. An approximate algorithm returns a VC for us, but the size (cost) may not be minimized.

Another Example is we are considering for a **maximum size Independent set (IS)**. An approximate Algorithm returns an IS for us, but the size (cost) may not be maximum. Let C be the cost of the solution returned by an approximate algorithm, and C* is the cost of the optimal solution.

We say the approximate algorithm has an approximate ratio P (n) for an input size n, where

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq P(n)$$

Intuitively, the approximation ratio measures how bad the approximate solution is distinguished with the optimal solution. A large (small) approximation ratio measures the solution is much worse than (more or less the same as) an optimal solution.

Observe that P (n) is always ≥ 1, if the ratio does not depend on n, we may write P. Therefore, a 1-approximation algorithm gives an optimal solution. Some problems have polynomial-time approximation algorithm with small constant approximate ratios, while others have best-known polynomial time approximation algorithms whose approximate ratios grow with

Randomized Algorithms

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm. For example, in Randomized Quick Sort, we use random number to pick the next pivot (or we randomly shuffle the array). Typically, this randomness is used to reduce time complexity or space complexity in other standard algorithms.

What is a Randomized Algorithm?

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm.. For example, in Randomized Quick Sort, we use random number to pick the next pivot (or we randomly shuffle the array). And in Karger's algorithm, we randomly pick an edge.

How to analyse Randomized Algorithms?

Some randomized algorithms have deterministic time complexity. For example, this implementation of Karger's algorithm has time complexity as O(E). Such algorithms are called Monte Carlo Algorithms and are easier to analyse for worst case.
On the other hand, time complexity of other randomized algorithms (other than Las Vegas) is dependent on value of random variable. Such Randomized algorithms are called Las Vegas Algorithms. These algorithms are typically analysed for expected worst case. To compute expected time taken in worst case, all possible values of the used random variable needs to be considered in worst case and time taken by every possible value needs to be evaluated. Average of all evaluated times is the expected worst case time complexity. Below facts are

generally helpful in analysis os such algorithms.
Linearity of Expectatio
Expected Number of Trials until Success.

For example consider below a randomized version of QuickSort.

A **Central Pivot** is a pivot that divides the array in such a way that one side has at-least 1/4 elements.

// Sorts an array arr[low..high]

randQuickSort(arr[], low, high)

1. If low >= high, then EXIT.

2. While pivot 'x' is not a Central Pivot.

(i) Choose uniformly at random a number from [low..high].

Let the randomly picked number number be **x**.

(ii) Count elements in arr[low..high] that are smaller

than arr[x]. Let this count be **sc**.

(iii) Count elements in arr[low..high] that are greater

than arr[x]. Let this count be **gc**.

(iv) Let **n** = (high-low+1). If sc >= n/4 and

gc >= n/4, then x is a central pivot.

3. Partition arr[low..high] around the pivot x.

4. // Recur for smaller elements

randQuickSort(arr, low, sc-1)

5. // Recur for greater elements

randQuickSort(arr, high-gc+1, high)

The important thing in our analysis is, time taken by step 2 is O(n).

How many times while loop runs before finding a central pivot?
The probability that the randomly chosen element is central pivot is 1/2.

Therefore, expected number of times the while loop runs is 2

Thus, the expected time complexity of step 2 is O(n).

What is overall Time Complexity in Worst Case?
In worst case, each partition divides array such that one side has n/4 elements and other side has 3n/4 elements. The worst case height of recursion tree is $\text{Log}_{3/4}$ n which is O(Log n).

$T(n) < T(n/4) + T(3n/4) + O(n)$

$T(n) < 2T(3n/4) + O(n)$

Solution of above recurrence is O(n Log n)

Note that the above randomized algorithm is not the best way to implement randomized Quick Sort. The idea here is to simplify the analysis as it is simple to analyse.
Typically, randomized Quick Sort is implemented by randomly picking a pivot (no loop). Or by shuffling array elements.

Searching

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

There are two popular search methods that are widely used in order to search some item into the list. However, choice of the algorithm depends upon the arrangement of the list.

- Linear Search
- Binary Search

Linear Search

Linear search is the simplest search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match found then location of the item is returned otherwise the algorithm return NULL.

Linear search is mostly used to search an unordered list in which the items are not sorted. The algorithm of linear search is given as follows.

Algorithm

- LINEAR_SEARCH(A, N, VAL)
- **Step 1:** [INITIALIZE] SET POS = -1
- **Step 2:** [INITIALIZE] SET I = 1
- **Step 3:** Repeat Step 4 while I<=N
- **Step 4:** IF A[I] = VAL
  SET POS = I
  PRINT POS
  Go to Step 6

[END OF IF]
SET I = I + 1
[END OF LOOP]

- **Step 5:** IF POS = -1
  PRINT " VALUE IS NOT PRESENTIN THE ARRAY "
  [END OF IF]
- **Step 6:** EXIT

Complexity of algorithm

Complexity	Best Case	Average Case	Worst Case
Time	O(1)	O(n)	O(n)
Space			O(1)

Binary Search

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.

Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Binary search algorithm is given below.

BINARY_SEARCH(A, lower_bound, upper_bound, VAL)

- **Step 1:** [INITIALIZE] SET BEG = lower_bound
  END = upper_bound, POS = - 1
- **Step 2:** Repeat Steps 3 and 4 while BEG <=END
- **Step 3:** SET MID = (BEG + END)/2
- **Step 4:** IF A[MID] = VAL
  SET POS = MID
  PRINT POS
  Go to Step 6
  ELSE IF A[MID] > VAL
  SET END = MID - 1
  ELSE
  SET BEG = MID + 1
  [END OF IF]
  [END OF LOOP]
- **Step 5:** IF POS = -1
  PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
  [END OF IF]
- **Step 6:** EXIT

Complexity

SN	Performance	Complexity
1	Worst case	O(log n)

2	Best case	O(1)
3	Average Case	O(log n)
4	Worst case space complexity	O(1)

Example

Let us consider an array arr = {1, 5, 7, 8, 13, 19, 20, 23, 29}. Find the location of the item 23 in the array.

In 1st step :

1. BEG = 0
2. END = 8ron
3. MID = 4
4. a[mid] = a[4] = 13 < 23, therefore

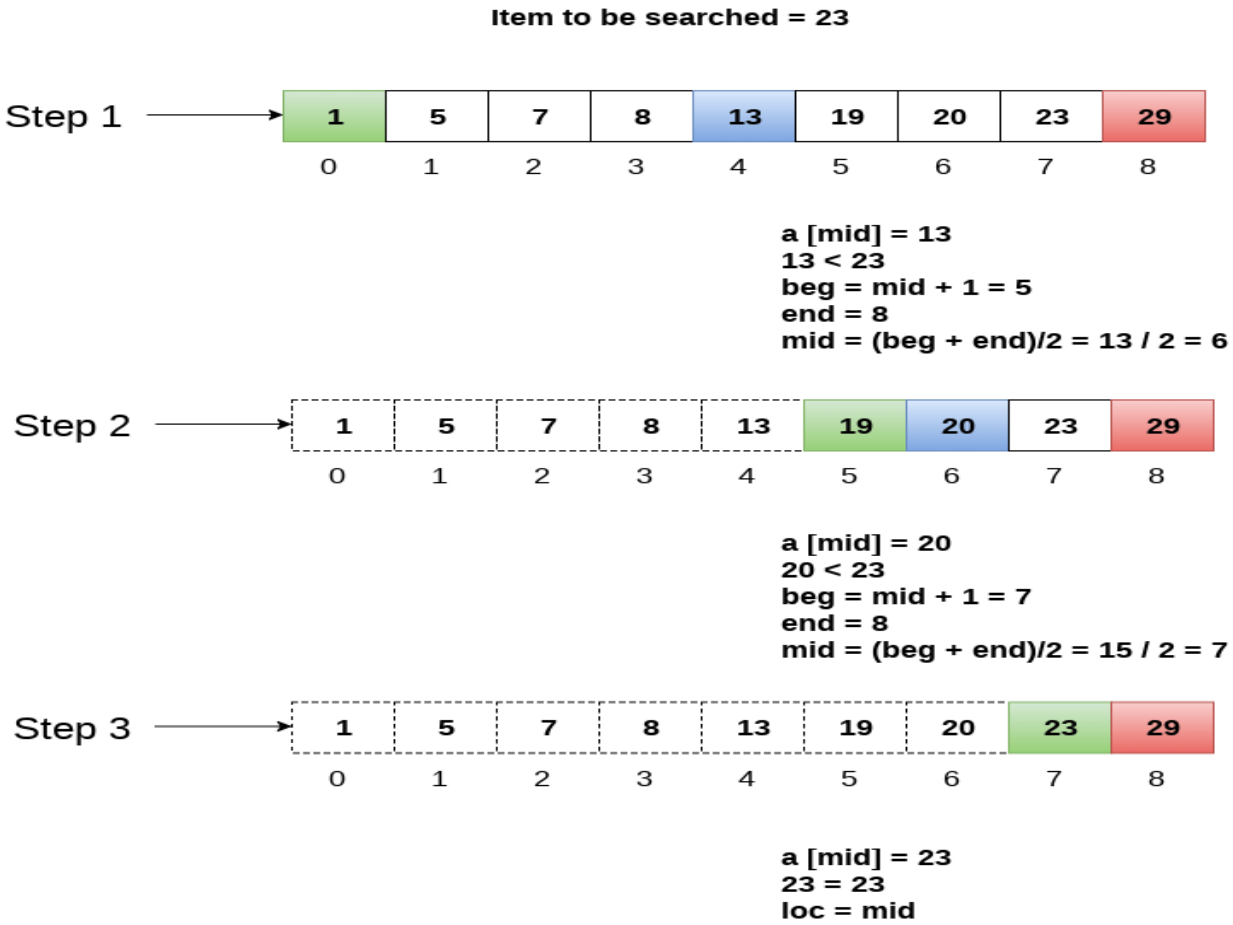
in Second step:

1. Beg = mid +1 = 5
2. End = 8
3. mid = 13/2 = 6
4. a[mid] = a[6] = 20 < 23, therefore;

in third step:

1. beg = mid + 1 = 7
2. End = 8

3. mid = 15/2 = 7
4. a[mid] = a[7]
5. a[7] = 23 = item;
6. therefore, set location = mid;
7. The location of the item will be 7.



DIWAKAR EDUCATION HUB

Data Structures and Algorithms Unit – 7 MCQs

As per updated syllabus

DIWAKAR EDUCATION HUB

2020

THE LEARN WITH EXPERTIES

1. Which line should be inserted in the blank to complete the following dynamic programming implementation of the maximum sub-array sum problem?

```
#include<stdio.h>
int max_num(int a,int b)
{
      if(a> b)
         return a;
      return b;
}
int maximum_subarray_sum(int *arr, int len)
{
      int sum[len], idx;
      sum[0] = arr[0];
      for(idx = 1; idx < len; idx++)
         sum[idx] = _____;
      int mx = sum[0];
      for(idx = 0; idx < len; idx++)
         if(sum[idx] > mx)
             mx =sum[idx];
         return mx;
}
int main()
{
      int arr[] = {-2, -5, 6, -2, 3, -1, 0,-5, 6}, len = 9;
      int ans = maximum_subarray_sum(arr, len);
      printf("%d",ans);
      return 0;
}
```

a) max_num(sum[idx – 1] + arr[idx], arr[idx])
b) sum[idx – 1] + arr[idx].
c) min_num(sum[idx – 1] + arr[idx], arr[idx])
d) arr[idx].

Answer: a
Explanation: The array “sum” is used to store the maximum sub-array sum. The appropriate way to do this is by using: sum[idx] = max_num(sum[idx – 1] + arr[idx], arr[idx]).

2. What is the space complexity of the following dynamic programming algorithm used to find the maximum sub-array sum?

```
#include<stdio.h>
int max_num(int a,int b)
{
      if(a> b)
         return a;
      return b;
}
int maximum_subarray_sum(int *arr, int len)
{
      int sum[len], idx;
      sum[0] = arr[0];
      for(idx = 1; idx < len; idx++)
         sum[idx] = max_num(sum[idx - 1] + arr[idx], arr[idx]);
      int mx = sum[0];
      for(idx = 0; idx < len; idx++)
         if(sum[idx] > mx)
             mx =sum[idx];
         return mx;
}
int main()
{
      int arr[] = {-2, -5, 6, -2, 3, -1, 0,-5, 6}, len = 9;
      int ans = maximum_subarray_sum(arr, len);
      printf("%d",ans);
      return 0;
}
```

a) O(n)
b) O(1)
c) O(n!)
d) O(n^2)

Answer: a
Explanation: The above dynamic programming algorithm uses space equal to the length of the array to store the sum values. So, the space complexity is O(n).

3. Consider the following code snippet:

```
1. int sum[len], idx;
2. sum[0] = arr[0];
3. for(idx = 1; idx < len; idx++)
4.       sum[idx] = max(sum[idx - 1] + arr[idx], arr[idx]);
5. int mx = sum[0];
6. for(idx = 0; idx < len; idx++)
7.      if(sum[idx] > mx)
8.            mx =sum[idx];
9. return mx;
```

Which method is used by line 4 of the above code snippet?
a) Divide and conquer
b) Recursion
c) Both memoization and divide and conquer
d) Memoization

Answer: d
Explanation: The array “sum” is used to store the previously calculated values, so that they aren’t recalculated. So, line 4 uses the memoization technique.

4. Find the maximum sub-array sum for the following array:
{3, 6, 7, 9, 3, 8}
a) 33
b) 36
c) 23
d) 26

Answer: b
Explanation: All the elements of the array are positive. So, the maximum sub-array sum is equal to the sum of all the elements, which is 36.

5. Which of these best describes an array?
a) A data structure that shows a hierarchical behaviour
b) Container of objects of similar types
c) Arrays are immutable once initialised
d) Array is not a data structure

Answer: b
Explanation: Array contains elements only of the same type.

6. How do you initialize an array in C?
a) int arr[3] = (1,2,3);
b) int arr(3) = {1,2,3};
c) int arr[3] = {1,2,3};
d) int arr(3) = (1,2,3);

Answer: c
Explanation: This is the syntax to initialize an array in C.

7. How do you instantiate an array in Java?
a) int arr[] = new int(3);
b) int arr[];
c) int arr[] = new int[3];
d) int arr() = new int(3);

Answer: c
Explanation: Note that option b is declaration whereas option c is to instantiate an array.

8. Which of the following is a correct way to declare a multidimensional array in Java?
a) int[] arr;
b) int arr[[]];
c) int[][]arr;
d) int[[]] arr;

Answer: c
Explanation: The syntax to declare multidimensional array in java is either int[][] arr; or int arr[][];

9. What is the output of the following piece of code?

```
public class array
{
        public static void main(String
args[])
        {
                int []arr = {1,2,3,4,5};

        System.out.println(arr[2]);

        System.out.println(arr[4]);
        }
}
```

a) 3 and 5
b) 5 and 3
c) 2 and 4
d) 4 and 2

Answer: a
Explanation: Array indexing starts from 0.

10. What is the output of the following piece of code?

```
public class array
{
        public static void main(String
args[])
        {
                int []arr = {1,2,3,4,5};

        System.out.println(arr[5]);
        }
}
```

a) 4
b) 5
c) ArrayIndexOutOfBoundsException
d) InavlidInputException

Answer: c
Explanation: Trying to access an element beyond the limits of an array gives ArrayIndexOutOfBoundsException.

11. When does the Array Index Out Of Bounds Exception occur?
a) Compile-time
b) Run-time
c) Not an error
d) Not an exception at all

Answer: b
Explanation: ArrayIndexOutOfBoundsException is a run-time exception and the compilation is error-free.

12. Which of the following concepts make extensive use of arrays?
a) Binary trees
b) Scheduling of processes
c) Caching
d) Spatial locality

Answer: d
Explanation: Whenever a particular memory location is referred, it is likely that the locations nearby are also referred, arrays are stored as contigous blocks in memory, so if you want to access array elements, spatial locality makes it to access quickly.

13. What are the advantages of arrays?
a) Objects of mixed data types can be stored
b) Elements in an array cannot be sorted
c) Index of first element of an array is 1
d) Easier to store elements of same data type

Answer: d
Explanation: Arrays stores elements of

same data type and present in continuous memory locations.

14. What are the disadvantages of arrays?
a) Data structure like queue or stack cannot be implemented
b) There are chances of wastage of memory space if elements inserted in an array are lesser than the allocated size
c) Index value of an array can be negative
d) Elements are sequentially accessed

Answer: b
Explanation: Arrays are of fixed size. If we insert elements less than the allocated size, unoccupied positions can't be used again. Wastage will occur in memory.

15. Assuming int is of 4bytes, what is the size of int arr[15];?
a) 15
b) 19
c) 11
d) 60

Answer: d
Explanation: Since there are 15 int elements and each int is of 4bytes, we get 15*4 = 60bytes.

16. In general, the index of the first element in an array is _____
a) 0
b) -1
c) 2
d) 1

Answer: a
Explanation: In general, Array Indexing starts from 0. Thus, the index of the first element in an array is 0.

17. Elements in an array are accessed _____
a) randomly
b) sequentially
c) exponentially
d) logarithmically

Answer: a
Explanation: Elements in an array are accessed randomly. In Linked lists, elements are accessed sequentially.

18. Process of inserting an element in stack is called _____
a) Create
b) Push
c) Evaluation
d) Pop

Answer: b
Explanation: Push operation allows users to insert elements in stack. If stack is filled completely and trying to perform push operation stack – overflow can happen.

19. Process of removing an element from stack is called _____
a) Create
b) Push
c) Evaluation
d) Pop

Answer: d
Explanation: Elements in stack are removed using pop operation. Pop operation removes the top most element in the stack i.e. last entered element.

20. In a stack, if a user tries to remove an element from empty stack it is called _____
a) Underflow
b) Empty collection

c) Overflow
d) Garbage Collection

Answer: a
Explanation: Underflow occurs when the user performs a pop operation on an empty stack. Overflow occurs when the stack is full and the user performs a push operation. Garbage Collection is used to recover the memory occupied by objects that are no longer used.

21. Pushing an element into stack already having five elements and stack size of 5, then stack becomes
a) Overflow
b) Crash
c) Underflow
d) User flow

Answer: a
Explanation: The stack is filled with 5 elements and pushing one more element causes a stack overflow. This results in overwriting memory, code and loss of unsaved work on the computer.

22. Entries in a stack are "ordered". What is the meaning of this statement?
a) A collection of stacks is sortable
b) Stack entries may be compared with the '<' operation
c) The entries are stored in a linked list
d) There is a Sequential entry that is one by one

Answer: d
Explanation: In stack data structure, elements are added one by one using push operation. Stack follows LIFO Principle i.e. Last In First Out(LIFO).

23. Which of the following applications may use a stack?
a) A parentheses balancing program
b) Tracking of local variables at run time
c) Compiler Syntax Analyzer
d) Data Transfer between two asynchronous process

Answer: d
Explanation: Data transfer between the two asynchronous process uses the queue data structure for synchronisation. The rest are all stack applications.

24. Consider the usual algorithm for determining whether a sequence of parentheses is balanced.
The maximum number of parentheses that appear on the stack AT ANY ONE TIME when the algorithm analyzes: (()(()(()))) are:
a) 1
b) 2
c) 3
d) 4 or more

Answer: c
Explanation: In the entire parenthesis balancing method when the incoming token is a left parenthesis it is pushed into stack. A right parenthesis makes pop operation to delete the elements in stack till we get left parenthesis as top most element. 3 elements are there in stack before right parentheses comes. Therefore, maximum number of elements in stack at run time is 3.

25. Consider the usual algorithm for determining whether a sequence of parentheses is balanced.
Suppose that you run the algorithm on a sequence that contains 2 left parentheses and 3 right parentheses (in some order).
The maximum number of parentheses that appear on the stack AT ANY ONE TIME during the computation?
a) 1
b) 2
c) 3
d) 4 or more

Answer: b
Explanation: In the entire parenthesis balancing method when the incoming token is a left parenthesis it is pushed into stack. A right parenthesis makes pop operation to delete the elements in stack till we get left parenthesis as top most element. 2 left parenthesis are pushed whereas one right parenthesis removes one of left parenthesis. 2 elements are there before right parenthesis which is the maximum number of elements in stack at run time.

26. What is the value of the postfix expression 6 3 2 4 + – *:
a) 1
b) 40
c) 74
d) -1

Answer: d
Explanation: Postfix Expression is (6+(3-(2*4))) which results -18 as output.

27. Here is an infix expression: 4 + 3*(6*3-12). Suppose that we are using the usual stack algorithm to convert the expression from infix to postfix notation.
The maximum number of symbols that will appear on the stack AT ONE TIME during the conversion of this expression?
a) 1
b) 2
c) 3
d) 4

Answer: d
Explanation: When we perform the conversion from infix to postfix expression +, *, (, * symbols are placed inside the stack. A maximum of 4 symbols are identified during the entire conversion.

28. The postfix form of the expression (A+ B)*(C*D- E)*F / G is?
a) AB+ CD*E – FG /**
b) AB + CD* E – F **G /
c) AB + CD* E – *F *G /
d) AB + CDE * – * F *G /

Answer: c
Explanation: (((A+ B)*(C*D- E)*F) / G) is converted to postfix expression as
(AB+(*(C*D- E)*F)/ G)
(AB+CD*E-*F) / G
(AB+CD*E-*F * G/). Thus Postfix expression is AB+CD*E-*F*G/

29. The data structure required to check whether an expression contains balanced parenthesis is?
a) Stack
b) Queue
c) Array
d) Tree

Answer: a
Explanation: The stack is a simple data structure in which elements are added and removed based on the LIFO principle. Open parenthesis is pushed into the stack and a closed parenthesis pops out elements till the top element of the stack is its corresponding open parenthesis. If the stack is empty, parenthesis is balanced otherwise it is unbalanced.

30. What data structure would you mostly likely see in a non recursive implementation of a recursive algorithm?
a) Linked List
b) Stack
c) Queue
d) Tree

Answer: b
Explanation: In recursive algorithms, the order in which the recursive process comes back is the reverse of the order in which it goes forward during execution. The compiler uses the stack data structure to implement recursion. In the forwarding phase, the values of local variables, parameters and the return address are pushed into the stack at each recursion level. In the backing-out phase, the stacked address is popped and used to execute the rest of the code.

31. The process of accessing data stored in a serial access memory is similar to manipulating data on a _____
a) Heap
b) Binary Tree
c) Array
d) Stack

Answer: d
Explanation: In serial access memory data records are stored one after the other in which they are created and are accessed sequentially. In stack data structure, elements are accessed sequentially. Stack data structure resembles the serial access memory.

32. The postfix form of A*B+C/D is?
a) *AB/CD+
b) AB*CD/+
c) A*BC+/D
d) ABCD+/*

Answer: b
Explanation: Infix expression is (A*B)+(C/D)
AB*+(C/D)
AB*CD/+. Thus postfix expression is AB*CD/+.

33. Which data structure is needed to convert infix notation to postfix notation?
a) Branch
b) Tree
c) Queue
d) Stack

Answer: d
Explanation: The Stack data structure is used to convert infix expression to postfix expression. The purpose of stack is to reverse the order of the operators in the expression. It also serves as a storage structure, as no operator can be printed until both of its operands have appeared.

34. The prefix form of A-B/ (C * D ^ E) is?
a) -/*^ACBDE
b) -ABCD*^DE
c) -A/B*C^DE
d) -A/BC*^DE

Answer: c
Explanation: Infix Expression is (A-B)/(C*D^E)
(-A/B)(C*D^E)
-A/B*C^DE. Thus prefix expression is -A/B*C^DE

35. What is the result of the following operation?
Top (Push (S, X))
a) X
b) X+S
c) S
d) XS

Answer: a
Explanation: The function Push(S,X) pushes the value X in the stack S. Top() function gives the value which entered last. X entered into stack S at last.

36. The prefix form of an infix expression (p + q) – (r * t) is?
a) + pq – *rt
b) – +pqr * t
c) – +pq * rt
d) – + * pqrt

Answer: c
Explanation: Given Infix Expression is ((p+q)-(r*t))
(+pq)-(r*t)
(-+pq)(r*t)
-+pq*rt. Thus prefix expression is -+pq*rt.

37. Which data structure is used for implementing recursion?
a) Queue
b) Stack
c) Array
d) List

Answer: b
Explanation: Stacks are used for the implementation of Recursion.

38. The result of evaluating the postfix expression 5, 4, 6, +, *, 4, 9, 3, /, +, * is?
a) 600
b) 350
c) 650
d) 588

Answer: b
Explanation: The postfix expression is evaluated using stack. We will get the infix expression as
(5*(4+6))*(4+9/3). On solving the Infix Expression, we get
(5*(10))*(4+3)
= 50*7
= 350.

39. Convert the following infix expressions into its equivalent postfix expressions
(A + B ∧D)/(E – F)+G
a) (A B D ∧ + E F – / G +)
b) (A B D +∧ E F – / G +)
c) (A B D ∧ + E F/- G +)
d) (A B D E F + ∧ / – G +)

Answer: a
Explanation: The given infix expression is (A + B ∧D)/(E – F)+G.
(A B D ^ +) / (E – F) +G

(A B D ^ + E F –) + G. '/' is present in stack.
A B D ^ + E F – / G +. Thus Postfix Expression is A B D ^ + E F – / G +.

40. Convert the following Infix expression to Postfix form using a stack
x + y * z + (p * q + r) * s, Follow usual precedence rule and assume that the expression is legal.
a) xyz*+pq*r+s*+
b) xyz*+pq*r+s+*
c) xyz+*pq*r+s*+
d) xyzp+**qr+s*+

Answer: a
Explanation: The Infix Expression is x + y * z + (p * q + r) * s.
(x y z) + (p * q + r) * s. '+', '*' are present in stack.
(x y z * + p q * r) * s. '+' is present in stack.
x y z * + p q * r + s * +. Thus Postfix Expression is x y z * + p q * r + s * +.

41. Which of the following statement(s) about stack data structure is/are NOT correct?
a) Linked List are used for implementing Stacks
b) Top of the Stack always contain the new node
c) Stack is the FIFO data structure
d) Null link is present in the last node at the bottom of the stack

Answer: c
Explanation: Stack follows LIFO.

42. Consider the following operation performed on a stack of size 5.
Push(1);
Pop();
Push(2);
Push(3);
Pop();
Push(4);
Pop();
Pop();
Push(5);
After the completion of all operation, the number of elements present in stack are
a) 1
b) 2
c) 3
d) 4

Answer: a
Explanation: Number of elements present in stack is equal to the difference between number of push operations and number of pop operations. Number of elements is 5-4=1.

43. Which of the following is not an inherent application of stack?
a) Reversing a string
b) Evaluation of postfix expression
c) Implementation of recursion
d) Job scheduling

Answer: d
Explanation: Job Scheduling is not performed using stacks.

44. The type of expression in which operator succeeds its operands is?
a) Infix Expression
b) Prefix Expression
c) Postfix Expression
d) Both Prefix and Postfix Expressions

Answer: c
Explanation: The expression in which operator succeeds its operands is called postfix expression. The expression in which operator precedes the operands is called prefix expression. If an operator is present between two operands, then it is called infix expressions.

45. Assume that the operators +,-, X are left associative and ^ is right associative.
The order of precedence (from highest to lowest) is ^, X, +, -. The postfix expression for the infix expression a + b X c – d ^ e ^ f is
a) abc X+ def ^^ –
b) abc X+ de^f^ –
c) ab+c Xd – e ^f^
d) -+aXbc^ ^def

Answer: b
Explanation: Given Infix Expression is a + b X c – d ^ e ^ f.
(a b c X +) (d ^ e ^ f). '–' is present in stack.
(a b c X + d e ^ f ^ -). Thus the final expression is (a b c X + d e ^ f ^ -).

46. If the elements "A", "B", "C" and "D" are placed in a stack and are deleted one at a time, what is the order of removal?
a) ABCD
b) DCBA
c) DCAB
d) ABDC

Answer: b
Explanation: Stack follows LIFO(Last In First Out). So the removal order of elements are DCBA.

47. A linear list of elements in which deletion can be done from one end (front) and insertion can take place only at the other end (rear) is known as a ?
a) Queue
b) Stack
c) Tree
d) Linked list

Answer: a
Explanation: Linear list of elements in which deletion is done at front side and insertion at rear side is called Queue. In stack we will delete the last entered element first.

48. The data structure required for Breadth First Traversal on a graph is?
a) Stack
b) Array
c) Queue
d) Tree

Answer: c
Explanation: In Breadth First Search Traversal, BFS, starting vertex is first taken and adjacent vertices which are unvisited are also taken. Again, the first vertex which was added as an unvisited adjacent vertex list will be considered to add further unvisited vertices of the graph. To get first unvisited vertex we need to follows First In First Out principle. Queue uses FIFO principle.

49. A queue follows _____
a) FIFO (First In First Out) principle
b) LIFO (Last In First Out) principle
c) Ordered array
d) Linear tree

Answer: a
Explanation: Element first added in queue will be deleted first which is FIFO principle.

50. Circular Queue is also known as

_____

a) Ring Buffer
b) Square Buffer
c) Rectangle Buffer
d) Curve Buffer

Answer: a
Explanation: Circular Queue is also called as Ring Buffer. Circular Queue is a linear data structure in which last position is connected back to the first position to make a circle. It forms a ring structure.

51. If the elements “A”, “B”, “C” and “D” are placed in a queue and are deleted one at a time, in what order will they be removed?
a) ABCD
b) DCBA
c) DCAB
d) ABDC

Answer: a
Explanation: Queue follows FIFO approach. i.e. First in First Out Approach. So, the order of removal elements are ABCD.

52. A data structure in which elements can be inserted or deleted at/from both the ends but not in the middle is?
a) Queue
b) Circular queue
c) Dequeue
d) Priority queue

Answer: c
Explanation: In dequeuer, we can insert or delete elements from both the ends. In queue, we will follow first in first out principle for insertion and deletion of elements. Element with least priority will be deleted in a priority queue.

53. A normal queue, if implemented using an array of size MAX_SIZE, gets full when
a) Rear = MAX_SIZE – 1
b) Front = (rear + 1)mod MAX_SIZE
c) Front = rear + 1
d) Rear = front

Answer: a
Explanation: When Rear = MAX_SIZE – 1, there will be no space left for the elements to be added in queue. Thus queue becomes full.

54. Queues serve major role in

_____

a) Simulation of recursion
b) Simulation of arbitrary linked list
c) Simulation of limited resource allocation
d) Simulation of heap sort

Answer: c
Explanation: Simulation of recursion uses stack data structure. Simulation of arbitrary linked lists uses linked lists. Simulation of resource allocation uses queue as first entered data needs to be given first priority during resource allocation. Simulation of heap sort uses heap data structure.

55. Which of the following is not the type of queue?
a) Ordinary queue

b) Single ended queue
c) Circular queue
d) Priority queue

Answer: b
Explanation: Queue always has two ends. So, single ended queue is not the type of queue.

56. A linear collection of data elements where the linear node is given by means of pointer is called?
a) Linked list
b) Node list
c) Primitive list
d) Unordered list

Answer: a
Explanation: In Linked list each node has its own data and the address of next node. These nodes are linked by using pointers. Node list is an object that consists of a list of all nodes in a document with in a particular selected set of nodes.

57. Consider an implementation of unsorted singly linked list. Suppose it has its representation with a head pointer only. Given the representation, which of the following operation can be implemented in O(1) time?
i) Insertion at the front of the linked list
ii) Insertion at the end of the linked list
iii) Deletion of the front node of the linked list
iv) Deletion of the last node of the linked list
a) I and II
b) I and III
c) I, II and III
d) I, II and IV

Answer: b
Explanation: We know the head node in the given linked list. Insertion and deletion of elements at the front of the linked list completes in O (1) time whereas for insertion and deletion at the last node requires to traverse through every node in the linked list. Suppose there are n elements in a linked list, we need to traverse through each node. Hence time complexity becomes O(n).

58. In linked list each node contain minimum of two fields. One field is data field to store the data second field is?
a) Pointer to character
b) Pointer to integer
c) Pointer to node
d) Node

Answer: c
Explanation: Each node in a linked list contains data and a pointer (reference) to the next node. Second field contains pointer to node.

59. What would be the asymptotic time complexity to add a node at the end of singly linked list, if the pointer is initially pointing to the head of the list?
a) O(1)
b) O(n)
c) θ(n)
d) θ(1)

Answer: c
Explanation: In case of a linked list having n

elements, we need to travel through every node of the list to add the element at the end of the list. Thus asymptotic time complexity is θ(n).

60. What would be the asymptotic time complexity to insert an element at the front of the linked list (head is known)?
a) O(1)
b) O(n)
c) $O(n^2)$
d) $O(n^3)$

Answer: b
Explanation: To add an element at the front of the linked list, we will create a new node which holds the data to be added to the linked list and pointer which points to head position in the linked list. The entire thing happens within O (1) time. Thus the asymptotic time complexity is O (1).

61. What would be the asymptotic time complexity to find an element in the linked list?
a) O(1)
b) O(n)
c) $O(n^2)$
d) $O(n^4)$

Answer: b
Explanation: If the required element is in the last position, we need to traverse the entire linked list. This will take O (n) time to search the element.

62. What would be the asymptotic time complexity to insert an element at the second position in the linked list?
a) O(1)
b) O(n)
c) $O(n^2)$
d) $O(n^3)$

Answer: a
Explanation: A new node is created with the required element. The pointer of the new node points the node to which the head node of the linked list is also pointing. The head node pointer is changed and it points to the new node which we created earlier. The entire process completes in O (1) time. Thus the asymptotic time complexity to insert an element in the second position of the linked list is O (1).

63. The concatenation of two list can performed in O(1) time. Which of the following variation of linked list can be used?
a) Singly linked list
b) Doubly linked list
c) Circular doubly linked list
d) Array implementation of list

Answer: c
Explanation: We can easily concatenate two lists in O (1) time using singly or doubly linked list, provided that we have a pointer to the last node at least one of the lists. But in case of circular doubly linked lists, we will break the link in both the lists and hook them together. Thus circular doubly linked list concatenates two lists in O (1) time.

64. Consider the following definition in c programming language

```
struct node
{
```

```
  int data;
  struct node * next;
}
typedef struct node NODE;
NODE *ptr;
```

Which of the following c code is used to create new node?
a) ptr = (NODE*)malloc(sizeof(NODE));
b) ptr = (NODE*)malloc(NODE);
c) ptr = (NODE*)malloc(sizeof(NODE*));
d) ptr = (NODE)malloc(sizeof(NODE));

Answer: a
Explanation: As it represents the right way to create a node.

65.What kind of linked list is best to answer question like “What is the item at position n?”
a) Singly linked list
b) Doubly linked list
c) Circular linked list
d) Array implementation of linked list

Answer: d
Explanation: Arrays provide random access to elements by providing the index value within square brackets. In the linked list, we need to traverse through each element until we reach the nth position. Time taken to access an element represented in arrays is less than the singly, doubly and circular linked lists. Thus, array implementation is used to access the item at the position n.

66. Linked lists are not suitable to for the implementation of?
a) Insertion sort
b) Radix sort
c) Polynomial manipulation
d) Binary search

Answer: d
Explanation: It cannot be implemented using linked lists.

67. Linked list is considered as an example of _____ type of memory allocation.
a) Dynamic
b) Static
c) Compile time
d) Heap

Answer: a
Explanation: As memory is allocated at the run time.

68. In Linked List implementation, a node carries information regarding _____
a) Data
b) Link
c) Data and Link
d) Node

Answer: b
Explanation: A linked list is a collection of objects linked together by references from an object to another object. By convention these objects are names as nodes. Linked list consists of nodes where each node contains one or more data fields and a reference(link) to the next node.

69. Linked list data structure offers considerable saving in _____
a) Computational Time
b) Space Utilization
c) Space Utilization and Computational Time
d) Speed Utilization

Answer: c
Explanation: Linked lists saves both space and time.

70. Which of the following points is/are not true about Linked List data structure when it is compared with array?
a) Arrays have better cache locality that can make them better in terms of performance
b) It is easy to insert and delete elements in Linked List
c) Random access is not allowed in a typical implementation of Linked Lists
d) Access of elements in linked list takes less time than compared to arrays

Answer: d
Explanation: To access an element in a linked list, we need to traverse every element until we reach the desired element. This will take more time than arrays as arrays provide random access to its elements.

71. What does the following function do for a given Linked List with first node as head?

```
void fun1(struct node* head)
{
   if(head == NULL)
   return;
   fun1(head->next);
   printf("%d  ", head->data);
}
```

a) Prints all nodes of linked lists
b) Prints all nodes of linked list in reverse order
c) Prints alternate nodes of Linked List
d) Prints alternate nodes in reverse order

Answer: b
Explanation: fun1() prints the given Linked List in reverse manner.
For Linked List 1->2->3->4->5, fun1() prints 5->4->3->2->1.

72. Which of the following sorting algorithms can be used to sort a random linked list with minimum time complexity?
a) Insertion Sort
b) Quick Sort
c) Heap Sort
d) Merge Sort

Answer: d
Explanation: Both Merge sort and Insertion sort can be used for linked lists. The slow random-access performance of a linked list makes other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible. Since worst case time complexity of Merge Sort is O(nLogn) and Insertion sort is $O(n^2)$, merge sort is preferred.

73. What is the output of following function for start pointing to first node of following linked list?

```
1->2->3->4->5->6
void fun(struct node* start)
{
   if(start == NULL)
   return;
   printf("%d  ", start->data);
   if(start->next != NULL )
   fun(start->next->next);
   printf("%d  ", start->data);
}
```

a) 1 4 6 6 4 1
b) 1 3 5 1 3 5

c) 1 2 3 5
d) 1 3 5 5 3 1

Answer: d
Explanation: fun() prints alternate nodes of the given Linked List, first from head to end, and then from end to head.
If Linked List has even number of nodes, then skips the last node.

74. The following C function takes a simply-linked list as input argument.
It modifies the list by moving the last element to the front of the list and returns the modified list. Some part of the code is left blank. Choose the correct alternative to replace the blank line.

```
typedef struct node
{
  int value;
  struct node *next;
}Node;

Node *move_to_front(Node *head)
{
  Node *p, *q;
  if ((head == NULL: || (head->next ==
NULL))
  return head;
  q = NULL; p = head;
  while (p-> next !=NULL)
  {
    q = p;
    p = p->next;
  }
  _____________________________
 return head;
}
```

a) q = NULL; p->next = head; head = p;
b) q->next = NULL; head = p; p->next = head;
c) head = p; p->next = q; q->next = NULL;
d) q->next = NULL; p->next = head; head = p;

Answer: d
Explanation: When while loop completes its execution, node 'p' refers to the last node whereas the 'q' node refers to the node before 'p' in the linked list. q->next=NULL makes q as the last node. p->next=head places p as the first node. the head must be modified to 'p' as 'p' is the starting node of the list (head=p). Thus the sequence of steps are q->next=NULL, p->next=head, head=p.

75. The following C function takes a single-linked list of integers as a parameter and rearranges the elements of the list.
The function is called with the list containing the integers 1, 2, 3, 4, 5, 6, 7 in the given order. What will be the contents of the list after the function completes execution?

```
struct node
{
  int value;
  struct node *next;
};
void rearrange(struct node *list)
{
  struct node *p, * q;
  int temp;
  if ((!list) || !list->next)
    return;
  p = list;
  q = list->next;
  while(q)
  {
      temp = p->value;
      p->value = q->value;
      q->value = temp;
      p = q->next;
```

```
        q = p?p->next:0;
    }
}
```

a) 1, 2, 3, 4, 5, 6, 7
b) 2, 1, 4, 3, 6, 5, 7
c) 1, 3, 2, 5, 4, 7, 6
d) 2, 3, 4, 5, 6, 7, 1

Answer: b
Explanation: The function rearrange() exchanges data of every node with its next node. It starts exchanging data from the first node itself.

76. In the worst case, the number of comparisons needed to search a singly linked list of length n for a given element is
a) log 2 n
b) $^{n}/_{2}$
c) log 2 n – 1
d) n

Answer: d
Explanation: In the worst case, the element to be searched has to be compared with all elements of linked list.

77. Given pointer to a node X in a singly linked list. Only one pointer is given, pointer to head node is not given, can we delete the node X from given linked list?
a) Possible if X is not last node
b) Possible if size of linked list is even
c) Possible if size of linked list is odd
d) Possible if X is not first node

Answer: a
Explanation:
Following are simple steps.

```
struct node *temp  = X->next;
X->data  = temp->data;
X->next  = temp->next;
free(temp);
```

78. You are given pointers to first and last nodes of a singly linked list, which of the following operations are dependent on the length of the linked list?
a) Delete the first element
b) Insert a new element as a first element
c) Delete the last element of the list
d) Add a new element at the end of the list

Answer: c
Explanation: Deletion of the first element of the list is done in O (1) time by deleting memory and changing the first pointer. Insertion of an element as a first element can be done in O (1) time. We will create a node that holds data and points to the head of the given linked list. The head pointer was changed to a newly created node. Deletion of the last element requires a pointer to the previous node of last, which can only be obtained by traversing the list. This requires the length of the linked list. Adding a new element at the end of the list can be done in O (1) by changing the pointer of the last node to the newly created node and last is changed to a newly created node.

79. In the worst case, the number of comparisons needed to search a singly linked list of length n for a given element is
a) log2 n
b) $^{n}/_{2}$
c) log2 n – 1
d) n

Answer: d
Explanation: The worst-case happens if the required element is at last or the element is

absent in the list. For this, we need to compare every element in the linked list. If n elements are there, n comparisons will happen in the worst case.

80. Which of the following is not a disadvantage to the usage of array?
a) Fixed size
b) There are chances of wastage of memory space if elements inserted in an array are lesser than the allocated size
c) Insertion based on position
d) Accessing elements at specified positions

Answer: d
Explanation: Array elements can be accessed in two steps. First, multiply the size of the data type with the specified position, second, add this value to the base address. Both of these operations can be done in constant time, hence accessing elements at a given index/position is faster.

81. What is the time complexity of inserting at the end in dynamic arrays?
a) O(1)
b) O(n)
c) O(logn)
d) Either O(1) or O(n)

Answer: d
Explanation: Depending on whether the array is full or not, the complexity in dynamic array varies. If you try to insert into an array which is not full, then the element is simply stored at the end, this takes O(1) time. If you try to insert into an array which is full, first you will have to allocate an array with double the size of the current array and then copy all the elements into it and finally insert the new element, this takes O(n) time.

82. What is the time complexity to count the number of elements in the linked list?
a) O(1)
b) O(n)
c) O(logn)
d) $O(n^2)$

Answer: b
Explanation: To count the number of elements, you have to traverse through the entire list, hence complexity is O(n).

83. What is the functionality of the following code?

```
public void function(Node node)
{
        if(size == 0)
                head = node;
        else
        {
                Node temp,cur;
                for(cur = head; (temp = cur.getNext())!=null; cur = temp);
                cur.setNext(node);
        }
        size++;
}
```

a) Inserting a node at the beginning of the list
b) Deleting a node at the beginning of the list
c) Inserting a node at the end of the list
d) Deleting a node at the end of the list

Answer: c
Explanation: The for loop traverses through the list and then inserts a new node as cur.setNext(node);

84. What is the space complexity for deleting a linked list?
a) O(1)
b) O(n)

c) Either O(1) or O(n)
d) O(logn)

Answer: a
Explanation: You need a temp variable to keep track of current node, hence the space complexity is O(1).

85. Which of these is not an application of linked list?
a) To implement file systems
b) For separate chaining in hash-tables
c) To implement non-binary trees
d) Random Access of elements

Answer: d
Explanation: To implement file system, for separate chaining in hash-tables and to implement non-binary trees linked lists are used. Elements are accessed sequentially in linked list. Random access of elements is not an applications of linked list.

86. Which of the following piece of code has the functionality of counting the number of elements in the list?
a)

```
public int length(Node head)
{
        int size = 0;
        Node cur = head;
        while(cur!=null)
        {
           size++;
           cur = cur.getNext();
        }
        return size;
}
```

b)

```
public int length(Node head)
{
    int size = 0;
        Node cur = head;
        while(cur!=null)
        {
           cur = cur.getNext();
           size++;
        }
        return size;
}
```

c)

```
public int length(Node head)
{
        int size = 0;
        Node cur = head;
        while(cur!=null)
        {
           size++;
           cur = cur.getNext();
        }
        return size;
}
```

d)

```
public int length(Node head)
{
        int size = 0;
        Node cur = head;
        while(cur!=null)
        {
           size++;
           cur = cur.getNext().getNext();
        }
        return size;
}
```

Answer: a
Explanation: 'cur' pointer traverses through list and increments the size variable until the end of list is reached.

87. How do you insert an element at the

beginning of the list?
a)

```
public void insertBegin(Node node)
{
        node.setNext(head);
        head = node;
        size++;
}
```

b)

```
public void insertBegin(Node node)
{
        head = node;
        node.setNext(head);
        size++;
}
```

c)

```
public void insertBegin(Node node)
{
        Node temp = head.getNext()
        node.setNext(temp);
        head = node;
        size++;
}
```

d)

```
public void insertBegin(Node node)
{
        Node temp = head.getNext()
        node.setNext(temp);
        node = head;
        size++;
}
```

Answer: a
Explanation: Set the 'next' pointer point to the head of the list and then make this new node as the head.

88. What is the functionality of the following piece of code?

```
public int function(int data)
{
        Node temp = head;
        int var = 0;
        while(temp != null)
        {
                if(temp.getData() == data)
                {
                        return var;
                }
                var = var+1;
                temp = temp.getNext();
        }
        return Integer.MIN_VALUE;
}
```

a) Find and delete a given element in the list
b) Find and return the given element in the list
c) Find and return the position of the given element in the list
d) Find and insert a new element in the list

Answer: c
Explanation: When temp is equal to data, the position of data is returned.

89. Which of the following is false about a doubly linked list?
a) We can navigate in both the directions
b) It requires more space than a singly linked list
c) The insertion and deletion of a node take a bit longer
d) Implementing a doubly linked list is easier than singly linked list

Answer: d
Explanation: A doubly linked list has two pointers 'left' and 'right' which enable it to traverse in either direction. Compared to singly liked list which has only a 'next' pointer, doubly linked list requires extra space to store this extra pointer. Every insertion and deletion requires manipulation of two pointers, hence it takes a bit longer time. Implementing doubly linked list involves setting both left and right pointers to correct nodes and takes more time than singly linked list.

90. What is a memory efficient double linked list?
a) Each node has only one pointer to traverse the list back and forth
b) The list has breakpoints for faster traversal
c) An auxiliary singly linked list acts as a helper list to traverse through the doubly linked list
d) A doubly linked list that uses bitwise AND operator for storing addresses

Answer: a
Explanation: Memory efficient doubly linked list has only one pointer to traverse the list back and forth. The implementation is based on pointer difference. It uses bitwise XOR operator to store the front and rear pointer addresses. Instead of storing actual memory address, every node store the XOR address of previous and next nodes.

91. How do you calculate the pointer difference in a memory efficient double linked list?
a) head xor tail
b) pointer to previous node xor pointer to next node
c) pointer to previous node – pointer to next node
d) pointer to next node – pointer to previous node

Answer: b
Explanation: The pointer difference is calculated by taking XOR of pointer to previous node and pointer to the next node.

92. What is the worst case time complexity of inserting a node in a doubly linked list?
a) O(nlogn)
b) O(logn)
c) O(n)
d) O(1)

Answer: c
Explanation: In the worst case, the position to be inserted maybe at the end of the list, hence you have to traverse through the entire list to get to the correct position, hence O(n).

93. Consider the following doubly linked list: head-1-2-3-4-5-tail
What will be the list after performing the given sequence of operations?

```
        Node temp = new
Node(6,head,head.getNext());
        Node temp1 = new
Node(0,tail.getPrev(),tail);
        head.setNext(temp);
        temp.getNext().setPrev(temp);
        tail.setPrev(temp1);
        temp1.getPrev().setNext(temp1);
```

a) head-0-1-2-3-4-5-6-tail
b) head-1-2-3-4-5-6-tail
c) head-6-1-2-3-4-5-0-tail
d) head-0-1-2-3-4-5-tail

Answer: c
Explanation: The given sequence of operations perform addition of nodes at the head and tail of the list.

94. What is the functionality of the following piece of code?

```
public int function()
{
        Node temp = tail.getPrev();
        tail.setPrev(temp.getPrev());
        temp.getPrev().setNext(tail);
        size--;
        return temp.getItem();
}
```

a) Return the element at the tail of the list but do not remove it
b) Return the element at the tail of the list and remove it from the list
c) Return the last but one element from the list but do not remove it
d) Return the last but one element at the tail of the list and remove it from the list

Answer: b
Explanation: The previous and next pointers of the tail and the last but one element are manipulated, this suggests that the last node is being removed from the list.

95. Consider the following doubly linked list: head-1-2-3-4-5-tail
What will be the list after performing the given sequence of operations?

```
        Node temp = new Node(6,head,head.getNext());
        head.setNext(temp);
        temp.getNext().setPrev(temp);
        Node temp1 = tail.getPrev();
        tail.setPrev(temp1.getPrev());
        temp1.getPrev().setNext(tail);
```

a) head-6-1-2-3-4-5-tail
b) head-6-1-2-3-4-tail
c) head-1-2-3-4-5-6-tail
d) head-1-2-3-4-5-tail

Answer: b
Explanation: A new node is added to the head of the list and a node is deleted from the tail end of the list.

96. What is the functionality of the following code? Choose the most appropriate answer.

```
public int function()
{
        if(head == null)
                return Integer.MIN_VALUE;
        int var;
        Node temp = head;
        while(temp.getNext() != head)
                temp = temp.getNext();
        if(temp == head)
        {
                var = head.getItem();
                head = null;
                return var;
        }
        temp.setNext(head.getNext());
        var = head.getItem();
        head = head.getNext();
        return var;
}
```

a) Return data from the end of the list
b) Returns the data and deletes the node at the end of the list
c) Returns the data from the beginning of

the list
d) Returns the data and deletes the node from the beginning of the list

Answer: d
Explanation: First traverse through the list to find the end node, then manipulate the 'next' pointer such that it points to the current head's next node, return the data stored in head and make this next node as the head.

97. What is the functionality of the following code? Choose the most appropriate answer.

```
public int function()
{
        if(head == null)
                return
Integer.MIN_VALUE;
        int var;
        Node temp = head;
        Node cur;
        while(temp.getNext() != head)
        {
                cur = temp;
                temp = temp.getNext();
        }
        if(temp == head)
        {
                var = head.getItem();
                head = null;
                return var;
        }
        var = temp.getItem();
        cur.setNext(head);
        return var;
}
```

a) Return data from the end of the list
b) Returns the data and deletes the node at the end of the list
c) Returns the data from the beginning of the list
d) Returns the data and deletes the node from the beginning of the list

Answer: b
Explanation: First traverse through the list to find the end node, also have a trailing pointer to find the penultimate node, make this trailing pointer's 'next' point to the head and return the data stored in the 'temp' node.

98. Which of the following is false about a circular linked list?
a) Every node has a successor
b) Time complexity of inserting a new node at the head of the list is O(1)
c) Time complexity for deleting the last node is O(n)
d) We can traverse the whole circular linked list by starting from any point

Answer: b
Explanation: Time complexity of inserting a new node at the head of the list is O(n) because you have to traverse through the list to find the tail node.

99. Consider a small circular linked list. How to detect the presence of cycles in this list effectively?
a) Keep one node as head and traverse another temp node till the end to check if its 'next points to head
b) Have fast and slow pointers with the fast pointer advancing two nodes at a time and slow pointer advancing by one node at a time
c) Cannot determine, you have to pre-define if the list contains cycles
d) Circular linked list itself represents a cycle. So no new cycles cannot be generated

Answer: b
Explanation: Advance the pointers in such a way that the fast pointer advances two nodes at a time and slow pointer advances one node at a time and check to see if at any given instant of time if the fast pointer points to slow pointer or if the fast pointer's 'next' points to the slow pointer. This is applicable for smaller lists.

100. What is the functionality of the following piece of code? Select the most appropriate

```
public void function(int data)
{
        int flag = 0;
        if( head != null)
        {
                Node temp =
head.getNext();
                while((temp != head) &&
(!(temp.getItem() == data)))
                {
                        temp =
temp.getNext();
                        flag = 1;
                        break;
                }
        }
        if(flag)

        System.out.println("success");
        else

        System.out.println("fail");
}
```

a) Print success if a particular element is not found
b) Print fail if a particular element is not found
c) Print success if a particular element is equal to 1
d) Print fail if the list is empty
View Answer
Answer: b
Explanation: The function prints fail if the given element is not found. Note that this option is inclusive of option d, the list being empty is one of the cases covered.

101. What is the time complexity of searching for an element in a circular linked list?
a) O(n)
b) O(nlogn)
c) O(1)
d) $O(n^2)$

Answer: a
Explanation: In the worst case, you have to traverse through the entire list of n elements.

102. Which of the following application makes use of a circular linked list?
a) Undo operation in a text editor
b) Recursive function calls
c) Allocating CPU to resources
d) Implement Hash Tables

Answer: c
Explanation: Generally, round robin fashion is employed to allocate CPU time to resources which makes use of the circular linked list data structure. Recursive function calls use stack data structure. Undo Operation in text editor uses doubly linked lists. Hash tables uses singly linked lists.

103. What differentiates a circular linked list from a normal linked list?
a) You cannot have the 'next' pointer point to null in a circular linked list
b) It is faster to traverse the circular linked list
c) You may or may not have the 'next'

pointer point to null in a circular linked list
d) Head node is known in circular linked list

Answer: c
Explanation: The 'next' pointer points to null only when the list is empty, otherwise it points to the head of the list. Every node in circular linked list can be a starting point(head).

104. Which of the following real world scenarios would you associate with a stack data structure?
a) piling up of chairs one above the other
b) people standing in a line to be serviced at a counter
c) offer services based on the priority of the customer
d) tatkal Ticket Booking in IRCTC

Answer: a
Explanation: Stack follows Last In First Out (LIFO) policy. Piling up of chairs one above the other is based on LIFO, people standing in a line is a queue and if the service is based on priority, then it can be associated with a priority queue. Tatkal Ticket Booking Follows First in First Out Policy. People who click the book now first will enter the booking page first.

105. What does the following function check for? (all necessary headers to be included and function is called from main)

```
#define MAX 10

typedef struct stack
{
    int top;
        int item[MAX];
}stack;
```

```
int function(stack *s)
{
    if(s->top == -1)
            return 1;
          else return 0;
}
```

a) full stack
b) invalid index
c) empty stack
d) infinite stack

Answer: c
Explanation: An empty stack is represented with the top-of-the-stack('top' in this case) to be equal to -1.

106. What does 'stack underflow' refer to?
a) accessing item from an undefined stack
b) adding items to a full stack
c) removing items from an empty stack
d) index out of bounds exception

Answer: c
Explanation: Removing items from an empty stack is termed as stack underflow.

107. What is the output of the following program?

```
public class Stack
{
        protected static final int CAPACITY
= 100;
        protected int size,top = -1;
        protected Object stk[];

        public Stack()
        {
                stk = new
Object[CAPACITY];
        }

        public void push(Object item)
```

```
        {
                if(size_of_stack==size)
                {

        System.out.println("Stack
overflow");

        return;
                }
                else
                {
                        top++;
                        stk[top]=item;
                }
        }
        public Object pop()
        {
                if(top<0)
                {
                        return -999;
                }
                else
                {
                        Object
ele=stk[top];
                        top--;
                        size_of_stack--;
                        return ele;
                }
        }
}

public class StackDemo
{
        public static void main(String
args[])
        {
                Stack myStack = new
Stack();
                myStack.push(10);
                Object element1 =
myStack.pop();
                Object element2 =
myStack.pop();
                System.out.println(element2);
                }
}
```

a) stack is full
b) 20
c) 0
d) -999

Answer: d
Explanation: The first call to pop() returns 10, whereas the second call to pop() would result in stack underflow and the program returns -999.

108. What is the time complexity of pop() operation when the stack is implemented using an array?
a) O(1)
b) O(n)
c) O(logn)
d) O(nlogn)

Answer: a
Explanation: pop() accesses only one end of the structure, and hence constant time.

109. Which of the following array position will be occupied by a new element being pushed for a stack of size N elements(capacity of stack > N).
a) S[N-1]
b) S[N]
c) S[1]
d) S[0]

Answer: b
Explanation: Elements are pushed at the end, hence N.

110. What happens when you pop from an empty stack while implementing using the Stack ADT in Java?
a) Undefined error

b) Compiler displays a warning
c) EmptyStackException is thrown
d) NoStackException is thrown

Answer: c
Explanation: The Stack ADT throws an EmptyStackException if the stack is empty and a pop() operation is tried on it.

111. What is the functionality of the following piece of Java code?
Assume: 'a' is a non empty array of integers, the Stack class creates an array of specified size and provides a top pointer indicating TOS(top of stack), push and pop have normal meaning.

```
public void some_function(int[] a)
{
        Stack S=new Stack(a.length);
        int[] b=new int[a.length];
        for(int i=0;i<a.length;i++)
        {
                S.push(a[i]);
        }
        for(int i=0;i<a.length;i++)
        {
                b[i]=(int)(S.pop());
        }
        System.out.println("output :");
        for(int i=0;i<b.length;i++)
        {
                System.out.println(b[i]);
        }
}
```

a) print alternate elements of array
b) duplicate the given array
c) parentheses matching
d) reverse the array

Answer: d
Explanation: Every element from the given array 'a' is pushed into the stack, and then the elements are popped out into the array 'b'. Stack is a LIFO structure, this results in reversing the given array.

112. Array implementation of Stack is not dynamic, which of the following statements supports this argument?
a) space allocation for array is fixed and cannot be changed during run-time
b) user unable to give the input for stack operations
c) a runtime exception halts execution
d) improper program compilation

Answer: a
Explanation: You cannot modify the size of an array once the memory has been allocated, adding fewer elements than the array size would cause wastage of space, and adding more elements than the array size at run time would cause Stack Overflow.

113. Which of the following array element will return the top-of-the-stack-element for a stack of size N elements(capacity of stack > N).
a) S[N-1]
b) S[N]
c) S[N-2]
d) S[N+1]

Answer: a
Explanation: Array indexing start from 0, hence N-1 is the last index.

114. Consider these functions:
push() : push an element into the stack
pop() : pop the top-of-the-stack element
top() : returns the item stored in top-of-the-stack-node
What will be the output after performing these sequence of operations

```
push(20);
push(4);
```

```
top();
pop();
pop();
pop();
push(5);
top();
```

a) 20
b) 4
c) stack underflow
d) 5

Answer: d
Explanation: 20 and 4 which were pushed are popped by the two pop() statements, the recent push() is 5, hence top() returns 5.

115. Which of the following data structures can be used for parentheses matching?
a) n-ary tree
b) queue
c) priority queue
d) stack

Answer: d
Explanation: For every opening brace, push it into the stack, and for every closing brace, pop it off the stack. Do not take action for any other character. In the end, if the stack is empty, then the input has balanced parentheses.

116. Minimum number of queues to implement stack is _____
a) 3
b) 4
c) 1
d) 2

Answer: c
Explanation: Use one queue and one counter to count the number of elements in the queue.

117. What is the best case time complexity of deleting a node in Singly Linked list?
a) O (n)
b) O (n^2)
c) O (nlogn)
d) O (1)

Answer: d
Explanation: Deletion of the head node in the linked list is taken as the best case. The successor of the head node is changed to head and deletes the predecessor of the newly assigned head node. This process completes in O(1) time.

118. Which of the following statements are not correct with respect to Singly Linked List(SLL) and Doubly Linked List(DLL)?
a) Complexity of Insertion and Deletion at known position is O(n) in SLL and O(1) in DLL
b) SLL uses lesser memory per node than DLL
c) DLL has more searching power than SLL
d) Number of node fields in SLL is more than DLL

Answer: d
Explanation: To insert and delete at known positions requires complete traversal of the list in worst case in SLL, SLL consists of an item and a node field, while DLL has an item and two node fields, hence SLL occupies lesser memory, DLL can be traversed both ways(left and right), while SLL can traverse in only one direction, hence more searching power of DLL. Node fields in SLL is 2 (data and address of next node) whereas in DLL is

3(data, address to next node, address to previous node).

119. What does the following function do?

```
public Object some_func()throws
emptyStackException
{
        if(isEmpty())
                throw new
emptyStackException("underflow");
        return first.getEle();
}
```

a) pop
b) delete the top-of-the-stack element
c) retrieve the top-of-the-stack element
d) push operation

Answer: c
Explanation: This code is only retrieving the top element, note that it is not equivalent to pop operation as you are not setting the 'next' pointer point to the next node in sequence.

119. What is the functionality of the following piece of code?

```
public void display()
{
        if(size == 0)

        System.out.println("underflow");
        else
        {
                Node current = first;
                while(current != null)
                {

        System.out.println(current.getEle(
));
                        current =
current.getNext();
                }
        }
}
```

a) reverse the list
b) display the list
c) display the list excluding top-of-the-stack-element
d) reverse the list excluding top-of-the-stack-element

Answer: b
Explanation: An alias of the node 'first' is created which traverses through the list and displays the elements.

120. What does 'stack overflow' refer to?
a) accessing item from an undefined stack
b) adding items to a full stack
c) removing items from an empty stack
d) index out of bounds exception

Answer: b
Explanation: Adding items to a full stack is termed as stack underflow.

121. What is the space complexity of a linear queue having n elements?
a) O(n)
b) O(nlogn)
c) O(logn)
d) O(1)

Answer: a
Explanation: Because there are n elements.

122. Which of the following properties is associated with a queue?
a) First In Last Out
b) First In First Out
c) Last In First Out
d) Last In Last Out

Answer: b
Explanation: Queue follows First In First Out structure.

123. In a circular queue, how do you increment the rear end of the queue?
a) rear++
b) (rear+1) % CAPACITY
c) (rear % CAPACITY)+1
d) rear–

Answer: b
Explanation: Ensures rear takes the values from 0 to (CAPACITY-1).

124. What is the term for inserting into a full queue known as?
a) overflow
b) underflow
c) null pointer exception
d) program won't be compiled

Answer: a
Explanation: Just as stack, inserting into a full queue is termed overflow.

125. What is the time complexity of enqueue operation?
a) O(logn)
b) O(nlogn)
c) O(n)
d) O(1)

Answer: d
Explanation: Enqueue operation is at the rear end, it takes O(1) time to insert a new item into the queue.

126. What does the following piece of code do?

```
public Object function()
{
        if(isEmpty())
        return -999;
        else
        {
                Object high;
                high = q[front];
                return high;
        }
}
```

a) Dequeue
b) Enqueue
c) Return the front element
d) Return the last element

Answer: c
Explanation: q[front] gives the element at the front of the queue, since we are not moving the 'front' to the next element, it is not a dequeue operation.

127. What is the need for a circular queue?
a) effective usage of memory
b) easier computations
c) to delete elements based on priority
d) implement LIFO principle in queues

Answer: a
Explanation: In a linear queue, dequeue operation causes the starting elements of the array to be empty, and there is no way you can use that space, while in a circular queue, you can effectively use that space. Priority queue is used to delete the elements based on their priority. Higher priority elements will be deleted first whereas lower priority elements will be deleted next. Queue data structure always follows FIFO principle.

128. In linked list implementation of queue, if only front pointer is maintained, which of the following operation take worst case linear time?
a) Insertion
b) Deletion

c) To empty a queue
d) Both Insertion and To empty a queue

Answer: d
Explanation: Since front pointer is used for deletion, so worst time for the other two cases.

129. In linked list implementation of a queue, where does a new element be inserted?
a) At the head of link list
b) At the centre position in the link list
c) At the tail of the link list
d) At any position in the linked list

Answer: c
Explanation: Since queue follows FIFO so new element inserted at last.

130. In linked list implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into a NONEMPTY queue?
a) Only front pointer
b) Only rear pointer
c) Both front and rear pointer
d) No pointer will be changed

Answer: b
Explanation: Since queue follows FIFO so new element inserted at last.

131. In linked list implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into EMPTY queue?
a) Only front pointer
b) Only rear pointer
c) Both front and rear pointer
d) No pointer will be changed

Answer: c
Explanation: Since its the starting of queue, so both values are changed.

132. In case of insertion into a linked queue, a node borrowed from the _____ list is inserted in the queue.
a) AVAIL
b) FRONT
c) REAR
d) NULL

Answer: a
Explanation: All the nodes are collected in AVAIL list.

133. In linked list implementation of a queue, from where is the item deleted?
a) At the head of link list
b) At the centre position in the link list
c) At the tail of the link list
d) Node before the tail

Answer: a
Explanation: Since queue follows FIFO so new element deleted from first.

134. In linked list implementation of a queue, the important condition for a queue to be empty is?
a) FRONT is null
b) REAR is null
c) LINK is empty
d) FRONT==REAR-1

Answer: a
Explanation: Because front represents the deleted nodes.

135. The essential condition which is checked before insertion in a linked queue is?
a) Underflow
b) Overflow
c) Front value
d) Rear value

Answer: b
Explanation: To check whether there is space in the queue or not.

136. The essential condition which is checked before deletion in a linked queue is?
a) Underflow
b) Overflow
c) Front value
d) Rear value

Answer: a
Explanation: To check whether there is element in the list or not.

137. Which of the following is true about linked list implementation of queue?
a) In push operation, if new nodes are inserted at the beginning of linked list, then in pop operation, nodes must be removed from end
b) In push operation, if new nodes are inserted at the beginning, then in pop operation, nodes must be removed from the beginning
c) In push operation, if new nodes are inserted at the end, then in pop operation, nodes must be removed from end
d) In push operation, if new nodes are inserted at the end, then in pop operation, nodes must be removed from beginning

Answer: a
Explanation: It can be done by both the methods.

138. With what data structure can a priority queue be implemented?
a) Array
b) List
c) Heap
d) Tree

Answer: d
Explanation: Priority queue can be implemented using an array, a list, a binary search tree or a heap, although the most efficient one being the heap.

139. Which of the following is not an application of priority queue?
a) Huffman codes
b) Interrupt handling in operating system
c) Undo operation in text editors
d) Bayesian spam filter

Answer: c
Explanation: Undo operation is achieved using a stack.

140. What is the time complexity to insert a node based on key in a priority queue?
a) O(nlogn)
b) O(logn)
c) O(n)
d) $O(n^2)$

Answer: c
Explanation: In the worst case, you might have to traverse the entire list.

141. What is not a disadvantage of priority scheduling in operating systems?

a) A low priority process might have to wait indefinitely for the CPU
b) If the system crashes, the low priority systems may be lost permanently
c) Interrupt handling
d) Indefinite blocking

Answer: c
Explanation: The lower priority process should wait until the CPU completes the processing higher priority process. Interrupt handling is an advantage as interrupts should be given more priority than tasks at hand so that interrupt can be serviced to produce desired results.

142. Which of the following is not an advantage of priority queue?
a) Easy to implement
b) Processes with different priority can be efficiently handled
c) Applications with differing requirements
d) Easy to delete elements in any case

Answer: d
Explanation: In worst case, the entire queue has to be searched for the element having highest priority. This will take more time than usual. So deletion of elements is not an advantage.

143. What is the time complexity to insert a node based on position in a priority queue?
a) O(nlogn)
b) O(logn)
c) O(n)
d) $O(n^2)$

Answer: c
Explanation: In the worst case, you might have to traverse the entire list.

144. What is the functionality of the following piece of code?

```
public Object delete_key()
{
        if(count == 0)
        {
                System.out.println("Q is
empty");
                System.exit(0);
        }
        else
        {
                Node cur =
head.getNext();
                Node dup =
cur.getNext();
                Object e = cur.getEle();
                head.setNext(dup);
                count--;
                return e;
        }
}
```

a) Delete the second element in the list
b) Return but not delete the second element in the list
c) Delete the first element in the list
d) Return but not delete the first element in the list

Answer: c
Explanation: A pointer is made to point at the first element in the list and one more to point to the second element, pointer manipulations are done such that the first element is no longer being pointed by any other pointer, its value is returned.

145. What is a dequeue?
a) A queue with insert/delete defined for both front and rear ends of the queue
b) A queue implemented with a doubly linked list
c) A queue implemented with both singly and doubly linked lists
d) A queue with insert/delete defined for front side of the queu

Answer: a
Explanation: A dequeue or a double ended queue is a queue with insert/delete defined for both front and rear ends of the queue.

146. What is the functionality of the following piece of code?

```
public void function(Object item)
{
        Node temp=new Node(item,trail);
        if(isEmpty())
        {
                head.setNext(temp);
                temp.setNext(trail);
        }
        else
        {
                Node
cur=head.getNext();

        while(cur.getNext()!=trail)
                {

        cur=cur.getNext();
                }
                cur.setNext(temp);
        }
        size++;
}
```

a) Insert at the front end of the dequeue
b) Insert at the rear end of the dequeue
c) Fetch the element at the rear end of the dequeue
d) Fetch the element at the front end of the dequeue

Answer: b
Explanation: If the list is empty, this new node will point to 'trail' and will be pointed at by 'head'. Otherwise, traverse till the end of the list and insert the new node there.

147. What are the applications of dequeue?
a) A-Steal job scheduling algorithm
b) Can be used as both stack and queue
c) To find the maximum of all sub arrays of size k
d) To avoid collision in hash tables

Answer: d
Explanation: All of the mentioned can be implemented with a dequeue.

148. What is the time complexity of deleting from the rear end of the dequeue implemented with a singly linked list?
a) O(nlogn)
b) O(logn)
c) O(n)
d) $O(n^2)$

Answer: c
Explanation: Since a singly linked list is used, first you have to traverse till the end, so the complexity is O(n).

149. After performing these set of operations, what does the final list look contain?

```
InsertFront(10);
InsertFront(20);
InsertRear(30);
DeleteFront();
InsertRear(40);
InsertRear(10);
```

```
DeleteRear();
InsertRear(15);
display();
```

a) 10 30 10 15
b) 20 30 40 15
c) 20 30 40 10
d) 10 30 40 15

Answer: d
Explanation: A careful tracing of the given operation yields the result.
10
20 10
20 10 30
10 30
10 30 40
10 30 40 10
10 30 40
10 30 40 15

150. A Double-ended queue supports operations such as adding and removing items from both the sides of the queue. They support four operations like addFront(adding item to top of the queue), addRear(adding item to the bottom of the queue), removeFront(removing item from the top of the queue) and removeRear(removing item from the bottom of the queue). You are given only stacks to implement this data structure. You can implement only push and pop operations. What are the total number of stacks required for this operation?(you can reuse the stack)
a) 1
b) 2
c) 3
d) 4

Answer: b
Explanation: The addFront and removeFront operations can be performed using one stack itself as push and pop are supported (adding and removing element from top of the stack) but to perform addRear and removeRear you need to pop each element from the current stack and push it into another stack, push or pop the element as per the asked operation from this stack and in the end pop elements from this stack to the first stack.

151. You are asked to perform a queue operation using a stack. Assume the size of the stack is some value ‘n’ and there are ‘m’ number of variables in this stack. The time complexity of performing deQueue operation is (Using only stack operations like push and pop)(Tightly bound).
a) O(m)
b) O(n)
c) O(m*n)
d) Data is insufficient

Answer: a
Explanation: To perform deQueue operation you need to pop each element from the first stack and push it into the second stack. In this case you need to pop ‘m’ times and need to perform push operations also ‘m’ times. Then you pop the first element from this second stack (constant time) and pass all the elements to the first stack (as done in the beginning)(‘m-1’ times). Therfore the time complexity is O(m).

152. Consider you have an array of some random size. You need to perform dequeue operation. You can perform it using stack operation (push and pop) or using queue operations itself (enQueue and Dequeue). The output is guaranteed to be same. Find some differences?
a) They will have different time complexities
b) The memory used will not be different
c) There are chances that output might be different
d) No differences

Answer: a
Explanation: To perform operations such as Dequeue using stack operation you need to empty all the elements from the current stack and push it into the next stack, resulting in a O(number of elements) complexity whereas the time complexity of dequeue operation itself is O(1). And there is a need of a extra stack. Therefore more memory is needed.

153. Consider you have a stack whose elements in it are as follows.
5 4 3 2 << top
Where the top element is 2.
You need to get the following stack
6 5 4 3 2 << top
The operations that needed to be performed are (You can perform only push and pop):
a) Push(pop()), push(6), push(pop())
b) Push(pop()), push(6)
c) Push(pop()), push(pop()), push(6)
d) Push(6)

Answer: a
Explanation: By performing push(pop()) on all elements on the current stack to the next stack you get 2 3 4 5 << top.Push(6) and perform push(pop()) you'll get back 6 5 4 3 2 << top. You have actually performed enQueue operation using push and pop.

154. A double-ended queue supports operations like adding and removing items from both the sides of the queue. They support four operations like addFront(adding item to top of the queue), addRear(adding item to the bottom of the queue), removeFront(removing item from the top of the queue) and removeRear(removing item from the bottom of the queue). You are given only stacks to implement this data structure. You can implement only push and pop operations. What's the time complexity of performing addFront and addRear? (Assume 'm' to be the size of the stack and 'n' to be the number of elements)
a) O(m) and O(n)
b) O(1) and O(n)
c) O(n) and O(1)
d) O(n) and O(m)

Answer: b
Explanation: addFront is just a normal push operation. Push operation is of O(1). Whereas addRear is of O(n) as it requires two push(pop()) operations of all elements of a stack.

155. Why is implementation of stack operations on queues not feasible for a large dataset (Asssume the number of

elements in the stack to be n)?
a) Because of its time complexity O(n)
b) Because of its time complexity O(log(n))
c) Extra memory is not required
d) There are no problem

Answer: a
Explanation: To perform Queue operations such as enQueue and deQueue there is a need of emptying all the elements of a current stack and pushing elements into the next stack and vice versa. Therfore it has a time complexity of O(n) and the need of extra stack as well, may not be feasible for a large dataset.

156. Consider yourself to be in a planet where the computational power of chips to be slow. You have an array of size 10.You want to perform enqueue some element into this array. But you can perform only push and pop operations .Push and pop operation both take 1 sec respectively. The total time required to perform enQueue operation is?
a) 20
b) 40
c) 42
d) 43

Answer: d
Explanation: First you have to empty all the elements of the current stack into the temporary stack, push the required element and empty the elements of the temporary stack into the original stack. Therfore taking 10+10+1+11+11= 43 seconds.

157. You have two jars, one jar which has 10 rings and the other has none. They are placed one above the other. You want to remove the last ring in the jar. And the second jar is weak and cannot be used to store rings for a long time.
a) Empty the first jar by removing it one by one from the first jar and placing it into the second jar
b) Empty the first jar by removing it one by one from the first jar and placing it into the second jar and empty the second jar by placing all the rings into the first jar one by one
c) There exists no possible way to do this
d) Break the jar and remove the last on

Answer: b
Explanation: This is similar to performing dequeue operation using push and pop only. Elements in the first jar are taken out and placed in the second jar. After removing the last element from the first jar, remove all the elements in the second jar and place them in the first jar.

158. Given only a single array of size 10 and no other memory is available. Which of the following operation is not feasible to implement (Given only push and pop operation)?
a) Push
b) Pop
c) Enqueue
d) Returntop

Answer: c
Explanation: To perform Enqueue using just push and pop operations, there is a need of another array of same size. But as there is

no extra available memeory, the given operation is not feasible.

159. Given an array of size n, let's assume an element is 'touched' if and only if some operation is performed on it(for example, for performing a pop operation the top element is 'touched'). Now you need to perform Dequeue operation. Each element in the array is touched atleast?
a) Once
b) Twice
c) Thrice
d) Four times

Answer: d
Explanation: First each element from the first stack is popped, then pushed into the second stack, dequeue operation is done on the top of the stack and later the each element of second stack is popped then pushed into the first stack. Therfore each element is touched four times.

160. To implement a stack using queue(with only enqueue and dequeue operations), how many queues will you need?
a) 1
b) 2
c) 3
d) 4

Answer: b
Explanation: Either the push or the pop has to be a costly operation, and the costlier operation requires two queues.

161. What is the functionality of the following piece of code?

```
public void fun(int x)
{
        q1.offer(x);
}
```

a) Perform push() with push as the costlier operation
b) Perform push() with pop as the costlier operation
c) Perform pop() with push as the costlier operation
d) Perform pop() with pop as the costlier operation

Answer: b
Explanation: offer() suggests that it is a push operation, but we see that it is performed with only one queue, hence the pop operation is costlier.

162. Reversing a word using stack can be used to find if the given word is a palindrome or not.
a) True
b) False

Answer: a
Explanation: This application of stack can also be used to find if the given word is a palindrome because, if the reversed is same as that of the original word, the given word is a palindrome.

163. Which is the most appropriate data structure for reversing a word?
a) queue
b) stack
c) tree
d) graph

Answer: b
Explanation: Stack is the most appropriate

data structure for reversing a word because stack follows LIFO principle.

164. Operations required for reversing a word or a string using stack are push() and pop().
a) True
b) False

Answer: a
Explanation: Push operation inserts a character into the stack and pop operation pops the top of the stack.

165. What is the time complexity of reversing a word using stack algorithm?
a) O (N log N)
b) O (N^2)
c) O (N)
d) O (M log N)

Answer: c
Explanation: The time complexity of reversing a stack is mathematically found to be O (N) where N is the input.

166. What is a bit array?
a) Data structure for representing arrays of records
b) Data structure that compactly stores bits
c) An array in which most of the elements have the same value
d) Array in which elements are not present in continuous locations

Answer: b
Explanation: It compactly stores bits and exploits bit-level parallelism.

167. Which of the following bitwise operations will you use to set a particular bit to 1?
a) OR
b) AND
c) XOR
d) NOR

Answer: a
Explanation: 1 OR 1 = 1, 0 OR 1 = 1, any bit OR'ed with 1 gives 1.

168. Which of the following bitwise operations will you use to set a particular bit to 0?
a) OR
b) AND
c) XOR
d) NAND

Answer: b
Explanation: 1 AND 0 = 0, 0 AND 0 = 0, any bit AND with 0 gives 0.

169. Which of the following bitwise operations will you use to toggle a particular bit?
a) OR
b) AND
c) XOR
d) NOT

Answer: c
Explanation: 1 XOR 1 = 0, 0 XOR 1 = 1, note that NOT inverts all the bits, while XOR toggles only a specified bit.

170. Which of the following is not an advantage of bit array?
a) Exploit bit level parallelism
b) Maximal use of data cache
c) Can be stored and manipulated in the

register set for long periods of time
d) Accessing Individual Elements is easy

Answer: d
Explanation: Individual Elements are difficult to access and can't be accessed in some programming languages. If random access is more common than sequential access, they have to be compressed to byte/word array. Exploit Bit parallelism, Maximal use of data cache and storage and manipulation for longer time in register set are all advantages of bit array.

171. What is a dynamic array?
a) A variable size data structure
b) An array which is created at runtime
c) The memory to the array is allocated at runtime
d) An array which is reallocated everytime whenever new elements have to be added

Answer: a
Explanation: It is a varying-size list data structure that allows items to be added or removed, it may use a fixed sized array at the back end.

172. What is meant by physical size in a dynamic array?
a) The size allocated to elements
b) The size extended to add new elements
c) The size of the underlying array at the back-end
d) The size visible to users

Answer: c
Explanation: Physical size, also called array capacity is the size of the underlying array, which is the maximum size without relocation of data.

173. The number of items used by the dynamic array contents is its _____
a) Physical size
b) Capacity
c) Logical size
d) Random size

Answer: c
Explanation: The number of items used by the dynamic array contents is called logical size. Physical size is the size of the underlying array, which is the maximum size without reallocation of data.

174. How will you implement dynamic arrays in Java?
a) Set
b) Map
c) HashMap
d) List

Answer: d
Explanation: ArrayList is used to implement dynamic arrays in Java.

175. Which of the following is the correct syntax to declare an ArrayList in Java?
a) ArrayList al = new ArrayList();
b) ArrayList al = new ArrayList[];
c) ArrayList al() = new ArrayList();
d) ArrayList al[] = new ArrayList[];

Answer: a
Explanation: This is a non-generic way of creating an ArrayList.

176. Array is divided into two parts in _____

a) Hashed Array Tree
b) Geometric Array
c) Bounded-size dynamic array
d) Sparse Array

Answer: c
Explanation: The first part stores the items of the dynamic array and the second part is reserved for new allocations.

177. Which of the following is a disadvantage of dynamic arrays?
a) Locality of reference
b) Data cache utilization
c) Random access
d) Memory leak

Answer: d
Explanation: Dynamic arrays share the advantage of arrays, added to it is the dynamic addition of elements to the array. Memory can be leaked if it is not handled properly during allocation and deallocation. It is a disadvantage.

178. What are parallel arrays?
a) Arrays of the same size
b) Arrays allocated one after the other
c) Arrays of the same number of elements
d) Arrays allocated dynamically

Answer: c
Explanation: Different arrays can be of different data types but should contain same number of elements. Elements at corresponding index belong to a record.

179. Which of the following is a disadvantage of parallel array over the traditional arrays?
a) When a language does not support records, parallel arrays can be used
b) Increased locality of reference
c) Ideal cache behaviour
d) Insertion and Deletion becomes tedious

Answer: d
Explanation: Insertion and deletion of elements require to move every element from their initial positions. This will become tedious. For Record collection, locality of reference and Ideal Cache behaviour we can use parallel arrays.

180. Which of the following is an advantage of parallel arrays?
a) Poor locality of reference for non-sequential access
b) Very little direct language support
c) Expensive to shrink or grow
d) Increased Locality of Reference

Answer: d
Explanation: Elements in the parallel array are accessed sequentially as one arrays holds the keys whereas other holds the values. This sequential access generally improves Locality of Reference. It is an advantage.

181. What is a sorted array?
a) Arrays sorted in numerical order
b) Arrays sorted in alphabetical order
c) Elements of the array are placed at equally spaced addresses in the memory
d) All of the mentioned

Answer: d
Explanation: The array can be sorted in any way, numerical, alphabetical or any other

way but the elements are placed at equally spaced addresses.

182. To search for an element in a sorted array, which searching technique can be used?
a) Linear Search
b) Jump Search
c) Binary Search
d) Fibonacci Search

Answer: c
Explanation: Since the array is sorted, binary search is preferred as its time complexity is O(logn).

183. Which of the following is not an application of sorted array?
a) Commercial computing
b) Priority Scheduling
c) Discrete Mathematics
d) Hash Tables

Answer: d
Explanation: Sorted arrays have widespread applications as all commercial computing involves large data which is very useful if it is sorted. It makes best use of locality of reference and data cache. Linked lists are used in Hash Tables not arrays.

184. What is the worst case time complexity of inserting an element into the sorted array?
a) O(nlogn)
b) O(logn)
c) O(n)
d) $O(n^2)$

Answer: c
Explanation: In the worst case, an element must added to the front of the array, which means that rest of the elements have to be shifted, hence the worst case time complexity becomes O(n).

185. What is the order of a matrix?
a) number of rows X number of columns
b) number of columns X number of rows
c) number of rows X number of rows
d) number of columns X number of columns

Answer: a
Explanation: The order of the matrix is the number of rows X number of columns.

186. Which of the following property does not hold for matrix multiplication?
a) Associative
b) Distributive
c) Commutative
d) Additive Inverse

Answer: c
Explanation: In matrix multiplication, AB != BA

187. How do you allocate a matrix using a single pointer in C?(r and c are the number of rows and columns respectively)
a) int *arr = malloc(r * c * sizeof(int));
b) int *arr = (int *)malloc(r * c * sizeof(int));
c) int *arr = (int *)malloc(r + c * sizeof(int));
d) int *arr = (int *)malloc(r * c * sizeof(arr));

Answer: b
Explanation: Total number of elements in the matrix will be r*c

188. If row-major order is used, how is the following matrix stored in memory?
a b c
d e f
g h i
a) ihgfedcba
b) abcdefghi
c) cfibehadg
d) adgbehcfi

Answer: b
Explanation: It starts with the first element and continues in the same row until the end of row is reached and then proceeds with the next row. C follows row-major order.

189. If column-major order is used, how is the following matrix stored in memory?
a b c
d e f
g h i
a) ihgfedcba
b) abcdefghi
c) cfibehadg
d) adgbehcfi

Answer: d
Explanation: It starts with the first element and continues in the same column until the end of column is reached and then proceeds with the next column. Fortran follows column-major order.

190. Which of the following don't use matrices?
a) In solving linear equations
b) Image processing
c) Graph theory
d) Sorting numbers

Answer: d
Explanation: Numbers uses arrays(1-D) for sorting not matrices(2-D arrays). Solving linear equations is a separate field in Mathematics involving matrices, Image processing stores the pixels in the form of matrices, and the graphs are represented with the help of matrices to indicate the nodes and edges.

191. Which of the following is an advantage of matrices?
a) Internal complexity
b) Searching through a matrix is complex
c) Not space efficient
d) Graph Plotting

Answer: d
Explanation: Adjacency and Incidence Matrices are used to store vertices and edges of a graph. It is an advantage to plot graphs easily using matrices. But Time complexity of a matrix is $O(n^2)$ and sometimes the internal organization becomes tedious. They are all disadvantages of matrices.

192. Matrix A when multiplied with Matrix C gives the Identity matrix I, what is C?
a) Identity matrix
b) Inverse of A
c) Square of A
d) Transpose of A

Answer: b
Explanation: Any square matrix when multiplied with its inverse gives the identity matrix. Note that non square matrices are not invertible.

193. What does the following piece of code do?

```
for(int i = 0; i < row; i++)
{
   for(int j = 0; j < column; j++)
   {
      if(i == j)
         sum = sum + (array[i][j]);
   }
}
System.out.println(sum);
```

a) Normal of a matrix
b) Trace of a matrix
c) Square of a matrix
d) Transpose of a matrix

Answer: b
Explanation: Trace of a matrix is the sum of the principal diagonal elements.

194. Which matrix has most of the elements (not all) as Zero?
a) Identity Matrix
b) Unit Matrix
c) Sparse Matrix
d) Zero Matrix

Answer: c
Explanation: Sparse Matrix is a matrix in which most of the elements are Zero. Identity Matrix is a matrix in which all principle diagonal elements are 1 and rest of the elements are Zero. Unit Matrix is also called Identity Matrix. Zero Matrix is a matrix in which all the elements are Zero.

195. What is the relation between Sparsity and Density of a matrix?
a) Sparsity = 1 – Density
b) Sparsity = 1 + Density
c) Sparsity = Density*Total number of elements
d) Sparsity = Density/Total number of elements

Answer: a
Explanation: Sparsity of a matrix is equal to 1 minus Density of the matrix. The Sparsity of matrix is defined as the total number of Zero Valued elements divided total number of elements.

196. Who coined the term Sparse Matrix?
a) Harry Markowitz
b) James Sylvester
c) Chris Messina
d) Arthur Cayley

Answer: a
Explanation: Harry Markowitz coined the term Sparse Matrix. James Sylvester coined the term Matrix. Chris Messina coined the term Hashtag and Arthur Cayley developed the algebraic aspects of a matrix.

197. Is O(n) the Worst case Time Complexity for addition of two Sparse Matrix?
a) True
b) False

Answer: a
Explanation: In Addition, the matrix is traversed linearly, hence it has the time complexity of O(n) where n is the number of non-zero elements in the largest matrix amongst two.

198. The matrix contains m rows and n columns. The matrix is called Sparse Matrix

if _____
a) Total number of Zero elements > (m*n)/2
b) Total number of Zero elements = m + n
c) Total number of Zero elements = m/n
d) Total number of Zero elements = m-n

Answer: a
Explanation: For matrix to be Sparse Matrix, it should contain Zero elements more than the non-zero elements. Total elements of the given matrix is m*n. So if Total number of Zero elements > (m*n)/2, then the matrix is called Sparse Matrix.

199. Which of the following is not the method to represent Sparse Matrix?
a) Dictionary of Keys
b) Linked List
c) Array
d) Heap

Answer: d
Explanation: Heap is not used to represent Sparse Matrix while in Dictionary, rows and column numbers are used as Keys and values as Matrix entries, Linked List is used with each node of Four fields (Row, Column, Value, Next Node) (2D array is used to represent the Sparse Matrix with three fields (Row, Column, Value).

200. Is Sparse Matrix also known as Dense Matrix?
a) True
b) False

Answer: b
Explanation: Sparse Matrix is a matrix with most of the elements as Zero elements while Dense Matrix is a matrix with most of the elements as Non-Zero element.

201. Which one of the following is a Special Sparse Matrix?
a) Band Matrix
b) Skew Matrix
c) Null matrix
d) Unit matrix

Answer: a
Explanation: A band matrix is a sparse matrix whose non zero elements are bounded to a diagonal band, comprising the main diagonal and zero or more diagonals on either side.

202. In what way the Symmetry Sparse Matrix can be stored efficiently?
a) Heap
b) Binary tree
c) Hash table
d) Adjacency List

Answer: b
Explanation: Since Symmetry Sparse Matrix arises as the adjacency matrix of the undirected graph. Hence it can be stored efficiently as an adjacency list.

203. What does the number of inversions in an array indicate?
a) mean value of the elements of array
b) measure of how close or far the array is from being sorted
c) the distribution of values in the array
d) median value of the elements of array

Answer: b
Explanation: The number of inversions in an

array indicates how close or far the array is from being completely sorted. The array is sorted if the number of inversions are 0.

204. How many inversions does a sorted array have?
a) 0
b) 1
c) 2
d) cannot be determined

Answer: a
Explanation: When an array is sorted then there cannot be any inversion in the array. As the necessary condition for an inversion is arr[i]>arr[j] and i<j.

205. What is the condition for two elements arr[i] and arr[j] to form an inversion?
a) arr[i]<arr[j]
b) i < j
c) arr[i] < arr[j] and i < j
d) arr[i] > arr[j] and i < j

Answer: d
Explanation: For two elements to form an inversion the necessary condition is arr[i] > arr[j] and i < j. The number of inversions in an array indicate how close or far the array is from being completely sorted.

206. Under what condition the number of inversions in an array are maximum?
a) when the array is sorted
b) when the array is reverse sorted
c) when the array is half sorted
d) depends on the given array

Answer: b
Explanation: Number of inversions in an array are maximum when the given array is reverse sorted. As the necessary condition for an inversion is arr[i]>arr[j] and i<j.

207. What will be the resulting array after rotating arr[]={1, 2, 3, 4, 5} by 2?
a) 2, 1, 3, 4, 5
b) 3, 4, 5, 1, 2
c) 4, 5, 1, 2, 3
d) 1, 2, 3, 5, 4

Answer: b
Explanation: When the given array is rotated by 2 then the resulting array will be
Rotation 1: {2,3,4,5,1}
Rotation 2: {3,4,5,1,2}.
Thus, the final array is {3,4,5,1,2}.

208. What will be the minimum number of jumps required to reach the end of the array arr[] = {1,3,6,3,6,8,5}?
a) 1
b) 2
c) 3
d) not possible to reach the end

Answer: c
Explanation: Each element of the array represents the maximum number of steps that can be taken forward from that element. If the first element is 0 then it is not possible to reach the end.

209. What will be the minimum number of jumps required to reach the end of the array arr[] ={0,1,3,6,3,6,8,5}?
a) 1
b) 2
c) 3
d) not possible to reach the end

Answer: d
Explanation: Each element of the array represents the maximum number of steps that can be taken forward from that element. So as the first element here is 0 so we cannot move any further from the first element. Thus, it is not possible to reach the end of the array.

210. What is a skip list?
a) a linkedlist with size value in nodes
b) a linkedlist that allows faster search within an ordered sequence
c) a linkedlist that allows slower search within an ordered sequence
d) a tree which is in the form of linked list

Answer: b
Explanation: It is a datastructure, which can make search in sorted linked list faster in the same way as binary search tree and sorted array (using binary search) are faster.

211.Which of the following is true about the Move-To-Front Method for rearranging nodes?
a) node with highest access count is moved to head of the list
b) requires extra storage
c) may over-reward infrequently accessed nodes
d) requires a counter for each node

Answer: c
Explanation: In Move-To-front Method the element which is searched is moved to the head of the list. And if a node is searched even once, it is moved to the head of the list and given maximum priority even if it is not going to be accessed frequently in the future. Such a situation is referred to as over-rewarding.

212. What is xor linked list ?
a) uses of bitwise XOR operation to decrease storage requirements for doubly linked lists
b) uses of bitwise XOR operation to decrease storage requirements for linked lists
c) uses of bitwise operations to decrease storage requirements for doubly linked lists
d) just another form of linked list

Answer: a
Explanation: Why we use bitwise XOR operation is to decrease storage requirements for doubly linked lists.

213. What does a xor linked list have ?
a) every node stores the XOR of addresses of previous and next nodes
b) actuall memory address of next node
c) every node stores the XOR of addresses of previous and next two nodes
d) every node stores xor 0 and the current node address

Answer: a
Explanation: Every node stores the XOR of addresses.

214. Consider a situation of writing a binary tree into a file with memory storage efficiency in mind, is array representation of tree is good?
a) yes because we are overcoming the need of pointers and so space efficiency
b) yes because array values are indexable
c) No it is not efficient in case of sparse trees and remaning cases it is fine

d) No linked list representation of tree is only fine

Answer: c
Explanation: In case of sparse trees (where one node per level in worst cases), the array size (2^h)-1 where h is height but only h indexes will be filled and (2^h)-1-h nodes will be left unused leading to space wastage.

215. Why is heap implemented using array representations than tree(linked list) representations though both tree representations and heaps have same complexities?

for binary heap
-insert: O(log n)
-delete min: O(log n)

for a tree
-insert: O(log n)
-delete: O(log n)

Then why go with array representation when both are having same values ?
a) arrays can store trees which are complete and heaps are not complete
b) lists representation takes more memory hence memory efficiency is less and go with arrays and arrays have better caching
c) lists have better caching
d) In lists insertion and deletion is difficult

Answer: b
Explanation: In memory the pointer address for next node may not be adjacent or nearer to each other and also array have wonderful caching power from os and manipulating pointers is a overhead. Heap data structure is always a complete binary tree.

216. Can a tree stored in an array using either one of inorder or post order or pre order traversals be again reformed?
a) Yes just traverse through the array and form the tree
b) No we need one more traversal to form a tree
c) No in case of sparse trees
d) Yes by using both inorder and array elements

Answer: b
Explanation: We need any two traversals for tree formation but if some additional stuff or techniques are used while storing a tree in an array then one traversal can facilitate like also storing null values of a node in array.

217. How many children does a binary tree have?
a) 2
b) any number of children
c) 0 or 1 or 2
d) 0 or 1

Answer: c
Explanation: Can have atmost 2 nodes.

218. What is/are the disadvantages of implementing tree using normal arrays?
a) difficulty in knowing children nodes of a node
b) difficult in finding the parent of a node
c) have to know the maximum number of nodes possible before creation of trees
d) difficult to implement

Answer: c
Explanation: The size of array is fixed in normal arrays. We need to know the number of nodes in the tree before array

declaration. It is the main disadvantage of using arrays to represent binary trees.

219. What must be the ideal size of array if the height of tree is 'l'?
a) 2^l-1
b) l-1
c) l
d) 2l

Answer: a
Explanation: Maximum elements in a tree (complete binary tree in worst case) of height 'L' is 2^L-1. Hence size of array is taken as 2^L-1.

220. What are the children for node 'w' of a complete-binary tree in an array representation?
a) 2w and 2w+1
b) 2+w and 2-w
c) w+1/2 and w/2
d) w-1/2 and w+1/2

Answer: a
Explanation: The left child is generally taken as 2*w whereas the right child will be taken as 2*w+1 because root node is present at index 0 in the array and to access every index position in the array.

221. What is the parent for a node 'w' of a complete binary tree in an array representation when w is not 0?
a) floor(w-1/2)
b) ceil(w-1/2)
c) w-1/2
d) w/2

Answer: a
Explanation: Floor of w-1/2 because we can't miss a node.

222. If the tree is not a complete binary tree then what changes can be made for easy access of children of a node in the array?
a) every node stores data saying which of its children exist in the array
b) no need of any changes continue with 2w and 2w+1, if node is at i
c) keep a seperate table telling children of a node
d) use another array parallel to the array with tree

Answer: a
Explanation: Array cannot represent arbitrary shaped trees. It can only be used in case of complete trees. If every node stores data saying that which of its children exists in the array then elements can be accessed easily.

223. Advantages of linked list representation of binary trees over arrays?
a) dynamic size
b) ease of insertion/deletion
c) ease in randomly accessing a node
d) both dynamic size and ease in insertion/deletion

Answer: d
Explanation: It has both dynamic size and ease in insertion and deletion as advantages.

224. Disadvantages of linked list representation of binary trees over arrays?
a) Randomly accessing is not possible
b) Extra memory for a pointer is needed with every element in the list

c) Difficulty in deletion
d) Random access is not possible and extra memory with every element

Answer: d
Explanation: Random access is not possible with linked lists.

225. Which of the following traversing algorithm is not used to traverse in a tree?
a) Post order
b) Pre order
c) Post order
d) Randomized

Answer: d
Explanation: Generally, all nodes in a tree are visited by using preorder, inorder and postorder traversing algorithms.

226. Level order traversal of a tree is formed with the help of
a) breadth first search
b) depth first search
c) dijkstra's algorithm
d) prims algorithm

Answer: a
Explanation: Level order is similar to bfs.

227. Identify the reason which doesn't play a key role to use threaded binary trees?
a) The storage required by stack and queue is more
b) The pointers in most of nodes of a binary tree are NULL
c) It is Difficult to find a successor node
d) They occupy less size

Answer: d
Explanation: Threaded binary trees are introduced to make the Inorder traversal faster without using any stack or recursion. Stack and Queue require more space and pointers in the majority of binary trees are null and difficulties are raised while finding successor nodes. Size constraints are not taken on threaded binary trees, but they occupy less space than a stack/queue.

228. The following lines talks about deleting a node in a binary tree.(the tree property must not be violated after deletion)
i) from root search for the node to be deleted
ii)
iii) delete the node at
what must be statement ii) and fill up statement iii)
a) ii)-find random node,replace with node to be deleted. iii)- delete the node
b) ii)-find node to be deleted. iii)- delete the node at found location
c) ii)-find deepest node,replace with node to be deleted. iii)- delete a node
d) ii)-find deepest node,replace with node to be deleted. iii)- delete the deepest node

Answer: d
Explanation: We just replace a to be deleted node with last leaf node of a tree. this must not be done in case of BST or heaps.

229. What may be the psuedo code for finding the size of a tree?
a) find_size(root_node–>left_node) + 1 + find_size(root_node–>right_node)
b) find_size(root_node–>left_node) + find_size(root_node–>right_node)
c) find_size(root_node–>right_node) – 1
d) find_size(root_node–>left_node + 1

Answer: a
Explanation: Draw a tree and analyze the expression. we are always taking size of left subtree and right subtree and adding root value(1) to it and finally printing size.

230. What is missing in this logic of finding a path in the tree for a given sum (i.e

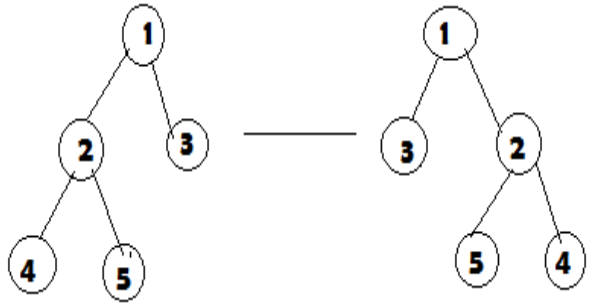
checking whether there will be a path from roots to leaf nodes with given sum)?

```
checkSum(struct bin-treenode *root , int sum) :
 if(root==null)
  return sum as 0
 else :
   leftover_sum=sum-root_node-->value
   //missing
```

a) code for having recursive calls to either only left tree or right trees or to both subtrees depending on their existence
b) code for having recursive calls to either only left tree or right trees
c) code for having recursive calls to either only left tree
d) code for having recursive calls to either only right trees

Answer: a
Explanation: if(left subtree and right subtree) then move to both subtrees
else if only left subtree then move to left subtree carrying leftover_sum parameter
else if only right subtree then move to right subtree carrying leftover_sum parameter.

231. What must be the missing logic below so as to print mirror of a tree as below as an example?




```
if(rootnode):
 mirror(rootnode-->left)
 mirror(rootnode-->right)
 //missing

end
```

a) swapping of left and right nodes is missing
b) swapping of left with root nodes is missing
c) swapping of right with root nodes is missing
d) nothing is missing

Answer: a
Explanation: Mirror is another tree with left and right children of nodes are interchanged as shown in the figure.

232. What is the code below trying to print?

```
void print(tree *root,tree *node)
{
 if(root ==null) return 0
 if(root-->left==node || root-->right==node
|| print(root->left,node)||printf(root-
>right,node)
 {
   print(root->data)
 }
}
```

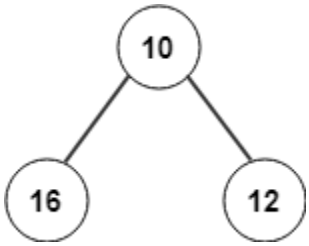
a) just printing all nodes
b) not a valid logic to do any task
c) printing ancestors of a node passed as argument
d) printing nodes from leaf node to a node passed as argument

Answer: c
Explanation: We are checking if left or right node is what the argument sent or else if not the case then move to left node or right node and print all nodes while searching for the argument node.

233. What is the maximum number of children that a binary tree node can have?
a) 0
b) 1
c) 2
d) 3

Answer: c
Explanation: In a binary tree, a node can have atmost 2 nodes (i.e.) 0,1 or 2 left and right child.

234. The following given tree is an example for?



a) Binary tree
b) Binary search tree
c) Fibonacci tree
d) AVL tree

Answer: a
Explanation: The given tree is an example for binary tree since has got two children and the left and right children do not satisfy binary search tree's property, Fibonacci and AVL tree.

235. A binary tree is a rooted tree but not an ordered tree.
a) true
b) false

Answer: b
Explanation: A binary tree is a rooted tree and also an ordered tree (i.e) every node in a binary tree has at most two children.

236. How many common operations are performed in a binary tree?
a) 1
b) 2
c) 3
d) 4

Answer: c
Explanation: Three common operations are performed in a binary tree- they are insertion, deletion and traversal.

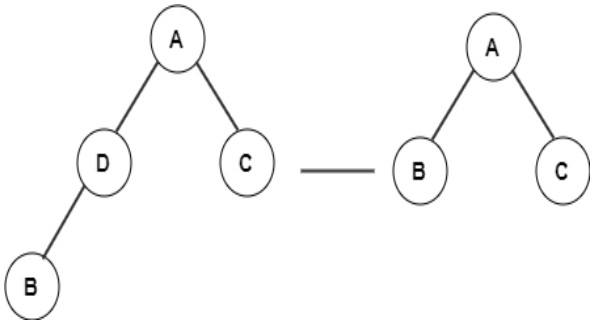
237. What is the traversal strategy used in the binary tree?
a) depth-first traversal
b) breadth-first traversal
c) random traversal
d) Priority traversal

Answer: b
Explanation: Breadth first traversal, also known as level order traversal is the traversal strategy used in a binary tree. It involves visiting all the nodes at a given level.

238. How many types of insertion are performed in a binary tree?
a) 1
b) 2
c) 3
d) 4

Answer: b
Explanation: Two kinds of insertion operation is performed in a binary tree- inserting a leaf node and inserting an internal node.

239. What operation does the following diagram depict?



a) inserting a leaf node
b) inserting an internal node
c) deleting a node with 0 or 1 child
d) deleting a node with 2 children

Answer: c
Explanation: The above diagram is a depiction of deleting a node with 0 or 1 child since the node D which has 1 child is deleted.

240. General ordered tree can be encoded into binary trees.
a) true
b) false

Answer: a
Explanation: General ordered tree can be mapped into binary tree by representing them in a left-child-right-sibling way.

241. How many bits would a succinct binary tree occupy?
a) n+O(n)
b) 2n+O(n)
c) n/2
d) n

Answer: b
Explanation: A succinct binary tree occupies close to minimum possible space established by lower bounds. A succinct binary tree would occupy 2n+O(n) bits.

242. The average depth of a binary tree is given as?
a) O(N)
b) O(√N)
c) $O(N^2)$
d) O(log N)

Answer: d
Explanation: The average depth of a binary tree is given as O(√N). In case of a binary search tree, it is O(log N).

243. How many orders of traversal are applicable to a binary tree (In General)?
a) 1
b) 4
c) 2
d) 3

Answer: d
Explanation: The three orders of traversal that can be applied to a binary tree are in-order, pre-order and post order traversal.

244. If binary trees are represented in arrays, what formula can be used to locate a left child, if the node has an index i?
a) 2i+1
b) 2i+2
c) 2i
d) 4i

Answer: a
Explanation: If binary trees are represented in arrays, left children are located at indices 2i+1 and right children at 2i+2.

245. Using what formula can a parent node be located in an array?
a) (i+1)/2
b) (i-1)/2
c) i/2
d) 2i/2

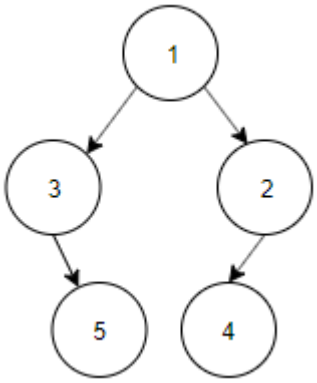
Answer: b
Explanation: If a binary tree is represented in an array, parent nodes are found at indices (i-1)/2.

246. Which of the following properties are obeyed by all three tree – traversals?
a) Left subtrees are visited before right subtrees
b) Right subtrees are visited before left subtrees
c) Root node is visited before left subtree
d) Root node is visited before right subtree

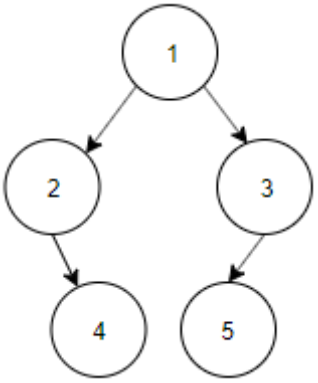
Answer: a
Explanation: In preorder, inorder and postorder traversal the left subtrees are visited before the right subtrees. In Inorder traversal, the Left subtree is visited first then the Root node then the Right subtree. In postorder traversal, the Left subtree is visited first, then Right subtree and then the Root node is visited.

247. Construct a binary tree using the following data.
The preorder traversal of a binary tree is 1, 2, 5, 3, 4. The inorder traversal of the same binary tree is 2, 5, 1, 4, 3.

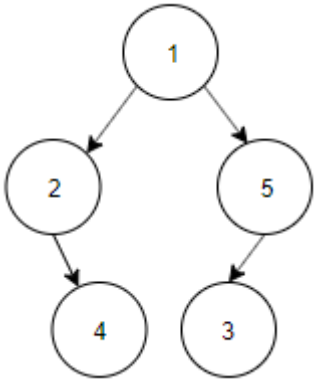
a)



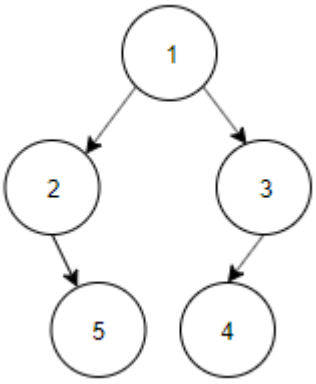
b)



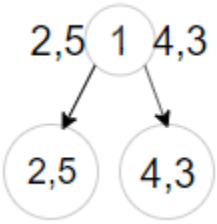
c)



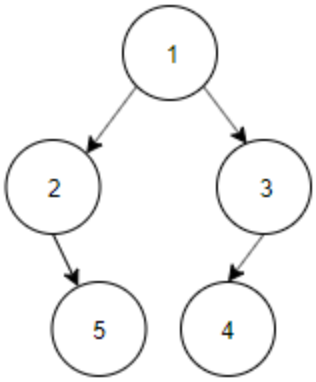
d)



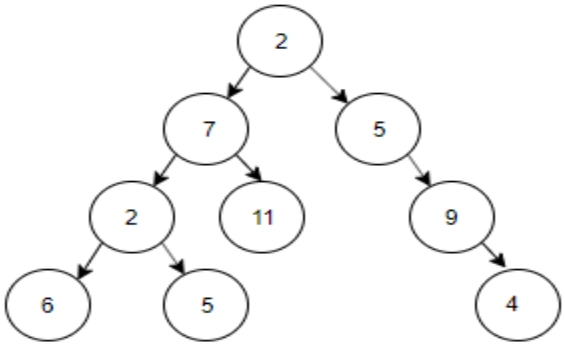
Answer: d
Explanation: Here,
Preorder Traversal is 1, 2, 5, 3, 4
Inorder Traversal is 2, 5, 1, 4, 3
Root node of binary tree is the first node in Preorder traversal.
The rough sketch of tree is:



Second node in preorder traversal is 2. This makes 5 as right child to node 2. The fourth node in preorder traversal is 3. This makes 4 as right child to node 3. Thus the final tree is:



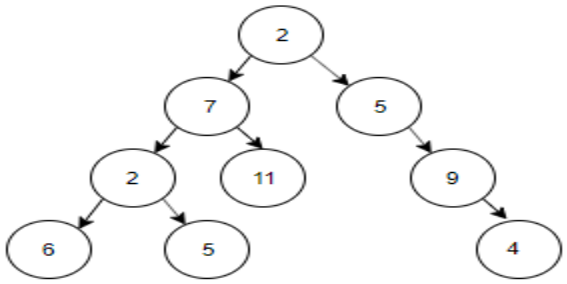
1. For the tree below, write the pre-order traversal.



a) 2, 7, 2, 6, 5, 11, 5, 9, 4
b) 2, 7, 5, 2, 6, 9, 5, 11, 4
c) 2, 5, 11, 6, 7, 4, 9, 5, 2
d) 2, 7, 5, 6, 11, 2, 5, 4, 9

Answer: a
Explanation: Pre order traversal follows NLR(Node-Left-Right).

248. For the tree below, write the post-order traversal.



a) 2, 7, 2, 6, 5, 11, 5, 9, 4
b) 2, 7, 5, 2, 6, 9, 5, 11, 4
c) 2, 5, 11, 6, 7, 4, 9, 5, 2
d) 2, 7, 5, 6, 11, 2, 5, 4, 9

Answer: c
Explanation: Post order traversal follows LRN(Left-Right-Node).

249. What is the time complexity of pre-order traversal in the iterative fashion?
a) O(1)
b) O(n)
c) O(logn)
d) O(nlogn)

Answer: b
Explanation: Since you have to go through all the nodes, the complexity becomes O(n).

250. What is the space complexity of the post-order traversal in the recursive

fashion? (d is the tree depth and n is the number of nodes)
a) O(1)
b) O(nlogd)
c) O(logd)
d) O(d)

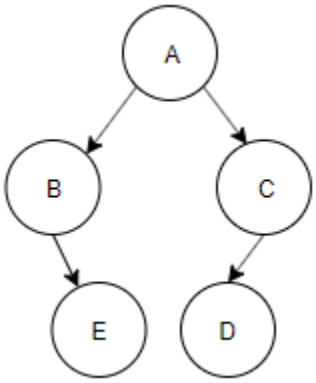
Answer: d
Explanation: In the worst case we have d stack frames in the recursive call, hence the complexity is O(d).

251. To obtain a prefix expression, which of the tree traversals is used?
a) Level-order traversal
b) Pre-order traversal
c) Post-order traversal
d) In-order traversal

Answer: b
Explanation: As the name itself suggests, pre-order traversal can be used.

252. Consider the following data. The pre order traversal of a binary tree is A, B, E, C, D. The in order traversal of the same binary tree is B, E, A, D, C. The level order sequence for the binary tree is _____
a) A, C, D, B, E
b) A, B, C, D, E
c) A, B, C, E, D
d) D, B, E, A, C

Answer: b
Explanation: The inorder sequence is B, E, A, D, C and Preorder sequence is A, B, E, C, D. The tree constructed with the inorder and preorder sequence is



The levelorder traversal (BFS traversal) is A, B, C, E, D.

253. Consider the following data and specify which one is Preorder Traversal Sequence, Inorder and Postorder sequences.
S1: N, M, P, O, Q
S2: N, P, Q, O, M
S3: M, N, O, P, Q
a) S1 is preorder, S2 is inorder and S3 is postorder
b) S1 is inorder, S2 is preorder and S3 is postorder
c) S1 is inorder, S2 is postorder and S3 is preorder
d) S1 is postorder, S2 is inorder and S3 is preorder

Answer: c
Explanation: Preorder traversal starts from the root node and postorder and inorder starts from the left child node of the left subtree. The first node of S3 is different and for S1 and S2 it's the same. Thus, S3 is preorder traversal and the root node is M. Postorder traversal visits the root node at last. S2 has the root node(M) at last that implies S2 is postorder traversal. S1 is inorder traversal as S2 is postorder traversal and S3 is preorder traversal. Therefore, S1

is inorder traversal, S2 is postorder traversal and S3 is preorder traversal.

254. What is the time complexity of pre-order traversal in the iterative fashion?
a) O(1)
b) O(n)
c) O(logn)
d) O(nlogn)

Answer: b
Explanation: Since you have to go through all the nodes, the complexity becomes O(n).

255. What is the space complexity of the post-order traversal in the recursive fashion? (d is the tree depth and n is the number of nodes)
a) O(1)
b) O(nlogd)
c) O(logd)
d) O(d)

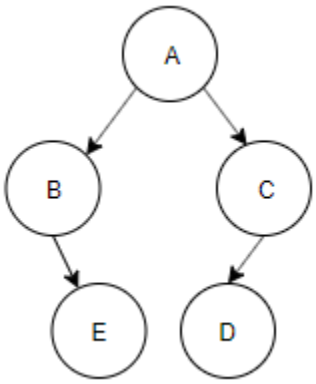
Answer: d
Explanation: In the worst case we have d stack frames in the recursive call, hence the complexity is O(d).

256. To obtain a prefix expression, which of the tree traversals is used?
a) Level-order traversal
b) Pre-order traversal
c) Post-order traversal
d) In-order traversal

Answer: b
Explanation: As the name itself suggests, pre-order traversal can be used.

257. Consider the following data. The pre order traversal of a binary tree is A, B, E, C, D. The in order traversal of the same binary tree is B, E, A, D, C. The level order sequence for the binary tree is _____
a) A, C, D, B, E
b) A, B, C, D, E
c) A, B, C, E, D
d) D, B, E, A, C

Answer: b
Explanation: The inorder sequence is B, E, A, D, C and Preorder sequence is A, B, E, C, D. The tree constructed with the inorder and preorder sequence is

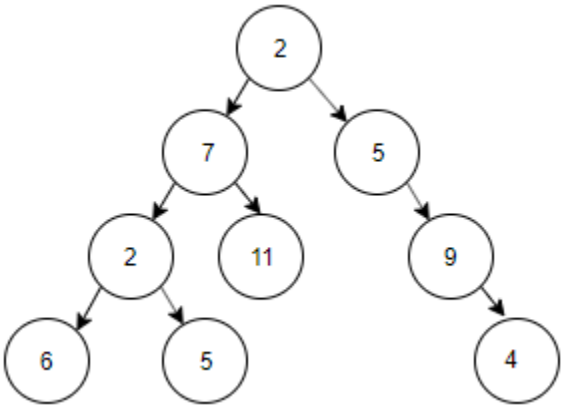


The levelorder traversal (BFS traversal) is A, B, C, E, D.

258. Consider the following data and specify which one is Preorder Traversal Sequence, Inorder and Postorder sequences.
S1: N, M, P, O, Q
S2: N, P, Q, O, M
S3: M, N, O, P, Q
a) S1 is preorder, S2 is inorder and S3 is postorder
b) S1 is inorder, S2 is preorder and S3 is postorder
c) S1 is inorder, S2 is postorder and S3 is preorder
d) S1 is postorder, S2 is inorder and S3 is preorder

Answer: c
Explanation: Preorder traversal starts from the root node and postorder and inorder starts from the left child node of the left subtree. The first node of S3 is different and for S1 and S2 it's the same. Thus, S3 is preorder traversal and the root node is M. Postorder traversal visits the root node at last. S2 has the root node(M) at last that implies S2 is postorder traversal. S1 is inorder traversal as S2 is postorder traversal and S3 is preorder traversal. Therefore, S1 is inorder traversal, S2 is postorder traversal and S3 is preorder traversal.

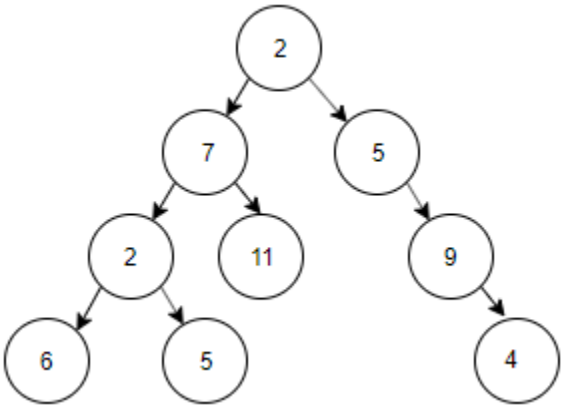
259. For the tree below, write the in-order traversal.



a) 6, 2, 5, 7, 11, 2, 5, 9, 4
b) 6, 5, 2, 11, 7, 4, 9, 5, 2
c) 2, 7, 2, 6, 5, 11, 5, 9, 4
d) 2, 7, 6, 5, 11, 2, 9, 5, 4

Answer: a
Explanation: In-order traversal follows LNR(Left-Node-Right).

260. For the tree below, write the level-order traversal.



a) 2, 7, 2, 6, 5, 11, 5, 9, 4
b) 2, 7, 5, 2, 11, 9, 6, 5, 4
c) 2, 5, 11, 6, 7, 4, 9, 5, 2
d) 2, 7, 5, 6, 11, 2, 5, 4, 9

Answer: b
Explanation: Level order traversal follows a breadth first search approach.

261. The number of edges from the root to the node is called _____ of the tree.
a) Height
b) Depth
c) Length
d) Width

Answer: b
Explanation: The number of edges from the root to the node is called depth of the tree.

262. The number of edges from the node to the deepest leaf is called _____ of the tree.
a) Height
b) Depth
c) Length
d) Width

Answer: a
Explanation: The number of edges from the

node to the deepest leaf is called height of the tree.

263. What is a full binary tree?
a) Each node has exactly zero or two children
b) Each node has exactly two children
c) All the leaves are at the same level
d) Each node has exactly one or two children

Answer: a
Explanation: A full binary tree is a tree in which each node has exactly 0 or 2 children.

264. What is a complete binary tree?
a) Each node has exactly zero or two children
b) A binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from right to left
c) A binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right
d) A tree In which all nodes have degree 2

Answer: c
Explanation: A binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right is called complete binary tree. A Tree in which each node has exactly zero or two children is called full binary tree. A Tree in which the degree of each node is 2 except leaf nodes is called perfect binary tree.

265. What is the average case time complexity for finding the height of the binary tree?
a) h = O(loglogn)
b) h = O(nlogn)
c) h = O(n)
d) h = O(log n)

Answer: d
Explanation: The nodes are either a part of left sub tree or the right sub tree, so we don't have to traverse all the nodes, this means the complexity is lesser than n, in the average case, assuming the nodes are spread evenly, the time complexity becomes O(logn).

266. Which of the following is not an advantage of trees?
a) Hierarchical structure
b) Faster search
c) Router algorithms
d) Undo/Redo operations in a notepad

Answer: d
Explanation: Undo/Redo operations in a notepad is an application of stack. Hierarchical structure, Faster search, Router algorithms are advantages of trees.

267. In a full binary tree if number of internal nodes is I, then number of leaves L are?
a) L = 2*I
b) L = I + 1
c) L = I – 1
d) L = 2*I – 1

Answer: b
Explanation: Number of Leaf nodes in full binary tree is equal to 1 + Number of Internal Nodes i.e L = I + 1

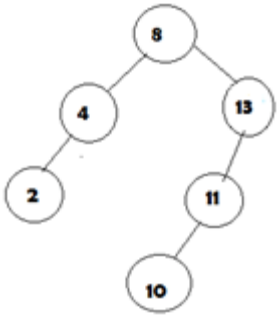
268. What is an AVL tree?
a) a tree which is balanced and is a height balanced tree
b) a tree which is unbalanced and is a height balanced tree
c) a tree with three children
d) a tree with atmost 3 children

Answer: a
Explanation: It is a self balancing tree with height difference atmost 1.

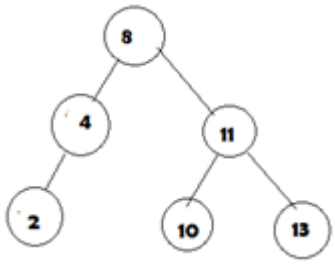
269. Why we need to a binary tree which is height balanced?
a) to avoid formation of skew trees
b) to save memory
c) to attain faster memory access
d) to simplify storing

Answer: a
Explanation: In real world dealing with random values is often not possible, the probability that u are dealing with non random values(like sequential) leads to mostly skew trees, which leads to worst case. hence we make height balance by rotations.

270. Which of the below diagram is following AVL tree property?



i.



ii.
a) only i
b) only i and ii
c) only ii
d) i is not a binary search tree

Answer: b
Explanation: The property of AVL tree is it is height balanced tree with difference of atmost 1 between left and right subtrees. All AVL trees are binary search tree.

271. What is the maximum height of an AVL tree with p nodes?
a) p
b) log(p)
c) log(p)/2
d) $^{p}/_{2}$

Answer: b
Explanation: Consider height of tree to be 'he', then number of nodes which totals to p can be written in terms of height as N(he)=N(he-1)+1+N(he-2). since N(he) which is p can be written in terms of height as the beside recurrence relation which on solving gives N(he)= O(logp) as worst case height.

272. To restore the AVL property after inserting a element, we start at the insertion point and move towards root of that tree. is this statement true?
a) true
b) false

Answer: a
Explanation: It is interesting to note that

after insertion, only the path from that point to node or only that subtrees are imbalanced interms of height.

273. Given an empty AVL tree, how would you construct AVL tree when a set of numbers are given without performing any rotations?
a) just build the tree with the given input
b) find the median of the set of elements given, make it as root and construct the tree
c) use trial and error
d) use dynamic programming to build the tree

Answer: b
Explanation: Sort the given input, find the median element among them, make it as root and construct left and right subtrees with elements lesser and greater than the median element recursively. this ensures the subtrees differ only by height 1.

274. What maximum difference in heights between the leafs of a AVL tree is possible?
a) log(n) where n is the number of nodes
b) n where n is the number of nodes
c) 0 or 1
d) atmost 1

Answer: a
Explanation: At every level we can form a tree with difference in height between subtrees to be atmost 1 and so there can be log(n) such levels since height of AVL tree is log(n).

275. Consider the pseudo code:

```
int avl(binarysearchtree root):
  if(not root)
   return 0
  left_tree_height = avl(left_of_root)

  if(left_tree_height== -1)
   return left_tree_height

  right_tree_height= avl(right_of_root)

  if(right_tree_height==-1)
   return right_tree_height
```

Does the above code can check if a binary search tree is an AVL tree?
a) yes
b) no

Answer: a
Explanation: The condition to check the height difference between left and right subtrees is missing. if (absolute(left_tree_height – right_tree_height)>1) must be added.

276. Consider the below left-left rotation pseudo code where the node contains value pointers to left, right child nodes and a height value and Height() function returns height value stored at a particular node.

```
avltree leftrotation(avltreenode z):
  avltreenode w =x-left
  x-left=w-right
  w-right=x
  x-height=max(Height(x-left),Height(x-right))+1
  w-height=max(missing)+1
 return w
```

What is missing?
a) Height(w-left), x-height
b) Height(w-right), x-height
c) Height(w-left), x
d) Height(w-left)

Answer: a
Explanation: In the code we are trying to

make the left rotation and so we need to find maximum of those two values.

277. Why to prefer red-black trees over AVL trees?
a) Because red-black is more rigidly balanced
b) AVL tree store balance factor in every node which costs space
c) AVL tree fails at scale
d) Red black is more efficient

Answer: b
Explanation: Every node in an AVL tree need to store the balance factor (-1, 0, 1) hence space costs to O(n), n being number of nodes. but in red-black we can use the sign of number (if numbers being stored are only positive) and hence save space for storing balancing information. there are even other reasons where redblack is mostly prefered.

278. What is a threaded binary tree traversal?
a) a binary tree traversal using stacks
b) a binary tree traversal using queues
c) a binary tree traversal using stacks and queues
d) a binary tree traversal without using stacks and queues

Answer: d
Explanation: This type of tree traversal will not use stack or queue.

279. What are the disadvantages of normal binary tree traversals?
a) there are many pointers which are null and thus useless
b) there is no traversal which is efficient
c) complexity in implementing
d) improper traversals

Answer: a
Explanation: As there are majority of pointers with null value going wasted we use threaded binary trees.

280. In general, the node content in a threaded binary tree is _____
a) leftchild_pointer, left_tag, data, right_tag, rightchild_pointer
b) leftchild_pointer, left_tag
c) leftchild_pointer, left_tag, right_tag, rightchild_pointer
d) leftchild_pointer, left_tag, data

Answer: a
Explanation: It contains additional 2 pointers over normal binary tree node structure.

281. What are null nodes filled with in a threaded binary tree?
a) inorder predecessor for left node and inorder successor for right node information
b) right node with inorder predecessor and left node with inorder successor information
c) they remain null
d) some other values randomly

Answer: a
Explanation: If preorder or postorder is used then the respective predecessor and successor info is stored.

282. Which of the following tree traversals work if the null left pointer pointing to the predecessor and null right pointer pointing to the successor in a binary tree?
a) inorder, postorder, preorder traversals
b) inorder
c) postorder
d) preorder

Answer: a
Explanation: In threaded binary trees, the null left pointer points to the predecessor and the right null pointer point to the successor. In threaded binary trees, we can use in-order, preorder and postorder traversals to visit every node in the tree.

283. What are double and single threaded trees?
a) when both left, right nodes are having null pointers and only right node is null pointer respectively
b) having 2 and 1 node
c) using single and double linked lists
d) using heaps and priority queues

Answer: a
Explanation: They are properties of double and single threaded binary trees respectively.

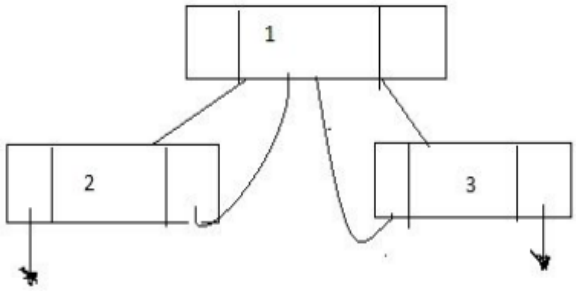
284. What is wrong with below code for inorder traversal of inorder threaded binary tree:

```
  inordertraversal(threadedtreenode root):
  threadedtreenode q =
inorderpredecessor(root)
  while(q!=root):
  q=inorderpredecessor(q)
  print q.data
```

a) inordersuccessor instead of inorderpredecessor must be done
b) code is correct
c) it is code for post order
d) it is code for pre order

Answer: a
Explanation: Property of inorder threaded binary tree is left node with inorder predecessor and right node with inorder successor information are stored.

285. What is inefficient with the below threaded binary tree picture?



a) it has dangling pointers
b) nothing inefficient
c) incorrect threaded tree
d) space is being used more

Answer: a
Explanation: The nodes extreme left and right are pointing to nothing which could be also used efficiently.

286. Which of the following is the most widely used external memory data structure?
a) AVL tree
b) B-tree
c) Red-black tree
d) Both AVL tree and Red-black tree

Answer: b
Explanation: In external memory, the data is transferred in form of blocks. These blocks have data valued and pointers. And B-tree can hold both the data values and pointers. So B-tree is used as an external memory data structure.

287. B-tree of order n is a order-n multiway tree in which each non-root node contains _____
a) at most (n – 1)/2 keys
b) exact (n – 1)/2 keys

c) at least 2n keys
d) at least (n – 1)/2 keys

Answer: d
Explanation: A non-root node in a B-tree of order n contains at least (n – 1)/2 keys. And contains a maximum of (n – 1) keys and n sons.

288. A B-tree of order 4 and of height 3 will have a maximum of _____ keys.
a) 255
b) 63
c) 127
d) 188

Answer: a
Explanation: A B-tree of order m of height h will have the maximum number of keys when all nodes are completely filled. So, the B-tree will have $n = (m^{h+1} – 1)$ keys in this situation. So, required number of maximum keys = $4^{3+1} – 1 = 256 – 1 = 255$.

289. Five node splitting operations occurred when an entry is inserted into a B-tree. Then how many nodes are written?
a) 14
b) 7
c) 11
d) 5

Answer: c
Explanation: If s splits occur in a B-tree, 2s + 1 nodes are written (2 halves of each split and the parent of the last node split). So, if 5 splits occurred, then 2 * 5 + 1 , i.e. 11 nodes are written.

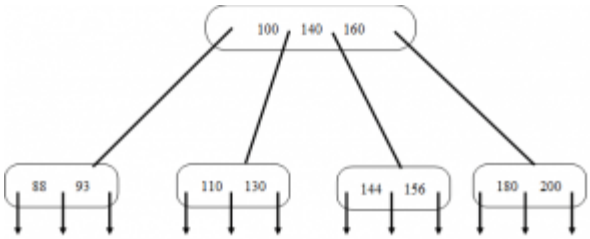
290. B-tree and AVL tree have the same worst case time complexity for insertion and deletion.
a) True
b) False

Answer: a
Explanation: Both the B-tree and the AVL tree have O(log n) as worst case time complexity for insertion and deletion.

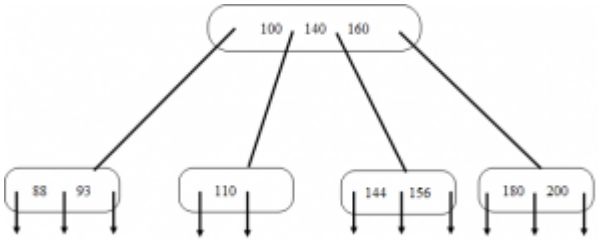
291. 2-3-4 trees are B-trees of order 4. They are an isometric of _____ trees.
a) AVL
b) AA
c) 2-3
d) Red-Black

Answer: d
Explanation: 2-3-4 trees are isometric of Red-Black trees. It means that, for every 2-3-4 tree, there exists a Red-Black tree with data elements in the same order.

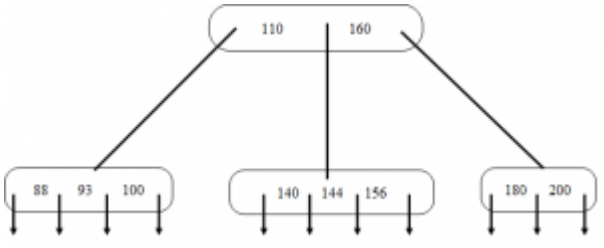
292. Figure shown below is B-tree of order 5. What is the result of deleting 130 from the tree?



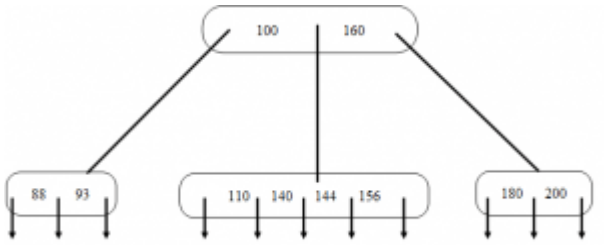
a)



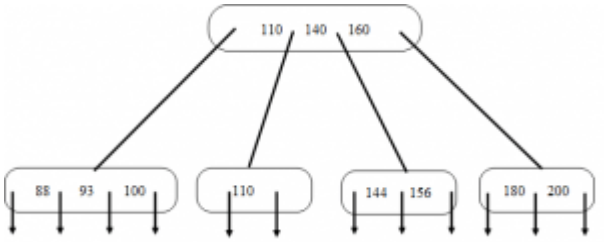
b)



c)



d)



Answer: c
Explanation: Each non-root in a B-tree of order 5 must contain at least 2 keys. Here, when the key 130 is deleted the node gets underflowed i.e. number of keys in the node drops below 2. So we combine the node with key 110 with it's brother node having keys 144 and 156. And this combined node will also contain the separator key from parent i.e. key 140, leaving the root with two keys 110 and 160.

293. What is the best case height of a B-tree of order n and which has k keys?
a) logn (k+1) – 1
b) nk
c) $\log_k (n+1) - 1$
d) $k\log_n$

Answer: a
Explanation: B-tree of order n and with height k has best case height h, where h = logn (k+1) – 1. The best case occurs when all the nodes are completely filled with keys.

294. Compression techniques can be used on the keys to reduce both space and time requirements in a B-tree.
a) True
b) False

Answer: a
Explanation: The front compression and the rear compression are techniques used to reduce space and time requirements in B-tree. The compression enables to retain more keys in a node so that the number of nodes needed can be reduced.

295. Which of the following is true?
a) larger the order of B-tree, less frequently the split occurs
b) larger the order of B-tree, more frequently the split occurs
c) smaller the order of B-tree, more frequently the split occurs
d) smaller the order of B-tree, less frequently the split occurs

Answer: a
Explanation: The average probability of the split is 1/([m / 2] – 1), where m is the order of B-tree. So, if m larger, the probability of split will be less.

296. In a B+ tree, both the internal nodes and the leaves have keys.
a) True
b) False

Answer: b
Explanation: In a B+ -tree, only the leaves have keys, and these keys are replicated in non-leaf nodes for defining the path for locating individual records.

297. Which of the following is true?
a) B + tree allows only the rapid random access
b) B + tree allows only the rapid sequential access
c) B + tree allows rapid random access as well as rapid sequential access
d) B + tree allows rapid random access and slower sequential access

Answer: c
Explanation: The B+ -tree being a variation of B-tree allows rapid random access. In a B+ -tree the leaves are linked together, so it also provides rapid sequential access.

298. A B+ tree can contain a maximum of 7 pointers in a node. What is the minimum number of keys in leaves?
a) 6
b) 3
c) 4
d) 7

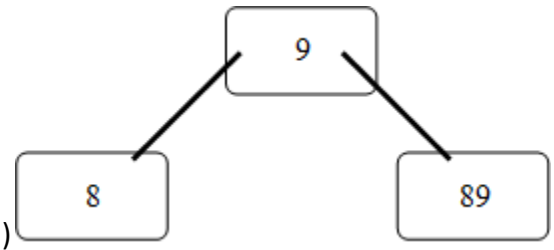
Answer: b
Explanation: Maximum number of pointers in a node is 7, i.e. the order of the B+ -tree is 7. In a B+ tree of order n each leaf node contains at most n – 1 key and at least ⌈(n – 1)/2⌉ keys. Therefore, a minimum number of keys each leaf can have = ⌈(7 – 1)/2⌉ = 3.

299. Which of the following is false?
a) A B+ -tree grows downwards
b) A B+ -tree is balanced
c) In a B+ -tree, the sibling pointers allow sequential searching
d) B+ -tree is shallower than B-tree

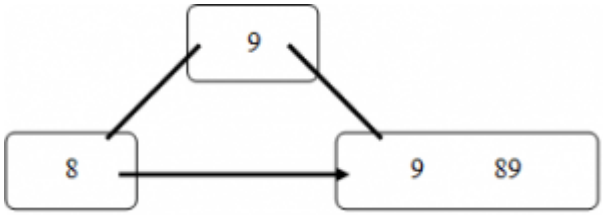
Answer: a
Explanation: A B+ -tree always grows upwards. And In a B+tree – i)The path from the root to every leaf node is of the same length, so the tree is balanced. ii) Leaves are linked, so allow sequential searching. iii) An index is built with a single key per block of data rather than with one key per data record, so it is shallower than B-tree.

300. A B+ -tree of order 3 is generated by inserting 89, 9 and 8. The generated B+ -tree is _____

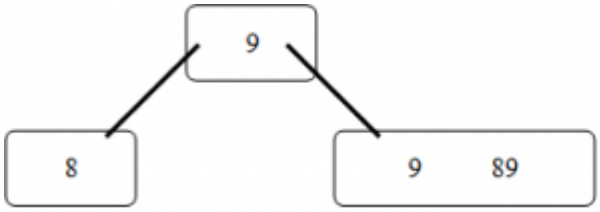
a)



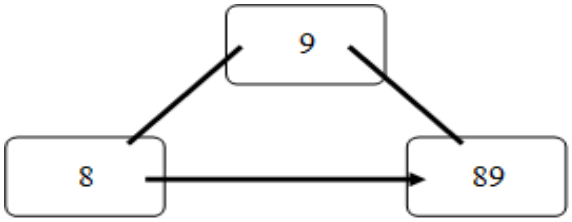
b)



c)

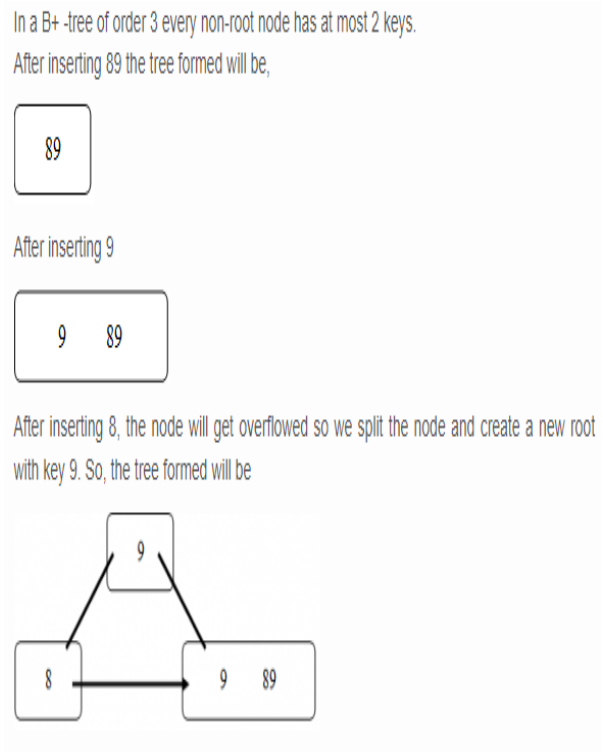


d)



Answer: b

Explanation:



6. Statement 1: When a node is split during insertion, the middle key is promoted to the parent as well as retained in right half-node.

Statement 2: When a key is deleted from the leaf, it is also deleted from the non-leaf nodes of the tree.

a) Statement 1 is true but statement 2 is false

b) Statement 2 is true but statement 1 is false

c) Both the statements are true

d) Both the statements are false

Answer: a

Explanation: During the split, the middle key is retained in the right half node and also promoted to parent node. When a key is deleted from the leaf, it is retained in non-leaves, because it can be still a valid separator between keys in nodes below.

301. Efficiency of finding the next record in B+ tree is _____

a) O(n)

b) O(log n)

c) O(nlog n)

d) O(1)

Answer: d

Explanation: In a B+ -tree finding the next recored (successor) involves accessing an additional leaf at most. So, the efficiency of finding the next record is O(1).

302. What is the maximum number of keys that a B+ -tree of order 3 and of height 3 have?

a) 3

b) 80

c) 27

d) 26

Answer: d
Explanation: A B+ tree of order n and height h can have at most $n^h - 1$ keys. Therefore maximum number of keys = 3^3 -1 = 27 -1 = 26.

303. Which of the following is false?
a) Compared to B-tree, B+ -tree has larger fanout
b) Deletion in B-tree is more complicated than in B+ -tree
c) B+ -tree has greater depth than corresponding B-tree
d) Both B-tree and B+ -tree have same search and insertion efficiencies

Answer: c
Explanation: A B+ -tree has larger fanout and therefore have a depth smaller than that of corresponding B-tree.

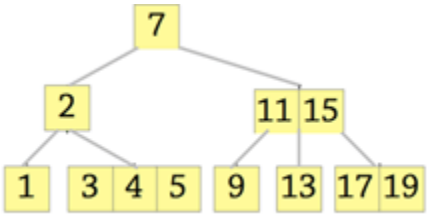
304. Which one of the following data structures are preferred in database-system implementation?
a) AVL tree
b) B-tree
c) B+ -tree
d) Splay tree

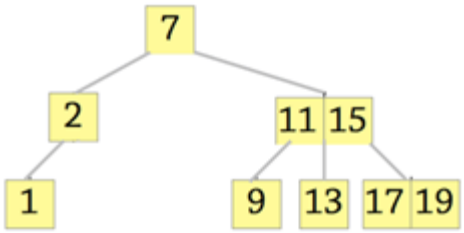
Answer: c
Explanation: The database-system implementations use B+ -tree data structure because they can be used for multilevel indexing.

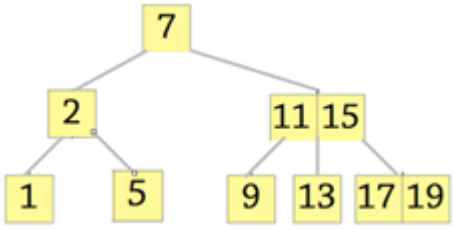
305. 2-3 tree is a specific form of _____
a) B – tree
b) B+ – tree
c) AVL tree
d) Heap

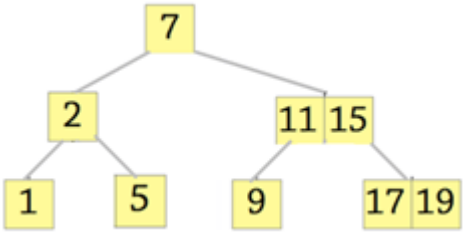
Answer: a
Explanation: The 2-3 trees is a balanced tree. It is a specific form the B – tree. It is B – tree of order 3, where every node can have two child subtrees and one key or 3 child subtrees and two keys.

306. Which of the following is the 2-3 tree?

a)


b)


c)


d)


Answer: c
Explanation: Tree should have two subtrees at node2, but it should not have three elements. The node with elements 11 and 15 should have three child subtrees.

307. AVL trees provide better insertion the 2-3 trees.
a) True
b) False

Answer: b
Explanation: Insertion in AVL tree and 2-3 tree requires searching for proper position for insertion and transformations for balancing the tree. In both, the trees searching takes O(log n) time, but rebalancing in AVL tree takes O(log n), while the 2-3 tree takes O(1). So, 2-3 tree provides better insertions.

308. How many child nodes does each node of Ternary Tree contain?
a) 4
b) 6
c) 5
d) 3

Answer: d
Explanation: Each node of Ternary tree contains at most 3 nodes. So Ternary tree can have 1, 2 or 3 child nodes but not more than that.

309. Which of the following is the name of the node having child nodes?
a) Brother
b) Sister
c) Mother
d) Parent

Answer: d
Explanation: Parent node is the node having child nodes and child nodes may contain references to their parents. Parent node is a node connected by a directed edge to its child.

310. What is the worst case efficiency for a path compression algorithm?
a) O(N)
b) O(log N)
c) O(N log N)
d) O(M log N)

Answer: d
Explanation: The worst case efficiency for a path compression algorithm is mathematically found to be O(M log N).

311. Path Compression algorithm performs in which of the following operations?
a) Create operation
b) Insert operation
c) Find operation
d) Delete operation

Answer: c
Explanation: Path compression algorithm is performed during find operation and is independent of the strategy used to perform unions.

312. What is the definition for Ackermann's function?
a) A(1,i) = i+1 for i>=1
b) A(i,j) = i+j for i>=j
c) A(i,j) = i+j for i = j
d) A(1,i) = i+1 for i<1

Answer: a
Explanation: The Ackermann's function is defined as A(1,i) = i+1 for i>=1. This form in text grows faster and the inverse is slower.

313. Which of the following is not a collision resolution strategy for open addressing?
a) Linear probing
b) Quadratic probing
c) Double hashing
d) Rehashing

Answer: d
Explanation: Linear probing, quadratic probing and double hashing are all collision resolution strategies for open addressing whereas rehashing is a different technique.

314. Hashing can be used in online spelling checkers.
a) True
b) False

Answer: a
Explanation: If misspelling detection is important, an entire dictionary can be pre-hashed and words can be checked in constant time.

315. Which of the following schemes does quadratic probing come under?
a) rehashing
b) extended hashing
c) separate chaining
d) open addressing

Answer: d
Explanation: Quadratic probing comes under open addressing scheme to resolve collisions in hash tables.

316. Which scheme uses a randomization approach?
a) hashing by division
b) hashing by multiplication
c) universal hashing
d) open addressing

Answer: c
Explanation: Universal hashing scheme uses a randomization approach whereas hashing by division and hashing by multiplication are heuristic in nature.

317. Which hash function satisfies the condition of simple uniform hashing?
a) h(k) = lowerbound(km)
b) h(k)= upperbound(mk)
c) h(k)= lowerbound(k)
d) h(k)= upperbound(k)

Answer: a
Explanation: If the keys are known to be random real numbers k independently and uniformly distributed in the range 0<=k<=1, the hash function which satisfies the condition of simple uniform hashing is h(k)= lowerbound(km).

318. A good hash approach is to derive the hash value that is expected to be dependent of any patterns that might exist in the data.
a) True
b) False

Answer: b
Explanation: A hash value is expected to be unrelated or independent of any patterns in the distribution of keys.

319. Interpret the given character string as an integer expressed in suitable radix notation.

Character string = pt

a) 14963
b) 14392
c) 12784
d) 14452

Answer: d
Explanation: The given character string can be interpreted as (112,116) (Ascii values) then expressed as a radix-128 integer, hence the value is 112*128 + 116 = 14452.

320. What is the hash function used in the division method?
a) h(k) = k/m
b) h(k) = k mod m
c) h(k) = m/k
d) h(k) = m mod k

Answer: b
Explanation: In division method for creating hash functions, k keys are mapped into one of m slots by taking the reminder of k divided by m.

321. What can be the value of m in the division method?
a) Any prime number
b) Any even number
c) $2^p - 1$
d) 2^p

Answer: a
Explanation: A prime number not too close to an exact power of 2 is often a good choice for m since it reduces the number of collisions which are likely to occur.

322. Which scheme provides good performance?
a) open addressing
b) universal hashing
c) hashing by division
d) hashing by multiplication

Answer: b
Explanation: Universal hashing scheme provides better performance than other schemes because it uses a unique randomisation approach.

323. Using division method, in a given hash table of size 157, the key of value 172 be placed at position ____
a) 19
b) 72
c) 15
d) 17

Answer: c
Explanation: The key 172 can be placed at position 15 by using the formula
H(k) = k mod m
H(k) = 172 mod 157
H(k) = 15.

324. How many steps are involved in creating a hash function using a multiplication method?
a) 1
b) 4
c) 3
d) 2

Answer: d
Explanation: In multiplication method 2 steps are involved. First multiplying the key value by a constant. Then multiplying this value by m.

325. What is the hash function used in multiplication method?
a) h(k) = floor(m(kA mod 1))
b) h(k) = ceil(m(kA mod 1))
c) h(k) = floor(kA mod m)
d) h(k) = ceil(kA mod m)

Answer: a
Explanation: The hash function can be computed by multiplying m with the fractional part of kA (kA mod 1) and then computing the floor value of the result.

326. What is the advantage of the multiplication method?
a) only 2 steps are involved
b) using constant
c) value of m not critical
d) simple multiplication

Answer: c
Explanation: The value of m can be simply in powers of 2 since we can easily implement the function in most computers. $m=2^p$ where p is an integer.

327. What is the table size when the value of p is 7 in multiplication method of creating hash functions?
a) 14
b) 128
c) 49
d) 127

Answer: b
Explanation: In multiplication method of creating hash functions the table size can be taken in integral powers of 2.
$m = 2^p$
$m= 2^7$
m = 128.

328. What is the value of h(k) for the key 123456?
Given: p=14, s=2654435769, w=32
a) 123
b) 456
c) 70
d) 67

Answer: d
Explanation: $A = s/2^w$
$A = 2654435769/ 2^{32}$
$k.A = 123456 * (2654435769/ 2^{32})$
$= (76300 * 2^{32}) + 17612864$
Hence r1= 76300; r0=17612864
Since w=14 the 14 most significant bits of r0 yield the value of h(k) as 67.

329. What is the average retrieval time when n keys hash to the same slot?
a) Theta(n)
b) Theta(n^2)
c) Theta(nlog n)
d) Big-Oh(n^2)

Answer: a
Explanation: The average retrieval time when n keys hash to the same slot is given by Theta(n) as the collision occurs in the hash table.

330. Collisions can be reduced by choosing a hash function randomly in a way that is independent of the keys that are actually to be stored.
a) True
b) False

Answer: a
Explanation: Because of randomization, the algorithm can behave differently on each execution, providing good average case performance for any input.

331. Double hashing is one of the best methods available for open addressing.
a) True
b) False

Answer: a
Explanation: Double hashing is one of the best methods for open addressing because the permutations produced have many characteristics of randomly chosen permutations.

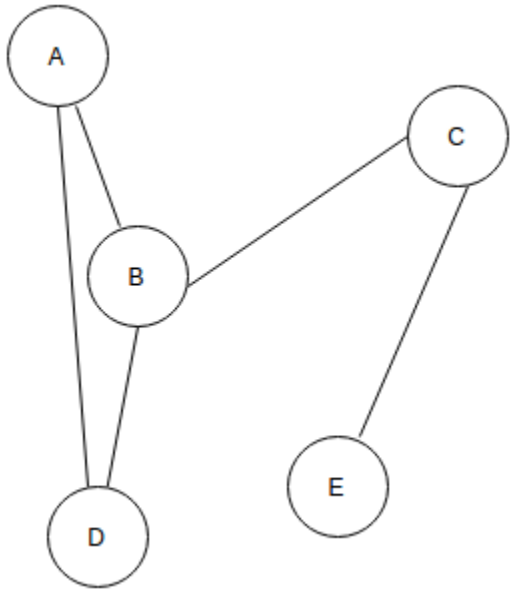
332. What is the hash function used in Double Hashing?
a) (h1(k) – i*h2(k))mod m
b) h1(k) + h2(k)
c) (h1(k) + i*h2(k))mod m
d) (h1(k) + h2(k))mod m

Answer: c
Explanation: Double hashing uses a hash function of the form (h1(k) + i*h2(k))mod m where h1 and h2 are auxiliary hash functions and m is the size of the hash table.

333. Which of the following statements for a simple graph is correct?
a) Every path is a trail
b) Every trail is a path
c) Every trail is a path as well as every path is a trail
d) Path and trail have no relation

Answer: a
Explanation: In a walk if the vertices are distinct it is called a path, whereas if the edges are distinct it is called a trail.

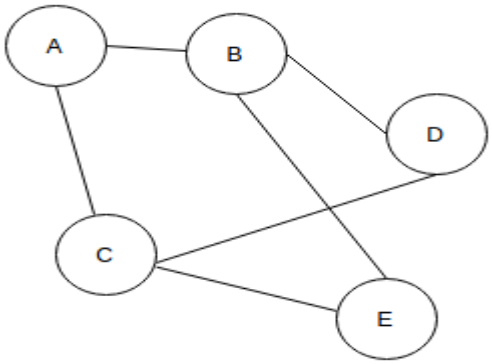
334. In the given graph identify the cut vertices.



a) B and E
b) C and D
c) A and E
d) C and B

Answer: d
Explanation: After removing either B or C, the graph becomes disconnected.

335. For the given graph(G), which of the following statements is true?



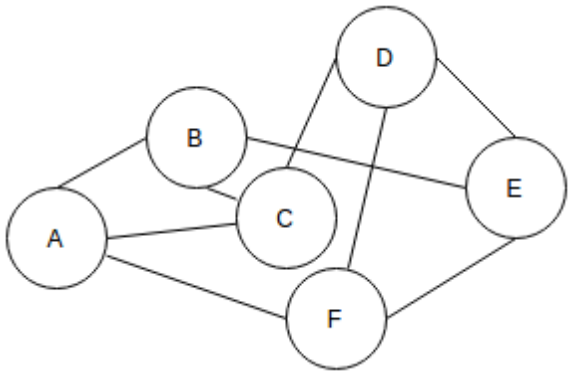
a) G is a complete graph
b) G is not a connected graph
c) The vertex connectivity of the graph is 2
d) The edge connectivity of the graph is 1

Answer: c
Explanation: After removing vertices B and C, the graph becomes disconnected.

336. What is the number of edges present in a complete graph having n vertices?
a) (n*(n+1))/2
b) (n*(n-1))/2
c) n
d) Information given is insufficient

Answer: b
Explanation: Number of ways in which every vertex can be connected to each other is nC2.

337. The given Graph is regular.



a) True
b) False

Answer: a
Explanation: In a regular graph, degrees of all the vertices are equal. In the given graph the degree of every vertex is 3.

338. In a simple graph, the number of edges is equal to twice the sum of the degrees of the vertices.
a) True
b) False

Answer: b
Explanation: The sum of the degrees of the vertices is equal to twice the number of edges.

339. A connected planar graph having 6 vertices, 7 edges contains _____ regions.
a) 15
b) 3
c) 1
d) 11

Answer: b
Explanation: By euler's formula the relation between vertices(n), edges(q) and regions(r) is given by n-q+r=2.

340. If a simple graph G, contains n vertices and m edges, the number of edges in the Graph G'(Complement of G) is _____
a) (n*n-n-2*m)/2
b) (n*n+n+2*m)/2
c) (n*n-n-2*m)/2
d) (n*n-n+2*m)/2

Answer: a
Explanation: The union of G and G' would be a complete graph so, the number of edges in G'= number of edges in the complete form of G(nC2)-edges in G(m).

341. Which of the following properties does a simple graph not hold?
a) Must be connected
b) Must be unweighted
c) Must have no loops or multiple edges
d) Must have no multiple edges

Answer: a
Explanation: A simple graph maybe connected or disconnected.

342. What is the maximum number of edges in a bipartite graph having 10 vertices?
a) 24
b) 21
c) 25
d) 16

Answer: c
Explanation: Let one set have n vertices another set would contain 10-n vertices. Total number of edges would be n*(10-n), differentiating with respect to n, would yield the answer.

343. Which of the following is true?
a) A graph may contain no edges and many vertices
b) A graph may contain many edges and no vertices
c) A graph may contain no edges and no vertices
d) A graph may contain no vertices and many edges

Answer: b
Explanation: A graph must contain at least one vertex.

344. For a given graph G having v vertices and e edges which is connected and has no cycles, which of the following statements is true?
a) v=e
b) v = e+1
c) v + 1 = e
d) v = e-1

Answer: b
Explanation: For any connected graph with no cycles the equation holds true.

345. For which of the following combinations of the degrees of vertices would the connected graph be eulerian?
a) 1,2,3
b) 2,3,4
c) 2,4,5
d) 1,3,5

Answer: a
Explanation: A graph is eulerian if either all of its vertices are even or if only two of its vertices are odd.

346. A graph with all vertices having equal degree is known as a _____
a) Multi Graph
b) Regular Graph
c) Simple Graph
d) Complete Graph

Answer: b
Explanation: The given statement is the definition of regular graphs.

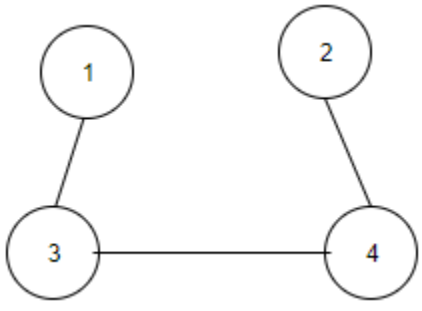
347. Which of the following ways can be used to represent a graph?
a) Adjacency List and Adjacency Matrix
b) Incidence Matrix
c) Adjacency List, Adjacency Matrix as well as Incidence Matrix
d) No way to represent

Answer: c
Explanation: Adjacency Matrix, Adjacency List and Incidence Matrix are used to represent a graph.

348. The number of elements in the adjacency matrix of a graph having 7 vertices is _____
a) 7
b) 14
c) 36
d) 49

Answer: d
Explanation: There are n*n elements in the adjacency matrix of a graph with n vertices.

349. What would be the number of zeros in the adjacency matrix of the given graph?



a) 10
b) 6
c) 16
d) 0

Answer: b
Explanation: Total number of values in the matrix is 4*4=16, out of which 6 entries are non zero.

350. Adjacency matrix of all graphs are symmetric.
a) False
b) True

Answer: a
Explanation: Only undirected graphs produce symmetric adjacency matrices.

351. Incidence matrix and Adjacency matrix of a graph will always have same dimensions?
a) True
b) False

Answer: b
Explanation: For a graph having V vertices and E edges, Adjacency matrix have V*V elements while Incidence matrix have V*E elements.

352. The column sum in an incidence matrix for a simple graph is _____
a) depends on number of edges
b) always greater than 2
c) equal to 2
d) equal to the number of edges

Answer: c
Explanation: For every edge only the vertices with which it is connected would have the value 1 in the matrix, as an edge connects two vertices sum will always be 2.

353. What are the dimensions of an incidence matrix?
a) Number of edges*number of edges
b) Number of edges*number of vertices
c) Number of vertices*number of vertices
d) Number of edges * (½ * number of vertices)

Answer: b
Explanation: Columns may represent edges and vertices may be represented by the rows.

354. A Graph Structured Stack is a _____
a) Undirected Graph
b) Directed Graph
c) Directed Acyclic Graph
d) Regular Graph

Answer: c
Explanation: A Graph Structured Stack is a Directed Acyclic Graph with each path representing a stack.

355. If a Graph Structured Stack contains {1,2,3,4} {1,5,3,4} {1,6,7,4} and {8,9,7,4}, what would be the source and sink vertices of the DAC?
a) Source – 1, 8 Sink – 7,4
b) Source – 1 Sink – 8,4
c) Source – 1, 8 Sink – 4
d) Source – 4, Sink – 1,8

Answer: c
Explanation: Every Stack of the Graph Structured Stack represents a path, each path starts with the source vertex and ends with the sink vertex.

356. Graph Structured Stack finds its application in _____
a) Bogo Sort
b) Tomita's Algorithm
c) Todd–Coxeter algorithm
d) Heap Sort

Answer: b
Explanation: Tomita's is a parsing algorithm which uses Graph Structured Stack in its implementation.

357. Space complexity for an adjacency list of an undirected graph having large values of V (vertices) and E (edges) is _____
a) O(E)
b) O(V*V)
c) O(E+V)
d) O(V)

Answer: c
Explanation: In an adjacency list for every vertex there is a linked list which have the values of the edges to which it is connected.

358. For some sparse graph an adjacency list is more space efficient against an adjacency matrix.
a) True
b) False

Answer: a
Explanation: Space complexity for adjacency matrix is always O(V*V) while space complexity for adjacency list in this case would be O(V).

359. How many of the following statements are correct?
i) All cyclic graphs are complete graphs.

ii) All complete graphs are cyclic graphs.
iii) All paths are bipartite.
iv) All cyclic graphs are bipartite.
v) There are cyclic graphs which are complete.
a) 1
b) 2
c) 3
d) 4

Answer: b
Explanation: Statements iii) and v) are correct.

360. All paths and cyclic graphs are bipartite graphs.
a) True
b) False

Answer: b
Explanation: Only paths and even cycles are bipartite graphs.

361. What is the best case for linear search?
a) O(nlogn)
b) O(logn)
c) O(n)
d) O(1)

Answer: d
Explanation: The element is at the head of the array, hence O(1).

362. What is the worst case for linear search?
a) O(nlogn)
b) O(logn)
c) O(n)
d) O(1)

Answer: c
Explanation: Worst case is when the desired element is at the tail of the array or not present at all, in this case you have to traverse till the end of the array, hence the complexity is O(n).

363. Where is linear searching used?
a) When the list has only a few elements
b) When performing a single search in an unordered list
c) Used all the time
d) When the list has only a few elements and When performing a single search in an unordered list

Answer: d
Explanation: It is practical to implement linear search in the situations mentioned in When the list has only a few elements and When performing a single search in an unordered list, but for larger elements the complexity becomes larger and it makes sense to sort the list and employ binary search or hashing.

364. What is the best case and worst case complexity of ordered linear search?
a) O(nlogn), O(logn)
b) O(logn), O(nlogn)
c) O(n), O(1)
d) O(1), O(n)

Answer: d
Explanation: Although ordered linear search is better than unordered when the element is not present in the array, the best and worst cases still remain the same, with the key element being found at first position or at last position.

365. Which of the following is a disadvantage of linear search?
a) Requires more space
b) Greater time complexities compared to other searching algorithms
c) Not easy to understand
d) Not easy to implement

Answer: b
Explanation: The complexity of linear search as the name suggests is O(n) which is much greater than other searching techniques like binary search(O(logn)). Linear search is easy to implement and understand than other searching techniques.

366. Is there any difference in the speed of execution between linear serach(recursive) vs linear search(Iterative)?
a) Both execute at same speed
b) Linear search(recursive) is faster
c) Linear search(Iterative) is faster
d) Cant be said

Answer: c
Explanation: The Iterative algorithm is faster than the latter as recursive algorithm has overheads like calling function and registering stacks repeatedly.

367. Is the space consumed by the linear search(recursive) and linear search(iterative) same?
a) No, recursive algorithm consumes more space
b) No, recursive algorithm consumes less space
c) Yes
d) Nothing can be said

Answer: a
Explanation: The recursive algorithm consumes more space as it involves the usage the stack space(calls the function numerous times).

368. What is the worst case runtime of linear search(recursive) algorithm?
a) O(n)
b) O(logn)
c) $O(n^2)$
d) O(nx)

Answer: a
Explanation: In the worst case scenario, there might be a need of calling the stack n times. Therfore O(n).

369. Linear search(recursive) algorithm used in _____
a) When the size of the dataset is low
b) When the size of the dataset is large
c) When the dataset is unordered
d) Never used

Answer: a
Explanation: It is used when the size of the dataset is low as its runtime is O(n) which is more when compared to the binary search O(logn).

370. What is the best case runtime of linear search(recursive) algorithm on an ordered set of elements?
a) O(1)
b) O(n)
c) O(logn)
d) O(nx)

Answer: a
Explanation: The best case occurs when the given element to be found is at the first position. Therefore O(1) is the correct answer.

371. Can linear search recursive algorithm and binary search recursive algorithm be performed on an unordered list?
a) Binary search can't be used
b) Linear search can't be used
c) Both cannot be used
d) Both can be used

Answer: a
Explanation: As binary search requires comparison, it is required that the list be ordered. Whereas this doesn't matter for linear search.

372. What is the advantage of recursive approach than an iterative approach?
a) Consumes less memory
b) Less code and easy to implement
c) Consumes more memory
d) More code has to be written

Answer: b
Explanation: A recursive approach is easier to understand and contains fewer lines of code.

373. What is the worst case complexity of binary search using recursion?
a) O(nlogn)
b) O(logn)
c) O(n)
d) $O(n^2)$

Answer: b
Explanation: Using the divide and conquer master theorem.

374. What is the average case time complexity of binary search using recursion?
a) O(nlogn)
b) O(logn)
c) O(n)
d) $O(n^2)$

Answer: b
Explanation: T(n) = T(n/2) + 1, Using the divide and conquer master theorem.

375. Which of the following is not an application of binary search?
a) To find the lower/upper bound in an ordered sequence
b) Union of intervals
c) Debugging
d) To search in unordered list

Answer: d
Explanation: In Binary search, the elements in the list should be sorted. It is applicable only for ordered list. Hence Binary search in unordered list is not an application.

376. Jump search algorithm requires which of the following condition to be true?
a) array should be sorted
b) array should have not be sorted
c) array should have a less than 64 elements
d) array should be partially sorted

Answer: a
Explanation: Jump sort requires the input array to be sorted. The algorithm would fail

to give the correct result if array is not sorted.

377. Which of the following examples represent the worst case input for an insertion sort?
a) array in sorted order
b) array sorted in reverse order
c) normal unsorted array
d) large array

Answer: b
Explanation: The worst case input for an insertion sort algorithm will be an array sorted in reverse order and its running time is quadratic.

378. Which of the following sorting algorithms is the fastest for sorting small arrays?
a) Quick sort
b) Insertion sort
c) Shell sort
d) Heap sort

Answer: b
Explanation: For sorting small arrays, insertion sort runs even faster than quick sort. But, it is impractical to sort large arrays.

379. For the best case input, the running time of an insertion sort algorithm is?
a) Linear
b) Binary
c) Quadratic
d) Depends on the input

Answer: a
Explanation: The best case input for an insertion sort algorithm runs in linear time and is given by O(N).

380. How many passes does an insertion sort algorithm consist of?
a) N
b) N-1
c) N+1
d) N^2

Answer: b
Explanation: An insertion algorithm consists of N-1 passes when an array of N elements is given.

381. Which of the following algorithm implementations is similar to that of an insertion sort?
a) Binary heap
b) Quick sort
c) Merge sort
d) Radix sort

Answer: a
Explanation: Insertion sort is similar to that of a binary heap algorithm because of the use of temporary variable to swap.

382. What is the average case running time of an insertion sort algorithm?
a) O(N)
b) O(N log N)
c) O(log N)
d) $O(N^2)$

Answer: d
Explanation: The average case analysis of a tight bound algorithm is mathematically achieved to be $O(N^2)$.

383. Any algorithm that sorts by exchanging adjacent elements require $O(N^2)$ on average.
a) True
b) False

Answer: a
Explanation: Each swap removes only one inversion, so $O(N^2)$ swaps are required.

384. Binary search can be used in an insertion sort algorithm to reduce the number of comparisons.
a) True
b) False

Answer: a
Explanation: Binary search can be used in an insertion sort algorithm to reduce the number of comparisons. This is called a Binary insertion sort.

385. Which of the following options contain the correct feature of an insertion sort algorithm?
a) anti-adaptive
b) dependable
c) stable, not in-place
d) stable, adaptive

Answer: d
Explanation: An insertion sort is stable, adaptive, in-place and incremental in nature.

386. Which of the following is correct with regard to insertion sort?
a) insertion sort is stable and it sorts In-place
b) insertion sort is unstable and it sorts In-place
c) insertion sort is stable and it does not sort In-place
d) insertion sort is unstable and it does not sort In-place

Answer: a
Explanation: During insertion sort, the relative order of elements is not changed. Therefore, it is a stable sorting algorithm. And insertion sort requires only O(1) of additional memory space. Therefore, it sorts In-place.

387. Which of the following sorting algorithm is best suited if the elements are already sorted?
a) Heap Sort
b) Quick Sort
c) Insertion Sort
d) Merge Sort

Answer: c
Explanation: The best case running time of the insertion sort is O(n). The best case occurs when the input array is already sorted. As the elements are already sorted, only one comparison is made on each pass, so that the time required is O(n).

388. Insertion sort is an example of an incremental algorithm.
a) True
b) False

Answer: a
Explanation: In the incremental algorithms, the complicated structure on n items is built by first building it on n − 1 items. And then we make the necessary changes to fix

things in adding the last item. Insertion sort builds the sorted sequence one element at a time. Therefore, it is an example of an incremental algorithm.

389. What is an in-place sorting algorithm?
a) It needs O(1) or O(logn) memory to create auxiliary locations
b) The input is already sorted and in-place
c) It requires additional storage
d) It requires additional space

Answer: a
Explanation: Auxiliary memory is required for storing the data temporarily.

390. In the following scenarios, when will you use selection sort?
a) The input is already sorted
b) A large file has to be sorted
c) Large values need to be sorted with small keys
d) Small values need to be sorted with large keys

Answer: c
Explanation: Selection is based on keys, hence a file with large values and small keys can be efficiently sorted with selection sort.

391. What is the worst case complexity of selection sort?
a) O(nlogn)
b) O(logn)
c) O(n)
d) $O(n^2)$

Answer: d
Explanation: Selection sort creates a sub-list, LHS of the 'min' element is already sorted and RHS is yet to be sorted. Starting with the first element the 'min' element moves towards the final element.

392. What is the advantage of selection sort over other sorting techniques?
a) It requires no additional storage space
b) It is scalable
c) It works best for inputs which are already sorted
d) It is faster than any other sorting technique

Answer: a
Explanation: Since selection sort is an in-place sorting algorithm, it does not require additional storage.

393. What is the average case complexity of selection sort?
a) O(nlogn)
b) O(logn)
c) O(n)
d) $O(n^2)$

Answer: d
Explanation: In the average case, even if the input is partially sorted, selection sort behaves as if the entire array is not sorted. Selection sort is insensitive to input.

394. What is the disadvantage of selection sort?
a) It requires auxiliary memory
b) It is not scalable
c) It can be used for small keys
d) It takes linear time to sort the elements

Answer: b
Explanation: As the input size increases, the performance of selection sort decreases.

395. What is an external sorting algorithm?
a) Algorithm that uses tape or disk during the sort
b) Algorithm that uses main memory during the sort
c) Algorithm that involves swapping
d) Algorithm that are considered 'in place'

Answer: a
Explanation: As the name suggests, external sorting algorithm uses external memory like tape or disk.

396. What is an internal sorting algorithm?
a) Algorithm that uses tape or disk during the sort
b) Algorithm that uses main memory during the sort
c) Algorithm that involves swapping
d) Algorithm that are considered 'in place'

Answer: b
Explanation: As the name suggests, internal sorting algorithm uses internal main memory.

397. What is the worst case complexity of bubble sort?
a) O(nlogn)
b) O(logn)
c) O(n)
d) $O(n^2)$

Answer: d
Explanation: Bubble sort works by starting from the first element and swapping the elements if required in each iteration.

398. What is the average case complexity of bubble sort?
a) O(nlogn)
b) O(logn)
c) O(n)
d) $O(n^2)$

Answer: d
Explanation: Bubble sort works by starting from the first element and swapping the elements if required in each iteration even in the average case.

399. Which of the following is not an advantage of optimised bubble sort over other sorting techniques in case of sorted elements?
a) It is faster
b) Consumes less memory
c) Detects whether the input is already sorted
d) Consumes less time

Answer: c
Explanation: Optimised Bubble sort is one of the simplest sorting techniques and perhaps the only advantage it has over other techniques is that it can detect whether the input is already sorted. It is faster than other in case of sorted array and consumes less time to describe whether the input array is sorted or not. It consumes same memory than other sorting techniques. Hence it is not an advantage.

400. Merge sort uses which of the following technique to implement sorting?

a) backtracking
b) greedy algorithm
c) divide and conquer
d) dynamic programming

Answer: c
Explanation: Merge sort uses divide and conquer in order to sort a given array. This is because it divides the array into two halves and applies merge sort algorithm to each half individually after which the two sorted halves are merged together.

401. What is the average case time complexity of merge sort?
a) O(n log n)
b) $O(n^2)$
c) $O(n^2 \log n)$
d) $O(n \log n^2)$

Answer: a
Explanation: The recurrence relation for merge sort is given by $T(n) = 2T(n/2) + n$. It is found to be equal to O(n log n) using the master theorem.

402. What is the auxiliary space complexity of merge sort?
a) O(1)
b) O(log n)
c) O(n)
d) O(n log n)

Answer: c
Explanation: An additional space of O(n) is required in order to merge two sorted arrays. Thus merge sort is not an in place sorting algorithm.

403. Merge sort can be implemented using O(1) auxiliary space.
a) true
b) false

Answer: a
Explanation: Standard merge sort requires O(n) space to merge two sorted arrays. We can optimize this merging process so that it takes only constant space. This version is known as in place merge sort.

404. What is the worst case time complexity of merge sort?
a) O(n log n)
b) $O(n^2)$
c) $O(n^2 \log n)$
d) $O(n \log n^2)$

Answer: a
Explanation: The time complexity of merge sort is not affected by worst case as its algorithm has to implement the same number of steps in any case. So its time complexity remains to be O(n log n).

405. Which of the following method is used for sorting in merge sort?
a) merging
b) partitioning
c) selection
d) exchanging

Answer: a
Explanation: Merge sort algorithm divides the array into two halves and applies merge sort algorithm to each half individually after which the two sorted halves are merged together. Thus its method of sorting is called merging.

406. What will be the best case time complexity of merge sort?
a) O(n log n)
b) $O(n^2)$
c) $O(n^2 \log n)$
d) $O(n \log n^2)$

Answer: a
Explanation: The time complexity of merge sort is not affected in any case as its algorithm has to implement the same number of steps. So its time complexity remains to be O(n log n) even in the best case.

407. Merge sort uses which of the following algorithm to implement sorting?
a) backtracking
b) greedy algorithm
c) divide and conquer
d) dynamic programming

Answer: c
Explanation: Merge sort uses the technique of divide and conquer in order to sort a given array. It divides the array into two halves and apply merge sort algorithm to each half individually after which the sorted versions of these halves are merged together.

408. What is the average case time complexity of standard merge sort?
a) O(n log n)
b) $O(n^2)$
c) $O(n^2 \log n)$
d) $O(n \log n^2)$

Answer: a
Explanation: The recurrence relation for merge sort is given by $T(n) = 2T(n/2) + n$. This can be solved using master's theorem and is found equal to O(n log n).

409. What is the auxiliary space complexity of standard merge sort?
a) O(1)
b) O(log n)
c) O(n)
d) O(n log n)

Answer: c
Explanation: The merging of two sorted arrays requires an additional space of n due to which the auxiliary space requirement of merge sort is O(n). Thus merge sort is not an in place sorting algorithm.

410. What is the space complexity of in place merge sort?
a) O(1)
b) O(n)
c) O(log n)
d) O(n log n)

Answer: c
Explanation: Space complexity of in place version of merge sort is O(log n) which is used for storing call stack formed due to recursion. Note that the algorithms with space complexity as O(log n) also qualifies as in place algorithms as the value of log n is close to 1.

411. Quick sort is a _____
a) greedy algorithm
b) divide and conquer algorithm
c) dynamic programming algorithm
d) backtracking algorithm

Answer: b
Explanation: Quick sort is a divide and conquer algorithm. Quick sort first partitions a large array into two smaller sub-arrays. And then recursively sorts the sub-arrays.

412. What is the worst case time complexity of the Quick sort?
a) O(nlogn)
b) O(n)
c) $O(n^3)$
d) $O(n^2)$

Answer: d
Explanation: The worst case running time for Quick sort is $O(n^2)$. In Quick sort, the worst case behaviour occurs when the partitioning routine produces two sub-arrays one with n – 1 element and other with 0 elements.

413. Quick sort is a stable sorting algorithm.
a) True
b) False

Answer: b
Explanation: In stable sorting algorithm the records with equal keys appear in the same order in the sorted sequence as they appear in the input unsorted sequence. Quick sort does not preserve the relative order of equal sort items. Therefore, Quick sort is not a stable sort.

414. Quick sort is a space-optimised version of ____
a) Bubble sort
b) Selection sort
c) Insertion sort
d) Binary tree sort

Answer: d
Explanation: Quick sort is a space-optimised version of the binary tree sort. In binary sort tree, the elements are inserted sequentially into the binary search tree and Quick sort organises elements into a tree that is implied by the recursive calls.

415. QuickSort can be categorized into which of the following?
a) Brute Force technique
b) Divide and conquer
c) Greedy algorithm
d) Dynamic programming

Answer: b
Explanation: First you divide(partition) the array based on the pivot element and sort accordingly.

416. Heap sort is an implementation of _____ using a descending priority queue.
a) insertion sort
b) selection sort
c) bubble sort
d) merge sort

Answer: b
Explanation: Heap sort is an implementation of selection sort using the input array as a heap representing a descending priority queue. Heap sort algorithm is divided into two phase. In first phase the max-heap is created and the second phase (selection phase) deletes the

elements from the priority queue using siftdown operation.

417. Which of the following sorting algorithm is used by C++ internally?
a) quicksort
b) introsort
c) merge sort
d) heap sort

Answer: b
Explanation: Introsort is the in built sorting algorithm used by C++. It is an example of a hybrid sorting algorithm which means it uses more than one sorting algorithm as a routine.

418. Which of the following sorting algorithm is not a constituent of introsort?
a) selection sort
b) quicksort
c) insertion sort
d) heap sort

Answer: a
Explanation: Introsort is a hybrid sorting algorithm which means it uses more than one sorting algorithm as a routine. It may use quick sort or heap sort or insertion sort depending on the given situation.

419. Which of the following is Python's standard sorting algorithm?
a) quick sort
b) introsort
c) merge sort
d) tim sort

Answer: d
Explanation: Tim sort has been python's standard sorting algorithm since its version 2.3. It is an example of hybrid sorting algorithm which means it uses more than one sorting algorithm as a routine.

420. Which of the following sorting algorithm is a constituent of tim sort?
a) selection sort
b) quick sort
c) merge sort
d) heap sort

Answer: c
Explanation: Tim sort is a hybrid sorting algorithm which means it uses more than one sorting algorithm as a routine. It is derived from insertion sort and merge sort.

421. Tim sort begins sorting the given array by using which of the following sorting algorithm?
a) selection sort
b) quick sort
c) insertion sort
d) merge sort

Answer: c
Explanation: Tim sort begins sorting any given array by using insertion sort for each run. The array is divided into smaller parts for this purpose, each part having a size equal to value of run. Then these small parts called runs are merged in order to obtain sorted array

422. Which of the following is an example of parallel sorting technique?
a) bogo sort
b) sleep sort

c) cube sort
d) merge sort

Answer: c
Explanation: Out of the given options only cube sort is a parallel sorting algorithm. It builds self balancing multi dimensional arrays from the input keys.

423. What is the worst case time complexity of cube sort?
a) O(n)
b) O(log n)
c) O(n log n)
d) O(n2)

Answer: c
Explanation: The worst case performance of cube sort is O(n log n). This is the fastest possible complexity with a comparison based sort.

424. What is the auxiliary space requirement of cube sort?
a) O(n)
b) O(1)
c) O(log n)
d) O(n log n)

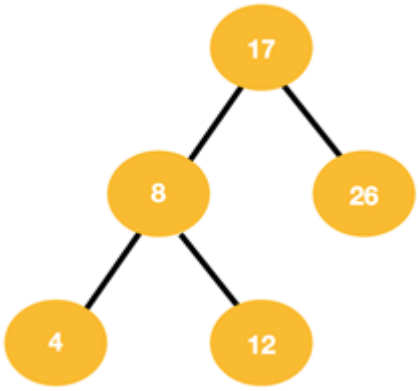
Answer: a
Explanation: Cube sort requires an auxiliary space of O(n). This is the worst case of auxiliary space complexity.

425. What is the best case time complexity of cube sort?
a) $O(n^2)$
b) O(n)
c) O(n log n)
d) O(1)

Answer: b
Explanation: Best case time complexity of cube sort occurs when the input array is almost sorted. So in such a case only O(n) time is required for sorting.

426. Consider the original array 17 8 12 4 26. How many comparisons are needed to construct the BST on the original array?
a) 5
b) 4
c) 7
d) 10

Answer: d
Explanation: Original array is 17 8 12 4 26. The BST built on this array is shown in the figure below.



To built the BST, we travel down the tree until a leaf is reached. Therefore, for every element we compare the element with the internal nodes until we the leaves and then once again compare the element with its parent to decide whether it is right child or left child. So, for given array we need to perform 10 comparisons to build the BST.

427. In binary tree sort, we first construct the BST and then we perform _____ traversal to get the sorted order.

a) inorder
b) postorder
c) preorder
d) level order

Answer: a
Explanation: In binary tree sort is a sort algorithm where a binary search tree is built from the elements to be sorted, and then we perform inorder traversal on the BST to get the elements in sorted order.

428. What is the worst case time complexity of the binary tree sort?
a) O(n)
b) O(nlogn)
c) $O(n^2)$
d) O(logn)

Answer: c
Explanation: For the binary tree sort the worst case when the BST constructed is unbalanced. BST gets unbalanced when the elements are already sorted. So, in the worst case, $O(n^2)$ time is required to built the BST and O(n) time to traverse the tree. Therefore, the worst case time complexity is $O(n^2) + O(n) = O(n^2)$.

429. Which of the following is an example of an unstable sorting algorithm?
a) cycle sort
b) insertion sort
c) bubble sort
d) merge sort

Answer: a
Explanation: Out of the given options only cycle sort is an unstable sorting algorithm. This implies that the relative position of equal valued elements in the input and sorted array does not remain the same when we use cycle sort.

431. Euclid's algorithm is used for finding _____
a) GCD of two numbers
b) GCD of more than three numbers
c) LCM of two numbers
d) LCM of more than two numbers

Answer: a
Explanation: Euclid's algorithm is basically used to find the GCD of two numbers. It cannot be directly applied to three or more numbers at a time.

432. Who invented Euclid's algorithm?
a) Sieve
b) Euclid
c) Euclid-Sieve
d) Gabriel lame

Answer: b
Explanation: Euclid invented Euclid's algorithm. Sieve provided an algorithm for finding prime numbers. Gabriel lame proved a theorem in Euclid's algorithm.

433. If 4 is the GCD of 16 and 12, What is the GCD of 12 and 4?
a) 12
b) 6
c) 4
d) 2

Answer: c
Explanation: Euclid's algorithm states that the GCD of two numbers does not change even if the bigger number is replaced by a

difference of two numbers. So, GCD of 16 and 12 and 12 and (16-12)=4 is the same.

435. Which of the following is not an application of Euclid's algorithm?
a) Simplification of fractions
b) Performing divisions in modular arithmetic
c) Solving quadratic equations
d) Solving diophantine equations

Answer: c
Explanation: Solving quadratic equations is not an application of Euclid's algorithm whereas the rest of the options are mathematical applications of Euclid's algorithm.

436. Strassen's algorithm is a/an_____ algorithm.
a) Non- recursive
b) Recursive
c) Approximation
d) Accurate

Answer: b
Explanation: Strassen's Algorithm for matrix multiplication is a recursive algorithm since the present output depends on previous outputs and inputs.

437. What is the running time of Strassen's algorithm for matrix multiplication?
a) $O(n^{2.81})$
b) $O(n^3)$
c) $O(n^{1.8})$
d) $O(n^2)$

Answer: a
Explanation: Strassen's matrix algorithm requires only 7 recursive multiplications of n/2 x n/2 matrix and $Theta(n^2)$ scalar additions and subtractions yielding the running time as $O(n^{2.81})$.

438. What is the running time of naïve matrix multiplication algorithm?
a) $O(n^{2.81})$
b) $O(n^4)$
c) O(n)
d) $O(n^3)$

Answer: d
Explanation: The traditional matrix multiplication algorithm takes $O(n^3)$ time. The number of recursive multiplications involved in this algorithm is 8.

439. Strassen's matrix multiplication algorithm follows _____ technique.
a) Greedy technique
b) Dynamic Programming
c) Divide and Conquer
d) Backtracking

Answer: c
Explanation: Strassen's matrix multiplication algorithm follows divide and conquer technique. In this algorithm the input matrices are divided into n/2 x n/2 sub matrices and then the recurrence relation is applied.

440. The number of scalar additions and subtractions used in Strassen's matrix multiplication algorithm is _____
a) $O(n^{2.81})$
b) $Theta(n^2)$
c) Theta(n)
d) $O(n^3)$

Answer: b
Explanation: Using Theta(n^2) scalar additions and subtractions, 14 matrices are computed each of which is n/2 x n/2. Then seven matrix products are computed recursively.

441. What is meant by number theory?
a) study of integers
b) study of complex numbers
c) numerology
d) theory of origination of mathematics

Answer: a
Explanation: Number theory is a branch of mathematics that deals with the study of integers. Partitioning of a number comes under the study of number theory.

442. What is meant by the power set of a set?
a) subset of all sets
b) set of all subsets
c) set of particular subsets
d) empty set

Answer: b
Explanation: Power set of a set is defined as the set of all subsets. Ex- S={1,2} then P={{},{1},{2}{1,2}}.

443. Number of elements in the power set of set S={1,2,3} will be?
a) 2
b) 4
c) 6
d) 8

Answer: d
Explanation: Power set of a set is defined as the set of all subsets. Number of elements in the power set of a set having n elements is given as 2^n. Thus, here number of elements will be 2^3=8.

444. Number of elements in the power set of set S={1,2,2} will be?
a) 2
b) 4
c) 6
d) 8

Answer: c
Explanation: For finding the number of elements in the power set of the given set we need to remove duplicates. So we will be left with 6 unique elements which will be P={{},{1},{2},{1,2},{2,2},{1,2,2}}.

445. Depth First Search is equivalent to which of the traversal in the Binary Trees?
a) Pre-order Traversal
b) Post-order Traversal
c) Level-order Traversal
d) In-order Traversal

Answer: a
Explanation: In Depth First Search, we explore all the nodes aggressively to one path and then backtrack to the node. Hence, it is equivalent to the pre-order traversal of a Binary Tree.

446. Time Complexity of DFS is? (V – number of vertices, E – number of edges)
a) O(V + E)
b) O(V)
c) O(E)
d) O(V*E)

Answer: a
Explanation: The Depth First Search explores every node once and every edge once (in worst case), so it's time complexity is O(V + E).

447. The Data structure used in standard implementation of Breadth First Search is?
a) Stack
b) Queue
c) Linked List
d) Tree

Answer: a
Explanation: The Depth First Search is implemented using recursion. So, stack can be used as data structure to implement depth first search.

448.Breadth First Search is equivalent to which of the traversal in the Binary Trees?
a) Pre-order Traversal
b) Post-order Traversal
c) Level-order Traversal
d) In-order Traversal

Answer: c
Explanation: The Breadth First Search Algorithm searches the nodes on the basis of level. It takes a node (level 0), explores it's neighbors (level 1) and so on.

449. Time Complexity of Breadth First Search is? (V – number of vertices, E – number of edges)
a) O(V + E)
b) O(V)
c) O(E)
d) O(V*E)

Answer: a
Explanation: The Breadth First Search explores every node once and every edge once (in worst case), so it's time complexity is O(V + E).

450. The Data structure used in standard implementation of Breadth First Search is?
a) Stack
b) Queue
c) Linked List
d) Tree

Answer: b
Explanation: The Breadth First Search explores every node once and put that node in queue and then it takes out nodes from the queue and explores it's neighbors.