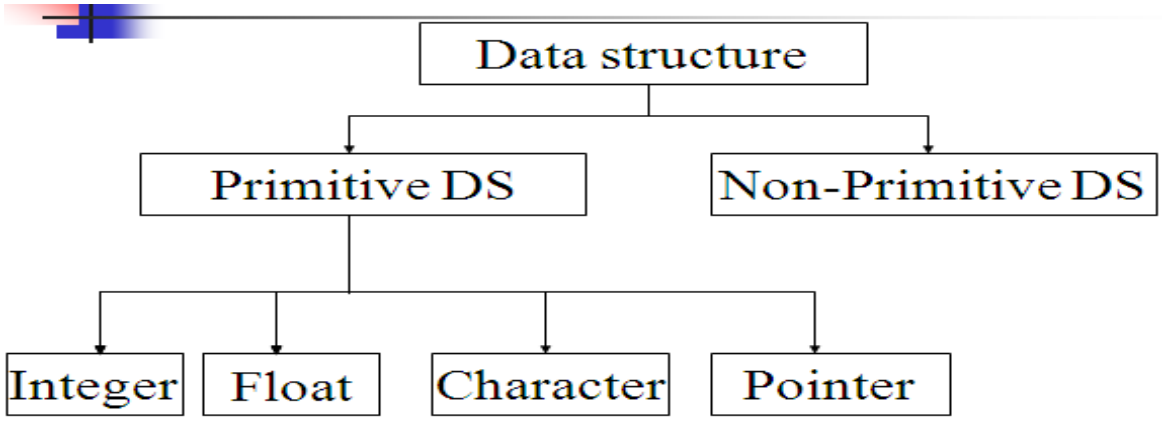
UNIT - I

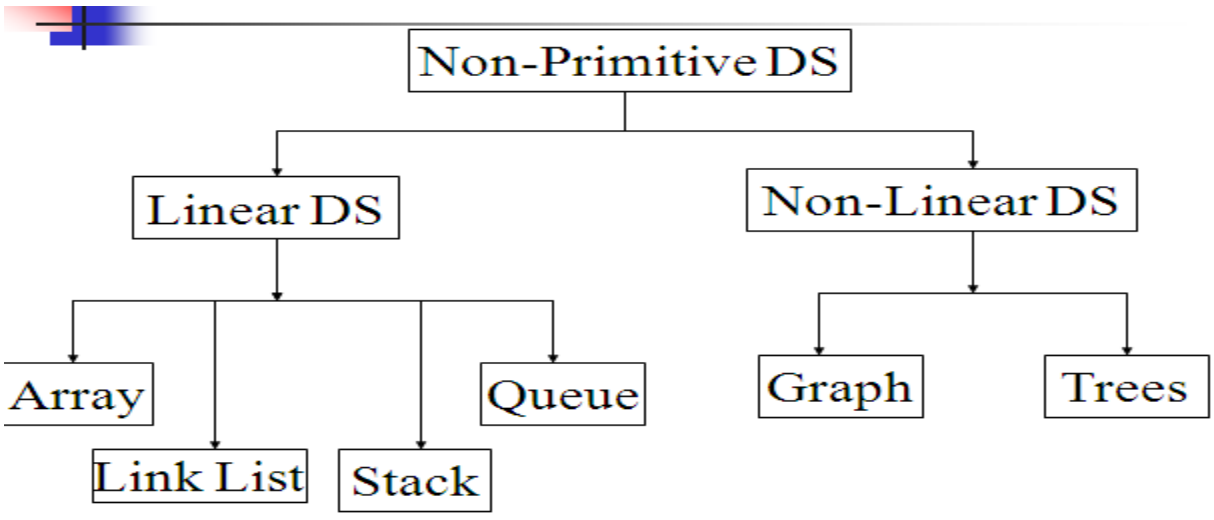
Introduction to Data Structures

- Data structure is collection of data elements and which defines a particular way of storing and organizing data in a computer so that it can be used efficiently.
- In other words we can store and retrieve data easily depends on the user application

Classification of Data Structure

- Data structure are normally divided into two broad categories:
    - Primitive Data Structure
    - Non-Primitive Data Structure





Primitive Data Structure

- There are basic structures and directly operated upon by the machine instructions.
- In general, there are different representation on different computers.
- Integer, Floating-point number, Character constants, string constants, pointers etc, fall in this category.

Non-Primitive Data Structure

- There are more sophisticated data structures.
- These are derived from the primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.
- Linked List, Stack, Queue, Tree, Graph are example of non-primitive data structures.
- The design of an efficient data structure must take operations to be performed on the data structure.

Data Structure Operations

The most commonly used operation on data structure are broadly categorized into following types:

- Traversing
- Searching
- Inserting
- Deleting
- Sorting
- Merging

Traversing- It is used to access each data item exactly once so that it can be processed.

Searching- It is used to find out the location of the data item if it exists in the given collection of data items.

Inserting- It is used to add a new data item in the given collection of data items.

Deleting- It is used to delete an existing data item from the given collection of data items.

Sorting- It is used to arrange the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.

Merging- It is used to combine the data items of two sorted files into single file in the sorted form.

Abstract Data Types

- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called "abstract" because it gives an implementation independent view.
- The process of providing only the essentials and hiding the details is known as abstraction. For example, we have been using int, float, char data types only with the knowledge with values that can take and operations that can be performed on them without any idea of how these types are implemented. So a user only needs to know what a data type can do but not how it will do it.

Stack ADT

A Stack contains elements of same type arranged in sequential order.

- All operations takes place at a single end that is top of the stack and following operations can be performed:
  push() – Insert an element at one end of the stack called top.
  pop () – Remove and return the element at the top of the stack, if it is not empty.

Queue ADT

A Queue contains elements of same type arranged in sequential order.

- Operations takes place at both ends, insertion is done at end and deletion is done at front. Following operations can be performed:
  enqueue() – Insert an element at the end of the queue.
  dequeue() – Remove and return the first element of queue, if the queue is not empty.

What is an algorithm?

An algorithm is a step by step procedure to solve a problem. In normal language, algorithm is defined as a sequence of statements which are used to perform a task. In computer science, an algorithm can be defined as follows...

An algorithm is a sequence of unambiguous instructions used for solving a problem, which can be implemented (as a program) on a computer.

Different Approaches to Design an Algorithm

- Divide and conquer
- Greedy Method
- Dynamic Programming
- Backtracking
- Branch & Bound

Divide and Conquer Method

The divide and conquer approach is applied in the following algorithms

- Binary search
- Quick sort
- Merge sort

Dynamic Programming

Recursive algorithm for Fibonacci Series is an example of dynamic programming.

Greedy Method

A greedy algorithm works recursively creating a group of objects from the smallest possible component parts. Recursion is a procedure to solve a problem in which the solution to a specific problem is dependent on the solution of the smaller instance of that problem.

Branch and Bound

The purpose of a branch and bound search is to maintain the lowest-cost path to a target. Once a solution is found, it can keep improving the solution. Branch and bound search is implemented in depth-bounded search and depth–first search.

Backtracking

Can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

Recursive Algorithm

A **recursive algorithm** is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input.

In general, recursive computer programs require more memory and computation compared with iterative algorithms, but they are simpler and for many cases a natural way of thinking about the problem.

Factorial

Factorial is an important mathematical function. A recursive definition of factorial is as follows.

$$n! = \begin{cases} 1 & \text{if } n \leq 1 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

Linear Search Algorithm

What is Search?

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Linear Search Algorithm (Sequential Search Algorithm)

Linear search algorithm finds a given element in a list of elements with **O(n)** time complexity where **n** is total number of elements in the list. This search process starts comparing search element with the first element in the list. If both are matched then result is element found otherwise search element is compared with next element in the list. Repeat the same until search element is compared with last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". That means, the search element is compared with element by element in the list.

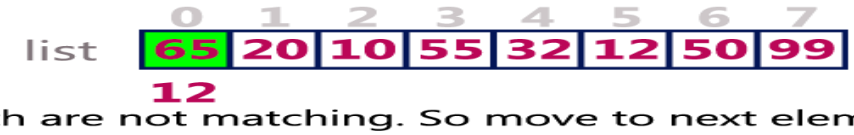
Linear search is implemented using following steps...

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with the first element in the list.
- **Step 3 -** If both are matched, then display "Given element is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then compare search element with the next element in the list.
- **Step 5 -** Repeat steps 3 and 4 until search element is compared with last element in the list.
- **Step 6 -** If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

search element 12

Step 1:

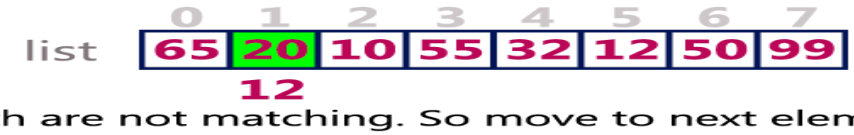
search element (12) is compared with first element (65)



Both are not matching. So move to next element

Step 2:

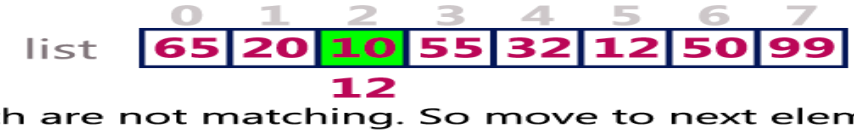
search element (12) is compared with next element (20)



Both are not matching. So move to next element

Step 3:

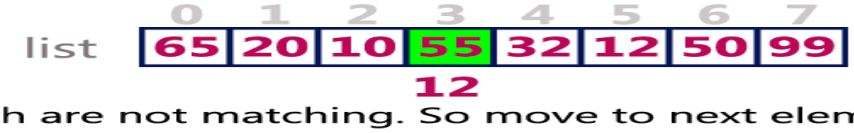
search element (12) is compared with next element (10)



Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)



Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

Both are matching. So we stop comparing and display element found at index 5.

Linear Search Program

//AIM: write a program to search the given element by using linear search and binary search

```
#include<stdio.h>
#include<conio.h>
void main()
 {
 int i,j,a[100],key,n,flag=0;
 clrscr();
 printf("Number of elements in the array\n");
 scanf("%d",&n);
 printf("\n Enter %d elements\n",n);
 for(i=0;i<n;i++)
 scanf("%d",&a[i]);
 printf("Element to be searched\n");
 scanf("%d",&key);
  for(i=0;i<n;i++)
   {
    if(key==a[i])
     {
       flag=1;
       break;
      }
    }
    if(flag==1)
    printf("key  value found at %d index",i);
    else
    printf("key value not found");
    getch();
}
```

Binary Search Algorithm

What is Search?

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Binary Search Algorithm

Binary search algorithm finds a given element in a list of elements with **O(log n)** time complexity where **n** is total number of elements in the list. The binary search algorithm can be used with only sorted list of elements. That means, binary search is used only with list of elements that are already arraged in an order. The binary search can not be used for list of elements arranged in random order. This search process starts comparing the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sublist of the middle element. If the search element is larger, then we repeat the same process for right sublist of the middle element. We repeat this process until we find the search element in the list or until we left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is " Element not found in the list".

Binary search is implemented using following steps...

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Find the middle element in the sorted list.
- **Step 3 -** Compare the search element with the middle element in the sorted list.
- **Step 4 -** If both are matched, then display "Given element is found!!!" and terminate the function.
- **Step 5 -** If both are not matched, then check whether the search element is smaller or larger than the middle element.
- **Step 6 -** If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- **Step 7 -** If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- **Step 8 -** Repeat the same process until we find the search element in the list or until sublist contains only one element.
- **Step 9 -** If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

Example

Consider the following list of elements and the element to be searched...

Step 1:

search element (12) is compared with middle element (50)

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

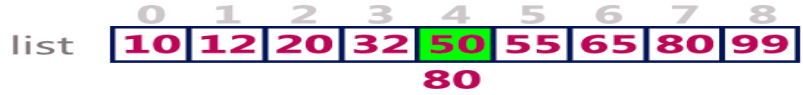
Step 2:

search element (12) is compared with middle element (12)

Both are matching. So the result is "Element found at index 1"

Step 1:

search element (80) is compared with middle element (50)



Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

Step 2:

search element (80) is compared with middle element (65)

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

Step 3:

search element (80) is compared with middle element (80)

Both are not matching. So the result is "Element found at index 7"

Binary Search Program

//AIM: write a program to search the given element by using binary search

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[100],i,key,n,flag=0,high,mid,low=0;
clrscr();
printf("enter the size array:");
scanf("%d",&n);
printf("enter an element of array:");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("enter the key value:");
scanf("%d",&key);
high=n-1;
while(low<=high)
{
mid=(low+high)/2;
if(key==a[mid])
{
flag=1;
break;
}
else if(key<a[mid])
high=mid-1;
else if(key>a[mid])
low=mid+1;
}
if(flag==0)
printf("key value notfound:");
else
printf("key value found at %d index:",mid);
getch();
}
```

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.

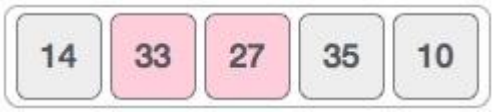
How Bubble Sort Works?

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.

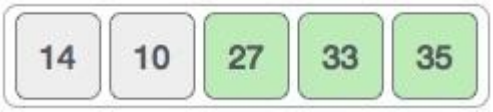
Then we move to the next two values, 35 and 10.

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

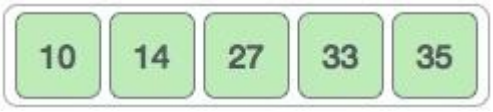
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

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.

Bubble Sort Program

```
#include <stdio.h>
#include <conio.h>
void main()
{
  int a[100], n, i, j, s;
  clrscr();
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
    for (i = 0; i < n; i++)
    scanf("%d", &a[i]);
    for (i= 0 ; i < ( n - 1 ); i++)
     {
       for (j = 0 ; j < n - i- 1; j++)
            {
              if (a[j] > a[j+1])
               {
                 s= a[j];
                 a[j]   = a[j+1];
                 a[j+1] = s;
               }
            }
     }

     printf("Sorted list in ascending order:\n");
     for ( i = 0 ; i < n ; i++ )
     printf("%d\n", a[i]);
     getch();
}
```

Selection Sort Algorithm

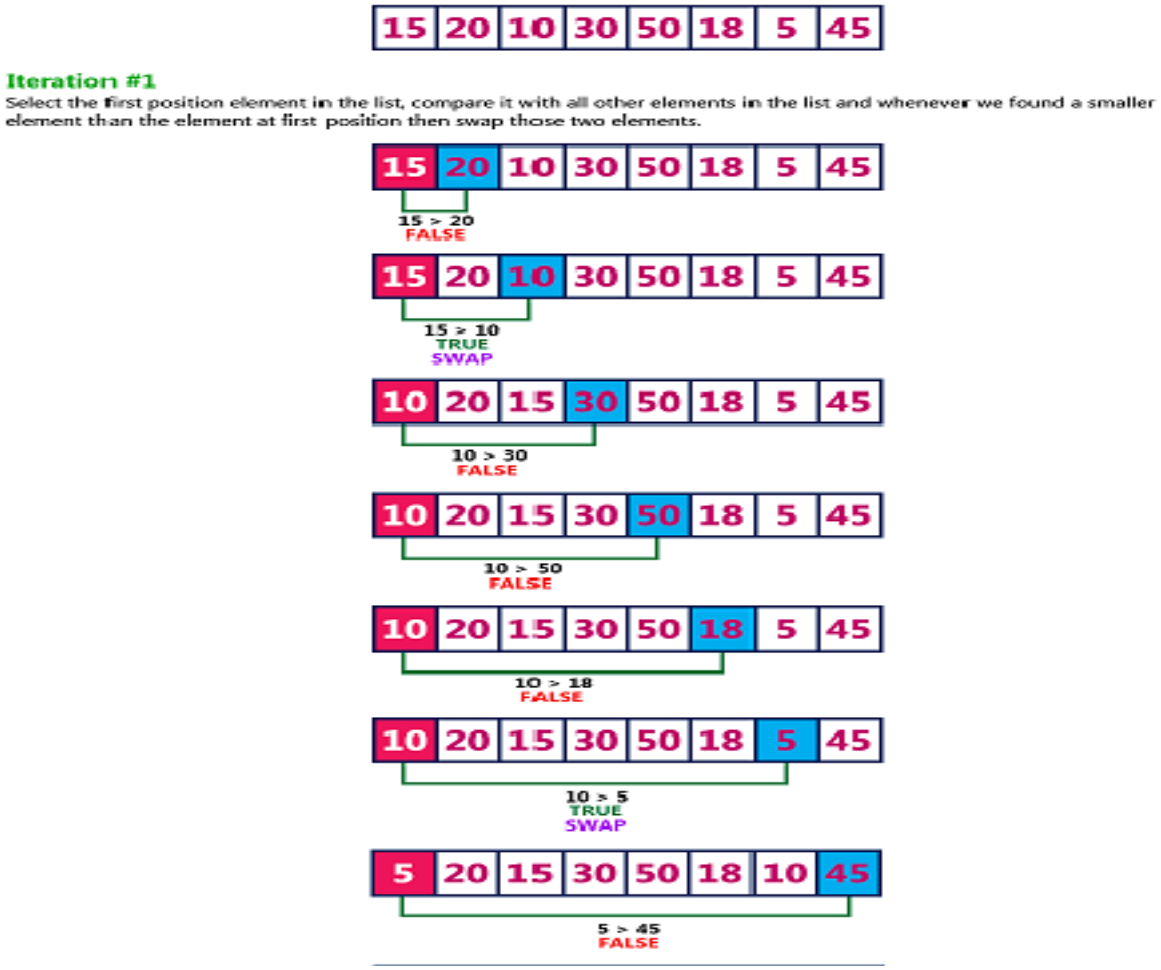
Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with all the remaining elements in the list. If any element is smaller than the selected element (for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

Step by Step Process

The selection sort algorithm is performed using 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 comparison, 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.

Example



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

Selection sort program

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,j,temp,n,a[20];
clrscr();
printf("Enter the number of elements:");
scanf("%d",&n);
printf("\nEnter the elements:");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
for(i=0;i<n;i++)
for(j=i+1;j<n;j++)
if(a[i]>a[j])
{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
printf("\nElements after sorting:");
for(i=0;i<n;i++)
printf("\n%d",a[i]);
getch();
}
```

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

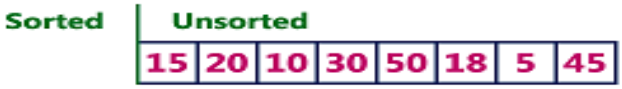
The insertion sort algorithm is performed using following steps...

- **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

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.

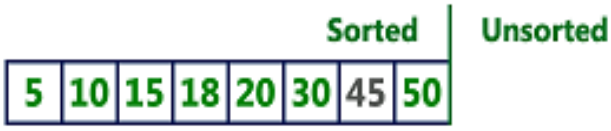
To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.

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...

Insertion sort program

```
#include<stdio.h>
void main()
{
 int i,j,t,a[10],n,p=0;
clrscr();
printf("enter the range of array:");
scanf("%d",&n);
printf("enter elements into array:");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
for(i=1;i<n;i++)
{
t=a[i];
for(p=i;p>0 && a[p-1]>t;p--)
a[p]=a[p-1];
a[p]=t;
}
printf("the sorted order is:");
for(i=0;i<n;i++)
printf("\t%d",a[i]);
getch();
}
```

Quick Sort

Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.
There are many different versions of quick Sort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quick Sort is partition().

Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.

All this should be done in linear time.

Illustration of partition() :

arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes: 0 1 2 3 4 5 6
low = 0, high = 6, pivot = arr[h] = 70
Initialize index of smaller element, **i = -1**

Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j // are same
j = 1 : Since arr[j] > pivot, do nothing // No change in i and arr[]
j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing // No change in i and arr[]
j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped
We come out of loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped
Now 70 is at its correct place.
All elements smaller than
70 are before it and all elements greater than 70 are after it.

Quick sort program:

```
#include<stdio.h>
#include<conio.h>
void quicksort(int[],int,int);
void main()
{
 int x[20],i,n;
 clrscr();
 printf("Enter size of list:");
 scanf("%d",&n);
 printf("Enter %d elements:",n);
 for(i=0;i<n;i++)
 scanf("%d",&x[i]);
 quicksort(x,0,n);
 printf("Sorted List:");
 for(i=0;i<n;i++)
 printf("%d\t",x[i]);
 getch();
}
void quicksort(int x[20],int first,int last)
{
 int pivot,i,j,t;
 if(first<last)
 {
  pivot=first;
  i=first;
  j=last;
  while(i<j)
  {
   while(x[i]<=x[pivot] && i<last)
   i++;
   while(x[j]>x[pivot])
   j--;
   if(i<j)
   {
    t=x[i];
    x[i]=x[j];
    x[j]=t;
   }
  }
  t=x[pivot];
  x[pivot]=x[j];
  x[j]=t;
  quicksort(x,first,j-1);
  quicksort(x,j+1,last);
 }}
```

Merge Sort

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.
The merge() function is used for merging two halves.
The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:

   middle m = (l+r)/2

2. Call mergeSort for first half:

   Call mergeSort(arr, l, m)

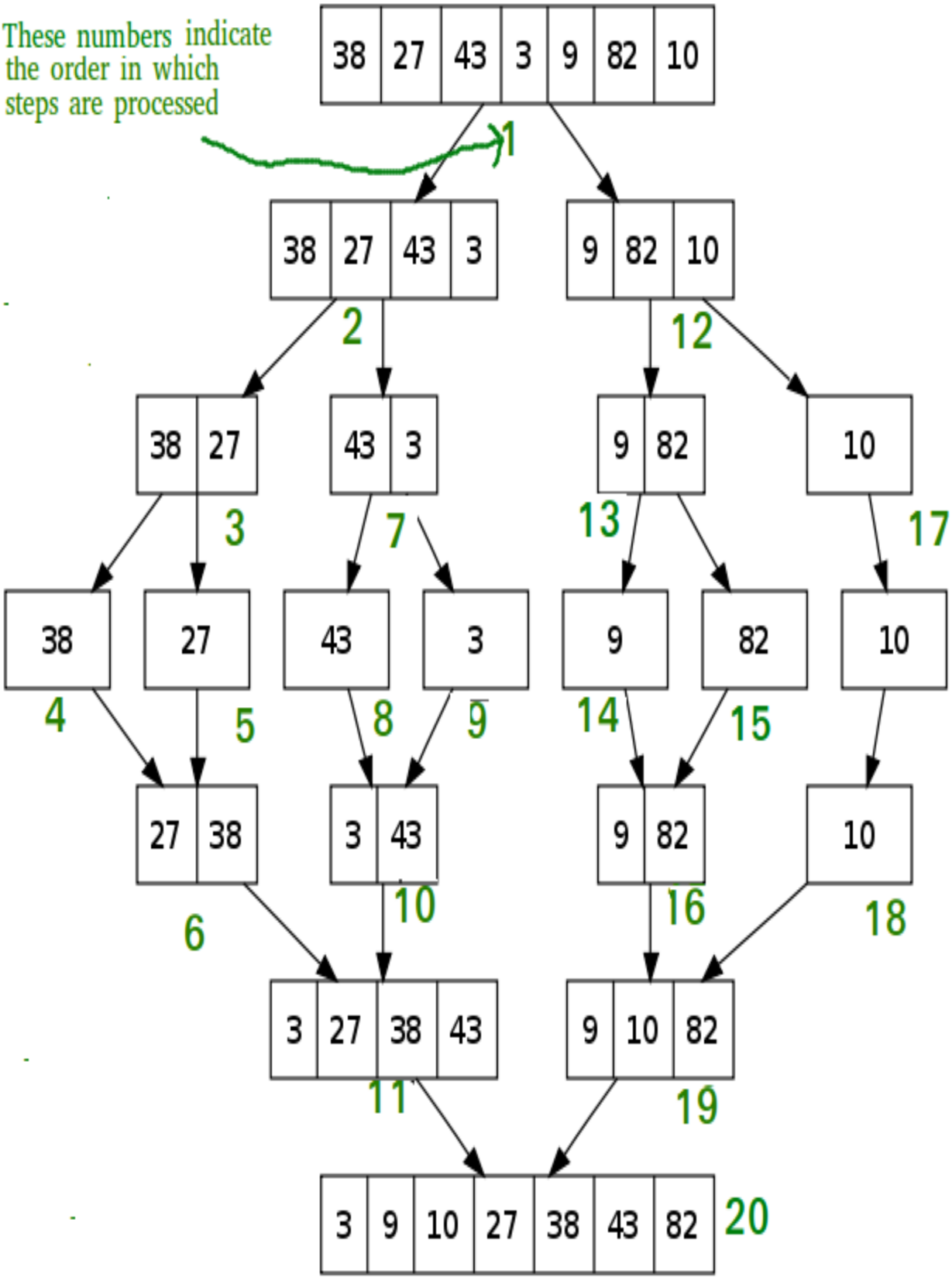
3. Call mergeSort for second half:

   Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

   Call merge(arr, l, m, r)

Example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

Merge Sort program:

```
#include <stdio.h>
#include <stdlib.h>
void merge(int a[],int,int,int);
void mergesort(int a[],int,int);
void main()
{
 int a[20],i,n;
 clrscr();
 printf("Enter size of List:");
 scanf("%d",&n);
 printf("\nEnter %d elements: \n",n);
 for(i=0;i<n;i++)
 scanf("%d",&a[i]);
 mergesort(a,0,n-1);
 printf("Sorted List:");
 for(i=0;i<n;i++)
 printf("%d\t",a[i]);
 getch();
}
void mergesort(int a[],int beg,int end)
{
 int mid;
 if(beg<end)
 {
  mid=(beg+end)/2;
  mergesort(a, beg, mid);
  mergesort(a, mid + 1, end);
  merge(a,beg,mid,end);
 }
}
void merge(int a[],int beg,int mid,int end)
{
 int i=beg,j = mid+1,index=beg,temp[20],k;
 while((i<=mid)&&(j<=end))
            {
  if(a[i]<a[j])
  {
   temp[index]=a[i];
   i++;
  }
  else
  {
   temp[index]=a[j];
   j++;
  }
```

```
 index++;
          }
if(i>mid)
{
 while(j<=end)
 {
  temp[index]=a[j];
  j++;
  index++;
 }
}
else
{
 while(i<=mid)
 {
  temp[index]=a[i];
  i++;
  index++;
 }
}
for(k=beg;k<index;k++)
a[k]=temp[k];
}
```

Comparison of Sorting Algorithms

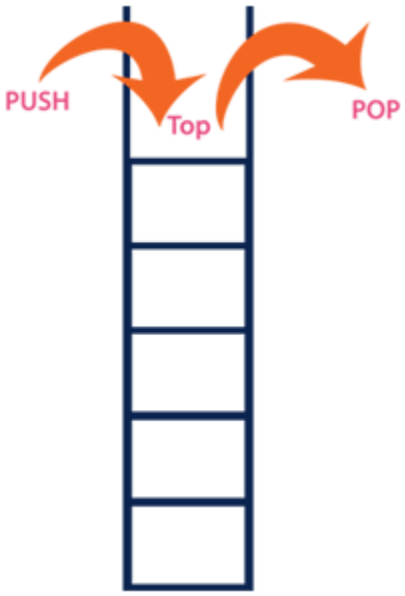
Algorithm	**Data Structure**	**Time Complexity**		
		Best	**Average**	**Worst**
Quick sort	Array	O(n log(n))	O(n log(n))	O(n^2)
Merge sort	Array	O(n log(n))	O(n log(n))	O(n log(n))
Bubble Sort	Array	O(n)	O(n^2)	O(n^2)
Insertion Sort	Array	O(n)	O(n^2)	O(n^2)
Select Sort	Array	O(n^2)	O(n^2)	O(n^2)

UNIT-II

Stack Data structure:

What is a Stack?

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at single position which is known as "**top**". That means, new element is added at top of the stack and an element is removed from the top of the stack. In stack, the insertion and deletion operations are performed based on **LIFO** (Last In First Out) principle.



In a stack, the insertion operation is performed using a function called **"push"** and deletion operation is performed using a function called **"pop"**.

In the figure, PUSH and POP operations are performed at top position in the stack. That means, both the insertion and deletion operations are performed at one end (i.e., at Top)

A stack data structure can be defined as follows...

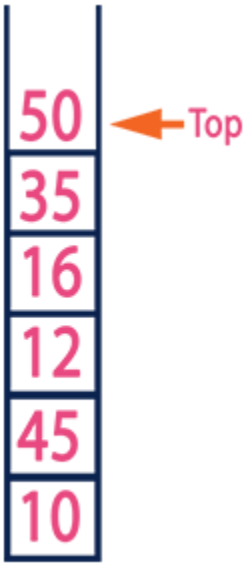
Stack is a linear data structure in which the operations are performed based on LIFO principle.

Stack can also be defined as

"A Collection of similar data items in which both insertion and deletion operations are performed based on LIFO principle".

Example

If we want to create a stack by inserting 10,45,12,16,35 and 50. Then 10 becomes the bottom most element and 50 is the top most element. The last inserted element 50 is at Top of the stack as shown in the image below...



Operations on a Stack

The following operations are performed on the stack...

1. **Push (To insert an element on to the stack)**
2. **Pop (To delete an element from the stack)**
3. **Display (To display elements of the stack)**

Stack data structure can be implemented in two ways. They are as follows...

1. **Using Array**
2. **Using Linked List**

When stack is implemented using array, that stack can organize only limited number of elements. When stack is implemented using linked list, that stack can organize unlimited number of elements.

Stack Using Array

A stack data structure can be implemented using one dimensional array. But stack implemented using array stores only fixed number of data values. This implementation is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable called **'top'**. Initially top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

Stack Operations using Array

A stack can be implemented using array as follows...

Before implementing actual operations, first follow the below steps to create an empty stack.

- **Step1 -** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2 -** Declare all the **functions** used in stack implementation.
- **Step 3 -** Create a one dimensional array with fixed size (**int stack[SIZE]**)
- **Step 4 -** Define a integer variable **'top'** and initialize with **'-1'**. (**int top = -1**)
- **Step 5 -** In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

Push (value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at **top** position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

- **Step 1 -** Check whether **stack** is **FULL**. (**top == SIZE-1**)
- **Step 2 -** If it is **FULL**, then display **"Stack is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT FULL**, then increment **top** value by one (**top++**) and set stack[top] to value (**stack[top] = value**).

Pop () - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

- **Step 1 -** Check whether **stack** is **EMPTY**. (**top = = -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top- -**).

Display () - Displays the elements of a Stack

We can use the following steps to display the elements of a stack...

- **Step 1 - Check whether** stack **is** EMPTY**. (**top == -1**)**
- **Step 2 - If it is** EMPTY**, then display** "Stack is EMPTY!!!" **and terminate the function.**
- **Step 3 -** If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top. Display **stack[i]** value and decrement **i** value by one (**i--**).
- **Step 3 -** Repeat above step until **i** value becomes '0'.

Expressions

What is an Expression?

In any programming language, if we want to perform any calculation or to frame a condition etc., we use a set of symbols to perform the task. These set of symbols makes an expression.

An expression can be defined as follows...

An expression is a collection of operators and operands that represents a specific value.

In above definition, **operator** is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,

Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types

Based on the operator position, expressions are divided into THREE types. They are as follows...

1. **Infix Expression**
2. **Postfix Expression**
3. **Prefix Expression**

Infix Expression

In infix expression, operator is used in between the operands.

The general structure of an Infix expression is as follows...

Operand1 Operator Operand2

Example

Postfix Expression

In postfix expression, operator is used after operands. We can say that "**Operator follows the Operands**".

The general structure of Postfix expression is as follows...

Operand1 Operand2 Operator

Example

Infix to Postfix Conversion using Stack Data Structure

To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps...

1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is operand, then directly print it to the result (Output).
3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
4. If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is popped and print each popped symbol to the result.
5. If the reading symbol is operator (+, - , *, / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

Example

Consider the following Infix Expression...

Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where ^ is an exponential operator.

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack, that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack, that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Resultant Postfix Expression: ABC*DEF^/G*-H*+

Advantage of Postfix Expression over Infix Expression

An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence).Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

Postfix Expression Evaluation

A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands.

Postfix Expression has following general structure...

Operand1 Operand2 Operator

Example

Postfix Expression Evaluation using Stack Data Structure

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped oparands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.

Infix Expression (5 + 3) * (8 - 2)

Postfix Expression 5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations		Evaluated Part of Expression
Initially	Stack is Empty		Nothing
5	push(5)		Nothing
3	push(3)		Nothing

+	value1 = pop()
	value2 = pop()
	result = value2 + value1
	push(result)
			value1 = pop(); // 3
			value2 = pop(); // 5
			result = 5 + 3; // 8
			Push(8)
			(5 + 3)

8	push(8)		(5 + 3)
2	push(2)		(5 + 3)

-	value1 = pop()
	value2 = pop()
	result = value2 - value1
	push(result)
			value1 = pop(); // 2
			value2 = pop(); // 8
			result = 8 - 2; // 6
			Push(6)
			(8 - 2)
			(5 + 3) , (8 - 2)

*	value1 = pop()
	value2 = pop()
	result = value2 * value1
	push(result)
			value1 = pop(); // 6
			value2 = pop(); // 8
			result = 8 * 6; // 48
			Push(48)
			(6 * 8)
			(5 + 3) * (8 - 2)

$ End of Expression	result = pop()		Display (result) 48 As final result

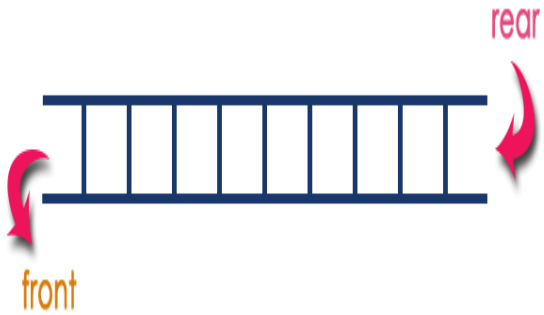
Infix Expression (5 + 3) * (8 - 2) = 48

Postfix Expression 5 3 + 8 2 - * value is 48

Queue

What is a Queue?

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. In a queue data structure, adding and removing of elements are performed at two different positions. The insertion is performed at one end and deletion is performed at other end. In a queue data structure, the insertion operation is performed at a position which is known as '**rear**' and the deletion operation is performed at a position which is known as '**front**'. In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.



In a queue data structure, the insertion operation is performed using a function called "**enQueue()**" and deletion operation is performed using a function called "**deQueue()**".

Queue data structure can be defined as follows...

Queue data structure is a linear data structure in which the operations are performed based on FIFO principle.

A queue data structure can also be defined as

"Queue data structure is a collection of similar data items in which insertion and deletion operations are performed based on FIFO principle".

Example

Queue after inserting 25, 30, 51, 60 and 85.

Operations on a Queue

The following operations are performed on a queue data structure...

1. **enQueue(value) - (To insert an element into the queue)**
2. **deQueue() - (To delete an element from the queue)**
3. **display() - (To display the elements of the queue)**

Queue data structure can be implemented in two ways. They are as follows...

1. **Using Array**
2. **Using Linked List**

When a queue is implemented using array, that queue can organize only limited number of elements. When a queue is implemented using linked list, that queue can organize unlimited number of elements.

Queue Data structure Using Array

A queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values. The implementation of queue data structure using array is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle**with the

help of variables '**front**' and '**rear**'. Initially both '**front**' and '**rear**' are set to -1. Whenever, we want to insert a new value into the queue, increment '**rear**' value by one and then insert at that position. Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

Queue Operations using Array

Queue data structure using array can be implemented as follows...

Before we implement actual operations, first follow the below steps to create an empty queue.

- **Step 1 -** Include all the **header files** which are used in the program and define a constant '**SIZE**' with specific value.
- **Step 2 -** Declare all the **user defined functions** which are used in queue implementation.
- **Step 3 -** Create a one dimensional array with above defined SIZE (**int queue[SIZE]**)
- **Step 4 -** Define two integer variables '**front**' and '**rear**' and initialize both with '**-1**'. (**int front = -1, rear = -1**)
- **Step 5 -** Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

enqueue (value) - Inserting value into the queue

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

- **Step 1 -** Check whether **queue** is **FULL**. (**rear == SIZE-1**)
- **Step 2 -** If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear]** = **value**.

deQueue() - Deleting a value from the Queue

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from **front** position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as deleted element. Then check whether both **front** and **rear** are equal (**front** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**'(**front** = **rear** = **-1**).

Display () - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

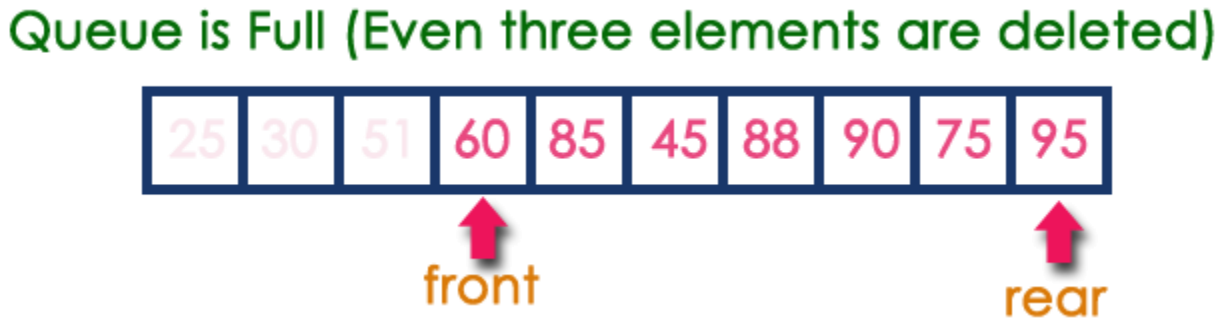
- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front+1**'.
- **Step 4 -** Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**)

Circular Queue Data structure

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once queue becomes full, we cannot insert the next element until all the elements are deleted from the queue. For example consider the queue below...

The queue after inserting all the elements into it is as follows...

Now consider the following situation after deleting three elements from the queue...



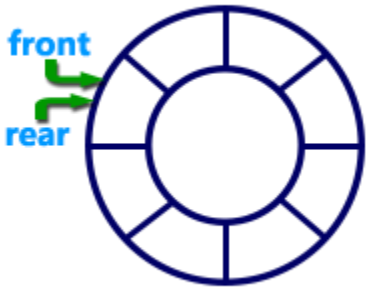
This situation also says that Queue is Full and we cannot insert the new element because, '**rear**' is still at last position. In above situation, even though we have empty positions in the queue we can not make use of them to insert new element. This is the major problem in normal queue data structure. To overcome this problem we use circular queue data structure.

What is Circular Queue?

A Circular Queue can be defined as follows...

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.

Graphical representation of a circular queue is as follows...



Priority Queue

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

Applications of Queue Data Structure

Queue is used when things don't have to be processed immediately, but have to be processed in First In First Out order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.
1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.
See this for more detailed applications of Queue and Stack.
References:

UNIT -III

Single Linked List

The disadvantages of arrays are:

• The size of the array is fixed. Most often this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required.

•Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.

• Deleting an element from an array is not possible. Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. Generally array's allocates the memory for all its elements in one block whereas linked lists use

• An entirely different strategy. Linked lists allocate memory for each element separately and only when necessary.

Advantages of linked lists:

Linked lists have many advantages. Some of the very important advantages are:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.

2. Linked lists have efficient memory utilization. Here, memory is not preallocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.

3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.

4. Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node.

2. Searching a particular element in list is difficult and also time consuming.

Types of Linked Lists:

Basically we can put linked lists into the following four items:

1. Single Linked List.

2. Double Linked List.

3. Circular Linked List.

4. Circular Double Linked List.

Comparison between array and linked list:

ARRAY	LINKED LIST
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

Trade offs between linked lists and arrays:

FEATURE	ARRAYS	LINKED LISTS
Sequential access	efficient	efficient
Random access	efficient	inefficient
Resigning	inefficient	efficient
Element rearranging	inefficient	efficient
Overhead per elements	none	1 or 2 links

What is Linked List?

When we want to work with unknown number of data values, we use a linked list data structure to organize that data. Linked list is a linear data structure that contains sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called as "Node".

What is Single Linked List?

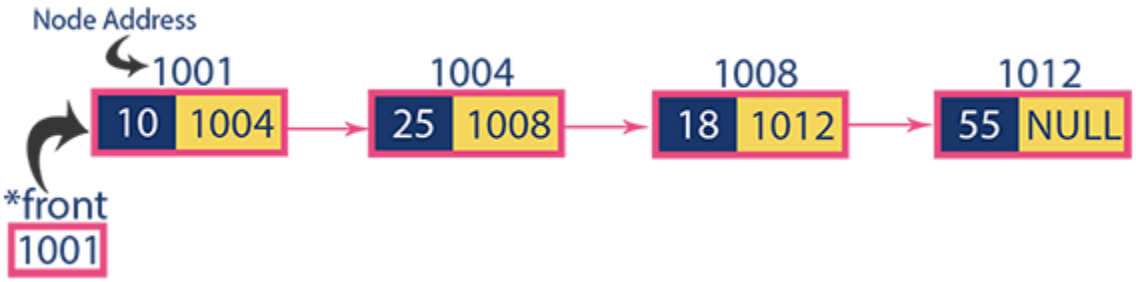
Simply a list is a sequence of data, and linked list is a sequence of data linked with each other. The formal definition of a single linked list is as follows...

Single linked list is a sequence of elements in which every element has link to its next element in the sequence.

In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data field and next field. The data field is used to store actual value of the node and next field is used to store the address of next node in the sequence.

The graphical representation of node in a single linked list is as follows...

Example



Operations on Single Linked List

The following operations are performed on a Single Linked List

- Insertion
- Deletion
- Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program.
- **Step 2 -** Declare all the **user defined functions**.
- **Step 3 -** Define a **Node** structure with two members **data** and **next**
- **Step 4 -** Define a Node pointer **'start'** and set it to **NULL**.
- **Step 5 -** Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

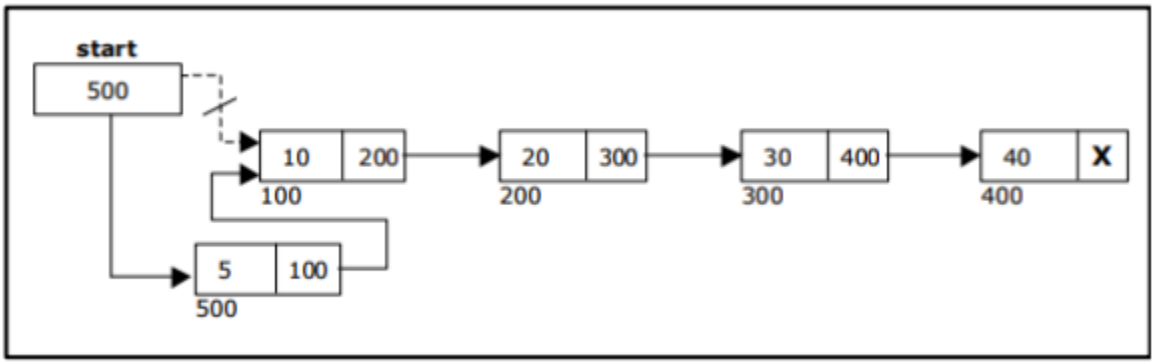
In a single linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

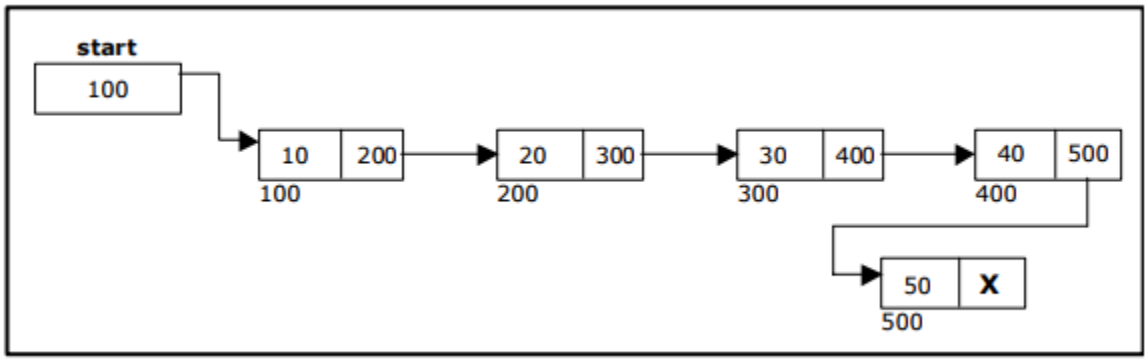
- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**start** = = **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode→next** = **NULL** and **start** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, set **newNode→next** = **start** and **start**= **newNode**.



Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode** → **next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**start** = = **NULL**).
- **Step 3 -** If it is **Empty** then, set **start** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **start**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** is not equal to **NULL**).
- **Step 6 -** Set **temp** → **next** = **newNode**.



Inserting at middle of the list

Step 1 - Create a newNode with given value

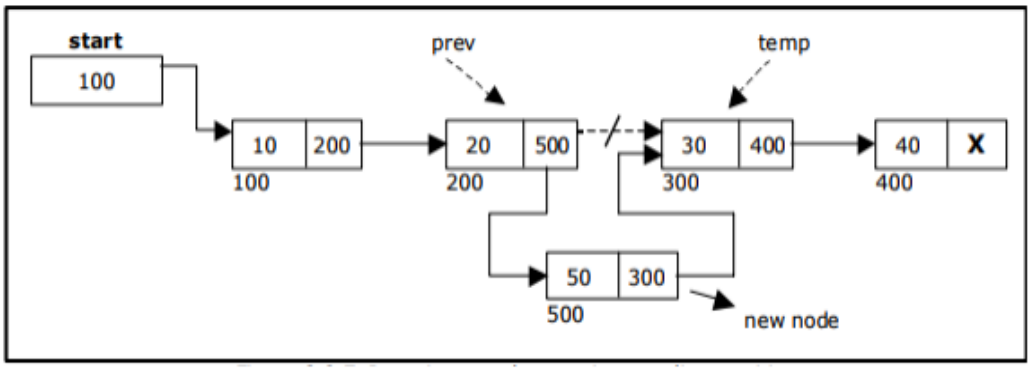
Step 2 - Check whether list is **Empty** (**start** == **NULL**)

Step 3 - If it is **Empty** then, set **newNode** → **next** = **NULL** and **start** = **newNode**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **start**.

Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode

Step 6-Finally, Set p->next=newnode, newnode->next=temp



Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

Step 1 - Check whether list is **Empty** (**start** == **NULL**)

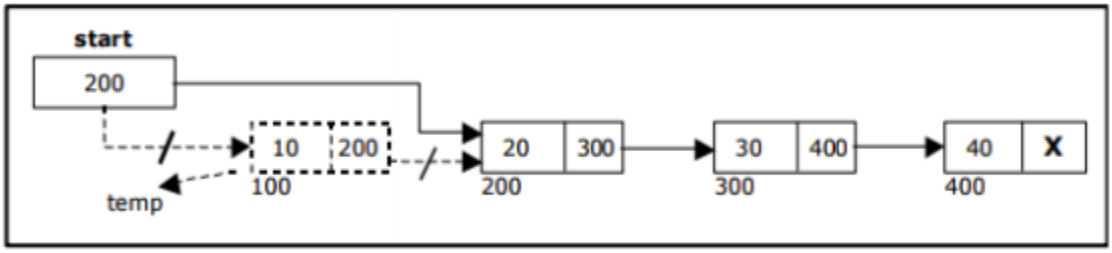
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **start**.

Step 4 - Check whether list is having only one node (**temp → next** == **NULL**)

Step 5 - If it is **TRUE** then set **start**= **NULL** and delete **temp** (Setting **Empty** list conditions)

Step 6 - If it is **FALSE** then set **start**= **temp → next**, and delete **temp(free(temp))**.



Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

Step 1 - Check whether list is **Empty** (**start** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

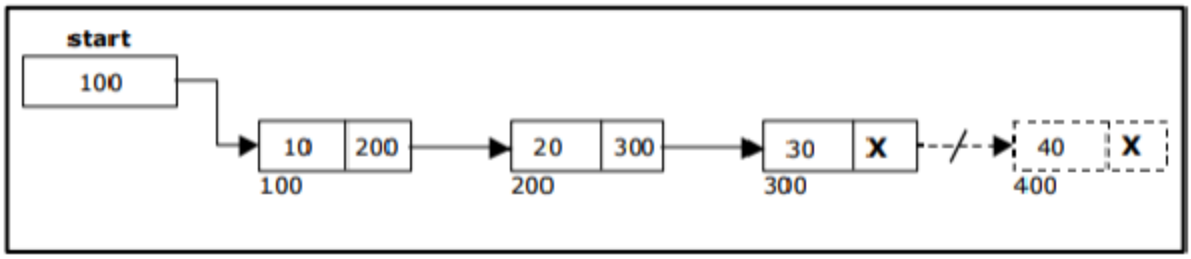
Step 3 - If it is **Not Empty** then, define two Node pointers **'temp'** and **'temp1'** and initialize **'temp'** with **start**.

Step 4 - Check whether list has only one Node (**temp → next** == **NULL**)

Step 5 - If it is **TRUE**. Then, set **start** = **NULL** and delete **temp**. And terminate the function. (Setting **Empty** list condition)

Step 6 - If it is **FALSE**. Then, set '**temp1 = temp** ' and move **temp** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1 → next** == **NULL**)

Step 7 - Finally, Set **temp1 → next** = **NULL** and delete **temp(free(temp))**.



Deleting a Specific Node from the list

Step 1 - Create a newNode with given value

Step 2 - Check whether list is **Empty** (**start** == **NULL**)

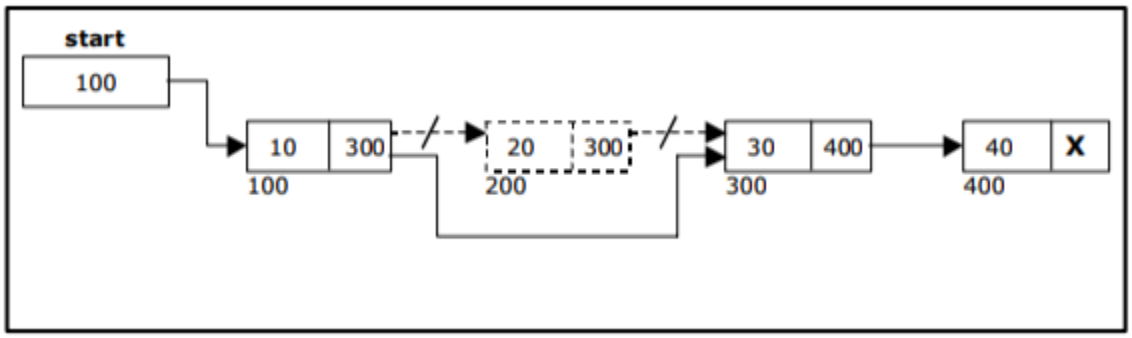
Step 3 - If it is **Empty** then, set **newNode** → **next** = **NULL** and **start** = **newNode**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **start**.

Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to delete the Node

Step 6-Finally, Set p->next=temp->next

Step 7-delete temp(free(temp)).



Traversing

- Assign the address of start pointer to a temp pointer.
- Display the information from the data field of each node.
- The function traverse () is used for traversing and displaying the information stored in the list from left to right.

Applications of Linked Lists

Polynomials and Sparse Matrix are two important applications of arrays and linked lists.

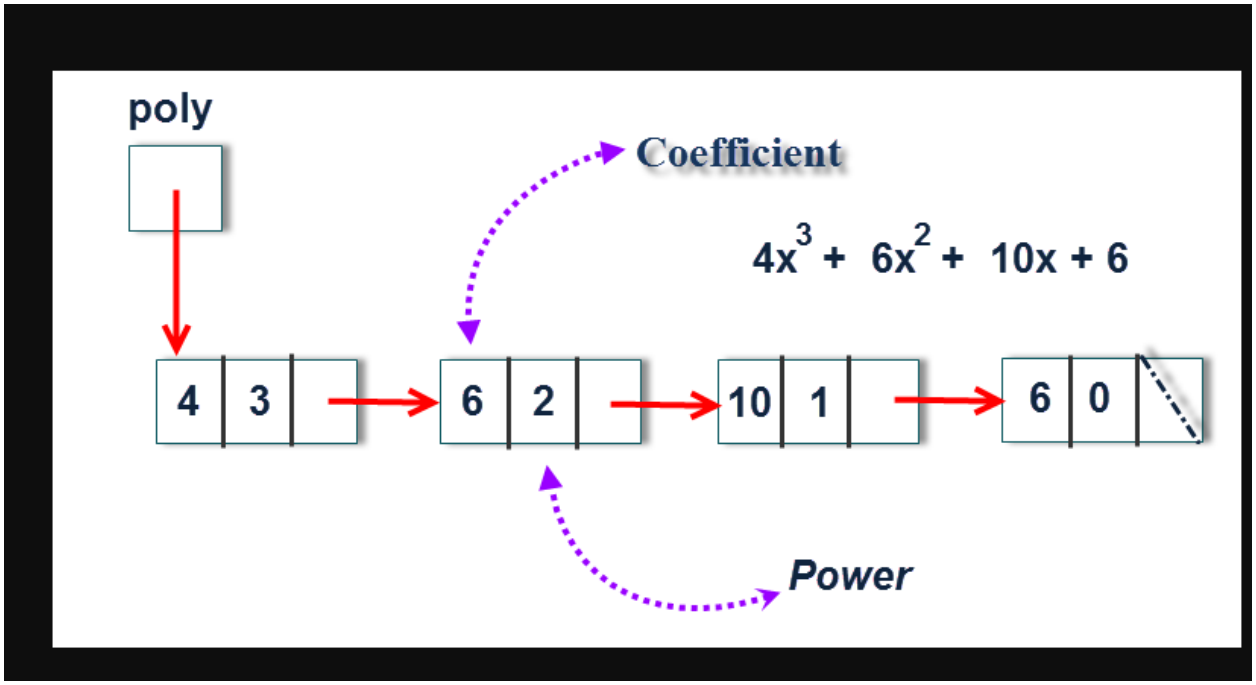
A polynomial is composed of different terms where each of them holds a coefficient and an exponent.

Polynomial Representation:

A polynomial may also be represented using a linked list. A structure may be defined such that it contains two parts- one is the coefficient and second is the corresponding exponent. The structure definition may be given as shown below:

```
struct polynomial
{
int coefficient;
int exponent;
struct polynomial *next;
};
```

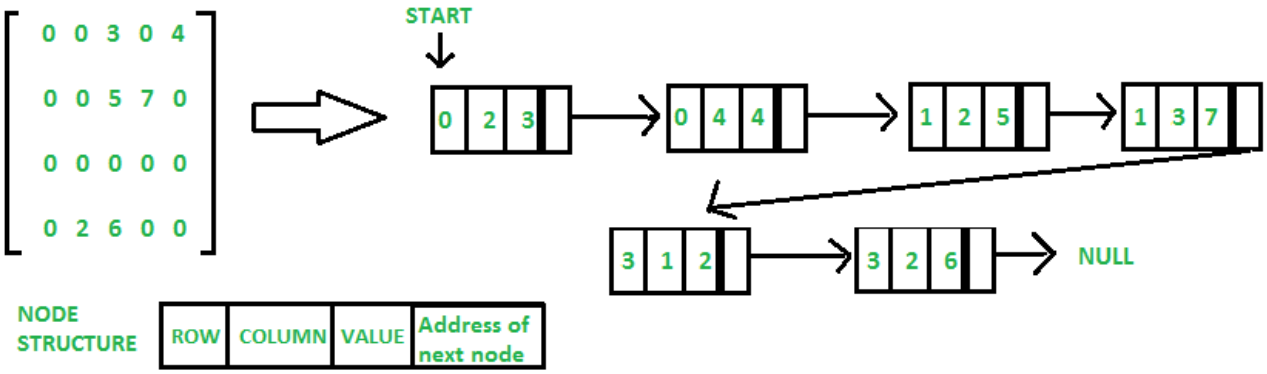
Thus the above polynomial may be represented using linked list as shown below:



Sparse Matrix Representation

In linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row, column)
- **Next node:** Address of the next node



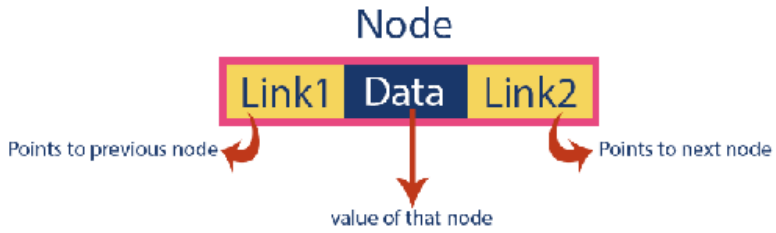
Double Linked List

What is Double Linked List?

In a single linked list, every node has link to its next node in the sequence. So, we can traverse from one node to other node only in one direction and we cannot traverse back. We can solve this kind of problem by using double linked list. Double linked list can be defined as follows...

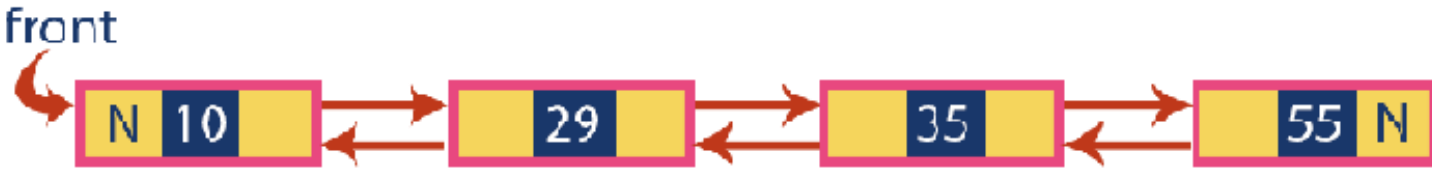
Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.

In double linked list, every node has link to its previous node and next node. So, we can traverse forward by using next field and can traverse backward by using previous field. Every node in a double linked list contains three fields and they are shown in the following figure...



Here, '**link1**' field is used to store the address of the previous node in the sequence, '**link2**' field is used to store the address of the next node in the sequence and '**data**' field is used to store the actual value of that node.

Example



Operations on Double Linked List

In a double linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Insertion

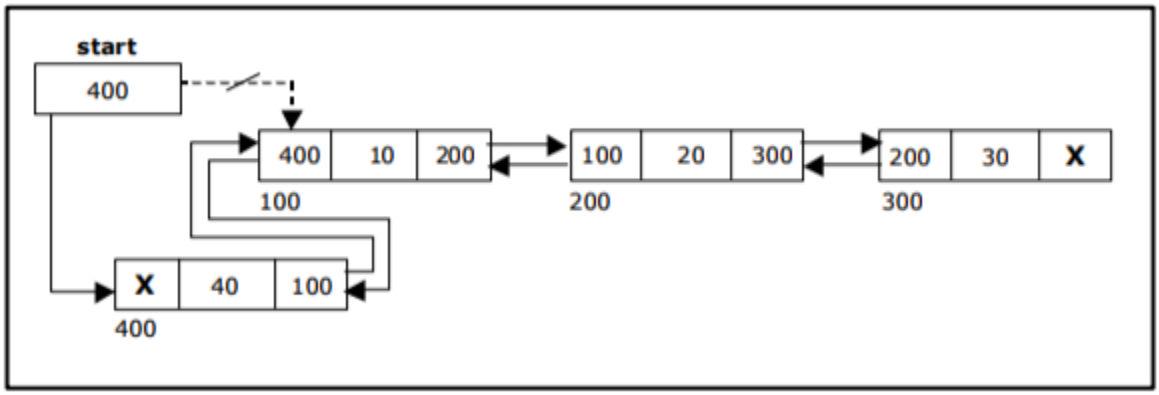
In a double linked list, the insertion operation can be performed in three ways as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

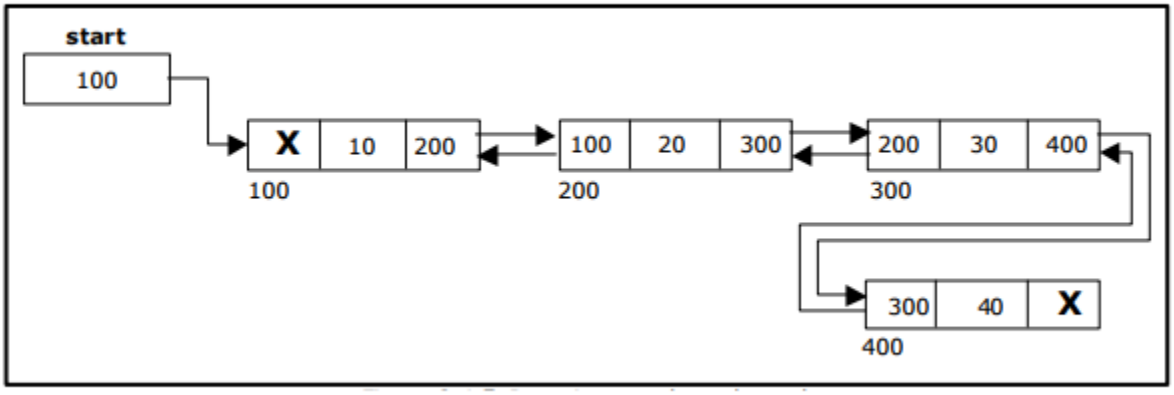
- **Step 1 -** Create a **newNode** with given value and **newNode → previous** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**start** = = **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **start**.
- **Step 4 -** If it is **not Empty** then, assign **start** to **newNode → next** and **newNode** to **start**.



Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode → next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**start** = = **NULL**)
- **Step3-** If it is **Empty**, then assign **NULL** to **newNode→previous** and **newNode** to **head**.
- **Step 4 -** If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
- **Step 6 -** Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.



Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to both **newNode → previous**& **newNode → next** and set **newNode** to **head**.
- **Step 4 -** If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.
- **Step 5 -** Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.
- **Step 7 -** Assign **temp1 → next** to **temp2**, **newNode** to **temp1 → next**, **temp1** to **newNode → previous**, **temp2** to **newNode → next** and **newNode** to **temp2 → previous**.

UNIT-IV

Tree Terminology

In linear data structure, data is organized in sequential order and in non-linear data structure, data is organized in random order. Tree is a very popular data structure used in wide range of applications. A tree data structure can be defined as follows...

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

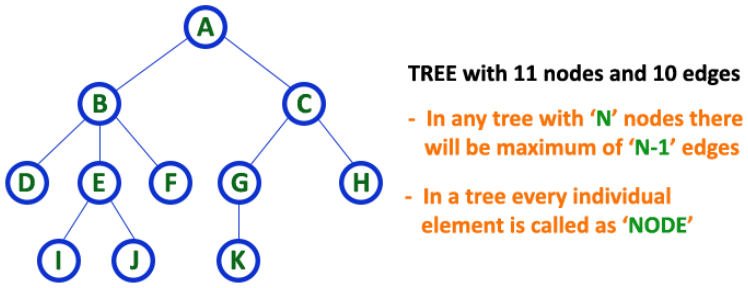
A tree data structure can also be defined as follows...

Tree data structure is a collection of data (Node) which is organized in hierarchical structure and this is a recursive definition

In tree data structure, every individual element is called as **Node**. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure.

In a tree data structure, if we have **N** number of nodes then we can have a maximum of **N-1** number of links.

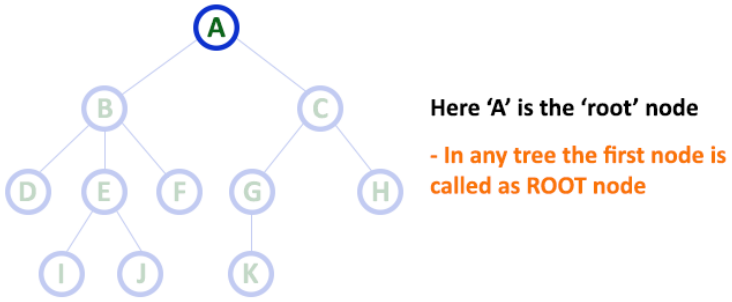
Example



In a tree data structure, we use the following terminology...

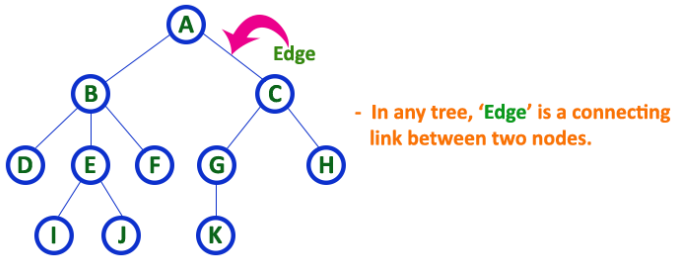
1. Root

In a tree data structure, the first node is called as **Root Node**. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.



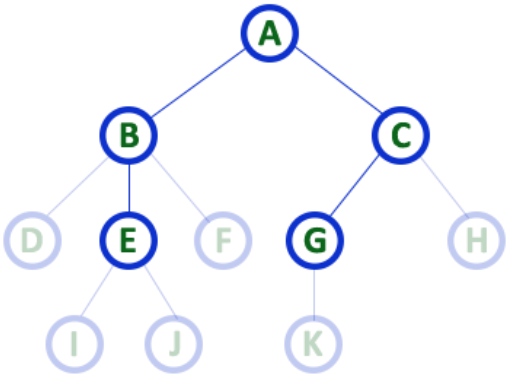
2. Edge

In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



3. Parent

In a tree data structure, the node which is predecessor of any node is called as **PARENT NODE**. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "**The node which has child / children**".

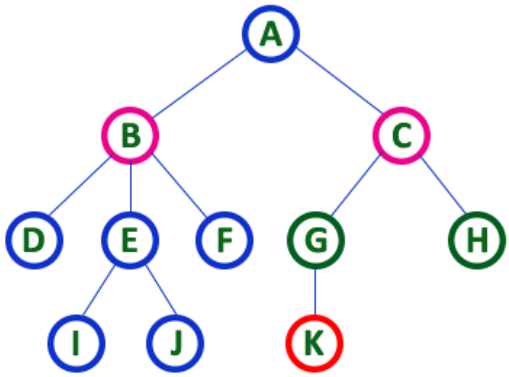


Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

4. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



Here B & C are Children of A

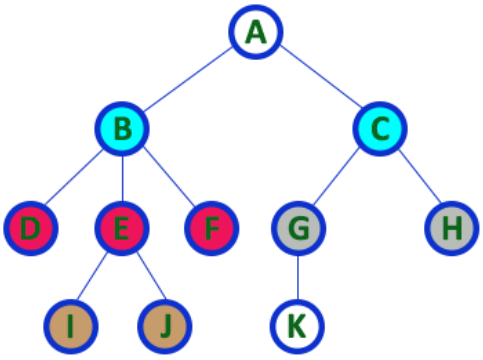
Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as Sibling nodes.



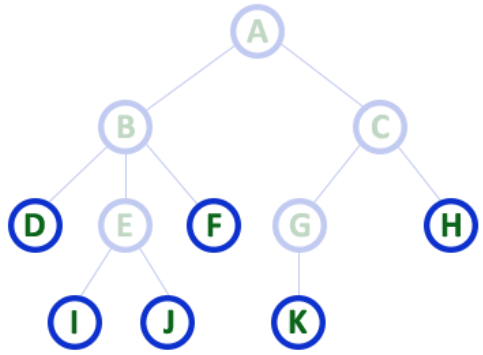
Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

6. Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.



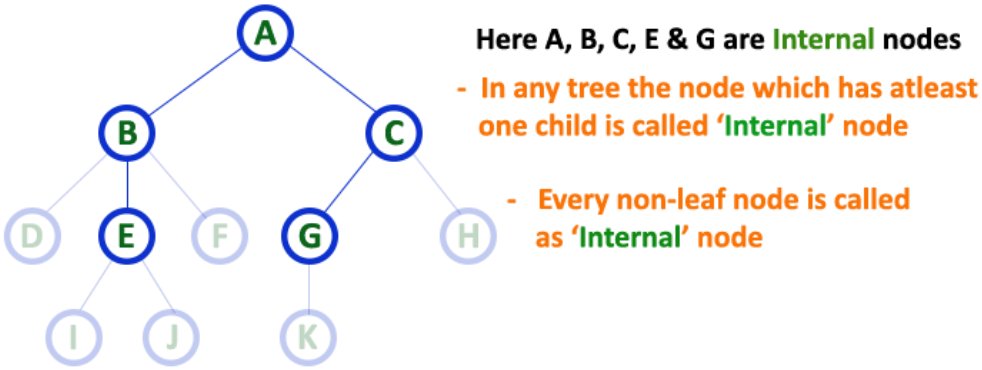
Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

7. Internal Nodes

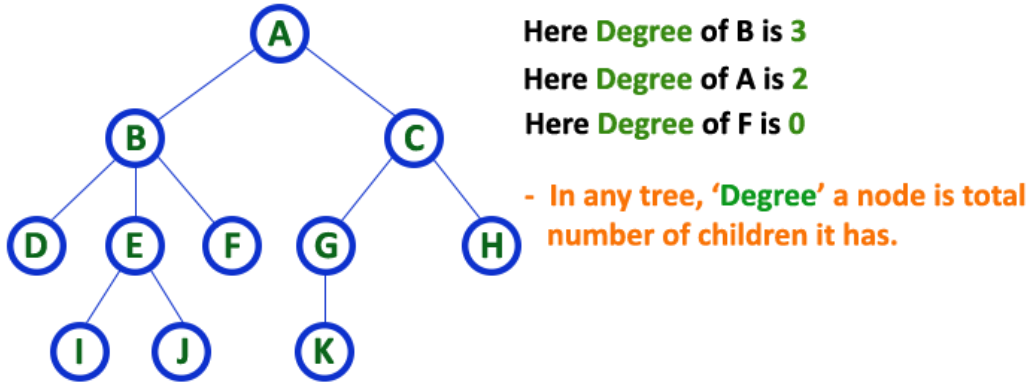
In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes.



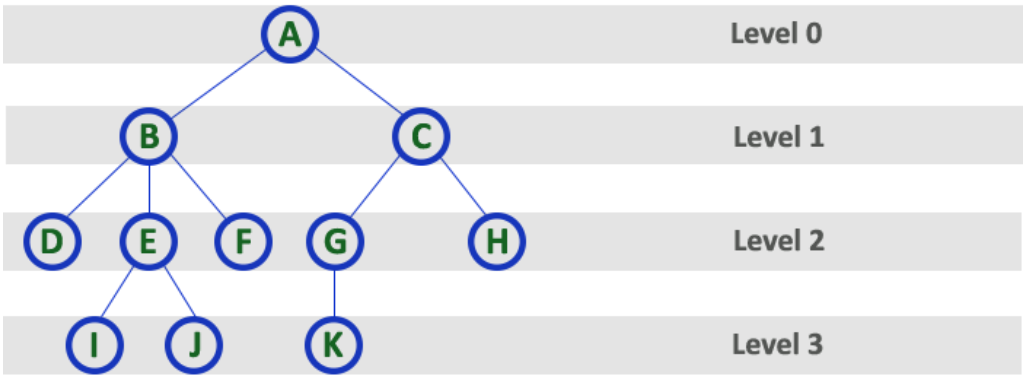
8. Degree

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'



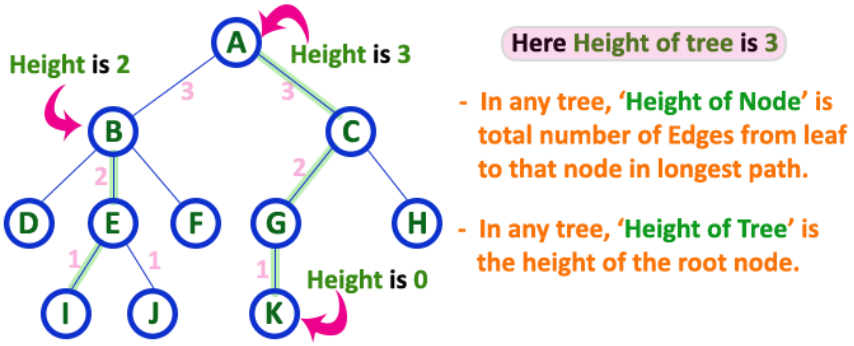
9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



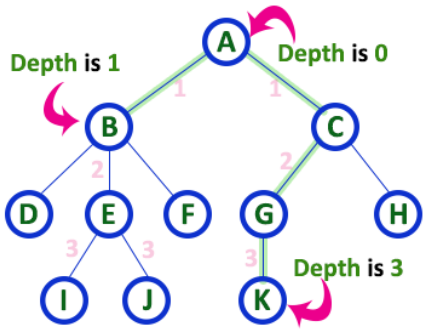
10. Height

In a tree data structure, the total number of egdes from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**. In a tree, **height of all leaf nodes is '0'.**



11. Depth

In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**

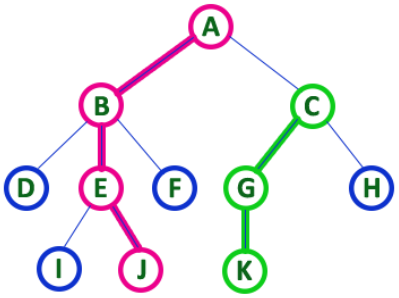


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4.**



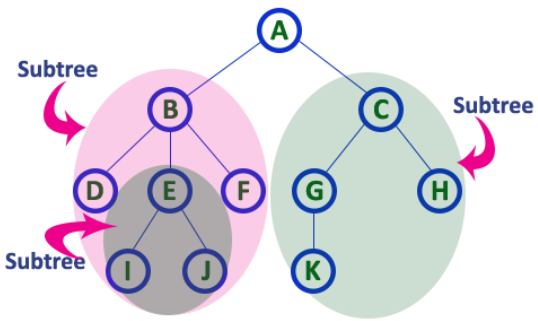
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K

13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



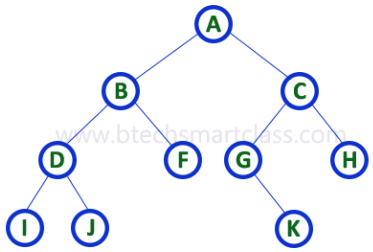
Binary Tree

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**. One is known as left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called as Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

Example



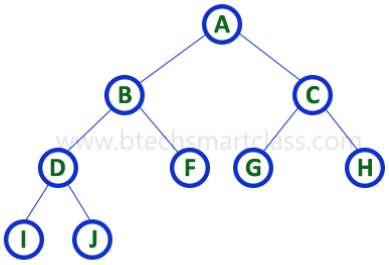
There are different types of binary trees and they are...

1. Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...

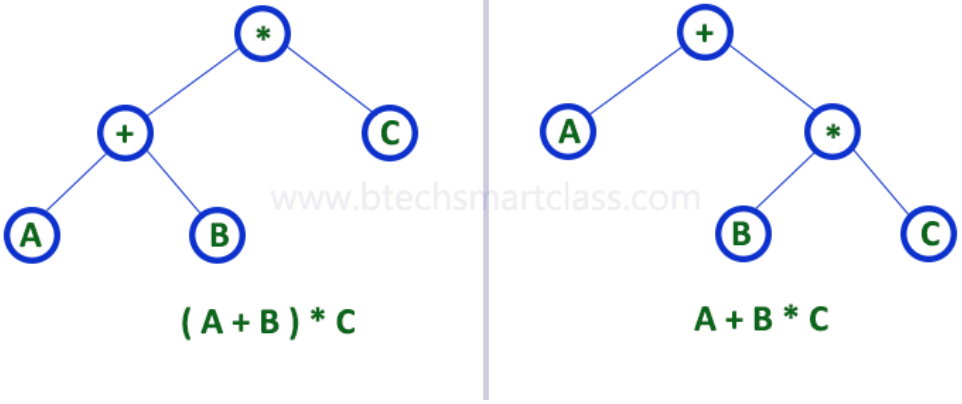
A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree

Strictly binary tree is also called as **Full Binary Tree** or **Proper Binary Tree** or **2-Tree**



Strictly binary tree data structure is used to represent mathematical expressions.

Example

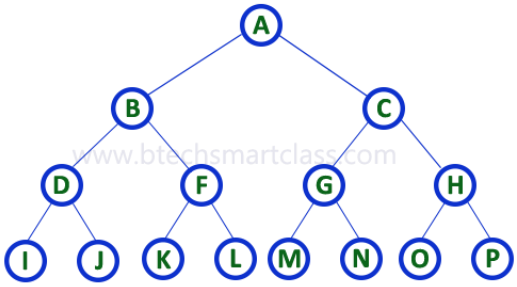


2. Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2^{level} number of nodes. For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes.

A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.

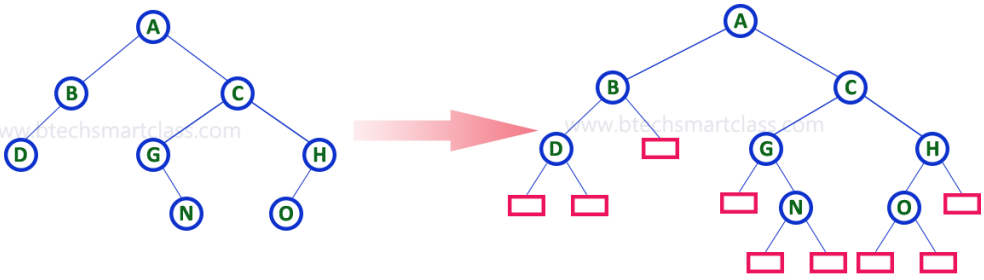
Complete binary tree is also called as **Perfect Binary Tree**



3. Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.



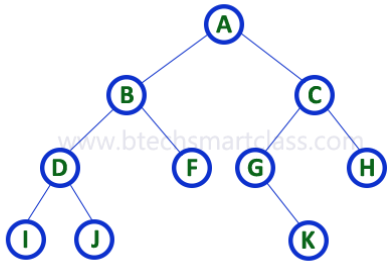
In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes .

Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation**
2. **Linked List Representation**

Consider the following binary tree...



1. Array Representation

In array representation of binary tree, we use a one dimensional array (1-D Array)to representation binarytree. Consider the above example of binary tree and it is represented as follows...

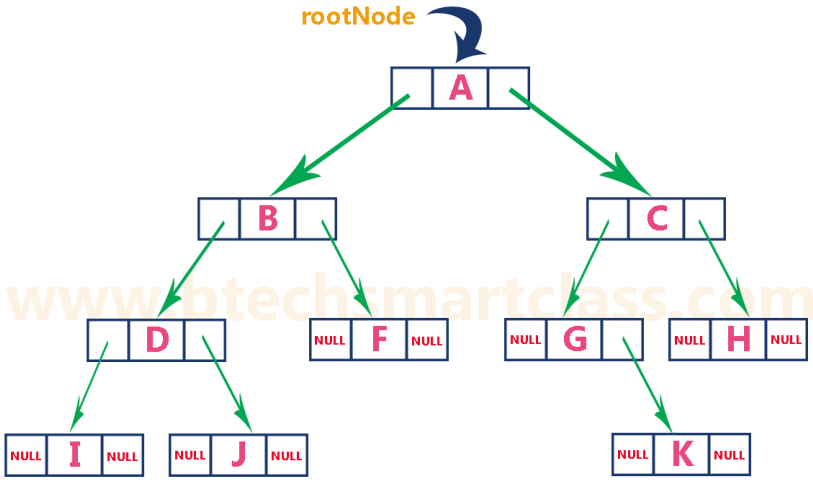
A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

To represent a binary tree of depth '**n**' using array representation, we need one dimensional array with a maximum size of $2^{n+1} - 1$.

2. Linked List Representation

We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address. In this linked list representation, a node has the following structure...

The above example of binary tree represented using Linked list representation is shown as follows...



Binary Tree Traversals

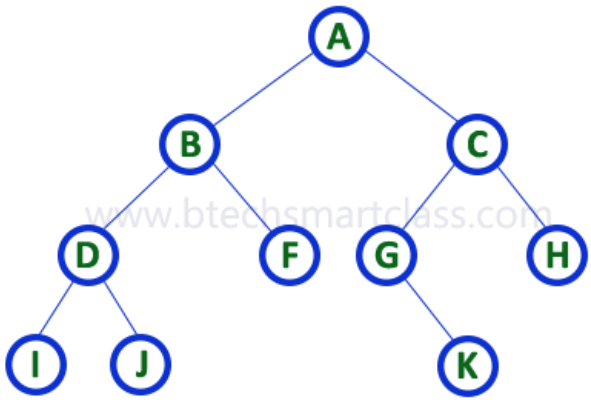
When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

There are three types of binary tree traversals.

1. **In - Order Traversal**
2. **Pre - Order Traversal**
3. **Post - Order Traversal**

Consider the following binary tree...



1. In - Order Traversal (leftChild - root - rightChild)

In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

In the above example of binary tree, first we try to visit left child of root

node 'A', but A's left child is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J. So we try to visit its left child 'I' and it is the left most child. So first we visit '**I**' then go for its root node '**D**' and later we visit D's right child '**J**'. With this we have completed the left part of node B. Then visit '**B**' and next B's right child '**F**' is visited. With this we have completed left part of node A. Then visit root node '**A**'. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit '**G**' and then visit G's right child K. With this we have completed the left part of node C. Then visit root node '**C**' and next visit C's right child '**H**' which is the right most child in the tree so we stop the process.

That means here we have visited in the order of **I - D - J - B - F - A - G - K - C - H** using In-Order Traversal.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C - H

2. Pre - Order Traversal (root - leftChild - rightChild)

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.

In the above example of binary tree, first we visit root node '**A**' then visit its left child '**B**' which is a root for D and F. So we visit B's left child '**D**' and again D is a root for I and J. So we visit D's left child '**I**'which is the left most child. So next we go for visiting D's right child '**J**'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child '**F**'. With this we have completed root and left parts of node A. So we go for A's right child '**C**' which is a root node for G and H. After visiting C, we go for its left child '**G**' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child '**K**'. With this we have completed node C's root and left parts. Next visit C's right child '**H**' which is the right most child in the tree. So we stop the process.

That means here we have visited in the order of **A-B-D-I-J-F-C-G-K-H** using Pre-Order Traversal.

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

2. Post - Order Traversal (leftChild - rightChild - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Here we have visited in the order of **I - J - D - F - B - K - G - H - C - A** using Post-Order Traversal.

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C - A

Tree variants(Types Of Trees):

1.General Tree

2.Binary Tree

3.Binary Search Tree

4.AVL Tree

5.B Tree

Applications of Trees:

- Decision Making
    - Next Move in games
    - Computer chess games build a huge tree (training) which they prune at runtime using heuristics to reach an optimal move.
- Networking
    - Router algorithms -Network Routing, where the next path/route of the packet is determined
    - Social networking is the current buzzword in CS research. It goes without saying that connections/relations are very naturally modeled using graphs. Often, trees are used to represent/identify more interesting phenomena.
- Representation
    - Chemical formulas representation
    - XML/Markup parsers use trees
    - Producers/consumers often use a balanced tree implementation to store a document in memory.
- Manipulate Hierarchical Data
    - Make information easy to search
    - Manipulate sorted lists of data.
- Workflow
    - As a workflow for compositing digital images for visual effects.
- Organizing Things
    - Folders/ files in the Operating system
    - HTML Document Object Model (DOM)
    - Company Organisation Structures
    - PDF is a tree-based format. It has a root node followed by a catalog node followed by a pages node which has several child page nodes.
- Faster Lookup
    - Auto-correct applications and spell checker
    - Syntax Tree in Compiler
- Task Tracker
    - Undo function in a text editor

Binary Search Tree

In a binary tree, every node can have maximum of two children but there is no order of nodes based on their values. In binary tree, the elements are arranged as they arrive to the tree, from top to bottom and left to right.

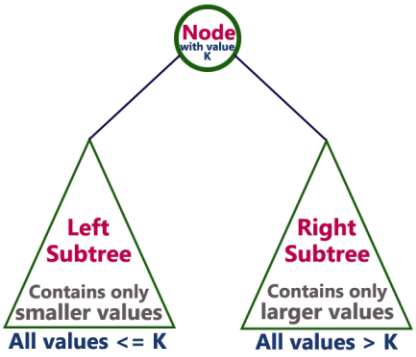
A binary tree has the following time complexities...

1. Search Operation - O(logn)
2. Insertion Operation - O(logn)
3. Deletion Operation - O(logn)

To enhance the performance of binary tree, we use special type of binary tree known as **Binary Search Tree**. Binary search tree mainly focus on the search operation in binary tree. Binary search tree can be defined as follows...

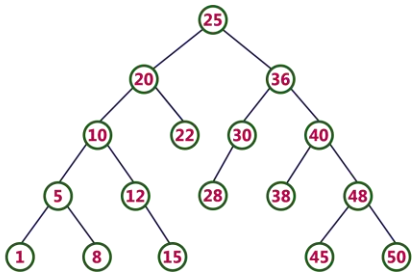
Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

In a binary search tree, all the nodes in left subtree of any node contains smaller values and all the nodes in right subtree of that contains larger values as shown in following figure...



Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



NOTE: Every Binary Search Tree is a binary tree but all the Binary Trees need not to be binary search trees.

The following operations are performed on a binary Search tree...

1. **Search**
2. **Insertion**
3. **Deletion**

Search Operation in BST

In a binary search tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows...

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the value of root node in the tree.
- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, then check whether search element is smaller or larger than that node value.
- **Step 5:** If search element is smaller, then continue the search process in left subtree.
- **Step 6:** If search element is larger, then continue the search process in right subtree.
- **Step 7:** Repeat the same until we found exact element or we completed with a leaf node
- **Step 8:** If we reach to the node with search value, then display "Element is found" and terminate the function.

- **Step 9:** If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in BST

In a binary search tree, the insertion operation is performed with **O(log n)** time complexity. In binary search tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

- **Step 1:** Create a newNode with given value and set its **left** and **right** to **NULL**.
- **Step 2:** Check whether tree is Empty.
- **Step 3:** If the tree is **Empty**, then set set **root** to **newNode**.
- **Step 4:** If the tree is **Not Empty**, then check whether value of newNode is **smaller** or **larger** than the node (here it is root node).
- **Step 5:** If newNode is **smaller** than **or equal** to the node, then move to its **left** child. If newNode is **larger** than the node, then move to its **right** child.
- **Step 6:** Repeat the above step until we reach to a **leaf** node (e.i., reach to NULL).
- **Step 7:** After reaching a leaf node, then isert the newNode as **left child** if newNode is **smaller or equal** to that leaf else insert it as **right child**.

Deletion Operation in BST

In a binary search tree, the deletion operation is performed with **O(log n)** time complexity. Deleting a node from Binary search tree has follwing three cases...

- **Case 1: Deleting a Leaf node (A node with no children)**
- **Case 2: Deleting a node with one child**
- **Case 3: Deleting a node with two children**

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

- **Step 1:** **Find** the node to be deleted using **search operation**
- **Step 2:** Delete the node using **free** function (If it is a leaf) and terminate the function.

Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

- **Step 1:** **Find** the node to be deleted using **search operation**
- **Step 2:** If it has only one child, then create a link between its parent and child nodes.
- **Step 3:** Delete the node using **free** function and terminate the function.

Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

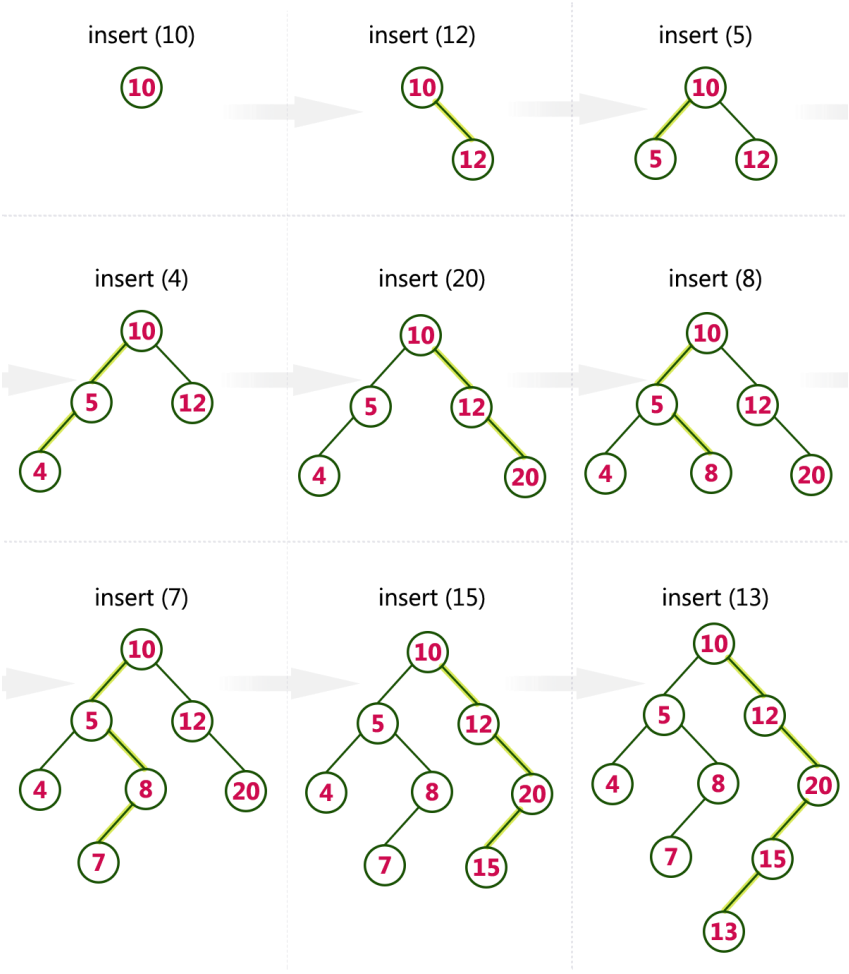
- **Step 1:** **Find** the node to be deleted using **search operation**
- **Step 2:** If it has two children, then find the **largest** node in its **left subtree** (OR) the **smallest** node in its **right subtree**.
- **Step 3:** **Swap** both **deleting node** and node which found in above step.
- **Step 4:** Then, check whether deleting node came to **case 1** or **case 2** else goto steps 2
- **Step 5:** If it comes to **case 1**, then delete using case 1 logic.
- **Step 6:** If it comes to **case 2**, then delete using case 2 logic.
- **Step 7:** Repeat the same process until node is deleted from the tree.

Example

Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13

Above elements are inserted into a Binary Search Tree as follows...



AVL Tree

AVL tree is a self balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced, if the difference between the hieghts of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one. In an AVL tree, every node maintains a extra information known as **balance factor**. The AVL tree was introduced in the year of 1962 by G.M. Adelson-Velsky and E.M. Landis.

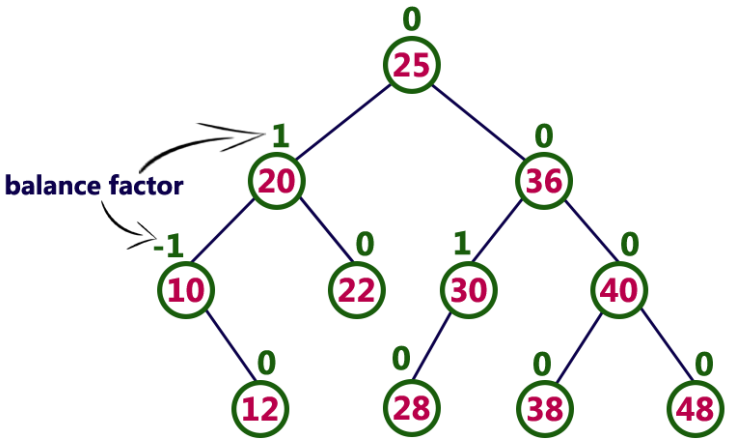
An AVL tree is defined as follows...

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor of a node is the difference between the heights of left and right subtrees of that node. The balance factor of a node is calculated either **height of left subtree - height of right subtree** (OR) **height of right subtree - height of left subtree**. In the following explanation, we are calculating as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

Example



The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

Every AVL Tree is a binary search tree but all the Binary Search Trees need not to be AVL trees.

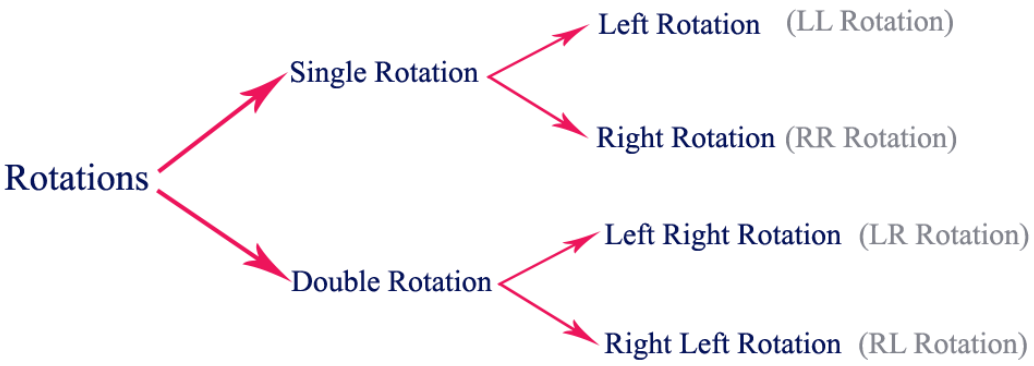
AVL Tree Rotations

In AVL tree, after performing every operation like insertion and deletion we need to check the **balance factor** of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. We use **rotation** operations to make the tree balanced whenever the tree is becoming imbalanced due to any operation.

Rotation operations are used to make a tree balanced.

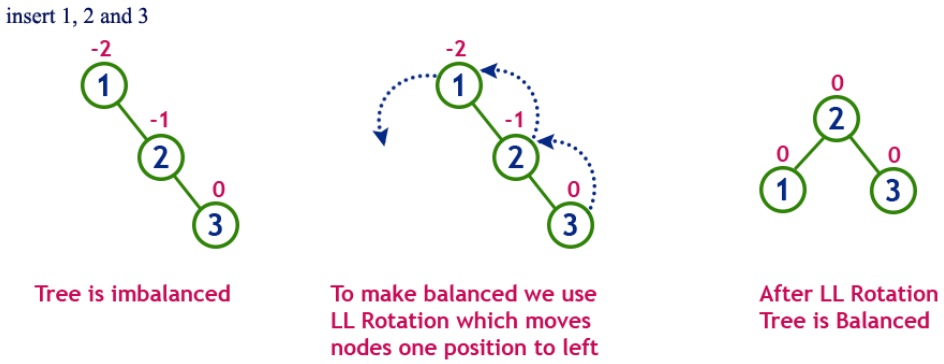
Rotation is the process of moving the nodes to either left or right to make tree balanced.

There are **four** rotations and they are classified into **two** types.



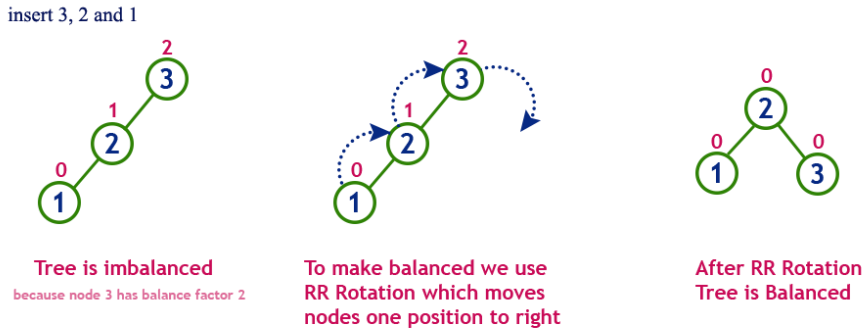
Single Left Rotation (LL Rotation)

In LL Rotation every node moves one position to left from the current position. To understand LL Rotation, let us consider following insertion operations into an AVL Tree...



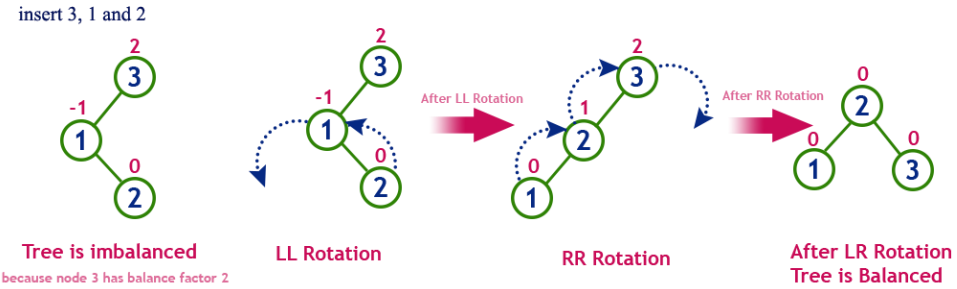
Single Right Rotation (RR Rotation)

In RR Rotation every node moves one position to right from the current position. To understand RR Rotation, let us consider following insertion operations into an AVL Tree...



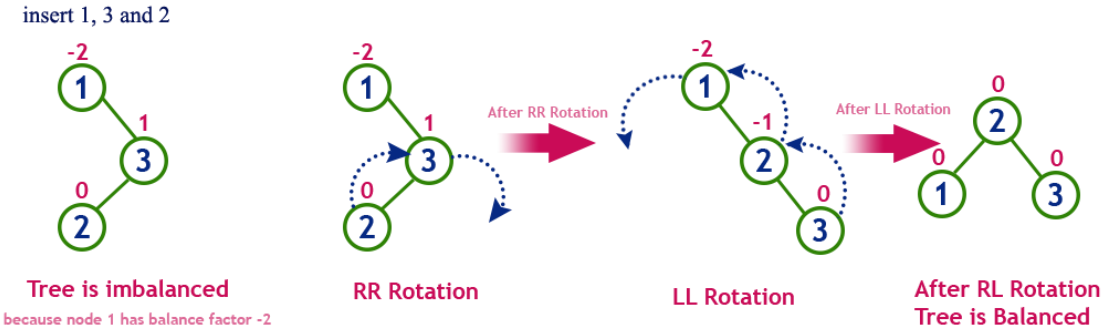
Left Right Rotation (LR Rotation)

The LR Rotation is combination of single left rotation followed by single right rotation. In LR Roration, first every node moves one position to left then one position to right from the current position. To understand LR Rotation, let us consider following insertion operations into an AVL Tree...



Right Left Rotation (RL Rotation)

The RL Rotation is combination of single right rotation followed by single left rotation. In RL Roration, first every node moves one position to right then one position to left from the current position. To understand RL Rotation, let us consider following insertion operations into an AVL Tree...



The following operations are performed on an AVL tree...

1. Search
2. Insertion
3. Deletion

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed similar to Binary search tree

search operation. We use the following steps to search an element in AVL tree...

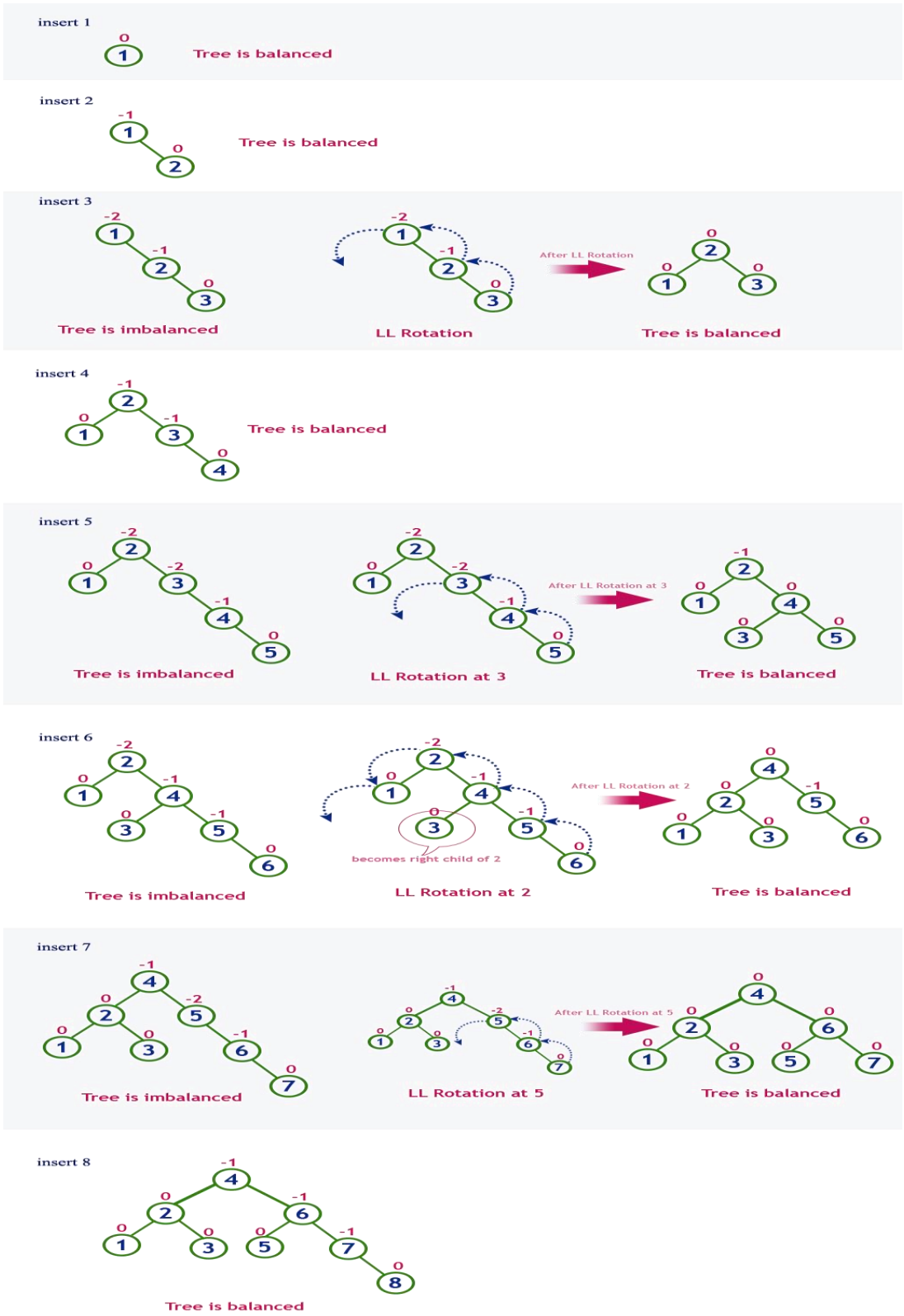
- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the value of root node in the tree.
- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, then check whether search element is smaller or larger than that node value.
- **Step 5:** If search element is smaller, then continue the search process in left subtree.
- **Step 6:** If search element is larger, then continue the search process in right subtree.
- **Step 7:** Repeat the same until we found exact element or we completed with a leaf node
- **Step 8:** If we reach to the node with search value, then display "Element is found" and terminate the function.
- **Step 9:** If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

- **Step 1:** Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step 2:** After insertion, check the **Balance Factor** of every node.
- **Step 3:** If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- **Step 4:** If the **Balance Factor** of any node is other than **0 or 1 or -1** then tree is said to be imbalanced. Then perform the suitable **Rotation** to make it balanced. And go for next operation.

Example: Construct an AVL Tree by inserting numbers from 1 to 8.



Deletion Operation in AVL Tree

In an AVL Tree, the deletion operation is similar to deletion operation in BST. But after every deletion operation we need to check with the Balance Factor condition. If the tree is balanced after deletion then go for next operation otherwise perform the suitable rotation to make the tree Balanced.

B - Trees

In a binary search tree, AVL Tree, Red-Black tree etc., every node can have only one value (key) and maximum of two children but there is another type of search tree called B-Tree in which a node can store more than one value (key) and it can have more than two children. B-Tree was developed in the year of 1972 by **Bayer and McCreight** with the name ***Height Balanced m-way Search Tree***. Later it was named as B-Tree. B-Tree can be defined as follows...

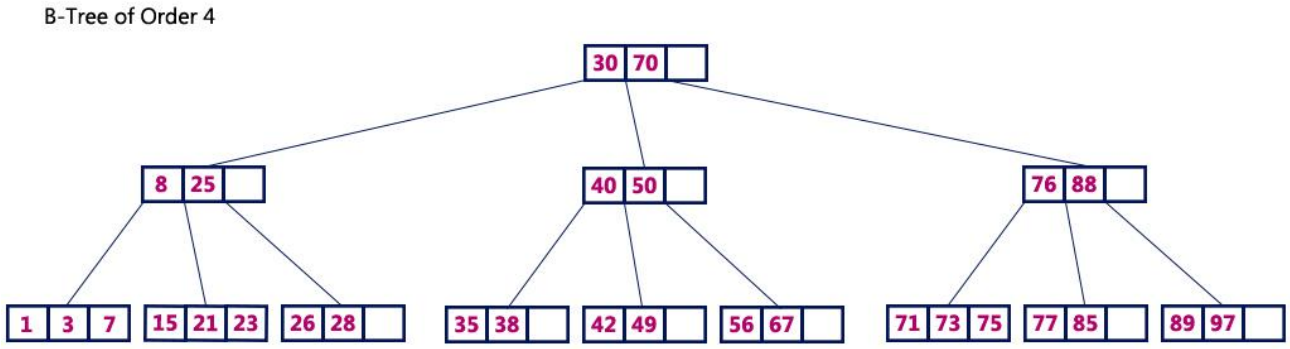
B-Tree is a self-balanced search tree with multiple keys in every node and more than two children for every node.

Here, number of keys in a node and number of children for a node is depend on the order of the B-Tree. Every B-Tree has order. **B-Tree of Order m** has the following properties...

- **Property #1** - All the **leaf nodes** must be **at same level**.
- **Property #2** - All nodes except root must have at least **[m/2]-1** keys and maximum of **m-1** keys.
- **Property #3** - All non leaf nodes except root (i.e. all internal nodes) must have at least **m/2**children.
- **Property #4** - If the root node is a non leaf node, then it must have **at least 2** children.
- **Property #5** - A non leaf node with **n-1** keys must have **n** number of children.
- **Property #6** - All the **key values within a node** must be in **Ascending Order**.

For example, B-Tree of Order 4 contains maximum 3 key values in a node and maximum 4 children for a node.

Example



The following operations are performed on a B-Tree...

1. **Search**
2. **Insertion**
3. **Deletion**

Search Operation in B-Tree

In a B-Ttree, the search operation is similar to that of Binary Search Tree. In a Binary search tree, the search process starts from the root node and every time we make a 2-way decision (we go to either left subtree or right subtree). In B-Tree also search process starts from the root node but every time we make n-way decision where n is the total number of children that node has. In a B-Ttree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows...

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with first key value of root node in the tree.
- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, then check whether search element is smaller or larger than that key value.

- **Step 5:** If search element is smaller, then continue the search process in left subtree.
- **Step 6:** If search element is larger, then compare with next key value in the same node and repeate step 3, 4, 5 and 6 until we found exact match or comparision completed with last key value in a leaf node.
- **Step 7:** If we completed with last key value in a leaf node, then display "Element is not found" and terminate the function.

Insertion Operation in B-Tree

In a B-Tree, the new element must be added only at leaf node. That means, always the new keyValue is attached to leaf node only. The insertion operation is performed as follows...

- **Step 1:** Check whether tree is Empty.
- **Step 2:** If tree is **Empty**, then create a new node with new key value and insert into the tree as a root node.
- **Step 3:** If tree is **Not Empty**, then find a leaf node to which the new key value cab be added using Binary Search Tree logic.
- **Step 4:** If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.
- **Step 5:** If that leaf node is already full, then **split** that leaf node by sending middle value to its parent node. Repeat tha same until sending value is fixed into a node.
- **Step 6:** If the spilting is occuring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example

Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10.

Construct a B-Tree of order 3 by inserting numbers from 1 to 10.

insert(1)

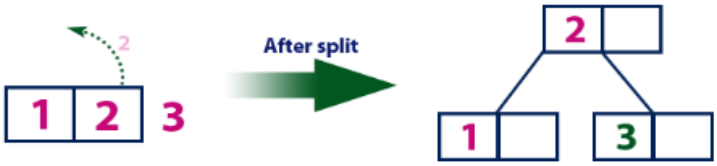
Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.

insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.

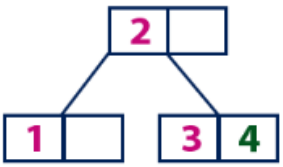
insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.



insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.

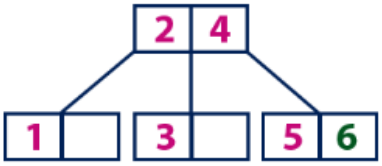


insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.

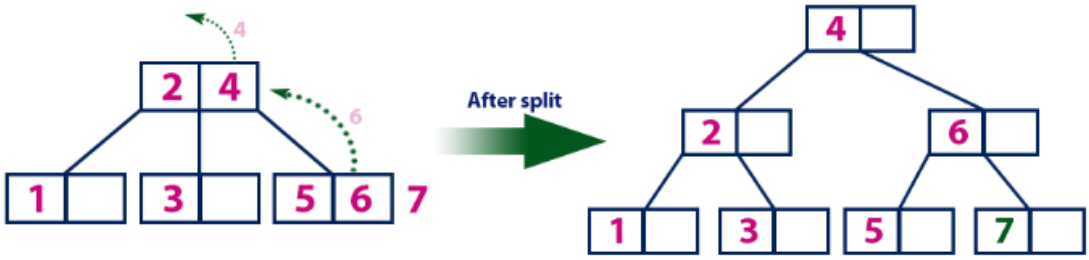
insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



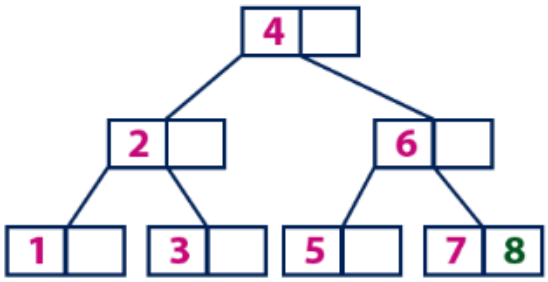
insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



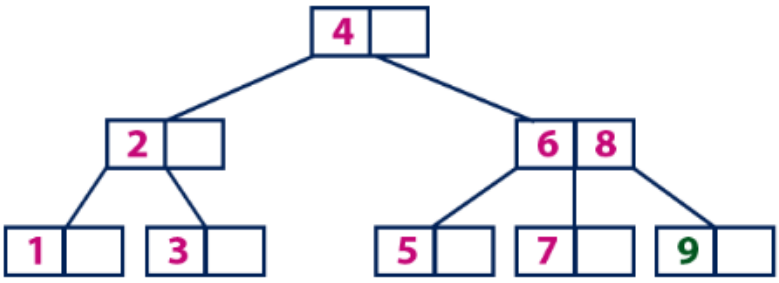
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



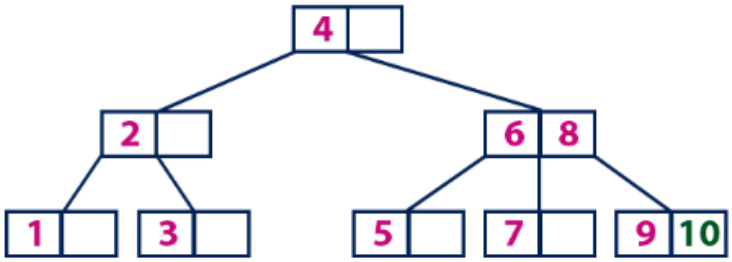
insert(9)

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8 '. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.



UNIT-V

Introduction to Graphs

Graph is a non linear data structure, it contains a set of points known as nodes (or vertices) and set of linkes known as edges (or Arcs) which connets the vertices. A graph is defined as follows...

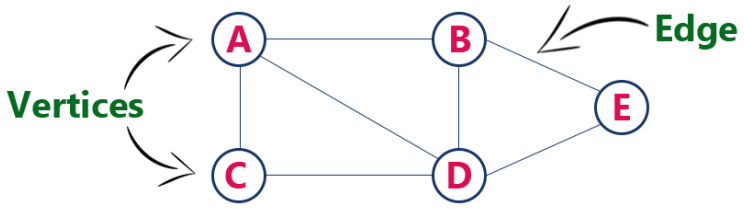
Graph is a collection of vertices and arcs which connects vertices in the graph

Graph is a collection of nodes and edges which connects nodes in the graph

Generally, a graph **G** is represented as **G = (V , E)**, where **V is set of vertices** and **E is set of edges**.

Example

The following is a graph with 5 vertices and 6 edges. This graph G can be defined as G = (V , E) Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



We use the following terms in graph data structure...

Vertex

A individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

Edge

An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph, the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

1. **Undirected Edge -** An undirected egde is a bidirectional edge. If there is a undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
2. **Directed Edge -** A directed egde is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. **Weighted Edge -** A weighted egde is an edge with cost on it.

Undirected Graph

A graph with only undirected edges is said to be undirected graph.

Directed Graph

A graph with only directed edges is said to be directed graph.

Mixed Graph

A graph with undirected and directed edges is said to be mixed graph.

End vertices or Endpoints

The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.

Origin

If an edge is directed, its first endpoint is said to be origin of it.

Destination

If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.

Adjacent

If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, Two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.

Incident

An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Outgoing Edge

A directed edge is said to be outgoing edge on its orign vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

Self-loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Path

A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

Graph Representations

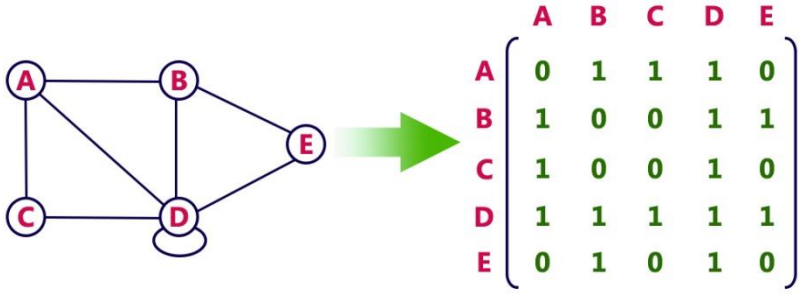
Graph data structure is represented using following representations...

1. **Adjacency Matrix**
2. **Adjacency List**

Adjacency Matrix

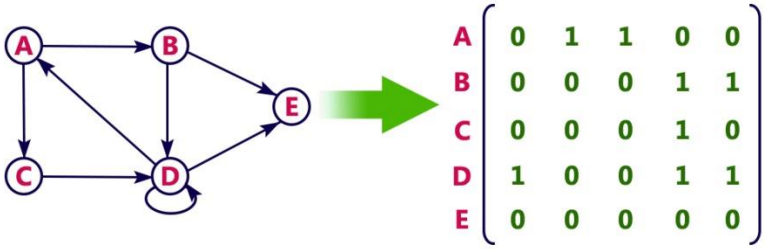
In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices. That means if a graph with 4 vertices can be represented using a matrix of 4X4 class. In this matrix, rows and columns both represents vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



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

Directed graph representation...

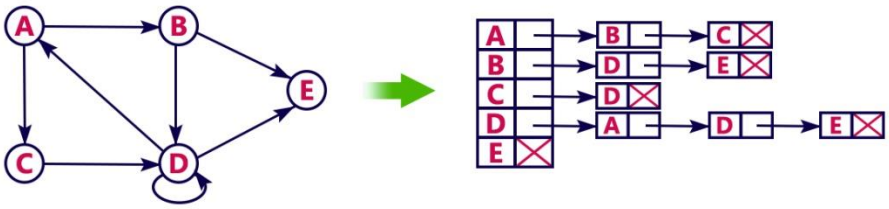


	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

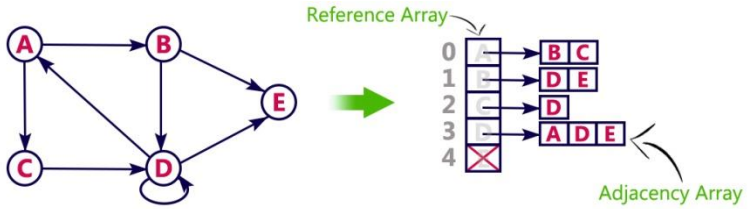
Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using array as follows..



Graph Traversals - DFS

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process. A graph traversal finds the egdes to be used in the search process without creating loops that means using graph traversal we visit all verticces of graph without getting into looping path. There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

DFS (Depth First Search)

DFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal of graph.
We use the following steps to implement DFS traversal...

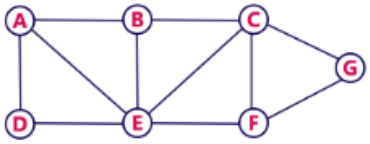
- **Step 1:** Define a Stack of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3:** Visit any one of the **adjacent** vertex of the verex which is at top of the stack which is not visited and push it on to the stack.
- **Step 4:** Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
- **Step 5:** When there is no new vertex to be visit then use **back tracking** and pop one vertex from the stack.
- **Step 6:** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7:** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Back tracking is coming back to the vertex from which we came to current vertex

Back tracking is coming back to the vertex from which we came to current vertex.

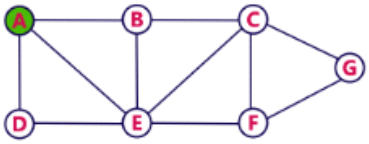
Example

Consider the following example graph to perform DFS traversal



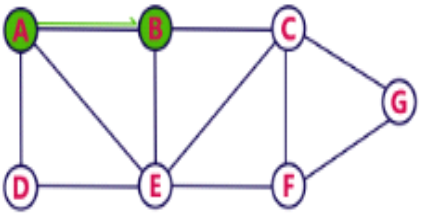
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



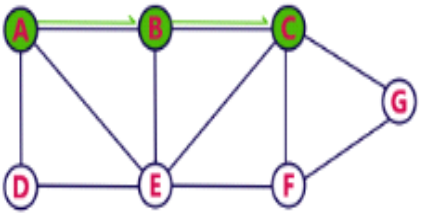
Step 2:

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



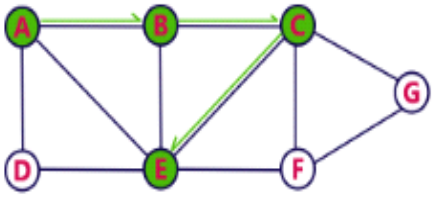
Step 3:

- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.



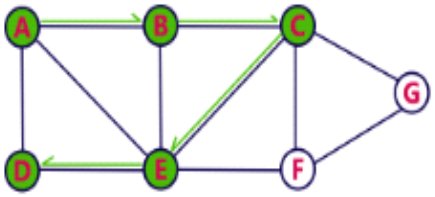
Step 4:

- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



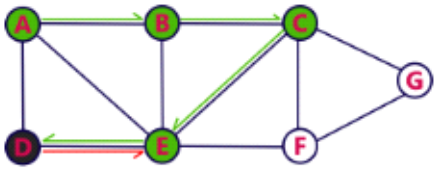
Step 5:

- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



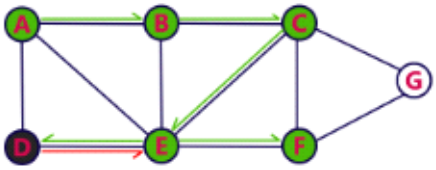
Step 6:

- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



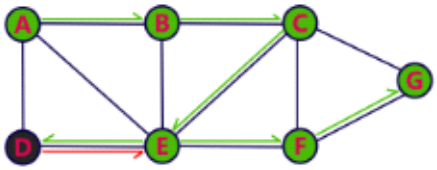
Step 7:

- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



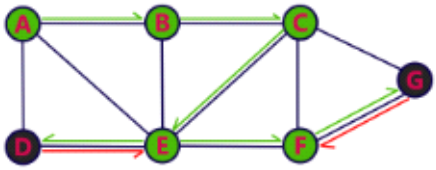
Step 8:

- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



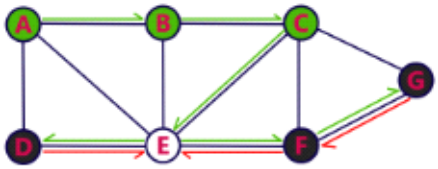
Step 9:

- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.



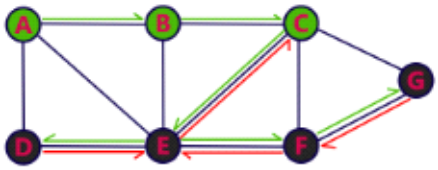
Step 10:

- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.



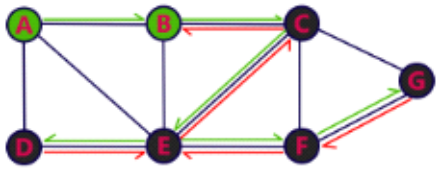
Step 11:

- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.



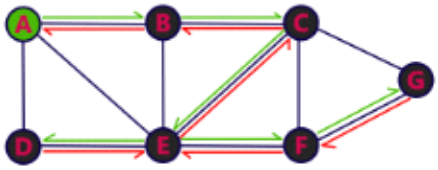
Step 12:

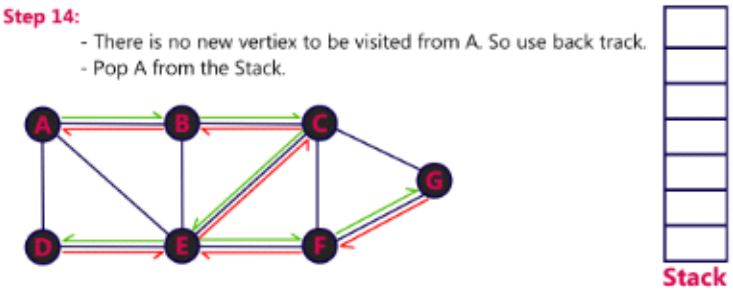
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.



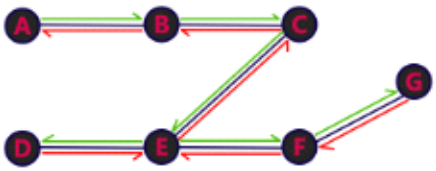
Step 13:

- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.





- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



Graph Traversals - BFS

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process. A graph traversal finds the egdes to be used in the search process without creating loops that means using graph traversal we visit all verticces of graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

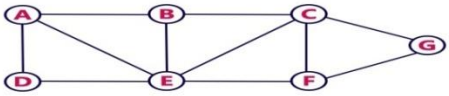
BFS (Breadth First Search)

BFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

We use the following steps to implement BFS traversal...

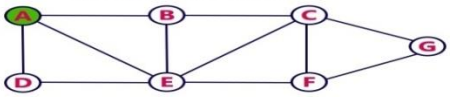
- **Step 1:** Define a Queue of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3:** Visit all the **adjacent** vertices of the verex which is at front of the Queue which is not visited and insert them into the Queue.
- **Step 4:** When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.
- **Step 5:** Repeat step 3 and 4 until queue becomes empty.
- **Step 6:** When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

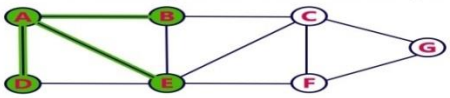


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

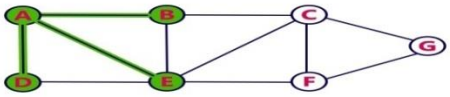


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

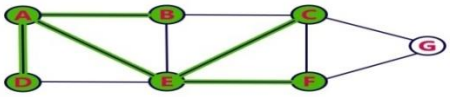


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

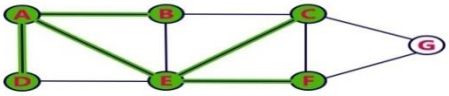


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

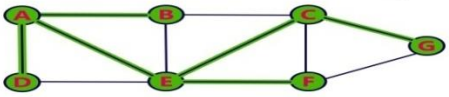


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

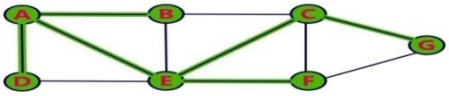


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

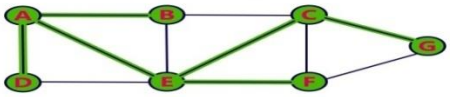


Queue

						G

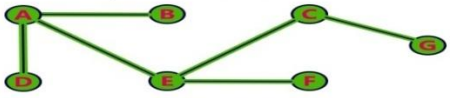
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Applications of Graphs:

Since they are powerful abstractions, graphs can be very important in modeling data. In fact, many problems can be reduced to known graph problems. Here we outline just some of the many applications of graphs.

1. Social network graphs: to tweet or not to tweet. Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.

2. Transportation networks. In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

3. Document link graphs. The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.

4. Scene graphs. In graphics and computer games scene graphs represent the logical or spacial relationships between objects in a scene. Such graphs are very important in the computer games industry.

5. Robot planning. Vertices represent states the robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths for autonomous vehicles.

6. Neural networks. Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 1011 neurons and close to 1015 synapses.

7. Graphs in compilers. Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.

8.Computer hardware: Compilers uses graph coloring algorithms for Register allocation to variables , Calculate parallelism degree, Very useful in analytical modeling[3] , Addressing the sequence of instruction execution , Resource allocation and Economizing the memory space(file organization).

Hashing

In all search techniques like linear search, binary search and search trees, the time required to search an element is depends on the total number of element in that data structure. In all these search techniquies, as the number of element are increased the time required to search an element also increased linearly.

Hashing is another approach in which time required to search an element doesn't depend on the number of element. Using hashing data structure, an element is searched with **constant time complexity**. Hashing is an effective way to reduce the number of comparisions to seach an element in a data structure.

Hashing is defined as follows...

Hashing is the process of indexing and retrieving element (data) in a datastructure to provide faster way of finding the element using the hash key.

Here, hash key is a value which provides the index value where tha actual data is likely to store in the datastructure.

In this datastructure, we use a concept called **Hash table** to store data. All the data values are inserted into the hash table based on the hash key value. Hash key value is used to map the data with index in the hash table. And the hash key is generated for every data using a **hash function**. That means every entry in the hash table is based on the key value generated using a hash function.

Hash Table is defined as follows...

Hash table is just an array which maps a key (data) into the datastructure with the help of hash function such that insertion, deletion and search operations can be performed with constant time complexity (i.e. O(1)).

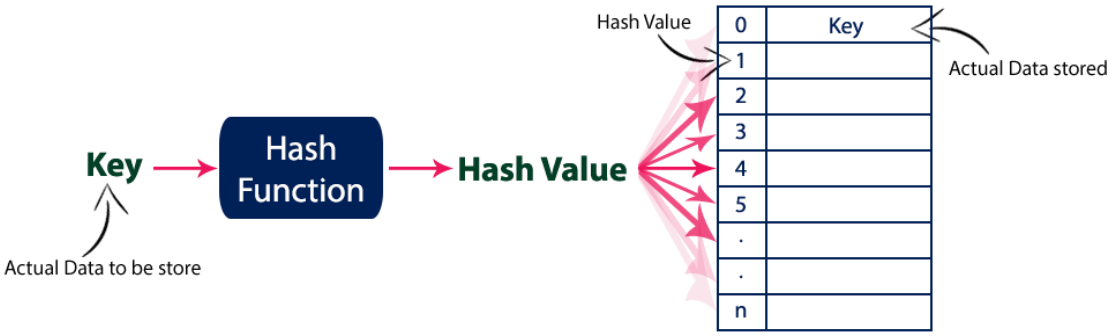
Hash tables are used to perform the operations like insertion, deletion and search very quickly in a datastructure. Using hash table concept insertion,

deletion and search operations are accoplished in constant time. Generally, every hash table make use of a function, which we'll call the **hash function** to map the data into the hash table.

A hash function is defined as follows...

Hash function is a function which takes a piece of data (i.e. key) as input and outputs an integer (i.e. hash value) which maps the data to a particular index in the hash table.

Basic concept of hashing and hash table is shown in the following figure...



Different Hash Functions:

1.Division Method:

It is the most simple method of hashing an integer x. this method divides x by M and then uses the remainder obtained. In this case, the hash function can be given as

h(X)=x mod M

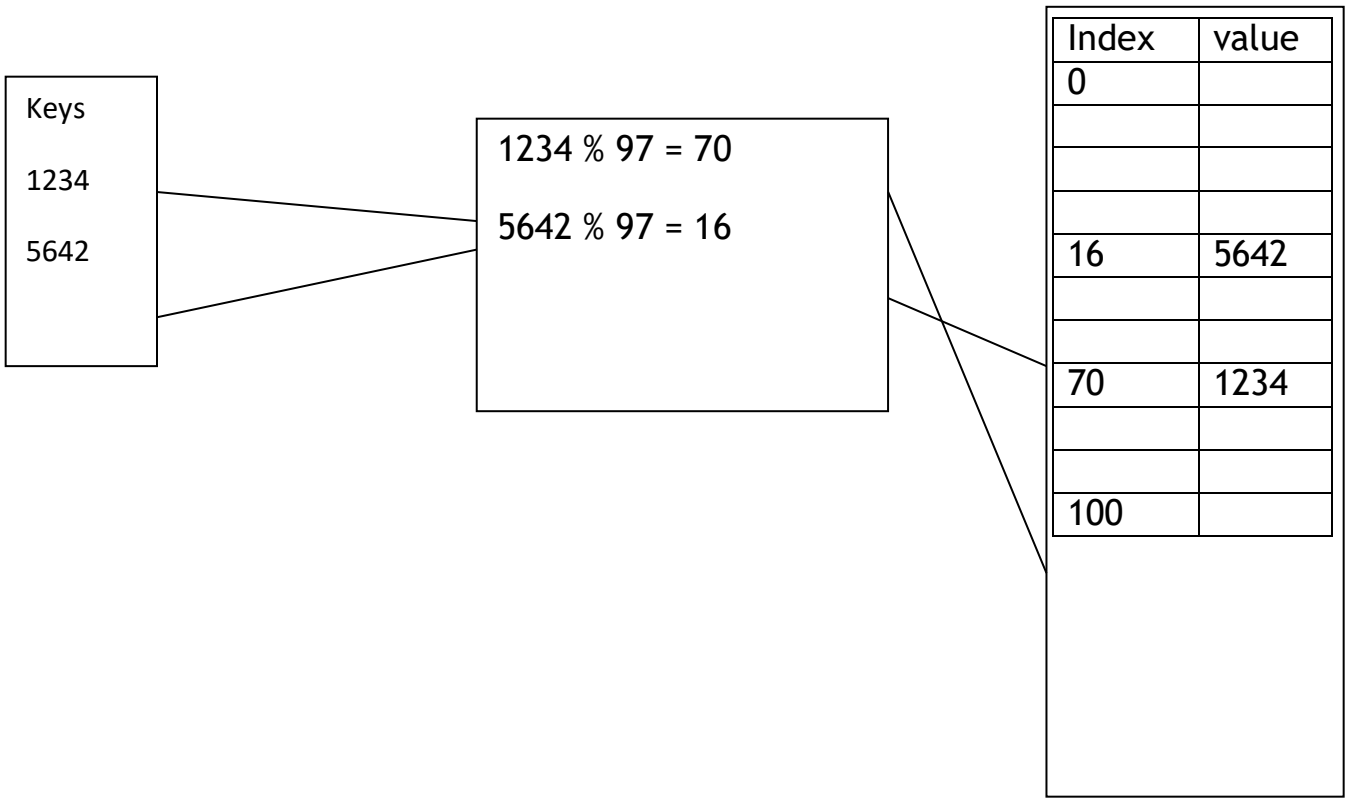
Example: Calculate the hash values of keys 1234 and 5462 .

Set M=97(prime no.)

hash values can be calculated as

h(1234)= 1234 % 97 = 70

h(5642)= 5642 % 97 = 16



Mid Square Method:

It works in two steps:

Step1: Square the value of the key. That is find k^2

Step2: Extract the middle r digits of the result obtained in step1.

In the mid-square method, the same r digits must be chosen from all the keys. Therefore, the hash function can be given as

h(k)=s where s is obtained by selecting r digits from k^2

Example: Calculate the hash value for keys 1234 and 5642 using mid-square method. The hash tale has 100 memory locations.

Solution: k=1234, k^2 = 152**27**56 h(1234)=27

k=5642, k^2 = 3183**21**64 h(5642)=21

observe that the 3rd and 4th digit starting from the right are chosen.

3) Folding Method:

It works in the following steps.

Step1: Divide the key value into a number of parts. (i.e.) k divided into parts k1,k2,k3,…kn where each part has the same number of digits, except the last part which may have lesser digits than the other parts.

Step2: Add the individual parts. i.e. obtain the sum of k1+k2+k3…+kn. The has value is produced by ignoring the last carry if any.

Example: Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678,321, and 34567.

Solution:

Key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56, and 7
Sum	**1**34	33	97
Hash value	34(ignore carry)	33	97

Collisions:

Collisions occur when the hash function maps two different key to the same location. Obviously, two records cannot be stored in the same location.

Therefore, a method used to solve the problem of collision also called collision resolution technique.

The two most popular methods of resolving collisions are :

1. Open addressing
2. Chaining

1. Collision resolution by open Addressing:

   Once a collision takes place, open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position.

   In this technique the hash table contains two types of values. Sentinel values(eg-1) and data values. The presence of a sentinel value indicates that the location contains no data value at present but can be used to hold a value.

   When a key is mapped to a particular memory location, then the value it holds is checked. If it contains a sentinel value, the n the location is free and the data value can be stored in it.

   However, if the location is already has some data value stored in in it, then other slots are examined systematically in the forward direction to find a free slot.

   If even a single free location is not found, then we have an overflow condition.

   The process of examining memory locations in the hash table is called probing.

Open addressing technique can be implemented using

i) Linear probing
ii) Quadratic probing
iii) Double hashing
iv) Rehashing

Linear probing: In this, we linearly probe for next slot. The following hash function is used to resolve the collision.

h(k,i)=(h(k)+i) mod m

ex: I = 1,2,3….

Example: consider has functions as key mod 7 and sequence of keys are 50,700,76,85,92,73,101.

Index	val	Index	val	Index	val	Index	val	Index	val	Index	val	Index	val	Index	val	Index	val
					7		7		7				7		7		
	-		-		0		0		0				0		0		70
0	1	0	15	0	0	0	0	0	0	0		0	0	0	0	0	0
	-				5		5		5				5		5		
1	1	1	0	1	0	1	0	1	0	1		1	0	1	0	1	50
	-		-						8				8		8		
2	1	2	1	2	-1	2	-1	2	5	2		2	5	2	5	2	85
	-		-										9		9		
3	1	3	1	3	-1	3	-1	3	-1	3		3	2	3	2	3	92
	-		-												7		
4	1	4	1	4	-1	4	-1	4	-1	4		4	-1	4	3	4	73
	-		-														10
5	1	5	1	5	-1	5	-1	5	-1	5		5	-1	5	-1	5	1
	-		-				7		7				7		7		
6	1	6	1	6	-1	6	6	6	6	6		6	6	6	6	6	76
								collistion occurred				collistion occurred				collistion occurred	
								insert 85at next				insert 92at next				insert101at next	
								free slot				free slot				free slot	

ii)Quadratic Probing:

We look for i^2 th slot in the hash table. The following hash function is used to resolve the collection.

$h(k,i)=(h,1k)+i^2)$ mod m.

i=1,2,3 …..

iii) Double hashing: We use another hash function has2(x) and look for i*hash2(x). slot in the rotation. The following hash function is used to resolve the collision.

h(k,i)=h(k)+i*hash2(x) mod m.

Where I =1,2,3…

Iv)Rehashing: When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations. In such cases, a better option is to create a new hash table with size double of the original hash table.

All the entries in the original hash table will then have to be moved to the new hash table. This is done by taking each entry, computing its new hash table. This is done by taking each entry, computing its new hash value, and then inserting it in the new hash table.

Through rehashing seems to be a simple process, it is quite expensive and must therefore not be done frequently.

Collision resolution by changing

In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location.

That is location 1 in the hash tale points to the head of the linked list of all the key values hashed to 1 then location 1 in hash table contains NULL.

The following Fig. show how the key values are mapped to a location in the hash table and stored in a linked list that corresponds to that location.