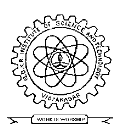


N.B.K.R. INSTITUTE OF SCIENCE AND TECHNOLOGY::VIDYANAGAR
(AUTONOMOUS)
*****

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

I-B.Tech II SEM(R-23)

23CS1201–DATA STRUCTURES

(COMMON TO CSE, IT, AI&DS, AND ALLIED BRANCHES)

UNIT-I

Introduction to Linear Data Structures: Definition and importance of linear data structures, Abstract data types (ADTs) and their implementation, Overview of time and space complexity analysis for linear data structures.

Searching Techniques: Linear and Binary Search.

Sorting Techniques: Bubble sort, Selection sort, and Insertion Sort.

UNIT-II

Linked Lists: Singly linked lists - representation and operations, doubly linked lists and circular linked lists, Comparing arrays and linked lists, Applications of linked lists.

UNIT-III

Stacks: Introduction, properties and operations, implementing stacks using arrays and linked lists, Applications of stacks in expression evaluation, balanced parentheses, reversing list etc.

Queues: Introduction, properties and operations, implementing queues using arrays and linked lists, Queue applications in OS and simulation experiments.

Types of Queues: Types - Circular Queues, Priority Queues, Deques, and supporting operations.

UNIT-IV

Trees- Introduction, Types and basic properties.

Binary Trees–Definition, Tree traversals, Tree representations. **Binary Search Trees** – Definition, properties and applications. **AVL trees-** Introduction and basic operations. **Heap** – Introduction and types, Heap sort.

UNIT-V

Graphs: Introduction, Basic terminologies, Graph Representations, Bi-connected components, Topological sorting. **Hashing:** introduction to hashing and hash functions, basic implementation and operations of Hash tables, Caching, **Collision resolution techniques -** chaining and open addressing.

UNIT-I

Introduction to Linear Data Structures: Definition and importance of linear data structures, Abstract data types (ADTs) and their implementation, Overview of time and space complexity analysis for linear data structures.

Searching Techniques: Linear and Binary Search.

Sorting Techniques: Bubble sort, Selection sort, and Insertion Sort.

1.1 Introduction to Linear Data Structures:

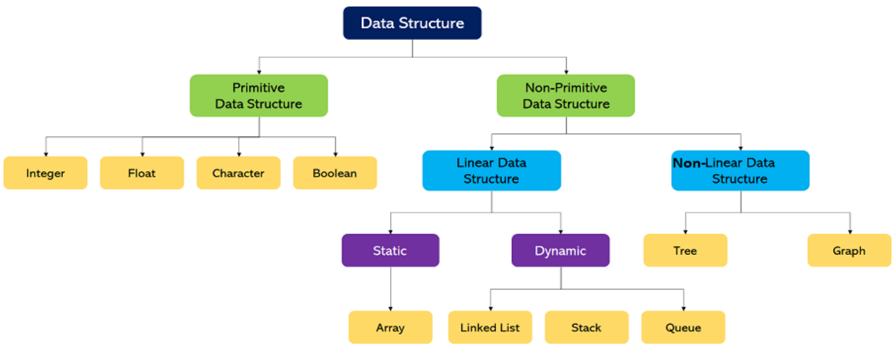
Introduction: Programs consists of two things, algorithms and data structures. A good program is a combination of both algorithm and a data structure. An algorithm is a step-by-step recipe for solving an instance of a problem. Every single procedure that a computer performs is an algorithm. An algorithm states the action to be executed and the order in which the actions are to be executed.

Definition: A data structure represents the logical relationship that exists between individual elements of data to carry out a certain task.

[Or]

A data structure is an arrangement of data in computer's memory (or on a disk).

Classification of data structures [or] Types of data structures:



Primary data structures: Primary data structures are the basic data structures that operate upon the machine instructions. The primary data structures are also called **primitive data structures.**

Example: Integers, floating –point numbers, character constants, string constants and pointers.

Secondary data structures: secondary data structures are more complicated data structures derived from primary data

structures. The secondary data structures are also called **non-primitive data structures.**

Example: Arrays, stacks, queues, records, files , linked list, trees, graphs etc.

The secondary data structures are classified as

a) Linear data structures b) Non-linear data structures

a) Linear data structures: Data structures using sequential allocation are called linear data structures.

Example: Arrays, records, stacks, queues , linked list

Based on memory allocation, the Linear Data Structures are further classified into two types:

a) Static data structures b)Dynamic data structures

a) Static data structures: If a data structures is created using static memory allocation. It is known as static data structures or fixed size data structures.

Example: Arrays and structures

b) Dynamic data structures: If a data structures is created using dynamic memory allocation. It is known as dynamic data structures or variable size data structures.

Example: Linked list

b) Non-linear data structures: data structures don’t have sequential allocation are called non-linear data structures.

Example: Trees and Graphs

1.2 Why data structure?

The following are the advantages of using the data structure:

- These are the essential ingredients used for creating fast and powerful algorithms.
- They help us to manage and organize the data.
- Data structures make the code cleaner and easier to understand.

1.3 Advantages of Linear Data Structures

- **Efficient data access:** Elements can be easily accessed by their position in the sequence.
- **Dynamic sizing:** Linear data structures can dynamically adjust their size as elements are added or removed.
- **Ease of implementation:** Linear data structures can be easily implemented using arrays or linked lists.
- **Versatility:** Linear data structures can be used in various applications, such as searching, sorting, and manipulation of data.

- **Simple algorithms:** Many algorithms used in linear data structures are simple and straightforward.

1.4 Abstract Data Type (ADT)

What is abstract data type?

An abstract data type is an abstraction of a data structure that provides only the interface to which the data structure must adhere. The interface does not give any specific details about something should be implemented or in what programming language.

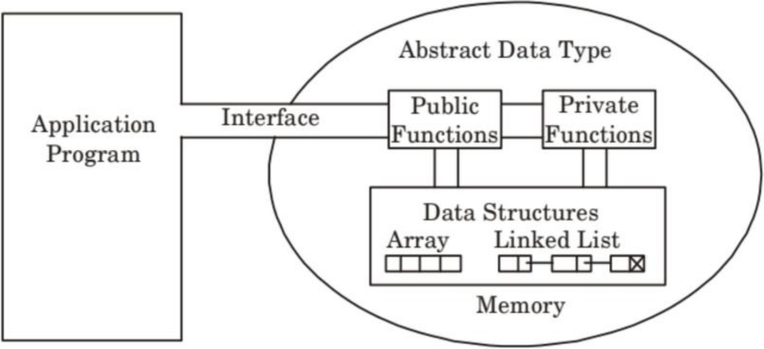
In other words, we can say that abstract data types are the **entities that are definitions of data and operations but do not have implementation details.** In this case, we know the data that we are storing and the operations that can be performed on the data, but we don't know about the implementation details.

Abstract data type model

Before knowing about the abstract data type model, we should know about abstraction and encapsulation.

Abstraction: It is a technique of hiding the internal details from the user and only showing the necessary details to the user.

Encapsulation: It is a technique of combining the data and the member function in a single unit is known as encapsulation.



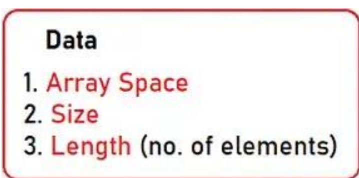
The above figure shows the ADT model. There are two types of models in the ADT model, i.e., the public function and the private function. The ADT model also contains the data structures that we are using in a program. In this model, first encapsulation is performed, i.e., all the data is wrapped in a single unit, i.e., ADT. Then, the abstraction is performed means showing the operations that can be

performed on the data structure and what are the data structures that we are using in a program.

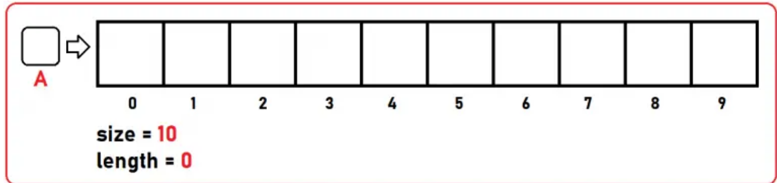
1.5 Array as ADT (Abstract Data Type) using C

Representation of Data on Array:

Now let's look at the representation of the data on an array.



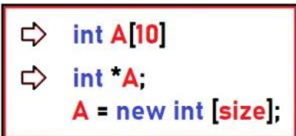
The first thing is we need an array space of some size. Here we initialize an array of size = 10 and length = 0 because there are no elements in the array.



Here we have two methods to initialize our Array:

1. **Inside Stack Memory:** As shown in the below image, A [10] will create a contiguous block of memory index from 0 to 9 inside Stack.

2. **Inside Heap Memory:** We can also create our array dynamically with the new keyword, The new operator denotes a request for memory allocation, as shown in the below image, we created a pointer of int* type and in the next step we point it to a contiguous block of memory inside Heap. We have created our array inside heap memory by using a pointer.



The Complete C Code:

```
#include <stdio.h>
#include <stdlib.h>
struct Array {
  int* A;
  int size;
  int length;
};
int main() {
   struct Array arr;
   printf("Enter Size of an Array: ");
```

```
    scanf("%d", &arr.size);
    arr.A = (int*)malloc(sizeof(int) * arr.size);
    arr.length = 0;
    printf("Enter Number of Elements: ");
    scanf("%d", &arr.length);
    printf("Enter All Elements: \n");
    for (int i = 0; i < arr.length; i++) {
        scanf("%d", &arr.A[i]);
    }
    getchar();
}
```

Output:

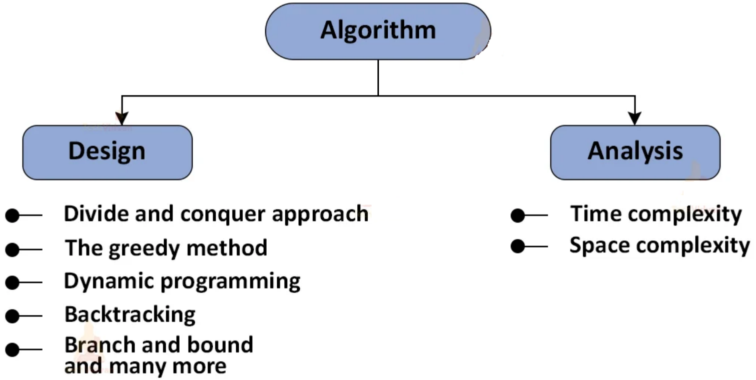
```
Enter Size of an Array: 10
Enter Number of Elements: 5
Enter All Elements:1 2  3   4   5
```

1.6 Overview of time and space complexity analysis for linear data structures.

1.6.1 Performance Analysis: The efficiency of an algorithm can be decided by measuring the performance of algorithm. We can measure the performance of an algorithm

by computing amount of time and storage requirement. We can analyse an algorithm by two ways.

1. By checking the correctness of an algorithm.
2. By measuring time and space complexity of an algorithm.



Time Complexity

Time complexity is defined in terms of how many times it takes to run a given algorithm, based on the length of the input.

Example

Statements	s/e	Frequency	Total steps
Algorithm sum(a,n)	0	-	0
{	0	-	0
s:=0;	1	1	1

for i:=1 to n do	1	n+1	n+1
s:=s+a[i]	1	n	n
return s;	1	1	1
}	0	-	0
			2n+3

Space Complexity

The amount of memory used by a program to execute it is represented by its space complexity. The space requirement S(P) can be given as

S(P)=C + SP

Example

Statements	**s/e**	**Frequency**	**Total steps**
Algorithm sum(a,n)	0	-	0
{	0	-	0
s:=0;	1	1	1
for i:=1 to n do	1	n+1	n+1
s:=s+a[i]	1	n	n
return s;	1	1	1
}	0	-	0
			2n+3

The space requirement for algorithm given in example is S(P)= 2n+3. Neglect the constants

∴ space complexity is O(n)

Time Complexity	Space Complexity
Calculates time needed	Calculates memory space needed
Counts time for all statements.	Counts memory space for all variables even inputs and outputs
Depends mostly on input data size	Depends mostly on auxiliary variables size
More important for solution optimization	Less important with modern hardwares
Time complexity of merge sort is $\mathcal{O}(nlogn)$	Space complexity of merge sort is $\mathcal{O}(n)$

1.6.2 Asymptotic Notations

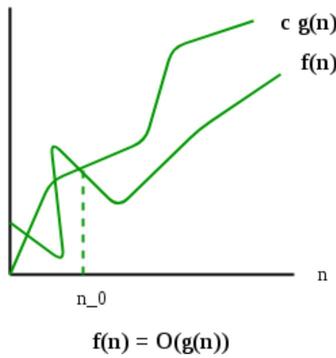
To choose the best algorithm, we need to check efficiency of each algorithm. The efficiency can be measured by computing time complexity of each algorithm. Asymptotic notation is a shorthand way to represent the time complexity. Using asymptotic notations we can give time complexity as "fastest possible", "slowest possible" or "average time". There are mainly three asymptotic notations:

1. Big-O Notation (O-notation)
2. Omega Notation (Ω-Notation)
3. Theta Notation (Θ-Notation)

1. Big-O Notation (O-notation)

Big-oh notation denoted by 'O' is a method of representing the upper bound of algorithms running time. Using big-oh notation we can give longest amount of time taken by the algorithm to complete.

Definition: The function f(n)=O(g(n)) iff there exist positive constants c and n_o such that f(n) ≤ cg(n) for all n ≥ n_0

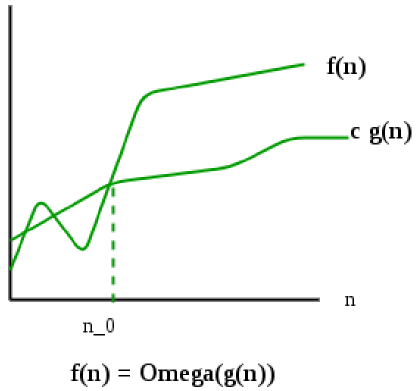


f(n) = O(g(n))

2. Omega Notation (Ω-Notation)

Omega notation denoted by Ω is a method of representing lower bound of algorithms running time. Using omega notation we can denote shortest amount of time taken by the algorithm to complete.

Definition: The function f(n)=Ω **(g(n)) iff there exist positive constants c and n_0 such that f(n) ≥c*g(n) for all n, n≥n_0**

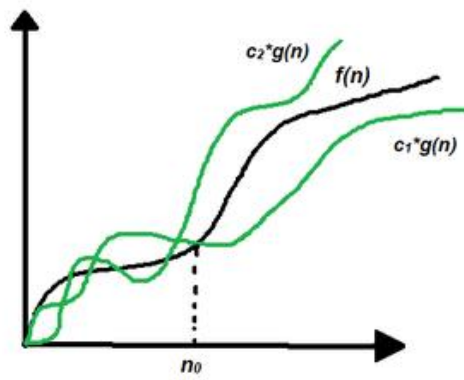


f(n) = Omega(g(n))

3. Theta Notation (Θ-Notation)

The Theta notation denoted as Θ is a method of representing running time between upper bound and lower bound.

Definition: The function f(n)= Θ(g(n)) iff there exist positive constants c1, c2 and n_0 such that **c1g(n) ≤f(n) ≤c2 g(n) for all n, $n \geq n_0$**



Linear Data structures - Time Complexities

Data Structure	Insert	Delete	Access	Search
Array	O(N)	O(N)	O(1)	O(N)
String	O(N)	O(N)	O(1)	O(N)
Queue	O(1)	O(1)	O(N)	O(N)
Stack	O(1)	O(1)	O(N)	O(N)
Linked List	O(1), if inserted on head O(N), elsewhere	O(1), if deleted on head O(N), elsewhere	O(N)	O(N)

Searching Algorithms - Time Complexities

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)

Sorting Algorithms - Time Complexities

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(k)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n + k)
Bucket Sort	O(n + k)	O(n + k)	O(n^2)	O(n)

1.7 Searching Techniques: Linear and Binary Search.

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

Sequential Search: In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search.

Interval Search: These algorithms are specifically designed for searching in sorted data-structures. This type of searching algorithms are much more efficient than Linear Search as they repeatedly target the centre of the search structure and divide the search space in half. For Example: Binary Search.

1.7.1 Linear Search

Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.

Example

Linear Search

Search = 6	1	2	4	6	7	9	13	17
	ind=0							
1 != 6 , search next index	1	5	3	6	7	18	13	17
		ind=1						
5 != 6 , search next index	1	5	3	6	7	18	13	17
			ind=2					
3 != 6 , search next index	1	5	3	6	7	18	13	17
				ind=3				
6 == 6 , found the answer 3	1	5	3	6	7	18	13	17

Example: Write a C program to check whether the given element is present in a list or not by using linear search.

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int  a[10]={11,21,35,46,62,67,83,95,99,105 };
 int key;
 printf("\n Enter the key value");
 scanf("%d",&key);
 for(i=0;i<n;i++)
   if(a[i]==key)
     {
      printf("\n Element is present at position %d", i+1);
      exit(0);
     }
     printf("\n Element is not present");
getch();
```

}

Output: Enter the key value 35

Element is present at position 3

Enter the key value 10

Element is not present

Linear search Time complexity

The above algorithm's runtime complexity is O(N) and space complexity is O(1).

1.7.2 Binary Search

Binary search is an efficient searching method. While searching the elements using this method the most essential thing is that the elements in the array should be sorted one.

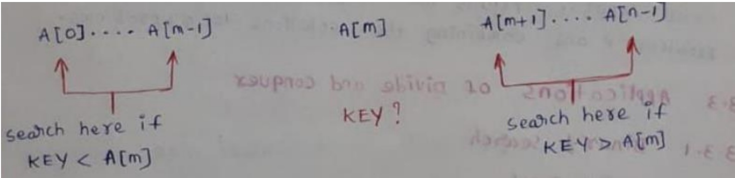
An element which is to be searched from the list of elements stored in array A[0……n-1] is called **KEY** element.

Let A[m] be the mid element of array A. then there are three conditions that needs to be tested while searching the array using this method.

1. If KEY =A[m] then desired element is present in the list.
2. Otherwise if KEY<A[m] then search the left sublist.

3. Otherwise if KEY> A[m] then search the right sublist.

This can be represented as



Binary search Algorithm

Algorithm for Binary Search

1) Initialize low=0, high=n-1 and found =0 [un successful]

2) Repeat step 3 and step 4 while(low<=high)

3) mid=(low+high)/2

4) if key <a[mid] then

High=mid-1

Else if key > a[mid] then

low=mid+1

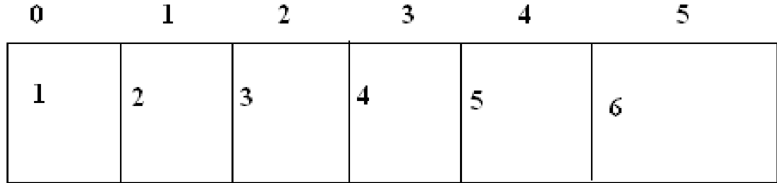
else

found=1

5) return(found)

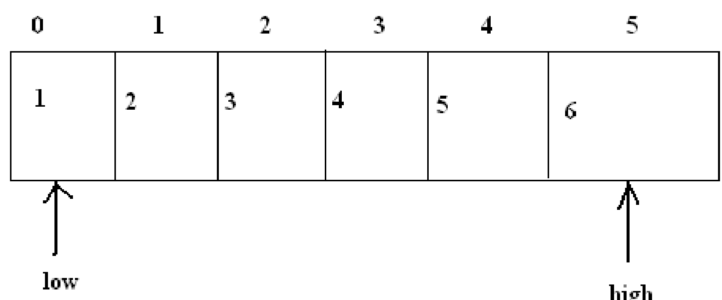
Example: Let us consider the elements 1 2 3 4 5 6

Sol:



Initially low=0 and high=n-1[high=6-1]

high=5



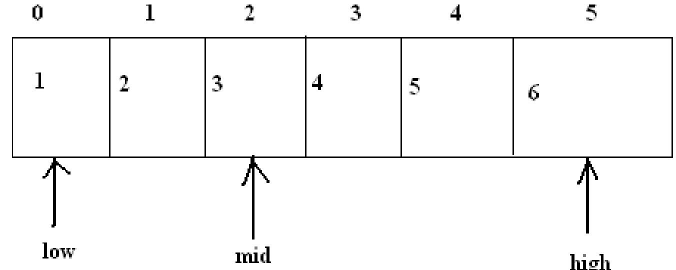
Here low<=high 0<= 5 so find mid value.

Now calculate mid=(low+high)/2

mid=(0+5)/2

mid=2

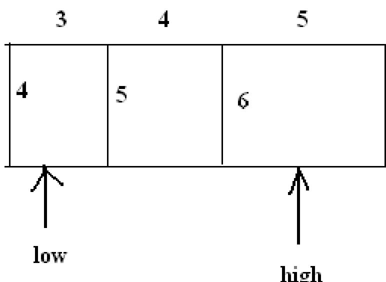
Select the searching element or key value is 5.



mid=3 and key =5. Here key value is greater than mid value so low=mid+1

Low=2+1

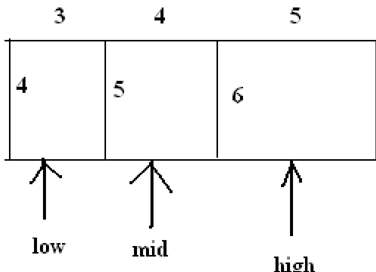
Low=3. Here key value greater than mid value then the searching will be done right half.



mid=(low+high)/2

mid=(3+5)/2

mid=4



if mid value is equal to the key value then we can say an element is present at position 4.

Example: Write a C program to check whether an element is present in the list or not by using binary search.

```
#include<stdio.h>
#include<conio.h>
main()
{
int a[10] = {11,21,35,46,62,67,83,95,99,105};
int key;
clrscr();
printf("\n Enter the key value");
scanf("%d",&key);
if(binary_search(a,10,key))
printf("\n Elements is present");
else
printf(”\n Elements is not preseny”);
getch();
}
```

```
int binary_search(int a[], int n, int key)
{
int found=0,mid, low=0,high=n-1;
while(low<=high)
{
mid=(low+high)/2;
if(key<a[mid])
high=mid-1;
else if (key>a[mid])
low=mid+1;
else
{
found=1;
break;
}
}
return(found);
}
```

OUTPUT

```
Enter the key value   11
Element is present
Enter the key value   88
Element is not present
```

Binary Search complexity

Now, let's see the time complexity of Binary search in the best case, average case, and worst case. We will also see the space complexity of Binary search.

1. Time Complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

Best Case Complexity - In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is O(1).

Average Case Complexity - The average case time complexity of Binary search is O(logn).

Worst Case Complexity - In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is O(logn).

2. Space Complexity

Space Complexity	O(1)

1.8 Sorting Techniques

Sorting: Arrangement of elements either in ascending order or descending order then it is said to be a sorting.

Exchange sort: Making repeated comparisons and exchanging the adjacent items, if some condition is satisfied.

Sorting will be useful to search, insert or delete a data item in a list. There are various sorting techniques

1.8.1 Bubble Sort

1.8.2 Selection Sort

1.8.3 Insertion Sort

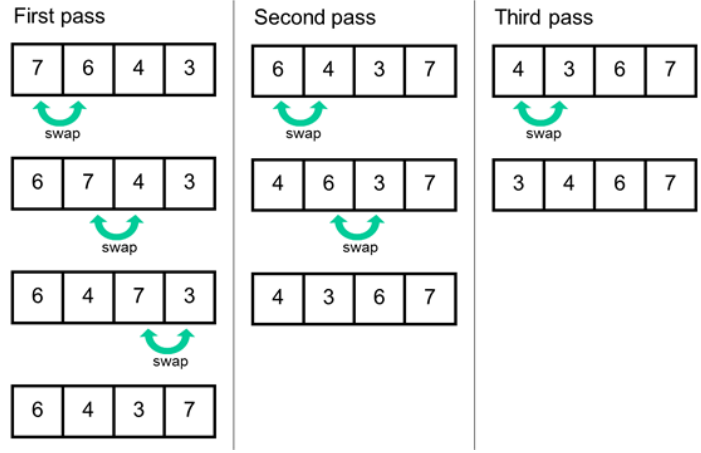
1.8.1 Bubble Sort

The bubble sort, also known as the ripple sort, is one of the least efficient sorting algorithms. However, it is probably the simplest to understand. At each step, if two adjacent elements of a list are not in order, they will be swapped. Thus, larger elements will "bubble" to the end, (or smaller elements will be "bubbled" to the front, depending on implementation) and hence the name.

The principle of a bubble sort is illustrated below: Compare the first two values and swap if necessary. Then compare the next pair of values and swap if necessary. This

process is repeated n-1 times, where n is the number of values being sorted.

Example



Algorithm for Bubble sort or Exchange sort

Bubble(Arr, n)
where Arr is an array of 'n' elements
Step 1:Repeat for i=0,1,2......n-1
Step 2:Repeat for j=i+1 to n-1
Step 3:if(Arr[i]>Arr[j]) then
    Interchage Arr[i] and Arr[j]
    [End if structure]
Step 4: Increment j by 1
Step 5: [End of step 2 for loop]
Step 6: [End of step 1 for loop]
Step 7:Print the sorted array
    End Bubble().

Example 1: Write a C program to sort the numbers using Bubble sort

```
#include<stdio.h>
#include<conio.h>
void bubble(int  a[], int n);
int i, j, n, temp,a[10];
void main()
{
 printf("\n How many elements you want");
 scanf("%d", &n);
 printf("\n Enter the elements");
 for(i=0;i<n;i++)
 scanf("%d", &a[i]);
 bubble(a, n);
 printf("\n The sorted list is");
 for(i=0;i<;i++)
 printf("%d\t", a[i]);
 getch();
}

void bubble(int a[], int n)
 {
  for(i=0;i<=n-1;i++)
   {
    for(j=i+1;j<=n-1;j++)
     {
      if(a[i]>a[j])
       {
        temp=a[i];
        a[i]=a[j];
        a[j]=temp;
       }
```

```
        }
    }
}
```

Output

How many elements you want 4
Enter the elements 7 6 4 3
The sorted list is 3 4 6 7

Unsorted List	56	91	35	72	48	68
	56	91	35	72	48	68
	35	56	91	72	48	68
	35	48	91	72	56	68
	35	48	56	91	72	68
	35	48	56	68	91	72
Sorted List	35	48	56	68	72	91

Fig: Trace of a bubble sort

1.8.2 Selection Sort

In selection sort find the smallest value in the array and exchange it with the first element. Find the next smallest and exchange it with the second element and continue in this manner till all elements are completed.

Algorithm for selection sort

selectionsort(a,n)
Where a is an array of n elements
Step 1:Repeat for i=0,1,2,......n-1
Step 2:Assign k=i, min=a[i]

```
Step 3:Repeat for j=i+1 to n-1
Step 4: if( a[j]<min) then
        min=a[j]
        k=j
        [End of if structure]
Step 5:  [End of step 3  for loop]
Step 6:Assign a[k]=a[i]
            a[i]=min
Step 7: [End of step 1 for loop]
Step 8: print the sorted array a
       End selectionsort()
```

Example : Write a C program to sort the numbers using selection sort

```
#include<stdio.h>
#include<conio.h>
void selectionsort(int  a[], int n);
int i, j, k,n,min,a[10];
void main()
{
  printf("\n How many elements you want");
  scanf("%d", &n);
  printf("\n Enter the elements");
  for(i=0;i<n;i++)
  scanf("%d", &a[i]);
  selectionsort(a, n);
  printf("\n The sorted list is");
  for(i=0;i<;i++)
  printf("%d\t", a[i]);
  getch();
}
```

```
void selectionsort(int a[], int n)
{
  for(i=0;i<=n-1;i++)
   {
    k=i;
    min=a[i];
    for(j=i+1;j<=n-1;j++)
      if(a[j]<min)
       {
        min=a[j];
        k=j;
       }
       a[k]=a[i];
       a[i]=min;

    }
 }
```

Output

How many elements you want 4
Enter the elements 7 6 4 3
The sorted list is 3 4 6 7

Unsorted List	56	91	35	72	48	68
	56	91	35	72	48	68
	35	91	56	72	48	68
	35	48	56	72	91	68
	35	48	56	68	91	72
Sorted List	35	48	56	68	72	91

Fig: Trace of a selection sort

1.8.3 Insertion Sort

The main idea of insertion sort is to consider each element at a time, into the appropriate position relative to the sequence of previously ordered elements, such that the resulting sequence is also ordered.

Example: The insertion sort can be easily understood if we know to play cards. Imagine that we are arranging cards after it has been distributed before we in front of the table. As each new card is taken and compared with the cards in hand. The card is inserted in proper place within the cards in hand, by pushing one position to the left or right. This procedure proceeds until all the cards are placed in the hand are in order.

Algorithm for Insertion sort

Insertionsort(a,n)

Where a is an array of n elements.

Step 1:Repeat for i=1,2,3....n-1

Step 2:Assign temp=a[i]

Step 3:Repeat for j=i t0 1

Step 4:if(temp<a[j-1]) then

a[j]=a[j-1]

else

goto step 7

[End of if structure]

Step 5:Decerement j by 1

Step 6:[End of step 3 for loop]

Step 7:Assign a[j]=temp

Step 8:[End of step1 for loop]

Step 9:print the sorted array a

End Insertsort()

Note: The insertion sort is also known as **straight sorting technique**.

Example: Write a C program to sort the numbers using insertion sort

```
#include<stdio.h>
#include<conio.h>
void insertionsort(int  a[], int n);
int i, j,n,temp,a[10];
void main()
{
  clrscr();
  printf("\n How many elements you want");
  scanf("%d", &n);
  printf("\n Enter the elements");
  for(i=0;i<n;i++)
  scanf("%d", &a[i]);
  insertionsort(a, n);
  printf("\n The sorted list is");
```

```
 for(i=0;i<n;i++)
 printf("%d\t", a[i]);
 getch();
}

void insertionsort(int a[], int n)
{
  for(i=1;i<=n-1;i++)
   {
     temp=a[i];
     for(j=i;j>=1;j--)
      {
       if(temp<a[j-1])
       a[j]=a[j-1];
       else
       break;
      }
        a[j]=temp;

    }
}
```

Output

How many elements you want 4
Enter the elements 7 6 4 3
The sorted list is 3 4 6 7

Unsorted List	56	91	35	72	48	68
	56	91	35	72	48	68
	35	56	91	72	48	68
	35	48	56	91	72	68
	35	48	56	91	72	68
	35	48	56	68	91	72
	35	48	56	68	72	91
Sorted List	35	48	56	68	72	91

Fig: Trace of a insertion sort

Algorithm	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$

UNIT-II

Linked Lists: Singly linked lists - representation and operations, doubly linked lists and circular linked lists, Comparing arrays and linked lists, Applications of linked lists.

2.1 Linked Lists

Linked list or list is an ordered collection of elements. Each element in the list is referred as a node. Each node contains two fields namely

a) Data field

b) Link field

a) Data field: The data field contains the actual data of the element to be stored in the list.

b) Link field: The link field also referred as the next address field contains the address of the next node in the list.

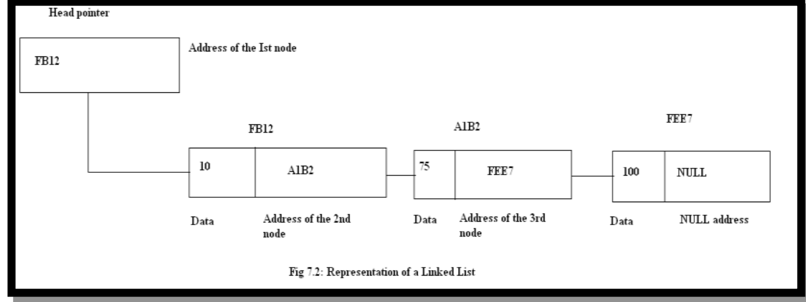


Fig 7.2: Representation of a Linked List

The linked list is shown in fig 7.2 consists of three nodes, each with a data field and a link field. A linked list contains a pointer, referred as the **head pointer.** The head pointer points to the first node in the list that stores the address of the first node of the list.

The data field contains the actual information which is to be stored in the list. The data field of the first node stores the value 10. The link field of the first node contains the address of the second node, similarly the second node of the list stores the value 75 in the data field and address of the third node in the link field. The last node of the list contains only the information part in the data field and the address field stores the NULL pointer. This NULL pointer is used to indicate the **end of the list.**

The address stored in the linked list are divided into three types namely

a) External address

b) Internal address

c) NULL address

a) External address: External address is the address of the first node in the list. This address is stored in the head pointer which points to the first node in the list. The entire

linked list can be accessed only with the help of the head pointer.

b) Internal address: Internal address is the address stored in each and every node of the linked list except the last node. The content stored in the link field is the address of the next node.

c) NULL address: NULL address is the address stored by the NULL pointer of the last node of the list, which indicates the end of the list.

2.2 Types of linked list: There are different types of linked lists. They can be classified as,

2.2.1 Singly linked list

2.2.2 Doubly linked list

2.2.3 Circular linked list

2.2.1 Singly linked list: It's having data field and linked field is referred to as the singly linked list. In which each node has a single link to its next node. This list is also referred as a linear linked list. The head pointer points to the first node in the list and the NULL pointer is stored in the link field of the last node in the list, which indicated end of list.

We can traverse (move) in a singly linked list in only one direction.

Basic operations on a singly linked list: The basic operations that can be performed on singly linked list are

1. Creation of a list
2. Insertion of a node
3. Deletion of a node
4. Display of a list(Traversal of a list)
5. Count the number of nodes

1. Creation of a list: The creation of a list involves three processes. They are

a) Creating a node

b) Reading details for a node from user

c) Connect the node with the list.

1. Creation of a list: The creation of a list involves three processes. They are

a) Creating a node

b) Reading details for a node from user

c) Connect the node with the list.

Algorithm for declaration of structure node

struct node
data: data field
Link: link field (Address of next struct node)
End struct

Algorithm for allocation memory for the new node

Getnode( )

size: integer; newnode:node

step 1: set size=get the size of the node

step 2:set newnode=allocate space in memory for the size of size and return the initial address.

step 3:return newnode

End Getnode( )

Algorithm for reading the content for the new node

Readnode(newnode:node)

step 1: read, newnode->data

step 2:set newnode->link=NULL

step 3:return

End readnode( )

Algorithm for create list

createlist( )
head, last, nenode:node
step 1: set newnode=Getnode( )
step 2:call readnode(newnode)
step 3:set head=newnode
step 4:set last=newnode
step 5:if we want to add another node proceed otherwise stop
step 6:set newnode=getnode( )
step 7: call readnode(newnode)
step 8:Assign last->link=newnode
step 9:Assign last=last->link
step 10:goto step 5
End createlist( )

2. Insertion of a node: One of the most important operation that can be done in a singly linked list is the insertion of a node. Memory is to be allocated for the new node before reading the data. The new node will contain empty data field and empty link field. The data field of the new node is then stored with the information read from the user. The link field of the new node is assigned to NULL.

The new node can then be inserted in the list at three different places namely,

a) Inserting as a first node in the list

b) Inserting as a last node in the list

c) Inserting an intermediate node in the list

a) Inserting as a first node in the list:

The following steps are followed to insert a new node in the start of the list.

1) Get the new node using Getnode( ), and read the details of the node using Readnode( ).

2) Check whether the list is empty or not (ie., check whether the head pointer is pointing to NULL or not).

3) If the list is empty, assign new node as head. If the list is not empty, follow the next steps.

4) The link field of the new node is made to point the data field of the first node(ie., head node) in the list by assigning the address of the first node.

5) The head pointer is made to point the data field of the new node by assigning the address of the new node.

InsertFirst() function is used for inserting a new node in the first position of the list.

Algorithm for inserting a node as the first node in the list

```
Insert_first(head:node)
New node:node
step 1:set new node=Getnode( )
step 2:call read node(new node)
step 3: if(head==NULL)
        set head =newnode
        return
step 4:Assign newnode->link=head
step 5:set head=newnode
      End Insert_first( )
```

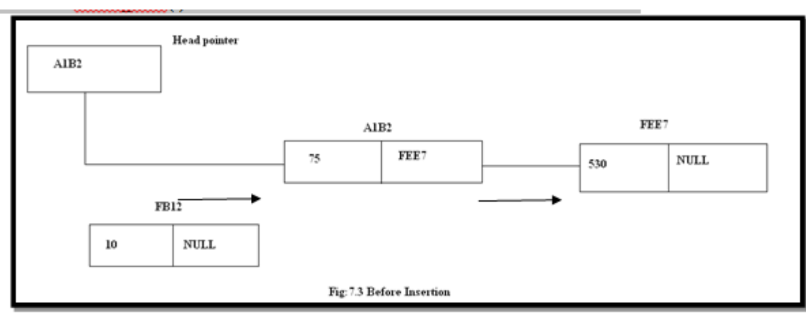


Fig: 7.3 Before Insertion

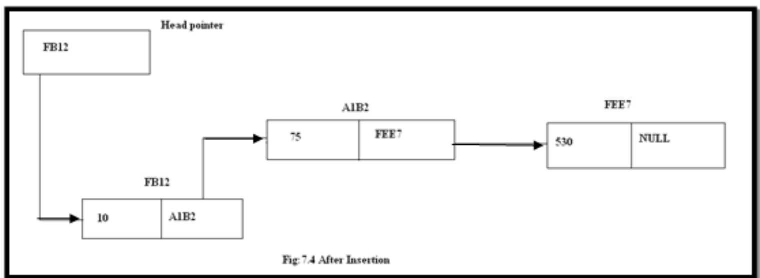


Fig: 7.4 After Insertion

b) Inserting as a last node in the list:

The following steps are followed to insert a new node in the end of the list.

1) Get the new node using GetNode(), and read the details of the node using ReadNode( ).

2) Check whether the list is empty or not. It the list is empty , assign new node as head. If the list is not empty, follow the next steps.

3) The link field of the last node is made to point the data field of the new node in the list by assigning the address of the new node.

4) The link field of the new node is set to NULL.

Algorithm for inserting a node as the last node in the list

```
Insert_last(head:node)
Last,newnode:node
step 1:Set newnode =getnode()
step 2:Call readnode(newnode)
step 3: if(head==NULL)
       Set head=newnode
       Return
step 4:Set last=head
step 5:Repeat  while(last->link!=NULL)
      Assign last=last->link
step 6:Assign last->link=newnode
      End insert_last( )
```

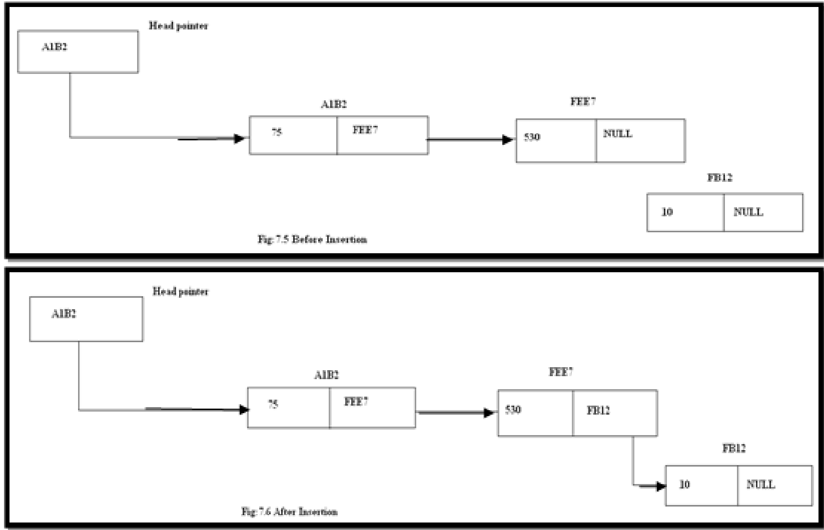


Fig 7.5 Before Insertion

Fig 7.6 After Insertion

c) Inserting an intermediate node in the list

The following steps are followed , to insert a new node in any intermediate position in the list.

1) Get the new node using GetNode( ) , and read the details of the node using ReadNode().

2) Check whether the list is empty or not. If the list is empty, assign new node as head. If the list is not empty, follow the next steps.

3) Get the address of the preceding node after which the new node is to be inserted.

4) The link field of the new node is made to point the data field of the next node by assigning its address.

5) The link field of the preceding node is made to point the data field of the new node by assigning the address of the new node.

Algorithm for inserting a node at any intermediate position in the list

```
Inser_middle(Head:node)
last, newnode:node
condition: Data of the any one node in the last for insert
Step 1:set newnode=Getnode()
Step 2:call readnode(new node)
Step 3:If(head==NULL)
       set head=newnode
       Return
       [End structure]
Step 4:print "Enter the data of node after which the insertion is to
be made"
Step 5:Read, condition
step 6:set last=head
Step 7:Repeat  while(last!=NULL)
Step 8:If(last->data==Condition) then
       Assign newnode->link=last->link
       Assign last->link=newnode
       Return
       else
       Assign last=last->link
       [End structure]
Step 9:[End of step 7 while structure]
Step 10:Print "condition is not available"
        End Insert_middle()
```

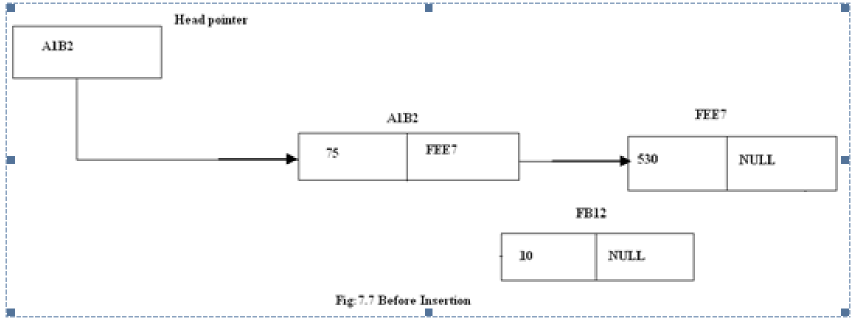


Fig:7.7 Before Insertion

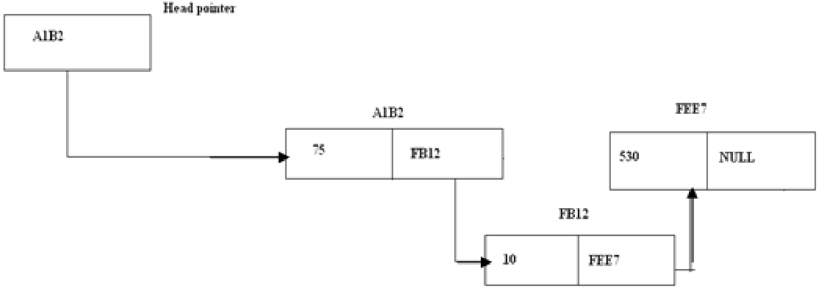


Fig: 7.7 After Insertion

3) Deletion of a node: Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely

1) Deleting the first node from the list.

2) Deleting the last node from the list.

3) Deleting an intermediated node from the list.

1) Deleting the first node from the list

The following steps are followed, to delete a node from the start of the list

a) Check whether the list is empty or not (i.e. check whether the head pointer is pointing to NULL or not). If the list is not empty, follow the next steps.

b) set the head pointer to the second node in the list.

c) Release the memory for the deleted node.

Algorithm for deleting the first node from the list

```
Delete_First(Head:node)
Delete:node
Step 1:if(Head==NULL)
       printf "list is empty"
       return
       [End of if structure]
Step 2:set delnode=Head
Step 3:Assign head=head->link
Step 4:Print "Deleted data is ", delnode->data
Step 5:Call releasenode(delnode)
       End Delete_First( )
```

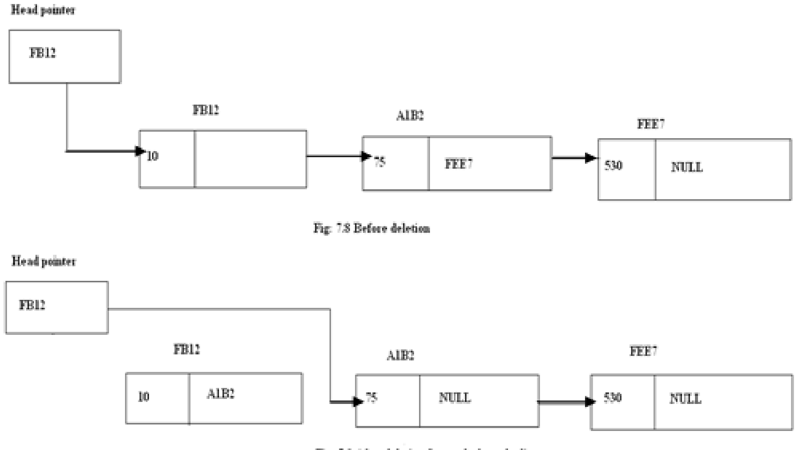


Fig: 7.8 Before deletion

Fig: 7.8 After deleting first node from the list

2) Deleting the last node from the list

The following steps are followed , to delete a node from the end of the list.

a) Check whether the list is empty or not. If the list is not empty, follow the next steps

b) The link field of the previous node is set to NULL.

c) Release the memory for the deleted node.

```
        Algorithm for deleting the last node from the list
Delete_lat(Head:node)
last, prev, Delnode:node
Step 1:if(Head==NULL)
        printf "list is empty"
        return
        [End of If structure]
Step 2:if(head->link==NULL) then
       set Delnode=Head
       Set Head=NULL
       printf "Deleted data is", delnode->data
```

```
      Return
      [End of if structure]
Step 3: set last=head
step 4:Repeat while(last->link!=NULL)
Step 5:Assign prev=last
Step 6:Assign last=last->link
step 7:[End of step 4 while loop]
Step 8:Set delnode=last
Step 9:prev->link=NULL
Step 10:print "Deleted data is " delnode->data
step 11:Call releasenode(Delnode)
       End Delete_last( )
```

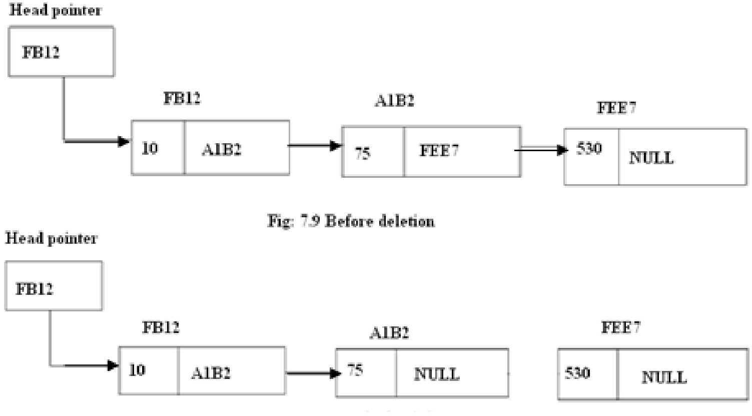


Fig: 7.9 Before deletion

Fig: 7.9 After deleting the last node from the list

c) Deleting an intermediate node from the list:

The following steps are followed , to delete a node from an intermediate position in the list.

a) Check whether the list is empty or not(i.e. check whether the head pointer is pointing to NULL or not). If the list is not empty , follow the next steps.

b) The link field of the previous node is made to point the data field of the next node, by assigning its address.

c) Release the memory for the deleted node

Algorithm for deleting a node at any intermediate position in the list

```
Delete_middle(head: node)
Last, prev, delnode:node
Deldata: data of the node is the node to delete
Step 1: if( head==NULLL)
Print "List is empty"
Return
[End of if structure]
Step 2: print  "Enter the data of the node in the list for deletion"
Step 3: read del data
Step 4: if (head->data ==del data) then
        Set del node=head
        Assign head=head->link
        Print "deleted data is ", del node-> data
        Call release node(del node)
        Return
        [End of if structure].
Step 5: set last=head->link
Step 6: set prev=head
Step 7: Repeat while(last!=NULL)
Step 8: if(last->data ==Del data) then
        Set del node= last
```

```
            Assign prev->link=lat->link
           Print  "The deleted data is ", del node->data
            Call return node(del node)
             Return
            Else
            Assign lat=last->link
            Assign prev=prev->link
            [End of if structure]
Step 9: [ End of step 7 while loop ]
Step 10: print "del data is not  available in the list"
             End delete_middle()
```

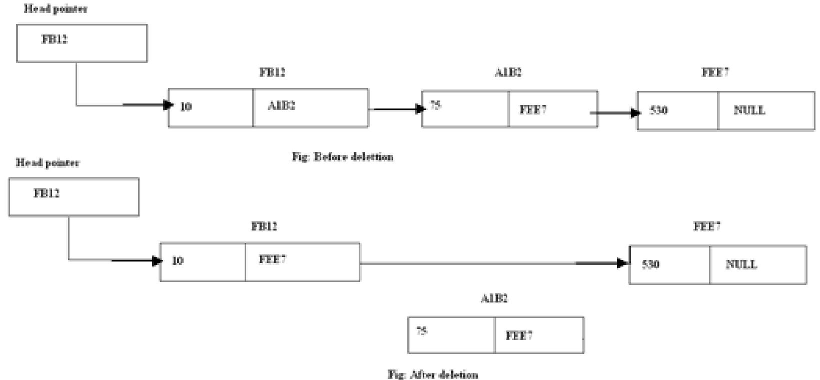


Fig: Deleting the intermediate node from the list

5) Traversal of a list: To read the information or to display the information in a linked list, we have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached. Traversing a list involves the following steps.

a) Check whether the head pointer is pointing to NULL or not. If yes display "list is empty" and terminate the process. Otherwise follow the next steps.

b) Display the information in the data field stored in the head pointer.

c) Traverse the list from one node to another by advancing the head pointer.

Algorithm for displaying the contents of the list

```
View_list(Head:node)
Step 1: if(head==NULL)
          Print  "List is empty"
         Return
         [End of if structure ]
Step 2: While(head!=NULL)
Step 3: print " The data is ", head->data
Step 4: head=head->link
Step 5: [ End of step2 while structure ]
          End view_list()
```

6) Count the number of nodes in the list: To count the nodes in a linked list, we have to traverse (move) a linked list node by node from the first node, until the end of the list is reached. Counting the number of nodes in the list involves the following steps.

Algorithm for counting the number of nodes in the list

Count_list(Head:node)

Count:integer

Step 1: set count=0

Step 2: if(head==NULL)

  Print "list is empty"

  Return count

  [End of if structure]

Step 3: while(head!=NULL)

Step 4: count=count+1

Step 5: head=head->link

Step 6: [End of while structure]

Step 7: return count

End count_list()

Example: Write a C program on Single Linked List

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
typedef struct linked node;
node *create_node();
void append();
void insert();
void delete();
void display();
```

```
void count();
node *start=NULL,*last=NULL;
struct linked
{
int data;
struct linked *next;
};
void main()
{
int ch;
clrscr();
do
{
printf("\n 1.append");
printf("\n 2.insert");
printf("\n 3.del");
printf("\n 4.display");
printf("\n 5.count");
printf("\n Enter the choice");
scanf("%d",&ch);
switch(ch)
{
case 1:append();
break;
case 2:insert();
break;
case 3:delete();
break;
case 4:display();
break;
case 5:count();
break;
default:printf("Invalid choice");
```

```
}
printf("\n do u want continue press(y/n)");
ch=getch();
}
while(ch=='y');
getch();
}
node *create_node()
{
node *temp;
int x;
temp=(node *)malloc(sizeof(node));
printf("Enter the data");
scanf("%d",&x);
temp->data=x;
temp->next=NULL;
return(temp);
}
void append()
{
node *temp=NULL;
temp=create_node();
if(start==NULL)
{
start=temp;
last=start;
}
else
{
last->next=temp;
last=temp;
}
}
```

```
void insert()
{
node *temp,*temp1,*q;
int pos,i=0;
temp=create_node();
temp1=start;
printf("Enter the position");
scanf("%d",&pos);
if(pos==1)       /*add at beg*/
{
temp->next=start;
start=temp;
}
else
{
for(i=1;i<pos-1;i++)      /*ada after*/
{
temp1=temp1->next;
}
q=temp1->next;
temp1->next=temp;
if(q==NULL)
last=temp;
temp->next=q;
}
}
void delete()
{
node *temp1,*q;
int pos,i=0;
if(start==NULL)
{
printf("list is empty");
```

```
return;
}
printf("Enter the position to delete");
scanf("%d",&pos);
if(pos==1)      /*delete at beg*/
{
temp1=start;
start=start->next;
free(temp1);
}
else
{
for(temp1=start,i=1;i<pos-1;i++)
{
temp1=temp1->next;
}
q=temp1->next;
if(q->next==NULL)
last=temp1;
temp1->next=temp1->next->next;
free(q);
}
}
void display()
{
node *temp;
if(start==NULL)
{
printf("list is empty");
return;
}
for(temp=start;temp!=NULL;temp=temp->next)
printf("%d",temp->data);
```

```
}
void count()
{
node *temp=start;
int c=0;
while(temp!=NULL)
{
temp=temp->next;
c++;
}
printf("No.of nodes %d",c);
}
```

2.2.2 Doubly linked list

Basic operations in a doubly linked list: The basic operations that can be performed on doubly linked list are

1) Creation of a list

2) Insertion of a node

3) Deletion of a node

4) Traversal of a list

5) Count the no.of nodes

1) Creation of a list : Creation of list involves three process. They are

1) Creating a node

2) Reading details for a node from user

3) Connect the node with the list

Algorithm for declaration of structure NODE

Struct node

Data: data field

Flink: link field (Address of next structure node)

Blink: link field(Address of previous struct node)

End struct

Algorithm for allocating memory for the new node

Getnode()
Size: integer
Newnode: node
Step 1: set size=get the size of the node
Step 2: set Newnode=allocate space in memory for the size of SIZE and return the initial address
Step 3: Return Newnode
End Getnode()

Algorithm for reading the content for the new node
Readnode(Newnode:node)
Step 1: read, newnode->data
Step 2: set newnode->flink=NULL
Step 3: set newnode->blink=NULL
Step 4: return
End Readnode()

Algorithm for createlist()

Createlist()
Head, last, newnode:node
Step 1: Set newnode=Getnode()
Step 2: Call readnode(newnode)
Step 3: Set head=newnode
Step 4: Set last=newnode
Step 5: If we want to add another node proceed otherwise return
Step 6:Set newnode=getnode()
Step 7: Call readnode(newnode)
Step 8: Assign last-> flink=newnode
Step 9: Assign newnode->blink=last
Step 10: Assign last=last->flink
Step 11: Goto step 5
End createlist()

2) Insertion of a node: One of the most primitive operations that can be done in a doubly linked list is the insertion of a node. Memory is to be allocated for the new node before reading the data. The newnode will contain empty data field and empty forward and backward link fields. The data field of he new node is then stored with the information read from the user. Both the link field of the newnode are assigned to NULL.

The new node can then be inserted in the list at three different places namely

a) Inserting as a first node in the list

b) Inserting as a last node in the list

c) Insertiong an intermediate node in the list

a) Inserting as a first node in the list:

Algorithm for inserting a node as the first node in the lsit
Insert_first(Head:node)
Newnode:node
Step 1: Set Newnode=Getnode()
Step 2:Call readnode(newnode)
Step 3: if(head==NULL)
    Set head=Newnode
    Return
    [End of if structure]
Step 4: Assign Newnode->flink=Head
Step 5: Assign Head->Blink=newnode
Step 6: Assign Head=Newnode
End Insert_first()

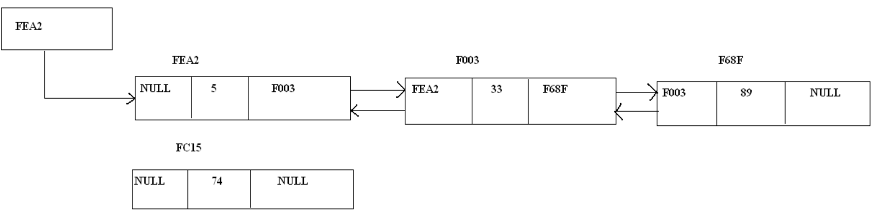


Fig: Before Insertion

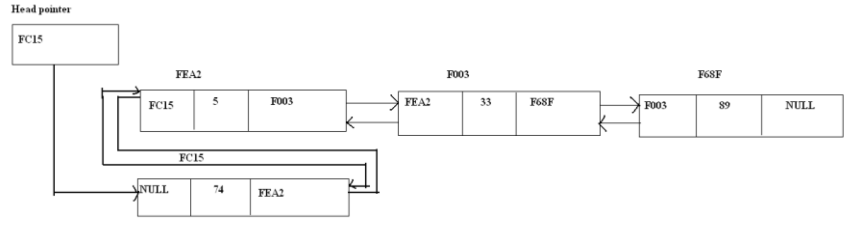


Fig: Inserting as a first node in the list

b) Inserting as a last node in the list:

Algorithm for inserting a node as the last node in the list

```
Insert_last(Head:node)
Last, newnode:node
Step 1: set newnode=Getnode()
Step 2: Call read node(newnode)
Step 3: if(Head==NULL)
            Set  head=newnode
            Return
           [ End of if structure]
Step 4: Set last=Head
Step 5: Repeat while(last->Flink!=NULL)
            Assign last=last->Flink
        [ End of while structure ]
Step 6: Assign last-> Flink=newnode
Step 7: Assign newnode->Blink=last
End Insert_last()
```

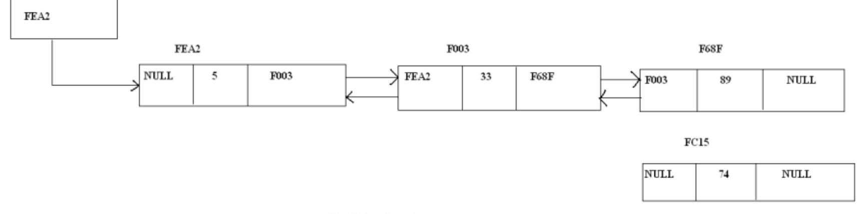


Fig: Before Insertion

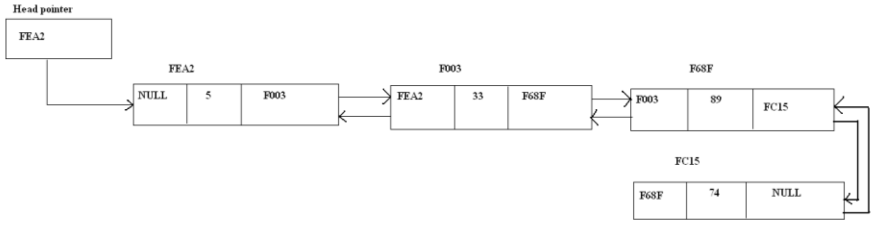


Fig: Inserting as a last node in the list

c) Inserting an intermediate node in the list:

Algorithm for inserting a node at any intermediate position in the list

Insert_middle(Head:node)
Last, next, newnode:node
Condition: Data of the any one Node in the last for insert.
Step 1: set Newnode=Getnode()
Step 2: call readnode(newnode)
Step 3: if(Head==NULL)
 Set head=newnode
 Return
 [End of if structure]
Step 4: print " Enter the data of node after which the insertion is to be made"
Step 5: read, condition
Step 6: set last=head
Step 7: repeat while(last!=NULL)
Step 8: if(last->data ==Condition) then

```
            Assign next=last->Flink
            Assign newnode->Flink=Next
           Assign newnode->Blink=last
           Assign last->Flink=Newnode
           If( Next!=NULL)
           Assign next-> Blink=new node
            [ End of  if structure ]
 Step 9:  [ End of step 7 while structure]
Step 10:  print  " Condition is not available"
Step 11: Return
End Insert_middle()
```

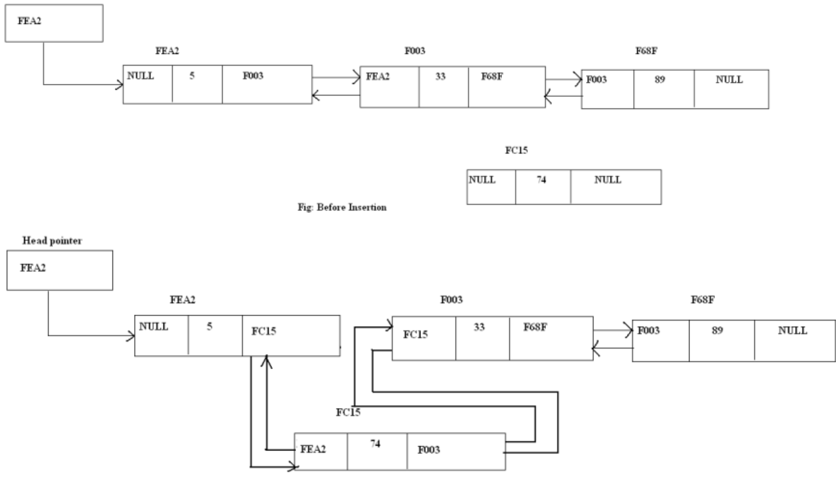


Fig: Before Insertion

Fig: Inserting as an intermediate node in the list

4) Delection of a node: Another primitive operation that can be done in a doubly linked list is the deletion of a node.

Memory is released for the node to be deleted. A node can be deleted fro m the list from three different places namely

a) Deleting the first node from the list

b) Deleting the last node from the list

c) Deleting an intermediated node from the list.

a) Deleting the first node from the list:

Algorithm for deleting the first node from the list

```
Delete_first(Head: node)
Delnode:node
Step 1: if(head==NULL)
            Print "list is empty"
            Return
          [End of if structure]
Step 2: set delnode=head
Step 3: assign head=heade->Flink
Step 4: if(head!=NULL)
            Assign head->Blink=NULL
             [ End of if structure]
Step 5: print " Deleted data is", delnode->data
Step 6: call release node(delnode)
Step 7: Return
            End delete_first()
```

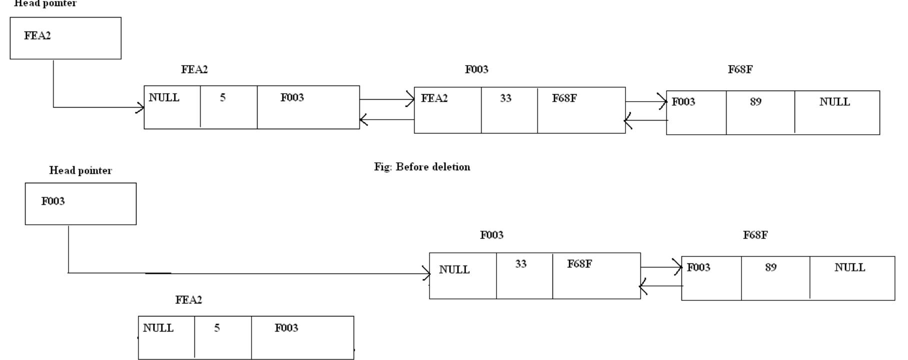


Fig: Deleting the first node from the list

b) Deleting the last node from the list:

Algorithm for deleting the last node from the list

Del_last(Head:node)
Last, prev, delnode:node
Step 1: if(head==NULL)
    Print "list is empty"
    Return
    [End of if structure]
Step 2: if(head->Flink==NULL) then
    Set delnode=head
    Print " deleted data is ", delnode->data
    Return
    [End of if structure]
Step 3: set last=head
Step 4: repeat while(last-> Flink!=NULL)
    Assign last=last->Flink
    [End of while loop]
Step 5: set delnode=last
Step 6: assign last->Blink->Flink=NULL
Step 7: print " deleted data is", delnode->data
Step 8: call releasenode(delnode)

Step 9: return
 End delete_last()

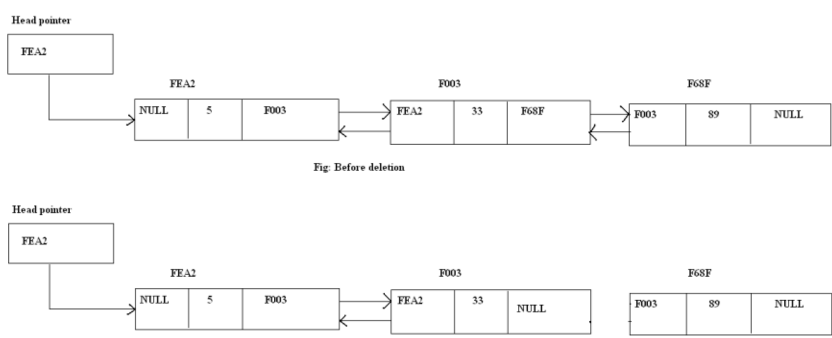


Fig: Before deletion

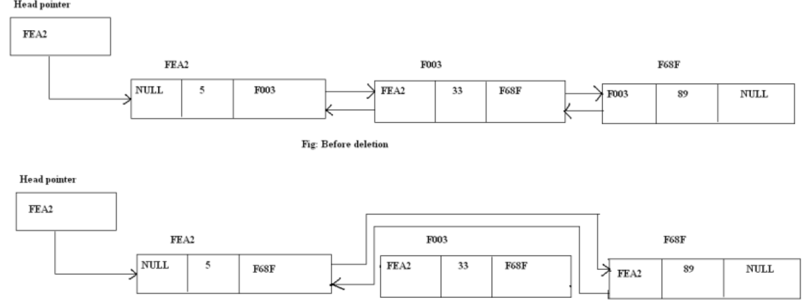
Fig: Deleting the last node from the list

c) Deleting an intermediate node from the list:

Algorithm for deleting a node at any intermediate position in the list

Delete_middle(Head:node)
Next, prev, last, delnode: node
Deldata: data of the node is the list to delete
Step 1: if(head==NULL)
 Print " list is empty"
 Return
 [End of if structure]
Step 2: print " Enter the data of the any node in the list for delete"
Step 3: read del data
Step 4: if(head->data==Deldata) then
 Set delnode=head
 Assign head=head->Flink
 If(head!=NULL
 Assign head->Blink=NULL

```
          [ End of  if structure ]
         Print " deleted data is ", delnode->data
         Call releasenode(delnode)
         Return
        [ End of if structure]
Step 5: set last=head->Flink
Step 6: repeat while(last!=NULL)
Step 7: if(last->data==deldata) then
           Set  delnode=last
         Set prev=last->Blink
Set    next=last->Flink
Assign  prev->Flink=Next
  If(Next!=NULL)
 Assign  next->Blink=prev
[ End of if structure]
Print "  The deleted data is ", delnode->data
Call  releasenode(delnode)
Return
Else
Assign  last=last->Flink
[ End of if structure ]
Step 8:  [ End of step 6 while loop ]
Step 9: print  " deldata is not available in the list"
Step 10: return
           End delete_middle()
```



5) Traversal of a list: To read the information or to display the information in a linked list, we have to traverse(move) a linked list node by node from the first node until the end of the list is reached.

Algorithm for displaying the contents of the list

```
View_list(head: node)
Step 1: if( head==NULL)
            Print  " list is empty "
            Return
             [ End of  if structure ]
Step 2: while(head!=NULL)
Step 3: print  " The data is ", head->data
Step 4: head=head->Flink
Step 5: [ End of step2 while structure ]
          End  view_list()
```

6) Count the number of nodes in the list: To count the nodes in a linked list , we have to traverse (move) in a linked list, made by node from the first node, until the end of the list is reached.

Algorithm for counting the number of nodes in the list

```
Count_list(head:node)
Count:integer
Step 1: set count=0
Step 2: if(head==NULL)
          Print  " List is empty"
          Return count
          [ End of  if structure ]
Step 3: while(head!=NULL)
Step 4: count=count+1
Step 5: head=head->Flink
Step 6: [ End of step2 while structure ]
Step 7: return count
         End Count_list()
```

Example: Write a C program on doubly linked list

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   struct node *prev;
   struct node *next;
   int data;
};
struct node *head;
void insertion_beginning();
void insertion_last();
void insertion_specified();
void deletion_beginning();
void deletion_last();
void deletion_specified();
void display();
void search();
```

```
void main ()
{
int choice =0;
   while(choice != 9)
   {
      printf("\n*********Main Menu*********\n");
      printf("\nChoose one option from the following list
...\n");

printf("\n===============================================
===========\n");
         printf("\n1.Insert in begining\n2.Insert at
last\n3.Insert at any random location\n4.Delete from
Beginning\n5.Delete from last\n6.Delete the node after the
given data\n7.Search\n8.Show\n9.Exit\n");
         printf("\nEnter your choice?\n");
      scanf("\n%d",&choice);
      switch(choice)
      {
         case 1:
         insertion_beginning();
         break;
         case 2:
               insertion_last();
         break;
         case 3:
         insertion_specified();
         break;
         case 4:
         deletion_beginning();
         break;
         case 5:
         deletion_last();
```

```
            break;
            case 6:
            deletion_specified();
            break;
            case 7:
            search();
            break;
            case 8:
            display();
            break;
            case 9:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
}
void insertion_beginning()
{
   struct node *ptr;
   int item;
   ptr = (struct node *)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
    printf("\nEnter Item value");
    scanf("%d",&item);

   if(head==NULL)
```

```
  {
     ptr->next = NULL;
     ptr->prev=NULL;
     ptr->data=item;
     head=ptr;
  }
  else
  {
     ptr->data=item;
     ptr->prev=NULL;
     ptr->next = head;
     head->prev=ptr;
     head=ptr;
  }
  printf("\nNode inserted\n");
}

}
void insertion_last()
{
  struct node *ptr,*temp;
  int item;
  ptr = (struct node *) malloc(sizeof(struct node));
  if(ptr == NULL)
  {
     printf("\nOVERFLOW");
  }
  else
  {
     printf("\nEnter value");
     scanf("%d",&item);
      ptr->data=item;
     if(head == NULL)
```

```
      {
          ptr->next = NULL;
          ptr->prev = NULL;
          head = ptr;
      }
      else
      {
         temp = head;
         while(temp->next!=NULL)
         {
             temp = temp->next;
         }
         temp->next = ptr;
         ptr ->prev=temp;
         ptr->next = NULL;
         }

      }
    printf("\nnode inserted\n");
    }
void insertion_specified()
{
   struct node *ptr,*temp;
   int item,loc,i;
   ptr = (struct node *)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\n OVERFLOW");
   }
   else
   {
      temp=head;
      printf("Enter the location");
```

```
        scanf("%d",&loc);
        for(i=0;i<loc;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                printf("\n There are less than %d elements", loc);
                return;
            }
        }
        printf("Enter value");
        scanf("%d",&item);
        ptr->data = item;
        ptr->next = temp->next;
        ptr -> prev = temp;
        temp->next = ptr;
        temp->next->prev=ptr;
        printf("\nnode inserted\n");
    }
}
void deletion_beginning()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\n UNDERFLOW");
    }
    else if(head->next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }
```

```
    else
    {
        ptr = head;
        head = head -> next;
        head -> prev = NULL;
        free(ptr);
        printf("\nnode deleted\n");
    }

}
void deletion_last()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\n UNDERFLOW");
    }
    else if(head->next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }
    else
    {
        ptr = head;
        if(ptr->next != NULL)
        {
            ptr = ptr -> next;
        }
        ptr -> prev -> next = NULL;
        free(ptr);
        printf("\nnode deleted\n");
```

```
    }
}
void deletion_specified()
{
    struct node *ptr, *temp;
    int val;
    printf("\n Enter the data after which the node is to be
deleted : ");
    scanf("%d", &val);
    ptr = head;
    while(ptr -> data != val)
    ptr = ptr -> next;
    if(ptr -> next == NULL)
    {
        printf("\nCan't delete\n");
    }
    else if(ptr -> next -> next == NULL)
    {
        ptr ->next = NULL;
    }
    else
    {
        temp = ptr -> next;
        ptr -> next = temp -> next;
        temp -> next -> prev = ptr;
        free(temp);
        printf("\nnode deleted\n");
    }
}
void display()
{
    struct node *ptr;
    printf("\n printing values...\n");
```

```
    ptr = head;
    while(ptr != NULL)
    {
        printf("%d\n",ptr->data);
        ptr=ptr->next;
    }
}
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("\nitem found at location %d ",i+1);
                flag=0;
                break;
            }
            else
            {
                flag=1;
            }
            i++;
```

```
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("\nItem not found\n");
        }
    }

}
```

2.2.3 Circular linked list

The basic operations that can be performed on circular singly linked list are similar to the singly linked list, except that the last node is made to point the first node in the list. The basic operations that can be performed on circular singly linked lists are

1) Creation of a list

2) Insertion of a node

3) Deletion of a node

4) Traversal of a node

5) Count the number of nodes

1) Creation of a list: Creation of list involves three process.

a) Creating a node

b) Reading details for a node from user

c) Connect the node with the list

Algorithm for declaration of the structure node
Struct node
Data:data field
Link:link field
End struct

Algorithm for allocating memory for the new node
Getnode()
Size: integer; newnode:node
Step 1: set size=get the size of the node
Step 2: set newnode=allocate space in memory for the size of size and return the initial address
Step 3: return newnode
End Getnode()

Algorithm for reading the content for the new node
Readnode(newnode:node)
Step 1:Read , newnode->data
Step 2: set newnode->link=newnode
Step 3: return
End Readnode()

Algorithm for create list( )
Createlist()
Head, last newnode:node
Step 1: Assign newnode=Getnode()
Step 2: Call readnode(newnode)
Step 3: set head=Newnode
Step 4: set last=newnode
Step 5: if we want to add another node proced otherwise return head

```
Step 6: set newnode=Getnode()
Step 7: call readnode(newnode)
Step 8: assign last-> link=newnode
Step 9: assing  newnode->link=head
Step 10: assign  last=last->link
Step 11: goto step 5
End createlist()
```

2) Insertion of a node: The new node can then be inserted in the list at three different places namely.

a) Inserting as a first node in the list

b) Inserting as a last node in the list

c) Inserting as an intermediate node in the list

a) Inserting as a first node in the list:

Algorithm for inserting a node as the first node in the list

```
Inser_first(head:node)
Newnode, last:node
Step 1: set newnode=getnode()
Step 2: call readnode(newnode)
Step 3: if(head==NULL)
          Set head=newnode
          Return
         [ End of if structure]
Step 4: set last=head
Step 5: Repeat while(last->link!=head)
          Assign last=last->link
          [ End of while structure ]
Step 6: assign last->link=newnode
Step 7: assign newnode->link=head
```

Step 8: assign head=newnode
End Insert_first()

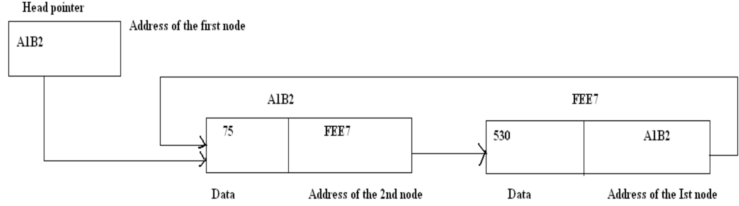


Fig: Before Insertion

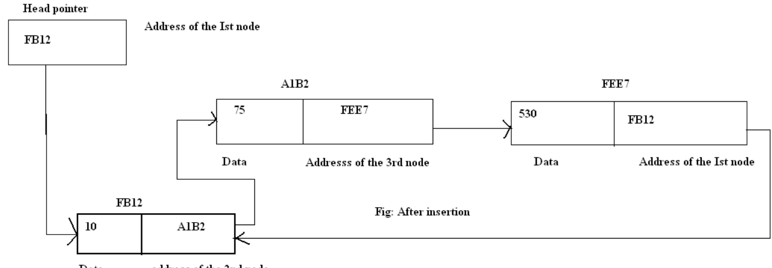


Fig: Inserting as a first node in the list

b) Inserting as a last node in the list:

Algorithm for inserting a node as the last node in the list

```
Insert_last(head: node)
Last, newnode:node
Step 1: set newnode=getnode()
Step 2: call  readnode(newnode)
Step 3: if(head==NULL)
        Set head=newnode
        Return
        [ End of if structure]
Step 4: set last=head
Step 5:Repeat while(last->link!=head)
```

Assign last=last->link
[End of while structure]
Step 6: Assign last->link=newnode
Step 7: Assign newnode->link=head
End Insert_last()

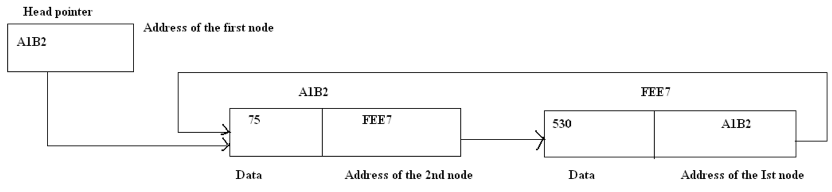


Fig: Before Insertion

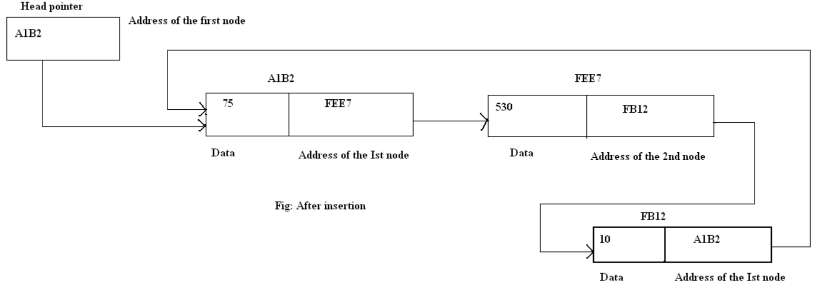


Fig: After insertion

Fig: Inserting as a last node in the list

c) Inserting an intermediate node in the list:

Algorithm for inserting a node at any intermediate position in the list

Insert_middle(head:node)
Last, newnode:node
Condition: data of the any one node in the list for insert
Step 1: set newnode=getnode()
Step 2: call readnode(newnode)

```
Step 3: if(head==NULL)
          Set head=newnode
          Return
         [ End of  if structure]
Step 4: print " Enter the data of node after which the
insertion is to be made"
Step 5: read condition
Step 6: set last=head
Step 7: repeat
Step 8: if(last->data==condition) then
          Assign newnode->link=last->link
         Assign last->link=newnode
         Return
         Else
         Assign last=last->link
         [ End of if structure]
Step 9: until(last==head)
Step 10: print  " condition is not available"
 End Insert_middle()
```

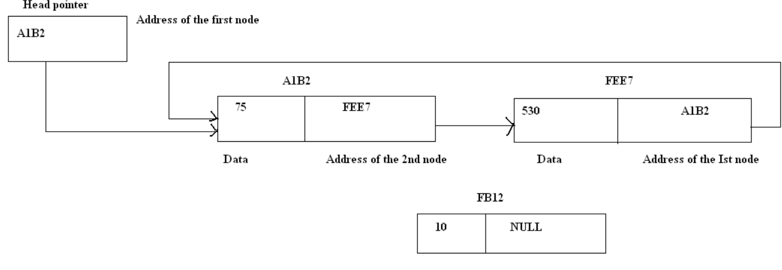


Fig: Before Insertion

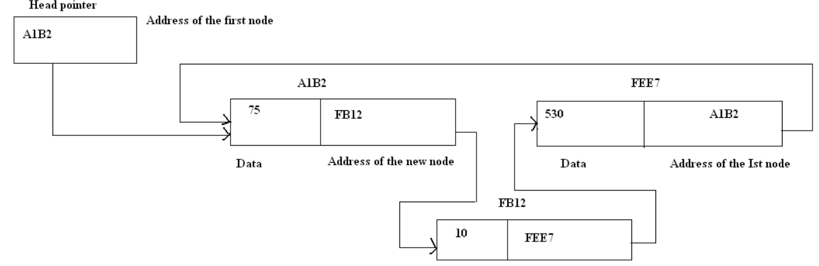


Fig: Inserting as an intermediate node in the list

3) Deletion of a node: A node can be deleted from the list from three different places namely

a) Deleting the first node from the list

b) Deleting the last node from the list

c) Deleting an intermediate node from the list

a) Deleting the first node from the list:

Algorithm for deleting the first node from the list

```
 Delete_first(head:node)
 Last, prev, delnode:node
Step 1: if(head==NULL)
            Print  " List is empty"
            Return
           [ End of if structure ]
Step 2: if(head->link==head) then
            Delnode=head
            Print " Deleted data is", delnode->data
           Set head=NULL
           Call releasenode(delnode)
           Return
          [ End of if structure ]
Step 3: set last=head
```

Step 4: repeat
    Assign prev=last
    Assign last=last->link
    Until(last->link==head)
Step 5: delnode=last
Step 6: prev->link=head
Step 7: print " Deleted data is", delnode->data
Step 8: call releasenode(delnode)
End Delete_first()

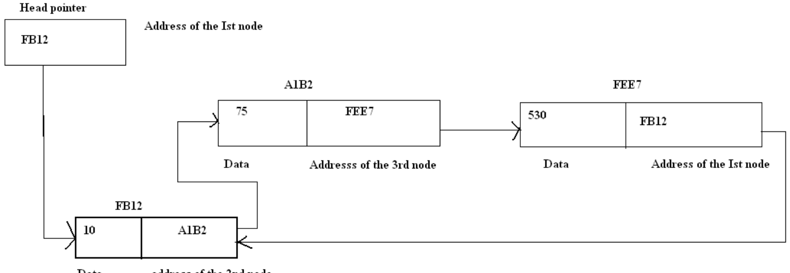


Fig: Before deletion

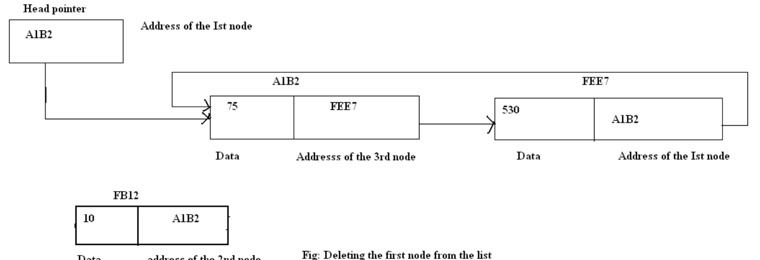


Fig: Deleting the first node from the list

b) Deleting the last node from the list:

```
        Algorithm for deleting the last node from the list
Delete_last(head:node)
Last, prev, delnode:node
Step 1: if( head==NULL)
        Print " List is empty"
        Return
       [ End of if structure ]
Step 2: if(head->link==head) then
        Set delnode=head
        Set head=NULL
        Print " Deleted data is", delnode->data
        Call releasenode(delnode)
         Return
       [ End of if structure ]
Step 3: set last=head
Step 4: Repeat while( last->link!=head)
Step 5: assign prev=last
Step 6: assign last=last->link
Step 7: [ End of step4 while loop ]
Step 8: set delnode=last
Step 9: prev->link=head
Step 10: print "Deleted data is", delnode->data
Step 11: call releasenode(delnode)
 End Delete_last()
```

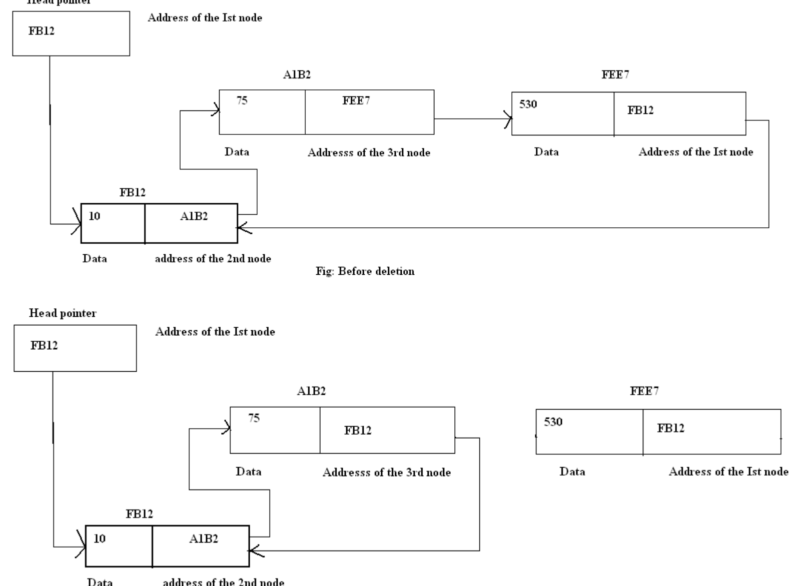


Fig: Before deletion

Fig: Deleting the last node from the list

c) Deleting an intermediate node from the list:

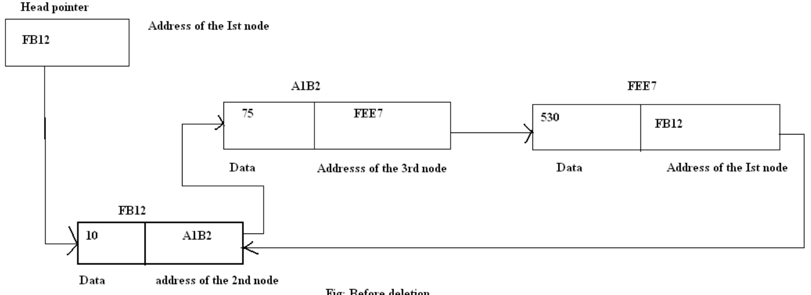


Fig: Before deletion

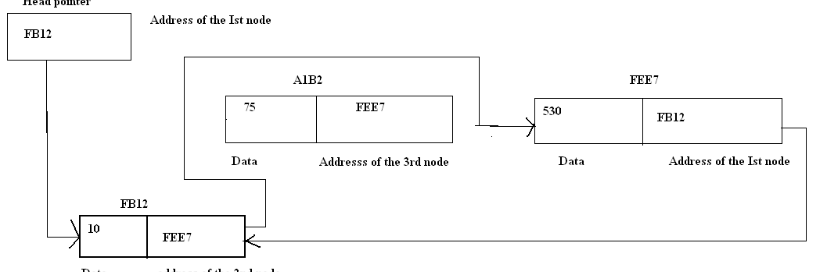


Fig: Deleting an intermediate node from the list

Example: Write a program on circular singly linked list

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *head;

void beginsert ();
void lastinsert ();
void randominsert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search();
void main ()
{
   int choice =0;
   while(choice != 7)
   {
```

```
        printf("\n*********Main Menu*********\n");
    printf("\nChoose one option from the following list ...\n");

printf("\n===============================\n");
        printf("\n1.Insert in begining\n2.Insert at last\n3.Delete
from Beginning\n4.Delete from last\n5.Search for an
element\n6.Show\n7.Exit\n");
        printf("\nEnter your choice?\n");
        scanf("\n%d",&choice);
        switch(choice)
        {
            case 1:
            beginsert();
            break;
            case 2:
            lastinsert();
            break;
            case 3:
            begin_delete();
            break;
            case 4:
            last_delete();
            break;
            case 5:
            search();
            break;
            case 6:
            display();
            break;
            case 7:
            exit(0);
            break;
            default:
```

```
            printf("Please enter valid choice..");
        }
    }
}
void beginsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter the node data?");
        scanf("%d",&item);
        ptr -> data = item;
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
            temp = head;
            while(temp->next != head)
                temp = temp->next;
            ptr->next = head;
            temp -> next = ptr;
            head = ptr;
        }
        printf("\nnode inserted\n");
```

```
    }

}
void lastinsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
    }
    else
    {
        printf("\nEnter Data?");
        scanf("%d",&item);
        ptr->data = item;
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
            temp = head;
            while(temp -> next != head)
            {
                temp = temp -> next;
            }
            temp -> next = ptr;
            ptr -> next = head;
        }
```

```
        printf("\nnode inserted\n");
    }

}

void begin_delete()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\nUNDERFLOW");
    }
    else if(head->next == head)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }

    else
    {   ptr = head;
        while(ptr -> next != head)
            ptr = ptr -> next;
        ptr->next = head->next;
        free(head);
        head = ptr->next;
        printf("\nnode deleted\n");

    }
}
void last_delete()
{
    struct node *ptr, *preptr;
```

```
    if(head==NULL)
    {
        printf("\nUNDERFLOW");
    }
    else if (head ->next == head)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");

    }
    else
    {
        ptr = head;
        while(ptr ->next != head)
        {
            preptr=ptr;
            ptr = ptr->next;
        }
        preptr->next = ptr -> next;
        free(ptr);
        printf("\nnode deleted\n");

    }
}

void search()
{
    struct node *ptr;
    int item,i=0,flag=1;
    ptr = head;
    if(ptr == NULL)
    {
```

```
    printf("\nEmpty List\n");
  }
  else
  {
    printf("\nEnter item which you want to search?\n");
    scanf("%d",&item);
    if(head ->data == item)
    {
    printf("item found at location %d",i+1);
    flag=0;
    }
    else
    {
    while (ptr->next != head)
    {
      if(ptr->data == item)
      {
        printf("item found at location %d ",i+1);
        flag=0;
        break;
      }
      else
      {
        flag=1;
      }
      i++;
      ptr = ptr -> next;
    }
    }
    if(flag != 0)
    {
      printf("Item not found\n");
    }
```

```
    }

}

void display()
{
    struct node *ptr;
    ptr=head;
    if(head == NULL)
    {
        printf("\nnothing to print");
    }
    else
    {
        printf("\n printing values ... \n");

        while(ptr -> next != head)
        {

            printf("%d\n", ptr -> data);
            ptr = ptr -> next;
        }
        printf("%d\n", ptr -> data);
    }

}
```

2.3 Comparing arrays and linked lists

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

2.4 Applications of linked lists

- Linked lists can be used to represent polynomials.
- Using a linked list, we can perform the polynomial manipulation.
- Arithmetic operations like addition or subtraction of long integers can also be performed using a linked list.
- The linked list can be used to implement stacks and queues.
- The linked list is also used in implementing graphs in which the adjacent vertices are stored in the nodes of the linked list popularly known as Adjacency list representation.

Applications of Linked Lists in the Real World :

- In music players, we can create our song playlist and can play a song either from starting or ending of the list. And these music players are implemented using a linked list.
- We watch the photos on our laptops or PCs, and we can simply see the next or previous images easily. This feature is implemented using a linked list.
- You must be reading this article on your web browser, and in web browsers, we open multiple URLs, and we can easily switch between those URLs using the previous and next buttons because they are connected using a linked list.

Some Other Applications of Linked List

- Allocation of Memory
- Email applications
- Reducing file sizes on disk
- Implementation of advanced data structures
- Advantages of Linked List Over Arrays
- A few advantages of linked lists over arrays are :
- Dynamic size
- Efficient implementation of data structures

- No memory wastage
- Efficient insertion and deletion operation

1. Polynomial Manipulation

Polynomials are algebraic expressions that contain coefficients and variables. Polynomial manipulation is doing mathematical operations, like addition, subtraction, etc., on polynomials.

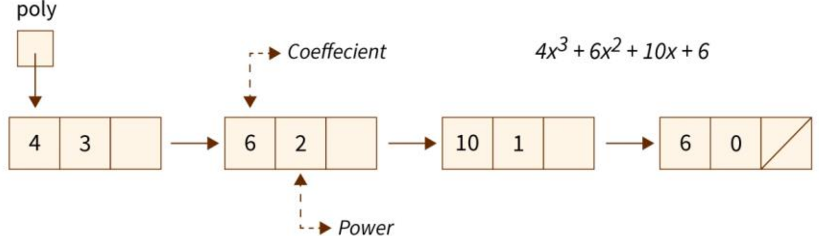
Polynomials are a very important part of mathematics, and there aren't any direct data structures present that can be used to store polynomials in memory. Thus, we take the help of a linked list to represent a polynomial.

To represent the polynomials using a linked list, we assume that each node of the linked list corresponds to each term of the polynomials.

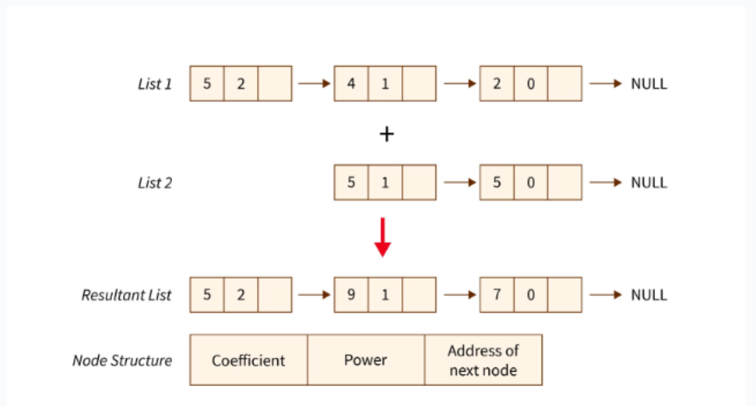
Let us see how a polynomial is represented in a linked list.

The node of the linked list contains three parts :

- The coefficient value
- The exponent value, and
- The link to the next term.

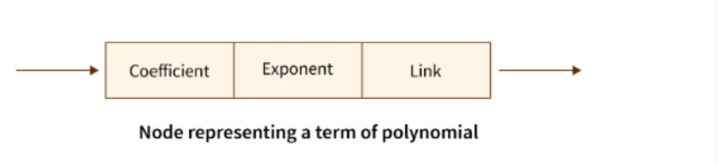


For example the polynomial $4x^3 + 6x^2 + 10x + 6$ can be represented as



To add two polynomials, we first represent both of them in the form of a linked list. And then, we add the coefficients having the same exponent.

For example, suppose we want to add two polynomials that are : $5x^3 + 4x^2 + 2x^0$ and $5x^1 - 5x^0$.



Node representing a term of polynomial

```
#include <bits/stdc++.h>
using namespace std;

// structure of a node
struct Node{
        int co_eff;
        int pwr;
        struct Node* nxt;
};

// add new node
void make_newnode(int x, int y,
                                        struct Node** temp){
        struct Node *r, *z;
        z = *temp;
        if (z == NULL){
        r = (struct Node*)malloc(sizeof(struct Node));
                r->co_eff = x;
                r->pwr = y;
                *temp = r;
        r->nxt = (struct Node*)malloc(sizeof(struct Node));
                r = r->nxt;
                r->nxt = NULL;
        }
        else{
                r->co_eff = x;
                r->pwr = y;
        r->nxt = (struct Node*)malloc(sizeof(struct Node));
                r = r->nxt;
                r->nxt = NULL;
        }
}
```

```
// add two polynomials
void add_poly(struct Node* first_poly,
                              struct Node* sec_poly,
                              struct Node* poly){
   // if the degree of the first polynomial is
   // greater than the second polynomial then
   // do not change the first polynomial and move
   // its pointer

         while (first_poly->nxt &&
                   sec_poly->nxt){

                   if (first_poly->pwr > sec_poly->pwr){
                              poly->pwr = first_poly->pwr;
                              poly->co_eff = first_poly->co_eff;
                              first_poly = first_poly->nxt;
                   }
      // if the degree of the second polynomial is
   // greater than the first polynomial then
   // do not change the second polynomial and move
   // its pointer

                   else if (first_poly->pwr < sec_poly->pwr){
                              poly->pwr = sec_poly->pwr;
                              poly->co_eff = sec_poly->co_eff;
                              sec_poly = sec_poly->nxt;
                   }

      // if the degree of both polynomials is
      // same then add the coefficients

                   else{
                              poly->pwr = first_poly->pwr;
```

```
                        poly->co_eff = (first_poly->co_eff +
                                            sec_poly-
>co_eff);
                        first_poly = first_poly->nxt;
                        sec_poly = sec_poly->nxt;
                }

poly->nxt =struct Node*)malloc(sizeof(struct Node));
                poly = poly->nxt;
                poly->nxt = NULL;
        }
        while (first_poly->nxt || sec_poly->nxt){
                if (first_poly->nxt){
                        poly->pwr = first_poly->pwr;
                        poly->co_eff = first_poly->co_eff;
                        first_poly = first_poly->nxt;
                }
                if (sec_poly->nxt){
                        poly->pwr = sec_poly->pwr;
                        poly->co_eff = sec_poly->co_eff;
                        sec_poly = sec_poly->nxt;
                }
                poly->nxt =
                        (struct     Node*)malloc(sizeof(struct
Node));
                poly = poly->nxt;
                poly->nxt = NULL;
        }
}
// print the linked list
void display_poly(struct Node* node){
        while (node->nxt != NULL){
                printf("%dx^%d", node->co_eff,
```

```
                    node->pwr);
            node = node->nxt;
            if (node->co_eff >= 0){
                    if (node->nxt != NULL)
                            printf("+");
            }
    }
    printf("\n");
}
// main function
int main(){
    struct Node *first_poly = NULL,
                            *sec_poly = NULL,
                            *poly = NULL;

    make_newnode(5, 2, &first_poly);
    make_newnode(4, 1, &first_poly);
    make_newnode(2, 0, &first_poly);


    make_newnode(-5, 1, &sec_poly);
    make_newnode(-5, 0, &sec_poly);

    printf("1st Number: ");
    display_poly(first_poly);

    printf("2nd Number: ");
    display_poly(sec_poly);

    poly = (struct Node*)malloc(sizeof(struct Node));

    add_poly(first_poly, sec_poly, poly);
```

```
	printf("Added polynomial: ");
	display_poly(poly);

	return 0;
}
```

Output

```
1st Number: 5x^2 +4x^1 + 2x^0
2nd Number:  -5x^1 - 5x^0
Added polynomial: 5x^2 - 1x^1 -3x^0
```

UNIT-III

Stacks: Introduction, properties and operations, implementing stacks using arrays and linked lists, Applications of stacks in expression evaluation, balanced parentheses, reversing list etc.

Queues: Introduction, properties and operations, implementing queues using arrays and linked lists, Queue applications in OS and simulation experiments.

Types of Queues: Types - Circular Queues, Priority Queues, Deques, and supporting operations.

3.1 Stack: A stack is an ordered collection of elements in which insertions and deletions are restricted to one end. The end from which elements are added/or removed is referred to as **top** of the stack.

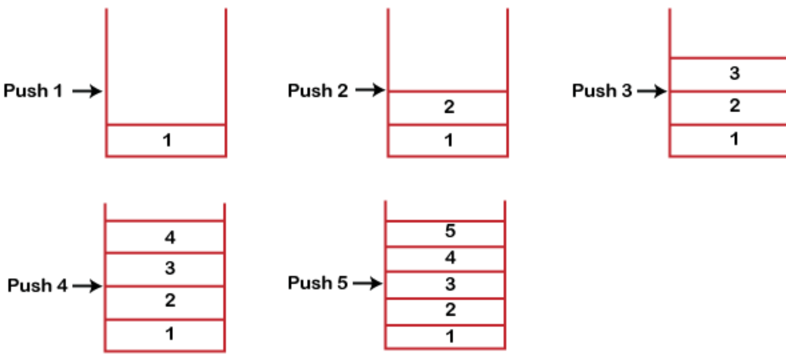
Stacks are also referred as **"piles"** and "**push-down lists".** The first element placed in the stack will be at the bottom of the stack. The last element added to stack is the first element to be removed. Hence stacks are referred to as **Last-In-First-Out (LIFO).**

Note: A stack is a non-primitive linear data structure.

Working of Stack

Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top. When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value

entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

3.2 Properties of Stack

There are following important properties of stack-

(i) All insertions and deletions can occur only at the top of stack.

(ii) The elements that are removed from stack in reverse order in which they were inserted.

(iii) Only one element can be pushed or popped from the stack at a time.

(iv) It works in last-in-first-out or LIFO manner.

3.3 Operations on stack: The basic operations that can be performed on stack are as follows

1. Create 2.Push 3.Pop 4.peek 5.Empty stack 6.Fully occupied stack.

1. Create: This operation creates a stack, and leaves it empty.

2. Push operation: Push is an operation used to add a new element in to a stack.

3. Pop operation: This operation deletes an item only from the top of the stack when stack is not empty.

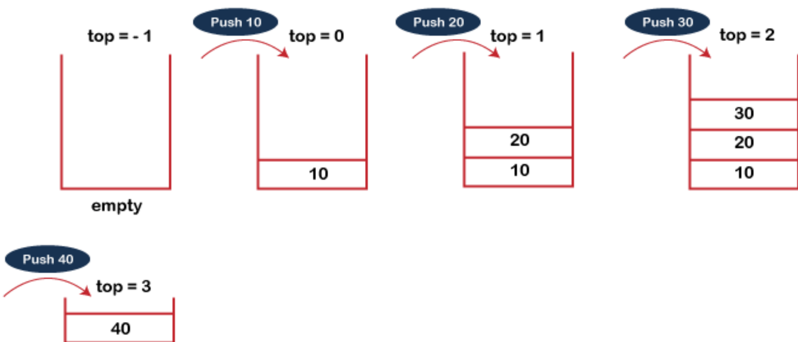
4. Peek operation: Peek is an operation used to display the element from the top of the stack.

5. Is Empty: This operation checks whether the stack is empty or not. It returns true if stack is empty, otherwise returns false.

6. Is Full: This operation checks whether the stack is full or not. It returns true if stack is full otherwise it returns false.

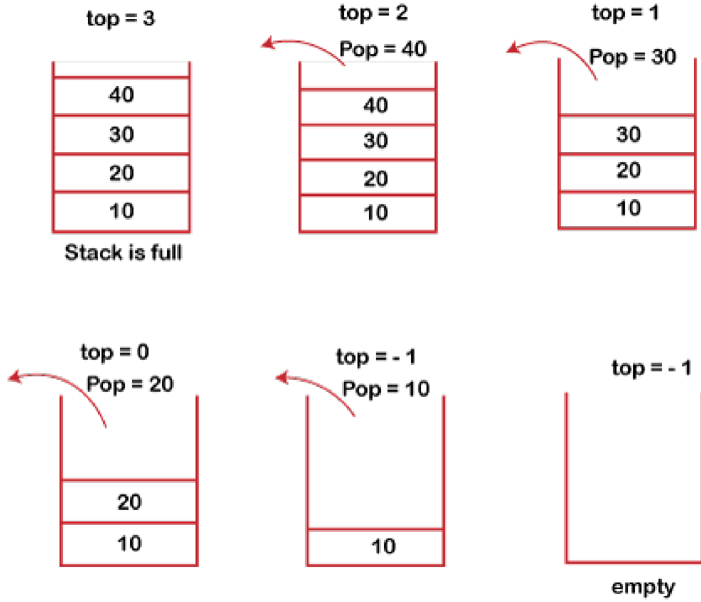
Algorithm for PUSH operation

```
[Check for the stack is full or not]
Step 1:if(top==SIZE-1)
        {
          printf("\n stack is full");
          return;
        }
Step 2:[Increment the top value by one]
        top=top+1;

Step 3:Insert the elements into the stack
       stack[top]=x;
```



Algorithm for POP operation

```
 [Check for the stack is empty or not]
Step 1:if(top==-1)
        {
          printf("\n stack is empty");
          return;
        }
Step 2:The elements into the stack:
        x=stack[top];
Step 3: top=top-1;
```



Algorithm for PEEK operation

```
[Check for the stack is empty or not]
Step 1:if(top==-1)
        {
         printf("\n stack is empty");
         return;
        }
Step 2:for(i=top; i>=0;i--)
       printf ("->%d", stack[i]);
```

3.4 Implementing stacks using arrays and linked lists

Example 1: Write a C program to perform stack operations using arrays

```
#include<stdio.h>
#include<conio.h>
#define SIZE  10
int stack[SIZE], top=-1,i;
void push();
void pop();
void display();
void main()
{
int  ch;
clrscr();
do
{
printf("\n 1.Push");
printf("\n 2.Pop");
printf("\n 3.Display");
printf("\n 4.Exit");
printf("\n Enter the choice");
scanf("%d",&ch);
switch(ch)
{
case 1:push();
       break;
case 2:pop();
       break;
case 3:display();
       break;
case 4:exit(0);
       break;
```

```
default: printf("\n Invalid choice");
}
}
while(ch!=4);
getch();
}


void  push()
{
if(top==SIZE-1)
 {
  printf("\n stack is full");
  return;
 }
  top=top+1;
  printf("\n Enter the elements");
  scanf("%d", &stack[top]);
}

void  pop( )
{
 if(top==-1)
 {
  printf("\n Stack is empty");
  return;
 }
printf("\n The Pop element is %d", stack[top]);
top=top-1;
}

void  display()
{
```

```
 if(top==-1)
 {
  printf("\n Stack is empty");
  return;
 }
printf("\n The elements in the stack is");
for(i=top; i>=0; i--)
printf("->%d", stack[i]);
}
```

Example 2: Write a C program to perform stack operations using linked list

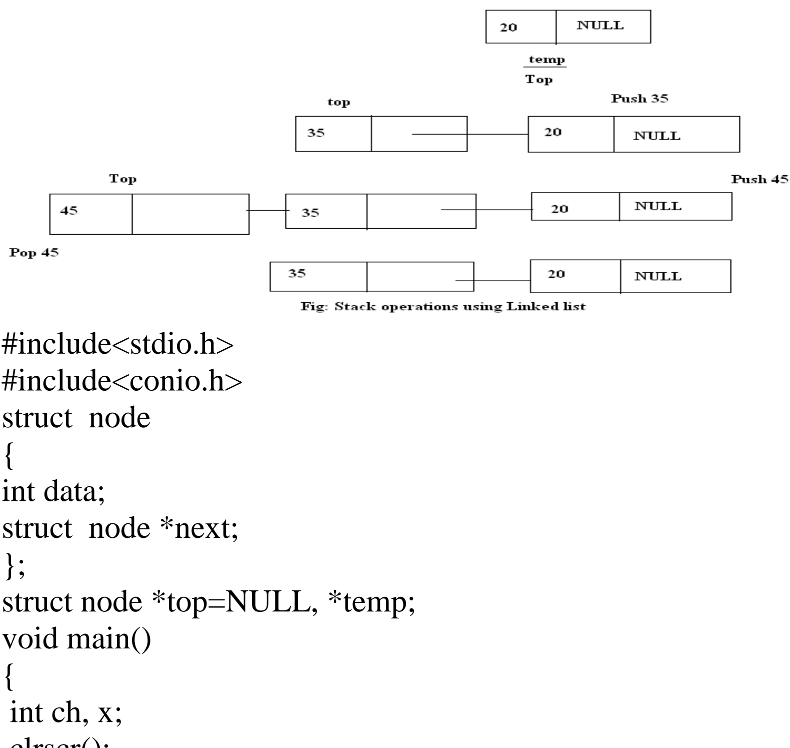


Fig: Stack operations using Linked list

```
#include<stdio.h>
#include<conio.h>
struct  node
{
int data;
struct  node *next;
};
struct node *top=NULL, *temp;
void main()
{
 int ch, x;
 clrscr();
```

```
while(1)
{
 printf("\n 1.Push");
 printf("\n 2.Pop");
 printf("\n 3.Display");
 printf("\n 4.Exit");
 printf("\n Enter the choice");
 scanf("%d",&ch);
 switch(ch)
 {
  case 1:temp=(struct  node *)malloc(sizeof(struct  node);
        printf("\n Enter the element");
        scanf("%d", &x);
        temp->data=x;
        temp->next=top;
        top=temp;
        break;
 case 2:if(top!=NULL)
        {
          printf("\n The pop element is %d", top->data);
          top=top->next;
        }
         else
          {
            printf("\n Stack is empty");
            return;
          }
         break;
  case 3: temp=top;
         if(temp==NULL)
          {
           printf("\n Stack is empty");
           return;
```

```
        }
        while(temp!=NULL)
         {
          printf("->%d", temp->data);
          temp=temp->next;
         }
        break;
case 4:exit(0);
      break;
    }
  }
 }
```

Difference between Arrays and Stack

SR	Array	Stack
i	We can insert an element at any location in an array.	A new element can be added only at the top of a stack.
ii	Any element of an array can be accessed.	Only the topmost element can be accessed.
iii	An array essentially contains homogenous elements.	A stack may contain diverse elements.
iv	An array is a static data structure i.e. Its size remains constant.	A stack is a dynamic data structure i.e. its size grows and shrinks as elements are pushed and popped.
v	There is an upper limit on the size of the array, which is specified during declaration.	Logically, a stack can grow to any size.

Advantages of Stack

- A Stack helps to manage the data in the 'Last in First out' method.
- When the variable is not used outside the function in any program, the Stack can be used.

- It allows you to control and handle memory allocation and deallocation.
- It helps to automatically clean up the objects.

Disadvantages of Stack

- It is difficult in Stack to create many objects as it increases the risk of the Stack overflow.
- It has very limited memory.
- In Stack, random access is not possible.

3.5 Applications of stacks

Some of the important applications of stacks include

1. Towers of Hanoi problem
2. Reversing a string
3. Recursion using stack
4. Evaluation of arithmetic expressions
5. Balanced Parenthesis

1. Towers of Hanoi problem

A tower of Hanoi is a game appeared in Europe in the 19th century. The French mathematician **Eduard Lucas** invented the towers of Hanoi puzzle in 1883. We are given a tower(tower1) of n disks initially stacked in increasing size

from top to bottom. We have two more towers, tower2 and tower3.

Fig: illustrates the initial setup of towers of Hanoi. The rules to be followed in moving the disks from tower1 to tower3 using tower2 as follows.

a). Only one disc can be moved at a time.

b). Only the top disc on any tower can be moved to any other tower.

c). A larger disc cannot be placed on a smaller disc.

Algorithm for TOH problem:

Let's consider move 'n' disks from source peg (A) to destination peg (C), using intermediate peg (B) as auxiliary.

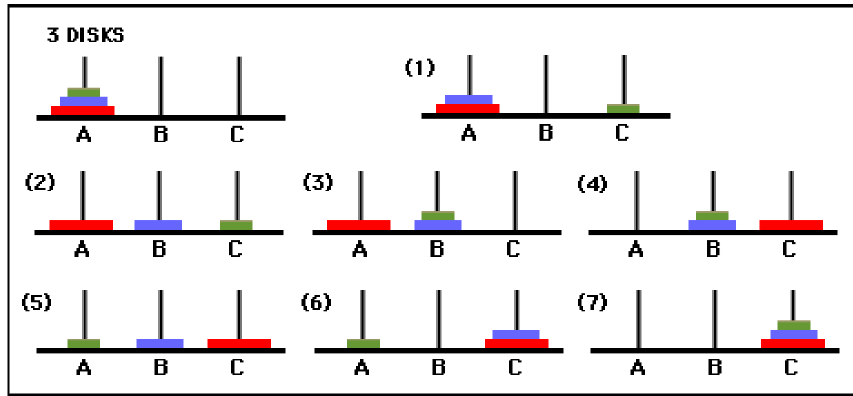
1. Assign three pegs A, B & C
2. If n==1

   Move the single disk from A to C and stop.
3. If n>1

   a) Move the top (n-1) disks from A to B.

   b) Move the remaining disks from A to C

   c) Move the (n-1) disks from B to C
4. Terminate



Since disks are moved from each tower in a LIFO manner, each tower may be considered as a stack. The least number of moves required to solve the problem according to our algorithm is given be

$O(N)= O(N-1) + 1+ O(N-1)$

$\quad =2^N-1$

If the time complexity is measured in no. of movements, i.e., $(O(2^N))$. The space complexity is O(N) due to the use of recursion.

Recursive solution of tower of Hanoi

```
#include <stdio.h>
#include <conio.h>
void TOH(int, char, char, char); //Function prototype
void main()
{
int n;
```

```
printf("Enter number of disks");
scanf("%d",&n);
TOH(n,'O','D','I');
getch();
}

void TOH(int n, char A, char B, char C)
{
if(n>0)

{
TOH(n-1, A, C, B);
Printf("Move disk %d from %c to%c\n", n, A, B);
TOH(n-1, C, B, A);
}
}
```

2. Reversing a string: One of the main characteristic of stack is reversing the order of its contents. This characteristic is exploited and used to reverse strings. This task can be accomplished by simply pushing each character until end of the string. Now the individual characters are popped off from the stack. Since the last character to be popped off from the stack. The string will come off the in the reverse order.

C Program To Reverse a String using Stack

```
#include <stdio.h>
#include <string.h>

#define max 100
int top,stack[max];

void push(char x){

      // Push(Inserting Element in stack) operation
      if(top == max-1){
          printf("stack overflow");
      }  else {
          stack[++top]=x;
      }

}

void pop(){
    // Pop (Removing element from stack)
      printf("%c",stack[top--]);
}


main()
{
   char str[]="sri lanka";
   int len = strlen(str);
   int i;

   for(i=0;i<len;i++)
        push(str[i]);
```

```
  for(i=0;i<len;i++)
     pop();
}
```

3. Recursion with stack: Recursion is a process of defining something in terms of itself. When a recursive program is executed the recursive function calls are not executed immediately. They are placed on a stack (LIFO) until the condition that terminates the recursive function. The function calls are then executed in reverse order, as they are poped off the stack.

Program to find factorial using stack

```
#include<stdio.h>
int stk[100]; // stack
int size = 100;    // size of stack
int ptr = -1; // store the index of top element of the stack
// push x to stack
void push(int x){
    if(ptr==size-1){
        printf("OverFlow \n");
    }
    else{
        ++ptr;
        stk[ptr] = x;
    }
}
// return top element of the stack
int top(){
```

```
    if(ptr==-1){
        printf("UnderFlow \n");
        return -1;
    }
    else{
        return stk[ptr];
    }
}
// remove top element from the stack
void pop(){

    if(ptr==-1){
        printf("UnderFlow \n");
    }
    else{
        --ptr;
    }
}
// check if stack is empty or not
int isempty(){
    if(ptr==-1)
        return 1;
    else
        return 0;
}
int main() {
    int i, n;

    printf("Enter a number: ");
    scanf("%d", &n);

    push(1);
    for(i=2;i<=n;++i){
```

```
        push(top() * i);
    }
    printf("Factorial: %d", top());
    return 0;
}
```

Output

Enter a number: 5
Factorial: 120

4. Evaluation of arithmetic expressions: An expression consist of two components namely operands and operators. Operators indicate the operation to be carried out on operands.

They are three ways of representing expressions in computers. They are

a). Infix notation b) Prefix notation c) Postfix notation

The table shows the different ways of representing an expression.

Notation	Arithmetic Expression
Infix	Operand Operator Operand
Prefix	Operator Operand Operand
Postfix	Operand Operand Operator

a) Infix notation: The normal way of expressing mathematical expressions is called as infix notation. In this form of expressing an arithmetic expression the operator comes in between its operands.

Example: (a + b)

Advantages of infix notations:

1. It is the mathematical way of representing the expression.
2. It is easier to see visually which operation is done from first to last.

b) Prefix notation: Prefix notation is also referred as polish notation. A polish mathematician Jan lukasiewicz introduced prefix notation. In this form of expressing an arithmetic expression the operator is written before its operands.

Example: (+ a b)

c) Postfix notation: Postfix notation also referred as suffix form or reverse polish notation (RPN). This was also introduced by Jan lukasiewicz. In this form of expressing an arithmetic expression the operator is written after its operands.

Example: (a b +)

Advantages of postfix notations:

1. We need not worry about the rules of precedence.

2. We need not worry about the rules for right to left associativity.

3. We need not need parenthesis to override the above rules.

Precedence of operators:

Priority	Operation(symbol)
1	Exponentiation (↑)
2	Multiplication (*), division(/)
3	Addition (+) , subtraction(-)

Rules to be followed during infix to postfix conversion:

1. Fully parenthesize the expression starting from left to right.

2. Move the operators one by one to their right, such that each operator replaces their corresponding right parenthesis.

3. The part of the expression, which has been converted into post fix, is to be treated as single operand.

4. Once the expression is converted into postfix form, remove all parentheses.

Example 1: Give the postfix from the infix expression x+ y*z

Sol:

x+(y*z)

x+(yz*)

x+A → [A= (yz*)]

(x+A)

(xA+)

(x(yz*)+)

xyz*+

Example 2: Give the postfix from the infix expression P+Q/R-S

Sol:

Character	Postfix	Stack
P	P	
+		+
Q	PQ	
/		+/

R	PQR/	+
-	PQR/+	-
S	PQR/+S	
	PQR/+S-	Stack is empty

Rules to be followed during infix to prefix conversion:

1. Fully parenthesize the expression starting from left to right.

2. Move the operators one by one to their left, such that each operator replaces their corresponding left parenthesis.

3. The part of the expression which has been converted into prefix. Is to be treated as single operand.

4. Once the expression is converted into prefix form remove all parentheses.

Example: Give the prefix form for the infix expression p/p * r+s

Sol:

(p/q)*r+s
(/pq)*r+s
A*r+s→ A=(/pq)
(A*r)+s
(*Ar)+s
B+s→ B=(*Ar)
(B+s)

(+Bs)
(+(*Ar)s)
(+(*(/pq)r)s)
+*/pqrs

Postfix evaluation:

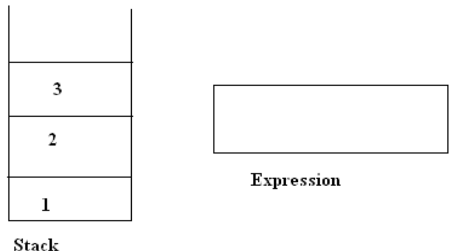
In normal algebra we use the infix notation like a+b*c. The corresponding postfix notation is abc*+. The algorithm for conversion is as follows.

1. Scan the postfix form left to right.
2. Initialize an empty stack.
3. If the scanned character is an operand, add it to the stack. If the scanned character is an operator, there will be at least two operands in the stack.
4. After all characters are scanned; we will have only one element in the stack. Return top stack.

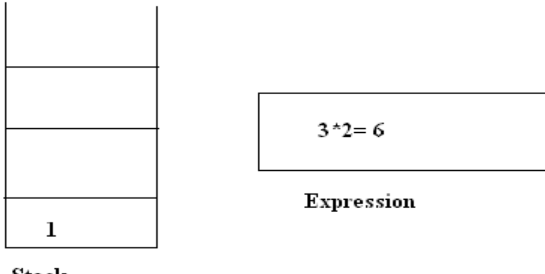
Example: Let us see how the above algorithm will be implemented using an example.

Postfix: 123* + 4 –

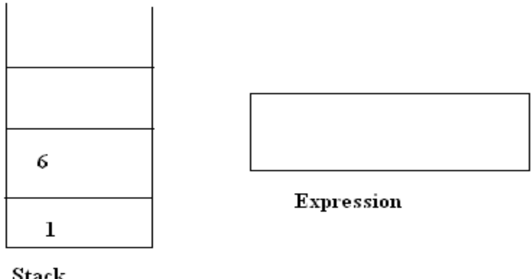
Initially the stack is empty. Now , the first three characters scanned are 1, 2 and 3. Which are operands. Thus they will be pushed into the stack in that order.



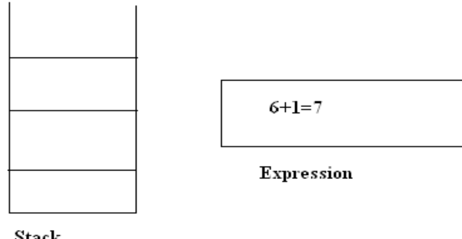
Next character scanned is “* ”which is an operator. Thus, we pop the top two elements from the stack and perform the “ * ” operation with the two operands. The second operand will be the first element that is popped.



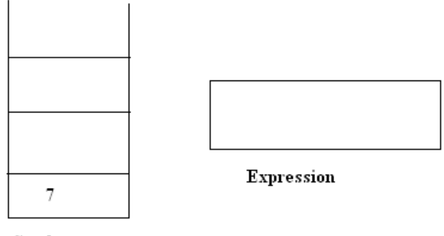
The value of the expression (3*2) that has been evaluated (6) is pushed into the stack.



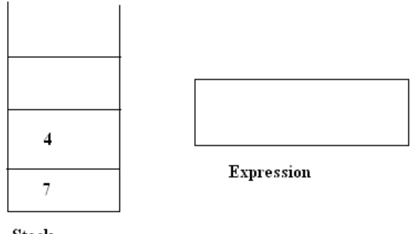
Next character is scanned is " + " which is an operator. Thus, we pop the top two elements from the stack and perform the " + " operation with two operands. The second operand will be the first element.



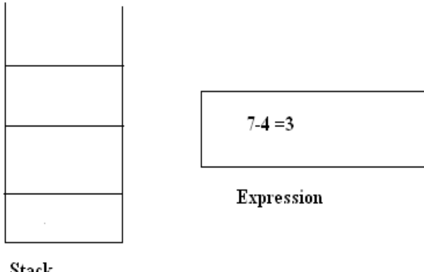
The value of the expression (6+1) that has been evaluated (7) is pushed into the stack.



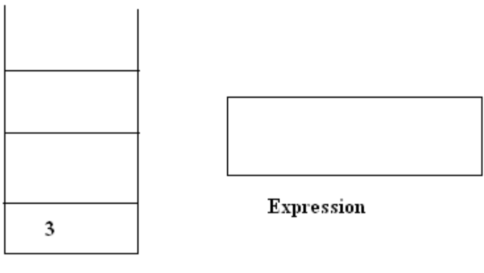
Next character is 4 which is added to the stack.



Next character is " – " which is an operator, so pop two elements.



The value of the expression (7-4) that has been evaluated (3) is pushed into the stack.



Now, since all characters are scanned, the remaining element in the stack is 3.

End result:

Postfix: 123 *+4-

Result: 3

Evaluation of postfix expression:

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack. When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage.

Example 1:

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed

8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

Example 2:

Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

Program to evaluate a postfix expression:

```
# include <conio.h>
# include <math.h>
# define MAX 20
int isoperator(char ch)
{
if(ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^')
return 1;
else
return 0;
}

void main(void)
{
char postfix[MAX];
int val;
char ch;
int i = 0, top = 0;
float val_stack[MAX], val1, val2, res;
clrscr();
printf("\n Enter a postfix expression: ");
scanf("%s", postfix);
while((ch = postfix[i]) != '\0')
{
if(isoperator(ch) == 1)
{
val2 = val_stack[--top];
val1 = val_stack[--top];
switch(ch)
{
case '+':
res = val1 + val2;
break;
```

```
case '-':
res = val1 - val2;
break;
case '*':
res = val1 * val2;
break;
case '/':
res = val1 / val2;
break;
case '^':
res = pow(val1, val2);
break;
}
val_stack[top] = res;
}
else
val_stack[top] = ch-48; /*convert character digit to integer
digit */
top++;
i++;
}
printf("\n Values of %s is : %f ",postfix, val_stack[0] );
getch();
}
```

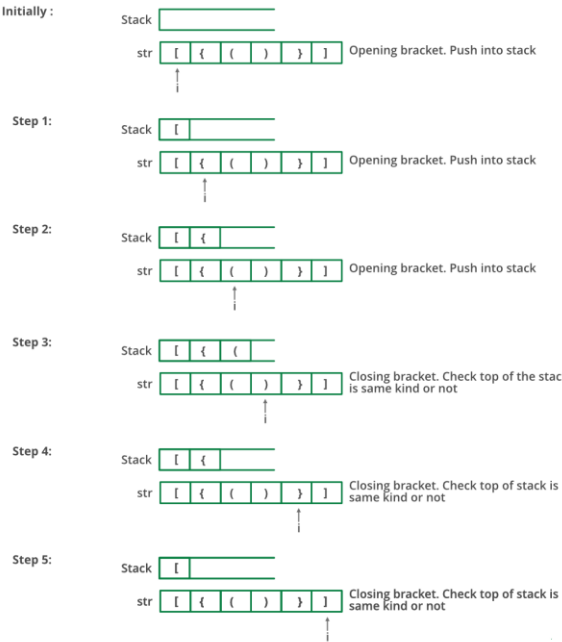
5. Balanced Parenthesis

Follow the steps mentioned below to implement the idea:

- Declare a character stack (say temp).
- Now traverse the string exp.

- If the current character is a starting bracket (‘(‘ or ‘{‘ or ‘[‘) then push it to stack.
- If the current character is a closing bracket (‘)’ or ‘}’ or ‘]’) then pop from the stack and if the popped character is the matching starting bracket then fine.
- Else brackets are Not Balanced.

- After complete traversal, if some starting brackets are left in the stack then the expression is Not balanced, else Balanced.



```
}

#include <stdio.h>
#include <stdlib.h>
#define MAX 100

struct stack {
  char stck[MAX];
  int top;
}s;

void push(char item) {
  if (s.top == (MAX - 1))
    printf("Stack is Full\n");

  else {
    s.top = s.top + 1;
    s.stck[s.top] = item;
  }
}

void pop() {
  if (s.top == -1)
    printf("Stack is Empty\n");

  else
    s.top = s.top - 1;
}

int checkPair(char val1,char val2){
    return (( val1=='(' && val2==')' )||( val1=='[' &&
val2==']' )||( val1=='{' && val2=='}' ));
}

int checkBalanced(char expr[], int len){
```

```
  for (int i = 0; i < len; i++)
  {
     if (expr[i] == '(' || expr[i] == '[' || expr[i] == '{')
     {
      push(expr[i]);
     }
     else
     {
        // exp = {{}}}
        // if you look closely above {{}} will be matched
with pair, Thus, stack "Empty"
        //but an extra closing parenthesis like '}' will never
be matched
        //so there is no point looking forward
     if (s.top == -1)
        return 0;
     else if(checkPair(s.stck[s.top],expr[i]))
     {
        pop();
        continue;
     }
     // will only come here if stack is not empty
     // pair wasn't found and it's some closing parenthesis
     //Example : {{}}(]
     return 0;
     }
  }
  return 1;
}
int main() {
 char exp[MAX] = "({})[]{}";
 int i = 0;
```

```
  s.top = -1;

  int len = strlen(exp);
  checkBalanced(exp, len)?printf("Balanced"): printf("Not
Balanced");

  return 0;
}
```

Output

```
INPUT THE STRING : ()(){{}}

BALANCED EXPRESSION
```

3.6 Queues

A queue is an ordered collection of elements in which insertions are made at one end and deletions are made at the other end. The end at which insertions are made is referred to as the **read end.** And the end form which deletions are made is referred to as the **front end.** The first element placed in a queue will be at the first of the queue. That last element placed in a queue will be at the last of the queue. In queue, the first element inserted will be the first element to be removed. So a queue is sometimes referred to as **First-In-First-Out (FIFO)** lists.

Example: Consider five persons waiting in front of a ticket counter in a line for buying their tickets. The person who is

standing in front of the line will get the first ticket, the second person will get the next ticket, and so on. If a new person wants to buy a ticket he should stand after the fifth person. These similar operations are carried out in a queue.

The queue shown in fig: consists of 5 elements 1, 2, 3, 4 and 5.

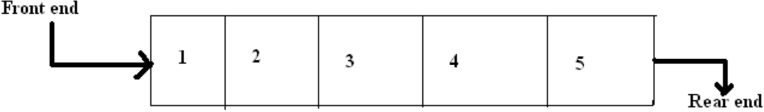


Fig: Representation of a queue

Where front end is the pointer pointing to the first element in the queue. Rear end is the pointer pointing to the last element in the queue. Element 1 is the first element of the queue and 5 is the last element of the queue. If you want to delete an element say 3 we have to first delete element 1 and then element 2 and then the element 3. The front end is shifted from 1st element 1 to 4.

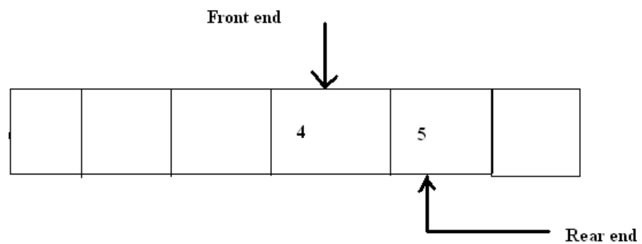


Fig: Queue representation after deletion

Similarly, if we want to add a new element (say 6) it is added after 5 because it is in the rear end. After inserting a new element, the rear end is shifted from element 5 to element 6. Which is the last position of the queue beyond this position; we cannot insert any elements into the queue.

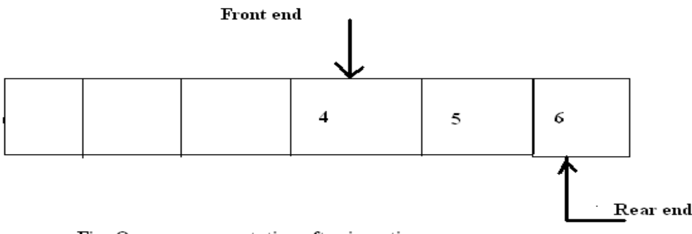


Fig: Queue representation after insertion

Queue operations: The primary operations that can be done on a queue are insertions and deletions. These operations on a queue are referred as **enqueue** and **dequeue** operations respectively.

Algorithm for enqueue operation

```
 [Check the queue is empty or not]
Step 1: if(rear==SIZE-1)
 {
  printf("\n Queue is full");
  return;
 }
```

```
Step 2: To increment the rear value
        rear=rear+1;

Step 3: Read the element from the key board
        printf("\n Enter the element");
        scanf("%d", &queue[front]);

Step 4:if(front==-1)
        front ++
```

Algorithm for dequeue operation

```
 [Check the queue if empty or not]
Step 1: if(front==-1)
         {
           printf("\n Queue is empty");
           return;
         }
Step 2: Printf("\n The deleted element is %d", queue[front]);

Step 3: if(front==rear)
        front=rear=-1;
        else
        front++;
```

Algorithm for display operation

```
[Check the queue if empty or not]
Step 1: if(front==-1)
        {
          printf("\n Queue is empty");
          return;
        }
Step 2: Printf("\n The element in the queue is");
       for(i=front;i<=rear;i++)
           printf("%d", queue[i]);
```

3.7 Properties of Queue

Queue can be considered as a line of items which has following essential properties-

(i) It has two ends that are front and rear.

(ii) Addition of new item can only be done at rear.

(iii) Deletion of an item can only be done from front end.

(iv) The item which is added first will be deleted first. Hence, the structure is frequently called 'FIFO'.

(v) Only one item can be added at a time.

(vi) Only one item can be deleted at a time.

(vii) No element other than front and rear elements are visible.

3.8 Implementing queues using arrays and linked lists

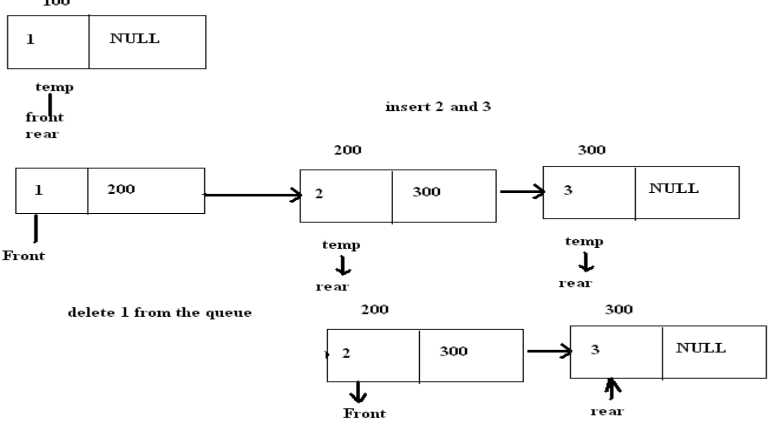
Example 1: Write a C program to perform operations on queue using arrays

```
#include<stdio.h>
#include<conio.h>
#define SIZE  10
int queue[SIZE], front=-1,rear=-1,i;
void enque();
void deque();
void display();
void main()
{
int  ch;
clrscr();
do
{
printf("\n 1.Enque");
printf("\n 2.Deque");
printf("\n 3.Display");
printf("\n 4.Exit");
printf("\n Enter the choice");
scanf("%d",&ch);
switch(ch)
{
case 1:enque();
      break;
case 2:deque();
      break;
case 3:display();
      break;
case 4:exit(0);
```

```
      break;
default: printf("\n Invalid choice");
}
}
while(ch!=4);
getch();
}

void  enque()
{
if(rear==SIZE-1)
 {
  printf("\n Queue is full");
  return;
 }
  rear=rear+1;
  printf("\n Enter the elements");
  scanf("%d", &queue[rear]);
  if(front==-1)
  front++;
}
void  deque( )
{
 if(front==-1)
 {
  printf("\n Queue is empty");
  return;
 }
printf("\n The deleted element is %d",queue[front]);
if(front==rear)
 front=rear=-1;
 else
 front++;
```

```
}

void  display()
{
 if(front==-1)
 {
  printf("\n Queue is empty");
  return;
 }
printf("\n The elements in the queue is front-->");
for(i=front; i<=rear; i++)
printf("%d", queue[i]);
printf(" <-- Rear");
}
}
```

Example 2: Write a C program to perform operations on queue by using linked list



```
#include<stdio.h>
#include<conio.h>
struct  node
{
int data;
struct  node *next;
};
struct node *front=NULL,*rear=NULL, *temp;
void main()
{
 int ch, x;
 clrscr();
 while(1)
 {
  printf("\n 1.Enque");
  printf("\n 2.Deque");
  printf("\n 3.Display");
  printf("\n 4.Exit");
  printf("\n Enter the choice");
  scanf("%d",&ch);
  switch(ch)
  {
   case 1:temp=(struct  node *)malloc(sizeof(struct  node);
          printf("\n Enter the element");
          scanf("%d", &x);
          temp->data=x;
          temp->next=NULL;
          if(front==NULL)
           front=rear=temp;
           else
             {
              rear->next=temp;
              rear=temp;
```

```
        }
      break;
case 2:if(front!=NULL)
      {
       printf("\n The deleted element is %d", front->data);
       front=front->next;
      }
       else
        {
         printf("\n Queue is empty");
         return;
        }
       break;
 case 3: temp=front;
       if(temp==NULL)
        {
        printf("\n Queue is empty");
        return;
        }
        while(temp!=NULL)
         {
          printf("->%d", temp->data);
          temp=temp->next;
         }
        break;
 case 4:exit(0);
      break;
     }
    }
   }
```

3.9 Queue applications

Some common applications of Queue data structure :

Task Scheduling: Queues can be used to schedule tasks based on priority or the order in which they were received.

Resource Allocation: Queues can be used to manage and allocate resources, such as printers or CPU processing time.

Batch Processing: Queues can be used to handle batch processing jobs, such as data analysis or image rendering.

Message Buffering: Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks.

Event Handling: Queues can be used to handle events in event-driven systems, such as GUI applications or simulation systems.

Traffic Management: Queues can be used to manage traffic flow in transportation systems, such as airport control systems or road networks.

Operating systems: Operating systems often use queues to manage processes and resources. For example, a process scheduler might use a queue to manage the order in which processes are executed.

Network protocols: Network protocols like TCP and UDP use queues to manage packets that are transmitted over the network. Queues can help to ensure that packets are delivered in the correct order and at the appropriate rate.

Printer queues: In printing systems, queues are used to manage the order in which print jobs are processed. Jobs are added to the queue as they are submitted, and the printer processes them in the order they were received.

Web servers: Web servers use queues to manage incoming requests from clients. Requests are added to the queue as they are received, and they are processed by the server in the order they were received.

Breadth-first search algorithm: The breadth-first search algorithm uses a queue to explore nodes in a graph level-by-level. The algorithm starts at a given node, adds its neighbors to the queue, and then processes each neighbor in turn.

FCFS (FIRST-COME, FIRST-SERVED) Scheduling

FCFS is considered as simplest CPU-scheduling algorithm. In FCFS algorithm, the process that requests the CPU first is allocated in the CPU first. The implementation of FCFS algorithm is managed with FIFO (First in first out) queue.

FCFS scheduling is non-preemptive. Non-preemptive means, once the CPU has been allocated to a process, that process keeps the CPU until it executes a work or job or task and releases the CPU, either by requesting I/O.

FCFS Scheduling Mathematical Examples

In CPU-scheduling problems some terms are used while solving the problems, so for conceptual purpose the terms are discussed as follows −

Arrival time (AT) − Arrival time is the time at which the process arrives in ready queue.

Burst time (BT) or CPU time of the process − Burst time is the unit of time in which a particular process completes its execution.

Completion time (CT) − Completion time is the time at which the process has been terminated.

Turn-around time (TAT) − The total time from arrival time to completion time is known as turn-around time. TAT can be written as,

Turn-around time (TAT) = Completion time (CT) – Arrival time (AT) or, TAT = Burst time (BT) + Waiting time (WT)

Waiting time (WT) − Waiting time is the time at which the process waits for its allocation while the previous process is in the CPU for execution. WT is written as,

Waiting time (WT) = Turn-around time (TAT) – Burst time (BT)

Response time (RT) − Response time is the time at which CPU has been allocated to a particular process first time.

In case of non-preemptive scheduling, generally Waiting time and Response time is same.

Gantt chart − Gantt chart is a visualization which helps to scheduling and managing particular tasks in a project. It is used while solving scheduling problems, for a concept of how the processes are being allocated in different algorithms.

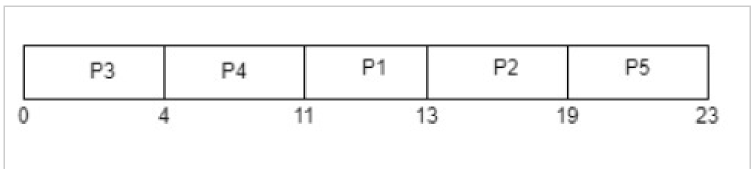
Example

Consider the given table below and find Completion time (CT), Turn-around time (TAT), Waiting time (WT), Response time (RT), Average Turn-around time and Average Waiting time.

Process ID	Arrival time	Burst time
P1	2	2
P2	5	6
P3	0	4
P4	0	7
P5	7	4

Solution

Gantt chart



For this problem CT, TAT, WT, RT is shown in the given table –

Process ID	Arrival time	Burst time	CT	TAT=CT-AT	WT=TAT-BT	RT
P1	2	2	13	13-2= 11	11-2= 9	9
P2	5	6	19	19-5= 14	14-6= 8	8
P3	0	4	4	4-0= 4	4-4= 0	0
P4	0	7	11	11-0= 11	11-7= 4	4
P5	7	4	23	23-7= 16	16-4= 12	12

Average Waiting time = (9+8+0+4+12)/5 = 33/5 = 6.6 time unit (time unit can be considered as milliseconds)

Average Turn-around time = (11+14+4+11+16)/5 = 56/5 = 11.2 time unit (time unit can be considered as milliseconds)

Example: Write a C program on FCFS

```
#include <stdio.h>
int main()
{
   int pid[15];
   int bt[15];
   int n;
   printf("Enter the number of processes: ");
   scanf("%d",&n);

   printf("Enter process id of all the processes: ");
   for(int i=0;i<n;i++)
   {
      scanf("%d",&pid[i]);
   }

   printf("Enter burst time of all the processes: ");
   for(int i=0;i<n;i++)
   {
      scanf("%d",&bt[i]);
   }

   int i, wt[n];
   wt[0]=0;

   //for calculating waiting time of each process
   for(i=1; i<n; i++)
   {
```

```
        wt[i]= bt[i-1]+ wt[i-1];
    }
 printf("Process  ID          Burst  Time          Waiting  Time
TurnAround Time\n");
    float twt=0.0;
    float tat= 0.0;
    for(i=0; i<n; i++)
    {
        printf("%d\t\t", pid[i]);
        printf("%d\t\t", bt[i]);
        printf("%d\t\t", wt[i]);

    //calculating and printing turnaround time of each process
        printf("%d\t\t", bt[i]+wt[i]);
        printf("\n");

        //for calculating total waiting time
        twt += wt[i];
  //for calculating total turnaround time
        tat += (wt[i]+bt[i]);
    }
    float att,awt;

    //for calculating average waiting time
    awt = twt/n;

    //for calculating average turnaround time
    att = tat/n;
    printf("Avg. waiting time= %f\n",awt);
    printf("Avg. turnaround time= %f",att);
}
```

Output

Enter the number of processes: 3

Enter process id of all the processes: 1 2 3

Enter burst time of all the processes: 5 11 11

Process ID	Burst Time	Waiting Time	TurnAround Time
1	5	0	5
2	11	5	16
3	11	16	27

Avg. waiting time= 7.000000

Avg. turnaround time= 16.000000

Round Robin algorithm: In round robin algorithm while executing the one process it goes allows another process for execution. So it is called a preemptive scheduling.

Process	Burst Time (mili sec)
P1	24
P2	10
P3	3

Time quantum (or) Time sliced is 4

P1	P2	P3	P1	P2	P1	P2	P1	P1	P1

0 4 8 11 15 19 23 25 29 33 37

Waiting time for p1= 0+(11-4)+(19-15)+(25-23)

=0+ 7+4+2

=13

P2= 4+(15-8) +(23-19)

=4+7+4

=15

P3= 8

Average waiting time= 13+15+8/3

= 12

Round robin scheduling follows on Windows 95 and window NT operating systems.

3.10 Types of Queues

Types of queues: They are different types of queues. They can be classified as

1. Linear queues
2. Circular queues
3. Deques
4. Priority Queues

1. Linear queues: The queue has two ends. i.e. front end and rear end. The rear end is where we insert elements and front end is where we delete elements. We can traverse

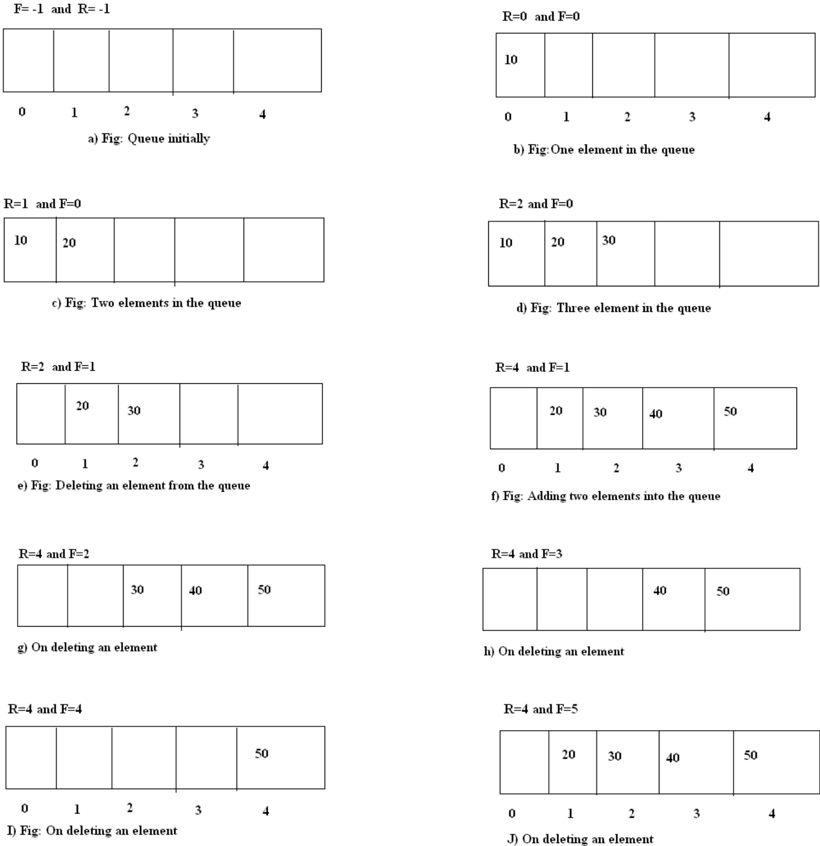
(move) in a linear queue in only one direction (i.e. from front to rear).

If the front pointer is in the first position and the rear pointer is in the last position then the queue is said to be fully occupied. Initially the front and rear ends are at same position (i.e. -1). When we insert elements the rear pointer moves one by one until the last index position is reached. Beyond this we cannot insert the data irrespective of the position of the front pointer. This is the main disadvantage of linear queues. Which overcome in **circular queues.**

When we delete the elements the front pointer moves one by one until the rear pointer is reached. If the front pointer reaches the rear pointer both their positions are initialized to -1 and the queue is said to be empty.

Limitation of simple queues:

The insertion and deletion processes are shown in the following figures.



a) Fig: Queue initially

b) Fig: One element in the queue

c) Fig: Two elements in the queue

d) Fig: Three element in the queue

e) Fig: Deleting an element from the queue

f) Fig: Adding two elements into the queue

g) On deleting an element

h) On deleting an element

I) Fig: On deleting an element

J) On deleting an element

2. Circular queues: Circular queue is another form of a linear queue in which the last position is connected to the first position of the list. It has two ends. The front end and the rear end. The rear end is where we insert elements and front end is where we delete elements. We can traverse in a circular queue in only one direction (i.e. from front to rear).

Initially the front end and rear ends are at same positions (i.e -1). When we insert elements the rear pointer moves one by one until the front end is reached. If the next position of the rear is front, the queue is said to be fully occupied. Beyond this we cannot insert any data. But if we delete and data, we can insert the data accordingly.

When we delete the elements the front pointer moves one by one until the rear pointer is reached. If the front pointer reaches the rear pointer, both their positions are initialized to -1 and the queue is said to be empty.

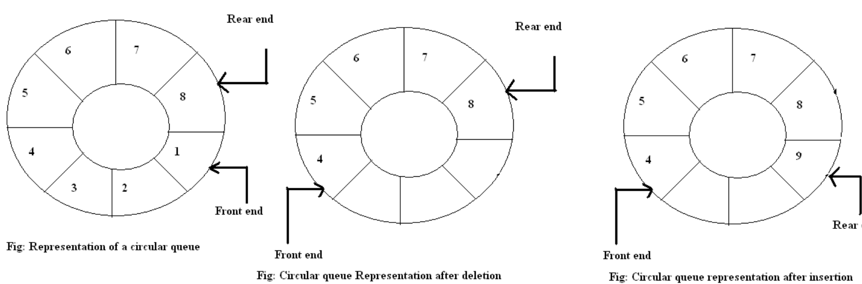


Fig: Representation of a circular queue

Fig: Circular queue Representation after deletion

Fig: Circular queue representation after insertion

Algorithm for circular enqueue operation

```
[Check the circular queue is full or not]
Step 1:if((front==0&&rear==SIZE-1)||(front==rear+1))
then
          printf("\n Circular queue is full");
          return;
Step 2:[check circular queue is empty or not]
       if(front==-1) then
        front=rear=0;
        else
Step 3:[check circular queue is at last position of queue]
      if(rear==SIZE-1)
      rear=0;
      else
      rear++;
Step 4:[Insert the element into circular queue]
      printf("\n Enter the element");
      scanf("%d",&cqueue[rear]);
```

Algorithm for circular dequeue operation

```
[check circular queue is empty or not]
Step 1:if(front==-1) then
      printf("\n Circular queue is empty");
      return;
[delete the element from the circular queue]
Step 2:printf("The deleted element is %d", cqueue[front]);
Step 3: if(front==rear) then
       front=rear=-1;
       else
       if(front==SIZE-1)
       front=0;
       else
       front++;
```

Algorithm for circular display operation

```
[check circular queue is empty or not]
Step 1:if(front==-1) then
      printf("\n Circular queue is empty");
      return;
[Display the elements into the circular queue]
Step 2:for(i=front;i<=rear; i++)
       printf("%d", cqueue[i]);
```

Example: Write a C program to perform operations on Circular Queue

```
#include<stdio.h>
#include<conio.h>
#define SIZE  10
int cqueue[SIZE], front=-1,rear=-1,i;
void cenqueue();
void cdequeue();
void display();
void main()
{
int ch;
clrscr();
do
{
printf("\n 1.Cenqueue");
printf("\n 2.Cdequeue");
printf("\n 3.Display");
printf("\n 4.Exit");
printf("\n Enter the choice");
scanf("%d",&ch);
switch(ch)
{
case 1:Cenqueue();
      break;
case 2:Cdequeue();
      break;
case 3:display();
      break;
case 4:exit(0);
      break;
default: printf("\n Invalid choice");
}
```

```
}
while(ch!=4);
getch();
}


void cenqueue()
{
if((Front==0)&&(rear==SIZE-1)||(front==rear+1))
 {
  printf("\n Circular queue is full");
  return;
 }
  if(front==-1)
   {
    front=rear=0;
   }
   else
    if(rear==SIZE-1)
    rear=0;
    else
    rear++;
  printf("\n Enter the elements");
  scanf("%d", &cqueue[rear]);
}

void cdequeue( )
{
 if(front==-1)
 {
  printf("\n Circular queue is empty");
  return;
 }
```

```
printf("\n The deleted element is %d",cqueue[front]);
if(front==rear)
 {
   front=rear=-1;
 }
 else if(front==SIZE-1)
 {
  front=0;
  else
  front++;
 }
void display()
{
 if(front==-1)
 {
  printf("\n Circular queue is empty");
  return;
 }
printf("\n front=%d  rear=%d",front,rear);
for(i=front;i<=rear;i++)
printf("%d", cqueue[i]);
}
```

3. Deques: Deque (Double-ended queue) is another form of a queue in which insertions and deletions are made at both the front and rear ends of the queue. There are two variations of a deque namely

1. Input restricted deque
2. Output restricted deque

1. Input restricted deque: The input restricted deque allows insertion at one end (It can be either front or rear) only.

2. Output restricted deque: The output restricted deque allows deletion at one end (It can be either front or rear) only.

The different types of deques are

1. Linear deque
2. Circular deque

1. Linear deque: The linear deque is similar to a linear queue except the following conditions.

a) The insertions and deletions are made at both the front and rear ends of the deque.

b) If the front end is in the first position, we cannot insert the data at front end.

c) If the rear end is in the last position, we cannot insert the data at rear end.

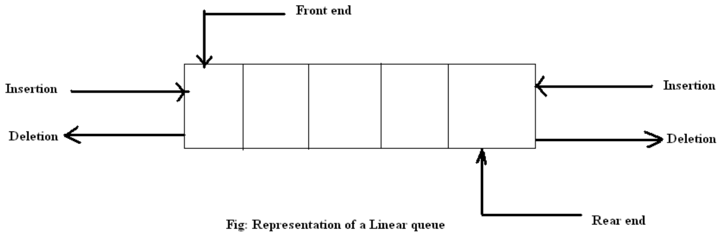


Fig: Representation of a Linear queue

2. Circular deque: Circular deque is similar to a circular queue except the following conditions.

a). The insertions and deletions are made at both the front end and rear ends of the deque.

b) Irrespective of the positions of the front and rear end, we can insert and delete data.

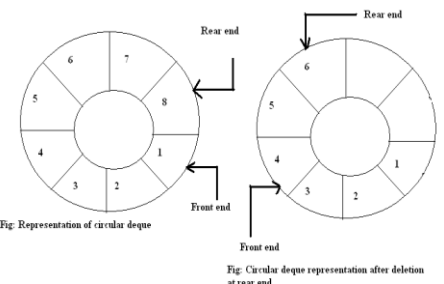


Fig: Representation of circular deque

Fig: Circular deque representation after deletion at rear end

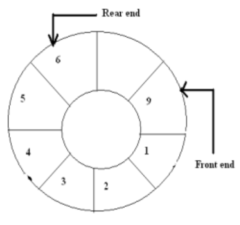


Fig: Circular queue representation after insertion at front end.

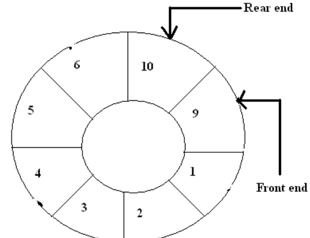


Fig: Circular deque representation after insertion at rear end

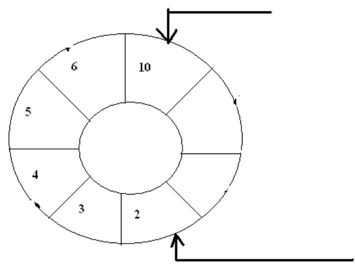


Fig: Circular deque representation after deletion at front end

4. Priority Queues

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

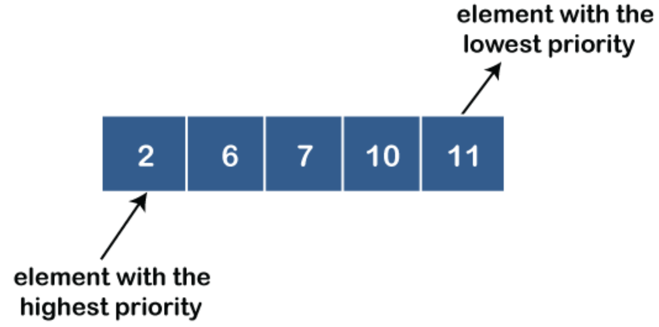
A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

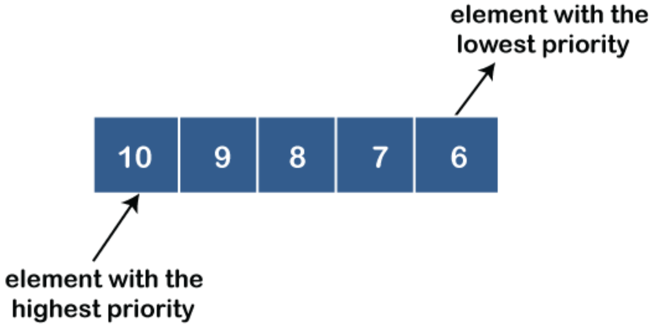
Types of Priority Queue

There are two types of priority queue:

Ascending order priority queue: In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.

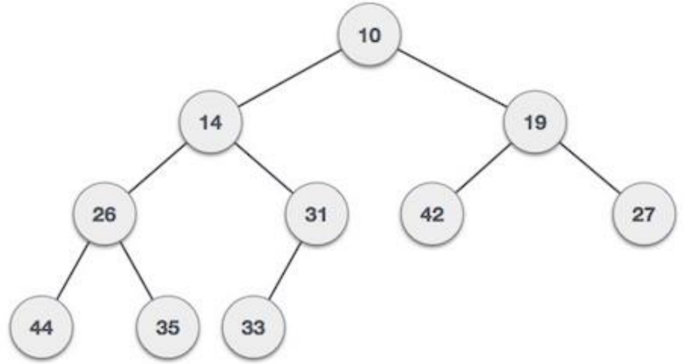


Descending order priority queue: In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.

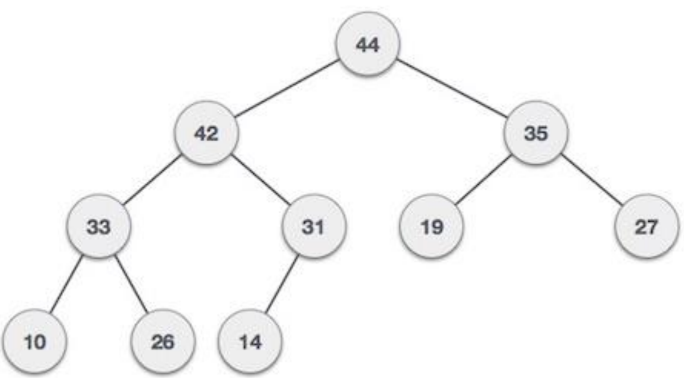


Priority queue can be implemented using an array, a linked list, a **heap data structure**, or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues.

Min-Heap – The value of the root node is less than or equal to either of its children.



Max-Heap − The value of the root node is greater than or equal to either of its children.



Max Heap Construction Algorithm

```
Step 1 − Create a new node at the end of heap.
Step 2 − Assign new value to the node.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.
```

Max Heap Deletion Algorithm

Step 1 − Remove root node.

Step 2 − Move the last element of last level to root.

Step 3 − Compare the value of this child node with its parent.

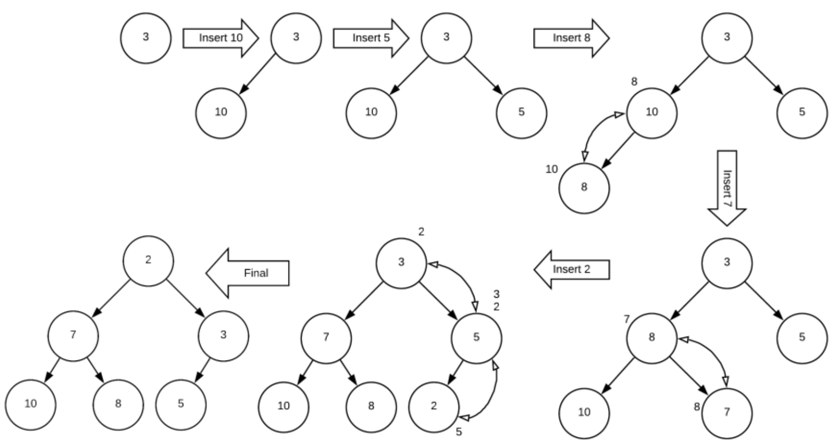
Step 4 − If value of parent is less than child, then swap them.

Step 5 − Repeat step 3 & 4 until Heap property holds.

Priority Queue Operations

The common operations that we can perform on a priority queue are insertion, deletion and peek.

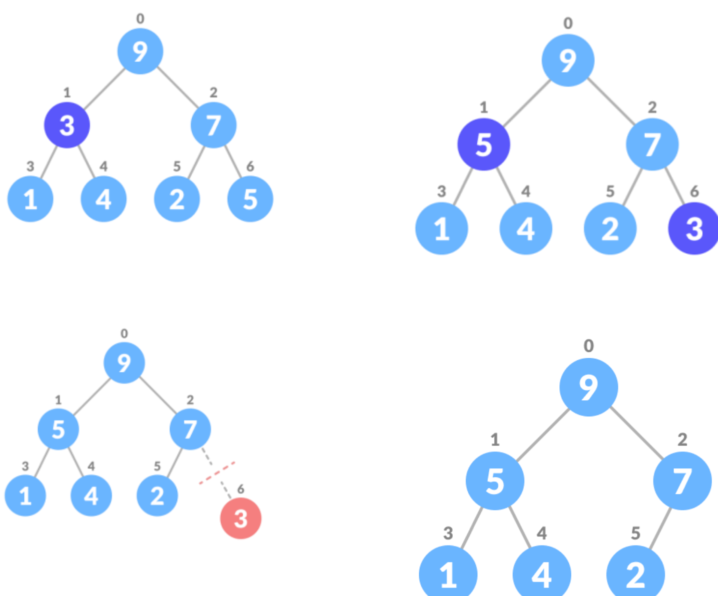
Insert



Delete

Deleting an element from a priority queue (max-heap) is done as follows:

- Select the element to be deleted.
- Swap it with the last element
- Remove the last element.
- Heapify the tree



UNIT-IV

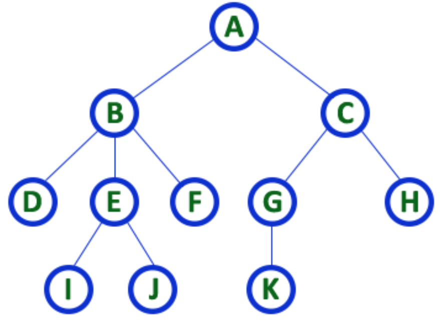
Trees- Introduction, Types and basic properties.

Binary Trees–Definition, Tree traversals, Tree representations. **Binary Search Trees** – Definition, properties and applications. **AVL trees-** Introduction and basic operations. **Heap** – Introduction and types, Heap sort.

4.1 Trees- Introduction, Types and basic properties.

What is a Tree?

A tree is a non-linear and hierarchical data structure that has a group of nodes. When it comes to the tree, each node stores a value.



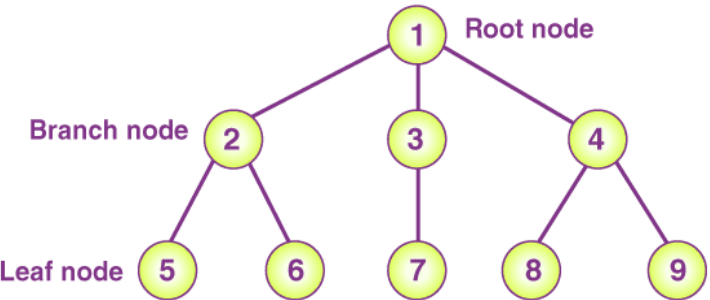
TREE with 11 nodes and 10 edges

- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

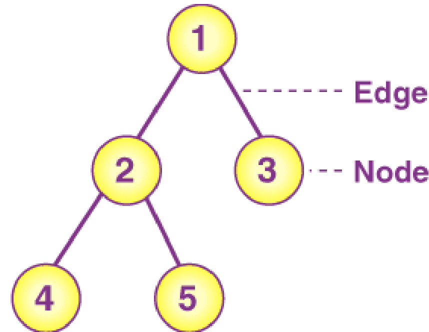
Important Terminologies in Tree

Root: The topmost node of a tree is known as the root.

Node: A node is an entity that contains a key or value and pointers to its child nodes.



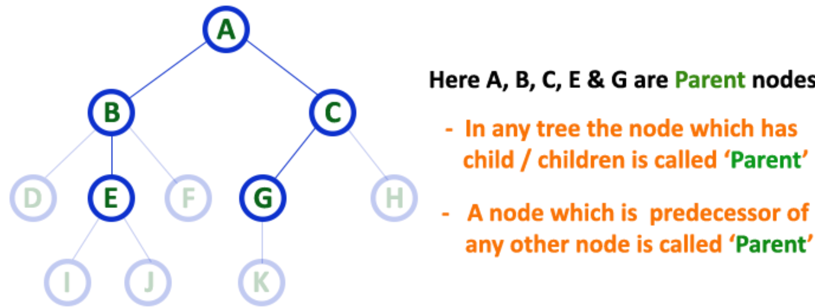
Edge: The connection between any two nodes is known as the edge. In a tree data structure, if we have N number of nodes then we can have a maximum of N-1 number of links.



Parent

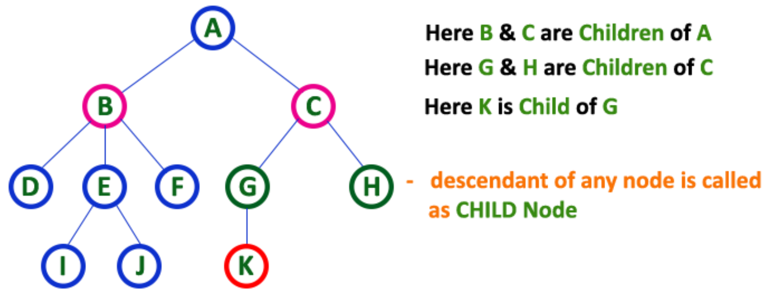
In a tree data structure, the node which is a predecessor of any node is called as PARENT NODE. In simple words, the

node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "The node which has child / children".



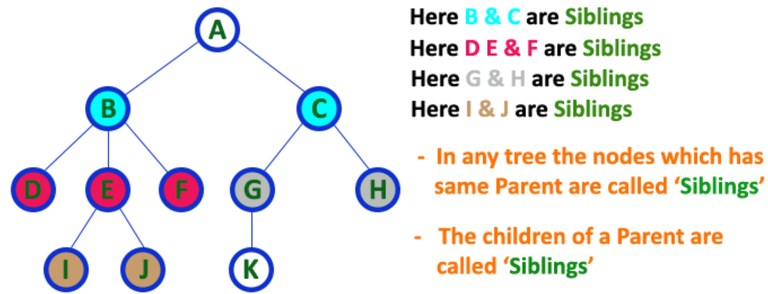
Child

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.



Siblings

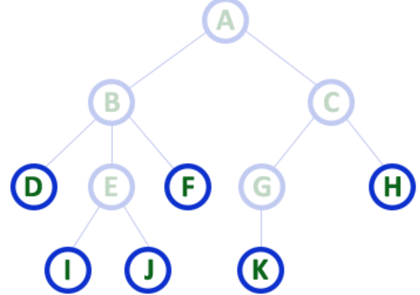
In a tree data structure, nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with the same parent are called Sibling nodes.



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.



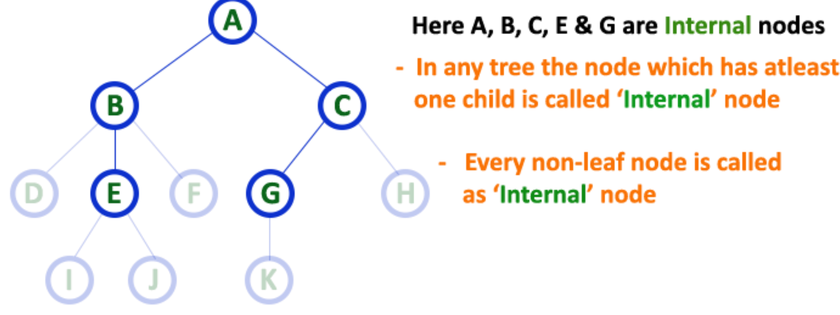
Here D, I, J, F, K & H are Leaf nodes

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

Internal Nodes

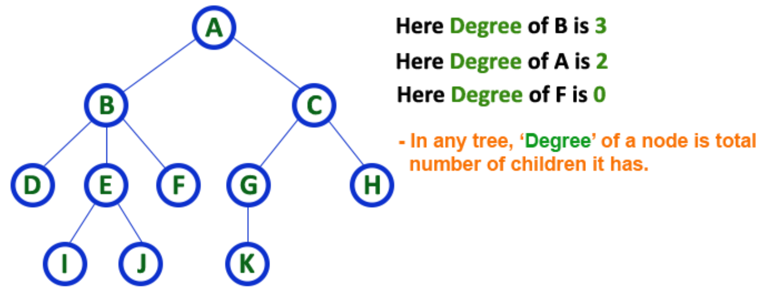
In a tree data structure, the node which has atleast one child is called as INTERNAL Node. In simple words, an internal node is a node with atleast one child.

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.



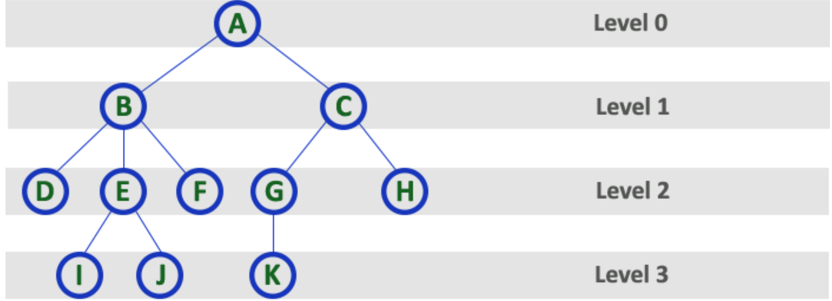
Degree

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'.



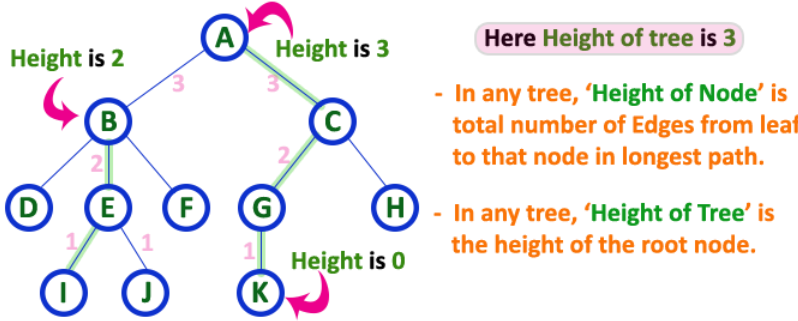
Level

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).



Height

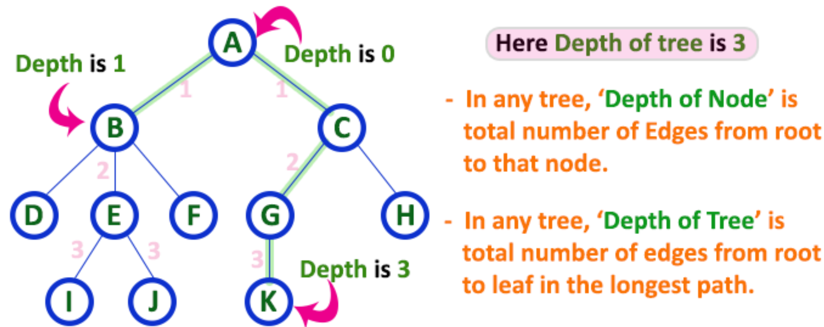
In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.



Depth

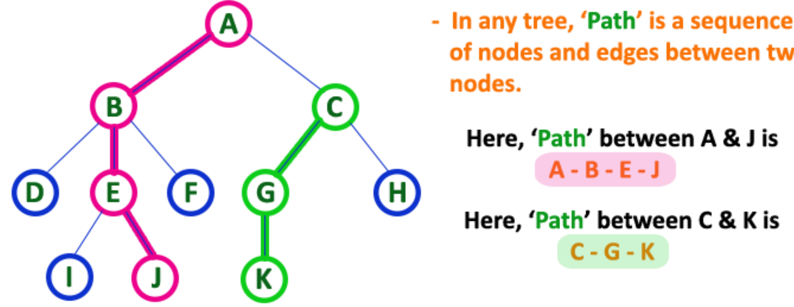
In a tree data structure, the total number of egdes from root node to a particular node is called as DEPTH of that Node.

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'.



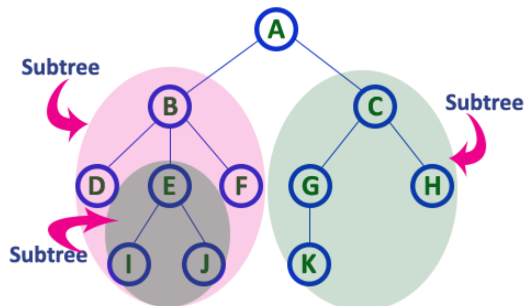
Path

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.



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.



4.2 Properties of Trees

- A tree is a hierarchical structure as it contains multiple levels.
- In a tree, the topmost node is known as the root node.

- A node that doesn't have a child node is known as a leaf node or terminal node.
- The highest number of nodes at every level of i is 2i.
- Height of the tree = the longest path from the root node to the leaf node.
- Depth of a node = the length of the path to its root.

4.3 Binary Tree Representation

A binary tree data structure is representing using two methods

- Array Representation
- Linked list Representation

Array Representation

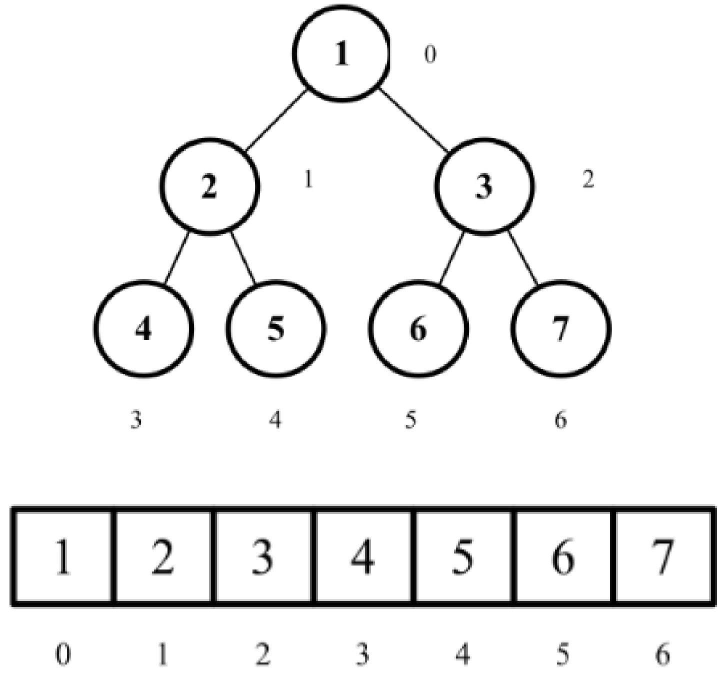
We use 1-D array to represent binary tree

Steps: 1. Consider the root node at index 0

Step 2: for every left child=2i+1

Step3: for every right child=2i+2

Example

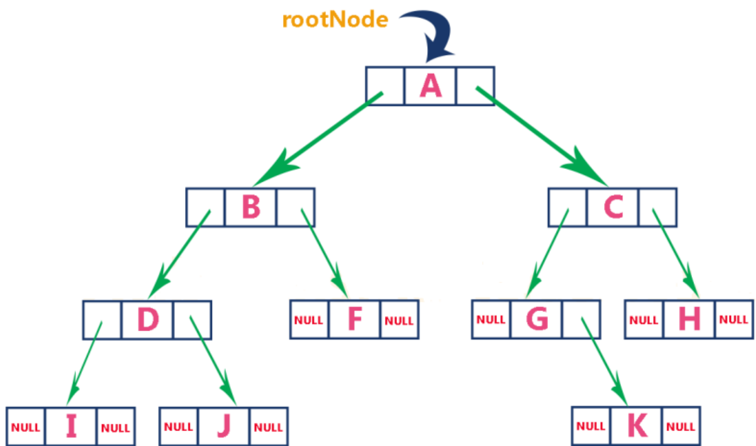


Linked list Representation

We use a 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 the binary tree represented using Linked list representation is shown as follows.



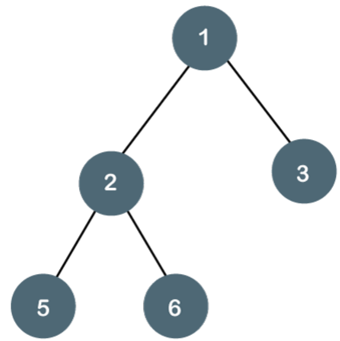
4.4 Types of Trees

1. Binary Tree
2. Binary Search Tree
3. AVL Tree

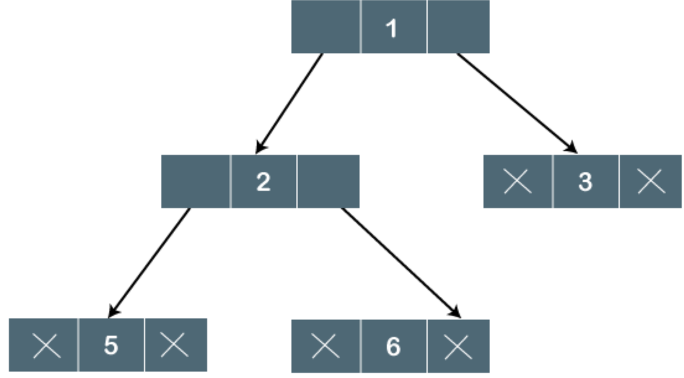
1. Binary Tree

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

Let's understand the binary tree through an example.



The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



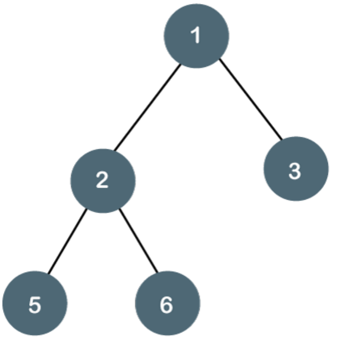
In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain NULL pointer on both left and right parts.

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

Types of Binary Tree

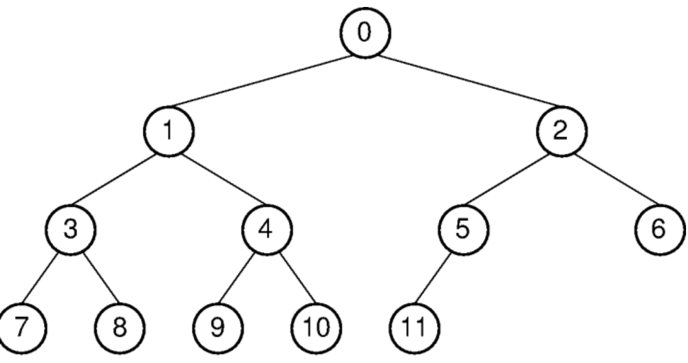
a) **Full/ proper/ strict Binary tree**

The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children. The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.



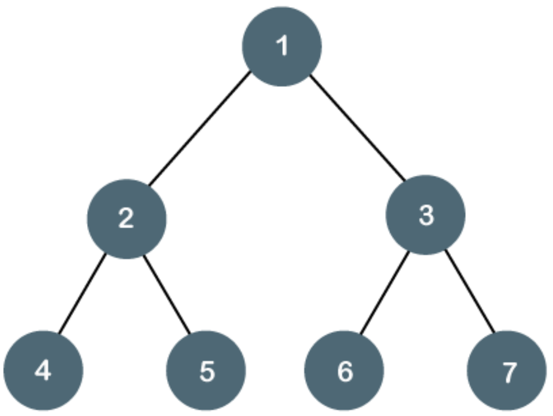
b) Complete Binary Tree

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.



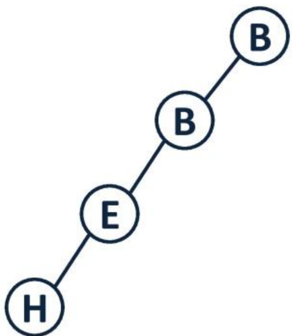
c) Perfect Binary Tree

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.



d) **Left Skewed Binary Tree**

These are those skewed binary trees in which all the nodes are having a left child or no child at all. It is a left side dominated tree. All the right children remain as null.

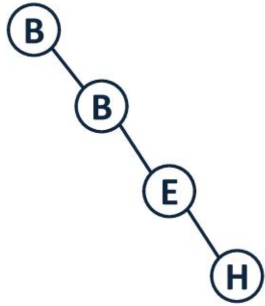


```
#include <bits/stdc++.h>
using namespace std;

// A Tree node
struct Node {
   int key;
   struct Node *left, *right;
};

// Utility function to create a new node
Node* newNode(int key)
{
   Node* temp = new Node;
   temp->key = key;
   temp->left = temp->right = NULL;

   return (temp);
}

// Driver code
int main()
{
   /*
          1
         /
        2
       /
      3
   */
   Node* root = newNode(1);
   root->left = newNode(2);
   root->left->left = newNode(3);
```

```
    return 0;
}
```

e) Right Skewed Binary Tree:

These are those skewed binary trees in which all the nodes are having a right child or no child at all. It is a right side dominated tree. All the left children remain as null.

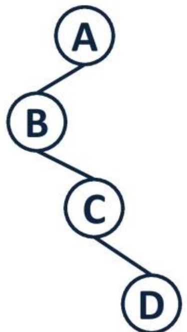



```
#include <bits/stdc++.h>
using namespace std;

// A Tree node
struct Node {
   int key;
   struct Node *left, *right;
};

// Utility function to create a new node
Node* newNode(int key)
```

```
{
   Node* temp = new Node;
   temp->key = key;
   temp->left = temp->right = NULL;

   return (temp);
}

// Driver code
int main()
{
   /*
       1
        \
         2
          \
           3
   */
   Node* root = newNode(1);
   root->right = newNode(2);
   root->right->right = newNode(3);

   return 0;
}
```

f) Degenerate Or Pathological Tree

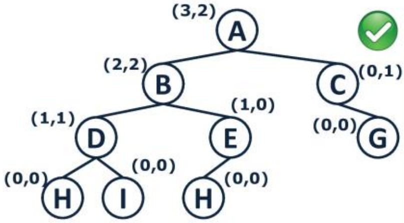
A degenerate or Pathological Tree is a Tree where every parent node has only one child either left or right.



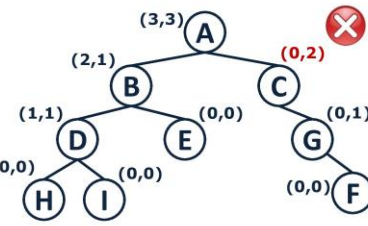
g) Balanced Binary Tree

Binary tree is called Balanced Binary Tree, if difference of left and right subtree height is maximum one for all the nodes.

Balanced Binary Tree



It is Balanced Binary Tree as for all nodes, difference of left and right subtree height is not more than one

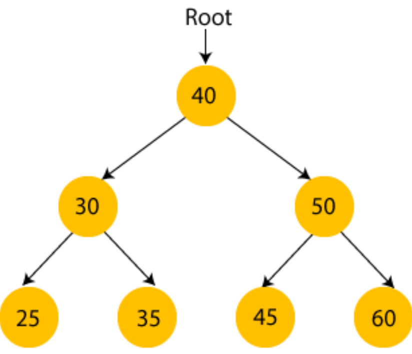


Node C violates the property of Balanced Binary Tree.

2. Binary Search Tree

In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

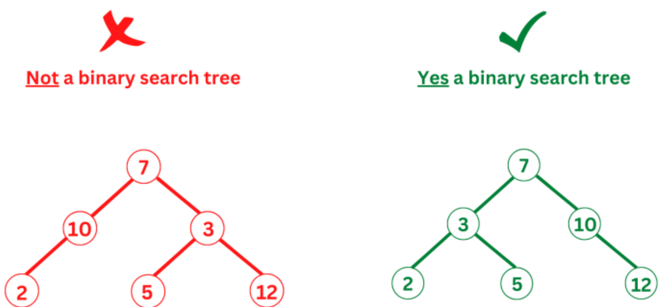
Let's understand the concept of Binary search tree with an example.



In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also

satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.



Binary Search Tree (BST) is a special binary tree that has the properties:

- The left subtree contains only the keys which are lesser than the key of the node.
- The right subtree contains only the keys which are greater than the key of the node.
- The left and right subtree both should be binary search tree.

Applications of BST

- A Self-Balancing Binary Search Tree is used to maintain sorted stream of data
- A Self-Balancing Binary Search Tree is used to implement doubly ended priority queue.

- One of the most common use cases of BSTs is searching for a particular element in the tree.
- A BST can be used to sort a large dataset.
- Used in Database indexing.

- BSTs can be used to implement symbol tables, which are used to store data such as variable and function names in a programming language.

Operations on Binary Search tree

The four basic operations of BST

- Insertion
- Deletion
- Searching
- Traversals

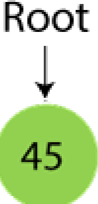
Example of creating a binary search tree

Suppose the data elements are - 45, 15, 79, 90, 10, 55, 12, 20, 50

- First, we have to insert 45 into the tree as the root of the tree.

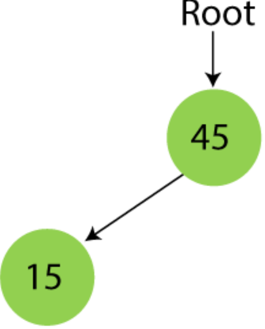
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

Step 1 - Insert 45.



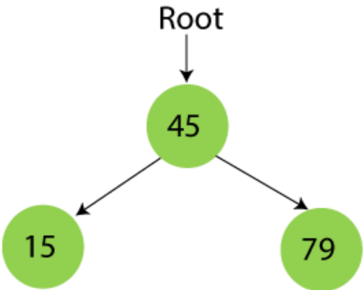
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



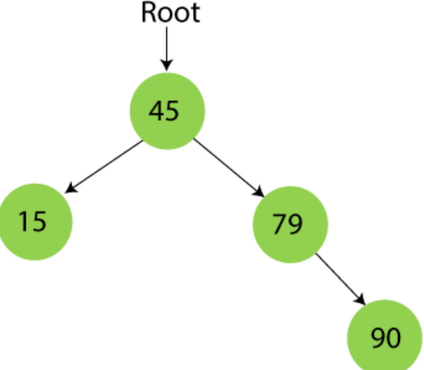
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



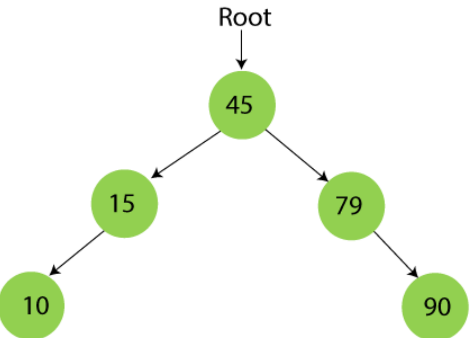
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



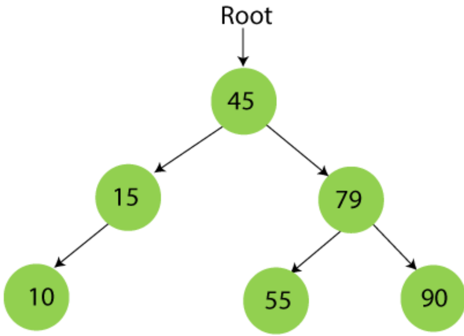
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



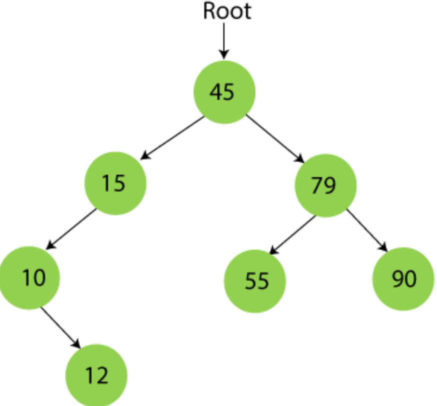
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



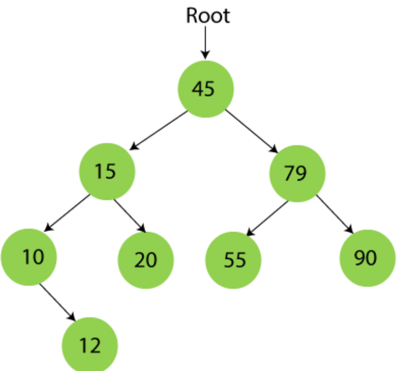
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



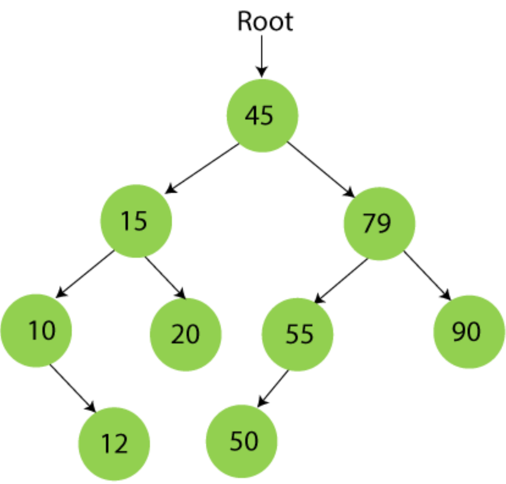
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Deletion in Binary Search tree

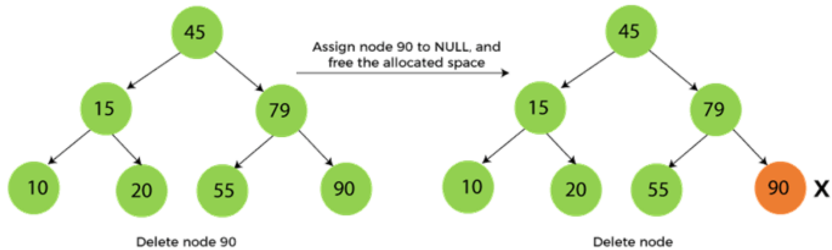
In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

The node to be deleted is the leaf node

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

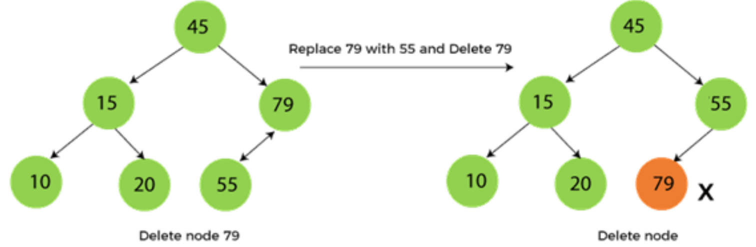
We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.



The node to be deleted has only one child

We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.

So, the replaced node 79 will now be a leaf node that can be easily deleted.

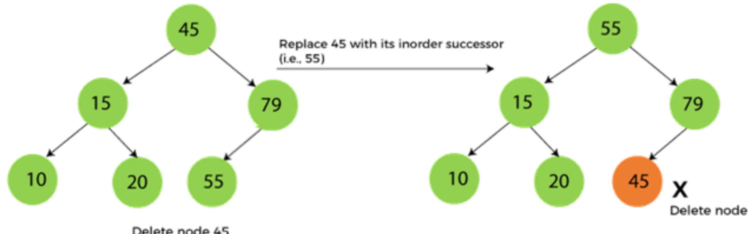


The node to be deleted has two children

This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -

- First, find the inorder successor of the node to be deleted.
- After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
- And at last, replace the node with NULL and free up the allocated space.

In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.

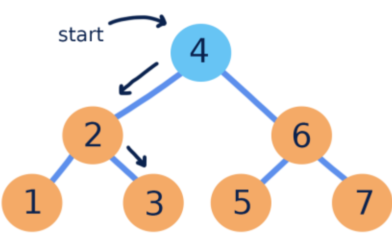


Searching in Binary search tree

Binary search trees are called “search trees” because they make searching for a certain value more efficient than in an unordered tree. In an ideal binary search tree, we do not have to visit every node when searching for a particular value.

Here is how we search in a binary search tree:

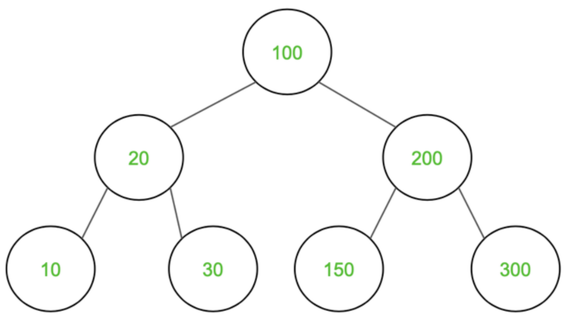
- Begin at the tree’s root node
- If the value is smaller than the current node, move left
- If the value is larger than the current node, move right



Binary Search Tree (BST) Traversals

- **Inorder**

Input:



Binary Search Tree

Output:

Inorder Traversal: 10 20 30 100 150 200 300

```
#include <stdio.h>
#include <stdlib.h>
struct node {
  int data;
  struct node *leftChild;
  struct node *rightChild;
};
struct node *root = NULL;
void insert(int data){

  struct node *tempNode = (struct node*) malloc(sizeof(struct node));
  struct node *current;
  struct node *parent;
  tempNode->data = data;
  tempNode->leftChild = NULL;
  tempNode->rightChild = NULL;

  //if tree is empty
  if(root == NULL) {
    root = tempNode;
  } else {
    current = root;
    parent = NULL;
    while(1) {
      parent = current;

      //go to left of the tree
      if(data < parent->data) {
        current = current->leftChild;

        //insert to the left
        if(current == NULL) {
```

```
            parent->leftChild = tempNode;
            return;
         }
      }//go to right of the tree
      else {
         current = current->rightChild;

         //insert to the right
         if(current == NULL) {
            parent->rightChild = tempNode;
            return;
         }
      }
    }
  }
}
void inorder_traversal(struct node* root){
  if(root != NULL) {
    inorder_traversal(root->leftChild);
    printf("%d ",root->data);
    inorder_traversal(root->rightChild);
  }
}
int main(){
  int i;
  int array[7] = { 27, 14, 35, 10, 19, 31, 42 };
  for(i = 0; i < 7; i++)
    insert(array[i]);
  printf("Inorder traversal: ");
  inorder_traversal(root);
  return 0;
}
```

Output

```
Inorder traversal: 10 14 19 27 31 35 42
```

The complexity of the Binary Search tree

1. Time Complexity

Operations	Best case time complexity	Average case time complexity	Worst case time complexity
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(n)

2. Space Complexity

Operations	Space complexity
Insertion	O(n)
Deletion	O(n)
Search	O(n)

3. AVL Tree

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

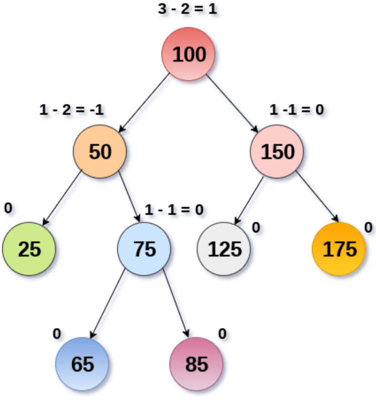
- AVL Trees are Self-Balanced Binary Search Trees.
- In AVL trees, the balancing factor of each node is either 0 or 1 or -1.
- Balance Factor of AVL Tree calculated as = Height of Left Sub-tree - Height of Right Sub-tree

Balance Factor

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

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

- If balance factor of any node is **1,** it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is **-1,** it means that the left sub-tree is one level lower than the right sub-tree.
- If balance factor of any node is **0,** it means that the left sub-tree and right sub-tree contain equal height.



AVL Tree

Operations on an AVL tree

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

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

AVL Rotations

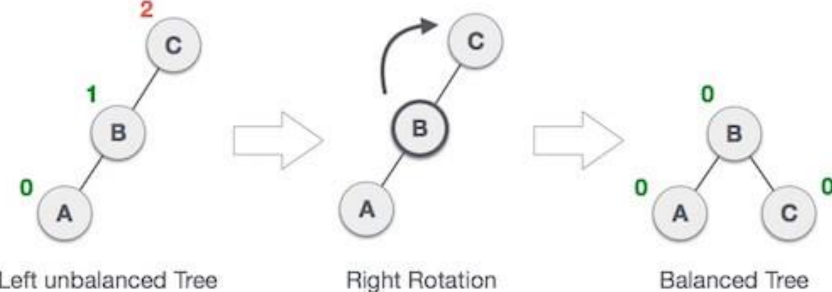
We perform rotation in AVL tree only in case if Balance Factor is other than -1, 0, and 1. There are basically four types of rotations which are as follows:

1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

The first two rotations **LL and RR are single rotations** and the next two rotations **LR and RL are double rotations**. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

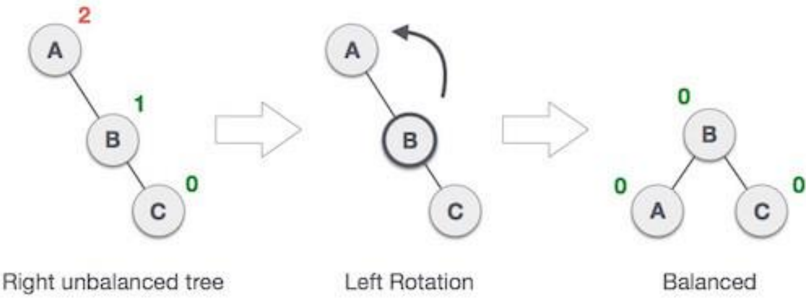
1. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



2. **RR Rotation**

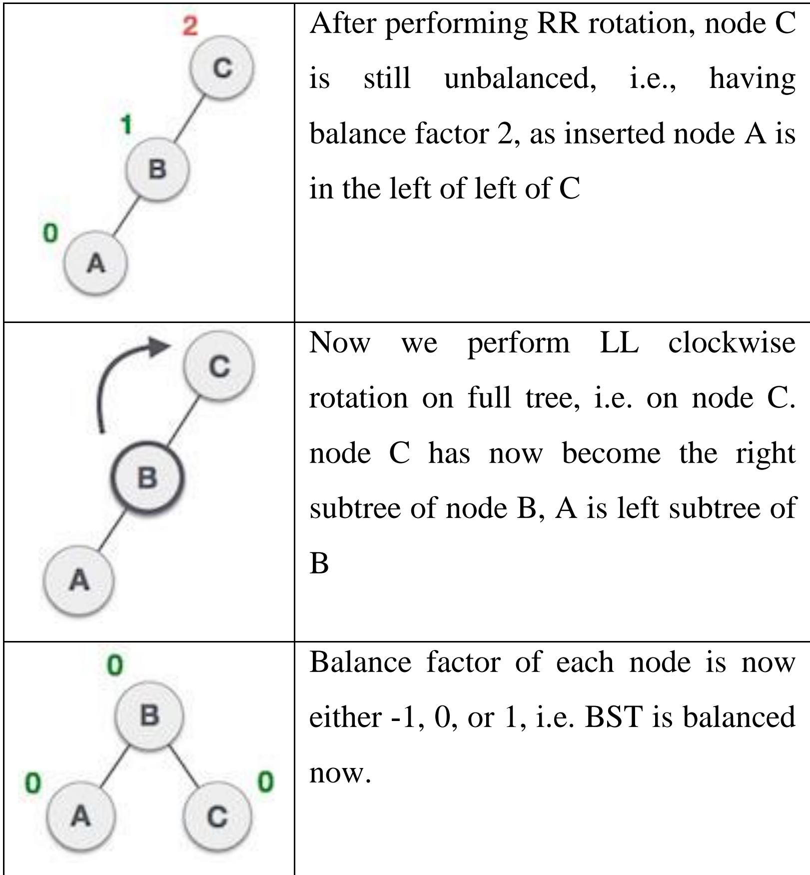
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



3. LR Rotation

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node A, has become the left subtree of B.



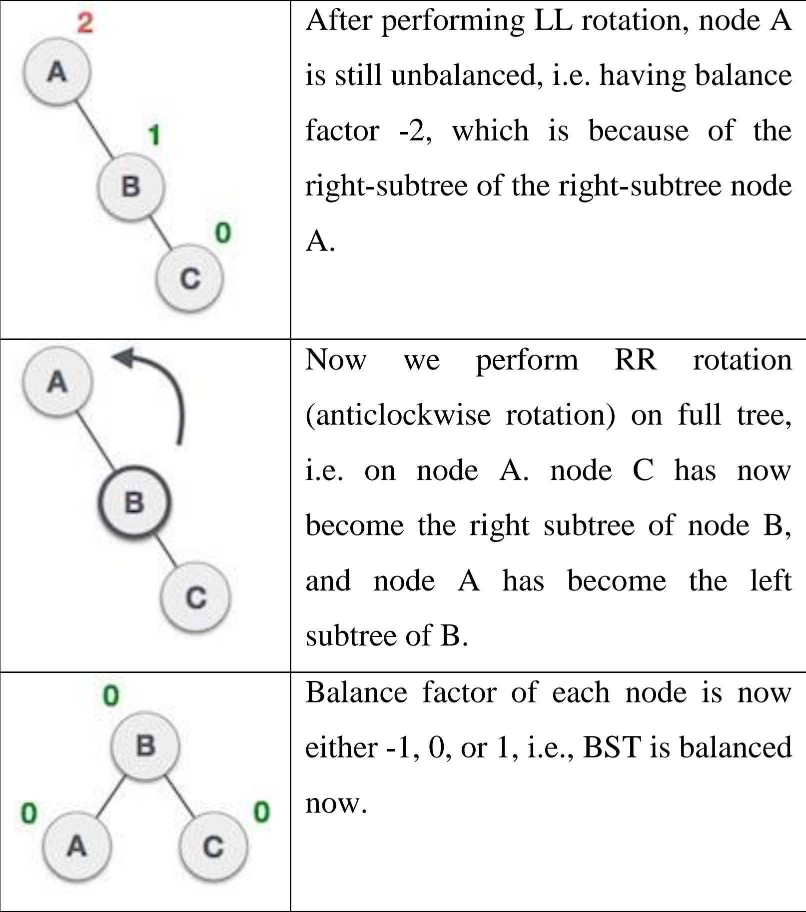
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C
	Now we perform LL clockwise rotation on full tree, i.e. on node C. node C has now become the right subtree of node B, A is left subtree of B
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

4. RL Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on

full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

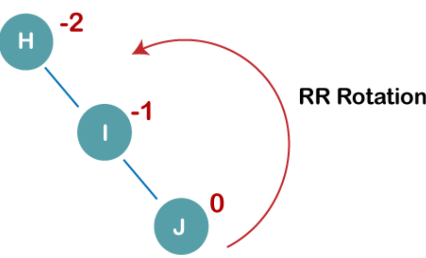
State	Action
2 A 1 C 0 B	A node B has been inserted into the left subtree of C the right subtree of A, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A
A C B	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at C is performed first. By doing RR rotation, node C has become the right subtree of B.



	After performing LL rotation, node A is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.
	Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node C has now become the right subtree of node B, and node A has become the left subtree of B.
	Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

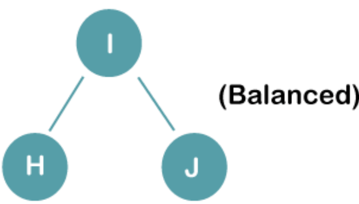
Q: Construct an AVL tree having the following elements H, I, J, B, A, E

Sol: 1. Insert H, I, J

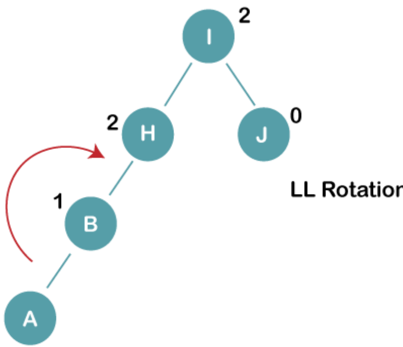


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H. The resultant balance tree is:

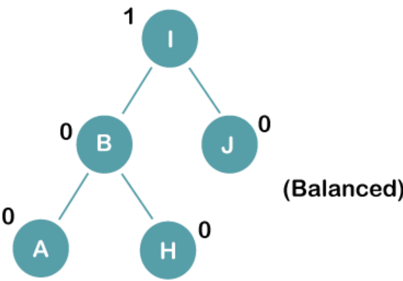
The resultant balance tree is:



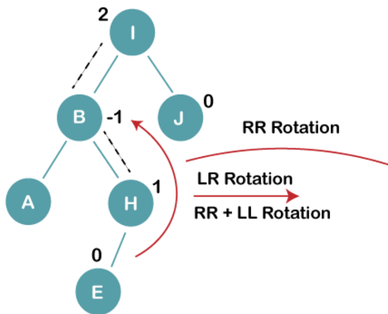
2. Insert B, A



On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H. The resultant balance tree is:



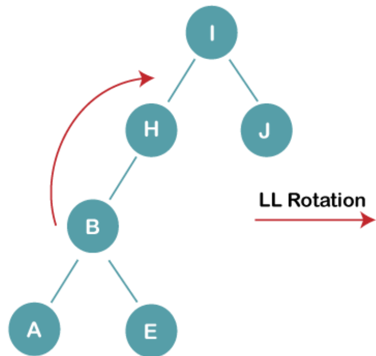
3. Insert E



On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = RR + LL rotation

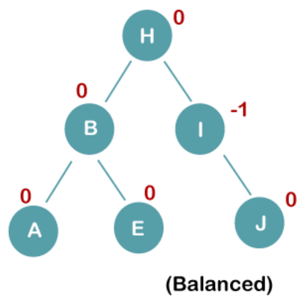
3 a) We first perform RR rotation on node B

The resultant tree after RR rotation is:



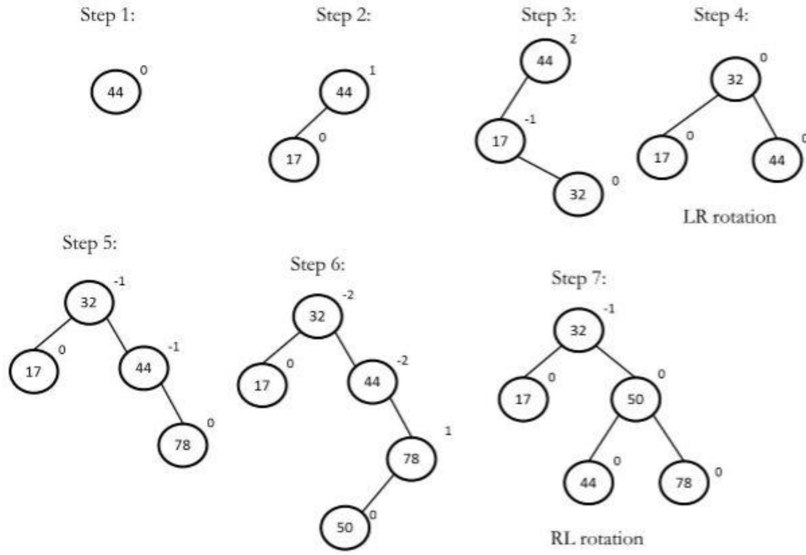
3b) We first perform LL rotation on the node I

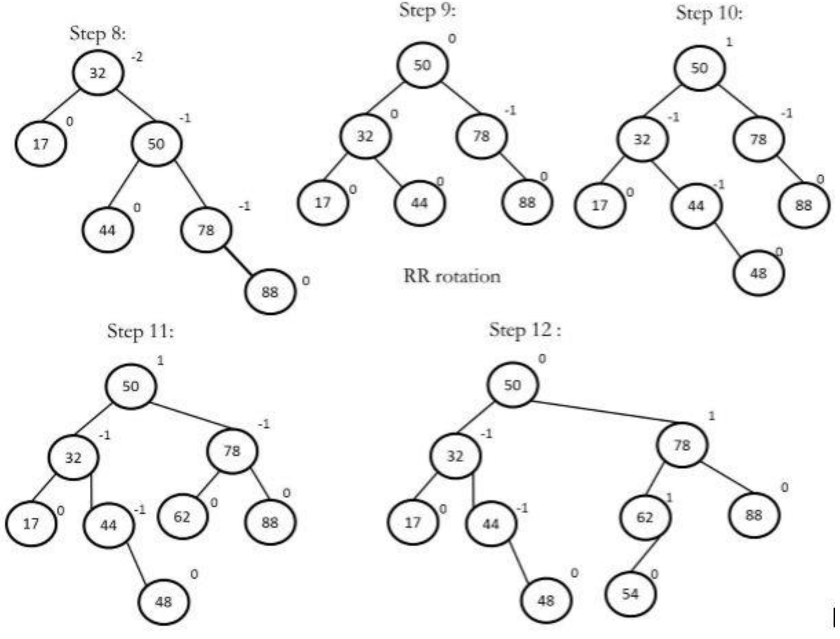
The resultant balanced tree after LL rotation is:



Construct an AVL tree having the following elements

44, 17, 32, 78, 50, 88, 48, 62, 54





4.5 Tree traversals

- **Preorder**
- **Inorder**
- **Postorder**

Preorder Traversal: 100 20 10 30 200 150 300

```
void pre_order_traversal(struct node* root){
   if(root != NULL) {
      printf("%d ",root->data);
      pre_order_traversal(root->leftChild);
      pre_order_traversal(root->rightChild);
   }
}
```

Inorder Traversal: 10 20 30 100 150 200 300

```
void inorder_traversal(struct node* root){
   if(root != NULL) {
      inorder_traversal(root->leftChild);
      printf("%d ",root->data);
      inorder_traversal(root->rightChild);
   }
}
```

Postorder Traversal: 10 30 20 150 300 200 100

```
void post_order_traversal(struct node* root){
   if(root != NULL) {
      post_order_traversal(root->leftChild);
      post_order_traversal(root->rightChild);
      printf("%d ", root->data);
   }
}
```

```
#include <stdio.h>
#include <stdlib.h>
struct node {
  int data;
  struct node *leftChild;
  struct node *rightChild;
};
struct node *root = NULL;
void insert(int data){
  struct node *tempNode = (struct node*) malloc(sizeof(struct node));
  struct node *current;
  struct node *parent;
  tempNode->data = data;
  tempNode->leftChild = NULL;
  tempNode->rightChild = NULL;

//if tree is empty
  if(root == NULL) {
    root = tempNode;
  } else {
    current = root;
    parent = NULL;
    while(1) {
      parent = current;

      //go to left of the tree
      if(data < parent->data) {
        current = current->leftChild;

        //insert to the left
        if(current == NULL) {
          parent->leftChild = tempNode;
```

```
            return;
          }
        }//go to right of the tree
        else {
          current = current->rightChild;

          //insert to the right
          if(current == NULL) {
            parent->rightChild = tempNode;
            return;
          }
        }
      }
   }
}
void pre_order_traversal(struct node* root){
  if(root != NULL) {
    printf("%d ",root->data);
    pre_order_traversal(root->leftChild);
    pre_order_traversal(root->rightChild);
  }
}
void inorder_traversal(struct node* root){
  if(root != NULL) {
    inorder_traversal(root->leftChild);
    printf("%d ",root->data);
    inorder_traversal(root->rightChild);
  }
}
void post_order_traversal(struct node* root){
  if(root != NULL) {
    post_order_traversal(root->leftChild);
    post_order_traversal(root->rightChild);
```

```
      printf("%d ", root->data);
   }
}
int main(){
   int i;
   int array[7] = { 27, 14, 35, 10, 19, 31, 42 };
   for(i = 0; i < 7; i++)
      insert(array[i]);
   printf("Preorder traversal: ");
   pre_order_traversal(root);
   printf("\nInorder traversal: ");
   inorder_traversal(root);
   printf("\nPost order traversal: ");
   post_order_traversal(root);
   return 0;
}
```

4.6 Heap

A heap is a complete binary tree in which the node can have the utmost two children.

What is heap sort?

Heap sort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list. Heap sort is the **in-place sorting algorithm.**

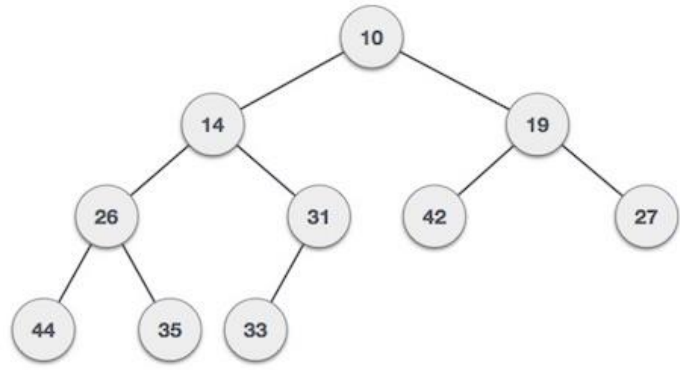
Types

There are two types of Heap

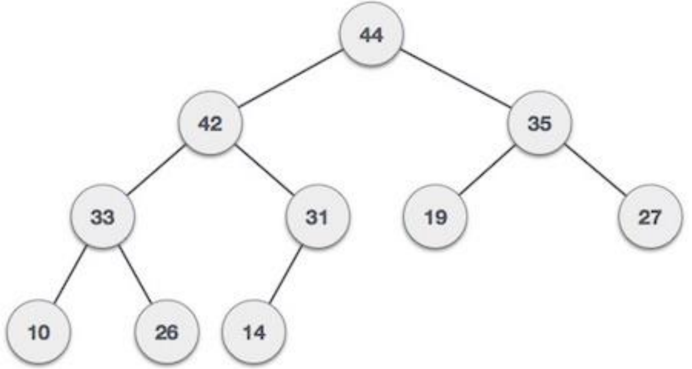
Max Heap: The value of each node is greater than its children.

Min Heap: The value of each node is Smaller than its children.

Min-Heap – The value of the root node is less than or equal to either of its children.



Max-Heap − The value of the root node is greater than or equal to either of its children.



Step by Step Process

The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

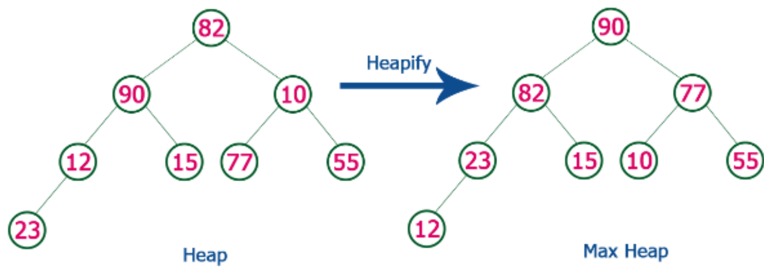
- **Step 1 -** Construct a **Binary Tree** with given list of Elements.
- **Step 2 -** Transform the Binary Tree into **Min Heap.**
- **Step 3 -** Delete the root element from Min Heap using **Heapify** method.
- **Step 4 -** Put the deleted element into the Sorted list.
- **Step 5 -** Repeat the same until Min Heap becomes empty.
- **Step 6 -** Display the sorted list.

Example

Consider the following list of unsorted numbers which are to be sort using Heap Sort

82, 90, 10, 12, 15, 77, 55, 23

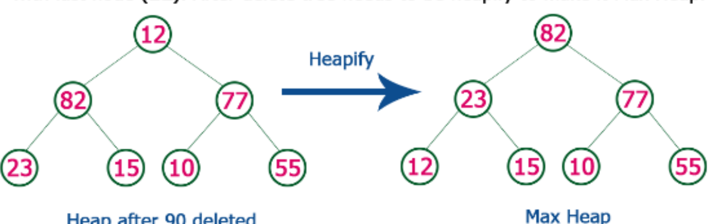
Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

90, 82, 77, 23, 15, 10, 55, 12

Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, **90**

Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.

Heap after 82 deleted Max Heap

list of numbers after swapping 82 with 55.

12, 55, 77, 23, 15, 10, **82**, **90**

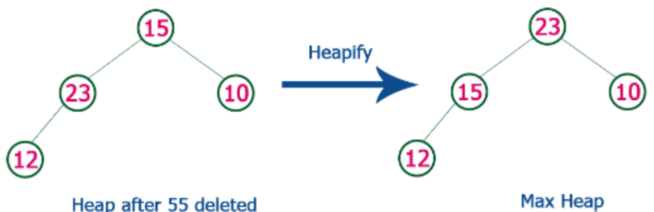
Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.

Heap after 77 deleted Max Heap

list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, **77**, **82**, **90**

Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.



Heap after 55 deleted Max Heap

list of numbers after swapping 55 with 15.

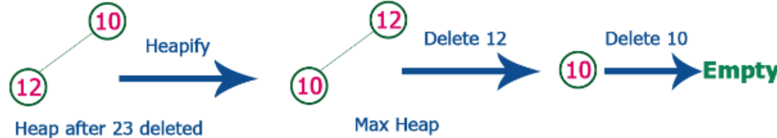
12, 15, 10, 23, **55**, **77**, **82**, **90**

Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.

list of numbers after swapping 23 with 12.

12, 15, 10, **23, 55, 77, 82, 90**

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after Deleting 15, 12 & 10 from the Max Heap.

10, 12, 15, 23, 55, 77, 82, 90

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

UNIT-V

Graphs: Introduction, Basic terminologies, Graph Representations, Bi-connected components, Topological sorting. **Hashing:** introduction to hashing and hash functions, basic implementation and operations of Hash tables, Caching, **Collision resolution techniques -** chaining and open addressing.

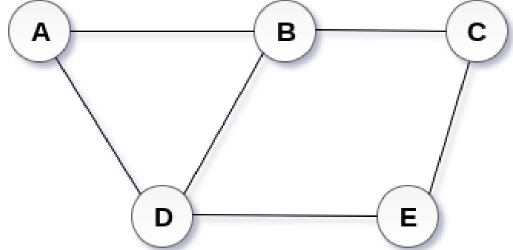
5.1 Graphs

A graph is a non-linear kind of data structure made up of nodes or vertices and edges. The edges connect any two nodes in the graph, and the nodes are also known as vertices.

Definition

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

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

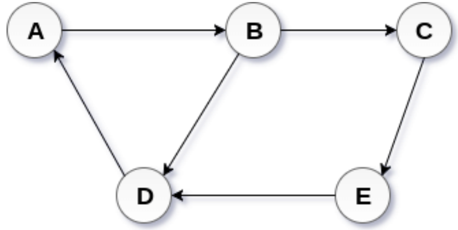


Undirected Graph

Directed and Undirected Graph

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

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



Directed Graph

5.2 Graph Terminology

Path

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

Connected Graph

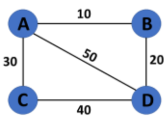
A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

Complete Graph

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

Weighted Graph

A graph G= (V, E) is called a labeled or weighted graph because each edge has a value or weight representing the cost of traversing that edge.



Digraph

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

Adjacent Nodes

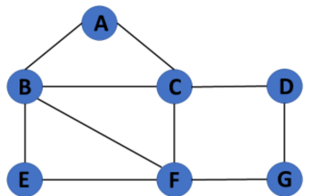
If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

Degree of the Node

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

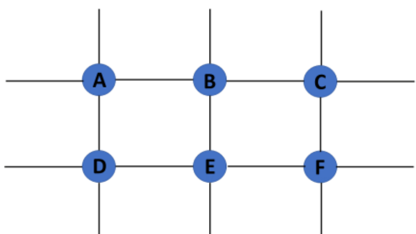
Finite Graph

The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is limited in number



Infinite Graph

The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is interminable.

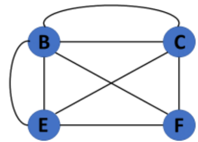


Trivial Graph

A graph G= (V, E) is trivial if it contains only a single vertex and no edges.

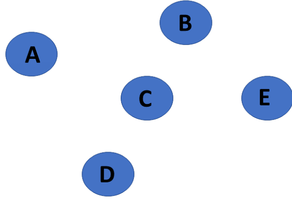
Multi Graph

If there are numerous edges between a pair of vertices in a graph G= (V, E), the graph is referred to as a multigraph. There are no self-loops in a Multigraph.



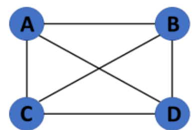
Null Graph

It's a reworked version of a trivial graph. If several vertices but no edges connect them, a graph G= (V, E) is a null graph.



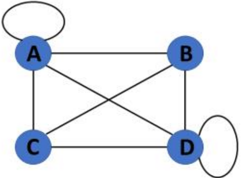
Complete Graph

If a graph G= (V, E) is also a simple graph, it is complete. Using the edges, with n number of vertices must be connected. It's also known as a full graph because each vertex's degree must be n-1.



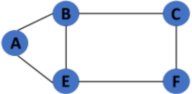
Pseudo Graph

If a graph G= (V, E) contains a self-loop besides other edges, it is a pseudograph.



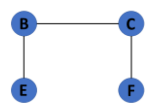
Cyclic Graph

If a graph contains at least one graph cycle, it is considered to be cyclic.



Acyclic Graph

When there are no cycles in a graph, it is called an acyclic graph.



5.3 Representation of Graphs in Data Structures

Graphs in data structures are used to represent the relationships between objects. Every graph consists of a set of points known as vertices or nodes connected by lines known as edges. The vertices in a network represent entities.

The most frequent graph representations are the two that follow:

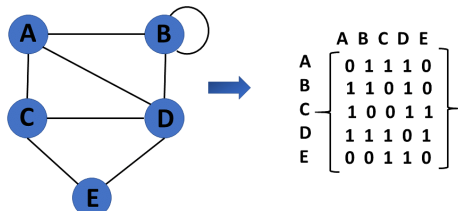
- Adjacency matrix
- Adjacency list

Adjacency matrix

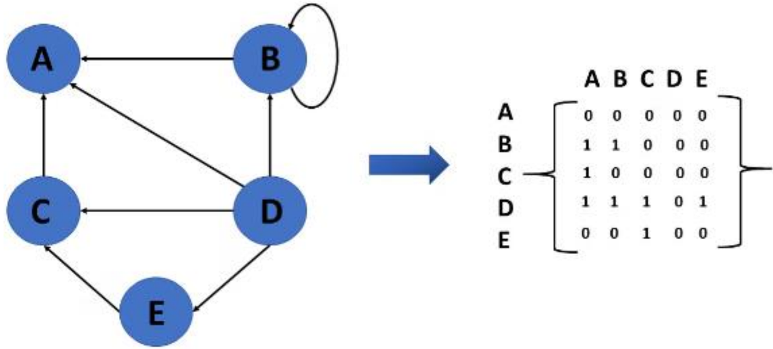
An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's). Let's assume there are n vertices in the graph So, create a 2D matrix adjMat[n][n] having dimension n x n.

- If there is an edge from vertex i to j, mark adjMat[i][j] as 1.
- If there is no edge from vertex i to j, mark adjMat[i][j] as 0.

Undirected Graph Representation

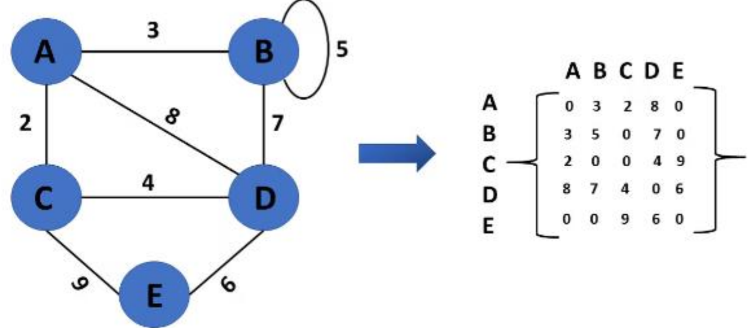


Directed Graph Representation



Weighted Undirected Graph Representation

Weight or cost is indicated at the graph's edge, a weighted graph representing these values in the matrix.

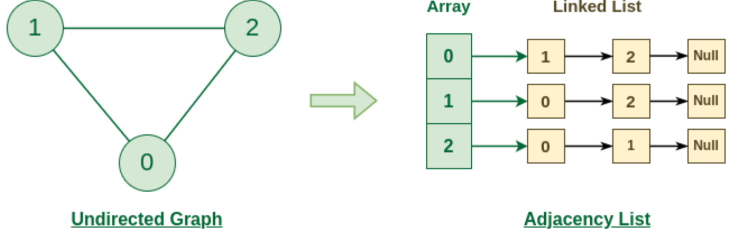


Adjacency List

- A linked representation is an adjacency list.
- You keep a list of neighbors for each vertex in the graph in this representation. It means that each vertex in the graph has a list of its neighboring vertices.

Representation of Undirected Graph to Adjacency list:

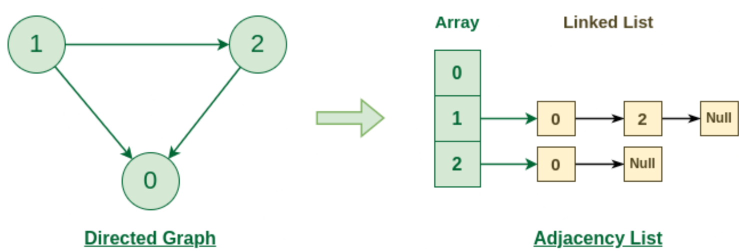
The below undirected graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has two neighbours (i.e, 1 and 2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex 1, it has two neighbour (i.e, 2 and 0) So, insert vertices 2 and 0 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



Graph Representation of Undirected graph to Adjacency List

Representation of Directed Graph to Adjacency list:

The below directed graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has no neighbours. For vertex 1, it has two neighbour (i.e, 0 and 2) So, insert vertices 0 and 2 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



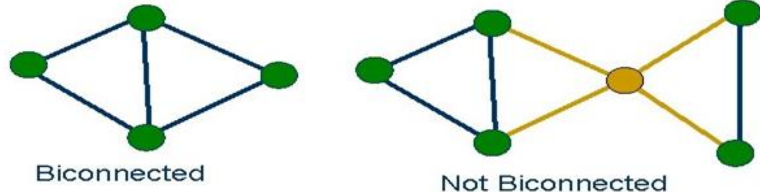
Graph Representation of Directed graph to Adjacency List

5.4 Bi-connected components

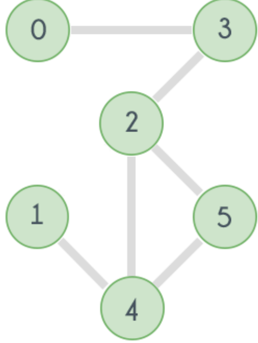
A graph is said to be Biconnected if:

- It is connected, i.e. it is possible to reach every vertex from every other vertex, by a simple path.
- Even after removing any vertex the graph remains connected.

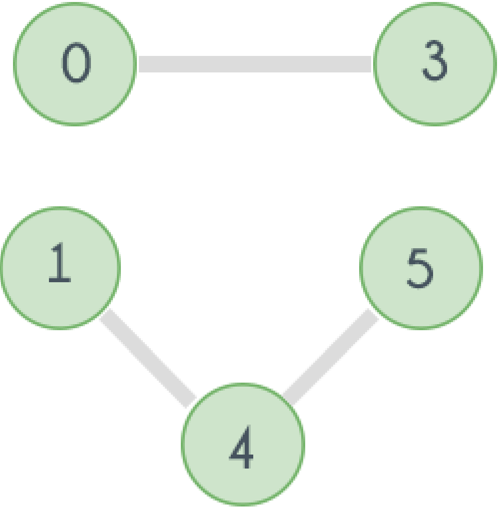
For example, consider the graph in the following figure



Removing any of the vertices does not increase the number of connected components. So the given graph is Biconnected.

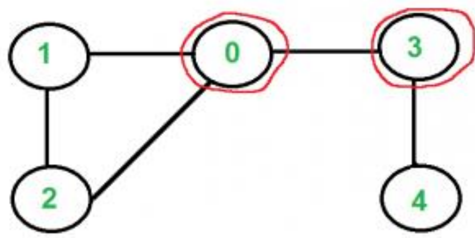


In the above graph if the vertex 2 is removed, then here's how it will look:



Articulation point or cut point

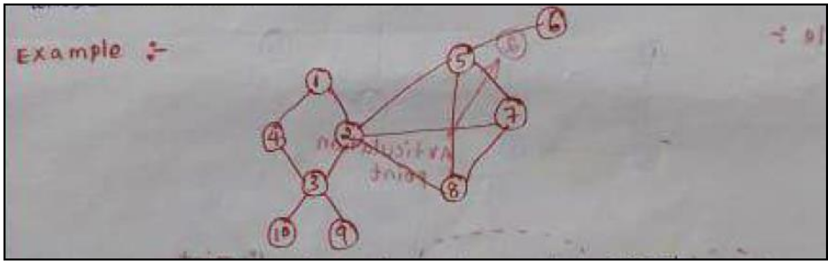
If a point in a graph becomes separated from the entire graph upon removal, it is referred to as an Articulation Point or Cut-Vertex.



Articulation points are 0 and 3

Construction of Bi-connected graph

- Check the given graph whether it is bi-connected or not.
- If the given graph is not bi-connected then identify all the articulation points.
- If articulation points exists, determine a set of edges whose inclusion makes the graph bi-connected.



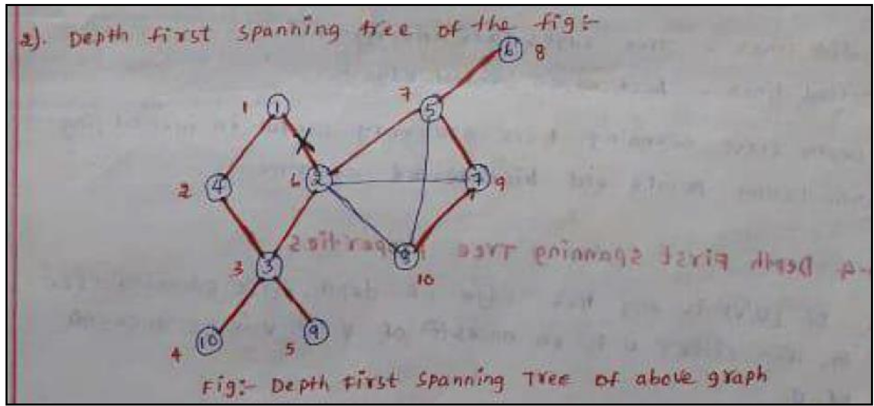
The articulation points are 2, 3, and 5

To transform the graph into bi-connected graph, the new edges are included corresponding to the articulation point.

Edges corresponding to the articulation point 3- (4,10)(10,9)

Edges corresponding to the articulation point 2-(1,5)(3,8)

Edges corresponding to the articulation point 5-(6,7)



In the fig: there is a number of outside each vertex, corresponds to the order in which a DFS visits these vertices and are named as depth first numbers(dfns) of the vertex ie

dfn[1] = 1
dfn[4] = 2
dfn[3] = 3
dfn[10] = 4
dfn[9] = 5
dfn[2] = 6
dfn[5] = 7
dfn[6] = 8
dfn[7] = 9
dfn[8] = 10

The solid edges of fig, will form a depth first spanning tree. The depth first spanning tree representation is

Solid lines: Tree edges

-Dotted lines: Back edges

-Depth first spanning trees are very useful in identifying articulation points and bi-connected components.

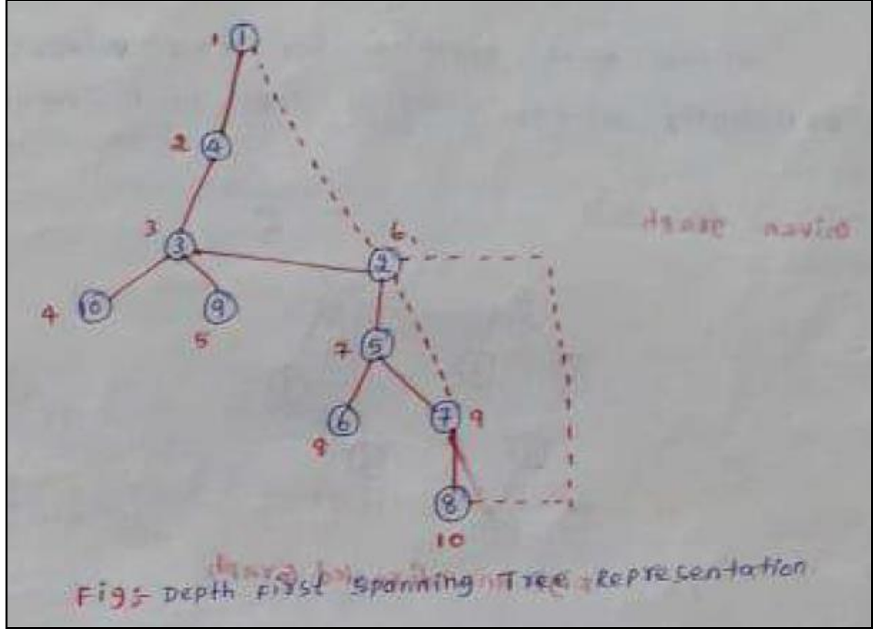


Fig:- Depth First Spanning Tree Representation

Depth First Spanning tree properties

2.7.4 Depth First Spanning Tree properties

1) If (u,v) is any tree edge of depth first spanning tree G, then either u is an ancestor of v or v is an ancestor of u.

Example :-

(4) u — (3) v

Fig:- A tree with (u,v) Edge

2) The root node of a depth first spanning tree is an articulation point iff it has atleast two children.

Example :-

(1) — (2), (3)

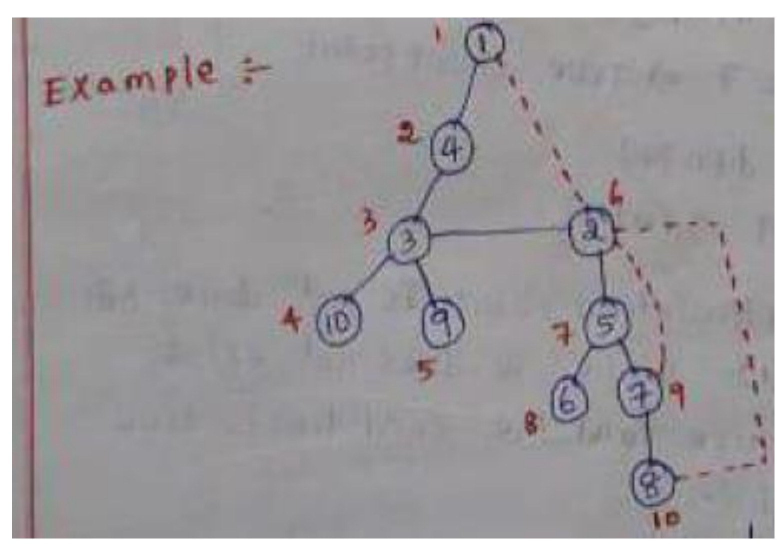
Fig:- A tree with Two children.

For each vertex, u, define L[u] as follows.

L[u] = min { dfn[u], min { L[w] | w is a child of u },
min { dfn[w] | (u,w) is a back edge } }

where L[u] is the lowest depth first number that can be reached from u using a path of descendants followed by atmost one back edge.

b) If u is an articulation point iff u has a child w such that L[w] ≥ dfn[u]



L[10] = min{4, -, -}
L[10] = 4
L[9] = min{5, -, -}
L[9] = 5
L[6] = min{8, -, -}
L[6] = 8
L[8] = min{10, -, 6}
L[8] = 6
L[7] = min{9, 6, 6}
L[7] = 6
L[5] = min{7, 6, -}
L[5] = 6.

L[2] = min { 6, 6, 1 }

L[2] = 1

L[3] = min { 3, 1, - }

L[3] = 1

L[4] = min { 2, 1, - }

L[4] = 1

L[1] = min { 1, 1, - }

L[1] = 1

(ie) L[1:10] = { 1, 1, 1, 1, 6, 8, 6, 6, 5, 4 }

For the spanning tree of fig: the articulation points are identified using the condition L[w] ≥ dfn[u], iff u has a child w and u is not the root.

① For vertex ② L[5] ≥ dfn[2]
6 ≥ 6 ⇒ True ⇒ cut point.

② For vertex ③ L[10] ≥ dfn[3]
4 ≥ 3 ⇒ True ⇒ cut point

③ For vertex ④ L[3] ≥ dfn[4]
1 ≥ 2 ⇒ false

④ For vertex ⑤ L[6] ≥ dfn[5]
8 ≥ 7 ⇒ True ⇒ cut point

⑤ For vertex ⑦ L[8] ≥ dfn[7]
6 ≥ 9 ⇒ false

The condition to check articulation point is not done for the vertices 6, 8, 9 and 10. Since w does not exist in the spanning tree structure and the condition is true for the vertices 2, 3 and 5. Therefore the articulation points are 2, 3 and 5.

5.5 Graph Traversal

The process of visiting or updating each vertex in a graph is known as graph traversal.

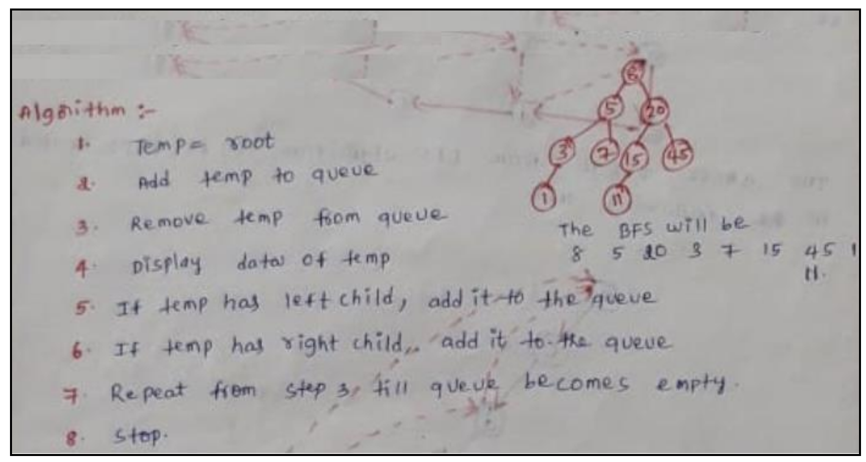
There are two techniques to implement a graph traversal algorithm:

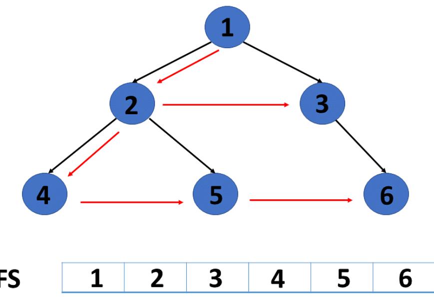
- Breadth-first search
- Depth-first search

Breadth-first search(BFS)

We traverse the tree level wise from left to right starting from the root. It is implemented using a queue.

Algorithm



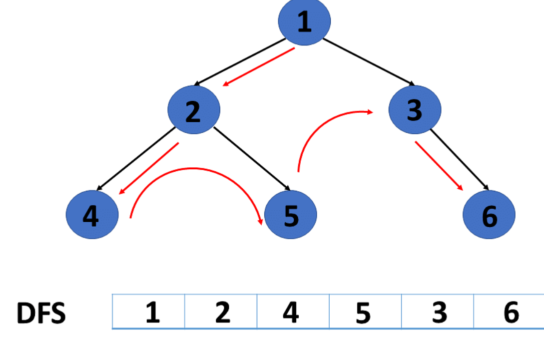


Depth-first search(DFS)

In the DFS traversal method, we start from the root and traverse left as far as we can go once the leftmost end is reached, we go to the right child of the node in the path and move to its left most end.

Algorithm for DFS :-

1. Temp = root
2. push temp into the stack
3. pop temp from stack
4. Display data of temp.
5. If temp has right child, push right child into the stack
6. If temp has left child, push left child into the stack
7. Repeat from step 3 till stack is empty.
8. Stop



5.6 Topological sorting

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u-v, vertex u comes before v in the ordering.

Note: Topological Sorting for a graph is not possible if the graph is not a DAG.

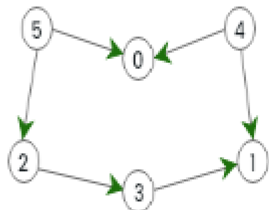
Topological Sorting vs Depth First Traversal (DFS):

Depth First Traversal	Topological Sorting
In DFS, we print a vertex and then recursively call DFS for its adjacent vertices.	In topological sorting, we need to print a vertex before its adjacent vertices.

Algorithm for Topological Sorting:

- We use a temporary stack.
- We don't print the vertex immediately,
- We first recursively call topological sorting for all its adjacent vertices, then push it to a stack.
- Finally, print the contents of the stack.

Example



Sol:

Adjacency List (G)

0	{}
1	{}
2	{3}
3	{1}
4	{0,1}
5	{2,0}

	0	1	2	3	4	5
Visited	False	False	False	False	False	False

Step 1 : Topological Sort (0), Visited[0] = True

↓

List Is Empty. No More Recursion Call

Stack | 0 |

Step 2 : Topological Sort (1), Visited[1] = True

↓

List Is Empty. No More Recursion Call

Stack | 0 | 1 |

Step 3 : Topological Sort (2), Visited[2] = True

↓

Topological Sort (3), Visited[3] = True

↓

'1 Is Already Visited, No More Recurrsion Call

Stack | 0 | 1 | 3 | 2 |

Step 4 : Topological Sort (4), Visited[4] = True

'0','1' Is Already Visited, No More Recurrsion Call

Stack | 0 | 1 | 2 | 3 | 4 |

Step 5 : Topological Sort (5), Visited[5] = True

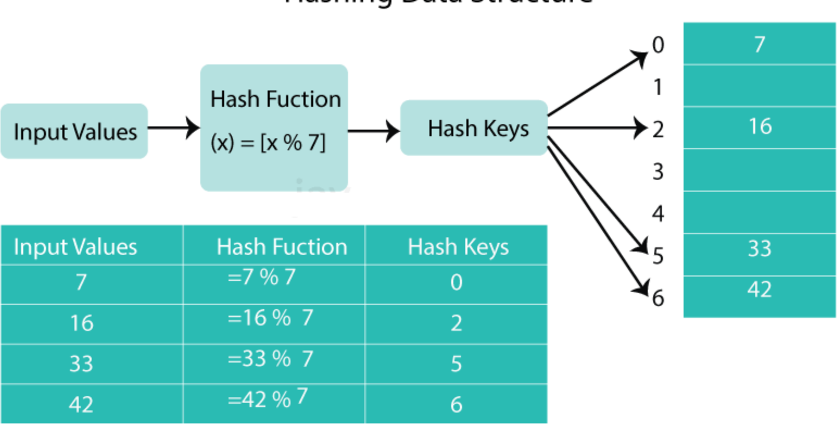
'2', '0' Are Already Visited, No More Recurrsion Call

Stack | 0 | 1 | 2 | 3 | 4 | 5 |

Step 6 : Print All Elements Of Stack From Top To Bottom

5.7 Hashing

Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.



Input Values	Hash Fuction	Hash Keys
7	=7 % 7	0
16	=16 % 7	2
33	=33 % 7	5
42	=42 % 7	6

Hash Functions

There are three ways of calculating the hash function:

a) Division method

b) Folding method

c) Mid square method

a) Division method

In the division method, the hash function can be defined as:

$h(k_i) = k_i \% m$;

where **m** is the size of the hash table.

For example, if the key value is 6 and the size of the hash table is 10. When we apply the hash function to key 6 then the index would be:

$h(6) = 6\%10 = 6$

The index is 6 at which the value is stored.

b) Multiplication method

The hash function used for the multiplication method is −

```
h(k) = floor( n( kA mod 1 ) )
```

Here, k is the key and A can be any constant value between 0 and 1. Both k and A are multiplied and their fractional part is separated. This is then multiplied with n to get the hash value. An example of the Multiplication Method is as follows –

```
k=123
n=100
A=0.618033
h(123) = 100 (123 * 0.618033 mod 1)
= 100 (76.018059 mod 1)
= 100 (0.018059)
= 1
```

c) Mid Square Method

The mid square method is a very good hash function. It involves squaring the value of the key and then extracting the middle r digits as the hash value. The value of r can be decided according to the size of the hash table. An example of the Mid Square Method is as follows –

```
k = 50
k*k = 2500
h(50) = 50

The hash value obtained is 50
```

K=	3205	7148	2345
K2=	10272025	51093904	5499025
H(K)=	72	93	99

Operations of Hash tables

The primary operations of a hash table are **search**, **insert,** and **delete.** The search operation is used to locate the element within a hash table. The insert operation is used to add elements into the hash table. The delete operation removes elements from the hash table.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define SIZE 20
struct DataItem {
   int data;
   int key;
};
struct DataItem* hashArray[SIZE];
struct DataItem* dummyItem;
struct DataItem* item;

int hashCode(int key) {
   return key % SIZE;
}
struct DataItem *search(int key) {
   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] != NULL) {

      if(hashArray[hashIndex]->key == key)
```

```
      return hashArray[hashIndex];

    //go to next cell
    ++hashIndex;

    //wrap around the table
    hashIndex %= SIZE;
  }
  return NULL;
}

void insert(int key,int data) {
  struct DataItem *item = (struct DataItem*)
malloc(sizeof(struct DataItem));
  item->data = data;
  item->key = key;

  //get the hash
  int hashIndex = hashCode(key);

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

```
  int hashIndex = hashCode(key);
  //move in array until an empty
  while(hashArray[hashIndex] != NULL) {

    if(hashArray[hashIndex]->key == key) {
      struct DataItem* temp = hashArray[hashIndex];

      //assign a dummy item at deleted position
      hashArray[hashIndex] = dummyItem;
      return temp;
    }
    //go to next cell
    ++hashIndex;

    //wrap around the table
    hashIndex %= SIZE;
  }
  return NULL;
}
void display() {
  int i = 0;

  for(i = 0; i<SIZE; i++) {
    if(hashArray[i] != NULL)
      printf("(%d,%d)",hashArray[i]->key,hashArray[i]-
>data);
  }

  printf("\n");
}
int main() {
  dummyItem = (struct DataItem*) malloc(sizeof(struct
DataItem));
```

```
   dummyItem->data = -1;
   dummyItem->key = -1;
   insert(1, 20);
   insert(2, 70);
   insert(42, 80);
   insert(4, 25);
   insert(12, 44);
   insert(14, 32);
   insert(17, 11);
   insert(13, 78);
   insert(37, 97);
   printf("Insertion done: \n");
   printf("Contents of Hash Table: ");
   display();
   int ele = 37;
   printf("The element to be searched: %d", ele);
   item = search(ele);
   if(item != NULL) {
      printf("\nElement found: %d\n", item->key);
   } else {
      printf("\nElement not found\n");
   }
   delete(item);
   printf("Hash Table contents after deletion: ");
   display();
}
```

5.8 Caching

Caching is the process of storing data in a cache, which is a temporary storage area that facilitates faster access to data with the goal of improving application and system performance.

A common example of caching is a web browser that stores page content on a local disk for a designated period of time. When the user first visits the website, the content is downloaded from the web server and saved to a local directory. If the user revisits the website, the content comes from the local cache rather than the server. In this way, page content loads much faster into the browser than it would if it were downloaded from the web server. This saves the user time, reduces network traffic and minimizes the load on the web server.

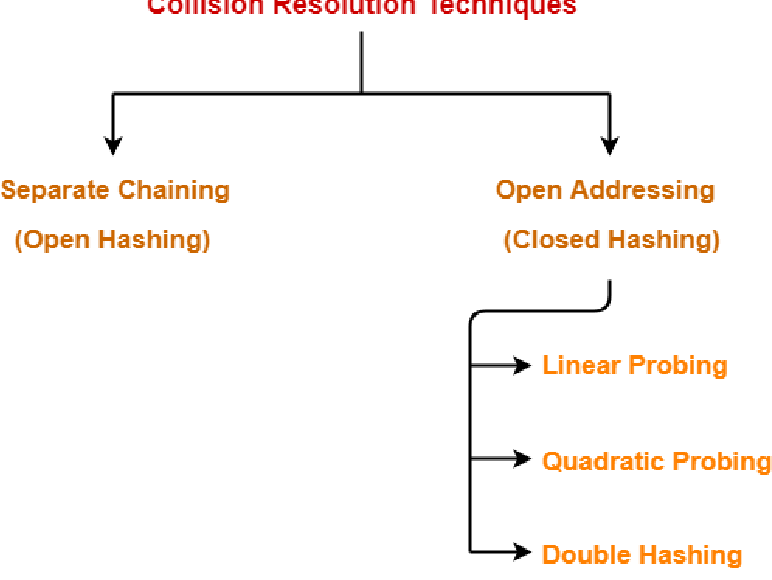
The idea behind caching is to temporarily copy data to a location that enables an application or component to access the data faster than if retrieving it from its primary source.

The uses of caching data

- Helps speed up application performance and increase efficiency
- Store data locally
- Improve user experience and encourage people to use their site.

5.9 Collision Resolution Techniques

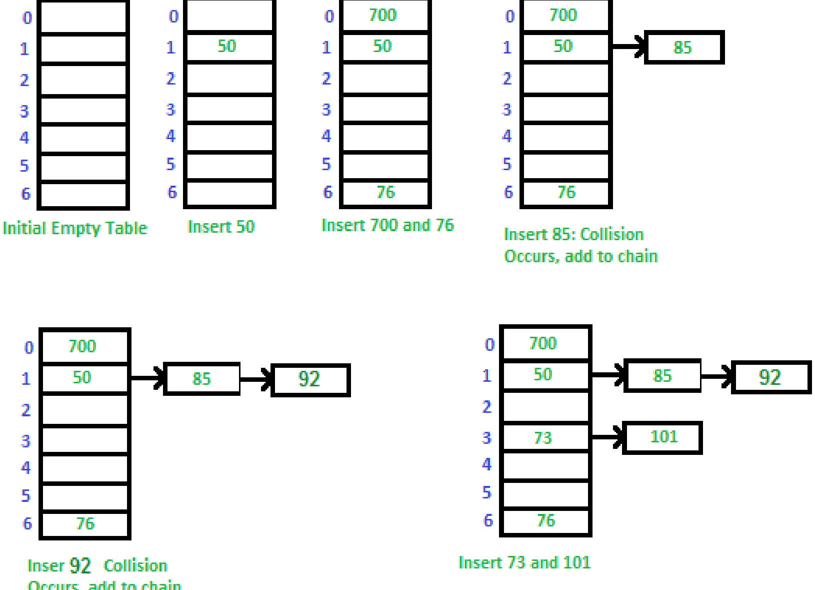
The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.



Separate chaining

- This technique creates a linked list to the slot for which collision occurs.
- The new key is then inserted in the linked list.
- These linked lists to the slots appear like chains.
- That is why, this technique is called as separate chaining.

Example: Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101



Advantages

- Simple to implement.
- Hash table never fills up, we can always add more elements to the chain.
- Less sensitive to the hash function or load factors.
- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages

- The cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
- Wastage of Space (Some Parts of the hash table are never used)
- If the chain becomes long, then search time can become O(n) in the worst case
- Uses extra space for links

Closed Hashing (Open Addressing)

This collision resolution technique requires a hash table with fixed and known size. During insertion, if a collision is encountered, alternative cells are tried until an empty bucket is found. These techniques require the size of the hash table to be supposedly larger than the number of objects to be stored (something with a load factor < 1 is ideal).

There are various methods to find these empty buckets:

- Liner Probing
- Quadratic probing
- Double hashing

Liner Probing

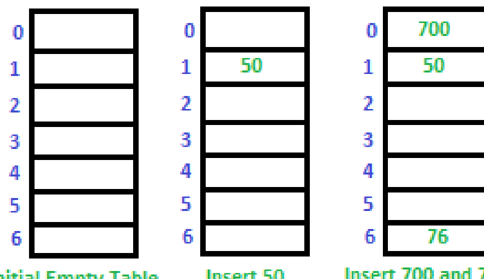
In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

The function used for rehashing is as follows:

rehash(key) = (n+1)%tablesize

Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101

which means hash(key)= key% S, here S=size of the table =7,indexed from 0 to 6.We can define the hash function as per our choice if we want to create a hash table, although it is fixed internally with a pre-defined formula.



0	700
1	50
2	85
3	
4	
5	
6	76

Insert 85: Collision Occurs, insert 85 at next free slot.

0	700
1	50
2	85
3	92
4	
5	
6	76

Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot

0	700
1	50
2	85
3	92
4	73
5	101
6	76

Insert 73 and 101

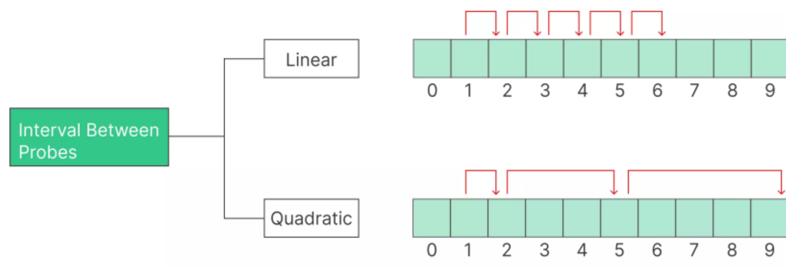
Quadratic probing

Quadratic probing is a method to resolve collisions that can occur during the insertion of data into a hash table. When a collision takes place (two keys hashing to the same location), quadratic probing calculates a new position by adding successive squares of an incrementing value (usually starting from 1) to the original position until an empty slot is found.

How Quadratic Probing Works

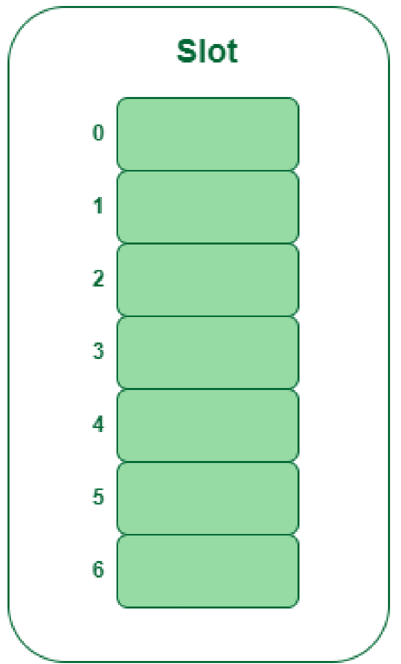
Quadratic probing involves three main steps:

- Calculate the initial hash position for the key.
- If the position is occupied, apply the quadratic probing formula to find the next available slot. **(hash(key) + i^2) % table_size**,
- Repeat this process until an empty slot is found, and insert the data.



Example: Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be f(i) = i2 . Insert = 22, 30, and 50.

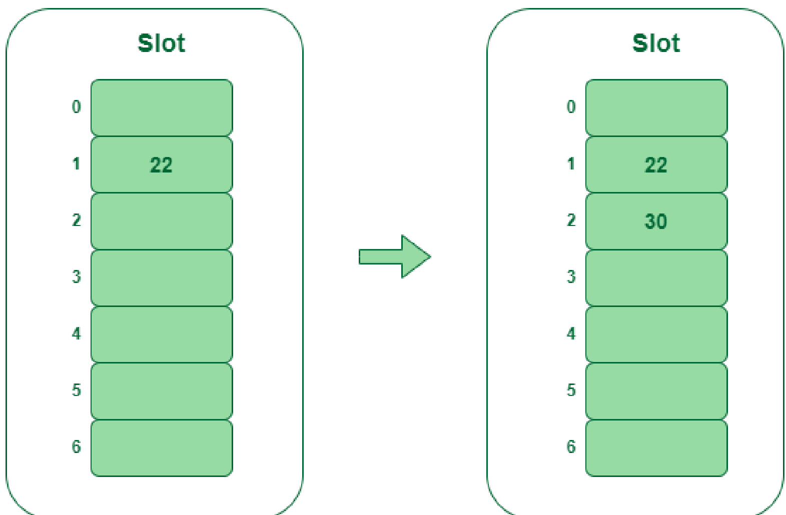
Step 1: Create a table of size 7.



Step 2 – Insert 22 and 30

Hash(22) = 22 % 7 = 1, Since the cell at index 1 is empty, we can easily insert 22 at slot 1.

Hash(30) = 30 % 7 = 2, Since the cell at index 2 is empty, we can easily insert 30 at slot 2.



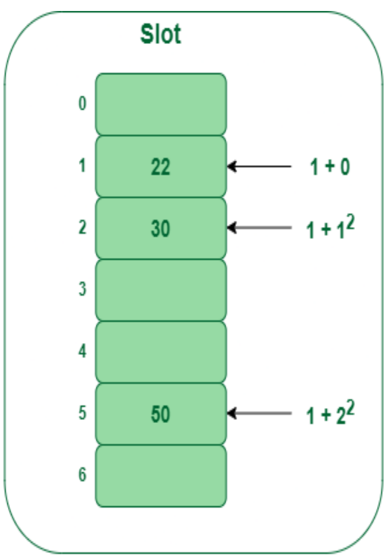
Step 3: Inserting 50

Hash(50) = 50 % 7 = 1

In our hash table slot 1 is already occupied. So, we will search for slot 1+12, i.e. 1+1 = 2,

Again slot 2 is found occupied, so we will search for cell 1+22, i.e.1+4 = 5,

Now, cell 5 is not occupied so we will place 50 in slot 5.



Double Hashing

In double hashing, we make use of two hash functions. The first hash function is $h1$(k), this function takes in our key and gives out a location on the hash-table. If the new location is empty, we can easily place our key in there without ever using the secondary hash function.

However, in case of a collision, we need to use secondary hash-function $h2$(k) in combination with the first hash-function $h1$(k) to find a new location on the hash-table. The combined hash-function used is of the form

h(k,i) = ($h1$(k) +i * $h2$(k))%m.

Here, i is an non-negative integer which signifies the collision number, k = element/key which is being hashed and m= hash table size.

Method 1:

- First hash function is typically hash1(key) = key % TABLE_SIZE
- A popular second hash function is hash2(key) = PRIME – (key % PRIME) where PRIME is a prime smaller than the TABLE_SIZE.

A good second Hash function is:

- It must never evaluate to zero
- Just make sure that all cells can be probed

Example

Table Size = 10 elements
$Hash_1(key) = key \% 10$
$Hash_2(key) = 7 – (k \% 7)$

Lets say, Hash1 (key) = key % 13

Hash2 (key) = 7 – (key % 7)

Hash1(19) = 19 % 13 = 6

Hash1(27) = 27 % 13 = 1

Hash1(36) = 36 % 13 = 10

Hash1(10) = 10 % 13 = 10

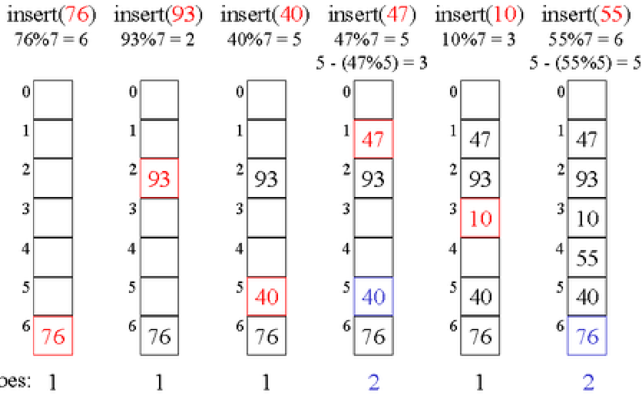
Hash2(10) = 7 – (10%7) = 4

(Hash1(10) + 1*Hash2(10))%13= 1

(Hash1(10) + 2*Hash2(10))%13= 5

Collision

Example



Practice Problem Based on Double Hashing

Problem Statement 1:

Given the two hash functions,

$h1$ (k) = k mod 23 and $h2$(k) = 1 + k mod 19. Assume the table size is 23. Find the address returned by double hashing after 2nd collision for the key = 90.

Solution:

We will use the formula for double hashing-

h(k,i) = ($h1$(k) + i * $h2$(k))%m

As it is given, k = 90, m = 23

Since the 2nd collision has already occurred, i = 2.

Substituting the values in the above formula we get,

h(90,2) = [(90 % 23) + 2 * (1 + 90 % 19)] % 23

= [21 + 2 * 15] % 23

= 5

Hence after the second collision, the address returned by double hashing for Key = 90 is 5.

Time Complexity:

Insertion: O(n)

Search: O(n)

Deletion: O(n)

Advantages of Double Hashing

- No extra space is required.
- Primary Clustering does not occur.
- Secondary Clustering also does not occur.
- It can be used with different types of hash functions.
- It reduces the clustering effect of keys and decreases the chance of keys being stored in consecutive buckets.

Disadvantages of Double Hashing

- Poor cache performance.
- It's a little complicated because it uses two hash functions.
- It has a bit harder implementation.
- It consumes more memory than other collision resolution techniques.
- It may not be the best choice for small tables.