KINGS
ENGINEERING COLLEGE

An Autonomous Institution

Affiliated to Anna University, Chennai

Department of Computer Science and Engineering

Regulation 2024

II Year – III Semester

CS242301 – Data Structures

CS242301 – DATA STRUCTURES

REGULATION – 2024

II YEAR / III SEMESTER

CS242301	DATASTRUCTURES	L	T	P	C
		3	0	0	3

COURSE OBJECTIVES:

- To understand the concepts of ADTs.
- To Learn linear data structures–lists, stacks, and queues.
- To understand non-linear data structures–trees and graphs.
- To understand sorting, searching and hashing algorithms.
- To apply Tree and Graph structures.

UNIT I LISTS 9

Abstract Data Types (ADTs) – List ADT-Array-based implementation Linked list implementation- Singly linked lists–Circularly linked lists–Doubly-linked lists Applications of lists–Polynomial ADT– Radix Sort– Multi lists.

UNIT II STACKSANDQUEUES 9

StackADT–Operations–Applications–Balancing Symbols–Evaluating arithmetic expressions Infix to Postfix conversion – Function Calls – Queue ADT – Operations – Dequeue – Applications of Queues.

UNIT III TREES 9

Tree ADT – Tree Traversals – Binary Tree ADT – Expression Trees – Binary Search Tree ADT – AVL Trees - Priority Queue(Heaps) – Binary Heap – Applications.

UNIT IV GRAPHS 9

Graph Definition – Representation of Graphs – Types of Graph – Graph Traversals– Bi-connectivity – Topological Sort -Shortest Path Algorithm -Dijkstra's algorithm – Minimum Spanning Tree – Prim's algorithm – Kruskal's algorithm.

UNIT V SEARCHING, SORTING AND HASHING TECHNIQUES 9

Searching–Linear Search–Binary Search. Sorting–Bubblesort–Selection sort -Insertion sort – Merge Sort – Hashing–Hash Functions – Separate Chaining – Open Addressing –Rehashing. Case Study: Tree Data structure

TOTAL: 45 PERIODS

COURSE OUTCOMES:

At the end of this course, the students wilable to:
CO1: Define linear and non-linear data structures.
CO2: Implement linear and non–linear data structure operations.
CO3: Use appropriate linear/non–linear data structure operations for solving a given problem.

CO4: Apply appropriate graph algorithms for graph applications.
CO5: Analyze the various searching and sorting algorithms.

TEXT BOOKS:

1. Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd Edition, Pearson Education, 2005.
2. Kamthane, Introduction to Data Structures in C, 1st Edition, Pearson Education, 2007

REFERENCES:

1. Langsam, Augenstein and Tanenbaum, Data Structures Using C and C++, 2nd Edition, Pearson Education, 2015.
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms", Fourth Edition, McGraw Hill/ MIT Press, 2022.
3. Alfred V. Aho, Jeffrey D. Ullman, John E. Hopcroft ,Data Structures and Algorithms, 1st edition, Pearson, 2002.
4. Kruse, Data Structures and Program Design in C, 2nd Edition, Pearson Education, 2006.

CO's-PO's & PSO's MAPPING

CO's	PO's												PSO's		
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**1**	**2**	**3**
1	2	3	1	2	2	-	-	-	-	-	-	-	2	-	-
2	1	2	1	2	2	-	-	-	-	-	-	-	2	-	-
3	2	3	1	2	3	-	-	-	-	-	-	2	2	-	-
4	2	1	-	1	1	-	-	-	-	-	-	2	2	-	-
5	1	2	1	2	2	-	-	-	-	-	-	2	2	-	-
AVg.	**2**	**2**	**1**	**2**	**2**	**-**	**-**	**-**	**-**	**-**	**-**	**2**	**2**	**-**	**-**

1 - low, 2 - medium, 3 - high, '-' - no correlation

COURSE OBJECTIVES:

- To understand Object Oriented Programming concepts and basics of Java programming language
- To know the principles of classes and inheritance.
- To access the concepts of packages and interfaces.
- To define exceptions and use I/O streams
- To develop a java application with threads and generics classes

UNIT I- LIST

Abstract Data Types (ADTs) – List ADT – Array-based implementation – Linked list implementation – Singly linked lists – Circularly linked lists – Doubly-linked lists – Applications of lists – Polynomial ADT– Radix Sort – Multilists.

Data Structure:

Data structure is basically a group of data elements that are put together under one name. It also defines a particular way of storing and organizing data in a computer, so that it can be used efficiently.

Data Structure is the structural representation of logical relationships between data or element. It represents the way of storing, organizing and retrieving data.

It is classified as

- **Linear data structure**
- **Non Linear data structure**

Examples:

Linear : Lists, Stacks, Queues etc.

Non Linear : Trees, graphs etc.

Normally all the data structures can be implemented using 2 methods

1. Array implementation
2. Linked List implementation

Applications of data structure:

- Compiler design
- Statistical analysis package
- Numerical Analysis
- Artificial Intelligence
- Operating System
- Data base management system
- Simulation
- Graphics

Abstract Data Type:

Abstract data type is a set of operations of data. It also specifies the logical and mathematical model of the data type.ADT specification includes description of the data, list of operationsthat can be carried out on the data and instructions how to use these instructions. But ADT does not mention how the set of operations is implemented.

Examples for ADT: lists, sets and graphs along with their operations for ADT.

Examples for data type: Integers, reals and Booleans.

List ADT:

A list is a collection of elements. It can be represented as A_1,A2,A3 ..... A_N and its size is N.A list with size 0 is called as an empty list.

For any list except the empty list, we say that A_{i+1} follows A_i ($i<N$) and A_{i-1} precedes A_i($i>1$). The first element of the list is A_1 and the last element is A_N. the position of A_i in a list is i.

Some popular operations on the list are

- **Find** which returns the position of the first occurrence of an element or key.
- **Insert** which inserts an element in a given position.
- **Delete** which deletes an element from the list.
- **Find Kth** which returns the element in Kth position.

Example: Consider 34,12,52,16,12 as a list , then

Find (52) returns 3

After **Insert (70,3)** the list is 34,12,70,52,16,12

After **Delete (52)** the list is 34,12,70,16,12

IMPLEMENTATION OF LIST ADT:

There are 2 ways of implementation

- Using Arrays
- Using Linked list

1. List ADT –Array based Implementation:

A list can be implemented using an array. An array is a collection of similar data items or elements. In the array implementation maximum size of the array is required earlier. That is the maximum number of elements in the array is fixed and it can be modified only through the code.

Since the maximum size should be known earlier, required high estimate, which wastes considerable space.

- An Array implementation allows **printlist** and **Find** to be carried out in **linear time**.
- **Find kth**takes **constant time**.
- **Insertion** and **Deletion** are expensive.

For Example inserting an element at position 0 (i.e) first positions requires shifting the entire elements one position to right.

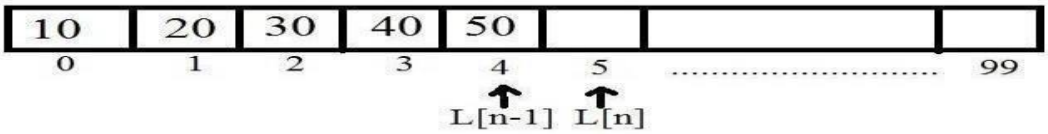
For deleting the first element requires shifting all the elements one position to the left.

Operations:

- Insertion -This inserts a given element X in to the List.
  - Insertion at First.
  - Insertion at any position.
  - Insertion at last.
- Deletion -> deletes a given element from the list
- Find - This returns the position of the first occurrence of an element orkey.
- Find kth -returns the element in the kth position.

Consider an array L[100] and insert ‘n’ elements into the

array. Assume n is 5 and elements are10,20,30,40,50

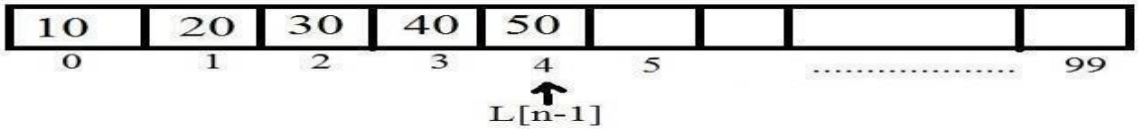


The elements are placed as above and the last element present at location L[n-1] (i.e) L[4] .

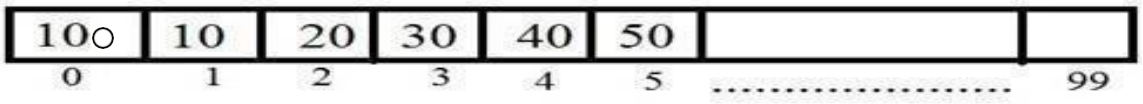
INSERTION

a) Insertion at First:

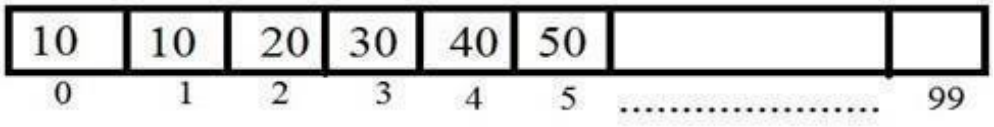
Consider a new element 'X' is to be inserted at the first position (i.e) at L[0]. For this ,shift all the elements to the right one bit position. Let us start shifting from the lastlocation(i.e) L[n-1].



After shifting the array looks like as below.



Now inserts the new element in to first location that is L[0]=X; Assume X=100,Then



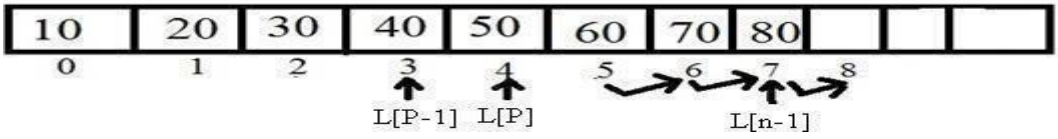
After insertion the no of elements in the list will be increased by one. Therefore n will be updated as n+1 .

Routine:

```
void InsertFirst(int L[], int x, int n)
{
int n;
     for(i=n-1;i>=0;i--) //shifting the
          L[i+1]=L[i];    //elements
L[0]=x;
    n=n+1;
}
```

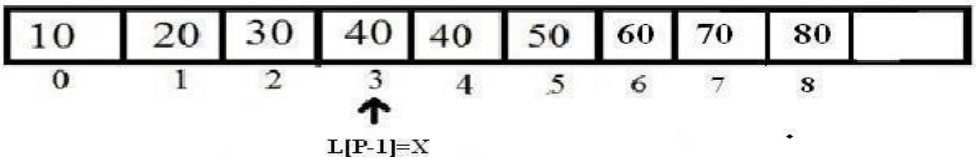
b) Inserting at any position:

Insert an element 'X' at position 'p' , the elements from the next position will be shifted to the right and then the new element will be inserted. Assume p=4 (i.e) 4th position. In array 4th is L[3] .

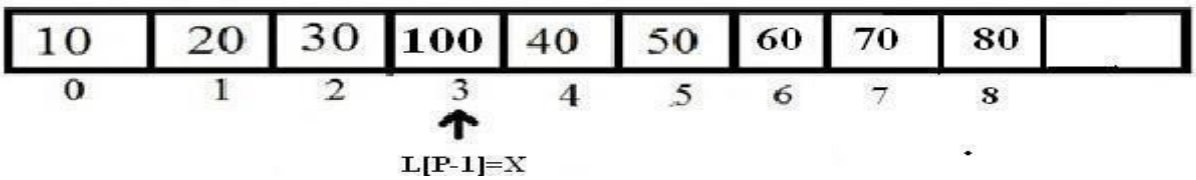


Shift all the elements from p-1th location upto L[n-1].

After shifting, the array looks like as below



Now X can be inserted at position L[p-1] i.e. L[p-1]=X and update the n value i.e. n+1 .Assume the value of X=100,Then the array is

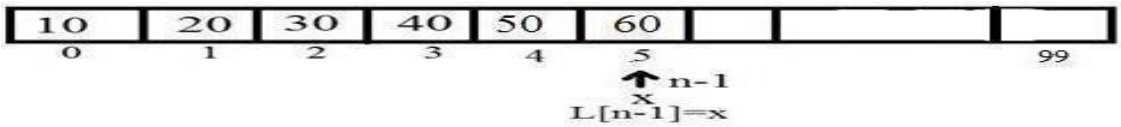


Routine :

```
void InsertAtAny(int L[] , int x , int p , int n)
{
   for(i=n-1;i>p-1;i--)
   {L[i+1]=L[i];
   }
   L[p-1]=x;
   n=n+1;
}
```

c) Insertion at Last:

To insert an element at last no shifting is required. Just assign the new element to the position L[n-1].

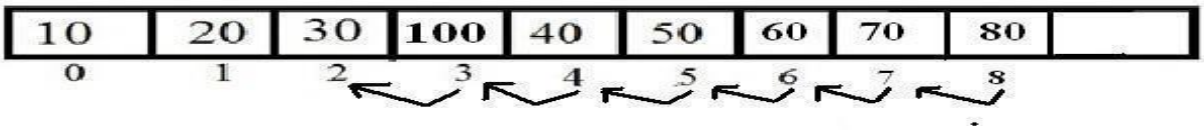


Routine:

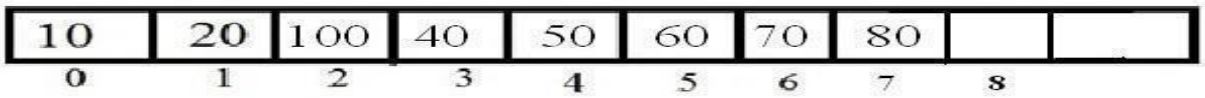
```
void InsertLast(int L[] , int x , int n )
{
   L[n]=x;
   n=n+1;
}
```

DELETION:

In deletion a given element will be removed from the list. For this initially the element is searched and the location of the element is identified, then the element is removed by shifting all the elements to the left from next location of the element to be deleted to the next location of the element to be deleted. Assume the element to be deleted 30,



After deletion,



Update the value of n (i.e) n-1.

```
void Delete(int L[] , int x , int n)
{
   for(i=0;i<n;i++)
       if(L[i]= =x)
           break;
  for(j=i+1;j<n;j++)
        L[j-1]=L[j];
  n=n-1;
}
```

FIND:

This operation checks whether the given element is present or not .

```
void Find(int L[] , int x , int n)
{
int flag=0;
for(i=0;i<n;i++)
        if(L[i]==x)
  flag=1;
if(flag==1)
  printf("The element is present");
else
  printf("The element is Not Present");

}
```

Issues in Array:

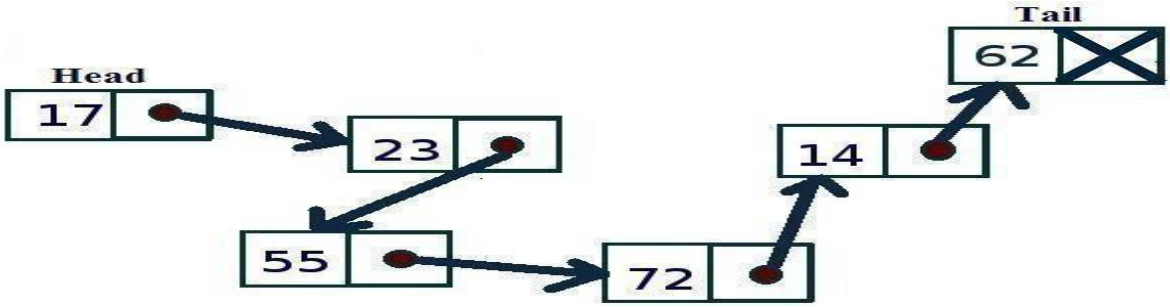
- Fixed size: Resizing is expensive
- Requires continuous memory for allocation
  – Difficult when array size is larger
- Insertions and deletions are inefficient
  – Elements are usually shifted
- Wastage of memory space unless the array is full

2. LINKED LIST based implementation:

Linked List is a list, which is implemented using structures and pointers. It can also be defined as group nodes or series of structures where each node has minimum 2 fields.

- **Data field :** Stores the actual information
- **Address field :**Points the next node

The link field of the last node points to NULL which indicates end of linked list. In order to avoid the linear cost of insertion and deletion, the elements are not stored contiguously in linked list.



Node Declaration

```
struct node
{
int data;
struct node *next;
};
```

Operations

- Insertion
- Deletion
- Find

Advantages:

- Linked lists allow dynamic memory management by allowing elements to be added or deleted at any time during program execution.
- Efficient utilization of memory space. Memory space is reserved only for the elements in a linked list.
- Easy to insert or delete elements in a linked list.

Disadvantages:

- Each element in the linked list requires more space when compared witharray.
- Accessing an element is more difficult, since to access an element it is mandatory to traverse all the preceding elements.

Types of Linked List:

3 types

- Singly Linked List
- Doubly Linked List
- Circularly Linked List

1. SINGLY LINKED LIST

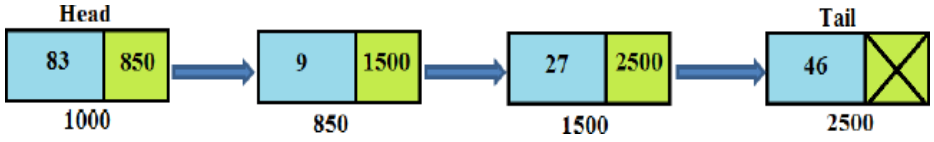
Singly linked list consists of a series of structures or nodes. Each node contains only 2 fields.

Data field: Actual information

Address field: Address of next node

Global declaration of a Node:

```
struct node
{
int data;
struct node *next;
} *head, *tail, *n, *t;
```



OPERATIONS

- Insertion
- Deletion
- Find

INSERTION

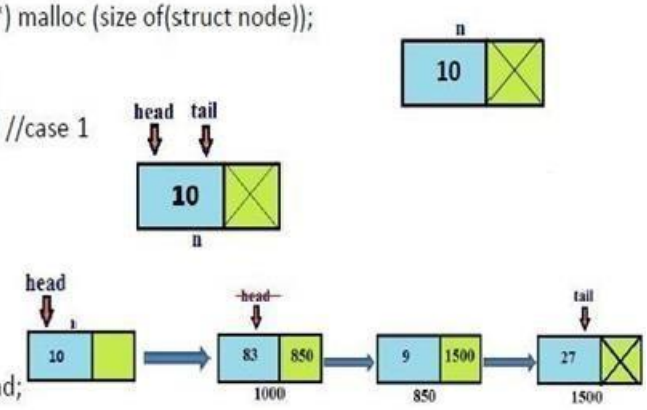
Insertion is the process of inserting node at the given position in the linked list. A new node can be inserted after node P. Here, P is the address of the node after which the new node is to be inserted.

For insertion, create a node first from Node structure and assign the value X to its data field. Insertion can be done at any position

a. Insertion atfirst
b. Insertion at anyposition
c. Insertion atlast

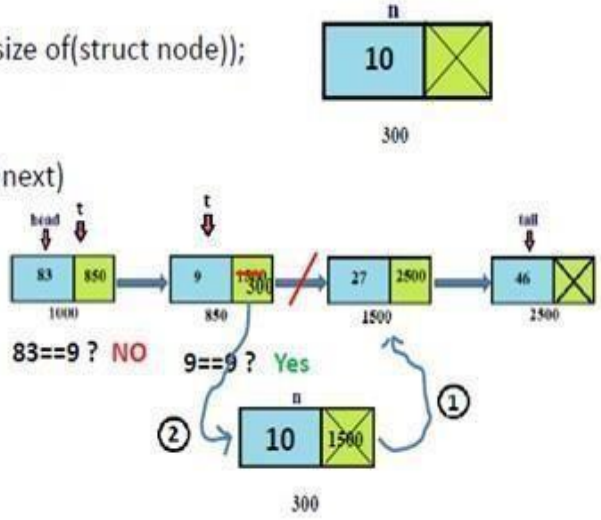
a) INSERTION AT FIRST

```
void ins_beg (int num) //Let num=10
{
 n=(struct node *) malloc (size of(struct node));
 n→data=num;
 n→next=NULL;
 if (head==NULL) //case 1
 {
   head=n;
   tail=n;
 }
 else //case 2
 {
   n→next=head;
   head=n;
 }
}
```



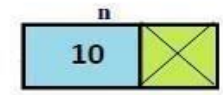
b) INSERTION AT MIDDLE

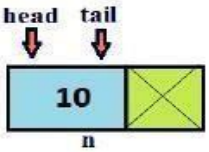
```
void ins_mid (int num, int mid_data) //Let num=10, mid_data=9
{
  n=(struct node *) malloc (size of(struct node));
  n→next=NULL;
  n→data=num;
  for(t=head; t!=NULL; t=t→next)
  {
    if(t→data==mid_data)
        break;
  }
  n→next=t→next;
  t→next=n;
 }

  head=n;
}
```

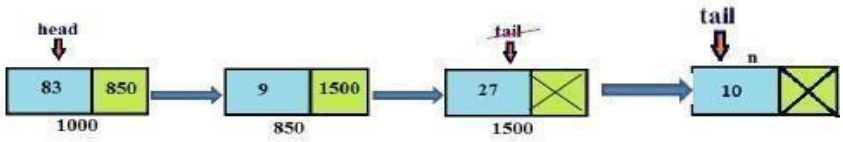


c) INSERTION AT END

```
void ins_end (int num) //Let num=10
{
  n=(struct node *) malloc (size of(struct node));
  n→next=NULL;
  n→data=num;
  if (head==NULL) //case 1
   {
     head=n;
     tail=n;
   }
   else //case 2
   {
     tail→next=n;
     tail=n;
   }
}
```







ROUTINE

Routine to Insert at First	Routine to Insert at Last	Routine to insert at middle

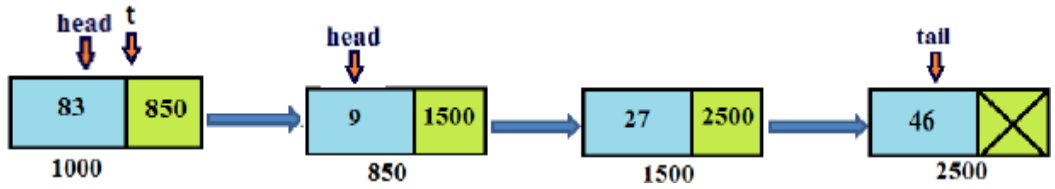
void ins_beg (int num) //Let num=10
{
n=(struct node *) malloc (size of(struct node));
n→next=NULL; n→data=num;
if (head==NULL) //case 1
{
head=n;
tail=n;
}
else //case 2
{
n→next=head;
head=n;
}
}
	void ins_end (int num) //Let num=10
	{
	n=(struct node *) malloc (size of(struct node));
	n→next=NULL; n→data=num;
	if (head==NULL) //case 1
	{
	head=n;
	tail=n;
	}
	else //case 2
	{
	tail→next=n
	tail=n;
	}
	}
		void ins_mid (int num, int mid_data)
		//Let num=10, mid_data=9
		{
		
		n=(struct node *) malloc (size of(struct node)); n→next=NULL;
		n→data=num;
		for(t=head; t!=NULL; t=t→next)
		{
		if(t→data==mid_data) break;
		}
		n→next=t→next;
		t→next=n;
		}
		;

DELETION

Deletion is the process of removing an element from the linked list. A node which is having the given element X will be removed from the list. For that, address of the previous node which contains X is needed. Address of the previous node of X is identified .

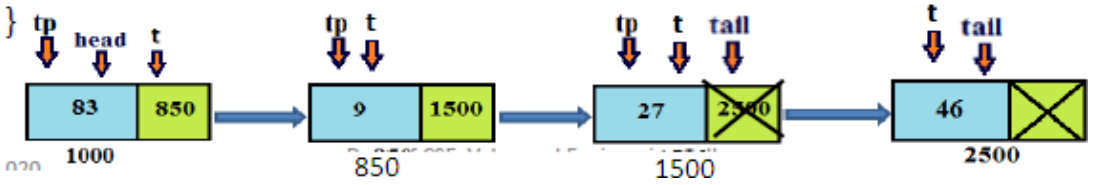
a) DELETION AT FIRST

```
void del_beg ()
 {
    t=head;
    head=t→next;
    free(t);
 }
```



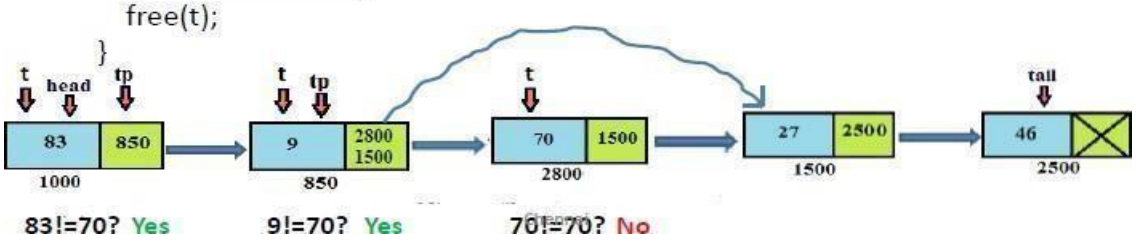
b) DELETION ATLAST

```
void del_end ()
 {
    struct node *tp;
    t=head;
    while(t→next!=NULL)
     {
       tp=t;
       t=t→next;
     }
      tail=tp;
      tail→next=NULL;
    free(t);
}
```



c) DELETION ATMIDDLE

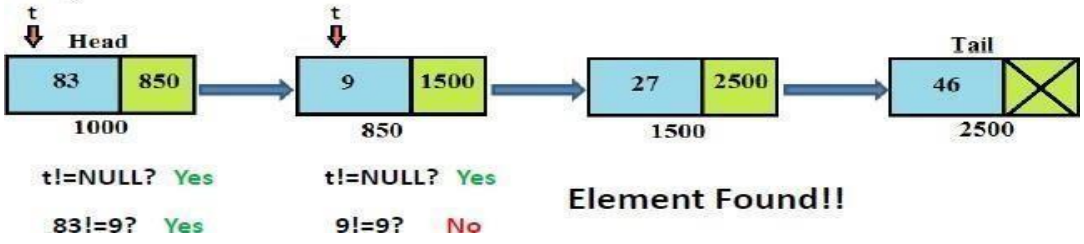
```
void del_mid (int num) //Let num=70
{
   struct node *tp;
   t=head;
   while(t→data!=num)
   {
    tp=t;
    t=t→next;
   }
   tp→next=t→next;
   free(t);
}
```



FIND

Find is the process of identifying the location of particular element in the linked list. This Function will return the position of the given element.

```
struct node* search (int num) //Let num=9
{
   t=head;
   while(t!=NULL && t→data!=num)
   {
      t=t→next;
   }
   return t;
}
```



Advantages:

- No wastage of memory
  – Memory allocated and deallocated as per requirement
- Efficient insertion and deletion operations

Disadvantages:

- Occupies more space than array to store a data
  – Due to the need of a pointer to the next node
- Does not support random or direct access

2. DOUBLY LINKED LIST

Doubly linked list is a list, which is implemented using structures and pointers. Each structure has three fields

- **Data :**Actual Information
- **Pointer to previous structure:** Points the previous node in thelist.
- **Pointer to next structure:** Points the next node in the list.

It can be represented as

Previous Node's Address	DATA	Next Node's address

Example:

Advantages:

Singly linked list cannot be traversed in the backward direction. But it is convenient to traverse the doubly linked list backwards. To do this, an extra field to point the previous structure is added.

Disadvantages:

This requires an extra link to point the previous node which doubles the cost of insertions and deletions because there are more pointers to fix. But it simplifies deletion.

Operations: The following operations can be performed on a doubly linked list

- **Insertion:** Inserts an element in a givenposition.
- **Deletion:** Deletes an element along with itnode.
- **Find:** Finds a given element and returns the address of its node.

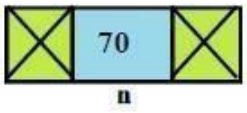
A Node in DLL

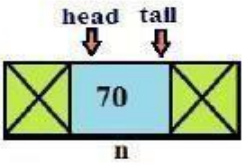
Declaration of a node:

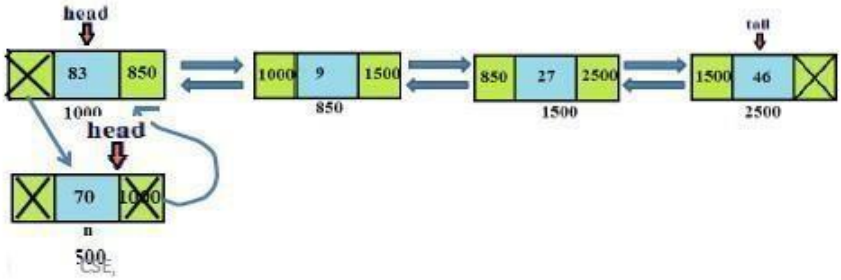
```
struct node
{
   int data;
    struct node *next, *prev;
};
```

A) INSERTION AT BEGINING:

```
void ins_beg_dll (int num) //Let num=70
{
    n=(struct node *) malloc(size of(struct node));
    n→next=NULL;
    n→prev=NULL;
    n→data=num;
    if(head==NULL) //case 1
    {
      head=n;
      tail=n;
    }
    else //case 2
    {
      n→next=head;
      head→prev=n;
      head=n;
    }
}
```

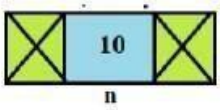


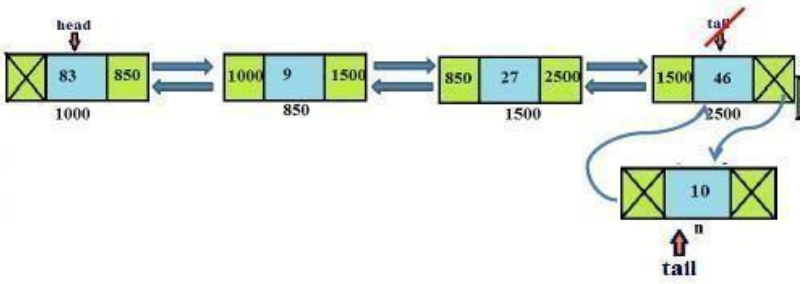




B) INSERTION AT END:

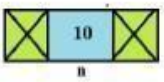
```
void ins_end_dll (int num) //Let num=10
{
  n=(struct node *) malloc (size of(struct node));
  n→next=NULL;
  n→prev=NULL;
  n→data=num;
  if (head==NULL) //case 1
   {
     head=n;
     tail=n;
   }
   else //case 2
   {
     tail→next=n;
     n→prev=tail;
     tail=n;
   }
}
```

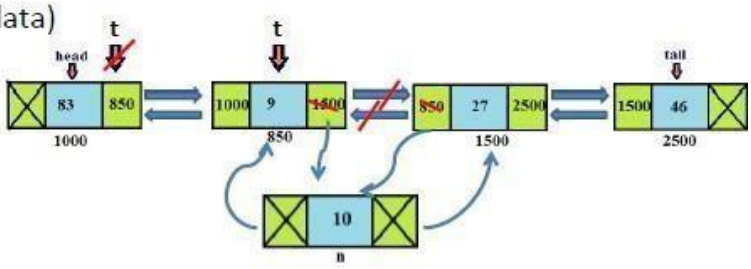




C) INSERTION AT MIDDLE:

```
void ins_end_dll (int num, int mid_data) //Let num=10, mid_data=9
{
  n=(struct node *) malloc (size of(struct node));
  n→next=NULL;
  n→prev=NULL;
  n→data=num;
  for(t=head; t!=NULL; t=t→next)
   {
     if(t→data==mid_data)
          break;
   }
  n→next=t→next;
  n→prev=t;
  t→next→prev=n;
  t→next=n;
 }
```

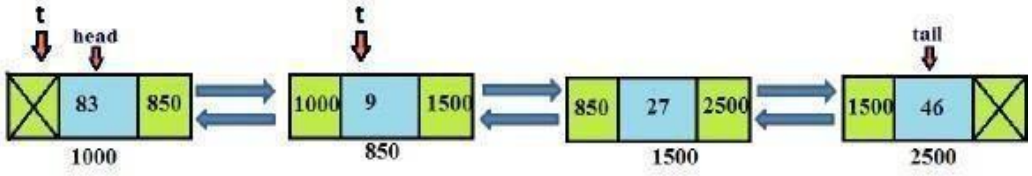




FIND:

- Finds a given element X from the list by traversing and returns the address ofit.

```
struct node* search_dll (int num) //Let num=9
{
   t=head;
   while(t!=NULL && t→data!=num)
    {
      t=t→next;
    }
   return t;
 }
```



Element Found!!
t is returned

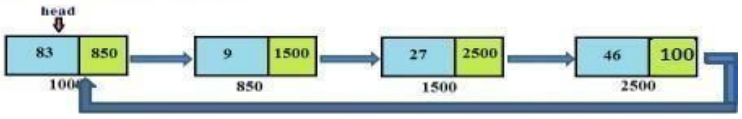
3. CIRCULAR LINKED LIST

Circular linked list a linked list, in which the next field of the last node points the first node. It can also be done with doubly linked list. It can be done with or without header node

Circular linked lists can be used to traverse the same list again and again if needed. In a circular linked list there are two methods to know if a node is the first node or not.

Global declaration of a node:

```
struct node
{
   int data;
    struct node *next, *prev;
}*head,*n,*t;
```



Various Operations

- Insertion
- Deletion
- Find
- Display all the elements

SINGLY CIRCULAR LINKED LIST:

In a normal singly linked list, for accessing any node of linked list, we start traversing from the first node. If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node. This problem can be solved by slightly altering the structure of singly linked list. In a singly linked list, next part (pointer to next node) is NULL; if we utilize this link to point to the first node then we can reach preceding nodes. In circularly linked list the pointer of last node points the address of the first node. It allows quick access to the first and last node. It can be implemented using singly linked list or doubly linked list.

OPERATIONS

- Insertion
- Deletion
- Find

Advantages:

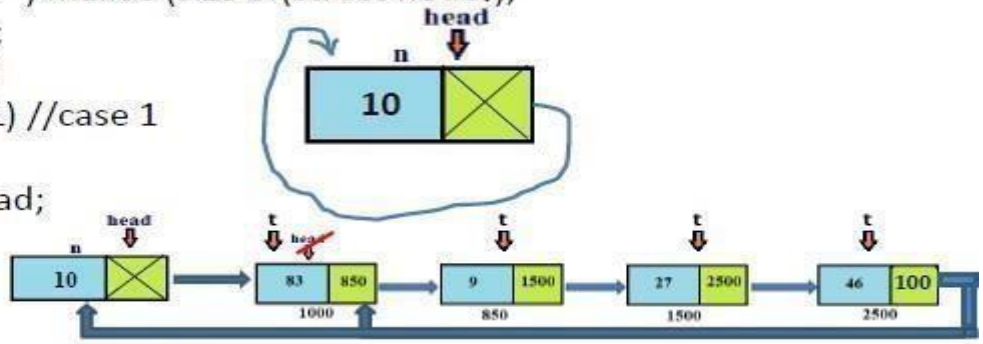
- Comparing to SLL, moving to any node from a node is possible

Disadvantages:

- Reversing a list is complex compared to linear linked list
- If proper care is not taken in managing the pointers, it leads to infinite loop
- Moving to previous node is difficult, as it is needed to complete an entire circle

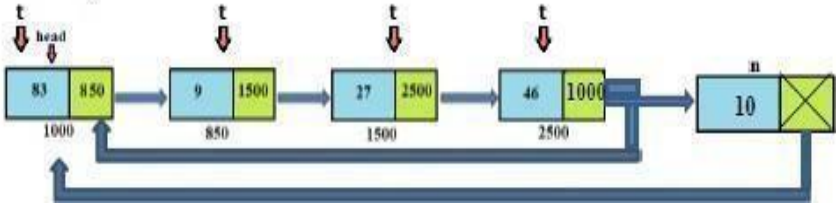
A) INSERTION AT BEGINING:

```
void ins_beg_cll (int num) //Let num=10
  {
    n=(struct node *) malloc (size of(struct node));
    n→next=NULL;
    n→data=num;
    if (head==NULL) //case 1
     {  head=n;
        n→next=head;
     }
     else //case 2
      { t=head;
          while(t→next!=head)
              t=t→next;
        t→next=n;
        n→next=head;
        head=n;
      }
  }
```



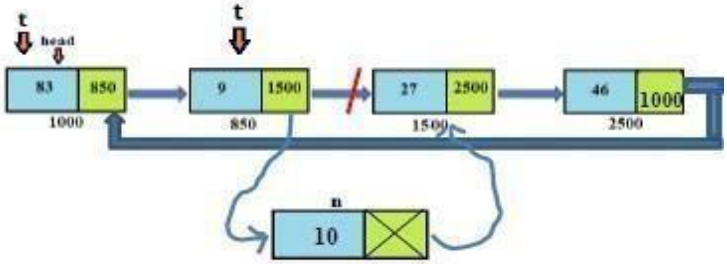
B) INSERTION AT END:

```
void ins_end_cll (int num) //Let num=10
   {
     n=(struct node *) malloc (size of(struct node));
     n→next=NULL;
     n→data=num;
       t=head;
         while(t→next!=head)
             t=t→next;
      t→next=n;
      n→next=head;
}
```



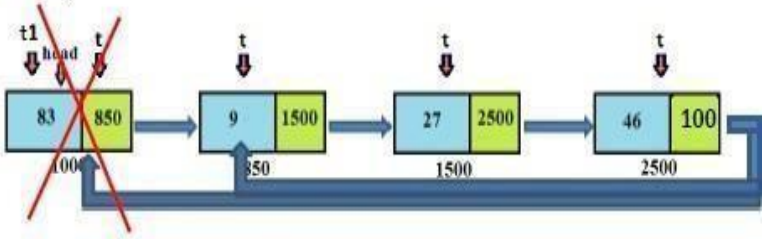
C) INSERTION AT MIDDLE:

```
void ins_end_cll (int num, int mid_data) //Let num=10, mid_data=9
{
  n=(struct node *) malloc (size of(struct node));
  n→next=NULL;
  n→data=num;
  for(t=head; t→next!=head; t=t→next)
  {
     if(t→data==mid_data)
          break;
  }
  n→next=t→next;
  t→next=n;
}
```



A) DELETION AT BEGINING:

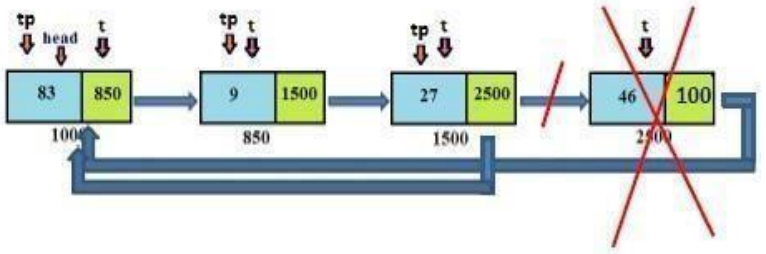
```
void del_beg_cll ()
{
   struct node *t1;
   t=head;
   t1=head;
      while(t→next!=head)
           t=t→next;
   t→next=head→next;
   head=t→next;
   free(t1);
}
```



t1 is freed!!

B) DELETION AT END:

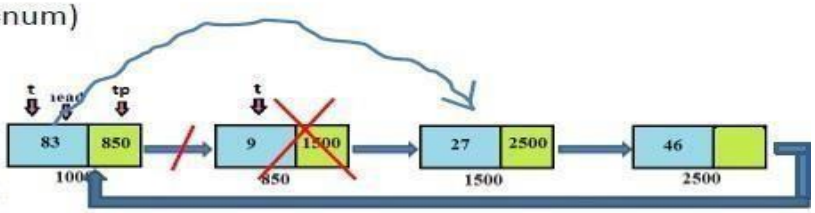
```
void del_end_cll ()
{
   struct node *tp;
   t=head;
      while(t→next!=head)
      {
         tp=t;
         t=t→next;
       }
   tp→next=head;
   free(t);
}
```



t is freed !!

C) DELETION AT MIDDLE:

```
void del_mid_cll (int num) //Let num=9
{
   struct node *tp;
   t=head;
      while(t→data!=num)
      {
         tp=t;
         t=t→next;
       }
   tp→next=t→next;
   free(t);
}
```



t is freed!!

SEARCH OPERATION

```
struct node* search_cll (int num) //Let num=9
{
   t=head;
   while(t→next!=head && t→data!=num)
    {
       t=t→next;
    }
   return t;
}
```

APPLICATIONS OF LIST OR LINKED LIST

1. Polynomial Addition
2. Radix Sort
3. Multi list
4. Buddy system
5. Dynamic memory allocation
6. Garbage Collection

POLYNOMIAL OPERATIONS

A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + \ldots. + jx + k$), where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

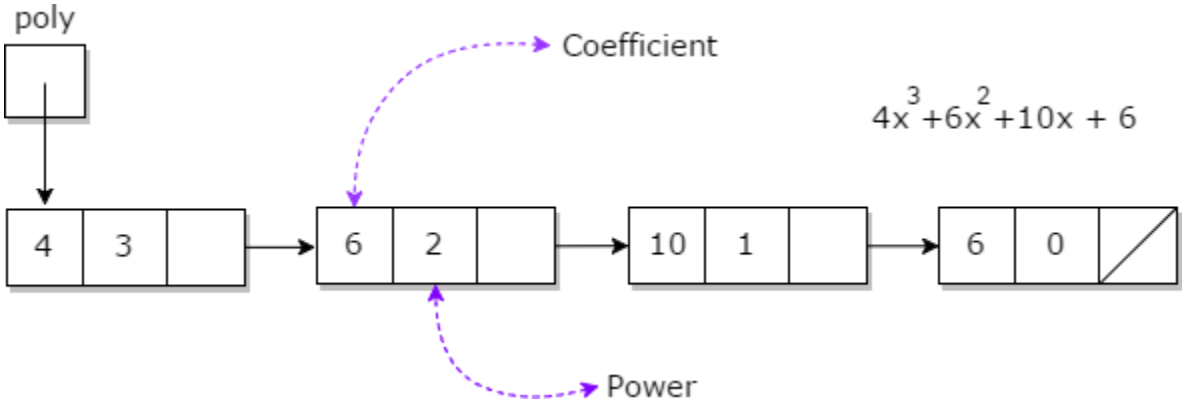
An essential characteristic of the polynomial is that each term in the polynomial expression consists oftwo parts:

- one is the coefficient
- other is the exponent

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in Mind while working with Polynomials:

- The sign of each coefficient and exponent is stored within the coefficient and the exponentitself
- Additional terms having equal exponent is possibleone
- The storage allocation for each term in the polynomial must be done in ascending anddescending order of theirexponent



Polynomial can be represented in the various ways. These are:

- By the use ofarrays
- By the use of LinkedList

Addition:

```
void polyadd(struct Node *poly1, struct Node *poly2, struct Node *poly)
{
while(poly1->next && poly2->next)
   {
      // If power of 1st polynomial is greater then 2nd, then store 1st as it is and move its pointer
      if(poly1->pow > poly2->pow)
      {

         poly->pow = poly1->pow;
         poly->coeff = poly1->coeff;
         poly1 = poly1->next;
      }

      // If power of 2nd polynomial is greater then 1st, then store 2nd as it is
      // and move its pointer
      else if(poly1->pow < poly2->pow)
      {
         poly->pow = poly2->pow;
         poly->coeff = poly2->coeff;
         poly2 = poly2->next;
      }

      // If power of both polynomial numbers is same then add their coefficients
      else
      {
         poly->pow = poly1->pow;
         poly->coeff =poly1->coeff+poly2->coeff;
         poly1 =poly1->next;
         poly2 =poly2->next;
      }

     // Dynamically create new node
      poly->next = (struct Node *)malloc(sizeof(struct Node));
      poly = poly->next;
      poly->next = NULL;
   }
while(poly1->next || poly2->next)
   {
      if(poly1->next)
      {
         poly->pow = poly1->pow;
         poly->coeff = poly1->coeff;
         poly1 = poly1->next;
      }
      if(poly2->next)
      {
         poly->pow = poly2->pow;
         poly->coeff = poly2->coeff;
         poly2 = poly2->next;
      }
```

```
 poly->next = (struct Node *)malloc(sizeof(struct Node));
       poly = poly->next;
       poly->next = NULL;
    }
}
```

Multiplication:

```
Node* multiply(Node* poly1, Node* poly2,
            Node* poly3)
{

   // Create two pointer and store the
   // address of 1st and 2nd polynomials
   Node *ptr1, *ptr2;
   ptr1 = poly1;
   ptr2 = poly2;
   while (ptr1 != NULL) {
      while (ptr2 != NULL) {
         intcoeff, power;

         // Multiply the coefficient ofboth
         // polynomials and store it in coeff
         coeff = ptr1->coeff *ptr2->coeff;

         // Add the powerer of both polynomials
         // and store it in power
         power = ptr1->power + ptr2->power;

         // Invoke addnode function to create
         // a newnode by passing three parameters
         poly3 = addnode(poly3, coeff, power);

         // move the pointer of 2nd polynomial
         // two get its next term
         ptr2 = ptr2->next;
      }

            ptr2 = poly2;
        ptr1 = ptr1->next;
   }

      removeDuplicates(poly3);
   return poly3;
}
```

RADIX SORT

- A second example where linked lists are used is called radix sort.
- Radix sort is sometimes known as card sort, because it was used, until the advent of modern computers, to sort old-style punch cards.
- If we have N integers in the range 1 to M (or 0 to M-t 1), we can use this information to obtain a fast sort known as bucket sort.
- We keep an array cells. Count, of size M, which is initialized to zero.
- Thus, Count has M cells(or buckets)
- In this example, suppose we have 10 numbers, in the range 0 to 999, that we would like to sort.
- In general, this is N numbers in the range 0 to N — I for some constant p.
- The following shows the action of radix sort on10numbers.
- The input is 64, 8, 216, 512, 27, 729, 0, 1, 343, 125 (the first 10 cubes, arranged randomly).

In first step, the bucket sort, sorts by the least significant digit. In first pass, the status of the bucket is,

0	1	512	343	64	125	216	27	8	729
0	1	2	3	4	5	6	7	8	9

Buckets after first step of radix sort.

Sorted by least significant digit, is 0, 1, 512, 343, 64, 125, 216, 27, 8, 729

(i) Now sort the list by the next least significant digit. This list is now sorted with respectto the two least significant digits.

8		729							
1	512	125							
0	216	27		343		64			
0	1	2	3	4	5	6	7	8	9

Buckets after the second pass of radix sort

Pass 2 gives output 0, 1, 8, 512, 2 125, 27, 729, 343, 64.

(ii) In final pass, it sorts by the most significant digit.

64									
27									
8									
1							729		
0	125	216	343		512				
0	1	2	3	4	5	6	7	8	9

The final list is 1, 8, 27,64 125, 216, 343, 512, 729

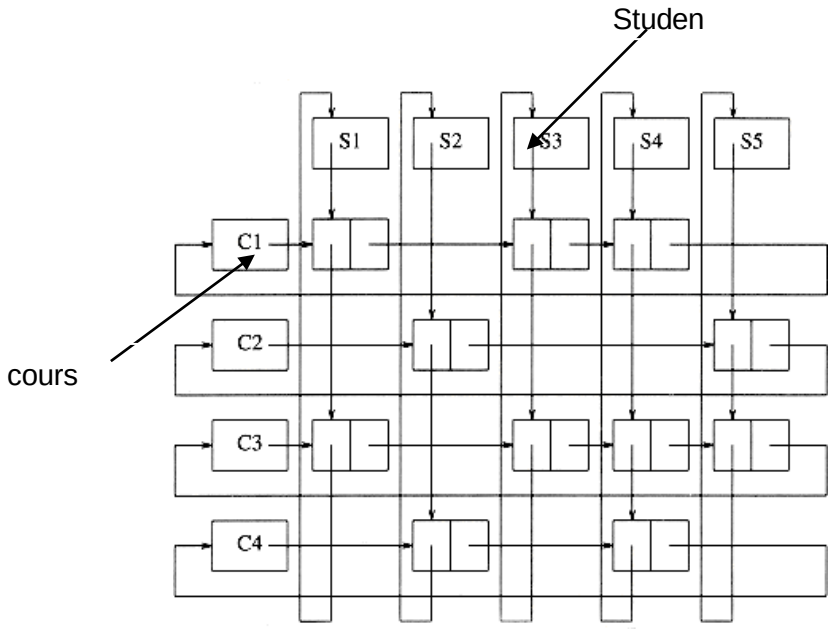
MULTILISTS

A university with 40,000 students and 2,500 subjects needs to generate 2 reports:

1. Lists of registration for each class.
2. Classes that each student registered for.

Implementation:

As the figure shows two lists are combined into one. All lists use a starter and are circular. To list the entire students in class c3, start at c3 and traverse its list by going right.
The first cell belongs to student s1. This can be determined by following the student's linked list until the starter is reached. Once this is done return to C3's list and finds another cell which belongs to s3.
In a similar manner we determine for any student, all of the class in which the students is registered. Using a circular list saves space but does so at expense of time.



UNIT 2

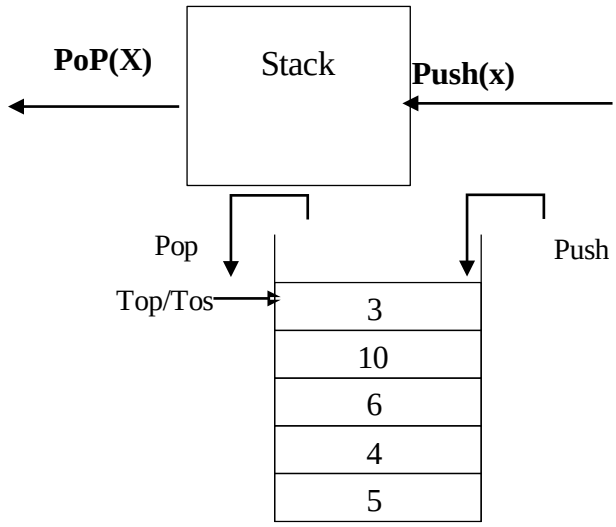
STACKS & QUEUES

Stack ADT – Operations – Applications – Balancing Symbols – Evaluating arithmetic expressions- Infix to Postfix conversion – Function Calls – Queue ADT – Operations – Circular Queue – DeQueue –Applications of Queues.

Stacks ADT:

- A stack is an ordered list in which all insertions and deletions are made at one end, called the top.
- Stack is a list with the restriction that insertions and deletions can be performed in only one position, namely the end of the list called Top.
- It follows **LIFO** approach. LIFO represents **"Last In First Out".** The basic operations
  - **Push**: Equivalent to insert.
  - **Pop :** Equivalent to delete. It deletes the most recently inserted element.

Stack Model:



The Basic Operations performed in the stack are:

- PUSH()
- POP()
- IsEmpty()
- IsFull()

EXCEPTION CONDITIONS IN STACK:

Overflow:

This is the process of trying to insert an element when the stack is full.

Underflow:

This is the process of trying to delete an element when the stack is empty.

Stack can be implemented in following ways

- Array Implementation
- Linked List Implementation

Primitive operations on the stack

- To create a stack
- To insert an element on to the stack.
- To delete an element from the stack.
- To check which element is at the top of the stack.
- To check whether a stack is empty or not.
- To check whether the stack is full or not

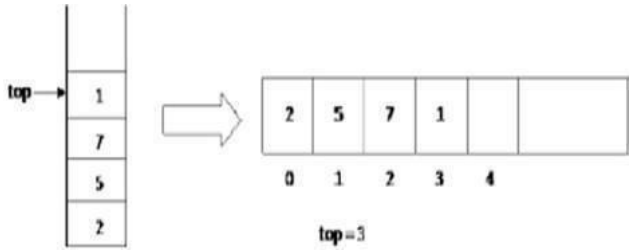
Implementation of Stack:

There are two methods of implementing stack operations.

- Array implementation
- Linked List implementation

1. Array Implementation of Stack:

```
struct stack
{
   int stk[MAXSIZE];
   int top;
};
```



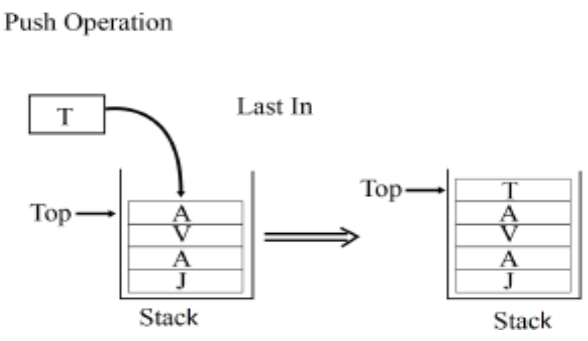
IS FULL Operation:

```
int isfull()
{
if(top == MAXSIZE)
 return 1;
else
 return 0;
}
```

PUSH Operation:

Inserting an element into the stack is called PUSH operation.

STEP 1 START
STEP 2 Store the element to push into array
STEP 3 Check if top== (MAXSIZE-1) then stack is full else goto step 4
STEP 4 Increment top as top = top+1
STEP 5 Add element to the position stk[top]=num
STEP 6 STOP




```
int push(int data)
{
 if(!isfull())
{
   top = top + 1;
stack[top] = data;
}
```

Explanation: If top is max size, (Max size is the maximum size of the stack) it implies that the stack is full; no more elements can be added into the stack. Otherwise, increment top by 1. Store the data in stack[top].

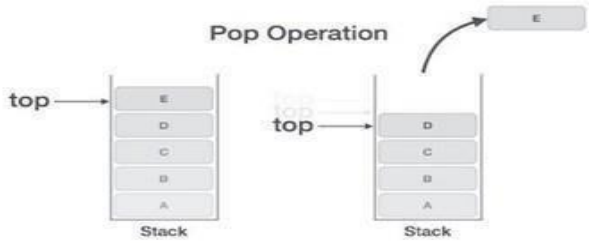
IS EMPTY Operation

```
int isempty()
{
if(top==-1)
return 1;
else
return 0;
}
```

Function for POP Operation:

- Step 1 – Checks if the stack is empty.
- Step 2 – If the stack is empty, produces an error and exit.
- Step 3 – If the stack is not empty, accesses the data element at which top is pointing.
- Step 4 – Decreases the value of top by 1.
- Step 5 – Returns success.

```
int pop()
 {
 int data;
if(!isempty())
{
  data = stack[top];
 top = top - 1;
 return data;} else
{ printf(" stack is empty"); }
```



Explanation:

If top=-1, it implies that the stack is empty; no element can be deleted from the stack. Otherwise, decrement top by1.

Function for Display Operation:

```
void display()
{
        int i;
        if(top==-)
                printf("\n stack is empty");
                        else
                        {
                                printf("\nThe elements in the stack are\n");
                                 for(i=top;i>=0;i--)
                                        printf("%d\t",Stack[i]);
                        }
                }
```

Explanation:

If top =-1, it implies that there are no elements in the stack; stack is empty.
Otherwise, display each element in the stack by formulating a loop; where i is initialized to zero, I is incremented till it reaches top.

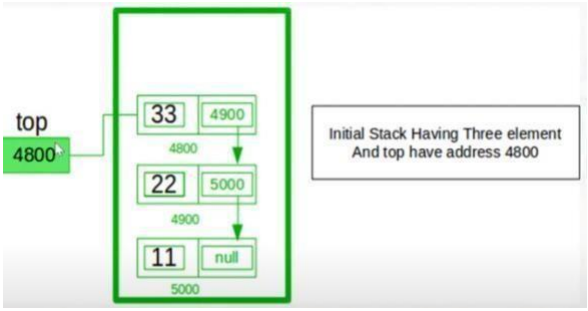
Disadvantages of stack using array implementation:

- The size of the Stack is limited.
- If an element is to be inserted into the stack and if the array is fully utilized then the stack will overflow.

2. Linked List Implementation

- Linked list operations perform based on Stack operations LIFO(last in first out) and with the help of that knowledge we are going to implement a stack using single linked list.
- Weare storing the information in the form of nodes and we need to follow the stack rules.

Logical Representation Of Stack:



Global Declaration:

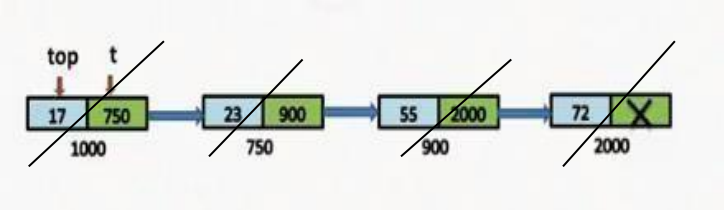
```
struct node
{
 int data;
 struct node *next
 };
 struct node *top=NULL;
```

PUSH Operation:

```
void push(int num)
{
n=(struct node*)malloc(sizeof(struct node));
n→data=num;
n→next=NULL;
if(top==NULL)
{
top=n;
}
else
{
n→data=num;
n→next=top;
top=n;
}
}
```

POP Operation:

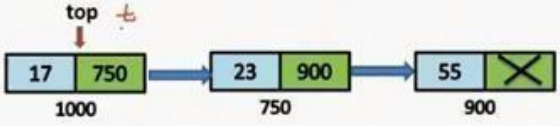
```
 void pop()
{
struct node *t;
if(top= =NULL)
{
printf("stack under flow");
}
else
{
t=top;
printf("%d",top→top);
top=top→next;
free(t);
}
}
```



Explanation: First it checks whether Stack is Empty. If yes, it implies that the stack does not contain any nodes. Otherwise, Stack pointer S moved to Next pointer. It is only a logical deletion and not physical deletion.

Diplay operation:

```
void display()
{
    struct node *t;
    if(top==NULL)
        printf("\nStack is empty")
    else
    {
        printf("\nElements in the stack:");
        for(t=head;t!=NULL;t=t->next)
            printf("\t%d",t->data);
    }
}
```



APPLICATIONS OF STACK. (OR) Discuss any two applications of stack with relevant examples. (Nov/Dec 2015)

Some of the applications which uses Stacks are,

- Balancing Symbols
- Evaluation of postfix expression
- Conversion of infix to postfix expression
- Function calls.

1. Balancing Symbols:

Compilers checks the program for syntax errors.During this process it checks for balancing of, parenthesis, braces and brackets. The number left parenthesis should be equal to the number of right parenthesis in the expression.

A stack is used to balance the parenthesis of the given expression .The simple algorithm uses a stack is as follows .

1. Make an empty stack. Read characters until end of file.
2. If the character is an opening symbol, push it onto stack.
3. If it is a closing symbol, then if the stack is empty, report an error. Otherwise, pop the stack.
4. If the symbol popped is not the corresponding opening symbol, then report an error.
5. At the end of file, if the stack is not empty, report an error.

2. Evaluation of Postfix Expressions:

Expression:

The collection of operators and operands are called expression.

- Operands: numbers or alphabets
- Operators: +, -, *, /
- Parenthesis: ( )

• Precedence:

- '(' and ')' have the highest precedence
- '*' and '/ 'have lower precedence than '(' and ')'
- '+' and '-' have lower precedence than '*' and '/'

Types of expression:

Infix → (operand 1) operator (operand 2)
Prefix → operator (operand 1) (operand 2)
Postfix → (operand 1) (operand 2) operator

The expressions in which the operators are placed at the end of operands on which they are going to operate is called **Postfix expression.**

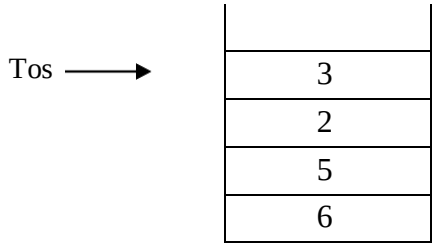
Algorithm to evaluate postfix expression:

1. When an operand is seen, it is pushed on to the stack.
2. When an operator is seen, the operator is applied to the two numbers that are popped from the top of the stack, and return the result pushed back on to the stack.

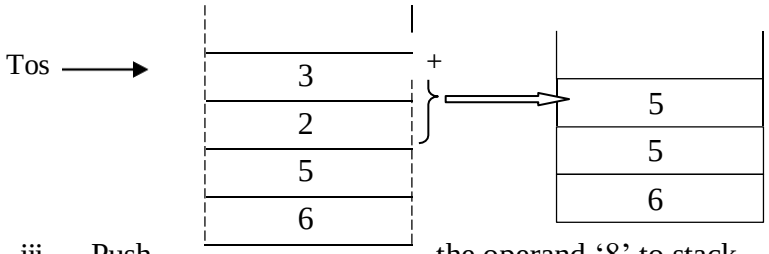
e.g : 6 5 2 3 + 8 * + 3 + *

Consider TOS as a stack pointer.

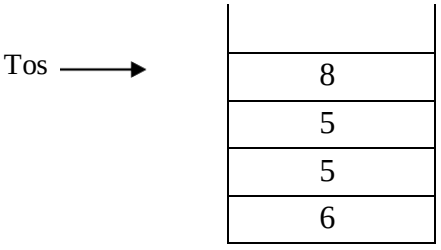
i. First four numbers are operand. So push it on to the stack.



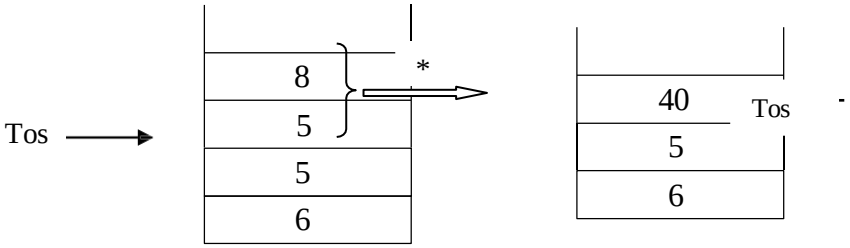
ii. Next symbol is aoperator , '+'. So, pop the most recent two operands from the stack and do the operation. i.e., (3+2) = 5 and return the result(5) back to stack.



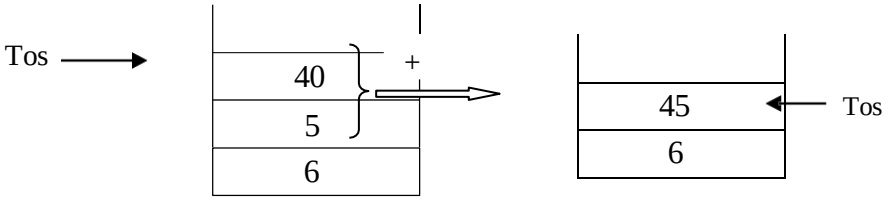
iii. Push the operand '8' to stack.



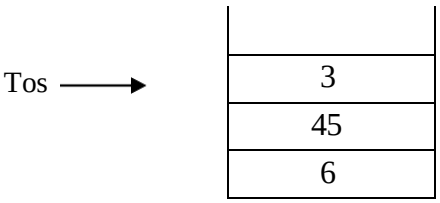
iv. Next symbol is a operator, '*' so pop two elements from the stack and apply the operation i.e., 8*5=40 then push the result 40 again to the stack.



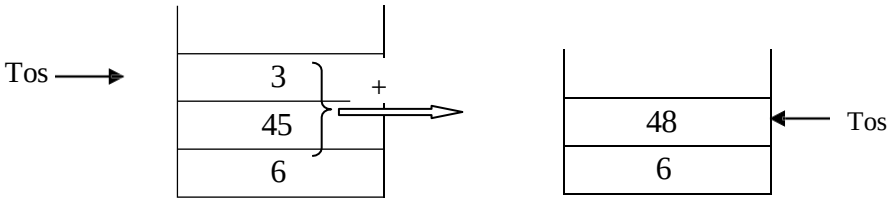
v. Next symbol is a operator, '+' => (40+5 = 45).



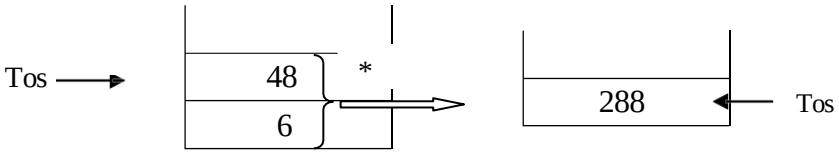
vi. Next symbol is a operand push it on to the stack.(i.e. 3)



vii. Next symbol is a operator, '+' => (3+45 =48)



viii. Next symbol is a operator, '*' => (48 * 6 = 288)



After Evaluation, The Compiler will result 288 as output.

3. Infix to postfix Conversion:

[Show the simulation using stack for the following expression to convert infix to postfix: a + b * c + (d * e + f) * .
(OR) Convert the following infix expression into postfix expression.
(OR) Explain the need for infix and postfix expressions. (May/June 2014)
(OR) Write an algorithm to convert the infix expression to postfix expression. (May/June 2016)]

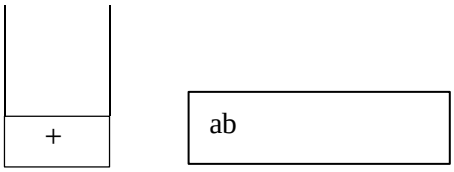
Convert the infix expression **a + b * c + (d * e + f) * g** into postfix expression.

Algorithm steps to follow:

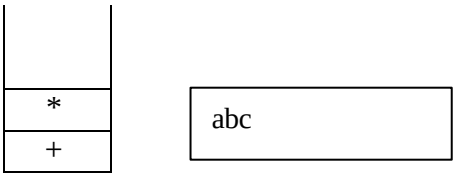
(1) Initialize a stack to empty.
(2) Read a symbol
  (2.1) If it is an operand, place onto the output.
  (2.2) If it is an operator
    (a) If it is a right parenthesis, then pop the stack, write symbols, until a left parenthesis is encountered.
      Remark: The '(' is popped, but not written as output.
    (b) If it is any other symbol, such as '+', '*', '(', pop entries from the stack until an entry of lower priority is found, or '(' is found. When the popping is done, push the operator onto the stack.
      Remark: If '(' is found, don't pop it.
(3) Go to step (2).
(4) If read the end of input, pop the stack until it is empty, writing symbols onto the output.

Input : a + b*c +(d*e+f)*g

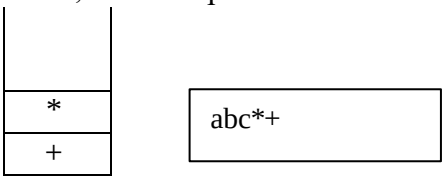
➢ First, the symbol 'a' is read, so it is passed through to the output. Then '+' is read and pushed onto the stack. Next b is read and passed through to the output.



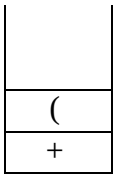
➢ Next a '*' is read. The top entry on the operator stack has lower precedence than '*', so nothing is output and '*' is put on the stack. Next, c is read and output. Thus far, we have



➢ The next symbol is a '+'. Checking the stack, we find that we will pop a '*' and place it on the output; pop the other '+', which is not of lower but equal priority, on the stack; and then push the '+'.

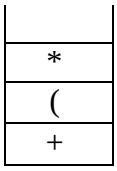


➢ The next symbol read is a '(', which, being of highest precedence, is placed on the Stack Then d is read and output.



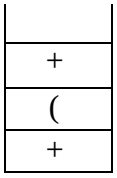
abc*+d

➢ Next by reading a '*', the open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, e is read and output.



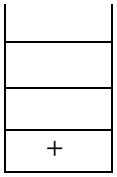
abc*+de

➢ The next symbol read is a '+'. We pop and output and then push '+'. Thenread and output f.



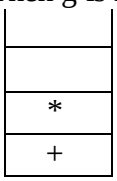
abc*+de*f

➢ Now we read a ')', so the stack is emptied back to the' We output a '+'



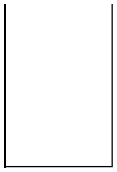
abc*+de*f+

➢ Now read a '*' next; it is pushed onto the stack. Then g is read and output.



abc*+de*f+g

➢ The input is now empty, so we pop and output symbols from the stack until itempty.



abc* + de * f + g *+

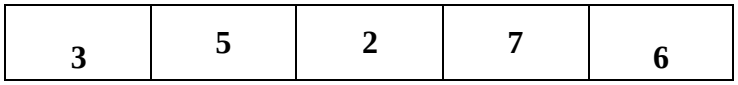
4. **Function Calls:**

- The algorithm to check balanced symbols suggests a way to implement function calls.
    - The problem here is that when a call is made to a new function, all the variables local to the calling routine need to be saved by the system, since otherwise the new function will overwrite the calling routine's variables.
    - Furthermore, the current location in the routine must be saved so that the new function knows where to go after it is done. When there is a function call, all the important information that needs to be saved such as register values and the return address is saved "on a piece of paper" in an abstract way and put at the top of a pile.
    - Then the control is transferred to the new function, which is free to replace the registers with its valued. If it makes other function calls, it follows the same procedure.
    - When the function wants to return, it looks at the "paper" at the top of the pile and restores all the registers. It then makes the return jump.

- Clearly, all of this work can be done using a stack, and that is exactly what happens in virtually every programming language that implements recursion. The information saved is called either an **activation record** or **stack frame**.

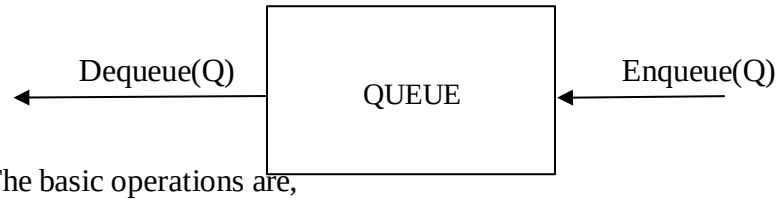
QUEUE ADT

Queue is an ordered collection of data items. It delete item at front of the queue. It inserts item at rear of thequeue. It has FIFO structure i.e. "**First In First Out".**



front **rear**

Queue Model:



The basic operations are,

- **Enqueue :**which inserts an element at the end of the list called **rear end**.
- **Dequeue:** Which deletes an element at the other end (front)of the list called **frontend.**

Types of Queue:

- Simple Queue
- Circular Queue
- Double Ended Queue
- Priority Queue

Implementation of Simple Queue(Queue):

Like stack, queue can also be implemented in two methods.

- Using Array
- Using Linked list

1.Array Implementation of Queue(Simple Queue):

Global declaration:

```
#define MAX 5

int Q[MAX];
int front=-1,rear=-1;
```

Enqueue Operation:

```
void enq(int  num)

{
if(rear==MAX-1)
{
printf("Queue overflow")
```

```
}
else if(front==-1 && rear==-1)
{
front=rear=0;
q[rear]=num;
}
else
{
rear++;
q[rear]=num;
}
}
```

Dequeue Operation:

```
void deq( )
{
if(front==-1 && rear==-1)
{
printf("queue is underflow");
}
else if(front==rear)
{
printf("deleted element :%d",Q[front]);
front=rear=-1;
}
else
{
printf("Deleted element :"%d",Q[front]);
Front++;
}
```

Get front:

```
int get_front()
{
if(!isempty())
return Q[front];
}
```

Get rear:

```
int getRear()
{
if(!isEmpty())
return Q[rear];
}
```

isFull:

```
int isFull()
{
return(rear==MAX-1)
}
```

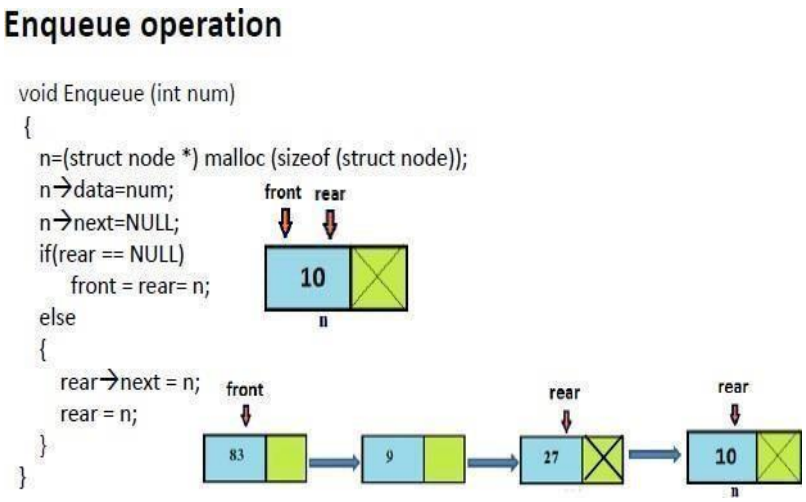
isempty

```
int isEMPTY()
{
return(front==-1)
}
```

Disadvantages of Queue using array implementation:

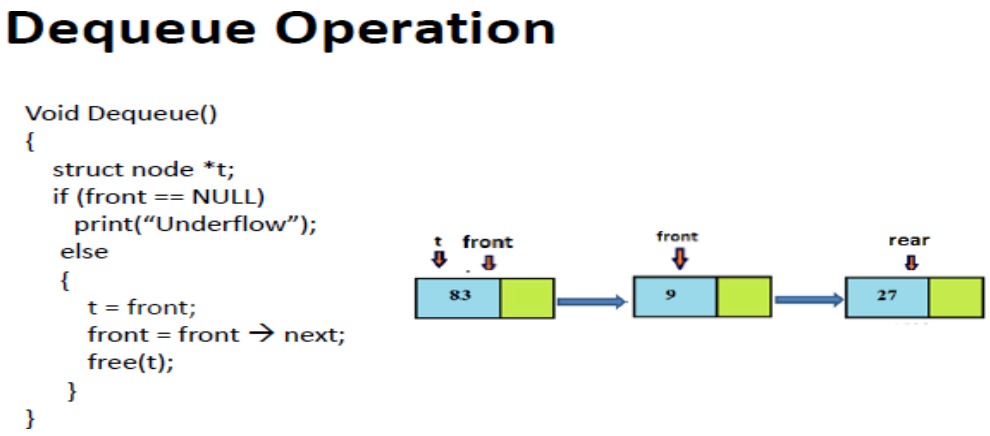
- The size of the Queue is limited.
- If an element is deleted from the queue, all the remaining elements to be shifted by 1 position forward. This operation takes more cost.
- This problem can be overcome by Circular Queue.

Linked List implementation of Simple Queue:
Enqueue Operation:



Explanation: This function creates a new node called temp. The data is stored in the data part of the temp. If front==NULL, it implies that the queue is not created. The node temp becomes front and rear. The next contains NULL. Otherwise, insert the new node at rear->next and make temp as rear.

Dequeue (Deletion) operation:



Circular Queue:

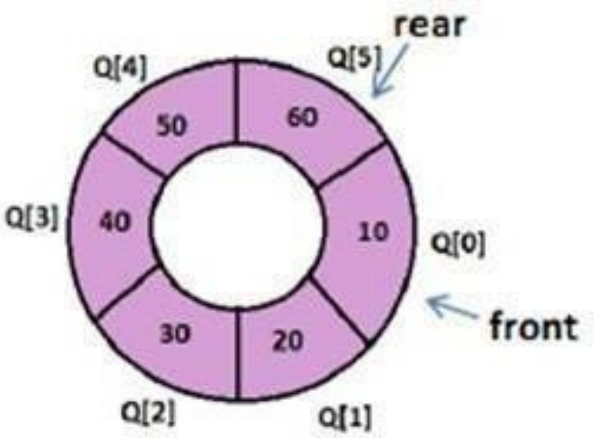
The drawbacks of simple queues are, time consuming and there is a chance to declare queue is full even if there is a empty space at front.

These can be overcome by circular queue.

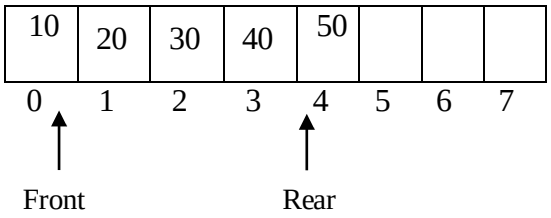
Circular queue is a wrap around queue, after insertion of last position of the queue, if there is an empty space at first, the insertion pointer will insert at first. ie., it won't show queue full message.

It is possible to insert new elements into the circular queue if the array slots at the beginning of the queue is empty.

Pictorial representation of Circular Queue:



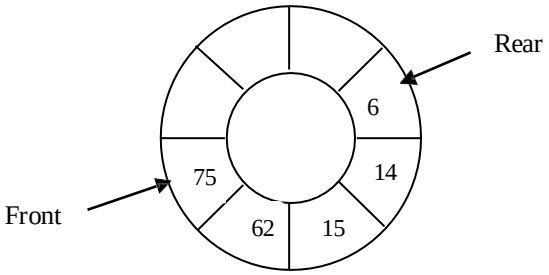
Logical representation of Circular Queue:



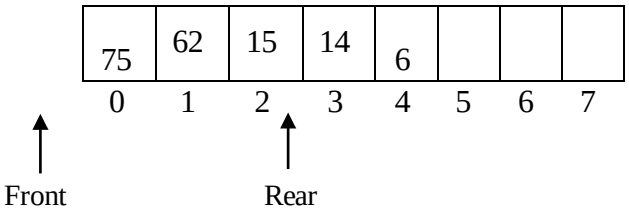
Circular queue operations are,

- Enqueue(Insertion)
- Dequeue(Deletion)
- Display

Pictorial representation of Circular Queue:



Logical representation of Circular Queue:



Circular queue operations are,

- Enqueue(Insertion)
- Dequeue(Deletion)
- Display

Global declarations:

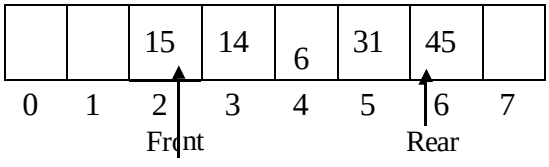
```
intCQueue[100],front=-1,rear=-1,count=0,maxsize=99;
```

Function to Enqueue at Rear:

```
voidenqueue_rear(int x)
{
If(count==maxsize)
      Printf("Queue is full");
Else
{
      Rear=(rear+1)%maxsize;
      CQueue[rear]=x;
      Count++;
}
}
```

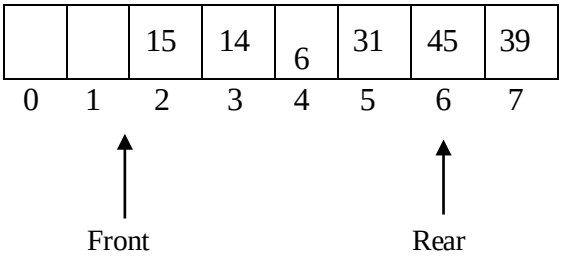
Example:

Maxsize=8
Initial status of the queue is,



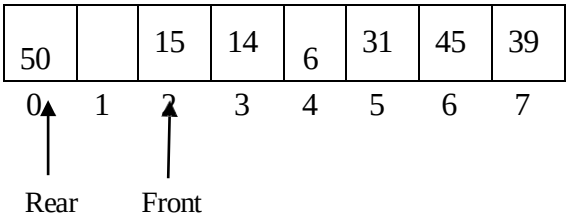
Enqueue 39.

Front=2, rear=(6+1)%8=>*7*



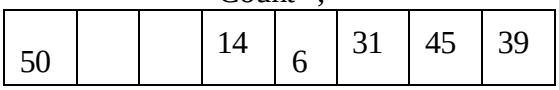
Enqueue 50

Front=2, rear=(7+1)%8=>0



Function to dequeue at front:

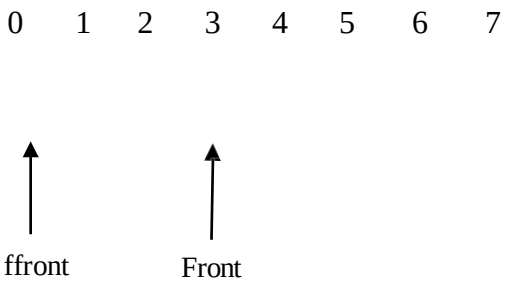
```
voiddequeue(int x)

{
If(count==0)
        Printf("Queue is empty");
Else
{
        Front=(front+1)%maxsize;
        Count--;
}
}
```



Example:

Dequeue,

Front=(front+1)%maxsize;

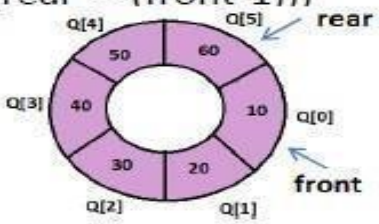


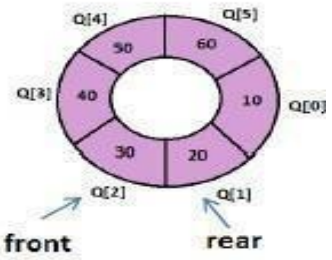
Function to display Queue elements:

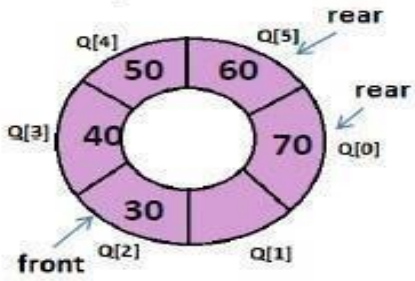
```
Void display()
{
int i;
for(i=1;i<=count;i++)
{
        printf(" %d ",CQueue[j]);
        j=(j+1)%maxsize;
}
}
```

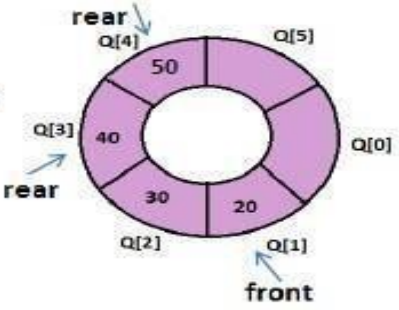
Enqueue Operation

```
void Cir_Enqueue (int num)
{
   if((front == 0 && rear == max-1) || (rear ==(front-1)))
        print("Queue Overflow");
   else if (front == -1 && rear == -1)
   {   front = rear = 0;
       Q[rear] = num;
    }
    else if (rear == max-1 && front != 0)
    {   rear = 0;
        Q[rear] = num;
    }
    else
    {    rear ++;
         Q[rear] = num;
     }
}
```







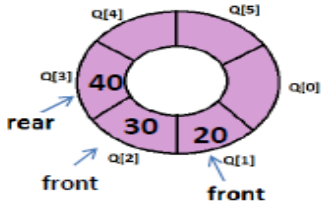


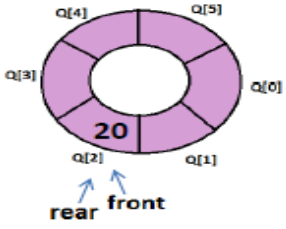
Double Ended Queue.

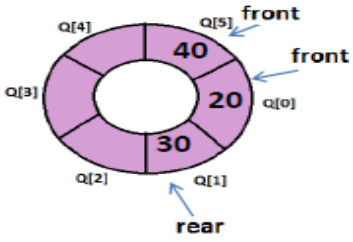
It is another type of queue and this is also calle

Dequeue Operation

```
Void Cir_Dequeue()
{
   int  val;
   if (front ==-1)
        print ("Queue Underflow");
   val = Q[front];
   if (front == rear)
        front = rear = – 1;
   else if (front == max – 1)
        front = 0;
   else
        front ++;
}
```







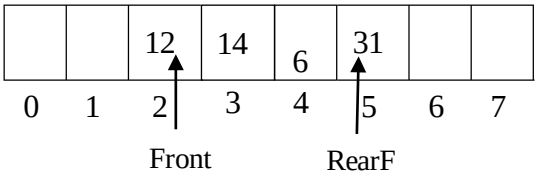
DEQUE.(Double Ended Queue)

A Deque is a special type of data structure in which insertions and deletions can be performed in boththe ends. ie., at the front end and the rear end of the queue. There are 4 operations, 1. Insertion at front,
2. Insertion at rear, 3. Deletion at front, 4. Deletion at rear.

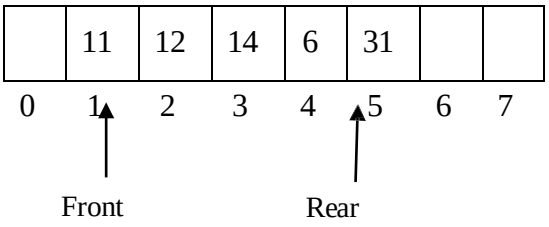
Function to enqueue at front:

```
voidenqueue_front(int x)
{       If(front==0)
        Printf("Insertion at front is not possible.");
Else
{
        Front=front-1;
        Deque[front]=x;


        Count++;
}
}
```

Example:



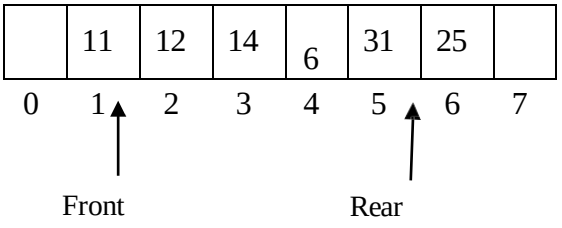
Enqueue_Front 11.



Function to enqueue at rear:

```
voidenqueue_rear(int x)
{       If(rear==maxsize)
        Printf("Insertion at rear is not possible:");
Else
{


        Rear=rear+1;
        Deque[rear]=x;
        Count++;
}
}
```

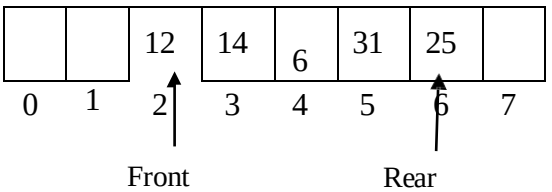
Enqueue_Rear 25



Function to dequeue at front:

```
Void dequeue_front()
{
If(front>rear)
        Printf("Queue is empty");
Else
{       Front=front+1;
        Count--;
}
}
```

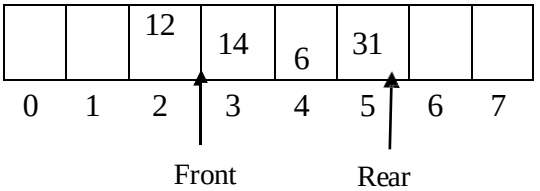
Dequeue_Front



Function to dequeue at rear:

```
Void dequeue_rear()
{
If(front>rear)
        Printf("Queue is empty");
Else
{       Rear=rear-1;
        Count--;
```

Dequeue_Rear

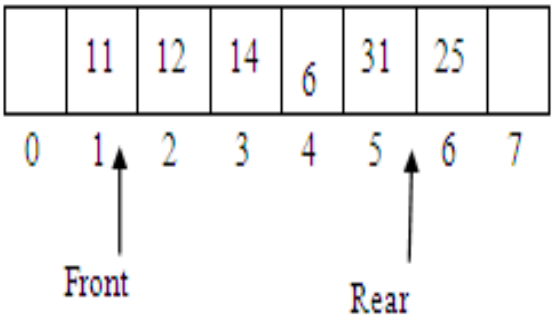


```
    }
    }
```

Function to enqueue at rear:

```
void enqueue_rear(int x)
{       If(rear==maxsize)
                Printf("Insertion at rear is not possible:");
        Else
        {

                Rear=rear+1;
                Deque[rear]=x;
                Count++;
        }
}
```

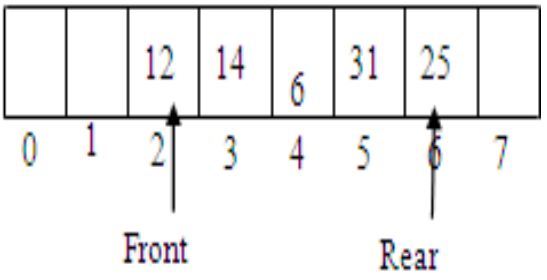
Enqueue_Rear25



Function to dequeue at front:

```
Voiddequeue_front()
{
        If(front>rear)
                Printf("Queue is empty");
        Else
        {       Front=front+1;
                Count--;
        }
}
```

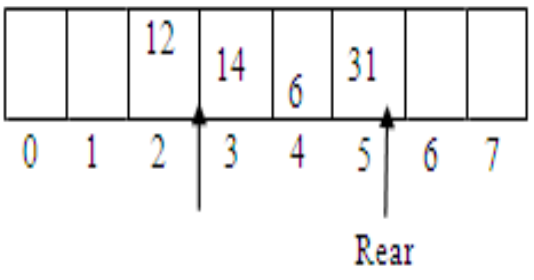
Dequeue_Front



Function to dequeue at rear:

```
Voiddequeue_rear()
{
        If(front>rear)
                Printf("Queue is empty");
        Else
        {       Rear=rear-1;
                Count--;Front

        }
}
```

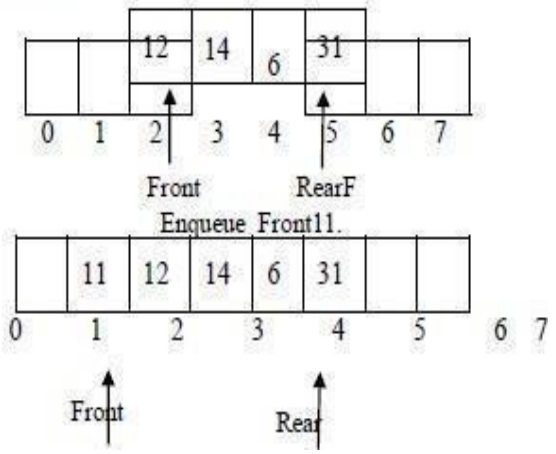
Dequeue_Rear



Function to enqueueat front:

```
void enqueue_front(int x)
{       If(front==0)
                Printf("Insertion at front is not possible.");
        Else
        {
                Front=front-1;
                Deque[front]=x;
Count++;
        }
```

Example:



Enqueue Front11.

Applications of Queues.

Print jobs

When jobs are submitted to a printer, they are arranged in the order they arrive. Then jobs sent to a line printer are placed on a queue.

Computer networks

There are many network setup of personal computers in which the disk is attached to one machine called file server. Users on other machines are given access to files on a first come first served basis where the data structure is queue.

OS

Operating system performs various tasks namely memory management, CPU management etc. The operating system follows a technique called **First Come Fist Serve (FCFS)** to allocate CPU for the waiting processes.

Real-life waiting lines:

i. Calls to large companies are generally placed on a Queue when all the operators are busy.

ii. In large Universities, where resources are limited, students must sign a waiting list if all terminals are occupied.

Mathematics:

There is a branch of Mathematics called Queuing Theory, which deals with computing, probabilistically, how long users expect to wait on a line.

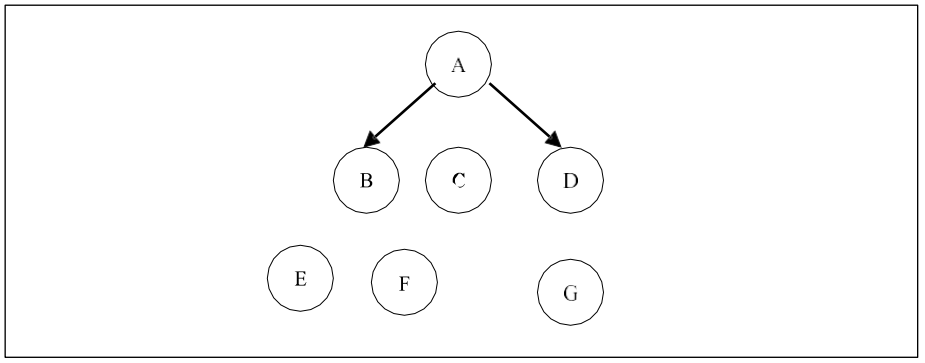
UNIT III

TREES

Tree ADT – Tree Traversals - Binary Tree ADT – Expression trees – Binary Search Tree ADT – AVL Trees – Priority Queue (Heaps) – Binary Heap.

TREE ADT:

- A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes can form sub trees.



- Tree represents the nodes connected by edges.
- Binary Tree is a special data structure used for data storage purposes.
- A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.

Important Terms

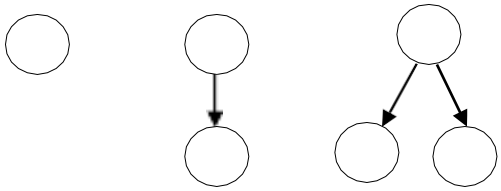
- **Path:** Path refers to the sequence of nodes along the edges of a tree.
- **Root:** The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent:** Any node except the root node has one edge upward to a node called parent.
- **Child:** The node below a given node connected by its edge downward is called its child node.
- **Leaf**: The node which does not have any child node is called the leaf node.
- **Sub tree:** Sub tree represents the descendants of a node.
- **Traversing:** Traversing means passing through nodes in a specific order.
- **Levels:** Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and soon.
- **Keys:** Key represents a value of a node based on which a search operation is to be carried out for a node.
- **Siblings**: All the nodes that share the same parent are called siblings.
- **Depth**: The depth of a node N is the length of the path from the root to the node N
- **Height**: The Height of a node N is the length of the path from the node to the deepest leaf.

Types of Trees:

1. General Trees
2. Forests
3. Binary Trees
4. Binary Search Tree(BST)
5. Tournament Trees

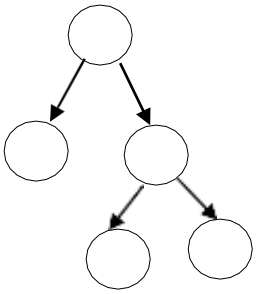
Forests:

A forest is a disjoint union of trees. A forest is also defined as an ordered set of zero or more general trees



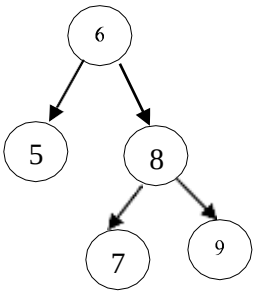
Binary Trees:

A binary tree is tree with not more than two children. Every node can have zero, one or at most two children



Binary Search Tree (BST)

Binary Search Tree is a tree with a condition that left child is smaller than the root and right is greater than the root.



Tournament Trees:

In a tournament tree, each external node represents a player and each internal node represents the winner of the match played between the players represented by its children

TREE TRAVERSAL:

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

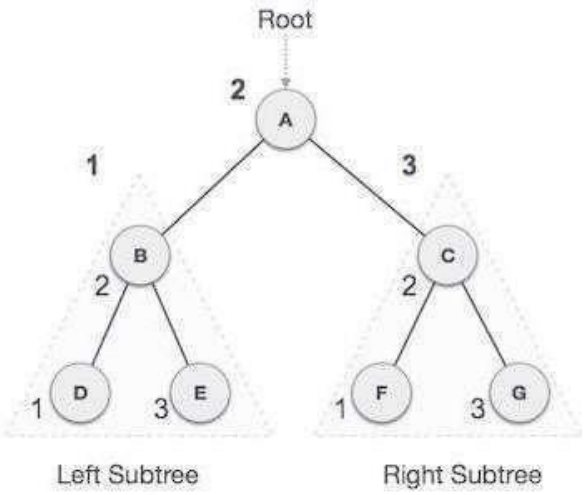
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left sub tree is visited first, then the root and later the right sub-tree.
We should always remember that every node may represent a sub tree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.




```
void Inorder(Tree T)
{
   if(T!=NULL)
          {
          Inorder(T→Left);
              Write(T→Data);
          Inorder (T→Right)
          }
}
```

We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree wills be−

D → B → E → A → F → C → G

Algorithm

Until all nodes are traversed −

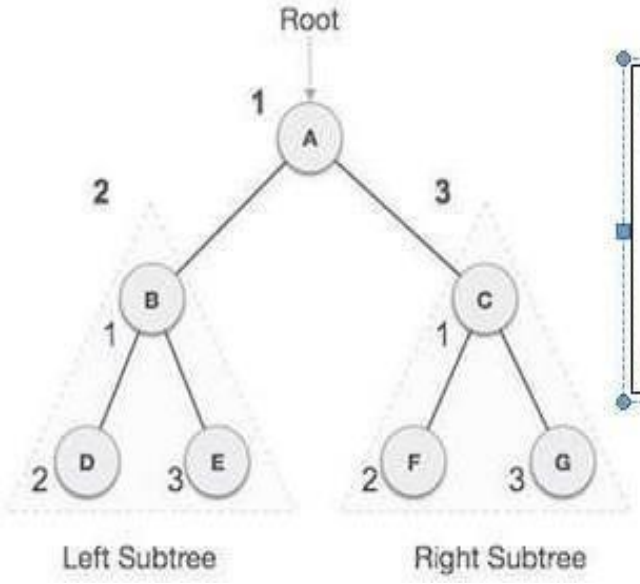
Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.




```
void Preorder(Tree T)
{
  if(T!=NULL)
         {
         Write(T→Data);
         Preorder(T→Left);
         Preorder (T→Right)
         }
}
```

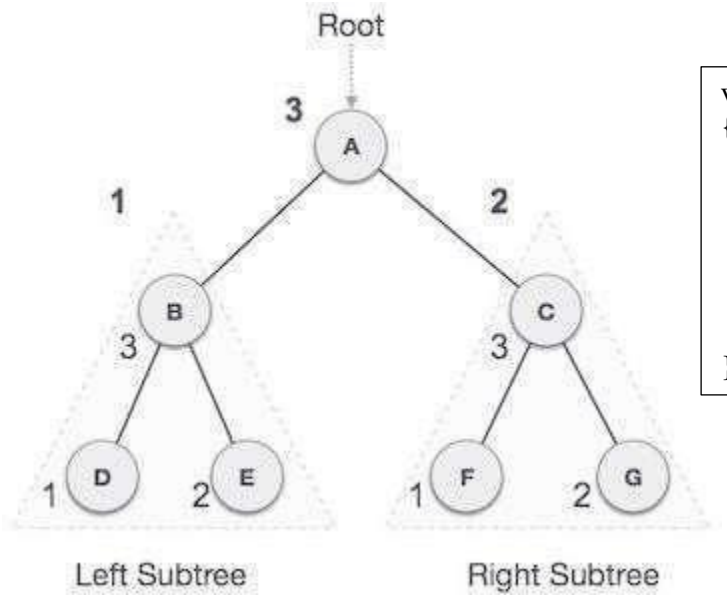
We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B. B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree willbe−

A → B → D → E → C → F →G

Algorithm
Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.




```
void Postorder(Tree T)
{
   if(T!=NULL)
          {
          Postorder(T→Left);
          Postorder (T→Right)
          Write(T→Data);
          }
}
```

We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

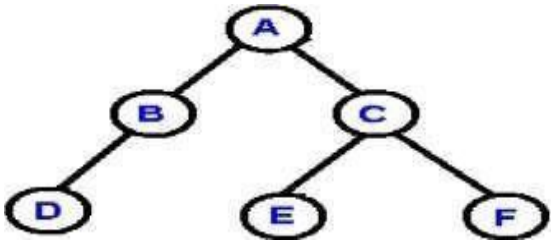
D → E → B → F → G → C → A

Algorithm
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.

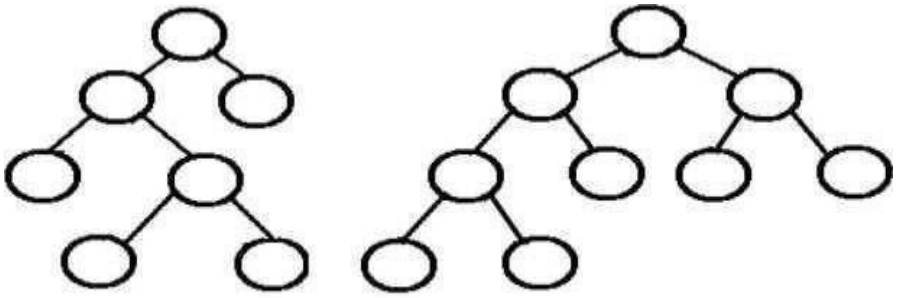
BINARY TREE ADT:

A binary tree is tree with not more than two children. Every node can have zero, one or at most two children A binary tree is made of nodes, where each node contains a "left" reference, a "right" reference, and a data element. The topmost node in the tree is called the root.

Every node (excluding a root) in a tree is connected by a directed edge from exactly one other node. This node is called a parent. On the other hand, each node can be connected to arbitrary number of nodes, called children. Nodes with no children are called leaves, or external nodes. Nodes which are not leaves are called internal nodes. Nodes with the same parent are called siblings.



- The depth of a node is the number of edges from the root to the node.
- The height of a node is the number of edges from the node to the deepest leaf.
- The height of a tree is a height of the root.
- A full binary tree is a binary tree in which each node has exactly zero or two children.
- A complete binary tree is a binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right.

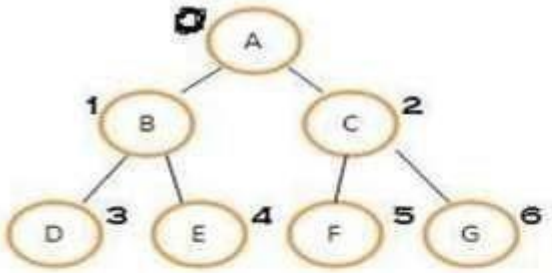


A complete binary tree is very special tree, it provides the best possible ratio between the number of nodes and the height. The height h of a complete binary tree with N nodes is at most O(log N). We can easily prove this by counting nodes on each level, starting with the root, assuming that each level has the maximum number of nodes:

$$n = 1 + 2 + 4 + ... + 2^{h-1} + 2^h = 2^{h+1} - 1$$

Solving this with respect to h, we obtain

$$h = O(\log n)$$



Root = i

leftchild=2i+1

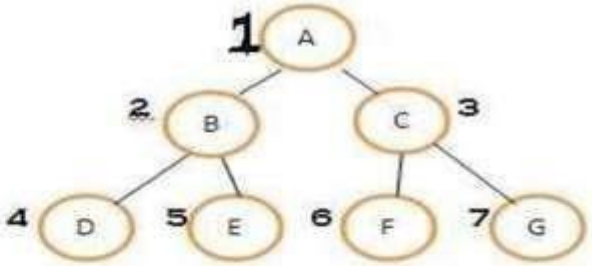
rightchild=2i+2

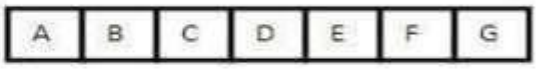
leftchild's parent position = i/2

$2^{n+1} - 1$ => array size

n => number of levels of a tree

rightchild's position= (i-1)/2





Root = i

leftchild=2i

rightchild=2i+1

parent position= i/2

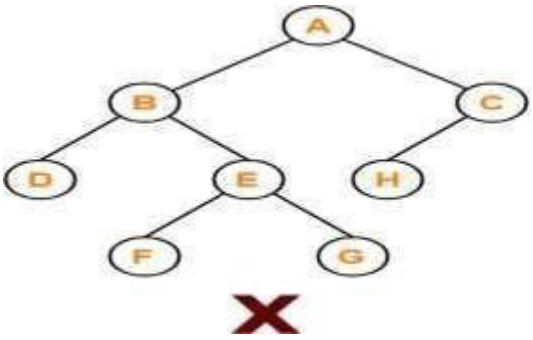
$2^{n+1} - 1$ => size of array

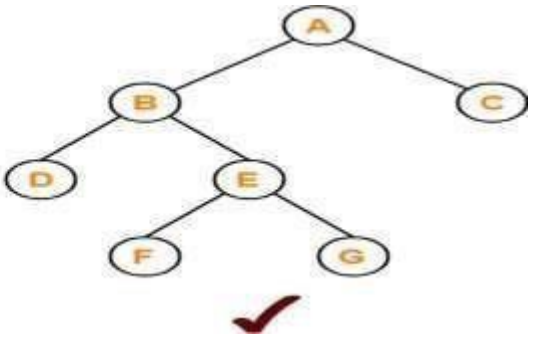
n => number of levels of a tree

Types of binary Tree

1. **Full Binary Tree**

- A binary tree in which every node has either 0 or 2 children is called as a Full binary tree.
- Full binary tree is also called as Strictly binary tree.

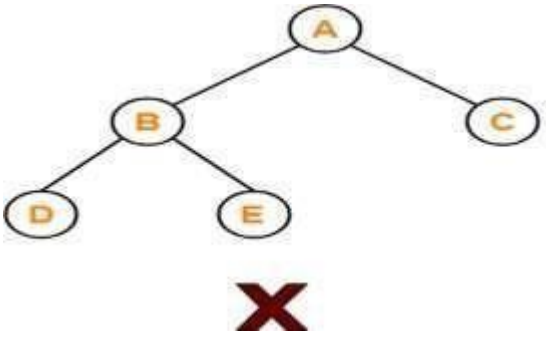


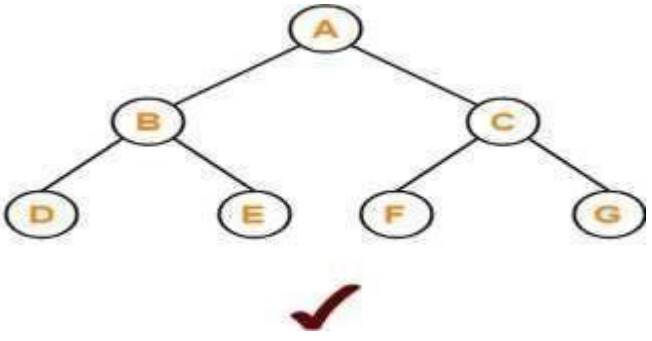


2. Perfect Binary Tree

A Perfect binary tree is a binary tree that satisfies the following 2 properties-

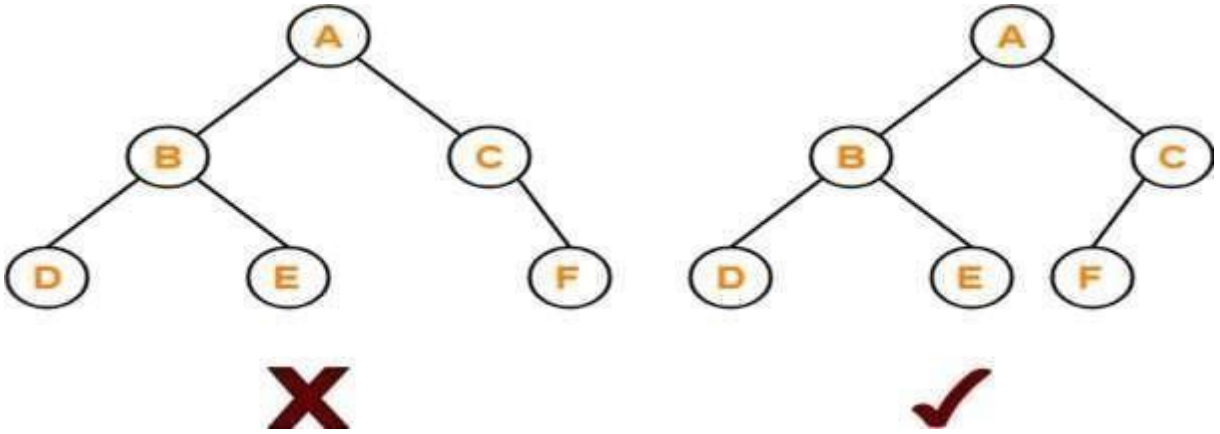
1. Every internal node has exactly 2 children.
2. All the leaf nodes are at the same level.





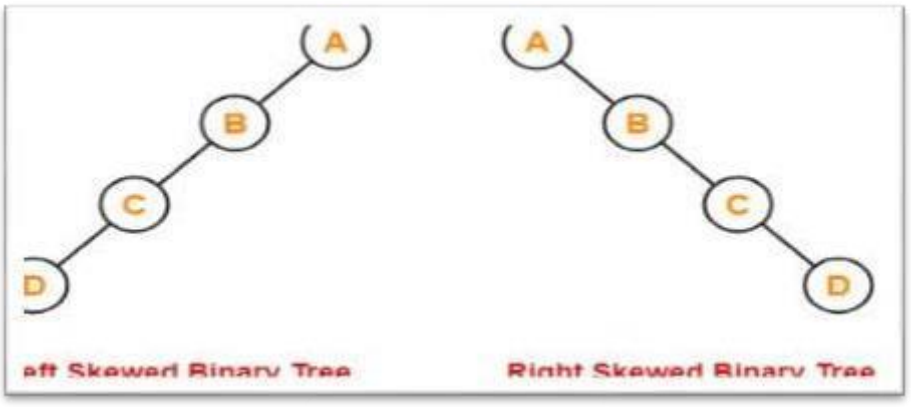
3. Complete Binary Tree

- A complete binary tree is a binary tree that satisfies the following 2 properties-
  1. All the levels are completely filled except possibly the last level.
  2. The last level must be strictly filled from left to right.



4. Skewed Binary Tree

- A skewed binary tree is a binary tree that satisfies the following 2 properties-
  1. All the nodes except one node has one and only one child.
  2. The remaining node has no child.



Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies
- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move subtrees around with minimum effort

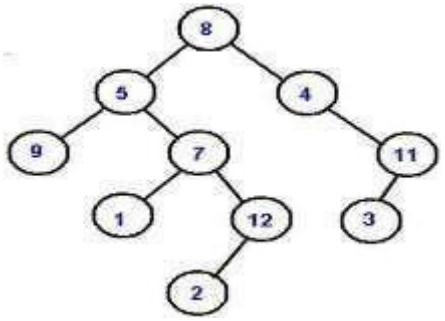
A traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal. We will consider several traversal algorithms with we group in the following two kinds

- depth-first traversal
- breadth-first traversal

There are three different types of depth-first traversals, :

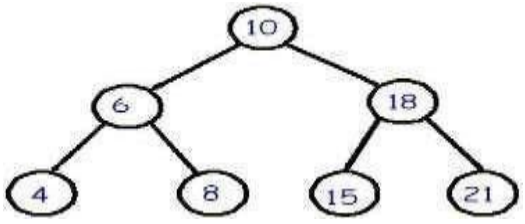
- Pre Order traversal - visit the root first and then left and right children;
- In Order traversal - visit the left child, then the root and the right child;
- Post Order traversal - visit left child, then the right child and then the root;

There is only one kind of breadth-first traversal--the level order traversal. This traversal visits nodes by levels from top to bottom and from left to right.
As an example consider the following tree and its four traversals:
Pre Order -8,5,9, 7,1,12,2,4,11,
In Order - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
Post Order -9,1,2,12,7,5,3, 11,4,8
Level Order - 8, 5, 4, 9, 7, 11, 1, 12, 3, 2



BINARY SEARCH TREE:

A particular kind of a binary tree called a Binary Search Tree (BST). The basic idea behind this data structure is to have such a storing repository that provides the efficient way of data sorting, searching and retrieving.



A BST is a binary tree where nodes are ordered in the following way:

- each node contains one key (also known as data)
- the keys in the left subtree are less then the key in its parent node, in short L<P;
- the keys in the right subtree are greater the key in its parent node, in short P<R;
- duplicate keys are not allowed.

In the following tree all nodes in the left subtree of 10 have keys < 10 while all nodes in the right subtree > 10. Because both the left and right subtrees of a BST are again search trees; the above definition is recursively applied to all internal nodes.

Declaration

```
struct TreeNode;
```

```
typedef struct TreeNode *Position;
typedef struct TreeNode *SearchTree;
struct TreeNode
{
ElementType Element;
SearchTree Left;
SearchTree Right;          }
```

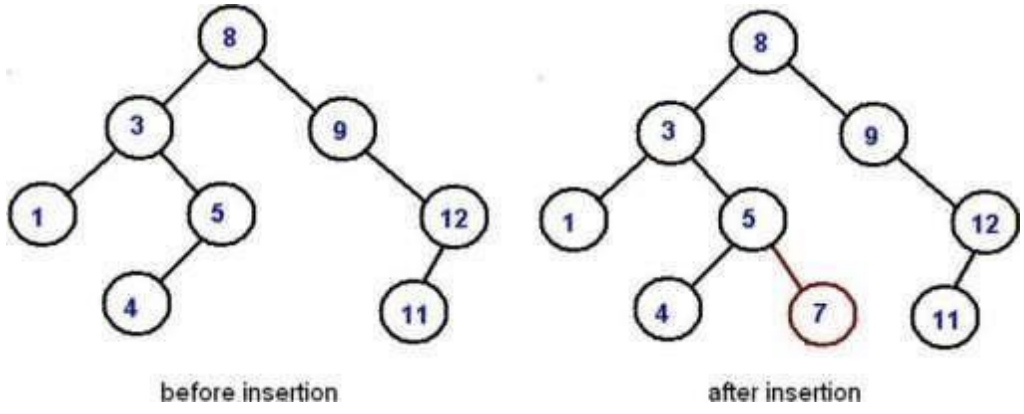
Routine to make an empty tree

This operation is mainly for initialization. Some programmers prefer to initialize the first element as a one-node tree, but our implementation follows the recursive definition of trees more closely.

```
SearchTree MakeEmpty( SearchTree T)
{
   if( T != NULL)
   {
            MakeEmpty(T->Left);
            MakeEmpty(T->Right);
            free(T);
   }
   return NULL;
}
```

Insertion

The insertion procedure is quite similar to searching. We start at the root and recursively go down the tree searching for a location in a BST to insert a new node. If the element to be inserted is already in the tree, we are done (we do not insert duplicates). The new node will always replace a NULL reference.



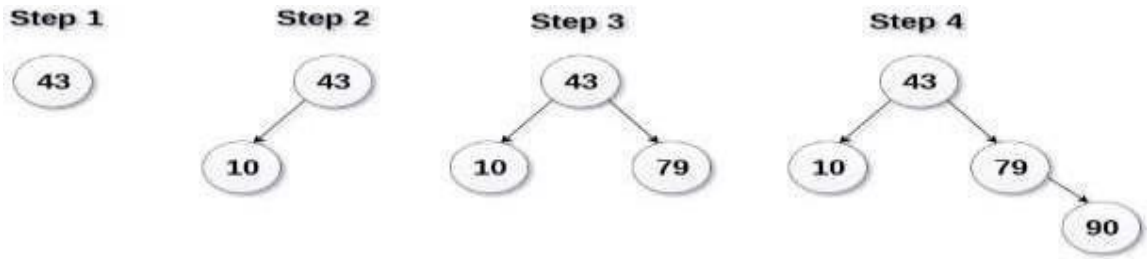
before insertion after insertion

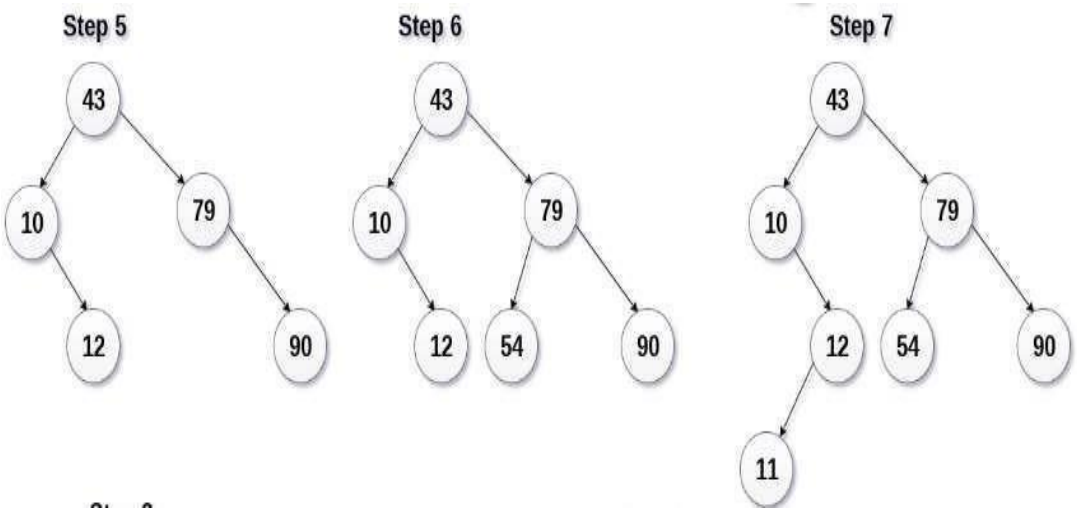
Routine for insertion

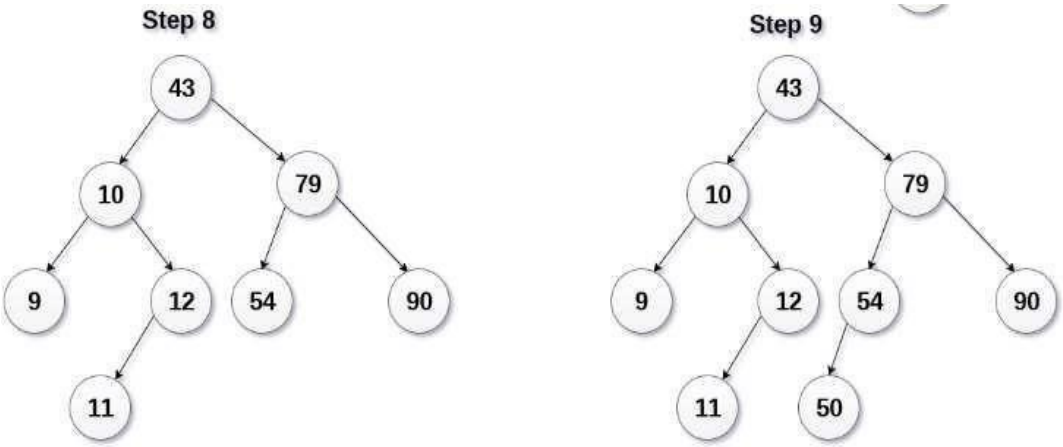
```
SearchTree Insert( ElementType X, SearchTree T)
{
 if(T == NULL)
 {
          T=malloc(sizeof(struct TreeNode));
 if(T == NULL)
                   FatalError("Out of Space !!!");
```

```
        else
        {

T->Element = X;
T->Left = T->Right = NULL;

    }

    else
        if(X < T->Element)
        T->Left = Insert(X, T->Left);
    else
    if(X > T->Element)
        T->Right = Insert(X, T->Right);
    Return T;
    }
```

Example: **43, 10, 79, 90, 12, 54, 11, 9, 50**



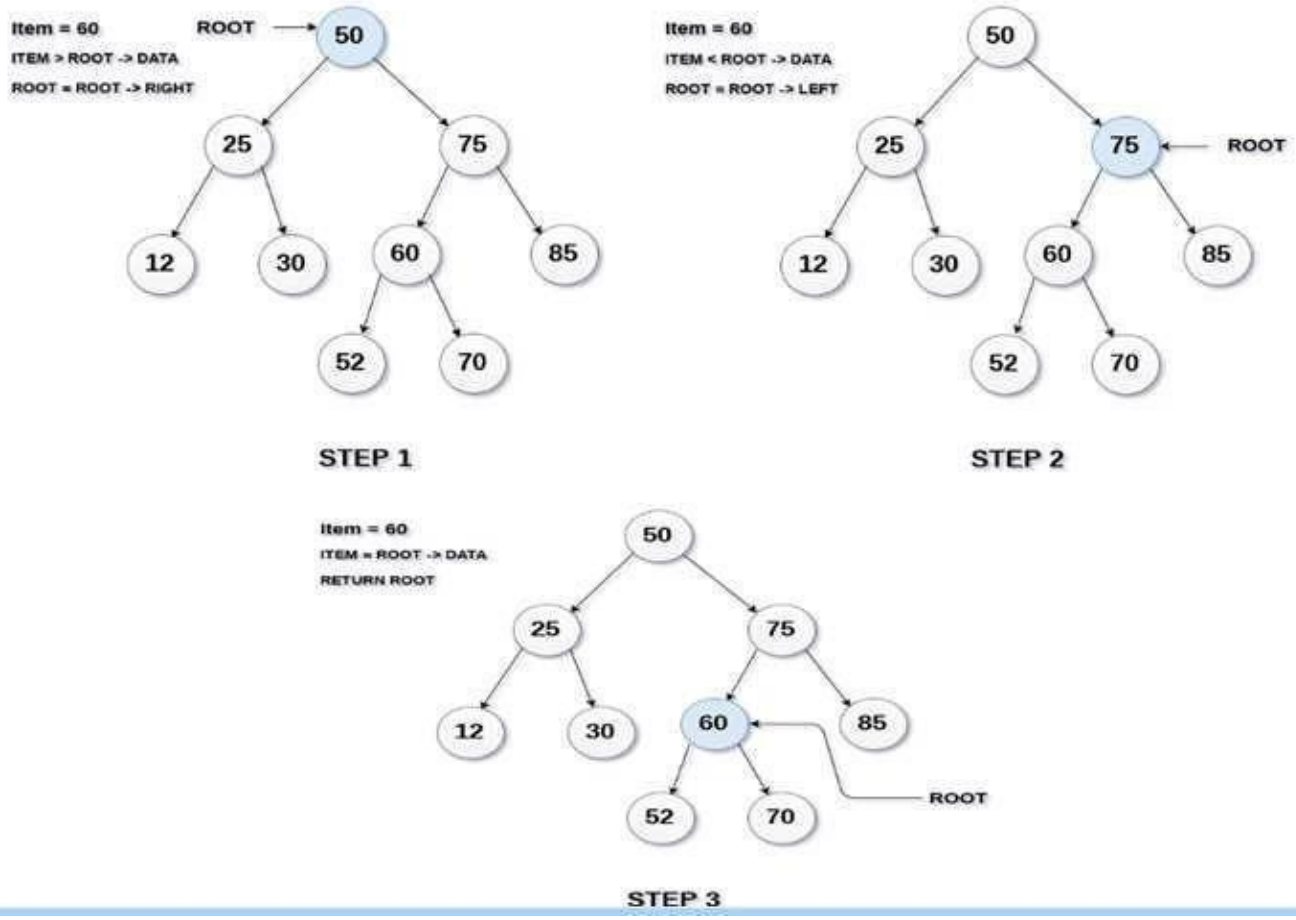




Searching

Searching means **finding or locating** some specific element or node within a data structure. However, searching for some specific node in binary search tree is pretty easy due to the fact that, element in BST is stored in a particular order.

1. Compare the element with the root of the tree.
2. If the item is matched then return the location of the node.
3. Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
4. If not, then move to the right sub-tree.
5. Repeat this procedure recursively until match found.
6. If element is not found then return NULL.



Routine Find operation

```
Position Find( ElementType X, SearchTree T)
{
 if( T == NULL)
        return NULL;
 if( X < T->Element)
        return Find(X, T->Left);
 else
 if(X > T->Element)
```

```
        return Find(X, T->Right);
else    return T;

}
```

FindMin & FindMax:

These routines return the position of the smallest and largest elements in the tree, respectively.

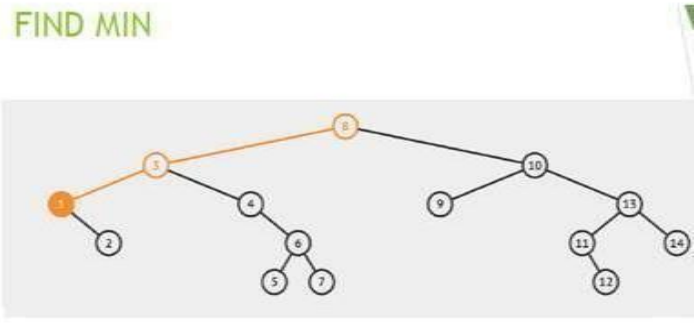
To perform a findmin, start at the root and go left as long as there is a left child. The stopping point is the smallest element.

The findmax routine is the same, except that branching is to the right child.

Routine to Find Minimum Value

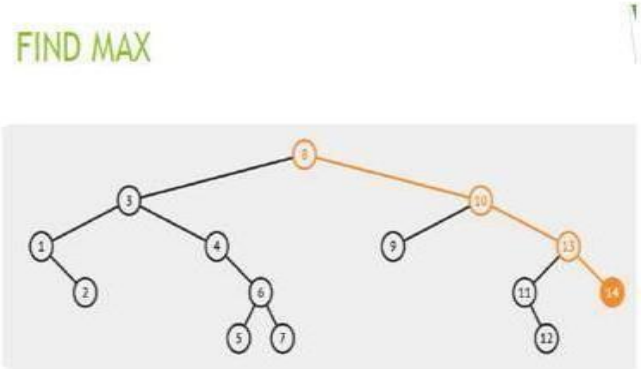
```
Position findmin( SearchTree T )
{
if( T == NULL )
        return NULL;
else
        if( T->left == NULL )
        return( T );
else
        return( findmin ( T->left ) );
}
```

```
Position
FindMin( SearchTree T)
{
   if(T == Null)
        return NULL;
   else
   if(T->Left == NULL)
        return T;
   else
    return FindMin(T->Lef
}
```



Routine to Find Maximum Value

```
Position
FindMax( SearchTree T)
{
   if(T == Null)
        return NULL;
   else
   if(T->Right == NULL)
        return T;
   else
    return FindMax(T->Rig

}
```

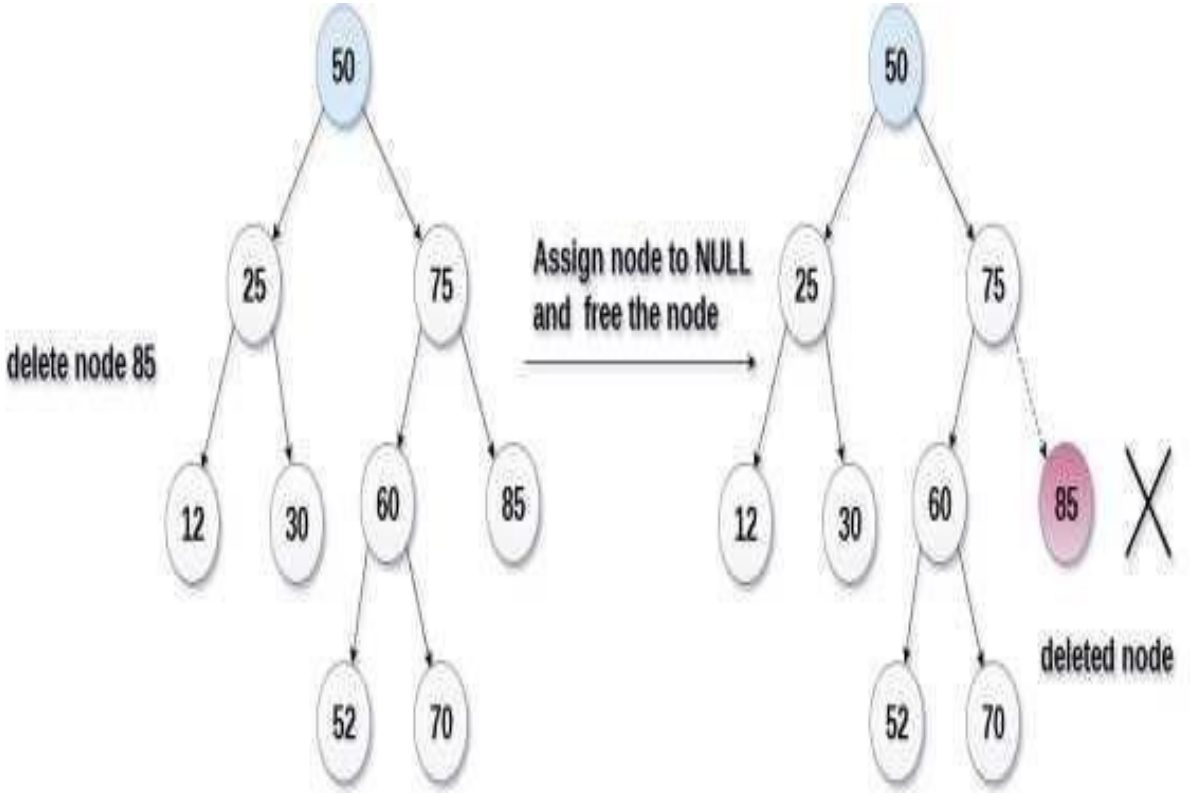


Deletion

- Delete function is used to delete the specified node from a binary search tree.
- However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate.
- There are **three situations** of deleting a node from binary search tree.

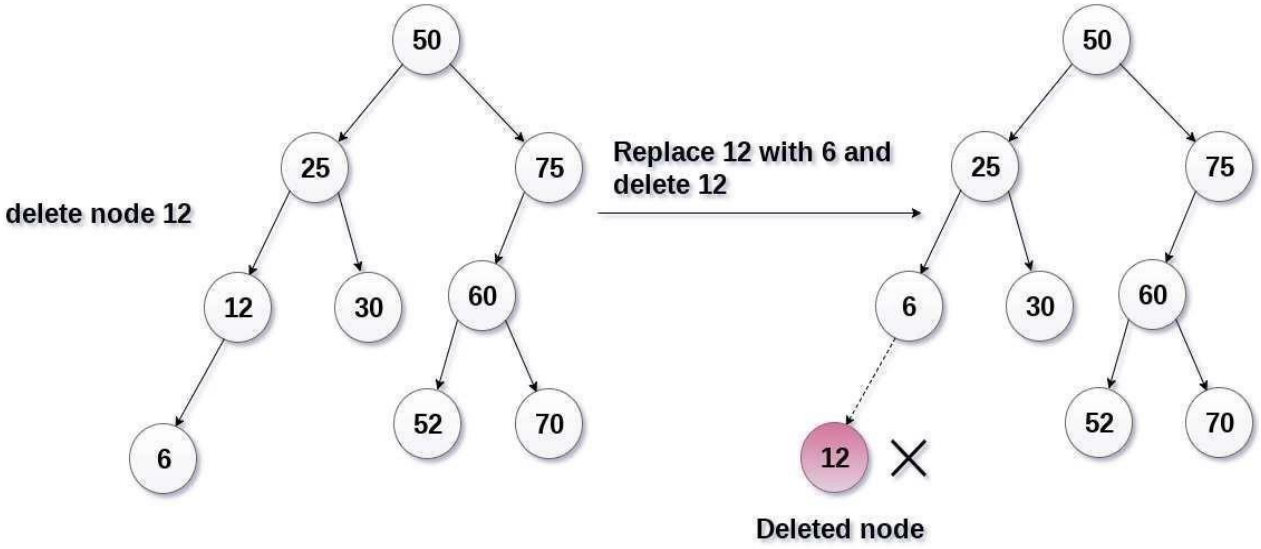
Case: 1 The node to be deleted is a leaf node

It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



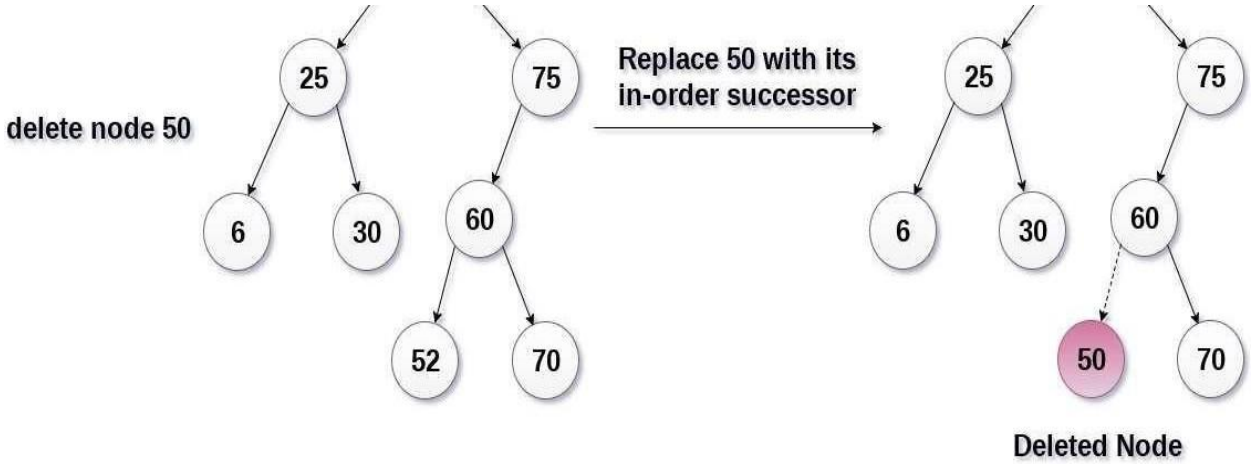
Case: 2 The node to be deleted has only one child.

- In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.
- In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.



Case : 3 The node to be deleted has two children

- It is a bit complex case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree.
- After the procedure, replace the node with NULL and free the allocated space.
- In the following image, the node 50 is to be deleted which is the root node of the tree.
- The in- order traversal of the tree given below.
- 6, 25, 30, 50, 52, 60, 70, 75.
- Replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



-

Routine for deletion

```
SearchTree
Delete(ElementType X, SearchTree T)
{
        Position Tmpcell;
        if(T == NULL)
                Error("Element not found");
        else
        if(X < T->Element)                      /* Go left */
                T->Left = Delete(X, T->Left);
        else
        if(X > T->Element)                      /* Go right */

    T->Right = Delete(X, T->Right);
else    /* Found element to be deleted */
if( T->Left && T->Right) /* Two children */
{
        /* Replace With smallest in right subtree */
        Tmpcell = FindMin(T->Right);
        T->Element = Tmpcell - >Element;
        T->Right = Delete(T->Element, T->Right);
}
else
{
        Tmpcell = T;
        if(T->Left == NULL) /* Also handles 0 children */
                T = T->Right:
```

```
        else if(T->Right == NULL)
                T = T->Left;
        free(Tmpcell);
}
        return T;
}
```

Advantages:

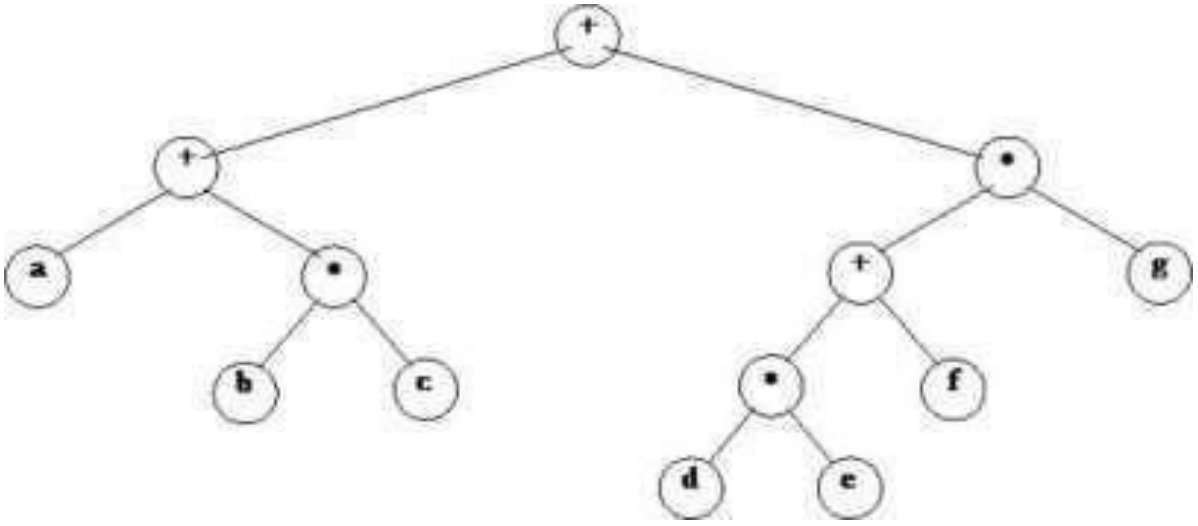
- Searching become very efficient in a binary search tree.
- The binary search tree is considered as efficient data structure in compare to arrays and linked lists.
- Searching for an element in a binary search tree takes o($\log_2 n$) time. In worst case, the time it takes to search an element is 0(n).
- It also speed up the insertion and deletion operations as compare to that in array and linked list.

Applications of BST

1) Used to express arithmetic expressions
2) Used to evaluate expression trees.
3) Used for indexing IP addresses.
4) It is used to implement dictionary.
5) It is used to implement multilevel indexing in DATABASE.
7) To implement Huffman Coding Algorithm.
8) It is used to implement searching Algorithm.
9) Implementing routing table in router.

EXPRESSION TREES:

Trees are used in many other ways in the computer science. Compilers and database are two major examples in this regard. In case of compilers, when the languages are translated into machine language, tree-like structures are used. We have also seen an example of expression tree comprising the mathematical expression. Let's have more discussion on the expression trees. We will see what are the benefits of expression trees and how can we build an expression tree. Following is the figure of an expressiontree.



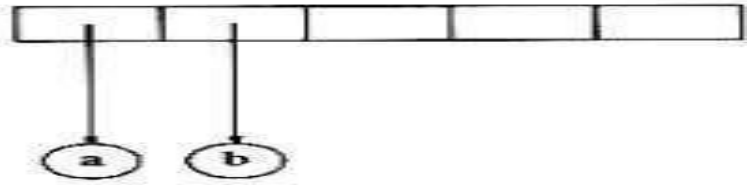
In the above tree, the expression on the left side is a + b * c while on the right side, we have d * e + f * g. If you look at the figure, it becomes evident that the inner nodes contain operators while leaf nodes have

Construction Expression Tree

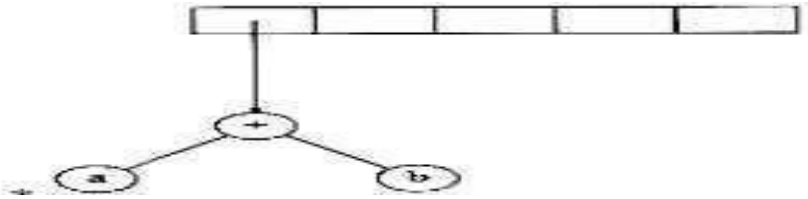
1. Read one symbol at a time from the postfix expression
2. Check whether the symbol is an operand or operator
    1. If the symbol is an operand, create a one-node tree and push a pointer on to the stack
    2. If the symbol is an operator pop two pointers from the stack namely T1 and T2 and form a new tree with root as the operator and T2 as a left child and T1 as a right child. A pointer to this new tree is then pushed onto the stack.

    For example: a b + c d e + * *

- The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack.*
- *For convenience, we will have the stack grow from left to right in the diagrams.

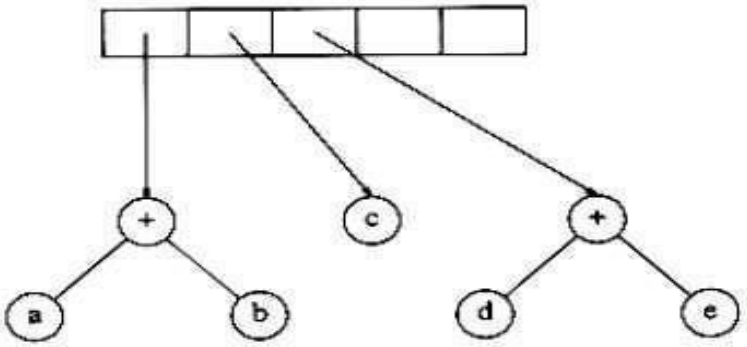


- Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.*

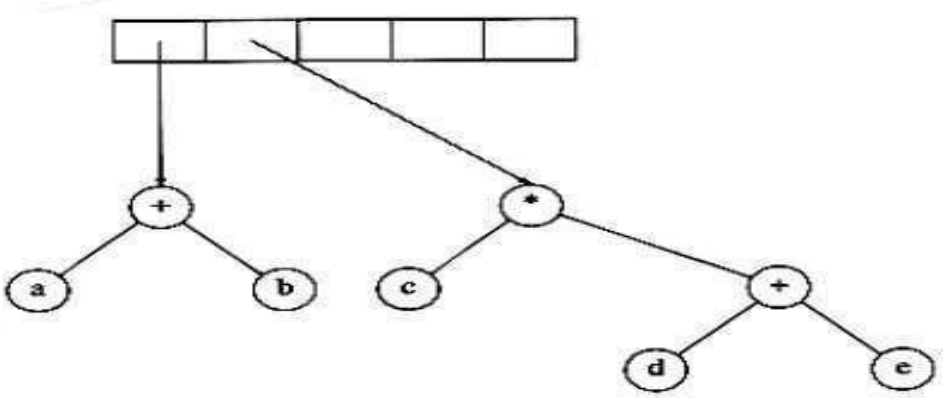


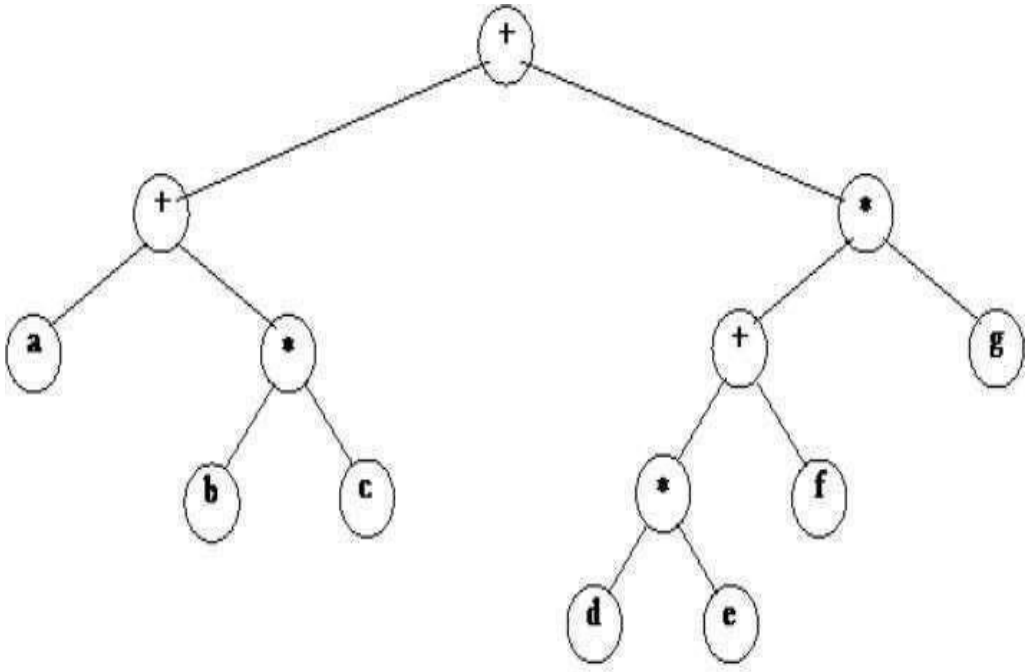
- Next, c, d, and e are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.

    - Now a '+' is read, so two trees are merged



- Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root

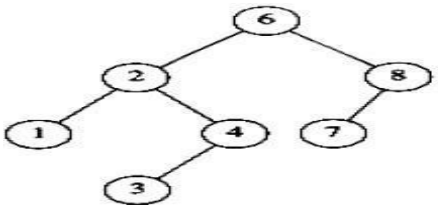


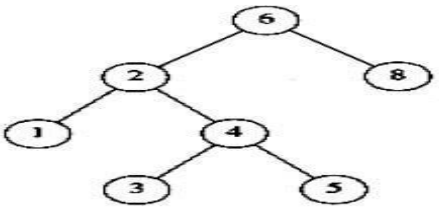


Postorder traversal: a b c * + d e * f + g * +

AVL Tree : - (Adelson - Velskill and Landis)

- An AVL tree is a binary search tree except that for every node in the tree, the height of the left and right sub trees can differ by atmost1.
- The height of the empty tree is defined to be - 1. A balance factor is the height of the left subtree minus height of the right subtree.
- For an AVL tree all balance factor should be +1, 0, or -1. If the balance factor of any node in an AVL tree becomes less than -1 or greater than 1, the tree has to be balanced by making either single or double rotations.





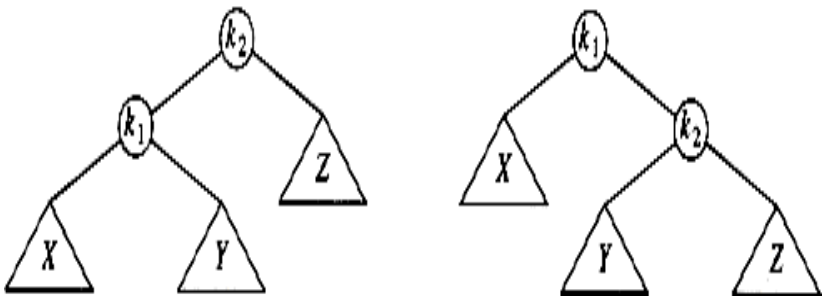
In Figure, the tree on the left is an AVL tree, but the tree on the right is not.
When we do an insertion, we need to update all the balancing information for the nodes on the path back to the root, but the reason that insertion is difficult is that inserting a node could violate the AVL tree property. Inserting a node into the AVL tree would destroy the balance condition.

An AVL tree causes imbalance, when any one of the following conditions occur

Case 1: An insertion into the left subtree of the left child of node.
Case 2: An insertion into the right subtree of the left child of node.
Case 3: An insertion into the left subtree of the right child of node.
Case 4: An insertion into the right subtree of the right child of node.

These imbalances can be overcome by
1. SingleRotation
2. DoubleRotation.

Single Rotation with left(case 1)



The two trees in the above Figure contain the same elements and are both binary search trees.
First of all, in both trees k1 < k2.
Second, all elements in the subtree X are smaller than k1 in both trees.

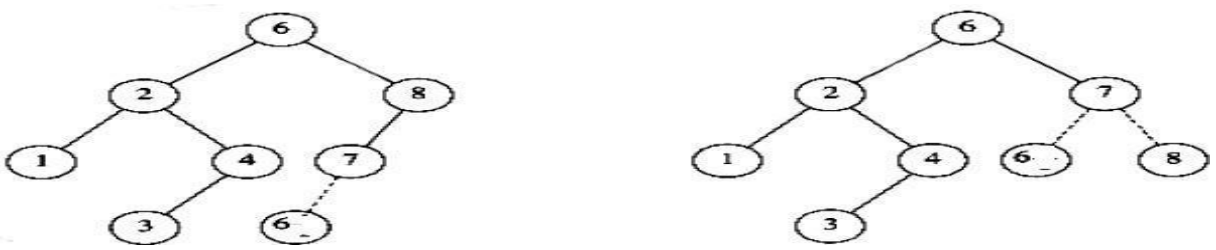
Third, all elements in subtree Z are larger than k2. Finally, all elements in subtree Y are in between k1 and k2. The conversion of one of the above trees to the other is known as **a rotation.**

In an AVL tree, if an insertion causes some node in an AVL tree to lose the balance property:

Do a rotation at that node.

The basic algorithm is to start at the node inserted and travel up the tree, updating the balance information at every node on the path.

Ex.



In the above figure, after the insertion of the in the original AVL tree on the left, node 8 becomes unbalanced. Thus, we do a single rotation between 7 and 8, obtaining the tree on the right.

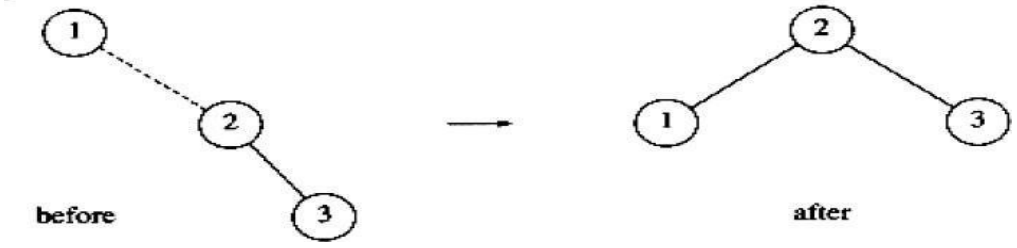
Routine Single rotation with left

```
static Position SingleRotateWithLeft ( Position K2 )
{
Position K1;
K1 =K2->Left;
 K2->Left = K1->Right;
K1->Right = K2;
 K2->Height = Max ( Height (K2 ->Left), Height(K2->Right))+1;
 K1->Height = Max ( Height (K1 ->Left), Height(K1->Right))+1;
 return K1;
 }
```

Single rotation with right:(case 4)

Suppose we start with an initially empty AVL tree and insert the keys 1 through 3 in sequential order. The first problem occurs when it is time to insert key 3, because the AVL property is violated at the root. We perform a single rotation between the root and its right child to fix the problem.

The tree is shown in the following figure, before and after the rotation.

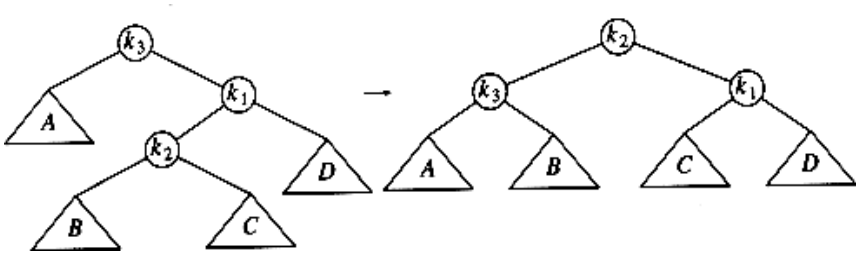


static Position SingleRotateWithRight (Position K1)

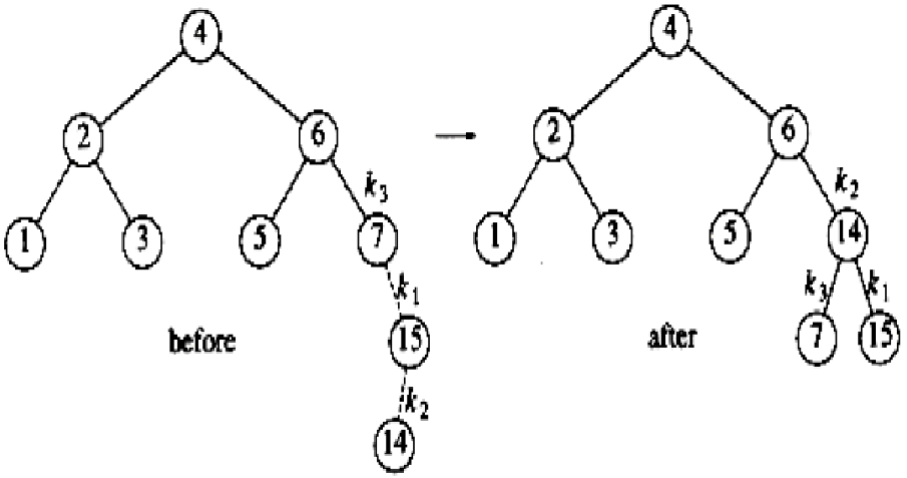
```
 {
Position K2;
 K2 = K1->Right;
 K1->Right = K2->Left;
K2->Left = K1;
K2->Height = Max ( Height (K2 ->Left), Height(K2->Right))+1;
K1->Height = Max ( Height (K1 ->Left), Height(K1->Right))+1;
return K2;
}
```

Double Rotation

(Right-left) double rotation (case 2)



Example:



Routine to perform double rotation with left

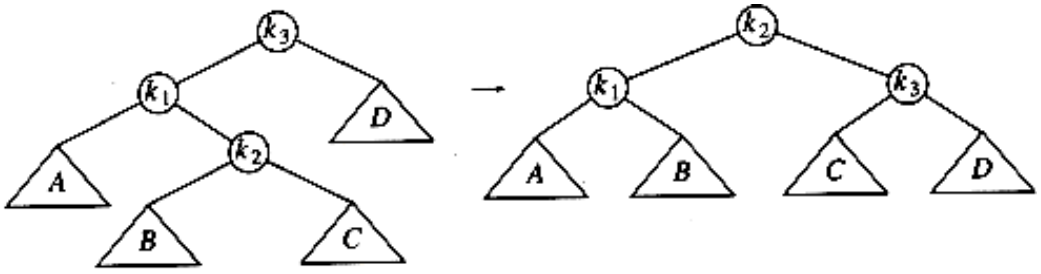
```
static Position DoubleRotateWithLeft( Position K3)
 {
/* Rotate between K1 and K2 */

 K3 -> Left = SingleRotateWithRight(K3 ->Left);

 /* Rotate between K3 and K2 */
 return SingleRotateWithLeft (K3);
}
```

(Left-Right)Double Rotation with Right(case 3)

Double rotation with right is used to perform case 4. An insertion into the left sub tree of the right child of node α. Double Rotation with right is performed by first performing single rotation with left, and then performing single rotation with right.

Routine to perform double rotation with right

```
static Position DoubleRotateWithRight( Position K1)
{
 K1 -> Right = SingleRotateWithLeft(K1 ->Right);
return SingleRotateWithRight (K1);
}
```



Example(Refer class notes)

Function to compute height of an Avl node

```
static int Height( Position P)
 {
 if( P = = NULL) return -1;
 else return P -> Height;
}
```

Insertion into an Avl tree

```
AvlTree Insert( ElementType X, AvlTree T)
{
 if( T = = NULL)
 {
 T = malloc(sizeof(struct AvlNode));
if( T = = NULL) FatalError( " Out of Space");
else
{
 T -> Element = X;
```

```
 T -> Height = 0;
 T -> Left = T -> Right = NULL;
 }
}
else if( X< T -> Element)
{
T -> Left = Insert(X, T-> Left);
 if( Height( T -> Left) – Height( T -> Right) = = 2)
if( X < T -> Left -> Element)
T = SingleRotateWithLeft( T );
 else
 T = DoubleRotateWithLeft(T);
}
 else if( X > T -> Element)
 {
 T -> Right = Insert(X, T-> Right);
if( Height( T -> Right) – Height( T -> Left) = = 2)
 if( X > T -> Right -> Element)
 T = SingleRotateWithRight( T );
else
  T = DoubleRotateWithRight(T);
}
T -> Height = Max( Height( T-> Left), Height( T -> Right)) + 1; return T;
 }
```

PRIORITY QUEUE:(HEAP)

A priority queue is a data structure that allows at least the following two operations: insert, which does the obvious thing, and delete_min, which finds, returns and removes the minimum element in the heap. The insert operation is the equivalent of enqueue, and delete_min is the priority queue equivalent of the queue's dequeue operation.

HEAPS:

Heap data structure is a specialized binary tree based data structure. Heap is a binary tree with special characteristics. In a heap data structure, nodes are arranged based on thier value.

There are two types of heap data structures and they are as follows...

1. **MaxHeap**
2. **MinHeap**

Every heap data structure has the following properties...

Property #1 (Ordering): Nodes must be arranged in a order according to values based on Max heap or Min heap.

Property #2 (Structural): All levels in a heap must full, except last level and nodes must be filled from left to right strictly.

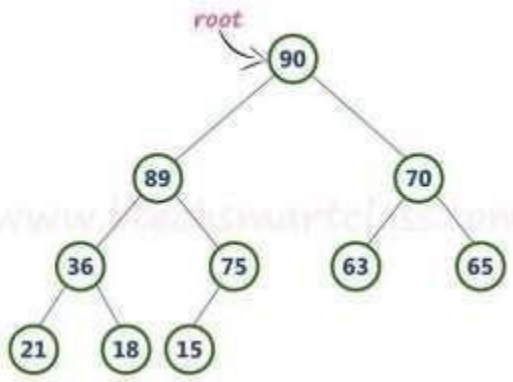
Max Heap

Max heap data structure is a specialized full binary tree data structure except last leaf node can be alone. In a max heap nodes are arranged based on node value.

Max heap is defined as follows...

Max heap is a specialized full binary tree in which every parent node contains greater or equal value than its child nodes. And last leaf node can be alone.

EXAMPLE:



Above tree is satisfying both Ordering property and Structural property according to the Max Heap data structure.

Operations on Max Heap

The following operations are performed on a Max heap data structure...

1. **FindingMaximum**
2. **Insertion**
3. **Deletion**

Finding Maximum Value Operation in Max Heap

Finding the node which has maximum value in a max heap is very simple. In max heap, the root node has the maximum value than all other nodes in the max heap. So, directly we can display root node value as maximum value in max heap.

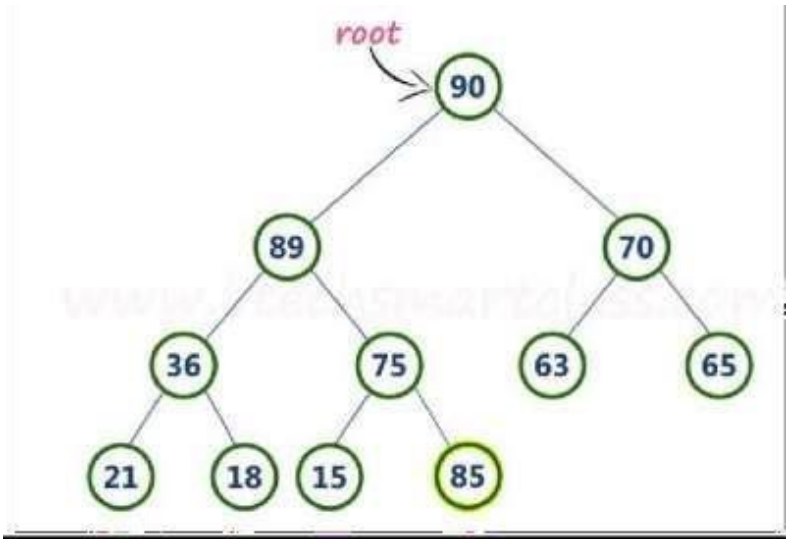
Insertion Operation in Max Heap

Insertion Operation in max heap is performed as follows...

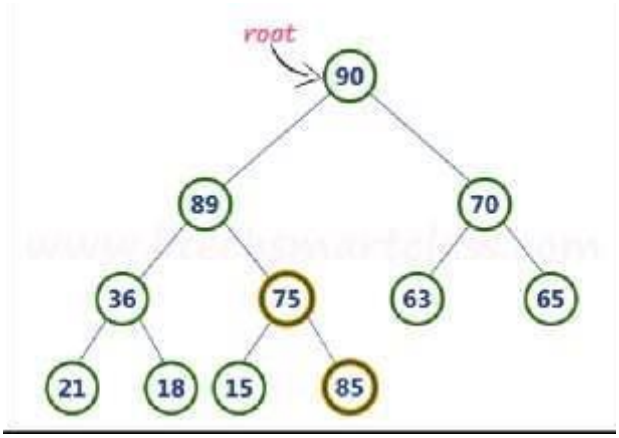
- **Step 1:** Insert the **newNode** as **last leaf** from left toright.
- **Step 2:** Compare **newNode value** with its**Parentnode**.
- **Step 3:** If **newNode value is greater** than its parent, then **swap** bothofthem.
- **Step 4:** Repeat step 2 and step 3 until newNode value is less than its parent nede (or) newNode reachedtoroot.

Example
Consider the above max heap. **Insert a new node with value 85.**

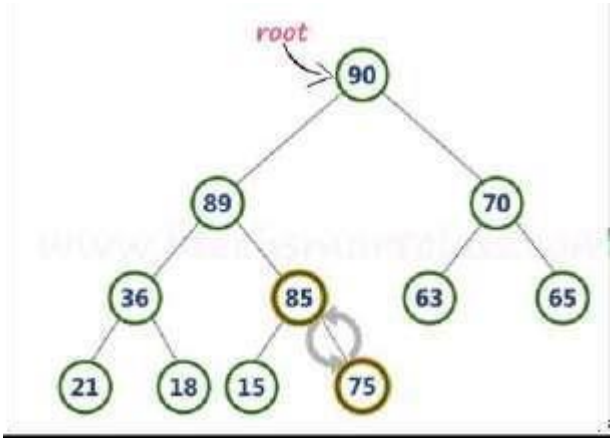
- **Step 1:** Insert the **newNode** with value 85 as **last leaf** from left to right. That means new Node is added as a right child of node with value 75. After adding max heap is as follows...



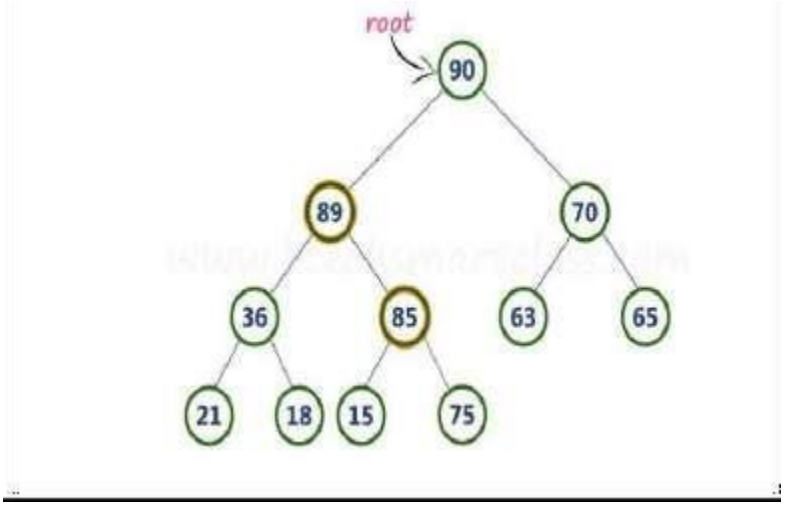
Step 2: Compare **newNode value (85)** with its **Parent node value (75)**. That means **85 > 75**



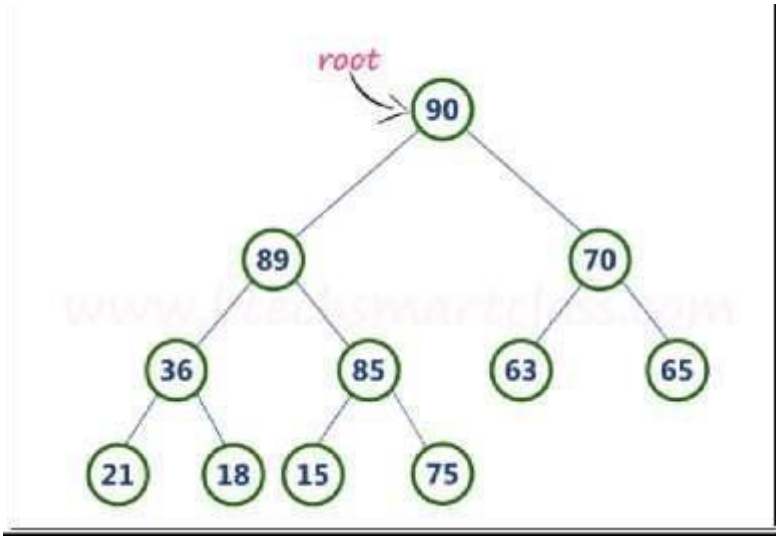
Step 3: Here **newNode value (85) is greater** than its **parent value (75)**, then **swap** both of them. After wsapping, max heap is as follows...



Step 4: Now, again compare newNode value (85) with its parent nede value (89).



Here, newNode value (85) is smaller than its parent node value (89). So, we stop insertion process. Finally, heap after insetion of a new node with value 85 is asfollows...



Deletion Operation in Max Heap

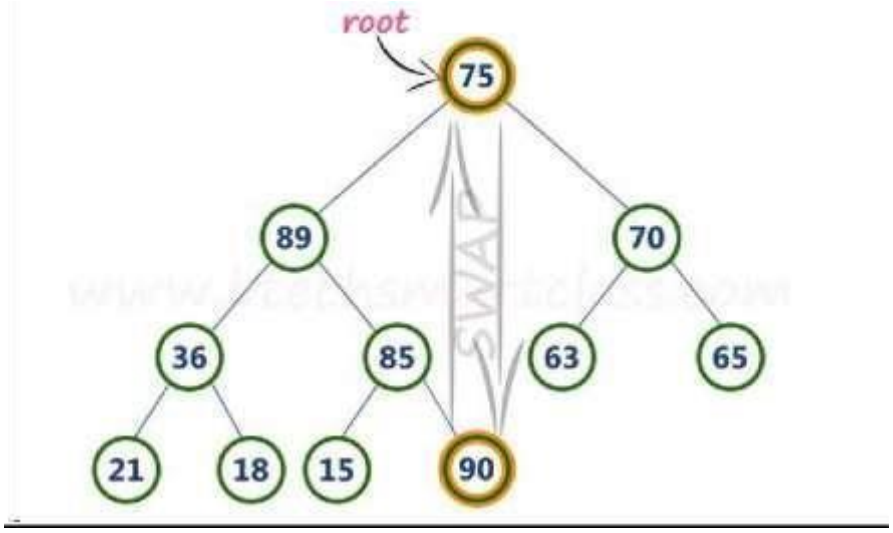
In a max heap, deleting last node is very simple as it is not disturbing max heap properties.

Deleting root node from a max heap is title difficult as it disturbing the max heap properties. We use the following steps to delete root node from a max heap...

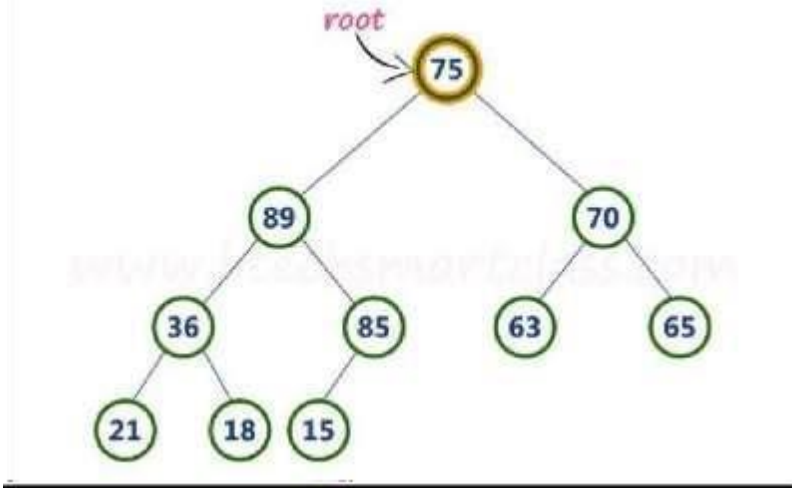
- **Step 1: Swap** the **root** node with **last** node in max heap
- **Step 2: Delete** last node.
- **Step 3:** Now, compare **root value** with its **leftchild value**.
- **Step 4:** If **root value is smaller** than its left child, then compare **left child** with its **right sibling**. Else goto**Step6**
- **Step 5:** If **left child value is larger** than its **right sibling**, then **swap root** with **leftchild**. Otherwise **swap root** with its **right child**.
- **Step6:**If**rootvalueislarger**thanitsleftchild,thencompare**rootvalue**withits**rightchild** value.
- **Step7:**If **root values smaller** than its **right child** ,then **swap root** with **rightchild** .otherwise **stop the process**.
- **Step 8:** Repeat the same until root node is fixed at its exact position.

Consider the above max heap. **Delete root node (90) from the max heap.**

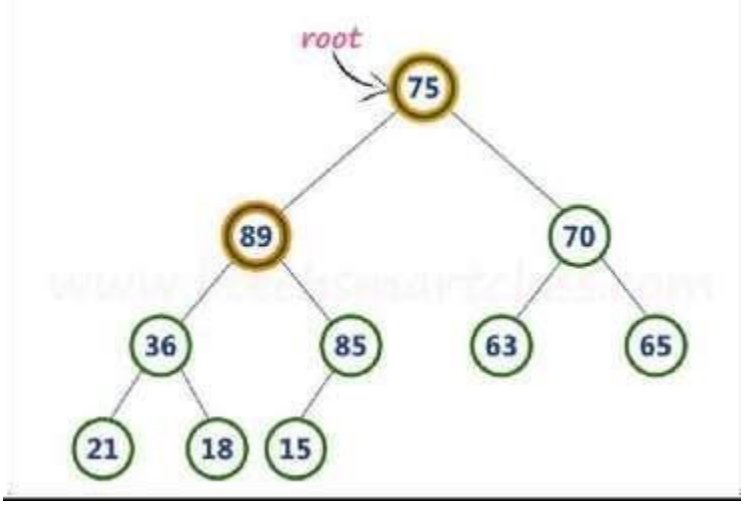
- **Step 1: Swap** the **root node (90)** with **last node 75** in max heap After swapping max heap is as follows



Step 2: Delete last node. Here node with value 90. After deleting node with value 90 from heap, max heap is as follows...



Step 3: Compare **root node (75)** with its **left child (89)**.



Here, **root value (75) is smaller** than its left child value (89). So, compare left child (89) with its right sibling (70).

Step 4: Here, **left child value (89) is larger** than its **right sibling (70)**, So, **swap root (75)** with **left child (89)**.

Step 5: Now, again compare **75** with its **left child (36)**.

Here, node with value **75** is larger than its left child. So, we compare node with value **75** is compared with its right child **85**.

Step 6: Here, node with value **75** is smaller than its **right child (85)**. So, we swap both of them. After swapping max heap is as follows...

Step 7: Now, compare node with value **75** with its left child (**15**).

Here, node with value **75** is larger than its left child (**15**) and it does not have right child. So we stop the process.

Finally, max heap after deleting root node (**90**) is as follows...

Applications of Heap Data Structure

Heap Data Structure is generally taught with Heapsort. Heapsort algorithm has limited uses because Quicksort is better in practice. Nevertheless, the Heap data structure itself is enormously used. Following are some uses other than Heapsort.

Priority Queues: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time. Binomoial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also in O(logn) time which is a O(n) operation in Binary Heap. Heap Implemented priority queues are used in Graph algorithms likePrim's Algorithm and Dijkstra's algorithm.

BINARY HEAP:

A binary heap is a complete binary tree in which every node satisfies the heap property which states that:

If B is a child of A, then key(A) ≥ key(B)

This implies that elements at every node will be either greater than or equal to the element a its left and right child. Thus, the root node has the highest key value in the heap. Such a heap is commonly known as a max-heap**.**

Alternatively, elements at every node will be either less than or equal to the element at its left and right child. Thus, the root has the lowest key value. Such a heap is called a min-heap.

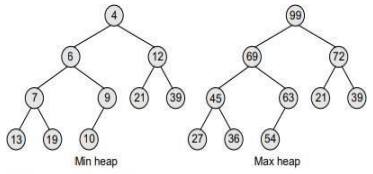


Figure Binary heaps

Example Build a max heap H from the given set of numbers: 45, 36, 54, 27, 63, 72, 61, and 18. Also draw the memory representation of the heap.

Solution

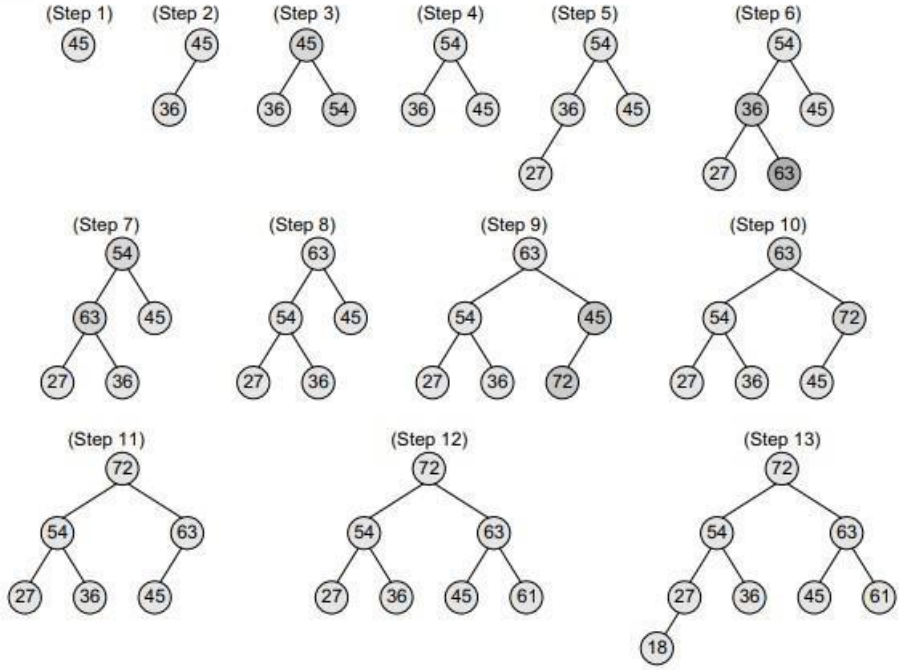


Figure 12.5

The memory representation of H can be given as shown in Fig. 12.6.

HEAP[1]	HEAP[2]	HEAP[3]	HEAP[4]	HEAP[5]	HEAP[6]	HEAP[7]	HEAP[8]	HEAP[9]	HEAP[10]
72	54	63	27	36	45	61	18		

Figure Memory representation of binary heap H

```
Step 1: [Add the new value and set its POS]
        SET N = N + 1, POS = N
Step 2: SET HEAP[N] = VAL
Step 3: [Find appropriate location of VAL]
        Repeat Steps 4 and 5 while POS > 1
Step 4:     SET PAR = POS/2
Step 5:     IF HEAP[POS] <= HEAP[PAR],
            then Goto Step 6.
            ELSE
                  SWAP HEAP[POS], HEAP[PAR]
                  POS = PAR
            [END OF IF]
      [END OF LOOP]
Step 6: RETURN
```

Figure Algorithm to insert an element in a max heap

Deleting an Element from a Binary Heap

Example Consider the max heap H shown in Fig. 12.8 and delete the root node's value.

Solution

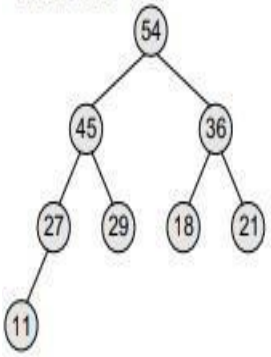
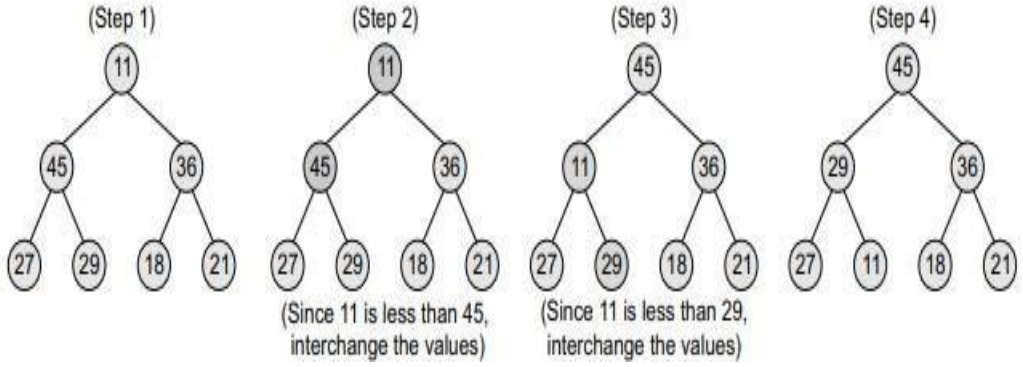


Figure 12.8 Binary heap

Consider a max heap H having n elements. An element is always deleted from the root of the heap. So, deleting an element from the heap is done in the following three steps:

1. Replace the root node's value with the last node's value so that H is still a complete binary tree but not necessarily a heap.
2. Delete the last node.
3. Sink down the new root node's value so that H satisfies the heap property. In this step, interchange the root node's value with its child node's value (whichever is largest among its children).

Here, the value of root node = 54 and the value of the last node = 11. So, replace 54 with 11 and delete the last node.




```
Step 1: [Remove the last node from the heap]
        SET LAST = HEAP[N], SET N = N - 1
Step 2: [Initialization]
        SET PTR = 1, LEFT = 2, RIGHT = 3
Step 3: SET HEAP[PTR] = LAST
Step 4: Repeat Steps 5 to 7 while LEFT <= N
Step 5: IF HEAP[PTR] >= HEAP[LEFT] AND
         HEAP[PTR] >= HEAP[RIGHT]
              Go to Step 8
        [END OF IF]
Step 6: IF HEAP[RIGHT] <= HEAP[LEFT]
              SWAP HEAP[PTR], HEAP[LEFT]
              SET PTR = LEFT
        ELSE
              SWAP HEAP[PTR], HEAP[RIGHT]
              SET PTR = RIGHT
        [END OF IF]
Step 7: SET LEFT = 2 * PTR and RIGHT = LEFT + 1
        [END OF LOOP]
Step 8: RETURN
```

Figure 12.10 Algorithm to delete the root element from a max heap

UNIT IV MULTIWAY SEARCH TREES AND GRAPHS 9

B-Tree – B+ Tree – Graph Definition – Representation of Graphs – Types of Graph – Breadth-first traversal – Depth-first traversal — Bi-connectivity – Euler circuits – Topological Sort – Dijkstra's algorithm – Minimum Spanning Tree – Prim's algorithm – Kruskal's algorithm

B-Tree:

A B tree is a specialized M-way tree developed by Rudolf Bayer and Ed McCreight in 1970 that is widely used for disk access.

A B tree of order m can have a maximum of m–1 keys and m pointers to its sub-trees.

A B tree may contain a large number of key values and pointers to sub-trees.

Storing a large number of keys in a single node keeps the height of the tree relatively small.

A B tree is designed to store sorted data and allows search, insertion, and deletion operations to be performed in logarithmic amortized time.

A B tree of order m (the maximum number of children that each node can have) is a tree with all the properties of an M-way search tree.

In addition it has the following properties:

Every node in the B tree has at most (maximum) m children.

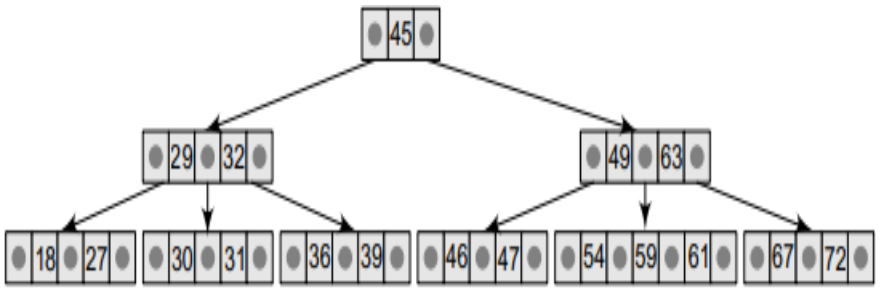
Every node in the B tree except the root node and leaf nodes has at least (minimum) m/2children. This condition helps to keep the tree bushy so that the path from the root node to
the leaf is very short, even in a tree that stores a lot of data.

The root node has at least two children if it is not a terminal (leaf) node.

All leaf nodes are at the same level.

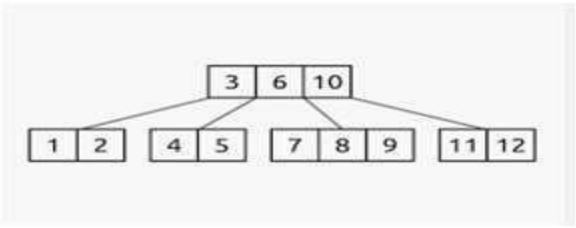
An internal node in the B tree can have n number of children, where 0 £ n £ m. It is not necessary that every node has the same number of children, but the only restriction is that the node should have at least m/2 children.

Ex. As B tree of order 4:



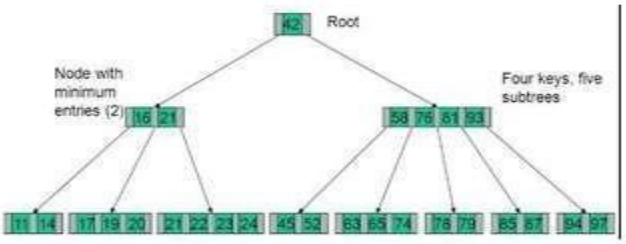
B tree of order 4

B tree Example With Order m=4 & m=5



m=4

Maximum Children - 4
Minimum Children - Ceil (m/2) = 2
Maximum Keys - (m-1) =3
Min Keys - Ceil(m/2)-1 =1



m=5

Maximum Children - 5
Minimum Children - Ceil (m/2) = 3
Maximum Keys - (m-1) =4
Min Keys - Ceil(m/2)-1 =2

While performing insertion and deletion operations in a B tree, the number of child nodes may change. So, in order to maintain a minimum number of children, the internal nodes may be joined or split.

Inserting a New Element in a B Tree:

In a B tree, all insertions are done at the leaf node level. A new value is inserted in the B tree

using the algorithm given below.

1. Search the B tree to find the leaf node where the new key value should be inserted.

2. If the leaf node is not full, that is, it contains less than m–1 key values, then insert the new element in the node keeping the node's elements ordered.

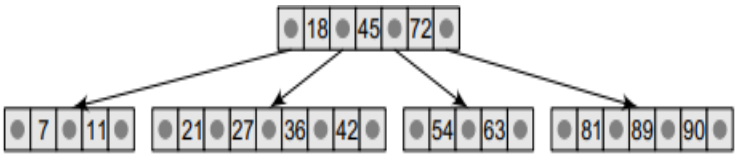
3. If the leaf node is full, that is, the leaf node already contains m–1 key values, then

(a) insert the new value in order into the existing set of keys,

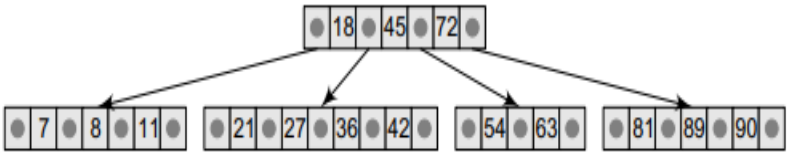
(b) split the node at its median into two nodes (note that the split nodes are half full), and

(c) push the median element up to its parent's node. If the parent's node is already full, then split the parent node by following the same steps.

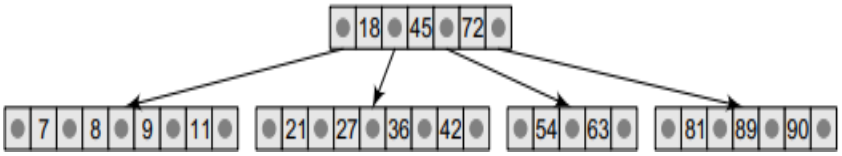
Ex.The B tree of order 5 given below and insert 8, 9, 39, and 4 into it.



Step 1: Insert 8

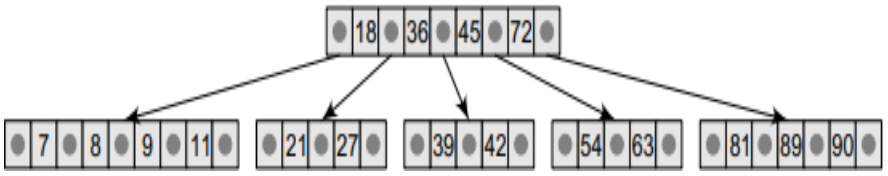


Step 2: Insert 9



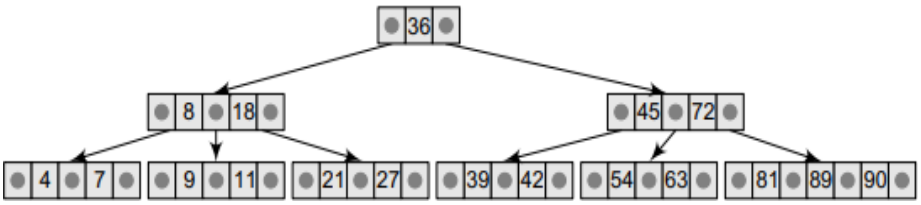
Till now, we have easily inserted 8 and 9 in the tree because the leaf nodes were not full. But now, the node in which 39 should be inserted is already full as it contains four values. Here we split the nodes to form two separate nodes. But before splitting, arrange the key values in order (including the new value). The ordered set of values is given as 21, 27, 36, 39, and 42. The median value is 36, so push 36 into its parent's node and split the leaf nodes.

Step 3: Insert 39



Now the node in which 4 should be inserted is already full as it contains four key values. Here we split the nodes to form two separate nodes. But before splitting, we arrange the key values in order (including the new value). The ordered set of values is given as 4, 7, 8, 9, and 11. The median value is 8, so we push 8 into its parent's node and split the leaf nodes. But again, we see that the parent's node is already full, so we split the parent node using the same procedure

Step 4: Insert 4



Deleting an Element from a B Tree:

Like insertion, deletion is also done from the leaf nodes. There are two cases of deletion. In the

first case, a leaf node has to be deleted. In the second case, an internal node has to be deleted. Let us first see the steps involved in deleting a leaf node.

1. Locate the leaf node which has to be deleted.

2. If the leaf node contains more than the minimum number of key values (more than m/2

elements), then delete the value.

3. Else if the leaf node does not contain m/2 elements, then fill the node by taking an element

either from the left or from the right sibling.

(a) If the left sibling has more than the minimum number of key values, push its largest key into its parent's node and pull down the intervening element from the parent node to the leaf node where the key is deleted.

(b) Else, if the right sibling has more than the minimum number of key values, push its

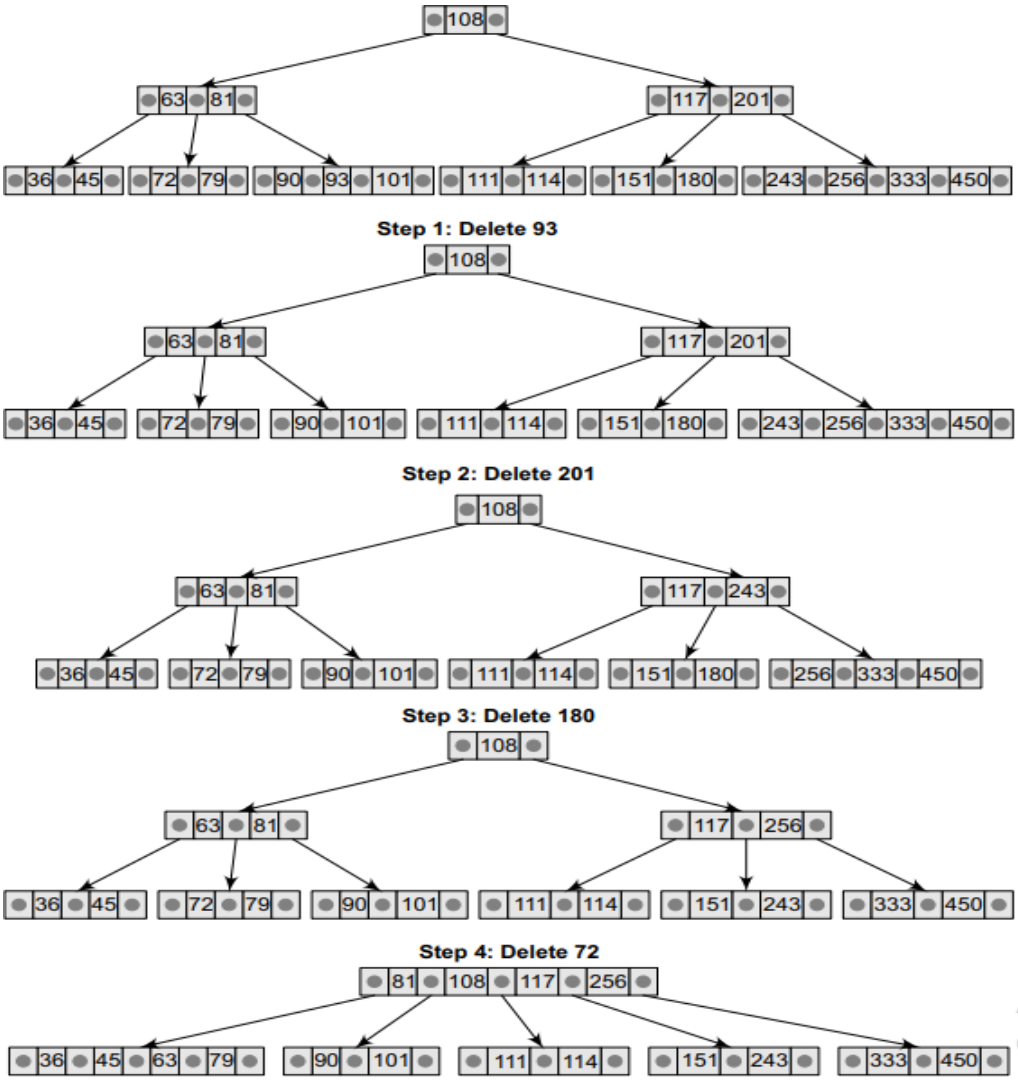
smallest key into its parent node and pull down the intervening element from the parent node to the leaf node where the key is deleted.

4. Else, if both left and right siblings contain only the minimum number of elements, then create a new leaf node by combining the two leaf nodes and the intervening element of the parent node (ensuring that the number of elements does not exceed the maximum number of elements a node can have, that is, m). If pulling the intervening element from the parent

node leaves it with less than the minimum number of keys in the node, then propagate the process upwards, thereby reducing the height of the B tree.

To delete an internal node, promote the successor or predecessor of the key to be deleted to occupy the position of the deleted key. This predecessor or successor will always be in the leaf node. So the processing will be done as if a value from the leaf node has been deleted.

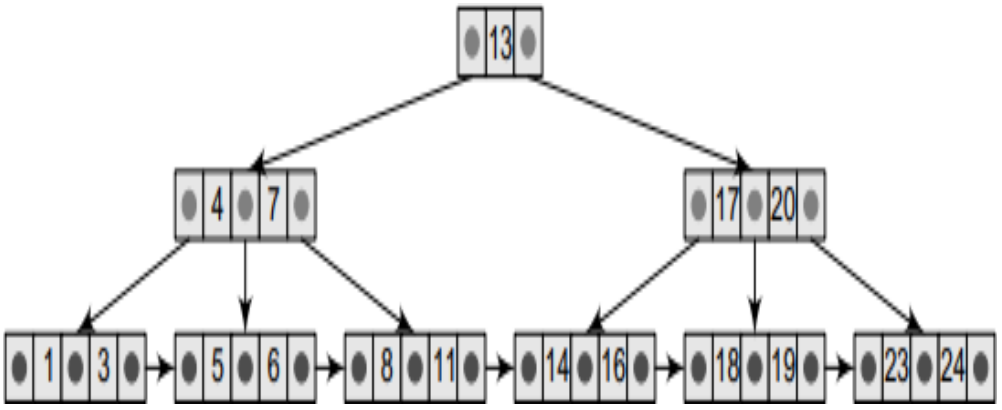
Ex.Consider the following B tree of order 5 and delete values 93, 201, 180, and 72 from it



B+ TREES:

- A B+ tree is a variant of a B tree which stores sorted data in a way that allows for efficient insertion, retrieval, and removal of records, each of which is identified by a key.
- While a B tree can store both keys and records in its interior nodes, a B+ tree, in contrast, stores all the records at the leaf level of the tree; only keys are stored in the interior nodes.
- The leaf nodes of a B+ tree are often linked to one another in a linked list. This has an added advantage of making the queries simpler and more efficient.
- Typically, B+ trees are used to store large amounts of data that cannot be stored in the main memory.
- With B+ trees, the secondary storage (magnetic disk) is used to store the leaf nodes of trees and the internal nodes of trees are stored in the main memory.
- B+ trees store data only in the leaf nodes.
- All other nodes (internal nodes) are called index nodes or i-nodes and store index values.
- This allows us to traverse the tree from the root down to the leaf node that stores the desired data item

Ex. A B+ tree of order 3.



Inserting a New Element in a B+ Tree:

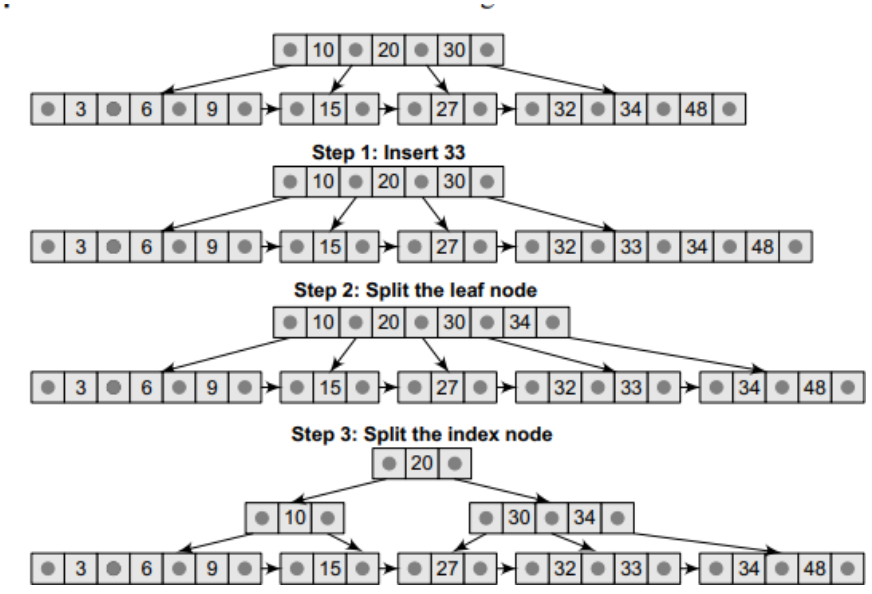
The steps to insert a new node in a B+ Tree

Step 1: Insert the new node as the leaf node.

Step 2: If the leaf node overflows, split the node and copy the middle element to next index node.

Step 3: If the index node overflows, split that node and move the middle element to next index page.

Consider the B+ tree of order 4 given and insert 33.



Deleting an Element from a B+ Tree:

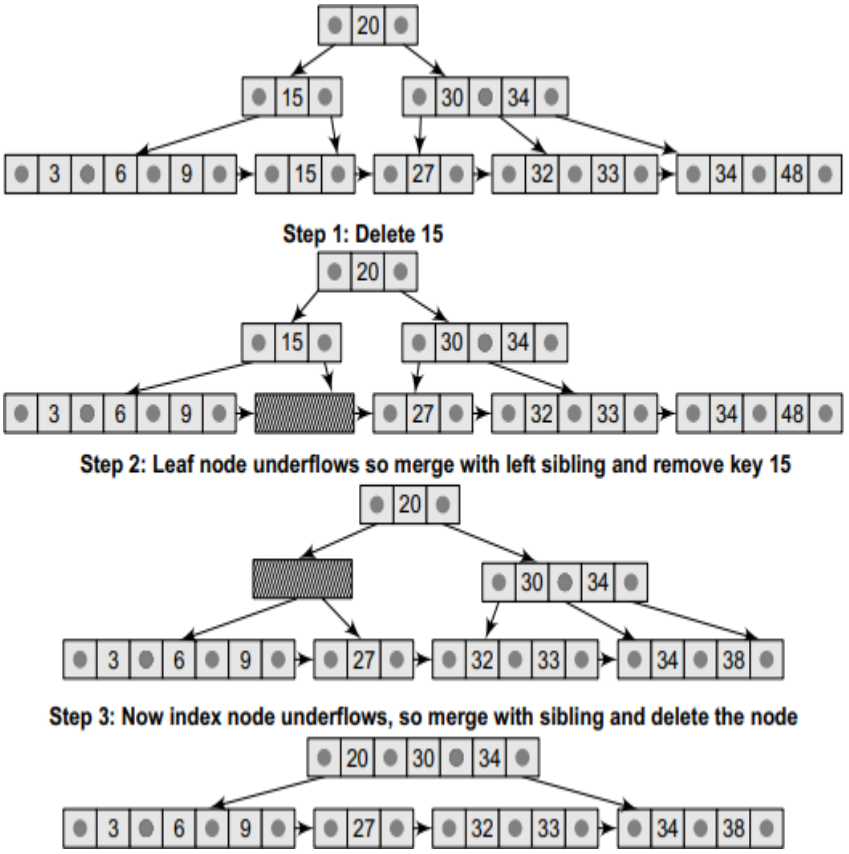
The steps to delete a node from a B+ tree

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

Step 2: If the leaf node underflows, merge that node with the sibling and delete the key in between them.

Step 3: If the index node underflows, merge that node with the sibling and move down the key in between them

Ex.Consider the B+ tree of order 4 given below and delete node 15 from it



Comparison between B trees and to B+ trees

B Tree	B+ Tree
1. Search keys are not repeated	1. Stores redundant search key
2. Data is stored in internal or leaf nodes	2. Data is stored only in leaf nodes
3. Searching takes more time as data may be found in a leaf or non-leaf node	3. Searching data is very easy as the data can be found in leaf nodes only
4. Deletion of non-leaf nodes is very complicated	4. Deletion is very simple because data will be in the leaf node
5. Leaf nodes cannot be stored using linked lists	5. Leaf node data are ordered using sequential linked lists
6. The structure and operations are complicated	6. The structure and operations are simple

GRAPH

A graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices. Figure 13.1 shows a graph with V(G) = {A, B, C, D and E} and E(G) = {(A, B), (B, C), (A, D), (B, D), (D, E), (C, E)}. Note that there are five vertices or nodes and six edges in the graph

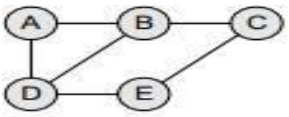


Figure 13.1 Undirected graph

Why are Graphs Useful?

Graphs are widely used to model any situation where entities or things are related to each other in pairs.

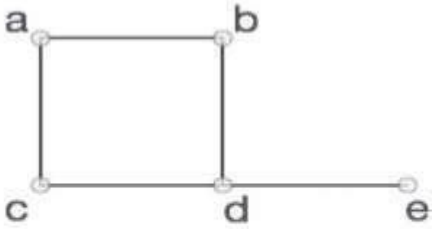
For example, the following information can be represented by graphs:

- Family trees in which the member nodes have an edge from parent to each of their children.
- Transportation networks in which nodes are airports, intersections, ports, etc. The edges can be airline flights, one-way roads, shipping routes, etc

Graph-definition:

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

V = {a, b, c, d, e}

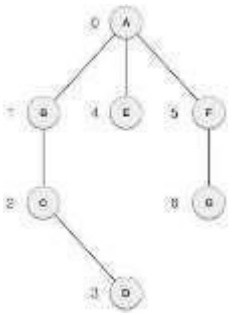
E = {ab, ac, bd, cd, de}

Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with

some important terms −

- **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 andsoon.
- **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two- dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column2 and so on, keeping other combinationsas0.
- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, andsoon.
- **Path** − Path represents a sequence of edges between the two vertices. In the following example,



ABCD represents a path from Ato D.

Basic Operations

Following are basic primary operations of a Graph −

- **Add Vertex** − Adds a vertex tothe graph.
- **Add Edge** − Adds an edge between the two vertices ofthe graph.
- **Display Vertex** − Displays a vertex ofthe graph.

Terminologies Used

LENGTH

The no of edges in a path is called as length of the path in a graph. For example, the length of the path (V1,V4) in a above graph is 2 because it contains two edges there are (V1,V2),(V2,V4).

DEGREE

The number of edges incident on a vertex in a graph is called as degree. It is classified in to two types.

- Indegree
- Outdegree

INDEGREE

The number of incoming edges to a vertex is called as indegree

Indegree of V1 =1 Indegree of V2 =1

Indegree of V3 =0 Indegree of V4 =2

OUTDEGREE

- The number of outgoing edges from a vertex is called as outdegree

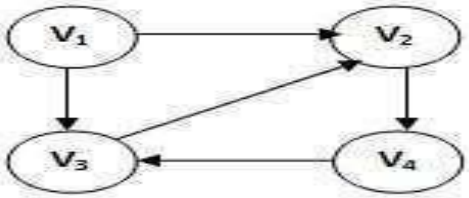
Outdegree of V1=1 Outdegree of V2=1

Outdegree of V3=2 Outdegree of V4=0

2. TYPES OF GRAPHS

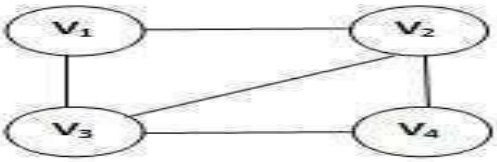
1. DIRECTED GRAPH

Directed graph is a graph in which edges are directed. Here each edge is unidirectional. In directed graph the edges (V1,V2) is not same as (V2,V1). It is also called as digraph.



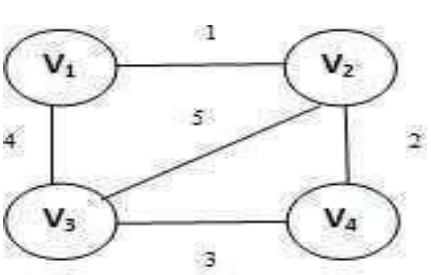
2. Undirected graph

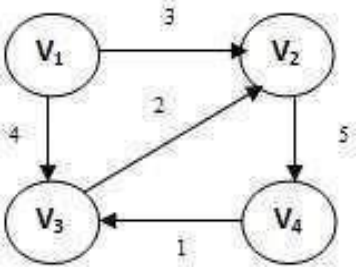
Undirected graph is a graph in which edges are undirected. Here each edge is Bidirectional. In undirected graph, (V1,V2) = (V2,V1)



3. WEIGHTED GRAPH

Weighted graph is a graph in which edges are assigned by some a weight or value. This value is considered as cost of traversing from one vertex to another vertex.Weighted graph can be either directed or undirected.

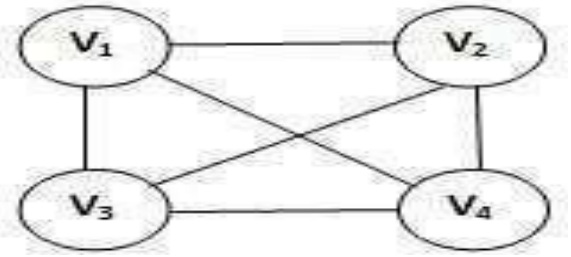




4. COMPLETE GRAPH

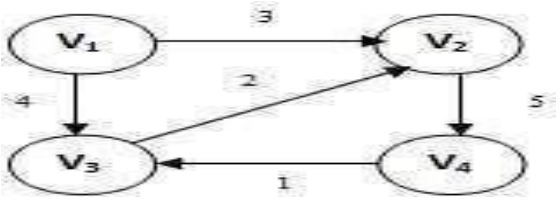
Complete graph is a graph in which there is an edge between each pair of vertices. Here there is a path

from each vertex to every other vertex. A complete graph with n vertices should have n(n-1)/2 edges.



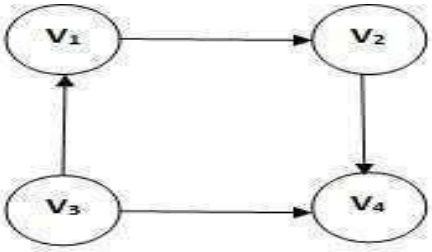
5. CYCLIC GRAPH

Cyclic graph is a graph which has cycles.Cycle is nothing but path, in which start and end at same vertex.



6. ACYCLIC GRAPH

Acyclic graph is a graph in which does not have cycles in it. It is also called as Directed Acyclic Graph(DAG).



3. Representation of Graphs

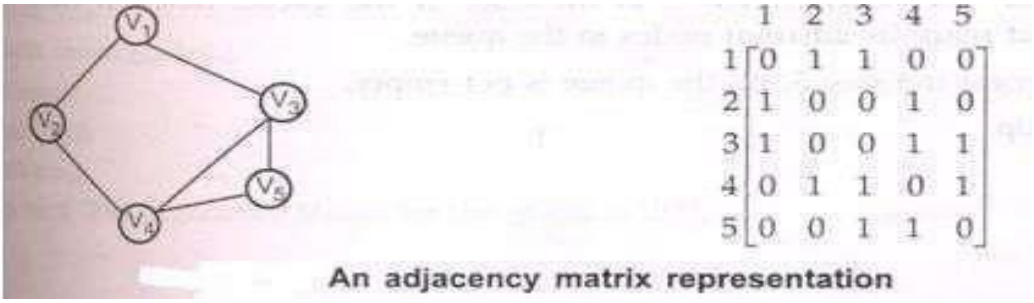
The two commonly used representations of Graphs are:

1. AdjacencyMatrix
2. AdjacencyList

1. AdjacencyMatrix

Consider a graph C of n vertices and the matrix M. If there is an edge present between vertices $V_i$and V_j, then M[i][j] = 1 else M[i][j] = 0. For an undirected graph, M[i][j] = M[i][j = 1.

Example:



An adjacency matrix representation

Creating a Graph using Adjacency Matrix

Creation of graph using adjacency matrix is a simple task. The adjacency matrix is nothing but a two dimensional array. The algorithm for creation of graph using adjacency matrix is given below:

1) Declare an array of M [size][size] which storesthegraph.
2) Enter the no. of nodes inthegraph.
3) Enter the edges of the graph by two vertices each, say V_i, V_j, which indicatestheedge.
4) If the graph is directed set M [i][j]= 1. If graph is undirected setM[i][j]=M[j][i]=1.
5) When all the edges for the desired graph is entered print thegraphM[i][j].

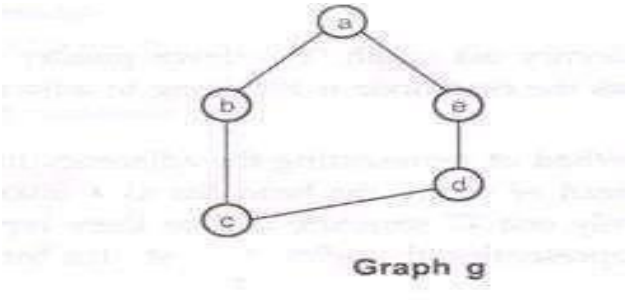
2. AdjacencyList

The type in which a graph is created with the linked list is called adjacency list. So all the advantages of linked list can be obtained in this type of graph. We need not have a prior knowledge of maximum number of nodes.

Construction of Adjacency List:

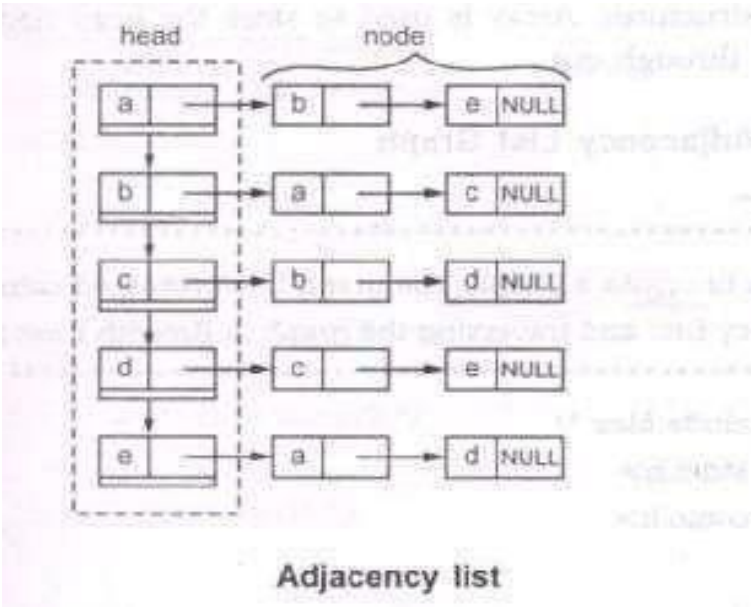
Since graph is a set of vertices and edges, we will maintain the two structures, for vertices and edges respectively.

Consider the graph given below. It has vertices a,b,c,d& e.



Graph g

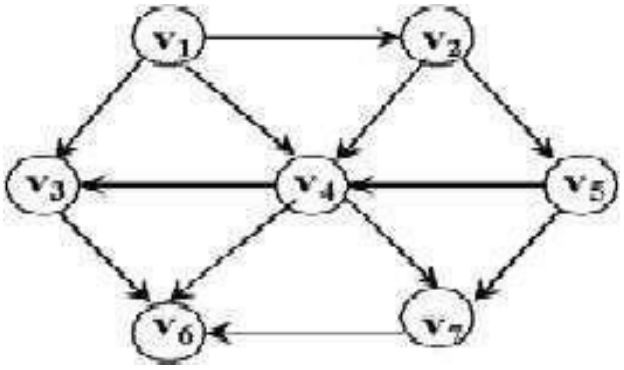
The linked list representation of the head nodes and the adjacent nodes are given below.



Adjacency list

4. Topological Sort

Topological Sort- An ordering of vertices in a directed acyclic graph, such that if there is a path from vi to vj, then vj appears after vi in the ordering.



An Acyclic Graph

Routine:

```
/* Simple topological sort */
Void Graph::topsort ()
{
  int Counter;
  Vertex V, W;
  for (Counter=0; Counter<NUM_VERTEX; Counter++)
  {
    V = FindNewVertexOfInDegreeZero ();
if (V == NOT_A_VERTEX)
throwCycleFound();
    V. topNum = Counter;
     for each W adjacent toV
       W.indegree--;
  }
}
```

- An improved algorithm
  - Keep all the unassigned vertices of indegree 0 inaqueue.
  - While queuenotempty
    - Remove a vertex in thequeue.
    - Decrement the indegree of all adjacent vertices.
    - If the indegreeof an adjacent vertex becomes 0, enqueue the vertex.
  - Running time is $O(|E|+|V|)$.

	Indegree Before Dequeue #						
Vertex	**1**	**2**	**3**	**4**	**5**	**6**	**7**
v_1	0	0	0	0	0	0	0
v_2	1	0	0	0	0	0	0
v_3	2	1	1	1	0	0	0
v_4	3	2	1	0	0	0	0
v_5	1	1	0	0	0	0	0
v_6	3	3	3	3	2	1	0
v_7	2	2	2	1	0	0	0
Enqueue	v_1	v_2	v_5	v_4	v_3, v_7		v_6
Dequeue	v_1	v_2	v_5	v_4	v_3	v_7	v_6

Result of applying topological sort to the graph

Pseudocode for Topological sort :

```
Void Graph::topsort ()
                    {
                    Queue q(NUM_VERTICES);
                      int Counter = 0;
                      Vertex V, W;
                        q.makeEmpty;
                      for each vertex V
                        if (V.Indegree== 0)
                    q.enqueue(V)
                    Enqueue (V, Q);
                        while (!q.isEmpty ( ))
                        {
                          V = q.dequeue ( );
                          V.topNum = ++Counter; /* Next No. */
                          for each W adjacent to V
                            if (--W.Indegree == 0)
                              q.enqueue (W);
                        }
                    if (Counter != NUM_VERTEX)
                              throw CycleFound();
                    }
```

4. GRAPH TRAVERSAL

- A traversal is a symmetric procedure for exploring a graph by examining all its vertices and edges.
- There are two types of graph traversals. They are,
  - Depth First search
  - Breadth First search

Depth First Traversal of A Graph

Depth First Traversal also called as Depth First Search (DFS)

In this method, we follow the path as deeply as we can go. When there is no adjacent vertex present we traverse back and search for unvisited vertex. We will maintain a visited array to mark all the visited vertices. In case of DFS the depth of a graph should be known.

Algorithm:

Step 1: Choose any node in the graph. Designate it as the search node and mark it as visited.

Step 2: Using the adjacency matrix of the graph, find a node adjacent to the search node that has not been visited yet. Designate this as the new search node and mark it as visited.

Step 3: Repeat Step 2 using the new search node. If no node satisfying Step 2 can be found, return to the previous search node and continue.

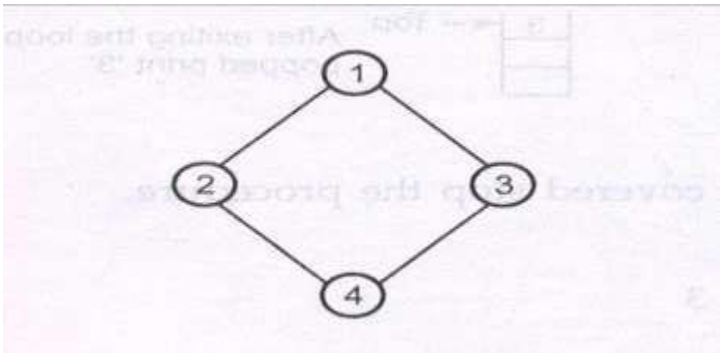
Step 4: When a return to the previous node in step 3 is impossible, the search from the originally chosen search node is complete.

Step 5: If the graph still contains unvisited nodes, choose any node that has not been visited and repeat Step 1 through Step 4.

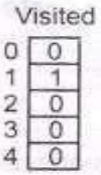
Routine for DFS:

```
Void DFS(Vertex V)
{
        visited[V]=True;
        for each W adjacent to V
        if(!visisted[W])
        DFS(W);
}
```

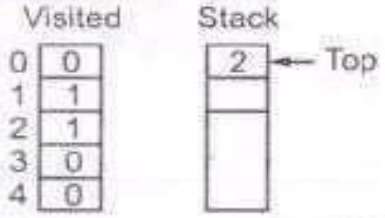
Example:



In DFS the basic data structure for storing the adjacent vertices is stack. Step 1: Start with vertex 1, print it so '1' gets printed. Mark I as visited.

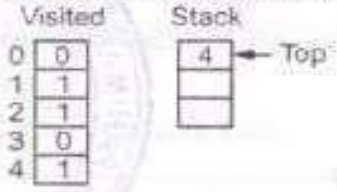


Step 2 : Find adjacent vertex to 1, say i.e. 2 if it is not visited, call DFS(2 will get inserted in the stack, mark is as visited.



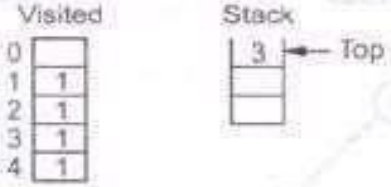
After exiting the loop 2 will be popped print '2'

Step 3 : Find adjacent to '2' i.e. vertex 4 if it is not visited call DFS(4) i.e., get pushed on to the stack mark it as visited.



After exiting the loop 4 will be popped print '4'

Step 4 : Find adjacent to '4' i.e. vertex 3 if it is not visited call DFS(3) i.e. 3 w pushed onto the stack mark it visited.



After exiting the loop 3 will be popped print '3'

Since all the nodes are covered stop the procedure.

So output of DFS is

1 2 4 3

Breadth First Traversal also called as Breadth First Search (BFS)

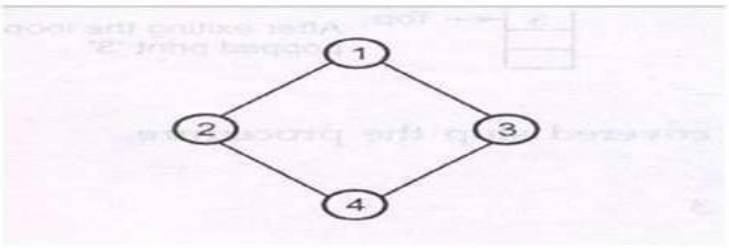
In the BFS, a vertex V is taken visited first. Then the adjacent vertices of V are visited. The same procedure is performed for all the vertices in the graph.

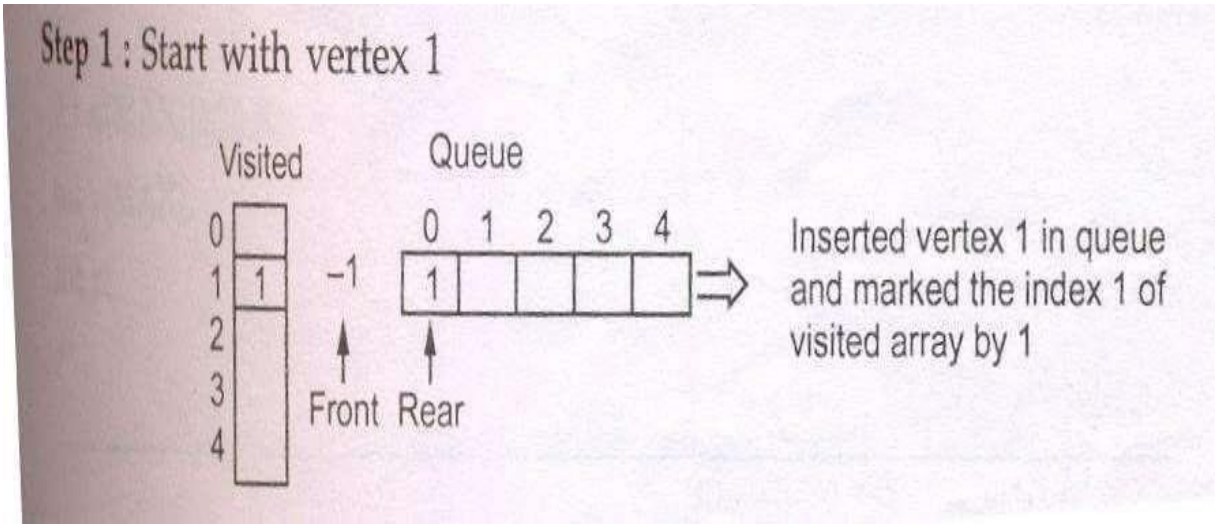
Algorithm:

1. Create a graph.
2. Read the vertex from which you want to traverse the graph say V_i.
3. Initialize the visited array to 1 at the index of V_i
4. Insert the visited vertex V_i in the queue
5. Visit the vertex which is at the front of the queue. Delete it from the queue and place its adjacent nodes in the queue.
6. Repeat the step 5, till the queue is not empty
7. Stop.

Explanation of logic of BFS

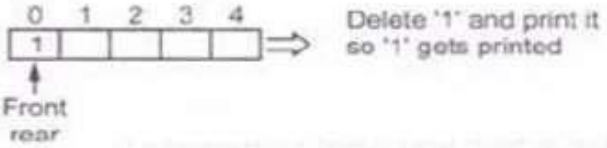
In BFS the queue is maintained for storing the adjacent nodes and an array "visited" is maintained for keeping the track of visited nodes. i.e. once a particular t is visited it should not be revisited again.



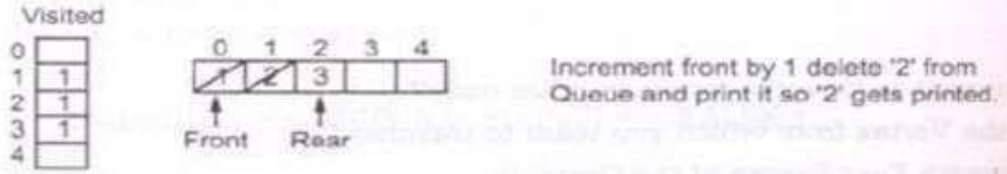


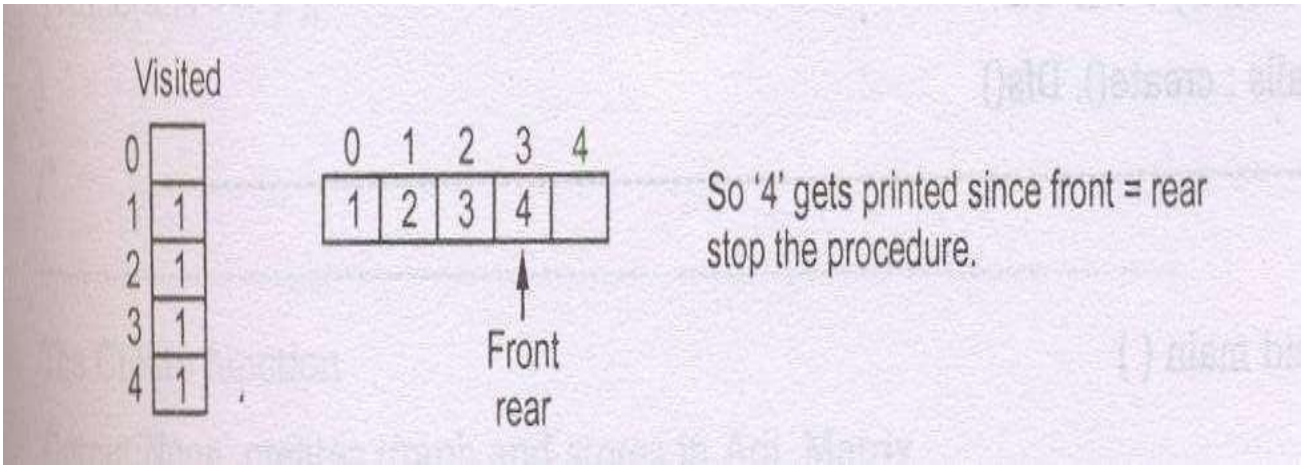
Increment front, delete the node from Queue and print it.

Step 2 :



Step 3 : Find adjacent vertices of vertex 1 and mark them as visited, insert this in Queue.





So output will be - BFS for above graph as

1 2 3 4

Routine:

Algorithm bfs(*G*)

1. Initialise Boolean array *visited*, setting all entries to FALSE.
2. Initialise *Queue Q*
3. **for all** $v \in V$ **do**
4. **if** *visited*[*v*] = FALSE **then**
5. bfsFromVertex(*G*, *v*)

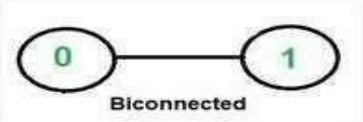
Difference between DFS and BFS

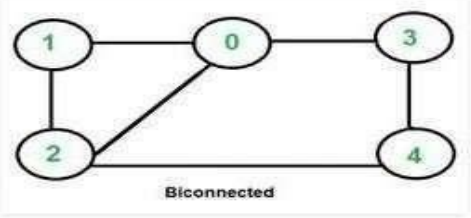
DFS	BFS
DFS visit nodes of graph depth wise. It visits nodes until reach a leaf or a node which doesn't have non-visited nodes.	BFS visit nodes **level by level** in Graph.
Usually implemented using a stack data structure.	Usually implemented using a queue data structure.
Generally requires less memory than BFS.	Generally requires more memory than DFS.
Not Optimal for finding the shortest distance.	optimal for finding the shortest distance.
DFS is better when target is far from source.	BFS is better when target is closer to source.

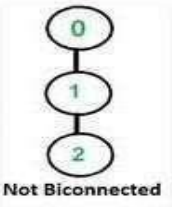
6.Biconnected graph

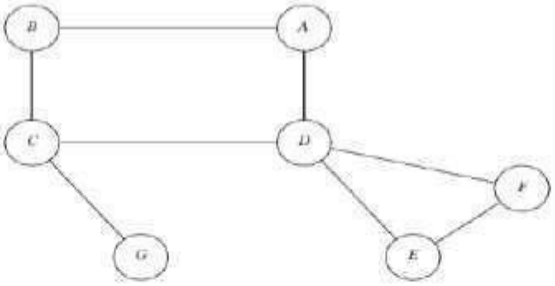
An undirected graph is called Biconnected if there are two vertex-disjoint paths between any two vertices. In a Biconnected Graph, there is a simple cycle through any two vertices. By convention, two nodes connected by an edge form a biconnected graph, but this does not verify the above properties. For a graph with more than two vertices, the above properties must be there for it to be Biconnected.

Following are some examples.









Step 1: Find DFN

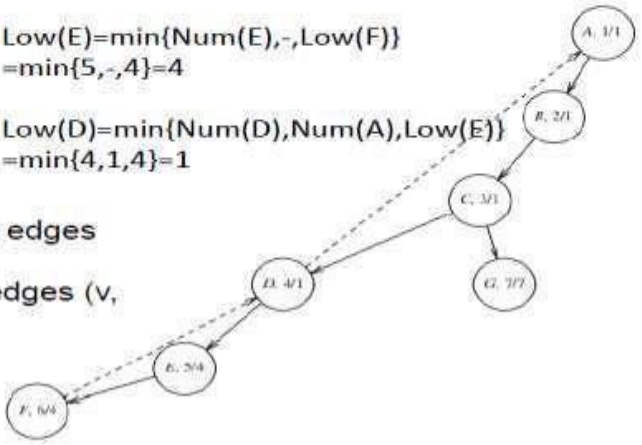
First, starting at any vertex, we perform a depth-first search and number the nodes as they are visited.

For each vertex, v, call this preorder number Num(v)

Low(F)=min{Num(F),Num(D),-)
=min{6,4,-}=4

Low(E)=min{Num(E),-,Low(F)}
=min{5,-,4}=4

Low(D)=min{Num(D),Num(A),Low(E)}
=min{4,1,4}=1



Step 2: Do Post order Traversal and Find low value

Low(v) is the minimum of

1. Num(v)
2. the lowest Num(w) among all back edges (v, w)
3. the lowest Low(w) among all tree edges (v, w)

Step 3: Rules to find Articulation Point

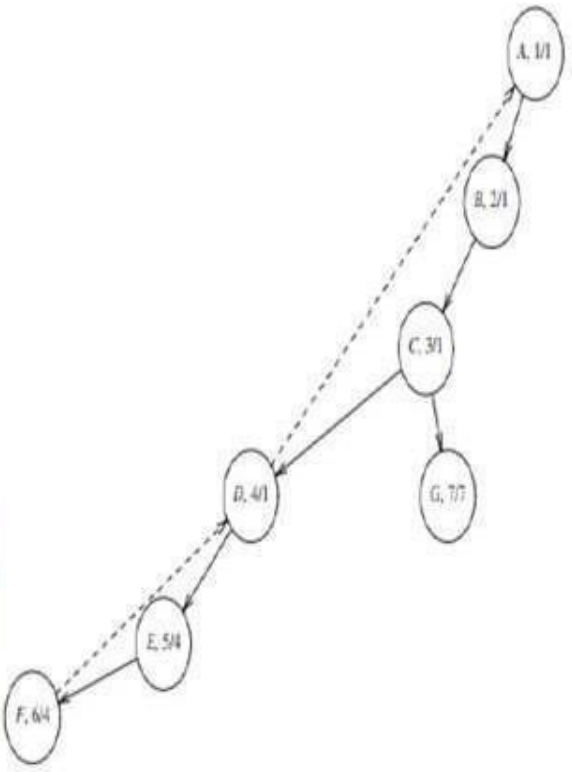
The root is an articulation point if and only if it has more than one child

Any other vertex v is an articulation point if and only if v has some child w such that Low(w) ≥ Num(v)

C and D are articulation points

C has a child G and Low(G) ≥ Num(C).

D has a child E, and Low(E) ≥ Num(D)

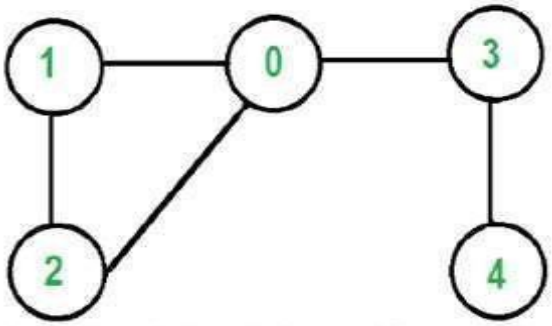


VERTEX	A	B	C	D	E	F	G
DFN	1	2	3	4	5	6	7
LOW	1	1	1	1	4	4	7

6. EULER CIRCUIT

Eulerian path and circuit for undirected graph

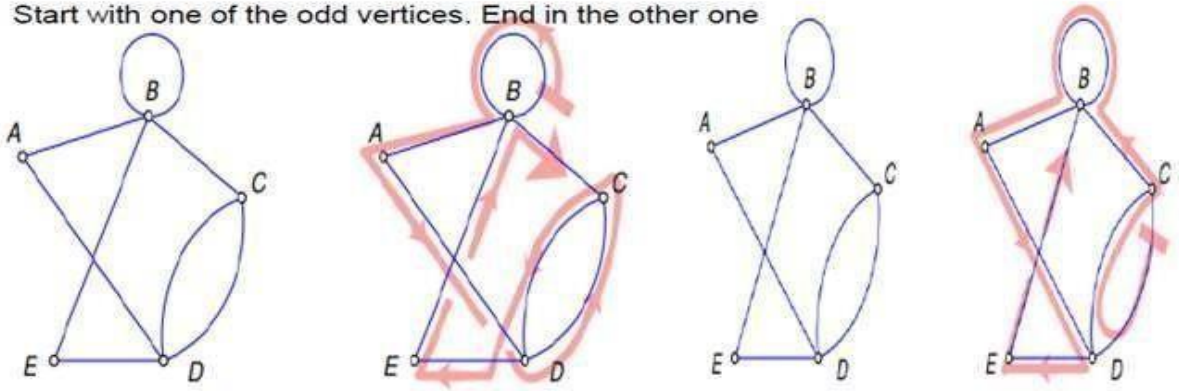
- Eulerian Path is a path in graph that visits every edge exactly once.
- Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.
- An Euler path starts and ends at different vertices.
- An Euler circuit starts and ends at the same vertex
- A graph is called Eulerian if it has an Eulerian Cycle and called Semi-Eulerian if it has an Eulerian Path



The graph has Eulerian Paths, for example "4 3 0 1 2 0", but no Eulerian Cycle. Note that there are two vertices with odd degree (4 and 0)

Euler Path to exist in a graph, exactly 2 vertices must have odd degree

Start with one of the odd vertices. End in the other one



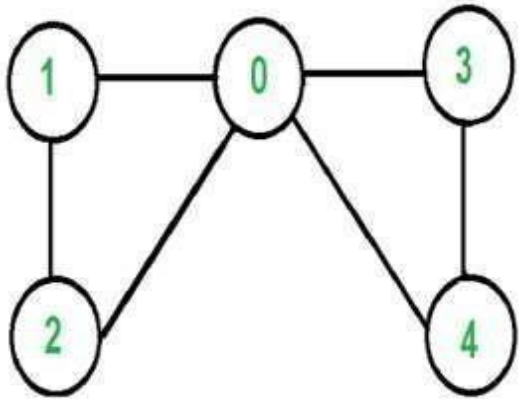
An Euler path: BBADCDEBC

Another Euler path: CDCBBADEB

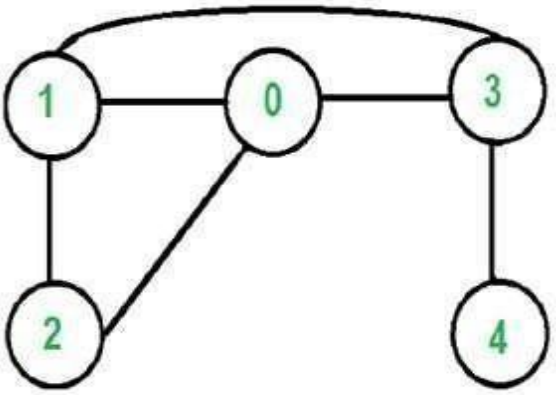
Condition for Euler Path and Euler Circuit

# odd vertices	Euler path?	Euler circuit?
0	No	**Yes***
2	**Yes***	No
4, 6, 8, . . .	No	No
1, 3, 5,	No such graphs exist	

* *Provided the graph is connected.*

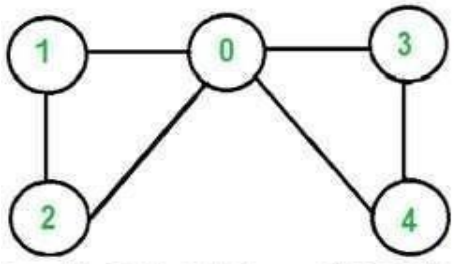


The graph has Eulerian Cycles, for example "2 1 0 3 4 0 2"
Note that all vertices have even degree



The graph is not Eulerian. Note that there are four vertices with odd degree (0, 1, 3 and 4)

DFS to find Euler circuit



The graph has Eulerian Cycles, for example "2 1 0 3 4 0 2"
Note that all vertices have even degree

Solution 1

2 , 1, 0, 2

2, 1,0, 3,4,0,2

Solution 2

1,0,2,1

1,0,3,4,0,2,1

- Start with any vertex s.
- First, using DFS find any circuit starting and ending in s.
- Mark all edges on the circuit as visited
- While there are still edges in the graph that are not marked visited:
  - Find the first vertex v on the circuit that has unvisited edges.
  - Find a circuit starting in v and splice this path into the first circuit

APPLICATION OF GRAPH:

Graphs are nothing but connected nodes(vertex). So any network related, routing, finding relation, path etc related real life applications use graphs.

- Connecting with friends on social media, where each user is a vertex, and when users connectthey create an edge.
- Using GPS/Google Maps/Yahoo Maps, to find a route based onshortest route.
- Google, to search for webpages, where pages on the internet are linked to each otherby hyperlinks; each page is a vertex and the link between two pages isanedge.
  - On ecommerce websites relationship graphs are used to show recommendations.

UNIT V SEARCHING, SORTING AND HASHING TECHNIQUES

Searching – Linear Search – Binary Search. Sorting – Bubble sort – Selection sort – Insertion sort – Shell sort –. Merge Sort – Hashing – Hash Functions – Separate Chaining – Open Addressing – Rehashing – Extendible Hashing.

SEARCHING:

- Searching is a process of locating a particular element present in a given set of elements. The element may be a record, a table, or a file.
- A search algorithm is an algorithm that accepts an argument 'a' and tries to find an element whose value is 'a'.
- It is possible that the search for a particular element in a set is unsuccessful if that element does not exist.
- There are a number of techniques available for searching and the most popular among them include:

1. **LINEAR SEARCH**
2. **BINARY SEARCH**

1. LINEAR SEARCH

In Linear Search the list is searched sequentially and the position is returned if the key element to be searched is available in the list, otherwise -1 is returned.

The search in Linear Search starts at the beginning of an array and move to the end, testing for a match at each item.

All the elements preceding the search element are traversed before the search element is traversed. i.e. if the element to be searched is in position 10, all elements from 1 to 9 are checked before 10.

Example:

Assume the element 45 is searched from a sequence of sorted elements 12, 18, 25, 36, 45, 48, 50. The Linear search starts from the first element 12, since the value to be searched is not 12 (value 45), the next element 18 is compared and is also not 45, by this way all the elements before 45 are compared and when the index is 5, the element 45 is compared with the search value and is equal, hence the element is found and the element position is 5.

List	i	Result of comparison
12 18 25 36 45 48 50	1	12 not equal to 45 (false)
12 18 25 36 45 48 50	2	18 not equal to 45 (false)
12 18 25 36 45 48 50	3	25 not equal to 45 (false)
12 18 25 36 45 48 50	4	36 not equal to 45 (false)
12 18 25 36 45 48 50	5	45 equal to 45 (true)

Program to implement Linear Search

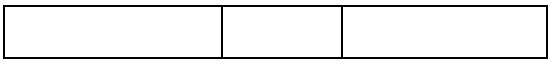
```
#include<stdio.h>
int
main()
{
int a[10],i,n,x,c=0;
printf("Enter the size of an array: ");
scanf("%d",&n);
printf("Enter the elements of the
array: ");
for(i=0;i<=n-1;i++)
{
scanf("%d",&a[i]);
}
printf("Enter the number to be search:
");
```

Sample output: Enter the size of an array: 5
Enter the elements of the array: 4 6 8 0 3
Enter the number to be search: 0

```
scanf("%d",&x);
for(i=0;i<=n-1;i++)
{
        if(a[i]==x)
        {
                c=1;
                break
        ;
        }
}
if(c==0)
        printf("The number is not in the
list");
 else
      printf("The number is found");
return 0;
 }
```

2. Binary Search.

Linear Search is slow and to some extent inefficient. To overcome this, a faster search method named Binary Search has been formulated. IN Binary search, the original list has to be sorted, and then the search for an element takes place using divide and conquers technique. The list is divided into two halves separated by the middle element as shown below:



1 N/2 N

Binary search follows the below steps:

1. The middle element is tested for the required key value searched.
2. If key value<middle search the left half of the list, else search the right half of the list.
3. Repeat step 1 and 2 on the selected half until the key value is found otherwise report failure. Example:

Consider the same example of searching 45 in the list

List	low	high	mid	Result
12 18 25 ***36*** *45 48 50*	0	6	3	45 > 36: right half
12 18 25 *36 45* ***48*** *50*	4	6	5	45 < 48: left half
12 18 25 *36* ***45*** 48 50	4	4	4	45 = 45: found

Program to implement Binary Search

```
#include<stdio.h>
int main()
{
int a[10],i,n,x,c=0,low,high,mid;
printf("Enter the size of an array: ");
scanf("%d",&n);
printf("Enter the elements in ascending order: ");
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
printf("Enter the number to be search: ");
scanf("%d",&x);
low=0,high=n-1;
while(low<=high)
{
mid=(low+high)/2;
if(x==a[mid])
{
c=1;
break;
}
else if(x<a[mid])
{
high=mid-1;
}
else
low=mid+1;
}
if(c==0)
printf("The number is not found.");
else
```

```
printf("The number is found.");

}
```

Sample output:

```
Enter the size of an array: 5
Enter the elements in ascending order: 4
7 8 11 21 Enter the number to be search:
11
The number is found.
```

Searching Method	Advantage	Disadvantage
Linear Search	• Simple • Elements need not be in order	• Less efficient
Binary Search	• Efficient and Fast	• Elements need to be in order • Not as simple as Linear Search.

SORTING:

Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, lexicographical, or any user-defined order. Sorting is a process through which the data is arranged in ascending or descending order.

Sorting can be classified in two types;

Internal Sorts:- This method uses only the primary memory during sorting process.

- All data items are held in main memory and no secondary memory is required this sorting process.
- If all the data that is to be sorted can be accommodated at a time in memory is called internal sorting.
- There is a limitation for internal sorts; they can only process relatively small lists due to memory constraints.
- There are 3 types of internal sorts.

(i) SELECTION SORT :- Ex:- Selection sort algorithm, Heap Sort algorithm

(ii) INSERTION SORT :- Ex:- Insertion sort algorithm, Shell Sort algorithm

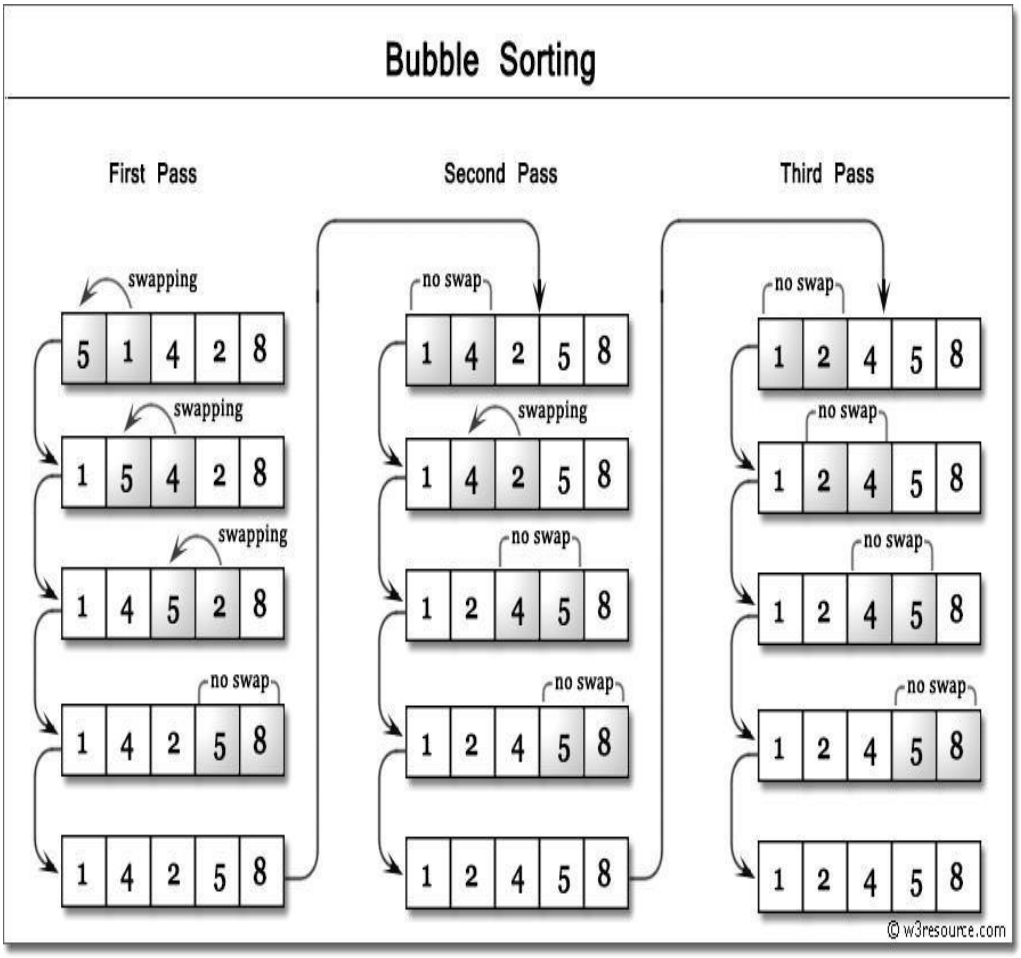
(iii) EXCHANGE SORT :- Ex:- Bubble Sort Algorithm, Quick sort algorithm

External Sorts:- Sorting large amount of data requires external or secondary memory. This process uses external memory such as HDD, to store the data which is not fit into the main memory. So, primary memory holds the currently being sorted data only. All external sorts are based on process of merging. Different parts of data are sorted separately and merged together.

Ex:- Merge Sort

1. BUBBLE SORTING OR OR SINKING SORT:

- It compares each pair of adjacent items and swaps them if they are in the wrong order.
- The passes through the list is repeated until no swaps are needed, which indicates that the list is sorted.



#include<stdio.h>

```
void main()
{
int i,n,t,j,a[10];
printf("\n Enter upper limit: ");
 scanf("%d", &n);
printf("\n Enter elements,…");
for(i=0;i<n;i++)
scanf("%d", &a[i]);
 for(i=0;i<n-1;i++)
{
for(j=i;j<=n-1;j++)
{
if(a[i]>a[j])
{
        t=a[i];
        a[i]=a[j];
        a[j]=t;
}
}
}
printf("\n numbers in ascending order\n");
for(i=0;i<n;i++)
printf("%d\n",a[i]);
}
```

2. INSERTION SORTING :

The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list.

- ✓ The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted.
- ✓ The approach is the same approach that you use for sorting a set of cards in yourhand.

Basic Idea:

Find the location for an element and move all others up, and insert the element. The process involved in insertion sort is as follows:

1. The left most value can be said to be sorted relative toitself.
2. Check to see if the second value is smaller than the first one. If it is, swap these two values. The first two values are now relatively sorted.
3. Next, we need to insert the third value in to the relatively sorted portion so that after insertion, the portion will still be relatively sorted.
4. Remove the third value first. Slide the second value to make room for insertion. Insert the value in the appropriate position.
5. Now the first three are relatively sorted.

6. Do the same for the remaining items in the list.

Original	34	8	64	51	32	21	Positions Moved
i=1	8	34	64	51	32	21	1
i=2	8	34	64	51	32	21	0
i=3	8	34	51	64	32	21	1
i=4	8	32	34	51	64	21	3
i=5	8	21	32	34	51	64	4

Program to sort array using insertion sort:

```
#include<stdio.h>
 void main()
{
int a[10],i,j,t,n;
printf("Enter the size of the array:\n");
        scanf("%d",&n);
printf("Enter %d elements of the array:\n",n);
for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
for(j=1;j<n;j++)
{
t=a[j];
i=j-1;
while(t<a[i]&&i>=0)
{
        a[i+1]=a[i];
        i=i-1;
}
a[i+1]=t;

}
printf("Sorted array is:\n"); for(i=0;i<n;i++)
{
printf("%d ",a[i]);
}
}
```

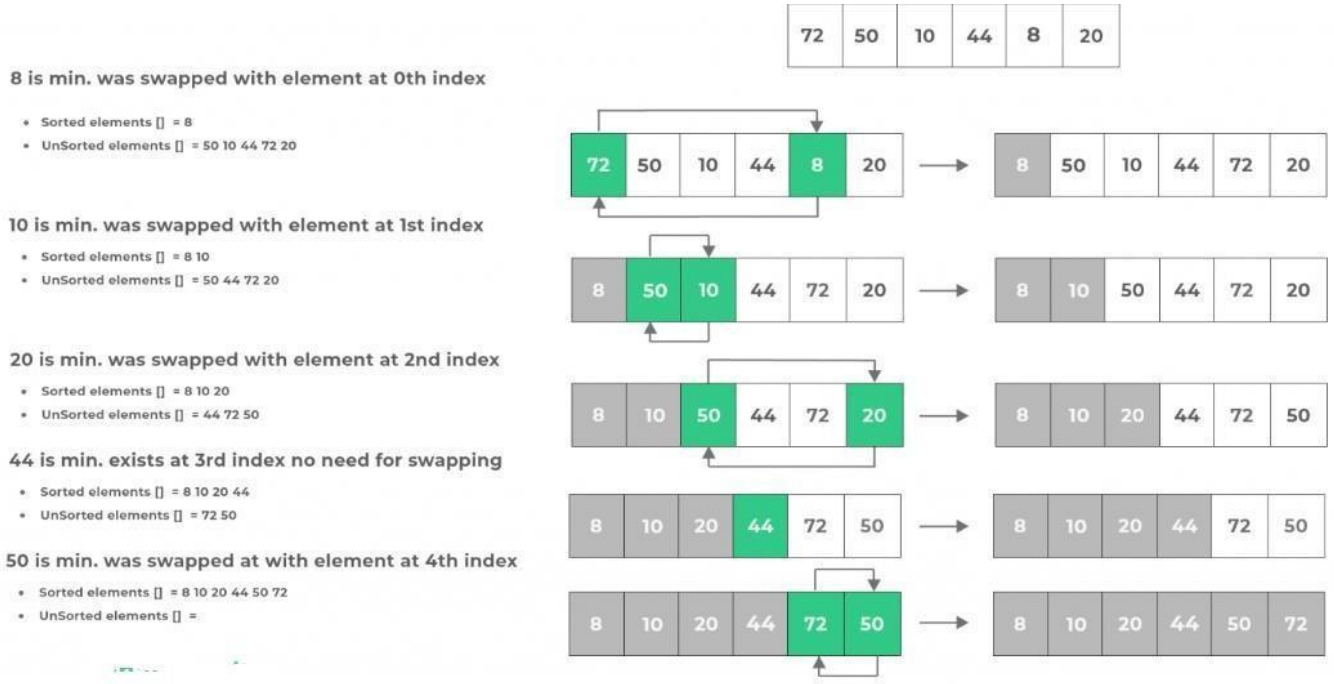
3. SELECTION SORTING

- The list is divided into two sublists, sorted and unsorted, which are divided by an imaginary wall.
- We find the smallest element from the unsorted sublist and swap it with the element at the beginning of the unsorted data.

- After each selection and swapping, the imaginary wall between the two sublists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed a sort pass.
- A list of n elements requires n-1 passes to completely rearrange the data

Original	64	25	12	22	11	19	Sorted List
i=1	11	25	12	22	64	19	{11}
i=2	11	12	25	22	64	19	{11,12}
i=3	11	12	19	22	64	25	{11,12,19}
i=4	11	12	19	22	64	25	{11,12,19,22}
i=5	11	12	19	22	25	64	{11,12,19,22,25,64}

Ex.2.



Program to sort array elements using selection sort method:

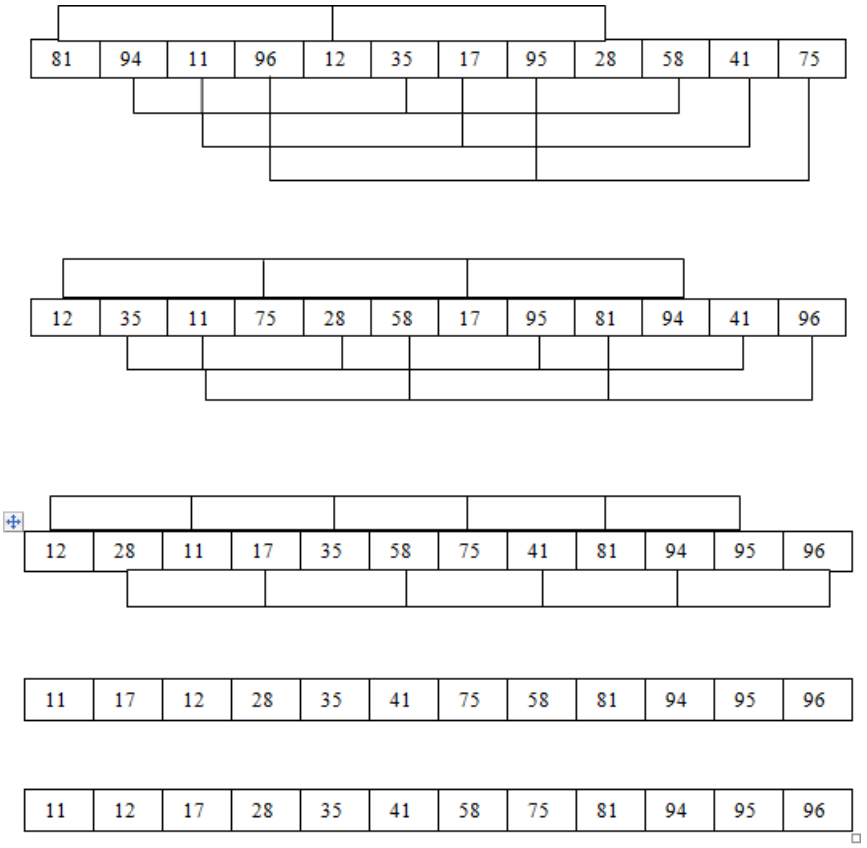
```
#include<stdio.h>
void main()
{
int i,a[10],j,min,t,pos,n;
printf("Enter the size of the array:");
scanf("%d",&n);
printf("Enter %d elements:",n);
 for(i=0;i<n;i++)
{
scanf("%d",&a[i]);
}
for(j=0;j<n-1;j++)
```

```
{
min=a[j]; pos=j;
for(i=j+1;i<n;i++)
{
if(min>a[i])
{
min=a[i];
pos=i;
}
}
t=a[pos];
a[pos]=a[j];
a[j]=t;
}
printf("Sorted Array is:");
 for(i=0;i<n;i++)
{
printf("%d ",a[i]);
}
}
```

4. SHELL SORT:

- It is also referred as diminishing increment sort. Shell sort uses a sequence h1,h2,……ht. called increment sequence.
- Steps to be followed,
    - Increment sequence used to determine how far apart elements to be sorted are: h1,h2…..ht.
    - At first elements at distance ht are sorted, them elements at distance ht-1 are sorted etc. until finally the array is sorted using insertion sort.
    - An array is said to be bk-sorted. if all elements spaced a distance hk apart are sorted relative to each other.

Original: 81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75.



Sorted data is, 11, 12, 17, 28, 35, 41, 58, 75, 81, 94, 95, 96.

C Code for Shel Sort:

```
void Shell_sort(int a[], int N)
{
        int j,p,gap,temp;
        for(gap=N/2;gap>0;gap=gap/2)
        {
                for(p=gap;p<N;p++)
                {
                        temp=a[p];
                        for(j=p;j>=gap&&temp<a[j-gap];j=j-gap)
                                a[j]=a[j-gap];
                        a[j]=temp;
                }
        }
}
```

5. MERGE SORT

Like quick sort, merge sort uses divide and conquer strategy and its time complexity is O(nlogn). This method uses following two concepts:

- If a list is empty or it contains only one element, then the list is already sorted. A list that contains onlyone element is also called singleton.

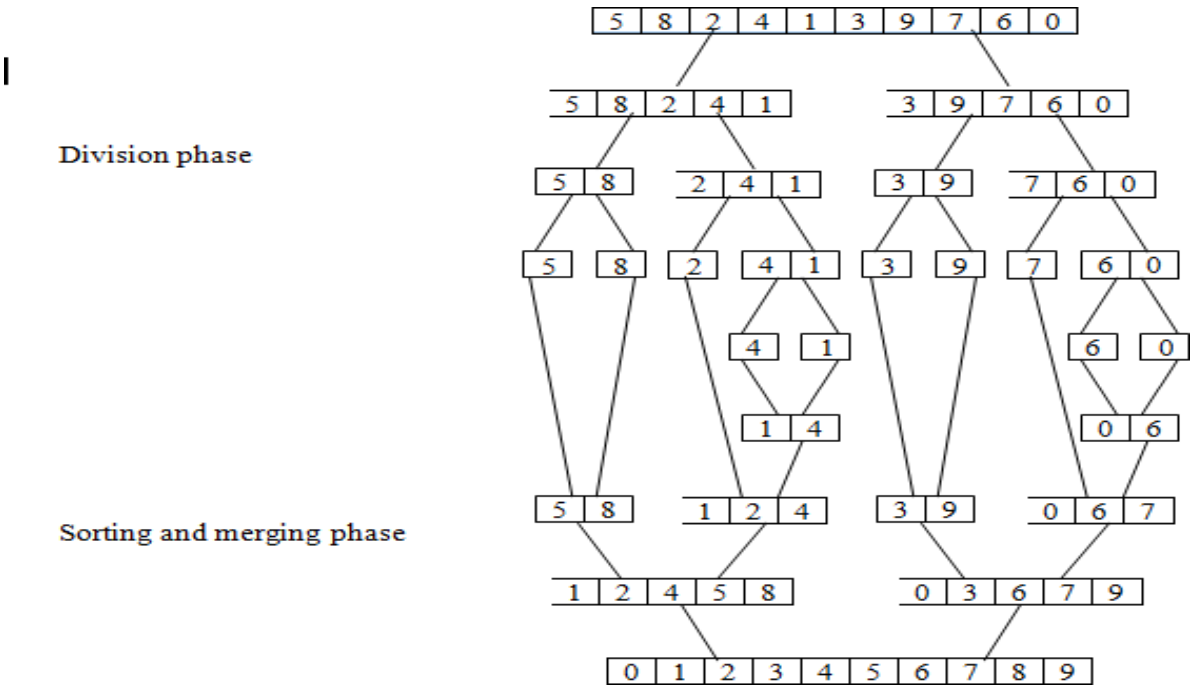
- It uses the old proven technique of '**divide and conquer'** to recursively divide the list into sub-lists until it is left with either empty or singleton lists.

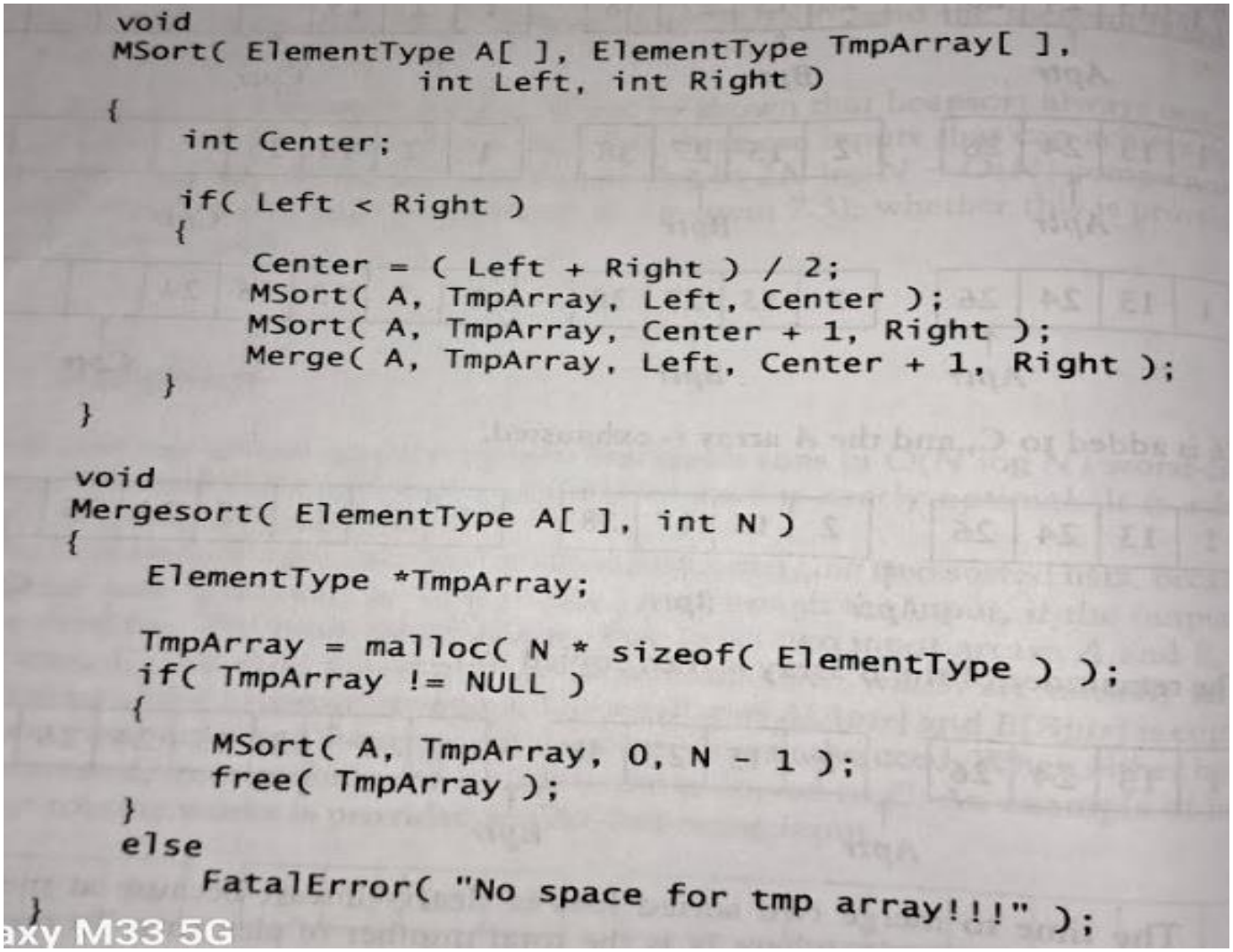
Two sub-lists can be safely joined when every element in the first sub-list is smaller than every element in the second sub-list. **Since 'join' is a faster operation as compared to a 'merge' operation, this sort is rightly named as a 'quick sort**'.

Algorithm:

7. Divide the array in to two halves.
8. Recursively sort the first n/2 items.
9. Recursively sort the last n/2 items.
10. Merge sorted items (using an auxiliary array). Example: Sort the following list using merge sortalgorithm.

5	8	2	4	1	3	9	7	6	0





```
void
MSort( ElementType A[ ], ElementType TmpArray[ ],
                  int Left, int Right )
{
    int Center;

    if( Left < Right )
    {
        Center = ( Left + Right ) / 2;
        MSort( A, TmpArray, Left, Center );
        MSort( A, TmpArray, Center + 1, Right );
        Merge( A, TmpArray, Left, Center + 1, Right );
    }
}

void
Mergesort( ElementType A[ ], int N )
{
    ElementType *TmpArray;

    TmpArray = malloc( N * sizeof( ElementType ) );
    if( TmpArray != NULL )
    {
        MSort( A, TmpArray, 0, N - 1 );
        free( TmpArray );
    }
    else
        FatalError( "No space for tmp array!!!" );
}
```

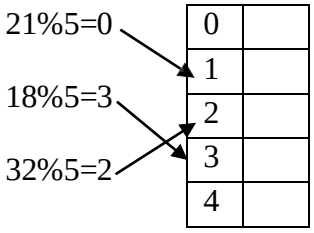

C Code for Merge Sort:

```
void Merge_Sort(int s1,int s2)
{
        int t,k=0;i=0,j=0;
        while(j<s2||i<s1)
        {
                if(v1[i]<v2[j])
                {
                        v3[k]=v1[i];
                        k++;
                        i++;
                }
                else
                {
                        v3[k]=v2[j];
                        k++;
                        j++;
                }
        }
}
```

HASHING

Hashing is the process of mapping large amount of data item to a smaller table with the help of a **hashing function**. Modulo operator is used to get the key value from the actual data/information.

Hash table:
The hash table data structure is merely an array of some fixed size, containing the keys. A key is a string with an associated value.
Each key is mapped into some number in the range 0 to tablesize-1 and placed in the appropriate cell. In the following example, tablesize is 5 ie., 0 to 4.



Hash function:
A hash function is a key to address transformation which acts upon a given key to compute the relative position of the key in an array.
The choice of hash function should be simple and it must distribute the data evenly.

```
Index Hash(int key,int tablesize)
{
      return key%tablesize;
}
```

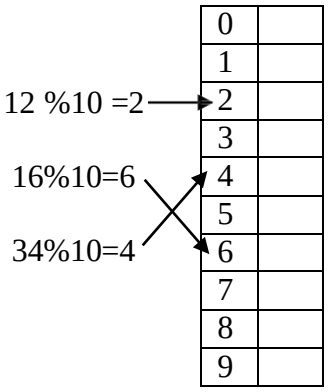
Importance of hashing:

- Maps key with the corresponding value using hash function.
- Hash tables support the efficient addition of new entries and the time spent on searching for the required data is independent of the number of items stored.
- A *hash* function is any function that can be used to map data of arbitrary size to data of fixed size.
- A perfect hash function has no blanks and no collisions.

1. Division method: The hash function depends upon the remainder of division.

H(key) = record % table size

For ex, Insert 12,16,34.



2. Mid square : In the mid square method , the key is squared and the middle or mid part of the result is used as the index.

Consider that if we want to place a record 3111 then for the hash table size 1000

3111^2 = 96**783**21

H(3111) = 783 (the middle 3 digits)

3. Digital Folding : The Key is divided into separat$_8$e$_6$part and using some simple operation these parts are combined to produce the hash key.

For example,consider a record 12365412

H(key) = 123 + 654 +12

= 789

The record will be placed at location 789 in the hash table.

COLLISION HANDLING TECHNIQUES OR COLLISION RESOLUTION TECHNIQUES:

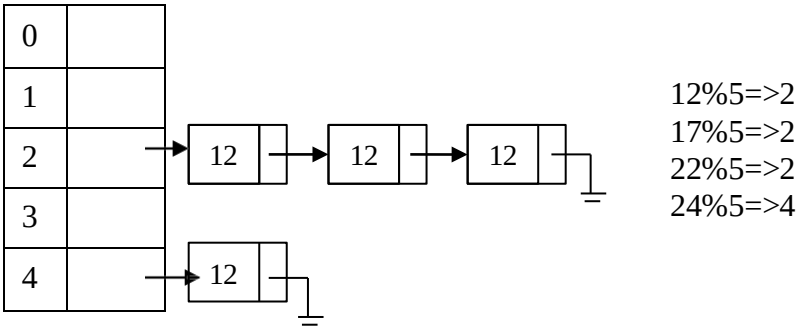
Collision: When an element is inserted, it hashes to the same value as an already inserted element, and then it produces collision.

- Separate chaining
- Open addressing
- Rehashing
- Extendible hashing

1. Separate Chaining:

Separate chaining is a collision resolution technique to keep the list of all elements that *hash to the same value*. This is called separate chaining because each hash table element is a separate chain (linked list). Each linked list contains all the elements whose keys hash to the same index.

- More number of elements can be inserted as it uses linked lists. For ex, insert 12,17,22,24.



Advantages of separate chaining:

1. *Simple* to implement.
2. *Hash table never fills up.*
3. *Less sensitive to the hash function* or load factors.
4. It is mostly used when it is *unknown how many and how frequently keys* may be inserted or deleted.

Disadvantages of separate chaining.

1. *Cache performance of chaining is not good.*
2. *Wastage of Space.*

3. If the chain becomes long, then *search time can become O(n)* in worst case.

4. *Uses extra space* for links.

Type declaration for separate chaining

```
struct ListNode
        {
        ElementType Element;
                Position        Next;
        };

typedef Position List;

struct HashTbl
        {
        int TableSize;
                List *TheLists;
        };
```

Find routine for separate chaining

```
  Position Find (ElementType Key, HashTable H)
  {
        Position P;
        List L;
        L = H->List [Hash (Key, H->TableSize)];
        P = L->Next;
        while (P != NULL && P->Element != Key)
                P=P->Next;
        return P;
  }
```

Insert routine for separate chaining

```
  void  Insert (ElementType Key, HashTable H)
  {
        Position Pos, NewCell;
        List L;
        Pos = Find (Key, H);
        NewCell = malloc (sizeof (struct ListNode));
L = H->List [Hash (Key, H->TableSize)];
        NewCell->Next = L->Next;
        NewCell->Element = Key;  /* Probably need strcpy! */
L->Next = NewCell;
  }
```

Performance Evaluation of Separate Chaining:
m = Number of slots in hash table.
n = Number of keys to be inserted in has table.

Load factor $\alpha = n/m$.
Expected time to search = $O(1 + \alpha)$.

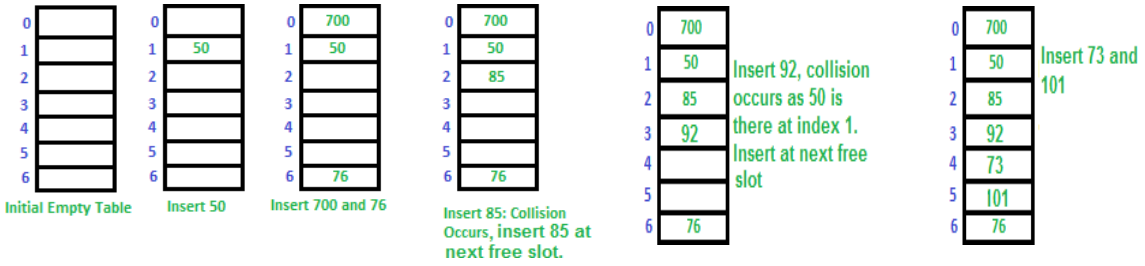
Expected time to insert/delete = O(1 + α).

Time complexity of search insert and delete is O(1) if Load Factor(α) is O(1).

2. Open addressing:

- Open addressing is a collision resolving strategy in which, if collision occurs alternative cells are tried until an empty cell is found.
- The cells h0(x), h1(x), h2(x),…. are tried in succession, where,

hi(x)=(Hash(x)+F(i))mod Tablesize with F(0)=0.
The function F is the collision resolution strategy.
Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Collision Resolution Strategy in Open Addressing:

1. **Linear probing** - In which F is a linear function of i, F(i)=i. This amounts to trying sequentially in search of an empty cell. If the table is big enough, a free cell can always be found, but the time to do so can get quite large.

Example:

Consider following keys that are to be inserted in the hash table. The table size is 7 ie., 0 to 6

76, 93, 27, 70, 4.

Insert 76 76%7= 6	Insert 93 93%7= 2	Insert 27 27%7=6 (6 is Full, so move to next empty location ie., 0)	Insert 70 70%7=0 (0 is Full, So move to 1)	Insert 4 4%7=4
0	0	0 27	0 27	0 27
1	1	1	1 70	1 70
2	2 93	2 93	2 93	2 93
3	3	3	3	3
4	4	4	4	4 4
5	5	5	5	5
6 76	6 76	6 76	6 76	6 76

2. Quadratic probing - It eliminates the primary clustering problem. If collision occurs, alternative cells are tried until an empty cell is found. In linear probing method, the hash table is represented one-dimensional array with indices that range from 0 to the desired table.
In Quadratic probing the alternative cells are calculated using the formula, $F(i) = i^2$.

H = (Hash(key)+i^2) mod m Where m is a table size or any prime number.

Example:

If we have to insert following elements in the hash table with size 10.

37, 90, 55, 22, 17, 49, 87.

Insert 37
37%7=
2

0	
1	
2	37
3	
4	
5	
6	

Insert 90
90%7=
6

0	
1	
2	37
3	
4	
5	
6	90

Insert 55
55%7=6
(6 is Full, so
$(55+1^2)\%7=0$)

0	55
1	
2	37
3	
4	
5	
6	90

Insert 21
21%7=0
(0 is Full,
so
$(21+1^2)\%$
7=1)

0	55
1	21
2	37
3	
4	
5	
6	90

Insert 14
14%7=0
(0 is Full, so
$(14+1^2)\%7=1$

1 is also full, so
$(14+2^2)\%7=4$)

0	55
1	21
2	37
3	17
4	14
5	
6	90

Insert routine for hash tables with quadratic probing

```
Void  Insert( ElementType Key, HashTable H )
{
Position Pos;
Pos =Find( Key, H );
if( H—>TheCells[Pos ].Info ! = Legitimate )
{
      H—>TheCells[ Pos ].Info = Legitimate;
      H->TheCells [Pos].Element =Key;
}
}
```

3. Double hashing - in which $F(i)=i.hash_2(X)$. This formula says that we apply a second hash function to X and probe at a distance hash2(X), 2hash2(X),….,and so on.
A function such as hash2(X)=R-(XmodR), with R a prime smaller than Tablesize.

Example:
Insert 37, 90, 55, 21, 14 into a hashtable with size 7 using Double Hashing method.
Here the prime no chosen is, 5.

Insert 37
37%7=2

0	
1	
2	37
3	
4	
5	
6	

Insert 90
90%7=6

0	
1	
2	37
3	
4	
5	
6	90

Insert 55
55%7=6
(6 is Full, so 5-(55%5)=5 Next 5th loc)

0		1
1		2
2	37	3
3		4
4	55	5
5		
6	90	

Insert 21
21%7=0

0	21
1	
2	37
3	
4	
5	55
6	90

Insert 14
14%7=0
(0 is Full, so 5-16%5=4)

0	21	
1		1
2	37	2
3		3
4	14	4
5	55	
6	90	

Comparison of above three:
Linear probing has the best cache performance, but suffers from clustering. One more advantage of Linear probing is easy to compute.

Quadratic probing lies between the two in terms of cache performance and clustering.

Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed

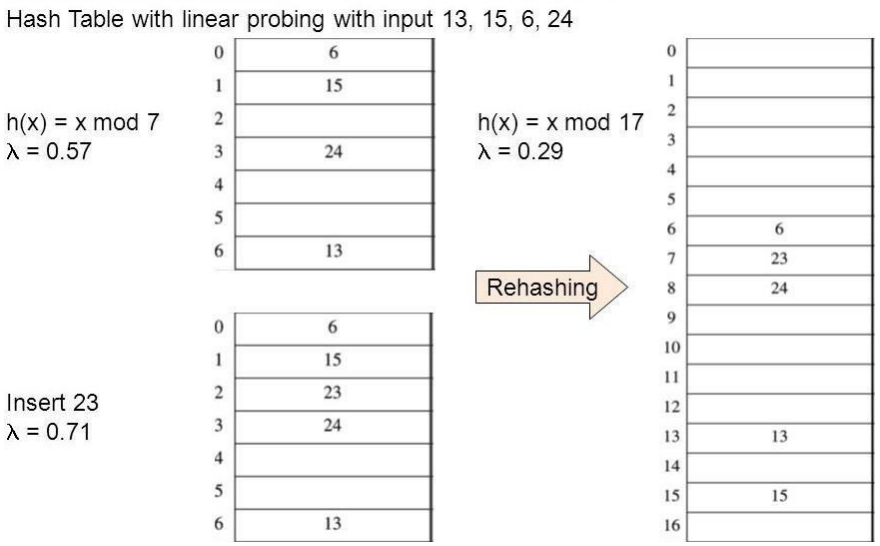
REHASHING

Rehashing is a technique in which the table is resized, i.e., the size of table is doubled by creating a new table. It is preferable if the total size of table is a prime number. There are situations in which the rehashing is required-

- When table is completelyfull.
- With quadratic probing when the table is filled half.
- When insertions fail due to overflow.

In such situations, we have to transfer entries from old table to the new table by re-computing their positions using suitable hash functions.
Example

Insert 13, 15, 6, 24, 23.




```
HashTable Rehash(HashTable H)
{
        Int i,OldSize;
        Cell *OleCells;
        OldCells=H->TheCells;
        OldSize=H->TableSize;
        H=InitializeTable(2*OldSize);
        For(i=0;i<OldSize;i++)
        If(OldCells[i].Info==Legitimate)
                Insert(OldCells[i].Elements,H);
        Free(OldCells);
        Return H;
}
```

Advantages:

1. This technique provides the programmer a flexibility to enlarge the table size if required.
2. Only the space gets doubled with simple hash function which avoids occurrence of collisions.

EXTENDIBLE HASHING

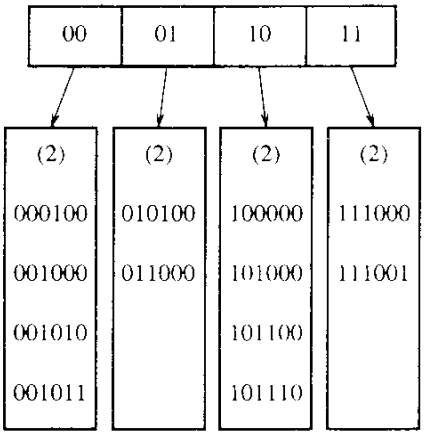
Extendible Hashing: Extendible hashing is a technique, which handles a large amount of data. The data to be placed in the hash table is by extracting certain number of bit.

Reasons for extensible Hashing

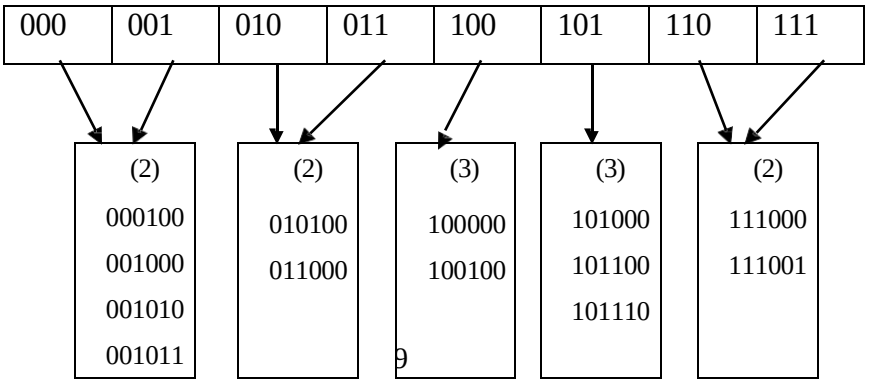
- Extensible hashing grows and shrinks similar to B-trees.
- If either open addressing or separate chaining is used, collision my cause several blocks to be examine during a Find Operation.
- If the table is full, an expensive Rehashing should be performed. To overcome this, the clever alternative is extensible Hashing.

Let us assume that there are N records to store in the disk; at most M records fit in one disk block. Consider the data consists of several six-bit integers.

Extendible hashing: original data

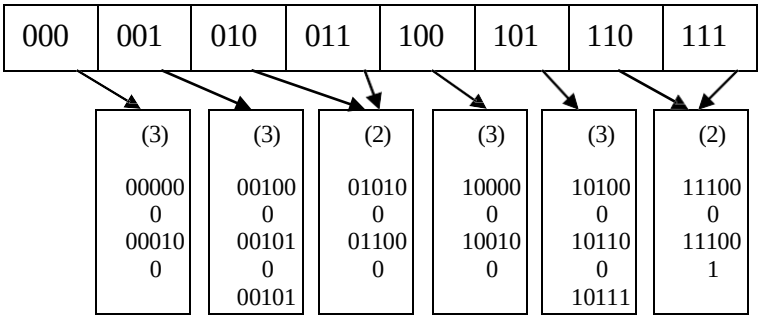


- The root of the tree contains four pointer determined by the leading two bits of the data.
- Each leaf has up to M=4 elements.
- It happens that in each leaf the first two bits are identical; this is indicated by the number in parenthesis. D will represent the number of bits used by the root, which is sometimes known as directory.
- The number of entries in the directory is 2^D. d_L is the number of leading bits that all the elements of some leaf L have in common. d_L will depend on the particular leaf, and $d_L <= D$.
- Suppose that we want to insert the key 100100. This would go into the third leaf, but as the third leaf is already full, there is no room.
- We thus split this leaf into two leaves, which are now determined by the first three bits.
- This requires increasing the directory size to 3. These changes are reflected in the figure below.



All of the leaves not involved in the split are now pointed to by two adjacent directory entries. Thus, although an entire directory is rewritten, none of the other leaves is actually accessed.

If the key 000000 is now inserted, then the first leaf is split, generating two leaves with $d_L = 3$. Since D = 3, the only change required in the directory is the updating of the 000 and 001 pointers. This is given in the figure below:



This very simple strategy provides quick access times for Insert and Find operations on large databases