<u>UNIT –I</u>

Linked Lists

Objective:

- To gain knowledge on linked lists.

Syllabus:

<u>Unit-I: Linked lists</u>

Introduction – Concept of Data Structures, Overview of Data Structures, Implementation of Data Structures.

Linked Lists- Single linked list, Circular linked list, Double linked list., Circular Double Linked list.

Learning Outcomes:

At the end of the unit student will be able to:

1. Define a self referential structure.
2. Describe about linked lists.
3. Implement the operations on linked lists.
4. Choose an appropriate linked list for a given problem.
5. Distinguish between single, double , circular and circular double linked list.

Learning Material

Introduction:

Concept of data Structures:

Data: Data means value or set of values.

Examples of data: 1) 34

2) 21/01/1943

3) 21,25,45,67, 98

Information: The term information is used for data with its attribute(s).Information can be defined as meaningful data or processed the data.

Entity: An entity is one that has certain attributes and which may be assigned values.

Example**:** An employee in an organization is an entity.

Entity: Employee

Attributes:	NAME	DOB	DESIGNATION
Values :	Ravi	21/11/1991	Director

Data type: A data type is a term which refers to the kind of data that may appear in computation**.**

Examples: int, float, char, double.

- ***Abstract DataType:***
  - **ADT** specifies a set of data and collection of operations that can be performed on that data.
  - The definition of ADT only mentions ***What Operations are to be Performed but not how it is implemented***.
  - It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
  - It is called "**abstract**" because it gives an implementation independent view (overview).

- The process of providing only the essentials and hiding the details is known as **Abstraction.**
- Abstract data type is also known as **user- defined data type.**

Overview of data structures:

Definition of data structure: The mathematical or logical representation of data elements is known as data structure. Data structures are also known as fundamentals of data structures or classic data structures.

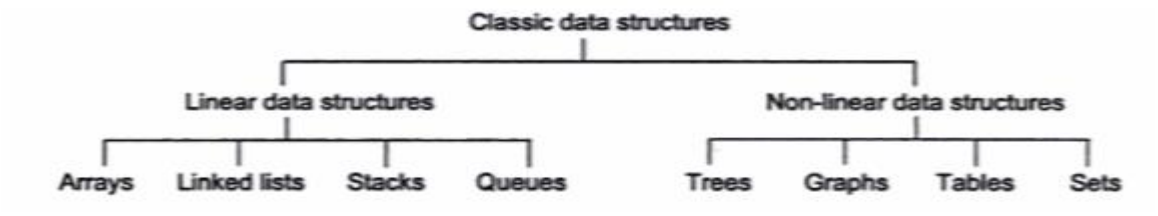


Fig: Classification of data structures

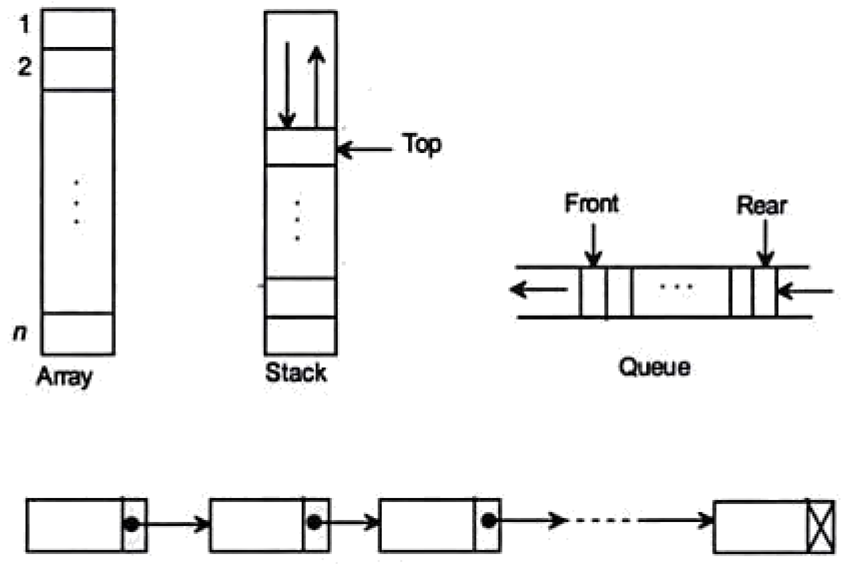
The classic data structures are classified into two types:

1). Linear data structure: In linear data structure data is stored in consecutive memory location or in a sequential form.

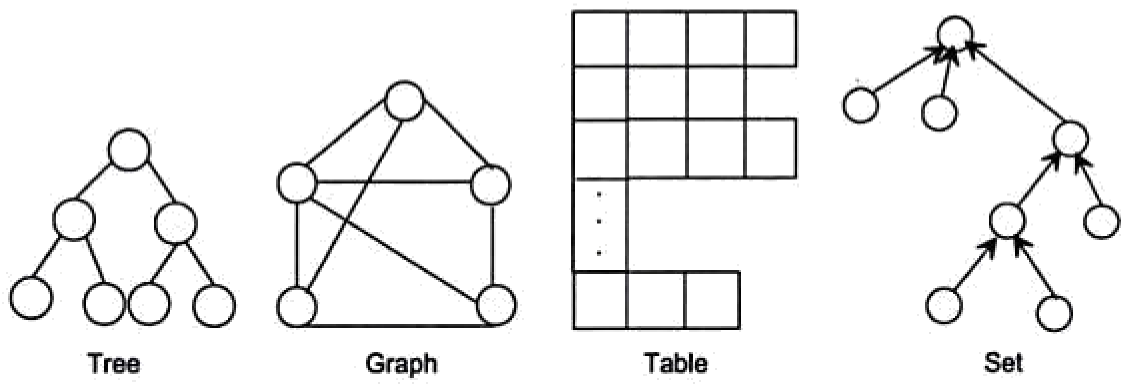
Ex: Arrays ,linked lists, stacks, queues.

2). Non-linear data structure: In non linear data structure data is stored in non consecutive memory locations or not in a sequential form.

Ex: Trees, graphs, tables, sets.



(a) Linear data structures



(b) Non-linear data structures

Implementation of data structure:

➢ **OPERATIONS ON DATA STRUCTURES:**

The basic operations that are performed on data structures are as follows:

1) ***Traversing:*** It means to access each data item exactly once so that it can be processed.

For example, to print the names of all the students in a class.

2) ***Searching:*** It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items.

For example, to find the names of all the students who secured 100 marks in mathematics?

3) ***Inserting:*** ***It*** is used to add new data items to the given list of data items.

   For example, to add the details of a new student who has recently joined the course?

4) ***Deleting:*** ***It*** means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.

5) ***Sorting:*** Data items can be arranged in some order like ascending order or descending order depending on the type of application.
   For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.

6) ***Merging:*** *Lists* of two sorted data items can be combined to form a single list of sorted data items.

➢ **LINKED LISTS:**

- In arrays once memory is allocated, it can't extended any more. So array is known as static Data Structure.
- Linked List is dynamic Data Structure, where amount of memory required can be vary during it use.

Definition: A Linked List is an ordered collection of finite, homogeneous data elements called nodes.
Where the linear order is maintained by means of links or pointers.

- The representation of node is as follows:

Data	Link

Node: an element in Linked List

- A node consists of two parts. i. e . **Data part** and **Link part**.
- The data part contains actual data to be represented.
- The link part is also referred as address field, which contains address of the next node.

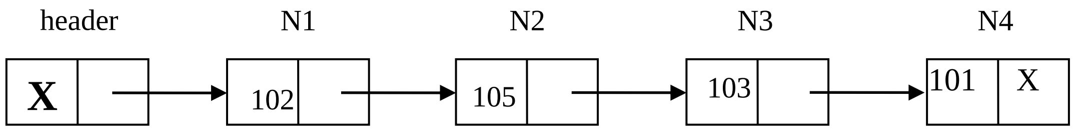
Types of linked lists

1. Single Linked List (SLL)

2. Circular Linked List (CLL)
3. Double Linked List (DLL)
4. Circular Double Linked List (CDLL)

1.Single Linked List (SLL):

- In SLL, each node contains only one link, which points to the next node in the list.
- The pictorial representation of SLL is as follows.



SLL with 4 nodes

- Here header is an empty node, i.e. data part is NULL, represented by X mark.
- The link part of the header node contains address of the first node in the list.
- In SLL, the last node link part contains NULL.
- In SLL we can move from left to right only. So SLL is called as one way list.

- **Operations on Single Linked List**

  1. Traversing a SLL
  2. Insertion of a node in to SLL
  3. Deletion of a node from SLL
  4. Search for a node in SLL

1. Traversing a SLL

Traversing a SLL means, visit every node in the list starting from first node to the last node.

Algoritm SLL_Traverse(header)
Input: header is a header node.
Output: Visiting of every node in SLL.

```
1. ptr=header
2. while(ptr.link != NULL)
        a) ptr=ptr.link go to step(b)
        b) print “ptr.data”
        3. end loop
End SLL_Traverse
```

2. Insertion of a node into SLL

- The Insertion of a node in to SLL can be done in various positions.
    - i) Insertion of a node into SLL at beginning.
    - ii) Insertion of a node into SLL at ending.
    - iii) Insertion of a node into SLL at any position.
- For insertion of a node into SLL, we must get node from memory bank.
- The procedure for getting node from memory bank is as follows:

Procedure for getnewnode( )

```
        1. Check for availability of node in memory bank
        2. if ( AVAIL = NULL)
                a) print "Required node is not available in memory"
                b) return NULL
        3. else
                a) return address of node to the caller
        4. end if
End Procedure for getnewnode
```

i) Insertion of a node into SLL at beginning

AlgoritmSLL_Insert_Begin(header,x)

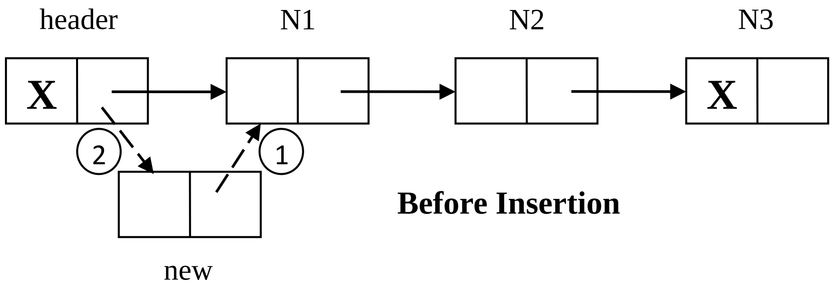
Input: header is a pointer to the header node, x is data part of new node to be inserted.

Output: SLL with new node inserted at beginning.

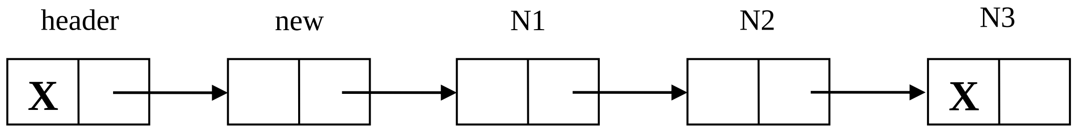
```
        1. new=getnewnode( )
        2. if(new = = NULL)
                a) print "required node was not available in memory, so unable to process"
        3. else
                a) new.link=header.link           /* 1 */
                b) header.link=new          /* 2 */
                c) new.data=x
        4. end if
End SLL_Insert_Begin
```

1. Link part of new node is replaced with address of first node in list, i.e. link part of header node.

2. Link part of header node is replaced with new node address



Before Insertion



After Insertion

.

<u>ii) Insertion of a node into SLL at ending</u>

- To insert a node into SLL at beginning first we need to traverse to last node, then insert as new node as last node.

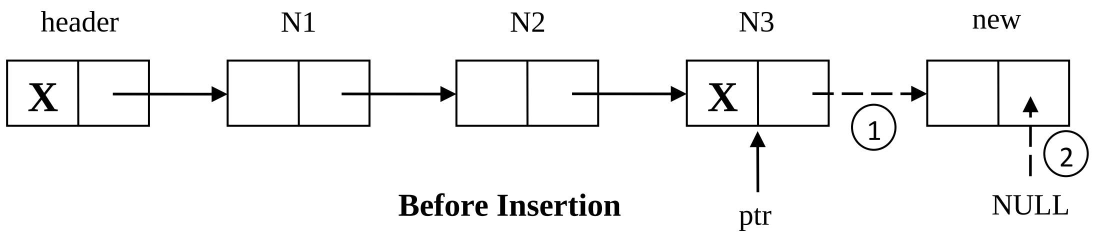
Algorithm SLL_Insert_Ending(header,x)

Input: header is header node, x is data part of new node to be insert.

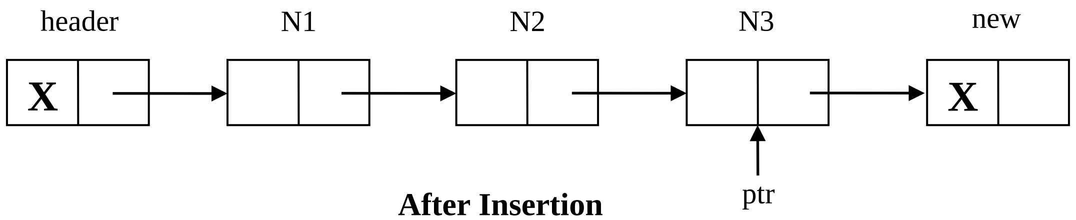
Output: SLL with new node at ending.

```
1. new=getnewnode()
2. if(new = NULL)
        a) print "Required node was not available in memory bank, so unable to
        process"
3. else
        a) ptr=header
        b) while(ptr.link!=NULL)
                i) ptr=ptr.link go to step(b)
        c) end loop
        d) ptr.link=new                    /* 1 */
        e) new.link=NULL           /* 2 */
        f) new.data=x
4. end if
```

End_SLL_Insert_Ending



Before Insertion



After Insertion

1. Previous last node link part is replaced with address of new node.
2. Link part of new node is replaced with NULL, because new node becomes the last node.

iii) Insertion of a node into SLL at any position.

- For insertion of a node at any position in SLL, a key value is specified. Where key being the data part of a node, after this node new node has to be inserting.

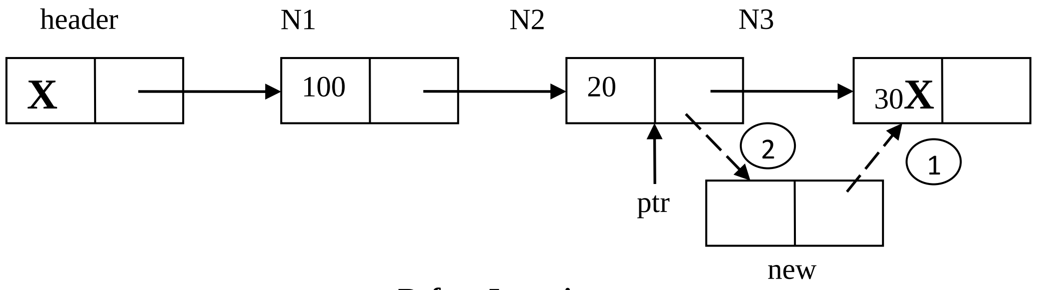
Algorithm SLL_Insert_ANY(header,x,key)

Input: header is header node, x is data pat of new node to be inserting, key is the data part of a node, after this node we want to insert new node.

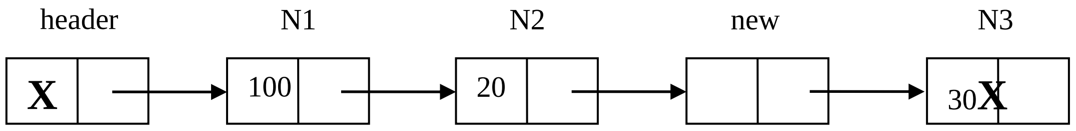
Output: SLL with new node at ANY

```
1. new=Getnewnode( )
2 .if(new = = NULL)
        a. print "required node was not available in memory bank, so unable to
    process"
3. else
        i. ptr=header
        ii. while(ptr.data!=key and ptr.link!=NULL)
                a) ptr=ptr.link
        iii. end loop
        iv. if(ptr.link=NULL and ptr.data!=key)
        a) print "required node with data part as key value is not available, so unable to
    process"
        v. else
                a) new.link=ptr.link         /* 1 */
                b) ptr.link=new                  /* 2 */
                c) new.data=x
        vi. end if
    4. end if.
```

End SLL_insert_ANY



Before Insertion



After Insertion

1. Link part of new node is replaced by the address of next node. i.e. in the above example N3 becomes next node for newly inserting node.
2. Link part of previous node is replaced by the address of new node. i.e. in the above example N2 becomes previous node for newly inserting node.

3. Deletion of a node from SLL

The deletion of a node in from SLL can be done in various positions.

i) Deletion of a node from SLL at beginning.

ii) Deletion of a node from SLL at ending.

iii) Deletion of a node from SLL at any position.

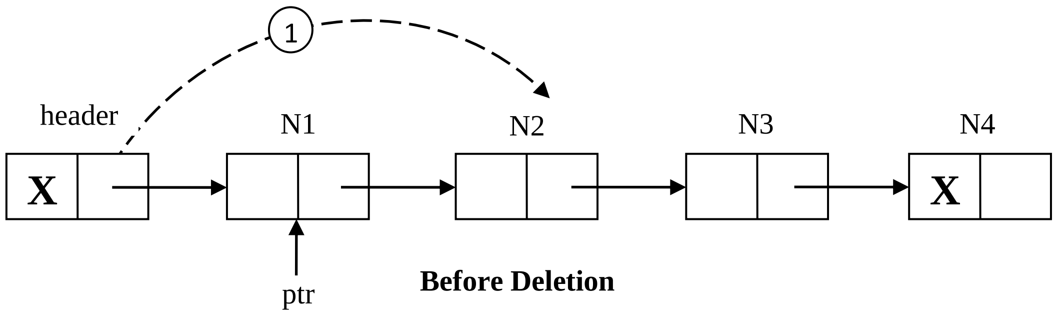
<u>i) Deletion of a node from SLL at beginning</u>

Algorithm SLL_Delete_Begin(header)

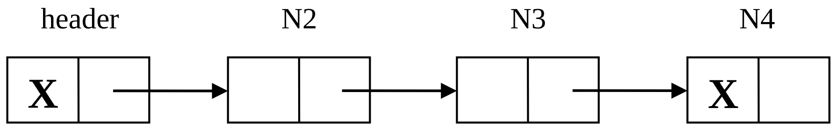
Input: Header is a header node.

Output: SLL with node deleted at Beginning.

```
1. if(header.link = = NULL)
    a) print "SLL is empty, so unable to delete node from list"
    2. else                 /*SLL is not empty*/
            i. ptr=header.link              /* ptr points to first node into list*/
            ii. header.link=ptr.link              /* 1 */
            iii. return(ptr)                /*send back deleted node to memory bank*/
    3. end if
End SLL_Delete_Begin
```



Before Deletion



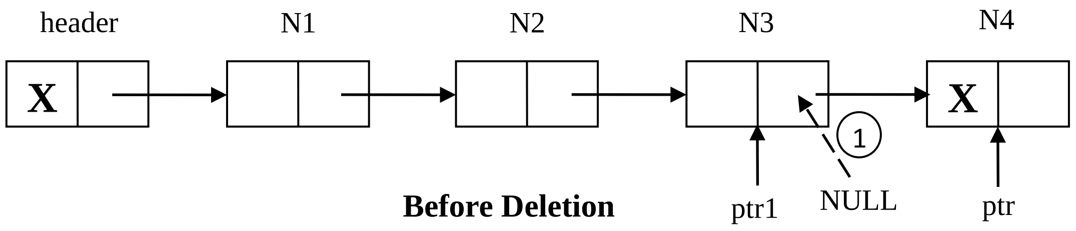
After Deletion

1. Link part of the header node is replaced with address of second node. i.e. address of second node is available in link part of first node.

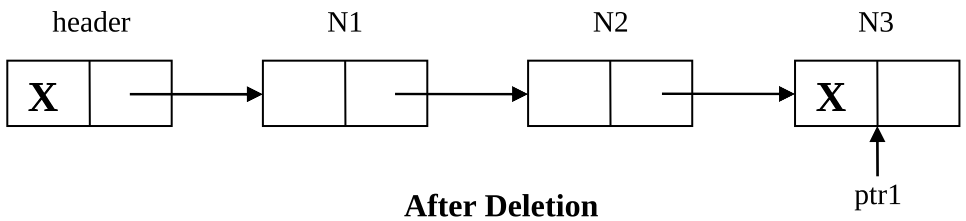
ii) Deletion of a node from SLL at ending

- To delete a node from SLL at ending, first we need to traverse to last node in the list.
- After reach the last node in the list, last but one node link part is replaced with NULL.

```
Algorithm   SLL_ Delete_End (header)
Input:  header is a header node
Output: SLL with node deleted at ending.
        1. if(header.link = = NULL)
                a)print "SLL is empty, so unable to delete the node from list"
        2. else                               /*SLL is not empty*/
                a) ptr=header                         /*ptr   initially  points  to  header
        node*/
                b) while(ptr.link!=NULL)
                        i) ptr1=ptr
                        ii) ptr=ptr.link                /*go to step b*/
                c) end loop
                d) ptr1.link=NULL   /* 1 */
                e) return(ptr)
        3. end if
End   SLL_ Delete_ End
```



Before Deletion



After Deletion

Link part of last but one node is replaced with NULL. Because after deletion of last node in the list, last but one node become the last node.

iii) Deletion of a node from SLL at any position

- For deletion of a node from SLL at any position, a key value is specified.
- Where key being the data part of a node to be deleting.

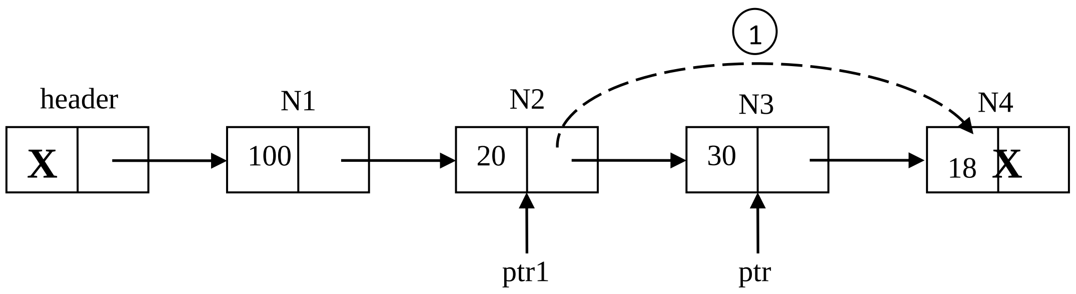
Algorithm SLL_ Delete_ ANY (header,key)

Input: header is a header node, key is the data part of the node to be delete.

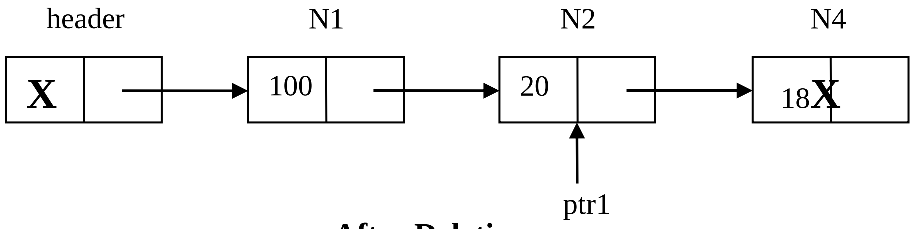
Output: SLL with node deleted at Any position. i.e. Required element.

```
    1. if(header.link = = NULL)
            a)print "SLL is empty, so unable to delete the node from list"
    2. else                                 /*SLL is not empty*/
            a) ptr=header                      /*ptr  initially points to header node*/
            b) while(ptr.link!=NULL and ptr.data!=key)
                    i) ptr1 = ptr
                    ii) ptr=ptr.link        go to step b
            c) end loop
            d) if(ptr.link = = NULL and ptr.data!=key)
            i) print “Required node with data part as key value is not available”
        e) else                         /* node with data part as key value available */
            i) ptr1.link = ptr.link         /* 1 */
            ii) return(ptr)
        f) end if
    3. end if
End   SLL_ Delete_ ANY
```

1. Previous node link part is replaced with address of next node in the list. i.e. in the above example N2 becomes the previous node and N4 becomes the next node for the node to be delete.



Before Deletion



After Deletion

4. Search for a node in SLL:

- For searching a node in SLL, a key value is specified by the user.
- If any node’s data part is equal to key value, then search is successful.
- Otherwise the required node is not available or unsuccessful search.

Algorithm SLL_Search(header,key)

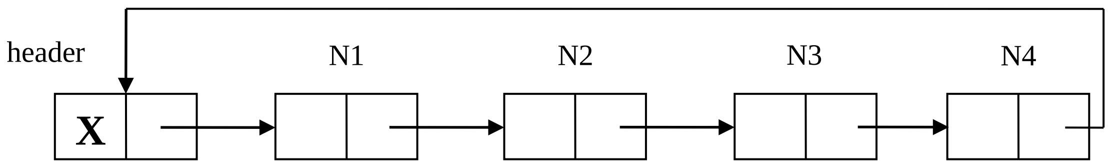
Input: header is a header node
Output: Location is pointer to a node with data part as key value.

```
        ptr = header
        pos=0
        if(ptr==NULL)
          print “list is empty”
        while(ptr.link != NULL && ptr.data!=key)
                ptr = ptr.link
                  pos++
       end loop
                if(ptr.link==NULL)
                  print “key not found,unsuccessful search”
        else
                print “key found, successful search”
        end if
```

End SLL_Search

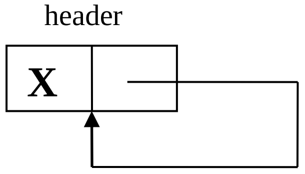
2. <u>Circular Linked List</u> :

- Circular linked list is a special type of linked list.
- In a Circular Linked list, the field of the last node points to the first node of the list.
- It is mainly used in lists that allow to access to nodes in the middle of the list without starting at the beginning.



Circular Linked List

- If a CLL is empty, then the link part of the header node points to itself.



Empty Circular Linked List

➢ **<u>Operations on Circular Linked List:</u>**

1. Insertion of a node in to CLL

2. Deletion of a node from CLL

1. Insertion of a node in to CLL

The Insertion of a node in to CLL can be done in various positions.

i) Insertion of a node into CLL at beginning.

ii) Insertion of a node into CLL at ending.

iii) Insertion of a node into CLL at any position.

<u>i) Insertion of a node into CLL at beginning.</u>

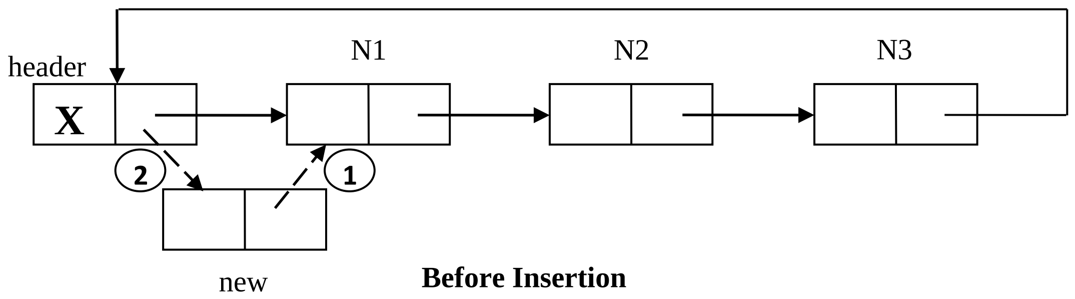
AlgorithmCLL_Insert_Begin(header,x)

Input: header is a header node, x is data part of new node to be insert.

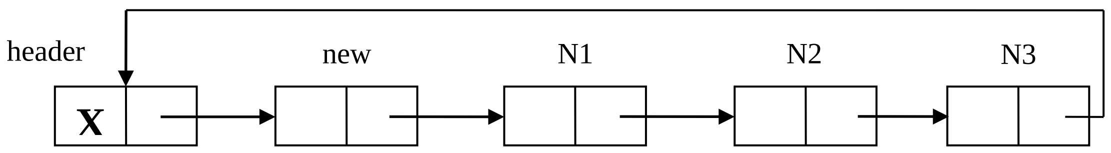
Output: CLL with new node inserted at beginning.

```
1. new=getnewnode( )
2. if(new = = NULL)
      a) print "required node was not available in memory, so unable to
process"
3. else
      a) new.link = header.link                /* 1 */
      b) header.link = new                /* 2 */
c )new.data = x
4. end if
```

End CLL_Inset_Begin



Before Insertion



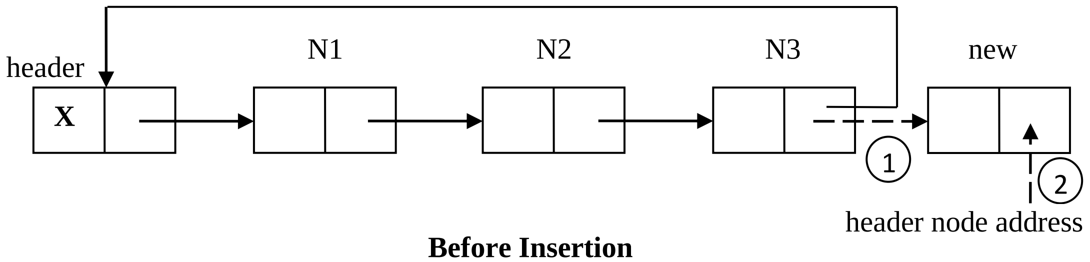
After Insertion

1. Link part of new node is replaced with address of first node in list, i.e. link part of header node.

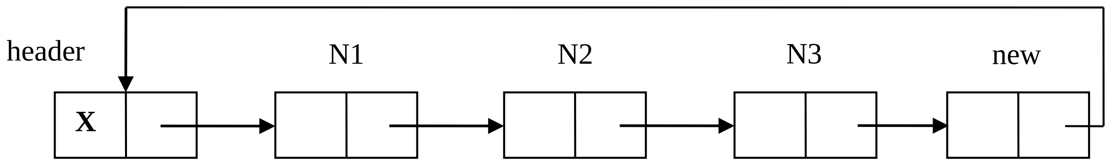
2. Link part of header node is replaced with new node address.

ii) Insertion of a node into CLL at ending.

```
AlgorithmCLL_Insert_END(header,x)
Input: header is a header node, x is data part of new node to be insert.
Output: CLL with new node inserted at ending.
    1. new=getnewnode()
    2 .if(new = = NULL)
        a) print "required node was not available at memory bank, so unable to
    process"
    3. else
        a) ptr=header
        b) while(ptr.link != header)
            i) ptr=ptr.link      //goto step b
        c) end loop
        d) ptr.link=new
        e) new.link=header
        f) new.data=x
    4. end if
End CLL_Insertion_END
```



Before Insertion



After Insertion

1. Previous last node link part is replaced with address of new node.
2. Link part of new node is replaced with address of header node, because new node becomes the last node.

iii) Insertion of a node into CLL at any position.

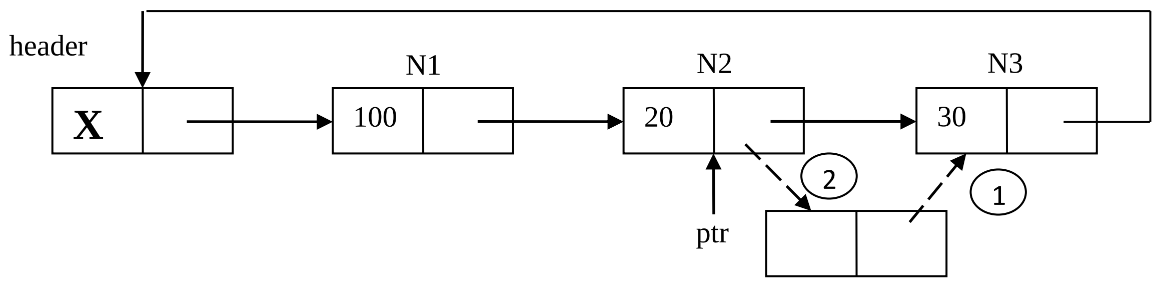
Algorithm CLL_Insert_ANY(header,x,key)

Input: header is header node, x is data pat of new node to be insert, key is the data part of a node, after this node we want to insert new node.

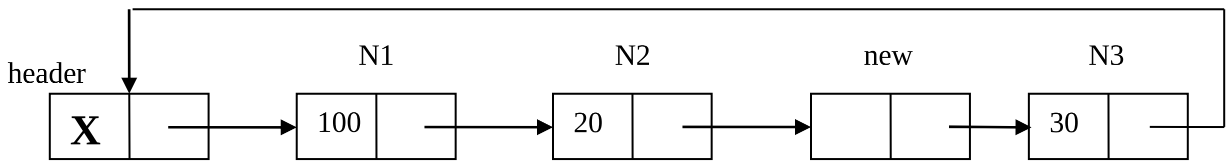
Output: CLL with new node at ANY

```
    1. new=Getnewnode()
    2 .if(new=NULL)
        a. print "required node was not available in memory bank, so unable to
    process"
    3. else
        i. ptr=header
        ii. while(ptr.data! =key and ptr.link!=header)
            a) ptr=ptr.link
        iii. end loop
        iv. if(ptr.link=header and ptr.data!=key)
            a) print "required node with data part as key value is not available, so
    unable to process"
        v. else
            a) new.link=ptr.link
            b) ptr.link=new
            c) new.data=x
        vi. end if
    4. end if.
```

End CLL_insert_ANY



Before Insertion



After Insertion

1. Link part of new node is replaced by the address of next node. i.e. in the above example N3 becomes next node for newly inserting node.
2. Link part of previous node is replaced by the address of new node. i.e. in the above example N2 becomes previous node for newly inserting node.

2. Deletion of a node from CLL:

The Deletion of a node from CLL can be done in various positions.

i) Deletion of a node from CLL at beginning.

ii) Deletion of a node from CLL at ending.

iii) Deletion of a node from CLL at any position

<u>i) Deletion of a node from CLL at beginning</u>

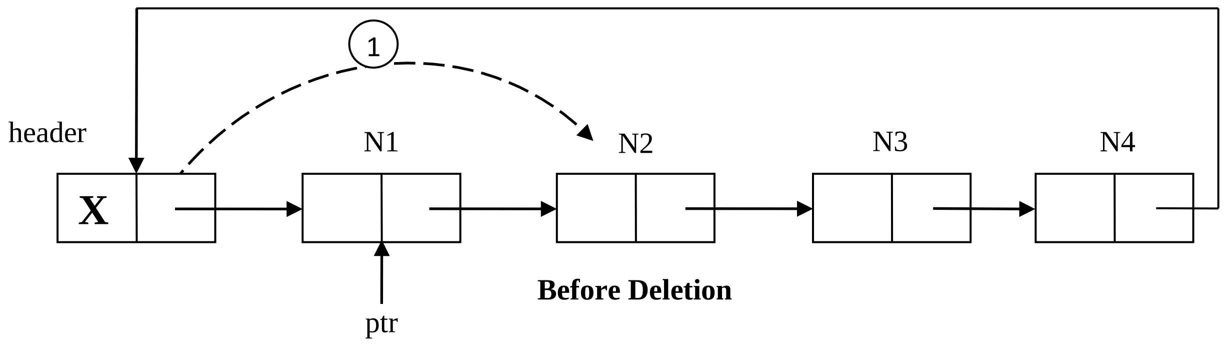
Algorithm CLL_Delete_Begin(header)

Input: Header is a header node.

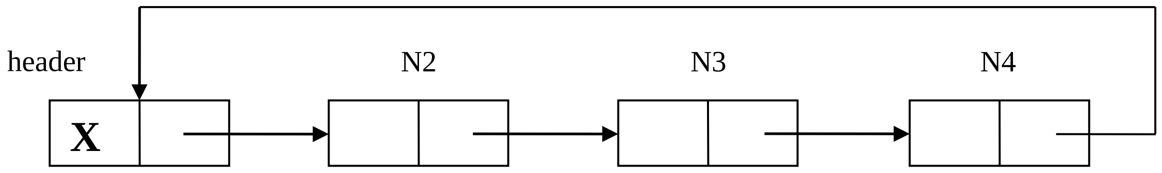
Output: CLL with node deleted at Beginning.

```
1. if(header.link = = header)
        a) print "CLL is empty, so unable to delete node from list"
2. else                    /*DLL is not empty*/
        i. ptr=header.link            /* ptr points to first node into list*/
        ii. header.link=ptr.link      /* 1 */
```

```
            iii. return(ptr)          /*send back deleted node to memory bank*/
        3. end if
    End CLL_Delete_Begin
```



Before Deletion



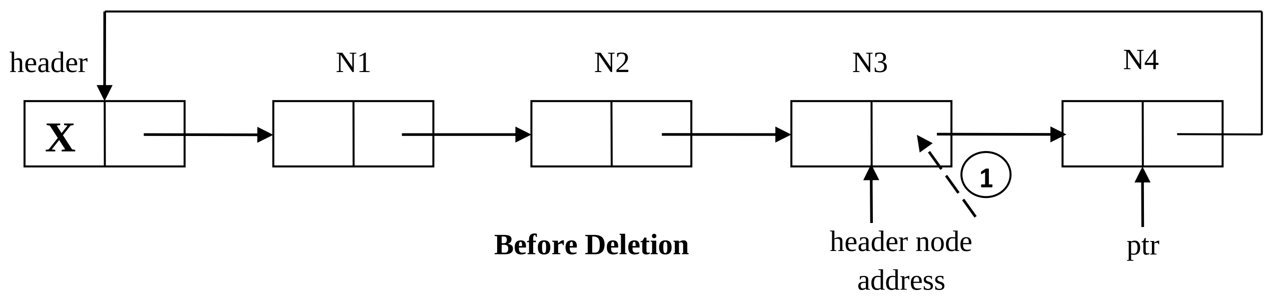
After Deletion

1. Link part of the header node is replaced with address of second node. i.e. address of second node is available in link part of first node.

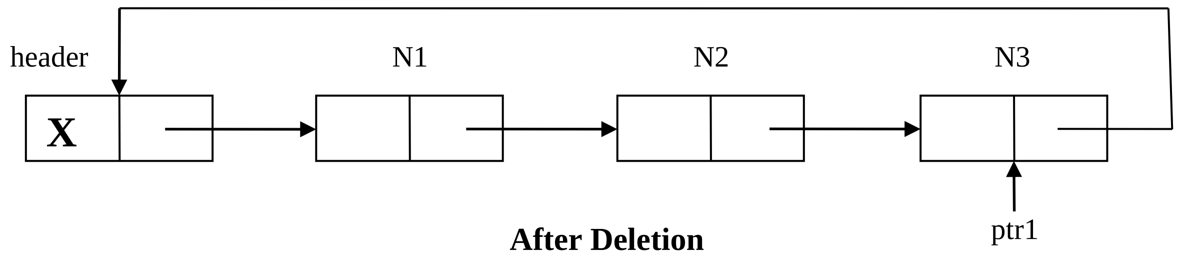
ii) Deletion of a node from CLL at ending

```
Algorithm   CLL_ Delete_ End (header)
Input:  header is a header node
Output: CLL with node deleted at ending.
    1. if(header.link = = header)
        a)print "CLL is empty, so unable to delete the node from list"
    2. else                                /*CLL is not empty*/
        a) ptr=header  /*ptr  initially points to header node*/
        b) while(ptr.link!=header)
            i) ptr1=ptr
            ii) ptr=ptr.link       /*go to step b*/
        c) end loop
        d) ptr1.link=header        /* 1 */
        e) return(ptr)
        3. end if
End   CLL_ Delete_ End
```

1. Link part of last but one node is replaced with address of header node. Because after deletion of last node in the list, last but one node become the last node.



Before Deletion



After Deletion

iii) Deletion of a node from CLL at any position

- For deletion of a node from CLL at any position, a key value is specified. Where key being the data part of a node to be deleting.

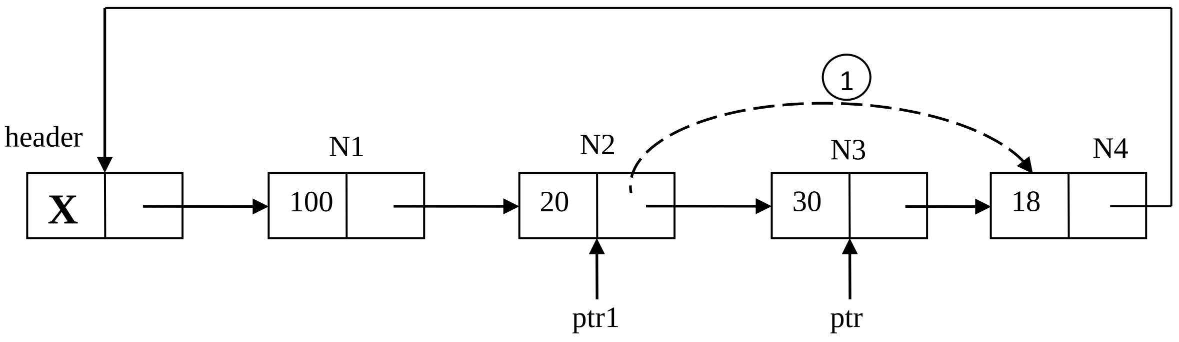
Algorithm   CLL_ Delete_ ANY (header,key)

Input: header is a header node, key is the data part of the node to be delete.

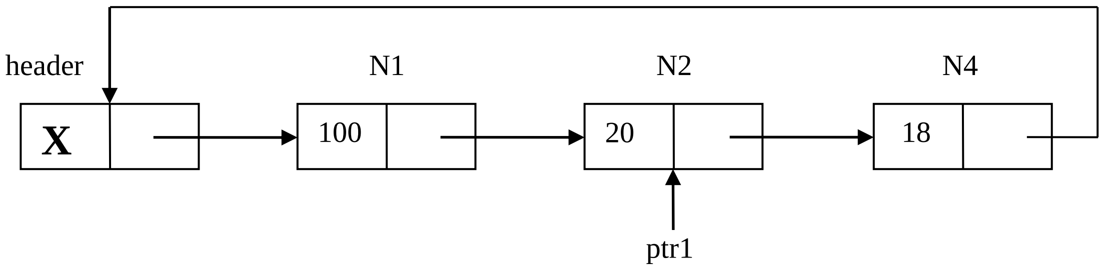
Output: CLL with node deleted at Any position. i.e. Required element.

```
        1. if(header.link = = header)
                a)print "SLL is empty, so unable to delete the node from list"
        2. else                                   /*CLL is not empty*/
                a) ptr=header  /*ptr  initially points to header node*/
                b) while(ptr.link!=header and ptr.data!=key)
```

```
                    i) ptr1 = ptr
                    ii) ptr=ptr.link          go to step b
               c) end loop
               d) if(ptr.link = = header and ptr.data!=key)
                    i) print "Required node with data part as key value is not available"
               e) else                              /*node with data part as key value available
                    i) ptr1.link = ptr.link                    /* 1 */
                    ii) return(ptr)
               f) end if
          3. end if
End   CLL_ Delete_ ANY
```



Before Deletion



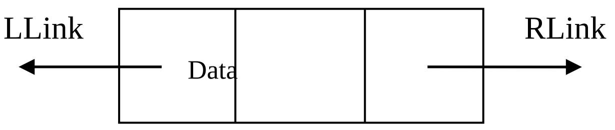
After Deletion

1. Previous node link part is replaced with address of next node in the list. i.e. in the above example N2 becomes the previous node and N4 becomes the next node for the node to be delete.

3. Double Linked List:

- In a SLL one can move from the header node t o any node in one direction only. i.e. from left to right.

- A DLL is a two way list. Because one can move either from left to right or right to left.
- In DLL, each node maintains two links.



Structure of a node in DLL

- Here**LLink** refers Left Link and **RLink** refers Right Link.
- The LLink part of a node in DLL always points to the previous node. i.e. LLink part of a node Consists address of previous node.
- The RLink part of a node in DLL always points to the next node. i.e. RLink part of a node Consists address of next node.

➢ <u>**Operations on Double Linked List:**</u>

1. Insertion of a node in to DLL
2. Deletion of a node from DLL
3. Search and Traversal of a node from DLL

1. **Insertion of a node in to DLL:**

The Insertion of a node in to DLL can be done in various positions.

i) Insertion of a node into DLL at beginning.

ii) Insertion of a node into DLL at ending.

iii) Insertion of a node into DLL at any position.

- For insertion of a node into DLL, we must get node from memory bank. The procedure for getting node from memory bank is same as getting node for SLL from memory bank.

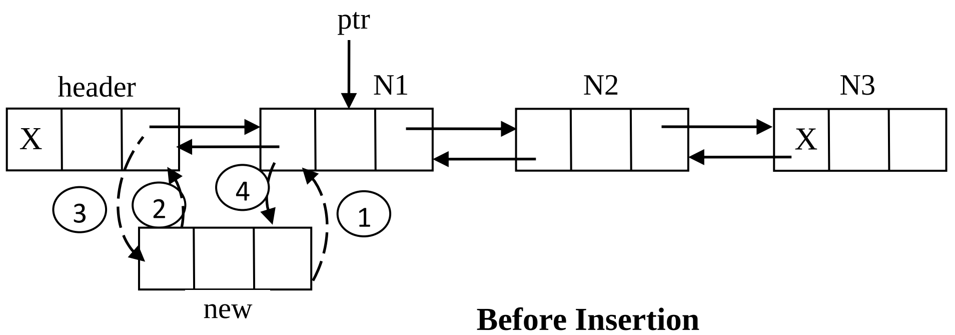
<u>**i) Insertion of a node into DLL at beginning**</u>

Algorithm DLL_insertion_Begin(header,X)

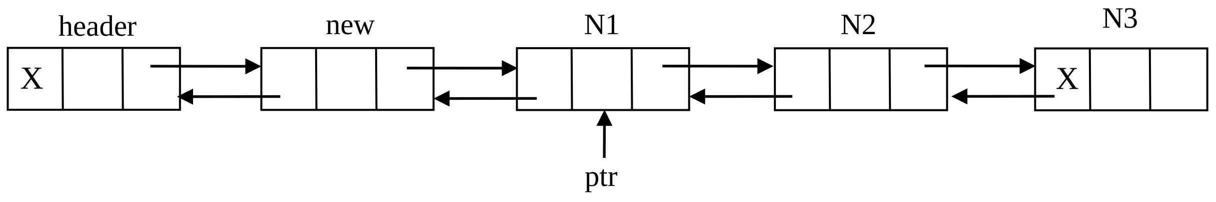
Input: header is a header node.

Output: DLL with new node at begin.

```
1. new=getnewnode()
2. if(new = = NULL)
        a) print "required node is not available in memory"
3. else
        a) ptr=header.rlink
        b) new.rlink=ptr            /* 1 */
        c) new.llink=header         /* 2 */
        d) header.rlink=new         /* 3 */
        e) ptr.llink=new            /* 4 */
    4. end if
End DLL_insertion_Begin
```



Before Insertion



After Insertion

1. Rlink part of new node is replaced with the address of first node in the DLL. i.e. address of first node is available in Rlink part of header node.
2. Llink part of new node is replaced with the address of header.
3. Rlink part of header node is replaced with the address of new node.
4. Llink part of previous first node is replaced with the address of new node.

<u>ii) Insertion of a node into DLL at ending.</u>

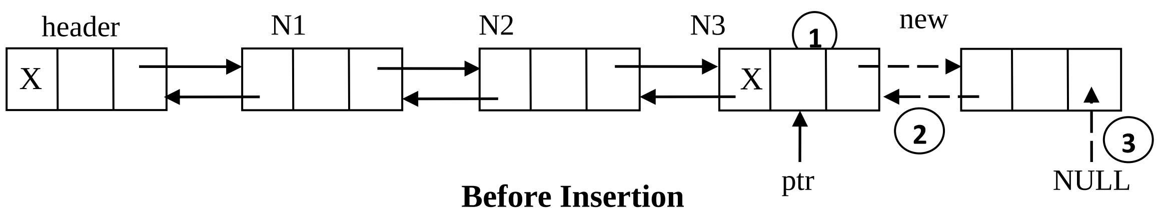
Algorithm DLL_Insert_Ending(Header,x)

Input: Header is the header node, x is the data part of new node to be inserted.

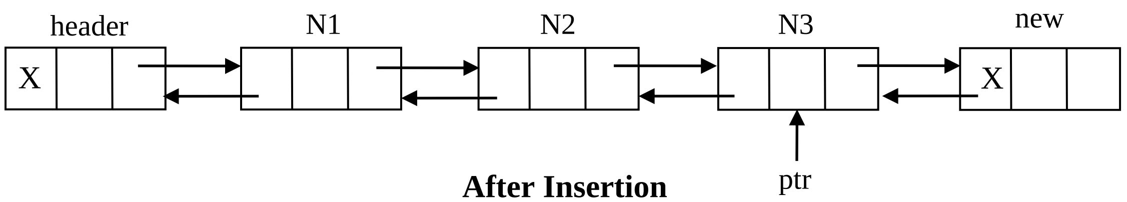
Output: DLL with new node inserted at the ending.

```
1. new=getnewnode()
2. if(new = = NULL)
        a)print "Required node was not available"
3. else
        a) ptr=header
        b) while(ptr.rlink != NULL)
                i) ptr=ptr.rlink                goto step(b)
        c) end while loop
        d) ptr.rlink=new                        /* 1 */
        e) new.llink=ptr                               /* 2 */
        f) new.rlink=NULL                       /* 3 */
        g) new.data=x
    4. end if
```

End DLL_Insertion_Ending



Before Insertion



After Insertion

1. RLink part of last node in the DLL is replaced with address of new node.
2. LLink part of new node is replaced with address of previous last node.
3. RLink part of new node is replaced with NULL. Because newly inserted node becomes the last node in the list.

iii) Insertion of a node into DLL at any position

- For insertion of a node at any position in DLL, a key value is specified.
- Where key being the data part of a node, after this node new node has to be inserting.

Algorithm DLL_Insertion_ANY(header,x,key)

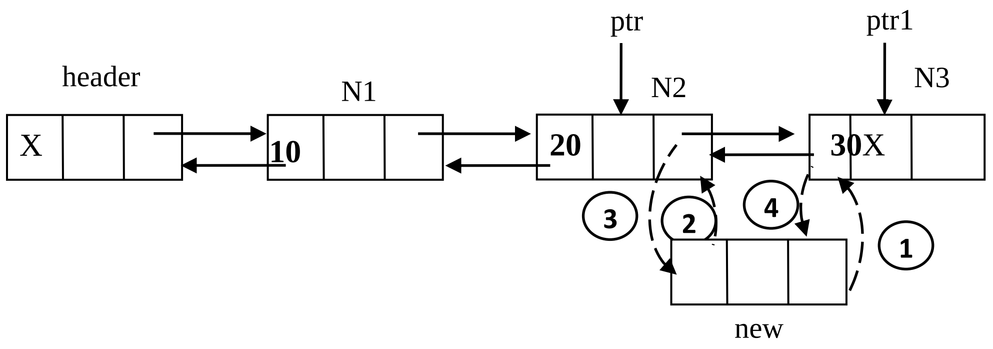
Input: Header is a header node, key is the data part of a node, after that node new node is inserted, x is data part of new node to be insert.

Output: DLL with new node inserted after the node with data part as key value

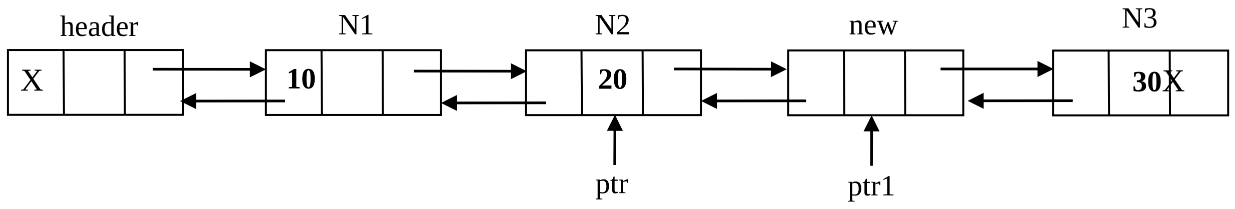
```
    1. new=getnewnode()
    2. if(new = = NULL)
            a) print"required node is not available in memory"
    3. else
```

```
        a) ptr=header
        b) while(ptr.data!=key and ptr.rlink!=NULL)
                i) ptr=ptr.rlink     go to step(b)
        c) end loop
        d) if(ptr.rlink = = NULL and ptr.data != key)
                i) print "required node with key value was not available"
        e) else
                i) ptr1=ptr.rlink
                ii) new.rlink=ptr1              /* 1 */
                iii) new.llink=ptr              /* 2 */
                iv) ptr.rlink=new               /* 3 */
                v) ptr1.llink=new               /* 4 */
                vi)new.data=x
        f) end if
    4. end if
```

EndDLL_Insertion_ANY



Before Insertion



After Insertion

1. RLink part of new node is replaced with the address of next node. i.e. in the above example N3 becomes the next node for newly inserting node.
2. LLink part of new node is replaced with the address of previous node. i.e. in the above example N2 becomes the previous node for newly inserting node.
3. RLink part of previous node is replaced with address of new node.

4. LLink part of next node is replaced with address of new node

2. Deletion of a node from DLL

The deletion of a node in from DLL can be done in various positions.

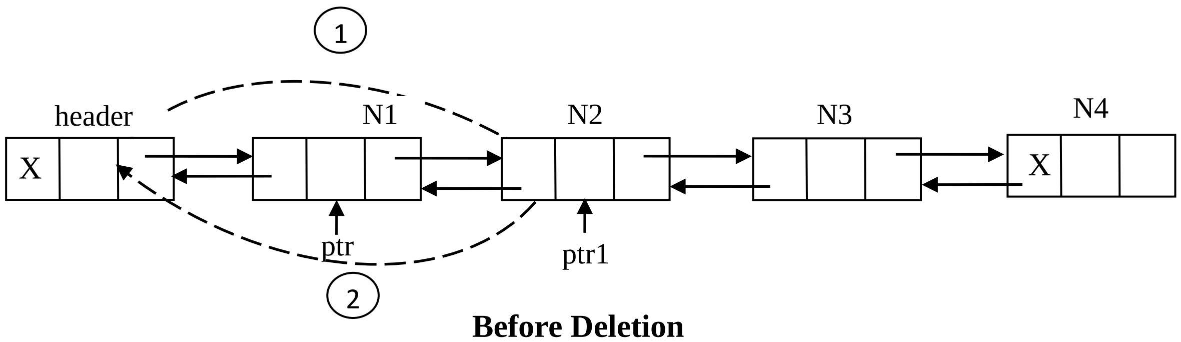
i) Deletion of a node from DLL at beginning.

ii) Deletion of a node from DLL at ending.

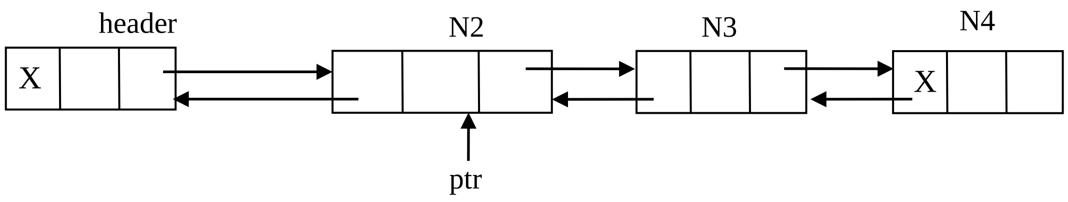
iii) Deletion of a node from DLL at any position.

i**) <u>Deletion of a node from DLL at beginning</u>**

```
Algorithm DLL_Deletion_Begin(header)
Input: header is a header node
Output:  DLL with node deleted at begin
    1. if(header.rlink = = NULL)
        a)  Print "DLL is empty, not possible to perform deletion operation"
    2. else
        a)  ptr=header.rlink
        b)  ptr1=ptr.rlink
        c)  header.rlink=ptr1              /* 1 */
        d)  ptr1.llink=header              /* 2 */
        e)  return(ptr)
    3. End if
        End DLL_Deletion_Begin
```



Before Deletion



After Deletion

1. RLink part of header node is replaced with the address of second node. i.e. address of second node is available RLink part of first node.
2. LLink part of second node is replaced with the address of header node.

ii) Deletion of a node from DLL at ending.

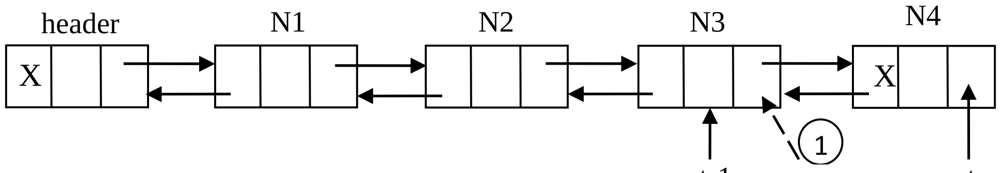
Algorithm DLL_Deletion_End(header)

Input: header is a header node.

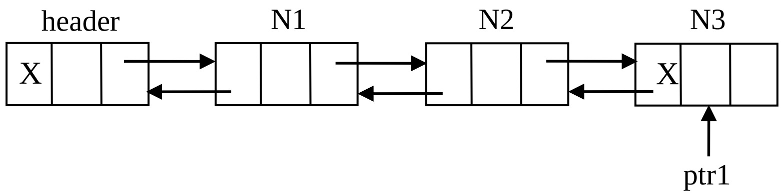
Output: DLL with deleted node at ending.

```
    1. if(header.rlink=NULL)
            a) Print "DLL is empty, not possible to perform deletion operation"
    2. else
            a) ptr=header
            b) while(ptr.rlink != NULL)
                    i) ptr1=ptr
                    ii) ptr=ptr.rlink
            c) end loop
            d) ptr1.rlink=NULL                    /* 1 */
            e) return(ptr)
    3. end if
```

End DLL_Deletioon_Ending



Before Deletion



AfterDeletion

1. RLink part of last but one node in DLL is replaced with NULL. Because last but one node becomes last node.

iii) Deletion of a node from DLL at any position.

- For deletion of a node from DLL at any position, a key value is specified. Where key being the data part of a node to be deleting.

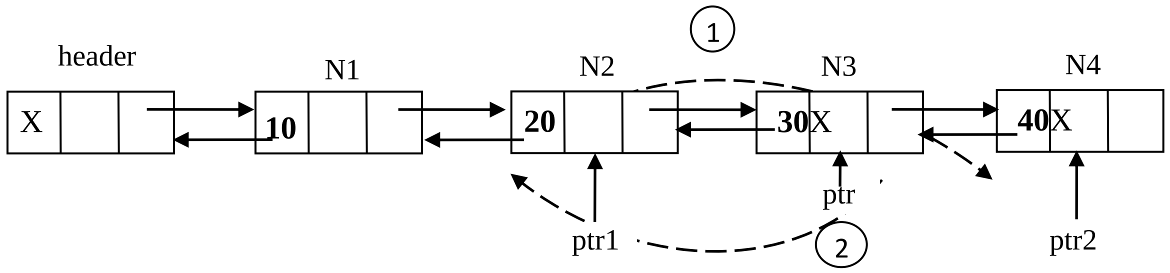
Algorithm DLL_Deletion_Any(header,key)

Input: header is header node, key is the data part of a node to be delete.

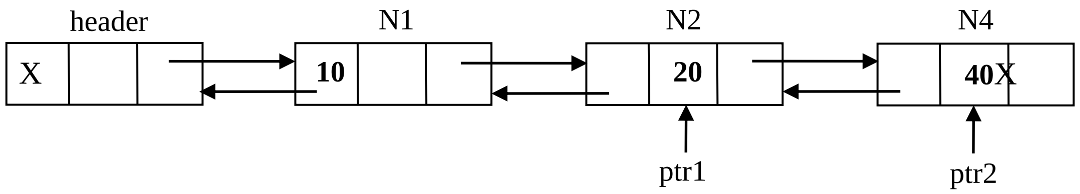
Output: DLL without node as data part is key value.

```
1. if(header.rlink = = NULL)
        a) print "DLL is empty, not possible for deletion operation"
2. else
        i) ptr=header
        ii) while(ptr.data!=key and ptr.rlink!=NULL)
                a) ptr=ptr.rlink
        iii) end loop
        iv) if(ptr.rlink=NULL and ptr.data != key)
                a) print "required node was not available in list"
        v) else
                a) ptr1 = ptr.llink
                b) ptr2 = ptr.rlink
                c) ptr1.rlink = ptr2          /* 1 */
                d) ptr2.rlink = ptr1          /* 2 */
```

```
            vi) end if
      3. end if
End DLL_Deletion_Any
```



Before Deletion



After Deletion

1. RLink part of previous node is replaced with the address of next node. i.e. in the above example N2 become the previous node to node to be delete, N4 becomes the next node to node to be delete.
2. LLink part of next node is replaced with the address of previous node. i.e. in the above example N2 become the previous node to node to be delete, N4 becomes the next node to node to be delete.

Circular Double Linked List:

Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by previous and next pointer and the last node points to first node by next pointer and also the first node points to last node by previous pointer.

```
struct  node
{
        int data;
        struct node *rlink;  //pointer to next node
        struct node *llink;  //pointer to previous node
    };
```

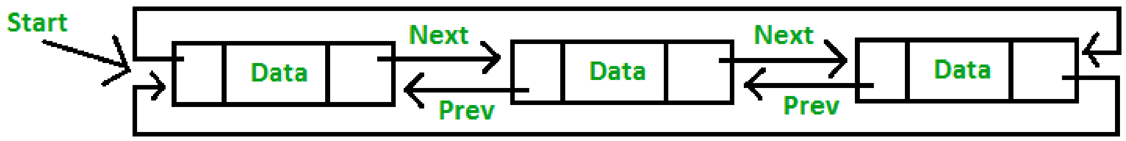


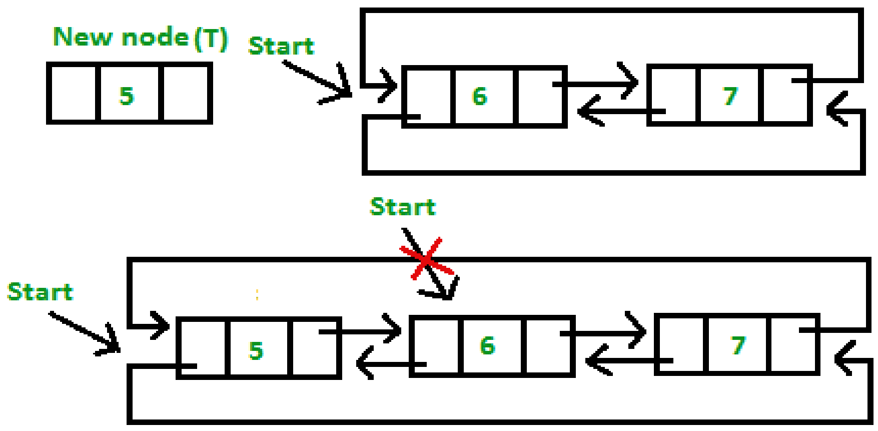
Fig: Circular Double Linked List

➢ <u>Operations on Circular Double Linked List:</u>

1. Insertion of a node into CDLL
2. Deletion of a node from CDLL
3. Search and Traversal of a node from CDLL

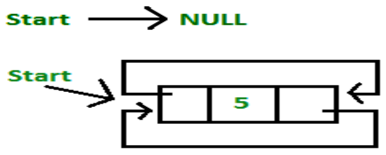
Insertion of a node into CDLL

i) Insertion at the beginning of the list:

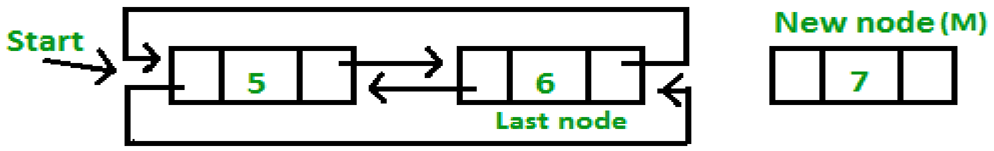


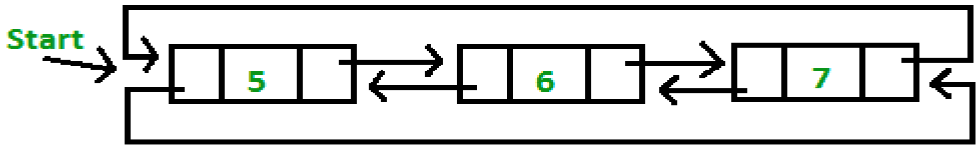
ii) Insertion at the end of list or in an empty list

- **Empty List**

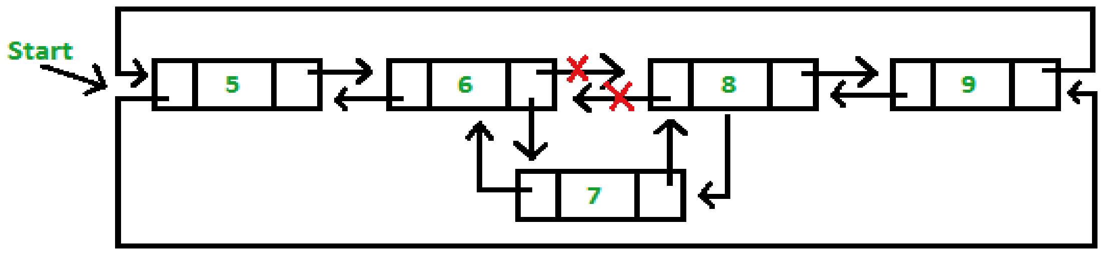


- **At the end of list**



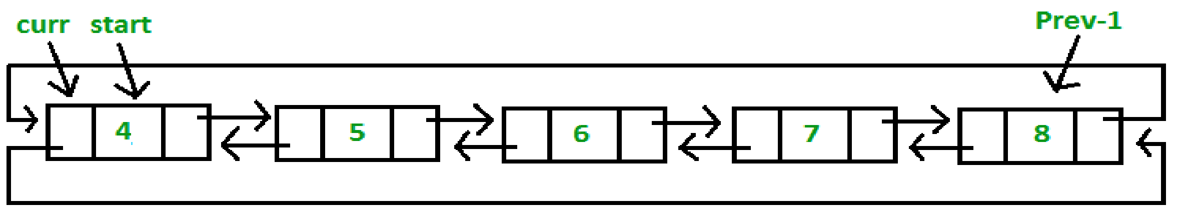


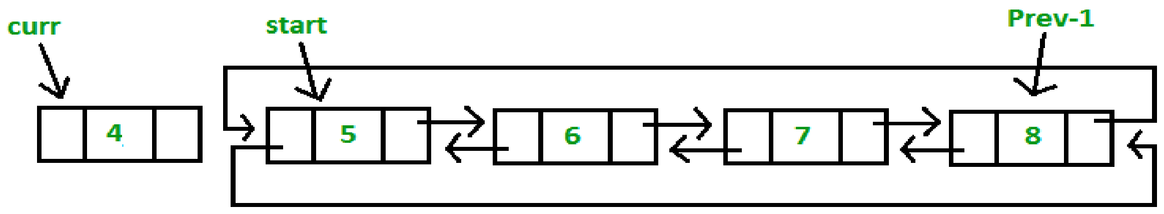
iii) Insertion at any position of the CDLL



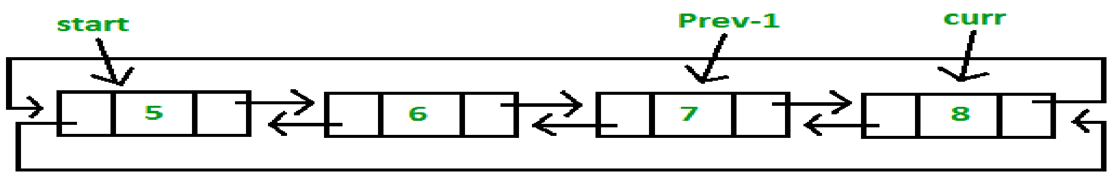
<u>Deletion of a node from CDLL</u>

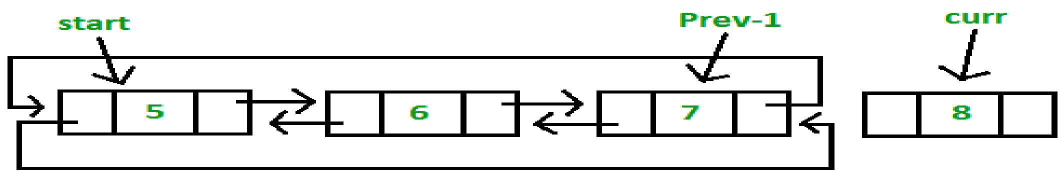
i) Deletion at the beginning of the list:



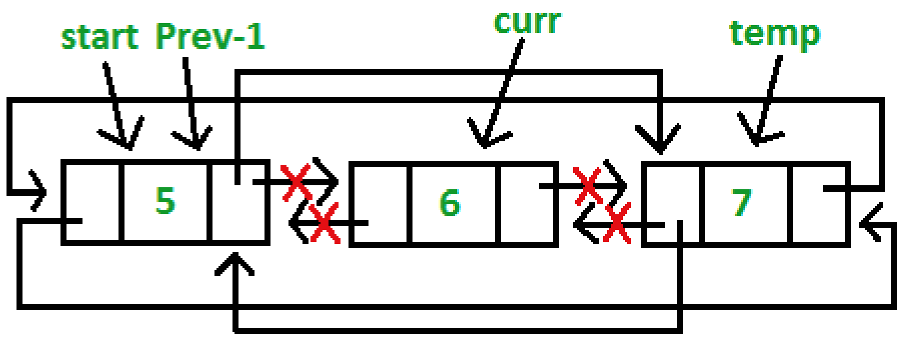


ii) Deletion at the end of the list:





i) Deletion at any position of the list:



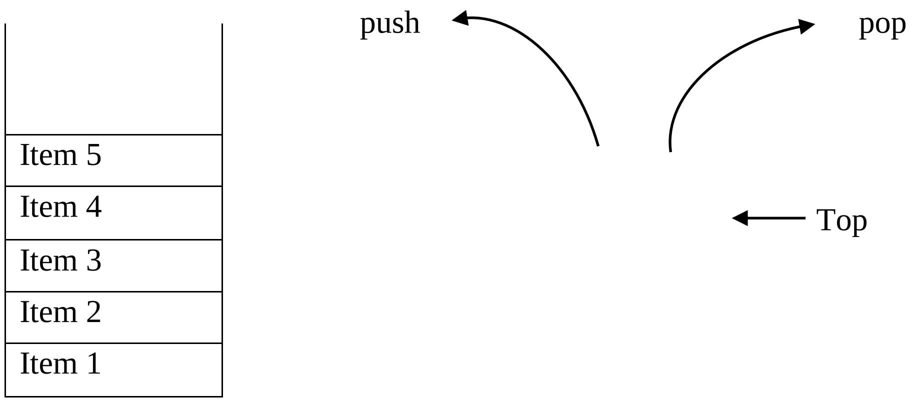
UNIT-II

STACKS

STACKS

- Stack is a linear data structure.

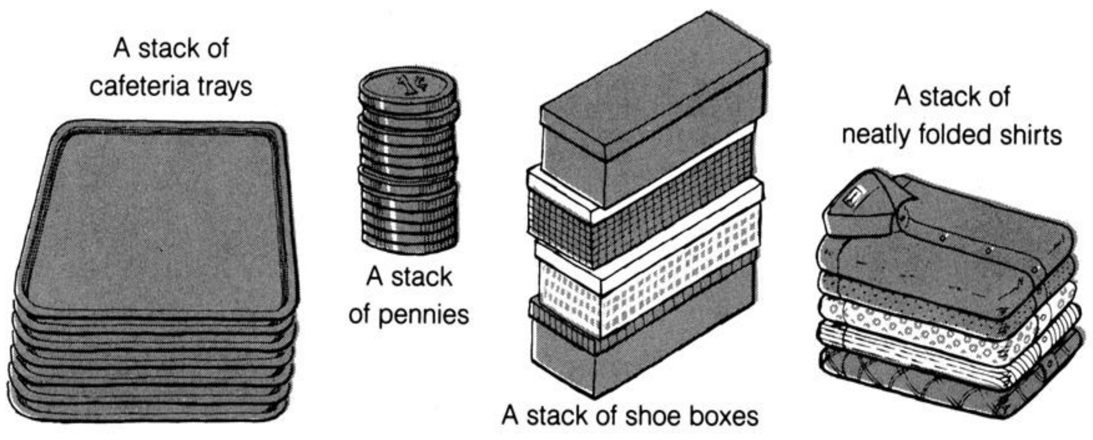
- Stack is an ordered collection of homogeneous data elements, where insertion and deletion operations take place at only end.
- The insertion operation is termed as PUSH and deletion operation is termed as POP operation.
- The PUSH and POP operations are performed at TOP of the stack.
- An element in a stack is termed as ITEM.
- The maximum number of elements that stack can accommodate is termed as SIZE of the stack.
- Stack follows LIFO principle. i.e. Last In First Out.



Schematic diagram of a stack

<u>Applications of Stack</u>:

<u>Real time Applications:</u>



<u>Computer Applications:</u>

- **Expression Evaluation**: Stack is used to evaluate prefix, postfix and infix expressions.
- **Expression Conversion:** An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another

- **Recursive Function Evaluation**
- **Parenthesis Checking:** Stack is used to check the proper opening and closing of parenthesis
- **String Reversal:** Stack is used to reverse a string
- **Syntax Parsing:** Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.
- **Backtracking:** Suppose we are finding a path for solving maze problem

<u>Representation of stack</u>:

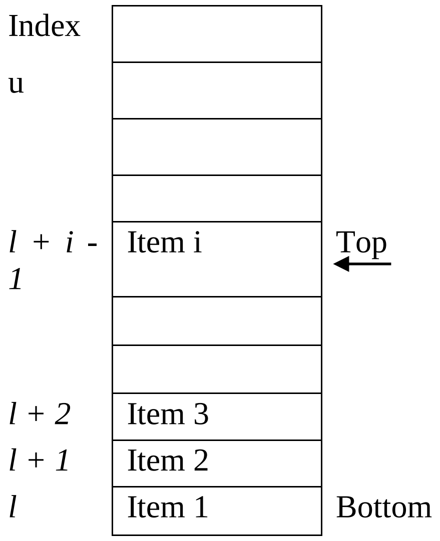
There are two ways of representation of a stack.

1. Array representation of a stack.
2. Linked List representation of a stack.

<u>1. Array representation of a stack.</u>

First we have to allocate memory for array.

Starting from the first location of the memory block, items of the stack can be stored in sequential fashion.



Array representation of stack

In the above figure item i denotes the ith item in stack.

l and u denotes the index ranges.

Usually *l* value is 1 and u value is size.

From the above representation tthe following two status can be stated.

Empty Stack: top < *l* i.e. top < 1

Stack is full: top > = *u*+*l*-1

i.e. top >= size + 1 -1

top >= size

Stack overflow

Trying to PUSH an item into full stack is known as stack overflow.

Stack overflow condition is top >= size

Stack underflow

Trying to POP an item from empty stack is known as Stack underflow.

Stack underflow condition is top < 1 or top = 0

Operations on Stack

PUSH	:	To insert element in to stack
POP	:	To delete element from stack
Status	:	To know present status of the stack

Algorithm Stack_PUSH(item)

Input: item is new item to push into stack

Output: pushing new item into stack at top whenever stack is not full.

```
1. if(top >= size)
    a) print(stack is full, not possible to perform push operation)
2. else
    a) top=top+1
    b) s[top]=item
3.End if
```

End Stack_PUSH

Algorithm Stack_POP( )

Input: Stack with some elements.

Output: item deleted at top most end.

```
1. if(top < 1)
    a) print(stack is empty not possible to pop)
2. else
    a) item=s[top]
    b) top=top-1
    c) print(deleted item)
    3.End if
End Stack_POP
```

Algorithm Stack_Status( )

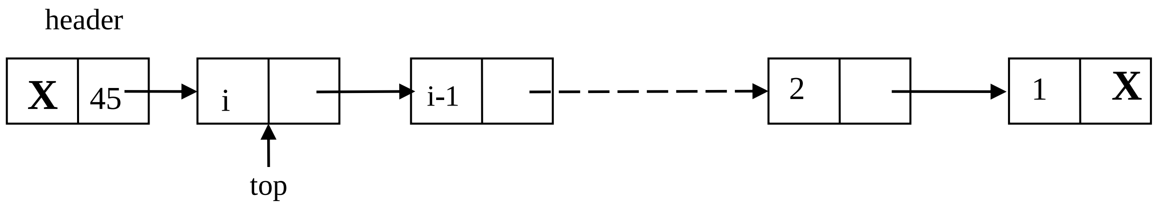
Input: Stack with some elements.

Output: Status of stack. i.e. Stack is empty or not, full or not, top most element in Stack.

```
    1. if(top > = size)
        a) print(stack is full)
    2. else if(top < 1)
        a. print(stack is empty)
    3. else
        a) print(top most item in stack is s[top])
    4. end if
End Stack_Status
```

2. Linked List representation of a stack

- The array representation of stack allows only fixed size of stack. i.e. static memory allocation only.
- To overcome the static memory allocation problem, linked list representation of stack is preferred.
- In linked list representation of stack, each node has two parts. One is data field is for the item and link field points to next node.



Linked List representation of stack

- Empty stack condition is

  top = NULL or header.link=NULL
- Full condition is not applicable for Linked List representation of stack. Because here memory is dynamically allocated.

- In linked List representation of stack, top pointer always points to top most node only. i.e. first node in the list.

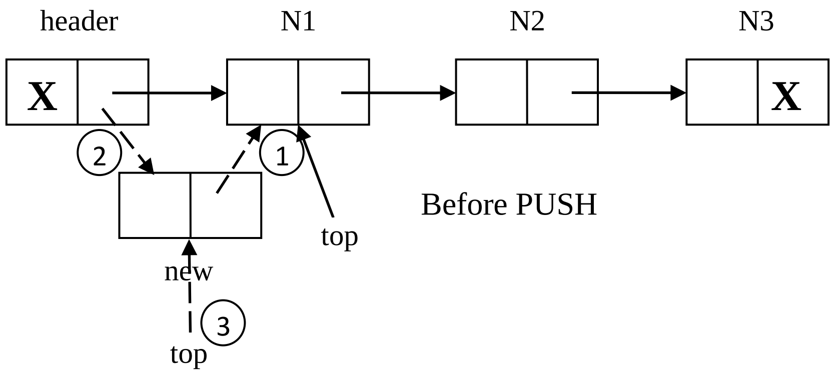
Operations on Stack with linked list representation

PUSH	:	To insert element in to stack
POP	:	To delete element from stack
Status	:	To know present status of the stack

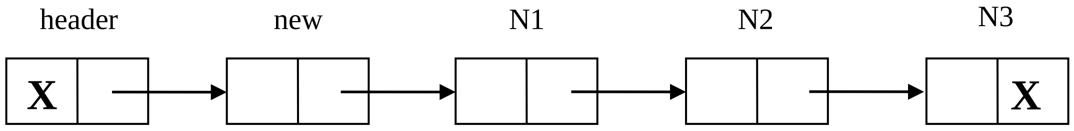
Algorithm Stack_PUSH_LL(item)

```
1. new = getnewnode( )
2. if( new = = NULL)
        a) print(Required node is not available in memeory)
3. else
        a)  new.link=header.link
        b)  header.link=new
        c) top=new
        d) new.data=item
```

End Stack_PUSH_LL



Before PUSH



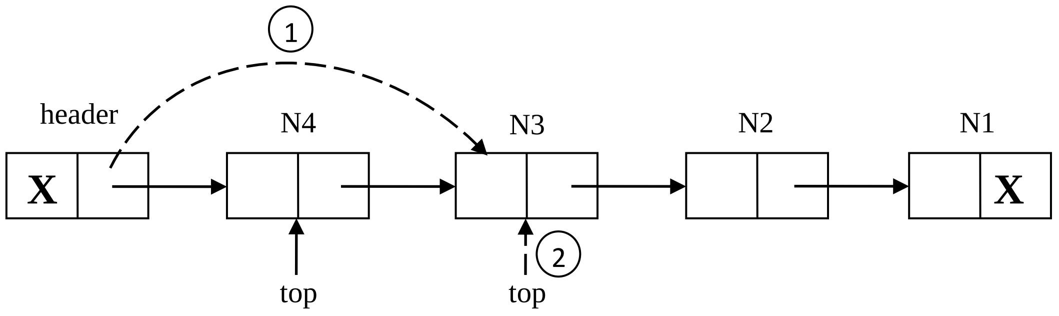
After PUSH

1. The link part of the new node is re placed with address of the previous top most node.
2. The link part of the header node is replaced with address of the new node.
3. Now the new node becomes top most node. So top is points to new node.

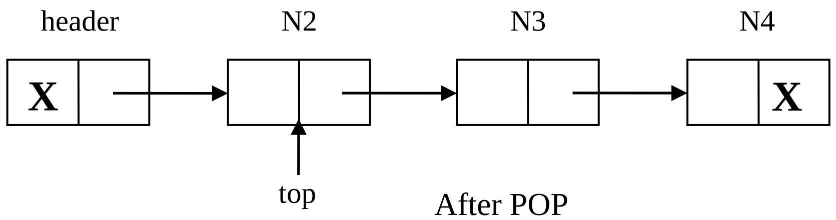
Algorithm Stack_POP_LL( )

```
    1. if( header.link = = NULL)
        a) print(Stack is empty, unable to perform POP operartion)
    2. else
        a) header.link=top.link
        b) item=top.data
        c) top=header.link
```

End Stack_POP_LL



Before POP



After POP

1. Link part of the header node is replaced with the address of the second node in the list.
2. After deletion of top most node from list, the second node becomes the top most node in the list. So top points to the second node.

Algorithm Stack_Status_LL( )

Input: Stack with some elements.

Output: Status of stack. i.e. Stack is empty or not, top most element in Stack.

```
    1. if header.link = = NULL or top = = NULL)
        a) print(Stack is empty)
    2. else
```

```
        a) print( Element present at top of stack is top.data)
    3,end if
```
End Stack_Status_LL

Applications of stack

1. Factorial calculation
2. Infix to postfix conversion
3. Evaluation of postfix expression
4. Reversing list of elements

1. Factorial Calculation

- To calculate the factorial of a given number using stack, we require two stacks. One for storing the parameter n and another stack is hold the return address.
- Let us assume the two stacks, one PARAM for parameter and ADDR for return address.
- Assume PUSH(X,Y) operation for pushing the items X and Y into the stack PARAM and ADDR respectively.

Algorithm Factorial_Stack(n <integer>)

Input: An integer n

Output: Factorial of n

```
    1. top = 0, fact=1
    2. Repeat step 3 and 4 until  n>0
    3. Push(s,n)
    4. n- -
    5. repeat step 6 and 7  until stack empty
    6. f=pop(s)
    7. fact=fact*f
    8. print fact value.
 End Factorial_Stack
```

2. Infix to postfix conversion

An expression is a combination of operands and operators.

Eg. c= a + b

In the above expression a, b, c are operands and +, = are called as operators.

We have 3 notations for the expressions.

i. Infix notation
ii. Prefix notation
iii. Postfix notation

<u>Infix notation</u>: Here operator is present between two operands.

eg. a + b

The format for Infix notation as follows

<operand> <operator> <operand>

<u>Prefix notation</u>: Here operator is present before two operands.

eg. + a b

The format for Prefix notation as follows

<operator> <operand> <operand>

<u>Postfix notation:</u> Here operator is present after two operands.

eg. a b +

The format for Prefix notation as follows

<operand> <operand> <operator>

While conversion of infix expression to postfix expression, we must follow the precedence and associativity of the operators.

Operator	**Precedence**	**Associativity**
^ or $ (exponential)	3	Right to Left
* / %	2	Left to Right
+ -	1	Left to Right

In the above table * and / have same precedence. So then go for associativity rule, i.e. from Left to Right.

Similarly + and - same precedence. So then go for associativity rule, i.e. from Left to Right.

Eg. 1 (A + B) * (C – D)

A B + * (C – D)

A B + * C D –

A B + C D - *

Eg. 2

([(A - { B + C }) * D] $ E + F)

([(A - BC+) * D] $ E + F)

([ABC+-   *   D] $ E + F)

(ABC+-D* $ E   +   F)

(ABC+-D*E$ + F)

ABC=-D*E$F+

To convert an infix expression to postfix expression, we can use one stack.

- Within the stack, we place only operators and left parenthesis only. So stack used in conversion of infix expression to postfix expression is called as operator stack.

Algorithm Conversion of infix to postfix

Input: Infix expression.

Output: Postfix expression.

```
1. Perform the following steps while reading of infix expression is not over
    a) if symbol is left parenthesis then push symbol into stack.
    b) if symbol is operand then add symbol to post fix expression.
    c) if symbol is operator then check stack is empty or not.
        i) if stack is empty then push the operator into stack.
        ii) if stack is not empty then check priority of the operators.
            (I) if priority of current operator > priority of operator present at top of stack
            then push operator into stack.
            (II) else if priority of operator present at top of stack >= priority of current
            operator then pop the operator present at top of stack and add popped operator
            to postfix expression (go to step I)
```

d) if symbol is right parenthesis then pop every element form stack up corresponding left parenthesis and add the poped elements to postfix expression.

2. After completion of reading infix expression, if stack not empty then pop all the items from stack and then add to post fix expression.

End conversion of infix to postfix

3. Evaluation of postfix expression

- To evaluate a postfix expression we use one stack.
- For Evaluation of postfix expression, in the stack we can store only operand. So stack used in Evaluation of postfix expression is called as operand stack.

Algorithm PostfixExpressionEvaluation

Input: Postfix expression

Output: Result of Expression

```
1. Repeat the following steps while reading the postfix expression.
    a) if the read symbol is operand, then push the symbol into stack.
    b) if the read symbol is operator then pop the top most two items of the stack and apply
       the operator on them, and then push back the result to the stack.
2. Finally stack has only one item, after completion of reading the postfix expression. That
   item is the result of expression.
```

End PostfixExpressionEvaluation

4. Reversing List of elements

- A list of numbers can be reversed by reading each number from an array starting from 1st index and pushing into stack.
- Once all the numbers have been push into stack, the numbers can be poped one by one from stack and store ito array from the 1st index.

Algorithm Reverse_List_Stack(a <array>, n <intetger>)

Input : Array a with n elements

Output: Reversed List of elements

```
1. i=1 , top =0
2. while ( I <=n)
    a) top = top + 1
    b) s[top] = a[i]
    c) i = I + 1
3. end while loop
4. i = 1
```

```
    5. while( i <= n)
        a) a[i] = s[top]
        b) top = top – 1
        c) i = i + 1
    6. end while loop
End Reverse_List_Stac
```

****

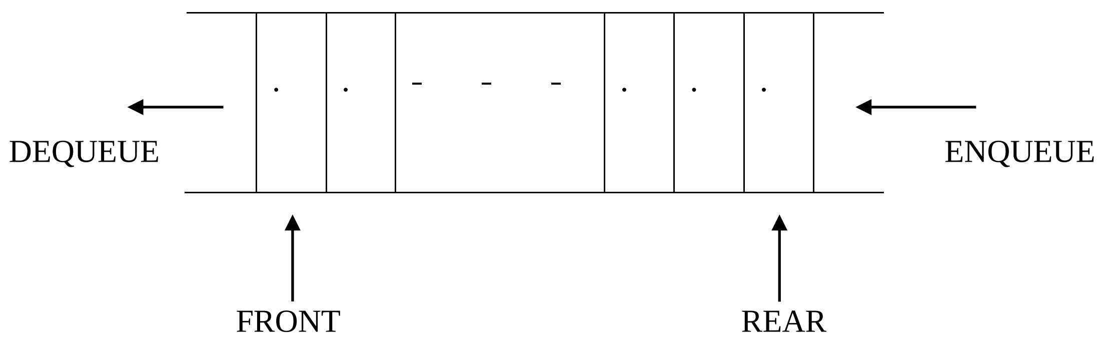
UNIT-III

QUEUES

Queue is a linear Data structure.

Definition: Queue is a collection of homogeneous data elements, where insertion and deletion operations are performed at two extreme ends.

- The insertion operation in Queue is termed as ENQUEUE.
- The deletion operation in Queue is termed as DEQUEUE.
- An element present in queue are termed as ITEM.
- The number of elements that a queue can accommodate is termed as LENGTH of the Queue.
- In the Queue the ENQUEUE (insertion) operation is performed at REAR end and DEQUEUE (deletion) operation is performed at FRONT end.
- Queue follows FIFO principle. i.e. First In First Out principle. i.e. a item First inserted into Queue, that item only First deleted from Queue, so queue follows FIFO principle.



Schematic Representation of Queue

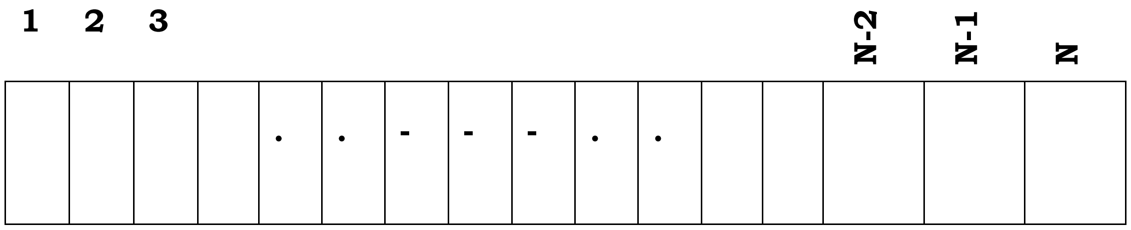
Representation of Queue

A Queue can be represented in two ways

1. Using arrays
2. Using Linked List

1. Representation of Queue using arrays

A one dimensional array Q[1-N] can be used to represent a queue.



Array representation of Queue

In array representation of Queue, two pointers are used to indicate two ends of Queue.

The above representation states as follows.

1. Queue Empty condition

   Front = 0 and Rear = 0

2. Queue Full condition

   Rear = N where N is the size of the array we are taken

3. Queue contains only one element

   Front = = Rear

4. Number of items in Queue is

   Rear – Front + 1

<u>Queue overflow:</u> Trying to perform ENQUEUE (insertion) operation in full Queue is known as Queue overflow.

Queue overflow condition is Rear > = N

Queue Underflow: Trying to perform DEQUEUE (deletion) operation on empty Queue is known as Queue Underflow.

Queue Underflow condition is Front = 0

Operations on Queue

1. ENQUEUE	:	To insert element in to Queue
2. DEQUEUE	:	To delete element from Queue
3. Status	:	To know present status of the Queue

Algorithm Enqueue(item)

Input: item is new item insert in to queue at rear end.

Output: Insertion of new item queue at rear end if queue is not full.

```
1. if(rear = = N)
        a)print(queue is full, not possible for enqueue operation)
2. else
        i) if(front = = 0 and rear = = 0)   /* Q is Empty */
                a) rear=rear+1
                b) Q[rear]=item
                c) front=1
        ii) else
                a) rear=rear+1
                b) Q[rear]=item
        iii) end if
3.end if
```

End Enqueue

While performing ENQUEUE operation two situations are occur.

1. if queue is empty, then newly inserting element becomes first element and last element in the queue. So Front and Rear points to first element in the list.

2. If Queue is not empty, then newly inserting element is inserted at Rear end.

Algorithm Dequeue( )

Input: Queue with some elements.

Output: Element is deleted from queue at front end if queue is not empty.

```
1. if(front = = 0 and rear = = 0)
        a) print(Q is empty, not possible for dequeue operation)
2. else
        i) if(front = = rear)          /* Q has only one element */
                a) item=Q[front]
                b) front=0
                c) rear=0
        ii) else
                a)item=Q[front]
                b)front=front+1
        iii) end if
        iv) print(deleted item is item)
3. end if
```

End Dequeue

While performing DEQUEUE operation two situations are occur.

1. if queue has only one element, then after deletion of that element Queue becomes empty. So Front and Rear becomes 0.
2. If Queue has more than one element, then first element is deleted at Front end.

Algorithm Queue_Status( )

Input: Queue with some elements.

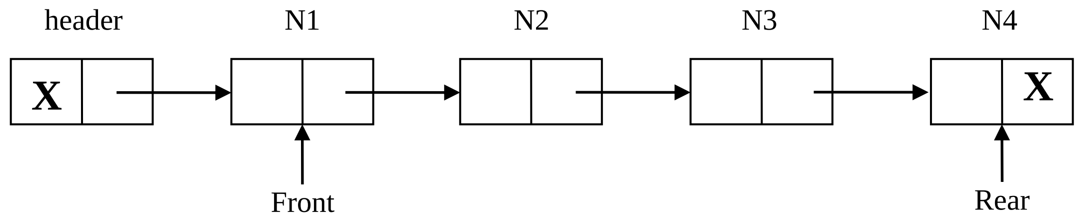
Output: Status of the queue. i.e. Q is empty or not, Q is full or not, Element at front end and rear end.

```
1. if(front = = 0 and rear = = 0)
        a) print(Q is empty)
2. else if (rear = = size )
```

```
        a) print(Q is full)
    3. else
        i) if(front = = rear)
            a) print(Q has only one item)
        ii) else
            a) print(element at front end is Q[front])
            b) print(element at rear end is Q[rear])
        iii) end if
    4. end if
End Queue_Status
```

2. Representation of Queue using Linked List

- Array representation of Queue has static memory allocation only.
- To overcome the static memory allocation problem, Queue can be represented using Linked List.



Linked List Representation of Queue

- In Linked List Representation of Queue, **Front** always points to **First** node in the Linked List and **Rear** always points to **Last** node in the Linked List.

The Linked List representation of Queue stated as follows.

1. Empty Queue condition is

    Front = NULL and Rear = NULL or header.link == NULL

2. Queue full condition is not available in Linked List representation of Queue, because in Linked List representation memory is allocated dynamically.

3. Queue has only one element

    Front = = Rear

Operation on Linked List Representation of Queue

1. ENQUEUE : To insert element in to Queue
2. DEQUEUE : To delete element from Queue
3. Status : To know present status of the Queue

Algorithm Enqueue _LL(item)

Input: item is new item to be insert.

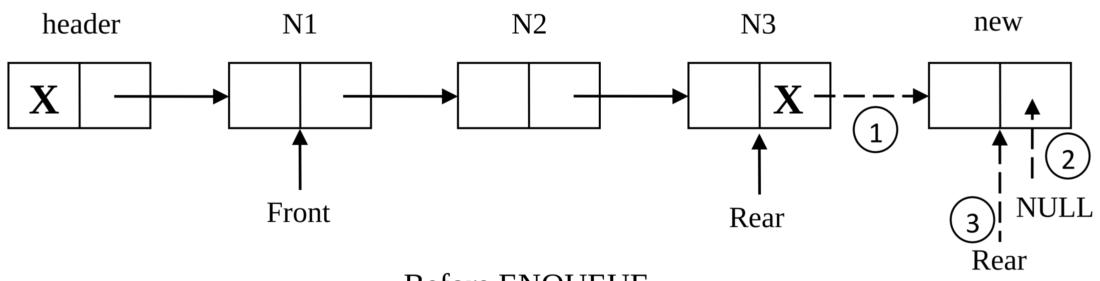
Output : new item i.e new node is inserted at rear end.

```
1. new=getnewnode()
2. if(new = = NULL)
        a) print(required node is not available in memory)
3. else
        i) if(front = = NULL and rear = = NULL)       /* Q is EMPTY */
                a) header.link=new
                b) new.link=NULL
                c) front=new
                d) rear=new
                e) new.data=item
        ii) else                                       /* Q is not EMPTY */
                a) rear.link=new          /* 1 */
                b) new.link=NULL          /* 2 */
                c) rear=new               /* 3 */
                d) new.data=item
        iii) end if
4. end if
```

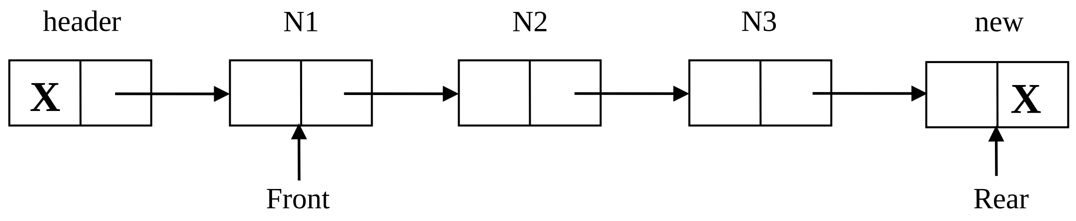
End_Enqueue_LL

While performing ENQUEUE operation two situations are occur.

1. if queue is empty, then newly inserting element becomes first node and last node in the queue. So Front and Rear points to first node in the list.
2. If Queue is not empty, then newly inserting node is inserted at last.



Before ENQUEUE



After ENQUEUE

1. Previous last node link part is replaced with address of new node.
2. Link part of new node is replaced with NULL, because new nodes becomes the last node.
3. Rear is points to last node in the list. i.e. newly inserted node in the list.

Algorithm Dequeue_LL( )

Input: Queue with some elements

Output: Element is deleted at front end if queue is not empty.

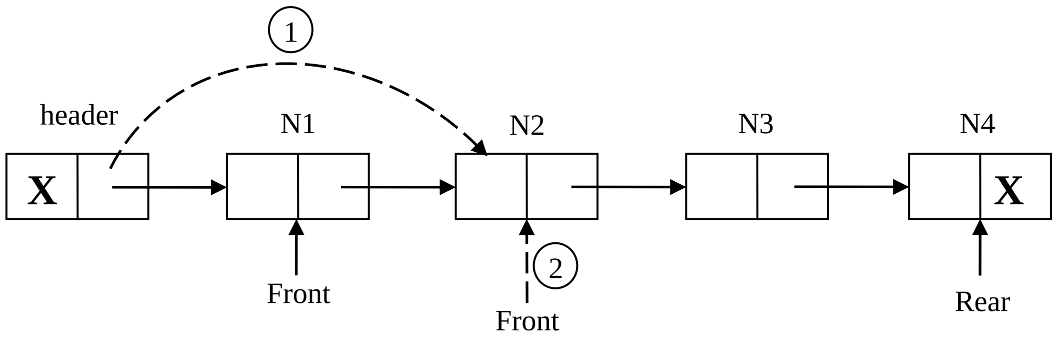
```
    1.if(front= =NULL and rear = =NULL)
        a) print(queue is empty, not possible to perform dequeue operation)
    2. else
        i) if(front = = rear)            /* Q has only one element */
            a) header.link = NULL
            b) item=front.data
            c) front=NULL
            d) rear=NULL
        b) else                               /* Q has more than one element
    */
```

```
                a) header.link = front.link            /* 1 */
                b) item=front.data
                c) free(front)
                d)front=header.link                    /* 2 */
            c) end if
            d) print(deleted element is item)
        3. end if
End_Dequeue_LL
```

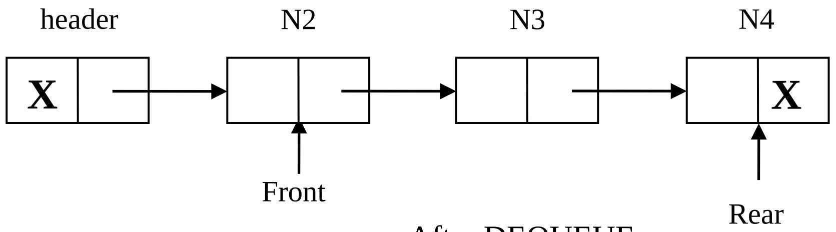
While performing DEQUEUE operation two situations are occur.

1. if queue has only one element, then after deletion of that element Queue becomes empty. So Front and Rear points to NULL.

2. If Queue has more than one element, then first node is deleted at Front end.



Before DEQUEUE



After DEQUEUE

1. Link part of the header node is replaced with address of second node. i.e. address of second node is available in link part of first node.

2. Front is set to first node in the list.

```
Algorithm Queue_Status_LL
Input: Queue with some elements
Output: Status of the queue. i.e. Q is empty or not, Q is full or not, Element at
front end and rear end.
    1. if(front = = NULL and rear = = NULL)
        a) print(Q is empty)
    2. else if(front = = rear)
        a) print(Q has only one item)
    3. else
        a) print(element at front end is front.data)
        b) print(element at rear end is rear.data)
    4. end if
End Queue_Status_LL
```

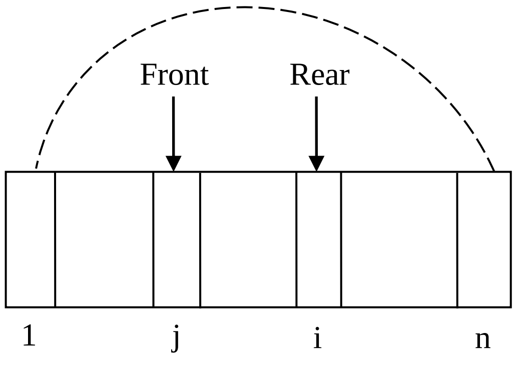
Various Queue Structures

1. **Circular Queues**
2. **Deque or Double-Ended Queue**
3. **Priority Queue**

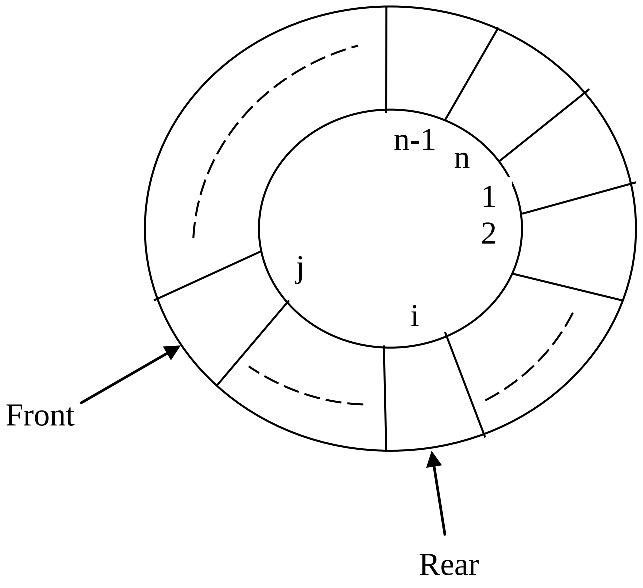
1. Circular Queues

Physically a circular array is same as ordinary arra, say a[i-N], but logically it implements that a[1] comes after a[N] or a[N] comes after a[1].

The following figure shows the physical and logical representation for circular



Circular Array (Physical)



Circular Queue (Logical)

array

Logical and physical view of a Circular Queue

- Here both Front and Rear pointers are move in clockwise direction. This is controlled by the MOD operation.
- For e.g. if the current pointer is at i, then shift next location will be
  (i mod LENTH) +1, 1<= i <= Length

Circular Queue empty condition is

Front = 0 and Rear = 0

Circular Queue is full

Front = = (Rear Mod Length) + 1

Advantages

1. It takes up less memory than the linear queue.

2. A new item can be inserted in the location from where a previous item is deleted.

3. Infinite number of elements can be added continuously but deletion must be used.

Algorithm CQ_Enqueue(item)

Input: item is new item insert in to Circular queue at rear end.

Output: Insertion of new item Circular queue at rear end if queue is not full.

```
    1. next = (rear % N) = 1
    2. if( front == next)
        a)print(Circular queue is full, not possible for enqueue operation)
    3. else
        i) if(front = = 0 and rear = = 0)  /* CQ is Empty */
            a) rear=( rear % N) = 1
            b) CQ[rear]=item
            c) front=1
        ii) else
            a) rear=( rear % N) = 1
            b) CQ[rear]=item
        iii) end if
    4.end if
```

End CQ_Enqueue

Algorithm CQ_Dequeue( )

Input: Circular Queue with some elements.

Output: Element is deleted from circular queue at front end if circular queue is not empty.

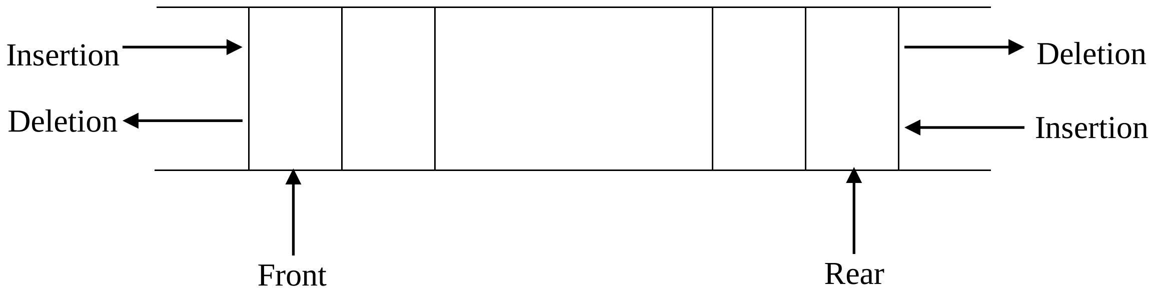
```
    1. if(front = = 0 and rear = = 0)
        a) print(CQ is empty, not possible for dequeue operation)
    2. else
        i) if(front = = rear)          /* Q has only one element */
            a) item=CQ[front]
            b) front=0
            c) rear=0
        ii) else
            a)item=CQ[front]
```

```
                b)front=(front % N)+1
            iii) end if
            iv) print(deleted item is item)
        3. end if
End CQ_Dequeue
```

2. <u>Deque</u>

In Deque, both ENQUEUE (insertion) and DEQUEUE (deletion) operations can be made either of the ends.

Deque is organized from Double Ended Queue.



A DEQue structure

Here DEQue structure is general representation of stack and Queue. In other words, a DEQue can be used as stack and Queue.

DeQue can be represented in two ways.

1. Using Double Linked List
2. Using a Circular Queue

Here Circular array is popular representation of DEQue.

On DEQue, the following four operations can be performed.

1. PUSHDQ(item)	:	To insert item at FRONT end of DEQue.
2. POPDQ( )	:	To delete the FRONT end item from DEQue.
3. INJECT(item)	:	To insert item at REAR end of DEQue.
4. EJECT( )	:	To delete the REAR end item from DEQue.

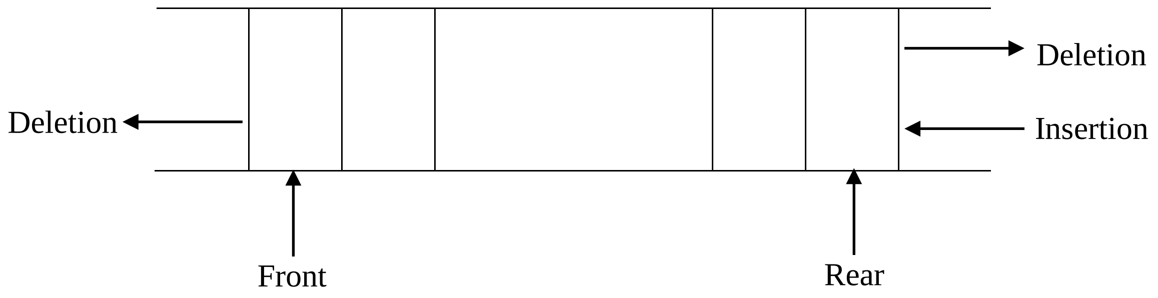
There are two variations of DEQue known as

Input restricted DEQue

Output restricted DEQue

Input restricted DEQue

Here DEQue allows insertion at one end (say REAR end) only, but allows deletion at both ends.

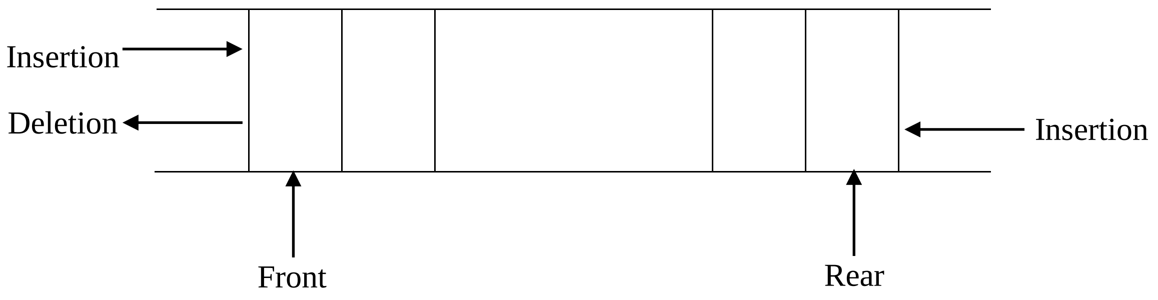


Input restricted DEQue

Output restricted DEQue

Here DEQue allows deletion at one end (say FRONT end) only, but allows insertion at both ends.

DEQue is organized from Double Ended Queue.



Output restricted DEQue

3.Priority Queue

Priority Queue is an extension of queue with following properties.

1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have the same priority, they are served according to their order in the queue.

Queue using stack

To perform Queue operations using stack, we require two stacks named as stack1 and stack2.

The Queue operations using stack can perform in two ways.

1. Enqueue operation is cost effective
2. Dequeue operation is cost effective

1. Enqueue operation is cost effective

```
Algorithm Enqueue_stack(item)
    1. While stack1 is not empty, PUSH every element from stack1 to stack2.
    2. PUSH item in to stack1.
    3. While stack2 is not empty, PUSH every element from stack2 to stack1.
End Enqueue_stack
Algorithm Dequeue_stack( )
    1. If stack1 is empty then error occurs. i.e. queue is empty.
    2. Else POP an item from stack1.
End Dequeue_stack
```

2. Dequeue operation is cost effective

```
Algorithm Enqueue_stack(item)
    1. PUSH item into stack1
End Enqueue_stack
Algorithm Dequeue_stack( )
    1. If stack1 and stack2 are empty then error occur. i.e. Queue is empty.
    2. Else if stack2 is empty
            a) While stack1 is not empty, PUSH ever element from stack1 to stack2.
            b) POP element from stack2.
            c) While stack2 is not empty, PUSH ever element from stack2 to stack1
    3. End if
End Dequeue_stack.
```

****

Unit –IV

TREES

Objective:

- To impart knowledge of linear and non-linear data structures.

Syllabus:

Binary Trees: Basic tree concepts, properties, representation of binary trees using arrays and linked list, binary tree traversals,Threaded Binary Tree.

Binary Search Trees: Basic concepts, BST operations: search, insertion, deletion and traversals, creation of binary search tree from in-order and pre (post) order traversals.

Learning Outcomes:

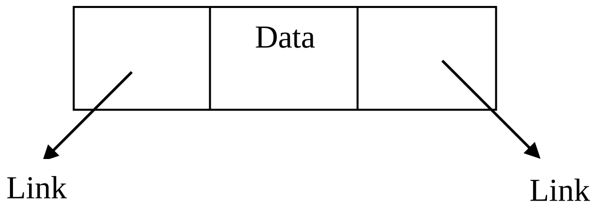
At the end of the unit student will be able to:

1. represent Binary Trees using Arrays and Linked Lists.
2. Implement operations on Binary Search Trees.
3. construct Binary Search Trees from its Traversals.

Learning Material

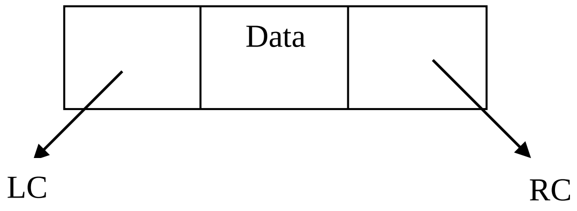
Basic Terminology:

1. Node: It is a main component of tree. It stores the actual data and links to other nodes.



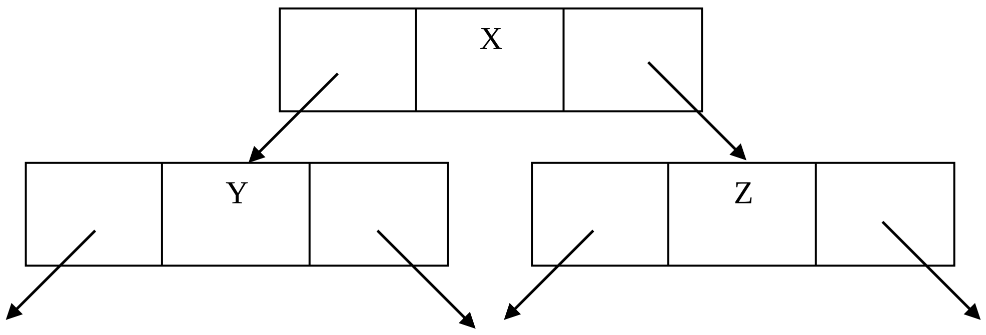
Structure of a node in Tree

2. Link / Edge / Branch: Link is point to other nodes in a tree.



Here *LC* Points To **Left Child** and *RC* Points To **Right Child**.

3. Parent Node: The Immediate Predecessor of a Node is called as Parent Node.

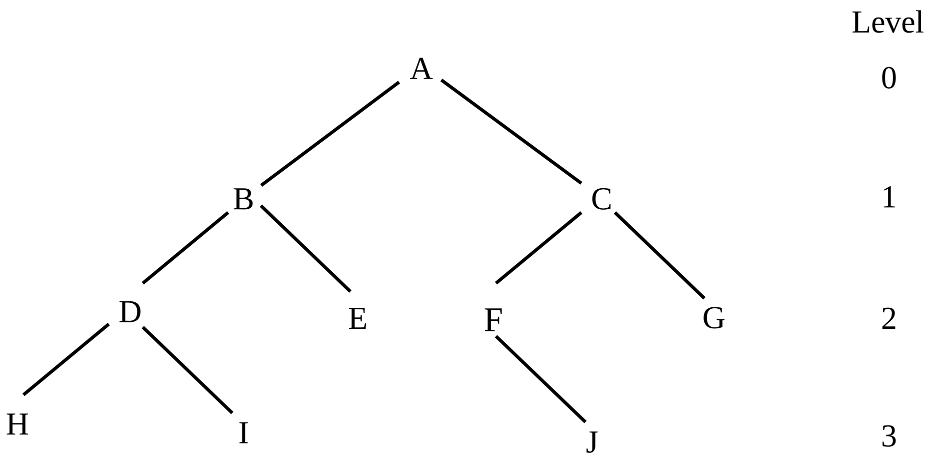


Here X is Parent Node to Node Y and Z.

4. Child Node: The Immediate Successor of a Node is called as Child Node.

In the above diagram Node Y and Z are child nodes to node X.

5. Root Node: Which is a specially designated node, and does not have any parent node.



In the above diagram node A is a Root Node.

6. Leaf node: The node which does not have any child nodes is called leaf node.

- In the above diagram node H, I, E, J, G are Leaf nodes.

7. Level: It is the rank of the hierarchy and the Root node is present at level **0**. If a node is present at level ***l*** then its parent node will be at the level ***l*-1**and child nodes will present at level ***l*+1**.

8. Height / Depth: The number of nodes in the longest path from Root node to the Leaf node is called the height of a tree.

- Height of above tree is 4.
- Height of a tree can be easily obtained as l_{max} + 1. Where l_{max} is the maximum level of a tree.
- In the above example l_{max}= 3. So height = 3 + 1 = 4

9. Siblings: The nodes which have same parent node are called as siblings.

- In the above example nodes B and C are siblings, nodes D and E are siblings , nodes F and G are siblings , nodes H and I are siblings.

10. Degree / Arity: Maximum number of child nodes possible for anode is called as degree of a node.

TREE:

Tree is a nonlinear data structure.

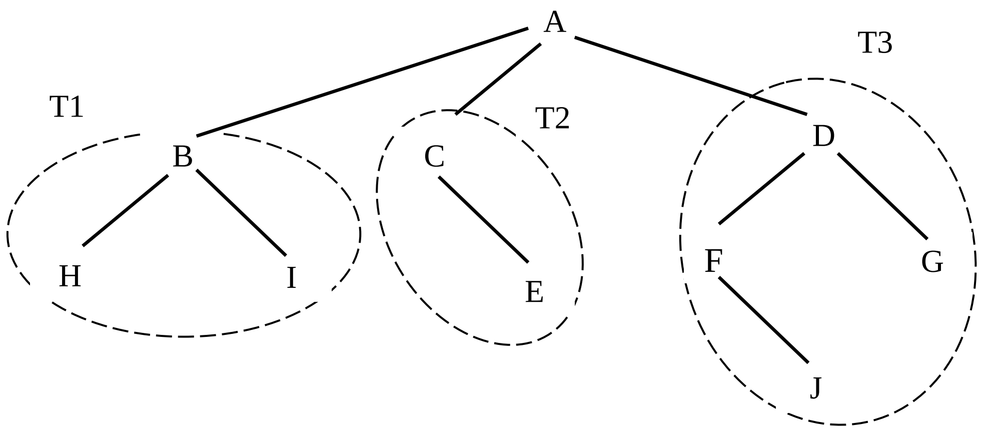
Definition

A tree T is a finite set of one or more nodes such that:

(i) There is a special node called as *root* node.

(ii) The remaining nodes are partitioned into ***n*** disjoint sets T_1, T_2, T_3. . . $T_{n.}$ where n>0.Where each disjoint set is a tree.

T_1, T_2, T_3. . .T_n are called as *sub trees*.



A sample Tree

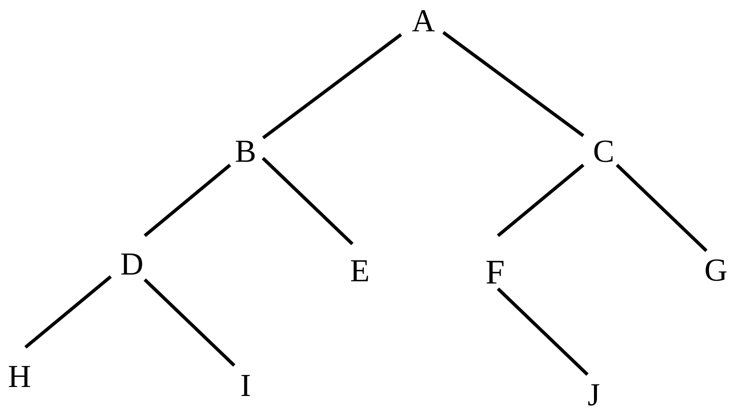
BINARY TREE: Is a special form of a tree.

Definition:

A binary tree is T is a finite set of nodes such that,

(i) T is empty called as empty binary tree.

(ii) T has specially designed node called as Root Node and remaining node of binary tree are partitioned into 2 disjoint sets. One is Left sub tree and another one is Right sub tree.



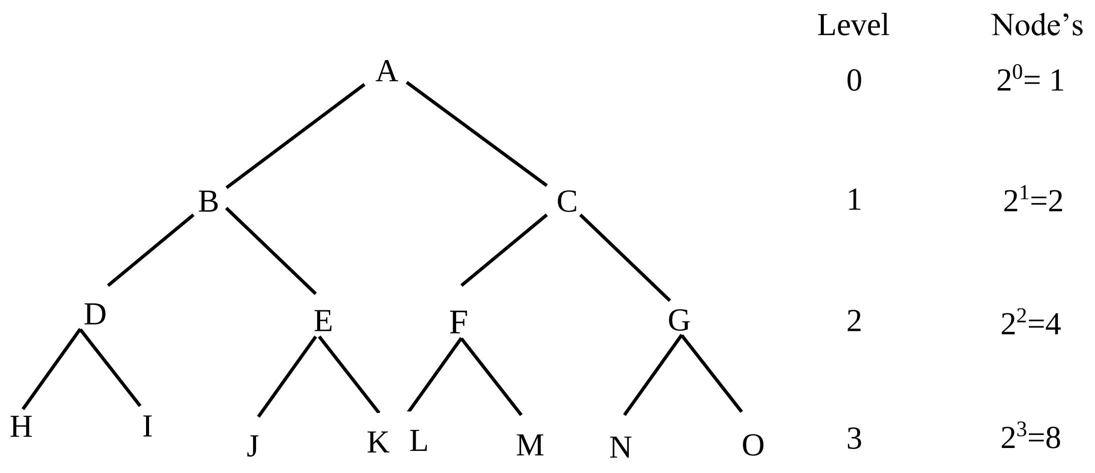
A sample Binary Tree

TWO SPECIAL CASES OF BINAERY TREE

1. Full binary tree
2. Complete binary tree

1. Full Binary Tree: a binary tree is said to be full binary tree, if each level has maximum number of possible nodes.

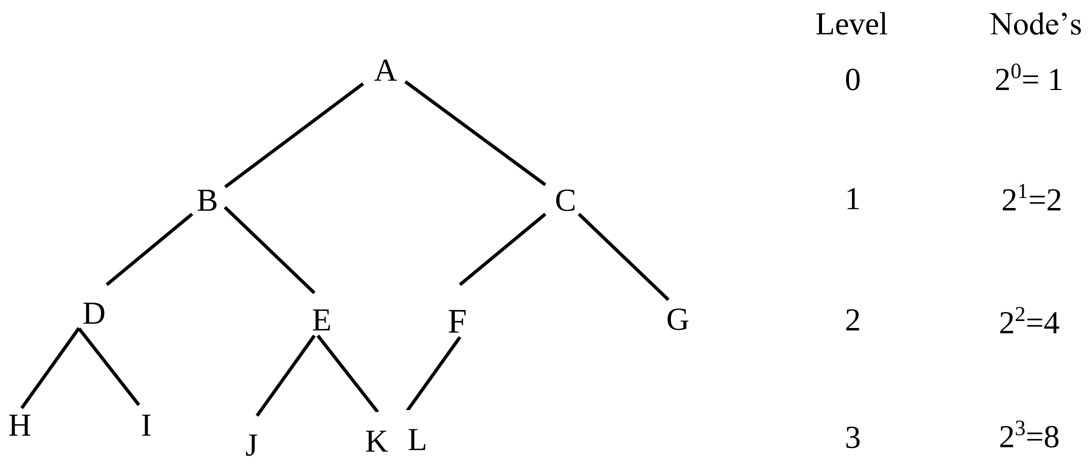
Eg:



A Full Binary Tree of height 4

2. Complete Binary Tree: A binary tree is said to be complete binary tree, if all levels except the last level has maximum number of possible nodes, last level nodes are appeared as far left as possible.

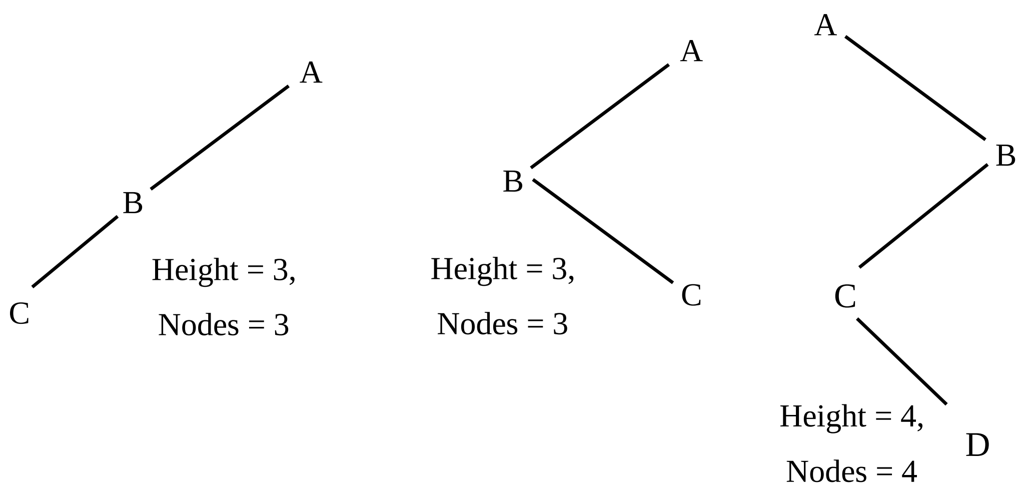
Eg:



A Complete Binary Tree of height 4

Properties of Binary Trees

1. In any binary tree, maximum number of nodes on level l is 2^l (where l>= 0).
2. Maximum number of nodes possible in a binary tree of height h is 2^h-1.
3. Minimum number of nodes possible in a binary tree of height h is h.



- Whenever every parent node has only one child, such kind of binary trees are called as ***Skew binary trees***.

4. For any non-empty binary tree, if ***n*** is the number of nodes and ***e*** is the number of edges, then ***n = e + 1***.
   i.e. number of nodes = number of edges +1.
5. For any non-empty binary tree T, if $\boldsymbol{n_0}$ is the number of leaf nodes (degree = 0) and $\mathbf{n_2}$ is the number of intermediate nodes (degree = 2) then $\mathbf{n_0 = n_2 + 1}$.
   i.e. number of leaf nodes = number of non-leaf nodes + 1.
6. The height of a complete binary tree with **n** nodes is $\lceil \log_2(n+1) \rceil$
7. Total number of binary trees possible with **n** number of nodes is $\frac{1}{n+1} 2n_{c_n}$
8. The maximum and minimum size that an array may require to store a binary tree with ***n*** number of nodes are:

   Maximum size =$2^n - 1$

   Minimum size =$2^{\lceil \log_2(n+1) \rceil} - 1$
9. In a linked list representation of binary tree, if there are ***n*** number of nodes, then the number of *NULL* link are ***λ = n +1.***

Representation of Binary Tree:

Binary tree can be represented in two ways.

1. Linear (or) Sequential representation using arrays.
2. Linked List representation using pointers.

1. Linear (or) Sequential representation using arrays

- In this representation, a block of memory for an array is to be allocated before going to store the actual tree in it.
- Once the memory is allocated, the size of the tree will be restricted to memory allocated.
- In this representation, the nodes are stored level by level starting from zero level, where only *ROOT* node is present.
- The *ROOT* node is stored in the first memory location. i.e. first element in the array.

The following rules are used to decide the location on any node of tree in the array. (Assume the array index start from 1)

1. The ROOT node is at index**1**.
2. For any node with index i, $1 \leq i \leq n$.

   (a) **Parent (i)** = $\left\lfloor \frac{i}{2} \right\rfloor$

   For the node when i = 1, there is ***no parent node***.

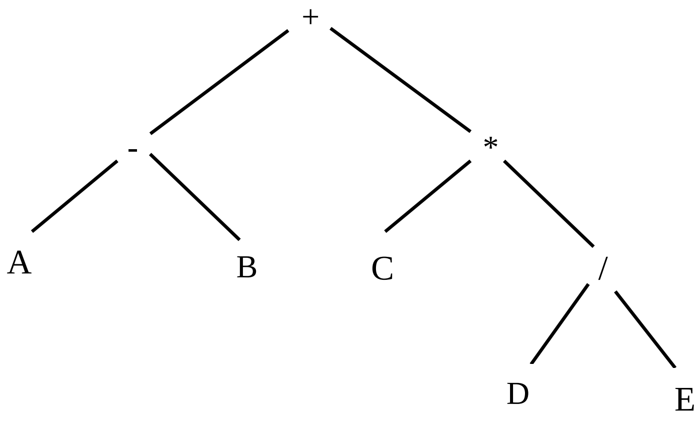
   (b) **LCHILD (i)** = 2 * i

   If 2 * i > n then **i** has ***no left child***.

   (c) **RCHILD (i)** = 2*i +1

   If 2*i + 1 > n then **i** has ***no right child***.

Eg. (A-B) + C * (D / E)



1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

+	-	*	A	B	C	/							D	E

Array representation of above binary tree.

Advantages of Linear representation of Binary Tree

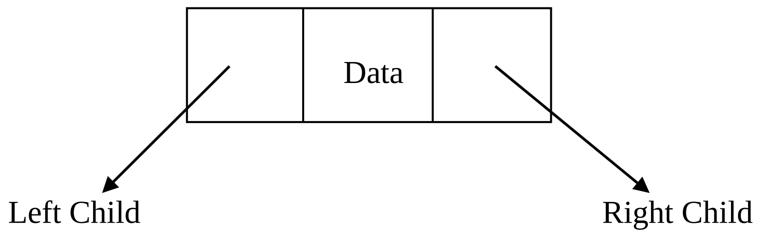
1. Any node can be accessed from any other node by calculating the index and this is efficient from execution point of view.
2. Here only data is stored without any pointers to their successor (or) predecessor.

Disadvantages of Linear representation of Binary Tree

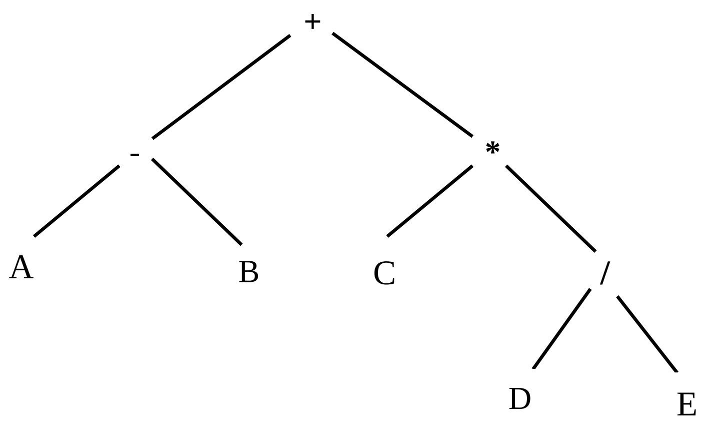
1. Other than full binary tree, majority of entries may be empty.
2. It allows only static memory allocation.

2. Linked List representation of Binary Tree using pointers

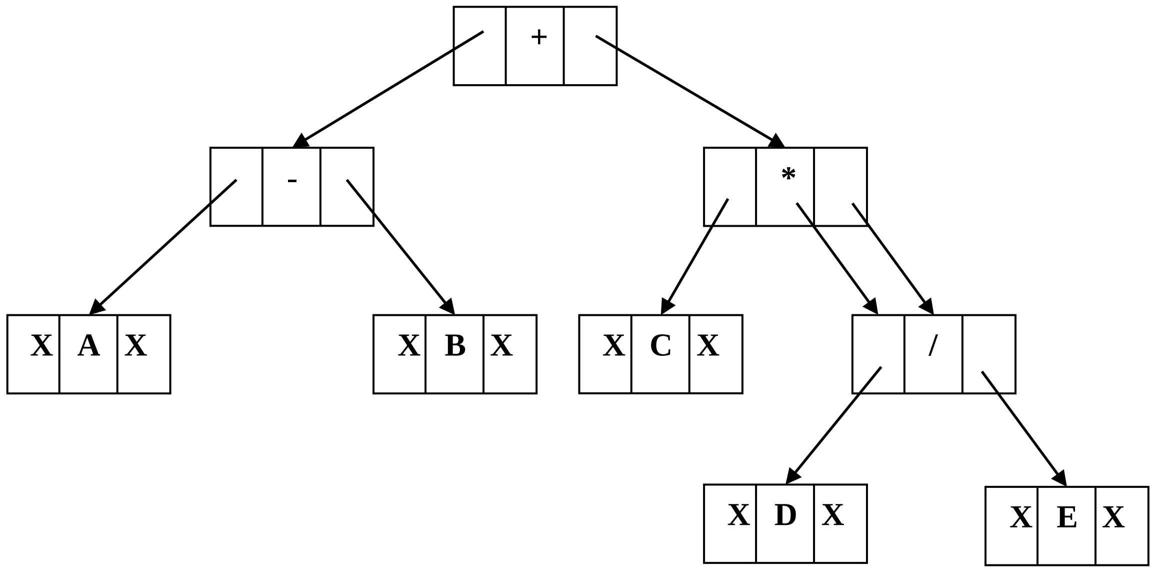
- Linked list representation of assumes structure of a node as shown in the following figure.



- With linked list representation, if one knows the address of *ROOT* node, then any other node can be accessed.



Binary Tree



Linked List representation of Binary Tree

Advantages of Linked List representation of Binary Tree

1. It allows dynamic memory allocation.
2. We can overcome the drawbacks of linear representation.

Disadvantages of Linked List representation of Binary Tree

1. It requires more memory than linear representation. i.e. Linked list representation requires extra memory to maintain pointers.

Binary Tree Traversals

Traversal operation is used to visit each node present in binary tree exactly once.

A Binary tree can be traversed in 3 ways.

1. Preorder traversal
2. Inorder traversal
3. Postorder traversal

1. Preorder traversal

Here first **ROOT** node is visited, then **LEFT** sub tree is visited in *Preorder* fashion and then **RIGHT** sub tree is visited in *Preorder* fashion.

i.e. *ROOT*, LEFT, RIGHT (or) ROOT, RIGHT, LEFT

2. Inorder traversal

Here first **LEFT** subtree is visited in *inorder* fashion, then **ROOT** node is visited and then **RIGHT** sub tree is visited in *inorder* fashion.

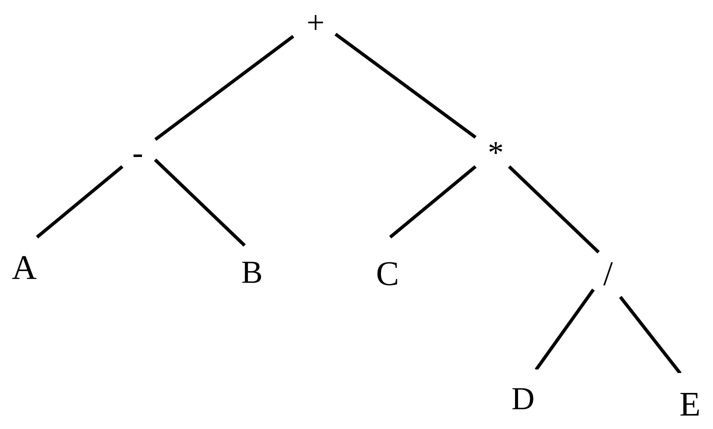
i.e. LEFT, *ROOT*, RIGHT (or) RIGHT, *ROOT*, LEFT

3. Postorder traversal

Here first **LEFT** subtree is visited in *postorder* fashion, then **RIGHT** sub tree is visited in *postorder* fashion and then **ROOT** node is visited.

i.e. LEFT, RIGHT, *ROOT* (or) RIGHT, LEFT, *ROOT*

Eg:



Preorder traversal	:	+ - A B * C / D E
Inorder traversal	:	A – B + C * D / E
Postorder traversal	:	A B – C D E / * +

Recursive Binary Tree Traversals

Algorithm preorder(ptr)

Input: Binary Tree with some nodes.

Output: *preorder* traversal of given Binary Tree.

```
1. if(ptr != NULL)
        a) print(ptr.data)
        b) preorder(ptr.lchild)
        c) preorder(ptr.rchild)
2. end if
```

End preorder

Algorithm inorder(ptr)

Input: Binary Tree with some nodes.

Output: *inorder* traversal of given Binary Tree.

```
1. if(ptr != NULL)
```

```
        a) inorder(ptr.lchild)
        b) print(ptr.data)
        c) inorder(ptr.rchild)
    2. end if
End inorder
```

```
Algorithm postorder(ptr)
Input: Binary Tree with some nodes.
Output: postorder traversal of given Binary Tree.
    1. if(ptr != NULL)
        a) postorder(ptr.lchild)
        b) postorder(ptr.rchild)
        c) print(ptr.data)
    2. end if
End postorder
```

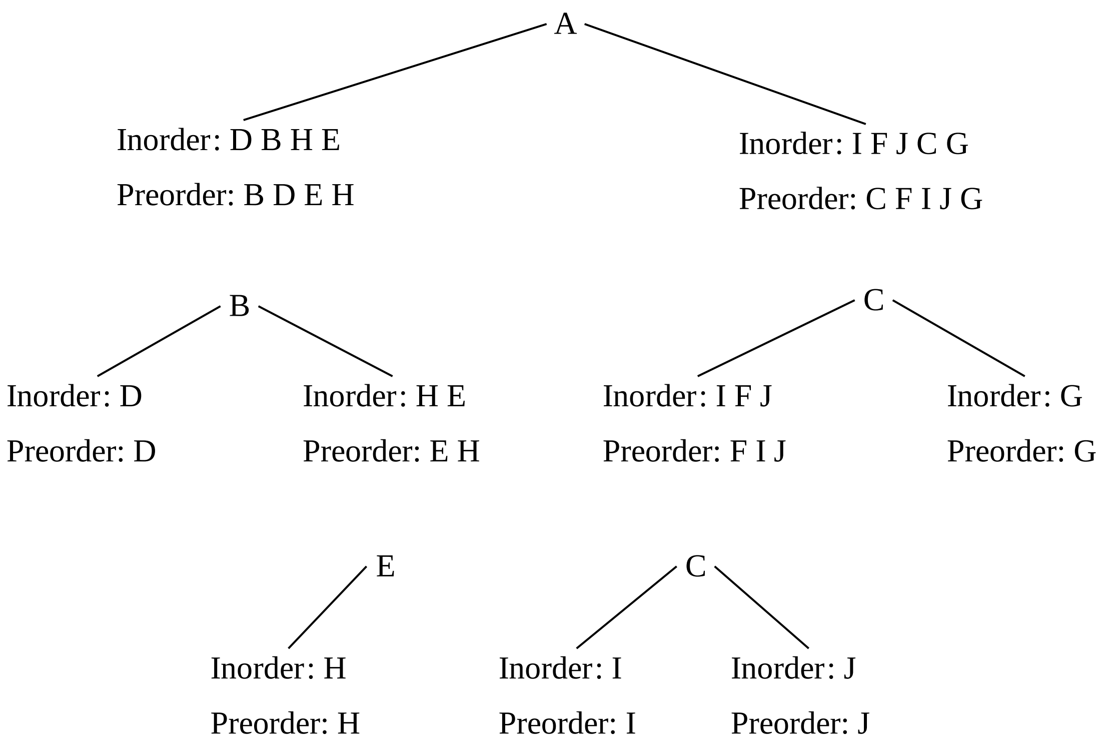
Creation of Binary Tree from its Tree traversals

- A binary tree can be constructed from its traversals.
- If the *Preorder* traversals is given, then the ***first node*** is *ROOT* node and *Postorder* traversal is given then ***last node*** is the *ROOT* node.
- For construction of a binary tree from its traversals, two traversals are essentials. Out of which one should be *inorder* traversal and another one is either *preorder* (or) *postorder* traversal.
- If preorder and post order is given to construct a binary tree, then binary tree can't be obtain *uniquely*.

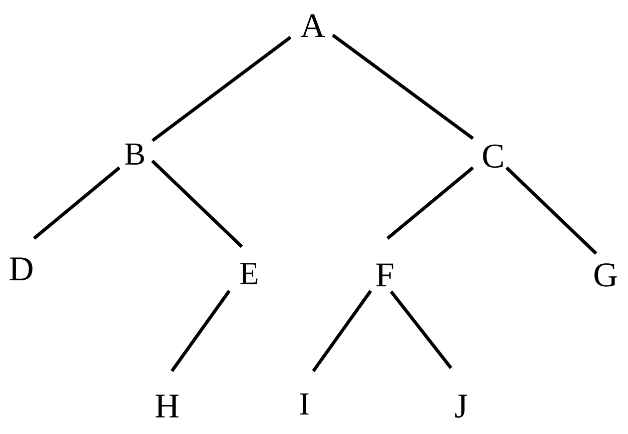
Eg.1.

Inorder:	D	B	H	E	A	I	F	J	C	G
Preorder:	A	B	D	E	H	C	F	I	J	G

1. From the preorder traversal, *A* is the *ROOT* node.
2. In the inorder traversal, all the nodes which are LEFT side of **A** belongs to LEFT sub tree and those node which are RIGHT side of **A** belongs to RIGHT sub tree.
3. Now the problem is reduced to two sub trees and same procedure can be applied repeatedly.

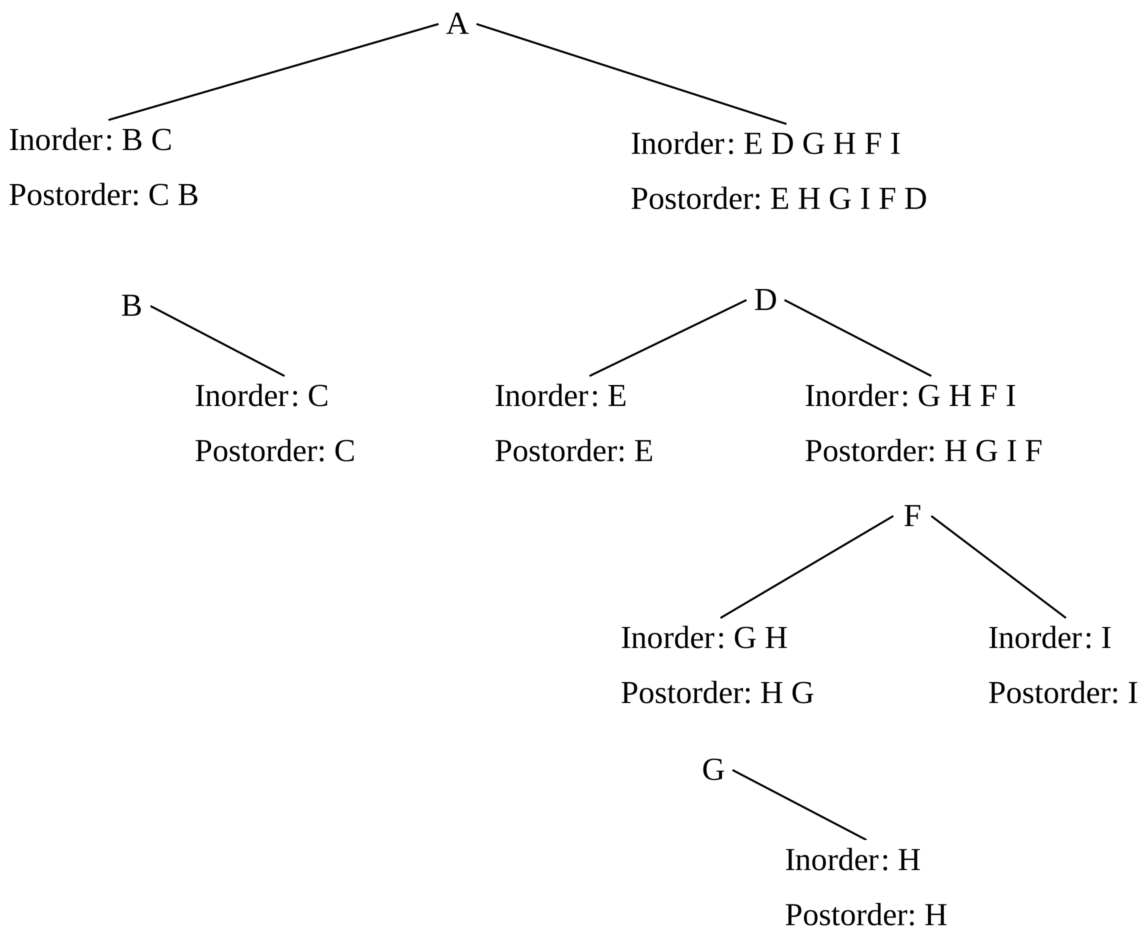


Final Binary tree from the Inorder and Preorder as follows:

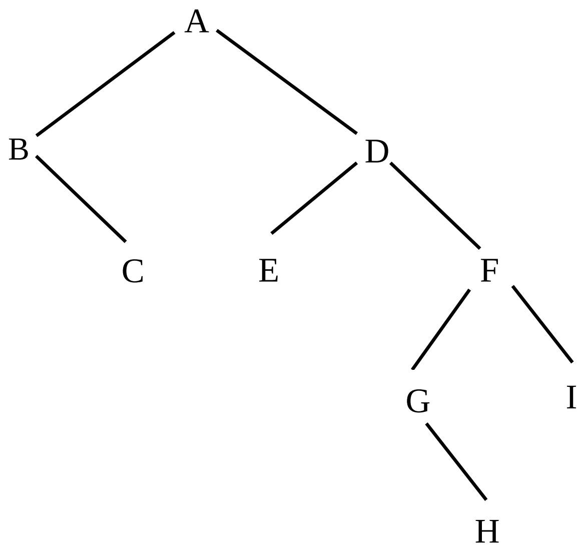


Eg. 2.

Inorder:	B	C	A	E	D	G	H	F	I
Postorder:	C	B	E	H	G	I	F	D	A



Final Binary tree from the Inorder and Postorder as follows:



Types of Binary Tress

1. Expression Trees
2. Binary Search Trees
3. Threaded Binary Trees
4. Heap Tree
5. Height Balanced Binary Trees
6. Decision Trees
7. Huffman Tree

Threaded Binary Trees: When a binary tree is represented using linked list ,if any node is not having a child we use a NULL pointer. These special pointers are threaded and the binary tree having such pointers is called a threaded binary tree.

Advantages:

Non-recursive pre-order, in-order and post-order traversal can be implemented without a stack.

Disadvantages:

1. Insertion and deletion operation becomes more difficult.
2. Tree traversal algorithm becomes difficult.
3. Memory required to store a node increases. Each node has to store the information whether the links is normal links or threaded links.

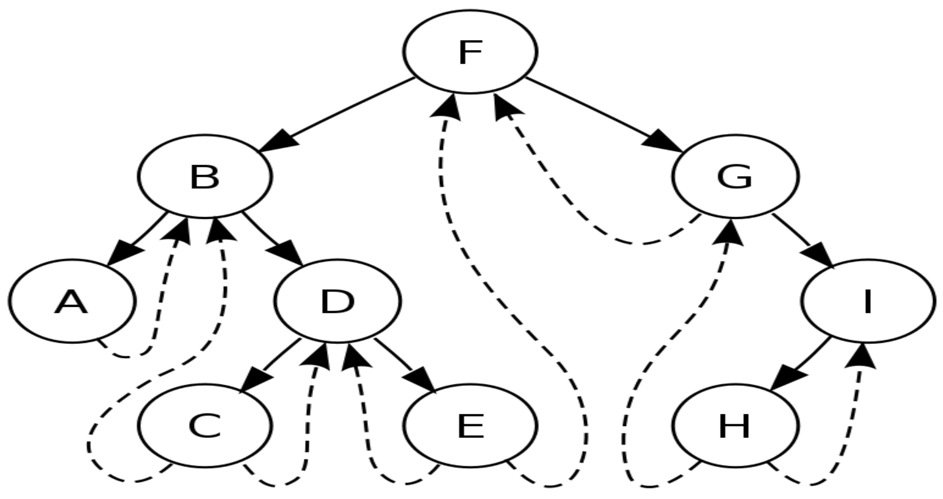


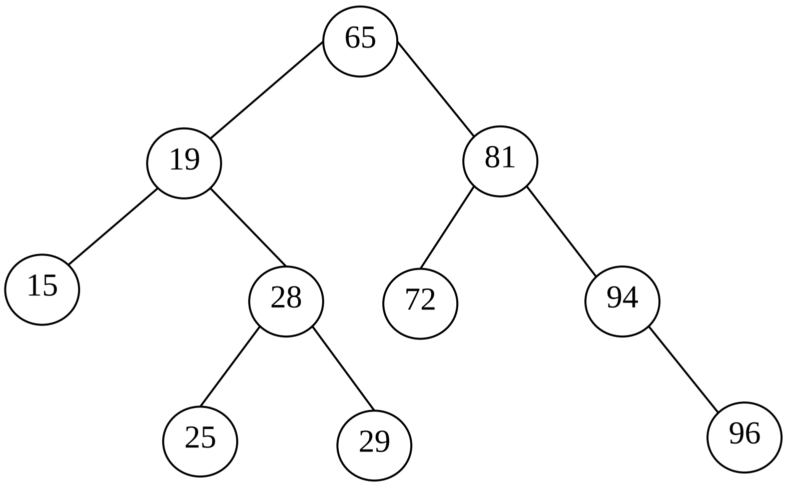
Fig:Representation of Threaded Binary Tree

Binary Search Tree

Definition:

A binary tree T is termed binary search tree (or binary sorted tree) if each node N of T satisfies the following property:

The value at N is greater than every value in the left sub-tree of N and is less than every value in the right sub-tree of N.



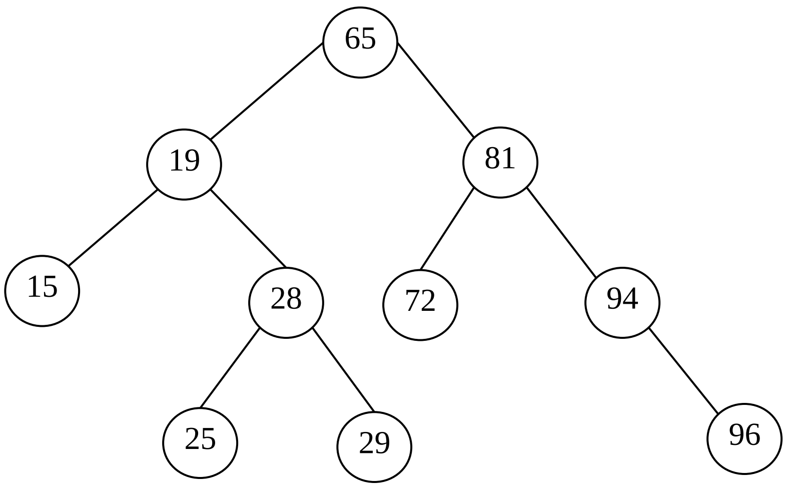
Binary Search Tree with numeric data

Operations on Binary Search Trees:

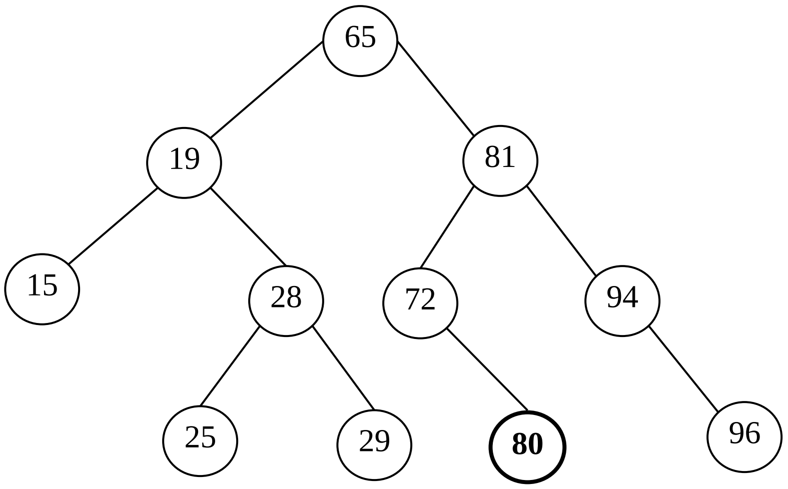
1. Inserting data
2. Deleting Data
3. Traversing the Tree

1. Inserting data into a binary search tree

To insert a node with data say item into a binary search tree, first binary search tree is searched starting from ROOT node for the item. If the item is found, do nothing. Otherwise item is to be inserted as LEAF node where search is halt.



Inserting 80 into the above figure



After insertion of new node 80

Algorithm BST_Insert(item)

Input: *item* is data part of new node to be insert into BST.

Output: BST with new node has data part *item*.

```
1. ptr = Root
2. flag = 0
3. while(ptr != NULL && flag == 0 )
    a) if( item == ptr.data)
        i) flag = 1
        ii) print "item already exist"
    b) else if( item < ptr.data)
        i) ptr1 = ptr
```

```
            ii) ptr = ptr.LCHILD
        c) else if( item > ptr.data)
            i) ptr1 = ptr
            ii) ptr = ptr.RCHILD
        d) end if
    4. end loop
    5. if(ptr == NULL)
        a) new = getnewnode()
        b) new.data = item
        c) new.lchild = NULL
        d) new.rchild = NULL
        e) if(root.data == NULL)
            i) root = new
                A) print "New node inserted successfully  as ROOT Node"
        f) else if( item < ptr1.data)       /* inserting new node as left child to its parent*/
            i) ptr1.lchild = new
            ii) print "New Node is inserted successfully as LEFT child"
        g) else                             /* inserting new node as right child to its parent*/
            i) ptr1.rchild = new;
            ii) print "New Node is inserted successfully as Right Child"
        h) end if
    6. end if
End BST_Insert
```

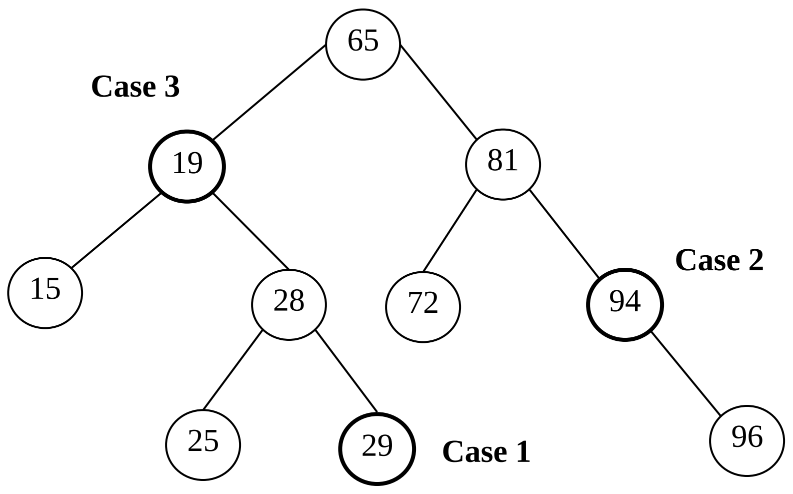
2. Deleting data from a Binary Search Tree

- If ITEM is the information given which is to be deleted from a BST. Let N be the node which contains the information ITEM. Assume PARENT(N) denotes the parent node of N and SUCC(N) denotes the inorder successor of N.
- Then the deletion of the node N depends on the number of its children. Hence, 3 cases may arise and they are:

  Case 1: N is the leaf node. i.e. no child nodes.

  Case 2: N has exactly one child.

  Case 3: N has two children.

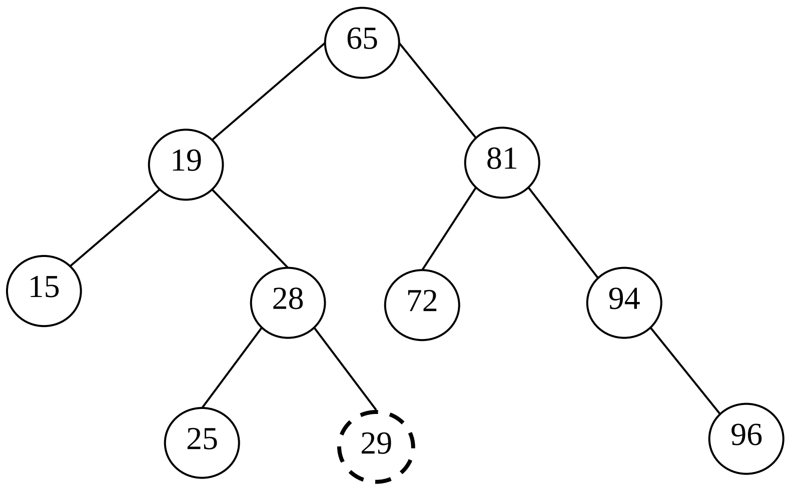


Three cases from deleting a node from BST

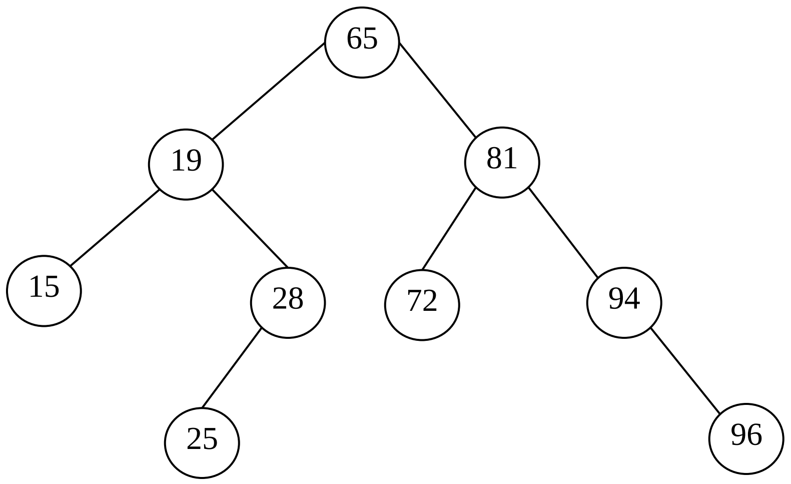
In the above Binary Search Tree (BST) deletion of a node *29* leads to *Case 1*. Here node *29* is a leaf node i.e. which does not have any child nodes. Deletion of node *94* leads to *Case 2*. Here node *94* has only one child node. Deletion of a node *19* leads to *Case 3*. Here node 19 has two child nodes.

Case 1:

N is a leaf node, this node is to be delete. N is deleted from T by simply setting the pointer of N in the parent node PARENT(N) by **NULL** value.



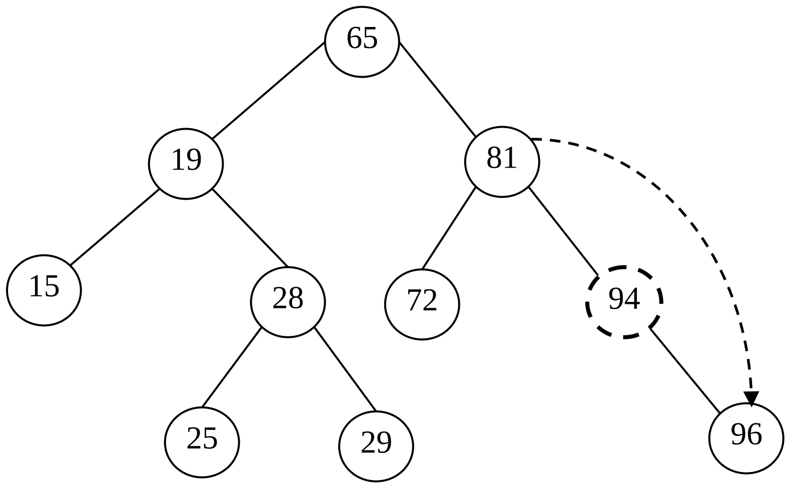
Deletion of node 29



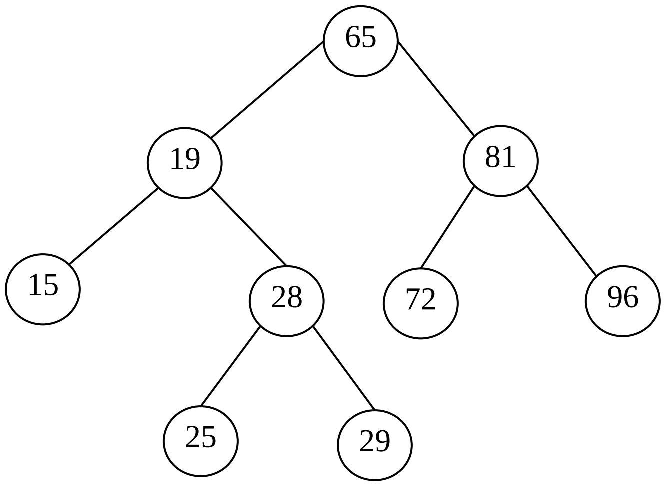
After deletion of node 29 form given BST

Case 2: N has exactly one child.

N is deleted from T by simply replacing the pointer of N in PARENT(N) by the pointer of the only child of N.



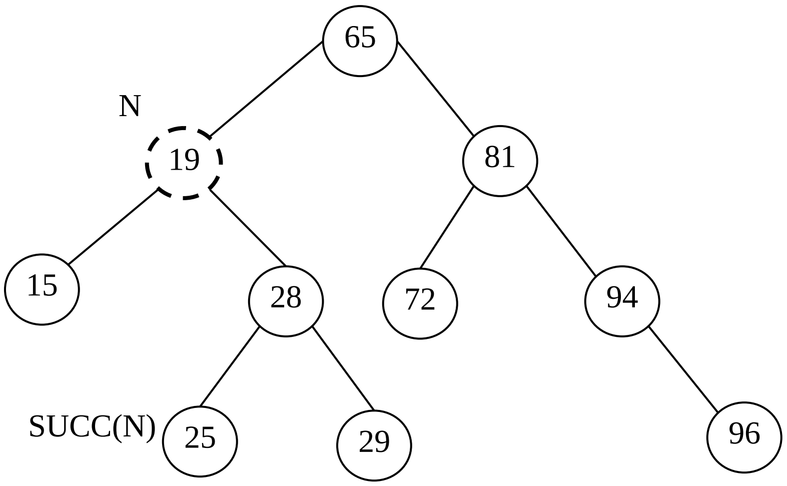
Deletion of node 94



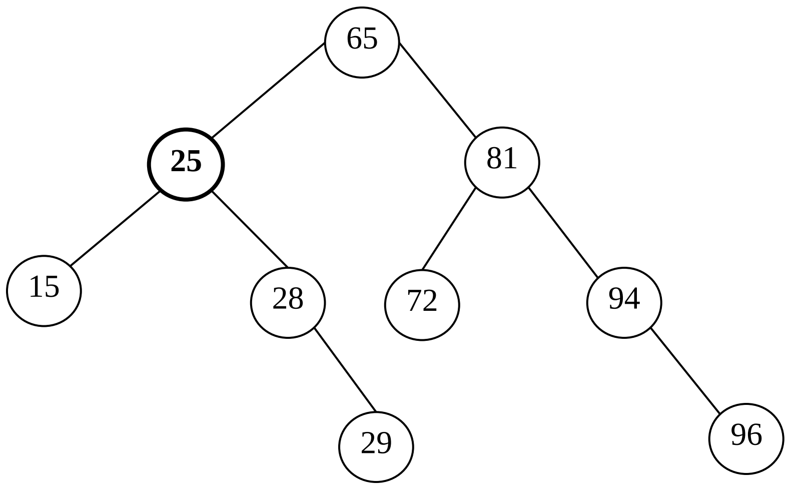
After deletion of node 94 form given BST

Case 3: N has two child nodes.

N is deleted from T by first deleting SUCC(N) from T(by using Case 1 or Case 2 it can be verified that SUCC(N) never has a left child) and then replacing the data content in node N by the data content in node SUCC(N).



Deletion of node 19



After deletion of node 19 form given BST

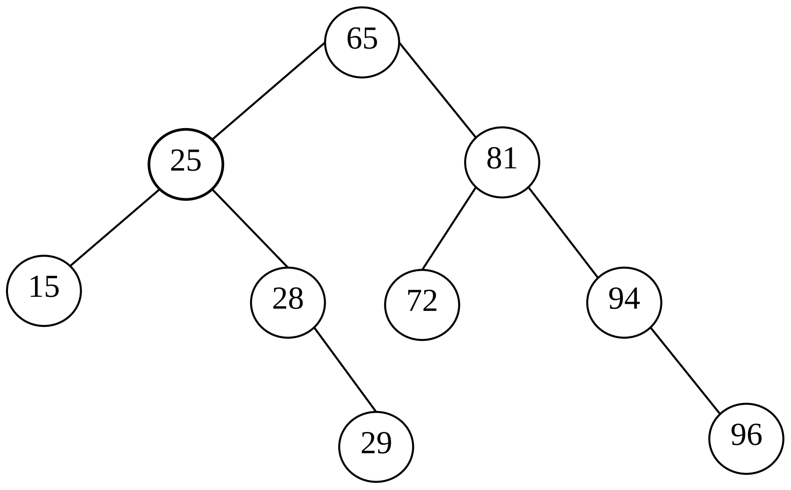
3. Binary Search Tree Traversals

Traversal operation is used to visit each node present in binary search tree exactly once.

A Binary search tree can be traversed in 3 ways.

1. Preorder traversal
2. Inorder traversal
3. Postorder traversal

Example:



Inorder traversal for above BST is: 15, 25, 28, 29, 65, 72, 81, 94, 96

(Inorder traversal of BST always gives Ascending order of elements)

Preorder traversal for above BST is: 65, 25, 15, 28, 29, 81, 72, 94, 96

Postorder traversal for above BST is: 15, 29, 28, 25, 72, 96, 94, 81, 65

Recursive Binary Search Tree Traversals

Same as Recursive implementation of Binary Tree Traversals

Algorithm preorder(ptr)

Input: Binary Tree with some nodes.

Output: *preorder* traversal of given Binary Tree.

```
    1. if(ptr != NULL)
            a) print(ptr.data)
            b) preorder(ptr.lchild)
            c) preorder(ptr.rchild)
    2. end if
```

End preorder

Algorithm inorder(ptr)

Input: Binary Tree with some nodes.

Output: *inorder* traversal of given Binary Tree.

```
    1. if(ptr != NULL)
            a) inorder(ptr.lchild)
            b) print(ptr.data)
            c) inorder(ptr.rchild)
    2. end if
```

End inorder

Algorithm postorder(ptr)

Input: Binary Tree with some nodes.

Output: *postorder* traversal of given Binary Tree.

```
    1. if(ptr != NULL)
            a) postorder(ptr.lchild)
            b) postorder(ptr.rchild)
            c) print(ptr.data)
    2. end if
```

End postorder

Creation of Binary Search Tree from its Tree traversals

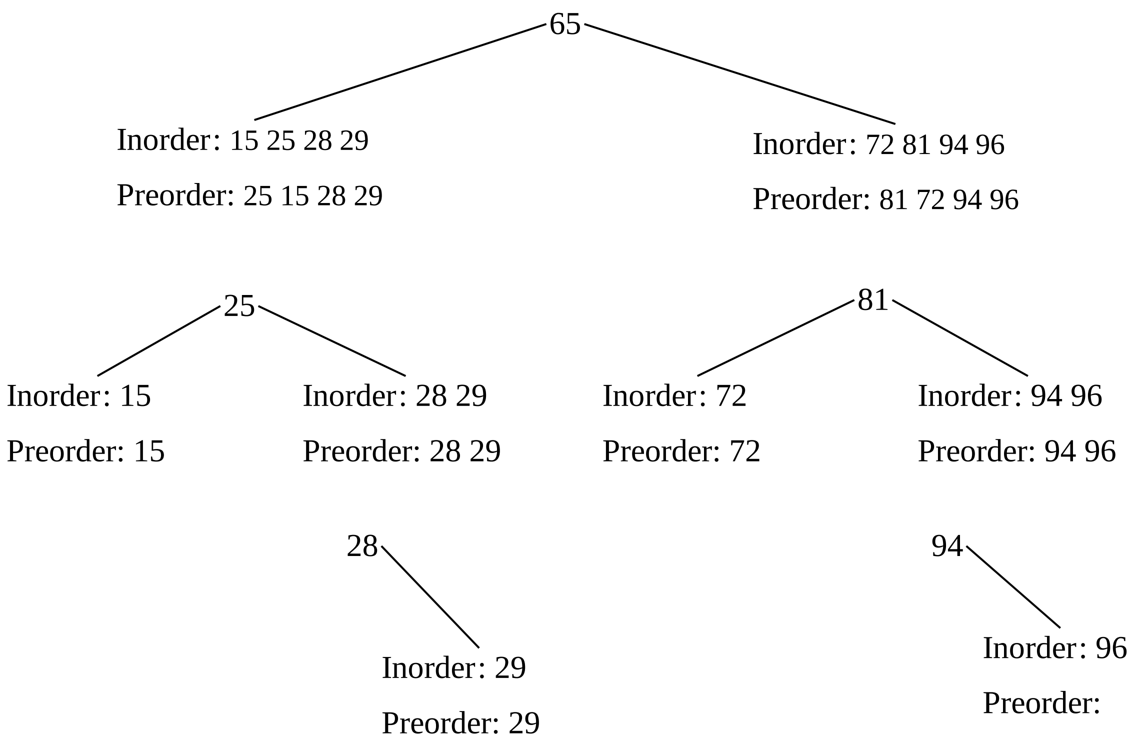
- A binary search tree can be constructed from its traversals.
- If the *Preorder* traversals is given, then the ***first node*** is *ROOT* node and *Postorder* traversal is given then ***last node*** is the *ROOT* node.

- For construction of a binary search tree from its traversals, two traversals are essentials. Out of which one should be *inorder* traversal and another one is either *preorder* (or) *postorder* traversal.

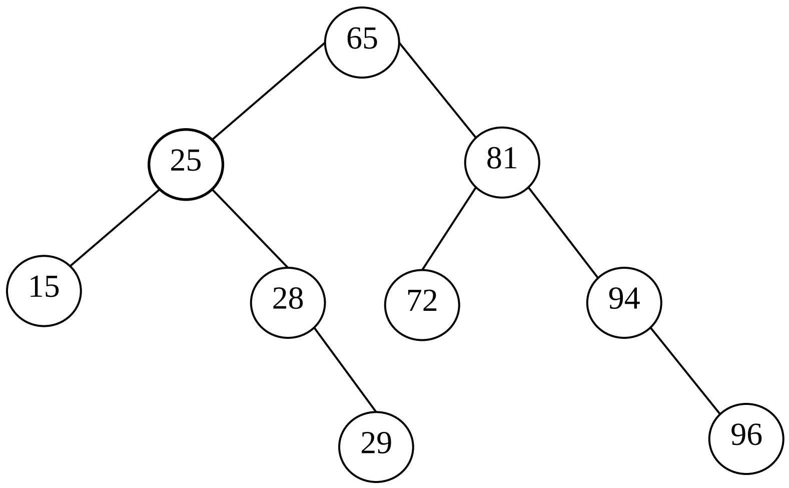
Eg.1:

Inorder:	15	25	28	29	65	72	81	94	96
Preorder:	65	25	15	28	29	81	72	94	96

1. From the preorder traversal, *65* is the *ROOT* node.
2. In the inorder traversal, all the nodes which are LEFT side of **65** belongs to LEFT sub tree and those node which are RIGHT side of **65** belongs to RIGHT sub tree.
3. Now the problem is reduced to two sub trees and same procedure can be applied repeatedly.



Final Binary Search Tree from the Inorder and Preorder as follows:



****

UNIT – V

Sorting and Searching

Objective :

- To impart the concepts of searching and sorting.

Syllabus:

Sorting and Searching

Searching: Linear search, Binary search , Fibonacci search

Sorting (Internal): Basic concepts, sorting by: insertion (Insertion sort), selection(Selection sort), exchange (Bubble sort, Quick sort),distribution(Radix sort) and merging(Merge sort).

Learning Outcomes:

At the end of the unit student will be able to:

- demonstrate the working process of sorting (bubble, insertion, selection , quick, merge andradix).
- Use and apply searching (linear , binary and fibonacci search) methods .

LEARNING MATERIAL

Searching:

Searching is a process of verifying whether the given element is available in the given set of elements or not. Types of Searching techniques are:

1. Linear Search
2. Binary Search
3. Fibonacci Search

1. Linear Search

In linear search, search process starts from starting index of array. i.e. **0th** index of array and end's with ending index of array. i.e. **(n-1)th** index. Here searching is done in Linear fashion (Sequential fashion).

Algorithm Linearsearch(a<array>, n, ele)

Input: ***a*** is an array with *n* elements, *ele* is the element to be searcheD)

Output: Position of required element in array, if it is available.

```
1. found =
2. i =0
3. while(i < n)
        a) if(ele == a[i]
                i) found = 1
                ii) print( element found at ith position)
                iii) break
        b) end if
        c) i = i +1
4. end loop
5. if( found == 0)
        a) print( required element to be search is not available)
```

```
    6. end if
End Linearsearch
```

Algorithm Linearsearch_Recurssion(a<array>, i, n, ele)

Input: *a* is an array with n elements, *ele* is the element to be searched, *i* is starting index of array and *n* is the ending index.

Output: Position of required element in array, if it is available.

```
    1. found = 0
    2. if(i<n)
        a) if(a[i] == ele)
            i) found = 1
            ii) print( element found at $i^{th}$ position)
        b) end if
        c) Linearsearch_Recurssion(a, i+1, n, ele)
    4. end if
    5. if( found == 0)
        a) print( required element to be search is not available)
    6. end if
End Linearsearch_Recurssion
```

2. *Binary Search*

The input to binary search must be in *ascending order*. i.e. set of elements be in ascending order.

Searching process in Binary search as follows:

- First, the element to be search is compared with middle element of array.
- If the required element to be searched is equal to middle element of array then Successful Search.
- If the required element to be searched is *less than* the middle element of array, then search in LEFT side of the midpoint of the array.
- If the required element to be search is *greater than* middle element of array, then

search in RIGHT side of the midpoint of the array.

Algorithm Binarysearch(a<array>, n, ele)

Input: *a* is an array with *n* elements, *ele* is the element to be searcheD)

Output: Position of required element in array, if it is available.

```
1. found = 0
2. low = 0
3. high = n-1
4. while(low <= high)
        a) mid = (low+high )/2.
        b) if(ele == a[mid])
                i) print(required element was found at mid position)
                ii) found = 1
                iii) break
        c) else if(ele < a[mid])
                i) high = mid - 1
        d) else if(ele > a[mid])
                i) low= mid + 1
        e) end if
5. end if
6. if(found == 0)
        a) print(required element is not available)
7. end if
```

End Binarysearch

Algorithm Binarysearch_Recursion(a<array>,ele, low, high)

Input: ***a*** is array with n elements, *ele* is the element to be searched, *low* is starting index, *high* is ending index of array.

Output: Position of required element in array, if it is available.

```
1. found = 0
2. if(low <= high)
        a) mid = (low+high )/2.
        b) if(ele == a[mid])
```

```
                i) print(required element was found at mid position)
                ii) found = 1
            c) else if(ele < a[mid])
                i) Binarysearch _Recursion(a,ele,low,mid-1)
            d) else if(ele > a[mid])
                i) Binarysearch_Recursion(a,ele,mid+1,high)
            e) end if
        3. end if
        4. if(found == 0)
            a) print(required element to be search is not available)
        5. end if
End Binarysearch_Recursion
```

3. Fibonacci Search: Fibonacci Search is a comparison-based technique that uses Fibonacci numbers to search an element in a sorted array.

Fibonacci search is a process of searching a sorted array by utilizing divide and conquer algorithm. Fibonacci search examines locations whose addresses have lower dispersion. When the search element has non-uniform access memory storage, the Fibonacci search algorithm reduces the average time needed for accessing a storage location.

Similarities with Binary Search:

1. Works for sorted arrays
2. A Divide and Conquer Algorithm.
3. Has Log n time complexity.

Differences with Binary Search:

1. Fibonacci Search divides given array in unequal parts
2. Binary Search uses division operator to divide range. Fibonacci Search doesn't use /, but uses + and -. The division operator may be costly on some CPUs.
3. Fibonacci Search examines relatively closer elements in subsequent steps. So when input array is big that cannot fit in CPU cache or even in RAM, Fibonacci Search can be useful

Algorithm Fibonacci Search :

Let the searched element be x.

1. Find the smallest Fibonacci Number greater than or equal to n. Let this number be fibM [m’th Fibonacci Number]. Let the two Fibonacci numbers preceding it be fibMm1 [(m-1)’th Fibonacci Number] and fibMm2 [(m-2)’th Fibonacci Number].
2. While the array has elements to be inspected:
    1. Compare x with the last element of the range covered by fibMm2
    2. **If** x matches, return index
    3. **Else If** x is less than the element, move the three Fibonacci variables two Fibonacci down, indicating elimination of approximately rear two-third of the remaining array.
    4. **Else** x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Together these indicate elimination of approximately front one-third of the remaining array.
3. Since there might be a single element remaining for comparison, check if fibMm1 is 1. If Yes, compare x with that remaining element. If match, return index.

Sorting: Sorting means arranging the elements either in ascending or descending order.
There are two types of sorting.

1. Internal Sorting.
2. External Sorting.

1. Internal Sorting: For sorting a set of elements, if we use only primary memory (Main memory), then that sorting process is known as internal sorting. i.e. internal sorting deals with data stored in computer memory.

2. External Sorting: For sorting a set of elements, if we use both primary memory (Main memory) and secondary memory, then that sorting process is known as external sorting. i.e. external sorting deals with data stored in files.

Different types of sorting techniques.

1. Bubble sort
2. Insertion sort
3. Selection sort
4. Merging sort
5. Quick sort
6. Radix sort

1. ***Bubble sort:***

- In bubble sort for sorting n elements, we require (n-1) passes (or) iterations and in each pass compare every element with its successor. i.e. i th index element will compare with (i+1) th index element, if they are not in ascending order, then swap them.
- Here for each pass, the largest element is moved to height index position of array to be sort.

Process:

1) In pass1, a[0] and a[1] are compared, then a[1] is compared with a[2], then a[2] is compared with a[3] and so on. Finally a[n-2] is compared with a[n-1]. Pass1 involves (n-1comparisons and places the biggest element at the highest index of the array to be sorted)
2. In pass2, a[0] and a[1] are compared, then a[1] is compared with a[2], then a[2] is compared with a[3] and so on. Finally a[n-3] is compared with a[n-2]. Pass2 involves (n-2 comparisons and places the biggest element at the highest index of the array to be sorted)
3. In pass (n-1), a[0] and a[1] are compared. After this step all the elements of the array are arranged in ascending order.

Eg: sort the elements 72, 85, 4, 32 and 16 using Bubble sort.

Pass 1	**0**	**1**	**2**	**3**	**4**	
	72	85	24	32	16	
	72	85	24	32	16	(0, 1) No Exchan
	72	24	85	32	16	(1, 2) Exchange
	72	24	32	85	16	(2, 3) Exchange
	72	24	32	16	**85**	(3, 4) Exchange
Pass 2						
	72	24	32	16	85	
	24	72	32	16	85	(0, 1) Exchange
	24	32	72	16	85	(1, 2) Exchange
	24	32	16	**72**	**85**	(2, 3) Exchange
Pass 3	24	32	16	72	85	
	24	32	16	72	85	(0, 1) No Exchange
	24	16	**32**	**72**	**85**	(1, 2) Exchange
Pass 4	24	16	32	72	85	
	16	**24**	**32**	**72**	**85**	(0, 1) Exchange

Sorted List is **16** **24** **32** **72** **85**

Algorithm Bubblesort(a<array>, n)

Input: *a* is an array with n elements to be sort.

Output: array elements in ascending order.

```
1. for( i = 0 to n-1)
        a) for( j = 0 to n-i-1)
                i) if ( a[j] > a[j+1] )
                        A) t = a[j]
                        B) a[j] = a[j+1]
                        C) a[j+1] = t
                ii) end if
        b) end j for loop
2. end i for loop
```

End Bubblesort

2. Insertion sort

- In the insertion sort, initially consider the 0th index element value as only sorted element, and then take remaining elements of the given set one by one.
- For every pass, compare unsorted elements one by one with sorted list.
- If sorted list value is GREATER than the unsorted element value, then **move** sorted list element to **next index.**
- Continue the above moving process up to sorted list element value is LESS than given unsorted element value.
- Continue the above process for all the elements in the given array.

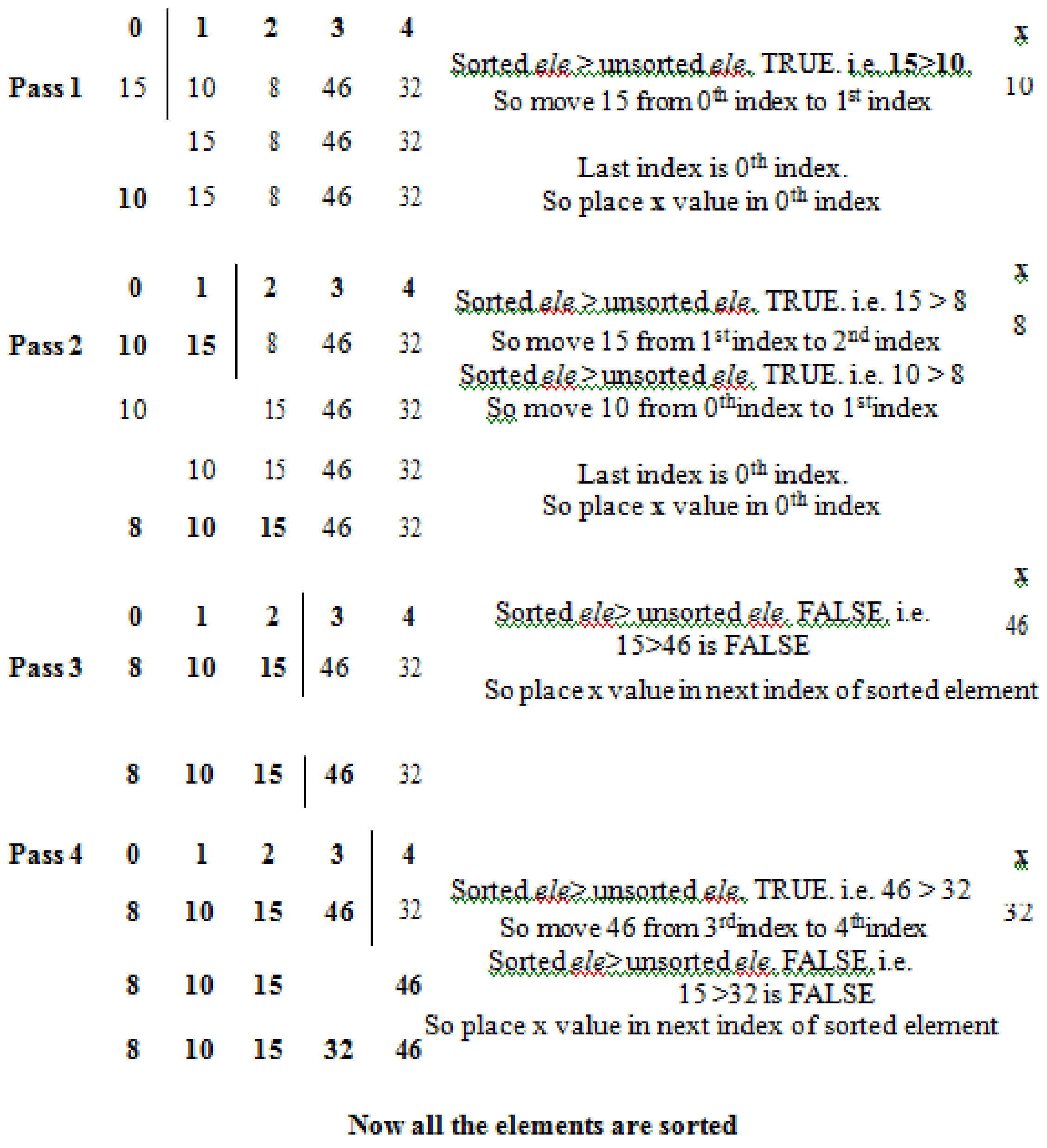
Algorithm insertionsort(a<array>,
n) Input: a is array with n elements to
be sorteD) **Output:** array elements in
ascending order.

```
1. i = 1
2. while( i < n)
```

```
        a) x = a[i]                    /* x is unsorted element */
        b) j = i -1
        c) while( j >= 0 &&
                a[j] > x ) i)
                a[j+1] = a[j]
                ii) j = j -1
        d) end loop
        e) a[j+1] = x
        f) i = i + 1
    3. end loop
```

End insertionsort

Eg: sort the elements 15,10, 8, 46, 32 using Insertion sort. Where **x** is an unsorted element



```
          0 | 1    2    3    4                                                  x
Pass 1   15 | 10   8    46   32   Sorted ele > unsorted ele. TRUE. i.e. 15>10.  10
                                   So move 15 from 0th index to 1st index
              15   8    46   32
                                   Last index is 0th index.
         10   15   8    46   32    So place x value in 0th index

          0    1 | 2    3    4                                                  x
Pass 2   10   15 | 8    46   32   Sorted ele > unsorted ele. TRUE. i.e. 15 > 8  8
                                   So move 15 from 1st index to 2nd index
         10        15   46   32   Sorted ele > unsorted ele. TRUE. i.e. 10 > 8
                                   So move 10 from 0th index to 1st index
              10   15   46   32
                                   Last index is 0th index.
          8   10   15   46   32    So place x value in 0th index

                                                                                x
          0    1    2 | 3    4    Sorted ele> unsorted ele. FALSE. i.e.         46
Pass 3    8   10   15 | 46   32        15>46 is FALSE
                                   So place x value in next index of sorted element

          8   10   15 | 46   32

Pass 4    0    1    2    3 | 4                                                  x
          8   10   15   46 | 32   Sorted ele> unsorted ele. TRUE. i.e. 46 > 32  32
                                   So move 46 from 3rd index to 4th index
          8   10   15        46   Sorted ele> unsorted ele. FALSE. i.e.
                                        15 >32 is FALSE
                                   So place x value in next index of sorted element
          8   10   15   32   46
```

Now all the elements are sorted

3. Selection Sort

In selection sort first find the smallest element in the array and place it in to the array to the 0th position. Then find the second smallest element in the array and place it in 1st position. Repeat this procedure until array is sorteD)

Process:

- In Pass1, find the position of the smallest element in the array and then swap a[pos] and a[0]. Now a[0] is sorteD)

- In Pass2, find the position of the smallest element in sub array of n-1 elements, then swap a[pos] and a[1]. Now a[1] is sorteD)
- In Pass n-1, find the position of the smallest element from a[n-2] and a[n-1], then swap a[pos] and a[n-21]. So that a[0], a[1], a[2], a[3], ..., a[n-1] is sorteD)

Eg: sort the elements 15, 10, 11, 41, 3 using selection sort

	0	**1**	**2**	**3**	**4**		pos
Pass 1	15	10	11	41	3	pos is initially assigned at 0th index	0
	$\underline{15}$	$\underline{10}$	11	41	3	a[1] < a[pos] is TRUE. So pos =1	1
	15	$\underline{10}$	$\underline{11}$	41	3	a[2] < a[pos] is FALSE. So No change in pos	1
	15	$\underline{10}$	11	$\underline{41}$	3	a[3] < a[pos] is FALSE. So No change in pos	1
	15	$\underline{10}$	11	41	$\underline{3}$	a[4] < a[pos] is TRUE. So pos =4	4

Now swap a[pos] and a[0]

0	**1**	**2**	**3**	**4**
3	10	11	41	**15**

Now a[0] is sorted

	0	**1**	**2**	**3**	**4**		**pos**
Pass 2	**3**	10	11	41	15	Now pos is assigned at 1st index	1
	3	$\underline{10}$	$\underline{11}$	41	15	a[2] < a[pos] is FALSE. So No change in pos	1
	3	$\underline{10}$	11	$\underline{41}$	15	a[3] < a[pos] is FALSE. So No change in pos	1
	3	$\underline{10}$	11	41	$\underline{15}$	a[4] < a[pos] is FALSE. So No change in pos	1

Now swap a[pos] and a[1]

0	**1**	**2**	**3**	**4**
3	**10**	11	41	15

Now a[1] is sorted

	0	**1**	**2**	**3**	**4**		**pos**
Pass 3	**3**	**10**	11	41	15	Now pos is assigned at 2ndindex	2
	3	**10**	$\underline{11}$	$\underline{41}$	15	a[3] < a[pos] is FALSE. So No change in	2

						pos	
3	**10**	11	41	15	a[4] < a[pos] is FALSE. So No change in pos	2

Now swap a[pos] and a[2]

0	**1**	**2**	**3**	**4**
3	**10**	**11**	41	15

Now a[2] is sorted

	0	**1**	**2**	**3**	**4**		**pos**
Pass 4	**3**	**10**	**11**	41	15	Now pos is assigned at 3rd index	3
	3	**10**	**11**	41	15	a[4] < a[pos] is TRUE. So pos = 4	4

Now swap a[pos] and a[3]

0	**1**	**2**	**3**	**4**
3	**10**	**11**	**15**	**41**

Now all the elements are sorted

Algorithm selectionsort (a<array>, n)

Input: *a* is an array with *n* elements to be sorteD)

Output: array elements in ascending order.

```
1. i = 0
2. while( i < n)
        a) pos = i
        b) j = i+1
        c) while ( j < n)
                i) if ( a[j] < a[pos] )
                        A) pos = j
                ii) end if
                iii) j = j +1
        d) end loop
        e) t = a[pos]
        f) a[pos] = a[i]
```

```
            g) a[i] =t
            h) i= i+1
        3. end loop
End selectionsort
```

4. Quick Sort:

Quick Sort is also based on the concept of **Divide and Conquer**, just like merge sort. But in quick sort all the heavy lifting(major work) is done while **dividing** the array into subarrays, while in case of merge sort, all the real work happens during **merging** the subarrays. In case of quick sort, the combine step does absolutely nothing.

It is also called **partition-exchange sort**. This algorithm divides the list into three main parts:

1. Elements less than the **Pivot** element
2. Pivot element(Central element)
3. Elements greater than the pivot element

Pivot element can be any element from the array, it can be the first element, the last element or any random element. In this tutorial, we will take the rightmost element or the last element as **pivot**.

For example: In the array {52, 37, 63, 14, 17, 8, 6, 25}, we take 25 as **pivot**. So after the first pass, the list will be changed like this.

{6 8 17 14 **25** 63 37 52}

Hence after the first pass, pivot will be set at its position, with all the elements **smaller** to it on its left and all the elements **larger** than to its right. Now 6 8 17 14 and 63 37 52 are considered as two separate sunarrays, and same recursive logic will be applied on them, and we will keep doing this until the complete array is sorted.

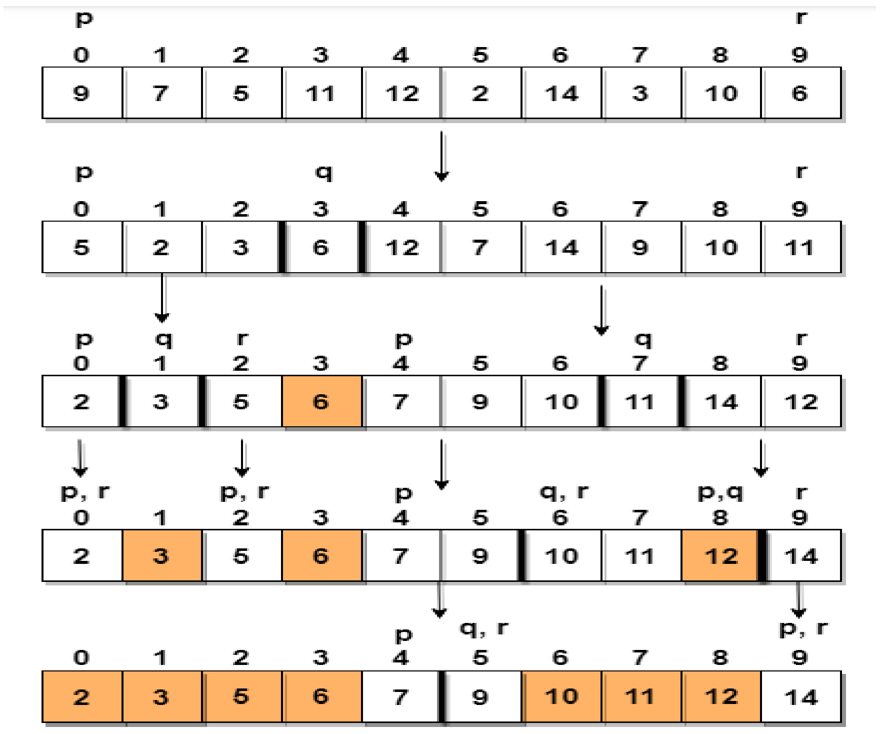
Algorithm :

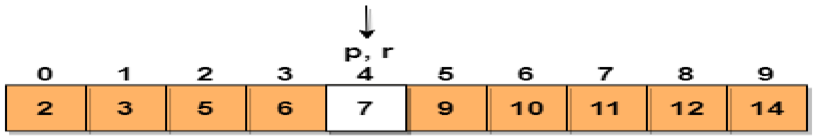
Following are the steps involved in quick sort algorithm:

1. After selecting an element as **pivot**, which is the last index of the array in our case, we divide the array for the first time.
2. In quick sort, we call this **partitioning**. It is not simple breaking down of array into 2 subarrays, but in case of partitioning, the array elements are so positioned that all the elements smaller than the **pivot** will be on the left side of the pivot and all the elements greater than the pivot will be on the right side of it.
3. And the **pivot** element will be at its final **sorted** position.
4. The elements to the left and right, may not be sorted.
5. Then we pick subarrays, elements on the left of **pivot** and elements on the right of **pivot**, and we perform **partitioning** on them by choosing a **pivot** in the subarrays.

Let's consider an array with values {9, 7, 5, 11, 12, 2, 14, 3, 10, 6}

Below, we have a pictorial representation of how quick sort will sort the given array.





In step 1, we select the last element as the **pivot**, which is `6` in this case, and call for `partitioning`, hence re-arranging the array in such a way that `6` will be placed in its final position and to its left will be all the elements less than it and to its right, we will have all the elements greater than it.

Then we pick the subarray on the left and the subarray on the right and select a **pivot** for them, in the above diagram, we chose `3` as pivot for the left subarray and `11` as pivot for the right subarray.

And we again call for `partitioning`.

5. Merge Sort:

Merge Sort follows the rule of **Divide and Conquer** to sort a given set of numbers/elements, recursively, hence consuming less time.

In the last two tutorials, we learned about Selection Sort and Insertion Sort, both of which have a worst-case running time of `O(n²)`. As the size of input grows, insertion and selection sort can take a long time to run.

Merge sort , on the other hand, runs in `O(n*log n)` time in all the cases.

Before jumping on to, how merge sort works and it's implementation, first lets understand what is the rule of **Divide and Conquer**?

Divide and Conquer

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

Let's take an example, **Divide and Rule**.

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

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

In **Merge Sort**, the given unsorted array with `n` elements, is divided into `n` subarrays, each having **one** element, because a single element is always sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.

The concept of Divide and Conquer involves three steps:

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

How Merge Sort Works?

As we have already discussed that merge sort utilizes divide-and-conquer rule to break the problem into sub-problems, the problem in this case being, **sorting a given array**.

In merge sort, we break the given array midway, for example if the original array had `6` elements, then merge sort will break it down into two subarrays with `3` elements each.

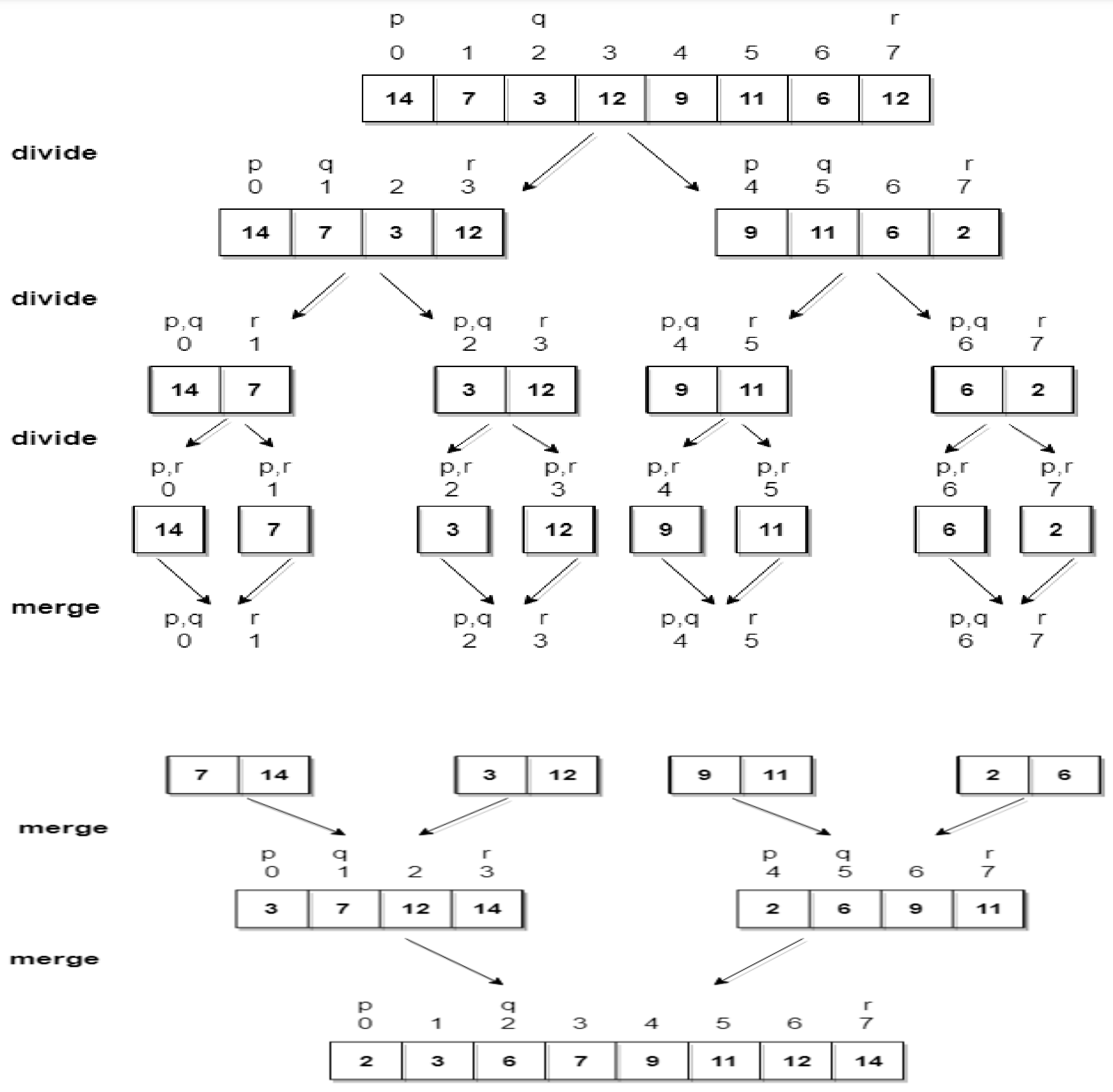
But breaking the orignal array into 2 smaller subarrays is not helping us in sorting the array.

So we will break these subarrays into even smaller subarrays, until we have multiple subarrays with **single element** in them. Now, the idea here is that an array with a single element is already sorted, so once we break the original array into subarrays which has only a single element, we have successfully broken down our problem into base problems.

And then we have to merge all these sorted subarrays, step by step to form one single sorted array.

Let's consider an array with values `{14, 7, 3, 12, 9, 11, 6, 12}`

Below, we have a pictorial representation of how merge sort will sort the given array.



In merge sort we follow the following steps:

1. We take a variable p and store the starting index of our array in this. And we take another variable r and store the last index of array in it.
2. Then we find the middle of the array using the formula (p + r)/2 and mark the middle index as q, and break the array into two subarrays, from p to q and from q + 1 to r index.
3. Then we divide these 2 subarrays again, just like we divided our main array and this continues.
4. Once we have divided the main array into subarrays with single elements, then we start merging the subarrays.

6. **Radix Sort** :
   Radix sort also known as Bin sort or Bucket sort.

Radix sort is one of the sorting algorithms used to sort a list of integer numbers in an order. In radix sort algorithm, list of integer numbers will be sorted based on the digits of individual numbers. Sorting is performed from **least significant digit to the most significant digit**.

- Radix sort algorithm requires number of passes which are equal to the number of digits present in the largest number among the list of numbers.
- For example, if the largest number is a 3 digit number then that list is sorted with 3 passes.

Step by Step Process

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

Step 1 - Define 10 queues each representing a bucket for each digit from 0 to 9.

Step 2 - Consider the least significant digit of each number in the list which is to be sorted.

Step 3 - Insert each number into their respective queue based on the least significant digit.

Step 4 - Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.

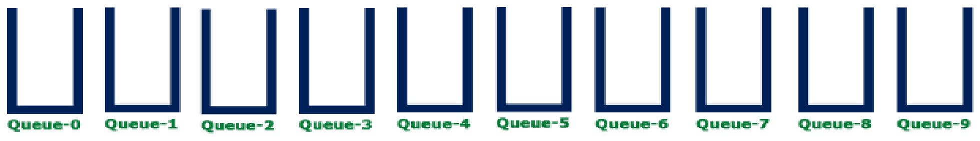
Step 5 - Repeat from step 3 based on the next least significant digit.

Step 6 - Repeat from step 2 until all the numbers are grouped based on the most significant digit.

Consider the following list of unsorted integer numbers

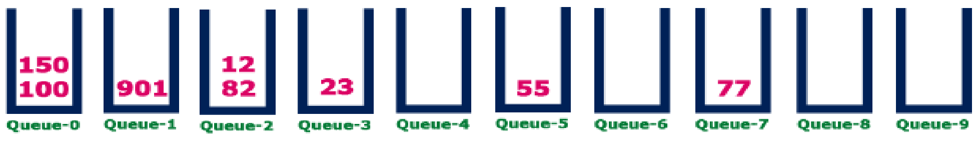
82, 901, 100, 12, 150, 77, 55 & 23

Step 1 - Define 10 queues each represents a bucket for digits from 0 to 9.



Step 2 - Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

8**2**, 90**1**, 10**0**, 1**2**, 15**0**, 7**7**, 5**5** & 2**3**

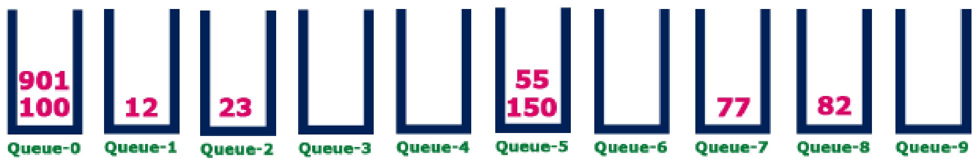


Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 150, 901, 82, 12, 23, 55 & 77

Step 3 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Tens placed digit) of every number.

1**0**0, 1**5**0, 9**0**1, **8**2, **1**2, **2**3, **5**5 & **7**7

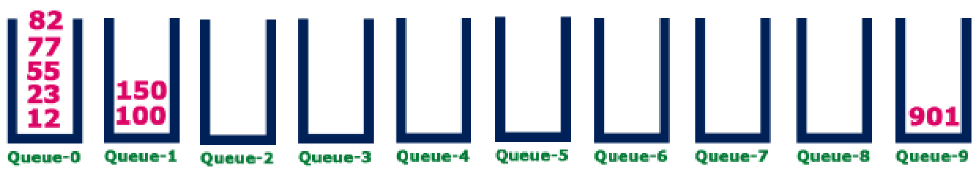


Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

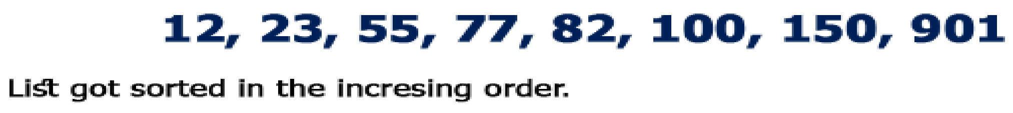
100, 901, 12, 23, 150, 55, 77 & 82

Step 4 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Hundres placed digit) of every number.

100, **9**01, 12, 23, **1**50, 55, 77 & 82



Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.



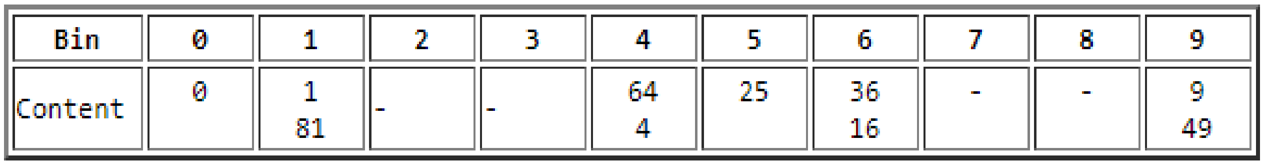
12, 23, 55, 77, 82, 100, 150, 901

List got sorted in the incresing order.

Example2: Consider the list of integers:

36, 9, 0, 25, 1, 49, 64, 16, 81, 4.

n is 10 and the numbers all lie in (0,99). After the first phase, we will have:

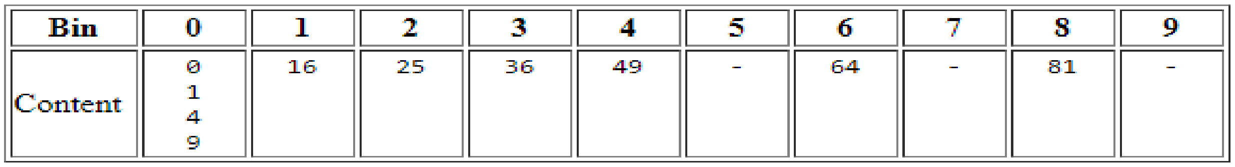


Bin	0	1	2	3	4	5	6	7	8	9
Content	0	1 81	-	-	64 4	25	36 16	-	-	9 49

Note that in this phase, we placed each item in a bin indexed by the least significant decimal digit.

Output of the phase:0,1,81,64,4,25,36,16,9,49.

Repeating the process, will produce:



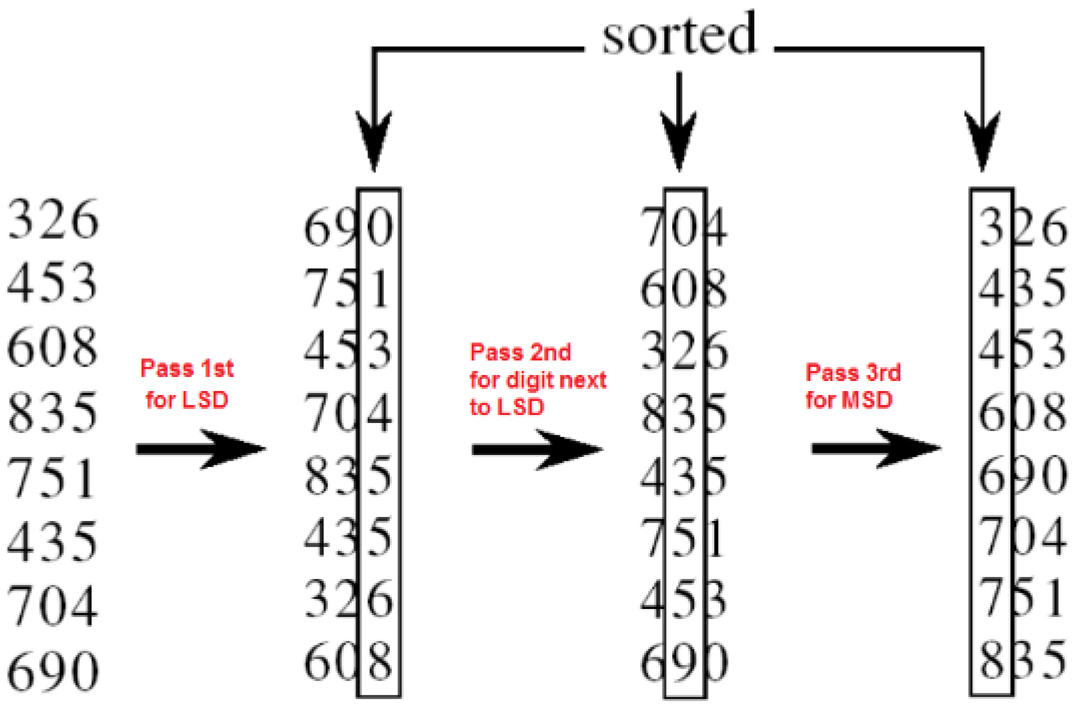
Bin	0	1	2	3	4	5	6	7	8	9
Content	0 1 4 9	16	25	36	49	-	64	-	81	-

In this second phase, we used the leading decimal digit to allocate items to bins, being careful to add each item to the end of the bin.

Output of the phase: **0,1,4,9,16,25,36,49,64,81.**

We can apply this process to numbers of any size expressed to any suitable base or *radix*.

Example 3: *Sort the list of elements - 326,453,608,835,751,435,704,690*



After sorting the list: 326,435,453,608,690,704,751,835.

****

Learning Material

UNIT – VI

GRAPHS

Objective:

- To gain knowledge of graph data structure.

Syllabus:

Graphs

Graphs- Basic concepts, representations of graphs, operations on graphs-vertex insertion, vertex deletion, find vertex, edge addition, edge deletion, graph traversals-breadth first search and depth first search)

Learning Outcomes:

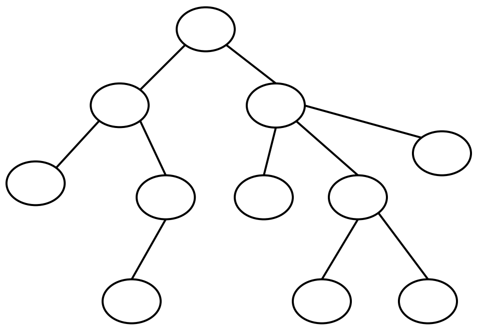
At the end of the unit student will be able to

- know various graph representation techniques.
- perform various operations on graphs.
- implement graph traversal techniques.

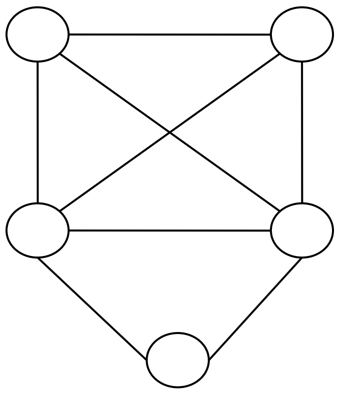
Graphs

Basic Concepts:

- **Graph** is another important non-linear data structure.
- Graph is a collection of vertices and edges that connect these vertices
- The tree structure is a special kind of graph structure.
- In tree structure there is a hierarchical relationship between parent and children, i.e. one parent and many children.
- In the graph relationship is from many parents to many children.



TREE



GRAPH

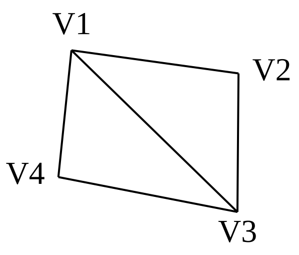
- **Graph Terminology**

  Graph: A graph consists of two sets.

  (i) A set V(G) called as set of all vertices.

  (ii) A set E(G) called as set of all edges (arcs). This set of edges is pair of elements from V(G).

  Eg:

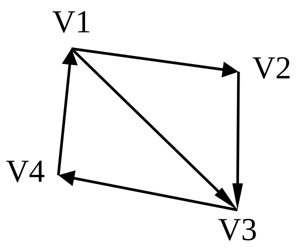


For the above graph V(G)={V1, V2, V3, V4}

E(G) = {(V1,V2), (V2,V3), (V1,V3), (V1,V4), (V3,V4)}

- **Digraph:** A digraph is also called as directed graph. It is a graph G, such that G=<V,E>, where V is set of all the vertices and E is set of ordered pair of elements from V.

  Eg:

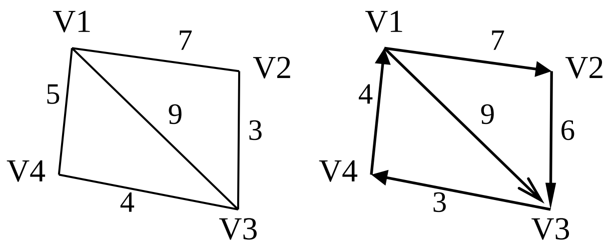


  V(G)={V1, V2, V3, V4}

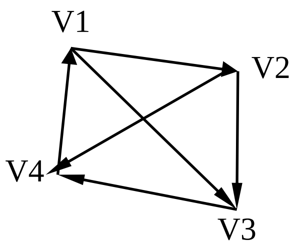
  E(G) = {(V1,V2), (V2,V3), (V3,V4), (V4,V1), (V1,V3)}

- **Weighted Graph:** A graph (or digraph) is termed as weighted graph, if all the edges in the graph are labeled with some weights.

  Eg:



- **Adjacent Vertex:** A vertex V_i is adjacent (neighbor of) of another vertex say V_j, if there is an edge from V_i to V_j.



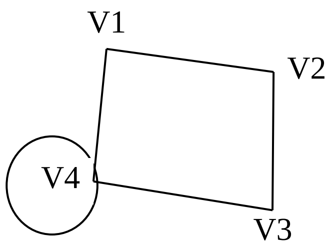
  Eg:

  V2 is adjacent to V3 and V4.

  V1 is not adjacent to V4.

- **Self-loop:** if there is an edge whose starting and ending vertices are same, i.e. (Vi, Vi), then that edge is called as self-loop (loop).

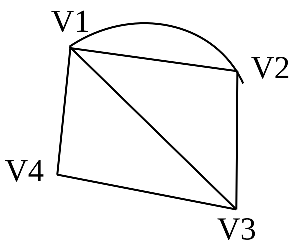
Eg:



In the above graph Vertex V4 has self-loop.

- **Parallel edges:** if there is more than one edge between the same pair of vertices, then they are known as parallel edges.

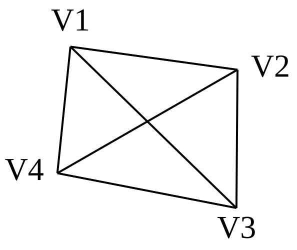
  Eg:

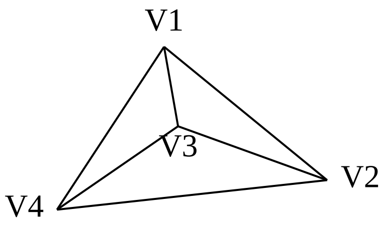


In the above graph two parallel edges between vertices V1 and V2.

- **Multi graph:** A graph which has either self-looped (or) parallel edges (or) both, that type of graph is called as multi graph.
- **Simple Graph (digraph):** A graph (digraph), if it doesn't have any parallel edges or self-loops, such types of graphs are called as Simple Graph (digraph).
- **Complete Graph:** A graph G is said to be complete graph, if there are edges from any vertex to all other vertices present in Graph.

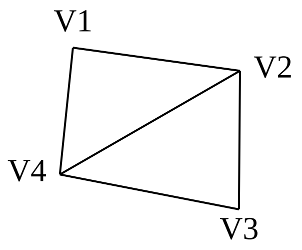
  Eg:

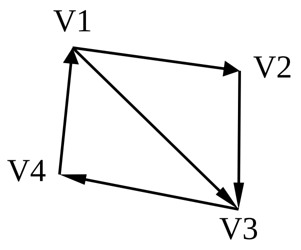




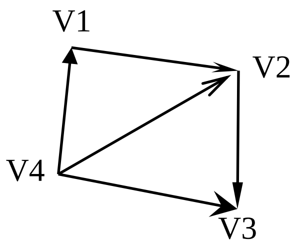
For **n** number of vertices present in complete graph, the total no.of edges are $\frac{n(n-1)}{2}$.

- **Cycle:** If there is a path containing one or more edges, which start from vertex V_i and terminates with same vertex V_i, then that path is known as Cyclic path (or) cycle.



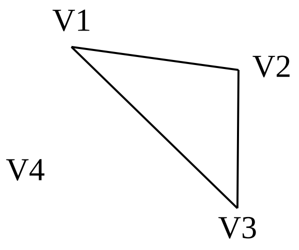


- **Cyclic Graph**: A graph (digraph) that have cycle(s) is called Cyclic Graph (digraph).
- **Acyclic Graph:** A graph (digraph) that does not have cycle(s) is called Acyclic Graph (digraph).



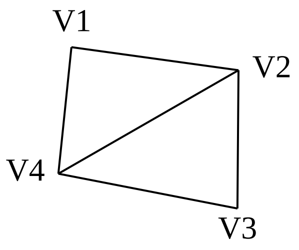
- **Isolated Vertex:** In a graph, a vertex is isolated, if there is no edge is connected from any vertex to the other vertex.

  Eg:



In the above graph V4 is a isolated vertex.

- **Degree of Vertex:** The no.of edges are connected to a vertex is called as degree of a vertex and is denoted by degree(V_i).

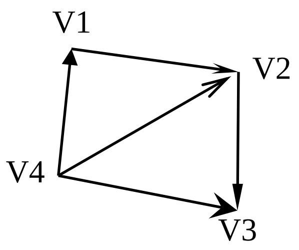


degree(V1) = 2

degree(V2) = 3

- For **digraph**, there are two edges. i.e. indegree and outdegree.
- Indegree of V_i denoted as indegree(V_i) = no.of edges coming towards vertex V_i.
- Outdegree of Vi denoted as outdegree(V_i) = no.of edges coming out from vertex V_i.

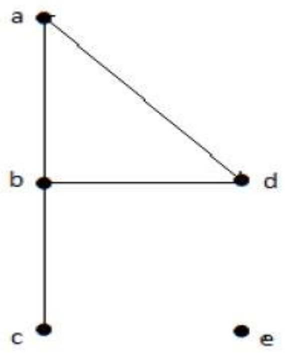
Eg:



Indegree(V1) = 1	Indegree(V2) = 2
Indegree(V4) = 0	Outdegree(V4) = 3

Pendent Vertex: A vertex of a graph is said to be pendant if its neighbourhood contains exactly one vertex. (or) A vertex with degree one is called a pendent vertex.

Eg:



In the above Undirected Graph,

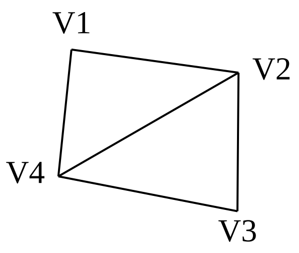
- deg(a) = 2, as there are 2 edges meeting at vertex 'a'.
- deg(b) = 3, as there are 3 edges meeting at vertex 'b'.
- deg(c) = 1, as there is 1 edge formed at vertex 'c'. So 'c' is a **pendent vertex.**
- deg(d) = 2, as there are 2 edges meeting at vertex 'd'.
- deg(e) = 0, as there are 0 edges formed at vertex 'e'. So 'e' is an **isolated vertex.**

From the above graph, vertex 'a' and vertex 'b' has degree as one which are also called as the pendent vertex.

- **Connected Graph:** In a graph (not digraph) G, two vertices V_i and V_j are said to be connected, if there is a path in G from V_i to V_j (or) V_j to V_i.

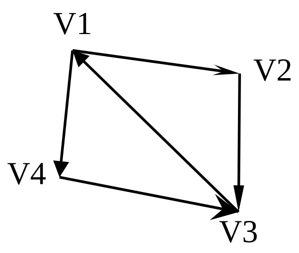
A Graph G is said to be the connected, if for every pair of distinct vertices V_i, V_j in graph, there is a Path.

Eg:

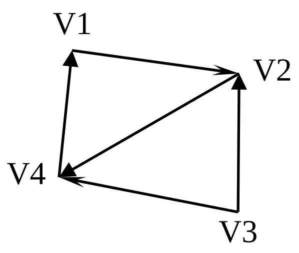


A **digraph** with the above property is called as **Strongly Connected graph**.i.e. a digraph G is said to be Strongly Connected, if for every pair of distinct vertices V_i, V_j in G, there is a Direct path from V_i to V_j and also fromV_j to V_i.

Eg:



Strongly Connected Graph

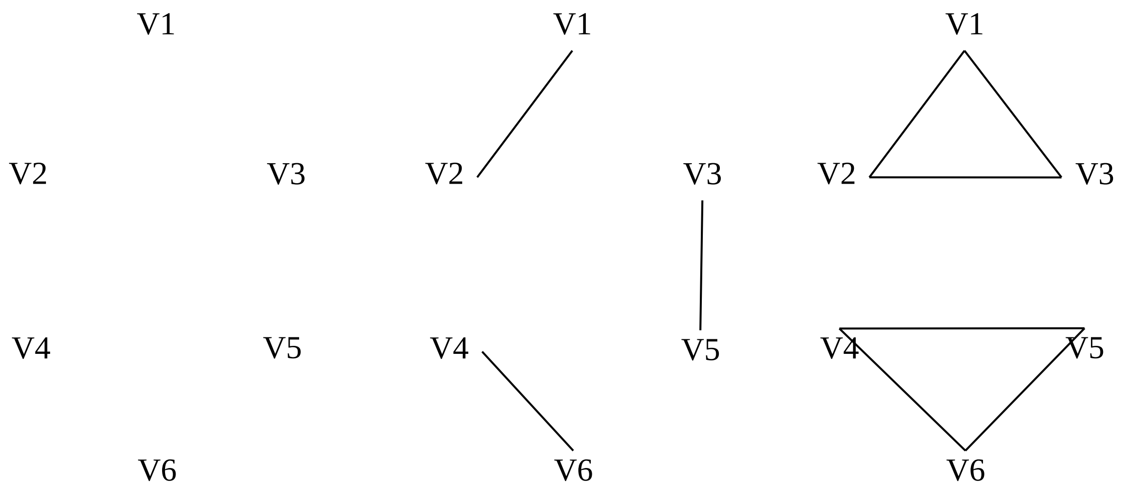


Not Strongly Connected Graph

- **Regular Graph:** In a graph, where every vertex has same degree, such type of graph is called as Regular Graph.

A Regular Graph with vertices of degree K is called as **K-Regular Graph**.

Eg:



0- Regular Graph **1- Regular Graph** **2- Regular Graph**

Representation of a Graph

A graph can be represented in the following ways.

1. Set Representation
2. Linked List Representation (or) Adjacency List Representation
3. Matrix (or) Adjacency Matrix Representation

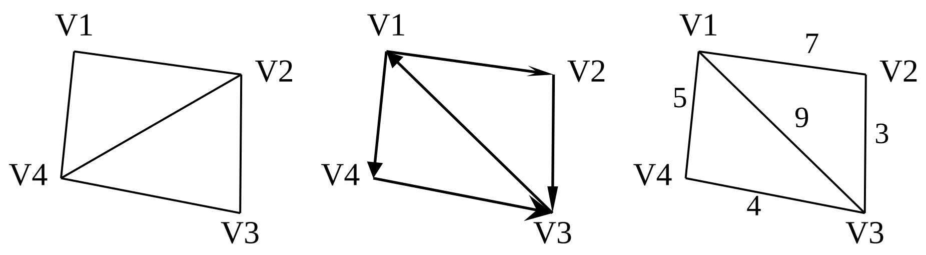
1. Set Representation:

- This is straight forward method of representing graph.
- In this method, 2 sets are maintained V(G) and E(G).

V(G) is set of Vertices.

E(G) is set of Edges.

Eg: G1 G2 G3



V(G1)={V1, V2, V3, V4}

E(G1) = {(V1,V2), (V1,V4), (V2,V4), (V2,V3), (V3,V4)}

V(G2)={V1, V2, V3, V4}

E(G2)={(V1,V2), (V1,V4), (V2,V3), (V3,V1), (V4,V3)}

For representation of weighted graphs, the edge set consist of 3 tuples. i.e. E= W × V × V, where W is the set of edge weights.

V(G3)={V1, V2, V3, V4}

E(G3)={(7,V1,V2), (5,V1,V4), (9,V1,V3), (3,V2,V3), (4,V4,V3)}

* Multi graphs (undirected) can't be able to represent with the help of Set representation.

2. Linked List Representation:

Linked List Representation is another space saving way of graph representation.

In this graph representation, two types of node structures are assumed.

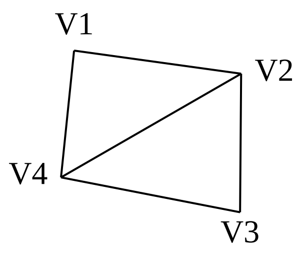
Node_Label	Adj_List

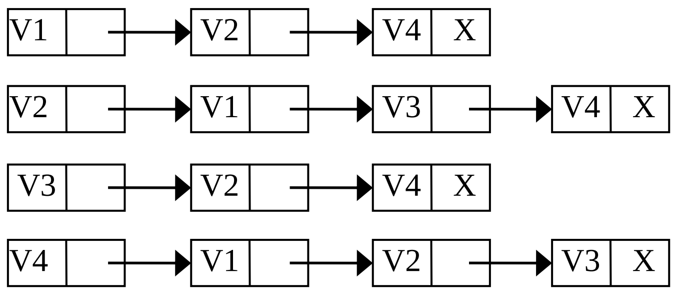
Node Structure for Un-weighted Graph

Weight	Node_Label	Adj_List

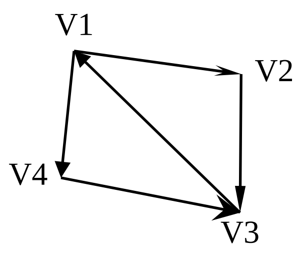
Node Structure for weighted Graph

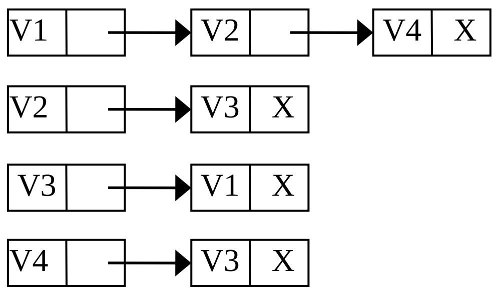
For Graph G1:



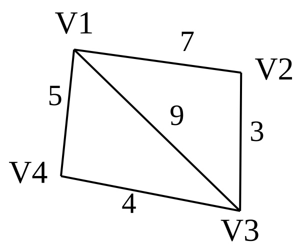


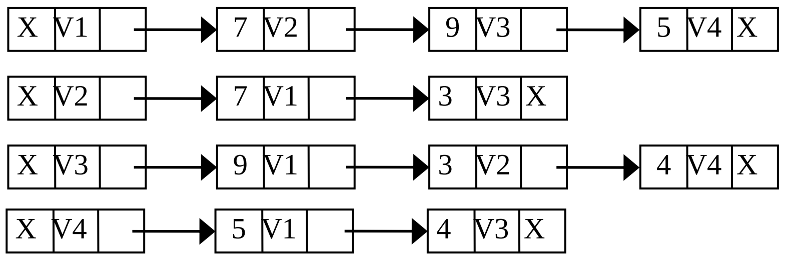
For Graph G2:





For Graph G3:



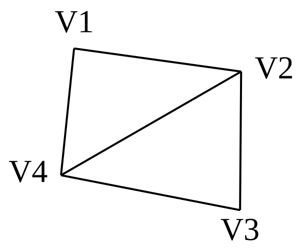


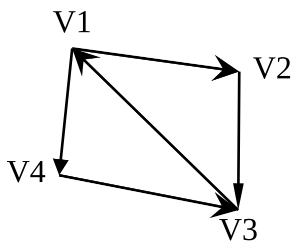
3. Adjacency Matrix Representation:

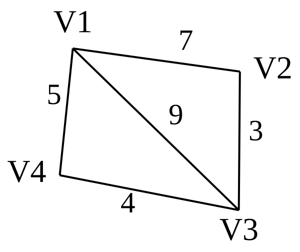
This representation uses a square matrix of order n × n, where n is no.of vertices in graph.

Adjacency Matrix is also termed as Bit matrix (or) Boolean Matrix as the entries are 0 (or) 1.

Eg: G1 G2 G3







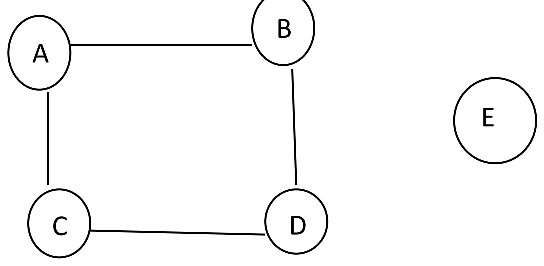
$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 \\ 4 & 1 & 1 & 1 & 0 \end{array} \qquad \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 0 & 0 & 1 & 0 \\ 3 & 1 & 0 & 0 & 0 \\ 4 & 0 & 0 & 1 & 0 \end{array} \qquad \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 7 & 9 & 5 \\ 2 & 7 & 0 & 3 & 0 \\ 3 & 9 & 3 & 0 & 4 \\ 4 & 5 & 0 & 4 & 0 \end{array}$$

Operations on Graphs:

1. **Vertex Insertion**

   If a new vertex ‘v’ is inserted into the graph G then it returns a graph with vertex ‘v’ inserted. The vertex ‘v’ has no incident edges.

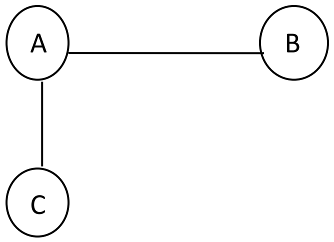
   Eg: A new vertex E is inserted in the graph.



2. **Vertex Deletion**

If a vertex ‘v’ is deleted from the graph G then it returns a graph in which vertex ‘v’ and all edges incident to it are removed.

Eg: Verex D is deleted



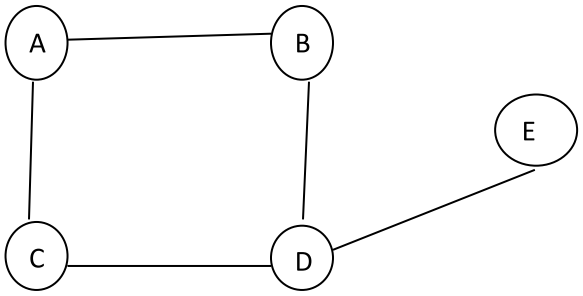
3. **Find Vertex**

This operation is used to find whether the given vertex is present in a graph or not.

4. **Edge Addition**

If a new edge ‘e’ is inserted between two vertices v1 and v2 in the graph G then it returns a graph with a new edge between v1 and v2.

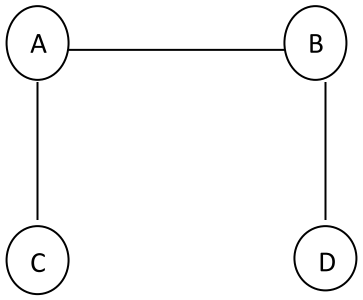
Eg: A new edge is inserted between D and E



5. **Edge Deletion**

If an edge ‘e’ between vertices v1 and v2 is deleted from the graph G then it returns a graph in which edge between v1 and v2 is removed so that the incident vertices v1 and v2 remains in the graph.

Eg: An edge between C and D is deleted



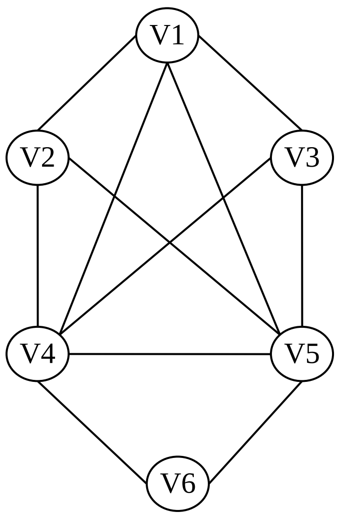
Graph Traversals:

Two techniques are available.

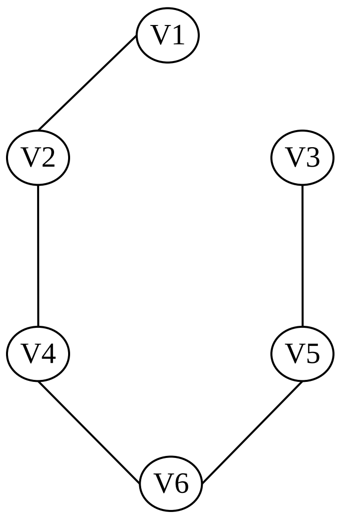
1. Depth First Search (DFS)
2. Breadth First Search (BFS)

Depth First Search (DFS):

- It is similar to the inorder traversal of a binary tree.
- Here, starting from the given node, DFS traversal visit all the nodes upto the deepest level and so on.
- While traversing the vertices, Cyclic path (Closed path) can't be occur. i.e. we can visit a vertex only once.



Graph



DFS of Graph

The sequence of visiting of vertices for the above as follows.

V1-V2-V4-V6-V5-V3

- To traverse a graph in DFS, a stack and one single linked list is required.
- A stack can be used to maintain the track of all paths from a vertex. Let the name of the stack is OPEN.
- A Single Linked List, VISIT can be maintained to store the vertices already visited.
- Here OPEN is the name of the stack and VISIT is the name of the Single Linked List.

<u>Process:</u>

- Initially the starting vertex will be pushed on to the stack OPEN.
- To visit a vertex, POP a vertex from stack OPEN, and then PUSH all the adjacent vertices into stack OPEN.

- Whenever a vertex s popped, check whether it is already visited or not by searching in Single Linked List VISIT.
- If the vertex is already visited, then we will simply ignore it and we will POP the stack OPEN for the next vertex to be visited. This procedure is continued till the stack is not empty.

Informal description for DFS Traversal of a graph using *array representation* as follows:

Algorithm DFS( )

Input: Adjacent matrix representation of a graph.

Output: DFS traversal of graph.

```
1. PUSH the starting vertex into stack
2. While stack is not empty
   a) POP a vertex v from stack
   b) if vertex v is not visited
      (i) visit the vertex v
      (ii) PUSH all the adjacent vertices of v into stack
   c) end if
3. end loop
```

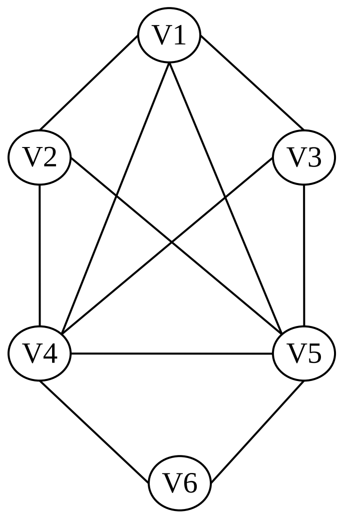
End DFS

Breadth First Search (BFS)

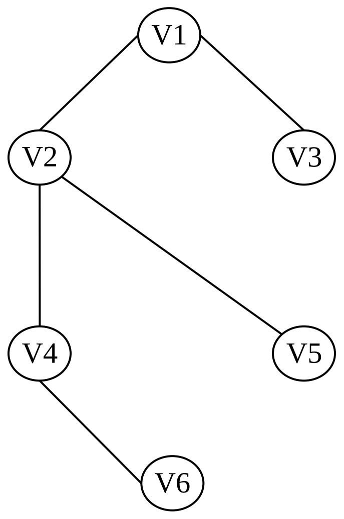
Here any vertex in level i will be visited only after the visiting of all the vertices present in the preceding level. i.e. level i-1.

Simply BFS can be call as Level-by-Level traversal.

Eg:



Graph



BFS of Graph

The order of visiting of vertices in the above BFS traversal of a graph as follows:

V1-V2-V3-V4-V5-V6

- The implement idea of BFS traversal is almost same as DFS traversal except that, BFS uses Queue Data Structure instead of Stack Data Structure in DFS**.**
- Let us assume the name of the Queue is OPENQ to use it in BFS and Single Linked List is VISIT, to store the order of vertices visited during the BFS traversal.

Informal description for BFS Traversal of a graph using array representation as follows:

Algorithm BFS( )

Input: Adjacent matrix representation of a graph.

Output: BFS traversal of graph.

```
1. ENQUEUE the starting vertex into queue
2. While queue is not empty
   a) DEQUEUE a vertex v from queue
   b) if vertex v is not visited
      (i) visit the vertex v
      (ii) ENQUEUE all the adjacent vertices of v into queue
   c) end if
3. end loop
```

End BFS

LINKED LISTS
UNIT-I
Assignment-Cum-Tutorial Questions

SECTION-A

Objective Questions

1. Which of the following points is/are true about Linked List data structure when it is compared with array [ ]

   a. It is easy to insert and delete elements in Linked List.

   b. Random access is not allowed in a typical implementation of Linked Lists

   c. The size of array has to be pre-decided, linked lists can change their size any time.

   d. All of the above

2. A linear collection of data elements where the linear node is given by means of pointer is called? [ ]

   a) Linked list
   b) Node list
   c) Primitive list
   d) None

3. In single linked list each node contain minimum of two fields. One field is data field to store the data and what is the purpose of the second field is used to store ? [ ]

   a. A. Pointer to character B. Pointer to integer

   b. C. Pointer to next node D. None

4. Identify the memory allocation process in Linked list [ ]

   a. A. Dynamic B. Compile Time C. Static D. None of these

5. A variant of linked list, identify in which last node of the list points to the first node of the list is? [ ]

   a. A. Singly linked list B. Doubly linked list

   C. Circular linked list D. Multiply linked list

6. In doubly linked lists, identify which type of traversal can be performed?[ ]

   a. A. Only in forward direction B. Only in reverse direction

   b. C. In both directions D. None

7. A variant of the linked list, identify in which none of the node contains NULL pointer is? [ ]

   a. A. Singly linked list B. Doubly linked list

   b. C. Circular linked list D. None

8. Identify non-linear Data Structure from the following [ ]

   a. A. Array B. Stack C. Graph D. Linked list

9. A node in single linked list can refer the previous node. [True/False]

10. Which type of structure is used to create a linked list? [ ]

    a. A. Nested structure B. Self referential structure

    b. C. Array of structure D. pointers to structure

11. Predict, which type of linked list occupies more memory? [ ]

    a. A.SLL B. DLL C.CLL D.None

12. Compute how many pointers need to modify in inserting a node at the beginning of the single linked list

    a. A. 1 B. 2 C. 3 D. 0 [ ]

13. What does the following function do for a given Linked List with first node as head? [ ]

```
    void fun1(struct node* head)
    {
 if(head == NULL)
```

```
return;
    fun1(head->next);
    printf("%d  ", head->data);
    }
```

a. Prints all nodes of linked lists
b. Prints all nodes of linked list in reverse order
c. Prints alternate nodes of Linked List
d. Prints alternate nodes in reverse order

14. Deleting a node at any position (middle) of the single linked list needs to modify _________ pointers.

   a. A. 1   B. 2   C. 3   D. 0   [ ]

SECTION-B

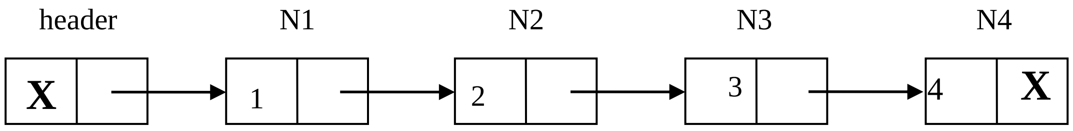
SUBJECTIVE QUESTIONS

1. Write short notes on data structures. Differentiate linear and non linear data structures.
2. What are the different types of Linked Lists? List any four applications of Linked Lists.
3. List the advantages and disadvantages of linked list.
4. Explain how different ways of insertions are performed in singly linked list with suitable examples.
5. Explain about delete operation in single linked list.
6. Write an algorithm for search and traversal operations in single linked list.
7. Compare single linked list and circular single linked list.
8. Explain the following operations on circular linked list.
   A. Insertion B. Deletion C. Search D. Traversal
9. Write an algorithm for following operations on double linked list.
   A. Insertion B. Deletion C. Search D. Traversal

10. Explain insertion , deletion, search and traversal operations on circular double linked list with an example.

11. Compare circular linked list and circular double linked list.

12. Consider the following single linked list.

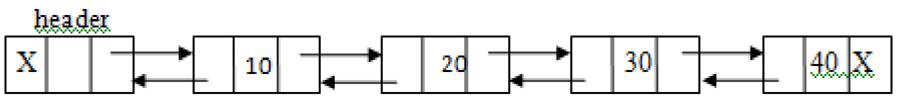


Demonstrate the following operations on this list and draw the updated single linked list after each operation.

1. Insert 5 at end 2. Insert 6 at begin 3.Insert 9 after 2

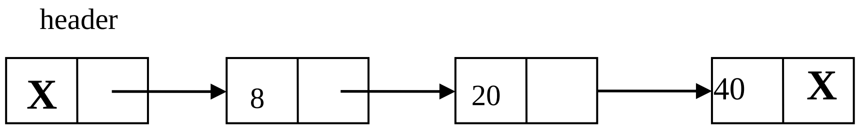
4. Delete 6 5. Delete 5 6. Delete 3.

13. Consider the following double linked list.



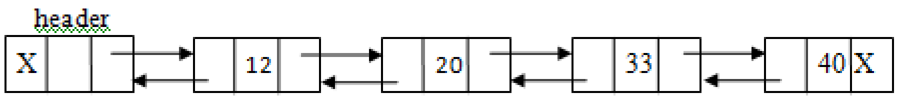
Illustrate the following operations on this list and draw the updated double linked list after each operation.
1. Insert 50 at end 2. Insert 60 at begin 3.Insert 90 after 20
4. Delete 60 5. Delete 50 6. Delete 30

14. Consider the following single linked list.



Insert the following elements into the list 2,15,30,50. Such that the list will be in ascendingorder and draw the updated single linked list after each insertion operation.

15. Consider the following double linked list.



Insert the following elements into the list 2,15,30,50. Such that the list will be in ascendingorder and draw the updated double linked list after each insertion operation.

16. Write algorithm to insert a node into Circular Linked List at ending. **(April 2019).**

17. Illustrate deletion of a node from Double Linked List at beginning and write algorithm to it. **(April 2019).**

18. Consider the elements 85, 26, 42 and 39 are in Circular Double Linked List, where 85 is at beginning and 39 at ending. Now insert 45 at beginning, insert 69 at ending, delete a node from beginning, insert 49 after 42, delete at ending, insert 100 after 85, delete 56. Draw the updated Circular Double Linked List after each operation. **(April 2019).**

19. Write algorithm to delete a node from Single Linked List at Any position. **(April 2019).**

SECTION-C

QUESTIONS AT THE LEVEL OF GATE

1. Consider the function f defined below. **[CS GATE 2003]**

```
struct item
{
  int data;
  struct item * next;
};
int f(struct item *p)
{
  return( (p == NULL) | | (p->next == NULL) | | (( P->data <= p->next->data)
&& f(p->next)) );
}
```

For a given linked list p, the function f returns 1 if and only if

a) the list is empty or has exactly one element

b) the elements in the list are sorted in non-decreasing order of data value

c) the elements in the list are sorted in non-increasing order of data value

d) not all elements in the list have the same data value.

2. A circularly linked list is used to represent a Queue. A single variable p is used to access the Queue. To which node should p point such that both the operations enQueue and deQueue can be performed in constant time?

[CS GATE 2004]

a) rear node b)front node

c)not possible with a single pointer d)node next to front

3. In the worst case, the number of comparisons needed to search a singly linked list of length n for a given element is

[CS GATE 2002]

a) log 2 n b) n/2 c) log 2 n – 1 d) n

STACKS

UNIT-II

Assignment-Cum-Tutorial Questions

SECTION-A

Objective Questions

1. To add and remove nodes from a stack __________ access is used. [ ]
   A. LIFO, Last In First Out B. FIFO, First In First Out
   C. FILO, First in Last Out D. Both A and C
2. Which one of the following is an application of Stack Data Structure? [ ]
   A. Arithmetic Expression Evaluation.
   B. Factorial calculation
   C. Reversing the given String
   D. All of the above
3. In linked list implementation of a stack, where does a new element be deleted? [ ]
   A. At the head of linked list
   B. At the tail of the linked list
   C. At the centre position in the linked list
   D. None of the above
4. What is the time complexity of pop() operation when the stack is implemented using an array? [ ]
   A.O(1) B. O(n) C. O(logn) D.O(nlogn)
5. Which of the following is true about linked list implementation of stack?[ ]
   A. In push operation, if new nodes are inserted at the beginning of linked list, then in pop operation, nodes must be removed from end.
   B. In push operation, if new nodes are inserted at the end, then in pop operation, nodes must be removed from the beginning.
   C. Both of the above
   D. None of the above
6. Which of the following permutation can be obtained in the same order using a stack assuming that input is the sequence 5, 6, 7, 8, 9 in that order?
   A. 7, 8, 9, 5, 6 B. 5, 9, 6, 7, 8 [ ]
   C. 7, 8, 9, 6, 5 D. 9, 8, 7, 5, 6

7. If the sequence of operations – push (1), push (2), pop, push (1), push (2), pop, pop, pop, push (2), pop are performed on a stack, the sequence of popped out values [ ]
   A. 2,2,1,1,2 B. 2,2,1,2,2
   C. 2,1,2,2,1 D. 2,1,2,2,2
8. The postfix form of the expression (A+ B)*(C*D- E)*F / G is? [ ]
   A. AB+ CD*E – FG /** B. AB + CD* E – F **G /
   C. AB + CD* E – *F *G / D. AB + CDE * – * F *G /
9. The postfix form of A*B+C/D is? [ ]
   A. *AB/CD+ B. AB*CD/+
   C. A*BC+/D D. ABCD+/*
10. The prefix form of A-B/ (C * D ^ E) is? [ ]
   A. -/*^ACBDE B.-ABCD*^DE
   C. -A/B*C^DE D. -A/BC*^DE
11. The result of evaluating the postfix expression 5,4,6,+,*,4,9,3,/,+,* is? [ ]
   A. 600 B. 350 C. 650 D. 588
12. Which of the following data structures can be used for parentheses matching? [ ]
   A. n-ary tree B. queue C. priority queue D. stack

SECTION-B

SUBJECTIVE QUESTIONS

1. Define what is stack? Why do we use stack? What are the operations performed on stacks?
2. List out Applications of Stacks? Explain Stack operations using Arrays.
3. Implement Stack using Linked List?
4. What is the prefix and post fix notation of (a + b) * (c + d) ?
5. Convert the expression **(a+b)/d-((e-f)%g)** into reverse polish notation using stack and show the contents of stack for every operation.
6. Evaluate the expression **12/3*6+6-6+8%2** using stack.
7. Convert the expression **a+b*c/d%e-f** into postfix expression using stack.
8. Explain in detail about the Factorial Calculation with an example?.

SECTION-C

QUESTIONS AT THE LEVEL OF GATE

1. To evaluate an expression without any embedded function calls: [ ]

A. One stack is enough [CS GATE-2014]
B. Two stacks are needed
C. As many stacks as the height of the expression tree are needed
D. A Turing machine is needed in the general case

2. Assume that the operators +, -, × are left associative and ^ is right associative. The order of precedence (from highest to lowest) is ^, x , +, -. The postfix expression corresponding to the infix expression a + b × c - d ^ e ^ f is [ ]

   A. abc × + def ^ ^ - [CS GATE-2016]

   B. abc × + de ^ f ^ -

   C. ab + c × d - e ^ f ^

   D. - + a × bc ^ ^ def

3. The following postfix expression with single digit operands is evaluated using a stack: [ ]**[CS GATE-2015]**

   8 2 3 ^ / 2 3 * + 5 1 * -

Note that ^ is the exponentiation operator. The top two elements of the stack after the first * is evaluated are:

A. 6, 1 B. 5, 7 C. 3, 2 D. 1, 5

4. A single array A[1..MAXSIZE] is used to implement two stacks. The two stacks grow from opposite ends of the array. Variables top1 and top2 (topl< top 2) point to the location of the topmost element in each of the stacks. If the space is to be used efficiently, the condition for "stack full" is **[CS GATE CS 2004]**

   A. (top1 = MAXSIZE/2) and (top2 = MAXSIZE/2+1) [ ]
   B. top1 + top2 = MAXSIZE
   C. (top1= MAXSIZE/2) or (top2 = MAXSIZE)
   D. top1= top2 -1

QUEUES

UNIT-III
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. To add and remove nodes from a queue __________ principle is used.[ ]
   A. LIFO, Last In First Out B. FIFO, First In First Out
   C. Both a and b D. None
2. Which one of the following is an application of Queue Data Structure? [ ]
   A. When a resource is shared among multiple consumers.
   B. When data is transferred asynchronously
   C. Load Balancing
   D. All of the above
3. Which of the following is not the type of queue? [ ]
   A.Ordinary queue B. Single ended queue
   C. Circular queue D. Priority queue
4. What is the need for a circular queue? [ ]
   A. effective usage of memory B. easier computations
   C. all of the mentioned D. none
5. What is the space complexity of a linear queue having n elements? [ ]
   A. O(n) B. O(nlogn) C. O(logn) D. O(1)
6. In linked list implementation of a queue, where does a new element be deleted? [ ]
   A. At the head of linked list
   B. At the tail of the linked list
   C. At the centre position in the linked list
   D. None of the above
7. In a circular queue, how do you increment the rear end of the queue? [ ]
   A. rear++ B. (rear+1) % CAPACITY
   C. (rear % CAPACITY)+1 D.rear–
8. In linked list implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into a NONEMPTY queue? [ ]
   A. Only front pointer B. Only rear pointer

C. Both front and rear pointer D. None of the mentioned

9. The value of REAR is increased by 1 when ……. [ ]

A. An element is deleted in a queue
B. An element is traversed in a queue
C. An element is added in a queue
D. None

10. Which of the following is true about linked list implementation of queue? [ ]

A. In push operation, if new nodes are inserted at the beginning of linked list, then in pop operation, nodes must be removed from end.
B. In push operation, if new nodes are inserted at the end, then in pop operation, nodes must be removed from the beginning.
C. Both of the above
D. None of the above

11. How many stacks are needed to implement a queue. Consider the situation where no other data structure like arrays, linked list is available to you. [ ]

A.1 B.2 C.3 D.4

12. How many queues are needed to implement a stack. Consider the situation where no other data structure like arrays, linked list is available to you. [ ]

A.1 B.2 C.3 D.4

13. If the elements “A”, “B”, “C” and “D” are placed in a queue and are deleted one at a time, in what order will they be removed? [ ]

A.ABCD B.DCBA C.DCAB D.ABCD

14. A circular queue is implemented using an array of size 10. The array index starts with 0, front is 6, and rear is 9. The insertion of next element takes place at the array index. [ ]

A.0 B.7 C.9 D.10

15. In a queue, the initial values of front pointer rear pointer should be …….. and ……….. respectively. [ ]

A.0 and 1 B.0 and -1 C.-1 and 0 D.1 and 0

SECTION-B

SUBJECTIVE QUESTIONS

1. Define Queue. Discuss various representations of queue.
2. Implement various operations on queue using arrays?
3. Explain various operations on queue using Linked List with an example.
4. What is Circular Queue? Discuss the types of Queues and explain why we are going for circular queue?
5. List out applications of queues.
6. Implement queue using stack with an example.
7. Given an empty Queue, after performing ENQUEUE(4), ENQUEUE(5), DEQUEUE(), ENQUEUE(3), DEQUEUE(), ENQUEUE(6), ENQUEUE(7).Rear and front pointers of queue point to?
8. A circular queue is implemented using an array of size 10. The array index starts with 0, front is 5, and rear is 8. The insertion of next element takes place at the array index. Perform the following operations DEQUEUE(), ENQUEUE(6), DEQUEUE(), ENQUEUE(16), ENQUEUE(70), ENQUEUE(16), ENQUEUE(70). Rear and front pointers of queue point to?

SECTION-C

QUESTIONS AT THE LEVEL OF GATE

1. Dequeue is called [ ] **[CS GATE 2012]**

   A. Double ended queue B.Single ended queue
   C. an operation D.None of the above

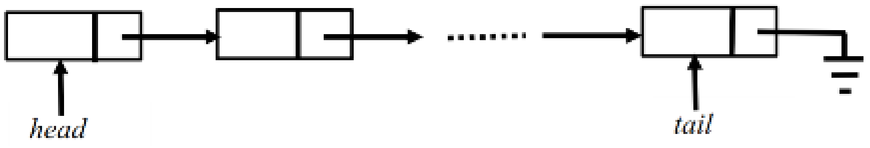
2. Suppose a circular queue of capacity $(n-1)$ elements is implemented with an array of n elements. Assume that the insertion and deletion operations are carried out using REAR and FRONT as array index variables, respectively. Initially, REAR = FRONT = 0. The conditions to detect *queue full* and *queue empty* are [ ] **[CS GATE 2012]**

   A. full: (REAR+1) mod n == FRONT
   empty: REAR == FRONT

B. *full*: REAR == FRONT
empty: (REAR+1) mod *n* == FRONT
C. *full*: (REAR+1) mod *n* == FRONT
empty: (FRONT+1) mod *n* == REAR

D. *full*: (FRONT+1) mod *n* == REAR
empty: REAR == FRONT

3. A queue is implemented using a non-circular singly linked list. The queue has a head pointer and a tail pointer, as shown in the figure. Let *n* denote the number of nodes in the queue. Let enqueue be implemented by inserting a new node at the head, and dequeue be implemented by deletion of a node from the tail.



Which one of the following is the time complexity of the most time-efficient implementation of enqueue and dequeue, respectively, for this data structure? [ ] **[CS GATE 2018]**

A. θ(1),θ(1) B. θ(1),θ(n) C. θ(n),θ(1) D. θ(n),θ(n)

4. A queue is implemented using an array such that ENQUEUE and DEQUEUE operations are performed efficiently. Which one of the following statements is **CORRECT** (n refers to the number of items in the queue)?

[ ]**[CS GATE 2016]**

A. Both operations can be performed in *O* (1) time

B. At most one operation can be performed in *O* (1) time but the worst case time for the other operation will be Ω(n)Ω(n)

C. The worst case time complexity for both operations will be Ω(n)Ω(n)

D. Worst case time complexity for both operations will be Ω(logn)

5. A circular queue has been implemented using a singly linked list where each node consist of a value and a single pointer pointing to the next node. We maintain exactly two external pointers **FRONT** and **REAR** pointing to the front node and rear node of the queue, respectively. Which of the following statement is/ are CORRECT for such a circular queue, so that insertion and deletion operations can be performed in *O* (1) time? **[ ] [CS GATE 2017]**

I. Next pointer of front node point to the rear node.

II. Next pointer of rear node points to the front node.

(A) I only

(B) II only

(C) Both I and II

(D) Neither I nor II

TREES

UNIT-IV
Assignment-Cum-Tutorial Questions

SECTION-A

Objective Questions

1. How many nodes in a tree have ***no*** ancestors? [ ]

   (A) 0 (B) 1 (C) 2 (D) n

2. What is the maximum possible number of nodes in a binary tree at level ***6***? [ ]

   (A) 6 (B) 12 (C) 64 (D) 32

3. A full binary tree with *2n+1* nodes contain_____________? [ ]

   (A) n leaf nodes (B) n non-leaf nodes

   (C) n-1 leaf nodes (D) n-1 non-leaf nodes

4. A full binary tree with *n* leaves contains___________? [ ]

   (A) n nodes (B) log_2^n nodes (C) 2n –1 nodes (D) 2^n nodes

5. The pre-order and post order traversal of a Binary Tree generates the same output. The tree can have maximum___________. [ ]

   (A) Three nodes (B) Two nodes

   (C) One node (D) Any number of nodes

6. The height of a tree is the length of the longest root-to-leaf path in it. The maximum and minimum number of nodes in a binary tree of height 5 are: ______ [ ]

   (A) 63 and 6, respectively (B) 64 and 5, respectively

   (C) 32 and 6, respectively (D) 31 and 5, respectively

7. In order to get the contents of a Binary search tree in ascending order, one has to traverse it in _______ fashion? [ ]

   (A) pre-order (B) in-order (C) post order (D) Not possible

8. A BST is traversed in the following order recursively: **right, root, left.** The output sequence will be in____________ . [ ]

   (A) Ascending order (B) Descending order

   (C) Bitomic sequence (D) No specific order

9. In order to get the information stored in a Binary Search Tree in the descending order, one should traverse it in which of the following order? [ ]

   (A) left, root, right (B) root, left, right

   (C) right, root, left (D) right, left, root

10. What is common in three different types of traversals (Inorder, Preorder and Postorder)? [ ]

    (A) Root is visited before right subtree

    (B) Left subtree is always visited before right subtree

    (C) Root is visited after left subtree

    (D) All of the above

11. A binary search tree contains the numbers 1, 2, 3, 4, 5, 6, 7, 8. When the tree is traversed in pre-order and the values in each node printed out, the sequence of values obtained is 5, 3, 1, 2, 4, 6, 8, 7. If tree is traversed in post-order, the sequence obtained would be____ [ ]

    (A) 8, 7, 6, 5, 4, 3, 2, 1 (B) 1, 2, 3, 4, 8, 7, 6, 5

    (C) 2, 1, 4, 3, 6, 7, 8, 5 (D) 2, 1, 4, 3, 7, 8, 6, 5

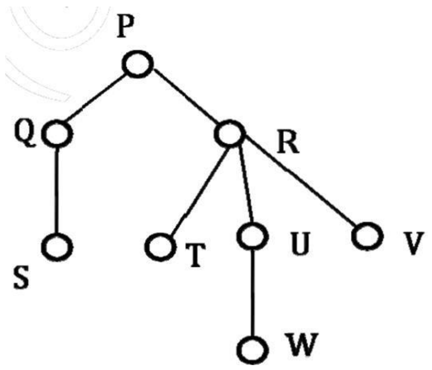
12. Suppose that we have numbers between 1 and 100 in a binary search tree and want to search for the number 55. Which of the following sequences CANNOT be the sequence of nodes examined? [ ]

    (A) {10, 75, 64, 43, 60, 57, 55} (B) {90, 12, 68, 34, 62, 45, 55}

    (C) {9, 85, 47, 68, 43, 57, 55} (D) {79, 14, 72, 56, 16, 53, 55}

13. Consider the following rooted tree with the vertex P labeled as root. The order in which the nodes are visited during in-order traversal is________? [ ]

    (A) SQPTRWUV (B) SQPTURWV (C) SQPTWUVR (D) SQPTRUWV



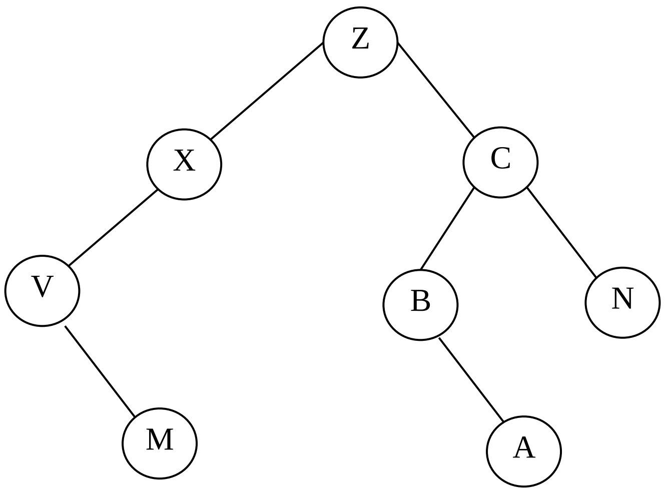
Explanation:

The only confusion in this question is, there are 3 children of R. So when should R appear – after U or after T? There are two possibilities: SQPTRWUV and SQPTWURV. Only 1st possibility is present as an option A, the 2nd possibility is not there. Therefore option A is the right answer.

SECTION-B

Descriptive Questions

1. Write recursive algorithms for Binary Search Tree Traversals.
2. Define Threaded binary tree. Give an example.
3. What is the inorder, preorder and postorder for the following binary tree?



4. Construct Binary Tree for the following tree traversals.

Inorder:	W U R O P I T Y E
Preorder:	P O U W R I Y T E

What is the **Post order** traversal for the above constructed binary tree?

5. Construct Binary Tree for the following tree traversals.

Inorder: N Z V A M C B S X D

Postorder: Z A V N C S D X B M

What is the **Preorder** traversal for the above constructed binary tree?

6. Create Binary Search with the following elements.

   20 30 15 25 42 61 72 18 10 8

What is the **Inorder** traversal for the above constructed Binary Search tree?

7. Insert the following elements into an empty Binary search tree(BST)

   100 90 110 80 95 125 115 108 104 76 49 62

Show each step of insertion. What is the **Inorder** traversal for the above constructed BST?

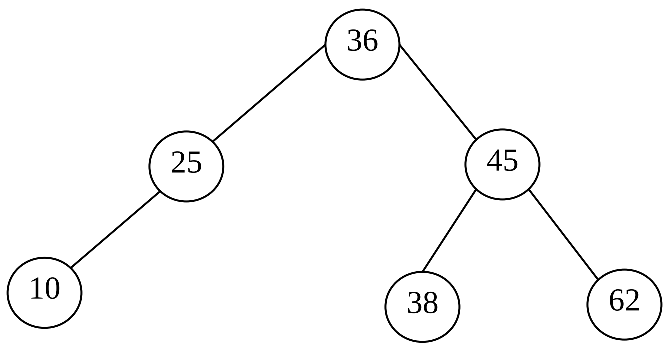
8. Create Binary Search with the following elements.

   60 50 70 45 90 80 30 35 20 25 100 90 70

What is the **Inorder** traversal for the above constructed Binary Search tree?

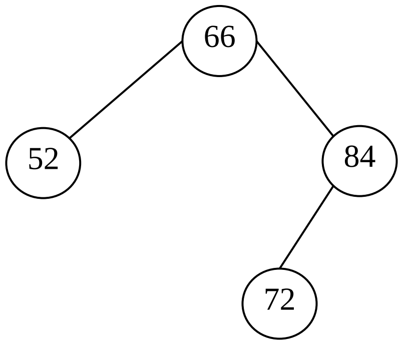
Delete node 30, 100, 80 from the constructed BST.redrwa BST after every delete operation.

9. Consider the following Binary Search Tree and perform the following sequence of operations.



   Insert the elements 55, 68, 49, 18, 28, 27, 30. Now **delete** the elements 55, 45, 36, 10 and 18. Finally what is the root node?

10. Consider the following Binary Search Tree and perform the following sequence of operations.



Insert the elements 89, 46, 48, 26, 76, 98, 100. Now **delete** the elements 84, 48, 52 and 66. Finally what is the root node?

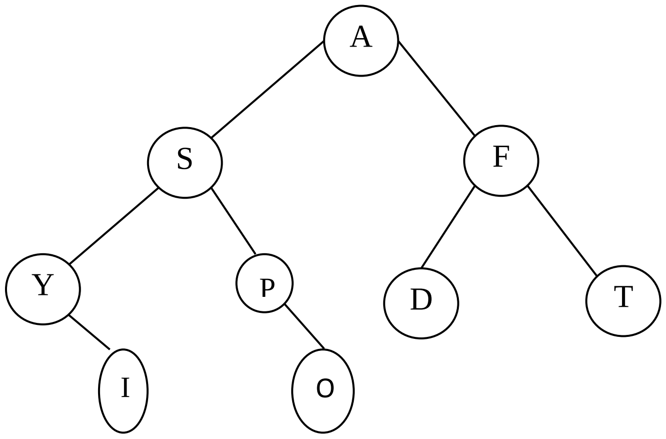
11.Write algorithm to search for an element in a Binary Search Tree. **(April 2019)**

12. Illustrate Linked List representation of Binary Tree. **(April 2019)**

13. Create Binary Tree with the following inorder, postorder and obtain preorder for binary tree. **(April 2019)**

Inorder: J T C D N U F Q T S O P Postorder: J C T N D F T O P S Q U

14. What are different tree traversal techniques? Write them for the following binary tree.



15. Construct a binary search tree with the following elements (taken in the order given) and find the number of nodes in the right sub tree of the constructed binary search tree.

76, 92, 11, 68, 80, 120, 22, 70, 110, 50, 85, 99

16. Construct a binary tree with the following tree traversals. (4M)

Inorder: A M R Y P O E I L C

Preorder: Y M A R E O P C I L

17. How to represent a binary tree using an array. Explain with suitable example.

Section - C

1. Consider a binary tree T that has 200 leaf nodes. Then, the number of nodes in T that have exactly two children are? **(GATE 2016) [ ]**

   (A) 201 (B) 100 (C) 199 (D) 50

2. The maximum number of binary trees that can be formed with three unlabelled nodes is: ________ **(GATE 2007)** [ ]

   (A) 1 (B) 5 (C) 4 (D) 3

3. The height of a binary tree is the maximum number of edges in any root to leaf path. The maximum number of nodes in a binary tree of height h is: **(GATE 2007)**[ ]

   (A) 2^h -1 (B) 2^(h-1) – 1 (C) 2^(h+1) -1 (D) 2*(h+1)

4. The inorder and preorder traversal of a binary tree are ***d b e a f c g*** and ***a b d e c f g***, respectively. The postorder traversal of the binary tree is: **(GATE 2007)**[ ]

   (A) d e b f g c a (B) e d b g f c a (C) e d b f g c a

   (D) d e f g b c a

5. Consider the label sequences obtained by the following pairs of traversals on a labelled binary tree. Which of these pairs identify a tree uniquely? **(GATE CS 2004)** [ ]

   i) preorder and postorder

   ii) inorder and postorder

   iii) preorder and inorder

   iv) level order and postorder

   (A) (i) only (B) (ii), (iii) only (C) (iii) only (D) (iv) only

6. Let **LASTPOST**, **LASTIN** and **LASTPRE** denote the last vertex visited in a postorder, inorder and preorder traversal. Respectively, of a complete binary tree. Which of the following is always true? **(GATE CS 2000)** [ ]

   (A) LASTIN = LASTPOST (B) LASTIN = LASTPRE

   (C) LASTPRE = LASTPOST (D) None of the above

7. While inserting the elements 71, 65, 84, 69, 67, 83 in an empty binary search tree (BST) in the sequence shown, the element in the lowest level is? **(GATE 2015)[ ]**

(A) 65 (B) 67 (C) 69 (D) 83

8. Suppose the numbers 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 are inserted in that order into an initially empty *binary search tree*. The binary search tree uses the usual ordering on natural numbers. What is the in-order traversal sequence of the resultant tree?

(GATE CS 2003)[ ]

(A) 7 5 1 0 3 2 4 6 8 9 (B) 0 2 4 3 1 6 5 9 8 7

(C) 0 1 2 3 4 5 6 7 8 9 (D) 9 8 6 4 2 3 0 1 5 7

9. Which of the following is/are *correct* inorder traversal sequence(s) of binary search tree(s)? **(GATE 2016) [ ]**

I. 3, 5, 7, 8, 15, 19, 25

II. 5, 8, 9, 12, 10, 15, 25

III. 2, 7, 10, 8, 14, 16, 20

IV. 4, 6, 7, 9 18, 20, 25

(A) I and IV only (B) II and III only (C) II and IV only (D) II only

10. Postorder traversal of a given binary search tree, T produces the following sequence of keys ***10, 9, 23, 22, 27, 25, 15, 50, 95, 60, 40, 29.*** Which one of the following sequences of keys can be the result of an in-order traversal of the tree T?

(GATE CS 2004) [ ]

(A) 9, 10, 15, 22, 23, 25, 27, 29, 40, 50, 60, 95

(B) 9, 10, 15, 22, 40, 50, 60, 95, 23, 25, 27, 29

(C) 29, 15, 9, 10, 25, 22, 23, 27, 40, 60, 50, 95

(D) 95, 50, 60, 40, 27, 23, 22, 25, 10, 9, 15, 29

11. The following numbers are inserted into an *empty binary search tree* in the given order: 10, 1, 3, 5, 15, 12, 16. What is the height of the binary search tree (the height is the maximum distance of a leaf node from the root)? **(GATE CS 2004)**[ ]

(A) 2 (B) 3 (C) 4 (D) 6

12. The *preorder* traversal sequence of a *binary search tree* is 30, 20, 10, 15, 25, 23, 39, 35, and 42. Which one of the following is the postorder traversal sequence of the same tree?

(GATE CS 2013)[ ]

(A) 10, 20, 15, 23, 25, 35, 42, 39, 30 (B) 15, 10, 25, 23, 20, 42, 35, 39, 30

(C) 15, 20, 10, 23, 25, 42, 35, 39, 30 (D) 15, 10, 23, 25, 20, 35, 42, 39, 30

13. Let T be a binary search tree with 15 nodes. The minimum and maximum possible heights of T are: **(GATE-CS-2017)[ ]**

Note: The height of a tree with a single node is 0.

(A) 4 and 15 respectively (B) 3 and 14 respectively

(B) 4 and 14 respectively (D) 3 and 15 respectively

14. Let T be a tree with 10 vertices. The sum of the degrees of all the vertices in T is _____.

(GATE-CS-2017 - Set 1) [ ]

(A) 18 (B) 19 (C) 20 (D) 21

15.The postorder traversal of a binary tree is 8, 9, 6, 7, 4, 5, 2, 3, 1. The inorder traversal of the same tree is 8, 6, 9, 4, 7, 2, 5, 1, 3. The height of a tree is the length of the longest path from the root to any leaf. The height of the binary tree above is ________

(GATE-CS-2018 - Set 1)

(A) 2 (B) 3 **(C) 4** (D) 5

****

SEARCHING AND SORTING

UNIT - V

Assignment-Cum-Tutorial Questions

SECTION -A

Objective Questions

1. The Worst case occur in linear search algorithm when
   A) Item is somewhere in the middle of the array
   B) Item is not in the array at all
   C) Item is the last element in the array
   D) Item is the last element in the array or is not there at all
2. Which of the following is false? [ ]
   A) A linear search begins with the first array element
   B) A linear search continues searching, element by element, either until a match is found or until the end of the array is encountered
   C) A linear search is useful when the amount of data that must be search is small
   D) For a linear search to work, the data in the array must be arranged in either alphabetical or numerical order
3. Which characteristic will be used by binary search but the linear search ignores is [ ]
   A) Order of the elements of the list B) Length of the list
   B) Maximum value in list D) Type of elements of the list
4. Choose the false statement . [ ]
   A) A binary search begins with the middle element in the array.
   B) A binary search continues having the array either until a match is found or until there are no more elements to search.
   C) If the search argument is greater than the value located in the middle of the binary, the binary search continues in the lower half of the array.

D) For a binary search to work, the data in the array must be arranged in either alphabetical or numerical order.

5. Which of the following is *not* a limitation of binary search algorithm? [ ]

A) Must use a sorted array

B) Requirement of sorted array is expensive when a lot of insertion and deletions are needed

C) There must be a mechanism to access middle element directly

D) Binary search algorithm is not efficient when the data elements more than 1500

6. What is the complexity of searching an element from a set of n elements using Binary search algorithm is [ ]

A) O(n) B) O(log n) C) O(n 2) D) O(n log n)

7. The process of arranging values either in ascending or decending order is called as ________.

8. In which sorting technique , consecutive adjacent pairs of elements in the array are compared with each other. [ ]

A) Bubble sort B) Selection Sort C) Insertion Sort D) None

9. Identify the number of comparisons required to sort a list of 10 numbers in pass 2 by using Bubble Sort is___________. [ ]

A) 10 B) 9 C) 8 D) 7

10. Consider an array of elements arr[5]= {99,22,55,44,33}, what are the steps done while doing bubble sort in the array [ ]

A) 22 55 44 33 99 33 22 44 99 55 22 44 99 33 55 44 22 55 33 99

B) 22 55 44 33 99 22 44 33 55 99 22 33 44 55 99 22 33 44 55 99

C) 55 44 33 99 22 44 22 33 99 55 55 33 99 22 44 99 55 44 33 22

D) None of the above

11. Which sorting technique sorts a list of elements by moving the current data element past the already sorted values with the preceding value until it is in its correct place. [ ]

A) Insertion sort B) Bubble Sort C) Selection Sort D) None

12. Identify t h e number of passes required by insertion sort for the list **size 15.** [ ]

A) 15 B) 16 C) 14 D) 13

13. Which of the following sorting algorithms in its implementation gives best performance when applied on an array which is sorted or almost sorted (maximum 1 or two elements are misplaced). [ ]

A) Insertion sort B) Bubble Sort C) Selection Sort D) None

14. Consider an array of elements arr[5]= {5,4,3,2,1} , what are the steps of iinsertions done while doing insertion sort in the array. [ ]

A) 4 5 3 2 1	3 4 5 2 1	2 3 4 5 1	1 2 3 4 5
B) 5 4 3 1 2	5 4 1 2 3	5 1 2 3 4	1 2 3 4 5
C) 4 3 2 1 5	3 2 1 5 4	2 1 5 4 3	1 5 4 3 2
D) 4 5 3 2 1	2 3 4 5 1	3 4 5 2 1	1 2 3 4 5

15. Consider the array A[]={6,4,8,1,3} apply the *insertion sort* to sort the array. Consider the cost associated with each sort is 25 rupees, what is the total cost of the insertion sort when element 1 reaches the first position of the array? [ ]

A) 50 B) 25 C) 75 D) 100

16. Consider a situation where swap operation is very costly. Which of the following sorting algorithms should be preferred so that the numbers of swap operations are minimized in general? [ ]

A) Bubble Sort B) Selection Sort C) Insertion Sort D) None

17. Which one of the following in -place sorting algorithms needs the minimum number of swap [ ]

A) Insertion Sort B) Bubble Sort C) Selection Sort D) All of the above

18. Discover the comparisons needed to sort an array of length 5 if a straight selection sort is used and array is already in the opposite order? [ ]

A) 1 B) 10 C) 5 D) 20

19. Which of the following is the advantage o f bubble sort over other sorting techniques? [ ]

A) It is faster

B) Consumes less memory

C) Detects whether the input is already sorted

D) All of the mentioned

20. Which of the following is not a stable sorting algorithm ? [ ]

A) Insertion sort B) Merge sort C) Selection sort D) Bubble sort

SECTION-B

Descriptive Questions

1. Write recursive algorithm to implement linear search. **(April 2019)**
2. Apply linear search for an element 18 and 100 in the following list.
   36, 72, 19, 45, 18, 22, 12, 55
3. Write non recursive algorithm to implement binary search. **(April 2019)**
4. Explain binary search to search 54 and 100 in the following list.
   13, 27, 91, 54, 81, 6, 51, 59, 45, 69
5. Enlighten fibonacci search and apply fibonocci search for an element 54 and 100 in the following list.13, 27, 91, 54, 81, 6, 51, 59, 45, 69
6. How do you differentiate linear search with binary search, explain with an example.
7. Use bubble sort technique to sort the following elements.
   30, 52, 29, 87, 63, 27, 19, 54,85
8. Write insertion sort algorithm. Apply insertion sort to sort the elements 14, 31, 12, 18, 25, 16, 44, 32, 45, 15. **(April 2019)**
9. Apply selection sort for the following elements.
   36, 12, 81, 45, 90, 27, 72, 18, 64, 92
10. Sort the elements 23, 17, 19, 54, 18, 16, 55, 49, 45, 29 using Quick sort. **(April 2019)**
11. Interpret how merge sort, sorts the following elements
    65, 34, 56, 23, 87, 75,15, 67, 19,12
12. Illustrate how radix sort sorts the given list of elements

366, 9120, 235, 10, 4923, 6134, 169, 181, 34,97,89

****

GRAPHS

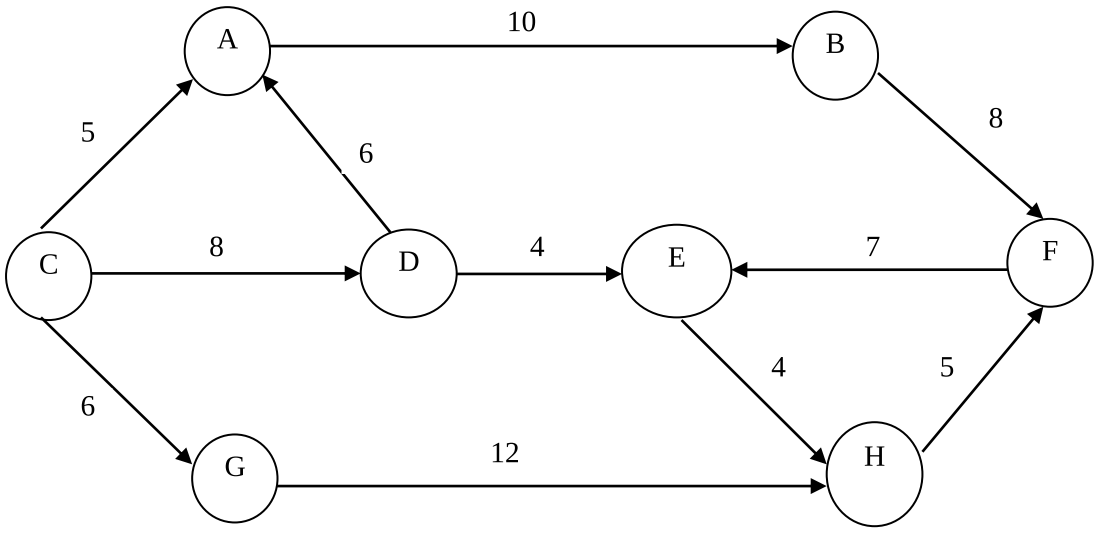
UNIT-VI

Assignment-Cum-Tutorial Questions

SECTION-A

Objective Questions

Consider the following graph and answer to the questions 1 to 6



1. The above graph is ____________ [ ]

   A) Complete Graph B) Weighted Graph

   C) Multi Graph D) None of the above

2. In the above graph which of the following is a pendant vertex? [ ]

   A) vertex B B) vertex D C) vertex E D) None of the above

3. In the above graph indegree and outdegree of vertex H is ___ [ ]

   A) indegree - 2 outdegree – 0 B) indegree - 3 outdegree - 0

   C) indegree - 3 outdegree – 1 D) indegree - 2 outdegree - 1

4. The above graph is a __________ [ ]

   A) Connected Graph B) Complete Graph

C) Cyclic graph D) None of the above

5. The node A is adjacent to __________ node. [ ]

   A) B B) C C) D D) None

6. In a graph if e=(u,v) means ……. [ ]

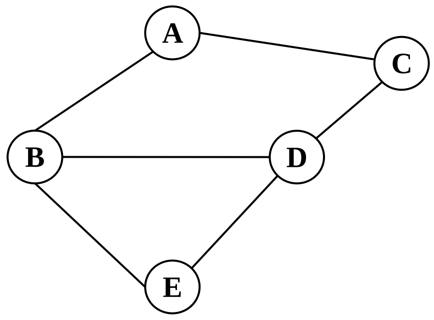
   A) u is adjacent to v but v is not adjacent to u.

   B) e begins at u and ends at v

   C) u is node and v is an edge.

   D) both u and v are edges.

Consider the following graph to answer the questions 7 to 9



7. The above graph is a _____________ [ ]

   A) Weighted graph B) Directed graph C) Cyclic Graph D) None

8. The adjacent vertices of node A are _________ [ ]

   A) B, D, E B) B, D, C C) E, D D) None

9. The above graph is a ______________ [ ]

   A) Connected graph B) Complete graph C) Both A&B D) None

10. An adjacency matrix representation of a graph cannot contain information of ______

    (A) Nodes (B) edges [ ]

    (C) Direction of edges (D) parallel edges

11. How many undirected graphs (not necessarily connected) can be constructed out of a given set V= {V 1, V 2,…V n} of n vertices ? [ ]

    (A) n(n-l)/2 (B) 2^n (C) n! (D) 2^(n(n-1)/2)

12. The data structure required for Breadth First Traversal on a graph is _________ [ ]

    (A) Queue (B) Stack (C) Array (D) Tree

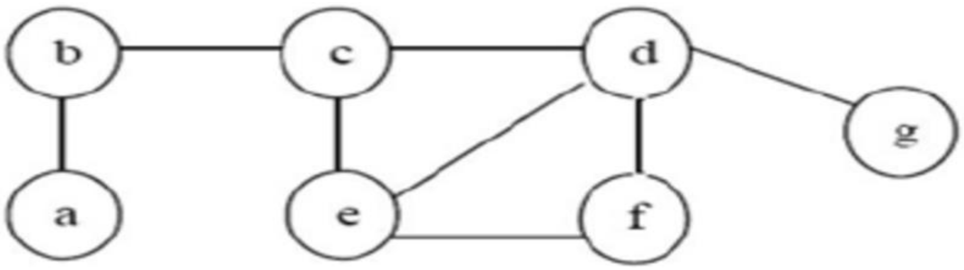
13. The minimum number of edges in a connected cyclic graph on n vertices is___? [ ]

    (A) n-1 (B) n (C) n+1 (D) None of the above

14. The number of simple graphs on n labled vertices is___ [ ]

    (A) n (B) n(n-1)/2 (C) $2^{n(n-1)/2}$ (D) n(n+1)/2

15. The BFS Algorithm has been implemented using Queue Data Structure. Which one is not possible order of visiting nodes(source vertex b) for the following graph [ ]



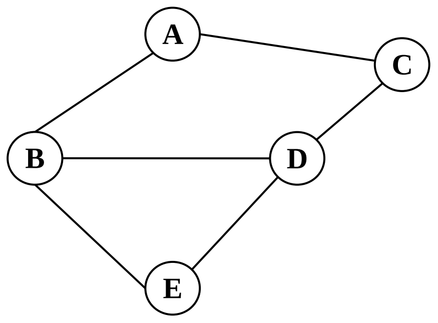
    (A) b a c e d f g (B) b c a d e f g (C) b c a e d f g (D) b a c d f e g

SECTION-B

Descriptive Questions

1. Define Graph. List different real time applications of graph.
2. Write adjacency matrix representation of graph with an example.
3. Explicate Linked List representation of graph with an example.
4. Explain following graph operations with an example
    a. Vertex Insertion
    b. Vertex Deletion
    c. Find Vertex
    d. Edge Addition
    e. Edge Deletion

5. Perform following operations on this graph



   a) Insert vertex 'F'
   b) Insert edge between 'E' and 'F'
   c) Delete vertex 'D'
   d) Delete edge between 'A' and 'C'

6. Which Data structure is used to implement DFS traversal of a graph? Describe algorithm to implement DFS traversal of a graph. With an example explain Breadth first Traversal algorithm. **[April 2019]**

7. Consider the graph given below
   a) Write the adjacency matrix of G1.
   b) Give Linked list representation of G1.
   c) Give Set representation of G1.
   d) Is the graph complete?
   e) Is the graph strongly connected?
   f) Find out the degree of each node.
   g) Is the graph regular?

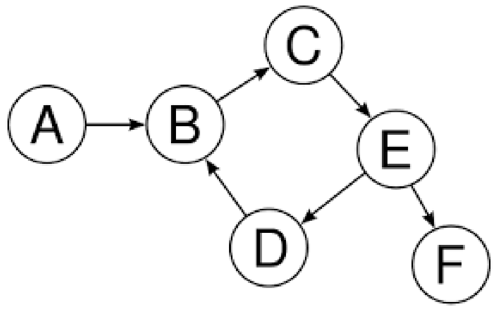
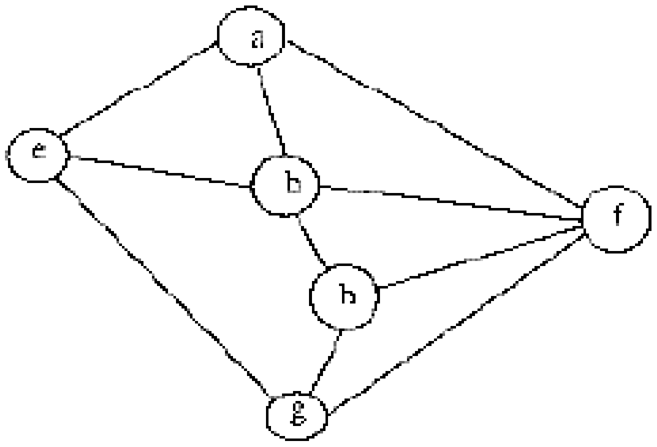


Fig. Graph G1

8. Consider the following adjacency matrix, draw the weighted graph.

$$\begin{pmatrix} 0 & 4 & 0 & 2 & 0 \\ 0 & 0 & 0 & 7 & 0 \\ 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

9. Consider the following graph

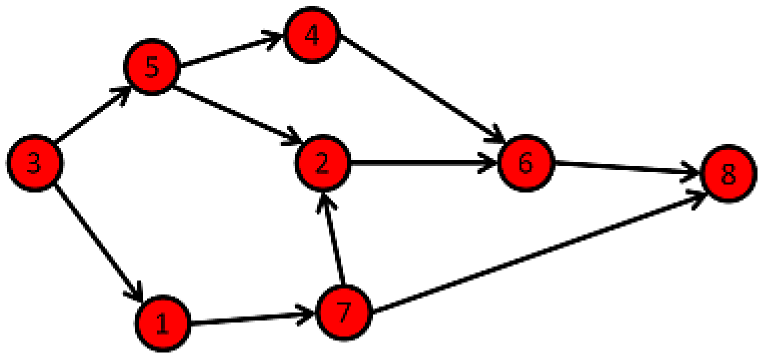


Among the following sequences

i) a b e g h f   ii) a b f e h g   iii) a b f h g e   iv) a f g h b e

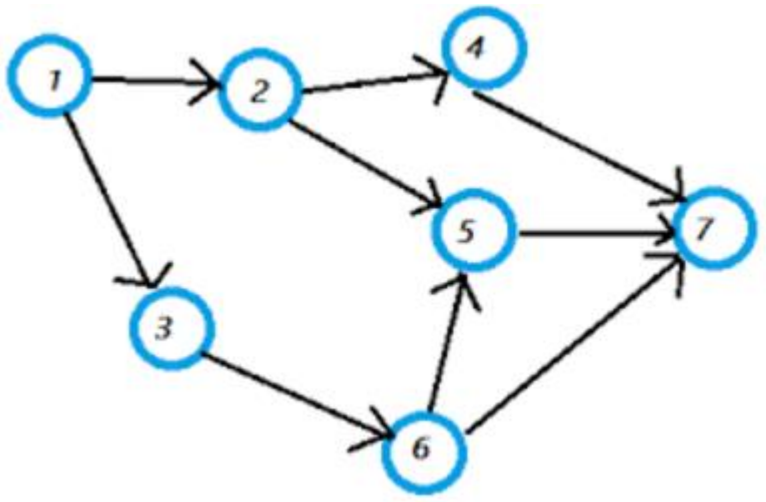
Which are depth first traversals of the above graph?

10. Consider the following graph



What is breadth first traversal of the above graph if starting vertex is 3?

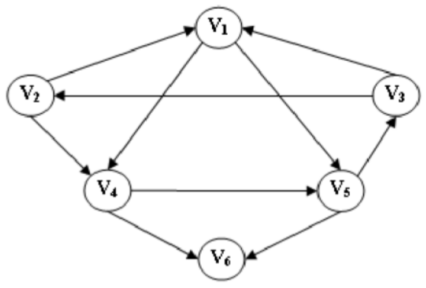
11. Consider the following graph



What is the depth first traversal of the above graph if starting vertex is 1?

12. Identify the relationship between Complete graph and Connected Graph and give examples ? **[April 2019]**

13. What data structure is used for BFS traversal of a graph? With neat sketch obtain BFS traversal of the following graph with V_1 as starting vertex. **[April 2019]**



Section C

Questions asked in GATE (CSE)

1. Which of the following statements is/are TRUE for undirected graphs?
   a. P: Number of odd degree vertices is even.

b. Q: Sum of degrees of all vertices is even. **(GATE 2013)**
[ ]

A) P only B) Q only C) Both P and Q D) Neither P nor Q

2. What is the largest integer m such that every simple connected graph with n vertices and n edges contains at least m different spanning trees? **(GATE 2016)** [ ]

   A) 1 B) 2 C) 3 D) N

3. Given an undirected graph G with V vertices and E edges, the sum of the degrees of all vertices is

   (GATE 2013) [ ]

   A) E B) 2E C) V D) 2V

4. Let G be a simple undirected planar graph on 10 vertices with 15 edges. If G is a connected graph, then the number of bounded faces in any embedding of G on the plane is equal to_________

   (GATE 2012) [ ]

   A) 3 B) 4 C) 5 D) 6

5. Which one of the following is TRUE for any simple connected undirected graph with more than 2 vertices?

   (GATE 2009) [ ]

   A) No two vertices have the same degree.
   B) At least two vertices have the same degree.
   C) At least three vertices have the same degree.
   D) All vertices have the same degree.

6. Consider an undirected random graph of eight vertices. The probability that there is an edge between a pair of vertices is 1/2. What is the expected number of unordered cycles of length three?

   (GATE 2007) [ ]

   A) 1/8 B) 1 C) 7 D)8

7. Which of the following data structure is useful in traversing a given graph by breadth first search?
   (GATE 2005) [ ]

   A) STACK   B) LIST   C) QUEUE   D) NONE

****