COMPUTER SCIENCE

B. Sc. I
Semester-II
2023-2024 (CBCS)

1CS2 : Data Structure & OOPS

Unit-II : Queue, Linked List & Trees

PROF. V. V. AGARKAR
Assistant Professor & Head
Department of Computer Science

Shri. D. M. Burungale Science & Arts College, Shegaon, Dist. Buldana

UNIT – II

Syllabus: **Queues**: Definition and concepts, Memory Representations, Operations: Traversing, Insertion, Deletion. Types of Queue. **Linked List**: Definition and concepts, Memory Representations, Types of Linked List, and Operations: Traversing, Insertion, Deletion. **Tree**: Definition and Terminologies, Memory Representations of Trees, Types of Trees : Binary Trees, Complete Binary Trees, Binary Search Trees, Traversing : Preorder, Inorder, Postorder, Insertion, Deletion.

Queue

A ***queue*** is a liner list of elements in which deletions can take place only at one end, called the ***front***, and insertions can take place only at the other end, called the ***rear***. Queues are also called First-In-First-Out (**FIFO**), since the first element inserted in a queue will be the first element deleted from the queue. In other words, the elements can be removed in the same order in which they are inserted into the queue.

Following figure (1) shows Insertion and Deletion operations on Queue.

1	2	3	4	5

Initially queue empty
Front=Rear=0
(a)

AA				
1	2	3	4	5

Element AA inserted
Front=1, Rear=1
(b)

AA	BB			
1	2	3	4	5

Element BB inserted
Front=1, Rear=2
(c)

AA	BB	CC		
1	2	3	4	5

Element CC inserted
Front=1, Rear=3
(d)

	BB	CC		
1	2	3	4	5

Element AA deleted
Front=2, Rear=3
(e)

		CC		
1	2	3	4	5

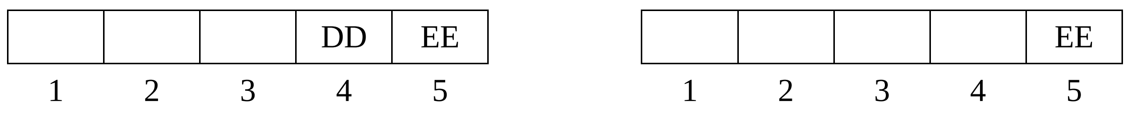
Element BB deleted
Front=3, Rear=3
(f)

		CC	DD	
1	2	3	4	5

Element DD inserted
Front=3, Rear=4
(g)

		CC	DD	EE
1	2	3	4	5

Element EE inserted
Front=3, Rear=5
(h)



Element CC deleted
Front=4, Rear=5
(i)

Element DD deleted
Front=5, Rear=5
(j)

[Fig. 1 : Insertion and Deletion operations on to Queue. **]**

The queues are used in a computer for serving requests on a single shared resource, like a printer, CPU scheduling etc. and also for handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i. e. First come first served.

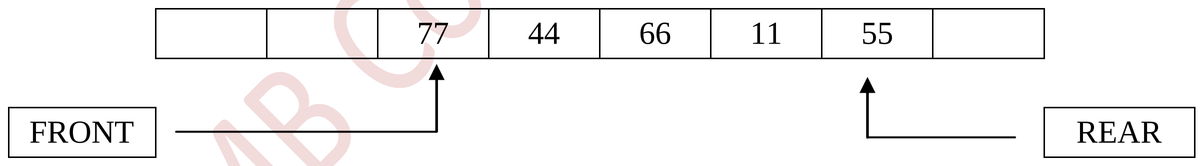
Representation of queues in the Computer memory

There are two major ways to represent a queue –

1) An array implementation
2) A linked list implementation

1) An array representation

In this representation, Queues are represented in the computer by means of a linear array. An array longer than the expected length of the queue is defined. So queue is maintained by linear array and two pointer variables: FRONT, containing the location of the front element of the queue; and REAR, containing the location of the rear element of the queue. Initially, the value of front and queue is -1 which represents an empty queue.



[Fig. 2 : Array representation of Queue. **]**

When an element is inserted into queue the value of REAR is increased by 1 and FRONT is unchanged, if queue is not full. Similarly, when an element is deleted from queue the value of FRONT is increased by 1 and REAR is unchanged, if queue is not empty.

- **Drawback of array implementation**

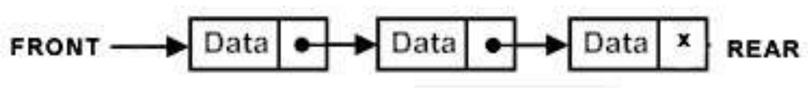
Although, this technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.

i) **Memory wastage:** The space of the array can never be reused to store the elements of that queue.

ii) **Deciding the array size:** An array must be declared large enough so that user can store queue elements as enough as possible.

2) Linked List Implementation

Queue using array is not suitable when we don't know the size of data which we are going to use. A queue can be implemented using linked list. The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data. In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.



[Fig. 3 : Linked List representation of Queue. **]**

The advantage of using linked list over arrays is that it is possible to implement a queue that can grow or shrink as much as needed.

Inserting an element into the queue

Inserting an element into a queue is called "E***nqueue***". While inserting an element into queue, first test whether there is a room in the queue for the new element; if not, then the condition *Overflow* occurs. Following is algorithm to insert an element into the queue:

Algorithm

```
ENQUEUE(QUEUE, FRONT, REAR, N, ITEM)
FRONT and REAR are pointers to front and rear
elements of queue QUEUE. The QUEUE consisting
of N elements. This procedure inserts an
element ITEM at the rear of queue.

1.  [Check for queue overflow]
    If REAR >= N
          Then print "Queue overflow" and exit.

2.  [Increment the REAR pointer by one]
    REAR = REAR + 1

3.  [Insert element ITEM in new REAR position]
    QUEUE[REAR] = ITEM

4.  [Is FRONT pointer properly set?]
     IF FRONT = 0 Then
            FRONT = 1
     End If

5.   Exit.
```

Deleting an element from the queue

Deleting an element from a queue is called "***Dequeue***". While deleting an element from queue, first test whether there is an element in the queue to be deleted; if not, then the condition *Underflow* occurs. Following is algorithm to delete an element from queue:

Algorithm

```
DEQUEUE(QUEUE, FRONT, REAR, ITEM)
FRONT and REAR are pointers to front and rear
elements of QUEUE. This procedure deletes and
returns the first element of the queue. ITEM
is a temporary variable.

1.  [Check for queue underflow]
    If FRONT = 0
          Then print "Queue underflow" and exit.
2.  [Delete element]
    ITEM = QUEUE[FRONT]
3.  [Set FRONT pointer properly]
    IF FRONT = REAR Then
           FRONT = 0
           REAR = 0
    Else
          FRONT = FRONT + 1
    End If
4.  Exit.
```

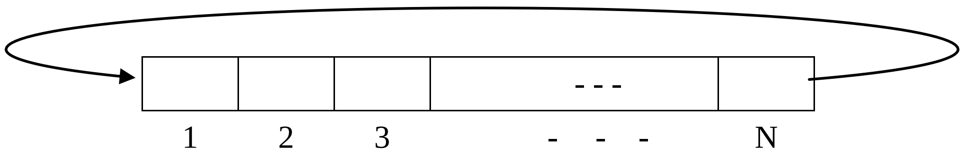
Types of Queues:

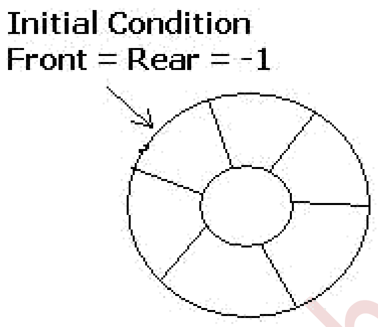
1. Circular queue
2. Dequeue
3. Priority Queue

1. Circular Queue

A circular queue is a particular implementation of a queue in which the last element is connected back to the first element.

Circular queue is a linear data structure. It follows FIFO principle. In circular queue elements are added at the rear end and the elements are deleted at front end. In circular queue, the elements are physically arranged in sequential manner but logically they are treated as circularly arranged. The FRONT and REAR moves in the clockwise direction. Diagrammatically Circular Queue is shown as in Fig. 4.





[Fig. 4: Circular Queue **]**

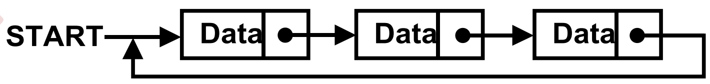
- ***Implementation of circular queue***

Circular Queue can be created in three ways they are:

1) Using single linked list
2) Using double linked list
3) Using arrays

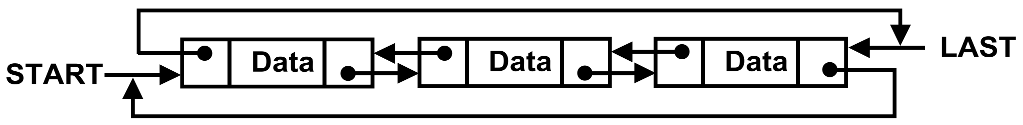
1) Using single linked list

It is an extension for the basic single linked list. In circular linked list instead of storing a Null value in the last node of a single linked list, store the address of the first node (root) forms a circular linked list. Using circular linked list it is possible to directly traverse to the first node after reaching the last node. The following figure shows circular single linked list:



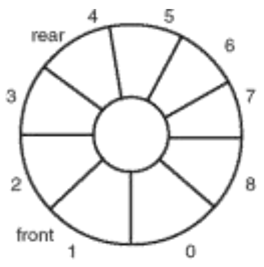
2) Using double linked list

In double linked list the right side pointer points to the next node address or the address of first node and left side pointer points to the previous node address or the address of last node of a list. The following figure shows circular double linked list:



3) Using arrays

In arrays the range of subscripts is 0 to n-1 where n is the maximum size. To make the array as a circular array by making the subscript 0 as the nest address of the subscript n-1 by using formula **subscript = (subscript+1)%maximum** size. In circular queue the front and rear pointer are updated by using above formula. The following figure shows circular array.



2. Dequeue

Deque stands for Double Ended Queue. Deque is a generalized version of Queue data structure that allows insert and delete at both ends. New items can be added at either the front or the rear ends. Likewise, existing items can be removed from either end. However, no element can be added or deleted from the middle. In a deque, two pointers are maintained, **LEFT** and **RIGHT**, which point to either end of the deque. It is also known as a *head-tail linked list*.

Deque is a hybrid linear structure that provides all the capabilities of stacks and queues in a single data structure. Even though the deque can assume many of the characteristics of stacks and queues, it does not require the LIFO and FIFO orderings that are enforced by those data structures.

There are two common ways to implement a deque: with a dynamic circular array or with a doubly linked list.

There are two variations of deque, as follows:

1) Input restricted Deque

   In input restricted Deque, the insertion is allow only one end i.e. RIGHT end but the deletion can made at both the ends.

2) Output restricted Deque

   In output restricted Deque, the deletion is allow only one end i.e. LEFT end but the insertion can made at both the ends.

Important applications of the deque are: storing a web browser's history and storing a software application's list of undo operations.

3. Priority Queue

A priority queue is an extension of queue with some properties. It is like a regular queue, but additionally every element has a priority associated with it. In a priority queue, an element with high priority is served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue.

A priority assigned to each element is represented by an integer value. An element with the highest priority is always removed first. Priority queues often have a fixed size. For example, in an operating system the runnable processes might be stored in a priority queue, where certain system processes are given a higher priority than user processes.

Priority queues are implemented by using Linked list, Binary search tree, or Binary heap. Binary heap implementation is better than BST because it does not support links.

- ***Memory Representation of priority queue***

Priority queues can represented in two ways.

1) **Array or Sequential representation**

With this representation, an array can be maintained to hold the item and its priority value. The element will be inserted at the REAR end as usual. The deletion operation will then be performed in either of the two following ways:

a) Starting from the FRONT pointer, traverse the array for an element of the highest priority. Delete this element from the queue. If this is not the front-most element, shift all its trailing elements after the deleted element one stroke each to fill up the vacant position.

b) Add the elements at the REAR end as earlier. Sort the elements of the queue so that the highest priority element is at the FRONT end. When a deletion is required, delete it from the FRONT end only.

2) **Linked List or One way List representation**

This representation assumes the node structure as shown in following Figure. Every node in the list will have three values: DATA is to store the actual content, PRIORITY is to store the priority value of the item and LINK is to store address of next node. We will consider FRONT and REAR as two pointers pointing the first and last nodes in the queue, respectively.

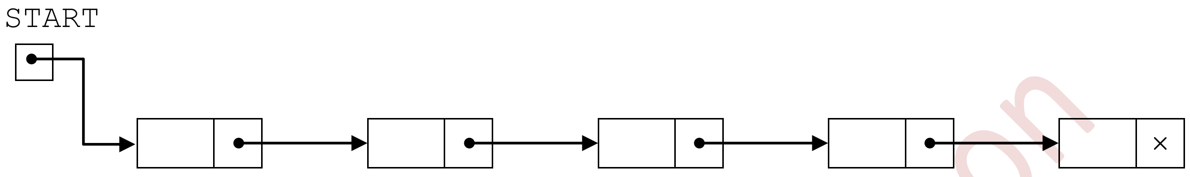
DATA	PRIORITY	LINK

With this structure, to delete an item having priority p, the list will be searched starting from the node under pointer REAR and the first occurring node with PRIORITY = P will be deleted. Similarly, to insert a node containing an item with priority p, the search will begin from the node under the pointer FRONT and the node will be inserted before a node found first with priority value p, or if not found then before the node with the next priority value.

Linked List

A linked list is a dynamic data structure. The number of nodes in a linked list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list. As Linked List is dynamic structure, therefore it can be easily extended or reduced to fit the data set. In the linked list a node can be inserted or deleted at any position.

Elements of the linked list are called ***node***. Nodes are connected to each other with the pointers i.e. every node points to its next node. Each node of a linked list comprises of two parts (fields)- ***Information field*** and ***link*** or ***nextpointer field***. The Information field contains the data of a node and the Link field contains the address of next node in the list. The Link field of the last node contains ***null*** value. Linked list also contains a list pointer variable called ***START*** or NAME -- which contains the address of the first node in the list. The list that has no nodes is called ***null list*** or ***empty list*** and is denoted by null pointer in the variable START.

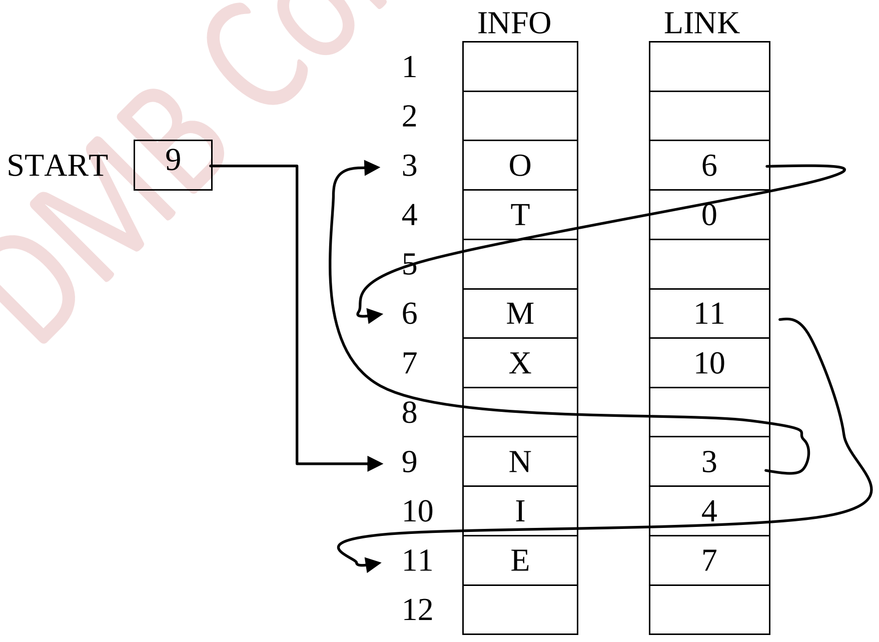


[Fig. 5 : Linked List with 5 nodes **]**

Representation of Linked Lists in memory

Let LIST be a linked list. Then LIST will be maintained in memory by two linear arrays—we will call them here INFO and LINK. The INFO[K] contains the information part of a node of LIST, and LINK[K] contains the nextpointer field of a node of LIST. The linked list LIST also requires a variable name—such as START—which contains the location of the beginning of the list, and a nextpointer field denoted by NULL—which indicates the end of the list. Since the subscripts of the arrays INFO and LINK will usually be positive, we will choose NULL = 0.

Following Fig. 6 shows the array representation of linked list in memory.



[Fig. 6 : Representation of Linked Lists in memory **]**

Traversing Linked List

Traversing a linked list means visit every node in the list beginning from first node to last node of the list exactly once. The following algorithm traverses a LIST.

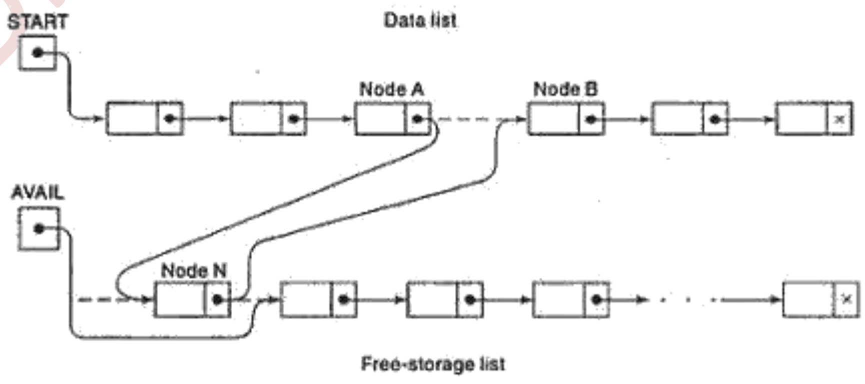
Algorithm

```
Here LIST is a linked list in memory. This
algorithm traverses LIST, applying an
operation PROCESS to each node of LIST.
The variable PTR points to the node
currently being processed.

1. [Initialize pointer PTR]
      Set PTR := START
2. Repeat Steps 3 and 4 while PTR ≠ NULL
3.    [Visit node]
       Apply PROCESS to INFO[PTR]
4.    [PTR points to next node]
       Set PTR := LINK[PTR]
   [End of Step 2 loop]
5. Exit
```

Insertion into Linked List

Let **LIST** be a linked list with successive nodes **A** and **B** as shown in Fig. 7. Suppose a node N is to be inserted into the list between nodes **A** and **B**, the node **A** now points to the new node **N**, and node **N** points to node **B**, to which **A** previously pointed.

The new node **N** will come from the **AVAIL** list. For easier processing, the first node in the **AVAIL** list will be used for the new node **N**.



[Fig. 7 : Insertion into Linked List **]**

Insertion Algorithm

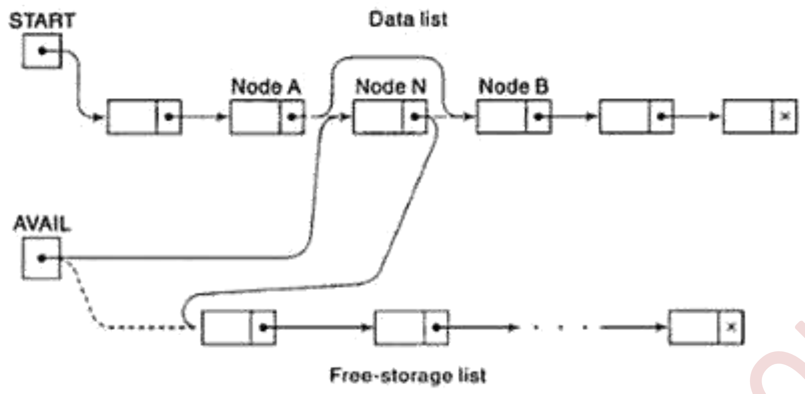
Suppose the value of LOC is given, where either LOC is the location of a node A in a linked list or LOC = NULL. The following is an algorithm which inserts ITEM into LIST so that ITEM follows node A, or when LOC = NULL, ITEM is the first node.

Algorithm

```
INSLOC(INFO, LINK, START, AVAIL, LOC, ITEM)
       This algorithm inserts ITEM so that
       ITEM follows the node with location
       LOC or inserts ITEM as the first
       node when LOC = NULL.
1. [overflow?]
     If AVAIL = NULL then
          Print OVERFLOW
          and Exit.
2.  [Remove first node from AVAIL list.]
     Set NEW := AVAIL
     Set AVAIL := LINK[AVAIL]
3.  [Copy new data into new node]
     Set INFO[NEW] := ITEM
4.  If LOC = NULL then
          [Insert as first node]
          Set LINK[NEW] := START
          Set START := NEW
     Else
          [Insert after node with location LOC]
          Set LINK[NEW] := LINK[LOC]
          Set LINK[LOC] := NEW
     [End of If structure]
5.  Exit.
```

Deletion from a linked list

Let LIST be a linked list with a node N between nodes A and B. as shown in Fig. 8. Suppose node N is to be deleted from the linked list. The schematic diagram of such a deletion shows in Fig. 4. The deletion occurs as soon as the nextpointer field of node A is changed so that it points to node B. The deleted node will return its memory space to the AVAIL list.



[Fig. 8 : Deletion from Linked List & added to AVAIL**]**

Deleting the Node following a given Node

Let LIST be a linked list in memory. Suppose we are given the location LOC of a node N in LIST and also given the location LOCP of the node preceding N or, when N is the first node, we are given LOCP = NULL. The following algorithm deletes N from the list.

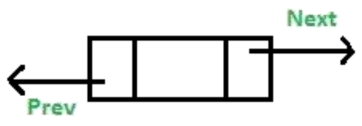
Deletion Algorithm

```
DEL (INFO, LINK, START, AVAIL, LOC, LOCP)
    This algorithm deletes the node N with
    location LOC. LOCP is the location of the
    node which precedes N or, when N is the
    first node, LOCP = NULL.

1. If LOCP = NULL then
           [Deletes first node]
           Set START := LINK[START]
   Else
           [Deletes node N]
           Set LINK[LOCP] := LINK[LOC]
   [End of If structure]

2. [Return deleted node to the AVAIL list]
     Set LINK[LOC] := AVAIL
     Set AVAIL := LOC

3. Exit.
```

- ***Types of Linked List***

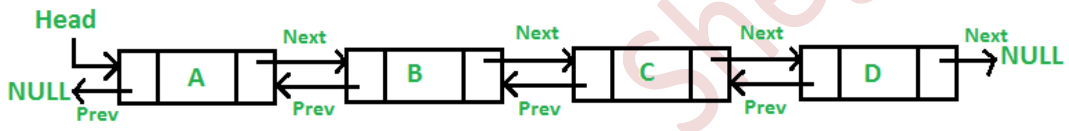
1) Doubly Linked List

A doubly Linked List is a variation of linked list data structure in which each node not only has a link to the next node but also has a link to the previous node in the sequence. Hence, in doubly linked list navigation is possible in both ways, either forward or backward easily. In a doubly linked list, a node consists of three fields: node data called DATA, pointer to the next node in sequence (next pointer) called NEXT, pointer to the previous node (previous pointer) called PREV. A sample node in a doubly linked list is shown in the figure.



[Fig.9 : A node in doubly linked list]

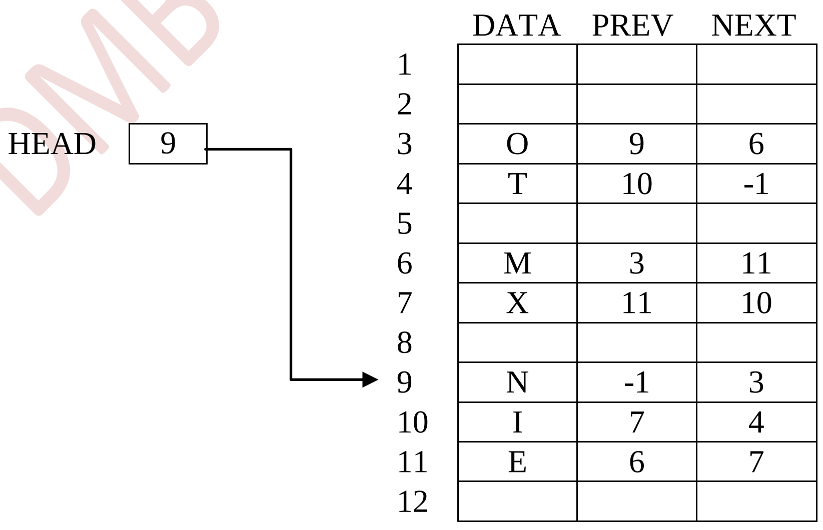
Following figure shows a doubly linked list.



[Fig.10 : A doubly linked list]

In the above example, each node has three fields. Previous pointer (first field) stores the address of previous node and next pointer (third field) stores address of next node. Second field stores information. First node from the list contains Null value in prev. pointer. Similarly last node from the list contains Null value in next field. Apart from these two, the other remaining nodes in between contains address of previous node in prev. pointer and address of next node in next pointer.

Memory Representation of a doubly linked list



	DATA	PREV	NEXT
1			
2			
3	O	9	6
4	T	10	-1
5			
6	M	3	11
7	X	11	10
8			
9	N	-1	3
10	I	7	4
11	E	6	7
12			

[Fig.11 : Memory representation of doubly linked list]

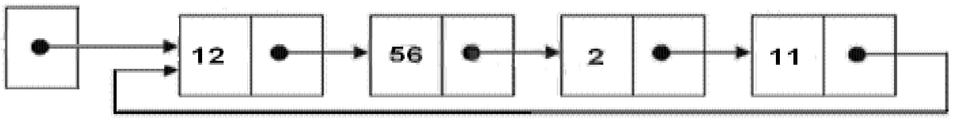
The doubly linked list is maintained in memory by three linear arrays—DATA, PREV and NEXT. The DATA[K] contains the information part of a kth node of list, and PREV[K] contains the previouspointer field of a node of list and NEXT[K] contains the nextpointer field of a node of list. The doubly linked list also requires a variable name such as HEAD, which contains the location of the first node of the list, and a nextpointer and previouspointer fields denoted by NULL or -1, which indicates the end of the list.

In the above figure, the first element of the list that is N stored at address 9. The head pointer points to the starting address 9. Since this is the first element being added to the list therefore the PREV of the list contains null. The next node of the list resides at address 3 therefore the first node contains 3 in its next pointer. We can traverse the list in this way until we find any node containing null or -1 in its next part.

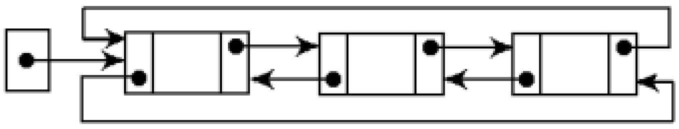
2) Circular Linked List

Circular Linked List is a variation of Linked list. A circular linked list is basically a linear linked list that may be singly or doubly. The only difference is that there is no any NULL value terminating the list. In fact in the list every node points to the next node and last node points to the first node, thus forming a circle. Since it forms a circle with no end to stop hence it is called as circular linked list.

In a *circular singly linked list*, the last node of the list contains a pointer to the first node of the list, and in a *circular doubly linked list*, the first element points to the last element and the last element points to the first element.



[Fig.12 : Circular Singly Linked List]

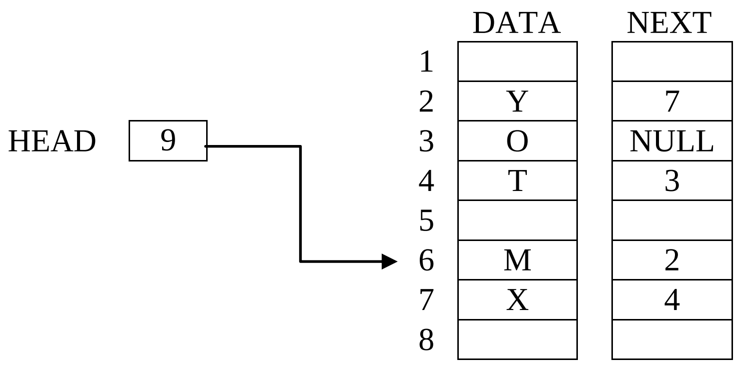

[Fig.13 : Circular Doubly Linked List]

In circular linked list there can be no starting or ending node, it can be traversed from any node. In order to traverse the circular linked list only once we need to traverse entire list until the starting node is not traversed again.

Memory Representation of circular linked list:

The circular linked list is maintained in memory by two linear arrays - DATA and NEXT. The DATA[K] contains the information part of a kth node and NEXT[K] contains the nextpointer field i.e. address of next node in the list. The circular linked list also requires a variable name such as HEAD, which contains the location of the first node of the list, and a nextpointer field denoted by NULL or -1, which indicates the end of the list.

In the following figure shows a glance of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 6 and containing M and data and 2 in the next part.



[Fig.14 : Memory representation of circular linked list]

- **Advantages of linked list (over Array)**

  Linked list provides following advantages over arrays:

  1) **Dynamic size:** The size of linked list is not required to be known in advance and therefore, linked list uses exactly as much memory as it needs and memory is allocated as and when necessary.
  2) **No single allocation of memory needed:** No need of contiguous memory location. Fragmented memory can be put to a better use.
  3) **Ease of insertion/deletion:** In a linked list, insertions and deletions can be handled efficiently without fixing the size of the memory in advance.
  4) **Flexibility:** Insertion or deletion in linked list at any position is in constant time.
  5) There is no upper limit for the number of nodes in the linked list.
  6) Linear data structures such as stacks and queues are easily executed with linked list.

- **Applications of linked list**

  Following are some important applications of linked list:

  1) **Polynomial Manipulation:** Linked list is used to implement polynomial operations such as addition, subtraction, multiplication, division, integration, and differentiation.
  2) **Dynamic Memory Management:** Linked list is used in dynamic memory management to allocate and release memory at runtime.
  3) **Symbol Tables:** Linked list is used in compiler to construct and maintain a dictionary of names and their associated values.
  4) **Multiple-Precision Arithmetic:** Linked list is used in number of applications to obtain a particular accuracy in the results.

Tree

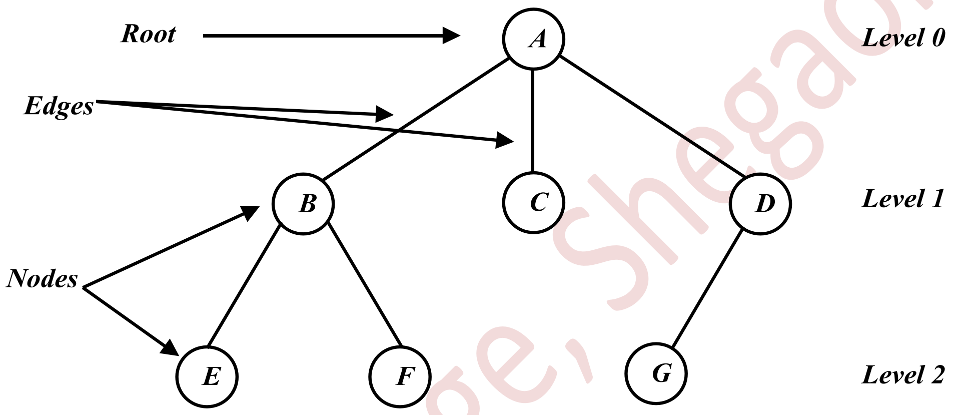
A tree is a nonlinear data structure that models a **hierarchical** organization. A tree is a collection of elements called ***nodes*** and edges. Tree provides us with a means of organizing information so that it can be accessed very quickly and also insertion and deletion of items quickly with compared to arrays and linked lists.

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

i) There is a specially designated node called the *root*;

ii) The remaining nodes are partitioned into *n* disjoint sets $T_1, ..., T_n$ where each of these sets is a tree. $T_1, ..., T_n$ are called the *subtrees* of the root.

Following Fig.-15 shows a tree:



[Fig. 15 : Tree **]**

- ***Basic Terminologies used with trees***

Root

It is the mother node of a tree structure. This node does not have parent. It is the first node in the hierarchical arrangement. There can be only one root node in a tree. In *Fig-1* the node A is the root of the tree.

Node

Node is the main component of the tree. It stores the data. Nodes are connected by means of links with other nodes. In *Fig.1* A, B, C, D, E, F, and G are the nodes.

Parent

The parent of a node is the immediate predecessor of that node. In *Fig.1,* A is the parent of nodes B, C and D, the node B is parent of nodes E and F, and the node D is a parent of G.

Child

The immediate successors of a node are called its child nodes. In *Fig.1,* the nodes B, C and D are the child nodes of A, the nodes E and F are Childs of B, and the node G is a child of D.

Leaf Nodes

A leaf node is a node which does not have any child node. It is located at the end of tree. In *Fig.1*, nodes E, F, C and G are leaf nodes. Degree of leaf is zero.

Siblings

The child nodes of same parent are called siblings. They are also called brother nodes. In *Fig.1* the nodes B, C and D are siblings, also, nodes E and F are siblings.

Link (Branch or Edge)

A link is a pointer to a node in the tree. In other words, link connects two nodes. The line drawn from one node to other is called link. A node can have more than one link. In *Fig.1*, the node A has three links.

Level

The level of a node in the tree is the rank of the hierarchy. The root node is placed in level 0. If a node is at level l then its child is at level $l + 1$ and its parent is at level $l – 1$ except for root. In Fig.1, the node A is at level 0. Nodes B, C and D are at level 1. Nodes E, F and G are at level 2.

Path Length

It is the number of successive edges from source node to destination node. In *Fig.1*, the path length from node A to node F is 2 because there are two edges.

Degree of a node

The maximum number of children that can exist for a node is called as a degree of a node. In *Fig.1*, the degree of node A is 3, the degree of node B is 2, and the degree of node D is 1.

Height

The highest number of nodes that is possible in a way starting from a root to leaf node is called a height of a tree. In *Fig.1*, the height of the tree is 3, either by referring A-B-E or A-B-F or A-D-G.

Binary Tree

A **binary** tree is a specialization of a tree where each node can have **at most two children nodes.** A node's children are called its **left child** and **right child.**

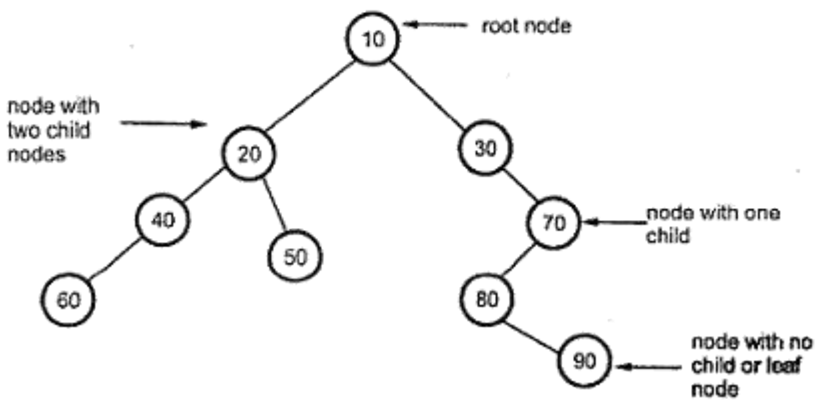
A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called the left subtree and right subtree.

Definition: A binary tree is a set of zero or more nodes T such that:

i) There is a specially designated node called the ***root*** of the tree

ii) The remaining nodes are partitioned into two disjointed sets, T_1 and T_2, each of which is a binary tree. T_1 is called the ***left subtree*** and T_2 is called ***right subtree***.

Following Fig. 16 shows binary tree.



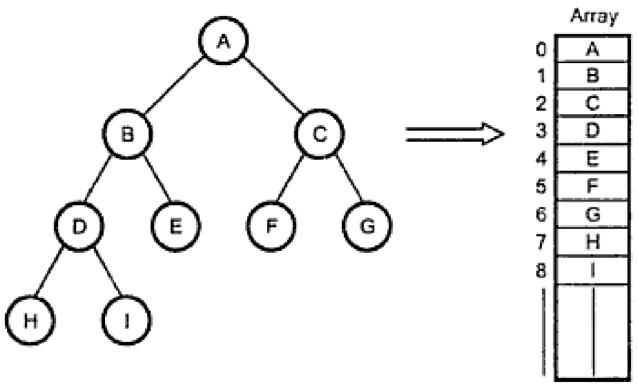
[Fig. 16 : Binary tree **]**

- ***Representation of Binary trees***

There are two ways of representing the binary tree.

1. Sequential representation
2. Linked representation.

1. Sequential representation or array representation of binary tree



[Fig. 17 : Sequential representation of binary trees **]**

A binary tree can be represented using an array capable of holding ***n*** elements (where ***n*** is the number of nodes in a binary tree). The ***root*** of a binary tree is the first element in the array; its left child is the second, right child is the third, and so on. The data value of the ***i***th node of a binary tree at an index ***i*** in an array tree. This will be shown above in Fig.17.

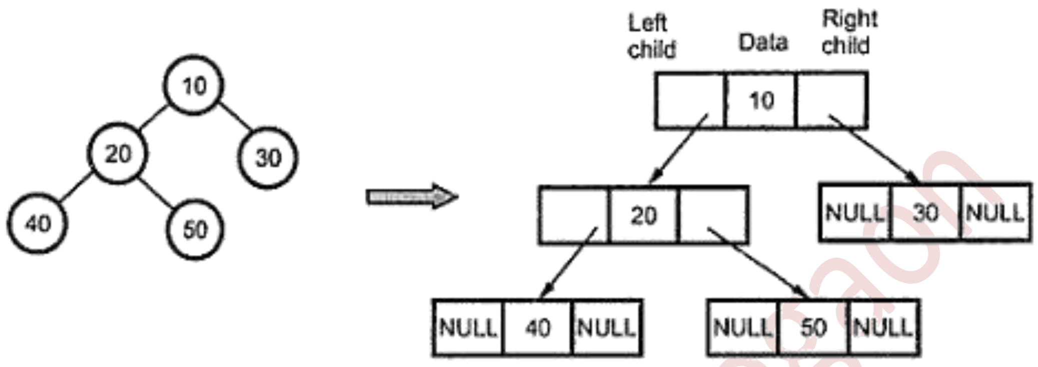
An array representation of a binary tree is not suitable for frequent insertions and deletions. Therefore, instead of using an array representation, a linked representation is used.

2. Linked representation or node representation of binary trees

In a linked representation every node is represented as a structure with three fields: one for holding data, one for linking it with the left subtree, and the third for linking it with right subtree as shown here:

Left child link	Data	Right child link

The *left child link* is a pointer which points to the address of left subtree whereas *right child link* is also a pointer which points to the address of right subtree. And the data field gives the information about the node. The binary tree with linked representation is as shown below in Fig. 18:



[Fig. 18 : Linked representation of binary trees **]**

Traversing Binary Trees

A process in which each node of the binary tree is "visited", or processed, exactly once is called a traversal. A complete traversal of a binary tree yields a linear arrangement of the nodes of the tree.

In a binary tree, there are three different entities involved: a root, its left subtree, and its right subtree. The sequence in which these entities are processed, defines a particular traversal method.

There are three standard ways to traversing a binary tree T with root R. these three algorithms, called:

1. Preorder (*also called Node-Left-Right* ***NLR*** *algorithm*)
2. Inorder (*also called Left-Node-Right* ***LNR*** *algorithm*)
3. Postorder (*also called Left-Right-Node* ***LRN*** *algorithm*)

• Preorder Traversal

The functioning of preorder traversal of a non-empty binary tree is as follows:

1) First, process the root R.
2) Next, traverse the left subtree of R in preorder.
3) Lastly, traverse the right subtree of R in preorder.

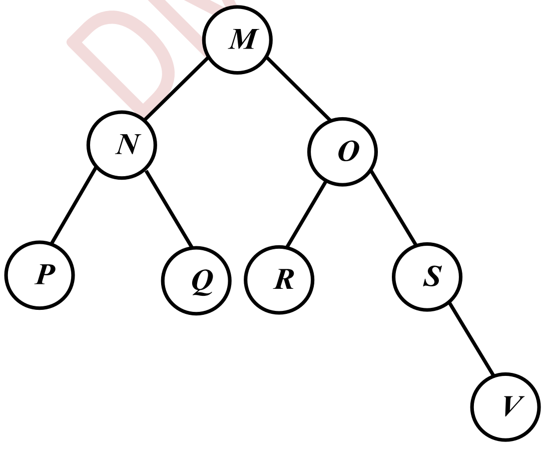
In the preorder traversal firstly the root node is traversed. Then left sub-tree is traversed recursively in preorder. And lastly, the right sub-tree is traversed recursively in preorder.

Algorithm:

```
PREORDER(INFO, LEFT, RIGHT, ROOT)
   A binary tree T is in memory. The algorithm does a preorder
   traversal of T, applying an operation PROCESS to each of
   its nodes. An array STACK is used to temporarily hold the
   addresses of nodes.

1.  [Initially push NULL onto STACK, and initialize PTR.]
      Set TOP := 1, STACK[1] := NULL and PTR := ROOT
2.  Repeat steps 3 to 5 while PTR ≠  NULL:
3.       Apply PROCESS to INFO[PTR].
4.       [Right child?]
         If RIGHT[PTR] ≠ NULL then
                [Push onto STACK]
                Set TOP := TOP+1
                Set STACK[TOP] := RIGHT[PTR]
         [End of IF structure.]
5.    [Left child?]
      If LEFT[PTR] ≠ NULL then
            Set PTR := LEFT[PTR]
      Else
            [Pop from the STACK]
            Set PTR := STACK[TOP] and
            Set TOP := TOP-1.
      [End of IF structure.]
   [End of step 2 loop.]
6.  Exit.
```

Examples:

1] Consider the following binary tree in Fig. 19:

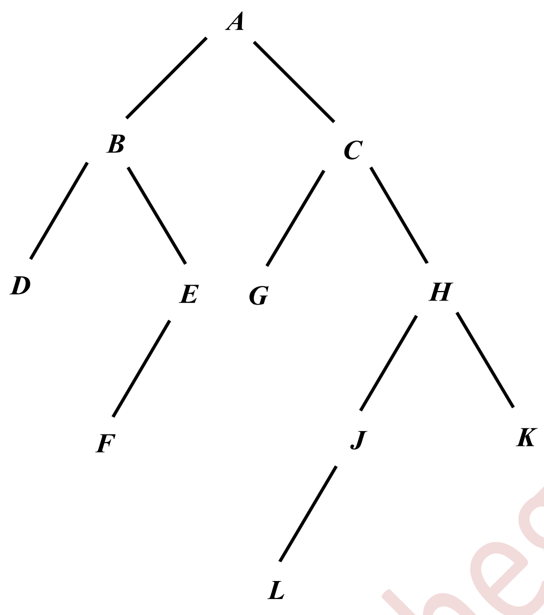


[Fig. 19 : Binary tree **]**

The preorder traversing of the binary tree in Fig.5 is:

Preorder : ***M N P Q O R S V***

2] Consider the following binary tree in Fig. 20:



[Fig. 20 : Binary tree **]**

The preorder traversing of the binary tree in Fig.6 is:

Preorder : ***A B D E F C G H J L K***

• Inorder Traversal

The functioning of inorder traversal of a non-empty binary tree is as follows:

1) First, traverse the left subtree of R in inorder.
2) Next, process the root R.
3) Lastly, traverse the right subtree of R in inorder.

In the inorder traversal firstly the left sub-tree is traversed recursively in inorder. Then the root node is traversed. And lastly, the right sub-tree is traversed recursively in inorder.

Algorithm:

```
INORDER(INFO, LEFT, RIGHT, ROOT)
   A binary tree T is in memory. The algorithm does a
   inorder traversal of T, applying an operation PROCESS
   to each of its nodes. An array STACK is used to
   temporarily hold the addresses of nodes.

1.  [Initially push NULL onto STACK, and initialize PTR.]
     Set TOP := 1, STACK[1] := NULL and PTR := ROOT
```

```
2.  Repeat while PTR ≠  NULL:
        [Pushes left-most path onto STACK.]
      a) Set TOP := TOP+1 and STACK[TOP] := PTR. [Saves node]
      b) Set PTR := LEFT[PTR].     [Updates PTR]
    [End of loop.]
3.  Set PTR := STACK[TOP] & TOP := TOP-1.     [Pops node]
4.  Repeat Steps 5 to 7 while PTR ≠  NULL:   [Backtracking]
5.        Apply PROCESS to INFO[PTR].
6.        [Right child?]
          If RIGHT[PTR] ≠ NULL then
               a) Set PTR := RIGHT[PTR]
               b) Go to Step 2.
          [End of IF structure.]
7.  Set PTR := STACK[TOP] and TOP := TOP-1.    [Pops node]
    [End of step 4 loop.]
8.  Exit.
```

Examples:

1] The inorder traversing of the binary tree in Fig.5 is:

Inorder : ***P N Q M R O S V***

2] The inorder traversing of the binary tree in Fig.6 is:

Inorder : ***D B F E A G C L J H K***

• Postorder Traversal

The functioning of postorder traversal of a non-empty binary tree is as follows:

1) First, traverse the left subtree of R in postorder.
2) Next, traverse the right subtree of R in postorder.
3) Lastly, process the root R.

In the postorder traversal firstly the left sub-tree is traversed recursively in postorder. Then the right sub-tree is traversed recursively in postorder. Lastly, the root node is traversed.

Algorithm:

```
POSTORDER(INFO, LEFT, RIGHT, ROOT)
    A binary tree T is in memory. The algorithm does a
    postorder traversal of T, applying an operation
    PROCESS to each of its nodes. An array STACK is used
    to temporarily hold the addresses of nodes.

1.   [Initially push NULL onto STACK, and initialize PTR.]
          Set TOP := 1, STACK[1] := NULL and PTR := ROOT
2.   [Push left-most path onto STACK]
     Repeat steps 3 to 5 while PTR ≠ NULL:
3.        [Pushes PTR onto STACK]
          Set TOP := TOP+1 and STACK[TOP] := PTR.
4.        If RIGHT[PTR] ≠ NULL then
               [Push onto STACK]
               Set TOP := TOP+1 and
               STACK[TOP] := -RIGHT[PTR].
          [End of IF structure.]
5.        Set PTR := LEFT[PTR] [Updates pointer PTR]
     [End of step 2 loop.]
6.   [Pops node from the STACK]
     Set PTR := STACK[TOP] and TOP := TOP-1.
7.   Repeat while PTR >0
          a) Apply PROCESS to INFO[PTR].
          b) Set PTR := STACK[TOP] and TOP := TOP-1
            [Pops node from the STACK]
     [End of loop.]
8.   If PTR < 0 then
          a) Set PTR := -PTR
          b) Got to Step 2.
          [End of IF structure.]
9.   Exit.
```

Examples:

1] The postorder traversing of the binary tree in Fig.5 is:

Postorder : ***P Q N R V S O M***

2] The postorder traversing of the binary tree in Fig.6 is:

Postorder : ***D F E B G L J K H C A***

• Representing arithmetic expressions

One use of the binary trees is to represent arithmetic expression. A binary tree which represents arithmetic expression is called "*binary expression tree*". The binary expression tree contains number, variables and unary and binary operators. Some of the common operators are (*) multiplication, (/) division, (+) addition, (-) subtraction, (^) exponentiation and (-) negation.

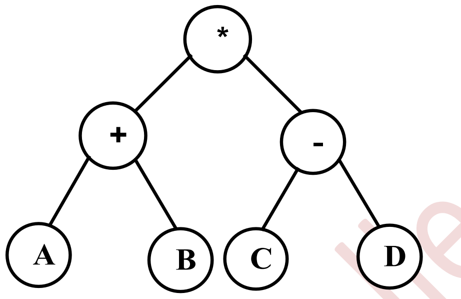
A binary tree stores a binary expression in such a way that each leaf node contains an *operand* (number or variable) of the expression and each interior node contains an *operator* of the expression.

Internal nodes = operators
External nodes = operands

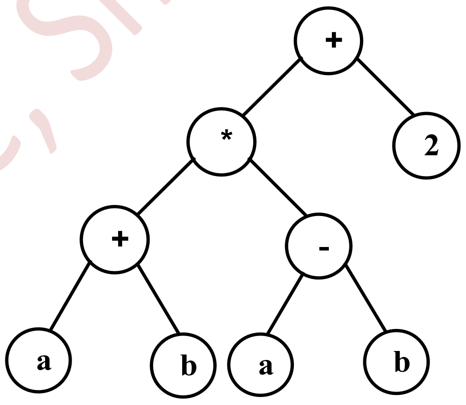
Examples :-

1) Draw the binary tree for the following expressions

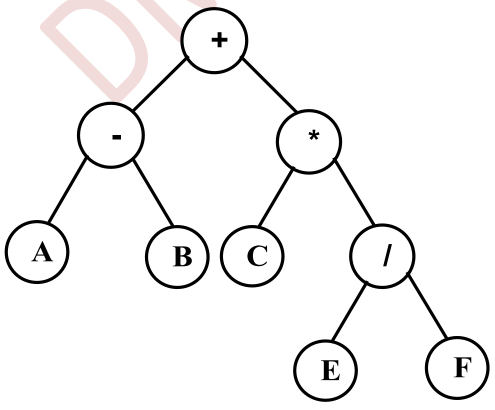
i) (A + B) * (C – D) **ii)** (a + b) * (a – b) + 2
iii) [A – B] + C * [E / F] **iv)** (2 x (a-1) + (3 x b))



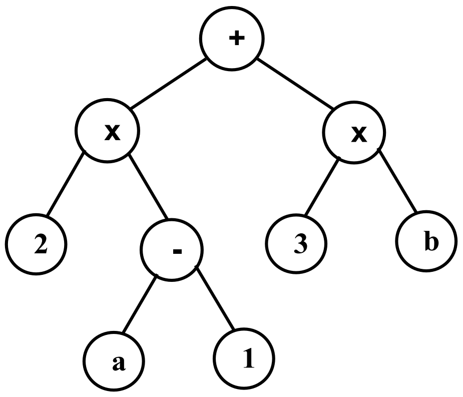
(A + B) * (C – D)



(a + b) * (a – b) + 2



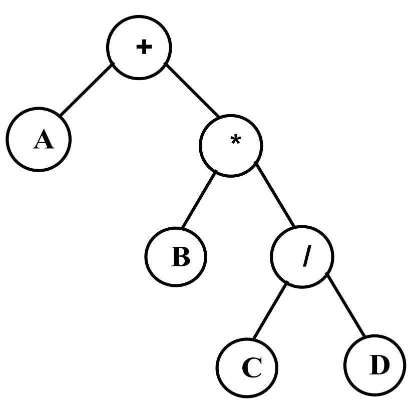
[A – B] + C * [E / F]



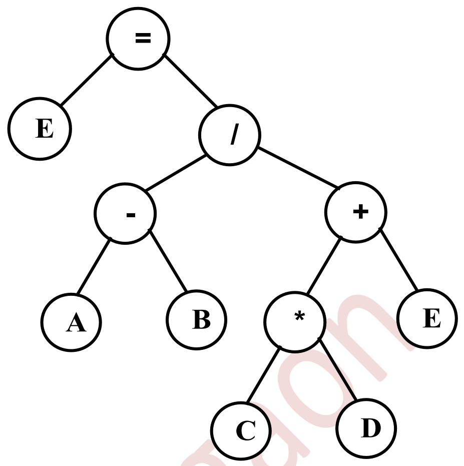
(2 x (a-1) + (3 x b))

2) Draw the binary tree for the expressions

i) A + (B * (C / D)) **ii)** E = (A – B) / ((C * D) + E)



A + (B * (C / D))



E = (A – B) / ((C * D) + E)

Types of Trees in data structure

Following are the types of trees in data structure:

1. General Tree
2. Binary Tree
3. Complete Binary Tree
4. Binary Search Tree

1. General Tree

In the data structure, General tree is a tree in which each node can have either zero or many child nodes. Every node may have infinite numbers of children in General Tree. It cannot be empty. In general tree, there is no limitation on the degree of a node. The topmost node of a general tree is called the root node. There are many subtrees in a general tree. General trees are used to model applications such as file systems. A tree data structure is shown in Figure 15 (page No. 15).

(*Please refer page no. 15 for more details of the tree*)

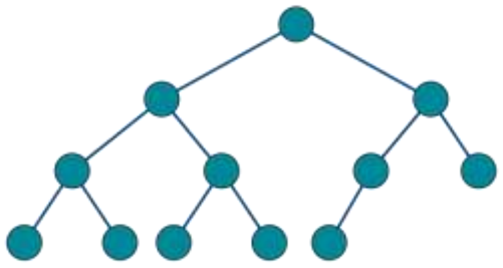
2. Binary Tree

The binary tree is a type of tree data structure where every node has a maximum of two child nodes. As the name suggests, binary means two, therefore, each node can have 0, 1, or 2 nodes. A binary tree data structure is shown in Figure 16 (page No. 16).

(*Please refer page no. 16 and 17 for more details of the binary tree*)

3. Complete Binary Tree

A complete binary tree is another specific type of binary tree where all the tree levels are filled entirely with nodes, except the lowest level of the tree. Also, in the last or the lowest level of this binary tree, every node should possibly reside on the left side. Here is the structure of a complete binary tree:



[Fig. 21 : Complete Binary Tree **]**

4. Binary Search Tree

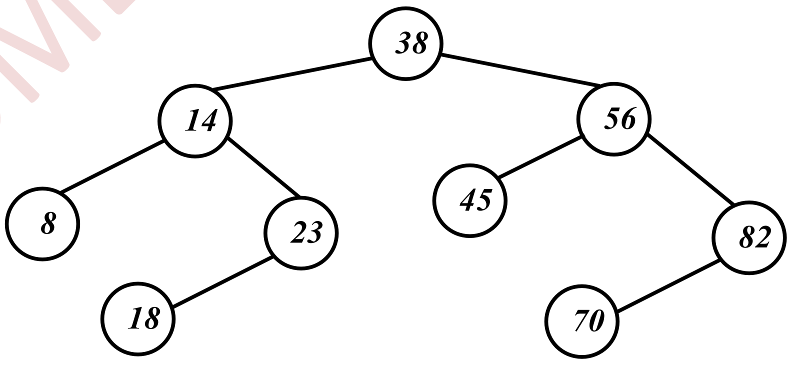
A binary search tree (***or binary sorted tree or BST***) is a binary tree which has the following properties:

- The left subtree of a node contains only nodes with values less than the node's value.
- The right subtree of a node contains only nodes with values greater than the node's value.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes.

The major advantage of binary search trees over other data structures is that the related sorting and search algorithms such as in-order traversal can be very efficient.

Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.

Example :-



[Fig. 22 : Binary Search Tree **]**

■ ■ ■ ■ ■