Unit2: Queue and Trees

Queue: Definition and Concepts

A queue in C is basically a linear data structure to store and manipulate the data elements. It follows the order of First In First Out (FIFO). In queues, the first element entered into the array is the first element to be removed from the array. A queue is open at both ends. One end is provided for the insertion of data which is called as REAR and the other end is used for the deletion of data which is called as FRONT. 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.

Operations on Queue

A queue provides the following operations for manipulation on the data elements:

1. isEmpty(): To check if the queue is empty
2. isFull(): To check whether the queue is full or not
3. dequeue(): Removes the element from the frontal side of the queue
4. enqueue(): It inserts elements to the end(REAR of the queue

Working of Queue Data Structure

Queue follows the First-In-First-Out pattern. The first element is the first to be pulled out from the list of elements. Front and Rear pointers keep the record of the first and last element in the queue. At first, we need to initialize the queue by setting Front = -1 and Rear = -1.

In order to insert the element (enqueue), we need to check whether the queue is already full i.e. check the condition for Overflow. If the queue is not full, we'll have to increment the value of the Rear index by 1 and place the element at the position of the Rear pointer variable.

When we get to insert the first element in the queue, we need to set the value of Front to 0. In order to remove the element (dequeue) from the queue, we need to check whether the queue is already empty i.e. check the condition for Underflow.

If the queue is not empty, we'll have to remove and return the element at the position of the Front pointer, and then increment the Front index value by 1. When we get to remove the last element from the queue, we will have to set the values of the Front and Rear index to -1.

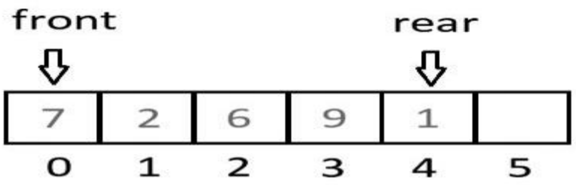
Representation of queues in the Computer memory

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.

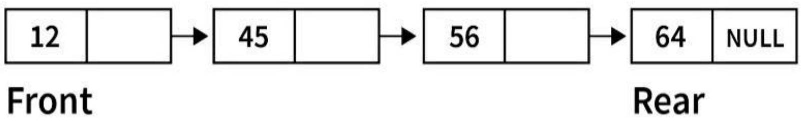


Initially, the value of front and queue is -1 which represents an empty queue. when an element is inserted into queue the value of REAR is increased by 1 and FRONT is unchanged, if queue is not full. Similarly, when an element is deleted from queue the value of FRONT is increased by 1 and REAR is unchanged, 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. Memory wastage: The space of the array can never be reused to store the elements of that queue. 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'. 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.

Operations on Queue:

Some basic operations on Queue:

1. Traversal on Queue
2. Insertion Operation
3. Deletion Operation

Inserting an element into the queue

Inserting an element into a queue is called ―Enqueue".

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.

```
STEP1. [Check for queue overflow]
    If REAR >= N
    Then print "Queue overflow" and exit.
STEP2. [Increment the REAR pointer by one]
    REAR = REAR + 1
STEP3. [Insert element ITEM in new REAR position]
    QUEUE[REAR] = ITEM
STEP4. [Is FRONT pointer properly set?]
    IF FRONT = 0 Then
    FRONT = 1
    End If
```

STEP5.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.

```
STEP1. [Check for queue underflow]
        If FRONT = 0
        Then print "Queue underflow" and exit.
STEP2. [Delete element]
        ITEM = QUEUE[FRONT]
STEP3. [Set FRONT pointer properly]
        IF FRONT = REAR Then
       FRONT = 0
        REAR = 0
        Else
        FRONT = FRONT + 1
        End If
STEP4. Exit.
```

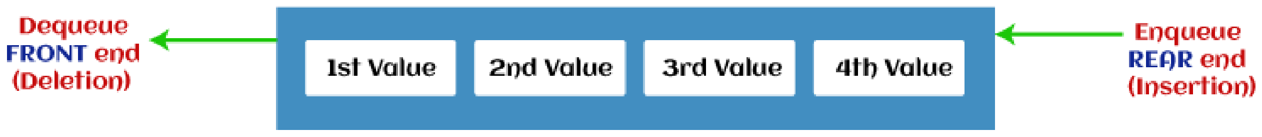
Types of Queue:

There are four different types of queue that are listed as follows –

- Simple Queue or Linear Queue

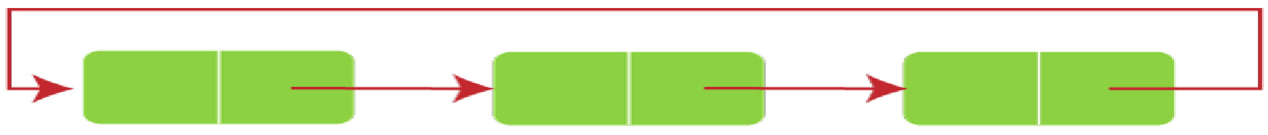
- Circular Queue
- Priority Queue
- Double Ended Queue (or Deque)

Simple Queue or Linear Queue: In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.



The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

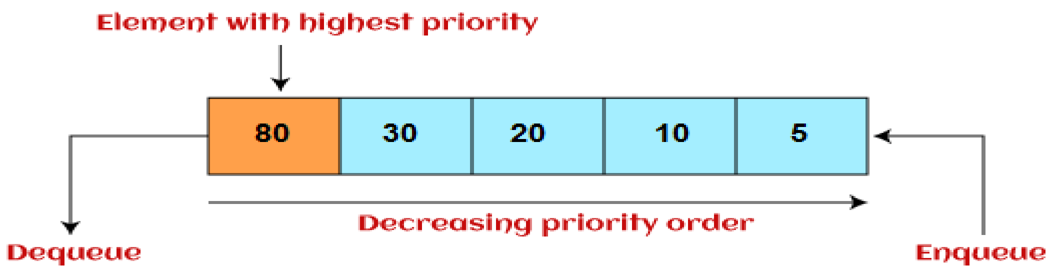
Circular Queue: In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer, as all the ends are connected to another end. The representation of circular queue is shown in the below image -



Circular Queue

The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear. The main advantage of using the circular queue is better memory utilization.

Priority Queue: It is special type of queue in which the elements are arranged based on the priority. It is a special type of queue data structure in which every element has a priority associated with it. Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle. The representation of priority queue is shown in the below image -

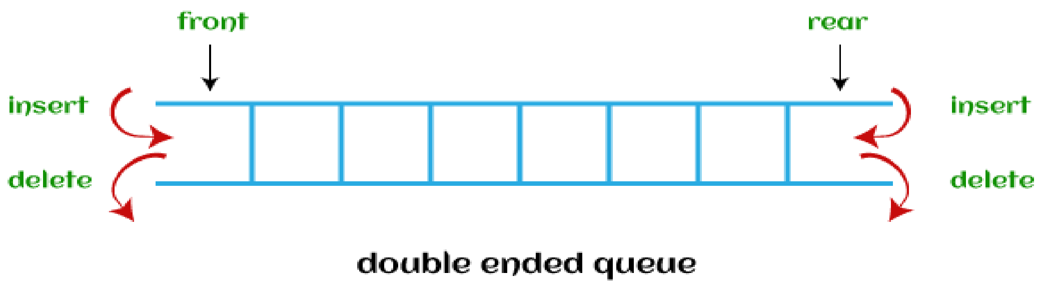


Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms.

There are two types of priority queue that are discussed as follows -

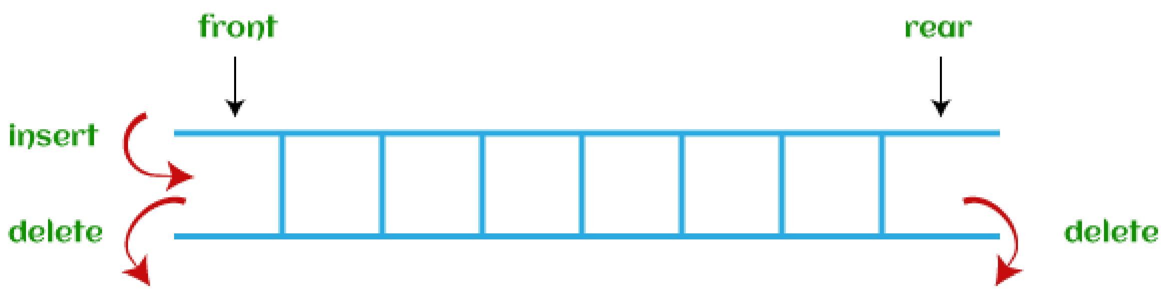
- **Ascending priority queue -** In ascending priority queue, elements can be inserted in arbitrary order, but only smallest can be deleted first. Suppose an array with elements 7, 5, and 3 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 3, 5, 7.
- **Descending priority queue -** In descending priority queue, elements can be inserted in arbitrary order, but only the largest element can be deleted first. Suppose an array with elements 7, 3, and 5 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 7, 5, 3.

Deque (or, Double Ended Queue): In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear. It means that we can insert and delete elements from both front and rear ends of the queue. Deque can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same. Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends. Deque can be considered as stack because stack follows the LIFO (Last In First Out) principle in which insertion and deletion both can be performed only from one end. And in deque, it is possible to perform both insertion and deletion from one end, and Deque does not follow the FIFO principle. The representation of the deque is shown in the below image -



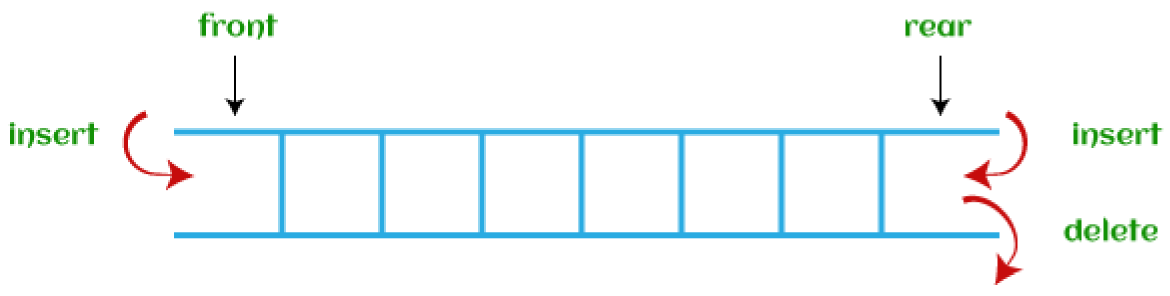
There are two types of deque that are discussed as follows -

- **Input restricted deque -** As the name implies, in input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

- **Output restricted deque -** As the name implies, in output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



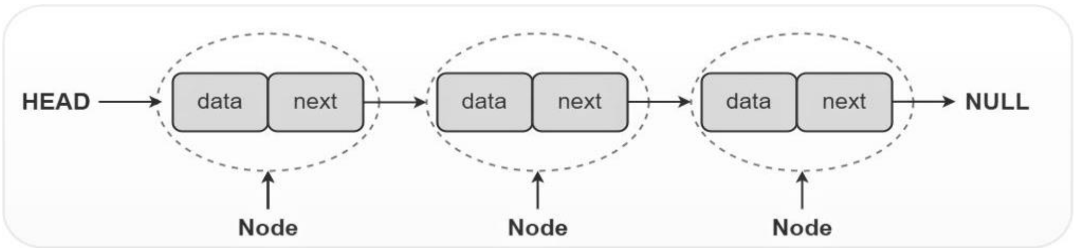
Output restricted double ended queue

Linked List

A linked list is the dynamic linear data structure which is collection of multiple nodes. A linked list consists of a data element known as a node. And each node consists of two fields: one field has data, and in the second field, the node has an address that keeps a reference to the next node. The last node contains null in its second field because it will point to no node. A Linked list can grow and shrink its size, as per the requirement. It does not waste memory space.

Representation of a Linked List

This representation of a linked list depicts that each node consists of two fields. The first field consists of data, and the second field consists of pointers that point to another node. In given fig, the start pointer named as HEAD stores the address of the first node, and at the end, there is a null pointer that states the end of the Linked List.



Creation of Node and Declaration of Linked Lists

```
struct node
{
  int data;
  struct node * next;
};
  struct node * n;
   n=(struct node*)malloc(sizeof(struct node*));
```

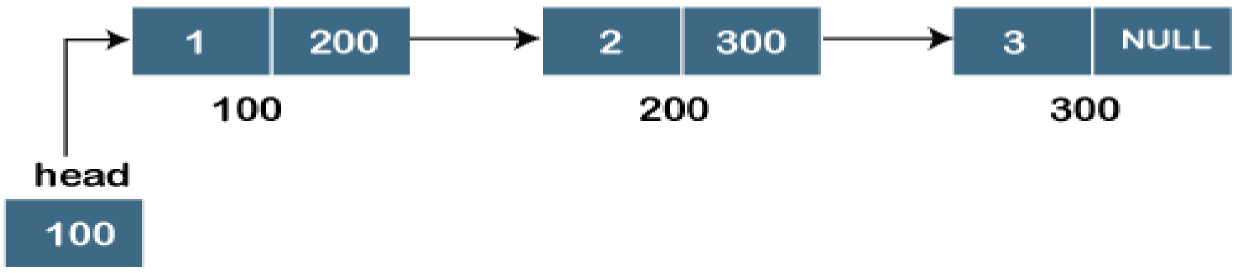
It is a declaration of a node that consists of the first variable as data and the next as a pointer, which will keep the address of the next node. Here you need to use the malloc function to allocate memory for the nodes dynamically.

Types of Linked list

The following are the types of linked list:

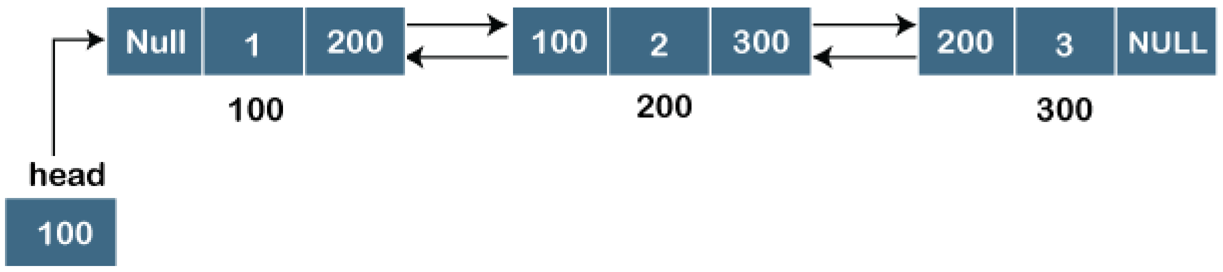
1. Singly Linked list
2. Doubly Linked list
3. Circular Linked list
4. Doubly Circular Linked list

Singly Linked list: It is the commonly used linked list in programs. If we are talking about the linked list, it means it is a singly linked list. The singly linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node. The address part in a node is also known as a pointer. Suppose we have three nodes, and the addresses of these three nodes are 100, 200 and 300 respectively. The representation of three nodes as a linked list is shown in the below figure:



We can observe in the above figure that there are three different nodes having address 100, 200 and 300 respectively. The first node contains the address of the next node, i.e., 200, the second node contains the address of the last node, i.e., 300, and the third node contains the NULL value in its address part as it does not point to any node. The pointer that holds the address of the initial node is known as a *head pointer*. In this list, only forward traversal is possible; we cannot traverse in the backward direction as it has only one link in the list.

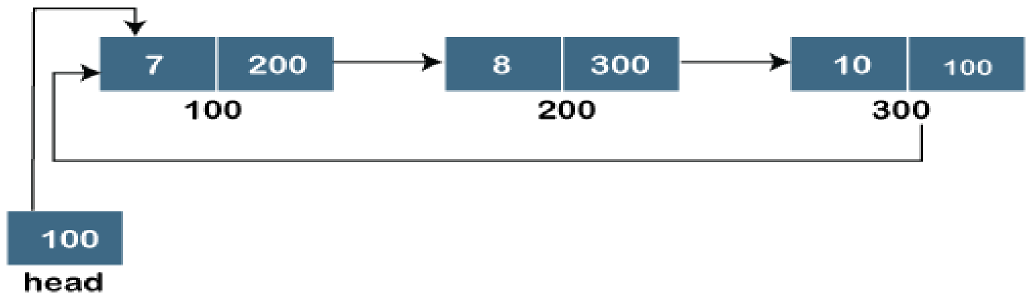
Doubly linked list: As the name suggests, the doubly linked list contains two pointers. We can define the doubly linked list as a linear data structure with three parts: the data part and the other two address part. In other words, a doubly linked list is a list that has three parts in a single node, includes one data part, a pointer to its previous node, and a pointer to the next node. Suppose we have three nodes, and the address of these nodes are 100, 200 and 300, respectively. The representation of these nodes in a doubly-linked list is shown below:



As we can observe in the above figure, the node in a doubly-linked list has two address parts; one part stores the *address of the next* while the other part of the node stores the *previous node's address*. The initial node in the doubly linked list has the NULL value in the address part, which provides the address of the previous node.

Circular linked list: circular linked list is a variation of a singly linked list. The only difference between the singly linked list *and a* circular linked list is that the last node does not point to any node in a singly linked list, so its link part contains a NULL value. On the other hand, the circular linked list is a list in which the last node connects to the first node, so the link part of the last node holds the first node's address. The circular linked list has no starting and ending node. We can traverse in any direction, i.e., either backward or forward. The diagrammatic representation of the circular linked list is shown below:

A circular linked list is a sequence of elements in which each node has a link to the next node, and the last node is having a link to the first node. The representation of the circular linked list will be similar to the singly linked list, as shown below:



Doubly Circular linked list: The doubly circular linked list has the features of both the circular linked list and doubly linked list. The above figure shows the representation of the doubly circular linked list in which the last node is attached to the first node and thus creates a circle. It is a doubly linked list also because each node holds the address of the previous node also. The main difference between the doubly linked list and doubly circular linked list is that the doubly circular linked list does not contain the NULL value in the previous field of the node. As the doubly circular linked contains three parts, i.e., two address parts and one data part so its representation is similar to the doubly linked list

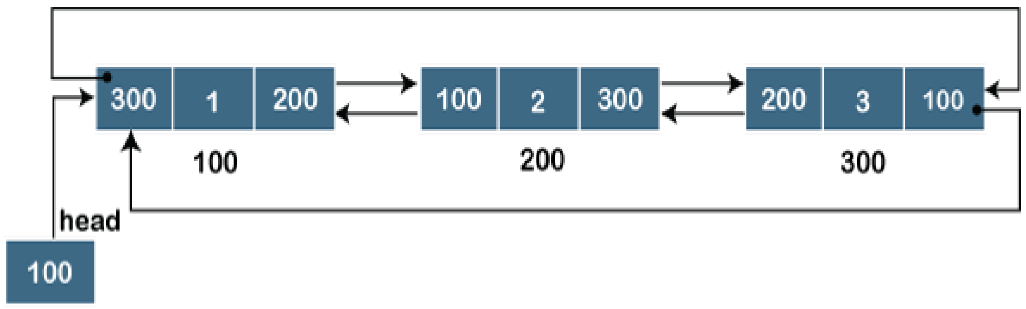


Fig: Doubly Circular Linked List

Operations on Linked List:

There are three basic operations:

1. Traversal on Linked List
2. Insertion Operation
3. Deletion Operation

Traversing Operation: Traversing is the most common operation that is performed in almost every scenario of singly linked list. Traversing means visiting each node of the list once in order to perform some operation on that. This will be done by using the following Algorithm.

ALGORITHM

```
STEP 1: SET PTR = HEAD

STEP 2: IF PTR = NULL

        WRITE "EMPTY LIST"

        GOTO STEP 7
        END OF IF
STEP 4: REPEAT STEP 5 AND 6 UNTIL PTR != NULL
STEP 5: PRINT PTR→ DATA
STEP 6: PTR = PTR → NEXT
        [END OF LOOP]
STEP 7: EXIT
```

Insertion Operation: The Insertion Operation can be doned at three positions in Linked List

1. Insertion At Beginning
2. Insertion At Specified Position
3. Insertion At End Position

Insert Node At Beginning:

There are the following steps which need to be followed in order to inser a new node in the list at beginning. Allocate the space for the new node and store data into the data part of the node. Make the link part of the new node pointing to the existing first node of the list. At the last, we need to make the new node as the first node of the list.

Algorithm

```
Step 1: IF PTR = NULL
        WRITE OVERFLOW
        GOTO STEP 7
        [END OF IF]
Step 2: SET NEW_NODE = PTR
Step 3: SET PTR = PTR → NEXT
Step 4: SET NEW_NODE → DATA = VAL
Step 5: SET NEW_NODE → NEXT = HEAD
Step 6: SET HEAD = NEW_NODE
```

Step 7: EXIT

Insertion at specified positions involves the following procedure: In order to insert an element after the specified number of nodes into the linked list, we need to skip the desired number of elements in the list to move the pointer at the position after which the node will be inserted. Allocate the space for the new node and add the item to the data part of it. Now, we just need to make a few more link adjustments and our node at will be inserted at the specified position. Since, at the end of the loop, the loop pointer temp would be pointing to the node after which the new node will be inserted. Therefore, the next part of the new node ptr must contain the address of the next part of the temp (since, ptr will be in between temp and the next of the temp). Now, we just need to make the next part of the temp, point to the new node ptr.

Insert Node At Specified Position

This Algorithm is used to insert the new node ptr, at the specified position.

ALGORITHM

```
STEP 1: IF PTR = NULL
        WRITE  OVERFLOW  & GOTO STEP12
        END IF
STEP 2: SET NEW_NODE = PTR
STEP 3: NEW_NODE → DATA = VAL
STEP 4: SET TEMP = HEAD
STEP 5: SET I = 0
STEP 6: REPEAT STEP 5 AND 6 UNTIL I<loc< li=""></loc<>
STEP 7: TEMP = TEMP → NEXT
STEP 8: IF TEMP = NULL
        WRITE “DESIRED NODE NOT PRESENT”
        GOTO STEP 12
        END IF
```

```
    END OF LOOP

STEP 9: PTR → NEXT = TEMP → NEXT

STEP 10: TEMP → NEXT = PTR

STEP 11: SET PTR = NEW_NODE

STEP 12: EXIT
```

Deletion Operation

The Deletion of a node from a singly linked list can be performed at different positions. It can be doned at three different positions:

1. Deletion At Beginning
2. Deletion At Specified Position
3. Deletion At End Position

Deletion at Specified Position

In order to delete the node, which is present after the specified node, we need to skip the desired number of nodes to reach the node after which the node will be deleted. We need to keep track of the two nodes. The one which is to be deleted the other one if the node which is present before that node. For this purpose, two pointers are used: ptr and ptr1. Now, our task is almost done, we just need to make a few pointer adjustments. Make the next of ptr1 (points to the specified node) point to the next of ptr (the node which is to be deleted).

ALGORITHM

```
STEP 1: IF HEAD = NULL

    WRITE UNDERFLOW

    GOTO STEP 10

    END IF

STEP 2: SET TEMP = HEAD

STEP 3: SET I = 0

STEP 4: REPEAT STEP 5 TO 8 UNTIL I<loc< li=""></loc<>
STEP 5: TEMP1 = TEMP

STEP 6: TEMP = TEMP → NEXT

STEP 7: IF TEMP = NULL

        WRITE “DESIRED NODE NOT PRESENT”

         GOTO STEP 12

        END IF

STEP 8: I = I+1 END OF LOOP

STEP 9: TEMP1 → NEXT = TEMP → NEXT

STEP 10: FREE TEMP

STEP 11: EXIT
```

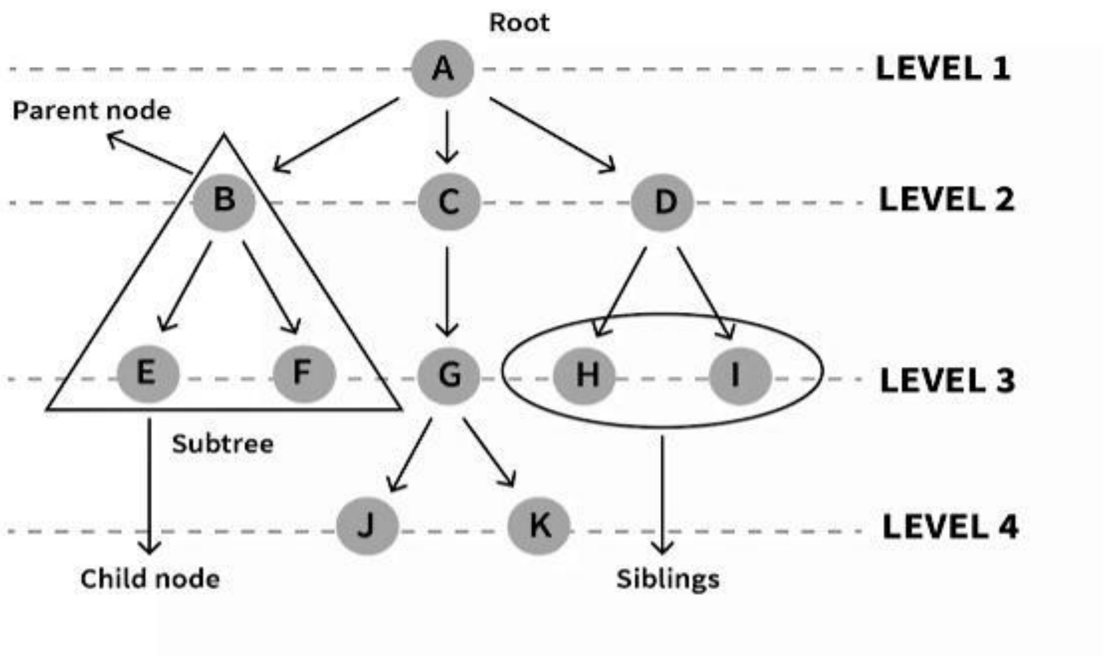
Tree Data Structure

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. A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.

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

1. There is a specially designated node called the root;
2. The remaining nodes are partitioned into n disjoint sets T1, ...,Tn where

Each of these sets is a tree. T1, ...,Tn are called the subtrees of the root.



Basic Terminologies of Tree

Root: The root node is the topmost node present in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.

Child node: If the node is a descendant of any node, then the node is known as a child node.

Parent: If the node contains any sub-node, then that node is said to be the parent of that sub-node.

Sibling: The nodes that have the same parent are known as siblings.

Leaf Node:- The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.

Internal nodes: A node has atleast one child node known as an internal.

Ancestor node:- An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes A, C and G are the ancestors of node J and K.

Descendant: The immediate successor of the given node is known as a descendant of a node. In the above figure, J and K is the descendant of node G.

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.

Level: The level of a node in the tree is the rank of the hierarchy. In above fig 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 1. Nodes B, C and D are at level 2. Nodes E, F and G, I are at level 3. Nodes J and K are at level 4.

Path Length: It is the number of successive edges from source node to destination node. In given 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 given Fig the degree of node A is 3, the degree of node B is 2, and the degree of node C 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.

Representation of Tree Data Structure :

A tree consists of a root, and zero or more subtrees T1, T2, … , Tk such that there is an edge from the root of the tree to the root of each subtree.

In Programming, the structure of a node can be defined as:

```
  struct Node
{
          int data;
         struct Node *first_child;
         struct Node *second_child;
         struct Node *third_child;
          ..........
          struct Node *nth_child;
};
```

This syntax is used to define general tree, because general tree can have more than two childs. Syntax to define binary tree are given below:

```
struct node
 {
 int data;
struct node *leftchild;
struct node *rightchild;
}
```

This syntax is used to define binary tree, The structure of the node for generic trees would be different as compared to the binary tree.

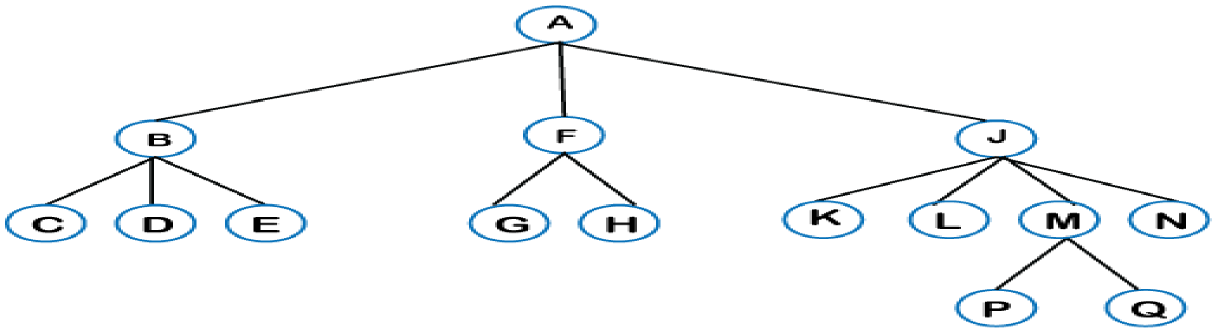
Types of Tree data structure:

The following are the types of a tree data structure:

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

General tree: The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The topmost node in a general tree is known as a root node

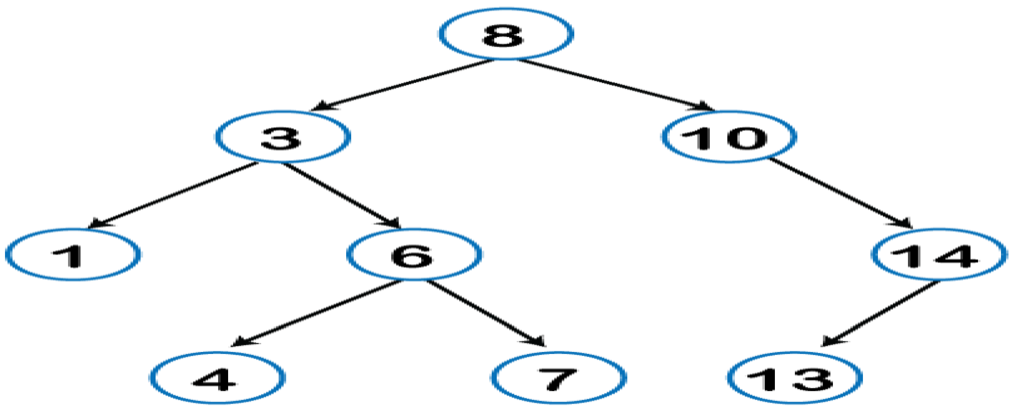
The children of the parent node are known as ***subtrees***.



There can be ***n*** number of subtrees in a general tree. In the general tree, the subtrees are unordered as the nodes in the subtree cannot be ordered. Every non-empty tree has a downward edge, and these edges are connected to the nodes known as ***child nodes***. The root node is labeled with level 0. The nodes that have the same parent are known as ***siblings***.

Binary Tree in Data Structure:

Here, binary name itself suggests two numbers, i.e., 0 and 1. In a binary tree, each node in a tree can have utmost two child nodes. Here, utmost means whether the node has 0 nodes, 1 node or 2 nodes, but do not have more than 2 nodes.



A "binary tree" is a tree data structure where every node has two child nodes (at the most) that form the tree branches. These child nodes are called left and right child nodes. Each node of a binary tree consists of three items:

- data item
- address of left child
- address of right child

Types of binary trees: binary trees are divided into different types based on their structure. Some basic types of binary trees are:

1. Full Binary Tree
2. Completed Binary Tree
3. Perfect Binary Tree

Full Binary Tree: A binary tree node can have either two children or no child. A full binary tree is a binary tree with either zero or two child nodes for each node.

Completed Binary Tree:

Complete binary tree is another specific binary tree where each node on all levels except the last level has two children. And at the lowest level, all leaves should reside possibly on the left side. For a complete binary tree, elements are stored in a level by level manner and filled from the leftmost side of the last level. Some properties of binary tree: A complete binary tree is just like a full binary tree, but with two major differences

- Every level must be completely filled
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.

Perfect Binary Tree: binary tree is said to be perfect if every internal node must have two children and every leaf is present on the same level.

Binary Search tree:

Binary search tree is a non-linear data structure in which one node is connected to ***n*** number of nodes. It is a node-based data structure. A node can be represented in a binary search tree with three fields, i.e., data part, left-child, and right-child. A node can be connected to the utmost two child nodes in a binary search tree, so the node contains two pointers (Left child and Right child pointer) Every node in the left subtree must contain a value less than the value of the root node, and the value of each node in the right subtree must be bigger than the value of the root node.

The major advantage of binary search trees over other data structures is that the related sorting algorithms 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.

Properties of Binary Search Trees:

- 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 values.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes

Operations on Tree Data Structure:

These are three operations on tree

1. Traversal on tree (Pre-order, Inorder, Post-order)
2. Insertion in tree
3. Deletion in tree

Tree Traversal: The term 'tree traversal' means traversing or visiting each node of a tree. There is a single way to traverse the linear data structure such as linked list, queue, and stack. According to this structure, every tree is a combination of 1) A node carrying data and 2) Two subtrees

Remember that our goal is to visit each node, so we need to visit all the nodes in the subtree, visit the root node and visit all the nodes in the right subtree as well. Depending on the order in which we do this, there can be three types of traversal.

- Preorder traversal
- Inorder traversal
- Postorder traversal

Preorder traversal: It means that, first root node is visited after that the left subtree is traversed recursively, and finally, right subtree is recursively traversed. As the root node is traversed before (or pre) the left and right subtree, it is called preorder traversal. So, in a preorder traversal, each node is visited before both of its subtrees.

The applications of preorder traversal include -

- It is used to create a copy of the tree.
- It can also be used to get the prefix expression of an expression tree.

Algorithm: TRAVERSE (Until all nodes of the tree are not visited)

```
Step 1 – Visit the root node.
        Display(root->data)
Step 2 – Traverse the left subtree recursively.
        Preorder(root->left)
Step 3 – Traverse the right subtree recursively.
```

```
        Preorder(root->right)
```

Postorder traversal: It means that the first left subtree of the root node is traversed, after that recursively traverses the right subtree, and finally, the root node is traversed. As the root node is traversed after (or post) the left and right subtree, it is called postorder traversal. So, in a postorder traversal, each node is visited after both of its subtrees.

The applications of postorder traversal include –

- It is used to delete the tree.
- It can also be used to get the postfix expression of an expression tree.

Algorithm TRAVERSE (Until all nodes of the tree are not visited)

```
Step 1 – Traverse the left subtree recursively.
        Postorder(root->left)
Step 2 – Traverse the right subtree recursively.
        Postorder(root->right)
Step 3 – Visit the root node.
        Display(root->data)
```

Inorder traversal: It means that first left subtree is visited after that root node is traversed, and finally, the right subtree is traversed. As the root node is traversed between the left and right subtree, it is named inorder traversal. So, in the inorder traversal, each node is visited in between of its subtrees.

The applications of Inorder traversal includes –

- It is used to get the BST nodes in increasing order.
- It can also be used to get the prefix expression of an expression tree.

Algorithm: TRAVERSE (Until all nodes of the tree are not visited.)

```
Step 1 - Traverse the left subtree recursively.
        Inorder(root->left)
Step 2 - Visit the root node.
```

```
    Display(root->data)
Step 3 - Traverse the right subtree recursively.
    Inorder(root->right)
```

Insertion in Binary Search Tree:

Insert function is used to add a new element in a binary search tree at appropriate location. Insert function is to be designed in such a way that, it must node violate the property of binary search tree at each value.

1. Allocate the memory for tree.
2. Set the data part to the value and set the left and right pointer of tree, point to NULL.
3. If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.
4. Else, check if the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left of the root.
5. If this is false, then perform this operation recursively with the right sub-tree of the root.

Algorithm: Insert (TREE, ITEM)

```
Step 1: IF TREE = NULL
          Allocate memory for TREE
          SET TREE -> DATA = ITEM
         SET TREE -> LEFT = TREE -> RIGHT = NULL
         ELSE
          IF ITEM < TREE -> DATA
           Insert(TREE -> LEFT, ITEM)
         ELSE
          Insert(TREE -> RIGHT, ITEM)
         [END OF IF]
         [END OF IF]
Step 2: END
```

Deletion in Binary Search Tree:

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

There are three situations of deleting a node from binary search tree.

- The node to be deleted is a leaf node
- The node to be deleted has only one child.
- The node to be deleted has two children.

Algorithm: Delete (TREE, ITEM)

```
Step 1: IF TREE = NULL
             Write "item not found in the tree"
            ELSE IF ITEM < TREE -> DATA
              Delete(TREE->LEFT, ITEM)
              ELSE IF ITEM > TREE -> DATA
               Delete(TREE -> RIGHT, ITEM)
              ELSE IF TREE -> LEFT AND TREE -> RIGHT
              SET TEMP = find Largest Node(TREE -> LEFT)
              SET TREE -> DATA = TEMP -> DATA
               Delete(TREE -> LEFT, TEMP -> DATA)
              ELSE
               SET TEMP = TREE
               IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
               SET TREE = NULL
              ELSE IF TREE -> LEFT != NULL
              SET TREE = TREE -> LEFT
              ELSE
               SET TREE = TREE -> RIGHT
              [END OF IF]
              FREE TEMP
```

```
        [END OF IF]
Step 2: END
```

References:

- *https://www.javatpoint.com/data-structure-queue*
- *https://www.javatpoint.com/tree*
- *https://www.programiz.com/dsa/trees*
- *https://www.geeksforgeeks.org/data-structures/linked-list/*

Sample Questions:

1. Explain selection sort technique with suitable example.
2. What do you mean by traversing a binary tree? Write preorder, inorder, postorder traversing of the following binary tree.
3. What is searching? Explain the linear search technique with suitable example.
4. Explain bubble sort method with suitable example.
5. What is tree traversing? Explain its type with suitable example.
6. Explain the following sorting techniques with example: (i) Selection sort (ii) Merge sort
7. Explain the algorithm to find the elements using sequential search method.
8. What is tree? Explain various types of trees with example.
9. What is binary tree? Explain linear representation of binary tree with example.
10. What is traversal of binary tree? Explain various types of traversals with example.
11. Explain inorder, preorder and postorder tree traversal with example.