UNIT 4

Syllabus: Trees – Definition, terminology, Binary trees-definition, Properties of Binary Trees, Binary Tree ADT, representation of Binary Trees - array and linked representations, Binary Tree traversals, Inorder, Postorder, Preorder, Binary Search Tree ADT – BST traversal, BFS and DFS.

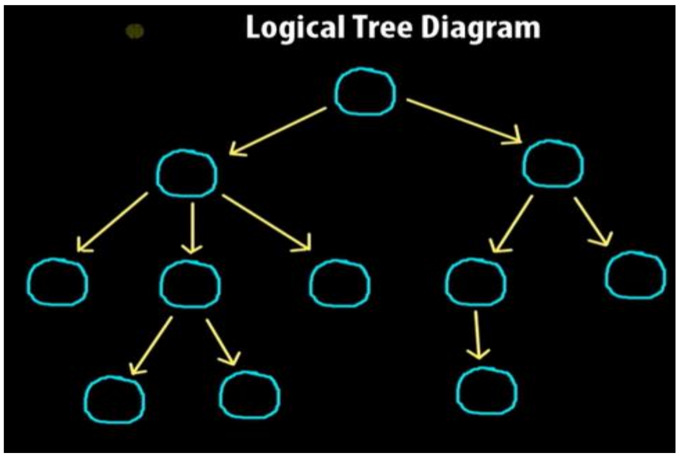
Priority Queues–Definition and applications, Max Priority Queue ADT-implementation-Max Heap-Definition, Insertion into a Max Heap, Deletion from a Max Heap
==

Tree Data Structure: It is a way of storing or organizing data in the memory, in such a way that access, management and modification becomes efficient.

Linear Data Structure: Data is arranged in a sequential manner where a logical start and logical end. Every element has a next element and previous element.

Examples: Arrays, Stack, Queue and Linked Lists.

Non-Linear Data Structure: The data structure that simulates a hierarchical tree structure with a root value and sub-trees of children with parent node, represented as a set of linked nodes.



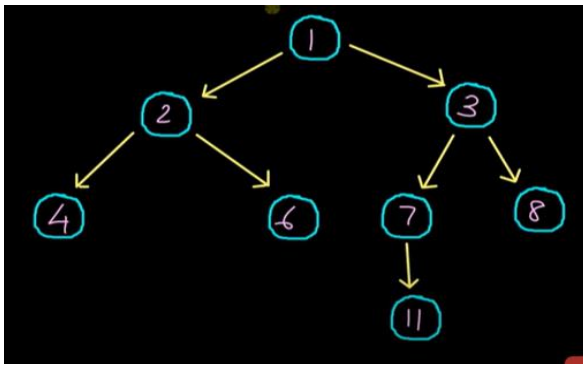
Terminology

1. Root: Root is a special node in a tree. The entire tree is referenced through it. It does not have a parent.
2. Parent node: Parent node is an immediate predecessor of a node.
3. Child node: All immediate successors of a node are its children.
4. Siblings: Nodes with the same parent are called Siblings.
5. Leaf: Last node in the tree. There is no node after this node.
6. Edge: Edge is a connection between one node after this node.
7. Path: Path is a number of successive edges from source node to destination node.
8. Degree of Node: Degree of a node is equal to number of children a node has.

Properties of a Tree

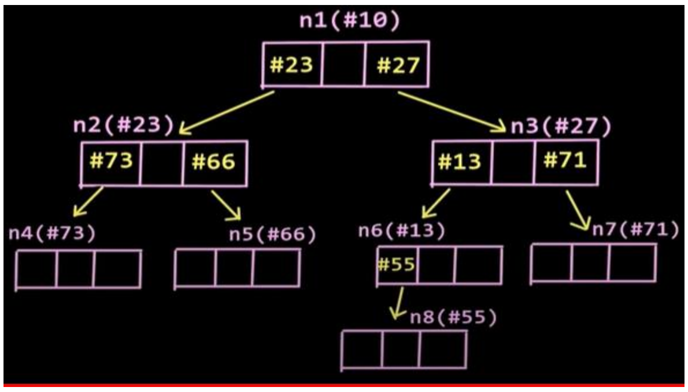
1. Tree can be termed a recursive data structure.
2. A valid tree for N Nodes has N-1 edges or links.
3. Depth of Node: Depth of a node represents the number of edges from the tree's root node to the node.
4. Height of Node: Height of a node is the number of edges on the longest path between that node and a leaf.
5. Height of tree is the height of its root node.

Binary Tree: Every node in the tree has at most two children.



Binary Tree – Implementation View

Node (*left, data, *right)



Node class

Node
{
*Node *left;*
Int data;
*Node *right;*
}

Applications of Tree Data Structure

1. Store hierarchical data like folder structure, organization structure data.
2. Binary Search Tree is a tree that allows fast search, insert, delete on a sorted data.
3. Heap is a tree data structure that is implemented using arrays and used to implement priority queues.
4. B-Tree and B+ Trees are used to implement indexing in databases.
5. Used to store router-tables in routers.
6. Used by compiler to build syntax trees.
7. Used to complement expression parser and expression solvers.

Binary Tree: A binary tree is a tree data structure in which each node has at most two children which are referred to as the left child and the right child.

Representation of Binary Tree Using Arrays

A binary tree is a hierarchical data structure composed of nodes, where each node has at most two children, referred to as the left child and the right child. One way to represent a binary tree in a computer's memory is by using an array. This representation is known as an array representation of a binary tree.

Terminology and Properties.

Node Structure

Each node in a binary tree contains data, a reference (link) to the left child, and a reference to the right child.

In the array representation, the index of an element corresponds to its position in the tree.

Array Representation - Level Order Traversal:

- In a binary tree represented by an array, nodes are filled level by level, starting from the root.
- The root is at index 0, and for any node at index i:
- Its left child is at index 2*i + 1.
- Its right child is at index 2*i + 2.

- A binary tree is called strict binary tree when each node has 2 or 0 children.
- A binary tree is called complete binary tree if all levels except the last are completely filled and all nodes are as left as possible.
- A binary tree is called perfect binary tree of all levels are completely filled with 2 children each.
- Maximum number of nodes at level 'x' = 2^x
- For a binary tree, maximum number of nodes with height 'h' = $2^0 + 2^1 + \ldots\ldots + 2^h$
  $= 2^{(h+1)} - 1$

Binary Tree implementation using Arrays

```
#include <iostream>
#include <cmath>
using namespace  std;

class BinaryTree
{
private:
   int *tree;
   int size;

public:
   BinaryTree(int maxSize)
   {
      size = maxSize;
      tree = new int[size];
      for (int i = 0; i < size; ++i)
        {
         tree[i] = -1;
      }
   }

   void insert(int value)
   {
      insertRecursive(value, 0);
   }
```

```
    void display()
          {
          cout << "Binary Tree - Array Representation ";
        for (int i = 0; i < size; ++i)
          {
            if (tree[i] != -1)
             {
                cout << tree[i] << " ";
            }
          else
            {
                cout << "- ";
            }
        }
        cout << endl;
    }

private:
    void insertRecursive(int value, int index)
          {
        if (index < size)
          {
            if (tree[index] == -1)
             {
                tree[index] = value;
             }
            else
             {
                if (value <= tree[index])
                                        {
                    insertRecursive(value, 2 * index + 1);
                }
              else
                {
                    insertRecursive(value, 2 * index + 2);
                }
            }
        }
    }
};

int main()
{
    BinaryTree binaryTree(15);

    int elements[] = {8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15};
    int numElements = sizeof(elements) / sizeof(elements[0]);

    for (int i = 0; i < numElements; ++i)
          {
```

```
        binaryTree.insert(elements[i]);
    }

    binaryTree.display();
    return 0;
}
```

Representation of Binary Tree Using Linked Lists

A binary tree is a hierarchical data structure where each node has at most two children, referred to as the left child and the right child. One common way to represent a binary tree is through linked lists. In this representation, each node in the binary tree is represented by a linked list node, and pointers are used to connect nodes to represent the left and right children.

Basics of Binary Trees
Node Structure:

Each node in a binary tree contains data, a reference (link) to the left child, and a reference to the right child.

Linked List Representation:

Node Definition
A binary tree node is represented as a struct or class with data and two pointers (left and right) pointing to the left and right children.

Binary tree representation using linked lists

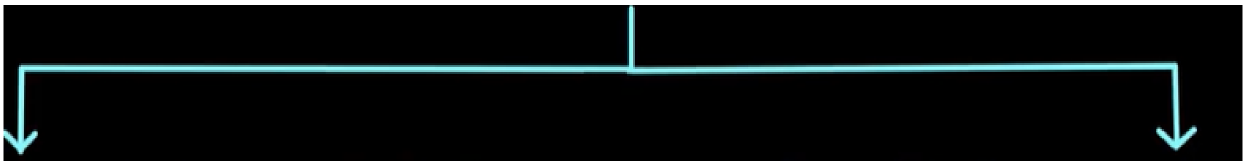
```
#include <iostream>
using namespace std;

struct TreeNode
{
    int data;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int value)
          {
                data= value;
                left = NULL;
                right = NULL;
          }
};

void insert(TreeNode*& root, int value)
{
    if (root == NULL)
          {
        root = new TreeNode(value);
    }
```

```
        else
        {
      if (value <= root->data)
              {
        insert(root->left, value);
      }
              else
              {
        insert(root->right, value);
      }
    }
}

void inOrderTraversal(TreeNode* root)
{
    if (root != NULL)
        {
        inOrderTraversal(root->left);
        cout << root->data << " ";
        inOrderTraversal(root->right);
    }
}

int main()
{
    TreeNode* root = NULL;

    int elements[] = {8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15};
    int numElements = sizeof(elements) / sizeof(elements[0]);

    for (int i = 0; i < numElements; ++i)
        {
        insert(root, elements[i]);
    }

    cout << "In-Order Traversal: ";
    inOrderTraversal(root);
    cout << endl;

    return 0;
}
```

Binary Tree -Traversal Techniques

Tree traversal also known as tree search and walking the tree refers to the process of visiting (checking or updating) each node in a tree data structure, exactly once. The traversals are classified by the order in which the nodes are visited.

Tree Traversal



Depth-first-Search / traversal (DFS)

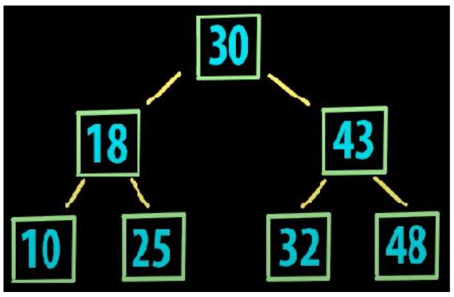
1. Pre-order (NLR)
2. In-order (LNR)
3. Post-order (LRN)

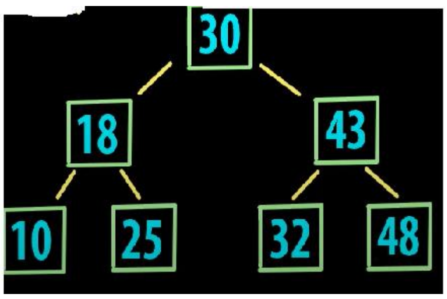
Breadth first search / traversal (BFS)

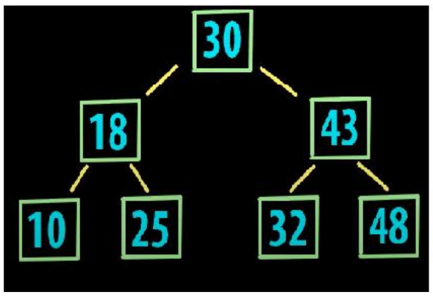
Trees can also be traversed in level-order, where visiting every node on a level before going to a lower level. This Search is referred to as breadth-first-search as the search tree is broadened as much possible on each depth before going to the next level.

Pre-order (NLR)	In-order (LNR)	Post-order (LRN)
Access the data part of the current node	Traverse the left subtree by recursively calling the in-order function.	Traverse the left subtree by recursively calling the post-order function.
Traverse the left subtree by recursively calling the pre-order function.	Access the data part of the current node.	Traverse the right subtree by recursively calling the post-order function.
Traverse the right subtree by recursively calling the pre-order function.	Traverse the right subtree by recursively calling the in-order function.	Access the data part of the current node.
	In BST in-order traversal retrieves the keys in ascending sorted order.	
Pseudocode	Pseudocode	Pseudocode
void printPreorder (TreeNode r) { 1. if (r==NULL) return; 2. print (r-> value); 3. printPreorder (r -> left) 4. printPreorder (r -> right) }*	*void printInorder (TreeNode* r) { 1. if (r==NULL) return; 2. printInorder (r -> left) 3. print (r-> value); 4. printInorder (r -> right) }*	*void printPostorder (TreeNode* r) { 1. if (r==NULL) return; 2. printPostorder (r -> left) 3. printPostorder (r -> right) 4. print (r-> value); }*

Example

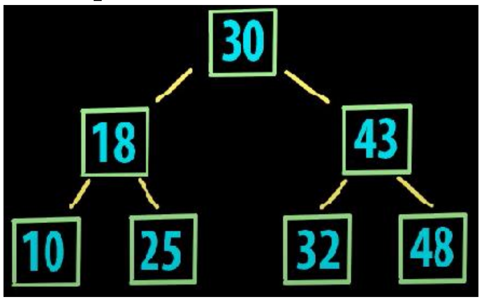






30,18,10, 25, 43,32, 48 10, 18, 25, 30,32, 43,48 10, 25, 18, 32, 48, 43, 30

Level – order (BFS): Tree can be traversed level-order where visiting every node in the same level before going to lower level.
Example

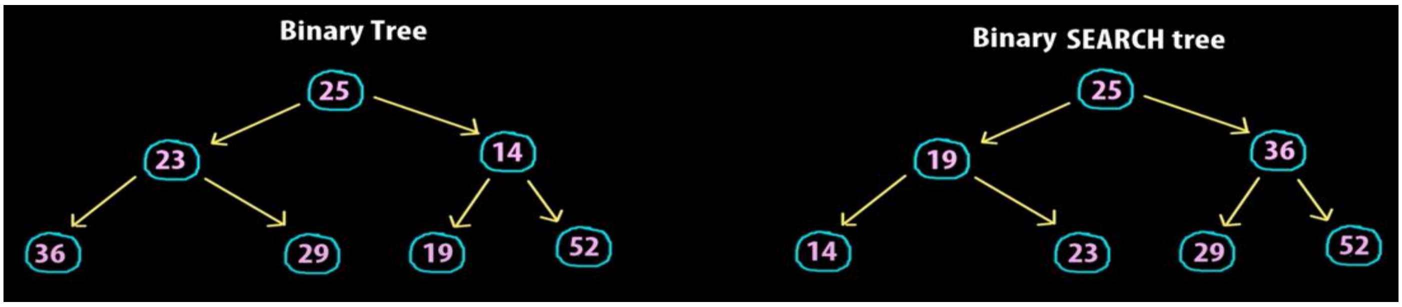


30, 18, 43, 10, 25, 32, 48

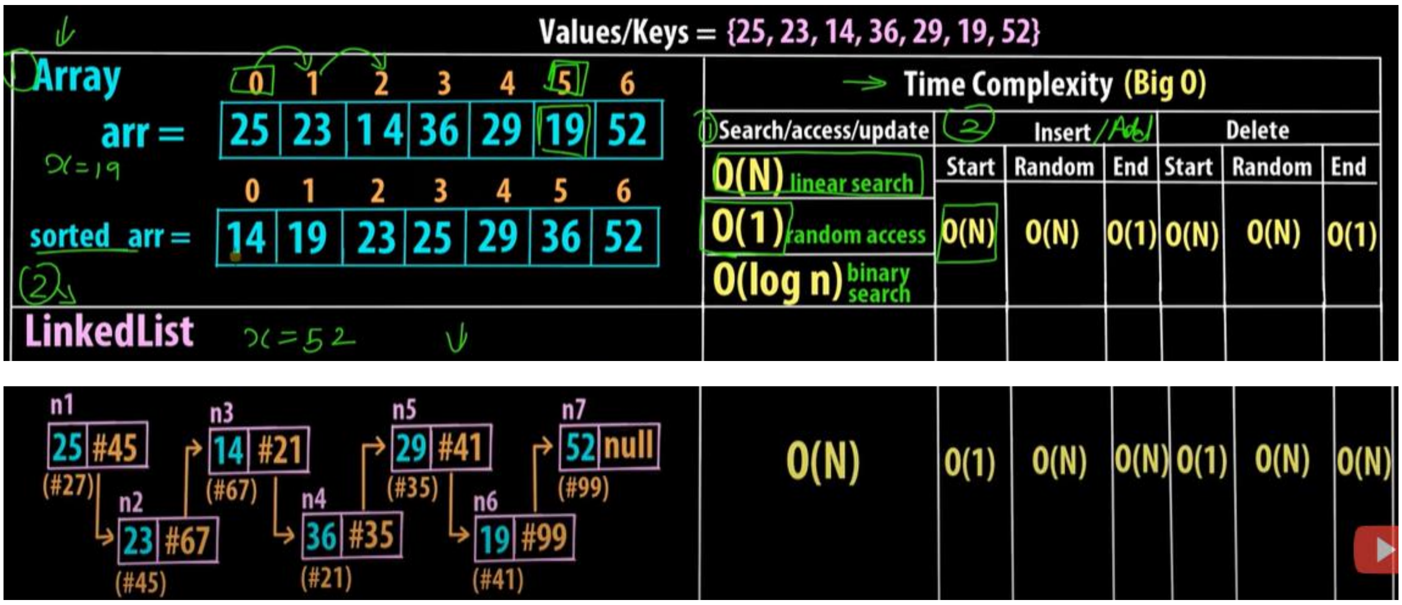
Binary Search Tree (BST)

BST is also a binary tree data structure in which the values in the left subtrees of every node are smaller and the values in the right subtree of every node are larger.

Values | Keys = {25, 23, 14, 36, 29, 19, 52}



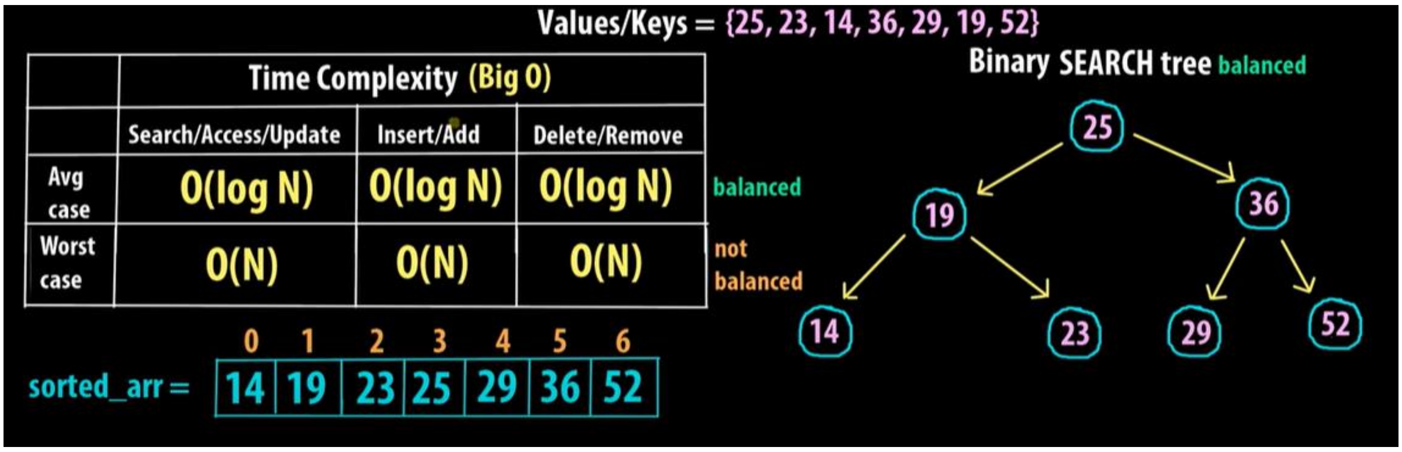
Time Complexity with Arrays and LinkedList


Values/Keys = {25, 23, 14, 36, 29, 19, 52}

Time Complexity (Big O)

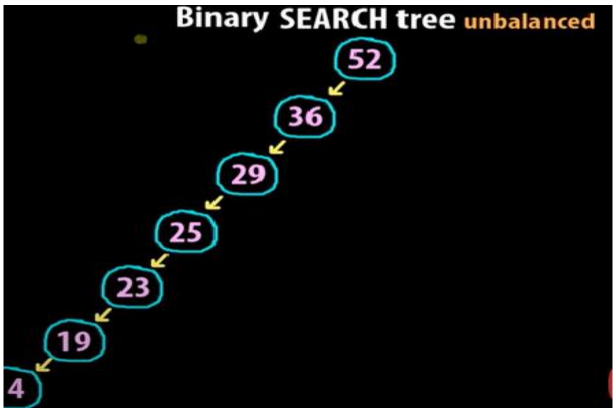
	Search/access/update	Insert /Add Start	Insert /Add Random	Insert /Add End	Delete Start	Delete Random	Delete End
Array	O(N) linear search O(1) random access O(log n) binary search	O(N)	O(N)	O(1)	O(N)	O(N)	O(1)
LinkedList	O(N)	O(1)	O(N)	O(N)	O(1)	O(N)	O(N)

Time Complexity with Binary Search Tree (for best case and worst case)


Values/Keys = {25, 23, 14, 36, 29, 19, 52}

	Time Complexity (Big O)		
	Search/Access/Update	Insert/Add	Delete/Remove
Avg case	O(log N)	O(log N)	O(log N)
Worst case	O(N)	O(N)	O(N)

Time Complexity with Binary Search Tree (for worst case)



Binary search Tree	Linked List
O (log N)	*O (N)*

Suppose there are 1000 elements in the list for one element takes 1sec

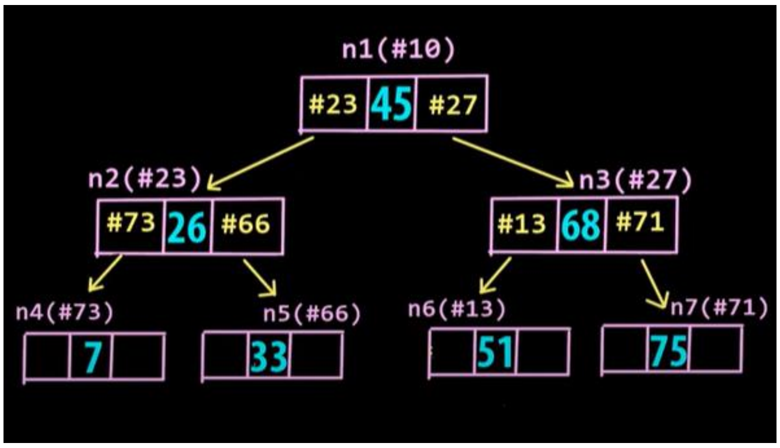
O (log 1000)	*O (1000)*
9.96 sec	*1000 sec*

Binary Search Tree Implementation

Check Lab 8

Binary Search Tree as dynamic nodes in memory

*Node (*left, data, *right);*



Node Class

```
class TreeNode
{
Data Members:
        1. int value;
        2. TreeNode * left;
        3. TreeNode *right;
Member Functions:
        TreeNode ();          //default constructor
        TreeNode (int v)    //parameterized constructor
}
```

BST Class

```
Class BST
{
Data Members:
        1. TreeNode * root
Member Functions:
        1. bool isTreeEmpty ()
        2. void insertNode
(TreeNode *new_node) // insertion
                //Depth First Traversal approach
        3a. void printPreorder (TreeNode * r) print and traverser
        3b. void printInorder (TreeNode* r) //print and traverse
        3c. void printPostorder (TreeNode* r) //print and traverse
        3d. void print2D (TreeNode* r, int space) //print and traverse
        3e. void printLeverOrder (TreeNode* r) // print and traverse
                //breadth first traversal approach
        4. TreeNode* search (TreeNode* r, int v) //search
        5a. TreeNode* minValueNode (TreeNode* node) // delete
        5b. TreeNode* deleteNode (TreeNode* r, int v) // delete
}
```

Pseudocode for Inserting a node into BST

```
void insertNode(new_node)
{
    1. If root == NULL then root = new_node
    2. else
       2.1 set temp = root
       2.2 while temp! = NULL
           2.2.1 if new_node. value == temp.value
               2.2.1.1 then return //duplicate values are allowed

           2.2.2 else if (new_node. value <temp.value) && (temp. left == NULL)
               2.2.2.1 then temp. left = new_node // value inserted on left
               2.2.2.2 break // get out of the function

           2.2.3 else if (new_node.value < temp.value)
               2.2.3.1 then temp = temp. left

           2.2.4 else if (new_node.value > temp. value) && (temp. right == NULL)
               2.2.4.1 temp. right - new_node // value inserted on right
```

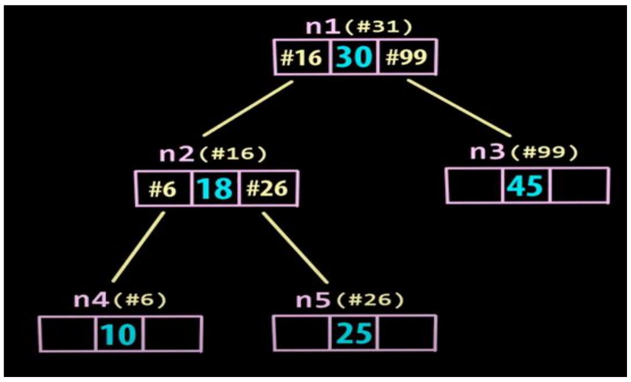
```
                    2.2.4.2 break        // get out of the function

                2.2.5 else
                    2.2.5.1 temp = temp. right
                End while
        End if
} End function

}
```

Height of a Tree

The height of a binary tree is a number of edges between the tree's root node and farthest leaf node.

Pseudocode for Finding height of Binary Tree | BST

```
int height (TreeNode* t)
{
        1. if (r == NULL)
                1.1 return -1
        2. else
                2.1 lheifht = height (r -> left)
                2.2 rheight = height (r -> right)
                2.3 if lheight > rheight
                        2.3.1 return (lheight + 1)
                2.4 else
                        2.4.1 return (rheight + 1)
}
```



Algorithm for Deletion operation on Binary Tree | Binary Search Tree

1. Deletion of a leaf node or node with only 1 child

1. Traverse to the leaf node or node with single child n.

2. Check if n has left child (n -> left).
   If it does not link the right child (n->right) with parent node of n.

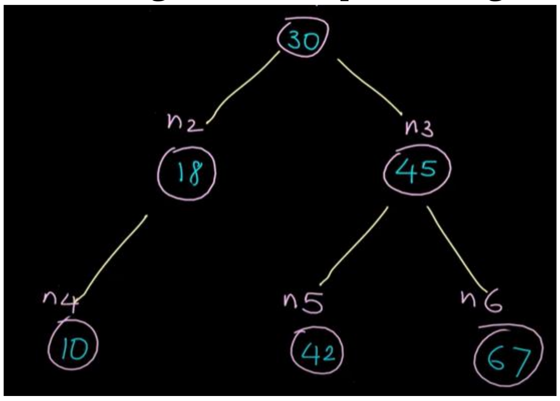
3. If n has left child, then check if n has right child.
   If it does not, link the left child (n->left) with parent node of n.

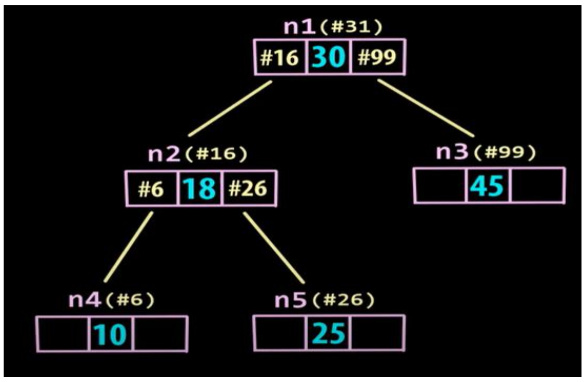
2. Deletion of a leaf node or node with only 1 child

1. Traverse to the node with 2 children to be deleted (n).
2a. Find the smallest node (nMin) in the right sub tree of n.

OR

2b. Find the largest node (nMax) in the left subtree of n.
3. Replace this node (nMin) or nMax) with the node to be deleted (Replace n with nMin).
4. Now delete nMin or nMax using the delete process again.





42 67

Pseudocode for Minimum Value Node

```
TreeNode* minValNode (TreeNode* node)
{
        SET current = node
        while (current -> left != NULL)
                current = current -> left
        return current
}
```

Heap Data Structures

Heap is a specialized tree-based data structure that satisfies the heap property. The heap property is a condition that must be maintained for every node in the heap. There are two common types of heaps: the max heap and the min heap.

Applications of heaps: Task scheduling, graph algorithms and dynamic memory allocation.

Max Heap

In a max heap, for every node 'i', the value of 'i' is less than or equal to the value of its parent. In other words, the value of each node is greater than or equal to the values of its children except root node since the root node does have parent.

A [parent(i)] >= A [i];

The maximum element in the heap is always at the root.

- A max heap can be represented in an array where the root is at index 0. For any node at index i, its left child is at index 2i + 1 and its right child is at index 2i + 2.

- The process of building a max heap from an array is called heapify. You start from the last non-leaf node and move upwards, applying heapify operation at each step. This ensures that the heap property is satisfied.

- The time complexity of building a max heap from an array of size n is O(n).

- To insert an element into a max heap, adding a new element at the end of the heap and then perform a heapify operation to restore the heap property.

- *Algorithm*

```
function heapifyUp(heap, index):
   while index > 0 && heap[parent(index)] < heap[index]:
      swap(heap[parent(index)], heap[index])
      index = parent(index)
```

Suppose the Heap is a Max-Heap as

```
       10
      /    \
     5       3
    / \
   2   4
```

The new element to be inserted is 15.

Step 1: Insert the new element at the end.

```
       10
      /    \
    5       3
   / \     /
  2   4   15
```

Step 2: Heapify the new element following bottom-up approach.
since15 is greater than its parent 3, swap them.

```
       10
      /    \
     5      15
    / \     /
   2   4   3
```

15 is again more than its parent 10, swap them.

```
       15
      /    \
     5      10
    / \     /
   2   4   3
```

Therefore, the final heap after insertion is:

```
       15
      /    \
 5      10
/ \    /
2   4  3
```

- The time complexity of inserting an element into a max heap is O(log n), where n is the number of elements in the heap.

- To extract largest element from a heap, swap it with the last element, remove the last element and then perform a heapify operation to maintain the heap property.

- The time complexity of extracting the maximum element from a max heap is O(log n), where n is the number of elements in the heap.

- A priority queue can be efficiently implemented using a max heap. Enqueueing (inserting) an element and dequeuing (extracting the maximum) both have logarithmic time complexity.

- To sort an array using a max heap, build a max heap from the array and then successively extract the maximum element until the heap is empty.

- To delete an element from a heap
    1. Locate the element to be deleted.
    2. Swap it with the last element in the heap.
    3. Remove the last element.
    4. Perform heapify operation starting from the index of the swapped element.

Suppose the Heap is a Max-Heap as:

```
      10
     /    \
    5       3
   / \
  2   4
```

The element to be deleted is root, i.e. 10.

The last element is 4.

Step 1: Replace the last element with root, and delete it.

```
      4
     /    \
   5        3
  /
 2
```

Step 2: Heapify root. Final Heap:

```
      5
     /    \
    4       3
   /
  2
```

Min Heap

A Minimum Priority Heap, a Min-Heap, is a type of binary heap data structure that maintains the heap property. The heap property for a Min-Heap is defined as follows: for every node i other than the root, the value of i is greater than or equal to the value of its parent.

Structure

A Min-Heap is typically implemented as a complete binary tree, where the values are stored in such a way that the key of each node is less than or equal to the keys of its children.
The tree is usually represented as an array, where the root is at index 0 and for any element at index i, its left child is at index 2*i + 1 and the right child is at index 2*i + 2.

Insertion Operation

When a new element is inserted into the Min-Heap, it is placed at the next available position in the array.
The heap property may be violated after insertion, so a "happify-up" operation is performed to restore the heap property. This involves swapping the new element with its parent until the heap property is satisfied.

Deletion Operation

The minimum element in the Min-Heap is always at the root (index 0).
To remove the minimum element (extract-min), the last element in the array is moved to the root, and a "happify-down" operation is performed to restore the heap property. This involves swapping the root with its smallest child until the heap property is satisfied.

Time Complexity

The time complexity for insertion and deletion in a Min-Heap is O(log n), where n is the number of elements in the heap.
This is because the height of a binary heap is logarithmic with respect to the number of elements.

Applications

Min-Heaps are commonly used in priority queues, where the smallest element needs to be efficiently accessed and removed.
Dijkstra's algorithm for finding the shortest path in a graph often uses Min-Heaps to efficiently extract the minimum distance vertex.

Heap Sort

Heap Sort is a sorting algorithm that uses the concept of a binary heap. It builds a max-heap (or min-heap for descending order) and repeatedly extracts the maximum (or minimum) element.

==00oo00==