

(MCAL11) Advanced Data Structures Lab**Title – Trees****Binary Search Tree (BST):**

A Binary Search Tree (BST) is a type of binary tree data structure in which each node has at most two children, and the following properties hold:

Properties of a BST:

1. Left Subtree Property: All nodes in the left subtree of a node have values less than the value of the node itself.
2. Right Subtree Property: All nodes in the right subtree of a node have values greater than the value of the node itself.
3. No Duplicate Values: Typically, BSTs do not allow duplicate values (though variations exist).

Basic Operations:

Here are the common operations on a BST and their time complexity (average case):

1. Search:
 - Traverse the tree based on comparisons.
 - Time Complexity: $O(\log n)$ (if the tree is balanced).
2. Insertion:
 - Start at the root and find the correct location by comparing values.
 - Insert the new node as a leaf.
 - Time Complexity: $O(\log n)$.
3. Deletion:
 - Deletion has three cases:
 - Node to delete is a leaf (no children): Remove it directly.
 - Node to delete has one child: Replace the node with its child.
 - Node to delete has two children: Replace the node with its in-order successor (smallest value in the right subtree) or in-order predecessor (largest value in the left subtree).
 - Time Complexity: $O(\log n)$.
4. Traversal:
 - Common traversals:
 - In-order: Left, Root, Right (yields sorted order for BST).
 - Pre-order: Root, Left, Right.
 - Post-order: Left, Right, Root.
 - Level-order: Breadth-first traversal.
 - Time Complexity: $O(n)$.

Algorithm for Binary Search Tree:**1. Initialization**

1. Define a Node class with properties:
 - data (to store the value of the node).
 - left (pointer to the left child).
 - right (pointer to the right child).
2. Define a BinarySearchTree class with:
 - root (reference to the root node).

- Static variable count to keep track of the total nodes.

2. Insertion (insertNode)

1. Create a new node with the given value.
2. If the root is null, assign the new node as the root and increment count.
3. Traverse the tree starting from the root:
 - If the value is smaller than the current node, move to the left.
 - If the value is greater, move to the right.
 - If a duplicate is found, exit.
4. Attach the new node to the appropriate position (left or right of a leaf node) and increment count.

3. Traversal

1. **In-order Traversal:**
 - Recursively visit the left subtree.
 - Print the value of the current node.
 - Recursively visit the right subtree.
2. **Pre-order Traversal:**
 - Print the value of the current node.
 - Recursively visit the left subtree.
 - Recursively visit the right subtree.
3. **Post-order Traversal:**
 - Recursively visit the left subtree.
 - Recursively visit the right subtree.
 - Print the value of the current node.

4. Find Smallest Node (smallest)

1. Traverse the tree starting from the root.
2. Move to the left child until the leftmost node is reached.
3. Print the value of the leftmost node.

5. Find Largest Node (largest)

1. Traverse the tree starting from the root.
2. Move to the right child until the rightmost node is reached.
3. Print the value of the rightmost node.

6. Count Nodes (nodeCounts)

1. Return the value of the static variable count.

7. Search Node (search)

1. Traverse the tree starting from the root:
 - If the value is found, print a success message.
 - If the value is smaller than the current node, move to the left.
 - If the value is greater, move to the right.
2. If the traversal ends without finding the value, print a failure message.

8. Remove Node (removeNode)

1. Recursively locate the node to be removed:
 - If the node has one child, replace it with the child.

- If the node has two children, find the in-order predecessor (largest value in the left subtree), copy its value, and remove the predecessor.
- 2. Update the root with the result of the recursive function.
- 3. If the node is found and removed, decrement count.

Heap:

A **Heap** is a specialized tree-based data structure that satisfies the **heap property**. It is commonly implemented as a **binary heap**, which can be of two types:

Types of Heaps:

1. **Max-Heap:**
 - The value of each parent node is greater than or equal to the values of its children.
 - The root node contains the largest value.
2. **Min-Heap:**
 - The value of each parent node is less than or equal to the values of its children.
 - The root node contains the smallest value.

Properties of a Heap:

1. **Complete Binary Tree:**
 - All levels except possibly the last are fully filled.
 - The last level is filled from left to right.
2. **Heap Property:**
 - For Max-Heap: Parent node \geq Child nodes.
 - For Min-Heap: Parent node \leq Child nodes.

Heap Class Structure:

- **Node:** Represents a node in the doubly linked list. Each node has:
 - data: The value stored.
 - prev: Reference to the previous node.
 - next: Reference to the next node.
- **MaxHeap:** Represents the MaxHeap. Contains:
 - head: Reference to the first node.
 - tail: Reference to the last node.

Algorithm for MaxHeap:

Insertion into the MaxHeap

1. **Input:** An integer data.
2. **Create** a new Node with the given data.
3. **Check:**
 - If the heap is empty (head == null):
 - Set head and tail to the new node.
 - Else:
 - Add the new node at the end of the doubly linked list (tail.next = newNode).
 - Update the prev reference of the new node (newNode.prev = tail).
 - Update tail to point to the new node.

4. Call Reheap Up method:

- Compare the new node with its parent.
- Swap values if the new node's value is greater than the parent's value.
- Repeat until the new node is in the correct position or it becomes the root.

3. Deletion of the Root Node**1. Check:**

- If the heap is empty (head == null), print a message and exit.

2. Move the data of the last node (tail) to the root node (head).**3. Remove the Last Node:**

- If there is only one node in the heap, set head = null.
- Otherwise:
 - Update tail to tail.prev.
 - Set tail.next = null.

4. Call Reheap Down method:

- Compare the root node with its children.
- Swap values with the largest child if the root node's value is smaller.
- Repeat until the root node is in the correct position or it has no children.

4. Reheap Up

1. **Input:** A node to adjust upwards in the heap.
2. **Repeat Until:**
 - node is the root or node.data is less than or equal to its parent.
3. **Swap** the values of node and its parent.
4. **Move** to the parent node and repeat.

5. Reheap Down

1. **Input:** A node to adjust downwards in the heap.
2. **Repeat Until:**
 - node has no children or node.data is greater than or equal to the largest child's value.
3. **Find** the largest child.
4. **Swap** the values of node and the largest child.
5. **Move** to the largest child and repeat.

6. Display the Heap

1. **Initialize:** A temporary pointer temp pointing to head.
2. **Traverse** the linked list:
 - Print the data of each node.
 - Move temp to the next node.
3. **Stop** when temp is null.

Algorithm for MinHeap:**Insert Operation**

- Create a method insert(int data) to insert a new node:

1. Create a new node with the given data.
2. If the heap is empty:
 - Set head and tail to the new node.
3. Otherwise:
 - Attach the new node at the end of the list (tail.next = newNode).
 - Update the prev pointer of the new node to point to the current tail.
 - Update tail to point to the new node.
4. Call reheapUp(newNode) to maintain the MinHeap property.

Reheap Up

- Create a method reheapUp(Node node):
 1. Get the parent of the node using the getParent method.
 2. While the node is smaller than its parent:
 - Swap the data of the node and its parent.
 - Update node to its parent and repeat.

Delete Operation

- Create a method delete() to remove the root node:
 1. If the heap is empty, print a message and return.
 2. Replace the data of the root (head) with the data of the last node (tail).
 3. Remove the last node:
 - Update tail to its previous node (tail.prev).
 - If tail is not null, set tail.next = null. Otherwise, set head = null.
 4. Call reheapDown(head) to restore the MinHeap property.

Reheap Down

- Create a method reheapDown(Node node):
 1. While the node has children:
 - Get the leftChild and rightChild of the node.
 - Identify the smaller child (minChild).
 - If the node is smaller than or equal to minChild, break.
 - Otherwise, swap the data of the node and minChild.
 - Update node to minChild and repeat.

Utility Methods

- Create a method getParent(Node node):
 - Return the prev pointer of the node, which represents its parent in the list.
- Create a method printHeap() to print the data of all nodes in the heap:
 - Traverse the doubly linked list starting from head and print the data of each node.

Exercise:

- 1. Write a Java program to construct binary search tree and implement inorder, preorder and preordertraversers, find the largest node, find the smallest node, count the nodes in tree.**

```
package treepkg;
import java.util.Scanner;

class Node{
    int data;
    Node left,right;
    public Node(int val) {
        this.left=null;
        this.right=null;
        this.data=val;
    }
}

public class BinarySearchTree {

    public Node root;
    static int count=0;
    public BinarySearchTree() {
        this.root=null;
    }
    public static int nodeCount() {
        return count;
    }
    public void insertNode(int val) {
        Node newNode=new Node(val);
        if(root==null) {
            root=newNode;
            count++;
        }
        else {
            Node trav=root;
            Node hold=null;
            while(trav!=null) {
                hold=trav;
                if(val>trav.data) {
                    trav=trav.right;
                }
                else if(val<trav.data){
                    trav=trav.left;
                }
            }
            if(hold!=null) {
                if(val>hold.data) {
                    hold.right=newNode;
                }
                else {
                    hold.left=newNode;
                }
            }
        }
    }
}
```

```

        }
        else {
            System.out.println("Duplicate value");
            return;
        }
    }
    if(val>hold.data) {
        hold.right=newNode;
    }
    else {
        hold.left=newNode;
    }
    count++;
}
}
public void inOrder(Node root) {
    if(root !=null) {
        inOrder(root.left);
        System.out.print(root.data+" ");
        inOrder(root.right);
    }
}
public void inOrder() {
    inOrder(root);
}
public void preOrder(Node root) {
    if(root !=null) {
        System.out.print(root.data+" ");
        preOrder(root.left);
        preOrder(root.right);
    }
}
public void preOrder() {
    preOrder(root);
}
public void postOrder(Node root) {
    if(root !=null) {
        postOrder(root.left);
        postOrder(root.right);
        System.out.print(root.data+" ");
    }
}
public void postOrder() {

```

```
postOrder(root);
}

public void smallest() {
Node trav=root;
if(trav==null) {
    System.out.println("Tree is empty !!");
    return;
}
while(trav.left != null) {
    trav=trav.left;
}
System.out.println("Smallest node is : " +trav.data);
}

public void largest() {
Node trav=root;
if(trav==null) {
    System.out.println("Tree is empty !!");
    return;
}
while(trav.right != null) {
    trav=trav.right;
}
System.out.println("largest node is : " +trav.data);
}

public void search(int val) {
Node trav=root;
while(trav != null) {
    if(val >trav.data) {
        trav=trav.right;
    }
    else if(val<trav.data) {
        trav=trav.left;
    }
    else {
        System.out.println("Node with data "+val+" is found !!");
        return;
    }
}
System.out.println("Node with data "+val+" not found !!");
}

public void removeNode(int val) {
root=removeNodeRecursive(root,val);
}
```



```
        if(root!=null) {
            count--;
        }
    }
    public Node removeNodeRecursive(Node root,int val) {
        if(root==null) {
            System.out.println("Node not found !!");
            return root;
        }
        if(val<root.data) {
            root.left=removeNodeRecursive(root.left,val);
        }
        else if(val>root.data) {
            root.right=removeNodeRecursive(root.right,val);
        }
        else {
            if(root.left==null) {
                return root.right;
            }
            else if(root.right==null) {
                return root.left;
            }
            Node trav=root.left;
            while(trav.right != null) {
                trav=trav.right;
            }
            root.data=trav.data;
            root.left=removeNodeRecursive(root.left,root.data)
        }
    }
    return root;
}

public static void main(String[] args) {
    BinarySearchTree bt=new BinarySearchTree();
    int choice,data;
    Scanner sc=new Scanner(System.in);
    System.out.println("*****BINARY SEARCH TREE*****");
    do{
        System.out.println("1.Insert Node");
        System.out.println("2.Inorder Traversal");
        System.out.println("3.preOrder traversal");
        System.out.println("4.PostOrder traversal");
        System.out.println("5.Count Node");
        System.out.println("6.Largest node");
        System.out.println("7.Smallest node");
```

```
System.out.println("8.Search node");
System.out.println("9.Remove node");
System.out.println("10.exit");
System.out.println("---Enter your Choice---");

choice=sc.nextInt();
switch(choice) {
case 1:System.out.println("Enter Node data");
        data=sc.nextInt();
        bt.insertNode(data);
        break;
case 2:System.out.println("Inorder traversal : ");
        bt.inOrder();
        break;
case 3:System.out.println("preorder traversal : ");
        bt.preOrder();
        break;
case 4:System.out.println("Postorder traversal : ");
        bt.postOrder();
        break;
case 5:System.out.println("count of node : ");
        System.out.println(bt.nodeCount());
        break;
case 6:System.out.println("largest node : ");
        bt.largest();
        break;
case 7:System.out.println("smallest node : ");
        bt.smallest();
        break;
case 8:System.out.println("Enter Node data for search");
        data=sc.nextInt();
        bt.search(data);
        break;
case 9:System.out.println("Enter Node data for remove");
        data=sc.nextInt();
        bt.removeNode(data);
        break;
case 10:System.out.println("existing from program ");
        break;
default:System.out.println("Enter valid choice");
        break;
}
}while(choice!=10);
```

```

    }
}

```

Output

<pre> *****BINARY SEARCH TREE***** 1.Insert Node 2.Inorder Traversal 3.preOrder traversal 4.PostOrder traversal 5.Count Node 6.Largest node 7.Smallest node 8.Search node 9.Remove node 10.exit ---Enter your Choice--- 1 Enter Node data 10 </pre>	<pre> 1.Insert Node 2.Inorder Traversal 3.preOrder traversal 4.PostOrder traversal 5.Count Node 6.Largest node 7.Smallest node 8.Search node 9.Remove node 10.exit ---Enter your Choice--- 1 Enter Node data 30 </pre>
<pre> ---Enter your Choice--- 2 Inorder traversal : 10 20 30 1.Insert Node </pre>	<pre> ---Enter your Choice--- 3 preorder traversal : 10 20 30 1.Insert Node </pre>
<pre> ---Enter your Choice--- 4 Postorder traversal : 30 20 10 1.Insert Node </pre>	<pre> ---Enter your Choice--- 5 count of node : 3 </pre>
<pre> ---Enter your Choice--- 6 largest node : largest node is : 30 </pre>	<pre> ---Enter your Choice--- 8 Enter Node data for search 20 Node with data 20 is found !! </pre>
<pre> ---Enter your Choice--- 9 Enter Node data for remove 10 </pre>	<pre> ---Enter your Choice--- 2 Inorder traversal : 20 30 1.Insert Node </pre>

2. Write a Java program to create max heap and insert and delete node form heap. package maxHeap;

```

class Node {
    int data;
    Node prev, next;

    public Node(int val) {
        this.next = null;
        this.prev = null;
        this.data = val;
    }
}

public class MaxHeap{
    Node head, tail;

    public MaxHeap() {
        this.head = null;
        this.tail = null;
    }

    public void reheapUp(Node node) {
        Node parent = getParent(node);
        while (parent != null && node.data > parent.data) {
            int temp = node.data;
            node.data = parent.data;
            parent.data = temp;
            node = parent;
            parent = getParent(node);
        }
    }

    public void reheapDown(Node node) {
        while (node != null) {
            Node leftChild = node.next;
            Node rightChild = (leftChild != null) ? leftChild.next : null;
            if (leftChild == null) {
                break;
            }
            Node maxChild = leftChild;
            if (rightChild != null && rightChild.data > leftChild.data) {
                maxChild = rightChild;
            }
            if (node.data > maxChild.data) {
                break;
            }
            int temp = node.data;
            node.data = maxChild.data;
            maxChild.data = temp;
            node = maxChild;
        }
    }

    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {

```

```

        head = tail = newNode;
    } else {
        tail.next = newNode;
        newNode.prev = tail;
        tail = newNode;
        reheapUp(newNode);
    }
}

public void delete() {
    if(head==null) {
        System.out.println("Heap is empty !");
        return;
    }
    Node lastNode=tail;
    head.data=lastNode.data;
    if(tail.prev != null) {
        tail=tail.prev;
        tail.next=null;
    }
    else {
        head=null;
    }
    reheapDown(head);
}

private Node getParent(Node node) {
    return node.prev;
}

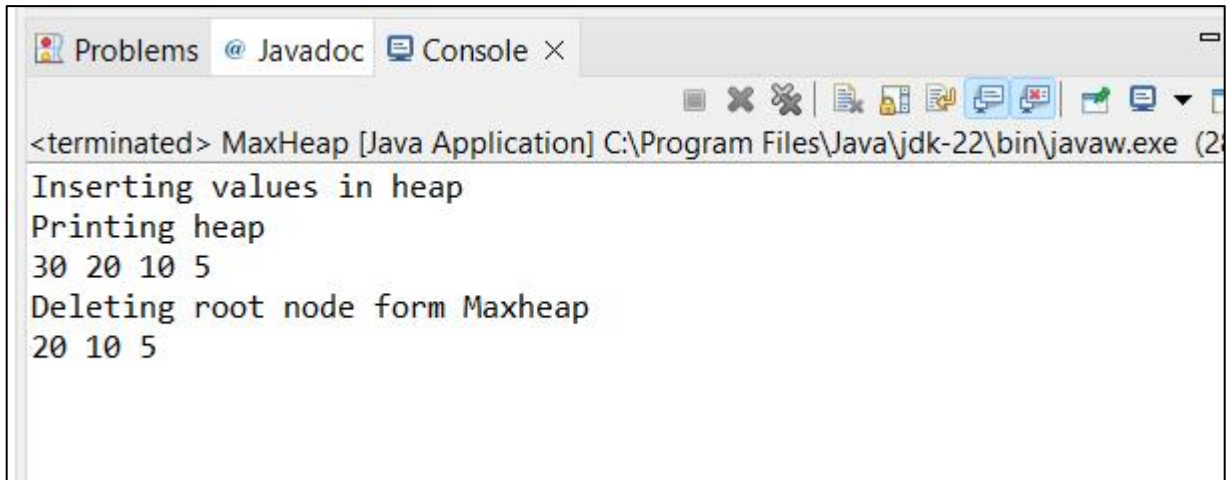
public void printHeap() {
    Node temp=head;
    while(temp!=null) {
        System.out.print(temp.data+ " ");
        temp=temp.next;
    }
    System.out.println();
}

public static void main(String[] args) {
    // TODO Auto-generated method stub

    MaxHeap hp=new MaxHeap();
    System.out.println("Inserting values in heap");
    hp.insert(10);
    hp.insert(20);
    hp.insert(5);
    hp.insert(30);
    System.out.println("Printing heap");
    hp.printHeap();
    System.out.println("Deleting root node form Maxheap");
    hp.delete();
    hp.printHeap();
}
}

```

Output



```
<terminated> MaxHeap [Java Application] C:\Program Files\Java\jdk-22\bin\javaw.exe (2
Inserting values in heap
Printing heap
30 20 10 5
Deleting root node form Maxheap
20 10 5
```

3. Write a Java program to create min heap and insert and delete node form heap.

```
package MinHeap;
class Node {
    int data;
    Node prev, next;

    public Node(int val) {
        this.next = null;
        this.prev = null;
        this.data = val;
    }
}
public class MinHeapImplement {
    Node head, tail;

    public MinHeapImplement() {
        this.head = null;
        this.tail = null;
    }

    public void reheapUp(Node node) {
        Node parent = getParent(node);
        while (parent != null && node.data < parent.data) {
            int temp = node.data;
            node.data = parent.data;
            parent.data = temp;
            node = parent;
            parent = getParent(node);
        }
    }

    public void reheapDown(Node node) {
```

```

        while (node != null) {
            Node leftChild = node.next;
            Node rightChild = (leftChild != null) ? leftChild.next : null;
            if (leftChild == null) {
                break;
            }
            Node minChild = leftChild;
            if (rightChild != null && rightChild.data < leftChild.data) {
                minChild = rightChild;
            }
            if (node.data <= minChild.data) {
                break;
            }
            int temp = node.data;
            node.data = minChild.data;
            minChild.data = temp;
            node = minChild;
        }
    }

    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = tail = newNode;
        } else {
            tail.next = newNode;
            newNode.prev = tail;
            tail = newNode;
            reheapUp(newNode);
        }
    }

    public void delete() {
        if (head == null) {
            System.out.println("Heap is empty !");
            return;
        }
        Node lastNode = tail;
        head.data = lastNode.data;
        if (tail.prev != null) {
            tail = tail.prev;
            tail.next = null;
        }
        else {
            head = null;
        }
        reheapDown(head);
    }

    private Node getParent(Node node) {
        return node.prev;
    }
}

```

```
public void printHeap() {
    Node temp=head;
    while(temp!=null) {
        System.out.print(temp.data+ " ");
        temp=temp.next;
    }
    System.out.println();
}

public static void main(String[] args) {
    // TODO Auto-generated method stub
    MinHeapImplement hp=new MinHeapImplement();
    System.out.println("Inserting values in minheap");
    hp.insert(10);
    hp.insert(20);
    hp.insert(5);
    hp.insert(8);
    System.out.println("Printing heap");
    hp.printHeap();
    System.out.println("Deleting root node form Minheap");
    hp.delete();
    hp.printHeap();
}
}
```

Output



The screenshot shows a Java IDE console window with the following output:

```
<terminated> MinHeapImplement [Java Application] C:\Program Files\Java\
Inserting values in minheap
Printing heap
5 8 10 20
Deleting root node form Minheap
8 10 20
```


Conclusion:

Binary Search Trees (BSTs) and Heaps are both essential tree-based data structures, each optimized for different purposes.

BSTs are ideal for efficiently managing and querying sorted data, with operations like insertion, deletion, and search performing best when the tree is balanced. They are particularly useful for tasks that require in-order traversal to retrieve sorted data or range queries.

Heaps, on the other hand, are specialized for priority-based operations. Max-Heaps quickly retrieve the maximum value, while Min-Heaps efficiently provide the minimum. These properties make Heaps ideal for implementing priority queues, heap sort, and scheduling algorithms.

While BSTs focus on maintaining order and supporting range-based queries, Heaps prioritize efficient access to the highest or lowest values. Both structures play critical roles in algorithm design, and their use depends on the specific requirements of the application.