**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**.

**Code:**

```
import java.util.Scanner;

class Node{
	int data;
	Node left,right;

	public Node( int val) {
		this.data=val;
		this.left=null;
		this.right=null;
	}
}
public class BinarySearchTree {
	private Node rootNode;
	static int count=0;

	public BinarySearchTree()
	{
		this.rootNode=null;
	}
	public static int nodeCounts()
	{
		return count;
	}
	public void insertNode(int val)
	{
		Node newNode=new Node(val);
		if (rootNode==null)
		{
			rootNode=newNode;
			count++;
		}
		else {
			Node travNode=rootNode;
			Node holdNode=null;
			while (travNode!=null)
			{
				holdNode=travNode;
				if (val>travNode.data)
				{
					travNode=travNode.right;
```

```
                        }
                        else if (val<travNode.data)
                        {
                                travNode=travNode.left;
                        }
                        else {
                                System.out.println("Duplicate Data");
                                return;
                        }
                }
                if (val>holdNode.data)
                {
                        holdNode.right=newNode;
                }
                else {
                        holdNode.left=newNode;
                }
                count++;
        }
}
public void inorder(Node root)
{
        if (root!=null)
        {
                inorder(root.left);
                System.out.println(root.data+" ");
                inorder(root.right);
        }
}
public void inorder()
{
        inorder(rootNode);
}
public void preorder(Node root)
{
        if (root!=null)
        {
                System.out.println(root.data+" ");
                preorder(root.left);
                preorder(root.right);
        }
}
public void preorder()
{
        preorder(rootNode);
}
public void postorder(Node root)
```

```
{
   if (root != null)
   {
      postorder(root.left);
      postorder(root.right);
      System.out.print(root.data + " ");
   }
}
public void postorder() {
   postorder(rootNode);
}
      public void smallest()
      {
            Node trav=rootNode;
            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=rootNode;
            if (trav==null)
            {
                  System.out.println("Tree is empty");
                  return;
            }
            while (trav.right!=null)
            {
                  trav=trav.right;
            }
            System.out.println("Smallest Node is : " + trav.data);
      }
      public void search(int val)
      {
            Node travNode=rootNode;
            while (travNode!=null)
            {
                  if (val>travNode.data)
                  {
                        travNode=travNode.right;
```

```
                }
                else if (val>travNode.data)
                {
                        travNode=travNode.left;
                }
                else {
                        System.out.println("Node with data "+val+" is  found");
                        return;
                }
        }
        System.out.println("Node with data "+ val+" is not found!!");
}
public void removeNode(int val)
{
        rootNode=removeNodeRecusrsive(rootNode,val);
        if (rootNode!=null)
        {
                count--;
        }
}
public Node removeNodeRecusrsive(Node rootNode,int val)
{
        if (rootNode==null)
        {
                System.out.println("Node Not found!!!");
                return rootNode;
        }
        if (val<rootNode.data)
        {
                rootNode.left=removeNodeRecusrsive(rootNode.left, val);
        }
        else if (val>rootNode.data)
        {
                rootNode.right=removeNodeRecusrsive(rootNode.right, val);
        }
        else {
                if (rootNode.left==null)
                {
                        return rootNode.right;
                }
                else if (rootNode.right==null)
                {
                        return rootNode.left;
                }
                Node travNode=rootNode.left;
                while (travNode.right!=null)
                {
```

```
                    travNode=travNode.right;
                }
                rootNode.data=travNode.data;
                rootNode.left=removeNodeRecusrsive(rootNode.left,
rootNode.data);
            }
            return rootNode;
    }

    public static void main(String[] args) {
    BinarySearchTree bt = new BinarySearchTree();
    //45,39,56,12,34,78,32,10,89,54,67,81

    Scanner sc = new Scanner(System.in);

    int data;

    int choice;

    System.out.print("\nBinary Search Tree\n\n");
      do
      {
            System.out.print("\n1.Insert Node\n");
            System.out.print("2.InOrder Traversal\n");
            System.out.print("3.PreOrder Traversal\n");
            System.out.print("4.PostOrder Traversal\n");
            System.out.print("5.Smallest Node\n");
            System.out.print("6.Largest Node\n");
            System.out.print("7.Count Nodes\n");
            System.out.print("8.Remove Node\n");
            System.out.print("9.Search Node\n");
            System.out.print("10.Exit\n");
            System.out.print("Enter your choice : ");
            choice = sc.nextInt();
            switch (choice)
            {
            case 1: System.out.print("\nInsert Node - Enter data : ");
                    data= sc.nextInt();
                    bt.insertNode(data);
                    break;
            case 2: System.out.print("\nInOrder Traversal : ");
                        bt.inorder();
                        break;
            case 3: System.out.print("\nPreOrder Traversal : ");
                        bt.preorder();
                    break;
            case 4: System.out.print("\nPostOrder Traversal : ");
```

```
                                bt.postorder();
                                break;
                        case 5: System.out.print("\nSmallest node is : ");
                                bt.smallest();
                                break;
                        case 6: System.out.print("\nLargest node is : ");
                                bt.largest();
                                break;
                        case 7: System.out.print("\nTotal node count : " + nodeCounts());
                                break;
                        case 8: System.out.print("\nRemove node - Enter key : ");
                                data=sc.nextInt();
                                bt.removeNode(data);
                                break;
                        case 9: System.out.print("\nSearch node - Enter Data : ");
                                data=sc.nextInt();
                                bt.search(data);
                                break;
                        case 10: System.out.println("Exiting the program.");
                                break;
                        default:
                                System.out.print("\nWrong choice! \n");

                        }
                }while(choice!=10);
                sc.close();
        }
}
```

**Output:**

```
Binary Search Tree


1.Insert Node
2.InOrder Traversal
3.PreOrder Traversal
4.PostOrder Traversal
5.Smallest Node
6.Largest Node
7.Count Nodes
8.Remove Node
9.Search Node
10.Exit
Enter your choice : 1

Insert Node - Enter data : 45
```

```
Enter your choice : 1

Insert Node - Enter data : 39
```

```
Enter your choice : 1

Insert Node - Enter data : 56
```

```
Enter your choice : 1

Insert Node - Enter data : 12
```

```
Enter your choice : 1

Insert Node - Enter data : 34
```

```
Enter your choice : 1

Insert Node - Enter data : 78
```

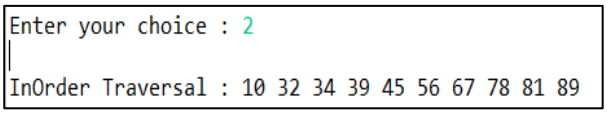
```
Enter your choice : 1

Insert Node - Enter data : 32
```

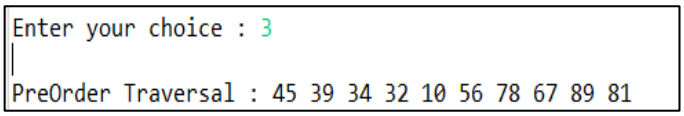
```
Enter your choice : 1

Insert Node - Enter data : 10
```

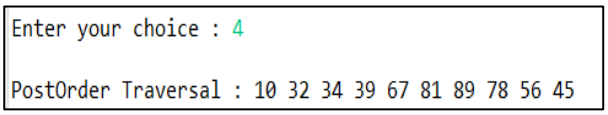
```
Enter your choice : 1

Insert Node - Enter data : 67
```

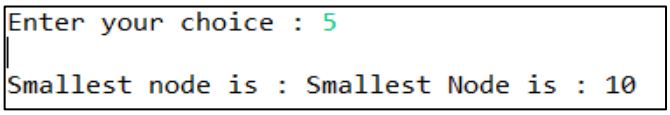
```
Enter your choice : 1

Insert Node - Enter data : 89
```

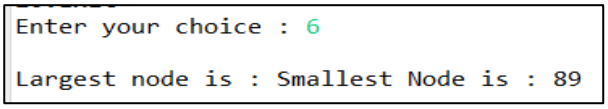
```
Enter your choice : 1

Insert Node - Enter data : 81
```

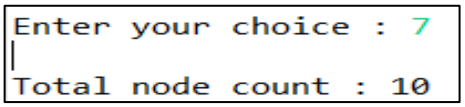


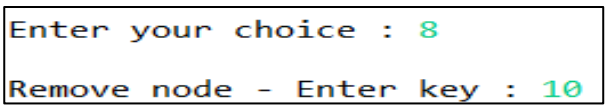



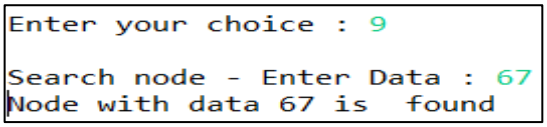




**2. Write a Java program to create max heap and insert and delete node form heap.**

```
package max;

class Node{
	int data;
	Node next,prev;

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

```
	private void reheapDown(Node node)
	{
		while (node!=null)
		{
			Node leftChiledNode=node.next;
			Node rightChildNode=(leftChiledNode!=null)?
leftChiledNode.next:null;
			if (leftChiledNode==null) {
				break;
			}
			Node maxChildNode=leftChiledNode;
			if (rightChildNode!=null &&
rightChildNode.data>leftChiledNode.data) {
				maxChildNode=rightChildNode;
			}
			if (node.data>=maxChildNode.data) {
				break;
			}
			int temp=node.data;
			node.data=maxChildNode.data;
			maxChildNode.data=temp;
			node=maxChildNode;
		}
	}
	public void insert(int data)
	{
		Node newNode=new Node(data);
		if (head==null) {
			head=newNode;
			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 tempNode=head;
            while (tempNode!=null) {
                System.out.println(tempNode.data+"  ");
                tempNode=tempNode.next;
            }
            System.out.println();
        }
        public static void main(String[] args) {
            MaxHeap maxHeap=new MaxHeap();
            System.out.println("\nIsnerting value in MaxHeap");
            maxHeap.insert(10);
            maxHeap.insert(20);
            maxHeap.insert(5);
            maxHeap.insert(8);
            maxHeap.printHeap();
            System.out.println("Deleting root node from MaxHeap");
            maxHeap.delete();
            maxHeap.printHeap();
        }

}
```

**Output:**

```
Isnerting value in MaxHeap
20
10
8
5

Deleting root node from MaxHeap
10
8
5
```

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

**Code:**

```
package mean;

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

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

    private void reheapDown(Node node) {
        while (node != null) {
            Node leftChildNode = node.next;
            Node rightChildNode = (leftChildNode != null) ? leftChildNode.next : null;

            if (leftChildNode == null) {
                break;
            }
            Node minNode = leftChildNode;
            if (rightChildNode != null && rightChildNode.data < leftChildNode.data) {
                minNode = rightChildNode;
            }
            if (node.data <= minNode.data) {
                break;
            }
            int temp = node.data;
```

```
            node.data = minNode.data;
            minNode.data = temp;
            node = minNode;
        }
    }
    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            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 tempNode = head;
        while (tempNode != null) {
            System.out.print(tempNode.data + "  ");
            tempNode = tempNode.next;
        }
        System.out.println();
    }
    public static void main(String[] args) {
        MeanHeap minHeap = new MeanHeap();
        System.out.println("\nInserting values into MinHeap");
        minHeap.insert(10);
```

```
        minHeap.insert(20);
        minHeap.insert(5);
        minHeap.insert(8);
        minHeap.printHeap();
        System.out.println("\nDeleting root node from MinHeap");
        minHeap.delete();
        minHeap.printHeap();
    }
}
```

## Output:

```
Inserting values into MinHeap
5  8  10  20

Deleting root node from MinHeap
8  10  20
```

## Conclusion:

The provided code demonstrates key operations of **binary search trees (BSTs)** and **heaps**, which are essential data structures in computer science. The BST example includes inserting nodes, traversals (in-order, pre-order, and post-order), finding the smallest and largest nodes, counting nodes, searching, and removing nodes. BSTs are ideal for organizing data efficiently for quick searching and sorting.

The heap examples cover **max-heaps** (where the parent is greater than its children) and **min-heaps** (where the parent is smaller). They showcase inserting and deleting nodes while maintaining heap properties using reheapification. These structures are vital for efficient data retrieval and priority-based tasks. Together, they emphasize the importance of trees in organizing and managing data.