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

**Code:-**

# package bst;

**import** java.util.Scanner; **import** java.util.\*;

**class** Node { **int** data; Node left, right;

# Node(int val) { this.data = val; this.left = null; this.right = null;

}

}

**public** **class** BinarySearchTree

{

# private Node root; static int *count* = 0;

**public** BinarySearchTree()

{

# this.root = null;

}

# public static int nodeCounts()

{

# return *count*;

}

// Insert a node into the binary tree **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; **else**

{

System.***out***.println("Duplicate data"); **return**;

} }

**if** (val > hold.data) hold.right = newNode; **else**

hold.left = newNode; *count*++;

}

}

// In-order traversal **public** **void** inorder(Node root)

{ **if** (root != **null**) { inorder(root.left);

System.***out***.print(root.data + " "); inorder(root.right);

}

}

**public** **void** inorder() { inorder(root);

}

// Pre-order traversal **public** **void** preorder(Node root)

{

# if (root != null)

{

System.***out***.print(root.data + " "); preorder(root.left); preorder(root.right);

}

} **public** **void** preorder() { preorder(root);

}

// Post-order traversal **public** **void** postorder(Node root)

{

# if (root != null)

{

postorder(root.left); postorder(root.right);

System.***out***.print(root.data + " ");

}

}

**public** **void** postorder() { postorder(root);

}

// Find the smallest element **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);

}

// Find the largest element **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);

}

// Search for a value in the tree **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 + " is not found!");

}

// Remove a node from the tree **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**

{

// Node with one child or no child **if** (root.left == **null**) **return** root.right; **else** **if** (root.right == **null**) **return** root.left;

// Node with two children: Get the inorder predecessor

Node trav = root.left; **while** (trav.right != **null**) trav = trav.right; root.data = trav.data;

root.left = removeNodeRecursive(root.left, root.data);

# } return root;

}

/\* private int findLargest(Node root) { while (root.right != null) root = root.right; return root.data;

} \*/

**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");

} /\*End of switch\*/

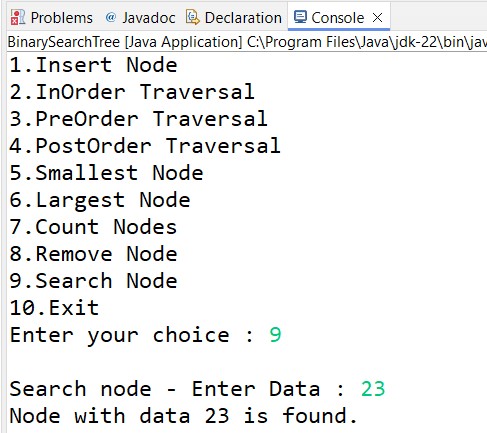
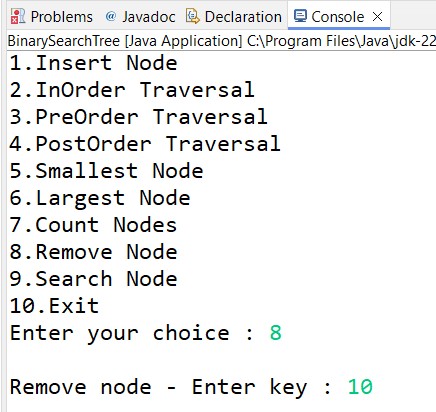
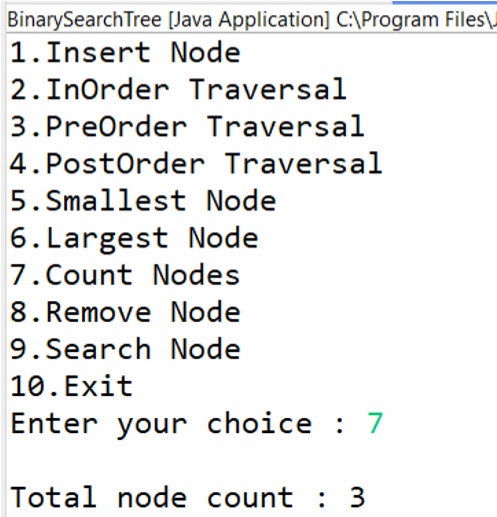
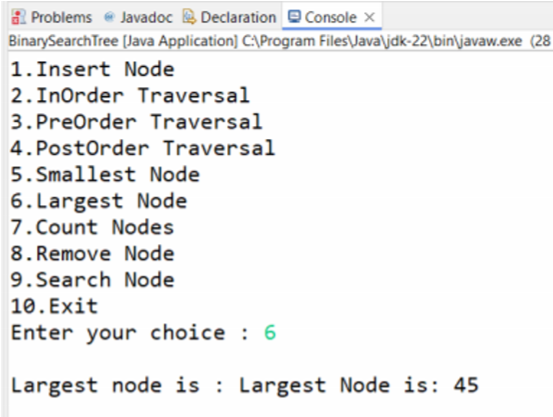
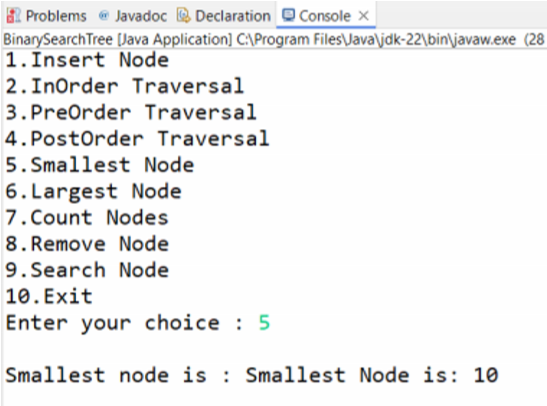
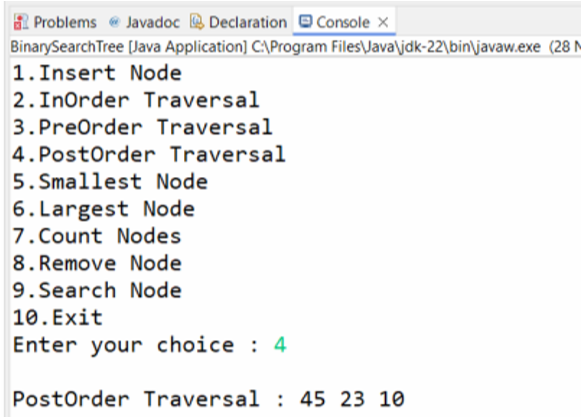
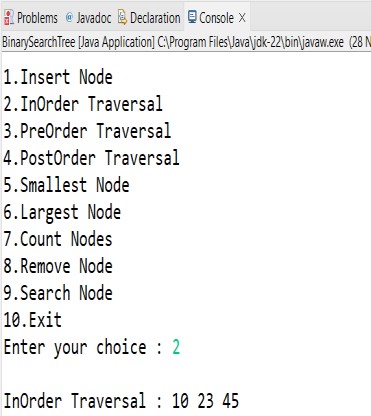
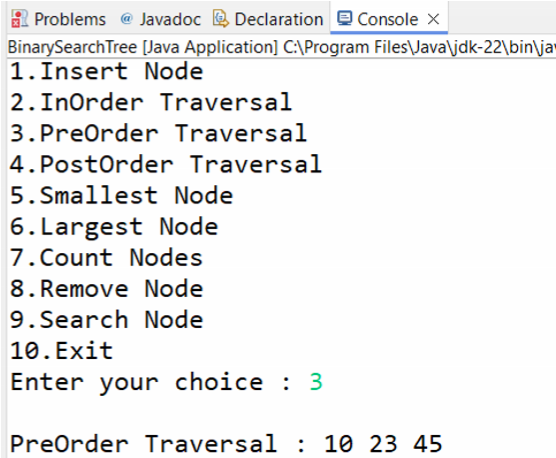
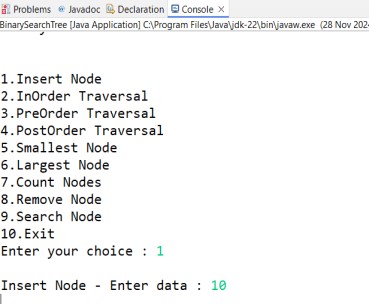
}**while**(choice!=10); /\*End of while\*/ sc.close();

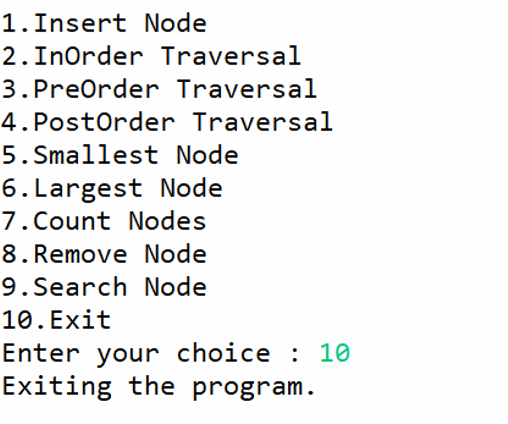
}

}

**Output:**

**-**





## 2. Write a Java program to create max heap and insert and delete node form heap. Code:-

# package heap;

**import** java.util.Scanner;

**class** Node { **int** data; Node prev, next;

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

}

}

# public class MaxHeap {

Node head, tail;

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

}

// Re-heap Up for MaxHeap **private** **void** reheapUp(Node node) { Node parent = getParent(node);

**while** (parent != **null** && node.data > parent.data)

{

// Swap the node with its parent **int** temp = node.data; node.data = parent.data; parent.data = temp; node = parent;

parent = getParent(node);

}

}

// Re-heap Down for Maxheap

**private** **void** reheapDown(Node node) { **while** (node != **null**) {

Node leftChild = node.next;

Node rightChild = (leftChild != **null**) ? leftChild.next : **null**;

// If there are no children, break out of the loop

# if (leftChild == null) { break;

}

Node maxChild = leftChild;

**if** (rightChild != **null** && rightChild.data > leftChild.data) {

maxChild = rightChild;

}

// If the node is greater than or equal to its largest child, stop

**if** (node.data >= maxChild.data) { **break**;

}

// Swap node with the largest child **int** temp = node.data; node.data = maxChild.data; maxChild.data = temp;

node = maxChild;

}

}

// Insert a node in the MaxHeap **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;

// Reheap up from the last inserted node reheapUp(newNode);

}

}

// Delete the root node in the MaxHeap **public** **void** delete() { **if** (head == **null**) {

System.***out***.println("Heap is Empty!!"); **return**;

}

Node lastNode = tail;

// Move the last node to the root position head.data = lastNode.data;

// Remove the last Node **if** (tail.prev != **null**) { tail = tail.prev; tail.next = **null**;

# } else { head = null;

}

// Re-heap down from the root reheapDown(head);

}

// Get the parent of a node in the doubly linked list **private** Node getParent(Node node) {

# return node.prev;

}

// Print the heap **public** **void** printHeap() { Node temp = head; **if** (temp == **null**) {

System.***out***.println("Heap is empty."); **return**;

}

# while (temp != null) {

System.***out***.print(temp.data + " "); temp = temp.next; }

System.***out***.println();

}

// Main function with choices

**public** **static** **void** main(String[] args) {

MaxHeap maxHeap = **new** MaxHeap();

Scanner scanner = **new** Scanner(System.***in***);

# while (true) {

System.***out***.println("\nMax Heap Operations

Menu:");

System.***out***.println("1. Insert a value into

MaxHeap");

System.***out***.println("2. Delete the root node from MaxHeap");

System.***out***.println("3. Print the MaxHeap");

System.***out***.println("4. Exit");

System.***out***.print("Enter your choice: ");

**int** choice = scanner.nextInt();

# switch (choice) { case 1:

System.***out***.print("Enter a value to insert into MaxHeap: ");

**int** value = scanner.nextInt(); maxHeap.insert(value); **break**; **case** 2:

System.***out***.println("Deleting root node from MaxHeap...");

maxHeap.delete(); **break**; **case** 3:

System.***out***.println("MaxHeap:"); maxHeap.printHeap();

# break; case 4:

System.***out***.println("Exiting the program...");

scanner.close(); **return**; // Exit the program **default**:

System.***out***.println("Invalid choice, please try again.");

}

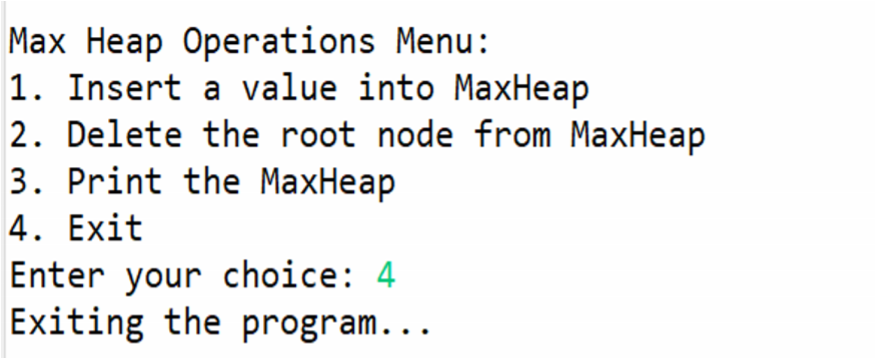
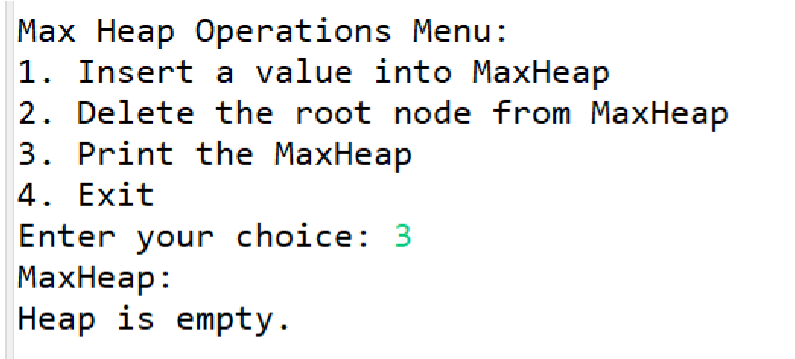
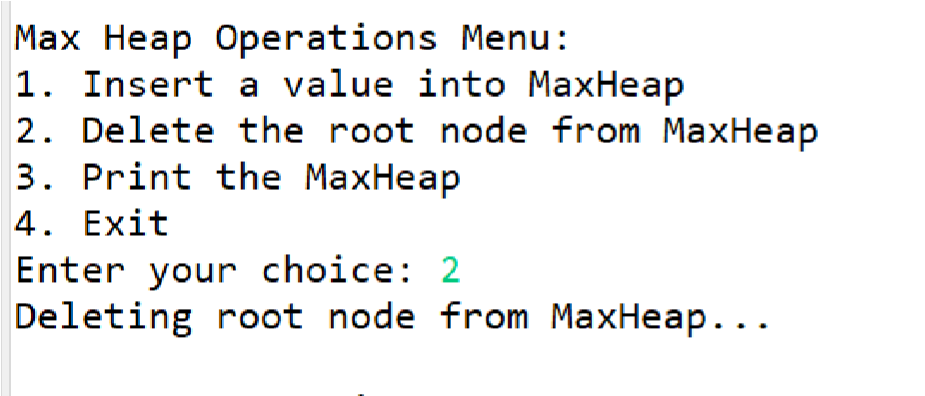
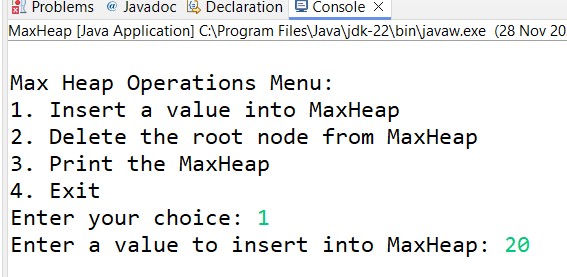
}

}

}

**Output:**

**-**



## 3. Write a Java program to create min heap and insert and delete node form heap. Code:-

**package** MinHeap; **import** java.util.Scanner;

**class** Node {

**int** data; Node prev, next;

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

}

}

# public class MinHeap {

Node head, tail;

# public MinHeap() { this.head = null; this.tail = null;

}

// Re-heap Up for MinHeap **private** **void** reheapUp(Node node) { Node parent = getParent(node);

**while** (parent != **null** && node.data < parent.data)

{ // Reverse the comparison for MinHeap // Swap the node with its parent **int** temp = node.data; node.data = parent.data; parent.data = temp;

node = parent;

parent = getParent(node);

}

}

// Re-heap Down for MinHeap **private** **void** reheapDown(Node node) { **while** (node != **null**) {

Node leftChild = node.next;

Node rightChild = (leftChild != **null**) ? leftChild.next : **null**;

// If there are no children, break out of the loop

# if (leftChild == null) { break;

}

Node minChild = leftChild;

**if** (rightChild != **null** && rightChild.data < leftChild.data) { // Reverse the comparison for MinHeap minChild = rightChild;

}

// If the node is less than or equal to its smallest child, stop

**if** (node.data <= minChild.data) { // Reverse the comparison for MinHeap **break**;

}

// Swap node with the smallest child **int** temp = node.data; node.data = minChild.data; minChild.data = temp;

node = minChild;

}

}

// Insert a node in the MinHeap **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;

// Reheap up from the last inserted node reheapUp(newNode);

}

}

// Delete the root node in the MinHeap **public** **void** delete() { **if** (head == **null**) {

System.***out***.println("Heap is Empty!!"); **return**;

}

Node lastNode = tail;

// Move the last node to the root position head.data = lastNode.data;

// Remove the last Node **if** (tail.prev != **null**) { tail = tail.prev; tail.next = **null**;

# } else { head = null;

}

// Re-heap down from the root reheapDown(head);

}

// Get the parent of a node in the doubly linked list **private** Node getParent(Node node) {

# return node.prev;

}

// Print the heap **public** **void** printHeap() { **if** (head == **null**) { System.***out***.println("Heap is empty."); **return**;

}

Node temp = head; **while** (temp != **null**) {

System.***out***.print(temp.data + " "); temp = temp.next; }

System.***out***.println();

} **public** **static** **void** main(String[] args) { MinHeap minHeap = **new** MinHeap();

Scanner sc = **new** Scanner(System.***in***);

**int** choice, data;

# do {

System.***out***.println("\nMinHeap Operations

Menu:");

System.***out***.println("1. Insert Node");

System.***out***.println("2. Delete Root Node");

System.***out***.println("3. Print Heap");

System.***out***.println("4. Exit"); System.***out***.print("Enter your choice: "); choice = sc.nextInt();

# switch (choice) { case 1:

System.***out***.print("\nEnter data to insert: ");

data = sc.nextInt(); minHeap.insert(data); **break**; **case** 2:

System.***out***.println("\nDeleting root node from MinHeap..."); minHeap.delete(); **break**; **case** 3:

System.***out***.println("\nCurrent MinHeap:

");

minHeap.printHeap();

# break; case 4:

System.***out***.println("Exiting the program.");

# break; default:

System.***out***.println("Invalid choice! Please try again."); **break**;

}

} **while** (choice != 4);

sc.close();

}

}

**Output:**

**-**

