DATA STRUCTURES LAB

ACTIVITY 1 SOLUTION

Binary Search Tree and Min-Heap Implementation

Topic:	Trees and Heaps
Date:	February 2026
Programming Language:	Java
Tools Used:	Java SE, PriorityQueue

1. Introduction

This document presents the complete solution for Activity 1 of the Data Structures Lab. The activity focuses on implementing Binary Search Tree (BST) operations and understanding Min-Heap functionality using Java. The implementation includes insertion, searching, and traversal operations for BST, along with heap operations using Java's PriorityQueue.

2. Task 1: Binary Search Tree Creation

In this task, we created a Binary Search Tree class with methods for insertion. The following elements were inserted: 50, 30, 70, 20, 40, 60, 80.

2.1 BST Structure After Insertion

The tree structure forms with 50 as the root. Elements less than the root go to the left subtree, and elements greater go to the right subtree. This property is maintained recursively for all nodes.

```
      50
     /  \
   30    70
  / \   / \
20 40 60 80
```

3. Task 2: Tree Traversals

Three different traversal methods were implemented to visit all nodes in the BST. Each traversal follows a specific order:

3.1 Inorder Traversal (Left-Root-Right)

This traversal visits the left subtree first, then the root node, and finally the right subtree. For a BST, inorder traversal gives elements in sorted ascending order.

Output: 20 30 40 50 60 70 80

Explanation: The output is sorted because of the BST property. Starting from the leftmost node (20), we visit each node in ascending order.

3.2 Preorder Traversal (Root-Left-Right)

This traversal visits the root node first, then recursively visits the left and right subtrees. It is useful for creating a copy of the tree.

Output: 50 30 20 40 70 60 80

Explanation: We start at root (50), then go to left subtree (30, 20, 40), and finally the right subtree (70, 60, 80).

3.3 Postorder Traversal (Left-Right-Root)

This traversal visits both subtrees before visiting the root node. It is useful for deleting the tree as we process children before parents.

Output: 20 40 30 60 80 70 50

Explanation: We process all leaf nodes first, then their parents, and finally reach the root as the last node.

4. Task 3: Search Operation

The search operation in BST is efficient because we can eliminate half of the tree at each step by comparing with the current node's value.

4.1 Searching for Element 40

Result: FOUND

Search Path: 50 → 30 → 40

Explanation: Starting from root (50), since 40 < 50, we go left to 30. Since 40 > 30, we go right and find 40.

4.2 Searching for Element 100

Result: NOT FOUND

Search Path: 50 → 70 → 80 → null

Explanation: Starting from root (50), since 100 > 50, we go right to 70. Since 100 > 70, we go right to 80. Since 100 > 80, we try to go right but find null, confirming the element doesn't exist in the tree.

5. Task 4 & 5: Min-Heap Implementation

We used Java's PriorityQueue class which implements a Min-Heap by default. A Min-Heap is a complete binary tree where each parent node is smaller than its children.

5.1 Random Number Generation and Insertion

Ten random numbers were generated and inserted into the Min-Heap. Example: 45, 12, 78, 23, 67, 89, 34, 56, 90, 15

5.2 Sorted Output

By repeatedly polling from the Min-Heap, we get elements in sorted order: 12, 15, 23, 34, 45, 56, 67, 78, 89, 90

Explanation: The poll() method removes and returns the minimum element (root) and automatically restructures the heap to maintain the Min-Heap property.

6. Time Complexity Analysis

6.1 Binary Search Tree Operations

Operation	Best Case	Average Case	Worst Case
Insert	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(n)
Delete	O(log n)	O(log n)	O(n)
Inorder Traversal	O(n)	O(n)	O(n)
Preorder Traversal	O(n)	O(n)	O(n)
Postorder Traversal	O(n)	O(n)	O(n)

Notes on BST Complexity:

• Best/Average case occurs when the tree is balanced (height = log n)
• Worst case occurs when the tree becomes skewed (like a linked list, height = n)
• Traversals always visit all n nodes, hence O(n) in all cases
• Space complexity for recursion: O(h) where h is the height of the tree

6.2 Min-Heap Operations

Operation	Time Complexity	Description
Insert	O(log n)	Element added at end, then bubbled up
Extract Min	O(log n)	Root removed, last element moved to root, then bubbled down
Get Min	O(1)	Simply return the root element
Heapify	O(n)	Build heap from unsorted array
Delete	O(log n)	Similar to extract, followed by heapify

Notes on Heap Complexity:

• Heap is always a complete binary tree, so height is always log n
• Insert and delete operations require bubbling up/down the height of the tree
• Space complexity: O(n) for storing n elements
• Heaps are more balanced than BST, guaranteeing logarithmic operations

7. Program Output

Below is the complete output from running the Java program:

```
========================================
DATA STRUCTURES LAB - ACTIVITY 1
Binary Search Tree and Min-Heap
========================================

TASK 1: Creating Binary Search Tree
--------------------------------------
Inserting elements: 50 30 70 20 40 60 80
BST created successfully!

TASK 2: Tree Traversals
--------------------------------------
Inorder Traversal: 20 30 40 50 60 70 80
Preorder Traversal: 50 30 20 40 70 60 80
Postorder Traversal: 20 40 30 60 80 70 50

TASK 3: Searching Elements in BST
--------------------------------------
Searching for 40: FOUND
Searching for 100: NOT FOUND

TASK 4 & 5: Min-Heap using PriorityQueue
--------------------------------------
Inserting 10 random numbers: 45 12 78 23 67 89 34 56 90 15

Numbers in sorted order (Min-Heap): 12 15 23 34 45 56 67 78 89 90

========================================
TIME COMPLEXITY ANALYSIS
========================================

Binary Search Tree Operations:
  - Insert: O(h) where h is height
    Best case: O(log n) for balanced tree
    Worst case: O(n) for skewed tree
  - Search: O(h) where h is height
    Best case: O(log n) for balanced tree
    Worst case: O(n) for skewed tree
  - Traversal: O(n) - visits all nodes

Min-Heap Operations:
  - Insert: O(log n)
  - Extract Min: O(log n)
  - Get Min: O(1)
  - Heapify: O(n)

========================================
```

8. Conclusion

This activity successfully demonstrated the implementation and understanding of Binary Search Trees and Min-Heaps in Java. Key learnings include:

• BST provides efficient searching when balanced, with O(log n) average time complexity
• Different traversal methods serve different purposes in tree manipulation
• Min-Heap guarantees O(log n) insertion and extraction with complete tree structure
• Java's PriorityQueue provides an efficient built-in Min-Heap implementation
• Understanding time complexity helps in choosing appropriate data structures

Both BST and Heap are fundamental data structures with unique advantages. BST excels at searching and maintaining sorted order, while Heaps are optimal for priority queue operations and finding minimum/maximum elements efficiently.