

# Algorithms and Data Structures (ADS2)

## Laboratory Sheet 4

This lab sheet contains material based on Lectures 13-14. This exercise is not assessed but you are advised to complete it to gain sufficient experience in the implementation of Binary Search Trees (BSTs) and related algorithms.

### Note

You are not expected to complete the exercise during your scheduled lab session. Some extra work might be required before and after your lab session.

### Exercise

You are to implement the BST data structure and some of its variants.

#### Part 1

- a) Implement the BST data structure including the following operations: `search`, `min`, `max`, `insert`, and `delete`. Use the following class structure.

```
import java.util.NoSuchElementException;

public class BST {
    private Node root;           // root of BST

    private class Node {
        private int key;          // sorted by key
        private Node left, right, p;
        private int size;         // number of nodes

        public Node(int key, int size) {
            this.key = key;
            this.size = size;
            this.left = null;
            this.right = null;
            this.p = null;
        }
    }

    public BST() {
        root = null;
    }

    ...
}
```

- b) Implement `inOrder`, `preOrder`, and `postOrder` traversals.
- c) Implement method `checkBST` that verifies whether a binary tree satisfies the BST property. Hint: checking if left node is smaller than the node and right node is greater than the node is not enough as the BST constraints apply to the whole left and right subtrees.

#### Part 2

The standard implementation of insertion in BSTs introduces new values as leaves of the tree. These require the traversal of a full path to be retrieved, making BST non suitable for applications in which newly inserted values are more likely to be retrieved again. Implement the following two strategies to alleviate this problem.

- a) Implement `insertRoot`, a method that inserts a new node at the root of a BST. Show that the last key inserted in the tree can be retrieved in  $O(1)$  time. Hint: rotations are required to bring the new node up.
- b) Reimplement `search` in such a way that a retrieved key is first removed from the BST and then reinserted at the root. Explain why this strategy is typically used to guarantee fast access to the keys that are used more frequently.

### Part 3

Extend class `Node` with attribute `private int count` storing the number of duplicates of a given key. Then, modify methods `insert` and `delete` to handle duplicates correctly.

### Part 4

Implement the following algorithm to balance a BST:

1. Perform an inorder traversal of the tree and store the keys in a sorted array.
2. Pick the middle element of the array as the root of the new tree.
3. Recursively do the same with the left half of the array and the right half of the array to get the roots of the left and right subtrees, respectively.

What is the complexity of this algorithm?