Algorithms and Data Structures (ADS2)

Laboratory Sheet 4

This lab sheet contains material based on Lectures 10-11. 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;
  }

  private int size(Node x){
    if (x == null) return 0;
    else return x.size;
  }

  public int size(){
    return size(root);
  }

  public boolean isEmpty(){
     return size() == 0;
  }

  private Node min(Node x){
    if (x.left == null) return x;
    else                return min(x.left);
  }

  public Node min(){
    if (isEmpty()) throw new NoSuchElementException("Empty BST ");
    return min(root);
  }

  private Node max(Node x){
    if (x.right == null) return x;
    else                return min(x.right);
  }

  public Node max(){
    if (isEmpty()) throw new NoSuchElementException("Empty BST ");
    return max(root);
  }

  private Node search(Node x, int key){
    if (x == null || x.key == key) return x;
    if (x.key < key) return search(x.left, key);
    else return search(x.right, key);
  }

  public Node search(int key){
    return search(root, key);
  }

  public insert(Node z){
    Node y = null;
    Node x = root;
    while (x != null){
      y = x;
      y.size = y.size + 1;
      if (z.key < x.key){
        x = x.left;
      }
      else {
        x = x.right;
      }
    }
    z.p = y;
    z.size = 1;
    if (y == null){
      root = z;
    }
    else {
      if (z.key < y.key){
        y.left = z;
      }
      else {
        y.right = z;
      }
    }

...
```

b) Implement `inOrder`, `preOrder`, and `postOrder` traversals.

Recursive implementation of preorder traversal:

```
private void preOrder(Node node){
  if (node == null) return;
  System.out.print(node.key + " ");
  preOrder(node.left);
  preOrder(node.right);
}

public void preOrder(){
  preOrder(root);
}
```

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.

   One way to implement this is to perform an inorder traversal and then check that the resulting output is a sorted sequence. A more efficient solution is to check the property at the root and then recursively at the left and right subtrees, keeping track of the maximum and minimum values allowed in each subtree.

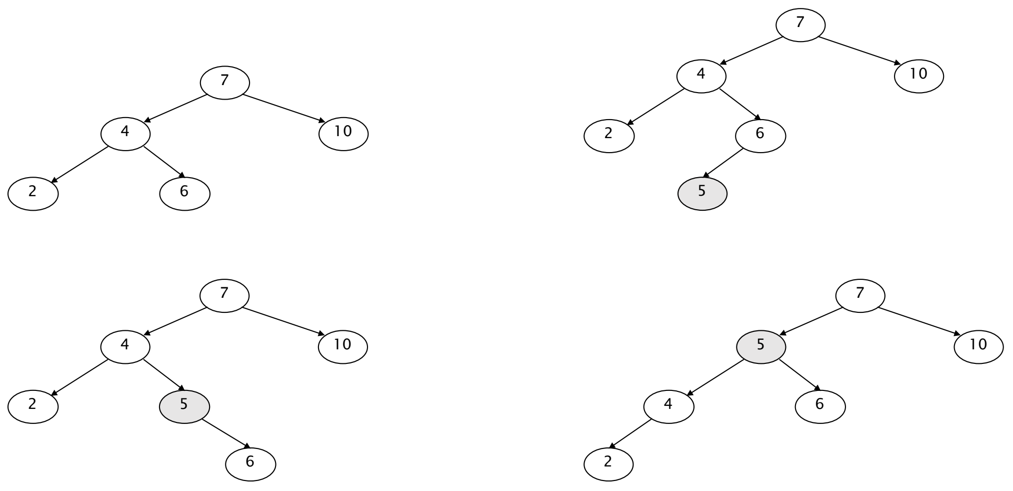
```
public boolean checkBST(){
  return checkBST(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

private boolean checkBST(Node x, int min, int max) {
  if (x == null) return true;
  if (x.key < min) return false;
  if (x.key > max) return false;
  return checkBST(x.left, min, x.key) && checkBST(x.right, x.key, max);
}
```

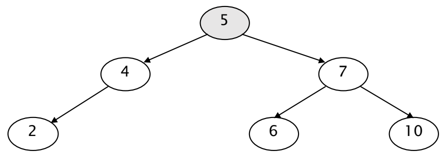
Part 2

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.

1. 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.

   This can be implemented by first inserting a new node z as usual as a leaf and then traversing the path upwards while performing a rotation at each step to bring z up: if z is a left child, then rotate right; rotate left if z is a right child. The procedure terminates when z is the new root. The following example illustrates the steps of the algorithm when node with key 5 is inserted:





2. 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.

   This can be implemented by first deleting z if it is successfully retrieved by normal search and then by calling `insertRoot(z)`. This method is used when the assumption that recently retrieved keys are likely going to be retrieved again holds.

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.

When a duplicated is inserted just increase the count and the size by one. To delete a node, decrease the counter and the size by one and then remove the node only if its count becomes 0.

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?

Implementation is straightforward. The complexity is O(n) as the recurrence equation is

$$T(n) = 2T(n/2) + c$$