Data Structures – BST and Red-Black Tree

Lab

Tasks for this week

- Implement Binary Search Tree (**insert** and **delete** methods)
- Implement Red-Black Tree (**rotateRight** and part of the **insert** method)
- Implement the method **lowest common ancestor (BST)**

- The code structure/skeleton is available on Wattle
- This lab contains assessable items!

- **Submission Guidelines**
  - The last slide contains information about the submission
  - Read it carefully to avoid losing marks!

Task 1 – Binary Search Tree (0.5 mark)

The main objective of this part is to understand how to **insert** and delete **nodes** in a Binary Search Tree (BST). The partial **code is available on Wattle/Repo.**

Step 1) Go through and read the file **`BinaryTree.java`**. It is just a simple abstract class for a BST.

*It is extremely important to understand this abstract class. It will help you to implement the **insert** and **delete** methods more quickly.*

We use a **recursive definition of binary search tree in this implementation**. A tree consist of a root node, left sub-tree (which is again a tree), and right sub-tree (which is again a tree).

```java
public abstract class BinaryTree <T extends Comparable<T>> {

    public abstract BinaryTree<T> insert(T d); // add an element to the tree, this returns the new/modified tree

    public abstract int size(); // the number of element in the tree

    public abstract int height(); // the height of the tree

    public abstract String preOrderShow(); // show the tree

    public abstract String treeshow(); // print the tree using an ascii drawing

    public abstract boolean isEmpty(); // check if the tree is empty

    public abstract BinaryTree<T> delete(T d); // remove an element from the tree, this return the new/modifited tree

    public abstract T biggest(); // find the biggest element in the tree

    public abstract T smallest(); // find the smallest element in the tree

    public abstract boolean find(T d); // check if the element is in the tree
}
```

Task 1 – Binary Search Tree

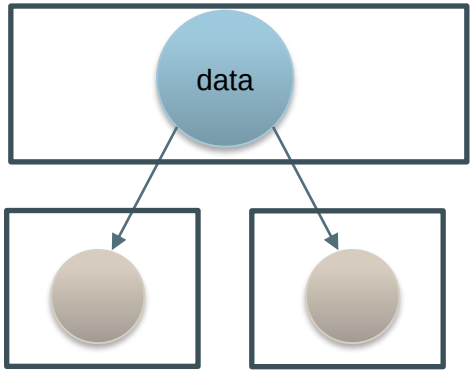
Step 2) Go through the implementation of **EmptyBinaryTree.java** and **NonEmptyBinaryTree.java** which inherit the **BinaryTree** class. Let's take a look at the insert method of the **EmptyBinaryTree.java**.

```
public class EmptyBinaryTree <T extends Comparable<T>> extends BinaryTree <T>  {

  public BinaryTree<T> insert(T data) {
      return new NonEmptyBinaryTree<T>(data);
  }
```

Note that the insert method implemented by the **EmptyBinaryTree** returns a subtree with the **root** node (instance of **NonEmptyBinaryTree**) and a **left** and **right** empty subtree (instance of **EmptyBinaryTree**).

Root node - Instance of NonEmptyBinaryTree



left, right subtrees - Instances of EmptyBinaryTree

Task 1 – Binary Search Tree

Note that by definition the method **isEmpty()** implemented in **`EmptyBinaryTree.java`** and **`NonEmptyBinaryTree.java`** return **true** and **false**, respectively.

`EmptyBinaryTree class`

```
@Override
public boolean isEmpty() {
    return true;
}
```

`NonEmptyBinaryTree class`

```
@Override
public boolean isEmpty() {
    return false;
}
```

Check all the methods now! Take time to explore the code and the differences between them. It can be useful.

Task 1 – Binary Search Tree

Step 3) Go to **`NonEmptyBinaryTree.java`** and note that some of the methods are not fully implemented. Your task is to implement them!

Implement your code within the block indicated by comments: 'YOUR CODE STARTS HERE' and 'YOUR CODE ENDS HERE' for each task.

***Important**: Do not change anything else in the source code (you may get zero marks).*

```
/**
 * Insert a new node whose value is d into the existing tree.
 * This function should return the binary tree with d inserted.
 * If the tree has already a node with d, do not create a new node
 * and return the original tree.
 *
 * Hint: You can implement insert function recursively.
 * (Each subtree (left or right) is a tree which has insert function)
 *
 * @param d data of the new node
 * @return BinaryTree<T>
 */
public BinaryTree<T> insert(T d) {
    // TODO: Add your implementation here
    // ########## YOUR CODE STARTS HERE ##########


    // ########## YOUR CODE ENDS HERE ##########
}
```

Task 1 – Binary Search Tree

It's time to code!

- Implement the following methods in the **`NonEmptyBinaryTree.java`**:

1. Method **insert()**

2. Method **delete()**

   if the target node, which we want to delete, has two children, replace the target node with its successor in its descendant (see lecture slides for details).

How to test your code?

Use the **`BinaryTreeTest.java`** file to check whether your implementation passes the test cases or not. It may be used as an indicator that your code is working correctly.

Please be aware that we will use **additional test cases** to verify and assess your code.

Task 1 – Binary Search Tree

How to test your code? (continue)

The `BinaryTreeTest.java` file has 6 test cases, check each of them.

To assess your code, we will use 10 different test cases, each worth 10%.

Again: check the submission guidelines to avoid losing marks (you may get zero marks if you do not follow it)!

Task 2 – Red-Black Tree(0.5 mark)

Red-Black Tree (R-B Tree) is a special Binary Search Tree (BST). It has all the properties of BST and some extras. For each node of the R-B Tree, it stores a color (red or black). Here are some properties of R-B Trees:

- Each node should be either black or red.
- The root node should be black.
- The leaf nodes (null) should be black.
- For each red node, its children nodes must be black.
- Each node should have the same number of black nodes from itself to all its children.

Task 2 – Red-Black Tree

The main objective of task 2 is:

Implement the **insert** part of the red-black tree and the **rotateRight**() method.

The code for this part is available on Wattle/Repo.

Read the code in **`RedBlackTree.java`**, implement your **insert**() method and **try your code with the `RBTreeTest.java`**.

Note that to complete **insert**() method, you also need to implement **rotateRight**() correctly.

Task 2 – Red-Black Tree

Code structure for the Red-Black tree:

```
public class RBTree<T extends Comparable<T>> {

    Node<T> root; // The root node of the tree

    /**
     * Initialize empty RBTree
     */
    public RBTree() {
        root = null;
    }
```

The RBTree class contains all methods to manipulate your tree.

A RB tree is always initialised as null (that is, an empty tree).

Go through the file RBTree.java and implement the missing parts.

Task 2 – Red-Black Tree

```
public enum Colour {
    RED, BLACK;
}
```

Code structure for the Red-Black tree:

Each node is composed of:

- *Colour (red or black)*
  a new node is always inserted as red – property 3
- *The node value*
- *An auxiliary pointer/node to the parent node*
  (it makes our implementation easier)
- *The children nodes (left and right)*

Read the constructor code.

*Note that **leaf nodes** are always **null** and **black** (check lecture slides)*

```
/**
 * Base class for node
 *
 * @param <T> data type
 */
public class Node<T> {

    Colour colour;          // Node colour
    T value;                // Node value
    Node<T> parent;         // Parent node
    Node<T> left, right;    // Children nodes

    public Node(T value) {
        this.value  = value;
        this.colour = Colour.RED; //property 3 (if a node is red, bot

        this.parent = null;

        // Initialise children leaf nodes
        this.left           = new Node<T>();  //leaf node
        this.right          = new Node<T>();  //leaf node
        this.left.parent    = this; //reference to parent
        this.right.parent   = this; //reference to parent
    }

    // Leaf node
    public Node() {
        this.value  = null; //leaf nodes are null
        this.colour = Colour.BLACK; //leaf nodes are always black
    }
}
```

Task 2 – Red-Black Tree

How to test your code?

The `RBTreeTest.java` file has 13 test cases, check each of them.

To assess your code, we will use 17 different test cases, each worth 0.5/17. We will test all Red-Black tree properties as well as the insertion method (rotating right and left).

Again: check the submission guidelines to avoid losing marks (you may get zero marks)!

Task 3 – BST Lowest Common Ancestor (1 mark)

Note that the partial code provided for this task is slightly different (and simpler) from the Task 1 code. Understand the code is part of this task, read it carefully and implement the required method.

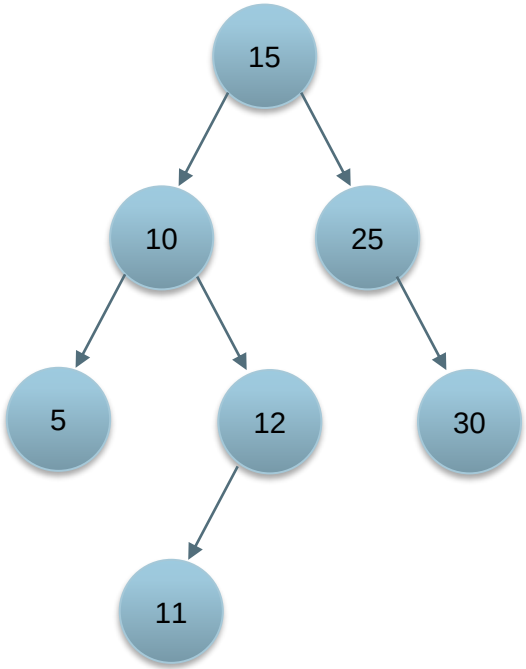
Given a binary search tree and two keys, **implement a method to find the key of the lowest common ancestor** (i.e. the shared ancestor that is located farthest from the root). You can define additional methods of BST and Node classes to complete the task. To simplify, consider that x and y always exist in the tree.

The method signature is:

public Integer **lowestCommonAncestor**(int x, int y)

Wikipedia: The lowest common ancestor (LCA) of two nodes **v** and **w** in a tree **T** is the lowest (i.e. deepest) node that has both **v** and **w** as descendants, where we define **each node to be a descendant of itself** (so if **v** has a direct connection from **w**, **w** is the lowest common ancestor).

Task 3 – BST Lowest Common Ancestor (1 mark)



Examples:

The lowest common ancestor for:

x = 10, y = 25 is the root node 15

The lowest common ancestor for:

x = 5, y = 12 is the node 10

The ancestors for x=5 and y=12 are: 10 and 15, but the lowest common ancestor is 10.

The lowest common ancestor for:

x = 5, y = 11 is the node 10

The ancestors for x=5 are: 10 and 15, and the ancestors for 11 are: 12, 10, 15. The lowest common ancestor is the node 10.

The lowest common ancestor for:

x = 25, y = 30 is the node 25 (check previous slide)

Task 3 – BST Lowest Common Ancestor

How to test your code?

The **`BSTTest.java`** file has 7 test cases, check each of them.

To assess your code, we will use 10 different test cases, each worth 1/10.

Again: check the submission guidelines to avoid losing marks (you may get zero marks)!

Submission Guidelines

- **Assignment deadline: see the deadline on Wattle (always!)**
- Submission mode: via Wattle (Lab Data Structures - Trees)

Submission format (IMPORTANT):

- Upload **only** your final version of:
  `NonEmptyBinaryTree.java` (for task 1), **`RBTree.java`** (for task 2) and **`BST.java`** (for task 3) to Wattle
- Each test case must **run for at most 1000ms**, otherwise it will fail (zero marks).
- **Do not** change the file names
- **Do not** upload any other files (only the specified files are needed)
- **Do not** upload a folder (your submission should be only **three java files**).
- The answers will be marked by an automated marker.
  - **Do not** change the structure of the source code including class name, package structure, etc.
  - You are only allowed to edit the designated code segment indicated in the comments.
- **Do not** import packages outside of the standard java SE package. The list of available packages can be found here: https://docs.oracle.com/en/java/javase/12/docs/api/index.html
- Any violation of the submission format will result in zero marks
- Reference: Introduction to Algorithms (Cormen, Leiserson, Rivest, Stein) Chap. 13. (available on Wattle)