CS150 Lab 6 Report

October 20, 2017

Oanh Doan

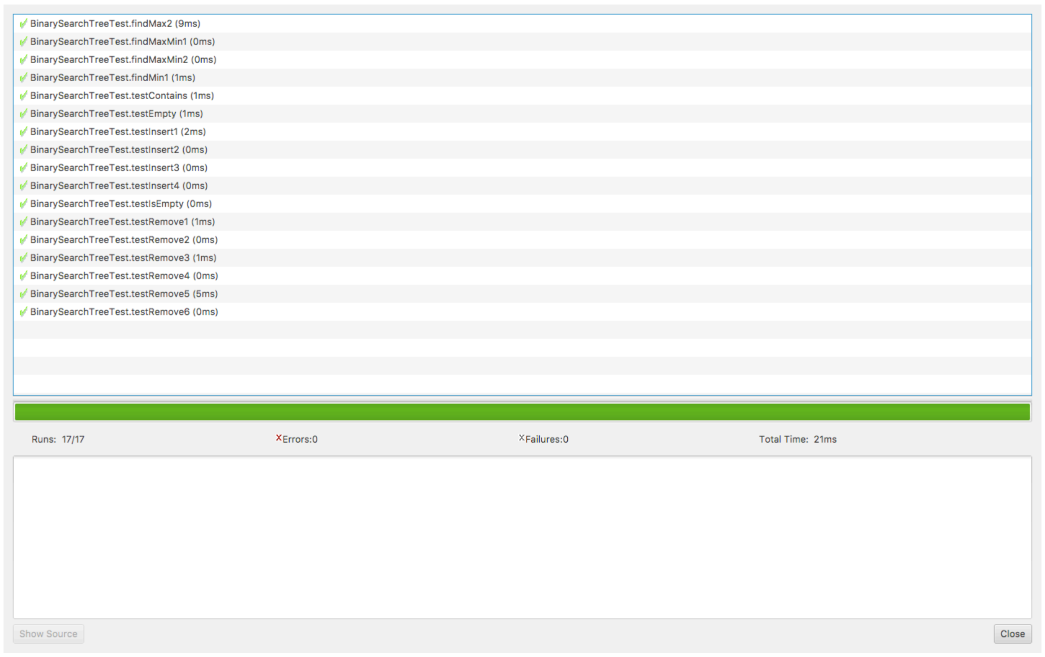
1. Introduction

The goal of this lab is to learn how to implement a binary search tree with basic methods, such as search, insert, and remove.

2. Algorithms

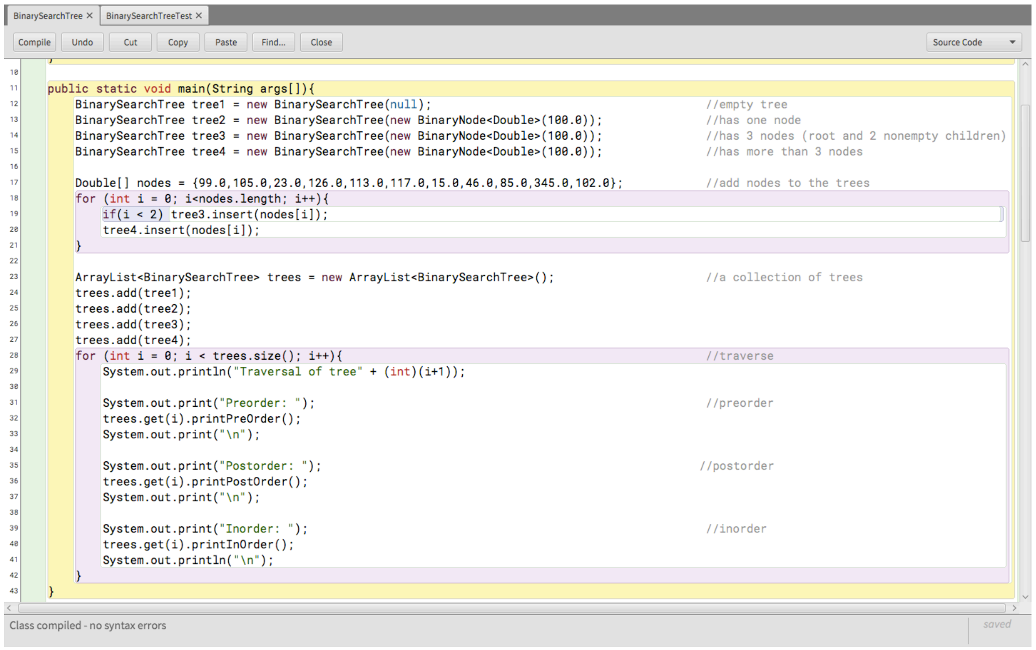
We note that in a binary search tree, all nodes in the right subtree have greater values than the root, while all the left nodes have smaller values. Each value in a binary search tree is unique.

- Search ("Tree Processing," n.d.): in each level, we either go to the left or the right, depending on the result of the comparison between searching value and root's value. We keep going down until the searching value is found or we reach the end of a path.
- Insert: The insert algorithm is very similar to search. It follows the same logic to find a path in which the new node will be inserted. The only difference is that a new node cannot be inserted along the way. It can only be inserted at the end of a path. The insertion includes creating a new node and setting the relationship between the node and its parent.
- Remove (Ghaderi, 2007): If a value is in a binary search tree, it can be removed from the tree. The deletion of the node falls into one of the three following cases:
    - The node is a leaf node: simply remove the node and its connection with its parent.
    - The node is an internal node (either with a non-empty left or right child): remove the node by setting its non-empty child to be its parent's child.
    - The node is an internal node with 2 nonempty children: replace the node's value with the smallest value in its right subtree and remove the smallest node so that there are no duplicates.
- Traversal : A process to visit all nodes in a tree and perform some action at each node. There are 3 types of traversal:
    - Preorder: parent-left-right
    - Postorder: left-right-parent
    - Inorder: left-parent-right



```
✓ BinarySearchTreeTest.findMax2 (9ms)
✓ BinarySearchTreeTest.findMaxMin1 (0ms)
✓ BinarySearchTreeTest.findMaxMin2 (0ms)
✓ BinarySearchTreeTest.findMin1 (1ms)
✓ BinarySearchTreeTest.testContains (1ms)
✓ BinarySearchTreeTest.testEmpty (1ms)
✓ BinarySearchTreeTest.testInsert1 (2ms)
✓ BinarySearchTreeTest.testInsert2 (0ms)
✓ BinarySearchTreeTest.testInsert3 (0ms)
✓ BinarySearchTreeTest.testInsert4 (0ms)
✓ BinarySearchTreeTest.testIsEmpty (0ms)
✓ BinarySearchTreeTest.testRemove1 (1ms)
✓ BinarySearchTreeTest.testRemove2 (0ms)
✓ BinarySearchTreeTest.testRemove3 (1ms)
✓ BinarySearchTreeTest.testRemove4 (0ms)
✓ BinarySearchTreeTest.testRemove5 (5ms)
✓ BinarySearchTreeTest.testRemove6 (0ms)

Runs:  17/17          ✗Errors:0          ✗Failures:0          Total Time:  21ms
```

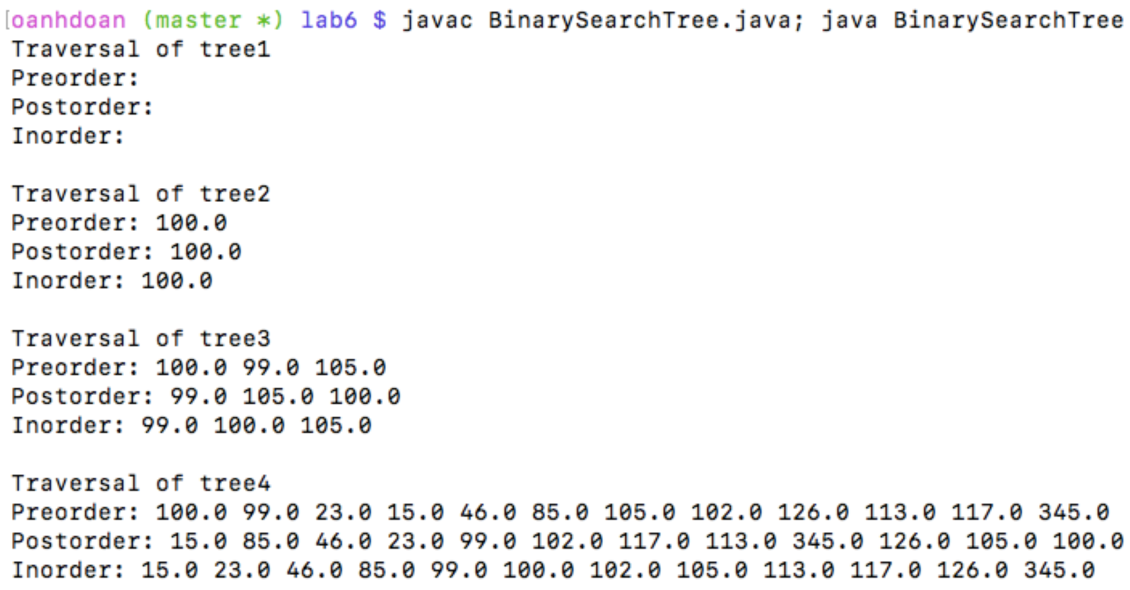
Figure 1. Results of unit tests



```java
public static void main(String args[]){
    BinarySearchTree tree1 = new BinarySearchTree(null);                              //empty tree
    BinarySearchTree tree2 = new BinarySearchTree(new BinaryNode<Double>(100.0));     //has one node
    BinarySearchTree tree3 = new BinarySearchTree(new BinaryNode<Double>(100.0));     //has 3 nodes (root and 2 nonempty children)
    BinarySearchTree tree4 = new BinarySearchTree(new BinaryNode<Double>(100.0));     //has more than 3 nodes

    Double[] nodes = {99.0,105.0,23.0,126.0,113.0,117.0,15.0,46.0,85.0,345.0,102.0};  //add nodes to the trees
    for (int i = 0; i<nodes.length; i++){
        if(i < 2) tree3.insert(nodes[i]);
        tree4.insert(nodes[i]);
    }

    ArrayList<BinarySearchTree> trees = new ArrayList<BinarySearchTree>();            //a collection of trees
    trees.add(tree1);
    trees.add(tree2);
    trees.add(tree3);
    trees.add(tree4);
    for (int i = 0; i < trees.size(); i++){                                           //traverse
        System.out.println("Traversal of tree" + (int)(i+1));

        System.out.print("Preorder: ");                                               //preorder
        trees.get(i).printPreOrder();
        System.out.print("\n");

        System.out.print("Postorder: ");                                             //postorder
        trees.get(i).printPostOrder();
        System.out.print("\n");

        System.out.print("Inorder: ");                                                //inorder
        trees.get(i).printInOrder();
        System.out.println("\n");
    }
}
```

```
Class compiled - no syntax errors
```

Figure 2. Code to traverse the tree



```
oanhdoan (master *) lab6 $ javac BinarySearchTree.java; java BinarySearchTree
Traversal of tree1
Preorder:
Postorder:
Inorder:

Traversal of tree2
Preorder: 100.0
Postorder: 100.0
Inorder: 100.0

Traversal of tree3
Preorder: 100.0 99.0 105.0
Postorder: 99.0 105.0 100.0
Inorder: 99.0 100.0 105.0

Traversal of tree4
Preorder: 100.0 99.0 23.0 15.0 46.0 85.0 105.0 102.0 126.0 113.0 117.0 345.0
Postorder: 15.0 85.0 46.0 23.0 99.0 102.0 117.0 113.0 345.0 126.0 105.0 100.0
Inorder: 15.0 23.0 46.0 85.0 99.0 100.0 102.0 105.0 113.0 117.0 126.0 345.0
```

Figure 3. Traversal

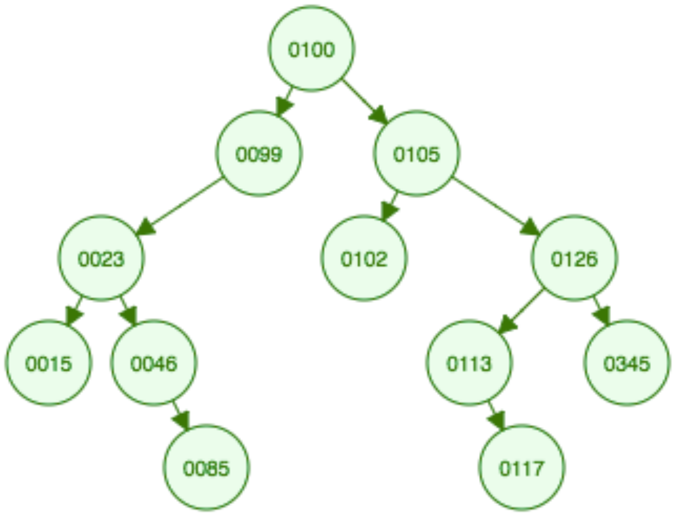


Figure 4. Tree 4 (“Binary Search Tree Visualization,” n.d.)

3. References:

Binary Search Tree Visualization. (n.d.). Retrieved October 20, 2017, from https://www.cs.usfca.edu/~galles/visualization/BST.html

Ghaderi, H. (2007). Recursive BST Operations. University of Toronto. Retrieved from http://www.cs.toronto.edu/~hojjat/148s07/lectures/week9/11bst.pdf

Tree Processing. (n.d.). Retrieved October 20, 2017, from https://www.cs.cmu.edu/~pattis/15-1XX/15-200/lectures/treeprocessing/index.html