CS 241 DATA STRUCTURES AND ALGORITHMS II
BINARY SEARCH TREE IMPLEMENTED IN JAVA

Eduardo Saenz
CS 241.01
Damavandi

Contents

A. PROJECT DESCRIPTION AND BACKGROUND

Aa. Project Description

This project requires students to write a program in Java which reads a sequence of integer values from input. Input values should be separated by a space.

This program should:

- Build a binary search tree using these values in the order they are entered.
- Print it in three traversal methods: pre-order, in-order, and post-order.
- Allow the user to Add or Remove a value. Once a new tree is generated, print it in in-order traversal.

NOTE:

- Recursion must be used to implement the add and remove methods.
- Duplicates are NOT allowed in this BST. In case of duplication, ask the user to enter values again.
- This program should have an interactive interface with the format shown (user inputs are underlined) [Fig. 1].
- Program should be tested with the given data set and our own.

TO SUBMIT:

- Java Source Code and .jar file
- A Detailed report in pdf format showing result of demo, problems encountered and solutions to them.
- Readme.txt on how to run the codeAll zipped as: yourname_CS241_Project1.zip

The purpose of this project is to get students very familiar with this new data structure, and to reinforce Java programming skills required to build one from scratch. The requirements for the project remain very surface level, but serve as building blocks for more advanced methods that can be implemented later. To build a full Binary Search Tree from the ground up is to fully understand how it works, what it does, and what it's good for.

```
% java Project1
Please enter values:
26 34 19 12 40 51 29 44 77 60 84 11 9 9 41 36 22 16 15
Duplication values are not allowed! Please enter values:
26 34 19 12 40 51 29 44 77 60 84 11 9 41 36 22 16 15
Pre-order: X X X ... X
In-order:  X X X ... X
Post-order:  X X X ... X
Main Menu
A : Add a value
R : Remove a value
E : Exit the program
What command would like to run? A
Please enter a value to add: 27
In-order:  X X X ... X
What command would like to run? A
Please enter a value to add: 11
11 already exists! Duplicated values are not allowed.
What command would like to run? R
Please enter a value to remove:12
In-order:  X X X ... X
What command would like to run? R
Please enter a value to remove: 10
10 doesn't exist!
What command would like to run? E
Exit!
%
```

Fig 1: Demo User Input from Project Description pdf

For the given data set, these are the expected traversal prints:

- Pre-Order:
  26, 19, 12, 11, 9, 16, 15, 22, 34, 29, 40, 36, 51, 44, 41, 77, 60, 84
- In-Order:
  9, 11, 12, 15, 16, 19, 22, 26, 29, 34, 36, 40, 41, 44, 51, 60, 77, 84
- Post-Order:
  9, 11, 15, 16, 12, 22, 19, 29, 36, 41, 44, 60, 84, 77, 51, 40, 34, 26
- In-Order (after adding 27):
  9, 11, 12, 15, 16, 19, 22, 26, 27, 29, 34, 36, 40, 41, 44, 51, 60, 77, 84
- In-Order (after removing 12):
  9, 11, 15, 16, 19, 22, 26, 27, 29, 34, 36, 40, 41, 44, 51, 60, 77, 84

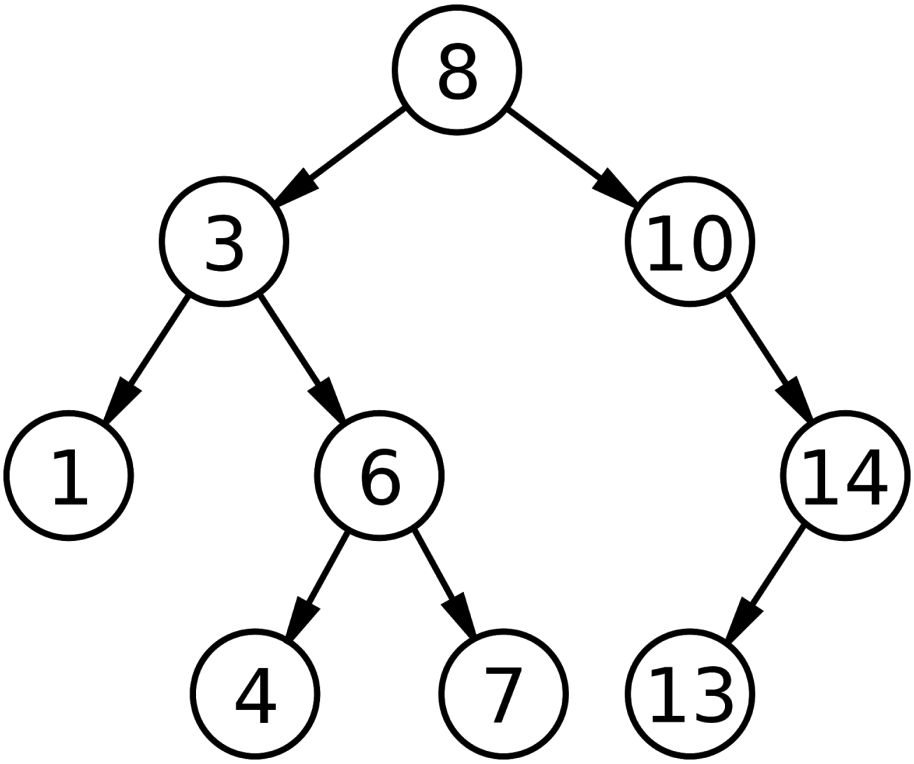

Fig. 2: A binary search tree

Ab. Binary Search Trees

Binary Search Trees (BST) are a special instance of a binary tree, used to make fast lookup, addition, and removal or values.

i) Description

In BST's, all values in the left subtree are less than the root node, and all values in the right subtree are greater. Both the left and right subtrees are also BST's. This pattern continues all the way down to the leaf nodes. BST's share the same attributes as regular Binary Trees. The BST shown has a size of 9, depth of 3, and 2 as the root. [Fig. 2]

ii) Traversal Methods

Binary Search Trees can be traversed in three ways: Pre-Order, In-Order, and Post-Order. Pre-Order traversal consists of visiting the root, then the left subtree, then the right subtree. In-Order traversal visits the numbers from least to greatest, or, left subtree, root, rightsubtree. Post-Order visits the left subtree, then the right subtree, then the root last.

This table summarized the three traversal methods of BST's, and their order for the shown binary tree [Fig 2]:

Method	Desc.	Order
Pre-Order	Left, Root, Right	2, 5, 6, 11, 7, 2, 5, 4, 9
In-Order	Root, Left, Right	2, 7, 2, 6, 5, 11, 5, 9, 4
Post-Order	Left, Right, Root	2, 5, 11, 6, 7, 4, 9, 5, 2

iii) Adding and Removing Values

BST's can have values added and removed using simple recursion algorithms. BST's have a stricter regimen for adding values, and a more streamlined way to remove values when compared to regular binary trees. Adding a value is simple: compare the value to the root, and if it is greater than the root, send it to the right subtree. If it is smaller, send it to the

Left subtree. This continues until it reaches a tree that has no child to send it to, and the value becomes that child. Removing a value is tricky, since BST's have rules the restrict which child is to replace a deleted node. If a node has two children, then it can be replaced by either the smallest number of its right subtree, or the largest number of its left subtree. If a node has one child, then it would be replaced by it. If a node has no children, then it is simply removed.

The table below summarizes these remove methods based on a node's number of children, using examples from the shown tree [Fig 2].

Node	Children	Method	Result
1	0	Self destruct	The Node is removed from the tree.
14	1	Replacement	The Node is replaced by its only child (13)
6	2	Replacement*	The Node is replaced by its successor (7 or 4), depending on replacement method.

B. DEVELOPMENT AND IMPLEMENTATION

Ba.Classes

There were three major classes that were required to complete this project. `Node.java` and `BinaryTree.java` both worked to create a fully functioning Binary Search Tree. `UI.java` created a

i) Node

This class serves as the base building block for the BST. Each Node represents a container that holds the value of a tree node, a reference to the Node's left child, and a reference to the Node's right child. These values are stored in the variables `data`, `leftChild`, and `rightChild`, respectively.

There are two constructors for this class. The main constructor takes in three parameters, an integer then two Nodes. These parameters are then set to be the Node's `data`, `leftChild`, and `rightChild`, respectively. This effectively creates a new Node with a specified value and two specified children. The secondary constructor only takes one parameter, an integer. This constructor calls the first, using the given integer as the first parameter, and `null` for the Node parameters, effectively creating a Node with the given value, but no children.

As the project went on, it was found that this first constructor was not needed. All tree building within this project's scope did not need to have a Node with children specified at the time of creation.

Below is the entire class source code from `Node.java`.

```
public class Node {
       public int data;
       public Node leftChild;
       public Node rightChild;

       Node(int data) {
          this(data, null, null);
       }

       Node(int data, Node leftChild,
Node rightChild) {
              this.data = data;
              this.leftChild = leftChild;
              this.rightChild =
rightChild;
       }
}
```

ii) Binary Tree

This class is where the BST is created, modified, and printed. Three class variables exist, `empty`, `root`, and `numNodes`. `root` is a reference to the root Node of the entire tree. `empty` is a boolean value that, if true, signifies that the tree is empty and has no root. `numNodes` keeps track of the number of Nodes that the tree has. `empty` and `numNodes` were deemed necessary by the time the project was completed.

Two constructors exist, one that

takes a single integer array as a parameter. It then builds the tree using that array, by calling the add() method for each array entry. The other constructor has no parameters, and simply calls the first, but with an empty array of size one.

The add() method is in two parts, a public add() method that takes only one parameter (an integer to be added), and a recursive, private add method that takes two (an integer to be added and the root node of a tree to add it to).

The recursive add function will compare the given value to be added to given node's value. If the given value is larger, then it will call another add() method using the given value and its right child as parameters. If the given value is smaller, it does the same thing, but with its left child. When there is no child to call add() upon, the method will instead create a new child to fill that place. This method will also increment numNodes after completing all this. The non-recursive, public add method simply call the recursive function with given number and root of the tree as parameters. Duplicate values are not allowed in this BST, thus, if one is found, DuplicateValueException is thrown, and the user is prompted to add a different number.

The remove() method is similarly in two parts : a public remove() method that takes only one parameter (an integer to be removed), and a recursive, private remove method that takes two (an integer to be removed and the root node of a tree to remove it from).

The recursive function will also similarly traverse the tree, calling itself on the right child if the given value is larger than the current Node's, and calling itself to the left child if it is smaller. When the given value and the Node's value are equal however, the function will begin to determine how the Node is to be deleted.

If the Node has two children, then it is replaced by the smallest Node in it's right subtree. Then, another remove function is called to remove that small Node from the right subtree.

If the Node has one child, then it is replaced by that child. Then, another remove from the subtree rooted at the Node to be removed.

If the node has no children, then it is simply detached from its parent.

A minVal() method was added to assist with the remove() method. This function finds the smallest value of a tree rooted at a given Node. It begins by checking for a left child, if it exists, it returns the minVal() of that child, otherwise, it returns itself. As a result, the function will return the leftmost node.

Finally, there are three printing functions, preOrder(), postOrder(), and inOrder(). Each has a public method and a private, recursive method. Like other functions, the private method uses a given Node while the public method uses the root of the tree. Each method recursively traverses the tree, and by changing the order at which it visits, the left child, visits the right child, and prints itself (the root) each method can be altered to print the tree in the three traversal methods.

iii) UI

This class serves at the User Interface, and naturally, the point of interaction between the user and the program. It has three values, a integer page, which tracks what operation the User Interface should currently be doing; A Scanner sc, which is used to take in user input, and a variable tree used to store a BinaryTree object. Each new UI is begins at a page value of 0 and a new Scanner object.

The method in which all UI operations run is the menuEngine(). This method dictates which of three main functions should be running in the UI: initialize(), printAll(), or mainMenu().

A continuous while loop is run, which contains a switch statement based upon the value of page. When page is 0, tree values are asked for and a new tree is built in initialize(), then page is incremented. When page is 1, the three traversals are printed to the screen, then page is incremented. When page is 2, the main

menu commands are printed, then the mainMenu() function begins.

In mainMenu(), another continuous while loop is executed, here, the user is prompted for a command, and then a switch statement handles the user input. From here, depending on the input, the function can call the add(), remove(), or help() functions, or exit the program.

The add and remove functions are mostly the same; each function prompts the user for a number to add or remove, then the corresponding function is called for the tree. Afterwards, the tree is printed in-order and the UI is returned to the main menu. The help() function will print some information about the program, as well as the main menu commands again.

Bb. Exceptions

i) DuplicateValueException

This exception was created to be thrown in the event that a the program attempts to add a duplicate value to the BST. This is generally going to be thrown during initialization of the three, or when the user adds another value to the tree post-initialization.

ii) ValueNotFoundException

This exception was created to be thrown in the event that the program attempts to remove a value from the BST that does not exist. This will generally be thrown when the user prompts the program to remove a value.

Bc. Major Problems Encountered

i) User Interface

Creating a User Interface was not new to me, since I had created a small text-based game for a previous class. The same general idea was used to make this UI, but I had debated whether the initialization() method belonged here. My idea of this class was to mainly just have sysout prints, with minimal calculation and real code involved. I also didn't think adding a method to BinaryTree like `addBulk(int[] numbers)` would be right, either. I ended up just letting the UI do the processing under the pretense that the UI was not doing real processing, just processing of the user input.

ii) Remove Method

The remove() method from the BinaryTree class game the most trouble. I initially thought giving setting the parameter given to null would delete a Node. This turned out obviously false, since these variables are not the objects themselves, but just a copy of them. I then thought about giving Node's a parental reference, but the solution would take a while to implement and seemed like more of a hack than I wanted it to be. I then decided to have the function actually *return* a Node. This Node would be a rebuilt copy of itself and its subtrees, but without the given value to be removed. Thus, I can assign the root of the tree to be equal to the result of this new remove() function. This ended up looking the same as my initial draft, this time the function just made a copy of the tree, edited it to remove the given value, then returned it. This solution ended up more elegant than the initial draft too, so it worked out.

C. FINAL THOUGHTS

This project was challenging to get off the ground, but was fun to put together and see working. I feel like the implementation was shallow, however. I would have liked to see requirements that demonstrated the practicality of creating a BST, or creating functions that built off our functions to work. This might be too advanced with our current knowledge, however.