University of Illinois @ Urbana-Champaign

CI 487: Data Structures for CS Teachers

Implementation #4: Implementing a Generic Binary Search Tree

1 Objectives and Overview

The objectives of this lab are as follows:

- Implement a generic Binary Search Tree class
- Gain familiarity with the recursive methods of creating, accessing, and modifying a binary search tree.

This document is organized into two sections. Section 2.1 describes the implementation of the individual elements that will make up our tree, the `TreeNode<E>` class. This class will be similar in spirit to the `ListNode<E>` class we implemented when constructing a linked list. Next, Section 2.2 will describe the implementation of the `BinarySearchTree<E>` class which is responsible for defining the construction of an access to a Binary Search Tree. You are provided the `BinarySearchTree.java` file which contains the wrappers and their respective methods stubs.

2 Step 0: Understanding what you're given

2.1 TreeNode Class

The `TreeNode<E>` class should be declared as an *inner class* with respect to the `BinarySearchTree<E>` class. This design choice is made for the same reasons as when we declared `ListNode<E>` class as an inner class in the linked list assignment. Review that assignment document regarding this topic for further information.

2.1.1 Attributes

The list node has three attributes:

1. A generic variable for holding that node's data.
2. A reference to a `TreeNode<E>` on the left.
3. A reference to a `TreeNode<E>` on the right.

It is acceptable to leave each of variables at the default level of access.

2.1.2 Constructor

This class has only a single constructor that initializes the data associated with the node to the data passed in via the constructors parameter. Additionally, it sets the `left` and `right` attributes to null.

2.2 BinarySearchTree Class - Structure and Specs

Since the `TreeNode<E>` is declared as an inner class to the `BinarySearchTree<E>` class this assignment will only involve one class file.

2.2.1 Attributes

This class should have two attributes:

1. `private TreeNode<E> root`: This variable contains the root of the whole tree. Much like the `head` attributes allowed access to the front of the linked list, the `root` allows us access to the top of the tree. It is from this point that we will start all of our traversals.
2. `private int size`: An integer containing the current number of nodes in the tree. **This attribute should be incremented and decremented every time a node is added to removed, respectively.**

In keeping with the design principle of encapsulation be sure to declare these variables with the access level of `private`. We never want the user of this class to have to directly intercut with either the root or the size. Rather, we will define methods that allow this to be abstracted away.

2.3 Constructors

This class has a single constructor of the following form: `public BinarySearchTree(){ ... }`: This is the basic constructor which initializes the `root` of the tree to `null` and `size` to 0.
We will begin with Section 3.1 which will detail how to add a node to a BST. This should be implemented first since we will need to add some nodes to the BST in order to perform all other methods. Then Section 3.2 should be implemented as this provides a method of printing the contents of the tree. Section 4.1 should then be implemented as the `findMin` method detailed in that section is crucial to implementing the final method you will implement in Section 5.1, the remove method. Lastly, you will end by adding a getter method for the size attribute.

3 Step 1: Adding and Traversing

We will begin the BST by implementing the methods for adding and traversing the tree. The purpose of this is we will need to both add nodes to the tree and have method of outputting the contents of tree to test future use of the removal method.

3.1 Adding a Node

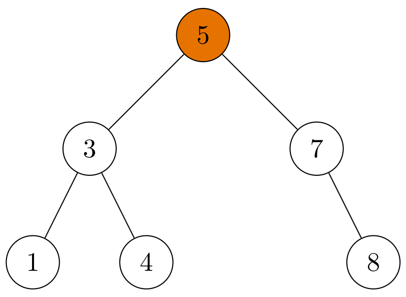


Figure 1: Current Node is 5 which is less than 6 so we proceed right

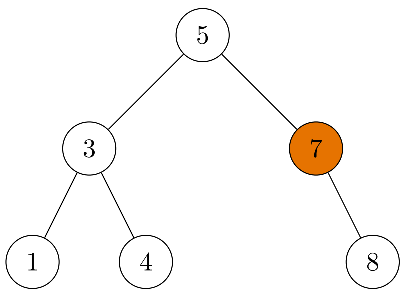


Figure 2: Current node is 7 which is less than 6. The left pointer is null so we can stop traversing and insert here.

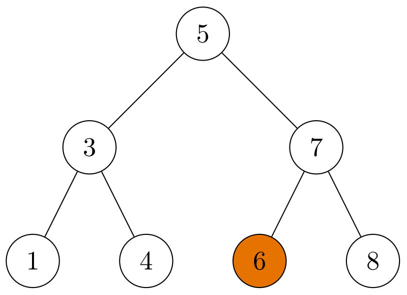


Figure 3: The final tree.

Adding a node is very similar to search in that you will use the BST property to "search" for the first available (e.g, `null`) spot in the tree. This process, as displayed in Figure 1-3, involves the following steps:

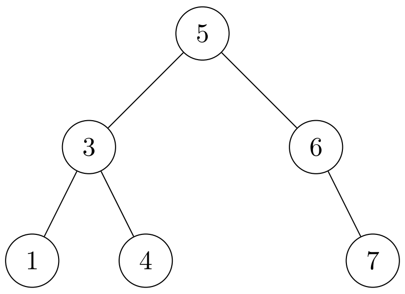
1. As we are traversing if the data associated with the current node is greater than the node we are attempting to insert and the left pointer is null then insert as that node's left child; otherwise, proceed left.
2. As we are traversing if the data associated with the current node is less than the node we are attempting to insert and the right pointer is null then insert as that node's left child; otherwise, proceed right.
3. As we are traversing if the data associated with the current node is equal to the one we are attempting to insert then we can stop traversing as we do not need to insert the node.

Your Task: Implement a method that takes a generic parameter `data`, instantiates a new `TreeNode` and inserts it into the tree according to BST insertion rules described above. One of the following method signatures should be used depending on whether you choose to implement this iteratively or recursively.

- **Iterative:** `public void add(T data){ ... }`
- **Recursive:** `public void add(TreeNode<T> curr, T data){ ... }`

3.2 Order Traversals

As a review, here are examples of the three types of tree traversals along with pseudocode for printing all of the nodes in the binary tree:



Algorithm 1: Inorder

```
Function Inorder(curr)
    if curr is null then
        return
    end
    print(node)
    Inorder(node.left)
    Inorder(node.right)
return
```

Inorder: 1 3 4 5 6 8

Algorithm 2: Preorder

```
Function Preorder(curr)
    if curr is null then
        return
    end
    Preorder(node.left)
    print(node)
    Preorder(node.right)
return
```

Preorder: 5 3 1 4 6 7

Algorithm 3: Postorder

```
Function Postorder(curr)
    if curr is null then
        return
    end
    Postorder(node.left)
    Postorder(node.right)
    print(node)
return
```

Postorder: 1 4 3 7 6 5

Printing is relatively straight forward given the ease with which you can recursively traverse and print nodes from a tree. Your task will be to modify this approach such that your methods return a list of references to the nodes in the order of the respective method's traversal. Though this can be done either iteratively or recursively we will be requiring you to form an iterative solution for this set of methods. Consider the details on the differences and similarities between recursive and iterative traversals of trees at the beginning of the lab to form this transformation (Section **??**).

Your Task: Implement the following methods that use the aforementioned traversals and return an `List` of references the instances of `TreeNode` in the tree.

1. `public List<TreeNode> getInorderTraversal(){ ... }`
2. `public List<TreeNode> getPostorderTraversal(){ ... }`
3. `public List<TreeNode> getPreorderTraversal(){ ... }`

4 Step 2: Implementing Search

4.1 Search

As the name suggests one of the primary methods of a Binary *Search* Tree is to provide the ability to *search* for an access a specific node in the tree. Searching will also form the basis for adding and removing nodes. In a BST you should keep in mind the following rules:

1. For a given root node, the left node is less than the root and the right node is greater than the root.
2. The leftmost node in a tree contains the smallest value.
3. The rightmost node in a tree contains the greatest value.

The first of these rules is the one with which we will concern ourselves with for performing a general search. The latter two can be used to search for the greatest node or the least node in a tree.

Your Task: Using these rules, implement the following methods that search for nodes in the BST. These methods can be implemented either recursively or iteratively. Whichever solution you choose to pursue, do take care to use the proper method signature as detailed below:

- A search method:
  - **Iterative:** `public TreeNode<E> search(E data){ ... }`
  - **Recursive:** `public TreeNode<E> search(TreeNode<E> curr, T data){ ... }`
- A method that retrieves the minimum node in a tree:
  - **Iterative:** `public TreeNode<E> getMinimum(){ ... }`
  - **Recursive:** `public TreeNode<E> getMinimum(TreeNode<E> curr){ ... }`
- A method that retrieves the maximum node in a tree:
  - **Iterative:** `public TreeNode<E> getMaximum(){ ... }`
  - **Recursive:** `public TreeNode<E> getMaximum(TreeNode<E> curr){ ... }`

5 Step 3: Implementing Remove

5.1 Removing a Node

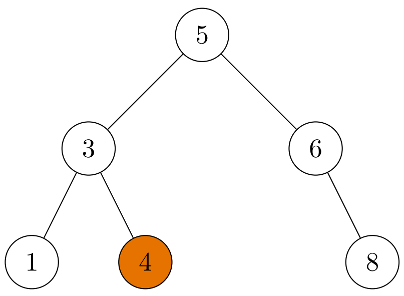


Figure 4: Leaf

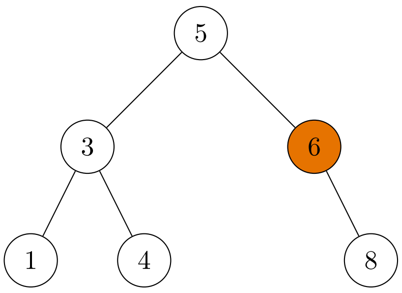


Figure 5: One Subtree

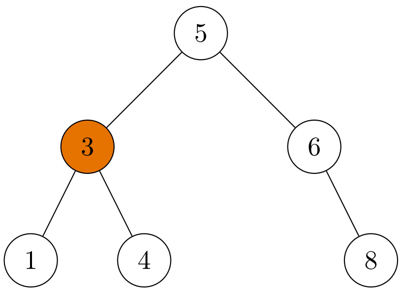


Figure 6: Two Subtrees

Removing a node once again begins with a search, however, once we have found the node we wish to remove, there are three situations we must consider:

1. The node we are removing is a leaf (Figure 7).
2. The node we are removing has one subtree(Figure 8).
3. The node we are removing has two subtrees (Figure 6).

Your Task: Your task will be to create the following method: `public void remove(??){ ... }`. The parameters of this method are up to you and the approach you take. A suggested strategy for dealing with the various removal cases enumerated above is detailed in the following sections.

Removing a Leaf

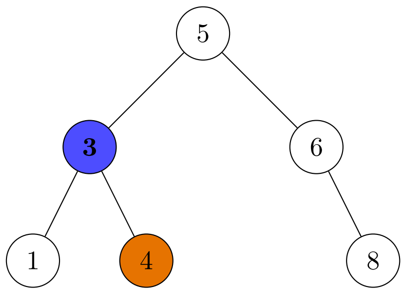


Figure 7: Find the node you want to remove (orange), in this case the one with value 4, and that node's parent (blue)

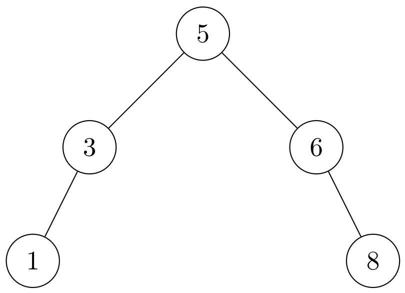


Figure 8: Set the parent's reference to that node, in this case `parent.left` equal to `null` to remove the node

The removal of a leaf is perhaps the most straightforward operation. In the event that both the right and the left points of a node are null then we only need remove the reference from that node's parent to it.

Removing a Node with One Subtree

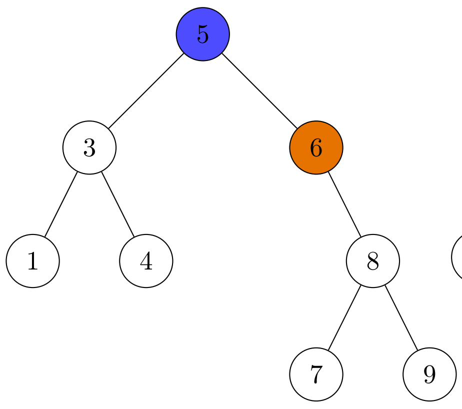


Figure 9: Find the node you want to remove (orange) and that node's parent (blue)

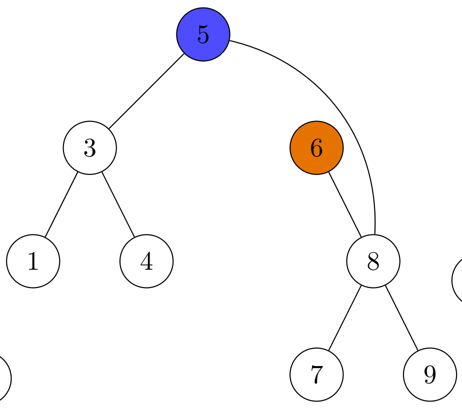


Figure 10: Set `parent.right` to the root of the node we want to remove's subtree.

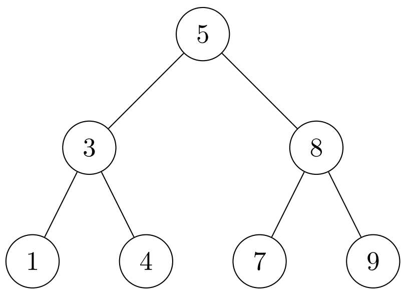


Figure 11: Set the node we want to remove's reference to it's subtree is `null`, thus removing it from the tree

The second case, where the node we want to remove has a right or left subtree, is still relatively straight forward. As with the first step we begin by finding the node we want to remove and it's parent (Figure 9). We then take the right pointer of the parent and short-circut the tree by updating it to point to root of the node we want to remove's existing subtree (Figure 10). Now, at this point the node is no longer reachable but it is still attached to the graph. In order to totally remove it we must set the reference from the node we want to remove to the root of it's subtree equal to `null` (Figure 11).

Removing a Node with Two Subtrees

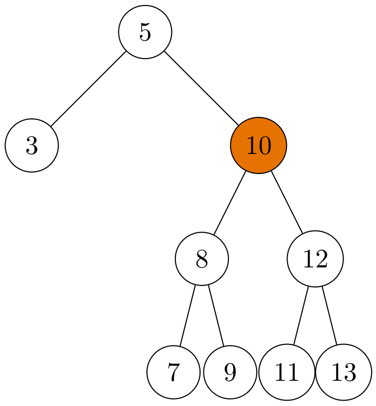


Figure 12: Find the node you want to remove (orange)

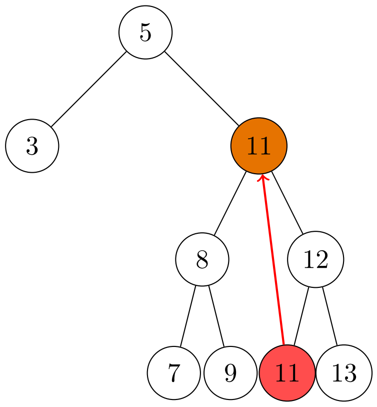


Figure 13: Find the successor node (red) and copy the successor's value to the node we want to remove.

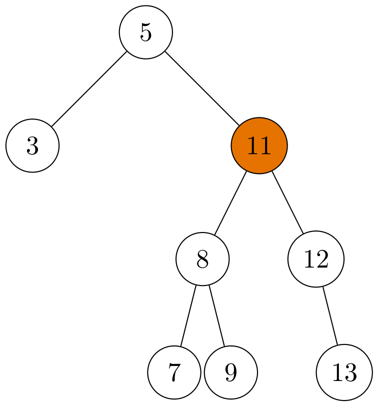


Figure 14: Call the removal method on the successor node.

The process of removing a node that has two subtrees begins with finding the node we want to remove (Figure 12). We then have to find the minimum node in the node we want to remove's right subtree; otherwise known as the node we want to remove's "inorder successor". At this point the most straightforward option to "remove" the orange node is to copy the data from the inorder successor node (Figure 14). Finally, we make a recursive call to the `remove` method to remove the successor node.

6 Final Step: Getting the size

Your Task: This class only has one attribute we are interested in and that is the variable containing the number of nodes in the tree. Create a getter using the appropriate format that returns that value.

7 Checklist

- The following accessors and tree modification methods have been implemented:
  - ☐ `add` (Section 3.1)
  - Order traversals implemented iteratively(Section 3.2)
    - ☐ `traverseInorder`
    - ☐ `traversePostorder`
    - ☐ `traversePreorder`
  - Search (Section 4.1)
    - ☐ `search`
    - ☐ `findMinNode`
  - ☐ `remove` (Section 5.1)
  - ☐ A getter method for the attribute containing the number of nodes in the tree.