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

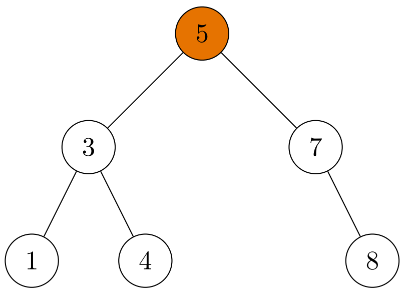


Figure 1: Current Node is 5 which is less than 6 so we proceed right

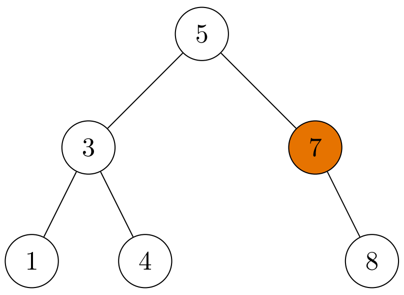


Figure 2: Current node is 7 which is less than 6. The recursive method attempts to traverse left but finds it null to instantiates and returns the node containing 6.

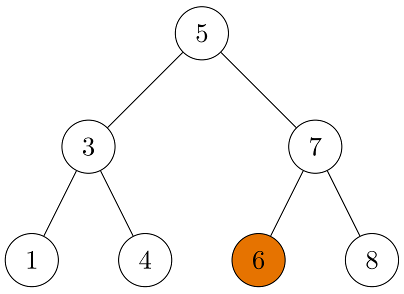


Figure 3: The recursion unwraps and this causes sevens left pointer to be updated to contain the new node returned in the last step.

As you will come to see, 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 where that node can be inserted such that the BST remains a BST. The psudeocode for this process is as follows:

```
Function Add(Data)
    Root ← Add(Root, Data);
return

Function Add(Curr)
    if Curr = NULL then
        return TreeNode(Data)
    end
    else if Data > Curr.Data then
        Curr.Right ← Add(Curr.Right, Data)
    end
    else
        Curr.Left ← Add(Curr.Left, Data)
    end
    return Curr
return
```

- **Wrapper (green):** We initially call the recursive add method on the root and set the root equal to the result of that method call. If the root is null, then it will immediately create and return a new node (Case 1) which will be set equal to the new root. Otherwise, it will begin the recursive traversal to find a place elsewhere in the tree to place the node.
- **Case 1 (red):** Curr is null so we create and return the new node. As the recursion unwraps, this node will then occupy the right or left attribute of the node which last called add (see Case 2).
- **Case 2 (blue):** Curr isn't null so we need to keep traversing according to BST rules. If the data we are inserting is greater than that of the current node then we want to proceed right by calling add on the right subtree and setting the right attribute of that node equal to the result of that method call. Otherwise, we proceed left by calling add on the left subtree and set the left attribute equal to the result of that call to add.

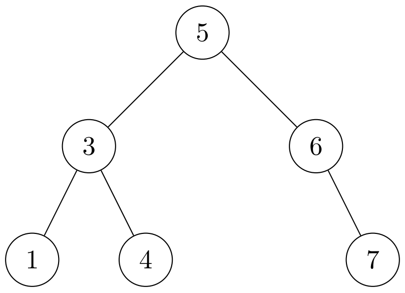
A visual example of this process is displayed in Figure 1-3. You are encouraged to draw similar examples using these rules as tracing through the algorithm manually and drawing the trees that result can be a useful method by which to break down the process.

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.

- **Wrapper** `public void add(E data){ ... }`: Since our root is private, and therefore can't be accessed by the user, we need a wrapper method that is responsible for passing in the initial root of the tree and updating it in the event it needs to updated. This method is provided and should not be modified. You are only responsible for implementing the recursive add method.
- **Recursive Add Method:** `private TreeNode<E> add(TreeNode<E> curr, E data){ ... }`: This is the recursive add method you will implement. At each recursive step you will determine if we have reached the bottom of the tree (i.e., `curr == null`) and, if so, return a new instance of a node containing `data`. Otherwise, you will determine if you need to traverse left or right by comparing the parameter `data` to the data attribute of `curr` using the `compareTo` method (see the Java docs).

3.2 Order Traversals

Below is the psudeocode for each of the tree traversal algorithms



Algorithm 1: Inorder

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

Inorder: 1 3 4 5 6 7

Algorithm 2: Preorder

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

Your Task: Implement the following methods that use the aforementioned traversals and print out the data of each instances of `TreeNode<E>` in the tree when performing said traversal.

1. `private void traverseInOrder(TreeNode<E> curr){ ... }`
2. `private void traversePostOrder(TreeNode<E> curr){ ... }`
3. `private void traversePreOrder(TreeNode<E> curr){ ... }`

Note that each of these method is private. Each method has an associated public wrapper method that passes in the initial root. These wrappers are provided and should remain unmodified.

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. Similar to adding, we will use the BST properties to traverse and find the node in the tree:

1. For a given node, the values of all the nodes in the left subtree are less than that node's value and all the nodes in the right subtree are greater than it.
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. The search method **must** be implemented recursively however the methods for getting the minimum can be implemented either recursively or iteratively.

- A search method:
  - **Public Wrapper:** `public TreeNode<E> search(E data){ ... }`: This is a provided public wrapper method that passes the initial `root` of the tree into the private, recursive search method. It should remain unmodified.
  - **Private Recursive Search:** `private TreeNode<E> search(TreeNode<E> curr, E data){ ... }`: This method should recursively search the BST and return either the node containing the value we are searching for (`data`) or null if no such node exists. When traversing, you will want to use the `compareTo` method again to compare nodes (see the Java docs).
- A method that retrieves the minimum node in a tree:
  - **Public Wrapper:** `public TreeNode<E> findMinNode(){ ... }`: This is a provided public wrapper method that passes the initial `root` of the tree into the private, recursive search method. It should remain unmodified.
  - **Private Recursive Search:** `private TreeNode<E> findMinNode(TreeNode<E> curr){ ... }`: This is the method you will be implementing. It should rind and return the minimum node in the BST (i.e., leftmost node with respect to the root). It can do this either recursively or iteratively.

5 Step 3: Implementing Remove

5.1 Removing a Node

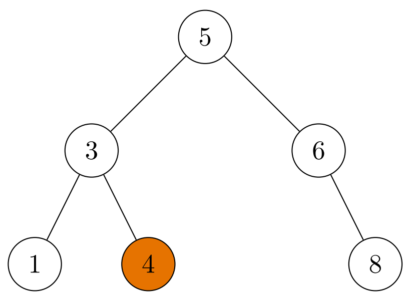


Figure 4: Leaf

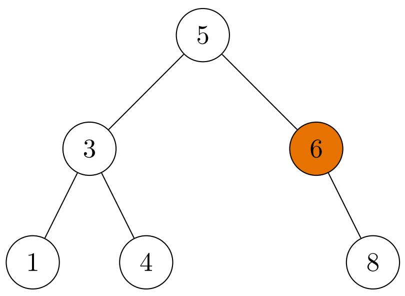


Figure 5: One Subtree

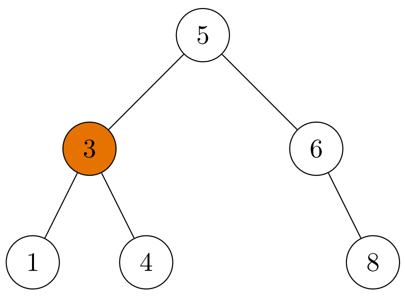


Figure 6: Two Subtrees

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).

```
Function Remove(Val)
    root = Remove(Root, Val);

Function Remove(Curr, Val)
    if curr = null then
        return null
    end
    if Curr.val < Val then
        Curr.right ← Remove(Curr.right, Val)
    end
    else if Curr.val > Val then
        Curr.left ← Remove(Curr.left, Val)
    end
    else
        if Curr.Left is NULL AND Curr.Right is NULL then
            return NULL
        end
        else if Curr.left is null then
            return Curr.right
        end
        else if Curr.right is null then
            return Curr.left
        end
        else
            MinNode ← Minimum(Curr.right)
            Curr.data ← MinNode.data
            Curr.right ← Remove(Curr.right, MinNode.data)
            return Curr
        end
    end
    return Curr
```

Your Task: Your task will be to create the following method: `private TreeNode<E> remove(TreeNode<E> curr, E data){ ... }`. This method, like the other methods, will be called via it's publicly accessible wrapper method. This method should recursively look for the node we want to remove it and remove it if such a node is found using the psudeocode to the left.

- **Step 1 (red):** Here we are recurring down the tree to search for the node we want to remove.
- **Step 2:** Remove using cases
  - **Case 1 (green):** No subtree.
  - **Case 2 (blue):** One subtree.
  - **Case 3 (purple):** Both subtrees.

A description of each situation and how the above psudeocode addresses that situation is below.

Removing a Leaf

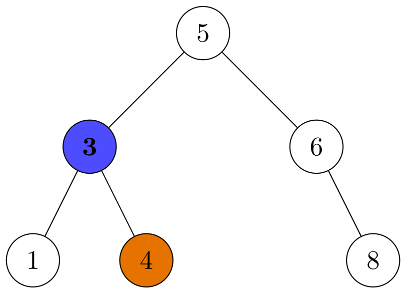


Figure 7: Find the node you want to remove (orange), in this case the one with value 4

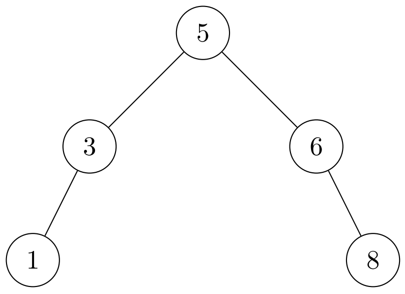


Figure 8: Return `null` so that the parent's `right.attribute` gets update to `null` as the recursion unwraps, thus removing the node

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

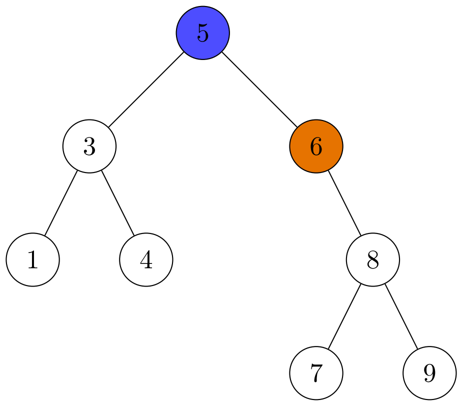


Figure 9: Find the node you want to remove (orange)

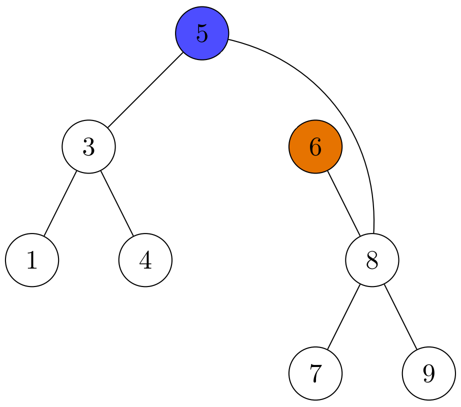


Figure 10: Return the node you found's `right` attribute so that, as the recursion unwraps, it's parents right attribute is update to be the value of the node we found's right attribute

The second case, where the node we want to remove has a right or left subtree. 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 skip over the node we are removing from the tree by updating it to point to root of the node we want to remove's existing subtree (Figure 10).

Removing a Node with Two Subtrees

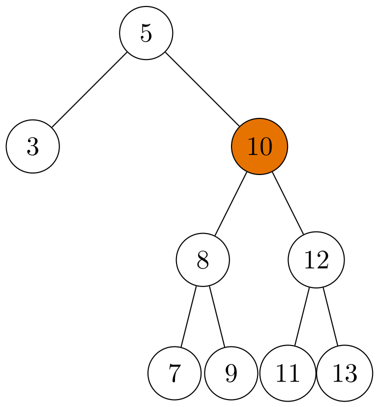


Figure 11: Find the node you want to remove (orange)

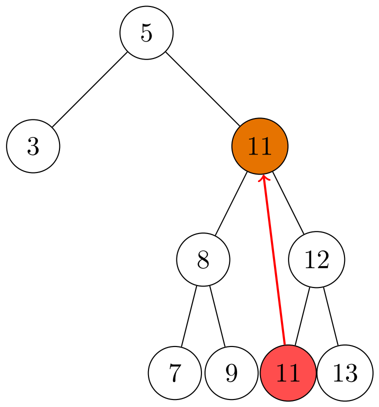


Figure 12: Find the successor node (red) and copy the successor's value to the node we want to remove.

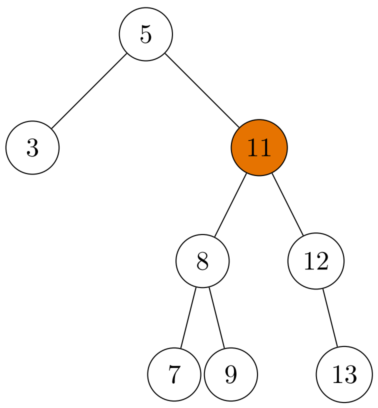


Figure 13: Call the removal method on the successor node.

The process of removing a node that has two subtrees begins with finding the node we want to remove (Figure 11). 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 13). Finally, we make a recursive call to the `remove` method to remove the successor node so that the duplicate we created is no longer in the tree.

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.