## Objective

- Create a Binary Search Tree data structure and several useful BST methods.

## Setup

1. Download the `Lab28StarterCode.zip` file from `msBrekke.com`
2. Unzip starter code in `H:/CSIII/Lab28` folder

## Testing

Note: None of the JAMTester tests will pass until the `add`, `preorder`, `postorder`, and `inorder` methods work correctly.

## Part 1: Binary Search Tree

### add Method

Write the `add` method in the `BST.java` file. This method wraps data in a `BNode` class and adds the `BNode` class to the BST. If the BST is empty, then this new `BNode` becomes the root. If the BST is not empty, then use the `addHelper` method to recursively search the BST for the correct place to add this new `BNode` to.

Follow the recursive algorithm:

- If newData < nodeData,
  - If there is already a node to the left, recursively add to the left
  - If there isn't already a node to the left, add newData to the left of this node
- If newData >= nodeData,
  - If there is already a node to the right, recursively add to the right
  - If there isn't already a node to the right, add newData to the right of this node

### addAll method

Write the `addAll` method in the `BST.java` file. This method uses the `add` method to add each value of the specified list to the BST (in order)

### Size method

Write the size method in the BST.java file. This method will use the sizeHelper method to recursively count each node in the tree. If a node is a leaf node (it has no children) then it will return 1. Otherwise, it will return:

```
1 + size of left subtree + size of right subtree
```

## preorder Method

Write the `preorder` method in the `BST.java` file. This method returns a string that contains the values stored in this BST in pre order. Each value should be separated by a comma. There should not be a trailing comma at the end of the string. The entire list of data should be wrapped in square brackets `[ ]`.

Example output:

```
[C, B, A, C, F, E, G]
```

You will use the `preorderHelper` method to recursively solve this problem. Whenever you would "print" some data, instead concatenate it to the `ret` string.

Let the `preorder` method worry about the trailing comma (Hint: use substring) and square brackets (Hint: just concatenate them).

## inorder Method

Write the `inorder` method in the `BST.java` file. This method works the same as the `preorder` method, except for the order in which the values are concatenated.

## postorder Method

Write the `postorder` method in the `BST.java` file. This method works the same as the `preorder` and `inorder` methods, except for the order in which the values are concatenated.

# Part 2: contains Method

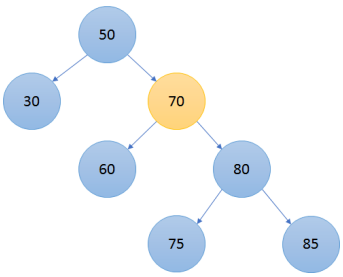
Write the `contains` method in the `BST.java` file. This method will return `true` if the BST contains the specified value, and `false` otherwise. Use the `containsHelper` method to recursively check the appropriate nodes for the existence of `data`. This method should run in O(log n) time.

## Part 3: remove Method

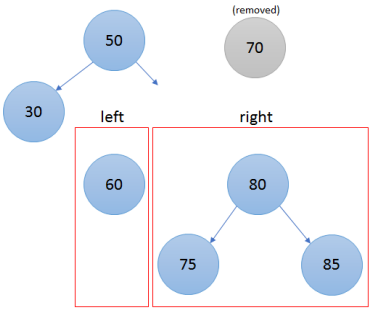
Write the `remove` method in the `BST.java` file. This method will remove the *first* occurrence of data from the BST.

Removing a node from a BST is tricky, because you have to reconnect that node's children somewhere.

Consider the following tree:




If you remove node 70, what happens to its left and right children?

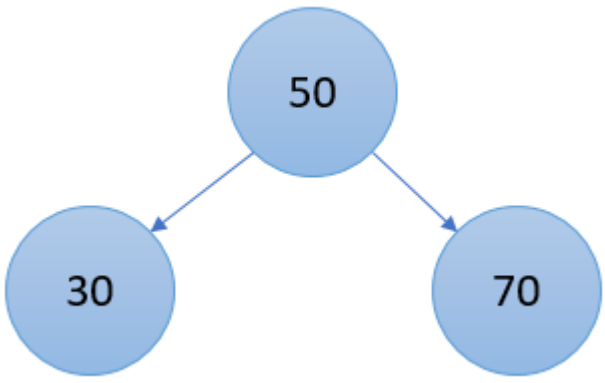



There are three situations you can find yourself in when removing a node from a BST:

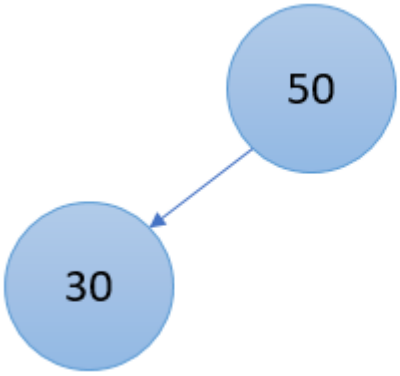
1. The removed node has 0 children
2. The removed node has 1 child
3. The removed node has 2 children

### Case 1: No children

This is the simplest case. If the node you want to remove has no children, you can just remove it (set the parent's left/right child to `null`)



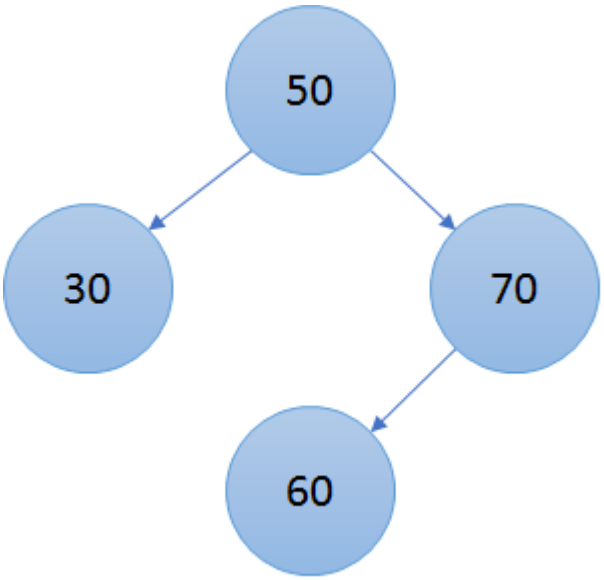

Removing 70 from this BST



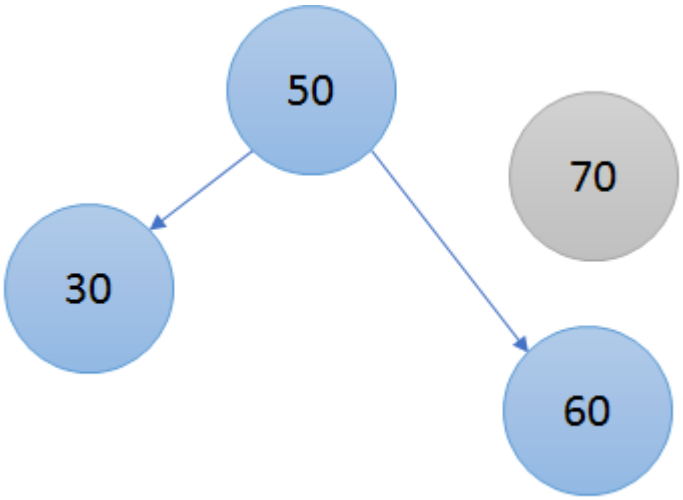

Set the right child of 50 to `null`

## Case 1: One Child

This is also a simple case. If the node you want to remove is greater than its parent, then all of its children are also greater than the parent node (ditto if the removed node is less than its parent). If this is the case, then when you remove the node, set its parent's left/right child to point to the remove node's only child.



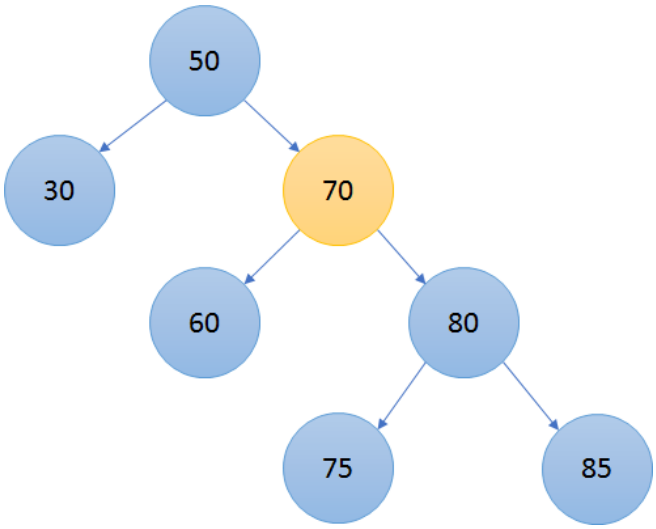

Removing 70 from BST, with one child



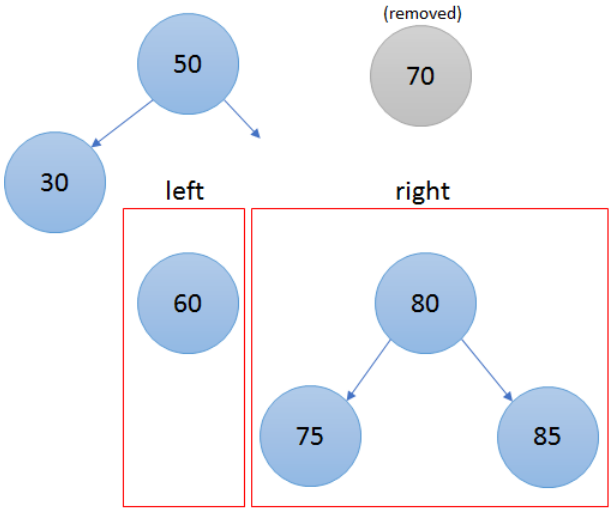

Set right child of 50 to 70's left (only) child

## Case 2: Two Children

This is the tricky case. When you remove a node with two children, you end up with two dangling subtrees.




Removing 70 from BST, with two children

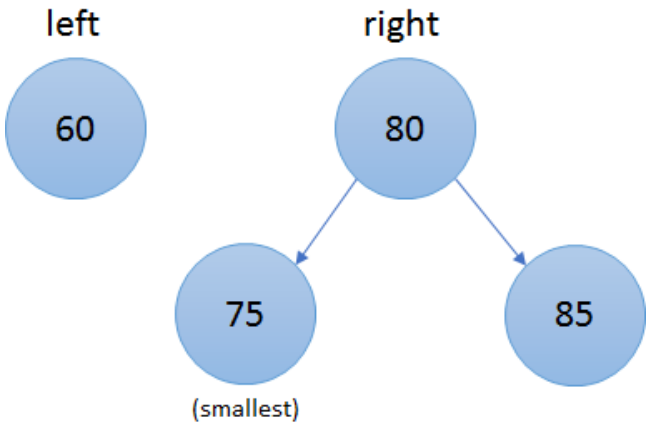



Two dangling subtrees are created when 70 is removed

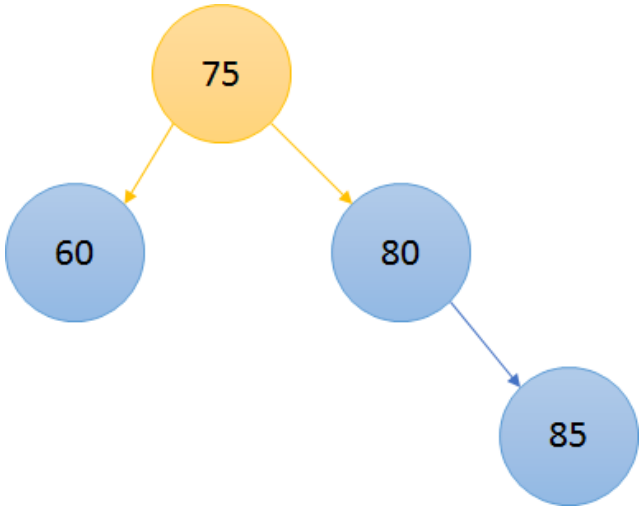
In order to reattach the dangling subtrees, you need to reform them into a single subtree and then reattach that to the original parent node.

Use the following algorithm to reform two BST's into a single BST:

1. Find the smallest value in the right subtree (this child will not have a left child), call it X
2. Detach X from its tree (set its parent's left/right child to null)
3. Add the left subtree as the left child of X
4. Add the right subtree as the right child of X

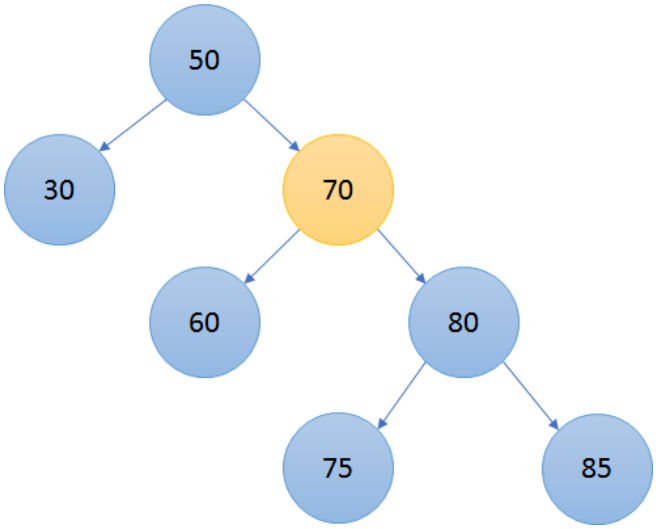



Find the smallest value in the right subtree

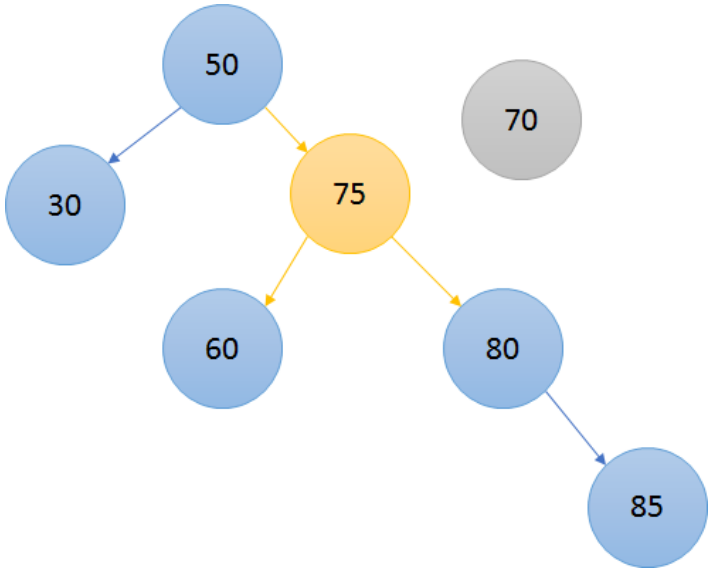



Turn the smallest node into the root node

Now you have a single BST that you can reattach to the parent of the node that is being removed!




Removing 70 from BST, with two children




Combine the children and add as right child of 50

## Remove Algorithm

1. Find the node that you want to remove (and its parent), call it X
    a. If the node does not exist, return `false`
2. Check how many children X has
    a. If 0 children
        i. remove it from its parent (set appropriate child of parent to `null`)
    b. If 1 child
        i. Set X's parent's left/right child to X's child
    c. If 2 children
        i. Combine X's children into a single BST
        ii. Set X's parent's left/right child to the combined BST
3. Return `true`

## Find Algorithm (iterative)

1. Keep track of a `node`, starting at `root`
2. Keep track of node's `parent`, starting at `null` (root doesn't have a parent)
3. While `node` is not `null` and `node`'s data is not equal to `data`
    a. Set `parent` to `node` (because you are going to change `node` to reference one of its children)
    b. If `data` is less than `node`'s data, set `node` to the left child
    c. If `data` is greater than `node`'s data, set `node` to the right child
4. If `node` is `null`, data doesn't exist in the BST
5. Otherwise
    a. `node` is the node that contains `data`
    b. `parent` is the parent of `node`

## Removing Smallest Child Algorithm

1. If `node` has no children or only has a right child
    a. Return `node`
2. If `node` has left child
    a. Recursively call `removeSmallestChild` on left child
    b. If return value of recursive call is == to this node's left child
        i. Set left child to the right child of the returned node
    c. Return recursive call's return value

## Combining Dangling Subtrees Algorithm

1. Remove the smallest node from the right subtree, call it X
    a. If the smallest node (that was just returned from the `removeSmallestChild` method) is == to `right` (the parameter) then you need to assign `right` the value of its right child.
2. Add the left subtree to the left of X
3. Add the right subtree to the right of X
4. Return X

## Grading

Use the JAMTester tests to make sure your code works properly. This lab will be tested auto-magically.

| | |
|---|---|
| Part 1 | 50 points |
| Part 2 | 25 points |
| Part 3 | 25 points |