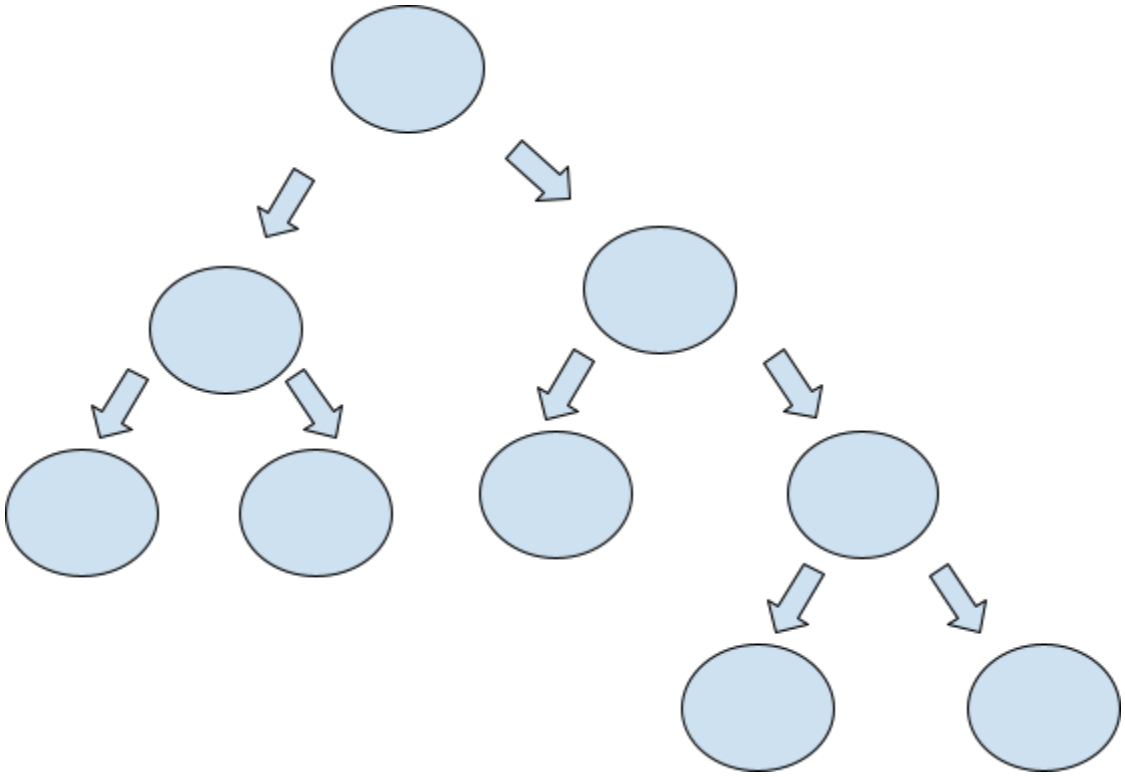
Quentin Creese
Jess
CS 260
Assignment 06
Due: February 26, 2024

Binary Search Tree

1. Before you start coding, create a design that shows how your binary tree functions and what attributes it keeps track of to function.
Design: The top of the tree should be the middle of the tree. The right of the tree should always have a higher value. The left of the tree should always have a lower value.



Attributes to keep track of to function:

- A starting location “root”
- A value called “val”
- Current value called “current”
- Call back value to help node traversal called “callback”
- Push and pop functions to remove and call nodes
- A temporary value to help rearrange nodes called “temp”
- Left and right pointers to help with node traversal called “left” and “right”.

2. Create some tests (at least one per function), before you start coding, that you want your Binary Search Tree (BST) to pass as evidence that it would be working correctly if it passed the tests

- I would like to test if my BST is empty to begin
- I would like to test that my add function is working by adding a node to my BST

- I would like to test that my remove function is working by removing a node in my BST
- I would like to test my function for pre-order traversal
- I would like to test my function for post-order traversal
- I would like to test my function for breadthFirstTraversal

3. Implement a binary search tree that includes:
    1. Nodes to store values:

```
// TreeNode struct representing a node in a binary tree
struct TreeNode {
    int val;         // Value stored in the node
    TreeNode* left; // Pointer to the left child node
    TreeNode* right; // Pointer to the right child node

    // Constructor to initialize a TreeNode with a given value
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

    2. An add function that adds a value to the appropriate location based on our ordering rules:

```
void BinarySearchTree::add(int val) {
    root = addNode(root, val); // Add a value to the binary search tree
}
```

```
// Private helper functions for the binary search tree operations
TreeNode* BinarySearchTree::addNode(TreeNode* node, int val) {
    if (node == nullptr) {
        return new TreeNode(val);
    }

    // Recursively insert the value based on the binary search tree property
    if (val <= node->val) {
        node->left = addNode(node->left, val);
    } else {
        node->right = addNode(node->right, val);
    }

    return node;
}
```

    3. A remove function that finds and removes a value and then picks an appropriate replacement node:

```cpp
void BinarySearchTree::remove(int val) {
    root = removeNode(root, val); // Remove a value from the binary search tree
}
```

```cpp
TreeNode* BinarySearchTree::removeNode(TreeNode* node, int val) {
    if (node == nullptr) {
        return nullptr;
    }

    // Recursively remove the value from the binary search tree
    if (val < node->val) {
        node->left = removeNode(node->left, val);
    } else if (val > node->val) {
        node->right = removeNode(node->right, val);
    } else {
        if (node->left == nullptr) {
            TreeNode* temp = node->right;
            delete node;
            return temp;
        } else if (node->right == nullptr) {
            TreeNode* temp = node->left;
            delete node;
            return temp;
        }

        TreeNode* temp = findMinNode(node->right);
        node->val = temp->val;
        node->right = removeNode(node->right, temp->val);
    }
    return node;
}
```

4. At least one tree traversal function:
    a. The three common traversals (pre-order, post-order, in-order):

```cpp
void BinarySearchTree::preOrder(std::function<void(TreeNode*)> callback) {
    preOrder(root, callback); // Perform pre-order traversal starting from the root
}

void BinarySearchTree::inOrder(std::function<void(TreeNode*)> callback) {
    inOrder(root, callback); // Perform in-order traversal starting from the root
}

void BinarySearchTree::postOrder(std::function<void(TreeNode*)> callback) {
    postOrder(root, callback); // Perform post-order traversal starting from the root
}
```

```cpp
// Traversal functions using recursion
void BinarySearchTree::preOrder(TreeNode* node, std::function<void(TreeNode*)> callback) {
    if (node == nullptr) return;
    callback(node); // Process the current node
    preOrder(node->left, callback); // Traverse left subtree
    preOrder(node->right, callback); // Traverse right subtree
}

void BinarySearchTree::inOrder(TreeNode* node, std::function<void(TreeNode*)> callback) {
    if (node == nullptr) return;
    inOrder(node->left, callback); // Traverse left subtree
    callback(node); // Process the current node
    inOrder(node->right, callback); // Traverse right subtree
}

void BinarySearchTree::postOrder(TreeNode* node, std::function<void(TreeNode*)> callback) {
    if (node == nullptr) return;
    postOrder(node->left, callback); // Traverse left subtree
    postOrder(node->right, callback); // Traverse right subtree
    callback(node); // Process the current node
}
```

b. A breadth-first traversal (sometimes called a level-order search):

```cpp
void BinarySearchTree::breadthFirst(std::function<void(TreeNode*)> callback) {
    if (root == NULL) return; // If the tree is empty, return immediately

    std::queue<TreeNode*> q;
    q.push(root);

    // Perform breadth-first traversal using a queue
    while (!q.empty()) {
        TreeNode* current = q.front();
        q.pop();
        callback(current); // Call the callback function for the current node

        // Enqueue left and right child nodes if they exist
        if (current->left != NULL)
            q.push(current->left);
        if (current->right != NULL)
            q.push(current->right);
    }
}
```

Main output:

```
quentin@Quentins-MacBook-Pro CS260_assignment06 % clang++ -o main main.cpp BinarySearchTree.cpp -std=c++11 -stdlib=libc++ && ./main
In-order traversal: 20 20 30 40 40 20 30 40 50 60 70 80 60 60 70 80 80
Pre-order traversal: 50 30 20 40 70 60 80 30 20 40 20 40 70 60 80 60 80
Post-order traversal: 20 40 20 40 30 60 80 60 80 70 20 40 30 60 80 70 50
Breadth-first traversal: 50 30 70 20 40 60 80
```

Test output:

```
quentin@Quentins-MacBook-Pro CS260_assignment06 % clang++ -o BST_Test BinarySearchTree.cpp BST_Test.cpp -std=c++11 -stdlib=libc++

quentin@Quentins-MacBook-Pro CS260_assignment06 % ./BST_Test
In-order traversal: 20 30 40 50 60 70 80
Pre-order traversal: 50 30 20 40 70 60 80
Post-order traversal: 20 40 30 60 80 70 50
Breadth-first traversal: 50 30 70 20 40 60 80
In-order traversal after removal: 40 50 60 70 80
quentin@Quentins-MacBook-Pro CS260_assignment06 %
```

4. Analyze and compare the complexity of insert and search as compared to a binary tree without any order in its nodes (what is the run time of an unordered list?)

The efficiency of the binary search tree (BST) we created today depends on how balanced the tree remains during insertion and removal operations. In a well-balanced BST, where nodes are inserted so the tree remains balanced, the efficiency of insertion and search operations is typically O(log n) on average, where n is the number of nodes in the tree. This is because the height of the tree remains logarithmic with respect to the number of nodes, allowing for efficient traversal and search.

However, if the BST becomes unbalanced, such as in the worst-case scenario where nodes are inserted in sorted order, the efficiency of insertion and search operations degrades to O(n), where n is the number of nodes in the tree. This worst-case scenario occurs when the BST resembles a linked list, with each node having only one child.

Therefore, to ensure efficient operations in the BST we created today, it's essential to maintain balance during insertion and removal operations.