Bishoy Pramanik

COP3530

Professor Kapoor

AVL Tree Documentation

Time Complexity Analysis:

insert(std::string name, int id):

While the first part of the *insert()* function is O(1), the final line, *insertOnNode()* will cause a recursive traversal down the tree, with a complexity of O(log(n)), since the method will go down the tree, going left or right until it has found its correct position in the tree. However, at the end of the *insertOnNode()*, the *retraceInsert()* function is called, recursively travelling back up the tree to visit all ancestors of the newly inserted node. While *retraceInsert()* calls the rotation functions, all rotation functions are O(1). Because *retraceInsert()* only touches the ancestors, the exact same nodes touched by the *insertOnNode()* function, it is also O(log(n)). Together, these make the *insert()* function O(1 + log(n) + log(n)). By reducing, we get *insert()* is O(log(n)) function, where n is the number of nodes in the AVL tree.

```
    bool insert(const std::string name, const int id)
    {
        std::shared_ptr<Node> insertion = std::make_shared<Node>(name, id);

        if (this->root == nullptr)
        {
            this->root = insertion;
            return true;
        }

        std::shared_ptr<Node> current = this->root;
        current->getID();
        return insertOnNode(current, insertion);
    }
```

```
bool insertOnNode(std::shared_ptr<Node> current, std::shared_ptr<Node> insertion)
    {

        if (insertion->getID() == current->getID())
        {
            return false;
```

```
        }
        else if (insertion->getID() > current->getID())
        {
            if (current->right)
            {
                return insertOnNode(current->right, insertion);
            }
            else
            {
                current->right = insertion;
                current->right->parent = current;

                retraceInsert(current, current->right);

                return true;
            }
        }
        else
        {
            if (current->left)
            {
                return insertOnNode(current->left, insertion);
            }
            else
            {
                current->left = insertion;
                current->left->parent = current;

                retraceInsert(current, current->left);

                return true;
            }
        }
    }
    void retraceInsert(std::shared_ptr<Node> upper, std::shared_ptr<Node> lower)
    {
        // CHeck if the last ancestor (or the root) has been updated
        if (upper == nullptr)
            return;

        // Adjust the balance factor based on which side the subtree is on
        if (lower.get() == upper->left.get())
        {
            upper->addBF(1);
        }
```

```
        else
        {
            upper->addBF(-1);
        }

        // If the BF after insertion is 0 at any node, then the height has
remained the same
        if (upper->getBF() == 0)
        {
            return;
        }

        // If the BF is +- 2, then a rotation is needed
        else if (upper->getBF() == 2)
        {
            if (lower->getBF() == 1)
            {
                rotateRight(upper, lower);
            }
            else
            {
                rotateLeftRight(upper, lower);
            }
        }
        else if (upper->getBF() == -2)
        {
            if (lower->getBF() == -1)
            {
                rotateLeft(upper, lower);
            }
            else
            {
                rotateRightLeft(upper, lower);
            }
        }

        // If the BF is +-1, then we need to keep retracing
        else if (upper->getBF() == 1 || upper->getBF() == -1)
        {
            if (upper->parent == nullptr)
                return;
            else
                return retraceInsert(upper->parent, upper);
        }
    }
```

remove(int id):

remove() traverses down the children of the tree using the properties of a BST, finding it's target node in O(log(n)) time because this is the look-up time for a BST. Once it has found the node, the rearranging of pointers is O(1). After rearrangement, *remove()* calls *retraceDelete()* which recursively travels up the tree, touching all ancestors of the removed node's parent, giving it O(log(n)) as well. Because these happen sequentially, the time complexity of *remove()* is O(log(n) + 1 + log(n)), which reduces to O(log(n)). For this reason, *remove()* is O(log(n)), where n is the number of nodes in the tree.

```
    bool remove(int id)
    {
        std::shared_ptr<Node> search = this->root;

        // Step 1: Find the node to be removed (search)

        // Short-Circuit abusers (should avoid the getID() nullptr error by
short-circuiting)
        while (search)
        {
            if (search->getID() == id)
                break;
            else if (search->getID() > id)
                search = search->left;
            else
                search = search->right;
        }

        // Step 2: Find search node's replacement

        // Check if the node actually exists
        if (search)
        {
            //** Case 1: Search is a leaf node
            if (!(search->right || search->left))
            {
                if (search.get() != this->root.get())
                {
                    if (search.get() == search->parent->right.get())
                    {
                        search->parent->right.reset();
                        search->parent->addBF(1);
```

```
                    }
                    else
                    {
                        search->parent->left.reset();
                        search->parent->addBF(-1);
                    }
                }
                else
                {
                    // If it's the root node, and only node in tree, then we can
just delete the root
                    this->root.reset();
                    return true;
                }
                // The parent should be the only pointer on the node, with
shared_ptr calling destructor
                // after this method falls out of scope

                retraceDelete(search->parent->parent, search->parent);
            }
            //** Case 2: Search has only one child
            else if (!search->left || !search->right)
            {
                // Replace this node with its one child
                // Case: Search has only a left child
                if (search->left)
                {
                    // Edge case on root node
                    if (search.get() != this->root.get())
                    {
                        // If search is the right child of its parent
                        if (search.get() == search->parent->right.get())
                        {
                            search->parent->right = search->left;
                            search->left->parent = search->parent;
                        }
                        else
                        {
                            search->parent->left = search->left;
                            search->left->parent = search->parent;
                        }
                    }
                    // Case: Search is a root node
                    else
                    {
```

```
                    this->root = search->left;
                    search->left->parent.reset();
                }

                retraceDelete(search->left->parent, search->left);
            }
            else
            // Case: Search has only a right child
            {
                if (search.get() != this->root.get())
                {
                    // If search is the right child of its parent
                    if (search.get() == search->parent->right.get())
                    {
                        search->parent->right = search->right;
                        search->right->parent = search->parent;
                    }
                    else
                    {
                        search->parent->left = search->right;
                        search->right->parent = search->parent;
                    }
                }
                else
                {
                    this->root = search->right;
                    search->right->parent.reset();
                }

                retraceDelete(search->right->parent, search->right);
            }
        }
        //** Case 3: Search has two children
        else
        {
            // We need to find the in-order successor, so we go right once,
then left as much as possible
            std::shared_ptr<Node> replacement = search->right;

            while (replacement->left)
                replacement = replacement->left;

            std::shared_ptr<Node> replacement_parent = replacement->parent;

            //**SWAP SEARCH AND REPLACEMENT
```

```
                // Manage replacement's right child (it can only have a right
child)
                if (replacement.get() == replacement->parent->right.get())
                    replacement->parent->right = replacement->right;
                else
                    replacement->parent->left = replacement->right;

                if (replacement->right)
                    replacement->right->parent = replacement->parent;

                // Setting Replacement's new parent relationship
                if (search.get() == this->root.get())
                // Case: Search is the root node
                {
                    this->root = replacement;
                    replacement->parent.reset();
                }
                else
                // Case: Search is not the root node
                {
                    replacement->parent = search->parent;

                    if (search.get() == search->parent->right.get())
                        search->parent->right = replacement;
                    else
                        search->parent->left = replacement;
                }

                // Setting Replacement's child relationship
                replacement->left = search->left;
                replacement->right = search->right;

                if (replacement->left)
                    replacement->left->parent = replacement;

                if (replacement->right)
                    replacement->right->parent = replacement;

            }

            return true;
        }
        else
            return false;
    }
```

search(int id):

```
std::string search(int id)
{
    return search(this->root, id);
}

std::string search(std::shared_ptr<Node> current, int id)
{
    if (!current)
        return "unsuccessful";
    else
    {
        if (current->getID() > id)
            return search(current->left, id);
        else if (current->getID() < id)
            return search(current->right, id);
        else
            return current->getName();
    }
}
```

The *search()* method takes advantage of the properties of BST, traversing the tree and splitting the amount of nodes it has to search for by half each time. For this reason, *search(int id)* is a O(log(n)), where n is the number of nodes in the tree.

search(std::string name):

Due to sorting by ID numbers, the name prevents us from using the properties of BST. Additionally, multiple nodes may share the same name. For this reason, all nodes in the tree must be touched. The *inOrderSearch(std::shared_ptr<Node> current, std::string name)* function goes through an in-order traversal of the AVL tree, only going through all the tree's nodes once, giving *search(std::string name)* a time complexity of O(n), where n is the number of nodes in the tree.

```
std::string inOrderSearch(std::string name)
{
    std::string printString = inOrderSearch(this->root, name);

    if (printString == "")
        printString = "unsuccessful";
    else
        printString.erase(printString.length() - 1, 1);

    return printString;
```

```
    }

    std::string inOrderSearch(std::shared_ptr<Node> current, std::string name)
    {
        if (current)
        {
            std::shared_ptr<Node> left = current->left;
            std::shared_ptr<Node> right = current->right;

            if (current->getName() == name)
                return inOrderSearch(left, name) + std::to_string(current-
>getID()) + "\n" + inOrderSearch(right, name);
            else
                return inOrderSearch(left, name) + inOrderSearch(right, name);
        }
        else
        {
            return "";
        }
    }
```

printInorder()

The method requires a full traversal of the tree, touching every node in an LNR order. Because the function does not loop more than once, *printInOrder()* is O(n), where n is the number of nodes in the tree.

```
    std::string printInOrder()
    {
        std::string printString = printInOrder(this->root);

        if (printString != "")
        {
            printString.erase(0, 2);
            //printString.erase(printString.length() - 2, 2);
        }

        return printString;
    }

    /**
     * @brief
     * Prints out a comma separated in-order traversal of the names in the tree,
starting at a certain node
```

```
     * @param current Starting node to begin the in-order traversal
     * @return String containing the list of nodes in order
     */
    std::string printInOrder(std::shared_ptr<Node> current)
    {
        if (current == nullptr)
        {
            return "";
        }
        else if (current->left == nullptr && current->right == nullptr)
        {
            return ", " + current->getName();
        }
        else
        {
            return printInOrder(current->left) + ", " + current->getName() +
printInOrder(current->right);
        }
    }
```

printPreOrder()

The method requires a full traversal of the tree, touching every node in an NLR order. Because the function does not loop more than once, *printPreOrder()* is O(n), where n is the number of nodes in the tree.

```
    std::string printPreOrder()
    {
        std::string printString = printPreOrder(this->root);
        if (printString != "")
            printString.erase(printString.length() - 2, 2);

        return printString;
    }

    /**
     * @brief
     * Prints out a comma separated pre-order traversal of the names in the tree,
starting at a certain node
     * @param current Starting node to begin the pre-order traversal
     * @return String containing the list of nodes in pre-order
     */
    std::string printPreOrder(std::shared_ptr<Node> current)
    {
```

```
        if (current)
        {
            std::shared_ptr<Node> left = current->left;
            std::shared_ptr<Node> right = current->right;
            return current->getName() + ", " + printPreOrder(left) +
printPreOrder(right);
        }
        else
        {
            return "";
        }
    }
```

printPostOrder()

The method requires a full traversal of the tree, touching every node in an LRN order. Because the function does not loop more than once, *printPostOrder()* is O(n), where n is the number of nodes in the tree.

```
    std::string printPostOrder()
    {
        std::string printString = printPostOrder(this->root);

        if (printString != "")
            printString.erase(0, 2);
        return printString;
    }

    /**
     * @brief
     * Prints out a comma separated post-order traversal of the names in the
tree, starting at a certain node
     * @param current Starting node to begin the post-order traversal
     * @return String containing the list of nodes in post-order
     */
    std::string printPostOrder(std::shared_ptr<Node> current)
    {
        if (current)
        {
            std::shared_ptr<Node> left = current->left;
            std::shared_ptr<Node> right = current->right;
```

```
            return printPostOrder(left) + printPostOrder(right) + ", " + current-
>getName();
        }
        else
        {
            return "";
        }
    }
```

printLevelCount()

Rather than storing subtree heights in each node, this method solves for the height. However, the method does a full traversal of the tree to find the max height of each subtree, touching every node on the way back down, as well as doing comparisons at every junction on the way back up the tree. Because *getHeight()* touches every node once on the way down and performs a calculation once on the way back up, this gives *getHeight()* a time complexity of O(n), since it does not loop over itself, goes over each node twice in the tree, where n is the number of nodes in the tree.

```
    int getHeight()
    {
        return getHeight(this->root);
    }

    int getHeight(std::shared_ptr<Node> current)
    {
        if (!current)
        {
            return 0;
        }

        std::shared_ptr<Node> left = current->left;
        std::shared_ptr<Node> right = current->right;

        int height_L = 1 + getHeight(left);
        int height_R = 1 + getHeight(right);

        if (height_L > height_R)
            return height_L;
        else
            return height_R;
    }
```

removeInOrder(int n):

```
bool removeInOrder(int n)
{
    std::shared_ptr<Node> current = this->root;

    int removeID = -1;

    removeInOrderSearch(current, n+1, removeID);

    if (removeID == -1)
        return false;
    else
        return remove(removeID);
}

void removeInOrderSearch(std::shared_ptr<Node> current, int n, int &removeID)
{
    static int i = 0;

    if (!current)
        return;

    if (i <= n)
    {
        std::shared_ptr<Node> left = current->left;
        std::shared_ptr<Node> right = current->right;

        removeInOrderSearch(left, n, removeID);
        i++;

        if (i == n)
            removeID = current->getID();

        removeInOrderSearch(right, n, removeID);
    }
    else
        return;
}
```

removeInOrder() has two parts: the search and the deletion. The search is O(n) time, because in the worst case, the method must traverse every node in the tree in order, an O(n) time complexity, where n is the number of nodes in the tree. However, the deletion simply calls the *remove(int id)* function, which has a time complexity of O(log(n)), where n is the number of

nodes in the tree. This gives the removeInOrder() function a time complexity of O(n + log(n)), which reduces to O(n), since the time complexity of both search and deletion is dependent on n, the number of nodes in the tree.

Reflection

I learned a lot about building sustainable and modular code. It was easy to think about solutions to parts of the AVL tree problems, and simple enough to code one function for it. But having to implement the entire tree forced me to consider how I could build my code in a way that would create a smooth workflow. I often ended up recoding many lines of code. An example of this was determining whether or not a given node was a left or right child of its parent. If I had made a function for it, I could have prevented many hours of debugging when manually writing each comparison to check for it. Not only would it have saved me time writing code, it would have made it easier to read, easier to fix, and easier to debug. This was one of many examples, of how not modularizing my code more caused problems. I learned my lesson after I completed the initial avl.h header file and made sure to modularize my code when working with the CLI for the main() method.

Two things I greatly improved on with this project were testing/debugging skills and memory management. I decided to use smart pointers in my C++ code, rather than typical pointers to prevent low-level memory management issues with nullptrs. This made memory management much simpler, with the only issue I ran into with shared_ptrs being their falling out of scope during recursive function calls. When it came to testing and debugging, I felt much more confident in my ability to solve a problem after using GTest with my code. The process of setting up unit tests made it very easy to set clear goals of what my code needed to do, as well as making the debugging process *magnitudes* faster with unit tests clearly pointing out issues in my code, as well as highlighting edge cases. I will continue to use GTest in the future, especially when managing large projects.