Project 5 - Map ADT: AVL Tree

Learning Objectives

- Demonstrate effective use of memory management techniques in C++
- Implement a data structure to meet given specifications
- Design, implement, and use an AVL tree data structure
- Analyze operations for time complexity

Overview

Your task for this assignment is to implement an AVL tree that serves as a map data type (sometimes also called a dictionary). A map allows you to store and retrieve *key-value* pairs. For this project, the *key* will be `std::string` and the *value* will be a `size_t`.

The AVLTree Class

You will create a C++ class named `AVLTree`. For this project, you must write your own AVL tree - not using code from outside sources except for class notes and the textbook. Your AVL tree should remain balanced by implementing **single** and **double** rotations when **inserting** and **removing** data. Your tree must implement the following methods (ensure you copy the names, parameter types, and return types exactly):

`bool AVLTree::insert(const std::string& key, size_t value)`

Insert a new *key-value* pair into the tree. After a sucessful insert, the tree is rebalanced if necessary. *Duplicate keys are disallowed.* The `insert()` method should return `true` if the insertion was successful, `false` otherwise. If the insertion was unsuccessful, such as when a duplicate is attempted to be inserted, the method should return `false`.

The time complexity for insert must be $\mathcal{O}(\log_2 n)$.

`bool AVLTree::remove(const std::string& key)`

If the *key* is in the tree, `remove()` will delete the *key-value* pair from the tree. The memory allocated for the node that is removed will be released. After removing the *key-value* pair, the tree is rebalanced if necessary. If the *key* was removed from the tree, `remove()` returns `true`, if the *key* was not in the tree, `remove()` returns `false`.

The time complexity for remove must be $\mathcal{O}(\log_2 n)$.

`bool AVLTree::contains(const std::string& key) const`

The `contains()` method returns `true` if the *key* is in the tree and `false` if the *key* is not in the tree.

The time complexity for `contains()` must be $\mathcal{O}(\log_2 n)$.

`std::optional<size_t> AVLTree::get(const std::string& key) const`

If the *key* is found in the tree, `get()` will return the *value* associated with that *key*. If the *key* is not in the tree, `get()` will return something called `std::nullopt`, which is a special *value* in C++. The `get()` method returns `std::optional<size_t>`, which is a way to denote a method might not have a valid *value* to return. This approach is nicer than designating a special *value*, like -1, to signify the return *value* is invalid. It's also much better than throwing an `exception` if the *key* is not found.

The time complexity for `get()` must be $\mathcal{O}(\log_2 n)$.

`size_t& AVLTree::operator[](const std::string& key)`

The bracket operator, `operator[]`, allows us to use our map the same way various programming languages such as C++ and Python allow us to use keys to access values. The bracket operator will work like `get()` in so that it will return the *value* stored in the node with the given *key*. We can retrieve the *value* associated with a *key* by saying:

```
size_t idNum = avlTree["James"];
```

However, the bracket operator returns a reference to that *value*. This means we can update the *value* associated with a *key* like:

```
avlTree["James"] = 1234;
```

You do not need to handle missing/invalid keys inside `operator[]`.

The time complexity for `operator[]` must be $\mathcal{O}(\log_2 n)$.

```
vector<std::string> AVLTree::findRange( const std::string& lowKey,
                                         const std::string& highKey) const
```

The `findRange()` method should return a C++ `std::vector` of `size_t` containing all the values associated with keys ≥ `lowKey` and keys ≤ `highKey`. For each *key* found in the given range, there will be one *value* in the vector. If no matching *key-value* pairs are found, the function should return an empty vector.

For example, if `findRange("G", "W")` were called with the tree in Figure 1 the resuling vector would contain: `{71, 74, 83}`, in no particular order.

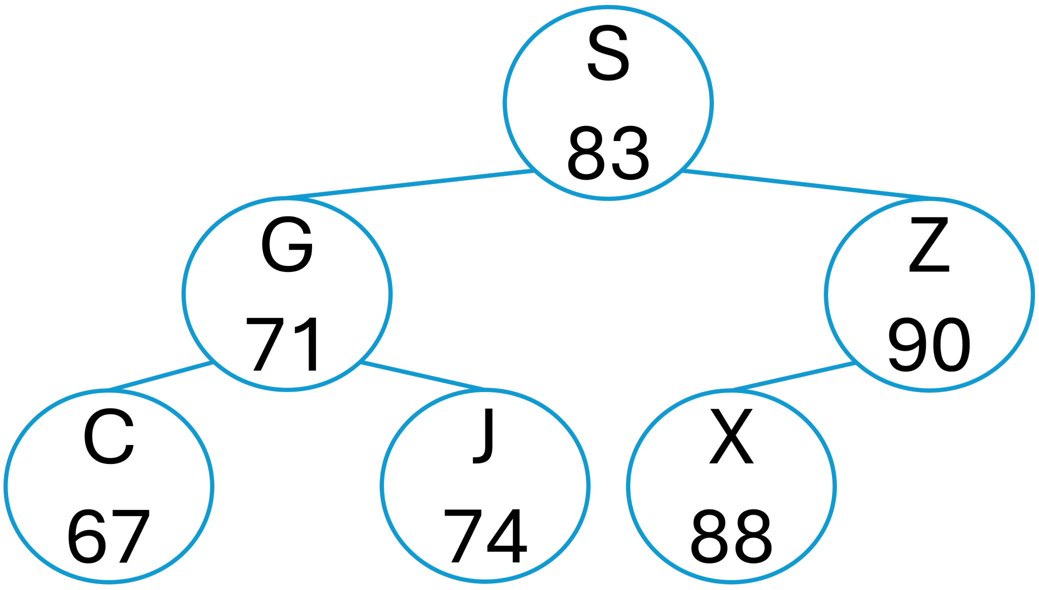

Figure 1: Example BST with string-int *key-value* pairs

```
std::vector<std::string> AVLTree::keys() const
```

The `keys()` method will return a `std::vector` with all of the keys currently in the tree. The length of the `vector` should be the same as the size of the tree.

```
size_t AVLTree::size() const
```

The `size()` method returns how many *key-value* pairs are in the tree.

The time complexity for this method must be $\mathcal{O}(1)$.

```
size_t AVLTree::getHeight() const
```

The `getHeight()` method will return the height of the AVL tree.

The time complexity for `getHeight()` must be $\mathcal{O}(1)$.

```
AVLTree::AVLTree(const AVLTree& other)
```

When an object is passed into a function in C++, unless it is passed by reference or pointer, all the data in the object is copied to the parameter. If we pass our AVLTree to a function, the only thing that gets copied is the root pointer (as well as any other member data you have). This means the AVLTree inside the method and the original object both have root pointers to the same root node in memory. Thus, any operation which mutates any of the nodes will affect both trees.

This is probably not what we want, so we need to create a **deep copy** of our tree. To do this in C++, we implement what is called the **copy constructor**. The copy constructor takes as a parameter the object being copied from. For our copy constructor, we want to visit all the nodes of the `other` tree (maybe a preorder traversal?) and create new nodes that contain the *key*, *value*, and whatever other member data the original node stored. The resulting copy will have the same structure as the original and be completely independent.

`void AVLTree::operator=(const AVLTree& other)`

Along with `operator[]` and `operator<<`, we can overload `operator=`, or the assignment operator. If we do not, the same thing happens if we don't implement the copy constructor. So, `operator=` needs to also create a **deep copy** of the `other` tree. The main difference is the tree we want to copy into may already have had elements inserted, so that memory needs to be released.

`AVLTree::~AVLTree()`

When an object goes out of scope, the memory used by its member variables is released back to the operating system. This means only the `root` variable is released, but not the memory `root` points to. In order to release the memory occupied by our nodes, we need to implement what C++ calls the **destructor**. A better name might be **deconstructor**, because as the constructor initializes any memory for the object, the destructor will release that memory. When an object goes out of scope, if it has a destructor defined, the destructor is called. Thus, we want our destructor to visit all the nodes in our tree (a postorder traversal?) and use `delete` to release the memory taken by each node.

`friend std::ostream& operator<<(ostream& os, const AVLTree & avlTree)`

In addition to these methods of AVLTree, you will also implement an operator to easily print out the tree. The stream insertion operator, `operator<<`, is not a method of the AVLTree class, and as such does not have access to the private data of AVLTree. To get around this, you can declare `operator<<` as a `friend` function inside of the AVLTree class (put the keyword `friend` before the declaration, but not the definition), which will give it access to the private member data and functions.

`operator<<` is another example of operator overloading in C++, similar to the bracket operator. This operator will allow us to print the contents of our tree using the normal syntax:

```
cout << myAvlTree << endl;
```

You should output the *key-value* pairs in the tree in a way that represents the structure of the tree. One approach would be to print the tree "sideways" using indentation to show the structure. For example, the tree in Figure 1 could be printed like Code Fig. 1 If you turn your head sideways, you can see how this represents the tree with `{S: 83}` as the `root` with no indentation, the root's children `{G: 71}` and `{Z: 90}` indented a certain amount, and their children `{C: 67}`, `{J: 74}` and `{X: 88}` indented by twice as much.

This style of printout can be achieved by doing a **right-child-first in-order traversal** of the tree, with each call passing in the current `depth + 1` to use as an indentation factor.

```
    {Z: 90}

        {X: 88}

{S: 83}

        {J: 74}

    {G: 71}

        {C: 67}
```

Code Fig. 1: Output of tree, but sideways

There are more methods you need to implement in order to implement the operations recursively, but the signature of those methods is up to you.

BSTNode Class

Similar to other projects, we will store everything associated with a *key-value* pair in a *node*. The `BSTNode` (or `AVLNode`) should have, at a minimum, `std::string key`, `size_t value`, `BSTNode* left`, and `BSTNode* right`. Additionally, since the `height()` method needs to be $\mathcal{O}(1)$, you will also want to store a member in each node for the `height` of the node. The `height` will then need to be updated on every call to `insert()` and `remove()`. Storing the height in the node allows you to calculate a node's height in $\mathcal{O}(1)$ by checking the `height` of it's `left` and `right` children. It also allows you to return the `height` of the tree in $\mathcal{O}(1)$ as the height of a tree is the height of it's root node.

I recommend having in your BSTNode class methods to perform operations such as finding the current `height` of the node, finding the `balance` of the node, finding how many **children** the node has, and possibly whether or not a node is a **leaf** or not.

Keep in mind, these methods contribute to the time complexity of the algorithm they are utilized in, such as `insert()` and `remove()`. Therefore, to keep those operations in $\mathcal{O}(\log_2 n)$, they cannot be any worse than $\mathcal{O}(\log_2 n)$. If the method to find the height of a node is $\mathcal{O}(n)$, that makes any operation that uses it *at least* $\mathcal{O}(n)$.

Recursion

For this project, the methods `insert()`, `remove()`, `contains()`, `get()`, `operator[]`, `findRange()`, `keys()`, the *copy constructor*, the *assignment operator*, and the *destructor* ***must*** utilize recursion. This means for most methods, you will need to write the recursive code inside a different method than the actual method, since a recursive method would need to pass a pointer to a node, which none of the required methods take.

Some tips for approaching recursion:

- **Base-case**: For most methods, the *base-case* will be a check that the current node is `nullptr`, meaning that the recursion has reached the bottom of the tree. This could indicate that the *key* you are searching for is not in the tree, or that this is where you will insert a new node. A second base-case could be when we find the *key*, such as when searching or removing from the tree.
- **You can do stuff after a recursive call** - inside of a recursive function. For example, after a recursive call to `insert()` is done, you will want to update the `height` and check the `balance` of the current node. Since the recursive calls pop off the stack in the *opposite* order of what they were called, that

means you are traversing back up the tree to the `root` and can check balances and do a rotation as the recursion is rewinding.

- **You can pass pointers by reference**. This isn't so much a recursive tip; however it will make your life a lot easier. This will let you assign a pointer is passed in to the memory address of a new or different object.

To demonstrate this, the `passInt()` function in Code Fig. 2 takes an integer and assigns it the *value* of `7`. Once `passInt()` is finished, whatever variable we pass into `passInt()` is not modified, because the variable is copied into the function. However, in `passIntRef()` in Code Fig. 3, the variable we pass is in fact assigned the *value* `7` (try it, that's what happens).

```
void passInt(int value) {
    value = 7;
}
```

Code Fig. 2: Passing `int` by *value*

```
void passRef(int& ref) {
    value = 7;
}
```

Code Fig. 3: Passing `int` by *reference*

Similarly, I can pass pointers by reference.

In `passIntPtr()` in Code Fig. 4, we create a new `int` and assign the memory address of where that integer is into the parameter `ptr`. When we pass a variable into `passIntPtr()`, although we are assigning it inside of `passIntPtr()`, the same thing happens as what happened in `passInt()`. The variable that is passed into `passIntPtr()` is not modified, it still has the same memory address before and after passing it.

```
void passIntPtr(int* ptr) {
    ptr = new int(7);
}
```

Code Fig. 4: Passing pointer by *value*

In `passIntPtrRef()` in Code Fig. 5, we are doing a similar thing that we did in `passIntRef()`. In this case, instead of passing an integer by *reference*, we are passing an integer *pointer by reference*. So, whatever variable we pass as the parameter to `passIntPtrRef()` will be reassigned to a different memory address, in this case it will be a pointer to a memory location where the number 7 is stored. (again, try it, that's what happens).

```
void passIntPtrRef(int*& ptrRef) {
    ptrRef = new int(7);
}
```

Code Fig. 5: Passing pointer by *reference*

How does that help us? If instead of a pointer to an integer, what if instead I passed a pointer to a `BSTNode` *by reference*? It would work the same. I could assign the node pointer to the address of a new or different node.

In `passNodePtrRef()`, shown in Code Fig. 6, like `passIntRef()` and `passIntPtrRef()`, when I assign node to a `new Node`, the variable I pass into `passNodePtrRef()` will be assigned the pointer to the `new`

Node. If I passed in **tree.root**, after the function returned, **tree.root** would be pointing to a node that contained 7 (in this case **Node** has a member variable called **data** that will be assigned the *value* 7).

```
void passNodePtrRef(Node*& node) {
  node = new Node(7);
}
```

Code Fig. 6: Passing a **Node** pointer by *reference*

If your AVLTree does not utilize recursion for insert, remove, contains, get, operator[], findRange, keys, the copy constructor, the assignment operator, and the destructor, you will receive no credit for this assignment.

Turn in and Grading

You will implement AVLTree.h and AVLTree.cpp provided in the GitHub Classroom assignment, making commits and pushing them to the remote branch. The last commit that appears in the remote branch will be graded.

While you are implementing the AVLTree, make sure you commit *regularly* and *frequently*. My recommendation is, each time you implement a new feature, you should test it, and once it works, make a commit saying, "feature *x* working". If your repository has just a few very large commits in relation to the rest of the commits in the repo history, that can be grounds for an academic integrity violation.

Please remember the policy for this course is **A.I is disallowed on everything**. A.I use will result in an academic integrity violation. If you have any questions about this policy, or you are having difficulties implementing this assignment, please ask me as soon as possible, over e-mail, during my office hours, in person after class, or over Discord. You may also consult with our TA for additional assistance. This project may very well be the most difficult assignment you will encounter in your entire undergraduate career, so please plan accordingly.

This project is worth 50 points, distributed as follows:

Test	**Points**
AVLTree::insert() stores *key-value* pairs in the tree and correctly rejects duplicate keys. After **insert()** the tree remains balanced. Time complexity of **insert()** is $\mathcal{O}(\log_2 n)$.	6
AVLTree::remove() correctly finds and deletes nodes from the tree without interfering with subsequent search and insert operations. After **remove()** tree remains balanced. Time complexity of remove is $\mathcal{O}(\log_2 n)$.	6
AVLTree::contains() correctly returns **true** if the *key* is in the tree and **false** otherwise. Time complexity of **contains** is $\mathcal{O}(\log_2 n)$.	5
AVLTree::get() correctly finds, and returns the *value* associated if the *key* is found. Returns **nullopt** when the *key* is not in the tree. Time complexity of get is $\mathcal{O}(\log_2 n)$.	4
AVLTree::operator[] correctly returns a reference to the *value* associated with the given *key*. There is no requirement for your project to handle missing keys in **operator[]** Time complexity of **operator[]** is $\mathcal{O}(\log_2 n)$.	4

Test	**Points**
`AVLTree::findRange()` correctly returns a C++ `std::vector<size_t>` matching the keys in the specified range	4
`AVLTree::keys()` correctly returns a `std::vector<size_t>` with all of the keys currently stored in the tree	2
`AVLTree::size()` correctly returns the number of *key-value* pairs in the tree in $\mathcal{O}(1)$. time	1
`AVLTree::getHeight()` correctly returns the height of the tree in $\mathcal{O}(1)$. time.	1
`operator<<` is correctly overloaded as described above	3
Copy constructor correctly creates an independent copy of an AVL tree	3
`AVLTree::operator=` correctly creates an independent copy of the tree	3
Code has no memory leaks	3
Code is well organized, documented, and formatted according to the course Coding Guidelines.	5
Total	**50**