

## Project #4 - Indexing with AVL Trees

### 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

---

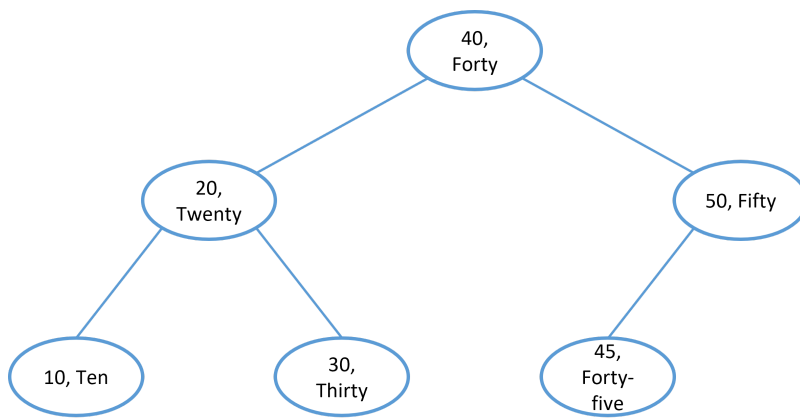
Your task for this assignment is to implement an AVL tree that serves as a *map* data type (sometimes also called a *dictionary*). A map allows you to store and retrieve key/value pairs. For this project, the key will be an integer and the value will be a string.

### The AVLTree Class

---

The map will be implemented as an AVL tree. For this project, you must write your own AVL tree - not using code from outside sources. Your AVL tree should remain balanced by implementing single and double rotations when inserting new data. Your tree must support the following operations:

- `bool AVLTree::insert(int key, string value)` – Insert a new key/value pair into the tree. For this assignment the duplicate keys are not allowed. This function should return **true** if the key/value pair is successfully inserted into the map, and **false** if the pair could not be inserted (for example, due to a duplicate key already found in the map). The time complexity for insert should be  $O(\log_2 n)$ .
- `int AVLTree::getHeight()` – return the height of the AVL tree. The time complexity for getHeight should be  $O(1)$ .
- `int AVLTree::getSize()` – return the total number of nodes (key/value pairs) in the AVL tree. The time complexity for getSize should be  $O(1)$ .
- `friend ostream& operator<<(ostream& os, const AVLTree& me)` - print the tree using the << operator. You should overload the << operator to print the AVL tree “sideways” using indentation to show the structure of the tree. For example, consider the following AVL tree (each node contains a key, value pair):



This tree would be printed as follows:

```

      50, Fifty
        45, Forty-five
40, Forty
      30, Thirty
    20, Twenty
      10, Ten
  
```

*(Note: If you turn your head sideways, you can see how this represents the tree.)*

*(Also note: This style of printout can be directly implemented as a right-child-first inorder traversal of the tree.)*

- **bool AVLTree::find(int key, string& value)** – if the given key is found in the AVL tree, this function should return **true** and place the corresponding value in **value**. Otherwise this function should return **false** (and the value in **value** can be anything). The time complexity for find should be  $O(\log_2 n)$ .
- **vector<string> AVLTree::findRange(int lowkey, int highkey)** – this function should return a C++ vector of strings containing all of the values in the tree with keys  $\geq$  lowkey and  $\leq$  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.

**Example:** Suppose the call `resultVector = myTree.findRange(30, 47)` were called on the tree pictured above. The findRange function would then return a vector containing the strings: {"Thirty", "Fourty", "Forty five"}.

## Turn in and Grading

- The AVLTree class should use a separate AVLTree.h and AVLTree.cpp file.
- Please zip your entire project directory into a single file called project4.zip and upload to the dropbox in Pilot.

This project is worth 50 points, distributed as follows:

Task	Points

<b>AVLTree::insert</b> stores key/value pairs in the correct locations in the AVLTree, and correctly rejects duplicate keys	3
<b>AVLTree::getHeight()</b> correctly returns the height of the tree	3
<b>AVLTree::getSize()</b> correctly returns the number of key/value pairs in the tree	3
The tree maintains correct balance, regardless of the order in which keys are inserted	10
<b>operator&lt;&lt;</b> prints the tree in a neat and readable manner, using indentation or some other appropriate mechanism to clearly show the structure of the tree	4
<b>AVLTree::find</b> correctly finds and returns key/value pairs in the tree in $\Theta(\log n)$ time, and returns <b>false</b> when no matching key is found	4
<b>AVLTree::findRange</b> correctly returns a C++ vector of strings matching keys in the specified range	6
<b>AVLTree::operator=</b> correctly creates an independent copy of an AVL tree	4
Copy constructor correctly creates an independent copy of an AVL tree	4
Code has no memory leaks	4
Code is well organized, well documented, and properly formatted. Variable names are clear, and readable. Your AVLTree class is declared and implemented in separate (.cpp and .h) files.	5