* Due: Friday, December 6th, 11:59pm
* Directory name for this homework (case sensitive): hw6
  + This directory should be in your hw\_username repository
  + This directory needs its own README.md file
  + You should provide a Makefile to compile the code.

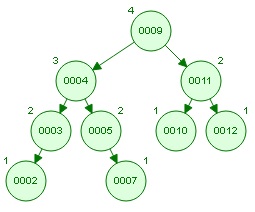
**Problem 0 (Course Evaluations, 0%)**

Please make sure to fill out the online course evaluations for this course. The instructor takes these seriously, and you can have a positive impact on how this class is taught in future semesters. Pay it forward!

**Problem 1: Making sure you understand Trees (10%)**

**AVL Trees**

Consider the following initial configuration of an AVL Tree:



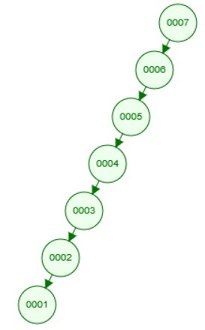
Draw the tree representation of the AVL tree after each of the following operations, using the method presented in class. Submit it as hw6.txt, or any other file extension you prefer, but indicate the filename in your README.

Your operations are done in **sequence**, so your tree should have 11 values in it when you are done. Make sure to clearly indicate each of your final answers.

* Insert 1
* Insert 6
* Insert 8
* Insert 13
* Remove 5
* Remove 1

**Splay Trees**

Consider the following initial configuration of a Splay Tree:



Draw the tree representation of the Splay tree after each of the following operations, using the method presented in class. Continue your answers in hw6.txt.

Your operations are done in **sequence**, so your tree should have 7 values in it when you are done. Make sure to clearly indicate each of your final answers.

* Find 1
* Insert 8
* Remove 4

We highly recommend you try to solve these by hand before using any tools to verify your answers.

**Problem 2: Implementing Binary Search Trees (15%)**

We are providing for you a half-finished file bst.h (in the homework-resources repository) which implements a simple binary search tree. You will need to fill in the implementation for the following functions inside BinarySearchTree:

* Constructor
* Destructor
* Insert
* Clear
* InternalFind
* GetSmallestNode

Note: For the insert() method, you should handle duplicate entries by overwriting the current value with the updated value.

**Problem 3: Implementing AVL Trees (35%)**

We are providing you a half-finished file avlbst.h (in the homework-resources repository) for implementing an AVL Tree. It builds on the file you completed for the previous question.

Complete this file by implementing the insert() and remove() functions for AVL Trees. You are strongly encouraged to use private/protected helper functions, and to put some of them in the BinarySearchTree file, so that your Splay Tree for problem 4 can access them (such as a BST::remove() function).

**Notes and Requirements**

* When deleting a key that has 1 child, simply have the parent of the deleted node adopt the child.
* When deleting a key that has 2 children, always swap it with its successor.
* When writing a class (AVLTree) that is derived from a templated class (BinarySearchTree), accessing the inherited members must be done by scoping or preceding with the keyword this->.
* Your AVLTree will inherit from your BST. This means the root member of BST is a Node type pointer that points to an AVLNode (since you will need to create AVLNodes with height values for your AVLBST), which is fine because a base Node pointer can point at derived AVLNodes. If you need to call AVLNode-specific functions (i.e., members that are part of AVLNode but not Node) then one simple way is to cast your Node pointer to an AVLNode pointer, as in static\_cast<AVLNode<K,V>\*>(root) to temporarily access the AVLNode-specific behavior/data.
* When you compile code that includes avlbst, you will need to add the option --std=c++11 to the g++ compilation command. This is because of the usage of the override keyword for various virtual functions. You can read about it online but it mainly provides some additional compiler checks to ensure the signatures of a virtual function in the derived class matches the one you are attempting to "override" in the base class (i.e. if your base virtual function is a const-member but you forget to add const to the derived and thus are creating a whole new member function, the compiler will catch the error).
* Feel free to add any private/protected functions you like.

**Problem 4 Implementing Splay Trees (25%)**

Complete the file splay.h for implementing a Splay Tree. You will need to implement the insert() and remove() functions.

Recall that when you delete from a Splay Tree, you should take the parent of the deleted node (which is likely to be the predecessor or successor of the value you deleted) and splay it to the top.

**Problem 5 (Analyze the Performance of your Data Structures, 15%)**

Create a program analysis that should be created using make analysis, and will be called as follows:

./analysis [input] [output]

Here, [input] will be an input text file, which you will run your program on. Your program should take each string in the text file (a sequence of characters separated by 1 or more newlines, tabs, or spaces) and insert them into your data structures. [output] is an output text file, into which you will store your results.

Here are the steps that your program should perform:

1. Read all of the strings into a vector<string>.
2. Create an AVL Tree and populate it (insert) with all the strings.
3. Create a Splay Tree and populate it with all the strings.
4. Report timing information to the output file.

Your program should separately time and report the duration for 2 and 3. Information on how to time your code is given below. The reason we make you read/write everything into arrays first is that file access is really really slow compared to memory access. Thus, if you ran steps 2 and 3 directly from the input file, or wrote directly to the output file, the time for file access would almost certainly completely drown out the differences between AVL Trees and Splay Trees. So please don't try to take the shortcut of interacting with files while you're timing.

Note that you don't really need the Value of the (key, value) pair for this application, so feel free to place whatever value you like here.

Your output file should report the following:

* total number of insertions performed
* total time for insertions into the AVL Tree
* total time for insertions into the Splay Tree
* total number of insertions into the Splay Tree where the node was added at a level strictly greater than 2 \* log(n)

Report these statistics as in the following example (all these numbers are completely made up, so don't read anything into them at all). Please follow this format so that our script can match.

1000 insertions

AVL: 0.1417 seconds

Splay: 0.1243 seconds

Splay expensive: 42

**Timing Your Code**

For more detailed information on how to time code, check the course coding guide. Here is a briefer summary. Basically, you do something like the following:

#include <iostream>

#include <ctime>

int main() {

clock\_t start;

double duration;

/\* Preprocessing here that you don't want to time \*/

start = clock();

/\* Your algorithm here \*/

duration = ( clock() - start ) / (double) CLOCKS\_PER\_SEC;

std::cout << duration/r << endl;

}

The problem with that is if the algorithm runs fast enough, duration will always be 0, and you can't distinguish 0.000001 seconds from 0.0000000001 seconds, which is a huge difference. The solution is to repeat the algorithm many times, then take the average. See the course coding guide if you need more detail.

**What to Report**

In your README file, report on the following:

* Which sizes of input cases did you test your program on? This would apply both to the length m of the strings in the file, and the number n of strings/queries.
* For each of the two data structures, report how long each of your input cases took. How long did it take per operation? Did it depend on the parameters, and if so how? How did that depend on the parameters?
* Explain why you think the results turned out the way they did.
* How do you think the running time would compare if you were to implement and analyze the following data structures? Briefly justify each, although you do not need to quantify how much slower/faster with any specificity.
  + Unsorted List
  + Sorted list
  + Binary search tree, non-rotating variety
  + Hash Table, with universal hash function

**Chocolate Problem (1 Chocolate Bar): Merging Startups, Revisited**

Remember the merging startups from Homework 3? We will modify the problem a little bit now. Companies will not split any more. And you don't need to worry about any destructors. So what you want to be able to do is just merge the companies of two given students (assuming that they aren't already in the same company), and to query whether two given students are currently in the same company or not. That's it. But now, every operation has to run in time O(log n), where n is the total number of students.

Design a data structure that supports these two operations to run in worst-case time O(log n), and prove that this is in fact the worst-case running time. You do not need to provide actual C++ code - clear pseudo-code is enough. The important part is the proof.

If you are **really** ambitious, you can try to design a data structure in which the worst-case **amortized** cost per operation is less than O(log n). One can get it down to almost constant, definitely much less than O(log log n). But this is quite a lot harder.