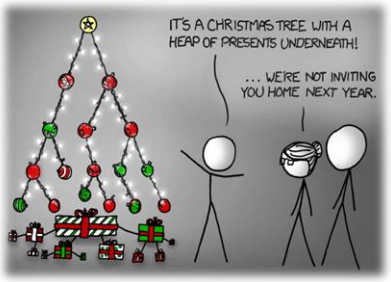

CS515 – Lab 9

The purpose of this lab is to review AVL and B-trees, as well as the implementations of some of their basic functionalities.

Download all files from `~cs515/public/9L`

Task 1: Implement AVL rotations

The starter code is a partial implementation of the Set ADT using an AVL tree, which is one of the first-invented self-balancing binary search trees. An AVL tree is "almost" balanced. Local rotations are used for rebalancing following some insert and delete operations. In this (unfinished) implementation, each AVL tree node knows its own height, rather than knowing its balance factor. Thus, the balance factor, when needed, is calculated on the fly from the heights of the node's left and right nodes.

First look at the output of our driver program `settest0.cpp` saved in `settest0.output`. Check carefully how the AVL tree rotates, either singly or doubly, as new elements are inserted into the tree or deleted from the tree. Identify cases of single rotation and double rotation from the output of the program. Make sure you understand how these two types of rotation work.

Your task is to complete the following four rotation methods

`rotateLeft`, **`doubleRotateLeft`**, **`rotateRight`** and **`doubleRotateRight`**

You must not modify the header file **`set.h`** while implementing these methods.

Task 2: Implementing the internal-node insert() for a B-tree

You are given a partial implementation of a B-Tree in the files `btree.h` and `btree.cpp`, along with a test driver `btreetest0.cpp`. First look at the driver program's output `btreetest0.output`. Carefully note how the tree is built and how node splitting happens, especially that no data is stored in internal nodes—only copies of keys! Make sure you understand the final structure of the tree printed from the test driver. Note the tree is printed lying down; smaller values are on the top/right and larger values are on the left/bottom.

You need to finish the implementation of `btree.cpp` by completing the **`insert()`** method for internal nodes. You should not modify the header file `btree.h`. Once you are finished, compile the test driver with your `btree.cpp` file and verify the outputs are the same. You should not use any STL containers in your program. Use the following steps to get started.

1. **Look at btree.h**. The key structure here is the BTreeNode. It is an abstract class because it has no implementations of insert and dump.
2. The key pieces of information in a BTree node are the **keys** (each has 3 keys starting at index 1, and a "dummy" key at index 0) and the **size** (how many keys are present in the node).
3. Child classes of BTreeNode are BLeaf and BInternal. BLeaf has only an array of **data**, since BLeaf's cannot have children
4. BInternal has only an array of **child** nodes, since BInternal's cannot store data.
5. Look at **btree.cpp**. Near the top, you can see the **insert method for a BLeaf** node. Note that it assumes there will be a place for the new data item, and starts moving things over until it find the right place. It then increments the size, and if the size is too big, it splits the node into two and **returns a pointer to the new node**.
6. Your job is to implement the **insert method for a Binternal** node, done roughly as follows:
   a) Find the child pointer the the key would insert to. Since this is an internal node, you are going to start off by just delegating the "actual" insertion work to the appropriate child--nothing changes in your node yet. Just find the key in your keys array that is less-than or equal-to the newKey being inserted. We'll use the index of that key in the next step.

b) Use the index you just found to access the **child** array, and then recursively call that child's insert with the same key/item information. We're going to need to change something here later, but this is ok for now.

c) If the child's insert returns NULL, then no new nodes were formed directly beneath us. However, if it did return a value, then let's store it in a BTreeNode called **split**. We need copy any higher-valued **keys** and their corresponding **child** pointers to the right to make room for the new **split** node. How do we know what value to use for the new key? Well, it turns out that the child is going to modify **its newKey** argument (notice that it is passed by reference) to tell us that. This means we don't actually want to pass **our original newKey** value to it in step b. Instead, we want the call in step b to use a copy, say **keyCopy**. Then we can use the value the child set in **keyCopy** when we're setting the value of the newly inserted key.

d) If this node overflows (size == ORDER), we need to split the node in two (using **new BInternal**) and return the new node. Half the items go in the node on the left, and half go in the new node on the right (remember to update their sizes). We need to set **newKey** to the value of the first key in the new node, and then also return a pointer to the new node.

You don't need to finish the Big Four methods for the given classes.

Important: You must not modify the existing header files `set.h` and `btree.h`, nor any other existing methods in the source files.

Submission:

- Submit the files `set.cpp` and `btree.cpp`.
- You may change the makefile locally, but we will be testing with our own.
- You should also fill out the README file and submit it as well. To submit the files in Mimir, follow the link for this lab in MyCourses. Here is a list of files you are expected to submit:

  `set.cpp btree.cpp README`

- As always, do not turn in executables or object code, and make sure your submission compiles successfully on Agate. Programs that produce compile time errors or warnings will receive a zero mark (even if it might work perfectly on your home computer.) To check this, use the make command with the provided makefile, and check that your submission passes tests on Mimir.

Important:
You must include the standard comment block in each of your source files, including your name, section, date and collaboration details. You must also finish the README file along with your programs.

You should also include detailed collaboration declaration information in your comments. If you worked in pairs in this lab, each of you must include the partner's name in your program comment. Both you and your partner must complete an individual submission in order to earn a grade. If you work by yourself, you must indicate in the program comments that you have worked on your own independently.

You can resubmit as needed—your last submission will be graded. Here is a tentative grading scheme:

```
set test run 1                  25
set test run 2                  10
set test run 3                  10
set test run 4                  25
btree test run 0                10
btree test run 1                10
btree test run 2                10
```