Dept. of Computer Science, University of California, Davis
ECS60 Instructor: Rob Gysel 3/2/18

Programming Assignment #3*
Due date: 3/9/18 11:59pm

Programs are to be submitted using handin on the CSIF by the due date using the command:

```
handin rsgysel 60-Program3 file1 file2 ... fileN
```

1 Overview & Learning Objectives

In this program you will implement an AVL tree. There are multiple objectives of this assignment:

1. strengthen your knowledge of JSON,
2. strengthen your understanding of code testing,
3. understand and implement an AVL Tree.

In this program, no autograding script will be given. Instead, instructions in this document will guide you to rigorously testing your code.
Refer to the syllabus for group work policies. You may use Rust, Java, or C++ for your code.
Students that work alone for all programs a receive a B- or better on the programs will receive 1% extra credit at the end of the quarter. Programs submitted up to 24 hours late will still be accepted but incur a 10% grade penalty.

2 AVL Trees

Create an AVL Tree as a C++ `class`, a Java `class`, or Rust `struct` called `AVLTree`. I have implemented a binary search tree (BST) in C++ using C++11 smart pointers `weak_ptr` and `shared_ptr`, which are built to help you manage memory. For a tutorial on smart pointers, see:

*Last updated March 2, 2018

`https://www.codeproject.com/Articles/541067/Cplusplus-Smart-Pointers`
Briefly, abide by the following rules if you wish to modify my BST tree to implement your AVL tree in C++:

- If v is a parent of u, then u has a `std::weak_ptr` that points to v.
- If w is a child of u, then u has a `std::shared_ptr` that points to w.
- Use `std::shared_ptr` to modify what is being pointed to. In order to do this for a parent, you must convert a `std::weak_ptr` to a `std::shared_ptr` using `lock`.
- To check for an empty smart pointer, compare a `std::shared_ptr` to `nullptr`.
- A `std::weak_ptr` may not be assigned `nullptr`; instead, use `reset` to have a `std::weak_ptr` "point to null".

Implement the following.

Insert Implement an `Insert` function that utilizes the rotations as described at GeeksForGeeks:
`https://www.geeksforgeeks.org/?p=17679`

Deletion Implement a `Delete` function as described at GeeksForGeeks:
`https://www.geeksforgeeks.org/avl-tree-set-2-deletion/`
Note that the article links to BST deletion:
`https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/`
When a node is deleted in a BST, if it has two children, either the inorder successor *or* the inorder predecessor is copied. Either choice is fine for this programming assignment.

DeleteMin Implement a `DeleteMin` function which deletes the minimum key in the AVL tree. *Any implementation that runs in $O(\log n)$ time is acceptable.*

As usual, input will be in the form of a JSON file, and output should be printed to the screen. You can use `CreateData.exe` to generate input. Calling `./CreateData.exe 5` on the command line results in a file like this:

```
{
  "1": {
    "key": 2109242329,
    "operation": "Insert"
  },
  "2": {
    "key": -948648564,
    "operation": "Insert"
  },
  "3": {
    "key": -948648564,
    "operation": "Delete"
  },
  "4": {
    "operation": "DeleteMin"
  },
  "5": {
    "key": -289891961,
    "operation": "Insert"
  },
  "metadata": {
    "numOps": 5
  }
}
```

Which results in a tree with one node with key -289891961. Your program will read in input file such as the above, perform the operations in order, and print a JSON object to the screen (`stdout`) with the following format:

- A `height` key (JSON key) whose value is the tree's height,
- A `root` key (JSON key) whose value is the root's key (AVL tree key),
- A `size` key whose value is the number of nodes in the tree, and
- For each node, a key/value pair where the JSON key is the node's key and the fields are:

  height the height of the node,

  balance factor the balance factor of the node, (use the right-left formula for this)

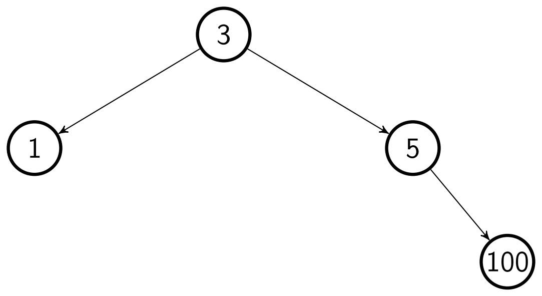
  parent the key of the parent node, if it exists

  left the key of the left child node, if it exists

  right the key of the right child node, if it exists

  root the value of true, if the node is the root, otherwise do not include this key.

For example, the tree given by:



Is encoded by the JSON object:

```
{
  "1": {
    "balance factor": 0,
    "height": 0,
    "parent": 3
  },
  "100": {
    "balance factor": 0,
    "height": 0,
    "parent": 5
  },
  "3": {
    "balance factor": 1,
    "height": 2,
    "left": 1,
    "right": 5,
    "root": true,
  },
  "5": {
    "balance factor": 1,
    "height": 1,
    "parent": 3,
    "right": 100
  },
  "height": 2,
  "root": 3,
  "size": 4
}
```

3 Testing your code

To test your code, modify `BSTSanityCheck.cxx` for use with your AVL tree. The code does the following:

lines 29-39: Creates a sequence of operations to test, including `Insert`s, `Delete`s, and `DeleteMin`s.

lines 40-44: Removes elements in sorted order from the tree, then compares the result to a sorted array.

In addition, you should incorporate the following facts:

1. The balance factor of every node should be between -1, 0, and 1.

2. The height of every node should be 0 for leaves, 1 + child height if a node has a single child, and 1 + the minimum of the children's height if a node has two children.

3. It turns out that for a tree of n nodes, an AVL tree has height less than $2 \log n$.

The grading process will take these factors into account. To thoroughly test your program, modify `BSTSanityCheck.cxx` to take the above considerations into account, then make `SAMPLE_SIZE` and `NUM_TESTS` large. Here, "large" means as large as you are willing to run for your program.

4 Files to turn in

Grading will be done by running a `compile.sh` script and `AVLCommands.sh` script. One way to implement this is as follows:

1. Create a file `AVLCommands.cpp`

2. Create the executable `AVLCommands` through your `compile.sh` script

3. Use the following contents for `AVLCommands.sh`:

```
#!/bin/bash
./AVLCommands $1
```

To determine what files you should turn in, do the following:

1. Upload your files to the CSIF using `scp`, `sftp`, or another `ftp` client.

2. Determine the list of files you intend to turn in. Write this list down.

3. Copy each file in your list to a directory called `P3Sub`.

4. Move to directory `P3Sub` and copy all files to a directory called `temp` (use something like the command `cp * temp` for this).

5. Move to directory `temp`. Use the `pwd` command to check that you are in the `temp` directory, and use the `ls` command to check that all relevant files are in `temp`.

6. Run `./compile.sh` to ensure everything compiles in `temp`.

7. Run `./AVLCommands.sh` on a sample input file (e.g. `SampleInput.json`) and ensure your code executes correctly.

8. If the above steps work, delete the `temp` directory, move to the `P3Sub` directory, and hand in all files in `P3Sub` using `handin`.