Due: 3/17/2019
ECS 36C Winter 2019

Program 4

Programs must be written in C++ and are to be submitted using handin on the CSIF by the due date (see Canvas) using the command:

`handin`

Your programs must compile and run on the CSIF. Use `handin` to submit *all* files that are required to compile (even if they come from the prompt). Programs that do not compile with `make` on the CSIF will lose points and possibly get a 0. Programs that have warnings during compilation will lose 10 out of 100 points.

An autograder along with examples and solutions will be available shortly.

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.

2 AVL Trees

Create a source file `AVL.cpp` and a header file `AVL.h` and implement an AVL Tree as a C++ `class` called `AVLTree`. A binary search tree is implemented in `BST.cpp` and `BST.h` 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:

`https://www.codeproject.com/Articles/541067/Cplusplus-Smart-Pointers`
Briefly, abide by the following rules to use smart pointers for your AVL tree:

- 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.

AVL Insertion: that utilizes the rotations as described at GeeksForGeeks:
https://www.geeksforgeeks.org/?p=17679

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)

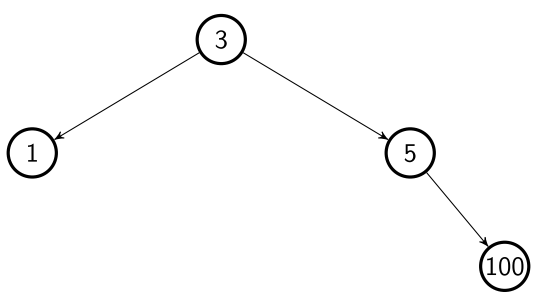
  parent the key of the parent node, if it exists

  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 AVLCommands

Write a C++ program `AVLCommands.cxx` that does the following:

1. takes an input JSON filename representing a sequence of AVL tree operations (`Insert`,) as its first command-line argument,
2. creates an AVL tree using the operations described in the input JSON file,
3. prints the resulting AVL tree JSON object to the screen (`stdout`).

Name your executable `AVLCommands.exe` and add lines to your `Makefile` to compile `AVL.o` and `AVLCommands.exe`.

4 Compilation

A `Makefile` has been provided for you. When you add new files to your program, you will need to edit this `Makefile`.

5 Testing your code

In addition to the autograder, you may do the following 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 $2 \log n$.

6 Files to turn in

1. `AVL.cpp`
2. `AVL.h`
3. `AVLCommands.cxx`
4. `Makefile`
5. `BST.cpp` (even if you have not modified it)
6. `BST.h` (even if you have not modified it)
7. `CreateData.cxx` (even if you have not modified it)