BINARY SEARCH TREES and SELF-BALANCING TREES

This lab must be done as an individual; no team work is permitted.

This is the third lab: it will be submitted via the submit command with an arg of 2.

This is due during your lab session on the week of Jan 28.

The concepts underlying this lab are simple; this can lull you into a sense that the implementation is similarly simple: a largely incorrect impression, especially the first time around. Start early, and work on this incrementally.

You will use the C programming language (the subset that we discussed in CSC180, that will compile error-free and warning-free via "gcc -pedantic").

A. PRELIMINARIES

You will require two print routines to help you debug your code:
print in order
print in level order.

I will not check your print functions, thus you can choose whether to, how to, etc. implement them.

Why do you need "print in order"?
Why do you need "print in level order"?
What additional data structure do you require to do the level order print?

I will define, and so will you, the following structs:

For binary search trees:
```
struct bstNode {
   int val;
   struct bstNode *l;
   struct bstNode *r;
};
typedef struct bstNode bstNode;
```

The "avlNode" struct will be used to realize your AVL tree.
Observe, it is a binary tree, with an additional element in the struct called the balance factor. The balance factor indicates whether the node is heavy in the left or right directions, or balanced. Since these are AVL trees, we can tolerate an imbalance of at most 1 in any direction.

```
struct avlNode {
   int balance; /* -1 Left, 0 balanced, +1 Right */
   int val;
   struct avlNode *l;
   struct avlNode *r;
};
typedef struct avlNode avlNode;
```

The "qNode" struct will be used to implement a linked list of avlNode *.
This is for one reason: the print in level order function. Why is that?
What helper functions do you require for this linked list: specifically, is this a general linked list, or a specialized list? If a specialized list, what kind is it specifically?

```
struct qNode {
   avlNode *pval;
   struct qNode *nxt;
};
typedef struct qNode qNode;
```

Below is the skeleton of an add function for a bst:

```
int add_bst(bstNode **root,int val) {
   if (root == NULL) { return -1; }
   if (*root == NULL) {
      /* DO SOMETHING ... */
   } else {
      /* DO SOMETHING ... */
   }
}
```

Complete the function.

As an exercise, consider the following main function. Write the
support functions for this (i.e., printTreeInOrder and printLevelOrder).

```
int main(void) {
   bstNode *root=NULL;
   add_bst(&root,5);
   add_bst(&root,3);
   add_bst(&root,1);
   add_bst(&root,4);
   add_bst(&root,7);
   add_bst(&root,6);
   add_bst(&root,8);
   printTreeInOrder(root);
   printLevelOrder(root);
   return 0;
}
```

If you run your program, you should see:

```
1
3
4
5
6
7
8
5 3 7 1 4 6 8
```

Do you see how these two functions can work to help you debug whether you
have properly implemented things or not?

B. ASSIGNMENT

I. TreeSort
The tree sort algorithm has a very straightforward premise:
to sort a set of numbers, you merely read them into a bst and then
print out the in order traversal of the bst.

Write treesort.c which will read integers from the user until EOF
(refer to getData.c from lab1). The integers should be added into your
bst using the add_bst function above.
After EOF, the program should print out the list in sorted order: one
int per line.

Compile via "gcc -pedantic treesort.c" and test it via
"cat numbers.txt | ./a.out > sorted.txt"

This means:
1. cat numbers.txt ... read out the contents of numbers.txt
2. | ./a.out ... pipe the output from the prior command to ./a.out
3. > sorted.txt ... redirect the output from the prior command to a file
   called sorted.txt

Clearly you have to create a file called numbers.txt and fill it with
random numbers for this to work.

```
Does your program work?

What is the Time Complexity of TreeSort? The Space Complexity?

Submit this via submit... 2 treesort.c

Don't print anything other than the sorted list of numbers, one int per line.

II. Rotations
Write the following in avlrot.c.
Write a function:
int isAVL(avlNode **root);
that will return 0 if the avl tree pointed to by root is an avl tree
or -1 if not.
Assume that you have to determine AVL-ness by calculating the Left and
Right subtree depths for all nodes --- i.e., no not assume that the
imbalance measure stored in each node is valid, determine the
AVL property by computing depths directly.

Write the function:
int rotate(avlNode **root,unsigned int Left0_Right1);
which will rotate the tree defined by root+pivot in the direction
defined by Left0Right1 (if 0, go left; if 1 go right).

If you're rotating Right, where is the pivot relative to the root?
If you're rotating Left, where is the pivot relative to the root?

Write:
int dblrotate(avlNode **root,unsigned int MajLMinR0_MajRMinL1);

If you're rotating MajorRightMinorLeft:
where is the pivot for the MinorLeft rotation relative to the root?
where is the pivot for the MajorRight rotation relative to the root
(after the MinorLeft rotation)?

Ditto for the MinorLeftMajorRight rotation.

Copy your add_bst funtion to avlrot.c and modify it to use your
avlNode struct instead of bstNode.

There should be no main function in avlrot.c.

Create some basic trees (in rotmain.c) using add_bst and demonstrate
to your self that your rotate functions do indeed work
(compile via: gcc -pedantic avlrot.c rotmain.c).
Create some trees that have the avl property and some that don't and confirm your
isAVL function works.

Submit avlrot.c and rotmain.c

III. AVL TREES
Sketch out the function:
int addAvl(avlNode **root,int val);

This will add using the bst ordering policy, and will update the balance
factors of the predecessor nodes (relative to the node you're adding) and the
node you're adding. If an imbalance is observed that exceeds the +-1 limit for
AVL trees, the appropriate rotation is selected.

Sketch out the selection logic.

Hint: after adding a new node to the tree, you will compute the balance measure
for all nodes starting at the node you added, right up to the root. When you
first observe an imbalance, fix it first via the correct rotation, and then
```

move on up towards the root, continuing your "measure-then-fix" scheme.

How would you accomplish this in a sane manner?

Suppose I said recursion is one such method, can you formulate/sketch
an algorithm for this operation?

Now suppose you wanted an alternative to recursion. What data structure
would you use to give you similar properties?

File this knowledge away for the exam.

Nothing to submit for this thought experiment.