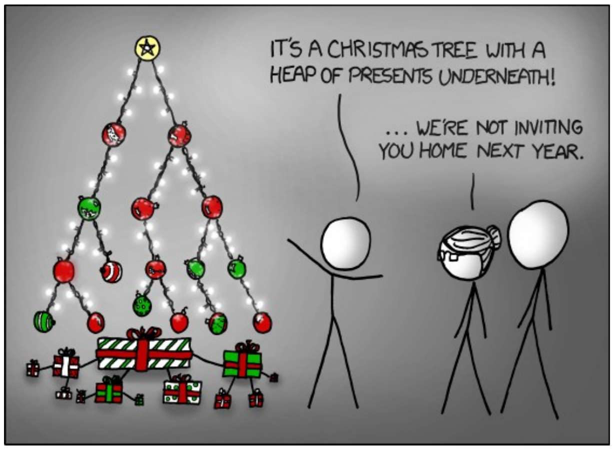
CS 302
Homework, Asst. #05

Purpose: Learn concepts regarding balanced binary trees.
Due: Tuesday (9/30) → Must be submitted on-line before class.
Points: Part A → 75 pts, Part B → 50 pts

Assignment:

Part A:
Design and implement a C++ template class, ***avlTree.h***, to implement an AVL Tree[1] data structure. A main will be provided that performs a series of tests using AVL tree with different data types.


IT'S A CHRISTMAS TREE WITH A HEAP OF PRESENTS UNDERNEATH!
... WE'RE NOT INVITING YOU HOME NEXT YEAR.
Source: http://xkcd.com/835/

Part B:
When completed, create and submit a write-up (open document, word or PDF format) not too exceed 200 words including the following:

- Name, Assignment, Section
- Summary of the AVL tree data structure.
- Compare using an AVL tree to a binary search tree.
  - Include advantages and disadvantages of each implementation approach.
- Big-O for the various tree operations.

Visualization
The following web site provides a visualization for AVL Trees, including the rotate operations.
https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

Make File:
You will need to develop a make file. You should be able to type:

```
make
```

Which should create the executables.

Submission:
- Submit a compressed zip file of the program source files, header files, and makefile via the on-line submission by 23:50.

All necessary files must be included in the ZIP file. The grader will download, uncompress, and type **make** (so you must have a valid, working *makefile*).

1 For more information, refer to: http://en.wikipedia.org/wiki/AVL_tree

Class Descriptions

- AVL Tree Class
  The AVL tree template stack class will implement functions specified below.
  We will use the following node structure definition.

```
template <class myType>
struct nodeType {
      myType      keyValue;
      int         nodeHeight;
      nodeType<myType>  *left;
      nodeType<myType>  *right;
};
```

Additionally, include the following enumeration definition:

```
enum  treeTraversalOptions { INORDER, PREORDER,
            POSTORDER, LEVELORDER, NONE };
```

avlTree<myType>
-nodeType<myType> *root
+avlTree()
+~avlTree()
+destroyTree(): void
+countNodes() const: int
+height() const: int
+search(myType) const: bool
+printTree(treeTraversalOptions) const: void
+insert(myType): void
+deleteNode(myType): void
-destroyTree(nodeType<myType> *): void
-countNodes(nodeType<myType> *) const: int
-height(nodeType<myType> *) const: int
-search(myType, nodeType<myType> *) const: nodeType<myType> *
-printTree(nodeType<myType> *, treeTraversalOptions) const: void
-printGivenLevel(nodeType<myType> *, int) const: void
-insert(myType, nodeType<myType> *): nodeType<myType> *
-rightRotate(nodeType<myType> *): nodeType<myType> *
-leftRotate(nodeType<myType> *): nodeType<myType> *
-getBalance(nodeType<myType> *) const: int
-deleteNode(myType, nodeType<myType> *): nodeType<myType> *
-minValueNode(nodeType<myType> *) const: nodeType<myType> *

Function Descriptions

- The *avlTree()* constructor should initialize the tree to an empty state.
- The ~*avlTree()* destructor should delete the tree by calling the private *destroyTree()* function.
- The public *destroyTree()* function should delete the tree by calling the private *destroyTree()* function.
- The private *destroyTree()* function should delete the tree (including releasing all the allocated memory).
- The public *countNodes()* function should return the total count of nodes in the tree by calling the private *countNodes()* function.
- The private *countNodes()* function should recursively return the total count of nodes in the tree. Must be recursive.
- The public *height()* function should return the maximum height of the tree by calling the private *height()* function.
- The private *height()* function should recursively return maximum height of the tree. Must be recursive.
- The public *search()* function should call the private *search()* function to determine if the passed node key is in the tree. If the node if found, the function should return true and return false otherwise.
- The private *search()* function should recursively search the tree for the passed node key. Must be recursive.
- The public *printTree()* function should call the private *printTree()* function to print the tree in the order passed.
- The private *printTree()* function should recursively print the tree in the specified order. Must be recursive. *Note*, the LEVELORDER option calls the *printGivenLevel()* function which performs recursion for that specific print option.
- The private *printGivenLevel()* function should print all nodes the passed level. Finding the nodes at the passed level should be performed recursively.
- The public *insert()* function should call the private *insert()* function to insert the passed key value into the tree. If the node is already in the tree, it should not be inserted again and no error message is required.
- The private *insert()* function should recursively insert the passed key value into the tree. The function will use the private *leftRotate()*, *rightRotate()*, and *getBalance()* functions.
- The public *deleteNode()* function should call the private *deleteNode()* function to delete the passed key value from the tree (if it exists). If the key does not exist, no error message is required.
- The private *deleteNode()* function should recursively delete the passed key value from the tree (if it exists). The function will use the private *leftRotate(), rightRotate(), getBalance()* functions, and *minValueNode()* functions.
- The *minValuenode()* function should search the tree starting from the passed node and return the node with the minimum key value. Does not need to be recursive. *Hint*, need only follow the left tree brnach.
- The private *getBalance()* function should return the balance factor (left subtree height – right subtree height) of the passed node.
- The private *rightRotate()* function should perform a right tree rotate operation (as described in class, in the lecture notes, and in the text).
- The public *leftRotate()* function should perform a left tree rotate operation (as described in class, in the lecture notes, and in the text).

Refer to the example executions for output formatting. Make sure your program includes the appropriate documentation. See Program Evaluation Criteria for CS 302 for additional information. ***Note, points will be deducted for especially poor style or inefficient coding.***

Level Order → Print Algorithm

A level order print of a binary tree can be performed multiple ways. Below is one possible approach.

```
// ------------------------------
// Calling routine:
//     level order traversal of tree

       for d = 1 to height(tree)
              printGivenLevel(tree, d);

// ------------------------------
// PrintGivenLevel function:
//     print all nodes at a given level

printGivenLevel(tree, currLvl)
       if tree is NULL then return;
       if currLvl is 1, then
              print(tree->data);
       else if currLvl greater than 1, then
              printGivenLevel(tree->left, currLvl-1);
              printGivenLevel(tree->right, currLvl-1);
```

Example Execution:

Below is an example program execution for the main.

```
ed-vm% ./main
-----------------------------------------------------------------
CS 302 - Assignment #5

-----------------------------------------------------------------
Test Set
       Nodes:  7
       Height: 3

In-order traversal:
5  6  8  10  11  14  18

Pre-order traversal:
10  6  5  8  14  11  18

Post-order traversal:
5  8  6  11  18  14  10

BFS traversal:
10
6 14
5 8 11 18

-----------------------------------------------------------------
Test Set
       Nodes:  24
       Height: 5

In-order traversal:
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22
23  24

Pre-order traversal:
16  8  4  2  1  3  6  5  7  12  10  9  11  14  13  15  20  18  17  19  22  21
23  24

Post-order traversal:
1  3  2  5  7  6  4  9  11  10  13  15  14  12  8  17  19  18  21  24  23  22
```

```
20  16

BFS traversal:
16
8 20
4 12 18 22
2 6 10 14 17 19 21 23
1 3 5 7 9 11 13 15 24

BFS traversal:
16
8 20
4 12 18 22
2 6 10 14 17 19 21 23
1 3 7 9 11 13 15 24

-----------------------------------------------------------------
Test Set
       Nodes:  15
       Height: 5

In-order traversal:
3  11  17  21  28  29  32  44  54  65  76  80  82  88  97

Pre-order traversal:
44  28  11  3  17  21  32  29  82  65  54  76  80  88  97

Post-order traversal:
3  21  17  11  29  32  28  54  80  76  65  97  88  82  44

BFS traversal:
44
28 82
11 32 65 88
3 17 29 54 76 97
21 80

Modified Tree:
       Nodes:  11
       Height: 4

BFS traversal:
44
21 88
11 29 76 97
3 32 54 80

-----------------------------------------------------------------
Test Set
       Nodes:  5000
       Height: 13

Modified Tree:
       Nodes:  37
       Height: 6

BFS traversal:
9955
15 9983
7 23 9967 9991
3 11 19 27 9959 9975 9987 9995
1 5 9 13 17 21 25 9957 9963 9971 9979 9985 9989 9993 9997
9961 9965 9969 9973 9977 9981 9999

-----------------------------------------------------------------
Game Over, thank you for playing.
ed-vm%
```