Program 2

Due Jul 15 by 11:59pm **Points** 20 **Submitting** a text entry box or a file upload
Available until Jul 16 at 12:30am

This assignment was locked Jul 16 at 12:30am.

Binary Search Tree

Purpose

This program is to create a **binary search tree** class called BinTree along with some additional functions (remove function is not required).

Description

Your code should be able to read a data file consisting of many lines (an example file called **inputdata.txt** will be given containing lines as shown below) to build binary search trees.

iii not tttt eee r not and jj r eee pp r sssss eee not tttt ooo ff m m y z $$

b a c b a c $$

c b a $$

Each line consists of many strings and terminates with the string “$$”. Each line will be used to build one tree. The duplicated strings (i.e., having equal values with existing node data) are discarded, smaller strings go left, and bigger go right. For instance, the internal tree built from the first line of **inputdata.txt** should look like in Figure 1.

```
                 iii
              /       \
          eee            not
         /   \         /     \
     and       ff    jj        tttt
                       \      /   \
                        m    r      y
                            / \      \
                          pp   sssss   z
                          /
                       ooo
```

Figure 1. example binary search tree

You will also be given **nodedata.h** and **nodedata.cpp** files which implements a NodeData class. You must define your tree nodes using NodeData (i.e., each node holds a NodeData* for the data).

Develop the class:

1. **A default constructor** (creates an empty tree), **a copy constructor** (deep copy), and **destructor**.

2. **Operators:**

   (a) The assignment operator (**=**) to assign one tree to another.

   (b) The equality and inequality operators (**==**, **!=**). Define two trees to be equal if they have the same data and same structure. For example,

```
T1:  b      T2:  b      T3:  c    T4:   b        T1 == T2
    / \         / \          /         / \       T1 != T3 != T4
   a   c       a   c        b         c   a      (Note that T4 is not a Bin Search Tree)
                           /
                          a
```

3. Accessors:

(a) Overrrid **<<** to display the tree using inorder traversal. The NodeData class is responsible for displaying its own data. You can see the example output format in the **output.txt**

(b) retrieve function to get the NodeData* of a given object in the tree (via pass-by-reference parameter) and to report whether the object is found (true or false).

bool **retrieve**(const NodeData &, NodeData* &);

The 2nd parameter in the function signature may initially be garbage. Then if the object is found, it will point to the actual object in the tree.

(c) getHeight function to find the height of a given value in the tree. SPECIAL INSTRUCTION: For this function only, you do <u>not</u> get to know that the tree is a binary search tree. You <u>must</u> solve the problem for a general binary tree where data could be stored anywhere (e.g., tree T4 above). Use this height definition: the height of a node **at a leaf is 1**, height of a node at the next level is 2, and so on. The height of a value not found is zero.

int getHeight (const NodeData &) const;

4. **Others:**

(a)bstreeToArray function to fill an array of Nodedata* by using an inorder traversal of the tree. It leaves the tree empty. (Since this is just for practice, assume a statically allocated array of 100 NULL elements. No size error checking necessary.)

void bstreeToArray(NodeData* []);

After the call to **bstreeToArray**, the tree in Figure 1 should be empty and the array should be filled with

and, eee, ff, iii, jj, m, not, ooo, pp, r, sssss, tttt, y, z (in this order)

(b) arrayToBSTree function to builds a **balanced BinTree** from a sorted array of NodeData* leaving the array filled with NULLs. The root (recursively) is at (low+high)/2 where low is the lowest subscript of the array range and high is the highest.

void arrayToBSTree(NodeData* []);

After the call to **arrayToBSTree,** the array in figure 2 should be filled with NULLs and the tree built should look like:

```
             not
          /       \
      ff             sssss
    /    \          /     \
and       jj      pp        y
   \     /  \    /  \      /  \
   eee iii   m ooo   r tttt    z
```

Figure 2. bstreeToArray result for the example tree

List of supporting files

1. **inputdata.txt**: input data file;

2. nodedata.h and nodedata.cpp: NodeData class;
3. driver.cpp: containing main(), to help clarify the functional requirements;
4. output.txt: correct output in using driver.cpp;
5. classAndSideway.txt: structure of BinTree class and 2 functions for helping display a binary tree as if you are viewing it from the side. *NOTE:* They will be part of your program. You can adjust this definition to your implementation (although you must have the three pointers in the Node and it must work with driver.cpp).

Submission Requirements:

Submit the followings:

1.output file

2. your own driver.cpp

3. bintree.h (add comments)

4. bintree.cpp (implement and add comments)

All the rules, submission requirements, and evaluation criteria are the same as the program assignment 1.

==

Grading guide

```
See Grading Rubric for grading information.
See Coding Standards for coding, documentation, and style guidelines.

1. Documentation (2pts): will grade your output.txt and driver file
   BSTs are correctly printed as specified  (1 pt)
   driver file test all methods and operators (1pt)

2. Correctness (16 pts)
   Successful compilation (2 pts)
+ correct retrieve ( 2 pt)
+ correct insert (1 pt)
+ correct constructor/destructor ( 1 pt)
+ correct operator for printing (<<) (1 pt)
+ correct implementation of the copy instructor, = (1 pts)
+ correct implementation of ==, != operators (2 pts)
+ correct implementation of getHeight (2 pts)
+ correct implementation of arrayToBST (2 pts)
+ correct implementation of bstToArray (1 pts)
+ no memory leak (check with valgrind) (1 pt)

3. Program Organization (2pts)
   Write comments to help the professor or the grader understand your operations. (see Coding Standards)
   Proper comments
   Good (1pt)      Poor/No comments(0 pts)
   Coding style (proper identations, blank lines, variable names, and non- redundant code)
   Good (1pt)      Poor(0 pts)
```