Assignment 2

Part 1 (7 pts). Programming

Description:

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

Your code should be able to read a data file consisting of many lines (an example file called **data2.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 data2.txt should look like in Figure 1.

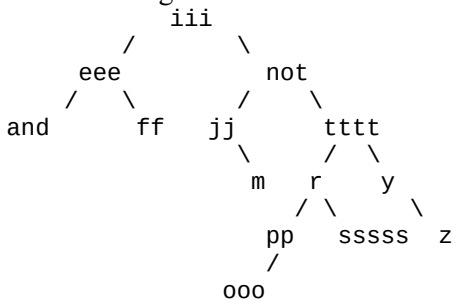


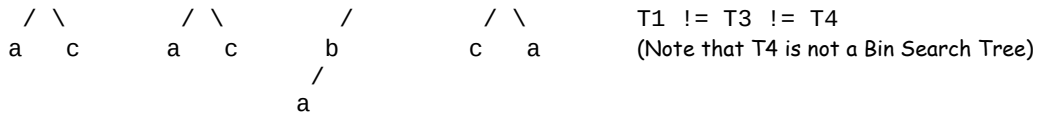
Figure 1. example binary search tree

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. Overload 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
```



3. Accessors:

(a). Overload << to display the tree using inorder traversal. The NodeData class is responsible for displaying its own data.

(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 not get to know that the tree is a binary search tree. You must 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

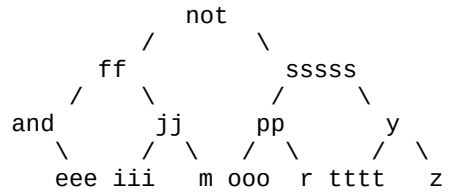
`and, eee, ff, iii, jj, m, not, ooo, pp, r, sssss, tttt, y, z` (in this order)

Figure 2. bstreeToArray result for the example tree

(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:



List of supporting files

1. **data2.txt:** input data file;
2. **nodedata.h** and **nodedata.cpp**: NodeData class;
3. **lab2.cpp**: containing main(), to help clarify the functional requirements;
4. **lab2output**: correct output in using **lab2.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 **lab2.cpp**).

Submission Requirements:
All the rules, submission requirements, and evaluation criteria are the same as the assignment 1. The only difference is that Valgrind is required in this assignment for 1 point total grade. Part of the point will be deducted for "definitely lost" memory leak.

Part 2 (2 pts). Use as many 8½ by 11 inch pieces of paper to make your work clear. Scan or take pictures and upload to canvas.

1. Use Huffman coding to encode these symbols with given frequencies:
   a: 0.10 b: 0.25 c: 0.05 d: 0.15 e: 0.30 f: 0.07 g: 0.08

2. Build (draw) a heap (minheap, smallest value at root) by inserting one item at a time with the values:
   15 20 3 4 8 10 9 5 14 2 25

3. Build (draw) a heap (minheap, smallest value at root) using the linear algorithm with the values:
   10 2 8 6 20 3 5 12 4 10 15