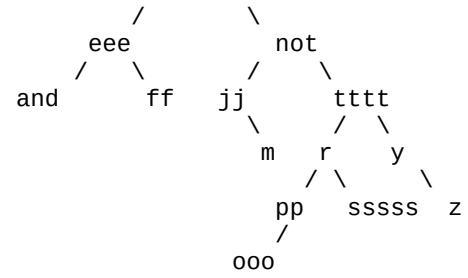
Part 1, Programming

Create a binary search tree class (no remove) called BinTree along with some additional functions. To test your tree class, a data file consisting of many lines is used to build binary trees. One line, which consists of many strings, will be used to build one tree. Each line is terminated with the string "$$" . The NodeData, holding a string, is stored once in the tree. Even though the object only contains a string, use a NodeData class so your tree class is not tied to any particular type of data beyond NodeData. You must use a tree node which holds a NodeData* for the data.

Sample input

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

The internal tree built would look like

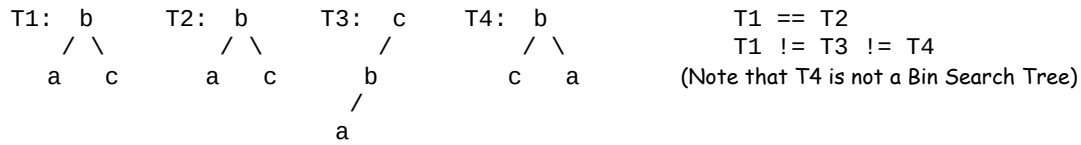


Lab requirements and output includes the following.

-- A default constructor (initially creates an empty tree), a copy constructor, and destructor (calls interface makeEmpty()).

-- Overloaded `operator=`, `operator==`, and `operator!=`. You would not use insert for the `copy` or `operator=`. Right?? Make sure that you understand that it is inefficient to use insert. Define two trees to be equal if they have the same data and same structure.

For example,



T1 == T2
T1 != T3 != T4
(Note that T4 is not a Bin Search Tree)

-- The `operator<<` will display as an inorder traversal. The NodeData class is responsible for displaying its data.

-- Retrieve a NodeData* of a given object in the tree (via pass-by-reference parameter). Return whether found. The second parameter may initially be garbage, will point to the actual object in the tree if it is found.

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

-- Find the depth of a given value in the tree. SPECIAL INSTRUCTIONS: 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 as the definition for depth: the depth of a node at the root is 1, depth of a node at the next level is 2, and so on. The depth of a value not found is zero.

```
int getDepth(const NodeData &) const;
```

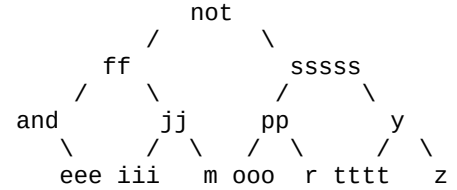
-- A routine fills an array of NodeData* by using an inorder traversal of the tree. It leaves the tree empty. (Since it is just for practice, assume a statically allocated array of 100 NULL elements. No size error checking necessary.)

```
void bstreeToArray(NodeData* []);
```

-- A routine builds a balanced BinTree from a sorted array (assumed) 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 bstreeToArray, the tree above is built into the following tree from a call to arrayToBSTree:



A sample driver file that uses the class and a sample text file can be found linked on the class web site. The NodeData class code is also linked from the website. You must use it unchanged.

BinTree class

```
class BinTree {              // you add class/method comments and assumptions
   ...
public:
   BinTree();                                // constructor
   BinTree(const BinTree &);                 // copy constructor
   ~BinTree();                               // destructor, calls makeEmpty
   bool isEmpty() const;                     // true if tree is empty, otherwise false
   void makeEmpty();                         // make the tree empty so isEmpty returns true
   BinTree& operator=(const BinTree &);
   bool operator==(const BinTree &) const;
   bool operator!=(const BinTree &) const;
   bool insert(NodeData*);
   bool retrieve(const NodeData&, NodeData*&) const;
   void displaySideways() const;             // displays the tree sideways
   ...
private:
   struct Node {
      NodeData* data;                        // pointer to data object
      Node* left;                            // left subtree pointer
      Node* right;                           // right subtree pointer
   };
   Node* root;                               // root of the tree

   // utility functions
   void inorderHelper( ... ) const;
   void sideways(Node*, int) const;
   ...
};
```

Adjust this definition to your implementation (although you must have the three pointers in the Node and it must work with the given main from the website). Node could be a class and whether a struct or a class, it doesn't necessarily have to be nested in the BinTree class. If a class, it is acceptable for the BinTree to be a *friend* of Node.

Part 2, written, turn in a hardcopy.

(Use as many 8½ by 11 inch pieces of paper, one-sided, to make your work clear.)

1. Draw an execution tree for the getDepth("pp") routine for the following tree of data (using your code). Show all function calls and relevant information at each call. (Recall the getDepth is written for a general binary tree, not a binary search tree.)

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

2. Math text, section 11.2, problem 24 (only the find the encoding). Show all work. Here is the problem:

   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

3. Build (draw) a heap (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

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

You can find other practice problems (with solutions) studied linked off the course website on the *Course Notes & Practice Problems* page (eventually).