Section Handout #8

Problem 1: Extending Editor Buffer

Write the method `MoveToWordBegin` for each of the following versions of the buffer object:

a) array implementation.

b) stack implementation.

c) singly-linked list implementation.

This function should move the cursor to the beginning of the word the cursor is currently positioned in. If the cursor were between the 'e' and 'd' in "jumped":

```
T h e  q u i c k  b r o w n  f o x  j u m p e d  o v e r  t h e  l a z y
                                              ^
```

calling `MoveCursorToWordBegin` would position the cursor before the 'j'.

```
T h e  q u i c k  b r o w n  f o x  j u m p e d  o v e r  t h e  l a z y
                                           ^
```

Be careful to handle gracefully the cases where the cursor is already at the beginning of a word (by moving cursor to beginning of the previous word), or at the beginning of the buffer. What is the running time of this operation (in Big-O notation) for the three implementations? You may use the function isspace which takes a character and returns a boolean which is true if the character is a space, false otherwise.

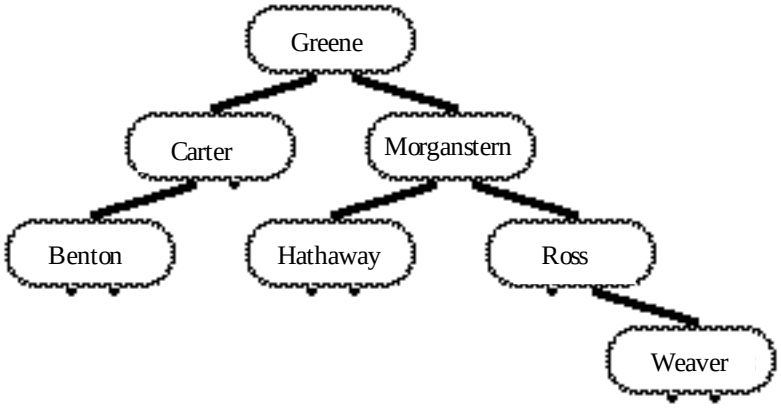
Problems 2-5:

For the following tree problems, assume that the following type definition provides the underlying structure for the tree:

```
struct nodeT
{
      string key;
      nodeT * left;
      nodeT * right;
};
```

Problem 2: Tree Trivia

Given the following binary search tree:



a) What is the height of this tree? Which nodes in this tree are the leaves? Who are the siblings of the node **`Morganstern`**?
b) What is the sequence of nodes visited on a pre-order tree traversal? in-order? post-order?
c) If we entered the node **`Del Amico`**, where would it end up in this tree?
d) What was the first node inserted into this tree, assuming that the tree hasn't been rebalanced?
e) Say we ran the following code on the above tree, where **`root`** points to the root node of the tree (i.e. **`Greene`**):

```
Queue<nodeT*> queue;
queue.enqueue(root);

while (!queue.isEmpty())
{
      nodeT *tree = queue.dequeue();
      cout << tree->key << endl;

      queue.enqueue(tree->left);
      queue.enqueue(tree->right);
}
```

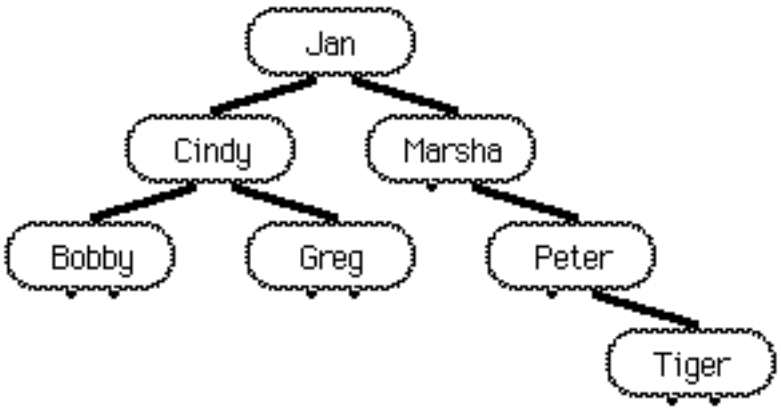
What is the sequence of nodes visited by this code? In what order have we essentially visited the tree?

Problem 3: Tree Equal

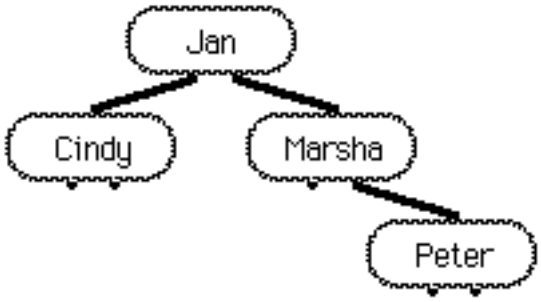
Write a function which takes two binary trees of type **`nodeT*`** as defined above and returns a boolean value which indicates whether the two trees are equal (i.e. have the same structure and values).

Problem 4: TrimLeaves

a) Write a function **`TrimLeaves(nodeT * & tree)`**, which will take a binary tree and remove all of its leaves. Thus, given the following tree:



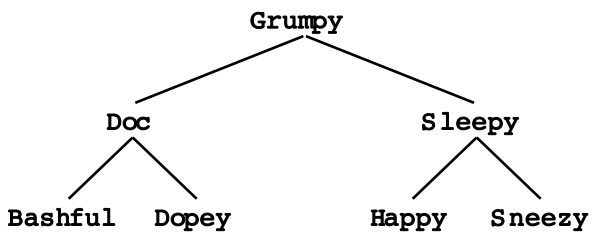
The function will produce the following tree:

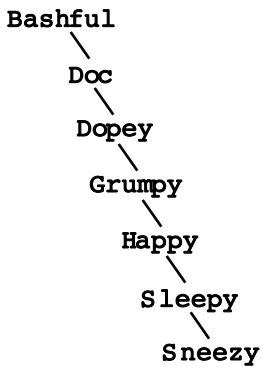


b) Note that the prototype of **`TrimLeaves`** as given passes tree by reference. What would happen if it was instead passed by value?

Problem 5: Balanced Trees

As we saw in class, the definition of a binary search tree does not in fact guarantee that the root node falls in the middle of the complete list of keys. For example, both of the following trees are binary search trees containing the names of the seven dwarves:



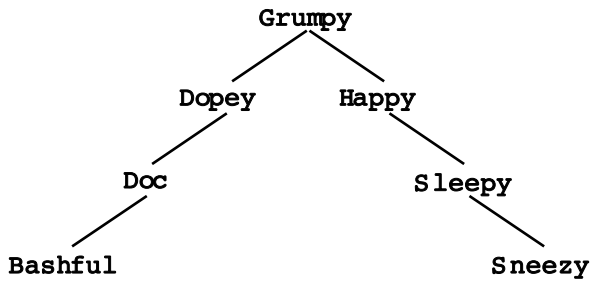


Only the tree on the left guarantees *O(log N)* search performance. The tree on the right is extremely unbalanced—it's essentially a linked list—and requires *O*(*N*) time to search.

A binary tree is **balanced** if each of the two following conditions is met:

1. The height of its left and right subtree differ by no more than one, where the **height** of a tree is defined to be the length of the longest path from the root to a leaf. Thus, the height of the tree appearing at the left of the example on the previous page is 3 while the height of the example at the right is 7.
2. Both of its left and right subtrees are themselves balanced.

Both conditions are important. For example, the tree shown below



is a legal binary search tree that meets the first condition for being balanced but not the second.

Write recursive implementations of the two functions:

```
int TreeHeight(nodeT *t);
bool IsBalanced(nodeT *t);
```

that return the height of a tree and whether it is balanced, respectively.