Section Solutions #8

Problem 1: Extending Editor Buffer

a) array implementation:

```
void Buffer::moveToWordBegin() {
      // if the character to the left of the cursor is a space
      // move back until it no longer is to skip over any
      // spaces that appear after the word
      while (cursor - 1 >= 0 && isspace(text[cursor - 1])) {
          moveCursorBackward();
      }

      // now move to the front of the word we are currently at -
      // meaning the letter to the left of the cursor is a space or
      // the cursor is at the beginning of the buffer
      while (cursor - 1 >= 0 && !isspace(text[cursor - 1])) {
          moveCursorBackward();
      }
}
```

Since we potentially have to go through every character in the buffer, this works out to O(n) time.

b) stack implementation:

```
void Buffer::moveToWordBegin() {
      // if the cursor is on a space,
      // move to the last character of the preceding word
      while (!before->IsEmpty() && isspace(before->peek())) {
            moveCursorBackward();
      }

      // move to the front of the word we're currently on
      while (!before->IsEmpty() && !isspace(before->peek())) {
            moveCursorBackward();
      }
}
```

Again, since we potentially have to go through every character in the buffer, this works out to O(n) time.

c) singly linked-list implementation:

```
/* moveToWordBegin
 * The function starts at the beginning of buffer and walks down the
 * list until it is pointing to the character just before the one the
 * cursor points to. As it walks down the list, it keeps track of the
```

```
 * last beginning of a word it saw. When it reaches the cursor, it
 * knows where the first word beginning to the left of the current word
 * is, and moves the cursor there. If there is no word beginning to the
 * left of the current word, the function moves the cursor to the
 * beginning of the buffer. If we had a doubly-linked list, we could
 * search backward from the cursor to find the first space. Given it is
 * singly-linked, we had to go from the head forward to find where we
 * were.
 */
void Buffer::moveToWordBegin() {
      cellT *wordStart, curr, prev;
      space = head;
      if (cursor != head) {
            curr = head->link;
            prev = head;
            while (curr != cursor)
            {
                  if (isspace(prev->ch) && !isspace(curr))
                        wordStart = curr;
                  prev = curr;
                  curr = curr->link;
            }
            cursor = wordStart;
      }
}
```

Since we potentially have to go through every character in the buffer, this works out to O(n) time.

Another, more inefficient, way to write this function would be to start at the cursor position and repeatedly call our `moveCursorBackwards` method, as in the previous stack and array implementation solutions. This version would be quicker and easier to write, and it would have the benefit of working without changes even if we drastically changed the underlying class implementation later. Of course, it comes at a cost of making the function run much less efficiently due to the repeated searches from the beginning required for each of the backward steps.

Problem 2: Tree Trivia

a) The height of the tree is 4 (just one node would be height 1). The leaf nodes are **Benton**, **Hathaway**, and **Weaver**, and the sibling of **Morganstern** is **Carter**.

b)

pre-order	in-order	post-order
Greene	**Benton**	**Benton**
Carter	**Carter**	**Carter**
Benton	**Greene**	**Hathaway**
Morganstern	**Hathaway**	**Weaver**
Hathaway	**Morganstern**	**Ross**
Ross	**Ross**	**Morganstern**
Weaver	**Weaver**	**Greene**

c) `Del Amico` would be the right child of `Carter`.

d) `Greene` was the first node inserted (since it's the root).

e) We will visit the nodes in the following order:

```
Greene, Carter, Morganstern, Benton, Hathaway, Ross, Weaver
```

Which amounts to doing a level by level traversal of the tree. Using the queue helps us achieve this. We start by just putting the root on the tree (the first level). We then visit it and place all of its children on the queue. Then because of the queue's FIFO property, we visit in the order things are placed on the queue, which causes us to visit the entire second level next. During this visit each node places all of its children on the tree (the third level), which we then visit next, and so on. Using a queue in this way then gives us the ability to visit the tree level by level. This is often referred to as a breadth first traversal of the tree, since we are visiting the nodes in order of increasing distance from the root.

Problem 3: Tree Equal

```
/**
 * Function: TreeEqual
 * Usage: if (TreeEqual(t1, t2)) …
 * ------------------------------
 * Function takes in two trees and returns whether they are equal (i.e.
 * have the same structure and values).
 */
bool TreeEqual(nodeT *t1, nodeT *t2)
{
      if ((t1 == NULL) && (t2 == NULL))
            return true;
      if ((t1 == NULL) || (t2 == NULL))
            return false;
      return ((t1->key == t2->key) &&
              TreeEqual(t1->left, t2->left) &&
              TreeEqual(t1->right, t2->right));
}
```

Problem 4: TrimLeaves

a)

```
/**
 * Function: IsLeaf
 * Usage: if (IsLeaf(node)) …
 * --------------------------
 * Function returns whether a tree is a leaf.
 * A tree is considered a leaf if its two children are
 * both NULL.
 */
bool IsLeaf(nodeT *node)
{
      return((node->left == NULL) && (node->right == NULL));
}

/**
 * Function: TrimLeaves
 * Usage: TrimLeaves(tree)
 * -----------------------
 * This function removes all of the leaves from the passed in binary
 * tree. The basic strategy is to recursively trim the trees.  Either
 * the current tree itself is a leaf, and we need to delete it from the
 * tree, or we need to recursively trim its left and right subtrees.
 */
void TrimLeaves(nodeT * & tree)
{
  if (tree != NULL)
  {
    if (IsLeaf(tree))
    {
      delete tree;
      tree = NULL;
    }
    else
    {
      TrimLeaves(tree->left);
      TrimLeaves(tree->right);
    }
  }
}
```

b)

If we did not pass tree by reference we would free all of the leaf node memory but the other nodes in the tree would still point to the freed memory. This would cause all of the usual problems with pointing to freed memory, and would also make the tree look like it still had the leaf nodes (because the pointers didn't get set to NULL).

Problem 5: Balanced trees

```
/**
 * Function: TreeHeight
 * Usage: TreeHeight(tree)
 * ----------------------------
 * Function returns the height of the passed in tree.
 */
int TreeHeight(nodeT *tree)
{
    if (tree == NULL)
    {
        return 0;
    }
    else
    {
      return (1 + max(TreeHeight(tree->left),
                      TreeHeight(tree->right)));
    }
}

/**
 * Function: IsBalanced
 * Usage: IsBalanced(tree)
 * ----------------------------
 * Function returns whether the passed in tree is balanced, meaning
 * that the height of its left and right subtree differ by no more
 * than one and that the left and right subtrees are themselves
 * balanced.
 */
bool IsBalanced(nodeT *tree)
{
    if (tree == NULL)
    {
      return true;
    }
    else
    {
      return((abs(TreeHeight(tree->left)-TreeHeight(tree->right)) <= 1)
              && IsBalanced(tree->left) && IsBalanced(tree->right));
    }
}
```