CS 106B Section 7 (Week 8)

This week is all about binary trees and hashing.

Recommended problems: #3, #5, #6, #7

---

1. Traversals.
Write the elements of each tree below in the order they would be seen by a pre-order, in-order, and post-order traversal

a)

```
        3
      /   \
    5       2
   /      /  \
  1      4    6
```

b)

```
          19
        /    \
      47       63
     /  \        \
   23    -2       94
         /          \
       55            28
```

c)

```
           2
             \
               1
             /   \
           7       6
          /          \
        4              9
       / \            /
      3   5          8
```

---

2. BST Insertion
Draw the binary search tree that would result from inserting the following elements in the given order.

a) Leia, Boba, Darth, R2D2, Han, Luke, Chewy, Jabba

b) Meg, Stewie, Peter, Joe, Lois, Brian, Quagmire, Cleveland

c) Kirk, Spock, Scotty, McCoy, Chekov, Uhuru, Sulu, Khaaaan!

```
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
    ...
};


class BinaryTree {
public:
    member functions;
private:
    TreeNode* root; // NULL if empty
};
```

*For each coding problem, you are to write a new public member function for the `BinaryTree` class from lecture that performs the given operation. You may define additional private functions to implement your public members. For functions that remove nodes, remember that you **must not leak memory**. You can assume that there is a helper function `deleteTree` that frees all memory associated with a given subtree.*

CS 106B Section 7 (Week 8)

3. height.

Write a member function called **height** that returns the height of a tree. The height is defined to be the number of levels (i.e., the number of nodes along the longest path from the root to a leaf). For example, an empty tree has height 0. A tree of one node has height 1. A node with one or two leaves as children has height 2, etc.

4. countLeftNodes.

Write a member function **countLeftNodes** that returns the number of left children in the tree. A left child is a node that appears as the root of the left-hand subtree of another node. For example, the tree in Problem 1 (a) above has 3 left children (the nodes storing the values 5, 1, and 4).

5. isBalanced.

Write a member function **isBalanced** that returns whether or not a binary tree is balanced. A tree is balanced if its left and right subtrees are *also balanced trees* whose heights differ by at most 1. The empty (NULL) tree is balanced by definition. You may call solutions to other section exercises to help you.

```
     balanced            balanced          not balanced        not balanced

         8                   4                    8                  4
       /   \               /   \                 /                 /   \
      4     9             3     9               4                 3     9
           / \           /                     / \               /       \
          4   6         1                     2   7             1         5
                                                                         /
                                                                        2
```

CS 106B Section 7 (Week 8)

6. removeLeaves.

Write a member function **removeLeaves** that removes the leaf nodes from a tree. A leaf is a node that has empty left and right subtrees. If a variable t refers to the tree below at left, the call of t.removeLeaves(); should remove the four leaves from the tree (the nodes with data 1, 4, 6 and 0). A second call would eliminate the two new leaves in the tree (the ones with data values 3 and 8). A third call would eliminate the one leaf with data value 9, and a fourth call would leave an empty tree because the previous tree was exactly one leaf node. If your function is called on an empty tree, it does not change the tree because there are no nodes of any kind (leaf or not). Free the memory for any removed nodes.

```
   Before call        After 1st call      After 2nd call     After 3rd call     After 4th call

        7                   7                 7                  7                 NULL
      /   \               /   \                \
     3     9             3     9                9
    / \   / \                   \
   1   4 6   8                   8
              \
               0
```

7. completeToLevel.

Write a member function **completeToLevel** that accepts an integer k as a parameter and adds nodes with value -1 to a tree so that the first k levels are complete. A level is complete if every possible node at that level is non-NULL. We will use the convention that the overall root is at level 1, its children are at level 2, and so on. Preserve any existing nodes in the tree. For example, if a variable called t refers to the tree below and you make the call of t.completeToLevel(3); you should fill in nodes to ensure that the first 3 levels are complete. Notice that level 4 of this tree is not complete. Keep in mind that you might need to fill in several different levels. You should throw an integer exception if passed a value for *k* that is less than 1.

```
          Before call                         After call

              17                                  17
            /    \                              /    \
          83       6                          83       6
         /          \                        /  \     /  \
       19            87                    19   -1  -1    87
         \          /                        \            /
          48      75                          48        75
```

CS 106B Section 7 (Week 8)

8. Pointer tracing.

Pointers to arrays are different in many ways from Vector or Map in how they interact with pass-by- value and the = operator. To better understand how they work, trace through the following program. What is its output?

```
void print(int* first, int* second) {
    for (int i = 0; i < 5; i++) {
       cout << i << ": " << first[i] << ", " << second[i] << endl;
    }
}

void transmogrify(int* first, int* second) {
    for (int i = 0; i < 5; i++) {
        first[i] = 137;
    }
}

void change(int* first, int* second) {
    first = new int[5];
    second = new int[5];
    for (int i = 0; i < 5; i++) {
       first[i] = second[i] = 271;
    }
}
```

```
int main() {
    int* one = new int[5];
    int* two = new int[5];
    for (int i = 0; i < 5; i++) {
        one[i] = i;
        two[i] = 10 * i;
    }

    print(one, two);
    transmogrify(one, two);
    print(one, two);
    change(one, two);
    print(one, two);

    delete[] one;
    delete[] two;
    return 0;
}
```

9. Hashing (part 1).

Let's say we have a class `StRiNg` where two `StRiNgs` are considered equal if they are equal, ignoring upper and lower case. Other than that, they are the same as normal strings. Which of the following functions are legal hash functions for `StRiNgs`? Which functions are *good* hash functions?

```
int hash1(StRiNg& s) {
  return 0;
}
```

```
int hash3(StRiNg& s) {
  int product = 1;
  for (int i = 0; i < s.length(); i++) {
    product *= tolower(s[i]);
  }
  return product;
}
```

```
int hash2(StRiNg& s) {
  int sum = 0;
  for (int i = 0;
       i < s.length(); i++) {
    sum += s[i];
  }
  return sum;
}
```

```
int hash4(StRiNg& s) {
  return (int) &s;
}
```

10. Hashing (part 2).

If our hash table has 6 buckets, diagram the result of putting the following values into the hash table, using a hash function that adds up the values of each letter in the string (where 'a' is 1, 'b' is 2, etc.) and mods by the hash table length (6). If two strings collide, put them into a linked list.

`cabbage, baggage, deadbeef, cafe, badcab, feed`