CS106B
Spring 2016 Cynthia Lee

Section 6 (Week 7) Handout

Problem and solution authors include Marty Stepp, Jerry Cain and Cynthia Lee.

Binary Tree Reference:

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

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

Binary Tree Member Functions

Problems 1 and 2 ask you to add a member function to the `BinaryTree` class from lecture. In all cases, if your function deletes a node from the tree, free the associated memory for the node.

1. **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 |                |                |                |
```

2. **tighten.** Write a member function tighten that eliminates branch nodes that have only one child. For example, if a variable *t* stores the tree below at left, the call of `t.tighten();` should leave *t* storing the tree at right. The nodes that stored 28, 19, 32, and -8 have been eliminated because each had one child. When a node is removed, it is replaced by its child. This can lead to multiple replacements because the child might itself be replaced (as in 19 which is replaced by 32, replaced by 72). Free memory as needed.

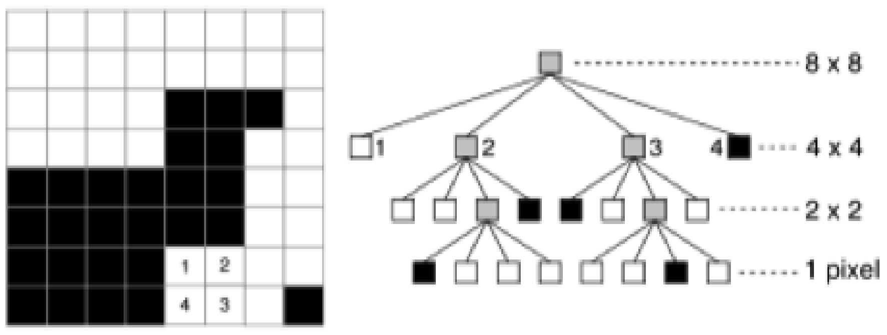
```
before call
           12
          /  \
        28    19
       /      /
     94      32
    /  \       \
  65   -8       72
         \     /  \
         10   42  50
```

```
after call
         12
        /  \
      94    72
     / \    / \
   65  10  42  50
```

3. **Quadtrees**

A quadtree is a rooted tree structure where each internal node has precisely four children. Every node in the tree represents a square, and if a node has children, each encodes one of that square's four quadrants.

Quadtrees have many applications in computer graphics, because they can be used as in-memory models of images. That they can be used as in-memory versions of black and white images is easily demonstrated via the following (borrowed from Wikipedia.org):



The 8 by 8 pixel image on the left is modeled by the quadtree on the right. Note that all leaf nodes are either black or white, and all internal nodes are shaded gray. The internal nodes are gray to reflect the fact that they contain both black **and** white pixels. When the pixels covered by a particular node are all the same color, the color is stored in the form of a Boolean and all four children are set to `NULL`. Otherwise, the node's sub-region is recursively subdivided into four sub-quadrants, each represented by one of four children.

Given a `Grid<bool>` representation of a black and white image, implement the `gridToQuadtree` function, which reads the image data, constructs the corresponding quadtree, and returns its root. Frame your implementation around the following data structure:

```
struct quadtree {
   int lowx, highx; // smallest and largest x value covered by node
   int lowy, highy; // smallest and largest y value covered by node
   bool isBlack; // entirely black? true.  Entirely white? False. Mixed? ignored
   quadtree *children[4]; // 0 is NW, 1 is NE, 2 is SE, 3 is SW
};
```

Assume the lower left corner of the image is the origin, and further assume the image is square and that the dimension is a perfect power of two.

```
static quadtree *gridToQuadtree(Grid<bool>& image);
```

4. **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;
}
```

5. **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. Bonus: diagram the resulting bucket arrangement after rehashing it to have 12 buckets.

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