CS106B
Spring 2017

Chris Gregg

Section Handout #6

This week has practice with trees. When working on your solutions, remember that you must not leak memory. Assume the following structures been declared:

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

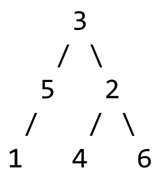
CharNode
```
struct CharNode {
  char ch;
  CharNode *left;
  CharNode *right;
};
```

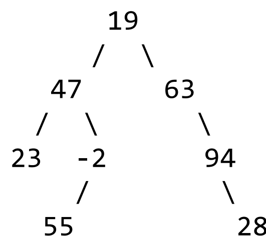
StringNode
```
struct StringNode {
  string str;
  StringNode *left;
  StringNode *right;
};
```

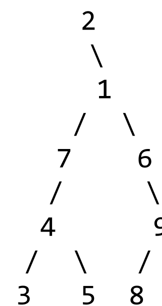
You can also assume that there is a helper function, `deleteTree`, that takes in the pointer to a tree and frees all the memory associated with that tree.

1. No, *You're* Out of Order

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







2. Height

Write a function that takes in the root of a tree of integers and returns the height of the tree. The height of a tree is defined to be the number of nodes along the longest path from the root to a leaf. An empty tree is defined to have height 0.

3. Count Left Nodes

Write a function that takes in the root of a tree of integers and returns the number of left children in the tree. A left child is a node that appears as the root of a left-hand subtree of another node. For example, the tree in problem 1(a) has 3 left children (the nodes containing 5, 1, and 4).

4. Balanced

Write a function that takes in the root of a tree of integers and returns whether or not the tree is balanced. A tree is balanced if its left and right subtrees are balanced trees whose heights differ by at most 1. The empty tree is defined to be balanced. You may call solutions to other section handouts to help you.

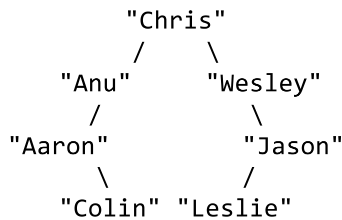
5. Prune a Tree

Write a function that takes in the root of a tree of integers and removes the tree's leaf nodes. Recall that a leaf is a node that has empty left and right subtrees. If your function is called on an empty tree, it should not change the tree. You must free the memory for any removed nodes.

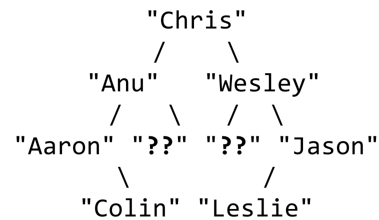
Thanks to Aaron Broder, Marty Stepp, Victoria Kirst, Jerry Cain, and other past CS106 instructors / TAs for contributing content on this handout. Thanks to Jason Chen and Leslie Tu for proofreading.

6. Complete to Level
Write a function that takes in the root of a tree of strings and an integer *k* and adds nodes with the value `"??"` to the tree so that the first *k* levels are complete. A level is defined to be complete if every possible node at that level is non-null. The root is defined to be level 1, its children are defined to be level 2, and so on. Preserve any existing nodes on the tree. You should throw an integer exception if passed a value for *k* that is less than 1.

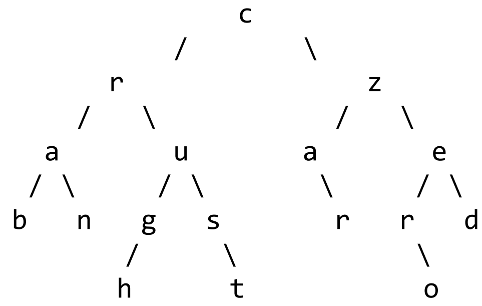
Before call to `completeToLevel(root, 3)`



After call



7. Word Trees
Write a function that takes in the root of a tree of characters and a string and returns whether or not that particular string can be found along any path from the root to the leaves of a given tree. For example, suppose you are given the following tree:

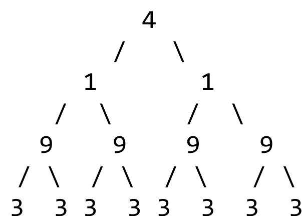


Your function would return true for `"crab"`, `"ran"`, `"rug"`, `"crust"`, `"rust"`, `"us"`, `"ugh"`, `"czar"`, `"zero"`, and `"zed"`. In particular, note that `"rug"` and `"us"` are completely internal – they neither start at the root nor end in the leaf nodes. You might also notice that the word `"arc"` exists in the tree, but your function shouldn't return true as it's going the wrong way along the path. Use backtracking recursion to prune the search space.

8. List to Tree
Write a function that takes in a singly-linked list of numbers and returns the root to a complete binary tree where the *n*th item of the singly-linked list occupies all positions at the *n*th level of the binary tree structure. For example, suppose you are given the following linked list:

```
4 -> 1 -> 9 -> 3
```

You should return the following binary tree:

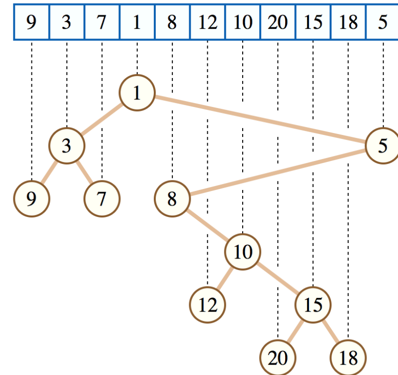


You can assume that you have the ListNode from last week's section handout. As a refresher, that structure was defined as follows:

```
struct ListNode {
  int data;
  ListNode *next;
}
```

9. Cartesian Trees

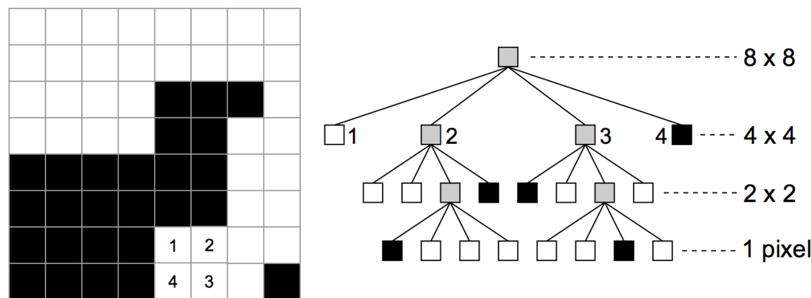
A Cartesian tree is a binary tree structure derived from an array of numbers such that the tree respects a min-heap property and an in-order traversal of the tree produces the original array sequence. The min-heap property means that the value of a parent is always less than both of its children. This diagram, courtesy of Wikipedia, illustrates the relationship between an array of numbers and the corresponding Cartesian tree.



Write a function that takes in a Vector of unique positive integers and returns the root of the corresponding Cartesian tree.

10. Quad Trees

A quadtree is a tree structure with applications in computer graphics, where they are used as in-memory models of images. In a quadtree, each internal node has precisely four children. Each node in the tree represents a square, and if a node has children, each encodes one of the square's four quadrants. This diagram, courtesy of Wikipedia, illustrates you might use a quad tree to represent a black-and-white image.



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 grey 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-subquadrants, each represented by one of four children.

Write a function that takes in a Grid of `booleans` (where a black pixel is `true` and a white pixel is `false`) and returns the root of the corresponding quadtree. Assume the lower left corner of the image is the origin, the image is square, and the dimension is a power of two. You should use the following structure for each node in the tree:

```
struct QuadTreeNode {
  int minX, maxX; // smallest and largest x value covered by node
  int minY, maxY; // smallest and largest y value covered by node
  bool isBlack;   // true if node is black; false if node is white; ignored otherwise
  QuadTreeNode *children[4]; // 0 is NW, 1 is NE, 2 is SE, 3 is SW
};
```