Nick Troccoli
CS 106X

Section Handout #6: Binary Trees

Based on handouts by various current and past CS106B/X instructors and TAs.

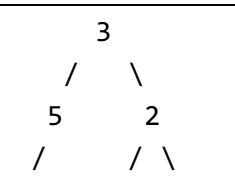
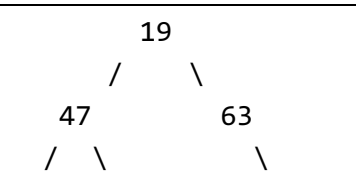
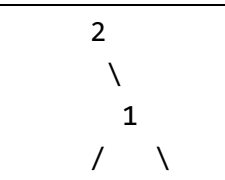
Extra practice problems: CodeStepByStep – height, isBalanced

This week is all about **binary trees**. The recursive structure of trees makes writing recursive functions very natural. In this handout, each function we write will accept a pointer to the root of a binary tree along with any other parameters needed. You may define additional helper functions as needed to implement your behavior. Remember that you **must not leak memory**. On this handout, we'll be using the following `TreeNode` structures:

`struct TreeNode {`
`  int data;`
`  TreeNode* left;`
`  TreeNode* right;`
`};`
	`struct TreeNodeChar {`
	`  char data;`
	`  TreeNodeChar* left;`
	`  TreeNodeChar* right;`
	`};`
		`struct TreeNodeString {`
		`  string data;`
		`  TreeNodeString* left;`
		`  TreeNodeString* right;`
		`};`

1. Traversals *(binary trees)*

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

a)	b)	c)
		

2. BST Insertion *(binary trees)*

Draw the binary search tree that would result from inserting the following elements in the given order. Here's the alphabet in case it's helpful: :-) **ABCDEFGHIJKLMNOPQRSTUVWXYZ**

a) Leia, Boba, Darth, R2D2, Han, Luke, Chewy, Jabba
b) Meg, Stewie, Peter, Joe, Lois, Brian, Quagmire, Cleveland
c) Peralta, Santiago, Doyle, Terry, Hitchcock, Scully, Holt, Diaz

3. countLeftNodes *(binary trees)*

Write a 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).

4. isBST *(binary trees)*

Write a function `isBST` that returns whether or not a binary tree is arranged in valid binary search tree (BST) order. Remember that a BST is a tree in which every node *n*'s left subtree is a BST that contains only values less than *n*'s data, and its right subtree is a BST that contains only values greater than *n*'s data.

Valid BST	Not a BST	Not a BST

`7`
`/ \`
`4 9`
`/ \`
`8 10`
	`4`
	`/ \`
	`3 9`
	`/ \`
	`6 8`
		`1`
		`/ \`
		`3 6`
		`/ \ / \`
		`2 4 5 7`

5. removeLeaves *(binary trees)*

Write a function `removeLeaves` that removes the leaf nodes from a tree. If `t` is the tree on the left, `removeLeaves(t);` should remove the four leaves from the tree (the nodes storing 1, 4, 6 and 0). A second call would eliminate the two new leaves in the tree (the nodes storing 3 and 8), and so on for further calls to this function. If your function is called on an empty tree, it should not change the tree. You must free the memory for any removed nodes.

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

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

6. completeToLevel *(binary trees)*

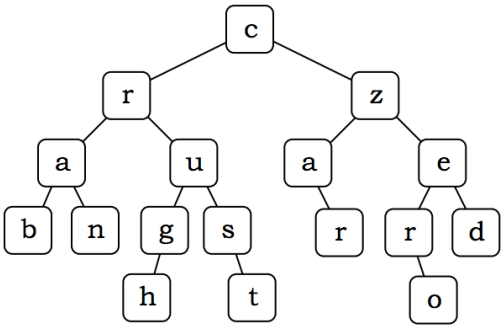
Write a function `completeToLevel` that accepts the root of a tree of strings and an integer *k* as parameters and adds nodes with value `"??"` to the tree so that the first *k* levels are complete. A level is considered *complete* if every possible node at that level is non-null. We will use the convention 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 `root` points to the root of the tree in each "before call" tree below and you call `completeToLevel(root, 3);` then the result should match the "after" pictures. If the *k* passed in is less than 1, then you should throw the invalid *k* parameter as an integer exception.

Before call	After call	Before call	After call

`"Aleksander"`
`/ \`
`"Jared" "Rachel"`
`/ \`
`"Evan" "Jack"`
`\ /`
`"Jennie" "Nolan"`
	`"Aleksander"`
	`/ \`
	`"Jared" "Rachel"`
	`/ \ / \`
	`"Evan" "??" "??" "Jack"`
	`\ /`
	`"Jennie" "Nolan"`
		`"sahami"`
		`\`
		`"stepp"`
			`"sahami"`
			`/ \`
			`"??" "stepp"`
			`/ \ / \`
			`"??" "??" "??" "??"`

7. wordExists *(binary trees)*



Write a function `wordExists` that determines whether a particular word can be found along a downward path of consecutive nodes in a given tree of characters. Your function should accept two parameters (a pointer to the root of a tree of chars and the string to search for), it should search from the root to the leaves of the tree, and it should return true when a path for the supplied text exists in the tree and false otherwise.

For example, if a pointer named `root` points to the root of the tree on the right, the call of `wordExists(root, "crust")` should return true because that path can be formed from the root downward to the "t" leaf. Note that the path need not start at the root, nor end at a leaf. For example, your function should also return true for this tree if passed "rug", "us", "crab", "ran", "rust", "ugh", "czar", "zero", "zed", and so on.

The call of `wordExists(root, "arc")` should return false because "arc" cannot be formed in the right order. Similarly, `wordExists(root, "cut")` should return false because the 'c', 'u', and 't' are not immediate neighbors in the tree.

Your code should return as soon as an answer can be determined. Prune your search and do not explore useless paths.