Binary tree

Binary Trees Page: 1 by Nick Parlante This article introduces the basic concepts of binary trees, and then works through a series of practice problems with solution code in C/C++ and Java. Binary trees have an elegant recursive pointer structure, so they are a good way to learn recursive pointer algorithms. Contents Section 1 . Binary Tree Structure a quick introduction to binary trees and the code that operates on them Section 2. Binary Tree Problems practice problems in increasing order of difficulty Section 3. C Solutions solution code to the problems for C and C++ programmers Section 4.

Java versions how binary trees work in Java, with solution code Stanford CS Education Library #110 This is article #110 in the Stanford CS Education Library. This and other free CS materials are available at the library (http://cslibrary. stanford. edu/). That people seeking education should have the opportunity to find it. This article may be used, reproduced, excerpted, or sold so long as this paragraph is clearly reproduced. Copyright 2000-2001 , Nick Parlante, nick.[email protected]stanford. edu. Related CSLibrary Articles Linked List Problems (http://cslibrary. stanford. du/105/) a large collection of linked ist problems using various pointer techniques (while this binary tree article concentrates on recursion) Pointer and Memory (http://cslibrary. stanford. edu/102/) basic concepts of pointers and memory The Great Tree-List Problem (http:// cslibrary. stanford. edu/109/) a great pointer recursion problem that uses both trees and lists Introduction To Binary Trees Section 1 A binary tree is made of nodes, where each node contains a " left" pointer, a " right" pointer, and a data element. The " root" pointer points to the topmost node in the tree.

The left and right pointers recursively point to smaller " subtrees" on either side. A null pointer represents a binary tree with no elements the empty tree. The formal recursive definition is: a binary tree is either empty (represented by a null pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a binary tree. http://cslibrary. stanford. edu/110/ BinaryTrees. html Page: 2 A " binary search tree" (BST) or " ordered binary tree" is a type of binary tree where the nodes are arranged in order: for each node, all elements in its left subtree are less-or-equal to the node 0.

The tree shown above is a binary search tree the " root" ode is a 5, and its left subtree nodes (1, 3, 4) are 5. Recursively, each of the subtrees must also obey the binary search tree constraint: in the (1, 3, 4) subtree, the 3 is the root, the 1 3. Watch out for the exact wording in the problems a " binary search tree" is different from a " binary tree". The nodes at the bottom edge of the tree have empty subtrees and are called " leaf" nodes (1, 4, 6) while the others are " internal" nodes (3, 5, 9). Binary Search Tree Niche Basically, binary search trees are fast at insert and lookup.

The next section presents the code for these two algorithms. On average, a binary search tree algorithm can locate a node in an N node tree in order lg(N) time (log base 2). Therefore, binary search trees are good for " dictionary" problems where the code inserts and looks up information indexed by some key. The lg(N) behavior is the average case it's possible for a particular tree to be much slower depending on its shape. Strategy Some of the problems in this article use plain binary trees, and some use binary search trees. In any case, the problems concentrate on the combination of pointers and recursion. See the articles linked above for pointer articles that do not mphasize recursion.) For each problem, there are two things to understand... The node/pointer structure that makes up the tree and the code that manipulates it The algorithm, typically recursive, that iterates over the tree When thinking about a binary tree problem, it's often a good idea to draw a few little trees to think about the various cases. http://cslibrary. stanford. edu/110/ BinaryTrees. html Page: 3 Typical Binary Tree Code in C/C++ As an introduction, we'll look at the code for the two most basic binary search tree operations lookup() and insert().

The code here works for C or C++. Java programers can read the discussion here, and then look at the Java versions in Section 4. In C or C ++, the binary tree is built with a node type like this... struct node { int data; struct node* left; struct node* right; } Lookup() the target. The basic pattern of the lookup() code occurs in many recursive tree algorithms: deal with the base case where the tree is empty, deal with the current node, and then use recursion to deal with the subtrees. If the tree is a binary search tree, there is often some sort of less-than test on the node to decide if the recursion should go left or right.

Given a binary tree, return true if a node with the target data is found in the tree. Recurs down the tree, chooses the left or right branch by comparing the target to each node. static int lookup(struct node* node, int target) { // 1. Base case = empty tree // in that case, the target is not found so return false if (node = NULL) { return(false); } else { // 2. see if found here if (target = node-; data) return(true); else { // 3. otherwise recur down the correct subtree if (target ; node- ; data) return(lookup(node-; left, target)); else return(lookup(node-; right, target)); } } }

The lookup() algorithm could be written as a while-loop that iterates down the tree. Our version uses recursion to help prepare you for the problems below that require recursion. Pointer Changing Code There is a common problem with pointer intensive code: what if a function needs to change one of the pointer parameters passed to it? For example, the insert() function below may want to change the root pointer. In C and C++, one solution uses pointers- to-pointers (aka " reference parameters").

That's a fine technique, but here we will use the simpler technique that a function that wishes to change a pointer passed to it ill return the new value of the pointer to the caller. The caller is responsible for using the new value. Suppose we have a changeo function http:// cslibrary. stanford. edu/110/ BinaryTrees. html Page: 4 that may change the the root, then a call to changeo will look like this... // suppose the variable " root" points to the tree root = change(root); We take the value returned by change(), and use it as the new value for root.

This construct is a little awkward, but it avoids using reference parameters which confuse some C and C++ programmers, and Java does not have reference parameters at all. This allows us to ocus on the recursion instead of the pointer mechanics. (For lots of problems that use reference parameters, see CSLibrary #105, Linked List Problems, http:// cslibrary. stanford. edu/105/). Insert() Insert() given a binary search tree and a number, insert a new node with the given number into the tree in the correct place. The insert() code is similar to lookup(), but with the complication that it modifies the tree structure.

As described above, insert() returns the new tree pointer to use to its caller. Calling insert() with the number 5 on returns the tree… 2 / 1 / 5 The solution shown here introduces a newNode() helper unction that builds a single node. The base-case/recursion structure is similar to the structure in lookup() each call checks for the NULL case, looks at the node at hand, and then recurs down the left or right subtree if needed. Helper function that allocates a new node with the given data and NULL left and right pointers. truct node* NewNode(int data) { struct node* node = new(struct node); // " new" is like " malloc" node-> data = data; node-> left = NULL; node-> right = NULL; return(node); } 10 Give a binary search tree and a number, inserts a new node with the given number in the correct place in the tree. Returns the new root pointer which the caller should then use (the standard trick to avoid using reference parameters). struct node* insert(struct node* node, int data) { http://cslibrary. stanford. edu/110/ BinaryTrees. html Page: 5 // 1. If the tree is empty, return a new, single node if (node = NULL)
{ return(newNode(data)); } else { // 2.

Otherwise, recur down the tree if (data data) node-> left = insert(node-> left, data); else node-> right = insert(node-> right, data); return(node); // return the (unchanged) node pointer } } The shape of a binary tree depends very much on the order that the nodes are nserted. In particular, if the nodes are inserted in increasing order (1, 2, 3, 4), the tree nodes Just grow to the right leading to a linked list shape where all the left pointers are NULL. A similar thing happens if the nodes are inserted in decreasing order (4, 3, 2, 1). The linked list shape defeats the lg(N) performance.

We will not address that issue here, instead focusing on pointers and recursion. Section 2 Binary Tree Problems Here are 14 binary tree problems in increasing order of difficulty. Some of the problems operate on binary search trees (aka " ordered binary trees") while others ork on plain binary trees with no special ordering. The next section, Section 3, shows the solution code in C/C*+. Section 4 gives the background and solution code in Java. The basic structure and recursion of the solution code is the same in both languages the differences are superficial.

Reading about a data structure is a fine introduction, but at some point the only way to learn is to actually try to solve some problems starting with a blank sheet of paper. To get the most out of these problems, you should at least attempt to solve them before looking at the solution. Even if your olution is not quite right, you will be building up the right skills. With any pointer- based code, it's a good idea to make memory drawings of a a few simple cases to see how the algorithm should work. This is a very basic problem with a little pointer manipulation. You can skip this problem if you are already comfortable with pointers.) Write code that builds the following little 1-2-3 binary search tree... 2/ 1 3 Write the code in three different ways... a: by calling newNode() three times, and using three pointer variables b: by calling newNode() three times, and using only one ointer variable c: by calling insert() three times passing it the root pointer to build up the tree (In Java, write a buildl 230 method that operates on the receiver to change it to be the 1-2-3 tree with the given coding constraints. See

Section 4. struct node* buildi 230 { 2. size() Page: 6 This problem demonstrates simple binary tree traversal. Given a binary tree, count the number of nodes in the tree. int size(struct node* node) { 3. maxDepth() Given a binary tree, compute its " maxDepth" the number of nodes along the longest path from the root node down to the farthest leaf node. The maxDepth of the empty tree is O, the maxDepth of the tree on the first page is 3. int maxDepth(struct node* node) { 4. minValue() Given a non-empty binary search tree (an ordered binary tree), return the minimum data value found in that tree.

Note that it is not necessary to search the entire tree. A maxValue() function is structurally very similar to this function. This can be solved with recursion or with a simple while loop. int minValue(struct node* node) { 5. printTree() Given a binary search tree (aka an " ordered binary tree"), iterate over the nodes to print them out in increasing order. So the tree... 1 35 Produces the output " 1 23 4 5". This is known as an " inorder" traversal of the tree. Hint: For each node, the strategy is: recur left, print the node data, recur right. void printTree(struct node* node) { 6. rintpostorder() Given a binary tree, print out the nodes of the tree according to a bottom-up the node itself is printed, and each left subtree is printed before the right subtree. So the tree... Produces the output " 1 3 2 5 4". The description is complex, but the code is simple. This is the sort of bottom-up traversal that would be used, for example, to evaluate n expression tree where a node is an operation like '+' and its subtrees are, recursively, the two subexpressions for the http://cslibrary. stanford. edu/110/ page: 7 void printpostorder(struct node* node) { 7. asPathSum() We'll define a " root-to-leaf path" to be a sequence of nodes in a tree starting with the root node and proceeding downward to a leaf (a node with no children). We'll say that an empty tree contains no root-to-leaf paths. So for example, the following tree has exactly four root-to-leaf paths: 5/ 4/ 11 134 1 Root-to-leaf paths: path 1: 54 11 7 path 2: 54 11 2 path 3: 58 13 path 4: 584 1 For his problem, we will be concerned with the sum of the values of such a path for example, the sum of the values on the 5-4-11-7 path is 5+4+ 11 +7 = 27.

Given a binary tree and a sum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum. Return false if no such path can be found. (Thanks to Owen Astrachan for suggesting this problem.) int hasPathSum(struct node* node, int sum) { 8. printPaths() Given a binary tree, print out all of its root-to-leaf paths as defined above. This problem is a little harder than it looks, since the " path so far" needs to be ommunicated between the recursive calls.

Hint: In C, C++, and Java, probably the best solution is to create a recursive helper function printPathsRecur(node, int path[], int pathLen), where the path array communicates the sequence of nodes that led up to the current call. Alternately, the problem may be solved bottom-up, with each node returning its list of paths. This strategy works quite nicely in Lisp, since it can exploit the built in list and mapping primitives. (Thanks to Matthias Felleisen for suggesting this problem.) Given a binary tree, print out all of its root-to-leaf paths, one per line. oid printpaths(struct node* node) { 9. irror() Change a tree so that the roles of the left and right pointers are swapped at every node. So the tree... 4 / 2 5 / http://cslibrary. stanford. edu/110/ BinaryTrees. html Page: 8 3 is changed to... 4/ 5 2/3 1 The solution is short, but very recursive. As it happens, this can be accomplished without changing the root node pointer, so the return-the- new-root construct is not necessary. Alternately, if you do not want to change the tree nodes, you may construct and return a new mirror tree based on the original tree. void mirror(struct node* node) { 0. oubleTree() For each node in a binary search tree, create a new duplicate node, and insert the duplicate as the left child of the original node. The resulting tree should still be a binary search tree. So the tree... 2/ 1 3 is changed to... 2/23//1 3/1 As with the previous problem, this can be accomplished without changing the root node pointer. void doubleTree(struct node* node) { 1 1 . sameTree() Given two binary trees, return true if they are structurally identical they are made of nodes with the same values arranged in the same way. (Thanks to Julie Zelenski for suggesting this problem. nt sameTree(struct node* a, struct node* b) { 12. countTrees() This is not a binary tree programming problem in the ordinary sense it's more of a math/combinatorics recursion problem that happens to use binary trees. (Thanks to Jerry Cain for suggesting this problem.) Suppose you are building an N node binary search tree with the values 1 .. N. How many structurally different binary search trees are there that store those values? Write a recursive function that, given the number of distinct values, computes the number of structurally unique binary search trees that store those values.

For example, http://cslibrary. tanford. edu/110/ Page: 9 countTrees(4) should return 14, since there are 14 structurally unique binary search trees that store 1, 2, 3, and 4. The base case is easy, and the recursion is short but dense. Your code should not construct any actual trees; it's Just a counting problem. int countTrees(int numKeys) { This background is used by the next two problems: Given a plain binary tree, examine the tree to determine if it meets the requirement to be a binary search tree.