Section 1 Lesson 6

Section 1 lesson 6 tree

I Discuss new data structures, algorithms, or language features

We will look at the tree data structure for this lesson. They are good for keeping records in order as you add new records. Sorting is natural because of the way the tree is built and how it is traversed. Once again the data never moves; only the pointers change.

II Describe functions and modules – why

The program begins by commenting our source code then #includes the three libraries we need. Next build our now standard data structure with fields for age, salary, and seniority. Now we get to the tree's framework by defining its node structure. Instead of having pointers to previous and next nodes, this new structure has pointers to the left and to the right, as well as one to the data.

Our newData() and newNode() functions work together; the first is a helper function for the second. newData() allocates a new data node and fills it with randomly generated data. It is called by newNode() which allocates space for a tree node, links to the new data node, and sets both pointers to our standard guard value, NULL.

The printInorder() function is called by main() and prints the tree's data from the far left node to the last one, on the far right. There are three ways you can traverse a binary tree: pre order, post order, or in order. In order recurses down the left part of the tree, prints the node you started from, and then recurses down the right part. Pre order and post order simply change the order in which we call the recursion. Print the data first, and print the data last, respectively.

Insertion into the tree is handled next. treeInsert() takes a pointer to the root, and a pointer to the new node, as input. It recursively walks the tree until it finds the new node's correct location. The tree uses age as its criterion, to choose left or right, until it finds the correct location to insert the new data. By creating the tree you are also sorting your data as it arrives. This tree is always sorted (on the chosen criterion).

Our main() function defines a pointer to the root of the tree then seeds the random number generator. Starting with an empty tree, a for loop inserts 43 new employee nodes. Lastly, we print them all out in sorted order.

III Compile and run the code

- Open a terminal window
- Navigate to … S1L6_tree using cd (change directory in Linux or in Windows using MinGW)
- Type – **make** which will link and compile the application called tree (in Linux, in Windows it is called tree.exe)
- Type - **./tree** (in Linux) or **tree** (in Windows) to run the application
- Read tree.c in your editor to find how the code creates the application's output

IV Describe code line by line – how

Lines 1 through 13 are nothing new. The comment banner and your usual include libraries. Lines 15 through 26 create the data structure and the node structure for the tree. Our nodes are labelled left, right, and data.

Examine line 24 and remember the first data describes the structure type. The second data is a pointer to the problem's data structure. I thought of writing this differently to avoid the ambiguity then thought better of it. Data says exactly the right thing and the two 'versions' do not conflict with one another. I find it best to use exactly the right word for a variable, a function, or a #define statement. By using the best descriptive word possible you better understand what you are doing, while writing self-commenting code.

Our newData() helper function, lines 30 through 39, is the same as in previous lessons. It allocates the space required from the heap and fills its fields with randomly generated data. Now we get to the fun stuff – printInorder(). This function 'walks' our tree to provide a sorted list of output. There are three ways to traverse a tree: in order, pre order, or post order. The in order method prints the data from the left subtree, then from the present node, and then the data from the right subtree.

This function is recursive, which is a benefit of using a tree. You can write simple, readable code to do your work. Lines 54 and 55 handle the trivial case of an empty tree or subtree. Line 58 recurses down the leftmost subtree while lines 61 through 66 print the data in the current node. Lastly, line 69 recurses down the rightmost subtree.

Creating the tree is handled by the function treeInsert(). It takes a pointer to the root of the tree and a pointer to the node you wish to insert. Lines 75 through 79 handle the case when your tree is empty. Simply point your root pointer at your first node. Remember: lines 47 and 48 set the new node's left and right pointers both to NULL. So when you set root equal to myNode (line 78) both the left and

right paths are terminated properly. It is important that you set your guard values correctly, or else you will have errors caused by walking off of your tree into unknown memory space.

Lines 81 through 111 handle the more interesting case where you have something in the tree and want to add another node. Lines 82 through 84 set up a flag and two working pointers. The pointer prev points at the previous node as a reference. Now we use ptr to walk the list looking for the correct empty space for our new node. Line 89 is where we set prev to the current location ptr for use later at line 108 or 110. Line 91 asks a sorting question – in this case, is your new node's age less than the age in the node data we are looking at now. If it is smaller then we traverse the left hand subtree. If not then we proceed down the right subtree. Lines 77, 95, and 101 are used to make sure the function is working properly. When you are done testing simply comment them out or delete them. Line 96 sets the isLeft flag to true (1) so the new data is placed correctly. This flag is used by line 107, to direct where your new node will be attached. Lines 95 through 97 sent us down the leftmost path, while lines 102 to 103 walk the rightmost path. Lines 107 through 111 attach the new node at the correct point in the tree. Line 112 may seem odd but this is a recursive function. Each recursion needs to know where to start. Lines 87 through 105 find the correct open spot in the tree to add our new node. Lines 107 through 110 use the pointer prev and the flag isLeft to place the new node in the tree.

Next we get to main(). Lines 117 through 121 define a pointer and seed the random number generator. Setting the root pointer equal to NULL provides treeInsert() with an empty tree. Lines 123 through 126 build the tree with a for loop. The line 128 prints your tree in order to have sorted data without ever once calling a sort routine. See line 91 to determine the criterion for sorting the tree. Line 125 may need a little explanation. Start by looking inside the function treeInsert(). You pass it the root pointer, which is initially set to NULL, and the function newNode(), which creates a tree node, a data node, and fills the data node's fields. Once treeInsert() is done it returns the value of root. Root only changes once, on line 78, but you need to keep track of it nonetheless.

The function delTree(), lines 134 through 143 is provided for completeness but never used in our example. It recursively frees the nodes of the tree until it is empty. Since this is a trivial app the memory is freed when the OS exits. But for a real world application you will need to free memory as necessary during the life of the tree. When your done using the tree you would call delTree() to free the memory properly.

V Describe makefile line by line

The makefile chooses g++ as its compiler on line 3.

Lines 5 and 6 link tree.o to create tree (or tree.exe under Windows).

Lines 8 and 9 compile tree.c to create tree.o

Lines 11 through 13 clean up afterward.

VI Deeper

This lesson introduces a data structure with a built in sorting mechanism. The new structure is a called a tree. In this case a binary tree having left or right children for each parent node. I abstracted the data away from the data structure for speed, ease of coding, and modification. By separating the data from the data structure I can build tools which I can use repeatedly over the years. Once they are tested I can depend on them. The data node can be used in any structure necessary, while the data structure and its associated tools, are unaffected.

We define our data structure on line 15. This is what you would change in a real problem, leaving the tree structure alone. The data structure simply holds whatever it is you need to solve the problem. On line 22 comes the tree node structure. It includes pointers to each child node and a pointer to the data node. In line 24 you are creating a data struct pointer while in line 25 you are creating two node struct pointers. A little note here. You are creating a struct called node and in the struct you are creating pointers to the same data type. This seems circular but it is not. At least to the compiler. It sees line 22 followed by the opening brace on line 23 and puts struct node into its variable table. So when the compiler reads line 25 it already 'knows' struct node. This is the standard idiom for creating lists & trees.

On line 30 I declare a function. Another note. See struct data * newData()? The asterisk denoting pointer is separated from the following word. On line 32 you see struct data *myData; and the asterisk is attached to the word. Reason: line 30 is declaring a function which returns a pointer while line 32 is creating a pointer. I use this difference to remind me of what is happening. I am not sure of the different schools of thought on the subject but this is mine.

Anyhow, newData() allocates space for your data record then fills it with random data. We will get to reading and saving files next time. newNode() creates a new tree node and attaches the data you created with newData(). Notice how each function takes care of its own memory allocation. If I were doing any deleting in this app, I would have to make sure to free up the allocated space when I am done using it. As it is, I am letting the OS reclaim the space on its own when I exit the app. Not strictly kosher but OK during a teaching exercise. What should be done is a call to a recursive tree destroyer :

```
void delTree(struct node *root)
    {
    if (root != NULL)
        {
        delTree(root->left);
        delTree(root->right);
        free(root->data);
        free(root);
    }
```

That should do it. A recursive way to free up the tree memory when you're done with it. Notice how you free the data before you free the node. If you don't you will have a chunk of memory floating around saying it is being used. But you can't use it because you don't have a link to it. That chunk of

memory will float around until your app is killed. If you do this often enough there are big problems. Memory leaks can crash your system, be careful of them.

I will skip the next routine, printInorder(), for a moment until we build the tree. So jump to line 73, treeInsert() There are two main cases: an empty tree or a non-empty tree. The former is taken care of with root = myNode. The latter takes a little more work. Create a variable and two node pointers to walk the tree. Now you walk down the tree in the direction defined by your data; less than to the left, greater than or equal to moves to the right. Keep walking (the while loop does this) until you find an empty spot. The empty spot is defined with what is called a guard value. That is what all those NULLs are doing. When you create a new node it is not connected to anything. You manually connect it to the data and then from its parent (that is what the prev pointer is doing, pointing back to the parent of the new node). The NULL denotes the end of your search, you have found space to add something.

Once you have found an open spot you attach the parent to your new node. Returning root is necessary to give the calling program a pointer to the new and growing tree.

Next we come to main() starting on line 115. We create one pointer and one variable. The latter is only used for randomization. The pointer is to the root of your tree. I tend to think of trees growing up but you can conceptualize them up, down, or sideways depending on your problem of choice. On line 121 root is given its guard value NULL so we can create the first node.

Now it is time to go back to that ignored function on line 52, printInorder(). It is a good example of how to walk a tree. In this case it is inorder, as compared to preorder & postorder. The three ways to traverse a binary tree are: in order – left, node, right; pre order – node, left, right; and post order – left, right, node. Where I wrote node is where you do the work of your tree walk. Are you searching the tree? Or are you building the tree? For in order you could say recurse the left subtree, do work here, then recurse the right subtree. In our case, the work was to print what was at each node.

I normally don't use trees as dynamic structures. I just create a hybrid node which has pointers for both a linked list data structure and another set for a tree data structure. I use the tree data structure for searching and sorting; its divide and conquer technique speeds up both algorithms. You can have your data in sorted order as it is entered and you can keep it in its original order by walking the linked list structure. It is just a matter of setting all the pointers accurately. The really neat thing is the data never moves from where it was originally placed in memory. Only the links change, there is no exchange of the data itself.

VII Assignments

1. Add the code to create a linked list then fill it by walking the tree.
2. Convert the code to read from a data file instead of creating random data. This makes a more real world problem closer to your goal.
3. Change the sort criterion in the tree insertion routine.
4. Create a hybrid data structure which has links for both the binary tree data structure and for a linked list data as well. Write the code to input the data directly into the tree and then walk the list to determine the linked list pointers too.
5. Use the delTree function to clean up. It has added to the end of the source file. Even though our simple case cleans up on exit it is more polite to clean up after ourselves. Remember to declare the delTree function before you call it. If you forget the compiler will surely tell you about it.

VIII Links

http://cslibrary.stanford.edu/110/BinaryTrees.html

https://en.wikipedia.org/wiki/Binary_tree

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Trees/trees.html

https://www.geeksforgeeks.org/applications-of-tree-data-structure/

https://www.geeksforgeeks.org/binary-tree-data-structure/

https://www.programiz.com/dsa/tree-traversal

https://en.wikipedia.org/wiki/Tree_traversal

https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/