CPE 111 – Programming with Data Structures
International Sections - January 2021
Laboratory Exercise 5

Objective

This lab is intended to give you the opportunity to work with tree traversals. It also introduces the concept of function pointers as function arguments.

Instructions

1. Download all the files from the **demos/Lecture5** area of the website (or use the file **lecture5Demos.zip**). You will be modifying the program ***simpleBinaryTree.c*** and the corresponding header file ***simpleBinaryTree.h***. **Don't forget to change the header comments to add your name, nickname and student ID.**

2. Add a function to ***simpleBinaryTree.c*** and ***simpleBinaryTree.h*** that will print out the contents of all nodes in the tree in level order. Remember that a level order (breadth first) traversal is ***not*** recursive. Instead, it uses a queue to keep track of the tree nodes that have not yet been processed. **Hint: look at *insertNode()* and *insertItem()*** (which calls ***insertNode()***). These functions are used to **add** nodes in level order.

The function should have the following declaration.

```
/* Traverse the tree in level order (breadth first), printing the
 * contents of each node.
 * Argument
 *   pTree  -  Points to the tree to traverse.
 */
void printTreeBreadthFirst(TREE_T* pTree);
```

3. The test program ***simpleTreeTester.c*** already has a case for testing this function (case 6). Modify the code to call ***printTreeBreadthFirst*** rather than to print the message saying the function is not implemented. Test your function using several different trees, *including a situation where there is only a root node*.

4. Add a function to ***simpleBinaryTree.c*** and ***simpleBinaryTree.h*** that will print the data contents of all the leaf nodes in a tree. That is, this function should traverse the tree, and if a tree node is a leaf, print its data (otherwise, do nothing). The function should have the following declaration.

```
/* Traverse an existing tree and print the leaf node values
 * Argument
 *   pTree  -  Points to the tree to traverse.
 */
void printLeafNodes(TREE_T* pTree);
```

5. The test program ***simpleTreeTester.c*** already has a case for testing this function (case 7). Modify the code to call ***printLeafNodes*** rather than to print the message saying the function is not implemented. Test your function using several different trees, *including a situation where there is only a root node*.

6. Add a function to ***simpleBinaryTree.c*** and ***simpleBinaryTree.h*** that will print the maximum depth of a tree. The maximum depth is the length of the longest path from the root to any leaf. This function should have the following declaration.

```
/* Calculate the maximum depth of a tree.
 * Argument
 *   pTree  -  Points to the tree we want to process.
 * Returns the maximum depth of the tree (at least 1) or 0
 * if the tree has not been initialized.
 */
int calculateMaxDepth(TREE_T* pTree);
```

7. The test program ***simpleTreeTester.c*** already has a case for testing this function (case 8). Modify the code to call ***calculateMaxDepth*** rather than to print the message saying the function is not implemented. Test your function using several different trees, including a case where there is only a root node and one where the tree has not been initialized.

8. Upload ***simpleBinaryTree.c***, ***simpleBinaryTree.h***, and ***simpleTreeTester.c***.

Some hints:

- The ***printLeafNodes*** function needs to recursively traverse the tree. It does not matter what kind of traversal you use.
- The ***calculateMaxDepth*** function will need to keep some information between invocations of the recursive function. **DO NOT** use a global variable for this purpose.
- You may have to write some "private" functions in order to implement these public functions.

Extra Challenge!

If you finish the three tasks above, write a function called ***findParent(TREE_T *pTree, char * key)***. This function should traverse the tree looking for a node whose data matches the argument 'key' (since the data is a string, you can use **strcmp**). If it finds a tree node with the passed key value, it should return the key for that node's parent. Otherwise it should return NULL.

Pointers to functions

You might find the traversal functions in ***SimpleBinaryTree.c*** are a bit strange. For instance:

```
void traversePostOrder(NODE_T* pCurrent,void (*nodeFunction)(NODE_T* pNode));
```

In this function. the pointer ***pCurrent*** is obviously the current node in the tree. But what is the second argument?

```
   void (*nodeFunction)(NODE_T* pNode)
```

This is an example of an argument that is ***a pointer to a function***. Specifically, it means that as the second argument of the ***traversePreOrder*** function, you can pass ANY function that takes a single ***NODE_T**** data item as its argument and which has no return value (is a ***void*** function).

In **simpleBinaryTree.c**, we already have two examples of functions like this:

```
void printNodeData(NODE_T* pNode)
{
  printf("Node %p ==> %s\n", pNode, pNode->data);
}
```

and

```
void freeNode(NODE_T* pNode)
{
  free(pNode->data);  /* we allocated this, so we delete it */
  free(pNode);
}
```

For instance, to free all the data and nodes in the entire tree, we call ***traversePostOrder*** as follows:

```
      if (pTree->root != NULL)
          traversePostOrder(pTree->root,&freeNode);
```

The traversal calls the passed function every time it visits a node.

If you want to traverse the tree, but do something different at each node, all you have to do is create your own function that follows the same pattern. For instance, suppose you wanted to turn all the data in the tree to upper case:

```
void upcaseNodeData(NODE_T* pNode)
{
  int i;
  for (i=0; i < strlen(pNode->data); i++)
      pNode->data[i] = toupper(pNode->data[i]);
}
```

Then you could pass this function to one of the traversals (any of them would work):

```
traversePostOrder(pTree->root,&upcaseNodeData);
```

Using pointers to functions like this allows the programmer to write the traversal logic only once, then use it for many different purposes.