Tutorial n°7 - Working with pointers, binary trees, recursive functions, and application to dictionary construction

1 Binary Trees

We have seen in Labs4 how objects can be linked into lists. When an object contains two pointers to objects of the same type, structures can be created that are much more complicated than linked lists. Today, we'll look at one of the most basic and useful structures of this type: **binary trees**. Each of the objects in a binary tree contains two pointers, typically called $left$ and $right$. In addition to these pointers, of course, the nodes can contain other types of data.

For example, a binary tree of integers could be made up of objects of the following type:

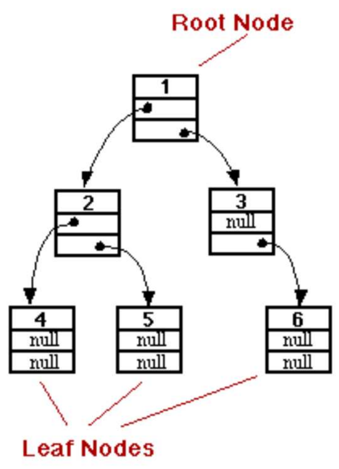
```
class TreeNode {
private:
        int item;         // The data in this node
        TreeNode *left;   // Pointer to the left subtree
        TreeNode *right;  // Pointer to the right subtree
        ...
};
```

The $left$ and $right$ pointers in a $TreeNode$ can be $NULL$ or can point to other objects of type $TreeNode$. A node that points to another node is said to be the parent of that node, and the node it points to is called a child. In the picture at the right, for example, node 3 is the parent of node 6, and nodes 4 and 5 are children of node 2. Not every linked structure made up of tree nodes is a binary tree.

A binary tree must have the following properties: There is exactly one node in the tree which has no parent. This node is called the **root** of the tree. Every other node in the tree has exactly one parent. Finally, there can be no loops in a binary tree. That is, it is not possible to follow a chain of pointers starting at some node and arriving back at the same node.

A node that has no children is called a **leaf**. A leaf node can be recognized by the fact that both the left and right pointers in the node are $NULL$. In the standard picture of a binary tree, the root node is shown at the top and the leaf nodes at the bottom.

Consider any node in a binary tree. Look at that node together with all its descendents (that is, its children, the children of its children, and so on). This set of nodes forms a **binary tree**, which is called a subtree of the original tree. For example, in the picture, nodes 2, 4, and 5 form a subtree. This subtree is called the left subtree of the root. Similarly, nodes 3 and 6 make up the right subtree of the root. We can consider any non-empty binary tree to be made up of a root node, a left subtree, and a right subtree. Either or both of the subtrees can be empty. This is a **recursive** definition, matching the recursive definition of the $TreeNode$ class. So it should not be a surprise that recursive functions are often used to process trees.

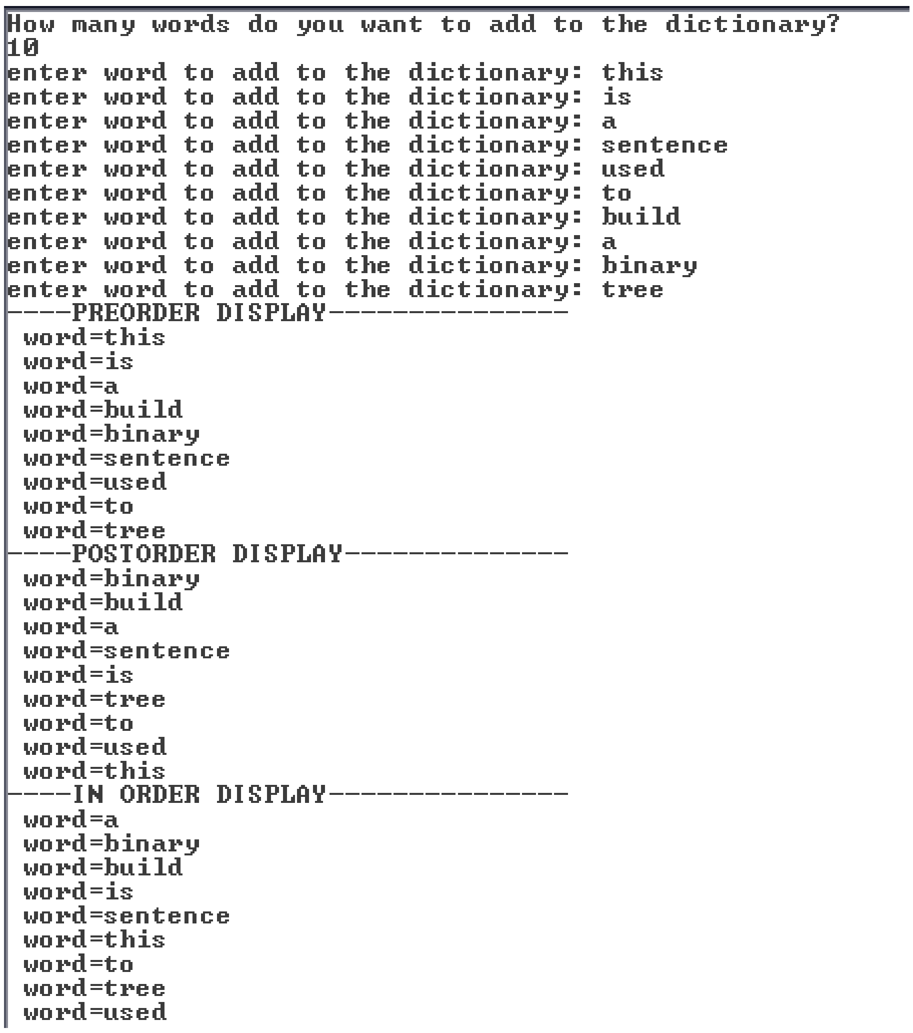


There exist 3 strategies to completely traverse a binary tree:

- The Pre-Order traversal: at each node the root is evaluated first, then the left sub tree, then the right subtree.
- The Post-Order traversal: the left subtree first, then the right subtree, then the root.
- The In Oder traversal: left, root, then right.

2 Application

Create a class Node to represent a binary tree. The data contained in each node will be of type *char*∗ to store words up to 255 characters. The program should first ask to the user to enter a given number of words and progressively build the boolean tree.
For each new word, the tree is constructed following a simple rule: it should represent a dictionary. In other word, the first word will always be the root. For the second word, if it is before (lexical order) the word contained in the root, it will be stored in the left child node, otherwise in the right node. The process goes on until all the words are stored in the tree.
Once the tree is built, the program should display the tree in Pre-Order, Post Order, and In-Order.
Here is a simple example where 10 words are read :



```
How many words do you want to add to the dictionary?
10
enter word to add to the dictionary: this
enter word to add to the dictionary: is
enter word to add to the dictionary: a
enter word to add to the dictionary: sentence
enter word to add to the dictionary: used
enter word to add to the dictionary: to
enter word to add to the dictionary: build
enter word to add to the dictionary: a
enter word to add to the dictionary: binary
enter word to add to the dictionary: tree
----PREORDER DISPLAY---------------
 word=this
 word=is
 word=a
 word=build
 word=binary
 word=sentence
 word=used
 word=to
 word=tree
----POSTORDER DISPLAY--------------
 word=binary
 word=build
 word=a
 word=sentence
 word=is
 word=tree
 word=to
 word=used
 word=this
----IN ORDER DISPLAY---------------
 word=a
 word=binary
 word=build
 word=is
 word=sentence
 word=this
 word=to
 word=tree
 word=used
```

One can remark that the recursive display using the In Order traversal corresponds to the lexical order.

String comparison

int strcmp (const char * str1, const char * str2);

Compare two strings

Compares the C string *str*1 to the C string *str*2. This function starts comparing the first character of each string. If they are equal to each other, it continues with the following pairs until the characters differ or until a terminating null-character is reached.

Parameters

*str*1 C string to be compared.
*str*2 C string to be compared.

Return Value

Returns an integral value indicating the relationship between the strings:

- A zero value indicates that both strings are equal.
- A value greater than zero indicates that the first character that does not match has a greater value in *str*1 than in *str*2; And a value less than zero indicates the opposite.