TRƯỜNG ĐẠI HỌC BÁCH KHOA TP.HCM

Khoa Khoa học & Kỹ thuật Máy tính

DATA STRUCTURES & ALGORITHMS

Lab session 3

RECURSION on BINARY TREE

1. OBJECTIVE

The objectives of Lab 3 are (1) to introduce an implementation of binary tree in C++ and (2) to practice recursion algorithms to manipulate a tree

2. FILE-LEVEL SEPARATION of INTERFACE and IMPLEMENTATION

Class interface and implementation

In Lab 2, we have learnt about the concept of separation between the interface and the implementation of a class. In practice, the separation is implemented as the file-level, i.e. we store the interface and the implementation in different files, whose extensions are respectively *.h* and *.cpp*. Listing 1 and Listing 2 illustrate the contents of two files, so-called *tree.h* and *tree.cpp*, corresponding to the interface and implementation of a binary tree. Please notice the red-colored statement #include "Tree.h" in the file *Tree.cpp*. It is very important to make the *Tree.cpp* "understand" what are declared in *Tree.h*. Please refer to manuals of the C language if you truly want to know the real meaning of the directive *#include*.

```
// this is the content of tree.h
template<class Node_entry>
struct TreeNode {
        Node_entry entry;
        TreeNode<Node_entry> *left, *right;
};

template <class Tree_entry>
class Tree  {
public:
        Tree();
        ~Tree();
```

```
protected:
        TreeNode<Tree_entry> *root;
        void destroy(TreeNode<Tree_entry> *);
};
```

Listing 1

```
// this is the content of tree.cpp
#include "Tree.h"

template<class Tree_entry>
Tree<Tree_entry>::Tree() {
        root = NULL;
}
//--------------------------------------------------------
template<class Tree_entry>
Tree<Tree_entry>::~Tree() {
        destroy(root);
        root = NULL;
}
//--------------------------------------------------------
template<class Tree_entry>
void Tree<Tree_entry>::destroy(TreeNode<Tree_entry> *root) {
        if (root != NULL) {
                destroy(root->left);
                destroy(root->right);
                delete root;
        }
}
```

Listing 2

3. RECURSION in BINARY TREE

Recursion is an unavoidable technique to handle many operations in a binary tree. In Listing 2, an example is given to illustrate how to use recursion to collect garbage when a tree is deleted. Basically, to construct an algorithm for a tree binary, we should do the following steps:

- Process the stop condition: think the simplest case (i.e. an empty) and what we should do in this case
- Process the recursion: assume that we had successfully done what we intended to do with the left and the right sub-trees, develop a way to combine the results to yield the desired purpose.

```
template<class Tree_entry>
int Tree<Tree_entry>::getSize () {
    return getSizeFrom(root);
}

template<class Tree_entry>
int Tree<Tree_entry>::getSizeFrom(TreeNode<Tree_entry> *subroot) {
    int nResult;
    //stop condition: what we should do for the simplest case – an empty tree
    if (subroot == NULL) nResult = 0;
    //recursive case: assume that we can count the size of left and right sub-trees
    // what should we do to get the final result?
    else nResult = getSizeFrom(subroot->left) + getSizeFrom(subroot->right) + 1;
    return nResult;
}
```

Listing 3

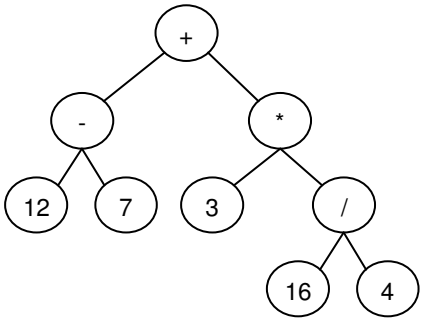
Listing 3 gives a scenario in which we try to develop a method *getSize*() to count the number of nodes of the tree. To fulfill this job, we implement another auxiliary method called *getSizeFrom*, which counts the number of nodes of a sub-tree whose root is a certain node. As you can see, *getSizeFrom* will be implemented in a recursive manner.

4. EXERCISES

Consider the files *main.cpp*, *Tree.h* and *TreeSample.cpp* attached. Use this initial code to accomplish the following tasks.

4.1. Using the provided method insertAt to build the following tree in the main program.

Print out the tree afterward.



4.2. Write methods to print the tree in LNR, LRN, NLR, NRL, RNL, and RLN.

4.3. Write a method to calculate the height of the tree.

4.4. Write a method to check if a tree is balanced such that at any node, the different between the height of the left sub-tree and the height of the right sub-tree is 0,-1, or 1.

4.5. Write a method to swap the left and the right sub-trees at any node.

4.6. Write a method to count the number of leaves in the tree.

4.7. Write a method to delete all leaves from the tree.

4.8. (somewhat advanced). Write methods to travel the tree in LNR and put all data in a linked list. The order of elements in the list should be the same with that of the result of the print method in LNR.

4.9. (advanced). Support that an expression can be presented as a string tree (each node is a node of string) as in question 4.1. In that expression tree, leaves are used to present operands, which are numbers. Other nodes on the tree are used to present operators (plus, minus, multiply, and divide). To convert a string (object of the class string) into an integer, one may use the function atoi. For example, the following prints 23 to the screen:

```
string s1 = "21";
int x = atoi(s1.c_str()); //x is 21
x = x+2;
cout << x;
```

Write a method named evaluate to calculate an expression expressed as a tree above. For example, the tree in question 4.1 should be evaluated to 17.

Hint: (1) write another auxiliary recursive function named evaluate_recur to recursively calculate an expression of a subtree; (2) you can also use the method build_tree_from_keyboard to build another expression tree from the keyboard to test your new method.

-- **End** --