how U doin'?

$$🐳^{🐳} = \int_{🎃}^{🎅} 🐑 \, d🍀$$

Tree*

10170437 Mark Taylor

July 3, 2020

Contents

1 Problems

1. 编写算法:输入树的边(双亲-孩子有序对)建立对应树的孩子-兄弟链表存储结构;
2. 编写算法:求树的深度;
3. 编写算法:输出从树根到所有叶子结点的路径;
4. 编写算法:统计树中叶子结点的个数。

2 Code

2.1 Tree.h

*This article was typeset by Mark Taylor using the LaTeX document processing system.

```cpp
// tree header
#ifndef TREE_H
#define TREE_H
#include "SeqStack.h"
#include <memory>
#include <cassert>
#include <iostream>

template<typename T>
struct TreeNode {
        T _data;
        std::shared_ptr<TreeNode<T>> _first_child,  _next_sibling;
        TreeNode(const T value = {}, std::shared_ptr<TreeNode<T>> fc = nullptr,
        ↪  std::shared_ptr<TreeNode<T>> ns = nullptr)
                : _data(value), _first_child(fc), _next_sibling(ns) {}
        ~TreeNode() { std::cout << "Destructing tree node with value " << _data << '\n'; }
};

template<typename T>
class Tree {
public:
        Tree() { _root = nullptr; }
        std::shared_ptr<TreeNode<T>> create_tree(std::shared_ptr<TreeNode<T>> t);

        std::shared_ptr<TreeNode<T>> get_root()const { return _root; }
        bool empty()const { return _root == nullptr; }
        size_t depth(std::shared_ptr<TreeNode<T>> t)const;
        size_t count_node(std::shared_ptr<TreeNode<T>> t)const;
        size_t count_leaves(std::shared_ptr<TreeNode<T>> t)const;

        // We do NOT set traverses as const member functions
        // given that we may need to modify those nodes.
        // see also <https://isocpp.org/wiki/faq/pointers-to-members>
        void preorder(std::shared_ptr<TreeNode<T>> t, void (*visit)(std::shared_ptr<TreeNode<T>>) = visit);
        void postorder(std::shared_ptr<TreeNode<T>> t, void (*visit)(std::shared_ptr<TreeNode<T>>) = visit);

        // print paths from root node to leaves
        void print_path(std::shared_ptr<TreeNode<T>> t)const;

private:
        std::shared_ptr<TreeNode<T>> _root;
        static void visit(std::shared_ptr<TreeNode<T>> t);
};

//create tree by level
template<>        // specialization for char type
std::shared_ptr<TreeNode<char>> Tree<char>::create_tree(std::shared_ptr<TreeNode<char>> t)
{
        // to be added..
        return std::make_shared<TreeNode<char>>();
}

template<typename T>
std::shared_ptr<TreeNode<T>> Tree<T>::create_tree(std::shared_ptr<TreeNode<T>> t)
{
        return std::make_shared<TreeNode<T>>();
}

template<typename T>
size_t Tree<T>::depth(std::shared_ptr<TreeNode<T>> t) const
{
```

```
61          if (t == nullptr) return 0;
62          size_t fc_depth = depth(t->_first_child) + 1;
63          size_t ns_depth = depth(t->_next_sibling);
64
65          return fc_depth > ns_depth ? fc_depth : ns_depth;
66  }
67
68  template<typename T>
69  size_t Tree<T>::count_node(std::shared_ptr<TreeNode<T>> t) const
70  {
71          if (t == nullptr) return 0;
72          size_t count = 1;
73          count += count_node(t->_first_child);
74          count += count_node(t->_next_sibling);
75
76          return count;
77  }
78
79  template<typename T>
80  size_t Tree<T>::count_leaves(std::shared_ptr<TreeNode<T>> t) const
81  {
82          static size_t count = 0;
83          if (t != nullptr) {
84                  if (t->_first_child == nullptr)
85                          ++count;
86                  else
87                          count_leaves(t->_first_child);
88
89                  count_leaves(t->_next_sibling);
90          }
91          return count;
92  }
93
94  template<typename T>
95  void Tree<T>::preorder(std::shared_ptr<TreeNode<T>> t, void(*visit)(std::shared_ptr<TreeNode<T>>))
96  {
97          if (t != nullptr) {
98                  visit(t);
99                  preorder(t->_first_child, visit);
100                 preorder(t->_next_sibling, visit);
101         }
102 }
103
104 template<typename T>
105 void Tree<T>::postorder(std::shared_ptr<TreeNode<T>> t, void(*visit)(std::shared_ptr<TreeNode<T>>))
106 {
107         if (t != nullptr) {
108                 postorder(t->_first_child, visit);
109                 visit(t);
110                 postorder(t->_next_sibling, visit);
111         }
112 }
113
114 template<typename T>
115 void Tree<T>::print_path(std::shared_ptr<TreeNode<T>> t) const
116 {
117         static SeqStack<T> s;
118         while (t != nullptr) {
119                 s.push(t->_data);
120                 if (t->_first_child == nullptr) {
121                         s.bottom_up_traverse();
```

```
122                         std::cout << '\n';
123                 }
124                 else
125                         print_path(t->_first_child);
126                 s.pop();
127                 t = t->_next_sibling;
128         }
129 }
130
131 // operation for node p
132 template<typename T>
133 inline void Tree<T>::visit(std::shared_ptr<TreeNode<T>> p)
134 {
135         assert(p != nullptr);
136         std::cout << p->_data;
137 }
138
139 #endif // !TREE_H
```

Listing 1: Tree header

2.2 Tree_test.cpp

```
1  #include "Tree.h"
2  #include <iostream>
3
4  using namespace std;
5
6  void tree_operations()
7  {
8          Tree<char> tree;
9          auto root{ tree.get_root() };
10         //tree.create_tree(root);        // to be added
11         /*
12                     A
13                   / | \
14                  B  C  D
15                  |     / \
16                  E     F  G
17         */
18         cout <<
19                 "            A\n"
20                 "          / | \\     \n"
21                 "         B  C  D     \n"
22                 "         |     / \\   \n"
23                 "         E     F   G   \n";
24
25         root = make_shared<TreeNode<char>>('A');
26         root->_first_child = make_shared<TreeNode<char>>('B');
27         root->_first_child->_first_child = make_shared<TreeNode<char>>('E');;
28         root->_first_child->_next_sibling = make_shared<TreeNode<char>>('C');
29         root->_first_child->_next_sibling->_next_sibling = make_shared<TreeNode<char>>('D');
30
31         root->_first_child->_next_sibling->_next_sibling
32                 ->_first_child = make_shared<TreeNode<char>>('F');
33
34         root->_first_child->_next_sibling->_next_sibling
```

```
                ->_first_child->_next_sibling = make_shared<TreeNode<char>>('G');

        cout << "All paths from root to leaves are as follows:\n";
        tree.print_path(root);
        cout << "number of tree nodes: " << tree.count_node(root) << '\n';
        cout << "number of tree leaves: " << tree.count_leaves(root) << '\n';
        cout << "depth:  " << tree.depth(root) << '\n';
        cout << "preorder: "; tree.preorder(root); cout << '\n';
        cout << "postorder: "; tree.postorder(root); cout << '\n';
}

int main()
{
        tree_operations();

        cout << "\n\nPress any key to leave...\n";
        char wait;
        cin >> noskipws >> wait;

        return 0;
}
```

Listing 2: Tree test

2.3 SeqStack.h

```
// sequential stack header
#pragma once
#ifndef SEQSTACK_H
#define SEQSTACK_H

#include "Stack.h"
#include <iostream>
#include <cassert>

const int defaultStackSize = 30;
const int stackIncreament = 20;                // let it double its capacity

template<typename T>
class SeqStack : public Stack<T> {
public:
        SeqStack(int sz = defaultStackSize);
        SeqStack(const SeqStack<T>& stack);
        SeqStack<T>& operator=(const SeqStack<T>& stack);
        ~SeqStack() { delete[] elem; }
        void push(const T& x);
        void pop();
        void pop(T& x);
        T& top()const;
        void clear() { _top = -1; }
        inline bool isEmpty()const { return _top == -1; }
        bool isFull()const { return _top == maxSize - 1; }
        int max_size()const { return maxSize; }
        inline int size()const { return _top + 1; }
        void bottom_up_traverse()const;

private:
```

```
32          T* elem;                                        // pointer to stack array
33          int _top;                                        // index of stack top: [0, size), when stack's
            ↪  empty, _top=-1
34          int maxSize;                                  // maximum volume of the stack
35          void overflowProcess();
36  };
37
38  template<typename T>
39  SeqStack<T>::SeqStack(int sz) {        // construct an empty stack with a size of sz
40          _top = -1;
41          maxSize = sz;
42          elem = new T[maxSize];
43          assert(elem != nullptr);
44  }
45
46  template<typename T>
47  SeqStack<T>::SeqStack(const SeqStack<T>& stack) { // copy constructor
48          _top = stack._top;
49          maxSize = stack.maxSize;
50          elem = new T[maxSize];
51          assert(elem != nullptr);
52          for (int i = 0; i <= _top; ++i) elem[i] = stack.elem[i];
53  }
54
55  template<typename T>
56  SeqStack<T>& SeqStack<T>::operator=(const SeqStack<T>& stack) { // copy assignment
57          if (&stack == this)        return *this;
58          delete[] elem;
59          _top = stack._top;
60          maxSize = stack.maxSize;
61          elem = new T[maxSize];
62          assert(elem != nullptr);
63          for (int i = 0; i <= _top; ++i) elem[i] = stack.elem[i];
64          return *this;
65  }
66
67  template<typename T>
68  void SeqStack<T>::push(const T& x){ // add a new element at the top of the stack
69          if (isFull()) overflowProcess();
70          elem[++_top] = x;
71  }
72
73  template<typename T>
74  void SeqStack<T>::pop() { // pop out the top element of the stack
75          assert(!isEmpty());
76          --_top;
77  }
78
79  template<typename T>
80  void SeqStack<T>::pop(T& x) { // pop out the top element of the stack and assign it to x
81          assert(!isEmpty());
82          x = elem[_top--];
83  }
84
85  template<typename T>
86  T& SeqStack<T>::top() const { // get the top element
87          assert(!isEmpty());
88          return elem[_top];
89  }
90
91  template<typename T>
```

```
92 void SeqStack<T>::bottom_up_traverse() const
93 {
94         for (int i = 0; i <= _top; ++i) {
95                 std::cout << elem[i] << '\t';
96         }
97 }
98
99 template<typename T>
100 void SeqStack<T>::overflowProcess() { // private member function, expanding the stack's size
101         T* newArray = new T[maxSize += stackIncreament];
102         assert(newArray != nullptr);
103         for (int i = 0; i <= _top; ++i) newArray[i] = elem[i];
104         delete elem;
105         elem = newArray;
106 }
107
108 #endif // !SEQSTACK_H
```

Listing 3: Sequential stack header

2.4 Stack.h

```
1  // stack header, abstract base class for interfaces
2  #pragma once
3  #ifndef STACK_H
4  #define STACK_H
5
6  template<typename T>
7  class Stack {
8  public:
9          virtual void push(const T& x) = 0;
10         virtual void pop() = 0;
11         virtual void pop(T& x) = 0;
12         virtual T& top()const = 0;
13         virtual void clear() = 0;
14         virtual bool isEmpty()const = 0;
15         virtual int size()const = 0;
16 };
17
18 #endif // !STACK_H
```

Listing 4: Stack header

3 Output

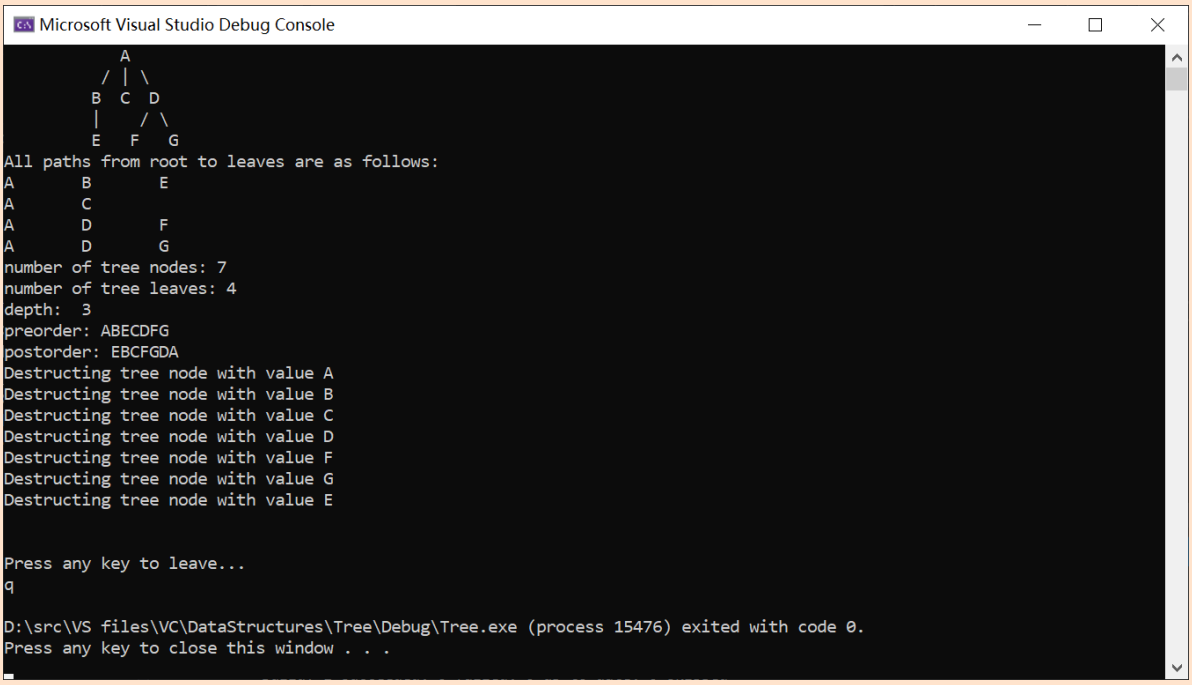

Figure 1: Tree test in Win10

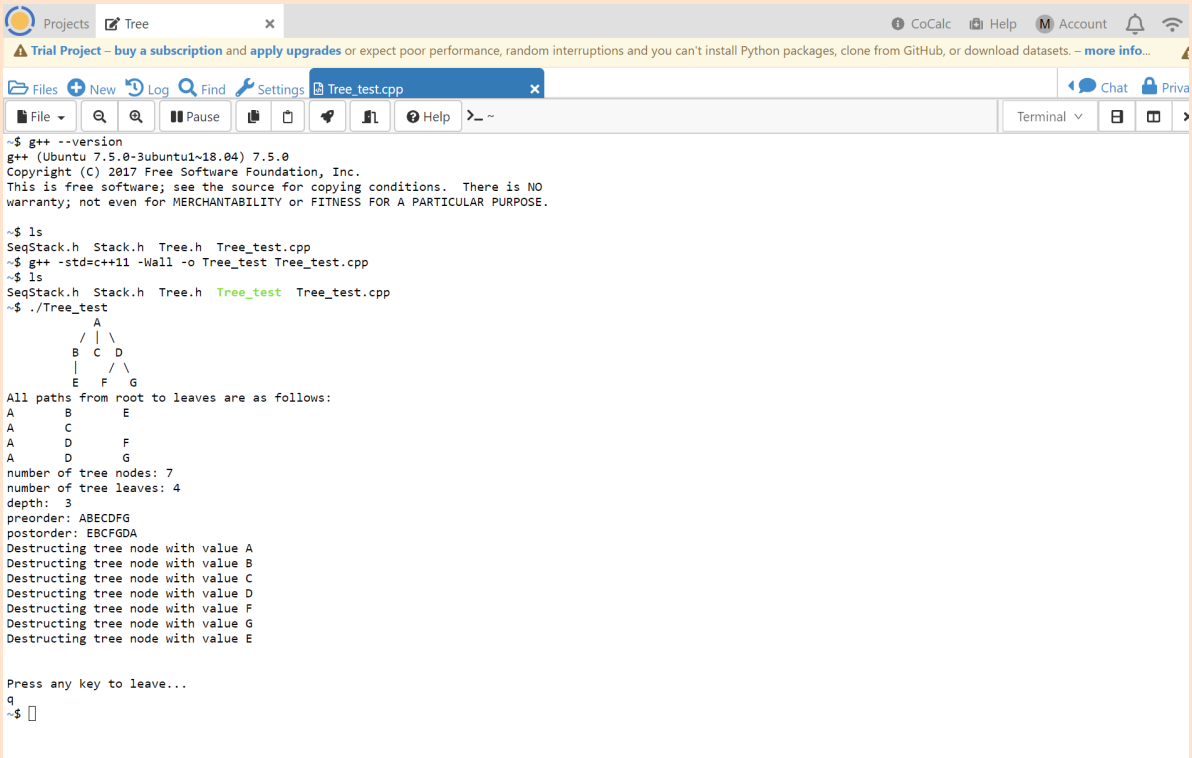

Figure 2: Tree test in Ubuntu

4 Appendix

Hello from the Beatles.😃