山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

学号：202000130143	姓名： 郑凯饶	班级： 计科 20.1
实验题目：二叉树操作		
实验学时：2	实验日期： 1124	

实验目的：

1. 掌握二叉树的基本概念，链表描述方法；二叉树操作的实现。

软件开发环境：

Windows 10 家庭中文版 64 位（10.0，版本 18363）

Dev-C++ IDE

1. 实验内容
2. 创建二叉树类。二叉树的存储结构使用链表。提供操作：前序遍历、中序遍历、后序遍历、层次遍历、计算二叉树结点数目、计算二叉树高度。
3. 接收二叉树前序序列和中序序列（各元素各不相同），输出该二叉树的后序序列。

4. 数据结构与算法描述 （整体思路描述，所需要的数据结构与算法）

```
// 不支持修改操作
template<class E>
class linkedBinaryTree // : public binaryTree<binaryTreeNode<E> >
{
    public:
        linkedBinaryTree() { root = NULL; Size = 0; }
        ~linkedBinaryTree() { erase(); root = NULL; Size = 0; }
        bool empty() const { return Size == 0; }
        int size() const { return Size; }
        E* rootele() const;
        void make(int n);
        void postOrder(void(*theVisit)(binaryTreeNode<E>*)) { visit = theVisit; postOrder(root); }
        void levelOrder(void(*theVisit)(binaryTreeNode<E>*)) { visit = theVisit; levelOrder(root); }

        void postOut() { postOrder(Output); cout << '\n'; }
        void levelOut() { levelOrder(Output); cout << '\n'; }

        void Count();

        int getnum(binaryTreeNode<E>* t) {
            if (t == NULL) return 0;
            int num = getnum(t->leftChild) + getnum(t->rightChild) + 1;
            return num;
        }
        int getheight(binaryTreeNode<E>* t) {
            if (t == NULL) return 0;
            int h = max(getheight(t->leftChild), getheight(t->rightChild)) + 1;
            return h;
        }
```

```
        binaryTreeNode<E>* getnode(int ind) { return node[ind]; }
        void erase();

        // rebuild tree
        void rebuild(int preord[], int inord[], int n) { root = crea(1, n, 1, n, preord, inord); }
        binaryTreeNode<E>* crea(int h1, int t1, int h2, int t2, int s1[], int s2[]);

    private:
        binaryTreeNode<E> *root;
        binaryTreeNode<E> **node;

        static void (*visit)(binaryTreeNode<E> *t);      // 访问函数
        static void preOrder(binaryTreeNode<E> *t);
        static void inOrder(binaryTreeNode<E> *t);
        static void postOrder(binaryTreeNode<E> *t);
        static void levelOrder(binaryTreeNode<E> *t);
        static void Output(binaryTreeNode<E> *t) { cout << t->element << " "; }
        static void dispose(binaryTreeNode<E> *t) { delete t; }

        int Size;
};
```

Private 成员：

Root:根节点

Node:由于题目输入的是某个结点的左右孩子，我们需要保存指向每个节点的指针，才能重建二叉树

Static 方法：

Visit：访问函数，形参为 binaryTreeNode<E>,可以将函数指针指向各个函数入口，融合进遍历方法中，实现遍历过程中的不同操作，提高代码的简洁性及效率。

*order():各种遍历方法

Output()：输出

Dispose()：回收节点

Size:二叉树大小

Public 成员：

make:根据输入重建树

postOrder()：后序遍历方法，形参为 void(*theVisit)(binaryTreeNode<E>*)，可接收不同的函数实现不同功能

levelOrder()：层次遍历方法

postOut():内部调用 postOrder 以及 output，输出树的后序序列

levelOut():输出树的层次序列

Count()：统计各个节点的子树大小，具体实现如下：

```
// start from root
//  recurve through stack
stack<int> stk;
stk.push(1);

while (!stk.empty()) {
    // element domain: nodeID
    int cur = stk.top(); vis[cur] = true;
    binaryTreeNode<E>* curNode = node[cur];

    int lc = -1, rc = -1;
    if (curNode->leftChild != NULL) lc = curNode->leftChild->element;
    if (curNode->rightChild != NULL) rc = curNode->rightChild->element;

    if (lc < 0 and rc < 0) {
        stk.pop();
    }
    else if (lc > 0 and rc > 0 and vis[lc] and vis[rc]) {
        stk.pop();
        sz[cur] += sz[lc] + sz[rc];
    }
    else if (lc > 0 and vis[lc]) {
        stk.pop();
        sz[cur] += sz[lc];
    }
    else if (rc > 0 and vis[rc]) {
        stk.pop();
        sz[cur] += sz[rc];
    }

    if (lc > 0 and !vis[lc]) stk.push(lc);          // 操作以 lc 为根的子树
    if (rc > 0 and !vis[rc]) stk.push(rc);
}
```

使用栈结构进行树的遍历（后序遍历），计算子树大小 sz[cur]=sz[lc]+sz[rc]，该方法 O（n）实现了对所有节点子树大小的统计。

Getnum()：递归计算该节点的子树大小
Getheight()：递归计算该节点的子树树高
Getnode()：获取索引为 ind 的节点指针

Rebuild()：根据二叉树的前序序列和中序序列，重建二叉树，具体实现：

```
template<class E>
binaryTreeNode<E>* linkedBinaryTree<E>::crea(int h1, int t1, int h2, int t2, int s1[], int s2[]) {
    if (h1 > t1 || h2 > t2) return NULL;
    binaryTreeNode<E> *head = new binaryTreeNode<E>;
    head->element = s1[h1];
    int i;
    for (i = h2; i <= t2; i++) {
        if (s2[i] == s1[h1]) break;
    }

    head->leftChild = crea(h1 + 1, h1 - h2 + i, h2, i - 1, s1, s2);
    head->rightChild = crea(h1 - h2 + i + 1, t1, i + 1, t2, s1, s2);
    return head;
}
```

对于前序序列 s1 和中序序列 s2，s1[0]是树的根节点，我们在 s2 中查找 s1[0]，在索引 i 处找到，以此可以得到以 s1[0]为根节点的左右子树，递归可重建整棵树。

```
template <class E> void (*linkedBinaryTree<E>::visit)(binaryTreeNode<E>*);
```

声明访问函数，在标程中注释直接使用会 compile error，但本地编译器和 OJ 平台可正常编译

5. 测试结果（测试输入，测试输出）
在 OJ 平台成功提交。

6. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）
这次实验我使用了许多局部对象！但未初始化！导致了许多内存访问错误。

```
template<class E>
void linkedBinaryTree<E>::make(int n) {

    root = new binaryTreeNode<E>(1);        // 根节点保证为 1
    node = new binaryTreeNode<E> *[n + 1];

    for (int i = 1; i <= n; i++) node[i] = NULL;    // **

    node[1] = root;
    for (int i = 1, a, b; i <= n; i++) {
        cin >> a >> b;
        if (node[i] == NULL) node[i] = new binaryTreeNode<E>(i);
        if (a != -1) {
            if (node[a] == NULL) {
                node[a] = new binaryTreeNode<E>(a);
                node[i]->leftChild = node[a];
            }
```

```
template <class E>
void linkedBinaryTree<E>::Count() {
    int *sz = new int[Size + 1];
    bool *vis = new bool[Size + 1];

    for (int i = 1; i <= Size; i++) {
        vis[i] = 0;
        sz[i] = 1;
    }

    // start from root
    //  recurve through stack
    stack<int> stk;
    stk.push(1);
```

7. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）

```
#include<bits/stdc++.h>
using namespace std;

/* 3221225477
3221226356
*/


template <class T>
struct binaryTreeNode
{
   T element;
   binaryTreeNode<T> *leftChild,   // left subtree
                     *rightChild;  // right subtree

   binaryTreeNode() {leftChild = rightChild = NULL;}
```

```
24.    binaryTreeNode(const T& theElement):element(theElement)
25.    {
26.       leftChild = rightChild = NULL;
27.    }
28.    binaryTreeNode(const T& theElement,
29.                   binaryTreeNode *theLeftChild,
30.                   binaryTreeNode *theRightChild)
31.                   :element(theElement)
32.    {
33.       leftChild = theLeftChild;
34.       rightChild = theRightChild;
35.    }
36.};
37.
38.// 不支持修改操作
39.template<class E>
40.class linkedBinaryTree // : public binaryTree<binaryTreeNode<E> >
41.{
42. public:
43.  linkedBinaryTree() { root = NULL; Size = 0; }
44.  ~linkedBinaryTree() { erase(); root = NULL; Size = 0; }
45.  bool empty() const { return Size == 0; }
46.  int size() const { return Size; }
47.  E* rootele() const;
48.  void make(int n);
49.  void postOrder(void(*theVisit)(binaryTreeNode<E>*)) { visit = theVisit; postOr
   der(root); }
50.  void levelOrder(void(*theVisit)(binaryTreeNode<E>*)) { visit = theVisit; level
   Order(root); }
51.
52.  void postOut() { postOrder(Output); cout << '\n'; }
53.  void levelOut() { levelOrder(Output); cout << '\n'; }
54.
55.  void Count();
56.
57.  int getnum(binaryTreeNode<E>* t) {
58.   if (t == NULL) return 0;
59.   int num = getnum(t->leftChild) + getnum(t->rightChild) + 1;
60.   return num;
61.  }
62.  int getheight(binaryTreeNode<E>* t) {
63.   if (t == NULL) return 0;
64.   int h = max(getheight(t->leftChild), getheight(t->rightChild)) + 1;
65.   return h;
66.  }
67.  binaryTreeNode<E>* getnode(int ind) { return node[ind]; }
```

```
68.  void erase();
69.
70.  // rebuild tree
71.  void rebuild(int preord[], int inord[], int n) { root = crea(1, n, 1, n, preor
   d, inord); }
72.  binaryTreeNode<E>* crea(int h1, int t1, int h2, int t2, int s1[], int s2[]);
73.
74. private:
75.  binaryTreeNode<E> *root;
76.  binaryTreeNode<E> **node;
77.
78.  static void (*visit)(binaryTreeNode<E> *t);  // 访问函数
79.  static void preOrder(binaryTreeNode<E> *t);
80.  static void inOrder(binaryTreeNode<E> *t);
81.  static void postOrder(binaryTreeNode<E> *t);
82.  static void levelOrder(binaryTreeNode<E> *t);
83.  static void Output(binaryTreeNode<E> *t) { cout << t->element << " "; }
84.  static void dispose(binaryTreeNode<E> *t) { delete t; }
85.
86.  int Size;
87.};
88.
89.template <class E> void (*linkedBinaryTree<E>::visit)(binaryTreeNode<E>*);
90.
91.template<class E>
92.binaryTreeNode<E>* linkedBinaryTree<E>::crea(int h1, int t1, int h2, int t2, int
    s1[], int s2[]) {
93. if (h1 > t1 || h2 > t2) return NULL;
94. binaryTreeNode<E> *head = new binaryTreeNode<E>;
95. head->element = s1[h1];
96. int i;
97. for (i = h2; i <= t2; i++) {
98.  if (s2[i] == s1[h1]) break;
99. }
100.
101.    head->leftChild = crea(h1 + 1, h1 - h2 + i, h2, i - 1, s1, s2);
102.    head->rightChild = crea(h1 - h2 + i + 1, t1, i + 1, t2, s1, s2);
103.    return head;
104.   }
105.
106.   template<class E>
107.   E* linkedBinaryTree<E>::rootele() const { // 返回指向根元素的指针
108.    if (!Size) return NULL;
109.    else return &root->element;
110.   }
111.
```

```
112.  template<class E>
113.  void linkedBinaryTree<E>::make(int n) {
114.
115.   root = new binaryTreeNode<E>(1);   // 根节点保证为 1
116.   node = new binaryTreeNode<E> *[n + 1];
117.
118.   for (int i = 1; i <= n; i++) node[i] = NULL; // **
119.
120.   node[1] = root;
121.   for (int i = 1, a, b; i <= n; i++) {
122.    cin >> a >> b;
123.    if (node[i] == NULL) node[i] = new binaryTreeNode<E>(i);
124.    if (a != -1) {
125.     if (node[a] == NULL) {
126.      node[a] = new binaryTreeNode<E>(a);
127.      node[i]->leftChild = node[a];
128.     }
129.     else {
130.      cout << node[a]->element << '\n';
131.      node[i]->leftChild = node[a];
132.     }
133.    }
134.    if (b != -1) {
135.     if (node[b] == NULL) {
136.      node[b] = new binaryTreeNode<E>(b);
137.      node[i]->rightChild = node[b];
138.     }
139.     else {
140.      node[i]->rightChild = node[b];
141.     }
142.    }
143.   }
144.   Size = n;
145.  }
146.
147.  template<class E>
148.  void linkedBinaryTree<E>::erase() {
149.   postOrder(dispose);
150.  }
151.
152.  template<class E>
153.  void linkedBinaryTree<E>::preOrder(binaryTreeNode<E> *t) // 根节点作为递归的
     起始点
154.  {// Preorder traversal.
155.     if (t != NULL)
156.     {
```

```
157.          linkedBinaryTree<E>::visit(t);
158.          preOrder(t->leftChild);
159.          preOrder(t->rightChild);
160.       }
161.    }
162.
163.    template<class E>
164.    void linkedBinaryTree<E>::inOrder(binaryTreeNode<E> *t)
165.    {// Inorder traversal.
166.       if (t != NULL)
167.       {
168.          inOrder(t->leftChild);
169.       linkedBinaryTree<E>::visit(t);
170.          inOrder(t->rightChild);
171.       }
172.    }
173.
174.    template<class E>
175.    void linkedBinaryTree<E>::postOrder(binaryTreeNode<E> *t)
176.    {// Postorder traversal.
177.       if (t != NULL)
178.       {
179.          postOrder(t->leftChild);
180.          postOrder(t->rightChild);
181.       linkedBinaryTree<E>::visit(t);
182.       }
183.    }
184.
185.    template <class E>
186.    void linkedBinaryTree<E>::levelOrder(binaryTreeNode<E> *t)
187.    {// Level-order traversal.
188.       queue<binaryTreeNode<E>*> q;
189.
190.       while (t != NULL) // 若根节点不为空
191.       {
192.          visit(t);  // visit t   *** theVisit -> visit
193.          // put t's children on queue
194.          if (t->leftChild != NULL)
195.             q.push(t->leftChild);
196.          if (t->rightChild != NULL)
197.             q.push(t->rightChild);
198.
199.          // get next node to visit
200.    //       try {t = q.front();}
201.    //       catch (queueEmpty) {return;}
202.      if (q.empty()) {
```

```
203.       return;
204.      }
205.      t = q.front();
206.         q.pop();
207.       }
208.    }
209.
210.    template <class E>
211.    void linkedBinaryTree<E>::Count() {
212.     int *sz = new int[Size + 1];
213.     bool *vis = new bool[Size + 1];
214.
215.     for (int i = 1; i <= Size; i++) {
216.      vis[i] = 0;
217.      sz[i] = 1;
218.     }
219.
220.     // start from root
221.     //  recurve through stack
222.     stack<int> stk;
223.     stk.push(1);
224.
225.     while (!stk.empty()) {
226.      // element domain: nodeID
227.      int cur = stk.top(); vis[cur] = true;
228.      binaryTreeNode<E>* curNode = node[cur];
229.
230.      int lc = -1, rc = -1;
231.      if (curNode->leftChild != NULL) lc = curNode->leftChild->element;
232.      if (curNode->rightChild != NULL) rc = curNode->rightChild->element;
233.
234.      if (lc < 0 and rc < 0) {
235.       stk.pop();
236.      }
237.      else if (lc > 0 and rc > 0 and vis[lc] and vis[rc]) {
238.       stk.pop();
239.       sz[cur] += sz[lc] + sz[rc];
240.      }
241.      else if (lc > 0 and vis[lc]) {
242.       stk.pop();
243.       sz[cur] += sz[lc];
244.      }
245.      else if (rc > 0 and vis[rc]) {
246.       stk.pop();
247.       sz[cur] += sz[rc];
248.      }
```

```
249.
250.      if (lc > 0 and !vis[lc]) stk.push(lc);   // 操作以 lc 为根的子树
251.      if (rc > 0 and !vis[rc]) stk.push(rc);
252.     }
253.
254.     for (int i = 1; i <= Size; i++) cout << sz[i] << " ";
255.     cout << '\n';
256.    }
257.
258.    void solve1() {
259.     linkedBinaryTree<int> tree;
260.     int n; cin >> n;
261.     tree.make(n);
262.     tree.levelOut();
263.
264.     tree.Count();
265.
266.     for (int i = 1; i <= n; i++) {
267.      cout << tree.getheight(tree.getnode(i)) << " ";
268.     }
269.     cout << endl;
270.    }
271.
272.    void solve2() {
273.     linkedBinaryTree<int> tree;
274.     int n; cin >> n;
275.     int *s1 = new int[n + 1];
276.     int *s2 = new int[n + 1];
277.     for (int i = 1; i <= n; i++) cin >> s1[i];
278.     for (int i = 1; i <= n; i++) cin >> s2[i];
279.
280.     tree.rebuild(s1, s2, n);
281.     tree.postOut();
282.    }
283.
284.    int main(){
285.
286.    // solve1();
287.     solve2();
288.
289.     return 0;
290.    }
```