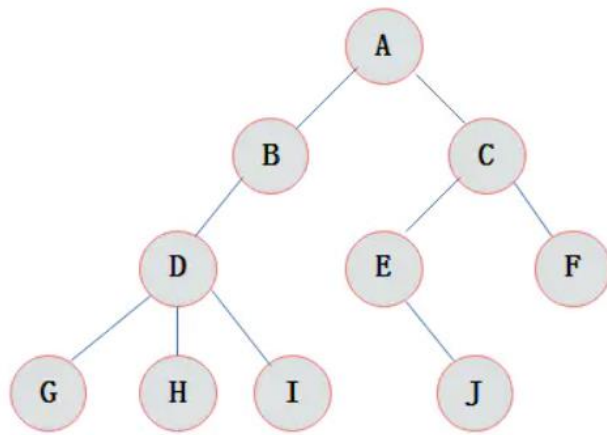


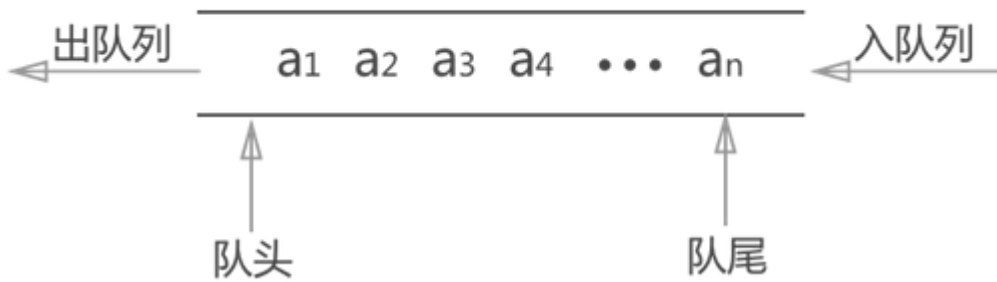
数据结构与算法 课程实验报告

学号：202000130198	姓名：隋春雨	班级：20.4
实验题目：二叉树操作		
实验学时：2	实验日期：2021-11-25	
实验目的： 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： CLION2020		
1. 实验内容 ① 题目描述： 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。 输入输出格式： 输入： 第一行为一个数字 n ($10 \leq n \leq 100000$)，表示有这棵树有 n 个节点，编号为 $1 \sim n$ 。之后 n 行每行两个数字，第 i 行的两个数字 a 、 b 表示编号为 i 的节点的左孩子节点为 a ，右孩子节点为 b ， -1 表示该位置没有节点。保证数据有效，根节点为 1 。 输出： 第一行， n 个数字，表示该树的层次遍历。 第二行， n 个数字，第 i 个数字表示以 i 节点为根的子树的结点数目。 第三行， n 个数字，第 i 个数字表示以 i 节点为根的子树的高度。 ② 题目描述： 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。 输入输出格式： 输入： 第一行为数字 n ； 第二行有 n 个数字，表示二叉树的前序遍历； 第三行有 n 个数字，表示二叉树的中序遍历。 输出： 输出一行，表示该二叉树的后序遍历序列。		
2. 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） 本次实验使用了两种方法解题，一种是递归的，一种是非递归的 ① A 题 a) 数据结构：二叉树。二叉树提供前序、中序、后序、层次遍历，同时通过遍历来计算二叉树的高度和结点的数目。对于二叉树的结点，有保存当前的元素和它的左右孩子，如果无左孩子或者右孩子，相应的设置为 <code>nullptr</code>		



b) 算法:

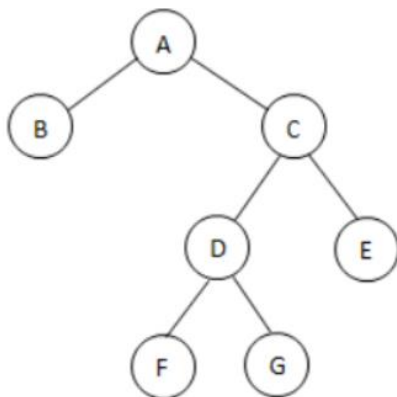
(一)层次遍历: 我们发现, 每一层中靠前的结点, 他们的子树也会靠前, 这是 FIFO 的性质, 我们使用队列来存储。如果队列非空, 那么我们继续。如果存在左右孩子 (非空), 那么我们就 Push 进去。



(二)计算二叉树的高度:

首先我们知道, 对于所有的递归函数, 都可以借助栈转换为非递归实现。因此本次实验也使用了递归和非递归来解题。

递归实现



对于每一个结点, 它的高度等于 $\max(\text{leftchild}, \text{rightchild}) + 1$, 所以我们可以很轻松的使用递归函数来进行求解即可。

非递归函数：我们使用一个 stack 来保存，对于每一个结点，如果是第一次被 push 进去，那么我们检查一下它的左右孩子是否非空，如果非空，那么 push。如果不是第一次被 push 进去，那么我们进行更新其父亲结点的高度， $\max(\text{now}, \text{child.height}+1)$ ，代码如下：

```
while (!s.empty())
{
    pair<binaryTreeNode<T>, binaryTreeNode<T>>node = s.top();
    if (!jud_push[node.second.element])//如果没有被push过
    {
        if (node.second.leftChild)//非空
        {
            s.push({ node.second,*node.second.leftChild });
        }
        if (node.second.rightChild)//非空
        {
            s.push({ node.second,*node.second.rightChild });
        }
        jud_push[node.second.element] = 1;
    }
    else
    {
        if (node.first.element!=node.second.element)//不是root结点
        {
            height[node.first.element] = max(height[node.first.element], height[node.second.element]);
        }
        s.pop();
    }
}
```

(三)计算结点数目：

递归实现：

每一个结点的 size 等于左右孩子的 size 和+1，所以我们可以使用递归函数来实现

```
139     template <class E>
140     int linkedBinaryTree<E>::height(binaryTreeNode<E> *t)//计算高度
141     { // 计算高度函数
142         if (t == NULL)//空树
143         {
144             return 0;//空树高度为0
145         }
146         int heightOfLeft = height(t->leftChild); //左子树高度
147         int heightOfRight = height(t->rightChild); //右子树高度
148         //取较高的那个，加上根节点，即为树高
149         if (heightOfLeft > heightOfRight)
150             return heightOfLeft+1;
151         else
152             return heightOfRight+1;
153     }
```

非递归实现：

同样的，类似上面的思路，我们发现对于每一个结点，先被访问的一定后出来，所以我们使用 stack 来

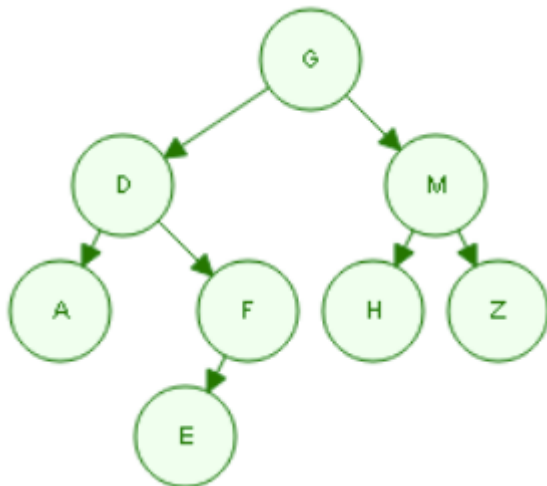
存储。每一个结点，如果没有被 Push 过，那么我们 push 其孩子结点。否则我们进行更新其父亲结点的 size，代码如下：

```
while (!s.empty())
{
    pair<binaryTreeNode<T>, binaryTreeNode<T>>node = s.top();
    if (!jud_push[node.second.element])//如果没有被push过
    {
        if (node.second.leftChild)//存在左孩子
        {
            s.push({ node.second,*node.second.leftChild });
        }
        if (node.second.rightChild)//存在右孩子
        {
            s.push({ node.second,*node.second.rightChild });
        }
        jud_push[node.second.element] = 1;//标记
    }
    else
    {
        if (node.second.element != node.first.element)//如果不是root结点
        {
            cnt[node.first.element] += cnt[node.second.element];//更新
        }
        s.pop();
    }
}
```

② B 题

由前序和中序确定后序：

比如这棵二叉树：



PreOrder: GDAFEMHZ

InOrder: ADEFGHMZ

那么我们怎么确定它的后序遍历呢？

首先我们知道，前序、中序为什么叫前序和中序，因为这标记着拜访 root 结点的顺序，前序是根、左、右，中序是左、根、右。给定两个序列，我们就可以唯一确定它的后序序列，因为对于一个前序序列，它的第一个结点就是 root，我们在中序序列中找到 root，就可以确定它的左右子树的 size。根据左右子树的 size 就可以确定左右子树的前序、中序遍历，递归下去，就可以求解出它的后序遍历。递归的终止条件为子树的 size 为 0。代码如下：

```

6 void output(vector<int> pre, vector<int> mid)
7 { //后序, 左、右、root
8     int root = pre[0];
9     int l_size = 0;
10    for (int i = 0; i < mid.size(); i++)
11    {
12        if (mid[i] == root)
13        {
14            break;
15        }
16        l_size++;
17    }
18    int r_size = mid.size() - 1 - l_size;
19    vector<int> l_pre( first: pre.begin() + 1, last: pre.begin() + 1 + l_size); //左子树前序
20    vector<int> l_mid(mid.begin(), mid.begin() + l_size); //左子树中序
21    vector<int> r_pre( first: pre.begin() + 1 + l_size, pre.end()); //右子树前序
22    vector<int> r_mid( first: mid.begin() + 1 + l_size, mid.end()); //右子树中序
23    if (l_size) //存在左子树
24    {
25        output(l_pre, l_mid);
26    }
27    if (r_size) //存在右子树
28    {
29        output(r_pre, r_mid);

```

3. 测试结果（测试输入，测试输出）

(1) a 题

输入：

输入

```

5
2 3
4 5
-1 -1
-1 -1
-1 -1

```

输出：

```
untitled110 x
C:\Users\4399\untitled11
5
2 3
4 5
-1 -1
-1 -1
-1 -1
1 2 3 4 5
5 3 1 1 1
3 2 1 1 1
```

提交 OJ 的结果：

✓Accepted

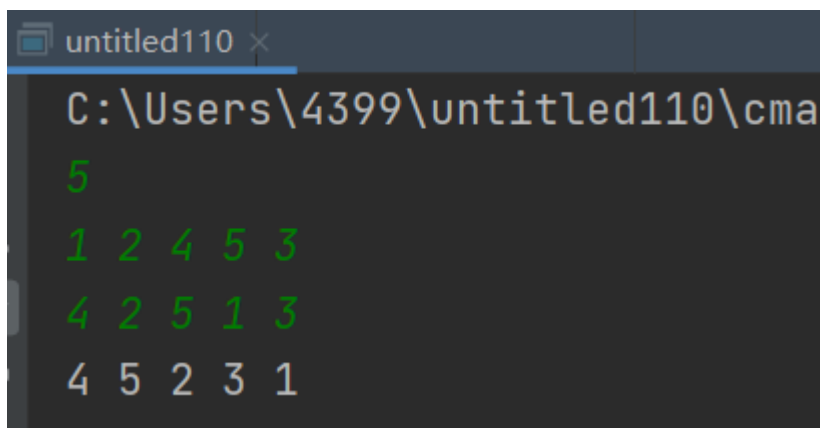
#	Result	Score	Time	Memory
1	✓Accepted	10	0 ms	3496 KiB
2	✓Accepted	10	0 ms	3444 KiB
3	✓Accepted	10	0 ms	3404 KiB
4	✓Accepted	10	1 ms	3368 KiB
5	✓Accepted	10	1 ms	3428 KiB
6	✓Accepted	10	2 ms	3488 KiB
7	✓Accepted	10	2 ms	3564 KiB
8	✓Accepted	10	3 ms	3508 KiB
9	✓Accepted	10	4 ms	3564 KiB
10	✓Accepted	10	6 ms	3788 KiB

(2)b 题
输入：

输入

```
5
1 2 4 5 3
4 2 5 1 3
```

输出：



```
untitled110 x
C:\Users\4399\untitled110\cma
5
1 2 4 5 3
4 2 5 1 3
4 5 2 3 1
```

提交 OJ 最后的结果：

✓ Accepted

#	Result	Score	Time	Memory
1	✓ Accepted	10	1 ms	3748 KiB
2	✓ Accepted	10	1 ms	4144 KiB
3	✓ Accepted	10	2 ms	4428 KiB
4	✓ Accepted	10	2 ms	4656 KiB
5	✓ Accepted	10	3 ms	5472 KiB
6	✓ Accepted	10	3 ms	5684 KiB
7	✓ Accepted	10	4 ms	6092 KiB
8	✓ Accepted	10	4 ms	6556 KiB
9	✓ Accepted	10	2 ms	6788 KiB
10	✓ Accepted	10	5 ms	7328 KiB

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

(1) 如何将递归实现转换为非递归实现？

解决：对于本道题来说，递归实现是不困难的。但是我们知道，所有的递归实现都可以借助循环与 stack 转换为非递归实现，考虑 A 题的操作。计算高度的问题，我们只有知道了子树的高度才能知道 root 结点的高度，因此这是一个后进先出的数据结构，使用 stack 来存储。

```
while (!s.empty())
{
    pair<binaryTreeNode<T>, binaryTreeNode<T>>node = s.top();
    if (!jud_push[node.second.element])//如果没有被push过
    {
        if (node.second.leftChild)//非空
        {
            s.push({ node.second,*node.second.leftChild });
        }
        if (node.second.rightChild)//非空
        {
            s.push({ node.second,*node.second.rightChild });
        }
        jud_push[node.second.element] = 1;
    }
    else
    {
        if(node.first.element!=node.second.element)//不是root结点
        {
            height[node.first.element] = max(height[node.first.element], height[node.second.element]);
        }
        s.pop();
    }
}
```

(2) 在提交 oj 的时候要把自己的测试删除，比如多输出了一个换行可能会导致全部的判错。

(3) 对于 B 题，不能使用 string 来存储。因为它有可能是两位数字，如果我们想当然的认为只有一位数字用 string 来存储的话，一定会 RE

(4) 对于 BOOL 数组的初始化，绝对不能想当然，每一次的定义都要对其进行初始化。在本次的实验中，因为 bool 数组没有初始化导致 debug 了很久。

```
bool* jud_push = new bool[_size + 1]();
```

(5) 对于下标从 1 开始的数组，要多动态分配一块内存。因为索引为 0 的地方我们是没有访问的。

```
int* height = new int[_size + 1];
```

(6) 对于一个只有两个私有成员的 struct，我们可以直接使用 pair 来存储。简化了我们的代码量。同时对于返回多个值的函数，我们只能使用引用来返回。这也是为什么兴起了 Go 语言的一个原因。

(7) 对于 height 等操作的计算，一定要特判是否合法。因为对于 root 结点来说，它的父节点是自身，不能直接增加。

```
if(node.first.element!=node.second.element)//非根结点
{
    height[node.first.element] = max(height[node.first.element], height[node.second.element]);
}
s.pop();
```


(8) 我们在写循环条件判断的时候，对于短路情况的判断一定要慎重，我们对于指针的使用一定要先判断是否为空，在进行取值操作，如下：

```
40     while (currentNode != NULL && //值不等于输入，且不越界，对于NULL值的判断要放到前边
41           currentNode->element.first != theKey)
42         currentNode = currentNode->next;
```

(9) 对于维护私有变量的时候，要特殊情况特殊判：比如说删除结点的时候，要考虑这个是不是头结点，如果是，那么更新私有变量。如下：

```
112     if (p != NULL && p->element.first == theKey)
113     {
114         if (tp == NULL) firstNode = p->next; //头结点特殊处理
115         else tp->next = p->next;
116         delete p; //删除
117         dSize--;
118     }
119 }
```

(10) 要注意私有成员的更新，public 函数知道自己所处的状态都是靠着私有成员才知道的，如果没有及时更新，那数据就成了垃圾数据，没有任何意义。我在写实验的时候，也经历过没更新导致的 Bug，最终 debug 查出来，就是下面这个：

```
    // switch to newQueue and set theFront and theBack
    theFront = 2 * arrayLength - 1; //更新私有成员
    theBack = arrayLength - 2;    // queue size arrayLength - 1
    arrayLength *= 2;
    queue = newQueue; //指针赋值
}

// put theElement at the theBack of the queue
theBack = (theBack + 1) % arrayLength;
queue[theBack] = theElement;
```

5. 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）

(1)A 题

```
1.  #include<iostream>
2.  #include <queue>
3.  #include <stack>
4.  using namespace std;
5.  template <class T>
6.  template<class T>
7.  class binaryTree
```

```

8.  {
9.    binaryTreeNode<T>* root;
10.   int _size;
11. public:
12.   binaryTree(binaryTreeNode<T>* root, int _size) : root(root), _size(_size) {}
13.   void levelOrder()
14.   {
15.       queue<binaryTreeNode<T>>q;
16.       if (root)
17.       {
18.           q.push(*root);
19.           while (!q.empty())//队列非空
20.           {
21.               binaryTreeNode<T> front = q.front();
22.               q.pop();
23.               cout << front.element << " ";
24.               if (front.leftChild)//有左孩子
25.               {
26.                   q.push(*front.leftChild);
27.               }
28.               if (front.rightChild)//有右孩子
29.               {
30.                   q.push(*front.rightChild);
31.               }
32.           }
33.           cout << endl;
34.       }
35.   }
36.   void size()
37.   {
38.       int* cnt = new int[1 + _size];
39.       for (int i = 1; i <= _size; i++)
40.       {
41.           cnt[i] = 1;//初始化
42.       }
43.       bool* jud_push = new bool[1 + _size]();
44.       pair<binaryTreeNode<T>, binaryTreeNode<T>>fa_son;//用pair 来存储
45.       stack<pair<binaryTreeNode<T>, binaryTreeNode<T>>>s;
46.       s.push({ *root,*root });//push 进去
47.       while (!s.empty())
48.       {
49.           pair<binaryTreeNode<T>, binaryTreeNode<T>>node = s.top();
50.           if (!jud_push[node.second.element])//如果没有被push 过
51.           {
52.               if (node.second.leftChild)//存在左孩子
53.               {

```

```

54.         s.push({ node.second,*node.second.leftChild });
55.     }
56.     if (node.second.rightChild)//存在右孩子
57.     {
58.         s.push({ node.second,*node.second.rightChild });
59.     }
60.     jud_push[node.second.element] = 1;//标记
61. }
62. else
63. {
64.     if (node.second.element != node.first.element)//如果不是 root 结点
65.     {
66.         cnt[node.first.element] += cnt[node.second.element];
67.     }
68.     s.pop();
69. }
70. }
71. for (int i = 1; i <= _size; i++)
72. {
73.     cout << cnt[i] << " ";
74. }
75. cout << endl;
76. delete []cnt;//收回内存
77. }
78. void height()
79. {
80.     int* height = new int[_size + 1];
81.     for (int i = 1; i <= _size; i++)
82.     {
83.         height[i] = 1;//初始化
84.     }
85.     bool* jud_push = new bool[_size + 1]();//初始化为0
86.     pair<binaryTreeNode<T>, binaryTreeNode<T>>fa_son;//父子
87.     stack<pair<binaryTreeNode<T>, binaryTreeNode<T>>>s;
88.     s.push({ *root,*root });
89.     while (!s.empty())
90.     {
91.         pair<binaryTreeNode<T>, binaryTreeNode<T>>node = s.top();
92.         if (!jud_push[node.second.element])//如果没有被 push 过
93.         {
94.             if (node.second.leftChild)
95.             {
96.                 s.push({ node.second,*node.second.leftChild });
97.             }
98.             if (node.second.rightChild)
99.             {

```

```

100.         s.push({ node.second,*node.second.rightChild });
101.     }
102.     jud_push[node.second.element] = 1;
103. }
104. else
105. {
106.     if(node.first.element!=node.second.element)//不是 root 结点
107.     {
108.         height[node.first.element] = max(height[node.first.element], height[node.second.element] + 1);
109.     }
110.     s.pop();
111. }
112. }
113. for (int i = 1; i <= _size; i++)
114. {
115.     cout << height[i] << " ";
116. }
117. cout << endl;
118. delete[]height;
119. }
120. };
121. int main()
122. {
123.     int n;
124.     cin >> n;
125.     binaryTreeNode<int>* node = new binaryTreeNode<int>[1 + n];
126.     for (int i = 1; i <= n; i++)
127.     {
128.         node[i].element = i;
129.         int l, r;
130.         cin >> l >> r;
131.         if (l >= 1)
132.         {
133.             node[i].leftChild = &node[l];
134.         }
135.         else
136.         {
137.             node[i].leftChild = nullptr;
138.         }
139.         if (r >= 1)
140.         {
141.             node[i].rightChild = &node[r];
142.         }
143.         else
144.         {
145.             node[i].rightChild = nullptr;

```

```

146.     }
147. }
148. binaryTree<int>b(node + 1, n);
149. b.levelOrder();
150. b.size();
151. b.height();
152. delete[]node;
153. return 0;
154. }

```

(2)B 题

```

1. #include<iostream>
2. using namespace std;
3. template<class T>
4. struct binaryTreeNode{//树节点类型
5.     T element;
6.     binaryTreeNode<T>* leftChild;
7.     binaryTreeNode<T>* rightChild;
8.     binaryTreeNode(){rightChild=leftChild=NULL;}
9.     binaryTreeNode(const T& theElement):element(theElement)//初始化
10.    {
11.        leftChild = rightChild = NULL;
12.    }
13.     binaryTreeNode(const T& theElement,binaryTreeNode<T>* theLeftChild,binaryTreeNode<T>* theRightChild)
14.    {
15.        element=theElement;
16.        leftChild=theLeftChild;
17.        rightChild=theRightChild;
18.    }
19. };
20. template<typename E>
21.     binaryTreeNode<E>* build(E* preo,int lp,int rp,E* ino,int li,int ri,int *re,int n)
22.    {//前序序列preo,处理区间[lp,rp],中序序列ino,处理区间[li,ri],辅助数组re,总节点个数n;
23.        if(li>ri) return NULL;
24.        binaryTreeNode<E>* now=new binaryTreeNode<E>;
25.        int pos=re[(int)preo[lp]];//当前子树的根在中序中的位置
26.        int numl=pos-li;//左子树的节点个数
27.        now->leftChild=build(preo,lp+1,lp+numl,ino,li,pos-1,re,n);
28.        now->rightChild=build(preo,lp+numl+1,rp,ino,pos+1,ri,re,n);
29.        now->element=preo[lp];

```

```
30. return now;
31. }
32. void work(binaryTreeNode<int>* t)
33. { //输出后序遍历
34. if(t==NULL) return ;
35. work(t->leftChild);
36. work(t->rightChild);
37. cout<<t->element<<' ';
38. }
39. int main()
40. {
41. int n;
42. cin>>n;
43. int* a=new int[n+1];
44. int* b=new int[n+1];
45. int* re=new int[n+1]; //初始化 re 数组
46. for(int i=1;i<=n;++i) cin>>a[i];
47. for(int i=1;i<=n;++i) cin>>b[i];
48. for(int i=1;i<=n;++i) re[b[i]]=i; //记录每个数值在中序遍历中的位置
49. work(build(a,1,n,b,1,n,re,n));
50. return 0;
51. }
```