

```

1  /*
2  树的定义：
3      根 + 子树，以一种递归的方式定义
4      子树是不相交的
5      除了根结点外，每个结点有且仅有一个父结点
6      一棵 N 个节点的树有 N-1 条边
7      规定根结点在第 1 层
8
9  树的表示：
10     用数组表示比较困难
11     用链表实现比较方便，但是每个结点的结构会因子树数目的不同而不同
12     如果令结点结构一致，会造成空间上的浪费
13
14     建议使用“儿子-兄弟表示法”
15     左指针指向第一个儿子，右指针指向下一个兄弟
16     这样，一般的树就转换成了二叉树
17
18  二叉树的定义：
19     这个集合可以为空
20     若不为空，则它由根结点和左子树、右子树两个不相交的二叉树组成
21     二叉树的子树有左右顺序之分
22
23  特殊的二叉树：
24     斜二叉树
25     完美/满二叉树
26     完全二叉树
27
28  二叉树的几个重要性质：
29     二叉树第 i 层的最大结点数为  $2^{(i-1)}$ ， $i \geq 1$ ，层数从第一层开始计算
30     深度为k的二叉树最大结点数为  $2^k - 1$ ， $k \geq 1$ 
31      $n_0$ 表示叶结点数， $n_2$ 表示度为2的结点数， $n_0 = n_2 + 1$ 
32
33  二叉树的遍历：
34     先序遍历
35     中序遍历
36     后序遍历
37     光有前序和后序序列，无法唯一确定二叉树
38     层次遍历
39  */
40
41
42  #include<stdio.h>
43  #include<malloc.h>
44  #include<vector>
45  #include<queue>
46  #include<algorithm>
47  using namespace std;
48
49
50  typedef struct TreeNode *BinTree;
51  struct TreeNode{
52      int Data;          // 存值
53      BinTree Left;     // 左儿子结点
54      BinTree Right;    // 右儿子结点
55  };
56  BinTree CreatBinTree(); // 创建一个二叉树
57  bool IsEmpty(BinTree BT); // 判断树 BT 是否为空
58  void PreOrderTraversal_1(BinTree BT); // 先序遍历，根左右
59  void PreOrderTraversal_2(BinTree BT); // 先序遍历，根左右
60  void InOrderTraversal_1(BinTree BT); // 中序遍历，左根右
61  void InOrderTraversal_2(BinTree BT); // 中序遍历，左根右
62  void PostOrderTraversal_1(BinTree BT); // 后序遍历，左右根
63  void PostOrderTraversal_2(BinTree BT); // 后序遍历，左右根
64
65
66  typedef struct SNode *Stack;
67  struct SNode{
68      BinTree Data;
69      Stack Next;
70  };
71  Stack CreateStack(); // 初始化链栈
72  int IsEmpty(Stack S); // 判断链栈是否为空
73  void Push(Stack S,BinTree item); // 入栈

```

```

74 BinTree Pop(Stack S); // 出栈
75
76
77 // 初始化
78 Stack CreateStack()
79 {
80     Stack S;
81     S = (Stack)malloc(sizeof(struct SNode));
82     S->Next = NULL; // 头结点不保存元素
83     return S;
84 }
85
86 // 判断是否为空
87 int IsEmpty(Stack S)
88 {
89     return (S->Next == NULL);
90 }
91
92 // 入栈
93 void Push(Stack S, BinTree item)
94 {
95     Stack tmp;
96     tmp = (Stack)malloc(sizeof(struct SNode));
97     tmp->Data = item;
98     tmp->Next = S->Next;
99     S->Next = tmp;
100 }
101
102 // 出栈
103 BinTree Pop(Stack S)
104 {
105     Stack popNode;
106     BinTree popVal;
107     if(IsEmpty(S))
108     {
109         printf("堆栈空");
110         return 0;
111     }
112     else
113     {
114         popNode = S->Next;
115         S->Next = popNode->Next;
116         popVal = popNode->Data; // 取出被删除结点的值
117         free(popNode); // 释放空间
118         return popVal;
119     }
120 }
121
122 BinTree Insert(int Data)
123 {
124     BinTree BT;
125     BT = (BinTree)malloc(sizeof(struct TreeNode));
126     BT->Data = Data;
127     BT->Left = NULL;
128     BT->Right = NULL;
129     return BT;
130 }
131
132 // 初始化二叉树
133 BinTree CreatBinTree()
134 {
135     BinTree BT;
136     BT = (BinTree)malloc(sizeof(struct TreeNode));
137     BT->Data = 1;
138     BT->Left = Insert(2);
139     BT->Right = Insert(3);
140     BT->Left->Left = Insert(4);
141     BT->Left->Right = Insert(6);
142     BT->Left->Right->Left = Insert(5);
143     BT->Right->Left = Insert(7);
144     BT->Right->Right = Insert(9);
145     BT->Right->Left->Right = Insert(8);
146     return BT;

```

```

147 }
148
149 // 根据前序序列初始化二叉树
150 // 1 2 4 0 0 6 5 0 0 0 3 7 0 8 0 0 9 0 0
151 void CreatBinTreeFomPreorder(BinTree &BT)
152 {
153     int val;
154     scanf("%d", &val);
155     if(val == 0)
156         BT = NULL;
157     else
158     {
159         BT = (BinTree)malloc(sizeof(struct TreeNode));
160         BT->Data = val;
161         CreatBinTreeFomPreorder(BT->Left);
162         CreatBinTreeFomPreorder(BT->Right);
163     }
164 }
165
166 // 先序递归
167 void PreOrderTraversal_1(BinTree BT)
168 {
169     if(BT)
170     {
171         printf("%d", BT->Data); // 打印根
172         PreOrderTraversal_1(BT->Left); // 进入左子树
173         PreOrderTraversal_1(BT->Right); // 进入右子树
174     }
175 }
176
177 // 先序非递归
178 void PreOrderTraversal_2(BinTree BT)
179 {
180     BinTree T = BT;
181     Stack S = CreateStack(); // 创建并初始化堆栈 s
182     while(T || !IsEmpty(S)) // 当树不为空或堆栈不空
183     {
184         while(T)
185         {
186             Push(S, T); // 压栈，第一次遇到该结点
187             printf("%d", T->Data); // 访问结点
188             T = T->Left; // 遍历左子树
189         }
190         if(!IsEmpty(S)) // 当堆栈不空
191         {
192             T = Pop(S); // 出栈，第二次遇到该结点
193             T = T->Right; // 访问右结点
194         }
195     }
196 }
197
198 // 中序递归
199 void InOrderTraversal_1(BinTree BT)
200 {
201     if(BT)
202     {
203         InOrderTraversal_1(BT->Left); // 进入左子树
204         printf("%d", BT->Data); // 打印根
205         InOrderTraversal_1(BT->Right); // 进入右子树
206     }
207 }
208
209 // 中序非递归
210 void InOrderTraversal_2(BinTree BT)
211 {
212     BinTree T = BT;
213     Stack S = CreateStack(); // 创建并初始化堆栈 s
214     while(T || !IsEmpty(S)) // 当树不为空或堆栈不空
215     {
216         while(T)
217         {
218             Push(S, T); // 压栈
219             T = T->Left; // 遍历左子树

```

```

220     }
221     if(!IsEmpty(S))    // 当堆栈不空
222     {
223         T = Pop(S);    // 出栈
224         printf("%d", T->Data); // 访问结点
225         T = T->Right;  // 访问右结点
226     }
227 }
228 }
229
230 // 后序递归
231 void PostOrderTraversal_1(BinTree BT)
232 {
233     if(BT)
234     {
235         PostOrderTraversal_1(BT->Left); // 进入左子树
236         PostOrderTraversal_1(BT->Right); // 进入右子树
237         printf("%d", BT->Data);        // 打印根
238     }
239 }
240
241 // 后序非递归
242 void PostOrderTraversal_2(BinTree BT)
243 {
244     BinTree T = BT;
245     Stack S = CreateStack(); // 创建并初始化堆栈 s
246     vector<BinTree> v;
247     Push(S, T);
248     while(!IsEmpty(S)) // 当前结点非空
249     {
250         T = Pop(S);
251         v.push_back(T);
252         if(T->Left)
253             Push(S, T->Left);
254         if(T->Right)
255             Push(S, T->Right);
256     }
257     reverse(v.begin(), v.end()); // 逆转
258     for(int i=0; i<v.size(); i++)
259         printf("%d", v[i]->Data);
260 }
261
262 // 层次遍历
263 void LevelOrderTraversal(BinTree BT)
264 {
265     queue<BinTree> q;
266     BinTree T;
267     if(!BT)
268         return;
269
270     q.push(BT); // BT 入队
271     while(!q.empty())
272     {
273         T = q.front(); // 访问队首元素
274         q.pop();       // 出队
275         printf("%d", T->Data);
276         if(T->Left)
277             q.push(T->Left);
278         if(T->Right)
279             q.push(T->Right);
280     }
281 }
282
283 // 输出叶子结点
284 void FindLeaves(BinTree BT)
285 {
286     if(BT)
287     {
288         if(!BT->Left && !BT->Right)
289             printf("%d", BT->Data); // 打印叶子结点
290         FindLeaves(BT->Left); // 进入左子树
291         FindLeaves(BT->Right); // 进入右子树
292     }

```

```

293 }
294
295 // 求树高度
296 int GetHeight(BinTree BT)
297 {
298     int hl, hr, maxh;
299     if(BT)
300     {
301         hl = GetHeight(BT->Left); // 求左子树高度
302         hr = GetHeight(BT->Right); // 求右子树高度
303         maxh = (hl>hr)?hl:hr;
304         return maxh + 1; // 当前结点高度为左右子树最大的高度+1
305     }
306     else
307         return 0;
308 }
309
310 int main()
311 {
312     BinTree BT, ST;
313
314     printf("请按照前序序列输入二叉树，空结点用0表示，结点之间用空格分隔：\n");
315     // CreatBinTreeFomPreorder(BT);
316     BT = CreatBinTree();
317
318     printf("先序遍历：\n");
319     PreOrderTraversal_1(BT);
320     printf("\n");
321     PreOrderTraversal_2(BT);
322
323     printf("\n中序遍历：\n");
324     InOrderTraversal_1(BT);
325     printf("\n");
326     InOrderTraversal_2(BT);
327
328     printf("\n后序遍历：\n");
329     PostOrderTraversal_1(BT);
330     printf("\n");
331     PostOrderTraversal_2(BT);
332
333     printf("\n层次遍历：");
334     LevelOrderTraversal(BT);
335
336     printf("\n输出叶子结点：");
337     FindLeaves(BT);
338
339     printf("\n输出树的高度： %d", GetHeight(BT));
340     return 0;
341 }
342

```