

文章目录

数据结构C++——二叉树

- 一、前言
- 二、二叉树的一些常见操作
 - ①二叉树的存储结构
 - ②二叉树的先序遍历
 - ③二叉树的中序遍历
 - ④二叉树的后序遍历
 - ⑤复制二叉树
 - ⑤计算二叉树的深度
 - ⑥统计二叉树中结点的个数
- 三、完整代码
- 三、总结

一、前言

树的遍历操作需要和栈相结合，虽然C++中有许多关于数据结构的头文件可以直接拿来使用，但对于数据结构的初学者，栈的常见操作、基本原理、代码实现都应该了熟于心。

关于数据结构——栈的文章：链接: [数据结构与算法分析（C++）——栈](#).

二、二叉树的一些常见操作

①二叉树的存储结构

二叉树的顺序存储表示

```
1  /*----- 二叉树的顺序存储表示-----*/
2  #define OK 1
3  #define ERROR 0
4  #define MAXSIZE 100
5  typedef TElemType SqBiTree[MAXSIZE];
6  SqBiTree bt;
```

二叉树的二叉链表存储表示

```
1
2  /*----- 二叉树的二叉链表存储表示-----*/
3  #define OK 1
4  #define ERROR 0
5  #define MAXSIZE 100
6  typedef int Status;
```

```

7 | typedef char TElemType; // 定义树结点的数据类型
8 | typedef struct BiTNode {
9 |     TElemType data; // 结点数据域
10 |     struct BiTNode* lchild, * rchild; // 结点指针域
11 |     bool isFirst; // 非递归的后序遍历用来判断某结点是否第一次出现在栈顶
12 | }BiTNode, *BiTree;

```

②二叉树的 先序遍历

先序遍历的递归算法

```

1 | 先序遍历的递归算法思路：
2 | 1: 访问根结点
3 | 2: 先序遍历左子树
4 | 3: 先序遍历右子树

```

```

1 | /*-----先序遍历二叉树T的递归算法-----*/
2 | void InOrderTraverse011(BiTree T) {
3 |     // 先序遍历的递归算法
4 |     if (T) {
5 |         cout << T->data; // 访问根结点
6 |         InOrderTraverse011(T->lchild); // 先序遍历左子树
7 |         InOrderTraverse011(T->rchild); // 先序遍历右子树
8 |     }
9 | }

```

先序遍历的非递归算法

```

1 | 先序遍历的非递归算法思路：
2 | 1: 从根结点开始，先访问根结点，输出根结点的数据域的值
3 | 2: 将根结点压入栈，以便于后续弹栈遍历右子树
4 | 3: 遍历左子树且不断输出子树根结点，并将结点压入栈
5 | 4: 从栈顶弹出无左子树的结点，开始遍历右子树

```

```

1 | /*-----先序遍历二叉树T的非递归算法-----*/
2 | void InOrderTraverse012(BiTree T) {
3 |     LinkStack S = new StackNode; // 定义一个链栈
4 |     InitStack(S); // 初始化此链栈
5 |     BiTNode* p = new BiTNode;
6 |     p = T;
7 |     BiTNode* q = new BiTNode; // q指针用于接收出栈元素
8 |     while (p || !StackEmpty(S)) {
9 |

```

```

9
10         if (p) {
11             cout << p->data; // 访问根结点

```

③ 二叉树的中序遍历

中序遍历的递归算法

```

1  中序遍历的递归算法思路：
2
3  1: 中序遍历左子树
4  2: 访问根结点
5  3: 中序遍历右子树

```

```

1  /*----- 中序遍历的递归算法-----*/
2  void InOrderTraverse001(BiTree T) {
3      // 中序遍历二叉树T的递归算法
4      if (T)
5      {
6          InOrderTraverse001(T->lchild); // 中序遍历左子树
7          cout << T->data; // 访问根节点
8          InOrderTraverse001(T->rchild); // 中序遍历右子树
9      }
10 }

```

中序遍历的非递归算法

```

1  中序遍历的非递归算法思路：
2
3  1: 从根结点开始，遇到结点则将结点压栈
4  2: 当遇到无左子树的结点时，将此结点弹栈且访问它并遍历它的右子树
5  3: 若该结点为叶子结点，则继续弹栈，开始遍历它的父节点的右子树

```

```

1  /*----- 中序遍历的非递归算法-----*/
2  void InOrderTraverse002(BiTree T) {
3      // 中序遍历二叉树T的非递归算法
4      LinkStack S = new StackNode; // 定义一个链栈
5      InitStack(S); // 初始化此链栈
6      BiTNode* p = new BiTNode;
7      p = T;
8      BiTNode* q = new BiTNode; // q指针用于接收出栈元素
9      while (p || !StackEmpty(S)) {
10

```



④二叉树的后序遍历

后序遍历的递归算法

```

1  后序遍历的递归算法思路：
2  1: 后序遍历左子树
3  2: 后序遍历右子树
4  3: 访问根结点

```

```

1  /*-----后序遍历二叉树T的递归算法-----*/
2  void InOrderTraverse021(BiTree T) {
3      if (T) {
4          InOrderTraverse021(T->lchild); //后序遍历左子树
5          InOrderTraverse021(T->rchild); //后序遍历右子树
6          cout << T->data; //访问根结点
7      }
8  }

```

后序遍历的非递归算法

```

1  后序遍历的非递归算法思路：
2  1: 从根结点开始，沿其左子树一直往下搜索且压栈，直至出现没有左子树的结点
3  2: 将此结点的isFirst域置为true，表明该结点第一次出现在栈顶
4  3: 取栈顶元素并弹栈，若该栈顶元素第一次出现在栈顶则压栈，并将其isFirst域置为false，开始
5  4: 若该栈顶元素第二次出现在栈顶则访问该结点，并将p指针置空以便继续弹栈，操作其父节点。

```



```

1  /*-----后序遍历二叉树T的非递归算法-----*/
2  void InOrderTraverse022(BiTree T) {
3      LinkStack S = new StackNode; //定义一个链栈
4      InitStack(S); //初始化此链栈
5      BiTNode* p = new BiTNode;
6      p = T;
7      BiTNode* q = new BiTNode; //出栈时接收栈顶元素
8      BiTNode* t = new BiTNode; //接收栈顶元素
9      while (p || !StackEmpty(S)) {
10         while (p) //沿左子树一直往下搜索，直至出现没有左子树的结点
11

```

⑤复制二叉树

复制二叉树。

```
1  /*-----复制二叉树-----*/
2  void Copy(BiTree T, BiTree& NewT) {
3      //复制一颗和T完全相同的二叉树
4      if (T == NULL) { //如果是空栈, 递归结束
5          NewT = NULL;
6          return;
7      }
8      else {
9          NewT = new BiTNode;
10         NewT->data = T->data; //复制根结点
11         Copy(T->lchild, NewT->lchild); //递归复制左子树
12         Copy(T->rchild, NewT->rchild); //递归复制右子树
13     }
14 }
```

⑤计算二叉树的深度

计算二叉树的深度。

```
1  /*-----计算二叉树的深度-----*/
2  int Depth(BiTree T) {
3      //计算二叉树T的深度
4      if (T == NULL) return 0; //如果是空树, 深度为0, 递归结束
5      else {
6          int m, n;
7          m = Depth(T->lchild); //递归计算左子树的深度记为m
8          n = Depth(T->rchild); //递归计算右子树的深度记为n
9          if (m > n) return (m + 1); //二叉树的深度为m与n的较大者加1
10         else return (n + 1);
11     }
12 }
```

⑥统计二叉树中结点的个数

统计二叉树中结点的个数。

```
1  /*-----统计二叉树中结点的个数-----*/
2  int NodeCount(BiTree T) {
3
```

```

3 //统计二叉树T中结点的个数
4
5 if (T == NULL) return 0; //如果是空树，则结点个数为0，递归结束
6 else return NodeCount(T->lchild) + NodeCount(T->rchild) + 1;
7 //否则结点个数为左子树的结点个数+右子树的结点个数+1
8 }

```

-----一道华丽的分割线-----

三、完整代码

操作二叉树的完整代码（含main函数）。

```

1  #include<iostream>
2  using namespace std;
3  #define OK 1
4  #define ERROR 0
5  #define MAXSIZE 100
6  typedef int Status;
7  typedef char TElemType; //定义树结点的数据类型
8
9
10 /*----- 二叉树的二叉链表存储表示-----*/
11 typedef struct BiTNode {
12     TElemType data; // 结点数据域
13     struct BiTNode* lchild, * rchild; // 结点指针域
14     bool isFirst; // 非递归的后序遍历用来判断某结点是否第一次出现在栈顶
15 }BiTNode, *BiTree;
16 typedef BiTree SElemType;
17 /*-----栈的存储结构-----*/
18 /*----- 栈的存储结构-----*/
19 typedef struct SqStack {
20     SElemType data; // 结点数据域
21     struct SqStack* next; // 结点的指针域
22 }StackNode, * LinkStack;
23 /*----- 栈的初始化-----*/
24 Status InitStack(LinkStack& S) {
25     // 栈的初始化
26     S = NULL;
27     return OK;
28 }
29 /*----- 判断是否栈空-----*/
30 Status StackEmpty(LinkStack& S) {
31     if (S == NULL) return OK;
32     return ERROR;
33 }
34 /*----- 链栈的入栈-----*/

```

```

35 Status Push(LinkStack& S, SElemType e) {
36     StackNode* p = new StackNode;
37     p->data = e;
38     p->next = S;
39     S = p;
40     return OK;
41 }
42 /*-----链栈的出栈-----*/
43 Status Pop(LinkStack& S, SElemType& e) {
44     if (S == NULL) return ERROR;
45     e = S->data;
46     StackNode* p = new StackNode;
47     p = S;
48     S = S->next;
49     delete p;
50     return OK;
51 }
52 /*-----链栈的取栈顶元素-----*/
53 SElemType GetTop(LinkStack& S) {
54     if (S!=NULL)
55     {
56         return S->data;
57     }
58 }
59 /*-----中序遍历的递归算法-----*/
60 void InOrderTraverse001(BiTree T) {
61     //中序遍历二叉树T的递归算法
62     if (T)
63     {
64         InOrderTraverse001(T->lchild); //中序遍历左子树
65         cout << T->data; //访问根节点
66         InOrderTraverse001(T->rchild); //中序遍历右子树
67     }
68 }
69 /*-----中序遍历二叉树T的非递归算法-----*/
70 void InOrderTraverse002(BiTree T) {
71     //中序遍历二叉树T的非递归算法
72     LinkStack S = new StackNode; //定义一个链栈
73     InitStack(S); //初始化此链栈
74     BiTNode* p = new BiTNode;
75     p = T;
76     BiTNode* q = new BiTNode; //q指针用于接收出栈元素
77     while (p || !StackEmpty(S)) {
78         if (p) { //p非空
79             Push(S, p); //根指针进栈
80             p = p->lchild; //遍历左子树

```

```

82         }
83         else {
84             Pop(S, q); //退栈
85             cout << q->data; //访问根结点
86             p = q->rchild; //遍历右子树
87         }
88     }
89 }
90
91 /*-----先序遍历二叉树T的递归算法-----*/
92 void InOrderTraverse011(BiTree T) {
93     //先序遍历的递归算法
94     if (T) {
95         cout << T->data; //访问根结点
96         InOrderTraverse011(T->lchild); //先序遍历左子树
97         InOrderTraverse011(T->rchild); //先序遍历右子树
98     }
99 }
100 /*-----先序遍历二叉树T的非递归算法-----*/
101 void InOrderTraverse012(BiTree T) {
102     LinkStack S = new StackNode; //定义一个链栈
103     InitStack(S); //初始化此链栈
104     BiTNode* p = new BiTNode;
105     p = T;
106     BiTNode* q = new BiTNode; //q指针用于接收出栈元素
107     while (p || !StackEmpty(S)) {
108         if (p) {
109             cout << p->data; //访问根结点
110             Push(S, p); //根指针进栈
111             p = p->lchild; //遍历左子树
112         }
113         else {
114             Pop(S, q); //退栈
115             p = q->rchild; //遍历右子树
116         }
117     }
118 }
119 /*-----后序遍历二叉树T的递归算法-----*/
120 void InOrderTraverse021(BiTree T) {
121     if (T) {
122         InOrderTraverse021(T->lchild); //后序遍历左子树
123         InOrderTraverse021(T->rchild); //后序遍历右子树
124         cout << T->data; //访问根结点
125     }
126 }
127 /*-----后序遍历二叉树T的非递归算法-----*/
128 void InOrderTraverse022(BiTree T) {
129     LinkStack S = new StackNode; //定义一个链栈

```



```

130     InitStack(S); // 初始化此链栈
131     BiTNode* p = new BiTNode;
132     p = T;
133     BiTNode* q = new BiTNode; // 出栈时接收栈顶元素
134     BiTNode* t = new BiTNode; // 接收栈顶元素
135     while (p || !StackEmpty(S)) {
136         while (p) // 沿左子树一直往下搜索, 直至出现没有左子树的结点
137         {
138             Push(S, p);
139             p->isFirst = true; // 此时结点均是第一次成为栈顶元素
140             p = p->lchild; // 遍历左子树
141         }
142         if (!StackEmpty(S))
143         {
144             t = GetTop(S); // 取栈顶元素
145             Pop(S, q); // 出栈
146             if (t->isFirst == true) { // 若t为第一次出现在栈顶元素
147                 Push(S, q);
148                 t->isFirst = false;
149                 p = q->rchild; // 遍历栈顶元素的右子树
150             }
151             else
152             { // 此时t为第二次出现在栈顶元素了
153                 cout << t->data; // 访问栈顶元素
154                 p = NULL; // 将指针置空
155             }
156         }
157     }
158 }
159
160 /*-----先序遍历的顺序建立二叉链表-----*/
161 void CreateBiTree(BiTree& T) {
162     // 按先序次序输入二叉树中结点的值 (一个字符)、创建二叉链表表示的二叉树T
163     TElemType ch;
164     cin >> ch;
165     if (ch == '#') T = NULL; // 递归结束, 建空树
166     else // 递归创建二叉树
167     {
168         T = new BiTNode; // 生成根结点
169         T->data = ch; // 根结点数据域置为ch
170         CreateBiTree(T->lchild); // 递归创建左子树
171         CreateBiTree(T->rchild); // 递归创建右子树
172     }
173 }
174
175 /*-----复制二叉树-----*/
176 void Copy(BiTree T, BiTree& NewT) {
177     // 复制一颗和T完全相同的二叉树
178     if (T == NULL) { // 如果是空栈, 递归结束
179

```

```

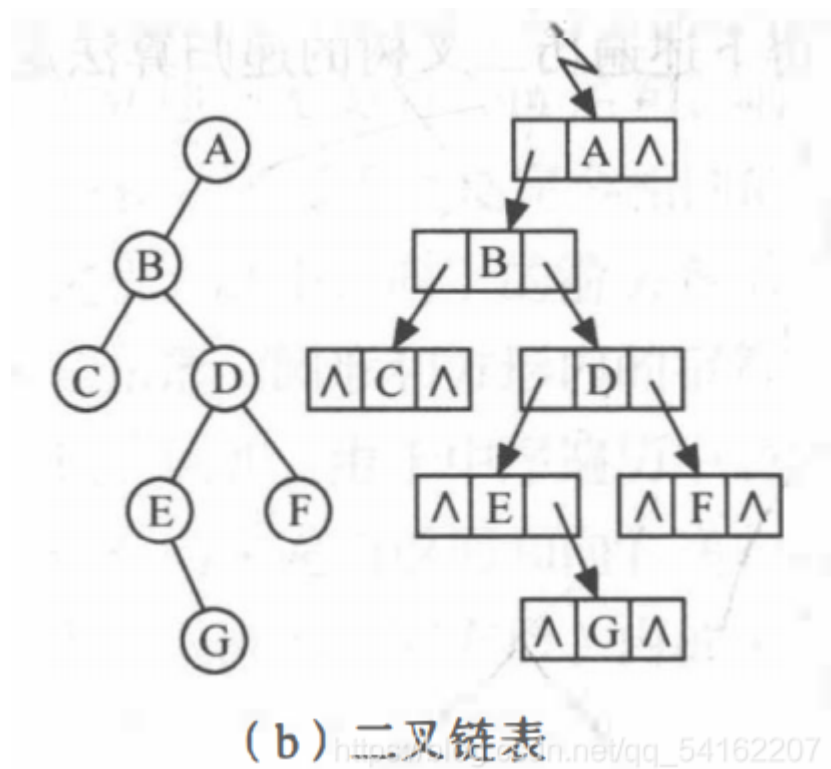
178         NewT = NULL;
179         return;
180     }
181     else {
182         NewT = new BiTNode;
183         NewT->data = T->data; //复制根结点
184         Copy(T->lchild, NewT->lchild); //递归复制左子树
185         Copy(T->rchild, NewT->rchild); //递归复制右子树
186     }
187 }
188 /*-----计算二叉树的深度-----*/
189 int Depth(BiTree T) {
190     //计算二叉树T的深度
191     if (T == NULL) return 0; //如果是空树, 深度为0, 递归结束
192     else {
193         int m, n;
194         m = Depth(T->lchild); //递归计算左子树的深度记为m
195         n = Depth(T->rchild); //递归计算右子树的深度记为n
196         if (m > n) return (m + 1); //二叉树的深度为m与n的较大者加1
197         else return (n + 1);
198     }
199 }
200 /*-----统计二叉树中结点的个数-----*/
201 int NodeCount(BiTree T) {
202     //统计二叉树T中结点的个数
203     if (T == NULL) return 0; //如果是空树, 则结点个数为0, 递归结束
204     else return NodeCount(T->lchild) + NodeCount(T->rchild) + 1;
205     //否则结点个数为左子树的结点个数+右子树的结点个数+1
206 }
207 /*-----运行主函数-----*/
208 int main()
209 {
210     BiTree T1 = new BiTNode;
211     CreateBiTree(T1);
212     InOrderTraverse021(T1);
213     cout << endl;
214     InOrderTraverse022(T1);
215     //BiTree T2 = new BiTNode;
216     //Copy(T1, T2);
217     //InOrderTraverse022(T2);
218     return 0;
219 }
220 //a#b#cdef#####
221 //ABC##DE#G##F###
222

```

测试结果

1 | 输入：ABC##DE#G##F###

先序遍历构建的二叉树：



先序遍历输出结果

1 | 输出：ABCDEGF

中序遍历输出结果

1 | 输出：CBEGDFA

后序遍历输出结果

输出：CGEFDBA