

## Homework 9.27 9.28 9.30

**2.38** 设有一个双向循环链表，每个结点中除有pre,data和next三个域外,还增设了一个访问频度域freq。在链表被起用之前，频度域freq的值均初始化为零，而每当对链表进行一次LOCATE(L,x)的操作后,被访问的结点(即元素值等于x的结点)中的频度域freq的值便增1，同时调整链表中结点之间的次序，使其按访问频度非递增的次序顺序排列，以便始终保持被频繁访问的结点总是靠近表头结点。试编写符合上述要求的LOCATE操作的算法。

类C描述:

```
1 Status LOCATE(DuLinkList &L,int x){
2     //每进行一次操作,被访问的结点freq值加1,同时调整结点次序,使其按freq值非递增排列
3     DuLinkList p,q;
4     if(!L) return ERROR;
5     p = L->next;
6     q = L->next;
7     while(p != L && p->data != x)//遍历找数
8         p = p->next;
9     if(p == L) return FALSE;    //表中无此数
10    p->freq++;                    //频数+1
11    p->pre->next = p->next;
12    p->next->pre = p->pre;        //先删除p结点
13    while(q != L && q->freq >= p->freq)
14        q = q->next;            //辅助指针定位到插入位
15    if(q == L){
16        p->next = q->next;
17        q->next = p;
18        p->pre = q->pre;
19        q->pre = p;
20    }//频数最小为0,所以q总是能找到插入位,如果q循环到头结点,表明原表只剩头结点
21    else{
22        p->next = q->pre->next;
23        q->pre->next = p;
24        p->pre = q->pre;
25        q->pre = p;
26    }//其余情况正常插入
27    return OK;
28 }
```

完整C实现:

```
1 #include<stdio.h>
2 #include<malloc.h>
3 #define OK      1
4 #define ERROR   0
5 #define FALSE   0
6 typedef int Status;
7
8 typedef struct DuLNode{
9     int data;//假设data为整型
```

```

10     int freq;
11     struct DuLNode *pre;
12     struct DuLNode *next;
13 }DuLNode,*DuLinkList;
14
15 Status InitDuList(DuLinkList &L){//初始化双向循环链表
16     L = (DuLinkList)malloc(sizeof(DuLNode));
17     L->next = L;
18     L->pre = L;
19 }
20
21 Status InsertDuList(DuLinkList &L){//插入
22     DuLinkList q=L,p;
23     int n,i = 1,f = 0;
24     printf("请输入结点个数: ");
25     scanf("%d",&n);
26     while(n--){
27         p = (DuLinkList)malloc(sizeof(DuLNode));
28         if(!p) return ERROR;
29         printf("请输入第%d个结点值: ",i++);
30         scanf("%d",&p->data);
31         p->next = q->pre->next;
32         q->pre->next = p;
33         p->pre = q->pre;
34         q->pre = p;
35         q = L;
36         if(!f++)    q->next = p;//循环
37     }
38 }
39
40 Status PrintDuList(DuLinkList L){//遍历打印
41     DuLinkList p = L->next;
42     while(p != L){
43         printf("%d ",p->data);
44         p = p->next;
45     }
46     printf("\n");
47 }
48
49 Status LOCATE(DuLinkList &L,int x){
50     DuLinkList p,q;
51     if(!L) return ERROR;
52     p = L->next;
53     q = L->next;
54     while(p != L && p->data != x)//遍历找数
55         p = p->next;
56     if(p == L) return FALSE;//表中无此数
57     p->freq++;
58     p->pre->next = p->next;
59     p->next->pre = p->pre; //先删除p结点
60     while(q != L && q->freq >= p->freq)
61         q = q->next;
62     if(q == L){
63         p->next = q->next;
64         q->next = p;
65         p->pre = q->pre;
66         q->pre = p;
67     }

```

```

68     else{
69         p->next = q->pre->next;
70         q->pre->next = p;
71         p->pre = q->pre;
72         q->pre = p;
73     }
74     return OK;
75 }
76
77 int main(){
78     DuLinkList L;
79     int x;
80     InitDuList(L);
81     InsertDuList(L);
82     do{
83         printf("序列为:");
84         PrintDuList(L);
85         printf("请输入要访问的值,111结束: ");
86         scanf("%d",&x);
87         LOCATE(L,x);
88     }while(x!=111);
89     return 0;
90 }

```

结果测试:

```

请输入结点个数: 5
请输入第1个结点值: 1
请输入第2个结点值: 2
请输入第3个结点值: 3
请输入第4个结点值: 4
请输入第5个结点值: 5
序列为:1 2 3 4 5
请输入要访问的值,111结束: 4
序列为:4 1 2 3 5
请输入要访问的值,111结束: 5
序列为:4 5 1 2 3
请输入要访问的值,111结束: 5
序列为:5 4 1 2 3
请输入要访问的值,111结束: 3
序列为:5 4 3 1 2
请输入要访问的值,111结束: 3
序列为:5 3 4 1 2
请输入要访问的值,111结束: 3
序列为:3 5 4 1 2
请输入要访问的值,111结束: 111

```

时空分析: InsertDuList、PrintDuList、LOCATE的时空复杂度都为 $O(n)$ ,其中 $n$ 为结点个数。

**3.15** 假设以顺序存储结构实现一个双向栈，即在一维数组的存储空间中存在着两个栈，它们的栈底分别设在数组的两个端点。试编写实现这个双向栈tws的三个操作:初始化inistack(tws),入栈push(tws,i,x)和出栈pop(tws,i)的算法，其中 $i$ 为0或1，用以分别指示设在数组两端的两个栈，并讨论按过程(正/误状态变量可设为变参)或函数设计这些操作算法各有什么优缺点。

类C描述:

```

1  #define STACK_INIT_SIZE 100
2  typedef struct{
3      int top[2];
4      int base[2];
5      SElemType s[STACK_INIT_SIZE];
6  }TwsStack;
7

```

```

8  Status Inistack(TwsStack &tws){
9      //初始化双向栈，base指向数组两侧不可取的位置
10     tws.base[0] = tws.top[0] = -1;
11     tws.base[1] = tws.top[1] = STACK_INIT_SIZE;
12     return OK;
13 }
14
15 Status Push(TwsStack &tws,int i,SElemType e){
16     //入栈
17     if(tws.top[0] + 1 == tws.top[1])    return ERROR;//栈满
18     if(i)    tws.top[1]--;
19     else    tws.top[0]++;
20     tws.s[tws.top[i]] = e;
21     return OK;
22 }
23
24 Status Pop(TwsStack &tws,int i,SElemType &e){
25     //出栈，为方便调试，增加参数e返回出栈元素
26     if(i){
27         if(tws.top[1] == tws.base[1])    return ERROR;//栈空
28         e = tws.s[tws.top[1]];
29         (tws.top[1]++) == NULL;
30     }
31     else{
32         if(tws.top[0] == tws.base[0])    return ERROR;//栈空
33         e = tws.s[tws.top[0]];
34         (tws.top[0]--) == NULL;
35     }
36     return OK;
37 }

```

完整C实现:

```

1  #include<stdio.h>
2
3  #define STACK_INIT_SIZE 100
4  #define OK 1
5  #define ERROR 0
6  typedef int Status;
7  typedef struct{
8     int top[2];
9     int base[2];
10    int s[STACK_INIT_SIZE];
11 }TwsStack;
12
13 Status Inistack(TwsStack &tws){
14     //初始化双向栈，base指向数组两侧不可取的位置
15     tws.base[0] = tws.top[0] = -1;
16     tws.base[1] = tws.top[1] = STACK_INIT_SIZE;
17     return OK;
18 }
19
20 Status Push(TwsStack &tws,int i,int e){
21     //入栈
22     if(tws.top[0] + 1 == tws.top[1])//栈满
23         return ERROR;
24     if(i)    tws.top[1]--;

```

```

25     else    tws.top[0]++;
26     tws.s[tws.top[i]] = e;
27     return OK;
28 }
29
30 Status Pop(TwsStack &tws,int i,int &e){
31     //出栈
32     if(i){
33         if(tws.top[1] == tws.base[1])//栈空
34             return ERROR;
35         e = tws.s[tws.top[1]];
36         (tws.top[1]++) == NULL;
37     }
38     else{
39         if(tws.top[0] == tws.base[0])//栈空
40             return ERROR;
41         e = tws.s[tws.top[0]];
42         (tws.top[0]--) == NULL;
43     }
44     return OK;
45 }
46
47 int main(){
48     TwsStack tws;
49     int i,e = 1,f;
50     Inistack(tws);
51     printf("请输入0号栈元素: \n");
52     scanf("%d",&e);
53     while(e != 111){
54         Push(tws,0,e);
55         scanf("%d",&e);
56     }
57     e = 1;
58     printf("请输入1号栈元素: \n");
59     scanf("%d",&e);
60     while(e != 111){
61         Push(tws,1,e);
62         scanf("%d",&e);
63     }
64     printf("0号栈出栈: \n");
65     while(tws.top[0] != tws.base[0]){
66         Pop(tws,0,f);
67         printf("%d ",f);
68     }
69     printf("\n");
70     printf("1号栈出栈: \n");
71     while(tws.top[1] != tws.base[1]){
72         Pop(tws,1,f);
73         printf("%d ",f);
74     }
75     return 0;
76 }

```

结果测试:

```

请输入0号栈元素:
1 3 5 7 9 111
请输入1号栈元素:
2 4 6 8 10 111
0号栈出栈:
9       7       5       3       1
1号栈出栈:
10      8       6       4       2

```

**时空分析:** inistack、push、pop的时空复杂度都为 $O(1)$ 。

**3.19** 假设一个算术表达式中可以包含三种括号:圆括号“(和)”、方括号 “[和]”和花括号 “{和}”，且这三种括号可按任意的次序嵌套使用(如: ...{...{...}...[...]}...(...))。编写判别给定表达式中所含括号是否正确配对出现的算法(已知表达式已存入数据元素为字符的顺序表中)。

**类C描述:**

```

1  Status MatchStack(Stack S,SqList L){
2      //括号匹配函数，元素已存入字符顺序表。TRUE为匹配，FALSE为不匹配
3      int i = 0;
4      char e;
5      if(L.length <= 0) return ERROR;
6      while(i < L.length){
7          if(L.elem[i] == '(' || L.elem[i] == '[' || L.elem[i] == '{')//左括号入栈
8              Push(S,L.elem[i]);
9          else if(L.elem[i] == ')' || L.elem[i] == ']' || L.elem[i] == '}'){
10             if(S.top == S.base) return FALSE;//出现右括号但栈空，不匹配
11             Pop(S,e);//出栈比对
12             switch(L.elem[i]){
13                 case ')':if(e != '(') return FALSE; break;
14                 case ']':if(e != '[') return FALSE; break;
15                 case '}':if(e != '{') return FALSE; break;
16                 default:break;
17             }
18         }
19         i++;
20     }
21     if(S.top != S.base) return FALSE;//左括号剩余，不匹配
22     return TRUE;
23 }

```

**完整C实现:**

```

1  #include<stdio.h>
2  #include<stdlib.h>
3  #define OK          1
4  #define ERROR       0
5  #define FALSE       0
6  #define TRUE        1
7  #define OVERFLOW    -2
8  #define LIST_INIT_SIZE 100 //初始分配容量
9  #define LISTINCREMENT 10  //分配增量
10 #define STACK_INIT_SIZE 100 //初始分配容量
11 #define STACKINCREMENT 10  //分配增量
12 typedef int Status;
13
14 typedef struct{
15     char *elem; //存储空间基址

```

```

16     int         length;      //当前长度
17     int         listsize;   //当前分配容量
18 }SqList;
19
20 typedef struct{
21     char *top;
22     char *base;
23     int stacksize;
24 }Stack;
25
26 Status InitList(SqList &L){//建表
27     L.elem = (char*)malloc(LIST_INIT_SIZE*sizeof(char));
28     if(!L.elem){
29         printf("建表出错\n");
30         exit(OVERFLOW);
31     }
32     L.length = 0;
33     L.listsize = LIST_INIT_SIZE;
34     return OK;
35 }
36
37 Status InputList(SqList &L,int n){//输入
38     int i;
39     if(n < 1 || n > L.listsize){
40         printf("输入出错\n");
41         return ERROR;
42     }
43     printf("请输入元素: \n");
44     getchar();
45     for(i = 0;i < n;i++)    scanf("%c",&L.elem[i]);
46     L.length = n;
47     return OK;
48 }
49
50 Status OutputList(SqList L,int n){//输出
51     int i;
52     if(n < 1 || n > L.listsize) return ERROR;
53     if(L.length == 0) return ERROR;
54     printf("元素为: ");
55     for(i = 0;i < n;i++)
56         printf("%c ",L.elem[i]);
57     return OK;
58 }
59
60 Status InitStack(Stack &S){//操作数栈
61     S.base=(char *)malloc(STACK_INIT_SIZE*sizeof(char));
62     if(!S.base){                                //存储分配失败
63         printf("分配空间时出错。 \n");
64         exit(OVERFLOW);
65     }
66     S.top = S.base;                                //初始化指针
67     S.stacksize = STACK_INIT_SIZE;
68 }
69
70 Status Push(Stack &S,char e){
71     if(S.top - S.base >= S.stacksize){
72         S.base=(char *)realloc(S.base,
73             (S.stacksize+STACKINCREMENT)*sizeof(char));

```

```

74         if(!S.base){
75             printf("追加空间时出错。\\n");
76             exit(OVERFLOW);
77         }
78         S.top = S.base + S.stacksize;
79         S.stacksize += STACKINCREMENT;
80     }
81     *S.top++ = e;
82 }
83
84 Status Pop(Stack &S, char &e){//出栈
85     if(S.top == S.base){
86         printf("出栈时出错。\\n");
87         return ERROR;
88     }
89     e = *--S.top;
90 }
91
92
93 Status MatchStack(Stack S, SqList L){
94     int i = 0;
95     char e;
96     if(L.length <= 0) return ERROR;
97     while(i < L.length){
98         if(L.elem[i] == '(' || L.elem[i] == '[' || L.elem[i] == '{')
99             Push(S, L.elem[i]);
100         else if(L.elem[i] == ')' || L.elem[i] == ']' || L.elem[i] == '}'){
101             if(S.top == S.base) return FALSE;//不匹配
102             Pop(S, e);
103             switch(L.elem[i]){
104                 case ')': if(e != '(') return FALSE; break;
105                 case ']': if(e != '[') return FALSE; break;
106                 case '}': if(e != '{') return FALSE; break;
107                 default: break;
108             }
109         }
110         i++;
111     }
112     if(S.top != S.base) return FALSE;
113     return TRUE;
114 }
115
116 int main(){
117     Stack S;
118     SqList L;
119     int n;
120     InitStack(S);
121     InitList(L);
122     printf("请输入元素个数: ");
123     scanf("%d", &n);
124     InputList(L, n);
125     OutputList(L, n);
126     if(MatchStack(S, L) == 1)
127         printf("括号匹配");
128     else printf("括号不匹配");
129     return 0;
130 }

```



## 结果测试:

```

请输入元素个数: 10
请输入元素, 以回车隔开:
12(4){[7]}
元素为: 1      2      (      4      )      {      [      7      ]      }      括号匹配

```

```

请输入元素个数: 8
请输入元素, 以回车隔开:
[23{1(7)
元素为: [      2      3      {      ]      (      7      )      括号不匹配

```

**时空分析:** InputList、OutputList、MatchStack的时空复杂度都为 $O(n)$ , Push、Pop的时空复杂度都为 $O(1)$ , 其中 $n$ 为元素个数。

3.27 已知Ackerman函数的定义如下:

$$akm(m, n) = \begin{cases} n + 1 & m = 0 \\ akm(m - 1, 1) & m \neq 0, n = 0 \\ akm(m - 1, akm(m, n - 1)) & m \neq 0, n \neq 0 \end{cases}$$

(1) 写出递归算法;

(2) 根据递归算法, 画出求 $akm(2,1)$ 时栈的变化过程。

(1)

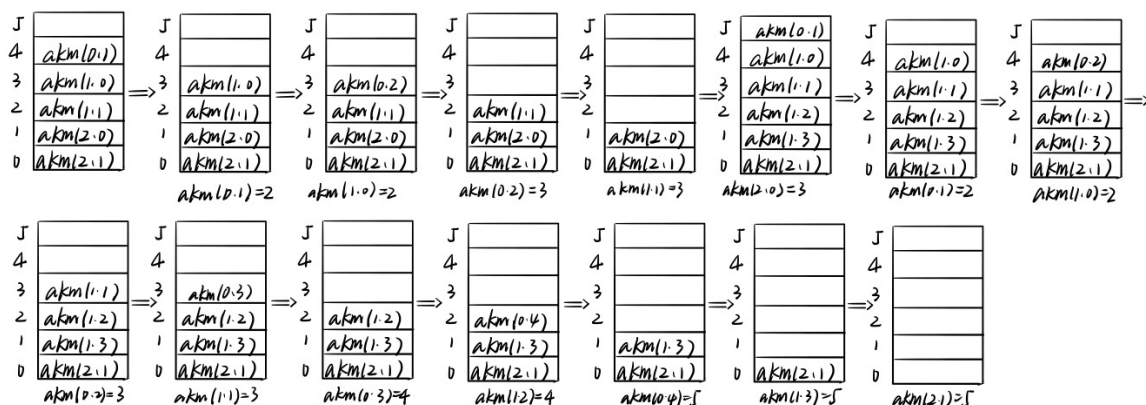
## 类C描述:

```

1 Status Ackerman(int m, int n){
2     if(m == 0) return (n + 1);
3     else if(n == 0) return Ackerman(m - 1, n);
4     else return Ackerman(m - 1, Ackerman(m, n - 1));
5 }

```

(2)



3.29 如果希望循环队列中的元素都能得到利用,则需设置一个标志域tag,并以tag的值为0或1来区分,尾指针和头指针值相同时的队列状态是“空”还是“满”。试编写与此结构相应的入队列和出队列的算法,并从时间和空间角度讨论设标志和不设标志这两种方法的使用范围(如当循环队列容量较小而队列中每个元素占的空间较多时,哪一种方法较好)。

## 类C描述:

```

1 Status EnQueue(Queue &Q, QElemType e){//入队
2     if(Q.front == Q.rear && Q.tag == 1) return ERROR;//队满

```

```
3     Q.base[Q.rear] == e;
4     Q.rear = (Q.rear + 1) % MAXQSIZE;
5     if(Q.front == Q.rear)    Q.tag = 1;
6     return OK;
7 }
8
9 Status DeQueue(Queue &Q, QElemType &e){//出队
10     if(Q.front == Q.rear && Q.tag == 0) return ERROR;//队空
11     e = Q.base[Q.rear];
12     Q.front = (Q.front + 1) % MAXQSIZE;
13     if(Q.front == Q.rear)    Q.tag = 0;
14     return OK;
15 }
```

当循环队列容量较小而队列中每个元素占的空间较多时，用tag表示可以节省更多存储空间，但也因为tag的关系，运行时间相应变长。

---

2020/9/29 16:38