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:

```
Status LOCATE(DuLinkList &L,int x){
//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
    DuLinkList p,q;
    if(!L)  return ERROR;
    p = L->next;
    q = L->next;
    while(p != L && p->data != x)//遍历找数
        p = p->next;
    if(p == L)  return FALSE;    //表中无此数
    p->freq++;                   //频数+1
    p->pre->next = p->next;
    p->next->pre = p->pre;       //先删除p结点
    while(q != L && q->freq >= p->freq)
        q = q->next;             //辅助指针定位到插入位
    if(q == L){
        p->next = q->next;
        q->next = p;
        p->pre = q->pre;
        q->pre = p;
    }//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
    else{
        p->next = q->pre->next;
        q->pre->next = p;
        p->pre = q->pre;
        q->pre = p;
    }//其余情况正常插入
    return OK;
}
```

完整C实现:

```
#include<stdio.h>
#include<malloc.h>
#define OK      1
#define ERROR   0
#define FALSE   0
typedef int Status;

typedef struct DuLNode{
    int data;//假设data为整型
```

```
    int freq;
    struct DuLNode *pre;
    struct DuLNode *next;
}DuLNode,*DuLinkList;

Status InitDuList(DuLinkList &L){//初始化双向循环链表
    L = (DuLinkList)malloc(sizeof(DuLNode));
    L->next = L;
    L->pre = L;
}

Status InsertDuList(DuLinkList &L){//插入
    DuLinkList q=L,p;
    int n,i = 1,f = 0;
    printf("请输入结点个数：");
    scanf("%d",&n);
    while(n--){
        p = (DuLinkList)malloc(sizeof(DuLNode));
        if(!p)  return ERROR;
        printf("请输入第%d个结点值：",i++);
        scanf("%d",&p->data);
        p->next = q->pre->next;
        q->pre->next = p;
        p->pre = q->pre;
        q->pre = p;
        q = L;
        if(!f++)    q->next = p;//循环
    }
}

Status PrintDuList(DuLinkList L){//遍历打印
    DuLinkList p = L->next;
    while(p != L){
        printf("%d ",p->data);
        p = p->next;
    }
    printf("\n");
}

Status LOCATE(DuLinkList &L,int x){
    DuLinkList p,q;
    if(!L) return ERROR;
    p = L->next;
    q = L->next;
    while(p != L && p->data != x)//遍历找数
        p = p->next;
    if(p == L) return FALSE;//表中无此数
    p->freq++;
    p->pre->next = p->next;
    p->next->pre = p->pre; //先删除p结点
    while(q != L && q->freq >= p->freq)
        q = q->next;
    if(q == L){
        p->next = q->next;
        q->next = p;
        p->pre = q->pre;
        q->pre = p;
    }
```

```
    else{
        p->next = q->pre->next;
        q->pre->next = p;
        p->pre = q->pre;
        q->pre = p;
    }
    return OK;
}

int main(){
    DuLinkList L;
    int x;
    InitDuList(L);
    InsertDuList(L);
    do{
        printf("序列为:");
        PrintDuList(L);
        printf("请输入要访问的值,111结束：");
        scanf("%d",&x);
        LOCATE(L,x);
    }while(x!=111);
    return 0;
}
```

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：

```
#define STACK_INIT_SIZE 100
typedef struct{
    int top[2];
    int base[2];
    SElemType s[STACK_INIT_SIZE];
}TwsStack;

```

```
Status Inistack(TwsStack &tws){
    //初始化双向栈，base指向数组两侧不可取的位置
    tws.base[0] = tws.top[0] = -1;
    tws.base[1] = tws.top[1] = STACK_INIT_SIZE;
    return OK;
}

Status Push(TwsStack &tws,int i,SElemType e){
    //入栈
    if(tws.top[0] + 1 == tws.top[1])    return ERROR;//栈满
    if(i)   tws.top[1]--;
    else    tws.top[0]++;
    tws.s[tws.top[i]] = e;
    return OK;
}

Status Pop(TwsStack &tws,int i,SElemType &e){
    //出栈，为方便调试，增加参数e返回出栈元素
    if(i){
        if(tws.top[1] == tws.base[1])   return ERROR;//栈空
        e = tws.s[tws.top[1]];
        (tws.top[1]++) == NULL;
    }
    else{
        if(tws.top[0] == tws.base[0])   return ERROR;//栈空
        e = tws.s[tws.top[0]];
        (tws.top[0]--) == NULL;
    }
    return OK;
}
```

完整C实现：

```
#include<stdio.h>

#define STACK_INIT_SIZE 100
#define OK 1
#define ERROR 0
typedef int Status;
typedef struct{
    int top[2];
    int base[2];
    int s[STACK_INIT_SIZE];
}TwsStack;

Status Inistack(TwsStack &tws){
    //初始化双向栈，base指向数组两侧不可取的位置
    tws.base[0] = tws.top[0] = -1;
    tws.base[1] = tws.top[1] = STACK_INIT_SIZE;
    return OK;
}

Status Push(TwsStack &tws,int i,int e){
    //入栈
    if(tws.top[0] + 1 == tws.top[1])//栈满
        return ERROR;
    if(i)   tws.top[1]--;
```

```
    else    tws.top[0]++;
    tws.s[tws.top[i]] = e;
    return OK;
}

Status Pop(TwsStack &tws,int i,int &e){
    //出栈
    if(i){
        if(tws.top[1] == tws.base[1])//栈空
            return ERROR;
        e = tws.s[tws.top[1]];
        (tws.top[1]++) == NULL;
    }
    else{
        if(tws.top[0] == tws.base[0])//栈空
            return ERROR;
        e = tws.s[tws.top[0]];
        (tws.top[0]--) == NULL;
    }
    return OK;
}

int main(){
    TwsStack tws;
    int i,e = 1,f;
    Inistack(tws);
    printf("请输入0号栈元素：\n");
    scanf("%d",&e);
    while(e != 111){
        Push(tws,0,e);
        scanf("%d",&e);
    }
    e = 1;
    printf("请输入1号栈元素：\n");
    scanf("%d",&e);
    while(e != 111){
        Push(tws,1,e);
        scanf("%d",&e);
    }
    printf("0号栈出栈：\n");
    while(tws.top[0] != tws.base[0]){
        Pop(tws,0,f);
        printf("%d  ",f);
    }
    printf("\n");
    printf("1号栈出栈：\n");
    while(tws.top[1] != tws.base[1]){
        Pop(tws,1,f);
        printf("%d  ",f);
    }
    return 0;
}
```

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：

```
Status MatchStack(Stack S,SqList L){
    //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
    int i = 0;
    char e;
    if(L.length <= 0)   return ERROR;
    while(i < L.length){
        if(L.elem[i] == '('||L.elem[i] == '['||L.elem[i] == '{')//左括号入栈
            Push(S,L.elem[i]);
        else if(L.elem[i] == ')'||L.elem[i] == ']'||L.elem[i] == '}'){
                if(S.top == S.base) return FALSE;//出现右括号但栈空，不匹配
                Pop(S,e);//出栈比对
                switch(L.elem[i]){
                    case ')':if(e != '(')   return FALSE; break;
                    case ']':if(e != '[')   return FALSE; break;
                    case '}':if(e != '{')   return FALSE; break;
                    default:break;
                }
        }
        i++;
    }
    if(S.top != S.base) return FALSE;//左括号剩余，不匹配
    return TRUE;
}
```

完整C实现：

```
#include<stdio.h>
#include<stdlib.h>
#define OK          1
#define ERROR       0
#define FALSE       0
#define TRUE        1
#define OVERFLOW    -2
#define LIST_INIT_SIZE  100     //初始分配容量
#define LISTINCREMENT   10      //分配增量
#define STACK_INIT_SIZE 100     //初始分配容量
#define STACKINCREMENT  10      //分配增量
typedef int Status;

typedef struct{
    char        *elem;      //存储空间基址
```

```
    int         length;     //当前长度
    int         listsize;   //当前分配容量
}SqList;

typedef struct{
    char *top;
    char *base;
    int stacksize;
}Stack;

Status InitList(SqList &L){//建表
    L.elem = (char*)malloc(LIST_INIT_SIZE*sizeof(char));
    if(!L.elem){
        printf("建表出错\n");
        exit(OVERFLOW);
    }
    L.length = 0;
    L.listsize = LIST_INIT_SIZE;
    return OK;
}

Status InputList(SqList &L,int n){//输入
    int i;
    if(n < 1 || n > L.listsize){
        printf("输入出错\n");
        return ERROR;
    }
    printf("请输入元素：\n");
    getchar();
    for(i = 0;i < n;i++)    scanf("%c",&L.elem[i]);
    L.length = n;
    return OK;
}

Status OutputList(SqList L,int n){//输出
    int i;
    if(n < 1 || n > L.listsize) return ERROR;
    if(L.length == 0) return ERROR;
    printf("元素为：");
    for(i = 0;i < n;i++)
        printf("%c  ",L.elem[i]);
    return OK;
}

Status InitStack(Stack &S){//操作数栈
    S.base=(char *)malloc(STACK_INIT_SIZE*sizeof(char));
    if(!S.base){                        //存储分配失败
        printf("分配空间时出错。\n");
        exit(OVERFLOW);
    }
    S.top = S.base;                     //初始化指针
    S.stacksize = STACK_INIT_SIZE;
}

Status Push(Stack &S,char e){
    if(S.top - S.base >= S.stacksize){
        S.base=(char *)realloc(S.base,
                    (S.stacksize+STACKINCREMENT)*sizeof(char));
```

```
        if(!S.base){
            printf("追加空间时出错。\n");
            exit(OVERFLOW);
        }
        S.top = S.base + S.stacksize;
        S.stacksize += STACKINCREMENT;
    }
    *S.top++ = e;
}

Status Pop(Stack &S,char &e){//出栈
    if(S.top == S.base){
        printf("出栈时出错。\n");
        return ERROR;
    }
    e = *--S.top;
}


Status MatchStack(Stack S,SqList L){
    int i = 0;
    char e;
    if(L.length <= 0)   return ERROR;
    while(i < L.length){
        if(L.elem[i] == '('||L.elem[i] == '['||L.elem[i] == '{')
            Push(S,L.elem[i]);
        else if(L.elem[i] == ')'||L.elem[i] == ']'||L.elem[i] == '}'){
                if(S.top == S.base) return FALSE;//不匹配
                Pop(S,e);
                switch(L.elem[i]){
                    case ')':if(e != '(')   return FALSE; break;
                    case ']':if(e != '[')   return FALSE; break;
                    case '}':if(e != '{')   return FALSE; break;
                    default:break;
                }
        }
        i++;
    }
    if(S.top != S.base) return FALSE;
    return TRUE;
}

int main(){
    Stack S;
    SqList L;
    int n;
    InitStack(S);
    InitList(L);
    printf("请输入元素个数：");
    scanf("%d",&n);
    InputList(L,n);
    OutputList(L,n);
    if(MatchStack(S,L)==1)
        printf("括号匹配");
    else    printf("括号不匹配");
    return 0;
}
```

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{](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；

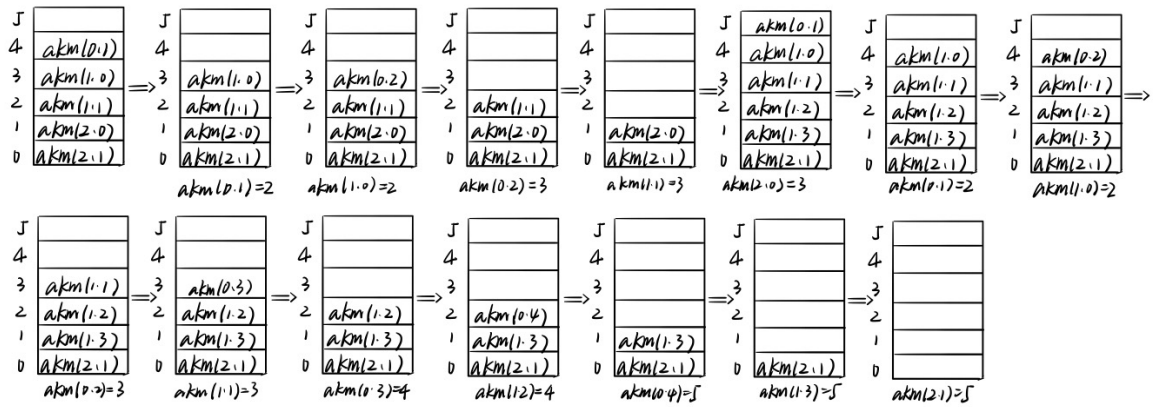
（2）根据递归算法，画出求akm(2,1)时栈的变化过程。

(1)

类C描述：

```
Status Ackerman(int m,int n){
    if(m == 0)  return (n + 1);
    else if(n == 0) return Ackerman(m - 1,n);
    else    return Ackerman(m - 1,Ackerman(m,n - 1));
}
```

(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：

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

```
    Q.base[Q.rear] == e;
    Q.rear = (Q.rear + 1) % MAXQSIZE;
    if(Q.front == Q.rear)   Q.tag = 1;
    return OK;
}

Status DeQueue(Queue &Q,QElemType &e){//出队
    if(Q.front == Q.rear && Q.tag == 0) return ERROR;//队空
    e =  Q.base[Q.rear];
    Q.front = (Q.front + 1) % MAXQSIZE;
    if(Q.front == Q.rear)   Q.tag = 0;
    return OK;
}
```

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