1.数据结构

1.1基础概念

数据元素：组成数据的基本单位

数据对象：性质相同的数据元素的集合，是数据的一个子集

数据结构：相互之间存在一种或多种特定关系的数据元素集合

数据类型：一组性质相同的值的集合；定义在这个集合内的一组操作的总称

抽象数据类型（ADT）：定义了一个数据对象，其中各元素间的关系和一组处理数据的操作。包括定义和实现两部分，其相互独立。

1.2数据结构

基本数据结构：集合、线性、树型、网状

存储结构：逻辑结构在计算机中存储映像，是逻辑结构在计算中的实现，包括元素表示和关系表示

存储结构是逻辑关系与元素本身的映像；逻辑结构是数据结构的抽象，存储结构时数据结构的实现。存储/逻辑结构综合建立了数据元素之间的结构关系。

数据元素在计算机中的表示方法：

顺序映像（如数组）/非顺序映像（链表）

数据结构内容的三部分：逻辑结构，存储结构，运算集合。

1.3算法与算法描述

算法特性：

有限性，确定性，输入，输出，可行性

算法设计要求：

正确性，可读性，健壮性，高效率和低存储量

性能评价：

执行时间、存储空间

2.线性表

2.1描述

```
typedef struct SeqList
{
    Elemtype elem[100];
    int last;
}SeqList;
//自定义类型seqlist
//变量定义方式有两种：
Seqlist L（L变量为seqlist类型）
"L.elem[i-1]"//访问序号i的元素
Seqlist *L（L变量为指向SeqList类型的指针变量）
"L->elem[i-1]"//访问序号i的元素
```

2.2查找

```
int Locate(Seqlist L,Elemtype k)
{
    i=0;
    while ((i<=L.last)&&(L.elem[i]!=k))
        i++;
    if (i<=L.last)
        return(i+1);
    else
        return -1;
}
```

2.3插入

```
int insert(SeqList *L,int i,ElemType e)
{
    int k;
    if((i<1)||(i>L->last+2))
        return ERROR;
    if(L->last>=maxsize - 1)
        return ERROR;
    for(k=L->last;k>=i-1;k--)
        L->elem[k+1]=L->elem[k];//元素后移
    L->elem[i-1]=e;
    L->last++;
    return OK;
}
```

2.4删除

```
int delete(Seqlist *L,int i,ElemType e)
{
    int k;
    if ((i<1)||(i>L->last+1))
        return ERROR;
    *e=L->elem[i-1];//把要删除的元素存放到e所指向的变量中
    for(k=i;k<L->last;k++)
        L->elem[k-1]=L->elem[k];
    L->last--;
    return OK;
}
```

2.5合并

```
void merge(SeqList *LA,SeqList *LB,SeqList *LC)
{
    int i=0,j=0,k=0,l;
    while(i<=LA->last&&j<=LB->last){
        if(LA->elem[i]<=LB->elem[j]){
            LC->elem[k]=LA->elem[i];
            j++;k++;
        }
        else if(LA->elem[i]>=LB->elem[i]){
            LC->elem[k]=LB->elem[i];
            j++;k++;
        }
    }
    while(i<=LA->last){
        LC->elem[k]=LA->elem[i];
        i++;k++;
    }
    while(j<=LB->last){
        LC->elem[k]=LB->elem[j];
        j++;k++;
    }
    LC->last=LA->last+LB->last+1;

}
```

3.链表

3.0 疑难问题

LinkList L;&L;*L的区别：
在链表定义中，Node与*LinkList等价，即Node*被重新命名为LinkList（LinkList为指向Node的指针），*L指向整个结构体；
而LinkList *L,L则是指向结构体的指针的指针，此时*L是指向Node的指针，**L才等价于NODE结构体。

对于LinkList L: L是指向定义的node结构体的指针,可以用->运算符来访问结构体成员,即L->elem,而(*L)就是个Node型的结构体了,可以用点运算符访问该结构体成员,即(*L).elem;

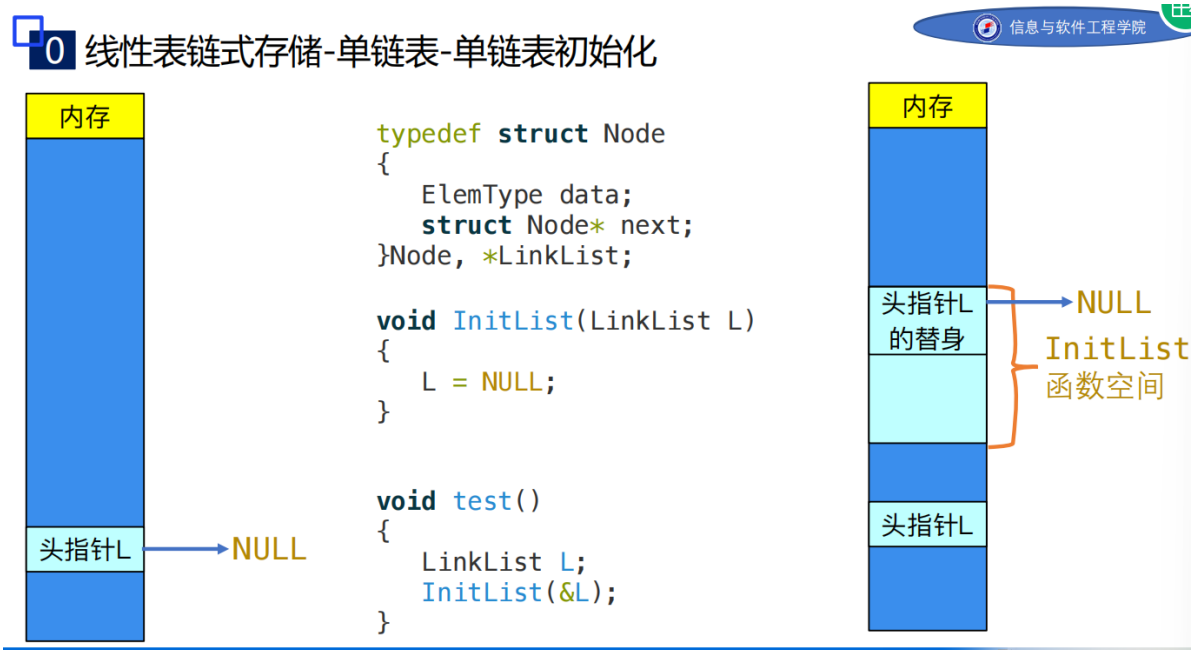
对于LinkList *L:L是指向定义的Node结构体指针的指针,所以(*L)是指向Node结构体的指针,可以用->运算符来访问结构体成员,即(*L)->elem,当然,(**L)就是Node型结构体了,所以可以用点运算符来访问结构体成员,即(**L).elem;

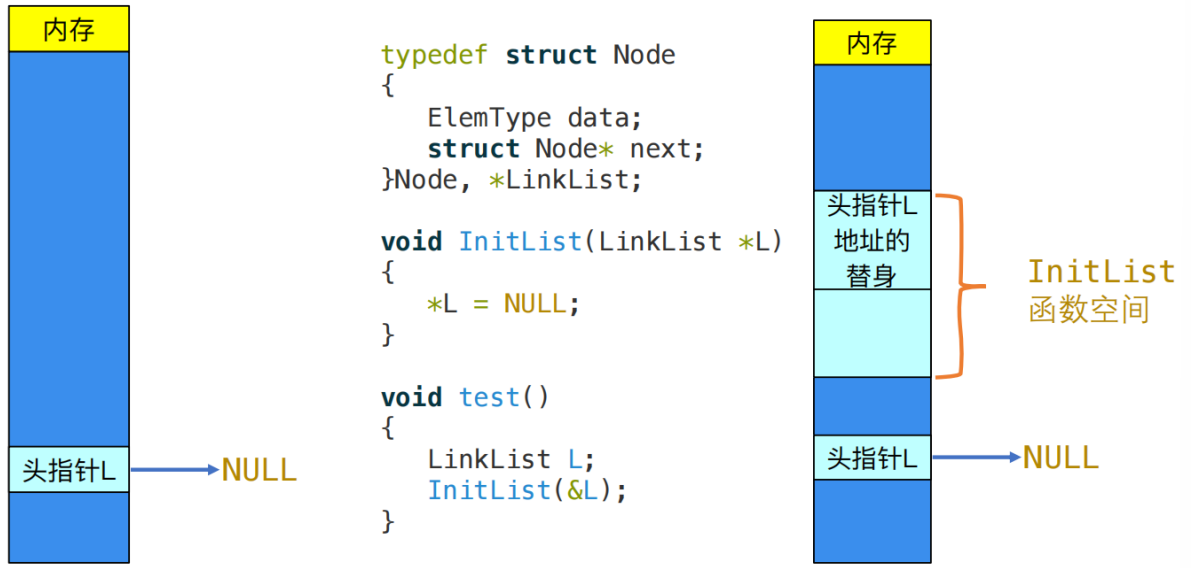
https://blog.csdn.net/gsggd/article/details/124275958

https://zhuanlan.zhihu.com/p/421431276

https://blog.csdn.net/qq_61959780/article/details/127298542

初始化链表的区别:

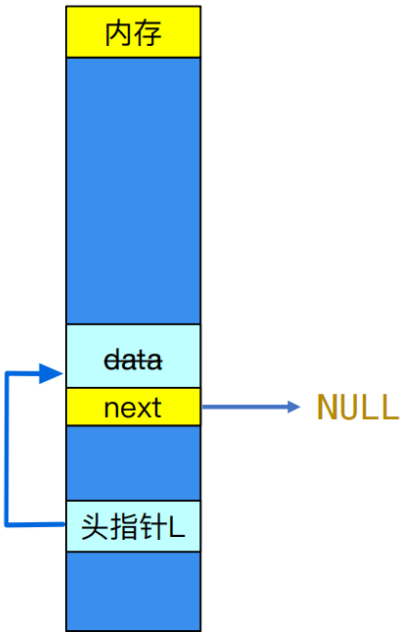




```
typedef struct Node
{
    ElemType data;
    struct Node* next;
}Node, *LinkList;

void InitList(LinkList* L)
{
    *L = (LinkList)malloc(sizeof(Node));
    (*L)->next = NULL;
}

void test()
{
    LinkList L;
    InitList(&L);
}
```



3.1链表分类

3.1.1单链表

每个节点由数据域和指针域组成，指针域存储数据元素后继的位置。每个结点的唯一后继依靠一个结点指针维持。

头指针指向第一个结点。

最后一个结点的指针域为NULL。

```
typedef struct Node//结点类型定义
{
    ElemType data;
    struct Node *next;
}Node , *LinkList;//结构指针类型
LinkList L;//*L为单链表的头指针
```

3.1.1.1 链表长度

```
int Length(LinkList L)
{
    Node *p;
    p = L->next;i=0;
    while(p!=NULL)
    {
        p=p->next;i++;
    }
    return i;
}
```

3.1.1.2 空表

```
InitList(LinkList *L)
{
    L = (LinkList)malloc(sizeof(Node));
    (*L)->next = NULL;
}
```

3.1.1.3 头插

```
LinkList CreateFromHead(LinkList L)
{
    Node *s;
    char c;int flag = 1;//flag起标志作用，初始值为1，当输入$时flag为0，建表结束
    while(flag){
        c=getchar();
        if(c!=$)
        {
            s=(Node*)malloc(sizeof(Node));//新结点
            s->data=c;
            s->next=L->next;
            L->next=s;
        }
        else{flag=o;}
    }
}
```

3.1.1.4 尾插

```
LinkList CreateFromTail(LinkList L)
{
    LinkList L;Node *r,*s;
    int flag = 1;
    r=L;//*r指针始终指向链表的尾部
    while(flag){
        c=getchar();
        if(c!=$){
            s=(Node*)malloc(sizeof(Node));
            s->data=c;
            r->next=s;
            r=s;
        }
        else{
            flag = 0;
            r->next = NULL;
        }
    }
}
```

3.1.1.5 查找

```
//按结点查找
Node* Find(LinkList L,int i)
{
    int j;
    Node *p;
```

```
    p=L;j=0;
    while((p->next!=NULL)&&(j<i)){
        p=p->next;j++;
    }
    if(i==j){return p;}
    else{return NULL;}
}
//按值查找
Node* Find(LinkList L,ElemType c){
    Node *p;
    p=L->next;
    while(p!=NULL){
        if(p->data!=c){
            p=p->next;
        }
        else break;
    }
    return p;
}
```

3.1.1.6 插入

```
LinkList Insert(LinkList L,int i,ElemTyoe key){
    int k;
    Node *pre,*s;
    pre=L;k=0;
    while((pre!=NULL)&&(k<i-1)){
        pre=pre->next;k++;
    }
    if(!pre){
        return ERROR;
    }
    s=(Node*)malloc(sizeof(Node));
    s->data=key;
    s->next=pre->next;
    pre->next=s;
    return OK;
}
```

3.1.1.7 删除

```
int DelList(LinkList L,int i,ElemType *e){
    Node *pre,*r;
    int k;
    pre=L;k=0;
    while(pre->next!=NULL && k<i-1){
        k+=1;
        pre=pre->next;
    }
    if((pre->next)==NULL){
        printf("结点删除位置不合理");
            return ERROR;
    }
    r=pre->next;
```

```
    pre->next=pre->next->next;
    *e = r->data;
    free(r);
    return OK;
}
```

3.1.1.8 合并

```
LinkList Merge(LinkList LA;LinkList LB){
    Node *a,*b;
    LinkList LC;
    a=LA->next;
    b=LB->next;
    LC=LA;LC->next=NULL;
    r=LC;
    while(a!=NULL && b!=NULL){
        if(a->data<=b->data){
            r->next=a;r=a;a=a->next;
        }
        else{
            r->next=b;r=b;r=r->next;
        }
        if(a){
            r->next=a;
        }
        else(){
            r->next=b;
        }
        free(LB);
        return(LC);
    }
}
```

3.1.2 循环链表

最后一个结点指向头结点：p->next==L

3.1.2.1 初始化

```
void initClinklist(LinkList *CL)
{
    *CL=(LinkList)malloc(sizeof(Node));
    (*CL)->next = *CL;
}
```

3.1.2.2 建立新结点

```
void Createlinklist(LinkList CL)
{
    Node *s,*rear;
    char c;
    rear = CL;
    c=getchar();
```

```
    while(c!='$')
    {
        s=(Node*)malloc(sizeof(Node));
        s->data = c;
        rear->next =s;
        rear = s;
        c=getchar();
    }
    rear->next =CL;
}
```

3.1.2.3 循环链表的合并

先找到两个链表的尾，并分别由指针p、q指向它们,然后将第一个链表的尾与第二个表的第一个结点链接起来,修改第二个表的尾q，使它的链域指向第一个表的头结点.

```
LinkList merge(LinkList LA,LinkList LB)
{
    Node *p,*q;
    p=LA;q=LB;
    while(p->next!=LA)
        p=p->next;
    while(q->next!=LB)
        q=q->next;
    q->next = LA;
    p->next = LB->next;
    free(LB);
    return(LA);
}
```

3.1.2.4 指针合并

```
LinkList hmerge(LinkList LA,LinkList LB){
    Node *p,*q;
    p=LA;q=LB;
    while(p->next!=LA)
        p=p->next;
    while(q->next!=LB)
        q=q->next;
    q->next=LA;//B尾指针指向LA的头
    p->next=LB->next;//A的尾指针指向B的第一个结点
    free(LB);
    return LA;
}

//尾指针合并，时间复杂度O（1）
LinkList tmerge(LinkList LA,LinkList LB)
{
    Node *p;
    p=LA->next;
    LA->next=LB->next->next;
    free(LB->next);
    LB->next=p;
    return LB;
}
```

3.1.3 双向链表

3.1.3.1 定义

```
typdef struct Dnode{
    ElemTyepe data;
    struct DNode *prior, *next;
}DNode, *DoubleList;
//从任意结点出发可以快速找到其前驱结点和后继结点
//从任意结点出发可以访问其他结点
```

3.1.3.2 插入与删除

```
int DlinkIns(DoubleList L,int i,Elemtype e)
{
    DNode *s,*p;
    p=L;k=0;
    while(p->next!=L&&k<i){
        p=p->next;
        k++;
    }
    if(p->next == L){
        return ERROR;
    }
    s=(DNode*)malloc(sizeof(Node));
    if(s){
        s->data = e;
        s->prior = p->prior;
        p->prior->next = s;
        s->next = p;
        p->prior = s;
        return OK;
    }
}
//上面是前插，后插的基本思想与前插一致
s->next = p->next;
p->next->prior = s;
s->prior = p;
p->next =s;
//删除只在最后的指针操作有区别
e=p->data;
p->prior->next=p->next;
p->next->prior=p->prior;
free(p);
```

4.栈

4.1顺序栈

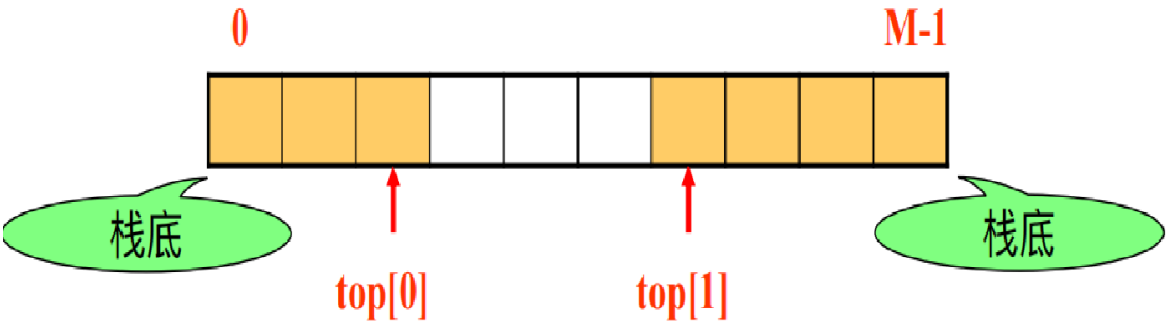
4.1.1算法

```
//顺序栈
#define Stack_Size 50
typedef struct{
    StackElementType elem[Stack_Size];
    int top;//用来存放栈顶元素的下标
}Seqstack;
//top总是指向栈顶元素，初始值为-1；top=Maxsize-1栈满。
//初始化
void InitStach(SeqStack *S){
    S->top = -1;
}
//状态判定
int IsEmpty(SeqStack *S){
    return(S->top==-1?TRUE:FALSE);
}
int IsFull(SeqStack *S){
    return(S->top==Stack_Size-1?TRUE:FALSE);
}
//入栈
int Push(SeqStack *S,StackElementType ){
    if(S->top==Stack_Size-1)
        return FALSE;
    S->top++;
    S->elem[S->top]=x;
    return TRUE;
}
int Pop(SeqStack *S,StackElementType *x){
    if(S->top==-1)
        return FALSE;
    else{
        *x=S->elem[S->top];
        S->top--;
        return TRUE;
    }
}
int GetStackPop(SeqStack *S,StackElementType *x){
    if(S->top==-1)
        return FALSE;
    else{
        *x=S->elem[S->top];
        return TRUE;
    }
}
```

4.1.2 共享栈

```
//共享栈定义
#define M 100
typedef struct{
    StackElementType Stack[M];
    StackElementType top[2];
}DqStack;
//入栈
```

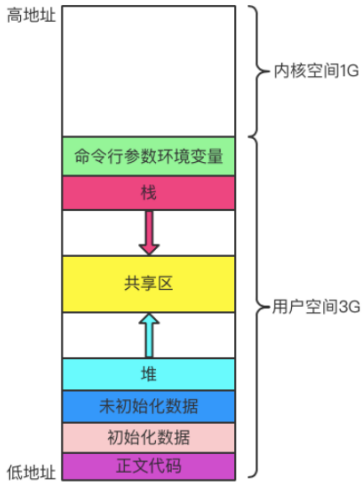
```
int Push(DqStack *S,StackElementType x,int i){
    if(S->top[0]+1==S->top[1])//满栈
        return(FALSE);
    switch(i){
        case 0:
            S->top[0]++;
            S->Stack[S->top[0]]=x;
            break;
        case 1:
            S->top[1]--;
            S->Stack[S->top[1]]=x;
            break;
        default:return FALSE;
    }
    return TRUE;
}
//出栈
int Pop(DqStack *S,StackElementType *x,int i){
    switch(i){
        case 0:
            if(S->top[0]==-1)
                return(FALSE);
            *x=S->Stack[S->top[0]];
            S->top[0]--;
            break;
        case 1:
            if(S->top[1]==M)
                return FALSE;
            *x=S->Stack[S->top[1]];
            S->top[1]++;
            break;
        default:return FALSE;
    }
    return TRUE;
}
```



1 栈实现-顺序栈-共享栈典型场景

栈：

执行期间**编译器**自动分配，编译器用它实现函数调用：调用函数时，栈增长；函数返回时，栈收缩。局部变量、函数参数、返回数据、返回地址等放在栈中



堆：

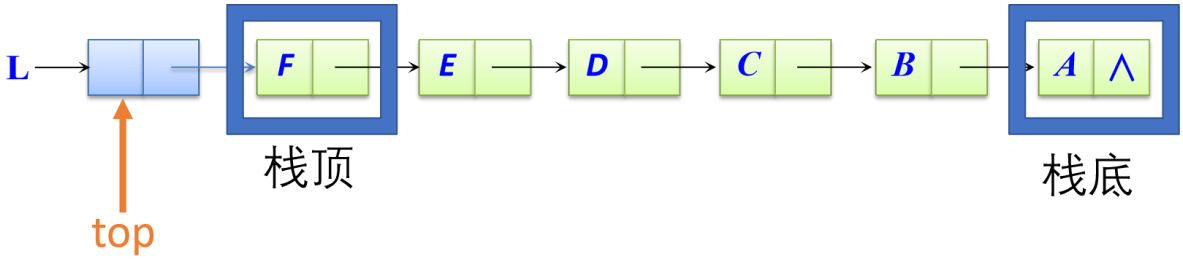
动态储存分配器维护着的一个进程的虚拟存储器区域。一般由程序员分配释放（堆在操作系统对进程初始化的时候分配），若程序员不释放，程序结束时可能由OS回收，每个进程，内核都维护着一个变量brk指向堆顶。

4.2 链栈

4.2.1 实现

链栈，即使用链表存储的栈。

```
typedef struct node{
    StackElementType data;
    struct node *next;//带头结点的单链表
}LinkStackNode;
typedef LinkStackNode *LinkStack;
//下图表示链栈的结构，其中top为栈顶指针，始终指向当前栈顶元素之前的头结点，当top->next==NULL时为空栈
//链栈在使用完毕之后应该释放空间
```




```
//入栈，流程与单链表类似
int Push(LinkStack top,StackElemnetType x){
    LinkStackNode *temp;
    temp=(LinkStackNode*)malloc(sizeof(LinkStackNode));
    if temp==NULL
        return FALSE;
    temp->data=x;
    temp->next=top->next;
    top->next=temp;
    return TRUE;
}
//出栈
int Pop(LinkStack top,StackElementType *x){
    LinkStackNode *temp;
```

```
    temp=top->next;
    if temp==NULL
        return FALSE;
    top->next=temp->next;
    *x=temp->data;
    free(temp);
    return TRUE;
}
```

4.2.2 链栈的应用

多栈:

```
typedef struct node{
    StackElementType x;
    struct  node *next;
}LinkStackNode, *LinkStack;
LinkStack top[M];
//把多个链栈的top指针存入一个数组
```

括号匹配:

```
#python解法
#思路：从字符串s的第一个字符开始遍历定义栈st，st 为空时，说明没有左括号，只有右括号(eg:)]})，不符合，返回False如果遍历的字符为左括号，则压入栈st中比较 当前字符 和 栈中的字符（st.pop()），不相等，则返回 False以上循环用来判断不符合条件的括号之后如果 st == [] 则 返回True，否则 返回 False。注意：这里排除 [(，左括号多的字符串。


class Solution:
    def isValid(self, s: str) -> bool:
        parens = "()[]{}"                               # 字符串中含有的所有 括号类型
        open_parens = "([{"                             # 开括号
        opposite = {")": "(", "]": "[", "}": "{"}   # 闭括号对应的开括号

        st = []                                         # 使用列表模拟栈，(左)开括号入栈
        for pr in s:
            if pr not in parens:
                return False                            # 字符串中含有除括号外的其它字符则返回 False
            elif pr in open_parens:                     # pr 如果为开括号 则压入栈 st 中
                st.append(pr)
            elif st == []:                              # 栈st为空，说明没有 左括号，只有右括号(eg:])，返回 False
                return False
            elif st.pop() != opposite[pr]:              # 右括号 和 左括号不匹配 eg:[[返回False
                return False
            else:
                pass

        # 检索完之后，如果出现 栈st 还有字符，可能字符串s只有单括号 如[，  返回False
        # if len(st) == 0:
        #     return True
        # else:
```

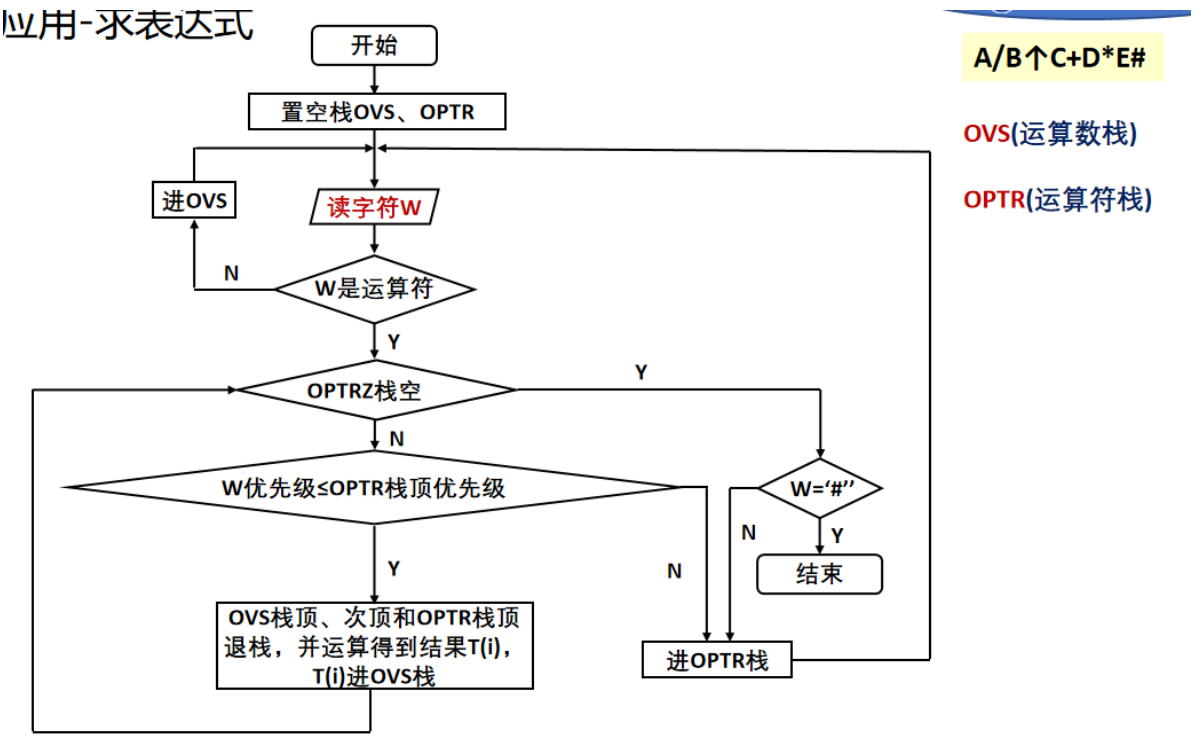
```
        #     return False
        return True if st == [] else False

if __name__ == '__main__':
    solution = Solution()
    res = solution.isValid("(()]}")
    print("res: ", res)
```

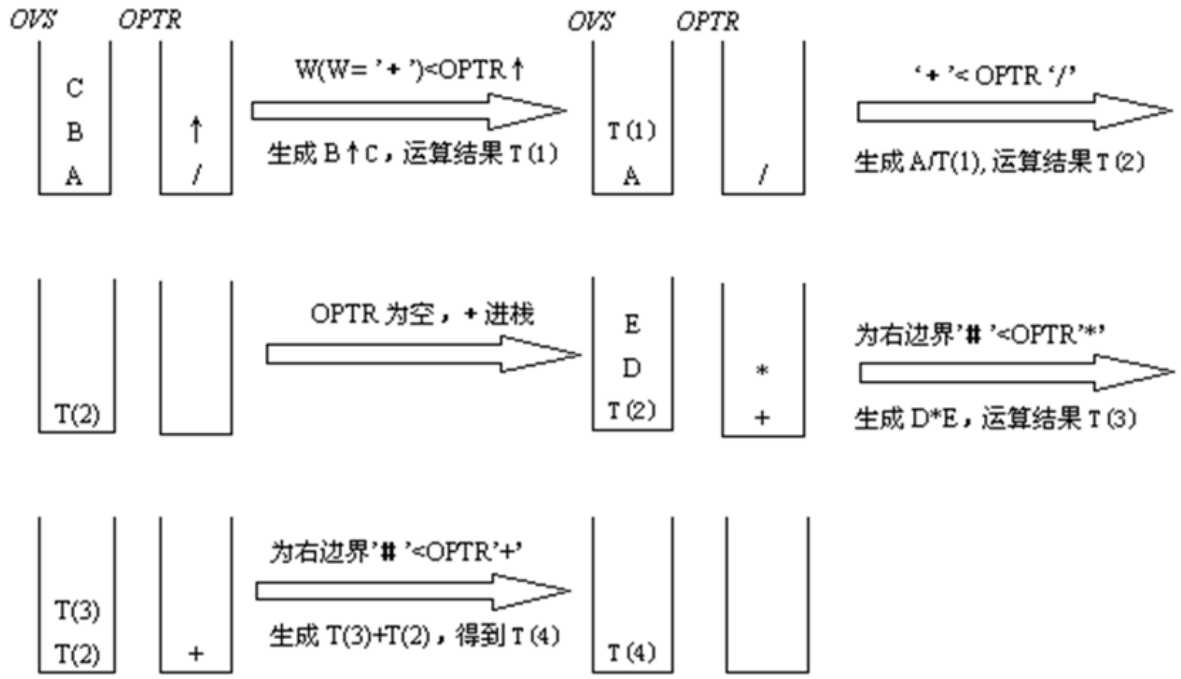
无括号求表达式：

https://blog.csdn.net/qq_41596915/article/details/104224560

https://blog.csdn.net/yehuozhili/article/details/83048955



例：实现A/B个C+D*E#的运算过程时栈区变化情况



4.3 队列

队列是一种运算受限的线性表。

把进行插入的一端称做队尾（rear）。

进行删除的一端称做队首或队头（front）。

向队列中插入新元素称为进队或入队，新元素进队后就成为新的队尾元素。

从队列中删除元素称为出队或离队，元素出队后，其后继元素就成为队首元素。

队列具有先进先出 (Fist In Fist Out，缩写为FIFO)的特性

链队列的定义由两部分组成：数据为单链表，队头与队尾的指针单独形成一个结构体

```
//链表部分
typedef struct node{
    QueueElementType data;
    struct Node *next;
}LinkQueueNode;
//指针结构体
typedef struct{
    LinkQueueNode *front;//队头指针
    LinkQueueNode *rear; //队尾指针
}LinkQueue;
```

```
//初始化
int InitQueue(LinkQueue *Q){
    Q->front =(LinkQueueNode*)malloc(sizeof(LinkQueueNode));
    if(Q->front!=NULL){
        Q->rear=Q->front;
        Q->front->next=NULL;
        return(TRUE);
    }
    else
        return(FALSE);//溢出
}
//入队
int EnterQueue(LinkQueue *Q,QueueElementType x){
    LinkQueueNode *Node;
    NewNode=(LinkQueueNode*)malloc(sizeof(LinkQueueNode));
    if(NewNode!=NULL){
        NewNode->data=x;
        NewNode->next=NULL;
        Q->rear->next=NewNode;
        Q->rear=NewNode;
        return(TRUE)
```

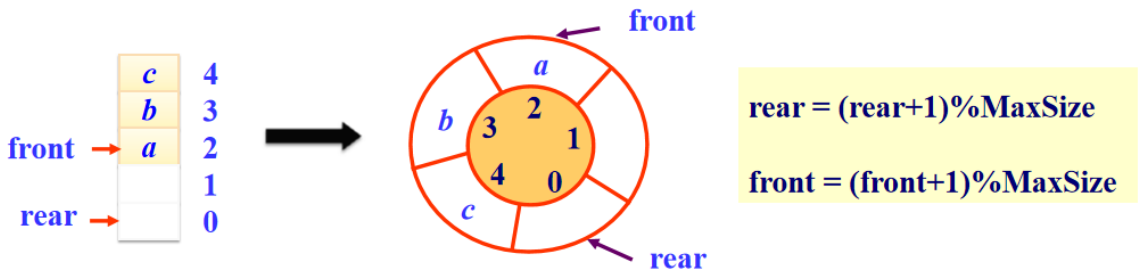
```
    }
}
//出队
int DeleteQueue(LinkQueue *Q,QueueElementType *x){
    LinkQueueNode *p;
    if(Q->front==Q->rear)
        return FALSE;//空队
    p=Q->front->next;
    Q->front->next=p->next;//队头元素出队
    if(Q->rear==p){
        Q->rear=Q->front;//变成空队
    *x=p->data;
    free(p);
        return(TRUE);
    }
}
```

循环队列：

```
//第一种方法
//定义
typedef struct{
    QueueElementType element[50];
    int front,rear;
}SeqQueue;
//初始化
void InitQueue(SeqQueue *Q){
    Q->front = Q->rear = 0;
}
//入队
int EnterQueue(SeqQueue *Q,QueueElementType x){
    if((Q->rear+1)%50==Q->front){
        return FALSE;
    }
    Q->element[Q->rear]=x;
    Q->rear=(Q->rear+1)%50;//重置队尾指针
    return TRUE;
}
//出队
int DeleteQueue(SeqQueue *Q,QueueElementType *x){
    if(Q->front==Q->rear)
        return FALSE;
    *x=Q->element[Q->front];
    Q->front = (Q->front+1)%MAXSIZE;
    return TRUE;
}
//第二种方法，使用标志域
//定义
typedef struct{
    QueueElementType element[MAXSIZE];
    int front;
    int rear;
    int tag;
}SeqQueue;
//初始化
```

```
void InitQueue(SeqQueue *Q){
    Q->front = Q->reaar =0;
}
//入队
int EnterQueue(SeqQueue *Q,QueueElementType x){
    if(Q->rear==Q->front&&Q->tag==1)
        return FALSE;
    Q->element[Q->reaar]=x;
    Q->rear=(Q->rear+1)%MAXSIZE;
    if(Q->front==Q->rear)
        Q->tag=1;
    return TRUE;
}
//出队
int DeleteQueue(SeqQueue *Q,QueueElementType){
    if(Q->rear==Q->front&&Q->tag==0)
        return FALSE;
    *x=Q->element[Q->front];
    Q->front=(Q->front+1)%MAXSIZE;
    if(Q->front==Q->rear)
        Q->tag=0;
    return TRUE;
}
```

把数组的前端和后端连接起来，形成一个环形的顺序表，即把存储队列元素的表从逻辑上看成一个环，称为循环队列或环形队列。



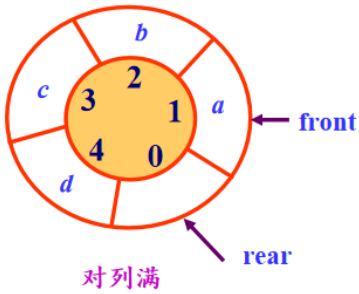
实际上内存地址一定是连续的，不可能是环形的，这里是通过逻辑方式实现环形队列

两个方法的区别：

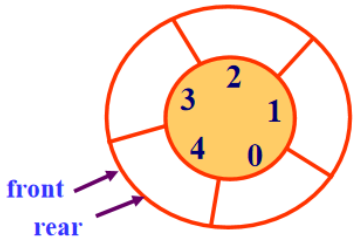
处理方法

- 方法一：是少用一个元素空间。
  - 当队尾指针所指向的空单元的后继单元是队头元素所在的单元时，则停止入队。
  - 现在队列“满”的条件为：

    (rear+1) mod MAXSIZE=front。
  - 判队空的条件不变，仍为rear=front。
- 方法二：是增设一个标志量的方法，以区别队列是“空”还是“满”。



对列满



对列空

5.串

5.1 基本定义

串是由零个或多个字符组成的有限序列。串是一种特殊的线性表，数据元素之间呈线性关系。串的数据对象限定为字符集。

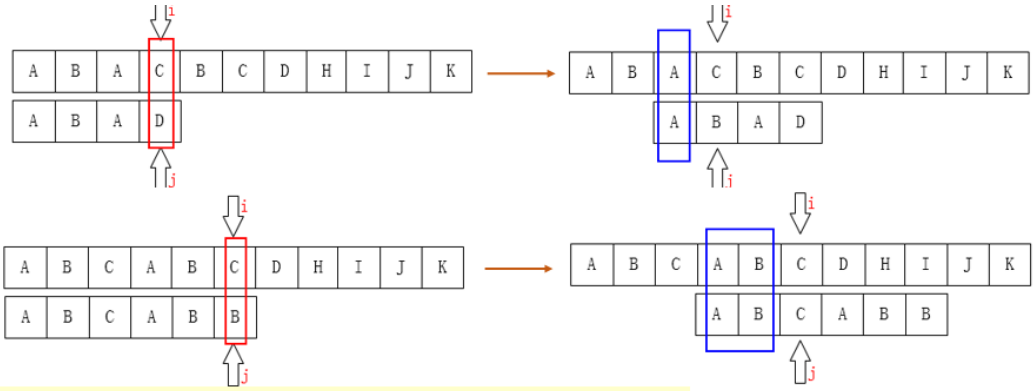
5.2代码实现

```
#define MAXLEN 255
typedef struct{
    char ch[MAXLEN];//每个元素存储一个字符
    int len;//串长度
}SString;//顺序串
typedef struct{
    char *ch;//串的首地址
    int len;
}HString;//堆串
HString S;
S.ch = (char*)malloc(sizeof(char));
S.len=0;
//链式串的实现
typedef struct StringNode{
    char ch[x];//x的值决定每一个结点内可以存储多少个字符
    struct StringNode *next;
}StringNode,*String;
//顺序串插入
int StrInsert(SString *s,int pos,SString t){
    int i;
    if(pos<0||pos>s->len)
        return 0;//插入位置不合法
    if(s->len +t.len<=MAXLEN){
        for(i=s->len+t.len-1;i>=t.len+pos;i--){
            s->ch[i]=s->ch[i-t.len];
        }
        for(i=0;i<t.len;i++)
            s->ch[i+pos]=t.ch[i];
        s->len +=t.len;
```

```
    }
    else{
        if(pos+t.len<=MAXLEN){
            for(i=MAXLEN-1;i>t.len+pos-1;i--){
                s->ch[i]=s->ch[i-t.len];
            }
            for(i=0;i<t.len;i++)
                s->ch[i+pos]=t.ch[i];
            s->len=MAXLEN;
        }
        else{
            for(i=0;i<MAXLEN-pos;i++)
                s->ch[i+pos]=t.ch[i];
            s->len=MAXLEN;
        }
    }
}
//顺序串删除
int StrDelete(SString *S,int pos,int len){
    int i;
    if(pos<0||pos>(s->len-len))
        return(0);
    for(i=pos+len;i<s->len;i++)
        s->ch[i-len]=s->ch[i];
    s->len=s->len-len;
    return 1;
}
//顺序串比较
int StrCompare(SString s,SString t){
    int i;
    for(i=0;i<s.len&&i<t.len;i++){
        if(s,ch[i]!=t,ch[i])
            return(s,ch[i]-t,ch[i]);
    }
    return(s.len-t.len);
}//s和t相等则返回0,s>t返回正数,s<t返回负数
//模式匹配-BF
/*求从主串s的下标pos起，串t第一次出现的位置，成功返回位置序号，不成功返回-1*/
int StrIndex(SString s,int pos, SString t) {
    int i, j, start;
    if (t.len==0)
          return(0);   /* 模式串为空串时，是任意串的匹配串 */
    start=pos;
    i=start;
    j=0;         /* 主串从pos开始，模式串从头（0）开始 */
    while (i<s.len && j<t.len)
        if (s.ch[i]==t.ch[j]) {
             i++;
             j++;
        }   /* 当前对应字符相等时推进 */
        else {
             start++;         /* 当前对应字符不等时回溯 */
             i=start;
             j=0;    /* 主串从start+1开始，模式串从头（0）开始*/
        }
```

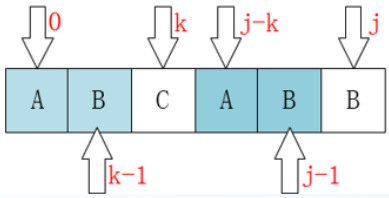
```
        if (j>=t.len)
                return(start);     /* 匹配成功时，返回匹配起始位置 */
        else
                return(-1);        /* 匹配不成功时，返回-1 */
}
//回溯的另一种方式：
i=i-j+1;j=0;
```

```
//另一种匹配方式：KMP(不需要指针回溯)
对于模式串T 的每个元素 tj，都存在一个实数 k ，使得模式串 T 开头的 k 个字符（t0 t1…tk-1）依次与 t j 前面的 k（tj-k tj-k+1…tj-1）个字符相等，这里第一个字符 tj-k 最多从 t1 开始，所以 k < j 。
如果这样的 k 有多个，则取最大的一个。
模式串 T 中每个位置 j 的字符都有这种信息，采用 next 数组表示，即 next[ j ]=MAX{ k }。
```



当匹配失败时，**j**要移动的下一个**位置k**。存在着这样的性质：**最前面的k个字符和j之前的最后k个字符是一样的**。如果用数学公式来表示是这样的：

P[0 ~ k-1] == P[j-k ~ j-1]



若令next[j]=k，则next[j]表明当模式中第j个字符与主串中相应字符‘失配’时，在模式中需重新和主串中该字符进行比较的字符的位置。由此可引出模式串的next函数的定义：

$$next[j]=\begin{cases}-1 & \text{当 } j=0 \text{ 时} \\ Max\{k \mid 0<k<j \text{ 且 } `P_0P_1\cdots P_{k-1}` = `P_{j-k}P_{j-k+1}\cdots P_{j-1}`\} & \text{当此集合不为空时} \\ 0 & \text{其他情况}\end{cases}$$

```
//KMP
int StrIndex(SString s,int pos,SString t,int next[]){
    int i,j;
    if(t.len==0)
```

```
        return 0;
    i=pos;
    j=0;
    while(i<s.len&&j<t.len)
        if(j==-1||s.ch[i]==t.ch[j]){
            i++;j++;
        }
    else
        j=next[j];
    if(j>=t.len)
        return(i-t.len);
    else
        return -1;
}
void GetNext(SString t,int next[]){
    int j,k;
    j=0;k=-1;
    next[0]=-1;
    while(j<t.length-1){
        if(k==-1||t.data[j]==t.data[k]){
            j++;k++;
            next[j]=k;
        }
        else
            k=next[k];
        }
    }
}
```

#next[]数组的定义和计算方式:

求解next数组前要懂得以下几个概念:
1、前缀：包含首位字符但不包含末位字符的子串。
2、后缀：包含末位字符但不包含首位字符的子串。
3、next数组定义：当主串与模式串的某一位字符不匹配时，模式串要回退的位置。
4、next[j]：其值 = 第j位字符前面j-1位字符组成的子串的前后缀重合字符数+1

手算Next数组

```
      j: 1 2 3 4 5 6 7 8
      P: a b a a b c a c
Next[j]: 0 1 1 2 2 3 1 2
```

当j = 1时，规定next[1] = 0
当j = 2时，j前子串为“a”,next[2] = 1
当j = 3时，j前子串为“ab”,next[3] = 1
当j = 4时，j前子串为“aba”,next[4] = 2
当j = 5时，j前子串为“abaa”,next[5] = 2
当j = 6时，j前子串为“abaab”,next[6] = 3
当j = 7时，j前子串为“abaabc”,next[7] = 1
当j = 8时，j前子串为“abaabca”,next[8] = 2

规律:
1、next[j]的值每次最多增加1
2、模式串的最后一位字符不影响next数组的结果

参考资料:

https://zhuanlan.zhihu.com/p/83334559

https://blog.csdn.net/qq_73949678/article/details/129393225

6.数组与广义表

6.1数组

6.1.1基本概念

数组是一组有固定个数的元素的集合。对于数组的一般操作有两类：

获取特定位置的元素值/修改特定位置的元素值。

*.稀疏矩阵：矩阵中大多数元素为0的矩阵。

为了压缩稀疏矩阵的存储，除了存储非零元素还需要存储对应的行号和列号（三元组表示法）

6.1.2稀疏矩阵代码

```
#define MAXSIZE 1000
typedef struct{
    int row,col;//非零元素的行/列下标
    ElementType e;//非零元素的值
}Triple;
typedef struct{
    Triple data[MAXSIZE];//非零元素的三元组表
    int m,n,len;//矩阵的行数列数与非零元素个数
}TSMatrix;
```

```
//矩阵转置
void TransMatrix(ElementType source[n][m],ElementType dest[m][n]){
    inti,j;
    for(i=0;i<m;i++){
        for(j=;j<n;j++){
            dest[i][j]=source[j][i];
        }
    }
}
```

```
//根据三元组的索引进行数据替换
void TransposeTSMatrix(TSMatrix  A,  TSMatrix  * B) {
    int  i , j, k ;
    B->m= A.n ; B->n= A.m ; B->len= A.len ;
    if(B->len>0) {
        j=0;    /* j为三元组表B的下标 */
        for(k=0; k<A.n; k++)    /* 扫描三元组表A共n次 */
            for(i=0; i<A.len; i++)      /* i为三元组表A的下标 */
                if(A.data[i].col==k){   //寻找三元组表A的列值为k的进行转置
                    B->data[j].row=A.data[i].col
                    B->data[j].col=A.data[i].row;
                    B->data[j].e=A.data[i].e;
                    j++;
                } /* 内循环if结束*/
    } /* if(B->len>0)结束*/
}
```

```
//算法优化
```

```
//num[col]：矩阵A中第col列中非零元的个数
//position[col]：矩阵A中第col列第一个非零元在B中的位置（下标）
 /*基于矩阵的三元组表示，采用快速转置法，将矩阵A转置为B所指的矩阵*/
FastTransposeTSMatrix (TSMatrix  A,  TSMatrix  * B) {
    int col , t , p, q;
    int num[MAXSIZE], position[MAXSIZE] ;
    B->len= A.len ; B->n= A.m ; B->m= A.n ;
    if(B->len) {
        for(col=0;col<A.n;col++)
            num[col]=0;          /*清零num数组*/
        for(t=0;t<A.len;t++)
            num[A.data[t].col]++;    /*计算三元组表A每一列的非零元素的个数*/
        position[0]=0;
        for(col=1;col<A.n;col++)     /*求col列中第一个非零元素在B.data[ ]中的正确位置*/
            position[col]=position[col-1]+num[col-1];
        for(p=0;p<A.len.p++) {  /* 从头扫描三元组表A一次 */
            col=A.data[p].col;
            q=position[col];      /*col列中第一个非零元素在B.data[ ]中的正确位置*/
            B->data[q].row=A.data[p].col;
            B->data[q].col=A.data[p].row;
            B->data[q].e=A.data[p].e
            position[col]++;     /*col列中下一个非零元素在B.data[ ]中的正确位置，修改了
position数组*/
        }
    }
}
```

$$A=\begin{bmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{bmatrix} \quad B=\begin{bmatrix} 0 & 0 & -3 & 0 & 0 & 15 \\ 12 & 0 & 0 & 0 & 18 & 0 \\ 9 & 0 & 0 & 24 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -7 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 14 & 0 & 0 & 0 \end{bmatrix}$$

i	j	a_{ij}
0	1	12
0	2	9
2	0	-3
2	5	14
3	2	24
4	1	18
5	0	15
5	3	-7

col	0	1	2	3	4	5	6
num[col]	2	2	2	1	0	1	0
position[col]	0	2	4	6	7	7	8

i	j	a_{ij}
		-3
		12
		9
		-7
		14

```
//十字链表法

//十字链表定义
typedef struct OLNode{
    int row,col;
    ElementType value;
    struct OLNode *down,*right;//非零元素所在行表列表的后继链域
}OLNode,*OLink;

typedef struct{
    OLink *row_head, *col_head;//行、列链表的头指针向量
    int m,n,len;
}CrossList;
```

```
//十字链表创建稀疏矩阵
CreateCrossList(CrossList *M){
    int m, n ,t,i,j;
    OLink p,q;
    scanf("%d,%d,%d",&m,&n,&t);  /*输入M的行数、列数和非零元素的个数*/
    M->m=m;M->n=n;M->len=t;
    if(!(M->row_head=(OLink *)malloc(m*sizeof(OLink)))){
        exit(OVERFLOW);
    }
    if(!(M->col_head=(OLink *)malloc(m*sizeof(OLink)))){
        exit(OVERFLOW);
    }
    for(i=0;i<m;i++){
        M->row_had[i]=NULL;
    }
    for(j=0;j<n;j++){
        M->col_head[j]=NULL;
    }
    for(scanf(&i,&j,&e);i!=-1;scanf(&i,&j,&e)){//分配结点空间并插入数据
        if(p!=(OLNode *)malloc(sizeof(OLNode)))
            exit(OVERFLOW);
        p->row=i;p->col=j;p->value=e;
        if(M->row_head[i]==NULL)
            M->row_head[i]=p;
        else{
            q=M->row_head[i];
            while(q->right&&q->right->col<j){
                q=q->right;//移动到链表末尾
                p->right=q->right;
                q->right=p;//插入节点
            }
        }
        if(M->col_head[j]==NULL)
            M->col_head[j]=p;
        else{
            q=M->col_head[j];
            while(q->down&&q->down->row<i){
                q=q->down;
                p->down=q->down;
                q->down=p;//插入节点
            }
        }

    }
}
```

广义表

广义表中有两种结点：

1.单元素结点（原子结点）

2.子表结点（表结点）

任何一个非空的广义表都可以分解成表头/表尾

一个表结点由三个域构成：标志域，指向表头的指针域，指向表尾的指针域

原子结点只有标志域和值域

```
//头尾类型定义
typedef enum {ATOM, LIST} ElemTag;   /*ATOM=0，表示原子；LIST=1，表示子表*/
typedef struct GLNode {
    ElemTag   tag;      /*标志位tag用来区别原子结点和表结点*/
    union  {
        AtomType  atom; /*原子结点的值域atom*/
        struct {
            struct GLNode  * hp, *tp;
        } htp;          /*表结点的指针域htp，  包括表头指针域hp和表尾指针域tp*/
    } atom_htp;   /* atom_htp 是原子结点的值域atom和表结点的指针域htp的联合体域*/
} GLNode,  *GList;


//同层结点链定义
typedef enum {ATOM, LIST} ElemTag;   /*ATOM=0，表示原子；LIST=1，表示子表*/
typedef struct GLNode {
    ElemTag   tag;      /*标志位tag用来区别原子结点和表结点*/
    union  {
        AtomType  atom; /*原子结点的值域atom*/
        struct GLNode  * hp;
    } atom_htp;
    struct GLNode  * tp;
} GLNode,  *GList;
```

icoding题选

1.串替换

题目：

不调用库函数，自己实现字符串替换操作，函数原型为：

```
int str_replace(const char *in, char *out, int outlen, const char *oldstr, const char *newstr);
```

参数说明:

in， 原始字符串，保持不变
out, 存放替换结果的字符串
outlen，out空间的大小
oldstr，要替换的旧字符串
newstr，替换成的新字符串
函数返回成功替换的次数，即有多少个子串被成功替换

在替换过程中，任何情况下所得字符串（及结束符）不应该超过 outlen，如果某次替换所得字符串的长度超过 outlen，则不进行这次替换操作，整个替换操作结束。如：
原始串为 "aaabbbccc"，outlen 为14, oldstr 为 "c"，newstr 为 "333" 时，两次替换后得 "aaabbb333333c"，此时字符串占用空间为14字节。
如果再进行替换，则会超出 out 所占用的空间，所以停止替换操作。此时函数应该返回 2, out指向的串为 "aaabbb333333c"
再如：原始串为 "aaabbbccc"，outlen 为10, oldstr 为 "bb"，newstr 为 "123456"，进行替换后所得的串长度为14，与结束符一共占15 个字节，超过outlen的10字节，此时不进行替换，函数应该返回 0。

```c
#include<stdio.h>
#include<stdlib.h>
#include "dsstring.h"
int str_replace(const char *in, char *out, int outlen, const char *oldstr, const
char *newstr);
void *memset(void *str, int c, size_t n){
    if(NULL==str||n<0) return NULL;
    char *temp=(char*)str;
    while(n-->0) *temp++=c;
    return str;
}//申请内存空间，将连续的大小为n的内存空间全部覆盖为字符c
char *Strncat(char *dest, const char *src, size_t n){
    char *ret=dest;
    while(*dest!=0) dest++;
    while(n--&&(*dest++=*src++)){
        ;
    }
    return ret;
}//字符串拼接函数，截止条件为'\0'；(n的作用)

char* Strcat(char* dest, const char* src){
    char *ret =dest;
    while(*dest!=0){
        dest++;
    }
    while(*dest++=*src++){
        ;
    }
    return ret;
}//字符串拼接函数，无截止条件
int Strlen(const char *str){
    int count =0;
    while(*str!='\0'){
        str++;
        count++;
    }
    return count;
}//字符串长度计算
int Strncmp(const char *str1,const char *str2,int size){
    for(int i=0;i<size;i++){
        if(*(str1+i)>*(str2+i)){
            return 1;
        }else if(*(str1+i)<*(str2+i)){
            return -1;
        }
        if(*(str1+i)==0||*(str2+i)==0){
            break;
        }
    }
    return 0;
}//字符串比较
int str_replace(const char *in, char *out, int outlen, const char *oldstr, const
char *newstr){
    memset(out,0,outlen);
    int count =0;
```

```
    int cur_len=Strlen(in);
    for( int i=0;i<Strlen(in);i++){

        if(Strncmp(in+i,oldstr,Strlen(oldstr))==0/*在oldstr的长度上新旧字符串相同*/&&
(cur_len+Strlen(newstr)-Strlen(oldstr)+1)<=outlen/*新字符串长度未超过限制*/){
            Strcat(out, newstr);
            cur_len=cur_len+Strlen(newstr)-Strlen(oldstr);
            i+=Strlen(oldstr)-1;
            count++;

    }else{//没有找到
        Strncat(out ,in+i,1);//返回原字符串
    }

     if(Strlen(out)+1>=outlen) break;//超出限制长度
}
    return count;
}
```