# 数据结构

## 线性表

### 顺序存储

#### 定义

#define MAX\_SIZE 128

typedef struct SqList{

int data[MAX\_SIZE]; //MAX\_SIZE是线性表的最大长度

int length; //线性表需要一个变量标识当前长度

}SqList;

#### 初始化

SqList \*s; //访问成员变量s->data

SqList s; //访问成员变量s.data

#### 获取元素

思路：将data数组的第i-1个字符赋值给指针型形参（注意必须是指针型的形参或引用），这样做可以获取输出

代码：

int SqListGetElement(SqList \*s,int I,int \*element){

if(s->length == 0 || i<1 || i>s->length)//对输入的参数进行检查

return -1;

\*element = s->data[i-1];

return 0;

}

#### 插入元素

思路：先将元素依次向后移动1个位置，然后将待插入的数据放到数组的特定位置

代码：

int SqListInsertElement(SqList \*s,int I, int \*element){

int j=0;

if(s->length==MAX\_SIZE || i<0 || i>s->length)

return -1;

if(i<s->length){

for(j=I;j<s->length;j++){

s->data[j+1]=s->data[j]; //这种方式不对，会发生覆盖

}

}

改写为：

if(i<s->length){

for(j=s->length-1;j>=i-1;j--){

s->data[j+1]=s->data[j];

}

}

s->data[i] = \*element;

s->length++;

return 0;

}

#### 删除元素

思路：先将指定元素删除，然后数组中元素依次向前移动1位

代码：

int SqListDeleteElement(SqList \*s, int i, int \*element){

int k;

if(s->length==0 || i<0 || i>s->length){

return -1;

}

\*element = s->data[i-1];

for(k=I;k<s->length;k++){

s->data[k-1]=s->data[k];

}

s->length--;

return 0;

}

### 链式存储

#### 定义

思路：链式存储即先定义一个独立的节点node，然后定义链表将节点串联起来，这就需要用到next指针。

代码：

typedef struct LinkListNode{

int data; //只表示一个元素，所以这里之定义一个int而不是数组

struct LinkListNode \*next;

}LinkListNode;

typedef struct LinkListNode \*LinkList; //指针型变量

或者：

typedef struct LinkListNode{

int data;

struct LinkListNode \*next;

}LinkListNode, \*LinkList; //这种定义方式比较好

或者：

typedef struct LinkListNode{

int data;

struct LinkListNode \*next;

}LinkListNode, \*LinkListPtr;

typedef struct LinkList{

LinkListPtr p;

int count; //这样定义不太合理，链式存储就没有必要设置当前长度length

}

#### 初始化

LinkList \*l;

LinkList l;

#### 获取元素

思路：获取头指针，然后依次遍历链表，找到对应的第i个节点后返回

代码：

int LinkListGetElement(LinkList l, int I, int \*element){//传引用而非指针LinkList \*l

int j=0;

LinkList p;//直接操作链表有危险，这里采用定义一个新的局部变量操作

j=1;

while(p && j<i){//依次遍历，更新指针位置

p=p->next;

j++;

}//掌握这种链表遍历的方法

if(!p||j>=i){

return -1;

}

\*element = p->data;

return 0;

}

#### 插入元素

思路：插入需要新建一个节点（malloc分配），然后更新这个新节点的成员变量，即数据data和指针next，接着就是以前的节点与新节点建立联系，这就形成了一个新的链表

代码：

int LinkListInsertElement(LinkList l, int i, int \*element){

//这里定义LinkList l是指针类型，因为前面已经定义为LinkListNode \*LinkList

//了，如果这里定义为LinkList \*l，则前面的定义就是LinkListNode LinkList

int j=0;

LinkList p,s;

//需要借助中间变量链表实现操作，

//1个用于接收形参链表，1个用于malloc新节点

j=1;

p=l->next;

for(p && j<i){

p=p->next;

j++;

}

s=(LinkList)malloc(sizeof(LinkListNode));

//链表插入元素就需要新分配一个节点的内存空间，然后更新指针的指向

s->data=\*element;

s->next=p->next;//更新这个新节点的成员变量

p->next=s;//建立节点的关联

return 0;

}

#### 删除元素

思路：

代码：

int LinkListDeleteElement(LinkList l, int I, int \*element){

int j=0;

LinkList p,q;//插入和删除都需要借助中间变量的一个节点完成

j=1;

p=l->next;

while(p && j<i){

p=p->next;

j++;

}

q=q->next;//待删除的节点

\*element=q->data;//待删除节点的形参

p->next=q->next;//待删除节点的指针

free(q);//释放掉这个节点，删除的就不需要了

return 0；

}

## 栈

### 顺序存储

#### 定义

typedf struct{

int data[MAX\_SIZE];

int top;

}SqStack;

#### 获取元素

int SqStackGetElement(SqStack \*s, int \*element){

if(s->top == -1){

return -1;

}

\*element=s->data[s->top];

return 0;

}

#### 插入元素

思路：插入需要判断是不是越界，即top指针域MAX\_SIZE对比大小

代码：

int SqStackPush(Stack \*s, int \*element){

if(s->top == MAX\_SIZE-1){

return -1;

}

s->top++;//top指针先自加1，然后赋值

s->data[s->top]=\*element;

return 0;

}

#### 删除元素

思路：删除栈的元素需要考虑栈是否为空

代码：

int SqStackPop(SqStack \*s, int \*element){

if(s->top == -1){

return -1;

}

\*element=s->data[s->top];

s->top--;

return 0;

}

### 链式存储

#### 定义

typedef struct LinkStackNode{

int data;

stuct LinkStackNode \*next;

}LinkStackNode \*LinkStackPtr;

typedef struct LinkStack{

LinkStackPtr top;

int count;//这个需要定义count

}LinkStack;//定义时采用指针类型

如下：

typedef struct LinkStackNode{

int data;

struct LinkStackNode \*next;

}LinkStackNode \*LinkStackPtr;

LinkStackPtr p;//这种定义的是指针类型，注意与前面的区别

#### 获取元素

int LinkStackGetElement(LinkStack \*l, int \*element){

//这个是定义LinkList \*l而不是LinkList l，与前面链表的定义有区别

LinkStackPtr s=(LinkStackPtr)malloc(sizeof\*(LinkStackNode));

s=l->top;

\*element=s->data;

return 0;

}

#### 插入元素

思路：

代码：

int LinkStackPush(LinkStack \*l, int \*element){

LinkStackPtr s = (LinkStackPtr)malloc(sizeof(LinkStackNode));

s->data = \*element;

s->next=l->top;

//为新节点分配内存并更新成员变量

l->top=s;

l->count++;

return 0;

}

#### 删除元素

思路：把top指针并更为原来top节点的next，这个借助于中间变量指针s完成

代码：

int LinkStackPop(LinkStack \*l, int \*element){

LinkStackPtr s;//只是声明指针，但是不需要分配内存

\*element = l->top->data;

s=l->top;//待删除节点初始化

l->top=s->top->next;

free(s);

l->count--;

return 0;

}

## 队列

### 顺序存储

#### 定义

typedef struct SqQueue{

int data[MAX\_SIZE];

int front,rear;

}SqQueue;

#### 初始化

int SqQueueInit(SqQueue \*q){

q->front=0;

q->rear=0;

return 0;

}

#### 队列长度

int SqQueueLength(SqQueue \*q){

return (q->rear-q->front+MAX\_SIZE)%MAX\_SIZE;

}

#### 获取元素

思路：获取元素也只能获取队尾和队首的数据

代码：

#### 插入元素

思路：插入元素需要更新rear变量

代码：

int SqQueueEnQueue(SqQueue \*q, int \*element){

if((q->rear+1)%MAX\_SIZE == q->front){

return -1;

}

q->data[q->rear]=\*element;

q->rear=(q->rear+1)%MAX\_SIZE;

return 0;

}

#### 删除元素

思路：删除元素需要判断是否为空（即front是都等于rear），然后更新front成员变量

代码：

int SqQueueDeQueue(SqQueue \*q, int \*element){

if(q->front == q->rear){//队列为空

return -1;

}

\*element = q->data[q->front];//将队头元素赋值给element

q->front = (q->front+1)%MAX\_SIZE;

//将front指针向后移动一个位置，如到最后则转到数组头部

return 0;

}

### 链式存储

#### 定义

typedef struct LinkQueueNode{

int data;

struct LinkQueueNode \*next;

}LinkQueueNode \*LinkQueuePtr;

typedef struct LinkQueue{

LinkQueuePtr front,rear;

}LinkQueue;

说明：LinkQueuePtr与LinkQueue的定义不同，LinkQueuePtr p表示指针，而LinkQueuePtr \*q才表示指针！

#### 获取元素

思路：获取元素只能获取队尾和队首的

#### 插入元素

思路：插入新元素，首先申请一个新节点，把新节点（此时还是孤立的节点）的成员变量（即data和next），然后更新原来节点rear与新节点的连接关系以及队列的rear。

代码：

int LinkQueueEnQueue(LinkQueue \*q, int \*element){

LinkQueuePtr p = (LinkQueuePtr)malloc(sizeof(LinkQueueNode));

p->data=\*element;

p->next=NULL;//新节点的下一个节点为NULL

//更新新节点的参数，接下来更新整个链表的成员变量，即front和rear

q->rear->next=p;

//操作队尾，原来的rear指向现在的新节点

q->rear=p;

//更新rear节点（前面是更新节点之间的指向关系）

return 0；

}

#### 删除元素

int LinkQueueDeQueue(LinkQueue \*q, int \*element){

LinkQueuePtr p;//只需要声明变量而不需要实际分配内存空间

if(q->front==q->rear){

return -1;

}

//更新新节点的成员变量

p=q->front->next;//该节点成了最新的头结点

\*element = p->data;//头结点的数据

q->front->next=p->next;

if(q->rear==p){

q->rear=q->front;

}

free(p);

return 0;

}

## 字符串

### 顺序存储

#include<stdio.h>

#include<stdlib.h>

#define MAX\_SIZE 128

typedef char String[MAX\_SIZE];

int StringIndex(String S, String T, int pos){

int i=pos;

int j=1;

while(i<S[0] && j<=TT[0]){

if(S[i]==T[i]){

++i;

++j;

}else{

i=i+1-(j-1);

j=i;

}

}

if(j>T[0])

return i-T[0];

else

return 0;

}

### 链式存储

## 树

## 图

# 算法

## 查找

## 排序