**数据结构复习资料**

**[一、 顺序表SqList 2](#_Toc22857)**

[1、 静态分配 2](#_Toc23903)

[2、 动态分配 7](#_Toc20367)

**[二、 链表 10](#_Toc5785)**

[1、单链表 10](#_Toc24026)

[2、双向链表 13](#_Toc4974)

[3、循环单链表 14](#_Toc28506)

**[三、 栈和队列 16](#_Toc18830)**

[1、链栈 16](#_Toc1339)

[2、顺序栈 17](#_Toc30406)

[3、链队列 18](#_Toc27167)

[4、循环队列 19](#_Toc22292)

**[四、 串 20](#_Toc7784)**

**[五、 树 28](#_Toc13379)**

**[六、 查找 30](#_Toc11229)**

[1、顺序查找和折半查找 30](#_Toc25472)

[2、 二叉排序树 32](#_Toc17262)

**[七、 排序 35](#_Toc10382)**

[1、简单排序 35](#_Toc24100)

[2、快速排序 38](#_Toc29555)

**[八、 一些经典问题 40](#_Toc8830)**

1. **顺序表SqList**
2. **静态分配**

//定义顺序表L的结构体

typedef struct

{

Elemtype data[MaxSize ];

int length;

}SqList;

* ****CreateList(SqList &L,int n)****

****参数：顺序表L，顺序表长度n****

****功能：创建长度为的顺序表****

* ****InitList(SqList &L)****

****功能：初始化****

* ****InsertList(SqList &L,int i,ElemType e)****

****参数:顺序表L,位置i,元素e****

****功能：位置i处插入元素e****

* ****ListDelete(SqList &L,int i)****

****参数:顺序表L,位置i****

****功能：删除位置i处元素****

* ****LocateElem(SqList L,ElemType e)****

****参数:顺序表L,元素e****

****功能：返回第一个等于e的元素的位置****

* ****Reverse(SqList &L)****

****功能：倒置函数，将原顺序表直接倒置****

* ****PrintList(SqList L)****

****功能：遍历L，并输出****

* ****SplitSort(SqList &L)****

****功能：分开奇偶，并分开排序****

* ****ClearList(SqList &L)****

****功能：清空顺序表****

#define MaxSize 100

using namespace std;

typedef int Status;

typedef int ElemType;

//初始化顺序表函数，构造一个空的顺序表

Status InitList(SqList &L){

memset(L.data, 0, sizeof(L)); //初始化数据为0

L.length = 0; //初始化长度为0

return 0;

}

//创建顺序表函数 初始化前n个数据

bool CreateList(SqList &L, int n){

if (n<0 || n>MaxSize) return false; //n非法

for (int i = 0; i<n; i++)

{

scanf("%d", &L.data[i]);

L.length++;

}

return true;

}

//插入函数 位置i插入数据 i及之后元素后移 1=<i<=length+1

bool InsertList(SqList &L, int i, ElemType e){

if (i<1 || i>L.length + 1) //判断位置是否有效

{

printf("位置无效！\n");

return false;

}

if (L.length >= MaxSize) //判断存储空间是否已满

{

printf("当前存储空间已满！！！\n");

return false;

}

for (int j = L.length; j >= i; j--) //位置i及之后元素后移

{

L.data[j] = L.data[j - 1];

}

L.data[i - 1] = e;

L.length++;

return true;

}

//删除函数 删除位置i的元素 i之后的元素依次前移

bool ListDelete(SqList &L, int i){

if (i<1 || i>L.length)

{

printf("位置无效！\n");

return false;

}

for (int j = i; j <= L.length - 1; j++) //位置i之后元素依次前移覆盖

{

L.data[j - 1] = L.data[j];

}

L.length--;

return true;

}

//查找函数，按位置从小到大查找第一个值等于e的元素，并返回其位置

int LocateElem(SqList L, ElemType e){

for (int i = 0; i<L.length; i++) //从低位置查找

{

if (L.data[i] == e)

return i + 1;

}

return 0;

}

//倒置函数，将原顺序表直接倒置

void Reverse(SqList &L){

if (L.length)

for (int i = 0; i<L.length - 1 - i; i++)

{

int t = L.data[i];

L.data[i] = L.data[L.length - 1 - i];

L.data[L.length - 1 - i] = t;

}

}

//奇偶分开并排序

void SplitSort(SqList &L){

int Even = 0;

int Odd = L.length - 1;

int i = 0;

int j = L.length - 1;

bool flag = false;

if (L.length)

for (; i < j; i++, j--)

{

while (L.data[i] % 2 != 0) i++;

while (L.data[j] % 2 == 0) j--;

if (L.data[i] % 2 == 0 && L.data[j] % 2 != 0&&i<j)

{

int temp = L.data[i];

L.data[i] = L.data[j];

L.data[j] = temp;

flag = true;

}

if(!flag) //没有交换

{

Even = L.length - 1; //全奇数

Odd = 0; //全偶数

}

}

if (flag)

{

for(int i=0;i<L.length;i++)

if (L.data[i] % 2 == 0)

{

Odd = i;

Even = i - 1;

break;

}

}

sort(L.data, L.data + Even + 1);

sort(L.data + Odd, L.data + L.length);

}

//清空顺序表

void ClearList(SqList &L) {

L.length = 0;

}

//输出功能函数 按位置从小到大输出顺序表所有元素

void PrintList(SqList L){

printf("当前顺序表所有元素:");

for (int i = 0; i<L.length; i++){

printf("%d ", L.data[i]);

}

printf("\n");

}

//创建顺序表函数

void Create(SqList &L){

int n;

bool flag;

L.length = 0;

printf("请输入要创建的顺序表长度(>1):");

scanf("%d", &n);

printf("请输入%d个数（空格隔开）:", n);

flag = CreateList(L, n);

if (flag) {

printf("创建成功！\n");

PrintList(L);

}

else printf("输入长度非法！\n");

}

//插入功能函数，调用InsertList完成顺序表元素插入，同时调用PrintList函数显示插入成功后的结果

void Insert(SqList &L){

int place; ElemType e; bool flag;

printf("请输入要插入的位置(从1开始)及元素:\n");

scanf("%d%d", &place, &e);

flag = InsertList(L, place, e);

if (flag)

{

printf("插入成功！！！\n");

PrintList(L);

}

}

//删除功能函数，调用ListDelete函数完成顺序表的删除，并调用PrintList函数显示插入成功后的结果

void Delete(SqList &L){

int place; bool flag;

printf("请输入要删除的位置(从1开始):\n");

scanf("%d", &place);

flag = ListDelete(L, place);

if (flag)

{

printf("删除成功！！！\n");

PrintList(L);

}

}

//查找功能函数 调用LocateElem查找元素

void Search(SqList L){

ElemType e;

int flag;

printf("请输入要查找的值:\n");

scanf("%d", &e);

flag = LocateElem(L, e);

if (flag){

printf("该元素位置为:%d\n", flag);

}

else

printf("未找到该元素！\n");

}

1. **动态分配**

typedef struct { //结构体

ElemType \*elem;

int length;

int listsize;

}SqList;

* ****初始化****
* ****插入元素（表尾/指定位置i）****
* ****删除元素（表头/表尾/指定位置i）****
* ****查询元素****

//1.初始化

int InitList\_Sq(SqList &L){ //构造一个空的线性表

L.elem=new ElemType[LIST\_INIT\_SIZE]; //申请动态空间

//L.elem = (int \*)malloc(LIST\_INIT\_SIZE \* sizeof(int));

if (L.elem==0) //申请动态空间未成功

{

printf("failure!\n");

return 0;

}

else{ //空间申请成功

L.length=0; //初始化表的实际长度为0

L.listsize=LIST\_INIT\_SIZE; //初始化表的最大长度100

return 1;

}

}

//2.插入表尾（在表尾插入数据）

int ListEndInsert\_Sq(SqList &L,ElemType e){

int \*newbase; //在表尾插入数据

if(L.length==L.listsize) //实际长度等于最大长度 ，空间不足，再做插入

{

newbase = (int \*)realloc(L.elem,(L.listsize+LIST\_INIT\_SIZE)\*sizeof(int)); //动态分配内存

L.elem = newbase; //内存分配成功之后再次指向

L.listsize+=LIST\_INIT\_SIZE;

}

if(!newbase){

printf("overflow!\n"); //输出溢出

return -1;

}

L.elem[L.length]=e; //若空间足够，在表尾插入数据

L.length++; //实际长度加一

return 1;

}

//3.插入i位置

int ListInsert\_Sq(SqList &L,int i,ElemType e){

//在动态顺序表L的第i个位置插入元素e

ElemType \*p,\*q; //定义两个指针变量，一个用于指向要插入的数据，一个指向要插入的位置的数据

int \*newbase; //用于扩展内存

if (i<=0 || i>L.length+1) //若插入位置不合法

{

printf("error!\n"); //输出错误

return 0;

}

if(L.length==L.listsize) //实际长度等于最大长度 ，空间不足，再做插入

{

newbase = (int \*)realloc(L.elem,(L.listsize+LIST\_INIT\_SIZE)\*sizeof(int)); //动态分配内存

L.elem = newbase; //内存分配成功之后再次指向基地址

L.listsize+=LIST\_INIT\_SIZE;

}

if(!newbase){

printf("overflow!\n"); //输出溢出

return -1;

}

q=&(L.elem[i-1]); //q指针指向要插入的位置

for(p=&(L.elem[L.length-1]);p>=q;--p)//循环从表尾开始，直到q指针所指的位置

{

\*(p+1)=\*p; //利用指针元素依次后移

}

\*q=e; //把移出的位置插入输入的数据

L.length++; //长度加一

return 1;

}

//4.输出

void Print\_Sq(SqList &L){ //输出动态线性表元素

ElemType \*p;

for(p = &(L.elem[0]); p <= &(L.elem[L.length-1]); p++)//利用指针从表头开始循环，到表尾结束

{

printf("%d ",\*p); //输出每个元素的值

}

printf("\n");

}

//5.删除表头

int DelHeadList\_Sq(SqList &L){ //删除表头数据元素

ElemType \*p; //定义一个指针变量，用于移动元素

if(L.length == 0){ //若为空表则输出UNDERFLOW！

printf("UNDERFLOW!");

return -1;

}

else

{

for(p = &(L.elem[1]); p <= &(L.elem[L.length-1]); p++){ //从第二个元素开始，依次向前移一位。

\*(p-1) = \*p; //后一个元素赋到前一个元素上

}

L.length--; //长度减一

return 1;

}

}

//6.删除表尾

int DelEndList\_Sq(SqList &L){

if(L.length == 0){ //若为空表

printf("UNDERFLOW!"); //输出溢出

return -1;

}

else{

L.length--; //表长度减一

return 1;

}

}

//7.删除第i个元素

int DelList\_Sq(SqList &L,int i){

ElemType \*p; //定义一个指针，用于循环移动表元素

if(i <= 0 || i > L.length){ //若i的位置小于0或大于表长度，输出Error

printf("Error!");

return 0;

}

if(L.length == 0){ //若长度为0输出溢出

printf("UNDERFLOW!");

return -1;

}else{

for(p = &(L.elem[i]); p <= &(L.elem[L.length-1]); p++){//循环从指定删除数据的位置开始，依次前移

\*(p-1) = \*p; //后一个元素覆盖前一个元素

}

L.length--; //长度减一

return 1;

}

}

//8.查询

int FindList\_Sq(SqList &L, ElemType e){

ElemType \*p;

int i=0;

for(p = &(L.elem[0]); p <= &(L.elem[L.length-1]); p++){

if(\*p == e){ //在顺序表L中查找元素e是否存在，

return (i+1); //如果存在返回对应的下标i+1

}

i++;

}

return 0; //否则返回0

}

1. **链表**

**1、单链表**

// 链表(结点)结构体的定义

typedef struct LNode{

ElemType data;

struct LNode\* next; // 指向下一个元素的指针

}LNode, \*LinkedList;

typedef LNode\* LinkedList;

* ****表头/表尾法创建单链表****
* ****指定位置插入元素****
* ****指定位置删除元素****
* ****遍历输出元素****

// 表头插入法创建表格

Status CreatingList\_H(LinkedList &L){

for(;;){ // 循环输入链表元素

LinkedList p = (LinkedList)malloc(sizeof(LNode)); // p指针指示当前结点

if(!p) return 0;

scanf("%d",&p->data); // 输入元素数值

// 如果输入的是0，则结束循环，停止输入元素

if(p->data == 0) break;

// 如果不是0，则按照表头插入法把结点插入链表

else{

p->next = L->next;

L->next = p;

}

} /\*for\*/

return 1;

}

// 表尾插入法创建表格

Status CreatingList\_T(LinkList &L){

int x;

L = (LinkedList)malloc(sizeof(LNode));

LinkedList s, r = L;

for(;;){

scanf("%d",&x); // 输入元素值

if(x == 0) break; // 如果输入的是0，则结束循环，停止输入

s = (LinkedList)malloc(sizeof(LNode)); //创建新的结点

s->data = x;

r->next = s;

r = s;

}

r->next = NULL;

return 1;

}

// 在指定位置插入指定元素

Status InsertElem(LinkedList &L, int i, ElemType e){

LinkedList p = L;

int j = 0;

// 定位第i-1个结点

while(p && j<i-1) {

p = p->next;

++j;

}

if(!p || j>i+1) return 0;

LinkedList s = (LinkedList)malloc(sizeof(LNode)); // 创建新结点s

// 插入新节点

s->data = e;

s->next = p->next;

p->next = s;

return 1;

}

// 删除指定元素

Status DeleteElem(LinkedList &L, int i){

LinkedList p = L;

int j = 0;

// 定位第i-1个元素

while(p && j<i-1)

{

p = p->next;

++j;

}

if(!(p->next) || j>i-1) return 0;

// 删除并释放指定结点

LinkedList q;

q = p->next;

p->next = p->next->next;

free(q);

return 1;

}

// 遍历（输出）所有元素

void GetAllElem(LinkedList &L){

LinkedList p = L->next;

if(!p) printf("empty list!");

else{

while(p){

printf("%d ",p->data);

p = p->next;

} /\*while\*/

printf("\n");

} /\*else\*/

}

**2、双向链表**

typedef struct line{

ElemType data;

struct line \* prior; //指向直接前趋

struct line \* next; //指向直接后继

} line;

* ****创建双向链表****
* ****遍历输出元素****
* ****示范****

// 创建一个双向链表

line\* CreateLine(line \* head){

head=(line\*)malloc(sizeof(line));//创建链表第一个结点（首元结点）

head->prior=NULL;

head->next=NULL;

head->data=1;

line \* list=head;

for (int i=2; i<=3; i++) {

//创建并初始化一个新结点

line \* body=(line\*)malloc(sizeof(line));

body->prior=NULL;

body->next=NULL;

body->data=i;

list->next=body;//直接前趋结点的next指针指向新结点

body->prior=list;//新结点指向直接前趋结点

list=list->next;

}

return head;

}

// 遍历输出双向链表的元素

void display(line \* head){

line \* temp=head;

while (temp) {

//如果该节点无后继节点，说明此节点是链表的最后一个节点

if (temp->next==NULL) {

printf("%d\n",temp->data);

}else{

printf("%d <-> ",temp->data);

}

temp=temp->next;

}

}

// 使用双向链表的一个示范如下

int main() {

line \* head=NULL; //创建一个头指针

head=initLine(head); //调用链表创建函数

display(head); //输出创建好的链表

//显示双链表的优点

printf("链表中第 4 个节点的直接前驱是：%d",head->next->next->next->prior->data);

return 0;

}

1. **循环单链表**

typedef struct node

{

int data;

struct node \*next; //指针域

int size; //循环链表的长度

}node,\*linklist; //linklist为定义的指针结构体变量

* ****创建循环链表（头插法/尾插法）****
* ****打印输出****
* ****示范****

//头插法建立循环链表

void create\_list\_head(linklist \*l){

int i, j, x;

linklist p,q;

printf("please input length:");

scanf("%d",&j);

(\*l)=(node\*)malloc(sizeof(node));

(\*l)->next=NULL; //必须定义，缺少error

(\*l)->size=j;

for(i=0;i<j;i++){ //头插法输入数据

scanf("%d",&x);

p=(node\*)malloc(sizeof(node));

p->data=x;

p->next=(\*l)->next;

(\*l)->next=p;

}

//q=(node\*)malloc(sizeof(node));

q=(\*l)->next;

while(q->next!=NULL)//找到最后一个结点

{

q=q->next;

}

//printf("%d",q->data);

q->next=(\*l)->next;//将最后一个结点指向第一个结点

}

//尾插法建立循环链表

void create\_list\_tail(linklist \*l){

int i,j,x;

linklist p,r,t;

printf("please input the length");

scanf("%d",&j);

(\*l)=(node\*)malloc(sizeof(node));

(\*l)->next=NULL;

(\*l)->size=j;

t=(\*l);

for(i=0;i<j;i++){

scanf("%d",&x);

p=(node\*)malloc(sizeof(node));

p->data=x;

t->next=p;

t=p; //每新建一个结点在结束时都为最后一个结点，下一个节点在此结点后插入

}

t->next=NULL;

r=(\*l)->next;

while(r->next!=NULL)

{

r=r->next;

}

r->next=(\*l)->next;

}

//打印输出

void print\_list\_he(linklist \*l){

int i;

linklist p;

p=(\*l)->next;

for(i=0;i<(\*l)->size;i++)

{

printf("%d\n",p->data);

p=p->next;

}

}

// 应用的一个示例如下

int main(){

linklist a;

create\_list\_head(&a);

create\_list\_tail(&a);

print\_list\_he(&a);

return 0;

}

1. **栈和队列**

**1、链栈**

// 链栈由链顶指针S唯一确定

typedef struct SNode{

ElemType data;

struct SNode \*next;

}SNode, \*LinkedStack;

* ****链栈初始化****
* ****入栈****
* ****出栈****

// 链栈初始化

status InitStack(LinkedStack &S){

S = NULL;

return 1;

}

// 入栈（压栈）

status Push(LinkedStack &S, ElemType e){

LinkedStack p = (LinkedStack)malloc(sizeof(SNode));

if(!p) return 0;

p->data = e;

p->next = S;

S = p;

return 1;

}

// 出栈

status Pop(LinkedStack &S, ElemType &e){

// 判断是否是空栈

if(S == NULL) return 0;

e = S->data;

LinkedStack p = S;

S = S->next;

free(p);

return 1;

}

**2、顺序栈**

#define INIT\_SIZE 100 //存储空间初始的分配量

#define STACKINCREMENT 10 //存储空间的分配增量<相对初始分配量较小，视具体情况而定>

// 动态分配的顺序栈结构

typedef struct{

ElemType \*base;

ElemType \*top; // 一般指向栈顶元素的下一个位置

int stacksize; // 栈已分配空间

}SqStack;

//(1)栈结构不存在：S.base = NULL; (2)空栈：S.base = S.top; (3)栈满：S.top - S.base >= S.stacksize

* ****初始化顺序栈****
* ****压栈****
* ****出栈****

// 初始化顺序栈

status InitStack(SqStack \*S){

S.base=(ElemType\*)malloc(sizeof(ElemType)\*INIT\_SIZE);

if(!S.base)

{

return 0;

}

S.top = S.base-1;

S.stacksize=INIT\_SIZE;

return 1;

}

// 压栈操作

void Push(SqStack &S, ElemType e){

if( S.top - S.base >= S.stacksize )

{ /\*栈满，追加空间\*/

S.base = (Elemtype\*)realloc(S.base,(S.stacksize+INCREMENT)\*sizeof(ElemType));

if(S.base != NULL) return 0; //判断是否扩容成功

S.top = S.base + S.stacksize; //修改栈顶指针

S.stacksize += INCREMENT; //扩容后栈的存储容量

}

\*S.top++ = e; //将e入栈

}

// 出栈操作

status Pop(SqStack &S, ElemType &e){

if(S.top == S.base) return 0;

e = \*--S.top;

return 1;

}

**注：静态分配的数据结构如下，基本操作略**

**//静态分配的顺序栈结构**

typedef struct

{

ElemType data[MAXSIZE];

int top; //栈的已分配空间

}SqStack;

//(1)空栈：top = -1;

//(2)栈满：top = MAXSIZE-1

**3、链队列**

// 队列的结点结构

typedef struct QNode{

ElemType data;

struct QNode \*next;

}QNode, \*QueuePtr;

// 链队列LinkedQueue Q

typedef struct LinkedQueue{

QueuePtr front; //队头指针

QueuePtr rear; //队尾指针

};

// 队空：Q.front = Q.rear(有头结点）/ Q.front = NULL(无头结点)

* ****初始化队列****
* ****入队****
* ****出队****

// 初始化链队列

status InitQueue(LinkedQueue &Q){

Q.rear = (QueuePtr)malloc(sizeof(QNode));

Q.front = (QueuePtr)malloc(sizeof(QNode));

if(!Q.front) return 0;

Q.front->next = NULL;

return 1;

}

// 入队列 (有头结点)

status EnQueue(LinkedQueue &Q, ElemType e){

QueuePtr p = (QueuePtr)malloc(sizeof(QNode));

if(!p) return 0;

p->data = e;

p->next = NULL;

Q.rear->next = p;

Q.rear = p;

return 1;

}

// 出队列 (有头结点)

status DeQueue(LinkedQueue &Q, ElemType &e){

if(Q.front == Q.rear) return 0;

QueuePtr p = Q.front->next;

e = p->data;

Q.front->next = p->next;

if(Q.rear == p) Q.rear = Q.front;

free(p);

return 1;

}

**4、循环队列**

typedef struct SqQueue{

ElemType \*base;

int front;

int rear;

};

/\*

约定：front始终指向队头元素，rear则指向队尾元素的下一个位置

（1）空队列Q.front = Q.rear; （2）入队列：Q.rear++; （3）出队列：Q.front++；

\*/

* ****初始化队列****
* ****入队****
* ****出队****

// 初始化循环队列

status InitQueue(SqQueue &Q){

Q.base = (ElemType \*) malloc(sizeof(ElemType)\*MAXSIZE);

if(!Q.base) return 0;

Q.front = Q.rear;

return 1;

}

// 入队列

status EnQueue(SqQueue &Q, ElemType e) {

if((Q.rear+1)%MAXSIZE) == Q.front) return 0;

Q.base[Q.rear] = e;

Q.rear = (Q.rear+1) % MAXSIZE;

return 1;

}

// 出队列

status DeQueue(SqQueue &Q, ElemType &e) {

if(Q.front == Q.rear) return 0;

e = Q.base[Q.front];

Q.front = (Q.front+1) % MAXSIZE;

return 1;

}

1. **串**

**1、顺序存储结构的串**

#define MAX\_NUM 100

// 串的顺序存储结构体

typedef struct str{

char ch[MAX\_NUM+1];

int length;

}\*SString;

* ****字符串初始化****
* ****判断字符串是否为空****
* ****字符串赋值****
* ****打印字符串****
* ****字符串复制****
* ****字符串连接****
* ****获取字符串的长度****
* ****在串str1的pos位置插入串str2****
* ****删除指定位置的字符****
* ****求子串****
* ****销毁字符串****
* ****示例****

//字符串初始化

SString initSString(){

SString str=(SString)malloc(sizeof(struct str));

if(!str)

printf("动态内存分配失败!\n");

str->length=0;

// printf("串初始化成功!\n");

return str;

}

//判断字符串是否为空

int isEmpty(SString str){

if(str->length==0)

{

return 1;

}

return 0;

}

//字符串赋值

SString strAssign(SString str, char s[]){

int i=0;

while(s[i])

{

str->ch[++str->length]=s[i];

i++;

}

return str;

}

//打印字符串

void printSString(SString str){

int i;

for(i=1;i<=str->length;i++)

printf("%c",str->ch[i]);

printf("\n");

}

//字符串复制

SString strCopy(SString str1,SString str2){

int i;

for(i=1;i<=str2->length;i++)

{

str1->ch[i]=str2->ch[i];

}

str1->length=str2->length;

return str1;

}

// 字符串连接

SString strCat(SString str1,SString str2){

SString str3=initSString();

int i;

for(i=1;i<=str1->length;i++)

str3->ch[i]=str1->ch[i];

for(i=1;i<=str2->length;i++)

str3->ch[i+str1->length]=str2->ch[i];

str3->length=str1->length+str2->length;

return str3;

}

//获取字符串的长度

int getStrLength(SString str){

return str->length;

}

//在串str1的pos位置插入串str2

SString strInsert(SString str1,int pos,SString str2){

if(pos<1||pos>str1->length){

printf("插入位置错误!\n");

exit(0);

}

int i;

for(i=pos;i<=str1->length;i++){

str1->ch[i+str2->length]=str1->ch[i];

}

for(i=1;i<=str2->length;i++)

{

str1->ch[i+pos-1]=str2->ch[i];

}

str1->length=str2->length+str1->length;

return str1;

}

//删除指定位置的字符

SString strDelete(SString str,int pos,int len){

if(pos<1||pos>str->length||len+pos-1>str->length){

printf("删除位置错误!\n");

exit(0);

}

int i;

for(i=pos+len;i<=str->length;i++){

str->ch[i-len]=str->ch[i];

}

str->length=str->length-len;

return str;

}

//求子串

SString subString(SString str,int pos,int len)

{

if(pos<1||pos>str->length||len+pos-1>str->length){

printf("输入位置错误!\n");

exit(0);

}

int i;

SString subStr=initSString();

for(i=1;i<=len;i++){

subStr->ch[i]=str->ch[i+pos-1];

}

subStr->length=len;

return subStr;

}

//销毁字符串

void destorySString(SString str1){

free(str1);

}

// 示例

int main(){

SString str=initSString(),str2=initSString();

char s[100],s2[100],s3[100],s4[100];

printf("请输入一个字符串str1:\n");

scanf("%s",s);

str=strAssign(str,s);

printf("打印字符串str1:\n");

printSString(str);

printf("拷贝字符串str1到字符串str2\n");

strCopy(str2,str);

printf("打印字符串str2:\n");

printSString(str2);

/\*SString str3=strCat(str,str2);

printSString(str3);\*/

int insertLocation;

printf("请输入插入位置、插入字符串str4:\n");

scanf("%d",&insertLocation);

scanf("%s",s4);

SString str4=initSString();

str4=strAssign(str4,s4);

str=strInsert(str,insertLocation,str4);

printf("打印插入str4后的字符串str1:");

printSString(str);

int delLocation,delLength;

printf("请输入字符串str1删除字符的位置和长度:\n");

scanf("%d %d",&delLocation,&delLength);

strDelete(str,delLocation,delLength);

printSString(str);

int subLocation,subLength;

printf("请输入字符串str1的子串位置和长度:\n");

scanf("%d %d",&subLocation,&subLength);

SString subStr= subString(str,subLocation,subLength);

printSString(subStr);

printf("销毁字符串st1:\n");

destorySString(str);

return 0;

}

**2、动态分配的串**

// 堆分配存储的串

typedef struct {

char \*ch;

int length;

}HString;

* ****字符串初始化****
* ****获取字符串的长度****
* ****打印字符串****
* ****字符串赋值****
* ****求子串****
* ****串比较****
* ****子串匹配****
* ****串插入****
* ****串删除****

// 初始化串

void StrInit(HString &str){

str.length = 0;

str.ch = (char\*)malloc(MaxStrlen\*sizeof(char));

}

// 求串长(返回字符串的长度，并更新字符串的length属性)

int StrLength(HString &str){

char\* chs = str.ch;

// 记录str的长度并且赋值给str.length

int i = 0;

while(chs[i] != '\0') ++i;

str.length = i;

return i;

}

// 输出串的内容

void StrContent(HString str){

int i = 0;

while(str.ch[i] != '\0' && i<=str.length-1){

printf("%c", str.ch[i]);

++i;

}

}

// 给串赋值(s2赋值给s1，s1原来的值被覆盖掉)

int StrAssign(HString &s1, HString s2){

// 先清空s1的值和长度

if(s1.ch){

free(s1.ch);

}

s1.length = 0;

char\* chs = s2.ch; // 记录s2的长度并且赋值

s1.length = s2.length;

s1.ch = (char\*)malloc((s1.length+1)\*sizeof(char)); // 重新分配空间

if(s1.ch == NULL) return 0; // 分配失败

// 开始赋值

int j = 0;

while(chs[j] != '\0'){

s1.ch[j] = chs[j];

++j;

}

return 1;

}

// 求子串(返回字符串str在索引index处长度为len的子串)

HString SubStr(HString str, int index, int len){

HString subs; // subs用于储存子串

StrInit(subs);

// 保证该子串存在

if(index<0 || len<0 || index>str.length-1 || index+len>str.length-1){

printf("对应子串不存在！");

return subs;

}

// 确定子串长度

subs.length = len;

int i = index;

while(str.ch[i] != '\0' && i<=str.length-1){

subs.ch[i-index] = str.ch[i];

++i;

}

return subs;

}

// 串比较（如果相等，返回0；前者大，返回1；后者大，返回-1 ）

int StrCmp(HString str1, HString str2){

// 用chs1和chs2指针指向两个串

char\* chs1 = str1.ch;

char\* chs2 = str2.ch;

int i = 0;

while(chs1[i] != '\0' && chs2[i] != '\0' ){

if(chs1[i] < chs2[i]) return 1;

else if(chs1[i] > chs2[i]) return -1;

++i;

}

// 如果相等，返回0；前者大，返回1；后者大，返回-1

if(chs1[i] == '\0' && chs2[i] == '\0' ) return 0;

else if(chs1[i] == '\0' && chs2[i] != '\0' ) return -1;

else if(chs1[i] != '\0' && chs2[i] == '\0' ) return 1;

return 0;

}

// 子串匹配（返回第一个字符的索引，如果无法匹配则返回-1）

int StrIndex(HString S, HString P){

int i = 0, j = 0; // i为循环参量，j记录已匹配到的长度

while (i < S.length && j < P.length){

if(S.ch[i] == P.ch[j]){ // 成功匹配到一个字符

++i;

++j;

} else{ // 未匹配到对应字符

i = i-j+1;

j = 0;

}

}

if (j>=P.length) return (i-P.length); //匹配成功

return -1; // 匹配失败

}

// 串插入(在str1的index索引处插入字符串str2，str1内容改变)

int StrInsert(HString &str1, int index, HString str2){

// 判断index是否正确

if(index<0 || index>str1.length-1){

printf("index数值错误！");

return 0;

}

// str储存改变后的字符串，然后赋值给str1

HString str;

str.length = 0;

str.ch = (char\*)malloc((str1.length+str2.length+1)\*sizeof(char));

int i = 0; // 定义声明i和j

// 把index之前str1的部分赋值给str

for(i=0; i<=index-1; i++){

str.ch[i] = str1.ch[i];

++str.length;

}

// 把str2赋值给str(拼接在后面)

for(i=index; i<=index+str2.length-1; i++){

str.ch[i] = str2.ch[i-index];

++str.length;

}

// 把index之后str1的部分赋值给str(拼接在后面)

for(i=index+str2.length; i<=index+str2.length+str1.length-1; i++){

str.ch[i] = str1.ch[i-str2.length];

++str.length;

}

str1 = str;

return 1;

}

// 串删除(在str的index索引处删除长度为len的字符串，str1内容改变)

int StrDelete(HString &str1, int index, int len){

// 保证该子串存在

if(index<0 || len<0 || index>str1.length-1 || index+len>str1.length-1){

return 0; // printf("对应子串不存在！");

}

// str储存改变后的字符串，然后赋值给str1

HString str;

str.length = str1.length-len;

str.ch = (char\*)malloc((str1.length-len+1)\*sizeof(char));

// 定义声明i和j

int i = 0;

// 把index之前str1的部分赋值给str

for(i=0; i<=index-1; i++){

str.ch[i] = str1.ch[i];

}

// 把index+len-1之后str1的部分赋值给str

for(i=index+len; i<=str1.length-1; i++){

str.ch[i-len] = str1.ch[i];

}

str1 = str;

return 1;

}

1. **树**

// 二叉树结构体

typedef struct TreeNode {

int val; // 默认大于0

struct TreeNode \*left;

struct TreeNode \*right;

} BiNode, \*BiTree;

* ****创建二叉树****
* ****先中后序遍历****
* ****求树的高度****
* ****求叶子节点的数目****

// 创建二叉树

struct TreeNode\* Create(){

int val;

scanf("%d", &val);

struct TreeNode\* root = (struct TreeNode\*)malloc(sizeof(struct TreeNode\*));

if (val <= 0) return NULL; // 若输入数据不大于0，则视该点为空

if (!root) {

printf("创建失败\n");

}

if (val > 0) {

root->val = val;

root->left = Create();

root->right = Create();

}

return root;

}

//先序遍历

void PreOrderTree(struct TreeNode\* root) {

if (root == NULL) return;

printf("%d ", root->val);

PreOrderTree(root->left);

PreOrderTree(root->right);

}

//中序遍历

void InOrderTree(struct TreeNode\* root) {

if (root == NULL) return;

InOrderTree(root->left);

printf("%d ", root->val);

InOrderTree(root->right);

}

//后序遍历

void PostOrderTree(struct TreeNode\* root) {

if (root == NULL) return;

PostOrderTree(root->left);

PostOrderTree(root->right);

printf("%d ", root->val);

}

//求树的高度

int maxDepth(struct TreeNode\* root) {

if (root == NULL) return 0;

else {

int maxLeft = maxDepth(root->left), maxRight = maxDepth(root->right);

if (maxLeft > maxRight) {

return 1 + maxLeft;

}

else {

return 1 + maxRight;

}

}

}

//求叶子节点个数

int LeafNodeNum(struct TreeNode\* root) {

if (root == NULL) {

return 0;

}

if (root->left == NULL&&root->right == NULL) {

return 1;

}

else {

return LeafNodeNum(root->left) + LeafNodeNum(root->right);

}

}

1. **查找**

**1、顺序查找和折半查找**

// 顺序查找（若找不到就返回0）

Status Order(int a[], int n, int key){

a[0] = key;

while(a[n] != key) --n;

return n;

}

// 折半查找（若找不到就返回0）

Status Half(int a[], int n, int key){

int i, j, t;

int low, high, mid;

// 冒泡排序,从小到大

for(i=1; i<n; i++){

for(j=1; j<n; j++){

if(a[j]>a[j+1]){

t = a[j];

a[j] = a[j+1];

a[j+1] = t;

}

}

}

low = 1;

high = n;

while(low<=high){

mid = low + (high-low)/2;

if (a[mid]==key) return mid;

else if(key<a[mid]) high = mid-1;

else low = mid+1;

}

return 0;

}

// 示例

int main(){

int a[1000], i, n, key;

printf("输入元素个数：");

scanf("%d", &n);

printf("请依次输入元素：");

for(i=1; i<=n; i++) scanf("%d", &a[i]);

printf("请输入要查找的元素:");

scanf("%d", &key);

printf("顺序查找-该元素的位置：");

printf("%d\n", Order(a, n, key));

printf("折半查找-该元素的位置：");

printf("%d\n", Half(a, n, key));

return 0;

}

1. **二叉排序树**

typedef struct BiTNode{

TElemType data;

struct BiTNode \*Lchild, \*Rchild;

}BiTNode, \*BiTree; // 左小右大

* ****查找****
* ****插入****
* ****删除****
* ****中序遍历****
* ****示例****

/\*

二叉排序树的查找（f指向T的父亲节点）

如果查找成功,则p指向该元素节点,并返回1

如果没有成功,则p指向查找路径的最后一个节点，并返回0

\*/

Status searchKey(BiTree T, int key, BiTree f, BiTree &p){

if (!T){

p = f;

return 0;

}

else if (key == T->data){

p = T;

return 1;

}

else if (key < T->data)

return searchKey(T->Lchild, key, T, p);

else

return searchKey(T->Rchild, key, T, p);

}

//二叉排序树的插入

Status insertKey(BiTree &T, int key){

BiTree p, s;

// search操作执行后,T指向该插入位置的父亲

if (searchKey(T, key, NULL, p) == 0){

s = (BiTree)malloc(sizeof(BiTNode));

s->data = key;

s->Lchild = s->Rchild = NULL;

// 若p为空，说明该树为空,则应在根节点处插入

if (!p) T = s;

else if(key < p->data) p->Lchild = s;

else p->Rchild = s;

return 1; //如果插入的值已经存在，则返回0；否则返回1

}

else return 0;

}

/\*

二叉排序树的删除——指定节点的删除

若p节点的右子树为空,则连接左子树；若p节点的左子树为空,则连接右子树；

若p节点的两个子树都是非空的,则用左子树最右的节点,替换待删除的节点

\*/

Status Delete(BiTree &p){

// q/s为辅助删除的节点

BiTree q, s;

// 若右子树空

if (p->Rchild == NULL){

q = p;

p = p->Lchild;

free(q);

}

// 若左子树空

else if (p->Lchild == NULL){

q = p;

p = p->Rchild;

free(q);

}

// 若两个子树都非空

else{

q = p;

s = p->Lchild;

while (s->Rchild){

q = s; // q指向左子树的最右孩子的父亲节点

s = s->Rchild; // s指向左子树的最右孩子

}

p->data = s->data;

// 若左子树最右的孩子有左孩子，连接q的右子树

if (q != p) q->Rchild = s->Lchild;

// 若左子树最右的孩子没有右孩子，连接q的左子树

else q->Lchild = s->Lchild;

free(s);

}

return 1; // 返回1则为删除成功

}

//二叉排序树的删除——根据已知值删除节点

Status deleteKey(BiTree &T, int key){

// 若二叉排序树为空

if (!T) return 0; //若二叉树存在与key相等的节点,则删除该节点,返回1；否则返回0

else{

if (key == T->data) return Delete(T);

else if (key < T->data) return deleteKey(T->Lchild, key);

else return deleteKey(T->Rchild, key);

}

}

// 中序遍历

void InOrderTraverse(BiTree T){

if (T){

InOrderTraverse(T->Lchild);

printf("%d ", T->data);

InOrderTraverse(T->Rchild);

}

}

// Demo测试主函数

int main(){

// 循环操作，保证程序可反复执行，如果一开始输入0则停止执行

while (1){

int n = 0, Key;

BiTree p;

BiTree T = NULL;

BiTree f = NULL;

// 二叉排序树的创建

printf("【二叉排序树的创建】\n");

printf("请输入节点个数：");

scanf("%d", &n);

if (n == 0) break;

printf("请输入这%d个节点的值：", n);

for (int i = 0; i < n; i++){

int a;

scanf("%d", &a);

insertKey(T, a);

}

printf("二叉排序树创建完成！\n该二叉树中序遍历结果为：");

InOrderTraverse(T);

puts("");

// 二叉排序树的查找/插入（如果能查找到，则返回是否能查找到;否则插入该值）

printf("【二叉排序树的查找/插入】\n");

printf("请输入待查找的关键值：");

scanf("%d", &Key);

if (searchKey(T, Key, f, p))

printf("关键值%d在二叉排序树中！\n", Key);

else{

printf("关键值%d不在二叉排序树中！\n", Key);

printf("则在二叉排序树中插入该关键值\n");

insertKey(T, Key);

printf("插入后，该二叉树中序遍历结果为：", Key);

InOrderTraverse(T);

printf("\n");

}

// 二叉排序树的删除

printf("【二叉排序树的删除】\n");

printf("请输入待删除的关键值：");

scanf("%d", &Key);

if (searchKey(T, Key, f, p)){

printf("删除关键值%d\n", Key);

deleteKey(T, Key);

printf("删除后，该二叉树中序遍历结果为：");

InOrderTraverse(T);

printf("\n\n");

}

else

printf("关键值%d不在二叉排序树中\n", Key);

}

return 0;

}

1. **排序**

**1、简单排序**

#include<sys/time.h>

// 顺序表结构体

typedef struct {

Elemtype data[MaxSize];

int length;

}SqList;

* ****冒泡排序****
* ****简单排序****
* ****插入排序****
* ****示例****

// 初始化顺序表

void InitList(SqList &L){

int i = 0;

L.length = 0;

for(i=0; i<=MaxSize-1; i++){ // 为每一项赋值（随机整数）

L.data[i] = rand();

L.length++;

}

}

// 冒泡排序-升序(升序),exNum表示进行交换的次数,coNum表示进行比较的次数

void BubbleSorting(SqList L){

InitList(L); // 初始化顺序表

printf("【冒泡排序】\n");

int i = 0, j = 0, t = 0;

int exNum = 0, coNum = 0; // 先把次数归零

// 开始冒泡排序

for(i=0; i<=MaxSize-2; i++){

for(j=0; j<=MaxSize-2; j++){

if(L.data[j] >= L.data[j+1]){

t = L.data[j];

L.data[j] = L.data[j+1];

L.data[j+1] = t;

exNum = exNum+1; // 如果需要交换次序，则交换次数加一

}

coNum = coNum+1;

}

}

printf("比较次数：%d\n", coNum);

printf("交换次数：%d\n", exNum);

}

// 简单排序-升序(升序),exNum表示进行交换的次数,coNum表示进行比较的次数

void SelectSorting(SqList L){

// 初始化顺序表

InitList(L);

printf("【简单排序】\n");

int i = 0, j = 0, t = 0;

int min = L.data[0];

int minn = 0;

// 先把次数归零

int exNum = 0, coNum = 0;

// 开始简单排序

for(i=0; i<=MaxSize-1; i++){

min = L.data[i];

minn = i;

for(j=i; j<=9999; j++){

if(min>=L.data[j]){

min = L.data[j];

minn = j;

exNum++;

}

coNum++;

}

t = L.data[i];

L.data[i] = L.data[minn];

L.data[minn] = t;

}

printf("比较次数：%d\n", coNum);

printf("交换次数：%d\n", exNum);

}

// 插入排序-升序(升序),exNum表示进行交换的次数,coNum表示进行比较的次数

void InsertSorting(SqList L){

InitList(L); // 初始化顺序表

printf("【插入排序】\n");

int i=0, j=0, t=0;

// 先把次数归零

int exNum = 0, coNum = 0;

// 开始插入排序

for(i=1; i<=MaxSize-1; i++){

if(L.data[i]<L.data[i-1]){

t = L.data[i];

L.data[i] = L.data[i-1];

exNum++;

for(j=i-1; t<=L.data[j] && j>=0; j--){

L.data[j+1] = L.data[j];

exNum++;

coNum++;

}

L.data[j+1] = t;

exNum++;

coNum++;

}

coNum++;

}

printf("比较次数：%d\n", coNum);

printf("交换次数：%d\n", exNum);

}

// 示例

int main(){

struct timeval start;

struct timeval end;

//记录两个时间差

unsigned long diff;

// 声明并初始化顺序表

SqList L;

// 进行冒泡排序

gettimeofday(&start, NULL);

BubbleSorting(L);

gettimeofday(&end,NULL);

diff = 1000000 \* (end.tv\_sec-start.tv\_sec)+ end.tv\_usec-start.tv\_usec;

printf("程序耗时：%ld\n", diff);

// 进行简单排序

gettimeofday(&start, NULL);

SelectSorting(L);

gettimeofday(&end,NULL);

diff = 1000000 \* (end.tv\_sec-start.tv\_sec)+ end.tv\_usec-start.tv\_usec;

printf("程序耗时：%ld\n", diff);

// 进行插入排序

gettimeofday(&start, NULL);

InsertSorting(L);

gettimeofday(&end,NULL);

diff = 1000000 \* (end.tv\_sec-start.tv\_sec)+ end.tv\_usec-start.tv\_usec;

printf("程序耗时：%ld\n", diff);

return 0;

}

**2、快速排序**

// 顺序表结构体

typedef struct {

Elemtype data[MaxSize];

int length;

}SqList;

// 初始化顺序表

void InitList(SqList &L){

int i = 0;

L.length = 0;

// 为每一项赋值（随机整数）

for(i=0; i<=MaxSize-1; i++){

L.data[i] = rand();

L.length++;

}

}

// 划分顺序表

int Partition(SqList &L, int low, int high, int &coNum, int &exNum){

int pivot = L.data[low]; // 枢轴元素（初始取第一个元素）

int i = low, j = high;

while(i<j){

while(i<j && L.data[j]>=pivot){

j--; // 从后往前寻找比枢轴元素小的元素

coNum++;

}

L.data[i] = L.data[j];

exNum++;

while(i<j && L.data[i]<=pivot){

i++; // 从前往后寻找比枢轴元素大的元素

coNum++;

}

L.data[j] = L.data[i];

exNum++;

}

L.data[i] = pivot;

return i; // 返回枢轴元素的位置

}

// 快速排序(对从low到high的元素进行排序)

int\* QuickSorting(SqList &L, int low, int high){

int num[2] = {0};

int coNum=0, exNum=0; // 先把次数归零

if(low<high){

int pivotloc = Partition(L, low, high, coNum, exNum);

QuickSorting(L, low, pivotloc-1);

QuickSorting(L, pivotloc+1, high);

}

num[0] = coNum;

num[1] = exNum;

return num;

}

// 示例

int main(){

struct timeval start;

struct timeval end;

//记录两个时间差

unsigned long diff;

// 声明并初始化顺序表

SqList L;

InitList(L);

// 进行快速排序

printf("【快速排序】\n");

gettimeofday(&start, NULL);

int\* num = QuickSorting(L, 0, MaxSize-1);

printf("比较次数：%d\n", num[0]);

printf("交换次数：%d\n", num[1]);

gettimeofday(&end,NULL);

diff = 1000000 \* (end.tv\_sec-start.tv\_sec)+ end.tv\_usec-start.tv\_usec;

printf("程序耗时：%ld\n", diff);

return 0;

}

1. **一些经典问题**
2. **约瑟夫环（循环链表）**

typedef int Status;

typedef int ElemType;

// 对于循环链表(结点)的定义

typedef struct LNode{

ElemType num;

ElemType pwd;

struct LNode\* next;

}LNode, \*LinkedList;

typedef LNode\* LinkedList;

// 初始化循环单链表（输入的参数n表示结点个数）

LinkedList InitList(int n){

int i = 1; // 当前结点个数

LinkedList head = (LinkedList)malloc(sizeof(LNode)); // 创建头结点（并不是真的头结点）

head->num = 1;

head->next = NULL;

LinkedList p = head; // 指示当前结点

// 依次创建后面的结点，并输入对应编号和密码

printf("请依次输入每个人的密码：");

if(i == 1){

scanf("%d", &p->pwd);

}

for(i=2;i<=n;i++){

p->next = (LinkedList)malloc(sizeof(LNode));

p = p->next;

scanf("%d", &p->pwd);

p->num = i;

p->next = NULL;

}

p->next = head; // 首尾相连

return head;

}

// 删除从当前结点开始起的第m个元素，并输出其编号，同时更换m值

Status DeleteElem(LinkedList &p, int m){

int flag = 1; // 指示结点次序

LinkedList q = NULL; // 用于标识指定结点

while(p->next!=p){

if(flag == m-1){ // 删除节点，输出其编号

m = p->next->pwd; // 用指定结点的pwd代替m

printf("%d ",p->next->num);

q = p->next; // q指向需要被删去的结点

p->next = q->next; // p指向被删结点的下一个结点

free(q); // 释放内存

p = p->next;

flag = 1; // 重置flag

} else{

p = p->next; // 循环定位第m-1个结点

++flag;

}

} /\*while\*/

return m; // 返回改变后的m

}

int main(){

int m = 1,n = 1; // m为给定的数，n为结点个数

printf("请输入m和n的值(空格隔开)：");

scanf("%d %d", &m, &n);

LinkedList L = InitList(n); // 初始化循环单链表

LinkedList p = L; // p用于指示当前结点

// 开始依次删除元素

printf("被删除者的编号依次为：");

while(p->next!=p){

m = DeleteElem(p, m);

}

printf("%d ",p->next->num);

free(p);

return 0;

}

1. **迷宫问题（栈）**

#define STACK\_INIT\_SIZE 100 // 储存空间初始分配量

#define STACKINCREMENT 10 // 存储空间分配增量

#define MAXLENGTH 25 // 迷宫最大行、列数

#define INIT\_SIZE 10

typedef int ElemType;

typedef int Status;

// 迷宫坐标

typedef struct {

ElemType x; // 行值

ElemType y; // 列值

}PosType;

// 栈元素

typedef struct {

ElemType ord; // 序号

ElemType di; // 方向（0123表示移动方向依次为东南西北）

PosType seat; // 位置

} SElemType;

// 顺序栈

typedef struct {

SElemType \*base; // 栈底指针

SElemType \*top; // 栈顶指针

ElemType stacksize; // 当前已分配的存储空间

}SqStack;

// 全局变量

typedef int MazeType[MAXLENGTH][MAXLENGTH]; // 迷宫数组[行][列]

PosType begin, end; // 迷宫入口坐标，出口坐标

PosType direc[4] = { {0,1},{1,0},{0,-1},{-1,0} }; // {行增量，列增量}，移动方向依次为东南西北

MazeType maze; // 迷宫数组

int x, y; // 迷宫的行、列

int curstep = 1; // 当前足迹

// 栈的初始化

Status InitStack(SqStack \*p) {

p->base = (SElemType\*)malloc(STACK\_INIT\_SIZE \* sizeof(SElemType));

if (p->base == NULL) return ERROR; // 内存分配失败

p->top = p->base; // 设为空栈

p->stacksize = STACK\_INIT\_SIZE;

return OK;

}

// 判断栈是否为空

Status EmptyStack(SqStack \*p) {

if (p->top == p->base) return OK;

else return ERROR;

}

// 压栈，e为指定加入的栈元素

Status Push(SqStack \*p, SElemType e) {

// 如果栈满，则追加储存空间

if ((p->top - p->base) >= p->stacksize) { p->base=(SElemType\*)realloc(p->base,(p->stacksize+STACKINCREMENT)\*sizeof(SElemType));

if (p->base == NULL) return ERROR; // 储存空间分配失败

p->top = p->base + p->stacksize;

p->stacksize += STACKINCREMENT; // 追加空间

}

\*(p->top) = e;

(p->top)++; // top指针指向下一个元素

return OK;

}

// 出栈

Status Pop(SqStack \*p, SElemType \*e) {

// 若栈不为空，则删除栈顶元素，用e表示其值

if (p->top == p->base) return ERROR;

--(p->top);

\*e = \*(p->top);

return OK;

}

// 清空栈

Status ClearStack(SqStack \*p) {

p->top = p->base;

return OK;

}

// 销毁栈

Status DestroyStack(SqStack \*p) {

free(p->base); // 释放栈底空间并置空（空间置零）

p->base = NULL;

p->top = NULL;

p->stacksize = 0;

return OK;

}

// 输出当前迷宫的各个单元格

void Print() {

int i, j; // 循环参量

for (i = 0; i < x; i++)

{

for (j = 0; j < y; j++)

printf("%3d", maze[i][j]);

printf("\n");

}

}

// 初始化迷宫

void Init() {

int i, j, x1, y1;

printf("请输入迷宫的行数和列数（行数 列数）(含外墙):");

scanf("%d %d", &x, &y);

for (i = 1; i < x - 1; i++)

for (j = 1; j < y - 1; j++)

maze[i][j] = 1; //定义通道初值为1

printf("请输入迷宫内墙所占的单元数:");

scanf("%d", &j);

printf("请输入迷宫内墙每个单元的坐标（行数 列数）:\n");

for(i=1;i<=j;i++) {

scanf("%d %d", &x1, &y1);

maze[x1][y1] = 0;

}

printf("迷宫的结构:\n");

Print();

printf("请输入入口的坐标（行数 列数）:");

scanf("%d %d", &begin.x, &begin.y);

printf("请输入出口的坐标（行数 列数）:");

scanf("%d %d", &end.x, &end.y);

}

// 标记迷宫该单元格（区块）不可通过

void MarkPrint(PosType seat){

maze[seat.x][seat.y] = -1;

}

// 当迷宫b点的值为1，返回OK，否则返回ERROR;

Status Pass(PosType b) {

if (maze[b.x][b.y] == 1) return OK;

else return ERROR;

}

// 把迷宫b点的值标记为足迹

void FootPrint(PosType b) {

maze[b.x][b.y] = curstep;

}

// 根据当前位置b及移动方向di，令b为下一位置

PosType NextPos(PosType b, int di) {

b.x += direc[di].x;

b.y += direc[di].y;

return b;

}

// 若迷宫m中存在一条通道，则求得一条路径存在栈中，并返回TRUE；否则返回FAlSE

Status MazePath(PosType start, PosType end) {

SqStack s, \*S;

S = &s;

PosType curpos;

SElemType e,\*pe;

pe = &e;

InitStack(S);

curpos = start;

while (!EmptyStack(S)) {

if (Pass(curpos)) { // 若当前可通过，则为未曾走的通道块

// 留下足迹并标记

FootPrint(curpos);

e.ord = curstep;

e.seat = curpos;

// 从当前位置出发，下一位置为东

e.di = 0;

Push(S, e); // 足迹加1

curstep++;

// 若已到达出口

if (curpos.x == end.x&&curpos.y == end.y) return TRUE;

// 由当前位置及移动方向确定下一个当前位置

curpos = NextPos(curpos, e.di);

}

else{ // 若当前位置不能通过

if (!EmptyStack(S)) { // 若栈不为空

Pop(S, pe); // 退栈到前一位置

curstep--;

// 当前一位置处于最后一个方向(北)

while (e.di == 3 && !EmptyStack(S)) {

// 留下不能通过的标记(-1)

MarkPrint(e.seat);

// 退一步

Pop(S, pe);

curstep--;

}

// 若未到最后一个方向(3)

if (e.di < 3) {

e.di++; // 换下一方向

Push(S, e); // 入栈下一个方向

curstep++; // 足迹加1

curpos = NextPos(e.seat, e.di);

}

}

}

}

return FALSE;

}

int main() {

Init(); // 初始化迷宫

if (MazePath(begin, end)) { // 若存在通路，则输出一条路径

printf("该迷宫的一条路径如下:\n");

Print();

}

else printf("该迷宫不存在可行的路径\n");

return 0;

}

1. **多项式计算（链表）**

typedef int Status;

typedef int ElemType;

// 对于单链表(结点)的定义

typedef struct LNode{

double coef; // 系数

ElemType expn; // 指数

struct LNode\* next;

}LNode, \*LinkedList;

typedef LNode\* Polynomial;

// 初始化多项式,其中n为项数

Polynomial InitPolynomial(int n){

int i = 1; // 循环参量i

Polynomial head = (Polynomial)malloc(sizeof(LNode)); // 创建头节点

head->coef = n; // 头结点的系数表示多项式的项数

head->next = NULL;

// p指示当前结点

Polynomial p = head;

for(i=1;i<=n;i++){

Polynomial node = (Polynomial)malloc(sizeof(LNode));

scanf("%lf,%d", &node->coef, &node->expn);

node->next = NULL;

p->next = node;

p = p->next;

}

return head;

}

// 两个多项式求和

Polynomial AddPolyn(Polynomial &Pa, Polynomial &Pb) {

// p1, p2分别指示两个多项式的当前结点

Polynomial p1 = Pa->next;

Polynomial p2 = Pb->next;

// 创建一个新的多项式，用于储存多项式的和

Polynomial new\_head = (Polynomial)malloc(sizeof(LNode));

new\_head->next = NULL;

Polynomial p3 = new\_head;

while(p1 && p2){

int a = p1->expn;

int b = p2->expn;

// 创建新多项式的新节点

Polynomial new\_node = (Polynomial)malloc(sizeof(LNode));

// 若两项指数相等，则系数相加（若系数和为0，则该项不计入结果）

if(a==b && (p1->coef + p2->coef) != 0){

new\_node->expn = a;

new\_node->coef = p1->coef + p2->coef;

p1 = p1->next;

p2 = p2->next;

} else if(a==b && (p1->coef + p2->coef) == 0) {

p1 = p1->next;

p2 = p2->next;

continue;

}

// 若两项指数不相等，则取指数较小的一项

else if(a>b) {

new\_node->expn = b;

new\_node->expn = p2->coef;

p2 = p2->next;

} else if(a<b) {

new\_node->expn = a;

new\_node->expn = p1->coef;

p1 = p1->next;

}

// 把新结点添加到新多项式里

new\_node->next = NULL;

p3->next = new\_node;

p3 = p3->next;

} /\*while\*/

// 若其中一个多项式已为空

while(p1 && !p2){

// 创建新多项式的新节点

p3->next = (Polynomial)malloc(sizeof(LNode));

// 把新结点添加到新多项式里

p3 = p3->next;

p3->next = NULL;

p3->expn = p1->expn;

p3->coef = p1->coef;

p1 = p1->next;

}

while(p2 && !p1){

p3->next = (Polynomial)malloc(sizeof(LNode)); // 创建新多项式的新节点

p3 = p3->next; // 把新结点添加到新多项式里

p3->next = NULL;

p3->expn = p2->expn;

p3->coef = p2->coef;

p2 = p2->next;

}

// 返回新多项式的头结点

return new\_head;

}

// 两个多项式求差 （前减后）

Polynomial SubPolyn(Polynomial &Pa, Polynomial &Pb) {

// p1, p2分别指示两个多项式的当前结点

Polynomial p1 = Pa->next;

Polynomial p2 = Pb->next;

// 创建一个新的多项式，用于储存多项式的和

Polynomial new\_head = (Polynomial)malloc(sizeof(LNode));

new\_head->next = NULL;

Polynomial p3 = new\_head;

while(p1 && p2){

int a = p1->expn;

int b = p2->expn;

// 创建新多项式的新节点

Polynomial new\_node = (Polynomial)malloc(sizeof(LNode));

// 若两项指数相等，则系数相加（若系数和为0，则该项不计入结果）

if(a==b && (p1->coef - p2->coef) != 0){

new\_node->expn = a;

new\_node->coef = p1->coef - p2->coef;

p1 = p1->next;

p2 = p2->next;

} else if(a==b && (p1->coef - p2->coef) == 0) {

p1 = p1->next;

p2 = p2->next;

continue;

}

// 若两项指数不相等，则取指数较小的一项

else if(a>b) {

new\_node->expn = b;

new\_node->expn = -1.00\*p2->coef;

p2 = p2->next;

} else if(a<b) {

new\_node->expn = a;

new\_node->expn = p1->coef;

p1 = p1->next;

}

new\_node->next = NULL; // 把新结点添加到新多项式里

p3->next = new\_node;

p3 = p3->next;

} /\*while\*/

while(p1 && !p2){ // 若其中一个多项式已为空

p3->next = (Polynomial)malloc(sizeof(LNode)); // 创建新多项式的新节点

p3 = p3->next; // 把新结点添加到新多项式里

p3->next = NULL;

p3->expn = p1->expn;

p3->coef = p1->coef;

p1 = p1->next;

}

while(p2 && !p1){

p3->next = (Polynomial)malloc(sizeof(LNode)); // 创建新多项式的新节点

p3 = p3->next; // 把新结点添加到新多项式里

p3->next = NULL;

p3->expn = p2->expn;

p3->coef = -1.00\*p2->coef;

p2 = p2->next;

}

// 返回新多项式的头结点

return new\_head;

}

int main(){

// 循环参数i

int i = 1;

// 输入两个多项式的项数

int n1, n2;

printf("请分别输入两个多项式的非零项项数（空格隔开）：");

scanf("%d %d", &n1, &n2);

// 创建两个多项式,按照指数从小到大排列

printf("输入多项式时，每项按”系数,指数“的格式，不同项间空格隔开\n");

printf("请依次输入第一个多项式的系数及其指数：");

Polynomial head1 = InitPolynomial(n1);

printf("请依次输入第二个多项式的系数及其指数：");

Polynomial head2 = InitPolynomial(n2);

//多项式相加

Polynomial addition = AddPolyn(head1, head2);

Polynomial point1 = addition->next;

printf("相加后的多项式为：");

while(point1){

printf("%.2lf x^(%d) ", point1->coef, point1->expn);

if(!point1->next) printf("\n");

point1 = point1->next;

}

//多项式相减

Polynomial subtraction = SubPolyn(head1, head2);

Polynomial point2 = subtraction->next;

printf("相减后的多项式为：");

while(point2){

printf("%.2lf x^(%d) ", point2->coef, point2->expn);

point2 = point2->next;

}

return 0;

}

1. **文字研究助手（串，含KMP算法）**

#include <string.h>

#define MAXSTRLEN 255

#define OVERFLOW -2

// 堆分配存储的串

typedef struct {

char \*ch;

int length;

}HString;

// 将长度为len的字符数组cha转换成字符串，并返回字符串对应的指针

char\* ToString(char ch[], int len){

char\* p;

int i;

p = (char\*)malloc(len+1);

for(i=0;i<len;i++){

\*(p+i)='0'+ch[i];

}

\*(p+i) = '\0';

return p;

}

// 初始化串，令S为空串

void InitString(HString \*S){

(\*S).ch = NULL;

(\*S).length = 0;

}

// 串赋值，生成一个值等于常量chars(字符数组)的串S

int StrAssign(HString \*T, char\* chars){

int i, j;

InitString(T); //初始化串T

i = strlen(chars);

if(!i) return 0;

else{

(\*T).ch = (char\*)malloc(i\*sizeof(char)); //分配空间

if(!((\*T).ch)) exit(OVERFLOW);//分配失败

for(j=0;j<i;j++){

(\*T).ch[j] = chars[j];

}

(\*T).length = i;

}

return 1;

}

// 求T的next的函数值，并存入数组next中

void getNext(HString T, int next[]){

int i = 1, j = 0;

next[1] = 0;

while(i<T.length){

if(j==0 || T.ch[j-1]==T.ch[i-1]){

++i;

++j;

next[i] = j;

} else j = next[j];

}

}

// KMP算法：串利用next函数求,T在S中第pos个字符后的位置

int Index\_KMP(HString S, HString T, int pos){

int i = pos, j=1;

// 确保T非空，1<=pos<=StrLength(S)

while(i!=S.length+1&&j!=T.length+1){

if(j==0||S.ch[i-1]==T.ch[j-1]){ // 继续比较后面的字符

++i;

++j;

}

else j = next[j]; // 模式串后移

}

if(j>T.length-1) return (i-T.length); // 匹配成功

else return 0; // 匹配失败

}

// 查找单词words

void Find(HString words){

StrAssign(HString &T, char\* chars); // 生成串T和字符数组（指针）chars

int count=0; // count用于计数

HString T; // 声明并初始化T

InitString(T);

int i=1;j=0;

FILE \*f;

getNext(words, next);

while(!feof(f)){ // feof功检测流上的文件结束符，如果文件结束，则返回非0值

fgets(text, MAXSTRLEN, f);

ToString(text,sizeof(text)/sizeof(text[0]));

chars = text;

StrAssign(T,chars);

j = Index(T,words,j+1);

if(j!=0){

printf("行: %d, 列: %d",i,j);

count++;

}

while(j!=0){

j=Index(T,words,j+1);

if(j!=0){

printf("行: %d, 列: %d",i,j);

count++;

}

}

i++;

}

printf("所查询单词的编号: %d", count);

}

**5、分解单链表**

#include<malloc.h>

typedef char datatype;

// 单链表结构

typedef struct node

{ datatype data;

struct node \*next;

}linklist;

void create(linklist\*&);

void resolve(linklist\*,linklist\*,linklist\*,linklist\*);

void insert(linklist\*,linklist\*);

void print1(linklist\*);

void print2(linklist\*);

int main(){

linklist \*head,\*letter,\*digit,\*other;

create(head);

print1(head);

letter=(linklist\*)malloc(sizeof(linklist)); //建立3个空循环链表

letter->next=letter;

digit=(linklist\*)malloc(sizeof(linklist));

digit->next=digit;

other=(linklist\*)malloc(sizeof(linklist));

other->next=other;

resolve(head,letter,digit,other);//调用分解单链表的函数

print2(letter);//输出循环链表

print2(digit);

print2(other);

}

//建立单链表

void create(linklist\*&head){

datatype x;

linklist \*s,\*r;

head=new linklist;

r=head;

while((x=getchar())!='\n'){

s=(linklist\*)malloc(sizeof(linklist));

s->data=x;

r->next=s;

r=s;

}

r->next=NULL;

}

//在循环链表中插入

void insert(linklist\*h,linklist\*p)

{ linklist \*q=h;

while(q->next!=h) q=q->next;

q->next=p;

p->next=h;

}

//输出单链表

void print1(linklist\*head){

linklist \*p=head->next;

while(p!=NULL){

printf("%c",p->data);

p=p->next;

}

printf("\n");

}

//输出循环链表

void print2(linklist\*head){

linklist \*p=head->next;

while(p!=head){

printf("%c",p->data);

p=p->next;

}

printf("\n");

}

//添加按字母、数字、其它字符分解单链表算法

void resolve(linklist\*head,linklist\*letter,linklist\*digit,linklist\*other){

linklist \*p;

p=head->next;

while(p!=NULL){

head->next=head->next->next;

if((p->data>='A'&&p->data<='Z')||(p->data>='a'&&p->data<='z')){

insert(letter,p);

}

else if(p->data>='0'&&p->data<='9'){

insert(digit,p);

}

else{

insert(other,p);

}

p=head->next;

}

}

**6、哈夫曼树（二叉树，功能不是很完善）**

#include <malloc.h>

#include <string.h>

// 哈夫曼树

typedef struct {

char info;

int weight;

int parent, lchild, rchild;

} HTNode, \*HuffmanTree;

typedef char\* \*HuffmanCode;

// 全局变量

HuffmanTree HT; // 哈夫曼树

HuffmanCode HC; // 哈夫曼树的结点

int n = 8; // 默认有八个字符及权重

// 选取最小权重点函数

void Select(HuffmanTree HT, int t, int \*s1, int \*s2) {

int i, temp1, temp2;

temp1 = temp2 = 1000;

// 在哈夫曼树HT中选择父亲不为0且权值最小的两结点，索引分别记为s1和s2

for (i = 1; i <= t; i++) {

if (HT[i].parent == 0 && (HT[i].weight<temp1 || HT[i].weight<temp2)) {

if (temp1<temp2) {

temp2 = HT[i].weight;

\*s1 = i;

} else {

temp1 = HT[i].weight;

\*s2 = i;

}

}

}

// s1为较小的序号，s2取较大的序号

if (\*s1 > \*s2) {

i = \*s1;

\*s1 = \*s2;

\*s2 = i;

}

}

// 哈夫曼树编码（n为字符串个数）

HuffmanTree HuffmanCoding(int \*w, int n, char \*info) {

// p为当前所指结点

HuffmanTree HT, p;

char \*cd;

// s1/s2记录权重最小的两点，f为父结点，c为子结点

int m, s1, s2, i, start, f, c;

if (n <= 1) {

return 0;

}

m = 2\*n-1;

HT = (HuffmanTree)malloc((m+1)\*sizeof(HTNode));

p = HT + 1;

// 初始化各点权重

for (i=1; i<= n; ++i, ++p, ++w) {

// 不用第0号单元

p->weight = \*w;

p->info = info[i-1];

p->parent = 0;

p->lchild = 0;

p->rchild = 0;

}

for (; i<= m; ++i, ++p) {

p->weight = 0;

p->parent = 0;

p->lchild = 0;

p->rchild = 0;

}

for (i = n+1; i <= m; ++i) {

// 调用select函数

Select(HT, i-1, &s1, &s2);

// 根据select返回的两个点，生成新节点

HT[s1].parent = i;

HT[s2].parent = i;

HT[i].lchild = s1;

HT[i].rchild = s2;

HT[i].weight = HT[s1].weight + HT[s2].weight;

}

// 逆向求哈夫曼编码

HC = (HuffmanCode)malloc((n+1)\*sizeof(char \*));

code = (char \*)malloc(n\*sizeof(char));

code[n-1] = '\0';

for (i = 1; i <= n; ++i) {

start = n - 1;

for (c = i, f = HT[i].parent; f != 0; c = f, f = HT[f].parent) {

// 从子叶到根逆向求编码

if (HT[f].lchild == c) {

code[--start] = '0';

} else {

code[--start] = '1';

}

}

HC[i] = (char \*)malloc((n-start)\*sizeof(char));

strcpy(HC[i], &cd[start]);

}

free(code);

return HT;

}

// 编码文本

void TextToCode() {

char str[50];

int i, j, p;

if (HT == NULL) {

printf("请先进行初始化！\n");

return ;

}

printf("请输入编码的文本（英文字母）：\n");

getchar();

gets(str);

printf("编码结果如下：\n");

p = strlen(str);

for (i = 1; i <= p; i++) {

for (j = 1; j <= n; j++) {

if (str[i-1] == HT[j].info) {

printf("%s", HC[j]);

break;

}

}

}

printf("\n");

}

// 译码

void CodeToText() {

int i = 1, j, key;

int i1, i2;

char str[100];

int a = 1;

int m = 2 \* n - 1;

if (HT == NULL) {

printf("请先进行初始化！\n");

return ;

}

printf("\n各字母对应的哈夫曼编码如下：\n");

/\* 以一定格式输出字母对应的编码 \*/

for (i1 = i; i1 <= 2; i1++) {

printf("┌———————┐ ┌———————┐ ┌———————┐ ┌———————┐\n");

for (i2 = 1; i2 <= 4; i2++) {

printf(" %c：%-11s ", HT[a].info, HC[a]);

a++;

}

printf("\n");

printf("└———————┘ └———————┘ └———————┘ └———————┘\n");

}

printf("\n请输入哈夫曼编码：\n");

scanf("%s", str);

j = strlen(str);

key = m;

printf("哈夫曼编码译码如下：\n");

while (i <= j) {

// 若左孩子不为0

while (HT[key].lchild != 0) {

if (str[i-1] == '0') {

key = HT[key].lchild;

i++; continue;

}

if (str[i-1] == '1') {

key = HT[key].rchild;

i++; continue;

}

}

printf("%c", HT[key].info);

key = m;

}

printf("\n");

}

// 菜单界面

void menu() {

printf("┌—————————————————————————┐\n");

printf("| 1 构造哈夫曼树 | \n");

printf("└—————————————————————————┘\n");

printf("┌—————————————————————————┐\n");

printf("| 2 哈夫曼编码 | \n");

printf("└—————————————————————————┘\n");

printf("┌—————————————————————————┐\n");

printf("| 3 哈夫曼译码 | \n");

printf("└—————————————————————————┘\n");

printf("┌—————————————————————————┐\n");

printf("| 4 退出 | \n");

printf("└—————————————————————————┘\n");

}

// 哈夫曼树的Demo主函数

int main() {

int flag, mark = 1;

// 默认的八个字母和权重（可修改）

int wei[8] = {1, 3, 5, 7, 8, 9, 10, 13};

char info[8] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'};

for (;mark;) {

menu();

scanf("%d", &flag);

switch(flag) {

case 1: HT = HuffmanCoding(wei, n, info); break;

case 2: TextToCode(); break;

case 3: CodeToText(); break;

case 4: j = 0;break;

default: printf("输入错误，重新输入！\n");

}

}

return 0;

}

**7、其他**









