

数据结构

实验报告（一）

线性表和链表

学号：3020205015

姓名：石云天

班级：智能 2 班

日期：2020.10.05

目 录

一、实验内容描述	3
(一) 完成顺序表的基本操作	3
(二) 实现有表头结点的链表的基本操作	3
(三) 实现两个链表的融合	3
二、实验步骤	4
三、程序设计	4
(一) 抽象数据类型 ADT	4
(二) 算法简述	6
四、调试分析	7
(一) 调试过程	7
(二) 时间复杂度	7
五、程序测试	8
六、代码附录	8

一、实验内容描述

本次实验主题为线性表和链表，包含三个方面：顺序表的实现、链表的实现和实现两个链表的融合。下面是详细的实验内容：

（一）完成顺序表的基本操作

- 初始化顺序表
- 销毁顺序表
- 添加元素
- 插入元素
- 删除元素
- 查找元素（按序号找）
- 查找元素（按值找）
- 倒序排列元素（使用原空间）
- 按序输出元素

（二）实现有表头结点的链表的基本操作

- 初始化链表
- 销毁链表
- 插入元素(头插法)
- 插入元素(尾插法)
- 插入元素
- 删除元素
- 查找元素（按序号找）
- 查找元素（按值找）
- 递增排序
- 倒序排列元素
- 按序输出元素

（三）实现两个链表的融合

基于上面的链表实现，设计一个算法 `void merge(LinkList& A, LinkList& B)`，读入 A 和 B 两个线性表，输入不需有序，输入后按元素值递增次序排列。编写程序将这两个单链表归并为一个按元素值递增次序排列的单链表，利用原来两个单

链表的结点存放归并后的单链表, 结果存在 A 线性表中。

二、实验步骤

(1) 根据上课所讲, 回顾线性表顺序存储和链式存储两种物理存储模式, 并总结所有操作, 构造抽象数据类型, 最后需要在链式存储的基础上完成两表合并的操作。本次实验帮助我们加深对线性表逻辑结构和物理结构的认识与理解, 提升代码编写能力, 同时多方面完善优化, 提升代码性能。

(2) 仔细阅读实验要求, 考虑线性表各个操作的主要算法。在算法设计层面, 思考顺序表和链式表的逻辑结构和表内元素的具体关系, 分别根据两种表的不同特点实现线性表的常用操作, 之后在此基础上实现链式表的合并操作。对算法进行深入分析, 我们可以发现: 线性表的顺序表示是指使用一组地址连续的存储单元依次存储线性表的数据元素, 以第一个存储单元的存储地址为数据元素的存储位置, 具有随机存取的特点。这与数组类型特点相似, 可以在此基础上加以实现。线性表的链式表示是指用一组任意的存储单元(既可以在物理地址上连续也可以非连续)存储线性表的数据元素, 因此每个数据元素除存储其本身的信息之外, 还需要存储下一个数据单元的存储位置, 利用指针、头结点的特性, 可以对其加以实现。

(3) 使用 Code::Blocks 环境, 基于 C++ 语言将算法用程序实现。

(4) 编译运行程序, 使用样例进行程序测试, 观察所编程序是否实现要求的功能。

(5) 考察算法的时间复杂度和空间复杂度, 评价算法的优劣, 进一步优化程序。

(6) 撰写实验报告, 进行实验反思。

三、程序设计

(一) 抽象数据类型 ADT

顺序表的抽象数据类型:

ADT SqList{

数据对象: $D = \{a_i | a_i \in \text{int}, i = 1, 2, 3, \dots\}$ (以整数为例)

数据关系: $R = \{ \langle a_{i-1}, a_i \rangle | a_i, a_{i-1} \in D \}$ (前后元素存在序偶关系, 存储时空位置相邻)

基本操作:

InitList(SqList &L)

操作结果: 初始化链表

DestroyList(Sqlist &L)

操作结果：销毁链表

AddList(Sqlist &L,int x)

操作结果：添加元素

ListInsert(Sqlist &L,int x,int i)

操作结果：插入元素

ListDelete(Sqlist &L,int i)

操作结果：删除元素

GetElem(Sqlist L,int i,int &x)

操作结果：按序查找

LocateElem(Sqlist L,int x)

操作结果：按值查找

ReverseList(Sqlist &L)

操作结果：元素倒序

PrintList(Sqlist L)

操作结果：打印顺序表

}

ADT LinkList{

数据对象： $D=\{a_i | a_i \in \text{int}, i = 1, 2, 3, \dots\}$ （以整数为例）

数据关系： $R=\{\langle a_{i-1}, a_i \rangle | a_i, a_{i-1} \in D\}$ （前后元素存在序偶关系，存储空间位置不必相邻）

基本操作：

InitLinkList(LinkList &L)

操作结果：初始化链表

DestroyLinkList(LinkList &L)

操作结果：销毁链表

HeadCreateLinkList(LinkList &L,int x)

操作结果：头插法建立链表

LastCreateLinklist(LinkList &L,int x)

操作结果：尾插法建立链表

```
LinkListInsert(LinkList &L,int i,int x)
```

操作结果：插入元素

```
LinkListDelete(LinkList &L,int i)
```

操作结果：删除元素

```
GetElem_LinkList(LinkList L,int i,int &x)
```

操作结果：按序查找

```
LocateElem_LinkList(LinkList L,int x)
```

操作结果：按值查找

```
SortLinkList(LinkList &L)
```

操作结果：递增排序

```
ReverseLinkList(LinkList &L)
```

操作结果：元素倒序

```
PrintLinkList(LinkList L)
```

操作结果：按序打印链表

```
}
```

（二）算法简述

因为顺序表和链表的基本操作的实现都较为基础的，此处不再赘述。

下面简述实现两个链表的融和并递减输出的算法：

1. 首先输入链表 A 和链表 B 的长度和数据，建立两个链表并完成初始化。

2. 调用 SortLinkList(&L) 和 ReverseLinkList(&L) 两个预设函数实现链表 A、B 内部元素的降序排列。

3. 创建两个指针分别指向 A、B 中的数据元素，从链表 A 和 B 的第一个元素开始比较，若 B 的元素大于 A 的元素，则将 B 的这一节点插入 A 的对应节点之前，并将 B 的指针后移；若 B 的元素等于 A 的元素，就将 A 和 B 的指针都后移；如果 A 的元素大于 B 的元素，就将 A 的指针后移。

4. 循环进行 3，直至某一链表为空，此时如果 B 表还有剩余元素，就将其全部链接至 A 表末尾。

5. 调用函数按序输出链表 A。

四、调试分析

（一）调试过程

在代码编写完成后，使用 Code::Blocks 编译并运行。编译后程序报错，经过修改，编译通过后，属于测试样例一直无法取得正确的结果，于是将 Merge 函数中各个部分进行单独测试，发现链表排序和链表翻转两部分都没有问题，最终将问题锁定在融合去重这一部分，再对该部分程序进行进一步的检查。发现了以下问题：

（1）首先是在融合过程中没有额外创建新的结点，而是将 B 中结点与 A 中结点直接按顺序相连，由于最终输出的是 A 链表，故会导致输出异常，无法得到两表融合后的结果，于是为 A 链表创造新结点用于容纳 B 中的数据，最终结果正确。

（2）其次还出现了内存非法访问的情况，对各链表内存地址进行进一步检查，最终结果正确。

在 Merge 算法以及链表排序部分编写之初，均使用了选择排序的方法，其时间复杂度为 $O(n^2)$ ，后改用时间复杂度为 $O(n\log n)$ 的 sort 函数进行排序，降低其时间复杂度，提升算法性能。

（二）时间复杂度

经过对融合算法的时间复杂度分析，设 A 表数据规模为 m ，B 表数据规模为 n 可以得到 Merge 算法的时间复杂度为 $O((m+n)^2)$ ，在实现过程中，使用了 sort 函数进行数组排序，其时间复杂度为 $O(n\log n)$ 。

五、程序测试

在 Merge 算法代码编写完成后，对其进行了多次测试，都满足要求。下面是其中一次测试样例，运行结果如图 1 所示。

测试样例：

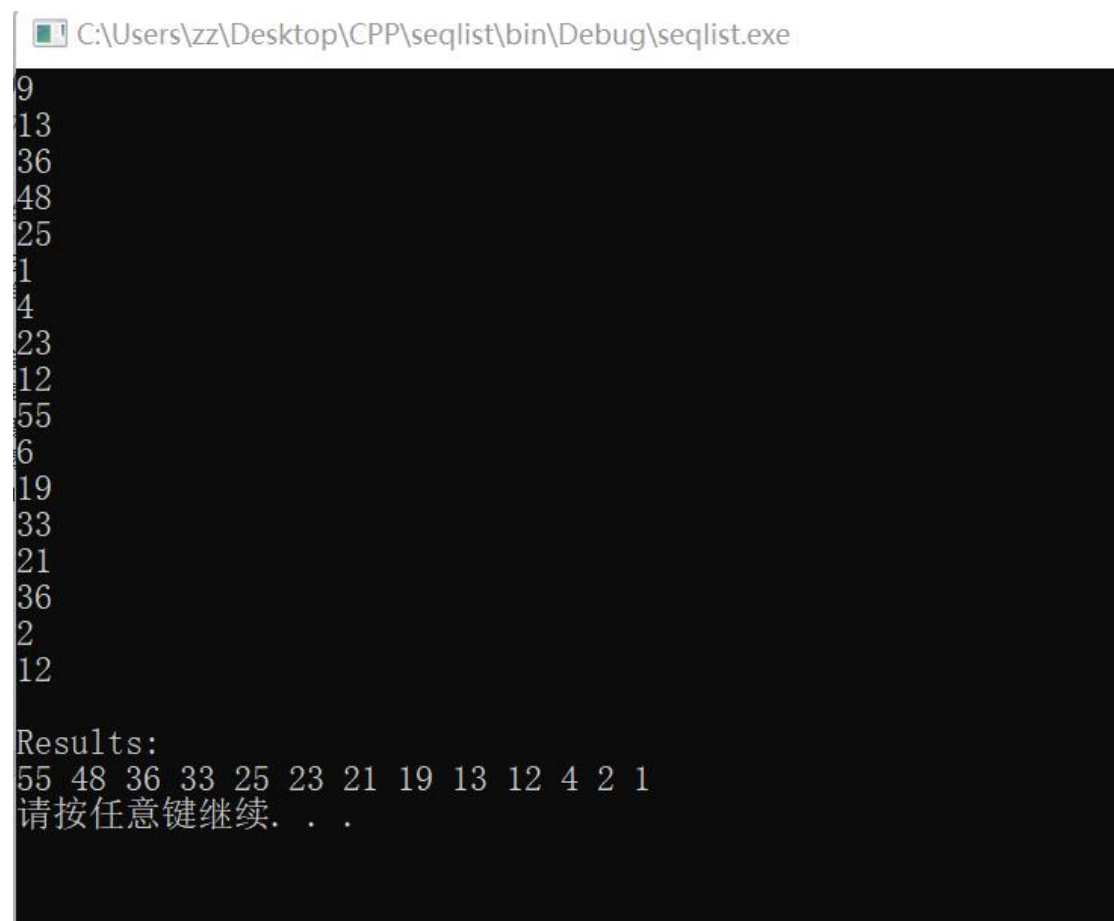
链表 A 的长度：9

链表 A 的元素：13 36 48 25 1 4 23 12 55

链表 B 的长度：6

链表 B 的元素：19 33 21 36 2 12

输出结果应为：55 48 36 33 25 23 21 19 13 12 4 2 1



```
C:\Users\zz\Desktop\CPP\seqlist\bin\Debug\seqlist.exe
9
13
36
48
25
1
4
23
12
55
6
19
33
21
36
2
12
Results:
55 48 36 33 25 23 21 19 13 12 4 2 1
请按任意键继续. . .
```

图 1：程序运行结果

六、代码附录

```
#include <iostream>
#include<algorithm>
```



```
using namespace std;

//顺序表基本操作

//顺序表存储结构
typedef struct{
    int * element;
    int length;//顺序表长
}SqList;

//初始化顺序表
int InitList(SqList &L){
    L.element=new int[1000];//声明最大容量

    if(!L.element) return 0;//分配空间失败

    L.length=0;
    return 1;
}

//销毁顺序表
void DestroyList(SqList &L){
    L.length=0;
    delete L.element;
}

//添加元素
int AddList(SqList &L,int x){
    if(L.length>1000) return 0;//元素个数达到上限

    L.element[L.length]=x;
    L.length++;
    return 1;
}

//插入元素（第 i 个元素前）
int ListInsert(SqList &L,int x,int i){
```

```

    if(i<1 or i>L.length+1 or L.length>=1000) return 0;//顺序表满或插入位置非法
    for(int j=L.length-1;j>=i-1;j--){
        L.element[j+1]=L.element[j];
    }
    L.element[i-1]=x;
    L.length++;
    return 1;
}

```

//删除元素

```

int ListDelete(SqList &L,int i){
    if(i<1 or i>L.length or L.length==0) return 0;//顺序表空或者删除位置非法
    for(int j=i-1;j<L.length-1;j++){
        L.element[j]=L.element[j+1];
    }
    L.length--;
    return 1;
}

```

//按序号查找元素

```

int GetElem(SqList L,int i,int &x){
    if(i<1 or i>L.length) return 0;//非法查找
    x=L.element[i-1];
    return 1;
}

```

//按值查找元素

```

int LocateElem(SqList L,int x){
    int i=0;
    while(i<L.length and L.element[i]!=x){
        i++;
    }
    if(i==L.length) return -1;//顺序表中无此元素

    else return i;//若找到，返回对应的数组下标
}

```

//倒序排列元素

```
void ReverseList(SqList &L){
    for(int i=0;i<(L.length-1)/2;i++){
        int a=L.element[i];
        L.element[i]=L.element[L.length-1-i];
        L.element[L.length-1-i]=a;
    }
}
```

//按序输出元素

```
void PrintList(SqList L){
    if(!L.length) cout<<"Empty List!"<<endl;//顺序表空
    else{
        for(int i=0;i<L.length;i++){
            cout<<L.element[i]<<endl;
        }
    }
}
```

//链表基本操作

//链表存储结构

```
typedef struct Lnode{
    int data;
    struct Lnode* next;
}Lnode,*LinkList;
```

//初始化链表

```
void InitLinkList(LinkList &L){
    L=new Lnode;//创建头结点
    L->next=NULL;
}
```

//销毁链表

```
void DestroyLinkList(LinkList &L){
    Lnode *p=L->next;
    Lnode *q=p;//利用双指针完成销毁操作
```

```

        while(p){
            p=p->next;
            delete q;
            q=p;
        }
    }
}

```

//插入元素（头插法）

```

void HeadCreateLinkList(LinkList &L,int x){
    Lnode* m=new Lnode;
    m->data=x;
    m->next=L->next;
    L->next=m;
}

```

//插入元素（尾插法）

```

void LastCreateLinklist(LinkList &L,int x){
    Lnode* m=new Lnode;
    m->data=x;
    m->next=NULL;
    Lnode *p=L;
    while(p->next){
        p=p->next;
    }//根据链表性质找到尾结点

    p->next=m;//链接至新结点
}

```

//插入元素

```

void LinkListInsert(LinkList &L,int i,int x){
    Lnode* m=new Lnode;
    m->data=x;
    int a=1;
    Lnode *p=L->next;
    while(p!=0 && a<i){
        p=p->next;
        a++;
    }

    m->next=p->next;//找到所需位置完成链接
}

```

```
    p->next=m;
}
```

//删除元素

```
void LinkListDelete(LinkList &L,int i){
    int a=1;
    Lnode *p=L->next;
    while(p!=0 && a<i){
        p=p->next;
        a++;
    }
    int b=1;

    Lnode *q=L->next;//利用双指针保证删除后链接顺序正确

    while(q!=0 && b<i-1){
        q=q->next;
        b++;
    }
    q->next=p->next;
    delete p;
}
```

//查找元素（按序号找）

```
int GetElem_LinkList(LinkList L,int i,int &x){
    if(i<1) return -1;//非法查找

    int a=1;
    Lnode *p=L->next;
    while(p!=0 && a<i){
        p=p->next;
        a++;
    }
    if(!p) return -1;
    else x=p->data;
}
```

//查找元素（按值找）

```
int LocateElem_LinkList(LinkList L,int x){
    Lnode *p=L->next;
    int a=1;
```

```

while(p!=0 and p->data!=x){
    p=p->next;
    a++;
}

if(!p) return -1;//链表中无此元素

return a;//链表中第 k 个元素即为所找
}

//获取元素个数

int GetNum(LinkList L){
    Lnode* p=L->next;
    int a=0;
    while(p!=0){
        p=p->next;
        a++;
    }
    return a;
}

//递增排序

int SortLinkList(LinkList &L){
    //先取出链表中所有元素并置于数组中

    int a[GetNum(L)];
    Lnode* p=L->next;
    int b=0;
    while(p!=0){
        a[b]=p->data;
        b++;
        p=p->next;
    }

    sort(a,a+GetNum(L));//利用 sort 函数进行数组排序

    Lnode* q=L->next;
    int c=0;
    while(q!=0){
        q->data=a[c];
        q=q->next;
        c++;
    }//再将排序后的数组元素一一存入链表中
}

```

```
}
```

//倒序排列元素

```
void ReverseLinkList(LinkList &L){
    Lnode* p=L->next;
    Lnode* q=p;
    L->next=NULL;
    while(p!=0){
        p=p->next;
        q->next=L->next;
        L->next=q;
        q=p;
    }//利用双指针改变链表指针方向，完成倒序操作
}
```

//按序输出元素

```
void PrintLinkList(LinkList L){
    Lnode* p=L->next;

    if(!p) cout<<"Empty LinkList!"<<endl;//链表空

    else{
        while(p!=0){
            cout<<p->data<<" ";
            p=p->next;
        }
    }
    cout<<endl;
}
```

//实现两个链表的融合

```
void Merge(LinkList &A,LinkList &B){

    InitLinkList(A);
    InitLinkList(B);//首先利用预设函数对两个链表进行初始化

    int A_length;
    cin>>A_length;
    int a=0;
    for(int i=0;i<A_length;i++){
        cin>>a;
```

```

        HeadCreateLinkList(A,a);
    }
    int B_length;
    cin>>B_length;
    int b=0;
    for(int i=0;i<B_length;i++){
        cin>>b;

        HeadCreateLinkList(B,b);//使用头插法建立两个链表
    }

    SortLinkList(A);
    SortLinkList(B);
    ReverseLinkList(A);

    ReverseLinkList(B);//利用递增排序后翻转实现两个链表的递减排序

    Lnode *c=A;
    Lnode *pa=A->next;
    Lnode *pb=B->next;
    while(pa!=0 && pb!=0){
        if((pa->data)<(pb->data)){
            Lnode* d=new Lnode;
            d->data=pb->data;
            d->next=pa;
            c->next=d;
            c=d;
            pb=pb->next;
        }
        else if((pa->data)==(pb->data)){
            c=c->next;
            pa=pa->next;
            pb=pb->next;
        }
        else{
            c=c->next;
            pa=pa->next;
        }
    }
    if(pa==0){
        c->next=pb;
    }
    //实现两表合并并完成去重

    cout<<endl;
    cout<<"Results:"<<endl;

```



```
    PrintLinkList(A);//按序输出 A 表
}

int main(){
    LinkList A;
    LinkList B;
    Merge(A,B);
    system("pause");
}
```