南京信息工程大学 数据结构 I 实验(实习)报告

实验(实习)名称 **线性表** 日期 2024.10. 得分 指导教师

学院 计算机 专业 计科

一、实验目的

1、理解线性表的逻辑结构和两种存储结构；
2、掌握线性表的顺序存储表示及实现；
3、掌握线性表的链接存储表示及实现；
4、能够应用线性表解决实际问题，并能分析其时间复杂度。

二、实验内容与步骤

用 C 语言编写程序（**建议定义函数时，形参尽量使用引用传递方式，即教材里形参前使用“&”符号的传递方式**），已知线性表 La={2,4,6,8,10} ，Lb={1,2,3,4,5,6,8}

1、用 SqList 类型（SqList 类型定义见教材 P22 页）自行建立两个有序的顺序表，然后将 La 与 Lb 合并成一个新的有序顺序表 Lc={ 1,2,2,3,4,4,5,6,6,8,8,10 }（即实现教材算法 2.7）。

（1）写出合并过程的算法思想：

两个顺序表 La 和 Lb 均为有序表，分别使用两个指针分别指向 La 和 Lb 中的元素，从头开始比较两个顺序表中的元素，将较小的元素依次放入新顺序表 Lc 中；如果一个顺序表中的所有元素已被放入 Lc，则将另一个顺序表中剩余的元素直接放入 Lc，最终得到的 Lc 是一个有序的顺序表，包含了 La 和 Lb 的所有元素。

（2）完整的程序代码（文本形式，禁止截图）：

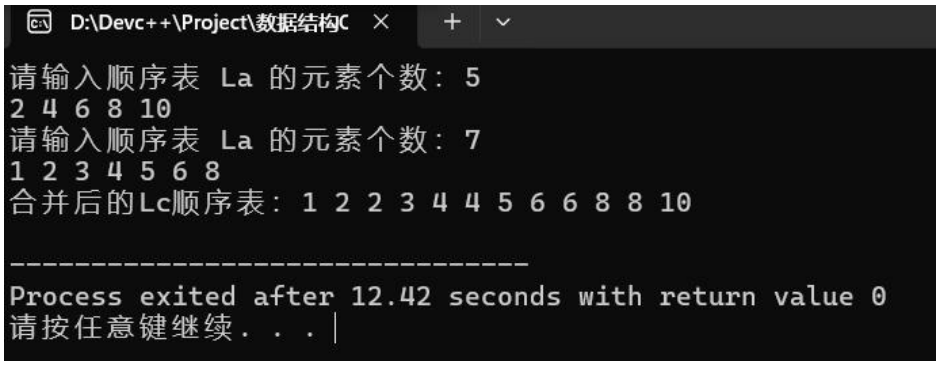
```
#include<iostream>
#include<stdlib.h>
using namespace std;
//常量定义
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
#define OK 1
#define ERROR 0
#define OVERFLOW -2
#define True 1
#define False 0
typedef int Status;
typedef int ElemType;
//顺序表类型定义
typedef struct{
    ElemType *elem;
    int length;
    int listsize;
}SqList;

//顺序表各操作声明
```

```
Status InitList_Sq(SqList &L);
Status DetroyList_Sq(SqList &L);
Status ListInsert_Sq(SqList &L,int i,ElemType e);
void MergeList_Sq(SqList La,SqList Lb,SqList &Lc);
void PrintList_Sq(SqList L);
//初始化
Status InitList_Sq(SqList &L)
{
   L.elem=(ElemType *)malloc(LIST_INIT_SIZE*sizeof(ElemType));
   if(!L.elem) exit(OVERFLOW);
   L.length=0;
   L.listsize=LIST_INIT_SIZE;
   return OK;
}
//销毁
Status DetroyList_Sq(SqList &L)
{
   if(L.elem) free(L.elem);
   return OK;
}
//插入
Status ListInsert_Sq(SqList &L, int i, ElemType e) {
      if (i < 1 || i > L.length + 1)
   {
         return ERROR;
      }
      if (L.length >= L.listsize)
   {
         ElemType *newbase = (ElemType *)realloc(L.elem, (L.listsize + LISTINCREMENT)
* sizeof(ElemType));
         if (!newbase)
        {
            exit(OVERFLOW);
         }
         L.elem = newbase;
         L.listsize += LISTINCREMENT;
      }
      for (int j = L.length - 1; j >= i - 1; j--) {
         L.elem[j + 1] = L.elem[j];
      }
      L.elem[i - 1] = e;
      L.length++;
      return OK;
   }
```

```
//合并
void MergeList_Sq(SqList La,SqList Lb,SqList &Lc)
{
   ElemType *pa, *pb, *pc;
      ElemType *pa_last, *pb_last;
   pa=La.elem;
   pb=Lb.elem;
   Lc.listsize=La.length+Lb.length;
   pc=Lc.elem=(ElemType *)malloc(Lc.listsize*sizeof(ElemType));
   if(!Lc.elem) exit(OVERFLOW);
   pa_last=La.elem+La.length-1;
   pb_last=Lb.elem+Lb.length-1;
   while(pa<=pa_last&&pb<=pb_last)
   {
         if(*pa<=*pb) *pc++=*pa++;
         else *pc++=*pb++;
   }
   while(pa<=pa_last) *pc++=*pa++;
   while(pb<=pb_last) *pc++=*pb++;
   Lc.length=La.length+Lb.length;
}
//输出表中元素
void PrintList_Sq(SqList L)
{
      for(int i = 0; i < L.length; i++)
   {
            cout << L.elem[i] << " ";
      }
      cout << endl;
}
int main()
{
   SqList La, Lb, Lc;
   InitList_Sq(La);
   InitList_Sq(Lb);
   InitList_Sq(Lc);
      int i, num, na, nb;
      cout << "请输入顺序表 La 的元素个数：";
      cin >> na;
      for (i = 0; i < na; i++)
      {
            cin >> num;
            ListInsert_Sq(La, i + 1, (ElemType)num);
      }
```

```
    cout << "请输入顺序表 La 的元素个数：";
    cin >> nb;
    for (i = 0; i < nb; i++)
    {
        cin >> num;
        ListInsert_Sq(Lb, i + 1, (ElemType)num);
    }
    MergeList_Sq(La, Lb, Lc);
    cout<<"合并后的 Lc 顺序表：";
    PrintList_Sq(Lc);
    DetroyList_Sq(La);
    DetroyList_Sq(Lb);
    DetroyList_Sq(Lc);
    return 0;
}
```

（3）运行结果（截图）：



以本题为例，在编写代码时注意：

（1）教材 SqList 类型定义中，线性表元素的 ElemType 类型是抽象的不确定的，需要根据实际问题改为某种具体的数据类型

（2）教材很多算法中出现的 ERROR、OK、OVERFLOW 等均是事先定义好的常量，见教材 P10 页

（3）完整程序的大体框架应为：

// 首先使用 define 定义函数中使用的所有常量，可参考教材 P10

// 然后定义本题所使用的数据类型 SqList，见教材 P22

// 接着定义根据需要在 SqList 类型上进行的一些操作，如教材中的 InitList_Sq(SqList &L)、ListInsert_Sq(SqList &L,int i,int e)、MergeList_Sq(SqList La,SqList Lb,SqList &Lc)等，以及其它的一些函数，如输出线性表各元素值等。

// 最后实现主函数 main()，通过调用上述各函数来解决实际问题

2、用 LinkList 类型（LinkList 类型定义见教材 P28 页）自行建立两个有序的单链表，其中使用头插法创建链表 La，使用尾插法创建链表 Lb。

（1）将 La 与 Lb 合并，得到有序链表 Lc（即实现教材算法 2.12）。

写出合并过程的算法思想：创建一个新的链表 Lc，使用两个指针 p 和 q，分别指向 La 和 Lb 的第一个节点。比较 p 和 q 所指向节点的值，将较小的节点插入到 Lc 中，并移动相应的指针。重复第这一操作，直到其中一个链表遍历完。将剩余链表的节点直接连接到 Lc 的末尾，最后返回合并后的链表 Lc。

（2）分别分析有序的顺序表合并和有序的单链表合并的算法时间复杂度。

有序的顺序表合并是基于数组存储的，两个顺序表 La 和 Lb 中的元素从头到尾逐个比较和插入，整个过程的时间复杂度为 O(m + n)，其中 m 和 n 分别是两个顺序表的长度。有序的单链表的合并操作本质上也是两个链表的逐个比较与插入操作。每个节点最多访问一次，因此单链表合并的时间复杂度同样为 O(m + n)，其中 m 和 n 是两个链表的长度。

（3）删除 Lc 中多余的重复元素，使得所有元素只保留一个。

写出算法思想：从第一个节点开始，逐个检查相邻节点。定义一个 p 为第一个有效节点，如果当前节点的数据与下一个节点的数据相同，将 p->next 指向 p->next->next，执行删除操作，然后释放被删除的节点，继续遍历；如果没有重复，p 指针继续向后移动，直到遍历完整个链表。

（4）在 Lc 存储空间基础上将其倒置，并输出内容。

写出算法思想：通过三个指针 prev、current 和 next，prev 初始指向 NULL，表示反转后链表的末尾，current 指向链表的第一个节点，next 用于保存 current 的下一个节点。在遍历链表的过程中，将每个节点的 next 指针指向它的前一个节点 prev，实现指针反转。然后依次将 prev 移动到当前节点，current 移动到下一个节点，直到 current 为 NULL 时遍历结束，此时 prev 指向反转后的链表头节点。最后，将链表的头指针更新为 prev，完成链表的反转。

（5）最终完整的程序代码（文本形式，禁止截图）：

```
#include<iostream>
#include<stdlib.h>
#include<malloc.h>
#include<time.h>

using namespace std;

//常量定义
#define OK 1
#define ERROR 0
#define OVERFLOW -2

typedef int Status;

// 表节点数据类型定义
typedef int ElemType;

// 定义链表节点结构
typedef struct LNode
{
    ElemType data;
    struct LNode *next;
} LNode, *LinkList;
// 链表各操作声明
Status InitList_L(LinkList &L);
Status ListInsert_L_1(LinkList &L, int i, ElemType e);
```

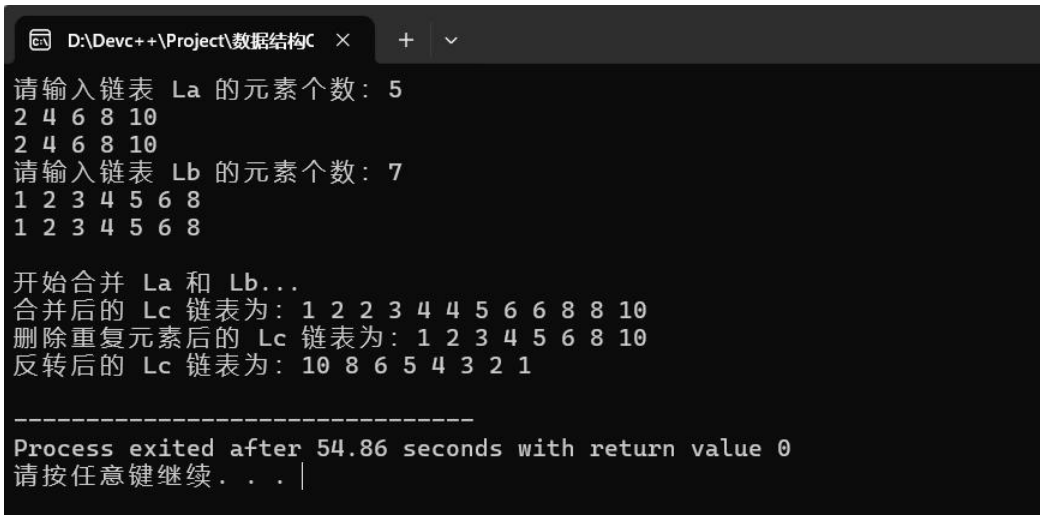
```
Status ListInsert_L_2(LinkList &L, ElemType e);
Status ListDelete_L(LinkList &L, int i, ElemType &e);
void MergeList_L(LinkList La, LinkList Lb, LinkList &Lc);
void RemoveList_L(LinkList L);
void ReverseList(LinkList L);
void PrintList_L(LinkList L);
// 初始化链表
Status InitList_L(LinkList &L)
{
    L = (LinkList)malloc(sizeof(LNode));  // 分配头节点 L
    if (!L) exit(OVERFLOW);
    L->next = NULL;  // 初始时链表为空
    return OK;
}
// 头插法
Status ListInsert_L_1(LinkList &L, int i, ElemType e)
{
    LinkList p = L;
    int j = 0;
    while (p && j < i - 1)
    {
        p = p->next;
        j++;
    }
    if (!p || j > i - 1) return ERROR;
    LinkList s = (LinkList)malloc(sizeof(LNode));
    if (!s) exit(OVERFLOW);
    s->data = e;
    s->next = p->next;
    p->next = s;
    return OK;
}
// 尾插法
Status ListInsert_L_2(LinkList &L, ElemType e)
{
    LinkList s = (LinkList)malloc(sizeof(LNode));  // 为新节点分配内存
    if (!s) exit(OVERFLOW);  // 如果内存分配失败，退出程序
    s->data = e;  // 将元素值赋给新节点
    s->next = NULL;  // 新节点的下一个指向 NULL，因为它是最后一个节点

    LinkList p = L;  // 从头节点开始遍历
    while (p->next) {  // 找到链表的最后一个节点
        p = p->next;
    }
```

```
    p->next = s;    // 将最后一个节点的 next 指向新插入的节点
    return OK;
}

// 删除链表中的元素
Status ListDelete_L(LinkList &L, int i, ElemType &e) {
    LinkList p = L;
    int j = 0;
    while (p->next && j < i - 1)
    {
        p = p->next;
        j++;
    }
    if (!(p->next) || j > i - 1) return ERROR;
    LinkList q = p->next;
    e = q->data;
    p->next = q->next;
    free(q);
    return OK;
}
// 合并两个有序链表
void MergeList_L(LinkList La, LinkList Lb, LinkList &Lc) {
    LinkList pa = La->next;
    LinkList pb = Lb->next;
    Lc = La;
    LinkList pc = Lc;
    while (pa && pb)
    {
        if (pa->data <= pb->data)
        {
            pc->next = pa;
            pc = pa;
            pa = pa->next;
        }
        else
        {
            pc->next = pb;
            pc = pb;
            pb = pb->next;
        }
    }
    pc->next = pa ? pa : pb;
    free(Lb);
}
```

```
//删除重复元素
void RemoveList_L(LinkList L){
	LinkList p = L->next;
	while(p && p->next){
		if(p->data == p->next->data)
		{
			LinkList temp = p->next;
			p->next = temp->next;
			free(temp);
		}
		else
			p = p->next;
	}
}
void ReverseList(LinkList L) {
	LNode *prev = NULL, *current = L->next, *next;
	while (current) {
		next = current->next;
		current->next = prev;
		prev = current;
		current = next;
	}
	L->next = prev;
}

// 打印链表
void PrintList_L(LinkList L) {
	LinkList p = L->next;
	if (!p) {
		cout << "链表为空" << endl;
		return;
	}
	while (p) {
		cout << p->data << " ";
		p = p->next;
	}
	cout << endl;
}
// 主函数
int main() {
	LinkList La, Lb, Lc;
	int i, num, na, nb;
	InitList_L(La);
	InitList_L(Lb);
```

```
    cout << "请输入链表 La 的元素个数：" ;
    cin >> na ;
    for (i = 1; i <= na; i++) {
        cin >> num;
        ListInsert_L_1(La, i, num);
    }
    PrintList_L(La);
    cout << "请输入链表 Lb 的元素个数：";
    cin >> nb;
    for (i = 1; i <= nb; i++) {
        cin >> num;
        ListInsert_L_2(Lb, num);
    }
    PrintList_L(Lb);
    cout << "\n 开始合并 La 和 Lb..." << endl;
    MergeList_L(La, Lb, Lc);
    cout << "合并后的 Lc 链表为：";
    PrintList_L(Lc);
    RemoveList_L(Lc);
    cout << "删除重复元素后的 Lc 链表为：";
    PrintList_L(Lc);
    ReverseList(Lc);
    cout << "反转后的 Lc 链表为：";
    PrintList_L(Lc);
    return 0;
}
```

（6）运行结果（截图）：



三、实验心得（必写）

通过这次实验，我主要学习了顺序表和链式表的基本操作及其应用。在顺序表的实验中，我学习了顺序表合并。链式表实验让我深入学习了链表如何通过指针操作合并链表。在处理链表中的去重和反转操作时，我理解了如何通过调整指针来操作链表的结构。特别是在链表

反转过程中，指针的灵活应用着实有难度，使我认识到链式表的优势和指针操作的复杂性。整体上，我收获颇多。