实验报告

# 一、设计人员相关信息

## 1.设计者

姓名：陈怡冰

学号：24012022

班号：24计算机科学与技术1班

## 2.设计日期：

2025年9月15日-2025年9月21日

# 二、程序设计相关信息

## 1.实验题及问题描述。

实验  单链表

结点数据定义如下：

**typedef   int  ElemType;**

**typedef   struct LNode        //链表结点**

**{  ElemType   data;       //结点数据域**

**struct LNode \*next;           //结点链域**

**} LinkList;**

**注意：本次作业，main（）中可以调用linklist.cpp中的初始化、创建和释放函数;**

**自己定义的子函数中必须写基本语句，不能调用linklist.cpp中的函数，可以调用自己写的子函数。**

参考单链表的基本运算，设计有序单链表的基本运算，主要功能函数如下，

1）初始化有序单链表L，L为空表。

2）main（）中存放一组有序数建立有序单链表，或产生一组随机数建立单链表并对单链表排序。

3）求有序单链表L的长度。

4）有序单链表L中取出第i个元素。

5）定位函数，确定有序单链表L值为e元素的位置。

6）在有序单链表L插入一个值为e的元素，使其仍有序。

7）假设值为e的元素有多个，删除有序单链表L中所有值为e的元素。

8）假设值为e的元素仅一个，将有序单链表L中值为e元素修改值为e1，使其仍有序。

9）将有序单链表L中属于区间[low,high]的数据取出存入新的有序单链表L1，原链表L不修改。

10）有序单链表LA和有序单链表LB，求出两个集合的并集，并存入LC中。单个集合中不能有重复元素，但LA和LB中可能有重复值。

11）有序单链表LA和有序单链表LB，求出两个集合的交集，并存入LC中。单个集合中不能有重复元素，但LA和LB中可能有重复值。

12）有序单链表LA和有序单链表LB，求出LA与LB的差集，并存入LC中。单个集合中不能有重复元素，但LA和LB中可能有重复值。

## 2.实验目的。

（1）掌握单链表的结构定义与动态内存管理机制；  
（2）熟练掌握单链表的创建、插入、删除、查找等基本操作；  
（3）理解有序单链表的特点，实现有序插入与删除；  
（4）掌握集合运算（并、交、差）在链表上的实现方法；  
（5）提升程序调试能力与对指针操作的理解。

## 3.数据结构设计。

元素（Elemtype）定义

1. **typedef** **int** ElemType;  // 假设元素为整型

顺序表(SeqList)的定义

1. **typedef** **struct** LNode {
2. ElemType data;          // 数据域
3. **struct** LNode \*next;     // 指针域，指向下一个节点
4. } LinkNode;

## 4.程序结构(程序中的函数调用关系图)。

main()

├── InitList() // 初始化链表

├── InsertElem() // 有序插入（构建有序表）

├── LengthList() // 求长度

├── GetElem() // 取第i个元素

├── LocateElem() // 查找元素位置

├── DeleteElemAll() // 删除所有值为e的元素

├── ModifyElem() // 修改元素值

│ └── DeleteElemAll() + InsertElem()

├── RangeExtract() // 提取区间元素

│ └── InsertElem()

├── UnionList() // 并集

│ └── InsertElem()

├── IntersectList() // 交集

│ └── InsertElem()

├── DiffList() // 差集

│ └── InsertElem()

└── DisList() // 打印链表

## 5.主要的算法描述。

（1）有序插入 InsertElem(L, e)

从头开始遍历，找到第一个 data >= e 的前驱节点 p；

若 p->next 存在且值等于 e，则不插入（避免重复）；

否则创建新节点 s，插入 p 之后；

保持链表有序性。

（2）求长度 LengthList(L)

从 L->next 开始遍历，计数器 n 自增；

直到 p == NULL，返回 n。

（3）取第 i 个元素 GetElem(L, i, e)

从首元节点开始，用指针 p 和计数器 j 遍历；

当 j == i 时返回 p->data；

若 p == NULL 则说明越界，返回失败。

（4）定位元素 LocateElem(L, e)

遍历链表，比较 p->data 与 e；

找到相等则返回位置（从1开始）；

若 p->data > e 或遍历结束仍未找到，则返回0。

（5）删除所有值为 e 的元素 DeleteElemAll(L, e)

使用 p 指向当前检查节点的前驱；

若 p->next->data == e，则删除该节点并释放内存；

继续向后扫描，直到 data > e 或链表结束。

（6）修改元素值 ModifyElem(L, e, e1)

先调用 DeleteElemAll(L, e) 删除所有 e；

再调用 InsertElem(L, e1) 插入新值；

自动保持有序。

（7）区间提取 RangeExtract(L, L1, low, high)

初始化新链表 L1；

遍历原链表 L，若元素在 [low, high] 范围内，则调用 InsertElem(L1, data) 插入；

保证结果有序。

（8）并集 UnionList(LA, LB, LC)

使用归并思想：

比较 LA 和 LB 当前元素；

较小者插入 LC，相等则插入一次；

遍历完成后将剩余部分插入；

结果自动去重且有序。

（9）交集 IntersectList(LA, LB, LC)

同时遍历 LA 和 LB；

只有当 pa->data == pb->data 时才插入 LC；

其他情况移动较小者的指针。

（10）差集 DiffList(LA, LB, LC)

遍历 LA 和 LB：

若 LA 元素小于 LB，则属于差集，插入 LC；

相等则跳过；

LA 元素大于 LB，则 LB 指针后移；

最后将 LA 剩余元素全部插入。

## 6.实验源程序。

1. #include<stdio.h>
2. #include<malloc.h>
3. #include<stdlib.h>
4. **typedef** **int** ElemType;
5. **typedef** **struct** LNode {
6. ElemType data;
7. **struct** LNode \*next;     //指向后继结点
8. } LinkNode;                 //声明单链表结点类型


12. // 1）初始化有序单链表L,L为空表
13. **void** InitList(LinkNode \*&L) {
14. L = (LinkNode \*)malloc(**sizeof**(LinkNode));   //创建头结点
15. L->next = NULL;
16. }
17. **void** DestroyList(LinkNode \*&L) {
18. LinkNode \*pre = L,\*p = pre->next;
19. **while** (p != NULL) {
20. free(pre);
21. pre = p;
22. p = pre->next;
23. }
24. free(pre);
25. }
27. **int** LengthList(LinkNode \*&L) {
28. LinkNode \*p = L;
29. **int** n = 0;
30. **while** (p->next != NULL) {
31. n++;
32. p = p->next;
33. }
34. **return** n;
35. }
37. **void** DisList(LinkNode \*&L) {        //打印单链表
38. LinkNode \*p = L->next;
39. printf("List: ");
40. **while** (p != NULL) {
41. printf("%d ", p->data);
42. p = p->next;
43. }
44. printf("\n");
45. }

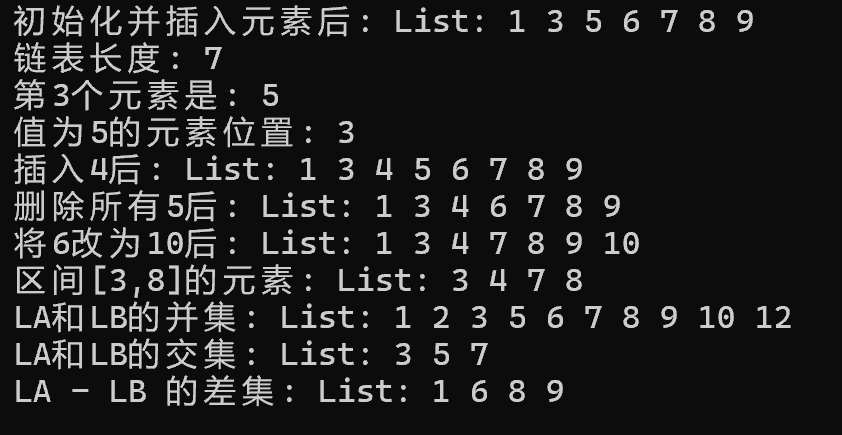
48. // 4）有序单链表L中取出第i个元素（i从1开始）
49. **int** GetElem(LinkNode \*L, **int** i, ElemType &e) {
50. **int** j = 1;
51. LinkNode \*p = L->next;
52. **while** (p != NULL && j < i) {
53. j++;
54. p = p->next;
55. }
56. **if** (p == NULL) **return** 0; // 不存在第i个结点
57. e = p->data;
58. **return** 1;
59. }
61. // 5）定位函数，确定有序单链表L值为e元素的位置（返回位置，从1开始）
62. **int** LocateElem(LinkNode \*L, ElemType e) {
63. **int** i = 1;
64. LinkNode \*p = L->next;
65. **while** (p != NULL && p->data < e) {
66. i++;
67. p = p->next;
68. }
69. **if** (p != NULL && p->data == e)
70. **return** i;
71. **else**
72. **return** 0; // 未找到
73. }
75. // 6）在有序单链表L插入一个值为e的元素，使其仍有序
76. **void** InsertElem(LinkNode \*L, ElemType e) {
77. LinkNode \*p = L;
78. **while** (p->next != NULL && p->next->data < e) {
79. p = p->next;
80. }
81. // 如果值已存在，可选择不插入（保持集合无重复），这里按集合处理，不插入重复值
82. **if** (p->next != NULL && p->next->data == e)
83. **return**;
85. LinkNode \*s = (LinkNode \*)malloc(**sizeof**(LinkNode));
86. s->data = e;
87. s->next = p->next;
88. p->next = s;
89. }
91. // 7）删除有序单链表L中所有值为e的元素
92. **void** DeleteElemAll(LinkNode \*L, ElemType e) {
93. LinkNode \*p = L;
94. **while** (p->next != NULL) {
95. **if** (p->next->data == e) {
96. LinkNode \*q = p->next;
97. p->next = q->next;
98. free(q);
99. } **else** **if** (p->next->data > e) {
100. **break**; // 后面不可能再有e
101. } **else** {
102. p = p->next;
103. }
104. }
105. }
107. // 8）将有序单链表L中值为e的元素修改为e1，使其仍有序
108. **void** ModifyElem(LinkNode \*L, ElemType e, ElemType e1) {
109. // 先删除所有e
110. DeleteElemAll(L, e);
111. // 再插入e1（保持有序）
112. InsertElem(L, e1);
113. }
115. // 9）将有序单链表L中属于区间[low,high]的数据取出存入新的有序单链表L1，原链表L不修改
116. **void** RangeExtract(LinkNode \*L, LinkNode \*&L1, ElemType low, ElemType high) {
117. InitList(L1);
118. LinkNode \*p = L->next;
119. **while** (p != NULL) {
120. **if** (p->data >= low && p->data <= high) {
121. InsertElem(L1, p->data);
122. }
123. p = p->next;
124. }
125. }
127. // 10）求两个有序单链表LA和LB的并集，存入LC（无重复）
128. **void** UnionList(LinkNode \*LA, LinkNode \*LB, LinkNode \*&LC) {
129. InitList(LC);
130. LinkNode \*pa = LA->next, \*pb = LB->next;
132. **while** (pa != NULL && pb != NULL) {
133. **if** (pa->data < pb->data) {
134. InsertElem(LC, pa->data);
135. pa = pa->next;
136. } **else** **if** (pa->data > pb->data) {
137. InsertElem(LC, pb->data);
138. pb = pb->next;
139. } **else** {
140. InsertElem(LC, pa->data);
141. pa = pa->next;
142. pb = pb->next;
143. }
144. }
145. // 处理剩余部分
146. **while** (pa != NULL) {
147. InsertElem(LC, pa->data);
148. pa = pa->next;
149. }
150. **while** (pb != NULL) {
151. InsertElem(LC, pb->data);
152. pb = pb->next;
153. }
154. }
156. // 11）求两个有序单链表LA和LB的交集，存入LC
157. **void** IntersectList(LinkNode \*LA, LinkNode \*LB, LinkNode \*&LC) {
158. InitList(LC);
159. LinkNode \*pa = LA->next, \*pb = LB->next;
161. **while** (pa != NULL && pb != NULL) {
162. **if** (pa->data < pb->data) {
163. pa = pa->next;
164. } **else** **if** (pa->data > pb->data) {
165. pb = pb->next;
166. } **else** {
167. InsertElem(LC, pa->data); // 相等才插入
168. pa = pa->next;
169. pb = pb->next;
170. }
171. }
172. }
174. // 12）求LA与LB的差集（LA - LB），即属于LA但不属于LB的元素
175. **void** DiffList(LinkNode \*LA, LinkNode \*LB, LinkNode \*&LC) {
176. InitList(LC);
177. LinkNode \*pa = LA->next, \*pb = LB->next;
179. **while** (pa != NULL && pb != NULL) {
180. **if** (pa->data < pb->data) {
181. InsertElem(LC, pa->data);
182. pa = pa->next;
183. } **else** **if** (pa->data > pb->data) {
184. pb = pb->next;
185. } **else** {
186. // 相等，都不插入（差集排除公共部分）
187. pa = pa->next;
188. pb = pb->next;
189. }
190. }
191. // LA剩余部分都属于差集
192. **while** (pa != NULL) {
193. InsertElem(LC, pa->data);
194. pa = pa->next;
195. }
196. }
198. // 尾插法创建有序单链表
199. **void** CreateSortedList(LinkNode \*&L, ElemType a[], **int** n) {
200. InitList(L);
201. **for** (**int** i = 0; i < n; i++) {
202. InsertElem(L, a[i]);
203. }
204. }
206. **int** main() {
207. LinkNode \*L, \*L1, \*LA, \*LB, \*LC;
208. ElemType a[] = {1, 3, 5, 6, 7, 8, 9};
209. ElemType b[] = {2, 3, 5, 7, 10, 12};
210. **int** len;
211. ElemType e;
213. // 1）初始化
214. InitList(L);
216. // 2）建立有序单链表
217. **for** (**int** i = 0; i < 7; i++) {
218. InsertElem(L, a[i]);
219. }
220. printf("初始化并插入元素后: ");
221. DisList(L);
223. // 3）求长度
224. len = LengthList(L);
225. printf("链表长度: %d\n", len);
227. // 4）取第3个元素
228. **if** (GetElem(L, 3, e)) {
229. printf("第3个元素是: %d\n", e);
230. }
232. // 5）查找值为5的元素位置
233. **int** pos = LocateElem(L, 5);
234. printf("值为5的元素位置: %d\n", pos);
236. // 6）插入元素4
237. InsertElem(L, 4);
238. printf("插入4后: ");
239. DisList(L);
241. // 7）删除所有值为5的元素
242. DeleteElemAll(L, 5);
243. printf("删除所有5后: ");
244. DisList(L);
246. // 8）将值为6的元素改为10
247. ModifyElem(L, 6, 10);
248. printf("将6改为10后: ");
249. DisList(L);
251. // 9）提取区间[3,8]的元素到L1
252. RangeExtract(L, L1, 3, 8);
253. printf("区间[3,8]的元素: ");
254. DisList(L1);
256. //创建LA和LB用于集合运算
257. CreateSortedList(LA, a, 7);
258. CreateSortedList(LB, b, 6);
260. // 10)并集
261. UnionList(LA, LB, LC);
262. printf("LA和LB的并集: ");
263. DisList(LC);
264. DestroyList(LC);
266. // 11)交集
267. IntersectList(LA, LB, LC);
268. printf("LA和LB的交集: ");
269. DisList(LC);
270. DestroyList(LC);
272. // 12)差集 LA - LB
273. DiffList(LA, LB, LC);
274. printf("LA - LB 的差集: ");
275. DisList(LC);
277. // 释放内存
278. DestroyList(L);
279. DestroyList(L1);
280. DestroyList(LA);
281. DestroyList(LB);
282. DestroyList(LC);
284. **return** 0;
285. }

## 7.实验数据和实验结果分析。

实验数据：

    ElemType a**[]** **=** **{**1**,** 3**,** 5**,** 6**,** 7**,** 8**,** 9**};**    ElemType b**[]** **=** **{**2**,** 3**,** 5**,** 7**,** 10**,** 12**};**

实验结果：



实验结果分析：

## （1）指针操作准确：通过前驱指针 p 实现插入删除，避免断链； （2）有序性维护良好：所有插入均基于比较定位，保持递增； （3）集合运算高效：利用有序特性，采用归并思想，时间复杂度接近 O(m+n)； （4）去重机制有效：插入时判断重复，确保集合性质； （5）模块化设计清晰：各函数职责明确，便于调试与复用。

## 8.实验体会。

通过本次实验，我深入理解了单链表的动态特性与指针操作的灵活性。相比顺序表，链表在插入和删除时无需移动大量元素，效率更高，但访问第 i 个元素需遍历，时间复杂度为 O(n)。

在实现有序插入和删除时，我特别注意了边界条件，如头结点、空表、重复值等情况。尤其是在删除多个相同元素时，若不及时调整指针会导致漏删，因此采用 while 循环配合条件判断，确保完整性。

集合运算部分让我体会到“有序”带来的优势——通过双指针归并策略，将交并差操作优化至线性时间，远优于暴力查找。

此外，动态内存管理也是一大挑战。每次 malloc 都需对应 free，否则会造成内存泄漏。调试过程中借助打印函数逐步验证每一步结果，极大提升了排错效率。

总体而言，本次实验不仅巩固了链表的基本操作，也锻炼了逻辑思维与代码组织能力。未来可尝试使用模板支持多种数据类型，或实现双向链表以支持反向遍历。

# 三、实验提交内容

实验报告、实验源程序清单和可执行文件。

实验源程序清单：

实验2-单链表.cpp