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22日-2025年9月28日

# 二、程序设计相关信息

## 1.实验题及问题描述。

实验  双链表和循环链表

双链表的定义

typedef  int ElemType;

typedef struct Dnode

{   Elemtype data;                          //数据元素

struct Dnode  \* prior, \* next;     //指针

int freq;

} DlinkList;

注意：

将题目要求定义为子函数，子函数中只能写基本语句完成功能。

main()函数中可以调用dlinklist.cpp和clinklist.cpp中的函数。

在main（）函数中调用子函数进行测试。

1、设有一个双链表h,每个结点中除了有prior、data和next几个域以外,还有一个访问频度域freq,在链表被启用之前,其值均初始化为零。每当进行 Locatenode(h,x)运算时,令元素值为x的结点中freq域的值加1,并调整表中结点的次序,使其按访问频度的递减次序排列,以便使频繁访问的结点总是靠近表头。试写一个符合上述要求的 Locatenode运算的算法。链表中元素值无重复。

2、有序循环单链表rear中,  中位数的元素值。

## 2.实验目的。

（1）掌握双链表、循环链表的结构定义与动态内存管理机制，理解两种链表与单链表的差异；

（2）熟练实现双链表的创建（头插法、尾插法）、遍历、销毁及基于频度的节点调整操作；

（3）理解循环链表 “首尾相连” 特性，掌握其创建、遍历及中位数求解方法；

（4）提升双向指针（双链表）、循环逻辑（循环链表）的操作能力，强化边界条件（表头、表尾）处理意识；

（5）培养模块化编程思维，确保函数职责明确，便于调试与复用。

## 3.数据结构设计。

元素（Elemtype）定义：

1. **typedef** **int** ElemType;

将 ElemType 定义为 int 类型

链表（DLink）的定义：

1. **typedef** **struct** DNode {
2. ElemType data;
3. **struct** DNode \*prior,\*next;
4. **int** freq;
5. } DLinkNode;

freq:专门为 Locatenode 函数设计，用于记录节点被查找的次数，为节点按频度排序提供依据。

## 4.程序结构(程序中的函数调用关系图)。

main()

├── 双链表相关操作

│ ├── ListCreateF() // 头插法创建双链表

│ ├── ListCreateR() // 尾插法创建双链表

│ ├── ListDisplay() // 展示双链表及freq值

│ ├── Locatenode() // 查找节点、更新频度并排序

│ └── ListDestory() // 销毁双链表（释放内存）

└── 循环单链表相关操作

├── ListLinkCreate() // 尾插法创建循环单链表

├── ListLinkDisplay() // 展示循环单链表

├── ListLinkMedian() // 求循环单链表中位数

└── ListDestory() // 销毁循环单链表（复用双链表销毁函数）

## 5.主要的算法描述。

（1）双链表头插法创建（ListCreateF）

遍历数组，创建新节点，插入到头节点与当前首节点之间，最终链表元素顺序与数组相反，freq初始化为 0。

（2）双链表频度排序（Locatenode）

遍历双链表查找值为 x 的节点，找到则freq加 1；

若节点freq大于前驱节点freq，则向前调整节点位置（修改前驱、后继指针），直至freq不大于前驱或到达表头。

（3）循环单链表创建（ListLinkCreate）

遍历数组，创建新节点尾插，最后使尾节点next指向头节点，形成循环结构。

（4）循环单链表中位数求解（ListLinkMedian）

遍历链表统计节点数 n（不含头节点）；

遍历 n/2 次定位到中位数前驱节点：n 为奇数时，中位数为前驱节点的后继数据；n 为偶数时，为前驱与后继节点数据的平均值。

## 6.实验源程序。

1. #include<stdio.h>
2. #include<malloc.h>
3. **typedef** **int** ElemType;
4. **typedef** **struct** DNode {
5. ElemType data;
6. **struct** DNode \*prior,\*next;
7. **int** freq;
8. } DLinkNode;
10. **void** ListCreateF(DLinkNode \*&L, ElemType a[], **int** n);       //头插法建立双链表
11. **void** ListCreateR(DLinkNode \*&L, ElemType a[], **int** n);       //尾插法建立双链表
12. **void** ListDestory(DLinkNode \*&L);                            //删除链表
13. **void** ListDisplay(DLinkNode \*&L);                            //展示双链表及freq值
14. **bool** Locatenode(DLinkNode \*&h, ElemType x);                 //访问双链表并按频度排序

17. **void** ListLinkCreate(DLinkNode \*&L, ElemType a[], **int** n);    //尾插法建立循环单链表
18. **void** ListLinkMedian(DLinkNode \*&L);                         //求循环单链表中位数
19. **void**  ListLinkMedian(DLinkNode \*&L);                        //展示循环单链表
21. //头插法建立双链表
22. **void** ListCreateF(DLinkNode \*&L, ElemType a[], **int** n) {
23. DLinkNode \*s;
24. L = (DLinkNode\*)malloc(**sizeof**(DLinkNode));                //头结点
25. L->next = L->prior = NULL;
26. **for** (**int** i = 0; i < n; i++) {
27. s = (DLinkNode\*)malloc(**sizeof**(DLinkNode));            //创建新结点
28. s->data = a[i];
29. s->next = L->next;
30. **if** (L->next != NULL)L->next->prior = s;
31. s->prior = L;
32. L->next = s;
33. s->freq = 0;
34. }
35. }
36. //尾插法建立双链表
37. **void** ListCreateR(DLinkNode \*&L, ElemType a[], **int** n) {
38. DLinkNode \*s,\*p;
39. L = (DLinkNode\*)malloc(**sizeof**(DLinkNode));
40. L->next = L->prior = NULL;
41. p = L;
42. **for** (**int** i = 0; i < n; i++) {
43. s = (DLinkNode\*)malloc(**sizeof**(DLinkNode));
44. s->data = a[i];
45. s->freq = 0;
46. s->prior = p;
47. p->next = s;
48. s->next = NULL;
49. p = s;
50. }
51. }

54. //删除链表
55. **void** ListDestory(DLinkNode \*&L) {
56. DLinkNode \*pre = L,\*p = pre->next;
57. **while** (p != NULL) {
58. free(pre);
59. pre = p;
60. p = pre->next;
61. }
62. free(pre);
63. }
65. //展示双链表及freq值
66. **void** ListDisplay(DLinkNode \*&L) {
67. DLinkNode \*p = L;
68. printf("List:(data,freq)");
69. **while** (p->next != NULL) {
70. p = p->next;
71. printf("(%d,%d) ", p->data, p->freq);
72. }
73. printf("\n");
74. }

77. //访问双链表并按频度排序
78. **bool** Locatenode(DLinkNode \*&h, ElemType x) {
79. //若找到x则排序并返回true，否则返回false
80. DLinkNode \*p = h;
81. **bool** flag = **false**;
82. //找到x并将对应freq+1，若找不到则返回false
83. **while** (p->next != NULL) {
84. p = p->next;
85. **if** (p->data == x) {
86. p->freq++;
87. flag = **true**;
88. **break**;
89. }
90. }
91. //若成功找到，判断x对应freq是否比前一个节点大，若比前一个节点大则前移一个节点，直到不大于前一个节点
92. **if** (flag) {
94. DLinkNode \*pre = p->prior;
95. **if** (pre == NULL) {
96. } **else** {
97. **while** (pre->prior != NULL && pre->freq < p->freq) {
98. pre->prior->next = p;
99. p->prior = pre->prior;
100. pre->next = p->next;
101. p->next->prior = pre;
102. pre->prior = p;
103. p->next = pre;
104. pre = p->prior;
105. }
107. }
108. }
110. **return** flag;
112. }
113. //尾插法建立循环单链表
114. **void** ListLinkCreate(DLinkNode \*&L, ElemType a[], **int** n) {
115. DLinkNode\*s,\*p;
116. L = (DLinkNode\*)malloc(**sizeof**(DLinkNode));
117. L->prior = NULL;
118. L->next = L;
119. p = L;
120. **for** (**int** i = 0; i < n; i++) {
121. s = (DLinkNode\*)malloc(**sizeof**(DLinkNode));
122. s->data = a[i];
123. s->prior = NULL;
124. s->next = NULL;
125. p->next = s;
126. p = s;
127. }
128. p->next = L;
129. }
130. //展示循环单链表
131. **void** ListLinkDisplay(DLinkNode \*&L) {
132. DLinkNode \*p = L->next;
133. printf("\nList:");
134. **while** (p != L) {
135. printf("%d ", p->data);
136. p = p->next;
137. }
138. printf("\n");
139. }
140. //求循环单链表中位数
141. **void** ListLinkMedian(DLinkNode \*&L) {
142. DLinkNode \*p;
143. **float** m;
144. **int** n = 0;
145. p = L;
146. //确定链表节点个数n，不包括头节点
147. **while** (p->next != L) {
148. p = p->next;
149. n++;
150. }
151. //找中位数
152. p = L;
153. **for** (**int** i = 0; i < n / 2; i++) {
154. p = p->next;
156. }
157. **if** (n % 2 == 0) {
158. m = **float**((p->data + p->next->data) ) / 2;
159. } **else** {
160. m = p->next->data;
161. }
162. printf("中位数为%f", m);
163. }
164. **int** main() {
165. //创建双链表h，验证Locatenode算法
166. DLinkNode \*h;
167. ElemType a[6] = {1, 2, 3, 4, 5, 6};
168. ListCreateF(h, a, 6);
169. ListDisplay(h);
171. //调用Locatenode
172. **for** (**int** i = 0; i <= 3; i++) {
173. Locatenode(h, 3);
174. Locatenode(h, 5);
175. }
176. **for** (**int** i = 0; i <= 4; i++) {
177. Locatenode(h, 6);
178. }
179. Locatenode(h, 4);
180. printf("分别访问4次元素3 ，4次元素5 ，6次元素6 ，1次元素4 \n");
181. ListDisplay(h);
182. ListDestory(h);
184. //创建循环单链表并求中位数
185. DLinkNode \*rear;
186. ElemType b[6] = {1, 2, 3, 4, 5, 6};
187. ListLinkCreate(rear, b, 6);
188. ListLinkDisplay(rear);
189. ListLinkMedian(rear);
190. ListDestory(rear);
192. **return** 0;
193. }

## 7.实验数据和实验结果分析。

实验数据：

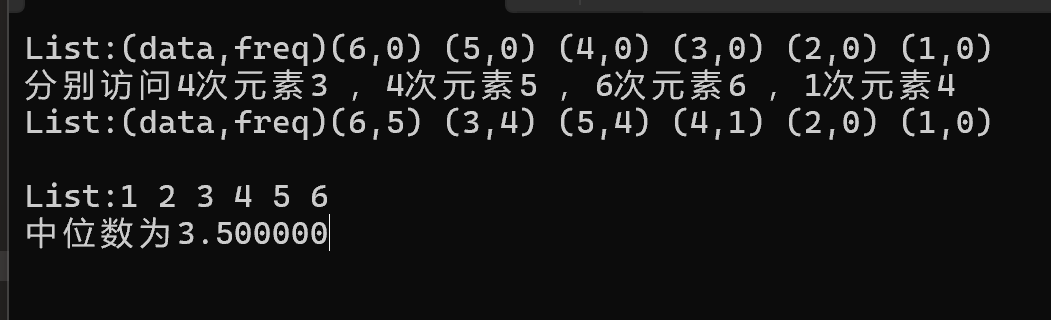
双链表：

ElemType a**[**6**]** **=** **{**1**,** 2**,** 3**,** 4**,** 5**,** 6**};**

循环链表：

ElemType b**[**6**]** **=** **{**1**,** 2**,** 3**,** 4**,** 5**,** 6**};**

实验结果：



实验结果分析：

（1）双链表功能验证：freq与访问次数一致，节点按freq递减排序，无断链、野指针问题，Locatenode逻辑正确；

（2）循环链表功能验证：节点顺序与数组一致，中位数计算符合 “偶数取平均、奇数取中间” 规则，结果正确；

（3）模块化与复用：ListDestory函数适配两种链表，内存释放完整，无泄漏。

## 8.实验体会。

双链表的prior指针解决了单链表 “查找前驱需重遍历” 的问题，使节点调整更高效，但需注意指针修改顺序，避免断链；

循环链表 “首尾相连” 特性简化了遍历终止判断，适合循环访问场景，但需警惕死循环（需判断p != 头节点）；

动态内存管理是关键：每次malloc需对应free，调试时通过打印中间结果可快速定位指针错误；

本次实验巩固了链表的进阶操作，未来可尝试优化Locatenode的调整效率（如跳跃调整），或扩展链表支持多数据类型。

# 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3-双链表和循环链表.cpp