

# 数据结构作业 第二周

霍斌 PB24111627

我将所有代码封装在了`LinearList.h`中,完整头文件请见附录

## 1 2.19

时间复杂度分析:

1. 第一个循环: 时间复杂度为  $O(n_1)$
2. 第二个循环: 时间复杂度为  $O(n_2)$
3. 删除节点循环: 时间复杂度为  $O(n_2 - n_1)$

综上, 时间复杂度为  $O(n)$

```
1  bool deleteFromMin2Max(LinearList *list, int mink, int maxk) {
2      //已知链表中元素以值递增有序排列
3      if (list->head->data == 0 ||
4          maxk < mink ||
5          (list->head->next != NULL && list->head->next->data >= maxk)
6      )
7          return false; //处理异常情况
8      Node *leftGapNode = list->head->next;
9
10     while ( leftGapNode->next != NULL && leftGapNode->next->data <= mink) {
11         leftGapNode = leftGapNode->next;
12     }
13     //如果一直到链表末端仍然没有人大于mink, 那么也没了
14     if (leftGapNode->next == NULL) {
15         return false;
16     }
17
18     Node *rightGapNode = leftGapNode->next;
19
20     while (rightGapNode != NULL && rightGapNode->data < maxk) {
21         rightGapNode = rightGapNode->next;
22     }
23
24     //删除leftGapNode和rightGapNode之间的所有节点
25     Node *curNode = leftGapNode->next;
26     while (curNode != rightGapNode) {
27         Node *nodeToDelete = curNode;
28         curNode = curNode->next;
29         free(nodeToDelete);
30         list->head->data--;
31     }
32
33     //最后弥合东西分裂
34     leftGapNode->next = rightGapNode;
35     return true;
36 }
```

## 2 2.21

```
1 bool reverse(LinearList *list) {
2     if ( list->head->data <= 1 ) return false;
3
4
5     Node *prev = NULL;
6     Node *current = list->head->next; // 跳过头结点
7     Node *next = NULL;
8
9     while (current != NULL) {
10         next = current->next; // 保存下一个节点
11         current->next = prev; // 反转当前节点的指针
12         prev = current;      // 移动prev和current指针
13         current = next;
14     }
15
16     list->head->next = prev; // 更新头结点的next指针
17     return true;
18 }
```

## 3 2.24

```
1 bool mergeListsByDecrease(LinearList *list1, LinearList *list2) {
2     // 仍假设两个链表均为递增有序排列，合并后链表使用list1->head，并释放list2->head和list2
3     if (!list1 || !list2 || !list1->head || !list2->head) return false;
4     if (list1->head->data == 0 && list2->head->data == 0) return false; // 两个链表均为
    空
5
6     Node *dummy = (Node *)malloc(sizeof(Node));
7     dummy->next = NULL;
8     Node *tail = dummy;
9     Node *cur1 = list1->head->next;
10    Node *cur2 = list2->head->next;
11
12    while (cur1 && cur2) {
13        if (cur1->data <= cur2->data) {
14            tail->next = cur1;
15            cur1 = cur1->next;
16        } else {
17            tail->next = cur2;
18            cur2 = cur2->next;
19        }
20        tail = tail->next;
21    }
22    if (cur1) tail->next = cur1;
23    if (cur2) tail->next = cur2;
24
25    // 更新list1的头结点
26    list1->head->next = dummy->next;
27    list1->head->data = list1->head->data + list2->head->data;
28
29    // 释放dummy节点和list2的头结点及结构体
30    free(dummy);
31    // free(list2->head);
```

```

32     // free(list2);
33     reverse(list1);
34     return true;
35 }

```

## 4 2.29

时间复杂度分析:

不妨设三个链表的长度分别为 $n_1, n_2, n_3$ .

1. 第一个循环: 最坏情况是遍历整个2, 3链表, 也即 $O(n_2 + n_3)$ .
2. 第二个循环: 无论如何最后也要遍历整个链表1, 但是总共只需遍历一次(并非和第一次循环倍乘, 而是叠加), 因此复杂度为 $O(n_1)$ .

综上, 复杂度为 $O(n_1 + n_2 + n_3)$ , 即 $O(n)$ .

```

1  bool deleteFrom1WhoBothAppearIn2and3(LinearList *list1, LinearList *list2, LinearList
    *list3) {
2      //从表1中删去表2和表3中共有的值
3      //使用双指针逐步得到BC的重复值, 然后再删除A中对应的人
4      if( list1->head->data == 0 || list2->head->data == 0 || list3->head->data == 0 )
        return false; //有一个链表为空, 那么就没有交集
5
6      Node *prev = list1->head;
7      Node *cur1 = list1->head->next;
8      Node *next = NULL;
9      Node *cur2 = list2->head->next;
10     Node *cur3 = list3->head->next;
11
12
13     int duplicate = 0;
14
15     while (cur2 && cur3) {
16         if (cur2->data < cur3->data) {
17             cur2 = cur2->next;
18         } else if (cur2->data > cur3->data) {
19             cur3 = cur3->next;
20         } else { //如果找到重复值了, 去往表1中删除对应元素
21             duplicate = cur2->data;
22             while(cur1 != NULL && cur1->data < duplicate) {
23                 prev = cur1;
24                 cur1 = cur1->next;
25             }
26             //找到了重复值
27             if (cur1->data == duplicate) { //如果发现了重复值, 那么删掉这个人
28                 next = cur1->next;
29                 free(cur1);
30                 cur1 = next;
31                 prev->next = cur1;
32                 list1->head->data--;
33             }
34             //如果cur1已经超过了重复值, 那么需要使cur2或者cur3再动一步防止死循环
35             if(cur1->data > duplicate) cur2 = cur2->next;
36         }
37     }

```

```
38 |
39 |     return true;
40 | }
```

## 附录

```
1  #ifndef LINEARLIST_H
2  #define LINEARLIST_H
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <stdbool.h>
7  //
8  typedef struct Node {
9      int data;
10     struct Node *next;
11 } Node;
12
13 typedef struct {
14     Node *head;    // use head node to store length of list
15 } LinearList;
16
17
18 void initList(LinearList *list) {
19     if (!list) return;
20
21     list->head = (Node *)malloc(sizeof(Node));
22     if (!list->head) {
23         fprintf(stderr, "Memory allocation failed\n");
24         return;
25     }
26     list->head->data = 0;
27     list->head->next = NULL;
28 }
29
30 int getSize(const LinearList *list) {
31     if (!list || !list->head) return 0;
32     return list->head->data;
33 }
34
35 bool isValidIndex(const LinearList *list, int index, bool allowEnd) {
36     if (!list || !list->head) return false;
37     int size = list->head->data;
38     if (allowEnd) {
39         return index >= 0 && index <= size;
40     } else {
41         return index >= 0 && index < size;
42     }
43 }
44
45 // 在指定位置插入元素
46 bool insert(LinearList *list, int index, int value) {
47     if (!list || !list->head) {
48         fprintf(stderr, "List not initialized\n");
49         return false;
```

```

50     }
51
52     if (!isValidIndex(list, index, true)) {
53         fprintf(stderr, "Index out of bounds\n");
54         return false;
55     }
56
57     Node *newNode = (Node *)malloc(sizeof(Node));
58     if (!newNode) {
59         fprintf(stderr, "Memory allocation failed\n");
60         return false;
61     }
62
63     newNode->data = value;
64     Node *curNode = list->head;
65
66
67     for (int i = 0; i < index; i++) {
68         curNode = curNode->next;
69     }
70
71     newNode->next = curNode->next;
72     curNode->next = newNode;
73     list->head->data++;
74
75     return true;
76 }
77
78
79 bool extract(LinearList *list, int index) {
80     if (!list || !list->head) {
81         fprintf(stderr, "List not initialized\n");
82         return false;
83     }
84
85     if (!isValidIndex(list, index, false)) {
86         fprintf(stderr, "Index out of bounds\n");
87         return false;
88     }
89
90     Node *curNode = list->head;
91
92     for (int i = 0; i < index; i++) {
93         curNode = curNode->next;
94     }
95
96     Node *nodeToDelete = curNode->next;
97     curNode->next = nodeToDelete->next;
98     free(nodeToDelete);
99     list->head->data--;
100
101     return true;
102 }
103
104 // 获取指定位置的元素
105 int get(const LinearList *list, int index) {

```

```

106     if (!list || !list->head) {
107         fprintf(stderr, "List not initialized\n");
108         return -1;
109     }
110
111     if (!isValidIndex(list, index, false)) {
112         fprintf(stderr, "Index out of bounds\n");
113         return -1;
114     }
115
116     Node *curNode = list->head->next;
117
118     for (int i = 0; i < index; i++) {
119         curNode = curNode->next;
120     }
121
122     return curNode->data;
123 }
124
125 // 打印链表
126 void printList(const LinearList *list) {
127     if (!list || !list->head) {
128         printf("List not initialized\n");
129         return;
130     }
131
132     printf("List (size: %d): ", list->head->data);
133     Node *curNode = list->head->next;
134
135     while (curNode != NULL) {
136         printf("%d", curNode->data);
137         if (curNode->next != NULL) {
138             printf(" -> ");
139         }
140         curNode = curNode->next;
141     }
142     printf("\n");
143 }
144
145 bool addToEnd(LinearList *list, int value) {
146     if (!list || !list->head) return false;
147     return insert(list, list->head->data, value);
148 }
149
150 bool addToFront(LinearList *list, int value) {
151     return insert(list, 0, value);
152 }
153
154 // 删除最小值到最大值之间的所有元素(不包括端点)
155 bool deleteFromMin2Max(LinearList *list, int mink, int maxk) {
156     // 已知链表中元素以值递增有序排列
157     if (list->head->data == 0 ||
158         maxk < mink ||
159         (list->head->next != NULL && list->head->next->data >= maxk)
160     )
161         return false; // 处理异常情况

```

```

162     Node *leftGapNode = list->head->next;
163
164     while ( leftGapNode->next != NULL && leftGapNode->next->data <= mink) {
165         leftGapNode = leftGapNode->next;
166     }
167     //如果一直到链表末端仍然没有人大于mink, 那么也没了
168     if (leftGapNode->next == NULL) {
169         return false;
170     }
171
172     Node *rightGapNode = leftGapNode->next;
173
174     while (rightGapNode != NULL && rightGapNode->data < maxk) {
175         rightGapNode = rightGapNode->next;
176     }
177
178     //删除leftGapNode和rightGapNode之间的所有节点
179     Node *curNode = leftGapNode->next;
180     while (curNode != rightGapNode) {
181         Node *nodeToDelete = curNode;
182         curNode = curNode->next;
183         free(nodeToDelete);
184         list->head->data--;
185     }
186
187     //最后弥合东西分裂
188     leftGapNode->next = rightGapNode;
189     return true;
190 }
191
192 bool reverse(LinearList *list) {
193     if ( list->head->data <= 1 ) return false;
194
195
196     Node *prev = NULL;
197     Node *current = list->head->next; // 跳过头结点
198     Node *next = NULL;
199
200     while (current != NULL) {
201         next = current->next; // 保存下一个节点
202         current->next = prev; // 反转当前节点的指针
203         prev = current;      // 移动prev和current指针
204         current = next;
205     }
206
207     list->head->next = prev; // 更新头结点的next指针
208     return true;
209 }
210
211 bool mergeListsByDecrease(LinearList *list1, LinearList *list2) {
212     // 仍假设两个链表均为递增有序排列, 合并后链表使用list1->head, 并释放list2->head和list2
213     if (!list1 || !list2 || !list1->head || !list2->head) return false;
214     if (list1->head->data == 0 && list2->head->data == 0) return false; // 两个链表均
    为空
215
216     Node *dummy = (Node *)malloc(sizeof(Node));

```

```

217     dummy->next = NULL;
218     Node *tail = dummy;
219     Node *cur1 = list1->head->next;
220     Node *cur2 = list2->head->next;
221
222     while (cur1 && cur2) {
223         if (cur1->data <= cur2->data) {
224             tail->next = cur1;
225             cur1 = cur1->next;
226         } else {
227             tail->next = cur2;
228             cur2 = cur2->next;
229         }
230         tail = tail->next;
231     }
232     if (cur1) tail->next = cur1;
233     if (cur2) tail->next = cur2;
234
235     // 更新list1的头结点
236     list1->head->next = dummy->next;
237     list1->head->data = list1->head->data + list2->head->data;
238
239     // 释放dummy节点和list2的头结点及结构体
240     free(dummy);
241     // free(list2->head);
242     // free(list2);
243     reverse(list1);
244     return true;
245 }
246
247 bool deleteFrom1WhoBothAppearIn2and3(LinearList *list1, LinearList *list2, LinearList
248 *list3) {
249     //从表1中删去表2和表3中共有的值
250     //使用双指针逐步得到BC的重复值，然后再删除A中对应的人
251     if( list1->head->data == 0 || list2->head->data == 0 || list3->head->data == 0 )
252         return false; //有一个链表为空，那么就没有交集
253
254     Node *prev = list1->head;
255     Node *cur1 = list1->head->next;
256     Node *next = NULL;
257     Node *cur2 = list2->head->next;
258     Node *cur3 = list3->head->next;
259
260     int duplicate = 0;
261
262     while (cur2 && cur3) {
263         if (cur2->data < cur3->data) {
264             cur2 = cur2->next;
265         } else if (cur2->data > cur3->data) {
266             cur3 = cur3->next;
267         } else { //如果找到重复值了，去往表1中删除对应元素
268             duplicate = cur2->data;
269             while(cur1 != NULL && cur1->data < duplicate) {
270                 prev = cur1;
271                 cur1 = cur1->next;

```



```
271
272     } //找到了重复值
273     if (cur1->data == duplicate) { //如果发现了重复值，那么删掉这个人
274         next = cur1->next;
275         free(cur1);
276         cur1 = next;
277         prev->next = cur1;
278         list1->head->data--;
279     }
280     //如果cur1已经超过了重复值，那么需要使cur2或者cur3再动一步防止死循环
281     if(cur1->data > duplicate) cur2 = cur2->next;
282 }
283 }
284
285     return true;
286 }
287 #endif // LINEARLIST_H
```