

西北工业大学



数据结构习题集

Problem Set of Data Structures

软件学院 钱锋 编

2024 年 4 月 14 日

目录

2	线性表	1	参考文献	15
3	栈和队列	13		

第2章 线性表

习题 2.0.1. 1) 在顺序表中插入或阐述一个元素, 平均需要移动 $n/2$ 个元素, 其中 n 为顺序表当前的已使用长度, 具体移动的元素个数与表长和插入元素的目的位置有关.

2) 顺序表中逻辑上相邻的元素物理位置一定相邻, 但单链表中逻辑上相邻的元素的物理位置未必相邻.

3) 在单链表中, 除了首元节点外, 任意节点的存储位置有由前驱元素的 `next` 指针导出.

4) 单链表中设置头节点的作用是插入/删除元素时无需进行特殊处理.

习题 2.0.2. 设顺序表中的数据元素递增有序. 试写一算法, 将 x 插入到顺序表的适当位置上, 以保持该表的有序性.

```
1  #include <stdio.h>
2
3  void insert(int x, int *arr, int *size);
4
5  int main() {
6      // 不妨设要插入元素的是一个整型数组
7      int arr[1000];
8      int elenum;
9      elenum = scanf(" %d", &elenum);
10
11     int i=0;
12     printf("请输入一行数字（以空格分隔，以回车结束）：\n");
13
14     // 从用户输入中读取数字，直到遇到换行符为止
15     while (scanf(" %d", &arr[i]) == 1) {
16         i++;
17     }
18
19     printf("您输入的数字为：\n");
20     for (int j = 0; j < i; j++) {
21         printf("%d ", arr[j]);
22     }
23     printf("\n");
24
25     int size = elenum;
26 }
```

```

27     // 插入元素测试
28     int newData = 3;
29     insert(newData, arr, &size);
30     for (int i=0; i<=size; i++) {
31         printf(" %d ", arr[i]);
32     }
33     return 0;
34 }
35
36 void insert(int x, int *arr, int *size){
37     if size >= 10) printf(" 无法插入新元素, 因为顺序表已满.");return;// 确定插入位置
        int i=0;while (x>=arr[i]) i+=1;// 移动数据 for (int j=*size-1; j>=i; j--)
        arr[j+1] = arr[j];// 插入数据 arr[i] = x;// 改变长度 *size += 1;

```

习题 2.0.3. 试写一算法, 实现顺序表的就地逆置, 即利用原表的存储空间将线性表 (a_1, a_2, \dots, a_n) 就地逆置.

```

1  #include <stdio.h>
2
3  void reverse(int *arr, int size) {
4      int *start = arr; // 指向起始位置的指针
5      int *end = arr + size - 1; // 指向末尾位置的指针
6
7      while (start < end) {
8          // 交换起始位置和末尾位置的元素
9          int temp = *start;
10         *start = *end;
11         *end = temp;
12
13         // 移动指针
14         start++;
15         end--;
16     }
17 }
18
19 int main() {
20     int arr[] = {1, 2, 3, 4, 5};
21     int size = sizeof(arr) / sizeof(arr[0]);
22
23     printf("Original array:");
24     for (int i = 0; i < size; i++) {
25         printf(" %d", arr[i]);
26     }
27     printf("\n");
28

```

```
29     reverse(arr, size);
30
31     printf("Reversed array:");
32     for (int i = 0; i < size; i++) {
33         printf(" %d", arr[i]);
34     }
35     printf("\n");
36
37     return 0;
38 }
```

习题 2.0.4. 假设有两个按元素值递增有序排列的线性表 A 和 B，均以单链表作存储结构，请编写算法将 A 表和 B 表归并成一个按元素值递减有序（即非递增有序，允许表中含有值相同的元素）排列的线性表 C，并要求利用原表（即 A 表和 B 表）的结点空间构造 C 表。

```
1     #include <stdio.h>
2     #include <stdlib.h>
3
4     // 定义单链表结点
5     typedef struct Node {
6         int data;
7         struct Node *next;
8     } Node;
9
10    // 将两个有序链表合并成一个有序链表，递减有序
11    Node* merge_lists(Node *A, Node *B) {
12        Node *C = NULL; // 归并后的链表
13        Node *temp;
14
15        // 比较两个链表的节点，将较小的节点插入 C 链表中
16        while (A != NULL && B != NULL) {
17            if (A->data >= B->data) {
18                temp = A;
19                A = A->next;
20            } else {
21                temp = B;
22                B = B->next;
23            }
24            temp->next = C;
25            C = temp;
26        }
27
28        // 将剩余节点插入到 C 链表中
29        while (A != NULL) {
30            temp = A;
```

```
31     A = A->next;
32     temp->next = C;
33     C = temp;
34 }
35 while (B != NULL) {
36     temp = B;
37     B = B->next;
38     temp->next = C;
39     C = temp;
40 }
41
42 return C;
43 }
44
45 // 打印链表
46 void print_list(Node *head) {
47     while (head != NULL) {
48         printf("%d ", head->data);
49         head = head->next;
50     }
51     printf("\n");
52 }
53
54 int main() {
55     // 初始化两个有序链表 A 和 B
56     Node *A = (Node *)malloc(sizeof(Node));
57     A->data = 2;
58     A->next = (Node *)malloc(sizeof(Node));
59     A->next->data = 5;
60     A->next->next = (Node *)malloc(sizeof(Node));
61     A->next->next->data = 7;
62     A->next->next->next = NULL;
63
64     Node *B = (Node *)malloc(sizeof(Node));
65     B->data = 3;
66     B->next = (Node *)malloc(sizeof(Node));
67     B->next->data = 4;
68     B->next->next = (Node *)malloc(sizeof(Node));
69     B->next->next->data = 6;
70     B->next->next->next = NULL;
71
72     printf("原始链表 A: ");
73     print_list(A);
74     printf("原始链表 B: ");
75     print_list(B);
```



```

76
77 // 归并两个有序链表为一个递减有序链表
78 Node *C = merge_lists(A, B);
79
80 printf("归并后的链表 C: ");
81 print_list(C);
82
83 // 释放内存
84 while (A != NULL) {
85     Node *temp = A;
86     A = A->next;
87     free(temp);
88 }
89 while (B != NULL) {
90     Node *temp = B;
91     B = B->next;
92     free(temp);
93 }
94 while (C != NULL) {
95     Node *temp = C;
96     C = C->next;
97     free(temp);
98 }
99
100 return 0;
101 }

```

习题 2.0.5. 已知 A,B 和 C 为三个递增有序的线性表，现要求对 A 表作如下操作：删去那些既在 B 表中出现又在 C 表中出现的元素。试对顺序表编写实现上述操作的算法，并分析你的算法的时间复杂度（注意：题中没有特别指明同一表中的元素值各不相同）。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // 定义线性表结构
5 typedef struct {
6     int *data; // 数据域
7     int length; // 线性表长度
8 } SeqList;
9
10 // 初始化线性表
11 void init_list(SeqList *list, int length) {
12     list->data = (int *)malloc(length * sizeof(int));
13     if (list->data == NULL) {
14         printf("内存分配失败\n");

```

```
15     exit(1);
16 }
17 list->length = length;
18 }
19
20 // 删除在B表中出现且在C表中也出现的元素
21 void delete_common_elements(SeqList *A, SeqList *B, SeqList *C) {
22     int i = 0, j = 0, k = 0;
23     while (i < A->length && j < B->length && k < C->length) {
24         if (B->data[j] < A->data[i]) {
25             j++;
26         } else if (B->data[j] > A->data[i]) {
27             i++;
28         } else { // B->data[j] == A->data[i]
29             if (C->data[k] < A->data[i]) {
30                 k++;
31             } else if (C->data[k] > A->data[i]) {
32                 i++;
33             } else { // C->data[k] == A->data[i]
34                 // 删除元素
35                 for (int m = i; m < A->length - 1; m++) {
36                     A->data[m] = A->data[m + 1];
37                 }
38                 A->length--;
39                 // 继续比较下一个元素
40                 j++;
41                 k++;
42             }
43         }
44     }
45 }
46
47 // 打印线性表
48 void print_list(SeqList *list) {
49     for (int i = 0; i < list->length; i++) {
50         printf("%d ", list->data[i]);
51     }
52     printf("\n");
53 }
54
55 int main() {
56     // 初始化线性表 A, B, C
57     SeqList A, B, C;
58     int length_A = 7;
59     int data_A[] = {1, 3, 5, 7, 9, 11, 13};
```

```

60     int length_B = 6;
61     int data_B[] = {3, 6, 7, 10, 11, 14};
62     int length_C = 5;
63     int data_C[] = {2, 5, 7, 8, 11};
64     init_list(&A, length_A);
65     init_list(&B, length_B);
66     init_list(&C, length_C);
67     A.data = data_A;
68     B.data = data_B;
69     C.data = data_C;
70
71     printf("初始线性表 A: ");
72     print_list(&A);
73     printf("线性表 B: ");
74     print_list(&B);
75     printf("线性表 C: ");
76     print_list(&C);
77
78     // 删除在 B 表中出现且在 C 表中也出现的元素
79     delete_common_elements(&A, &B, &C);
80
81     printf("执行操作后的线性表 A: ");
82     print_list(&A);
83
84     // 释放内存
85     free(A.data);
86     free(B.data);
87     free(C.data);
88
89     return 0;
90 }

```

该算法首先初始化了三个线性表 A, B, C, 并分别存储在数组中。然后使用双指针遍历它们, 比较元素大小, 删除在 B 表中出现且在 C 表中也出现的元素。删除元素的时间复杂度为 $O(n)$, 其中 n 为 A 表的长度。因为该算法涉及对 A 表的元素进行遍历和删除操作, 所以时间复杂度为 $O(n^2)$ 。

习题 2.0.6. 有一个双向循环链表, 每个结点中除有 `prior`, `data` 和 `next` 三个域外, 还增设了一个访问频度域 `freq`。在链表被起用之前, 频度域 `freq` 的值均初始化为零, 而每当对链表进行一次 `LOCATE(L, x)` 的操作后, 被访问的结点 (即元素值等于 x 的结点) 中的频度域 `freq` 的值便增 1, 同时调整链表中结点之间的次序, 使其按访问频度非递增的次序顺序排列, 以便始终保持被频繁访问的结点总是靠近表头结点。试编写符合上述要求的 `LOCATE` 操作的算法。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // 定义双向循环链表的结点

```

```
5 typedef struct Node {
6     int data; // 数据域
7     int freq; // 访问频度域
8     struct Node *prior; // 指向前一个结点的指针
9     struct Node *next; // 指向后一个结点的指针
10 } Node;
11
12 // 初始化双向循环链表
13 Node* init_list() {
14     Node *head = (Node *)malloc(sizeof(Node));
15     if (head == NULL) {
16         printf("内存分配失败\n");
17         exit(1);
18     }
19     head->data = -1; // 表头结点数据域设为-1
20     head->freq = 0;
21     head->prior = head;
22     head->next = head;
23     return head;
24 }
25
26 // 在链表尾部添加结点
27 void append_node(Node *head, int data) {
28     Node *new_node = (Node *)malloc(sizeof(Node));
29     if (new_node == NULL) {
30         printf("内存分配失败\n");
31         exit(1);
32     }
33     new_node->data = data;
34     new_node->freq = 0;
35     new_node->prior = head->prior;
36     new_node->next = head;
37     head->prior->next = new_node;
38     head->prior = new_node;
39 }
40
41 // 将结点移动到链表头部
42 void move_to_head(Node *head, Node *node) {
43     node->freq++;
44     Node *prev = node->prior;
45     Node *next = node->next;
46
47     // 从原位置删除结点
48     prev->next = next;
49     next->prior = prev;
```

```
50
51 // 插入到表头
52 node->next = head->next;
53 head->next->prior = node;
54 head->next = node;
55 node->prior = head;
56 }
57
58 // 查找指定值的结点并将其移动到链表头部
59 void locate(Node *head, int data) {
60     Node *current = head->next;
61     while (current != head) {
62         if (current->data == data) {
63             move_to_head(head, current);
64             return;
65         }
66         current = current->next;
67     }
68     printf("未找到值为 %d 的结点\n", data);
69 }
70
71 // 打印链表
72 void print_list(Node *head) {
73     Node *current = head->next;
74     while (current != head) {
75         printf("%d(f=%d) ", current->data, current->freq);
76         current = current->next;
77     }
78     printf("\n");
79 }
80
81 int main() {
82     Node *head = init_list();
83
84     // 在链表尾部添加一些结点
85     append_node(head, 1);
86     append_node(head, 2);
87     append_node(head, 3);
88     append_node(head, 4);
89     append_node(head, 5);
90
91     printf("初始链表: ");
92     print_list(head);
93
94     // 对链表进行 LOCATE 操作
```

```

95     locate(head, 3);
96     printf("执行 LOCATE 操作后: ");
97     print_list(head);
98
99     locate(head, 5);
100    printf("再次执行 LOCATE 操作后: ");
101    print_list(head);
102
103    // 释放内存
104    Node *current = head->next;
105    while (current != head) {
106        Node *temp = current;
107        current = current->next;
108        free(temp);
109    }
110    free(head);
111
112    return 0;
113 }

```

习题 2.0.7. 试以循环链表作稀疏多项式的存储结构, 编写求其导函数的算法, 要求利用原多项式中的结点空间存放其导函数 (多项式), 同时释放所有无用节点.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // 定义多项式的结点
5  typedef struct Node {
6      int coef; // 系数
7      int expo; // 指数
8      struct Node *next; // 指向下一个结点的指针
9  } Node;
10
11 // 创建空的循环链表
12 Node *create_empty_poly() {
13     Node *head = (Node *)malloc(sizeof(Node));
14     if (head == NULL) {
15         printf("内存分配失败\n");
16         exit(1);
17     }
18     head->next = head; // 使其指向自己形成循环链表
19     return head;
20 }
21
22 // 在多项式的最后添加一个节点

```

```
23 void append_node(Node *head, int coef, int expo) {
24     Node *new_node = (Node *)malloc(sizeof(Node));
25     if (new_node == NULL) {
26         printf("内存分配失败\n");
27         exit(1);
28     }
29     new_node->coef = coef;
30     new_node->expo = expo;
31     new_node->next = head->next;
32     head->next = new_node;
33 }
34
35 // 释放链表的所有结点
36 void free_poly(Node *head) {
37     Node *temp;
38     while (head->next != head) {
39         temp = head->next;
40         head->next = temp->next;
41         free(temp);
42     }
43     free(head);
44 }
45
46 // 打印多项式
47 void print_poly(Node *head) {
48     Node *current = head->next;
49     if (current == head) {
50         printf("多项式为\n");
51         return;
52     }
53     while (current != head) {
54         printf("%dx^%d ", current->coef, current->expo);
55         current = current->next;
56         if (current != head) {
57             printf("+ ");
58         }
59     }
60     printf("\n");
61 }
62
63 // 求多项式的导函数并存储在原多项式中
64 void derivative(Node *head) {
65     Node *current = head->next;
66     Node *prev = head;
67     while (current != head) {
```

```
68     // 计算导数
69     current->coef *= current->expo;
70     current->expo -= 1;
71
72     // 如果指数为负则删除该节点
73     if (current->expo < 0) {
74         Node *temp = current;
75         prev->next = current->next;
76         current = current->next;
77         free(temp);
78     } else {
79         prev = current;
80         current = current->next;
81     }
82 }
83 }
84
85 int main() {
86     // 创建一个空的循环链表
87     Node *poly = create_empty_poly();
88
89     // 添加多项式的各项
90     append_node(poly, 3, 4);
91     append_node(poly, -2, 3);
92     append_node(poly, 5, 2);
93     append_node(poly, 6, 1);
94     append_node(poly, -8, 0);
95
96     printf("原多项式: ");
97     print_poly(poly);
98
99     // 求导函数
100    derivative(poly);
101    printf("导函数: ");
102    print_poly(poly);
103
104    // 释放内存
105    free_poly(poly);
106
107    return 0;
108 }
```


第3章 栈和队列

习题 3.0.1. 简述下列算法的功能 (栈的元素类型 SElemType 为 int).

```
1 | Status algo1(Stack S) {  
2 |     int i, n, A[255];  
3 |     n=0;  
4 |     while (!StackEmpty(S)) {n++; pop(S, A[n]);};  
5 |     for (i=1; i<=n; i++) Push(S, A[i]);  
6 | }
```

评注. 对栈的操作主要有两种, 分别是入栈 `push()` 和出栈 `pop()`, 它们都是对栈顶进行的操作. 在严蔚敏版数据结构教材 [6] 中, 出栈操作定义为 `pop(Stack &S, SElemType &e)`, 即删除栈顶元素, 并将其值返回到变量 `e` 中. 因此我们不妨用一个例子来直观的叙述该算法的功能.

假设 S 是一个栈, 它的元素为 (a, b, c, d) , 我们约定 d 为当前的栈顶, 那么在满足 S 非空时, 我们不断的进行出栈操作, 并将每一次出栈的元素返回到数组 A 中, 因此这一系列操作中每一步的结果是:

```
1 | // 初始状态  
2 | S = (a, b, c, d)  
3 | A = []  
4 |  
5 | // 出栈  
6 | 1: S = (a, b, c), A = [NULL, d]  
7 | 2: S = (a, b),   A = [NULL, d, c]  
8 | 3: S = (a),      A = [NULL, d, c, b]  
9 | 4: S = (),       A = [NULL, d, c, b, a]
```

注意到数组 A 中索引为 0 处实际上是没有元素的, 在这里我们用 `NULL` 来表示这里没有被赋值. 接下来进行入栈操作, 我们知道当前 $n = 4, i = 1$, 在 $i \leq 4$ 的条件下进行循环来实现将数组中的元素按顺序压入栈中:

```
1 | // 入栈  
2 | 1: S = (d),      A = [NULL, d, c, b, a]  
3 | 2: S = (d, c),   A = [NULL, d, c, b, a]  
4 | 3: S = (d, c, b), A = [NULL, d, c, b, a]  
5 | 4: S = (d, c, b, a), A = [NULL, d, c, b, a]
```

不难发现该算法的功能实际上就是反转一个栈中的元素.

```
1 | Status algo2(Stack S, int e) {
```

```
2   Stack T; int d;  
3   initStack(T);  
4   while (!StackEmpty(S)) {  
5       Pop(S, d);  
6       if ( d != e ) Push(T, d);  
7   }  
8   while (!StackEmpty(T)) {  
9       Pop(T, d);  
10      Push(S, d);  
11  }  
12 }
```

评注. 这个算法首先初始化了一个空栈 T , 然后通过循环将 S 所有的元素全部出栈并将不等于 e 的出栈元素压入 T 中. 然后又从 T 中将所有的元素出栈并压入 S 中. 注意到由于栈 ADT 的后进先出原则, 因此每次的全部出栈 (all pop) 操作都会导致元素顺序的完全颠倒, 根据组合数学的知识知道, 一个排列的完全逆排列的完全逆排列就是这个排列本身, 因此 all pop 两次则将恢复原来的顺序, 因此栈 S 在该算法执行前后唯一的不同就是其中所有的元素 e 都被删除了. 因此该算法的功能是删除栈中的所有等于 e 的元素.

参考文献

- [1] [美] Mark Allen Weiss. 数据结构与算法分析——C 语言描述 (原书第 2 版) (典藏版) = Data Structures and Algorithm Analysis in C, Second Edition [M]. 冯舜玺译. 北京: 机械工业出版社, 2019.
- [2] [美] Michael Goodrich, [美] Roberto Tamassia, [美] Michael H. Goldwasser. 数据结构与算法: Python 语言实现 = Data Structures and Algorithms in Python [M]. 张晓等译. 北京: 机械工业出版社, 2018.
- [3] [美] Thomas H. Cormen, [美] Charles E. Leiserson, [美] Ronald L. Rivest, [美] Clifford Stein. 算法导论 (原书第 3 版) = Introduction to Algorithms, Third Edition [M]. 殷建平等译. 北京: 机械工业出版社, 2013.
- [4] 耿国华等. 数据结构: 用 C 语言描述 (第 2 版) [M]. 北京: 高等教育出版社, 2015.
- [5] 王道论坛组. 2024 年数据结构考研复习指导 [M]. 北京: 电子工业出版社, 2022.
- [6] 严蔚敏, 李冬梅, 吴伟民. 数据结构: C 语言版: 双色版: 第 2 版 [M]. 北京: 人民邮电出版社, 2022.
- [7] 李冬梅, 田紫薇. 数据结构习题解析与实验指导: 第 2 版 [M]. 北京: 人民邮电出版社, 2022.

I ♥ NPU

公诚勇毅 永矢毋忘
中华灿烂 工大无疆

本文档由**钱锋**编写, 钱锋保留一切权利.

文档中出现的部分素材来源于网络, 笔者承诺这些素材仅供学习交流之用, 它们的原作者保留一切权利.

2023 年 西北工业大学 中国西安