

数据结构作业 第三周

霍斌 PB24111627

1 2.25

输入list1和list2, 返回一个新链表newList, 完整头文件见附录.

```
1  LinkedList mergeListsByIncrease(LinkedList *list1, LinkedList *list2) {
2      LinkedList newList;
3      initList(&newList);
4      // 仍假设两个链表均为递增有序排列, 返回一个新链表
5      if (!list1 || !list2 || !list1->head || !list2->head) return newList;
6      if (list1->head->data == 0 && list2->head->data == 0) return newList; // 两个链表均
    为空
7
8      // 获取两个链表的第一个节点
9      Node *cur1 = list1->head->next;
10     Node *cur2 = list2->head->next;
11
12     while(cur1 != NULL && cur2 != NULL) {
13         if(cur1->data > cur2->data) {
14             cur2 = cur2->next;
15         } else if(cur1->data < cur2->data) {
16             cur1 = cur1->next;
17         } else {
18             addToEnd(&newList, cur1->data);
19             cur1 = cur1->next;
20             cur2 = cur2->next;
21         }
22     }
23
24     return newList;
25 }
```

2 2.38

LocateNode函数会搜索第一个匹配的节点, 并按照频率排序. 完整头文件见附录.

```
1  Node *LocateNode(Node* head, int data) {
2      Node *cur = head->next;
3      while (cur != head) {
4          if (cur->data == data) {
5              cur->freq++; // 更新访问频率
6
7              // 每访问一次, 按频率对链表进行排序
8              while(cur->prev != head && cur->freq > cur->prev->freq) {
9                  // 交换一次节点(冒泡排序)
10                 Node *prevNode = cur->prev;
11                 Node *nextNode = cur->next;
12
13                 nextNode->prev = prevNode;
14                 prevNode->next = nextNode;
15                 cur->prev = prevNode->prev;
```

```

16         cur->next = prevNode;
17         prevNode->prev->next = cur;
18         prevNode->prev = cur;
19     }
20     return cur;
21 }
22 cur = cur->next;
23 }
24 return NULL;
25 }

```

3 3.6

根据栈后进先出的特性,不可能出现输入序列后进的数比先进的数先出栈的情况,所以不存在这样的 $i < j < k$.

4 3.19

本函数用于判断是否匹配,完整头文件见附录.

```

1  // 提取出一个字符串的括号部分, 并判断括号是否匹配
2  void matchPairs(Stack *stack, char* str) {
3      if(!isEmpty(stack)) {
4          perror("Stack is not empty, cannot deal.");
5          return;
6      }
7
8      // 遇到() [] {} 就入栈, 如果匹配到了对应的反括号则Pop, 如果括号是匹配的, 那么最内层会
      // 被消掉, 进一步的外层括号也会逐层被消掉
9      for(char* p = str; *p != '\0'; p++) {
10         if(*p == '(' || *p == '[' || *p == '{') {
11             push(stack, *p);
12         } else if(*p == ')') {
13             if(!isEmpty(stack) && peek(stack) == '(') {
14                 pop(stack);
15             } else {
16                 printf("失败啦! ");
17                 return;
18             }
19         } else if(*p == ']') {
20             if(!isEmpty(stack) && peek(stack) == '[') {
21                 pop(stack);
22             } else {
23                 printf("失败啦! ");
24                 return;
25             }
26         } else if(*p == '}') {
27             if(!isEmpty(stack) && peek(stack) == '{') {
28                 pop(stack);
29             } else {
30                 printf("失败啦! ");
31                 return;
32             }
33         }
34     }
35
36     // 如果栈为空, 则说明匹配成功

```

```

37     if(isEmpty(stack)) {
38         printf("成功啦! ");
39         return;
40     } else {
41         printf("失败啦! ");
42     }
43 }

```

5 3.27(1)

5.1 算法

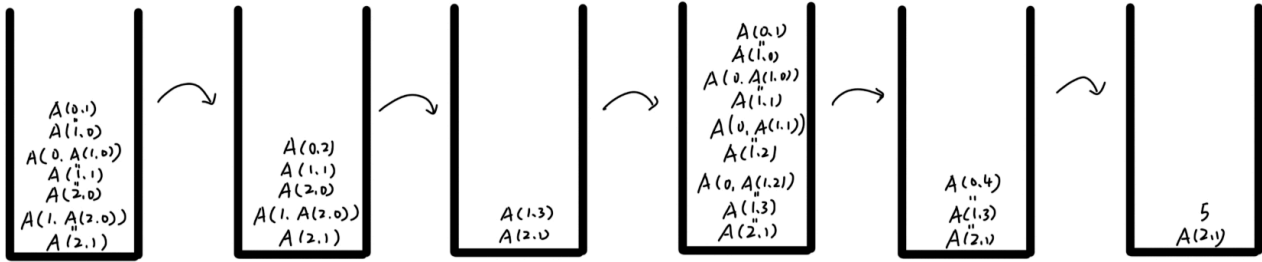
分别实现akm函数的递归和非递归(栈)实现, akm函数使用了IntStack.h, 完整头文件见附录.

```

1  // 递归算法
2  int akm_recursion(int m, int n) {
3      if(m == 0) {
4          return n + 1;
5      } else if(m != 0 && n == 0) {
6          return akm_recursion(m - 1, 1);
7      } else if(m != 0 && n != 0) {
8          return akm_recursion(m - 1, akm_recursion(m, n - 1));
9      }
10
11     return -1;
12 }
13
14 // 非递归算法
15 int akm_human(int m, int n) {
16     IntStack* s = createIntStack();
17     push(s, m);
18
19     while(!isEmpty(s)) {
20         m = peek(s);
21         pop(s);
22
23         if (m == 0) {
24             n = n + 1;
25         } else if (n == 0) {
26             push(s, m - 1);
27             n = 1;
28         } else {
29             push(s, m - 1);
30             push(s, m);
31             n = n - 1;
32         }
33     }
34     return n;
35 }

```

5.2 绘图



如图所示, 需要依次计算从栈顶到栈底的Akm函数, 直到栈清空, 最后会得到 $A(2,1) = 5$.

附录

1 LinearList

```
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;
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         return;
24     }
25     list->head->data = 0;
26     list->head->next = NULL;
27 }
28
29 int getSize(const LinearList *list) {
30     if (!list || !list->head) return 0;
31     return list->head->data;
32 }
33
34 bool isValidIndex(const LinearList *list, int index, bool allowEnd) {
35     if (!list || !list->head) return false;
36     int size = list->head->data;
37     if (allowEnd) {
38         return index >= 0 && index <= size;
39     } else {
```

```

40         return index >= 0 && index < size;
41     }
42 }
43
44 // 在指定位置插入元素
45 bool insert(LinearList *list, int index, int value) {
46     if (!list || !list->head) {
47         return false;
48     }
49
50     if (!isValidIndex(list, index, true)) {
51         return false;
52     }
53
54     Node *newNode = (Node *)malloc(sizeof(Node));
55     if (!newNode) {
56         return false;
57     }
58
59     newNode->data = value;
60     Node *curNode = list->head;
61
62
63     for (int i = 0; i < index; i++) {
64         curNode = curNode->next;
65     }
66
67     newNode->next = curNode->next;
68     curNode->next = newNode;
69     list->head->data++;
70
71     return true;
72 }
73
74
75 bool extract(LinearList *list, int index) {
76     if (!list || !list->head) {
77         return false;
78     }
79
80     if (!isValidIndex(list, index, false)) {
81         return false;
82     }
83
84     Node *curNode = list->head;
85
86     for (int i = 0; i < index; i++) {
87         curNode = curNode->next;
88     }
89
90     Node *nodeToDelete = curNode->next;
91     curNode->next = nodeToDelete->next;
92     free(nodeToDelete);
93     list->head->data--;
94
95     return true;

```

```

96 }
97
98 // 获取指定位置的元素
99 int get(const LinearList *list, int index) {
100     if (!list || !list->head) {
101         return -1;
102     }
103
104     if (!isValidIndex(list, index, false)) {
105         return -1;
106     }
107
108     Node *curNode = list->head->next;
109
110     for (int i = 0; i < index; i++) {
111         curNode = curNode->next;
112     }
113
114     return curNode->data;
115 }
116
117 // 打印链表
118 void printList(const LinearList *list) {
119     if (!list || !list->head) {
120         return;
121     }
122
123     printf("List (size: %d): ", list->head->data);
124     Node *curNode = list->head->next;
125
126     while (curNode != NULL) {
127         printf("%d", curNode->data);
128         if (curNode->next != NULL) {
129             printf(" -> ");
130         }
131         curNode = curNode->next;
132     }
133     printf("\n");
134 }
135
136 bool addToEnd(LinearList *list, int value) {
137     if (!list || !list->head) return false;
138     return insert(list, list->head->data, value);
139 }
140
141 bool addToFront(LinearList *list, int value) {
142     return insert(list, 0, value);
143 }
144
145 // 删除最小值到最大值之间的所有元素(不包括端点)
146 bool deleteFromMin2Max(LinearList *list, int mink, int maxk) {
147     // 已知链表中元素以值递增有序排列
148     if (list->head->data == 0 ||
149         maxk < mink ||
150         (list->head->next != NULL && list->head->next->data >= maxk))
151     )

```

```

152         return false; // 处理异常情况
153     Node *leftGapNode = list->head->next;
154
155     while ( leftGapNode->next != NULL && leftGapNode->next->data <= min ) {
156         leftGapNode = leftGapNode->next;
157     }
158     // 如果一直到链表末端仍然没有人大于min, 那么也没了
159     if (leftGapNode->next == NULL) {
160         return false;
161     }
162
163     Node *rightGapNode = leftGapNode->next;
164
165     while (rightGapNode != NULL && rightGapNode->data < max) {
166         rightGapNode = rightGapNode->next;
167     }
168
169     // 删除leftGapNode和rightGapNode之间的所有节点
170     Node *curNode = leftGapNode->next;
171     while (curNode != rightGapNode) {
172         Node *nodeToDelete = curNode;
173         curNode = curNode->next;
174         free(nodeToDelete);
175         list->head->data--;
176     }
177
178     // 最后弥合东西分裂
179     leftGapNode->next = rightGapNode;
180     return true;
181 }
182
183 bool reverse(LinearList *list) {
184     if ( list->head->data <= 1 ) return false;
185
186     Node *prev = NULL;
187     Node *current = list->head->next; // 跳过头结点
188     Node *next = NULL;
189
190     while (current != NULL) {
191         next = current->next; // 保存下一个节点
192         current->next = prev; // 反转当前节点的指针
193         prev = current;      // 移动prev和current指针
194         current = next;
195     }
196
197     list->head->next = prev; // 更新头结点的next指针
198     return true;
199 }
200
201 bool mergeListsByDecrease(LinearList *list1, LinearList *list2) {
202     // 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
203     if (!list1 || !list2 || !list1->head || !list2->head) return false;
204     if (list1->head->data == 0 && list2->head->data == 0) return false; // 两个链表均
205     为空
206

```

```

207 Node *dummy = (Node *)malloc(sizeof(Node));
208 dummy->next = NULL;
209 Node *tail = dummy;
210 Node *cur1 = list1->head->next;
211 Node *cur2 = list2->head->next;
212
213 while (cur1 && cur2) {
214     if (cur1->data <= cur2->data) {
215         tail->next = cur1;
216         cur1 = cur1->next;
217     } else {
218         tail->next = cur2;
219         cur2 = cur2->next;
220     }
221     tail = tail->next;
222 }
223 if (cur1) tail->next = cur1;
224 if (cur2) tail->next = cur2;
225
226 // 更新list1的头结点
227 list1->head->next = dummy->next;
228 list1->head->data = list1->head->data + list2->head->data;
229
230 // 释放dummy节点和list2的头结点及结构体
231 free(dummy);
232 // free(list2->head);
233 // free(list2);
234 reverse(list1);
235 return true;
236 }
237
238 LinearList mergeListsByIncrease(LinearList *list1, LinearList *list2) {
239     LinearList newList;
240     initList(&newList);
241     // 仍假设两个链表均为递增有序排列, 返回一个新链表
242     if (!list1 || !list2 || !list1->head || !list2->head) return newList;
243     if (list1->head->data == 0 && list2->head->data == 0) return newList; // 两个链表
244     均为空
245
246     // 获取两个链表的第一个节点
247     Node *cur1 = list1->head->next;
248     Node *cur2 = list2->head->next;
249
250     while (cur1 != NULL && cur2 != NULL) {
251         if (cur1->data > cur2->data) {
252             cur2 = cur2->next;
253         } else if (cur1->data < cur2->data) {
254             cur1 = cur1->next;
255         } else {
256             addToEnd(&newList, cur1->data);
257             cur1 = cur1->next;
258             cur2 = cur2->next;
259         }
260     }
261
262     return newList;

```



```

262 }
263
264 bool deleteFrom1WhoBothAppearIn2and3(LinearList *list1, LinearList *list2, LinearList
    *list3) {
265     // 从表1中删去表2和表3中共有的值
266     // 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
267     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
268
269     Node *prev = list1->head;
270     Node *cur1 = list1->head->next;
271     Node *next = NULL;
272     Node *cur2 = list2->head->next;
273     Node *cur3 = list3->head->next;
274
275
276     int duplicate = 0;
277
278     while (cur2 && cur3) {
279         if (cur2->data < cur3->data) {
280             cur2 = cur2->next;
281         } else if (cur2->data > cur3->data) {
282             cur3 = cur3->next;
283         } else { // 如果找到重复值了，去往表1中删除对应元素
284             duplicate = cur2->data;
285             while(cur1 != NULL && cur1->data < duplicate) {
286                 prev = cur1;
287                 cur1 = cur1->next;
288             }
289             // 找到了重复值
290             if (cur1->data == duplicate) { // 如果发现了重复值，那么删掉这个人
291                 next = cur1->next;
292                 free(cur1);
293                 cur1 = next;
294                 prev->next = cur1;
295                 list1->head->data--;
296             }
297             // 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
298             if(cur1->data > duplicate) cur2 = cur2->next;
299         }
300     }
301
302     return true;
303 }
304
305
306 #endif // LINEARLIST_H
307

```

2 DoublyCircularLinkedList

```

1 #ifndef DOUBLY_CIRCULAR_LINKED_LIST_H
2 #define DOUBLY_CIRCULAR_LINKED_LIST_H
3
4 #include <stdio.h>
5 #include <stdlib.h>

```

```

6  #include <stdbool.h>
7
8  typedef struct Node {
9      int data;
10     int freq; // 访问频率
11     struct Node *prev;
12     struct Node *next;
13 } Node;
14
15 // 创建链表头节点
16 Node *CreateList() {
17     Node *head = (Node *)malloc(sizeof(Node));
18     head->next = head;
19     head->prev = head;
20     head->data = 0; // 用于存储链表长度
21     head->freq = 0; // 用于存储访问频率
22     return head;
23 }
24
25 void InsertNode(Node* head, int data) {
26     Node* newNode = (Node*)malloc(sizeof(Node));
27     newNode->data = data;
28     newNode->freq = 0; // 访问频率初始化为0
29     newNode->prev = head->prev;
30     newNode->next = head;
31     head->prev->next = newNode;
32     head->prev = newNode;
33     head->data++;
34 }
35
36 Node *LocateNode(Node* head, int data) {
37     Node *cur = head->next;
38     while (cur != head) {
39         if (cur->data == data) {
40             cur->freq++; // 更新访问频率
41
42             // 每访问一次，按频率对链表进行排序
43             while(cur->prev != head && cur->freq > cur->prev->freq) {
44                 // 交换一次节点(冒泡排序)
45                 Node *prevNode = cur->prev;
46                 Node *nextNode = cur->next;
47
48                 nextNode->prev = prevNode;
49                 prevNode->next = nextNode;
50                 cur->prev = prevNode->prev;
51                 cur->next = prevNode;
52                 prevNode->prev->next = cur;
53                 prevNode->prev = cur;
54             }
55             return cur;
56         }
57         cur = cur->next;
58     }
59     return NULL;
60 }
61

```

```

62 void PrintList(Node* head) {
63     Node* cur = head->next;
64     while (cur != head) {
65         printf("Data: %d, Freq: %d\n", cur->data, cur->freq);
66         cur = cur->next;
67     }
68 }
69
70 #endif // DOUBLY_CIRCULAR_LINKED_LIST_H

```

3 CharStack(括号匹配)

```

1  #ifndef STACK_H
2  #define STACK_H
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <stdbool.h>
7
8  #define MAXSIZE 100
9  typedef struct Stack {
10     int top;
11     char data[MAXSIZE];
12 } Stack;
13
14 Stack* createStack();
15 bool isFull(Stack *stack);
16 bool isEmpty(Stack *stack);
17 void push(Stack *stack, char value);
18 void pop(Stack *stack);
19 char peek(Stack *stack);
20
21 Stack* createStack() {
22     Stack *stack = (Stack*)malloc(sizeof(Stack));
23     stack->top = -1;
24     return stack;
25 }
26
27 bool isFull(Stack *stack) {
28     return stack->top == MAXSIZE - 1;
29 }
30
31 bool isEmpty(Stack *stack) {
32     return stack->top == -1;
33 }
34
35 void push(Stack *stack, char value) {
36     if(!isFull(stack)) {
37         stack->data[stack->top + 1] = value;
38         stack->top++;
39     } else {
40         perror("StackOverflow");
41     }
42 }
43
44 void pop(Stack *stack) {

```

```

45     if(!isEmpty(stack)) {
46         stack->top--;
47     } else {
48         perror("Stack is Empty");
49     }
50 }
51
52 char peek(Stack *stack) {
53     if(!isEmpty(stack)) {
54         return stack->data[stack->top];
55     } else {
56         perror("Stack is Empty");
57         return '\0';
58     }
59 }
60
61 // 提取出一个字符串的括号部分，并判断括号是否匹配
62 void matchPairs(Stack *stack, char* str) {
63     if(!isEmpty(stack)) {
64         perror("Stack is not empty, cannot deal.");
65         return;
66     }
67
68     // 遇到() [] {} 就入栈，如果匹配到了对应的反括号则Pop，如果括号是匹配的，那么最内层会
    被消掉，进一步的外层括号也会逐层被消掉
69     for(char* p = str; *p != '\0'; p++) {
70         if(*p == '(' || *p == '[' || *p == '{') {
71             push(stack, *p);
72         } else if(*p == ')') {
73             if(!isEmpty(stack) && peek(stack) == '(') {
74                 pop(stack);
75             } else {
76                 printf("失败啦! ");
77                 return;
78             }
79         } else if(*p == ']') {
80             if(!isEmpty(stack) && peek(stack) == '[') {
81                 pop(stack);
82             } else {
83                 printf("失败啦! ");
84                 return;
85             }
86         } else if(*p == '}') {
87             if(!isEmpty(stack) && peek(stack) == '{') {
88                 pop(stack);
89             } else {
90                 printf("失败啦! ");
91                 return;
92             }
93         }
94     }
95
96     // 如果栈为空，则说明匹配成功
97     if(isEmpty(stack)) {
98         printf("成功啦! ");
99         return;

```

```

100     } else {
101         printf("失败啦! ");
102     }
103 }
104 #endif // STACK_H

```

4 IntStack(Ackerman使用了该文件)

```

1  #ifndef INTSTACK_H
2  #define INTSTACK_H
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <stdbool.h>
7
8  #define MAXSIZE 10000
9  typedef struct IntStack {
10     int top;
11     int data[MAXSIZE];
12 } IntStack;
13
14 IntStack* createIntStack();
15 bool isFull(IntStack *stack);
16 bool isEmpty(IntStack *stack);
17 void push(IntStack *stack, int value);
18 void pop(IntStack *stack);
19 int peek(IntStack *stack);
20
21 IntStack* createIntStack() {
22     IntStack *stack = (IntStack*)malloc(sizeof(IntStack));
23     stack->top = -1;
24     return stack;
25 }
26
27 bool isFull(IntStack *stack) {
28     return stack->top == MAXSIZE - 1;
29 }
30
31 bool isEmpty(IntStack *stack) {
32     return stack->top == -1;
33 }
34
35 void push(IntStack *stack, int value) {
36     if(!isFull(stack)) {
37         stack->data[stack->top + 1] = value;
38         stack->top++;
39     } else {
40         perror("StackOverflow");
41     }
42 }
43
44 void pop(IntStack *stack) {
45     if(!isEmpty(stack)) {
46         stack->top--;
47     } else {
48         perror("Stack is Empty");

```

```

49     }
50 }
51
52 int peek(IntStack *stack) {
53     if(!isEmpty(stack)) {
54         return stack->data[stack->top];
55     } else {
56         perror("Stack is Empty");
57         return '\0';
58     }
59 }
60
61 #endif // STACK_H

```

5 Ackerman(递归&非递归)

```

1  #ifndef ACKERMAN_H
2  #define ACKERMAN_H
3
4  #include "IntStack.h"
5  #include <stdio.h>
6
7  // 递归法
8  int akm_recursion(int m, int n) {
9      if(m == 0) {
10         return n + 1;
11     } else if(m != 0 && n == 0) {
12         return akm_recursion(m - 1, 1);
13     } else if(m != 0 && n != 0) {
14         return akm_recursion(m - 1, akm_recursion(m, n - 1));
15     }
16
17     return -1;
18 }
19
20 // 非递归(用栈模拟)
21 int akm_human(int m, int n) {
22     IntStack* s = createIntStack();
23     push(s, m);
24
25     while(!isEmpty(s)) {
26         m = peek(s);
27         pop(s);
28
29         if (m == 0) {
30             n = n + 1;
31         } else if (n == 0) {
32             push(s, m - 1);
33             n = 1;
34         } else {
35             push(s, m - 1);
36             push(s, m);
37             n = n - 1;
38         }
39     }
40     return n;

```

```
41 | }  
42 |  
43 | #endif // ACKERMAN_H
```