

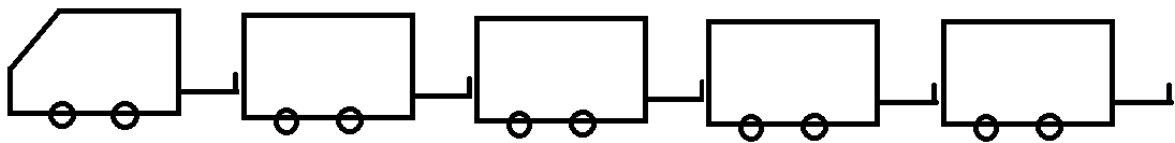
数据结构第二讲

一、链表基本知识

1.1 为啥学习链表

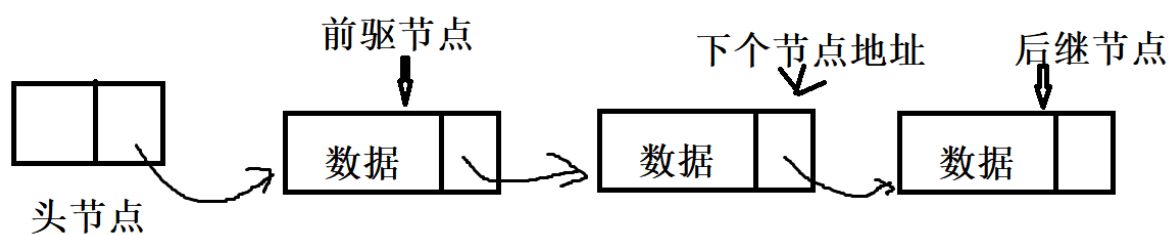
顺序表有个致命缺点：存储元素有上限，一旦到达数组的容量，就不能继续存储数据了

此时可以用链表有效解决该问题



1.2 基本概念

- 1> 链表：链式存储的线性表
- 2> 分类：单向链表、双向链表、单向循环链表、双向循环链表
- 3> 基本单位：结点--->由数据域和指针域两部分组成的基本单元
- 4> 前驱：当前结点前面的节点称为前驱节点
- 5> 后继：当前结点的后面的节点称为后继节点
- 6> 头指针：定义一个节点类型的指针，指向第一个结点的地址，可以表示整条链表
- 7> 头节点：虚设的一个结点，其指针域指向第一个结点的地址，数据域可以空着不用，也可以记录链表结点个数



二、单向链表

只能从前向后进行访问，不能从后向前访问的链表

2.1 结点结构体类型

结点中由两部分组成：数据（要存储的单个数据）和指针（记录下一个结点的地址）

```
1  typedef int datatype;
2
3  typedef struct Node
4  {
5      union
6      {
7          int len;           //存储头节点数据域，表示结点个数
8          datatype data;    //数据域，普通结点要存放的值
9      };
10     struct Node *next;     //指针域，记录下个结点的地址
11 }linkList;
12
13
```

2.2 创建链表

申请出来的结点，是一个头节点，讲数据域len赋值为0 指针域next为空

```
1  linkList *list_create()
2  {
3      linkList *L = (linkList*)malloc(sizeof(linkList));
4      if(NULL==L)
5      {
6          printf("创建失败\n");
7          return NULL;
8      }
9
10     //给节点初始化
11     L->len = 0;    //头节点数据域为0
12     L->next = NULL; //指针域为空
13
14     printf("创建成功\n");
15     return L;      //将创建好的头结点返回
16 }
```

2.3 判空

对于链表而言，无需进行判满，除非自己电脑内存不足，才会满

对于判空而言，最好使用 `L->next == NULL`，不要使用 `L->len==0` 因为在后期增删中，可能会忘记对长度的变化

也可以两个条件同时判断

```
1 int list_empty(linkList *L)
2 {
3     return L->next==NULL ? 1:0;    //1表示空  0表示非空
4 }
```

2.4 头插

- 1> 判断是否合法
- 2> 申请结点，将要插入的数据，封装到结点中
- 3> 插入逻辑：最好自己画图分析
- 4> 表长变化
- 5> 注意：头插法插入的数据是逆序的

```
1 int list_insert_head(linkList *L, datatype e)
2 {
3     //判断逻辑
4     if(NULL==L)
5     {
6         printf("所给链表不合法\n");
7         return -1;
8     }
9
10    //申请节点封装数据
11    linkList *p = (linkList*)malloc(sizeof(linkList));
12    if(NULL==p)
13    {
14        printf("空间申请失败\n");
15        return -2;
16    }
17    p->data = e;    //将数据封装到节点内
18    p->next = NULL;
19
20    //插入逻辑
21    p->next = L->next;
22    L->next = p;
23
24    //表的变化
```

```

25     L->len++;
26     printf("插入成功\n");
27     return 0;
28 }

```

2.5 遍历

定义遍历指针，从第一个结点出发，不断向后访问，只要访问的结点不是空，就输出该结点的数据域

```

1  void list_show(linkList *L)
2  {
3      //判断逻辑
4      if(NULL==L || list_empty(L))
5      {
6          printf("遍历失败\n");
7          return;
8      }
9
10     //遍历逻辑
11     linkList *q = L->next;    //定义遍历指针从第一个节点出发
12     printf("当前链表元素分别是: ");
13     while(q!=NULL)
14     {
15         printf("%d\t", q->data);    //输出当前结点的数据域
16
17         q = q->next;    //重新指向下个节点
18     }
19 }

```

2.6 尾插法

- 1> 判断表的合法性
- 2> 申请结点，将要插入的数据，封装到结点中
- 3> 插入逻辑：最好自己画图分析
- 4> 表长变化
- 5> 注意：头插法插入的数据是正序的

```

1  int list_insert_tail(linkList *L, datatype e)
2  {
3      //判断逻辑
4      if(NULL == L)
5      {
6          printf("所给链表不合法\n");

```

```

7         return -1;
8     }
9
10    //申请节点封装数据
11    linkList *p = list_create();
12    p->data = e;
13
14
15    //定义遍历指针找到最后一个节点
16    linkList *s = L;
17    while(s->next != NULL)
18    {
19        s = s->next;
20    }
21
22    //插入逻辑
23    s->next = p;
24    L->len++;
25    printf("插入成功\n");
26    return 0;
27 }
28

```

2.7 任意位置插入

- 1> 判断逻辑：表的合法性、位置合法性
- 2> 申请结点，将要插入的数据，封装到结点中
- 3> 插入逻辑：最好自己画图分析
- 4> 表长变化

```

1  int list_insert_pos(linkList *L,int pos, datatype e)
2  {
3      //判断逻辑
4      if(NULL==L || pos<1 || pos>L->len+1)
5      {
6          printf("插入失败\n");
7          return -1;
8      }
9
10     //申请节点封装数据
11     linkList *p = list_create();
12     p->data = e;
13
14     //定义遍历指针指向要插入节点的前一个节点位置

```

```

15     linkList *q = L;
16     for(int i=1; i<=pos-1; i++)
17     {
18         q = q->next;
19     }
20
21     //插入逻辑
22     p->next = q->next;
23     q->next = p;
24
25     //表的变化
26     L->len++;
27     printf("插入成功\n");
28     return 0;
29 }

```

2.8 头删法

- 1> 判断逻辑：表的合法性、是否为空
- 2> 记录要删除的结点
- 3> 孤立要删除的结点
- 4> 释放要删除的结点
- 5> 表的变化

```

1  int list_delete_head(linkList *L)
2  {
3      // 判断逻辑：表的合法性、是否为空
4      if(NULL==L || list_empty(L))
5      {
6          printf("删除失败\n");
7          return -1;
8      }
9      // 记录要删除的结点
10     linkList *p = L->next;
11
12     // 孤立要删除的结点
13     L->next = p->next;
14
15     // 释放要删除的结点
16     free(p);
17     p = NULL;
18
19     // 表的变化
20     L->len--;

```

```

21     printf("删除成功\n");
22     return 0;
23 }

```

2.9 按位置查找返回查找到的节点

- 1> 判断逻辑
- 2> 定义遍历指针从头结点出发
- 3> 循环找到要查找位置的节点
- 4> 将该节点返回

```

1  linkList *list_search_pos(linkList *L, int pos)
2  {
3      //判断逻辑
4      if(NULL==L || list_empty(L) || pos<1 || pos>L->len)
5      {
6          printf("查找失败\n");
7          return NULL;
8      }
9
10     //定义遍历指针从头结点出发
11     linkList *q = L;
12
13     //循环找到要查找位置的节点
14     for(int i=1; i<=pos; i++)
15     {
16         q = q->next;
17     }
18
19     //将该节点返回
20     return q;                //将找到的节点地址返回
21 }
22

```

2.10 任意位置删除

- 1> 判断逻辑：合法性、判断位置是否合法
- 2> 定义遍历指针，找到要删除位置的前一个结点
- 3> 保存要删除的结点
- 4> 孤立要删除的结点
- 5> 释放要删除的结点
- 6> 表的变化

```

1  int list_delete_pos(linkList *L, int pos)

```

```

2  {
3      //1>    判断逻辑：合法性、判断位置是否合法
4      if(NULL==L || pos<1 || pos>L->len)
5      {
6          printf("删除失败\n");
7          return -1;
8      }
9
10     //2>    定义遍历指针，找到要删除位置的前一个结点
11     linkList *q = list_search_pos(L, pos-1);
12
13     //3>    保存要删除的结点
14     linkList *p = q->next;
15
16     //4>    孤立要删除的结点
17     q->next = p->next; //q->next = q->next->next;
18
19     //5>    释放要删除的结点
20     free(p);
21
22     //6>    表的变化
23     L->len--;
24     printf("删除成功\n");
25     return 0;
26 }

```

2.11 按位置修改

- 1> 判断逻辑：判断合法性、判断位置释放合法
- 2> 查找到要修改的结点
- 3> 将要修改的结点值进行更新

```

1  //按位置修改
2  int list_update_pos(linkList *L, int pos, datatype new_e)
3  {
4      //1>    判断逻辑：判断合法性、判断位置释放合法
5      if(NULL==L || pos<1 || pos>L->len)
6      {
7          printf("修改失败\n");
8          return -1;
9      }
10
11     //2>    查找到要修改的结点

```



```

12     linkList *p = list_search_pos(L, pos);
13
14     //3>    将要修改的结点值进行更新
15     p->data = new_e;
16
17     printf("修改成功\n");
18     return 0;
19 }

```

2.12 按值查找

- 1> 判断逻辑：表的合法性、是否为空
- 2> 定义遍历指针，将所有结点遍历一遍
- 3> 如果某个结点的数据域，与要查找的值相等，则将该结点返回
- 4> 如果全部遍历结束，还没有找到，则返回NULL

```

1  linkList * list_search_value(linkList *L, datatype e)
2  {
3      //判断逻辑
4      if(NULL==L || list_empty(L))
5      {
6          printf("查找失败\n");
7          return NULL;
8      }
9
10     //定义遍历指针从第一个节点出发
11     linkList *q = L->next;
12     while(q!=NULL)
13     {
14         if(q->data == e)    //将要查找的数据跟链表中元素比较
15         {
16             return q;    //将查找到的节点返回
17         }
18         q = q->next;
19     }
20
21
22     printf("查找失败\n");
23     return NULL;
24     //如果没找到，返回空
25 }

```

2.13 链表销毁

1> 不能直接调用free函数将头节点释放，因为只是单纯释放头节点的空间，其余结点没有释放，造成内存泄漏

2> 可以不断调用删除函数，完成将所有普通结点全部删除，直到头节点的指针域为空

3> 最后再将头节点空间释放

```
1 void list_free(linkList *L)
2 {
3     //判断逻辑
4     if(NULL == L)
5     {
6         printf("释放失败\n");
7         return ;
8     }
9
10    //删除逻辑
11    while(L->next != NULL)
12    {
13        list_delete_head(L);    //不断调用头删，将后面节点释放
14    }
15
16    //将头结点空间释放
17    free(L);
18    L=NULL;    //防止野指针
19 }
```

三、全部代码

1> linkList.h

```
1 #ifndef __LINKLIST_H__
2 #define __LINKLIST_H__
3
4 //定义节点类型
5 typedef int datatype;
6 typedef struct Node
7 {
8     union
9     {
10         int len;    //存储头节点数据域，表示结点个数
```

```

11         datatype data;                //数据域，普通结点要存放的值
12     };
13     struct Node *next;                //指针域，记录下个结点的地址
14 }linkList;
15
16 //操作
17 //创建链表
18 linkList *list_create();
19
20 //判空
21 int list_empty(linkList *L);
22
23 //头插法
24 int list_insert_head(linkList *L, datatype e);
25
26 //遍历
27 void list_show(linkList *L);
28
29 //尾插法
30 int list_insert_tail(linkList *L, datatype e);
31
32 //任意位置插入
33 int list_insert_pos(linkList *L, int pos, datatype e);
34
35 //头删
36 int list_delete_head(linkList *L);
37
38 //按位置查找返回查找到的节点
39 linkList *list_search_pos(linkList *L, int pos);
40
41 //尾删
42 int list_delete_tail(linkList *L);
43
44 //任意删除
45 int list_delete_pos(linkList *L, int pos);
46
47 //按值修改
48 int list_update_value(linkList *L, datatype old_e, datatype
new_e);
49
50 //按位置修改
51 int list_update_pos(linkList *L, int pos, datatype new_e);
52
53 //按值查找
54 linkList* list_search_value(linkList *L, datatype e);

```

```
55
56 //销毁
57 void list_free(linkList *L);
58 #endif
```

2> linkList.c

```
1 #include<stdio.h>
2 #include<stdlib.h>
3 #include<malloc.h>
4 #include"linkList.h"
5
6 //创建链表
7 linkList *list_create()
8 {
9     linkList *L = (linkList*)malloc(sizeof(linkList));
10    if(NULL==L)
11    {
12        printf("创建失败\n");
13        return NULL;
14    }
15
16    //给节点初始化
17    L->len = 0;    //头结点数据域为0
18    L->next = NULL;    //指针域为空
19
20    printf("创建成功\n");
21    return L;    //将创建好的头结点返回
22 }
23
24 //判空
25 int list_empty(linkList *L)
26 {
27     return L->next==NULL ? 1:0;    //1表示空 0表示非空
28 }
29
30 //头插法
31 int list_insert_head(linkList *L, datatype e)
32 {
33     //判断逻辑
34     if(NULL==L)
35     {
36         printf("所给链表不合法\n");
37         return -1;
38     }
```

```

39
40 //申请节点封装数据
41 linkList *p = (linkList*)malloc(sizeof(linkList));
42 if(NULL==p)
43 {
44     printf("空间申请失败\n");
45     return -2;
46 }
47 p->data = e; //将数据封装到节点内
48 p->next = NULL;
49
50 //插入逻辑
51 p->next = L->next;
52 L->next = p;
53
54 //表的变化
55 L->len++;
56 printf("插入成功\n");
57 return 0;
58 }
59
60 //遍历
61 void list_show(linkList *L)
62 {
63     //判断逻辑
64     if(NULL==L || list_empty(L))
65     {
66         printf("遍历失败\n");
67         return;
68     }
69
70     //遍历逻辑
71     linkList *q = L->next; //定义遍历指针从第一个节点出发
72     printf("当前链表元素分别是: ");
73     while(q!=NULL)
74     {
75         printf("%d\t", q->data); //输出当前结点的数据域
76
77         q = q->next; //重新指向下个节点
78     }
79     printf("\n");
80 }
81
82 //尾插法
83 int list_insert_tail(linkList *L, datatype e)

```

```

84  {
85      //判断逻辑
86      if(NULL == L)
87      {
88          printf("所给链表不合法\n");
89          return -1;
90      }
91
92      //申请节点封装数据
93      linkList *p = list_create();
94      p->data = e;
95
96
97      //定义遍历指针找到最后一个节点
98      linkList *s = L;
99      while(s->next != NULL)
100     {
101         s = s->next;
102     }
103
104     //插入逻辑
105     s->next = p;
106
107     //表的变化
108     L->len++;
109     printf("插入成功\n");
110     return 0;
111 }
112
113 //任意位置插入
114 int list_insert_pos(linkList *L, int pos, datatype e)
115 {
116     //判断逻辑
117     if(NULL==L || pos<1 || pos>L->len+1)
118     {
119         printf("插入失败\n");
120         return -1;
121     }
122
123     //申请节点封装数据
124     linkList *p = list_create();
125     p->data = e;
126
127     //定义遍历指针指向要插入节点的前一个节点位置
128     linkList *q = L;

```

```

129     for(int i=1; i<=pos-1; i++)
130     {
131         q = q->next;
132     }
133
134     //插入逻辑
135     p->next = q->next;
136     q->next = p;
137
138     //表的变化
139     L->len++;
140     printf("插入成功\n");
141     return 0;
142 }
143
144 //头删
145 int list_delete_head(linkList *L)
146 {
147     // 判断逻辑: 表的合法性、是否为空
148     if(NULL==L || list_empty(L))
149     {
150         printf("删除失败\n");
151         return -1;
152     }
153     // 记录要删除的结点
154     linkList *p = L->next;
155
156     // 孤立要删除的结点
157     L->next = p->next;
158
159     // 释放要删除的结点
160     free(p);
161     p = NULL;
162
163     // 表的变化
164     L->len--;
165     printf("删除成功\n");
166     return 0;
167 }
168
169 //按位置查找返回查找到的节点
170 linkList *list_search_pos(linkList *L, int pos)
171 {
172     //判断逻辑
173     if(NULL==L || list_empty(L) || pos<0 || pos>L->len)

```

```

174     {
175         printf("查找失败\n");
176         return NULL;
177     }
178
179     //定义遍历指针从头结点出发
180     linkList *q = L;
181
182     //循环找到要查找位置的节点
183     for(int i=1; i<=pos; i++)
184     {
185         q = q->next;
186     }
187
188     //将该节点返回
189     return q;                //将找到的节点地址返回
190 }
191
192 //尾删
193 int list_delete_tail(linkList *L);
194
195 //任意删除
196 int list_delete_pos(linkList *L, int pos)
197 {
198     //1>    判断逻辑：合法性、判断位置是否合法
199     if(NULL==L || pos<1 || pos>L->len)
200     {
201         printf("删除失败\n");
202         return -1;
203     }
204
205     //2>    定义遍历指针，找到要删除位置的前一个结点
206     linkList *q = list_search_pos(L, pos-1);
207
208     //3>    保存要删除的结点
209     linkList *p = q->next;
210
211     //4>    孤立要删除的结点
212     q->next = p->next; //q->next = q->next->next;
213
214     //5>    释放要删除的结点
215     free(p);
216
217     //6>    表的变化
218     L->len--;

```



```

219     printf("删除成功\n");
220     return 0;
221 }
222
223 //按值修改
224 int list_update_value(linkList *L, datatype old_e, datatype
new_e);
225
226 //按位置修改
227 int list_update_pos(linkList *L, int pos, datatype new_e)
228 {
229     //1>    判断逻辑：判断合法性、判断位置释放合法
230     if(NULL==L || pos<1 || pos>L->len)
231     {
232         printf("修改失败\n");
233         return -1;
234     }
235
236     //2>    查找到要修改的结点
237     linkList *p = list_search_pos(L, pos);
238
239     //3>    将要修改的结点值进行更新
240     p->data = new_e;
241
242     printf("修改成功\n");
243     return 0;
244 }
245
246 //按值查找返回查找到的节点
247 linkList * list_search_value(linkList *L, datatype e)
248 {
249     //判断逻辑
250     if(NULL==L || list_empty(L))
251     {
252         printf("查找失败\n");
253         return NULL;
254     }
255
256     //定义遍历指针从第一个节点出发
257     linkList *q = L->next;
258     while(q!=NULL)
259     {
260         if(q->data == e)    //将要查找的数据跟链表中元素比较
261         {
262             return q;    //将查找到的节点返回

```

```

263     }
264     q = q->next;
265 }
266
267
268     printf("查找失败\n");
269     return NULL;
270     //如果没找到, 返回空
271 }
272
273
274 //销毁
275 void list_free(linkList *L)
276 {
277     //判断逻辑
278     if(NULL == L)
279     {
280         printf("释放失败\n");
281         return ;
282     }
283
284     //删除逻辑
285     while(L->next != NULL)
286     {
287         list_delete_head(L);    //不断调用头删, 将后面节点释放
288     }
289
290     //将头结点空间释放
291     free(L);
292     L=NULL;    //防止野指针
293 }

```

3> main.c

```

1  #include"linkList.h"
2  #include<stdio.h>
3  int main(int argc, const char *argv[])
4  {
5      linkList *L = list_create();
6      if(NULL == L)
7      {
8          return -1;
9      }
10
11     //调用头插函数

```

```
12     list_insert_head(L, 6);
13     list_insert_head(L, 2);
14     list_insert_head(L, 3);
15     list_insert_head(L, 5);
16
17     //调用遍历函数
18     list_show(L);
19
20     //调用尾插函数
21     list_insert_tail(L, 1);
22     list_insert_tail(L, 9);
23     list_show(L);
24
25     //调用任意位置插入函数
26     list_insert_pos(L, 3, 9);
27     list_show(L);
28
29     //调用头删函数
30     list_delete_head(L);
31     list_delete_head(L);
32     list_show(L);
33
34     //调用任意位置删除函数
35     list_delete_pos(L, 3);
36     list_show(L);
37
38     //调用任意位置修改函数
39     list_update_pos(L, 1, 7);
40     list_show(L);
41
42     //调用按值查找函数
43     linkList *res = list_search_value(L, 5);
44     if(res != NULL)
45     {
46         printf("您要找的节点在链表中\n");
47     }
48
49     //调用释放函数
50     list_free(L);
51     L = NULL;
52     list_show(L);
53
54     return 0;
55 }
```

作业

作业1：实现链表尾删函数、按值进行修改函数、按值查找并返回第一次出现的位置函数

作业2：实现链表的反转：例如原链表数据为：1 2 3 4 5 调用反转函数后，该链表数据为：5 4 3 2 1

作业3：将今天学习内容总结到思维导图上