

```

1  /* SList.c-单向非循环链表 */
2  #include <stdlib.h>
3  #include <malloc.h>
4  #include <assert.h>
5  #include <string.h>
6  #include "SList.h"
7
8  //链表的初始化
9  void ListNew(LIST *l, int keySize, ListCmp *cmpFn, ListFree *freeFn)
10 {
11     assert(keySize > 0);
12     l->keySize = keySize;
13     l->cmpFn = cmpFn;
14     l->freeFn = freeFn;
15     l->head.next = NULL;
16     l->size = 0;
17 }
18
19 //获取链表的节点数量
20 int ListSize(LIST *l)
21 {
22     return l->size;
23 }
24
25 //链表判空
26 int ListEmpty(LIST *l)
27 {
28     return (0 == l->size);
29 }
30
31 //链表的销毁
32 void ListDispose(LIST *l)
33 {
34     if (ListEmpty(l))
35     {
36         return ;
37     }
38     LISTNODE *cur, *post;
39     for (cur = l->head.next; NULL != cur; cur = post)
40     {
41         post = cur->next;
42         if (NULL != l->freeFn)
43         {
44             l->freeFn(cur->key);
45         }
46         free(cur);
47     }
48     //头节点不需要手动释放内存
49     l->head.next = NULL;
50     l->size = 0;
51 }
52
53 //根据关键码查找所在节点中的数据地址
54 void *ListSearch(LIST *l, const void *e)
55 {
56     if (NULL == l->cmpFn)
57     {
58         return NULL;
59     }
60     LISTNODE *cur = l->head.next;
61     while (NULL != cur)
62     {
63         if (0 == l->cmpFn(cur->key, e))
64         {
65             return cur->key;
66         }
67         cur = cur->next;
68     }
69     return NULL;
70 }
71
72 //链表关键码的插入, mode: 0--头插, 0!--尾插
73 //返回值: 0--成功, !0--失败

```

```

74 int ListInsert(LIST *l, const void *e, int mode)
75 {
76     LISTNODE *newNode = (LISTNODE *)malloc(sizeof(*newNode) + l->keySize);
77     if (NULL == newNode)
78     {
79         return -1;
80     }
81     if (LIST_FORWARD == mode) //头插
82     {
83         newNode->next = l->head.next;
84         l->head.next = newNode;
85     }
86     else //尾插
87     {
88         newNode->next = NULL;
89         LISTNODE *cur, *pre;
90         for (cur = &(l->head); NULL != cur; cur = cur->next)
91         {
92             pre = cur;
93         }
94         pre->next = newNode;
95     }
96     memcpy(newNode->key, e, l->keySize);
97     l->size ++;
98     return 0;
99 }
100
101 static LISTNODE *ListNodeSearchPrev(LIST *l, const void *e)
102 {
103     LISTNODE *cur, *post;
104     for (cur = &l->head; NULL != cur; cur = post)
105     {
106         post = cur->next;
107         if ((NULL != post) && (0 == l->cmpFn(post->key, e)))
108         {
109             break;
110         }
111     }
112     return cur;
113 }
114
115 //返回值: 0--成功, !0--失败
116 static int listRemoveAt(LIST *l, void *e, ListFree *freeFn)
117 {
118     if (NULL == l->cmpFn)
119     {
120         return -1;
121     }
122     LISTNODE *nodePrev = ListNodeSearchPrev(l, e);
123     if (NULL == nodePrev)
124     {
125         return -1;
126     }
127     LISTNODE *nodeRemove = nodePrev->next;
128     if (NULL == nodeRemove)
129     {
130         return -1;
131     }
132     nodePrev->next = nodeRemove->next;
133     if (NULL != freeFn)
134     {
135         freeFn(nodeRemove->key);
136     }
137     free(nodeRemove);
138     l->size --;
139     return 0;
140 }
141
142 //链表删除关键码所在节点, 返回值: 0--成功, !0--失败
143 int ListRemove(LIST *l, void *e)
144 {
145     return listRemoveAt(l, e, l->freeFn);
146 }

```

```
147
148 //链表删除关键码所在节点（无需深度删除关键码），返回值：0--成功，!0--失败
149 int ListRemoveU(LIST *l, void *e)
150 {
151     return listRemoveAt(l, e, NULL);
152 }
153
154 //链表的遍历
155 void ListTraverse(LIST *l, ListTraverseOp *traverseOpFn, void *outData)
156 {
157     if (NULL == traverseOpFn || ListEmpty(l))
158     {
159         return ;
160     }
161     LISTNODE *cur;
162     for (cur = l->head.next; NULL != cur; cur = cur->next)
163     {
164         traverseOpFn(cur->key, outData);
165     }
166 }
```