```c
/* DList.c-双向循环链表 */
#include <stdlib.h>
#include <malloc.h>
#include <assert.h>
#include <string.h>
#include "DList.h"

//链表的初始化
void ListNew(LIST *l, int keySize, ListCmp *cmpFn, ListFree *freeFn)
{
    assert(keySize > 0);
    l->keySize = keySize;
    l->cmpFn = cmpFn;
    l->freeFn = freeFn;
    l->head.next =  l->head.prev = &(l->head);
    l->size = 0;
}

//获取链表的节点数量
int ListSize(LIST *l)
{
    return l->size;
}

//链表判空
int ListEmpty(LIST *l)
{
    return (0 == l->size);
}

//链表的销毁
void ListDispose(LIST *l)
{
    if (ListEmpty(l))
    {
        return ;
    }
    LISTNODE *cur, *post;
    for (cur = l->head.next; cur != &(l->head); cur = post)
    {
        post = cur->next;
        if (NULL != l->freeFn)
        {
            l->freeFn(cur->key);
        }
        free(cur);
    }
    //头节点不需要手动释放内存
    l->head.next = l->head.prev= &(l->head);
    l->size = 0;
}

//根据关键码查找所在节点中的数据地址
void *ListSearch(LIST *l, const void *e)
{
    if (NULL == l->cmpFn)
    {
        return NULL;
    }
    LISTNODE *cur = l->head.next;
    while (&(l->head) != cur)
    {
        if (0 == l->cmpFn(cur->key, e))
        {
            return cur->key;
        }
        cur = cur->next;
    }
    return NULL;
}

//链表关键码的插入，mode：0--头插，0!--尾插
//返回值：0--成功，!0--失败
```

```c
int ListInsert(LIST *l, const void *e, int mode)
{
    LISTNODE *newNode = (LISTNODE *)malloc(sizeof(*newNode) + l->keySize);
    if (NULL == newNode)
    {
        return -1;
    }
    if (LIST_FORWARD == mode) //头插
    {
        newNode->next = l->head.next;
        newNode->prev = &(l->head);
        newNode->next->prev = newNode;
        newNode->prev->next = newNode;
    }
    else //尾插
    {
        newNode->next = &(l->head);
        newNode->prev = l->head.prev;
        newNode->next->prev = newNode;
        newNode->prev->next = newNode;
    }
    memcpy(newNode->key, e, l->keySize);
    l->size ++;
    return 0;
}
//根据关键码查找所在节点
static LISTNODE *listSearchNode(LIST *l, const void *e)
{
    if (NULL == l->cmpFn)
    {
        return NULL;
    }
    LISTNODE *cur = l->head.next;
    while (&(l->head) != cur)
    {
        if (0 == l->cmpFn(cur->key, e))
        {
            return cur;
        }
        cur = cur->next;
    }
    return NULL;
}

//返回值：0--成功，!0--失败
static int listRemoveAt(LIST *l, void *e, ListFree *freeFn)
{
    if (NULL == l->cmpFn)
    {
        return -1;
    }
    LISTNODE *nodeRemove = listSearchNode(l, e);
    if (NULL == nodeRemove)
    {
        return -1;
    }
    //将待删除的节点从链中摘除
    nodeRemove->prev->next = nodeRemove->next;
    nodeRemove->next->prev = nodeRemove->prev;
    if (NULL != freeFn)
    {
        freeFn(nodeRemove->key);
    }
    free(nodeRemove);
    l->size --;
    return 0;
}

//链表删除关键码所在节点，返回值：0--成功，!0--失败
int ListRemove(LIST *l, void *e)
{
    return listRemoveAt(l, e, l->freeFn);
}
```

```
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; cur != &(l->head); cur = cur->next)
163     {
164         traverseOpFn(cur->key, outData);
165     }
166 }
```