目录

Linked lists

Uses

- used to implement high-level data structures such as stack, queues and graphs
- used by O/Ss in dynamic memory allocation

Compared to arrays

- advantages
    - memory is allocated dynamically, so can be arbitrarily large
    - arrays require contiguous space: can be difficult to handle, unlike linked lists
    - can delete nodes easily/cleanly
- disadvantages
    - cannot do random access, which arrays can
    - can only sequentially access, but arrays do this faster
    - larger overhead (e.g. a *malloc()* for every *push()*)
    - an extra pointer for every element

If performance is an issue

- *binary search trees* almost always perform better than linked lists

What is a linked list?

A linked-list is a sequential collection of items that cannot be accessed randomly

- self-referential (nodes link to nodes)
- may be cyclic

A linked list consists of

- nodes
- a 'special' node that defines the **first** node, also called the **head**
- another 'special' node that acts to end the list
    - it may be NULL
    - it may be a dummy (also called sentinal) node
    - it may be the **first** node (hence the list is circular)

A **node** is a structure that contains

- data and

- a pointer to the **next** node if it is singly linked
  - or pointers to the **previous** and **next** nodes if it is doubly linked

Wikipedia's description of a linked list

There are many ways of declaring a linked list.

1. Typically:
   - first declare a *struct* that contains *data* and a pointer to itself:

```
切換行号显示
1   typedef struct node ListNode;
2   struct node {
3      int data;
4      ListNode *next;
5   }
```

   - then declare a *struct* containing a pointer to the first node:

```
切換行号显示
1   typedef struct FirstNode *LinkedList;
2   struct FirstNode{
3       ListNode *first;
4   }
```

2. Sedgwick:
   - declares it as simply a link to a structure

```
切換行号显示
1   typedef struct node *Link;
2   struct node{
3       int data;
4       Link next;
5   }
```

3. I will use:

```
切換行号显示
1   typedef struct node {
2       int data;
3       struct node *next;
4   } List;
```

Example: a 2-node linked list

This is an illustrative example that shows basic functionality.

- it shows the simplest possible meaningful linked list being created and destroyed
- it is just a main program, no functions are used

```
切換行号显示
   1 // twoNodesList.c: create a linked list of length 2 entered
with prompts
   2 #include <stdio.h>
   3 #include <stdlib.h>
   4
   5 /**/  typedef struct node {  /**/
   6 /**/     int data;           /**/
```

```
 7 /**/     struct node *next;  /**/
 8 /**/  } List;                /**/
 9
10 int main(void) {
11    List *l;
12    List *m;
13
14    l = malloc(sizeof(List)); // get memory space for the first
node
15    m = malloc(sizeof(List)); // get memory space for the second
node
16    if (l==NULL || m==NULL){
17        fprintf(stderr, "Out of memory\n");
18        return EXIT_FAILURE;
19    }
20    printf("Enter first integer: ");
21    if (scanf("%d", &l->data) == 1) {     // l's data is
assigned
22        l->next = m;                      // l's next is
assigned
23        printf("Enter second integer: ");
24        if (scanf("%d", &m->data) == 1) { // m's data is
assigned
25            m->next = NULL;               // m's next is
assigned
26            // print the 'list'
27            printf("Element 1 is %d\n", l->data);
28            printf("Element 2 is %d\n", l->next->data);
//m->data
29        }
30    }
31    // clean up: destroy both nodes
32    free(l);  // give back the memory to the heap!
33    free(m);
34    l = NULL; // zap the pointer so it cannot be reused
35    m = NULL;
36    return EXIT_SUCCESS;
37 }
```

Notice:

- a *malloc* is used to create memory for each node on the heap
  - need a *free* to give the memory back to the heap when finished
    - if you don't, you will ***leak memory***
  - as well the node pointers need to be *nulled*
    - if you don't, you will have ***dangling pointers***

Notice as well:

- there are no loops anywhere in the program
  - as there are only 2 nodes, they can be handled individually
  - in line 28, we print *l->next->data* which is the data in node *m*
    - this is not the normal way of using links, but is allowed
      - *l->next->data* is the same as *m->data* because of line 22
- by the way: if you had at least 3 nodes, and you verify that 3 nodes exist you can operate on data using links

```
切换行号显示
1  int i = a->data + a->next->data + a->next->next->data;
```

- but it is unusual and clumsy

Example: a more general example

The previous example is very limited as there are only 2 nodes.

Here is an example of creating an arbitrarily-long linked list, printing its contents, and 'destroying' the list

- as it is arbitrarily long, we must use loops to traverse the list
  - notice in the program
    - create the list: uses a *for* loop
    - print the list: uses a *while* loop
    - free the list: uses a *while* loop

切换行号显示

```
 1 // linkedFloats.c
 2 // create and de-create a linked list of 10 floating-point
numbers
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5
 6 #define MAX 10.0
 7
 8 /**/  typedef struct node {  /**/
 9 /**/     float  ship;        /**/
10 /**/     struct node *next;  /**/
11 /**/  } List;                /**/
12
13 void print(List *start) {
14    if (start != NULL) {
15       List *p;
16       p = start;
17       while (p != NULL) {
18           printf("%.1f ", p->ship);
19           p = p->next;
20       }
21       putchar('\n');
22    }
23    return;
24 }
25
26 int main(void) {
27    List *first = NULL;     // point to first node
28    List *previous = NULL;  // point to previous node, or NULL
29    List *n;
30
31    // create a linked list of MAX float nodes
32    for (float f = 0.0; f <= MAX; f++) {
33       n = malloc(sizeof(struct node));
34       if (n == NULL) {
35           fprintf(stderr, "Out of memory\n");
36           return EXIT_FAILURE;
37       }
38       n->ship = f;        // put data in the node
39       n->next = NULL;     // assume no next (maybe last node)
40
41       if (first == NULL) { // if NULL, this is the first node
42           first = n;      // REALLY IMPORTANT TO REMEMBER FIRST
NODE          如果first没有赋值，则就给他赋值
43       }
44       else {
```

```
  45            previous->next = n; // if not first, BACKPATCH
previous
  46        }
  47        previous = n;         // remember this node for next
iteration      把n记录下来下一轮使用
  48    }
  49    print(first);
  50
  51    // un-create, i.e. free, the linked list
  52    n = first;                // start at the first node
  53    while (n != NULL) {       // as long as there are more nodes
  54       List *tmp = n->next; // remember next in tmp before
freeing
  55        free(n);              // free = put memory back on heap
  56        n = tmp;              // n is now the next node
  57    }
  58    // print(first);          // WHAT DOES THIS DO?
  59    // print(previous);       // WHAT DOES THIS DO?
  60    // print(n);              // WHAT DOES THIS DO?
  61
  62    // don't leave anything dangling
  63    first = NULL;
  64    //print(first);          // WHAT DOES THIS DO?
  65    previous = NULL;
  66    n = NULL;
  67    return EXIT_SUCCESS;
  68 }
```

Compile and execute: head是第一个点，里面存储相关数据

```
prompt$ dcc linkedFloats.c
prompt$ ./a.out
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0
```

Standard linked-list functionality

List traversal

It depends on what you want to do during the traversal.

- Example 1: equivalent to *print()* in *linkedFloat.c*

切换行号显示

```
1  void printList(List *head) {
2      List *cur;
3      for (cur = head; cur != NULL; cur = cur->next) {
4          printf("%d\n", cur->data); // a newline for every
element
5      }
6      return;
7  }
```

- Example 2: free all the nodes in a linked list

切换行号显示

```
1  void freeList(List *head) {
2      List *cur;
3      cur = head;
4      while (cur != NULL) {
5          List *tmp = cur->next; // save ptr to next node
```

想将next地址存储，然后free当前的node

```
before free the node
   6          free(cur);
   7          cur = tmp;
   8      }
   9      return;
  10  }
```

List node creation

You can put the *malloc()* into a function

切换行号显示

根据参数值进行创建节点，但是是悬浮的
因此next是NULL

```
   1 List *makeNode(int v) {
   2     List *new;
   3     new = malloc(sizeof(List));
   4     if (new == NULL) {
   5        fprintf(stderr, "Out of memory\n");
   6        exit(1);
   7     }
   8     new->data = v;
   9     new->next = NULL; // play it safe and make it NULL
  10     return new;
  11 }
```

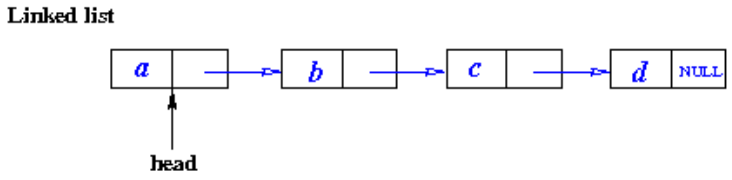
List node deletion

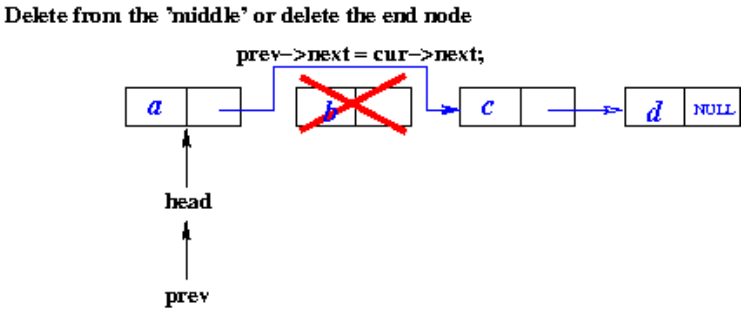
Delete a given node *n* from a linked list

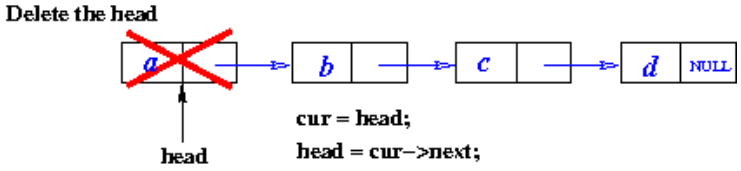
切换行号显示 需要找到 remn，然后删掉，并且释放空间

```
   1 List *deleteNode(List *head, List *remn) {
   2     // if node remn is found it is removed and freed
   3     // it is really important to make sure we do not leave the
list headless
   4     List *prev = NULL;  用于存储前一个节点，其实使用 node->next 来判断也可以
   5
   6     if (head == NULL) {                 // no list: something
wrong?
   7         return head;                    // this return is best
placed here
   8     }
   9
  10     List *cur = head;
  11     while (cur != remn && cur != NULL) { // look for remn
  12         prev = cur;          循环停止的点就是找到了 remn 的时候
  13         cur = cur->next;     此时 cur 为 remn 的地址
  14     }                        prev 为其前面节点的地址
  15     if (cur != NULL) {                  // cur must be remn
  16         if (prev == NULL) {             // if prev is NULL then
cur = head       head 节点
  17             head = cur->next;           // remove head, make
its next the head
  18         }
  19         else {                          // if cur has a prev
  20             prev->next = cur->next;     // jump over cur by
backpatching prev
  21         }
  22         free(cur);                      // either way, cur is
freed
  23         cur = NULL;
```

```
24     }
25     // if cur==NULL then remn is not in list so nothing to do
26     return head;
27 }
```







A linked-list based quack ADT

We must comply with the given ADT interface

- so we cannot use the functions *makeNode()* and *deleteNode()* above

The *Quack* interface was:

切换行号显示

```
 1 // quack.h: an interface definition for a queue/stack
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4
 5 typedef struct node *Quack;
 6
 7 Quack createQuack(void);    // create and return Quack
 8 void  push(int, Quack);     // put the given integer onto the
top of the quack
 9 void  qush(int, Quack);     // put the given integer onto the
bottom of the quack
10 int   pop(Quack);           // pop and return the top element
on the quack
11 int   isEmptyQuack(Quack);  // return 1 is Quack is empty, else
0
12 void  makeEmptyQuack(Quack);// remove all the elements on Quack
13 void  showQuack(Quack);     // print the contents of Quack,
from the top down
```

```
14
```

We implemented this interface in the ADT lecture using an array, with a maximum capacity.

We now implement it using a linked list, which has no (in-built) maximum capacity.

- each element in the *quack* is a node
- as the elements are pushed onto the quack, the list grows in length
- the length is unbounded (as an ADT)
    - *push()* cannot cause **overflow** (but a *malloc()* can fail)
    - *pop()* can still cause **underflow** of course

切换行号显示

```
 1 // quackLL.c: a linked-list-based implementation of a quack
 2 #include "quack.h"
 3 #include <limits.h>
 4
 5 #define HEAD_DATA INT_MAX // dummy data
 6
 7 struct node {
 8    int data;
 9    struct node *next;
10 };
11
12 Quack createQuack(void) { // returns a head node
13    Quack head;
14    head = malloc(sizeof(struct node));
15    if (head == NULL) {
16       fprintf (stderr, "createQuack: no memory, aborting\n");
17       exit(1);
18    }
19    head->data = HEAD_DATA; // should never be used
20    head->next = NULL;
21    return head;
22 }
23
24 void push(int data, Quack qs) {
25    Quack newnode;
26    if (qs == NULL) {
27       fprintf(stderr, "push: quack not initialised\n");
28    }
29    else {
30       newnode = malloc(sizeof(struct node));
31       if (newnode == NULL) {
32          fprintf(stderr, "push: no memory, aborting\n");
33          exit(1);
34       }
35       // insert the newnode at the head
36       newnode->data = data;       // assign the data
37       newnode->next = qs->next;   // link to 'old' linked list
38       qs->next = newnode;         // make it the head
39    }             将newnode与qs->next相连，即将newnode插入到qs头部节点的后面，
40    return;       也就是linkedlist的最前面，但是head节点是个无用节点，只用来记录地址
41 }
42
43 int pop(Quack qs) {
44    int retval = 0;
45    if (qs == NULL) {
46       fprintf(stderr, "pop: quack not initialised\n");
47    }
48    else {
```

```
 49       if (isEmptyQuack(qs)) {
 50          fprintf(stderr, "pop: quack underflow\n");
 51       }
 52       else {
 53          Quack topnode = qs->next; // top dummy node is always
there
 54          retval = topnode->data;   // grab the data
 55          qs->next = topnode->next; // remove the head
 56          free(topnode);            // clean up
 57       }
 58    }
 59    return retval;
 60 }
 61
 62 void makeEmptyQuack(Quack qs) {
 63    if (qs == NULL)
 64       fprintf(stderr, "makeEmptyQuack: quack not
initialised\n");
 65    else {
 66       while (!isEmptyQuack(qs)) {
 67          pop(qs);
 68       }
 69    }
 70    return;
 71 }
 72
 73 int isEmptyQuack(Quack qs) {
 74    int empty = 0;
 75    if (qs == NULL) {
 76       fprintf(stderr, "isEmptyQuack: quack not initialised\n");
 77    }
 78    else {
 79       empty = qs->next == NULL;
 80    }
 81    return empty;
 82 }
 83
 84 void showQuack(Quack qs) {
 85    if (qs == NULL) {
 86       fprintf(stderr, "showQuack: quack not initialised\n");
 87    }
 88    else {
 89       if (qs->data != HEAD_DATA) {
 90          fprintf(stderr, "showQuack: linked list head
corrupted\n");
 91       }
 92       else {
 93          printf("Quack: ");
 94          if (qs->next == NULL) {
 95             printf("<< >>\n");
 96          }
 97          else {
 98             printf("<<");                 // start with <<
 99             qs = qs->next;                // step over the head
link
100             while (qs->next != NULL) {
101                printf("%d, ", qs->data);  // print each element
102                qs = qs->next;
103             }
104             printf("%d>>\n", qs->data);    // last element ends
with >>
105          }
106       }
```

```
107     }
108     return;
109 }
```

Note that a *createQuack()*, which takes no arguments

- creates a special *HEAD* node of the linked list
- this node is permanent and contains 'dummy' data INT_MAX
- it cannot be deleted
- if the quack is empty, the HEAD node's *next* field is NULL
- if the quack is not empty, the HEAD node *next* field points to top node
- returns the head node to the client

If the client node is

切换行号显示

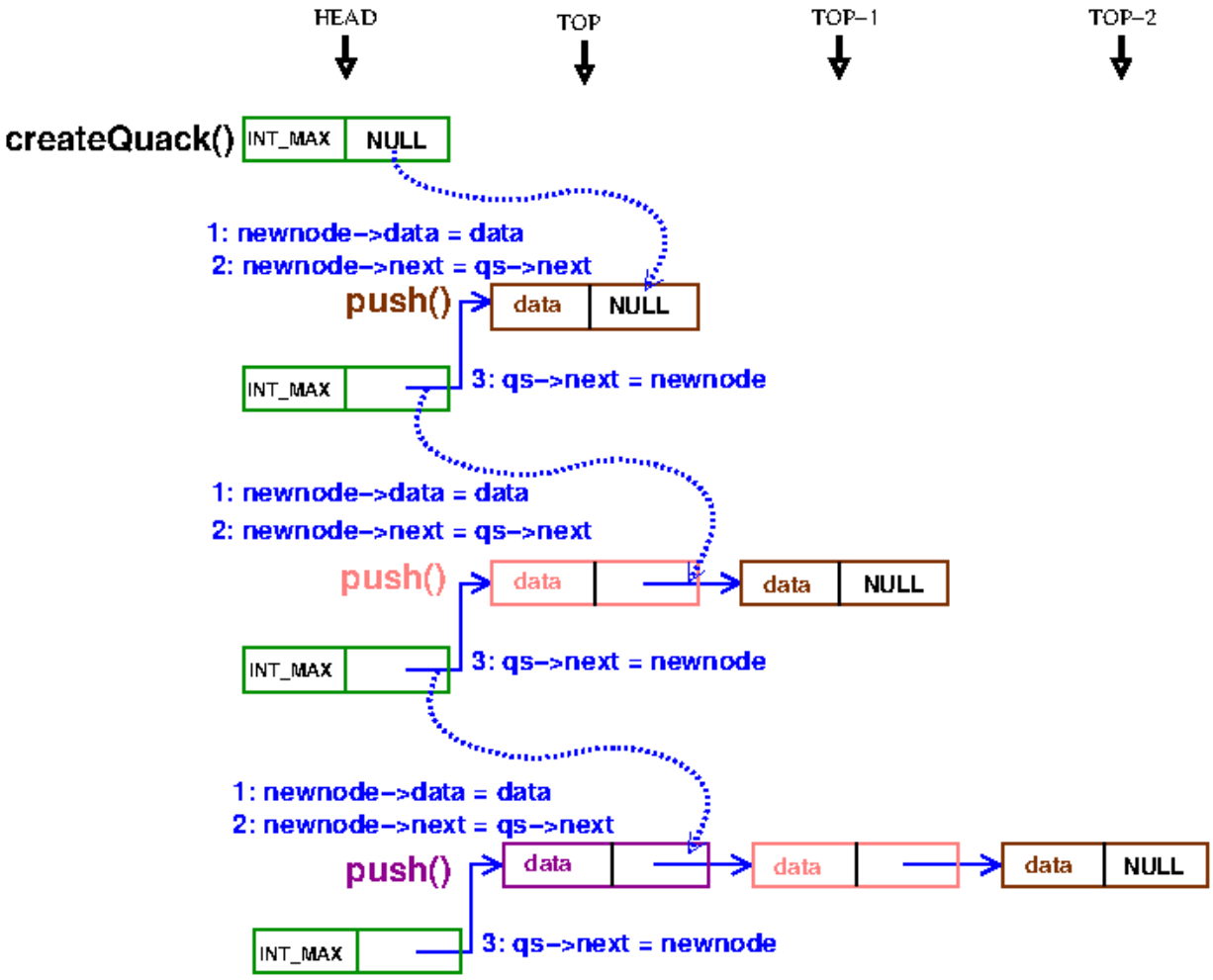
```
1 Quack qs = createQuack()
```

then every quack command uses the variable *qs* to change the quack. For example:

切换行号显示

```
1 push(123, qs);
2 int x = pop(qs);
3 makeEmptyQuack(qs);
4 showQuack(qs);
```

Let's create a new quack and push 3 data elements on it.

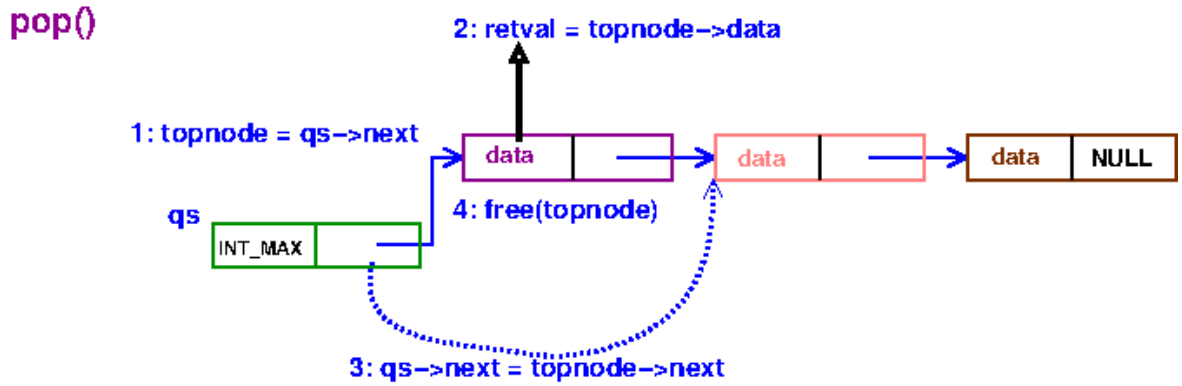


So,

- the top of quack is always the 2nd node in the linked list
- *push()* always inserts **a new 2nd node** in the linked list
- *pop()* always removes **the 2nd node** in the linked list
- *showQuack()* does not show the *HEAD* node

head只用来存储地址，不参与其他工作

Here is a pop in action.



Client: reverse a string using the linked list quack

Remember the client program that reverses the string on the command line

切换行号显示

```
   1 // revarg.c: reverse the chars in the first command-line
argument
   2 #include <stdio.h>
   3 #include <stdlib.h>
   4 #include "quack.h"
   5
   6 int main(int argc, char *argv[]) {
   7   Quack s = NULL;
   8
   9   if (argc >= 2) {
  10      char *inputc = argv[1];
  11      s = createQuack();
  12      while (*inputc != '\0') {
  13         push(*inputc++, s);
  14      }
  15      while (!isEmptyQuack(s)) {
  16         printf("%c", pop(s));
  17      }
  18      putchar('\n');
  19   }
  20   return EXIT_SUCCESS;
  21 }
```

We now have two implementations of a quack ADT

- the array version *quack.c*
- the linked-list version *quackLL.c*

Both use the same interface *quack.h* (of course)

Compile and run both with the client *revarg.c*

```
prompt$ dcc quack.c revarg.c
```

```
prompt$ ./a.out 0123456789
9876543210

prompt$ dcc quackLL.c revarg.c
prompt$ ./a.out 0123456789
9876543210
```

An ADT for linked lists

It is possible to make an ADT that lets you

- put elements on a linked list
  - insert at the head or the tail
  - much like a **push** or **qush**
- get elements from a linked list
  - take from the head or the tail
  - much like a **pop** or **qop**
- ask whether a linked list is empty
- print a linked list

This means that a *client* program:

- puts and gets data
- cannot see the nodes or pointers between the nodes

A linked list ADT interface

切换行号显示

```
 1 // LL.h
 2 // ADT interface for a linked list
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5
 6 typedef struct node *List;
 7
 8 List createList(void);    // creates and returns an empty
linked list
 9 void putHead(int, List);  // inserts data at the head of the
list
10 void putTail(int, List);  // inserts data at the tail of the
list
11 int getHead(List);        // removes and returns the head of
the list
12 int getTail(List);        // removes and returns the tail of
the list
13 int isEmptyList(List);    // 0/1 if the linked list is empty or
not
14 void showList(List);      // prints the linked list (not the
head node)
15
```

A linked-list ADT

切换行号显示

```
1 /*
2   LL.c
3   an ADT for a linked list
```

```
 */

#include "LL.h"
#include <limits.h>

struct node {
  int data;
  struct node *next;
};

List createList(void) { // creates a node, fills with INT_MAX and NULL
    List marker;
    marker = malloc(sizeof(struct node));
    if (marker == NULL) {
        fprintf (stderr, "createList: no memory, aborting\n");
        exit(1);
    }
    marker->data = INT_MAX;         // defined in <limits.h>
    marker->next = NULL;
    return marker;
}

void putTail(int n, List marker) { // add new data to the tail
    if (marker == NULL) {
        fprintf (stderr, "putTail: no linked list found\n");
    }
    else {
        List new = createList();   // re-use of createList to make a node
        new->data = n;             // overwrites INT_MAX with proper data
        List p = marker;
        while (p->next != NULL) {  // find the last node
            p = p->next;
        }
        p->next = new;             // append new to the list
    }
    return;
}

void putHead(int n, List marker) { // insert at the head
    // code not shown
}

int getTail(List marker) {         // get & delete last node
    // code not shown
    return 0; // here only to allow compilation
}

int getHead(List marker) {         // get & delete head node
    // code not shown
    return 0; // here only to allow compilation
}

int isEmptyList(List marker) {     // 0 is false, 1 is true
    int empty = 0;
    if (marker == NULL) {
        fprintf (stderr, "isEmptyList: no linked list found\n");
    }
    else {
        empty = marker->next == NULL;
    }
```

```
64    return empty;
65 }
66
67 void showList(List marker) {
68    if (marker == NULL) {
69       fprintf(stderr, "showList: no linked list found\n");
70    }
71    else {
72        printf("List: ");
73        if (marker->next == NULL) {
74           printf("<< >>\n");
75        }
76        else {
77           printf("<<");              // start with <<
78           List p = marker->next;     // get the head
79           while (p->next != NULL) {
80              printf("%d, ", p->data); // print each element
81              p = p->next;
82           }
83           printf("%d>>\n", p->data); // last element + >>
84        }
85    }
86    return;
87 }
```

The exercises to write the code for the missing functions in this ADT are in Week7Exercises

A test client for the linked-list ADT

切换行号显示

```
  1 // testLL.c: hard-coded tester for the linked-list ADT
  2 //           put testdata onto the head and tail of a linked
list
  3 //           get data from the head and tail of the linked list
  4 //           sum the data while emptying the list
  5 #include <stdio.h>
  6 #include <stdlib.h>
  7 #include "LL.h"
  8
  9 int main() {
 10     int testdata[7] = {10, 20, 30, 40, 50, 60}; // 1 extra for
the '\0'
 11     List ll = createList();
 12
 13     printf("Data is:\n\t");  // check what the test data is
 14     for (int i = 0; i < 6; i++) {
 15          printf("%d ", testdata[i]);
 16     }
 17     putchar('\n');
 18
 19     int *p = testdata;
 20     printf("Test 1. Show each putHead of testdata:\n");
 21     while (*p != '\0') {
 22         putHead(*p++, ll);
 23         putchar('\t'); showList(ll);
 24     }
 25
 26     printf("Test 2. Show 3 getTails and putHeads:\n");
 27     for (int i = 0; i < 3; i++) {
 28         putHead(getTail(ll), ll);
 29         putchar('\t'); showList(ll);
```

```
30      }
31
32      printf("Test 3. Show 3 getHeads and putTails:\n");
33      for (int i = 0; i < 3; i++) {
34          putTail(getHead(ll), ll);
35          putchar('\t'); showList(ll);
36      }
37
38      printf("Test 4. Show isEmpty working, sum from back onto
 front\n");
39      int oneleft = 0;
40      while (!oneleft) {
41          int tmp = getTail(ll) + getTail(ll);
42          if (isEmptyList(ll)) {
43              oneleft = 1;
44          }
45          putHead(tmp, ll);
46          putchar('\t'); showList(ll);
47      }
48
49      printf("Test 5. The final act, getHead the sum and check
 list isEmpty\n");
50      int sum = getHead(ll);
51      if (isEmptyList(ll)) {
52          putchar('\t'); showList(ll);
53          printf("\tSum = %d\n", sum);
54      }
55      return EXIT_SUCCESS;
56  }
```

Output is:

```
Data is:
        10 20 30 40 50 60
Test 1. Show each putHead of testdata:
        List: <<10>>
        List: <<20, 10>>
        List: <<30, 20, 10>>
        List: <<40, 30, 20, 10>>
        List: <<50, 40, 30, 20, 10>>
        List: <<60, 50, 40, 30, 20, 10>>
Test 2. Show 3 getTails and putHeads:
        List: <<10, 60, 50, 40, 30, 20>>
        List: <<20, 10, 60, 50, 40, 30>>
        List: <<30, 20, 10, 60, 50, 40>>
Test 3. Show 3 getHeads and putTails:
        List: <<20, 10, 60, 50, 40, 30>>
        List: <<10, 60, 50, 40, 30, 20>>
        List: <<60, 50, 40, 30, 20, 10>>
Test 4. Show isEmpty working, sum from back onto front
        List: <<30, 60, 50, 40, 30>>
        List: <<70, 30, 60, 50>>
        List: <<110, 70, 30>>
        List: <<100, 110>>
        List: <<210>>
Test 5. The final act, getHead the sum and check list isEmpty
        List: << >>
        Sum = 210
```

LinkedLists (2019-07-16 10:43:54由AlbertNymeyer编辑)