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:

     切换行号显示

     ```
     typedef struct node ListNode;
     struct node {
        int data;
        ListNode *next;
     }
     ```

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

     切换行号显示

     ```
     typedef struct FirstNode *LinkedList;
     struct FirstNode{
         ListNode *first;
     }
     ```

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

     切换行号显示

     ```
     typedef struct node *Link;
     struct node{
         int data;
         Link next;
     }
     ```

3. I will use:

   切换行号显示

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

切换行号显示

```
// twoNodesList.c: create a linked list of length 2 entered with prompts
#include <stdio.h>
#include <stdlib.h>

/**/  typedef struct node {   /**/
/**/      int data;           /**/
/**/      struct node *next;  /**/
/**/  } List;                 /**/

int main(void) {
   List *l;
   List *m;

   l = malloc(sizeof(List)); // get memory space for the first node
   m = malloc(sizeof(List)); // get memory space for the second node
   if (l==NULL || m==NULL){
       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

```
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

```
// linkedFloats.c
// create and de-create a linked list of 10 floating-point numbers
#include <stdio.h>
#include <stdlib.h>

#define MAX 10.0

/**/  typedef struct node {  /**/
```

```
/**/     float  ship;        /**/
/**/     struct node *next;  /**/
/**/  } List;                /**/

void print(List *start) {
   if (start != NULL) {
      List *p;
      p = start;
      while (p != NULL) {
          printf("%.1f ", p->ship);
          p = p->next;
      }
      putchar('\n');
   }
   return;
}

int main(void) {
   List *first = NULL;     // point to first node
   List *previous = NULL;  // point to previous node, or NULL
   List *n;

   // create a linked list of MAX float nodes
   for (float f = 0.0; f <= MAX; f++) {
      n = malloc(sizeof(struct node));
      if (n == NULL) {
          fprintf(stderr, "Out of memory\n");
          return EXIT_FAILURE;
      }
      n->ship = f;         // put data in the node
      n->next = NULL;      // assume no next (maybe last node)

      if (first == NULL) { // if NULL, this is the first node
          first = n;       // REALLY IMPORTANT TO REMEMBER FIRST NODE
      }
      else {
          previous->next = n; // if not first, BACKPATCH previous
      }
      previous = n;         // remember this node for next iteration
   }
   print(first);

   // un-create, i.e. free, the linked list
   n = first;               // start at the first node
   while (n != NULL) {      // as long as there are more nodes
      List *tmp = n->next; // remember next in tmp before freeing
      free(n);             // free = put memory back on heap
      n = tmp;             // n is now the next node
   }
   // print(first);         // WHAT DOES THIS DO?
   // print(previous);      // WHAT DOES THIS DO?
   // print(n);             // WHAT DOES THIS DO?

   // don't leave anything dangling
   first = NULL;
   //print(first);          // WHAT DOES THIS DO?
   previous = NULL;
   n = NULL;
   return EXIT_SUCCESS;
}
```

Compile and execute:

```
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

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

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

切换行号显示

```
void freeList(List *head) {
    List *cur;
    cur = head;
    while (cur != NULL) {
        List *tmp = cur->next; // save ptr to next node before free the node
        free(cur);
        cur = tmp;
    }
    return;
}
```

## List node creation

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

切换行号显示

```
List *makeNode(int v) {
    List *new;
    new = malloc(sizeof(List));
    if (new == NULL) {
       fprintf(stderr, "Out of memory\n");
       exit(1);
    }
    new->data = v;
    new->next = NULL; // play it safe and make it NULL
    return new;
}
```

## List node deletion

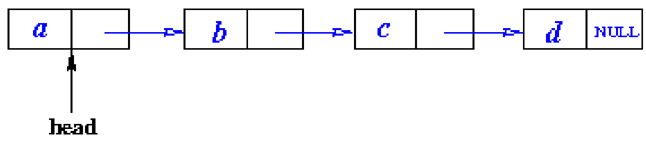
Delete a given node *n* from a linked list

切换行号显示

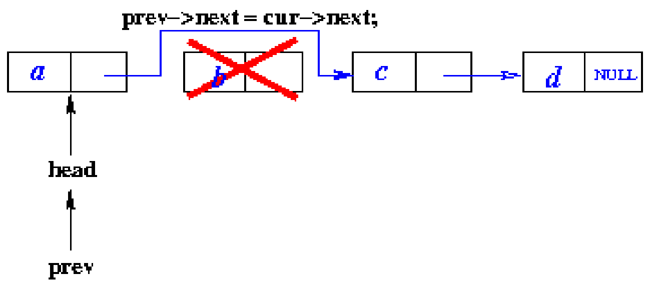
```
List *deleteNode(List *head, List *remn) {
    // if node remn is found it is removed and freed
    // it is really important to make sure we do not leave the list headless
    List *prev = NULL;

    if (head == NULL) {                     // no list: something wrong?
        return head;                        // this return is best placed here
    }

    List *cur = head;
    while (cur != remn && cur != NULL) {  // look for remn
        prev = cur;
        cur = cur->next;
    }
    if (cur != NULL) {                      // cur must be remn
        if (prev == NULL) {                 // if prev is NULL then cur = head
            head = cur->next;               // remove head, make its next the head
        }
        else {                              // if cur has a prev
```

```
            prev->next = cur->next;      // jump over cur by backpatching prev
        }
        free(cur);                       // either way, cur is freed
        cur = NULL;
    }
    // if cur==NULL then remn is not in list so nothing to do
    return head;
}
```

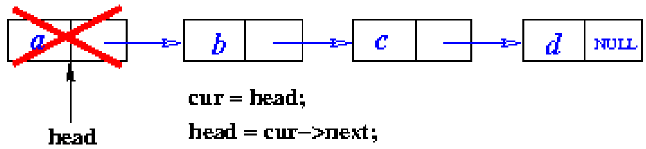
**Linked list**




**Delete from the 'middle' or delete the end node**




**Delete the head**




# 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

```
// quack.h: an interface definition for a queue/stack
#include <stdio.h>
#include <stdlib.h>

typedef struct node *Quack;

Quack createQuack(void);     // create and return Quack
Quack destroyQuack(Quack);   // remove the Quack
void  push(int, Quack);      // put the given integer onto the top of the quack
void  qush(int, Quack);      // put the given integer onto the bottom of the quack
int   pop(Quack);            // pop and return the top element on the quack
int   isEmptyQuack(Quack);   // return 1 is Quack is empty, else 0
void  makeEmptyQuack(Quack);// remove all the elements on Quack
void  showQuack(Quack);      // print the contents of Quack, from the top down

```

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

```
// QuackLL.c: a linked-list-based implementation of a quack
#include "Quack.h"
#include "limits.h"

#define MARKERDATA INT_MAX // dummy data

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

Quack createQuack(void) {
   Quack marker;
   marker = malloc(sizeof(struct node));
   if (marker == NULL) {
      fprintf (stderr, "createQuack: no memory, aborting\n");
      exit(1);
   }
   marker->data = MARKERDATA; // should never be used
   marker->next = NULL;
   return marker;
}

void push(int data, Quack qs) {
   Quack newnode;
   if (qs == NULL) {
      fprintf(stderr, "push: quack not initialised\n");
   }
   else {
      newnode = malloc(sizeof(struct node));
      if (newnode == NULL) {
         fprintf(stderr, "push: no memory, aborting\n");
         exit(1);
      }
      newnode->data = data;
      newnode->next = qs->next;
      qs->next = newnode;
   }
   return;
}

void qush(int data, Quack qs) {

   // code not shown

   return;
}

int pop(Quack qs) {
   int retval = 0;
   if (qs == NULL) {
      fprintf(stderr, "pop: quack not initialised\n");
   }
   else {
      if (isEmptyQuack(qs)) {
         fprintf(stderr, "pop: quack underflow\n");
      }
      else {
         Quack topnode = qs->next; // top dummy node is always there
         retval = topnode->data;
         qs->next = topnode->next;
         free(topnode);
      }
```

```
   }
   return retval;
}

Quack destroyQuack(Quack qs) { // remove the Quack  and returns NULL
   if (qs == NULL) {
      fprintf(stderr, "destroyQuack: quack not initialised\n");
   }
   else {
      Quack ptr = qs->next;
      while (ptr != NULL) {
         Quack tmp = ptr->next;
         free(ptr);
         ptr = tmp;
      }
      free(qs);
   }
   return NULL;
}

void makeEmptyQuack(Quack qs) {
   if (qs == NULL)
      fprintf(stderr, "makeEmptyQuack: quack not initialised\n");
   else {
      while (!isEmptyQuack(qs)) {
         pop(qs);
      }
   }
   return;
}

int isEmptyQuack(Quack qs) {
   int empty = 0;
   if (qs == NULL) {
      fprintf(stderr, "isEmptyQuack: quack not initialised\n");
   }
   else {
      empty = qs->next == NULL;
   }
   return empty;
}

void showQuack(Quack qs) {
   if (qs == NULL) {
      fprintf(stderr, "showQuack: quack not initialised\n");
   }
   else {
      printf("Quack: ");
      if (qs->next == NULL) {
         printf("<< >>\n");
      }
      else {
         printf("<<");                 // start with <<
         qs = qs->next;                // step over the marker link
         while (qs->next != NULL) {
            printf("%d, ", qs->data);  // print each element
            qs = qs->next;
         }
         printf("%d>>\n", qs->data);    // last element ends with >>
      }
   }
   return;
}
```

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

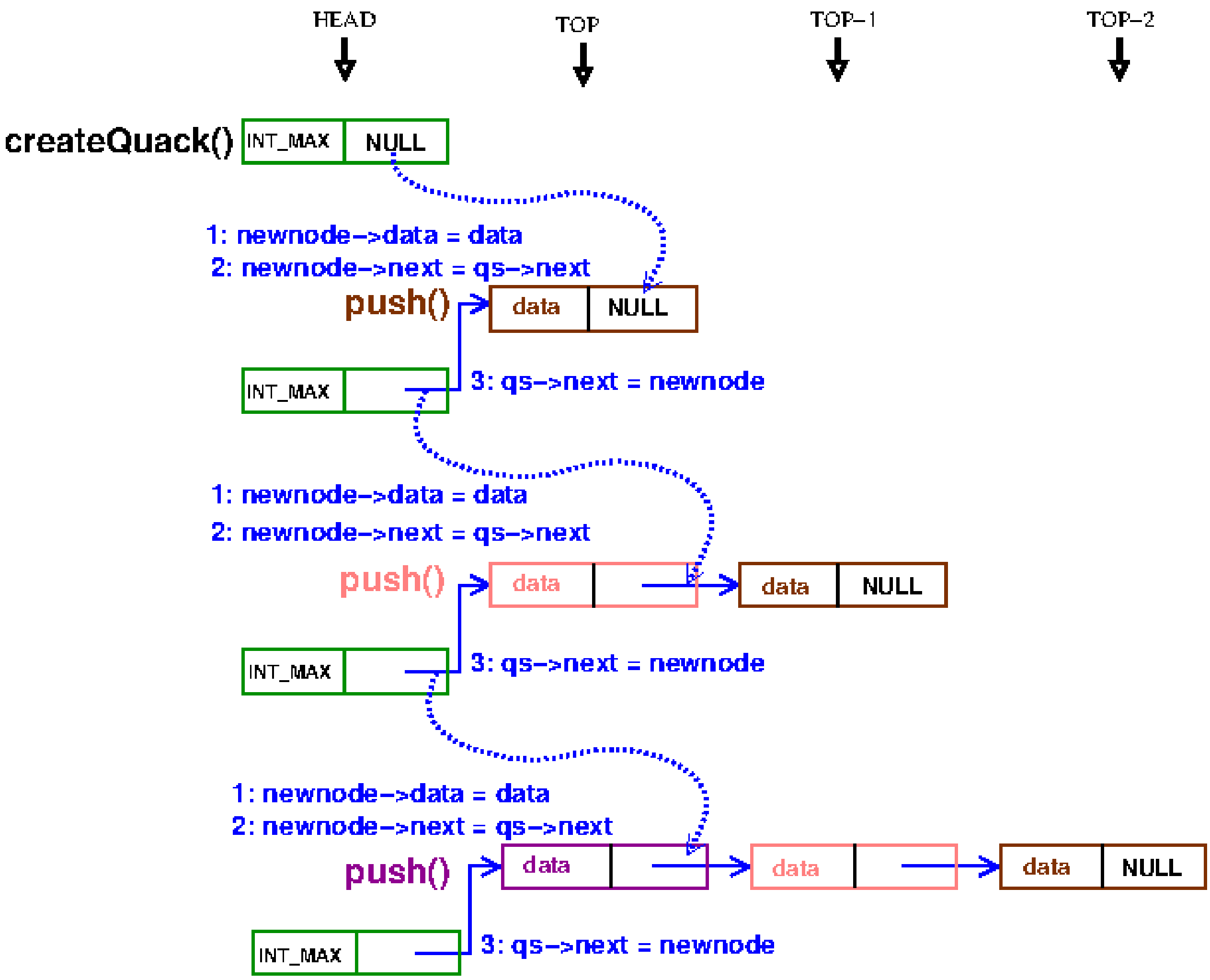
If the client node is

```
切换行号显示

Quack qs = createQuack()
```

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

```
切换行号显示

push(123, qs);
int x = pop(qs);
makeEmptyQuack(qs);
showQuack(qs);
```

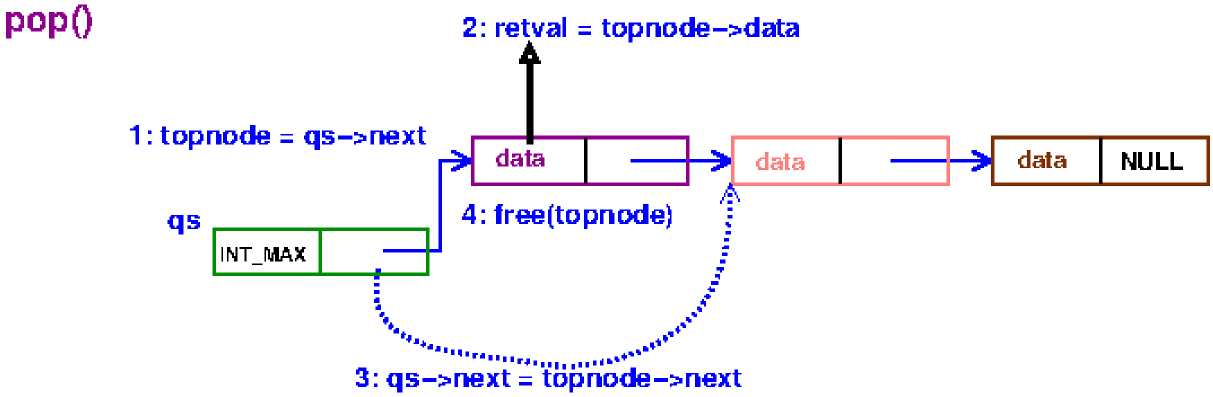
Let's create a new quack and push 3 data elements on it.




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

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

```
// revarg.c: reverse the chars in the first command-line argument
#include <stdio.h>
#include <stdlib.h>
#include "quack.h"

int main(int argc, char *argv[]) {
  Quack s = NULL;

  if (argc >= 2) {
     char *inputc = argv[1];
     s = createQuack();
     while (*inputc != '\0') {
        push(*inputc++, s);
     }
     while (!isEmptyQuack(s)) {
        printf("%c", pop(s));
     }
     putchar('\n');
  }
  return EXIT_SUCCESS;
}
```

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

```
// LL.h
// ADT interface for a linked list
#include <stdio.h>
#include <stdlib.h>

typedef struct node *List;

List createList(void);     // creates and returns an empty linked list
void putHead(int, List);   // inserts data at the head of the list
void putTail(int, List);   // inserts data at the tail of the list
int getHead(List);         // removes and returns the head of the list
int getTail(List);         // removes and returns the tail of the list
int isEmptyList(List);     // 0/1 if the linked list is empty or not
void showList(List);       // prints the linked list (not the head node)

```

## A linked-list ADT

切换行号显示

```
/*
  LL.c
  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;
   }
   return empty;
}

void showList(List marker) {
   if (marker == NULL) {
      fprintf(stderr, "showList: no linked list found\n");
   }
   else {
       printf("List: ");
       if (marker->next == NULL) {
          printf("<< >>\n");
       }
       else {
          printf("<<");                // start with <<
          List p = marker->next;       // get the head
          while (p->next != NULL) {
             printf("%d, ", p->data); // print each element
             p = p->next;
          }
          printf("%d>>\n", p->data); // last element + >>
       }
   }
   return;
}
```

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

```
// testLL.c: hard-coded tester for the linked-list ADT
//           put testdata onto the head and tail of a linked list
//           get data from the head and tail of the linked list
//           sum the data while emptying the list
#include <stdio.h>
#include <stdlib.h>
#include "LL.h"

int main() {
    int testdata[7] = {10, 20, 30, 40, 50, 60}; // 1 extra for the '\0'
    List ll = createList();

    printf("Data is:\n\t");  // check what the test data is
    for (int i = 0; i < 6; i++) {
         printf("%d ", testdata[i]);
    }
    putchar('\n');

    int *p = testdata;
```

```c
    printf("Test 1. Show each putHead of testdata:\n");
    while (*p != '\0') {
        putHead(*p++, ll);
        putchar('\t'); showList(ll);
    }

    printf("Test 2. Show 3 getTails and putHeads:\n");
    for (int i = 0; i < 3; i++) {
        putHead(getTail(ll), ll);
        putchar('\t'); showList(ll);
    }

    printf("Test 3. Show 3 getHeads and putTails:\n");
    for (int i = 0; i < 3; i++) {
        putTail(getHead(ll), ll);
        putchar('\t'); showList(ll);
    }

    printf("Test 4. Show isEmpty working, sum from back onto front\n");
    int oneleft = 0;
    while (!oneleft) {
        int tmp = getTail(ll) + getTail(ll);
        if (isEmptyList(ll)) {
            oneleft = 1;
        }
        putHead(tmp, ll);
        putchar('\t'); showList(ll);
    }

    printf("Test 5. The final act, getHead the sum and check list isEmpty\n");
    int sum = getHead(ll);
    if (isEmptyList(ll)) {
        putchar('\t'); showList(ll);
        printf("\tSum = %d\n", sum);
    }
    return EXIT_SUCCESS;
}
```

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 (2019-07-19 23:47:28由AlbertNymeyer编辑)