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;
   } LinkedL;
   ```

Example: a 2-node linked list

This is a 'silly', useless example that shows basic functionality. It is purely illustrative.

- no libraries are required
- created simply by
    - calling a malloc for each node
    - reading and inserting the data
    - linking the nodes to each other
    - also shows a traversal from *head* to *NULL*

切换行号显示

```
// ll2i.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;
} LinkedL;

int main(void) {
   LinkedL *n1;
   LinkedL *n2;

   n1 = malloc(sizeof(LinkedL)); // get memory space for the first node
   n2 = malloc(sizeof(LinkedL)); // get memory space for the second node
   if (n1==NULL || n2==NULL){
       fprintf(stderr, "Out of memory\n");
       return EXIT_FAILURE;
   }
   printf("Enter first integer: ");
   if (scanf("%d", &n1->data) == 1) {          // first data
       n1->next = n2;                          // first link

       printf("Enter second integer: ");
       if (scanf("%d", &n2->data) == 1) {      // second data
           n2->next = NULL;                    // second and last link

           // traverse the list (of 2 elements)
           for (LinkedL *p = n1; p != NULL; p = p->next) {
              printf("%d\n", p->data);
           }
       }
   }
   // tidy up
   free(n1);  // give back the memory!
   free(n2);
   n1 = NULL; // zap the pointer so it cannot be reused
   n2 = NULL;
   return EXIT_SUCCESS;
}
```

Example: Woolloomooloo in a linked list

Construct a linked list consisting of the letters *W o o l l o o m o o l o o*, and print the contents of the linked list.

- this extends the 2-node list above
- no library required
- no ADT

切换行号显示

```
// woolly.c: construct a linked list of the letters "Woolloomooloo", and print the list
#include <stdio.h>
#include <stdlib.h>

struct lll {
   char letter;
   struct lll *next;
};

int main(void) {
   typedef struct lll Letter; // this typedef allows me to use
```

```
the type 'Letter'
    Letter *l;
    Letter *firstl = NULL;
    Letter *previousl = NULL;
    char *woolly = "woolloomooloo";

    for (char *w = woolly; *w != '\0'; w++) {
       l = malloc(sizeof(Letter)); // make a node containing a letter
       if (l == NULL) {
           fprintf(stderr, "Out of memory\n");
           return EXIT_FAILURE;
       }
       l->letter = *w;           // this adds the data
       l->next = NULL;           // this adds the link to the next node (assume NULL)

       if (w == woolly){         // if w==woolly we are doing the first letter
           firstl = l;           // we MUST remember the address of the first node
       }
       else {                    // if not first, back-patch previous node
           previousl->next = l;
       }
       previousl = l;            // remember this node for the next iteration
    }

    // now let's see what the linked list looks like
    printf("The linked struct has stored ...\n");
    for (l = firstl; l != NULL; l = l->next) {
        printf("\tletter %c\n", l->letter);
    }

    l = firstl;
    // freeing is better with a while loop
    printf("Cleaning up: freeing ");
    while (l != NULL) {
       Letter *tmp = l->next;       // remember 'next' before freeing the element
       printf("%c ", l->letter);
       free(l);
       l = tmp;
    }
    putchar('\n');
    return EXIT_SUCCESS;
}
```

The output of the program is:

```
The linked struct has stored ...
        letter w
        letter o
        letter o
        letter l
        letter l
        letter o
        letter o
        letter m
        letter o
```

```
        letter o
        letter l
        letter o
        letter o
Cleaning up: freeing w o o l l o o m o o l o o
```

Standard linked-list functionality

List traversal

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

- Example 1: put the print traversal above in a function

切换行号显示

```
void printList(LinkedL *head) {
    LinkedL *cur;
    for (cur = head; cur != NULL; cur = cur->next) {
       printf("%d\n", cur->data);
    }
    return;
}
```

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

切换行号显示

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

这里要将next存储，如果存储当前的话，free之后两者都没了

List node creation

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

切换行号显示

```
LinkedL *makeNode(int v) {
    LinkedL *new;
    new = malloc(sizeof(LinkedL));
    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;
}
```

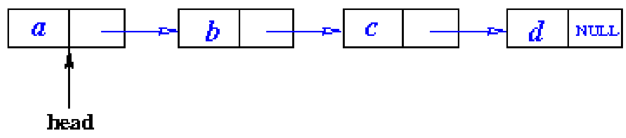
List node deletion

Delete a given node *n* from a linked list

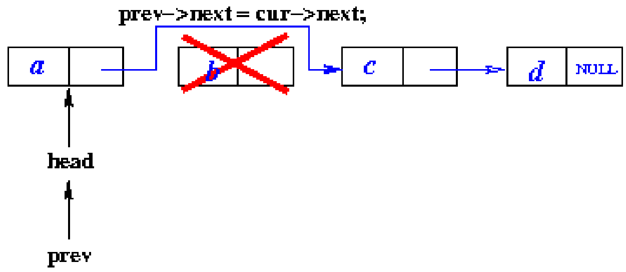
切换行号显示

```
LinkedL *deleteNode(LinkedL *head, LinkedL *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
    LinkedL *prev = NULL;

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

    LinkedL *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;
}
```

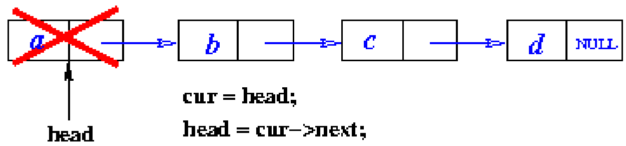
Linked list



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
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 HEAD_DATA INT_MAX // dummy data

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

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

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);
      }
      // insert the newnode at the head
      newnode->data = data;        // assign the data
      newnode->next = qs->next;    // link to 'old' linked list
      qs->next = newnode;          // make it the head
   }
   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;   // grab the data
         qs->next = topnode->next; // remove the head
         free(topnode);            // clean up
      }
   }
   return retval;
}

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 {
      if (qs->data != HEAD_DATA) {
         fprintf(stderr, "showQuack: linked list head corrupted\n");
      }
      else {
         printf("Quack: ");
         if (qs->next == NULL) {
            printf("<< >>\n");
         }
         else {
            printf("<<");                  // start with <<
            qs = qs->next;                 // step over the head 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

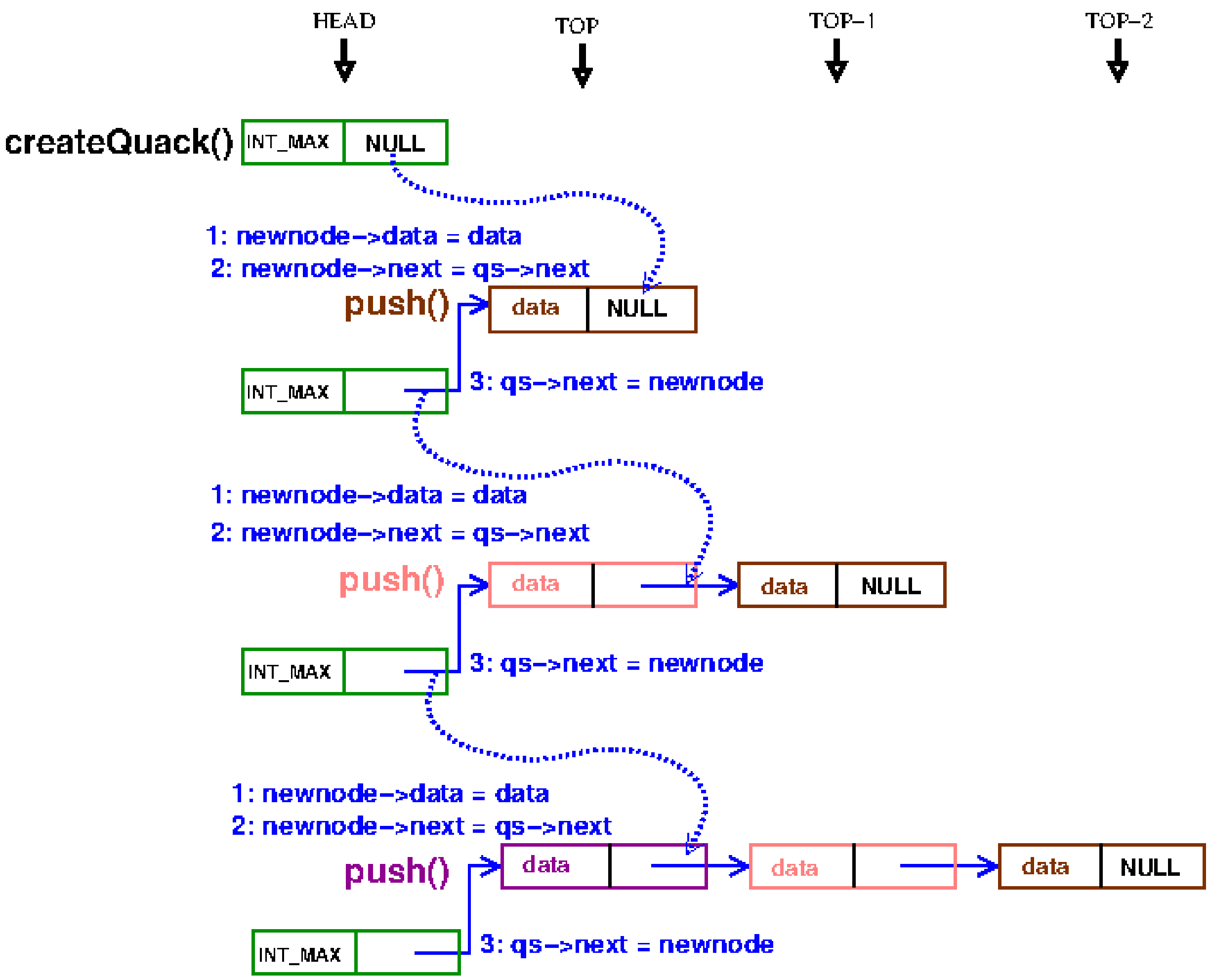
```
Quack qs = createQuack()
```

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.

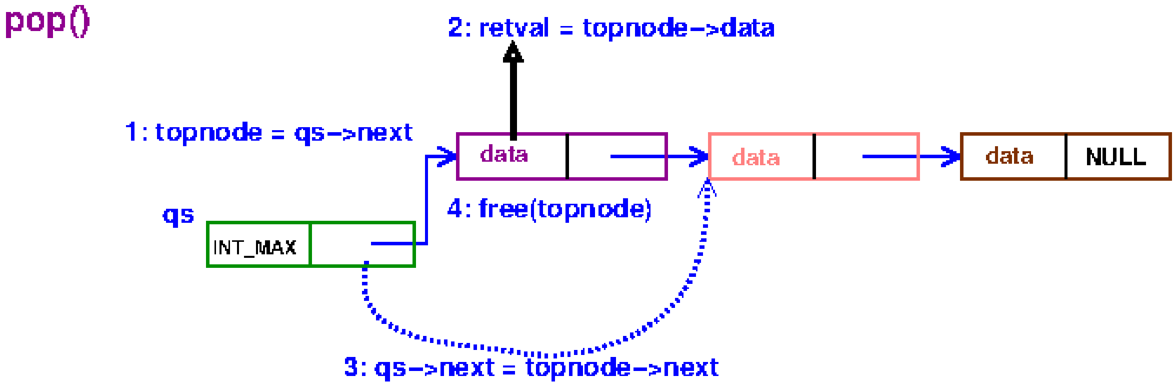


So,

- 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
```

LinkedLists (2019-06-25 11:14:14由AlbertNymeyer编辑)