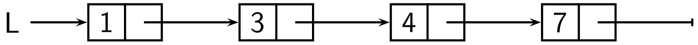
Linked Structures in C

T. Karvi

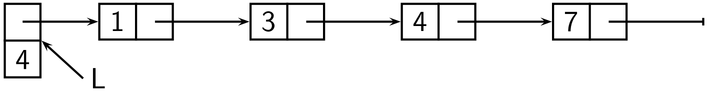
October 2015

Linked Lists I

A linked list consists of nodes. A node has a value and a link to a next node. As a diagram:

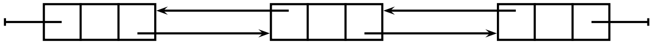


A list may have a header node which contains information about the list, for example the length of the list, and a link to the first data node.



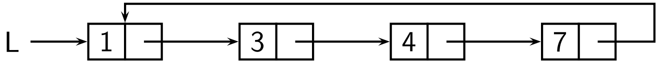
Linked Lists II

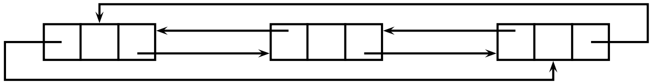
A list can be doubly linked:



Linked Lists III

Or a list can be circular:





Linked Lists IV

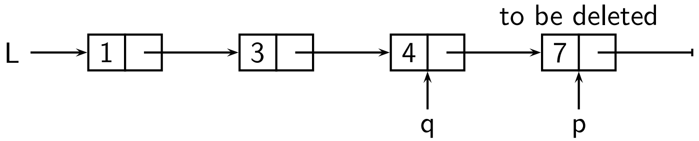
So what is a link? In C, it is typically a pointer. And what is a node? It is a structure. We have the definitions (singly linked list):

```
typedef struct node {
   Datatype value;
   struct node *next;
} NodeT, *NodeTpr;
```

Before giving examples of list functions, some words about the use of lists.

- A list is a data structure which contains several items of data of the same type.
- It is easy and efficient to add an item to a list as the first item.
- Deletion of an item an item is more complicated, if the item happens to be in the middle of the list.
- First, there should be a pointer pointing to a node to be deleted:

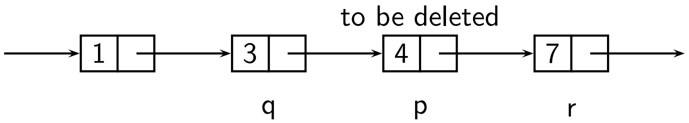
Linked Lists V



- However, how to find the previous node *q* which is needed when updating the links?
- Of course, it is always possible to start from the beginning of the list and proceed step by step forward until *q* and *p* are found.
- But this can be time-consuming, if a list is long.
- That is why you should never use lists for searching purposes, if lists can become long. For searches we have better data structures such as has tables and search trees.

Linked Lists VI

So how to arrange the deletion of a list node in such a way that the operation is efficient? Consider the following diagram:



- In the previous diagram we thought that *p* is given as a parameter to the delete function and *q* must be searched for.
- But if we give *q* as a parameter, then the node *p* can be found in one step and *r* can be found in two steps. So everything can be done in a constant time, independent of the length of the list.
- One question remains: How do we know *q* beforehand, if *p* is to be deleted?

Linked Lists VII

- Usually the operations in a list proceed step by step starting in the beginning of the list. For example, we must examine every node in a list and sometimes delete a node. In this case we can keep track of two or more nodes around the node we are dealing with. If we must do search operations, then probably we have a wrong data structure.

Insert Operation I

- Next we implement the operation `insert(L,p,a)` which inserts a value *a* into the list *L*.
- We assume that the list is singly linked and without a header. Then we must define the place into which the new node is put. It must be said that using a header node makes many operations simpler than without a header node.
- If `L==NUL`, then the list is empty and the insert operation creates a new list.
- If `(L<>NULL) and (p==NULL)`, then we add *a* as the first node.
- Finally, if `(L<>NULL) and (p<>NULL)`, then we add *a* to the right side of the node *p*.

Insert Operation II

Remember that all the parameters in a function are value parameters. Suppose we have the insert operation `insert(L,p,a)` and *L* is empty. If we create a new node and set *L* to point to this new node, then when the execution of the function is ended, *L* still points to its original value, i.e. it is NULL. What we must do in the function is to return the pointer to the new node and catch this return value in the main program.
The next code shows the insert function which adds a node into a list. (The outlook of the code seems bad. The verbatim environment in Latex Beamer does not work properly.)

Insert Operation III

```
NodeT* insert(NodeT *L, NodeT *p, int a){
     NodeT * q;


     if ( L== NULL){
    if ((q = malloc(sizeof(NodeT))) == NULL)
       return NULL;
     q -> value = a;
    q->next = NULL;
    return q;
     }
```

Insert Operation IV

```
  if (p == NULL) {
 if ((q = malloc(sizeof(NodeT))) == NULL)
   return NULL;
q->value = a;
q->next = L;
return q;
  }
```

Insert Operation V

```
  if ((q = malloc(sizeof(NodeT))) == NULL)
  return NULL;
q->value = a;
q->next = p->next;
p->next = q;
return L;

  }
```

Delete Operation I

We implement also the operation `delete(L,P)` which deletes the node after *p*. If *p* is NULL, then the first node is deleted. If *p* points to the last node, an error arises.

```
 NodeT * delete(NodeT *L, NodeT *p){
      NodeT * S;
      NodeT * q;
      NodeT * r;

      if (L == NULL) return NULL;

      if (p == NULL){
S = L->next;
free(L);
return S;
```

Delete Operation II

```
    }


    if (p->next == NULL) return NULL;

    r = p->next;
    q = r->next;
    p-> next = q;
    free(r);
    return L;
  }
```