Linked Lists – First Look

Related Material: text section 17.5

Linked Lists

An alternative to arrays

Dynamically allocated elements

Each element has a pointer to the next.

Keep track of an arbitrary number of elements.

Typically structs.

Linked Lists

- List structure can provide efficient insertion and deletion.
  - Unlike arrays.

- We can easily iterate over all items in the list just as you can with arrays.

- Drawback: we cannot access an arbitrary item directly, as we can in an array.

A linear linked list is a collection of objects, called nodes, each of which contains a data item and a pointer to the next node in the list.

One version

The last node in the list must have pointer field value NULL and the list is given by a pointer to the first node.

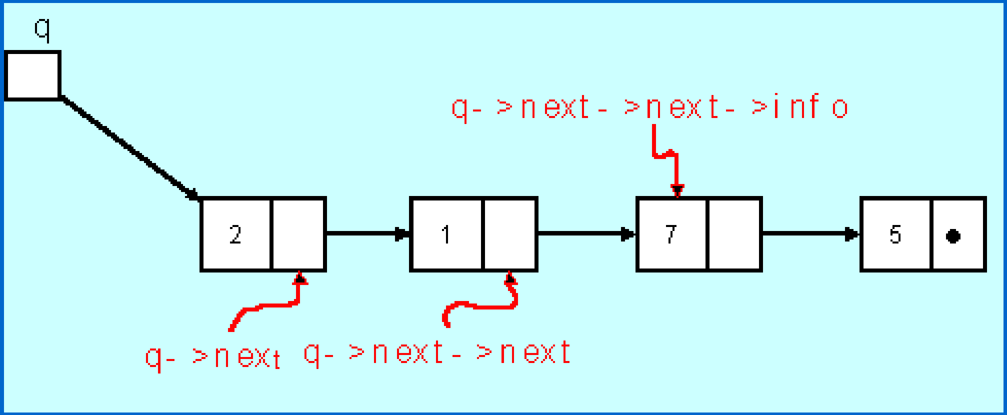
Here are the declarations.

```
    typedef struct node {
            int  info; // Could use any type here
            struct node* next;
    }  Node;


   Node* q = NULL; // corresponds to an empty list
```

Suppose we had a linked list containing the four values 2, 1, 7, 5 in exactly that order and that q points to the first node in the list.

How would we name the info field of the node containing the value 7?

The answer is `q->next->next->info`



If we were using the parentheses version of struct references, we would have to write

`(*(*(*q).next).next).info` *YUCK!*

Example Linked List Code

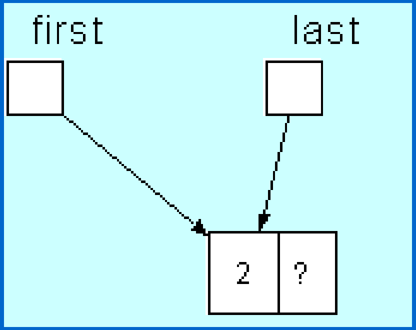
We now write a code segment that

- reads in a sequence of positive integers terminated by an entry of 0
- creates a linked list of the positive values in the order they were entered.
- prints the values from first to last.

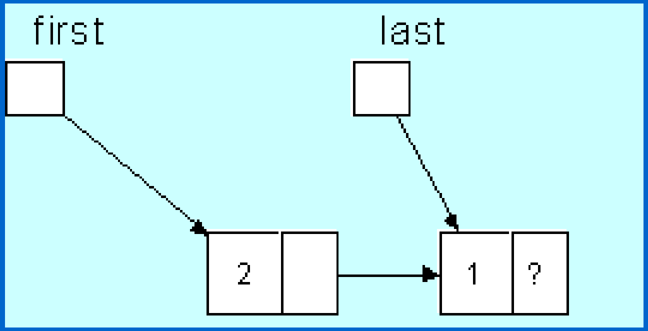
We assume that at least one positive integer is entered.

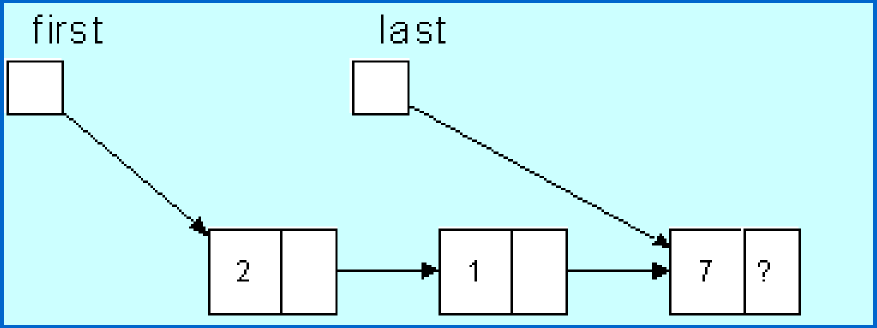
Note that this code depends on the fact that the next field of the last node has value NULL.

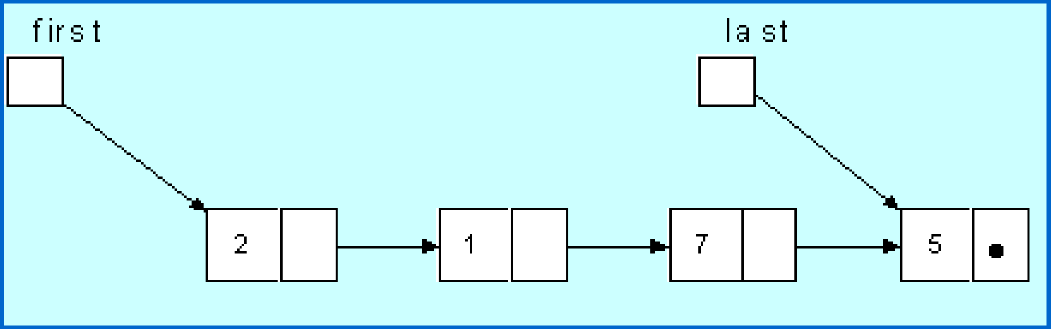
```
int  hold;
Node *first = NULL;
Node *last, *tmp;

printf("Enter the first "
       "positive number of "
       "your list: ");

scanf("%d",&hold);

first = malloc(sizeof(Node));
assert(first != NULL);

first->info = hold;
last = first;
```



```
  printf("Enter the remaining "
         "positive numbers,"
         "terminated by a "
         "zero: ");
  scanf("%d",&hold);

  while(hold != 0) {
     last->next = malloc(sizeof(Node));
     assert(last->next != NULL);
     last = last->next;
     last->info = hold;
     scanf("%d",&hold);
  }

/* Last node: NULL next field */
  last->next = NULL;
```


first
last
2
1
?


first
last
2
1
7
?


first
last
2
1
7
5

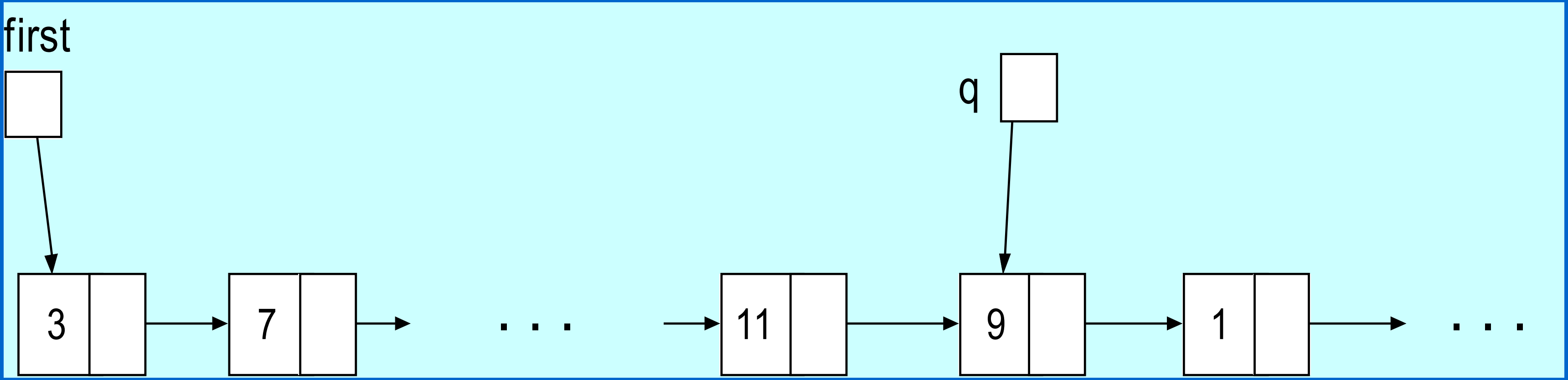
```
/* print entries */

   tmp = first;

   while (tmp != NULL){
      printf("%d ",tmp->info);
      tmp = tmp->next;
   }

   printf("\n");
```

Process info

Basic list traversal technique

More Linked List Tasks

Suppose we have a linked list given by a pointer *first* to the first node of the list and that q is a pointer to a node in the list.



We want to write four code segments to do the following tasks:

1. insert a node with info field value k just after q;
2. insert a node with info field value k just before q;
3. delete the node pointed at by q.
4. delete an entire list (make it empty)

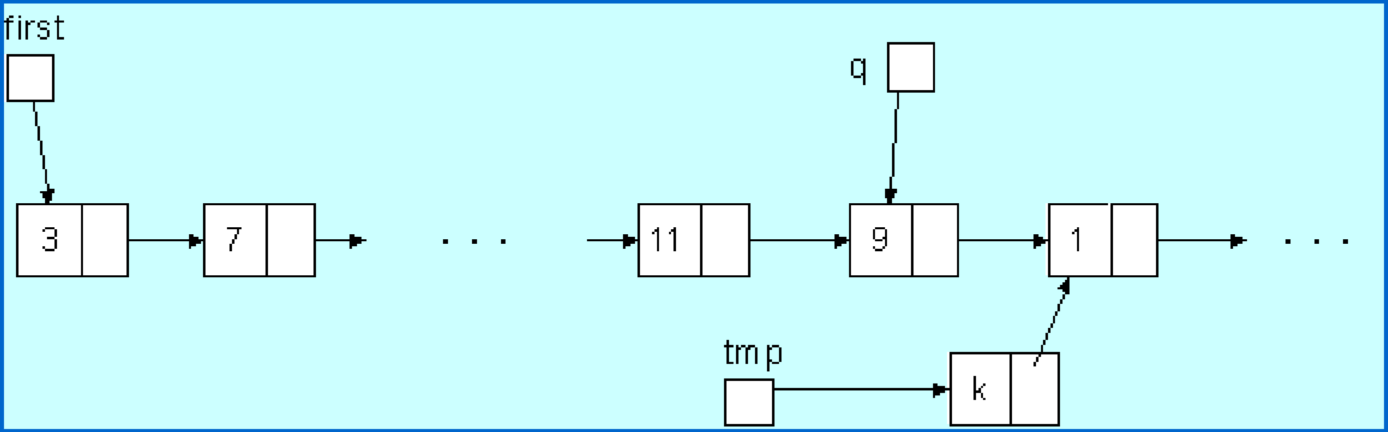
When solving problems like this it is very important to draw pictures to guide your code creation.

You should

- write the code for a generic picture
- check to see if the code works for special cases (first & last nodes)
- modify code to take care of special cases, if needed.

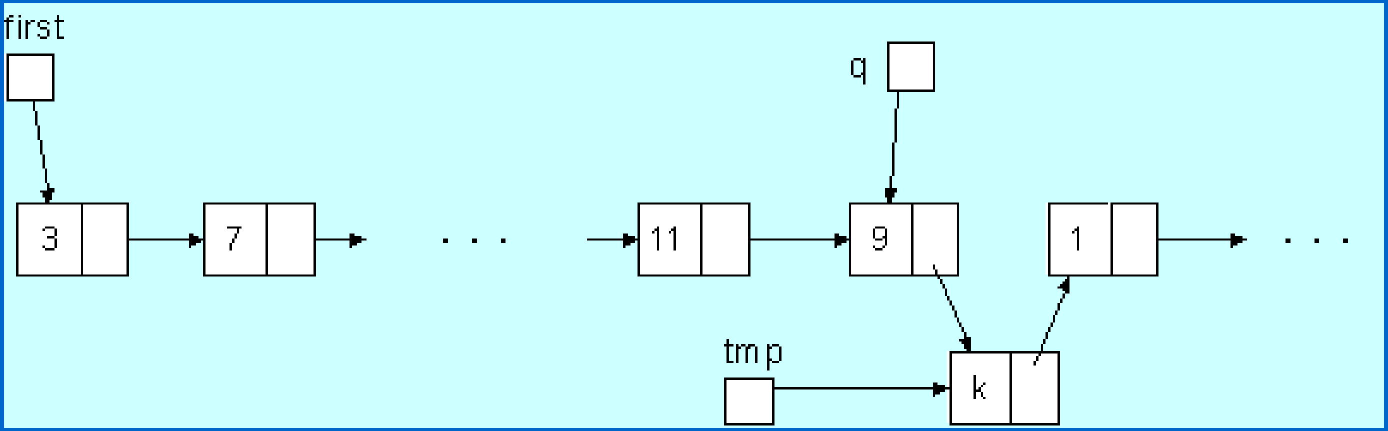
Problem 1: insert a node with info field value k just after q

```
Node * tmp = malloc(sizeof(Node));
assert(tmp != NULL);
tmp->info = k;
tmp->next = q->next;
```



Problem 1: insert a node with info field value k just after q

```
Node * tmp = malloc(sizeof(Node));
assert(tmp != NULL);
tmp->info = k;
tmp->next = q->next;
q->next = tmp;
```

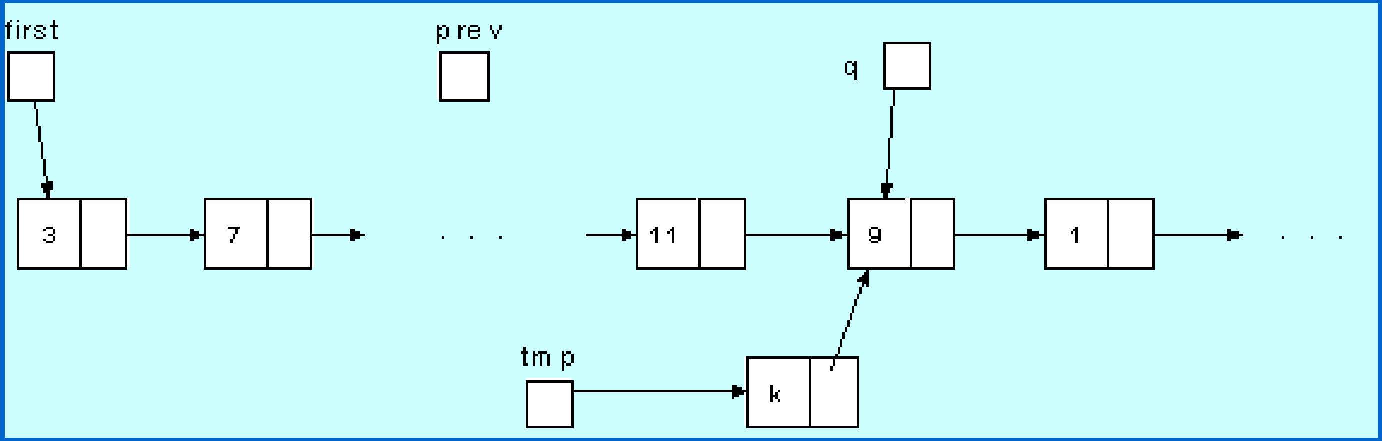


Question: does this still work if q is the first or last node in the list?

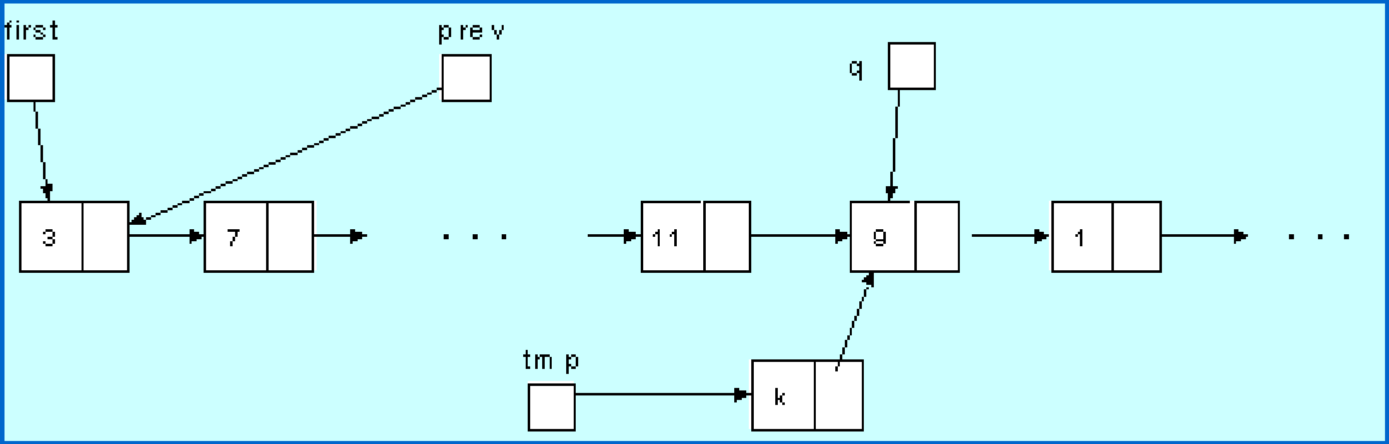
Problem 2: insert a node with info field value k just before q;

Generic case: we need to find the node directly before q and insert after that node.

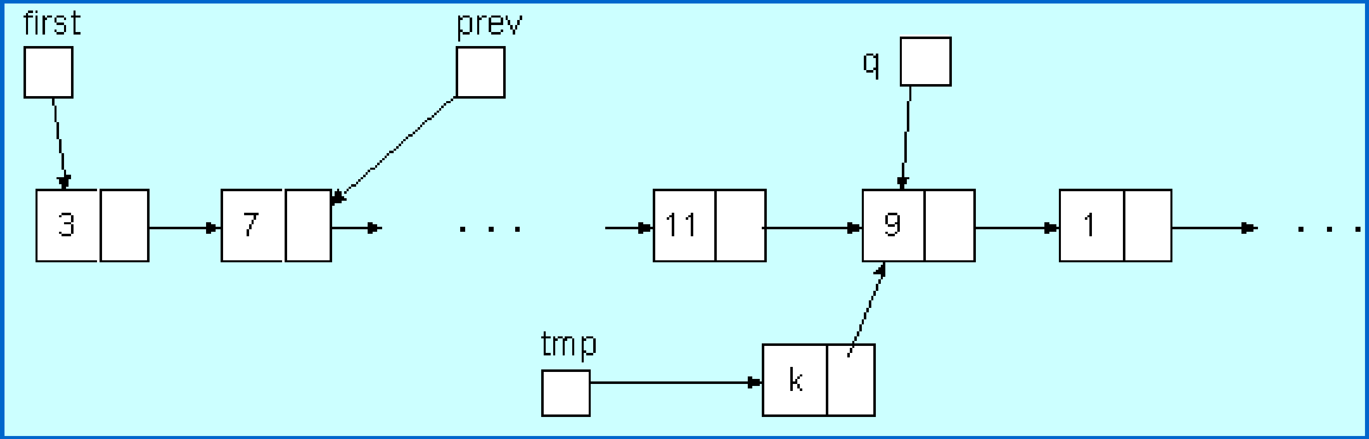
```
Node *tmp = malloc(sizeof(Node));
assert(tmp != NULL);
tmp->info = k;
tmp->next = q;
Node* prev;
```



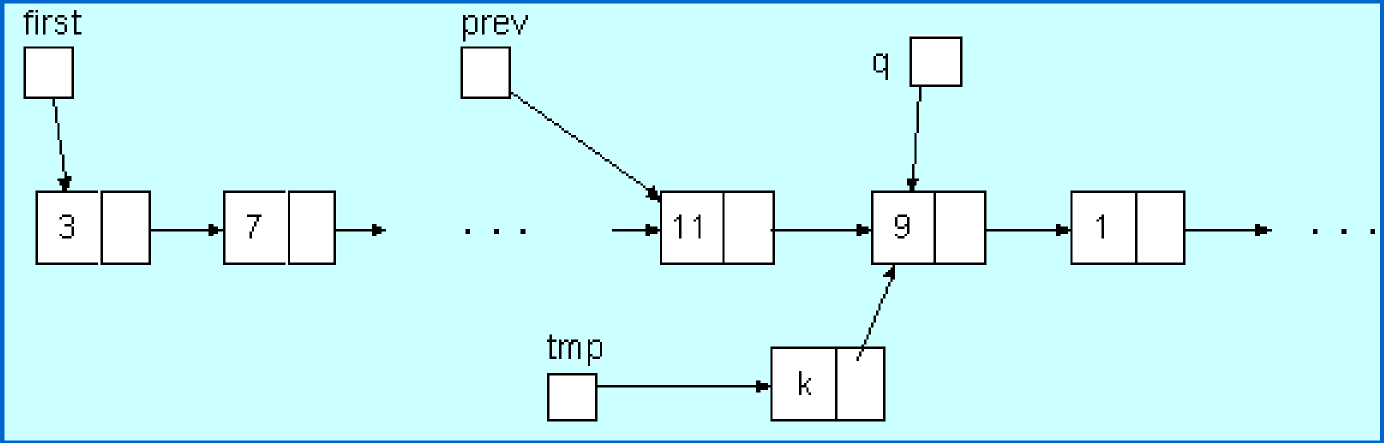
```
prev = first;
while( prev->next != q)
      prev = prev->next;
prev->next = tmp;
```



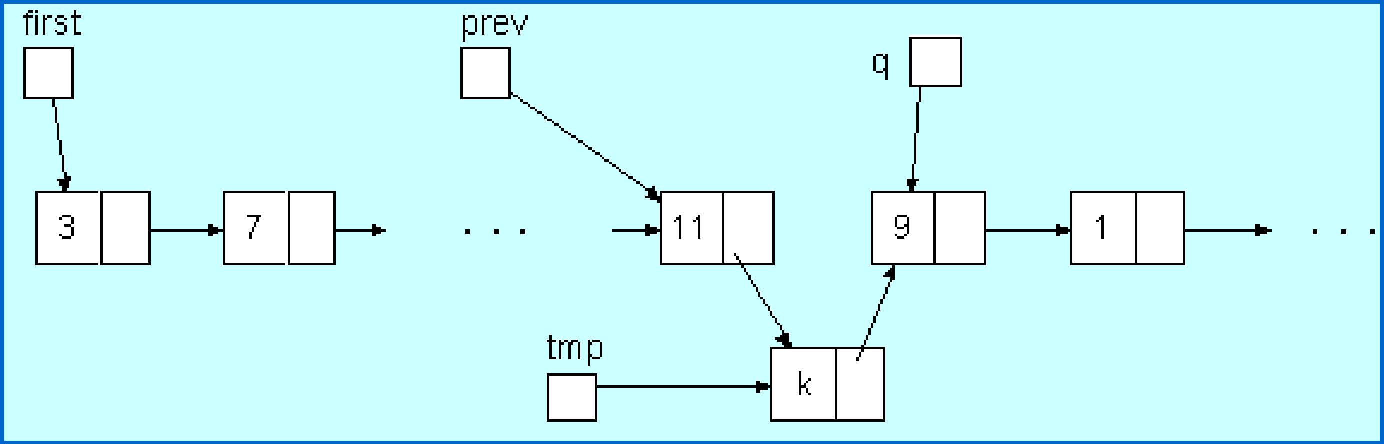
```
Node *prev = first;
while( prev->next != q)
      prev = prev->next;
prev->next = tmp;
```



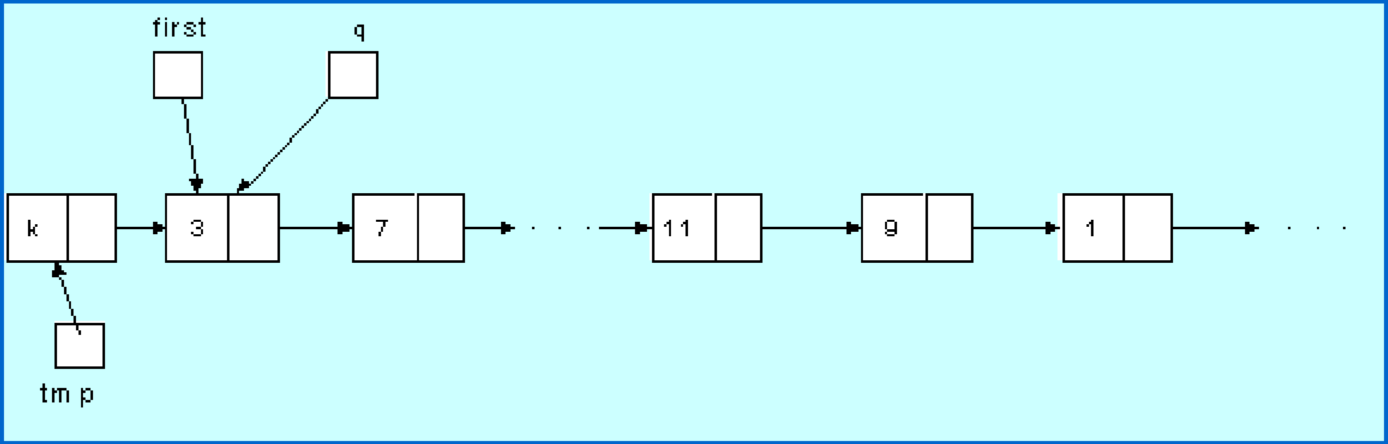
```
Node *prev = first;
while( prev->next != q)
      prev = prev->next;
prev->next = tmp;
```



```
Node *prev = first;
while( prev->next != q)
     prev = prev->next;
prev->next = tmp;
```

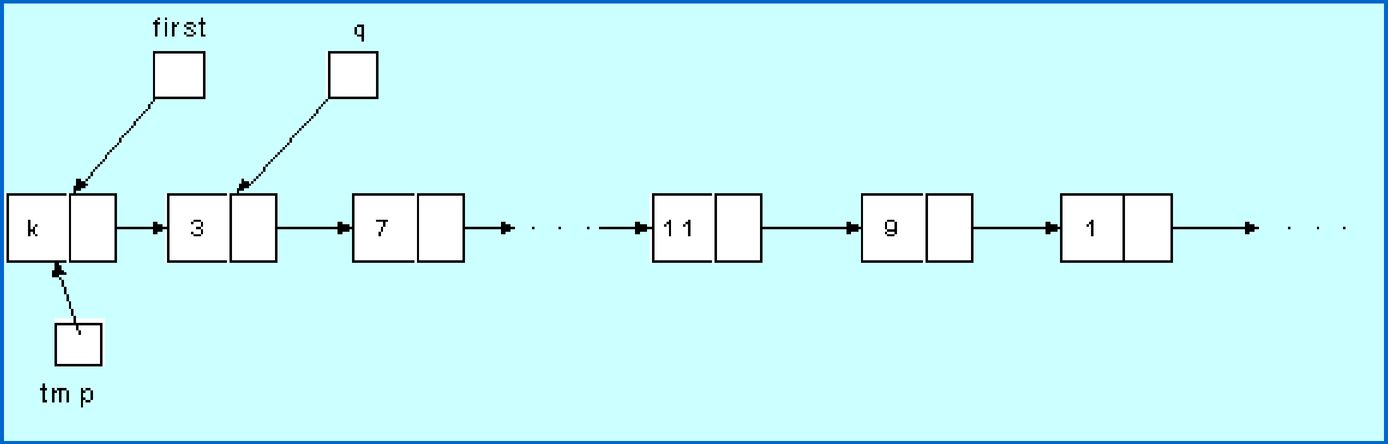


The above code does not work if q = = first, because there is no previous node! So we must treat that situation as a special case. In this case we must do the following:



The above code does not work if q = = first, because there is no previous node! So we must treat that situation as a special case. In this case we must do the following:

```
first = tmp;
```



The final code is

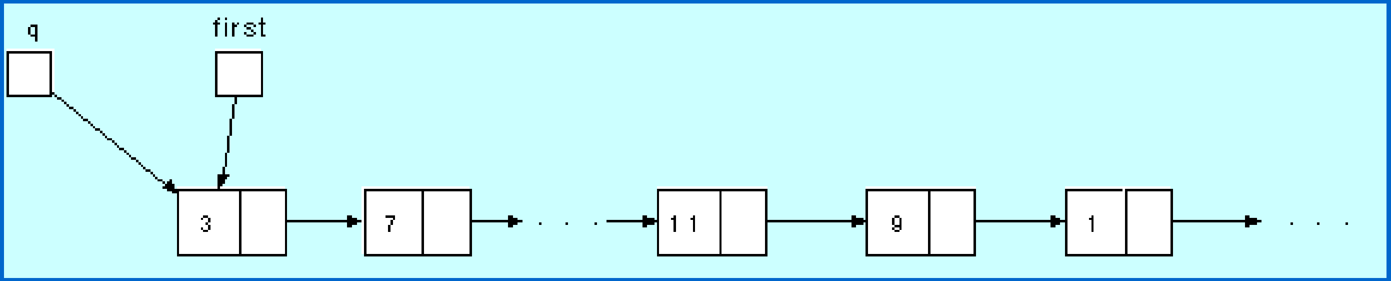
```
Node *prev;
Node *tmp = malloc(sizeof(Node));
assert(tmp != NULL);
tmp->info = k;
tmp->next = q;

if (q == first)
     first = tmp;
else {
    prev = first;
    while( prev->next != q)
          prev = prev->next;
    prev->next = tmp;
}
```

Note that the above works even if q is the last node in the list.

Problem 3: delete the node pointed at by q

First (and most obvious) approach: find the node just before q, make its next field point to the node following q, then free q. The first node is a special case. It is assumed that q != NULL.

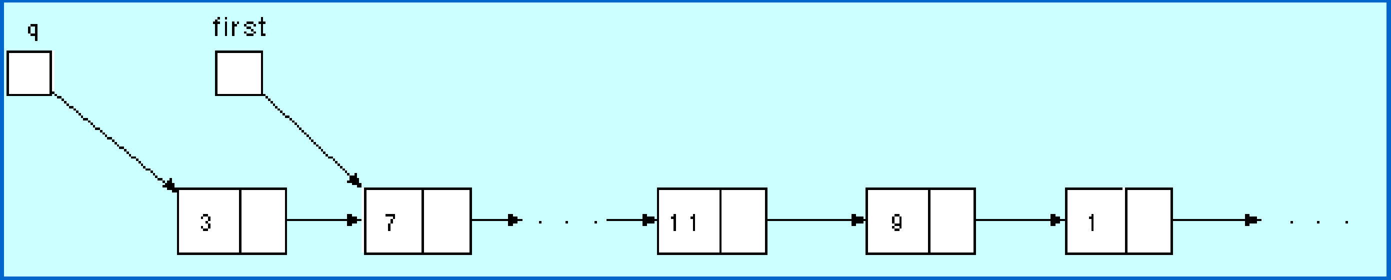
```
Node *prev;
if (q == first)
      first = first->next;
```



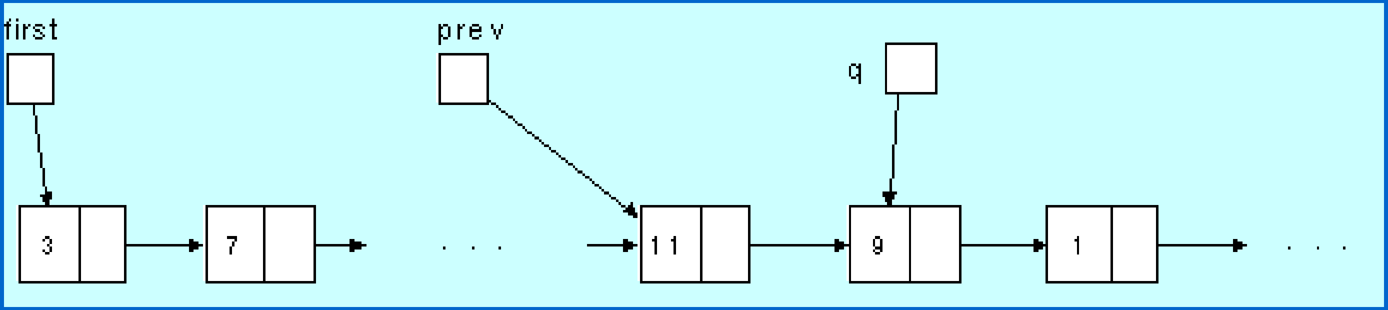
Problem 3: delete the node pointed at by q

First (and most obvious) approach: find the node just before q, make its next field point to the node following q, then free q. The first node is a special case. It is assumed that q != NULL.

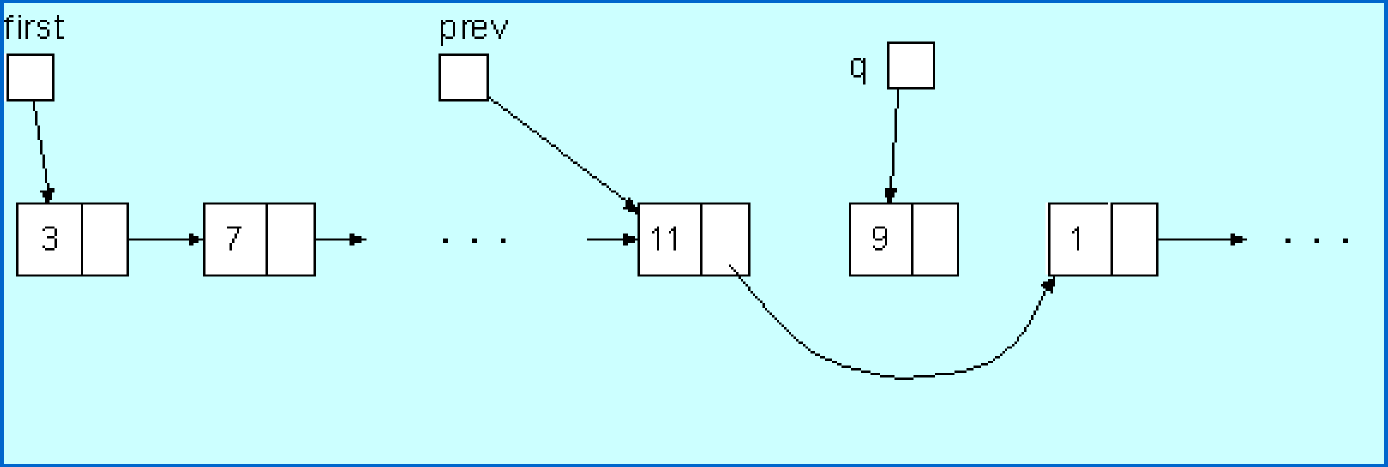
```
Node *prev;
if (q == first)
     first = first->next;
```



```
else {
      prev = first;
      while( prev->next != q)
            prev = prev->next;
```



```
else {
        prev = first;
        while( prev->next != q)
                prev = prev->next;
        prev->next = q->next;
}

free(q);
q = NULL;
```



Second approach:

if q is the last node (and not the first node) the code of the else statement is necessary.

If not, a little bit of cleverness can be used:

copy the info field of the node following q into the node q and delete the node following q (which is easy):

```
Node *tmp;
if (q == first)
     first = first->next;

else if (q->next == NULL) { // last node
     tmp = first;
     while( tmp->next != q)
      tmp = tmp->next;
     tmp->next = q->next;
   }

  else { // copy node info into next
       q->info = q->next->info;
       tmp = q; //hold onto q's node
       q = q->next; //advance to delete node
       tmp->next = q->next;
}
free(q);
```

Problem 4: delete all nodes in a linked list with first pointer p

You cannot just deallocate the target of p; you must deallocate all nodes in the list.

```
Node *tmp = p;
while(p != NULL) {
   p = p->next;
   free(tmp);
   tmp = p;
}
```