



Chap 4. Linked Lists (1)



Contents

Chapter 1. Basic Concepts

Chapter 2. Arrays And Structures

Chapter 3. Stacks And Queues

Chapter 4. Linked Lists

Chapter 5. Trees (Midterm exam)

Chapter 6. Graphs

Chapter 7. Sorting

Chapter 8. Hashing (Final exam)



Contents

4.1 Singly Linked Lists and Chains

4.2 Representing Chains in C

4.3 Linked Stacks and Queues

4.4 Polynomials

4.5 Additional List Operations

4.6 Equivalence Classes

4.7 Sparse Matrices

4.8 Doubly Linked Lists



4.1 Singly Linked Lists and Chains

❖ Ordered list

- (BAT, CAT, EAT, FAT, HAT, JAT, LAT, MAT, OAT, PAT, RAT, SAT, VAT, WAT)
- Sequential representation: *array*
- Linked representation: *linked list*



List ADT

- **Object: An ordered group with zero or more elements**
- **Operation:**
 - `add_last (list, item) ::` = Add an element to the end.
 - `add_first (list, item) ::` = Add the element to the beginning.
 - `add (list, pos, item) ::` = Add an element to pos.
 - `delete (list, pos) ::` = Removes the element at position pos.
 - `clear (list) ::` = Removes all elements of the list.
 - `replace (list, pos, item) ::` = Replace the pos element with item.
 - `is_in_list (list, item) ::` = item checks to see if it is in the list.
 - `get_entry (list, pos) ::` = Returns the element at position pos.
 - `get_length (list) ::` = Returns the length of the list.
 - `is_empty (list) ::` = Checks if the list is empty.
 - `display (list) ::` = Displays all elements in the list.



Sequential Representation

- ❖ **Sequential storage scheme**
- ❖ **Successive items of a list are located a fixed distance apart**
- ❖ ***The order of elements is the same as in the ordered list***
- ❖ **Insertion and deletion of arbitrary elements become expensive**
 - excessive data movement



Linked Representation

- ❖ **Successive items of a list may be placed anywhere in memory**
- ❖ ***The order of elements need not be the same as in the ordered list***
- ❖ **A linked list is comprised of *nodes***
 - one or more *link or pointer fields* to the next node
 - one or more *data fields*
- ❖ **Insertion and deletion of arbitrary elements become easier**
 - *no data movement*



Linked Representation

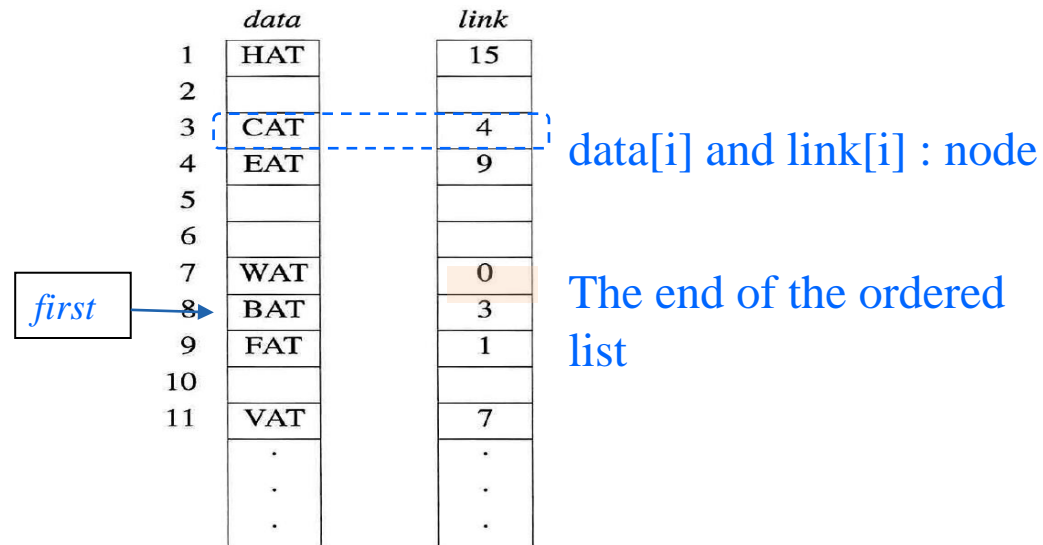


Figure 4.1: Nonsequential list-representation

using two arrays

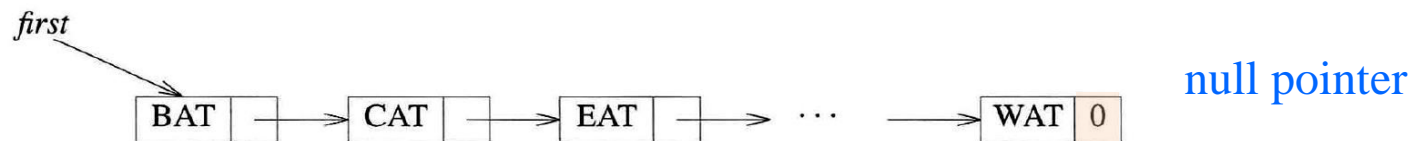


Figure 4.2: Usual way to draw a linked list



Linked Representation

- ❖ In a **singly linked list**, each node has exactly one pointer field.
- ❖ A **chain** is a singly linked list that is comprised of zero or more nodes.



Linked List : Insert(GAT)

❖ Insert GAT between FAT and HAT

- (1) Get a node a that is currently unused.
- (2) Set the data field of a to GAT.
- (3) Set the link field of a to point to the node after FAT, which contains HAT.
- (4) Set the link field of the node containing FAT to a.



Linked List : Insert(GAT)

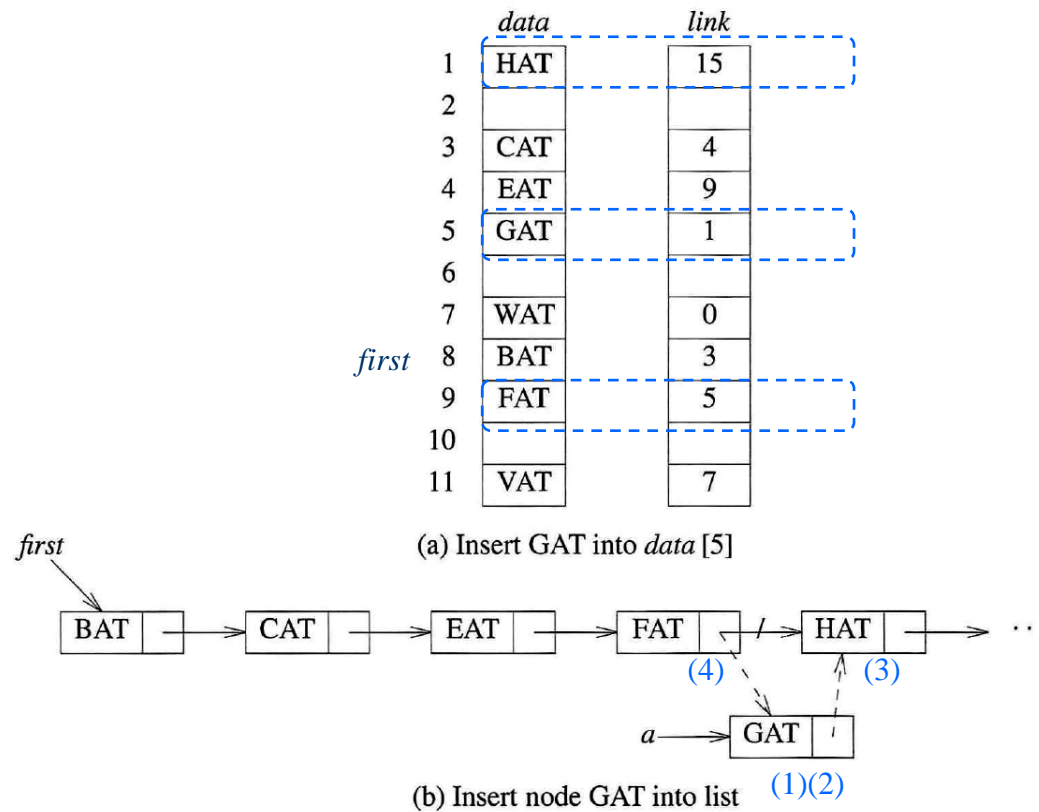


Figure 4.3: Inserting into a linked list



Linked List : Delete(GAT)

- (1) Find the element that immediately precedes GAT**
- (2) Set its link field to point to the node after GAT**

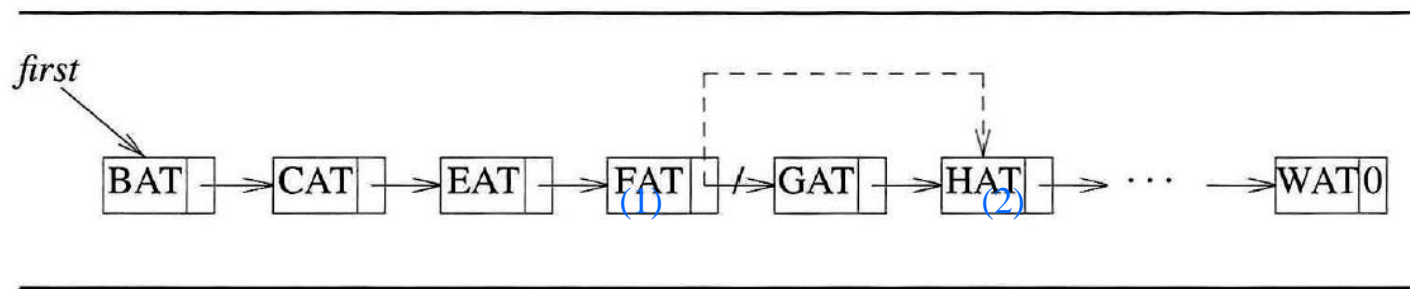


Figure 4.4: Delete GAT



Pointer

```
#include <stdio.h>>
```

```
void call_p(int*);
```

```
void call_pp(int**);
```

```
void main()
```

```
{
```

```
    int i, *ip;
```

```
    i = 100;
```

```
    ip = &i;
```

```
    printf("(i= %d, &i=%p)\n", i, &i);
```

```
    printf("(ip= %p *ip=%d), &ip=%p\n", ip, *ip, &ip);
```

```
    call_p(ip);
```

```
    printf("(ip= %p *ip=%d), &ip=%p\n", ip, *ip, &ip);
```

```
    call_pp(&ip);
```

```
    printf("(ip= %p *ip=%d), &ip=%p\n", ip, *ip, &ip);
```

```
}
```

```
void call_p(int*j)
```

```
{
```

```
    printf("(j= %p *j=%d, &j=%p)\n", j, *j, &j);
```

```
    j = (int*)malloc(sizeof(int));
```

```
    *j = 200;
```

```
    printf("(j= %p *j=%d)\n", j, *j);
```

```
}
```

```
void call_pp(int**k)
```

```
{
```

```
    printf("&k=%p, k= %p, *k=%p, **k=%d)\n", &k, k, *k, **k);
```

```
    *k = (int*)malloc(sizeof(int));
```

```
    **k = 300;
```

```
    printf("&k=%p, k= %p, *k=%p, **k=%d)\n", &k, k, *k, **k);
```

```
}
```

변수	ip		i				
주소	D44		D50				
값	D50		100				

```
(i= 100, &i=00BEFD50)
```

```
(ip= 00BEFD50 *ip=100), &ip=00BEFD44
```

```
(j= 00BEFD50 *j=100, &j=00BEFC70)
```

```
(j= 01125050 *j=200)
```

```
(ip= 00BEFD50 *ip=100), &ip=00BEFD44
```

```
(&k=00BEFC70, k= 00BEFD44, *k=00BEFD50, **k=100)
```

```
(&k=00BEFC70, k= 00BEFD44, *k=01129AA8, **k=300)
```

```
(ip= 01129AA8 *ip=300), &ip=00BEFD44
```

변수		j		ip	i	
주소	050	C70		D44	D50	
값	200	D50		D50	100	
값	200	050		AA8	100	

변수		k		ip	i	
주소	AA8	C70		D44	D50	
값		D44		D50	100	

값	300	D44		AA8		100
---	-----	-----	--	-----	--	-----

050

k=&ip, *k=ip, **k=*ip

*k

**k



4.2 Representing Chains in C

❖ Example 4.1 [List of words]

- Defining a node's structure

- *self-referential structure*

```
typedef struct listNode * listPointer;
```

```
typedef struct listNode {  
    char data[4];  
    listPointer link;  
} listNode;
```

```
struct aNode {  
    int data;  
    struct aNode* link;  
};  
struct aNode a, b, c, *plink;
```

- Creation of a new empty list

- `listPointer first = NULL;`

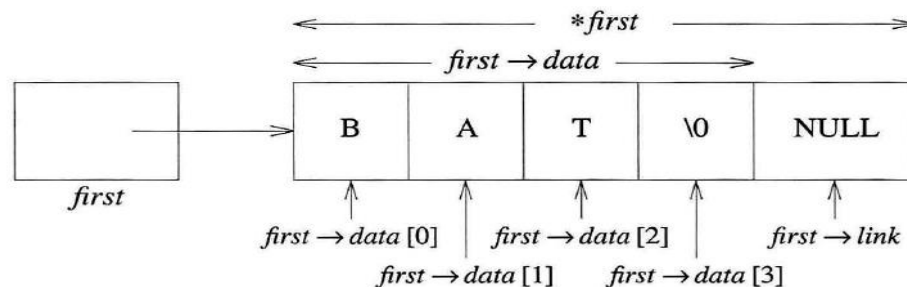
- Test for an empty list

- `#define IS_EMPTY(first) (! (first))`



Example 4.1 [List of words]

- Creation of a new node for the list
 - `MALLOC(first, sizeof(*first));`
 - `strcpy_s(first→data, strlen("BAT")+1, "BAT");`
 - `first→link = NULL;`
- Assigning values to the fields of the node



There is no next node.

Figure 4.5: Referencing the fields of a node



Example 4.2 [Two-node linked list]

listPointer create2()

```
{/ * create a linked list with two nodes */
```

```
listPointer first, second;
```

```
MALLOC(first, sizeof(*first));
```

```
MALLOC(second, sizeof(*second));
```

```
second->link = NULL;
```

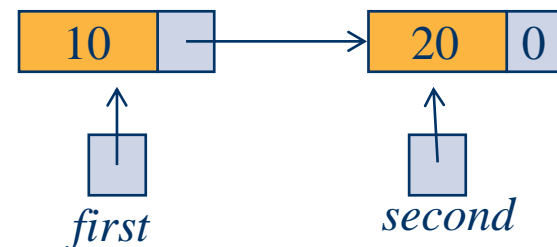
```
second->data = 20;
```

```
first->data = 10;
```

```
first->link = second;
```

```
return first;
```

```
}
```





Example 4.3 [List insertion]

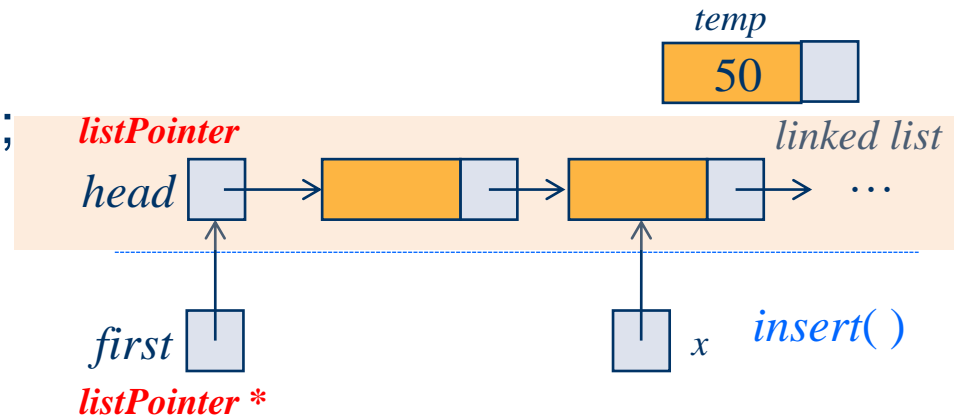
```
void insert(listPointer *first, listPointer x)
{ /* insert a new node with data = 50 into
  the chain first after node x */
```

```
listPointer temp;
MALLOCM(temp, sizeof(*temp));
temp->data = 50;
```

```
if (*first) {
    temp->link = x->link;
    x->link = temp;
}
```

```
else {
    temp->link = NULL;
    *first = temp;
}
```

```
}
```

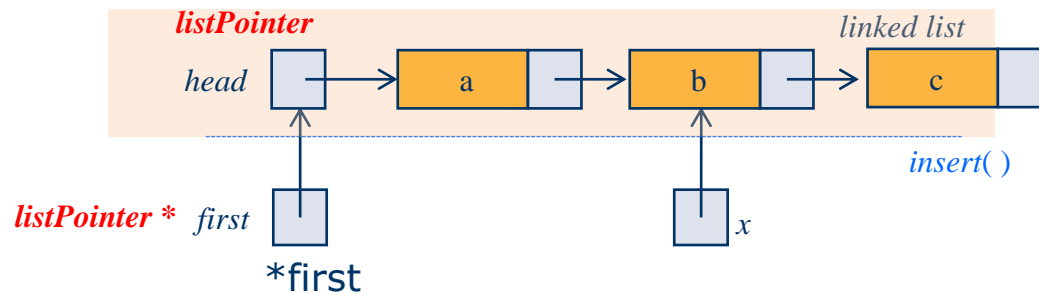


Can not insert the node at the first when list is not empty!!



Example 4.3 [List insertion]

리스트 제일 앞에 데이터 f를 삽입 temp
f function call : insert(&head, x)



변수			first		head													
주소			30		70			100						200			300	
값			70		100			a	200					b	300		c	0

*first

변수			first		head													
주소			30		70			100			150			200			300	
값			70		150			a	200		f	100		b	300		c	0



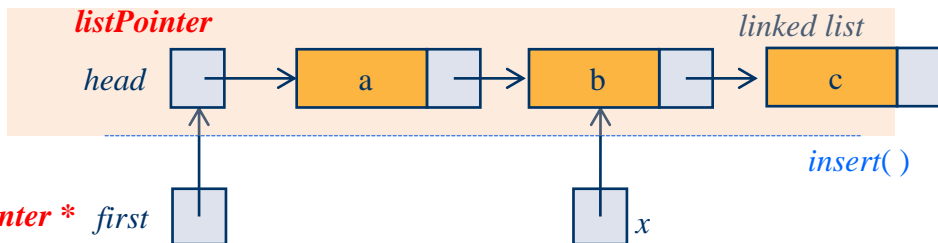
Example 4.3 [List insertion]

노드 b 뒤에 노드 f를 삽입

temp



function call : *insert(&head, x)*



			first		head														
			30		70			100						200				300	
			70		100			a	200					b	300			c	0

*first

			first	head														
			30	70			100			150			200				300	
			70	100			a	200		f	300		b	150			c	0



Example 4.3 [List insertion]

```
void insert(listPointer *first, listPointer x)  
{ /* insert a new node with data = 50 into  
   the chain
```

```
first after node x */
```

```
listPointer temp;
```

```
  MALLOC(temp, sizeof(*temp));
```

```
  temp->data = 50;
```

```
    if (*first) {  
      temp->link = x->link;  
      x->link = temp;
```

```
    else {  
      temp->link = NULL;  
      *first = temp;
```

```
    }
```

```
  }
```

(a) Inserting into an empty list

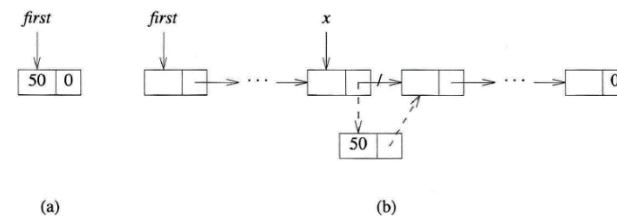
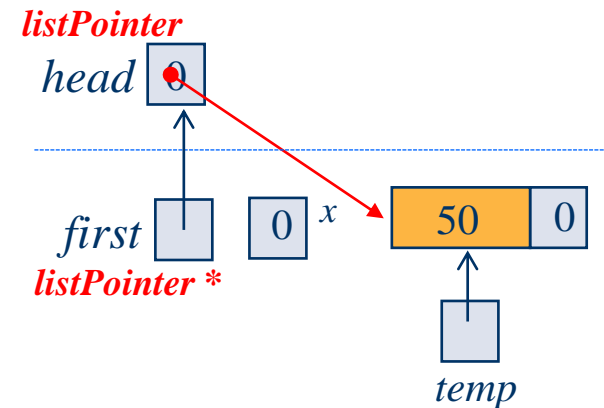


Figure 4.7: Inserting into an empty and nonempty list



Example 4.3 [List insertion]

```
void insert(listPointer *first, listPointer x)
{ /* insert a new node with data = 50 into the chain
```

```
first after node x */
```

```
listPointer temp;
```

```
MALLOC(temp, sizeof(*temp));
```

```
temp->data = 50;
```

```
if (*first) {
```

```
    temp->link = x->link;
```

```
    x->link = temp;
```

```
else {
```

```
    temp->link = NULL;
```

```
    *first = temp;
```

```
}
```

(b) Inserting into a nonempty list

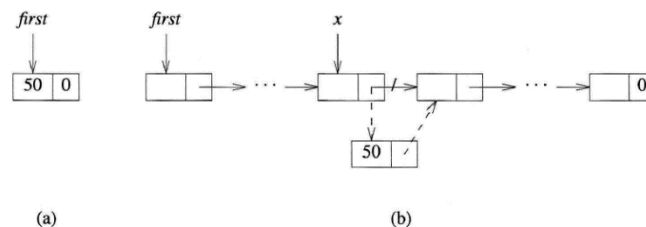
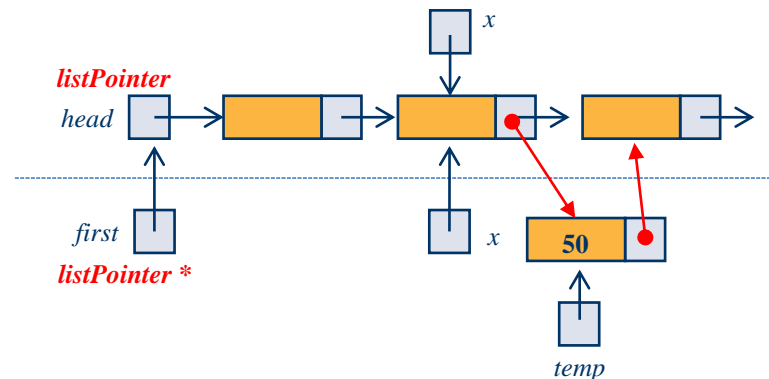


Figure 4.7: Inserting into an empty and nonempty list

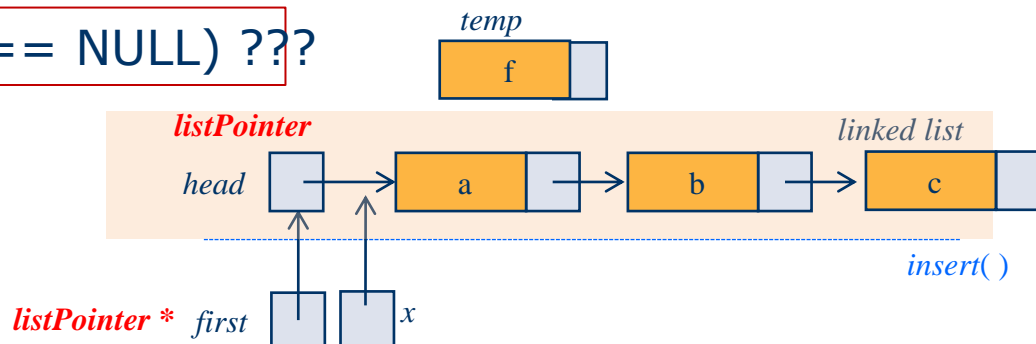


Example 4.3 [List insertion]

```
void insert(listPointer *first, listPointer x)
{ /* insert a new node with a data into the
  chain first after node x */
    listPointer temp;
    MALLOC(temp, sizeof(*temp));
    temp->data = 50;
    if(*first == NULL)
    { // add to empty list
        temp->link = NULL;
        *first = temp;
    }
```

```
else
{ // add to non-empty list
    if ( x == NULL )
    { // as a first node
        temp->link = *first;
        *first = temp;
    }
    else
    {
        temp->link = x->link;
        x->link = temp;
    }
}
```

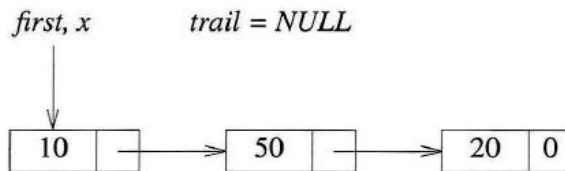
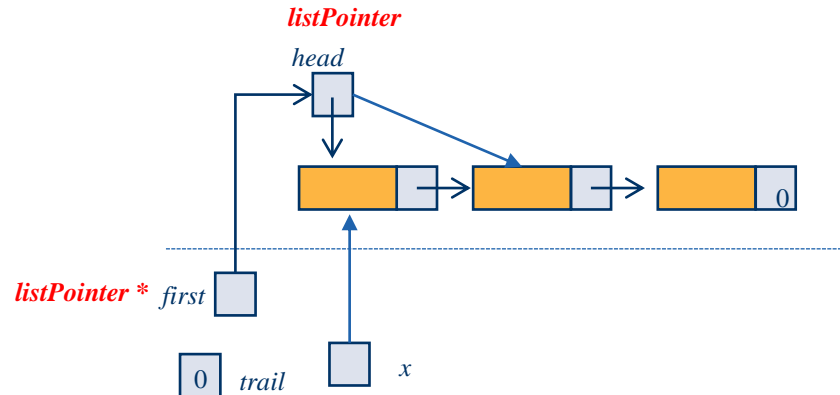
if (x == NULL) ???



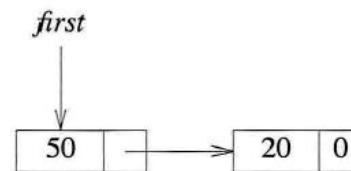


Example 4.4 [List deletion]

```
void delete(listPointer *first, listPointer trail, listPointer x )  
{ / * delete x from the list, trail is the preceding node  
  and *first is the front of the list * /  
  if (trail)  
    trail->link = x->link;  
  else  
    *first = ( * first)->link;  
  free(x);  
}
```



(a) Before deletion



(b) After deletion

Figure 4.8: List before and after the function call `delete(&first, NULL, first);`



Example 4.4 [List deletion]

```
void delete(listPointer *first, listPointer trail, listPointer x )
{ / * delete x from the list, trail is the preceding node
  and *first is the front of the list * /
  if (trail)
    trail->link = x->link;
  else
    *first = ( * first)->link;
  free(x);
}
```

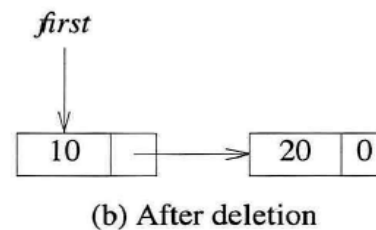
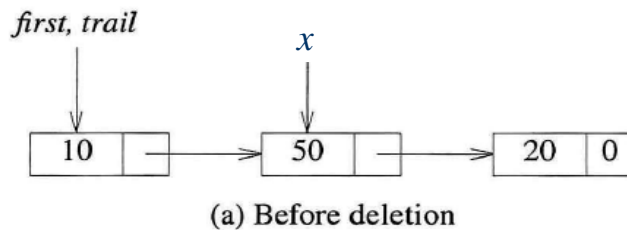
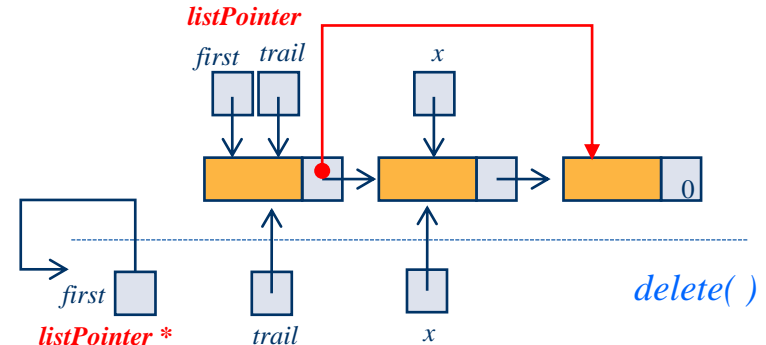


Figure 4.9: List after the function call *delete(&head, trail, x)*



Example 4.5 [Printing out a list]

```
void printList(listPointer first)
{
    printf("The list contains: ");
    for (; first; first = first->link)
        printf ( "%4d", first->data);
    printf ( " \n");
}
```

