Chap 4. Linked Lists (1)

Contents

Contents

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***
  - each node has zero or more *data fields* and one or more *link or pointer fields* to the next item
- Insertion and deletion of arbitrary elements become easier
  - no data movement

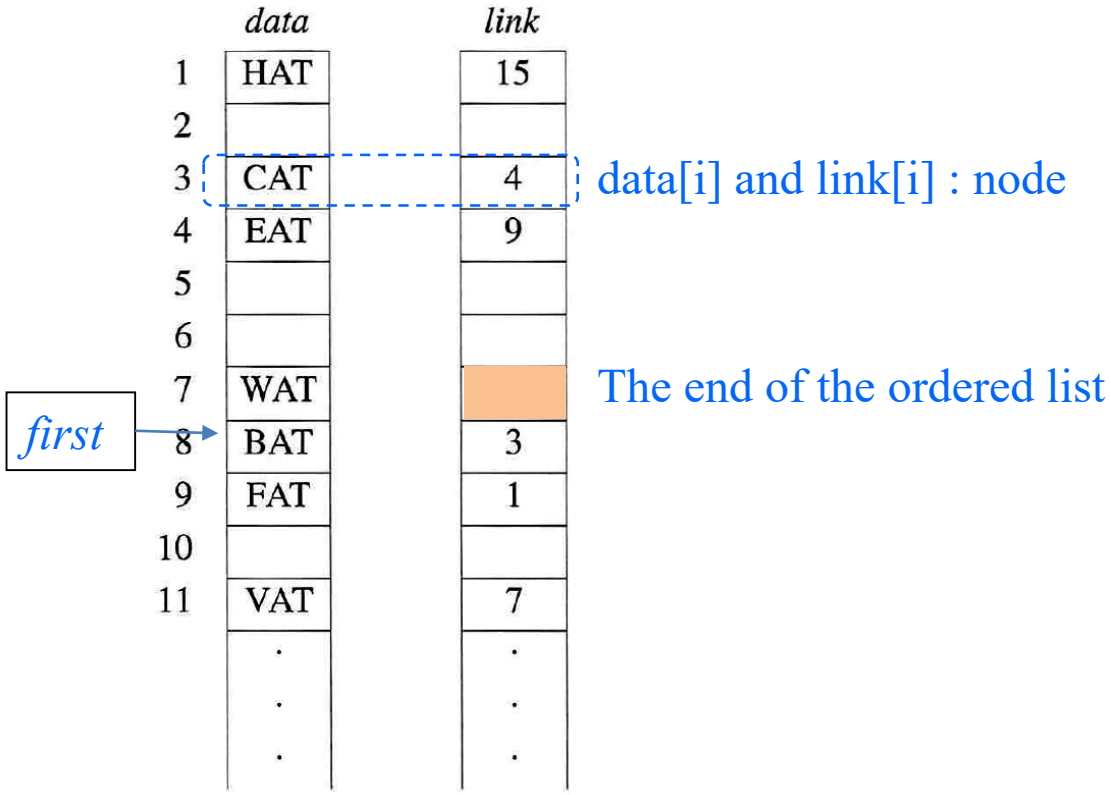


Figure 4.1: Nonsequential list-representation using two arrays

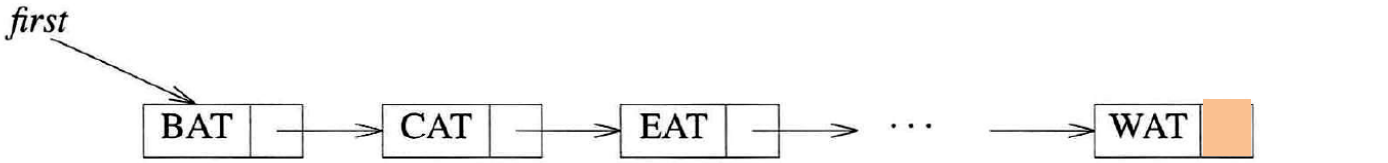


Figure 4.2: Usual way to draw a linked list

- 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*.

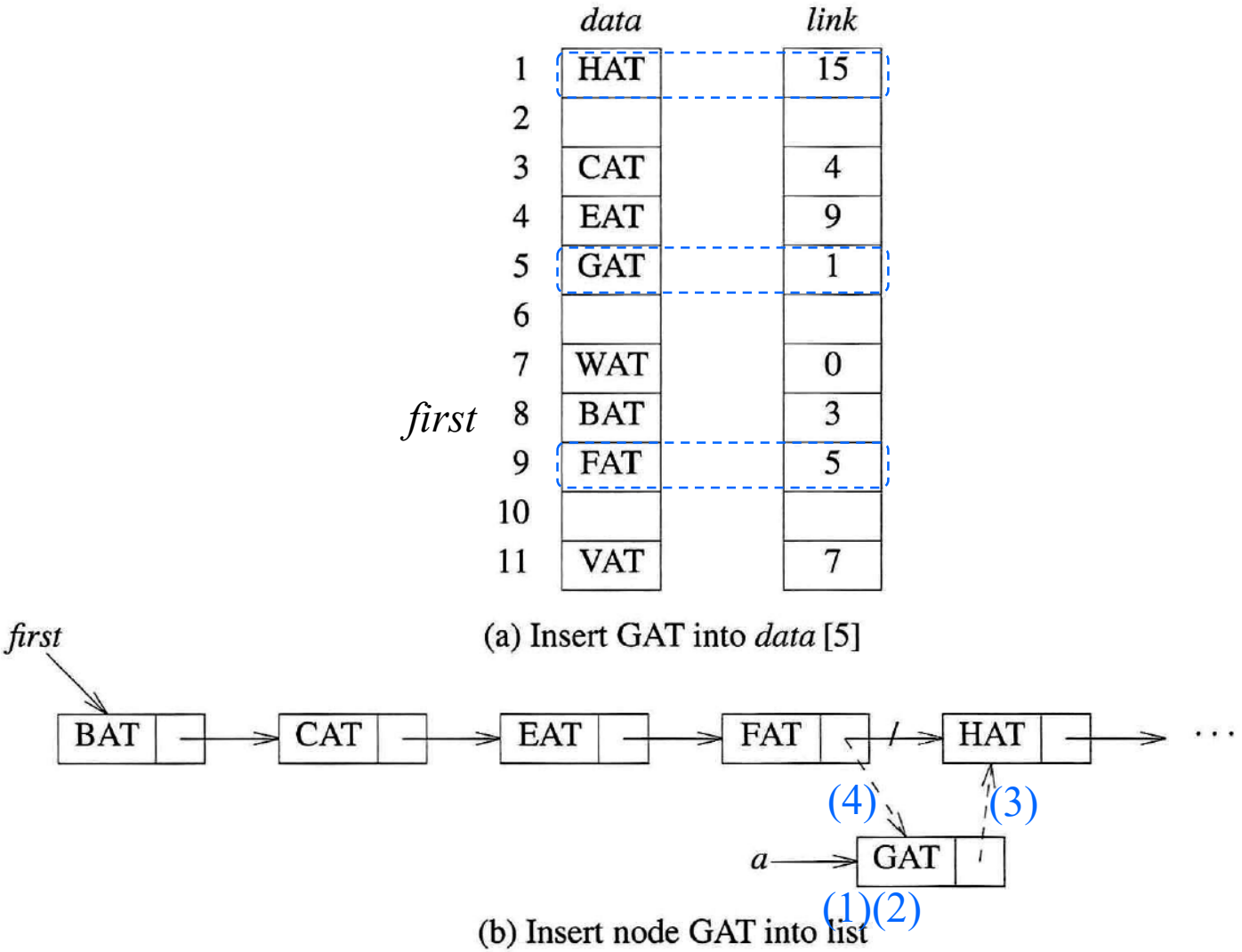


Figure 4.3: Inserting into a linked list

Linked List : Delete(GAT)

(1) Find the element that immediately precedes GAT

(2) Set its link filed to point to the node after GAT

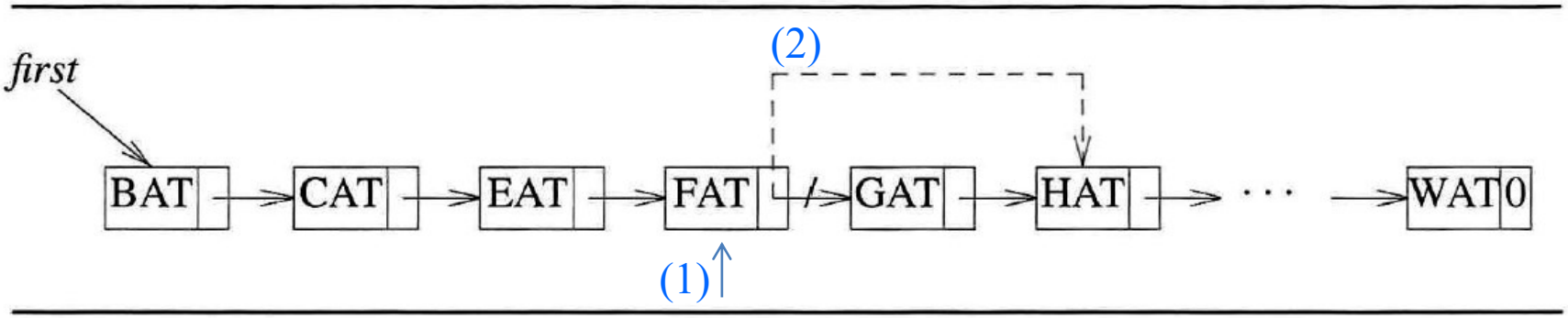


Figure 4.4: Delete GAT

```
#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);
}
```

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

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		C70			D44		D50	
값		D50			D50		100	

050 ip j

변수		k	*k	**k
주소		C70	D44	D50
값		D44	D50	100

값		D44	AA8	300

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;
```

  - 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));

- Assigning values to the fields of the node

  strcpy_s(first→data, strlen("BAT")+1, "BAT");
  first→link = NULL;

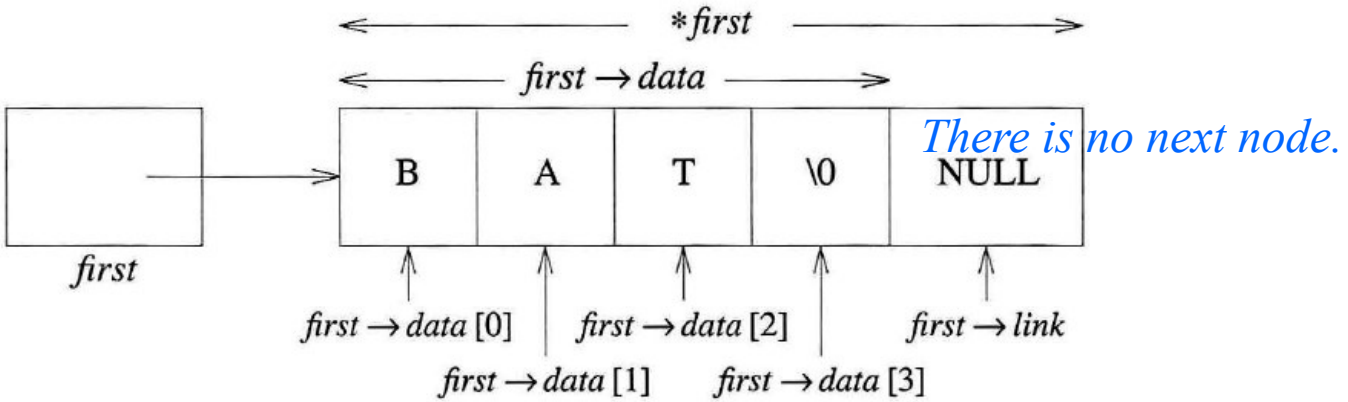
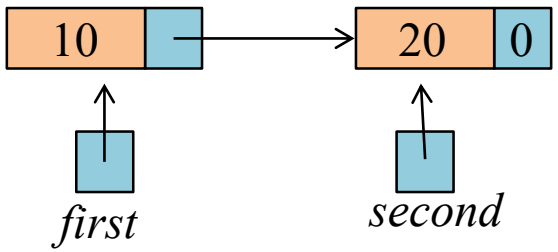


Figure 4.5: Referencing the fields of a node

Example 4.2 [Two-node linked list]

```
typedef struct listNode *listPointer;
typedef struct listNode {
        int data;
        listPointer link;
        }listNode;
```

```
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;
}
```



Program 4.1: Create a two-node list

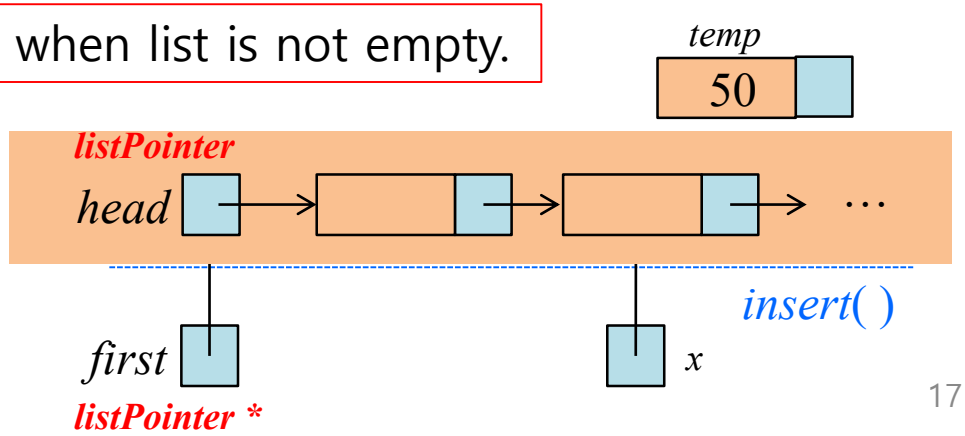
Example 4.3 [List insertion]

```
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;
   }
}
```

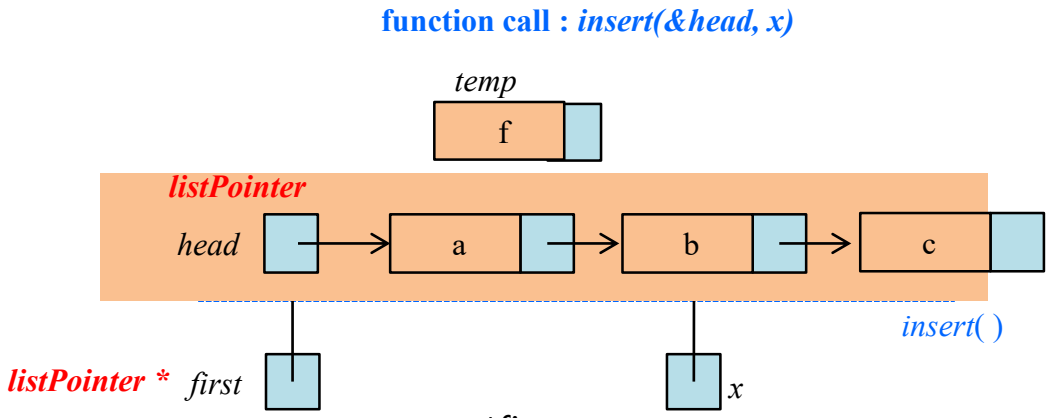
Program 4.2: Simple insert into **list**

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

can not insert node at the first when list is not empty.



Example 4.3 [List insertion]



*first

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

리스트 제일 앞에 데이터 f를 삽입

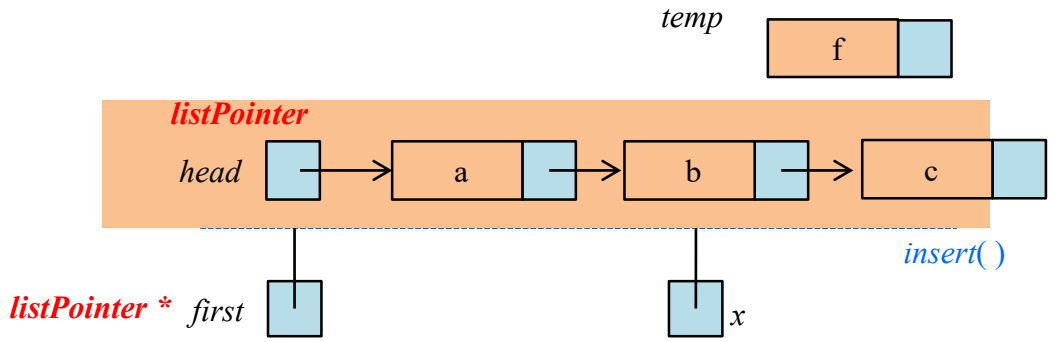
*first

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

Example 4.3 [List insertion]

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

```
void insert(listPointer *first, listPointer x)
```



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

노드 b 뒤에 노드 f를 삽입

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

*first

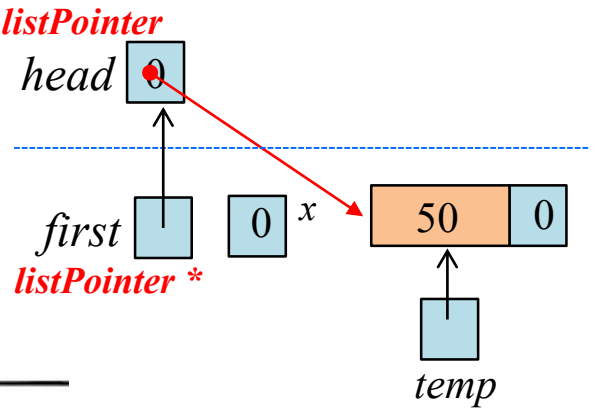
			first		head																
			30		70			100			150				200				300		
			70		**100**			a	200		f	300			b	**150**			c	0	

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

insert(&head, NULL)



Program 4.2: Simple insert into **list**

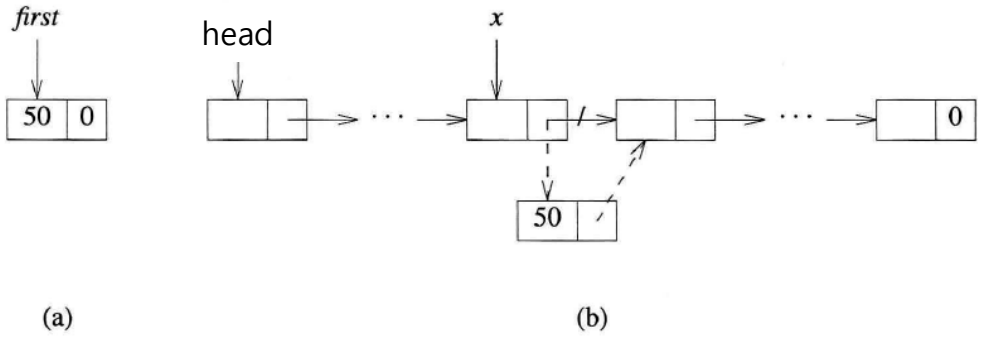


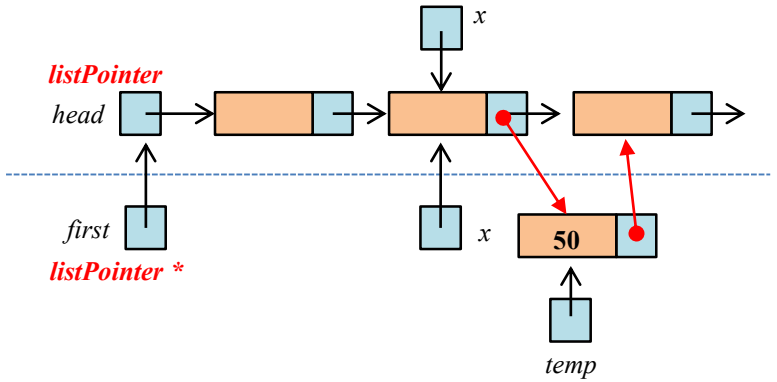
Figure 4.7: Inserting into an empty and nonempty list

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

insert(&first, x)



Program 4.2: Simple insert into **list**

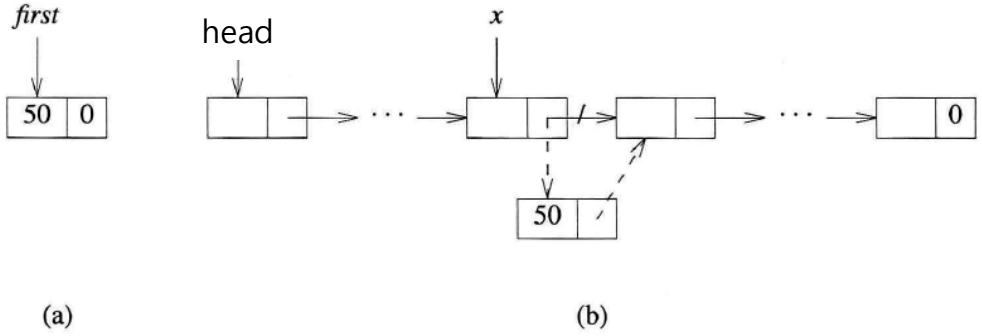
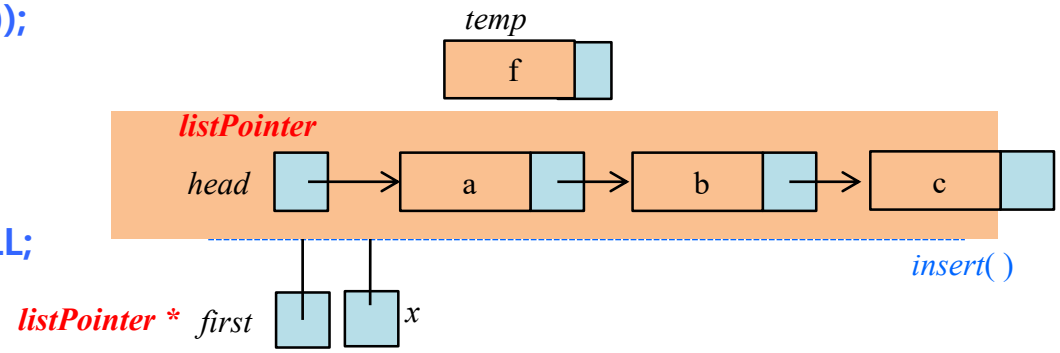


Figure 4.7: Inserting into an empty and nonempty list

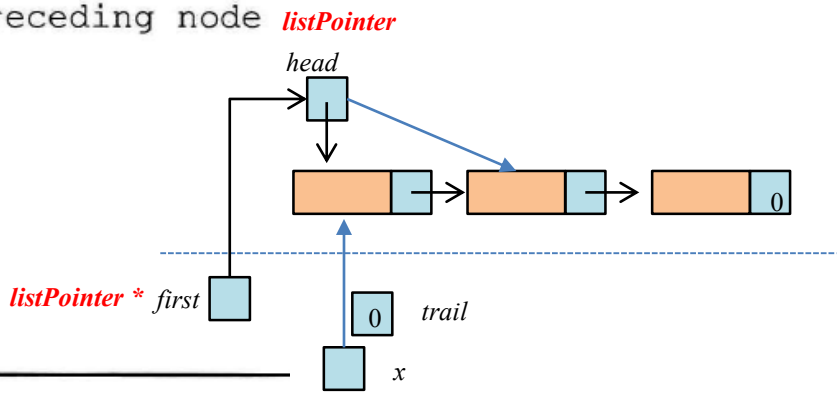
if (x == NULL) ???

```
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;
                }
        }
}
```



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);
}
```



Program 4.3: Deletion from a list

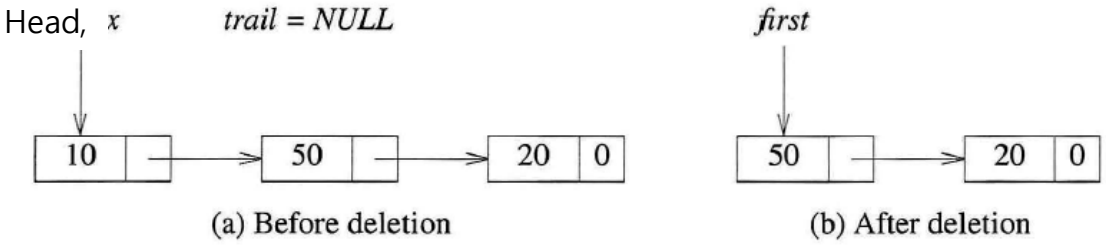
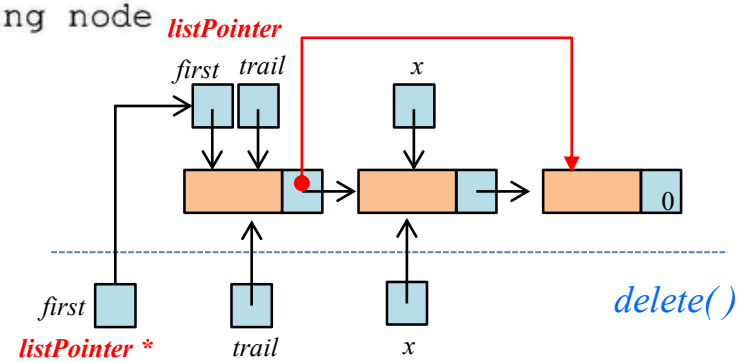


Figure 4.8: List before and after the function call *delete(&head, trail, x)*

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);
}
```



Program 4.3: Deletion from a list

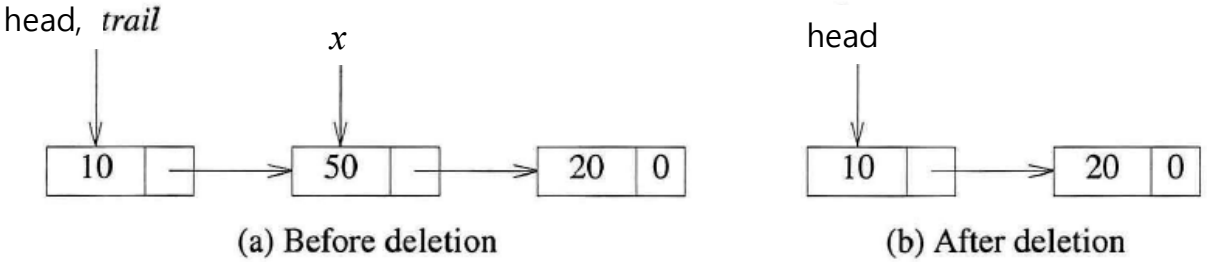


Figure 4.9: List after the function call *delete(&head, trail, x)*

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");
}
```

Program 4.4: Printing a list

printList(first)