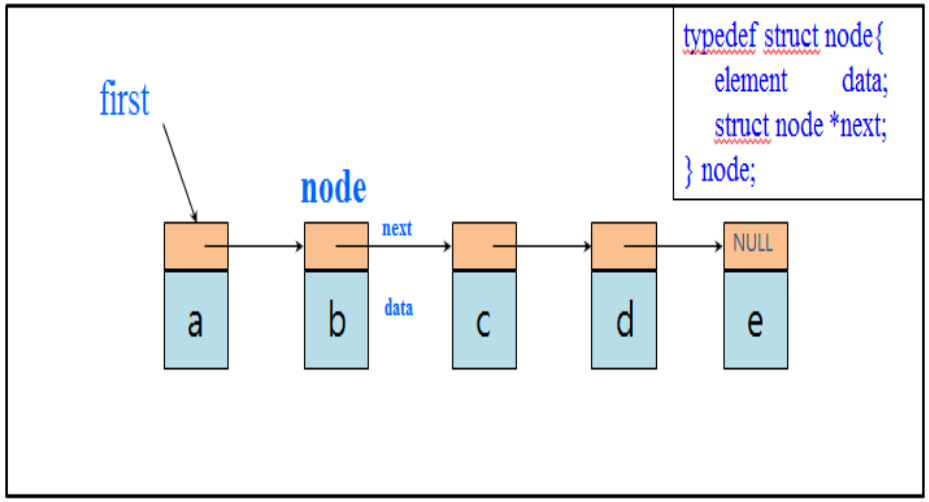
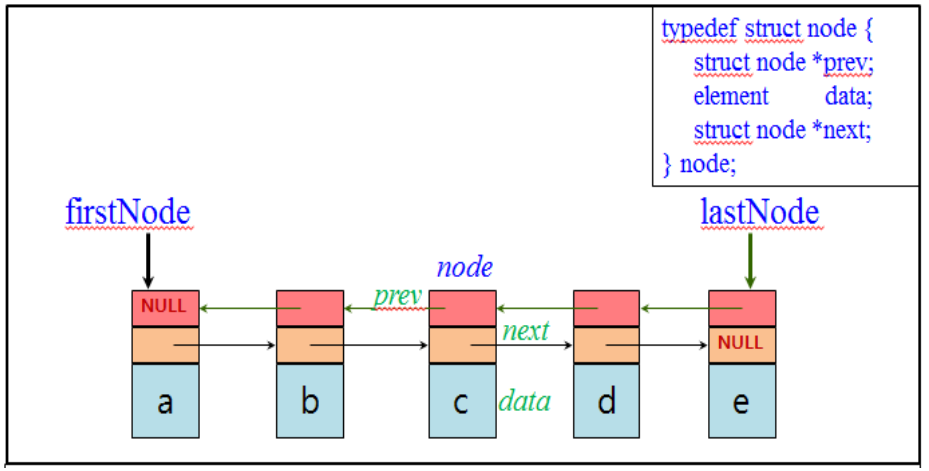
Chap 4. Linked Lists

Contents

Singly Linked Lists



Doubly Linked Lists



4.1 Singly Linked Lists and Chains

- A ***list*** or ***sequence*** is an abstract data type that implements a finite ordered collection of values, where the same value may occur more than once.

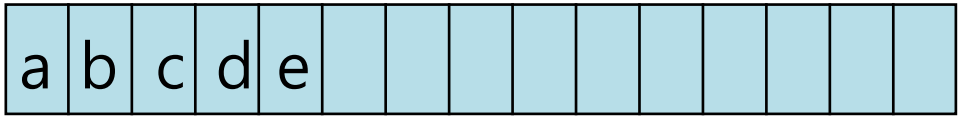
- Each instance of a value in the list is usually called an ***item***, ***entry***, or ***element*** of the list; if the same value occurs multiple times, each occurrence is considered a distinct item.

- ***Lists*** are typically implemented either as ***linked lists*** (either singly or doubly linked) or as ***arrays***, usually variable length or dynamic arrays.

List L = (a, b, c, d, e)

(1) Sequential Representation

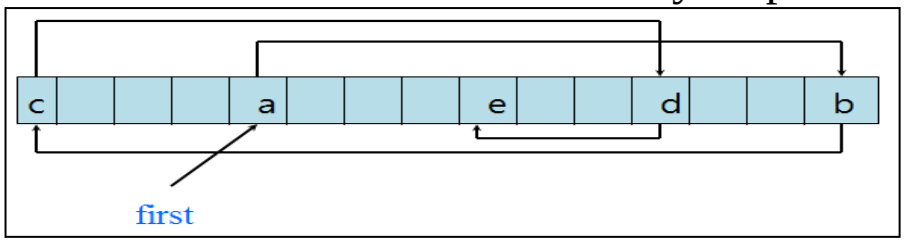
- using array and a sequential mapping

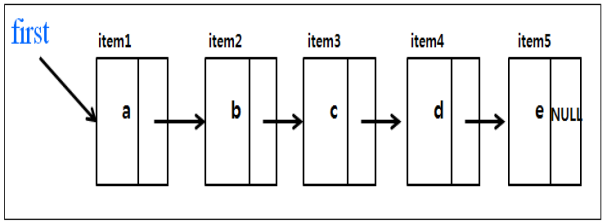


- *order of elements is the same as in the ordered list*
- insertion and deletion of arbitrary elements from ordered list become expensive
  - *excessive data movement.*

(2) Linked Representation

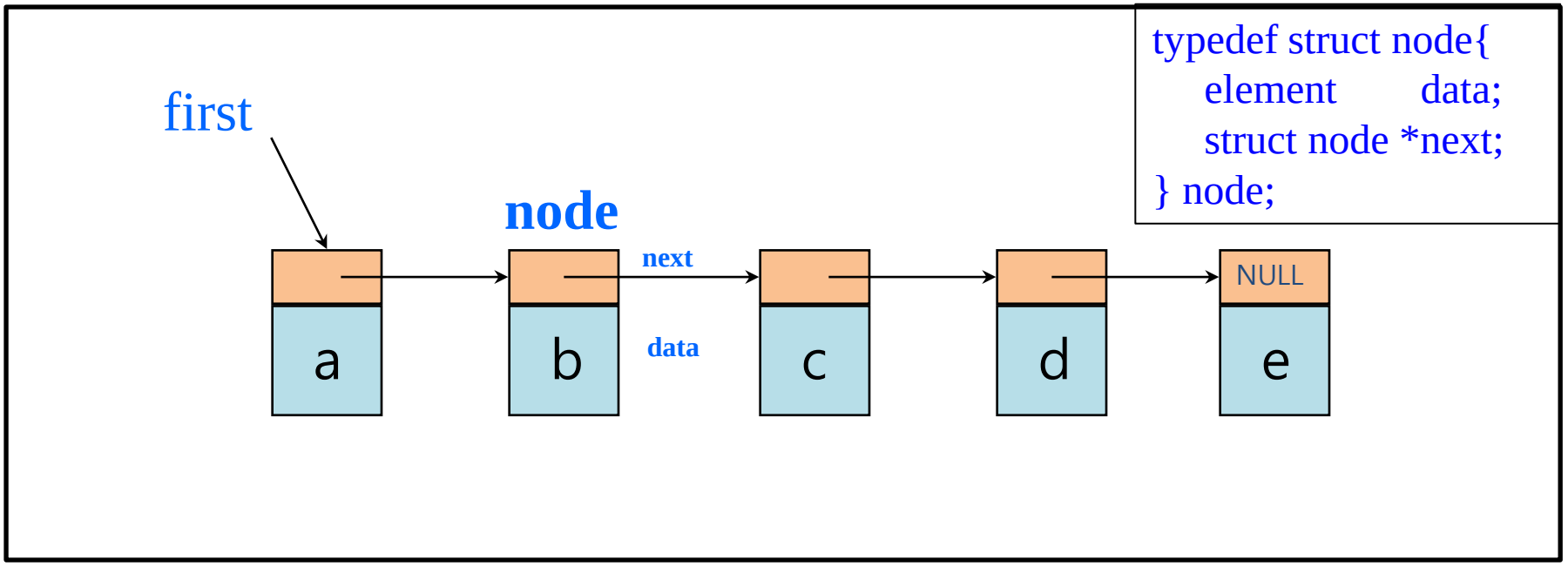
- successive items of a list may be placed anywhere in memory





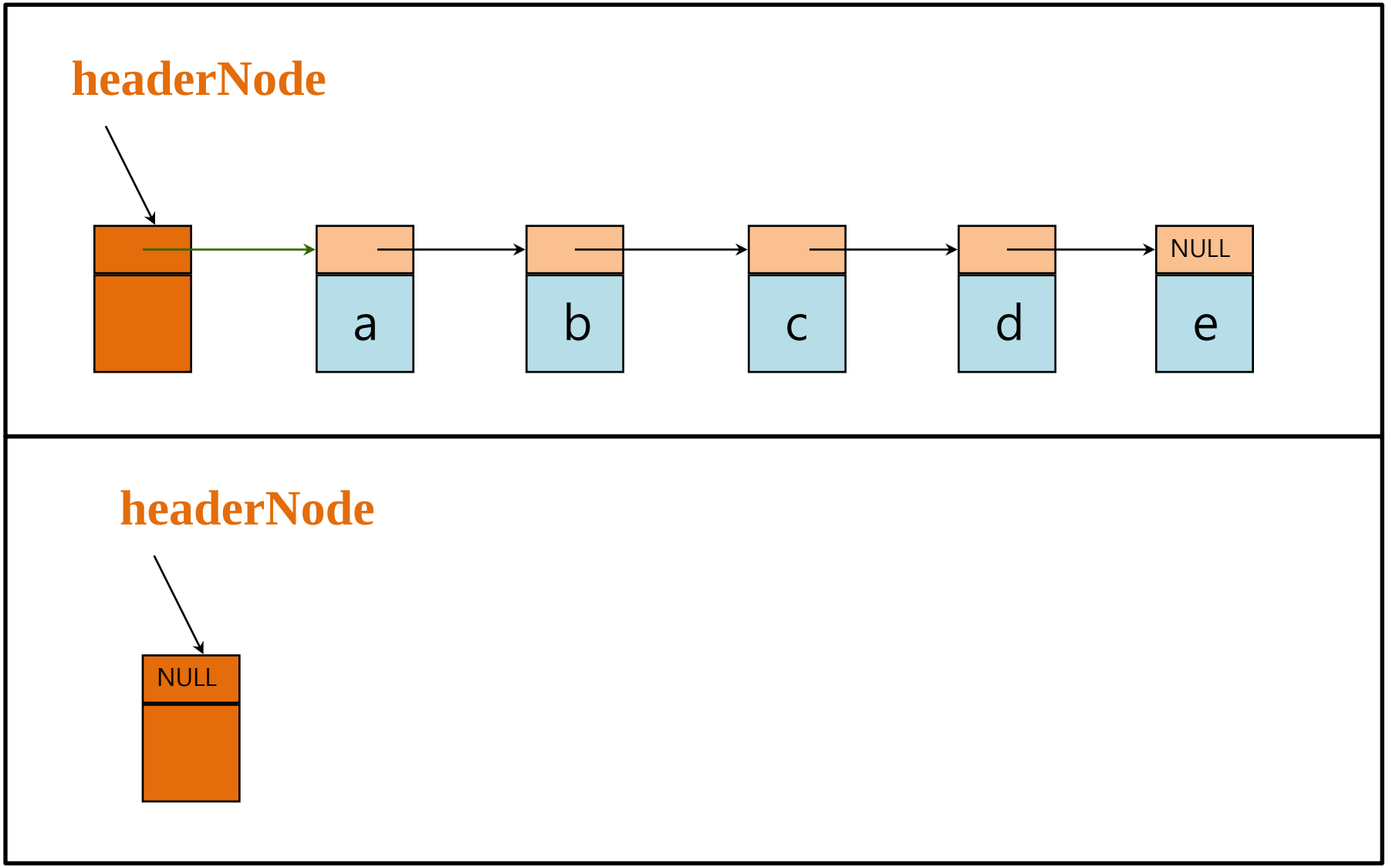
- *order of elements need not be the same as order in list*
  - use a variable ***first*** to get to the first element ***a***
  - each data item is associated with a pointer (link) to the next item
  - pointer (or link) in ***e*** is **NULL**

Normal Way To Draw A Linked List



- A *chain* is a singly linked list that is comprised of zero or more nodes.

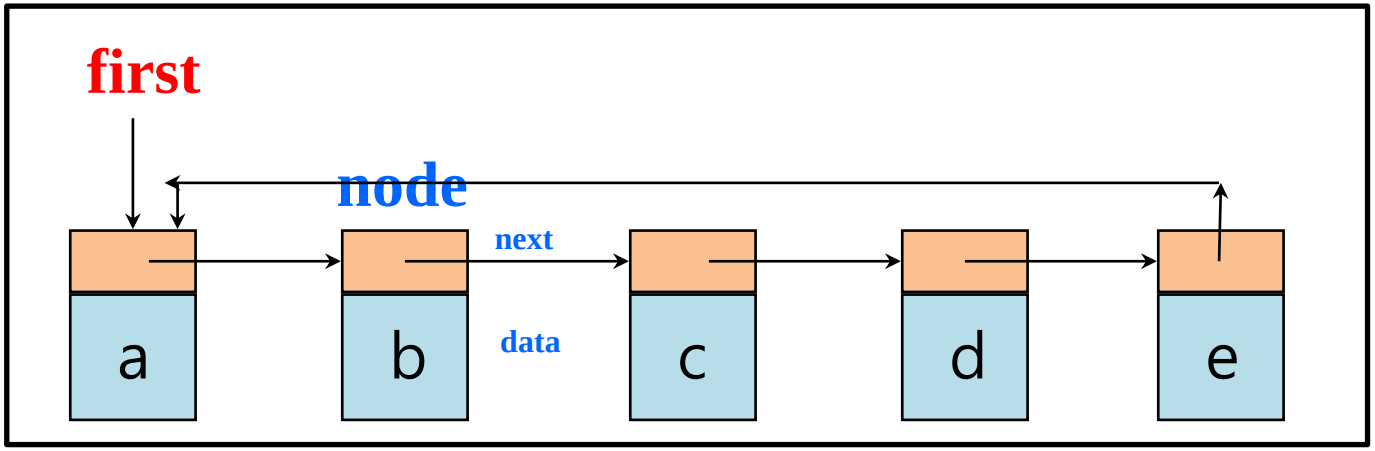
Chain With a Header Node



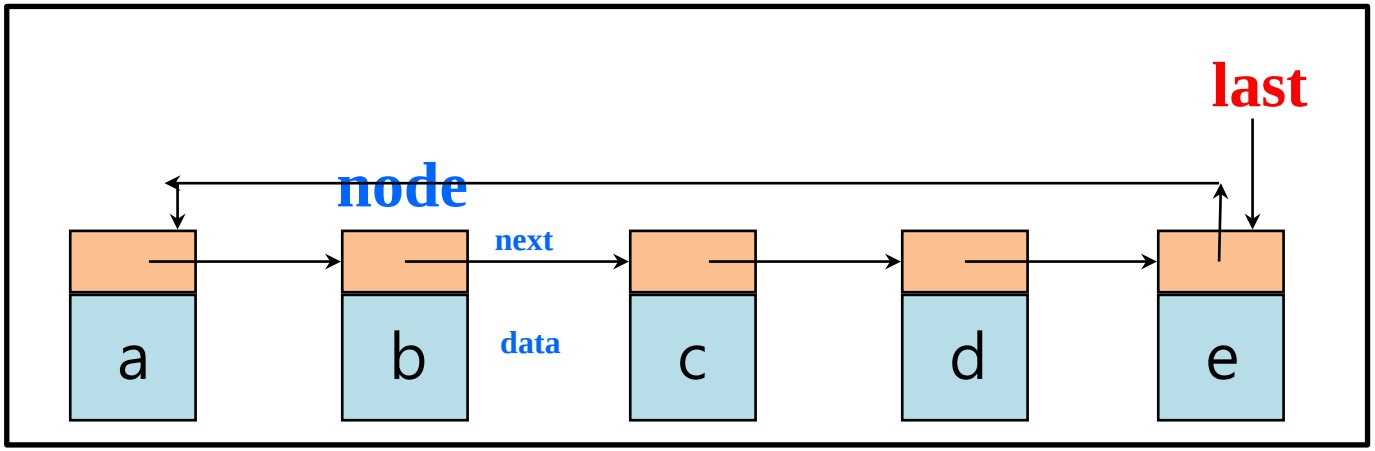
Singly Linked Circular lists

The link field of the last node points to the first node in the list.

- 방법 1



- 방법 2



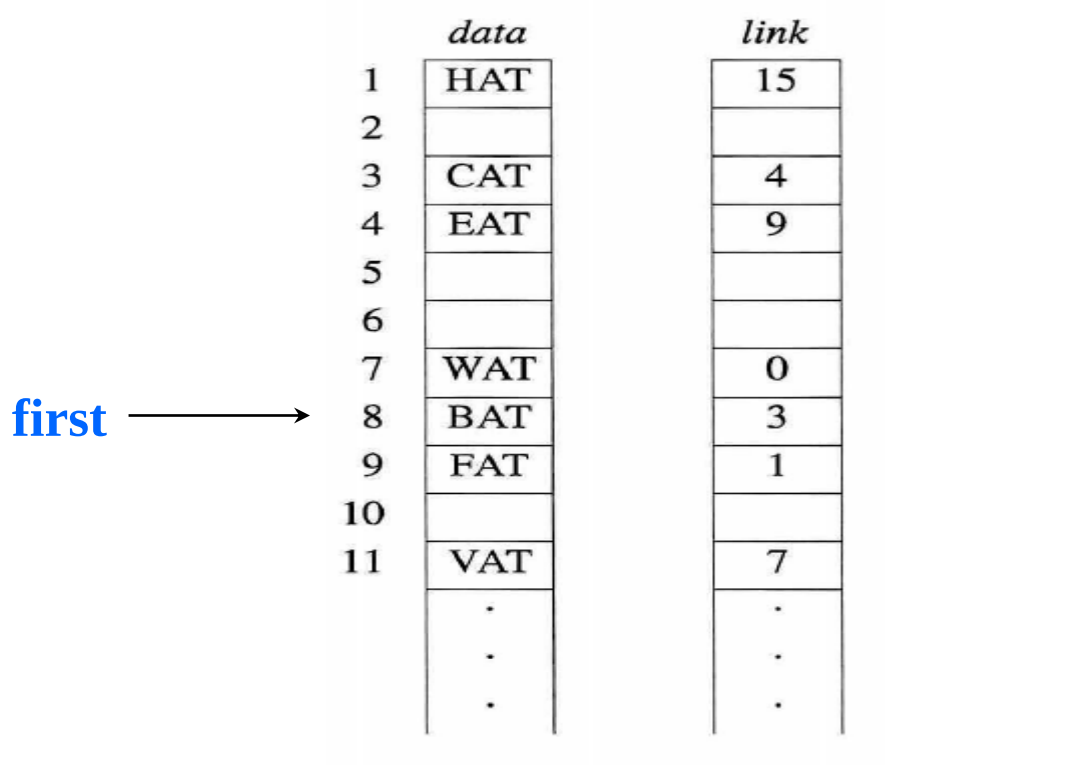


Figure 4.1: Nonsequential list-representation

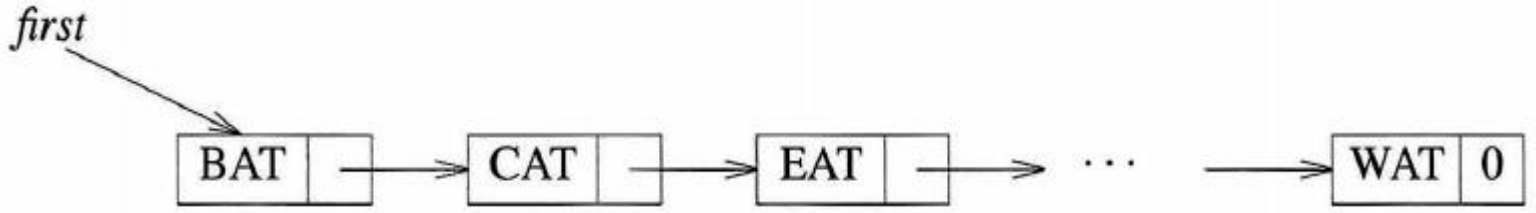


Figure 4.2: Usual way to draw a linked list

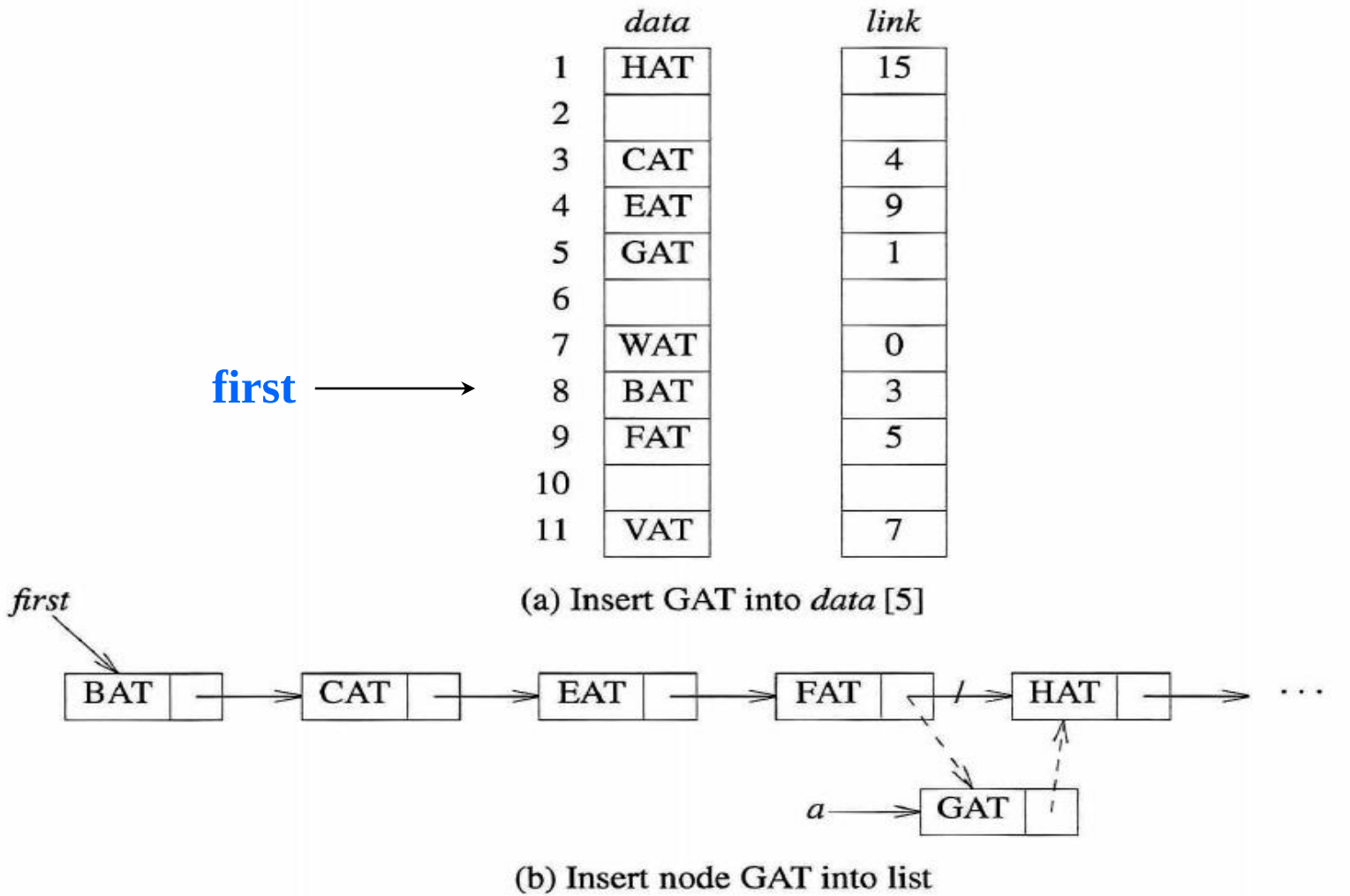


Figure 4.3: Inserting into a linked list

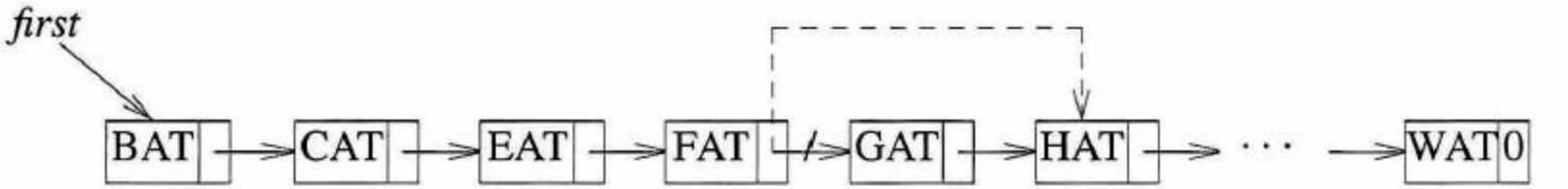


Figure 4.4: Delete GAT

4.2 Representing Chains in C

We need the following capabilities to make linked representations possible:

(1) A mechanism for defining a **node**'s structure, that is, the fields it contains. We use ***self-referential structures*** to do this.

(2) A way to create new nodes when we need them. The ***MALLOC*** macros defined in Section 1.2.2 handles this operation.

(3) A way to remove nodes that we no longer need. The ***free*** function handles this operation.

We will present several small examples to show how to create and use linked lists in C.

배열을 이용한 Singly Linked List

(1) 구조체 정의

```
typedef struct {
        int id;
        char name[20];
        char address[100];
} element;
typedef struct node{
        element   data;
        int       next;
} node;
```

(2) Creation of a list space

```
node list_array[MAX_LIST_SIZE];
int avail = -1;
```

(3) Creation of a new empty list

```
int first = -1;
```

(4) Test for an empty list

```
#define IS_EMPTY(first)    (first == -1)
```

(5) List space의 초기화

(6) Node의 할당과 반환

(7) Node의 연결

0				1
1				2
2				3
3				4
4				5
5				6
6				7
7				-1

first = -1

avail = 0

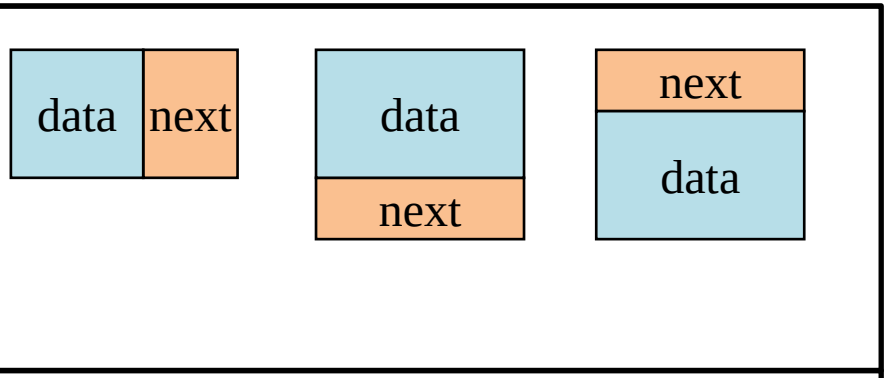
0	30	일지매	대구 수성구 지산동	2
1	60	홍길동	부산 동래구 수정동	-1
2	40	춘향	전남 남원	3
3	50	박지성	경기도 수원	1
4				5
5				6
6				7
7				-1

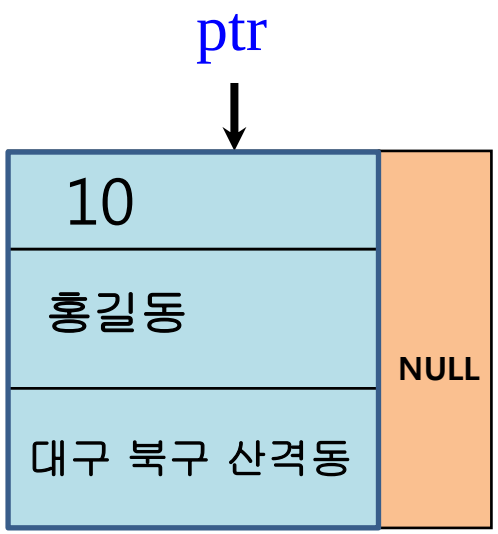
first = 0

avail = 4

동적할당을 이용한 Singly Linked List

```
(1) 구조체 정의
typedef struct {
    int id;
    char name[20];
    char address[100];
} element;
typedef struct node *nodePointer;
typedef struct node {
    element         data;
    nodePointer     next;
};
(2) Creation of a new empty list
nodePointer   first = NULL;
(3) Test for an empty list
#define IS_EMPTY(first)    (! (first) )
(4) Creation of a new node for the list
MALLOC(first, sizeof(*first) );
MALLOC(first, sizeof(struct node));
(5) Assigning values to the fields of the node
ptr->data.id = 10;
strcpy(ptr->data.name, "홍길동");
strcpy(ptr->data.address, "대구 북구 산격동");
ptr->next = NULL;
```





동적할당을 이용한 Singly Linked List

(1) 구조체 정의

```
typedef struct node *nodePointer;
typedef struct node {
        int             id;
        char            name[20];
        char            address[100];
        nodePointer     next;
};
```

(2) Creation of a new empty list

```
nodePointer   first = NULL;
```

(3) Test for an empty list

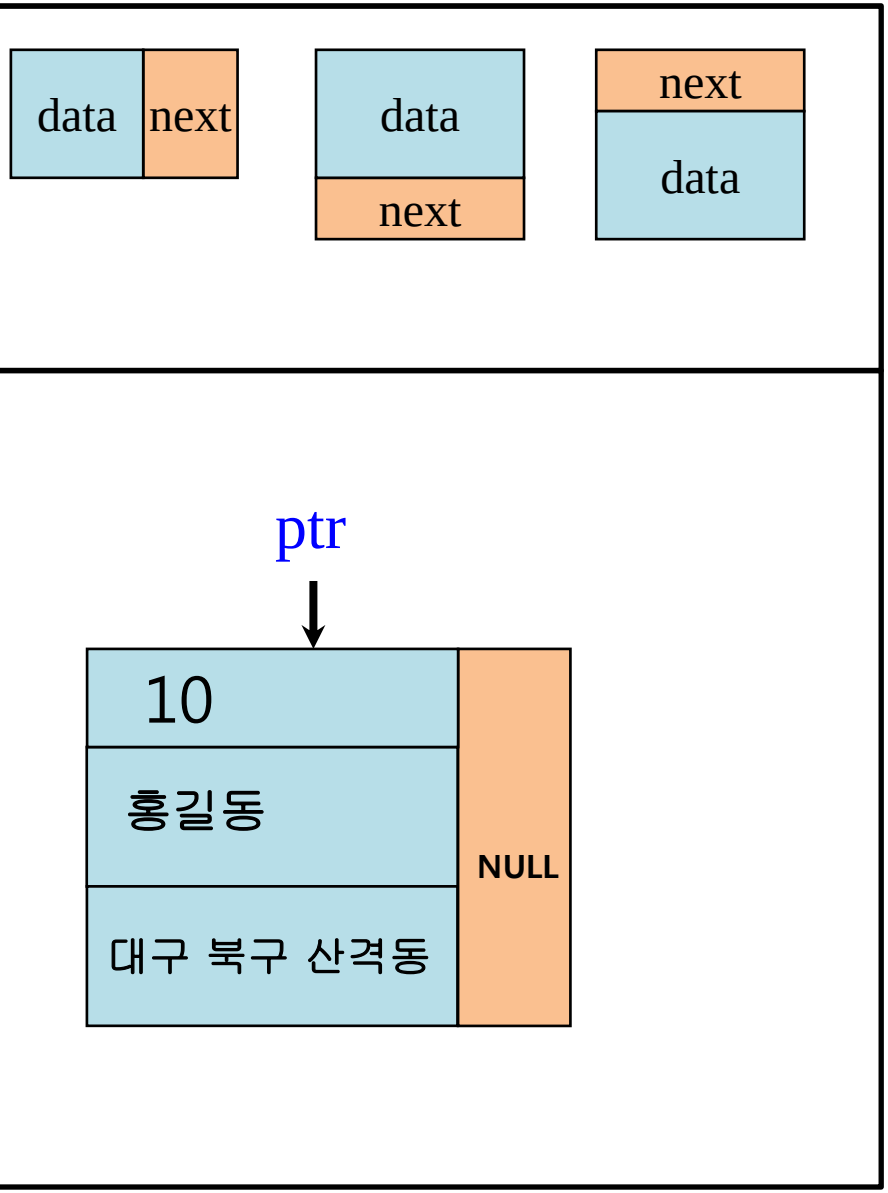
```
#define IS_EMPTY(first)    (! (first) )
```

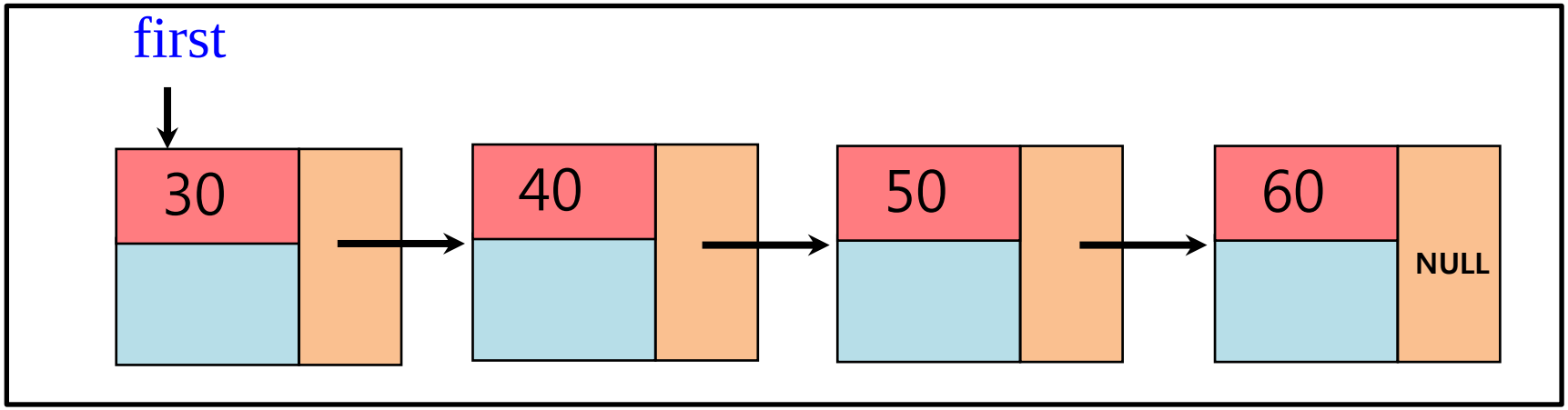
(4) Creation of a new node for the list

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

(5) Assigning values to the fields of the node

```
ptr->id = 10;
strcpy(ptr->name, "홍길동");
strcpy(ptr->address, "대구 북구 산격동");
ptr->next = NULL;
```






```
…
add(30);
add(60);
add(40);
add(50);

printList(first);
delete(&first, 50);
delete(&first, 30);
printList(first);
…
```

```
nodePointer allocate_node()
{
    nodePointer ptr;
    MALLOC(ptr, sizeof(node));
    return ptr;
}
void initialize_node(nodePointer current, int id)
{
    current->data.id = id;
    current->next = NULL;
}
void add(int id)
{
    nodePointer prev, current;
    current = allocate_node();
    initialize_node(current, id);
    prev = find_prev_node(first, current);
    insert(&first, prev, current);
}
```

```
/* find node prev to put the node current after that  in the chain first */
nodePointer  find_prev_node(nodePointer first, nodePointer current)
{
    nodePointer  prev, search;
    prev = NULL;
    search = first;
    while (1) {
            if (search == NULL) break;
            if (search->data.id <=  current->data.id) break;
            prev = search;
            search = search->next;
    }
    return prev;
}
/* insert current into the chain first after node prev */
void insert(nodePointer *first, nodePointer prev, nodePointer current)
{
    if (*first) {
                if (prev == NULL) { current->next = *first; *first = current; }
                else { current->next = prev->next;    prev->next = current; }
    } else {    *first = current;     }
}
/ * delete current from the list, prev is the preceding node and *first is the front of the list * /
void delete(nodePointer *first, nodePointer prev, nodePointer current)
{
    if (prev) prev->next = current->next;
    else *first = current->next;
    free(current);
}
```

```
void printList(nodePointer current)
{
    printf("The list contains: ");
    for (; current; current = current->next)
        printf("%4d", current->data.id);
    printf(" \n");
}
```

과제 1 (singly linked list)

[1] 하나의 배열을 이용하며 그 배열의 각 엔트리는 하나의 **node**의 모든 필드들을 다 가지는 **structured data type**을 가지도록 하여 **singly linked list**를 형성하도록 함으로써, 아래 기능을 수행하는 **C** 프로그램을 작성하시오. 단, 각 **node**는 삽입시에 배열로 부터 할당되어야 하며 <학번, 이름, 주소>를 가지며 그 리스트에서 학번의 증가순으로 정렬된 위치에 삽입되어야 한다.

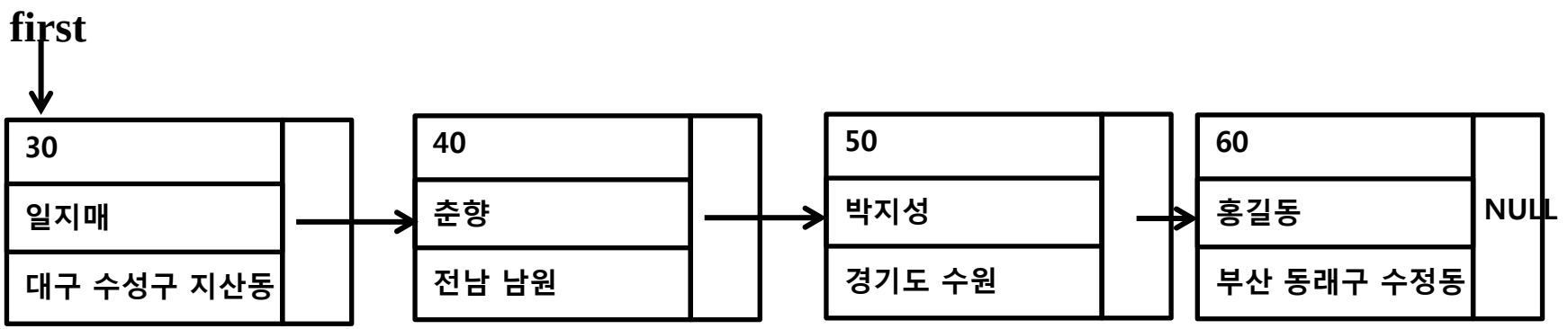
[2] 동적 할당으로 생성되는 각 **node**는 필요한 모든 필드들을 다 가지는 **structured data type**을 가지도록 하여 **singly linked list**를 형성하도록 함으로써, 아래 기능을 수행하는 **C** 프로그램을 작성하시오. 단, 각 **node**는 삽입시에 동적으로 할당되어야 하며 <학번, 이름, 주소>를 가지며 그 리스트에서 학번의 증가순으로 정렬된 위치에 삽입되어야 한다.

```
i(삽입), d(삭제), f(탐색), r(전체 읽기), q(작업종료)를 선택하시오: i
삽입할 자료(학번, 이름, 주소)를 입력하시오: 30  일지매 대구 수성구 지산동
삽입할 자료(학번, 이름, 주소)를 입력하시오: 60  홍길동 부산 동래구 수정동
삽입할 자료(학번, 이름, 주소)를 입력하시오: 40  춘향 전남 남원
삽입할 자료(학번, 이름, 주소)를 입력하시오: 50  박지성 경기도 수원
삽입할 자료(학번, 이름, 주소)를 입력하시오: ^z
i(삽입), d(삭제), f(탐색), r(전체 읽기), q(작업종료)를 선택하시오: f
탐색할 자료의 학번을 입력하시오: 40
<학번, 이름, 주소> = <40, 춘향, 전남 남원>
i(삽입), d(삭제), f(탐색), r(전체 읽기), q(작업종료)를 선택하시오: r
<학번, 이름, 주소> : <30, 일지매, 대구 수성구 지산동>, <40, 춘향, 전남 남원>, <50, 박지성, 경기도 수원>, <60, 홍길동, 부산 동래구 수정동>
 i(삽입), d(삭제), f(탐색), r(전체 읽기), q(작업종료)를 선택하시오: d
삭제할 자료의 학번을 입력하시오: 20
<학번, 이름, 주소> = <20, null, null>은 없는 자료입니다.
i(삽입), d(삭제), f(탐색), r(전체 읽기), q(작업종료)를 선택하시오: d
삭제할 자료의 학번을 입력하시오:40
<학번, 이름, 주소> = <40, 춘향, 전남 남원>이 삭제되었습니다.
i(삽입), d(삭제), f(탐색), r(전체 읽기), q(작업종료)를 선택하시오: r
<학번, 이름, 주소> : <30, 일지매, 대구 수성구 지산동>, <50, 박지성, 경기도 수원>, <60, 홍길동, 부산 동래구 수정동>
i(삽입), d(삭제), f(탐색), r(전체 읽기), q(작업종료)를 선택하시오: q
```

0	30	일지매	대구 수성구 지산동	2
1	60	홍길동	부산 동래구 수정동	-1
2	40	춘향	전남 남원	3
3	50	박지성	경기도 수원	1
4				5
5				6
6				7
7				-1

first = 0

avail = 4



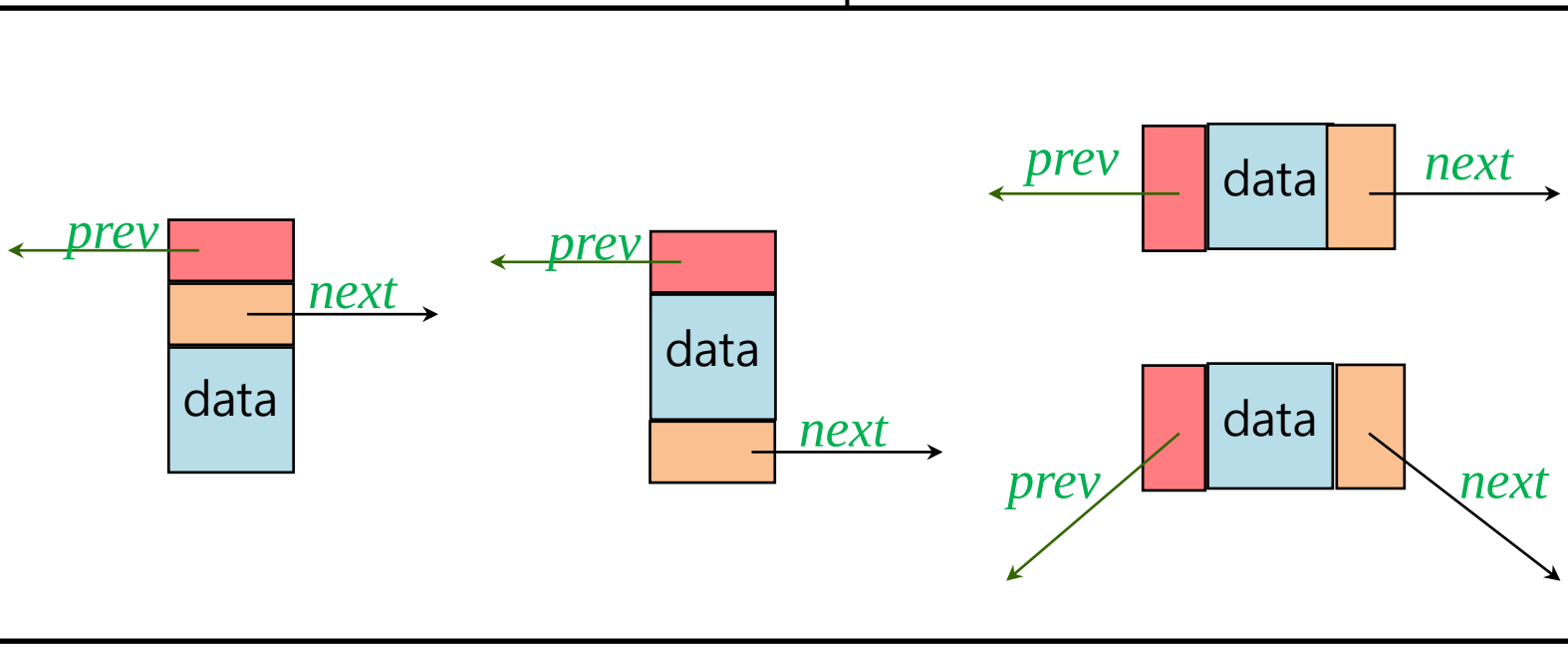
```
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  r
The list contains:
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  f
탐색할 자료의 학번을 입력하시오: 40
탐색할 학번 40가 존재하지 않음.
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  d
삭제할 자료의 학번을 입력하시오: 40
삭제할 학번 40가 존재하지 않음.
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  i
id name address: 30 일지매 대구 수성구 지산동
id name address: 60 홍길동 부산 동래구 수정동
id name address: 40 춘향 전남 남원
id name address: 50 박지성 경기도 수원
id name address: 10 메시 스페인 FC 바르셀로나
id name address: ^Z
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  r
The list contains:
  10 메시   스페인 FC 바르셀로나
  30 일지매 대구 수성구 지산동
  40 춘향   전남 남원
  50 박지성 경기도 수원
  60 홍길동 부산 동래구 수정동
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  f
탐색할 자료의 학번을 입력하시오: 40
  40 춘향   전남 남원
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  f
탐색할 자료의 학번을 입력하시오: 20
탐색할 학번 20가 존재하지 않음.
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  d
삭제할 자료의 학번을 입력하시오: 40
삭제:   40 춘향   전남 남원
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  d
삭제할 자료의 학번을 입력하시오: 40
삭제할 학번 40가 존재하지 않음.
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  r
The list contains:
  10 메시   스페인 FC 바르셀로나
  30 일지매 대구 수성구 지산동
  50 박지성 경기도 수원
  60 홍길동 부산 동래구 수정동
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  i
id name address: 20 차두리 서울 FC 서울
id name address: 80 이순신 조선 선조 임진왜란
id name address: ^Z
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  r
The list contains:
  10 메시   스페인 FC 바르셀로나
  20 차두리 서울 FC 서울
  30 일지매 대구 수성구 지산동
  50 박지성 경기도 수원
  60 홍길동 부산 동래구 수정동
  80 이순신 조선 선조 임진왜란
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  q
계속하려면 아무 키나 누르십시오 . . .
```

```
30 일지매 대구 수성구 지산동
60 홍길동 부산 동래구 수정동
40 춘향 전남 남원
50 박지성 경기도 수원
10 메시 스페인 FC 바르셀로나

20 차두리 서울 FC 서울
80 이순신 조선 선조 임진왜란
```

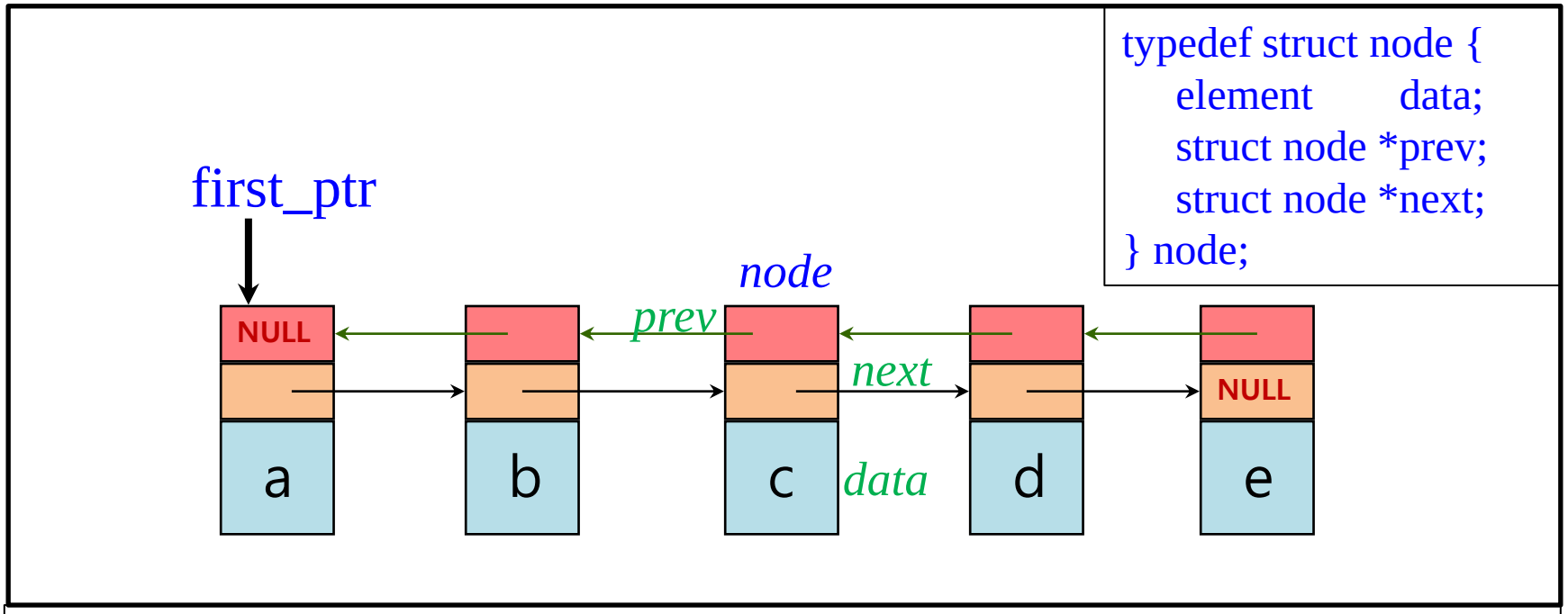
4.8 DOUBLY LINKED LISTS

- Limitation in chains and singly linked circular lists
- The only way to find the node that precedes a node *p* is to start at the beginning of the list.
- Easy deletion of an arbitrary node requires knowing the preceding node.
  - see Example 4.4[*List Deletion*].
- It is useful to have doubly linked lists.

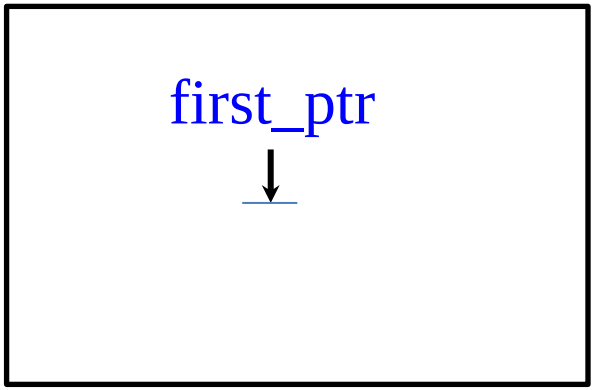
```
typedef struct {
        int id;
        char name[20];
        char address[100];
} element;
typedef struct node *nodePointer;
typedef struct node {
        nodePointer prev;
        element data;
        nodePointer next;
} ;
```



Doubly Linked List

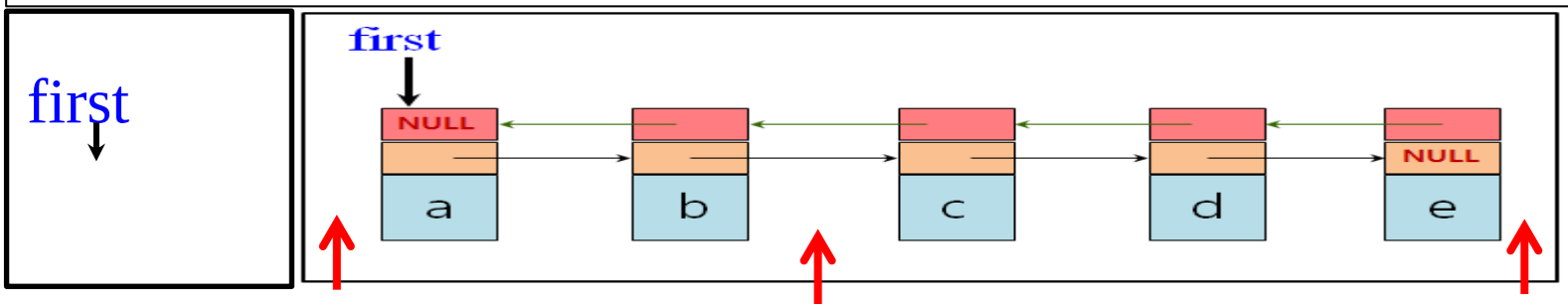


일반적으로 **Doubly Linked List**라 함은 이 구조를 의미한다.



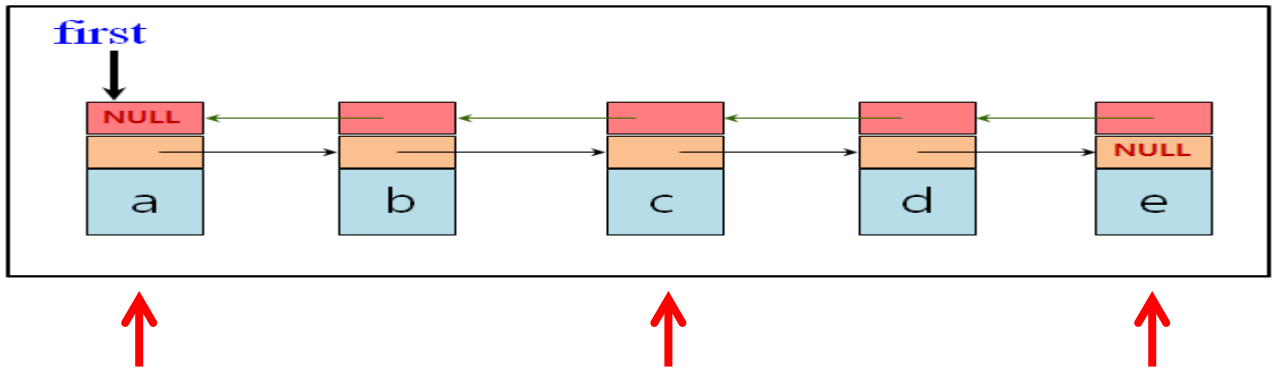
// insert *currentNode* to the right of *prevNode*

```
void duble_insert(nodePointer prev, nodePointer current)
{
     if (first != NULL) {
        if (prev == NULL) {
             current->next = first;
             first->prev = current;
             first = current;
        } else {
             current->prev = prev;
             current->next = prev->next;
             if (prev->next != NULL) prev->next->prev = current;
             prev->next = current;
        }
    } else {
        first = current;
    }
}
```

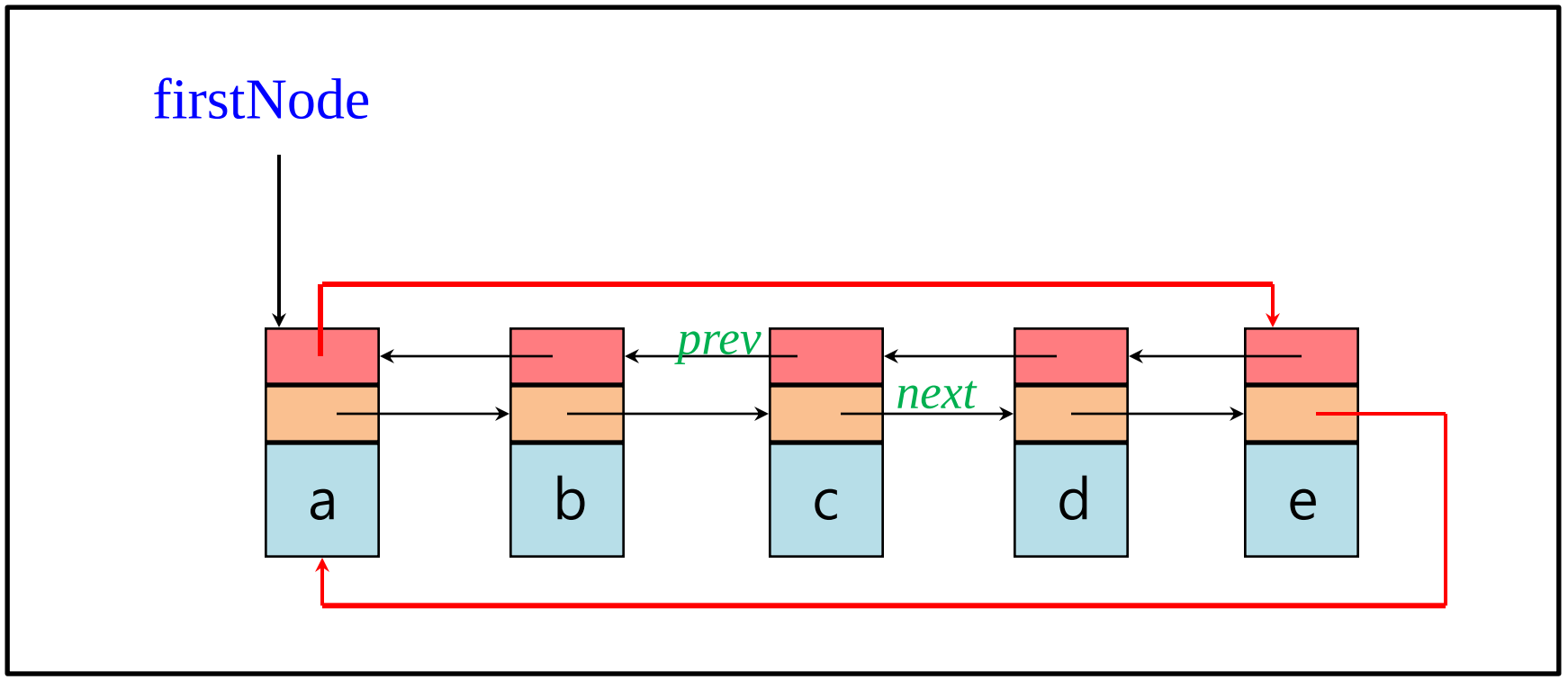


// delete *currentNode* from the right of *prevNode*

```
void duble_delete(nodePointer prev, nodePointer current)
{
     if (prev != NULL) {
            prev->next = current->next;
            if (current->next != NULL) current->next->prev = current->prev;
    } else {
            first = current->next;
            if (first != NULL) first->prev = NULL;
    }
    free(current);
}
```



Doubly Linked Circular List



Doubly Linked Circular List With Header Node

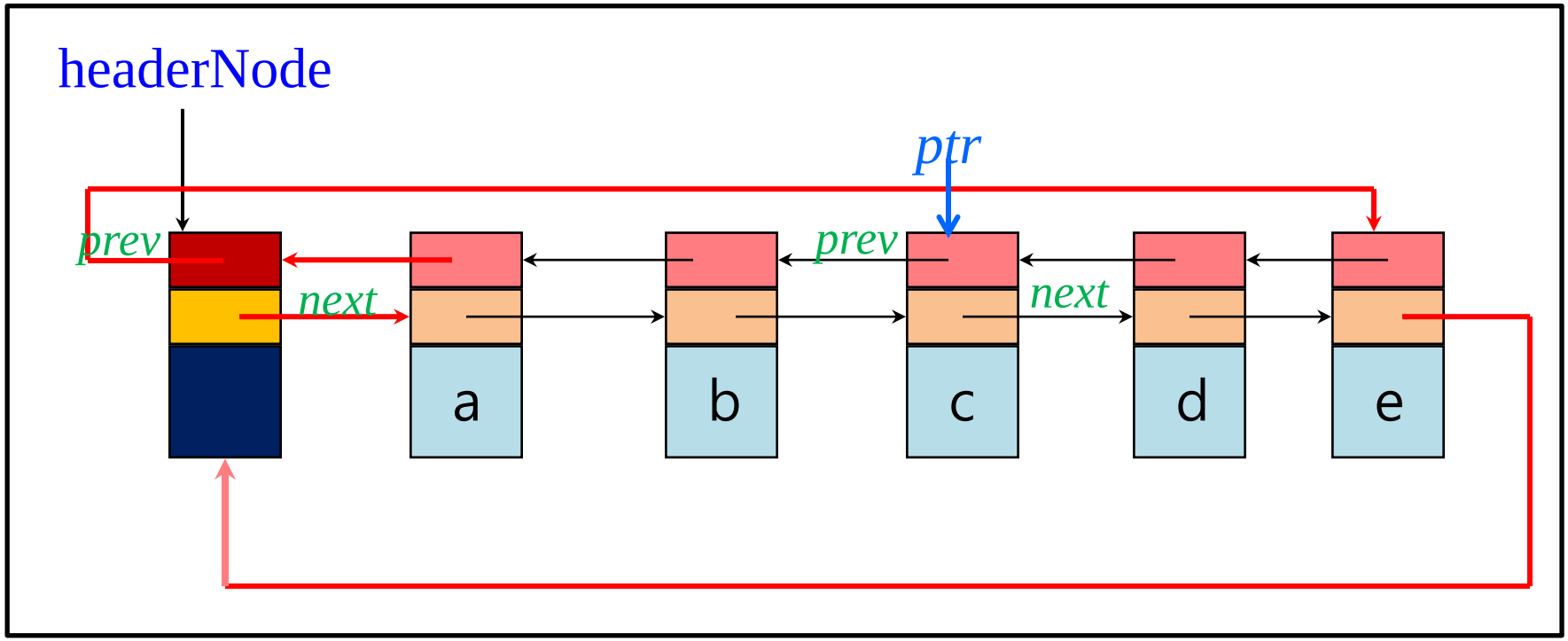
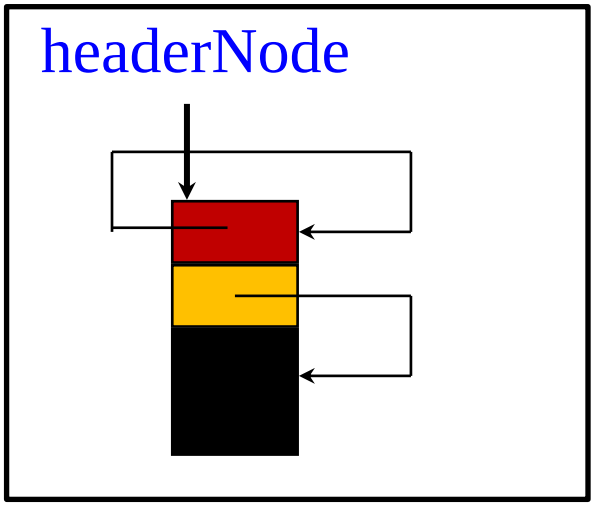


Figure 4.21: Doubly linked circular list with header node



ptr == ptr->prev->next
== ptr->next->prev

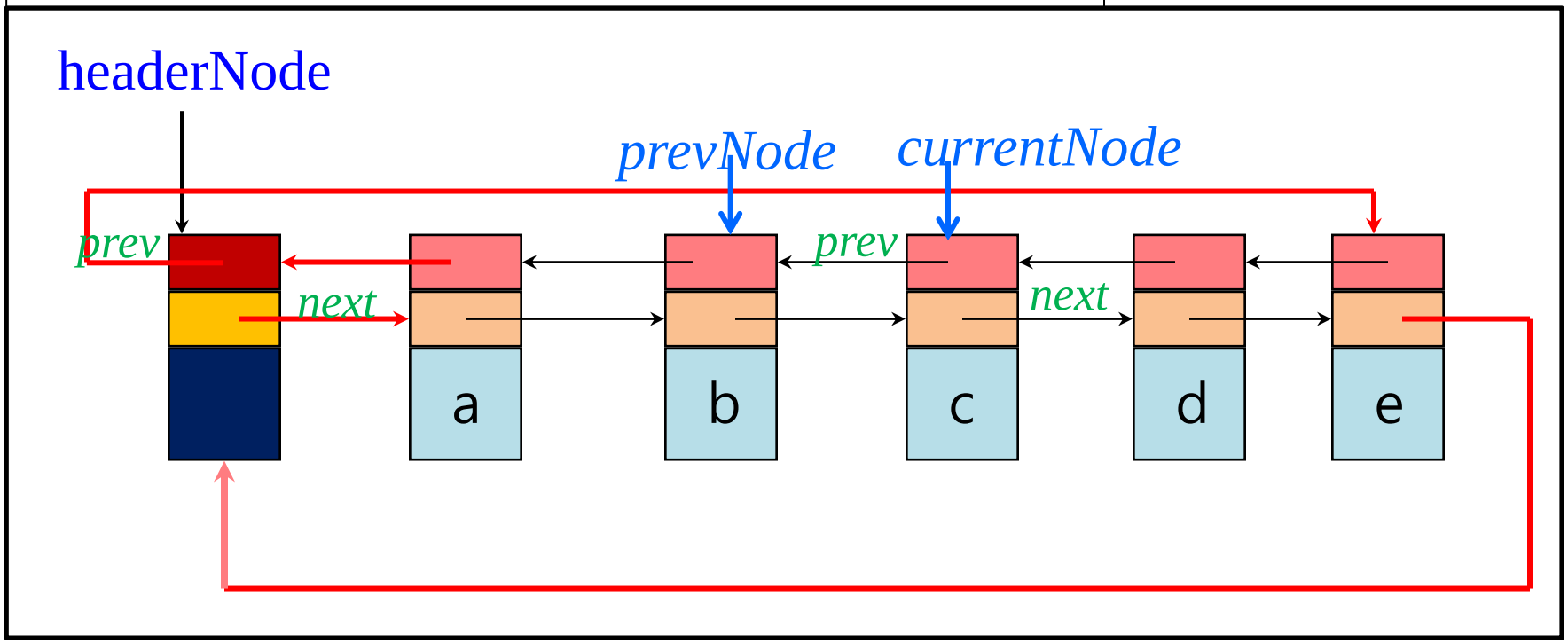
Figure 4.22: Empty doubly linked circular list with header node

insert *currentNode* to the right of *prevNode*

```
void dinsert(nodePointer prev, nodePointer current)
{/* insert currentNode to the right of prevNode */
        current->prev = prev;
        current->next = prev->next;
        prev->next->prev = current;
        prev->next = current;
}
```

Program 4.26: Insertion into a doubly linked circular list

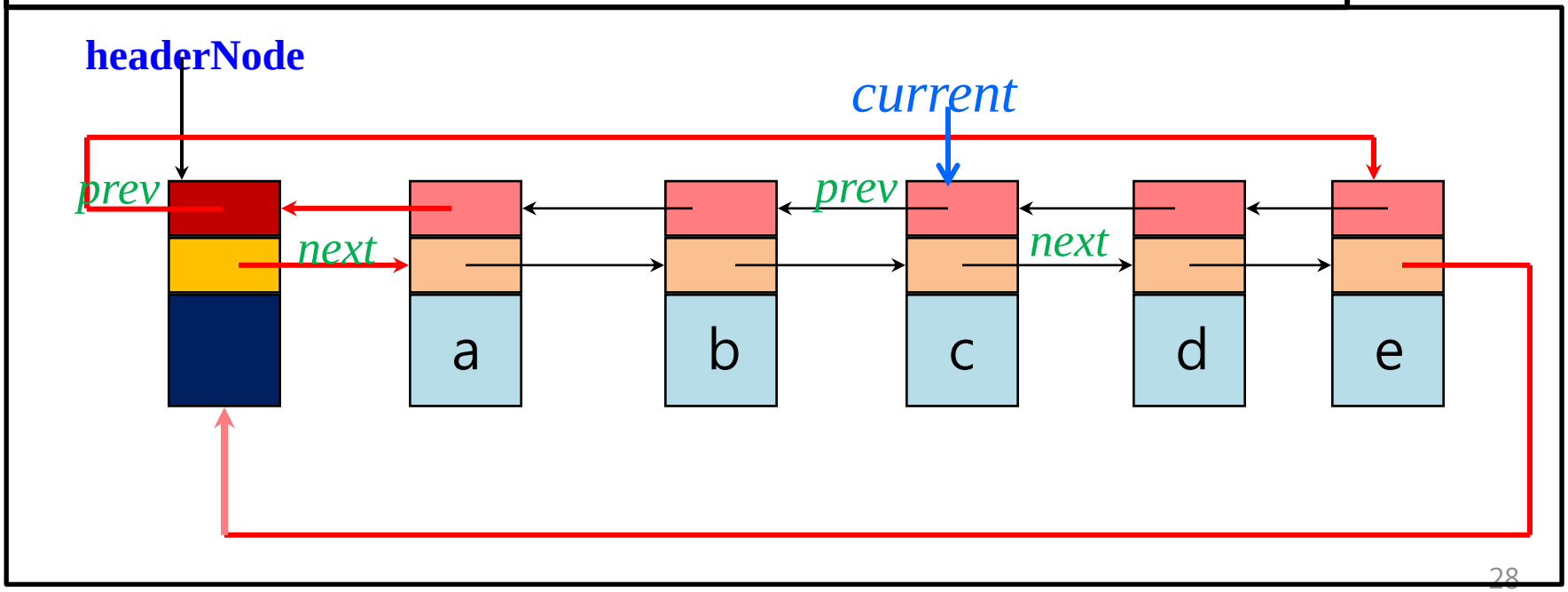
prevNode may be either a header node or an interior node in a list.

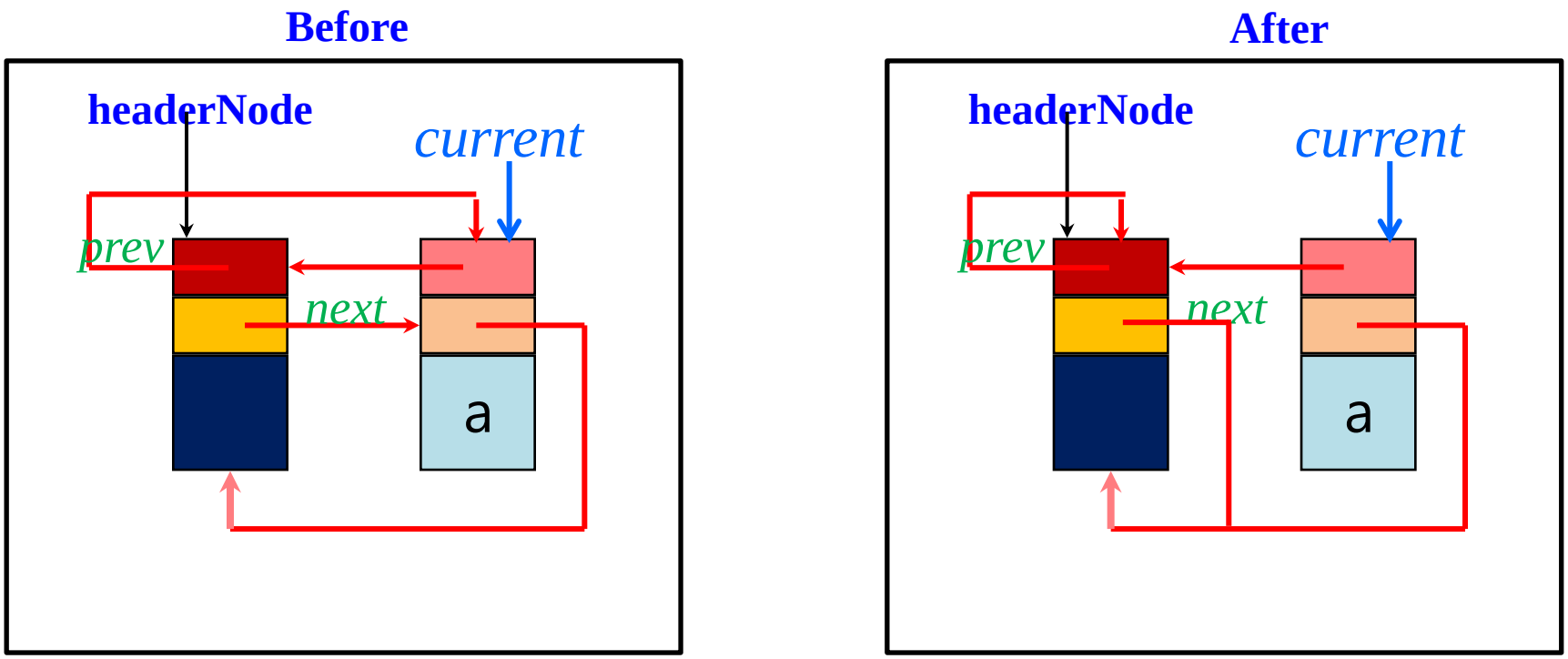


delete *currentNode* from the doubly linked list of the head node *headNode*

```
void ddelete(nodePointer headNode, nodePointer current)
{/* delete currentNode from the doubly linked list */
        if (headNode == current)
                printf("Deletion of header node is not permitted.\n");
        else {
                current->prev->next = current->next;
                current->next->prev = current->prev;
                free(current);
        }
}
```

Program 4.27: Deletion from a doubly linked circular list





Program 4.23: Deletion from a doubly linked circular list with a single node

배열을 이용한 Doubly Linked List

(1) 구조체 정의

```
typedef struct {
        int id;
        char name[20];
        char address[100];
} element;
typedef struct node{
        int     prev
        element data;
        int     next;
} node;
```

(2) Creation of a list space

```
node list_array[MAX_LIST_SIZE];
int avail = -1;
```

(3) Creation of a new empty list

```
int first = -1;
```

(4) Test for an empty list

```
#define IS_EMPTY(first)    (first == -1)
```

(5) List space의 초기화

(6) Node의 할당과 반환

(7) Node의 연결

0				0	1
1				0	2
2				0	3
3				0	4
4				0	5
5				0	6
6				0	7
7				0	-1

first= -1

avail = 0

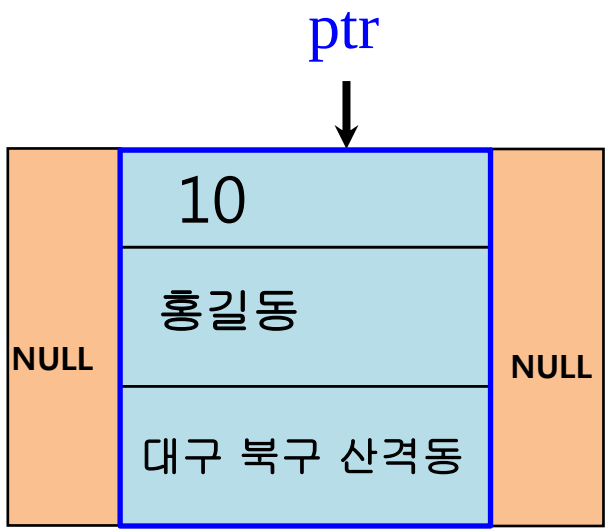
0	30	일지매	대구 수성구 지산동	-1	2
1	60	홍길동	부산 동래구 수정동	3	-1
2	40	춘향	전남 남원	0	3
3	50	박지성	경기도 수원	2	1
4				0	5
5				0	6
6				0	7
7				0	-1

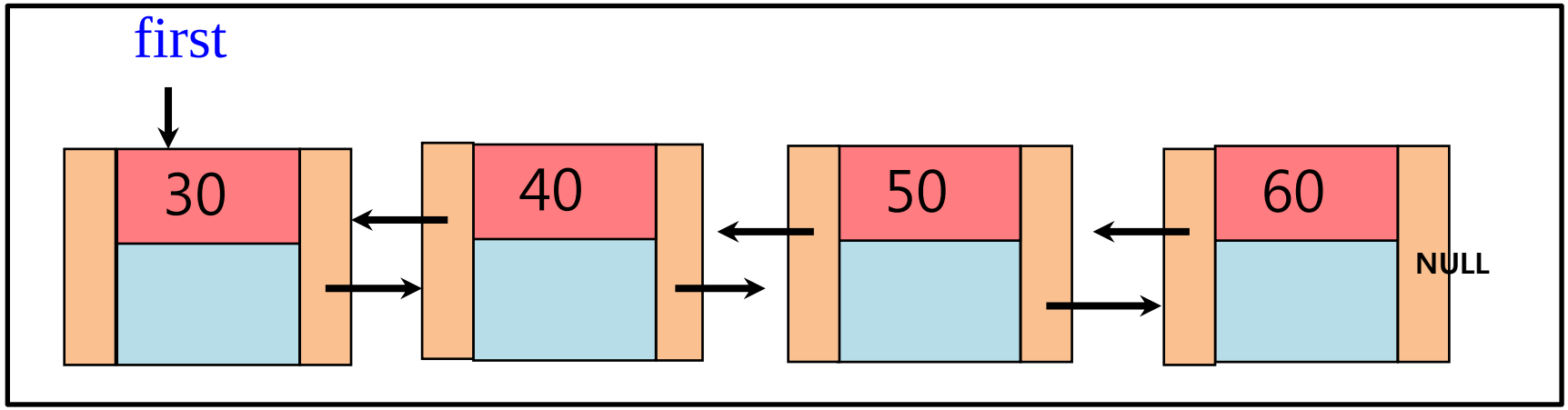
first = 0

avail = 4

동적할당을 이용한 Doubly Linked List

```
(1) 구조체 정의
typedef struct {
    int id;
    char name[20];
    char address[100];
} element;
typedef struct node *nodePointer;
typedef struct node {
    element         data;
    nodePointer     prev;
    nodePointer     next;
};
(2) Creation of a new empty list
nodePointer   first = NULL;
(3) Test for an empty list
#define IS_EMPTY(first)    (! (first) )
(4) Creation of a new node for the list
MALLOC(first, sizeof(*first) );
MALLOC(first, sizeof(struct node));
(5) Assigning values to the fields of the node
ptr->data.id = 10;
strcpy(ptr->data.name, "홍길동");
strcpy(ptr->data.address, "대구 북구 산격동");
ptr->prev = ptr->next = NULL;
```






```
…
add(30);
add(60);
add(40);
add(50);

printList(first);
delete(&first, 50);
delete(&first, 30);
printList(first);
…
```

```
nodePointer allocate_node()
{
    nodePointer current;
    MALLOC(current, sizeof(node));
    return current;
}
void initialize_node(nodePointer current, int id)
{
    current->data.id = id;
    current->prev = current->next = NULL;
}
void add(int id)
{
    nodePointer prev, current;
    current = allocate_node();
    initialize_node(current, id);
    prev = find_prev_node(first, current);
    insert(&first, prev, current);
}
```

과제 2 (doubly linked list)

[1] 하나의 배열을 이용하며 그 배열의 각 엔트리는 하나의 **node**의 모든 필드들을 다 가지는 **structured data type**을 가지도록 하여 **doubly linked list**를 형성하도록 함으로써, 아래 기능을 수행하는 **C** 프로그램을 작성하시오. 단, 각 **node**는 삽입시에 배열로 부터 할당되어야 하며 <학번, 이름, 주소>를 가지며 그 리스트에서 학번의 증가순으로 정렬된 위치에 삽입되어야 한다.

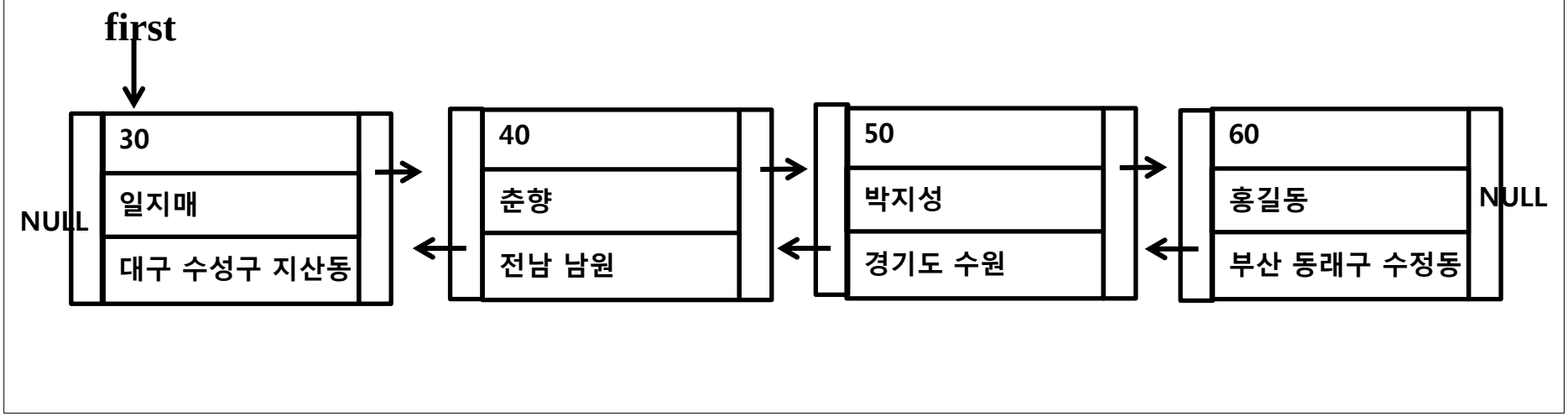
[2] 동적 할당으로 생성되는 각 **node**는 필요한 모든 필드들을 다 가지는 **structured data type**을 가지도록 하여 **doubly linked list**를 형성하도록 함으로써, 아래 기능을 수행하는 **C** 프로그램을 작성하시오. 단, 각 **node**는 삽입시에 동적으로 할당되어야 하며 <학번, 이름, 주소>를 가지며 그 리스트에서 학번의 증가순으로 정렬된 위치에 삽입되어야 한다.

```
i(삽입), d(삭제), f(탐색), r(전체 읽기), f(작업종료)를 선택하시오: i
삽입할 자료(학번, 이름, 주소)를 입력하시오: 30  일지매 대구 수성구 지산동
삽입할 자료(학번, 이름, 주소)를 입력하시오: 60  홍길동 부산 동래구 수정동
삽입할 자료(학번, 이름, 주소)를 입력하시오: 40  춘향 전남 남원
삽입할 자료(학번, 이름, 주소)를 입력하시오: 50  박지성 경기도 수원
삽입할 자료(학번, 이름, 주소)를 입력하시오: ^z
i(삽입), d(삭제), f(탐색), r(전체 읽기), f(작업종료)를 선택하시오: f
탐색할 자료의 학번을 입력하시오: 40
<학번, 이름, 주소> = <40, 춘향, 전남 남원>
i(삽입), d(삭제), f(탐색), r(전체 읽기), f(작업종료)를 선택하시오: r
<학번, 이름, 주소> : <30, 일지매, 대구 수성구 지산동>, <40, 춘향, 전남 남원>, <50, 박지성, 경기도 수원>, <60, 홍길동, 부산 동래구 수정동>
 i(삽입), d(삭제), f(탐색), r(전체 읽기), f(작업종료)를 선택하시오: d
삭제할 자료의 학번을 입력하시오: 20
<학번, 이름, 주소> = <20, null, null>은 없는 자료입니다.
i(삽입), d(삭제), f(탐색), r(전체 읽기), f(작업종료)를 선택하시오: d
삭제할 자료의 학번을 입력하시오:40
<학번, 이름, 주소> = <40, 춘향, 전남 남원>이 삭제되었습니다.
i(삽입), d(삭제), f(탐색), r(전체 읽기), f(작업종료)를 선택하시오: r
<학번, 이름, 주소> : <30, 일지매, 대구 수성구 지산동>, <50, 박지성, 경기도 수원>, <60, 홍길동, 부산 동래구 수정동>
i(삽입), d(삭제), f(탐색), r(전체 읽기), f(작업종료)를 선택하시오: f
```

0	30	일지매	대구 수성구 지산동	2	-1
1	60	홍길동	부산 동래구 수정동	-1	3
2	40	춘향	전남 남원	3	0
3	50	박지성	경기도 수원	1	2
4				5	-1
5				6	4
6				7	5
7				-1	6

first = 0

avail = 4



```
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  r
The list contains:
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  f
탐색할 자료의 학번을 입력하시오: 40
탐색할 학번 40가 존재하지 않음.
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  d
삭제할 자료의 학번을 입력하시오: 40
삭제할 학번 40가 존재하지 않음.
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  i
id name address: 30 일지매 대구 수성구 지산동
id name address: 60 홍길동 부산 동래구 수정동
id name address: 40 춘향 전남 남원
id name address: 50 박지성 경기도 수원
id name address: 10 메시 스페인 FC 바르셀로나
id name address: ^Z
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  r
The list contains:
  10 메시   스페인 FC 바르셀로나
  30 일지매 대구 수성구 지산동
  40 춘향   전남 남원
  50 박지성 경기도 수원
  60 홍길동 부산 동래구 수정동
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  f
탐색할 자료의 학번을 입력하시오: 40
  40 춘향   전남 남원
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  f
탐색할 자료의 학번을 입력하시오: 20
탐색할 학번 20가 존재하지 않음.
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  d
삭제할 자료의 학번을 입력하시오: 40
삭제:   40 춘향   전남 남원
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  d
삭제할 자료의 학번을 입력하시오: 40
삭제할 학번 40가 존재하지 않음.
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  r
The list contains:
  10 메시   스페인 FC 바르셀로나
  30 일지매 대구 수성구 지산동
  50 박지성 경기도 수원
  60 홍길동 부산 동래구 수정동
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  i
id name address: 20 차두리 서울 FC 서울
id name address: 80 이순신 조선 선조 임진왜란
id name address: ^Z
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  r
The list contains:
  10 메시   스페인 FC 바르셀로나
  20 차두리 서울 FC 서울
  30 일지매 대구 수성구 지산동
  50 박지성 경기도 수원
  60 홍길동 부산 동래구 수정동
  80 이순신 조선 선조 임진왜란
i<삽입>, d<삭제>, f<탐색>, r<전체 읽기>, q<작업종료>를 선택하시오:  q
계속하려면 아무 키나 누르십시오 . . .
```

```
30 일지매 대구 수성구 지산동
60 홍길동 부산 동래구 수정동
40 춘향 전남 남원
50 박지성 경기도 수원
10 메시 스페인 FC 바르셀로나

20 차두리 서울 FC 서울
80 이순신 조선 선조 임진왜란
```

4.3 LINKED STACKS AND QUEUES

Previously we represented stacks and queues **sequentially**. Such a representation proved efficient if we had only one stack or one queue. However, when several stacks and queues coexisted, there was no efficient way to represent them sequentially. Figure 4.10 shows ***a linked stack*** and ***a linked queue***. Notice that the direction of links for both the stack and the queue facilitate easy insertion and deletion of nodes. In the case of Figure 4.11(a), we can easily add or delete a node from the top of the stack. In the case of Figure 4.11(b), we can easily add a node to the rear of the queue and add or delete a node at the front, although we normally will not add items to the front of a queue.

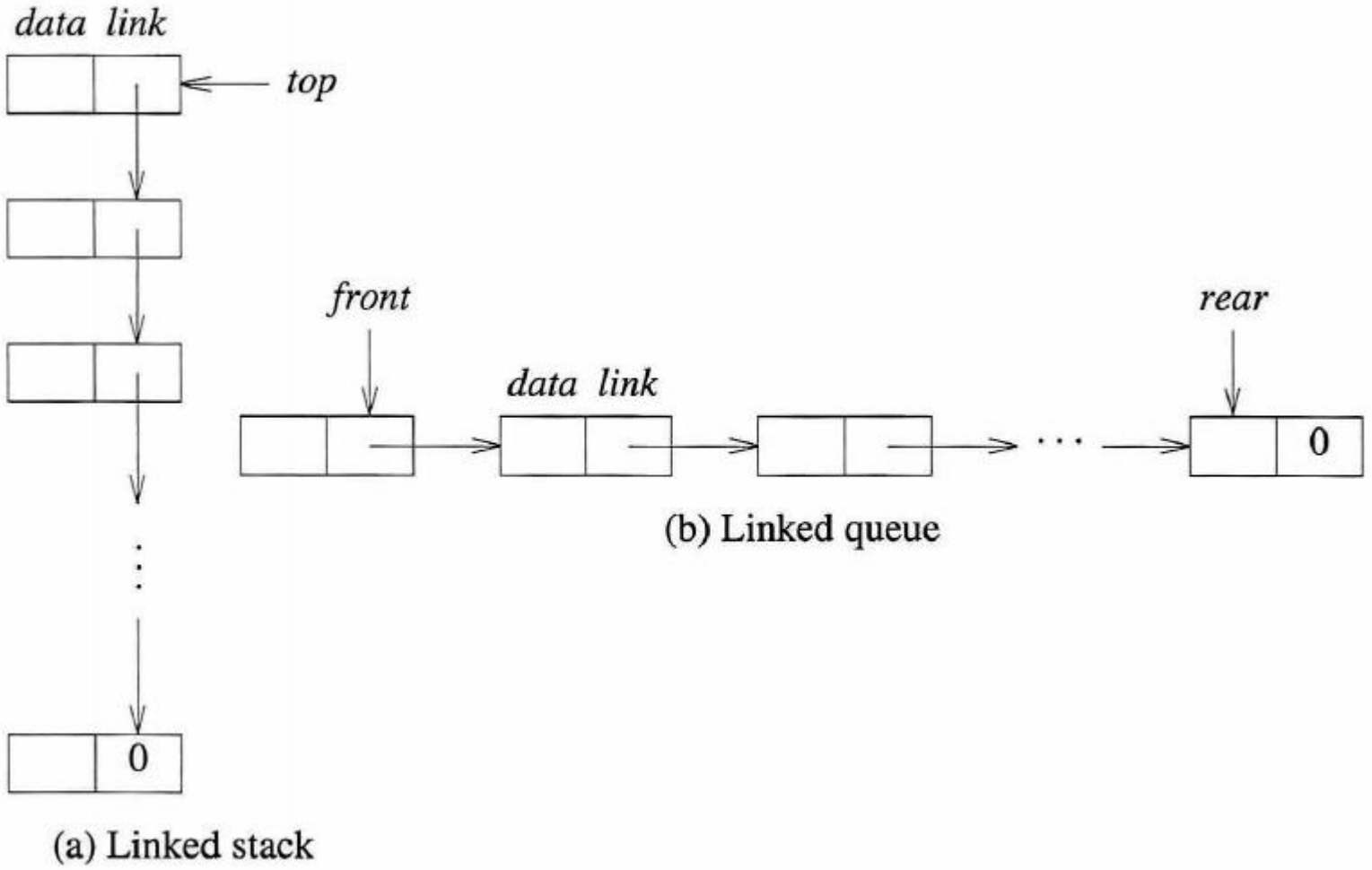
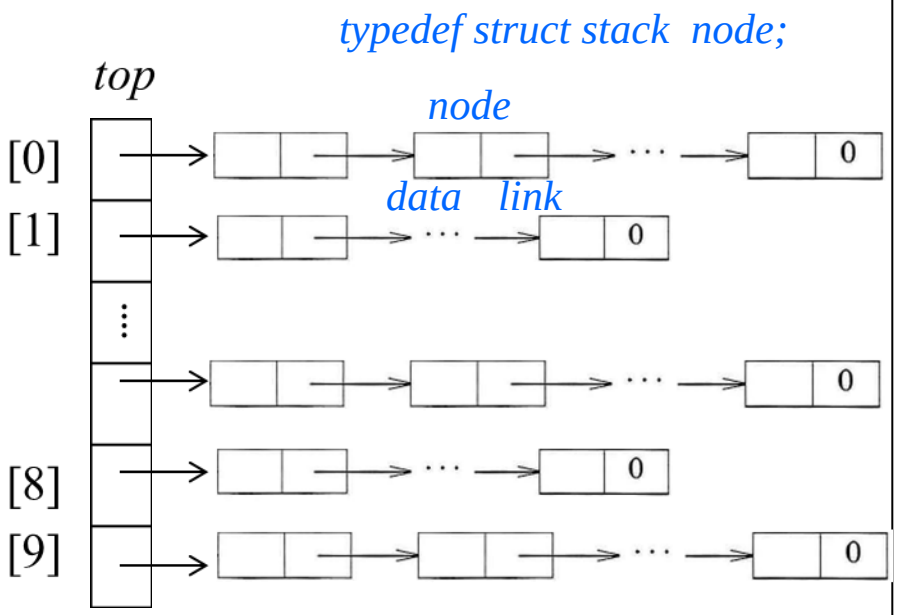


Figure 4.11: Linked stack and queue

If we wish to represent ***n ≤ MAX_STACKS* stacks** simultaneously, we begin with the declarations:

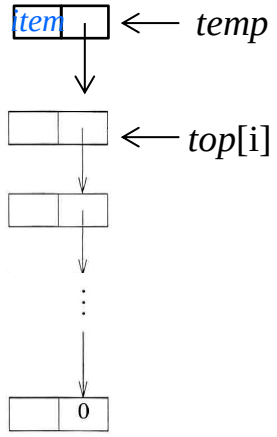
```
#define MAX_STACKS 10 /* maximum number of stacks */
typedef struct {
        int key;
        /* other fields */
} element;
typedef struct stack *stackPointer;
typedef struct stack {
        element data;
        stackPointer link;
} ;
stackPointer top[MAX_STACKS];
```




```
/* We assume that the initial condition for the stacks is: */
top[i] = NULL, 0 ≤ i <MAX_STACKS
/* and the boundary condition is: */
top[i] == NULL iff the ith stack is empty
```

```
void push(int i, element item)
{
        /* add item to the ith stack */
        stackPointer temp;
        MALLOC(temp, sizeof(*temp) );
        temp->data = item;
        temp->link = top[i];
        top[i] = temp;
}
```

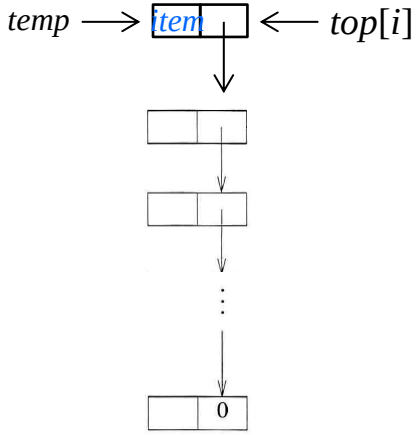
Program 4.5: Add to a linked stack



push(i, item)

```
element pop(int i)
{
        /* remove top element from the ith stack */
        stackPointer temp = top[i];
        element item;
        if  (!temp)
                return stackEmpty();
        item = temp->data;
        top[i] = temp->link;
        free(temp);
        return item;
}
```

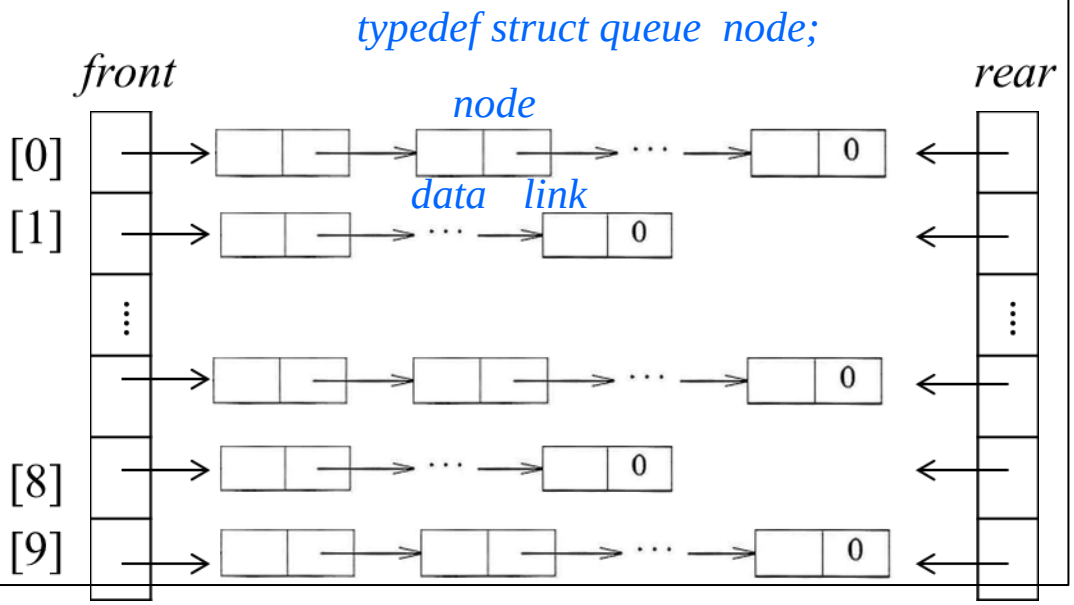
Program 4.6: Delete from a linked stack



item = pop(i)

To represent ***n ≤ MAX_QUEUE* queues** simultaneously, we begin with the declarations:

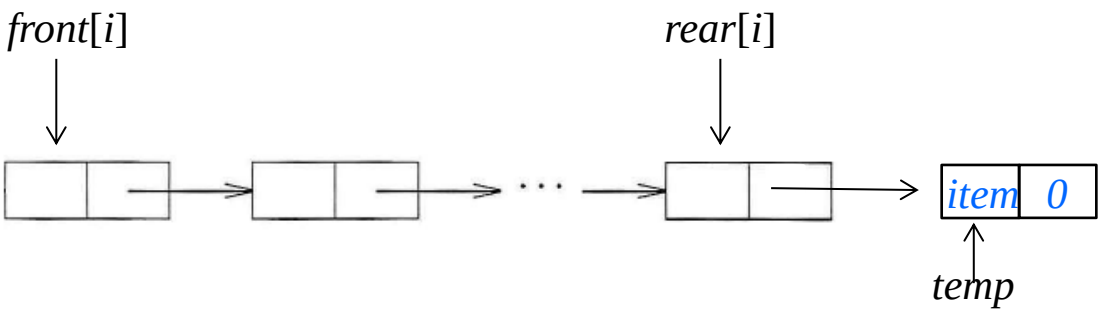
```
#define MAX_QUEUES 10 /* maximum number of queues */
typedef struct queue *queuePointer;
typedef struct queue {
        element data;
        queuePointer link;
} ;
queuePointer front[MAX_QUEUES], rear[MAX_QUEUES];

/* We assume that the initial condition for the queues is: */
front[i] = NULL, 0  ≤ i <MAX_QUEUES
/* and the boundary condition is: */
front[i] == NULL iff the ith queue is empty
```



```
void addq(int i, element item)
{
        /* add item to the rear of queue i */
        queuePointer temp;
        MALLOC(temp, sizeof(*temp));
        temp->data = item;
        temp->link = NULL;
        if (front[i])
                rear[i]->link = temp;
        else
                front[i] = temp;
        rear[i] = temp;
}
```

Program 4.7: Add to the rear of a linked queue

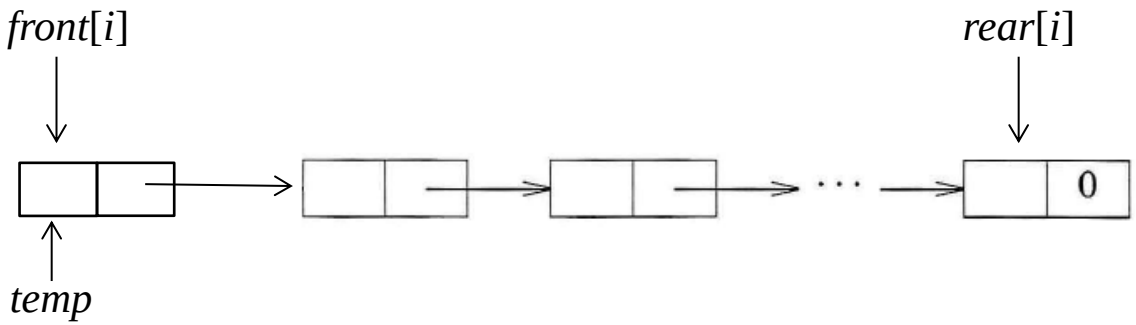
addq(i, item)



```
element deleteq(int i)
{
        /* delete an element from queue i */
        queuePointer temp = front[i];
        element item;
        if (!temp)
                return queueEmpty();
        item = temp->data;
        if (front[i] == rear[i]) {
                front[i] = rear[i] = NULL;
        } else
                front[i]= temp->link;
        free(temp) ;
        return item;
}
```

Program 4.8: Delete from the front of a linked queue

item = deleteq(i)



4.4 Polynomials

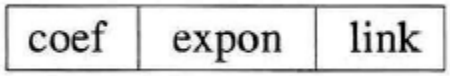
4.4.1 Polynomial Representation

- Polynomial

$$A(x) = a_{m-1}x^{e_{m-1}} + \ldots + a_0 x^{e_0}, \text{ where } a_{m-1} > a_{m-2} > \ldots > a_1 \geq 0$$

- Representation of Polynomial

```
typedef struct polyNode *polyPointer;
typedef struct polyNode {
        int coef;
        int expon;
        polyPointer link;
} ;
polyPointer a,b;
```



- Representation of polynomials

$$a = 3x^{14} + 2x^8 + 1$$

$$b = 8x^{14} - 3x^{10} + 10x^6$$

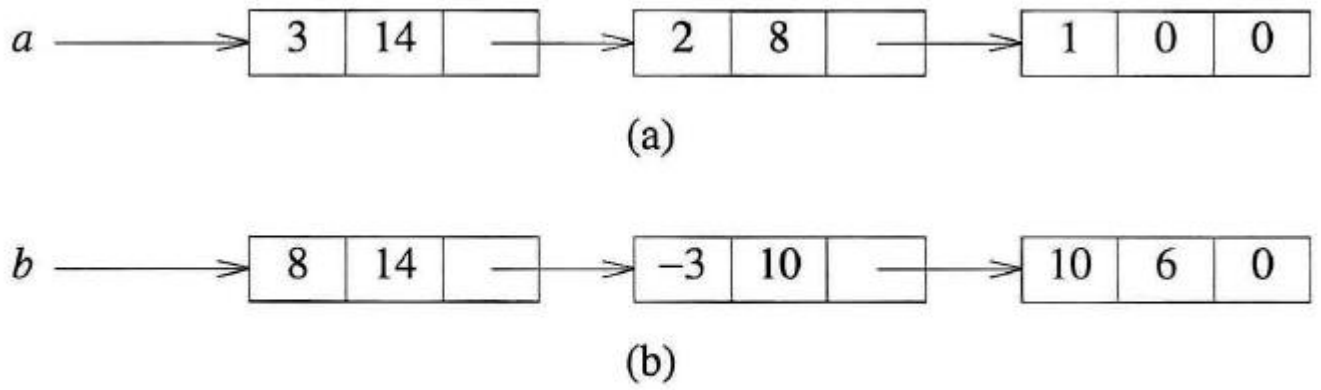
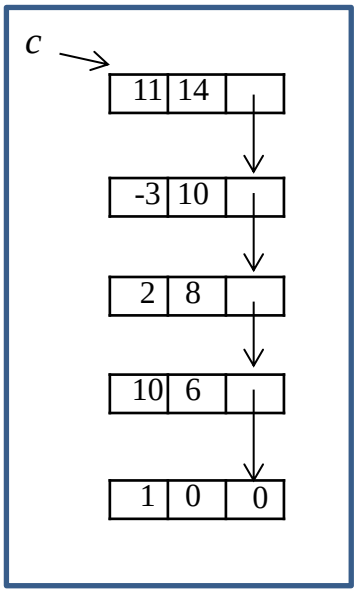


Figure 4.12: Representation of $3x^{14}+2x^8+1$ and $8x^{14}-3x^{10}+10x^6$

4.4.2 Adding Polynomials

$$a = 3x^{14} + 2x^{8} + 1$$

$$b = 8x^{14} - 3x^{10} + 10x^{6}$$



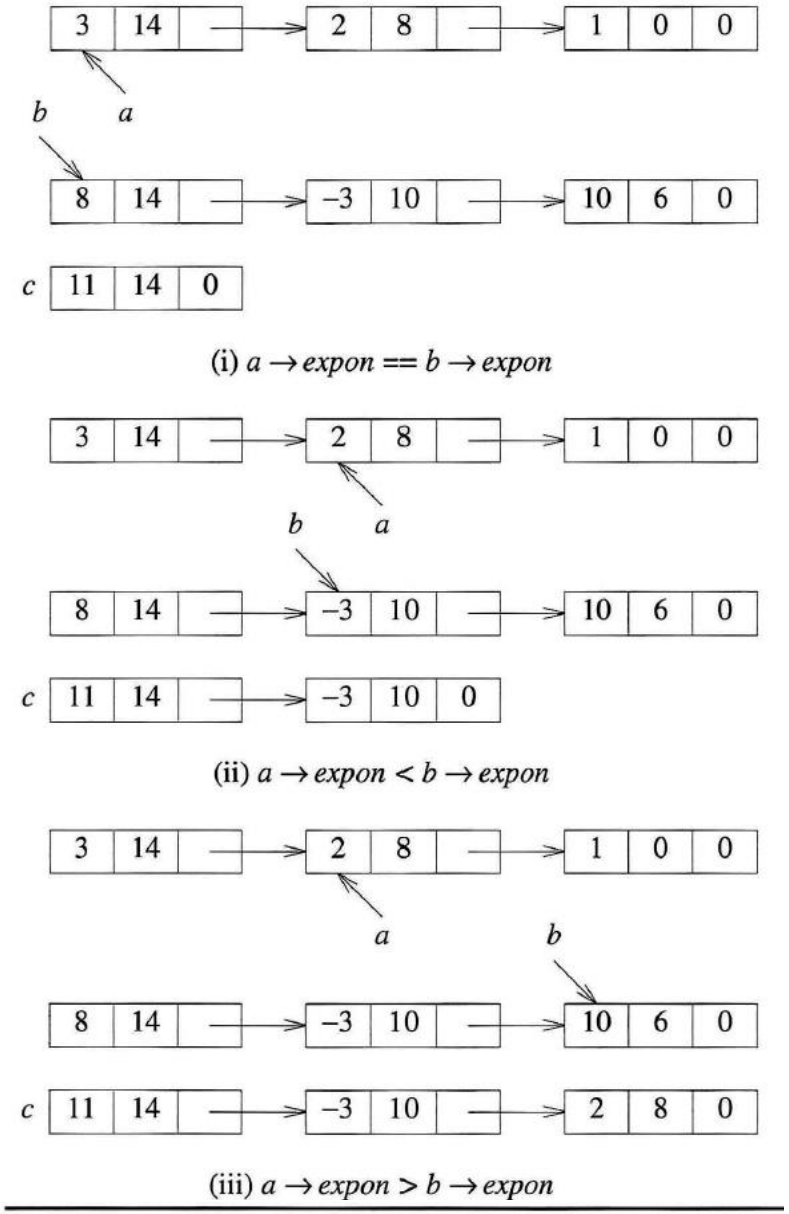
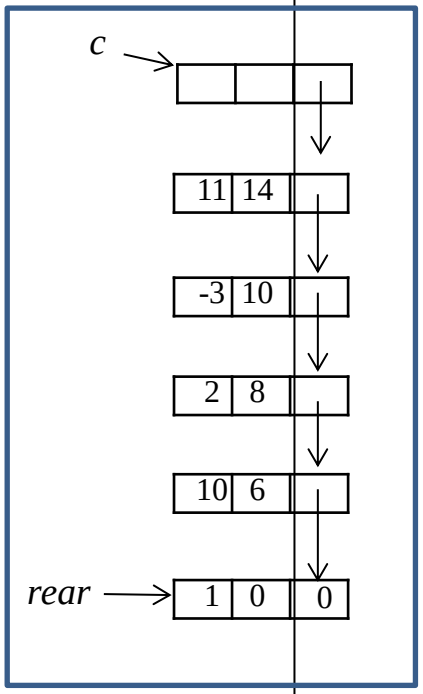


Figure 4.13: Generating the first three terms of $c = a + b$

```
polyPointer padd(polyPointer a, polyPointer b)
{
        /* return a polynomial which is the sum of a and b */
        polyPointer c, rear, temp;
        int sum;
        MALLOC(rear, sizeof(*rear));
        c = rear;
        while (a && b) {
                switch (COMPARE(a->expon, b->expon)) {
                        case -1: /* a->expon < b->expon */
                                attach(b->coef,b->expon,&rear);
                                b = b->link; break;
                        case 0: /* a->expon = b->expon */
                                sum = a->coef + b->coef;
                                if (sum) attach(sum,a->expon,&rear);
                                a = a->link; b = b->link; break;
                        case 1: /* a->expon > b->expon */
                                attach(a->coef,a->expon,&rear);
                                a = a->link;
                }
        }
        / * copy rest of list a and then list b */
        for (; a; a= a->link) attach(a->coef,a->expon,&rear);
        for (; b; b = b->link) attach(b->coef,b->expon,&rear);
        rear->link = NULL;
        / * delete extra initial node */
        temp= c; c = c->link; free(temp);
        return c;
}
```

Program 4.9: Add two polynomials




```
void attach(float coef, int expon, polyPointer
*rear)
{
    polyPointer temp;
    MALLOC(temp, sizeof(*temp));
    temp->coef = coef;
    temp->expon = expon;
    temp->link = NULL;
    (*rear)->link = temp;
    *rear = temp;
}
```

Program 4.10: Attach a node to the end of a list

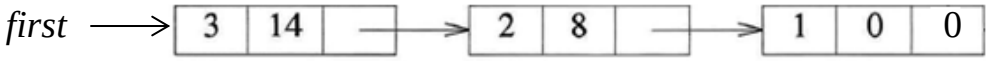
4.4.3 Erasing Polynomials

```
void erase(polyPointer *ptr)
{
        /* erase the polynomial pointed to by ptr */
        polyPointer temp;
        while (*ptr) {
                temp = *ptr;
                *ptr = (*ptr)->link;
                free (temp);
        }
}
```

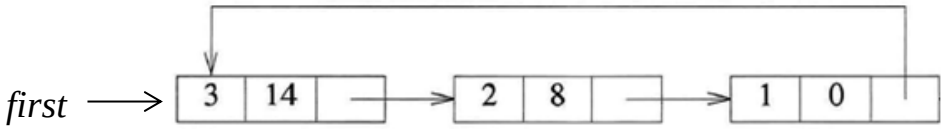
Program 4.11: Erasing a polynomial

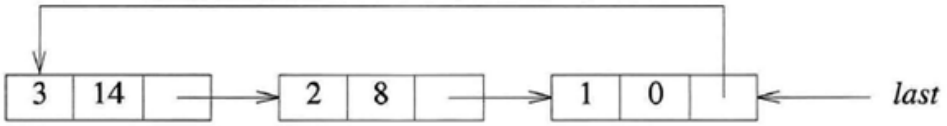
4.4.4 Circular List Representation of Polynomials

- ***Chain***
  - A singly linked list in which the last node has **a null link**



- ***Circular list***
  - The link filed of the last node points to the first node in the list



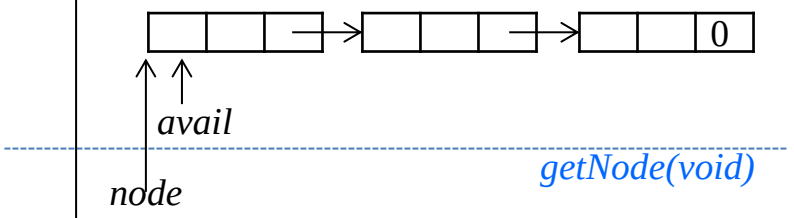


It is often useful to have a last node pointer rather than a first node pointer. By doing this, additions to the front and end become O(1).

- Let's maintain our own list (as a *chain*) of nodes that have been "freed".
  - When we need a new node, we examine this list. If the list is not empty, then we may use one of its nodes.
  - Only when the list is empty do we need to use *malloc* to create a new node.
  - Let ***avail*** be a variable of type *polyPointer* that points to the first node in our list of freed nodes. Henceforth, we call this list **the available space list** or ***avail*** list.
    - Initially, we set avail to *NULL*. Instead of using *malloc* and *free*, we now use *getNode* (Program 4.12) and *retNode* (Program 4.13).

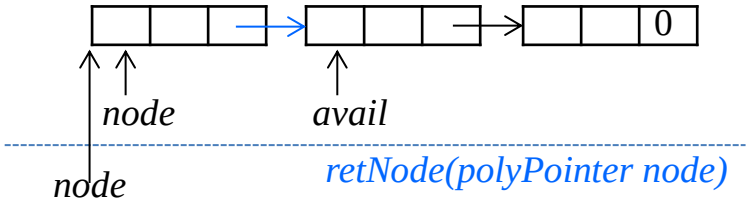
```
polyPointer getNode(void)
{
        /* provide a node for use */
        polyPointer node;
        if (avail) {
                node = avail;
                avail = avail->link;
        } else
                MALLOC(node, sizeof(*node));
        return node;
}
```

Program 4.12: getNode function



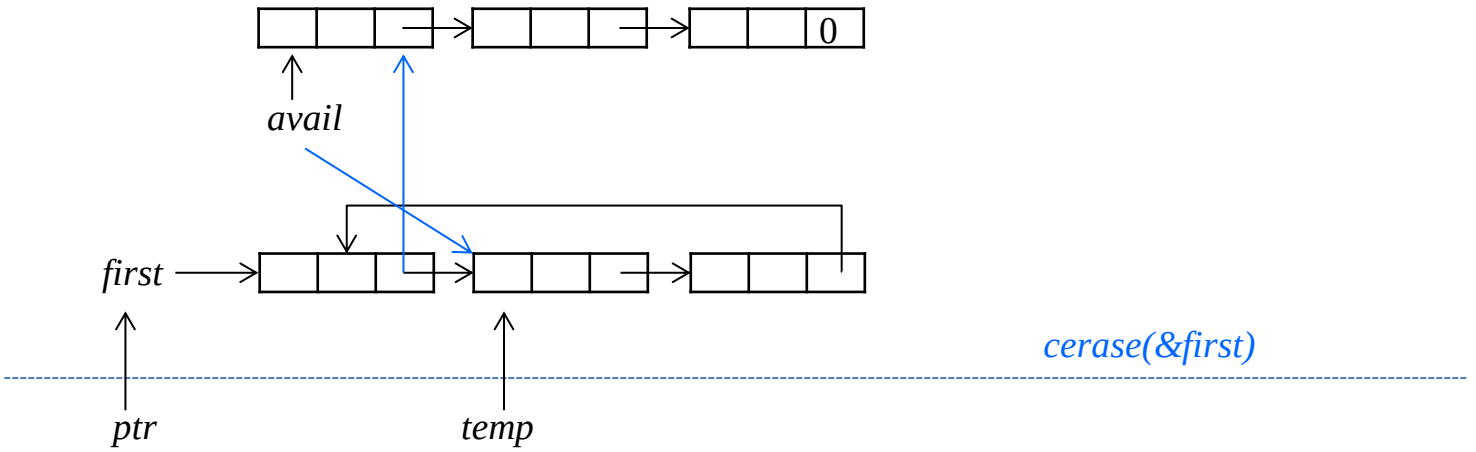

```
void retNode(polyPointer node)
{
        /* return a node to the available list */
        node->link = avail;
        avail = node;
}
```

Program 4.13: retNode function



```
void cerase(polyPointer *ptr)
{
        /* erase the circular list pointed to by ptr */
        polyPointer temp;
        if (*ptr) {
                temp = (*ptr)->link;
                (*ptr)->link = avail;
                avail = temp;
                *ptr = NULL;
        }
}
```

Program 4.14: Erasing a circular list



- To avoid handling **the zero polynomial** as a special case, *a* ***header node*** is added.

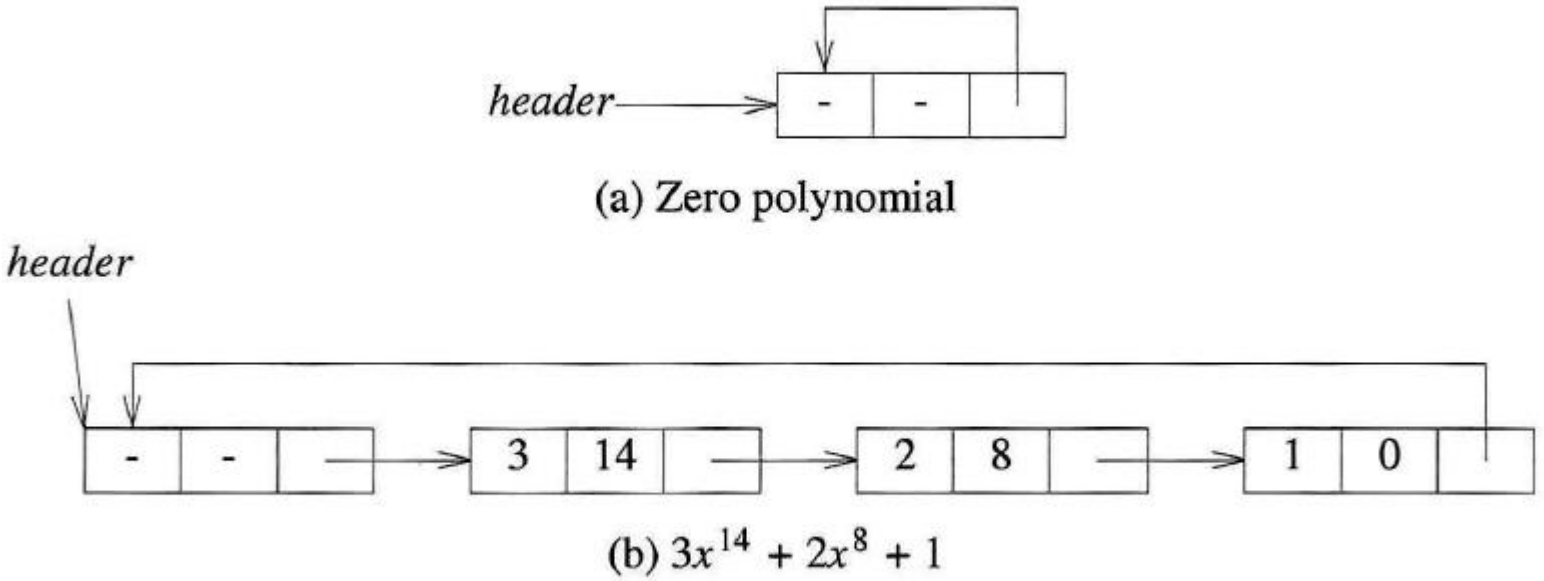
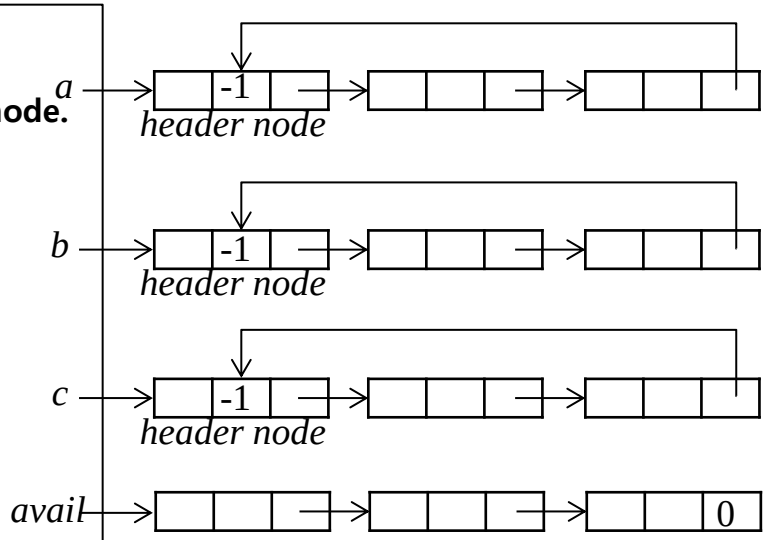


Figure 4.15: Example polynomials with header nodes

```
polyPointer cpadd(polyPointer a, polyPointer b)
{
    /* polynomials a and b are singly linked circular lists with a header node.
       Return a polynomial which is the sum of a and b */
    polyPointer startA, c, lastC;
    int sum, done = FALSE;
    startA = a;              /* record start of a */
    a = a->link; /* skip header node for a and b*/
    b = b->link;
    c = getNode() ; /* get a header node for sum */
    c->expon = -1; lastC = c;
    do {
        switch (COMPARE(a->expon, b->expon))
            case -1: /* a->expon < b->expon */
                     attach(b->coef,b->expon,&lastC);
                     b = b->link; break;
            case 0: /* a->expon == b->expon */
                     if (startA == a) done = TRUE;
                     else {
                         sum = a->coef + b->coef;
                         if (sum) attach(sum,a->expon,&lastC);
                         a = a->link; b = b->link;
                     }
                     break;
            case l: /* a->expon > b->expon */
                     attach(a->coef,a->expon,&lastC);
                     a = a->link;
        }
    } while (!done);
    lastC->link = c;
    return c;
}
```

Program 4.15: Adding two polynomials represented as circular lists with header nodes



Try to understand !!!
Now, It's your turn!

4.5 Additional List Operations

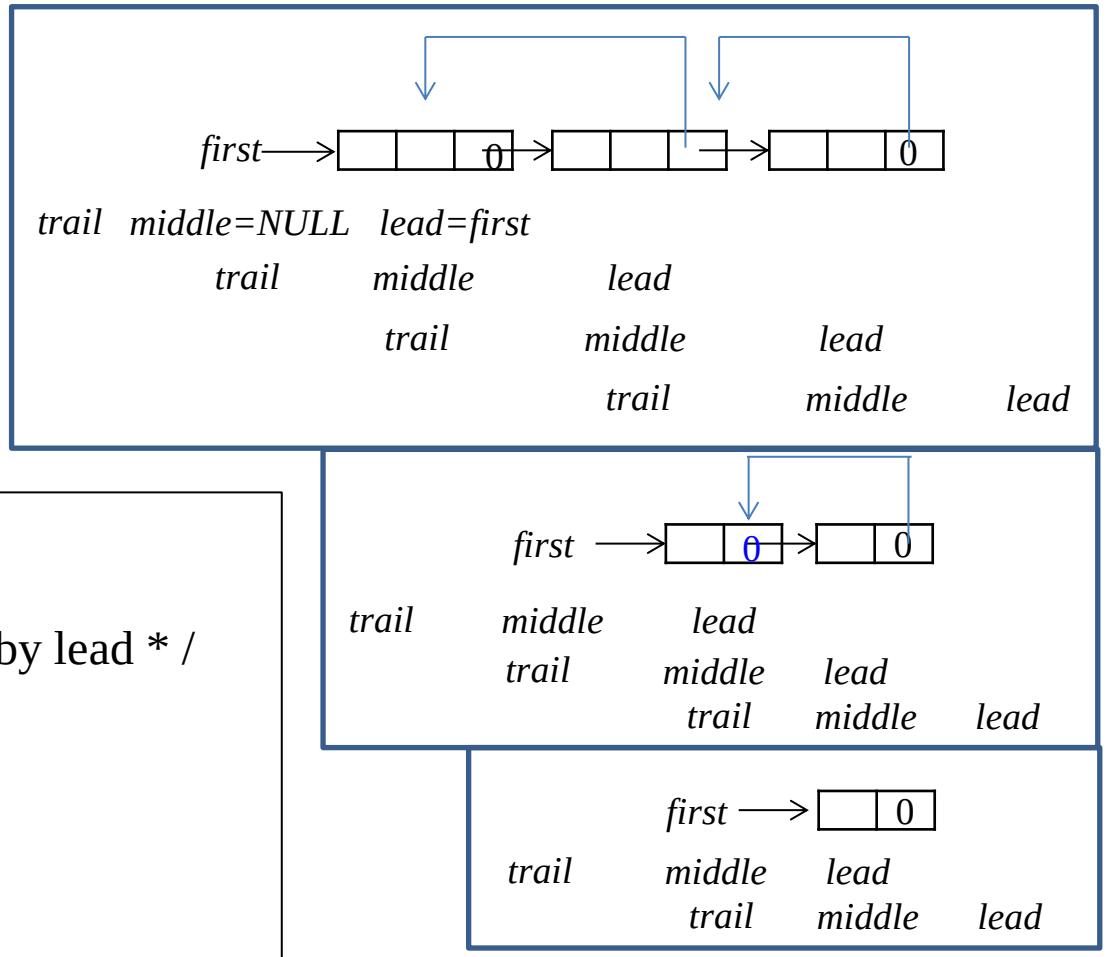
4.5.1 Operations For Chains

It is often necessary, and desirable, to build a variety of functions for manipulating singly linked lists. Some that we have seen already are *getNode* and *retNode*, which get and return nodes to the available space list. **Inverting (or reversing) a chain** (Program 4.16) is another useful operation. This routine is especially interesting because we can do it "in place" if we use three pointers. We use the following declarations:

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

Try out this function with at least three examples, an empty list and lists of one and two nodes, so that you understand how it works. For a list of *length* ≥ 1 nodes, the **while** loop is executed *length* times and so the computing time is linear or O(*length*).

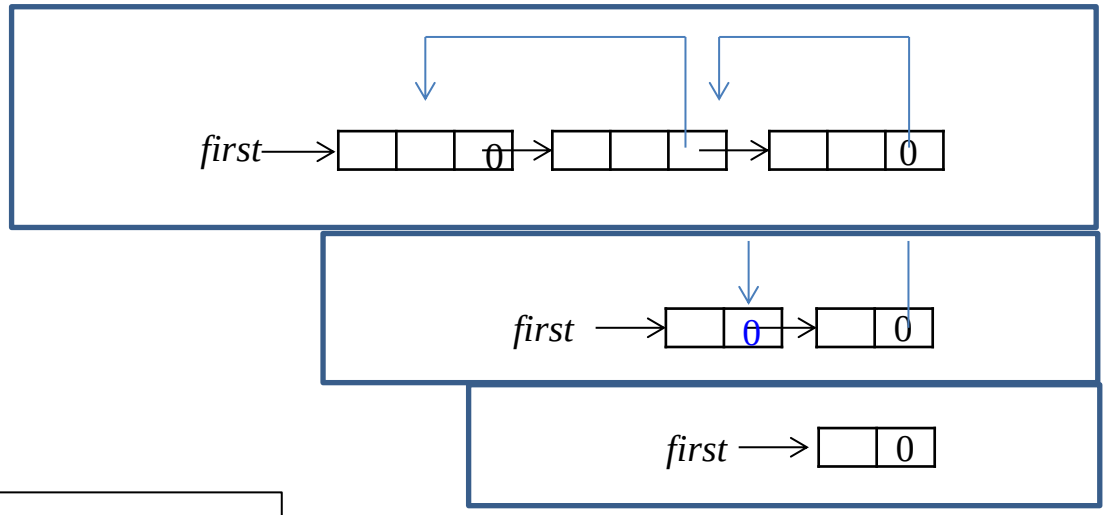
first = invert(first)




```
listPointer invert(listPointer lead)
{
        / * invert the list pointed to by lead * /
        listPointer middle, trail;
        middle = NULL;
        while (lead) {
                trail = middle;
                middle = lead;
                lead = lead->link;
                middle->link = trail;
        }
        return middle;
}
```

Program 4.16: Inverting a singly linked list

first = invert(first)




```
listPointer invert(listPointer lead)
{
    / * invert the list pointed to by lead * /
    listPointer prev, current, next;
    prev = NULL; current = NULL; next = lead;
    while (next) {
            prev = current;
            current = next;
            next = next->link;
            current->link = prev;
    }
    return current;
}
```

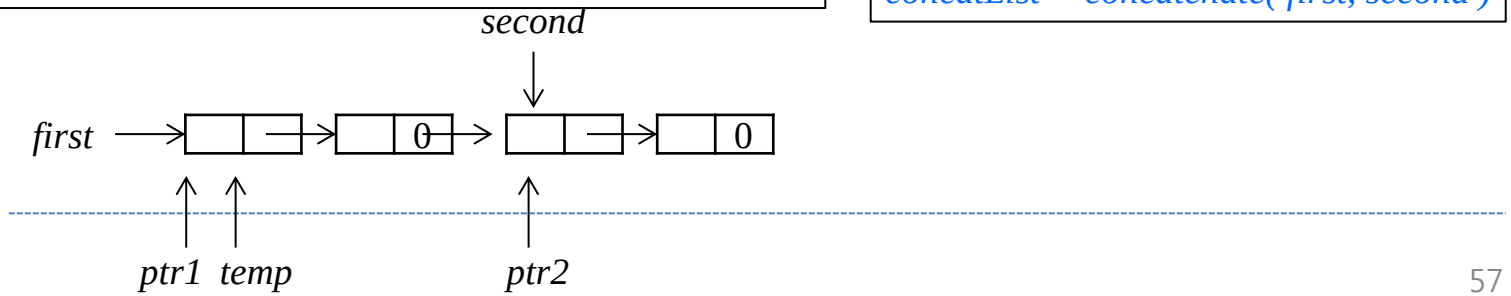
Program 4.16: Inverting a singly linked list

```
listPointer concatenate(listPointer ptrl, listPointer ptr2)
{
     /* produce a new list that contains the list ptrl
          followed by the list ptr2. The list pointed to
          by ptrl is changed permanently */
     listPointer temp;
     /* check for empty lists */
     if (!ptrl) return ptr2;
     if (!ptr2) return ptrl;

     /* neither list is empty, find end of the first list */
     for (temp = ptrl; temp->link; temp = temp->link);

     /* link the end of the first to start of the second */
     temp->link = ptr2;
     return ptr1;
}
```

Program 4.17: Concatenating singly linked lists

Another useful function is one that **concatenates two chains**, *ptr1* and *ptr2* (Program 4.17). The complexity of this function is O(length of list *ptr1*). Since this function does not allocate additional storage for the new list, *ptr1* also contains the concatenated list. (The exercises explore a concatenation function that does not alter *ptr1*.)

concatList = concatenate(first, second)



4.5.2 Operations For Circularly Linked Lists

Now let us take another look at circular lists like the one in Figure 4.14. By keeping a pointer ***last*** to the last node in the list rather than to the first, we are able to insert an element at both the front and end with ease. Had we kept a pointer to the first node instead of the last node, inserting at the front would require us to move down the entire length of the list until we find the last node so that we can change the pointer in the last node to point to the new first node. Program 4.18 gives the code at insert a node at the front of a circular list. To insert at the rear, we only need to add the additional statement **last* = *node* to the else clause of *insertFront* (Program 4.18).

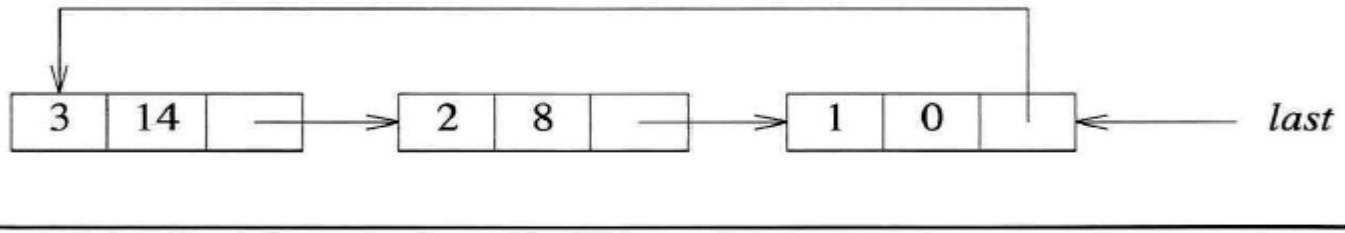
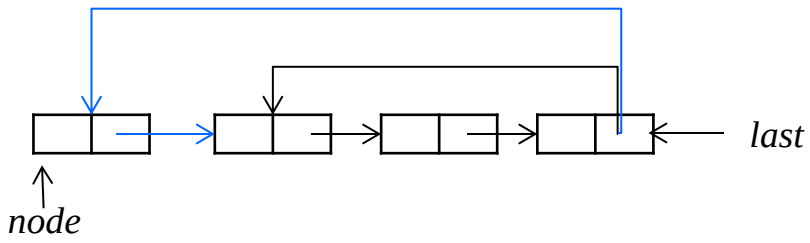


Figure 4.14: Circular representation of $3x^{14} + 2x^8 + 1$

```
void insertFront(listPointer *last, listPointer node)
{
        /* insert node at the front of the circular list whose last node is last */
        if (!(*last)) {
                /* list is empty, change last to point to new entry */
                *last = node;
                node->link = node;
        } else {
                /* list is not empty, add the new entry at front */
                node->link = (*last)->link;
                (*last)->link = node;
        }
}
```

Program 4.18: Inserting at the front of a list

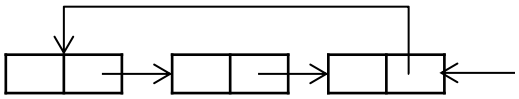
insertFront(&last, node)



As another example of a simple function for circular lists, we write a function (Program 4.19) that determines the length of such a list.

```
int length(listPointer last)
{
        /* find the length of the circular list last */
        listPointer temp;
        int count = 0;
        if (last) {
                temp = last;
                do {
                        count++;
                        temp = temp->link;
                } while (temp !=last);
        }
        return count;
}
```

Program 4.19: Finding the length of a circular list



length(last)

4.6 EQUIVALENCE CLASSES

Equivalence class <From Wikipedia, the free encyclopedia>

In mathematics, when the elements of some set *S* have a notion of **equivalence** (formalized as an **equivalence relation**) defined on them, then one may naturally split the set *S* into **equivalence classes**. These equivalence classes are constructed so that elements *a* and *b* belong to the same equivalence class if and only if *a* and *b* are equivalent.

Formally, given a set *S* and an equivalence relation ~ on *S*, the equivalence class of an element *a* in *S* is the set

$$\{ x \in S \mid x \sim a \}$$

of elements which are equivalent to *a*. It may be proven from the defining properties of "equivalence relations" that the equivalence classes form a partition of S. This partition – the set of equivalence classes – is sometimes called the **quotient set** or the **quotient space** of *S* by ~ and is denoted by *S* / ~.

When the set *S* has some structure (such as a **group operation** or a **topology**) and the equivalence relation ~ is defined in a manner suitably compatible with this structure, then the quotient set often inherits a similar structure from its parent set.

Let us put together some of the concepts on linked and sequential representations to solve a problem that arises in the design and manufacture of very large-scale integrated (VLSI) circuits. One of the steps in the manufacture of a VLSI circuit involves exposing a silicon wafer using a series of masks. Each mask consists of several polygons. **Polygons that overlap electrically are equivalent** and **electrical equivalence** specifies a relationship among mask polygons. This relation has several properties that it shares with other equivalence relations, such as the standard mathematical equals. Suppose that we denote an arbitrary **equivalence relation** by the symbol ≡ and that:

(1) For any polygon x, x ≡ x, that is, x is electrically equivalent to itself. Thus, ≡ is ***reflexive***.

(2) For any two polygons, x and y, if x ≡ y then y ≡ x. Thus, the relation ≡ is ***symmetric***.

(3) For any three polygons, x, y, and z, if x ≡ y and y ≡ z then x ≡ z. For example, if x and y are electrically equivalent and y and z are also equivalent, then x and z are also electrically equivalent. Thus, the relation ≡ is ***transitive***.

Definition: A relation, ≡, over a set, *S*, is said to be an ***equivalence relation*** over *S* iff it is **symmetric**, **reflexive**, and **transitive** over *S*.

Examples of equivalence relations are numerous. For example, **the "equal to" (=) relationship is an equivalence relation** since

(I) x = x

(2) x = y implies y = x

(3) x = y and y = z implies that x = z

We can use an equivalence relation to partition a set S into equivalence classes such that two members *x* and *y* of *S* are in the same **equivalence class** *iff* $x \equiv y$. For example, if we have twelve polygons numbered 0 through 11 and the following pairs overlap:

$0 \equiv 4, 3 \equiv 1, 6 \equiv 10, 8 \equiv 9, 7 \equiv 4, 6 \equiv 8, 3 \equiv 5, 2 \equiv 11, 11 \equiv 0$

then, as a result of the reflexivity, symmetry, and transitivity of the relation ≡, we can partition the twelve polygons into the following equivalence classes:

{ 0, 2, 4, 7, 11 }; {1, 3, 5}; { 6, 8, 9, 10}

These equivalence classes are important because they define a signal net that we can use to verify the correctness of the masks.

The algorithm to determine equivalence works in two phases. In the first phase, we read in and store the equivalence pairs <*i*, *j*>. In the second phase we begin at 0 and find all pairs of the form <0, *j*>, where 0 and *j* are in the same equivalence class. By transitivity, all pairs of the form <*j*, *k*> imply that k is in the same equivalence class as 0. We continue in this way until we have found, marked, and printed the entire equivalence class containing 0. Then we continue on.

Our first design attempt appears in Program 4.20. Let *m* and *n* represent the number of related pairs and the number of objects, respectively. We first must figure out which data structure we should use to hold these pairs. To determine this, we examine the operations that are required. The pair <*i*, *j*> is essentially two random integers in the range 0 to *n* -1. Easy random access would dictate an array, say *pairs*[*n*][*m*]. The *i*th row would contain the elements, *j*, that are paired directly to *i* in the input. However, this could waste a lot of space since very few of the array elements would be used. It also might require considerable time to insert a new pair, <*i*, *k*>, into row *i* since we would have to scan the row for the next free location or use more storage.

These considerations lead us to a linked list representation for each row. Our node structure requires only a data and a link field. However, since we still need random access to the *i*th row, we use a one-dimensional array, *seq*[*n*], to hold the header nodes of the *n* lists. For the second phase of the algorithm, we need a mechanism that tells us whether or not the object, *i*, has been printed. We use the array *out*[*n*] and the constants TRUE and FALSE for this purpose. Our next refinement appears in Program 4.21.

Let us simulate this algorithm, as we have developed it thus far, using the previous data set. After the while loop is completed the lists resemble those appearing in Figure 4.16. For each relation $i \equiv j$, we use two nodes. The variable *seq*[*i*] points to the list of nodes that contains every number that is directly equivalent to *i* by an input relation.

Input: 0 ≡ 4, 3 ≡ 1, 6 ≡ 10, 8 ≡ 9, 7 ≡ 4, 6 ≡ 8, 3 ≡ 5, 2 ≡ 11, 11 ≡ 0

Output: { 0, 2, 4, 7, 11 }; {1, 3, 5}; { 6, 8, 9, 10}

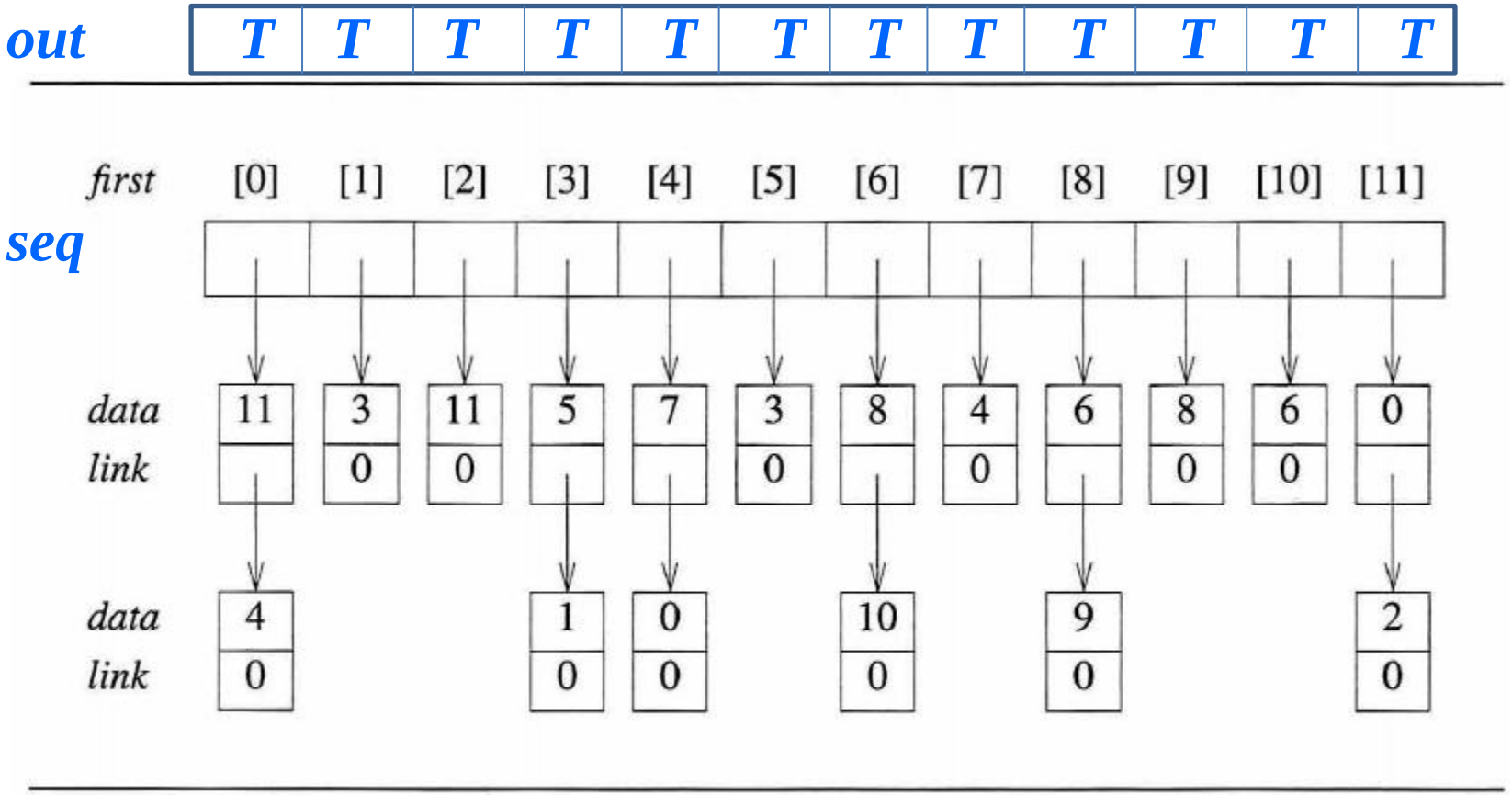


Figure 4.16: Lists after pairs have been input

출력 결과

New Class : 0 11 4 7 2

New Class : 1 3 5

New Class : 6 8 10 9

In phase two, we scan the *seq* array for the first i, $0 \leq i < n$, such that *out*[i] = TRUE. Each element in the list *seq*[i] is printed. To process the remaining lists which, by transitivity, belong in the same class as *i*, we create a stack of their nodes. We do this by changing the link fields so that they point in the reverse direction. Program 4.22 contains the complete equivalence algorithm.

```
#include <stdio.h>
#include <alloc.h>
#define MAX_SIZE 24
#define FALSE 0
#define TRUE 1
typedef struct node *nodePointer;
typedef struct node
        int data;
        nodePointer link;
} ;
void main(void)
{
        short int out[MAX_SIZE];
        nodePointer seq[MAX_SIZE];
        nodePointer current, next, top;
        int i,j,class_anchor, n;
        printf ("Enter the size (<= %d) ",MAX_SIZE);
        scanf ( "%d", &n);
        for (i = 0; i < n; i++) {
            /* initialize seq and out */
            out[i]=TRUE; seq[i] =NULL;
        }
        /* Phase 1: Input the equivalence pairs: */
        while (1) {
            printf ("Enter a pair of numbers (-1 -1 to quit): ");
            scanf("%d%d",&i,&j);
            if (i < 0) break;
            MALLOC(x, sizeof(*x));
            x->data = j; x->link = seq[i];  seq[i] = x;
            MALLOC(x, sizeof(*x));
            x->data = i; x->link = seq[j]; seq[j] = x;
        }
    /* Phase 2: output the equivalence classes */
    for (i = 0; i < n; i++) {
        if (out[i]) {
            printf("\nNew class: %5d", i);
            out[i] =FALSE;          /* set class to false */
            top= NULL;              /*initialize stack*/
            class_anchor = i;
             while (1) {                            /* find rest of the class */
                    /* class_anchor의 list를 처리한다 */
                    current = seq[class_anchor];
                    while (current)  {
                        j = current->data;
                        if (out[j]) {
                                printf("%5d", j);
                                out[j] =FALSE;
                                /* push node current onto the stack */
                                next = current->link;
                                current->link = top;
                                top = current;
                                current = next;
                        } else
                                current = current->link;
                    }
                    if (top==NULL) break;
                    class_anchor = top->data;
                    top = top->link;                /* pop*/
            }
        }
    }
}
```

Program 4.22: Program to find equivalence classes

과제 3(Finding Equivalence Classes)

Program 4.22는 equivalence classes를 찾기 위하여 stack을 사용하였다. 그런데 그 stack은 그림 4.16의 linked list를 흐트려 놓는다. 이제 stack을 사용하지만 그 linked list를 흐트리지 않는 stack을 사용하여 Program 4.22와 동일한 기능을 하는 프로그램을 개발하시오.

(0-1) stack의 선언.

```
AFTER:          #define MAX_STACK_SIZE   100
                int stack[MAX_STACK_SIZE];
```

(0-2) push(int value)와 int pop()의 구현

```
AFTER:          void push(int value) { ++top; stack[top] = value; }
                int pop() { return stack[top--]; }
```

(1) Top의 선언.

```
BEFORE:         nodePointer     top;
AFTER:          int             top;
```

(2) Top의 초기화

```
BEFORE:         top= NULL; /*initialize stack*/
AFTER:          top = -1;
```

(3) push()의 호출

```
BEFORE:         /* push node x onto the stack */
                y = x->link; x->link = top; top = x; x = y;
AFTER:          /* push node x onto the stack */
                        push(y->data); x = x->link;
```

(4) pop()의 호출.

```
BEFORE:         if (!top) break;
                x = seq[top->data];  top = top->link; /* unstack */
AFTER:          if (top == -1) break;
                x = seq[pop()]; /* unstack */
```