해달 자료구조 부트캠프

3. 연결리스트

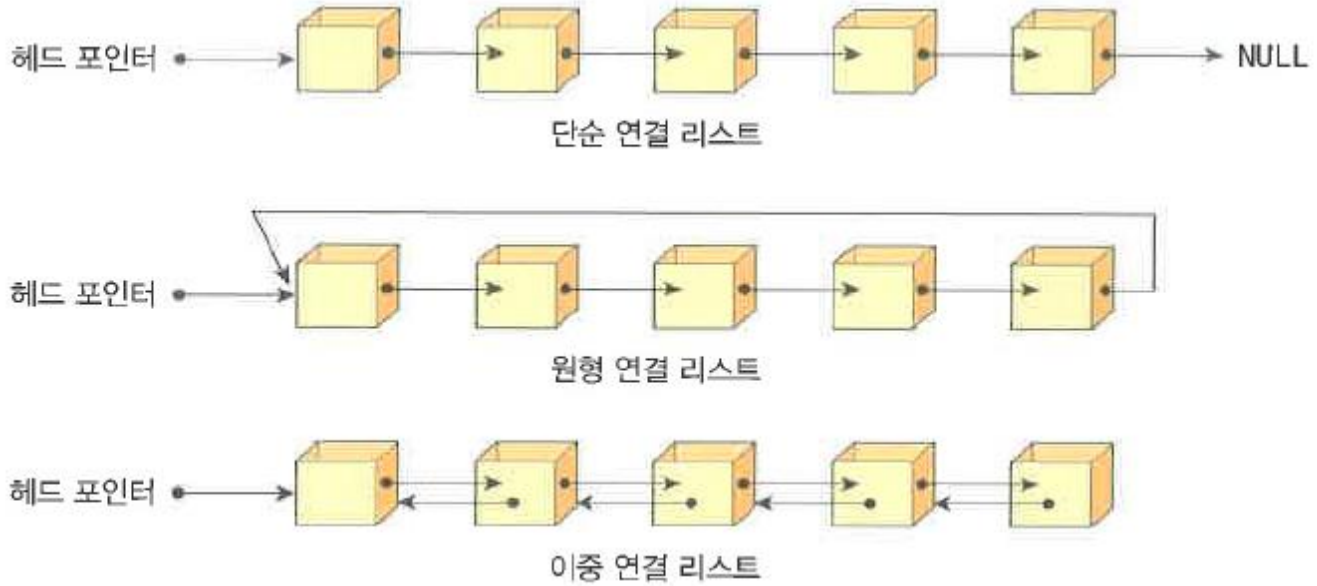
Table of Contents

Part 1

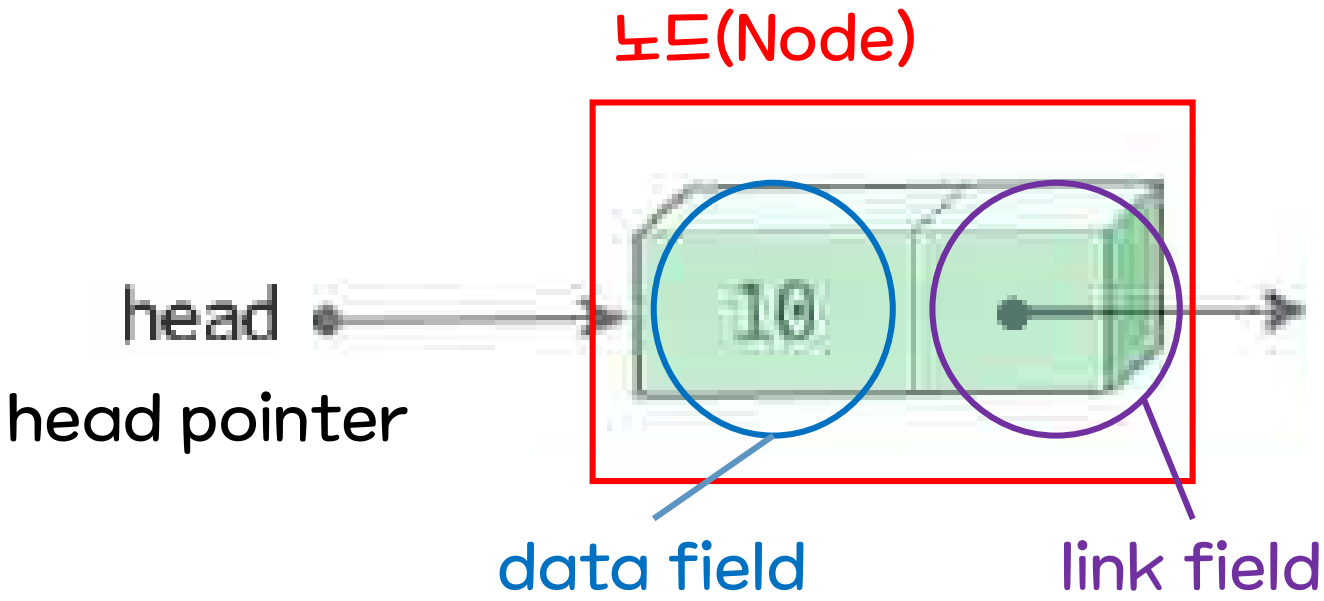
연결 리스트

소개

각 노드가 데이터와 포인터를 가지고 한 줄로 연결되어 있는 자료구조



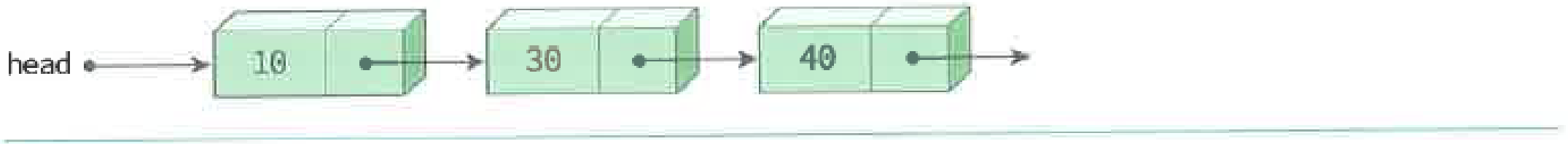
구조



연결 리스트에서는 연결 리스트의 첫 번째 노드를 알아야 전체 노드에 접근할 수 있음.
⇒ 연결 리스트마다 첫 번째 노드를 가리키는 head pointer가 필요

단순 연결 리스트

단순 연결 리스트(Singly linked list)는 하나의 방향으로만 연결되어 있는 연결 리스트.
마지막 노드의 링크는 NULL값을 가짐.



[그림 6-9] 단순 연결 리스트

단순 연결리스트에서 구현할 연산은 다음과 같음
- insert_first()
- insert()
- delete_first
- delete()
- print_list()

```c
typedef int element;

typedef struct ListNode {  // 노드 타입을 구조체로 정의한다.
    element data;
    struct ListNode *link;
} ListNode;
```

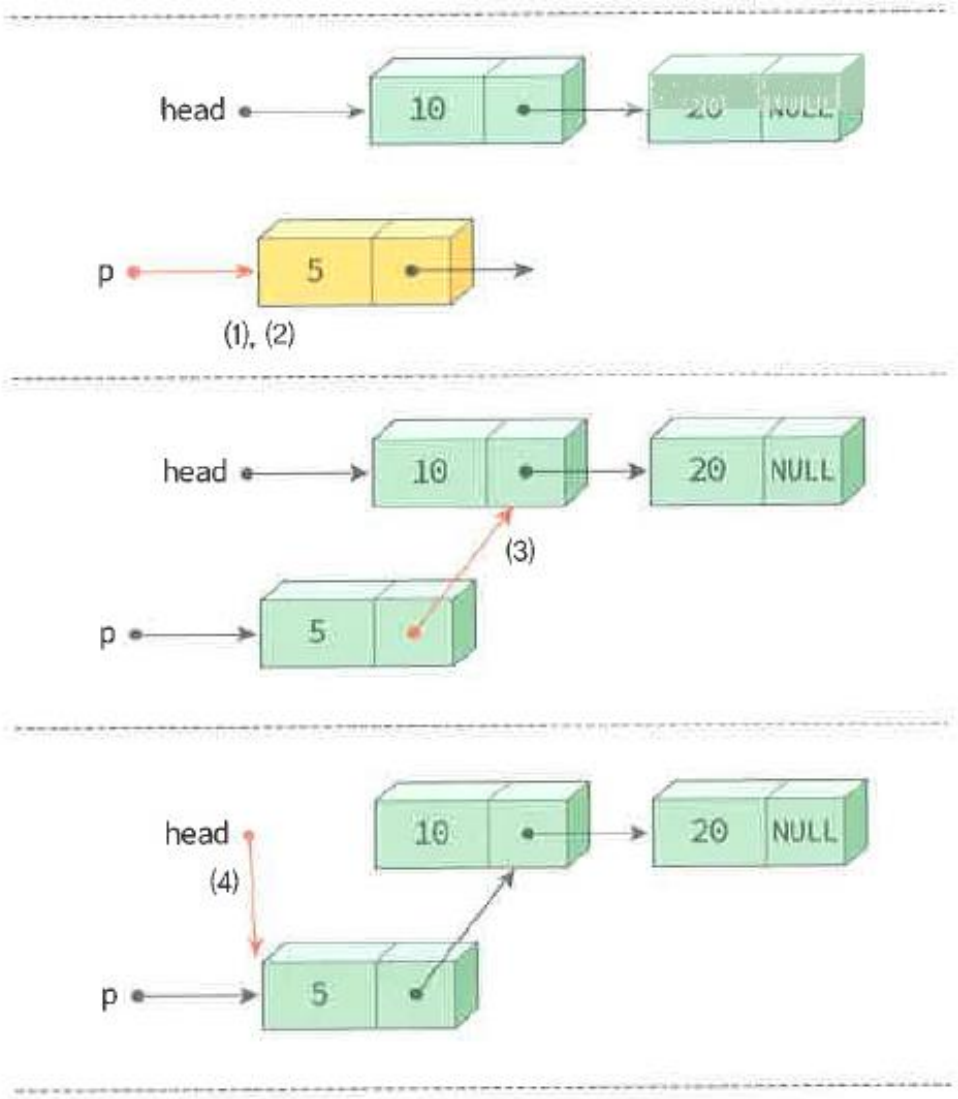
Part 1

단순 연결 리스트_노드 정의

```c
typedef int element;

typedef struct ListNode {  // 노드 타입을 구조체로 정의한다.
    element data;
    struct ListNode *link;
} ListNode;
```

Part 1

단순 연결 리스트_insert first

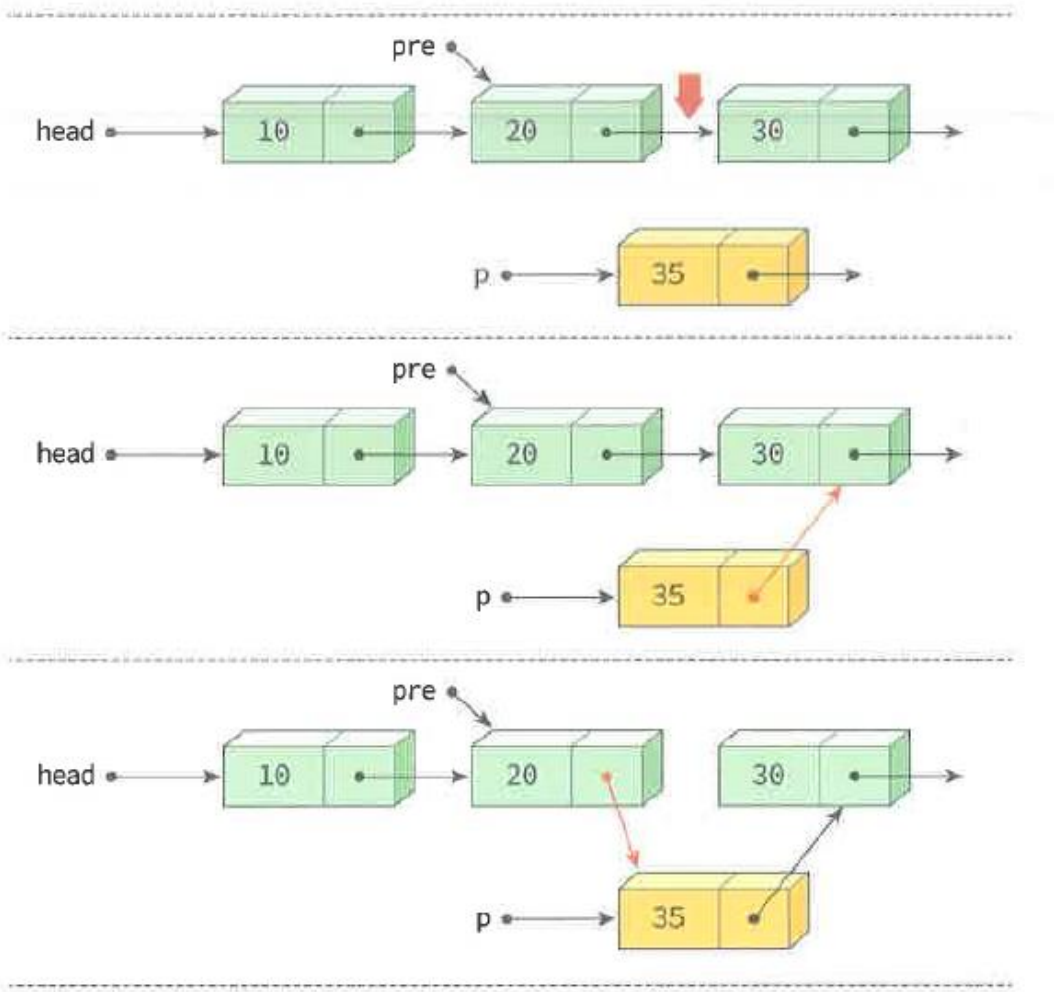


(1), (2): 새로운 노드를 생성하고 데이터 저장
(3): 새로운 노드가 현재 head가 가리키는 노드를 가리키게 함
(4): head pointer가 새로운 노드를 가리키게 함

프로그램 6.1 단순 연결 리스트의 삽입함수

```
ListNode* insert_first(ListNode *head, int value)
{
    ListNode *p = (ListNode *)malloc(sizeof(ListNode)); // (1)
    p->data = value;                                    // (2)
    p->link = head;      // 헤드 포인터의 값을 복사      // (3)
    head = p;            // 헤드 포인터 변경             // (4)
    return head;         // 변경된 헤드 포인터 반환
}
```

단순 연결 리스트_insert



[그림 6-10] 단순 연결 리스트에서의 삽입연산

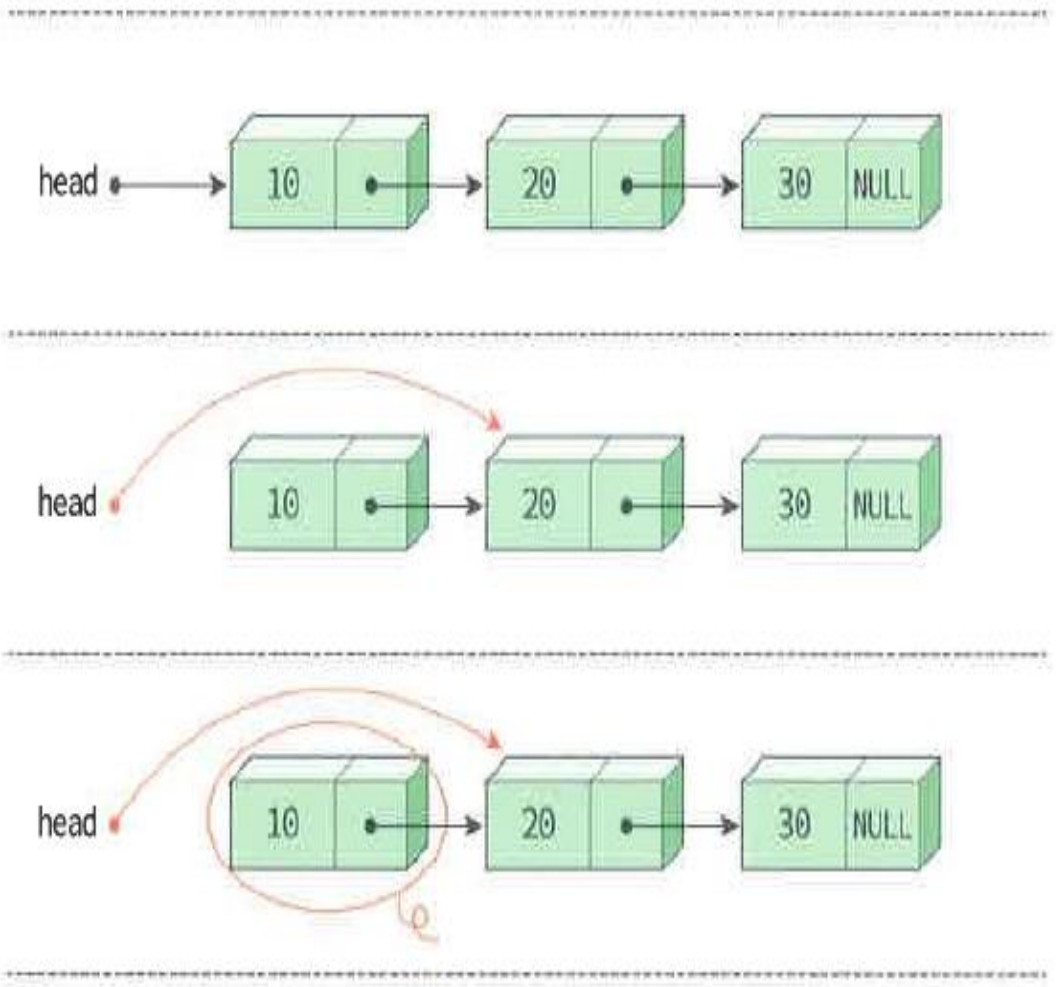
(1), (2): 새로운 노드를 생성하고 데이터 저장
(3): 새로운 노드가 현재 pre가 가리키는 노드를 가리키게 함
(4): pre 노드가 새로운 노드를 가리키게 함

프로그램 6.3 단순 연결 리스트의 삽입함수

```
// 노드 pre 뒤에 새로운 노드 삽입
ListNode* insert(ListNode *head, ListNode *pre, element value)
{
    ListNode *p = (ListNode *)malloc(sizeof(ListNode)); // (1)
    p->data = value;        // (2)
    p->link = pre->link;    // (3)
    pre->link = p;          // (4)
    return head;            // (5)
}
```

Part 1

단순 연결 리스트_delete first



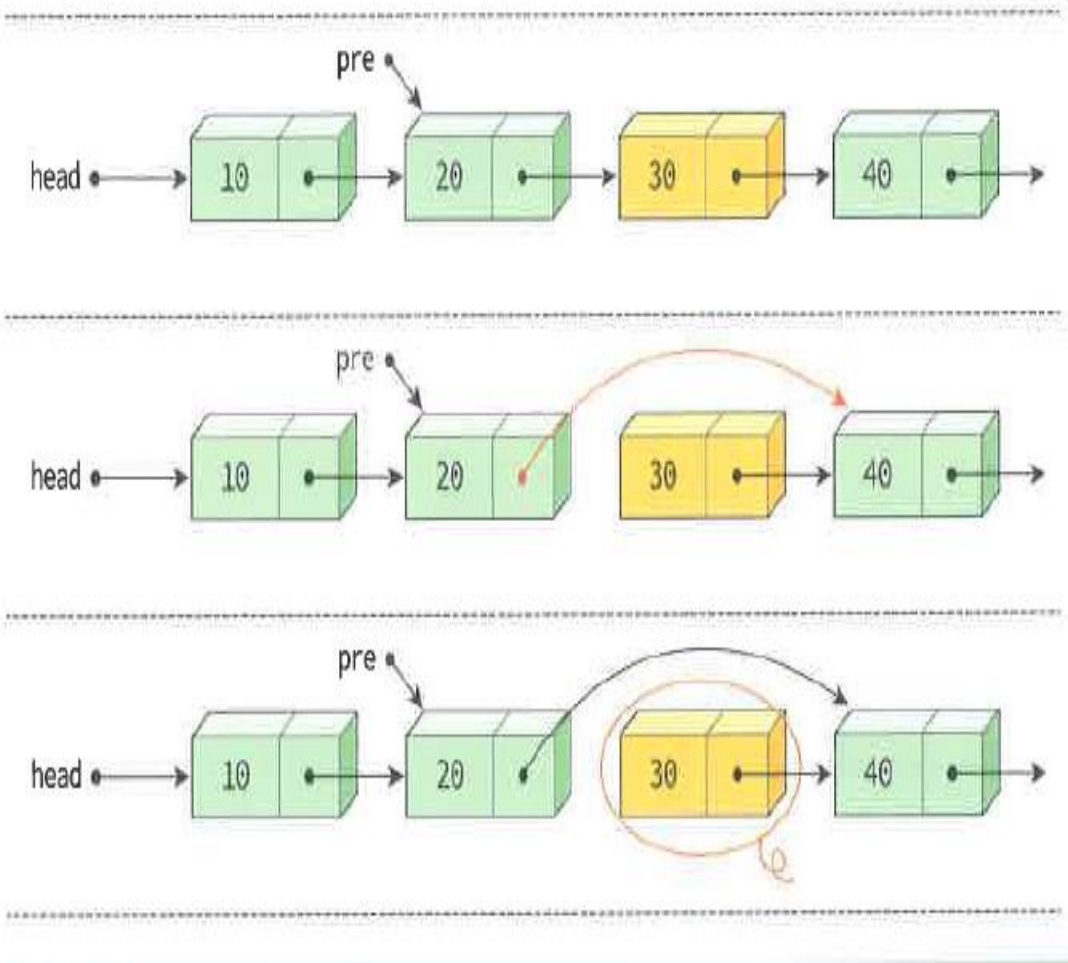
(1): head pointer의 값을 removed에 복사
(2): head pointer의 값을 removed의 link값으로 변경
(3): removed가 가리키는 메모리 반환

removed := 삭제할 노드를 가리키는 포인터

프로그램 6.4 단순 연결 리스트의 삭제함수

```
ListNode* delete_first(ListNode *head)
{
    ListNode *removed;
    if (head == NULL) return NULL;
    removed = head;          // (1)
    head = removed->link;    // (2)
    free(removed);           // (3)
    return head;             // (4)
}
```

단순 연결 리스트_delete



[그림 6-11] 단순 연결 리스트에서의 삭제연산

(1): pre pointer의 값을 removed에 복사
(2): pre pointer의 값을 removed의 link값으로 변경
(3): removed가 가리키는 메모리 반환

removed := 삭제할 노드를 가리키는 포인터

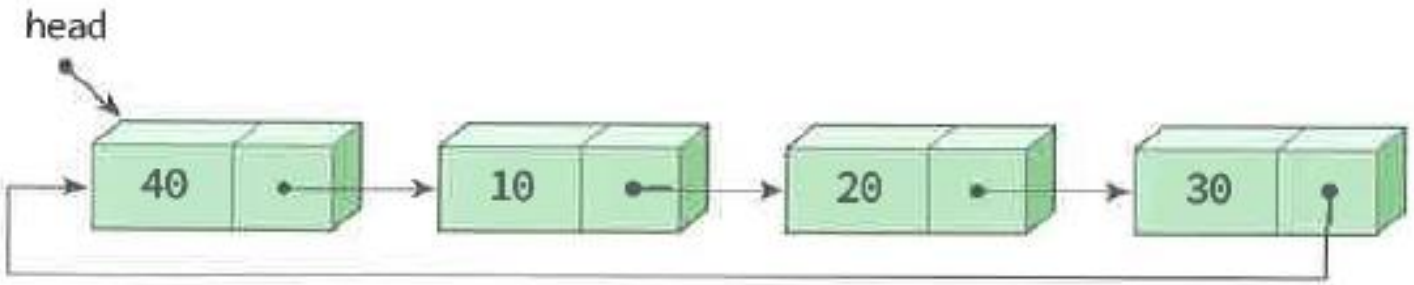
프로그램 6.5 단순 연결 리스트의 삭제함수

```
// pre가 가리키는 노드의 다음 노드를 삭제한다.
ListNode* delete(ListNode *head, ListNode *pre)
{
    ListNode *removed;
    removed = pre->link;
    pre->link = removed->link;    // (2)
```

Part 2

원형 연결 리스트

원형 연결 리스트

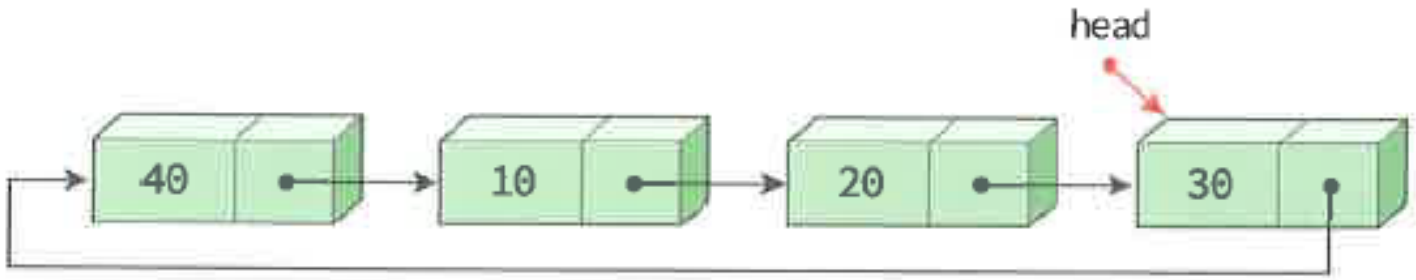


[그림 7-1] 원형 리스트

마지막 노드의 링크가 NULL이 아니라 첫 번째 노드 주소가 되는 리스트

노드의 삽입과 삭제가 단순 연결리스트보다 용이하다는 장점을 가짐
특히 리스트의 끝에 노드를 삽입하는 연산이 단순 연결리스트보다 효율적일 수 있음

소개



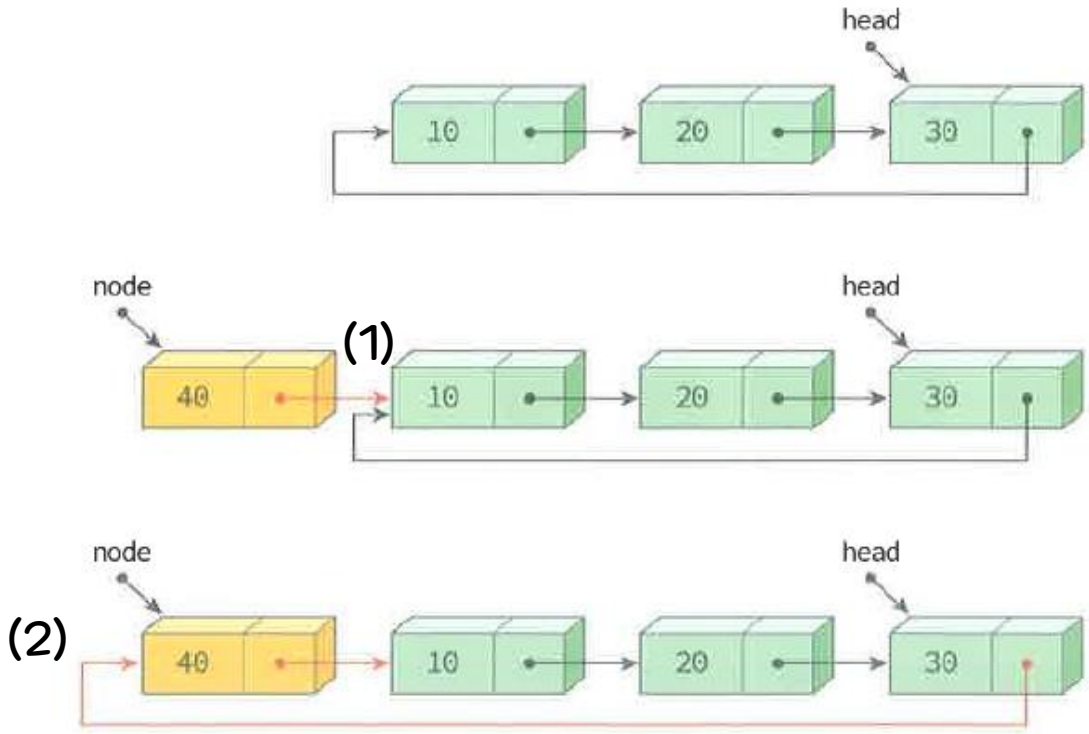
[그림 7-2] 변형된 원형 연결 리스트

head pointer가 마지막 노드를 가리키도록 하는 것

마지막 노드는 head가, 첫 번째 노드는 head->link 가 가리키고 있으므로, 리스트의 처음과 끝에 삽입하기 위해 리스트를 순회할 필요가 없음

Part 2

원형 연결 리스트 insert_first



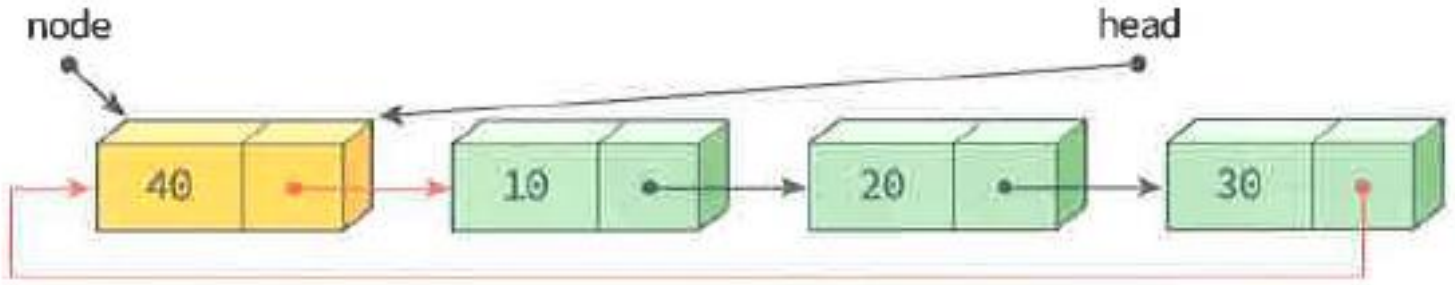
[그림 7-3] 원형 연결 리스트의 첫 번째 노드로 삽입

프로그램 7.1 원형 연결 리스트 처음에 삽입하는 함수

```
ListNode* insert_first(ListNode* head, element data)
{
    ListNode *node = (ListNode *)malloc(sizeof(ListNode));
    node->data = data;
    if (head == NULL) {
        head = node;
        node->link = head;
    }
    else {
        node->link = head->link;  // (1)
        head->link = node;        // (2)
    }
    return head; // 변경된 헤드 포인터를 반환한다.
}
```

Part 2

원형 연결 리스트 insert_last



[그림 7-4] 원형 연결 리스트의 끝에 삽입

앞선 코드에서 한 줄만 추가하면 리스트의 마지막에 삽입할 수 있음
head가 리스트의 마지막 노드를 가리키므로, head의 위치만 새로운 노드로 바꾸면 새로운 노드가 마지막 노드가 됨

```
    else {
        node->link = head->link;  // (1)
        head->link = node;        // (2)
        head = node;              // (3)
    }
    return head; // 변경된 헤드 포인터를 반환한다.
}
```

Part 3

이중 연결 리스트

단순 연결 리스트와 원형 연결 리스트의 단점

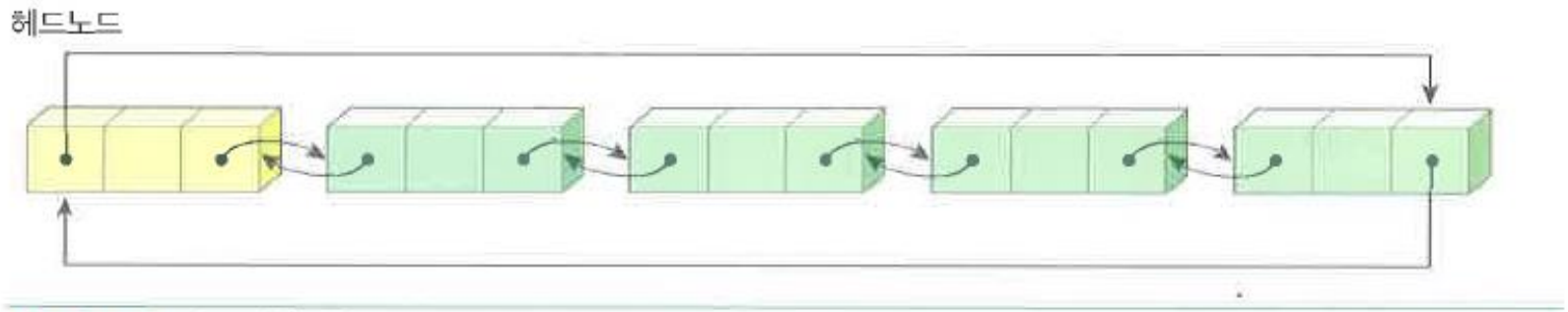
어떤 노드의 선행 노드를 찾는 것이 어려움. 원형 연결 리스트라 하더라도 거의 전체 노드를 거쳐야 함.

이를 해결하기 위한 것이 이중 연결 리스트

이중 연결 리스트

하나의 노드가 선행 노드와 후속 노드에 대한 두 개의 링크를 가지는 리스트
양방향으로 검색이 가능해지나 공간을 많이 차지하고 코드가 복잡해진다는 단점이 있음

이중 연결 리스트



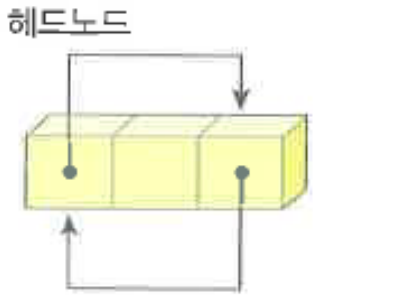
[그림 7-6] 이중 연결 리스트

실제 응용에서는 위 그림처럼 이중 연결 리스트와 원형 연결 리스트를 혼합한 형태가 많이 사용됨

헤드 노드(Head node)

데이터를 가지고 있지 않는 특별한 노드. 삽입과 삭제 알고리즘을 간편하게 하기 위하여 존재.
헤드 포인터와는 다른 것임에 유의.

이중 연결 리스트



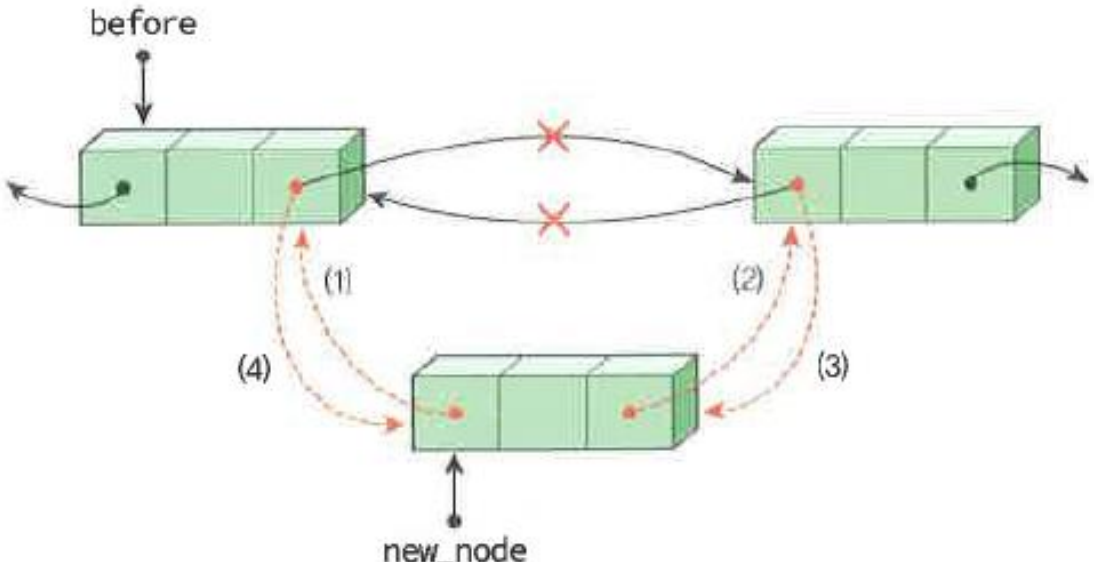
[그림 7-8] 공백상태

앞선 연결 리스트에서 공백 상태는 head == NULL인 상태였음.
그러나 이중 연결 리스트에서는 헤드 노드가 존재하기 때문에 공백 상태가 위 그림과 같은 상태가 됨.

이중 연결 리스트

```c
typedef int element;
typedef struct DListNode {    // 이중 연결 노드 타입
    element data;
    struct DListNode* llink;
    struct DListNode* rlink;
} DListNode;
```

이중 연결 리스트_삽입

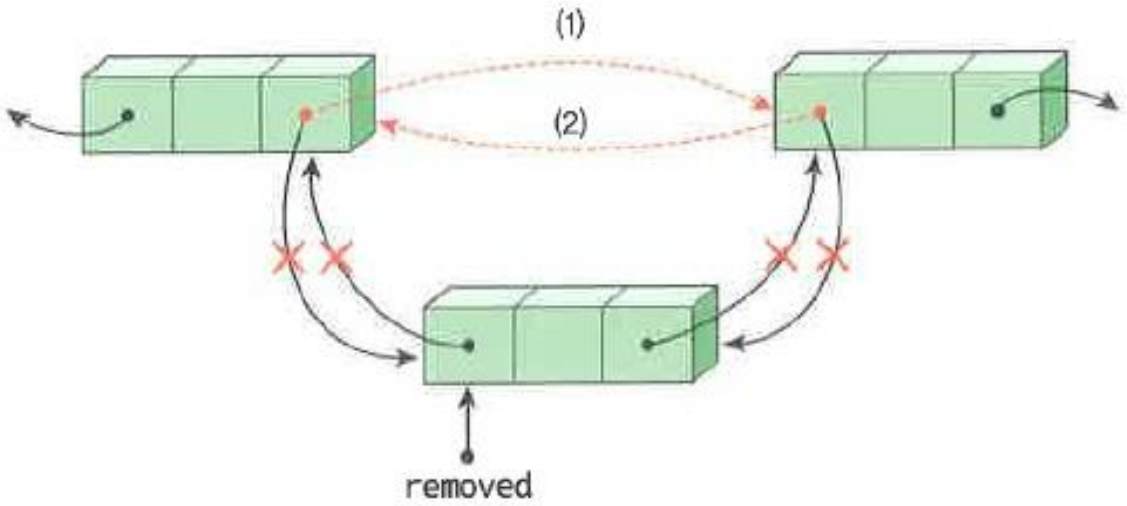


프로그램 7.5 이중 연결 리스트에서의 삽입함수

```
// 새로운 데이터를 노드 before의 오른쪽에 삽입한다.
void dinsert(DListNode *before, element data)
{
    DListNode *newnode = (DListNode *)malloc(sizeof(DListNode));
    newnode->data = data;
    newnode->llink = before; (1)
    newnode->rlink = before->rlink; (2)
    before->rlink->llink = newnode; (3)
    before->rlink = newnode; (4)
}
```

(1): new_node의 선행 노드가 before가 되게 함
(2): new_node의 후속 노드를 before의 후속 노드로 설정
(3): before의 후속 노드의 선행 노드가 new_node가 되게 함
(4): before의 후속 노드를 new_node로

이중 연결 리스트_삭제



[그림 7-10] 이중 연결 리스트에서의 삭제순서

프로그램 7.6 이중 연결 리스트에서의 삭제함수

```
// 노드 removed를 삭제한다.
void ddelete(DListNode* head, DListNode* removed)
{
    if (removed == head) return;
    removed->llink->rlink = removed->rlink; (1)
    removed->rlink->llink = removed->llink; (2)
    free(removed);
}
```

(1): removed의 이전 노드의 후속 노드가 removed의 다음 노드가 되게 함
(2): removed의 다음 노드의 선행 노드가 removed의 선행 노드가 되게 함

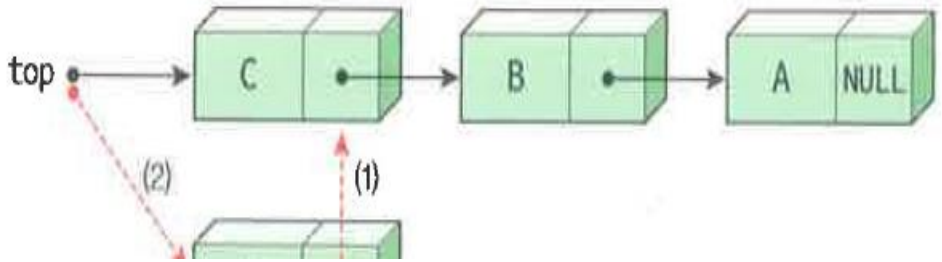
Part 4

연결 리스트를 이용한 스택&큐

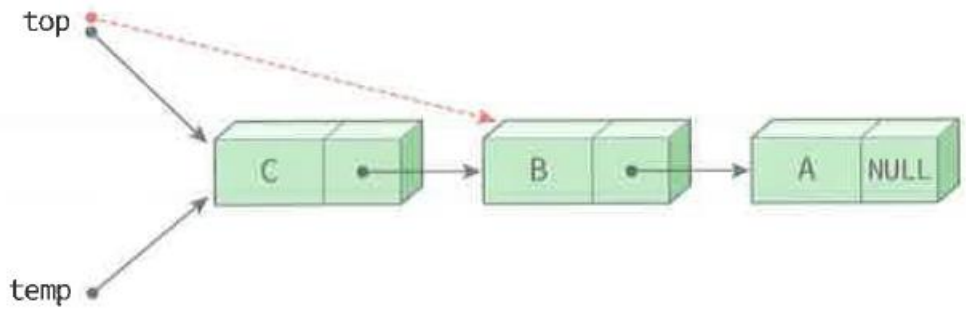
Part 3

연결 리스트를 이용한 스택

```
typedef int element;
typedef struct StackNode {
    element data;
    struct StackNode *link;
} StackNode;

typedef struct {
    StackNode *top;
} LinkedStackType;
```



삽입 연산

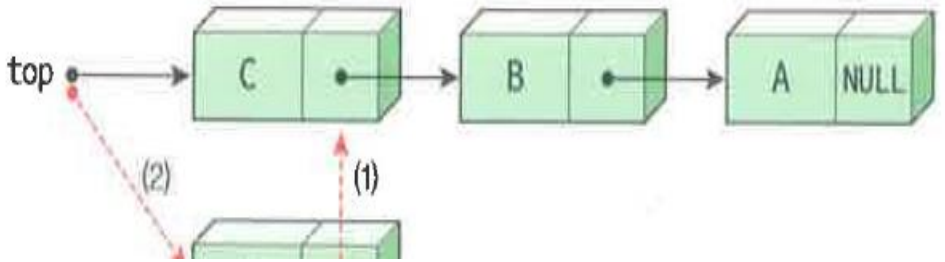


삭제 연산

연결 리스트를 이용한 스택_삽입

단순 연결 리스트에서 맨 앞에 삽입하는 것과 동일

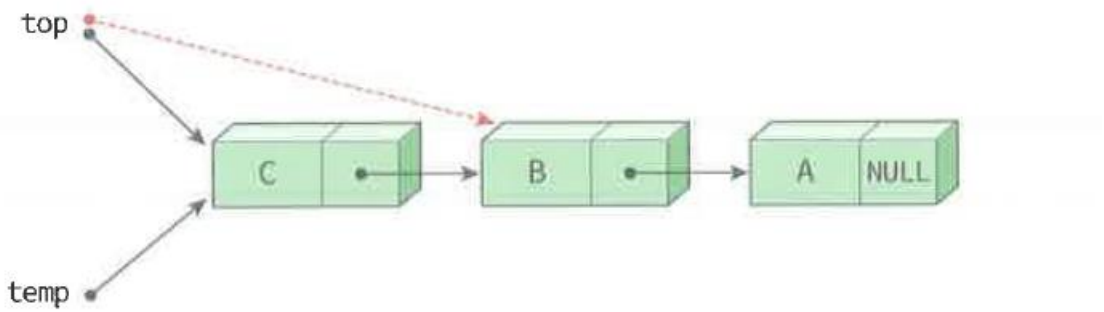
```
void push(LinkedStackType *s, element item)
{
    StackNode *temp = (StackNode *)malloc(sizeof(StackNode));
    temp->data = item;
    temp->link = s->top;
    s->top = temp;
}
```



Part 3

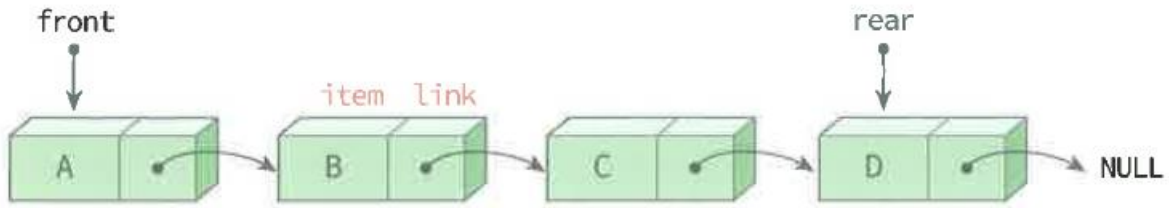
연결 리스트를 이용한 스택_삭제

```
// 삭제 함수
element pop(LinkedStackType *s)
{
    if (is_empty(s)) {
        fprintf(stderr, "스택이 비어있음\n");
        exit(1);
    }
    else {
        StackNode *temp = s->top;
        int data = temp->data;
        s->top = s->top->link;
        free(temp);
        return data;
    }
}
```



[그림 7-13] 동적 스택에서의 삽입 연산: 실선은 삽입전, 점선은 삽입후의 모습이다.

연결 리스트를 이용한 큐



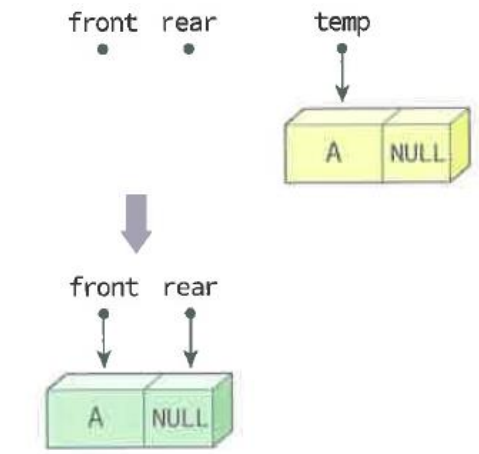
[그림 7-14] 연결리스트를 이용한 큐

기본적인 구조는 단순 연결 리스트에 front와 rear 포인터를 추가한 것

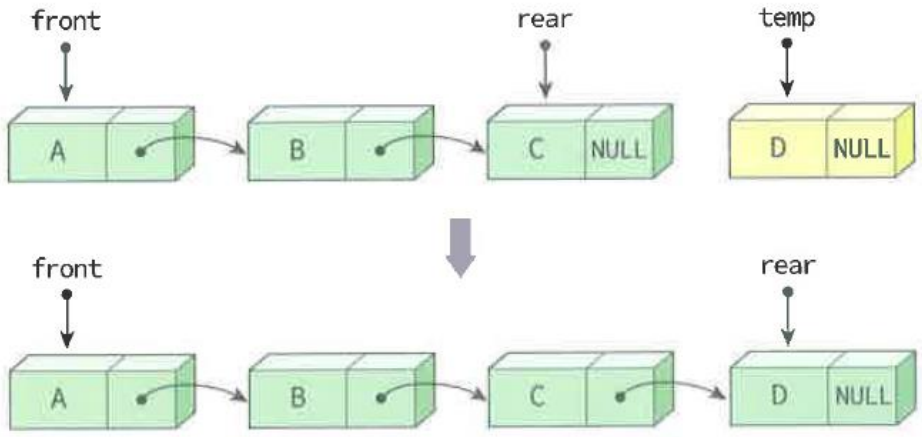
```
typedef int element;          // 요소의 타입
typedef struct QueueNode { // 큐의 노드의 타입
    element data;
    struct QueueNode *link;
} QueueNode;

typedef struct {              // 큐 ADT 구현
    QueueNode *front, *rear;
} LinkedQueueType;
```

Part 3

연결 리스트를 이용한 큐_삽입



(a) 연결된 큐가 공백상태 일 때의 삽입 연산

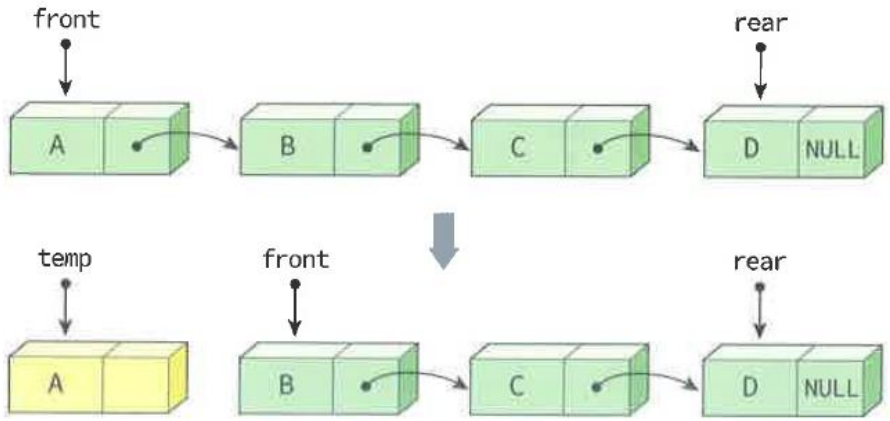


(b) 연결된 큐가 공백상태가 아닐 때의 삽입 연산

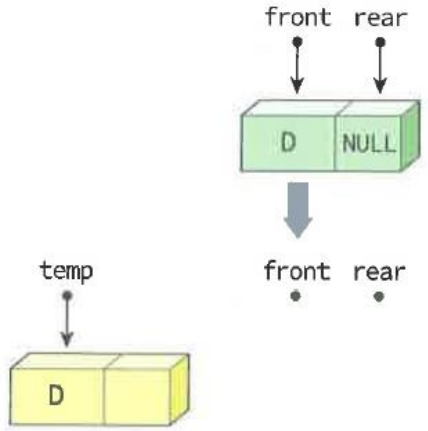
[그림 7-15] 연결된 큐에서 삽입 연산

```
void enqueuer(LinkedQueueType *q, element data)
{
   QueueNode *temp = (QueueNode *) malloc(sizeof(QueueNode));
  temp->data = data;
  temp->link = NULL;
   if (is_empty(q)) {
      q->front = temp;
      q->rear = temp;
   }
   else {
      q->rear->link=temp;
      q->rear = temp;
   }
}
```

연결 리스트를 이용한 큐_삭제



(a) 공백 상태가 아닌 연결된 큐에서의 삭제 연산



(b) 노드가 하나있는 연결된 큐에서의 삭제 연산

[그림 7-16] 연결된 큐에서 삭제 연산

프로그램 7.11 연결된 큐 삭제 연산

```
// 삭제 함수
element dequeue(LinkedQueueType *q)
{
    QueueNode *temp =q-> front;
    element data;
    if (is_empty(q)) {  // 공백상태
        fprintf(stderr, "스택이 비어있음\n");
        exit(1);
    }
    else {
        data = temp->data;              // 데이터를 꺼낸다.
        q->front = q->front->link;      // front를 다음노드를 가리키도록 한다.
        if (q->front == NULL)           // 공백 상태
            q->rear = NULL;
        free(temp);                     // 동적메모리 해제
        return data;                    // 데이터 반환
    }
}
```