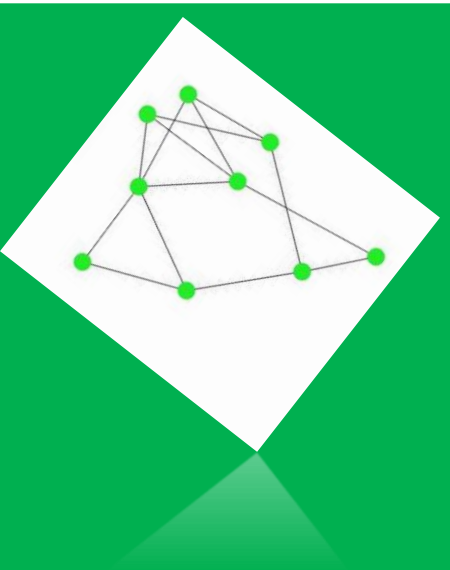
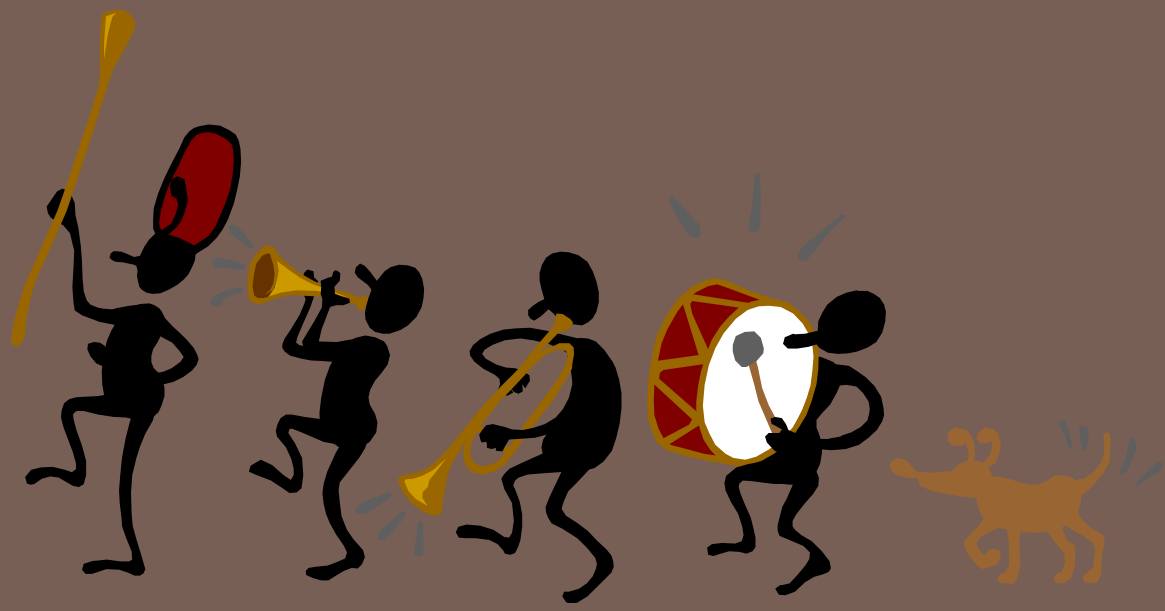


알고리즘(Algorithm)

담당교수 : 최희식
eMail:dali3054@ssu.ac.kr

6



강의 내용

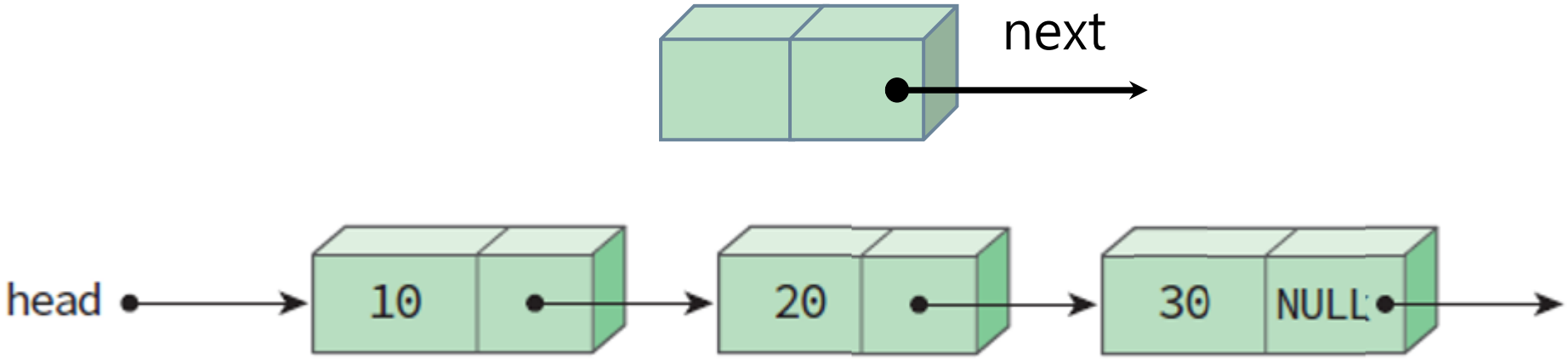
- **학습 목표**

  - **원형 연결리스트**
  - **이중 연결리스트**

1

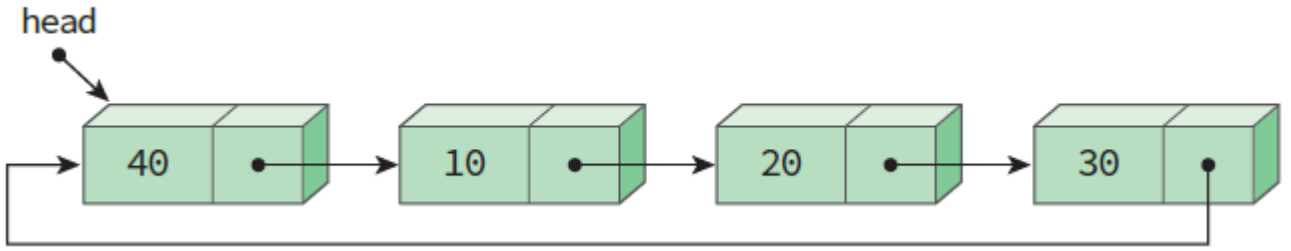
단순 연결 리스트(Singly Linked List)

- 각 노드는 저장할 데이터와 다음 노드를 가리키는 포인터로 이루어짐
- 리스트가 비어 있다면, 단일 연결 리스트에서와 같이 추가하는 노드를 head와 tail로 지정



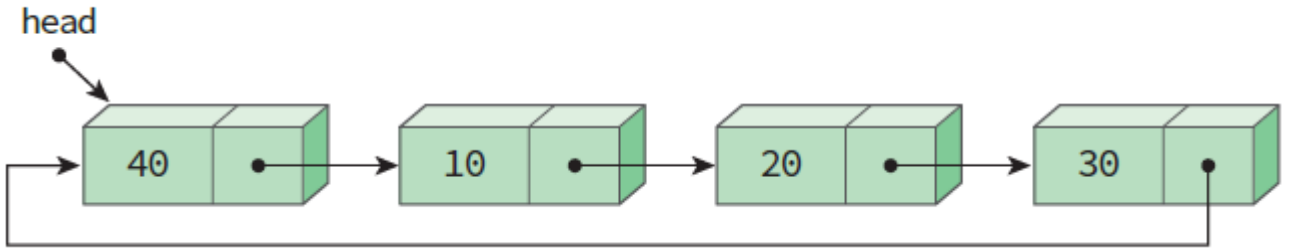
원형 연결 리스트(Circular Linked List)

- 마지막 노드의 링크가 첫 번째 노드를 가리키는 리스트
- 한 노드에서 다른 모든 노드로의 접근이 가능
- 리스트 끝에 노드를 추가하는 것이 단순 연결리스트보다 용이
  (첫번째 노드에 접근하여 모든 노드의 링크를 따라 가야할 필요가 없음)



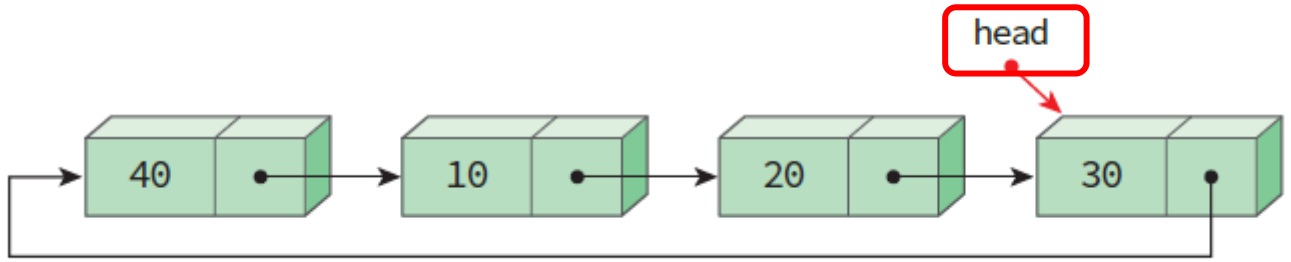
원형 연결 리스트(Circular Linked List)

- 장점
  - 하나의 노드에서 다른 모든 노드로의 접근이 가능
  - 하나의 노드에서 링크를 계속 따라가면 결국 모든 노드를 거쳐 자기 자신으로 되돌아오는 것이 가능
  - 노드의 삽입과 삭제가 단순 연결 리스트보다는 용이함
  - 특히 리스트의 끝에 노드를 삽입하는 연산이 단순 연결리스트보다 효율적
    헤드 포인터에서 시작하여 모든 노드를 거쳐 마지막에 삽입하는 것이 아니라
    헤드 포인터가 마지막 노드를 가리키도록 구성하면 리스트의 처음과 끝에 노드를 삽입할 수 있다.



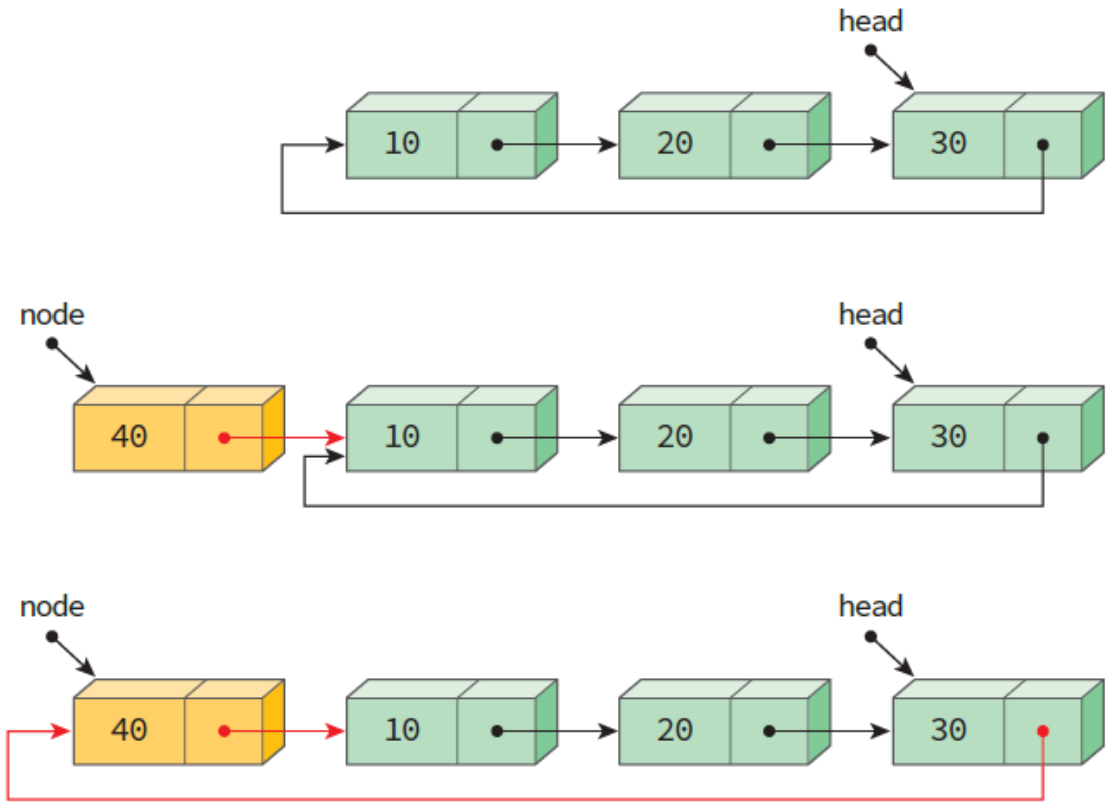
원형 연결 리스트(Circular Linked List)

➡보통 헤드포인터가 마지막 노드를 가리키게끔 구성하면 리스트의 처음이나 마지막에 노드를 삽입하는 연산이 단순 연결 리스트에 비하여 용이



- 가장 최근에 추가되었던 30 데이터를 가지고 있는 노드의 주소를 헤드포인터 값으로 변경하면 다음에 삽입할 때에는 이 노드 다음에 삽입이 됨

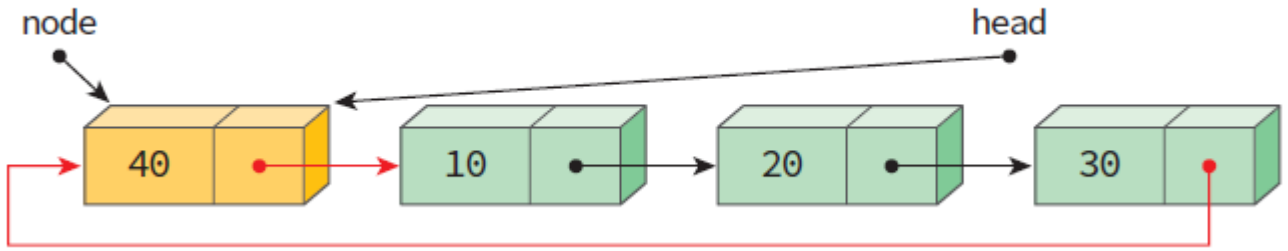
원형 연결 리스트의 처음에 삽입



원형 연결 리스트의 처음에 삽입

```
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.
}
```

원형 리스트의 끝에 삽입



원형 연결리스트 구조체 프로그램

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

typedef int element;
typedef struct ListNode {       // 노드 타입
            element data;
            struct ListNode *link;
} ListNode;
```

원형 리스트의 끝에 삽입

```
ListNode* insert_last(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)
                head = node;                    // (3)
        }
        return head;        // 변경된 헤드 포인터를 반환한다.
}
```

항목 출력 프로그램

```
void print_list(ListNode* head)
{
        ListNode* p;

        if (head == NULL) return;
        p = head->link;
        do{
                printf("%d->", p->data);
                p = p->link;
        } while (p != head->link);
}
```

테스트 프로그램

```
int main(void)
{
        ListNode *head = NULL;

        // list = 10->20->30->40
        head = insert_last(head, 20);
        head = insert_last(head, 30);
        head = insert_last(head, 40);
        head = insert_first(head, 10);
        print_list(head);
        return 0;
}
```

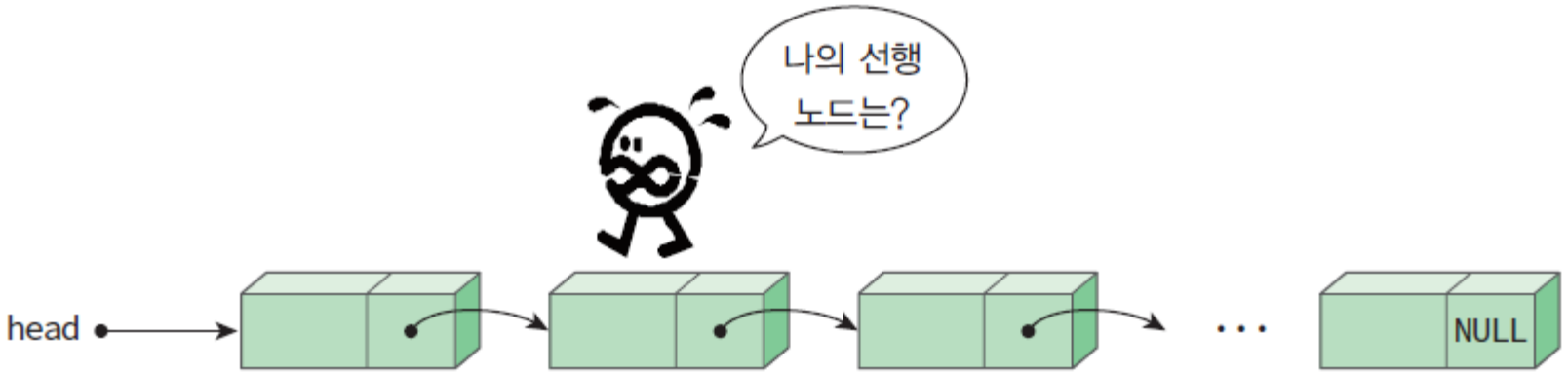
```
10->20->30->40->
```

감사합니다.

2

이중 연결 리스트

- 단순 연결 리스트는 자신의 뒤 노드를 가리키는 링크가 있다.

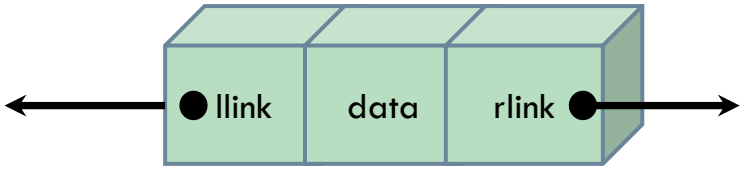


이중 연결 리스트

- 단방향 리스트 한계
  - 어떤 노드의 앞에 새로운 노드를 삽입하기 어려움
  - 삭제의 경우 항상 삭제할 노드의 앞 노드가 필요
  - 단방향의 순회만 가능.

이중 연결 리스트

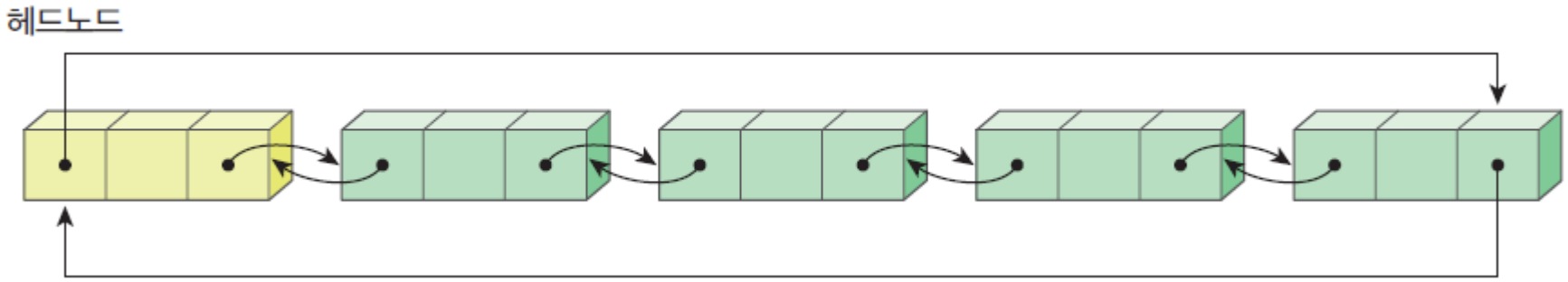
- 이중 연결 리스트는 자신의 앞 노드(previous)와 뒤 노드(nextr) 노드를 가지는 연결리스트로 양쪽 방향에 링크가 있다. (링크가 2개)
- 양항의 순회(traverse)가 가능
- 두 개의 링크 필드와 한 개의 데이터 필드로 구성
- llink(left link) 필드 : 왼쪽 링크 필드와 연결하는 포인터
- rlink(right link) 필드 : 오른쪽 링크필드와 연결하는 포인터



node

이중 연결 리스트

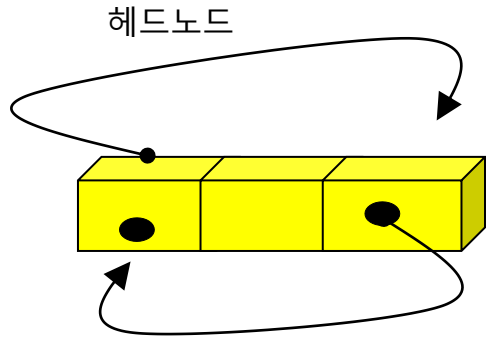
- 이중 연결 리스트: 하나의 노드가 선행 노드와 후속 노드에 대한 두 개의 링크를 가지는 리스트
- 단점은 공간을 많이 차지하고 코드가 복잡함



이중 연결 리스트

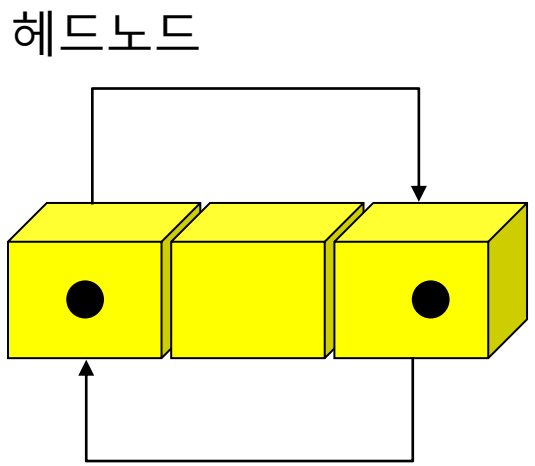
- 이중 연결 리스트: 임의의 노드를 가리키는 포인터를 p라고 하면, 다음관계가 성립한다.

```
p = p->llink->rlink = p->rlink->llink
```



헤드노드

- 헤드노드(head node): 데이터를 가지지 않고 단지 삽입, 삭제 코드를 간단하게 할 목적으로 만들어진 노드
  - 앞뒤로 똑같이 이동할 수 있음을 나타냄
  - 헤드 포인터와의 구별 필요
  - 공백상태에서는 헤드 노드만 존재



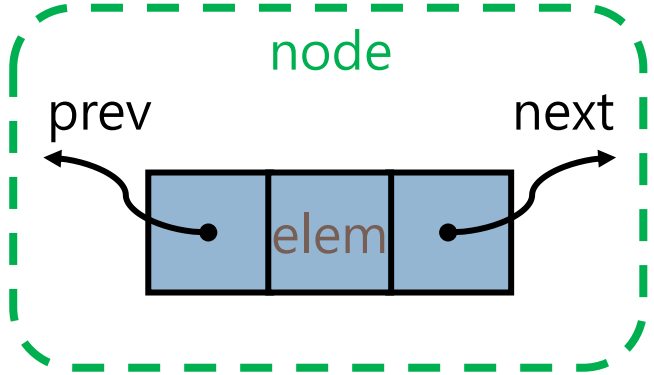
노드의 구조

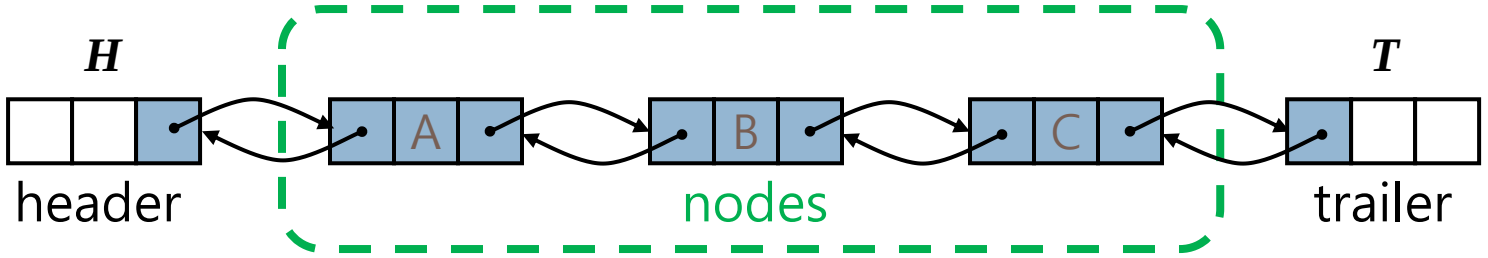
- 이중연결리스트에서의 노드의 구조

```
typedef int element;
typedef struct DlistNode {
        element data;
        struct DlistNode *llink;        //previous node
        struct DlistNode *rlink;        //next node
} DlistNode;
```

이중연결리스트

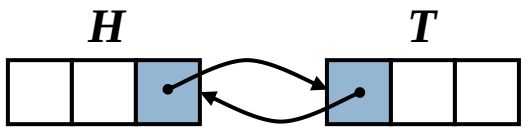
- **이중연결리스트**(doubly linked list)를 이용하면 리스트 ADT를 자연스럽게 구현할 수 있다
- 각 노드는 위치를 구현하며 각각 다음을 저장한다
  - **원소**
  - 이전 노드를 가리키는 **링크**
  - 다음 노드를 가리키는 **링크**
- 특별 헤더 및 트레일러 노드





초기화

- 초기에는 아무 노드도 없다
- **O**(1) 시간 소요



Alg ***initialize***()
input none
output an empty doubly linked list with header ***H*** and trailer ***T***

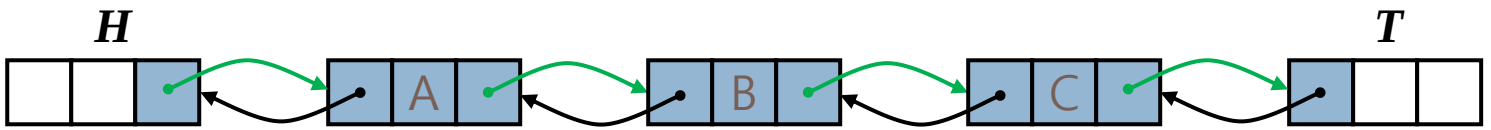
1. ***H*** ← ***getnode***()
2. ***T*** ← ***getnode***()
3. ***H*.next** ← ***T***
4. ***T*.prev** ← ***H***
5. ***n*** ← 0 {optional}
6. **return**

순회

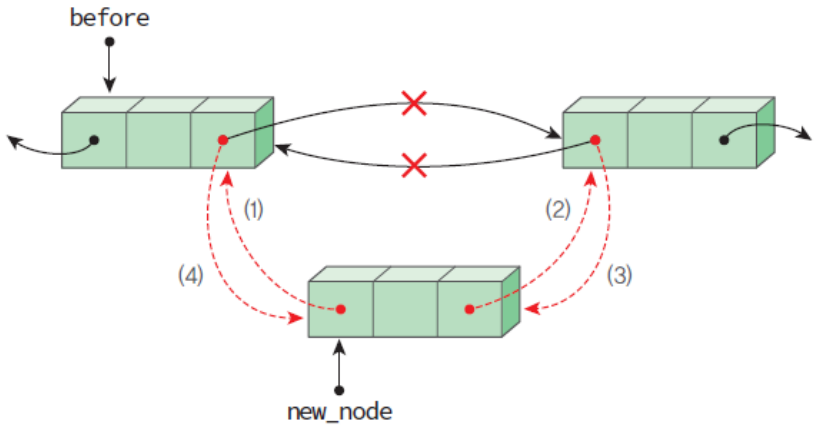
- 연결리스트의 모든 원소들을 방문
- **O(*n*)** 시간 소요

Alg ***traverse***()
input a doubly linked list with header ***H*** and trailer ***T***
output none

1. ***p*** ← ***H*****.next**
2. **while** (***p*** ≠ ***T***)
   visit(***p*****.elem**) {print, etc}
   p ← ***p*****.next**
3. **return**

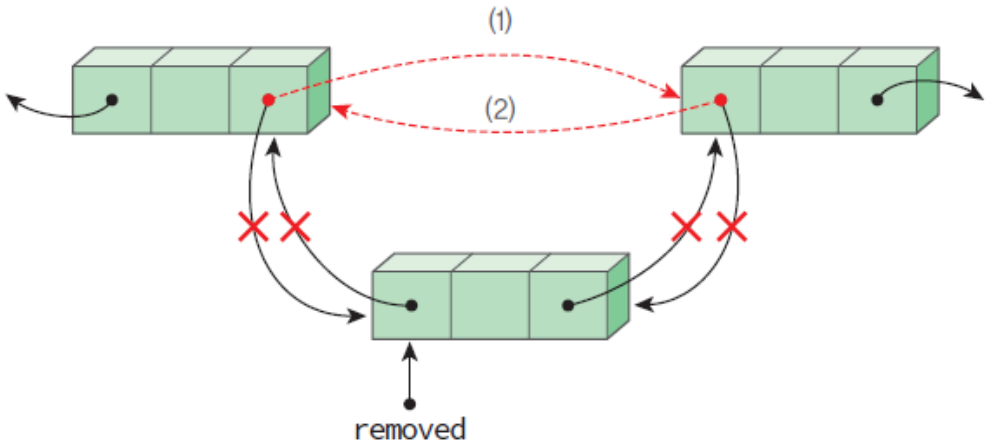


삽입연산




```
// 새로운 데이터를 노드 before의 오른쪽에 삽입한다.
void dinsert(DListNode *before, element data)
{
        DListNode *newnode = (DListNode *)malloc(sizeof(DListNode);
        strcpy(newnode->data, data);
        newnode->llink = before;        //왼쪽
        newnode->rlink = before->rlink;   //오른쪽
        before->rlink->llink = newnode;   //새 노드를 왼쪽
        before->rlink = newnode;        //새 노드를 오른쪽
}
```

삭제연산




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

테스트 프로그램

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

typedef int element;
typedef struct DListNode {     // 이중연결 노드 타입
            element data;
            struct DListNode* llink;
            struct DListNode* rlink;
} DListNode;

// 이중 연결 리스트를 초기화
void init(DListNode* phead)
{
            phead->llink = phead;
            phead->rlink = phead;
}
```

테스트 프로그램

```
// 이중 연결 리스트의 노드를 출력
void print_dlist(DListNode* phead)
{
        DListNode* p;
        for (p = phead->rlink; p != phead; p = p->rlink) {
                printf("<-| |%d| |-> ", p->data);
        }
        printf("\n");
}
// 새로운 데이터를 노드 before의 오른쪽에 삽입한다.
void dinsert(DListNode *before, element data)
{
        DListNode *newnode = (DListNode *)malloc(sizeof(DListNode));
        strcpy(newnode->data, data);
        newnode->llink = before;
        newnode->rlink = before->rlink;
        before->rlink->llink = newnode;
        before->rlink = newnode;
}
```

테스트 프로그램

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

테스트 프로그램

```
// 이중 연결 리스트 테스트 프로그램
int main(void)
{
        DListNode* head = (DListNode *)malloc(sizeof(DListNode));
        init(head);
        printf("추가 단계\n");
        for (int i = 0; i < 5; i++) {
                // 헤드 노드의 오른쪽에 삽입
                dinsert(head, i);
                print_dlist(head);
        }
        printf("\n삭제 단계\n");
        for (int i = 0; i < 5; i++) {
                print_dlist(head);
                ddelete(head, head->rlink);
        }
        free(head);
        return 0;
}
```

실행 결과

```
추가 단계
<-| |0| |->
<-| |1| |-> <-| |0| |->
<-| |2| |-> <-| |1| |-> <-| |0| |->
<-| |3| |-> <-| |2| |-> <-| |1| |-> <-| |0| |->
<-| |4| |-> <-| |3| |-> <-| |2| |-> <-| |1| |-> <-| |0| |->

삭제 단계
<-| |4| |-> <-| |3| |-> <-| |2| |-> <-| |1| |-> <-| |0| |->
<-| |3| |-> <-| |2| |-> <-| |1| |-> <-| |0| |->
<-| |2| |-> <-| |1| |-> <-| |0| |->
<-| |1| |-> <-| |0| |->
<-| |0| |->
```

감사합니다.

3

감사합니다.