

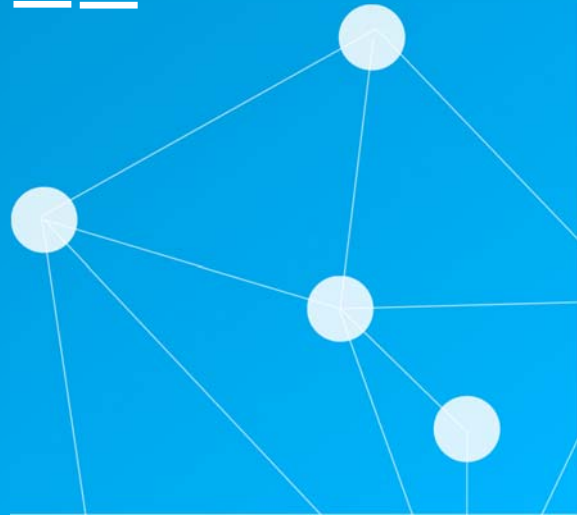
- 리스트의 개념과 추상 자료형을 이해한다.
- 리스트를 배열로 구현하는 방법을 이해한다.
- 리스트를 연결 리스트로 구현하는 방법을 이해한다.
- 시작 노드 표현 방법 2가지를 이해한다.
- 다양한 형태의 연결 리스트를 이해한다.
- 리스트를 이용한 프로그래밍 능력을 배양한다.

06

CHAPTER

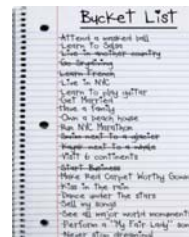
리스트

- 6.1 리스트란?
- 6.2 배열로 구현한 리스트
- 6.3 연결 리스트로 구현된 리스트
- 6.4 다양한 형태의 연결 리스트
- 6.5 연결 리스트의 응용: 라인 편집기



6.1 리스트란?

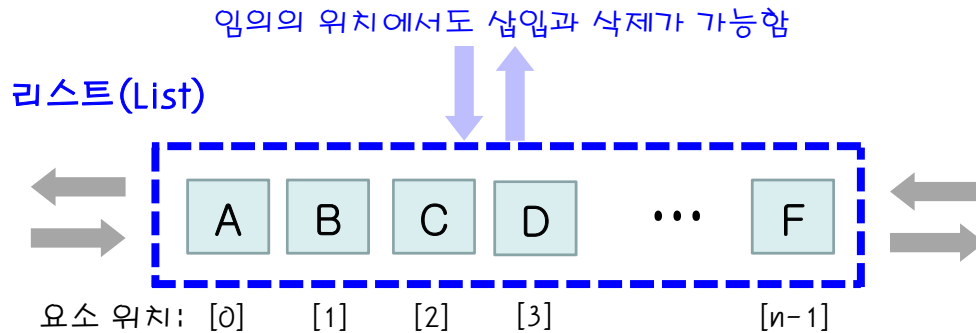
- 리스트(list), 선형리스트(linear list)
 - 순서를 가진 항목들의 모임
 - 집합: 항목간의 순서의 개념이 없음
- 리스트의 예
 - 요일: (일요일, 월요일, ..., 토요일)
 - 한글 자음의 모임: (ㄱ, ㄴ, ..., ㅎ)
 - 카드: (Ace, 2, 3, ..., King)
 - 핸드폰의 문자 메시지 리스트
 - 다항식의 각 항들



리스트의 구조



- Stack, Queue, Deque과의 공통점과 차이점
 - 선형 자료구조
 - 자료의 접근(삽입, 삭제) 위치
 - 리스트는 임의의 위치에 있는 항목에 대한 연산을 허용



리스트의 연산



- 기본 연산
 - 리스트의 어떤 위치에 새로운 요소를 삽입한다.
 - 리스트의 어떤 위치에 있는 요소를 삭제한다.
 - 리스트의 어떤 위치에 있는 요소를 반환한다.
 - 리스트가 비었는지를 살핀다.
 - 리스트가 가득 차 있는지를 체크한다.
- 고급 연산
 - 리스트에 어떤 요소가 있는지를 살핀다.
 - 리스트의 어떤 위치에 있는 요소를 새로운 요소로 대체한다.
 - 리스트 안의 요소의 개수를 센다.
 - 리스트 안의 모든 요소를 출력한다.

리스트 ADT



데이터: 임의의 접근 방법을 제공하는 같은 타입 요소들의 순서 있는 모임
연산:

- `init()`: 리스트를 초기화한다.
- `insert(pos, item)`: `pos` 위치에 새로운 요소 `item`을 삽입한다.
- `delete(pos)`: `pos` 위치에 있는 요소를 삭제한다.
- `get_entry(pos)`: `pos` 위치에 있는 요소를 반환한다.
- `is_empty()`: 리스트가 비어있는지를 검사한다.
- `is_full()`: 리스트가 가득 차 있는지를 검사한다.
- `find(item)`: 리스트에 요소 `item`이 있는지를 살핀다.
- `replace(pos, item)`: `pos` 위치를 새로운 요소 `item`으로 바꾼다.
- `size()`: 리스트안의 요소의 개수를 반환한다.

5

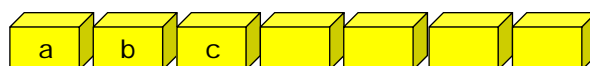
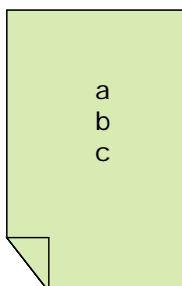
6.2 리스트 구현 방법



- 배열을 이용
 - 구현이 간단
 - 삽입, 삭제 시 오버헤드
 - 항목의 개수 제한
- 연결리스트를 이용
 - 구현이 복잡
 - 삽입, 삭제가 효율적
 - 크기가 제한되지 않음

리스트 ADT

배열을 이용한 구현



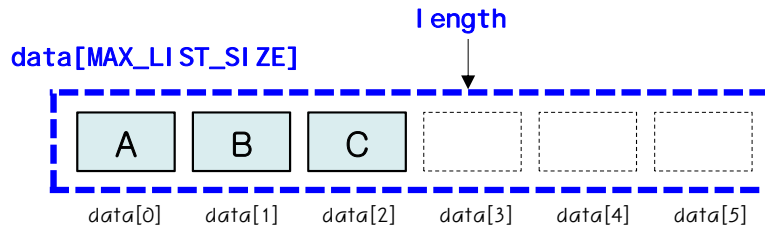
연결리스트를 이용한 구현



6

배열로 구현된 리스트

- 1차원 배열에 항목들을 순서대로 저장
 - $L=(A, B, C)$

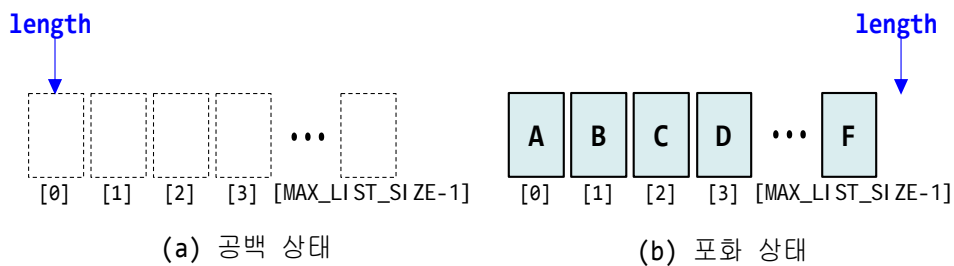


```
typedef int Element;  
Element data[MAX_LIST_SIZE];  
int length = 0;
```

7

공백상태 / 포화상태

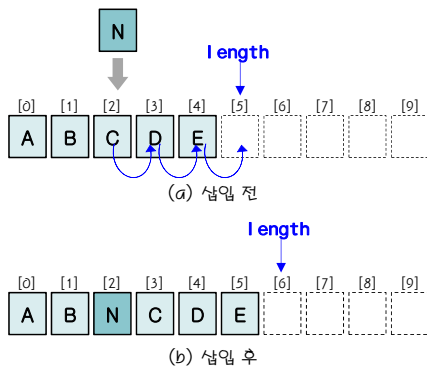
- 공백상태 / 포화상태



8

삽입 연산

- 삽입위치 다음의 항목들을 이동하여야 함.



```
void insert( int pos, Element e )
{
    int i;

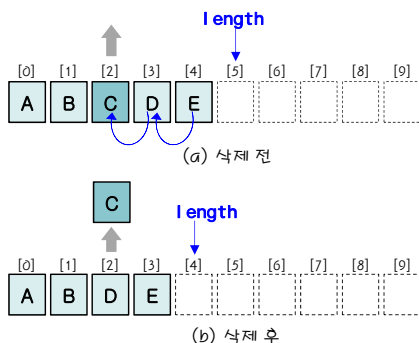
    if( is_full() == 0 && pos >= 0 && pos <= length ) {
        for( i = length; i > pos; i-- )
            data[i] = data[i-1];

        data[pos] = e;
        length++;
    }
    else error("포화상태 오류 또는 삽입 위치 오류");
}
```

9

삭제 연산

- 삭제위치 다음의 항목들을 이동하여야 함



```
void delete( int pos )
{
    int i;

    if( is_empty() == 0 && 0 <= pos && pos < length ) {
        for( i = pos+1; i < length; i++ )
            data[i-1] = data[i];

        length--;
    }
    else error("공백상태 오류 또는 삭제 위치 오류");
}
```

10

전체 프로그램

```
void main()
{
    init_list();

    insert(0, 10);
    insert(0, 20);
    insert(1, 30);
    insert(size(), 40);
    insert(2, 50);
    print_list("배열로 구현한 List(삽입x5)");

    replace(2, 90);
    print_list("배열로 구현한 List(교체x1)");

    delete(2);
    delete(size() - 1);
    delete(0);
    print_list("배열로 구현한 List(삭제x3)");

    clear_list();
    print_list("배열로 구현한 List(정리후)");
}
```

C:\WINDOWS\system32\cmd.exe

배열로 구현한 List(삽입x5) [5]: 20 30 50 10 40

배열로 구현한 List(교체x1) [5]: 20 30 90 10 40

배열로 구현한 List(삭제x3) [2]: 30 10

배열로 구현한 List(정리후) [0]:

계속하려면 아무 키나 누르십시오 . . .

5번의 insert() 연산 결과

교체 연산 결과 (2번 항목)

3번 delete () 연산 결과

11

- A라는 공백상태의 리스트가 있다고 가정하자. 이 리스트에 대하여 다음과 같은 연산들이 적용된 후의 리스트의 내용을 그려라.

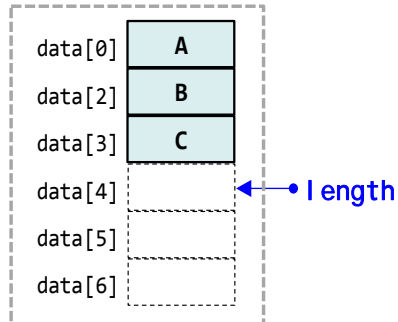
```
add_first(A, "first");
add(A, 1, "second");
add_last(A, "third");
add(A, 2, "fourth");
add(A, 4, "fifth");
delete(A, 2);
delete(A, 2);
replace(A, 3, "sixth");
```

12

6.3 연결 리스트로 구현한 리스트

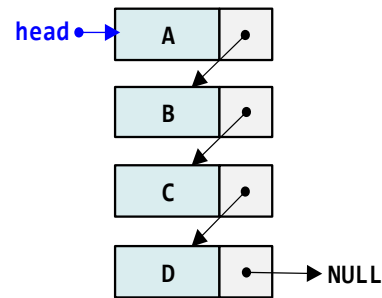
- 단순 연결 리스트(singly linked list) 사용
 - 하나의 링크 필드를 이용하여 연결
 - 마지막 노드의 링크 값은 NULL

```
Element data[MAX_LIST_SIZE]  
int length;
```



배열을 이용한 리스트

```
Node* head;
```



연결 리스트를 이용한 리스트

- 다음은 순차적 표현(배열)과 연결 리스트를 비교한 것이다. 설명이 틀린 것을 모두 고르시오.
 1. 연결 리스트는 포인터를 가지고 있어 상대적으로 크기가 작아진다.
 2. 연결 리스트는 삽입이 용이하다.
 3. 순차적 표현은 연결 리스트 보다 접근 시간이 많이 걸린다.
 4. 연결 리스트로 작성된 리스트를 2개로 분리하기가 쉽다.

데이터



- 구조체

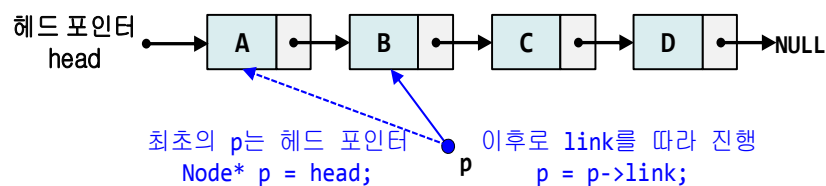
```
#define Element      int
typedef struct Li nkedNode {
    Element data;           // 데이터 필드
    struct Li nkedNode* l i nk; // 링크 필드
} Node;
```

- 데이터

```
Node* head; // 헤드 포인터
```

15

단순한 연산들



```
int size()
{
    Node* p;
    int count = 0;
    for (p = head; p != NULL; p = p->link)
        count++;
    return count;
}
```

```
Node* get_entry(int pos)
{
    Node* n = head;
    int i;
    for (i = 0; i < pos; i++, n = n->link)
        if (n == NULL) return NULL;
    return n;
}
```

16

항목 교체 및 탐색 연산

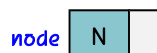
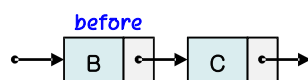


```
void replace(int pos, Element val)
{
    Node* node = get_entry(pos);
    if (node != NULL)
        node->data = val;
}

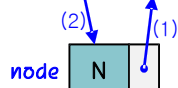
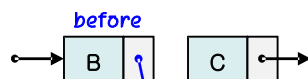
Node* find(Element val)
{
    Node* p;
    for (p = head; p != NULL; p = p->link)
        if (p->data == val) return p;
    return NULL;
}
```

17

삽입연산



(a) 삽입하기 전



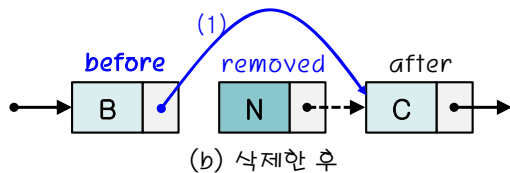
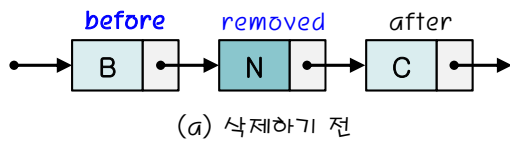
(b) 삽입한 후

```
void insert_next(Node *before, Node *n)
{
    if (n != NULL) {
        n->link = before->link;
        before->link = n;
    }
}

void insert(int pos, Element val)
{
    Node *new_node, *prev;
    new_node = (Node*)malloc(sizeof(Node));
    new_node->data = val;
    new_node->link = NULL;
    if (pos == 0)
        new_node->link = head;
        head = new_node;
    else {
        prev = get_entry(pos - 1);
        if (prev != NULL)
            insert_next(prev, new_node);
        else free(new_node);
    }
}
```

18

삭제연산



```
Node* remove_next(Node *before)
{
    Node* removed = before->link;
    if (removed != NULL)
        before->link = removed->link;
    return removed;
}

void delete(int pos)
{
    Node* prev, *removed;
    if (pos == 0 && is_empty() == 0) {
        removed = head;
        head = head->link;
        free(removed);
    }
    else {
        prev = get_entry(pos - 1);
        if (prev != NULL) {
            removed = remove_next(prev);
            free(removed);
        }
    }
}
```

19

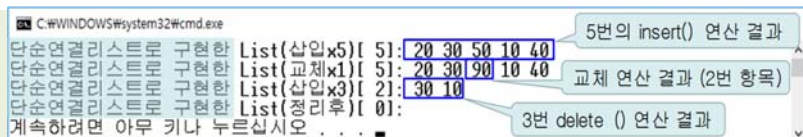
전체 프로그램

```
void main()
{
    init_list();
    insert(0, 10);
    insert(0, 20);
    insert(1, 30);
    insert(size(), 40);
    insert(2, 50);
    print_list("단순연결리스트로 구현한 List(삽입x5)");

    replace(2, 90);
    print_list("단순연결리스트로 구현한 List(교체x1)");

    delete(2);
    delete(size()-1);
    delete(0);
    print_list("단순연결리스트로 구현한 List(삽입x3)");

    clear_list();
    print_list("단순연결리스트로 구현한 List(정리후)");
}
```



20



- 단순 연결 리스트에서 첫 번째 노드를 가리키고 있는 포인터를 ()라고 한다.
- 노드는 데이터 필드와 ()필드로 이루어져 있다.
- 배열에 비하여 연결 리스트는 어떤 장점을 가지는가?

문제



- 덱(deque)은 삽입과 삭제가 양끝에서 임의로 수행되는 자료구조이다. 단순 연결리스트로 덱을 구현한다고 할 때 $O(1)$ 시간 내에 수행할 수 없는 연산은? (단, first와 last는 각각 덱의 첫 번째 원소와 마지막 원소를 가리키며, 연산이 수행된 후에도 덱의 원형이 유지되어야 한다)
 1. insertFirst 연산: 덱의 첫 번째 원소로 삽입
 2. insertLast 연산: 덱의 마지막 원소로 삽입
 3. deleteFirst 연산: 덱의 첫 번째 원소를 삭제
 4. deleteLast 연산: 덱의 마지막 원소를 삭제

문제



- 연결리스트를 역순으로 변환하는 함수 reverse를 완성하시오.

```
Node *reverse(Node *list)
{
    Node *p, *q, *r;
    p = list; //p는 역순으로 만들 리스트
    q = NULL; //q는 역순으로 만들 노드
    while (p != NULL) {
        r = q; // r은 역순으로 된 리스트, r은 q, q는 p를 따름
        q = p;
        p = p->link;
        (                ) // q의 링크 방향을 바꾼다
    }
    return q;
}
```

23



- 단순 연결 리스트에서 하나의 노드를 삭제하려면 어떤 노드를 가리키는 포인터 변수가 필요한가?

24



- 단순 연결 리스트의 노드 포인터 last가 마지막 노드를 가리킨다고 할 때 다음 수식 중, 참인 것은?
 1. `last == NULL`
 2. `last->data == NULL`
 3. `last->link == NULL`
 4. `last->link->link == NULL`



- 단순 연결 리스트의 노드들을 노드 포인터 p로 탐색하고자 한다. p가 현재 가리키는 노드에서 다음 노드로 가려면 어떻게 하여야 하는가?
 1. `p++;`
 2. `p--;`
 3. `p=p->link;`
 4. `p=p->data;`



- 단순 연결 리스트의 관련 함수 f가 헤드 포인터 head를 변경시켜야 한다면 함수 파라미터로 무엇을 받아야 하는가?
 1. head
 2. &head
 3. *head
 4. head->link;

27



- 단순 연결 리스트에서 다음 함수와 프로그램을 작성하여라.
 - 정수가 저장 되어 있는 단순 연결 리스트의 모든 데이터 값을 더한 합을 출력하는 프로그램을 작성하여라.
 - 특정한 데이터값을 갖는 노드의 개수를 계산하는 함수를 작성하라.
 - 탐색 함수를 참고하여 특정한 데이터값을 갖는 노드를 삭제하는 함수를 작성하라.

28

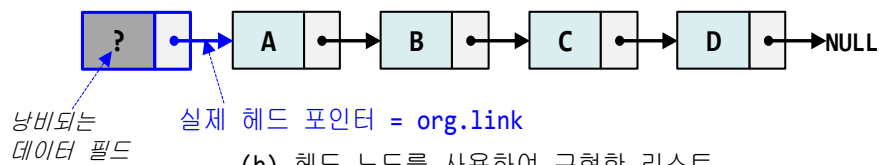
헤드포인터와 헤드 노드

헤드포인터 (head)



(a) 헤드 포인터를 사용하여 구현한 리스트

헤드 노드 (org)



(b) 헤드 노드를 사용하여 구현한 리스트

```
Node org; // 헤드 노드 org. 실제 헤드 포인터는 org.link가 됨
```

- 헤드 노드 사용 이유는?

29

실습

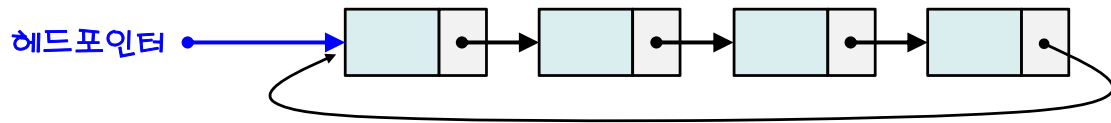
- 헤드 포인터 대신에 헤드 노드를 사용하여 프로그램 6.13을 재구현하라.
 - get_entry(-1) 연산이 헤드 노드의 주소를 반환하도록 수정
 - insert(pos, e)와 delete(pos) 연산을 간단한 버전으로 수정
 - 리스트의 모든 노드가 선행 노드를 갖게 됨

30

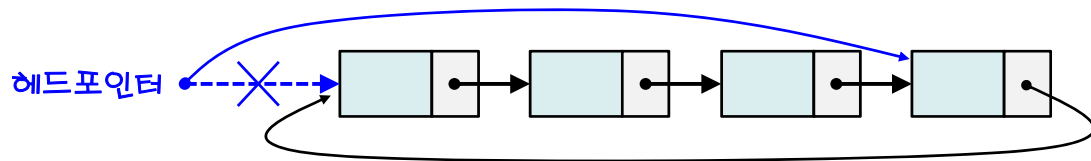
6.4 원형 연결 리스트



- Circular Linked List



- 변형된 원형 연결 리스트



31

- 단순 연결 리스트에 비하여 원형 연결 리스트의 장점은?



32



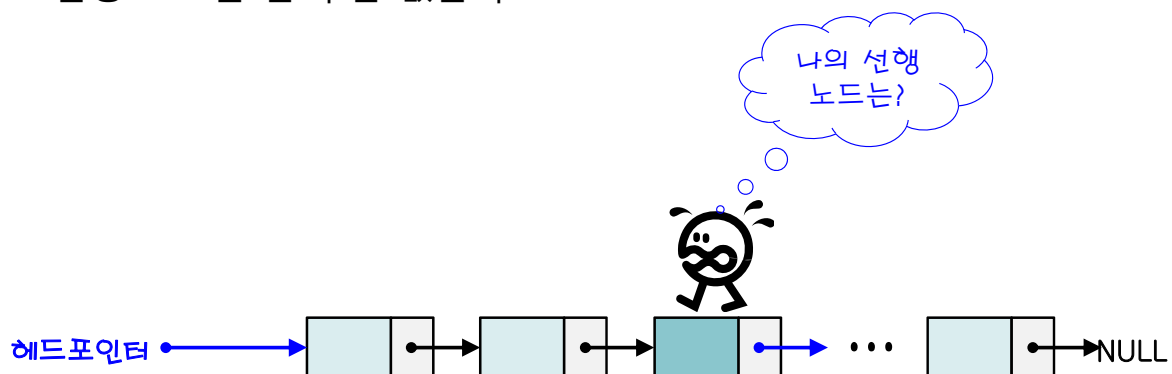
- 원형 연결 리스트에 존재하는 노드의 수를 계산하는 함수 `get_length()`를 작성하라.
- 보통 연결 리스트에서는 선행 노드를 알아야만 노드를 삭제할 수 있다. 그러나 다음과 같이 하면 선행 노드를 모르고도 노드를 삭제할 수 있다. 먼저 원형 연결 리스트라고 가정하자. 어떤 노드를 가리키는 포인터 `x`가 주어진 경우, 그 노드의 후속 노드를 쉽게 찾을 수 있다. 후속 노드를 `y`라고 하면 `x`에 `y`의 데이터 필드 값을 복사한다. 그리고 `y`를 삭제한다. 그러면 실질적으로는 `x`가 삭제된 것처럼 된다. 이런 식으로 노드를 삭제하는 함수 `remove_node3`를 작성하고 테스트하라.

33

이중 연결 리스트



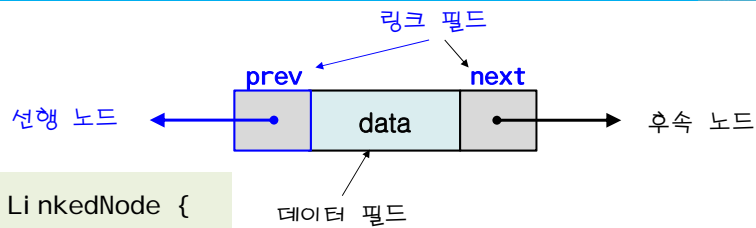
- 후속 노드는 쉽게 알 수 있다.(링크 필드)
- 선행 노드를 알 수는 없을까?



34

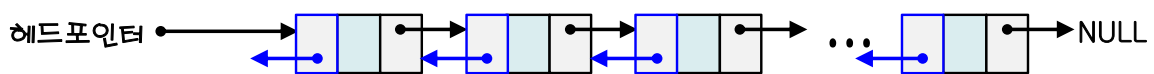
이중 연결 리스트 구조

- 노드 구조



```
typedef struct Dbl Li nkedNode {
    El ement data;
    struct Dbl Li nkedNode* prev;
    struct Dbl Li nkedNode* next;
} Node;
```

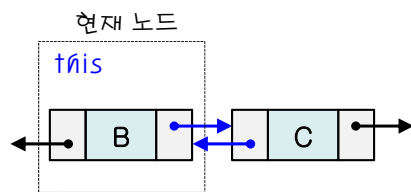
- 이중 연결 리스트의 구조



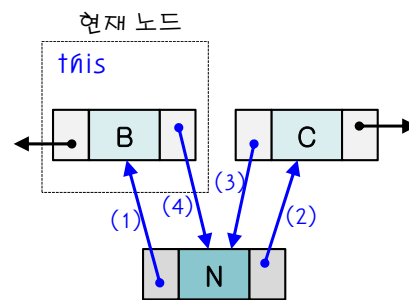
$p == p \rightarrow next \rightarrow prev == p \rightarrow prev \rightarrow next$

35

삽입연산



(a) 현재 노드 B에서 새 노드 N을 삽입하기 전

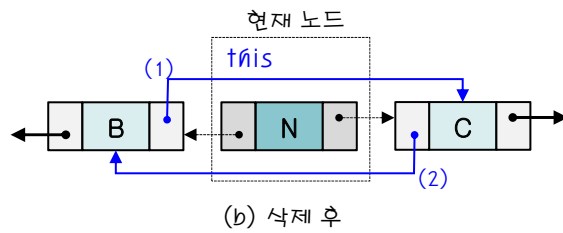
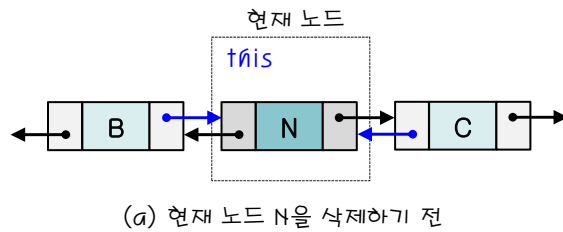


(b) 삽입 후

```
void insert_next(Node *p, Node *n)
{
    if (n != NULL) {
        n->prev = p;
        n->next = p->next;
        if (p->next != NULL)
            p->next->prev = n;
        p->next = n;
    }
}
```

36

삭제연산



```
void remove_curr(Node *n)
{
    if (n->prev != NULL)
        n->prev->next = n->next;
    if (n->next != NULL)
        n->next->prev = n->prev;
}
```

37

전체 프로그램

```
void main()
{
    init_list();
    insert(0, 10);
    insert(0, 20);
    insert(1, 30);
    insert(size(), 40);
    insert(2, 50);
    print_list("이중연결리스트로 구현한 List(삽입x5)");

    replace(2, 90);
    print_list("이중연결리스트로 구현한 List(교체x1)");

    delete(2);
    delete(size() - 1);
    delete(0);
    print_list("이중연결리스트로 구현한 List(삽입x3)");

    clear_list();
    print_list("이중연결리스트로 구현한 List(정리후)");
}
```

C:\WINDOWS\system32\cmd.exe

```
이중연결리스트로 구현한 List(삽입x5) [ 5]: 20 30 50 10 40
이중연결리스트로 구현한 List(교체x1) [ 5]: 20 30 90 10 40
이중연결리스트로 구현한 List(삽입x3) [ 2]: 30 10
이중연결리스트로 구현한 List(정리후) [ 0]:
```

5번의 insert() 연산 결과
교체 연산 결과 (2번 항목)
3번 delete () 연산 결과

38



- 이중 연결 리스트에서 헤드 노드를 사용하는 이유는 무엇인가?



- 다음은 연결 리스트에서 있을 수 있는 여러 가지 경우를 설명했는데 잘못된 항목은?
 1. 정적인 데이터보다는 다양하고 변화있는 데이터에서 효과적인 방법이다.
 2. 모든 노드는 데이터와 링크(포인터)를 가지고 있어야 한다.
 3. 연결 리스트에서 사용한 기억장소는 다시 사용할 수 없다.
 4. 데이터들이 메모리상에 흩어져서 존재할 수 있다.



- 삽입과 삭제 작업이 자주 발생할 때 실행 시간이 가장 많이 소요되는 자료구조는?
 1. 배열로 구현된 리스트
 2. 단순 연결 리스트
 3. 원형 연결 리스트
 4. 이중 연결 리스트



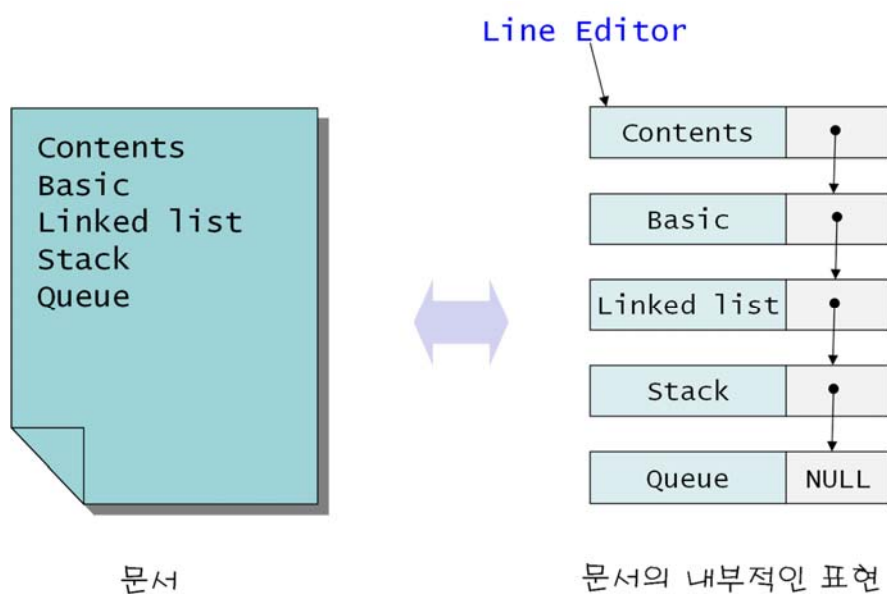
- 다음 중 NULL 포인터(NULL pointer)가 존재하지 않는 구조는 어느 것인가?
 1. 단순 연결 리스트
 2. 원형 연결 리스트
 3. 이중 연결 리스트
 4. 헤더 노드를 가지는 단순 연결 리스트



- 리스트의 n 번째 요소를 가장 빠르게 찾을 수 있는 구현 방법은 무엇인가?
 1. 배열
 2. 단순 연결 리스트
 3. 원형 연결 리스트
 4. 이중 연결 리스트

43

6.5 연결 리스트의 응용 : 라인 편집기



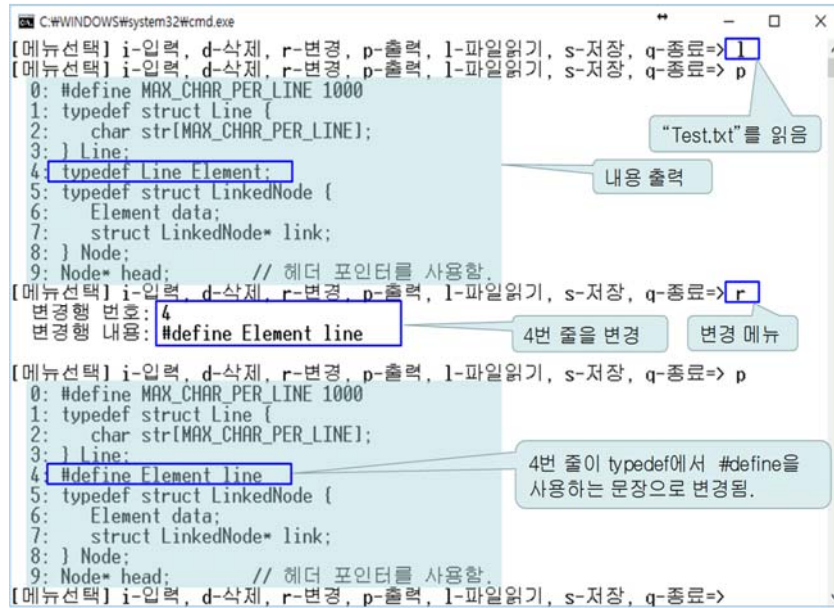
44

- ```
typedef struct Line {
 char str[MAX_CHAR_PER_LINE];
} Line;

typedef Line Element;
```



## 실행 결과 2



```
C:\WINDOWS\system32\cmd.exe
[메뉴선택] i-입력, d-삭제, r-변경, p-출력, l-파일읽기, s-저장, q-종료=> r
[메뉴선택] i-입력, d-삭제, r-변경, p-출력, l-파일읽기, s-저장, q-종료=> p
0: #define MAX_CHAR_PER_LINE 1000
1: typedef struct Line {
2: char str[MAX_CHAR_PER_LINE];
3: } Line;
4: typedef Line Element;
5: typedef struct ListNode {
6: Element data;
7: struct ListNode* link;
8: } Node;
9: Node* head; // 헤더 포인터를 사용함.
[메뉴선택] i-입력, d-삭제, r-변경, p-출력, l-파일읽기, s-저장, q-종료=> r
변경행 번호: 4
변경행 내용: #define Element line
[메뉴선택] i-입력, d-삭제, r-변경, p-출력, l-파일읽기, s-저장, q-종료=> p
0: #define MAX_CHAR_PER_LINE 1000
1: typedef struct Line {
2: char str[MAX_CHAR_PER_LINE];
3: } Line;
4: #define Element line
5: typedef struct ListNode {
6: Element data;
7: struct ListNode* link;
8: } Node;
9: Node* head; // 헤더 포인터를 사용함.
[메뉴선택] i-입력, d-삭제, r-변경, p-출력, l-파일읽기, s-저장, q-종료=>
```

“Test.txt”를 읽음

내용 출력

4번 줄을 변경

변경 메뉴

4번 줄이 typedef에서 #define을 사용하는 문장으로 변경됨.