YONSEI UNIVERSITY

Linked Lists II

Data Structure

Contents

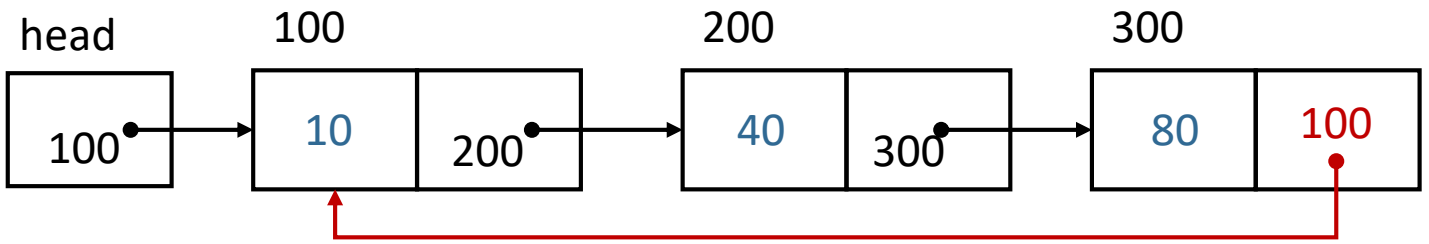
List ADT

- Data : n개의 element로 구성된 순서 있는 모임 (linkedList)
- Operations :
  - linkedList* initList() : 공백 리스트 생성
  - void insertFirst(linkedList* L, element x) : 리스트의 첫 번째 노드로 삽입
  - void insertLast(linkedList* L, element x) : 리스트의 마지막 노드로 삽입
  - void insert(linkedList* L, listNode* pre, element x) : 리스트 L 중간에 x(data) 노드 삽입 (pre: 삽입할 위치의 앞 노드)
  - int delete(linkedList* L, listNode* p) : 리스트에서 p 노드 삭제
  - listNode* search(linkedList* L, element x) : data로 x가 저장되어 있는 노드를 검색
  - int getLength(linkedList* L) : 리스트의 길이(항목의 개수)를 구함
  - void displayList(linkedList* L) : 리스트의 모든 요소를 표시
  - void clear(linkedList* L) : 리스트의 모든 요소 삭제

Circular Linked Lists

Circular Linked List Concept

- 단순 연결 리스트에서 **마지막 노드가 리스트의 첫 번째 노드를 가리키게** 하여 리스트의 구조를 **원형**으로 만든 연결 리스트
- 링크를 따라 계속 순회하면 이전 노드에 접근 가능

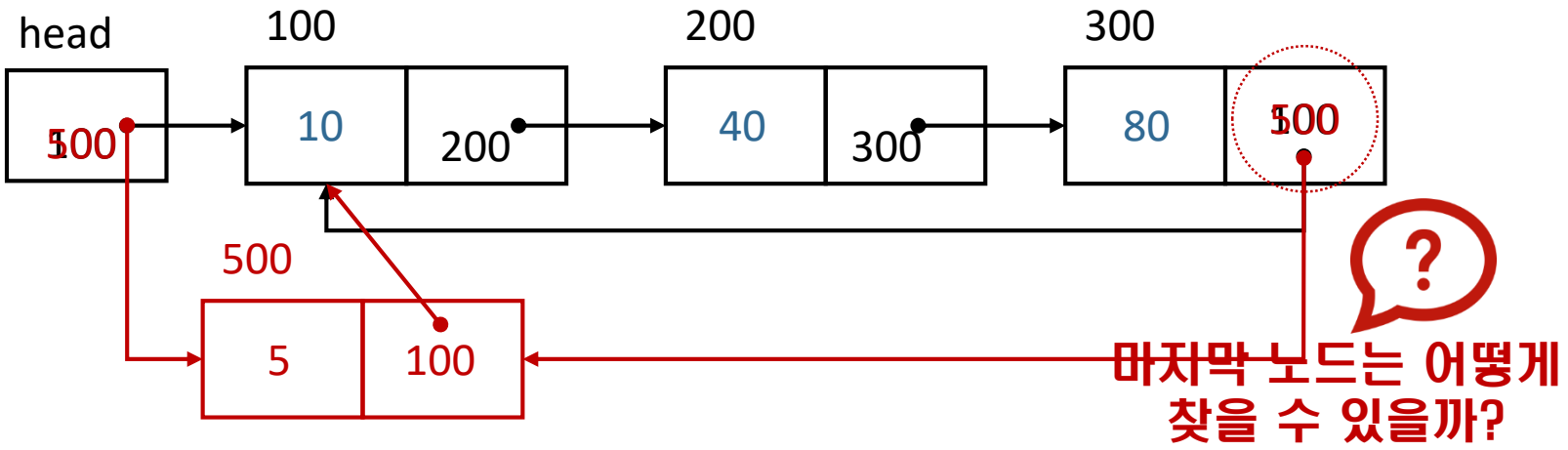


Circular Linked Lists
Insertion I : 첫 번째 노드로 삽입

- 마지막 노드의 링크를 첫 번째 노드로 연결하는 부분만 제외하면 단순 연결리스트에서의 삽입 연산과 같은 연산

- 첫 번째 노드로 5 노드 삽입

```
void insertFirst(linkedList* L, element x)
```



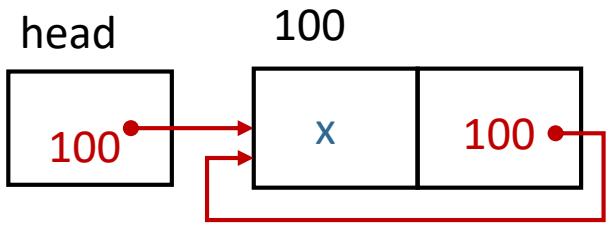
① 삽입할 노드 준비하고 데이터 필드에 값을 저장
② 새 노드의 링크 값 지정 (마지막 노드가 가리키고 있는 노드 주소값)
③ 마지막 노드가 새 노드를 가리키도록 함
④ head에 새 노드를 연결

Circular Linked Lists
Insertion I : 첫 번째 노드로 삽입

Algorithm | **void insertFirst(linkedList* L, element x)**

```
insertFirst(L, x)
    newNode.data ← x
    if (L = NULL) then
        L ← newNode
        newNode.link ← newNode
    else
        temp ← L
        while (temp.link ≠ L) do
            temp ← temp.link
        newNode.link ← temp.link
        temp.link ← newNode
        L ← newNode
    length++
end insertFirst()
```

공백리스트에 삽입



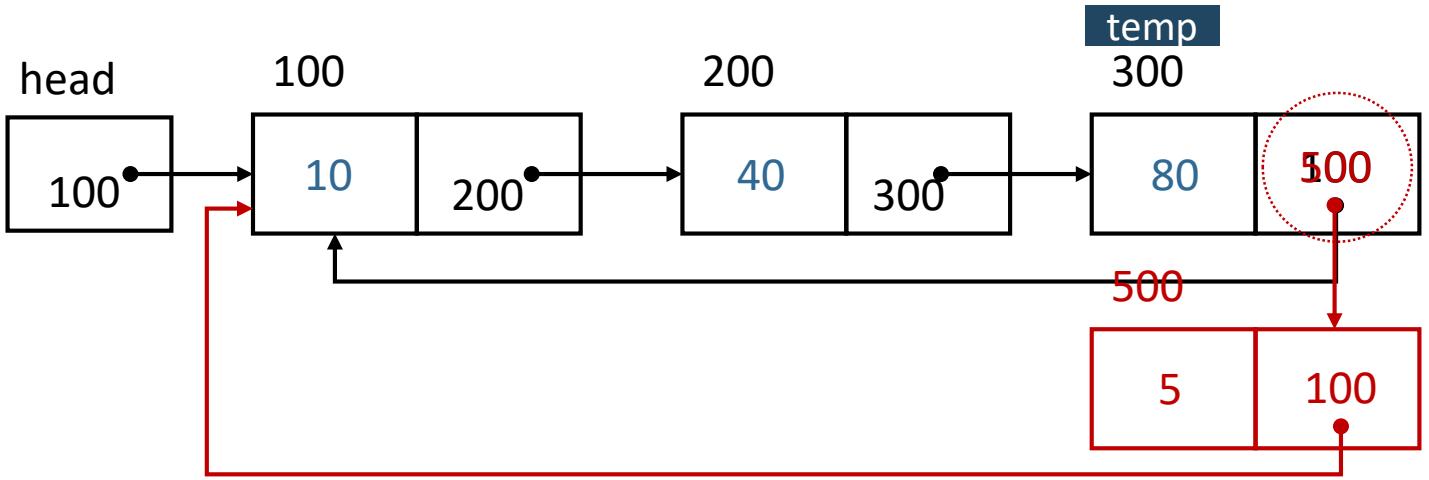
Q1. 첫 번째 노드로 삽입하는 경우 시간 복잡도는?

Q2. 시간 복잡도를 O(1)으로 만드는 방법은?

Circular Linked Lists
Insertion II : 마지막 노드로 삽입

void insertLast(linkedList* L, element x)

- 마지막 노드로 90 노드 삽입



① 삽입할 노드 준비하고 데이터 필드에 값을 저장
② 마지막 노드 검색
③ 마지막 노드가 가리키고 있던 첫 번째 노드를 새 노드가 가리키도록 함
④ 마지막 노드가 새 노드를 가리키도록 함

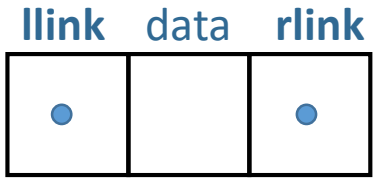
나머지 ADT

- 중간 노드로 삽입 연산 : 단순 연결 리스트와 동일
- Deletion
  - 첫 번째 노드, 마지막 노드가 삭제되는 경우 : 원형 연결 리스트의 특징 고려
  - 중간 노드가 삭제되는 경우 : 단순 연결 리스트와 동일
- 원형 연결 리스트의 ADT 연산 알고리즘은 스스로 생각해보자!

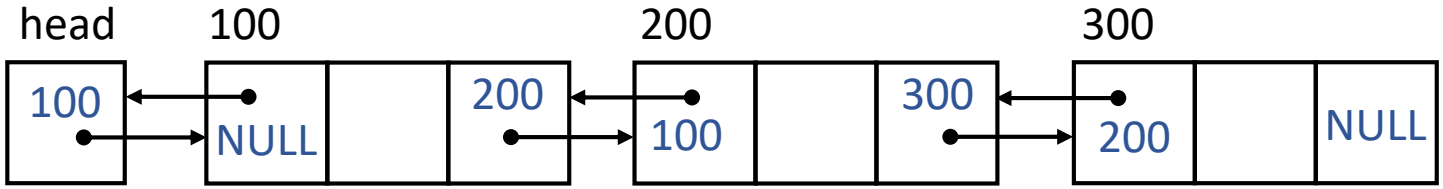
Doubly Linked Lists Concepts

Doubly Linked Lists Concept

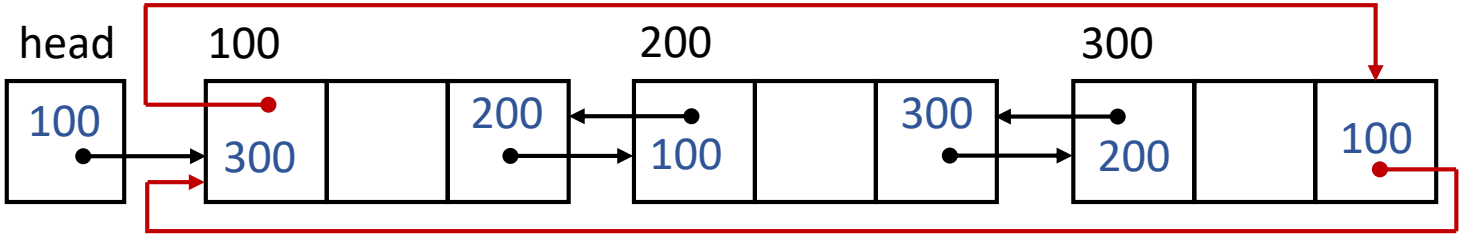
- **양쪽 방향으로 순회**할 수 있도록 노드를 연결한 리스트
- 노드 구조



- (단순) 이중 연결 리스트



- 원형 이중 연결 리스트

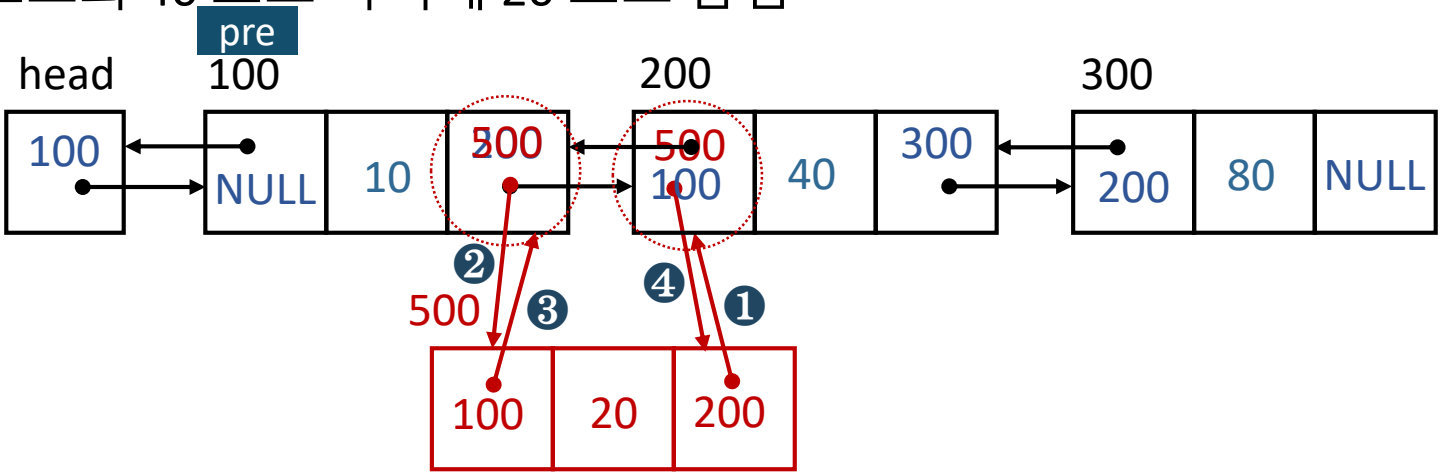


Doubly Linked Lists
- Insertion & Deletion

Doubly Linked Lists
Insertion I : 중간 노드로 삽입

void insert(linkedList* L, listNode* pre, element x)

- 10 노드와 40 노드 사이에 20 노드 삽입



① 삽입할 노드 준비

② 새 노드의 데이터 필드에 값을 저장

③ 새 노드 왼쪽 노드의 rlink 필드에 있던 값을 새 노드의 rlink 필드에 저장

④ 왼쪽 노드의 rlink 필드에 새 노드의 주소 저장

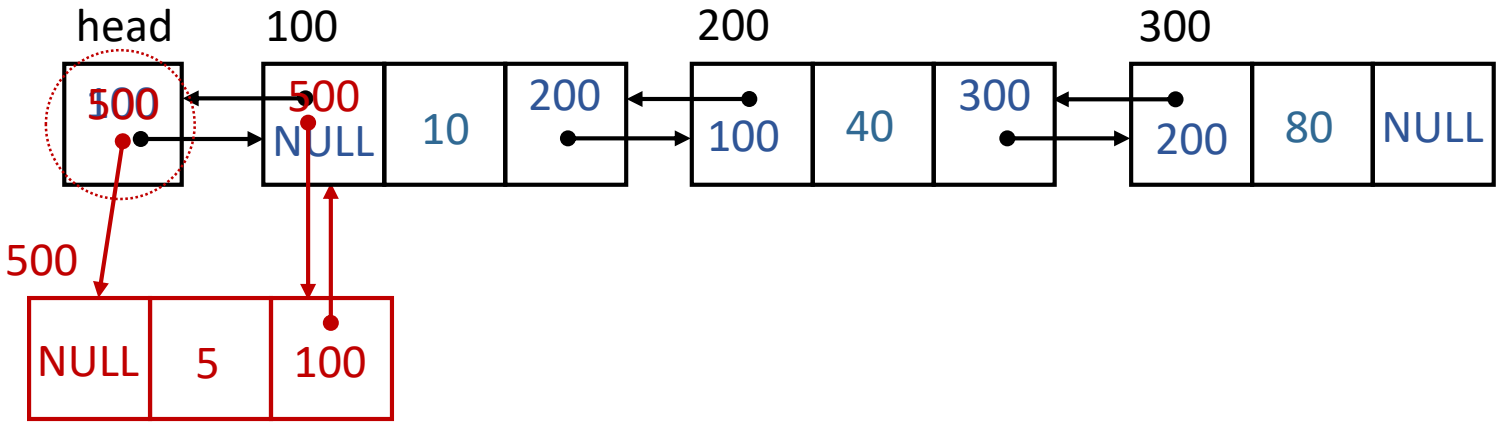
⑤ 새 노드의 llink 필드에 왼쪽 노드의 주소 저장

⑥ 오른쪽 노드의 llink 필드에 새 노드의 주소 저장

Doubly Linked Lists
Insertion II : 첫 번째 노드로 삽입

void insertFirst(linkedList* L, element x)

- 첫 번째 노드로 5 노드 삽입



① 삽입할 노드 준비

② 새 노드의 데이터 필드에 값을 저장

③ 새 노드의 왼쪽 노드가 없으므로 새 노드의 llink 필드에 NULL 저장

④ head가 가리키고 있던 첫 번째 노드의 주소값을 새 노드의 rlink 필드에 저장

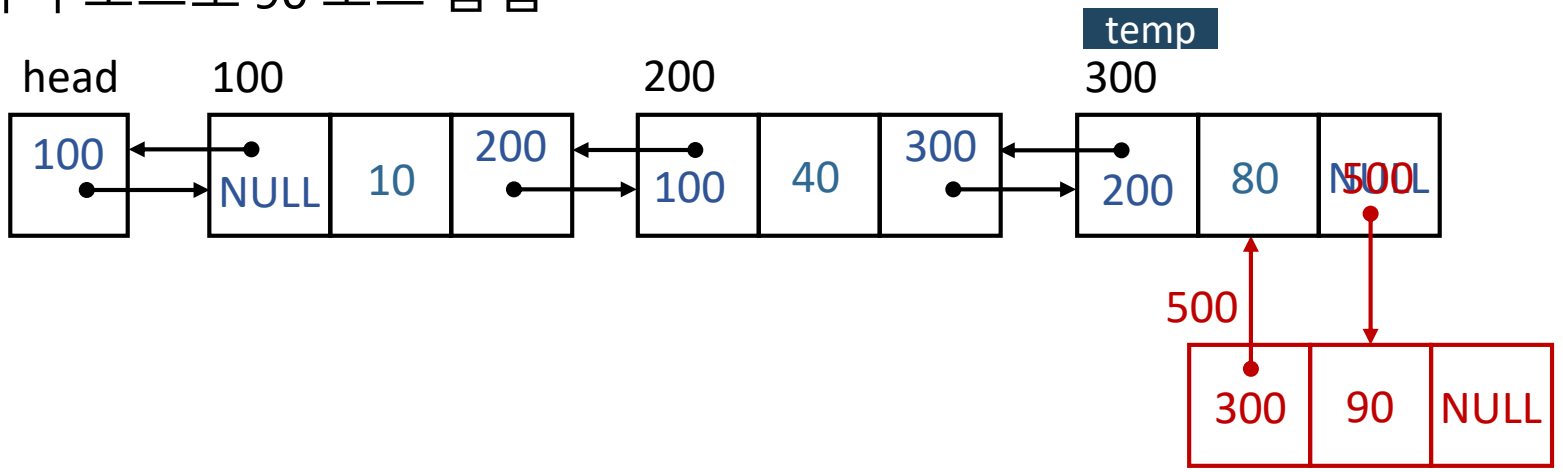
⑤ 첫 번째 노드였던 노드의 llink 필드에 새 노드의 주소값 저장

⑥ head에 새 노드 연결

Doubly Linked Lists
Insertion III : 마지막 노드로 삽입

void insertLast(linkedList* L, element x)

- 마지막 노드로 90 노드 삽입



① 삽입할 노드 준비

② 새 노드의 데이터 필드에 값을 저장

③ 새 노드의 오른쪽 노드가 없으므로 새 노드의 rlink 필드에 NULL 저장

④ 마지막 노드를 검색

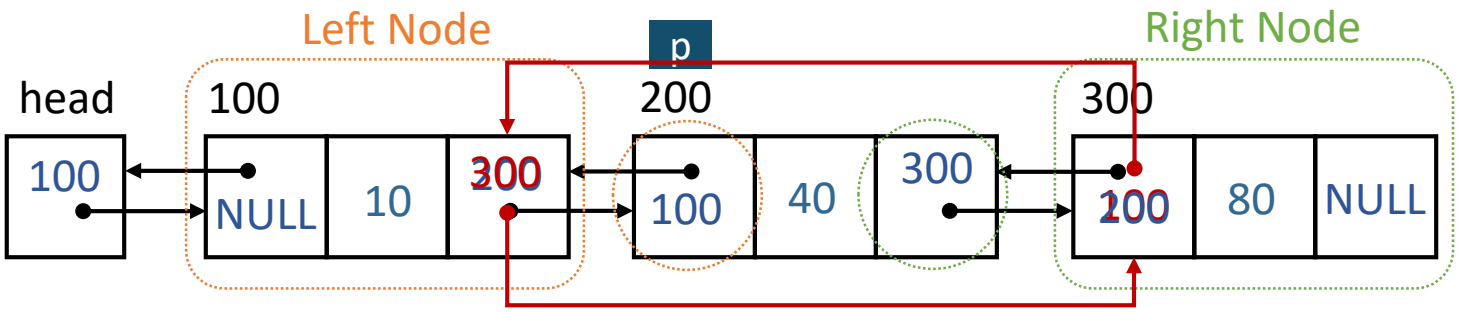
⑤ 새 노드의 llink 필드에 마지막 노드의 주소값 저장

⑥ 마지막 노드의 rlink 필드에 새 노드의 주소값 저장

Doubly Linked Lists
Deletion

int delete(linkedList* L, listNode* p)

- 40 노드 삭제



① 삭제할 노드의 오른쪽 노드와 왼쪽 노드 탐색

② 삭제할 노드의 오른쪽 노드의 주소를 삭제할 노드의 왼쪽 노드의 rlink에 저장

③ 삭제할 노드의 왼쪽 노드의 주소를 삭제할 노드의 오른쪽 노드의 llink에 저장

④ 삭제할 노드를 삭제

Doubly Linked Lists
- Implementation

리스트 노드 구조체 정의

- 노드 구조체
  - 항목들의 타입은 element로 정의
  - Member : element data, struct ListNode *llink, *rlink

```
typedef int element;
typedef struct ListNode {
    element data;
    struct ListNode *llink, *rlink;
}listNode;
```

리스트 노드의 Data field

왼쪽 노드와 연결하는 llink,
오른쪽 노드와 연결하는 rlink

- 리스트의 시작을 나타내는 head를 구조체로 정의

```
typedef struct LinkedList {
    listNode* head;
    int length;
}linkedList;
```

리스트의 첫 번째 항목을 가리키는 변수

리스트 항목 개수

Doubly Linked Lists
initList(), getLength() 연산

- linkedList* initList() 연산
  - 공백 리스트 생성

```
linkedList* initList{
    linkedList* L;
    L = (linkedList*)malloc(sizeof(linkedList));
    L->head = NULL;
    L->length = 0;
    return L;
}
```

- int getLength() 연산

```
int getLength(linkedList* L) {
    return L->length;
}
```

Doubly Linked Lists
Insertion I Algorithm

Algorithm | **void insert(linkedList* L, listNode* pre, element x)**

```
insert(L, pre, x)
   newNode.data ← x
   if (L = NULL) then
      newNode.rlink ← NULL
      newNode.llink ← NULL
      L ← newNode
   else if (pre = NULL) then
      newNode.llink ← NULL
      newNode.rlink ← L
      L ← newNode
   else
      newNode.rlink ← pre.rlink
      pre.rlink ← newNode
      newNode.llink ← pre
      if (newNode.rlink ≠ NULL) then
         newNode.rlink.llink ← newNode
   endif
length++
end insert()
```

공백리스트에 삽입

리스트의
맨 처음 노드로 삽입

리스트의
중간에 삽입

삽입 자리에
다음 노드가 있는 경우

Doubly Linked Lists
Deletion Algorithm

Algorithm | **int delete(linkedList* L, listNode* p)**

```
delete(L, p)
  if (L = NULL) then return FALSE
  if (p = NULL) then return FALSE
  if (p.llink = NULL) then
     L ← p.rlink
  else
     p.llink.rlink ← p.rlink
  endif
  if (p.rlink ≠ NULL) then
     p.rlink.llink ← p.llink
  endif
  delete p
  length—
  return TRUE
end delete()
```

공백리스트인 경우

삭제할 노드가 없는 경우

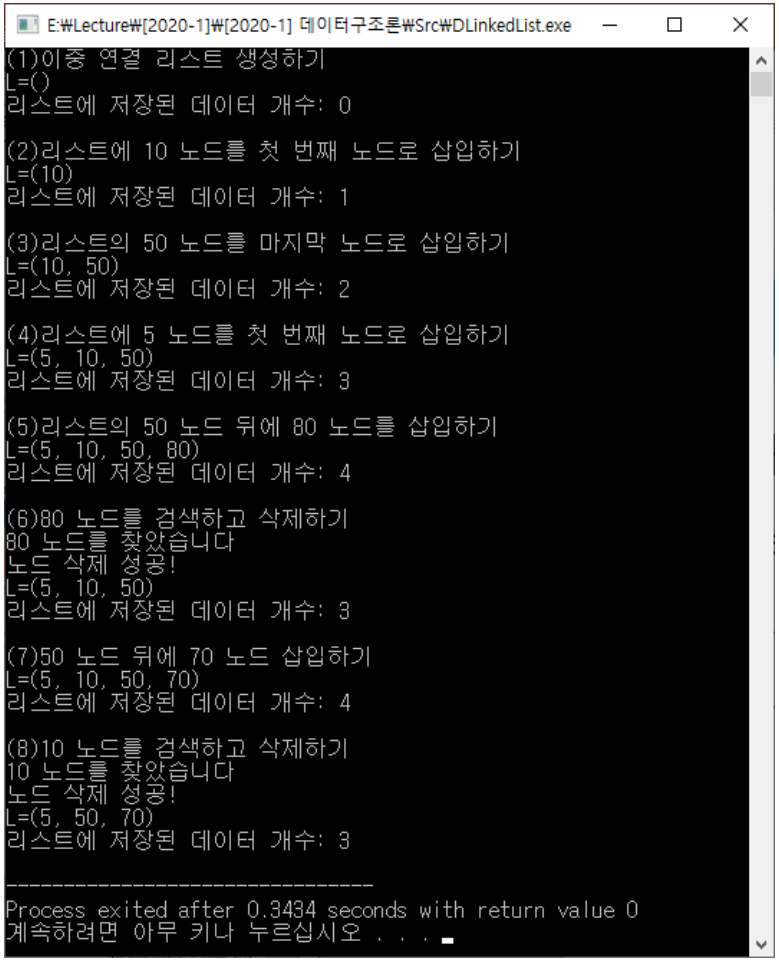
첫 번째 노드가 삭제되는 경우

삭제할 노드가
마지막 노드가 아닌 경우

Doubly Linked Lists

HW#4.1. Doubly Linked List 구현

- 아래와 같이 실행되도록 main()함수 구성
- Linked List ADT의 모든 연산 구현
- DLinkedList.h 및 DLinkedListMain.c제공
- DLinkedList.c 완성하여 제출



Linked Lists
- Ex: Polynomial

Doubly Linked Lists
Polynomial (다항식)

- 다항식의 개념
  - ax^e 형식의 항들의 합으로 구성된 식
    - ✓ a : 계수 (Coefficient)
    - ✓ x : 변수 (Variable)
    - ✓ e : 지수 (Exponent)
- 다항식의 특징
  - 지수에 따라 내림차순으로 항을 나열
  - 다항식의 차수 : 가장 큰 지수
  - 다항식 항의 최대 개수 = (차수 + 1) 개

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0 x^0$$

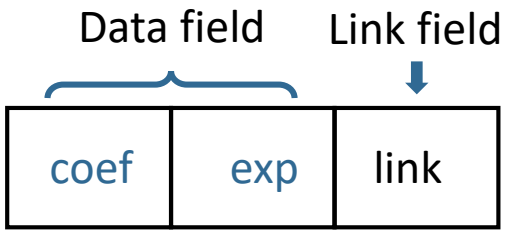
n차 다항식 $p(x)$

Doubly Linked Lists
Polynomial ADT

- Data : 지수(e_i), 계수(a_i)의 순서쌍 <e_i, a_i>의 집합으로 표현된 다항식
- Operations : A, B는 다항식, coef는 계수, exp는 지수
  - polyList* initList() : 공백 다항식 리스트 생성
  - void appendTerm(polyList* PL, float coef, int exp) : 다항식 PL에 계수가 coef, 지수가 exp인 노드 항 삽입
  - polyList* addPoly(polyList* A, polyList* B) : 다항식 A와 B의 합을 구함
  - polyList* mulPoly(polyList* A, polyList* B) : 다항식 A와 B의 곱을 구함
  - void displayPoly(polyList* PL) : 다항식 리스트 출력

Doubly Linked Lists
Polynomial의 Linked List 표현

- 다항식의 각 항을 표현하기 위한 노드 구조




```
typedef struct PolyNode {
    float coef;
    int exp;
    struct Node* link;
}Node;
```

다항식 항의 계수
다항식 항의 지수
다음 항의 노드

- 다항식 구조체 정의

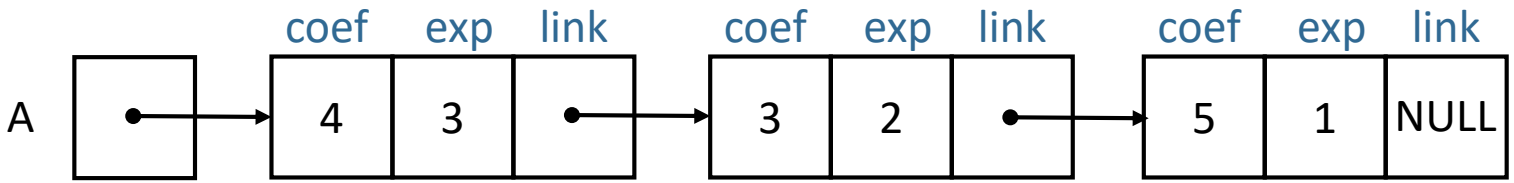
```
typedef struct PolyList {
    Node* head;
    Node* last;
}polyList;
```

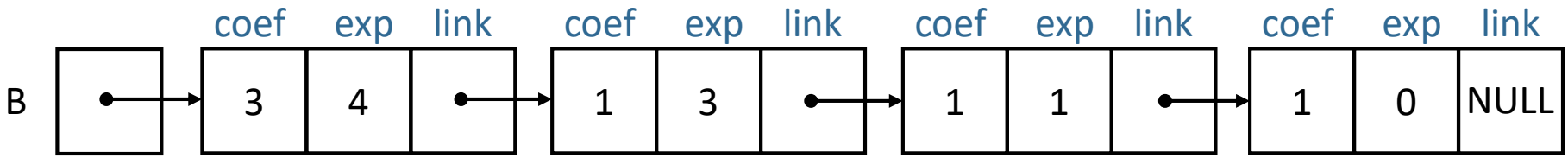
다항식의 첫 번째 항
다항식 마지막 항

Doubly Linked Lists
Polynomial의 Linked List 표현

$$A(x) = 4x^3 + 3x^2 + 5x$$

$$B(x) = 3x^4 + x^3 + 2x + 1$$





Doubly Linked Lists
Polynomial: 항 삽입

- 다항식 리스트의 마지막 요소 뒤에 항 삽입
  - 리스트의 마지막 노드로 삽입하는 알고리즘과 유사

Algorithm	void appendTerm(polyList* PL, float coef, int exp)

```
appendTerm(PL, coef, exp)
    newNode.coef ← coef
    newNode.exp ← exp
    newNode.link ← NULL
    if (PL = NULL) then
        PL.head ← newNode
    else
        PL.last.link ← newNode
    endif
    PL.last ← newNode
end appendTerm()
```

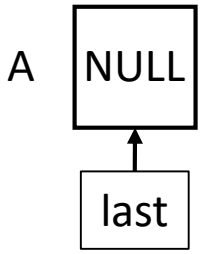
Doubly Linked Lists
Polynomial: 항 삽입

$$A(x) = 4x^3 + 3x^2 + 5x$$

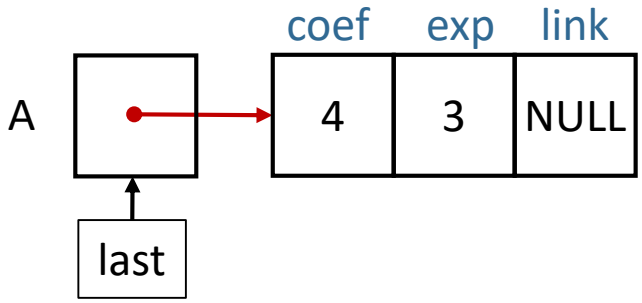
Algorithm	void appendTerm(polyList* PL, float coef, int exp)

```
appendTerm(PL, coef, exp)
    newNode.coef ← coef
    newNode.exp ← exp
    newNode.link ← NULL
    if (PL = NULL) then
        PL.head ← newNode
    else
        PL.last.link ← newNode
    endif
    PL.last ← newNode
end appendTerm()
```

1) 공백 다항식 리스트 A 생성



2) $4x^3$ 추가



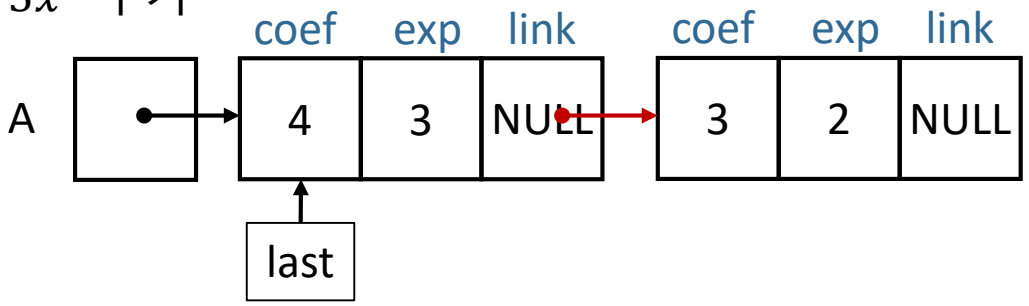
Doubly Linked Lists
Polynomial: 항 삽입

$$A(x) = 4x^3 + 3x^2 + 5x$$

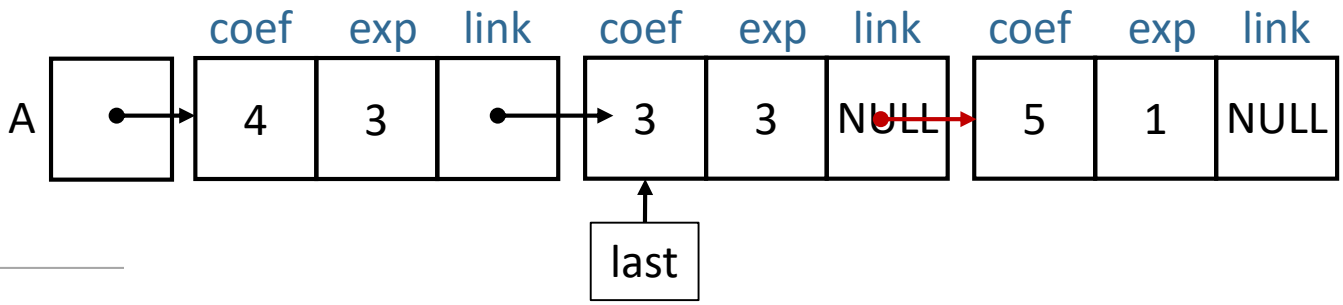
Algorithm | **void appendTerm(polyList* PL, fl**

```
appendTerm(PL, coef, exp)
    newNode.coef ← coef
    newNode.exp ← exp
    newNode.link ← NULL
    if (PL = NULL) then
        PL.head ← newNode
    else
        PL.last.link ← newNode
    endif
    PL.last ← newNode
end appendTerm()
```

3) $3x^2$ 추가



4) $5x$ 추가



Doubly Linked Lists
Polynomial: 다항식 덧셈

Algorithm	polyList* addPoly(polyList* A, polyList* B)

```
addPoly(A, B)
   p ← A, q ← B
   C ← NULL
   while (p ≠ NULL and q ≠ NULL) do
      if (p.exp = q.exp) then
         sum ← p.coef + q.coef
         if (sum ≠ 0) then appendTerm(C, sum, p.exp)
         p ← p.link, q ← q.link
      else if (p.exp > q.exp) then
         appendTerm(C, p.coef, p.exp)
         p ← p.link
      else
         appendTerm(C, q.coef, q.exp)
         q ← q.link
      endif
   endwhile
   while (p ≠ NULL) do
      appendTerm(C, p.coef, p.exp)
      p ← p.link
   endwhile

   while (q ≠ NULL) do
      appendTerm(C, q.coef, q.exp)
      q ← q.link
   endwhile

   return C
end addPoly()
```

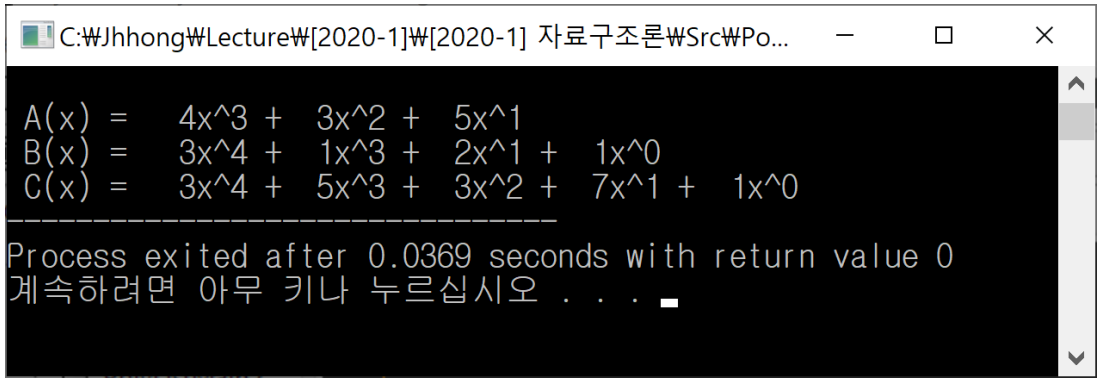
HW#4.2. 다항식의 덧셈

- 다항식 A, B가 아래와 같이 주어졌을 때, addPoly() 연산 구현

$$A(x) = 4x^3 + 3x^2 + 5x$$

$$B(x) = 3x^4 + x^3 + 2x + 1$$

- PolyList.h 및 PolyListMain.c제공, PolyList.c 완성



Linked Lists
Linear List vs. Linked List

	Linear Lists	Linked Lists
Pros	구현 간단	삽입, 삭제 효율적 크기 제한 없음
Cons	삽입, 삭제 시 오버헤드 항목의 개수 제한	구현 복잡