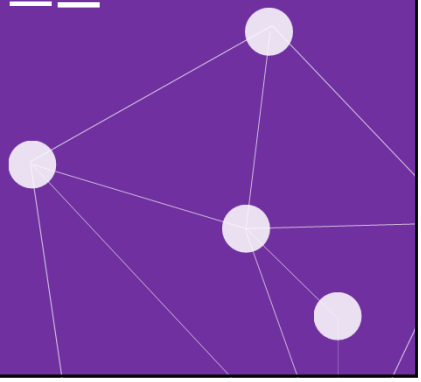
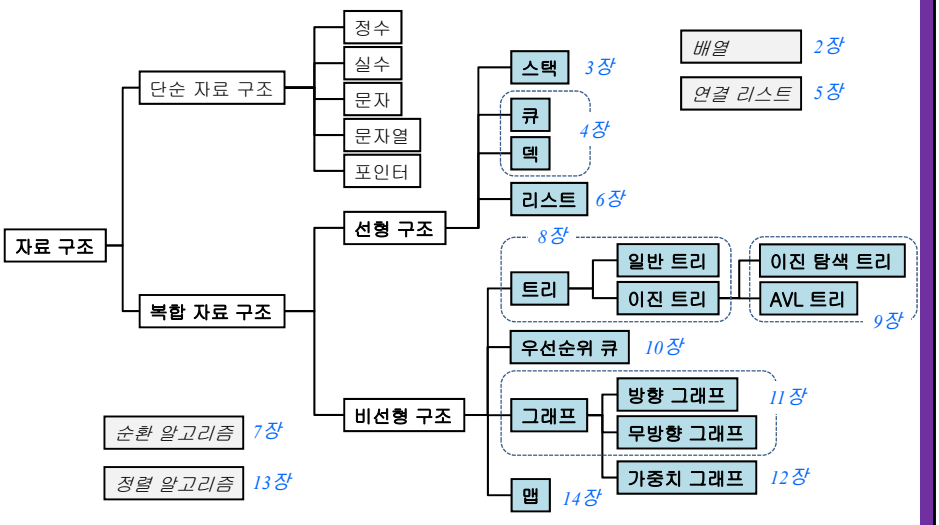
DATA
STRUCTURES
USING C

CHAPTER 06

리스트

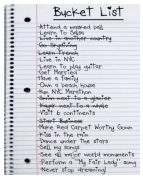


자료구조의 분류



리스트란?

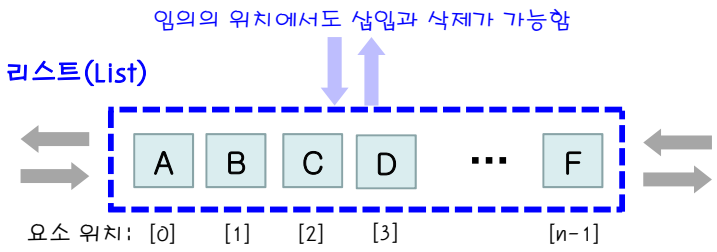
- 리스트(list), 선형리스트(linear list)
  - 순서를 가진 항목들의 모임
  - 집합: 항목간의 순서의 개념이 없음



- 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 위치



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 차있는지를 체크한다.
- 고급 연산
  - 리스트에 어떤 요소가 있는지를 살핀다.
  - 리스트의 어떤 위치에 있는 요소를 새로운 요소로 대치한다.
  - 리스트 안의 요소의 개수를 센다.
  - 리스트 안의 모든 요소를 출력한다.

5

리스트 ADT

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

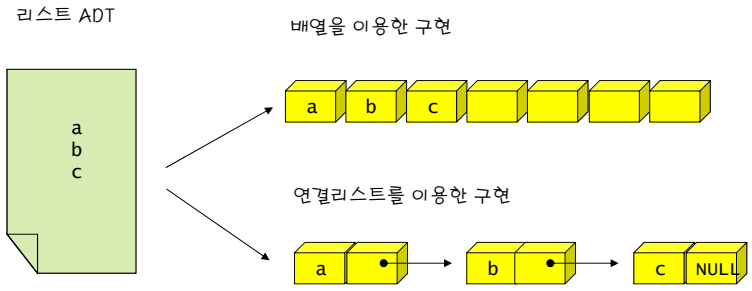
연산:

- `init()`: 리스트를 초기화한다.
- `insert(pos, item)`: pos 위치에 새로운 요소 item을 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 item이 있는지를 살핀다.
- `replace(pos, item)`: pos 위치를 새로운 요소 item으로 바꾼다.
- `size()`: 리스트안의 요소의 개수를 반환한다.

6

리스트 구현 방법

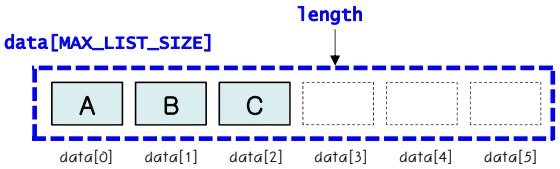
- 배열을 이용
  - 구현이 간단
  - 삽입, 삭제시 오버헤드
  - 항목의 개수 제한
- 연결리스트를 이용
  - 구현이 복잡
  - 삽입, 삭제가 효율적
  - 크기가 제한되지 않음



7

배열로 구현된 리스트

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

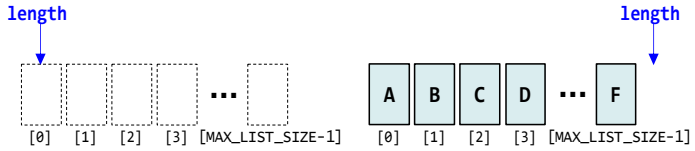



```
typedef int Element;
Element data[MAX_LIST_SIZE];
int length = 0;
```

8

공백상태 / 포화상태

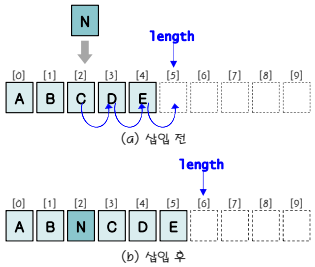
- 공백상태 / 포화상태



(a) 공백 상태 (b) 포화 상태

삽입 연산

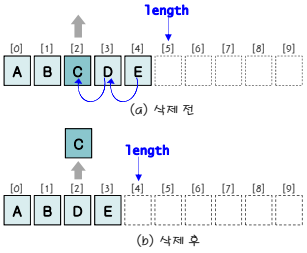
- 삽입위치 다음의 항목들을 이동하여야 함.




```
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");
}
```

삭제 연산

- 삭제위치 다음의 항목들을 이동하여야 함

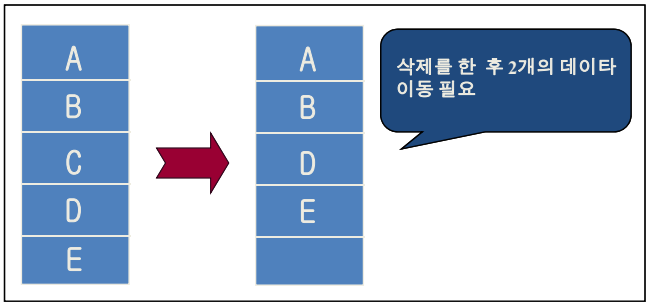



```
void delete( int pos )
{
  int i;

  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");
}
```


선형 리스트의 항목 삭제

■ 항목 삭제 (C 를 삭제 시)

- n개의 항목이 있을 경우 평균 이동횟수 = (n-1) / 2 회



선형 리스트의 삭제시 이동

- 삭제시 element의 평균 이동 횟수

$$m_d = \sum_{k=1}^{n} (n-k) \cdot p_k$$

(단, n은 현재 리스트에 있는 element의 수,
k는 삭제되는 위치,
p_k는 k번째 위치의 element 삭제확률)

$$m_d = \sum_{k=1}^{n} (n-k) \cdot \frac{1}{n} = \frac{1}{n}\left[\sum_{k=1}^{n} n - \sum_{k=1}^{n} k\right]$$

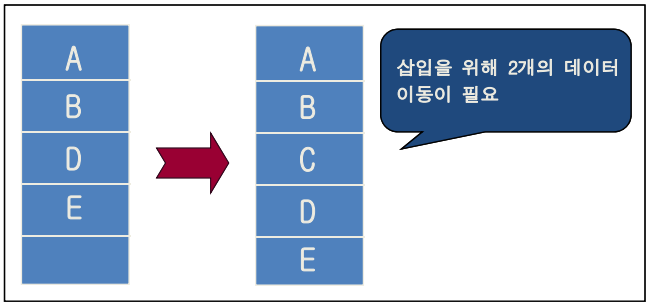
$$= \frac{1}{n}\sum_{k=1}^{n} n - \frac{1}{n}\sum_{k=1}^{n} k = n - \frac{n(n+1)}{2n} = \frac{n-1}{2}$$

13

선형 리스트의 항목 삽입

■ 항목 삽입 (C 삽입 시)

- **n개의 항목이 있을 경우 평균 이동횟수 = (n+1) / 2 회**



14

선형 리스트의 삽입시 이동

- 삽입시 element의 평균 이동 횟수

$$m_i = \sum_{k=1}^{n} (n-k+1) \cdot p_k$$

(단, n은 현재 리스트에 있는 element의 수,
k는 삽입되는 위치,
p_k는 k번째 위치의 element 삭입확률)

$$m_i = \sum_{k=1}^{n} (n-k+1) \cdot \frac{1}{n} = \frac{1}{n}\left[\sum_{k=1}^{n}(n+1) - \sum_{k=1}^{n} k\right]$$

$$= \frac{n(n+1)}{n} - \frac{1}{n} \cdot \frac{n(n+1)}{2} = n+1-\frac{n+1}{2} = \frac{n+1}{2}$$

$n \to \infty$이면 $m_i = \frac{n+1}{2} \approx \frac{n-1}{2} = m_d = \frac{n}{2}$

선형 리스트의 장단점

■ 선형리스트의 장단점

- 장점
  - 가장 간단한 데이터 구조이다.
  - 메모리 밀도(**memory density**) 측면 고려.
    - *메모리 밀도*: 일정 크기의 기억 장소에 얼마나 많은 데이터가 저장될 수 있는지를 나타내는 조밀도

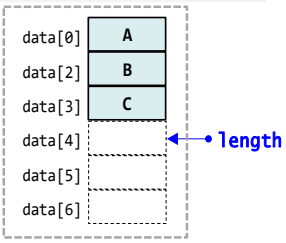
$$\text{메모리 밀도} = \frac{\text{정보들의 총수}}{\text{비트들의 총수}} \text{ 또는 } \frac{\text{논리적 정보 단위의 총수}}{\text{물리적 정보 단위의 총수}}$$

- 단점
  - 항목 삽입 시 연속적인 기억장소를 요구한다
  - 삽입, 삭제 시 선형구조를 유지해야 하므로 많은 데이타의 이동으로 인한 속도의 저하가 발생한다

연결 리스트로 구현한 리스트

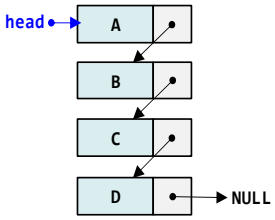
- 단순 연결 리스트(simply linked list) 사용
  - 하나의 링크 필드를 이용하여 연결
  - 마지막 노드의 링크 값은 NULL

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



배열을 이용한 리스트

```
Node* head;
```



연결 리스트를 이용한 리스트

데이터

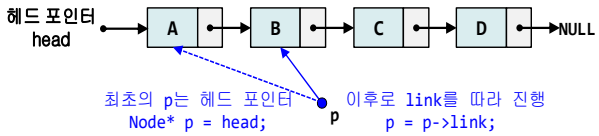
- 구조체

```
#define Element       int
typedef struct LinkedNode {
        Element data;                   // 데이터 필드
        struct LinkedNode* link;        // 링크 필드
} Node;
```

- 데이터

```
Node*   head;   // 헤드 포인터
```

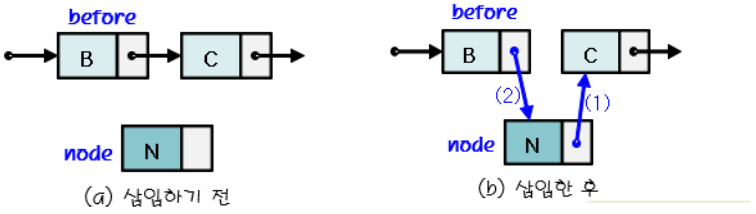
단순한 연산들




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

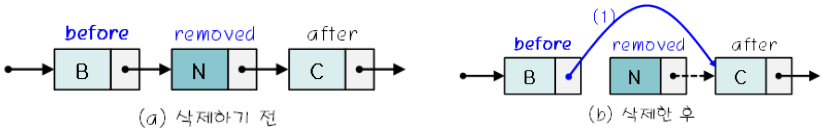
```
Node* get_entry(int pos)
{
  Node* p = head;
  int i;
  for (i = 0; i<pos; i++, p = p->link)
    if (p == NULL) return NULL;
  return p;
}
```


삽입연산




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

삭제연산



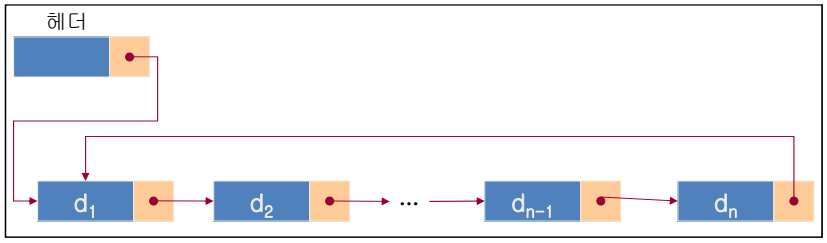

```
Node* remove_next(Node *prev)
{
  Node* removed = prev->link;
  if (removed != NULL)
    prev->link = removed->link;
  return removed;
}
```

21

원형 연결 리스트

- 원형 연결 리스트의 정의
  - 마지막 노드의 포인터를 **null**이 아닌 첫번째 노드의 주소를 가리키도록 구성하는 연결 리스트

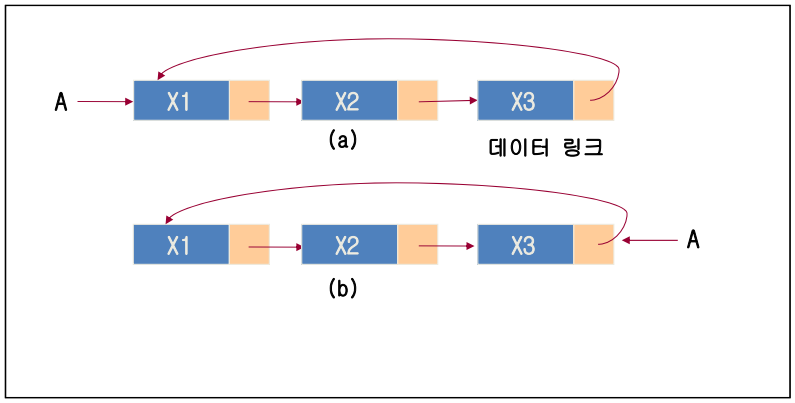
- 원형 연결 리스트의 구조



22

원형 연결 리스트의 기본연산

- 원형 리스트의 삽입 과정



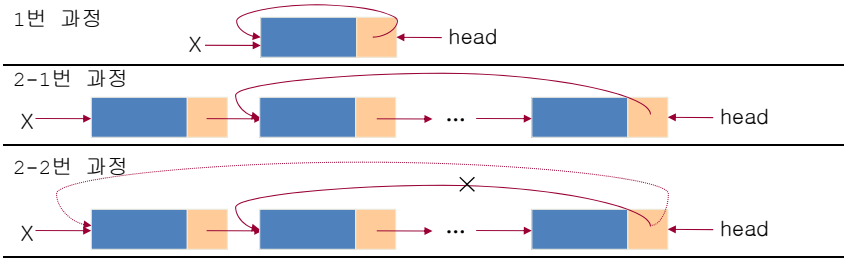
원형 연결 리스트의 기본연산

■ 원형 연결 리스트의 노드 삽입 알고리즘

```
void Insert_Front (struct Node *A, struct Node *X){
        if(A == NULL) {
                A = X;
                X->link = A;
        }
        else {
                X->link = A->link;
                A->link = X;
        }
}
```

원형 연결 리스트의 기본연산

- 원형 연결 리스트의 노드 삽입 과정

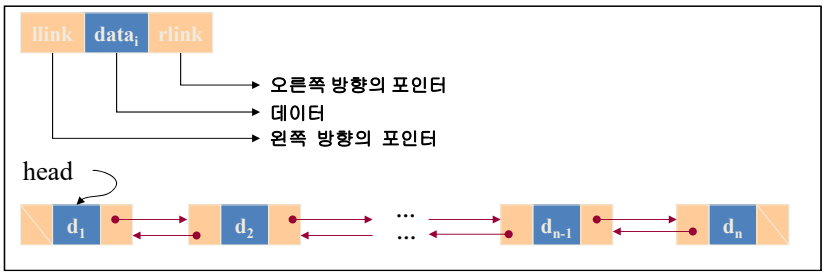



```
void Insert_Front (struct Node *A, struct Node *X){
    if(A == NULL) {
        A = X;
        X->link = A;
    }
else {
        X->link = A->link;
        A->link = X;
        }
}
```

25

이중 연결 리스트[1/6]

- 이중 연결 리스트의 정의
  - 오른쪽과 왼쪽 링크를 두고 양방향으로 탐색이 가능하도록 만든 연결 리스트
- 이중 연결 리스트의 구조



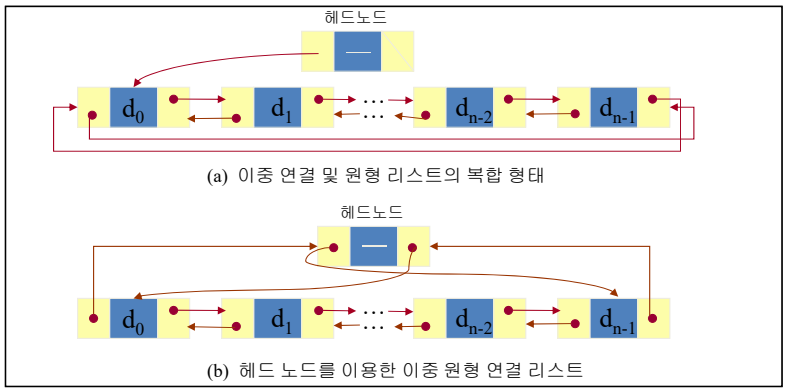
26

이중 연결 리스트 (2/6)

■ 이중 원형 연결 리스트의 정의

- 오른쪽과 왼쪽 링크를 두고 양방향으로 탐색이 가능하도록 만든 원형 연결 리스트

■ 이중 원형 연결 리스트의 구조



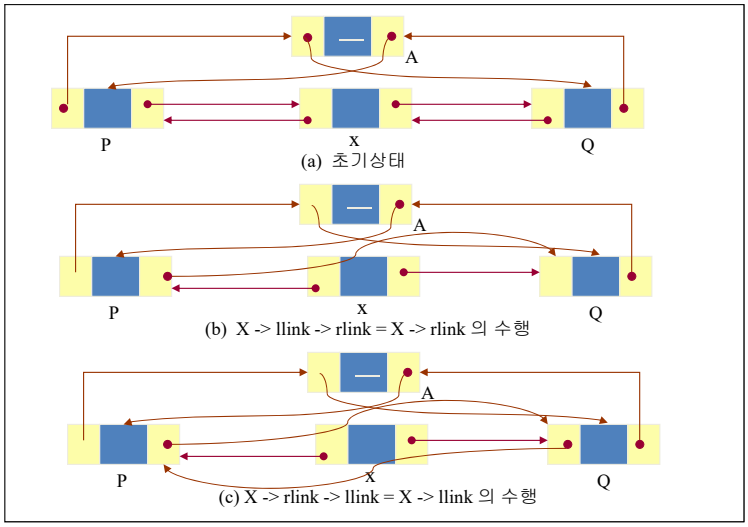
이중 연결 리스트 (3/6)

■ 이중 원형 연결 리스트의 기본 연산(삭제)

```
struct Dnode {
        int data;
        struct Dnode *llink;
        struct Dnode *rlink;
};

void Ddelete(struct Dnode *X, struct Dnode *L){
        if(X = = L)
                NO_MORE_NODES();
        else{
                X->llink->rlink = X->rlink;
                X->rlink->llink = X->llink;
                RET(X);
        }
}
```

이중 연결 리스트 (4/6)

- 이중 원형 연결 리스트의 노드 삭제(계속)

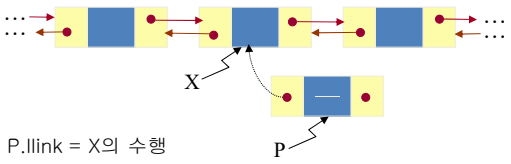


29

이중 연결 리스트 (5/6)

- 이중 원형 연결 리스트의 기본 연산(삽입)

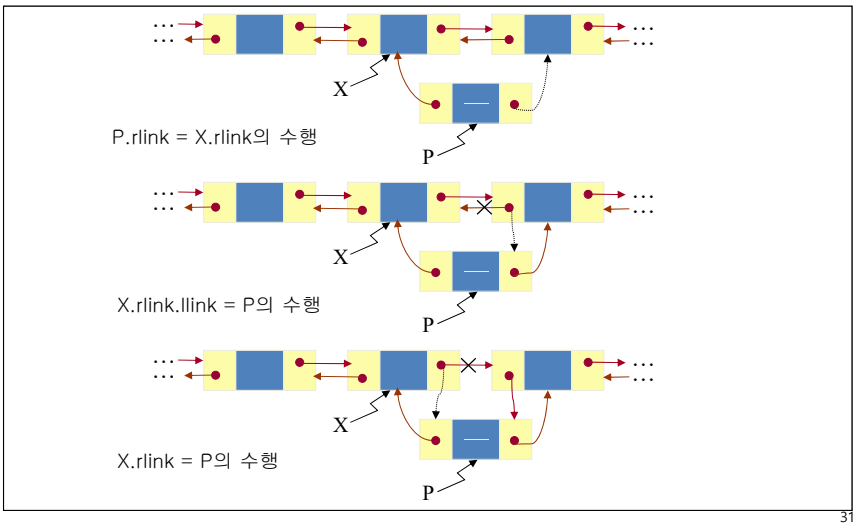
```
void Dinsert(struct Dnode *P, struct Dnode *X) {
        P->llink = X;
        P->rlink = X->rlink;
        X->rlink->llink = P;
        X->rlink = P;
}
```



30

이중 연결 리스트 (6/6)

- 이중 원형 연결 리스트의 노드 삽입 (계속)



31

다중 연결 리스트 (1/2)

- 정의
  - 각 구성 노드가 여러 개의 포인터를 갖고 있는 연결 리스트의 확장형
  - 필요한 만큼 포인터 필드를 두어 필요한 목적에 이용
- 장점
  - 찾고자 하는 레코드를 다른 연결 리스트보다 쉽게 찾을 수 있다.

32

다중 연결 리스트 (2/2)

■ 다중 연결 리스트의 예

- 이름과 계좌 번호를 키로 할 때도 쉽게 찾을 수 있도록 레코드를 구성할 수 있다

번호	이 름	이름 포인터	계좌 번호	계좌번호 포인터
1	정광식	4	1234	4
2	박찬열	1	5768	5
3	공기식	5	5678	2
4	최광희	–	3456	3
5	길준민	2	7788	–

- 이름 헤더 포인터: **3**
- 계좌 번호 헤더 포인터: **1**

연결리스트의 응용–다항식의 덧셈(1/4)

■ 다항식의 연결 리스트의 표현

- 다항식의 형태

$A(x) = a_m x^{em} + \ldots + a_1 x^{e1}$

(a_i는 계수 $[1 \le i \le n]$, e_i는 지수 $[0 \le e_1 < e_2 < \ldots < e_{m-1} < e_m]$로 가정)

- 다항식에 사용되는 노드의 형태

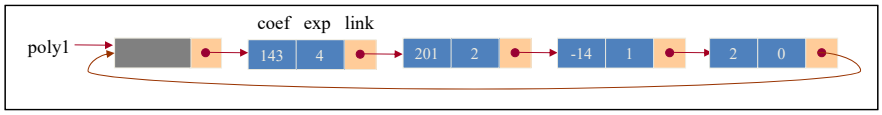
- 노드 구조를 위한 선언

```
struct Pnode {
        int coef;
        int exp;
        struct Pnode *link;
};
```

연결리스트의 응용–다항식의 덧셈(2/4)

- 다항식의 연결 리스트 형태

  예) poly1 = $143x^4+201x^2-14x+2$



- 다항식 두개의 덧셈

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

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

$C = A + B$

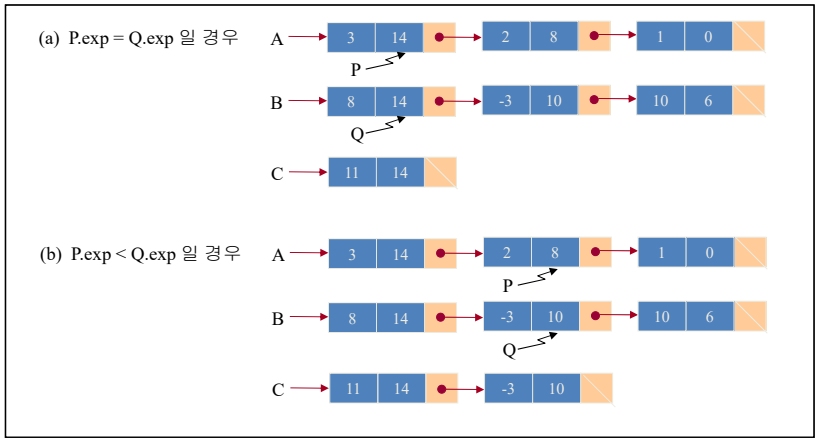
$A = 3x^{14} + 2x^8 + 1$

$+)\ B = 8x^{14} - 3x^{10} + 10x^6$

$C = 11x^{14} - 3x^{10} + 2x^8 + 10x^6 + 1$

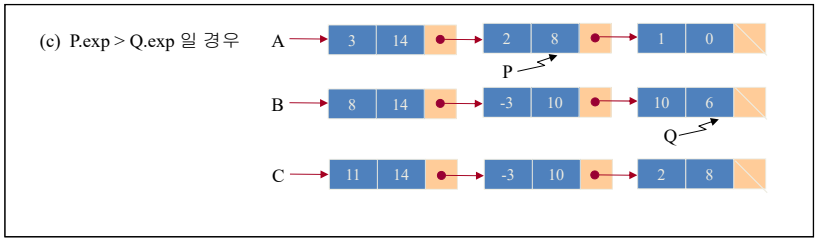
연결리스트의 응용–다항식의 덧셈(3/4)

- C = A + B 의 계산 과정
  - P와 Q가 null 일 때까지 반복



연결리스트의 응용–다항식의 덧셈(4/4)

- C = A + B 의 계산 과정(계속)



6장 정리

- **선형리스트**
  - 배열의 활용
  - 연결리스트의 활용
  - 리스트의 ADT
- **연결리스트**
  - 원형연결리스트
  - 이중연결리스트(다중)
  - 이의 응용

6장 리스트

-끝-

수고하셨습니다 ~

39