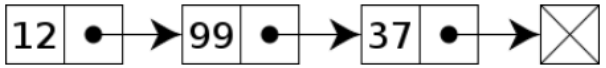
연결 리스트
(Linked Lists)

소프트웨어학과

노 서 영

단순 연결 리스트
(Singly Linked Lists and Chains)

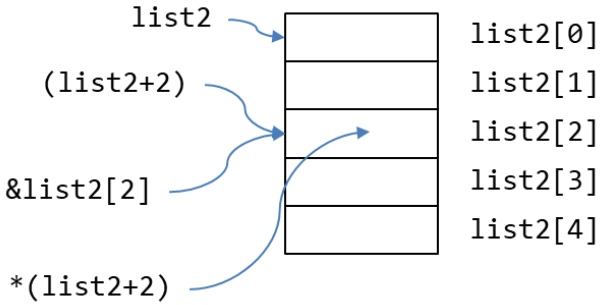
In computer science, a linked list is a linear collection of data elements, whose order is not given by their physical placement in memory. Instead, each element points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence.

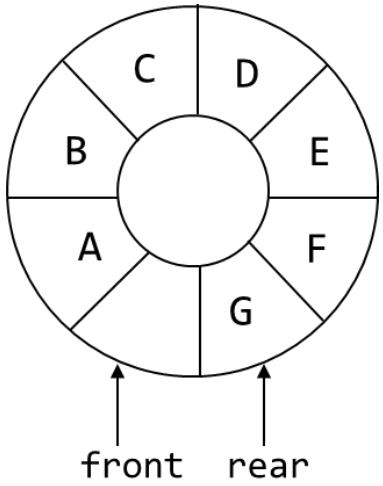


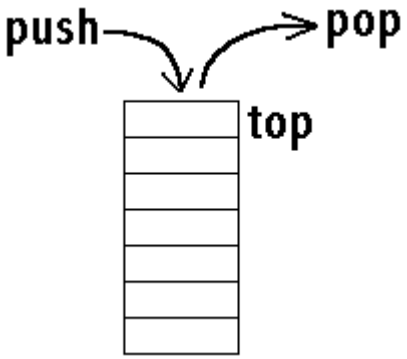
https://en.wikipedia.org/wiki/Linked_list

단순(단일) 연결리스트(Singly Linked Lists and Chains)

- **순차리스트의 특성**
  - 데이터 객체의 연속된 원소들이 일정한 거리만큼 떨어져 저장
  - **(배열)** $a_{i,j}$ → L_{ij}에 저장된다면, $a_{i,j+1}$ → L_{ij}+1에 저장
  - **(큐)** 큐의 i번째 요소가 L_i에 위치해있다면 $i+1$ → $(L_i+1)\%n$
  - **(스택)** 제일위의 원소가 L_T의 위치에 있다면, 바로 아래 원소 → L_T - 1

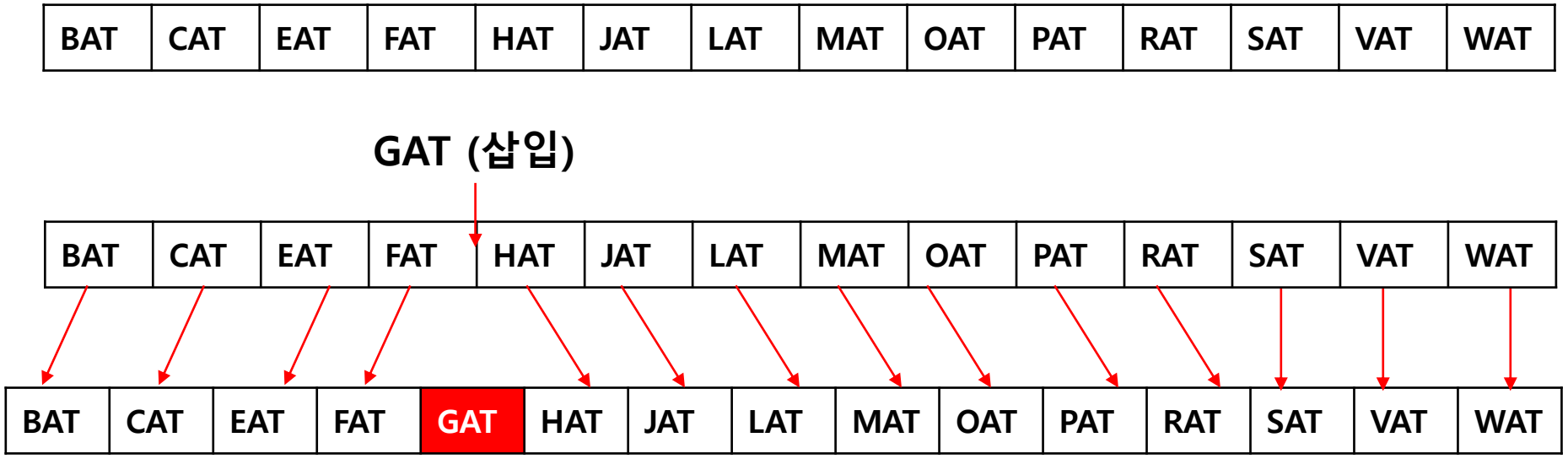






단순(단일) 연결리스트(Singly Linked Lists and Chains)

- Example: AT로 끝나는 3문자 영어 단어 리스트



새로운 배열을 할당, 복사

GAT 삽입 →

임의의 원소에 대한 삽입(insertion)과 삭제(deletion)에 많은 비용 초래

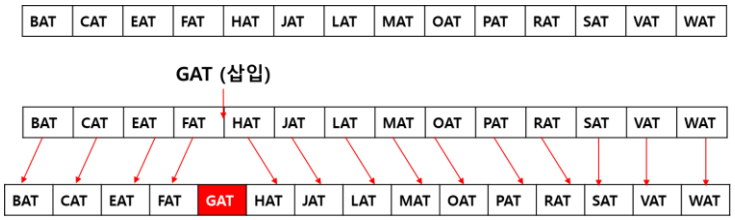
단순(단일) 연결리스트(Singly Linked Lists and Chains)

- **연결된(linked) 표현**
  - 순차 표현에서 제기된 데이터 이동 문제점을 해결
  - 각 원소들이 메모리 내의 어떤 곳에나 위치 가능
  - 원소에 정확한 순서로 접근하기 위한 정보가 부가적으로 필요
    - 원소의 주소정보

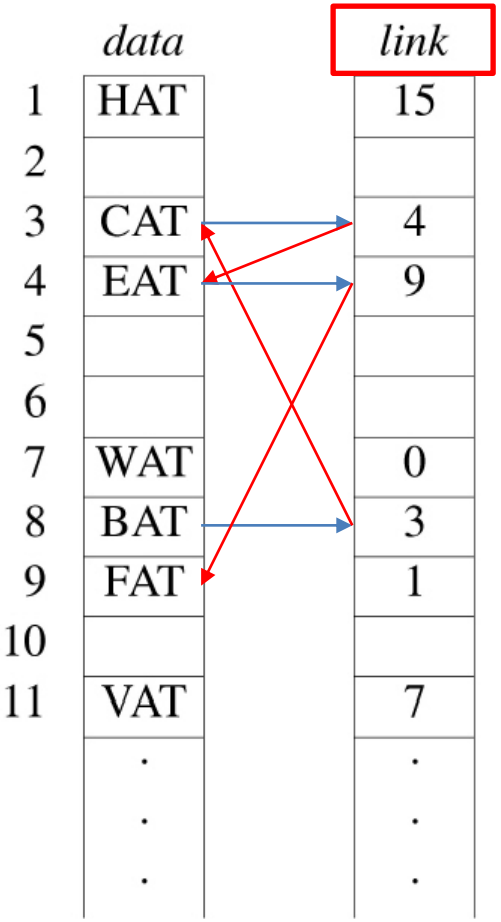
  - **노드(node)**
    - 0개 이상의 데이터 필드
    - 하나 이상의 링크(link) 또는 포인터(pointer): 다음 원소를 가리킴

단순(단일) 연결리스트(Singly Linked Lists and Chains)

- 노드(node)
  - 0개 이상의 데이터 필드
  - 하나 이상의 링크(link) 또는 포인터(pointer): 다음 원소를 가리킴



데이터의 비순차적 저장



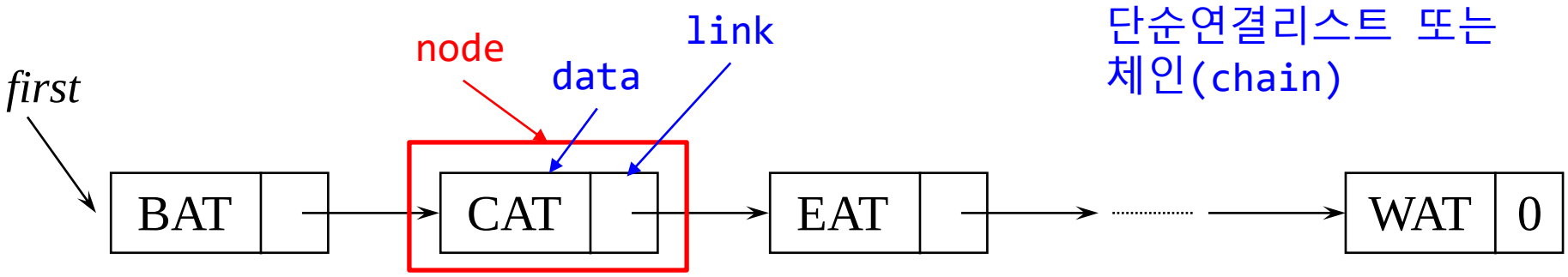
link: 위치정보

배열의 확대, 축소 시 여전히 많은 비용 발생

```
first = 8
data[8] = BAT  : 데이터
link[8] = 3    :  주소
data[3] = CAT  : 데이터
link[3] = 4    : 주소
data[link[3]] = EAT
```

단순(단일) 연결리스트(Singly Linked Lists and Chains)

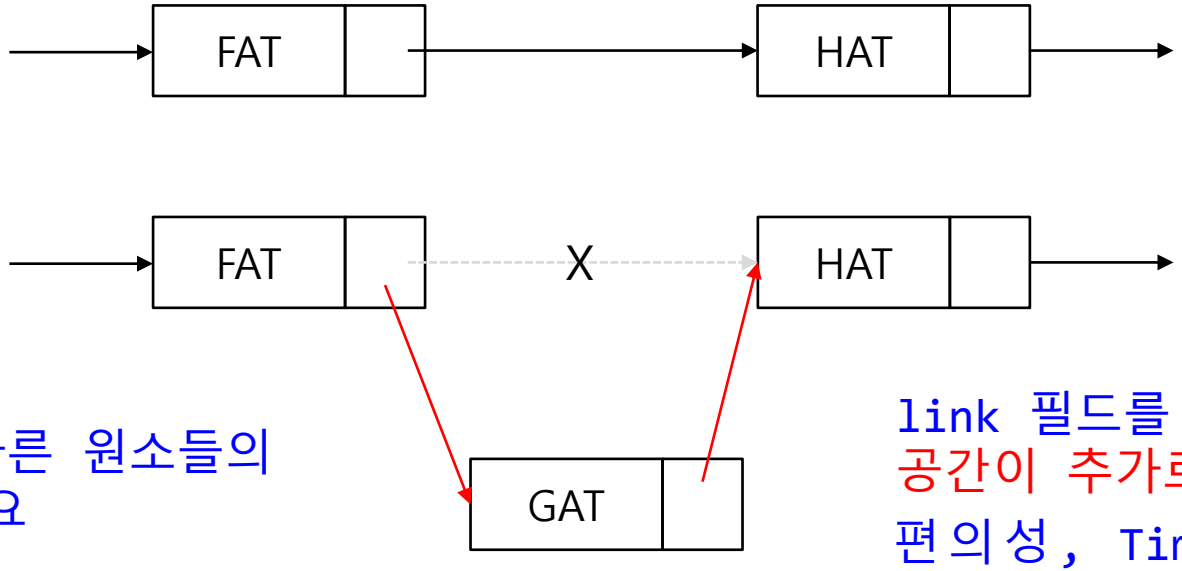
- **연결리스트의 일반적인 방법**
  - 노드들을 순차적으로 표시하고 링크를 화살표로 표현
  - 화살표로 다음 노드를 직관적으로 가리킴



- **Note**
  - 노드들은 실제로 순차적 위치에 존재하지 않는다 (memory 상에서).
  - 노드들의 위치(주소)는 실행 시마다 바뀔 수 있다.
  - link가 0인지(NULL)인지 검사하는 것 외에는 특정 주소 값을 찾지 않는다.

단순(단일) 연결리스트(Singly Linked Lists and Chains)

- **노드(node) 삽입: GAT원소를 FAT와 HAT 사이에 삽입**
  - 현재 사용하고 있지 않은 노드 하나(a)를 가져온다 (malloc())
  - 노드 a의 data 필드에 'GAT'를 설정한다.
  - a의 link 필드에 FAT 다음 노드, 즉 HAT를 저장하고 있는 노드를 가리키도록 한다.
  - FAT를 포함하고 있는 노드의 link 필드가 a를 가리키도록 한다.



삽입 시 다른 원소들의 이동 불필요

link 필드를 위한 저장 공간이 추가로 사용

편의성, Time Complexity 관점에서 감수할 만 함

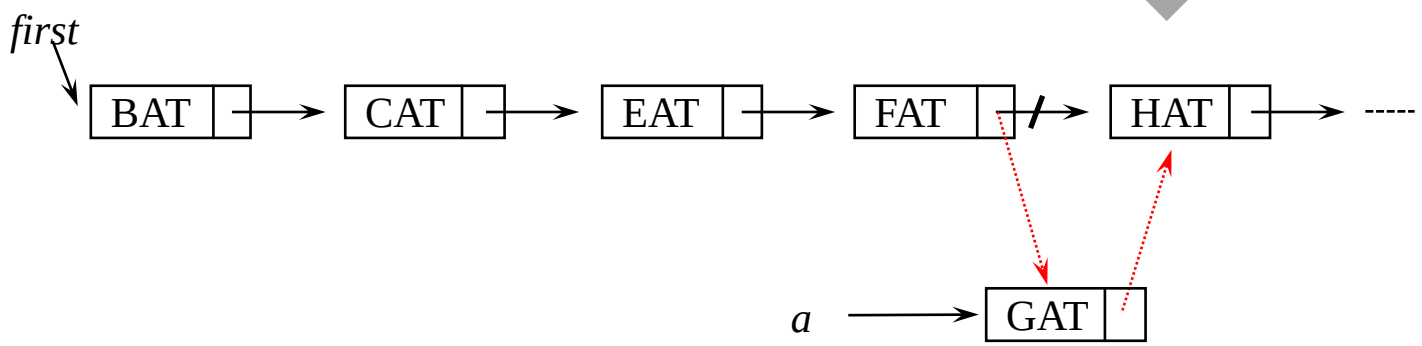
단순(단일) 연결리스트(Singly Linked Lists and Chains)

	data	link
1	HAT	15
2		
3	CAT	4
4	EAT	9
5	GAT	1
6		
7	WAT	0
8	BAT	3
9	FAT	5
10		
11	VAT	7

(a) Data[5]에 GAT 삽입

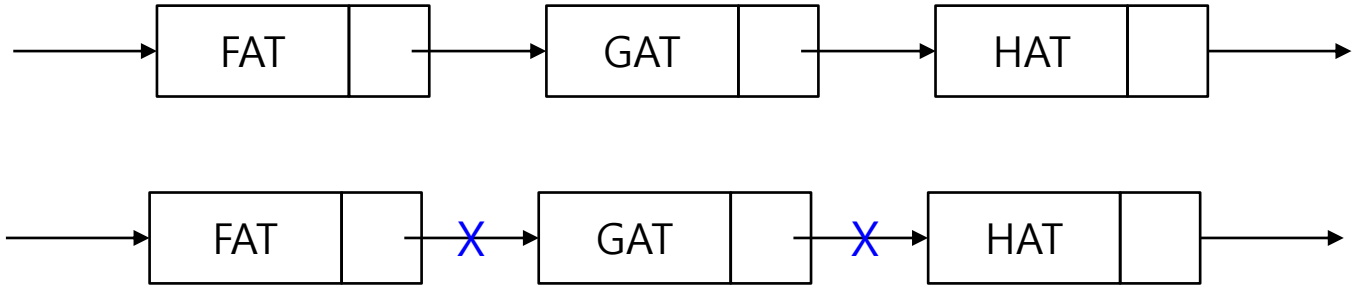
link 수정

(b) 리스트에 노드 GAT 삽입

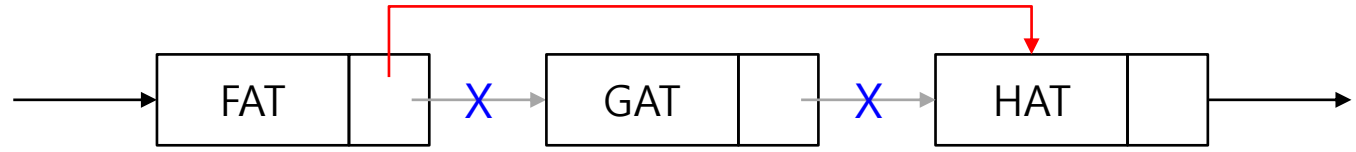


단순(단일) 연결리스트(Singly Linked Lists and Chains)

- **노드(node) 삭제: GAT원소를 삭제**
  - GAT 바로 앞에 있는 원소 FAT 찾기
  - FAT의 link를 HAT의 위치로 설정



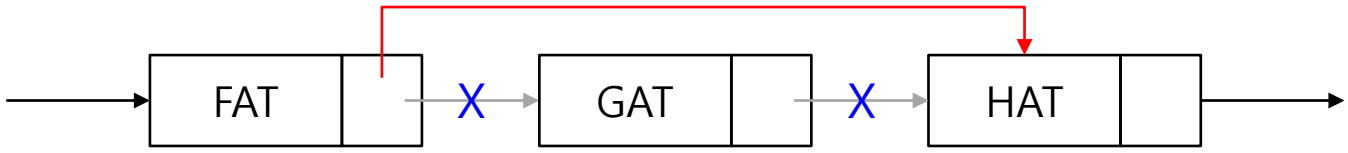
FAT → HAT 접근방법 X
GAT → HAT link 복사



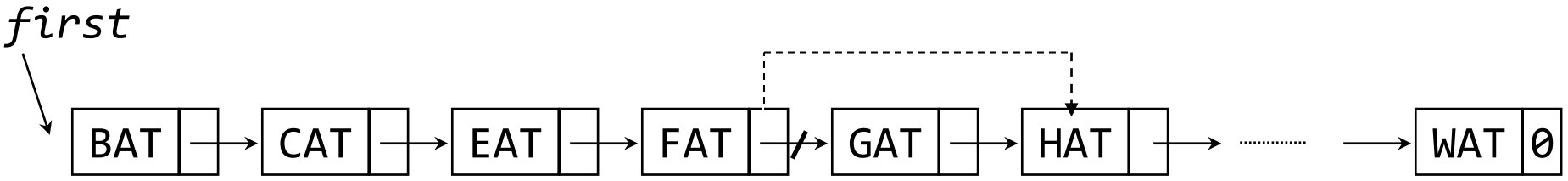
GAT → HAT link 삭제
GAT Node free

단순(단일) 연결리스트(Singly Linked Lists and Chains)

- 노드(node) 삭제: GAT원소를 삭제



GAT → HAT link 삭제
GAT Node free



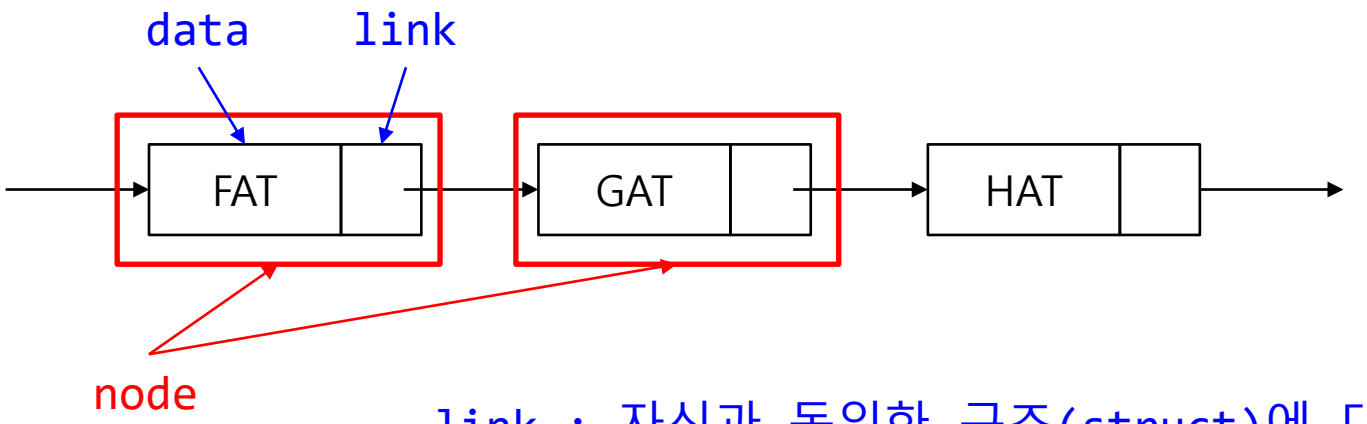
리스트에서 GAT 삭제

GAT으로 접근할 수 만 있다면, GAT → HAT link를 통해 HAT 이후의 데이터를 접근할 수 있다 (security hole)

C에서의 체인 표현
(Representing Chains in C)

C에서 체인표현(Representing Chains in C)

- **Linked List 구현을 위해 필요한 사항**
  - 노드 구조 (struct)
  - 자기와 동일한 타입에 대한 링크 (self-referential structure)
  - 노드 생성시 메모리할당 (malloc())
  - 더 이상 필요하지 않은 노드의 메모리 해제 (free())



link : 자신과 동일한 구조(struct)에 대한 포인터

C에서 체인표현(Representing Chains in C)

- [예제 4.1] C에서의 노드 정의: 리스트를 위한 노드 구조

```
typedef struct listNode *listPointer;
typedef struct listNode {
            char data[4];
            listPointer link;
            };
```

존재하지 않은 타입에 대한 포인터 생성가능

그렇지 않으면 모순

struct_test.c

- 새로운 공백 리스트 생성
  - listPointer first = NULL;

← first는 리스트의 시작 주소를 포함

- 공백 리스트인지 검사하는 매크로 IS_EMPTY
  - #define IS_EMPTY(first)(!(first))

struct_test.c

- 새로운 노드를 생성하기 위한 매크로 MALLOC
  - MALLOC(first, sizeof(*first));

malloc(sizeof(*first))
maclloc(sizeof(struct listNode))

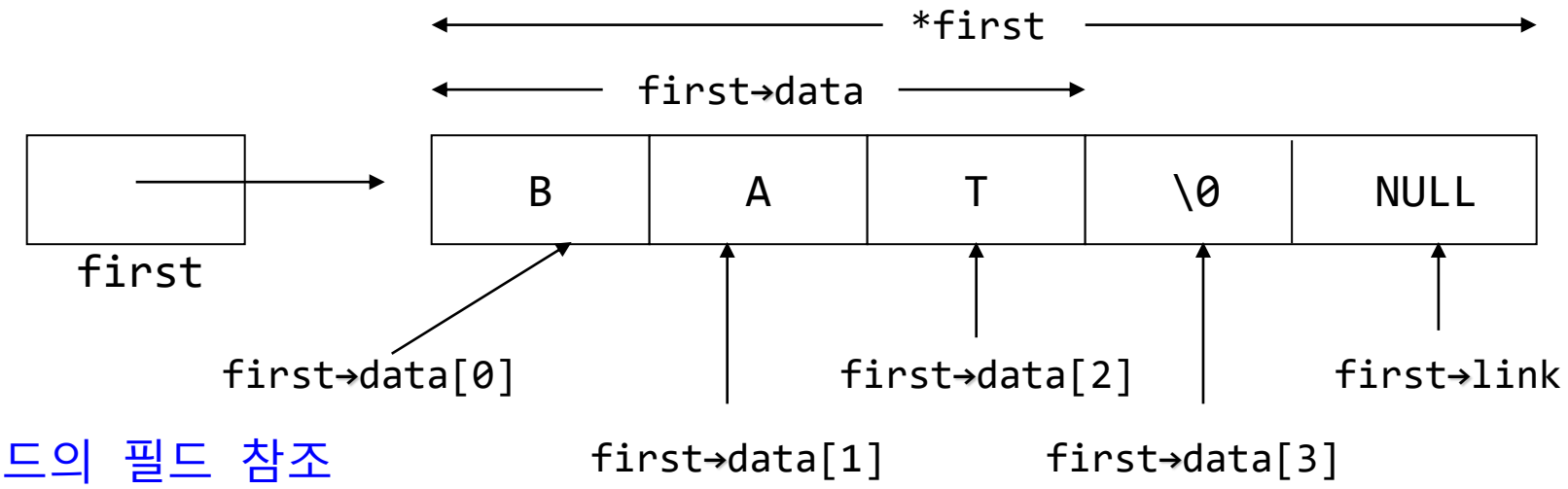
C에서 체인표현(Representing Chains in C)

- 노드의 필드값 정의: -> 연산자 사용
  - bat이란 단어를 리스트에 넣을 때
  - strcpy(first->data, “bat”); first->link = NULL;

*(first).data
*(first).link

```
typedef struct listNode *listPointer;
typedef struct listNode {
            char data[4];
            listPointer link;
            };
```

```
listPointer first = NULL;
MALLOC(first, sizeof(*first));
```



노드의 필드 참조

first->data ⟺ (*first).data

C에서 체인표현(Representing Chains in C)

- [예제 4.2] 2-노드 연결 리스트

```
typedef struct listNode *listPointer
typedef struct listNode {
            int data;
            listPointer link;
            };
```

정수들의 연결 리스트 생성을 위한 구조체

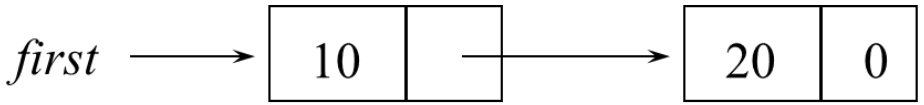
- 두 개의 노드를 가진 연결 리스트를 생성

```
listPointer create2()
{ /* 두 개의 노드를 가진 연결 리스트의 생성 */
      listPointer first, second;
      MALLOC(first, sizeof(*first));
      MALLOC(second, sizeof(*second));
      second->link = NULL;
      second->data = 20;
      first->data = 10;
      first->link = second;
      return first;
}
```

노드 생성

data에 값 저장

노드 연결

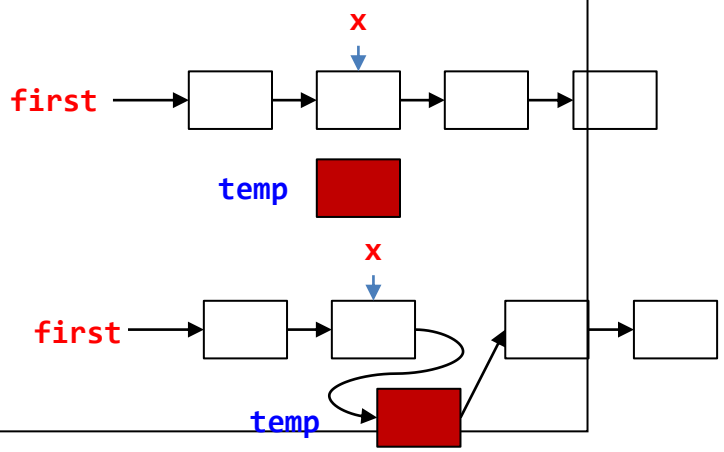


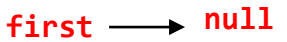
C에서 체인표현(Representing Chains in C)

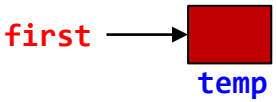
- **[예제 4.3] 리스트 삽입**
  - 임의의 노드 뒤에 데이터 필드 값이 50인 노드를 삽입

```
void insert(listPointer *first, listPointer x)
{ /* data=50인 새로운 노드를 리스트 first의 node 뒤에 삽입 */

        listPointer temp;
        MALLOC(temp, sizeof(*temp));
        temp->data = 50;
        if(*first) {
                temp->link = x->link;
                x->link = temp;
        }
        else {
                temp->link = NULL;
                *first = temp;
        }
}
```

first != NULL → 리스트 존재

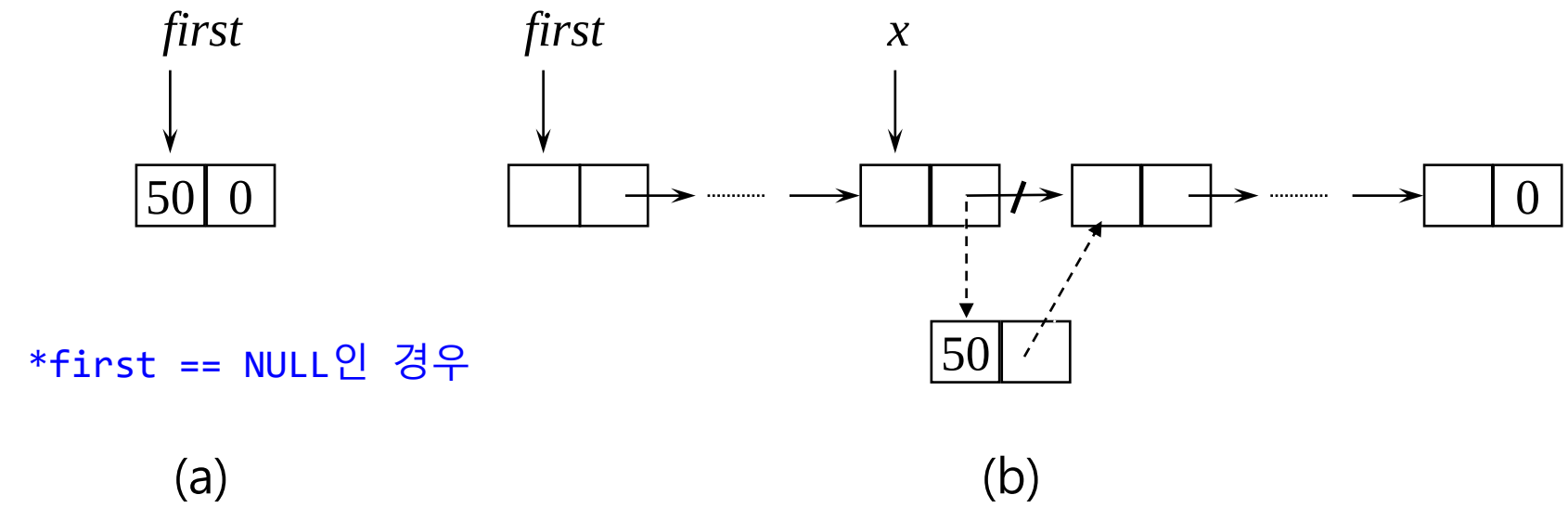






C에서 체인표현(Representing Chains in C)

- **[예제 4.3] 리스트 삽입**
  - 임의의 노드 뒤에 데이터 필드 값이 50인 노드를 삽입

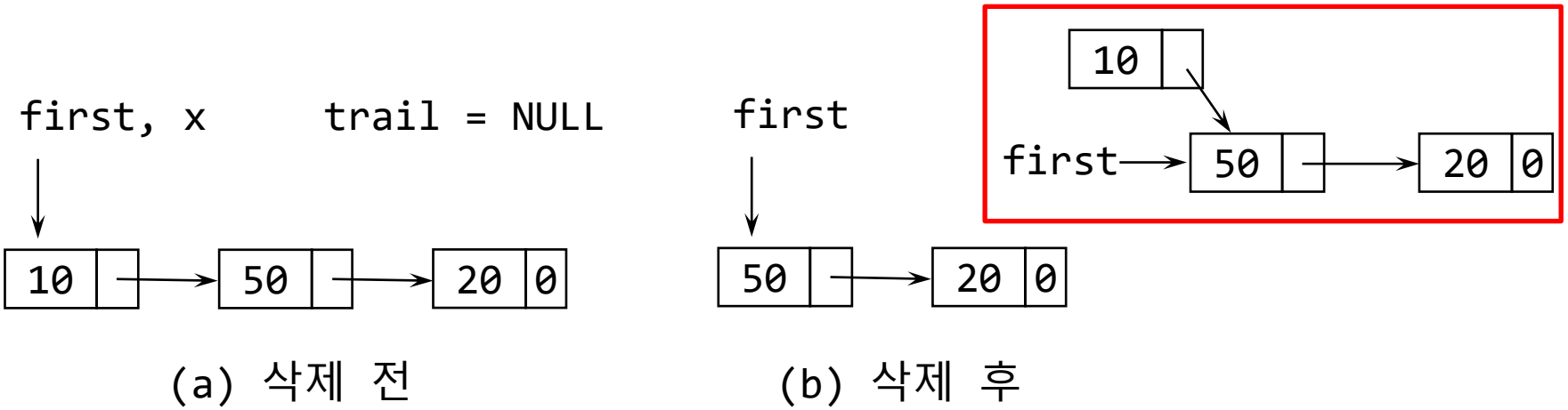


C에서 체인표현(Representing Chains in C)

- **[예제 4.4] 리스트 삭제**
  - 노드의 위치에 따라 삭제하는 방법이 다름
  - first : 리스트의 시작을 가리키는 포인터
  - x : 삭제하고자 하는 노드를 가리키는 포인터
  - trail : 삭제할 노드의 선행 노드를 가리키는 포인터

삭제하고자 하는 노드가 첫 번째 인 경우

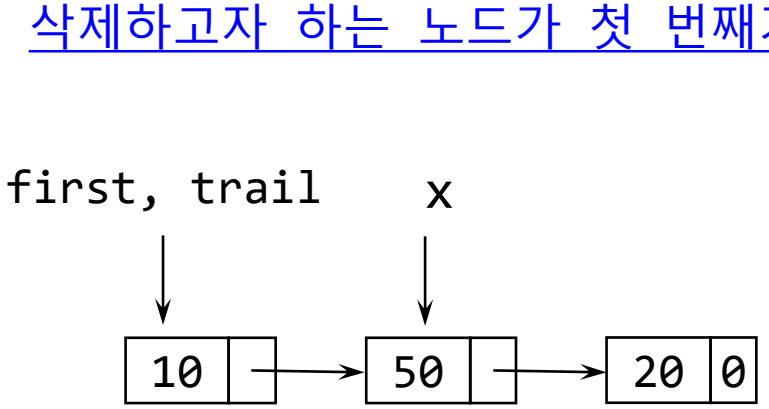
delete(&first, NULL, first); 전과 후의 리스트



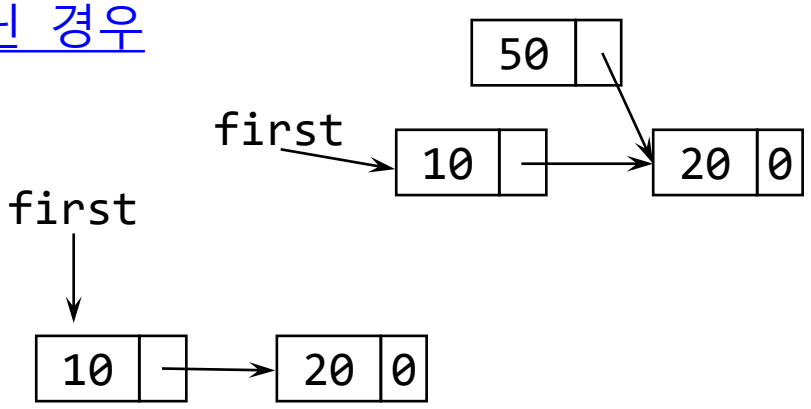
C에서 체인표현(Representing Chains in C)

- **[예제 4.4] 리스트 삭제**
  - first : 리스트의 시작을 가리키는 포인터
  - x : 삭제하고자 하는 노드를 가리키는 포인터
  - trail : 삭제할 노드의 선행 노드를 가리키는 포인터

삭제하고자 하는 노드가 첫 번째가 아닌 경우



(a) 삭제 전



(b) 삭제 후

```
trail->link = x->link;
```

delete(&first, trail, x) 함수 호출 전과 후

C에서 체인표현(Representing Chains in C)

- [예제 4.4] 리스트 삭제

first@delete

multiple_pointer.c

```
void delete(listPointer *first, listPointer trail, listPointer x)
{ /* 리스트로부터 노드를 삭제, trail은 삭제될 x의 선행 노드이며
     first는 리스트의 시작 */
     if (trail)
          trail->link = x->link;
     else
         *first = (*first)->link;
     free(x);
}
```

← 첫 번째 노드가 아닐 경우
delete(&first, trail, x);

← 첫 번째 노드 삭제
delete(&first, NULL, first);

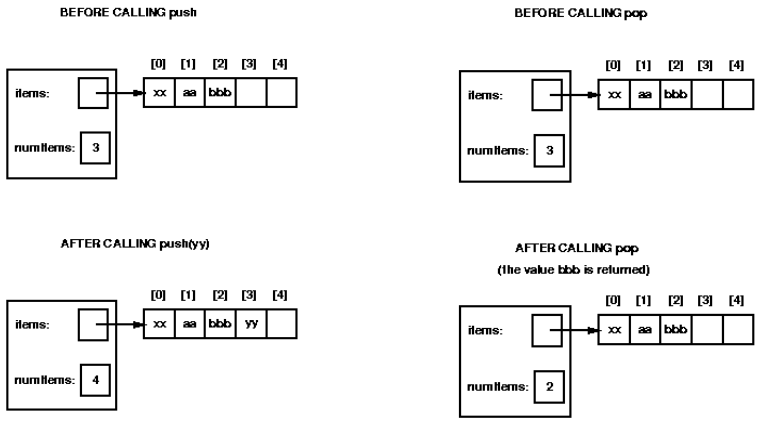
- [예제 4.5] 리스트의 출력

first@printList

```
void printList(listPointer first)
{
      printf("The list contains: ");
      for (; first; first = first->link)
            printf("%4d",first->data);
      printf("\n");
}
```

왜 first@delete는 이중 포인터이고
first@printList는 싱글 포인터여야 하는가?

연결 스택과 큐
(Linked Stacks and Queues)

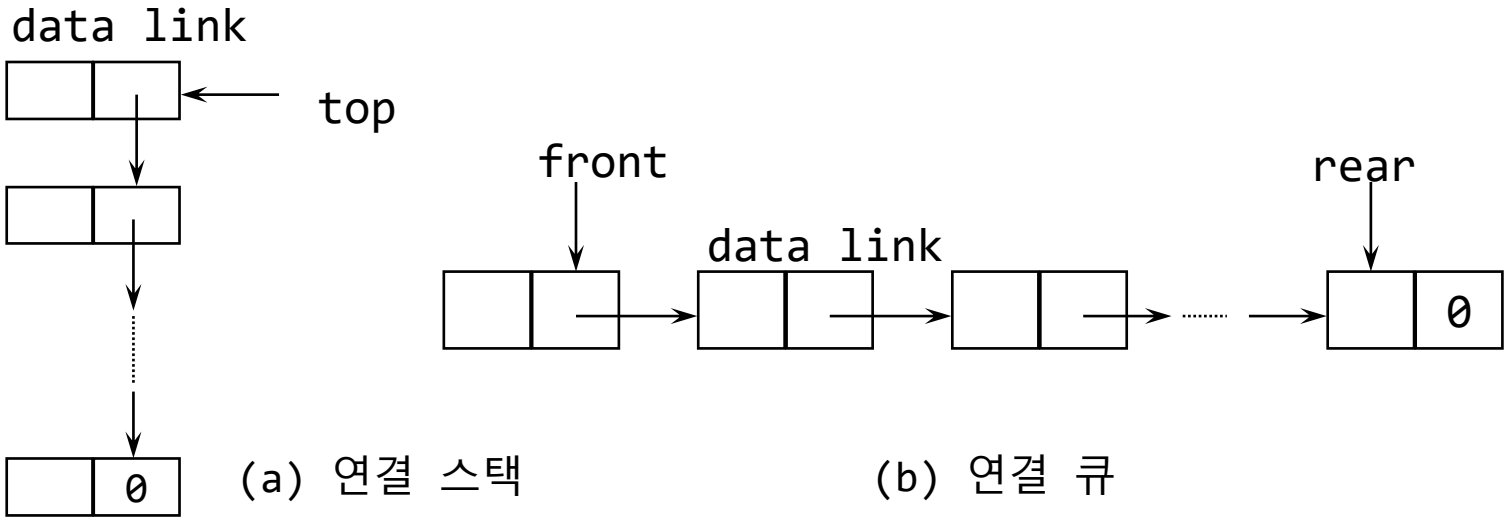


http://pages.cs.wisc.edu/~vernon/cs367/notes/5.STACKS-AND-QUEUES.html

연결 스택과 큐(Linked Stacks and Queues)

- **여러 개의 스택이나 큐가 동시에 있을 때**
  - 다중 스택과 큐를 순차적으로 표현할 때, 많은 오버헤드 발생

- **연결 스택과 큐**
  - 링크의 화살표 방향 : 노드의 삽입과 삭제가 편리하게 만들어 줌
  - **(연결 스택)** 톱에서 노드 삽입/삭제 용이
  - **(연결 큐)** 뒤에선 노드를 쉽게 삽입 / 앞에선 노드를 쉽게 삭제

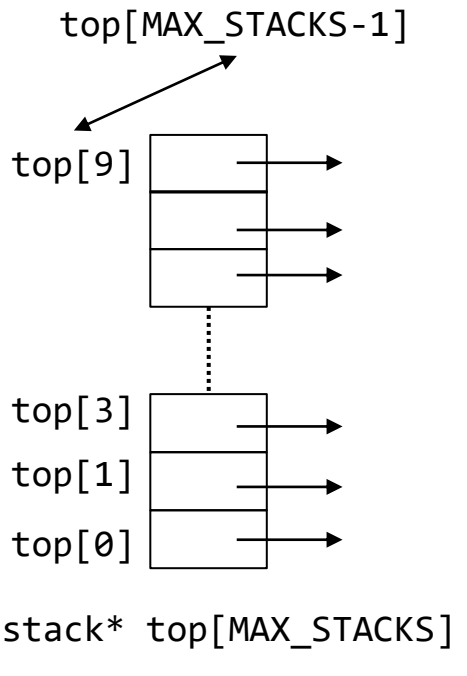


연결 스택과 큐(Linked Stacks and Queues)

- **연결 스택(Linked Stacks)**
  - n ≤ MAX_STACKS 개의 스택을 동시에 나타내려면

```
#define MAX_STACKS 10  /*  스택의 최대 수 */
typedef struct {
        int key;
        /* 기타 필드 */
        } element;

typedef struct stack *stackPointer;

typedef struct stack {
        element data;
        stackPointer link;
        };
stackPointer top[MAX_STACKS];
```

선언과 접근에 대한 구별



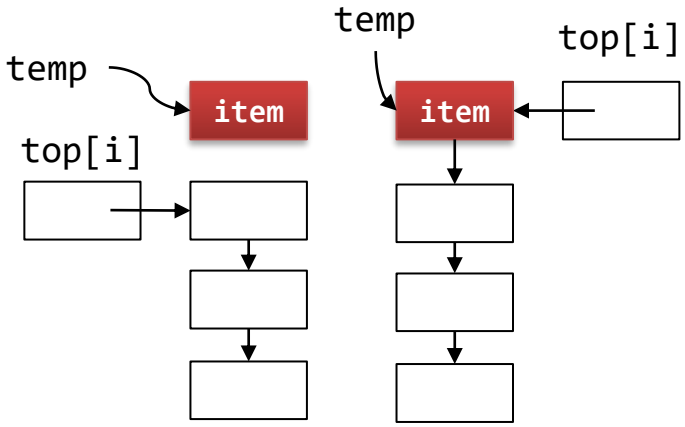
  - 스택의 초기 조건: top[i] = NULL, 0≤i<MAX_STACKS
  - 경계 조건: top[i]=NULL, i번째 스택이 공백이면 (그 역도 성립)

    top[i]=NULL ↔ i번째 스택이 공백

연결 스택과 큐(Linked Stacks and Queues)

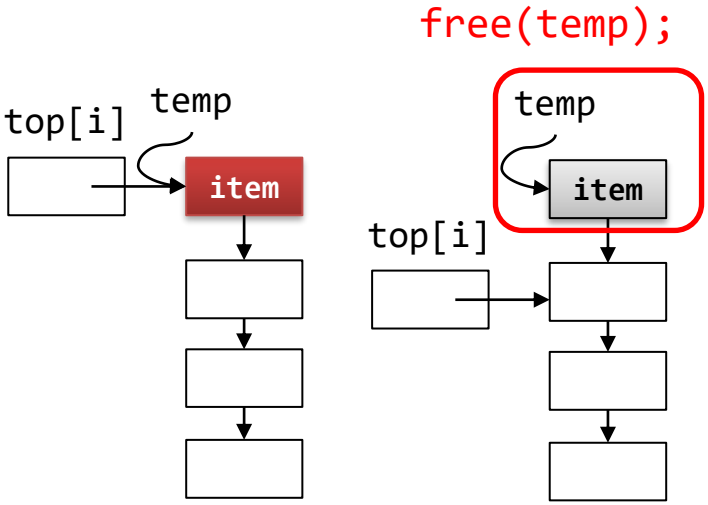
- 연결 스택에 삽입

```
void push(int i, element item)
{ /* i번째 스택에 원소를 삽입 */
        stackPointer temp;
        MALLOC(temp, sizeof(*temp));
        temp->data = item;
        temp->link = top[i];
        top[i] = temp;
}
```



- 연결 스택에서 삭제

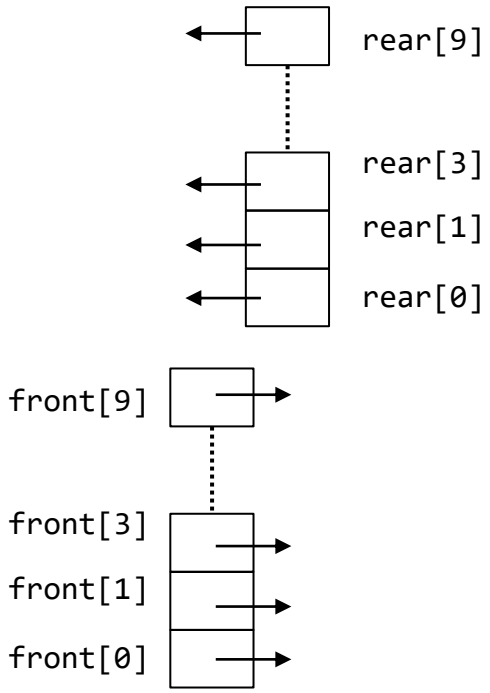
```
element pop(int i)
{ /* i번째 스택으로부터 톱 원소를 삭제 */
        stackPointer temp = top[i];
        element item;
        if(!temp)
                return stackEmpty();
        item = temp->data;
        top[i] = temp->link;
        free(temp);
        return item;
}
```



연결 스택과 큐(Linked Stacks and Queues)

- **연결 큐**
  - n ≤ MAX_QUEUE 개의 큐를 동시에 나타내려면

```
#define MAX_QUEUE 10    /* 큐의 최대 수 */
    typedef struct queue *queuePointer;

    typedef struct queue {
            element data;
            queuePointer link;
            };

    queuePointer front[MAX_QUEUES];
    queuePointer rear[MAX_QUEUES];
```



  - 큐의 초기 조건: front[i] = NULL, 0≤i<MAX_QUEUES
  - 경계조건: front[i]=NULL, i번째 큐가 공백이면 (그 역도 성립)

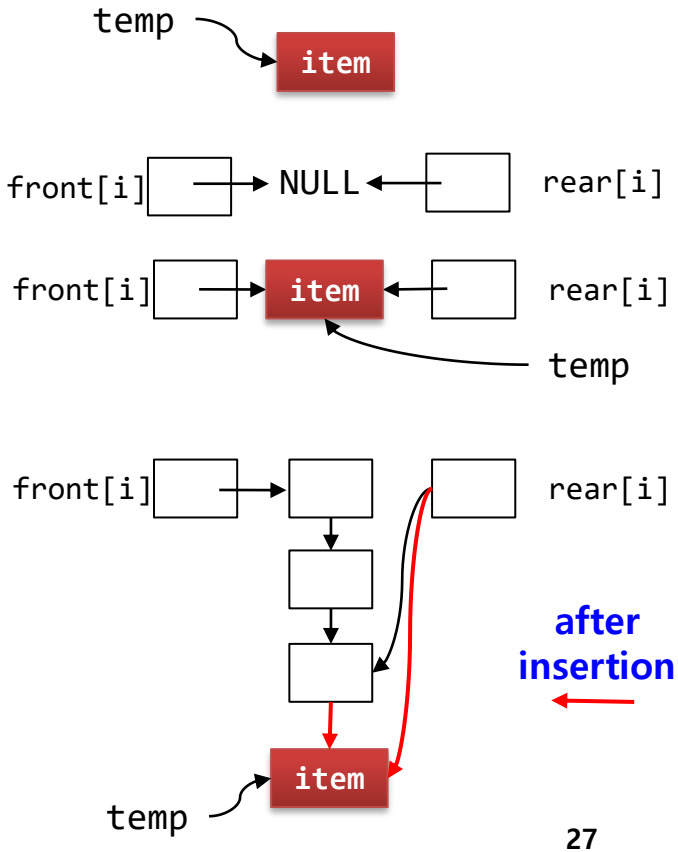
연결 스택과 큐(Linked Stacks and Queues)

- **연결 큐 뒤에 삽입**
  - 큐가 공백인지 조사해야 함
  - 공백이면, front를 새로운 노드를 가리키도록 변경시킴
  - 그렇지 않으면, rear의 링크 필드를 새로운 노드를 가리키도록

```
void addq(i, item)
{
     /* 큐 i의 뒤에 원소를 삽입 */
      queuePointer temp;
      MALLOC(temp, sizeof(*temp));
      temp->data = item;
      temp->link = NULL;

      if(front[i])
           rear[i]->link = temp;
      else
           front[i] = temp;
      rear[i] = temp;
}
```

empty queue →

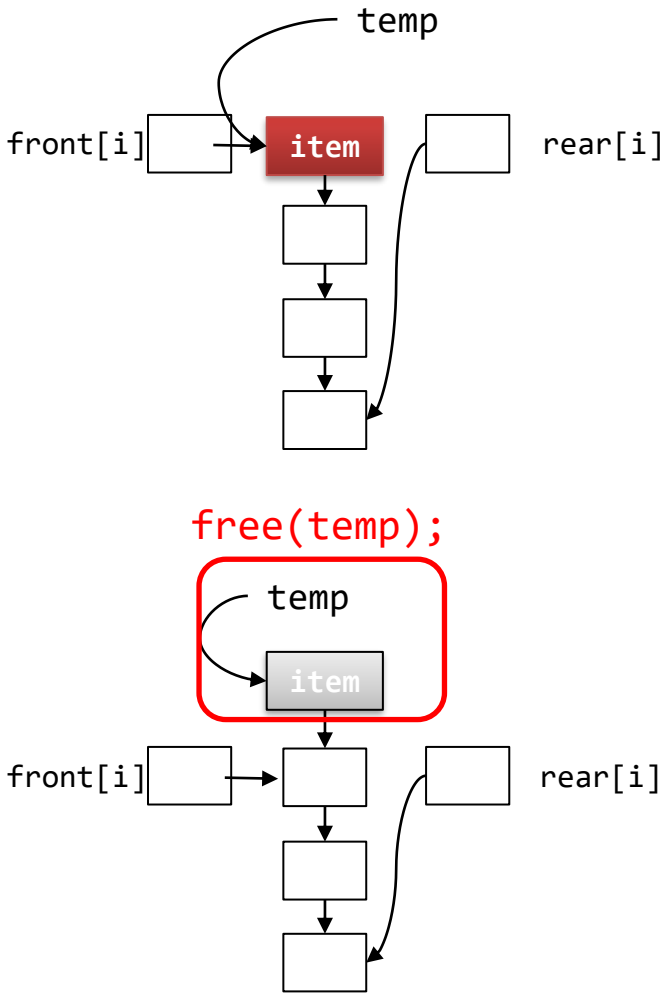
stack full 검사하는 로직 없음
→ malloc() 활용



연결 스택과 큐(Linked Stacks and Queues)

- **연결 큐의 앞으로부터 삭제**
  - 리스트의 현재 시작 노드를 삭제

```
element deleteq(int i)
{ /* 큐 i로부터 원소를 삭제 */
     queuePointer temp = front[i];
     element item;

     if(!temp)
          return queneEmpty();
     item = temp->data;
     front[i] = temp->link;

     free(temp);
     return item;
}
```



연결 스택과 큐(Linked Stacks and Queues)

- **n-스택, m-큐를 연결리스트로 구현 → 구현의 단순함, Time Complexity의 감소**
  - 추가공간을 만들기 위해 스택이나 큐를 이동시킬 필요가 없다.

- **가용메모리가 남아있을 때까지 크기를 증대시킬 수 있다 (malloc())**

- **링크에 대한 오버헤드가 발생하지만...장점이 많다.**
  - 간단한 방법으로 리스트를 표현할 수 있는 능력
  - 연결된 표현을 처리하기 위한 감소된 연산 시간

다항식
(Polynomials)

In mathematics, a polynomial is an expression consisting of variables (also called indeterminates) and coefficients, that involves only the operations of addition, subtraction, multiplication, and non-negative integer exponents of variables.

Polynomials

$2x^2 + 6x - 9$

$-x^3 + 9$

$4x^4 + 5x^3 - 8x^2 + 12x + 24$

Not Polynomials

$10x^{-1} + 6x^2$

$\sqrt{x} - 2$

$\frac{3}{x} + 5$

다항식의 표현(Polynomial Representation)

Page 70. 배열을 이용한 다항식표현과 비교

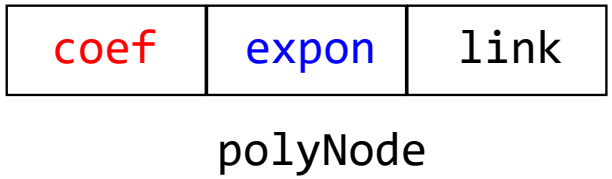
- 다항식의 표현

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0 x^{e_0}$$

a_i : 0이 아닌 계수
e_i : 음수가 아닌 정수 지수
$e_{m-1} > e_{m-2} > \ldots > e_1 > e_0 \geq 0$

- 리스트를 사용해서 다항식을 표현
  - 노드(node): 계수, 지수, 다음 항을 가리키는 포인터 등 3개의 필드
  - 계수가 정수라고 가정

```
typedef struct polyNode *polyPointer;
typedef struct polyNode {
        int coef;
        int expon;
        polyPointer link;
        };
polyPointer a,b;
```

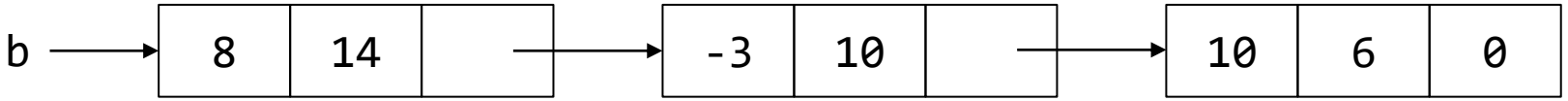


다항식의 표현(Polynomial Representation)

- **Example:**
  - a = $3x^{14}+2x^{8}+1$
  - b = $8x^{14}-3x^{10}+10x^{6}$

polyNode

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



$8x^{14} + 3x^{10} + 10x^{6}$

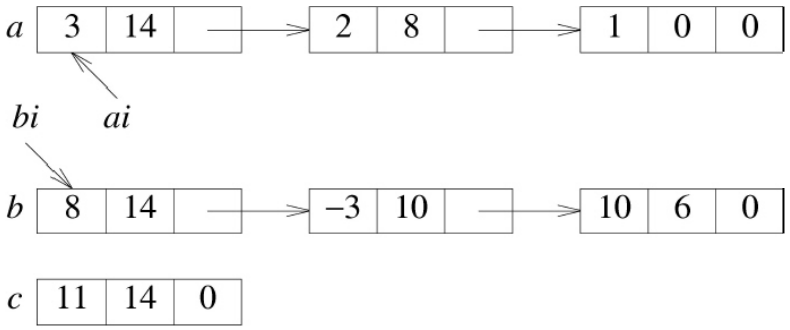
다항식의 덧셈(Adding Polynomials)

Page 74. 배열을 이용한 다항식 덧셈과 비교

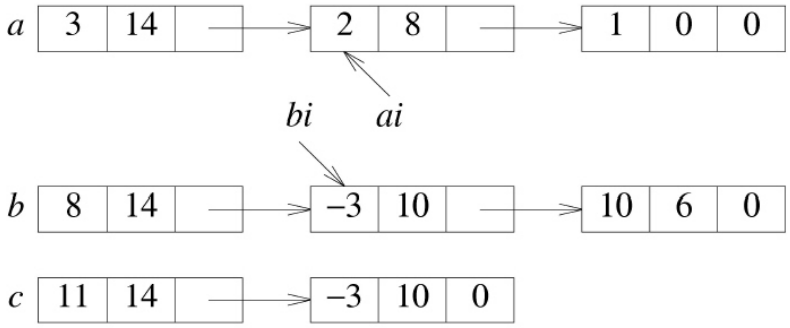
- 포인터 a와 b가 가리키는 노드에서 시작되는 항들을 비교
- (a->expon = b->expon)
  - 만일 두 항의 지수가 같으면 계수를 더함
  - 결과 다항식에 새로운 항을 만듦
  - 포인터 a,b를 다음 노드로 이동
- (a->expon < b->expon)
  - b의 항과 같은 항을 만들어 결과 다항식 d에 첨가
  - 포인터 b를 다음 노드로 이동
- (a->expon > b->expon)
  - a의 항과 같은 항을 만들어 결과 다항식 d에 첨가
  - 포인터 a를 다음 노드로 이동

다항식의 덧셈(Adding Polynomials)

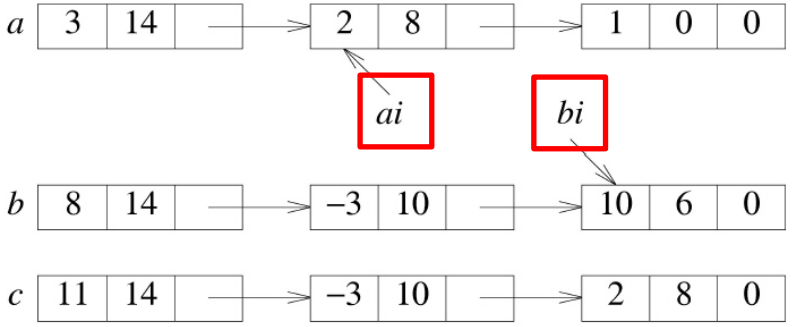
c = a + b



(i) a->expon == b->expon



(ii) a->expon < b->expon



(iii) a->expon > b->expon

다항식의 덧셈(Adding Polynomials)

```
polyPointer padd(polyPointer a, polyPointer b)
{
    polyPointer c, rear, temp;
    int sum;
    MALLOC(rear, sizeof(*rear));
    c= rear;
    while(a && b)
          switch(COMPARE(a->expon, b->expon)) {
               case -1: /* a->expon < b->expon */
                        attach(b->coef, b->expon, &rear);
                        b = b->link;
                        break;
               case 0 : /* a->expon = b->expon */
                        sum = a->coef + b->coef;
                        if(sum) attach(sum, a->expon, &rear);
                        a = a->link; b = b->link; break;
               case 1: /* a->expon > b->expon */
                        attach(a->coef, a->expon, &rear);
                        a = a->link;
          }
    /* 리스트 a와 리스트 b의 나머지를 복사 */
    for(; a; a=a->link) attach(a->coef, a->expon, &rear);
    for(; b; b=b->link) attach(b->coef, b->expon, &rear);
    rear->link = NULL;
    /* 필요 없는 초기 노드를 삭제 */
    temp = c; c= c->link; free(temp);
    return c;
}
```

/* a와 b가 합산된 다항식을 반환 */

← node를 연결하기위한 임시 node 생성

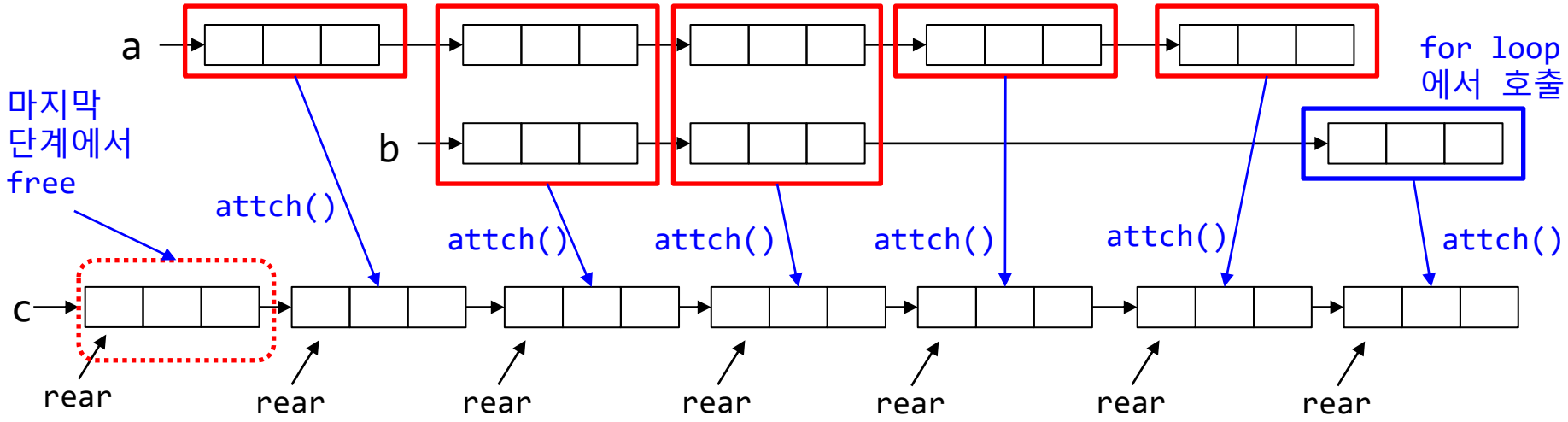
← node를 연결하기위한 임시 node 제거

for loop 전에 if a == NULL or b == NULL 검사 하지 않음 (code simplicity)

새로운 항을 만드는 attach()함수를 별로도 만드는 것이 효율적

다항식의 덧셈(Adding Polynomials)

```
void attach(float coefficient, int exponent, polyPointer *ptr)
{ /* coef=coefficient 이고 expon=exponent인 새로운 노드를 생성하고,
      그것을 ptr에 의해 참조되는 노드에 첨가한다. ptr을 갱신하여
      이 새로운 노드를 참조하도록 한다. */

      polyPointer temp;
      MALLOC(temp, sizeof(*temp));
      temp->coef = coefficient;
      temp->expon = exponent;
      (*ptr)->link = temp;
      *ptr = temp;
}
```



다항식의 덧셈(Adding Polynomials)

- **padd 함수 분석: 세가지 비용요소**
  - 계수 덧셈
  - 지수 비교
  - 결과값(c)을 위한 새로운 노드

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0 x^{e_0} \qquad a_i, b_i \neq 0$$

$$e_{m-1} > \cdots > e_0 \geq 0$$

$$B(x) = b_{n-1}x^{f_{n-1}} + \cdots + b_0 x^{f_0} \qquad f_{n-1} > \cdots > f_0 \geq 0$$

계수 덧셈 $0 \leq$ 계수덧셈의 횟수 $\leq \min\{m, n\}$ ← O(min(m,n))

지수 비교 while loop에서 각각 지수를 한번씩만 비교
항의 총 수 = m+n ← O(m+n)

$$e_{m-1} > f_{n-1} > e_{m-2} > f_{n-2} > \cdots > e_1 > f_1 > e_0 > f_0$$

결과 노드 최대 항 수가 m+n ← O(m+n)

padd의 Time Complexity = O(m+n)

Optimal Solution!! (p.176)

다항식의 곱셈(Multiplying Polynomials)

$$e(x) = a(x) * b(x) + d(x)$$

programming

```
polyPointer a, b, d, e;
.
.
.
a = readPoly();
b = readPoly();
d = readPoly();
temp = pmult(a, b);
e = padd(temp, d);
printPoly(e)
```

a(x)*b(x)의 결과 값을 temp에 저장하고, temp는 다시 d(x)와 연산.

temp를 사용하고 반환 →

메모리 가용도 높일 수 있음

다항식의 삭제(Erasing Polynomials)

- 함수 erase : temp에 있는 노드들을 하나씩 반환

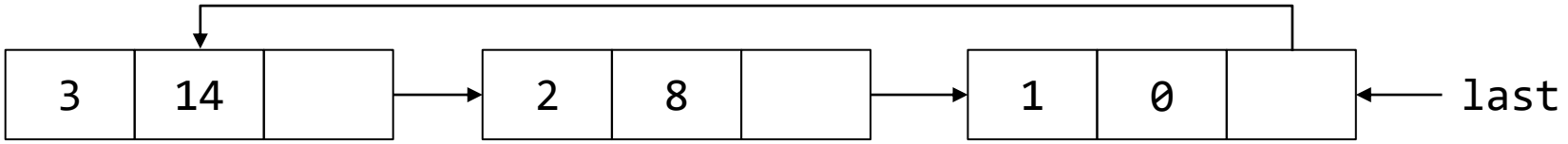
```
void erase(polyPointer *ptr)
{ /* ptr에 의해 참조되는 다항식을 제거 */
     polyPointer temp;
     while(*ptr) {
          temp = *ptr;
          *ptr = (*ptr)->link;
          free(temp);
     }
}
```

```
typedef struct polyNode *polyPointer;
typedef struct polyNode {
        int coef;
        int expon;
        polyPointer link;
        };
polyPointer a,b;
```

왜 ptr은 이중 포인터인가?
주의: ptr@erase

다항식의 원형 리스트 표현
(Circular List Representation of Polynomials)

- **표현방법**
  - 마지막 노드가 리스트의 첫 번째 노드를 가리키도록 함
    - (cf) 체인(chain): 마지막 노드의 링크 필드 값이 NULL
  - $3x^{14}+2x^{8}+1$의 원현 리스트 표현

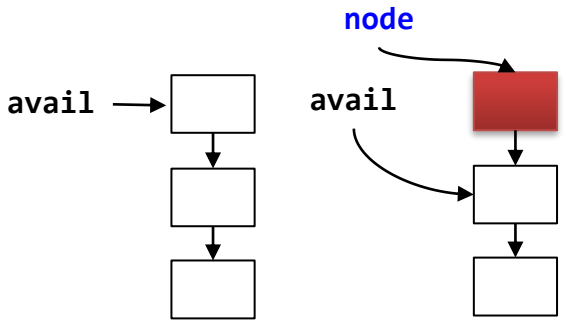


  - 해제(free)된 노드를 체인 형태의 리스트로 유지
  - 가용 공간 리스트(available space list) 혹은 avail 리스트
    - avail : 리스트에서 첫 번째 노드를 가리키는 포인터(초기값 NULL)
  - 새로운 노드가 필요하면 이 리스트를 조사
  - malloc, free 사용하는 대신 getNode, retNode 사용

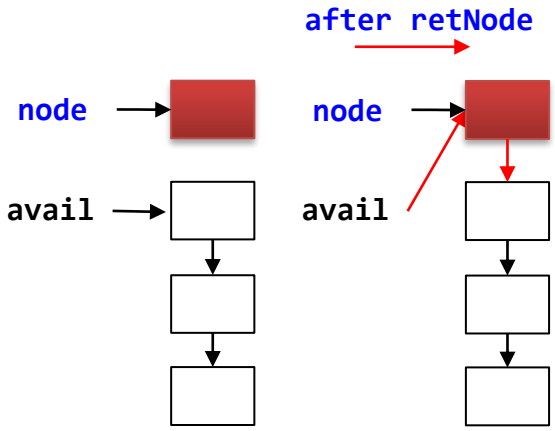
다항식의 원형 리스트 표현
(Circular List Representation of Polynomials)

- **getNode, retNode**

```
polyPointer getNode(void) /* 사용할 노드를 제공 */
{
        polyPointer node;
        if (avail) {
              node = avail;
              avail = avail->link;
        }
        else
              MALLOC(node, sizeof(*node));
        return node;
}
```




```
void retNode(polyPointer node)
{
         /* 가용 리스트에 노드를 반환 */
         node->link = avail;
         avail = node;
}
```

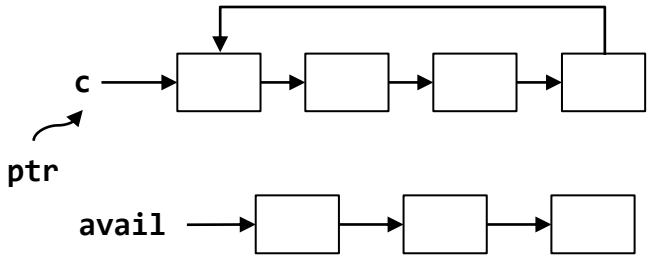


다항식의 원형 리스트 표현
(Circular List Representation of Polynomials)

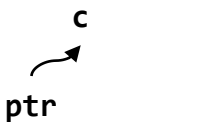
원형리스트 전체를 avail로 리턴

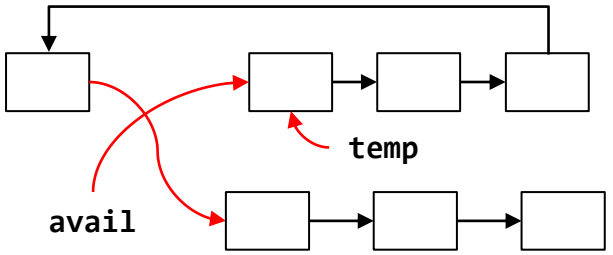
- cerase

```
void cerase(polyPointer *ptr)
{
   /* ptr가 가리키는 원형 리스트를 제거 */
   polyPointer temp;
   if (*ptr) {
      temp = (*ptr)->link;
      (*ptr)->link = avail;
      avail = temp;
      *ptr = NULL;
   }
}
```



ptr이 가리키는 원형 리스트의 크기에 상관없이 O(1) Time Complexity에 원형 리스트 제거 가능

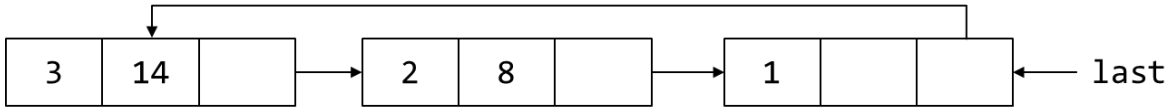




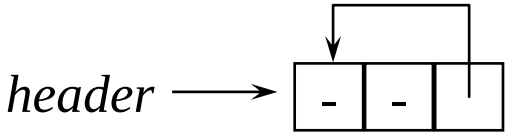
다항식의 원형 리스트 표현
(Circular List Representation of Polynomials)

- **제로 다항식(zero polynomials)**
  - 기존 방식으로는 "제로 다항식"에 별도케이스를 만들어 처리
    - 제로다항식인지 검사, 입력 삭제에 대해 제로다항식인지 검사 등
  - 헤더 노드(header node) 사용하여 다항식 처리
    - 제로 다항식에 대한 특별한 경우를 만들지 않기 위해 각 다항식은 헤더 노드를 갖도록 함

기존
다항식 표현

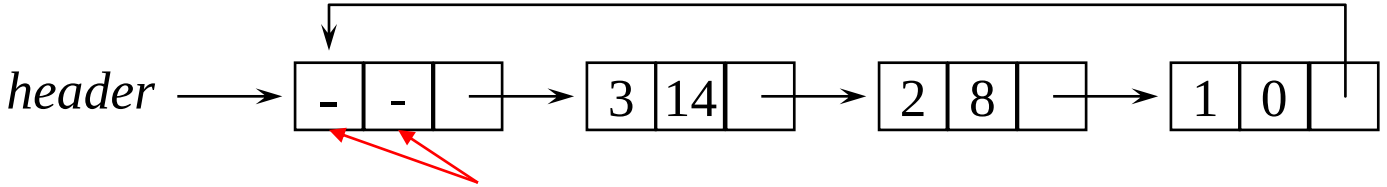


제로 다항식, 일반다항식
모두 header를 가짐



자기자신을 포인팅

(a) 제로 다항식



coef, expon
필드는 의미가 없다

(b) $3x^{14} + 2x^{8} + 1$

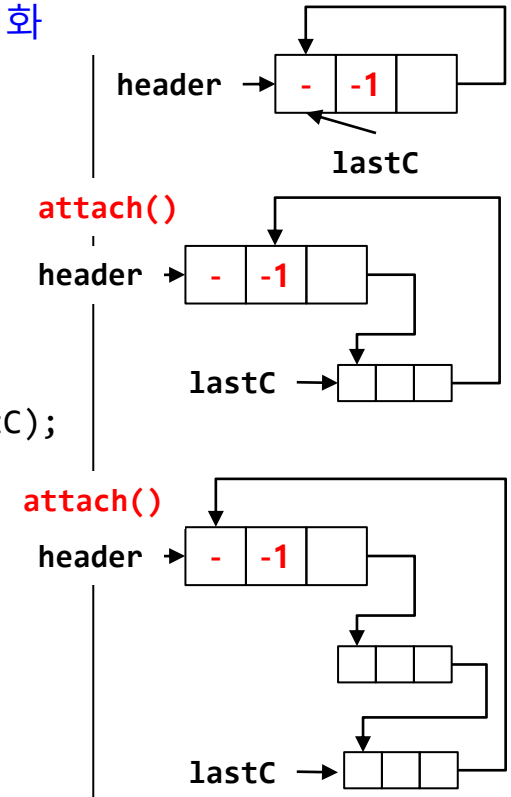
다항식(Polynomials)

cpadd (헤더를 가진 다항식의 덧셈)

```
polyPointer cpadd(polyPointer a, polyPointer b)
{
     polyPointer startA, c, lastC;
     int sum, done = FALSE;
     startA = a;
     a = a->link;     b = b->link;    c = getNode();
     c->expon = -1; lastC = c;
     do {
          switch(COMPARE(a->expon, b->expon)) {
               case -1: /* a->expon < b->expon */
                        attach(b->coef, b->expon, &lastC);
                        b = b->link; break;
               case 0: /* a->expon = b->expon */
                        if(startA == a) done = TRUE;
                        else {
                             sum = a->coef + b->coef;
                             if(sum) attach(sum, a->expon, &lastC);
                             a = a->link; b = b->link;
                        }
                        break;
               case 1: /* a->expon > b->expon */
                        attach(a->ceof, a->expon, &lastC);
                        a = a->link;
          }
      }while(!done);
      lastC->link = c;
      return c;
}
```

/* 다항식 a와 b는 헤더 노드를 가진 단순 연결 원형 리스트이고, a와 b가 합산된 다항식을 반환한다. */

← header node 초기화



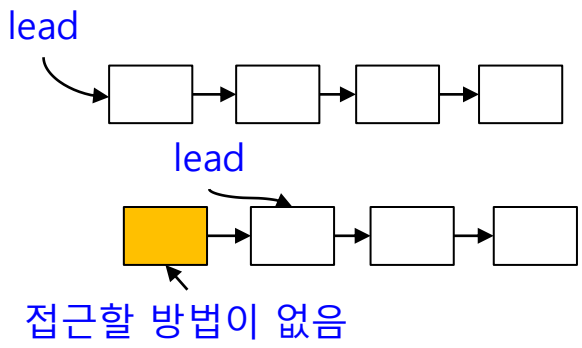
추가 리스트 연산
(Additional List Operations)

체인 연산(Operations for Chains)

가용 공간(available space) 리스트
탐색공간 (search space)

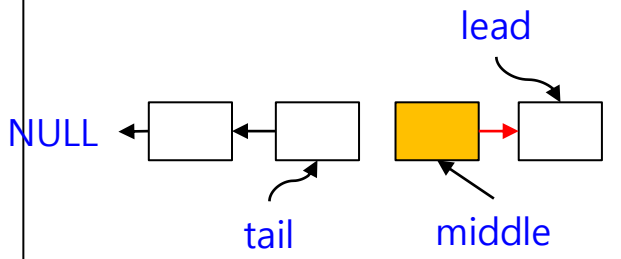
- **체인을 역순으로 만드는 (inverting, reversing) 연산**

```
typedef struct listNode *listPointer;
typedef struct listNode {
        char data;
        listPointer link;
        };
```




```
listPointer invert(listPointer lead)
{ /* lead가 가리키고 있는 리스트를 역순으로 만든다. */
    listPointer middle, trail;
    middle = NULL;
    while (lead) {
         trail = middle;
         middle = lead;
         lead = lead->link;
         middle->link = trail;
    }
   return middle;
}
```

trail을 middle쪽으로
middle을 lead쪽으로
lead를 다음링크로 이동



Time Complexity = O(length)
length = 리스트의 길이

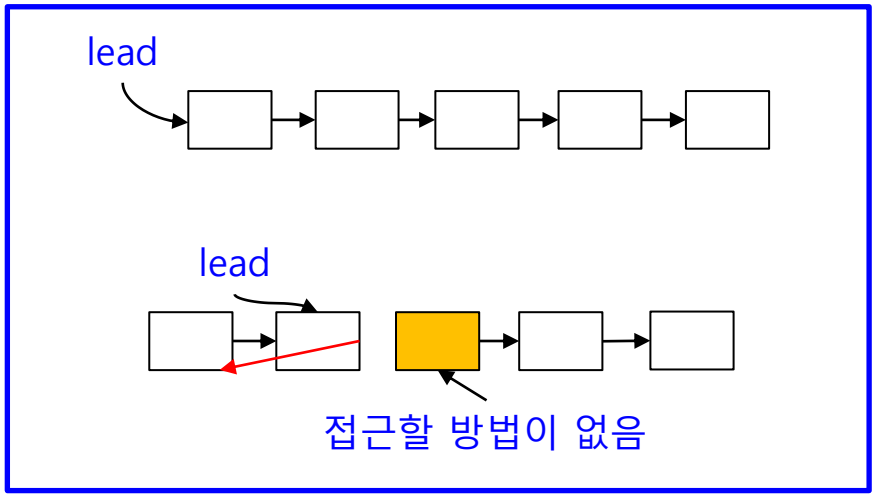
체인 연산(Operations for Chains)

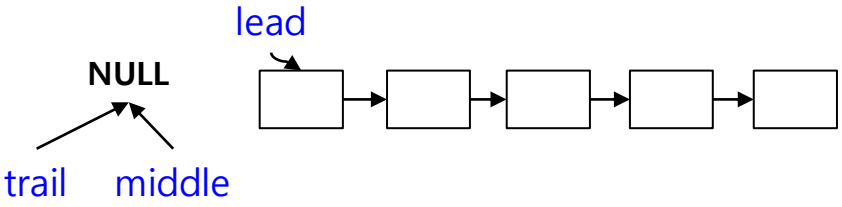
```
while (lead) {
        trail = middle;
        middle = lead;
        lead = lead->link;
        middle->link = trail;
    }
```

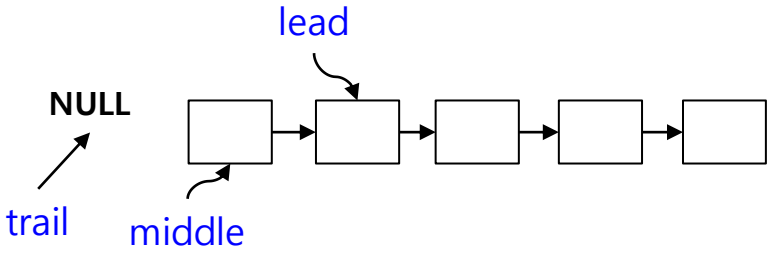
3개의 포인터를 적절히 이용하여 제자리(in place)에서 문제를 해결

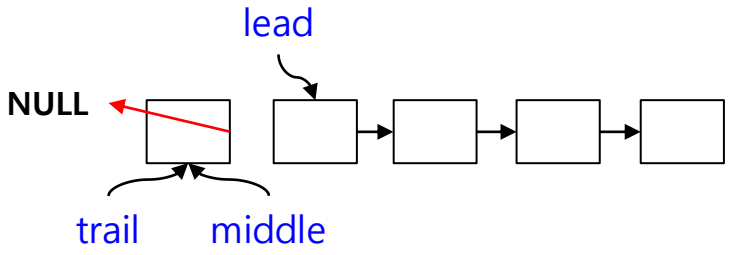
trail middle lead

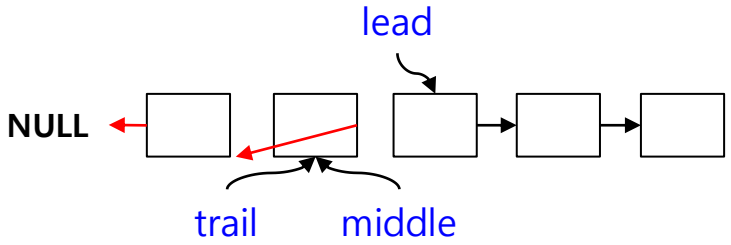
문제점









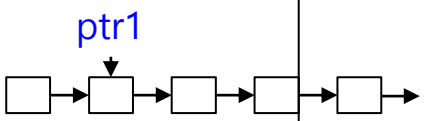


체인 연산(Operations for Chains)

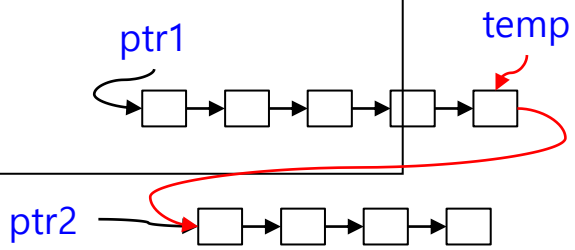
- 두 개의 체인 ptr1과 ptr2를 연결(concatenation)하는 연산

```
listPointer concatenate(listPointer ptr1, listPointer ptr2)
{
        /* 리스트 ptr1 뒤에 리스트 ptr2가 연결된 새로운 리스트를 생성한다.
        ptr1이 가리키는 리스트는 영구히 바뀐다. */

        listPointer temp;


         /* check for empty lists */
         if(!ptr1) return ptr2;
         if(!ptr2) return ptr1;

         /* neither list is empty, find end of first list */
         for(temp = ptr1; temp->link; temp = temp->link) ;

         /* link end of first to start of second */
         temp->link = ptr2;
}
```

← ptr1을 직접 핸들링하면 앞쪽으로 이동 불가



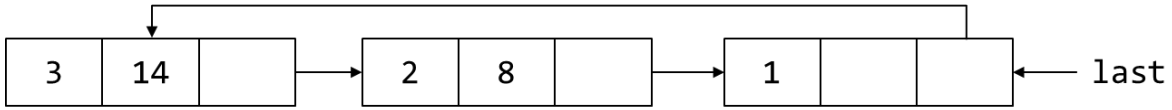
← 둘 중에 하나라도 NULL이면, 다른 리스트 리턴



원형 연결 리스트 연산
(Operations for Circularly Linked Lists)

왜 last는 이중 포인터인가?
주의: last@insertFront

- **원형 연결 리스트에 삽입**
  - 리스트의 마지막 노드에 대한 포인터 last를 유지함으로써 앞이나 뒤 양쪽에 쉽게 원소를 삽입 가능

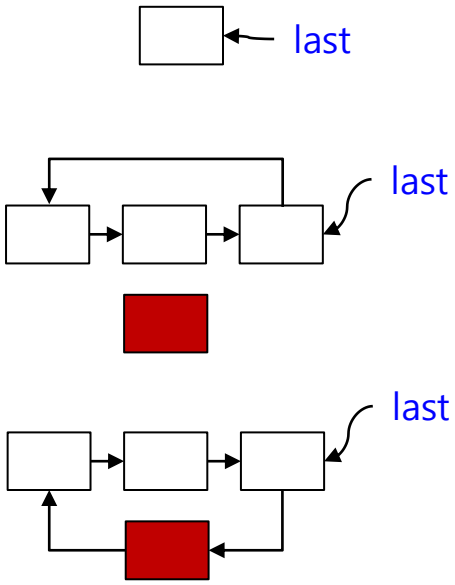


/* 리스트의 마지막 노드가 last인 원형 리스트의 앞에 노드 삽입 */

```
void insertFront(listPointer *last, listPointer node)
{
        if (IS_EMPTY(*first)) {
        /* 리스트가 공백일 경우, last이 새로운 항목을 가리킴 */
              *last = node;
               node->link = node;
        }
        else {
       /* 공백이 아니면, 리스트의 앞에 새로운 항목을 삽입 */
               node->link = (*last)->link;
               (*last)->link = node;
       }
}
```

첫 번 째 노드를 가리키는 first 사용한 다면 리스트 전체를 이동한 후 마지막 노드의 포인터 변경

공백이었을 때



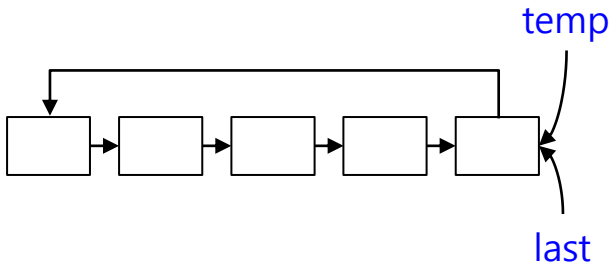
가장 앞쪽에 삽입한 효과

원형 연결 리스트 연산
(Operations for Circularly Linked Lists)

왜 last는 싱글 포인터인가?
주의: last@length

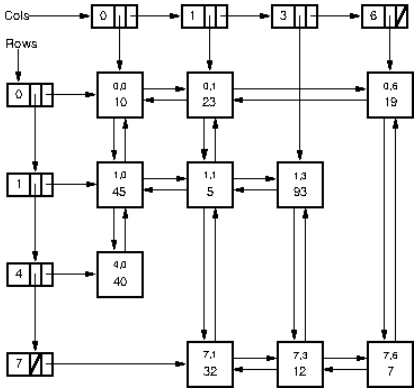
- 원형 연결 리스트의 길이 계산

```
int length(listPointer last)
{
    /* 원형 리스트 last의 길이를 계산한다. */
      listPointer temp;
      int count = 0;

      if (last) {
             temp = last;

            do {
                  count++;
                  temp = temp->link;
            } while (temp != last);
      }

      return count;
}
```



희소행렬
(Sparse Matrices)

In numerical analysis and scientific computing, a sparse matrix or sparse array is a matrix in which most of the elements are zero. By contrast, if most of the elements are nonzero, then the matrix is considered dense.

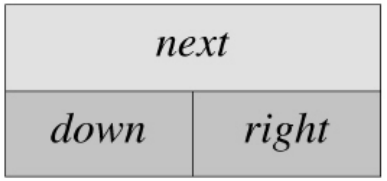


https://en.wikipedia.org/wiki/Sparse_matrix
https://www.ida.liu.se/opendsa/Books/TDDD86S19/html/Sparse.html

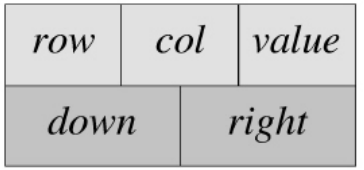
희소행렬 표현(Sparse Matrix Representation)

- **희소 행렬의 연결 리스트 표현**
  - 희소 행렬의 각 열과 행을 헤더 노드가 있는 원형 연결 리스트로 표현
  - 각 노드에는 헤더노드와 엔트리 노드를 나타내기 위한 tag 필드가 있음

- **헤더 노드(header node)**
  - down : 열 리스트로 연결하는데 사용
  - right : 행 리스트로 연결하는데 사용
  - next : 헤드 노드들을 서로 연결하는데 사용
  - 헤더 노드의 총 수 : max{행의 수, 열의 수}



- **엔트리 노드(entry node)**
  - tag필드 외에 5개의 필드(row, col, down, right, value)가 있음
  - down: 같은 열에 있는 0이 아닌 다음 항 연결
  - right : 같은 행에 있는 0이 아닌 다음 항 연결



$a_{ij} \neq 0$ → tag = entry, value=a_{ij}, row=i, col=j

희소행렬 표현(Sparse Matrix Representation)

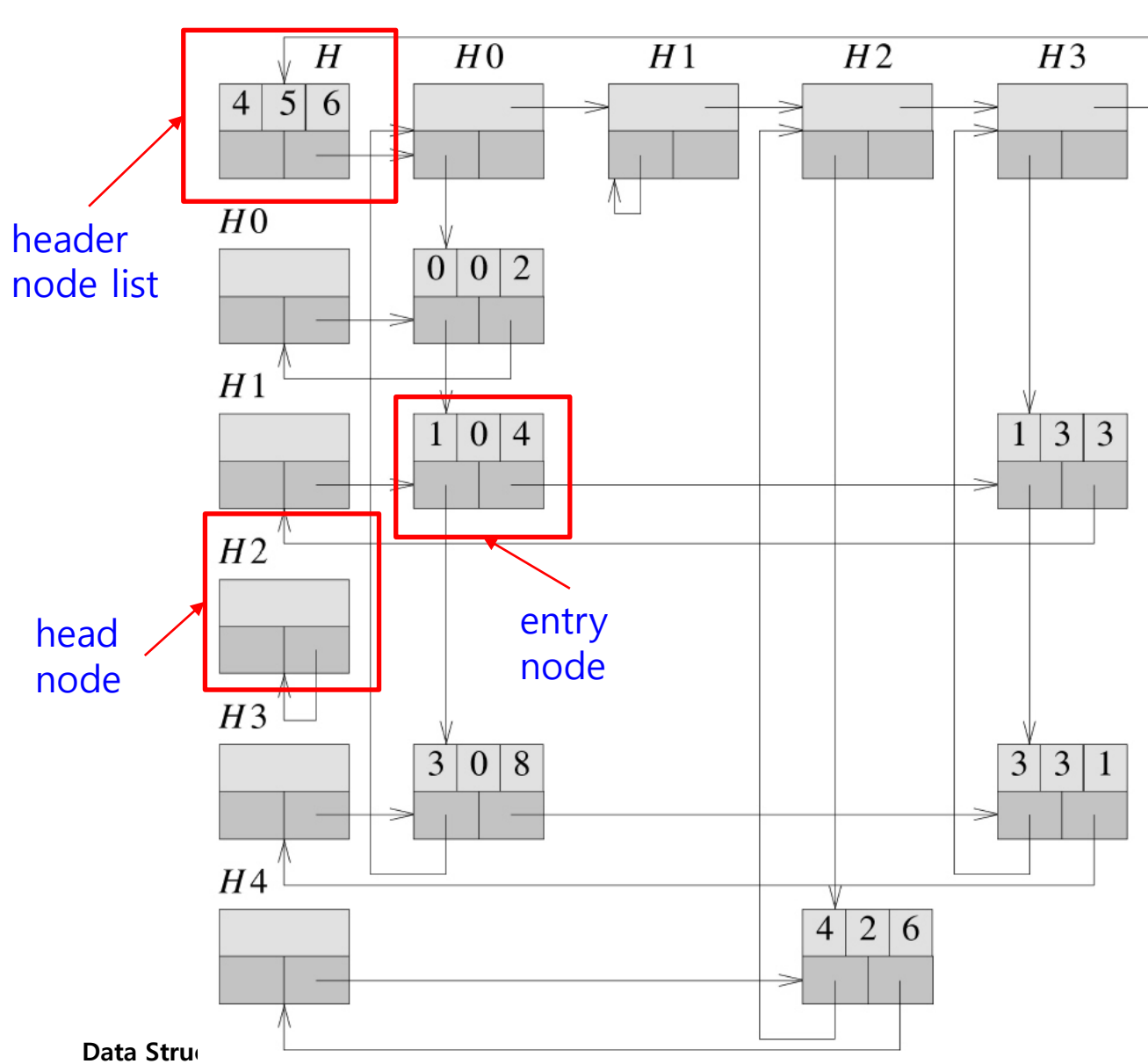
- 5*4 희소 행렬 a

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

- 필요한 노드 수
  - numTerms개의 0이 아닌 항을 가진 numRows * numCols 희소 행렬 이라면 max{numRows, numCols} + numTerms + 1개

header node　　entry node　　header node list

희소행렬 표현(Sparse Matrix Representation)



5번째 열 X

max{numRows, numCols}

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

희소 행렬 a의 연결 표현
(노드의 tag필드는 생략)

희소행렬 표현(Sparse Matrix Representation)

- 희소 행렬의 표현을 위한 자료 구조 정의

```
#define MAX_SIZE 50    /* 최대 행렬 크기 */
typedef enum {head,entry} tagfield;
typedef struct matrixNode *matrixPointer;
typedef struct entryNode {
        int row;
        int col;
        int value;
        };
typedef struct matrixNode {
        matrixPointer down;
        matrixPointer right;
        tagfield tag;
        union {
               matrixPointer next;
               entryNode entry;
              } u;
        };
matrixPointer hdnode[MAX_SIZE];
```

← head 인지 entry인지 나타내는 플래그(flag)

← 상이한 두 종류의 노드를 하나의 matrixNode로 관리하기 위해

희소행렬 입력(Sparse Matrix Input)

- **희소 행렬 입력 (1)**
  - 첫 번째 입력 라인
    - 행의 수(numRows), 열의 수(numCols), 0이 아닌 항의 수(numTerms)
  - 다음 numTerms개의 입력 라인
    - row, col, value의 형태로 구성되었다고 가정
    - 행 우선으로, 행 내에서는 열 우선으로 정렬되어 있다고 가정

	[0]	[1]	[2]
[0]	5	4	6
[1]	0	0	2
[2]	1	0	4
[3]	1	3	3
[4]	3	0	8
[5]	3	3	1
[6]	4	2	6

희소 행렬의 입력 예

희소행렬 입력(Sparse Matrix Input)

- **입력을 위해 보조 배열 hdnode를 사용**
  - 적어도 입력될 행렬의 가장 큰 차원의 크기라고 가정
  - 변수 hdnode[i]: 열 i와 행 i에 대한 헤더노드를 가리키는 포인터
    - 입력 행렬을 구성하는 동안 임의의 열을 효과적으로 접근할 수 있게 함

```
matrixPointer mread(void)
{ /* 행렬을 읽어 연결 표현으로 구성한다. 전역 보조 배열 hdnode가 사용된다. */
   int numRows, numCols, numTerms, numHeads, i;
   int row, col, value, currentRow;
   matrixPointer temp,last,node;

   printf("Enter the number of rows, columns, and number of nonzero terms: "
   scanf("%d%d%d", &numRows, &numCols, &numTerms);
   numHeads = (numCols > numRows) ? numCols : numRows;          ← max{numRow, numCols}
   /* 헤더 노드 리스트에 대한 헤더 노드를 생성한다. */
   node = newNode(); node->tag = entry;
   node->u.entry.row = numRows;                                 ← header node list node
   node->u.entry.col = numCols;
   if (!numHeads) node->right = node;
   else { /* 헤더 노드들을 초기화한다. */
        for (i = 0; i < numHeads; i++) {                        ← O(max{numRow, numCols})
            temp = newNode;
            hdnode[i] = temp; hdnode[i]->tag = head;
            hdnode[i]->right = temp; hdnode[i]->u.next = temp;
       }
```

```
      currentRow = 0;
      last = hdnode[0]; /* 현재 행의 마지막 노드 */
      for (i = 0; i < numTerms; i++) {
         printf("Enter row, column and value: ");
         scanf("%d%d%d", &row,&col,&value);
         if (row > currentRow) { /* 현재 행을 종료함 */
            last->right = hdnode[currentRow];
            currentRow = row; last = hdnode[row];
         }
         temp = newNode();
         temp->tag = entry;  temp->u.entry.row = row;
         temp->u.entry.col = col;
         temp->u.entry.value = value;
         last->right = temp; /* 행 리스트에 연결 */
         last = temp;
         /* 열 리스트에 연결 */
         hdnode[col]->u.next->down = temp;
         hdnode[col]->u.next = temp;
      }
      /* 마지막 행을 종료함 */
      last->right = hdnode[currentRow];
      /* 모든 열 리스트를 종료함 */
      for (i = 0; i < numCols; i++)
         hdnode[i]->u.next->down = hdnode[i];
      /* 모든 헤더 노드들을 연결함 */
      for (i = 0; i < numHeads-1; i++)
         hdnode[i]->u.next = hdnode[i+1];
      hdnode[numHeads-1]->u.next = node;
      node->right = hdnode[0];
   }
   return node;
}
```

← O(numTerms)

← O(numCols)

← O(max{numRow, numCols})

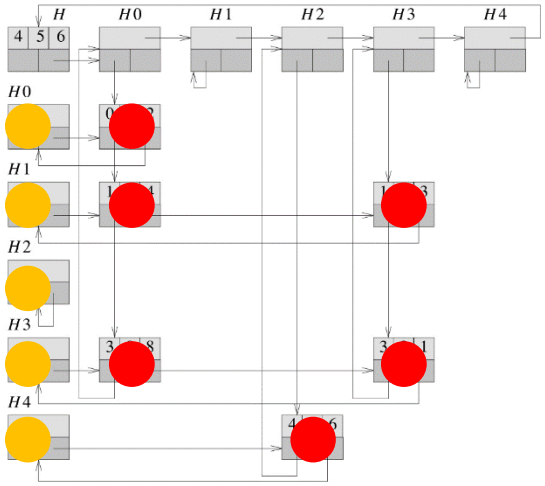
Time Complexity
O(max{numRow, numCols}) + O(numTerms)
+ O(numCols) + O(max{numRow, numCols})
= O(numRow + numCols + numTerms)

2차원 배열
O(numRows x numCols)

순차적 방법
O(numTerms)

각 row에 대해 entry node 출력

희소행렬 출력(Sparse Matrix Output)



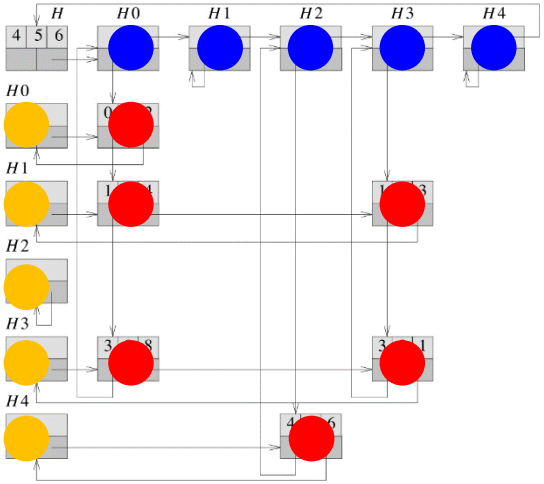

```
void mwrite(matrixPointer *node)
{ /* 행렬을 행 우선으로 출력한다. */
   int i;
   matrixPointer temp, head = node->right;

   /* 행렬의 차원 */
   printf("\n numRows = %d, numCols = %d \n",
                node->u.entry.row, node->u.entry.col);
   printf(" The matrix by row, column, and value: \n\n");

   for (i = 0; i < node->u.entry.row; i++) {
   /* 각 행에 있는 엔트리들을 출력 */
      for (temp = head->right; temp != head; temp = temp->right)
         printf("%5d%5d%5d \n", temp->u.entry.row,
                temp->u.entry.col, temp->u.entry.value);
      head - head->u.next;  /* 다음 행 */
   }
}
```

Time Complexity = O(numRows + numTerms)

희소행렬 삭제(Erasing a Sparse Matrix)

- 함수 free를 사용하여 한번에 한 노드씩 반환

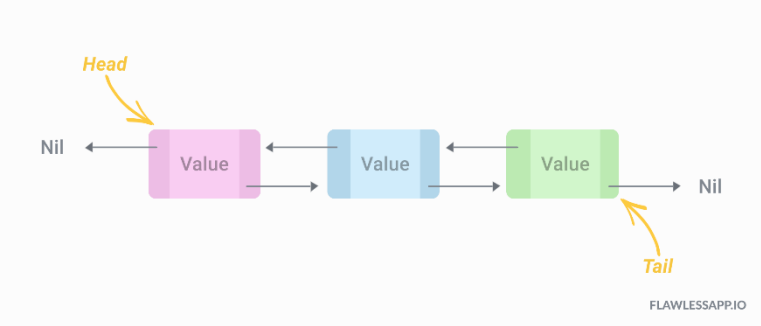
```
void merase(matrixPointer *node)
{
   /* 행렬을 삭제하고, 노드들을 히프로 반환한다. */
   matrixPointer x,y, head = (*node)->right;
   int i, numHeads;
   /* 엔트리 노드와 헤더 노드들을 행 우선으로 반환한다. */
   for (i = 0; i < (*node)->u.entry.row; i++) {
      y = head->right;
      while (y != head) {
         x = y;  y = y->right; free(x);
      }
      x = head; head = head->u.next; free(x);
   }
   /* 나머지 헤더 노드들을 반환한다. */
   y = head;
   while (y != *node) {
      x = y; y = y->u.next; free(x);
   }
   free(*node); *node = NULL;
}
```



Time Complexity = O(numRows + numCols + numTerms)

이중 연결 리스트
(Doubly Linked List)

In computer science, a doubly linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains three fields: two link fields (references to the previous and to the next node in the sequence of nodes) and one data field.

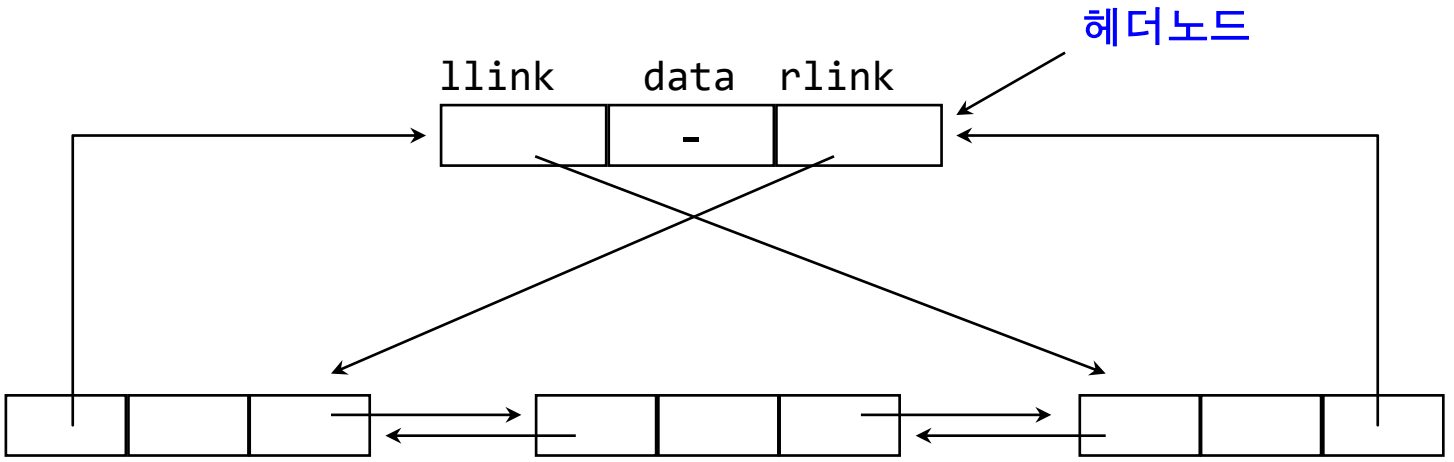


https://en.wikipedia.org/wiki/Doubly_linked_list

이중 연결리스트(Doubly Linked Lists)

- **이중 연결 리스트(doubly linked list)**
  - 포인터를 양방향으로 이동해야 할 필요가 있을 때
  - 임의의 노드를 삭제해야 되는 문제에 적합
  - 각 노드는 전방과 후방을 가리키는 두 개의 링크를 가짐

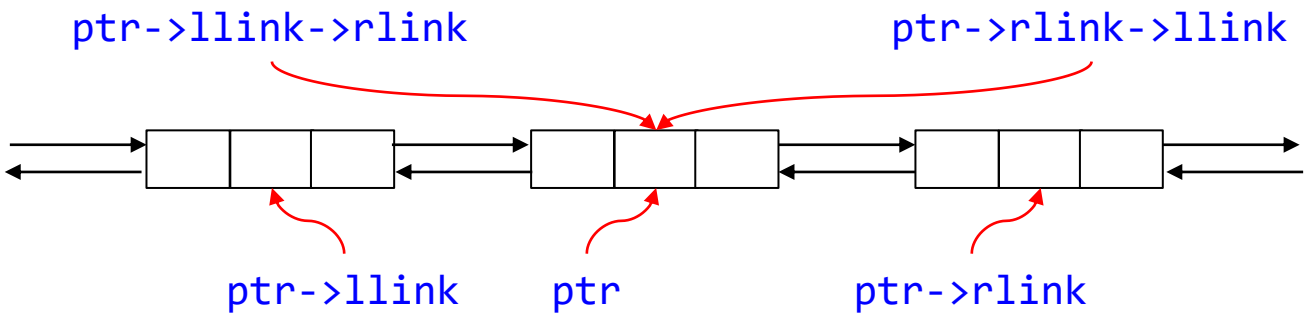
```
typedef struct node *nodePointer;
typedef struct node {
            nodePointer llink;
            element data;
            nodePointer rlink;
            }
```

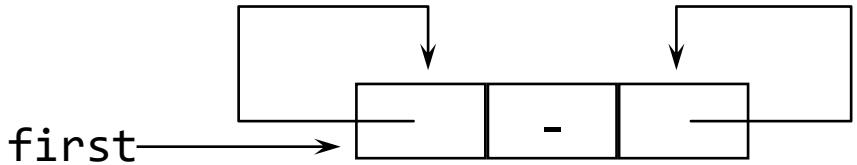


헤더 노드가 있는 이중 연결 원형 리스트

이중 연결리스트(Doubly Linked Lists)

- 이중 연결 리스트에서 ptr이 임의의 노드를 가리키고 있다면
  - ptr = ptr->llink->rlink = ptr->rlink->llink 성립
  - 전위 노드나 후위 노드로 쉽게 이동할 수 있음



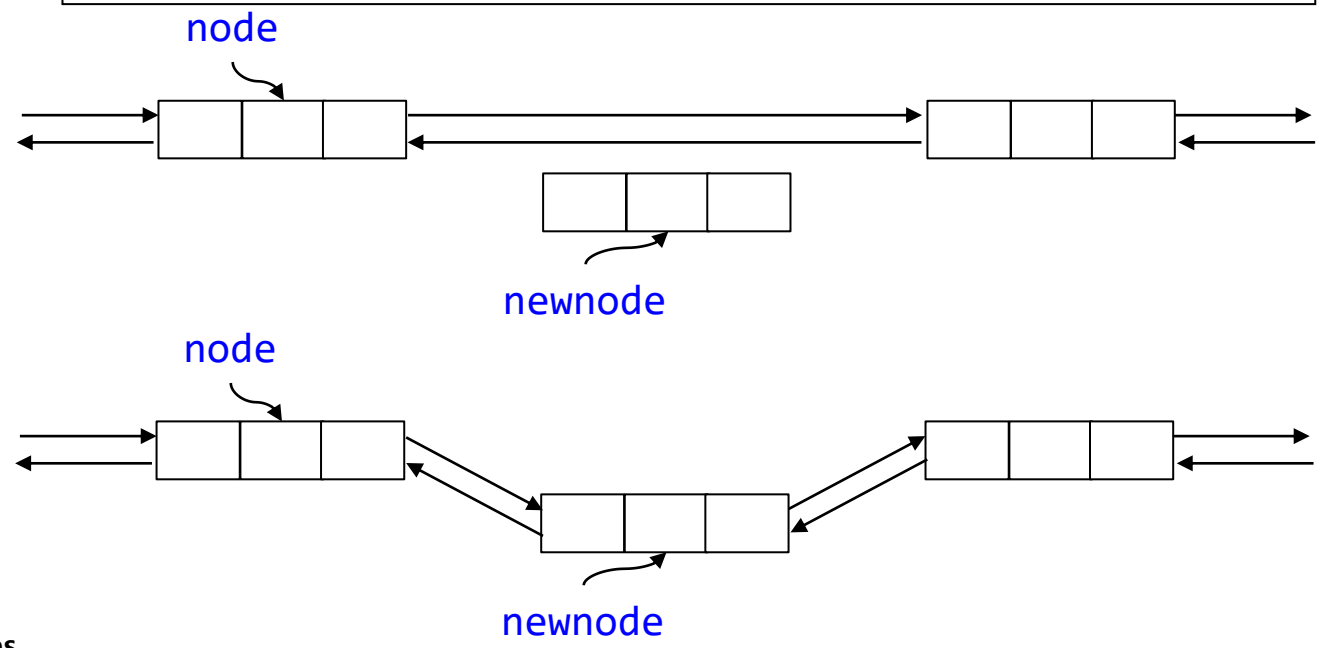


헤더 노드를 가진 공백 이중 연결 원형 리스트

이중 연결리스트(Doubly Linked Lists)

- 이중 연결 원형 리스트에 삽입

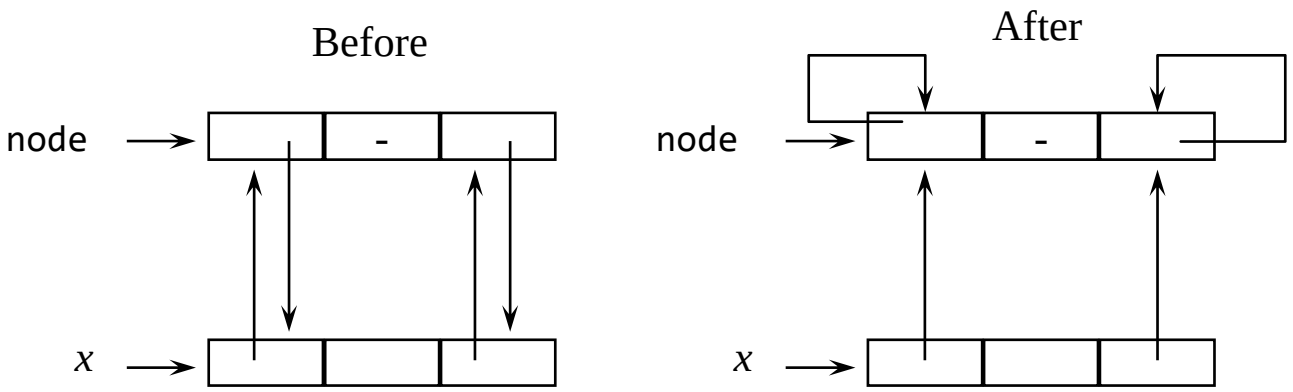
```
void dinsert(nodePointer node, nodePointer newnode)
{
   /* newnode를 node의 오른쪽에 삽입 */
   newnode->llink = node;
   newnode->rlink = node->rlink;
   node->rlink->llink = newnode;
   node->rlink = newnode;
}
```



이중 연결리스트(Doubly Linked Lists)

- 이중 연결 원형 리스트로부터 삭제

```
void ddelete(nodePointer node, nodePointer deleted)
{
   /* 이중 연결 리스트에서 삭제 */
   if (node == deleted)
      printf("Deletion of head node not permitted.\n");
   else {
      deleted->llink->rlink = deleted->rlink;
      deleted->rlink->llink = deleted->link;
      free(deleted);
   }
}
```



Additional Slides

동치부류(Equivalence Classes)

- **동치 관계(equivalence relation), ≡의 특성**
  - 반사적(reflexive) : 어떤 다각형 x에 대해서, x≡x가 성립
  - 대칭적(symmetric) : 어떤 두 다각형 x, y에 대해서, x≡y이면 y≡x
  - 이행적(transitive): 어떤 세 다각형 x, y, z에 대해서, x≡y이고 y≡z이면 x≡z

- **동치 관계(equivalence relation)의 정의**
  - 집합 S에 대하여 관계 ≡가 대칭적, 반사적, 이행적이면 관계 ≡를 집합 S에 대해 동치 관계라 함. 그 역도 성립.

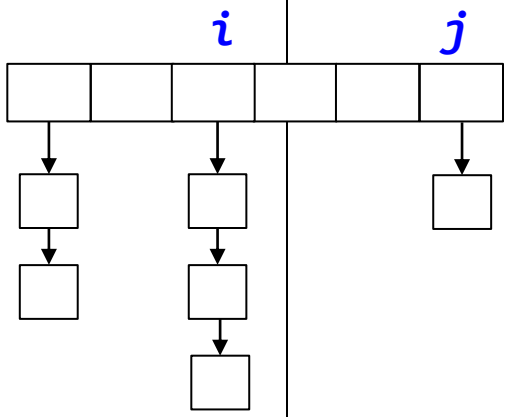
- **동치부류(equivalence class)**
  - S의 두 원소 x, y에 대하여 x≡y이면 x, y는 같은 동치부류 (역도 성립)
  - ex) 0≡4, 3≡1, 6≡10, 8≡9, 7≡4, 6≡8, 3≡5, 2≡11, 11≡0
    → { 0, 2, 4, 7, 11}; {1, 3, 5}; {6, 8, 9, 10}

0≡4, 3≡1, 6≡10, 8≡9, **7≡4**, 6≡8, 3≡5, **2≡11**, **11≡0**

동치부류(Equivalence Classes)

- **동치 결정 알고리즘**
  - 동치 쌍(equivalence pair) <i,j>를 읽어 기억
  - 0에서부터 시작해서 <0,j> 형태의 모든 쌍을 찾는다. <j,k>가 있으면 k도 0과 같은 동치부류에 포함시킨다.
  - 0이 포함된 동치 부류의 모든 객체를 찾아 표시하고 출력
  - 남은 동치 부류에 대해서도 같은 방법으로 계속한다.

```
void equivalence()
{
        initialize;
        while (there are more pairs) {
                read the next pair <i,j>;
                process this pair;
        }
        initialize the output;
        do
                output a new equivalence class;
        while (not done);
}
```



동치부류(Equivalence Classes)

- **배열을 사용 한 동치 알고리즘**
  - m : 동치 쌍의 수, n : 객체 수
  - 쌍 <i, j>를 나타내기 위해 배열 pairs[i][j] 사용 (pairs[n][m])
  - 배열의 극히 일부 원소만 사용 : 기억 장소 낭비
  - 새로운 쌍 <i,k>를 i행에 넣기 위해 i행 내의 빈 공간을 찾거나 추가 공간을 사용해야 하므로 많은 시간 소모

동치부류(Equivalence Classes)

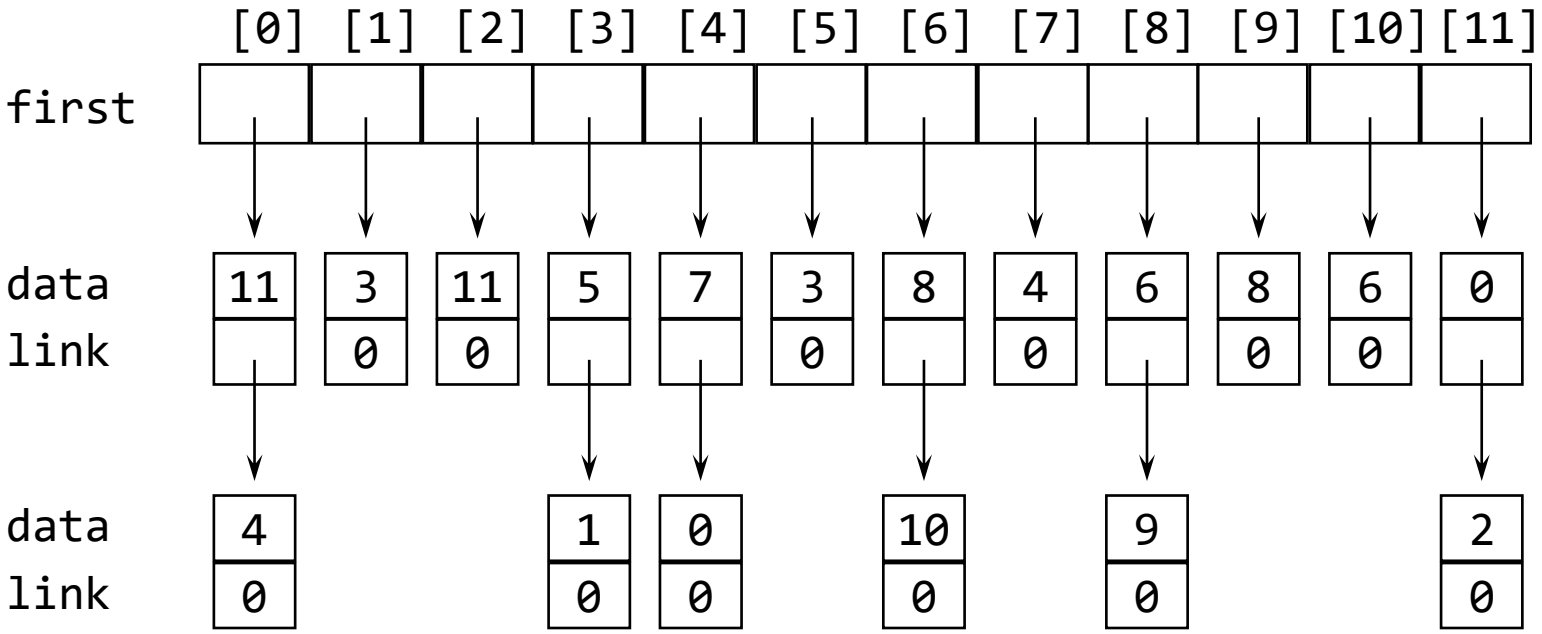
- **연결 리스트를 사용한 동치 알고리즘**
  - seq[n] : n개의 리스트 헤더 노드 저장
    - i번째 행에 대한 임의 접근을 위해 필요
  - out[n], 상수 TRUE, FALSE : 이미 출력된 객체 표시

```
void equivalence()
{
        initialize seq to NULL and out to TRUE;
        while (there are more pairs) {
               read the next pair <i,j>;
               put j on the seq[i] list;
               put i on the seq[j] list;
        }

        for (i = 0; i < n; i++)
               if (out[i]) {          ← 객체가 출력되었는지에 대한 정보
                     out[i] = FALSE;
                     output this equivalence class;
               }
}
```


좀더 구체화된 동치 알고리즘

동치부류(Equivalence Classes)

- 쌍들이 입력된 뒤의 리스트
  - 0≡4, 3≡1, 6≡10, 8≡9, 7≡4, 6≡8, 3≡5, 5≡11, 11≡0
  - while 루프가 끝난 뒤의 리스트
  - 관계 i≡j에 대해 두 개의 노드를 사용



동치부류(Equivalence Classes)

실습: Program 4.22

```
#include <stdio.h>
#inlcude <alloc.h>
#define MAX_SIZE 24
#define IS_FULL(first) (!(first))
#define FALSE 0
#define TRUE 1

typedef struct node *nodePointer;
typedef struct node {
        int data;
        nodePointer link;
        };
void main(void)
{
        short int out[MAX_SIZE];
        nodePointer seq[MAX_SIZE];
        nodePointer x,y,top;
        int i,j,n;

        printf("Enter the size (<= %d) ", MAX_SIZE);
        scanf("%d", &n);
        for (i = 0; i < n; i++) {
        /* seq와 out을 초기화 */
              out[i] = TRUE;  seq[i] = NULL;
       }
```

← Time Complexity = O(n)

```
    /* 1 단계: 동치 쌍들을 입력 */
    printf("Enter a pair of numbers (-1 -1 to quit): ");
    scanf("%d%d", &i, &j);                          ← Time Complexity = O(m+n)
    while (i >= 0) {
            MALLOC(x, sizeof(*x));
            x->data = j; x->link = seq[i]; seq[i] = x;
            MALLOC(x, sizeof(*x));
            x->data = i; x->link = seq[j]; seq[j] = x;
            printf("Enter a pair of numbers (-1 -1 to quit) : ");
            scanf("&d&d", &i, &j);
    }
    /* 2 단계: 동치 부류들을 출력 */
    for (i = 0; i < n; i++)                         ← Time Complexity = O(m+n)
          if (out[i]) {
                printf("\nNew class: %5d", i);
                out[i] = FALSE;          /* 부류들을 FALSE로 함 */
                x = seq[i]; top = NULL; /* 스택을 초기화 */
                for (;;) {
                      while (x) {      /* 리스트 처리 */
                            j = x->data;
                            if (out[j]) {
                                  printf(%5d", j);   out[j] = FALSE;
                                  y = x->link; x->link = top; top = x; x = y;
                            }
                            else x = x->link;
                      }
                      if (!top) break;
                      x = seq[top->data]; top = top->link; /* 스택에서 제거 */
                }
          }
}
```

시간 복잡도 : O(m+n)
공간 복잡도 : O(m+n)