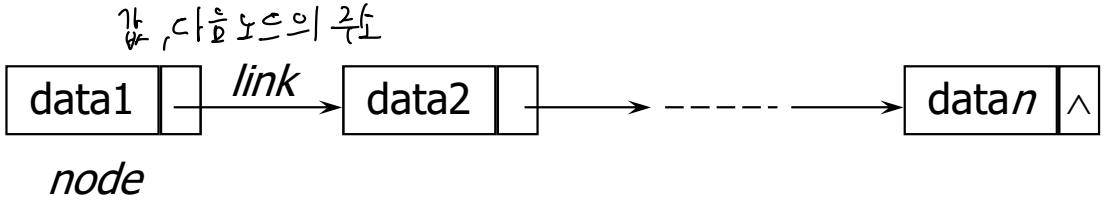
4장. 리스트(Lists)

목차

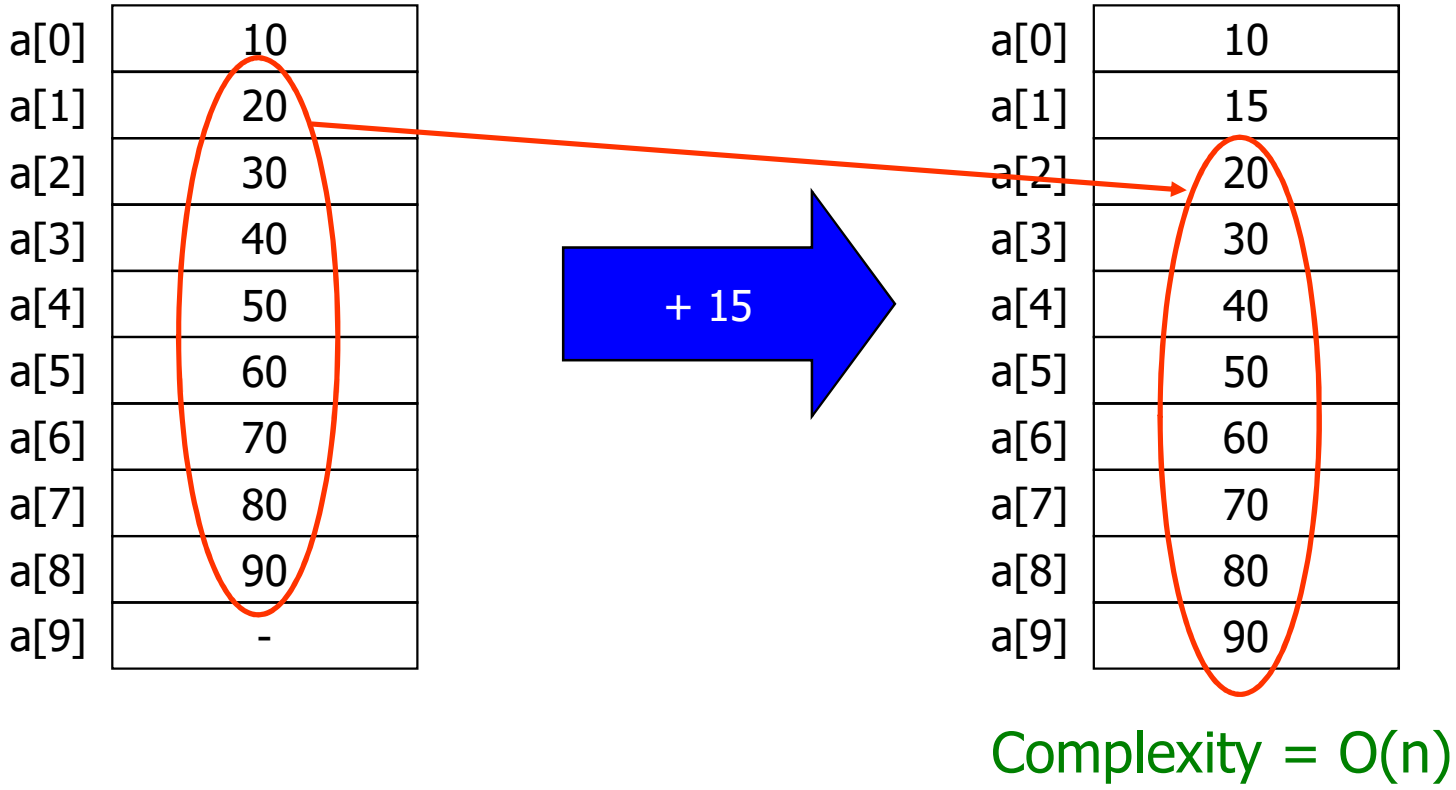
1. 포인터(Pointers)

- 순서화 리스트의 (배열을 이용한) 순차적 구현
  - 임의의 위치에 대한 삽입과 삭제 연산이 곤란(다음 장 참조)
  - 가변 길이의 순서화 리스트 지원 불가

- 순서화 리스트의 연결 리스트를 이용한 구현
  - 리스트의 연속된 항목들은 메모리의 임의의 장소에 위치
  - 리스트는 (연결된) 노드**(node)**들로 구성
    - 각 노드는 데이터와 링크**(link)**로 구성
    - 링크는 리스트의 다음 노드를 가리킴. (마지막 노드의 링크?)



Insertion at Array



Complexity = O(n)

2. 단일 연결 리스트(Singly Linked Lists)

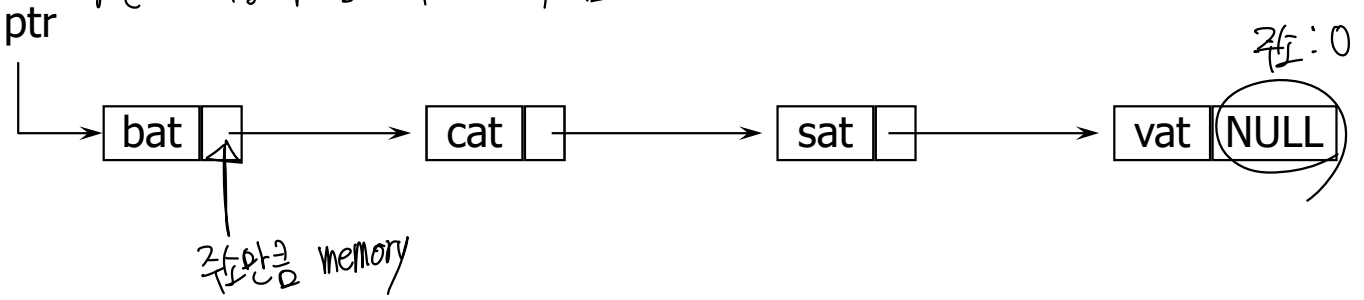
- 연결 리스트의 표현
  - 각 노드들은 메모리의 인접한 곳에 위치하지 않는다.
  - 각 노드의 주소는 프로그램 실행시 매번 틀릴 수 있다.

리스트의 이름 = 첫 번째 노드의 주소

- 삽입과 삭제
  - 기존 노드들의 위치를 변경할 필요가 없다.
  - 링크 필드를 위한 추가적인 메모리 공간 필요

data x sizeof(자료형) + link x sizeof(int)

첫번째 주소가 노드의 이름이 됨



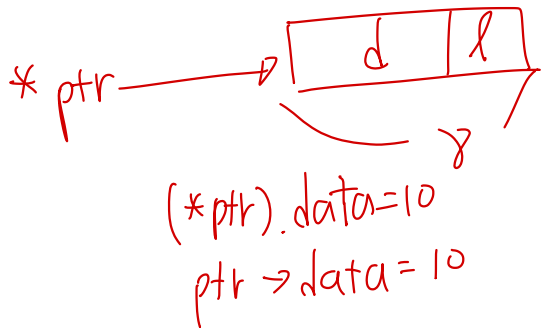
C 언어에서 리스트 구현

- 자기 참조 구조체를 이용

```
struct node {
    int data;
    struct node *link;
};
struct node *ptr = NULL;
ptr = (struct node *) malloc(sizeof(struct node));
```

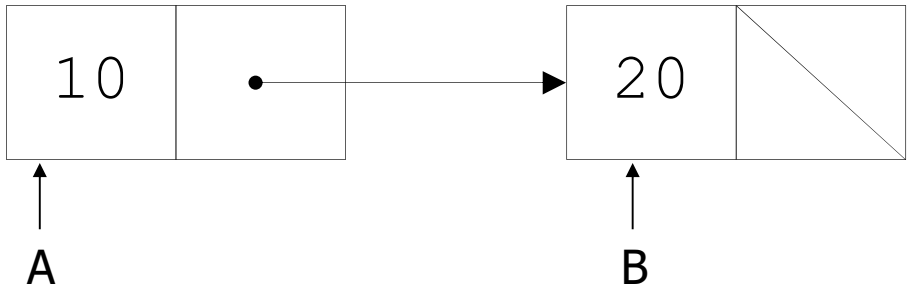
  - #define NULL ((void *) 0) // defined in stdio.h or stddef.h

- 노드의 각 필드에 값 할당
  - ptr→data = 10; // (*ptr).data = 10;
  - ptr→link = NULL;

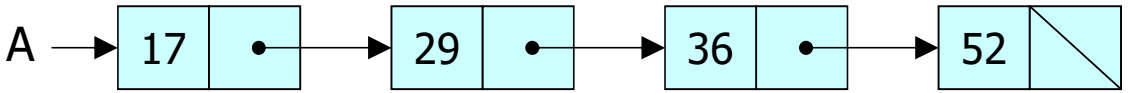


예: 두 개의 노드를 연결

```
struct node {
   int data;
   struct node *link;
};

struct node *A, *B;

A = (struct node *) malloc(sizeof(struct node));
A→data = 10;
B = (struct node *) malloc(sizeof(struct node));
B→data = 20;

A→link = B;
B→link = NULL;
```



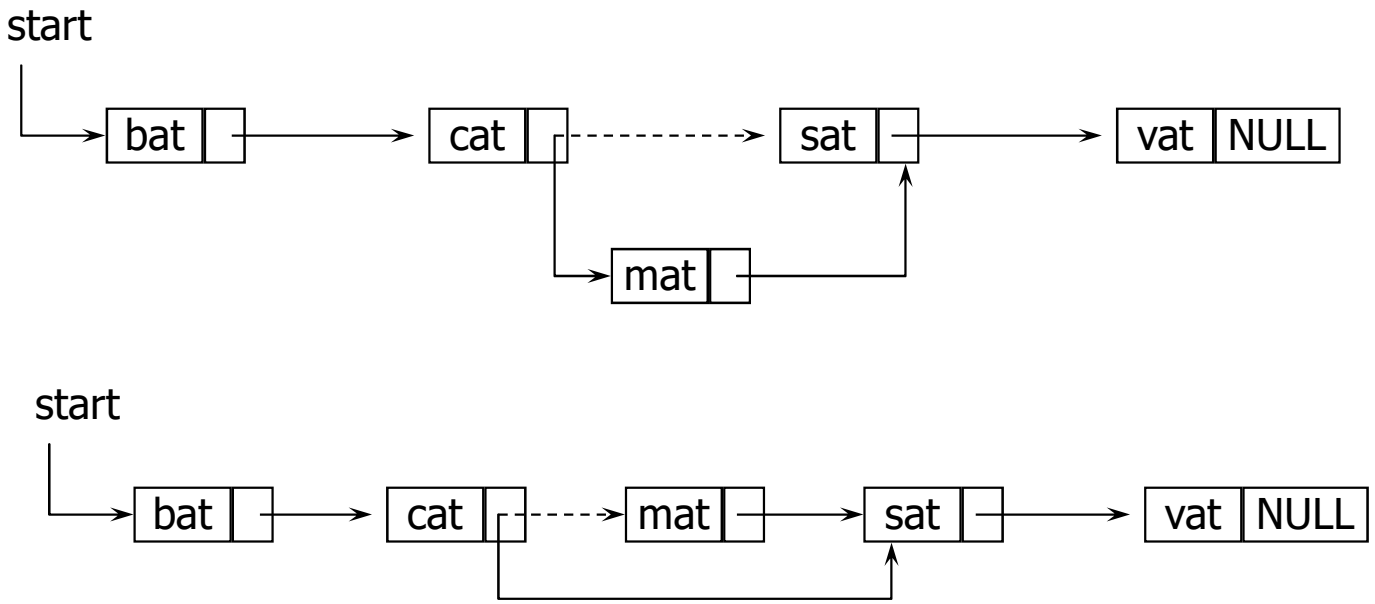
연결 리스트 연산 – 리스트 출력




```
struct node *ptr;

for (ptr = A; ptr != NULL; ptr = ptr→link)
  printf("%d ", ptr→data);
```

```
int A[4], i;  // 배열

for (i = 0; i < 4; i++)
  printf("%d ", A[i]);
```

예: 연결 리스트에서 삽입과 삭제



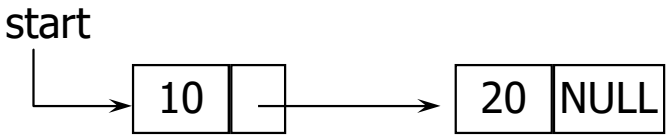
리스트에 새로운 노드를 삽입

바꾸는 순서 중요

temp->link = B->link

A->link = temp) 20의 주소가 사라짐

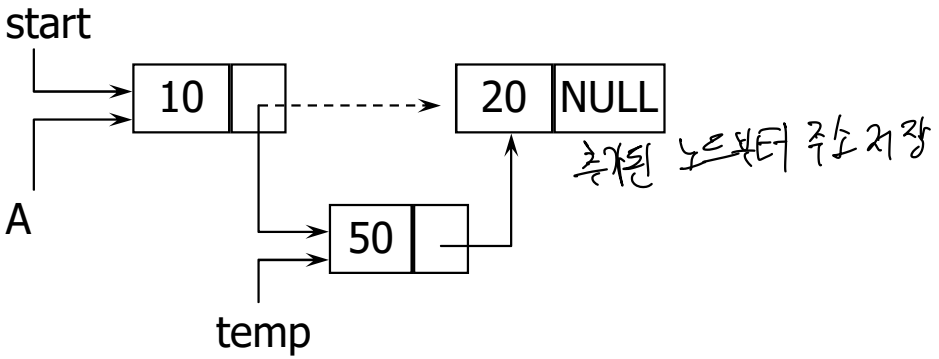
(1) 두 개의 노드로 구성된 리스트



(2) A 다음에 temp라는 새로운 노드를 삽입: insert(&start, A)

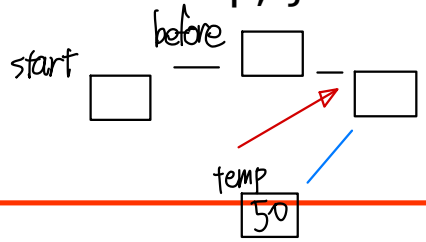
- Program 4.3

노드의 전체 주소 삽입 전 주소



Program 4.2: 리스트에 삽입

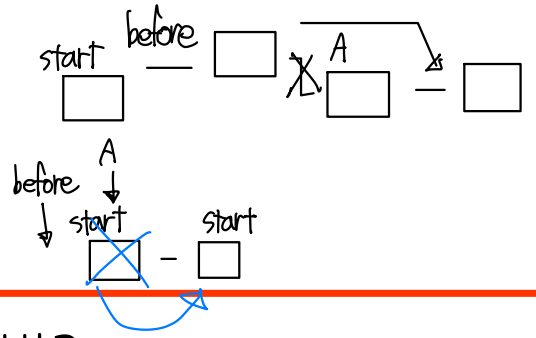
```
void insert(struct node **start, struct node *before)
{  /* data = 50인 새로운 노드를 start가 가리키는 리스트의 before 노드 다음에 삽입.
      start는 NULL일 수 있음 */

   struct node *temp;
   temp = (struct node *) malloc(sizeof(struct node));
   if (temp == NULL)
   {     fprintf(stderr, "The memory is full\n");    exit(1); }
   temp→data = 50;
   if (*start != NULL)
   {     temp→link = before→link; before→link = temp; }
   else
   {     temp→link = NULL; *start = temp; }
}
```



리스트에서 노드 삭제

- **delete(&start, before, A)**
  - start가 가리키는 리스트에서 노드 A를 삭제
  - before는 A 노드의 이전 노드를 가리키고 있음

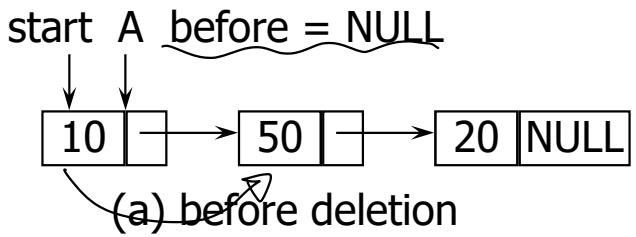
```
void delete(struct node **start, struct node *before, struct node *A)
{    // start: 시작 노드의 주소(call by reference), before와 A: 포인터
     if (before != NULL)
                before→link = A→link;
     else       *start = (*start)→link;
     free(A);
}
```

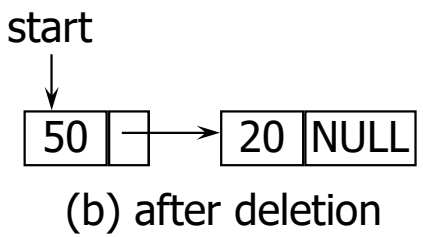


- before가 없을 경우, A를 삭제하는 방법?
- 연습문제 **4, 6, 7, 8(**페이지 **165)** 풀 것**!**

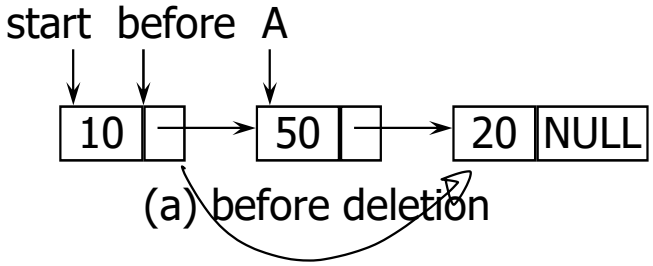
예:삭제

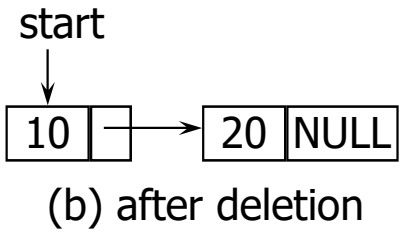
(1) delete(&start, NULL, start)





(2) delete(&start, start, start→link)

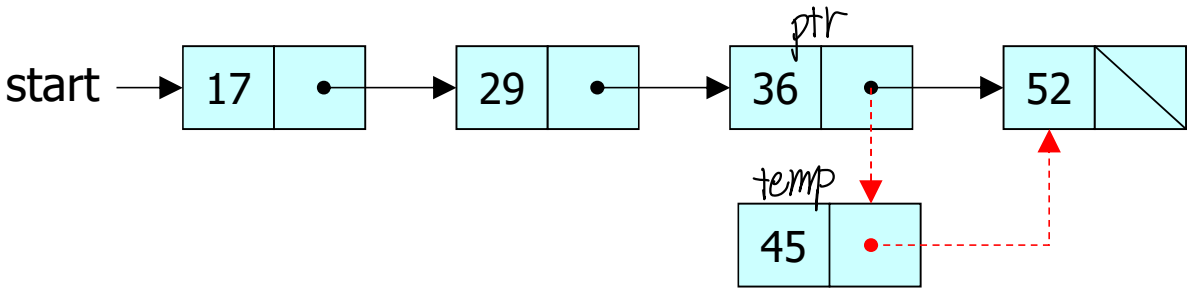




연결 리스트 연산 – 노드 추가

- **36** 다음에 **45**를 추가

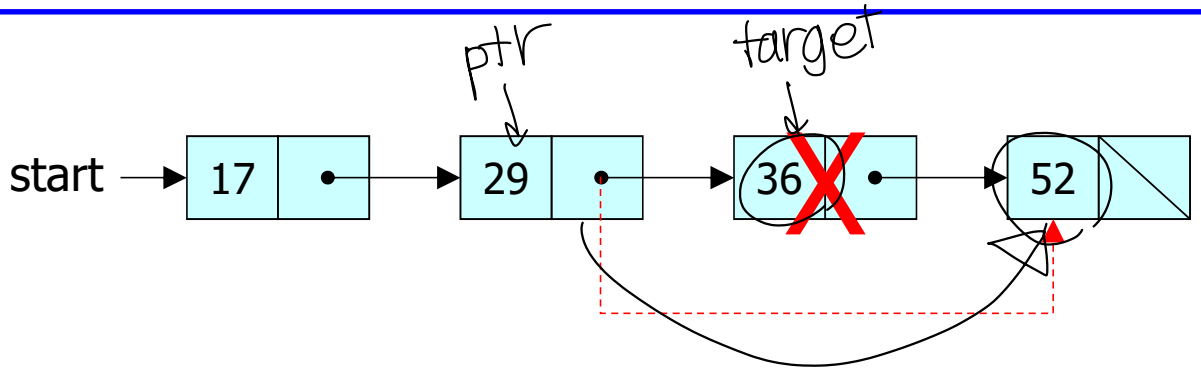
```
struct node *ptr, *temp;

temp = (struct node *) malloc(sizeof(struct node));
temp→data = 45;
for (ptr = start; ptr→data != 36; ptr = ptr→link) ;

temp→link = ptr→link;
ptr→link = temp;
```



연결 리스트 연산 – 노드 삭제

- **36** 노드를 삭제

```
struct node *ptr, *target;

for (ptr = start; ptr→link→data != 36; ptr = ptr→link) ;

target = ptr→link;
ptr→link = target→link;
free(target);
```



배열과 연결 리스트의 비교(1)

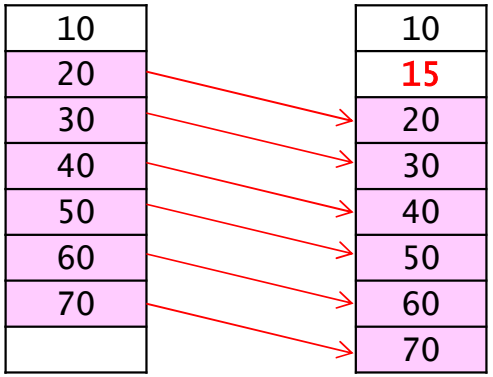
- 저장 방식의 차이
  - 배열: int A[4]; ← 메모리의 인접한 곳에 저장
    - 다음 데이터에 대한 주소를 알 필요 없음
  - 연결 리스트: struct node에 대한 네 번의 malloc
    - 각 노드들은 메모리의 여러 곳에 나누어 저장
    - link 포인터를 이용하여 다음 노드의 주소 유지

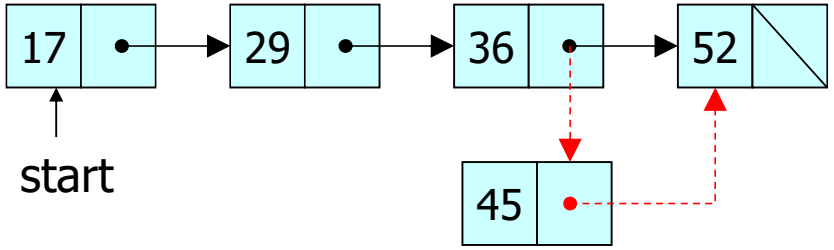
- 메모리 사용 측면
  - 저장될 데이터의 수를 안다면 배열이 효과적
  - 데이터의 수를 모를 경우, 연결 리스트가 유리
    - 새로 데이터가 입력될 때마다 malloc 실행 후 연결

realloc → 배열은 쓰레기가 생김
연결리스트는 쓰레기가 생기지X

배열과 연결 리스트의 비교(2)

- 정렬된 데이터의 유지
  - 배열:
    - 데이터가 추가될 때 기존 데이터의 위치 변경 가능
    - 이진 검색 가능
  - 연결 리스트
    - 기존 데이터의 위치 변경은 발생하지 않음
    - 이진 검색은 불가능 임의의 접근 불가능

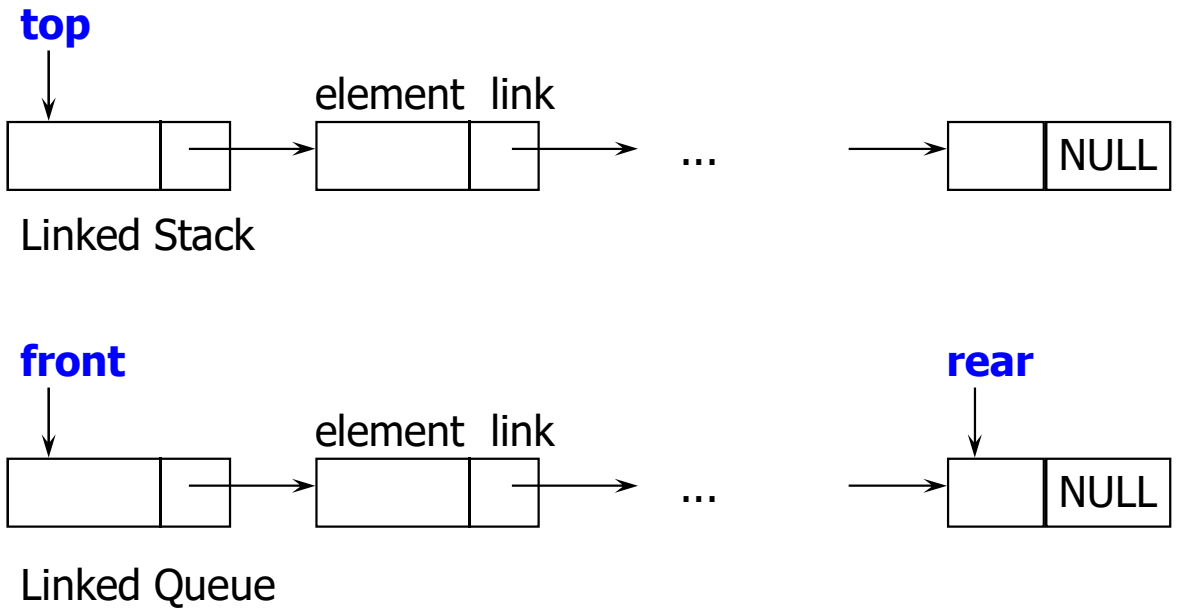




3. 리스트를 이용한 스택과 큐

- 배열을 이용한 스택과 큐의 구현 방법의 문제점
  - 메모리 낭비
  - Stack Full의 발생 가능성

- 리스트를 이용한 스택과 큐의 구현



리스트를 이용한 스택의 선언

```
typedef struct {
    int   key;
    /* other fields */
} element;
struct stack {
    element data;
    struct stack *link;
};

struct stack *top;
```

(1) 스택의 초기 조건:

top = NULL;

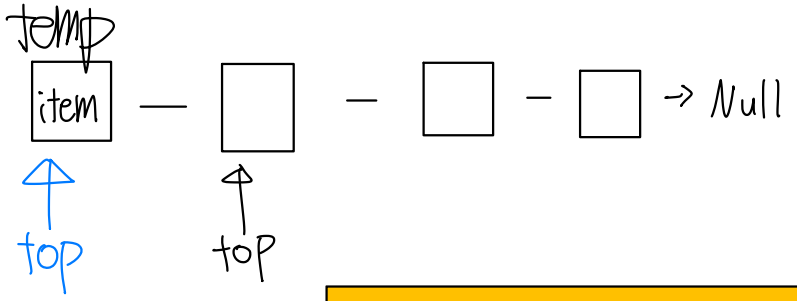
(2) 경계 조건:

top = NULL if stack is empty.

IS_FULL(temp) if the memory is full.

Program 4.5: 리스트 스택에 노드 추가

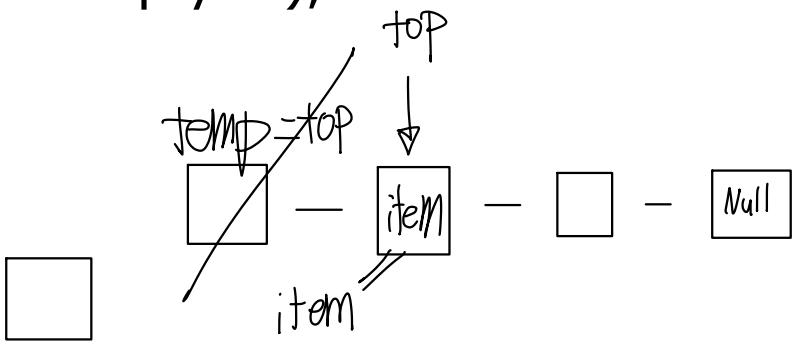
```
void push(element  item)
{
   // 스택 top에 새로운 item 추가
   struct stack *temp  =
          (struct stack *) malloc(sizeof(struct stack));
   if (temp == NULL)  {     // memory full
        fprintf(stderr, "The memory is full₩n");
        exit(1);
   }
   temp→data =  item;
   temp→link  =  top;
   top = temp;
}
```



리스트의 제일 뒤에 item을 추가할 경우?

Program 4.6: 리스트 스택에서 삭제

```
element pop()
{
   // 스택의 top이 가리키는 element를 삭제하여 return
   struct stack *temp = top;
   element  item;
   if (temp == NULL)   {    // top == NULL
         fprintf(stderr, "The stack is empty\n");
         exit(1);
   }
   item = temp→data;
   top = temp→link;
   free(temp);
   return item;
}
```



리스트를 이용한 큐의 선언

```
typedef struct {
    int   key;
    /* other fields */
} element;

struct Que {
    element data;
    struct Que *link;
};

struct Que *front, *rear;
```

큐를 위한 초기 설정:

front = NULL; rear의 초기화는 필요X

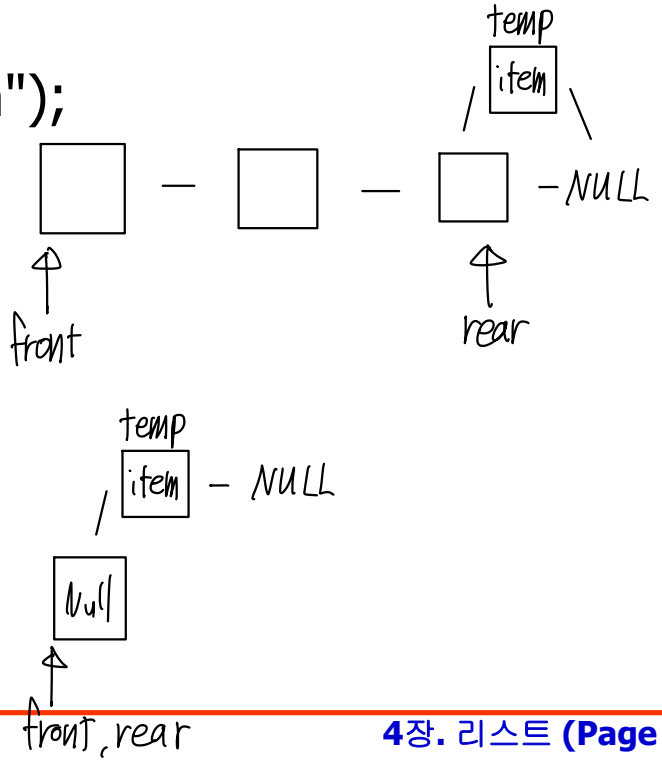
경계 조건:

front = NULL if queue is empty.

IS_FULL(temp) if the memory is full.

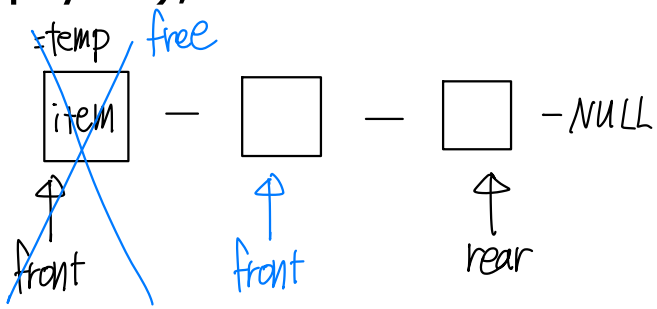
Program 4.7: 리스트 큐에 노드 추가

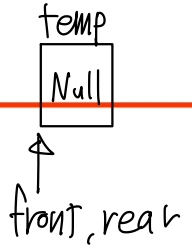
```
void addq(element item)
{
   // 큐의 rear에 새로운 element 추가
   struct Que *temp = (struct Que *) malloc(sizeof(struct Que));
   if (temp == NULL)  {   // memory full
         fprintf(stderr, "The memory is full\n");
         exit(1);
   }
   temp→data =  item;
   temp→link  =  NULL;
   if (front)   rear→link  =  temp;
   else front = temp;
   rear = temp;
}
```



Program 4.8: 리스트 큐에서 삭제

```
element deleteq()
{
    // 큐에서 front가 가리키는 노드 삭제 후 element는 return
    struct Que *temp = front;
    element item;
    if (temp == NULL)   {    // *front == NULL
          fprintf(stderr, "The queue is empty\n");
          exit(1);
    }
    item = temp→data;
    front = temp→link;
    free(temp);
    return item;
}
```





4. 다항식(Polynomials)

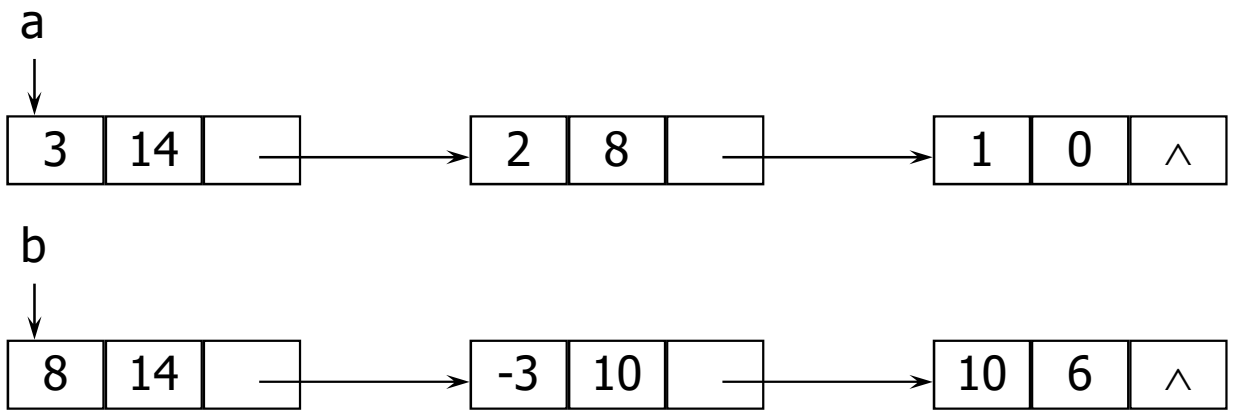
- 다항식을 단일 연결 리스트로 표현

```
struct poly {
        int coef;
        int expon;
        struct poly *link;
};
struct poly *a, *b, *d;
```

다항식의 예

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

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



- 다항식의 합
  - d = a + b의 처음 세 단계 (Figure 4.13) insert last (queue 방식), rear 알아야 함
  - Program 4.9 & Program 4.10: 두 개의 다항식의 합
  - 시간 복잡성: O(m + n)

그림 4.13: d = a + b의 첫 번째 단계

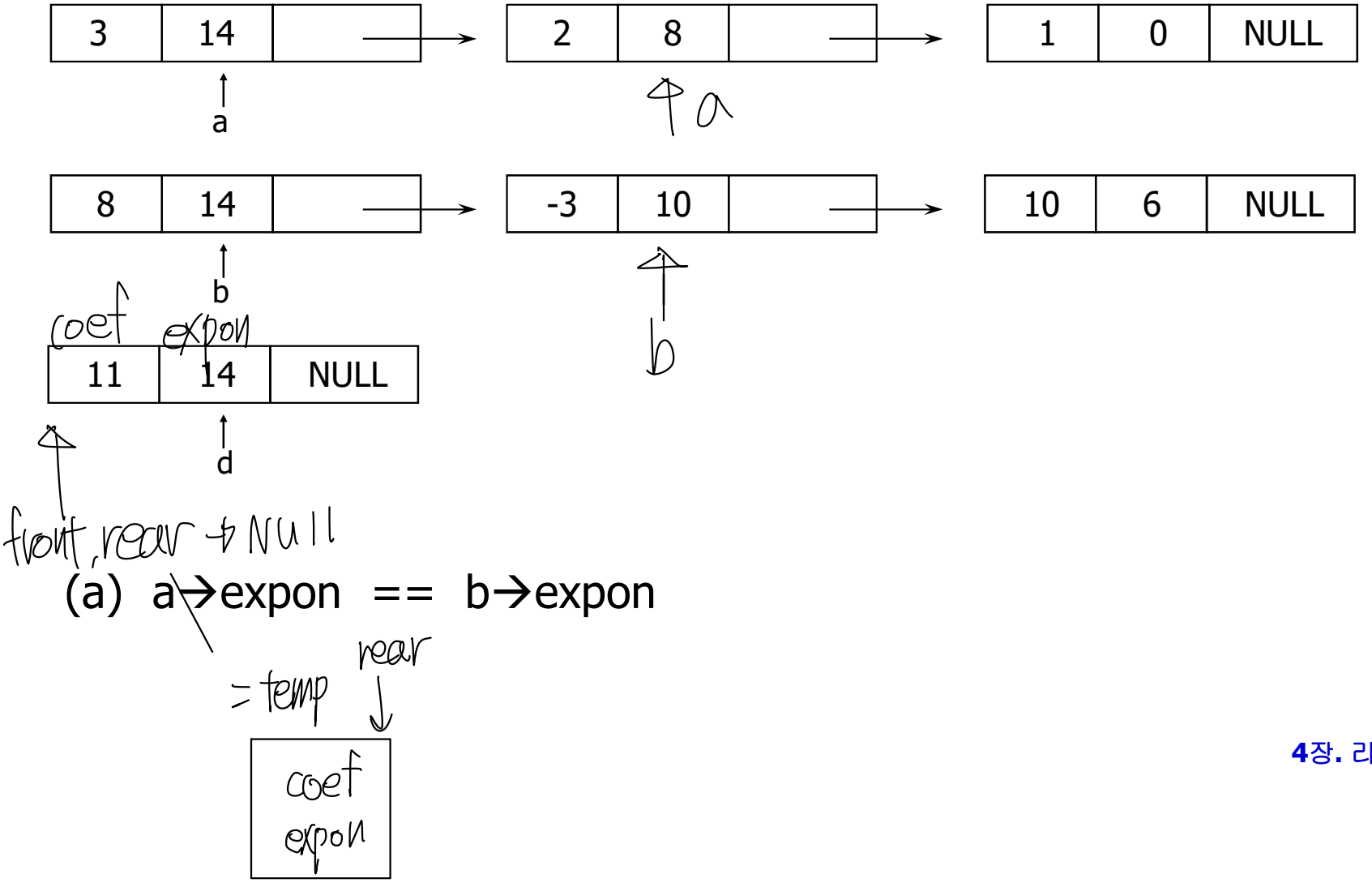
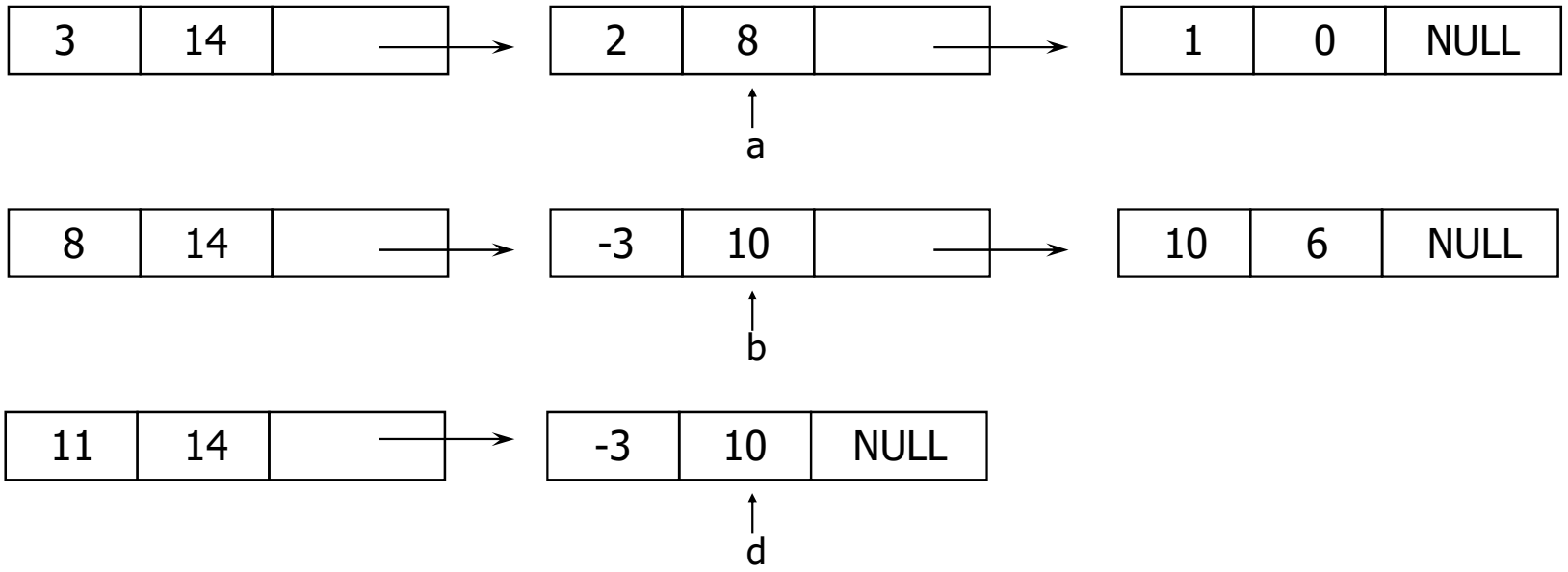
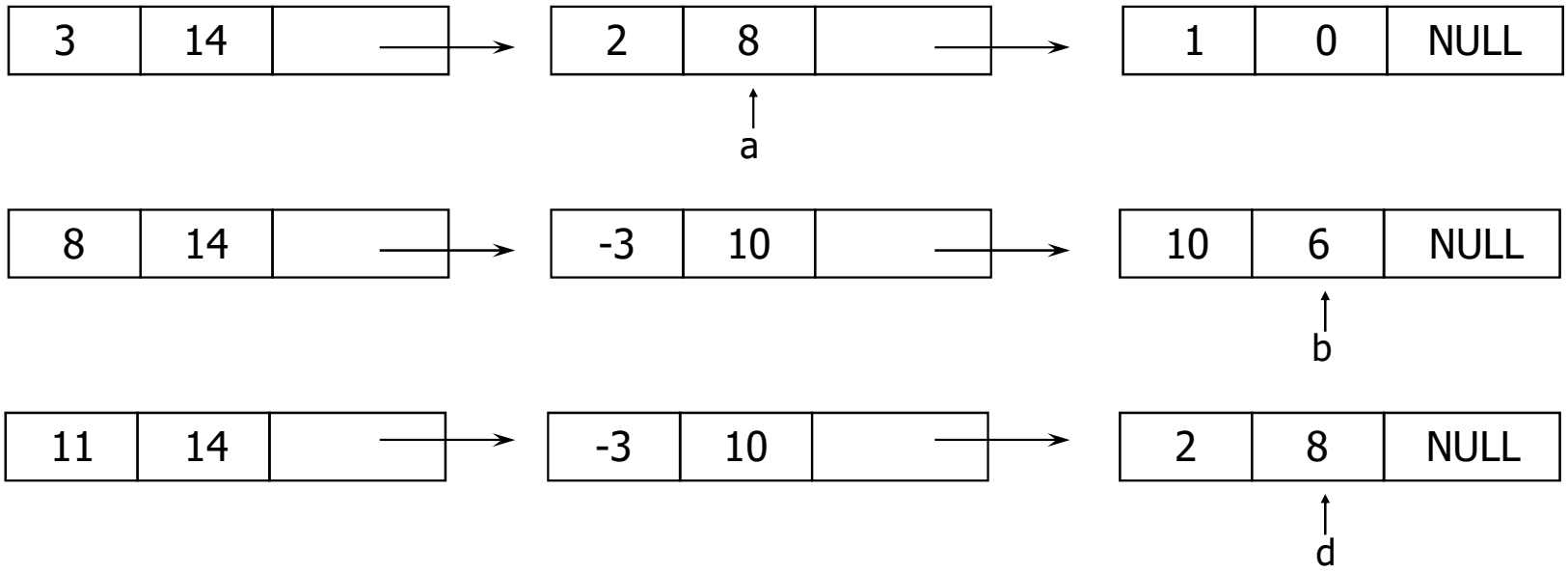


그림 4.13: d = a + b의 두 번째 단계



(b) a→expon < b→expon

그림 4.13: d = a + b의 세 번째 단계



(c) a→expon > b→expon

다항식 d의 항 추가는 리스트의 끝에서 발생 → Queue 방식!

Program 4.9: 두 개의 다항식 더하기(1)

```
struct poly *padd(struct poly *a, struct poly *b)
{
    // d = a + b인 다항식 d를 return
    struct poly *front, *rear, *temp;
    int sum;
    // 사용하지 않는 초기 노드를 생성. 이유는?
    rear = (struct poly *)malloc(sizeof(struct poly));
    if (rear == NULL) {
        fprintf(stderr, "The memory is full\n");
        exit(1);
    }
    front = 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;
```

front ≠ null

Program 4.9: 두 개의 다항식 더하기(2)

```
            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 b의 나머지 부분을 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;
  // 초기 노드를 삭제
  temp = front;  front = front→link;  free(temp);
  return front;
}
```

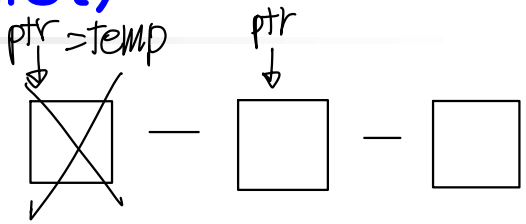
temp에서 front->link부터 값을 넣고 있기 때문에 front 첫 시작을 free해줌

Program 4.10: 리스트의 끝에 새로운 항 추가

```
void attach(float coefficient, int exponent, struct poly **rear)
{
/*  coef = coefficient이고 expon = exponent인 새로운 노드를 생성한 후, ptr이 가리키
    는 노드 다음에 연결. ptr은 생성된 노드를 가리키도록 변경 */

    struct poly *temp;
    temp = (struct poly *) malloc(sizeof(struct poly));
    if (temp == NULL)  {
          fprintf ( stderr, "The memory is full \n" );
          exit(1);
    }
    temp→coef  =  coefficient;
    temp→expon  =  exponent;
    (*rear)→link  =  temp;
    *rear =  temp;
}
```

가용 노드 리스트(Available Node List)

- 리스트의 모든 노드 삭제:
  - 복잡성 = O(리스트에서 노드의 수)



heap

맨처 malloc 해고 필요할때 쓰기 필요없을때 날려놓기

```
void erase(struct poly **ptr)
{
   // ptr이 가리키는 다항식 리스트를 삭제
   struct poly *temp;
   while (*ptr != NULL)
   {      temp = *ptr; *ptr = (*ptr)→link; free(temp); }
}
```

O(n)

erase (&start)

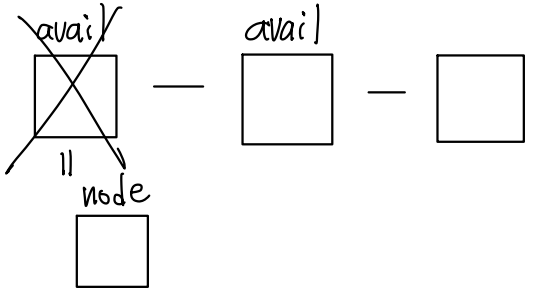
free(start)
→ start한 free 발생

malloc → system call (brk) 발생 (unix, linux)
↓
많이 쓰면 성능

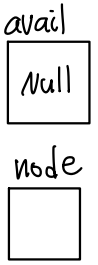
- 가용 노드 리스트(Available node list)
  - list of "freed" nodes // malloc()과 free() 함수의 호출 빈도수를 줄이자!
  - get_node( ): Program 4.12 & ret_node( ): Program 4.13

Program 4.12: get_node() 함수

```
struct poly *get_node (void)
{  // 사용가능한 하나의 노드를 가용 리스트나 malloc()을 이용하여 반환
    struct poly *node;
    if ( avail != NULL )  {
          node  =  avail;
          avail  =  avail→link;
    }
    else  {
          node  =  (struct poly *) malloc(sizeof(struct poly));  ← 랜덤 값 가져옴
          if ( IS_FULL(node) )  {
                fprintf ( stderr, "The memory is full \n" );
                exit(1);
          }
    }
    return  node;
}
```



메모리 부족

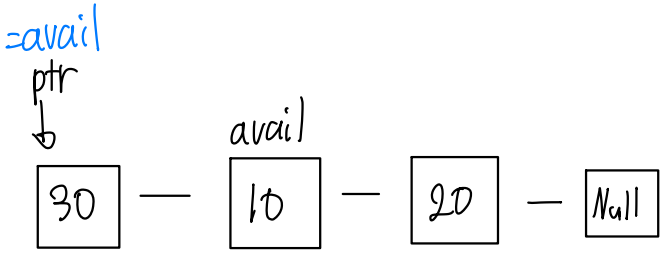


Program 4.13: ret_node() 함수

```
void ret_node ( struct poly *ptr )
{
        // ptr이 가리키는 노드를 가용 리스트에 반환
        ptr→link = avail;
        avail = ptr;
}
```

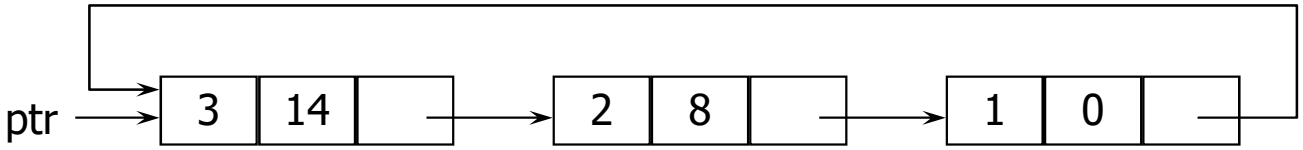
avail: 가용리스트의 첫번째 노드

ptr: 가용리스트에 반환하려는 노드



원형 리스트(Circular List)

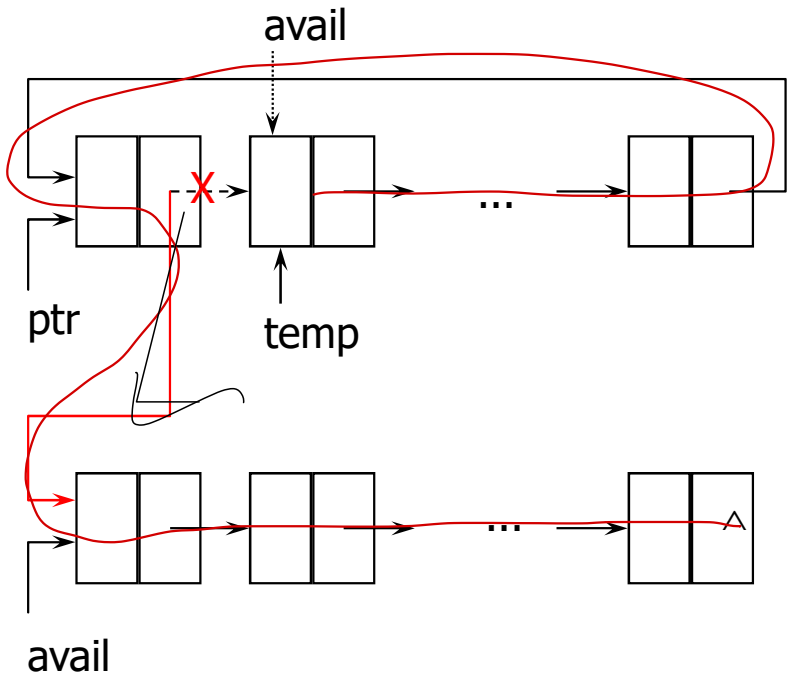
- 다항식을 원형 리스트로 표현
  - 원형 리스트의 정의
    - 마지막 노드의 link가 처음 노드를 가리키는 연결 리스트
    - Chain: 마지막 노드의 link가 null인 연결 리스트
  - 원형 리스트의 장점
    - 리스트의 모든 노드를 효율적으로 반환

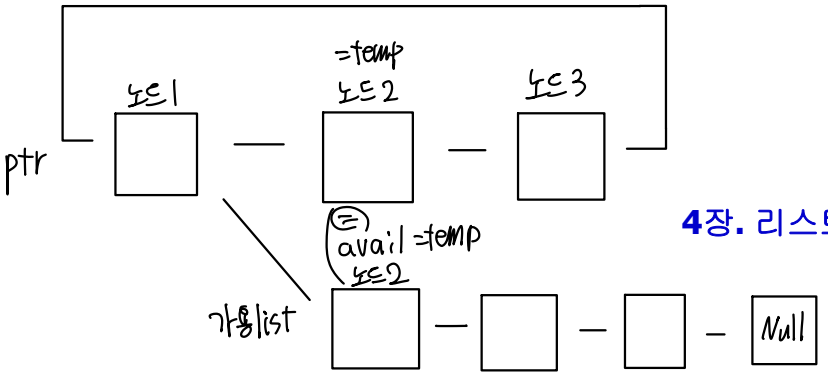


원형 리스트의 반환

```
void cerase(struct poly **ptr)
{
   /* ptr이 가리키는 원형 리
   스트를 반환 */
   struct poly *temp;
   if (*ptr) {
         temp = (*ptr)→link;
         (*ptr)→link = avail;
         avail = temp;
         *ptr = NULL;
   }
}
```

하나씩 free할 필요X





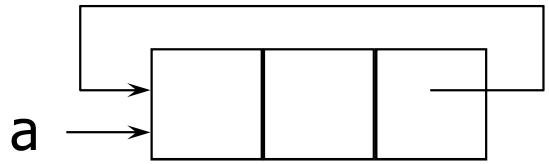
헤드 노드(Head Node)

- 노드가 없는 원형 리스트 → head node

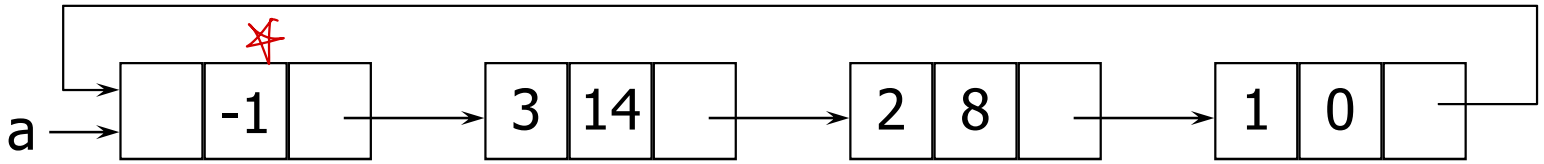
원형리스트 → null X

자신의 링크가 자신을 가리킴

노드 → head node



Zero polynomials



exponent가 -1

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

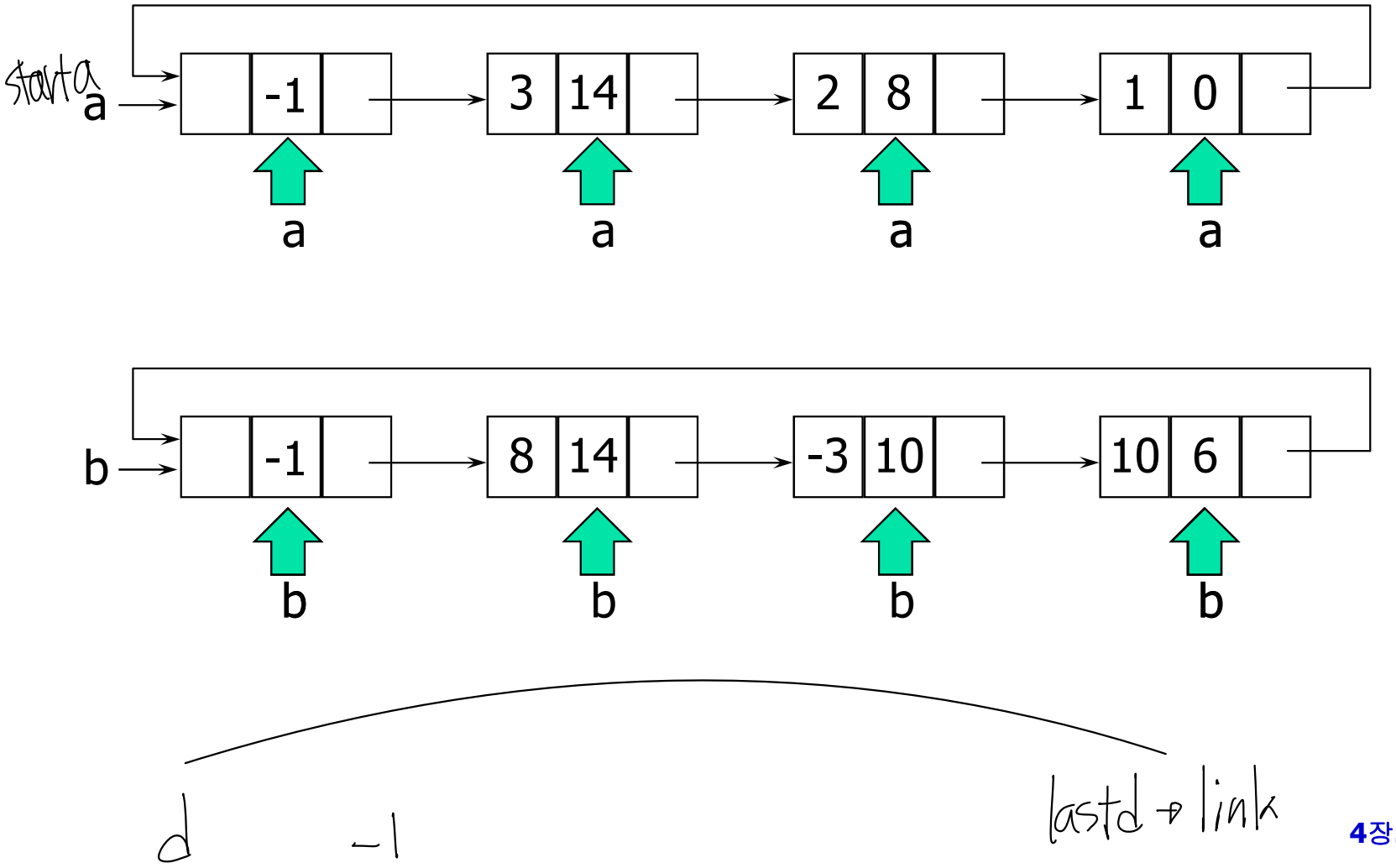
P4.15: 원형 리스트로 구현한 다항식의 더하기(1)

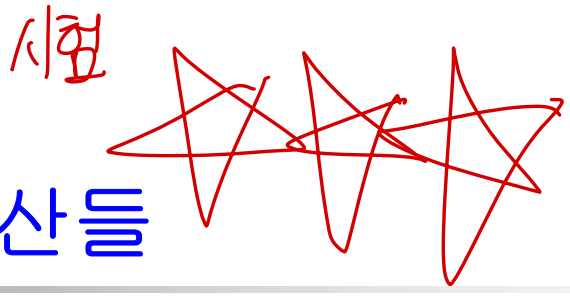
```
struct poly *cpadd(struct poly *a, struct poly *b)
{
/* 다항식 a와 b: singly linked circular lists with a head node.
   d = a + b를 계산한 후, d를 반환   */
   struct poly *starta, *d, *lastd;
   int sum, done = FALSE;
   starta = a;                              // a의 시작 노드를 기록
   a = a→link;    b = b→link;               // a와 b의 head node를 skip
   d = get_node();                          // 합을 위한 head node 할당
   d→expon = -1;    lastd = d;
   do   {
      switch (COMPARE(a→expon, b→expon))   {
          case -1:   // a→expon < b→expon
                     attach(b→coef, b→expon, &lastd);
                     b = b→link;  break;
```

P4.15: 원형 리스트로 구현한 다항식의 더하기(2)

```
        case 0:    // a→expon == b→expon
                   if (starta == a) done = TRUE;
                   else   {
                              sum = a→coef + b→coef;
                              if (sum) attach(sum, a→expon, &lastd);
                              a = a→link; b = b→link;
                   }
                   break;
        case: 1    // a→expon > b→expon
                   attach(a→coef, a→expon, &lastd);
                   a = a→link;
        }
    }   while ( !done );
    lastd→link  =  d;
    return  d;
}
```

원형 리스트로 구현한 다항식 더하기의 예





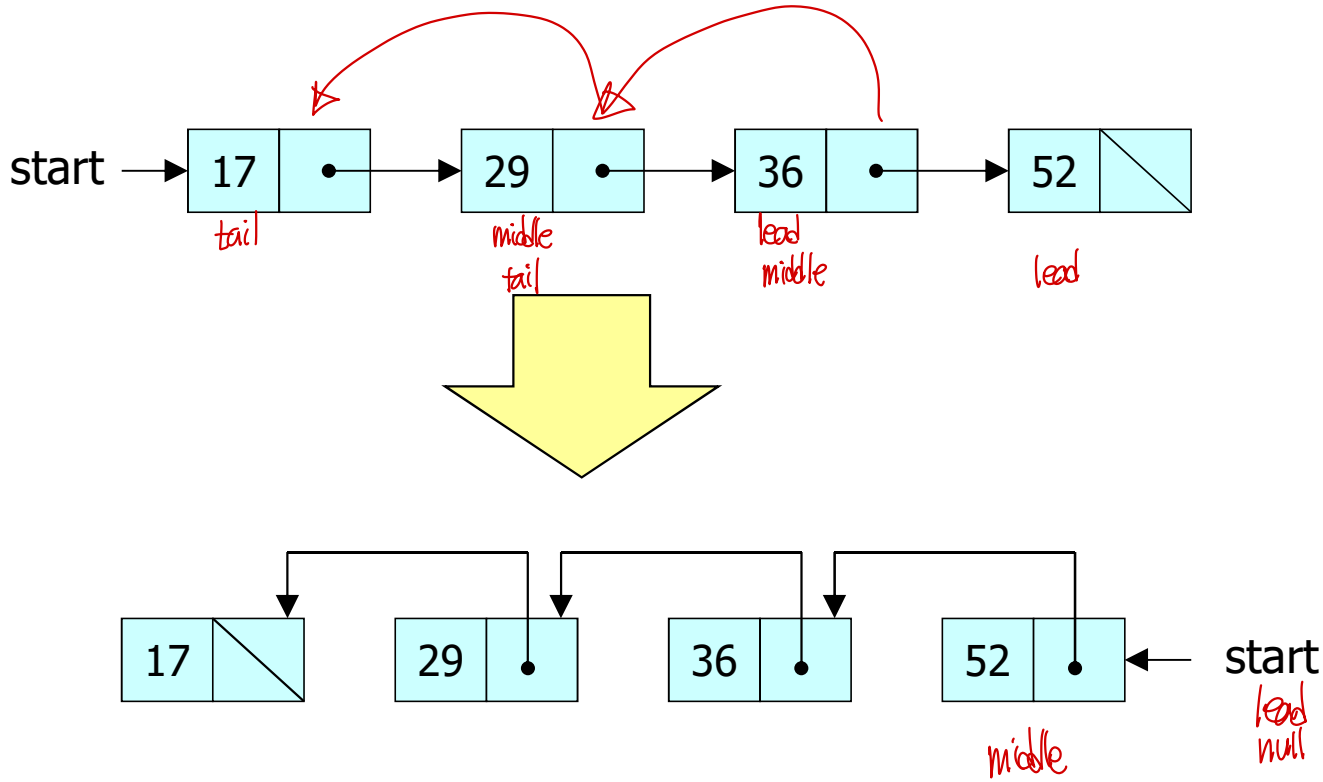
5. 추가적인 리스트 연산들

- **Chain**에 대한 연산
  - Chain의 방향을 반대로 : invert()
  - 두 개의 chain을 통합 : concatenate()

- 원형 리스트에 대한 연산
  - 원형 리스트의 선두에 새로운 노드를 삽입할 경우, 마지막 노드의 링크를 변경 ← 마지막 노드까지 가야 한다!
  - 해결 방법: insert_front()
    - 원형 리스트의 이름은 첫 번째 노드가 아니라 마지막 노드를 가리키도록 하자.

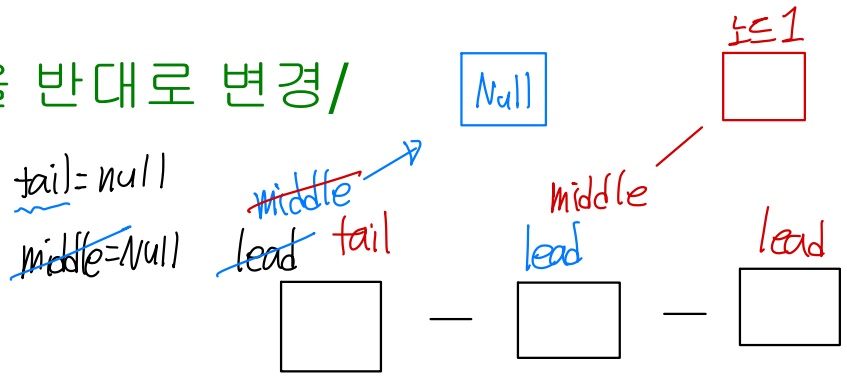
  - 원형 리스트의 길이를 계산하는 연산: length()

invert() 함수


start
17
29
36
52
tail
middle
tail
lead
middle
lead
17
29
36
52
start
middle
lead
null

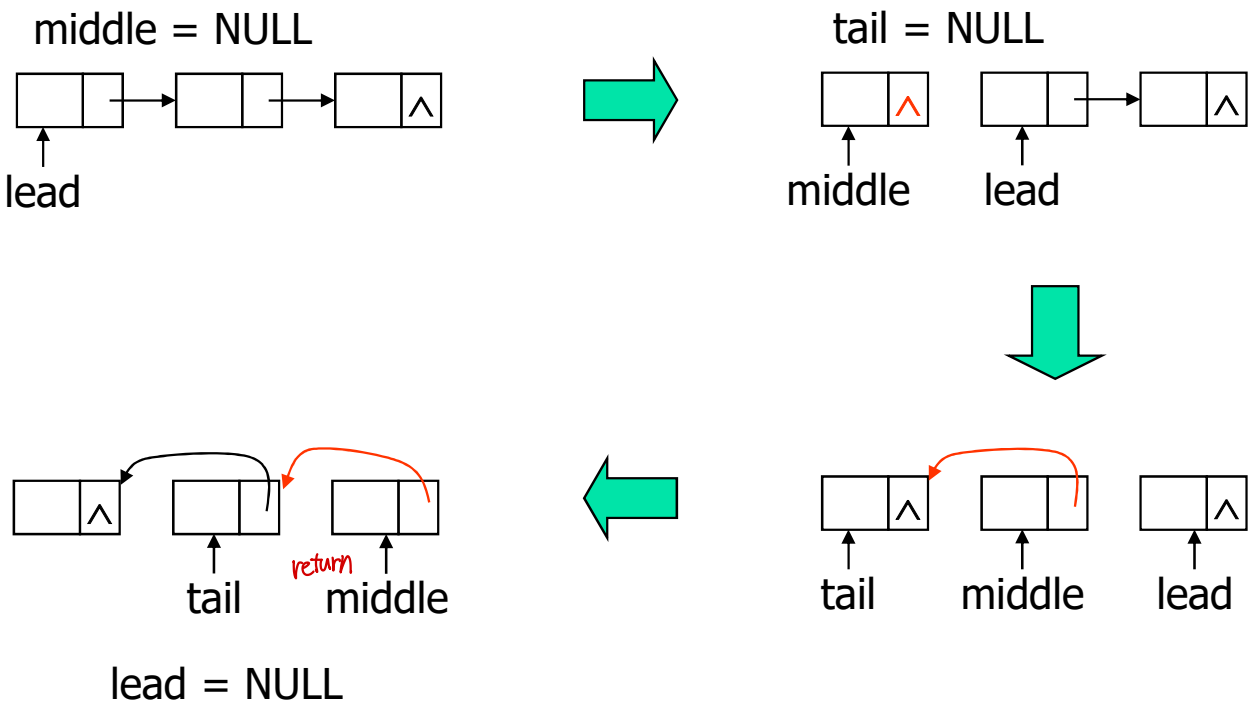
Program 4.16: invert() 함수

```
list_pointer invert(struct node *lead)
{
// lead가 가리키는 리스트의 방향을 반대로 변경/
    struct node *middle, *tail;
    middle = NULL;
    while (lead)   {
          tail = middle;
          middle = lead;
          lead = lead→link;
          middle→link = tail;
    }
    return middle;
}
```

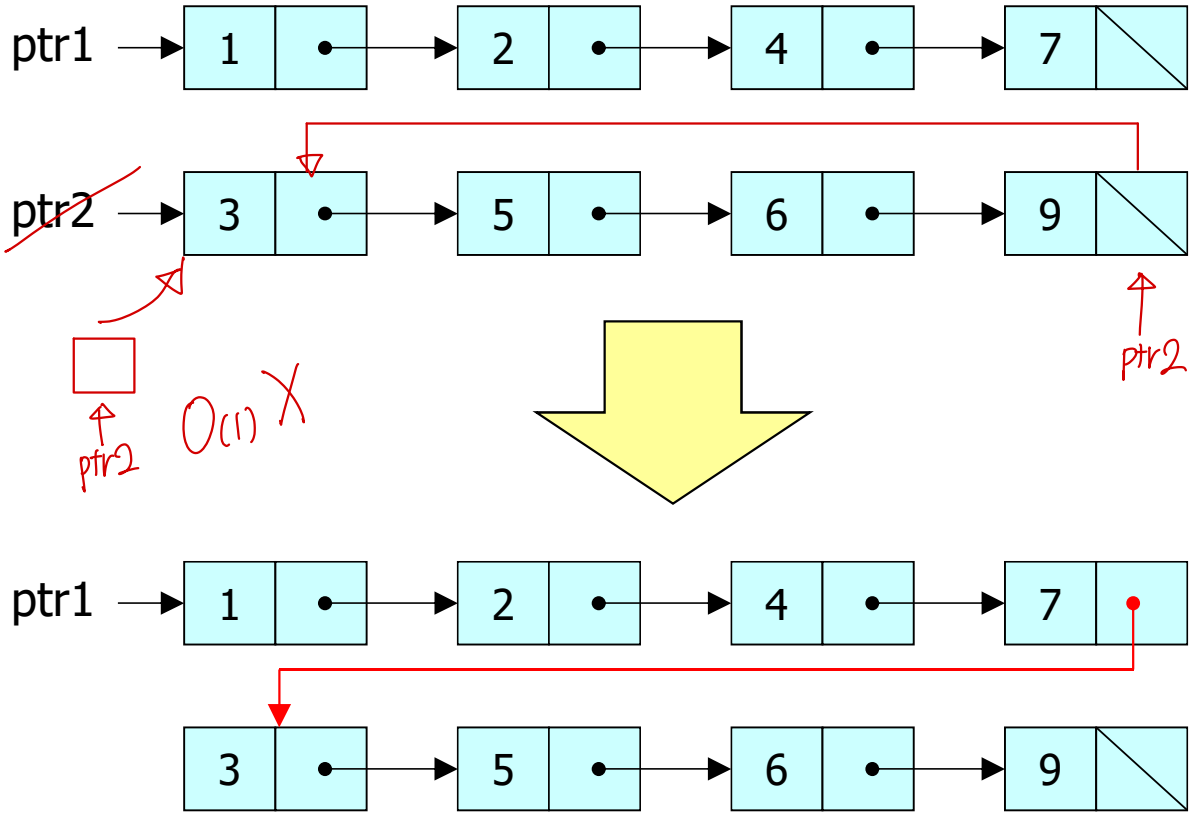


O(n)

invert() 함수의 동작 과정



Concatenate() 함수

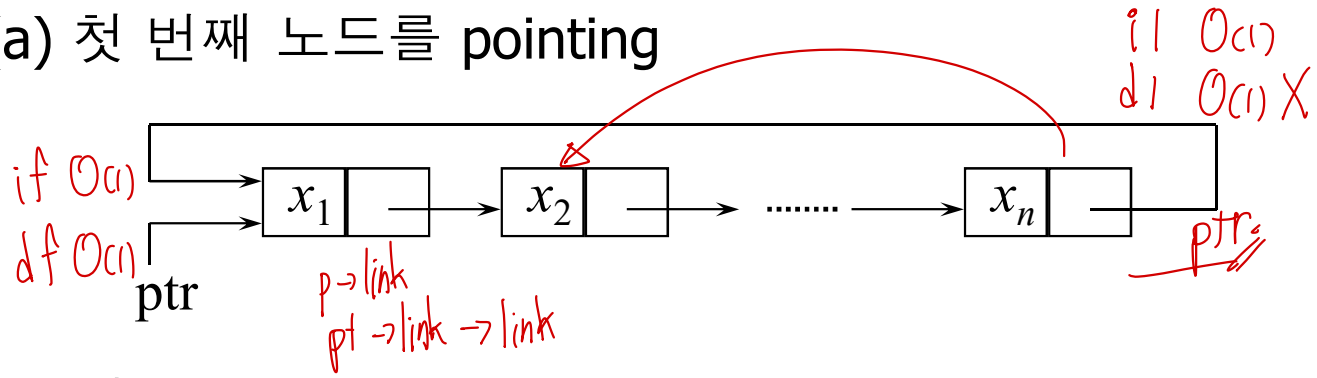


Program 4.17: concatenate()

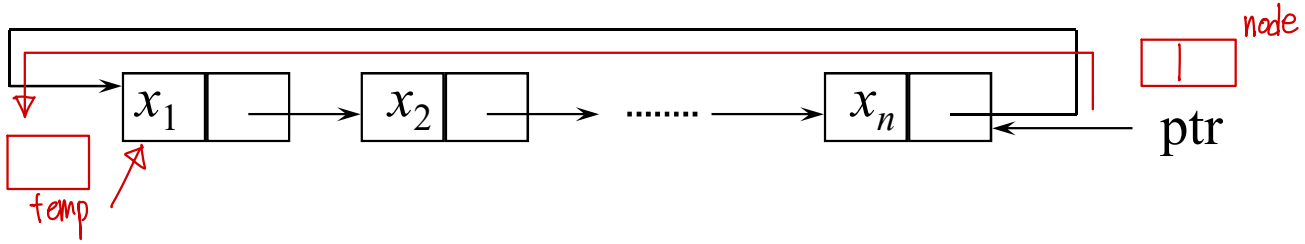
```
struct node *concatenate (struct node *ptr1, struct node  *ptr2)
{
/* ptr1다음에 ptr2를 연결한 새로운 리스트를 반환.
   ptr1이 가리키는 리스트는 새로운 리스트로 변경됨. */
   struct node *t;
   if (ptr1 == NULL)   return ptr2;
   else   {
        if (ptr2 != NULL)   {
             for (t = ptr1; t→link != NULL; t = t→link)
                  ;
             t→link = ptr2;
        }
        return  ptr1;
   }
}
```

원형 리스트의 이름

(a) 첫 번째 노드를 pointing



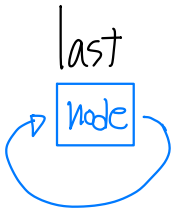
(b) 마지막 노드를 pointing

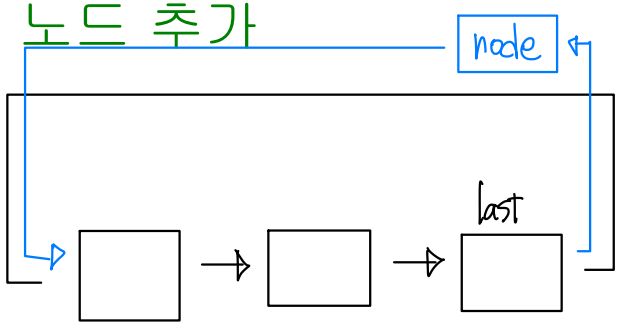


- 마지막 노드를 **pointing**할 경우, 다음 연산의 복잡성은 **O(1)**
  - 리스트의 첫번째 위치에 새로운 노드 추가
  - 리스트의 마지막 위치에 새로운 노드 추가

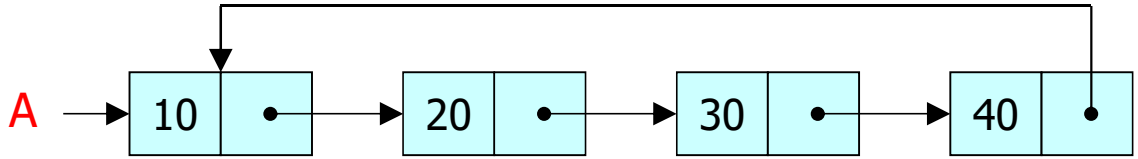
Program 4.18: 리스트의 선두에 노드 추가

```
void  insert_front (struct node **last, struct node *node )
 /* last가 가리키는 원형 리스트의 선두에 node 추가.
    last는 원형 리스트의 마지막 노드를 가리키고 있음. */
{
    if (*last == NULL )   {
    // 리스트가 비었음: last가 새로운 노드를 가리키도록 변경
          *last = node;
          node→link = node;
    }
    else   {
    // 리스트에 노드가 존재: 선두에 노드 추가
          node→link = (*last)→link;
          (*last)→link = node;
    }
}
```





원형 연결 리스트의 순회



조심해서 사용

```
struct node *ptr;

for (ptr = A; ptr != null; ptr = ptr→link)
    sum += ptr→data;
```

debugging 쉬움

Chain의 순회

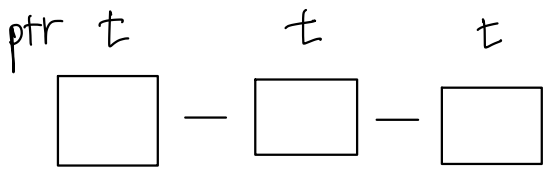
```
struct node *ptr = A;
sum = ptr→data;      // 처음 데이터는 별도로 처리
for (ptr = A→link; ptr != A; ptr = ptr→link)
    sum += ptr→data;
```

error 있으면 무한 loop

원형 연결 리스트의 순회

Program 4.19: 원형 리스트의 길이 계산

```
int length (struct node *ptr)
{
// ptr이 가리키는 원형 리스트의 노드 수를 return
        struct node *t;
        int count = 0;
        if (ptr != NULL)  {
                t = ptr;
                do  {
                        count++;
                        t = t→link;
                }  while (t != ptr);
        }
        return count;
}
```

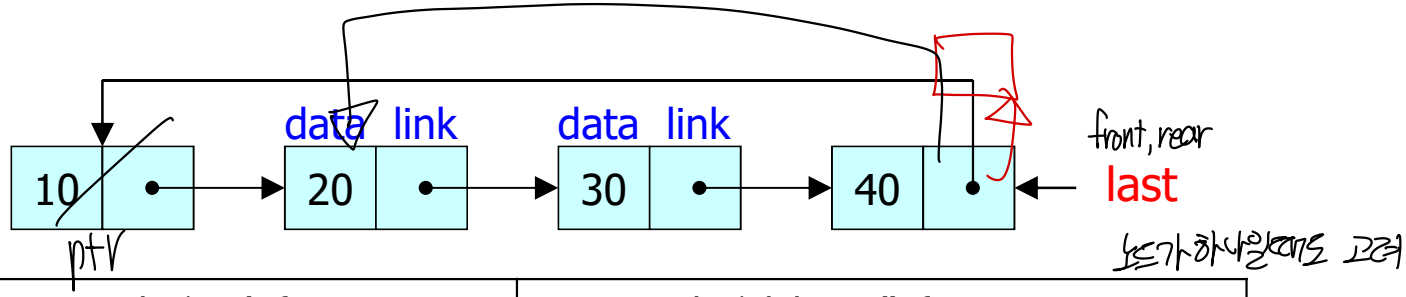


비효율

Chain에서 노드 수를 return하는 함수?

연습 문제

- 원형 연결 리스트를 이용하여 큐(queue)를 구현할 경우, 아래와 같이 하나의 포인터 last로 front와 rear를 모두 담당할 수 있다. addq()와 deleteq() 함수에서 (가)와 (나)에 들어갈 내용은 무엇인가?




```
void addq(struct node *ptr) {
    // ptr 노드를 큐의 제일 뒤에 추가
    if (last == NULL) {
        last = ptr;
        last→link = last;
    }
    else {
        (가)
    }
}
```

(가)
ptr->link = last->link
last->link = ptr
last = ptr

```
struct node *deleteq() {
    // 큐의 제일 앞 노드를 return
    struct node *ptr;
    if (last == NULL) {
        return NULL;    // QueueEmpty
    }
    else {
        (나)
    }
}
```

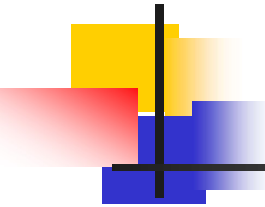
(나)
ptr = last->link
last->link = ptr->link
ptr = last->link
return ptr

6. 동치 관계(Equivalence Relations)

- 동치 관계
  - reflexive, symmetric, transitive 성질을 만족
  - "equal to"(=) 관계는 동치 관계임.
    - x = x
    - x = y 이면 y = x
    - x = y 이고 y = z 이면 x = z

- 동치 관계를 이용하여 집합 **S**를 "동치 클래스"로 분할
  - 동일한 클래스내의 원소 x, y에 대해서는 x ≡ y 관계 성립
  - 예: 0 ≡ 4, 3 ≡ 1, 6 ≡ 10, 8 ≡ 9, 7 ≡ 4, 6 ≡ 8, 3 ≡ 5, 2 ≡ 11, 11 ≡ 0
  - 동치 클래스: {0, 2, 4, 7, 11}; {1, 3, 5}; {6, 8, 9, 10}

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



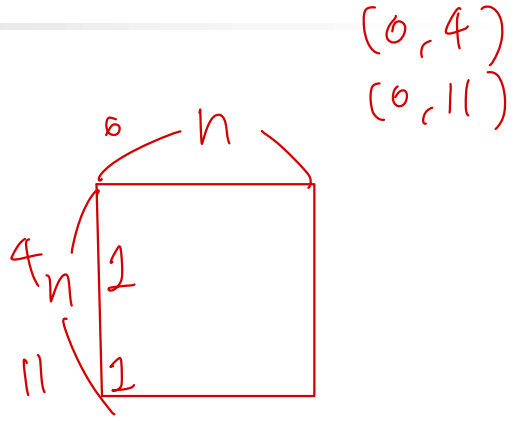
동치 클래스 검색 알고리즘

1. 동치항 <i, j>를 입력한 후 저장

2. 원소 0부터 시작하여 0의 동치항 <0, j> 항들을 검색한 후, j를 0과 동일한 클래스에 포함
   - Transitivity 속성에 의해 <j, k> 항의 원소 k도 0과 동일한 클래스에 포함됨
   - 이 과정을 반복하여 0을 포함하는 모든 원소를 발견

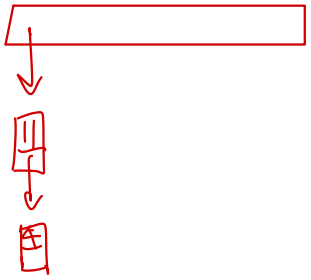
3. 클래스에 포함되지 않은 다른 원소들에 대해 단계 2를 반복

동치 클래스 검색을 위한 자료 구조(1)

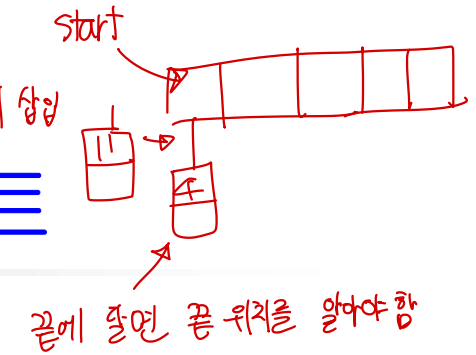


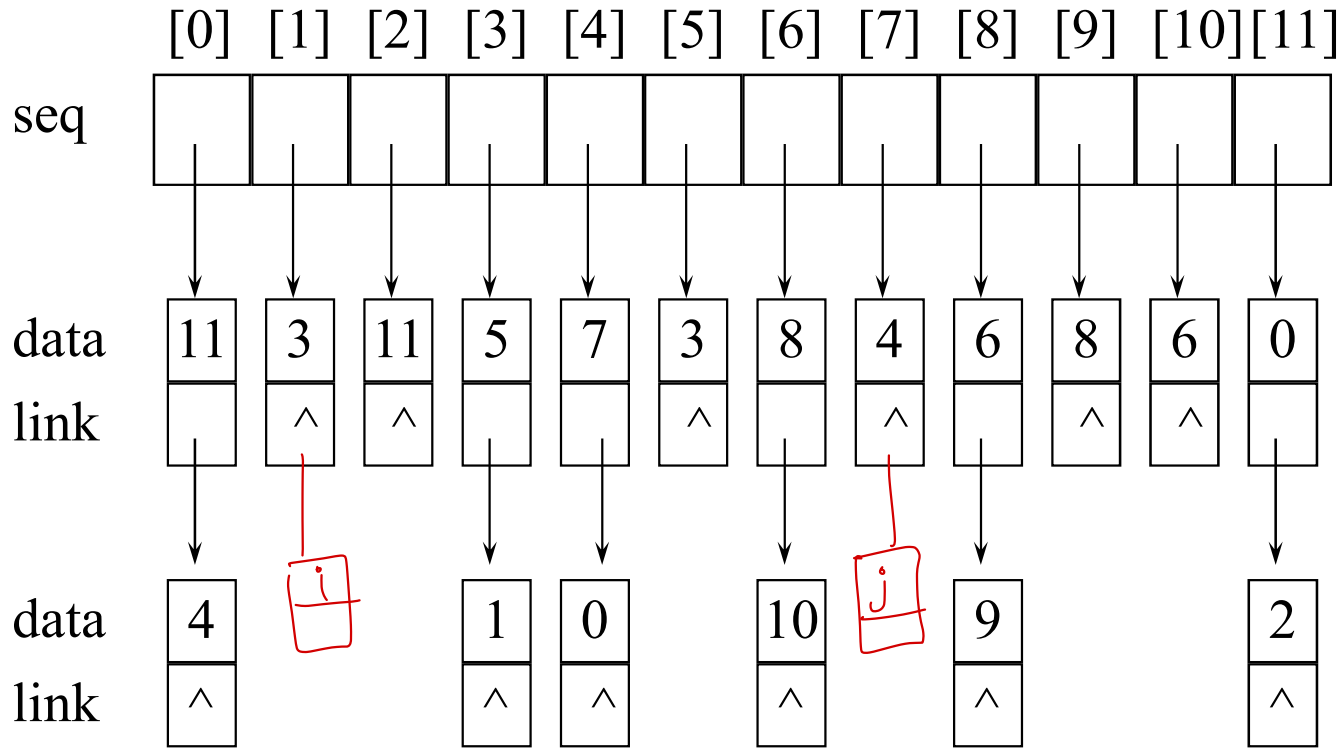
- 변수 선언
  - m: 입력된 동치항의 수
  - n: 원소의 수

- 동치항의 구현 방법
  - 배열: pairs[n][n]
    - <i, j>가 입력될 경우, pairs[i][j]와 pairs[j][i]를 1로 설정
    - 원소의 수에 비해 동치항의 수가 적을 경우, 메모리 낭비
  - n개의 연결 리스트: seq[n]
    - <i, j>가 입력될 경우
      - 노드 j를 seq[i]가 가리키는 리스트에 추가
      - 노드 i는 seq[j]가 가리키는 리스트에 추가



예: 동치항의 입력이 완료된 후의 리스트





(i, j)

out[ ] = true / false

{0, 11, 4, 7, ✗, ✗, 2, ✗}

저장해놓는 스택

이미 참

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

동치 클래스 검색을 위한 자료 구조(2)

- 일차원 boolean 배열 **out[n]**
  - out[i] = TRUE: 원소 i를 출력하여야 함. (초기값)
  - out[i] = FALSE: i가 이미 출력되어 다시 출력할 필요 없음.

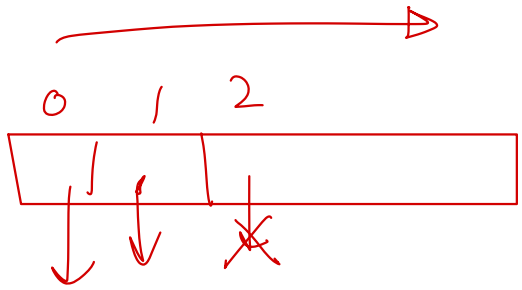
- 리스트 표현을 이용한 동치 클래스 검색
  - Step 1: 동치항을 입력받아 seq[]를 이용한 리스트 구성
  - Step 2: out[i] = TRUE인 첫번째 원소 i ($0 \le i < n$) 를 선택하여 seq[i] 리스트를 스캔하면서 리스트의 원소들을 출력
    - seq[i]의 원소 중에서 out[] 배열의 값이 TRUE인 원소들의 리스트들을 현재 스캔을 완료한 후 스캔하여야 함. (Stack이 필요) ← 연결배열X
    - Stack을 구현하기 위하여 해당 노드의 link 필드를 stack의 다음 원소를 가리키는 포인터로 변경

Program 4.20: 초기 동치 알고리즘

```
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 );
}
```

Program 4.21: 수정된 동치 알고리즘

```
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])    {
                    out[i] = FALSE;
                    output this equivalence class;
            }
}
```

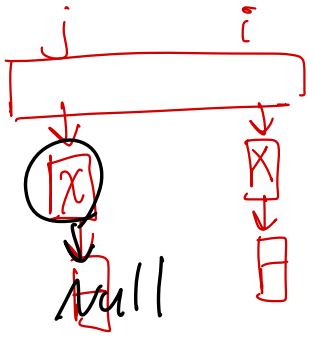


Program 4.22: 최종 동치 알고리즘(1)

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

struct node  {
    int data;                    // 정수형의 데이터가 입력된다고 가정
    struct node *link;
};

void main(void)
{   short int  out[MAX_SIZE];
    struct node *seq[MAX_SIZE], *x, *y, *top;
    int i, j, n;
    printf("Enter the size (<= %d)  ", MAX_SIZE );
    scanf("%d",  &n);
                          0~n-1
```

Program 4.22: 최종 동치 알고리즘(2)

```
    for (i = 0; i < n; i++)          // seq[]와 out[] 배열을 초기화
    {     out[i] = TRUE;             seq[i] = NULL;  }

/* Phase 1: Input the equivalence pairs: */
    printf("Enter a pair of numbers (-1 -1 to quit): ");
    scanf("%d%d", &i, &j);
    while (i >= 0)   {     // 음수가 입력되면 리스트 생성 종료
          x = (struct node *) malloc(sizeof(struct node));
          x→data = j;   x→link = seq[i];   seq[i] = x;      // j를 리스트 i의 앞에 추가

          x = (struct node *) malloc(sizeof(struct node));
          x→data = i;   x→link = seq[j];   seq[j] = x;      // i를 리스트 j의 앞에 추가

          printf("Enter a pair of numbers (-1 -1 to quit): ");
          scanf("%d%d", &i, &j);
    }
```

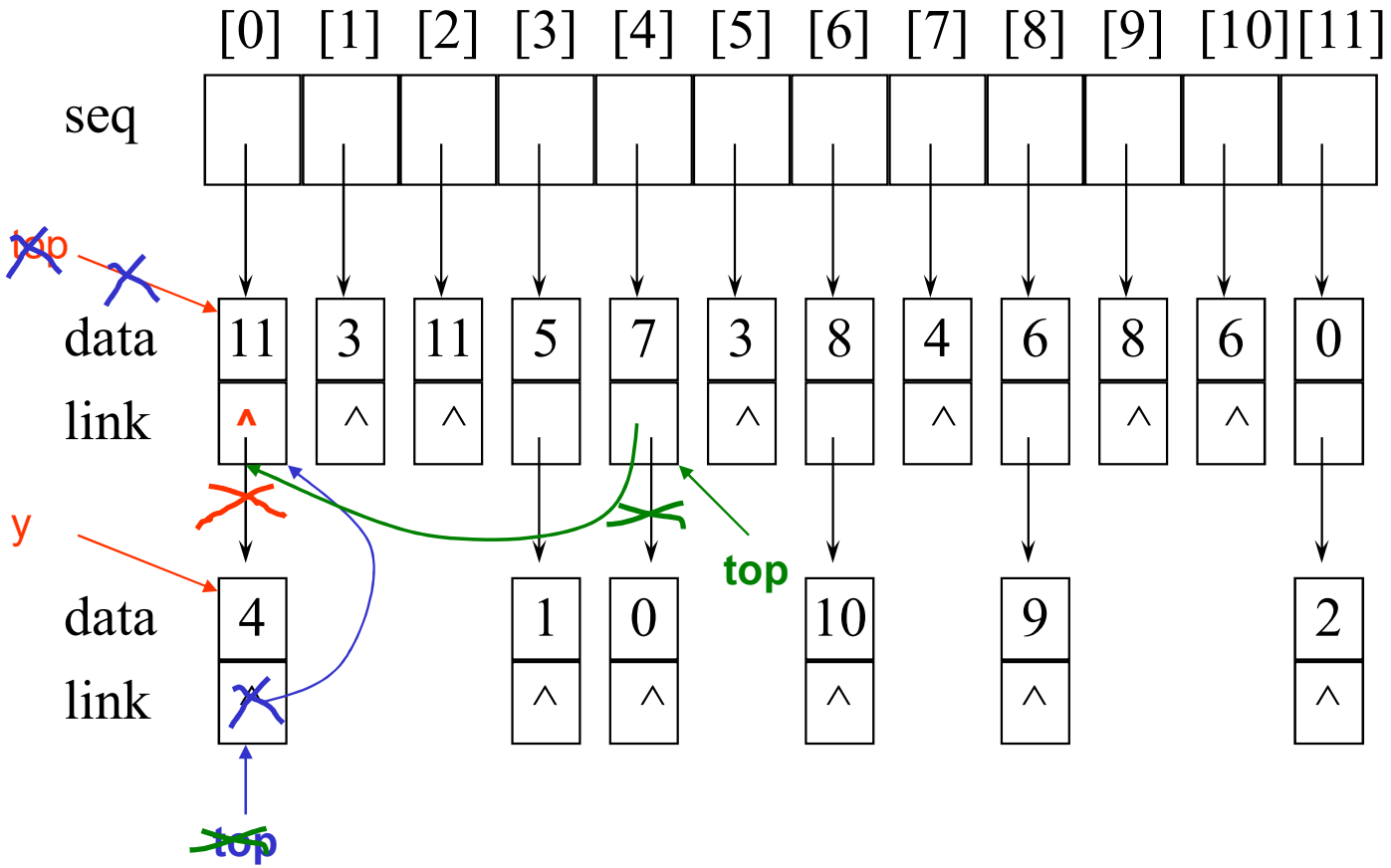


Program 4.22: 최종 동치 알고리즘(3)

```
    for (i = 0; i < n; i++)      /* Phase 2: output the equivalence classes */
        if (!out[i])        continue;
        printf("\n New class: %5d ", i );        // 새로운 클래스 출력 시작
        out[i] = FALSE;                          // i를 출력하였음.
        x = seq[i]; top = NULL;                  // 스택 초기화
        for ( ; ; )    {                         // 클래스의 나머지 원소를 찾자
            while (x)   {                        // 리스트를 스캔
                j = x→data;
                if (out[j])   {                  // j가 아직 출력되지 않았다면...
                    printf("%5d", j );   out[j] = FALSE;          // j를 출력한 후,
                    y = x→link; x→link = top; top = x; x = y;     // push()
                }
                else x = x→link;
            }
            if (!top)                            // 현재 클래스의 모든 원소를 출력하였음.
                break;
            x = seq[top→data]; top = top→link;           // pop()
}       }
```

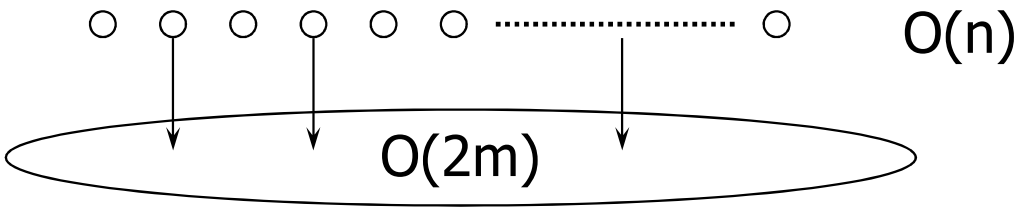
i=1

예: 스택의 삽입과 삭제 과정



동치 알고리즘의 분석

- m: 입력된 동치항의 수, n: 원소의 수
- 초기화 및 동치항의 입력 단계: O(m+n)
- 동치 클래스 출력 단계: O(m+n)
  - 각 노드는 기껏해야 한번 스택에 저장
  - 2*m 개의 노드가 존재하며, for loop는 n 번 실행

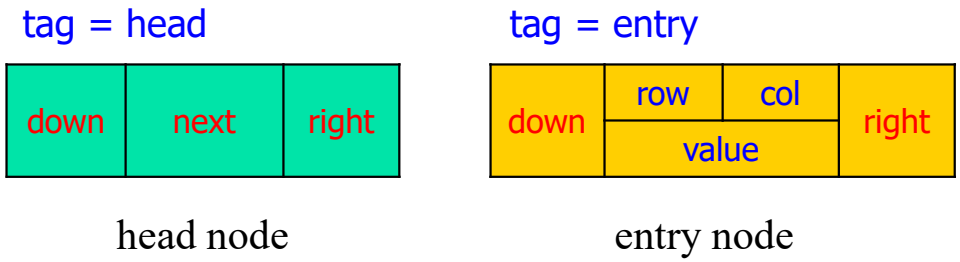


- 전체적인 시간 복잡도 = **O(m+n)**

7. 희소 행렬(Sparse Matrices)

- 배열을 이용한 표현 방식의 문제점
  - 행렬마다 0이 아닌 원소의 수가 가변

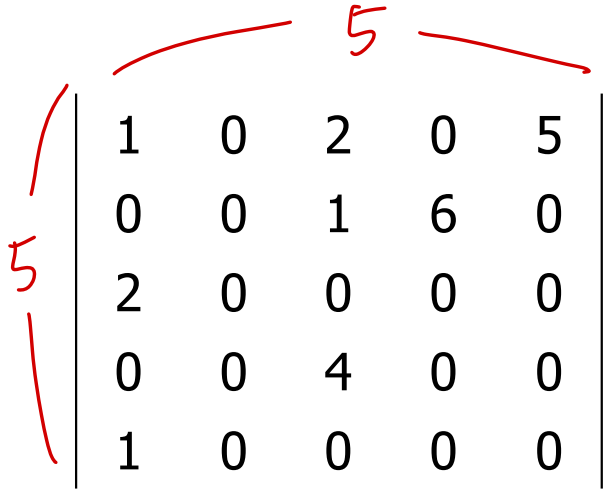
- 연결 리스트를 이용한 희소 행렬
  - 행과 열: 헤드 노드(head node)를 갖는 환형 연결 리스트



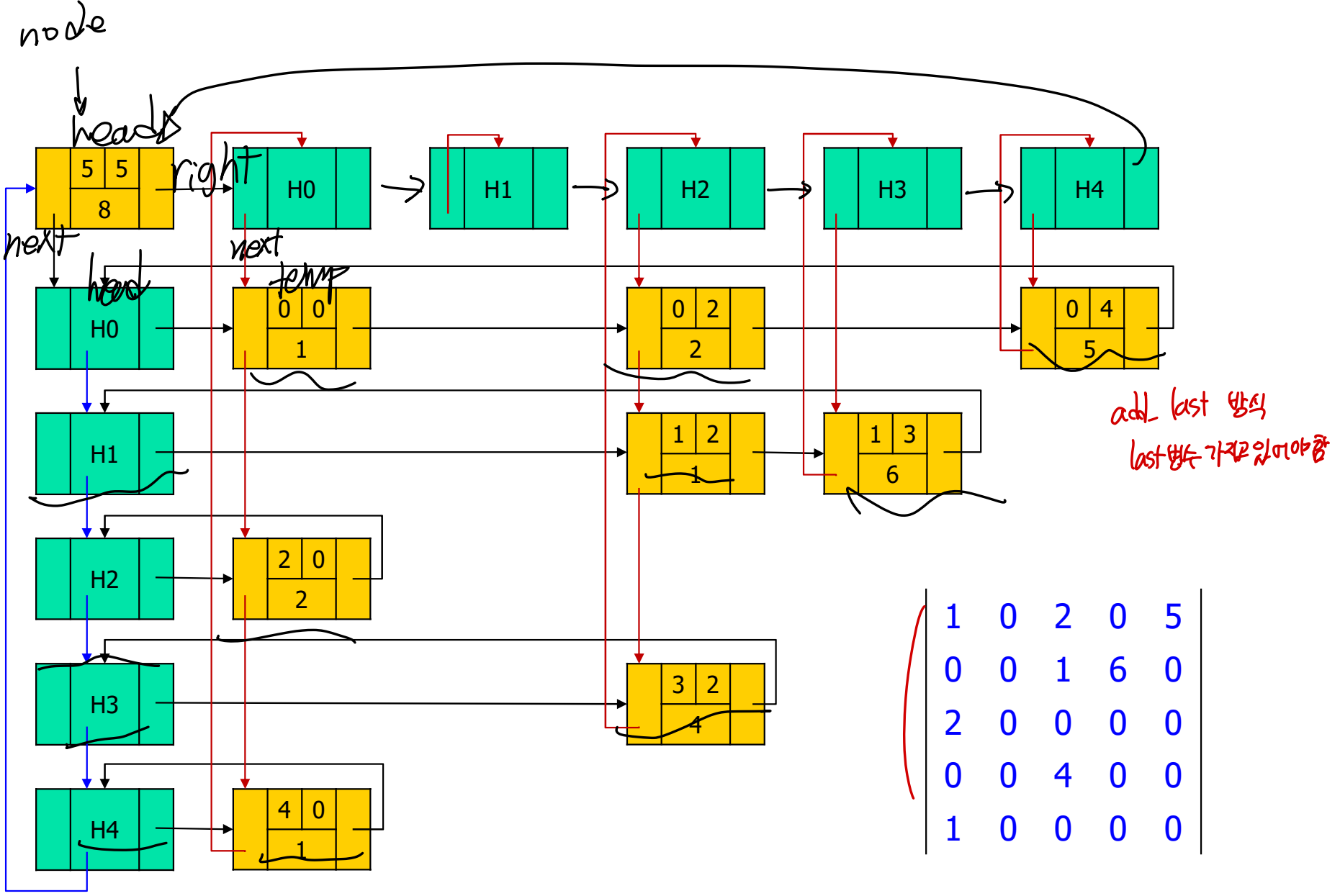
C 언어를 이용한 희소 행렬의 선언

```
#define MAX_SIZE 50   // 행과 열의 최대 크기
typedef enum {head, entry} tagfield;
typedef struct matrix_node *matrix_pointer;
struct entry_node {
        int       row;
        int       col;
        int       value;
};
struct matrix_node {
        matrix_pointer  down;   // 동일한 열의 다음 원소를 연결
        matrix_pointer  right;  // 동일한 행의 다음 원소를 연결
        tagfield        tag;
        union {
                matrix_pointer  next;
                struct entry_node entry;
        } u;
};
matrix_pointer  hdnode[MAX_SIZE];
```

예제 희소 행렬



	row	col	value
[0]	5	5	8
[1]	0	0	1
[2]	0	2	2
[3]	0	4	5
[4]	1	2	1
[5]	1	3	6
[6]	2	0	2
[7]	3	2	4
[8]	4	0	1



1	0	2	0	5
0	0	1	6	0
2	0	0	0	0
0	0	4	0	0
1	0	0	0	0

Program 4.23: 희소 행렬의 입력(1)

```
matrix_pointer mread(void) {
/* 희소 행렬을 입력한 후 리스트 구축. hdnode[] 배열은 전역 변수임. */
    int num_rows, num_cols, num_terms, num_heads, i;
    int row, col, value, current_row;
    matrix_pointer temp, last, node;

    printf("Enter # of rows, columns, and number of nonzero terms: ");
    scanf("%d%d%d", &num_rows, &num_cols, &num_terms);
    num_heads = (num_cols > num_rows) ? num_cols : num_rows;

    node = new_node(); node→tag = entry;      // 행렬의 시작 노드: headnode
    node→u.entry.row = num_rows;    node→u.entry.col = num_cols;
    node→u.entry.value = num_terms;

    if (!num_heads) node→right = node;
    else {                                    // 헤드 노드의 초기화
        for (i = 0; i < num_heads; i++) {
            hdnode[i] = new_node();          hdnode[i]→tag = head;
            hdnode[i]→right = hdnode[i];     hdnode[i]→u.next = hdnode[i];
        }
```

yellow

green

i번째 행의 last

Program 4.23: 희소 행렬의 입력(2)

```
current_row = 0;  last = hdnode[0];     // 현재 행의 마지막 노드
for (i = 0; i < num_terms; i++)   {
        printf("Enter row, column and value: ");
        scanf("%d%d%d", &row, &col, &value);
        if (row > current_row) {          // 현재 행을 마무리한 후, 다음 행으로 변경
                last→right = hdnode[current_row];
                current_row = row;  last = hdnode[row];
        }
        temp = new_node();
        temp→tag = entry;  temp→u.entry.row = row;
        temp→u.entry.col = col;  temp→u.entry.value = value;
        last→right = temp;                          // 행 리스트의 마지막에 연결
        last = temp;
        hdnode[col]→u.next→down = temp;     // 열 리스트의 마지막에 연결
        hdnode[col]→u.next = temp;
}
```

Program 4.23: 희소 행렬의 입력(3)

```
        last→right = hdnode[current_row];       // 마지막 행을 마무리
        for (i = 0; i < num_cols; i++)          // 모든 열 리스트들을 마무리
                hdnode[i]→u.next→down = hdnode[i];
        for (i = 0; i < num_heads-1; i++)       // 모든 헤드 노드들을 연결
                hdnode[i]→u.next = hdnode[i+1];
        hdnode[num_heads-1]→u.next = node;
        node→right = node→down = hdnode[0];
    }
    return node;
}
```

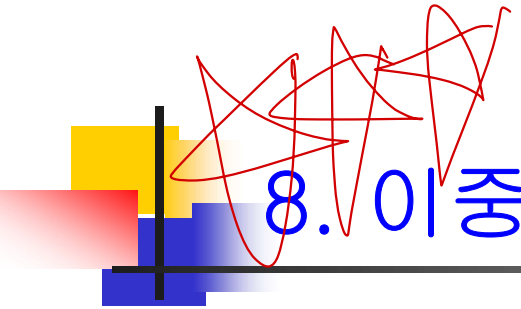
Program: new_node()

```
matrix_pointer  new_node ( void )
{
// 새로운 matrix node를 할당
    matrix_pointer  temp;
    temp  =  (matrix_pointer) malloc (sizeof(struct matrix_node));
    if (temp == NULL)   {
            fprintf (stderr, "The memory is full ₩n");
            exit (1);
    }
    return  temp;
}
```

Program 4.24: 희소 행렬의 출력

```
void mwrite(matrix_pointer node)
{
// 행렬의 모든 원소들을 row major 순서로 출력
    int i;
    matrix_pointer temp, head = node→right;
    // 행렬의 차수 출력
    printf("num_rows = %d, num_cols = %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 포함된 원소들 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;  // next row
}   }
```

head에는 data X



8. 이중 연결 리스트(Doubly Linked Lists)

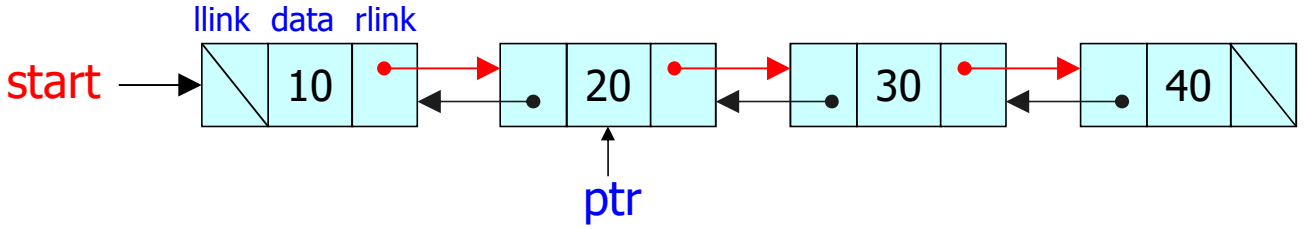
- 이중 연결 리스트(Doubly linked list)란?
  - 한 노드에 두 개의 link가 저장

```
struct node {
        struct node *llink;        // 이전 노드를 포인트
        int         data;
        struct node  *rlink;       // 다음 노드를 포인트
};
```

- 이중 연결 리스트는 양 방향으로 이동 가능
  - 단일 연결 리스트의 경우, 한 방향으로만 이동 가능

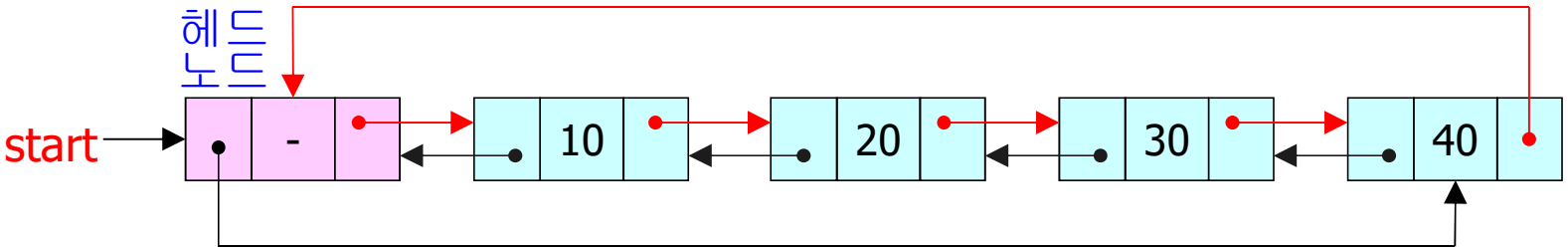
이중 연결 리스트의 종류

- 체인
  - 처음 노드의 llink와 마지막 노드의 rlink는 NULL

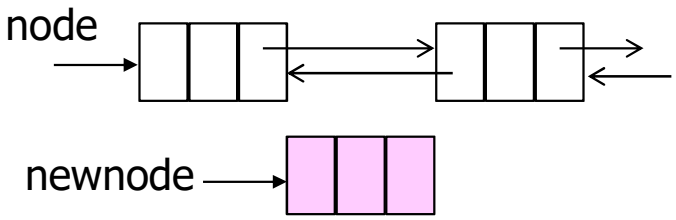


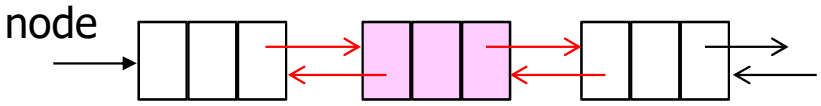
  - ptr = ptr→llink→rlink = ptr→rlink→llink

- 원형 이중 연결 리스트



원형 이중 연결 리스트에 노드 추가



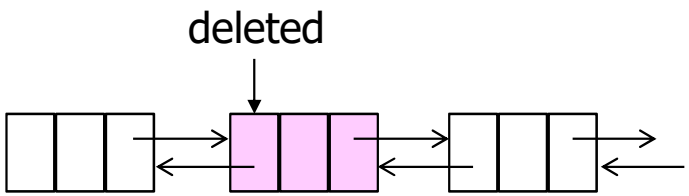



```
void dinsert(struct node *node,
struct node *newnode)
{
// newnode를 node의 오른쪽에 추가
    newnode→llink = node;
    newnode→rlink = node→rlink;
    node→rlink→llink = newnode;
    node→rlink = newnode;
}
```

참조하고 있는 것은 마지막에 연결

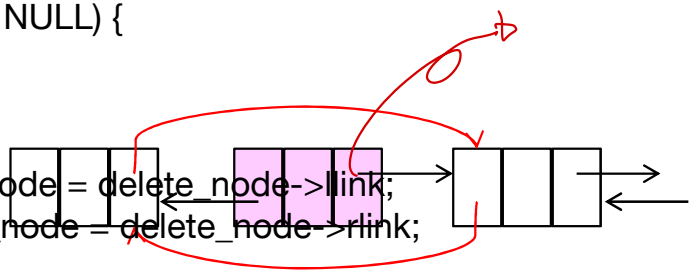
다른 문제들:
(1) newnode를 node의 왼쪽에 추가
(2) 원형 리스트가 아닌 이중 연결 chain의 왼쪽과 오른쪽에 노드 추가

원형 이중 연결 리스트에서 노드 삭제




```
void ddelete(struct node *deleted)
{
    deleted→llink→rlink = deleted→rlink;
    deleted→rlink→llink = deleted→llink;
    free(deleted);
}
```

```
void delete_neighbors(struct node *delete_node) {
    if (delete_node == NULL) {
        return;
    }

    struct node *left_node = delete_node->llink;
    struct node *right_node = delete_node->rlink;

    if (left_node != NULL && left_node != delete_node) {
        left_node->llink->rlink = delete_node;
        delete_node->llink = left_node->llink;
        free(left_node);
    }

    if (right_node != NULL && right_node !=
delete_node) {
        right_node->rlink->llink = delete_node;
        delete_node->rlink = right_node->rlink;
        free(right_node);
    }
}
```

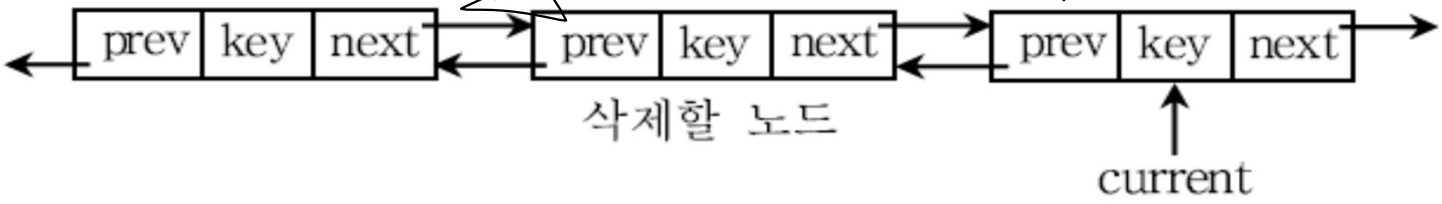


왼쪽이 널일때

다른 문제들:
(1) deleted 노드의 이웃 노드 삭제
(2) 원형 리스트가 아닌 이중 연결 chain 에서 노드 삭제

연습 문제

- 포인터 current가 가리키는 노드의 왼쪽 노드를 삭제할 때 수행할 C 언어 문장을 순서대로 바르게 나열한 것은?



① current->next = current->next->next;
current->next->prev = current;

② current->next->prev = current;
current->next = current->next->next;

③ current->prev = current->prev->prev;
current->prev->next = current;

④ current->prev->next = current;
current->prev = current->prev->prev;