제 4 장. 리스트

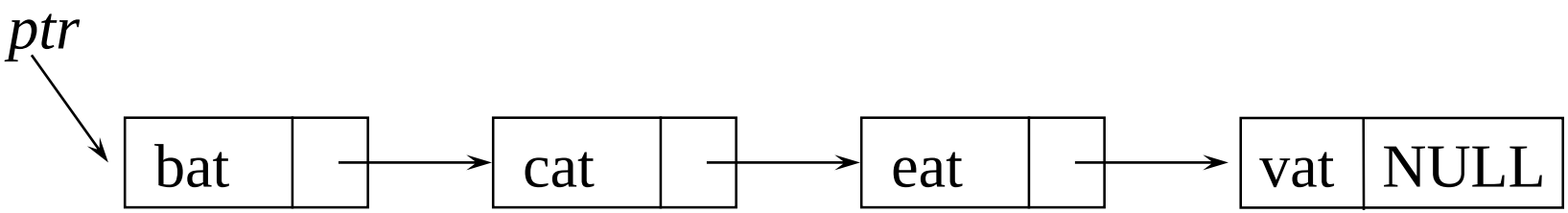
- 순차적 표현
  - 데이터 객체의 연속된 원소들이 일정한 거리만큼 떨어져서 저장
  - 예) (BAT,CAT,EAT,FAT,HAT,JAT,LAT,MAT,OAT,...)에 GAT 삽입, LAT 삭제 시 많은 이동 필요

- 연결된(linked) 표현
  - 연속된 원소들이 일정한 거리 만큼 떨어져 있지 않고, 저장 순서가 리스트에 표현된 순서와 다를 수 있음
  - 각 리스트 원소들에 대하여 다음 원소를 접근하기 위해서, 다음 원소를 가리키는 포인터(링크)를 이용
  - 노드 = 데이터 + 링크

- 포인터 복습
- &: 주소연산자, *: 역참조 연산자

```
int i, *pi ;
pi = &i ; i = 10 ; *pi = 10 ;

int *pi ;
float *pf ;
pi = (int *) malloc (sizeof(int)) ;//#include <malloc.h>
pf = (float *) malloc (sizeof(float)) ;
*pi = 1024 ;
*pf = 3.14 ;
printf (“an integer = %d, a float = %f₩n”, *pi, *pf) ;
free (pi) ;
free (pf) ;
```

- 단순 연결 리스트



  - 화살표는 노드들이 순차적으로 저장되어 있지 않고, 실행할 때마다 바뀔 수 있음을 나타냄

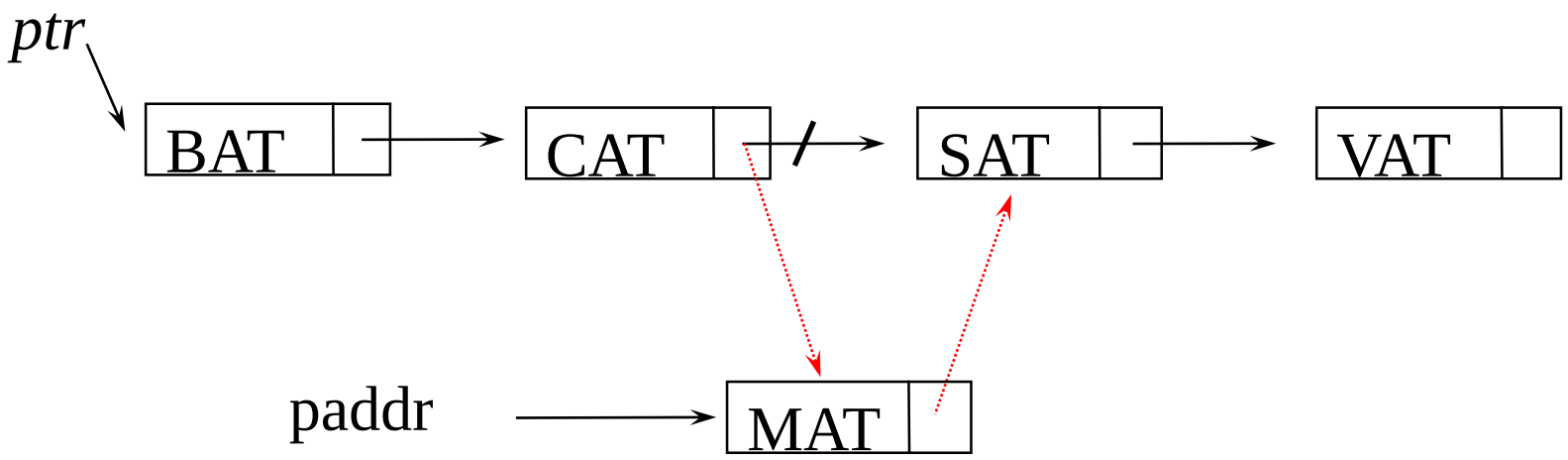
- cat과 sat사이에 mat을 삽입하는 과정

(1) 사용하고 있지 않는 노드를 가져옴, 이 주소를 paddr

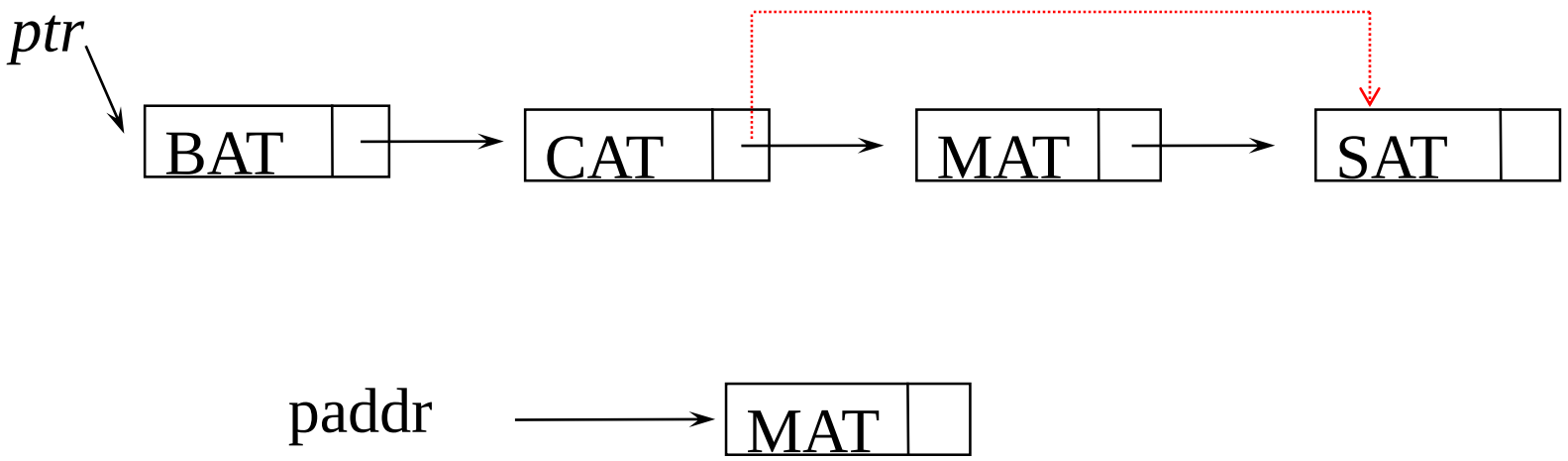
(2) data 필드에 mat를 저장

(3) paddr의 링크 필드를 현재 cat를 가진 노드의 링크 필드내의 주소를 가리키도록 한다.

(4) cat를 가진 노드의 link 필드에 paddr를 저장



- mat을 삭제하는 과정
  - mat 바로 앞의 원소 cat를 찾아서 그 링크 필드값을 mat의 링크 필드값으로 대체



- 연결 리스트 구성을 위한 기능
  - 노드의 구조 정의 : 자체참조 구조
  - 노드 생성 방법 : malloc
  - 노드 삭제 방법 : free

예제 [at로 끝나는 단어 리스트]

```
typedef struct list_node *list_pointer;
typedef struct list_node {
        char data[4];    //데이터 필드
        list_pointer link; //링크 필드
        } ;
list_pointer ptr = NULL;
```

- 공백 리스트 생성

```
list_pointer ptr = NULL;
```

- 공백 리스트 검사

```
#define IS_EMPTY(ptr) (!(ptr))
```

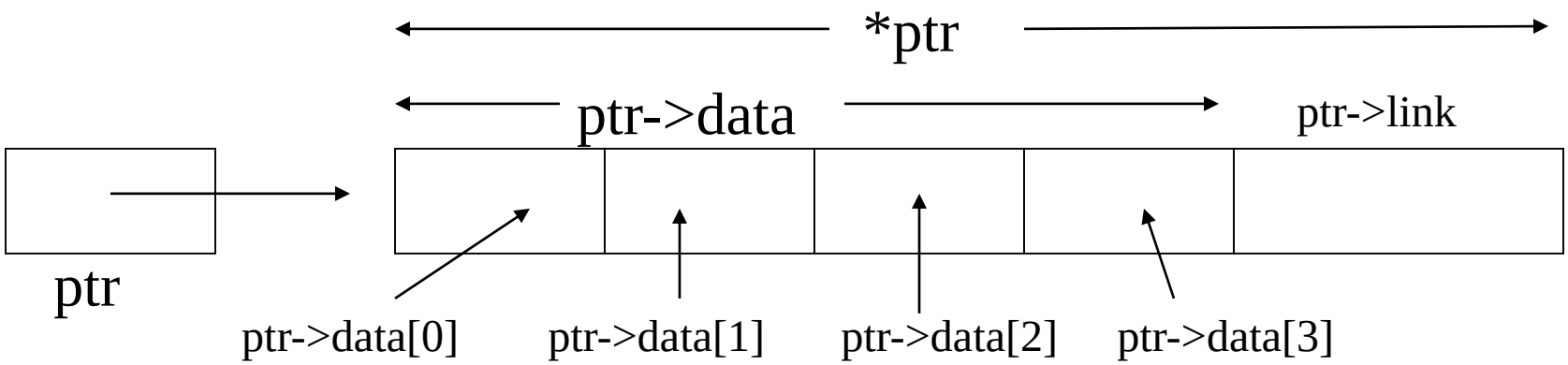
- 새 노드 생성

```
ptr = (list_pointer) malloc(sizeof(list_node));
```

- 노드 필드에 값 지정
  - 구조 멤버(*structure member*) 연산자 사용: '→'

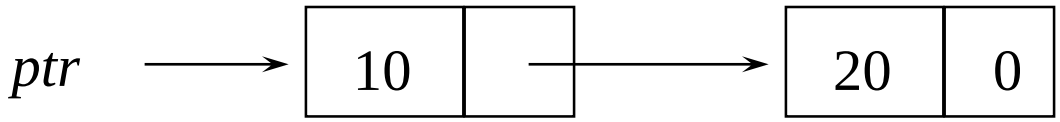
e->name ≡ (*e).name

```
strcpy(ptr->data,"bat");
ptr->link = NULL;
```



예제 [2-노드 연결 리스트]

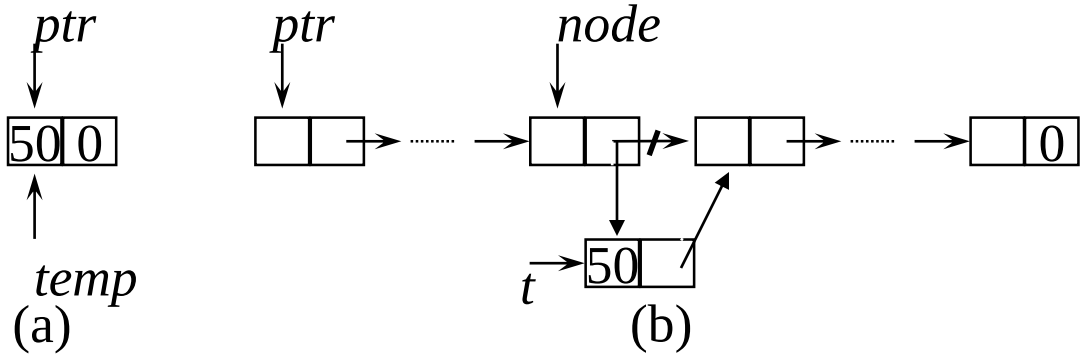
```
typedef struct list_node *list_pointer;
typedef struct list_node {
        int data;
        list_pointer link;
} ;
list_pointer ptr = NULL;

list_pointer create2()
{ /* 두개의 노드를 가진 연결 리스트의 생성 */
   list_pointer first, second;
   first = (list_pointer)malloc(sizeof(list_node));
   second = (list_pointer)malloc(sizeof(list_node));
   second->link = NULL;
   second->data = 20;
   first->data = 10;
   first->link = second;
   return first;
}
```



예제 [리스트 삽입]

함수 호출 : *insert(&ptr, node)*

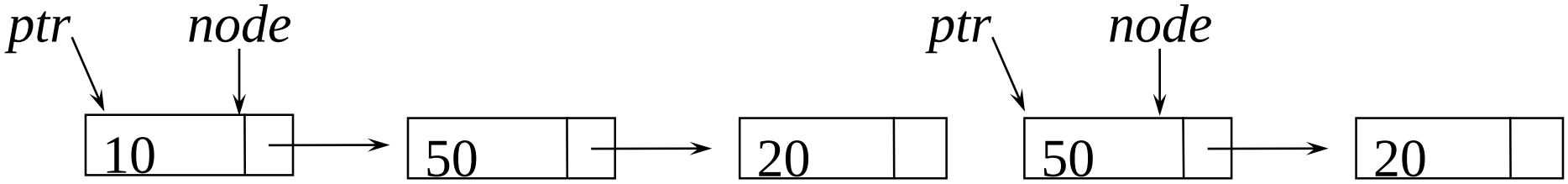
```
#define IS_FULL(ptr) (!(ptr))
void insert(list_pointer *ptr, list_pointer node)
 { /* data=50인 새로운 노드를 리스트 ptr의 node 뒤에 삽입 */
   list_pointer temp;
   temp = (list_pointer)malloc(sizeof(list_node));
   if (IS_FULL(temp)) {
      fprintf(stderr, "The memory is full");
      exit(1);
   }
   temp->data = 50;
   if (*ptr) {
      temp->link = node->link;
      node->link = temp;
   }
   else {
      temp->link = NULL;
      *ptr = temp;
   }
```



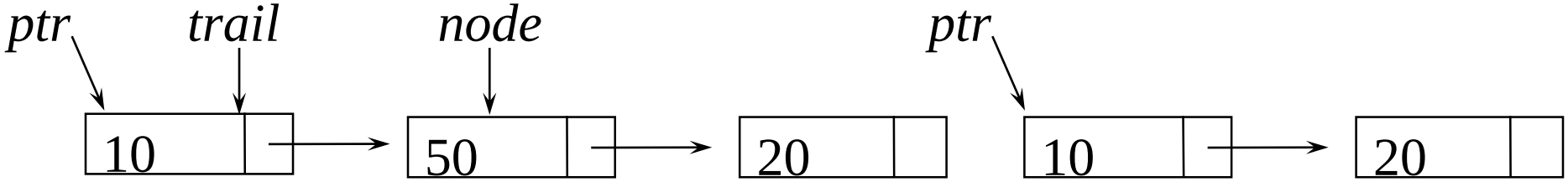
예제 [리스트의 삭제]

```
void delete(list_pointer *ptr, list_pointer trail, list_pointer node)
{  /* node 삭제, trail은 삭제될 node의 선행노드이며 ptr은 리스트의 시작 */
   if (trail)
      trail->link = node->link;
   else
      *ptr = (*ptr)->link;
   free(node);
}
```

delete(&ptr, NULL, ptr) : trail = NULL



delete(&ptr, ptr, ptr->link)

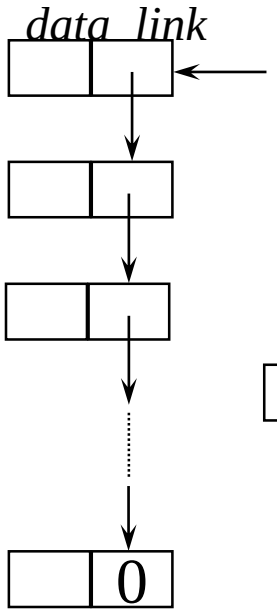


예제 [리스트의 출력]

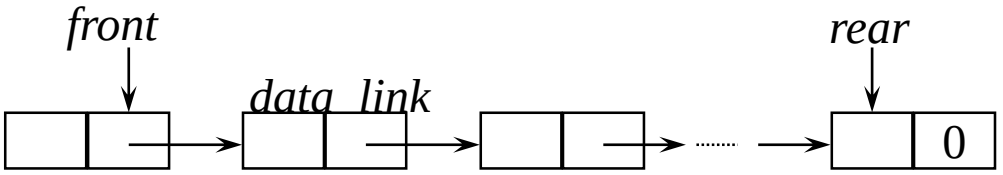
```
void print_list(list_pointer ptr)
{
   printf("The list contains: ");
   for (; ptr; ptr = ptr->link)
      printf("%4d",ptr->data);
   printf(“₩n");
}
```

동적 연결 스택과 큐

- 몇 개의 스택과 큐가 동시에 사용하는 경우에 효율적



(a) 연결 스택



(b) 연결 큐

- 스택은 top에서 삽입/삭제가 용이하고, 큐는 front/rear 에서는 삽입/삭제가 용이

- 여러 개의 스택 선언

```
#define MAX_STACKS 10 /* 스택의 최대 원소수 */
typedef struct {
                int key;
                /* 기타 필드 */
               }  element;
typedef struct stack *stack_pointer;
typedef struct stack {
              element item;
              stack_pointer link;
             } ;
stack_pointer top[MAX_STACKS];
```

- 스택의 초기 조건

  top[i]=NULL, 0≤i<MAX_STACKS
- 경계 조건을 다음과 같이 가정

  i번째 스택이 공백이면, top[i]=NULL

  메모리가 가득차면, IS_FULL(temp)

- add(&top[stack_no],item);

```
void add(stack_pointer *top, element item)
{
/* 스택의 톱에 원소를 삽입 */
   stack_pointer temp = (stack_pointer) malloc(sizeof (stack));
   if (IS_FULL(temp)) {
      fprintf(stderr,"The memory is full");
      exit(1);
   }
   temp->item = item;
   temp->link = *top;
   *top = temp;
}
```

- item=delete(&top[stack_no]);

```
element delete(stack_pointer *top)
{/* 스택으로부터 원소를 삭제 */
   stack_pointer temp = *top;
   element item;
   if (IS_EMPTY(temp)) {
      fprintf(stderr,"The stack is empty");
      exit(1);
   }
   item = temp->item;
   *top = temp->link;
   free(temp);
   return item;
}
```

- 여러 개의 큐의 선언

```
#define MAX_QUEUE 10    /* 큐의 최대 원소수 */
typedef struct queue *queue_pointer;
typedef struct queue {
        element item;
        queue_pointer link;
        } ;
queue_pointer front[MAX_QUEUES], rear[MAX_QUEUES];
```

– 초기 조건

front[i]=NULL, 0≤i<MAX_QUEUES

– 경계 조건

i번째 큐가 공백이면, front[i]=NULL

메모리가 가득차기만 하면, IS_FULL(temp)

- addq(&front,&rear,item);

```
void addq(queue_pointer *front, queue_pointer *rear,
   element item)
{    /* 큐의 rear에 원소를 삽입 */
   queue_pointer temp = (queue_pointer) malloc(sizeof
   (queue));
   if (IS_FULL(temp)) {
      fprintf(stderr,"The memory is full");
      exit(1);
   }
   temp->item = item;
   temp->link = NULL;
   if (*front) (*rear)->link = temp;
   else *front = temp;
   *rear = temp;
}
```

- item=deleteq(&front);

```
element deleteq(queue_pointer *front)
{   /* 큐에서 원소를 삭제 */
   queue_pointer temp = *front;
   element item;
   if (IS_EMPTY(*front))  {
      fprintf(stderr,"The queue is empty");
      exit(1);
   }
   item = temp->item;
   *front = temp->link;
   free(temp);
   return item;
}
```

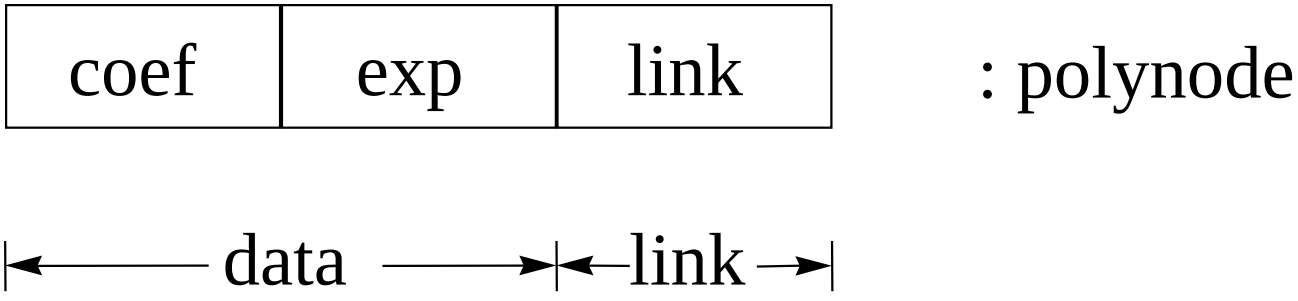
다항식의 연결 리스트 표현

$$A(x) = a_m x^{e_m} + \cdots + a_1 x^{e_1},$$

$$e_m > e_{m-1} > \cdots > e_2 > e_1 \geq 0,$$

$$a_i \neq 0 (m \leq i \leq 1)$$

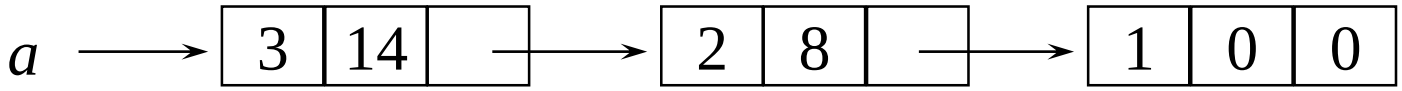
- 다항식의 표현



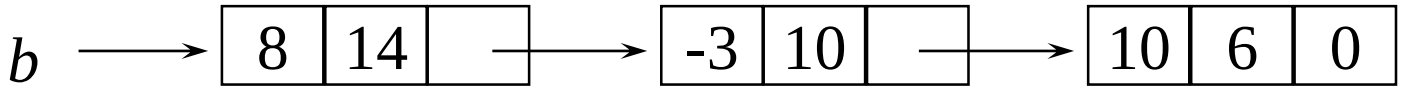
다항식 타입의 선언

```
typedef struct poly_node *poly_pointer ;
typedef struct poly_node {
        int coef ;
        int expon ;
        poly_pointer link ;
        } ;
poly_pointer a, b, d ;
```

- $3x^{14} + 2x^{8} + 1$과 $8x^{14} - 3x^{10} + 10x^{6}$의 다항식 표현

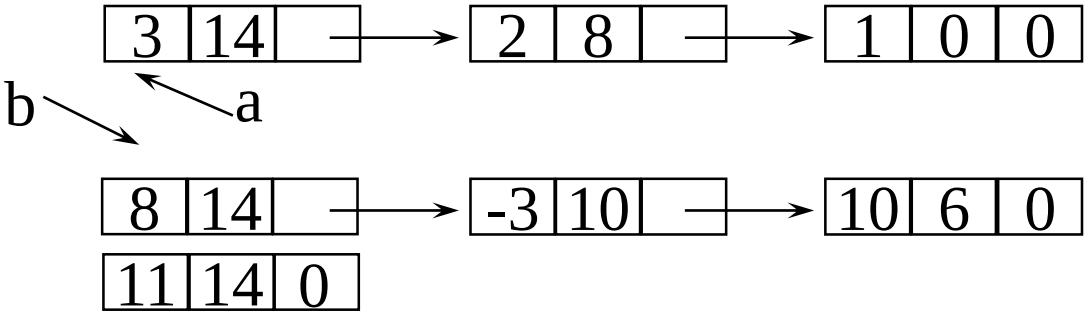


(a)

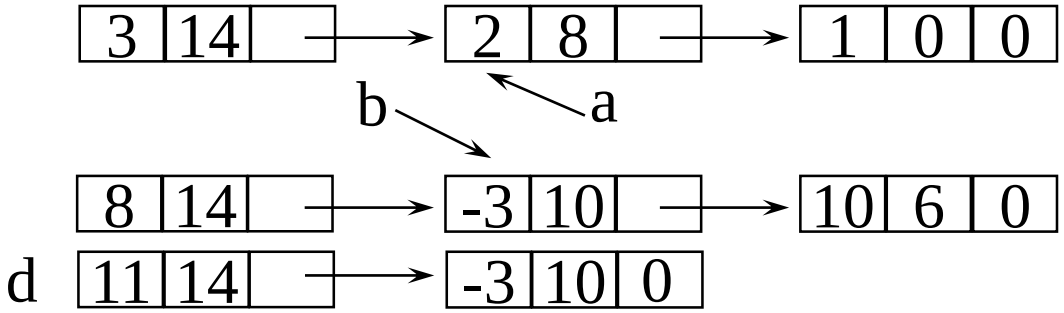


(b)

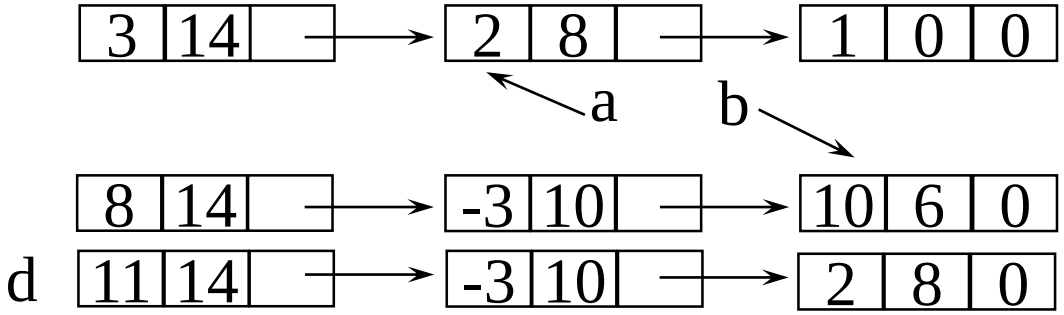
- 다항식의 덧셈 알고리즘
  - 다항식 a, b, d
  - 만일 두 항의 지수가 같으면 계수를 더해서 그 합이 0이 아니면 결과를 다항식 d에 새로운 항을 만든다. 다음 노드를 가리키도록 포인터 a, b를 이동
  - 만약 b의 현재 항의 지수보다 a의 현재 항의 지수가 작으면, b의 항과 똑같은 항을 만들어 결과 다항식 d에 첨가 시키고, 다음 노드를 가리키도록 b를 이동
  - 새로운 노드가 만들어질 때마다 coef, expon 필드에 값을 지정하고, 그것을 d의 끝에 첨가, d의 마지막 노드를 찾는 작업을 피하기 위해, d의 마지막 노드를 가리키는 포인터 rear를 유지
  - 노드를 만들어 d의 마지막에 첨가하기 위해 attach()사용
  - 연산의 편의를 위해 d는 값을 갖지 않는 초기 노드



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



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



(iii) a->exp > b->exp

```
poly_pointer padd(poly_pointer a, poly_pointer b)
{ /* a와 b가 합산된 다항식을 반환 */
   poly_pointer front, rear, temp;   int sum;
   rear = (poly_pointer)malloc(sizeof(poly_node));
   if (IF_FULL(rear)) { fprintf(stderr, "The memory is full");    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;
         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 > a->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;
   /* 필요없는 초기 노드를 삭제 */
   temp = front; front = front->link; free(temp);
   return front;
}
void attach(float coefficient, int exponent, poly_pointer *ptr)
{   /* coef=coefficient이고 expon=exponent인 새로운 노드를 생성하고,
   그것을 ptr에 의해 참조되는 노드에 첨가한다. ptr을 갱신하여 이 새로
   운 노드를 참조하도록  한다. */
   poly_pointer temp;
   temp = (poly_pointer)malloc(sizeof(poly_node));
   if (IS_FULL(temp)){ fprintf(stderr, "The memory is full"); exit(1);  }
   temp->coef = coefficient;
   temp->expon = exponent;
   (*ptr)->link = temp;
   *ptr = temp;
}
```

- padd 에서 총 연산 시간에 영향을 미치는 연산
  (1) 계수 덧셈
  (2) 지수 비교
  (3) d를 위한 새로운 노드 생성

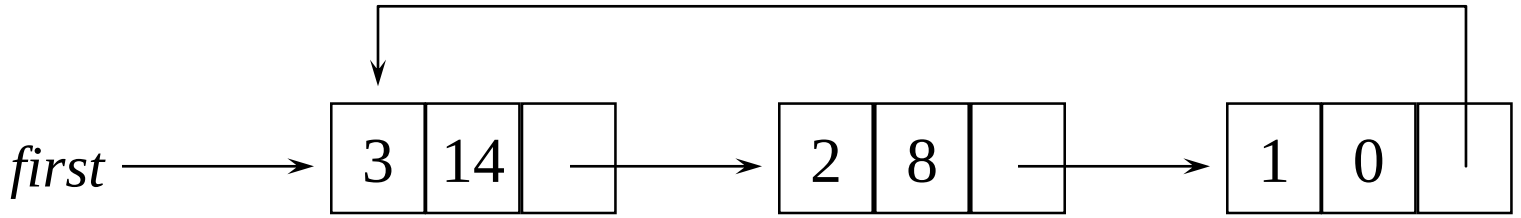
- 계수 덧셈의 횟수
  $0 \le$ 계수 덧셈의 횟수 $\le \min\{m, n\}$
- 지수 비교 회수 최대 $m+n-1$
- 최대 항의 수는 $m+n$이므로 최대 $m+n$개 노드 생성
- 연산 시간은 $O(m+n)$이다.

- 다항식의 제거

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

다항식의 원형 리스트 표현

- ptr=$3x^{14}+2x^{8}+1$의 원형 리스트 표현
  - 효율적인 삭제 알고리즘의 구현이 가능한 표현



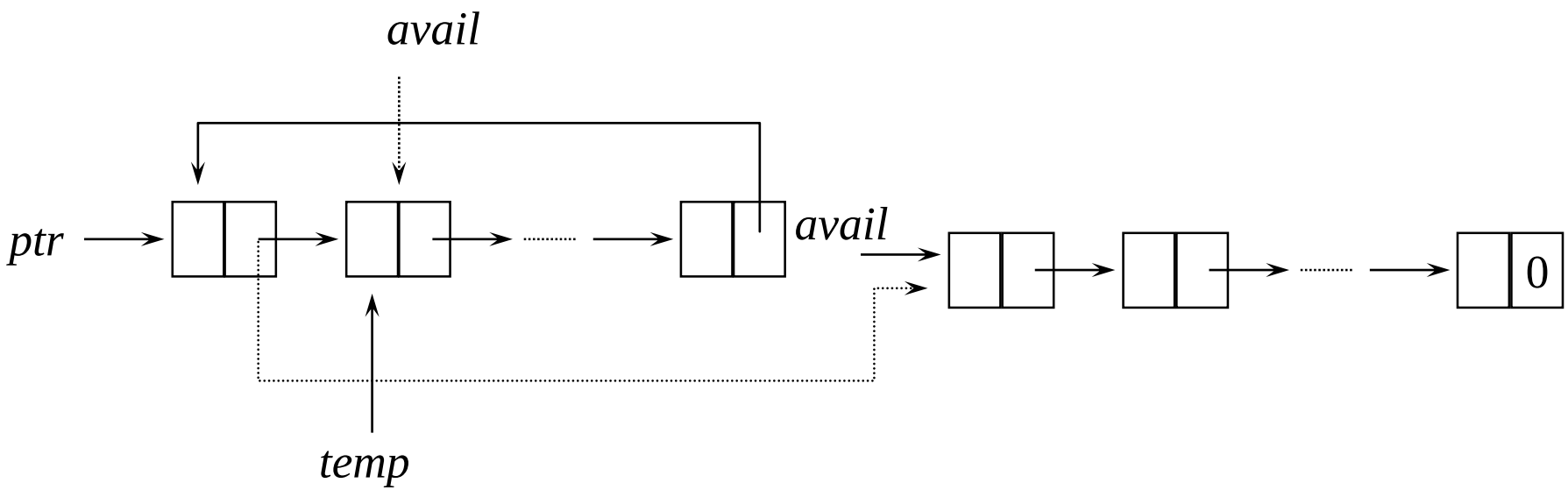
- 원형리스트: 마지막 노드가 첫번째 노드를 가리킴
- 체인: 마지막 노드의 링크 필드값이 NULL

- 가용 공간 리스트(available space list)
  - 삭제된 노드를 체인으로 유지
  - 새로운 노드가 필요하면 이 리스트에서 할당
  - 공백일 때는 함수 malloc 이용하여 새로운 노드를 생성
  - 가용 공간 첫번째 노드를 가리키는 poly_pointer 타입의 변수를 avail이라 하자
  - malloc, free 대신에 노드를 가져오고 반환하는 get_node, ret_node 사용

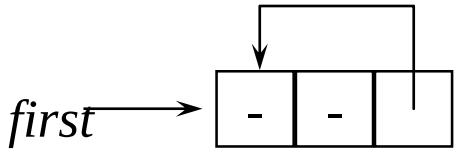
```
void ret_node(poly_pointer ptr)
{  /* 가용 리스트에 노드를 반환 */
  ptr->link = avail;
  avail = ptr;
}
```

```
poly_pointer get_node(void)
{    /* 사용할 노드를 제공 */
  poly_pointer node;
   if (avail) {
      node = avail;
      avail = avail->link;
   }
   else {
      node = (poly_pointer) malloc(sizeof(poly_node));
      if (IS_FULL(node)) {
         fprintf(stderr, "The memory is full");
         exit(1);
      }
   }
   return node;
}
```

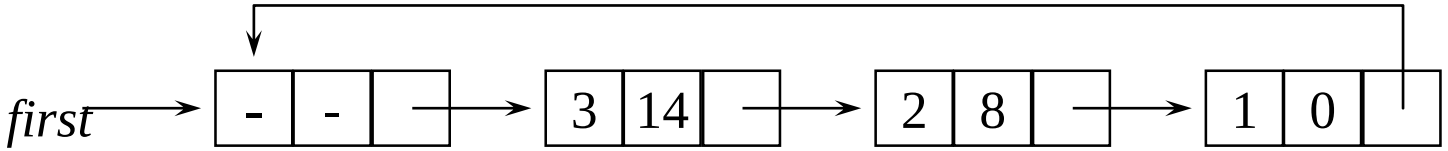
```
void cerase(poly_pointer *ptr)
{  /* 원형 리스트 ptr을 제거 */
   poly_pointer temp;
   if (*ptr) {
      temp = (*ptr)->link;
      (*ptr)->link = avail;
      avail = temp;
      *ptr = NULL;
   }
}
```



- 헤드 노드를 가진 리스트
  - 제로 다항식을 특별한 경우로 처리하지 않기 위해
  - 부가적인 노드로서, expon, coef는 의미가 없음



(a) 제로 다항식



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

- 원형 리스트로 표현된 다항식의 덧셈
  - 헤드노드 expon필드를 –1로 놓으면 알고리즘이 간단해짐
  - a의 모든 노드를 검사한 후에는 starta = a이고 starta→expon = -1이므로 b의 나머지 함들은 switch 문에 의해서 그대로 복사됨

```
poly_pointer cpadd(poly_pointer a, poly_pointer b)
{   /* 다항식 a와 b는 헤드 노드를 가진 단순 연결 원형 리스트이고, a와 b가 합산된 다항식을 반환한다. */
   poly_pointer starta, d, lastd;
   int sum, done = FALSE;
   starta = a;                              /* a의 시작을 기록 */
   a = a->link;    b = b->link; /*a와 b의 헤드노드를 건너 뜀*/
   d = get_node();                  /* 합산용 헤드 노드를 가져 옴 */
   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;
       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 > a->expon */
          attach(a->coef,a->expon,&lastd);
          a = a->link;
    }
  } while (!done);
  lastd->link =d; //마지막 노드의 링크가 첫번째 노드의 주소를 가짐
  return d;
```

제 4.6 동치 관계

- 동치 관계
  - 관계가 대칭적, 반사적, 이행적이면 동치관계라 함

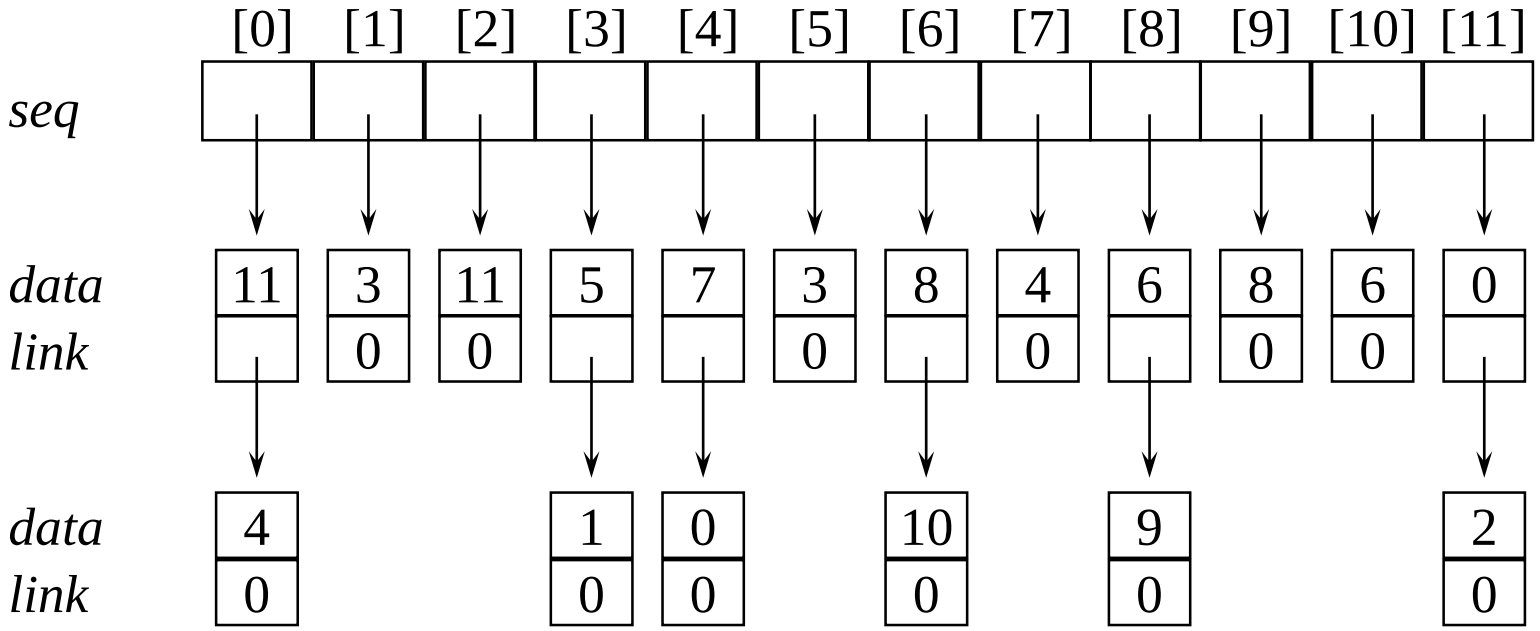
- 동치 부류 (*equivalence class*)
  - x = y 이면 x, y는 같은 동치 부류에 속함
  - 0=4, 3=1, 6=10, 8=9, 7=4, 6=8, 3=5, 2=11, 11=0
  - {0,2,4,7,11}; {1,3,5}; {6,8,9,10}

- 동치 알고리즘
  - Equivalence class를 찾는다.
  - 첫번째 단계에서 모든 동치쌍을 찾아서 저장하고,
  - 두번째 단계에서 0에서부터 시작하여 0과 동치관계인 <0,j>를 찾고, <j, k>이면 k도 0과 같은 부류에 속하므로 출력

  - 동치쌍을 저장하기 위한 배열 pair[m][n]은 기억장소가 낭비되므로, 연결된 리스트 이용
  - Seq[n]: n개의 리스트 헤드 노드를 저장
  - Out[n]은 출력이 안되었을 때 TRUE, 아니면 FALSE

```
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;
      }
}
```

- while loop후에 쌍들이 입력된 리스트



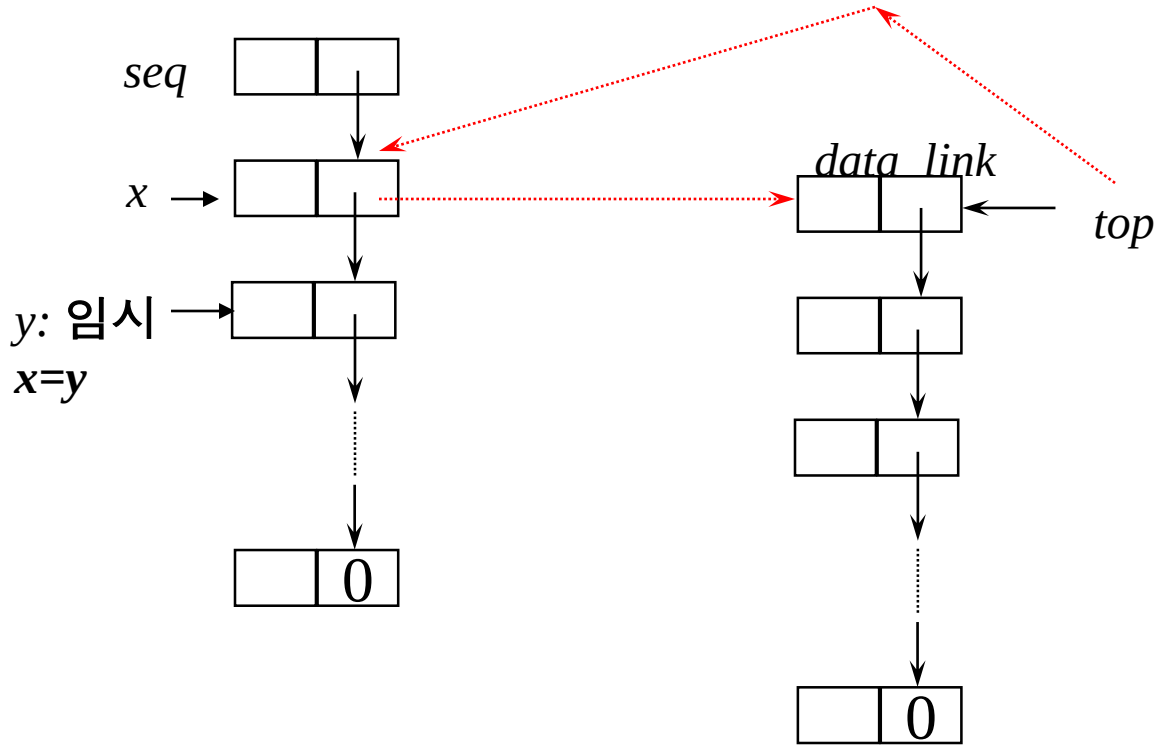
- 리스트 seq[I]의 각 원소를 출력할때, I와 같은 부류에 속하는 나머지 리스트들을 처리하기 위해, 노드들로 구성된 스택을 사용

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

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

```
void main(void)
{
   short int out[MAX_SIZE];
   node_pointer seq[MAX_SIZE];
   node_pointer 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;
    }
```

```
/*1 단계:동치쌍들을 입력 */
 printf("Enter a pair of numbers (-1 -1 to quit): ");
 scanf("%d%d", &i, &j);
  while (i >= 0) {
      x = (node_pointer)malloc(sizeof(node));
      if (IS_FULL(x)) {
         fprintf(stderr, "The memory is full");      exit(1);
      }
      x->data = j;   x->link = seq[i];   seq[i] = x;
      x = (node_pointer)malloc(sizeof(node));
      if (IS_FULL(x)) {
         fprintf(stderr, "The memory is full");      exit(1);
      }
      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);
  }
```

```
for (i = 0; i < n; i++)        /* 2 단계: 동치 부류들을 출력 */
      if (out[i]) {
         printf("₩n New 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 */
         }//end for(;;)
}
```



같은 부류에 속하는
나머지 요소 처리위한 스택

- 동치 프로그램의 분석
  - seq, out 초기화 O(n)
  - 단계 1에서 동치쌍들을 입력하는데 각 쌍마다 상수 시간, 입력쌍의 수가 m이라고 하면 O(m)
  - 단계 2에서 각 노드는 연결 스택에 기껏해야 한 번씩 들어가고, 2m개의 노드가 있다. 바깥쪽의 for 루프는 n번 실행되므로 O(m+n)
  - 따라서, O(m+n)
  - 최소한 m개의 동치쌍과 n개의 값들은 최소한 한번씩 조사해야 함으로, 연산시간은 O(m+n)보다 적은 것은 없으므로 상수시간내에서 최적

이중 연결 리스트

- 단순 연결 리스트의 문제점
  - 링크의 방향으로만 이동 가능
  - 현노드 이전 노드를 찾기 어려움 (삭제시 필요)

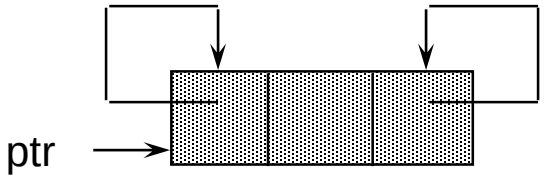
- 이중 연결 리스트
  - 포인터를 양방향으로 이동 가능

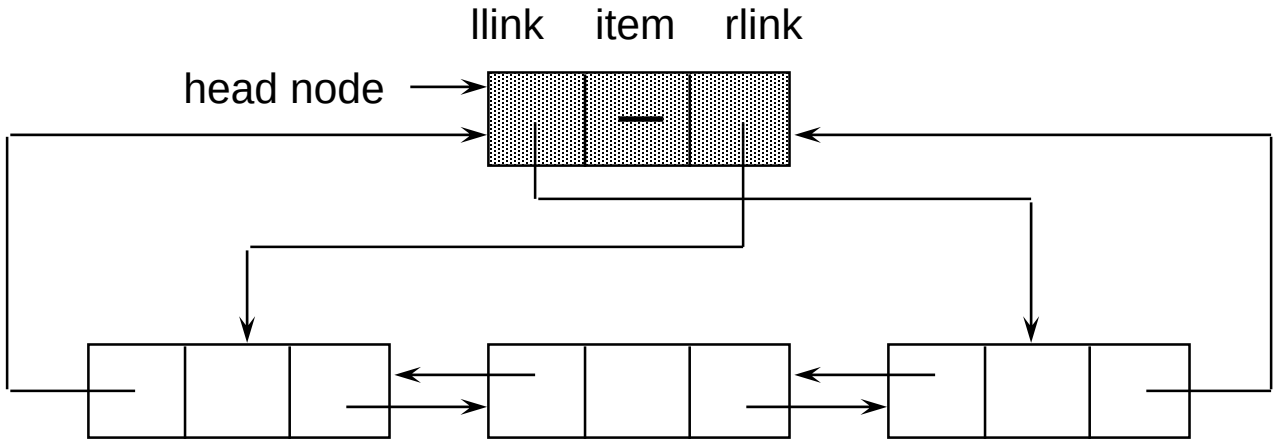
```
typedef struct node *node_ptr;
typedef struct node {
      node_ptr llink;
      element item;
      node_ptr rlink;
};
```

참고 ptr = ptr->llink->rlink = ptr->rlink->llink

이중 연결 원형 리스트

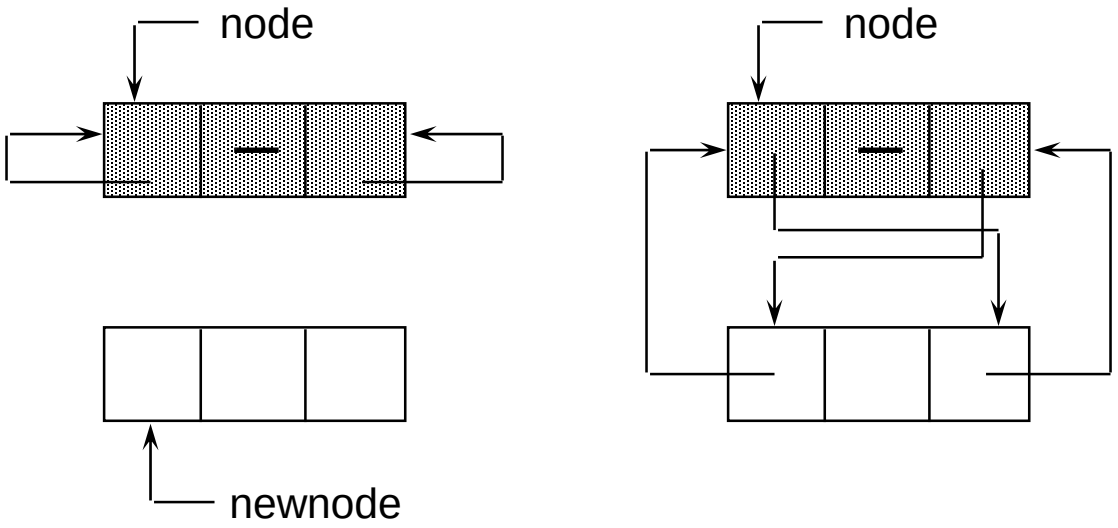
- 헤드 노드 사용
  - 공백인 리스트
  - 연산을 쉽게 수행하게 해줌
  - 데이터 필드에 정보 없음





이중 연결 리스트에서 삽입

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



이중 연결 리스트에서 삭제

```
void ddelete(node_ptr node, node_ptr deleted)
{ // node는 헤드노드
     if (node == deleted)
          printf("Deletion of head node not permitted.₩n");
     else {
          deleted->llink->rlink = deleted->rlink;
          deleted->rlink->llink = deleted->llink;
          free (deleted);
     }
}
```