제 4 장 리스트

□ 순차적 표현 (sequential representation)

- 리스트의 원소들이 기억장소내에서도 리스트에서와 같은 순서로 서로 인접해 있다
- 임의의 위치에 대한 자료의 삽입이나 삭제가 힘들다
- ✓ (bat, cat, eat, fat, hat, jat, lat, mat, oat, pat, rat, sat, tat, vat, wat)
  - add 'gat' & delete 'lat'

□ 연결된 표현 (linked representation)

- 연속된 원소들이 기억장소내의 어떤 곳에나 위치할 수 있다
- 리스트의 원소들을 순서대로 찾기 위해서는 각 원소마다 다음 원소를 가리키는 주소나 위치에 대한 정보를 기억시킨다
- 각 자료 단위마다 다음 원소를 지시하는 포인터(pointer)를 유지

  → 링크(link)

⇒ 연결 리스트 (linked list)

4.1 포인터

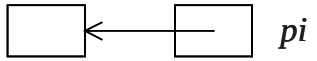
□ C 언어에서의 포인터

- 어떤 타입 *T*에 대해서 *T*의 포인터 타입이 존재한다
  - 포인터 타입의 실제 값은 메모리의 주소
- 2가지 주요 연산
  - &　　: 주소 연산자
  - *　　: 역참조 연산자
- 포인터의 사용은 위험을 초래할 수도 있다
  - 실제로 어떤 대상을 가리키고 있지 않을 때는 값을 *NULL*로 설정하는 것이 바람직하다
  - 포인터 타입 간의 변환을 할 때 명시적인 타입 변환을 사용한다

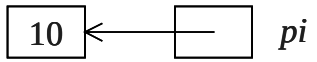
✧ **페이지 138 ~ 140, C 언어에서의 포인터 예**

```
int i, *pi;
float *pf;
```


```
pi = &i;                    /* pi는 i를 가리킨다 */
```

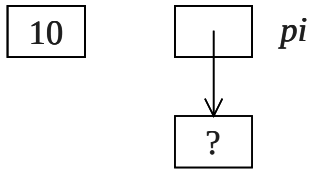



```
i = 10;          *pi = 10;  /* i에 10을 저장 */
```




```
if (pi = NULL)   if (!pi)   /* pi가 널 포인터인지 검사 */
```

```
pi = malloc(sizeof(int));
```




```
pf = (float *)pi;
```

□ 동적 할당 기억 장소의 사용

- 프로그램을 수행하는 도중에 새로운 정보를 저장할 공간이 필요한 경우
- C에서는 히프(heap) 메커니즘을 제공
  - malloc() & free()
- malloc 호출의 반환값은 적당한 크기의 메모리 영역에 대한 첫번째 주소를 가리키는 포인터이다
  - 반환값(포인터)의 타입은 char* 또는 void*이므로 타입 변환을 하는 것이 좋다

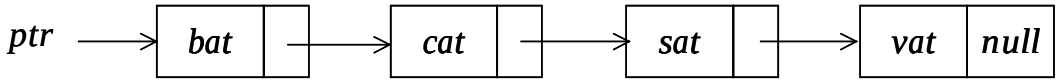
✧ 페이지 140, 프로그램 4.1 : 포인터의 할당과 반환

```
int i, *pi;
float f, *pf;
pi = (int *) malloc(sizeof(int));
pf = (float *) malloc(sizeof(float));
*pi = 1024;
*pf = 3.14;
printf("an integer = %d, a float = %f\n", *pi, *pf);
free(pi);
free(pf);
```

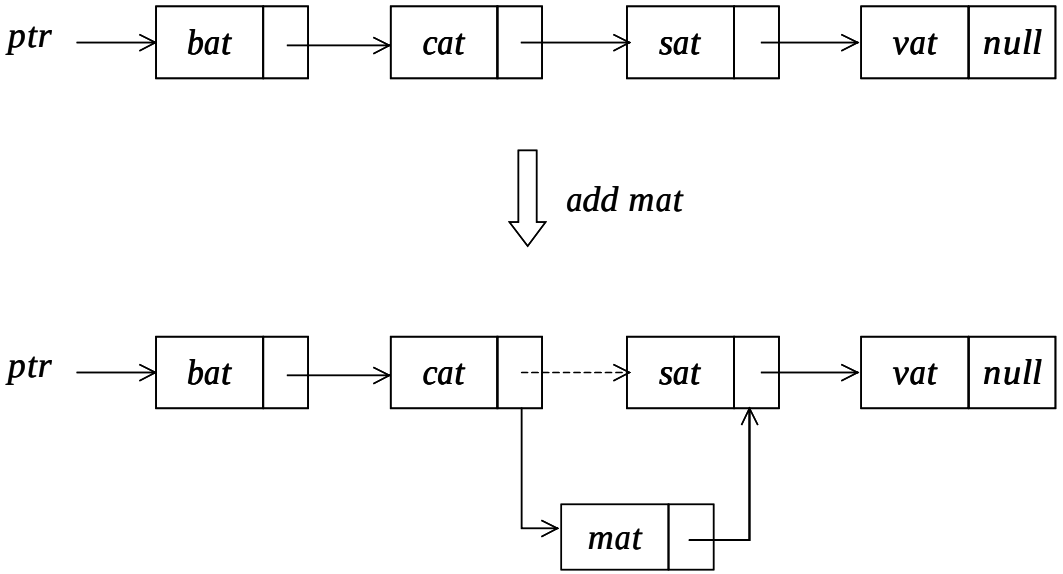
4.2 단순 연결 리스트

▶ **페이지 141, 그림 4.1 : 연결 리스트를 표현하는 일반적인 방법**

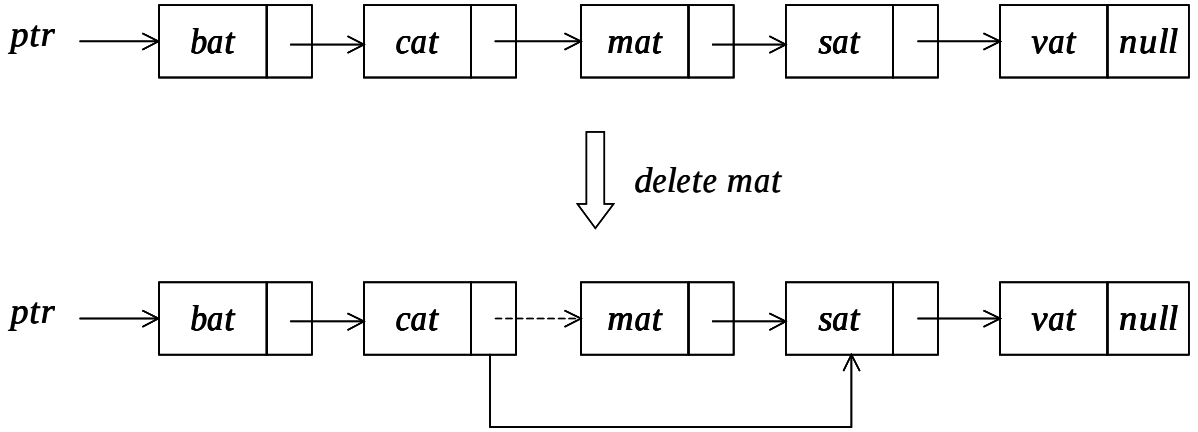
- 노드들은 순차적 위치에 존재하지 않는다
- 노드들의 위치는 실행시마다 바뀔 수 있다



✧ **페이지 142, 그림 4.2 : *cat* 뒤에 *mat*를 삽입**



✧ **페이지 142, 그림 4.3 : 리스트에서 *mat*를 삭제**



□ 연결 리스트 구성을 위한 기능

✧ 페이지 143 ~ 144, 연결 리스트의 구성

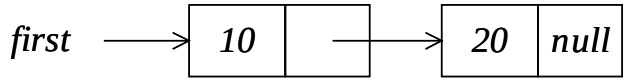
- 노드의 구조 정의
  - 자체 참조 구조를 사용

```
typedef struct list_node *list_pointer;
typedef struct list_node {
        char data[4];
        list_pointer link;
};
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 : '->'
  - *e->name* ≡ (*e).name*;
    strcpy(*ptr->data*,"bat");

▶ **페이지 145, 프로그램 4.2 : 2-노드 리스트의 생성**

& 페이지 145, 그림 4.5 : 2-노드 리스트

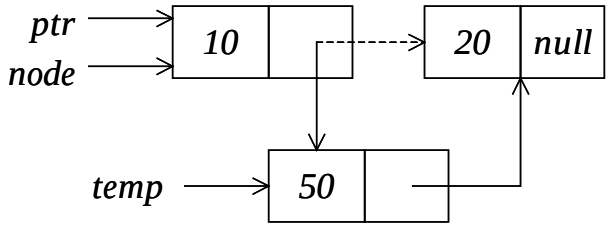



```
list_pointer create2()
{
    /* 두 개의 노드를 가진 연결 리스트의 생성 */
    list_pointer first, second;
    first = (list_pointer)malloc(sizeof(list_node));
    second = (list_pointer)malloc(sizeof(list_node));
    second->link = NULL;
    second->data = 20;
    first->data = 10;
    first->link = second;
   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

▶ **페이지 146, 프로그램 4.3 : 리스트의 앞에 단순 삽입**

& 페이지 147, 그림 4.6 : 함수 호출 (insert(&*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\n");
        exit(1);
    }
    temp->data = 50;
    if (*ptr) {
        temp->link = node->link;
        node->link = temp;
    }
    else {
        temp->link = NULL; *ptr = temp;
    }
}
```

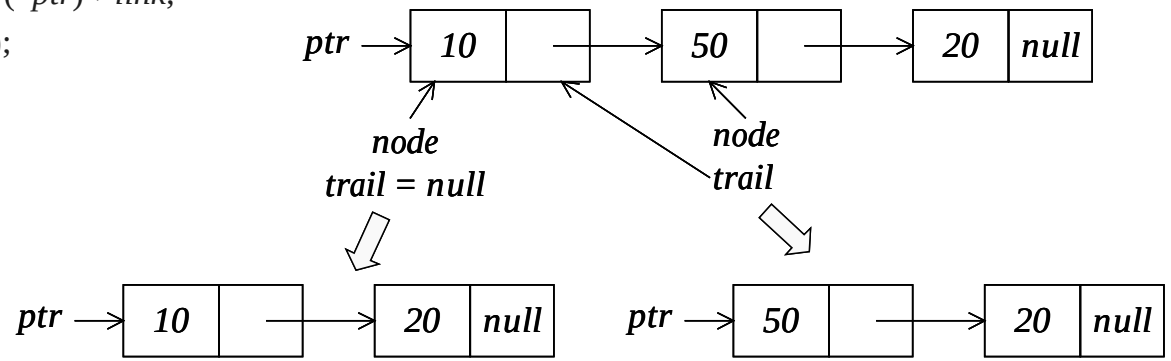


▶ **페이지 148, 프로그램 4.4 : 리스트에서의 삭제**

& 페이지 147, 그림 4.7 : 함수 호출 (delete(&*ptr*, *NULL*, *ptr*))

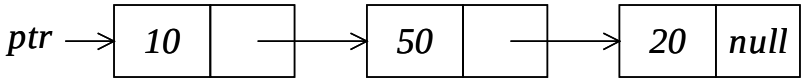
& 페이지 147, 그림 4.8 : 함수 호출 (delete(&*ptr*, *ptr*, *ptr->link*))

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



▶ 페이지 148, 프로그램 4.4 : 리스트의 출력

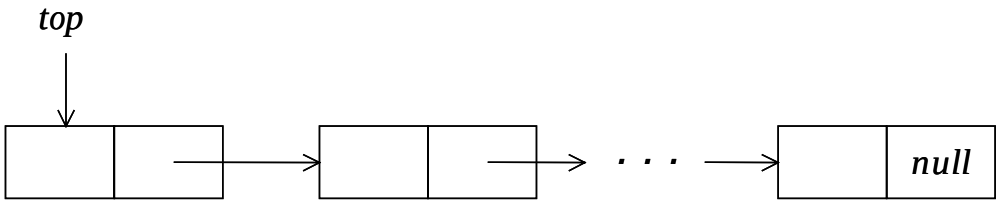
```
void print_list(list_pointer ptr)
{
    printf("The list contains: ");
    for ( ; ptr; ptr = ptr->link)
        printf("%4d", ptr->data);
    printf("\n");
}
```



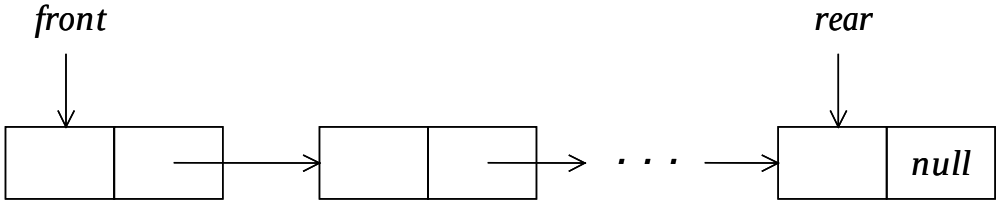
4.3 동적 연결 스택과 큐

▶ **페이지 150, 그림 4.10 : 연결된 스택과 큐**

- 링크 필드를 위해서 기억장소가 더 사용되나 2배를 넘지 않는다



연결된 스택



연결된 큐

□ 장점

- 같은 배열 안에 복잡한 여러 개의 리스트를 표현할 수 있다
- 리스트를 처리하는데 필요한 연산시간이 순차적 표현을 처리하는데 필요한 시간보다 적다

□ *n*개의 스택을 동시에 정의하는 경우

✧ **페이지 150, 선언문 및 초기 조건**

```
#define MAX_STACKS 10 /* 스택의 최대 수 */
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

$top[i] = NULL, \quad 0 \le i < MAX\_STACKS$

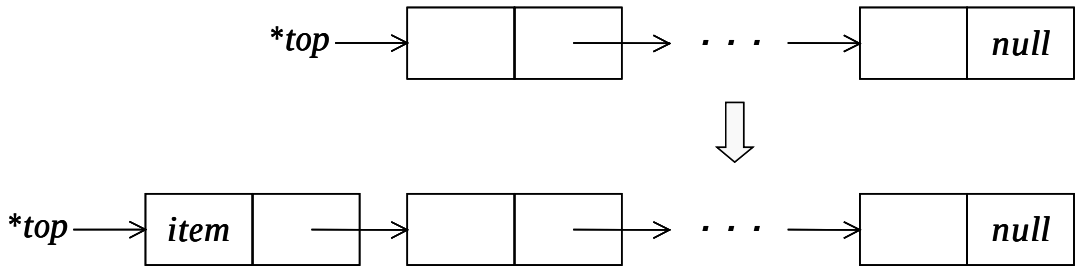
✓경계 조건

*i*번째 스택이 공백이면, $top[i]=NULL$

메모리가 가득차면, IS_FULL(*temp*)

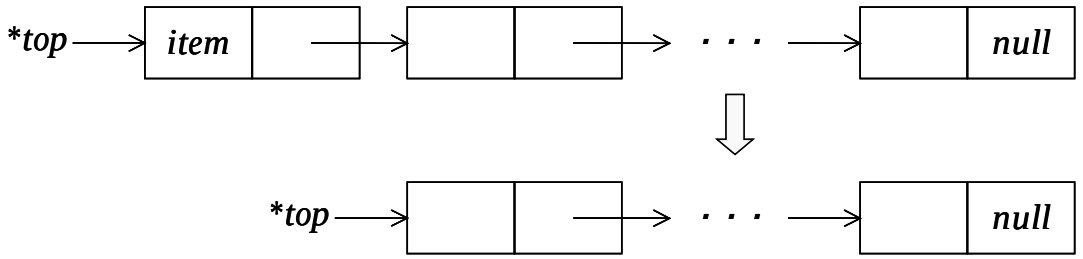
✧ **페이지 151, 프로그램 4.6 : 연결된 스택에서의 삽입**

```
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\n");
        exit(1);
    }
    temp->item = item;
    temp->link = *top;
    *top = temp;
}
```



✧ **페이지 151 ~ 152, 프로그램 4.7 : 연결된 스택에서의 삭제**

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



□ *n*개의 큐를 동시에 정의하는 경우

✧ **페이지 152, 선언문 및 초기 조건**

```
 #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, $0 \leq i <$MAX_QUEUES

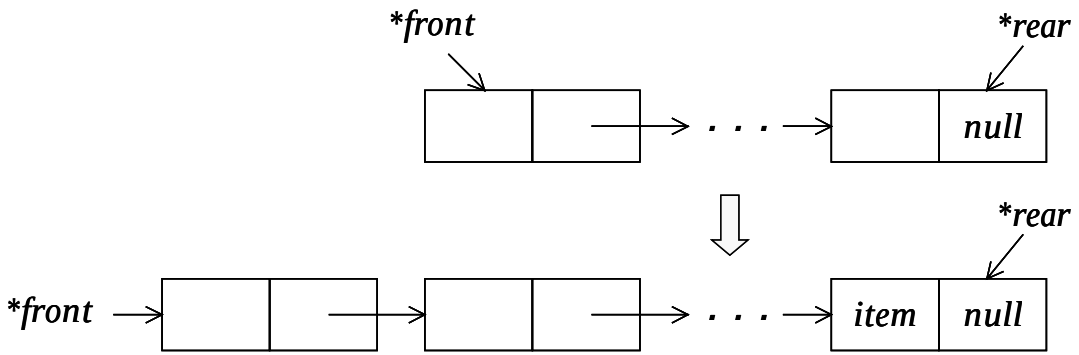
✓ 경계 조건

*i*번째 큐가 공백이면, *front*[*i*]=NULL

메모리가 가득차기만 하면, IS-FULL(*temp*)

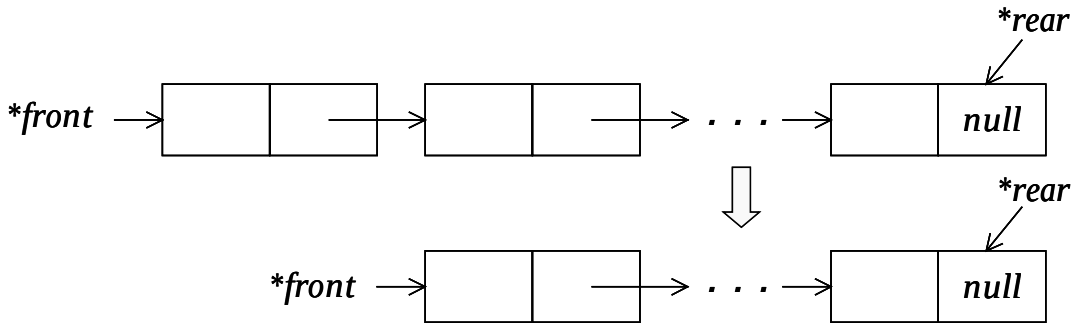
✧ **페이지 152 ~ 153, 프로그램 4.8 : 연결된 큐의 rear에 삽입**

```
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\n");
        exit(1);
    }
    temp->item = item;
    temp->link = NULL;
    if (*front) (*rear)->link = temp;
    else *front = temp;
    *rear = temp;
}
```



✧ **페이지 153, 프로그램 4.9 : 연결된 큐의 front로부터 삭제**

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



4.4 다항식

4.4.1 다항식의 단순 연결 리스트 표현

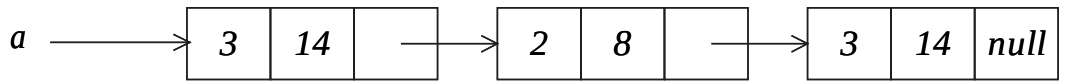
□ 심볼로 표시된 다항식을 다루는 문제 → 리스트 처리의 전형적인 예

✧ **페이지 154, 타입 선언**

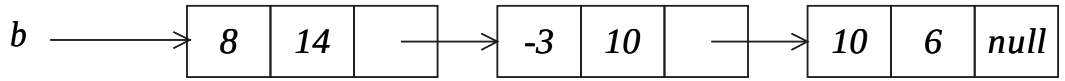
```
typedef struct poly_node *poly_pointer;
typedef struct poly_node {
        int coef;
        int expon;
        poly_pointer link;
};
poly_pointer a, b, d;
```

✧ **페이지 155, 그림 4.11 : 다항식 표현**

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

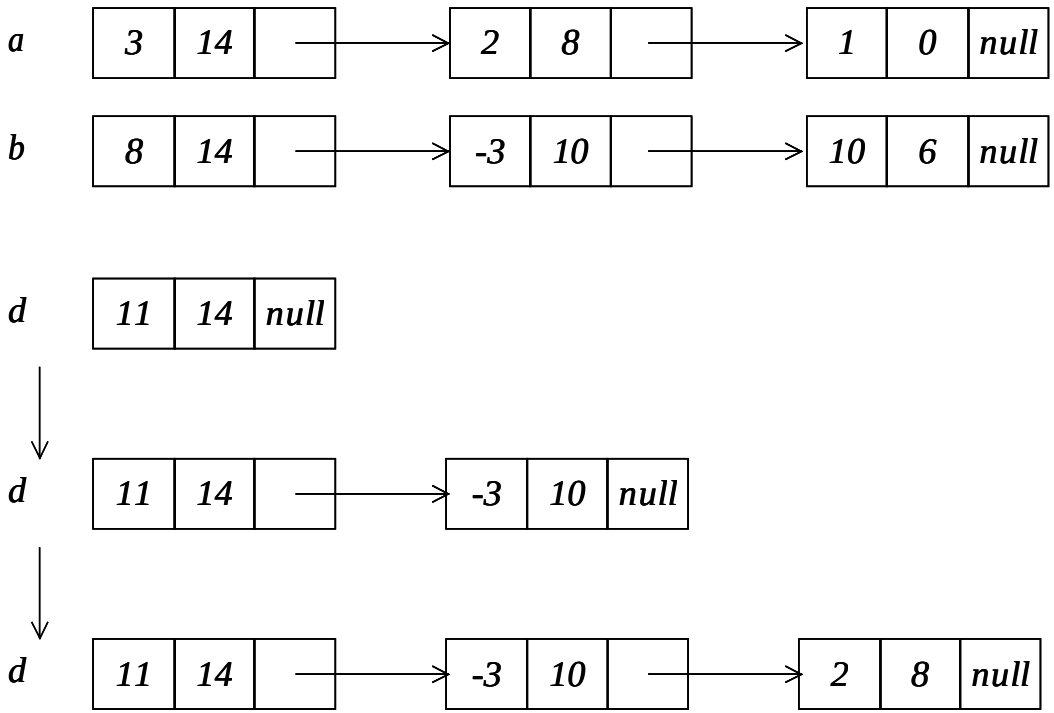


$b = 8x^{14} + 3x^{10} + 10x^6$



4.4.2 다항식의 덧셈

▶ **페이지 156, 그림 4.12 : d = a + b의 처음 세 항을 생성**



▶ **페이지 156 ~ 157, 프로그램 4.10 : 두 다항식의 덧셈**

& 페이지 158, 프로그램 4.11 : 리스트의 끝에 노드를 첨가

```
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\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;
            case 0: /* a->expon = b->expon */
                sum = a->coef + b->coef;
                if (sum) attach(sum, a->expon, &rear);
                a = a->link; 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 새로운 노드를
       참조하도록 한다. */
    poly_pointer temp;
    temp = (poly_pointer)malloc(sizeof(poly_node));
    if (IS_FULL(temp)) {
        fprintf(stderr, "The memory is full\n");
        exit(1);
    }
    temp->coef = coefficient;
    temp->expon = exponent;
    (*ptr)->link = temp;
    *ptr = temp;
}
```

4.4.3 다항식의 제거

□ 다항식의 다양한 연산을 위해 임시로 생성된 다항식의 제거가 필요 !

▶ 페이지 160, 프로그램 4.12 : 다항식의 삭제

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

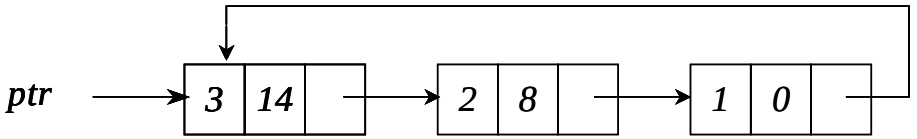
4.4.4 다항식의 원형 연결 리스트 표현

□ 원형 리스트(circular list) ↔ 체인(chain)

- 마지막 노드가 리스트의 첫번째 노드를 가리키도록 한다

✧ **페이지 160, 그림 4.13 : 원형 리스트 표현**

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



- 리스트(다항식)의 모든 노드를 효과적으로 반환할 수 있다
  - 참고) 다항식의 제거 (앞 페이지)

□ 가용 공간 관리

- 원형 리스트를 사용한 가용 공간 리스트(available space list) 유지
  - malloc, free를 대신하여 get_node, ret_node 함수 사용

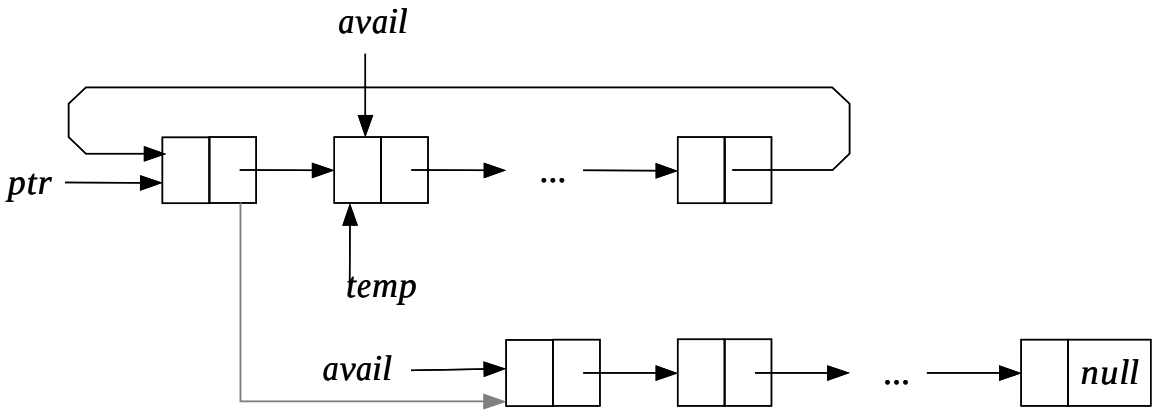
✧ 페이지 161, 프로그램 4.13 : 함수 get_node

```
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\n");
            exit(1);
        }
    }
    return node;
}
```

✧ **페이지 161, 프로그램 4.14 : 함수 ret_node**

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

✧ **페이지 162, 그림 4.14 : 가용 리스트에 원형 리스트를 반환**



✧ **페이지 161 ~ 162, 프로그램 4.15 : 원형 리스트의 제거**

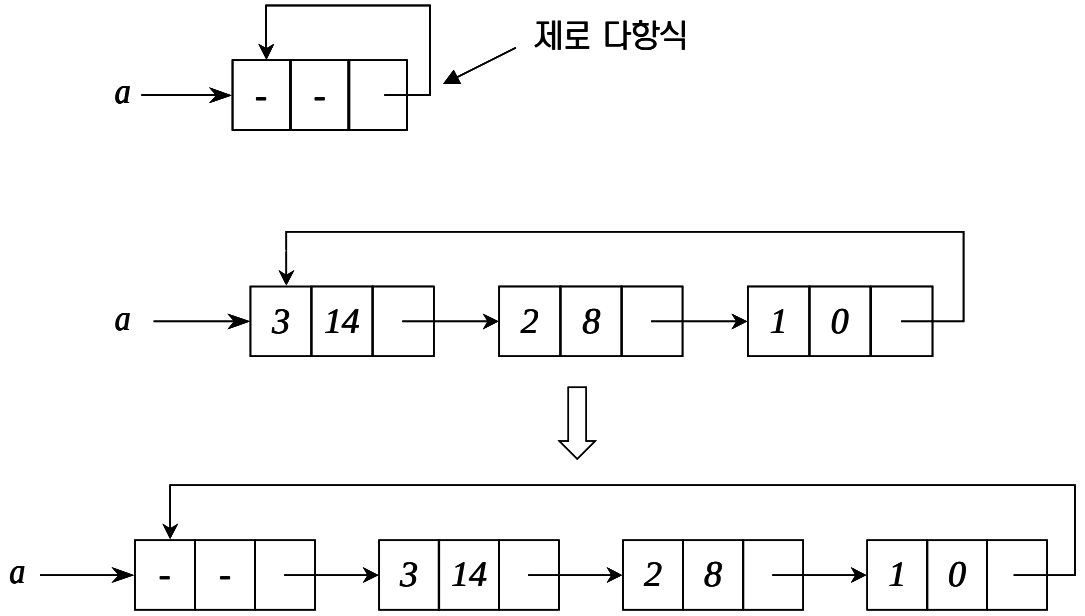
```
void cerase(poly_pointer *ptr)
/* 원형 리스트 ptr을 제거 */
{
    poly_pointer temp;
    if (*ptr) {
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```

– 체인 형태 대신에 원형의 리스트를 이용하여 훨씬 효과적으로 반환

□ 헤드 노드의 사용

◆ 제로 다항식을 특별히 처리하지 않아도 된다

✧ **페이지 162, 그림 4.15 : 다항식 표현**



✧ **페이지 164, 프로그램 4.16 : 원형 리스트로 표현된 다항식의 덧셈**

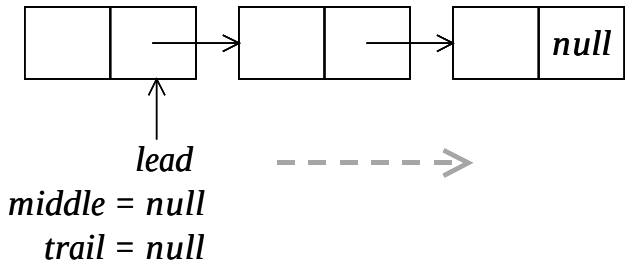
```
poly_pointer cpadd(poly_pointer a, poly_pointer b)
/* 다항식 a와 b는 헤드 노드를 가진 단순 연결 원형 리스트이고, a와 b가 합산된 다항식을 반환한다. */
{
    poly_pointer starta, d, lastd; int sum, done = FALSE;
    starta = a;                          /* a의 시작을 기록 */
    a = a->link;    b = b->link;         /* a와 b의 헤드 노드를 건너 뜀 */
    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->link;
                }
                break;
            case: 1    /* a->expon > a->expon */
                attach(a->coef, a->expon, &lastd); a = a->link;
        }
    } while (!done);
    lastd->link =d;
    return d;
}
```

4.5 추가 리스트 연산

4.5.1 체인 연산

▶ **페이지 166 ~ 167, 프로그램 4.17 : 단순 연결 리스트의 역순화**

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



▶ 페이지 167, 프로그램 4.18 : 단순 연결 리스트의 연결

```
list_pointer concatenate(list_pointer ptr1, list_pointer ptr2)
/* 리스트 ptr1 뒤에 리스트 ptr2가 접합된 새로운 리스트를 생성한다.
    ptr1이 가리키는 리스트는 영구히 바뀐다.*/
{
     list_pointer temp;
     if (IS_EMPTY(ptr1)) return ptr2;
     else {
          if (!IS_EMPTY(ptr2)) {
               for (temp = ptr1; temp->link; temp = temp->link)
                    ;
               temp->link = ptr2;
          }
          return ptr1;
     }
}
```

4.5.2 원형 연결 리스트 연산

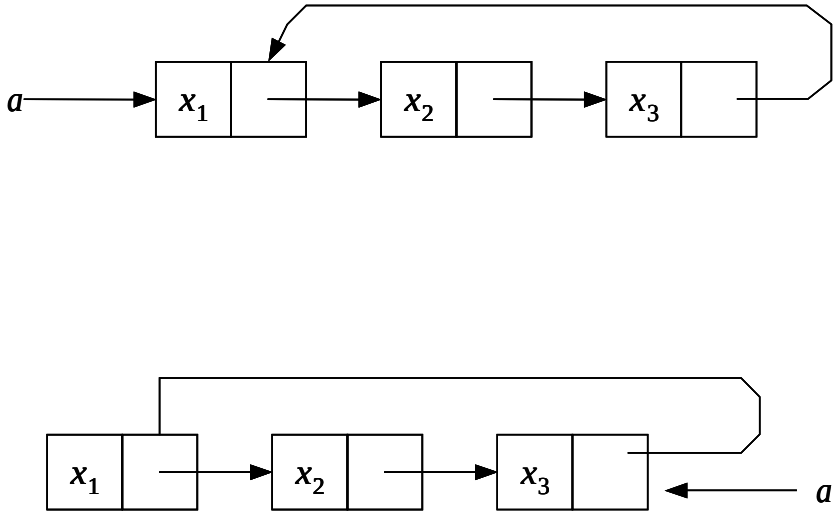
□ 일반 원형 리스트

- 새로운 노드를 이 리스트의 앞에 첨가하려면 마지막 노드의 링크 필드를 변경해야 하므로 이 마지막 노드를 찾을 때까지 모든 노드를 따라 이동해야 한다

⇒ 원형 리스트의 이름이 마지막 노드를 가리키도록 한다

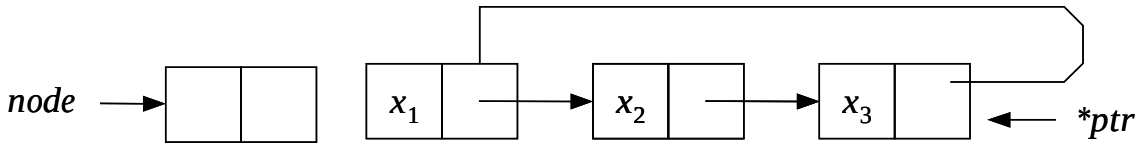
✧ **페이지 168, 그림 4.16 : 원형 리스트 예**

& 페이지 168, 그림 4.17 : 원형 리스트의 마지막 노드에 대한 지시



▶ 페이지 168, 프로그램 4.19 : 리스트의 앞에 삽입하는 프로그램

```
void insert_front(list_pointer *ptr, list_pointer node)
/* ptr이 리스트의 마지막 노드를 가리키는 원형 리스트 ptr의 앞에
   노드를 삽입한다. */
{
    if (IS_EMPTY(*ptr)) {
        /* 리스트가 공백일 경우, ptr이 새로운 항목을 가리키도록 변경 */
        *ptr = node;
        node->link = node;
    }
    else {
        /* 리스트가 공백이 아닌 경우, 리스트의 앞에 새로운 항목을 삽입 */
        node->link = (*ptr)->link;
        (*ptr)->link = node;
    }
}
```



▶ 페이지 169, 프로그램 4.20 : 원형 리스트의 길이 계산

```
int length(list_pointer ptr)
{
/* 원형 리스트 ptr의 길이를 계산한다. */
     list_pointer temp;
     int count = 0;
     if (ptr) {
          temp = ptr;
          do {
               count++;
               temp = temp->link;
          } while (temp != ptr);
     }
     return count;
}
```

4.6 동치 관계

4.6.1 동치 관계와 동치 부류

□ 동치 관계(equivalence relation)의 특성

- 반사적(reflexive)
  - 모든 다각형 x에 대해 $x \equiv x$가 성립
- 대칭적(symmetric)
  - 두 다각형 x, y에 대해 $x \equiv y$이면 $y \equiv x$
- 이행적(transitive)
  - 세 다각형 x, y, z에 대해 $x \equiv y$이고 $y \equiv z$이면 $x \equiv z$

집합 S에 대해 관계 ≡가 대칭적, 반사적, 이행적이면 관계 ≡를 집합 S에 대해 동치 관계 (equivalence relation)라 한다

□ 동치 부류 (equivalence class)

- 집합 S의 두 원소 x와 y에 대하여 $x \equiv y$가 성립하면 x와 y는 같은 동치 부류에 속하며 그 역도 성립한다

✓ $0 \equiv 4, 3 \equiv 1, 6 \equiv 10, 8 \equiv 9, 7 \equiv 4, 6 \equiv 8, 3 \equiv 5, 2 \equiv 11, 11 \equiv 0$

- 동치 부류 → { 0, 2, 4, 7, 11 } ; { 1, 3, 5 } ; { 6, 8, 9, 10 }

4.6.2 동치 결정 알고리즘

▶ **페이지 171, 프로그램 4.21 : 개략적인 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);
}
```

□ 배열 *pairs*[*n*][*m*]의 사용

- 기억장소의 낭비가 심하다
- 새로운 쌍 <*i*, *k*>를 *i*행에 넣기 위해 *i*행 내에서 새로운 빈 열을 찾는데 상당한 시간을 소모

⇒ 연결 리스트의 사용

- 노드는 *data* 필드와 *link* 필드로 구성
- 헤드 노드를 위해 *seq*[*n*] 사용
- 출력 여부를 위해 *out*[*n*] 사용

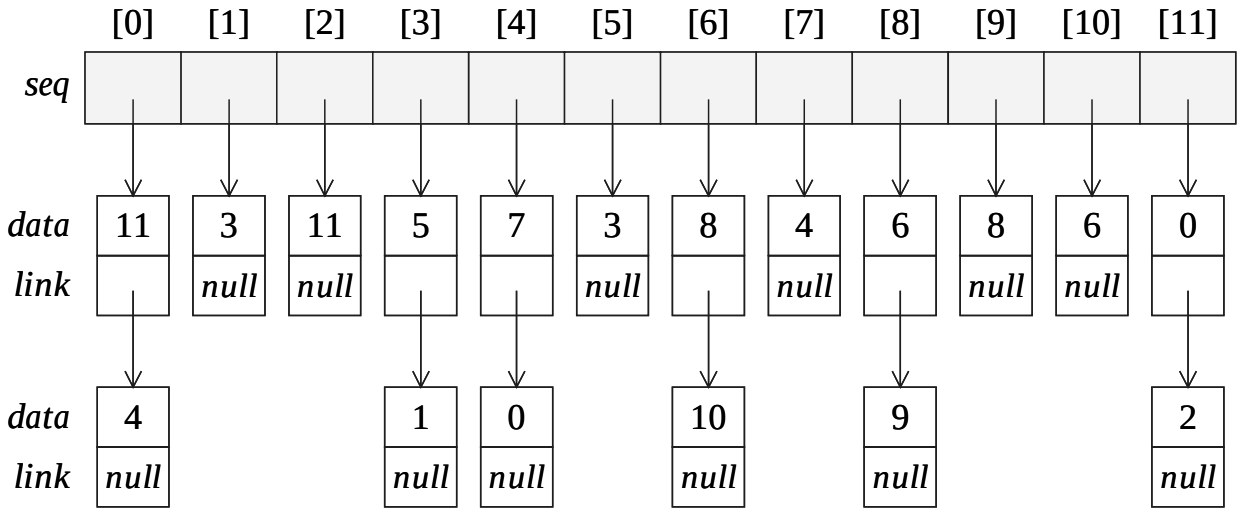
→ *n*이 클수록 유리

▶ 페이지 171 ~ 172, 프로그램 4.22 : 구체적인 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;
        }
}
```

▶ **페이지 172, 그림 4.18 : 쌍들이 입력된 뒤의 리스트**

$0 \equiv 4, 3 \equiv 1, 6 \equiv 10, 8 \equiv 9, 7 \equiv 4, 6 \equiv 8, 3 \equiv 5, 2 \equiv 11, 11 \equiv 0$



▶ 페이지 172 ~ 174, 프로그램 4.23 : 동치 부류를 구하는 프로그램

```
#include <stdio.h>
#inlcude <alloc.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: 동치 쌍들을 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\n");
        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\n");
        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);
}
```

```
    /* 2 단계: 동치 부류들을 출력 */
    for (i = 0; i < n; i++)
        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 */
            }
        }
}
```

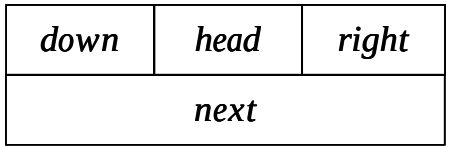
4.7 희소 행렬

□ 연결 리스트 표현

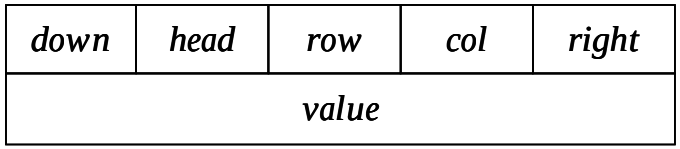
◆ 헤드 노드가 있는 원형 연결 리스트의 사용

✧ **페이지 175, 그림 4.19 : 희소 행렬을 위한 노드 구조**

헤드 노드



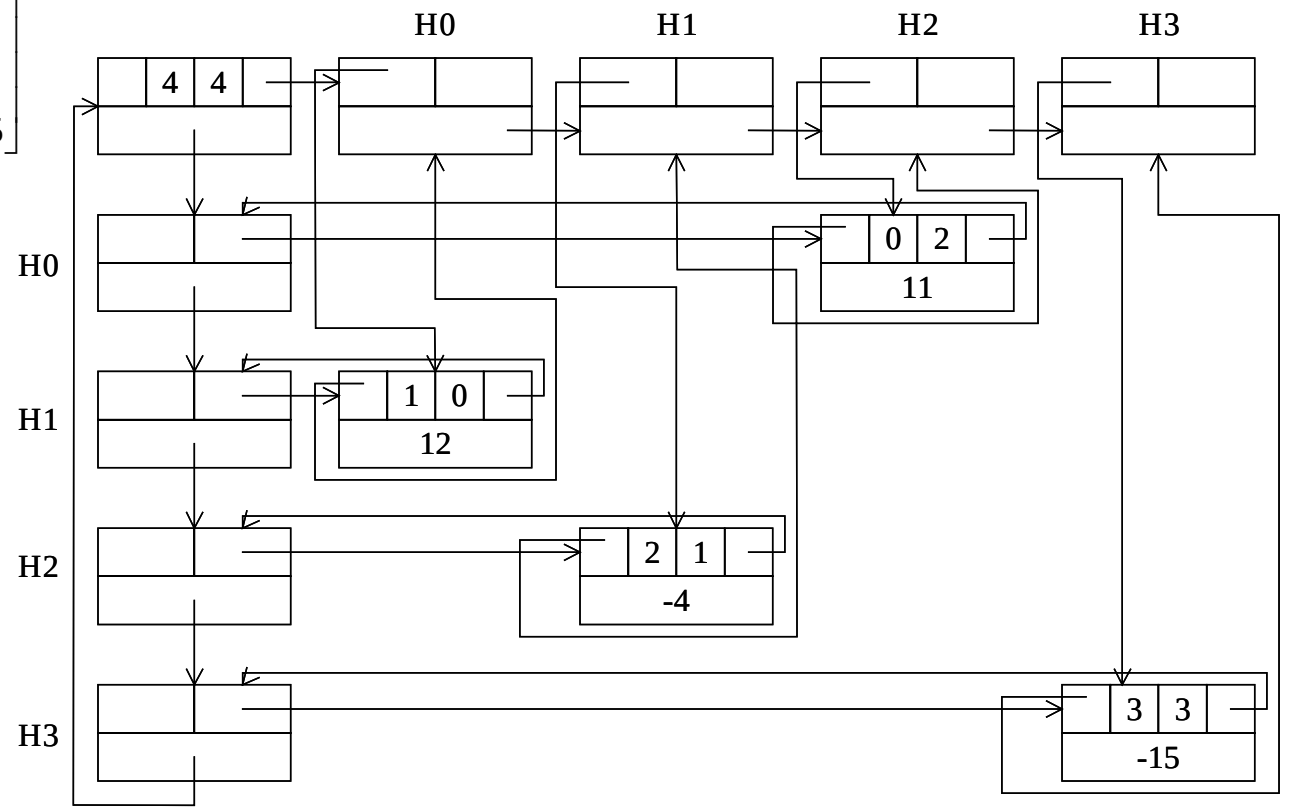
엔트리 노드



✧ **페이지 176, 그림 4.20 : 4 × 4 희소 행렬**

& 페이지 176, 그림 4.21 : 희소행렬의 연결 표현

$$\begin{bmatrix} 0 & 0 & 11 & 0 \\ 12 & 0 & 0 & 0 \\ 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & -15 \end{bmatrix}$$



✧ **페이지 177, 선언문**

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

▶ **페이지 177, 그림 4.22 : 희소 행렬의 입력 예**

& 페이지 178 ~ 179, 프로그램 4.24 : 희소 행렬에서의 읽기

& 페이지 179 ~ 180, 프로그램 4.25 : 새 행렬 노드를 얻음

```
matrix_pointer mread(void)                                          < 4, 4, 4 >
/* 행렬을 읽어 연결 표현으로 구성한다. 전역 보조배열 hdnode를 사용한다. */   < 0, 2, 11 >
{                                                                   < 1, 0, 12 >
    int num_rows, num_cols, num_terms, num_heads, i;                < 2, 1, -4 >
    int row, col, value, current_row;                               < 3, 3, -15 >
    matrix_pointer temp, last, node;

    printf("Enter the number 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;

    /* 헤드 노드 리스트에 대한 헤드 노드를 생성 */
    node = new_node(); node->tag = entry;
    node->u.entry.row = num_rows;  node->u.entry.col = num_cols;
    if (!num_heads) node->right = node;
    else {   /* 헤드 노드들을 초기화한다. */
        for (i = 0; i < num_heads; i++) {
            temp = new_node; hdnode[i] = temp; hdnode[i]->tag = head;
            hdnode[i]->right = temp; hdnode[i]->u.next = temp;
        }
```

```
        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 종료함 */
                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; last = temp;          /* 행 리스트에 연결 */
            hdnode[col]->u.next->down = temp;         /* 열 리스트에 연결 */
            hdnode[col]->u.next = temp;
        }
        last->right = hdnode[current_row];            /* 마지막 행을 종료함 */
        for (i = 0; i < num_cols; i++)                /* 모든 열 리스트를 종료함 */
            hdnode[i]->u.next->down = hdnode[i];
        for (i = 0; i < num_heads - 1; i++)           /* 모든 헤드 노드들을 연결함 */
            hdnode[i]->u.next = hdnode[i + 1];
        hdnode[num_heads - 1]->u.next = node;
        node->right = hdnode[0];
    }
    return node;
}
```

▶ 페이지 180, 프로그램 4.26 : 희소 행렬의 출력

```
void mwrite(matrix_pointer *node)
/* 행렬을 행우선으로 출력한다. */
{
    int i;
    matrix_pointer temp, head = node->right;
    /* 행렬의 차원 */
    printf("\n 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 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 */
    }
}
```

▶ 페이지 181, 프로그램 4.27 : 희소 행렬의 삭제

```
void merase(matrix_pointer *node)
/* 행렬을 삭제하고, 노드들을 히프로 반환한다. */
{
    matrix_pointer x, y, head = (*node)->right;
    int i, num_heads;
    /* 엔트리 노드와 헤드 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 헤드 노드들을 반환한다. */
    y = head;
    while (y != *node) { x = y; y = y->u.next; free(x); }
    free(*node); *node = NULL;
}
```

4.8 이중 연결 리스트

□ 단순연결 선형리스트(singly linked linear list)의 문제점

- 임의의 노드 P에서 링크 방향으로는 쉽게 이동하나 P의 전위 노드(predecessor)를 가려면 리스트의 처음부터 다시 찾아야 한다

→ 임의 노드 삭제에 문제

□ 이중 연결 리스트(doubly linked list)

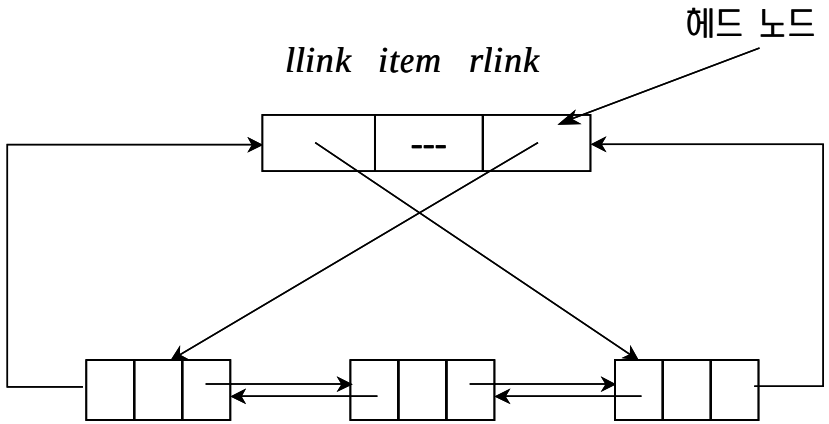
- 노드는 두개의 링크로 구성
  - 후위 방향(RLINK), 전위 방향(LLINK)

✧ **페이지 183, 노드 구조**

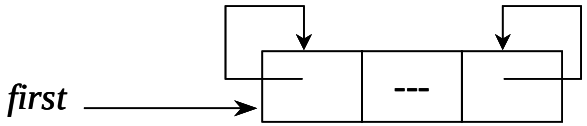
```
typedef struct node *node_pointer;
typedef struct node {
    node_pointer llink;
    element item;
    node_pointer rlink;
}
```

▶ **페이지 183, 그림 4.23 : 헤드 노드를 가진 이중 연결 원형 리스트**

& 페이지 184, 그림 4.24 : 헤드 노드를 가진 공백 이중 연결 리스트



p == p->llink->rlink == p->rlink->llink

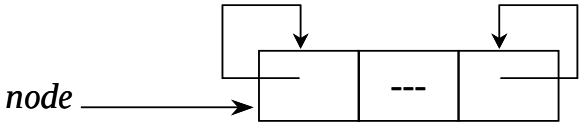


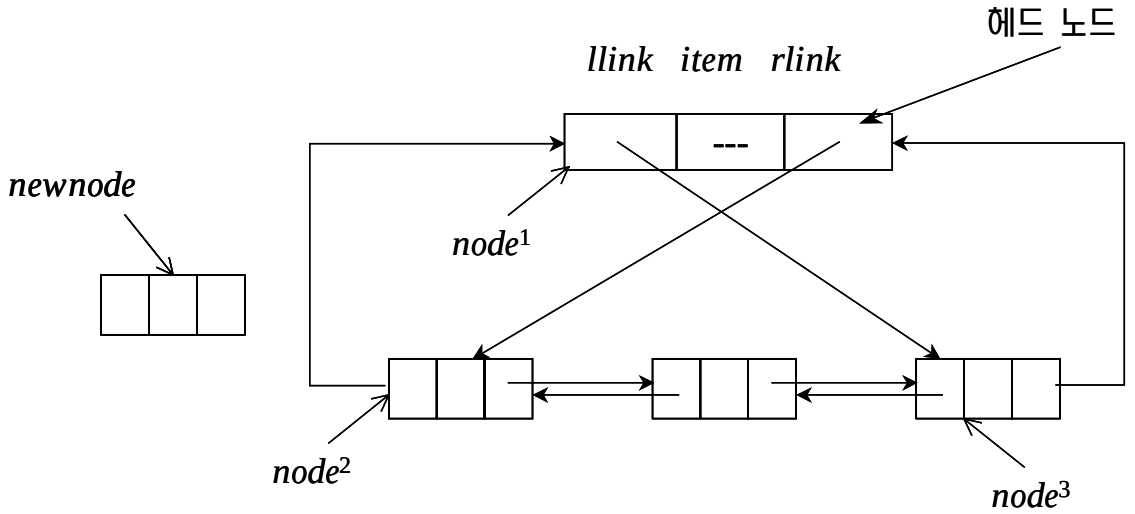
공백 이중 연결 리스트

▶ **페이지 184, 프로그램 4.28 : 이중 연결 원형 리스트에 삽입**

& 페이지 184, 그림 4.25 : 공백 이중 연결 원형 리스트에 삽입

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





▶ **페이지 185, 프로그램 4.29 : 이중 연결 원형 리스트에서의 삭제**

& 페이지 185, 그림 4.26 : 이중 연결 리스트에서의 삭제

```
void ddelete(node_pointer node, node_pointer 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);
    }
}
```