3. 리스트 (Lists)

한국항공대학교 안준선

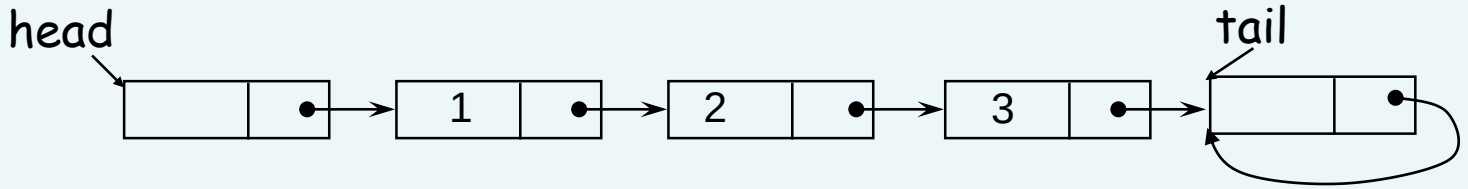
주요 내용

- 연결 리스트
  - 연결 리스트의 특징과 사용법, 관련 함수를 익힌다.
  - 배열과 리스트를 용도에 따라 선택하여 사용할 수 있다.
  - 연결 리스트를 사용한 주소록, 희소행렬, 2차원 희소행렬 응용 프로그램을 이해하고 응용할 수 있다.

- C 프로그래밍
  - 동적 메모리의 특성을 이해하고 관련 함수를 사용할 수 있다.
  - 구조체의 생성 방법과 사용법을 익힌다.
  - 리스트 구조를 위한 포인터 연산을 이해한다.
  - 파일 IO (이진 파일)를 이해하고 사용할 수 있다.

연결 리스트

- 연결 리스트(Linked List)
  - 데이터를 저장한 노드들을 포인터로 연결한 자료구조
  - 동적인 자료구조 : 노드를 리스트에 추가하거나 제거함으로써 자료구조의 크기가 프로그램 실행 중에 달라짐
  - 자료형 T의 원소들을 저장한 리스트에 대한 연산들
    - 리스트 생성 : ( ) ⇢ $List^T$
    - 노드 탐색 : T x $List^T$ ⇢ $Node^T$
    - 노드 삽입 : T x $Node^T$ ⇢$Node^T$, T x T x $List^T$ ⇢ $List^T$
    - 노드 삭제 : T x $List^T$ ⇢ $List^T$
  - 하나의 리스트의 각각의 노드는 연속된 공간이 아니라 힙(heap) 공간에 분산되어 위치된다.

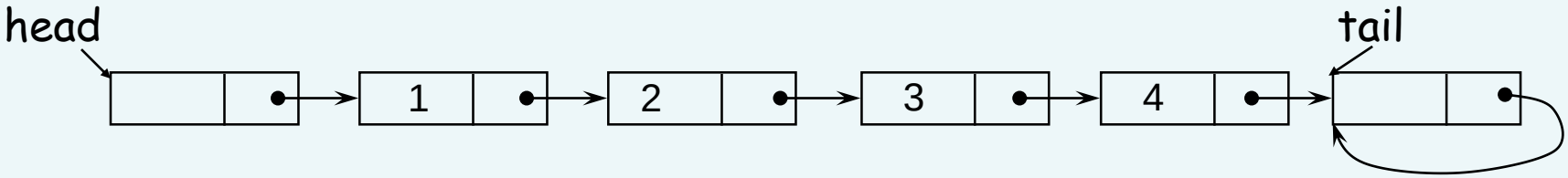


연결 리스트

- 장점
  - 원소의 삽입 삭제 연산이 용이 (<-> 배열: O(n) 시간 소요)
  - 메모리의 낭비를 막을 수 있다. (← 동적인 자료구조)
- 단점
  - 포인터를 위한 공간 필요
  - 동적 메모리의 관리가 필요 (malloc, calloc, free)
  - 리스트 중간의 원소로 직접 접근할 수 없다. (→ 순차접근)
- 종류
  - 단순 연결 리스트
  - 이중 연결 리스트
  - 환형 리스트
  - 이중 환형 연결 리스트 등

단순 연결 리스트

- 단순 연결 리스트 (single linked list)
  - 특징
    - 노드들을 일렬로 연결한 것
    - 각 노드는 자료값과 다음 노드에 대한 포인터로 이루어짐
    - 이전 노드를 바로 접근할 수 없다.



  - C 언어에서의 단순 연결 리스트의 노드의 선언

```
struct _node {
    int key;
    struct _node *next;
} x;
struct _node *head;
```

```
typedef struct _node *nodeptr;
typedef struct _node {
    int key;
    nodeptr next;
} node;
node *head; /* nodeptr head;*/
```

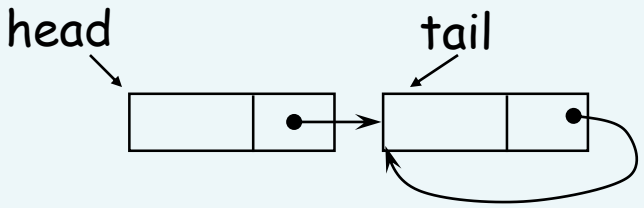
단순 연결 리스트

- 단순 리스트의 생성

```
node* init_list(void){
    node *head = (node*)malloc(sizeof(node));
    node *tail = (node*)malloc(sizeof(node));
    // malloc이 NULL을 리턴할 경우 처리 방법은??
    head->next = tail;
    tail->next = tail;

    return head;
}
```

```
main() {
    node *h1 = init_list();
}
```

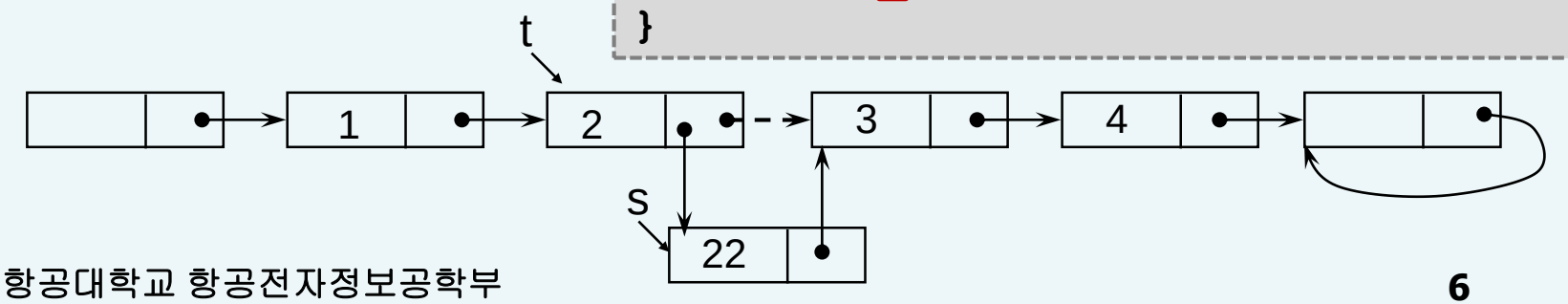


단순 연결 리스트

- 삽입 연산

```
node *insert_after(int k, node* t){
    node *s;
    if (!t || t->next == t) return t;
    s = (node*)malloc(sizeof(node));
    if (s==NULL) return s;
    s->key = k;
    s->next = t->next;
    t->next = s;
    return s;
}
```

```
main() {
    node *h1 = init_list();
    node *s1;
    s1 = insert_after(1,h1);
    s1 = insert_after(2,s1);
    s1 = insert_after(3,s1);
    s1 = insert_after(4,s1);
    insert_after(h1->next->next, 22)
}
```

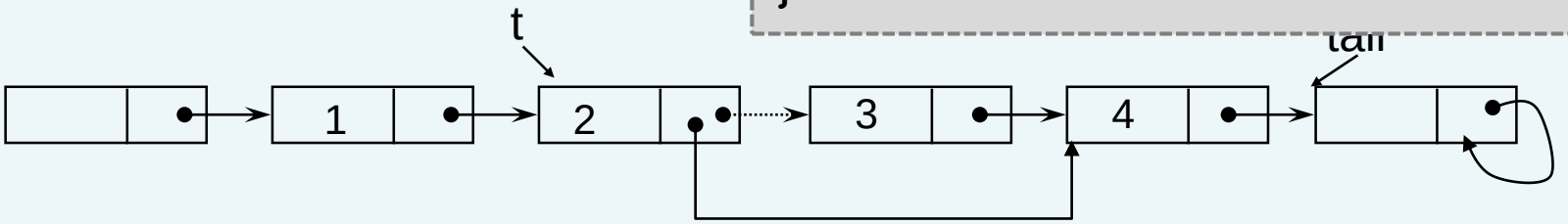


단순 연결 리스트

- 삭제 연산

```
int delete_next(node *t) {
    node *s;    /* if (!t) return 0; */
    if (t->next->next==t->next) /* t or t->next is tail */
          return 0;  /* can't delete tail */
    s = t->next;
    t->next = t->next->next;
    free(s);
    return 1;
}
```

```
main() {
   …
   delete_next(h1->next->next)
}
```



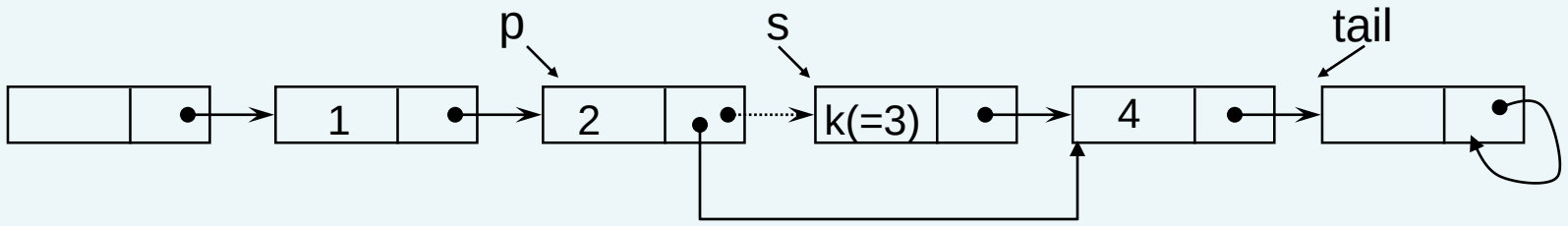
단순 연결 리스트

- 키값을 사용한 노드 탐색

```
node *find_node(node* head, int k){
    node *s;  // if (!head) return head;
    s = head->next;
    while (s->key != k && s->next != s)
        s = s->next;
    return s;
}
```

```
main() {
   …
   s1 = find_node(h1, 3)
}
```

- 키 값을 이용한 노드의 삭제



단순 연결 리스트

```
int delete_node(node* head, int k) {
    node *s, *p;
    p = head;
    s = p->next;
    while (s->key != k && s->next != s){
        p = p->next;
        s = p->next;
    }
    if (s->next != s) { /* if k is found */
        p->next = s->next;
        free(s);
        return 1;
    }
    else return 0;
}
```

```
main() {
   …
   delete_node(h1, 3)
}
```

```
int delete_node(node* h, int k) {
    while (h->next->key != k && h->next != h)
        h = h->next
    return (delete_next(h));
}
```

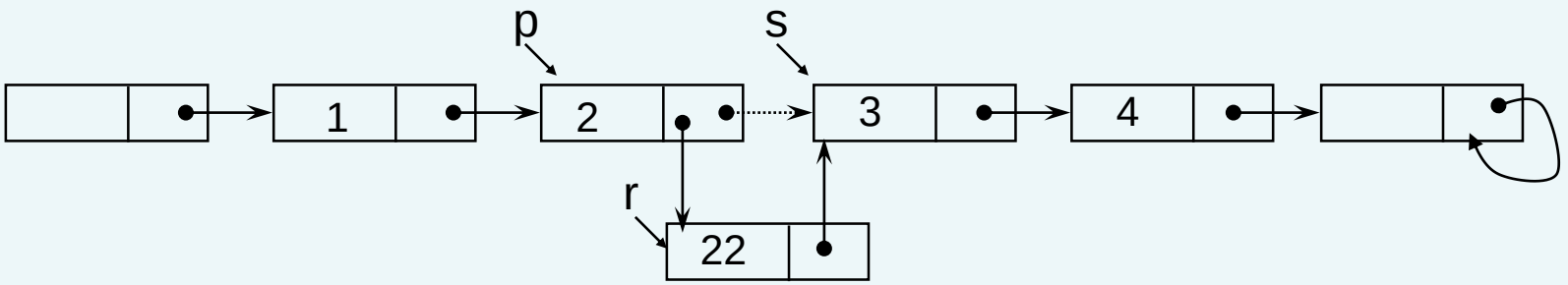
단순 연결 리스트

- 해당 키값을 가진 노드 앞에 삽입

```
node *insert_node(node *head, int t, int k) {
          /* insert t before k */
    node *s;
    node *p = head;
    node *r;
    s = p->next;
    while (s->key != k && s->next != s) {
        p = s; s = s->next;
    }
    if (s != tail) { /* if k is found */
        r = (node*)malloc(sizeof(node));
        r->key = t;
        p->next = r;
        r->next = s;
    }
    return p->next;
}
```

```
main() {
   …
   s1 = insert_node(h1, 22, 3)
}
```

단순 연결 리스트




```
node *insert_node(node *h, int t, int k) {
    while (h->next->key != k &&
           h->next->next != h->next) {
       h = h->next
    }
    if (h->next->next != h->next)
       insert_after(t, h);
    return h->next;
}
```

단순 연결 리스트

- 동적인 정렬 (순서대로 삽입하기)

```
node *ordered_insert(node *h, int k) {
     node *s;
     node *r;
     s = h->next;
     while (s->key <= k && s->next != s)
     {
          h = h->next;
          s = h->next;
     }
     r = (node*)malloc(sizeof(node));
     r->key = k;
     h->next = r;
     r->next = s;
     return r;
}
```

단순 연결 리스트

- 모든 리스트 원소의 출력

```
void print_list(node* h) {
    h = h->next;
    printf("\n");
    while (h != h->next){
        printf("%-8d", h->key);
        h = h->next;
    }
}
```

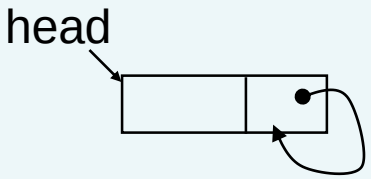
- 모든 노드의 삭제 : empty list로 만든다.

```
node *delete_all(node *head) {
    node *s, node *t;
    t = head->next;
    while (t->next != t) {
        s = t; t = t->next; free(s);
    }
    head->next = t;
    return head;
}
```

단순 연결 리스트

- 환형 연결 리스트 (Circular Linked List)
  - 특징 : tail이 없음 (제일 마지막 노드는 head를 가리킴)
  - 환형 연결 리스트의 생성

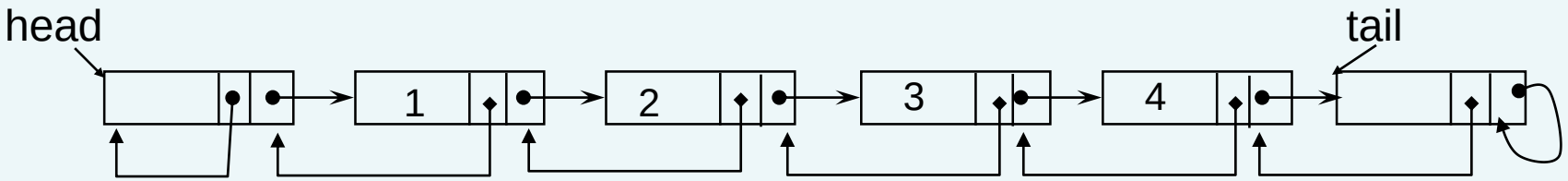
```
node *init_list(){
    node* head = (node*)malloc(sizeof(node));
    head->next = head;
    return head;
}
```



  - 환형 연결 리스트의 삽입, 삭제, . . .

이중 연결 리스트

- 이중 연결 리스트 (doubly linked list)
  - 리스트의 각 노드에는 이전 노드와 다음 노드로의 링크가 존재
  - 이전 노드를 쉽게 접근할 수 있음



  - 노드형의 선언

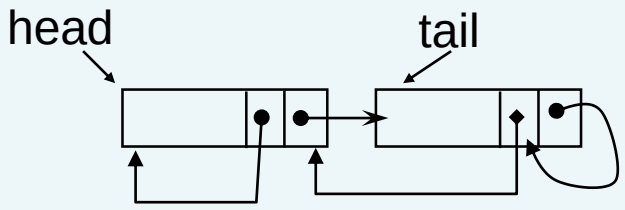
```
typedef struct _dnode
{
    int key;
    struct _dnode *prev;
    struct _dnode *next;
} dnode;
```

이중 연결 리스트

- 이중 연결 리스트의 생성

```
dnode *head, *tail;

void init_dlist(void)
{
    head = (dnode*)malloc(sizeof(dnode));
    tail = (dnode*)malloc(sizeof(dnode));
    head->next = tail;
    head->prev = head;
    tail->next = tail;
    tail->prev = head;
}
```



이중 연결 리스트

- 이중 연결 리스트의 삽입 : 해당 노드 앞에 삽입이 가능

```
dnode *insert_dnode_ptr(int k, dnode *t)
/* insert k, before t */
{
    dnode *i;
    if (t == head)    /* (t->prev == t) */
        return NULL;
    i = (dnode*)malloc(sizeof(dnode));
    i->key = k;
    t->prev->next = i;
    i->prev = t->prev;
    t->prev = i;
    i->next = t;
    return i;
}
```

이중 연결 리스트

- 이중 연결 리스트의 노드 삭제 : 바로 삭제 가능

```
int delete_dnode_ptr(dnode *p)
{
    if (p == head || p == tail)
        return 0;
    p->prev->next = p->next;
    p->next->prev = p->prev;
    free(p);
    return 1;
}
```

이중 연결 리스트

- 키값으로 탐색 후에 해당 노드 앞에 삽입

```
dnode *insert_dnode(int k, int t)
/* insert k, before t */
{
    dnode *s;
    dnode *i = NULL;
    s = find_dnode(head,t);
    /* if (s->next != s) i=insert_dnode_ptr(k,s);*/
    if (s.next != s) /* s is not tail:t is found */
    {
        i = (dnode*)malloc(sizeof(dnode));
        i->key = k;
        s->prev->next = i;
        i->prev = s->prev;
        s->prev = i;
        i->next = s;
    }
    return i;
}
```

이중 연결 리스트

- 키값으로 탐색 후에 해당 노드 삭제

```
int delete_dnode(int k)
{
     dnode *s;
     s = find_dnode(head, k);
     if (s->next != s)  /* if found */
          {
          s->prev->next = s->next;
          s->next->prev = s->prev;
          free(s);
          return 1;
          }
     return 0;
}
```

이중 연결 리스트

- 동적인 정렬 (순서대로 삽입하기)

```
dnode *ordered_insert(int k)
{
    dnode *s;
    dnode *i;
    s = head->next;
    while (s->key <= k && s != tail)
        s = s->next;
    i = (dnode*)malloc(sizeof(dnode));
    i->key = k;
    s->prev->next = i;
    i->prev = s->prev;
    s->prev = i;
    i->next = s;
    return i;
}
```

연결 리스트의 응용

- 단순 연결 리스트 응용 : 명함 관리 프로그램

  - 프로그램의 기능
    - 명함의 구성 : 이름, 회사, 전화 번호 (char의 배열)
    - 명함 입력, 명함 삭제, 명함 검색, 디스크(=화일)에서 명함 읽기, 디스크에 명함 저장, 명함들의 리스트를 출력, 프로그램 종료

  - 주요 내용
    - 리스트와 관련한 연산 익히기
    - 구조체의 디스크 입출력

연결 리스트의 응용

- 자료 구조의 생성

```
#define NAME_SIZE  21
#define CORP_SIZE  31
#define TEL_SIZE   16
#define REC_SIZE   (NAME_SIZE+CORP_SIZE+TEL_SIZE)

typedef struct _card {
    char name[NAME_SIZE];
    char corp[CORP_SIZE];
    char tel[TEL_SIZE];
    struct _card *next;
} card;

card *head, *tail;
void init_card(void) {
    head = (card*)malloc(sizeof(card));
    tail = (card*)malloc(sizeof(card));
    head->next = tail;
    tail->next = tail;
}
```

연결 리스트의 응용

- 입력

```
void input_card(void)
{
    card *t;
    t = (card*)malloc(sizeof(card));

    printf("\nInput namecard menu : ");
    printf("\n    Input name -> ");
    gets(t->name);  fgets(t->name,NAME_SIZE,stdin);
    printf("\n    Input corporation -> ");
    gets(t->corp);  fgets(t->corp,CORP_SIZE,stdin);
    printf("\n    Input telephone number -> ");
    gets(t->tel);  fgets(t->tel,TEL_SIZE,stdin);

    t->next = head->next;   /* insert at first */
    head->next = t;
}
```

연결 리스트의 응용

- 삭제

```
int delete_card(char *s)
{
    card *t;
    card *p;
    p = head;
    t = p->next;
    while (strcmp(s, t->name) != 0 && t != tail)
    {
        p = p->next;
        t = p->next;
    }
    if (t == tail)
        return 0;    /* not found */
    p->next = t->next;
    free(t); /* How much should we free ? */
    return 1;
}
```

연결 리스트의 응용

- 탐색

```
card *search_card(char *s)
    {
    card *t;
    t = head->next;
    while (strcmp(s, t->name) !=0 && t != tail)
        t = t->next;
    if (t == tail)
        return NULL;  /* not found */
    else
        return t;
    }
```

연결 리스트의 응용

- 연결 리스트를 디스크에 저장

```
void save_cards(char *s)
{
    FILE *fp;
    card *t;
    if ((fp = fopen(s, "wb")) == NULL)
    {
        printf("\n    Error : Disk write failure.");
        return;
    }
    t = head->next;
    while (t != tail)
    {
        fwrite(t, REC_SIZE, 1, fp);
        t = t->next;
    }
    fclose(fp);
}
```

연결 리스트의 응용

- 연결 리스트를 디스크에서 읽어오기

```
void load_cards(char *s){
    FILE *fp;
    card *t;
    card *u;
    if ((fp = fopen(s, "rb")) == NULL) {
        printf("\n    Error : %s is not exist.", s);
        return;
    }
    t = head->next;
    while (t != tail) {
        u = t;
        t = t->next;
        free(u);
    }
    head->next = tail;
    while (1) {
        t = (card*)malloc(sizeof(card));
        if (!fread(t, REC_SIZE, 1, fp)) {
            free(t);
            break;
        }
        t->next = head->next;
        head->next = t;
    }
    fclose(fp);
}
```

연결 리스트의 응용

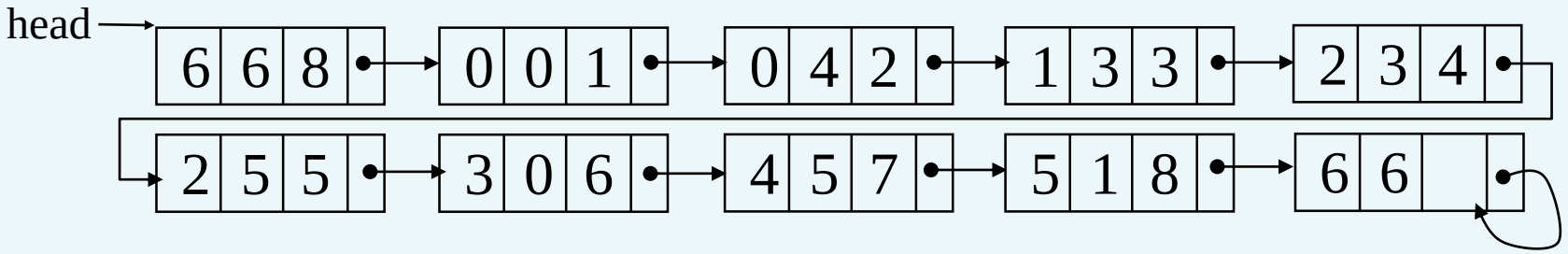
- main 함수

```
void main(void) {
    char *fname = "NAMECARD.DAT";
    char name[NAME_SIZE]; int i; card *t;
    init_card();
    while ((i = select_menu()) != 8) {
        switch (i)  {
            case 1 : input_card(); break;
            case 2 : printf("\n Input name to delete -> ");
                     fgets(name, NAME_SIZE, stdin);
                     if (!delete_card(name))
                         printf("\n Can't find that name.");
                     break;
            case 3 : printf("\n Input name to search -> ");
                     fgets(name, NAME_SIZE, stdin);
                     t = search_card(name);
                     if (t == NULL) {
                         printf("\n Can't find that name.");
                         break;
                     }
                     print_header(stdout);
                     print_card(t,stdout);
                     break;
            case 4 : load_cards(fname); break;
            ....
        }
    } printf("\n\nProgram ends...");
}
```

Sparse Matrix

- 단순 연결 리스트 응용 2 : Sparse Matrix
  - 단순 연결 리스트를 사용한 Sparse Matrix의 구현
    - 헤더 노드에는 행과 열의 갯수와 원소의 갯수가 저장됨
    - 각각의 노드에는 해당 원소의 위치 및 원소값 저장
    - 배열을 사용한 구현과는 달리 값을 새로 저장하는 것이 용이

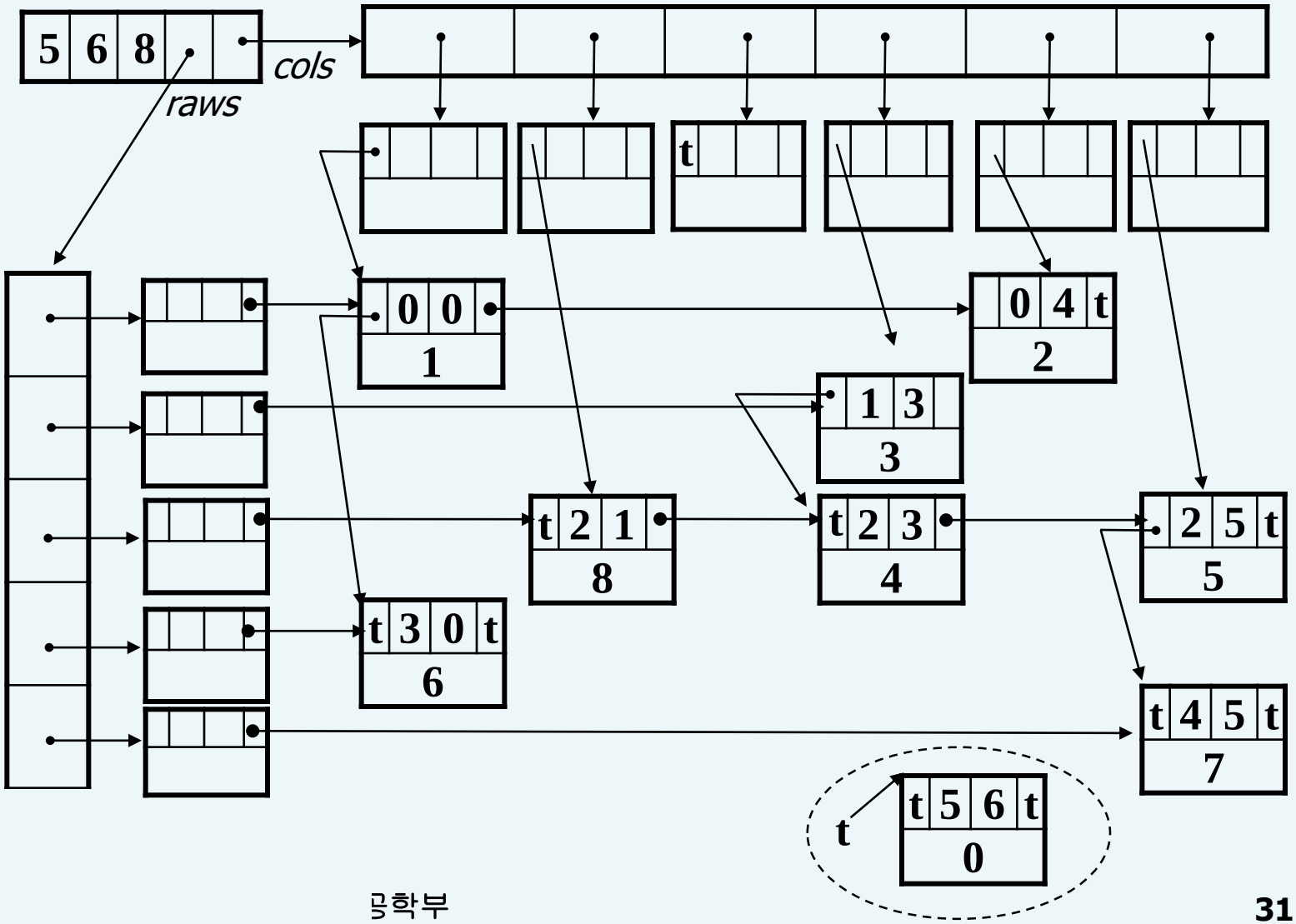
예) (0,0), (0,4), (1,3), (2,3), (2,5), (3,0), (4,5) (5,1)위치에 각각 1, 2, 3, 4, 5, 6, 7, 8 이 저장된 6x6 배열



→ (i, j)원소의 탐색 시간이 오래 걸림 (O(t))

  - 2차원 단순 연결 리스트를 사용한 Sparse Matrix의 구현
    - » 헤더 노드에는 행과 열, 0이 아닌 원소의 수와 열/행 배열을 저장
    - » 각각의 행과 열은 단순 연결 리스트의 형태를 갖는다.

2차원 연결구조를 사용한 Sparse Matrix



2차원 연결구조를 사용한 Sparse Matrix

- Node 타입 선언

```
#include <stdio.h>

typedef struct _node *nodeptr;
typedef struct _headnode {
    int raw;
    int col;
    int num;
    nodeptr *raws;
    nodeptr *cols;
} headnode;

typedef struct _node {
    int raw;
    int col;
    int val;
    nodeptr nextraw;
    nodeptr nextcol;
} node;
```

2차원 연결구조를 사용한 Sparse Matrix

- Node 생성

```
nodeptr makenode(int i, int j, int val,
                 nodeptr nextcol,
                 nodeptr nextraw) {
   nodeptr p = (nodeptr)malloc(sizeof(node));
   p->raw = i;
   p->col = j;
   p->val = val;
   p->nextcol = nextcol;
   p->nextraw = nextraw;

   return p;
}
```

2차원 연결구조를 사용한 Sparse Matrix

- 초기 행렬 생성

```
headnode *init_sparse_array(int n, int m){
    int i;
    headnode *head;
    nodeptr tail;
    tail = makenode(n,m,0,NULL,NULL);
    tail->nextcol = tail;tail->nextraw = tail;
    head = (headnode *)malloc(sizeof(headnode));
    head->raw = n; head->col = m; head->num = 0;
    head->raws =
            (nodeptr *)malloc(sizeof(nodeptr)*n);
    head->cols =
            (nodeptr *)malloc(sizeof(nodeptr)*m);
    for (i=0;i<n; i++) {
      head->raws[i] = malloc(sizeof(node));
      head->raws[i]->nextcol = tail; }
    for (i=0;i<m; i++) {
      head->cols[i] = malloc(sizeof(node));
      head->cols[i]->nextraw = tail; }
    return head;
}
```

2차원 연결구조를 사용한 Sparse Matrix

- Node 출력

```
void print_array(headnode *head) {
   int i,j;
   nodeptr* raws = head->raws;
   nodeptr t;
   int raw = head->raw;
   int col = head->col;
   printf("\n-----------------------\n");
   for (i=0; i<raw; i++) {
      t = raws[i]->nextcol;
      for (j=0; j<col; j++ )
         if (t->col == j) {
            printf(" %d", t->val);
            t = t->nextcol;
         }
         else printf(" _");
      printf("\n");
   }
   printf("-----------------------\n");
 }
```

2차원 연결구조를 사용한 Sparse Matrix

- 배열 원소의 값 변경

```
void replace_ele(headnode *head, int i, int j, int v){
   nodeptr t;
   nodeptr raw = head->raws[i];
   nodeptr col = head->cols[j];
   while (raw->nextcol->col < j) raw = raw->nextcol;
   if (raw->nextcol->col == j)
      if (v !=0 ) raw->nextcol->val = v;
      else {
        t = raw->nextcol;
        raw->nextcol = raw->nextcol->nextcol;
        while (col->nextraw != t) col = col->nextraw;
        col->nextraw = col->nextraw->nextraw;
        free(t); --(head->num);
      }
   else
      if (v != 0) {
        while (col->nextraw->raw <i)
          col = col->nextraw;
        t = makenode(i,j,v,raw->nextcol,col->nextraw);
        raw->nextcol = t;
        col->nextraw = t; ++(head->num);
      }
}
```

2차원 연결구조를 사용한 Sparse Matrix

- 전치 행렬의 생성

```
headnode *transpose(headnode *head) {
  int i,j;
  headnode *h;
  nodeptr *cp;
  nodeptr p1,p2,p3,tail;
  h = init_sparse_array(head->col,head->raw);
  h->num = head->num;
  tail = h->cols[0]->nextcol;
  cp = (nodeptr*)malloc(sizeof(nodeptr)*(h->col));
  for (i=0; i<h->col; i++) cp[i] = h->cols[i];
  for (i=0; i<h->raw; i++) {
     p1 = head->cols[i];
     p2 = h->raws[i];
     while (p1->nextraw->nextraw != p1->nextraw) {
        p3 = makenode(p1->nextraw->col,p1->nextraw->raw,
                      p1->nextraw->val, tail, tail);
        p2->nextcol = p3;
        cp[p3->col]->nextraw = p3;
        cp[p3->col]= p3;
        p2 = p3; p1 = p1->nextraw;
     }
  }
  return h;
}
```

2차원 연결구조를 사용한 Sparse Matrix

- Main 함수

```
main() {
        headnode *h = init_sparse_array(5,8);
        headnode *h2;
        replace_ele(h, 0,2,3);
        replace_ele(h, 1,1,0);
        replace_ele(h, 2,2,2);
        replace_ele(h, 3,6,4);
        replace_ele(h, 3,7,5);
        replace_ele(h, 4,3,9);
        replace_ele(h, 4,7,8);
        print_array(h);
        h2 = transpose(h);
        print_array(h2);
        exit(0);
}
```

정리

- 단순 연결 리스트
- 이중 연결 리스트
- 연결리스트의 응용
  - 주소록 프로그램의 구현
  - Sparse Matrix의 구현