알고리즘

2주차(17. 03. 13)

알고리즘 복잡도

○ **알고리즘** : 컴퓨터로 문제를 풀기 위한 단계적인 절차

* **알고리즘 효율성**

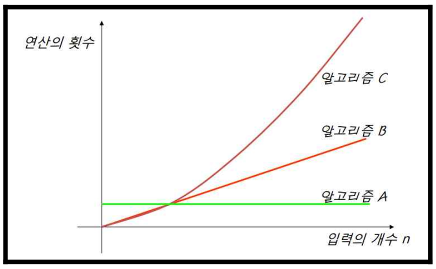
- 시간복잡도(수행시간) : 연산의 수행횟수는 고정된 숫자가 아니라 입력의 개수 n에 대한 함수
  // ex) 연산의 수 = 8 -> 3n+2
- 공간복잡도(메모리)

* **유한성** : 한정된 수의 단계 후에는 반드시 종료되어야 한다. // 알고리즘과 프로그램의 차이

* **복잡도 분석의 예 (n을 n번 더하는 문제)**

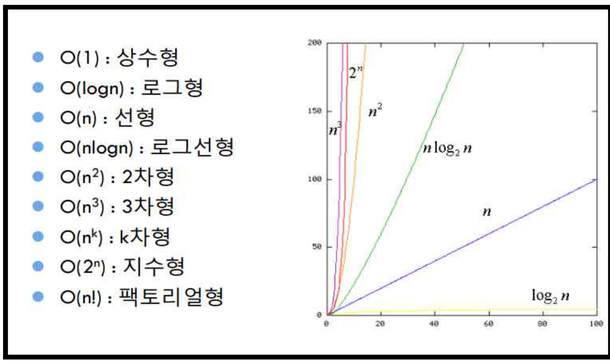
	알고리즘A	알고리즘B	알고리즘C	알고리즘D
코드	sum <- n*n;	sum <- 0; for i <- 1 to n do sum <- sum+n;	sum <- 0; for i <- 1 to n do for i <- 1 to n do sum <- sum+1;	sum <- 0; for i <- 1 to n for j <- 1 to n k <- A[1..n]에서 임의 n/2개 최대값 sum <- sum + k;
대입연산	1	1 + n - 1	1 + (n-1)*(n-1)	n*n*n/2 + 1
덧셈연산		n	n + n*(n-1)	n*n
곱셈연산	1			
나눗셈연산				
return	1	1	1	1
전체연산수	2 + 1(return)	2n + 1	$2n^2 - 2n + 2$ + 1(return)	$n^3/2 + n^2 + 1$ + 1(return)
빅오표기법	O(3) [상수시간대]	O(n)	$O(n^2)$	$O(n^3)$

* **상수시간대** : 입력 개수와 연산의 횟수가 고정되어 있는 것 // 알고리즘A



* **빅오 표기법** : 연산의 횟수를 대략적으로 표기한 것 / 데이터의 값에 따라 수행시간이 달라짐

- 두 개의 함수 f(n)과 g(n)이 주어졌을 때, 모든 $n \geq n0$에 대하여 $|f(n)| \leq c|g(n)|$을 만족하는 2개의 상수 c와 n0가 존재하면 f(n)=O(g(n))이다.
- 함수의 상한을 표시한다.(최고차항만 표시) // ex) $n \geq 5$ 이면 2n+1 < 10n 이므로 2n+1 = O(n)
- 빅오의 데이터 이상의 시간이 걸리지 않는다. // ex) $O(n^2)$



시간복잡도	n					
	1	2	4	8	16	32
1	1	1	1	1	1	1
logn	0	1	2	3	4	5
n	1	2	4	8	16	32
nlogn	0	2	8	24	64	160
n^2	1	4	16	64	256	1024
n^3	1	8	64	512	4096	32768
2^n	2	4	16	256	65536	4294967296
n!	1	2	24	40326	20922789888000	26313×10^{33}

* 빅오메가 표기법 // 점근적 시간 복잡도 : 많은 양의 데이터를 계산
  - 모든 $n \geq n0$에 대하여 $|f(n)| \geq c|g(n)|$을 만족하는 2개의 상수 c와 n0가 존재하면 $f(n)=\Omega(g(n))$이다
  - 빅오메가는 함수의 하한을 표시한다.
  - ex) $n \geq 1$ 이면 $2n+1 > n$ 이므로 $n = \Omega(n)$

* 점근 성능의 효율성
  - $1 < \log n < n < n^2 < n^2 \log n < n^3 < \ldots < 2^n$

* 연습문제 0 : 코드의 각 명령문의 수행 횟수를 계산하여 시간 복잡도 함수를 계산하시오

시간 복잡도 구하기

```
int test(int n){
     int i;
     int total;

     total=1;        // 1번
     for(i=1; i<n; I++){
          total *=n; // n-2번
     }
     return n;       // 1번
}
```

1+n-2+1 = n
시간 복잡도 : O(n)번

시간 복잡도 구하기2

```
float sum(float list[], int n){
     float tempsum;
     int i;
     tempsum=0; // 1번

     for(i=0;i<n;i++){
          tempsum+=list[i]; // n-1번
     }
     tempsum += 100;  // 1번
     tempsum += 200;  // 1번
     return tempsum;  // 1번
}
```

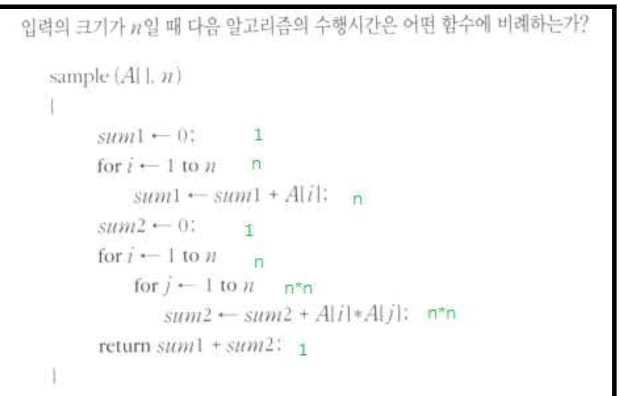
1 + n-1 + 1 + 1 + 1 = n+3
시간 복잡도 : O(n)번

시간 복잡도 구하기3

```
int test(int n){
     int i,b;

     b=1;       // 1번
     i=1;       // 1번
     while(i <= n)
          i = i*b;    // logn 번
}
```

1+1+logn = logn+2
O(logn)번

* 연습문제


입력의 크기가 n일 때 다음 알고리즘의 수행시간은 어떤 함수에 비례하는가?

```
sample(A[ ], n)
{
     sum1 ← 0;                       1
     for i ← 1 to n                  n
          sum1 ← sum1 + A[i];        n
     sum2 ← 0;                       1
     for i ← 1 to n                  n
          for j ← 1 to n             n*n
               sum2 ← sum2 + A[i]*A[j];   n*n
     return sum1 + sum2;             1
}
```

$2n^2 + 3n + 3$
$O(n^2)$

* 최선, 평균, 최악의 경우

- (예) 순차탐색
- **최선의 경우**: 찾고자 하는 숫자가 맨 앞에 있는 경우
  ∴ O(1)

- **최악의 경우**: 찾고자 하는 숫자가 맨 뒤에 있는 경우
  ∴ O(n)

- **평균적인 경우**: 각 요소들이 균일하게 탐색된다고 가정하면
  $(1+2+\ldots+n)/n=(n+1)/2$
  ∴ O(n)

순환 (순환(재귀) 알고리즘(함수)) : 자기 자신을 호출하는 함수(매개변수만 달라짐)

* 점화식 : 어떤 함수를 자신보다 더 작은 변수에 대한 함수와의 관계로 표현한 것
  // $a_n = a_{n-1} + w$ -> f(n) = n f(n-1) -> f(n) = f(n-1) + f(-2)

○ 점화식 정리

(1) $T(n) = T(n-1)+O(1)$, $T(1) = O(1)$ => $T(n) = O(n)$ //ex) 팩토리얼

(2) $T(n) = T(n-1)+O(n)$, $T(1) = O(1)$ => $T(n) = O(n^2)$

(3) $T(n) = T(n/2)+O(1)$, $T(1) = O(1)$ => $T(n) = O(\log n)$

(4) $T(n) = T(n/2)+O(n)$, $T(1) = O(1)$ => $T(n) = O(n)$ //ex) 가짜 동전 찾기

(5) $T(n) = T(n/2)+O(1)$, $T(1) = O(1)$ => $T(n) = O(n)$

(6) $T(n) = T(n/2)+O(n)$, $T(1) = O(1)$ => $T(n) = O(n\log n)$ //ex) 병합 정렬 알고리즘

* T(값) 은 다음 함수에 들어가는 데이터의 개수
* O(1) : for문을 사용하지 않을 경우(마지막 return 값이 1일 경우)
* O(n) : for문을 이용하여 데이터 전체를 한 번씩 처리해야할 경우
* $O(n^2)$: for문을 이용하여 데이터 전체를 두 번씩 처리해야할 경우

○ 점화식

* 데이터를 반으로 나누어 계산

(6) 병합정렬 수행시간

- 점화식으로 표현 : $T(n) = 2T(n/2) + O(n)$, $T(1) = O(1)$ / $T(n) = O(n\log n)$
- T(n) : 데이터의 개수가 n개 일 때, mergeSort 함수의 수행 시간
- $T(1) = 1$
- $T(n) = 2T(n/2) + n$

$= 2(2T(n/2^2) + n/2) + n = 2^2T(n/2^2) + 2 * n$

$= 2^2(2T(n/2^3) + n/2^2) + 2*n = 2^3T(n/2^3) + 3 * n$

...

$= 2^iT(n/2^i) + i*n$ // T(1) = 1을 알기 때문에 $n/2^i$을 1로 만들어야함.

// $n/2^i = 1$ => $n = 2^i$ => $\log_2 n = \log_2 2^i = k$ // $\log_2$은 2와 나누어진다.

$= n*T(n/n) + \log_2 n * n$

$= n*T(1) + \log_2 n * n$

$= n+n\log n$

(1) 하노이탑 : 기둥 세 개에 원반 옮기기 (작은 것이 큰 것 밑에 오면 안됨) // 2^n-1 번이 경우의 수

- $T(n) = 2 * T(n-1) + O(1)$

$= 2 * (2*T(n-2) +1) + 1$

$= 2 * (2*(2*T(n-3) +1) + 1) + 1$

$= 2^3T(n-3) +2^2 + 2 + 1$

...

$= \{2^i * T(n-i)\} + \{1 + 2 + ... + 2^{i-1}\}$

// $n-i = 1$ => $n = i+1$

$= \{2^i * T(i+1-i)\} + \{1 * (2^i-1) / 2-1\}$ //초항(r^n-1) / r-1

$= 2^i * T(1) + (2^i-1)$

$= 2^{n-1} + 2^{n-1} - 1$

$= 2*2^{n-1} - 1$

$= 2^n - 1$

$= O(2^n)$

(1) **팩토리얼** : 순환 알고리즘은 멈추는 부분과 순환 호출하는 부분으로 나누어진다.

- $T(n) = T(n-1) + O(1)$
  $= (T(n-2) + 1) + 1$
  $= (T(n-3) + 1) + 1 + 1$
  ...
  $= (T(n-k))+k$
  // $n-k = 1 \Rightarrow n=k+1$
  $= (T(1+k-k)) + k$
  $= T(1) + k$
  $= T(1) + n - 1$
  $= n$

(4) 가짜 동전 찾기 수행시간

- 점화식으로 표현 : $T(n) = T(n/2) + O(n)$, $T(1) = O(1)$ / $T(n) = O(n)$
- $T(n)$: 데이터의 개수가 n개 일 때, findFakeCoin(n) 함수의 수행 시간
- $T(n) = T(n/2) + n + 1 \Rightarrow T(n/2) + n$
  $= (T(n/4) + n/2) + n = T(n/4) + n/2 + n$
  $= (T(n/8) + n/4) + n/2 + n = T(n/2^3) + n/2^2 + n/2 + n$
  ...
  $= T(n/2^i) + n/2^{i-1} + n/2^{i-2} + \dots + n/2 + n$ // $n/2^{i-1}$에 2를 곱하면 $n/2^{i-2}$ 가 된다.
  $= T(1) + n * \{1/2^{i-1} + 1/2^{i-2} + \dots + 1/2 + 1\}$ // 등비수열
  // 공식 : 초항(r^n-1) / $r-1$
  $= T(1) + n * \{ 1/2^{i-1}(2^i-1) / 2-1 \}$
  $= T(1) + n * \{1/2^{i-1}(2^i-1)\}$ // $2^{i-1} = n/2$ 가 된다.
  $= 1 + n * \{2/n(n-1)\}$
  $= 1 + n * \{2-2/n\}$
  $= 1 + 2n - 2 \Rightarrow 2n - 1$
  $= O(n)$

(2) 퀵정렬

- $T(n) = T(n-1) + n$
  $= T(n-2) + (n-1) + n$
  $= T(n-3) + (n-2) + (n-1) + n$
  ... // $n-k = 1 \Rightarrow k = n-1$
  $= T(n-k) + (n-k+1) + \dots + (n-2) + (n-1) + n$
  $= T(1) + 2 + 3 + \dots + (n-2) + (n-1) + n$
  $= 1 + 2 + 3 + \dots + n$
  $= n(n+1) / 2 = 1/2\ n^2 + 1/2n$

(2) 이진검색

- $T(n) = T(n/2) + c2$
  $= T(n/4) + c2 + c2$
  $= T(n/2^i) + I * c2$
  ... // $n/2^i = 1 \Rightarrow n = 2^i \Rightarrow i = \log n$
  $= T(1) + I*c2$
  $= c1 + c2\log n$
  $= O(\log n)$

정렬

○ 선택 정렬(Selection Sort) : O(n²)

* Max 값을 -1 하면서 하나하나 비교하여 스왑한다.

선택 정렬(Selection Sort)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void Swap(int &a, int &b){
        int tmp = a;
        a = b;
        b = tmp;
}

void Selection_Sort(int a[], int max){
        int maxindex;
        for (int i = n - 1; i >= 0; i--){
                maxindex = i;
                for (int j = i - 1; j >= 0; j--){
                        if (a[maxindex] < a[j])
                                maxindex = j;
                }
                Swap(a[i], a[maxindex]);
        }
}
```

```
void main(){
        srand((unsigned)time(NULL));
        int a[10];
        int max=9;

        printf("[ ");
        for (int i = 0; i<10; i++){
                a[i] = rand() % 101;
                printf("%2.d ", a[i]);
        } printf(" ]\n");

        Selection_Sort(a[], max);

        printf("[ ");
        for (int i = 0; i <= max; i++){
                printf("%2.d ", a[i]);
        } printf(" ]\n");
}
```

○ 버블 정렬(bubble sort) : O(n²)

* 근접한 데이터 크기를 비교하여 스왑한다.

버블 정렬(Bubble Sort)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void Swap(int &a, int &b){
        int tmp = a;
        a = b;
        b = tmp;
}

void Bubble_Sort(int a[], int max){
        for (int i = max; i > 1; i--){
                for (int j = 0; j < i; j++){
                        if (a[j] >= a[j + 1]){
                                Swap(a[j], a[j + 1]);
                                printf("[ ");
                        }
                }
        }
}
```

```
void main(){
        srand((unsigned)time(NULL));
        int a[10];
        int max=9;

        printf("[ ");
        for (int i = 0; i<10; i++){
                a[i] = rand() % 101;
                printf("%2.d ", a[i]);
        } printf(" ]\n");

        Bubble_Sort(a[], max);

        printf("[ ");
        for (int i = 0; i <= max; i++){
                printf("%2.d ", a[i]);
        } printf(" ]\n");
}
```

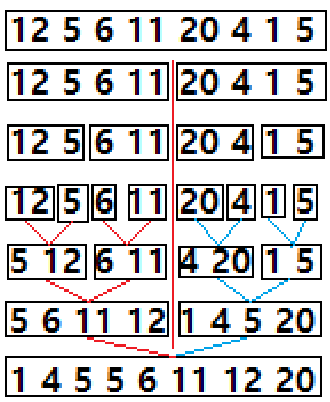
○ 병합 정렬(merge sort) : O(nlogn)

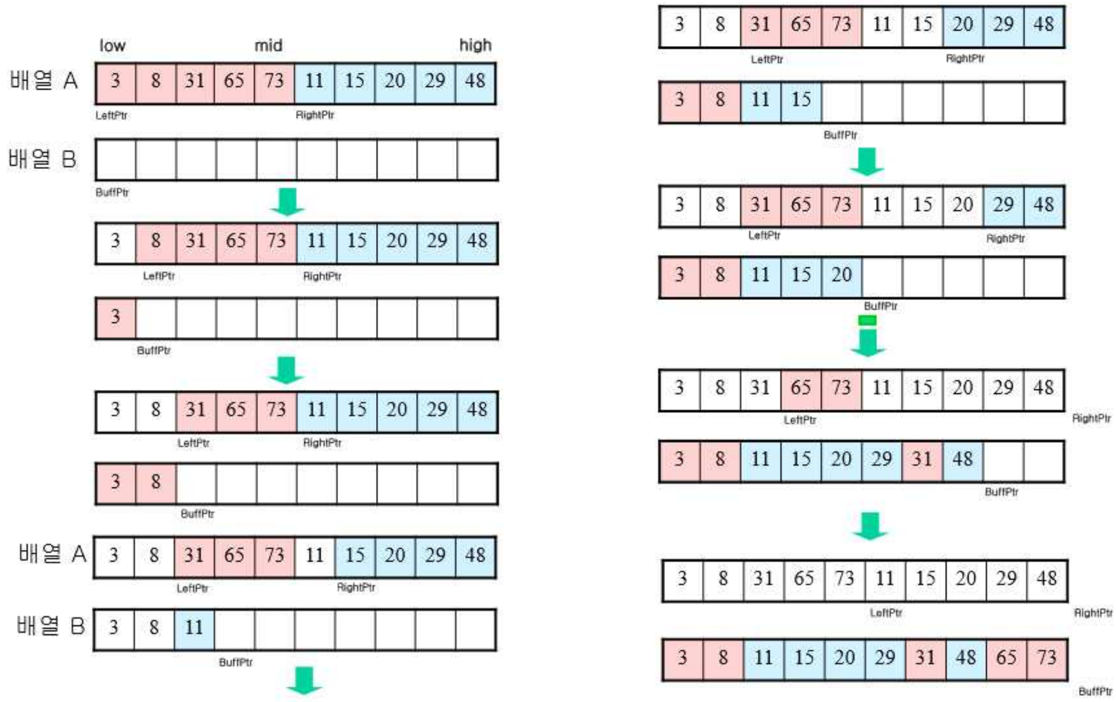
* 재귀함수 사용 / Row와 High 가 같아지면 데이터가 하나이므로 종료

병렬 정렬(Merge Sort)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 9

void Merge(int A[], int low, int mid, int high){
        int B[N+1];
        int LeftPtr, RightPtr, BufPtr;

        LeftPtr = low; RightPtr = mid + 1; BufPtr = low;
        while (LeftPtr <= mid && RightPtr <= high){
                if (A[LeftPtr] < A[RightPtr])
                        B[BufPtr++] = A[LeftPtr++];
                else B[BufPtr++] = A[RightPtr++];
        }
        if (LeftPtr > mid)
                for (int i = RightPtr; i <= high; i++)
                        B[BufPtr++] = A[i];
        else
                for (int i = LeftPtr; i <= mid; i++)
                        B[BufPtr++] = A[i];

// 합병이 끝나면 함수가 끝나기 때문에 넣어줘야한다.
        for (int i = low; i <= high; i++)
                A[i] = B[i];
}
```

```
void MergeSort(int A[], int low, int high){
        int mid;
        if (low < high){
                mid = (low + high) / 2;
                MergeSort(A, low, mid);
                MergeSort(A, mid+1, high);
                Merge(A, low, mid, high);
        }
}

void main(){
        srand((unsi
        int a[10];

        printf("[ ");
        for (int i =
                a[i
                pri
        } printf(" ]

        MergeSort(

        printf("[ ");
        for (int i =
                pri
        } printf(" ]
}
```





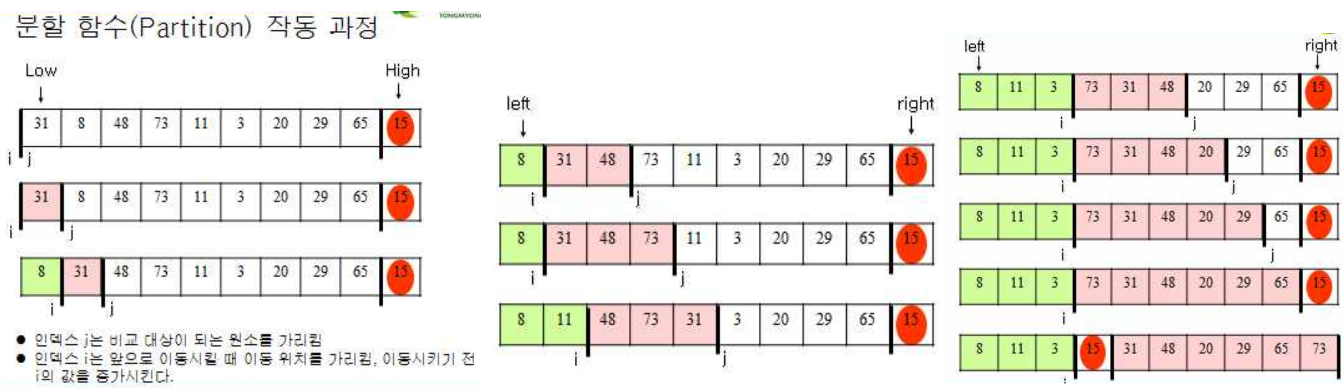
○ 퀵 정렬(Quick sort) : O()

* 재귀함수 사용 / Row와 High 가 같아지면 데이터가 하나이므로 종료

병렬 정렬(Merge Sort)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 9

void Swap(int &a, int &b){
        int tmp = a;
        a = b;
        b = tmp;
}

int Partition(int A[], int low, int high){
        int base = A[high];
        int i = low;

        for (int j = low; j < high; j++)
        if (A[i] <= base)
                Swap(A[i++], A[j]);

        Swap(A[i], A[high]);
        return i;
}
```

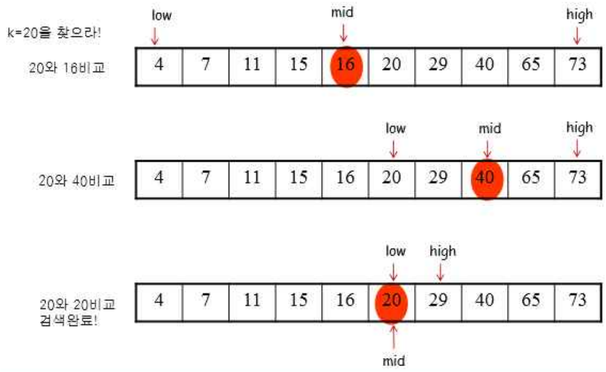
```
void QuickSort(int A[], int low, int high){
        int mid;
        if (low < high){
                mid = Partition(A, low, high);
                QuickSort(A, low, mid-1);
                QuickSort(A, mid + 1, high);
        }
}

void main(){
        srand((unsigned)time(NULL));
        int a[10];

        printf("[ ");
        for (int i = 0; i<10; i++){
                a[i] = rand() % 100;
                printf("%2.d ", a[i]);
        } printf(" ] / [START]\n\n");

        QuickSort(a, 0, N);

        printf("[ ");
        for (int i = 0; i <= N; i++){
                printf("%2.d ", a[i]);
        } printf(" ] / [END]\n");

}
```



검색 트리

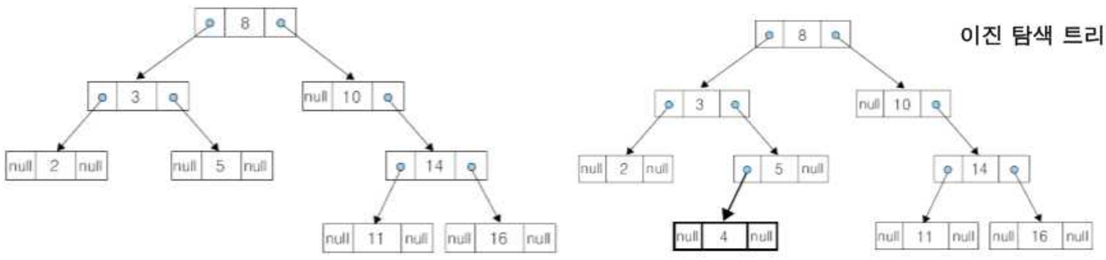
○ 이진 검색

* 정렬된 자료의 탐색에 적절
* 데이터는 배열에 저장되어 있어야 함
* 검색 시작 시 가운데 값부터 찾음



○ 이진 검색 트리 - 삽입

* 시간 복잡도 : 최악의 경우 O(n) / 평균 O(logn)



이진 탐색 트리에 원소 4를 삽입결과

(1) 이진 탐색 트리의 연결 표현_삽입 전

(2) 이진 탐색 트리의 연결 표현_삽입 후

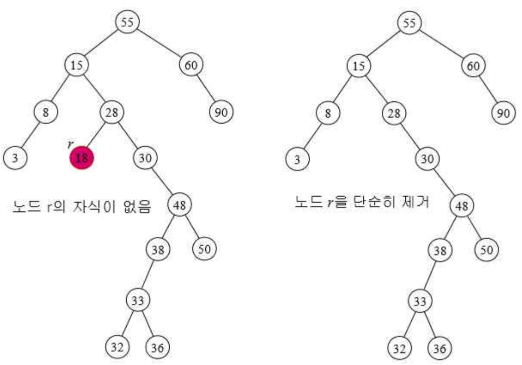
이진 검색 트리 - 삽입

```
treeNode* treeInsert(treeNode* t, int x)
{
     treeNode *tmp;
     if (t == NULL) {
          tmp = (treeNode*)malloc(sizeof(treeNode));
          //  tmp: 새 노드 할당
          tmp->data = x;
          tmp->left = tmp->right = NULL;
          return tmp;
     }
     if (x <  t->data) {
          t->left = treeInsert( t->left, x);
          return t;
     }
     else if (x >  t->data) {
          t->right = treeInsert( t->right, x);
          return t;
     }
     else {
          printf("이미 같은 키가 있습니다. \n");
          return t;
     }
}
```

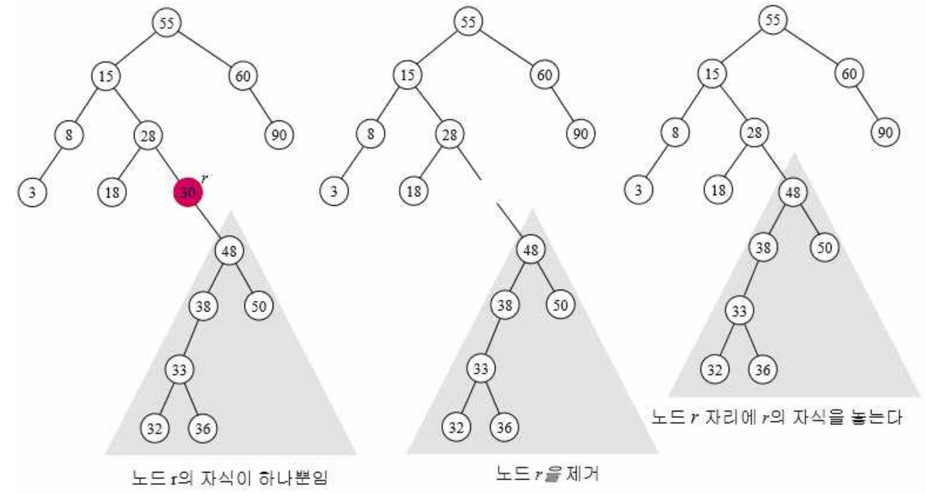
```
// 10개의 랜덤 값을 생성하여 이진 검색 트리에 삽입
for( int i = 0 ; i < 10 ; i++ ) {
     data = rand()%100 + 1;
     root = treeInsert( root, data );
}
```

○ 이진 검색 트리 - 삭제

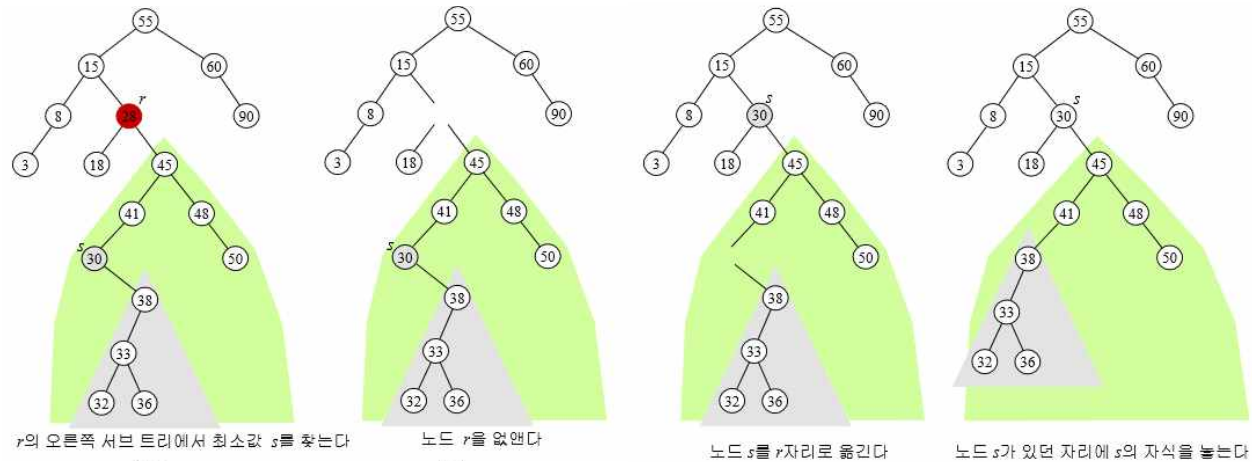
* 삭제할 노드가 단말 노드인 경우 (자식이 없는 경우) : 바로 삭제



* 삭제할 노드가 하나의 자식 노드를 가진 경우 : 자식 노드를 삭제할 노드 자리에 놓음



* 삭제할 노드가 두 개의 자식노드를 가진 2가지 경우
  - 왼쪽 자식에서 제일 큰 값을 찾아낸다.
  - 오른쪽 자식에서 제일 작은 값을 찾아낸다.
  - while을 돌려 자식 노드가 NULL이 될 때까지 찾아낸다.



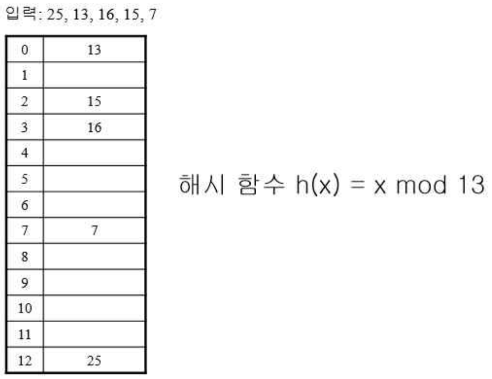
이진 검색 트리 - 삭제

```
// r : 트리의 루트 노드
// del : 삭제하고자하는 노드
// parent : r의 부모 노드

void treeDelete(treeNode**t, treeNode*del, treeNode*parent) {
    if (del == *t)   *t = deleteNode(*t);       // r이 루트 노드인 경우
    else if (del == parent->left)               // r이 루트가 아닌 경우
        parent->left = deleteNode(del);         // r이 p의 왼쪽 자식
    else parent->right = deleteNode(del);  // r이 p의 오른쪽 자식
}

treeNode* deleteNode( treeNode* del)  {
    if ( del->left ==NULL && del->right==NULL) return NULL;
    else if (del->left == NULL && del->right != NULL) return del->right;
    else if (del->left != NULL && del->right == NULL ) return del->left;
    else {
        treeNode *s=del->right;
        treeNode* p;
        while (s->left != NULL ){
            p = s;
            s = s->left;
        }
        del->data = s->data;
        if (s == del->right)  del->right  = s->right;
        else p->left = s->right;
        return del;
    }
}
```

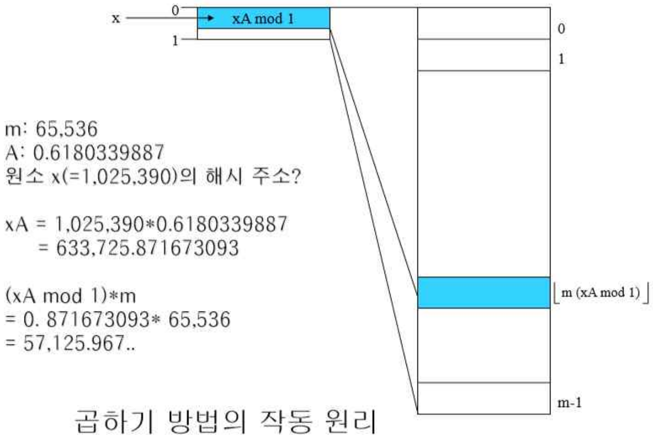
해시 테이블

○ 해시 테이블

* 시간 복잡도 : O(1)
* 나누기 방법 : h(x) = x mod m // 시험에 나옴
  - mod : 나머지 연산 / x : 키 / m : 테이블의 사이즈(소수여야 한다)
  - ex) h(15) = 15 % 13 = 2 => 2의 위치에 데이터가 있으면 맞는 값
  - ex) h(29) = 29 % 13 = 3 => 3의 위치에 데이터가 있으면 맞는 값



* 곱하기 방법 : h(x) = (x A mod 1) * m
  - A : 0 < A < 1 인 상수 // m은 굳이 소수일 필요 없음

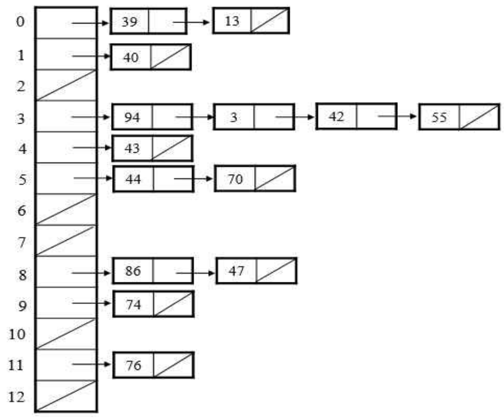


곱하기 방법의 작동 원리

1. 1을 나머지 연산을 해주면 소수점만 남음
2. 소수점만 남은 것에 m(65,536)을 곱하여줌
3. 나머지연산으로 소수점을 없애 줌

○ 연쇄법

* 같은 주소로 해싱되는 원소를 모두 하나의 연결리스트로 관리
* 추가적인 연결 리스트가 필요
* 장점 : 삭제가 용이 => 리스트 주소를 삭제
* 단점 : 포인터 저장을 위한 기억 공간이 필요, 기억 장소는 동적 할당



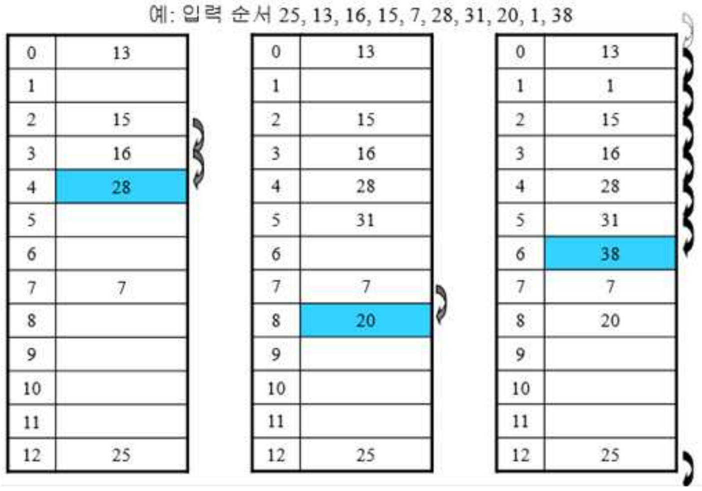
○ 개방 주소법

* 충돌이 일어나더라도 어떻게든 주어진 테이블 공간에서 해결하는 방법
* 추가적인 공간이 필요하지 않음
* 해시 값을 만드는 세가지 방법

  - 선형 조사　　- 이차원 조사　　- 이중 해싱

* 선형 조사

- 만약 충돌이 일어나면 다음 인덱스에 넣어준다
- 군집된 배열에 취약하다
- $h_i(x) = (h(x) + i) \bmod m$
- ex) $h_i(x) = (h(x) + i) \bmod 13$


예: 입력 순서 25, 13, 16, 15, 7, 28, 31, 20, 1, 38

0	13	13	13
1			1
2	15	15	15
3	16	16	16
4	28	28	28
5		31	31
6			38
7	7	7	7
8		20	20
9			
10			
11			
12	25	25	25

=> 28인 경우 13나머지 연산하면 2가 되기에 충돌이 일어난다. => 다음 인덱스를 다시 비교

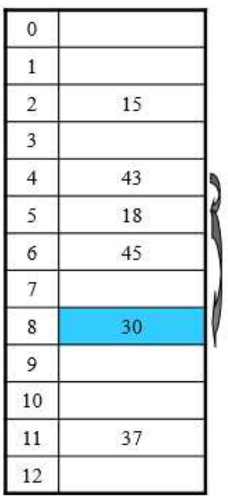
* 선형 조사 예제

* 크기가 13인 해시 테이블, 해시 함수 $h(x) = x \bmod 13$
* 충돌은 선형조사 사용 => $h_i(x) = (h(x) + i) \bmod m$
* 원소 10, 20, 30, 40, 33, 46, 50, 60

0	1	2	3	4	5	6	7	8	9	10	11	12
	40			30			20	33	46	10	50	60

* 이차원 조사

- 2차 군집된 배열에 취약하다
- $h_i(x) = (h(x) + c_1 i^2 + c_2 i) \bmod m$
- ex) $h_i(x) = (h(x) + i^2) \bmod 13$

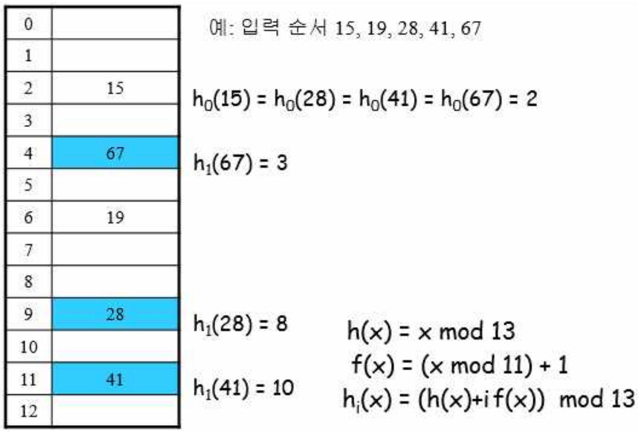


0	
1	
2	15
3	
4	43
5	18
6	45
7	
8	30
9	
10	
11	37
12	

예: 입력 순서 15, 18, 43, 37, 45, 30

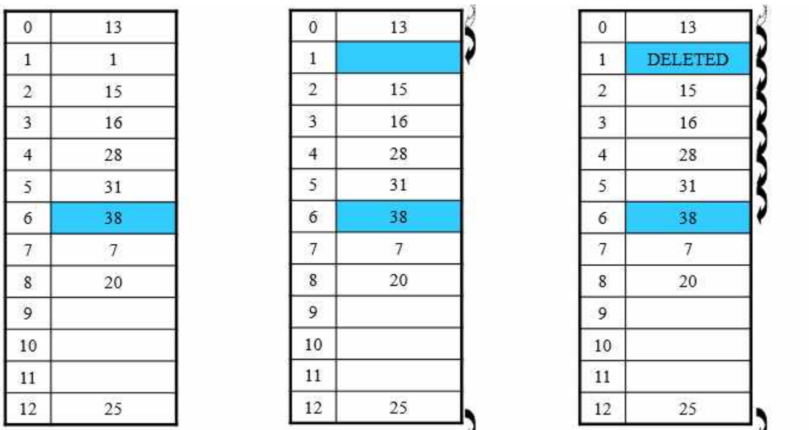
* 이중 해싱

- $h_i(x) = (h(x) + i * f(x)) \bmod m$



* 개방 주소법의 키 삭제

- 탐사 순서가 끊어지지 않게 ‘삭제 표시’를 해야함



(a) 원소 1 삭제 (b) 38 검색, 문제발생 (c) 표식을 해두면 문제없다

○ 체이닝

* 충돌이 일어나면 리스트로 연결함 => 충돌이 일어난 곳 앞에다가 연결
* 체이닝을 쓰면 리해싱(다른 번호에 넣는 것)을 할 필요가 없음

* 체이닝 예제

* 크기가 13인 해시 테이블, 해시 함수 h(x) = x mod 13
* 원소 10, 20, 30, 40, 33, 46, 50, 60
* 충돌이 일어나면 충돌이 일어난 곳 앞에 쪽에 주소를 삽입한다.

0	1	2	3	4	5	6	7	8	9	10	11	12
	40			30			46	60		33	50	
							20			10		

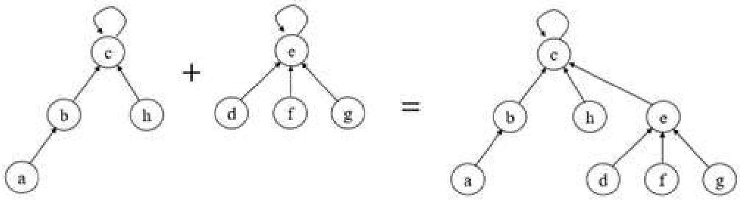
○ 해시테이블 정리

① 해시 테이블은 상수시간에 검색, 삽입, 삭제가 가능한 검색 알고리즘이다.
② 해시 테이블에서 저장 위치를 계산하는 것을 해시 함 수라고 한다.
③ 해시 함수를 만드는 대표적인 두 가지 방법은 나누기 방법과 곱하기 방법이다.
④ 좋은 해시 함수는 충돌이 적게 발생하는 함수이다.
⑤ 충동 해결방법은 크게 연쇄법과 개방주소법이 있다.
⑥ 적재율이 0.5이하이면 해시테이블의 검색 성능은 상수 시간을 유지한다.

상호배타적 집합

○ 트리를 이용한 집합

* 트리에서 루트의 부모는 자기 자신이다.
* 합집합을 할 때 두 번째 집합의 부모는 자기 자신에서 첫 번째 집합의 루트로 바뀐다.



트리를 이용한 집합 처리 알고리즘

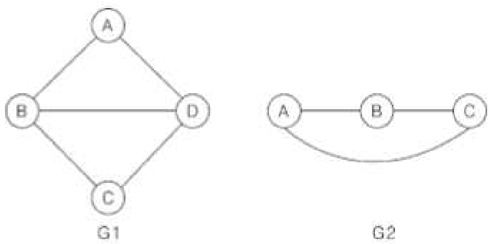
```
void MakeSet(int x) {
     p[x] = x ;
}

void Union(int x, int y) {
     p[FindSet(y)] = FindSet(x) ;
}

int FindSet(int x) {
     if (x == p[x])
          return x ;
     else
          return FindSet(p[x]) ;
}
```

그래프

○ 그래프 종류

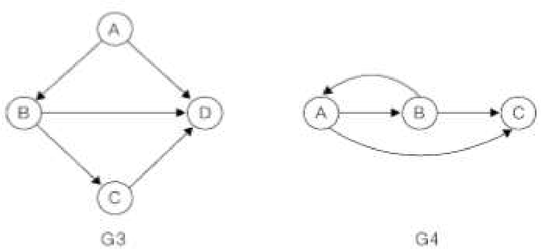
* 무방향 그래프 (undirected graph)

* 두 정점을 연결하는 간선의 방향이 없는 그래프
* 정점 Vi와 정점 Vj를 연결하는 간선을 (Vi, Vj)로 표현
  - (Vi, Vj)와 (Vj, Vi)는 같은 간선
  - V(G1) = {A, B, C, D}　　E(G1) = {(A, B), (A, D), (B, C), (B, D), (C, D)}
  - V(G2) = {A, B, C}　　E(G2) = {(A, B), (A, C), (B, C)}

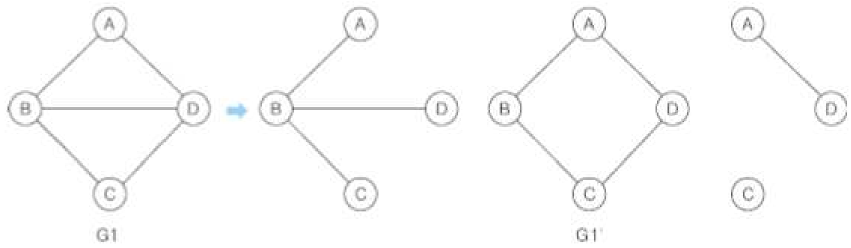


* 방향 그래프 (directed graph), 다이 그래프 (digraph)

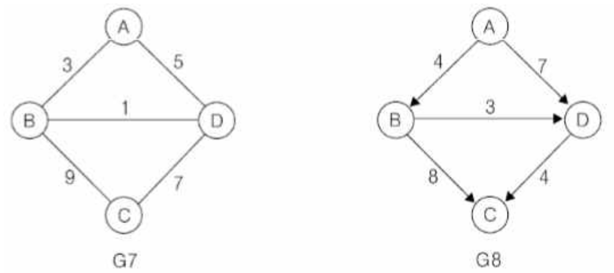
* 간선이 방향을 가지고 있느 그래프
* 정점 Vi에서 정점 Vj를 연결하는 간선 Vi->Vj를 <Vi, Vj>로 표현
  - Vi는 꼬리(tail), Vj는 머리(head)
  - <Vi, Vj>와 <Vj, Vi>는 다른 간선
  - V(G3) = {A, B, C, D}　　E(G3) = {<A, B>, <A, D>, <B, C>, <B, D>, <C, D>}
  - V(G4) = {A, B, C}　　E(G4) = {<A, B>, <A, C>, <B, A>, <B, C>}



* 부분 그래프 (subgraph)
  * 원래의 그래프에서 일부의 정점이나 간선을 제외하여 만든 그래프
  * 그래프 G와 부분 그래프 G'의 관계
    - V(G')⊆V(G), E(G')⊆E(G)



* 가중 그래프 (weight graph)
  * 정점을 연결하는 간선에 가중치(weight)를 할당한 그래프



○ 그래프 용어

* 차수(Degree) : 정점에 부속되어 있는 간선의 수
  * 그래프 G1에서 정점 A의 차수는 2, 정점 B의 차수는 3
  * 방향 그래프의 정점의 차수 = 진입차수 + 진출차수
    » 방향 그래프의 진입차수(in-degree) : 정점을 머리로 하는 간선의 수
    » 방향 그래프의 진출차수(out-degree) : 정점을 꼬리로 하는 간선의 수
    » 방향 그래프 G3에서 정점 B의 진입차수는 1, 진출차수는 2
  * 정점 B의 전체 차수는 (진입차수 + 진출차수) 이므로

* 경로(Path)
  * 경로 길이(path length) : 경로를 궁성하는 간선의 수

  * 단순 경로 : 모두 다른 정점으로 구성된 경로
    - 지나온 점으로 다시 돌아오지 않는 경로

  * 사이클 : 단순경로 중에서 경로의 시작 정점과 마지막 정점이 같은 경로
    - ex) 광민 - 배성 - 경렬 - 광민

* DAG (Directed Acyclic Graph) : 방향이 그래프이면서 사이클이 없는 그래프

* 연결 그래프 (Connected Graph)
  * 서로 다른 모든 쌍의 정점들 사이에 경로가 있는 그래프 (떨어져 있는 정점이 없는 그래프)
  * 그래프에서 두 정점 Vi에서 Vj까지의 경로가 있으면 정점 Vi와 Vj가 연결되어 있다고 함
  * 트리는 사이클이 없는 연결 그래프이다.

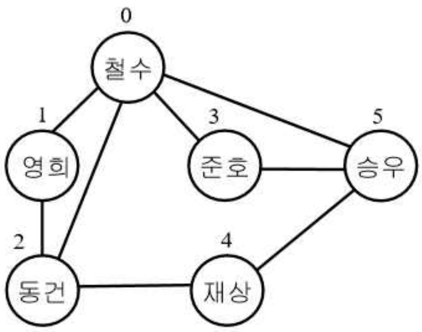
* 단절 그래프 (Disconnected Graph) : 연결되지 않은 정점이 있는 그래프

○ 그래프 - 인접 행렬(Adjacency Matrix) 표현

* 무향 그래프 (N*N 행렬로 표현)

* 1 : 정점과 정점 사이의 간선이 있음
* 0 : 정점과 정점 사이의 간선이 없음

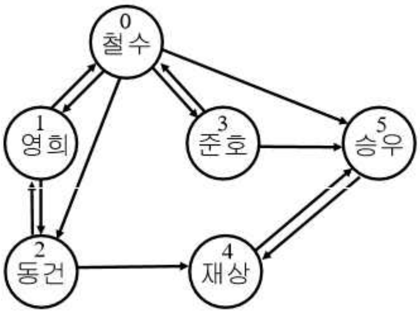
```
enum {철수=0,영희, 동건, 준호, 재상, 승우}
```



	0	1	2	3	4	5
0	0	1	1	1	0	1
1	1	0	1	0	0	0
2	1	1	0	0	1	0
3	1	0	0	0	0	1
4	0	0	1	0	0	1
5	1	0	0	1	1	0

* 유향 그래프

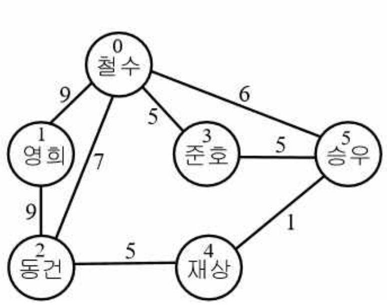
* 정점 I로부터 정점 j로 연결되는 간선이 있는지 나타냄



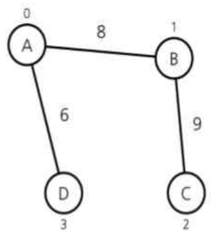
	0	1	2	3	4	5
0	0	1	1	1	0	1
1	1	0	1	0	0	0
2	0	1	0	0	1	0
3	1	0	0	0	0	1
4	0	0	0	0	0	1
5	0	0	0	0	1	0

* 가중치 그래프

* 1대신 가중치를 가짐
* 0대신 ∞로 표현하기도 함

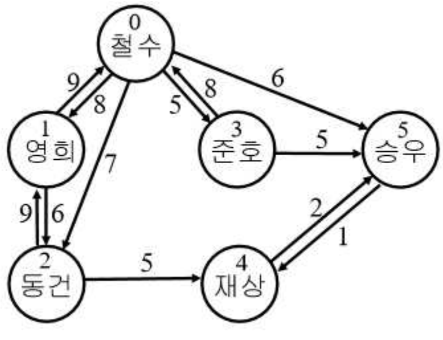


	0	1	2	3	4	5
0	0	9	7	5	0	6
1	9	0	9	0	0	0
2	7	9	0	0	5	0
3	5	0	0	0	0	5
4	0	0	5	0	0	1
5	6	0	0	5	1	0



	0 A	1 B	2 C	3 D
0 A	∞	8	∞	6
1 B	8	∞	9	∞
2 C	∞	9	∞	∞
3 D	6	∞	∞	∞

* 가중치+방향 그래프

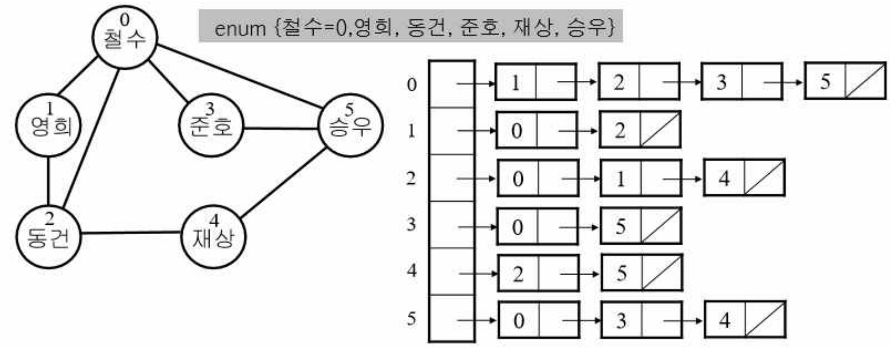


	0	1	2	3	4	5
0	0	8	7	5	0	6
1	9	0	6	0	0	0
2	0	9	0	0	5	0
3	8	0	0	0	0	5
4	0	0	0	0	0	2
5	0	0	0	0	1	0

○ 그래프 - 인접 리스트(Adjacency List) 표현

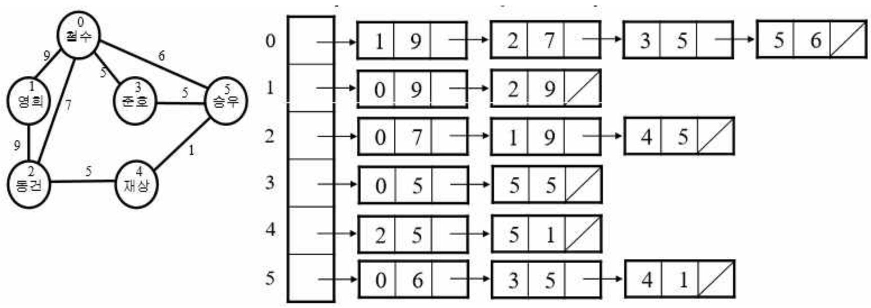
* 무향 그래프

* 리스트를 이용하여 연결



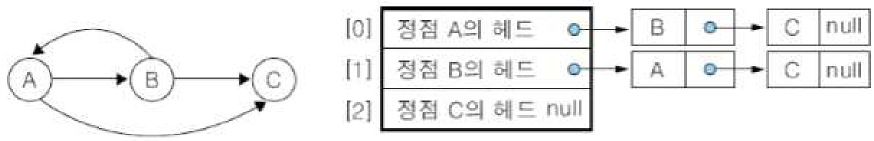
* 무향 - 가중치 그래프

* 리스트에 가중치를 넣는 공간을 추가한다.



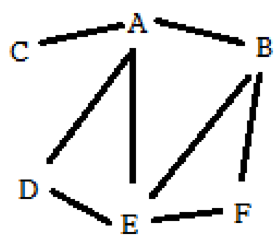
* 방향 그래프

* 각 정점에서 해당하는 방향의 리스트를 만듦



* 그래프 예제

* 해당 그래프를 인접 행렬과 인접 리스트로 표현



	A	B	C	D	E	F
A	0	1	1	1	1	0
B	1	0	0	0	1	1
C	1	0	0	0	0	0
D	1	0	0	0	1	0
E	1	1	0	1	0	1
F	0	1	0	0	1	0

	인접 리스트			
A	B	C	D	E
B	A	E	F	
C	A			
D	A	E		
E	A	B	D	F
F	B	E		