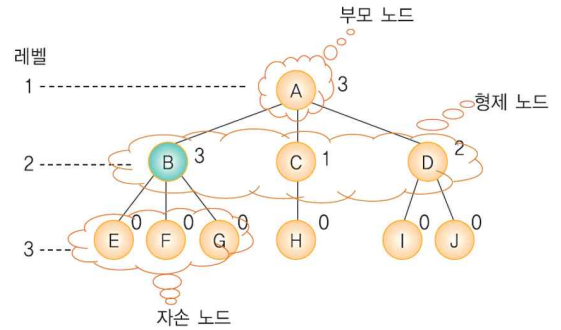
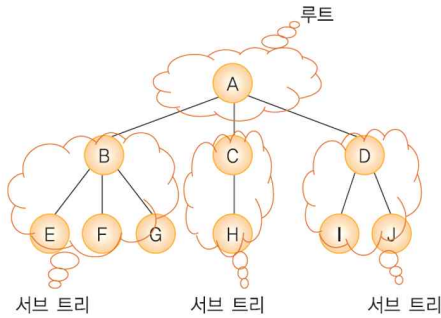


# 트리

리스트, 스택, 큐와는 다른 계층적인 구조를 나타낸다. 이러한 트리는 계층적인 조직을 표현하거나, 컴퓨터의 디렉터리 구조, 인공지능의 결정 트리에 자주 쓰인다.

## 트리의 용어

- **노드(node)**: 트리의 구성요소
- **루트(root)**: 부모가 없는 노드(A)
- **서브트리(subtree)**: 하나의 노드와 그 노드들의 자손들로 이루어진 트리



- 자식, 부모, 형제, 조상, 자손 노드: 인간과 동일
- **레벨(level)**: 트리의 각층의 번호
- **높이(height)**: 트리의 최대 레벨(3)
- **차수(degree)**: 노드가 가지고 있는 자식 노드의 개수

- **단말노드(terminal node)**: 자식이 없는 노드(A,B,C,D)
- **비단말노드**: 적어도 하나의 자식을 가지는 노드(E,F,G,H,I,J)

## 이진 트리

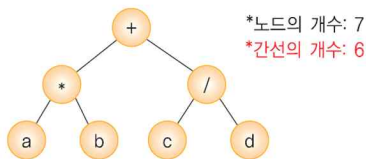
모든 노드가 2개의 서브 트리를 가지고 있는 트리이다.

## 이진트리와 일반 트리의 차이

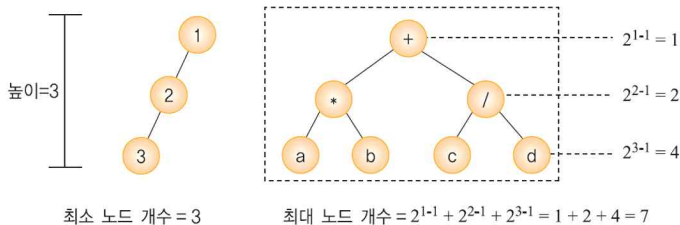
1. 모든 노드는 차수가 2여야 한다. 일반 트리는 이러한 제한이 없다
2. 일반 트리와 달리 이진 트리는 노드를 하나도 갖지 않을 수도 있다.
3. 서브 트리간에 순서가 존재한다. 왼쪽과 오른쪽을 구분한다.

## 이진 트리의 성질

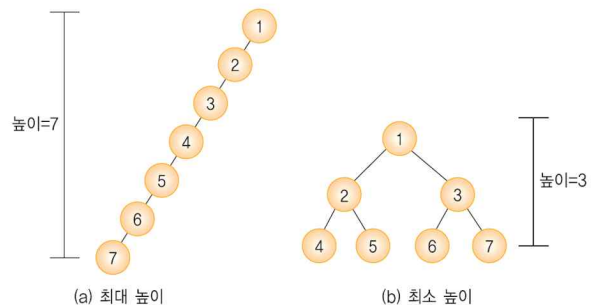
- 노드의 개수가 n개이면 간선의 개수는 n-1



- **높이가 h인 이진트리의 경우, 최소 h개의 노드를 가지며 최대  $2^h - 1$ 개의 노드를 가진다.**



- **n개의 노드를 가지는 이진트리의 높이**
- **최대 n**
- **최소  $\lceil \log_2(n+1) \rceil$**

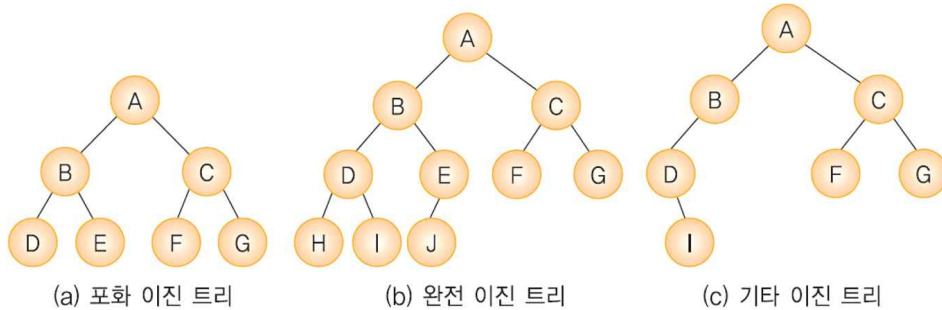


## 노드 개수와 높이간의 상관 관계 식

1. 최대 높이  $h =$  최소 노드  $x$ ,
2. 최소 높이  $2^{h-1} = x$  최대 노드

## 이진 트리의 분류

- 포화 이진 트리(full binary tree)
- 완전 이진 트리(complete binary tree)
- 기타 이진 트리



- 1) 포화 이진 트리 : 용어 그대로 트리의 각 레벨에 노드가 꽉 차있는 이진 트리
- 2) 완전 이진 트리 : 높이가  $k$ 일 때 레벨 1부터  $k-1$ 까지는 노드가 모두 채워져 있고 마지막 레벨  $k$ 에서는 왼쪽부터 오른쪽으로 노드가 순서대로 채워져 있는 이진트리이다.
- 3) 기타 이진 트리 : 위의 이진트리가 아닌 것

○ 전위 순회, 중위 순회, 후위 순회를 순환코드로 작성하시오

```
void inorder(TreeNode* root) {
    if (root == NULL)
        return;
    inorder(root->left);
    printf("[%d]", root->data);
    inorder(root->right);
}

void preorder(TreeNode* root) {
    if (root == NULL)
        return;
    printf("[%d]", root->data);
    preorder(root->left);
    preorder(root->right);
}

void postorder(TreeNode* root) {
    if (root == NULL)
        return;
    postorder(root->left);
    postorder(root->right);
    printf("[%d]", root->data);
}
```

### ○ 중위 순회를 반복으로 작성하시오

```
int top = -1;
TreeNode* stack[100];

void push(TreeNode* node) {
    if (top < SIZE - 1)
        stack[++top] = node;
}

TreeNode* pop() {
    if (top >= 0)
        return stack[top--];
}

void inorder(TreeNode* root) {
    while (1) { // 이걸 while(root != NULL) 이면 안되는게 NULL이 뜨는 것을 가정한 알고리즘이다.
        for (; root; root = root->left)
            push(root);
        root = pop();
        if (root==NULL) break; // 스택이 비어있으면 (pop이 NULL을 반환하면) 반복을 중지한다.
        printf("[%d] ", root->data);
        root = root->right; // 여기서 만약에 맨 왼쪽이라면, NULL이 담기고 이게 위에서 반복이 안된다.
    }
}
```

### ○ 레벨 순회를 구현하시오

```
typedef struct TreeNode {
    int data;
    struct TreeNode* left, * right;
}TreeNode;

typedef TreeNode* element;

typedef struct {
    element data[MAX_QUEUE_SIZE];
    int front;
    int rear;
} queue;

void init(queue* q) {
    q->front = 0;
    q->rear = 0;
}

int is_full(queue* q) {
    return q->front == (q->rear + 1) % MAX_QUEUE_SIZE;
}
```

```

int is_empty(queue* q) {
    return q->front == q->rear;
}

void enqueue(queue* q, element item) {
    if (is_full(q)) {
        fprintf(stderr, "error");
        exit(1);
    }
    q->rear = (q->rear + 1) % MAX_QUEUE_SIZE;
    q->data[q->rear] = item;
}

element dequeue(queue* q) {
    if (is_empty(q)) {
        fprintf(stderr, "error");
        exit(1);
    }
    q->front = (q->front + 1) % MAX_QUEUE_SIZE;
    return q->data[q->front];
}

void level_order(TreeNode* root) {
    if (root == NULL) return;
    queue q;
    init(&q);
    enqueue(&q, root);

    while (!is_empty(&q)) {
        root = dequeue(&q);
        printf(" [%d] ", root->data);
        if (root->left)
            enqueue(&q, root->left);
        if (root->right)
            enqueue(&q, root->right);
    }
}

```

#### ○ 노드 개수를 세는 코드 작성

```

int count_node(TreeNode* root) {
    if (root == NULL) return 0;
    int count = 1 + count_node(root->left) + count_node(root->right);
    return count;
}

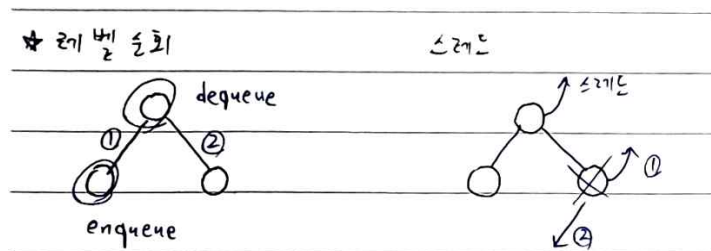
```

## ○ 이진트리 높이 계산

```
int hight(TreeNode* root) {  
    if (root == NULL) return 0;  
    int count = 1 + max(hight(root->left), hight(root->right));  
    return count;  
}
```

## ○ 단말 노드 계산

```
int get_left_count(TreeNode* root){  
    if (root == NULL) return 0;  
    if (root->left == NULL && root->right == NULL) return 1;  
    int count = get_left_count(root->left) + get_left_count(root->right);  
}
```



## ○ 스레드 이진 트리 (스레드의 존재 때문에 반복이 될 수 밖에 없다.)

```
TreeNode* find_thread(TreeNode* root) {  
    TreeNode* right = root->right;  
    if (right == NULL || right->thread) return right;  
    while (right->left != NULL) right = right->left;  
    return right;  
}  
  
void inorder(TreeNode* root) {  
    while (root->left) root = root->left;  
    do {  
        printf("%c", root->data);  
        root = find_thread(root);  
    } while (root);  
}
```

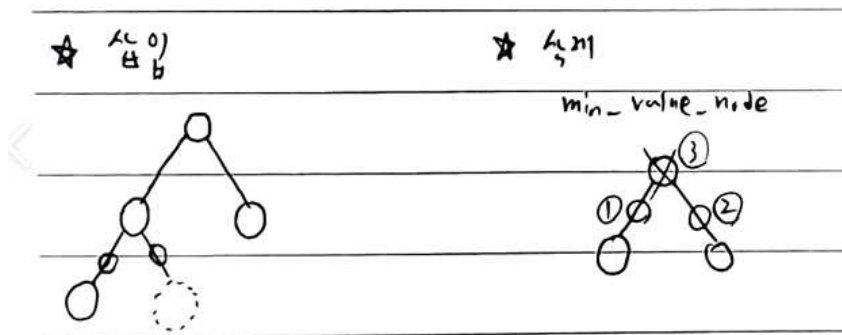
순환 호출은 함수를 호출해야 해서 비효율적일 수 있다. 스택을 사용하는 방법 이외에도 효율적인 순환 호출 방법이 없나?

그 방법이 이 스레드 이진 트리 방법이다. NULL을 잘 사용하자는 방침이다. NULL 링크에 중위 순회시에 후속 노드인 중위 후속자(inordersuccessor)를 저장시켜 놓은 트리가 스레드 이진 트리(threaded binary tree)

## ○ 이진 트리 탐색

```
TreeNode* search(TreeNode* root, int key) {  
    if (root == NULL) return NULL;  
    if (root->data == key) return root;  
    else if (key < root->data) return search(root->left, key);  
    else return search(root->right, key);  
}
```

```
TreeNode* search(TreeNode* root, int key) {  
    while (root != NULL) {  
        if (root->data == key) return root;  
        else if (key < root->data) root = root->left;  
        else root = root->right;  
    }  
    return NULL;  
}
```

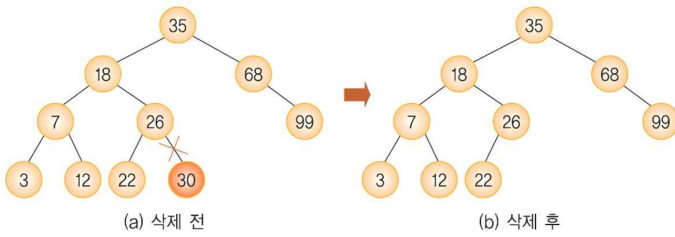


## ○ 이진 탐색 트리 삽입 연산

```
TreeNode* create_node(int item) {  
    TreeNode* node = (TreeNode*)malloc(sizeof(TreeNode));  
    node->data = item;  
    node->left = node->right = NULL;  
}  
  
TreeNode* insert(TreeNode* node, int key) {  
    if (node == NULL) return create_node(key);  
    if (key < node->data)  
        node->left = insert(node->left, key);  
    else if (key > node->data) // 탐색이 되면 삽입 안되면 실패이기 때문에, else가 되면 안된다!!!  
        node->right = insert(node->right, key);  
  
    return node;  
}
```

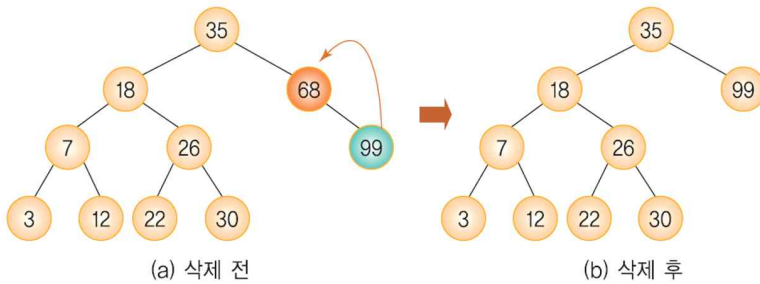
## ○ 이진 탐색 트리 삽입 삭제

### 1. 삭제하려는 노드가 단말 노드 일 경우



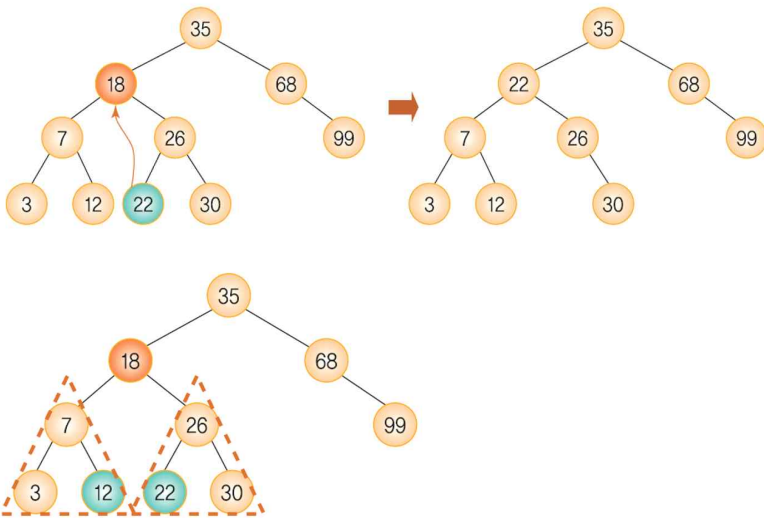
단말노드의 부모노드를 찾아서 연결을 끊으면 된다.

### 2. 삭제하려는 노드가 하나의 왼쪽이나 오른쪽 서브 트리 중 하나만 가지고 있는 경우



그 노드는 삭제하고 서브 트리는 부모 노드에 붙여준다.

### 3. 삭제하려는 노드가 두개의 서브 트리 모두 가지고 있는 경우



삭제노드와 가장 비슷한값을 가진 노드를 삭제노드 위치로 가져온다.

```

TreeNode* min_node(TreeNode* node) {
    while (node->left) node = node->left;
    return node;
}

TreeNode* delete(TreeNode* node, int key) {
    // 탐색 먼저
    if (node == NULL) return node;
    if (key < node->data) node->left = delete(node->left, key);
    else if (key > node->data) node->right = delete(node->right, key);
    else {
        if (node->left == NULL) {
            TreeNode* delete_node = node->right;
            free(node);
            return delete_node;
        }
        else if (node->right == NULL) {
            TreeNode* delete_node = node->left;
            free(node);
            return delete_node;
        }
        TreeNode* delete_node = min_value_node(node);
        node->data = delete_node->data;
        node->right = delete(node->right, node->data);
    }
    return node;
}

```

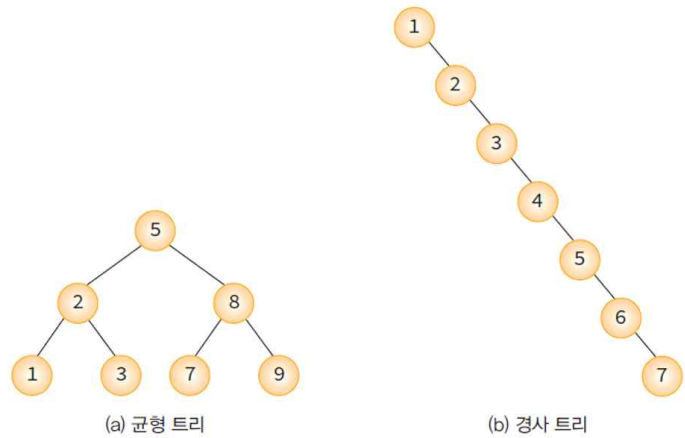


○ 균형 이진 탐색 트리

이진 탐색과 이진 탐색 트리의 근본적으로 같은 원리에 기초한다. 하지만 이진 탐색은 자료들이 배열에 저장되어 있기에 삽입과 삭제가 힘들다. 반대로 이진 탐색 트리는 빠른 시간안에 삽입과 삭제가 가능하다. 따라서 삽입과 삭제가 빈번하게 일어난다면 이진 탐색 트리를 사용하는 것이 좋다.

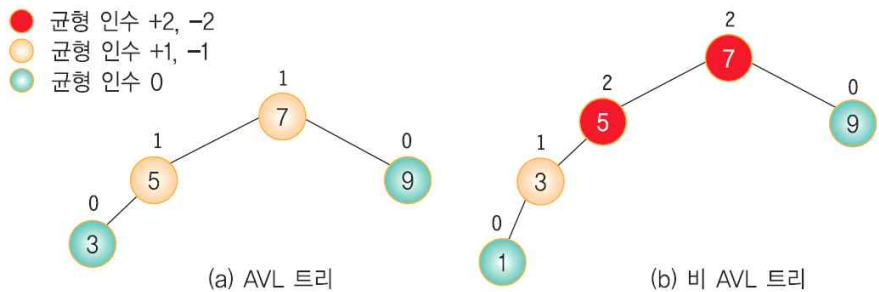
아쉽지만 앞서 구현한 이진 탐색 트리는 항상 균형이 있는건 아니다. 예를 들어 무작위가 아닌 정렬된 식 1,2,3,4,5,6이 삽입될 경우 불균형트리가 될 것이다.

- 이진탐색트리에서의 시간복잡도
  - 균형트리:  $O(\log(n))$
  - 불균형트리:  $O(n)$ , 순차탐색과 동일



○ AVL 트리

왼쪽 서브 트리의 높이와 오른쪽 서브 트리의 높이 차이가 1이하인 트리



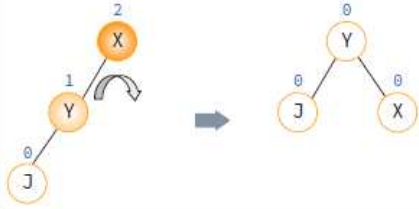
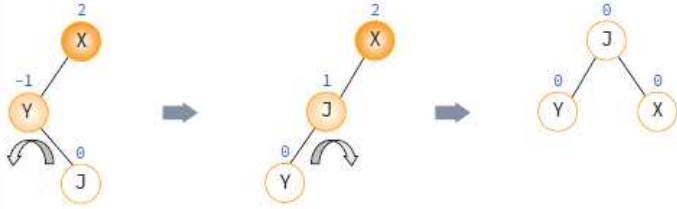
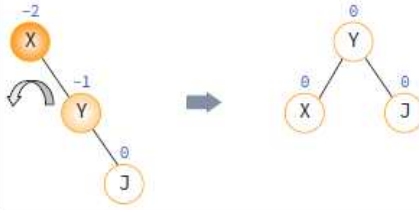
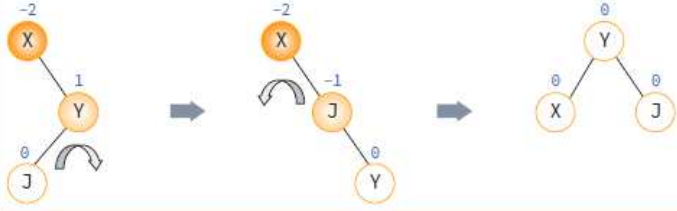
앞서 말한 이진 트리의 상위버전으로, 이 트리는 비균형 상태가 되면 스스로 노드들을 재배치해서 균형 상태를 유지한다. 항상 균형 상태를 유지하기에 평균, 최선 최악의 시간 복잡도가  $O(\log n)$ 으로 고정된다.

균형 인수는 왼쪽 서브 트리의 높이에서 오른쪽 서브 트리의 높이를 뺀 값이다.

균형 인수(balance factor) = (왼쪽 서브 트리의 높이 - 오른쪽 서브 트리의 높이)  
모든 노드의 균형 인수가  $\pm 1$  이하이면 AVL 트리

AVL트리는 일반적인 이진 탐색 트리와 연산 방법이 동일하나, 삽입 연산이 크게 다르다. 삽입 상태시에 노드의 균형이 깨질 우려가 크다. 깨진 트리를 균형있게 되돌리는 방법은, 새로운 노드부터 균형 인수가  $\pm 2$ 가 된 가장 가까운 조상 노드까지를 회전시키는 것이다.

균형이 깨지는 경우는 총 4가지가 있는데, 다음 표와 같다.

4가지의 경우	해결방법	설명
LL 타입		LL 회전: 오른쪽 회전
LR 타입		LR 회전: 왼쪽 회전 → 오른쪽 회전
RR 타입		RR 회전: 왼쪽 회전
RL 타입		RL 회전: 왼쪽 회전 → 오른쪽 회전

```

// 노드의 균형인수를 반환
int get_balance(AVLNode* node) {
    if (node == NULL) return 0;
    return get_height(node->left) - get_height(node->right);
}

// 오른쪽으로 회전시키는 함수
AVLNode* rotate_right(AVLNode* parent)
{
    AVLNode* child = parent->left;
    parent->left = child->right;
    child->right = parent;

    // 새로운 루트를 반환
    return child;
}

// 왼쪽으로 회전시키는 함수
AVLNode* rotate_left(AVLNode* parent)
{
    AVLNode* child = parent->right;

```

```

    parent->right = child->left;
    child->left = parent;

    // 새로운 루트 반환
    return child;
}

AVLNode* insert(AVLNode* node, int key)
{
    // 이진 탐색 트리의 노드 추가 수행
    if (node == NULL)
        return(create_node(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else    // 동일한 키는 허용되지 않음
        return node;

    // 노드들의 균형인수 재계산
    int balance = get_balance(node);

    // LL 타입 처리
    if (balance > 1 && key < node->left->key)
        return rotate_right(node);

    // RR 타입 처리
    if (balance < -1 && key > node->right->key)
        return rotate_left(node);

    // LR 타입 처리
    if (balance > 1 && key > node->left->key)
    {
        node->left = rotate_right(node->left);
        return rotate_right(node);
    }

    // RL 타입 처리
    if (balance < -1 && key < node->right->key)
    {
        node->right = rotate_right(node->right);
        return rotate_left(node);
    }

    return node;
}

```

## 우선 순위 큐

표현 방법	삽입	삭제
순서없는 배열	$O(1)$	$O(n)$
순서없는 연결 리스트	$O(1)$	$O(n)$
정렬된 배열	$O(n)$	$O(1)$
정렬된 연결 리스트	$O(n)$	$O(1)$
힙	$O(\log n)$	$O(\log n)$

우선 순위가 높은 데이터가 먼저 나가게 되는 큐

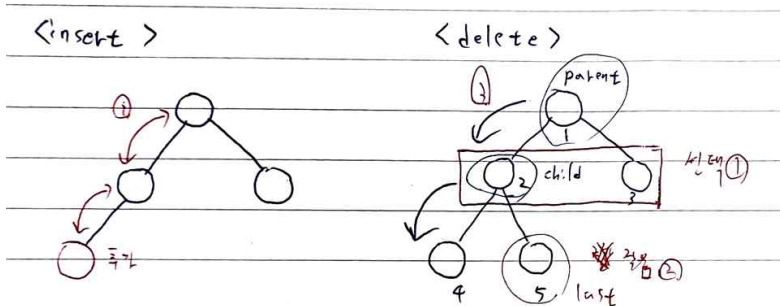
- 배열을 사용하는 방법 - 정렬이 안된 배열을 사용하면, 삽입은 그냥 넣으면 되기에 간단하다. 문제는 삭제할 때 모든 요소를 찾아야 한다.
- 정렬된 배열 사용하는 방법 - 삽입할 때 다른 요소값을 비교하면서 적절한 삽입 위치를 결정해야 한다. 반대로 삭제는 찾을 필요 없이 맨 뒤에 위치한 요소를 지우면 된다.
- 힙을 이용한 방법 - 완전 트리의 일종, 노드의 키들이  $\text{key}(\text{부모노드}) \geq \text{key}(\text{자식노드})$ 를 만족 하는 경우를 최대 힙,  $\text{key}(\text{부모노드}) \leq \text{key}(\text{자식노드})$ 를 만족하는 경우를 최소 힙이라고 한다. 보통 데이터들은 완벽하게 정렬될 필요가 없는데, 힙의 본래 목적은 삭제 연산이 수행될 때마다 루트 노드를 지우면 되는 것이므로 전체를 정렬할 필요는 없다.

### ○ 힙 구현 방법

주로 배열을 쓴다. 편의상 인덱스 0은 쓰지 않는다.

- 왼쪽 자식의 인덱스 = (부모의 인덱스)\*2
- 오른쪽 자식의 인덱스 = (부모의 인덱스)\*2 + 1
- 부모의 인덱스 = (자식의 인덱스)/2

### ○ 힙 삽입, 삭제 코드



```

#define MAX_HEAP_SIZE 100

typedef struct {
    int key;
}element;

typedef struct {
    element data[MAX_HEAP_SIZE];
    int size;
}Heap;

void insert(Heap* h, element item) {
    int i = ++(h->size);
    while ((i != 1) && (item.key > h->data[i / 2].key)) { // item->key는 안된다!! 이걸 포인터용이고 저
    건 실제 값이다.
        h->data[i / 2] = h->data[i]; // 부모와 자식을 바꾼다!
        i /= 2; // 인덱스 이동
    }
    h->data[i] = item;
}

element delete(Heap* h) {
    // 초기 상태 설정
    int parent, child;
    element first, last;
    parent = 1;
    child = 2;
    first = h->data[1]; // 맨 윗 대가리
    last = h->data[(h->size)--]; // 젤 말단

    while (child <= h->size) {
        if ((h->data[child].key) < (h->data[child + 1].key)) // 왼쪽과 오른쪽 둘 중 뭐가 큰지 모르
        니 결정하는 코드
            child++;
        if (last.key > h->data[child].key) break; // 말단이 이동하려는 녀석보다 크면 종료
        h->data[parent] = h->data[child]; // 교환
        parent = child; // 이동
        child *= 2;
    }
    h->data[parent] = last; // 교환
    return first;
}

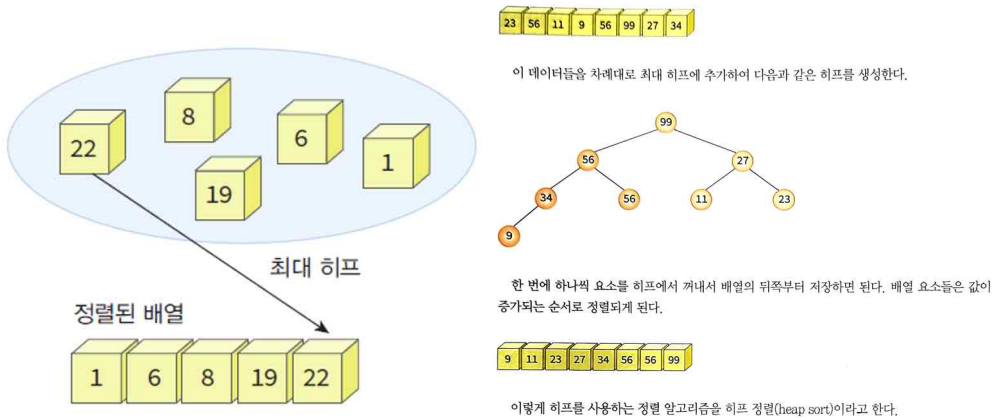
```

### ○ 힙의 복잡도 분석

- 1) 삽입 연산에서 최악의 경우, 루트 노드까지 올라가야 하므로 트리의 높이에 해당하는 비교 연산 및 이동 연산이 필요하다. ->  $O(\log n)$
- 2) 삭제도 최악의 경우, 가장 아래 레벨까지 내려가야 하므로 역시 트리의 높이 만큼의 시간이 걸린다. ->  $O(\log n)$

## ○ 힙 정렬

먼저 정렬해야 할  $n$ 개의 요소들을 최대 힙에 삽입한 뒤 한번에 하나씩 요소를 힙에서 삭제하여 저장하면 된다. 삭제되는 요소들은 값이 가장 큰 데이터이며, 하나의 요소를 힙에 삽입하거나 삭제할 때 시간이  $O(\log n)$  만큼 소요되고 요소의 개수가  $n$ 개이므로 전체적으로  $O(n \log n)$  시간이 걸린다. 프 정렬이 최대로 유용한 경우는 전체 자료를 정렬하는 것이 아니라 가장 큰 값 몇 개만 필요할 때이다. 이렇게 힙을 사용하는 정렬 알고리즘을 힙 정렬이라고 한다.



```
void heap_sort(element data[], int n) {
    Heap* h = create();
    init(h);
    for (int i = 0; i < n; i++) {
        insert(h, data[i]);
    }
    for (int i = (n - 1); i >= 0; i--) {
        data[i] = delete(h);
    }
    free(h);
}
```

## ■ 그래프

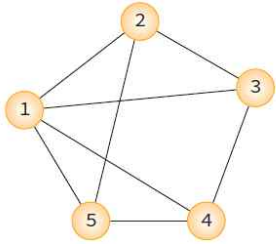
간단하게 객체 사이의 연결 관계를 표현할 수 있는 자료구조이다. 정점(vertex)과 간선(edge)로 이루어져 있다. 여기에 간선에 가중치를 부여한 경우, 네트워크라고도 한다.

### ○ 부분 그래프

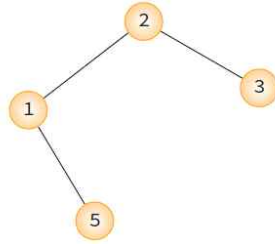
어떠한 완성된 그래프가 존재할 때, 그 그래프의 일부 경로만을 타나낸 것을 부분 그래프라고 한다.

$$V(S) \subseteq V(G)$$

$$E(S) \subseteq E(G)$$

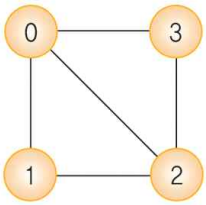


(a) 그래프



(b) 부분 그래프

### ○ 무방향 그래프



인접 정점 - 간선에 의해 직접 연결된 정점

정점의 차수 - 선택한 정점에 인접한 정점의 수 예를 들어 0번은 차수가 3임

### ○ 방향 그래프

진입 차수 : 외부에서 오는 간선의 수

진출 차수 : 외부로 향하는 간선의 수

방향 그래프의 모든 진입(진출) 차수의 합은 간선의 수

### ○ 경로에 따라서...

1. 단순 경로 : 경로 중에서 반복되는 간선이 없는 경로
2. 사이클 : 단순 경로의 시작 정점과 종료 정점이 동일한 경우

### ○ 연결 그래프

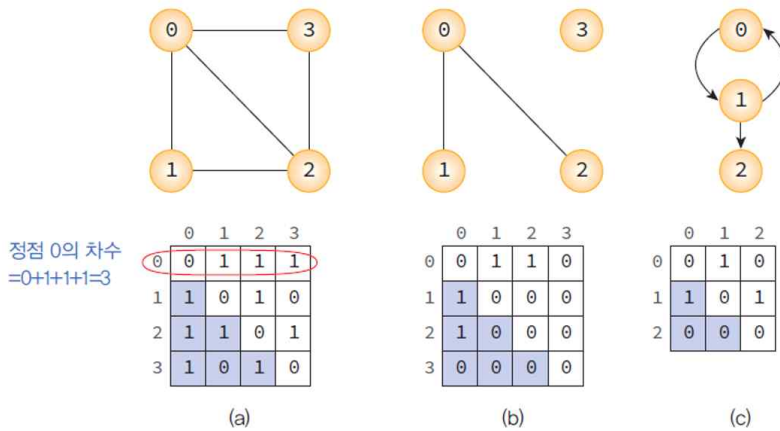
모든 정점쌍에 대해 항상 경로가 존재하는 그래프

### ○ 완전 그래프

모든 정점이 연결되어 있는 그래프

n개의 정점 가진 무방향 완전 그래프의 간선 수  $\text{edge} = n*(n-1)/2$

## ○ 인접 행렬로 구현한 그래프



## ○ dfs

깊이 우선 탐색 (DFS: depth-first search)

탐색 : 하나의 정점으로부터 시작하여 차례대로 모든 정점들을 한번씩 방문

1. 한 방향으로 갈 수 있을 때까지 가다가 더 이상 갈 수 없게 되면 가장 가까운 갈림길로 돌아와서 다른 방향으로 다시 탐색 진행
2. 되돌아가기 위해서는 스택 필요(순환함수 호출로 묵시적인 스택 이용 가능)
3. 인접 행렬  $\rightarrow O(n^2)$ , 인접 리스트  $\rightarrow O(n+e)$

# 행렬

```
void dfs(Graph* g, int v) {
    visited[v] = True;
    printf("정점 %d ->", v);
    for (int i = 0; i < g->n; i++) {
        if (g->data[v][i] && !visited[i])
            dfs(g, i);
    }
}
```

# 인접 리스트

```
void dfs(Graph* g, int v) {
    visited[v] = True;
    printf("정점 %d ->", v);
    for (GraphNode* i = g->data[v]; i; i = i->link) {
        if (!visited[i->vertex])
            dfs(g, i->vertex);
    }
}
```



## ○ bfs

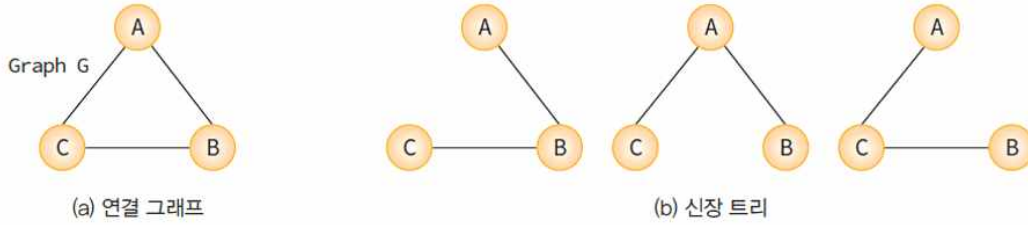
1. 너비 우선 탐색(BFS: breadth-first search)
2. 시작 정점으로부터 가까운 정점을 먼저 방문하고 멀리 떨어져 있는 정점을 나중에 방문하는 순회 방법
3. 큐를 사용하여 구현됨

```
void bfs(Graph* g, int v)
{
    Queue q;

    queue_init(&q);    // 큐 초기화
    visited[v] = TRUE;    // 정점 v 방문 표시
    printf("%d 방문 -> ", v);
    enqueue(&q, v);    // 시작 정점을 큐에 저장
    while (!is_empty(&q)) {
        v = dequeue(&q);    // 큐에 정점 추출
        for (int i = 0; i < g->n; i++)    // 인접 정점 탐색
            if (g->data[v][i] && !visited[i]) {
                visited[i] = TRUE;    // 방문 표시
                printf("%d 방문 -> ", i);
                enqueue(&q, i);    // 방문한 정점을 큐에 저장
            }
    }
}
```

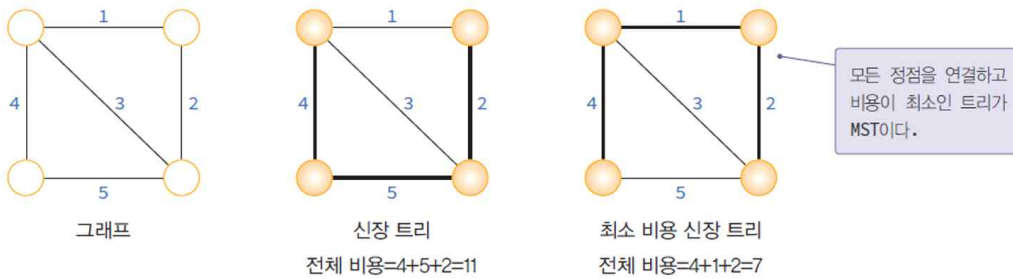
```
void bfs(Graph* g, int v) {
    Queue q;
    queue_init(&q); //큐 초기화
    visited[v] = TRUE; //정점 v 방문 표시
    printf("%d 방문 -> ", v);
    enqueue(&q, v); //시작정점을 큐에 저장
    while (!is_empty(&q)) {
        v = dequeue(&q); //큐에 저장된 정점 선택
        for (GraphNode* node = g->data[v]; node; node = node->link) { //인접 정점 탐색
            if (!visited[node->vertex]) { //미방문 정점 탐색
                visited[node->vertex] = TRUE; //방문 표시
                printf("%d 방문 -> ", node->vertex);
                enqueue(&q, node->vertex); //정점을 큐에 삽입
            }
        }
    }
}
```

## 신장 트리



1. 신장 트리는 그래프의 모든 정점을 포함하는 트리다. 사이클을 포함해서는 안 된다. 신장 트리는 그래프에 있는  $n$ 개의 정점을 정확히  $n-1$ 개의 간선으로 연결하게 한다.

### ○ 최소 비용 신장 트리(MST)



네트워크에 있는 모든 정점들을 가장 적은 수의 간선과 비용으로 연결한다.

응용할 수 있는 분야

1. 도로 건설 - 도시들을 모두 연결하면서 도로의 길이를 최소가 되도록 하는 문제
2. 전기 회로 - 단자들을 모두 연결하면서 전선의 길이를 가장 최소로 하는 문제
3. 통신 - 전화선의 길이가 최소가 되도록 전화 케이블 망을 구성하는 문제
4. 배관 - 파이프를 모두 연결하면서 파이프의 총 길이를 최소로 하는 문제

### ○ 그리드 알고리즘

각 단계에서 최선의 답을 선택하는 과정을 반복함으로써 최종적인 해답에 도달 탐욕적인 방법은 항상 최적의 해답을 주는지 검증 필요

### ○ 크루스컬의 최소 비용 신장 트리(MST) 알고리즘

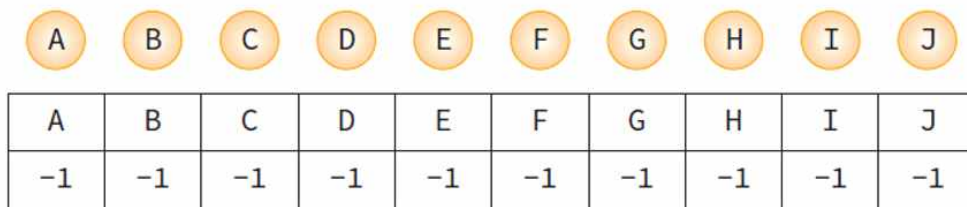
최소 비용 신장 트리가 최소 비용의 간선으로 구성됨과 동시에 사이클을 포함하지 않는다는 조건에 근거해서 각 단계에서 사이클을 이루지 않는 최소 비용 간선을 선택한다. 사이클을 포함하지 않아야 하므로 union-find 연산을 해야한다. 이는 원소가 어떤 집합에 속하는지 알아내는 알고리즘이다.

크루스컬의 최소 비용 알고리즘은 그리드 알고리즘이며, 주요하게 사용되는 연산은 union-find 연산이다.

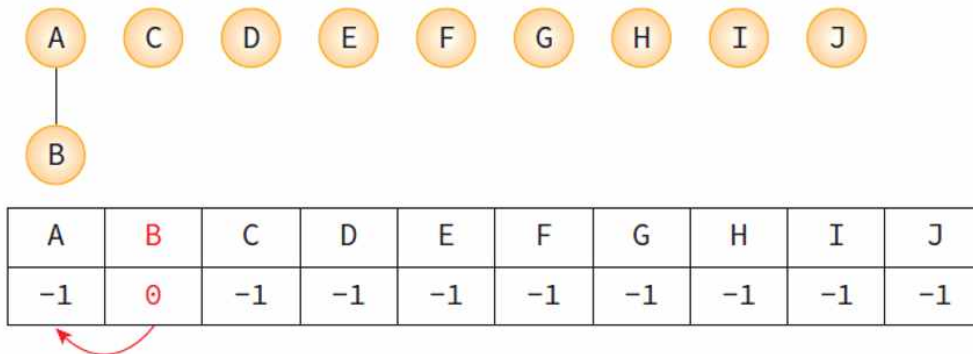
union(x,y) 연산은 그저 원소 x와 y가 속해있는 집합을 입력으로 받아 2개의 집합의 합집합으로 만드는 연산이다. find(x,y) 연산은 원소 x가 속해있는 집합을 반환한다.

이 구현을 가장 효율적으로 구현하는 방법은 트리 형태를 사용하는 것이다. 우리는 부모 노드만 알면 되므로 “부모 포인터 표현”을 사용한다. 이는 각 노드에 대해 그 노드의 부모에 대한 포인터만 저장하는 것이다.

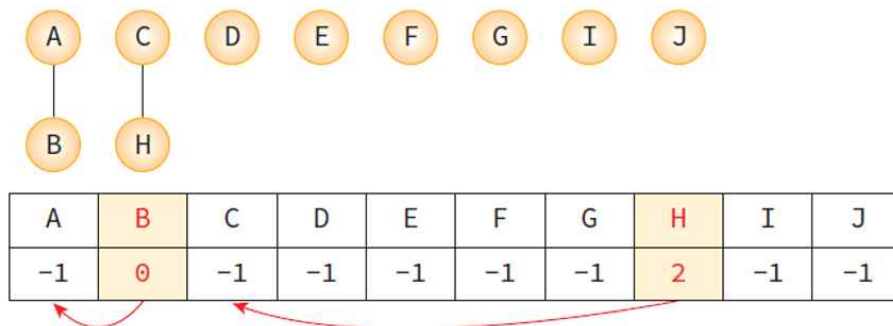
예를 들어



처럼 있다가 union(A, B)가 실행된다면,



가 된다. 그리고 union(C, H)가 호출된다면,



가 된다.

즉, 오로지 부모 노드만 가리키도록 배열을 형성한 것이다.

```
int parent[MAX_VERTICES];           // 부모 노드
                                   // 초기화

void set_init(int n)
{
    for (int i = 0; i < n; i++)
        parent[i] = -1;
}

// curr가 속하는 집합을 반환한다.
int set_find(int child)
{
    if (parent[child] == -1)
        return child;           // 루트
    while (parent[child] != -1) child = parent[child];
    return child;
}
```

```
// 두개의 원소가 속한 집합을 합친다.
void set_union(int a, int b)
{
    int root1 = set_find(a); // 노드 a의 루트를 찾는다.
    int root2 = set_find(b); // 노드 b의 루트를 찾는다.
    if (root1 != root2)      // 합한다.
        parent[root1] = root2;
}
```

이를 이용해서 크루스컬 알고리즘을 짤다면,

```
typedef struct Edge { // 간선을 나타내는 구조체
    int start, end, weight;
} Edge;

typedef struct{
    int n; // 간선의 개수
    int v; // 정점의 개수
    Edge edges[2 * MAX_VERTICES];
} Graph;

// 그래프 초기화
void graph_init(Graph* g)
{
    g->n = g->nvertex = 0;
    for (int i = 0; i < 2 * MAX_VERTICES; i++) {
        g->edges[i].start = 0;
        g->edges[i].end = 0;
        g->edges[i].weight = INF;
    }
}

// 간선 삽입 연산
void insert_edge(Graph* g, int start, int end, int w)
{
    g->edges[g->n].start = start;
    g->edges[g->n].end = end;
    g->edges[g->n].weight = w;
    g->n++;
}

// qsort()에 사용되는 함수
int compare(const void* a, const void* b)
{
    struct Edge* x = (struct Edge*)a;
    struct Edge* y = (struct Edge*)b;
    return (x->weight - y->weight);
}

// kruskal의 최소 비용 신장 트리 프로그램
void kruskal(Graph* g)
```

```

{
    int edge_accepted = 0;    // 현재까지 선택된 간선의 수
    int uset, vset;           // 정점 u와 정점 v의 집합 번호
    Edge e;
    set_init(g->v);           // 정점의 갯수만큼 부모 노드를 추적할 수 있게 집합 초기화

    qsort(g->edges, g->n, sizeof(Edge), compare); // 대충 대충 넣는 코드로 insert를 만들어버렸으니 이
    건 정렬해줘야 한다.

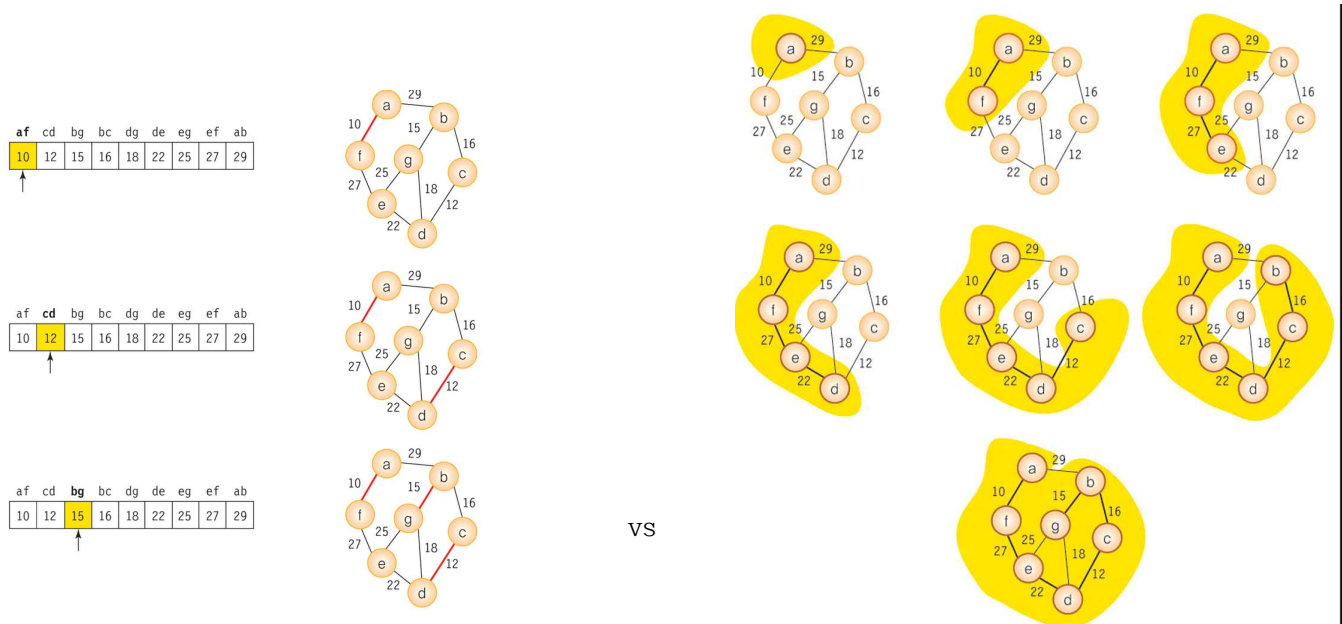
    int i = 0;
    while (edge_accepted < (g->v - 1))    // 선택된 간선의 수 < 정점보다 1개 낮아야 함
    {
        e = g->edges[i]; // 간선을 하나 선택하자
        uset = set_find(e.start);           // 정점 u의 집합 번호
        vset = set_find(e.end);             // 정점 v의 집합 번호
        if (uset != vset) {                 // 서로 속한 집합이 다르면
            printf("간선 (%d,%d) %d 선택\n", e.start, e.end, e.weight);
            edge_accepted++;
            set_union(uset, vset);           // 두개의 집합을 합친다.
        }
        i++;
    }
}

```

### ○ prim의 MST 알고리즘

시작 정점에서부터 출발하여 신장 트리 집합을 단계적으로 확장해나가는 방법이다. 신장 트리 집합에 인접한 정점 중에서 최저 간선으로 연결된 정점을 선택해서 신장 트리 집합에 추가한다.

크루스컬과 다른 점은 뭘 중심으로 봤느냐? 이다. 크루스컬 알고리즘의 경우, 주 관심사는 간선이었다. 반면 프림 알고리즘의 주 관심사는 노드이다.



크루스컬 vs 프림

프림은 distance라는 정점의 개수 크기의 배열이 필요하다.

```

typedef struct Graph {
    int n; // 정점의 개수
    int weight[MAX_VERTICES][MAX_VERTICES]; // 비용
} Graph;

int selected[MAX_VERTICES]; // 현재까지 알려진 거리 정보
int distance[MAX_VERTICES];

// 최소 dist[v] 값을 갖는 정점을 반환
int get_min_vertex(int n)
{
    int v; // 가장 작은 거리에 있는 노드 번호
    for (int i = 0; i < n; i++) {
        if (!selected[i]) { // 만약 선택이 안된 노드라면
            v = i; // 선택하고 반복문 종료
            break;
        }
    }

    for (int i = 0; i < n; i++) // 또 다른 나머지 노드를 하나씩 살펴볼건데
        if (!selected[i] && (distance[i] < distance[v])) v = i; // 선택이 안되었으며, 거리가 아까 선택된 것보
다 더 짧은게 존재한다면 선택

    return v; // 가장 짧은 거리를 반환
}

void prim(Graph* g, int s)
{
    int u;
    for (u = 0; u < g->n; u++) // 일단 모든 거리를 무한하다고 설정해야함
        distance[u] = INF;

    distance[s] = 0; // 스타트 지점은 당연히 거리가 0

    for (int i = 0; i < g->n; i++) {
        u = get_min_vertex(g->n); // 지금 알 수 있는 가장 작은 거리를 가진 노드 설정
        selected[u] = TRUE; // 노드 선택됨
        if (distance[u] == INF) return; // 근데 선택된 노드의 거리가 무한이면 다 살펴본것이므로 종료
        printf("정점 %d 추가\n", u);

        for (int v = 0; v < g->n; v++) // 다음 목표 지점 전부 살펴보자
            if (g->weight[u][v] != INF) // 비용이 무한이 아니면
                if (!selected[v] && g->weight[u][v] < distance[v]) // 선택되지 않았고 알려진 비용보다 훨씬
적다면
                    distance[v] = g->weight[u][v]; // 그 지점 가는데 비용을 지금의 비용으로 업데이트
    }
}

```

- **Kruskal** 알고리즘은 대부분 간선들을 정렬하는 시간에 좌우됨
  - 사이클 테스트 등의 작업은 정렬에 비해 매우 신속하게 수행됨
  - 네트워크의 간선  $e$ 개를 퀵정렬과 같은 효율적인 알고리즘으로 정렬한다면 **Kruskal** 알고리즘의 시간 복잡도는  $O(e \cdot \log(e))$ 가 된다
- **Prim**의 MST 알고리즘 복잡도
  - 주 반복문이 정점의 수  $n$ 만큼 반복하고, 내부 반복문이  $n$ 번 반복하므로 **Prim**의 알고리즘은  $O(n^2)$ 의 복잡도를 가진다.
  - 희박한 그래프 :  $O(e \cdot \log(e))$ 인 **Kruskal**의 알고리즘이 유리
  - 밀집한 그래프 :  $O(n^2)$ 인 **Prim**의 알고리즘이 유리

## 최단 경로

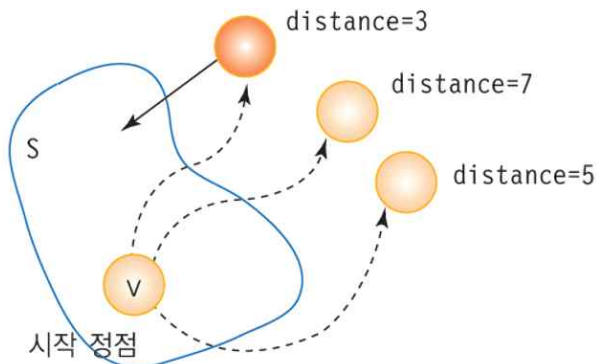
네트워크에서 정점  $u$ 와 정점  $v$ 를 연결하는 경로 중에서 간선들의 가중치 합이 최소가 되는 경로  
간선의 가중치는 비용, 거리, 시간 등

Dijkstra 알고리즘: 하나의 시작 정점에서 다른 정점까지의 최단경로 계산

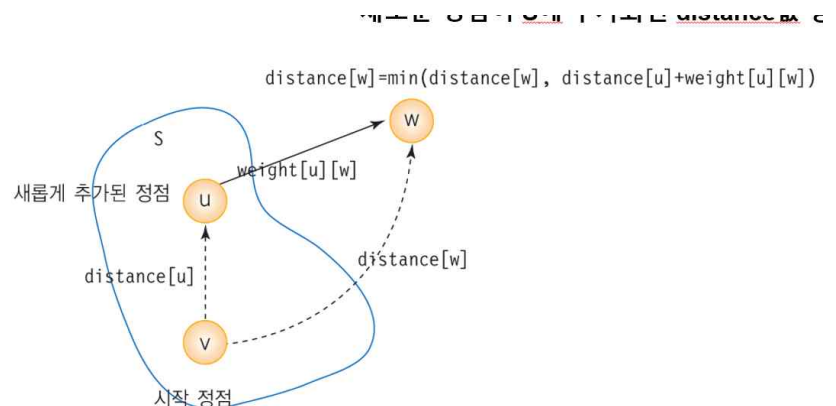
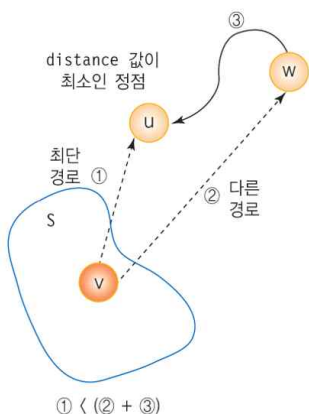
Floyd 알고리즘은 모든 정점에서 다른 모든 정점까지의 최단 경로를 계산

### ○ Dijkstra 알고리즘

하나의 시작 정점으로부터 모든 다른 정점까지의 최단 경로 찾는 알고리즘이다.



집합  $S$ 를 시작 정점  $v$ 로부터의 최단경로가 이미 발견된 정점들의 집합이라고 하자, 시작 정점에서 집합  $S$ 에 있는 정점만을 거쳐서 다른 정점으로 가는 최단거리를  $distance$  배열이라고 한다. 이제 매 단계에서 가장  $distance$  값이 작은 정점을  $S$ 에 추가할 것이다. 각 단계에서  $S$ 안에 있지 않은 정점 중에서 가장  $distance$ 값이 작은 정점을  $S$ 에 추가한다.



만약 정점  $w$ 를 거쳐서 정점  $u$ 로 가는 가상의 더 짧은 경로가 있다고 가정해보자, 그러면 정점  $v$ 에서 정점  $u$ 까지의 거리는 정점  $v$ 에서 정점  $w$ 까지의 거리 ②와 정점  $w$ 에서 정점  $u$ 로 가는 거리③을 합한 값이 된다. 그러나 경로 ②는 경로 ①보다 항상 길 수 밖에 없다. 왜냐하면 현재 distance 값이 가장 작은 정점은  $u$ 이기 때문이다.

따라서 오른쪽과 같은 코드를 사용해서 비교해야한다.

```
typedef struct Graph {
    int n; // 정점의 개수
    int weight[MAX_VERTICES][MAX_VERTICES];
} Graph;

int distance[MAX_VERTICES]; /* 시작정점으로부터의 최단경로 거리 */
int found[MAX_VERTICES]; /* 방문한 정점 표시 */

/* 최소 거리 정점을 선택하는 함수 */
int choose(int n) {
    int minpos = -1; // 가장 작은 노드의 인덱스
    int min = INT_MAX; // 가장 작은 거리 저장 변수

    // 방문하지 않은 정점 중에서 최소 거리 정점을 찾음
    for (int i = 0; i < n; i++) {
        if (distance[i] < min && !found[i]) {
            min = distance[i];
            minpos = i;
        }
    }
    return minpos;
}

/* 다익스트라 알고리즘을 이용한 최단 경로 찾기 함수 */
void shortest_path(Graph* g, int start) {
    // 초기화: 시작 정점으로부터의 거리와 방문 여부 설정
    for (int i = 0; i < g->n; i++) {
        distance[i] = g->weight[start][i]; // 시작 정점에서 각 정점으로의 거리 초기화
    }
    found[start] = TRUE; // 시작 정점 방문 표시
    distance[start] = 0; // 시작 정점의 거리는 0

    // 정점의 수만큼 반복
    for (int i = 0; i < g->n - 1; i++) {
        print_status(g); // 현재 상태 출력
        int u = choose(g->n); // 방문하지 않은 정점 중 최소 거리 정점 선택
        found[u] = TRUE; // 선택한 정점 방문 표시

        // 선택한 정점을 통해 인접한 정점들의 거리를 갱신
        for (int i = 0; i < g->n; i++) {
            if (!found[i]) { // 방문하지 않은 정점에 대해서만 처리
                if (distance[u] + g->weight[u][i] < distance[i]) {
```



```

        distance[i] = distance[u] + g->weight[u][i]; // 새로운 경로가 더 짧으면 거리 갱신
    }
}
}
}
}

```

### ○ 플로이드의 최단 경로 알고리즘

그래프에 존재하는 모든 정점 사이의 최단 경로를 구하려면 다익스트라 알고리즘을 정점 수만큼 반복 실행하면 된다. 그러나 모든 정점 사이의 최단 거리를 구하려면 플로이드 알고리즘을 써야 한다.

플로이드의 최단 경로 알고리즘은 2차원 배열 A를 이용해서 3중 반복을 하는 루프로 구성되어 있다. 인접 행렬 weight는 각 경로당 비용을 의미한다.

```

typedef struct Graph {
    int n; // 정점의 개수
    int weight[MAX_VERTICES][MAX_VERTICES]; // 인접 행렬로 가중치 저장
} Graph;

int A[MAX_VERTICES][MAX_VERTICES]; // 플로이드 알고리즘에서 사용될 거리 행렬

/* 플로이드-워셜 알고리즘을 이용한 최단 경로 계산 함수 */
void floyd(Graph* g) {
    // 초기화: 인접 행렬을 거리 행렬 A에 복사
    for (int i = 0; i < g->n; i++)
        for (int j = 0; j < g->n; j++)
            A[i][j] = g->weight[i][j];
    printA(g); // 초기 상태 출력

    // 플로이드-워셜 알고리즘
    for (int k = 0; k < g->n; k++) { // 경유하는 정점 k
        for (int i = 0; i < g->n; i++) { // 시작 정점 i
            for (int j = 0; j < g->n; j++) { // 도착 정점 j
                if (A[i][k] + A[k][j] < A[i][j]) // 경유지를 거치는 것이 더 짧으면
                    A[i][j] = A[i][k] + A[k][j]; // 거리 갱신
            }
        }
        printA(g); // 매 단계마다 상태 출력
    }
}

```

## ○ 최단 경로 알고리즘 복잡도

- Dijkstra의 최단경로 알고리즘 복잡도
  - 네트워크에  $n$ 개의 정점이 있다면, Dijkstra의 최단경로 알고리즘은 주반복문을  $n$ 번 반복하고 내부 반복문을  $2n$ 번 반복하므로  $O(n^2)$ 의 복잡도를 가진다.
- Floyd의 최단경로 알고리즘 복잡도
  - 네트워크에  $n$ 개의 정점이 있다면, Floyd의 최단경로 알고리즘은 3중 반복문을 실행되므로 시간 복잡도는  $O(n^3)$  이 된다
  - 모든 정점상의 최단경로를 구하려면 Dijkstra의 알고리즘  $O(n^2)$ 을  $n$ 번 반복해도 되며, 이 경우 전체 복잡도는  $O(n^3)$  이 된다

## ○ 위상 정렬

방향 그래프에 존재하는 각 정점들의 선행 순서를 위배하지 않으면서 모든 정점을 나열하는 것을 방향 그래프의 위상 정렬이라고 한다.

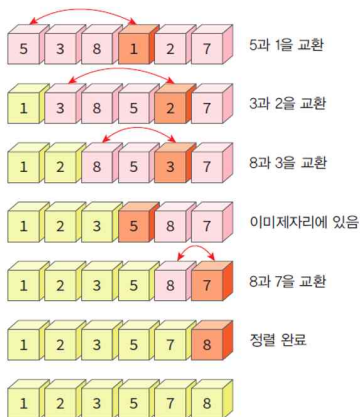
## 정렬

어떠한 데이터를 크기순으로 정리 하는 것을 말한다. 모든 경우에 최적인 정렬 알고리즘은 없으며, 단순하지만 비효율적인 방법 (삽입, 선택, 버블)과 복잡하지만 효율적인 방법(퀵, 히프, 합병, 기수)가 있을 뿐이다.

정렬은 내부정렬과 외부 정렬로 나눌 수도 있는데, 내부 정렬은 모든 데이터가 메모리에 저장되어진 상태에서 정렬 하는 것을 의미하고, 외부 정렬은 외부기억장치에 대부분의 데이터가 있고 일부만 메모리에서 처리하는 방법을 말한다.

정렬 알고리즘의 안정성을 체크하는 방법은 같은 값이라고 해도, 상대적인 위치가 변함이 없어야 한다.

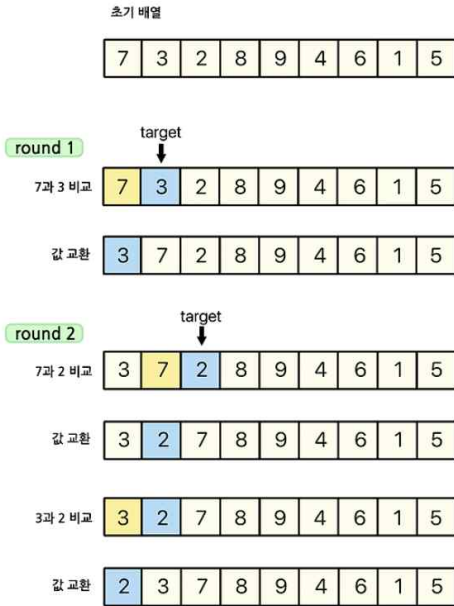
## ○ 선택 정렬



- 비교 횟수 :  $(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2 = O(n^2)$
- 이동 횟수 :  $3(n - 1)$
- 전체 시간적 복잡도:  $O(n^2)$
- 안정성을 만족하지 않음

```
#define SWAP(x, y, t) ( (t)=(x), (x)=(y), (y)=(t) )
void selection_sort(int list[], int n) // 이때 n은 원소 개수
{
    int temp;
    for (int i = 0; i < n; i++) {
        int min_index = i;
        for (int j = i + 1; j < n; j++) // 최소값 탐색
            if (list[min_index] > list[j])
                min_index = j;
        SWAP(list[i], list[min_index], temp);
    }
}
```

## ○ 삽입 정렬

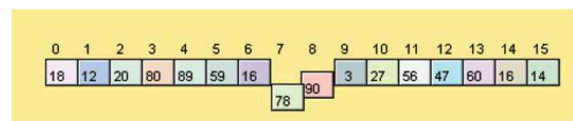


- 최선의 경우  $O(n)$  : 이미 정렬되어 있는 경우
  - 비교:  $n-1$  번
- 최악의 경우  $O(n^2)$  : 역순으로 정렬되어 있는 경우
  - 모든 단계에서 앞에 놓인 자료 전부 이동
  - 비교:  $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$
  - 이동:  $\frac{n(n-1)}{2} + 2(n-1) = O(n^2)$
- 평균의 경우  $O(n^2)$ 
  - 많은 이동 필요 -> 레코드가 클 경우 불리
- 안정된 정렬방법
  - 대부분 정렬되어 있으면 매우 효율적

```
void insertion_sort(int list[], int n)
{
    int key;
    int temp;
    for (int i = 1; i < n; i++) {
        key = list[i];
        for (int j = i; j >= 0; j--)
            if (list[j] < list[j - 1])
                SWAP(list[j], list[j - 1], temp);
    }
}
```

## ○ 버블 정렬

### 버블정렬(bubble sort)



- 인접한 2개의 레코드를 비교하여 순서대로 되어 있지 않으면 서로 교환

- 비교 횟수(최상, 평균, 최악의 경우 모두 일정)

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

- 이동 횟수

- 역순으로 정렬된 경우(최악의 경우)
  - 이동 횟수 = 3 \* 비교 횟수
- 이미 정렬된 경우(최선의 경우)
  - 이동 횟수 = 0
- 평균의 경우 :  $O(n^2)$



- 레코드의 이동 과다
  - 이동연산은 비교연산 보다 더 많은 시간이 소요됨

```

void bubble_sort(int list[], int n)
{
    int temp;
    for (int i = n - 1; i > 0; i--) {
        for (int j = 0; j < i; j++)
            /* 앞뒤의 레코드를 비교한 후 교체 */
            if (list[j] > list[j + 1]) SWAP(list[j], list[j + 1], temp);
    }
}

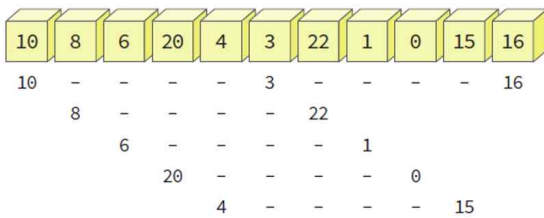
```

## ○ 셸 정렬

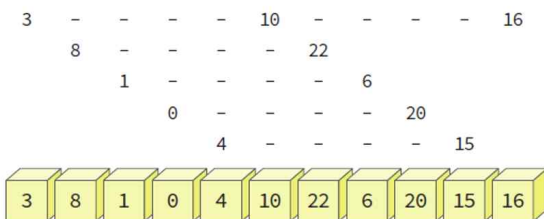
- 1) 삽입정렬이 어느 정도 정렬된 리스트에서 대단히 빠른 것에 착안
  - 삽입 정렬은 요소들이 이웃한 위치로만 이동하므로, 많은 이동에 의해서만 요소가 제자리를 찾아감
- 2) 요소들이 멀리 떨어진 위치로 이동할 수 있게 하면 보다 적게 이동하여 제자리 찾을 수 있음
- 3) 전체 리스트를 일정 간격(gap)의 부분 리스트로 나눔, 나뉘어진 각각의 부분 리스트를 삽입정렬 함

### 셸 정렬의 장점

불연속적인 부분 리스트에서 원거리 자료 이동으로 보다 적은 위치교환으로 제자리 찾을 가능성 증대한다. 부분 리스트가 점진적으로 정렬된 상태가 되므로 삽입정렬 속도 증가한다.



(a) 간격 5로 만들어진 부분 리스트



(b) 간격 5로 만들어진 부분 리스트

### 시간적 복잡도

- 최악의 경우  $O(n^2)$
- 평균적인 경우  $O(n^{1.5})$

간단하게 위와 같이 부분 리스트를 만들어 삽입 정렬을 한 뒤, 이번에는 간격을  $n/2$  정도 줄여서 또 다시 정렬한다.

```

void insertion_sort(int list[], int first, int last, int gap)
{
    int temp;
    for (int i = first + gap; i <= last; i += gap) {
        for (int j = i; j >= first; j--)
            if (list[j] < list[j - gap])
                SWAP(list[j], list[j - gap], temp);
    }
}

```

```

void shell_sort(int list[], int n)    // n = size
{
    for (int gap = n / 2; gap > 0; gap = gap / 2) {
        if ((gap % 2) == 0) gap++;
        for (int i = 0; i < gap; i++) // 부분 리스트의 개수는 gap
            insertion_sort(list, i, n - 1, gap);
    }
}

```

## ○ 합병 정렬

리스트를 두 개의 균등한 크기로 분할하고 분할된 부분리스트를 정렬  
정렬된 두 개의 부분 리스트를 합하여 전체 리스트를 정렬함

### ▪ 비교 횟수

- 크기  $n$ 인 리스트를 정확히 균등 분배하므로  $\log(n)$  개의 패스
- 각 패스에서 리스트의 모든 레코드  $n$ 개를 비교하므로  
 $n$ 번의 비교 연산

### ▪ 이동 횟수

- 레코드의 이동이 각 패스에서  $2n$ 번 발생하므로  
전체 레코드의 이동은  $2n \cdot \log(n)$ 번 발생
- 레코드의 크기가 큰 경우에는 매우 큰 시간적 낭비 초래
- 레코드를 연결 리스트로 구성하여 합병 정렬할 경우, 매우 효율적

- 최적, 평균, 최악의 경우 큰 차이 없이  $O(n \cdot \log(n))$ 의 복잡도
- 안정적이며 데이터의 초기 분산 순서에 영향을 덜 받음



```

int sorted[MAX_SIZE];    // 추가 공간이 필요

```

```

void merge_sort(int list[], int left, int right)
{
    int mid;
    if (left < right) {
        mid = (left + right) / 2;    /* 리스트의 균등 분할 */
        merge_sort(list, left, mid); /* 부분 리스트 정렬 */
        merge_sort(list, mid + 1, right); /* 부분 리스트 정렬 */
        merge(list, left, mid, right); /* 합병 */
    }
}

```

/\*  $i$ 는 정렬된 왼쪽 리스트에 대한 인덱스

$j$ 는 정렬된 오른쪽 리스트에 대한 인덱스

$k$ 는 정렬될 리스트에 대한 인덱스 \*/

```

void merge(int list[], int left, int mid, int right)

```

```

{
    int left_index1, mid_index, left_index2; // i, j, k
    left_index1 = left; mid_index = mid + 1; left_index2 = left;

```

```

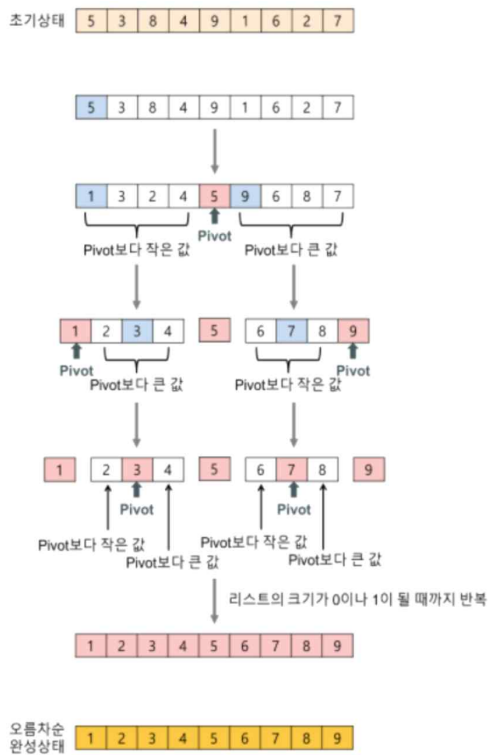
/* 분할 정렬된 list의 합병 */
while (left_index1 <= mid && mid_index <= right) {
    if (list[left_index1] <= list[mid_index]) sorted[left_index2++] = list[left_index1++];
    else sorted[left_index2++] = list[mid_index++];
}

if (mid_index > mid)/* 남아 있는 레코드의 일괄 복사 */
    for (int i = mid_index; i <= right; i++) sorted[left_index2++] = list[i];
else/* 남아 있는 레코드의 일괄 복사 */
    for (int i = left_index1; i <= mid; i++) sorted[left_index2++] = list[i];
/* 배열 sorted[]의 리스트를 배열 list[]로 재복사 */
for (int i = left; i <= right; i++) list[i] = sorted[i];
}

```

## ○ 퀵 정렬

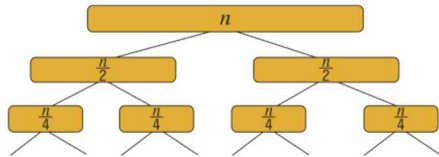
1. 평균적으로 가장 빠른 정렬 방법
2. 분할정복법 사용
3. 리스트를 2개의 부분리스트로 비균등 분할하고, 각각의 부분리스트를 다시 퀵정렬함(재귀호출)



## 퀵정렬(quick sort) 복잡도 분석

### 최선의 경우(거의 균등한 리스트로 분할되는 경우)

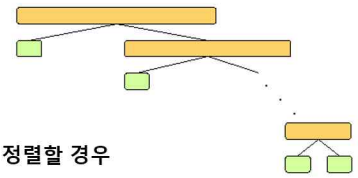
- 패스 수:  $\log(n)$ 
  - 2→1
  - 4→2
  - 8→3
  - ...
  - $n \rightarrow \log(n)$
- 각 패스 안에서의 비교횟수:  $n$
- 총 비교횟수:  $n \cdot \log(n)$
- 총 이동횟수: 비교횟수에 비하여 적으므로 무시 가능



### 최악의 경우(극도로 불균등한 리스트로 분할되는 경우)

- 패스 수:  $n$
- 각 패스 안에서의 비교횟수:  $n$
- 총 비교횟수:  $n^2$
- 총 이동횟수: 무시 가능
  - (예) 이미 정렬된 리스트를 정렬할 경우

(1 2 3 4 5 6 7 8 9)  
 1 (2 3 4 5 6 7 8 9)  
 1 2 (3 4 5 6 7 8 9)  
 1 2 3 (4 5 6 7 8 9)  
 1 2 3 4 (5 6 7 8 9)  
 1 2 3 4 5 6 7 8 9



- 중간값(medium)을 피벗으로 선택하면 불균등 분할 완화 가능

```

void quickSort(int* arr, int start, int end) {
    if (start >= end) return; // 원소가 1개인 경우 종료
    int pivot = start; // 피벗은 첫 번째 원소
    int left = start + 1;
    int right = end;
    while (left <= right) {
        // 피벗보다 큰 데이터를 찾을 때까지 반복
        while (left <= end && arr[left] <= arr[pivot]) left++;
        // 피벗보다 작은 데이터를 찾을 때까지 반복
        while (right > start && arr[right] >= arr[pivot]) right--;
        // 엇갈렸다면 작은 데이터와 피벗을 교체
        if (left > right) swap(arr[pivot], arr[right]);
        // 엇갈리지 않았다면 작은 데이터와 큰 데이터를 교체
        else swap(arr[left], arr[right]);
    }
    // 분할 이후 왼쪽 부분과 오른쪽 부분에서 각각 정렬 수행
    quickSort(arr, start, right - 1);
    quickSort(arr, right + 1, end);
}
    
```

## ○ 기수 정렬

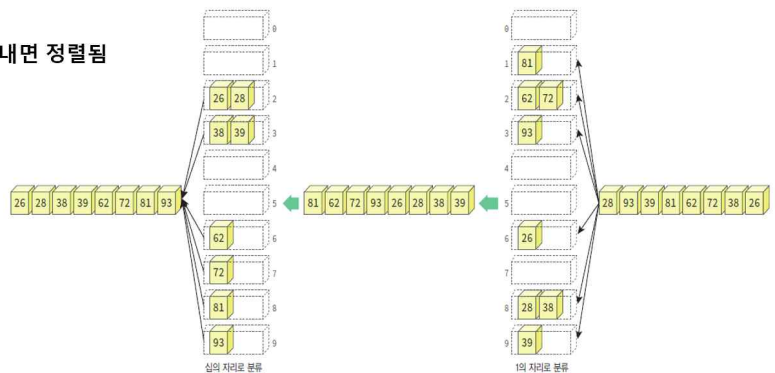
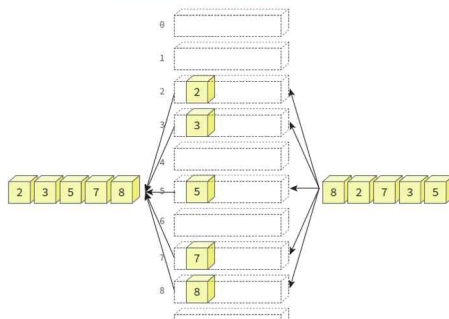
여태까지의 정렬 방법은 모두 레코드들을 비교하여 정렬했다. 하지만 기수 정렬은 레코드가 필요하지 않다. 이녀석은 기존 정렬의 통곡의 벽  $O(n \log n)$ 을 넘을 수 있다. 물론 단점은 추가적인 메모리가 필요하다는 것이다.

기수한 숫자의 자리수이다. 예를 들어 숫자 42는 숫자 4와 2 두 개의 자리수를 가지고 있다.

한자릿수의 작동 방식은 테이블로만 계산하는 “계수 정렬”과 같다. 하지만 2자리 수의 경우 100개의 테이블보단, 10개의 테이블만 형성하고 10의 자리로 분류한 뒤에 다시 1의 자리로 분류하면 된다.

테이블은 큐로 만드는데 제일 효율적이다.

- 대부분의 정렬 방법들은 레코드들을 비교함으로써 정렬 수행
- 기수 정렬(radix sort)은 레코드를 비교하지 않고 정렬 수행
  - 비교에 의한 정렬의 하한인  $O(n \log(n))$  보다 좋을 수 있음
  - 기수 정렬은  $O(dn)$ 의 시간적복잡도를 가짐(대부분  $d < 10$  이하)
- (예) 한자리수 (8, 2, 7, 3, 5)의 기수정렬
- 단순히 자리수에 따라 버킷(bucket)에 넣었다가 꺼내면 정렬됨



```
#define BUCKETS 10
#define DIGITS 4 // DIGITS는 정렬할 숫자의 최대 자릿수를 의미합니다. 예를 들어, DIGITS가 4로 설정되어
있으면, 최대 4자리 숫자(0에서 9999까지의 숫자)를 정렬할 수 있다는 의미입니다.
```

```
void radix_sort(int list[], int n)
{
    int factor = 1; // 자릿수를 위한 연산
    queue q[BUCKETS];

    for (int i = 0; i < BUCKETS; i++) init_queue(&q[i]); // 큐들의 초기화
    for (int i = 0; i < DIGITS; i++) {
        for (int j = 0; j < n; j++) // 데이터들을 자리수에 따라 큐에 삽입
            enqueue(&q[(list[j] / factor) % 10], list[j]);

        for (int k = i = 0; k < BUCKETS; k++) // 버킷에서 꺼내어 list로 합친다.
            while (!is_empty(&q[k]))
                list[i++] = dequeue(&q[k]);
        factor *= 10; // 그 다음 자리수로 간다.
    }
}
```



## 탐색

1. 여러 개의 자료 중에서 원하는 자료를 찾는 작업
2. 컴퓨터가 가장 많이 하는 작업 중의 하나
3. 탐색을 효율적으로 수행하는 것은 매우 중요

탐색키(search key) - 항목과 항목을 구별해주는 키(key)

### 순차 탐색

가장 간단하면서 직접적인 탐색 방법, 냅다 처음부터 끝까지 탐색하는 것을 말한다. 시간 복잡도는  $O(n)$

```
int seq_search(int key, int low, int high)
{
    for (int i = low; i <= high; i++)
        if (list[i] == key)
            return i; // 탐색 성공
    return -1;        // 탐색 실패
}
```

### 순차 탐색 업그레이드

앞선 순차 탐색의 문제는 매번마다 리스트의 값이 키 값인지 비교한다는 것이다. 이러한 비교를 딱 1번만 해주기 위해 아래와 같은 코드를 만들 수 있다.

```
int seq_search2(int key, int low, int high)
{
    list[high + 1] = key; // 키 값을 찾으면 종료
    for (int i = low; list[i] != key; i++)
        ;
    if (i == (high + 1)) return -1; // 탐색 실패
    else return i;                // 탐색 성공
}
```

### 이진 탐색

정렬된 배열의 중앙에 있는 값을 조사하여 찾고자 하는 항목이 왼쪽 또는 오른쪽 부분 배열에 있는지를 알아내어 탐색의 범위를 반으로 줄여가며 탐색 진행

• 5를 탐색하는 경우

7과 비교

1	3	5	6	7	9	11	20	30
---	---	---	---	---	---	----	----	----

5 < 7이므로 앞부분만을 다시 탐색

1	3	5	6
---	---	---	---

5를 3과 비교

1	3	5	6
---	---	---	---

5 > 3이므로 뒷부분만을 다시 탐색

5	6
---	---

5 == 5이므로 탐색 성공

5	6
---	---

(a) 탐색이 성공하는 경우

• 2를 탐색하는 경우

7과 비교

1	3	5	6	7	9	11	20	30
---	---	---	---	---	---	----	----	----

2 < 7이므로 앞부분만을 다시 탐색

1	3	5	6
---	---	---	---

2를 3과 비교

1	3	5	6
---	---	---	---

2 < 3이므로 앞부분만을 다시 탐색

1
---

2 > 1이므로 뒷부분만을 다시 검색

1
---

더 이상 남은 항목이 없으므로 탐색 실패

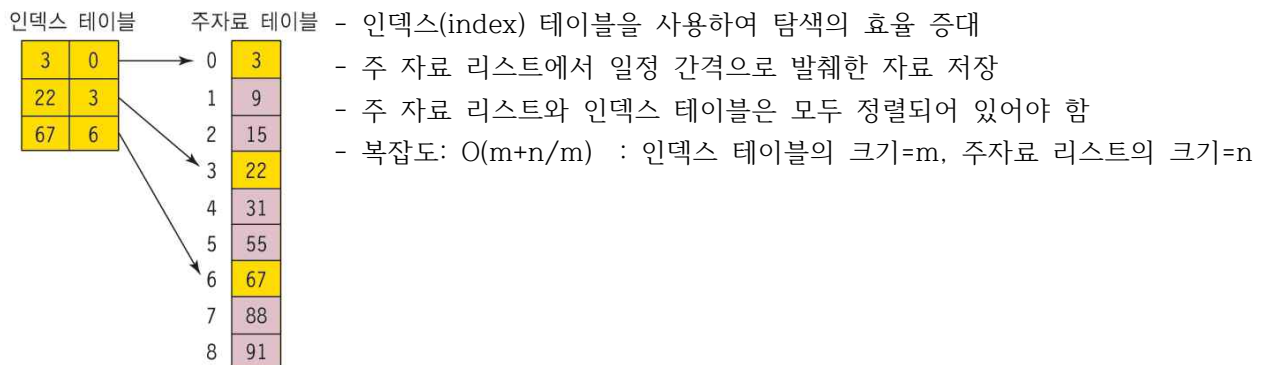
(b) 탐색이 실패하는 경우

```

int search_binary2(int key, int low, int high)
{
    int middle;
    while (low <= high) {                // 아직 숫자들이 남아 있으면
        middle = (low + high) / 2;
        if (key == list[middle]) return middle;    // 탐색 성공
        else if (key > list[middle]) low = middle + 1;    // 왼쪽 부분리스트 탐색
        else high = middle - 1;                // 오른쪽 부분리스트 탐색
    }
    return -1;                            // 탐색 실패
}

```

## ○ 색인 순차탐색



```

#define MAX_SIZE 1000
#define INDEX_SIZE 10

int list[MAX_SIZE] = { 3, 9, 15, 22, 31, 55, 67, 88, 91 };
int n = 9;
typedef struct {
    int key;
    int index;
} itable;

itable index_list[INDEX_SIZE] = { {3,0}, {15,3}, {67,6} };

int seq_search(int key, int low, int high)
{
    int i;
    for (i = low; i <= high; i++)
        if (list[i] == key) return i;    /* 탐색에 성공하면 키 값의 인덱스 반환 */
    return -1;    /* 탐색에 실패하면 -1 반환 */
}

/* INDEX_SIZE는 인덱스 테이블의 크기, n은 전체 데이터의 수 */
int index_search(int key)
{
    int i, low, high;
    /* 키 값이 리스트 범위 내의 값이 아니면 탐색 종료 */
}

```

```

if (key < list[0] || key > list[n - 1]) return -1;

/* 인덱스 테이블을 조사하여 해당키의 구간 결정 */
for (int i = 0; i < INDEX_SIZE; i++)
    if (index_list[i].key <= key && index_list[i + 1].key > key) break;

if (i == INDEX_SIZE) { /* 인덱스 테이블의 끝이면 */
    low = index_list[i - 1].index;
    high = n;
}
else {
    low = index_list[i].index;
    high = index_list[i + 1].index;
}
/* 예상되는 범위만 순차 탐색 */
return seq_search(key, low, high);
}

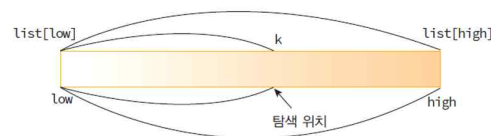
```

## ○ 보간 탐색

### 보간탐색 (interpolation search)

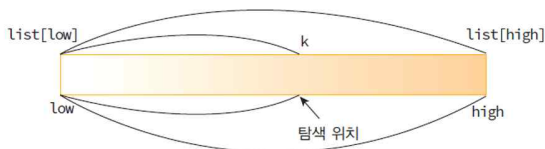
- 사전이나 전화번호부를 탐색하는 방법
  - 'ㅎ'으로 시작하는 단어는 사전의 뒷부분에서 찾을
  - 'ㄱ'으로 시작하는 단어는 앞부분에서 찾을
- 탐색기가 존재할 위치를 예측하여 탐색하는 방법:  $O(\log(n))$
- 보간 탐색은 이진 탐색과 유사하나 리스트를 불균등 분할하여 탐색

$$(list[high] - list[low]) : (k - list[low]) = (high - low) : (\text{탐색 위치} - low)$$



$$(list[high] - list[low]) : (k - list[low]) = (high - low) : (\text{탐색 위치} - low)$$

$$\text{탐색 위치} = (55 - 3) / (91 - 3) * (9 - 0) + 0 = 5.31 \approx 5$$



0	1	2	3	4	5	6	7	8	9
3	9	15	22	31	55	67	88	89	91

$$\text{탐색 위치} = \frac{(k - list[low])}{list[high] - list[low]} * (high - low) + low$$

```

int search_interpolation(int key, int n)
{
    int low, high, j;
    low = 0;
    high = n - 1;
    while ((list[high] >= key) && (key > list[low])) {
        j = ((float)(key - list[low]) / (list[high] - list[low]) * (high - low)) + low;
        if (key > list[j]) low = j + 1;
        else if (key < list[j]) high = j - 1;
        else low = j;
    }
    if (list[low] == key) return(low);
    else return -1;
}

```

주의점, 계산되어서 나오는 값은 실수이다. 실수를 정수로 변환할때 소수점 아래는 버리자