1971044 정희수
HW08_tech_02
전역 변수 / 구조체 / 함수 / main 함수 / 실행화면

#전역 변수

```
int weight[MAX_VERTICES][MAX_VERTICES] = {
    {0,3,INF,INF,INF,INF,INF,14},
    {3,0,8,INF,INF,INF,INF,10},
    {INF,8,0,15,2,INF,INF,INF},
    {INF,INF,15,0,INF,INF,INF,INF},
    {INF,INF,2,INF,0,9,4,5},
    {INF,INF,INF,INF,9,0,INF,INF},
    {INF,INF,INF,INF,4,INF,0,6},
    {14,10,INF,INF,5,INF,6,0}
};
```

위 변수는 그래프의 간선의 가중치를 2차원 배열에 넣어 구현한 것이다.
예를 들어 weight[0][1]의 값을 통해 0번 vertex와 1번 vertex를 잇는 edge의 가중치가 3이라는 것을 알 수 있다.

```
element selected[MAX_VERTICES];
Relation edge[MAX_VERTICES];
```

위 element 자료형의 1차원 배열은 MST를 구분하기 위해 선언한 변수이다.
위 Relation 자료형의 1차원 배열은 parent와 child를 구분하기 위해 선언한 변수이다.

#구조체

```
typedef struct {
    int num;//node number
    int key;//value
}element;
```

위 구조체는 힙에 넣을 노드의 vertex number와 key 값을 정의해 묶은 것이다.
int num; //node number
int key; //value

```
typedef struct {
    element heap[MAX_VERTICES];
    int size;
}HeapType;
```

위 구조체는 힙의 노드를 넣을 element 자료형의 1차원 배열과 힙의 노드 개수의 값을 정의해 묶은 것이다.
element heap[MAX_VERTICES]//MAX_VERTICES크기의 element 자료형 heap 배열 선언

int size//heap에 들어가있는 element의 개수

```
typedef struct {
    int parent;
    int child;
}Relation;
```

위 구조체는 노드의 부모와 자식의 vertex_number 값을 저장하기위해 묶은 것이다.
int parent; //부모 노드 vertex_number
int child; //자식 노드 vertex_number

#함수

```
void init(HeapType* h) {
    h->size = 0;
}
int is_empty(HeapType* h) {
    return(h->size == 0);
}
```

위 함수는 각각 힙을 초기화하고 힙이 비어있는지 확인하는 함수이다.
HeapType* h //h가 가리키는 HeapType의 구조체 포인터

```
element delete_min_heap(HeapType* h) {
    int parent, child;
    element item, temp;
    item = h->heap[0];
    temp = h->heap[--(h->size)];
    parent = 0;
    child = 1;
    while (child <= h->size) {
        if ((child < h->size) && (h->heap[child].key) > h->heap[child + 1].key)
            child++;
        if (temp.key <= h->heap[child].key)
            break;
        h->heap[parent] = h->heap[child];
        parent = child;
        child *= 2;
    }
    h->heap[parent] = temp;
    return item;
}
```

위 함수는 힙의 루트노드를 삭제하고 그 값을 return해준다.
HeapType* h //h가 가리키는 HeapType의 구조체 포인터
h는 min_heap으로 구성되어있기 때문에
h의 루트노드를 삭제한 후 min_heap의 규칙대로 다시 힙의 노드들을 sort해야 한다.
min_heap의 부모노드는 자식노드보다 항상 작아야 하므로 위 while문을 쓴다.

```
void Decrease_key_min_heap(HeapType* h, int i, int key) {
    element temp;
    if (key >= h->heap[i].key) {
        printf("error new key is not smaller than current key₩n");
        return;
    }
    h->heap[i].key = key;

    while ((i > 0) && (h->heap[i / 2].key > h->heap[i].key)) {
        temp = h->heap[i / 2];
        h->heap[i / 2] = h->heap[i];
        h->heap[i] = temp;
        i /= 2;
    }
}
```

heap의 노드안의 값보다 작은 값을 대입하여 min heap의 규칙대로 sort한다.
HeapType* h //h가 가리키는 HeapType의 구조체 포인터
int i //값을 변경할 heap의 노드의 index
int key //변경할 값
if문 // key값이 I번째 노드보다 크면 error 메시지를 출력하고 끝낸다.
원래 I번째 노드보다 작은 key값을 대입하면
min_heap이기 때문에 I번째 노드의 부모노드들만 비교해주면 된다.

```
int find_key(HeapType* h, int v_num) {
    for (int i = 0;i < MAX_VERTICES;i++) {
        if (h->heap[i].num == v_num)
            return h->heap[i].key;
    }
}
```

위 함수는 heap에서 vertex number가 v_num인 노드의 key 값을 return 한다.
HeapType* h //h가 가리키는 HeapType의 구조체 포인터
int v_num //vertex의 number

```
int find_num_idx(HeapType* h, int v_num) {
    for (int i = 0;i < MAX_VERTICES;i++) {
        if (h->heap[i].num == v_num)
            return i;
    }
}
```

위 함수는 heap에서 vertex number가 v_num인 노드의 heap idx를 return 한다.
HeapType* h //h가 가리키는 HeapType의 구조체 포인터
int v_num //vertex의 number

```
int find_parent(element item) {
    for (int i = 0;i < MAX_VERTICES;i++)
        for (int j = 0;j < MAX_VERTICES;j++) {
            if (weight[i][j] == item.key)
                return i;
        }
}
```

위 함수는 item의 key값과 같은 weight[i][j]값을 찾아 i를 return 한다.
여기서 I는 부모 노드의 vertex_num이다.
element item //element 자료형 변수

```
void insert_min_heap(HeapType* h, int item_num, int item_key) {
    int i;
    i = ++(h->size);
    while ((i != 1) && (item_key < h->heap[i / 2].key)) {
        h->heap[i].key = h->heap[i / 2].key;
        h->heap[i].num = h->heap[i / 2].num;
        i /= 2;
    }
    h->heap[i - 1].num = item_num;
    h->heap[i - 1].key = item_key;
}
```

위 함수는 힙에 노드를 삽입한다.
HeapType* h //h가 가리키는 HeapType의 구조체 포인터
int item_num //힙에 삽입할 item의 num(vertex_num)
int item_key //힙에 삽입할 item의 key(vertex_key)

```
void prim(HeapType* h, int s, int n) {
    int i, u, v;
    element item;
    for (i = 0;i < n;i++) { //selected와 edge배열을 초기화한다.
        selected[i].num = FALSE;
        selected[i].key = FALSE;
        edge[i].child = FALSE;
        edge[i].parent = FALSE;
    }
    Decrease_key_min_heap(h, find_num_idx(h, s), 0); //vertex_num이 0인 노드에 0 key 값을 대입한다.
    while (!is_empty(h)) { //힙이 빌 때까지 while문을 돌린다.
        item = delete_min_heap(h); //item은 힙에 있는 노드 중 가장 작은 key 값을 가지는 element를 받는다.
        u = item.num;
        selected[u] = item; //selected배열의 u번째 vertex 값이 FALSE가 아니게 된다.
        edge[u].child = u; //u의 child를 삽입한다.
        edge[u].parent = find_parent(item); //u의 parent를 찾아 edge에 삽입한다.
        for (v = 0;v < n;v++) {
            if (weight[u][v] != INF) { //weight값이 무한대가 아닐때
                if (selected[v].num==FALSE && weight[u][v] < find_key(h, v)) {
                    //아직 v번째 vertex가 MST에 없고, weight[u][v] 값이 기존의 find_key(h,v)값 보다 작다면
                    Decrease_key_min_heap(h, find_num_idx(h, v), weight[u][v]); //힙의 노드의 값을 바꿔준다.
                }
            }
        }
    }

}
```

위 함수는 주어진 graph에서 prim algorithm을 시행한다.
HeapType* h //h가 가리키는 HeapType의 구조체 포인터
int s //prim을 시작할 노드의 vertex 번호
int n //vertex 개수
prim 알고리즘은 시작 정점에서 출발하여 MST를 단계적으로 확장해 나간다.
(주석 참고)

```
void print_prim() {
    for (int i = 1;i < MAX_VERTICES;i++) {
        printf("Vertex %d -> %d", edge[i].parent, edge[i].child);
        printf("    edge : %d\n", selected[i].key);
    }
}
```

위는 prim algorithm을 시행한 결과물을 출력해주는 함수이다.

#main 함수

```
void main() {
    HeapType* h = (HeapType*)malloc(sizeof(HeapType));
    init(h);
    for (int i = 0;i < MAX_VERTICES;i++)
        insert_min_heap(h, i, INF);

    prim(h, 0, MAX_VERTICES);
    print_prim();
}
```

힙에 INF값을 가지는 노드를 MAX_VERTICES개수 만큼 삽입하고
prim 알고리즘을 시행했다.

#실행화면

```
Vertex 0 -> 1    edge : 3
Vertex 1 -> 2    edge : 8
Vertex 2 -> 3    edge : 15
Vertex 2 -> 4    edge : 2
Vertex 4 -> 5    edge : 9
Vertex 4 -> 6    edge : 4
Vertex 4 -> 7    edge : 5
```