REPORT

Data Structures

Student ID : 1771008
Name : Minjeong Kim

Homework8_1

변수(variable) 분석

element	
Type	**name**
int	key

HeapType	
Type	**name**
element[]	heap
int	heap_size

함수(function) 분석

int compare()
부모노드의 자식노드 중에 더 min값 갖는 것의 index를 리턴한다.

void increase_key_min_heap(element A[], int I, int key)
min_heap이기 때문에 parent에서 child로 값이 내려가는 양상을 보인다.
따라서, 자식 노드 중에 더 작은 값을 갖는 것을 parent로 올리기 위해 compare 함수를 사용한다.
선택한 인덱스 i의 key값을 바꾸고, 바뀐 key값에 따라 minheap을 재 정렬해야하기 때문에 번거로울 수 있지만, swap을 이용해서 자식노드 중에 더 작은 값을 갖는 것을 현재 위치(인덱스 i)의 노드와 위치를 바꾼다.
이 swap은 minheap이 만족될 때까지 반복한다.

void decrease_key_min_heap(element A[], int I, int key)
min_heap이기 때문에 child에서 parent로 값이 올라가는 양상을 보인다.
선택한 인덱스 i의 key값을 바꾸고, 바뀐 key 값에 따라 minheap을 재 정렬해야하기 때문에 번거로울 수 있지만, swap을 이용해서 부모노드와 현재위치(인덱스 i)의 노드와 위치를 바꾼다.
이 swap은 minheap이 만족될 때까지 반복한다.

전체 코드(full code)

```c

#include "stdlib.h"
#include "stdio.h"
#include "string.h"

 #define MAX_ELEMENT 100
 #define KEYS 10

 #define SWAP(x,y) { int t; t = x; x = y; y = t;}

typedef struct element {
    int key;
}element;

typedef struct HeapType {
    element heap[MAX_ELEMENT];
    int heap_size;
}HeapType;


void init(HeapType *h)
{
    h->heap_size = 0;
}

int is_empty(HeapType *h)
{
    if (h->heap_size == 0)
        return true;
    else
        return false;
}

int compare(int i2, int i21, element A[]) {
    if (A[i2].key < A[i21].key) return i2;
    else return i21;
}

void increase_key_min_heap(element A[], int i, int key) {
    if (key <= A[i].key) {
        printf("error: new key is not larger than current key");
        return;
    }
    A[i].key = key;
    int ind = compare(2 * i, 2 * i + 1, A);
    while ((i > 1) && (A[i].key > A[ind].key) && ind <= KEYS) {
        SWAP(A[i].key, A[ind].key);
        i = ind;
        ind = compare(2 * i, 2 * i + 1, A);
    }
}

void decrease_key_min_heap(element A[], int i, int key) {
    if (key >= A[i].key) {
        printf("error: new key is not smaller than current key");
        return;
    }
    A[i].key = key;
    while ((i > 1) && (A[i / 2].key > A[i].key)) {
        SWAP(A[i].key, A[i / 2].key);
        i /= 2;
    }
}

```

```c
int main()
{
    int keys[] = {1,4,2,7,5,3,3,7,8,9 };
    HeapType *h= (HeapType *)malloc(sizeof(HeapType));
    init(h);
    //hw8의 key 값으로 heap을 넣는다!
    for (int i = 0; i < KEYS; i++){
        h->heap[i+1].key = keys[i];
    }
    h->heap_size = KEYS+1;

    //index 4 (key = 7)에 있는 값을 3으로 감소한다.
    decrease_key_min_heap(h->heap, 4, 3);

    for (int i = 1; i < KEYS + 1; i++) {
        printf("%d ", h->heap[i].key);
    }
    printf("\n");

    //index 3 (key = 2)에 있는 값을 10으로 증가한다.
    increase_key_min_heap(h->heap, 3, 10);

    for (int i = 1; i < KEYS + 1; i++) {
        printf("%d ", h->heap[i].key);
    }
    return 0;
}
```

Console output

```
C:₩WINDOWS₩system32₩cmd.exe
1 3 2 4 5 3 3 7 8 9
1 3 3 4 5 3 10 7 8 9 계속하려면 아무 키나 누르십시오 . . .
```

Homework8_2

변수(variable) 분석

element	
Type	**name**
int	key
int	vertex

HeapType	
Type	**name**
element[]	heap
int	heap_size

TreeNode	
Type	**name**
element	node
TreeNode*	parent
TreeNode* []	child

TreeType	
Type	**name**
TreeNode*	root

사실 강의자료를 보면서, heap을 사용해서 hw를 할 때, 왜 그렇게 하는지 잘 이해가 안 갔다.

결국 본인은 heap을 사용한이유가 selected[] 배열을 대체하는 것이라고 결론을 내렸다.

heap으로 prim을 구현하기 까지는 element와 heaptype 변수가 사용되었고,
prim의 parent child관계를 구현하기 위해서는 TreeNode와 TreeType을 사용하였다.
따로 TreeNode와 TreeType을 선언한 이유는 HeapTyp의 element 배열 대신 TreeNode 배열을 넣어도,
prim의 과정에서 delete heap으로 인해서 어차피 다시 TreeNode를 다른 TreeType에 저장해야하기 때문이다.

함수(function) 분석

[prim 알고리즘 구현]

void build_min_heap()

distance가 담긴 배열인 dist의 모든 값을 element 타입의 적절한 값을 할당하여, insert_min_heap을 이용해서, heap에 insert 하는 작업을 dist의 크기만큼 반복한다.

void delete_min_heap / insert_min_heap

HeapType을 받아서 HeapType의 vertex값, 즉, dist 배열의 인덱스 값에 따라서 정렬하도록 하였다.

void heap_search()

prim알고리즘중에, 이미 heap에서 꺼내서 자신과 연결된 노드에 모두 접근하는데 성공한 (한번 선택된 selected 배열) node의 index를 선택하지않도록 k기 위한 함수이다.

prim()

1. distance의 값을 모두 INF로 설정한다 처음 시작하는 점의 dist 배열만 0으로 할당한다.
2. build_min_heap을 이용해서 distance의 값(사실상 distance의 인덱스)을 heap에 넣는다.
3. delete_min_heap을 이용해서 heap에 있는 element를 빼낸다. 만약에 이 heap에 존재하지 않는 인덱스라면 이미 prim알고리즘의 select된 순서임을 의미한다
4. heap으로 꺼낸 vertex값을 확인하고 이것은 dist의 index값임을 인지한다.
5. 해당 index의 index 값은 weight의 row로 사용되고, weight의 각 col에 접근하기 위해 for문을 사용한다.
6. weight의 row값의 인덱스를 가진 dist와 weight 값을 하나하나 비교하면서, dist의 값(vertex에 저장되있는 값)이 weight 값보다 클 경우 다시 할당한다. 이때, 자기 자신을 가리키는 weight 값이거나, heap에 없는 index 값에 접근하면 안된다.
7. prim 알고리즘에 맞게 dist 값, 즉, vertex의 값이 올바르게 변경되었는지 확인하기 위하여 출력한다.

[prim 알고리즘 + tree traversal]

TreeNode* find_indexNode()

해당하는 index를 갖고있는 node가 있는지를 탐색한다. node를 탐색하는 이유는 node가 linked list로 연결되어 있어, 같은 index값이라도 이미 있는 index라면 link를 이어가야하기 때문이다. 따라서 해당하는 인덱스의 TreeNode를 찾게되면 그 TreeNode의 주소값을 return한다.

void insert_node()

prim알고리즘에서 heap을 이용해서 꺼낸 node는 현재 노드 now_node(parent의 위치) 그리고, heap이 아니라 weight의 가중치를 확인하고 distance를 변경할 때의 element를 child_node라고하고 인자를 받는다. 받은 child element값을 node형태로 만드는데 이때 parent와 child 연결이 중요하다. prim알고리즘에서 parent와 child의 관계는 binary tree 관계가 아니기 때문에 하나의 parent에 연결된 여러 개의 child가 존재해서 child 배열을 선언했고, child node를 알맞게 넣는 것이 중요하다. 이 함수는 prim에서 parent에서 child를 지정하는 방식으로, prim 알고리즘이 진행됨에 따라 child에서 parent로 지정될 수 없기 때문에 선택하였다.

void setchild_init(n)

c언어에서 java의 arraylist에서 append를 사용하는 것처럼 사용하려면 또 새로운 ADT를 사용해야하기 때문에 임시적으로 child[i]의 값이 NULL인지 아닌지를 판단해서 append여부를 결정하기위해 child 배열의 모든 값을 NULL로 설정하는 함수이다.

next_child(n)

child배열에서 null인 값이 나오거나, 같은 vertex index를 가지는 값이 나오게 될 경우 해당 index를 리턴한다. 새로운 vertex index의 경우에는 append를 구현할 수 있게 해주고, 기존의 vertex index의 경우에는 set함수와 같은 역할을 한다.

전체 코드(full code)

```c

#include "stdlib.h"
#include "stdio.h"
#include "string.h"

#define MAX_ELEMENT 100
#define KEYS 10

#define MAX_VERTICES 8
#define INF 1000L

#define SWAP(x,y) { int t; t = x; x = y; y = t;}

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 } };


int selected[MAX_VERTICES]; // 이걸 heap으로 대신한다.
int dist[MAX_VERTICES + 1];



typedef struct element {
    int key; //dist
    int vertex; //index
}element;

typedef struct HeapType {
    element heap[MAX_ELEMENT];
    int heap_size;
}HeapType;

typedef struct TreeNode {
    element node; //vertex의 index값과 weight값이 담겨잇다.
    struct TreeNode* parent;
    struct TreeNode* child[MAX_VERTICES];
}TreeNode;

typedef struct TreeType {
    TreeNode* root;
}TreeType;

TreeType *t = (TreeType *)malloc(sizeof(TreeType));

void init(HeapType *h)
{
    h->heap_size = 0;
}

void t_init(TreeType *t) {
    t->root = NULL;
}

int is_empty(HeapType *h)
{
    if (h->heap_size == 0)
        return true;
    else
        return false;
}

```

```c
void insert_min_heap(HeapType *h, element item)
{
    int i;
    i = ++(h->heap_size);

    // compare it with the parent node in an order from the leaf to the root
    while ((i != 1) && (item.vertex < h->heap[i / 2].vertex)) {
        h->heap[i] = h->heap[i / 2];
        i /= 2;
    }
    h->heap[i] = item; // Insert new node
}

void build_min_heap(HeapType *h, int dist[], int n) { //n = dist의 개수
    for (int i = 0; i < n; i++) { //h->heap[i].key = dist와 같은것이라 하자.
        element item;
        item.key = dist[i];
        item.vertex = i;
        h->heap[i + 1] = item;
        insert_min_heap(h, item);
    }
    printf("\n");
}

bool heap_search(HeapType *h, int n) {
    for (int i = 1; i <= h->heap_size; i++) {
        if (h->heap[i].vertex == n)
            return true;
    }
    return false;
}

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

int compare(int i, element A[]) {
    if (A[i * 2].key < A[i * 2 + 1].key) return i * 2;
    else return i * 2 + 1;
}

void decrease_key_min_heap(element A[], int i, int key) {
    if (key >= A[i].key) {
        //printf("error: new key is not smaller than current key");
        return;
    }
    A[i].key = key;
    while ((i > 1) && (A[i / 2].key > A[i].key)) {
        SWAP(A[i].key, A[i / 2].key);
        SWAP(A[i].vertex, A[i / 2].vertex);
        i /= 2;
    }
}
```

```c
int next_child(TreeNode* node, int idx) {
    for (int i = 0; i < MAX_VERTICES; i++) {
        if (node->child[i] == NULL || node->child[i]->node.vertex == idx)
            return i;
    }
    return 0;
}

void setchild_init(TreeNode *n) {
    for (int i = 0; i < MAX_VERTICES; i++) {
        n->child[i] = NULL;
    }
}

void insert_node(TreeNode* now_node, element child_node, TreeType* h) {
    TreeNode *n = (TreeNode *)malloc(sizeof(TreeNode));
    if (n == NULL) return;

    n->node.key = child_node.key;
    n->node.vertex = child_node.vertex;
    setchild_init(n);

    if (now_node->node.vertex == 0) {
        now_node->parent = NULL;
        h->root = now_node;
    }
    n->parent = now_node;
    int idx = next_child(now_node, n->node.vertex);
    now_node->child[idx] = n;


}

TreeNode* find_indexNode(TreeNode* node, int idx, TreeType* th) {
    //처음 node는 treetype -> root로 한다.
    if (node != NULL) {
        if (node->node.vertex == idx)
            return node;
        else {
            for (int i = 0; i < MAX_VERTICES; i++) {
                find_indexNode(node->child[i], idx, th);
            }
            return th->root;
        }
    }
    return NULL;
}

void prim(int s, int n) { // n=vertex의 개수
                    //int i, v;
    int count = 0;

    for (int u = 0; u <= n; u++) {
        dist[u] = INF; //dist inf로 모두 설정한다.
                    //selected[u] = FALSE;
    }
    dist[s] = 0; //start 지점 0으로 설정

    HeapType *h = (HeapType *)malloc(sizeof(HeapType));
    init(h);


    build_min_heap(h, dist, n); //n = MAX_VERTECIES


    for (int i = 0; i < n; i++) {
        element p = delete_min_heap(h); //heap의 용도: 그냥 select의 기능을 없앤것...?

        int cost = p.key;
        int here = p.vertex;
        if (dist[here] == INF) return;
        for (int v = 0; v < n; v++) {
```

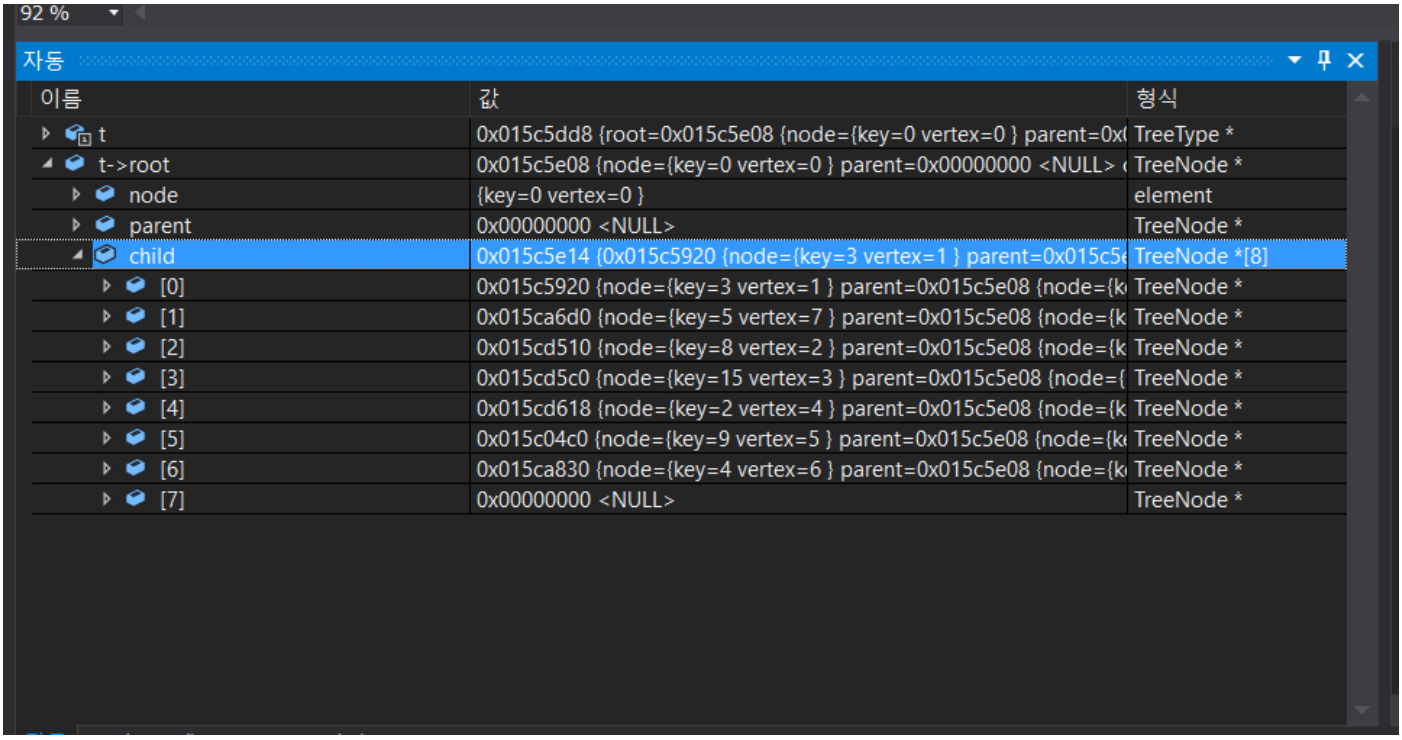
```
209                 if (weight[here][v] != INF) {
210                     if (dist[v] > weight[here][v] && i != v && heap_search(h, v)) {
211                         dist[v] = weight[here][v];
212 
213                         TreeNode* preNode;
214                         if (!count) {
215                             preNode = (TreeNode*)malloc(sizeof(TreeNode));
216                             preNode->node = p;
217                             setchild_init(preNode);
218                         }
219                         else {
220                             //p.key = dist[v];
221                             preNode = find_indexNode(t->root, p.vertex, t);
222                         }
223                         count++;
224 
225                         element next;
226                         next.key = dist[v];
227                         next.vertex = v;
228 
229                         insert_node(preNode, next, t);
230                     }
231                 }
232 
233             }
234 
235             for (int i = 0; i < MAX_VERTICES; i++) {
236                 printf("%d ", dist[i]);
237             }
238             printf("\n");
239 
240         }
241     }
242 
243 void print_prim(TreeNode *node) {
244     //child 입장에서!!
245     if (node != NULL && node->parent != NULL) {
246         printf("Vertex %d -> %d           edge: %d\n", node->parent->node.vertex, node->node.vertex, node->node.key);
247         for (int i = 0; i < MAX_VERTICES; i++) {
248             print_prim(node->child[i]);
249         }
250     }
251 }
252 
253 int main()
254 {
255     t_init(t);
256     prim(0, MAX_VERTICES);
257     print_prim(t->root);
258 }
```

Console output

```
C:₩WINDOWS₩system32₩cmd.exe

0 3 1000 1000 1000 1000 1000 14
0 3 8 1000 1000 1000 1000 10
0 3 8 15 2 1000 1000 10
0 3 8 15 2 1000 1000 10
0 3 8 15 2 9 4 5
0 3 8 15 2 9 4 5
0 3 8 15 2 9 4 5
0 3 8 15 2 9 4 5
계속하려면 아무 키나 누르십시오 . . .
```

Debug



child_parent에서 깊게 들어가도록 하는 부분에 대해서 코딩이 미완성인 상태이다.
따라서 parent child가 저런식으로 관계 맺어져있음을 확인했고, 조금만 시간을 투자하면 충분히 parent -> child로의 접근 tree travelsal을 할 수 있을 것 같다. 하지만 시간이 부족해서 완성하지 못했다.

Homework9_3

변수(variable) 분석

element	
Type	**name**
int	key
int	vertex

HeapType	
Type	**name**
element[]	heap
int	heap_size

TreeNode	
Type	**name**
element	node
TreeNode*	parent
TreeNode* []	child

TreeType	
Type	**name**
TreeNode*	root

함수(function) 분석

변수와 함수 모두 HW 9_2와 같으므로 설명을 생략하겠다. (tree traversal 문제역시 같은 양상이다.)
여기서 다른 부분을 꼽자면 213, 214 줄이다.

```
if (dist[v] > weight[here][v] + dist[here] && i != v && heap_search(h, v)) {
    dist[v] = weight[here][v] + dist[here];
```

원래는 dist[v] > weight[here][v] 였던것들이 현재 row의 dist를 반영하기위하여 + dist[here]을 이용해서 graph가 접근한 길이를 누적해서 저장하였다.

전체 코드(full code)

```c

#include "stdlib.h"
#include "stdio.h"
#include "string.h"

#define MAX_ELEMENT 100
#define KEYS 10

#define MAX_VERTICES 8
#define INF 1000L

#define SWAP(x,y) { int t; t = x; x = y; y = t;}

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 } };


int selected[MAX_VERTICES]; // 이걸 heap으로 대신한다.
int dist[MAX_VERTICES + 1];



typedef struct element {
    int key; //dist
    int vertex; //index
}element;

typedef struct HeapType {
    element heap[MAX_ELEMENT];
    int heap_size;
}HeapType;

typedef struct TreeNode {
    element node; //vertex의 index값과 weight값이 담겨잇다.
    struct TreeNode* parent;
    struct TreeNode* child[MAX_VERTICES];
}TreeNode;

typedef struct TreeType {
    TreeNode* root;
}TreeType;

TreeType *t = (TreeType *)malloc(sizeof(TreeType));

void init(HeapType *h)
{
    h->heap_size = 0;
}

void t_init(TreeType *t) {
    t->root = NULL;
}

int is_empty(HeapType *h)
{
    if (h->heap_size == 0)
        return true;
    else
        return false;
}

```

```c
void insert_min_heap(HeapType *h, element item)
{
    int i;
    i = ++(h->heap_size);

    // compare it with the parent node in an order from the leaf to the root
    while ((i != 1) && (item.vertex < h->heap[i / 2].vertex)) {
        h->heap[i] = h->heap[i / 2];
        i /= 2;
    }
    h->heap[i] = item; // Insert new node
}

void build_min_heap(HeapType *h, int dist[], int n) { //n = dist의 개수
    for (int i = 0; i < n; i++) { //h->heap[i].key = dist와 같은것이라 하자.
        element item;
        item.key = dist[i];
        item.vertex = i;
        h->heap[i + 1] = item;
        insert_min_heap(h, item);
    }
    printf("\n");
}

bool heap_search(HeapType *h, int n) {
    for (int i = 1; i <= h->heap_size; i++) {
        if (h->heap[i].vertex == n)
            return true;
    }
    return false;
}

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

int compare(int i, element A[]) {
    if (A[i * 2].key < A[i * 2 + 1].key) return i * 2;
    else return i * 2 + 1;
}

void decrease_key_min_heap(element A[], int i, int key) {
    if (key >= A[i].key) {
        //printf("error: new key is not smaller than current key");
        return;
    }
    A[i].key = key;
    while ((i > 1) && (A[i / 2].key > A[i].key)) {
        SWAP(A[i].key, A[i / 2].key);
        SWAP(A[i].vertex, A[i / 2].vertex);
        i /= 2;
    }
}
```

```c
int next_child(TreeNode* node, int idx) {
    for (int i = 0; i < MAX_VERTICES; i++) {
        if (node->child[i] == NULL || node->child[i]->node.vertex == idx)
            return i;
    }
    return 0;
}

void setchild_init(TreeNode *n) {
    for (int i = 0; i < MAX_VERTICES; i++) {
        n->child[i] = NULL;
    }
}

void insert_node(TreeNode* now_node, element child_node, TreeType* h) {
    TreeNode *n = (TreeNode *)malloc(sizeof(TreeNode));
    if (n == NULL) return;

    n->node.key = child_node.key;
    n->node.vertex = child_node.vertex;
    setchild_init(n);

    if (now_node->node.vertex == 0) {
        now_node->parent = NULL;
        h->root = now_node;
    }
    n->parent = now_node;
    int idx = next_child(now_node, n->node.vertex);
    now_node->child[idx] = n;


}

TreeNode* find_indexNode(TreeNode* node, int idx, TreeType* th) {
    //처음 node는 treetype -> root로 한다.
    if (node != NULL) {
        if (node->node.vertex == idx)
            return node;
        else {
            for (int i = 0; i < MAX_VERTICES; i++) {
                find_indexNode(node->child[i], idx, th);
            }
            return th->root;
        }
    }
    return NULL;
}

void prim(int s, int n) { // n=vertex의 개수
                    //int i, v;
    int count = 0;

    for (int u = 0; u <= n; u++) {
        dist[u] = INF; //dist inf로 모두 설정한다.
                    //selected[u] = FALSE;
    }
    dist[s] = 0; //start 지점 0으로 설정

    HeapType *h = (HeapType *)malloc(sizeof(HeapType));
    init(h);


    build_min_heap(h, dist, n); //n = MAX_VERTECIES


    for (int i = 0; i < n; i++) {
        element p = delete_min_heap(h); //heap의 용도: 그냥 select의 기능을 없앤것...?

        int cost = p.key;
        int here = p.vertex;
        if (dist[here] == INF) return;
        for (int v = 0; v < n; v++) {
```

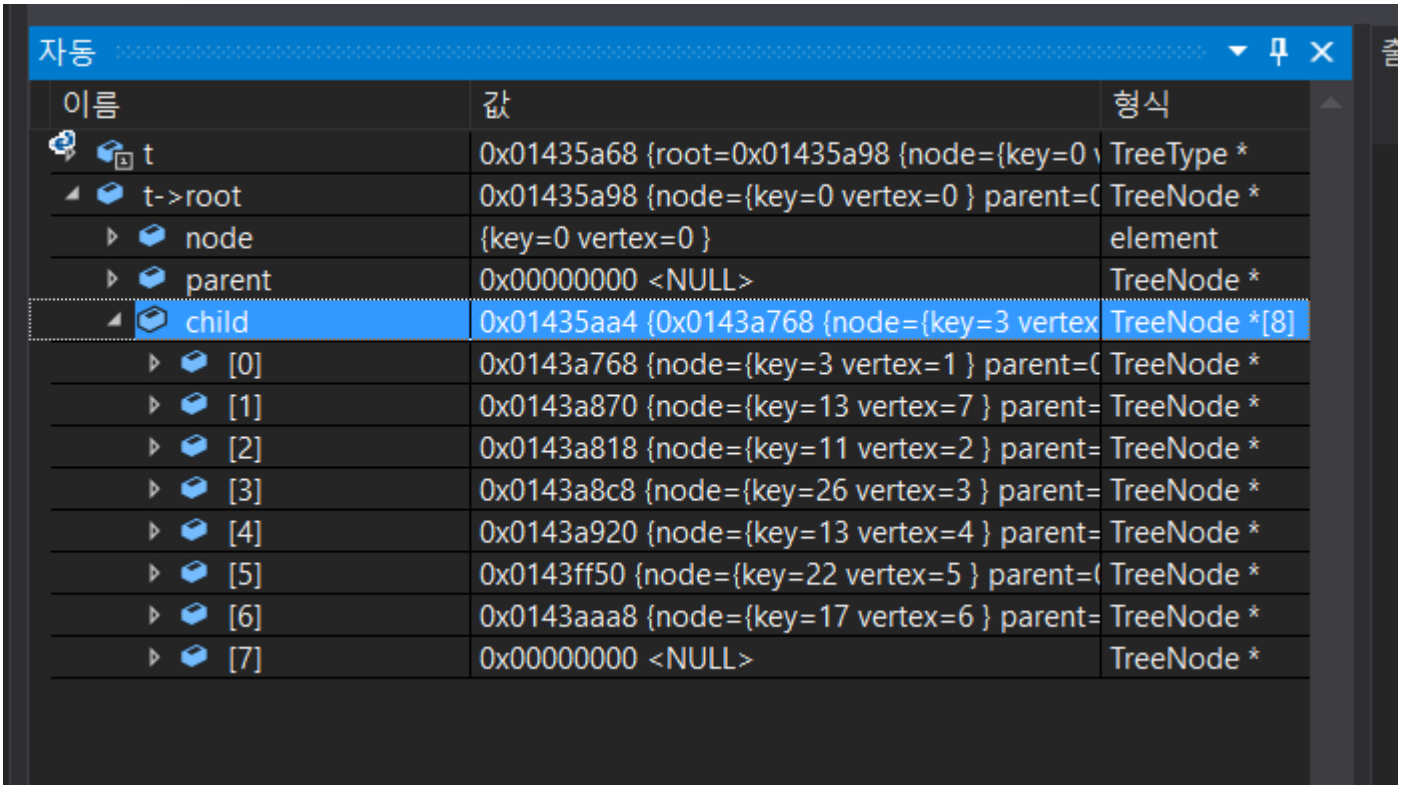
```
212             if (weight[here][v] != INF) {
213                 if (dist[v] > weight[here][v] + dist[here] && i != v && heap_search(h, v)) {
214                     dist[v] = weight[here][v] + dist[here];
215 
216                     TreeNode* preNode;
217                     if (!count) {
218                         preNode = (TreeNode*)malloc(sizeof(TreeNode));
219                         preNode->node = p;
220                         setchild_init(preNode);
221                     }
222                     else {
223                         //p.key = dist[v];
224                         preNode = find_indexNode(t->root, p.vertex, t);
225                     }
226                     count++;
227 
228                     element next;
229                     next.key = dist[v];
230                     next.vertex = v;
231 
232                     insert_node(preNode, next, t);
233                 }
234             }
235         }
236         for (int i = 0; i < MAX_VERTICES; i++) {
237             printf("%d ", dist[i]);
238         }
239         printf("\n");
240 
241     }
242 }
243 
244 
245 void print_difkstra(TreeNode *node) {
246     //child 입장에서!!
247     if (node != NULL && node->parent != NULL) {
248         printf("Vertex %d -> %d           edge: %d\n", node->parent->node.vertex, node->node.vertex, node->node.key);
249         for (int i = 0; i < MAX_VERTICES; i++) {
250             print_difkstra(node->child[i]);
249         }
250     }
251 }
252 
253 int main()
254 {
255     t_init(t);
256     prim(0, MAX_VERTICES);
257     print_prim(t->root);
258 }
```

Console output

```
C:₩WINDOWS₩system32₩cmd.exe

0 3 1000 1000 1000 1000 1000 14
0 3 11 1000 1000 1000 1000 13
0 3 11 26 13 1000 1000 13
0 3 11 26 13 1000 1000 13
0 3 11 26 13 22 17 13
0 3 11 26 13 22 17 13
0 3 11 26 13 22 17 13
0 3 11 26 13 22 17 13
계속하려면 아무 키나 누르십시오 . . .
```

Debug



HW2의 prim과 마찬가지로 child_parent에서 깊게 들어가도록 하는 부분에 대해서 코딩이 미완성인 상태이다. 따라서 parent child가 저런식으로 관계 맺어져있음을 확인했고, 조금만 시간을 투자하면 충분히 parent -> child로의 접근 tree travelsal을 할 수 있을 것 같다. 하지만 시간이 부족해서 완성하지 못했다.