**Homework 8\_1 [min\_heap.cpp]**

**Variable analysis**

#define MAX\_ELEMENT 100

#define INITIAL\_ELEMENT 10

int initial\_value[10] = { 1, 4, 2, 7, 5, 3, 3, 7, 8, 9};

heap : heap represented in array

heap\_ size : size of heap

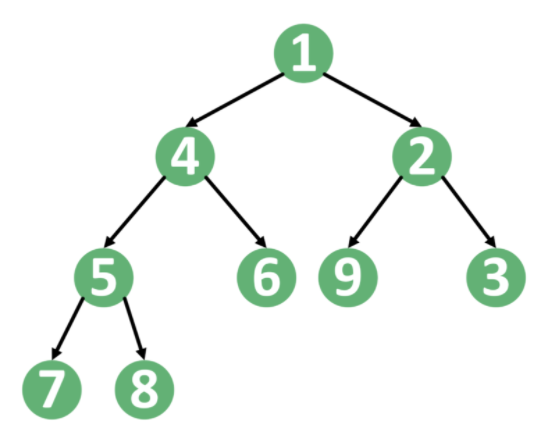
|  |  |
| --- | --- |
| **HeapType** | |
| **type** | **name** |
| Element [MAX\_ELEMENT] | heap |
| int | heap\_ size |

Key : the criteria for sorting heap.

|  |  |
| --- | --- |
| **element** | |
| **type** | **name** |
| int | key |

**Min heap**

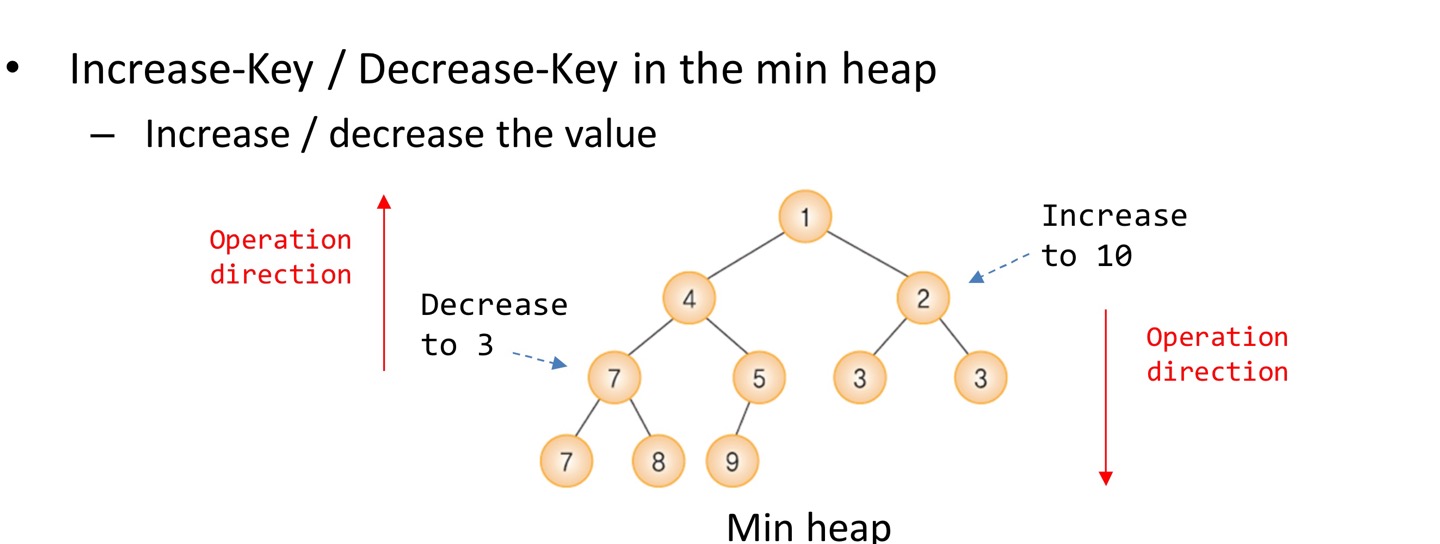
To begin with, the heap data structure is a complete binary tree. Therefore, the min-heap data structure is a complete binary tree, **where each node has a smaller value than its children**. Consequently, **each node has a larger value than its parent**.



As we can see, each key is smaller than its children and larger than its parent. Therefore, the smallest value will always be the root of the tree. However, storing the heap data structure in a complete binary tree is complex. Hence, we use an array to store it.

First, we store the value of the root in index 1. Next, we store the left child in index 2 and the right child in index 3. Generally, if we stored a node in index , **we store its left child in index 2\*i and its right child in index 2\*i.**

**The Increase and Decrease Key is needed for update key value in certain vertex.**

****

**Function analysis**

1) void init(HeapType\* h)

// Initialization

void init(HeapType\* h) {

h->heap\_size = 0;

for (int i = 0; i < INITIAL\_ELEMENT; i++) {

element node;

node.key = initial\_value[i];

h->heap[i+1] = node;

h->heap\_size++;

}

}

: creates node and insert it into heap.

2) void Decrease\_key\_min\_heap(HeapType\* h, int i, int key)

//decrease the element i’s value to ‘key’

void Decrease\_key\_min\_heap(HeapType\* h, int i, int key)

{

if (key >= h->heap[i].key)

fprintf(stderr, "new key is not smaller than current key\n");

h->heap[i].key=key;

element temp = h->heap[i];

// The process of comparing with the parent node

while ((i > 1) && (key < h->heap[i / 2].key)) { //child < parent

h->heap[i] = h->heap[i / 2]; // parent = child

i /= 2; // move Up

}

h->heap[i] = temp;

}

**: When Decrease-key happens, we don’t need to compare with its child node. Because it is originally smaller than its child. so the opertation direction is DOWN.**

3) void Increase\_key\_min\_heap(HeapType\* h, int i, int key)

//Increase the element i’s value to ‘key’

void Increase\_key\_min\_heap(HeapType\* h, int i, int key)

{

if (key <= h->heap[i].key)

fprintf(stderr, "new key is not larger than current key\n");

h->heap[i].key = key;

// The process of comparing with the child node

while ((i <= h->heap\_size) ){ // 부모 > 자식

int child = 2 \* i;

if (child >= h->heap\_size)

break;

if ((h->heap[child].key) > h->heap[child + 1].key)

child++; // child : left right중 더 작은값

if (h->heap[i].key <= h->heap[child].key) // 부모 < 자식

break;

// SWAP

element tmp = h->heap[i];

h->heap[i] = h->heap[child]; //부모 = 자식

h->heap[child] = tmp; //자식 = 부모

// move down one level

i = child;

}

}

**: When Increase-key happens, we don’t need to compare with its parent node. Because it is originally bigger than its parent.**

4) void print\_heap(HeapType\* h)

void print\_heap(HeapType\* h) {

printf("heap : [");

for (int i = 1; i <= INITIAL\_ELEMENT; i++) {

printf("%d ", h->heap[i].key);

}

printf("]\n");

}

: print heap state

**Simulation**

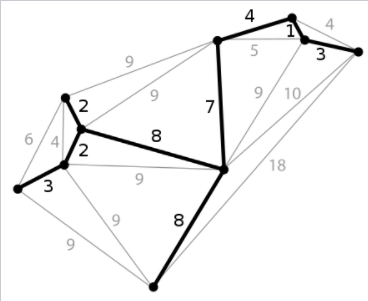
**[result]**

텍스트이(가) 표시된 사진

자동 생성된 설명

**Homework 8\_2 [prim.cpp]**

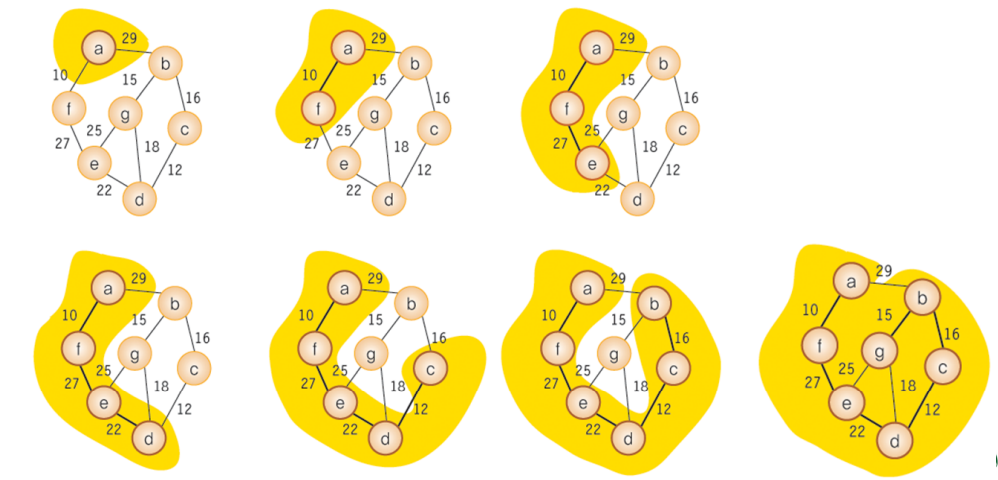
**Minimum Spanning Tree (MST)**

****

A **minimum spanning tree** (**MST**) or **minimum weight spanning tree** is a subset of the edges of a connected edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible.

**Prim Algorithm**

Prim’s algorithm is also a Greedy Algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.   
A group of edges that connects two set of vertices in a graph is called cut graph theory. *So, at* every step of Prim’s algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

****

The idea behind Prim’s algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

**1)** Create a set \**mstSet* that keeps track of vertices already included in MST.   
**2)** Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.   
**3)** While mstSet doesn’t include all vertices   
….**a)** Pick a vertex *u* which is not there in *mstSet*and has minimum key value.   
….**b)** Include *u*to mstSet.   
….**c)** Update key(distance) value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*

\* In this code, the **selected []** functions like mstSet[]

The idea of using key values is to pick the minimum weight edge from cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

**Variable analysis**

#define MAX\_VERTICES 8 : # of maximum vertexs

#define INF 1000L : infinite value to be used in comparing weight between edges

#define MAX\_ELEMENT 100 : # of maximum element in heap

int selected[MAX\_VERTICES] : check whether the vertex is visited or not. also means it is included in MST or not

int dist[MAX\_VERTICES] = {INF, } : dist[k] saves minimum distance to the vertex `k`.

int parent[MAX\_VERTICES] : parent[k] saves vertex k’s parent node

heap : heap represented in array

heap\_ size : size of heap

|  |  |
| --- | --- |
| **HeapType** | |
| **type** | **name** |
| Element [MAX\_ELEMENT] | heap |
| int | heap\_ size |

distance : the key value in sorting heap.

vertex : the id of vertex

|  |  |
| --- | --- |
| **element** | |
| **type** | **name** |
| int | distance |
| int | vertex |

**Function analysis**

1) void init(HeapType\* h)

// Initialization

void init(HeapType\* h) {

h->heap\_size = 0;

}

: initializes the heap.

2) void build\_min\_heap(HeapType\* h)

void build\_min\_heap(HeapType\* h)

{

int parent, child;

element temp;

int i;

for (int i = h->heap\_size / 2; i >= 1; i--) {

parent = i;

child = 2 \* i; //left child

temp = h->heap[parent]; //the element that we have to find proper place to be inserted in.

// The process of comparing with the parent node as it traverses the tree

while (child <= h->heap\_size) {

if ((child + 1 <= h->heap\_size) && (h->heap[child].distance > h->heap[child + 1].distance))

child++; // indicates the smaller value between left and right child.

if (temp.distance <= h->heap[child].distance) // if the

break;

h->heap[parent] = h->heap[child];

// Move down one level

parent = child;

child = parent \* 2;

}

h->heap[parent] = temp;

}

}

: In the min\_heap, the key at the parent node is always less than the key at both child nodes. **To build a min heap ,** From the n/2 to 1 , move the child up until you reach the root node and the heap property is satisfied.

3) void Decrease\_key\_min\_heap(HeapType\* h, int v, int key)

//decrease the vertex v’s value to ‘key’

void Decrease\_key\_min\_heap(HeapType\* h, int v, int key)

{

//find node with vertex 'v'

int j, i;

for (j = 1; j <= h->heap\_size; j++) {

if (h->heap[j].vertex == v) {

i = j;

break;

}

}

if (key >= h->heap[i].distance)

fprintf(stderr, "new key is not smaller than current key\n");

h->heap[i].distance = key;

element temp = h->heap[i];

// The process of comparing with the parent node

while ((i > 1) && (key < h->heap[i / 2].distance)) { //child < parent

h->heap[i] = h->heap[i / 2]; //child =parent

i /= 2;

}

h->heap[i] = temp;

}

: First, it finds the an element with a given vertex ID(To decrease the value of a certain key inside the min-heap, we need to reach this key first. )and initialize `i` to the index of that element. Then Moving Up to find proper place to insert that element. The detailed explanations are in **Homework 8\_1 [min\_heap.cpp].**

4) element delete\_min\_heap(HeapType\* h)

To remove/delete a root node in a min heap

* Delete the root node.
* Move the key of the last child to root.
* Compare the parent node with its children.
* If the value of the parent is greater than child nodes, swap them, and repeat until the heap property is satisfied.

// Delete the root at heap h, (# of elements: heap\_size)

element delete\_min\_heap(HeapType\* h)

{

int parent, child;

element item, temp;

item = h->heap[1];

temp = h->heap[(h->heap\_size)--]; //allocate last element to root.

parent = 1;

child = 2;

//move until you reach over the heap\_size of the heap property is satisfied

while (child <= h->heap\_size) {

// Find a smaller child node

if ((child < h->heap\_size) && (h->heap[child].distance > h->heap[child + 1].distance))

child++;

if (temp.distance <= h->heap[child].distance) break;

// Move down one level

h->heap[parent] = h->heap[child];

parent = child;

child \*= 2;

}

h->heap[parent] = temp;

return item;

}

5) void insert\_all\_vertices(HeapType\* h, int n)

void insert\_all\_vertices(HeapType\* h, int n) {

init(h);

for (int i = 0; i < MAX\_VERTICES; i++) {

element v;

v.vertex = i;

v.distance = dist[i]; // the initial distance to i is INF

h->heap[i+1] = v;

h->heap\_size++;

}

build\_min\_heap(h);

}

: Simply create vertex 1 to MAX\_VERTIVES and initializes each element. Then, insert them into min\_heap (priority queue) and make build\_min\_heap()

6) void print\_heap(HeapType\* h)

void print\_heap(HeapType\* h) {

printf("heap : [");

for (int i = 1; i < MAX\_VERTICES; i++) {

printf("%d ", h->heap[i].distance);

}

printf("]\n");

}

:This function is used to know heap state in each loop.

7) void print\_prim(int n)

void print\_prim(int n) {

for (int i = 1; i < n; i++) {

printf("Vertex %d -> %d edge : %d\n", parent[i], i, dist[i]);

}

}

: print out created MST by `prim` using parent[i] and dist[i]. parent[i]. After prim(), parent[i] saves the i’s parent node which has edge to i with minimum cost.

8) void prim(int s, int n, HeapType\* h)

// n: the number of vertices on the graph

// s: the starting point

void prim(int s, int n, HeapType\* h)

{

int i, u, v,d;

element node;

for (u = 0; u < n; u++)

{

dist[u] = INF;

selected[u] = FALSE;

}

dist[s] = 0;

insert\_all\_vertices(h, n);

for (i = 0; i <= n-1; i++) {

print\_heap(h);

print\_dist(MAX\_VERTICES);

node = delete\_min\_heap(h); // get the minimum dist node

u = node.vertex;

d = node.distance;

selected[u] = TRUE; // include to MSTset.

if ( d==INF) //there is no way to d.

return;

printf("[current vertex : %d ]\n", u);

for (v = 0; v < n; v++) {

if (weight[u][v] != INF) { // if there is a way (u->v)

if (!selected[v] && weight[u][v] < dist[v]) //if v is not included in MSTset and the dist is the smaller.

{

dist[v] = weight[u][v]; // renewal dist to v

printf("renewal distance %d(from) -> %d(to) as %d\n",u,v,weight[u][v]);

Decrease\_key\_min\_heap(h, v, weight[u][v]); // decreases key(distance) value at vertex `v` in the min\_heap.

parent[v] = u; // save v’s parent.

}

}

}

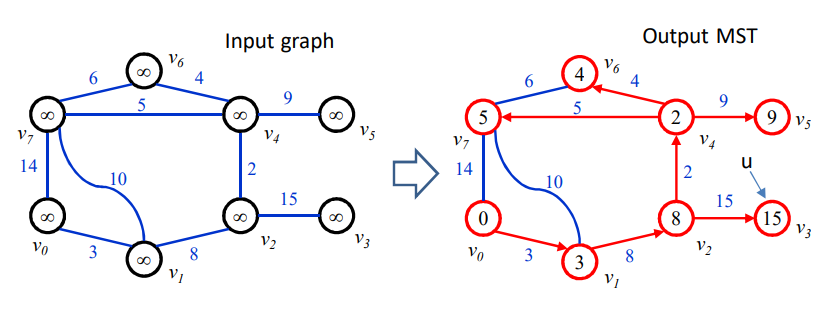
printf("\n\n");

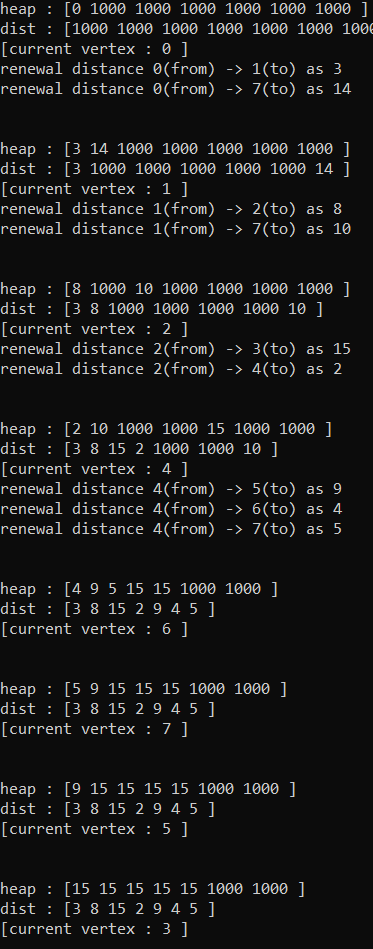
}

}

**Simulation**

**[Input]**

****

****

: the order of visiting node is 0->1->2->4->6->7->5->3

**[result]**

: the print\_prim() results are as below

**텍스트, 전자기기, 키보드이(가) 표시된 사진

자동 생성된 설명**