알 고 리 즘

Homework 4

Heapsort, Prim's algorithm

Professor	황호영 교수님
Department	컴퓨터공학과
Student ID	2012722028
Name	장 한 별
Date	2018. 11. 22.

Introduction.

이번 프로젝트의 목표는 **Heap sort**와 **Prim's algorithm**을 이해하는 것이다. Heap sort를 binary heap을 이용하여 구현해보고, 어떻게 동작하는지 하나하나 과정을 살펴보도록 한다. 또한 Priority Queue(Min Heap)은 Prim's algorithm을 이해하는 부분에 있어 필수적이다. 어떤 과정에서 사용되는지 확인해보고 Prim's algorithm 또한 어떻게 동작하는지 직접 구현하며 알아보는 것을 목표로 한다.

1. Consider the heapsort algorithm using the binary heap.

(a) Write your program with your comments.

addition.h

addition.h | prob4-1.cpp

Project9 | (전역 범위) | Arrlength(int A[])

```
#include <iostream>
using namespace std;
#define len 15

void PRINT_TREE(int A[]);       // 배열을 Tree 모양으로 출력하는 함수
int Parent(int i);      // Tree에서 i index인 node 의 parent node의 index를 가져오는 함수
int Left(int i);        // 왼쪽 child 의 index를 가져오는 함수
int Right(int i);       // 오른쪽 child 의 index를 가져오는 함수
void PRINT_ARRAY(int A[]); // 배열의 모든 요소들을 순차적으로 출력하는 함수
int Arrlength(int A[]);     // 배열의 길이를 가져오는 함수

void PRINT_TREE(int A[])
{
    int i = 1;  // 입력된 배열의 첫번째 요소 부터
    while (1)
    {
        if (i == len + 1 || A[i] == 0)  // 배열의 끝까지 이르면 종료
            break;
        if (i == 1)     // level 1
            printf("\t\t\t%d\n\n", A[i]);
        else if (i >= len / 2 / 2 / 2 + 1 && i <= len / 2 / 2) {    // level 2
            printf("\t\t%d", A[i]);
            if (i == len / 2 / 2)
                printf("\n\n");
        }
        else if (i >= len / 2 / 2 + 1 && i <= len / 2) {        // level 3
            printf("\t    %d", A[i]);
            if (i == len / 2)
                printf("\n\n\t");
        }
        else if (i >= len / 2 + 1 && i <= len) {        // level 4
            printf("   %d", A[i]);
        }
        i++;        // 다음 index로 이동
    }
    printf("\n-------------------------------------------------------\n");
}
```

```
39  int Parent(int i)
40  {
41      return i / 2;
42  }
43  int Left(int i)
44  {
45      return i * 2;
46  }
47  int Right(int i)
48  {
49      return i * 2 + 1;
50  }
51  void PRINT_ARRAY(int A[])
52  {
53      for (int i = 1; i <= len; i++)  // 0번째 index는 X
54          if(A[i] !=0)
55              printf("%d ", A[i]);
56      printf("\n");
57  }
```

```
58  int Arrlength(int A[])
59  {
60      int length = 0;
61
62      while (1)
63      {
64          if (A[length] == 0) // index를 하나씩 증가시키다가
65              break;          // 끝에 이르면 종료
66          length++;
67      }
68      length--;           // 배열의 길이 + 1 되어있으므로 하나 빼줌
69
70      return length;
71  }
72
```

prob4-1.cpp

addition.h prob4-1.cpp

Project9 (전역 범위)

```
1   #include "addition.h"
2
3   int MAX_HEAPIFY(int A[], int i); // MAX Heap 화하는 함수
4   void BUILD_MAX_HEAP(int A[]);
5   void HEAPSORT(int A[]);
6
7   int main(void)
8   {
9       // index 계산의 간편함을 위해 0 자리에 99999999 삽입
10      int A[16] = {99999999, 4, 1, 3, 2, 16, 9, 10, 14, 8, 7};
11
12      printf("INPUT: ");
13      PRINT_ARRAY(A);
14      HEAPSORT(A);
15
16      return 0;
17  }
```

```
19  int MAX_HEAPIFY(int A[], int i)
20  {
21      i = Parent(i);      // i번째 값이 그 parent의 값보다 클 때 함수가 호출되므로
22      if (i == 0)
23          return 1;
24      int larger = 0;     // 변수 초기화
25      int temp = 0;
26      int l = 0, r = 0;
27      int length = 0;
28      l = Left(i);        // left child
29      r = Right(i);       // right child
30      length = Arrlength(A);
31
32      if (l <= length && A[l] > A[i]) // left child가 더 큰 경우
33          larger = l;     // left child 를 더 큰 값에
34      else
35          larger = i;     // 그대로
36
37      if (r <= length && A[r] > A[larger])    // right child가 더 큰 경우
38          larger = r;     // right child 를 더 큰 값에
39
40      if (larger != i)    // parent가 child 보다 큰 상황
41      {
42          temp = A[i];        // swap parent with child
43          A[i] = A[larger];
44          A[larger] = temp;
45          MAX_HEAPIFY(A, temp);       // sub tree 까지 확인을 위해 재귀 함수 호출
46      }
47      return 1;
48  }
49  void BUILD_MAX_HEAP(int A[])
50  {
51      int i = 1, j = 0;
52      int length = 0;
53      length = Arrlength(A);
54      j = length / 2; // build 시 leaf node 들은 Max Heapify 해줄 필요 없음
55
56      while (1)
57      {
58          if (j < 1)  // 배열의 끝까지 이르면 종료
59              break;
60
61          i = 1;  // Build 를 위한 index 초기화
62
```

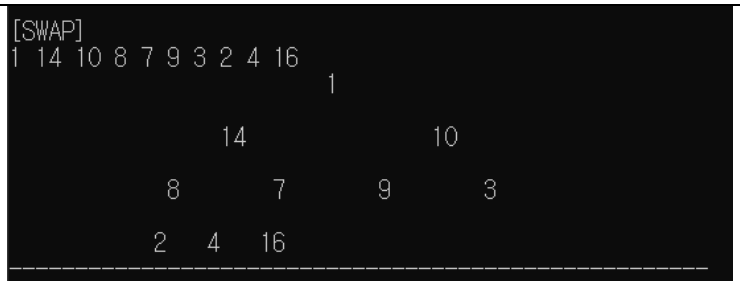
```
63          while (1)
64          {
65              if (i == length + 1 || A[i] == 0)   // 배열의 끝까지 이르면 종료
66                  break;
67
68              if (A[Parent(i)] < A[i])    // 해당 값이 parent 보다 클 때
69              {
70                  MAX_HEAPIFY(A, i);
71                  i = 1;      // index 초기화
72                  continue;
73              }
74              i++;    // index 하나씩 증가
75          }
76          j--;
77      }
78  }
```

```
79  void HEAPSORT(int A[])
80  {
81      int i = 0;
82      int temp = 0;
83      int temparr[16] = { 100, };
84
85      BUILD_MAX_HEAP(A);
86
87      int length = 0;             // 마지막 index에 접근 하기위해
88      length = Arrlength(A);      // 배열의 길이 저장
89
90      while (1)
91      {
92          if (length < 1)
93              break;
94
95          printf("[MAX HEAPIFY]\n");
96          PRINT_ARRAY(A);
97          PRINT_TREE(A);
98
99          temp = A[1];            // root 와
100         A[1] = A[length];       // 가장 마지막 index에 있는 값
101         A[length] = temp;       // 교환
102
```

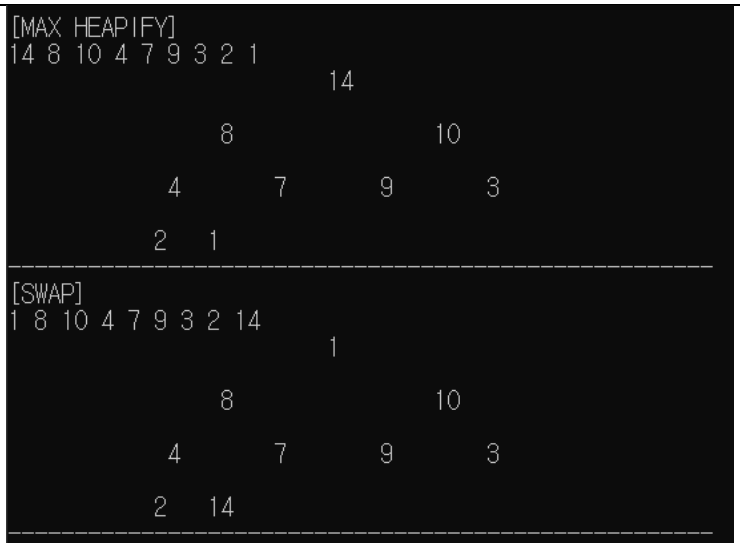
```
103         printf("[SWAP]\n");
104         PRINT_ARRAY(A);
105         PRINT_TREE(A);
106
107         temparr[length] = A[length]; // 가장 큰 수가 배열 마지막에 위치하게
108         A[length] = 0;          // 빠진 자리 0으로 채움
109         length--;               // 가장 큰 수 빼줬으니 -1
110
111         BUILD_MAX_HEAP(A);      // 다시 Heap화 한다
112     }
113
114     A = temparr;            // temp에 넣어놨던 가장 큰 수 옮김
115     printf("OUTPUT: ");
116     PRINT_ARRAY(A);
117 }
118
```

(b) For at least one example, show the simulation results in step-by-step detail.

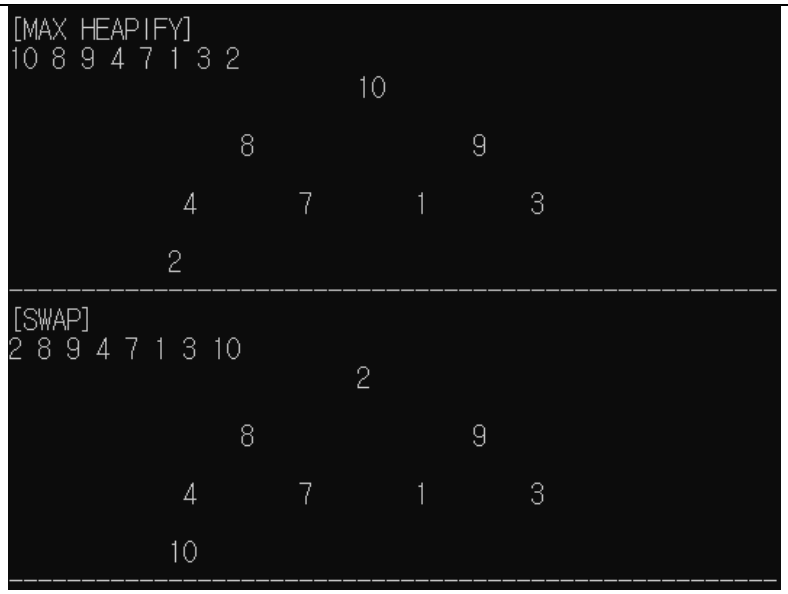
위 그림은 강의자료의 예시인 배열 {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}을 입력 후 Max Heapify 한 후, binary tree 형태로 출력한 결과이다. 최대값인 16이 root 자리에 있고, 각 요소들이 Max heap을 이루고있는 것을 확인할 수 있다. Heap Sort는 이렇게 구현된 Max Heap 에서 최대값인 root 와 leaf 노드들 중 제일 마지막 노드와 swap 후, 그 수를 sort될 정렬의 마지막 자리에 놓는 것으로 이루어진다.



위 그림은 Root 자리에 있는 16과 leaf 노드들 중 제일 마지막 노드인 1이 swap 된 화면이다. 이후 16은 heap에서 제외한 후 나머지 노드들로 다시 Max heap 형태를 갖춰야한다.



다시 Max heapify 로 남은 요소들중 최대값인 14가 root 자리에 오른 것을 확인할 수 있다. 그 후 14와 leaf노드들 중 제일 마지막 노드인 1을 swap후 14 역시 heap에서 제외시킨다. 아래 그림들은 이와 같은 과정을 순차적으로 출력한 화면이다.

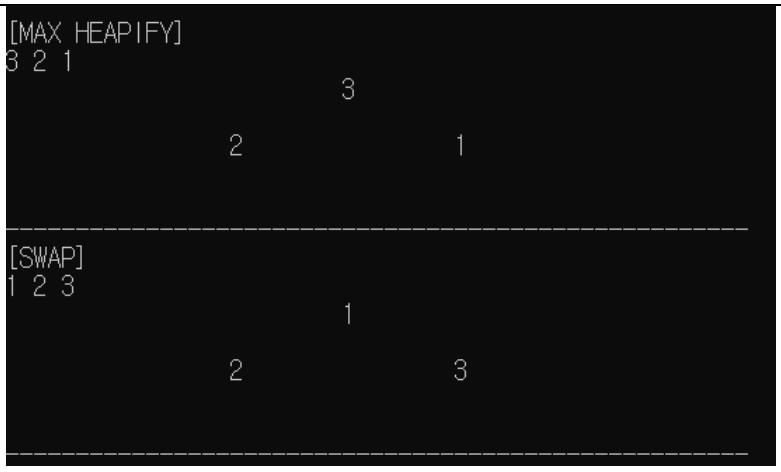

[MAX HEAPIFY]
10 8 9 4 7 1 3 2
10
8 9
4 7 1 3
2
[SWAP]
2 8 9 4 7 1 3 10
2
8 9
4 7 1 3
10

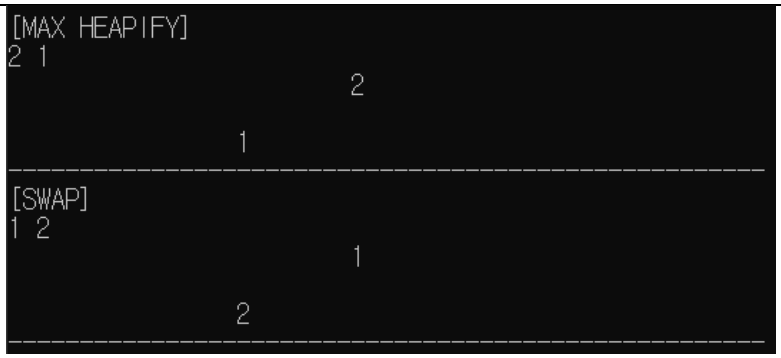

[MAX HEAPIFY]
9 8 3 4 7 1 2
9
8 3
4 7 1 2
[SWAP]
2 8 3 4 7 1 9
2
8 3
4 7 1 9

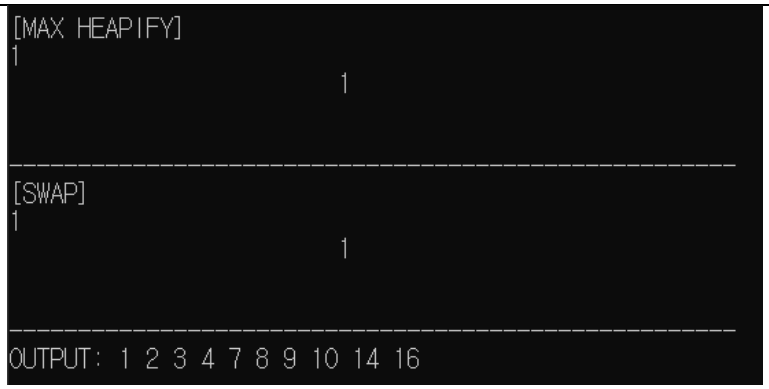

[MAX HEAPIFY]
8 7 3 4 2 1
8
7 3
4 2 1
[SWAP]
1 7 3 4 2 8
1
7 3
4 2 8


[MAX HEAPIFY]
7 4 3 1 2
7
4 3
1 2
[SWAP]
2 4 3 1 7
2
4 3
1 7


[MAX HEAPIFY]
4 2 3 1
4
2 3
1
[SWAP]
1 2 3 4
1
2 3
4


[MAX HEAPIFY]
3 2 1
3
2 1
[SWAP]
1 2 3
1
2 3


[MAX HEAPIFY]
2 1
2
1
[SWAP]
1 2
1
2



제일 처음 가장 큰 수였던 16이 빠져나가 배열의 마지막에 위치하고, 그 다음 14, 10, 9... 뒤에서부터 완성되어 sort 된 것을 확인할 수 있다. Output이 {1, 2, 3, 4, 7, 8, 9, 10, 14, 16} 순으로 제대로 출력된 것으로 보아 heap sort가 올바르게 동작했음을 확인할 수 있다.

(c) Explain the program and the simulation results (at least five lines).

addition.h 파일의 설명을 우선으로 한다. addition.h 파일은 heap sort algorithm에 큰 영향이 있지않지만 heap sort가 이루어짐을 확인함에 있어 필요한 함수들을 넣어놨다. Tree구조(PRINT_TREE)나 배열의 형태(PRINT_ARRAY)를 출력해주는 함수, 배열의 길이를 가져오는 함수(Arrlength), parent 노드(Parent), left child(Left), right child(Right)를 추적하는 함수가 여기에 존재한다.

prob4-1.cpp 파일에는 max heapify하는 함수(MAX_HEAPIFY), max heap을 build하는 함수(BUILD_MAX_HEAP), heap sort 하는 함수(HEAPSROT) 그리고 main함수가 여기에 존재한다. Main 함수에 sort를 원하는 배열의 요소들을 넣어준다. 필자는 강의자료의 예시인 {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}을 배열 A에 넣어줬다. 실제 구현에 있어 index 계산의 편리함을 위해 0번째 index에는 99999999(Sort를 구현하는 algorithm에는 크게 영향이 없다.)를 넣어줬다.

먼저 HEAPSORT를 호출하면 max heap에 대한 build가 이루어진다. Build시, leaf node들은 max heapify 해줄 필요가 없으므로, 제외시킨다. 배열의 요소들을 첫번째 index부터 하나하나 비교하며 그 값이 parent 보다 클 때 max heapify 시킨다. 배열의 끝까지 수행했다면 함수는 종료된다. Max heapify는 child node가 parent보다 클 때 호출되는데, left child가 parent node보다 큰지 right child가 parent node보다 큰 지 구분 해준 후, 두 node를 swap한다. 해당 sub tree 까지 모두 max heap을 만족할 때 까지 recursive 로 호출하여 root 부터 leaf까지 모두 max heapify 하도록 한다.

Build 와 heapify로 완성된 Max heap tree의 root는 모든 요소들의 최대값이다. 이는 index 1에 위치하고, 가장 마지막 노드는 배열의 길이만큼의 index에 존재하므로 이를 swap한다. Swap된 최대값은 heap에서 제외시키고, sort 배열의 가장 마지막 위치에 넣어둔다. 그 후 다시 max heap 형태를 만들어, 만들어진 max heap tree의 최대값을 다시 마지막 노드와 swap 후, 제외 시켜, sort 배열의 가장 마지막 위치 바로 앞부분에 삽입한다. 이런 식으로 큰 수가 뒤에서부터 정렬된다.

실행 결과에 대한 설명은 편의를 위해 1. (b)에 그림과 같이 적어두었다.

2. Consider Prim's algorithm using the binary heap.

(a) Write your program with your comments.

prob4-2.cpp

Project10 (전역 범위)

```cpp
#include <iostream>
#include <queue>
#include <cstdlib>
#include <ctime>
using namespace std;

int A[8][8] = {             // 그래프 및 edge 정보를 저장
    {0, 6, 12, 0, 0, 0, 0, 0},
    {6, 0, 5, 0, 14, 0, 0, 8},
    {12, 5, 0, 9, 0, 7, 0, 0},
    {0, 0, 9, 0, 0, 0, 0, 0},
    {0, 14, 0, 0, 0, 0, 0, 3},
    {0, 0, 7, 0, 0, 0, 15, 10},
    {0, 0, 0, 0, 0, 15, 0, 0},
    {0, 8, 0, 0, 3, 10, 0, 0}
};

class V {           // vertex에 대한 각종 정보를 저장
public:
    int Vertex;     // 몇 번째 vertex 인지
    int Extract;    // Extract 되었는지
    int Weight;     // 가중치
    int Path;       // 이전 노드 저장
};

priority_queue< int, vector<int>, greater<int> > Q;

void Print_Table(V v[]);    // 그래프 정보 table 형태로 출력하는 함수
void Print_Min();           // 최소값 출력하는 함수
int Extract_Min(V v[]);     // 최소값 Extract 하는 함수
int PRIM(V v[], int vtx);   // Prim's algorithm 적용하는 함수

int flag = 0;       // Recursive 몇 번 하는지 정하는 변수

int main(void)
{
    int num = 0;
    V v[8];                 // 8개의 vertex

    int i = 0, j = 0;
    int vtx = 0;

    for (i = 0; i < 8; i++)         // vertex table 초기화
    {
        v[i].Vertex = i;
        v[i].Extract = 0;           // Extract 0으로 초기화, 나중에 Extract 되면 1
        v[i].Weight = 99999999;     // 무한대를 99999999로 표시
        v[i].Path = -1;             // 이전 노드를 저장, 아직 접근하지 않는 vertex는 -1로 초기화
    }
    Print_Table(v);
```

```
52      srand((unsigned int)time(NULL));        // 임의의 숫자로
53      vtx = rand() % 8;                       // 맨 처음 vertex 정함
54
55      //vtx = 5;
56      v[vtx].Weight = 0;              // 선택된 vertex의 key 0으로
57
58      PRIM(v, vtx);
59
60      return 0;
61  }
62
63  void Print_Table(V v[])         // Vertex 정보 table 형태로 출력하는 함수
64  {
65      int i = 0;
66
67      for (i = 0; i < 8; i++)     // table(2차원 배열)로 출력
68          printf("%d %d %d %d\n", v[i].Vertex, v[i].Extract, v[i].Weight, v[i].Path);
69      printf("-------------------\n");
70  }
71  void Print_Min()
72  {
73      printf("%d ", Q.top());         // Min Heap 의 root(top) 는 최소값
74      printf("\n-------------------\n");
75  }
```

```
76  int Extract_Min(V v[])
77  {
78      int i = 0, j = 0;
79
80      for (i = 0; i < 8; i++)
81      {
82          if (Q.empty())                  // queue에서 빼낼 값이 없으면
83          {
84              for (j = 0; j < 8; j++)
85              {
86                  if (v[j].Extract != 1)      // 아직 Extract 되지 않은 vertex를
87                  {
88                      Q.push(v[j].Weight);        // queue 에 삽입
89                      printf("q.top:%d ", Q.top());
90                  }
91              }
92
93              j = 0;
94              for (j = 0; j < 8; j++)
95              {
96                  if (Q.top() == v[j].Weight && v[j].Extract != 1)    // 최소값이 root와 일치하면
97                                                                      // (top이 root)
98                      v[j].Extract = 1;               // Extract 하고
99
100                     while (!Q.empty())              // queue 를 모두 빼낸다.
101                         Q.pop();
```

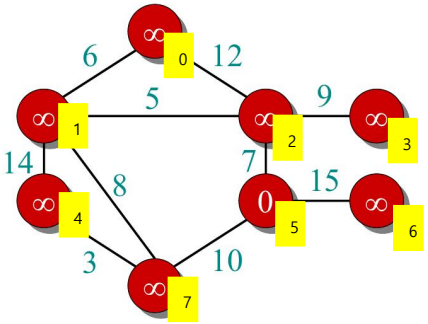
```
102
103                     return v[j].Vertex;             // Extract 된 vertex 반환
104                 }
105             }
106         }
107         printf("Q.top:%d ", Q.top());
108         if (Q.top() == v[i].Weight && v[i].Extract != 1)    // Extract 되지 않은 것중
109         {                                                   // 최소값이 root와 일치하면
110             v[i].Extract = 1;                   // Extract 하고
111             break;
112         }
113     }
114     while (!Q.empty())                  // queue 를 모두 빼낸다.
115         Q.pop();
116
117     printf("\n");
118
119     return v[i].Vertex;             // Extract 된 vertex 반환
120 }
121
```

```
122 int PRIM(V v[], int vtx)
123 {
124     int i = 0, j = 0;
125     int next_vtx = 0;
126
127     printf("Vertex: %d\n", v[vtx].Vertex);      // 어떤 vertex가 Extract 되었는지 출력
128
129     flag++;
130     if (flag == 8)              // vertex 수 만큼
131         return 0;
132
133     if (v[vtx].Weight == 0)     // 맨 처음 vertext 는 가중치가 0이다
134         v[vtx].Extract = 1;     // 그럼 Extract 1로
135
136     v[vtx].Extract == 1;        // 반환된 vertex, Extract 되었으므로 1로
137
138     Print_Table(v);
139
```

```
139
140     for (i = 0; i < 8; i++)
141     {
142         if (A[vtx][i] != 0 && v[i].Extract != 1)    // edge 값이 0이 아니고, 아직 Extract 되지않은 vertex
143         {
144             if (v[i].Weight > A[vtx][i])            // edge 값이 기존 가중치보다 작다면
145             {
146                 v[i].Weight = A[vtx][i];            // edge 값으로 upate
147                 v[i].Path = vtx;                    // 어디에 연결되었는지 저장
148                 Q.push(v[i].Weight);                // 최소값 선정을 위해 가중치 queue에 삽입
149             }
150         }
151     }
152     Print_Table(v);
153     next_vtx = Extract_Min(v);              // 최소값으로 선정된 vertex가 다음 vertex로 선택된다.
154     Print_Table(v);
155
156     PRIM(v, next_vtx);              // 선택된 vertex를 인수로 넣어 다시 Recursive 함수 호출
157
158     return 0;
159 }
160
```

(b) For at least one example, show the simulation results in step-by-step detail.

강의자료 예시



먼저 각 vertex의 index를 왼쪽 그림에서 노란색 네모 같이 설정하고 시작한다.

```
0 0 99999999 -1
1 0 99999999 -1
2 0 99999999 -1
3 0 99999999 -1
4 0 99999999 -1
5 0 99999999 -1
6 0 99999999 -1
7 0 99999999 -1
--------------------
```

Vertex 와 edge 정보들을 담고 있는 table은 Index, Extract, Weight, Path 순으로 출력된다.

왼쪽 그림을 보면 아직 Extract 된 vertex가 아무것도 없으므로, 모든 Extract를 0으로, Path를 -1로 초기화하고, 무한대 값을 표현하기위해 Weight에는 99999999를 입력했음을 확인할 수 있다.

```
Vertex: 5
0 0 99999999 -1
1 0 99999999 -1
2 0 99999999 -1
3 0 99999999 -1
4 0 99999999 -1
5 1 0 -1
6 0 99999999 -1
7 0 99999999 -1
--------------------
```

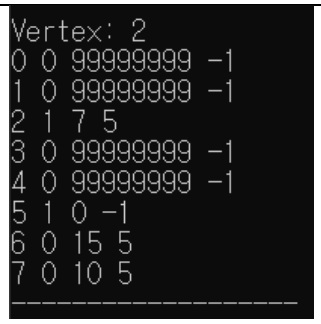
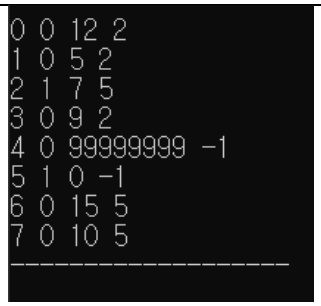
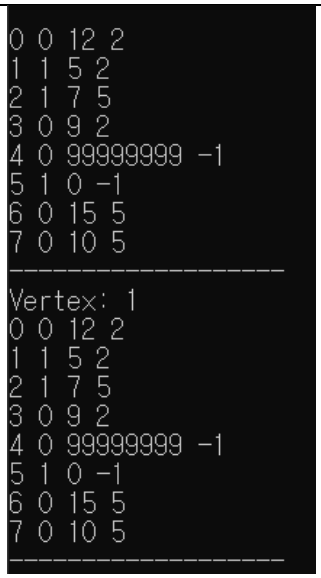
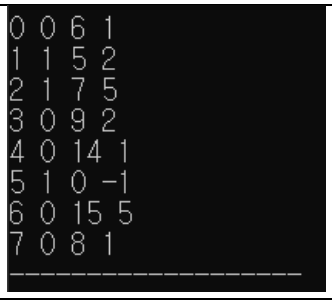
5번 vertex가 선택되었다. Weight를 0, Extract를 1로 update 한다.

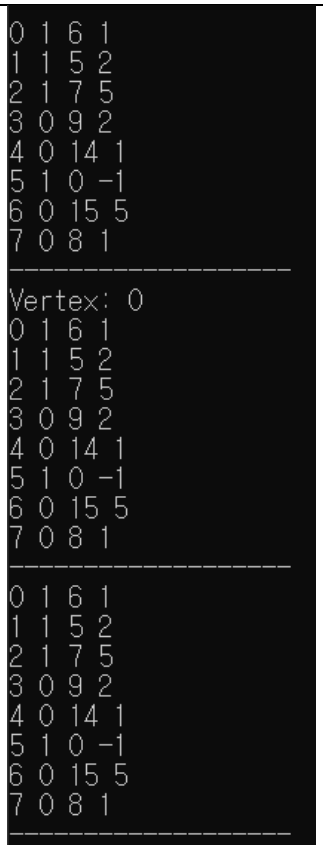
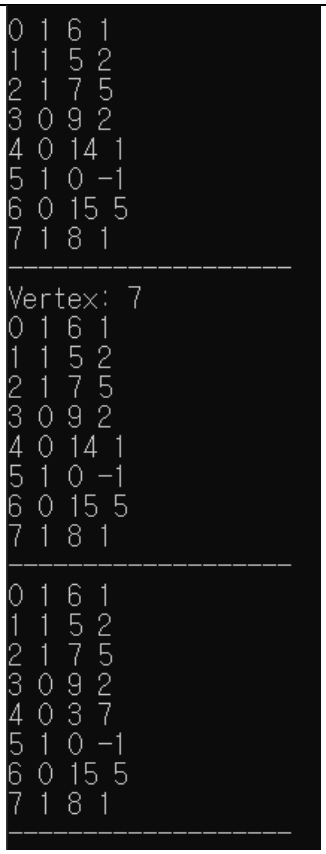
```
0 0 99999999 -1
1 0 99999999 -1
2 0 7 5
3 0 99999999 -1
4 0 99999999 -1
5 1 0 -1
6 0 15 5
7 0 10 5
--------------------
```

그 후 5번 vertex와 연결된 vertex인 2, 6, 7번 vertex의 key들을 그 edge들이 가지는 weight 만큼의 값으로 update한다. 각각의 vertex의 weight 들은 7, 15, 10으로 Path는 연결된 node의 index 인 5로 update 된 것을 확인할 수 있다.

```
0 0 99999999 -1
1 0 99999999 -1
2 1 7 5
3 0 99999999 -1
4 0 99999999 -1
5 1 0 -1
6 0 15 5
7 0 10 5
--------------------
```

7, 15, 10 중 가장 작은 값은 7이다 해당 weight를 가지는 vertext를 Extract 해야한다.

		7을 가지는 vertex인 2번 vertex가 다음 vertex로 선택되어 Extract 값이 1로 update되었다.
		그 후 2번 vertex와 연결된 0, 1, 3 vertex가 선택되어 key와 Path가 update 된 것을 확인할 수 있다. (5번 vertex는 이미 Extract 되었으니 고려하지 않는다.)
		마찬 가지로 2번 vertex 의 연결된 weight 중 가장 작은 값인 5를 가지는 vertex가 다음 vertex로 선정된다. 해당 vertex는 1번째 vertex 이고 Extract 된다.
		1번 vertex와 연결된 0, 7 번 vertex를 update한다.

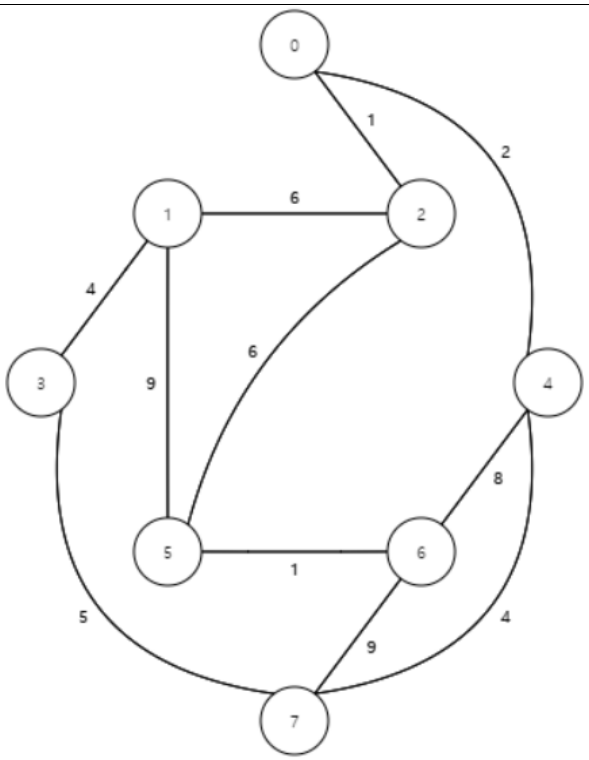
	최소값 6을 가지는 vertex인 0번째 vertex가 선정되어 Extract 된다. 그후 0번째 vertex에 연결된 edge는 더 이상 없는 것으로 확인할 수 있다. 2번째 vertex로 가는 edge가 있긴하지만 2번째 vertex는 이미 extract 되었고, 연결된 edge의 weight가 12로 매우 높다. 이후엔 0번째 vertex에 연결되지 않은 vertex 중 update 된 key 값이 가장 작은 값을 가지는 vertex가 다음 vertex로 선정된다.
	9, 14, 15, 8 중 최소값인 8을 가지는 7번째 vertex가 선정되어 Extract 된다. 그 후 이 vertex에 연결된 4번 vertex가 update 된다.

		4번 vertex를 extract한 후 상황에서 또 연결된 edge가 없는 상황이 발생했다. 아직 extract 되지 않은 vertex는 남아 있으니 그 key들을 비교해 최소값을 가지는 vertex를 선정한다.
		9와 15 중 최소값인 9를 가지는 3번째 vertex가 선정되고, Extract 되었다.

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

마지막으로 남은 6번 vertex 역시 Extrat 해주면서, Prim's algorithm은 마무리 된다.

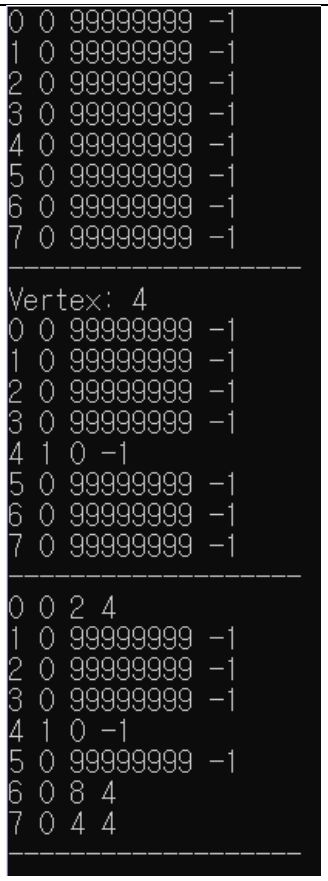
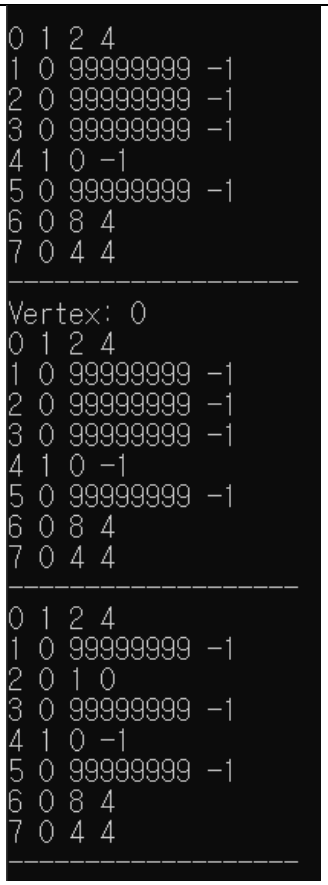
추가



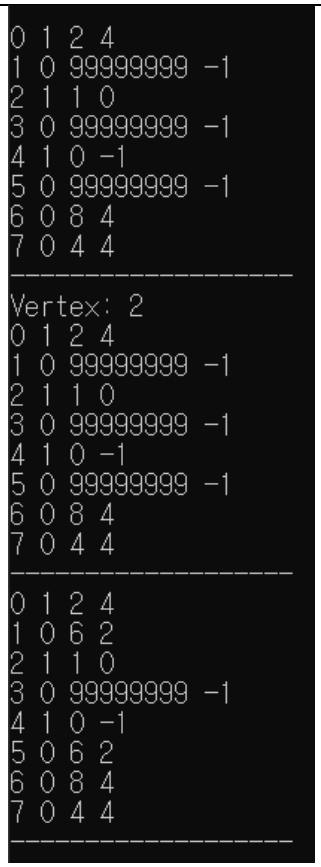
강의자료 예시는 Prim's algorithm의 기본적인 형태를 확인했다. Weight가 같은 가진 값을 가질 때, 어떤 순서로 적용되는지 확인하기 위해 더 복잡한 예시를 수행해 본다.

```
int A[8][8] = {             //
    {0, 0, 1, 0, 2, 0, 0, 0},
    {0, 0, 6, 4, 0, 9, 0, 0},
    {1, 6, 0, 0, 0, 6, 0, 0},
    {0, 4, 0, 0, 0, 0, 0, 5},
    {2, 0, 0, 0, 0, 0, 8, 4},
    {0, 9, 6, 0, 0, 0, 1, 0},
    {0, 0, 0, 0, 8, 1, 0, 9},
    {0, 0, 0, 5, 4, 0, 9, 0}
};
```

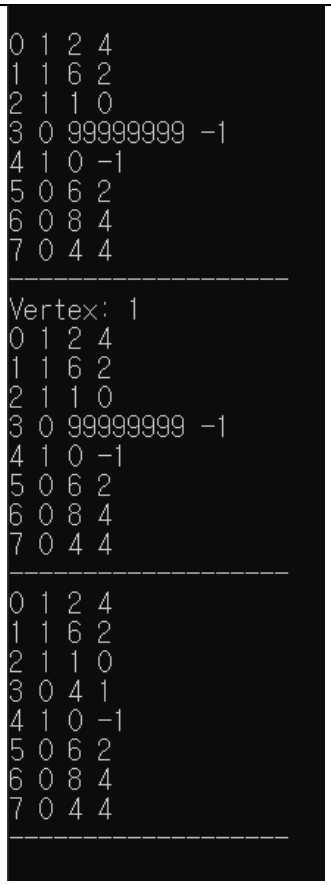
2차원 배열 A를 위 그림과 같이 update 한다.

	Random 으로 선택된 4번째 vertex를 update 한다. 첫 번째로 선택되었으니 Weight를 으로, Extract 를 1로 바꾼다. 그 후 4번 vertex와 연결된 0, 6 ,7번째 vertex도 update 해준다.
	연결된 edge 중 최소값 2를 가지는 vertex인 0번째 vertex가 선정되어 update 한다. Extract는 1이 된다.

<table>
<tr>
<td>



</td>
<td>0번째 vertex와 연결된 2번째 vertex를 update 한다. 그 후 연결된 edge는 1, 5번째 vertex인 것을 확인한다.</td>
</tr>
<tr>
<td>



</td>
<td>하지만, 1번째 vertex 와 5번째 vertex 모두 key 값이 6인 것을 알 수 있다. Key가 동일 하기 때문에 먼저 들어온 1번째 vertex를 선택한다.</td>
</tr>
</table>

	그 후 다시 1번째 vertex와 연결된 3번째 vertex를 update 한다. 3번째 vertex는 이미 1번째 vertex와 연결이 되어있어 아직 extract 되지 않은 vertex를 찾는다.
	아직 extract 되지않은 vertex를 찾아 weight를 비교한 후, 가장 최소값인 4를 가지는 7번째 vertex가 다음 vertex로 선정되었다. 역시 extract를 1로 바꾼다.

이번에도 마찬가지로 아직 Extract 되지 않은 vertex를 찾는 경우이다. 아직 extract 되지않은 vertex 중 최소값 6을 가지는 5번째 vertex가 선정되어 update 되었다.

그후 5번째 vertex 와 연결된 6번째 vertex를 선정해 update 한다. Extract를 1로 바꾸고, Path를 5로 바꾸는 것으로 마무리된다.

(c) Explain the program and the simulation results (at least five lines).

먼저 Prim's algorithm을 적용시키고 싶은 graph와 edge 정보를 2차원 배열 형태로 저장한다. 이 graph의 vertex들이 수시로 update 되는데, 이를 table 형태로 관리하기위해 별도의 class(V)를 선언했다. class V에는 몇 번째 vertex인지 나타내는 Vertex, Extract가 되었는지 확인할 수 있는 Extract(Extract가 되었으면 0, 아직 이루어지지않았으면 1), 각 edge의 key(가중치)들을 저장하는 Weight, 그리고 update 시 참조된 노드를 저장하기위한 Path로 이루어져있다.

먼저 이 class 변수들을 초기화한다. Vertex는 순차적으로 입력하고, 아직 하나도 Extract 되지않았으니 Extract는 0, Weight를 무한대를 나타내기 위해 99999999, Path는 -1로 초기화한다. 이 vertex들 중 임의의 vertex를 선택하기 위해 srand(time(NULL))을 사용 후, 이 값을 vtx 변수에 넣

어준다. (물론, 원하는 수를 직접 입력해도 된다.) 이렇게 첫번째로 선택된 vertex의 Weight는 0으로 바꾼다. 그 후 PRIM 함수를 호출한다.

Prim 함수가 호출될 때, 입력된 vertex는 Extract 된다. (해당 vertex의 Extract 값을 1로 바꿔준다.) 그 후 2차원 배열 A를 통해 해당 vertex에 연결된 edge가 있는지 확인하고 Extract가 이미 되었는지 확인한다. 연결된 edge가 있고, Extract되지 않은 vertex라면 vertex의 key를 확인하여 연결된 edge과 비교한다. 연결된 edge값이 더 작다면 해당 key를 edge 값으로 update하고, Path를 도 이 vertex로 update한다. 그 후 그 key를 queue에 push한다. 처음 vertex 부터 마지막 vertex 까지 이 작업을 거친다면, queue에 weight 값들이 push 되었을 테니(*없을 수도 있다.) 이중 최소값을 Extract 하는 Extract_Min 함수를 호출한다.

Extract_Min 함수에서는 priority queue에 쌓인 값들 중 최소값을 extract하는 함수이다. 최소값은 priority queue(min heap)의 root(top)에 위치하므로 이 값을 가지는 vertex를 extract한다.(Extract 값을 1로 바꾼다.) 그 후 queue를 모두 비우고 Extract 된 vertex 값을 반환한다.

*만약 Extract_Min 함수가 호출되었을 때 priority queue에 push 된 값이 하나도 없다면, Extract 되지 않은 것들 중 모든 weight 값을 Push 하여 확인하도록 한다. 이후엔 마찬가지로 priority queue의 root 를 Extract 하고, 이 값을 가지는 vertex 값을 반환한다.

이렇게 반환된 vertex는 Prim's algorithm을 사용할 다음 vertex로 선정된다. 이후 이 vertex를 인수로 넣어 다시 Recursive하게 함수를 호출한다.

실행 결과에 대한 설명은 편의를 위해 2. (b)에 그림과 같이 적어두었다.

Conclusion.

이번 과제는 Heap sort와 Prim's algorithm을 직접 구현하는 것이다. 먼저 Heap sort는 Max Heap을 구현했는데 C++ STL 을 사용하지 않고 모든 함수를 직접 구현했다. C++에서 배열은 0번째 index 부터 시작한다. 이 부분이 algorithm을 계산함에 있어 약간의 혼동이 있었다. 특히 parent node나 left child, right child 를 계산할 때 lower bound를 상황에 따라 적절히 넣어주는 것이 너무 복잡해져서 차라리 0번째 index에 아주 큰 수(99999999)를 넣어 1번째 index부터 차례로 계산할 수 있게 해줬다. 그 외 programming 하면서 생기는 문제점들은 디버깅을 통해 큰 어려움없이 구현했다.

Prim's algorithm은 priority queue STL을 사용해서 구현했다. 필자는 Queue STL 을 처음 사용해서 사용법부터 익히고, 시작해서 약간의 시간이 추가된 것 같다. Vertex에 대한 각종 정보를 저장하는 table을 처음에는 2차원 배열로 생성해 구현했지만, weight의 최소값을 선정해 해당 값을 가지는

vertex를 Extract 하는 부분에 있어 큰 문제점이 있었다. 그래서 이 부분을 여러 가지 변수를 한번에 담을 수 있는 class의 성질을 이용해야겠다는 생각이 들었고, class를 통해 구현해 이를 해결할 수 있었다. 또한, Extract 시 1로 수정이 되지 않거나, 1로 수정이 되었지만 weight 의 최소값을 선정하는 문제(queue에 삽입이 제대로 이루어지지 않음)들은 조건문을 상세하게 활용해서 이를 해결했다.

추가로 강의자료의 예시는 너무 순조로운 case 이기 때문에, 여러가지 자료구조에 대해 여러가지 case들을 시각화 해주는 web을 통해 더 많은 경우의 수를 파악하고 이를 구현하여 더욱 촘촘하게 완성할 수 있었던 것 같다.

Reference.

자료구조 시각화 및 table 활용 https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

C++ priority queue STL 사용법 https://twpower.github.io/93-how-to-use-priority_queue-in-cpp