

Data Structure

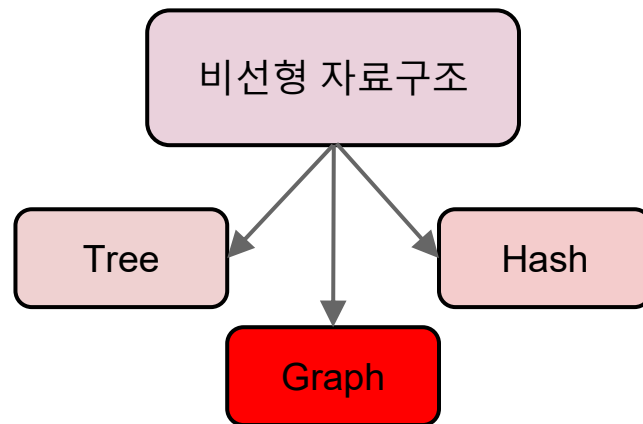
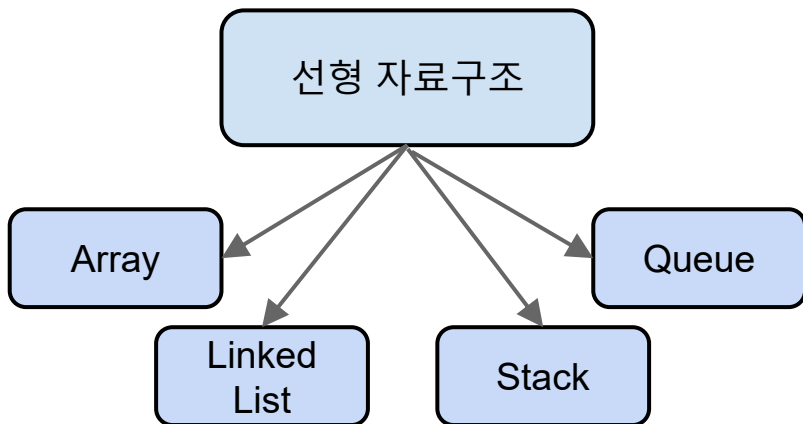
실습 10

0. 이번 주 실습 내용

- **Graph**
 - 그래프 정의, 표현법
 - DFS / BFS
- **DFS / BFS 실습**
 - DFS / BFS 구현 실습
- **Minimum cost Spanning Tree**
 - Prim algorithm

1. Graph

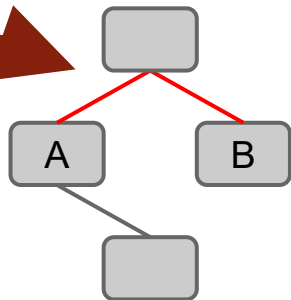
- Data Structure



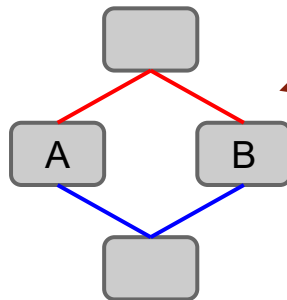
1. Graph

- **Graph 정의:** 하나 이상의 node들의 집합(V)과 두 node의 쌍으로 구성된 edge들의 집합(E)으로 이루어진 자료구조
 - A 노드에서 B 노드까지 경로가 유일하지 않을 수 있음
 - Tree는 Graph의 특수한 경우 ($\text{Tree} \subset \text{Graph}$)

트리(Tree)

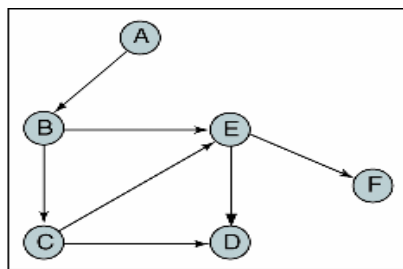


그래프(Graph)

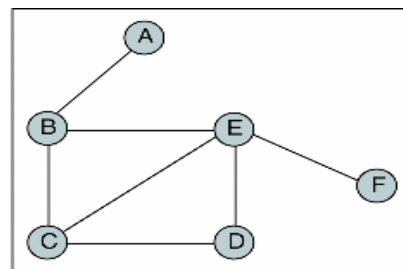


1. Graph

- **구성:** 정점 혹은 노드 (Vertex)와 간선 (Edge)로 이루어져 있고 간선은 두 노드를 연결시킨다.
 - Undirected Graph (무향 그래프)
 - 간선의 방향성이 없는 그래프로 두 정점 사이를 양 방향으로 연결한 것과 같은 의미
 - Directed Graph (유향 그래프)
 - 간선의 방향성이 있는 그래프로 시작 정점에서 끝 정점 쪽으로 한 쪽 방향으로 연결한 것과 같은 의미
 - Weighted Graph (가중치 그래프)
 - 각 간선들에 가중치가 부여되며 이는 정점 사이에 연결된 간선들의 길이가 다른 것과 같은 의미



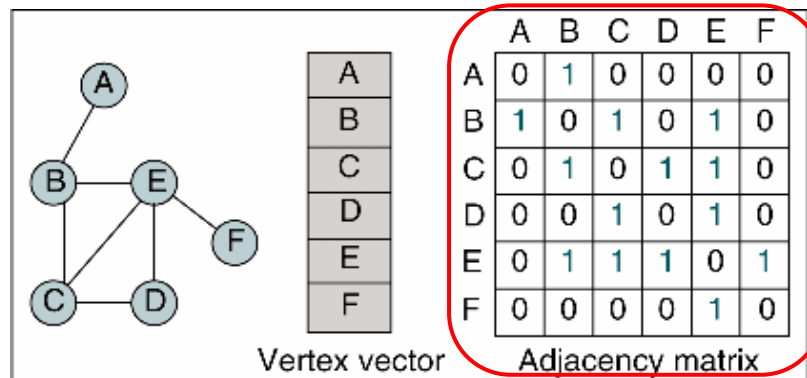
(a) Directed graph



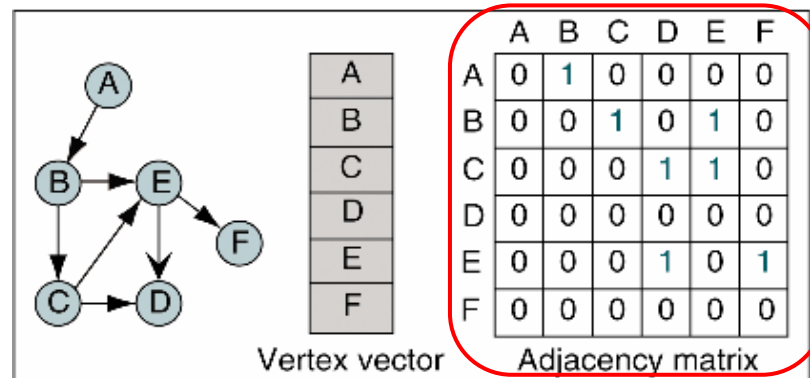
(b) Undirected graph

1. Graph

- **표현 방법 - Adjacency Matrix (인접 행렬)**
 - 2차원 행렬로 표현
 - node 1과 node 2가 연결되어 있다면 $\text{graph}[1][2] = 1$ 그렇지 않다면 $\text{graph}[1][2] = 0$
 - 만일 Weighted Graph라면 1 대신 가중치 값으로 표현



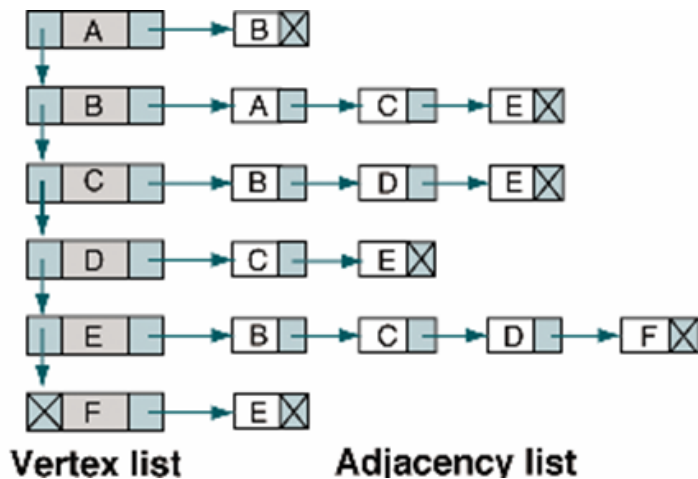
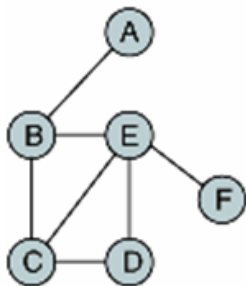
(a) Adjacency matrix for nondirected graph



(b) Adjacency matrix for directed graph

1. Graph

- 표현 방법 - Adjacency List (인접 리스트)
 - Linked-List 자료구조를 이용하여 Vertex에 연결된 Vertex들을 포인터를 이용해 연결



1. Graph

- **Graph Searching**

- **목적:** 시작 점에서 끝 점까지 연결된 경로를 찾거나 혹은 최단 경로(Weighted Graph)를 찾기 위해
 - Depth First Search(DFS) : 깊이 우선 탐색
 - Breath First Search(BFS) : 너비 우선 탐색
- ➔ Edge들의 가중치가 없을 때 혹은 다 같을 때 탐색하는 방법

1. Graph

• Depth First Search (DFS)

- 시작 Vertex와 연결된(&탐색 안 된) Vertex들 중 임의로 선택하여 탐색
- 선택된 Vertex를 기준으로 연결된(&탐색 안 된) Vertex들이 존재할 경우 위 과정을 반복
- 연결된 Vertex가 존재하지 않거나 이미 다 탐색한 Vertex인 경우 이전 Vertex로 돌아가서 탐색 수행

→ Stack의 작동 원리와 유사

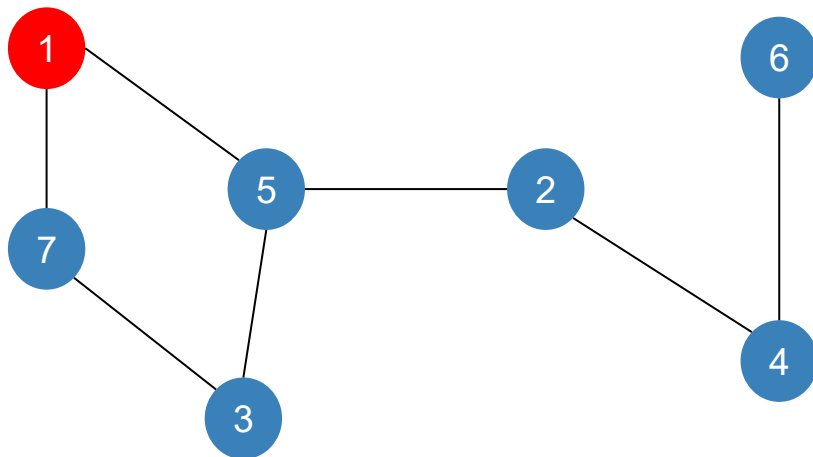
- Time Complexity
 $O(|V| + |E|)$: Vertex 개수 + Edge 개수
- Space Complexity
 $O(|V|)$: Vertex 개수

스택 기반

```
□ procedure DFS( $G, v$ ):  
  ■ label  $v$  as explored  
  ■ for all edges  $e$  in  $G.incidentEdges(v)$  do  
    □ if edge  $e$  is unexplored then  
      ■  $w \leftarrow G.opposite(v, e)$   
      ■ if vertex  $w$  is unexplored then  
        □ label  $e$  as a discovery edge  
        □ recursively call DFS( $G, w$ )  
      ■ else  
        □ label  $e$  as a back edge
```

1. Graph

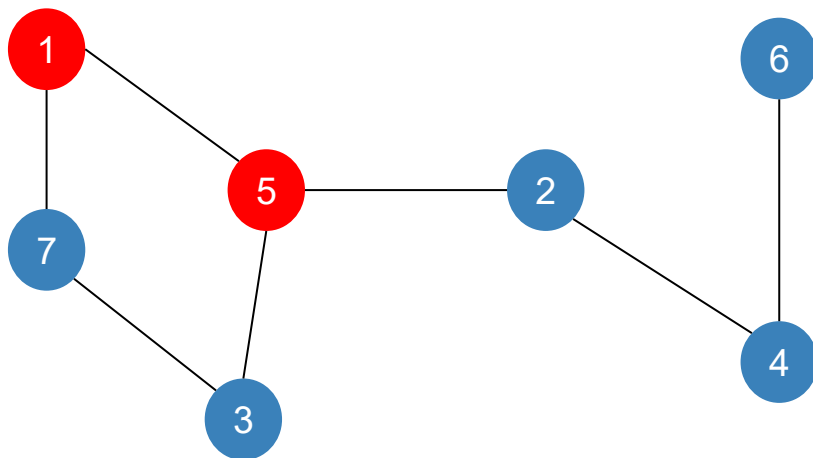
- Depth First Search (DFS)



Search: 1

1. Graph

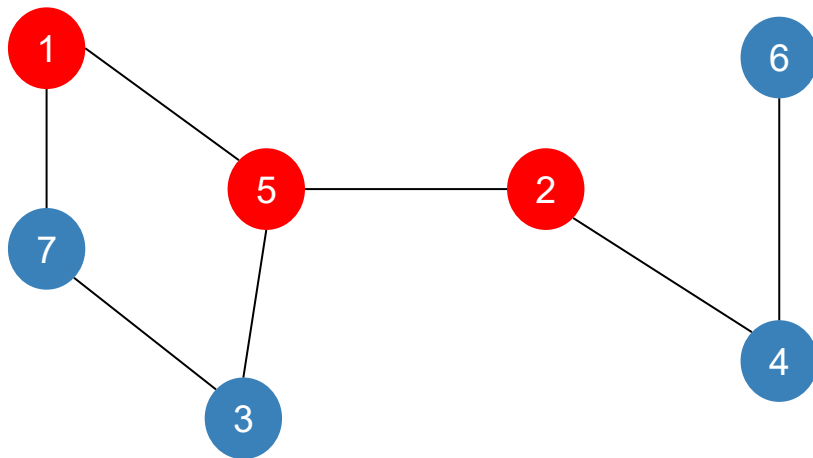
- Depth First Search (DFS)



Search: 1 - 5

1. Graph

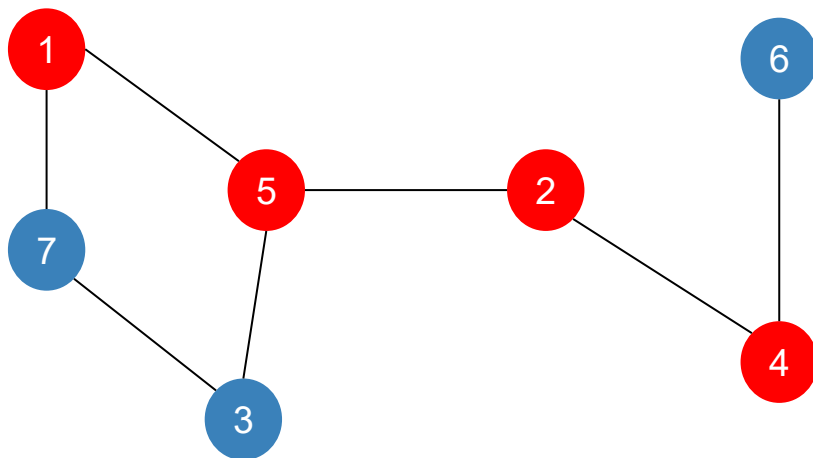
- Depth First Search (DFS)



Search: 1 - 5 - 2

1. Graph

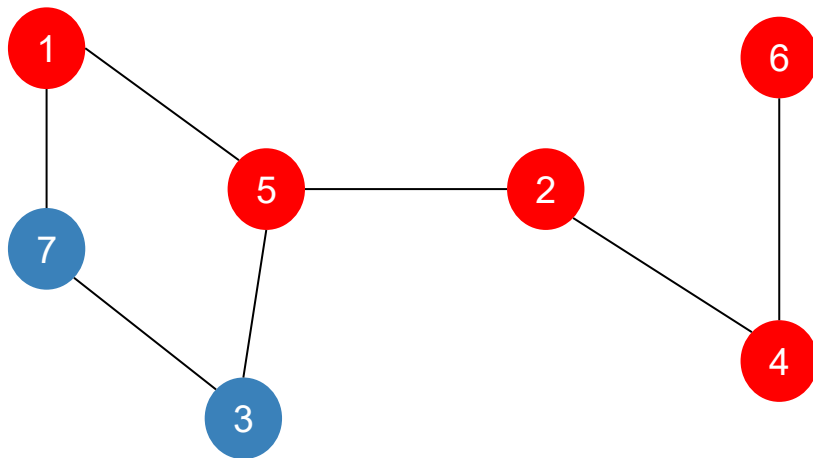
- Depth First Search (DFS)



Search: 1 - 5 - 2 - 4

1. Graph

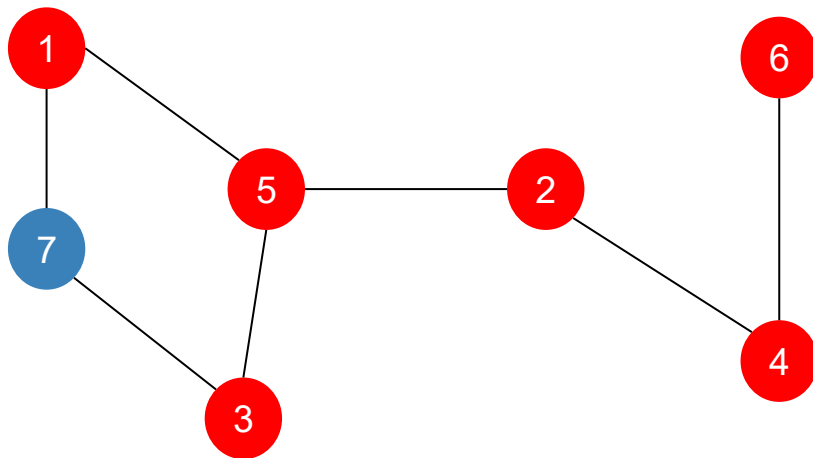
- Depth First Search (DFS)



Search: 1 - 5 - 2 - 4 - 6

1. Graph

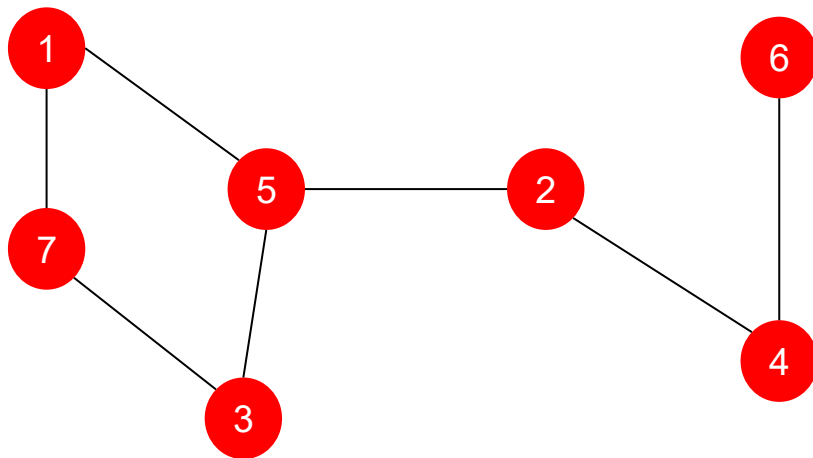
- Depth First Search (DFS)



Search: 1 - 5 - 2 - 4 - 6 - 3

1. Graph

- Depth First Search (DFS)



Search: 1 - 5 - 2 - 4 - 6 - 3 - 7

1. Graph

- **Breath First Search (BFS)**

- 시작 Vertex에 연결된 Vertex들을 각각 전부 탐색
- 탐색한 Vertex들에게 연결된 다른 Vertex들을 기록해 두었다가 순차적으로 탐색하면서 위 과정을 반복

→ 큐(Queue)의 작동 원리와 유사

- Time Complexity

$O(|V| + |E|)$: Vertex 개수 + Edge 개수

- Space Complexity

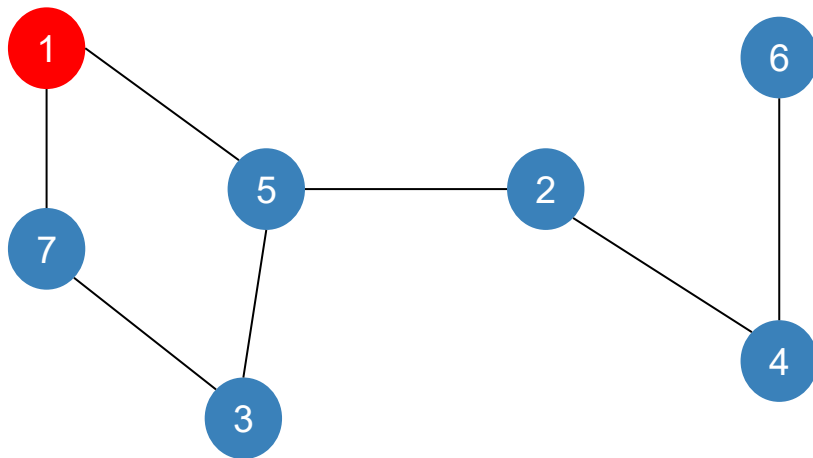
$O(|V|)$: Vertex 개수

실제 구현 시
큐를 사용

```
□ procedure BFS( $G, v$ ):  
  ■ create a queue  $Q$   
  ■ enqueue  $v$  onto  $Q$   
  ■ mark  $v$   
  ■ while  $Q$  is not empty:  
    □  $t \leftarrow Q.dequeue()$   
    □ if  $t$  is what we are looking for:  
      ■ return  $t$   
    □ for all edges  $e$  in  $G.incidentEdges(t)$  do  
      ■  $o \leftarrow G.opposite(t, e)$   
      ■ if  $o$  is not marked:  
        □ mark  $o$   
        □ enqueue  $o$  onto  $Q$ 
```

1. Graph

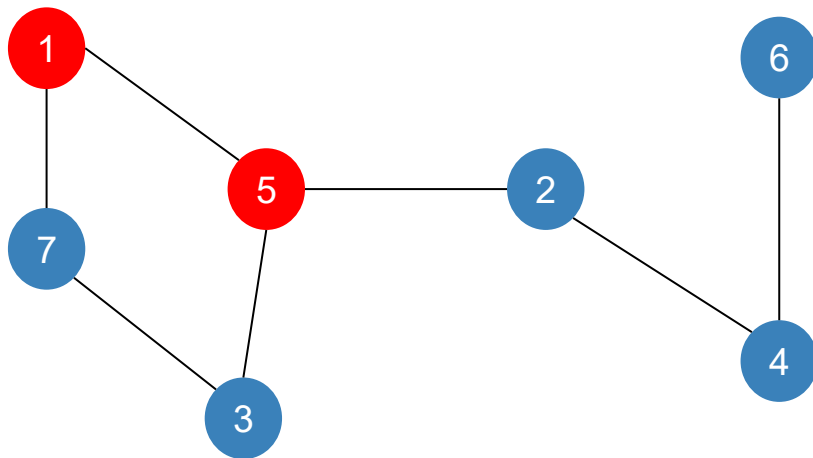
- Breath First Search (BFS)



Search: 1

1. Graph

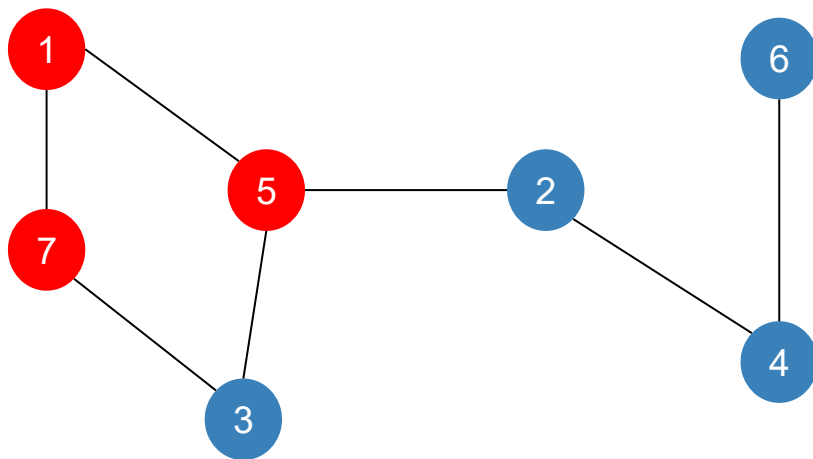
- Breath First Search (BFS)



Search: 1 - 5

1. Graph

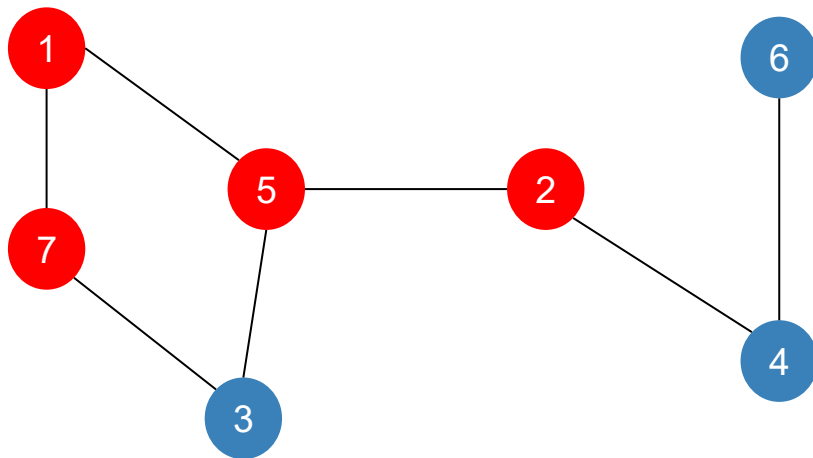
- Breath First Search (BFS)



Search: 1 - 5 - 7

1. Graph

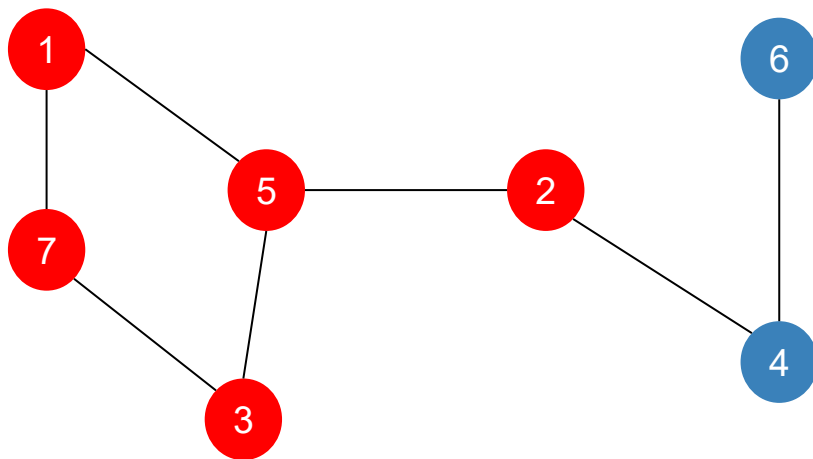
- Breath First Search (BFS)



Search: 1 - 5 - 7 - 2

1. Graph

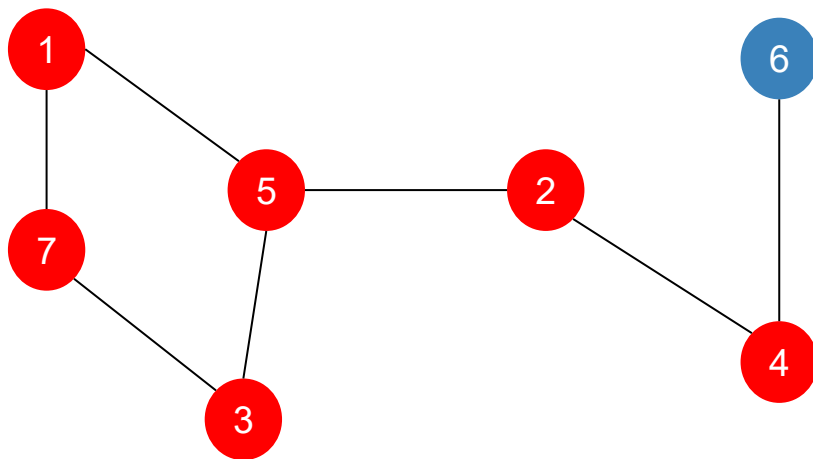
- Breath First Search (BFS)



Search: 1 - 5 - 7 - 2 - 3

1. Graph

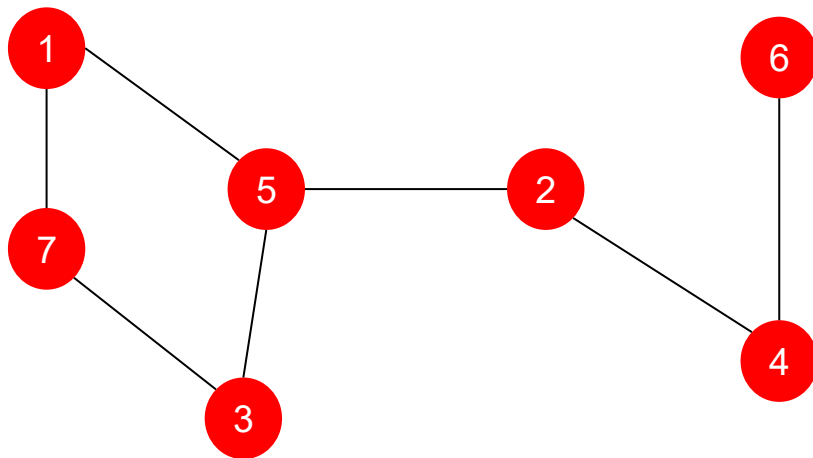
- Breath First Search (BFS)



Search: 1 - 5 - 7 - 2 - 3 - 4

1. Graph

- Breath First Search (BFS)



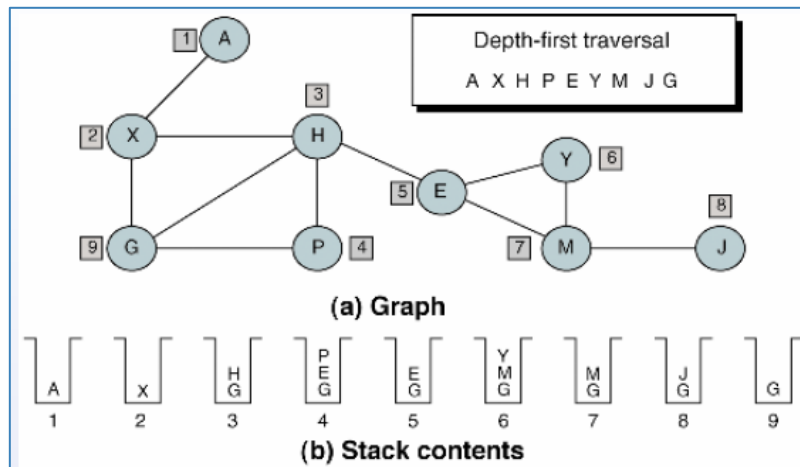
Search: 1 - 5 - 7 - 2 - 3 - 4 - 6

2. Graph 실습 (DFS)

• Depth First Search (DFS)

Stack 구조를 이용해 탐색할 Vertex들을 저장

1. 시작 Vertex를 기준으로 연결된(&탐색 안 된) Vertex들을 Stack에 저장
2. 더 이상 연결된(&탐색 안 된) Vertex들이 없을 경우 Stack에서 저장된 Vertex를 하나씩 꺼내서 1번 과정을 반복
3. Stack이 비워지면 탐색을 종료



2. Graph 실습 (DFS)

- Graph Structure

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_VERTICES 10
#define TRUE 1
#define FALSE 0

int visited[MAX_VERTICES];

typedef struct Graph {
    int adjMatrix[MAX_VERTICES][MAX_VERTICES];
    int n; // the number of vertex
}Graph;

void init(Graph *g) {
    int i, j;
    g->n = 0;
    for (i = 0; i < MAX_VERTICES; i++)
        for (j = 0; j < MAX_VERTICES; j++)
            g->adjMatrix[i][j] = 0;
}
```

- Graph Operation

```
void insertVertex(Graph *g, int v) {
    if (g->n == MAX_VERTICES)
    {
        printf("vertex 개수가 너무 많습니다. 노드 삽입 불가\n");
        return;
    }
    g->n++;
}

void insertEdge(Graph *g, int u, int v) {
    if (u >= g->n || v >= g->n) {
        printf("정점 번호가 잘못됨. 정점 번호는 0~N-1\n");
        return;
    }

    g->adjMatrix[u][v] = 1;
    g->adjMatrix[v][u] = 1;
}
```

2. Graph 실습 (DFS)

- Depth First Search (DFS)

- v는 현재 선택된 Vertex
- 선택된 Vertex로부터 다른 Vertex들이
연결된 Edge가 있는지
탐색하지 않았던 Vertex인지 확인
- 해당 Vertex에 대해 재귀적으로 DFS 수행



Stack 역할!

```
void dfs(Graph *g, int v) {  
    int w;  
    visited[v] = TRUE;  
    printf("%d ->", v);  
  
    [Redacted Code]  
}
```

2. Graph 실습 (DFS)

- Depth First Search (DFS)

- v는 현재 선택된 Vertex
- 선택된 Vertex로부터 다른 Vertex들이
연결된 Edge가 있는지
탐색하지 않았던 Vertex인지 확인
- 해당 Vertex에 대해 재귀적으로 DFS 수행



Stack 역할!

```
void dfs(Graph *g, int v) {  
    int w;  
    visited[v] = TRUE;  
    printf("%d ->", v);  
  
    for (w = 0; w < MAX_VERTICES; w++)  
    {  
        if (g->adjMatrix[v][w] == 1 && visited[w] == FALSE)  
            dfs(g, w);  
    }  
}
```

2. Graph 실습 (DFS)

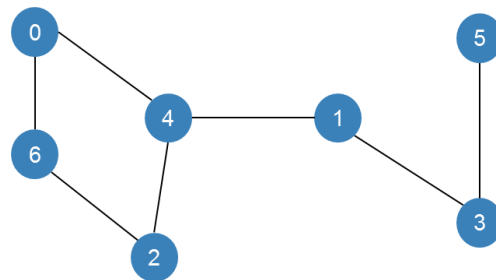
- main process

```
int main() {
    int i = 0;
    Graph g;

    init(&g);
    for (i = 0; i < MAX_VERTICES; i++) visited[i] = FALSE;
    for (i = 0; i < 7; i++) insertVertex(&g, i);
    insertEdge(&g, 0, 4);
    insertEdge(&g, 0, 6);
    insertEdge(&g, 1, 3);
    insertEdge(&g, 3, 5);
    insertEdge(&g, 4, 1);
    insertEdge(&g, 4, 2);
    insertEdge(&g, 6, 2);

    dfs(&g, 0);
}
```

- result



C:\WINDOWS\system32\cmd.exe

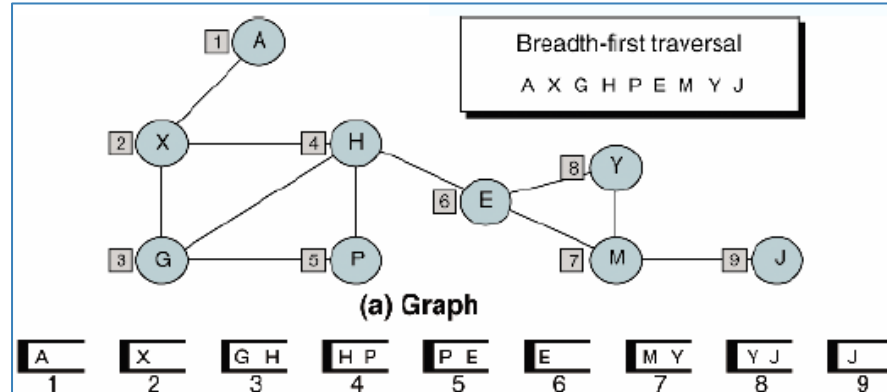
```
0 ->4 ->1 ->3 ->5 ->2 ->6 ->
계속하려면 아무 키나 누르십시오 . . .
```

2. Graph 실습 (BFS)

- Breath First Search (BFS)

Queue 구조를 이용해 탐색할 Vertex들을 저장

1. 시작 Vertex를 기준으로 연결된(&탐색 안 된) Vertex들을 Queue에 저장
2. 더 이상 연결된(&탐색 안 된) Vertex들이 없을 경우 Queue에서 저장된 Vertex를 하나씩 꺼내서 1번 과정을 반복
3. Queue가 비워지면 탐색을 종료



2. Graph 실습 (BFS)

- Graph & Queue Structure

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 1000
#define TRUE 1
#define FALSE 0
#define MAX_VERTICES 10

int visited[MAX_VERTICES];

typedef struct Queue {
    int queue[MAX_SIZE + 1];
    int rear;
    int front;
}Queue;

typedef struct Graph {
    int adjMatrix[MAX_VERTICES][MAX_VERTICES];
    int n;
}Graph;

void init(Graph *g) {
    int i, j;
    g->n = 0;
    for (i = 0; i < MAX_VERTICES; i++)
        for (j = 0; j < MAX_VERTICES; j++)
            g->adjMatrix[i][j] = 0;
}

void initQueue(Queue* q) {
    q->front = 0;
    q->rear = 0;
}
```

- Graph Operation

```
void insertVertex(Graph *g, int v) {
    if (g->n == MAX_VERTICES)
    {
        printf("vertex 개수가 너무 많습니다. 노드 삽입 불가\n");
        return;
    }
    g->n++;
}

void insertEdge(Graph *g, int u, int v) {
    if (u >= g->n || v >= g->n) {
        printf("정점 번호가 잘못됨. 정점 번호는 0~N-1\n");
        return;
    }

    g->adjMatrix[u][v] = 1;
    g->adjMatrix[v][u] = 1;
}
```

2. Graph 실습 (BFS)

- Queue Operation

```
int isFull(Queue* q) {  
    if ((q->rear + 1) % MAX_SIZE == q->front)  
        return 1;  
    else  
        return 0;  
}  
  
int isEmpty(Queue* q) {  
    if (q->front == q->rear)  
        return 1;  
    else  
        return 0;  
}
```

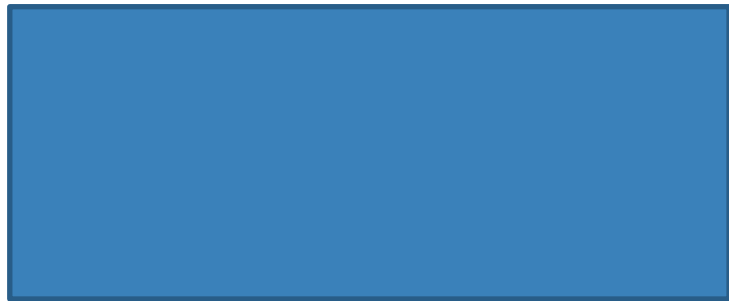
```
void enqueue(Queue* q, int data) {  
    if (isFull(q))  
        printf("큐가 가득 참\n");  
    else {  
        q->queue[q->rear] = data;  
        q->rear = (q->rear + 1) % MAX_SIZE;  
    }  
}  
  
int dequeue(Queue* q) {  
    int tmp = -1;  
    if (isEmpty(q))  
        printf("큐가 비어있음\n");  
    else {  
        tmp = q->queue[q->front];  
        q->front = (q->front + 1) % MAX_SIZE;  
    }  
    return tmp;  
}
```


2. Graph 실습 (BFS)

- v는 시작 Vertex
- BFS식 탐색을 위해 Vertex들을 저장할 Queue 자료구조 사용
- 시작 Vertex를 Queue에 저장
- Queue에서 Vertex를 하나씩 뽑아내면서 그 Vertex를 기준으로 다른 Vertex들이 연결된 Edge가 있는지 탐색하지 않았던 Vertex인지 확인
- 해당 Vertex들을 전부 Queue에 저장

• Breath First Search (BFS)

```
void bfs(Graph *g, int v) {  
    int w, search_v;  
    Queue q;  
    initQueue(&q);  
    visited[v] = TRUE;  
    enqueue(&q, v);  
}
```



2. Graph 실습 (BFS)

- v는 시작 Vertex
- BFS식 탐색을 위해 Vertex들을 저장할 Queue 자료구조 사용
- 시작 Vertex를 Queue에 저장
- Queue에서 Vertex를 하나씩 뽑아내면서 그 Vertex를 기준으로 다른 Vertex들이 연결된 Edge가 있는지 탐색하지 않았던 Vertex인지 확인
- 해당 Vertex들을 전부 Queue에 저장

• Breath First Search (BFS)

```
void bfs(Graph *g, int v) {  
    int w, search_v;  
    Queue q;  
    initQueue(&q);  
    visited[v] = TRUE;  
    enqueue(&q, v);  
  
    while (!isEmpty(&q)) {  
        search_v = dequeue(&q);  
        printf("%d->", search_v);  
  
        for (w = 0; w < MAX_VERTICES; w++)  
        {  
            if (g->adjMatrix[search_v][w] == 1 && visited[w] == FALSE){  
                visited[w] = TRUE;  
                enqueue(&q, w);  
            }  
        }  
    }  
}
```

2. Graph 실습 (BFS)

- main process

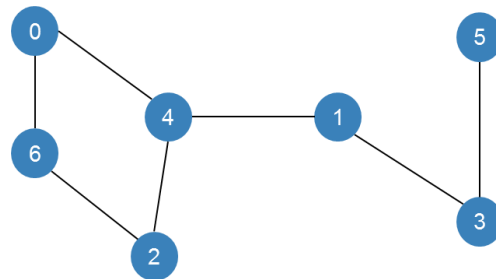
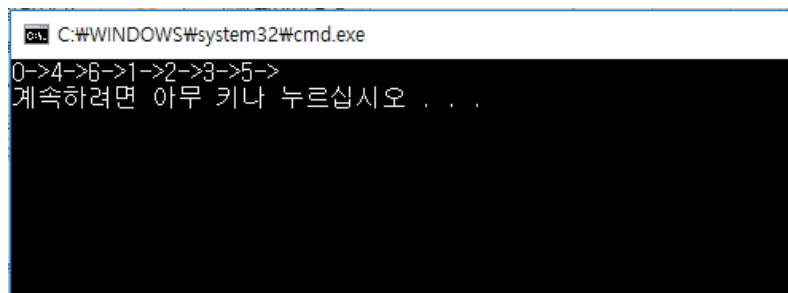
```
int main() {
    int i = 0;
    Graph g;

    init(&g);

    for (i = 0; i < MAX_VERTICES; i++) visited[i] = FALSE;
    for (i = 0; i < 7; i++) insertVertex(&g, i);
    insertEdge(&g, 0, 4);
    insertEdge(&g, 0, 6);
    insertEdge(&g, 1, 3);
    insertEdge(&g, 3, 5);
    insertEdge(&g, 4, 1);
    insertEdge(&g, 4, 2);
    insertEdge(&g, 6, 2);

    bfs(&g, 0);
}
```

- result

```

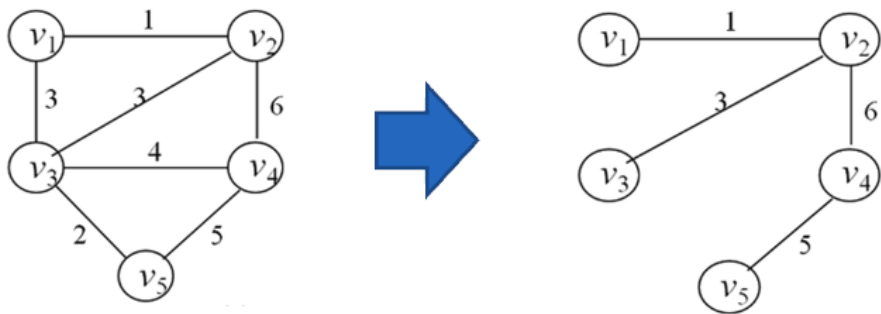
C:\WINDOWS\system32\cmd.exe
0->4->6->1->2->3->5->
계속하려면 아무 키나 누르십시오 . . .

```

3. Prim algorithm

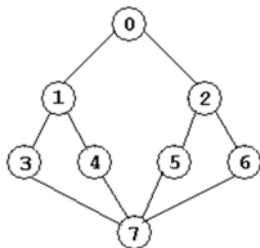
- **Spanning Tree**

- 그래프 내의 모든 Vertex들을 포함하는 트리
- Spanning Tree의 조건
 - 모든 Vertex들이 서로 연결되어 있어야 함
 - 사이클(Cycle)이 생겨서는 안됨

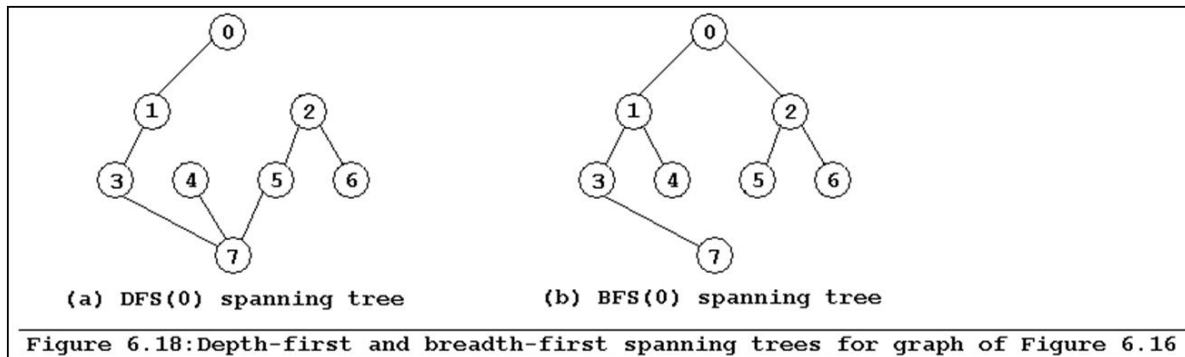


3. Prim algorithm

- Spanning Tree



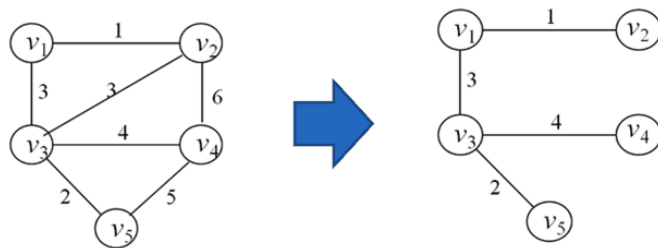
- DFS Spanning Tree / BFS Spanning Tree



3. Prim algorithm

- **Minimum cost Spanning Tree**

- Spanning Tree 중에서 Edge들의 가중치 합이 최소인 Tree (in Weighted Graph)
- Minimum cost Spanning Tree 조건
 - Edge들의 Weighted 값의 합이 최소
 - 반드시 $n-1$ 개의 edge만 사용
 - 사이클(Cycle)이 생겨서는 안됨
- Minimum cost Spanning Tree 생성 알고리즘
 - Prim algorithm (Using Vertex)
 - Kruskal algorithm (Using Edge)



3. Prim algorithm

• Prim algorithm

- 시작 Vertex에서 출발하여 Vertex Set을 단계적으로 확장해 나가면서 Spanning Tree를 구축하는 방법
- 현재 Vertex Set과 인접한 Vertex들 중 연결된 Edge의 Weight가 가장 작은 Vertex를 선택하여 Vertex Set에 추가
- 구현 방법: priority queue || array

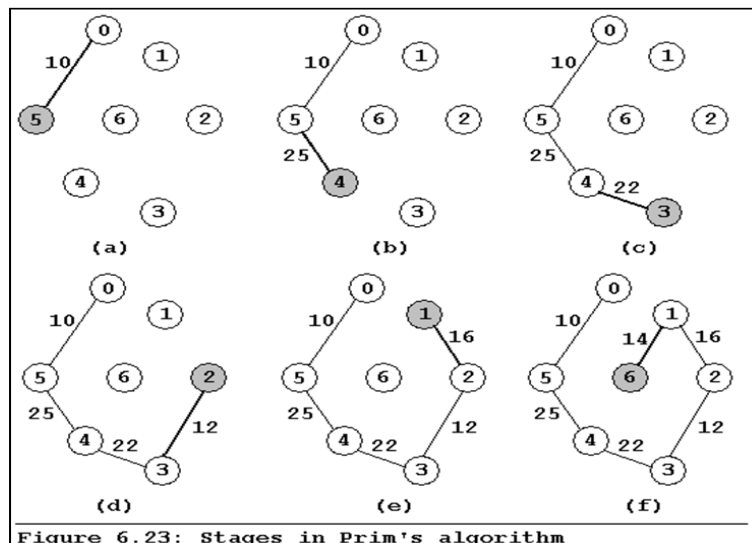
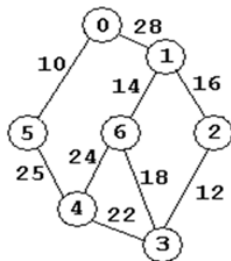


Figure 6.23: Stages in Prim's algorithm

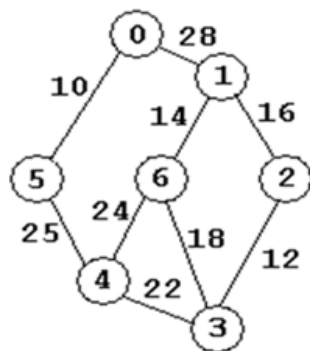
3. Prim algorithm

- Prim algorithm

1. 시작 Vertex 선택
2. 시작 Vertex의 거리는 0, 나머지는 INF로 초기화
3. 시작 Vertex에 연결된 Edge들의 Weight를 이용해 거리(distance[])를 갱신
4. distance[]값이 최소인 정점 u 를 선택
5. u 를 Vertex Set에 추가
6. Vertex Set에 포함되지 않은 모든 Vertex들에 대하여 u 와 인접한 Vertex들의 거리를 갱신
7. 4번으로 돌아가 반복

3. Prim algorithm

- Prim algorithm



```
#include <stdio.h>
#include <stdlib.h>

#define TRUE 1
#define FALSE 0
#define MAX 7
#define INF 1000

int graph[MAX][MAX] = {
    {0, 28, INF, INF, INF, 10, INF},
    {28, 0, 16, INF, INF, INF, 14},
    {INF, 16, 0, 12, INF, INF, INF},
    {INF, INF, 12, 0, 22, INF, 18},
    {INF, INF, INF, 22, 0, 25, 24},
    {10, INF, INF, INF, 25, 0, INF},
    {INF, 14, INF, 18, 24, INF, 0}
};

int selected[MAX]; //Vertex Set
int dist[MAX];     //distance[]
```

3. Prim algorithm

- `int getMinVertex(int n)`
 - Vertex Set의 Vertex들 중에서 연결된 Edge들 중 최소 Weight 값을 갖는 Vertex를 반환하는 함수
 - n은 총 Vertex 개수

```
int getMinVertex(int n) {
    int v, i;

    for (i = 0; i < n; i++)
    {
        if (!selected[i]) {
            v = i;
            break;
        }
    }
    for (i = 0; i < n; i++)
        if (!selected[i] && (dist[i] < dist[v]))
            v = i;
    return v;
}
```

- `void prim(int s, int n)`
 - prim algorithm을 수행하는 함수
 - s는 시작 Vertex, n은 총 Vertex 개수

```
void prim(int s, int n) {
    int i, u, v;

    for (u = 0; u < n; u++) dist[u] = INF;

    dist[s] = 0;
    for (i = 0; i < n; i++)
    {
        u = getMinVertex(n);
        selected[u] = TRUE;

        printf("%d->", u);
        for (v = 0; v < n; v++)
            if (graph[u][v] != INF)
                dist[v] = graph[u][v];
    }
}
```

3. Prim algorithm

• 실행 결과

```
int main() {
    prim(0, MAX);

    return 0;
}
```

C:\> 선택 C:\WINDOWS\system32\cmd.exe

0-> 5-> 4-> 3-> 2-> 1-> 6->

계속하려면 아무 키나 누르십시오 . . .

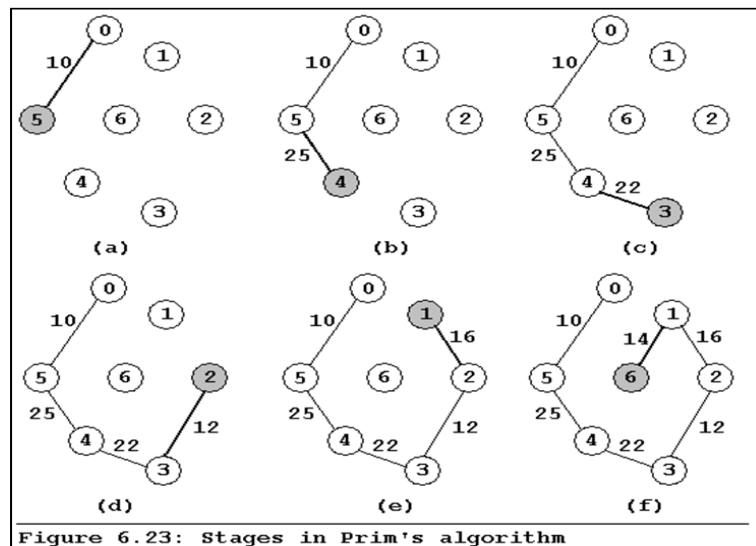
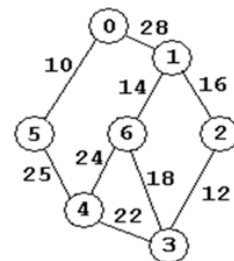


Figure 6.23: Stages in Prim's algorithm