

Algorithm Report

1Week. DFS / BFS (by Graph)

학번 : 201002513

이름 : 최 혁수

❑ 실습 1. Adjacency List(undirected Graph)

- 개요 & 알고리즘
- Adjacency List
 - Adjacency List Source Code
- Depth First Search(DFS)
 - DFS Source Code
 - 실행결과 분석 & 구현상의 오류 및 한계
- Breath First Search(BFS)
 - BFS Source Code
 - 실행결과 분석 & 구현상의 오류 및 한계

❑ 실습 2. Adjacency Matrix(undirected Graph)

- Adjacency Matrix
 - Adjacency Matrix Source Code
- Depth First Search(DFS)
 - DFS Source Code
 - 실행결과 분석 & 구현상의 오류 및 한계
- Breath First Search(BFS)
 - BFS Source Code
 - 실행결과 분석 & 구현상의 오류 및 한계

❑ 소감문 & 한계성

❑ All Source code

- Adjacency List
- Adjacency Matrix

실습 1. Adjacency List (undirected Graph)

○ 개요 & 알고리즘

○ Adjacency List

- Adjacency List Source Code

○ Depth First Search(DFS)

- DFS Source Code
- 실행결과 분석 & 구현상의 오류 및 한계

○ Breath First Search(BFS)

- BFS Source Code
- 실행결과 분석 & 구현상의 오류 및 한계

Adjacency List

□ 개요

링크드리스트(Linked List)를 이용하여 구현된 그래프(Graph)를 이해하고 탐색 알고리즘인 깊이우선탐색(DFS), 너비우선탐색(BFS) 구현을 통해 각 간선으로 연결된 정점들의 방문방법에 대해 인지하도록 한다.

□ 알고리즘

○ 그래프(Graph)

그래프는 각 정점(Vertex)과 이를 연결하는 간선(Edge)의 집합으로 무방향 그래프(Undirected Graph : $V1 \leftrightarrow V2$), 방향 그래프(Directed graph : $V1 \rightarrow V2$)로 나뉘며, 이는 링크드리스트(Linked List)의 노드(Node)값인 데이터와 포인터를 이용하여 ① 데이터=정점, ② 간선=포인터 의 상관관계에 따라 표현이 가능하다.

○ 깊이우선탐색(Depth First Search)

첫 시작 정점(V)을 기준으로 인접정점($V1$)을 방문하고, $V=V1$ 으로 인접정점($V1$)을 기준으로 다시 방문이 이루어지도록 재귀함수를 사용한다. 이때 정점(V)을 기준으로 인접정점이 $V1, V2, \dots, Vn$ 있을 때, 방문 정점($V1$)을 제외한 이외 정점은 스택에 값을 Push() 하고, $V1$ 에 대한 인접정점 탐색을 끝냈을 때, 스택 값을 Pop(); 하여 다시 방문이 이뤄지도록 한다.

○ 너비우선탐색(Breath First Search)

첫 시작 정점(V)을 기준으로 인접정점($V1, V2, \dots, Vn$)을 모두 방문하는데 이때의 $V1, V2, \dots, Vn$ 의 값을 Queue에 모두 Push()한 후 다시 하나씩 Pop()하면서 $V1, V2, \dots, Vn$ 에 대한 인접노드를 다시 Queue에 모두 Push() 한후 하나씩 Pop()하는 형식의 재귀적인 탐색방법으로 이루어진다.

Adjacency List Source Code(1/2)

□ DFSBFS_List.h

```
#define bool int
#define TRUE 1
#define FALSE 0

typedef struct node *pnode;
struct node {
    int vertex;
    struct node *link;
};
typedef struct queue *queue_pointer;
typedef struct queue {
    int vertex;
    queue_pointer link;
};
pnode* headNodes;
int numofVertex;           // 정점의 갯수를 저장
bool *visited;             // 방문된 정점을 표기
void createGraph(int);
void insertEdge(int, int);
void deleteVertex(int);
void printGraph();
void vertexInit();
void dfs(int);
void bfs(int);
// 큐값에 정점을 저장(push 기능)
void add_queue(queue_pointer *, queue_pointer *, int);
int delete_queue(queue_pointer *); // pop() 기능
```

□ void createGraph(int)

```
void createGraph(int numVertex){ // Graph 생성&초기화
    int i=0;
    // numofVertex(정점갯수) 초기화
    numofVertex = numVertex;
    headNodes = (pnode*)malloc(sizeof(pnode)*numVertex);

    while(i<numVertex){
        headNodes[i++] = NULL;
    }
    // 1차원 동적할당 * numofVertex
    visited = (int*)malloc(sizeof(int)*numofVertex);
    for(i=0; i<numofVertex; i++) visited[i] = 0;
}
```

□ void insertEdge(int, int)

```
void insertEdge(int v1, int v2){ // 정점 v1->v2 연결
    pnode new_node=NULL;
    pnode find=NULL;

    new_node = (pnode)malloc(sizeof(pnode)); // new_node 선언 및 vertex, link 정보 저장
    new_node->vertex = v2;
    new_node->link = NULL;

    find = headNodes[v1];
    if(find==NULL){ // 정점 v1에 연결된 정점이 없을때
        headNodes[v1] = new_node;
        return ;
    }
    while(find->link!=NULL){ // 마지막에 연결된 정점 찾기
        find = find->link;
    }
    find->link = new_node; // 마지막에 new_node 연결
}
```

Adjacency List Source Code(2/2)

❑ void deleteVertex(int)

```
void deleteVertex(int Vertex){           // Vertex = 삭제하고자 하는 정점
    pnode prev;
    pnode temp;
    int i=0;
    headNodes[Vertex] = NULL;           // Vertex로 부터 시작된 인접정점의 연결을 끊는다
    for(; i<numofVertex; i++){          // 이외의 headNodes 에 연결된 Vertex를 찾아 삭제
        prev = NULL;
        temp = NULL;
        temp=headNodes[i];
        while(temp!=NULL){
            if(temp->vertex == Vertex){
                // if) Vertex:0, 1->0->2 일경우, 1->2로 변환 또는 1->0 일 경우, 1->NULL
                if(temp==headNodes[i]) headNodes[i] = headNodes[i]->link;
                // if) Vertex:0 1->2->3->0->4 일경우, 1->2->3->4로 변환
                else prev->link = temp->link;
            }
            if(temp->link == NULL) break; // Vertex가 연결되어있지 않을때, break;
            prev = temp;
            temp = temp->link;           // 계속해서 인접노드로 연결
        }
    }
}
```

❑ void printGraph()

```
void printGraph()    // 그래프 출력
{
    pnode first;
    int i=0;
    for(i=0; i<numofVertex; i++)
    {
        if(headNodes[i]!=NULL){
            first = headNodes[i];
            printf("%d",i);
            while(first->link != NULL){
                printf(" -> %d", first->vertex);
                first = first->link;
            }
            printf(" -> %d\n", first->vertex);
        }
    }
}
```

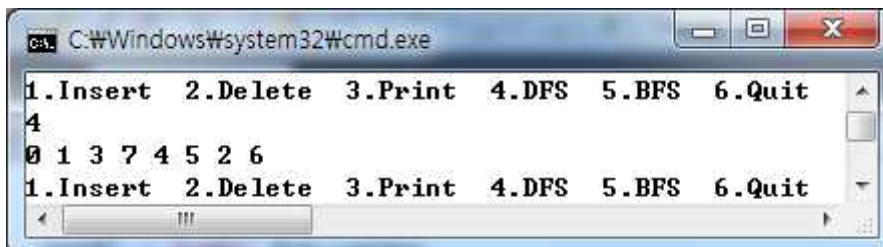
Depth First Search(DFS)

□ void dfs(int)

```
void dfs(int v){
    pnode w;
    visited[v]=TRUE; // 방문시 해당 정점 방문정보를 1로 변환
    printf("%d ",v);
    for(w=headNodes[v]; w; w=w->link){ // v:0, 0을 시작으로 인접노드 방문 실행
        if(!visited[w->vertex]){ // 방문하지 않은 정점이 있을경우, 재귀적 실행(w->vertex)
            dfs(w->vertex);
        }
    }
}
```

실행결과 분석 & 구현상의 오류 및 한계

□ 실행 결과화면



Input : (0 1), (0 2), (1 3), (1 4), (2 5), (2 6), (7 3), (7 4), (7 5), (7 6) 일 때,
Result-DFS : 0 1 3 7 4 5 2 6

□ 구현상의 오류 및 한계

위 DFS source code를 보면 DFS 기능을 1회 실행하면, visited[]=1 이 되어 다음 DFS 기능을 재실행하고자 할 때 DFS기능이 정상출력이 안되는 것을 볼수있었다.

= DFS function 2회 호출시, visited[]=1 이므로 방문하지 않은 정점(v)이 없어 출력이 안된다.
때문에 switch로 구현한 main()함수에 InitVertex();를 DFS function 이전에 실행하여 다시 visited[]=0;으로 초기화 하면서 보완하였다.

= switch()

...

case 4:

vertexInit(); // visited[] = 0; 으로 초기화하는 함수

dfs(0);

printf("Wn");

break;

Breath First Search(BFS) 1/2

❑ void add_queue(queue_pointer *, queue_pointer *, int)

/* queue를 동적할당 받아 link로 연결하면서 data값으로 BFS의 미방문 정점을 저장 */

void add_queue(queue_pointer *front, queue_pointer *rear, int node)

```
{
    queue_pointer ptr;

    ptr = (queue_pointer)malloc(sizeof(struct queue));
    ptr->vertex = node;
    ptr->link = NULL;
    if (!*front) // *front = NULL 일 때 실행
        *front = ptr; // front가 비어있기 때문에, front -> ptr로 저장
    else
        (*rear)->link = ptr;
    *rear = ptr; }
```

❑ int delete_queue(queue_pointer *)

/* FIFO에 의해 queue에 저장된 값을 하나씩 return & free */

int delete_queue(queue_pointer *front)

```
{
    queue_pointer ptr = *front;
    int vertex = ptr -> vertex;
    *front = ptr->link;
    free(ptr); // free(ptr) 메모리 해제
    return vertex; // vertex 값을 리턴
}
```

❑ void bfs(int v)

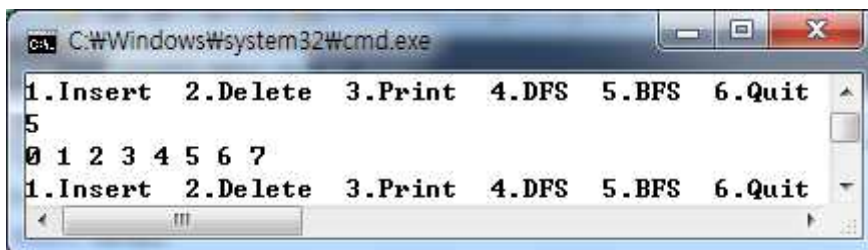
void bfs(int v) /* DFS 구현 */

```
{
    pnode w;
    queue_pointer front,rear;
    front = rear = NULL;
    visited[v] = TRUE;
    printf("%d ",v);
    add_queue(&front, &rear, v); // 방문정점을 모두 queue에 저장
    while (front) { // 미방문 정점이 없을때까지 실행
        v = delete_queue(&front); // FIFO에 의한 값을 v에 저장
        for (w = headNodes[v]; w; w=w->link)
            if (!visited[w->vertex]) { // 방문하지 않은 정점일때
                visited[w->vertex] = TRUE;
                printf("%d ", w->vertex);
                add_queue(&front,&rear,w->vertex); // 방문정점을 queue에 저장
            }
    }
}
```


Breath First Search(BFS) 2/2

실행결과 분석 & 구현상의 오류 및 한계

□ 실행 결과화면



Input : (0 1), (0 2), (1 3), (1 4), (2 5), (2 6), (7 3), (7 4), (7 5), (7 6) 일 때,
Result-BFS : 0 1 2 3 4 5 6 7

□ 구현상의 오류 및 한계

1. BFS function 또한 visited[]=1로 정점방문여부 표현을 하면서 인접노드를 방문하기 때문에 앞서 제기한 오류를 InitVertex()를 BFS function 이전에 실행하여 보완하였다.
2. BFS function을 구현하기 위해선 방문중인 정점(v)에 대한 인접노드(v1, v2, ..., vn)을 모두 queue에 저장후 순차적(FIFO)으로 빼내면서 인접노드의 인접노드를 다시 방문하는 재귀적인 구현이 필요한데 이를 본인은 queue를 사용하지 않고 단일 1차원 배열인 preVertex[]를 만들고 queue와 같이 작동하도록 만들려고 하였지만 FIFO의 구현을 하기에 코드가 queue에 비해 너무 길어질 뿐만 아니라 구현하기 위한 변수선언이 많아져 메모리 할당량이 많아짐을 느끼고 queue로 다시 돌아왔다. 또한 linked list로 구현하고자 하면 만든게 queue를 이용한 방법과 다르게 없었기 때문에 이론에서 배운 queue를 그대로 사용하고자 하였다.

실습 2. Adjacency Matrix (undirected Graph)

○ Adjacency Matrix

- Adjacency Matrix Source Code

○ Depth First Search(DFS)

- DFS Source Code
- 실행결과 분석 & 구현상의 오류 및 한계

○ Breath First Search(BFS)

- BFS Source Code
- 실행결과 분석 & 구현상의 오류 및 한계

Adjacency Matrix Source Code

* void createGraph(int) = Adjacency List Source Code 와 동일

□ DFSBFS_Array.h

```
#define bool int
#define TRUE 1
#define FALSE 0
/* BFS 구현을 위한 queue */
typedef struct queue *queue_pointer;
typedef struct queue {
    int vertex;
    queue_pointer link;
};

int numofVertex;
int **Vertex_Array;    // 2차원 배열
bool *visited;         // bool 자료형 사용
int delete_queue(queue_pointer *);
void add_queue(queue_pointer *, queue_pointer *, int);
void dfs(int);
void bfs(int);
void createGraph(int);
void insertEdge(int, int);
void deleteVertex(int);
void printGraph();
void vertexInit();
```

□ void printGraph()

```
/* Vertex_Array[][] 에 저장된 인접정점 출력 */
void printGraph()
{
    int i=0, j=0;
    printf("    ");
    for(i=0; i<numofVertex; i++){
        printf("[%d] ",i);
        printf("\n");
        for(i=0; i<numofVertex; i++){
            {
                printf("[%d] ",i);
                for(j=0; j<numofVertex; j++){
                    printf("%d ",Vertex_Array[i][j]);
                }
                printf("\n");
            }
        }
    }
}
```

□ void insertEdge(int, int)

```
/* Vertex_Array[][] 값에 1(Edge==Connet) 저장 */
void insertEdge(int v1, int v2){
    Vertex_Array[v1][v2] = 1;
}
```

□ void deleteVertex(int)

```
void deleteVertex(int Vertex){
    int i=0, j=0;
    for(i=0; i<numofVertex; i++){
        // vertex로부터 연결된 모든 정점 삭제
        Vertex_Array[Vertex][i]=0;
        // 타정점에서 vertex를 인접정점으로 가지고 있는 값을 삭제
        Vertex_Array[i][Vertex]=0;
    }
}
```

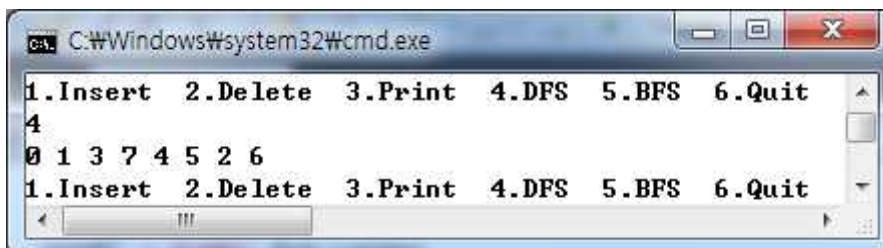
Depth First Search(DFS)

□ void dfs(int)

```
void dfs(int v)
{
    int i;
    printf("%d ", v);
    visited[v] = 1;    // 정점 v를 방문했다고 표시
    for (i = 0; i<numofVertex; i++)
    {
        // 정점 v와 정점 i가 연결되었고, 정점 i를 방문하지 않았다면
        if (Vertex_Array[v][i] == 1 && !visited[i])
        {
            // 정점 i에서 다시 DFS를 시작한다
            dfs(i);
        }
    }
}
```

실행결과 분석 & 구현상의 오류 및 한계

□ 실행 결과화면



Input : (0 1), (0 2), (1 3), (1 4), (2 5), (2 6), (7 3), (7 4), (7 5), (7 6) 일 때,
Result-DFS : 0 1 3 7 4 5 2 6

* Adjacency List_ Result-DFS 와 결과값이 같다

□ 구현상의 오류 및 한계

List & Array 모두 실습자료(PDF)에서 제시한대로 CreateGraph(8); 로 numofVertex=8; 즉, 정점의 개수를 8로 정의하고 있다. 따라서 vertex의 값의 증가를 하려면 main 함수 초기에 정점의 개수에 대한 정보를 사용자로부터 scanf()로 받아야 한다. 하지만 본인이 작성한 코드에는 그 코드가 제외되었고 기존에 제시한 개수 8을 유지하고 있으므로 더 큰 정점의 개수에 대한 프로그램은 돌아가지 않는다.

또한 중간에 정점의 개수를 바꾸기 위해선, CreateGraph()에서 **Vertex_Array를 사용자가 재정의한 정점의 개수로 realloc 해야 한다. 그러기 위해선 재정의를 위한 ReCreateGraph0; 라는 함수를 하나 더 선언하여 realloc 하는 방법으로 보완될 것이다.

Breath First Search(BFS)

/* Adjacency List Source Code 와 동일 */

1) int delete_queue(queue_pointer *);

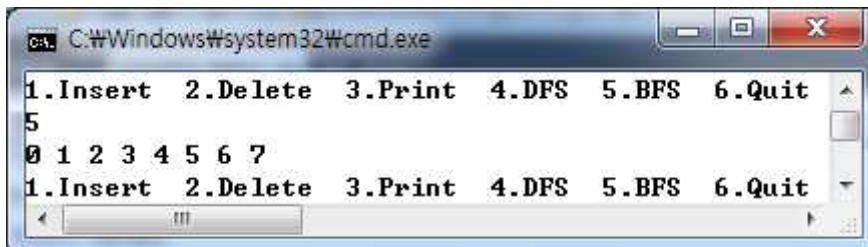
2) void add_queue(queue_pointer *, queue_pointer *, int);

□ void bfs(int) /* Adjacency List의 BFS와 작동원리는 같지만 조건식이 다르다 */

```
void bfs(int v){
    int i;
    queue_pointer rear, front;
    rear = front = NULL;
    visited[v] = 1;
    printf("%d ", v);
    add_queue(&front, &rear, v);
    while(front){
        v = delete_queue(&front);
        for(i=0; i<numofVertex; i++){ // I를 정점의 번호로 인식
            // 방문중인 정점(v)과 연결된 인접정점이 있고, 인접정점이 방문되지 않았을 때 if문 실행
            if(Vertex_Array[v][i] == 1 && !visited[i]){
                visited[i] = TRUE; // 방문표시 = 1
                printf("%d ", i); // 방문중인 정점 출력
                // 방문중인 I의 정점을 queue에 저장하고, 꺼낼때 정점 I에 대한 인접정점을 찾는다
                add_queue(&front, &rear, i);
            }
        }
    }
}
```

실행결과 분석 & 구현상의 오류 및 한계

□ 실행 결과화면



Input : (0 1), (0 2), (1 3), (1 4), (2 5), (2 6), (7 3), (7 4), (7 5), (7 6) 일 때,

Result-BFS : 0 1 2 3 4 5 6 7

* Adjacency List_ Result-BFS 와 결과값이 같다

□ 구현상의 오류 및 한계

Adjacency List로 구현한 BFS와 같은 오류 및 한계를 지닌다.

소감문 & 한계성

소감문 & 한계성

□ 소감문

눈과 글로 배운 이론적인 알고리즘을 실질적으로 프로그램상에 구현하는 일은 쉽지 않았습니다. 노드는 물론이거니와 이전에 배운 Stack, Queue를 DFS, BFS에 접목시켜 구현하는 일조차 이론은 장황하게 설명되는데 한줄되는 소스조차 잡기 힘들었기에 결론적으로 다가온 결과물은 값진 성과물이었습니다.

실질적인 단계를 밟지않고 눈으로 글로만 배운 사람은 손과 발을 잃을수 있다는 큰 경험이었고, 이 보고서를 작성하면서도 상당한 산물을 얻을수 있었는데 List와 Array의 비교를 시작해서 DFS/BFS의 개념과 코딩적인면의 정립이었습니다. 하물며 개인이 만든 코드를 누군가에게 설명한다는 그 자체적인 면에도 노력이 필요하다는 것까지, 이 실습 이외에 timing을 하여 프로그램의 시간적인 면도 검토해 보겠습니다.

□ 한계성

위 제시한 코드는 다양한 한계성을 지닌 불완전 프로그램입니다.

List & Array Size를 정의하고 시작한다는 문제점, 불필요한 데이터선언 및 코드 구현으로 불필요한 컴파일 시간의 증가, Adjacency List 구현시 Undirected Graph를 단일리스트 구현으로 InsertEdge(int, int)를 switch문에서 2번 이루어져 비정상 값이 들어가는 행위(ex. 0->1->1->3....의 1이 2번 중복될수 있습니다.) 등의 한계성등을 다양하게 지니고 있는 코드입니다.

이를 해결하기 위해 위에서 제시한 것 외 다양한 한계성에 대해 하나씩 고쳐나가야 하며 결론적인 한계성은 소스를 만든 본인에게 있다고 생각하며

알고리즘상의 한계성을 제시한다면, 정점을 방문하는 과정에서 방문한 정점을 다시 들어가서 확인하는 반복된 일련의 과정을 해야 간선으로 연결된 모든 정점을 방문할수 있다는 것에 있습니다. 이는 규칙적이지 않은 x번의 방문한 노드의 재방문이 이루어지고, 이는 프로그램상의 timing에도 영향이 있을것으로 생각됩니다.

All Source Code

- Adjacency List
- Adjacency Matrix

Adjacency List(1/5)

```
#include <stdio.h>
#include <stdlib.h>
#define bool int // bool 자료형 재정의(c에서 boolean 자료형은 c99부터 제공)
#define TRUE 1
#define FALSE 0
typedef struct node *pnode;
struct node {
    int vertex;           // 인접한 vertex 값을 저장
    struct node *link;    // 인접한 vertex 를 가리키는 포인터
};
typedef struct queue *queue_pointer;
typedef struct queue {
    int vertex;           // 인접한 vertex 값을 저장
    queue_pointer link;   // 인접한 queue 를 가리키는 포인터
};
pnode* headNodes;
int numofVertex;         // 정점의 갯수를 저장
bool *visited;           // 방문된 정점을 표기
void createGraph(int);    // Graph를 생성
void insertEdge(int, int); // 정점간에 간선을 연결해준다(=노드를 연결)
void deleteVertex(int);   // 정점을 삭제
void printGraph();        // 그래프 출력
void vertexInit();        // 방문된 정점표기 초기화

void dfs(int);            // dfs 구현
void bfs(int);            // bfs 구현
void add_queue(queue_pointer *, queue_pointer *, int); // 큐값에 정점을 저장(push 기능)
int delete_queue(queue_pointer *); // pop() 기능

/* headNodes & visited 동적할당 및 초기화 */
void createGraph(int numVertex){
    int i=0;
    numofVertex = numVertex;
    headNodes = (pnode*)malloc(sizeof(pnode)*numVertex);
    while(i<numVertex){
        headNodes[i++]=NULL;
    }
    visited = (int*)malloc(sizeof(int)*numofVertex);
    for(i=0; i<numofVertex; i++) visited[i] = 0;
}
/* visited 초기화 */
void vertexInit(){
    int i;
    for(i=0; i<numofVertex; i++) visited[i] = 0;
}
```

Adjacency List(2/5)

/* v1, v2 정점을 v1->v2로 연결 */

```
void insertEdge(int v1, int v2){
    pnode new_node=NULL;
    pnode find=NULL;
    new_node = (pnode)malloc(sizeof(pnode)); // node 선언 및 vertex, link 정보 저장
    new_node->vertex = v2;
    new_node->link = NULL;

    find = headNodes[v1];
    if(find==NULL){ // 정점 v1에 연결된 정점이 없을때
        headNodes[v1] = new_node;
        return ;
    }
    while(find->link!=NULL){ // 마지막에 연결된 정점 찾기
        find = find->link;
    }
    find->link = new_node;
}
```

/* 정점 삭제 */

```
void deleteVertex(int Vertex){ // Vertex = 삭제하고자 하는 정점
    pnode prev;
    pnode temp;
    int i=0;
    headNodes[Vertex] = NULL; // Vertex로 부터 시작된 인접정점의 연결을 끊는다
    for(; i<numofVertex; i++){ // 이외의 headNodes 에 연결된 Vertex를 찾아 삭제
        prev = NULL;
        temp = NULL;
        temp=headNodes[i];
        while(temp!=NULL){
            if(temp->vertex == Vertex){
                // if) Vertex:0, 1->0->2 일경우, 1->2로 변환 또는 1->0 일 경우, 1->NULL
                if(temp==headNodes[i]) headNodes[i] = headNodes[i]->link;
                // if) Vertex:0 1->2->3->0->4 일경우, 1->2->3->4로 변환
                else prev->link = temp->link;
            }
            if(temp->link == NULL) break; // Vertex가 연결되어있지 않을때, break;
            prev = temp;
            temp = temp->link; // 계속해서 인접노드로 연결
        }
    }
}
```

Adjacency List(3/5)

/* 인접노드를 방문하면서 출력 */

```
void printGraph()
{
    pnode first;
    int i=0;
    for(i=0; i<numofVertex; i++)
    {
        if(headNodes[i]!=NULL){
            first = headNodes[i];
            printf("%d",i);
            while(first->link != NULL){
                printf(" -> %d", first->vertex);
                first = first->link;
            }
            printf(" -> %d\n", first->vertex);
        }
    }
}
```

/* DFS 구현 */

```
void dfs(int v){
    pnode w;
    visited[v]=TRUE; // 방문시 해당 정점값을 1로 변환
    printf("%d ",v);
    for(w=headNodes[v]; w; w=w->link){ // v:0, 0을 시작으로 인접노드 방문 실행
        if(!visited[w->vertex]){ // 방문하지 않은 정점이 있을경우, 재귀적 실행(w->vertex)
            dfs(w->vertex);
        }
    }
}
```

/* queue를 동적할당 받아 link로 연결하면서 data값으로 BFS의 미방문 정점을 저장 */

```
void add_queue(queue_pointer *front, queue_pointer *rear, int node)
{
    queue_pointer ptr;

    ptr = (queue_pointer)malloc(sizeof(struct queue));
    ptr->vertex = node;
    ptr->link = NULL;
    if (!*front)
        *front = ptr;
    else
        (*rear)->link = ptr;
    *rear = ptr;
}
```

Adjacency List(4/5)

```
/* FIFO에 의해 queue에 저장된 값을 하나씩 return & free */
int delete_queue(queue_pointer *front)
{
    queue_pointer ptr = *front;
    int vertex = ptr -> vertex;
    *front = ptr->link;
    free(ptr);
    return vertex;
}
/* DFS 구현 */
void bfs(int v)
{
    pnode w;
    queue_pointer front,rear;
    front = rear = NULL;
    visited[v] = TRUE;
    printf("%d ",v);
    add_queue(&front, &rear, v); // 방문정점을 모두 queue에 저장
    while (front) { // 미방문 정점이 없을때까지 실행
        v = delete_queue(&front); // FIFO에 의한 값을 v에 저장
        for (w = headNodes[v]; w; w=w->link)
            if (!visited[w->vertex]) { // 방문하지 않은 정점일때
                visited[w->vertex] = TRUE;
                printf("%d ", w->vertex);
                add_queue(&front,&rear,w->vertex); // 방문정점을 queue에 저장
            }
    }
}
```

Adjacency List(5/5)

```
int main()
{
    int flag, vertex1, vertex2, del_vertex;
    createGraph(8);
    do{
        printf("1.Insert  2.Delete  3.Print  4.DFS  5.BFS  6.Quit\n");
        scanf("%d",&flag);
        switch(flag){
            case 1:
                printf("Start Vertex와 End Vertex를 입력하세요 (quit:-1 -1)\n");
                scanf("%d%d",&vertex1,&vertex2);
                while(vertex1!=-1){
                    insertEdge(vertex1, vertex2);
                    insertEdge(vertex2, vertex1);
                    printf("Start Vertex와 End Vertex를 입력하세요 (quit:-1 -1)\n");
                    scanf("%d%d",&vertex1,&vertex2);
                }
                break;
            case 2:
                printf("삭제할 Vertex를 입력하세요 : ");
                scanf("%d",&del_vertex);
                deleteVertex(del_vertex);
                break;
            case 3:
                printGraph();
                break;
            case 4:
                vertexInit();
                dfs(0);
                printf("\n");
                break;
            case 5:
                vertexInit();
                bfs(0);
                printf("\n");
                break;
        }
    }while(flag!=6);
    return 0;
}
```

Adjacency Matrix(1/4)

```
#include <stdio.h>
#include <stdlib.h>
#define bool int
#define TRUE 1
#define FALSE 0

typedef struct queue *queue_pointer; // BFS 구현을 위한 queue
typedef struct queue {
    int vertex;
    queue_pointer link;
};

int numofVertex;
int **Vertex_Array; // 2차원 배열
bool *visited; // bool 자료형 사용
int delete_queue(queue_pointer *);
void add_queue(queue_pointer *, queue_pointer *, int);
void dfs(int);
void bfs(int);
void createGraph(int);
void insertEdge(int, int);
void deleteVertex(int);
void printGraph();
void vertexInit();

void createGraph(int numVertex){
    int i=0, j=0;
    numofVertex = numVertex;
    Vertex_Array = (int**)malloc(sizeof(int)*(numVertex));
    for(i=0; i<numVertex ; i++) Vertex_Array[i] = (int*)malloc(sizeof(int)*(numVertex));

    for(i=0; i<numVertex; i++)
        for(j=0; j<numVertex; j++) Vertex_Array[i][j] = 0;

    visited = (int*)malloc(sizeof(int)*numofVertex);
    for(i=0; i<numofVertex; i++) visited[i] = 0;
}

void vertexInit(){
    int i;
    for(i=0; i<numofVertex; i++) visited[i] = 0;
}

void insertEdge(int v1, int v2){
    Vertex_Array[v1][v2] = 1;
}
```

Adjacency Matrix(2/4)

```
void deleteVertex(int Vertex){
    int i=0, j=0;
    for(i=0; i<numofVertex; i++){
        Vertex_Array[Vertex][i]=0;
        Vertex_Array[i][Vertex]=0;
    }
}

void printGraph()
{
    int i=0, j=0;
    printf("    ");
    for(i=0; i<numofVertex; i++)
        printf("[%d] ",i);
    printf("\n");
    for(i=0; i<numofVertex; i++)
    {
        printf("[%d] ",i);
        for(j=0; j<numofVertex; j++){
            printf("%d    ",Vertex_Array[i][j]);
        }
        printf("\n");
    }
}

void dfs(int v)
{
    int i;
    printf("%d ", v);
    visited[v] = 1; // 정점 v를 방문했다고 표시
    for (i = 0; i<numofVertex; i++)
    {
        // 정점 v와 정점 i가 연결되었고, 정점 i를 방문하지 않았다면
        if (Vertex_Array[v][i] == 1 && !visited[i])
        {
            // 정점 i에서 다시 DFS를 시작한다
            dfs(i);
        }
    }
}

void add_queue(queue_pointer *front, queue_pointer *rear, int node)
{
    queue_pointer ptr;

    ptr = (queue_pointer)malloc(sizeof(struct queue));
    ptr->vertex = node;
    ptr->link = NULL;
    if (!*front)
        *front = ptr;
    else
        (*rear)->link = ptr;
    *rear = ptr;
}
```

Adjacency Matrix(3/4)

```
int delete_queue(queue_pointer *front)
{
    queue_pointer ptr = *front;
    int vertex = ptr -> vertex;
    *front = ptr->link;;
    free(ptr);
    return vertex;
}

void bfs(int v){
    int i;
    queue_pointer rear, front;
    rear = front = NULL;
    visited[v] = 1;
    printf("%d ", v);
    add_queue(&front, &rear, v);
    while(front){
        v = delete_queue(&front);
        for(i=0; i<numofVertex; i++){    // i를 정점의 번호로 보면 된다.
            // 방문중인 정점(v)과 연결된 인접정점이 있고, 인접정점이 방문되지 않았을때 if 실행
            if(Vertex_Array[v][i] == 1 && !visited[i]){
                visited[i] = TRUE; // 방문됨을 1로 보인다
                printf("%d ", i);    // 방문중인 정점 출력
                // 방문중인 정점 i의 값을 queue에 저장하고, 꺼낼땐 i에 대한 인접정점을 찾는다
                add_queue(&front, &rear, i);
            }
        }
    }
}
```


Adjacency Matrix(4/4)

```
int main()
{
    int flag, vertex1, vertex2, del_vertex;
    createGraph(8);
    do{
        printf("1.Insert  2.Delete  3.Print  4.DFS  5.BFS  6.Quit\\n");
        scanf("%d",&flag);
        switch(flag){
            case 1:
                printf("Start Vertex와 End Vertex를 입력하세요 (quit:-1 -1)\\n");
                scanf("%d%d",&vertex1,&vertex2);
                while(vertex1!=-1){
                    insertEdge(vertex1, vertex2);
                    insertEdge(vertex2, vertex1);
                    printf("Start Vertex와 End Vertex를 입력하세요 (quit:-1 -1)\\n");
                    scanf("%d%d",&vertex1,&vertex2);
                }
                break;
            case 2:
                printf("삭제할 Vertex를 입력하세요 : ");
                scanf("%d",&del_vertex);
                deleteVertex(del_vertex);
                break;
            case 3:
                printGraph();
                break;
            case 4:
                vertexInit();
                dfs(0);
                printf("\\n");
                break;
            case 5:
                vertexInit();
                bfs(0);
                printf("\\n");
                break;
        }
    }while(flag!=6);
    return 0;
}
```