게임 자료구조와 알고리즘

-CHAPTER6-

SOULSEEK

목차

테이블(TABLE)과 해쉬(HASH)

1. 테이블(TABLE)과 해쉬(HASH)

테이블 자료구조

- 탐색 연산의 시간 복잡도는 O(1)
- 저장되는 데이터는 키(Key)와 값(Value)이 하나의 쌍을 이룬다.
- 키(Key)가 존재하지 않는 값(Value)은 저장할 수 없다. 그리고 모든 키는 중복되지 않는다.
- ex)사전에서 단어는 Key가 되고 단어의 설명은 Value가 된다.
- 사전 구조 또는 맵(map)이라고 부르기도 한다.

사번 : key	직원 : value
99001	양현석 부장
99002	한상현 차장
99003	손현주 과장
99004	이수진 사원

1. 테이블(TABLE)과 해쉬(HASH)

배열을 기반으로 하는 테이블을 고려해보자!

- 배열으로 이루어져 있을 때 배열의 인덱스를 Key값으로 온전히 사용할 수 없다.
  - 배열의 길이는 한정되어 있는데 Key값이 그 범위를 넘어가는 경우 Key값을 인위적으로 바꿔야 한다.
- 원하는 Key값의 범위를 담을 수 있는 사용하지 않는 크기가 남아도는 배열을 선언해야만 한다.

```
typedef struct _empInfo
{
	int empNum;
	int age;
} EmpInfo;

int main()
{
	EmpInfo empInfoArr[1000];
	EmpInfo e1;
	int eNum;

	printf("사번과 나이 입력:");
	scanf("%d %d", &(e1.empNum), &(e1.age));
	empInfoArr[e1.empNum] = e1;

	printf("확인하고픈 직원의 사번 입력: ");
	scanf("%d", &eNum);

	e1 = empInfoArr[eNum];
	printf("사번 %d, 나이 %d \n", e1.empNum, e1.age);

	return 0;
}
```

1. 테이블(TABLE)과 해쉬(HASH)

해쉬(HASH) 함수의 이용

- 제한사항의 조건을 만족하기 위해 키로 활용할 수 있는 것을 특정하여 범위를 좁혀서 사용하게 유도하는 연산이 들어가 있다.
- 동일한 키의 값이 들어가야 하는 충돌(collision)상황이 발생할 수 있다.

```
typedef struct _empInfo
{
        int empNum;
        int age;
} EmpInfo;

int GetHashValue(int empNum)
{
        return empNum % 100;
}

int main(void)
{
        EmpInfo empInfoArr[100];

        EmpInfo emp1 = { 20120003, 42 };
        EmpInfo emp2 = { 20130049, 33 };

        EmpInfo r1, r2;

        empInfoArr[GetHashValue(emp1.empNum)] = emp1;
        empInfoArr[GetHashValue(emp2.empNum)] = emp2;

        r1 = empInfoArr[GetHashValue(20120003)];
        r2 = empInfoArr[GetHashValue(20130049)];

        printf("사번 %d, 나이 %d \n", r1.empNum, r1.age);
        printf("사번 %d, 나이 %d \n", r2.empNum, r2.age);

        return 0;
}
```

충돌이 발생한다.

1. 테이블(TABLE)과 해쉬(HASH)

체계적인 테이블 해쉬 구현

- 예제를 같이 보자!

Person.h

```
typedef struct _person
{
	intssn;
	charname[STR_LEN];
	charaddr[STR_LEN];
} Person;

int GetSSN(Person* p);
void ShowPerInfo(Person* p);
Person* MakePersonData(int ssn, char* name, char* addr);
```

- Person 구조체의 변수가 테이블에 저장될 값 ssn은 Key, Key를 별도로 추출하는데 사용하는 GetSSN함수를 정의.
- MakePersonData함수로 구조체 변수의 생성 및 초기화의 편의를 제공한다.

Person.cpp

```
int GetSSN(Person* p)
{
	return p->ssn;;
}

void ShowPerInfo(Person* p)
{
	printf("주민등록번호: %d \n", p->ssn);
	printf("이름: %d \n", p->name);
	printf("주소: %d \n\n", p->addr);
}
```

```
Person* MakePersonData(int ssn, char* name, char* addr)
{
	Person* newP = (Person*)malloc(sizeof(Person));
	newP->ssn = ssn;
	strcpy(newP->name, name);
	strcpy(newP->addr, addr);

	return newP;
}
```

1. 테이블(TABLE)과 해쉬(HASH)

Slot.h

- 테이블을 이루는, 데이터를 저장할 수 있는 각각의 공간을 위한 정의이다.

```
typedef int         Key;    // 키 값이 되는 주민번호
typedef Person*     Value;  // Person 구조체 변수의 값

//데이터가 저장된적이 없는지, 현재 비워진 상태인지, 유효한 데이터가 저장되어있는지.
enum SlotStatus { EMPTY, DELETED, INUSE };

typedef struct _slot
{
    Key                 key;
    Value               val;
    enum SlotStatus     status; // 슬롯의 상태를 나타낸다.
} Slot;
```

1. 테이블(TABLE)과 해쉬(HASH)

Table.h

```
typedef int HashFunc(Key k);

typedef struct _table
{
	Slot tbl[MAX_TBL];
	HashFunc* hf;
}Table;

// 테이블의 초기화
void TBLInit(Table* pt, HashFunc* f);

// 테이블에 키와 값을 저장
void TBLInsert(Table* pt, Key k, Value v);

// 키를 근거로 테이블에서 데이터 삭제
Value TBLDelete(Table* pt, Key k);

// 키를 근거로 테이블에서 데이터 탐색
Value TBLSearch(Table* pt, Key k);
```

1. 테이블(TABLE)과 해쉬(HASH)

Table.cpp

```
//모든 슬롯 초기화
void TBLInit(Table* pt, HashFunc* f)
{
	for (int i = 0; i < MAX_TBL; i++)
		(pt->tbl[i]).status = EMPTY;

	pt->hf = f; // 해쉬 함수 등록
}

void TBLInsert(Table* pt, Key k, Value v)
{
	int hv = pt->hf(k);
	pt->tbl[hv].val = v;
	pt->tbl[hv].key = k;
	pt->tbl[hv].status = INUSE;
}
```

```
Value TBLDelete(Table* pt, Key k)
{
	int hv = pt->hf(k);

	if ((pt->tbl[hv]).status != INUSE)
	{
		return NULL;
	}
	else
	{
		(pt->tbl[hv]).status = DELETED;
		return (pt->tbl[hv]).val; // 소멸 대상의 값 반환
	}
}

Value TBLSearch(Table* pt, Key k)
{
	int hv = pt->hf(k);

	if ((pt->tbl[hv]).status != INUSE)
		return NULL;
	else
		return (pt->tbl[hv]).val; // 탐색 대상의 값 반환
}
```

1. 테이블(TABLE)과 해쉬(HASH)

학습과제

TableHashFunction 프로젝트를 확인하고 다시 해당 코드의 흐름들을 읽어보자.
TableHashFuncMain.cpp의 구현을 확인하고 또 다른 활용을 만들어서 공부해 보자

1. 테이블(TABLE)과 해쉬(HASH)

충돌문제의 해결책

- 해결책들의 각 방법을 알아보고 적합한걸 선택해서 그것을 기준으로 구현해 보자.
- 해쉬 값이 충돌하면 충돌 발생시 빈 슬롯을 찾기 위해서 접근하는 위치가 일정하다.

선형 조사법(Linear Probing)

- 들어갈 데이터의 옆 공간이 비었는지 확인하고, 비었을 경우 그 자리에 대신 저장하는 것.

해쉬 함수가 key % 7이라고 가정하고,
키가 9인 데이터는 해쉬 값이 2이므로 인덱스가 2인 위치에 저장된다.

해쉬 함수가 key % 7이라고 가정하고,
키가2인 데이터를 저장하게 되면 충돌이 발생한다. 9와 충돌하였으므로 인덱스2의 옆인 3을 검색해본다.

계속 옆자리를 비교하면서 자리를 찾아간다.
f(k) + 1... f(k) + 2... f(k) + 3... f(k) + 4.....

충돌 횟수가 증가함에 따라서 특정지역에 데이터가 몰리는 **클러스터 현상이 발생**한다.
이는 좋지 않은 해쉬 방법이다.

1. 테이블(TABLE)과 해쉬(HASH)

이차 조사법(Quadratic Probing)

- 선형 조사법 보다 조금 더 간격을 두고 저장한다.
- **f(k) +** 1^2**... f(k) +** 2^2 **... f(k) +** 3^2 **... f(k) +** 4^2 **.....**
- 해쉬 값이 충돌하면 충돌 발생시 빈 슬롯을 찾기 위해서 접근하는 위치가 일정하다.

이중 해쉬(Double Hash)

- 해쉬 조건을 불규칙하게 만들기 위해 두개의 해쉬 함수를 이용해서 빈 공간을 찾는 방법
- 키가 다르면 건너 뛰는 길이도 달라지기때문에 클러스터 현상의 발생 확률을 현저히 낮출 수 있기 때문에 **이상적인 해결책 중의 하나**이다.

ex)
1차 해쉬 함수 **:** 키를 근거로 저장위치를 결정하기 위한 것
2차 해쉬 함수 **:** 충돌 발생시 몇 칸 뒤를 살필지 결정하기 위한 것

1차 해쉬 함수 **: h1(k) = k % 15**
2차 해쉬 함수 **: h2(k) = 1 + (k % c)**

1차 해쉬 함수를 통해서 배열의 개수가 **15**인걸 예상**(**배열의 길이를 넘지 않기 위해**)** 할 수 있다**.**
이때**, c**는 **15**보다 작으면서 소수인 숫자 중 하나로 결정하게 된다**.**

1차 해쉬 함수 **: h1(k) = k % 15**
2차 해쉬 함수 **: h2(k) = 1 + (k % 7)** 로 결정할 수 있다**.**

1. 테이블(TABLE)과 해쉬(HASH)

체이닝

- 빈 공간을 찾는 것이 아닌 그 자리에 그대로 데이터를 추가 시키는 방법
- 연결 리스트를 이용해서 해당 키와 중복된 데이터들은 키를 헤더로 연결 시켜 놓는 방법

체이닝의 구현 – 앞선 **Person**을 활용한다.

- **Slot.h, Table.h**에 **DLinkedList**를 이용하면 변경되어지는 부분이 생기게 된다.

Slot.h

```
typedef int Key;
typedef Person * Value;

//enum SlotStatus { EMPTY, DELETED, INUSE };// 제외된 부분

typedef struct _slot
{
	Key key;
	Value val;
} Slot;
```

체이닝을 사용하는 방법에서는 슬롯의 상태가 필요하지 않기 때문에 **enum**문을 제외한다.

1. 테이블(TABLE)과 해쉬(HASH)

Table.h

```
typedef int HashFunc(Key k);

typedef struct _table
{
	//Slot tbl[MAX_TBL];
	List tbl[MAX_TBL];
	HashFunc * hf;
} Table;

void TBLInit(Table * pt, HashFunc * f);
void TBLInsert(Table * pt, Key k, Value v);
Value TBLDelete(Table * pt, Key k);
Value TBLSearch(Table * pt, Key k);
```

- **Table**의 저장소였던 **Slot**형 배열을 **List**형 배열로 교체

1. 테이블(TABLE)과 해쉬(HASH)

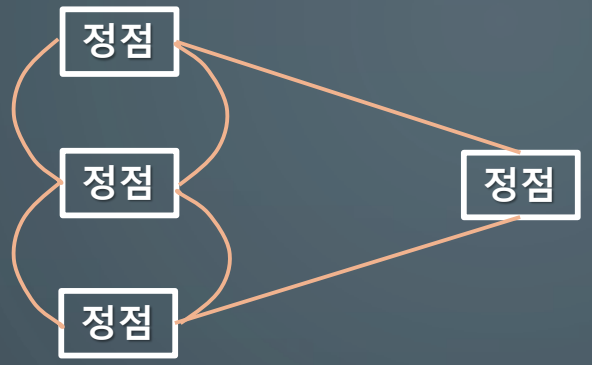
학습과제

ChainedTableHash 프로젝트를 확인하고 구현부분을 이해하자**.**
DLinkedList를 이해했다면 충분히 쉬울 것이다**.** 구현 되어 있는 **Main**함수를 확인하고 또 다른 예를 만들어 실행해 보자**.**

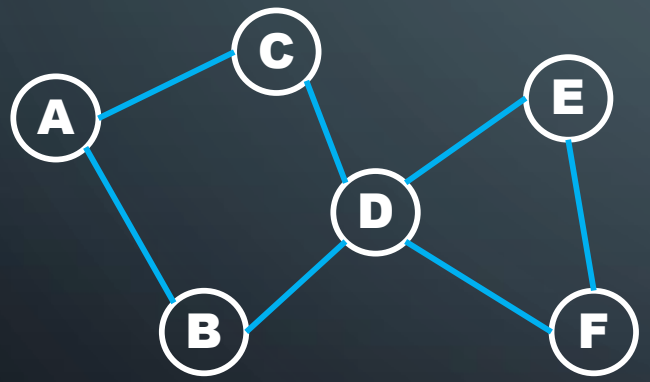
그래프(GRAPH)

2. 그래프(GRAPH)

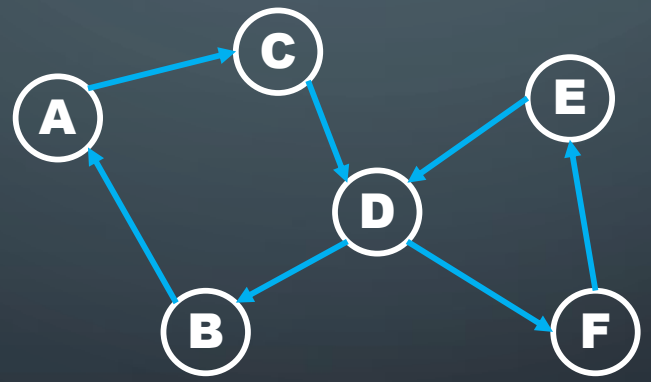
- 정점과 간선이 존재하고, 정점과 간선끼리 서로 연결되어 있는 형태



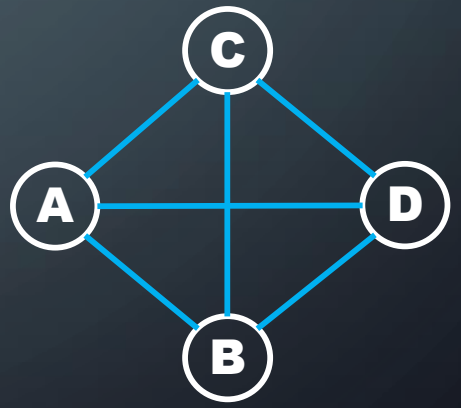
그래프의 종류



무방향 그래프



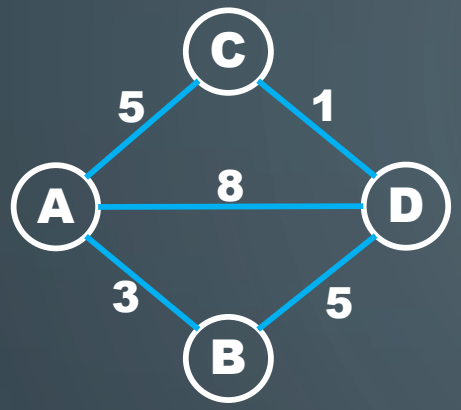
방향 그래프



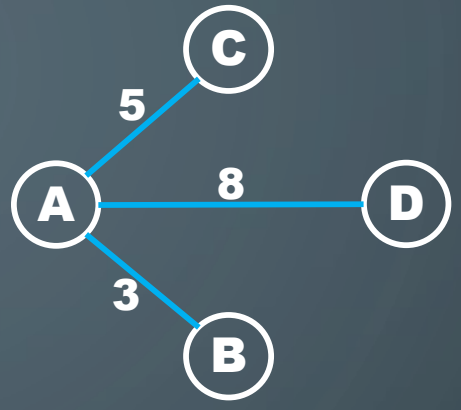
완전 그래프

각 정점이 나머지 모든 정점과 연결된 그래프

2. 그래프(GRAPH)



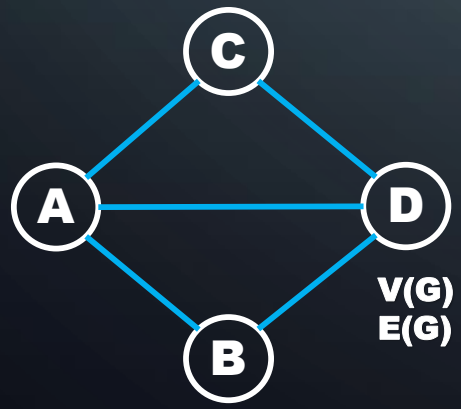
가중치 그래프
간선에 가중치 정보를
두어서 구성하는 그래프



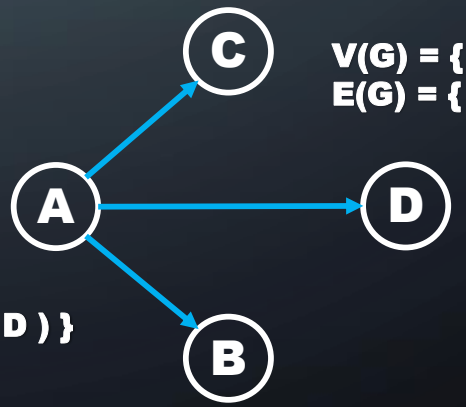
부분 그래프
일부 정점과 간선으로
이루어진 그래프

그래프의 집합 – 정점과 간선의 집합

간선의 집합을 V(G), 정점의 집합을 E(G)로 표현한다면..



V(G) = { A, B, C, D }
E(G) = { (A, B), (A, C), (A, D), (B, D), (C, D) }



V(G) = { A, B, C, D }
E(G) = { < A, B >, < A, C >, < A, D > }

2. 그래프(GRAPH)

그래프의 ADT

- 생성 및 초기화 할 때 간선의 방향성 여부를 선택할 수 있고, 가중치의 부여 여부도 선택할 수 있으며, 이후에는 정점과 간선을 얼마든지 언제든지 삭제 할 수 있다.
- 구성에 있어서 표현하는 부분을 먼저 구성한 후 필요한 부분을 완성해 나가자.

- **void GraphInit(UALGraph* pg, int nv);**
  - 그래프의 초기화를 진행한다.
  - 두 번째 인자로 정점의 수를 전달한다.

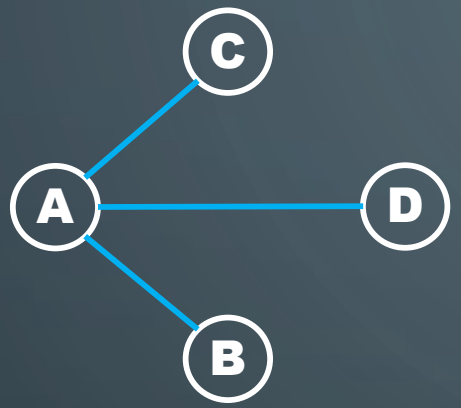
- **void GraphDestroy(UALGraph* pg);**
  - 그래프 초기화 과정에서 할당한 리소스를 반환한다.

- **void AddEdge(UALGraph* pg, int fromV, int toV);**
  - 매개변수 **fromV**와 **toV**로 전달된 정점을 연결하는 간선을 그래프에 추가한다.

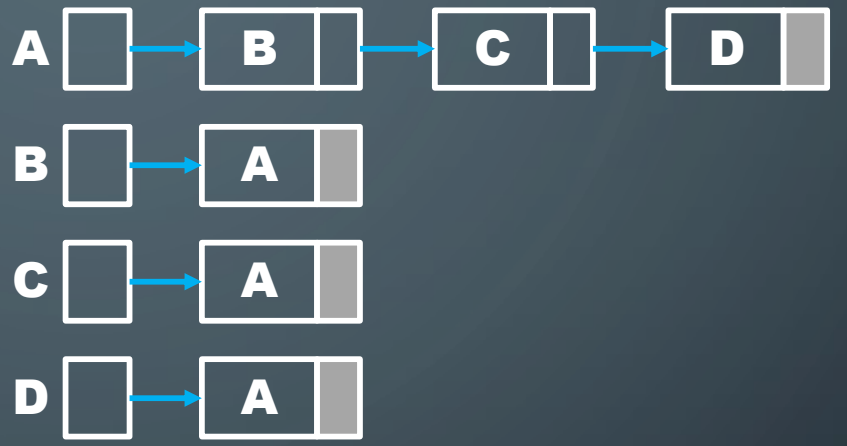
- **void ShowGraphEdgeInfo(UALGraph* pg);**
  - 그래프의 간선정보를 출력한다.

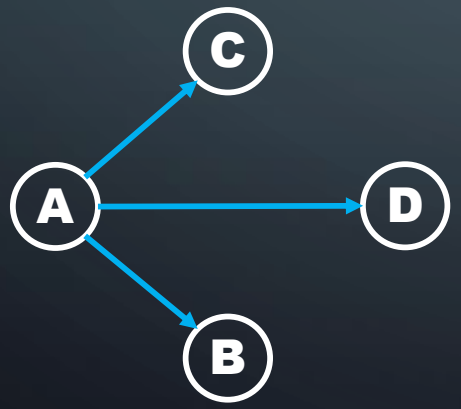
2. 그래프(GRAPH)

인접 리스트(Linked List 활용)를 이용하는 그래프의 구현

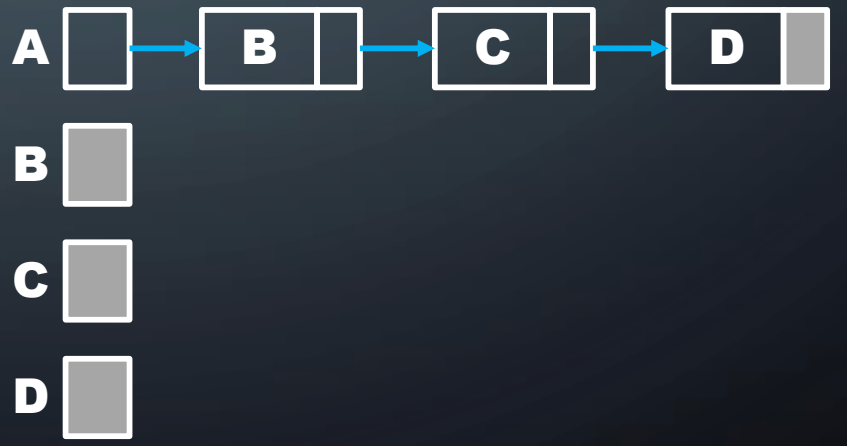


무방향 그래프





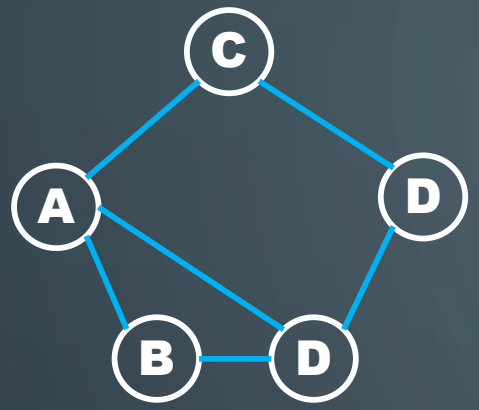
방향 그래프



2. 그래프(GRAPH)

```
// 정점의 이름들을 상수화
enum { A, B, C, D, E, F, G, H, I, J };

typedef struct _ual
{
	int numV;	// 정점의 수
	int numE;	// 간선의 수
	List * adjList;	// 간선의 정보
} ALGraph;

// 그래프의 초기화
void GraphInit(ALGraph * pg, int nv);

// 그래프의 리소스 해제
void GraphDestroy(ALGraph * pg);

// 간선의 추가
void AddEdge(ALGraph * pg, int fromV, int toV);

// 유틸리티 함수: 간선의 정보 출력
void ShowGraphEdgeInfo(ALGraph * pg);
```

2. 그래프(GRAPH)



"이런 연결 모양을 이루는 그래프를 만든다고 가정하자."

```
void GraphInit(ALGraph * pg, int nv)
{
	// 정점의 수에 해당하는 길이의 리스트 배열을 생성한다.
	pg->adjList = (List*)malloc(sizeof(List)*nv); // 간선정보를 저장할 리스트 생성
	pg->numV = nv;		// 정점의 수는 nv에 저장된 값으로 결정
	pg->numE = 0;		// 초기의 간선 수는 0개

	//정점의 수만큼 생성된 리스트들을 초기화 한다.
	for (i = 0; i < nv; i++)
	{
		ListInit(&(pg->adjList[i]));

		// 필수 요소는 아니다, 순서대로 보여주기 위한 구현.
		SetSortRule(&(pg->adjList[i]), WhoIsPrecede); // 리스트의 정렬기준을 설정
	}
}
```

2. 그래프(GRAPH)

```
void GraphDestroy(ALGraph * pg) // 그래프 리소스의 해제
{
    if (pg->adjList != NULL)
        free(pg->adjList);      // 동적으로 할당된 연결 리스트의 소멸
}

void AddEdge(ALGraph * pg, int fromV, int toV) // fromV, toV 연결하는 간선 추가.
{
    // 정점 fromV의 연결 리스트에 정점 toV의 정보 추가
    LInsert(&(pg->adjList[fromV]), toV);

    // 정점 toV의 연결 리스트에 정점 fromV의 정보 추가
    LInsert(&(pg->adjList[toV]), fromV);
    pg->numE += 1;
}
```

ALGraph 프로젝트를 확인하면 추가까지의 완성된 그래프의 출력을 볼 수 있다.

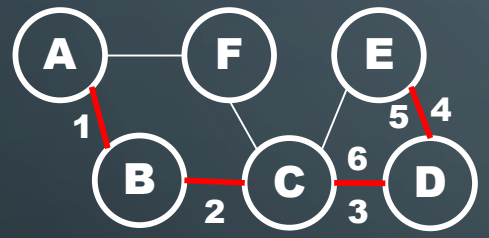
2. 그래프(GRAPH)

탐색

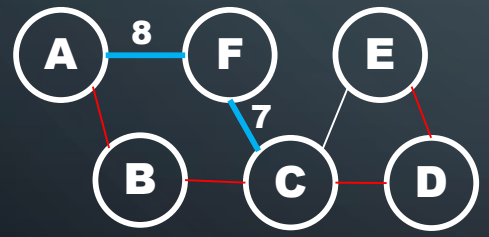
- 리스트와 트리와는 다르게 규칙적인 연결구조가 없기 때문에 탐색과정이 제일 어려운 구조.
- 깊이우선 탐색, 너비우선 탐색 방법이 존재한다.

깊이우선의 탐색(Depth First Search : DFS)

- 한 쪽 방향으로 연결된 최대한 끝까지 탐색한 후 다른 연결에 대한 탐색을 각각 진행한다.
- 한 정점에 여러 개의 간선이 있다면 우선순위의 선택은 사용자에게 있다.



단계1. A→B→C→D→E 순서로 탐색
E에서 탐색해야 하는데 이미 모두 탐색이 완료된 상태이기 때문에 **D**에게 탐색을 알려준다.
D에서 탐색을 시작하지만 이미 모두 탐색이 완료된 상태이기 때문에 **C**에게 탐색을 알려준다.



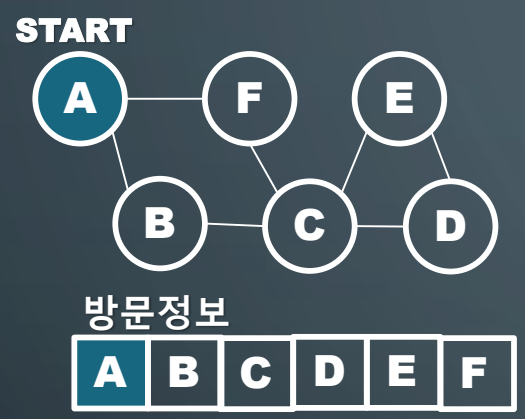
단계2. C→F→A 순서로 탐색
C와 연결된 탐색하지 않은 **F**를 탐색한다.
F와 연결되었지만 시작하면서 탐색한 **A**는 이미 탐색되었다.
하지만, 탐색되어진 **A**도 다른 연결이 존재할 수 있기때문에 **A**가 탐색하도록 탐색을 넘겨준다.

각 정점이 보여지는 그림에서는 양쪽 끝 또는 중간지점이다
단순히 그림만 그려 놓고 그대로 이해하면 어는 곳이 끝점이고 중간지점인지 필요해 진다.
정점마다 간선의 유무를 판단하는 것을 실행하는 것 하나로 어떤 루트로 진행할지 정해진다.

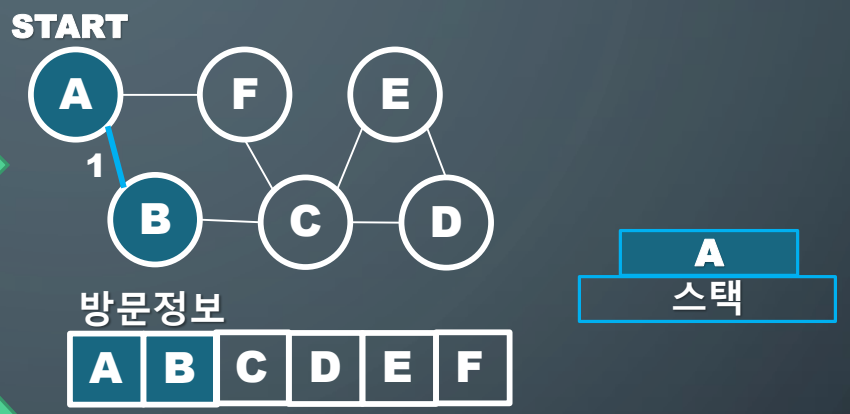
2. 그래프(GRAPH)

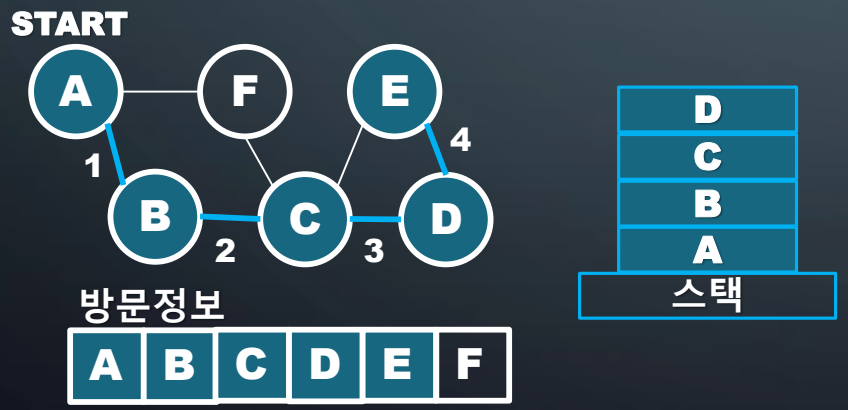
깊이우선 탐색의 구현

- 스택과 배열이 필요요소이다.
- 스택은 경로 정보의 추적을 목적으로 한다.
- 배열은 방문 정보의 기록을 목적으로 한다.



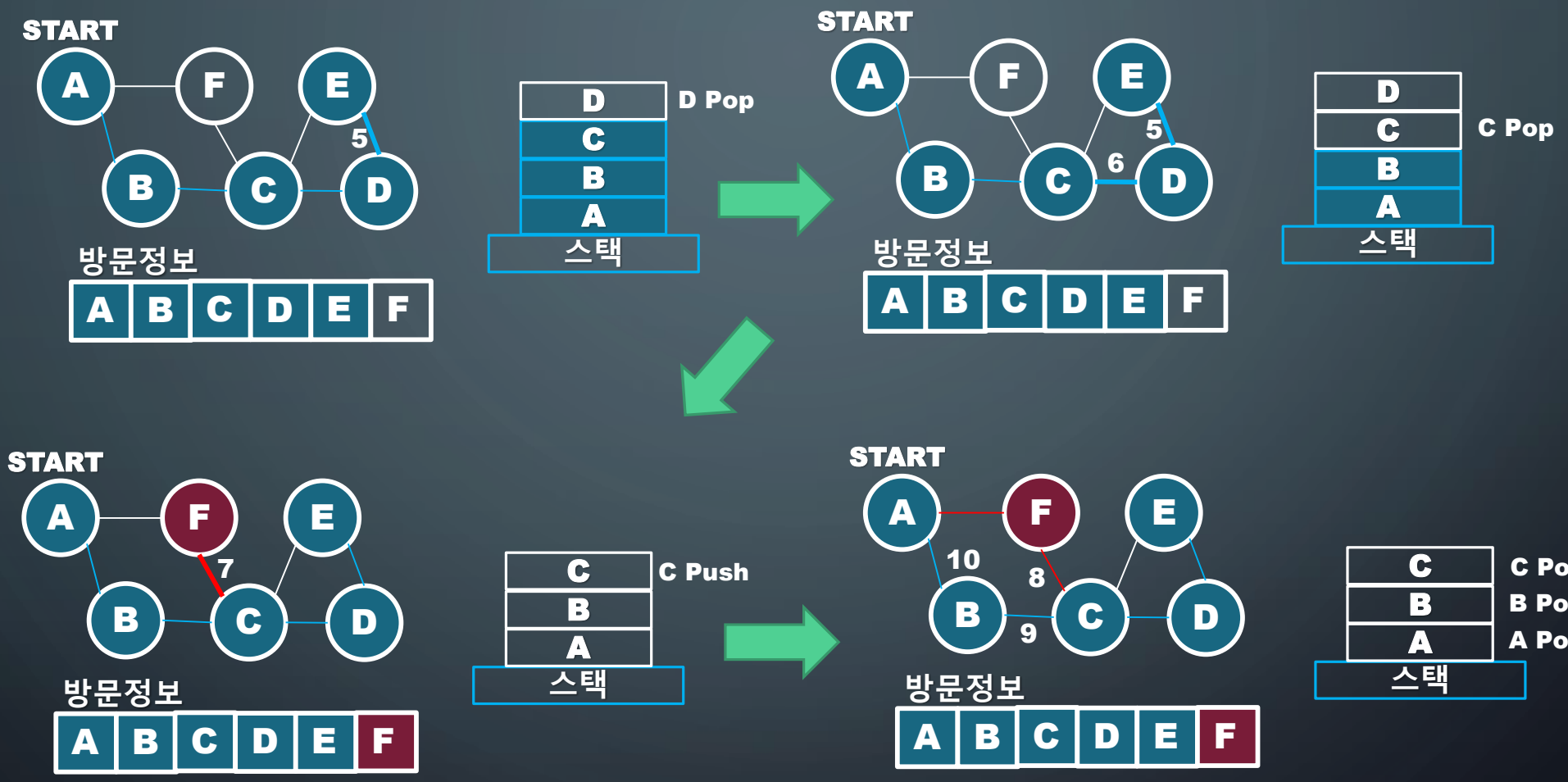
스택





방문하고 다음으로 이동하면 스택에 쌓아준다.

2. 그래프(GRAPH)



끝까지 탐색한 후 이미 탐색을 해서 더 이상 탐색을 할 것이 연결되지 않은 정점은 자신에게 탐색을 지시한 정점으로 다시 돌려줘야 한다. 그러면서 **Pop**을 해주고 탐색을 돌려주다가 다시 탐색 정점을 발견하면 탐색을 넘겨주고 **Push**해준다.
탐색과 더 이상 탐색이 없으면 다시 탐색을 돌려줘서 다른 간선이 없는지 확인해주는 과정이 필요하다.

2. 그래프(GRAPH)

```
typedef struct _ual
{
	int numV;   // 정점의 수
	int numE;   // 간선의 수
	List * adjList;   // 간선의 정보
	int * visitInfo;   // DFS 탐색과정의 정보를 담는다
} ALGraph;
```

초기화

```
void GraphInit(ALGraph * pg, int nv)
{
	pg->adjList = (List*)malloc(sizeof(List)*nv);
	pg->numV = nv;
	pg->numE = 0;     // 초기의 간선 수는 0개

	for (int i = 0; i < nv; i++)
	{
		ListInit(&(pg->adjList[i]));
		SetSortRule(&(pg->adjList[i]), WhoIsPrecede);
	}

	//정점의 수를 길이로 하여 배열을 할당
	Pg->visitInfo = (int *)malloc(sizeof(int) * pg->numV);
	//배열의 모든 요소를 0으로 초기화
	memset(pg->visitInfo, 0, sizeof(int) * pg->numV);
}
```

2. 그래프(GRAPH)

삭제

```
void GraphDestroy(ALGraph * pg)
{
	// 그래프의 리소스 해제
	if (pg->adjList != NULL)
		free(pg->adjList);

	// 할당된 배열의 소멸
	if (pg->visitInfo != NULL)
		free(pg->visitInfo);
}
```

정점의 방문과 탐색을 표시하기위한 정보출력

void DFShowGraphVertex(ALGraph * pg, int startV) // 그래프의 정점 정보 출력
Int VisitVertex(ALGraph * pg, int visitV) // 정점의 방문을 진행
- **DFShowGraphVertex**의 내부에서 탐색을 진행하면서 사용된다.

2. 그래프(GRAPH)

```
int VisitVertex(ALGraph * pg, int visitV)   //두 번째 매개변수로 전달된 이름의 정점을 방문
{
      If (pg->visitInfo[visitV] == 0) // 방문 기록이 있으면 방문하지 않는다.
      {
            pg->visitInfo[visitV] = 1;        // visitV에 방문한 것으로 기록 – 1로 기록하면 방문
            printf( " %c  " , visitV + 65);   // 방문한 정점의 이름을 출력
            return TRUE;                      //방문성공
      }

      return FALSE;     //방문 실패
}
```

2. 그래프(GRAPH)

```
// Depth First Search: 정점의 정보 출력
void DFShowGraphVertex(ALGraph * pg, int startV)
{
    // ... DFS를 위한 스택의 초기화 하고 시작 정점을 방문한다....

    // 모든 정점의 방문이 이루어 진다.
    while (LFirst(&(pg->adjList[visitV]), &nextV) == TRUE)
    {
        • LFirst 함수호출을 통해서 visitV에 연결된 정점 하나를 얻는다.
        • 정점의 정보는 nextV에 저장된다.

        // nextV에 담긴 정점의 정보를 가지고 방문을 시도한다.
        if (VisitVertex(pg, nextV) == TRUE)
        {
            • nextV의 방문에 성공했으니, visitV의 정보는 스택에 PUSH
            • nextV에 담긴 정보를 visitV에 담고서 while문 다시 시작
            • While문을 다시 시작하는 것은 또 다른 정점의 방문을 시도
        }

        ........
}
```

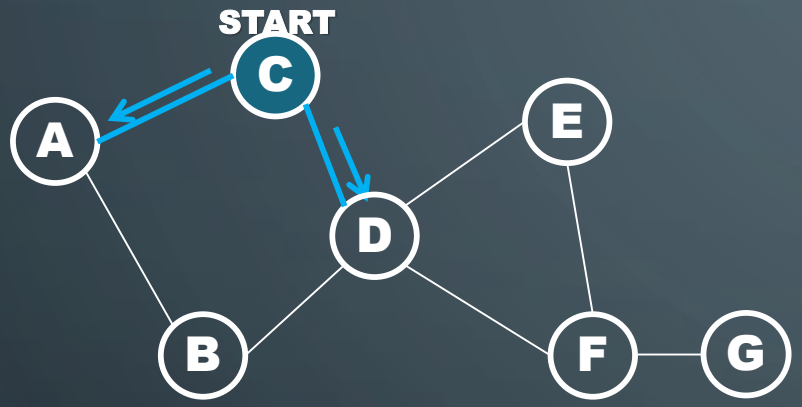
2. 그래프(GRAPH)

```
void DFShowGraphVertex(ALGraph * pg, int startV)
{
    while (LFirst(&(pg->adjList[visitV]), &nextV) == TRUE)
    {
        ..............
        // LFirst 함수호출을 통해서 얻은 정점의 방문에 실패한 경우 else 구문 실행
        else
        {
            //visitV에 연결된 정점을 찾을 때까지 반복한다.
            while (LNext(&(pg->adjList[visitV]), &nextV) == TRUE)
            {
                • LNext 함수호출을 통해서 visitV에 연결된 정점 하나를 얻는다.
                • 이렇게 해서 얻은 정점의 정보는 nextV에 저장된다.

                // nextV에 담긴 정점의 정보를 가지고 방문을 시도한다.
                If (VisitVertex(pg, nextV) == TRUE)
                {
                    • nextV의 방문에 성공했으니, visitV의 정보는 스택에 PUSH
                    • nextV에 담긴 정보를 visitV에 담고서 Break
                }
            }
        }

        //정점 방문에 실패했다면 그에 따른 처리를 진행.
        if (visitFlag == FALSE)
        {
            • 길을 되돌아 가거나 시작 위치로 되돌아와서 프로그램을 종료하자!
        }
```

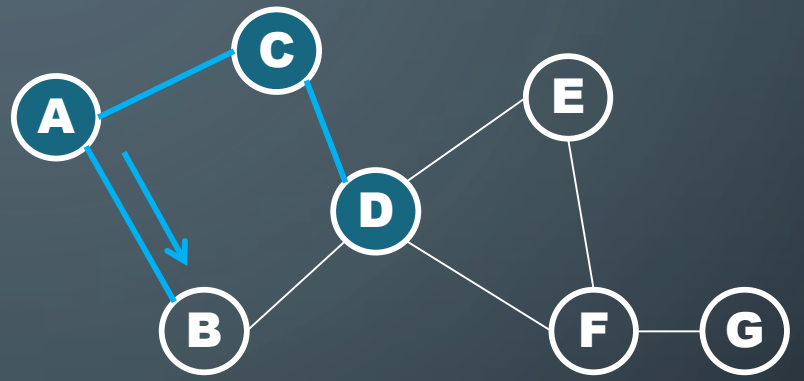
2. 그래프(GRAPH)

너비 우선 탐색(Breadth First Search : BFS)

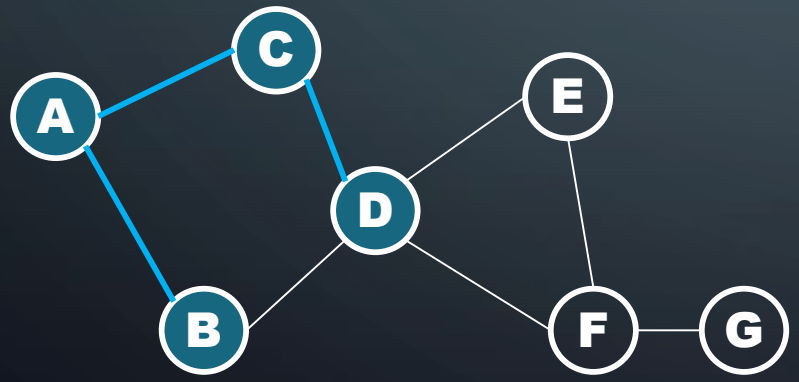
- 자신에게 연결된 모든 정점에 탐색을 요청하는 방식.
- 동시에 같은 정점으로 접근할 때는 사용자가 설정해서 우선순위를 정해주면 된다.



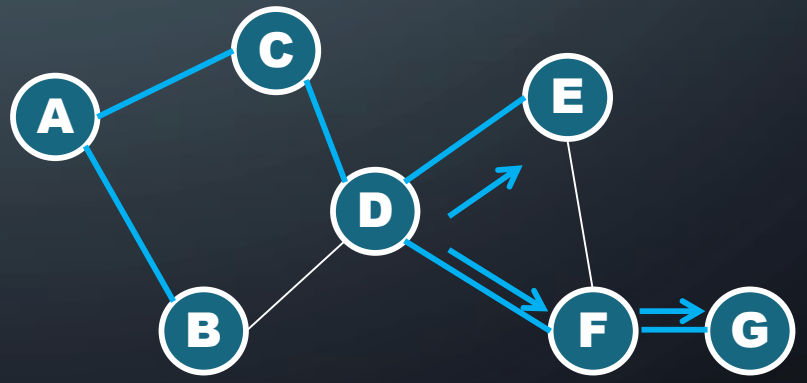
연결된 모든 정점에 탐색을 넘긴다.



우선순위가 A라면 A를 다음으로 탐색을 넘겨준다



D가 먼저 기다리고 있었기 때문에
B가 아닌D와 연결된 것을 탐색한다.

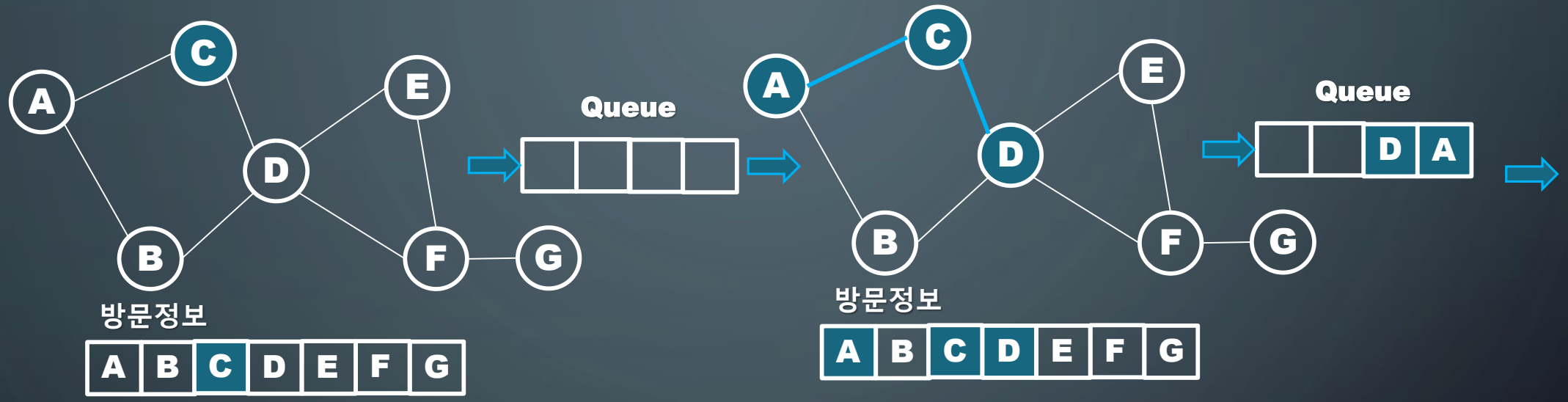


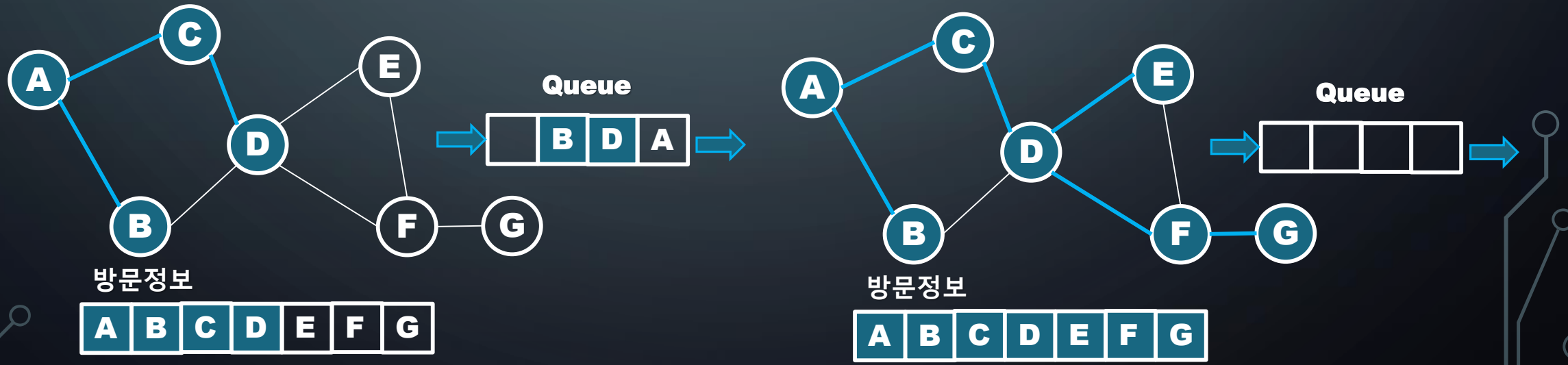
D에서 넘겨준 후 **E, F**를 탐색해서
F에서 **G**로 넘겨준다 남은 곳과 연결한 곳은 이미 탐색한곳

2. 그래프(GRAPH)

너비 우선 탐색의 구현

- 스택 대신 큐를 사용한다.





2. 그래프(GRAPH)

DFS랑 달라지는 것은**...**

핵심함수인 **DFShwoGraphVertex(ALGraph* pg, int startV)**를 바꾸는 일뿐이다.

학습과제

ALGraphDFS, ALGraphBFS를 확인해보고 우선순위(둘 중 어느 정점을 먼저 탐색할지에 대한..)
조건을 조정해서 변경해보자.