stack

≔ 태그

Stack
- 개요
- 구현
- 응용
- 재귀호출
  - factorial
  - 피보나치 수열
- Memoization
- **DP**
  - DFS

Stack

개요

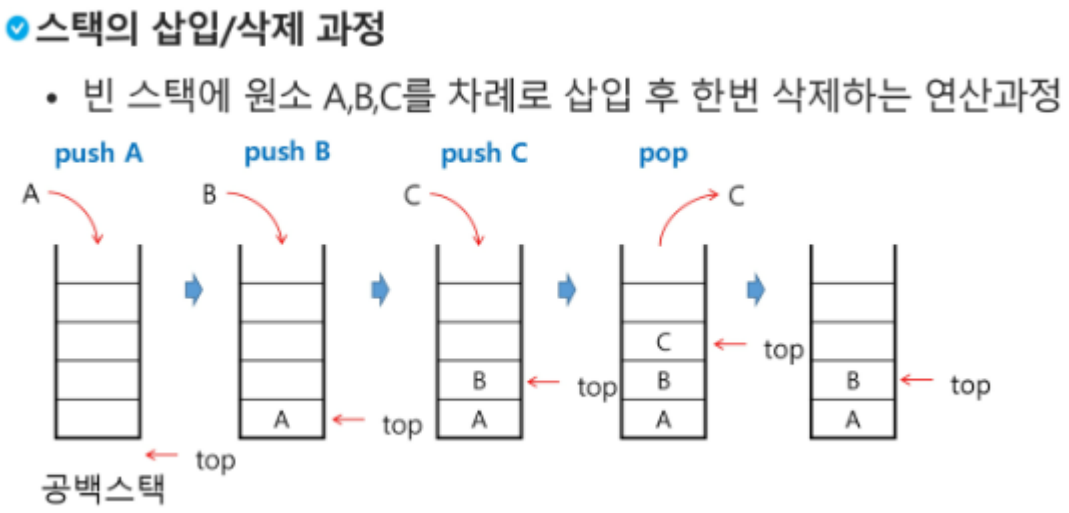
- 특성
  - 물건을 쌓아 올리듯 자료를 쌓아 올린 자료구조
  - 스택에 저장된 자료는 '선형 구조'를 갖는다. (1대1의 관계)
  - 스택에 자료를 삽입하거나 자료를 꺼낼 수 있다.
  - **가장 마지막에 삽입한 자료를 가장 먼저 꺼낸다 (후입 선출)**

구현

- 배열을 사용할 수 있다.
- 마지막에 삽입된 원소의 위치를 탑이라고 부른다.

- 연산
  - 삽입 : 저장소에 자료를 저장한다 push라고 부른다.
  - 삭제 : 저장소에서 자료를 꺼낸다. 꺼낸 자료는 삽입한 자료의 **역순으로 꺼낸다.**
  - 스택이 공백인지 확인하는 연산 : isEmpty
  - 스택의 top에 있는 item을 반환하는 연산 : peek

- 과정



1. 어느 자리에 저장할지 top의 위치로 결정할 수 있다.

   ex) push B 는 top + 1 의 위치에 B를 넣는것

   pop C 는 top 위치의 것을 꺼내고, top - 1 하는 것.

- 스택의 push 알고리즘
  - append 메소드를 통해 리스트의 마지막에 데이터를 삽입할 수 있다.

- stack 을 좀 넉넉하게 만들어서 top의 위치에 들어가게 할 수 있다.

```
def push(item, size):
    global top
    top += 1
    if top == size:
        print('overflow')
    else:
        stack[top] = item

size = 10
stack = [0] * size
top = -1

push(10,size)
top += 1
stack[top] = 20
```

- pop 알고리즘

```
def pop():
    if len(s) == 0:
        # underflow
        return
    else:
        return s.pop();


def pop():
    global top
    if top == -1:
        print('underflow')
        return 0
    else:
```

```
        top -= 1
        return stack[top+1]

print(pop())

if top > -1:
    top -= 1
    print(stack[top + 1])
```

- 실습

```
def push(n, size):
    global top
    top += 1
    if top == size:
        print('overflow')
    else:
        stack[top] = n



top = -1
stack = [0] * 10 # 최대 10개 push
size = 10

top += 1  # push(10)
stack[top] = 10
top += 1  # push(20)
stack[top] = 20

push(30,10)

while top >= 0:
    top -= 1
    print(stack[top+1])
```

- 구현 고려 사항

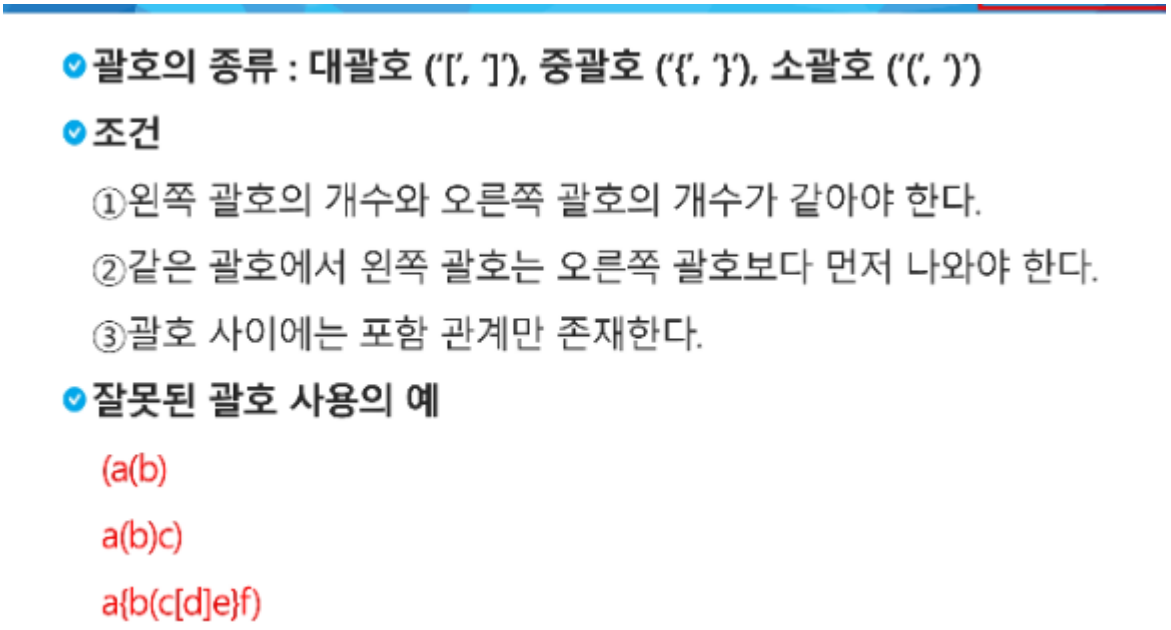
1차원 배열을 사용하여 구현할 경우 구현이 용이하다는 장점, 허나 스택의 크기를 변경하기 어렵다는 단점

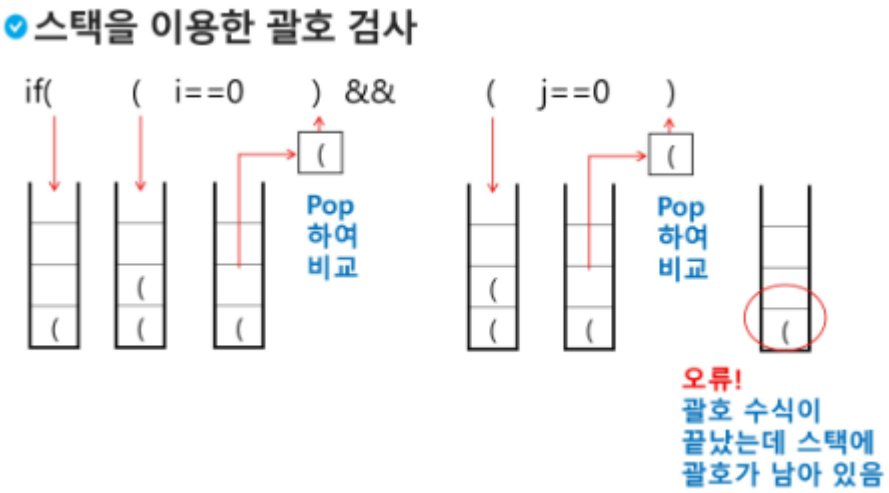
⇒ 저장소를 동적으로 할당하여 스택을 구현할 수 있다.

⇒ 동적 연결 리스트를 이용하여 구현하는 방법을 의미 ⇒ 메모리를 효율적으로 사용가능

응용

1. 괄호 검사





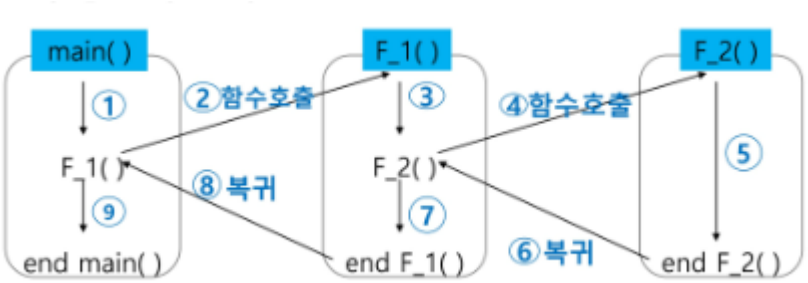
- 문자열에 있는 괄호를 차례대로 조사하면서, 왼쪽 괄호를 만나면 삽입, 오른쪽 괄호 만나면 top 괄호 삭제한 후, 오른쪽 괄호와 짝이 맞는지 검사.
  - 닫는괄호인데, 스택이 비어있으면 2에 위배, 짝이 마지 않으면 3에 위배
  - 마지막 괄호 조사했는데도 남아있으면 1에 위배

- 클래스를 활용한 괄호 검사

```
```

2. Function call

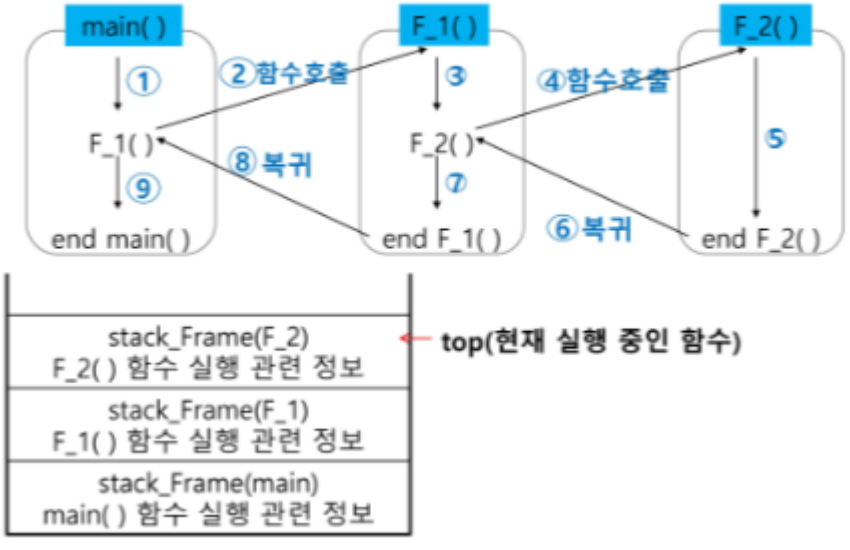
- 함수 호출과 복귀에 따른 수행 순서를 관리
  - 가장 마지막에 호출된 함수가 가장 먼저 실행 완료, 복귀하는 후입 선출 구조 → 스택이다.



- 정보를 스택 프레임에 저장하여, 시스템 스택에 삽입

- 수행순서

함수 호출과 복귀에 따른 전체 프로그램의 수행 순서



- 코드

```
def f2(n):
    n *= 2
    print(n)

def f1(c,d):
    e = c+d
    f2(e)

a = 10
b = 20
c = a + b
f1(a,b)
```

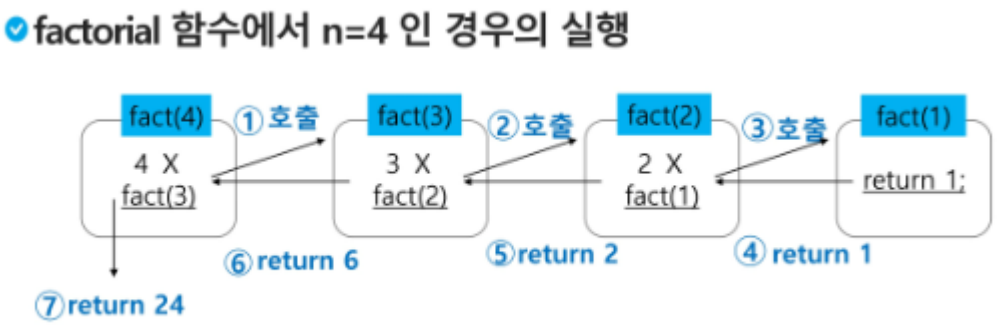
재귀호출

- 필요한 함수가 자신과 같은 경우 자신을 다시 호출하는 구조
- 함수에서 실행해야 하는 작업의 특성에 따라 일반적인 호출방식보다, 재귀호출방식을 사용하여 함수를 만들면 프로그램의 크기를 줄이고 간단하게 작성.

⇒ 다른 함수를 호출하는 것과 메모리상으로는 동일하다. ⇒ 스택에 쌓이는 것이다.

factorial

- 1부터 n까지의 모든 자연수를 곱하여 구하는 연산



⇒ 실제 다른 함수들을 계속 호출 하는 것과 같다..

피보나치 수열

- 0과 1로 시작하고, 이전의 두 수 합을 다음 항으로 한느 수열을 피보나치
- 피보나치 수열의 i번째 값을 계산하는 함수 F를 정의하면 다음과 같다.
- 피보나치 수열의 i번째 항을 반환하는 함수를 재귀함수로 구현할 수 있다.

```
def fibo(n):
    if n<2:
        return n
    else:
        return fibo(n-1) + fibo(n-2)

print(fibo(8))
```

- 가장 기본형 재귀함수

```
def f(i,k):    # 현재위치 i, 목표치 k
    if i == k:
        print(brr)
```

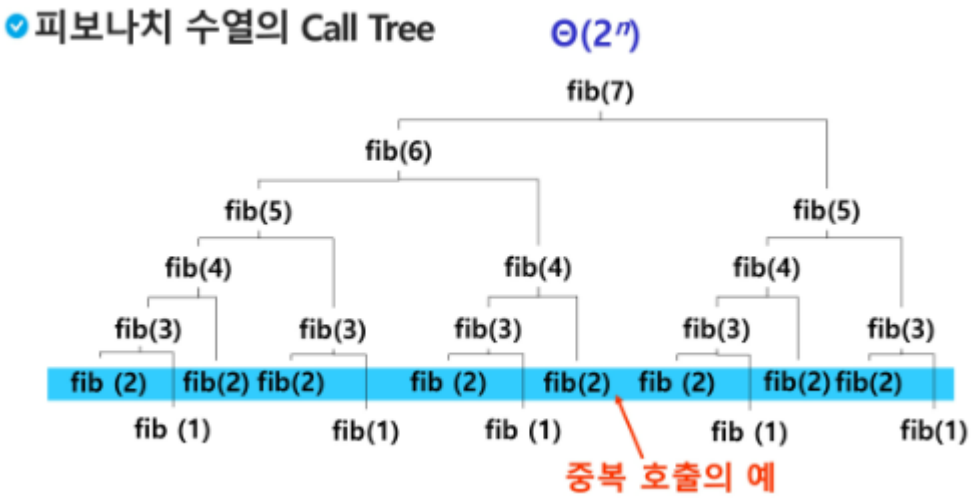
```
    else:
        brr[i] = arr[i]
        f(i+1,k)


arr = [10,20,30]
N   = len(arr)
brr = [0] * N
f(0, N)

=> 목표치에 다가가는 것, 재귀함수의 호출도 결국 다른 함수의 호출과 같다는
```

Memoization

- 피보나치의 수를 구하는 함수를 재귀함수로 구현한다면 ⇒ 너무 큰 중복 호출이 존재한다.



⇒ 한번 구한거는 다시 계산하지 말자 → 기억해놓고 쓰자 → **Memoization**

```
def fibo(n):
    global cnt
    cnt += 1
    if n<2:
        return n
    else:
        return fibo(n-1) + fibo(n-2)

def fibo_memo(n):
    global cnt
    cnt += 1
    if n!= 0 and memo[n] == 0:
        memo[n] = fibo_memo(n-1) + fibo_memo(n-2)
    return memo[n]

cnt = 0
n = 7
memo = [0] * (n+1)
memo[0] = 0
memo[1] = 1
print(fibo(7), cnt)
cnt = 0
print(fibo_memo(n), cnt)

피보나치에 메모이제이션을 적용했을 때의 동작 차이

13 41 # 재귀만 사용
13 13 # 메모이제이션 사용
```

DP

- 동적 계획 이란?? ⇒ 최적화 문제를 해결하는 알고리즘이다.
- 입력 부분이 작은 문제들을 해결하고, 큰 크기의 문제를 해결해서 다가가는 방법

- 피보나치 수에 DP 적용

1. 문제의 분할

1) Fibonacci(n)함수는 Fibonacci(n-1)과 Fibonacci(n-2)의 합

2) n-1은 n-2와 n-3의 합

3) 2 는 1과 0의 합

→ 부분집합 구성

2. 가장 작은 부분부터 문제 해결

3. 그 결과를 **테이블에 저장**

3) 그 결과는 테이블에 저장하고, 테이블에 저장된 부분 문제의 해를 이용하여 상위 문제의 해를 구한다.

테이블 인덱스	저장되어 있는 값
[0]	0
[1]	1
[2]	1
[3]	2
[4]	3
...	...
[n]	fibo(n)

⇒ 부분해를 골랐을 때 그게 최적해가 아니면 DP로 풀 수는 없다

- 구현방식

1. recursive 방식 : fib1()
2. iterative 방식 : fib2()

- memoization을 재귀적 구조에 사용하는 거 보다, 반복적 구조로 DP를 구현하는 것이 성능 면에서 효율적이다.
- 재귀적 구조는 내부에 시스템 호출 스택을 사용하는 오버헤드가 발생한다.

```
def f(i, k):
    if i == k:
        print('end')
    else:
        f(i+1, k)

f(0, 1000)

=> 오버헤드가 발생한다.
```

DFS

- 비선형구조인 그래프 구조는 그래프로 표현된 모든 자료를 빠짐없이 검색하는 것이 중요함.
- 깊이 우선 탐색(DFS) / 너비 우선 탐색(BFS)

깊이 우선 탐색이란?

- 시작 정점의 한 방향으로 갈 수 있는 경로가 있는 곳까지 깊이 탐색해 가다가, 더 이상 갈 곳이 없게 되면, 가장 마지막에 만났던 갈림길 간선이 있는 정점으로 되돌아와서 다른 방향의 정점으로 탐색을 계속 반복하여 결국 모든 정점을 방문하는 순회방법

- 가장 마지막에 만났던 갈림길의 정점으로 돌아가서, 다시 깊이 우선 탐색을 반복해야 하므로

'스택'을 사용하는 것이다.

- 과정

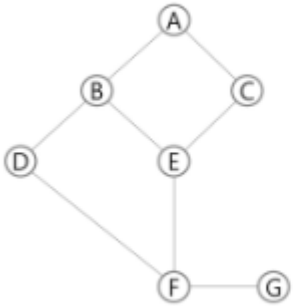
1. 시작 정점 v를 결정하여 방문한다.
2. 정점 v에 인접한 정점 중에서
    a. 방문하지 않은 정점 w가 있으면, v를 push, w 방문, w를 v로 하여 2)를 반복
    b. 방문하지 않은 정점이 없으면, pop하고 가장 마지막 방문 정점을 v로 하여 2) 반복
3. 스택이 공백이 될때까지 2)를 반복한다.

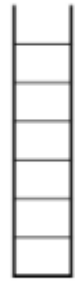
- 코드

```
visited[], stack[] 초기화
DFS(v)
    시작점 v 방문;
    visited[v] <- true;
    while {
        if ( v의 인접 정점 중 방문 안 한 정점 w가 있으면)
            push(v);
            v <- w;  (w에 방문)
            visited[w] <- true;
        else
            if (스택이 비어 있지 않으면)
                v <- pop(stack);
            else
                break
    }
end DFS()
```

- 예시

초기상태 : 배열 visited를 False로 초기화하고, 공백 스택을 생성



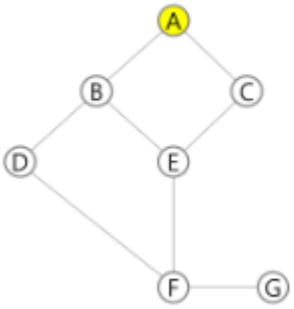


정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	F	F	F	F	F	F	F

⇒ 정점 A를 시작으로 깊이 우선 탐색을 시작한다.

1) 정점 A를 시작으로 깊이 우선 탐색을 시작

```
A 방문;
 visited[A] ←true;
```

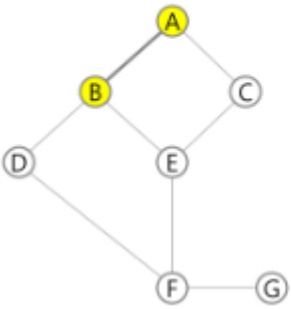


정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	F	F	F	F	F	F

⇒ 인접정점 중 오름차순에 따라 탐색하여 push 한다.

2) 정점 A에 방문하지 않은 정점 B, C가 있으므로 A를 스택에 push 하고, 인접정점 B와 C 중에서 오름차순에 따라 B를 선택하여 탐색을 계속한다.

```
push(A);
B 방문;
visited[B] ←true;
```

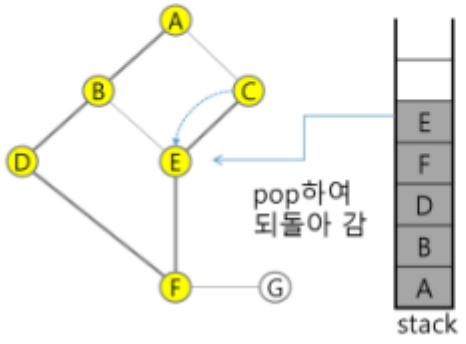


정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	F	F	F	F	F

⇒ 해당 순서대로 D, F, C를 방문한다.

7) 정점 C에서 방문하지 않은 인접정점이 없으므로, 마지막 정점으로 돌아가기 위해 스택을 pop 하여 받은 정점 E에 대해서 방문하지 않은 인접정점이 있는지 확인한다.

```
pop(stack);
```



E의 인접 정점

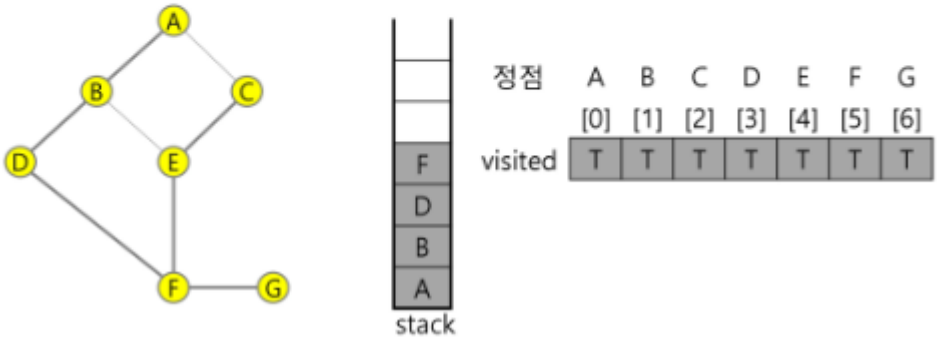
정점	A	B	C	D	E	F	G
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
visited	T	T	T	T	T	T	F

⇒ 방문안한 정점이 없으니까 pop 해서 e에 방문한 상태 ⇒ F로 방문한 상태로 만든다.

그 이후 G를 만난다.

9) 정점 F에 방문하지 않은 정점 G가 있으므로 F를 스택에 push 하고, 인접정점 G를 선택하여 탐색을 계속한다.

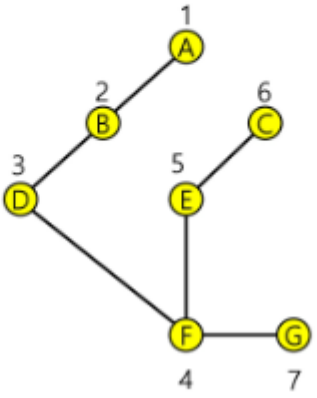
```
push(F);
G 방문;
visited[G] ←true;
```



이후 G에서도 없으니까, F → D → B → A 순으로 pop 하여 탐색을 마무리 한다.

14) 현재 정점 A에서 방문하지 않은 인접 정점이 없으므로 마지막 정점으로 돌아가기 위해 스택을 pop하는데, 스택이 공백이므로 깊이 우선 탐색을 종료한다.

깊이 우선 탐색 경로
: A-B-D-F-E-C-G



- 코드

```
'''
7 8
1 2 1 3 2 4 2 5 4 6 5 6 6 7 3 7
```

```
'''
def dfs(i, V): # 시작 i, 마지막 V
    visited = [0] * (V+1)# visited, stack 생성 및 초기화
    stack = []
    visited[i] = 1
    print(i)     # 정점에서 할 일
    while True: # 탐색
        for w in adjl[i]:# 현재 방문한 정점에 인접하고 방문안한 정점(
            if visited[w] == 0:
                stack.append(i) # push(i), i를 지나서
                i = w           # w에 방문
                visited[w] = 1  # 방문해서 할 일
                print(i)
                break
        else:                   # i에 남은 인접 정점이 없으면
            if stack:# 스택이 비어있지 않으면(지나온 정점이 남아 있으면
                i = stack.pop()
            else:    # 스택이 비어있으면(출발점에서 남은 정점이 없으면
                break

V, E = map(int,input().split())
arr  = list(map(int,input().split()))

# 인접리스트
adjl = [[] for _ in range(V+1)] # ardjl[i] 행에는 i에 인접인 정점
for i in range(E):
    n1, n2 = arr[i*2], arr[i*2+1]
    adjl[n1].append(n2)
    adjl[n2].append(n1) # 방향이 없는 경우에만..

visited = [0] * (V+1)# visited, stack 생성 및 초기화
dfs(1)
```