DFS & BFS

BABO 4기 이론반

22기 분석 원종빈

CONTENTS

1. post-OT
2. 자료구조 기초
3. DFS/BFS

1. post-OT

과제 풀이에 대하여

- 부담감?
- 난이도?
- 과제 개수?

Git commit message 작성법

- type: feat, fix, docs, refactor, test, chore, comment, remove, rename
- subject: 로그인 기능 구현, contents.md 파일 수정, 주석 오타 수정, 레거시 파일 정리
- body(optional)
- footer(optional)

- 예시
  - feat: solved BOJ 1105
  - refactor: update train.py with batch processing
  - remove: deleted legacy train.sh files

```
feat: "로그인 기능 구현"

로그인 시 JWT 발급

Resolves: #111
Ref: #122
related to: #30, #50
```

출처: https://hyunjun.kr/21

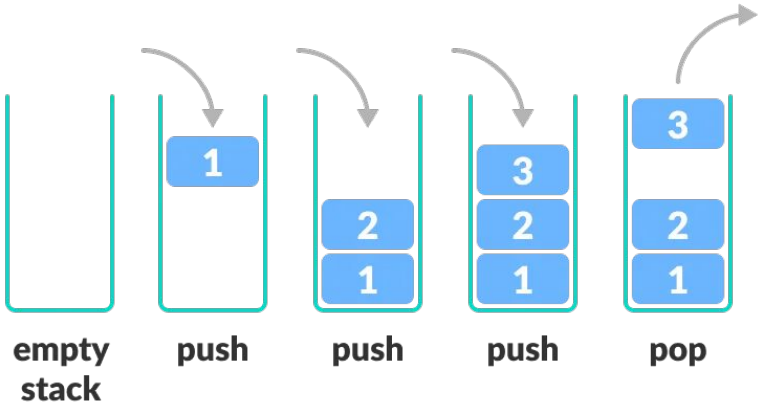
2. 기초 자료구조

스택(stack)

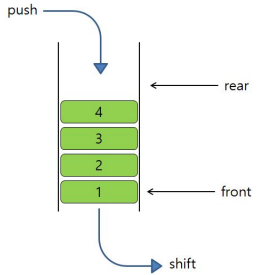
- LIFO(후입선출, Last-in First-out)
  - ctrl + Z가 대표적인 stack 자료구조의 활용
- 파이썬에서의 구현 → 리스트

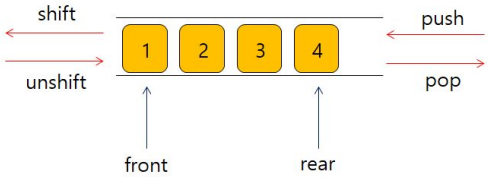
```
stack = []
stack.append(1) # [1]
stack.append(2) # [1, 2]
stack.append(3) # [1, 2, 3]
stack.pop()     # [1, 2]
```

- 스택 자료구조의 단점?



덱(deque, double-ended queue)





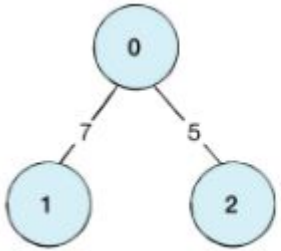
- 보통의 큐는 FIFO(선입선출, First-In First-Out).
  - 뒤에서는 push만, 앞에서는 pop만이 가능하다.
- 덱큐는 앞/뒤 모두에서 push와 pop이 가능하다!
- 파이썬에서의 구현 → `from collections import deque`

```
que = deque([1,2,3]) # 혹은 deque()도 가능
que.popleft()        # [2,3]
que.appendleft(10)   # [10, 2, 3]
que.append(4)        # [10, 2, 3, 4]
que.pop()            # [10, 2, 3]
```

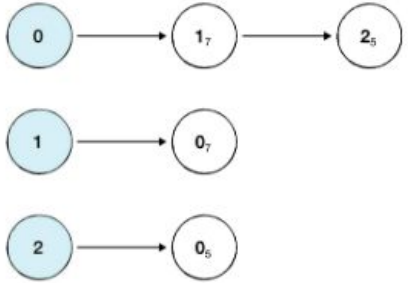
파이썬에서 덱과 큐의 구현은 같다.

`from queue import Queue` 사용 비추천

그래프(graph)



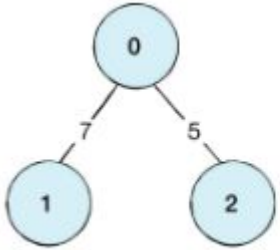
- 노드(node, vertex)와 간선(edge)로 이뤄진 자료구조
- 굉장히 일반화된 자료구조
- 다양한 변형 (단방향성, 양방향성, 비용 등)
- 응용: SNS, GNN...

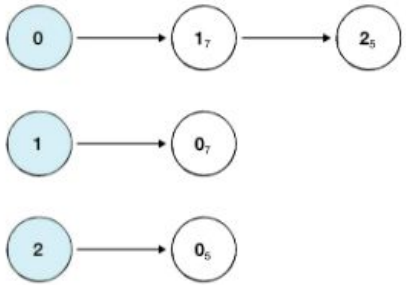


- 파이썬에서의 구현 → 인접 리스트 or 인접 행렬
  - 리스트는 연결 여부를 확인하는 데 시간 오래걸림
  - 행렬은 공간복잡도 측면에서 불리 O(V^2)

	0	1	2
0	0	7	5
1	7	0	무한
2	5	무한	0

그래프(graph)





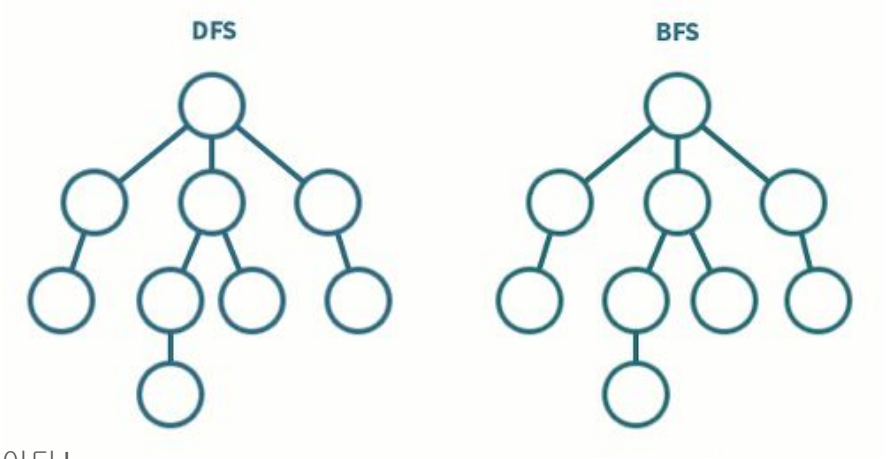
	0	1	2
0	0	7	5
1	7	0	무한
2	5	무한	0

```
# 인접 리스트 graph[from] = (to, cost)
graph = [
    [(1, 7), (2, 5)],
    [(0, 7)],
    [(0, 5)],
]
```

```
# 인접 행렬 graph[from][to] = cost
graph = [
    [0, 7, 5],
    [7, 0, float('inf')],
    [5, float('inf'), 0]
]
```

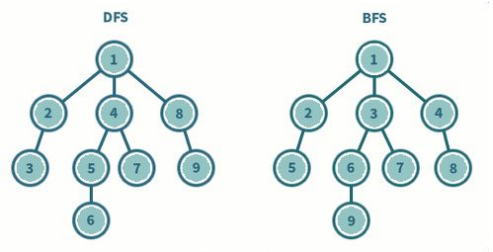
3. DFS & BFS

1. DFS & BFS 기초



- 공통점
  - **그래프를 끝까지 탐색하기 위한 알고리즘** 이다.
  - `to_visit`, `visited` 리스트를 관리하는 것이 핵심이다!

- 차이점
  - DFS는 **깊이를 우선 탐색** 하며, `to_visit`을 관리하고자 스택을 사용한다.
  - BFS는 **너비를 우선 탐색** 하며, `to_visit`을 관리하고자 큐(deque)를 사용한다.



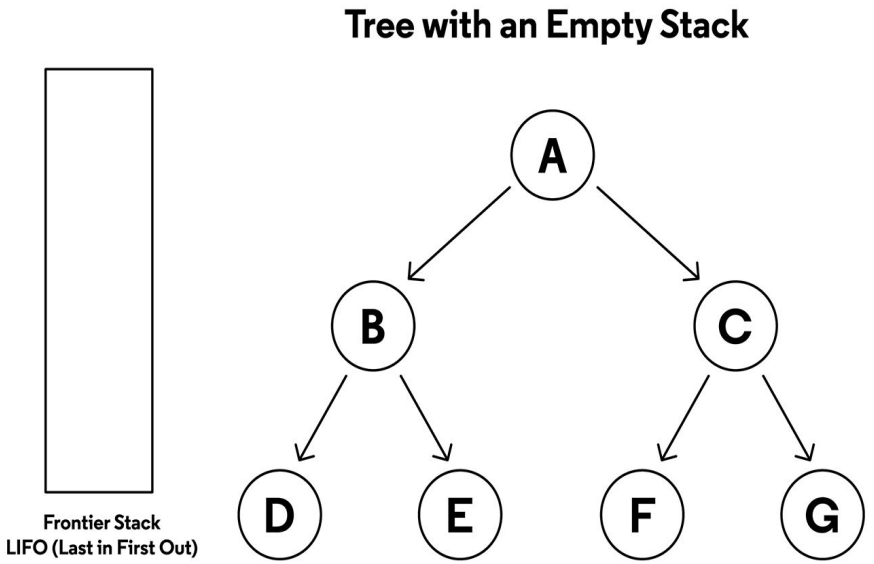
DFS vs BFS overview

```
def dfs(graph, start_node):
    to_visit, visited = list(), list()
    to_visit.append(start_node)

    while to_visit:
        node = to_visit.pop()

        if node not in visited:
            visited.append(node)
            to_visit.extend(graph[node])

    return visited
```

```
from collections import deque
def bfs(graph, start_node):
    to_visit, visited = deque(), []
    to_visit.append(start_node)


    while to_visit:
        node = to_visit.popleft()

        if node not in visited:
            visited.append(node)
            to_visit.extend(graph[node])

    return visited
```

DFS

```
graph = {
    'A': ['C', 'B'],
    'B': ['E', 'D'],
    'C': ['G', 'F'],
    'D': [],
    'E': [],
    'F': [],
    'G': [],
}

print(dfs(graph, 'A'))
# ['A', 'B', 'D', 'E', 'C', 'F', 'G']
print(bfs(graph, 'A'))
# ['A', 'C', 'B', 'G', 'F', 'E', 'D']
```



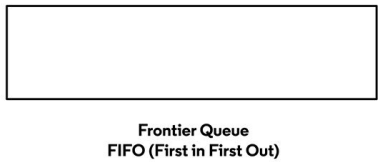
참고: 재귀로 구현한 DFS

```
def dfs_recursive(graph, start):
    visited = []
    visited.append(start)

    for node in graph[start]:
        if node not in visited:
            dfs_recursive(graph, node, visited)
    return visited
```

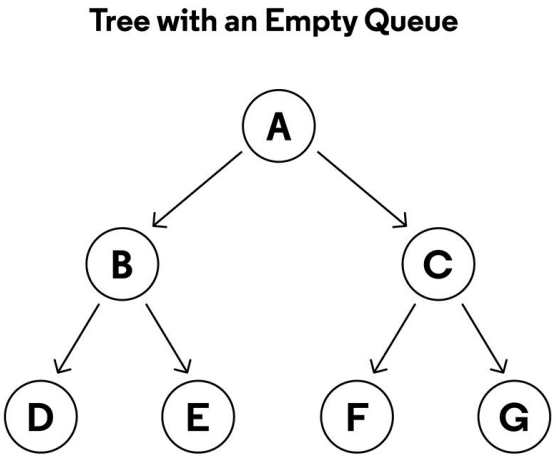
- 그러나, 코테에서 재귀를 사용하는 것은 추천하지 않는다! (출처)
- 이유: 함수를 여러번 부르는 과정에서 시간소요 + stack 메모리를 많이 차지해 에러 발생 가능성
  - `import sys; sys.setrecursionlimit(10 ** 6)` → 재귀 제한 해제 (출처)

BFS

```
graph = {
    'A': ['C', 'B'],
    'B': ['E', 'D'],
    'C': ['G', 'F'],
    'D': [],
    'E': [],
    'F': [],
    'G': [],
}

print(dfs(graph, 'A'))
# ['A', 'B', 'D', 'E', 'C', 'F', 'G']
print(bfs(graph, 'A'))
# ['A', 'C', 'B', 'G', 'F', 'E', 'D']
```



Frontier Queue
FIFO (First in First Out)



2. DFS & BFS 변형

- **지도**를 사용하는 경우
- 참고: 백준 1743 음식물 피하기
  - 상하좌우로 1이 연결된 최대 갯수?

1	0	0	0
0	1	1	0
1	1	0	0

```
def dfs(road, n, m):
    to_visit, visited = [], []
    to_visit.append((n, m))

    cnt = 0
    while to_visit:
        n, m = to_visit.pop()
        if (n, m) not in visited:
            visited.append((n, m))
            road[n][m] = 0
            cnt += 1
            for dn, dm in [(+1, 0), (-1, 0), (0, +1), (0, -1)]:
                if 0 <= (n + dn) < N and 0 <= (m + dm) < M \
                    and road[n + dn][m + dm] == 1:
                    to_visit.append((n + dn, m + dm))

    return cnt
```

2. DFS & BFS 변형

- global `visited`
- 참고: 백준 1743 음식물 피하기
  - 상하좌우로 1이 연결된 최대 갯수?

1	0	0	0
0	1	1	0
1	1	0	0

```
def dfs(road, n, m):
    to_visit = []
    to_visit.append((n, m))

    cnt = 0
    while to_visit:
        n, m = to_visit.pop()
        if road[n][m] == 1:
            road[n][m] = 0
            cnt += 1
            for dn, dm in [(+1, 0), (-1, 0), (0, +1), (0, -1)]:
                if 0 <= (n + dn) < N and 0 <= (m + dm) < M \
                   and road[n + dn][m + dm] == 1:
                    to_visit.append((n + dn, m + dm))

    return cnt
```

3. DFS & BFS 확장

- 지난 과제 1182 - 부분 수열의 합 역시 DFS로 풀 수 있다! (출처)
- 꼭 그래프가 아니더라도, 뭔가 연결짓거나 탐색해야하는 경우에는 DFS/BFS로 풀 수 있을지 고민해보자!

```
n_nums, target = map(int,input().split())
nums = list(map(int,input().split()))

cnt = 0

def dfs(idx,sum):
    global cnt
    if idx >= n_nums:
        return
    sum += nums[idx]
    if sum == target:
        cnt += 1

    dfs(idx+1, sum)
    dfs(idx+1, sum-nums[idx])

dfs(0,0)
print(cnt)
```

과제

- 백준 1260 - DFS와 BFS (S2)
- 백준 11724 - 연결 요소의 개수 (S2)
- 백준 1012 - 유기농 배추 (S2)

번외

- 백준 7576 - 토마토 (G5)
- 백준 1697 - 숨바꼭질 (S1)