디지털 교재

파이썬으로 시작하는

자료구조와 알고리즘

<5주차>

NAVER CONNECT

5 - 그래프 심화, 트리

수업 참고 자료

- DFS 코드 구현

```
# 1은 길이고, 0은 벽이다.
# 동서남북 4가지 방향과 대각선 4가지 방향 총 8가지로 1칸 이동할 수 있다.
# 0,0 좌표에서 dfs()

grid = [
   [1, 1, 1, 1],
   [0, 1, 0, 1],
   [0, 1, 0, 1],
   [1, 0, 1, 1],
]

def dfs(r,c):
    # 방문
    visited[r][c] = True
    print(f"{r},{c}")
```

```
    # 다음 노드 dfs 실행
    # 현재 노드의 좌표는 r,c ⇒ nr, nc
    for i in range(8):
        nr = r + dr[i]
        nc = c + dc[i]
        if 0 <= nr < row_len and 0 <= nc < col_len and grid[nr][nc] == 1:
            if not visited[nr][nc]:
                dfs(nr, nc)
    return

col_len = len(grid[0])
row_len = len(grid)
visited = [[False] * col_len for _ in range(row_len)]
dr = [0, 1, 1, 1, 0, -1, -1, -1]
dc = [1, 1, 0, -1, -1, -1, 0, 1]

dfs(0,0)
```

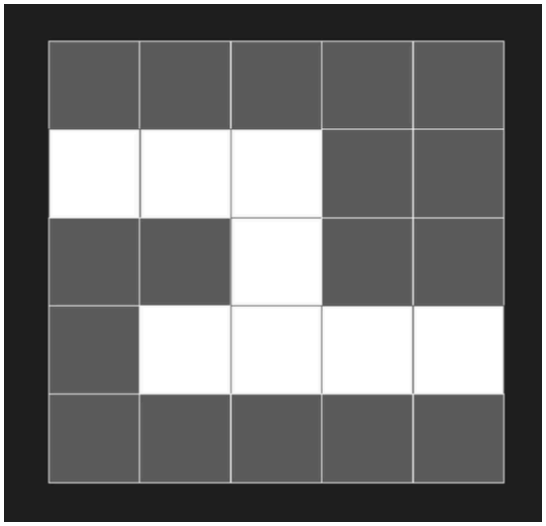
- bfs 코드 구현

```
grid = [
    [1, 1, 1, 1],
    [0, 1, 0, 1],
    [0, 1, 0, 1],
    [1, 0, 1, 1],
]
col_len = len(grid[0])
row_len = len(grid)
visited = [[False] * col_len for _ in range(row_len)]

dr = [0, 1, 1, 1, 0, -1, -1, -1]
dc = [1, 1, 0, -1, -1, -1, 0, 1]
def bfs(sr, sc):
    # 초기설정
    q = deque()
    # 시작점 예약
    q.append((sr, sc))
    visited[sr][sc] = True

    while q:
        # 방문
        r,c = q.popleft()
        print(r,c)
        # 예약
```

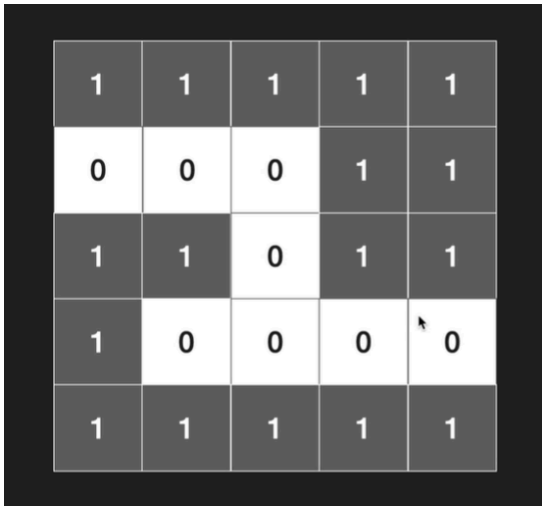
```
        for i in range(8):
            nr = r + dr[i]
            nc = c + dc[i]
            if 0 <= nr < row_len and 0 <= nc < col_len and grid[nr][nc] == 1:
                if not visited[nr][nc]:
                    q.append((nr,nc))
                    visited[nr][nc] = True

bfs(0, 0)
```

암시적 그래프

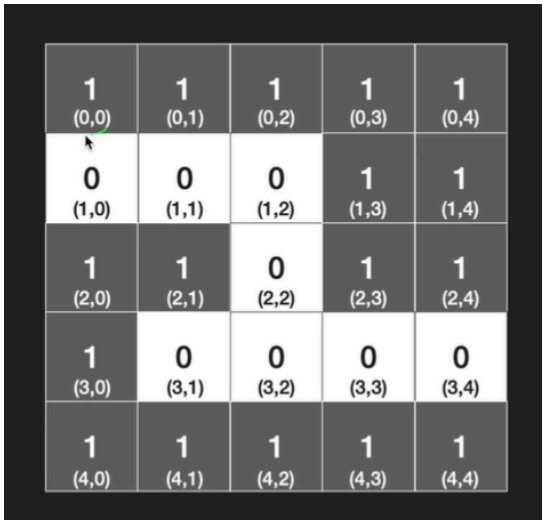
코딩 테스트에서 제일 많이 활용되는 **암시적 그래프**를 다루어 보도록 하겠습니다. 그래프 문제를 풀다 보면 다음과 같은 문제들을 자주 접해 볼 수 있습니다.



다음과 같이 흰색이 길이고, 검은색이 벽인 미로가 있다고 합시다. 전혀 그래프 같지 않지만, 암시적으로 그래프처럼 표현할 수 있습니다.



벽에는 `1`의 값을, 길에는 `0`의 값을 넣음으로써, 구분해 보도록 합시다. 그리고 각 영역을 좌표의 개념을 도입하여 표현할 수 있습니다.




```
graph = [
    [1, 1, 1, 1, 1],
    [0, 0, 0, 1, 1],
    [1, 1, 0, 1, 1],
    [1, 0, 0, 0, 0],
    [1, 1, 1, 1, 1],
]
```

위의 표현 방식 같은 경우, 연결 관계를 직접적으로 나타내지는 않았습니다.

하지만 상하좌우가 연결되어 있다는 것을 암시적으로 알 수 있습니다.

예를 들어 (1, 2)는 상(0, 2), 하(2, 2), 좌(1, 1), 우(1, 3)가 연결 되어있다는 것을 말이죠.

저희는 그래프의 값들을 `row`와 `col`이라는 변수명을 통해 값들에 접근할 것입니다.

	col 0	col 1	col 2	col 3	col 4
row 0	(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)
row 1	(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)
row 2	(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)
row 3	(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)
row 4	(4, 0)	(4, 1)	(4, 2)	(4, 3)	(4, 4)

가로를 `row`로 세로를 `col`로 생각하는 것입니다. `graph[row][col]`를 통해 원하는 값에 접근할 수 있게 됩니다. 즉, (2, 3)에 있는 값을 알고 싶으면, `graph[2][3]`을 통해 알 수 있습니다.

BFS 템플릿 코드

```
from collections import deque

def bfs(grid):
    def isValid(r, c):
        return (
            (r >= 0 and r < row_len)
            and (c >= 0 and c < col_len)
            and grid[next_r][next_c] == 1
        )
    row_len, col_len = len(grid), len(grid[0])
    visited = [[False] * col_len for _ in range(row_len)]
    dr = [0, 1, 0, -1]
    dc = [1, 0, -1, 0]
```

```
    queue = deque()
    queue.append(0, 0)
    visited[0][0] = True
    while queue:
        cur_r, cur_c = queue.popleft()
        # *---------------------------------------*
        # 그래프를 방문하며 처리해야할 일을 여기서 한다
        # (예시)
      # if cur_v == value:
        #     << 현재 노드가 특정 값을 만족할 때 해야할 일 >>
        #     return cur_v
        # *---------------------------------------*
        for i in range(4):
            next_r = cur_r + dr[i]
            next_c = cur_c + dc[i]
            if isValid(next_r, next_c):
                if not visited[next_r][next_c]:
                    queue.append((next_r, next_c))
                    visited[next_r][next_c] = True
```

DFS 템플릿 코드

```
# DFS
grid = [[0,0,1],[1,0,1],[1,0,0],[0,1,0]]
row_len, col_len = len(grid), len(grid[0])
visited = [[False] * col_len for _ in range(row_len)]

def dfs(grid, r, c):
    # *---------------------------------------*
    # 그래프를 방문하며 처리해야할 일을 여기서 한다
    # (예시)
    # << 현재 노드가 특정 위치에 도달했을 때 해야할 일 >>
    # if r == row_len - 1 and c == col_len - 1:
    #     print('도착')
    # *---------------------------------------*
    dr = [0, 1, 0, -1]
    dc = [1, 0, -1, 0]
    visited[r][c] = True
    for i in range(4):
        next_r = r + dr[i]
        next_c = c + dc[i]
        if 0 <= next_r < row_len and 0 <= next_c < col_len:
            if grid[next_r][next_c] == 0 and not visited[next_r][next_c]:
```

```
            dfs(grid, next_r, next_c)

dfs(grid, 0,0)
```

BFS와 DFS 시간 복잡도

각각의 순회는 모든 정점(V)들을 탐색해야 하고 그러기 위해서는 정점에 연결된 edge(E)들을 모두 확인해 봐야 합니다. 따라서 **BFS와 DFS 시간 복잡도는 $O(V+E)$입니다.**

코딩테스트 활용

그래프 변환

앞서 설명했듯이 그래프를 표현할 수 있는 방법은 인접 행렬, 인접 리스트, 암시적 그래프가 있습니다. 하지만 코딩 테스트에서는 위와 같은 형태보다는 edge의 집합으로 그래프를 주는 경우가 많습니다. 우리는 이와 같은 edge set 그래프를 우리가 원하는 그래프 표현 방식으로 변환해야 할 필요가 있습니다.

다음과 같은 edge set 그래프를 인접 행렬, 인접 리스트로 변환하는 과정을 보여드리겠습니다.

```
# num of vertex
n = 6

# edge = [u, v]
edge_set = [[1, 2], [2, 6], [2, 4], [4, 3], [3, 2], [3, 5]]
```

이때 우리는 이 그래프가 무향그래프인지 방향그래프인지를 파악해야 합니다.

무향 그래프의 경우 변환 코드는 다음과 같습니다.

인접 행렬

```
graph = [[0 for _ in range(n)] for _ in range(n)]

for u, v in edge_set:
    graph[u][v] = 1
    graph[v][u] = 1
```

▼ 결과

```
[0, 1, 0, 0, 0, 0]
[1, 0, 1, 1, 0, 1]
[0, 1, 0, 1, 1, 0]
[0, 1, 1, 0, 0, 0]
[0, 0, 1, 0, 0, 0]
[0, 1, 0, 0, 0, 0]
```

인접 리스트

```
# dict version
graph = {}

for u, v in edge_set:
    if u not in graph:
        graph[u] = []
    graph[u].append(v)

    if v not in graph:
        graph[v] = []
    graph[v].append(u)
```

```
# defaultdict version
from collections import defaultdict
graph = defaultdict(list)

for u, v in edge_set:
    graph[u].append(v)
    graph[v].append(u)
```

▼ 결과

```
1 :  [2]
2 :  [1, 6, 4, 3]
6 :  [2]
4 :  [2, 3]
3 :  [4, 2, 5]
5 :  [3]
```

방향 그래프의 경우 변환 코드는 다음과 같습니다.

인접 행렬

```
graph = [[0 for _ in range(n)] for _ in range(n)]

for u, v in edge_set:
    graph[u][v] = 1
```

▼ 결과

```
[0, 1, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 1]
[0, 1, 0, 0, 1, 0]
[0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
```

인접 리스트

```
# dict version
graph = {}

for u, v in edge_set:
    if u not in graph:
        graph[u] = []
    graph[u].append(v)
```

```
# defaultdict version
from collections import defaultdict
graph = defaultdict(list)

for u, v in edge_set:
    graph[u].append(v)
```

▼ 결과

```
1 :  [2]
2 :  [6, 4]
4 :  [3]
3 :  [2, 5]
```

또한 edge set 그래프에 edge의 가중치가 포함된 경우가 있습니다. 주어진 그래프가 방향 가중치 그래프인 경우 변환 코드는 다음과 같습니다.

```
# num of vertex
n = 6

# edge = [u, v, w]
edge_set = [[1, 2, 4], [2, 6, 1], [2, 4, 2], [4, 3, 3], [3, 2, 2], [3, 5, 1]]
```

인접 행렬

```
graph = [[0 for _ in range(n)] for _ in range(n)]

for u, v, w in edge_set:
    graph[u][v] = w
```

▼ 결과

```
[0, 4, 0, 0, 0, 0]
[0, 0, 0, 2, 0, 1]
[0, 2, 0, 0, 1, 0]
[0, 0, 3, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0]
```

인접 리스트

```
# dict version
graph = {}

for u, v, w in edge_set:
    if u not in graph:
        graph[u] = []
    # 그래프가 변동되지 않는 경우 immutable 객체인 tuple을 사용해도 좋습니다.
    graph[u].append([v, w])
```

```
# defaultdict version
from collections import defaultdict
graph = defaultdict(list)

for u, v, w in edge_set:
    # 그래프가 변동되지 않는 경우 immutable 객체인 tuple을 사용해도 좋습니다.
    graph[u].append([v, w])
```

▼ 결과

```
1 :  [[2, 4]]
2 :  [[6, 1], [4, 2]]
4 :  [[3, 3]]
3 :  [[2, 2], [5, 1]]
```

visited

딕셔너리(또는 해시셋) 구현

파이썬의 딕셔너리(dictionary)를 통해 visited를 만들 수 있습니다. key값으로 문자, 문자열, 숫자 등의 다양한 형태의 데이터를 담을 수 있고, value값으로 `True` 혹은 `False`를 담을 수 있는 불리언 (boolean) 자료형을 지정하여 방문 여부를 체크할 수 있습니다. 아래 코드에서 보면 visited를 dictionary(또는 set)로 구현했기 때문에 방문 체크를 할 때 if ___ not in visited: 를 이용해서 시간복잡도 O(1)으로 구현할 수 있습니다.

```
def dfs(cur_v, visited):
    visited[cur_v] = True
    for next_v in graph[cur_v]:
        if next_v not in visited:
            dfs(next_v, visited)
```

```
def bfs(graph, start_v):
    q = deque()
    visited = {start_v: True}
    q.append(start_v)

    while q:
        cur_v = q.popleft()
        print(cur_v)
        for next_v in graph[cur_v]:
            if next_v not in visited:
                visited[next_v] = True
                q.append(next_v)
    return visited
```

리스트 구현

딕셔너리뿐만 아니라 리스트로도 visited를 만들 수 있습니다. 리스트는 내재적으로 순서를 고려한 자료형이고, 정수형 인덱스로 빠르게 접근할 수 있다는 것이 큰 특징입니다.

다만 dictionary와 set과는 다르게 방문여부 체크를 할 때 인덱스에 직접 접근해서 판별을 해야 합니다.

ex) if visited[u] == False: 이런식으로 O(1)의 시간복잡도로 체크할 수 있습니다.

```
# 상황 : 그래프 노드가 1 ~ 10으로 정의된 경우 (연속적으로 존재하는 정수형)/ 또는 그리드형 그래프
visited = [False for i in range(10)]
```

```
visited = [[False] * col_len for _ in range(row_len)]


# 방문하지 않은 노드인지 아닌지 검사하기
if visited[u] == False: # O(1)
if value not in visited: # O(n) → 이렇게 검사하면 안된다!
```

💡 **in 연산자에 따른 시간복잡도 차이**

결론적으로 visited를 만드는 방법은 상황에 맞게 선택하시면 됩니다. visited를 False로 구성해서 리스트로 만들었다면,
`if visited[u] == False:` 조건문을 사용했을 때 딕셔너리와 리스트에 대한 시간복잡도는 모두 $O(1)$입니다. 다만 `in` 연산자로 방문 여부를 확인할 때 시간 복잡도 측면에서 리스트가 좋지 않을 수 있습니다. 아래의 코드로 그 차이를 확인해 봅시다.

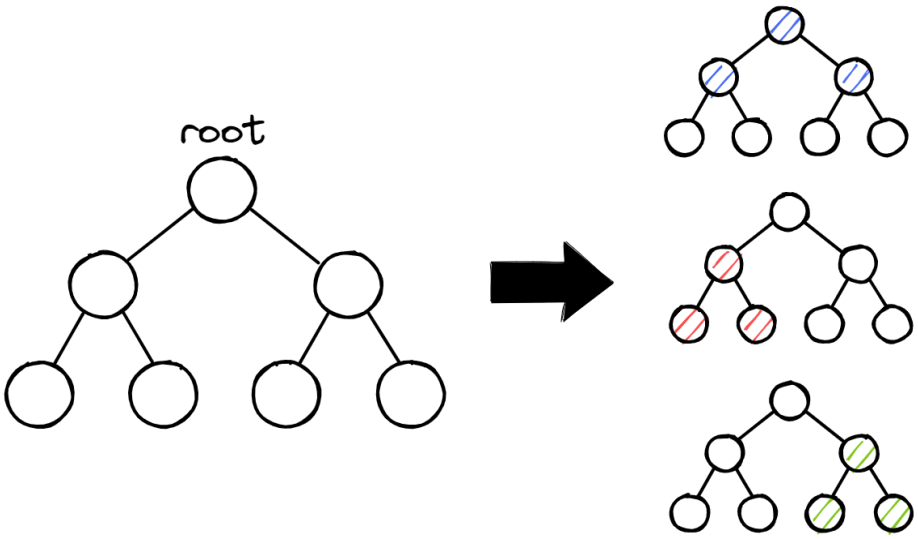
```
# 상황 : 그래프 노드가 1 ~ 10일 때, 현재 방문하는 노드는 4 (변수 u : 현재 방문하는 노드)
visited_d = {3:True, 1:True, 7:True, 2:True, 8:True}
visited_l = [3, 1, 7, 2, 8]
u = 4

'''
몇 번의 노드 방문 후
'''

if u not in visited_d: # 시간 복잡도 : O(1)
    ''' u가 방문하지 않은 노드일 때에 해야할 일 '''

if u not in visited_l: # 시간 복잡도 : O(n)
    ''' u가 방문하지 않은 노드일 때에 해야할 일 '''
```

방문한 노드만 저장하는 방식을 사용할 때, 저희는 `in` 연산자로 visited에 현재 방문하는 노드가 존재하는지를 검사할 것입니다. 이때, 딕셔너리는 시간복잡도가 $O(1)$이지만, 리스트는 처음부터 끝까지 확인해 보는 작업이 필요하기 때문에 $O(n)$의 시간복잡도가 발생합니다. 따라서 이 경우는 리스트보다 **딕셔너리**를 사용하는 것이 효율적입니다.

트리(Tree)

Tree는 서로 연결된 Node의 계층형 자료구조로써, root와 부모-자식 관계의 subtree로 구성되어 있습니다. 리스트가 단순히 순서를 매겨 데이터를 나열하는 선형 자료구조라면, 트리는 비선형적인 자료구조입니다.[1]

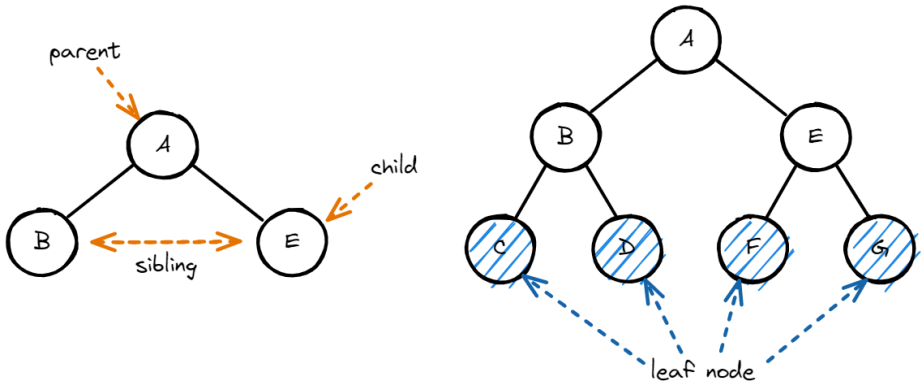
트리는 root에서 시작하여 여러 개의 tree가 중첩되는 형태로 만들어집니다. 그렇기에 하나의 tree안에 여러 개의 subtree가 존재합니다.



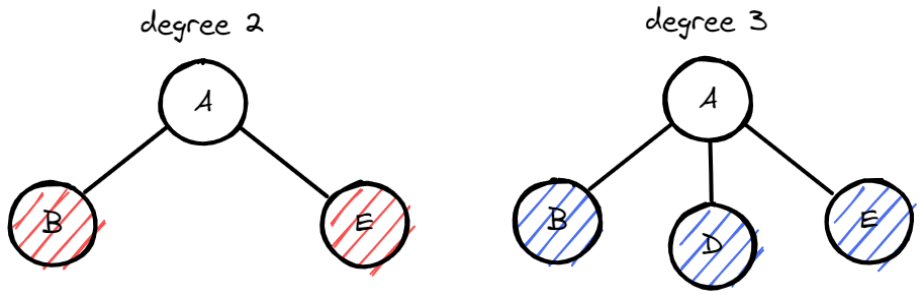
Tree 관련 개념

기본적인 트리 관련 용어를 정리해 보도록 하겠습니다.

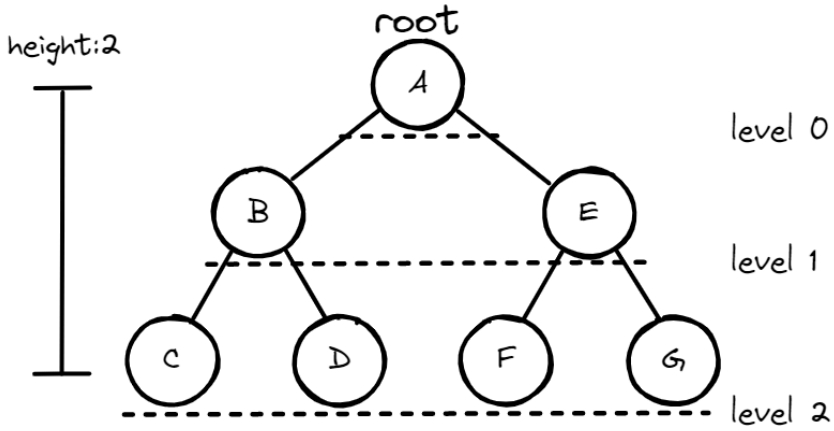
- 정점 (Vertex): A,B,C와 같은 값을 갖고 나타내며, 노드로 표현됩니다.
- 간선 (Edge): 정점 간에 연결된 선입니다.
- 자식 노드 (Child), 부모 노드 (Parent)
- 형제 노드(Sibling): 같은 부모를 가진 노드를 말합니다.
- 리프 노드 (Leef): 더 이상 뻗어나갈 수 없는 마지막 노드를 일컫습니다.



- 차수 (degree): 각 노드가 갖는 자식의 수. 모든 노드의 차수가 n개 이하인 트리를 n진 트리라고 합니다.



- 조상 (ancestor) 위쪽으로 간선을 따라가면 만나는 노드들 말합니다.
- 자손 (descendant): 아래쪽으로 간선을 따라가면 만나는 노드들을 말합니다.
- 루트 노드(Root): 트리의 시작 점입니다.
- 높이 (height): 루트 노드에서 가장 멀리 있는 리프 노드까지의 거리입니다.
- 레벨 (level): 루트 노드에서 떨어진 거리입니다.



트리 순회

선형 자료구조인 배열과 리스트의 경우, 순회하는 법이 인덱스 `0,1,2` 부터 `n-1` 까지로 방법이 하나로 정해져 있기 때문에, 순회하는 방법을 따로 배우지 않았습니다. `for` 문을 이용해서 쉽게 순회할 수 있기 때문이죠. 하지만 트리는 비선형 자료구조이기에 순회하는 방법이 여러가지 존재합니다. 대표적인 방법들로 level order traversal, preorder traversal, inorder traversal, postorder traversal이 있습니다.

순회에 대해 설명을 하기 전에 방문(visit)한다는 것과 순회(traversal)한다는 것에 대한 용어 정의가 필요합니다.

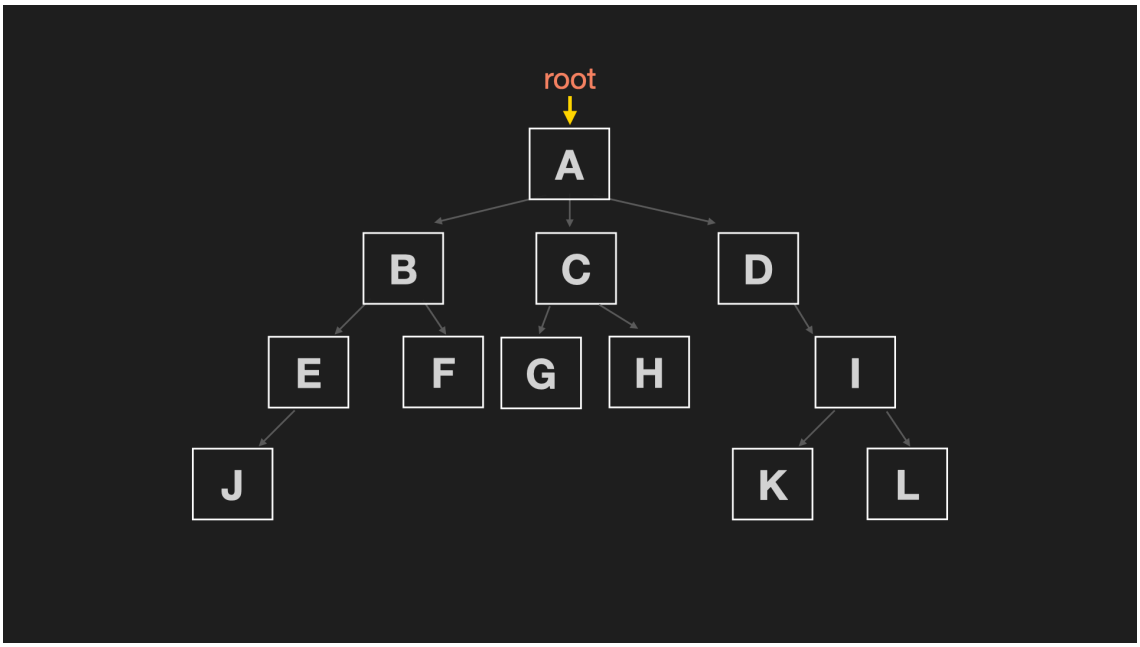
순회하다 (巡廻하다)
[순회하다/순훼하다]
동사
여러 곳을 돌아다니다.
전국을 순회하다.

방문하다 (訪問하다)
[방:문하다]
동사
어떤 사람이나 장소를 찾아가서 만나거나 보다.
오후에 고객의 사무실에 방문하기로 약속했다.

비슷하지만 약간의 뉘앙스가 다릅니다. 트리에서의 순회는 트리를 돌아다니는 것이고, 트리에서의 방문은 노드의 값에 접근하는 것(출력, 저장 등의 행위)입니다.

Level-order traversal

저희는 앞서서 level에 대해 정의 해보았습니다. level은 root 노드에서 떨어진 거리를 말합니다.



다음과 같이 root가 A인 tree가 있다고 합시다.

level 0	A
level 1	B C D
level 2	E F G H I
level 3	J K L

level order traversal은 말 그대로 level 별로 순회하는 것을 말합니다.

(A) →(B, C, D) → (E, F, G, H, I) → (J, K, L)

level order traversal 구현

level order traversal의 탐색 방법을 알았으니, 코드를 구현해 볼 차례입니다.

> 💡 level order의 구현 코드는 아주 기본적인 템플릿으로, 다음부터는 코드를 참고하지 않고도 직접 짤 수 있어야 합니다‼️

```
from collections import deque

def levelOrder(root):
    visited = []
    if root is None:
        return 0
    q = deque()
    q.append(root)
    while q:
        cur_node = q.popleft()
        visited.append(cur_node.value)

        if cur_node.left:
            q.append(cur_node.left)
        if cur_node.right:
            q.append(cur_node.right)
    return visited
```

`queue`를 사용함으로써, level order traversal를 구현해 보았습니다. `queue`를 `popleft`하고 `append`함으로써 각 노드를 방문하는 과정과 `queue`가 비게 되면, 완전 탐색이 끝난다는 것이 핵심입니다.

코딩 테스트에서는 위 과정들이 머릿속에서 그려지고, 코드들을 바로바로 짤 수 있어야 합니다.

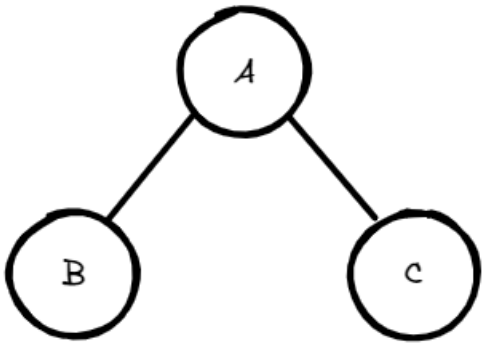
Level order traversal 시간 복잡도

`levelOrder`의 시간복잡도는 어려울 게 없습니다. 모든 정점의 개수가 `n`에 대해, `popleft`하고 `append`하는 과정이 `n`번 일어나므로 $O(n)$의 시간복잡도를 가집니다. 총 n개의 node를 탐방해야하므로 $O(n)$의 시간복잡도를 가진다고 생각해도 좋습니다.

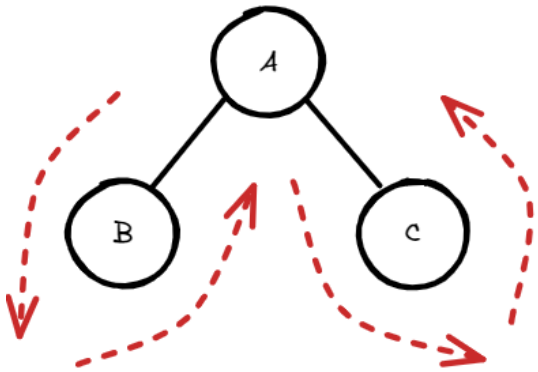
전위순회, 중위순회, 후위순회

전위 순회(preorder), 중위 순회(inorder), 후위 순회(postorder)의 구현은 앞서 배운 level order traversal에 비해 상대적으로 쉽습니다.

우선 순회하는 것을 먼저 보도록 하겠습니다.



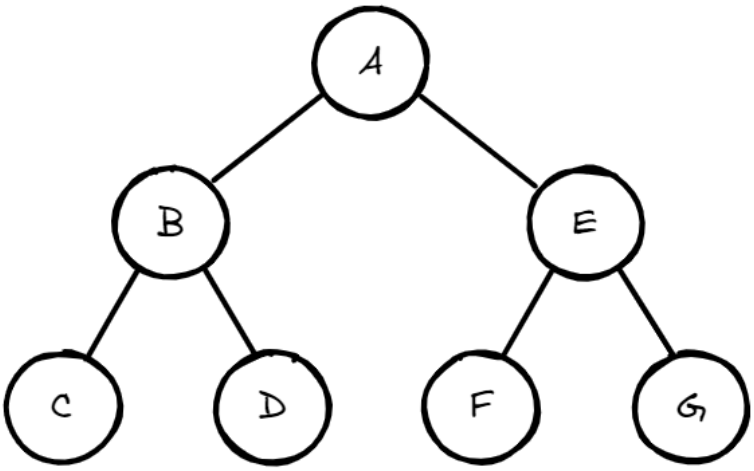
모든 트리는 기본적으로 다음과 같이 생겼습니다. 재귀적으로 정의 되어 있기 때문에 한 세트에서의 순회 과정을 정의하면 전체 트리를 순회할 수 있습니다.



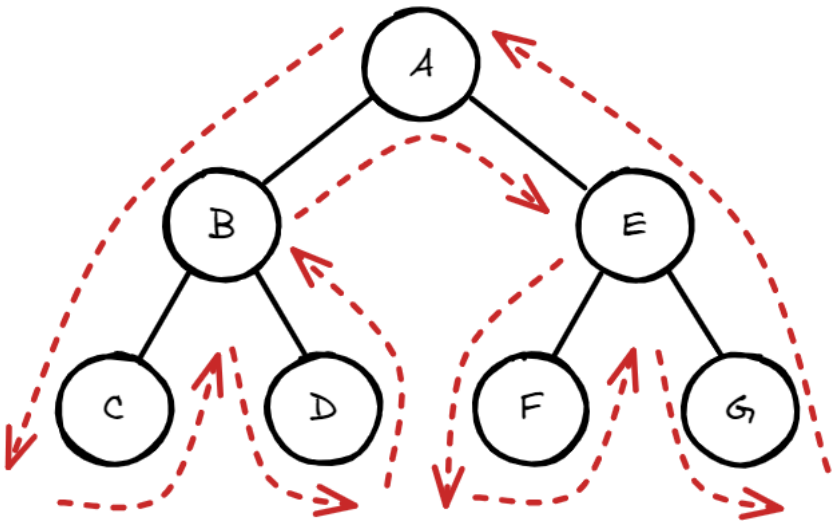
시작 노드에서 왼쪽 자식 노드, 다시 시작 노드를 거친 후 오른쪽 자식 노드를 찍고 마지막으로 시작 노드를 가는 것이 트리 순회의 과정입니다.

A→B→A→C→A

다른 예를 봐보도록 합시다.



어떤 식으로 순회할 수 있는지 한 번 생각 해보세요!




```
def traversal(root):
    if root is None:
        return
    traversal(root.left)
    traversal(root.right)
```

위와 같은 코드에서 언제 방문할지에 따라 전위, 중위, 후위의 코드가 결정됩니다.

각각의 코드를 구현해보도록 합시다.

전위 순회 구현

```
def preorder(root):
    if root is None:
        return
    print(root.value)
    preorder(root.left)
    preorder(root.right)
```

다음은 전위 순회 작동 순서를 보여주는 동영상입니다.

전위 순회 작동 동영상

https://prod-files-secure.s3.us-west-2.amazonaws.com/4ff76b2c-f77b-4aec-b3a7-852800e015b4/db80a89e-2c0e-4b32-b75c-4aaec4b19a6c/%E1%84%8C%E1%85%A5%E1%86%AB%E1%84%8B%E1%85%B1.m4v

중위 순회 구현

```
def inorder(root):
    if root is None:
        return
    inorder(root.left)
    print(root.value)
    inorder(root.right)
```

다음은 중위 순회 작동 순서를 보여주는 동영상입니다.

중위 순회 작동 동영상

https://prod-files-secure.s3.us-west-2.amazonaws.com/4ff76b2c-f77b-4aec-b3a7-852800e015b4/720bdd4d-a691-4f1b-a7fb-c358b6b09acf/%E1%84%8C%E1%85%AE%E1%86%BC%E1%84%8B%E1%85%B1.m4v

후위 순회 구현

```
def postorder(root):
    if root is None:
        return
    postorder(root.left)
    postorder(root.right)
    print(root.value)
```

다음은 후위 순회 작동 순서를 보여주는 동영상입니다.

후위 순회 작동 동영상

https://prod-files-secure.s3.us-west-2.amazonaws.com/4ff76b2c-f77b-4aec-b3a7-852800e015b4/bfb757db-1278-4503-a8e6-649d5f7bd0a8/%E1%84%92%E1%85%AE%E1%84%8B%E1%85%B1.m4v

전위, 중위, 후위 순회 시간복잡도

💡 재귀의 시간복잡도 = 재귀 함수 호출 수 x (재귀 함수 하나당) 시간복잡도

저희는 앞서서 재귀의 시간복잡도를 위와 같이 공부하였습니다. 전위, 중위, 후위 순회의 시간 복잡도도 마찬가지로 구할 수 있습니다.

- 재귀 함수 호출 수 : n
- 재귀 함수 하나당 시간 복잡도: $O(1)$

⇒ 전위, 중위, 후위 순회의 시간 복잡도는 $O(n)$입니다.

코딩테스트 활용

코딩테스트에서 트리 문제를 해결하기 위해서는, 별다른 팁이 존재한다기보다, **기본적인 트리의 구조**를 잘 알고 있어야 합니다. 트리 자료구조를 활용하는 문제보다 트리의 개념을 활용한 문제가 주로 나오기 때문이죠.

대표적인 예시로 백준의 후위 표기식 문제가 있습니다. 해당 문제는 중위 표기식을 후위 표기식으로 변환하는 문제로 트리의 순회 방법인 중위, 후위 순회의 개념이 사용되었지만, 실제로 푸는 방식은 스택 자료구조를 사용하여 해결하는 것이 대표적입니다. 이와 같이 트리의 개념과 추가로 전위 순회, 중위 순회, 후위 순회, 이 3가지 방식에 대해서도 이해하고 코드로 구현할 줄 아신다면, 트리를 활용한 문제를 잘 해결할 수 있을 것입니다.

구현 방법

```
    0
   / \
  1   2
      / \
     3   4
```

1. **인접리스트**

```
tree = [
    [1, 2],  # 노드 0의 자식들: 1, 2
    [],      # 노드 1의 자식들 없음
    [3, 4],  # 노드 2의 자식들: 3, 4
    [],      # 노드 3의 자식들 없음
    []       # 노드 4의 자식들 없음
]
```

2. **간선 리스트 형식 (Edge List)**

```
edges = [
    [0, 1], [0, 2], [2, 3], [2, 4]
]
```

3. **레벨 순서 배열 (완전이진트리, heap)**

```
tree = [0, 1, 2, None, None, 3, 4]  # None은 비어 있는 자리를 표시
```

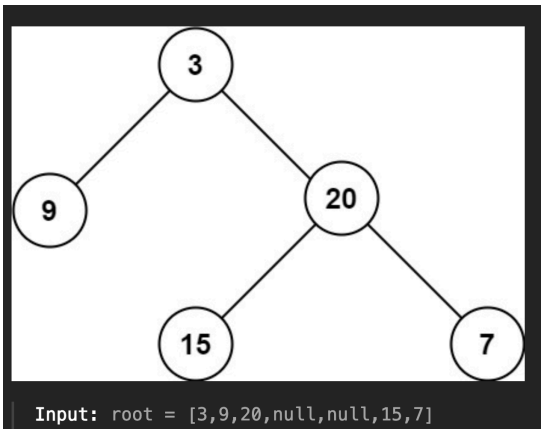
4. **클래스 기반 트리**

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []  # 자식 노드를 리스트로 저장

# 트리 생성
root = TreeNode(0)  # 루트 노드 생성
```

```
# 첫 번째 레벨 노드 추가
node1 = TreeNode(1)
node2 = TreeNode(2)
root.children.append(node1)
root.children.append(node2)

# 두 번째 레벨 노드 추가
node3 = TreeNode(3)
node4 = TreeNode(4)
node2.children.append(node3)
node2.children.append(node4)
```

```
class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

# 트리 생성
root = TreeNode(0)  # 루트 노드 생성

# 첫 번째 레벨 노드 추가
root.left = TreeNode(1)
root.right = TreeNode(2)

# 두 번째 레벨 노드 추가
root.right.left = TreeNode(3)
root.right.right = TreeNode(4)
```

5. **Leetcode**

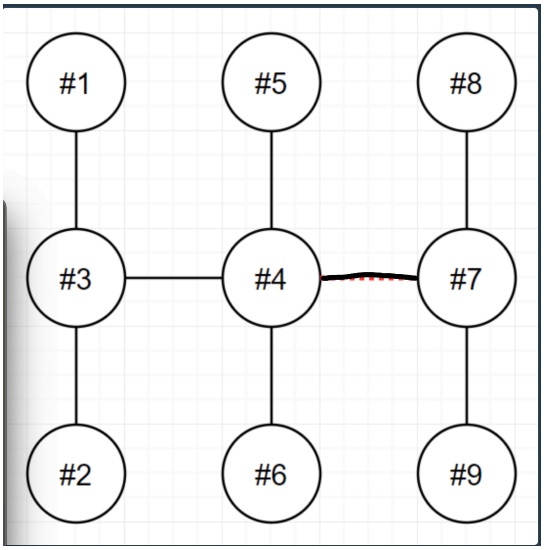


https://leetcode.com/problems/maximum-depth-of-binary-tree/description/

```
TreeNode{
   val: 3,
   left: TreeNode{
      val: 9,
      left: None,
      right: None
      },
   right: TreeNode{
      val: 20,
      left: TreeNode{
         val: 15,
         left: None,
         right: None
         },
      right: TreeNode{
         val: 7,
         left: None,
         right: None
         }
      }
   }
```

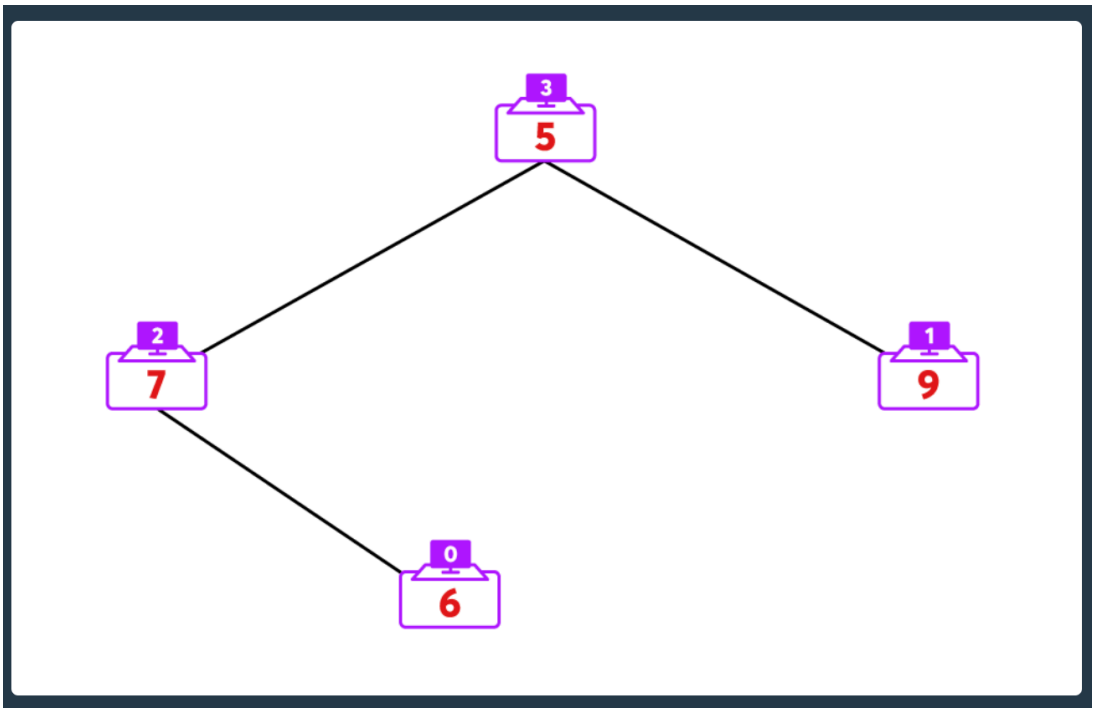
6. **프로그래머스**

```
wires = [[1,3],[2,3],[3,4],[4,5],[4,6],[4,7],[7,8],[7,9]]
```



https://school.programmers.co.kr/learn/courses/30/lessons/86971

```
links = [[-1, -1], [-1, -1], [-1, 0], [2, 1]]
```



https://school.programmers.co.kr/learn/courses/30/lessons/81305

1. 선형 자료구조 vs 비선형 자료구조

- 선형 자료구조 : 하나의 자료 뒤에 **하나의 자료**가 존재하는 것
- 비선형 자료구조 : 하나의 자료 뒤에 **여러개의 자료**가 존재하는 것