

알고리즘2 (2024-2)

4. 트리 자료구조의 활용

국립금오공과대학교 컴퓨터공학과

김 경 수

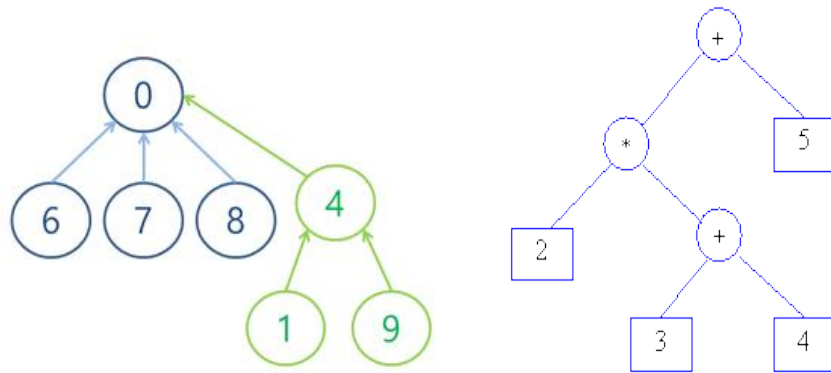
학습 목표

- ① 대표적 비선형 자료구조인 트리의 기본 개념을 복습하고 이진 탐색 트리의 삽입과 검색 연산을 즉석에서 구현할 수 있다.
- ② 코딩 테스트에서 트리가 사용되는 문제의 특성을 이해하고 왜 트리를 사용해야 하는가에 대하여 논리적으로 설명할 수 있다.
- ③ 주어진 문제를 해결하기 위해 트리를 이용하여 자료구조를 스스로 설계하고 이를 실제로 구현할 수 있다.
- ④ 실제 트리 자료구조가 활용되는 코딩 테스트 문제를 제한된 시간 내에 풀어봄으로써 코딩 테스트에서 트리를 활용하여 자료구조를 설계하는 방법과 문제를 해결하는 알고리즘을 빠르게 구현하는 방법을 숙달할 수 있다.

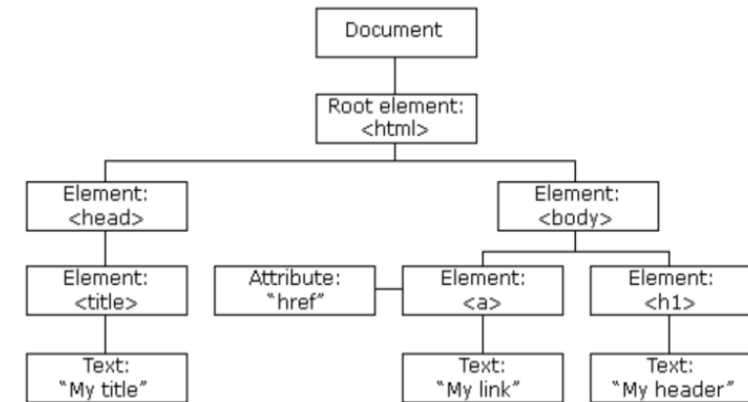
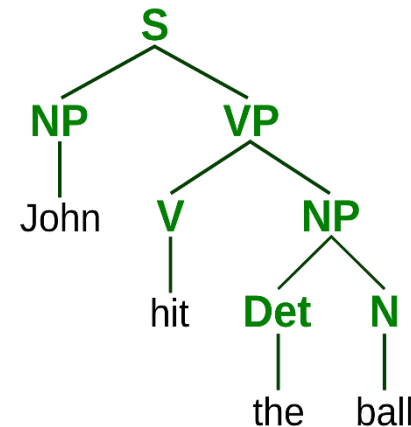
트리 자료구조 리뷰

트리(Tree)

- 트리(Tree)는 계층적 형태를 갖는 여러 개의 데이터를 다룰 때 사용하는 비선형 자료 구조이다.
- 자료의 구성 형태가 계층적으로 조직되어 있는 경우 주로 활용된다.
- 트리의 예



$2 * (3 + 4) + 5$

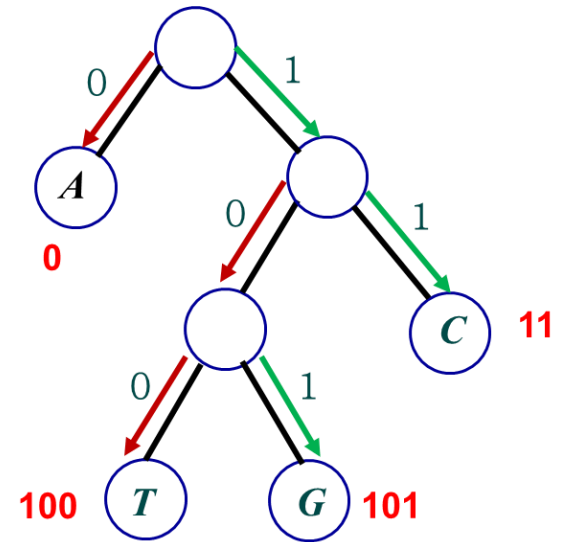
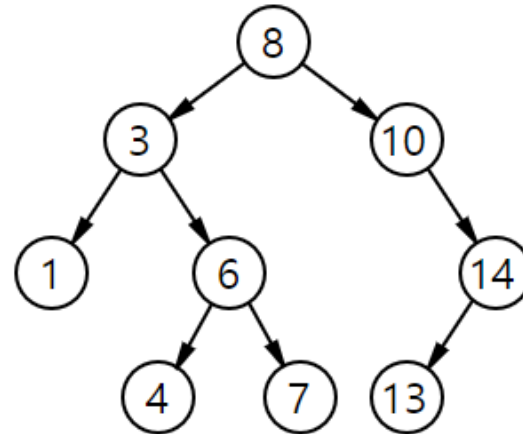
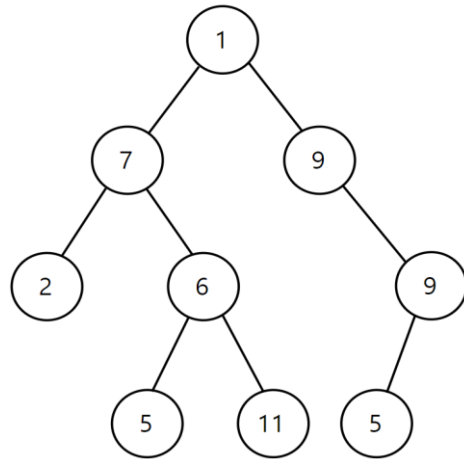


• 트리를 활용하는 이유

- ✓ 계층적 형태를 갖는 자료 또는 논리를 표현하거나 탐색을 위한 상태 공간을 표현할 때
- ✓ (예) 결정 트리(Decision tree), 파스 트리(Parsed tree), 상태 공간 트리(State space tree)

이진 트리(Binary Tree)

- 트리에서 모든 노드가 최대 두 개의 자식 노드를 갖는 트리를 **이진 트리 (Binary tree)**라 한다.
- 이진 트리의 예**



이진 트리의 주 활용 용도

- ✓ 이진 트리는 주로 대량의 데이터를 인덱싱(indexing)하는 용도로 널리 활용된다.
- ∴ **대량의 데이터를 빠른 속도로 검색(search) 또는 정렬(sort)할 수 있음.**

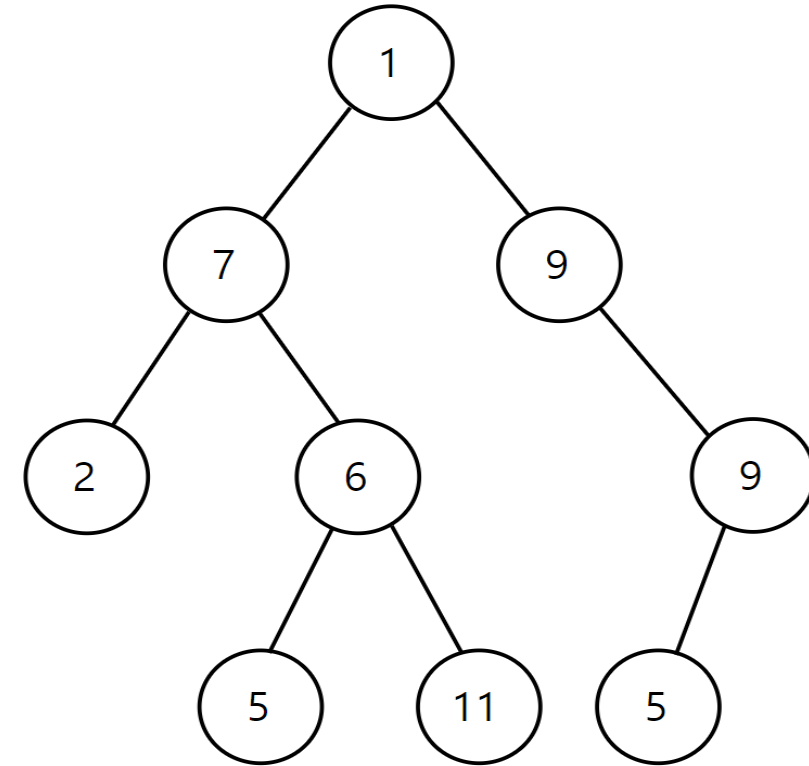
이진 트리(Binary Trees: BT)

• 이진 트리의 정의

- 공백이거나 루트와 왼쪽 서브 트리, 오른쪽 서브 트리라고 하는 2개의 분리된 이진 트리로 구성된 노드의 유한 집합

• 이진 트리의 속성

- n 개 노드를 갖는 이진 트리는 $n - 1$ 개의 간선을 갖는다.
- 노드의 개수가 n 개일 때 이진 트리의 높이 h 는 최소 $\lceil \log(n + 1) \rceil$, 최대 n 이다.
- 레벨 i 에서의 최대 노드 수는 2^{i-1} 이다.
- 높이가 h 인 이진 트리의 최대 노드 수는 최소 h 개, 최대 $2^h - 1$ 이다.

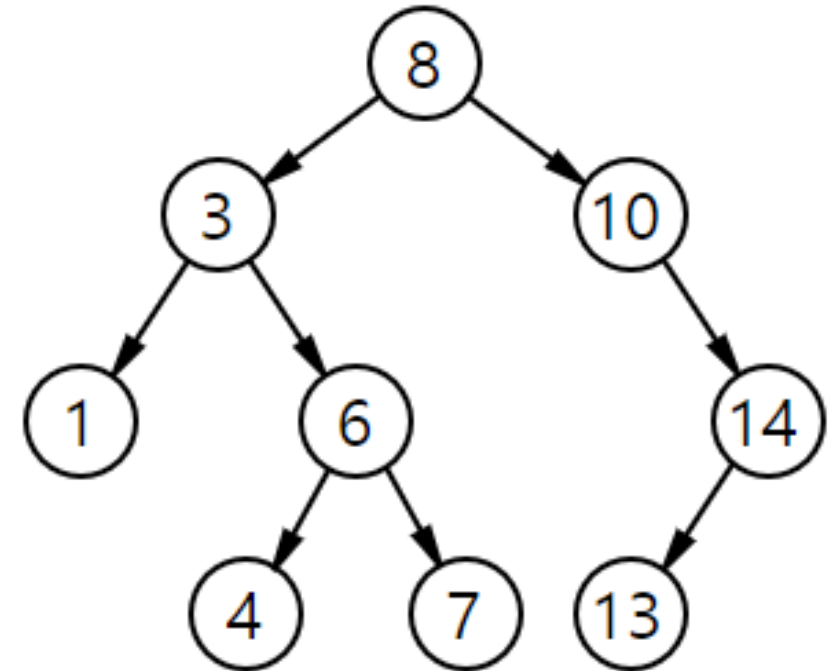


이진 탐색 트리(Binary Search Trees: BST)

• 이진 탐색 트리의 정의

이진 트리의 기본 속성을 만족하면서, 아래의 속성을 모두 만족하면 이진 탐색 트리이다.

- ① 모든 노드는 고유의 키를 가지며, 각각의 키는 상이함.
- ② 왼쪽 서브 트리에 포함된 키들은 그 루트 노드의 키보다 작다.
- ③ 오른쪽 서브 트리에 포함된 키들은 그 루트 노드의 키보다 크다.
- ④ 왼쪽과 오른쪽 서브 트리도 모두 이진 탐색 트리이다.



코딩 테스트에서 트리를 활용하는 경우

- ① 문제를 유한 개의 상태(state)와 유한 개의 취할 수 있는 행동(action)으로 모델링할 수 있는 경우
- ② 깊이 우선 탐색(DFS) 또는 너비 우선 탐색(BFS)을 수행해야 하는 경우
- ③ 계층적(Hierarchical) 형태의 자료 구조로 모델링할 수 있는 경우
- ④ 분할 정복 알고리즘 또는 동적 계획법을 사용하는 경우
- ⑤ 주어진 입력의 크기가 n 일 때 시간 복잡도를 $O(n) \rightarrow O(\log n)$ 또는 $O(n^2) \rightarrow O(n \log n)$ 으로 감소시켜야 하는 경우
- ⑥ 분리 집합(disjoint set)을 활용해야 하는 경우

상태 공간
탐색

이진 탐색
효과

[연습] 이진 탐색 트리의 주요 연산 구현

- 아래 코드는 이진 탐색 트리의 주요 연산을 구현한 함수의 일부분이다.
- 아래 코드의 함수에서 **pass**로 표시된 미구현된 부분을 모두 완성하시오.

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
```

```
class BST:
    def __init__(self): # BST 초기화
        self.root = None

    def insert(self, key): # 키 삽입
        pass

    def search(self, key): # 키 탐색
        pass
```

[연습] 이진 탐색 트리의 주요 연산 구현

insert, search 함수 구현 결과 테스트

```
lst = [5, 3, 8, 4, 2, 1, 7, 10]
```

```
search_lst = [1, 2, 5, 6]
```

```
bst = BST()
```

리스트의 각 요소를 이용하여 이진 탐색 트리 생성

```
for key in lst:
```

```
    bst.insert(key)
```

```
result = []
```

검색 리스트의 각 요소를 이진 탐색 트리에서 검색하고, 검색 결과를 리스트에 추가

```
for search_val in search_lst:
```

```
    if bst.search(search_val):
```

```
        result.append(True)
```

```
    else:
```

```
        result.append(False)
```

```
print(result)
```

[연습] 이진 탐색 트리의 주요 연산 구현 결과

키 삽입 함수

```
def insert(self, key):  
    if not self.root:  
        self.root = Node(key)  
    else:  
        curr = self.root  
        while True:  
            if key < curr.val:  
                if curr.left:  
                    curr = curr.left  
                else:  
                    curr.left = Node(key)  
                    break  
            else:  
                if curr.right:  
                    curr = curr.right  
                else:  
                    curr.right = Node(key)  
                    break
```

키 탐색 함수

```
def search(self, key):  
    curr = self.root  
  
    while curr and curr.val != key:  
        if key < curr.val:  
            curr = curr.left  
        else:  
            curr = curr.right  
  
    return curr
```

연습문제 1. 예상 대진표

문제 1. 예상 대진표 (1/3)

제한시간: 30분

- △△ 게임대회가 개최되었습니다.
- 이 대회는 N명이 참가하고, 토너먼트 형식으로 진행되며, N명의 참가자는 각각 1부터 N번을 차례대로 배정받습니다.
- 그리고, 1번↔2번, 3번↔4번, ... , N-1번↔N번의 참가자끼리 게임을 진행합니다.
- 각 게임에서 이긴 사람은 다음 라운드에 진출할 수 있습니다. 이때, 다음 라운드에 진출할 참가자의 번호는 다시 1번부터 N/2번을 차례대로 배정받습니다.
- 만약 1번↔2번 끼리 겨루는 게임에서 2번이 승리했다면 다음 라운드에서 1번을 부여받고, 3번↔4번에서 겨루는 게임에서 3번이 승리했다면 다음 라운드에서 2번을 부여받게 됩니다.
- 게임은 최종 한 명이 남을 때까지 진행됩니다.

문제 1. 예상 대진표 (2/3)

제한시간: 30분

- 이때, 처음 라운드에서 A번을 가진 참가자는 경쟁자로 생각하는 B번 참가자와 몇 번째 라운드에서 만나는지 궁금해졌습니다.
- 게임 참가자 수 N, 참가자 번호 A, 경쟁자 번호 B가 함수 solution의 매개변수로 주어질 때, 처음 라운드에서 A번을 가진 참가자는 경쟁자로 생각하는 B번 참가자와 몇 번째 라운드에서 만나는지 return 하는 **solution1** 함수를 완성해 주세요.
- 단, A번 참가자와 B번 참가자는 서로 붙게 되기 전까지 항상 이긴다고 가정합니다.
- **제한사항**
 - N : 2^1 이상 2^{20} 이하인 자연수 (2의 지수 승으로 주어지므로 부전승은 발생하지 않습니다.)
 - A, B : N 이하인 자연수 (단, $A \neq B$ 입니다.)

문제 1. 예상 대진표 (3/3)

제한시간: 30분

• 입출력 예제

N	A	B	answer
8	4	7	3

• 입출력 예제 설명

- 첫 번째 라운드에서 4번 참가자는 3번 참가자와 붙게 되고, 7번 참가자는 8번 참가자와 붙게 됩니다.
- 항상 이긴다고 가정했으므로 4번 참가자는 다음 라운드에서 2번이 되고, 7번 참가자는 4번이 됩니다. 두 번째 라운드에서 2번은 1번과 붙게 되고, 4번은 3번과 붙게 됩니다.
- 항상 이긴다고 가정했으므로 2번은 다음 라운드에서 1번이 되고, 4번은 2번이 됩니다.
- 세 번째 라운드에서 1번과 2번으로 두 참가자가 붙게 되므로 3을 return 하면 됩니다.

문제 1. 문제 분석

• 문제의 목표

- N명이 참여하는 토너먼트에서 A번째 사람이 B번째 사람과 만나게 되는 라운드가 몇 번째 라운드인지 찾는 문제

• 문제에서 고려해야 할 사항

- 토너먼트 문제 → 전형적인 계층적 구조 → 트리 구조
- 토너먼트에 참여하는 사람의 수가 $2 \sim 2^{20}$ 명
 - 실제로 트리를 구현하면 공간 복잡도 문제로 통과하지 못할 수 있음
 - 참여하는 사람의 수는 짝수이므로 부전승은 존재하지 않음
- 트리에서 각 라운드 → 트리의 계층(레벨)

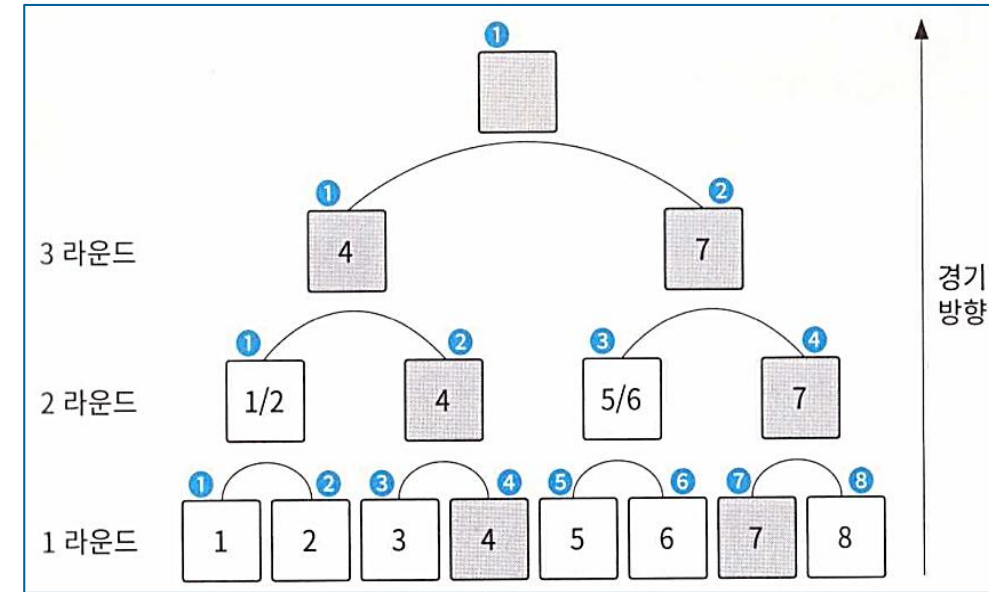
문제 1. 문제 해결 전략 수립

• 주어진 문제를 트리로 구조화

➤ 주어진 문제가 "트리"를 이용하는 문제임을 파악했으면, 즉시 문제의 내용을 트리로 그려서 표현해 본다.

➤ 트리에서 중점적으로 분석해야 할 부분

- ① 트리의 계층(또는 높이)
- ② 각 계층에서 노드의 인덱스(index)
- ③ 트리 내 각 노드의 값(value)
- ④ 자식 노드와 부모 노드의 관계가 의미하는 것은?



① 라운드(Round)

② 각 라운드에서 사용자에게 순차적으로
배정되는 고유 번호

③ 사용자

④ 부모 노드 = 두 자식 노드 중 승자

문제 1. 문제 해결 전략 수립

• 노드와 노드의 인덱스

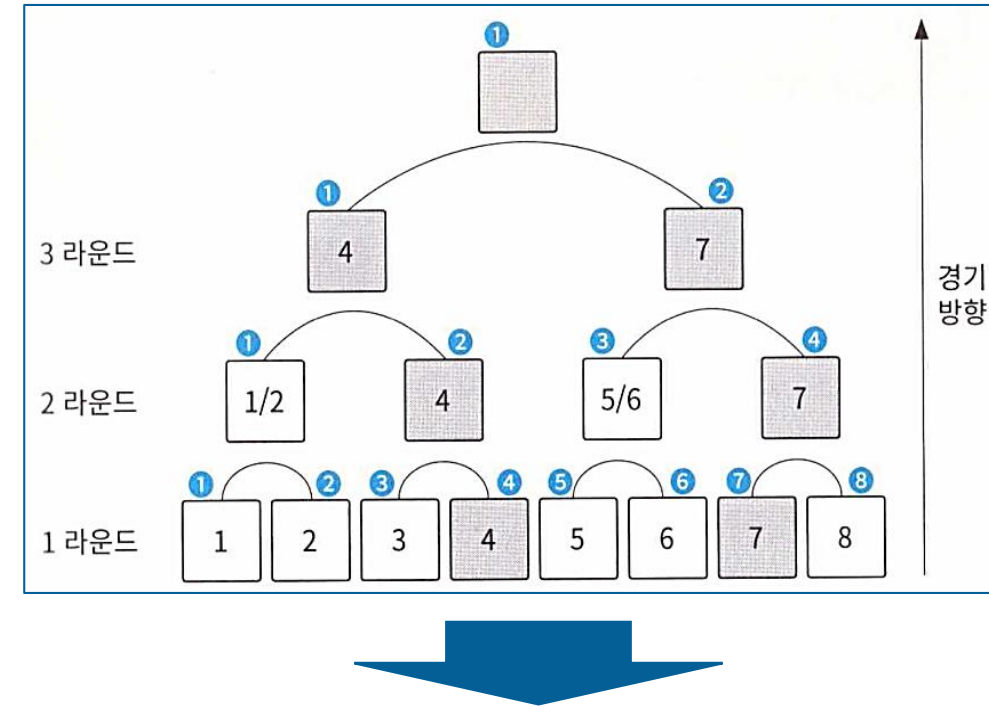
- 각 노드에 할당되는 최초 인덱스 = 해당 노드의 값
- 두 노드 중 승자 노드가 부모 노드가 됨

• 라운드가 진행됨에 따라 변하는 것은?

- 라운드 내 노드의 개수가 절반으로 줄어든다.
- 각 노드의 인덱스가 변한다.
 - 왼쪽 노드의 새로운 인덱스 \leftarrow "기존 인덱스 / 2"를 올림
 - 오른쪽 노드의 새로운 인덱스 \leftarrow "기존 인덱스 / 2"

• A와 B가 만나는 라운드는 어떻게 판단하는가?

- "A 노드의 인덱스 = B 노드의 인덱스"가 되는 라운드의 바로 직전 라운드
- 따라서, **A와 B노드의 인덱스의 변화**에 주목해야 한다.



- 라운드가 진행됨에 따라 **A와 B노드의 인덱스**가 어떻게 변화하는지 분석
- 두 노드의 인덱스가 같아질 때 까지 라운드의 수행 횟수를 카운트

문제 1. 문제 해결 방법 기술

- ① 노드 A와 B의 인덱스, Round의 수행 횟수를 초기화한다.
 - 노드 A의 인덱스 $\leftarrow A$
 - 노드 B의 인덱스 $\leftarrow B$
 - round $\leftarrow 0$
- ② 노드 A의 인덱스와 노드 B의 인덱스가 다른 경우, 아래의 작업을 반복한다.
 - 노드 A의 인덱스 $\leftarrow \text{ceil}(\text{노드 A의 인덱스} / 2)$ ※ **ceil(x): 올림 함수**
 - 노드 B의 인덱스 $\leftarrow \text{ceil}(\text{노드 B의 인덱스} / 2)$
 - round $\leftarrow \text{round} + 1$
- ③ 변수 round를 반환한다.

문제 1. 구현 및 검증

```
import math

def solution1(N, a, b):
    round = 0

    while a != b:
        a = math.ceil(a/2)
        b = math.ceil(b/2)
        round += 1

    return round
```

테스트 케이스

```
# Test
N = 8
A = 4
B = 7
print( solution1(N, A, B) )
```

연습문제 2. 미로 탈출

문제 2. 미로 탈출 (1/4)

제한시간: 80분

- 1 x 1 크기의 칸들로 이루어진 직사각형 격자 형태의 미로에서 탈출하려고 합니다.
- 각 칸은 통로 또는 벽으로 구성되어 있으며, 벽으로 된 칸은 지나갈 수 없고 통로로 된 칸으로만 이동할 수 있습니다.
- 통로들 중 한 칸에는 미로를 빠져나가는 문이 있는데, 이 문은 레버를 당겨서만 열 수 있습니다. 레버 또한 통로들 중 한 칸에 있습니다.
- 따라서, 출발 지점에서 먼저 레버가 있는 칸으로 이동하여 레버를 당긴 후 미로를 빠져나가는 문이 있는 칸으로 이동하면 됩니다.
- 이때 아직 레버를 당기지 않았더라도 출구가 있는 칸을 지나갈 수 있습니다. 미로에서 한 칸을 이동하는데 1초가 걸린다고 할 때, 최대한 빠르게 미로를 빠져나가는데 걸리는 시간을 구하려 합니다.
- 미로를 나타낸 문자열 배열 maps가 매개변수로 주어질 때, 미로를 탈출하는데 필요한 최소 시간을 return 하는 **solution2** 함수를 완성해주세요. 만약, 탈출할 수 없다면 -1을 return 해주세요.

문제 2. 미로 탈출 (2/4)

제한시간: 80분

• 제한사항

- $5 \leq \text{maps}$ 의 길이 ≤ 100
- $5 \leq \text{maps}[i]$ 의 길이 ≤ 100
- $\text{maps}[i]$ 는 다음 5개의 문자들로만 이루어져 있습니다.
- S : 시작 지점
- E : 출구
- L : 레버
- O : 통로
- X : 벽
- 시작 지점과 출구, 레버는 항상 다른 곳에 존재하며 한 개씩만 존재합니다.
- 출구는 레버가 당겨지지 않아도 지나갈 수 있으며, 모든 통로, 출구, 레버, 시작점은 여러 번 지나갈 수 있습니다.

문제 2. 미로 탈출 (3/4)


제한시간: 80분

입출력 예제

maps	result
["SOOOL","XXXXO","OOOOO","OXXXX","OOOOE"]	16
["LOOXs","OOOOX","OOOOO","OOOOO","EOOOO"]	-1

입출력 예제 #1 설명

- 주어진 문자열은 왼쪽과 같은 미로이며, 오른쪽과 같이 이동하면 가장 빠른 시간에 탈출할 수 있습니다.
- 4번 이동하여 레버를 당기고 출구까지 이동하면 총 16초의 시간이 걸립니다.
- 따라서 16을 반환합니다.

START	O	O	O		0	1	2	3	4
X	X	X	X	O	X	X	X	X	5
O	O	O	O	O	10	9	8	7	6
O	X	X	X	X	11	X	X	X	X
O	O	O	O	EXIT	12	13	14	15	16

문제 2. 미로 탈출 (4/4)


제한시간: 80분

입출력 예제

maps	result
["SOOOL","XXXXO","OOOOO","OXXXX","OOOOE"]	16
["LOOXS","OOOOX","OOOOO","OOOOO","EOOOO"]	-1

입출력 예제 #2 설명

- 주어진 문자열은 왼쪽과 같은 미로입니다.
- 시작 지점에서 이동할 수 있는 공간이 없어서 탈출할 수 없습니다.
- 따라서 -1을 반환합니다.

	O	O	X	START
O	O	O	O	X
O	O	O	O	O
O	O	O	O	O
EXIT	O	O	O	O


문제 2. 문제 분석

• 문제의 목표

- 미로를 탈출하는 데 필요한 최소 시간을 구한다. (※ 한 칸 이동 시 1초 소요)

• 문제에서 고려해야 할 사항

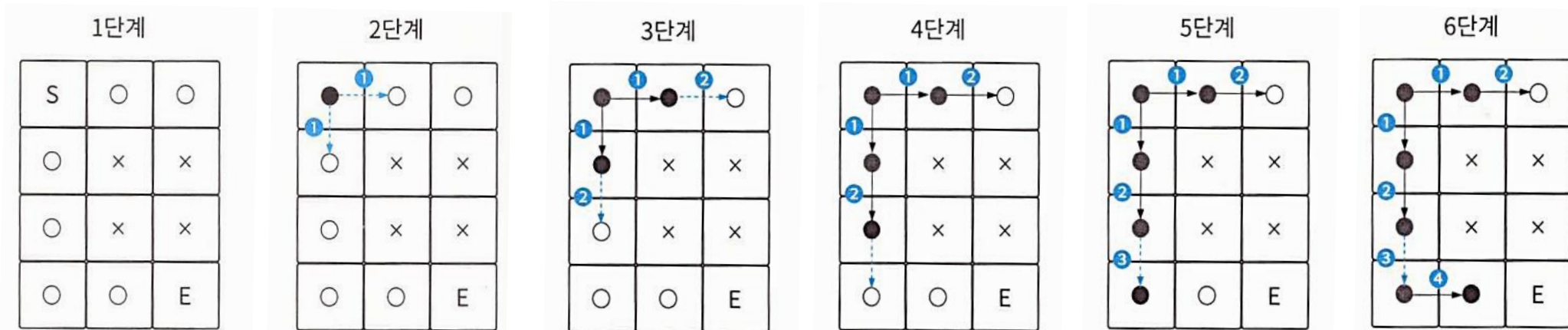
- 문제에서 제시된 키워드: **최소 시간, 최소 경로**
 - 최단 경로 알고리즘 vs 깊이 우선 탐색 vs 너비 우선 탐색 중 무엇을 사용해야 하는가?
- 레버의 존재와 출구(EXIT)와의 관계
 - 레버를 당겨야 출구에서 나갈 수 있음
 - 따라서 레버가 있는 칸으로 우선 이동 후 레버를 당겨야 함
 - 단, 레버를 당기지 않는 경우 출구를 지나갈 수는 있음(나가지는 X)

START	o	o	o	
x	x	x	x	o
o	o	o	o	o
o	x	x	x	x
o	o	o	o	EXIT

문제 2. 문제 해결 전략 수립

• 최단 경로 알고리즘 vs 깊이 우선 탐색 vs 너비 우선 탐색?

- 본 문제의 경우 가중치 간선이 존재하지 않으므로 최단 거리 알고리즘은 적합하지 않음
- 너비 우선 탐색의 경우 항상 최단 경로를 보장하므로, 너비 우선 탐색을 채택함
 - 너비 우선 탐색은 시작 지점에서 가장 가까운 노드부터 순차적으로 탐색하므로, 최단 경로를 탐색하는데 최적화되어 있음 (▶ 다익스트라 최단거리 알고리즘의 핵심 원리를 생각해 볼 것)
 - 미로 탐색에서 너비 우선 탐색은 각 지점의 단계별 탐색 길이가 같으므로(= 가중치가 모두 동일), 도착 지점까지 최단 거리를 찾을 수 있음



문제 2. 문제 해결 전략 수립

- **최단 경로 찾기**

- 너비 우선 탐색으로 이동 경로 추적(tracking) 가능
- 이를 위해 큐(queue)를 활용함

- **간 길 또 가지 않도록 구현하기**

- 최단 경로를 찾는 문제이므로 왔던 길을 되돌아가는 것은 바람직하지 않음
- 너비 우선 탐색은 각 과정마다 최선의 탐색을 하므로 이미 거쳐온 경로는 다시 탐색하지 않아도 됨

- **레버를 당긴 다음 출구로 가기**

- 레버를 당긴 상태 / 당기지 않은 상태로 특정 지점에 방문한 경우는 서로 다름
- 따라서 "해당 지점에 방문했는지" + "레버를 당겼는지 여부"를 함께 기록해야 함

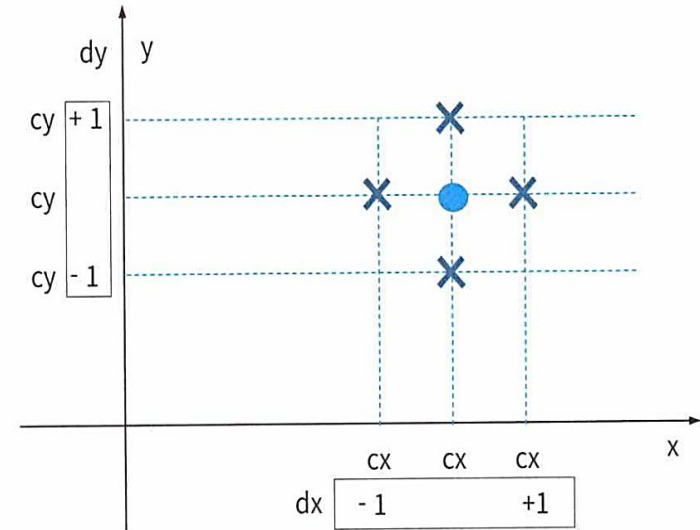
문제 2. 문제 해결 전략 수립

• 미로 내 이동을 표현하는 방법

- 미로 내 특정 지점의 위치: y 좌표 cy , x 좌표 cx
- 상하 / 좌우 이동: 상하 이동 dy , 좌우 이동 dx

• 미로 내 이동 방법

- 너비 우선 탐색(BFS)으로 탐색을 수행하므로 큐(queue)를 활용
- 미로 내 공간 탐색 시 필요한 정보
 - ① 현재 위치 정보 (= y 좌표, x 좌표)
 - ② 레버의 당김 여부 (= k)
 - ③ 현재 시점 까지 이동한 횟수 (= times)
- 미로 내 현재 위치에서의 상태에 따라 다음에 탐색할 위치와 현재까지의 탐색 정보를 큐에 추가
- 이때 현재 위치를 기준으로 다음에 탐색할 위치는 BFS 메커니즘으로 결정



문제 2. 구현 및 검증

```
from collections import deque
```

1. 이동 가능한 좌표인지 판단하는 함수

```
def is_valid_move(ny, nx, n, m, maps):  
    return 0 <= ny < n and 0 <= nx < m and maps[ny][nx] != "X"
```

#2. 방문한 적이 없으면 큐에 넣고 방문 여부를 표시

```
def append_to_queue(ny, nx, k, time, visited, q):  
    if not visited[ny][nx][k]:  
        visited[ny][nx][k] = True  
        q.append((ny, nx, k, time+1))
```

```
def solution2(maps):  
    n, m = len(maps), len(maps[0])  
    visited = [[[False for _ in range(2)] for _ in range(m)] for _ in range(n)]  
  
    # 3. 상, 하, 좌, 우 이동 방향  
    dy = [-1, 1, 0, 0]  
    dx = [0, 0, -1, 1]  
    q = deque()  
    end_y, end_x = -1, -1  
  
    # 4. 시작점과 도착점을 찾아 큐에 넣고 방문 여부 표시  
    for i in range(n):  
        for j in range(m):  
            if maps[i][j] == "S":  
                q.append((i, j, 0, 0)) # start point  
                visited[i][j][0] = True  
            if maps[i][j] == "E":  
                end_y, end_x = i, j # end point
```

```
while q:
    y, x, k, time = q.popleft() # 5. 큐에서 좌표와 이동 횟수를 꺼냄

    # 6. 도착점에 도달하면 결과 반환
    if y == end_y and x == end_x and k == 1:
        return time

    # 7. 네 방향으로 이동하는 각각의 경우에 대하여
    for i in range(4):
        ny, nx = y + dy[i], x + dx[i]

        # 8. 이동 가능한 경우인지 확인
        if not is_valid_move(ny, nx, n, m, maps):
            continue

        if maps[ny][nx] == "L": # 9. 다음 이동 지점이 레버인 경우
            append_to_queue(ny, nx, 1, time, visited, q)
        else: # 10. 다음 이동 지점이 레버가 아닌 경우
            append_to_queue(ny, nx, k, time, visited, q)

return -1 # end of solution3 method
```


문제 2. 구현 및 검증

테스트 케이스 #1

```
maps1 = ["S000L", "XXXX0", "00000", "0XXXX", "0000E"]  
print(solution2(maps1))
```

테스트 케이스 #2

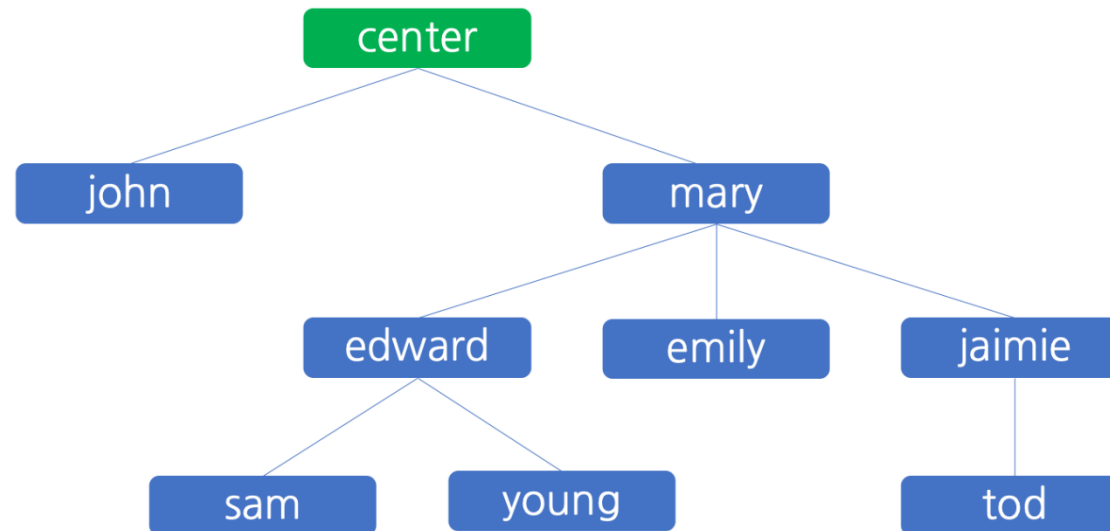
```
maps2 = ["L00XS", "0000X", "00000", "00000", "E0000"]  
print(solution2(maps2))
```

연습문제 3. 다단계 칫솔 판매

문제 3. 다단계 칫솔 판매 (1/14)

제한시간: 60분

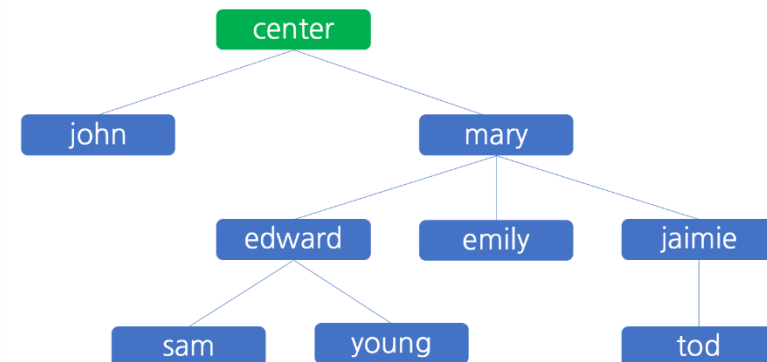
- 민호는 다단계 조직을 이용하여 칫솔을 판매하고 있습니다.
- 판매원이 칫솔을 판매하면 그 이익이 피라미드 조직을 타고 조금씩 분배되는 형태의 판매망입니다.
- 어느정도 판매가 이루어진 후, 조직을 운영하던 민호는 조직 내 누가 얼마만큼의 이득을 가져갔는지가 궁금해졌습니다.
- 예를 들어, 민호가 운영하고 있는 다단계 칫솔 판매 조직이 아래 그림과 같다고 합시다.



문제 3. 다단계 칫솔 판매 (2/14)

제한시간: 60분

- 민호는 center이며, 파란색 네모는 여덟 명의 판매원을 표시한 것입니다.
- 각각은 자신을 조직에 참여시킨 추천인에 연결되어 피라미드 식의 구조를 이루고 있습니다.
- 조직의 이익 분배 규칙은 간단합니다. 모든 판매원은 칫솔의 판매에 의하여 발생하는 이익에서 10% 를 계산하여 자신을 조직에 참여시킨 추천인에게 배분하고 나머지는 자신이 가집니다.
- 모든 판매원은 자신이 칫솔 판매에서 발생한 이익 뿐만 아니라, 자신이 조직에 추천하여 가입시킨 판매원에게서 발생하는 이익의 10% 까지 자신에 이익이 됩니다.
- 자신에게 발생하는 이익 또한 마찬가지로 자신의 추천인에게 분배됩니다.
- 단, 10% 를 계산할 때에는 원 단위에서 절사하며, 10%를 계산한 금액이 1 원 미만인 경우에는 이득을 분배하지 않고 자신이 모두 가집니다.



문제 3. 다단계 칫솔 판매 (3/14)

제한시간: 60분

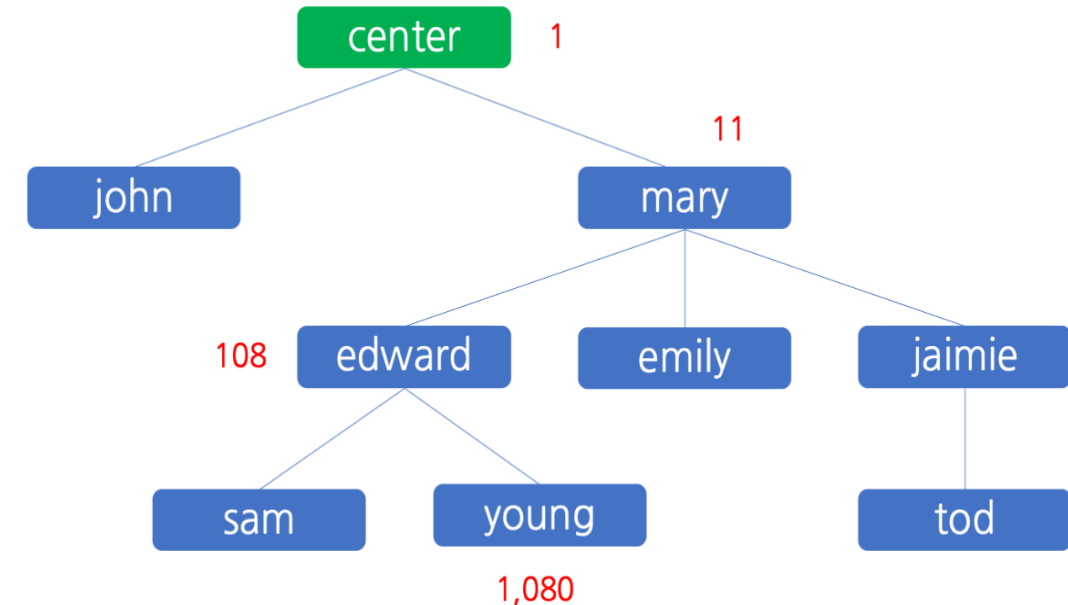
- 예를 들어, 아래와 같은 판매 기록이 있다고 가정하겠습니다.
- 칫솔의 판매에서 발생하는 이익은 개당 100 원으로 정해져 있습니다.

판매원	판매 수량	이익금
young	12	1,200 원
john	4	400 원
tod	2	200 원
emily	5	500 원
mary	10	1,000 원

문제 3. 다단계 칫솔 판매 (4/14)

제한시간: 60분

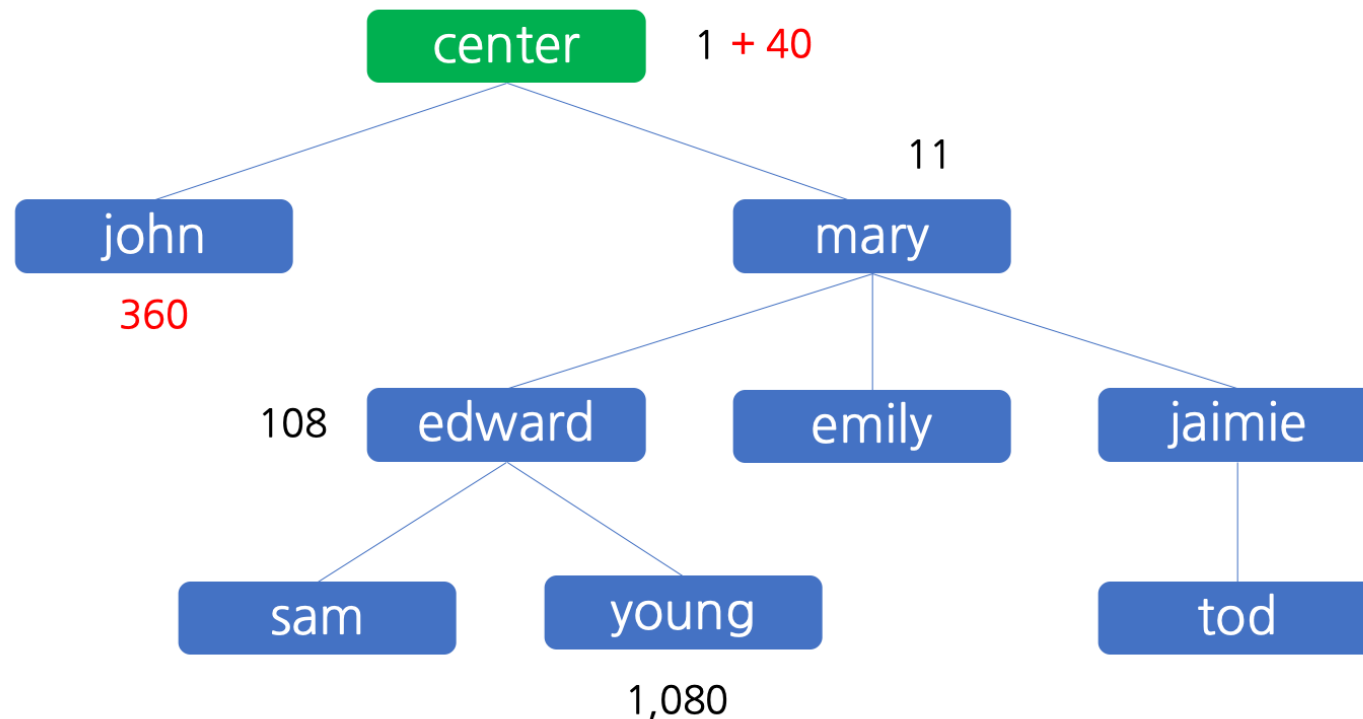
- 판매원 young 에 의하여 1,200 원의 이익이 발생했습니다. young 은 이 중 10% 에 해당하는 120 원을, 자신을 조직에 참여시킨 추천인인 edward 에게 배분하고 자신은 나머지인 1,080 원을 가집니다.
- edward 는 young 에게서 받은 120 원 중 10% 인 12 원을 mary 에게 배분하고 자신은 나머지인 108 원을 가집니다.
- 12 원을 edward 로부터 받은 mary 는 10% 인 1 원을 센터에 (즉, 민호에게) 배분하고 자신은 나머지인 11 원을 가집니다.
- 이 상태를 그림으로 나타내면 오른쪽과 같습니다.



문제 3. 다단계 칫솔 판매 (5/14)

제한시간: 60분

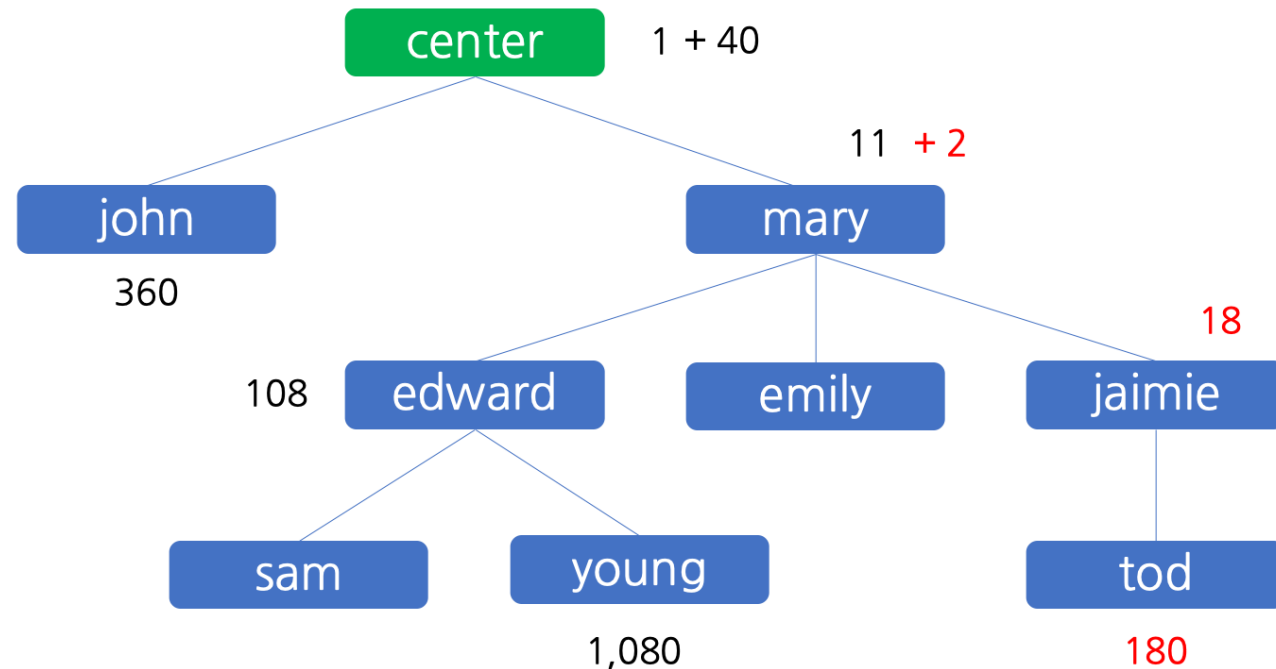
- 그 후, 판매원 john 에 의하여 400 원의 이익이 발생합니다. john 은 10% 인 40 원을 센터에 배분하고 자신이 나머진 360 원을 가집니다.
- 이 상태를 그림으로 나타내면 아래와 같습니다.



문제 3. 다단계 칫솔 판매 (6/14)

제한시간: 60분

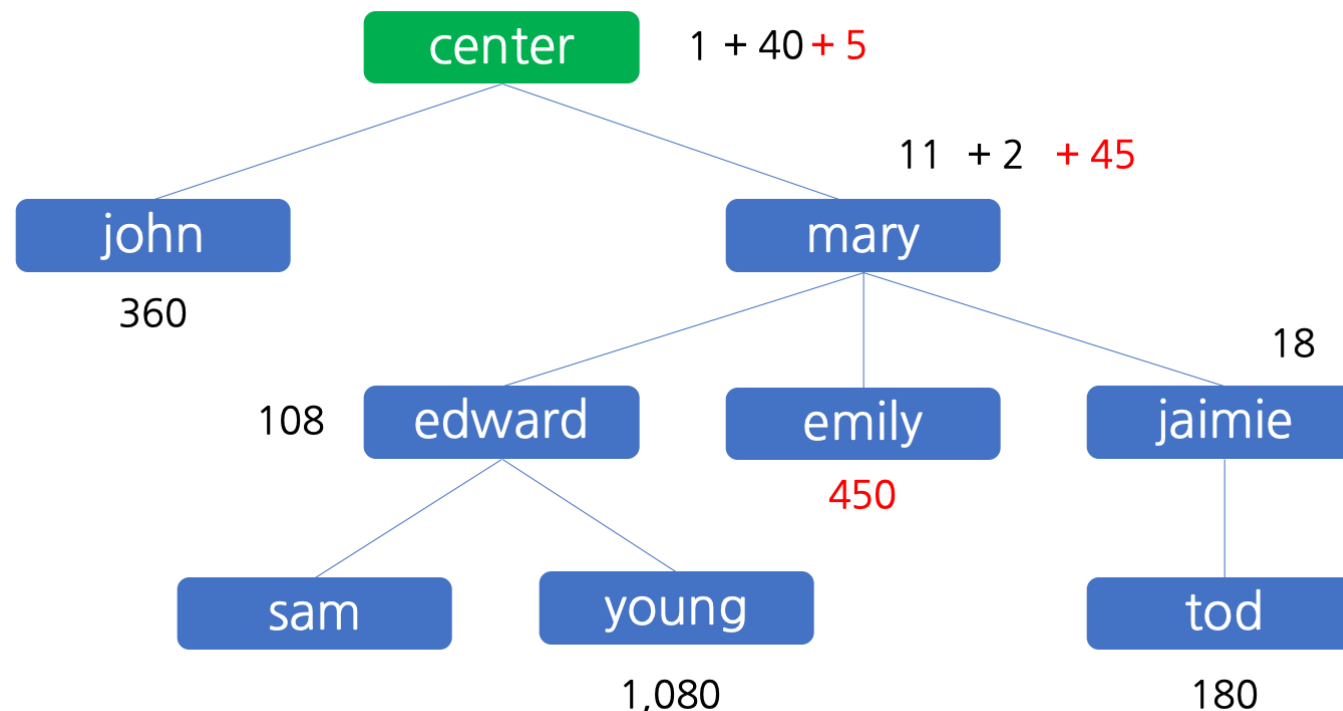
- 또 그 후에는 판매원 tod 에 의하여 200 원 이익이 발생하는데, tod 자신이 180 원을, 추천인인 jaimie 가 그 중 10% 인 20 원을 받아서 18 원을 가지고, jaimie 의 추천인인 mary 는 2 원을 받지만 이것의 10% 는 원 단위에서 절사하면 배분할 금액이 없기 때문에 mary 는 2 원을 모두 가집니다.
- 이 상태를 그림으로 나타내면 오른쪽과 같습니다.



문제 3. 다단계 칫솔 판매 (7/14)

제한시간: 60분

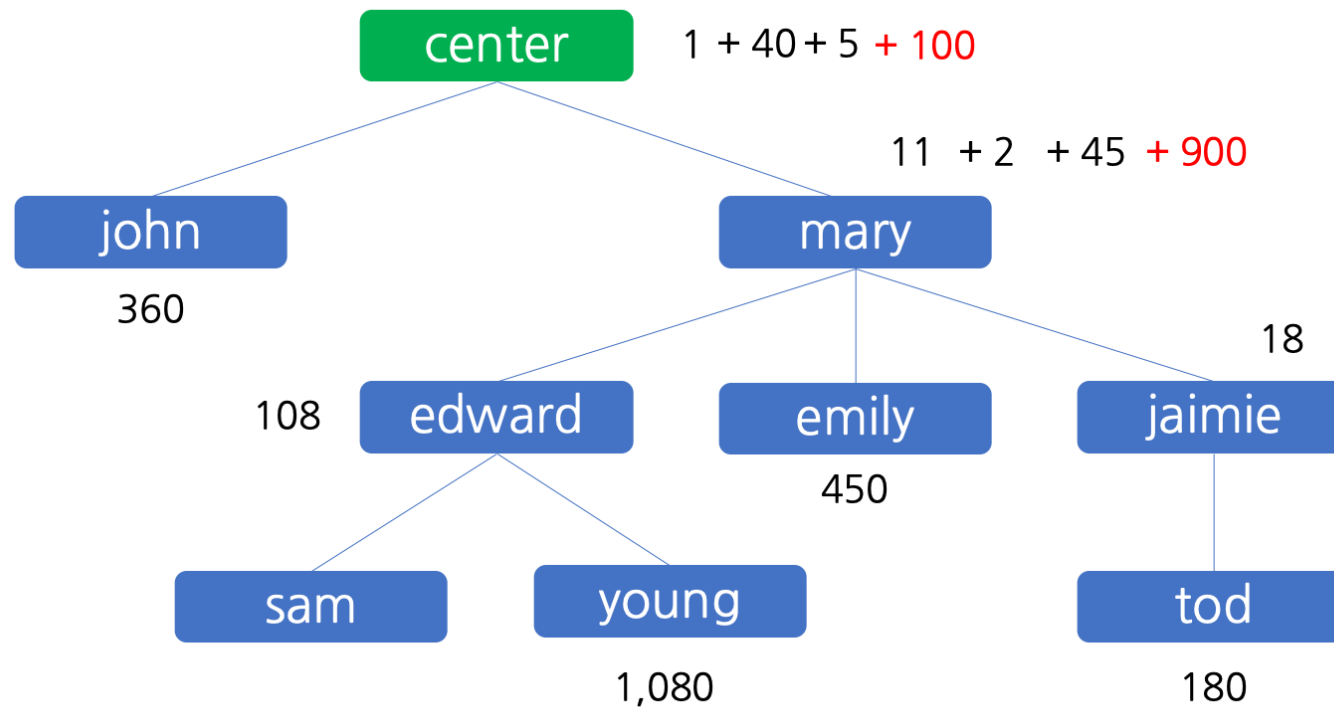
- 그 다음으로 emily 가 칫솔 판매를 통하여 얻은 이익 500 원은 마찬가지로의 규칙에 따라 emily 에게 450 원, mary 에게 45 원, 그리고 센터에 5 원으로 분배됩니다.
- 이 상태를 그림으로 나타내면 아래와 같습니다.



문제 3. 다단계 칫솔 판매 (8/14)

제한시간: 60분

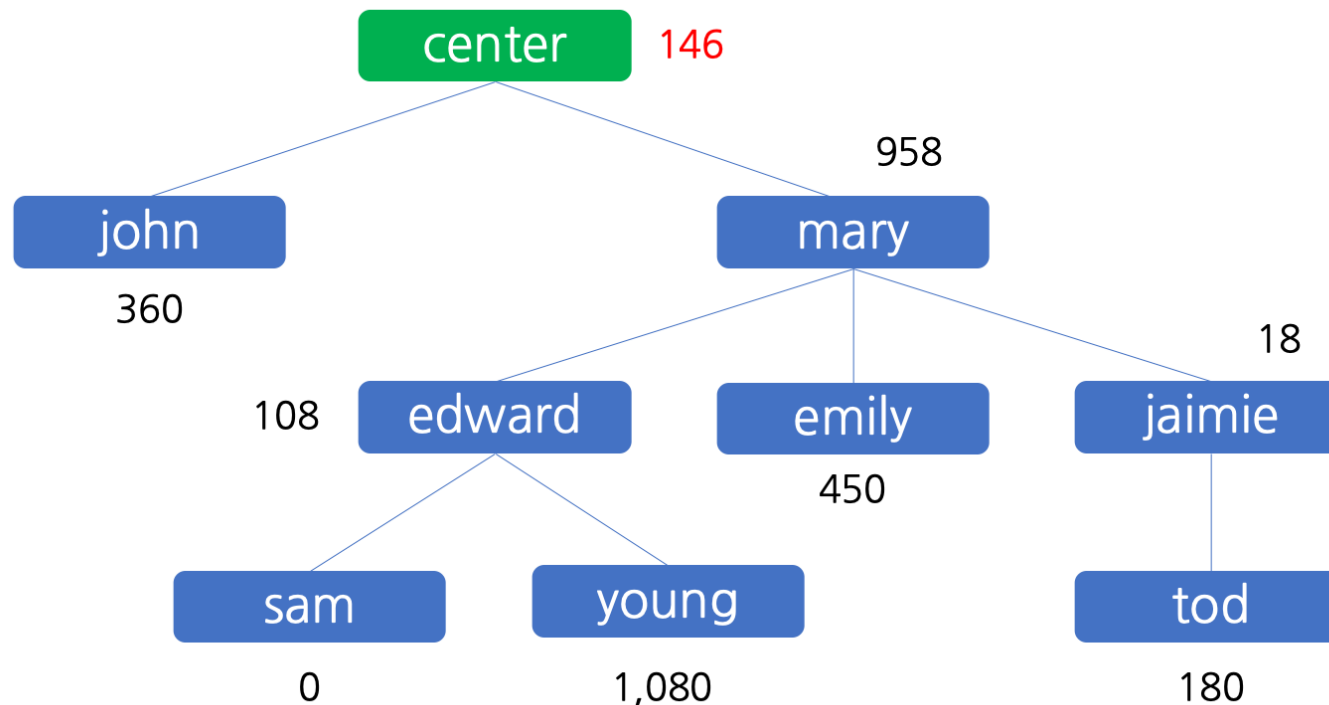
- 마지막으로, 판매원 mary 는 1,000 원의 이익을 달성하고, 이 중 10% 인 100 원을 센터에 배분한 후 그 나머지인 900 원을 자신이 가집니다.
- 이 상태를 그림으로 나타내면 아래와 같습니다.



문제 3. 다단계 칫솔 판매 (9/14)

제한시간: 60분

- 위와 같이 하여 모든 조직 구성원들의 이익 달성 현황 집계가 끝났습니다.
- 지금까지 얻은 이익을 모두 합한 결과를 그림으로 나타내면 아래와 같습니다.
- 이 결과가 민호가 파악하고자 하는 이익 배분 현황입니다.



문제 3. 다단계 칫솔 판매 (10/14)

제한시간: 60분

- 각 판매원의 이름을 담은 배열 `enroll`, 각 판매원을 다단계 조직에 참여시킨 다른 판매원의 이름을 담은 배열 `referral`, 판매량 집계 데이터의 판매원 이름을 나열한 배열 `seller`, 판매량 집계 데이터의 판매 수량을 나열한 배열 `amount`가 매개변수로 주어질 때, 각 판매원이 득한 이익금을 나열한 배열을 `return` 하도록 **solution3** 함수를 완성해주세요.
- 판매원에게 배분된 이익금의 총합을 계산하여(정수형으로), 입력으로 주어진 `enroll`에 이름이 포함된 순서에 따라 나열하면 됩니다.

문제 3. 다단계 칫솔 판매 (11/14)

제한시간: 60분

• 제한사항

- enroll의 길이는 1 이상 10,000 이하입니다.
 - enroll에 민호의 이름은 없습니다. 따라서 enroll의 길이는 민호를 제외한 조직 구성원의 총 수입니다.
- referral의 길이는 enroll의 길이와 같습니다.
 - referral 내에서 i 번째에 있는 이름은 배열 enroll 내에서 i 번째에 있는 판매원을 조직에 참여시킨 사람의 이름입니다.
 - 어느 누구의 추천도 없이 조직에 참여한 사람에 대해서는 referral 배열 내에 추천인의 이름이 기입되지 않고 "-" 가 기입됩니다. 위 예제에서는 john 과 mary 가 이러한 예에 해당합니다.
 - enroll 에 등장하는 이름은 조직에 참여한 순서에 따릅니다.
 - 즉, 어느 판매원의 이름이 enroll 의 i 번째에 등장한다면, 이 판매원을 조직에 참여시킨 사람의 이름, 즉 referral 의 i 번째 원소는 이미 배열 enroll 의 j 번째 ($j < i$) 에 등장했음이 보장됩니다.

문제 3. 다단계 칫솔 판매 (12/14)

제한시간: 60분

- seller의 길이는 1 이상 100,000 이하입니다.
 - seller 내의 i 번째에 있는 이름은 i 번째 판매 집계 데이터가 어느 판매원에 의한 것인지를 나타냅니다.
 - seller 에는 같은 이름이 중복해서 들어있을 수 있습니다.
- amount의 길이는 seller의 길이와 같습니다.
 - amount 내의 i 번째에 있는 수는 i 번째 판매 집계 데이터의 판매량을 나타냅니다.
 - 판매량의 범위, 즉 amount 의 원소들의 범위는 1 이상 100 이하인 자연수입니다.
- 칫솔 한 개를 판매하여 얻어지는 이익은 100 원으로 정해져 있습니다.
- 모든 조직 구성원들의 이름은 10 글자 이내의 영문 알파벳 소문자들로만 이루어져 있습니다.

문제 3. 다단계 칫솔 판매 (13/14)

제한시간: 60분

• 입출력 예제

enroll	referral	seller	amount	result
["john", "mary", "edward", "sam", "emily", "jaimie", "tod", "young"]	["-", "-", "mary", "edward", "mary", "jaimie", "edward"]	["young", "john", "tod", "emily", "mary"]	[12, 4, 2, 5, 10]	[360, 958, 108, 0, 450, 18, 180, 1080]
["john", "mary", "edward", "sam", "emily", "jaimie", "tod", "young"]	["-", "-", "mary", "edward", "mary", "jaimie", "edward"]	["sam", "emily", "jaimie", "edward"]	[2, 3, 5, 4]	[0, 110, 378, 180, 270, 450, 0, 0]

문제 3. 다단계 칫솔 판매 (14/14)

제한시간: 60분

- 입출력 예제 설명

- 입출력 예제 #1

- 문제의 예시와 같습니다.

- 입출력 예제 #2

- 문제에 주어진 예시와 동일한 조직 구성에 조금 다른 판매량 집계를 적용한 것입니다.

- 이익을 분배하는 규칙이 동일하므로, 간단한 계산에 의하여 표에 보인 결과를 얻을 수 있습니다.

문제 3. 문제 분석

• 문제의 목표

- 각 판매원이 얻은 최종 이익금을 계산하여 이를 리스트로 반환한다.
- 이때 판매원의 이익금은 정수 단위로 계산하고, enroll 리스트에 나타나는 순서대로 나열한다.

• 문제에서 고려해야 할 사항

- 이익의 계산 방법: 판매 수량 \times 100원
 - 단, 10%를 계산할 때에는 "원" 단위에서 자름 \rightarrow 즉, 소수점은 고려하지 않음
- 이익의 배분
 - 자신이 달성한 이익금의 10%를 부모 노드에게 분배하고 나머지는 자신이 가짐
 - 단, 자신이 달성한 이익금이 1원 미만이면 이익금을 분배하지 않고 자신이 다 가짐
- 판매자의 이익 계산 방법
 - 자신이 달성한 이익금 + 자식 노드 각각의 이익금의 10%

문제 3. 문제 해결 전략 수립

- 그림을 보고 판매자 간의 관계를 분석한다.
 - 부모 노드: referral (자신을 추천한 판매원)
 - 자식 노드: enroll (자신이 등록시킨 판매원)
 - 이때, referral과 enroll 간의 관계(= 트리 구조)를 나타내는 자료구조가 필요함 → 해시 테이블(딕셔너리)
- 문제에서 요구하는 내용은 “각 판매자의 수익 총액”
 - 각 판매자 별로 수익 총액을 저장하는 자료 구조가 필요함 → 해시 테이블(딕셔너리)

enroll	referral
“john”	“-”
“mary”	“-”
“edward”	“mary”
“sam”	“edward”
“emily”	“mary”
“jaimie”	“mary”
“tod”	“jaimie”
“young”	“edward”

merry가
edward를
추천

문제 3. 문제 해결 전략 수립

- Referral과 enroll 사이의 관계를 나타내는 자료 구조 설계

- 개념적으로 트리 구조라 하더라도 구현 단계에서 반드시 트리를 구현할 필요는 없음
 - 본 문제에서 트리는 "이익을 계산하는 방법"을 구체적으로 표현하기 위한 수단
 - 실제 구현 시에는 트리보다는 딕셔너리가 더욱 적합함
- 이익을 배분하는 계산 과정을 살펴보면...
 - 자기 자신이 취득한 이익의 10%를 부모 노드에 배당함
 - 즉, 자기 자신에서 부모 노드로, 즉 상향식으로 이익이 배분되는 흐름임
- 이 경우, "자기 자신의 부모 노드"를 참조하는 경우가 빈번함
 - 따라서 Key를 "enroll", Value를 "referral"로 갖는 딕셔너리를 사용하는 것이 바람직함

문제 3. 문제 해결 전략 수립

- Referral과 enroll 사이의 관계를 나타내는 자료 구조 설계

➤ 이에 따라, 아래와 같이 두 개의 딕셔너리를 구축한다.

- 추천자와 등록자 관계를 저장하는 딕셔너리 → Key: enroll, Value: referral인 딕셔너리
- 각 판매자 별 수익을 저장하는 딕셔너리 → Key: enroll 내 판매자 ID, Value: 수익

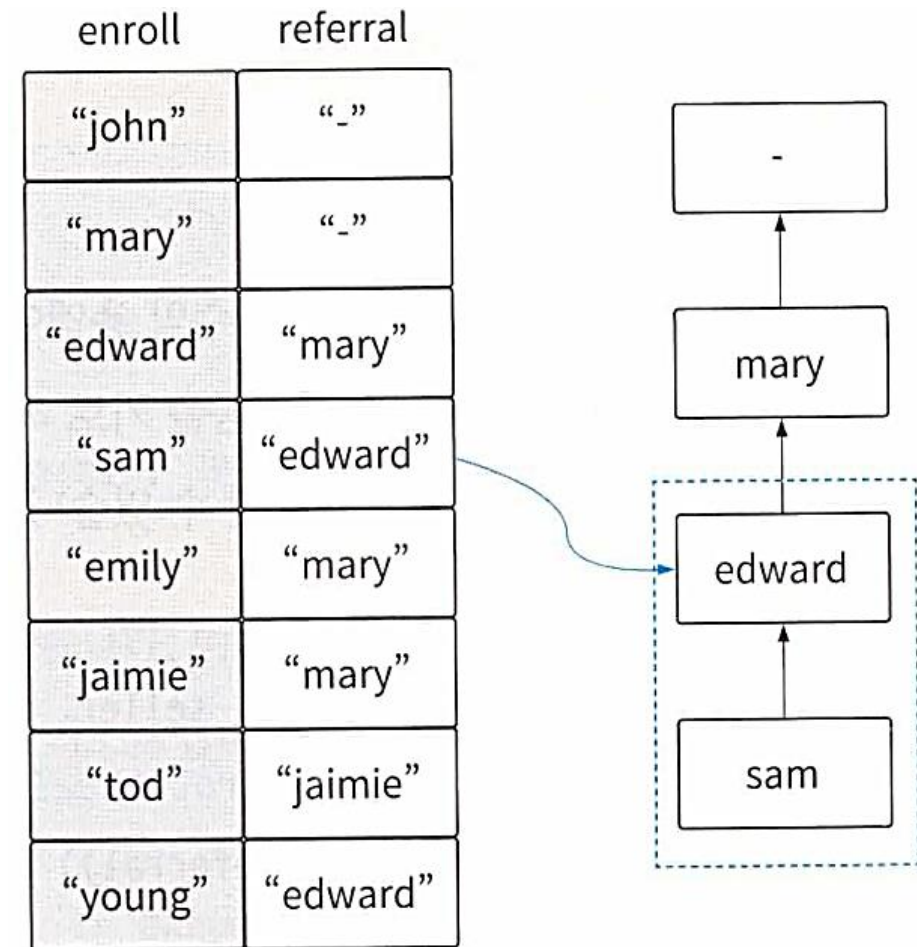
enroll	referral
"john"	"_"
"mary"	"_"
"edward"	"mary"
"sam"	"edward"
"emily"	"mary"
"jaimie"	"mary"
"tod"	"jaimie"
"young"	"edward"

"john"	0
"mary"	0
"edward"	0
"sam"	0
"emily"	0
"jaimie"	0
"tod"	0
"young"	0

문제 3. 문제 해결 전략 수립

• 예제 (1)

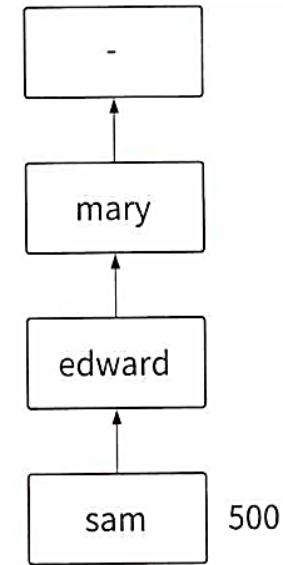
- 디렉토리를 참고하여 판매자의 추천자를 찾는다.
- 오른쪽 그림에서 sam의 추천자는 edward
- edward의 추천자는 mary
- mary의 추천자는 없음



문제 3. 문제 해결 전략 수립

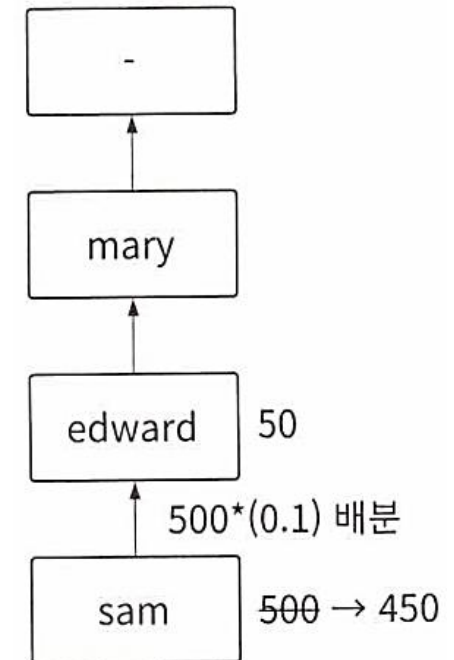
• 예제 (2)

➤ sam은 칫솔 5개를 팔았으므로 500원의 수익을 거둠



➤ sam은 자신이 달성한 이익 500원의 10% ($=500 \times 0.1 = 50$ 원)를 추천자 edward에게 분배함

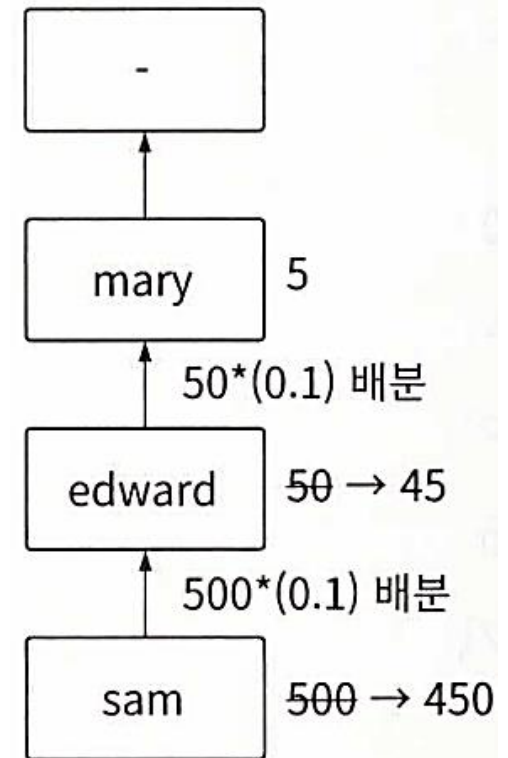
➤ 이에 따라 sam 자신의 이익은 $500 - 50 = 450$ 원이 됨



문제 3. 문제 해결 전략 수립

• 예제 (3)

- edward는 50원의 10%인 5원을 mary에게 분배함
- 이에 따라, edward에게 실질적으로 떨어지는 수익금은 45원이 됨



문제 3. 문제 해결 방법 기술

- **딕셔너리 생성 및 초기화**

- Dictionary "**parent**" ← Key: enroll, Value: referral
- Dictionary "**total**" ← Key: enroll, Value: 0

- **각 판매자의 이익을 계산하고 각 이익의 10%를 판매자의 부모 노드에게 분배한다.**

- 입력 리스트 "seller" 내 각 판매자에 대해서 아래의 작업을 반복한다.
 - 판매자의 이익을 계산한다.
 - 판매자의 상위 레벨에 존재하는 부모 노드에 순차적으로 이익을 분배한다.
 - 분배한 이익을 제외한 나머지 금액을 자신의 이익으로 취한다.

- **"enroll"에 등장하는 모든 판매자의 최종 이익을 리스트 타입으로 반환한다.**

문제 3. 구현 및 검증

```
def solution3(enroll, referral, seller, amount):  
    # 1. parent 딕셔너리 생성 및 초기화  
    parent = dict(zip(enroll, referral))  
  
    # 2. total 딕셔너리 생성 및 초기화  
    total = {name: 0 for name in enroll}  
  
    # 3. seller 리스트와 amount 리스트를 이용하여 이익을 분배한다.  
    for i in range(len(seller)):  
        # 4. 판매자가 판매한 총 금액을 계산한다.  
        money = amount[i] * 100  
        cur_name = seller[i]
```

문제 3. 구현 및 검증

5. 판매자로부터 차례대로 상위 노드로 이동하며 이익을 분배한다.

```
while money > 0 and cur_name != "-":
```

6. 상위 판매자가 받을 금액을 계산한다.

```
total[cur_name] += money - money // 10
```

```
cur_name = parent[cur_name]
```

7. 10%를 제외한 자신이 가질 금액

```
money //= 10
```

8. enroll 리스트의 모든 노드에 대해 해당하는 이익을 리스트로 반환한다.

```
return [total[name] for name in enroll]
```

문제 3. 구현 및 검증

테스트 케이스 #1

```
enroll1 = ["john", "mary", "edward", "sam", "emily", "jaimie", "tod", "young"]
referral1 = ["-", "-", "mary", "edward", "mary", "mary", "jaimie", "edward"]
seller1 = ["young", "john", "tod", "emily", "mary"]
amount1 = [12, 4, 2, 5, 10]
print(solution3(enroll1, referral1, seller1, amount1))
```

테스트 케이스 #2

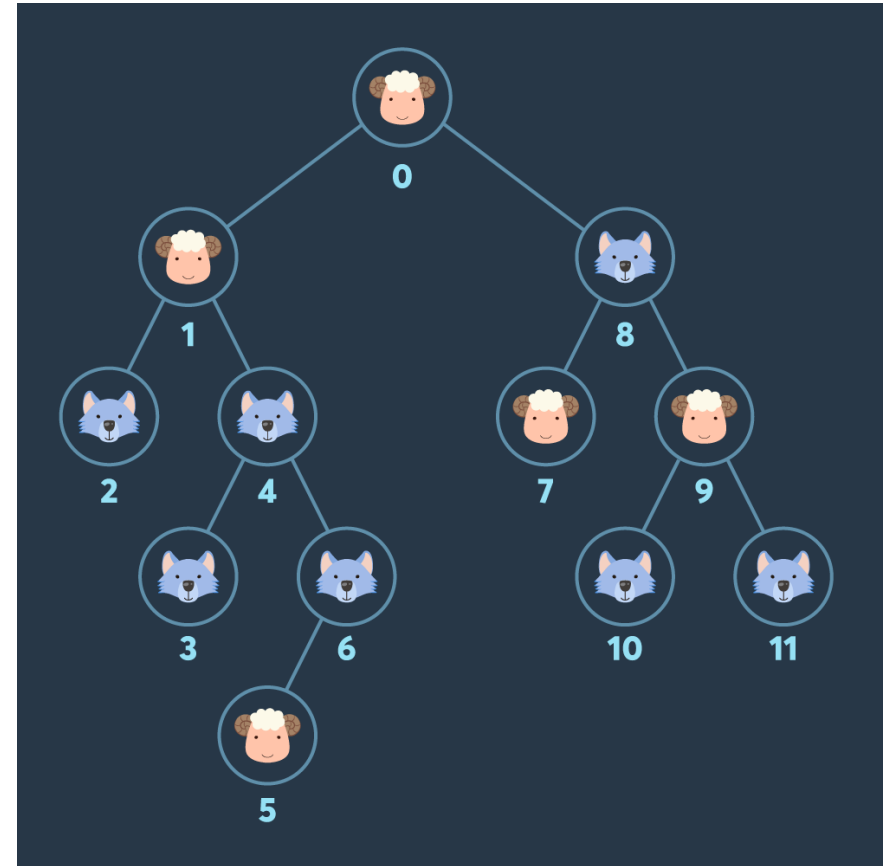
```
enroll2 = ["john", "mary", "edward", "sam", "emily", "jaimie", "tod", "young"]
referral2 = ["-", "-", "mary", "edward", "mary", "mary", "jaimie", "edward"]
seller2 = ["sam", "emily", "jaimie", "edward"]
amount2 = [2, 3, 5, 4]
print(solution3(enroll2, referral2, seller2, amount2))
```

연습문제 4. 양과 능력

문제 4. 양과 늑대 (1/6)

제한시간: 70분

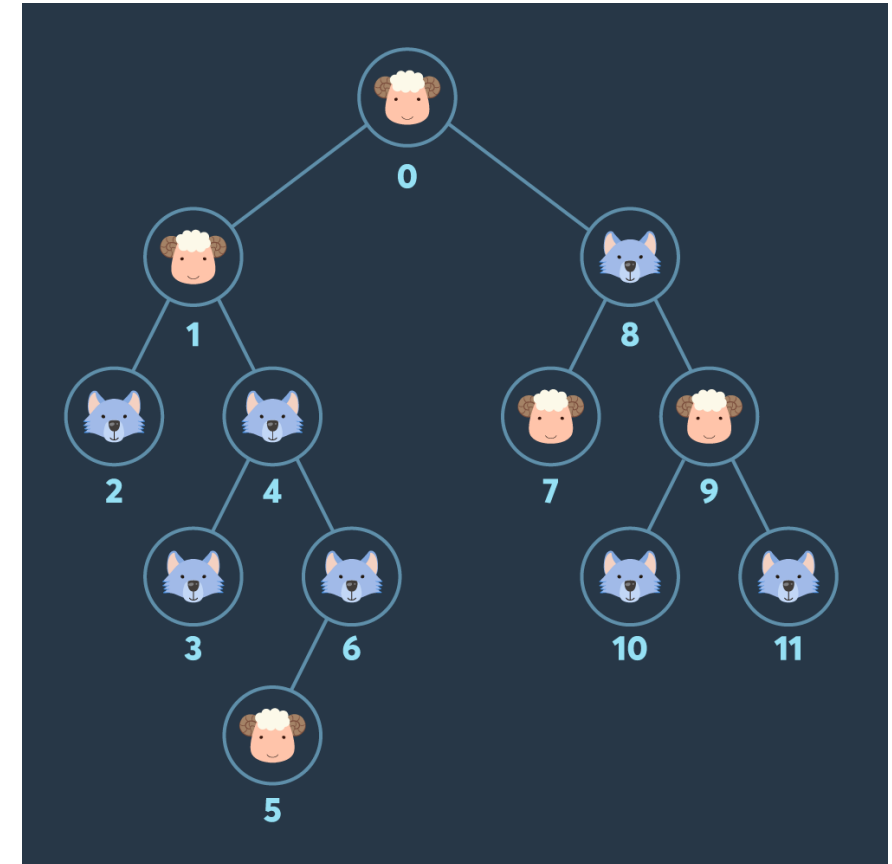
- 오른쪽 그림과 같이 2진 트리 모양 초원의 각 노드에 늑대와 양이 한 마리씩 놓여 있습니다.
- 이 초원의 루트 노드에서 출발하여 각 노드를 돌아다니며 양을 모으려 합니다.
- 각 노드를 방문할 때 마다 해당 노드에 있던 양과 늑대가 당신을 따라오게 됩니다.
- 이때, 늑대는 양을 잡아먹을 기회를 노리고 있으며, 당신이 모은 양의 수보다 늑대의 수가 같거나 더 많아지면 바로 모든 양을 잡아먹어 버립니다.
- 당신은 중간에 양이 늑대에게 잡아먹히지 않도록 하면서 최대한 많은 수의 양을 모아서 다시 루트 노드로 돌아오려 합니다.



문제 4. 양과 늑대 (2/6)

제한시간: 70분

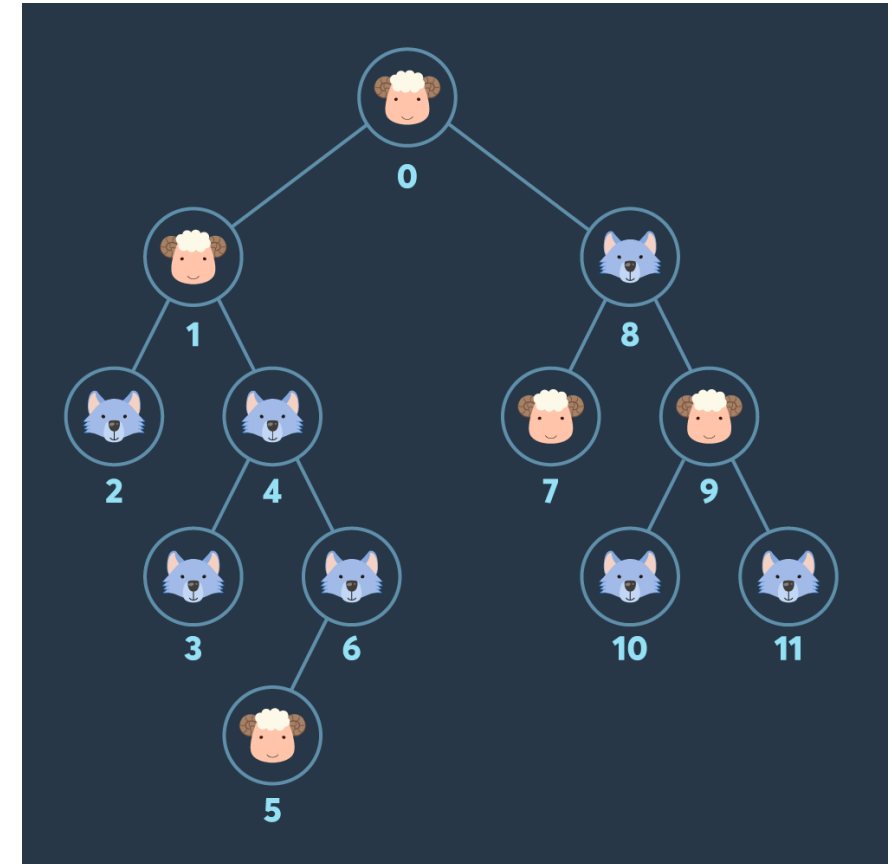
- 예를 들어, 위 그림의 경우(루트 노드에는 항상 양이 있습니다) 0번 노드(루트 노드)에서 출발하면 양을 한 마리 모을 수 있습니다.
- 다음으로 1번 노드로 이동하면 당신이 모은 양은 두 마리가 됩니다.
- 이때, 바로 4번 노드로 이동하면 늑대 한 마리가 당신을 따라오게 됩니다.
- 아직은 양 2마리, 늑대 1마리로 양이 잡아먹히지 않지만, 이후에 갈 수 있는 아직 방문하지 않은 모든 노드(2, 3, 6, 8번)에는 늑대가 있습니다.



문제 4. 양과 늑대 (3/6)

제한시간: 70분

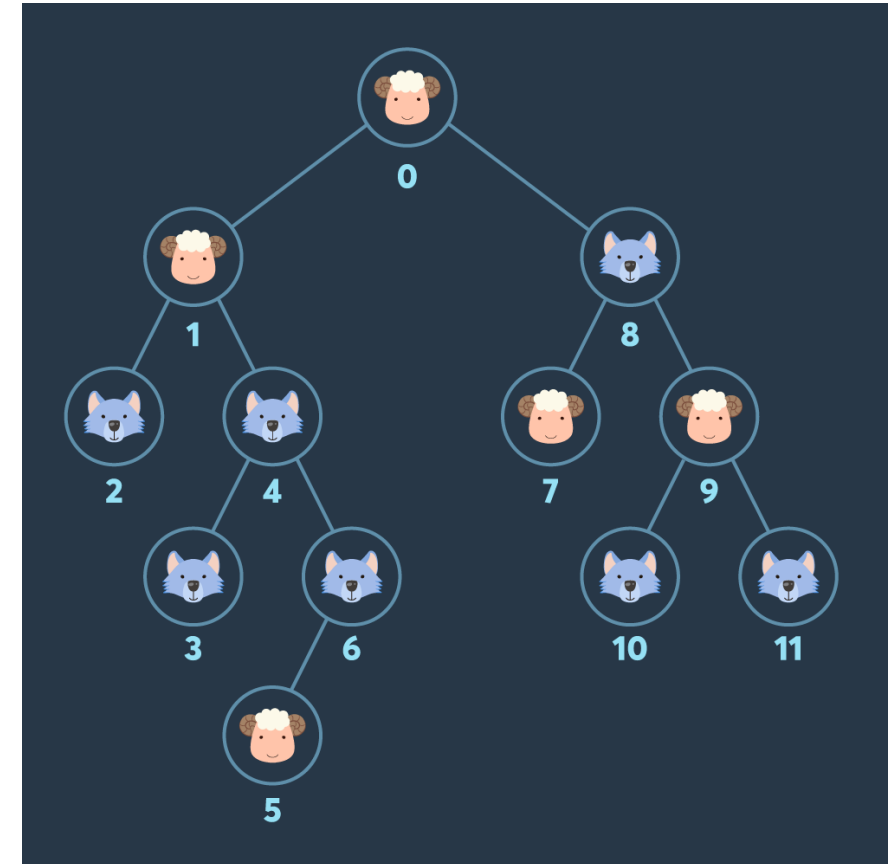
- 이어서 늑대가 있는 노드로 이동한다면(예를 들어 바로 6번 노드로 이동한다면) 양 2마리, 늑대 2마리가 되어 양이 모두 잡아먹힙니다.
- 여기서는 0번, 1번 노드를 방문하여 양을 2마리 모은 후, 8번 노드로 이동한 후(양 2마리 늑대 1마리) 이어서 7번, 9번 노드를 방문하면 양 4마리 늑대 1마리가 됩니다.
- 이제 4번, 6번 노드로 이동하면 양 4마리, 늑대 3마리가 되며, 이제 5번 노드로 이동할 수 있게 됩니다.
- 따라서 양을 최대 5마리 모을 수 있습니다.



문제 4. 양과 늑대 (4/6)

제한시간: 70분

- 각 노드에 있는 양 또는 늑대에 대한 정보가 담긴 배열 info, 2진 트리의 각 노드들의 연결 관계를 담은 2차원 배열 edges가 매개변수로 주어질 때, 문제에 제시된 조건에 따라 각 노드를 방문하면서 모을 수 있는 양은 최대 몇 마리인지 return 하도록 **solution4** 함수를 완성해주세요.



문제 4. 양과 늑대 (5/6)

제한시간: 70분

• 제한사항

① $2 \leq \text{info의 길이} \leq 17$

- info의 원소는 0 또는 1 입니다.
- info[i]는 i번 노드에 있는 양 또는 늑대를 나타냅니다.
- 0은 양, 1은 늑대를 의미합니다.
- info[0]의 값은 항상 0입니다. 즉, 0번 노드(루트 노드)에는 항상 양이 있습니다.

② edges의 세로(행) 길이 = info의 길이 - 1

- edges의 가로(열) 길이 = 2
- edges의 각 행은 [부모 노드 번호, 자식 노드 번호] 형태로, 서로 연결된 두 노드를 나타냅니다.
- 동일한 간선에 대한 정보가 중복해서 주어지지 않습니다.
- 항상 하나의 이진 트리 형태로 입력이 주어지며, 잘못된 데이터가 주어지는 경우는 없습니다.
- 0번 노드는 항상 루트 노드입니다.

문제 4. 양과 늑대 (6/6)

제한시간: 70분

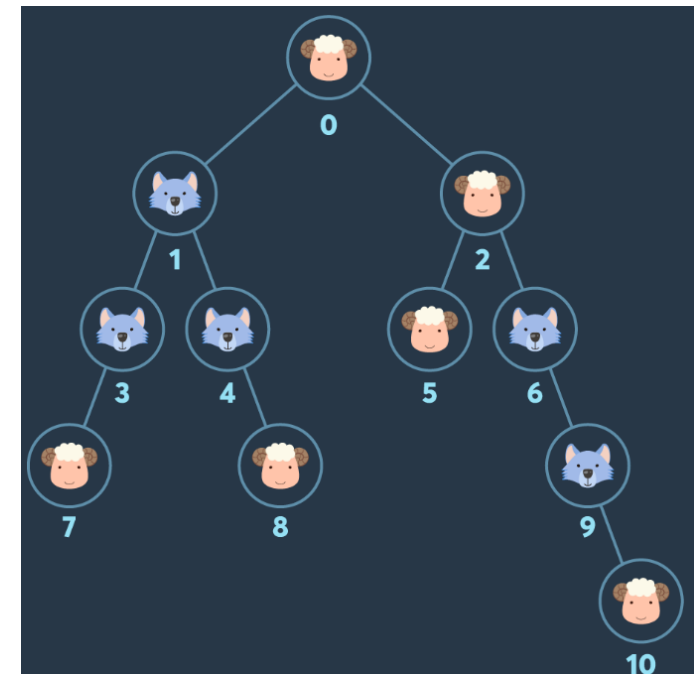
• 입출력 예제 설명

info	edges	result
[0,0,1,1,1,0,1,0,1,0,1,1]	[[0,1],[1,2],[1,4],[0,8],[8,7],[9,10],[9,11],[4,3],[6,5],[4,6],[8,9]]	5
[0,1,0,1,1,0,1,0,0,1,0]	[[0,1],[0,2],[1,3],[1,4],[2,5],[2,6],[3,7],[4,8],[6,9],[9,10]]	5

➤ 예제 #1 설명: 문제의 예시와 같습니다.

➤ 예제 #2 설명 (※ 오른쪽 그림 참고)

- 0번 - 2번 - 5번 - 1번 - 4번 - 8번 - 3번 - 7번 노드 순으로 이동하면 양 5마리 늑대 3마리가 됩니다.
- 여기서 6번, 9번 노드로 이동하면 양 5마리, 늑대 5마리가 되어 양이 모두 잡아먹히게 됩니다.
- 따라서 늑대에게 잡아먹히지 않도록 하면서 최대한 모을 수 있는 양은 5마리입니다.



문제 4. 문제 분석

- 문제의 목표

- 주어진 트리를 순차적으로 순회하면서 모을 수 있는 양의 최대 마리 수를 계산

- 문제에서 고려해야 할 사항

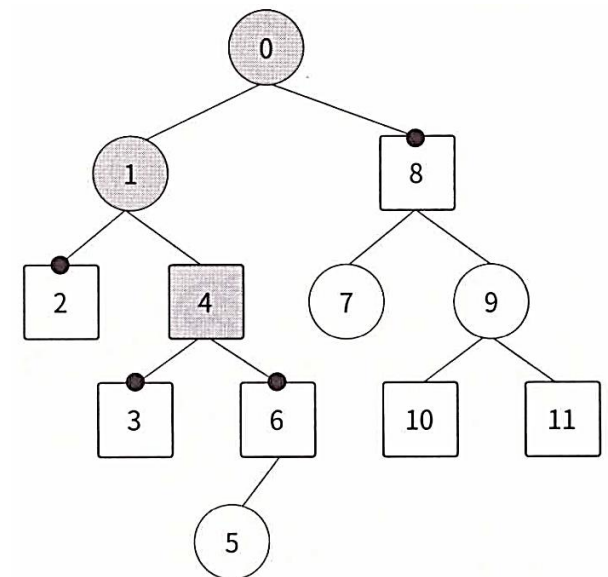
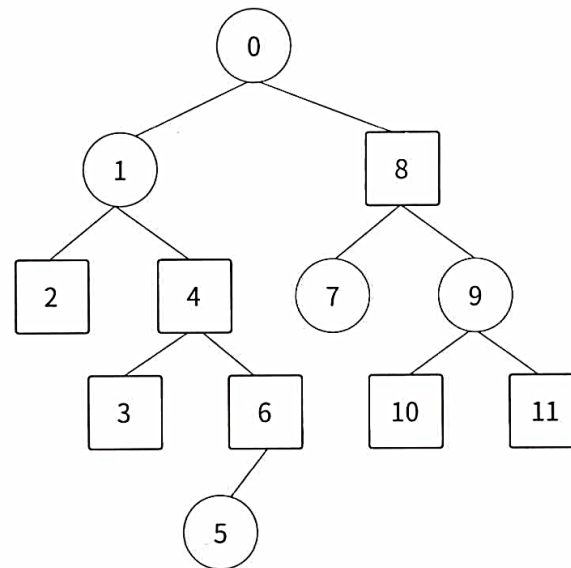
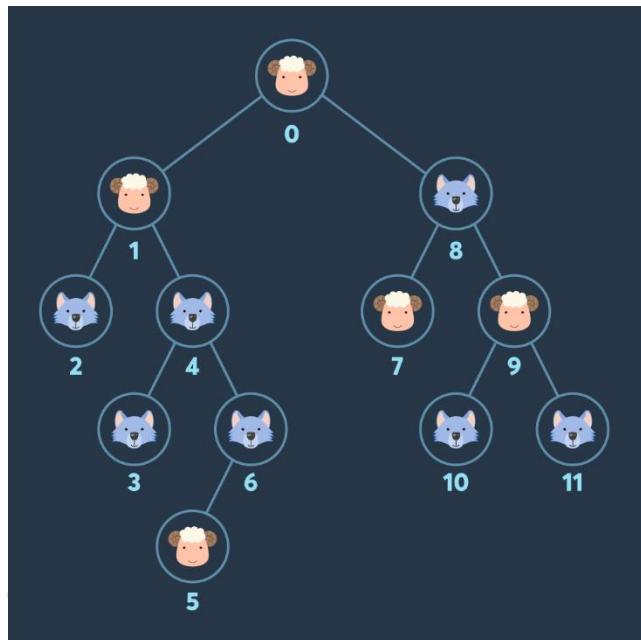
- 모은 양의 수가 모은 늑대의 수보다 많아야 한다.
 - "모은 양의 수 \leq 모은 늑대의 수"인 경우 양들이 모두 늑대에 잡아 먹히므로 허용 X
- 현재 방문한 노드를 기준으로 **인접한 모든 노드**가 방문의 대상이다.
 - 문제에 주어진 예제를 분석하면 **현재까지 방문한 노드들과 인접한 노드들만 방문할 수 있음**을 확인할 수 있다.
- 노드의 방문 순서에 따라 모을 수 있는 양의 수가 달라진다.
 - 즉, 순회 가능한 여러 경우 중에서 최적의 경우(= 해)를 탐색하는 문제이다.

너비 우선 탐색을 이용하는 것이 가장 적합함.

문제 4. 문제 해결 전략 수립

• 주어진 트리를 간소화하여 표현

- 입력 리스트 info의 값이 0인 경우(= 양)는 동그라미로, 1인 경우(= 늑대)는 네모로 표현
- 입출력 예제 #1번을 트리 표현하면 아래 가운데 그림과 같음
- 0번에서 출발하여 1번, 4번 노드까지 이동한 상황이라고 가정할 때, 다음 단계에서 이동할 수 있는 노드는 2, 3, 6, 8임 (※ 아래 오른쪽 그림 참고)



- tree

부모 노드 자식 노드

문제 4. 문제 해결 전략 수립

• BFS 수행

- 큐에서 탐색 상태를 가져온다.
 - 최대 양의 수를 업데이트한다.
 - 현재까지 방문한 노드 집합에 현재 노드의 이웃 노드를 추가
- 현재 노드와 인접한 노드들에 대한 탐색을 수행한다.
 - 인접한 노드에 늑대가 존재하는 경우
 - ✓ 늑대의 수가 1 증가했을 때에도 양의 수보다 많아지지 않으면 해당 상태 정보를 큐에 추가
<해당 노드 번호, 현재 양의 수, **현재 늑대의 수 + 1**, 방문한 노드 집합 - 현재 노드>
 - 인접한 노드에 양이 존재하는 경우
 - ✓ 양은 무조건 모아야 하는 대상이므로 양의 수를 1 증가시킨 후 해당 상태 정보를 큐에 추가
<해당 노드 번호, **현재 양의 수 + 1**, 현재 늑대의 수, 방문한 노드 집합 - 현재 노드>

문제 4. 구현 및 검증

```
from collections import deque
```

```
def solution4(info, edges):  
    # 1. 트리 함수 구축  
    def build_tree(info, edges):  
        tree = [[] for _ in range(len(info))]  
        for edge in edges:  
            tree[edge[0]].append(edge[1])  
        return tree  
  
    tree = build_tree(info, edges) # 2. 트리 생성  
    max_sheep = 0 # 3. 최대 양의 수  
  
    # 4. BFS를 위한 큐 생성 및 초기 상태 설정  
    # (현재 위치, 양의 수, 늑대의 수, 방문한 노드 집합)  
    q = deque([ (0, 1, 0, set()) ])
```

```
# BFS 시작
```

```
while q:
```

```
    current, sheep_count, wolf_count, visited = q.popleft() # 5. 큐에서 상태 가져오기
```

```
    max_sheep = max(max_sheep, sheep_count) # 6. 최대 양의 수 업데이트
```

```
    visited.update(tree[current]) # 7. 방문한 노드 집합에 현재 노드의 이웃 노드 추가
```

```
# 8. 인접한 노드들에 대해 탐색
```

```
for next_node in visited:
```

```
    if info[next_node]: # 9. 늑대인 경우
```

```
        if sheep_count != wolf_count + 1:
```

```
            q.append(
```

```
                (next_node, sheep_count, wolf_count + 1, visited - {next_node})
```

```
            )
```

```
    else: # 10. 양인 경우
```

```
        q.append(
```

```
            (next_node, sheep_count + 1, wolf_count, visited - {next_node})
```

```
        )
```

```
return max_sheep
```


문제 4. 구현 및 검증

테스트 케이스 #1

```
info1 = [0,0,1,1,1,0,1,0,1,0,1,1]
edges1 = [[0,1],[1,2],[1,4],[0,8],[8,7],[9,10],[9,11],[4,3],[6,5],[4,6],[8,9]]
print(solution4(info1, edges1))
```

테스트 케이스 #2

```
info2 = [0,1,0,1,1,0,1,0,0,1,0]
edges2 = [[0,1],[0,2],[1,3],[1,4],[2,5],[2,6],[3,7],[4,8],[6,9],[9,10]]
print(solution4(info2, edges2))
```

Q & A