[22.11.19] 조별경기

TS조별경기

김 태 현

문 제

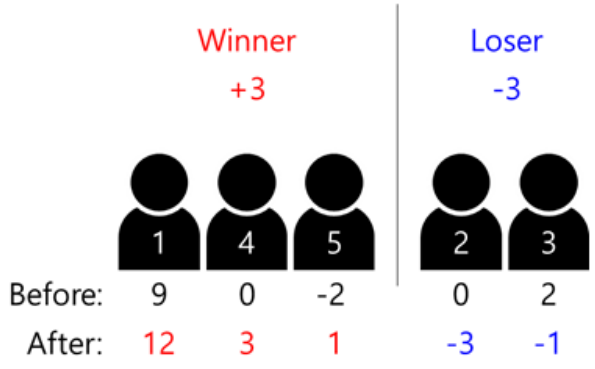
조별 경기에 1 ~ N의 ID를 가지는 선수가 N명 존재한다.

처음에는 1인 1조로 출전하여 서로 다는 조에 속해 있고 점수는 0이다.

서로 다른 두 조가 하나의 조로 합쳐질 수 있으며 선수들 개개인의 기존 점수는 그대로 유지된다.

합쳐진 조로 경기를 진행한 결과는 해당 조에 속한 모든 선수들에게 동일하게 반영된다.

조별 경기는 서로 다른 두 조를 선정하여 진행되며, 판정된 점수를 승리팀은 얻게 되고 패배팀은 잃게 된다(Fig.1)



[Fig. 1]

---

void init(int N)

N명의 선수가 1인 1조, 0점으로 출전한다.

void updateScore(int mWinnerID, int mLoserID, int mScore)

mWinnerID인 선수가 속한 조가 승리하여 해당 조에 속한 선수들은 mScore의 점수를 얻는다.
mLoserID인 선수가 속한 조가 패배하여 해당 조에 속한 선수들은 mScore의 점수를 잃는다.

※ 제약사항

4 <= N <= 100,000
updateScore() <= 50,000
unionTeam() <= N
getScore() <= 50,000

int getScore(int mID)

ID가 mID인 선수의 점수를 확인한다.

void unionTeam(int mPlayerA, int mPlayerB)

mPlayerA인 선수가 속한 조와 mPlayerB인 선수가 속한 조를 하나로 합친다.

Naive

updateScore()

각 그룹의 모든 선수 점수 업데이트
O(업데이트 되는 총 인원 수)
worst case O(100,000 * 50,000)

unionTeam()

그룹B 를 그룹A로 합병

O(합병되는 총 인원 수)
worst case O(1+2+3+..+N-1)

getScore()

return score[pid]

Player 별 그룹번호, 점수

pid	group	score
1	1	50
2	2	0
3	3	-50
4	3	-50

Group 별 선수 리스트 vector or list

gid	pids
1	1
2	2
3	3, 4

updateScore() 최적화

그룹단위로 점수가 업데이트 되므로 그룹점수 활용

updateScore()

gScore[gid]에 점수 업데이트
O(1 * 호출횟수)

getScore()

return score[pid] + gScore[gid]

unionTeam()

이동하는 플레이어 pid
기존 그룹 gid1, 이동 후 그룹 gid2

: score[pid] += gScore[gid1] – gScore[gid2]

Player 별 그룹번호, 점수

pid	group	score
1	1	0
2	2	0
3	3	0
4	3	0

Group 별 선수 리스트, 그룹점수

gid	pids	gScore
1	1	50
2	2	30
3	3, 4	-50

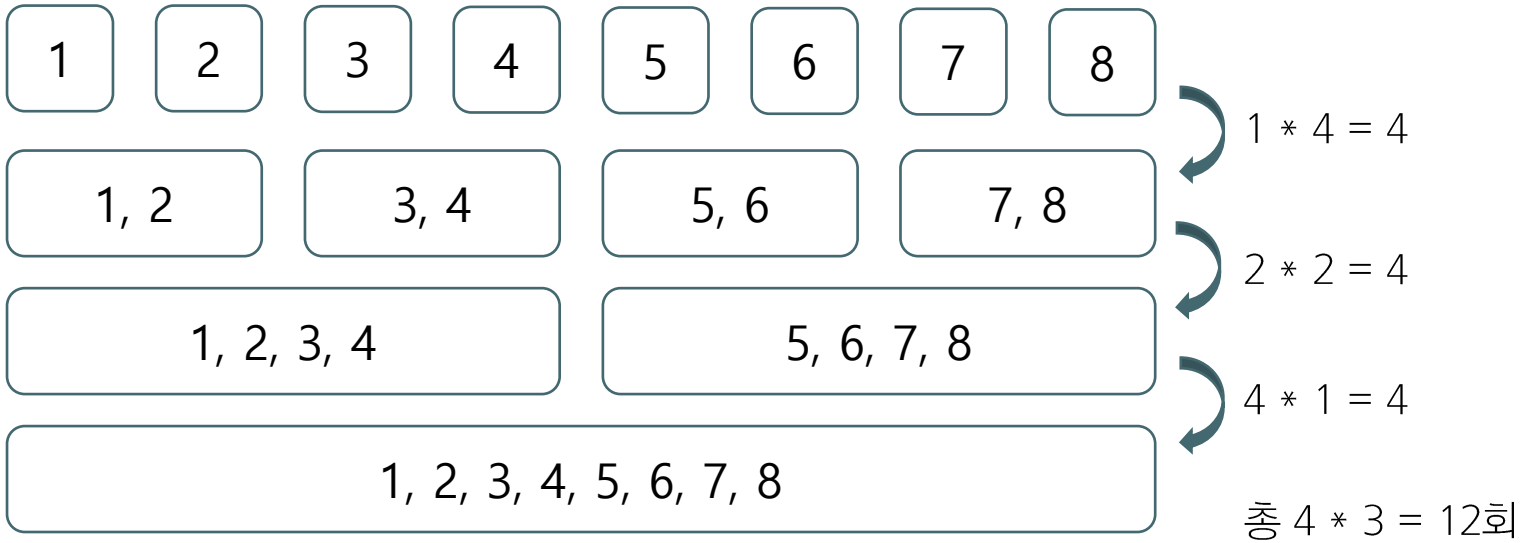
2->3 병합 시,
score[2] = 0 + 30 – (-50) = 80
group[2] = 3
점수 = score[2] + gScore[3] = 30

unionTeam() 최적화

1. 그룹 크기가 작은 그룹에서 큰 그룹으로 병합

이 방법만으로도 생각보다 상당히 적은 인원만 이동한다.
worst case 인 경우도 모두 병합되기까지 O(n log n) 으로 표현 가능하다.

worst case 예시



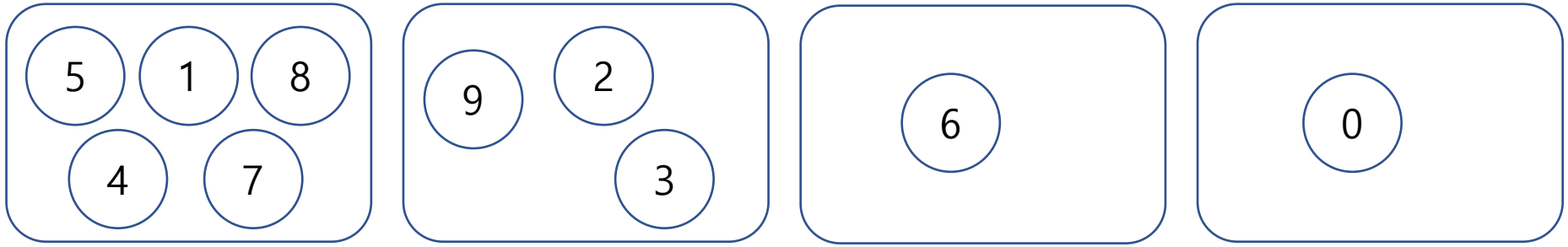
총 4 * 3 = 12회

: (N/2) log N

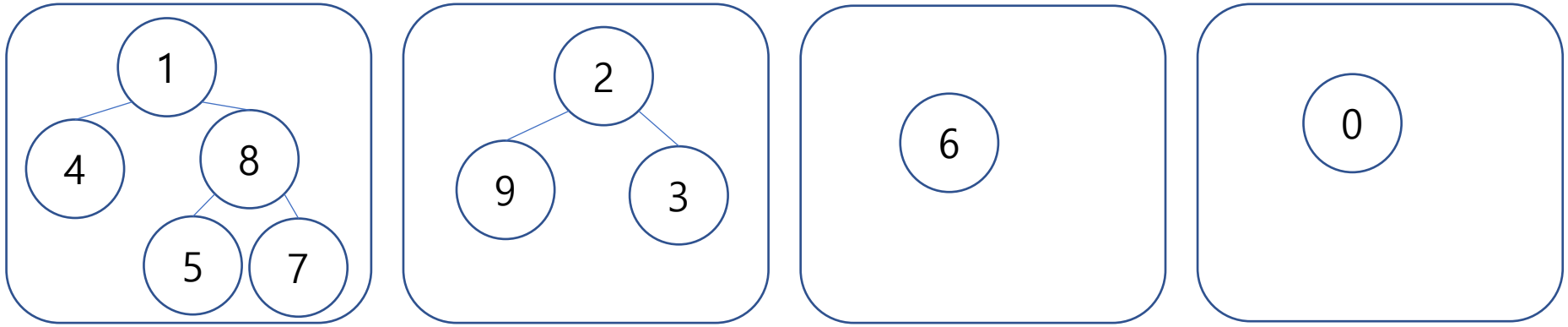
unionTeam() 최적화

2. Union-Find 알고리즘

Disjoint Set
: 서로소 집합 자료구조



Union-Find
: disjoint set을 트리로 표현하는 알고리즘 (앞서서는 vector, list로 sequence로 관리)

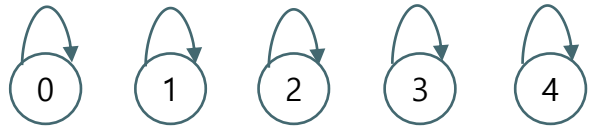


Union Find

```
const int LM = 100003;
int parent[LM], rank[LM];

void init() {
    for (int i = 0; i < N; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
}
```

```
int find(int x) {
    if (parent[x] == x) return x;
    return parent[x] = find(parent[x]);
}

void union(int x, int y) {
    x = find(x), y = find(y);
    if (x == y) return;

    if (rank[x] < rank[y]) {
        parent[x] = y;
    }
    else {
        parent[y] = x;
        if (rank[x] == rank[y]) rank[x]++;
    }
}
```

	0	1	2	3	4
parent	0	1	2	3	4
rank	0	0	0	0	0



union(x,y) : x, y가 속한 집합을 합침
union-by-rank 를 통해 최적화

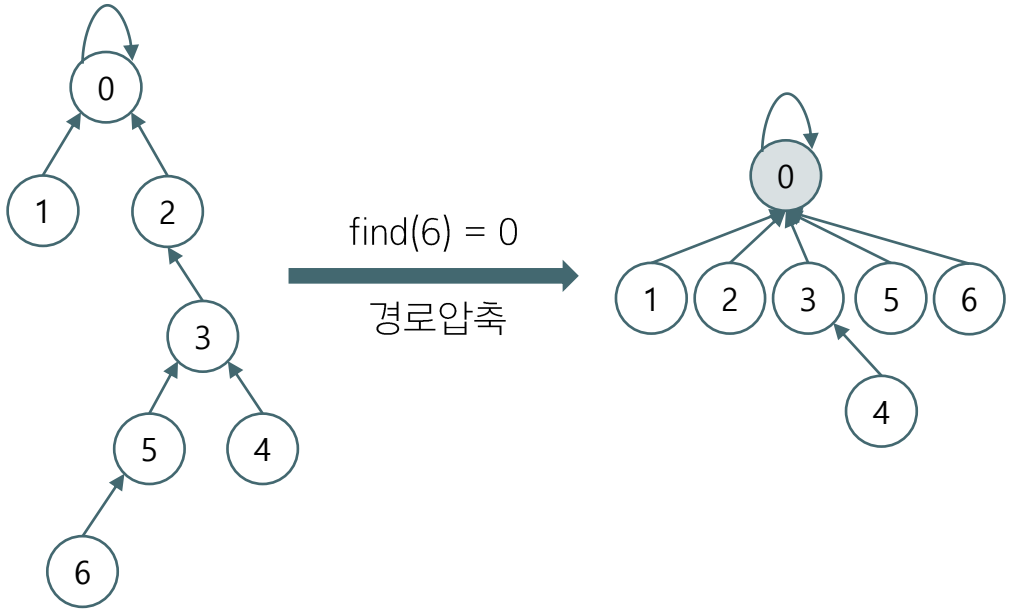
find(x) : x가 속한 집합의 root 노드를 반환
경로 압축을 통해 최적화

* 경로압축, union-by-rank 활용시, 상수시간으로 연산 가능
그룹의 root는 parent[x]=x 인 노드이며, 그룹의 모든 정보는 root에 기록
rank는 경로압축을 고려하지 않음

Union Find

```
const int LM = 100003;
int parent[LM], rank[LM];

void init() {
    for (int i = 0; i < N; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
}

int find(int x) {
    if (parent[x] == x) return x;
    return parent[x] = find(parent[x]);
}

void union(int x, int y) {
    x = find(x), y = find(y);
    if (x == y) return;

    if (rank[x] < rank[y]) {
        parent[x] = y;
    }
    else {
        parent[y] = x;
        if (rank[x] == rank[y]) rank[x]++;
    }
}
```

	0	1	2	3	4	5	6
parent	0	0	0	2	3	3	5
rank	4						

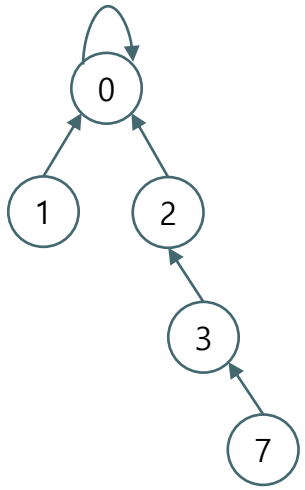


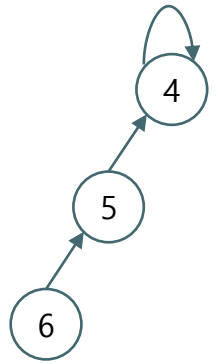
Union Find

	0	1	2	3	4	5	6	7
parent	0	0	0	2	4	4	5	3
rank	3				2			

```
const int LM = 100003;
int parent[LM], rank[LM];

void init() {
    for (int i = 0; i < N; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
}

int find(int x) {
    if (parent[x] == x) return x;
    return parent[x] = find(parent[x]);
}

void union(int x, int y) {
    x = find(x), y = find(y);
    if (x == y) return;

    if (rank[x] < rank[y]) {
        parent[x] = y;
    }
    else {
        parent[y] = x;
        if (rank[x] == rank[y]) rank[x]++;
    }
}
```

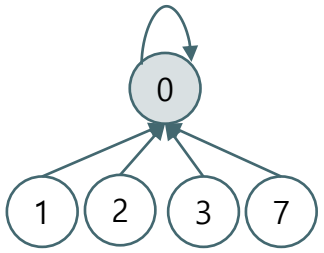
union(7, 6)

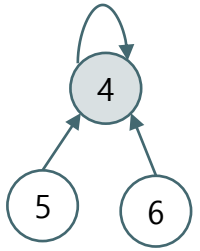




x = find(7) = 0

y = find(6) = 4

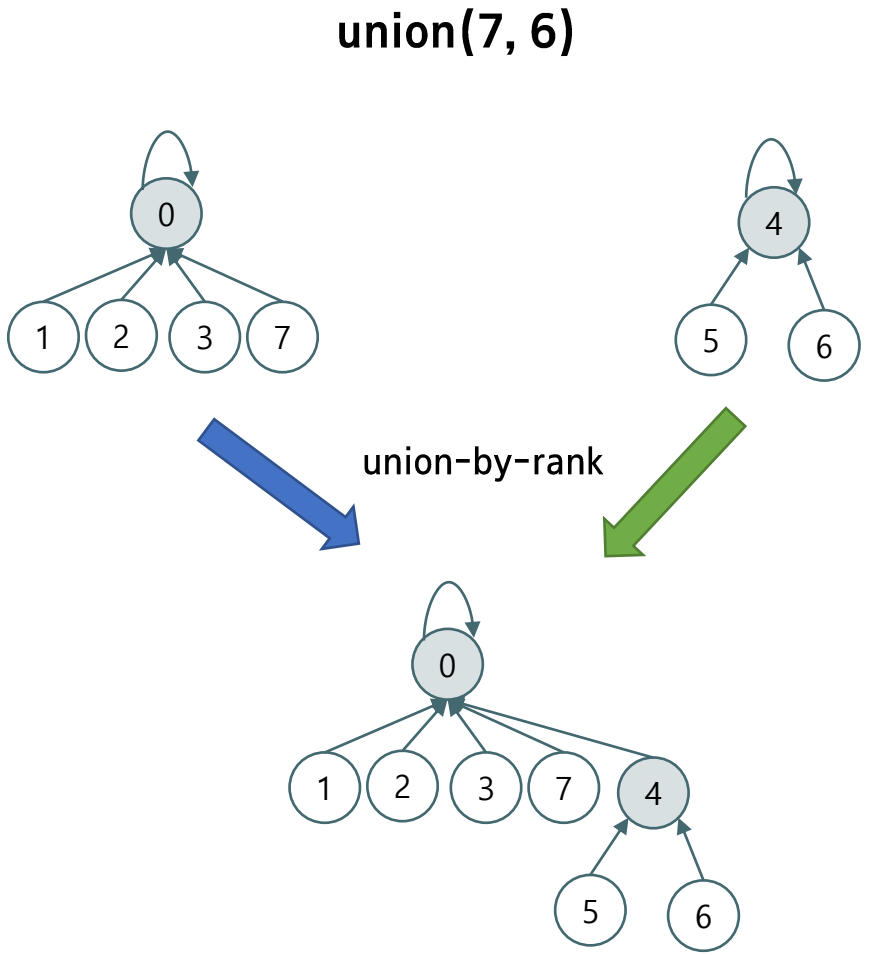




Union Find

	0	1	2	3	4	5	6	7
parent	0	0	0	0	0	4	4	0
rank	3							

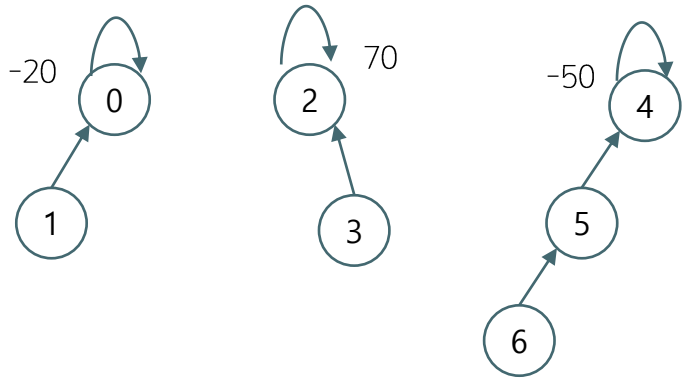
```
const int LM = 100003;
int parent[LM], rank[LM];

void init() {
    for (int i = 0; i < N; i++) {
        parent[i] = i;
        rank[i] = 0;
    }
}

int find(int x) {
    if (parent[x] == x) return x;
    return parent[x] = find(parent[x]);
}

void union(int x, int y) {
    x = find(x), y = find(y);
    if (x == y) return;

    if (rank[x] < rank[y]) {
        parent[x] = y;
    }
    else {
        parent[y] = x;
        if (rank[x] == rank[y]) rank[x]++;
    }
}
```



unionTeam() 최적화

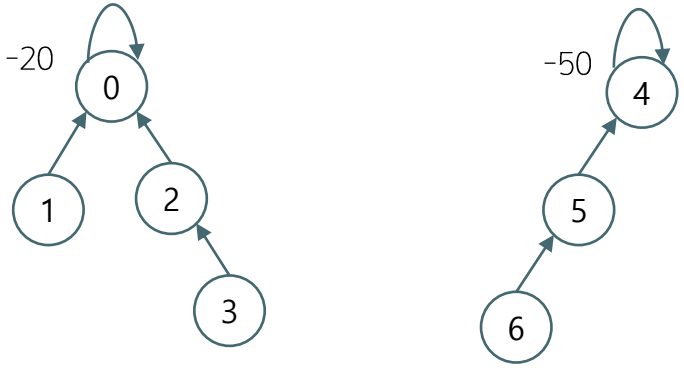
2. Union-Find 알고리즘

	0	1	2	3	4	5	6
parent	0	0	2	2	4	4	5
rank	1		1		2		
gScore	-20		+70		-50		
score	0	0	0	0	0	0	0



union(0, 2)

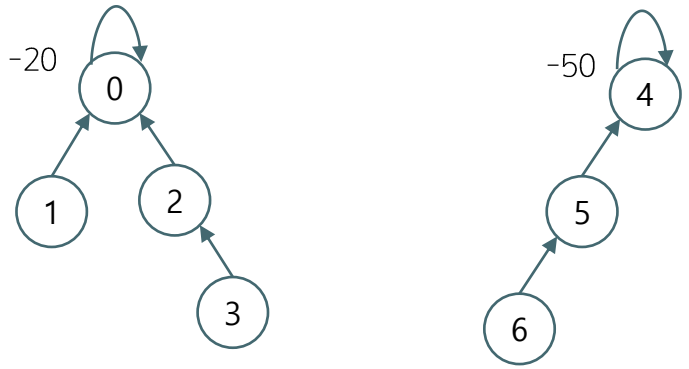
	0	1	2	3	4	5	6
parent	0	0	0	2	4	4	5
rank	2		1		2		
gScore	-20		+70		-50		
score	0	0	90	0	0	0	0



unionTeam() 최적화

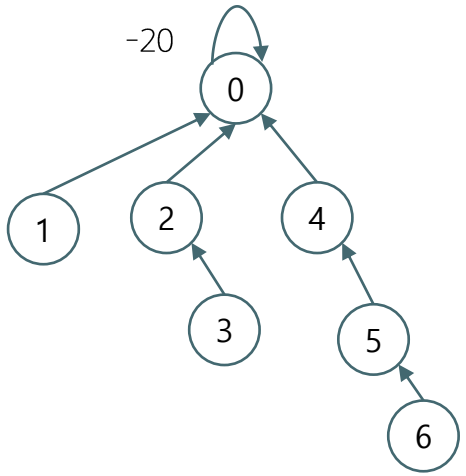
2. Union-Find 알고리즘

	0	1	2	3	4	5	6
parent	0	0	0	2	4	4	5
rank	2				2		
gScore	-20				-50		
score	0	0	90	0	0	0	0



union(0, 4)

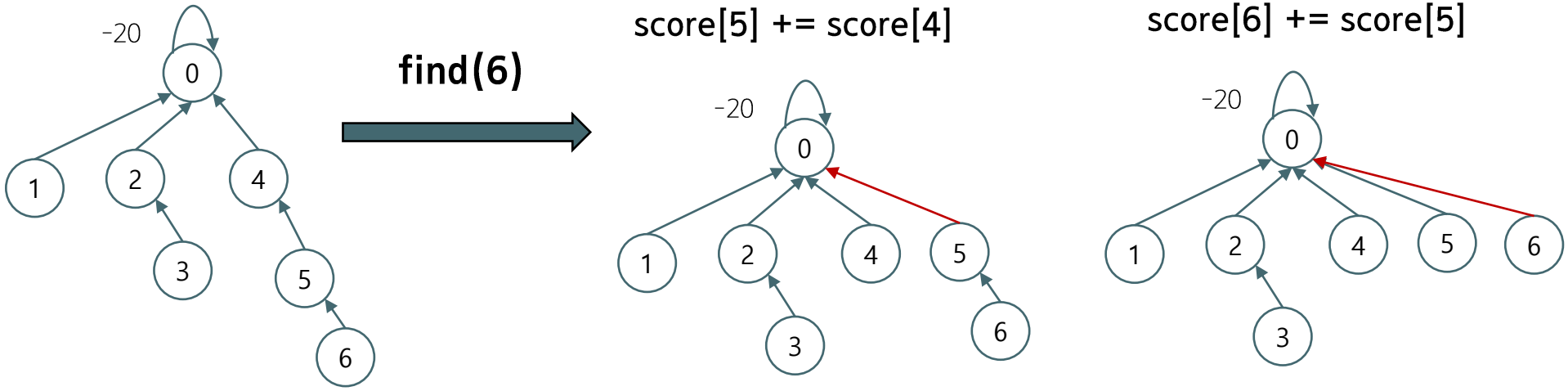
	0	1	2	3	4	5	6
parent	0	0	0	2	0	4	5
rank	3				2		
gScore	-20				-50		
score	0	0	90	0	-30	0	0



unionTeam() 최적화

2. Union-Find 알고리즘

	0	1	2	3	4	5	6
parent	0	0	0	2	0	0	0
rank	3						
gScore	-20						
score	0	0	90	0	-30	-30	-30



unionTeam() 최적화

2. Union-Find 알고리즘

updateScore()

gScore[root]에 점수 업데이트

getScore()

root = find(pid)
return score[pid] + gScore[root]

unionTeam()

root1 = find(pid1)
root2 = find(pid2)
root1 을 rank 큰 값으로 설정
parent[root2] = root1
score[root2] = gScore[root2] – gScore[root1]
rank가 같으면 rank[root1]++

find()

경로 압축 하는 과정에서
x가 root로 바뀌는 경우, score[x] += score[parent[x]]


감사합니다