

AVL Tree \iff ① BST

and ② 각 노드의

balance factor

$$\in \{-1, 0, 1\}$$

and ③ balance factor

= height of left subtree

- height of right subtree

AVL 트리의 삽입 Key Takeaway:

- 1) 삽입할 위치에 삽입한다 (BST와 동일)
- 2) 삽입후에도 AVL 트리가 맞는지 확인한다.
(삽입된 노드부터 부모로 올라가며 bf 계산)
→ AVL 트리 맞았을 break
- 3) bf가 깨진노드가 있다면
회전당할 노드 3개를 선정한다.

4) 회전시킨다.

① 회전당할 노드 3개 중 중앙값을 가진 노드를

부모가진 노드 위치로 옮기고

나머지 2개 중 작은값을 좌자식, 큰값을 우자식 차례로
옮긴다.

② 회전당할 노드 3개의 자식노드들이 있다면

그들의 관계를 킷손사기지 않는 위치로 이동시킨다.

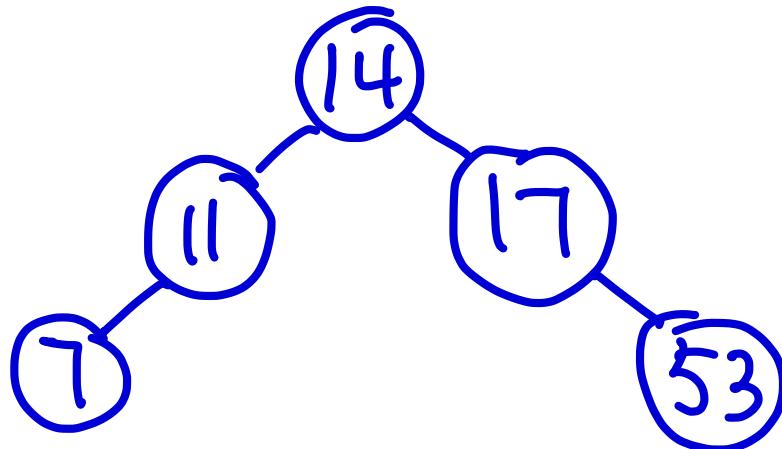
5) 회전 후에 AVL 트리가 맞는지 확인한다.

(트리 노드의 bf 계산)

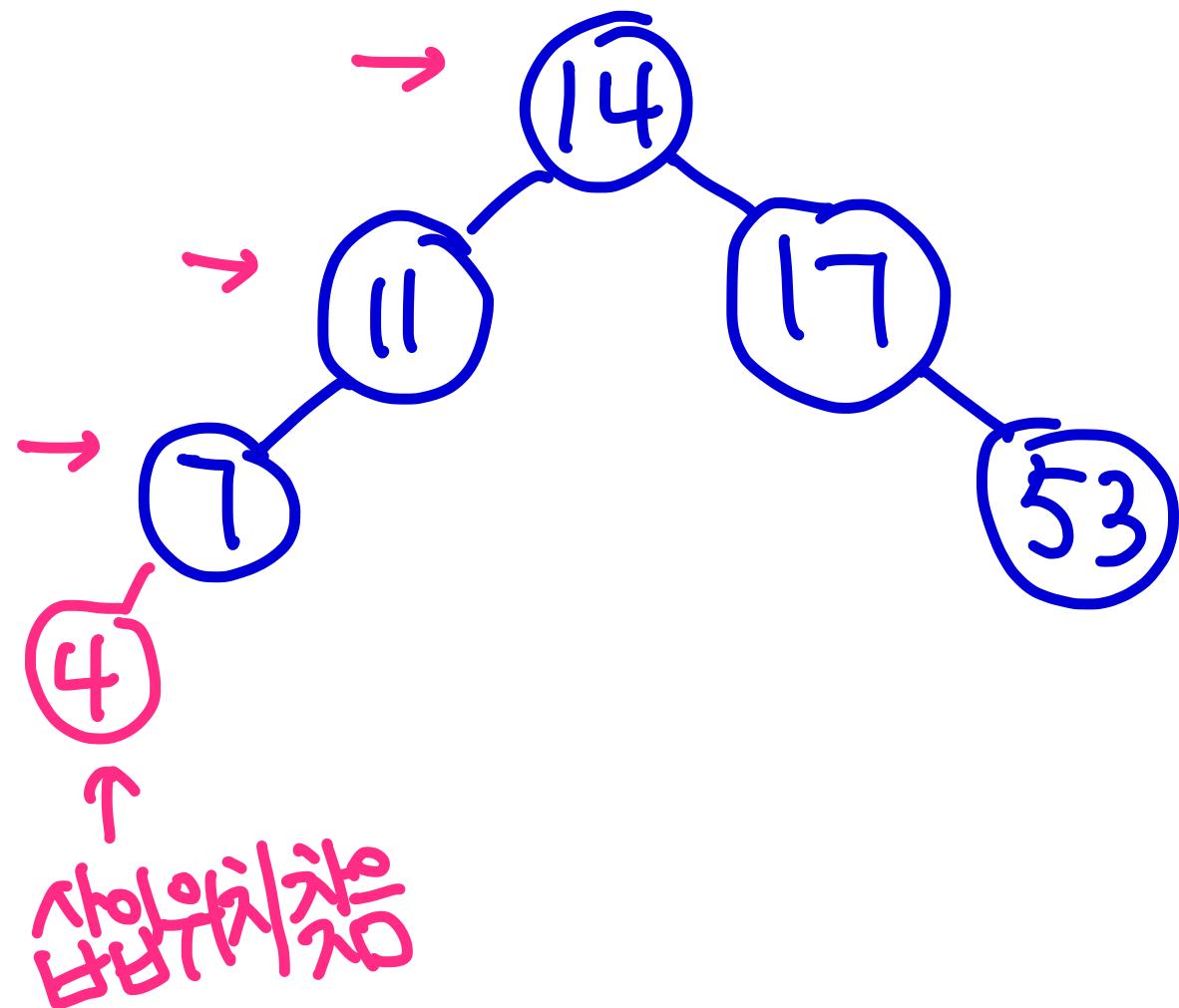
AVL 트리 맞으면 break

아니면 2)로 가서 AVL 트리가 뒤집어지지 않도록 한다

삽입 위치
[L Rota]

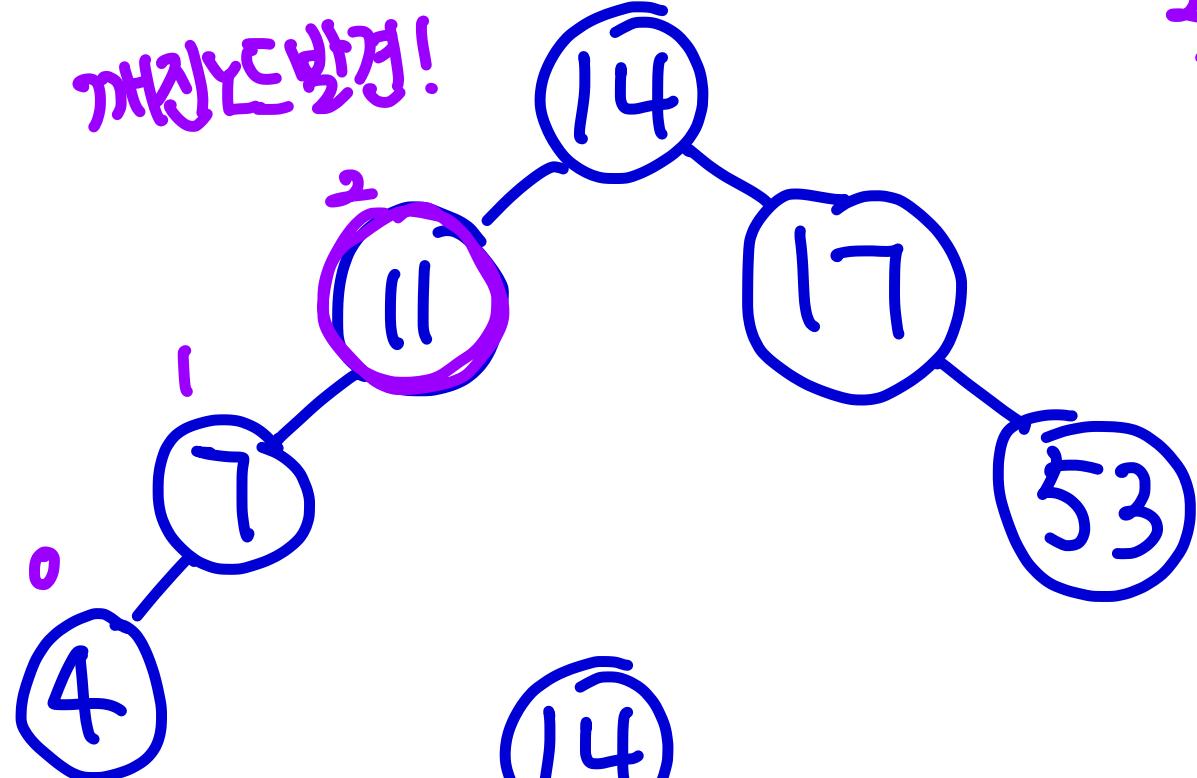


여기서 4삽입

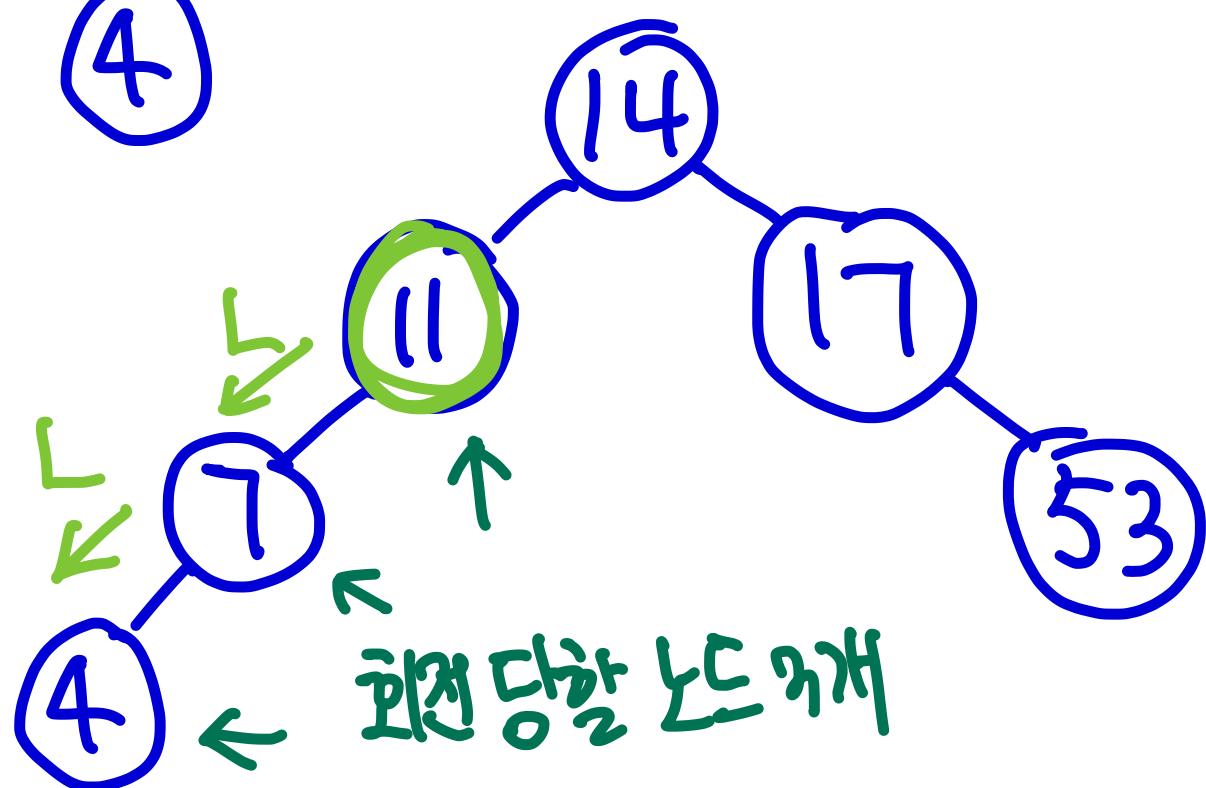


1) BST처럼 삽입위치
찾아내서 가기

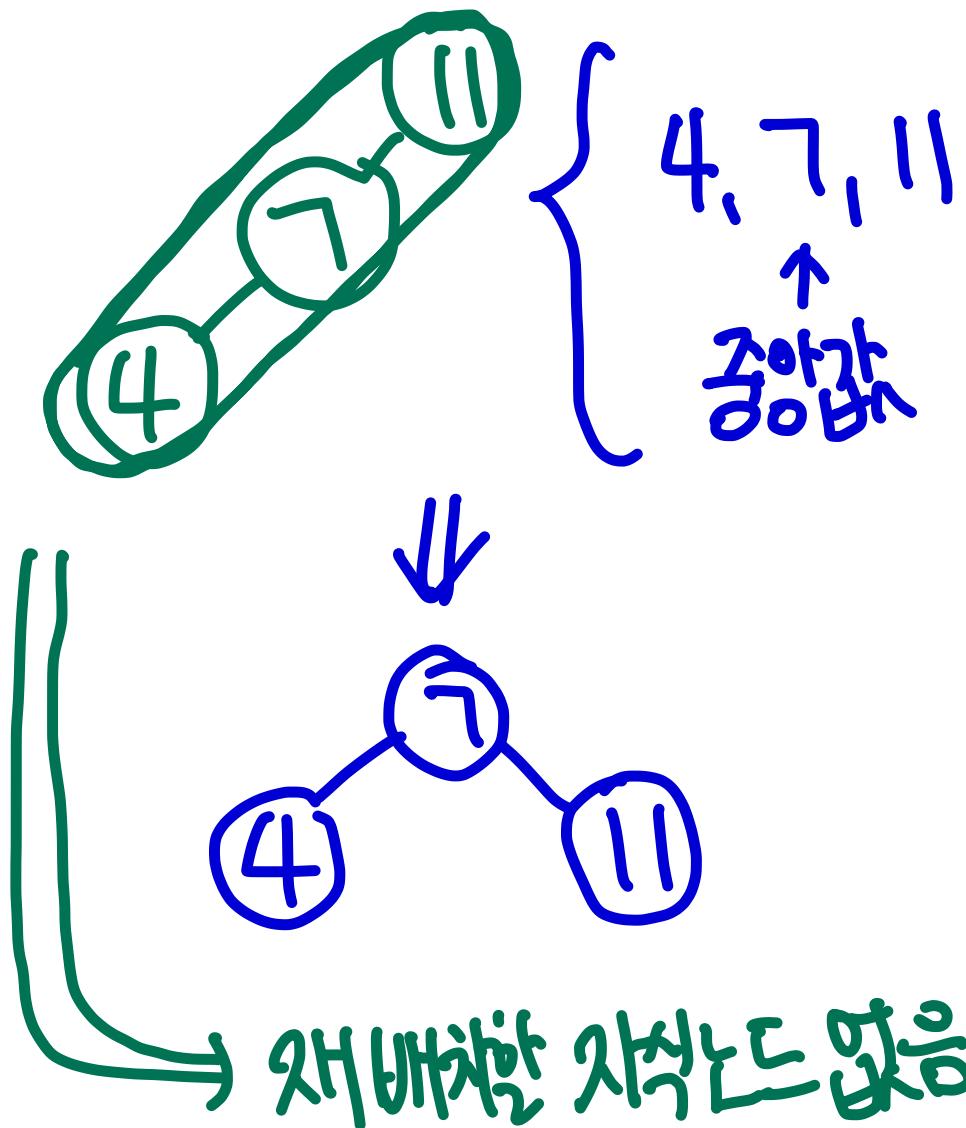
까진노드발견!



2) 삽입된 노드부터 부모로 올라가며 balance factor 계산하여 bf 까진노드가 있나 확인



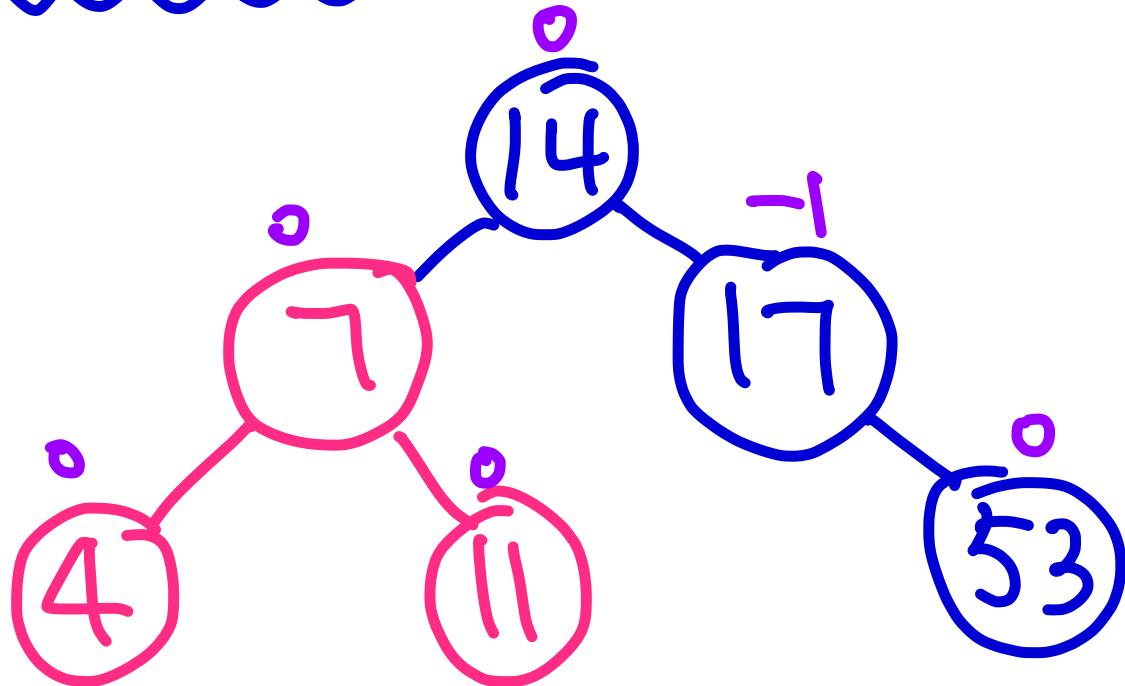
3) bf가 까진노드가 있다면 회전을 해야 함.
⇨ 회전의 대상 노드 3개는 무엇인가? 까진노드부터 LL
⇨ 11, 7, 4



- 4) 회전
- ① Rotation 대상
서 노드의 중앙값을 루트로
 - ② 자식 노드들을 재배치

⇒ 회전 완료

LL Rot



5) 회전 완료 후

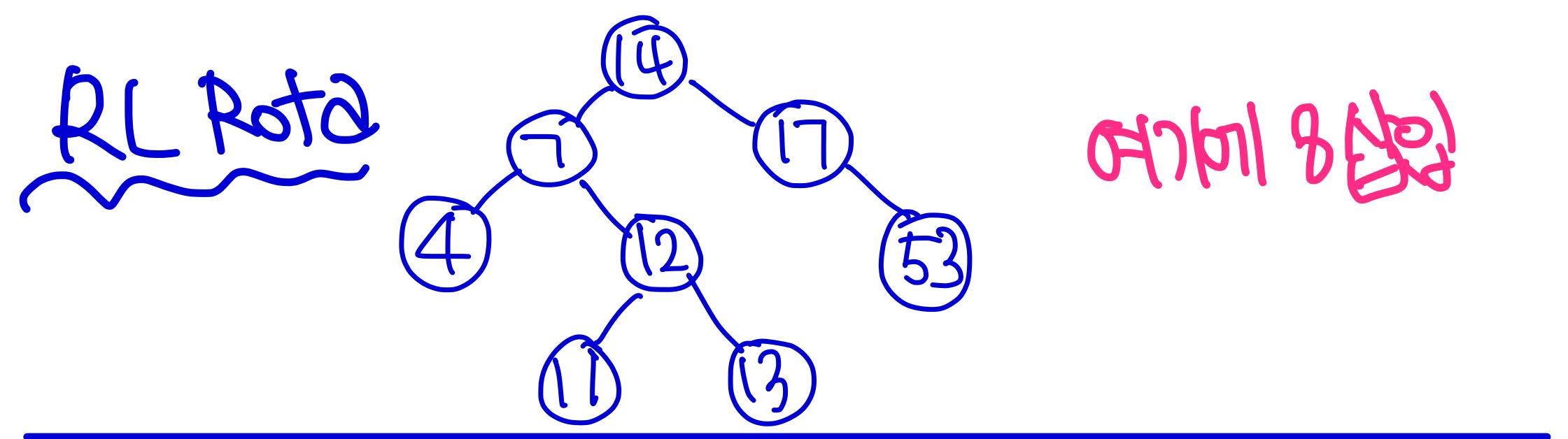
전체 bf 확인.

⇒ bf가 깨진 노드가 존재하면
2)로 돌아가서 다시 회전
⇒ bf가 깨진 노드가 없으면 삽입을

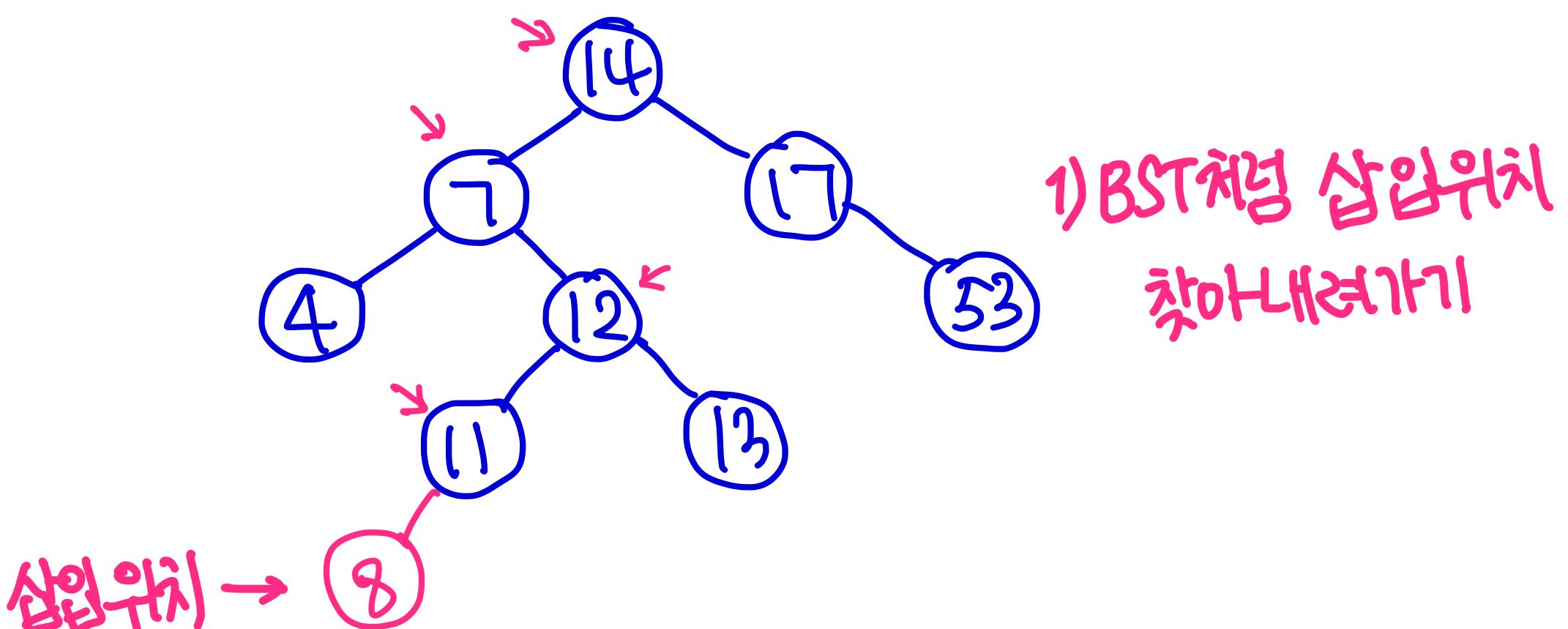
⇒ 전체 bf 확인

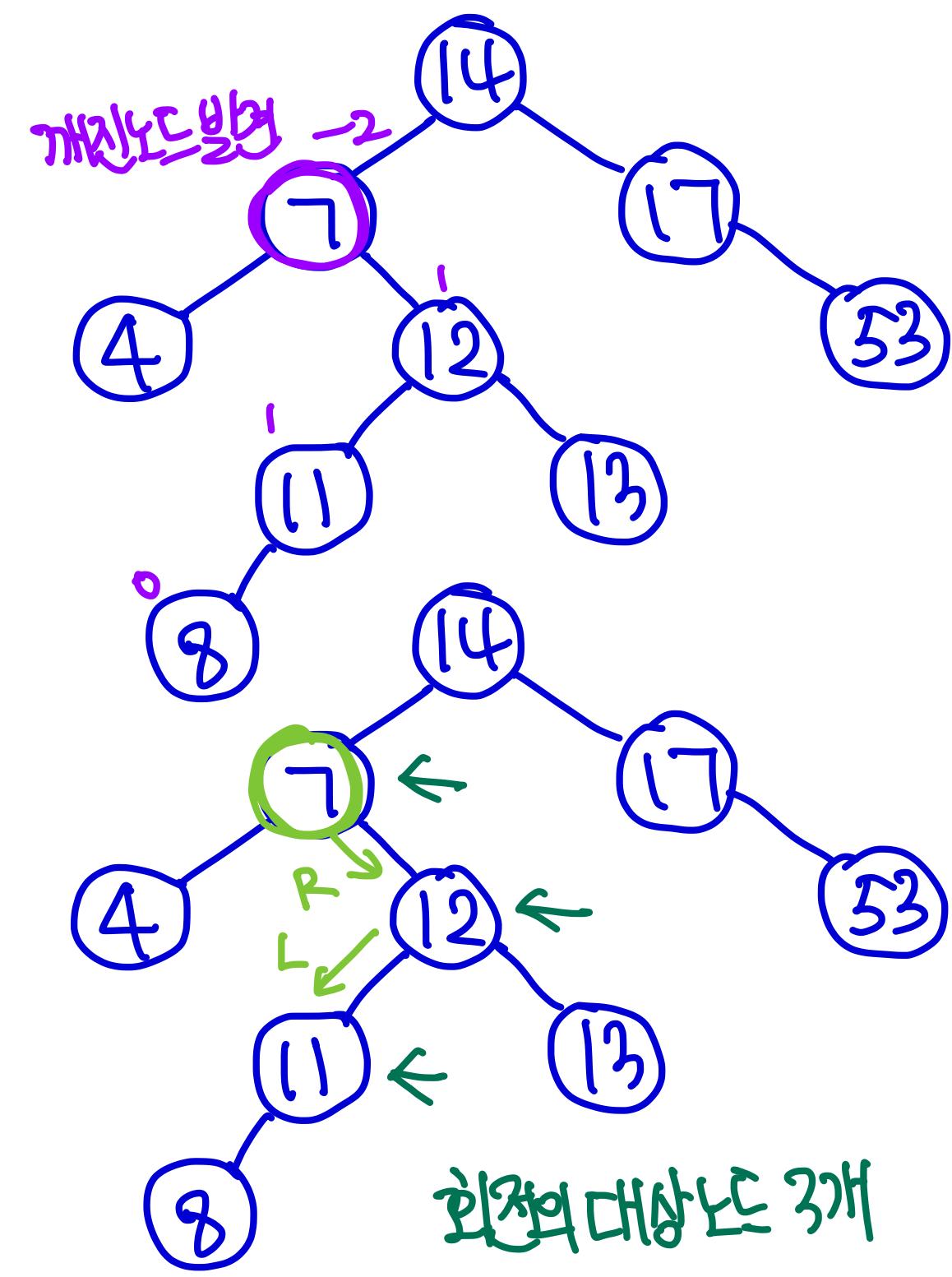
⇒ AVL Tree 맞음 (모든 $bf \in \{-1, 0, 1\}$)

⇒ 4삽입완료!



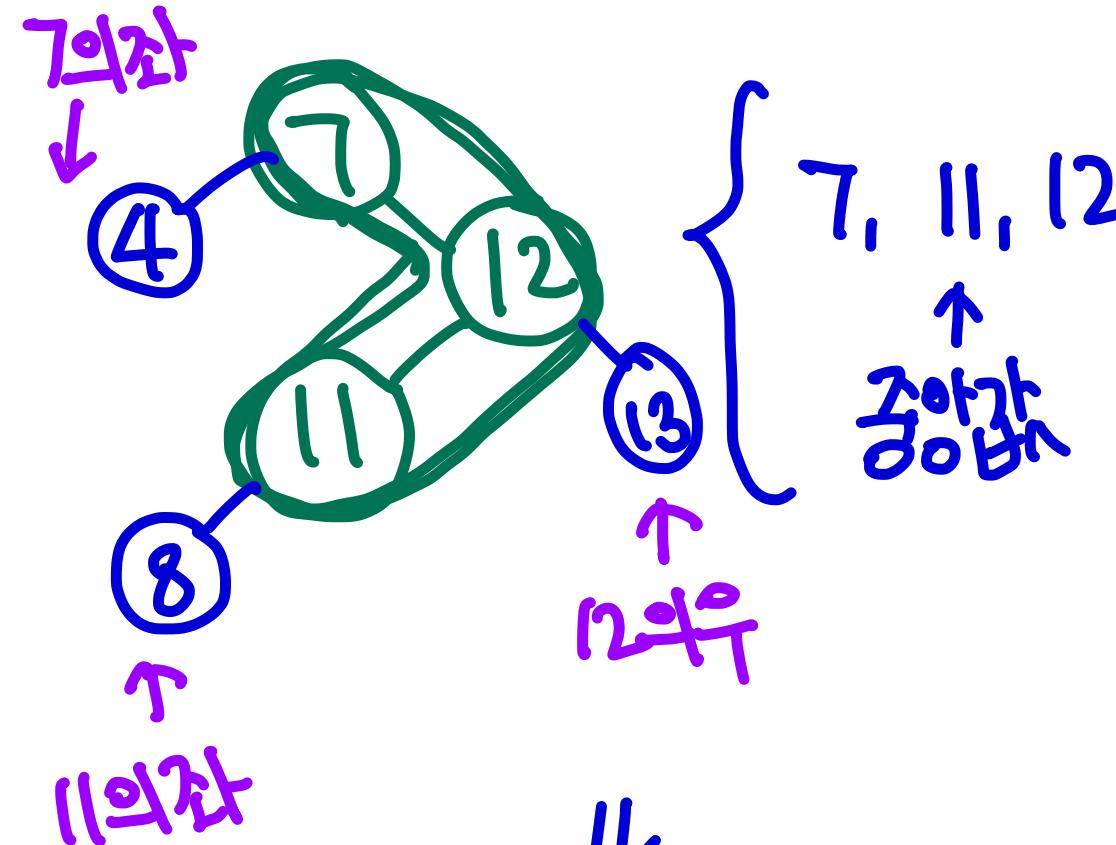
여기(에 8 삽입)





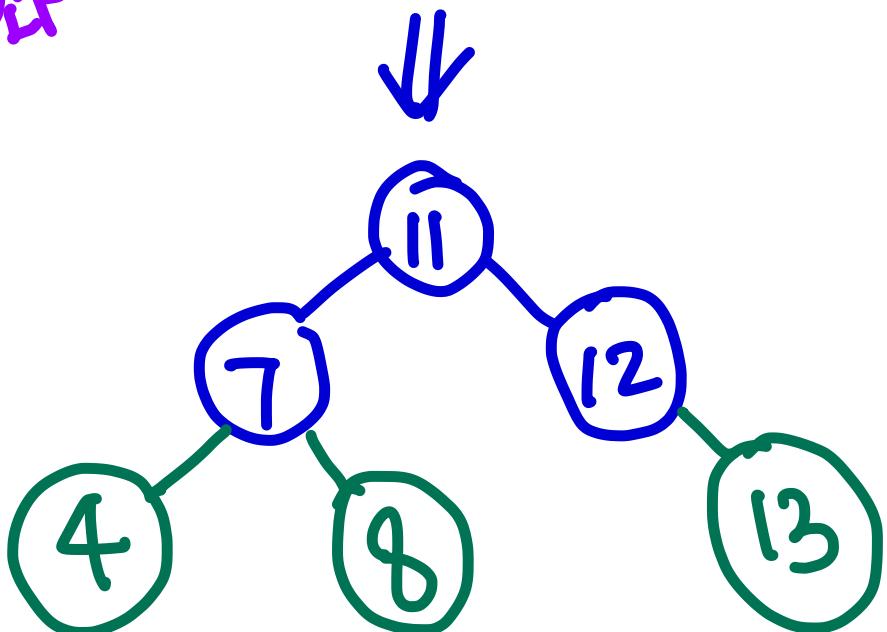
2) 삽입된 노드부터
부모로 올라가며
balance factor 계산하여
bf 개진노드가 있나 확인

3) bf가 개진노드가 있다면
회전을 해야 함.
⇨ 회전의 대상 노드 3개는
무엇인가? 개진노드부터 RL
 $\Rightarrow 7, 12, 11$

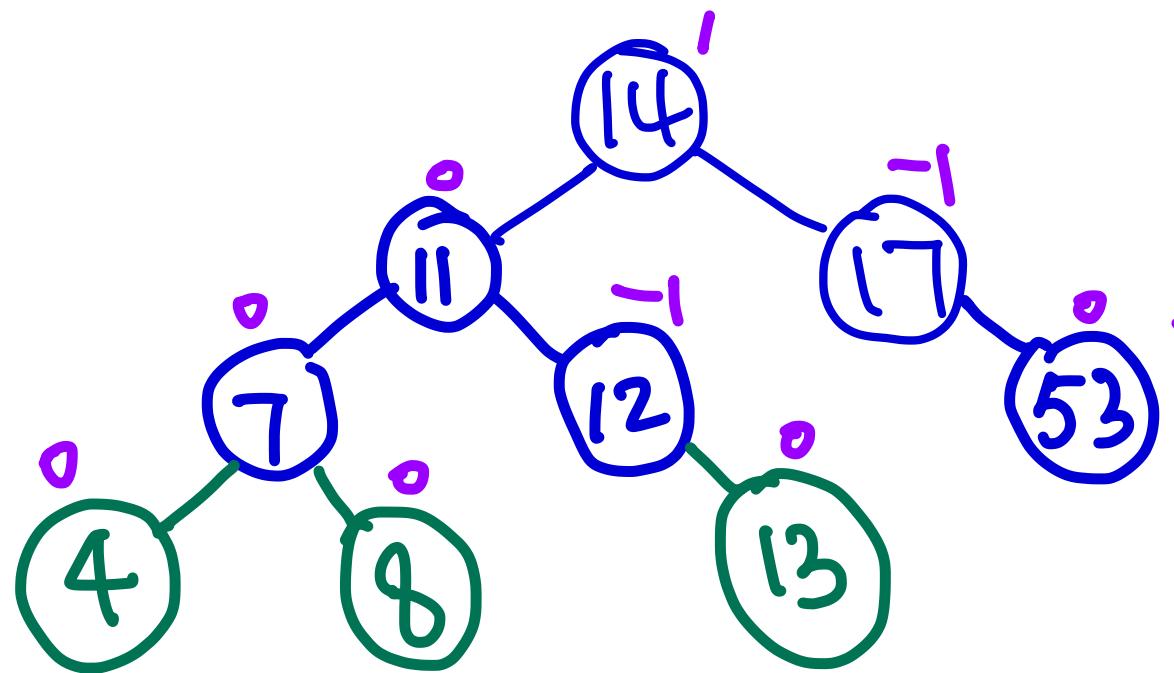


4) 회전

- ① Rotation 대상
서 노드의 중앙값을 루트로
- ② 자식 노드들을 재배치



① 자식 노드들 재배치 완료
⇒ 회전 완료



5) 회전 완료 후

전체 bf 확인.

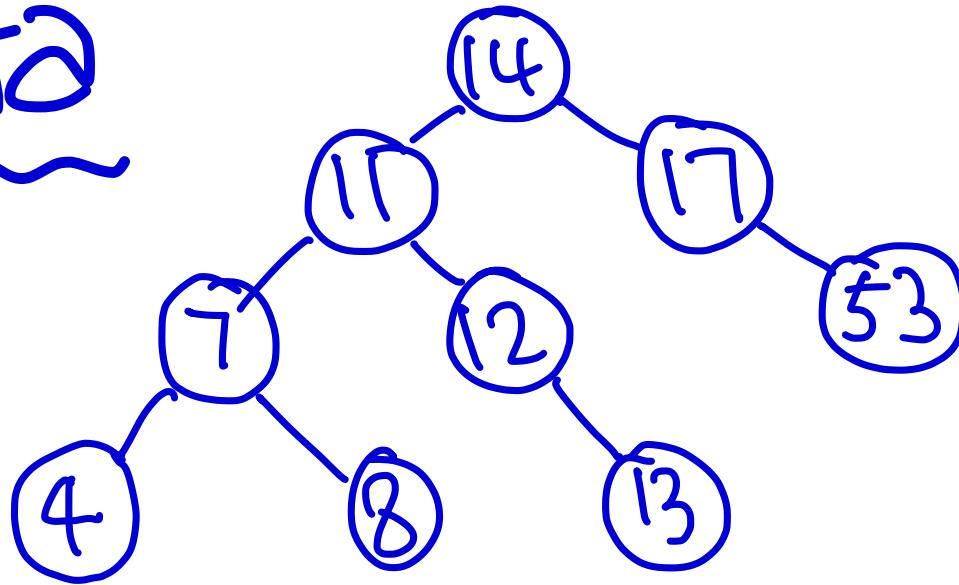
- ⇒ bf가 깨진 노드가 존재하면
2)로 돌아가서 다시 확인
- ⇒ bf가 깨진 노드가 없으면 삽입을

⇒ 전체 bf 확인

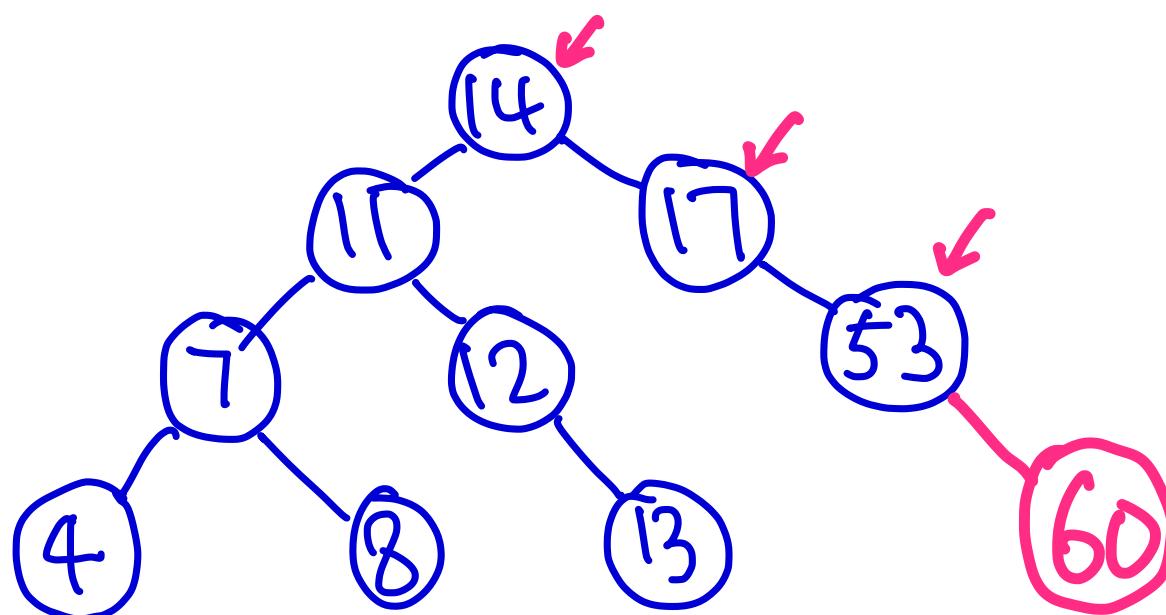
⇒ AVL Tree 맞음 (모든 $bf \in \{-1, 0, 1\}$)

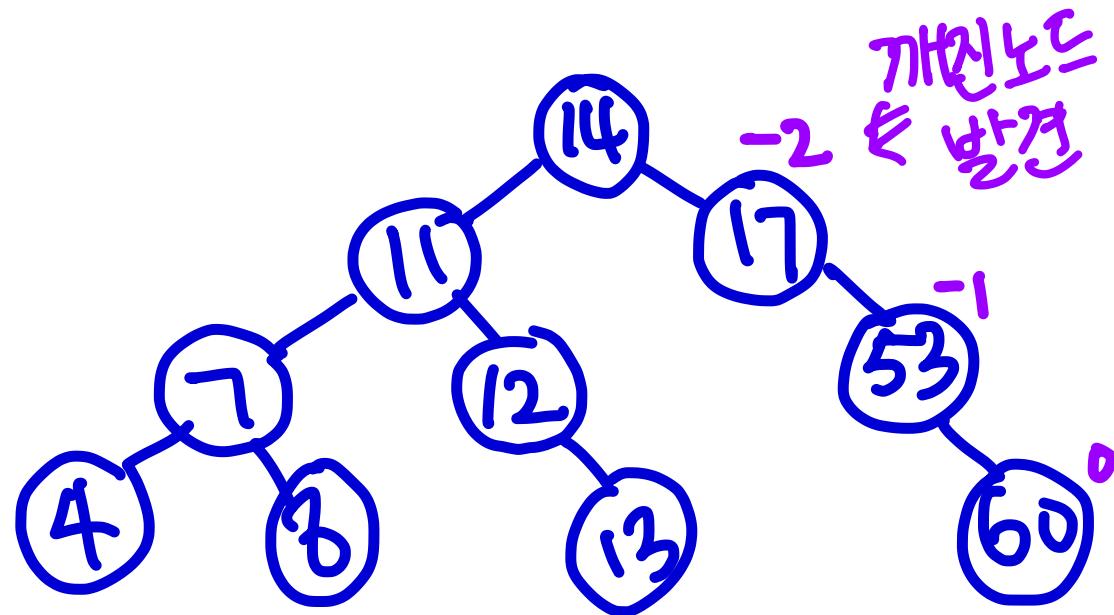
⇒ 8 삽입 완료!

RR Rota

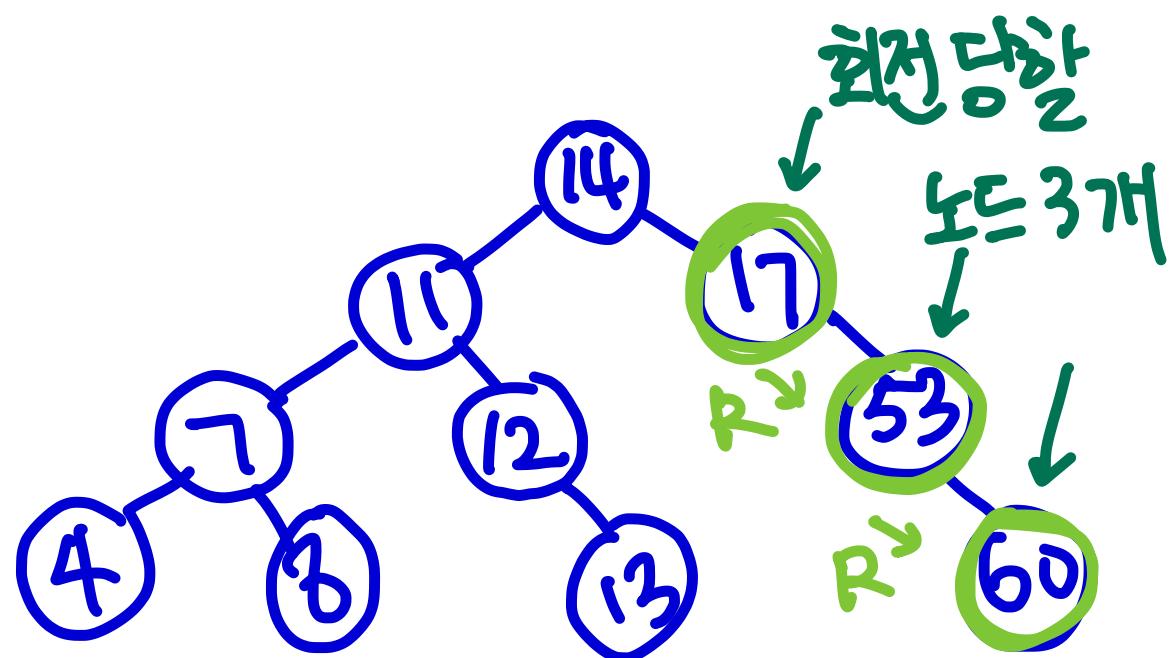


여기서 60 삽입

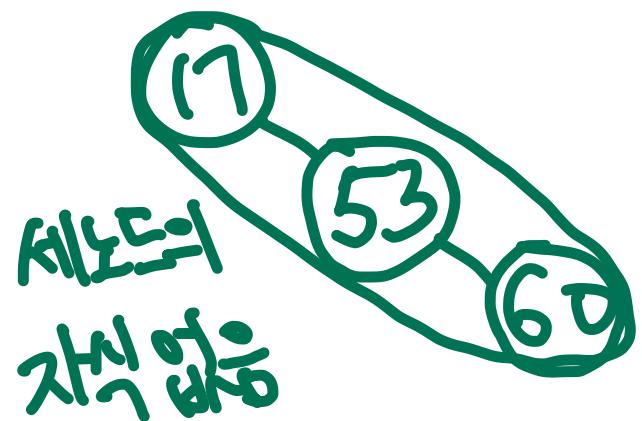




2) 삽입된 노드부터
부모로 올라가며
balance factor 계산하여
bf 개진노드가 있나 확인



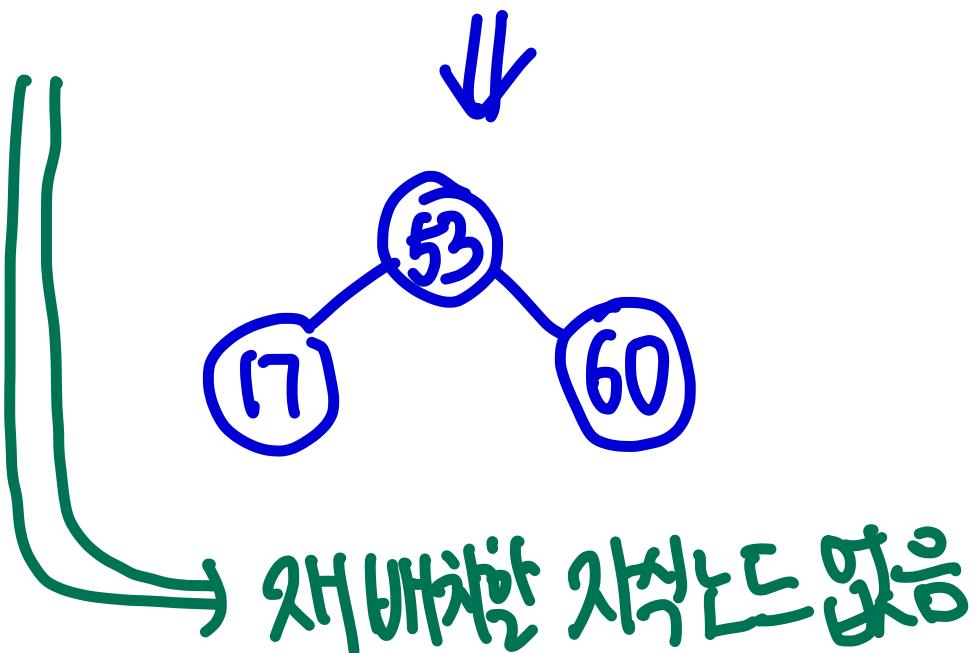
3) bf가 개진노드가 있다면
회전을 해야 함.
 ↗ 회전의 대상 노드 3개는
무엇인가? 개진노드부터 RR
 $\Rightarrow 17, 53, 60$



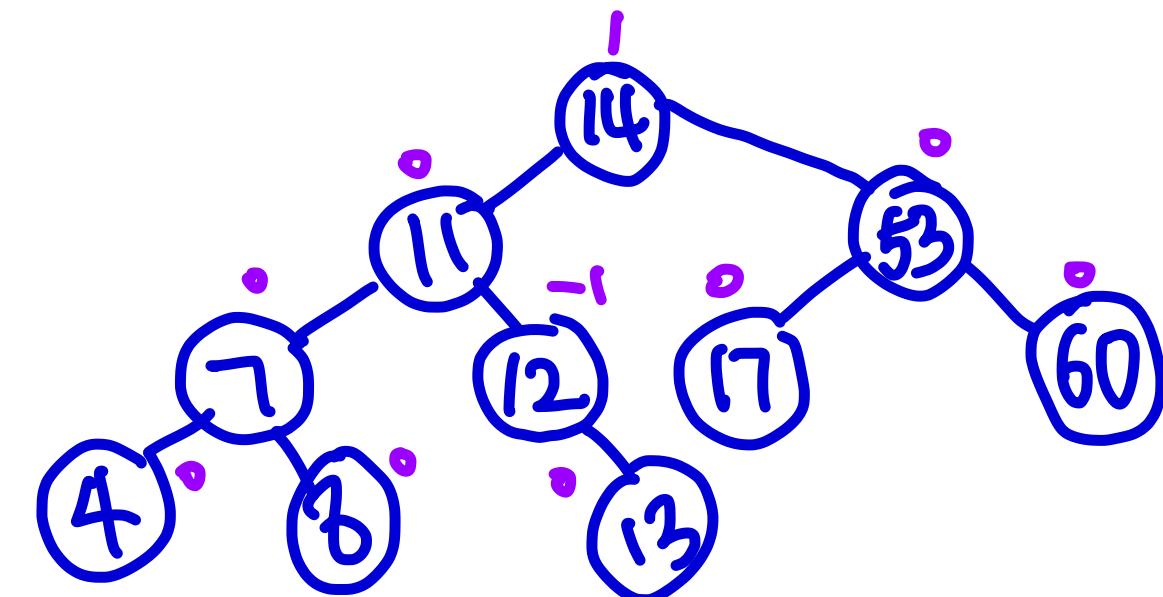
{ 17, 53, 60
↑
중앙값 }

4) 회전

- ① Rotation 대상
세노드의 중앙값을 루트로
- ② 자식노드들을 재배치



⇒ 회전완료



5) 회전 완료 후

전체 bf 확인.

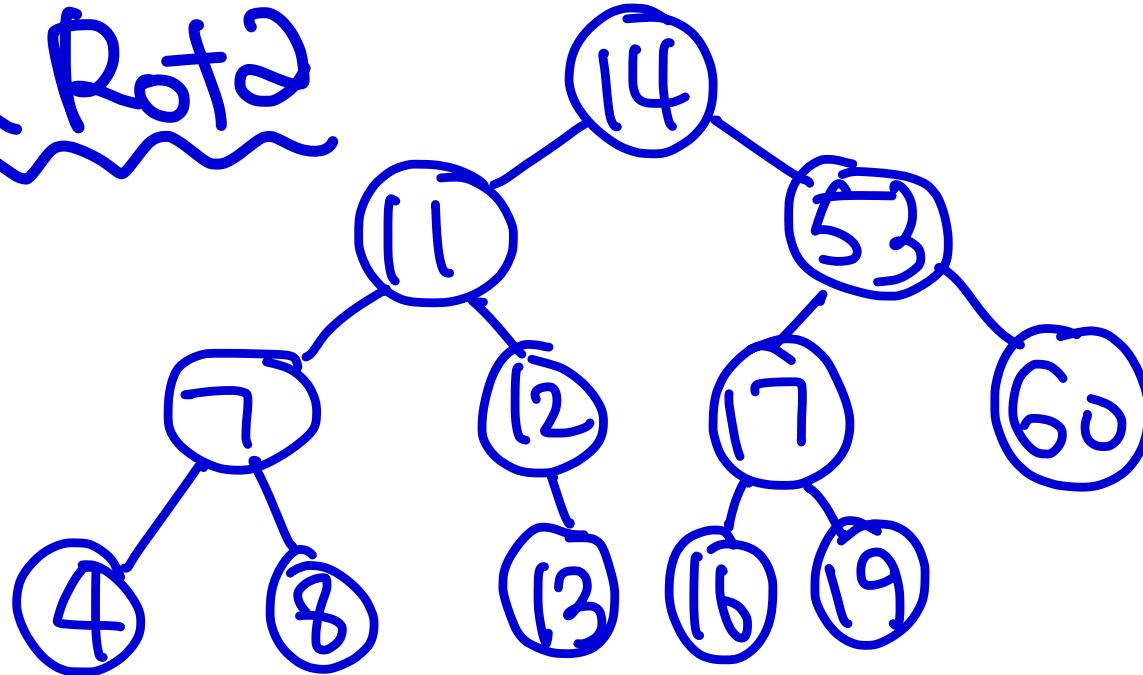
- ⇒ bf가 깨진 노드가 존재하면
2)로 돌아가서 다시 확인
- ⇒ bf가 깨진 노드가 없으면 삽입을

⇒ 전체 bf 확인

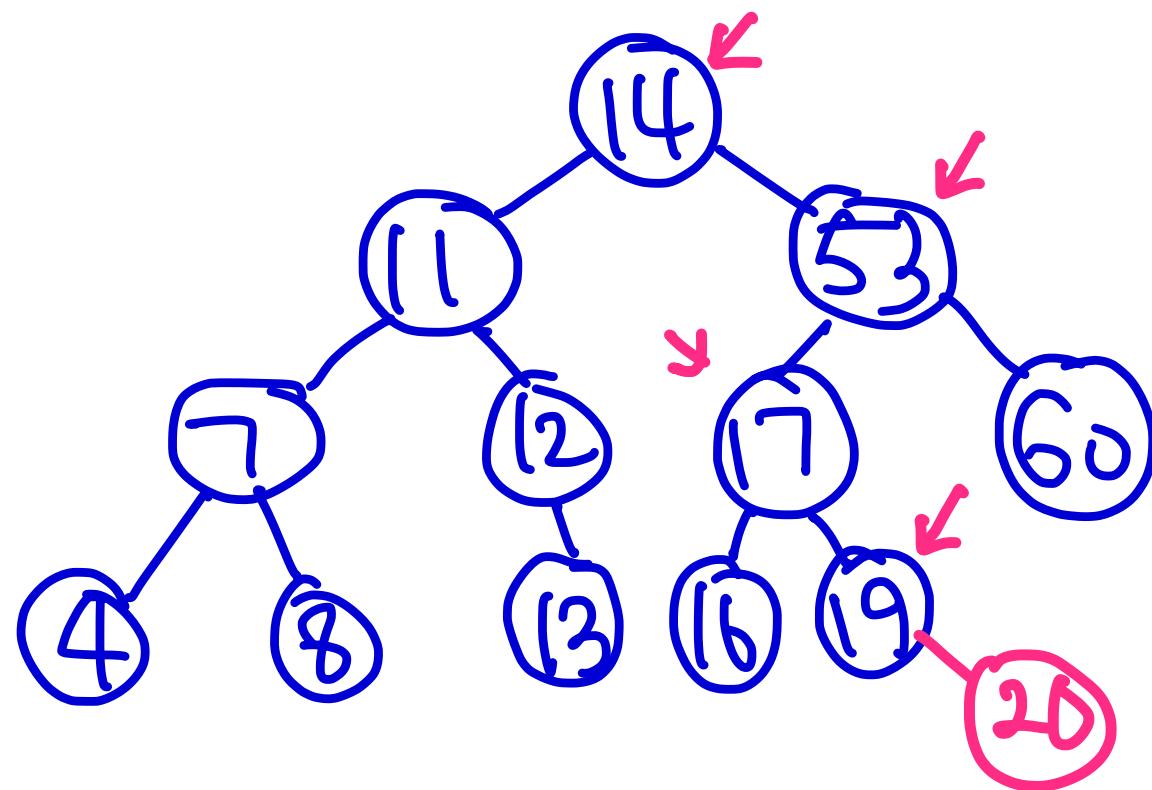
⇒ AVL Tree 맞음 (모든 $bf \in \{-1, 0, 1\}$)

⇒ 60 삽입 완료!

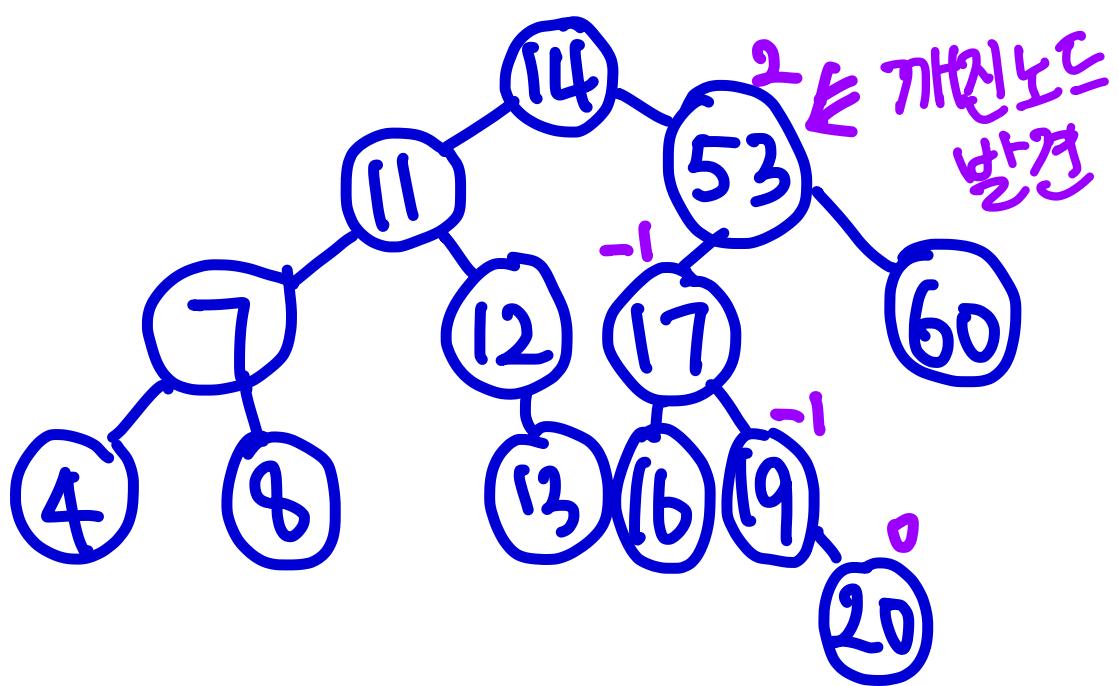
LR Rotate



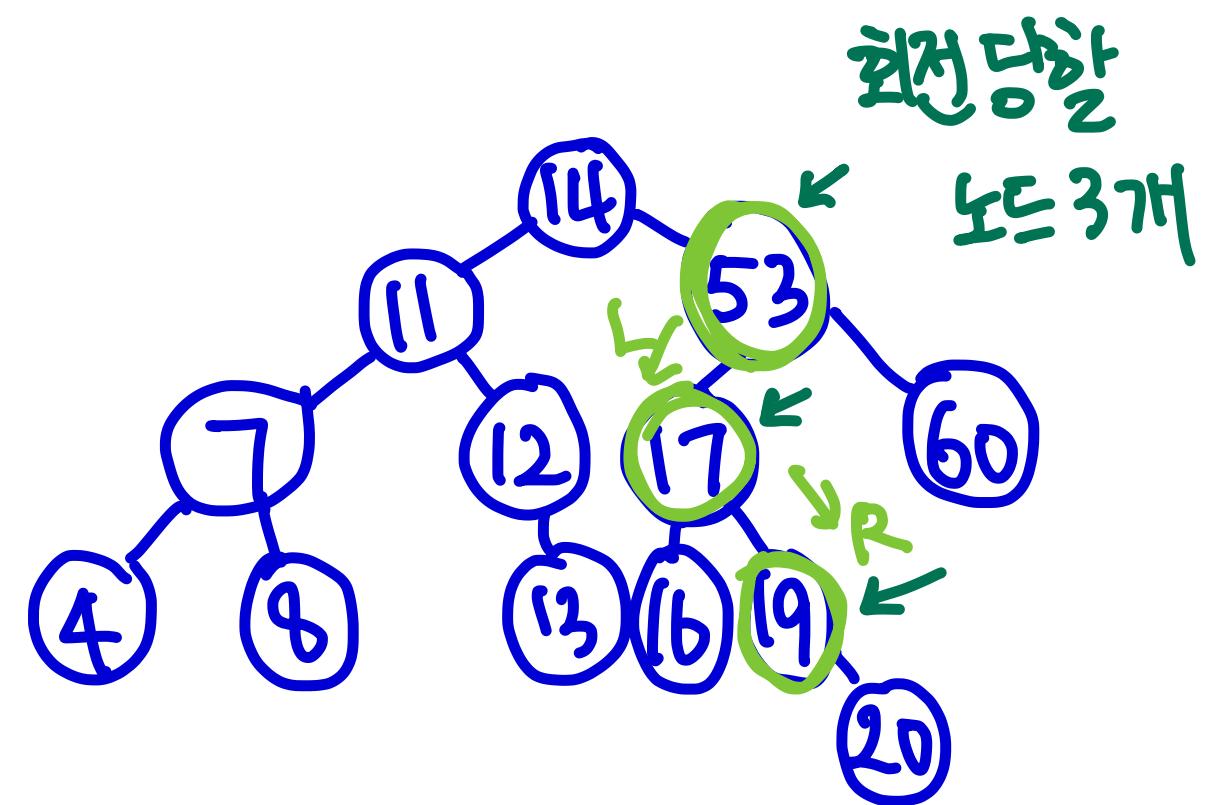
V�가지 20상



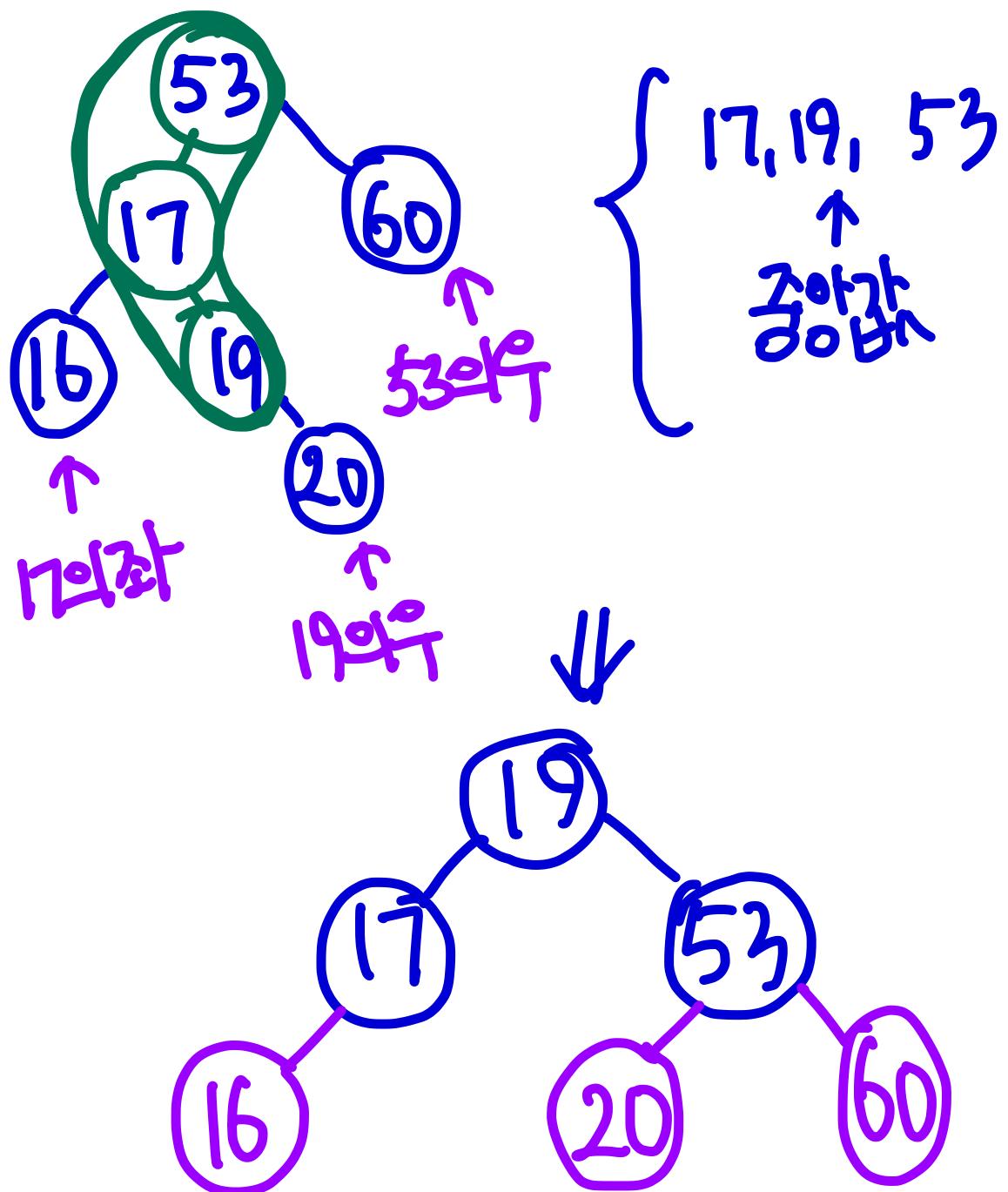
1) BST처럼 삽입위치
찾아내려면간후삽입



2) 삽입된 노드부터
부모로 올라가며
balance factor 계산하여
bf 까진노드가 있나 확인



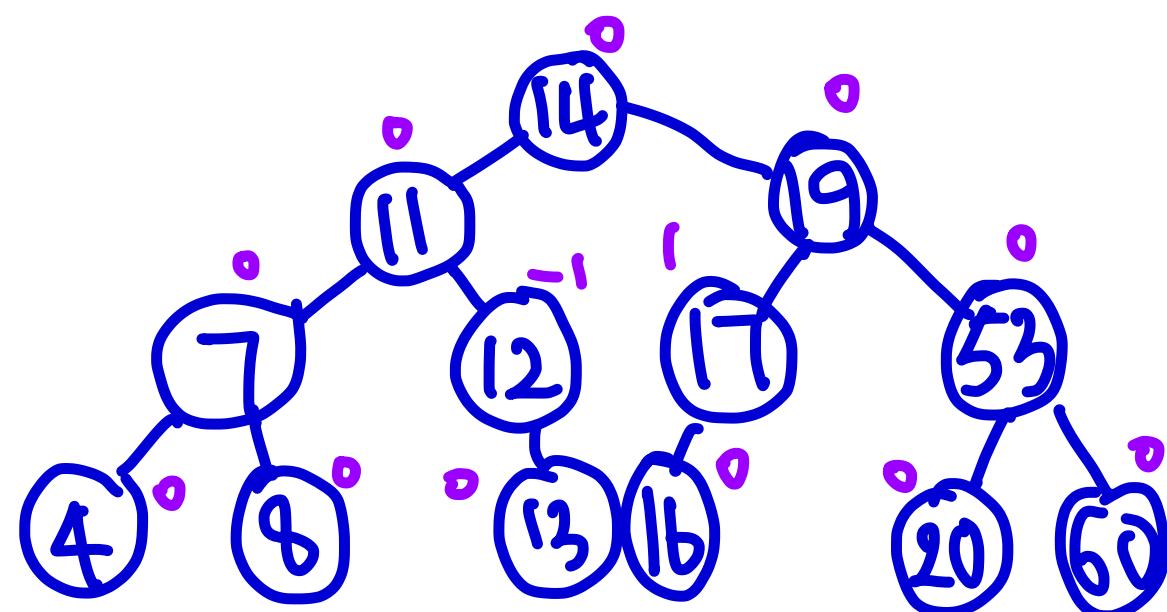
3) bf가 까진노드가 있다면
회전을 해야 함.
⇨ 회전의 대상 노드 3개는
무엇인가? 까진노드부터 LR
 $\Rightarrow 53, 17, 19$



4) 회전

- ① Rotation 대상
서 노드의 중앙값을 루트로
- ② 자식 노드들을 재배치

// 자식 노드들 재배치 완료
⇒ 회전 완료



5) 회전 완료 후

전체 bf 확인.

- ⇒ bf가 깨진 노드가 존재하면
2)로 돌아가서 다시 확인
- ⇒ bf가 깨진 노드가 없으면 삽입을

⇒ 전체 bf 확인

⇒ AVL Tree 맞음 (모든 $bf \in \{-1, 0, 1\}$)

⇒ 20 삽입 완료!

제이씨

AVL Tree의 제거 \leftrightarrow BST에서의 제거

but, 제거할 때마다

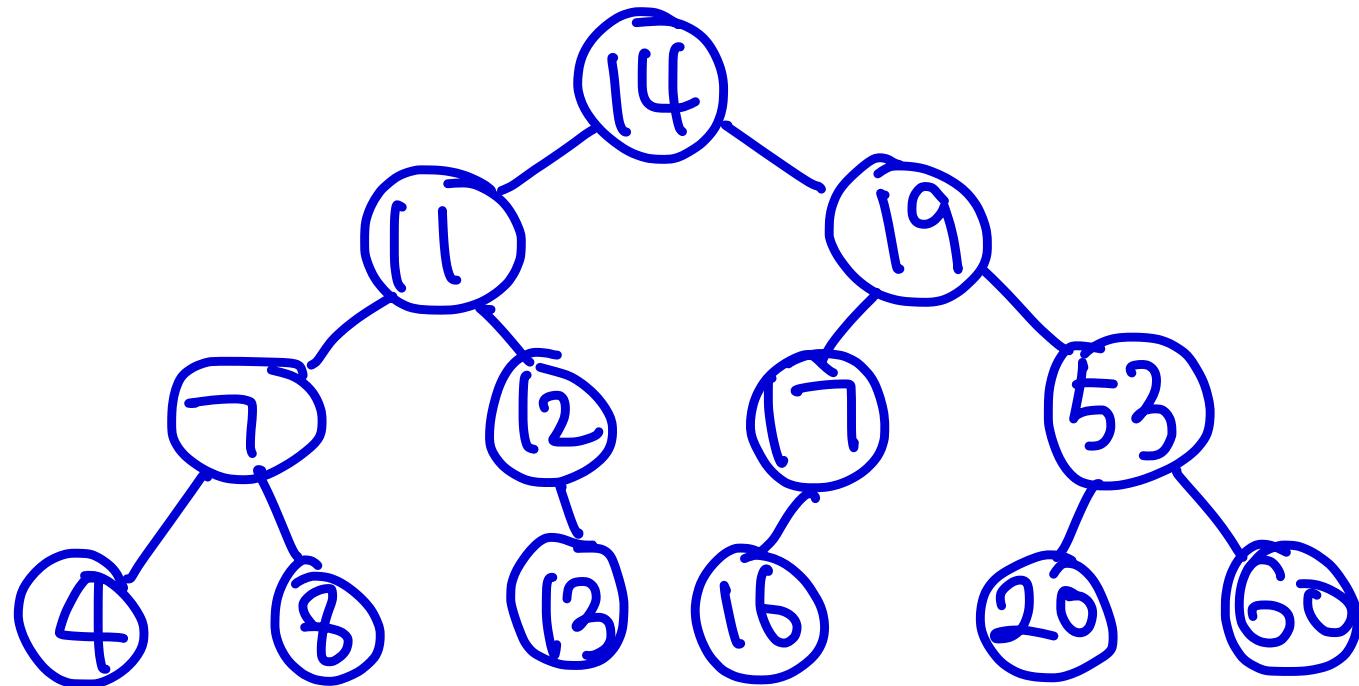
balance factor를

제이씨

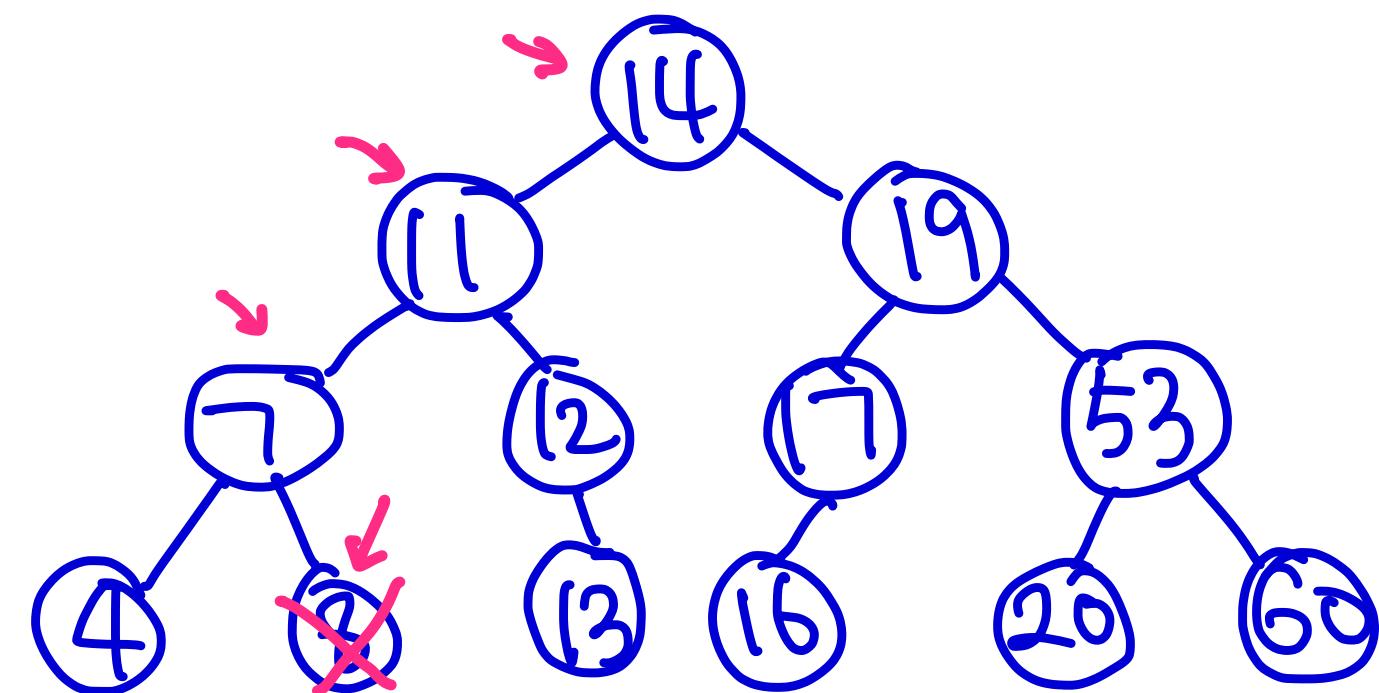
AVL Tree의 제거 \leftrightarrow BST에서의 제거

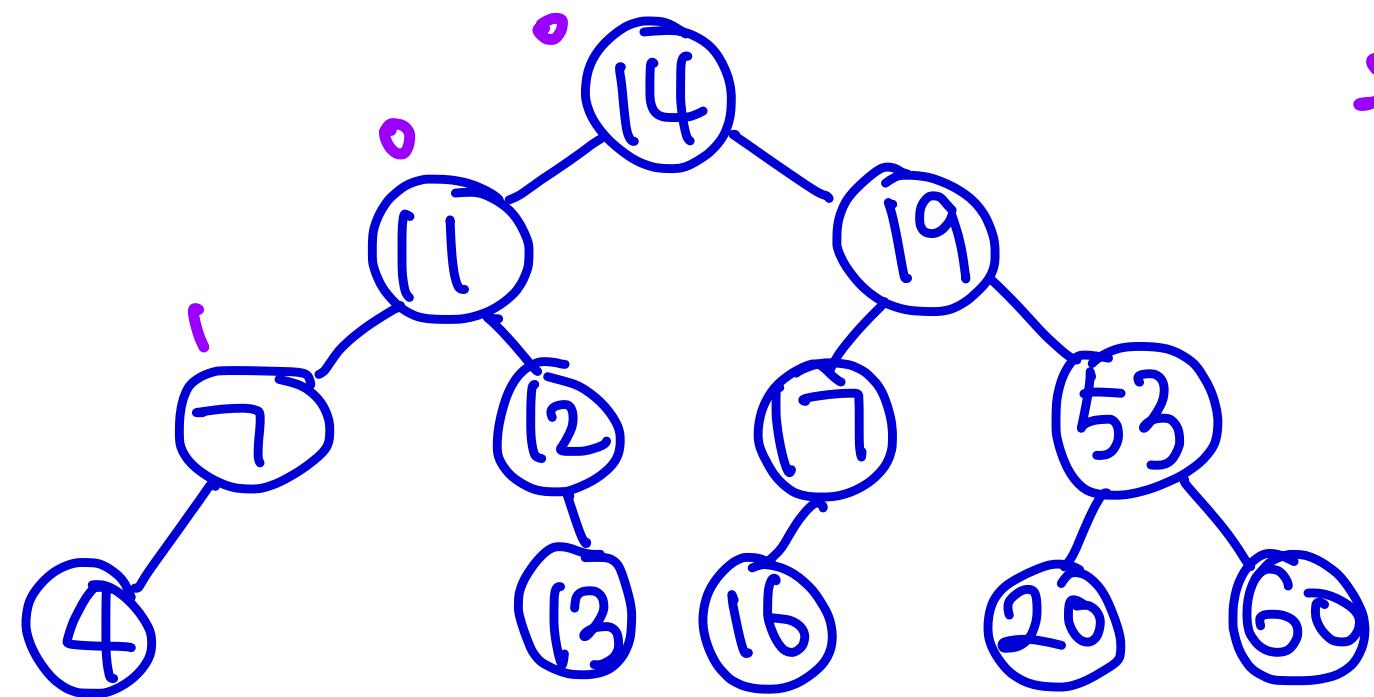
but, 제거할 때마다
AVL 트리인지 확인한다.

여기서 8 제거



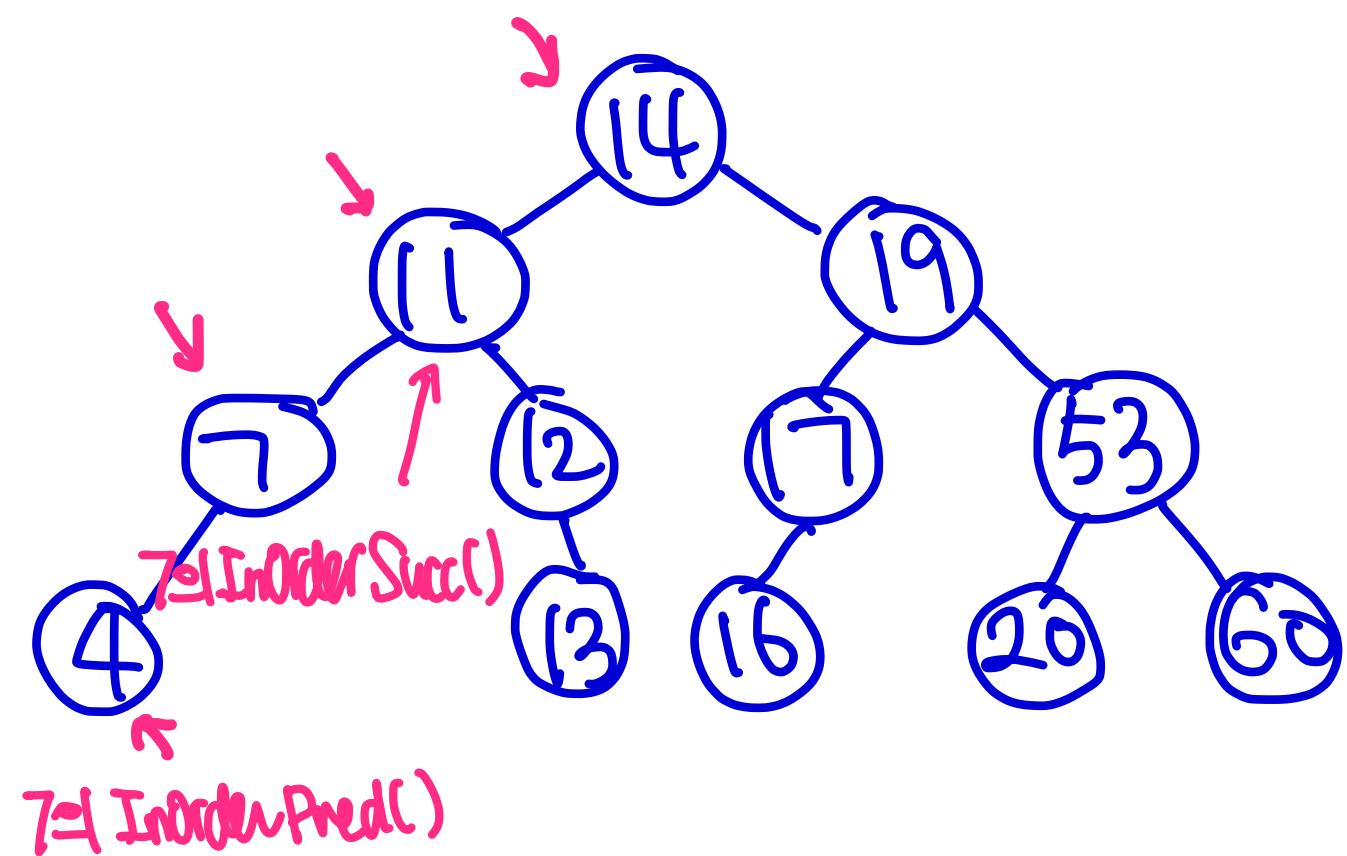
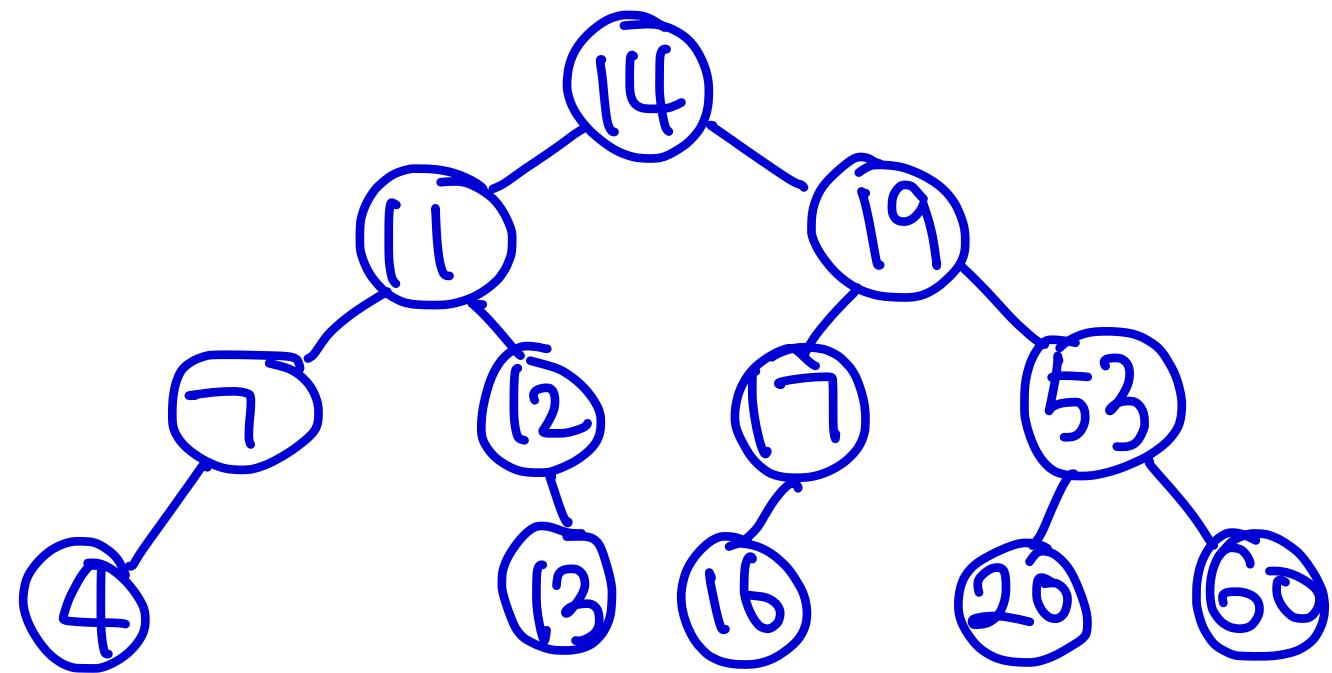
1) BST와 동일하게 제거
↔ 삭제하려는 노드까지
찾아내려간 후,
8의 자식이 없으므로
그냥 삭제





2) 제거할 때마다
AVL 트리인지 확인
(bf 계산)
⇒ AVL 트리 맞음
⇒ 8 제거 완료!

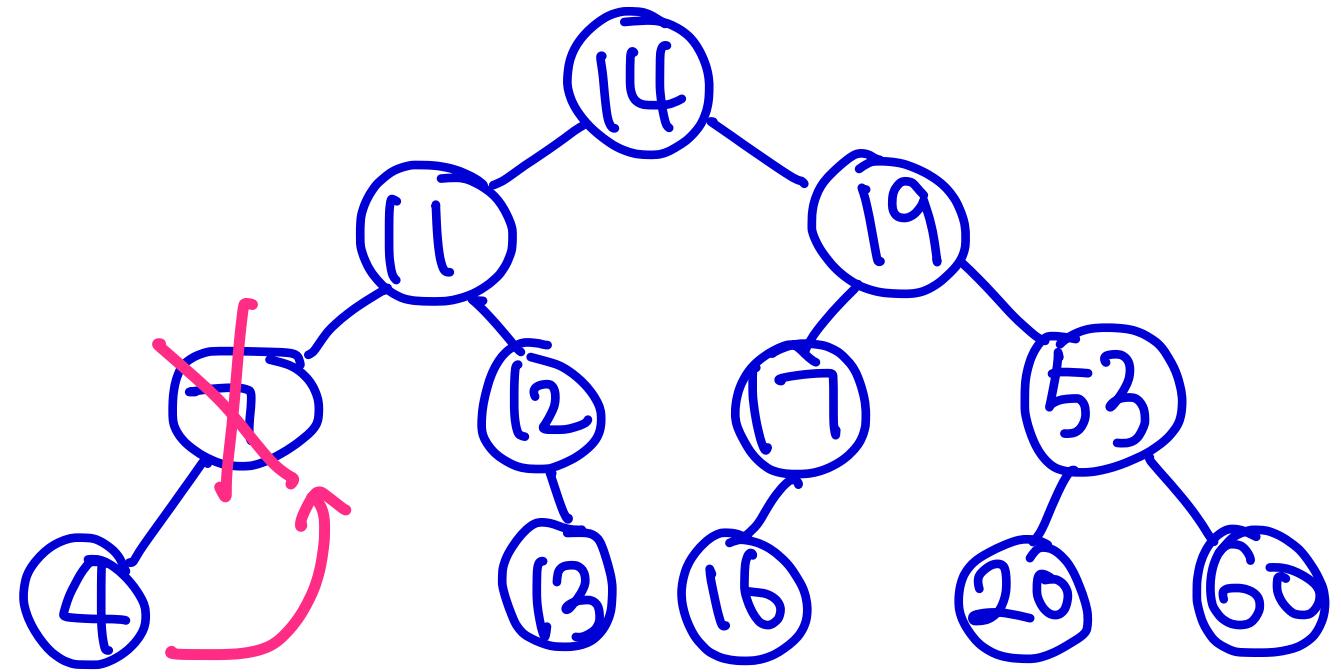
먼저 제거



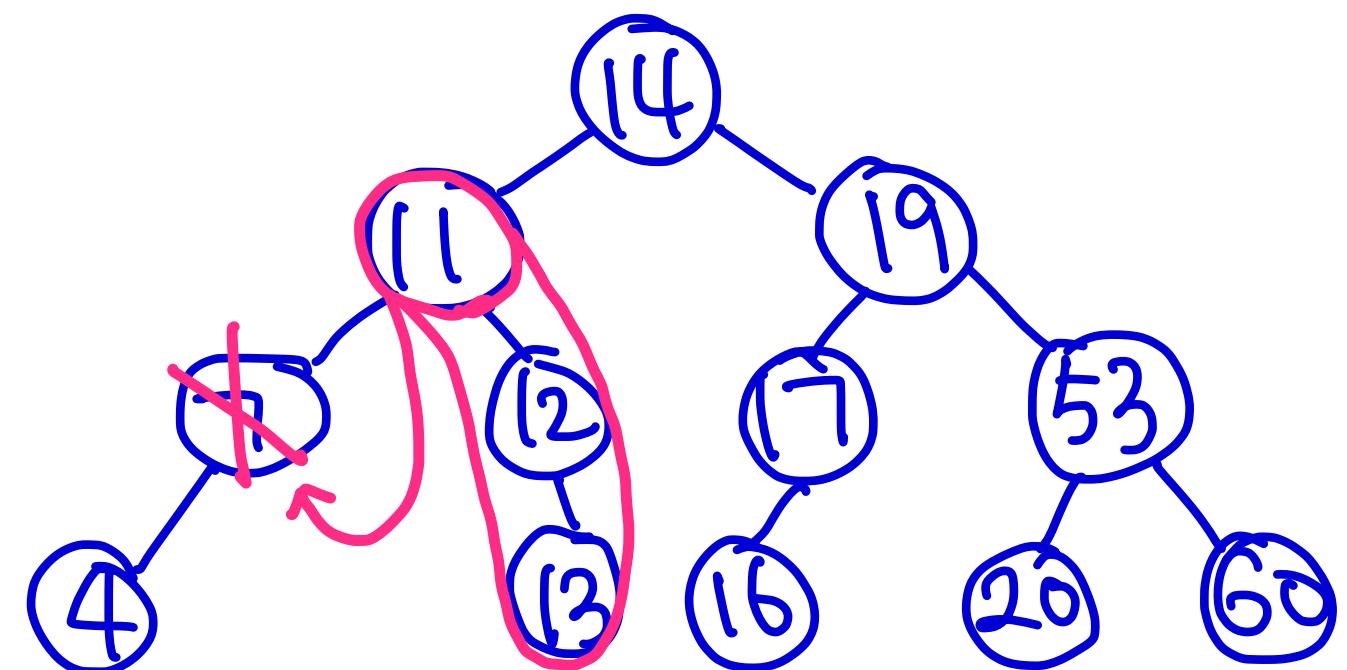
1) BST와 동일하게 제거
↔ 삭제하려는 노드까지
찾아내려간 후,
7은 자식이 있으므로
7의 InorderPred(),

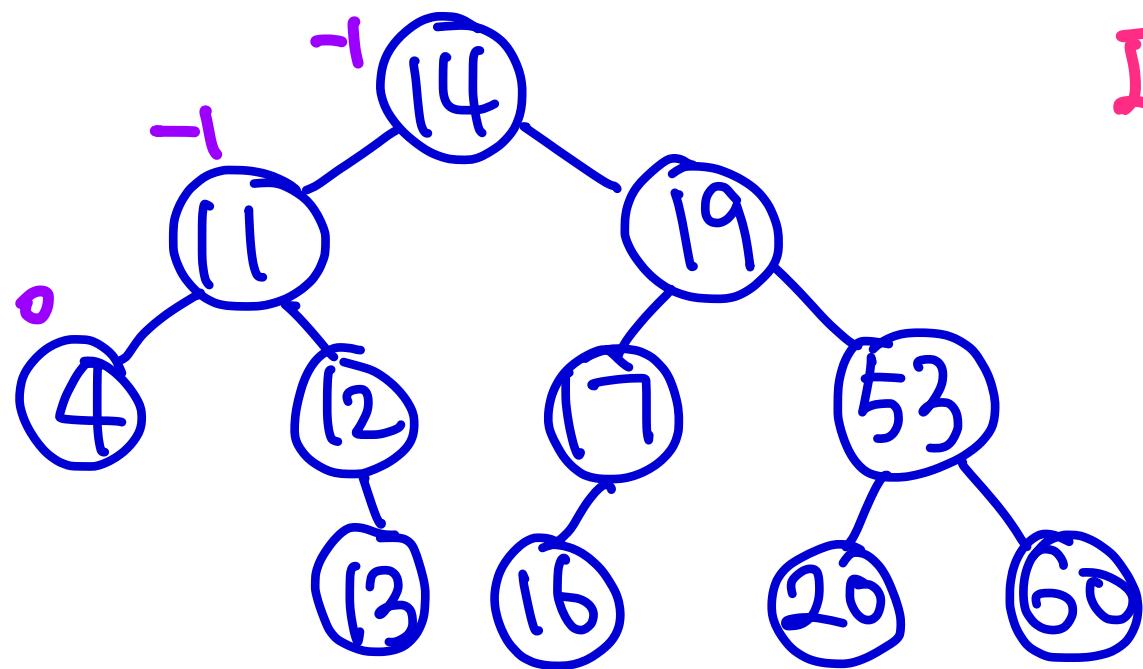
InOrderSucc() 중 택 1

여기서는 InOrderPred()²를
들리는 것이 편하므로
4를 들임.



그러나, InOrderSucc()²를
선택해도 크게 다르진 않음.
(여기서는 그거 그거 임)



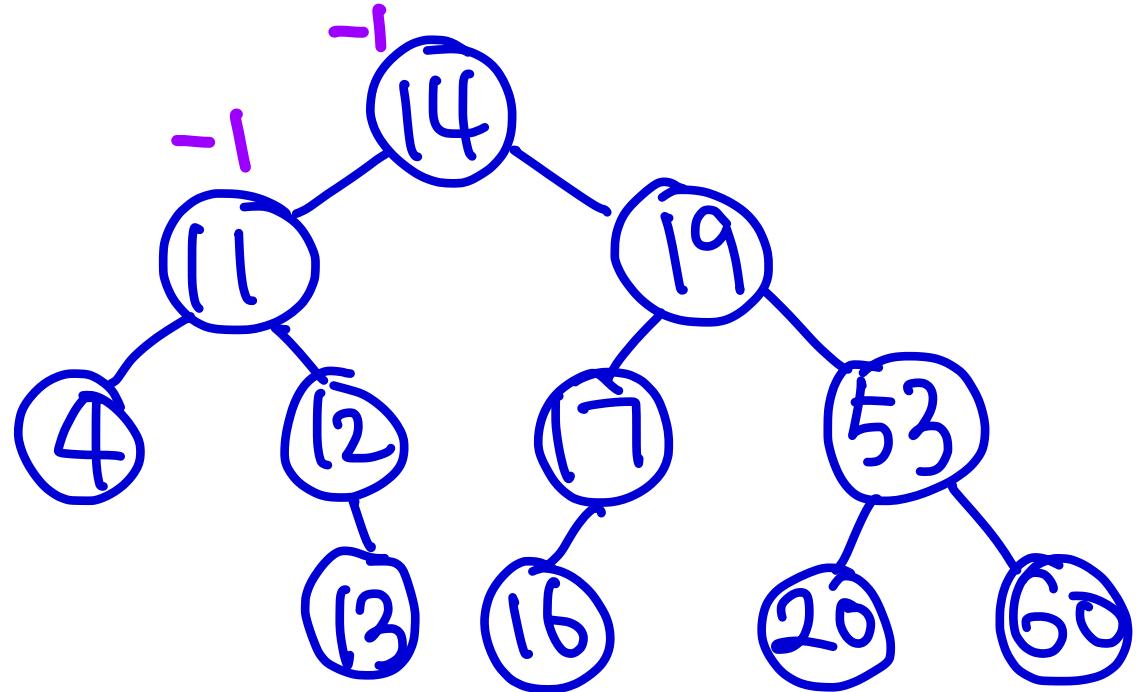


InOrderPred() 선택:

- 2) 제거 할 때마다 AVL 트리인지 확인
(bf 계산)

⇒ AVL 트리 맞음

⇒ 7 제거 완료!



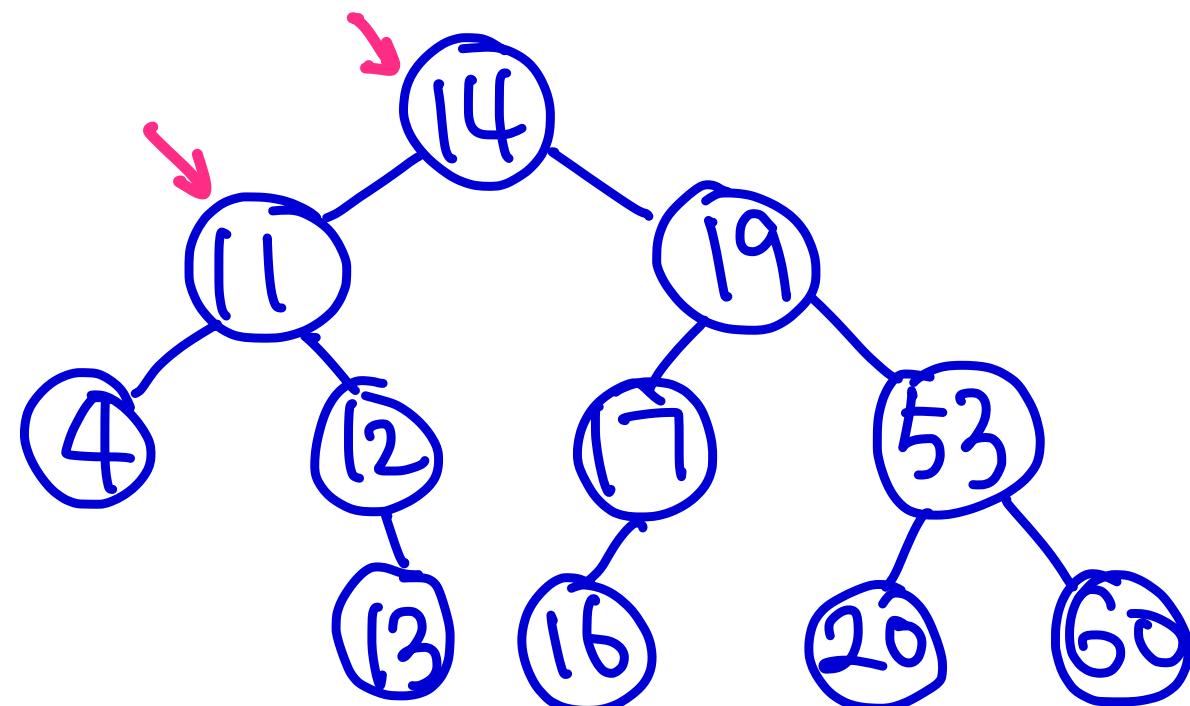
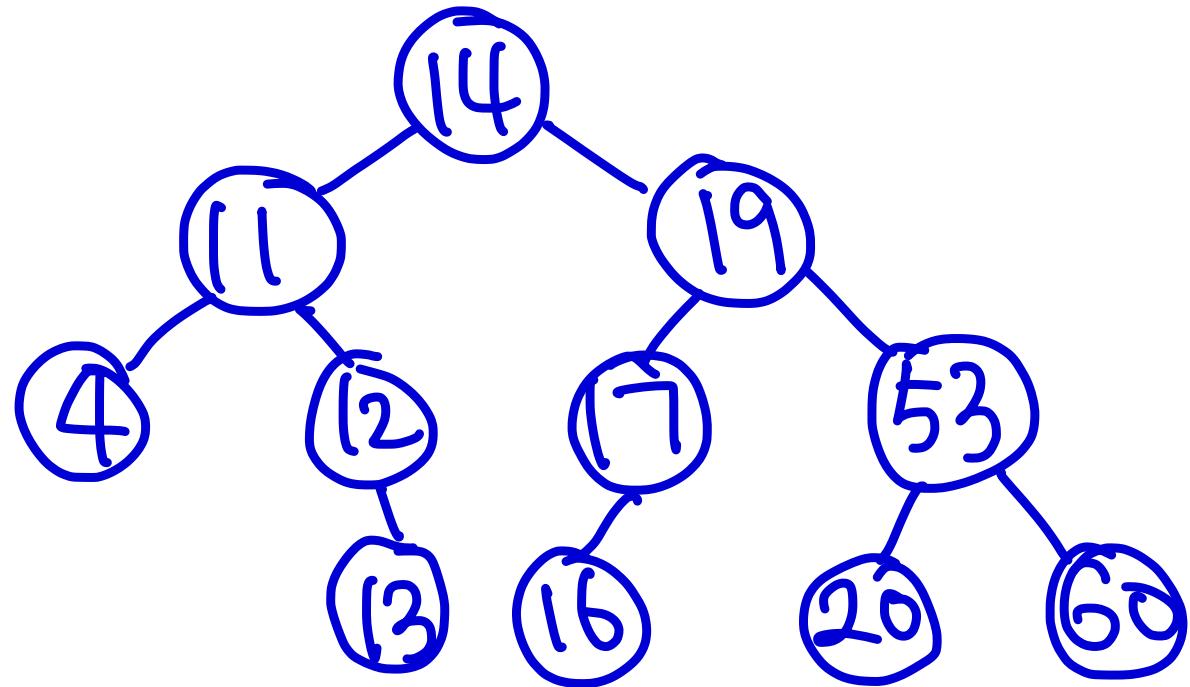
InOrderSucc() 선택:

- 2) 똑같이

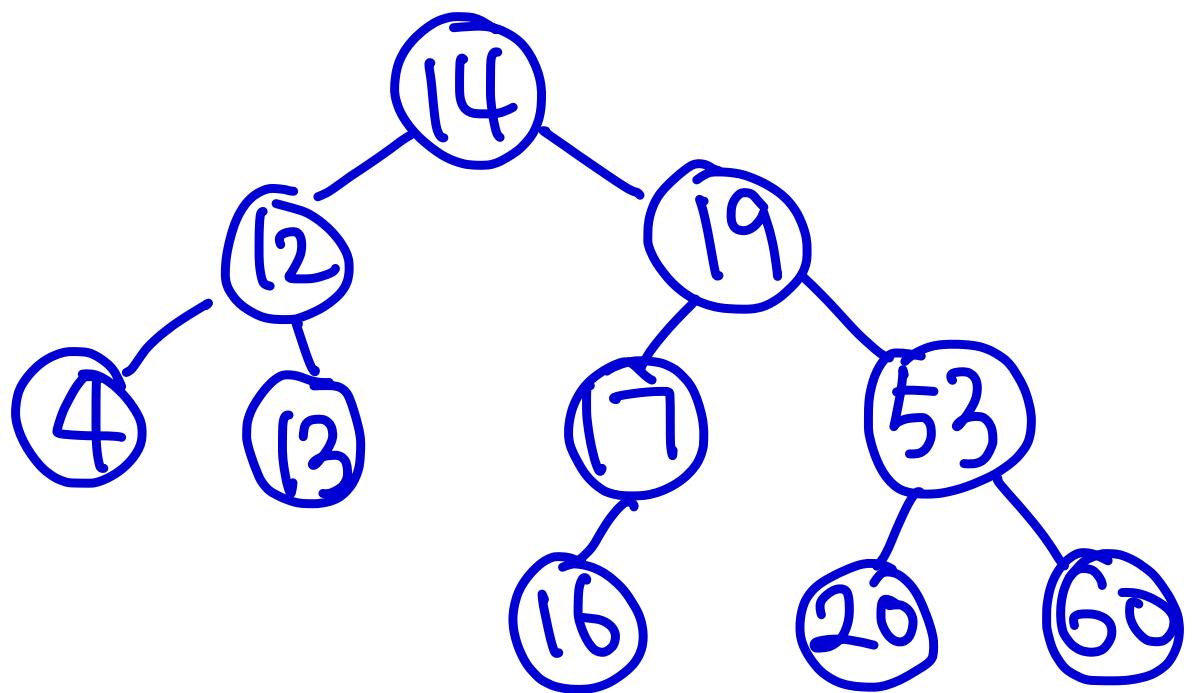
⇒ AVL 트리 맞음

⇒ 7 제거 완료!

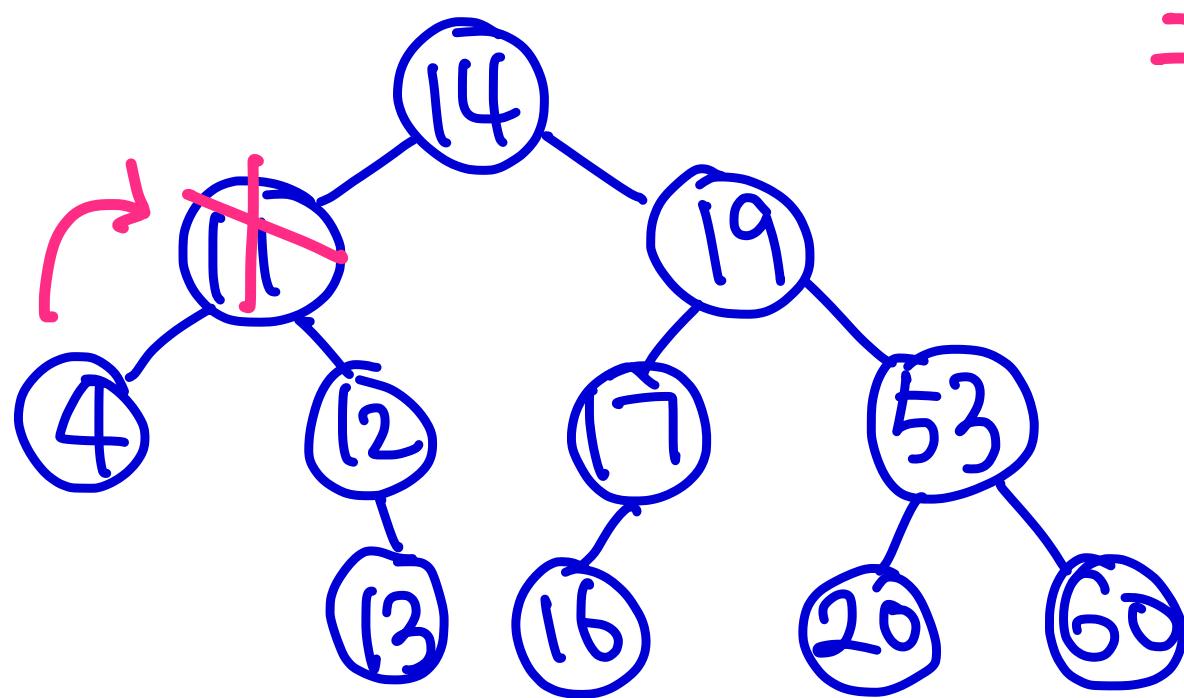
마지막 노드



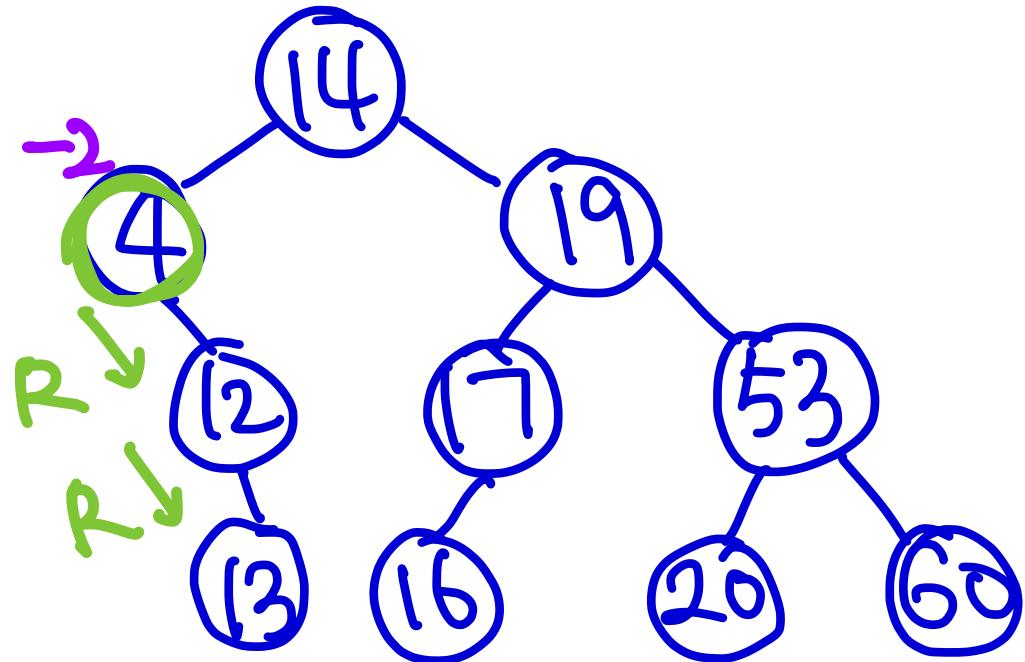
1) BST와 동일하게 제작
↔ 삭제하려는 노드까지
찾아내려간 후,
[]은 차례가 없으므로
[]의 InorderPred(),
InorderSucc() 중 택 1



cf) InOrderSucc()를 택하면
bf가 깨지지 않음
→ 성능 최적화의 여지가 있음



그런데 AVL 트리 제거시
회전을 보여드리기 위해
InOrderPred()를 택해보면
11의 InOrder Pred() = 4
4는 자식들이 없기 때문에
4를 바로 옮길 수 있음



2) 삽입할 때마다

AVL 트리인지 확인
(bf 계산)

3) bf가 깨진 노드가 있다면

회전을 해야 함

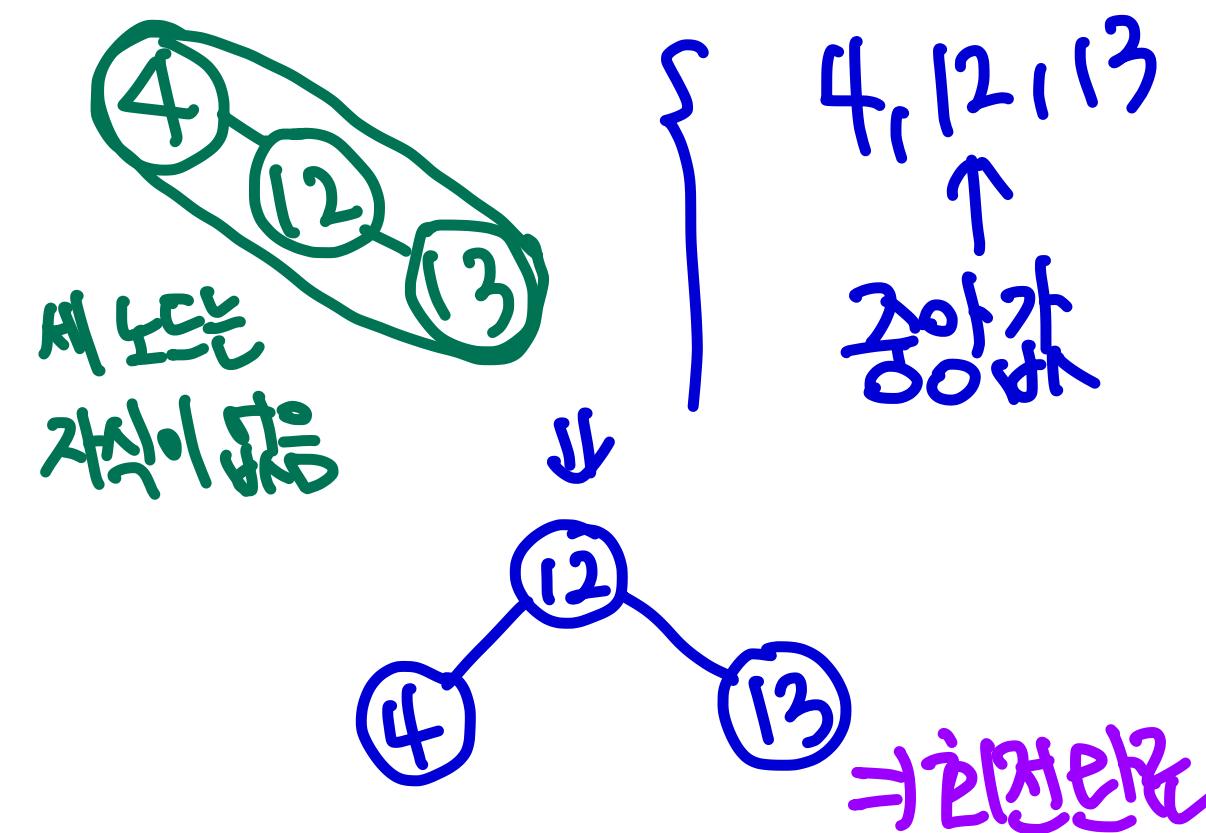
⇒ 회전 당하는 노드 3개는

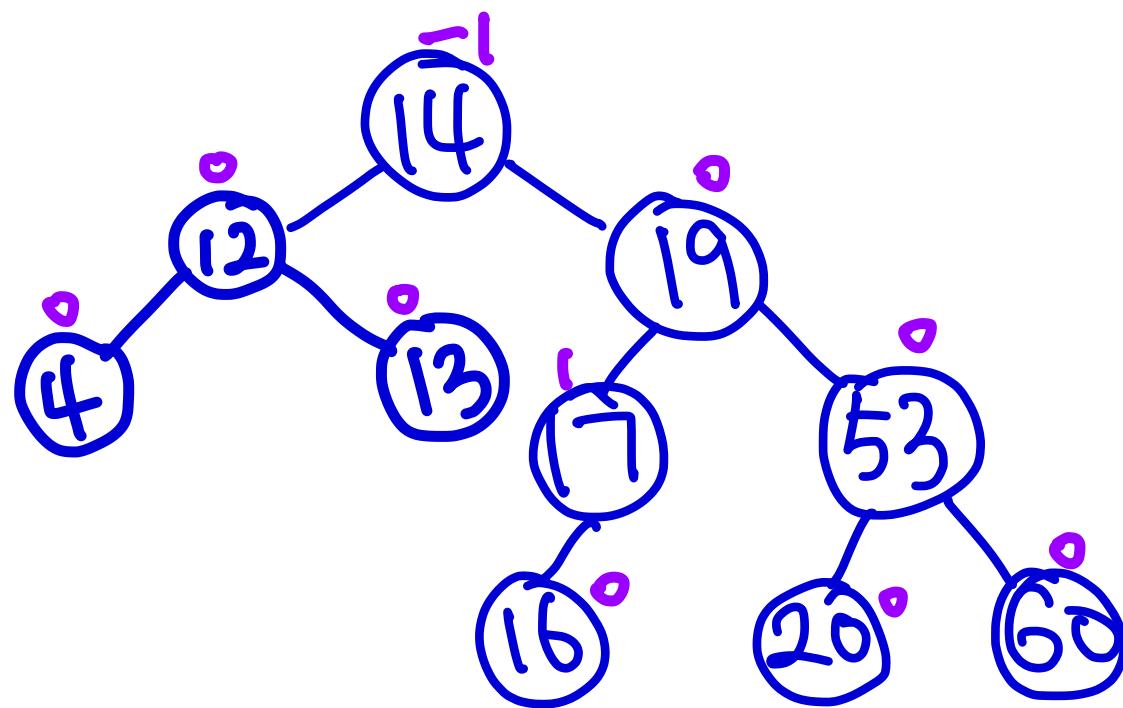
무엇인가? 깨진 노드부터 RR

⇒ 4, 12, 13

4) 회전 ① 중앙값 찾기로

② 자식 노드 재배치





5) 회전 완료 후

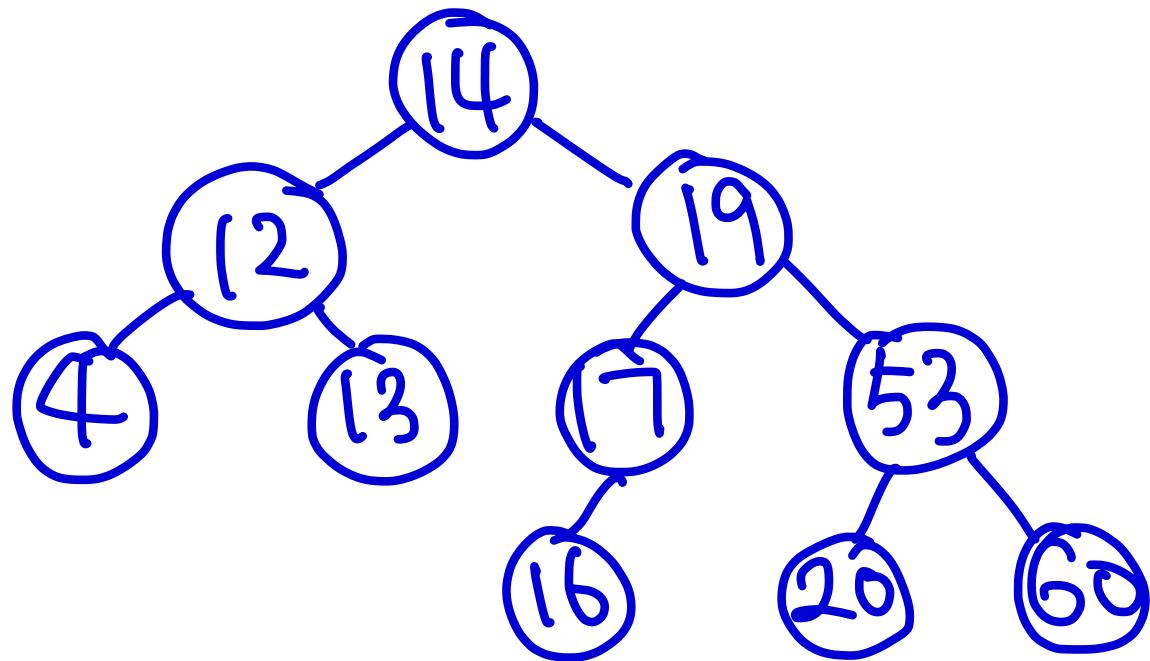
전체 bf 확인.

- ⇒ bf가 깨진 노드가 존재하면
루트까지 올라가며 깨진 노드 찾기
- ⇒ bf가 깨진 노드가 없으면 삽입

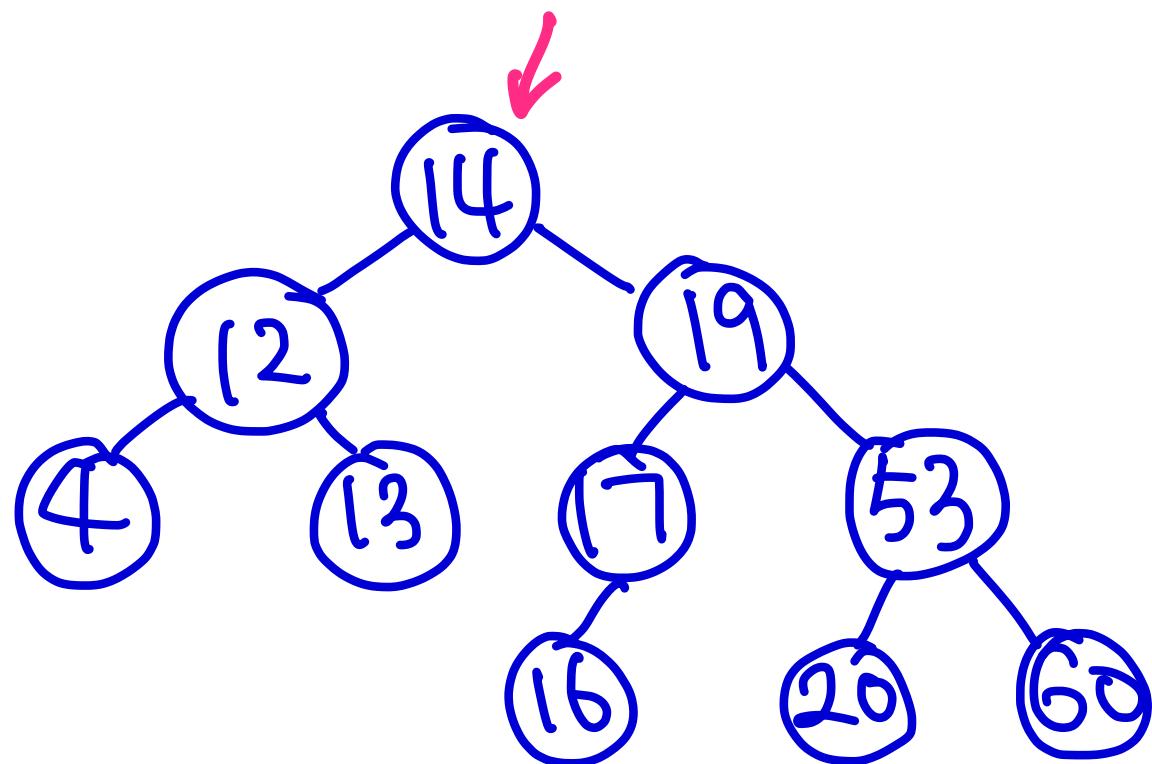
⇒ 전체 bf 확인

⇒ AVL Tree 맞음 (모든 $bf \in \{-1, 0, 1\}$)

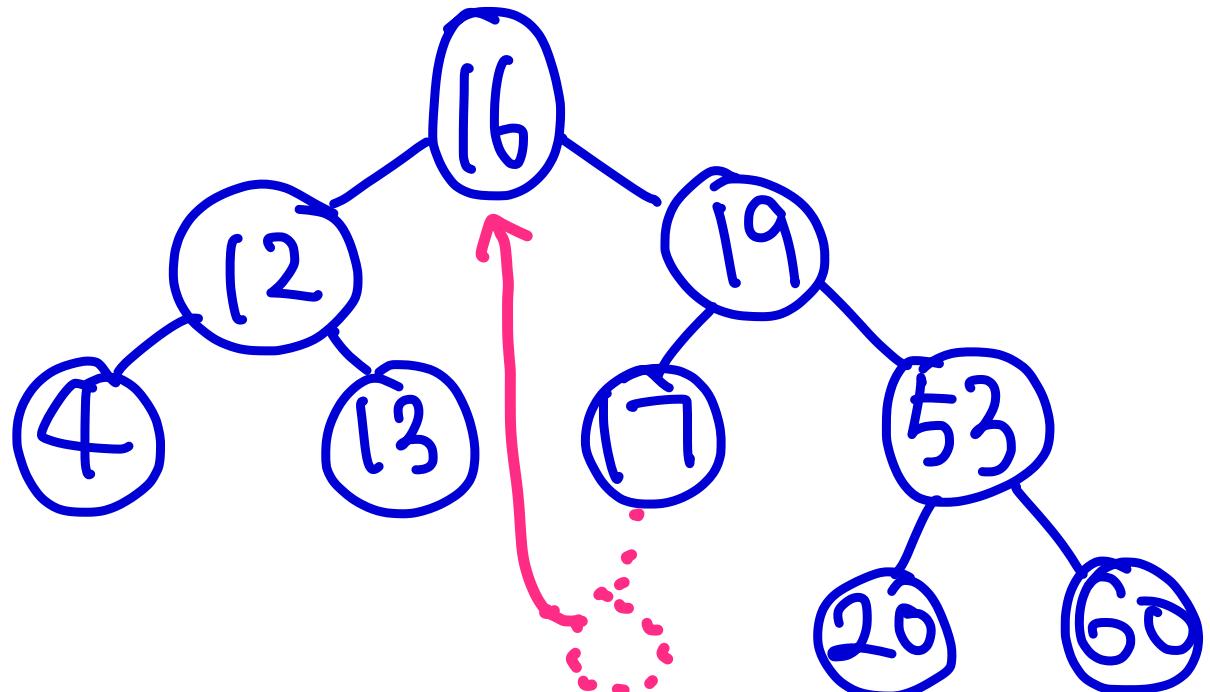
⇒ II 저항 완료



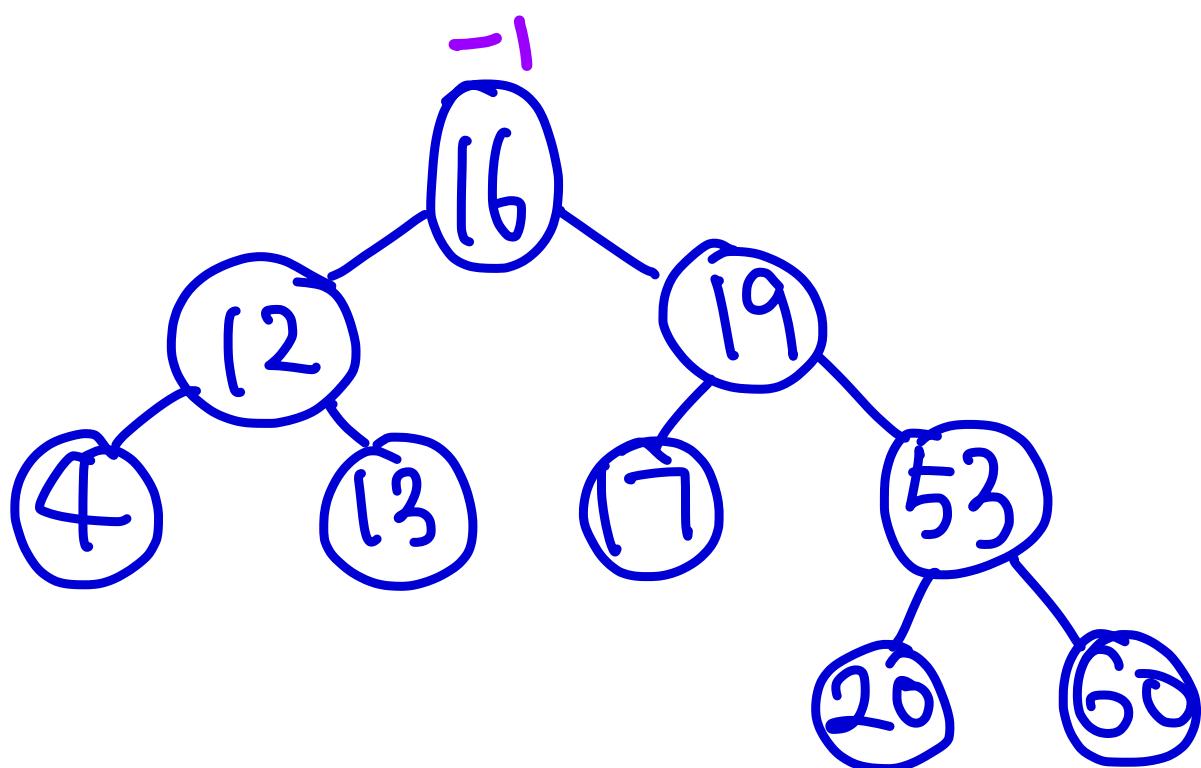
여기서 14 삭제



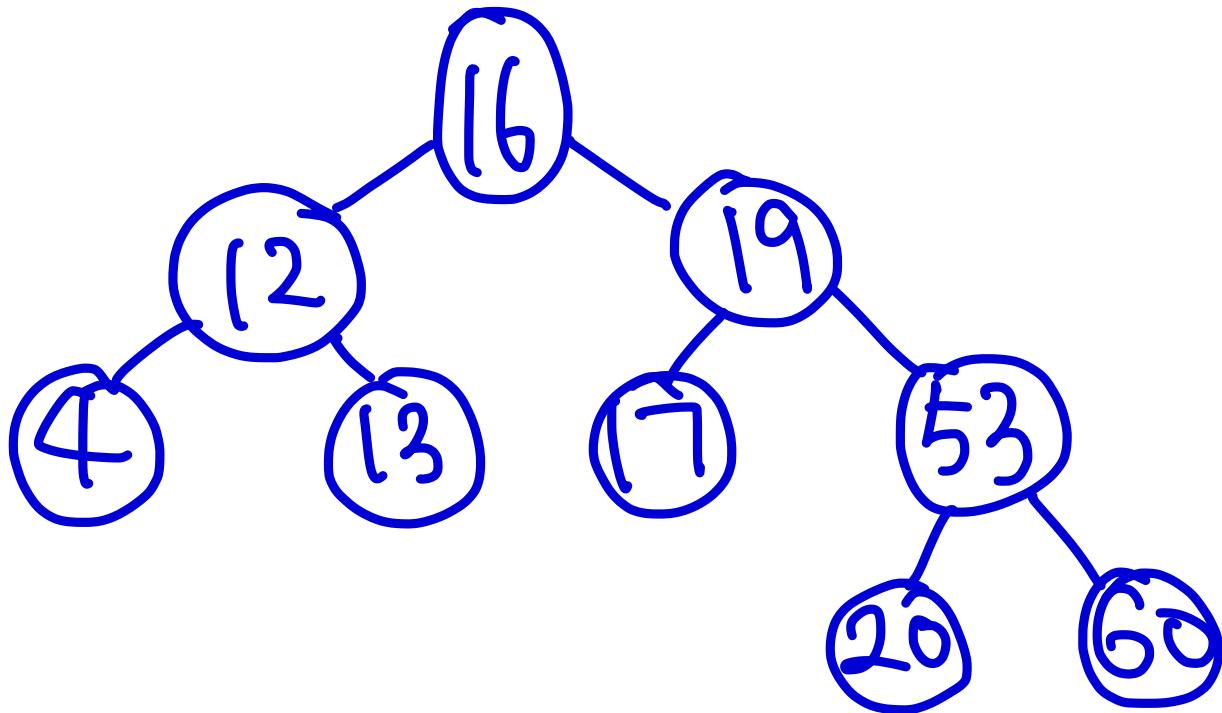
1) BST와 동일하게 제거
 \Leftarrow 삭제하려는 노드가자
 찾아내려간 후,
 14는 자식이 있으므로
 (14의 InorderPred(),
 InorderSucc()중 택 1)



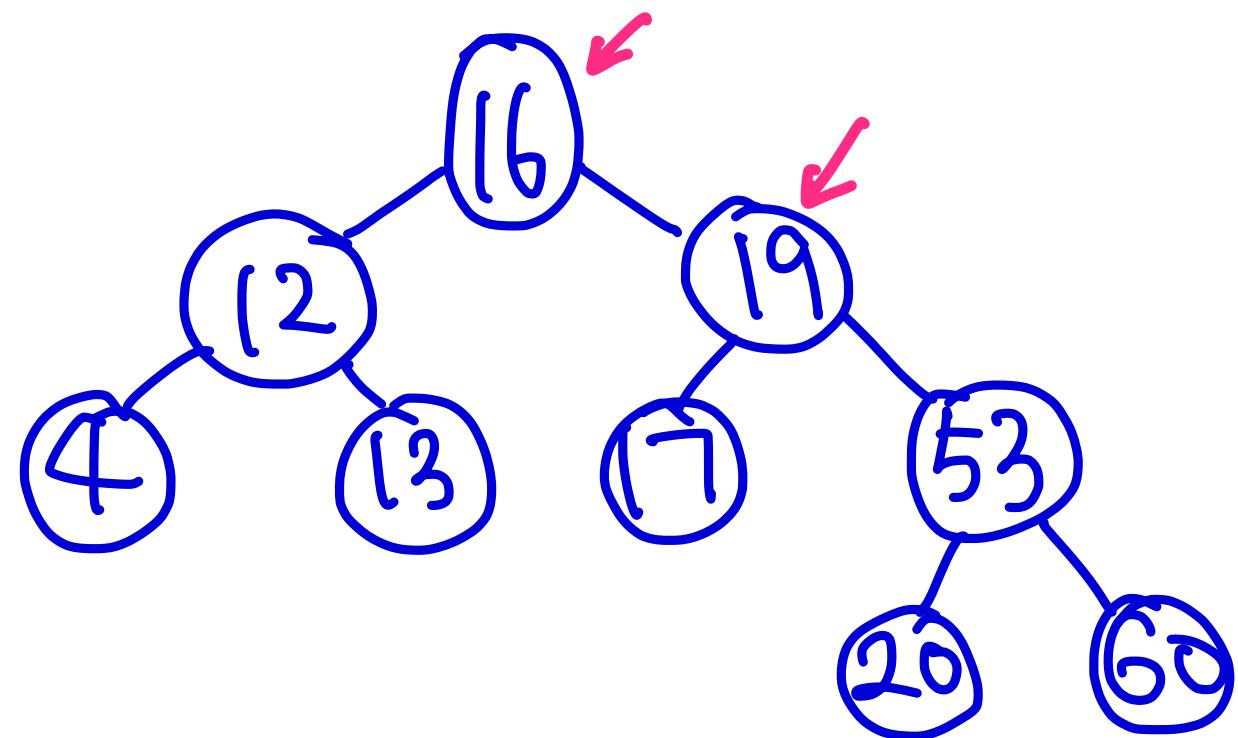
(4의 InOrderPred() = 13
 (4의 InOrderSucc() = 16
 둘중 무언가 틱해도 bf까지 X



2) 제거 할때마다
 AVL 트리인지 확인
 (bf 계산)
 ⇒ AVL 트리 맞음
 ⇒ 14 제거 안됨!

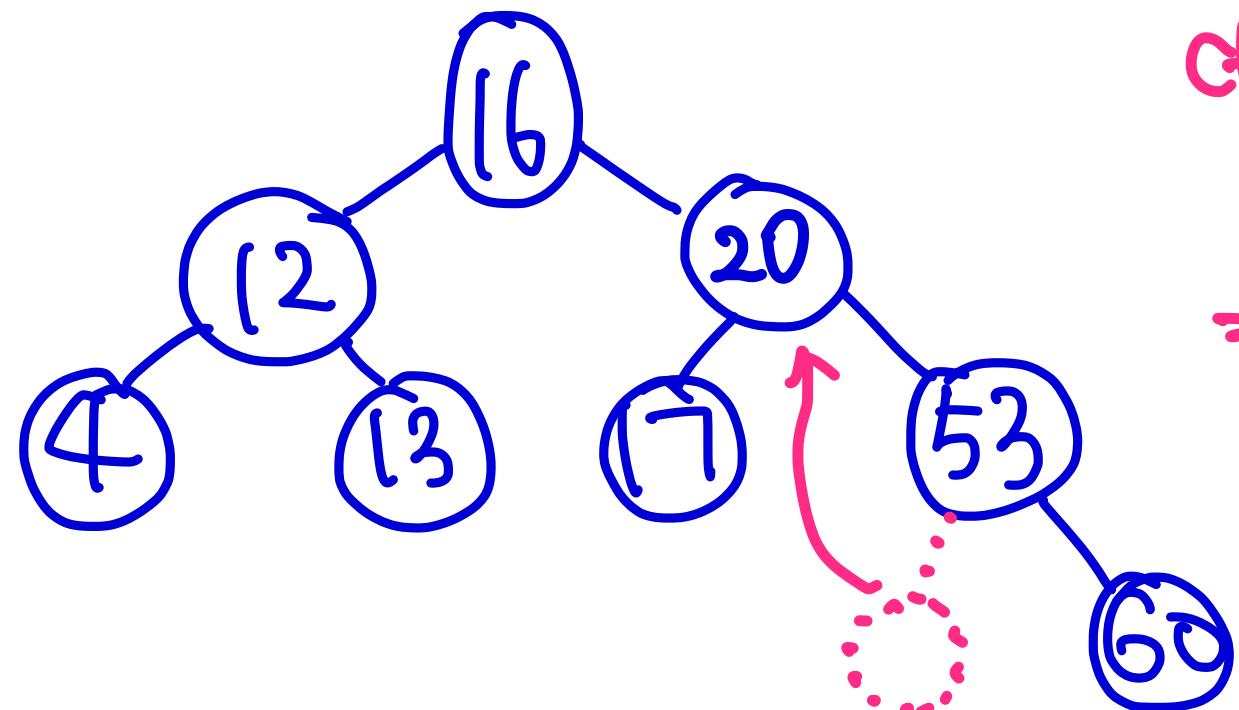


여기서 19 삭제

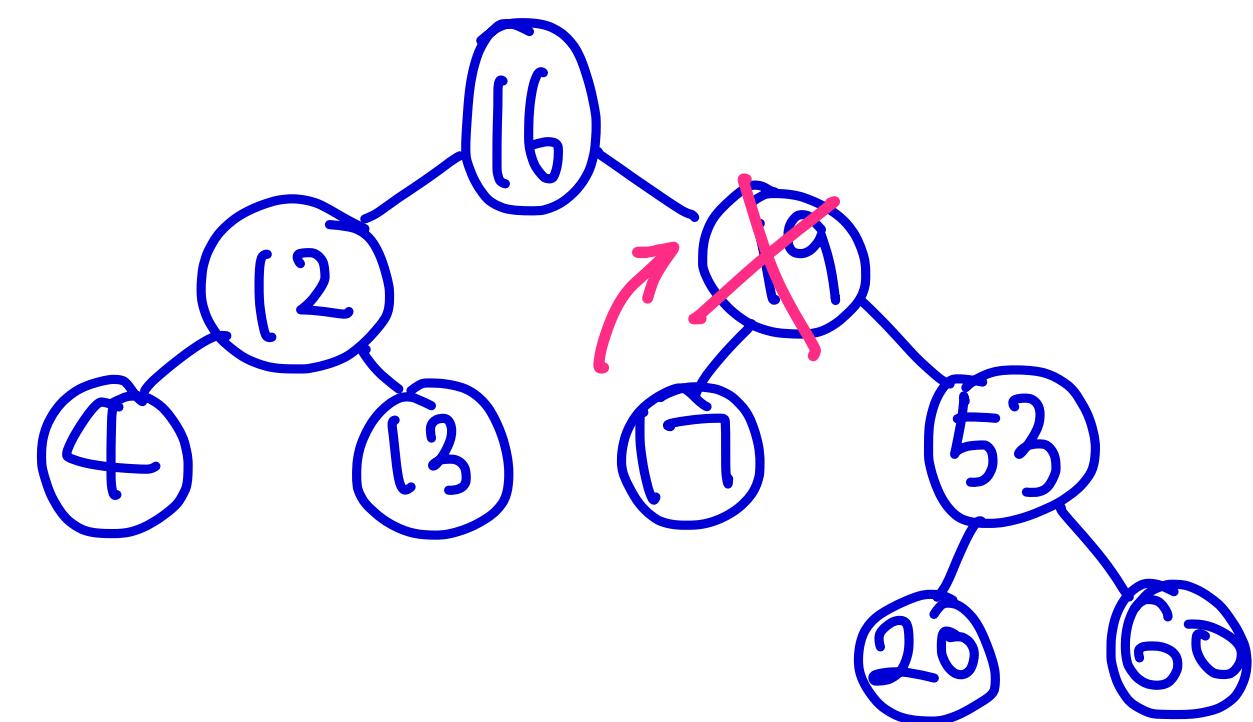


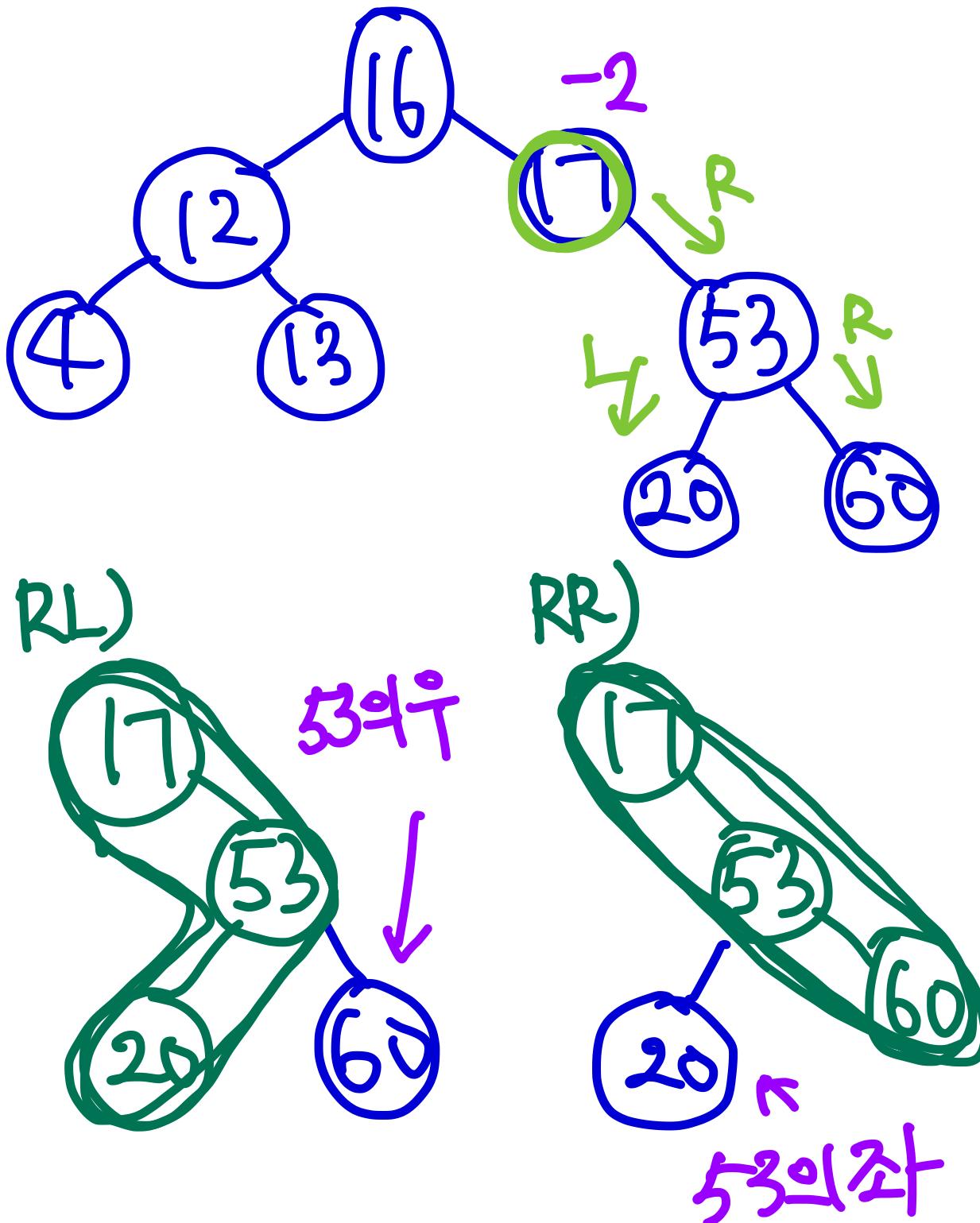
1) BST와 동일하게 제거
 ⇔ 삭제하려는 노드까지
 찾아내려면,
 19는 자식이 있으므로
 19의 InorderPred(),
 InorderSucc() 중 택 1

cf) InOrderSucc()를 택하면
bf가 깨지지 않음
→ 성능 최적화의 여지가 있다는 뜻



그러데 AVL 트리 제거시
최전을 보여드려야 하기 위해
InOrderPred()를 택해보면
19 의 InOrder Pred() = 17
17은 자식들이 없기 때문에
17을 바로 끝을 수 있음





2) 저거할때마다

AVL 트리인지 확인
(bf 계산)

3) bf가 깨진 노드가 있다면
회전을 해야함

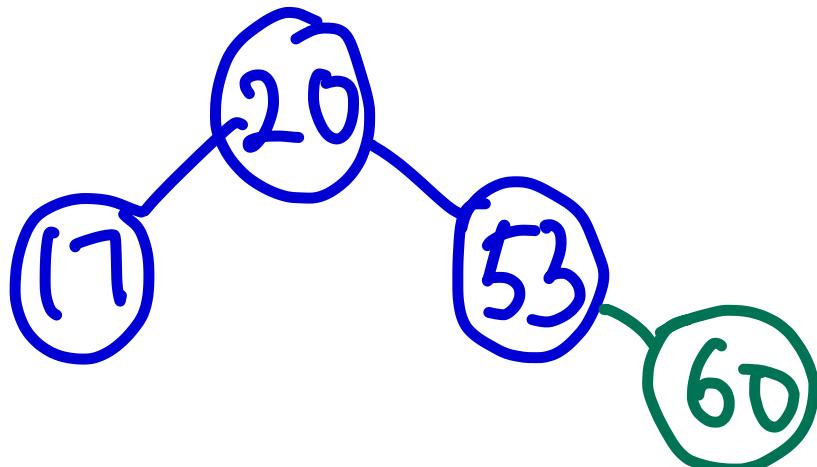
⇒ 회전 당하는 노드 3개는
무엇인가? 깨진 노드부터 RL

⇒ 17, 53, 20

OR 깨진 노드부터 RR

⇒ 17, 53, 60

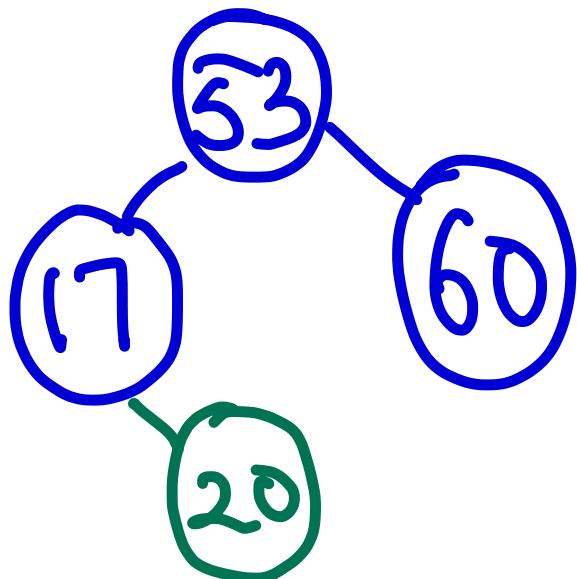
Case 1 17, 53, 20으로 잡은 경우 (RL)

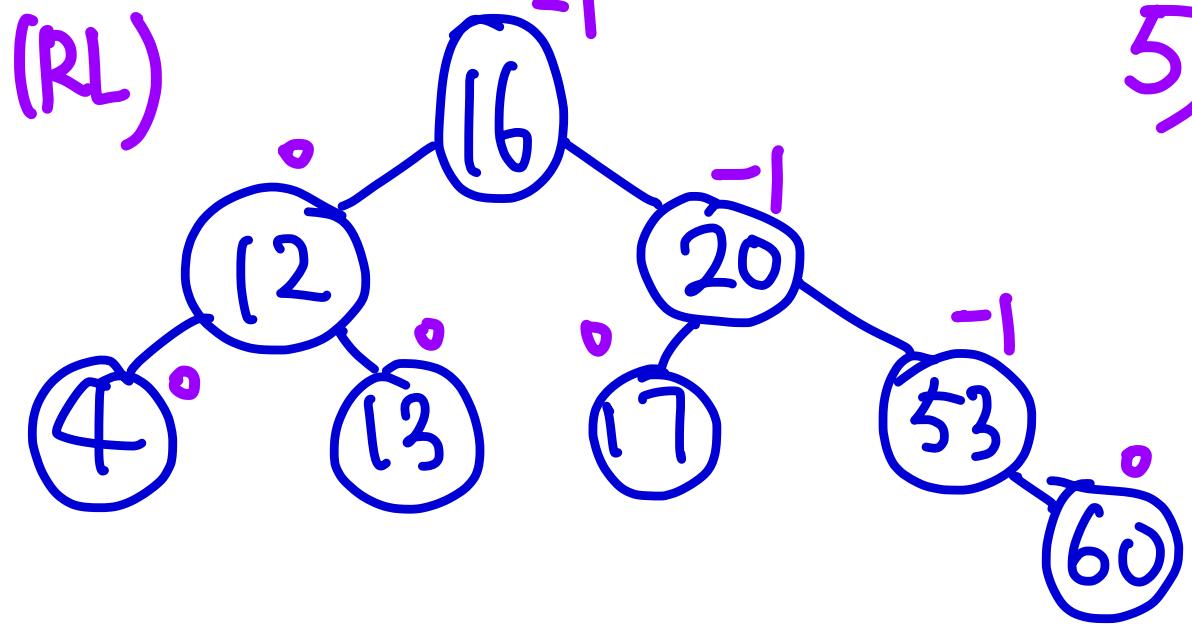


둘중 턱!

- 4) 회전 ① 중앙값 루트로
② 자식노드 재배치

Case 2 17, 53, 60으로 잡은 경우 (RR)



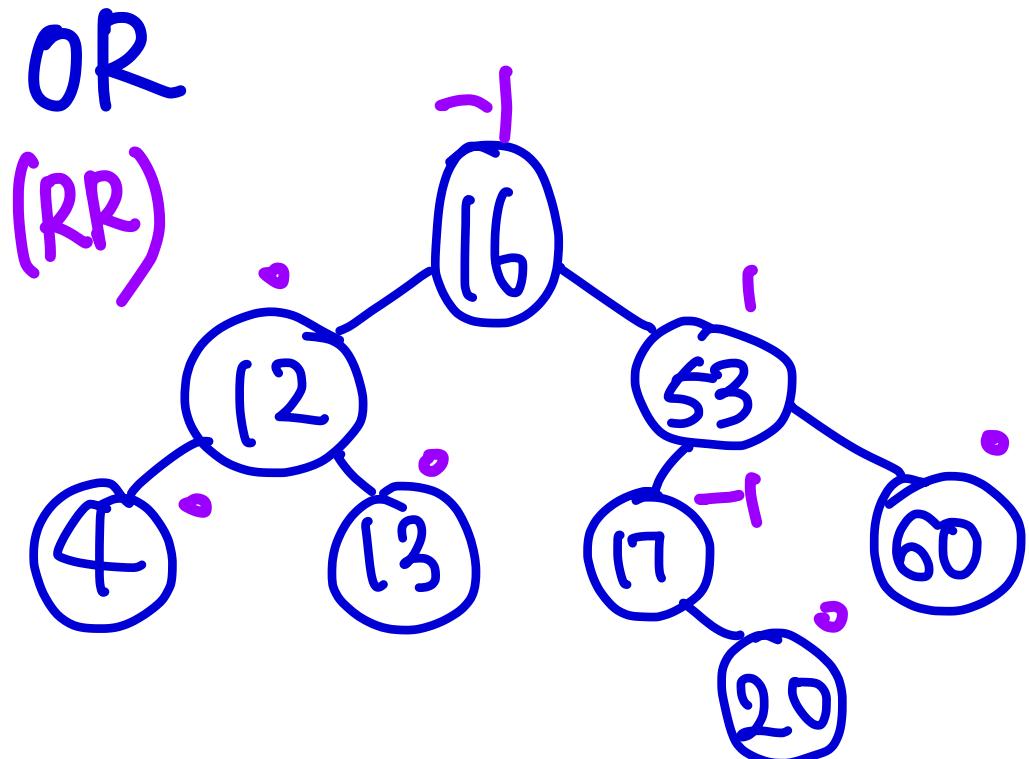


5) 회전 완료 후

전체 bf 확인.

⇒ bf가 깨진 노드가 존재하면
루트까지 올라가며 깨진 노드 찾기

⇒ bf가 깨진 노드가 없으면 삽입



⇒ 전체 bf 확인

⇒ AVL Tree 맞음

(모든 bf ∈ {-1, 0, 1})

⇒ 19 제거 완료