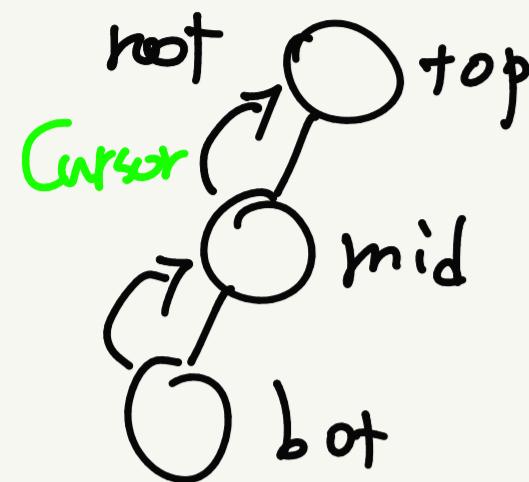


\* LL, RR의 예외상황

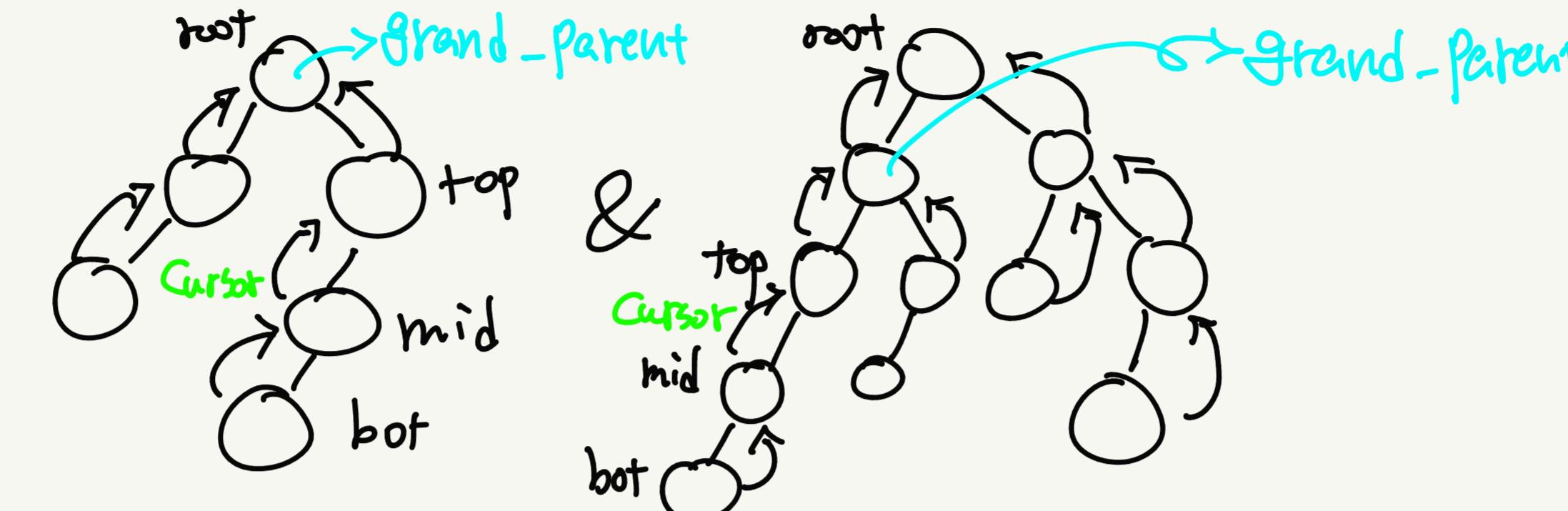
(1) LL의 예외상황

① Cursor와 root가 같을 때



→ mid를 root로 업데이트해준다.

② Cursor와 root가 다른 경우



→ Cursor와 root가 다른 경우는 2가지

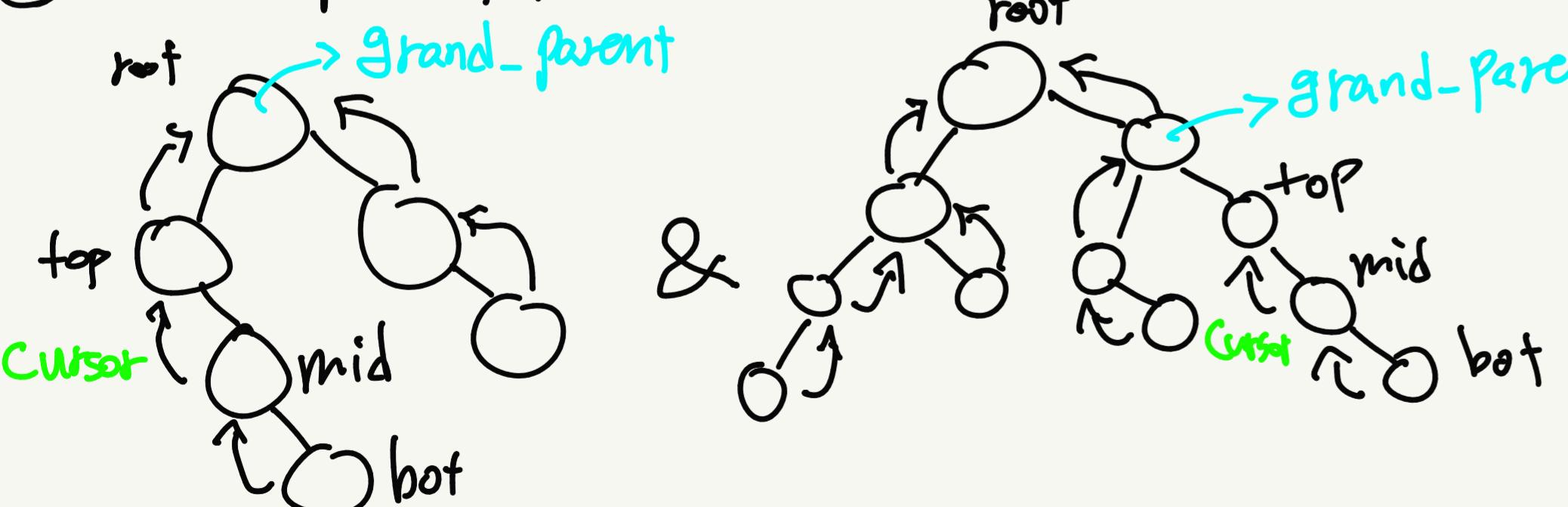
→ mid의 data > grand-parent의 data, grand-parent의 왼쪽을 mid로 변경

→ mid의 data < grand-parent의 data, grand-parent의 오른쪽을 mid로 변경

(2) RR의 예외상황

① LL의 ①상황과 동일

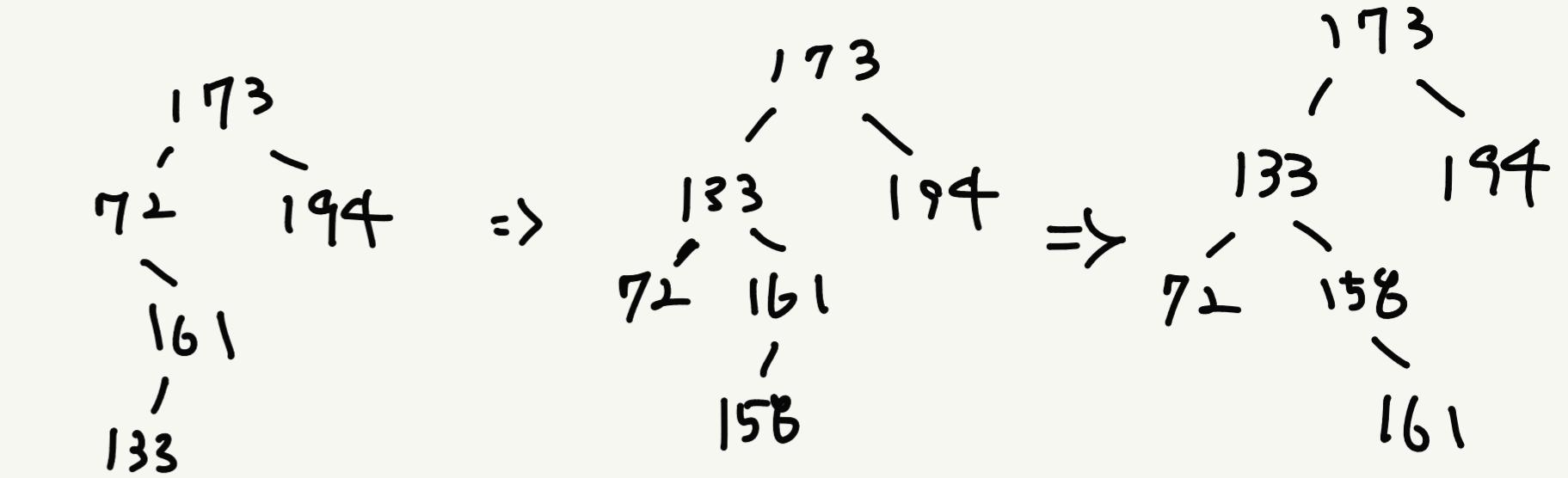
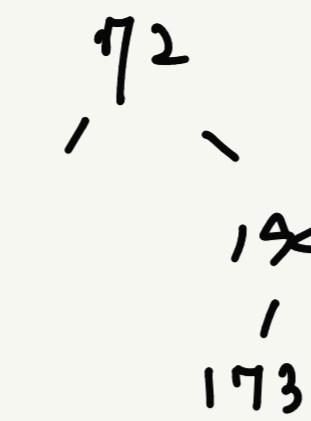
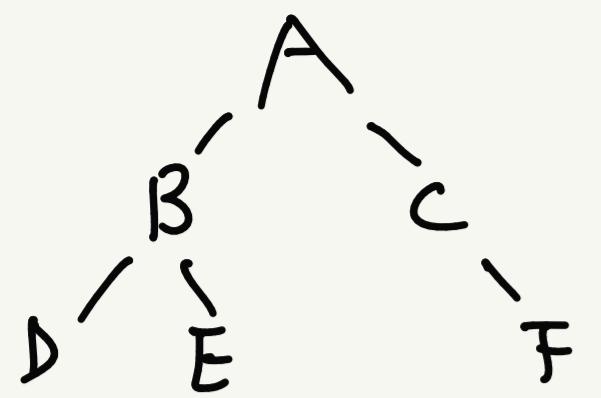
② Cursor와 root가 다른 경우



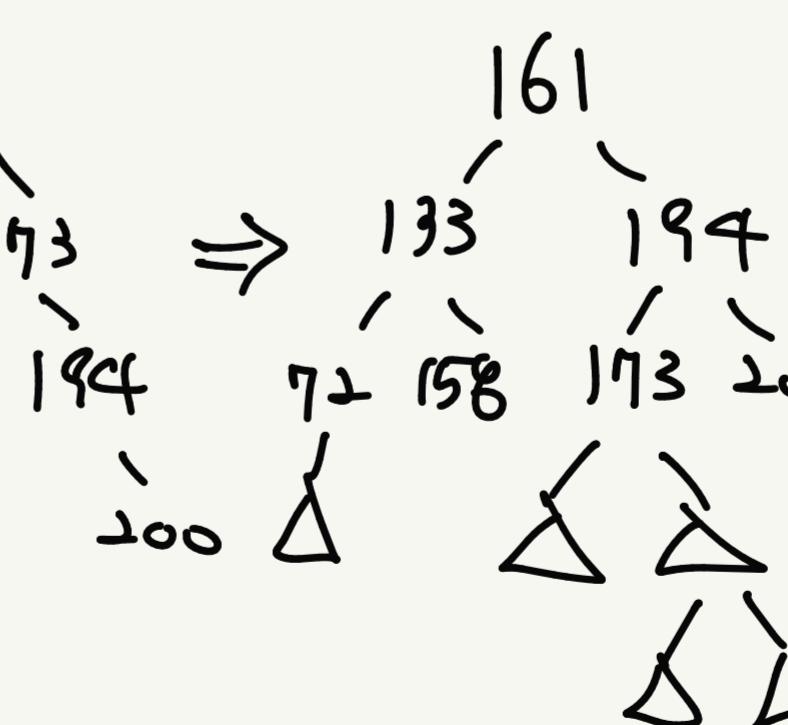
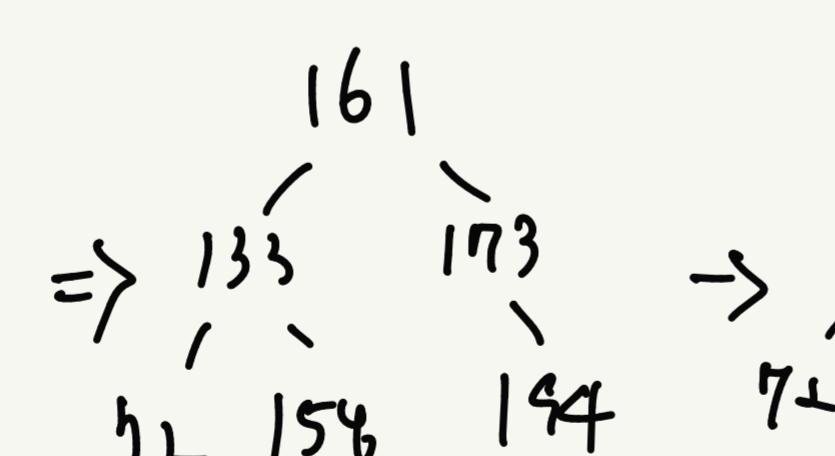
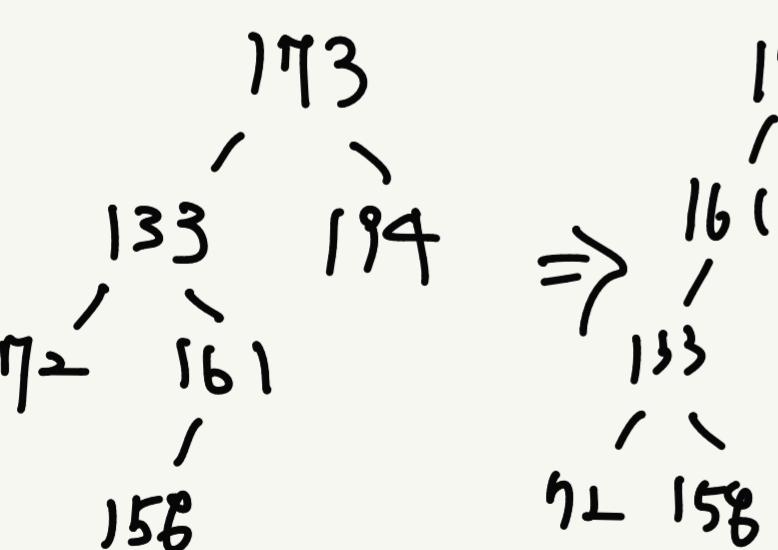
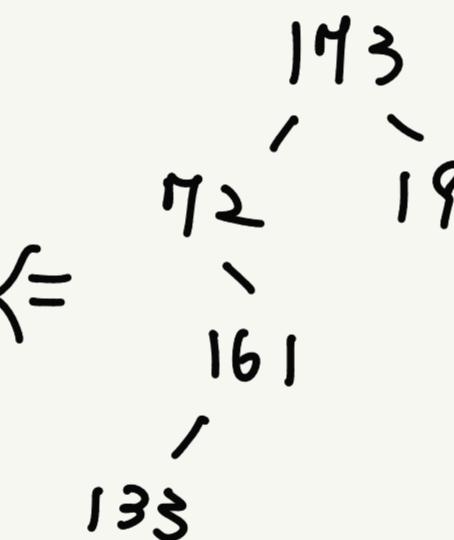
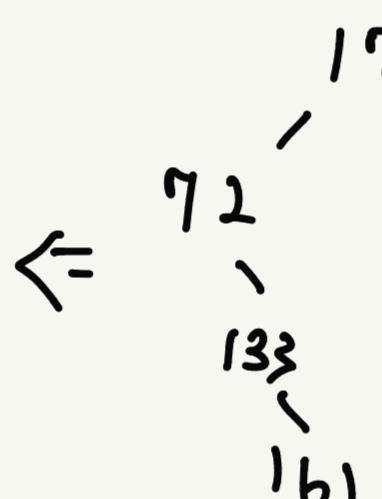
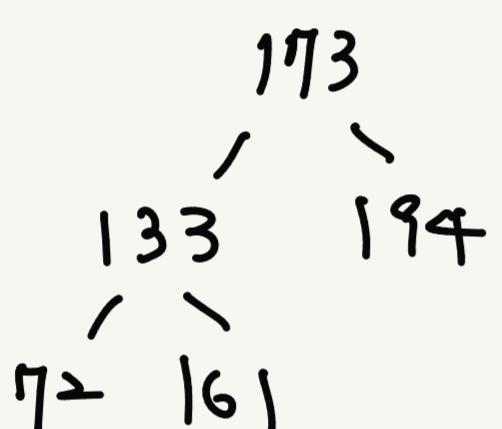
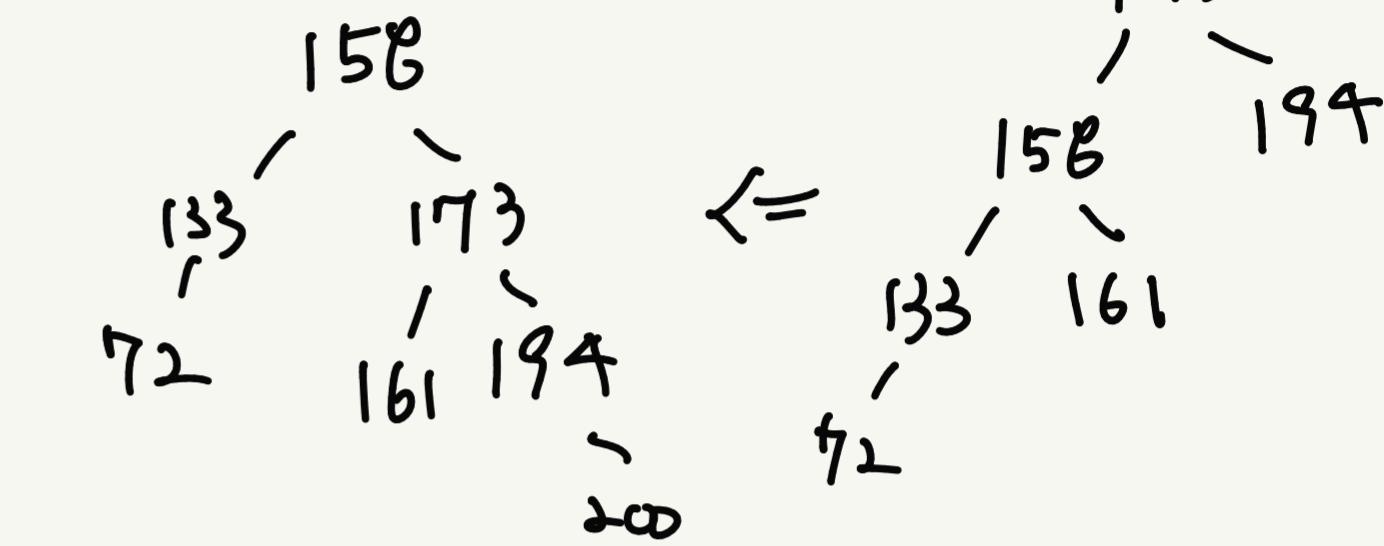
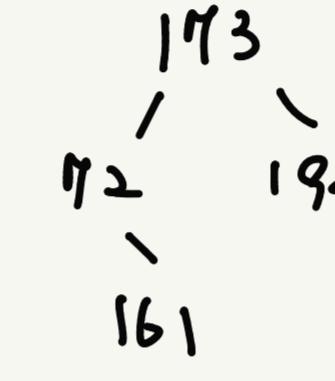
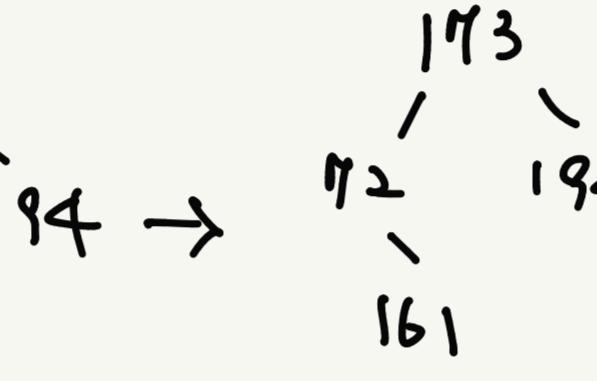
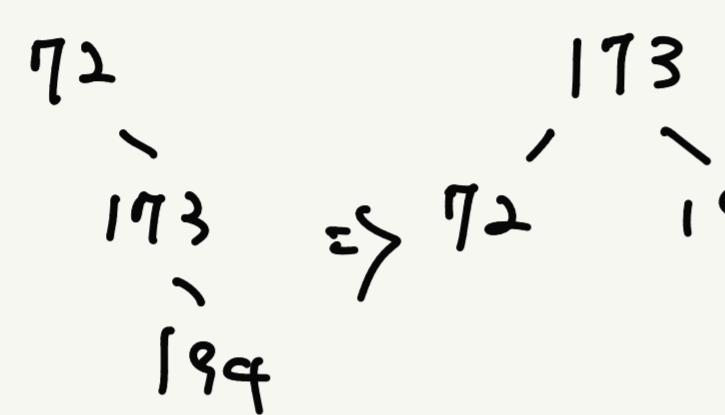
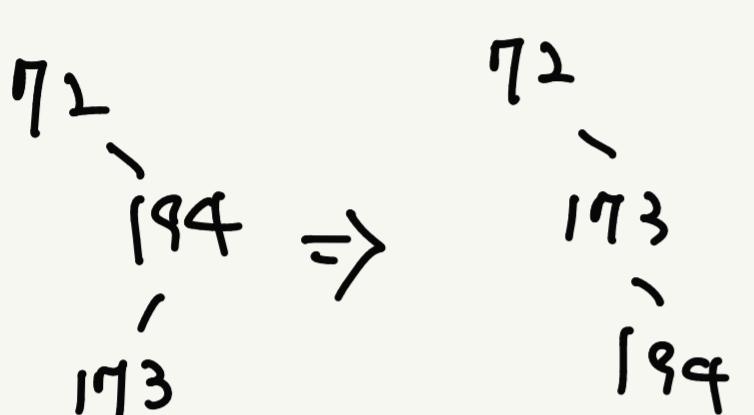
→ mid의 data < grand-parent의 data, grand-parent의 왼쪽을 mid로 변경

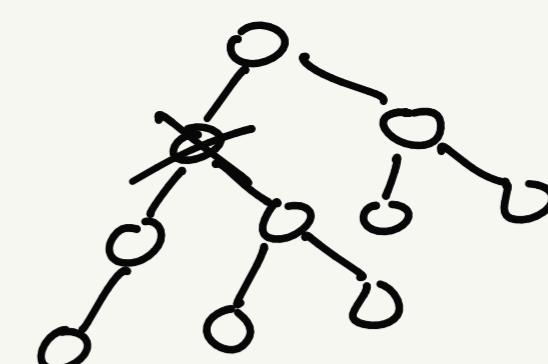
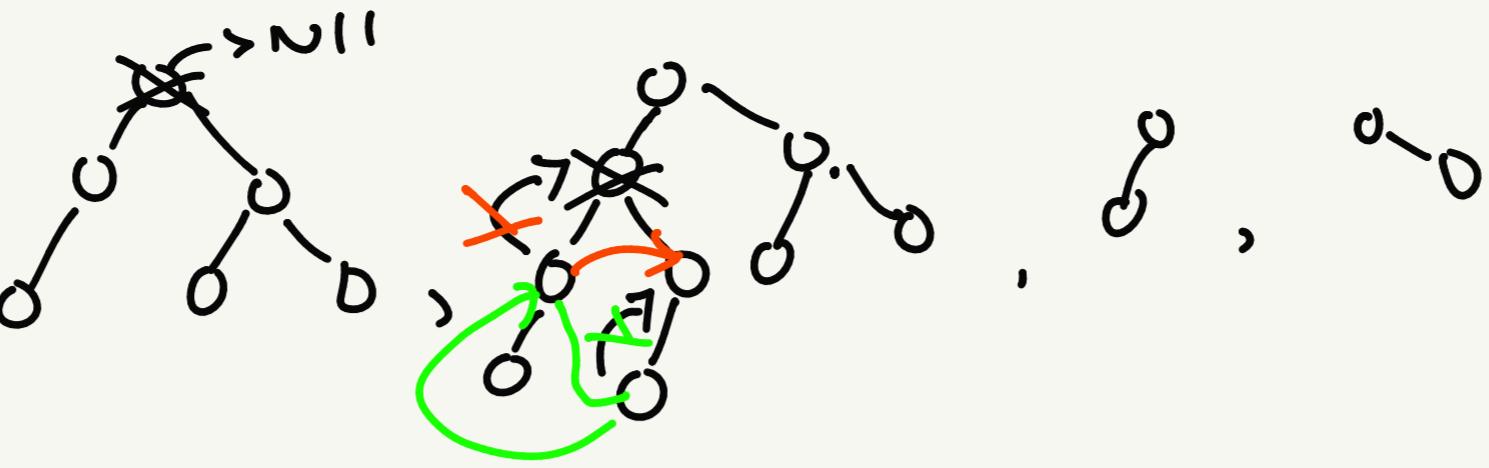
→ mid의 data > grand-parent의 data, grand-parent의 오른쪽을 mid로 변경

\* 72, 194, 173, 161, 133, 158, 200



• B 노드를 지우는 경우





\* 삭제 전략

1. 삭제할 노드의 `data`가 저장된 노드를 찾고, 삭제 노드의 `parent`를 백업한다.
2. 삭제 노드의 자식 노드가 있는지 확인  $\rightarrow$  `del_node`

(1) 양쪽 다 있는 경우 : 왼쪽 최대와 오른쪽 최소값을 삭제 노드의 위치로 변경

① 왼쪽 최대 값의 경우 : `left_max`

- `del_node->data = left_max->data`

- `left_max->left`가 존재하면

`left_max->left->parent = left_max->parent`

`left_max` 노드와 `left_max->left` 노드 위치를 서로 바꾼다.

- `left_max`의 빠진 위치 초기화

② 오른쪽 최소값의 경우 : `right_min`

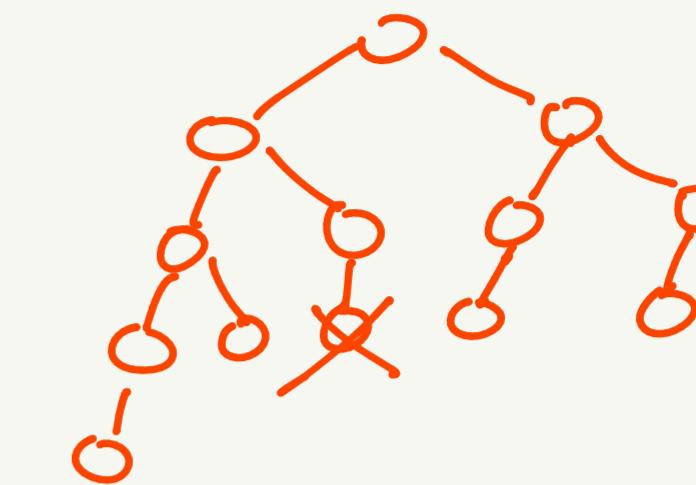
- `del_node->data = right_min->data`

- `right_min->right`가 존재하면

`right_min->right->parent = del_node`

`right_min` 노드와 `right_min->right` 노드 위치를 서로 바꾼다.

- `right_min`의 빠진 위치 초기화

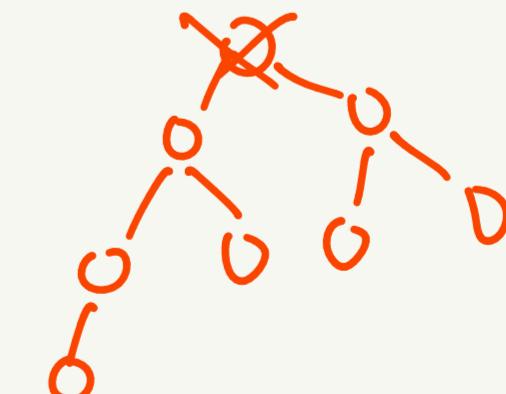


- 삭제된 노드의 `parent`, `grand-parent` 찾기

- Parent factor check ↗  
- Parent balancing  
- parent = parent->parent ↘  
    ↳ 부모

- parent->nil/leaf  
    ↳ 반복 종료

o root가 삭제되었는가?



left\_max의 `data`가 null

(2) 왼쪽만 있는 경우

①  $\text{del\_node} \rightarrow \text{left} \rightarrow \text{parent} = \text{del\_node} \rightarrow \text{parent}$

②  $\text{del\_Node} \rightarrow \text{data} = \text{del\_node} \rightarrow \text{left} \rightarrow \text{data}$

③  $\text{del\_node} \rightarrow \text{left} \text{의 } \text{left} \text{ 를 } \text{None}$

(3) 오른쪽만 있는 경우

①  $\text{del\_node} \rightarrow \text{right} \rightarrow \text{parent} = \text{del\_node} \rightarrow \text{parent}$

②  $\text{del\_node} \rightarrow \text{data} = \text{del\_node} \rightarrow \text{right} \rightarrow \text{data}$

③  $\text{del\_node} \rightarrow \text{right} \text{의 } \text{left} \text{ 를 } \text{None}$

(4) 자식노드가 없는 경우

-  $\text{del\_node} \text{의 } \text{left} \text{ 를 } \text{None}$

3.  $\text{del\_node}$ 의 level 재설정하기

~~4. find-root 함수를 이용하여 root를 찾는다.~~  $\rightarrow$  ~~root가 del\_node인 경우는 고려하지 del\_node 부터 update까지 root ~ del\_node까지~~

~~- del\_node 부터 parent까지 null이면~~

~~해당 노드를 반환~~

4.  $\text{root} - \text{del\_node} \rightarrow \text{left}$  re-balancing을 처리한다.

(1) rebalancing

① 삭제가 되면서 level이 변하는 노드

- left\_max의 parent

- del\_node의 parent

- root

② left\_max로 인한 불균형 처리  
factor를 계산

↓

체인여부 판단 ( $\text{factor} > 1$ )

↓

factor가 1보다 크면  
회전시작

+

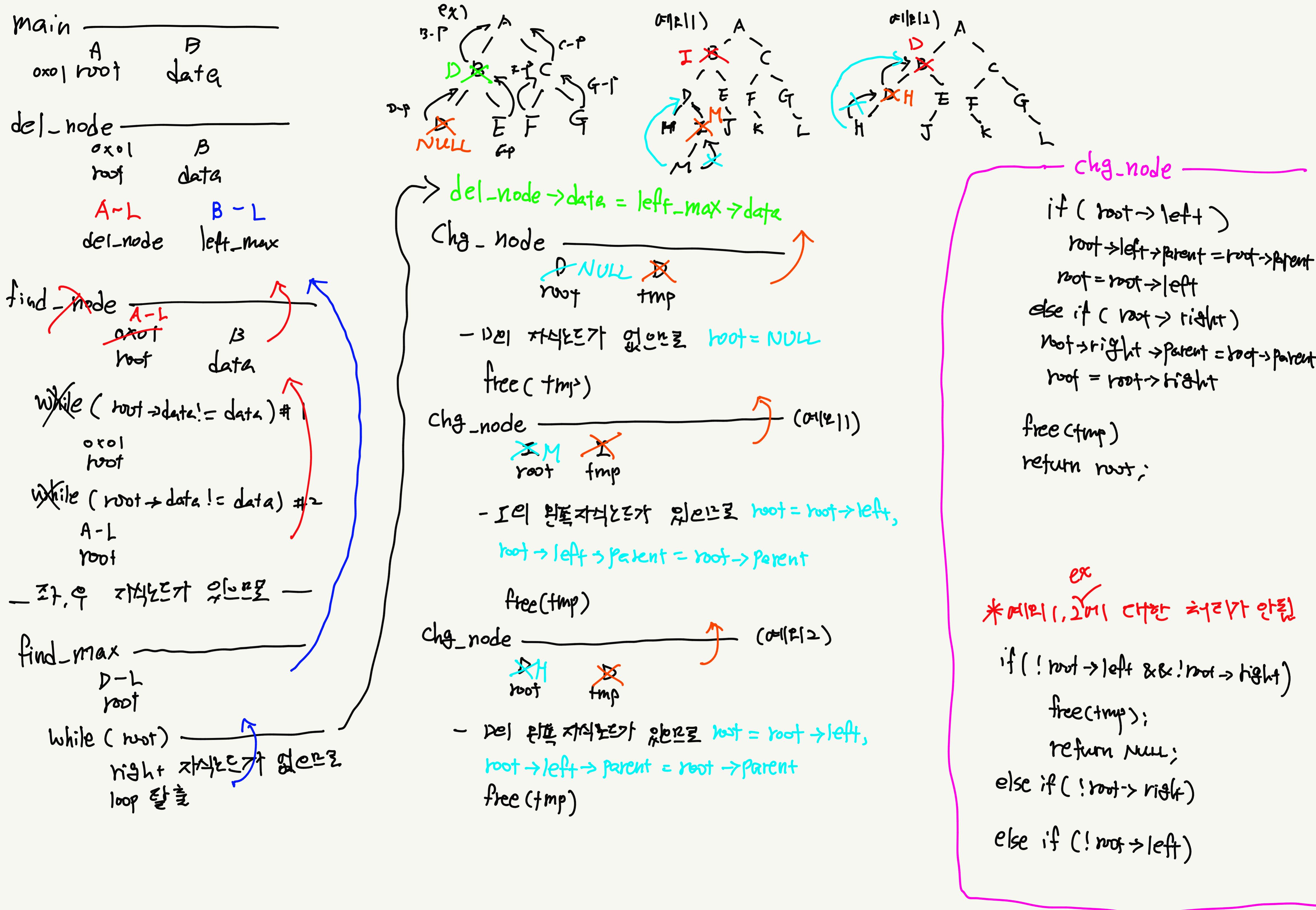
factor > 0 이면 LL, LR 경우  
factor < 0 이면 RR, RL 경우

↓

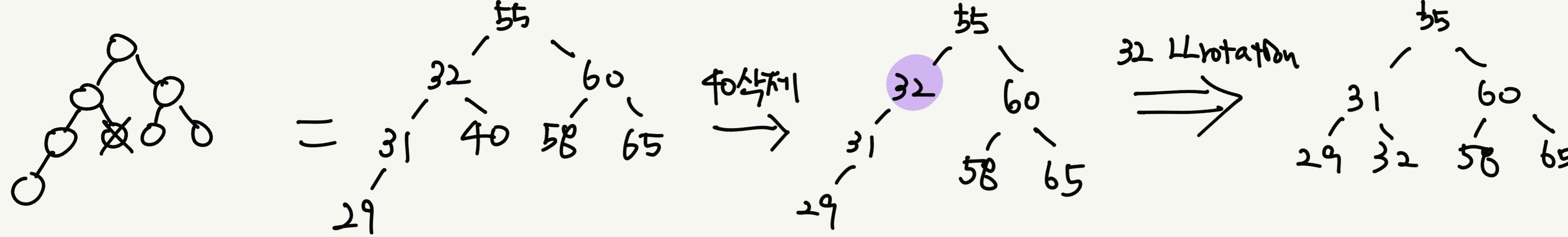
factor > 0 이면 왼쪽 자식노드의  
자식노드 여부 확인

↓

왼쪽에 있으면 LL  
오른쪽에 있으면 LR



## \* del\_node로 인해 불균형 발생



## ◦ random\_delete\_test 헤더 만들기

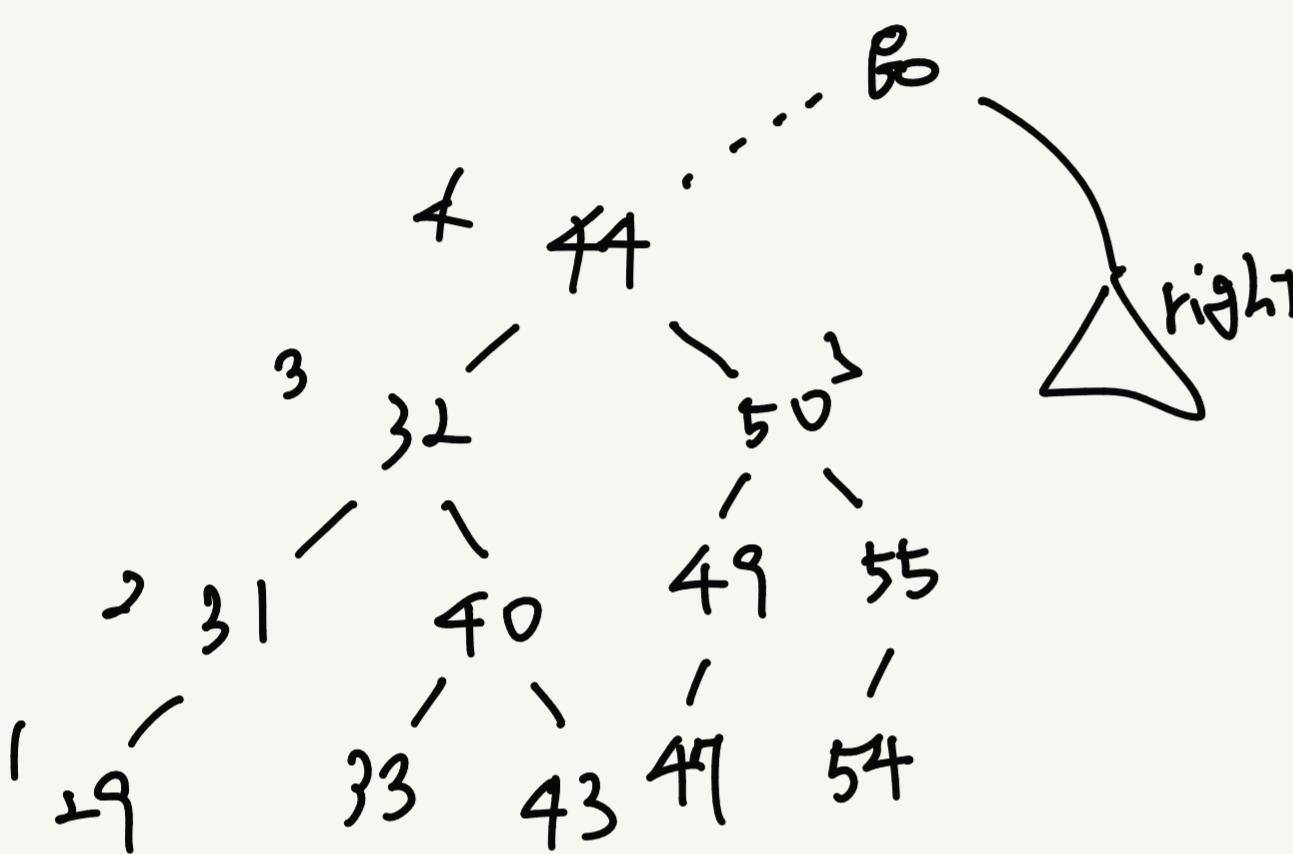
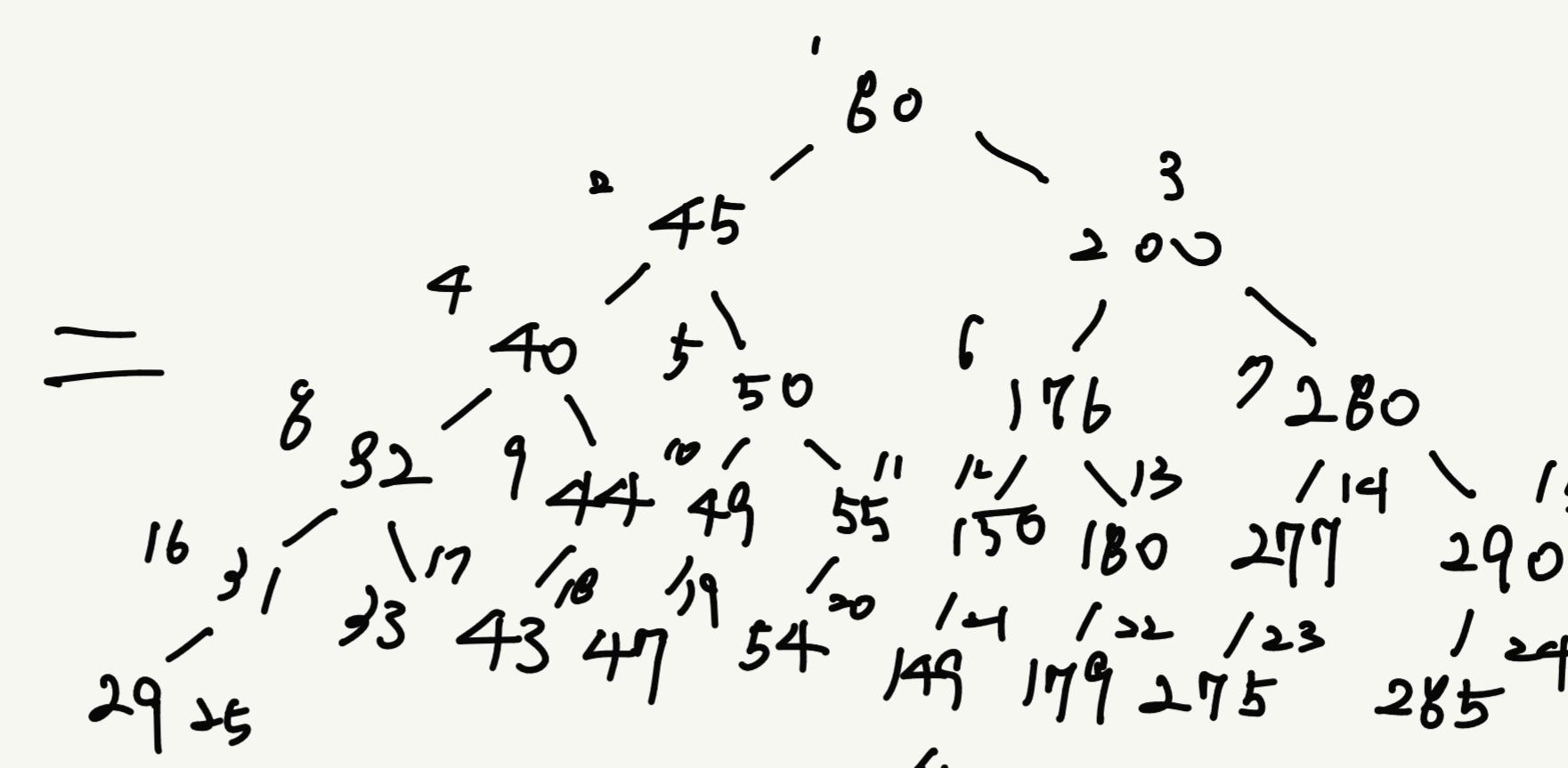
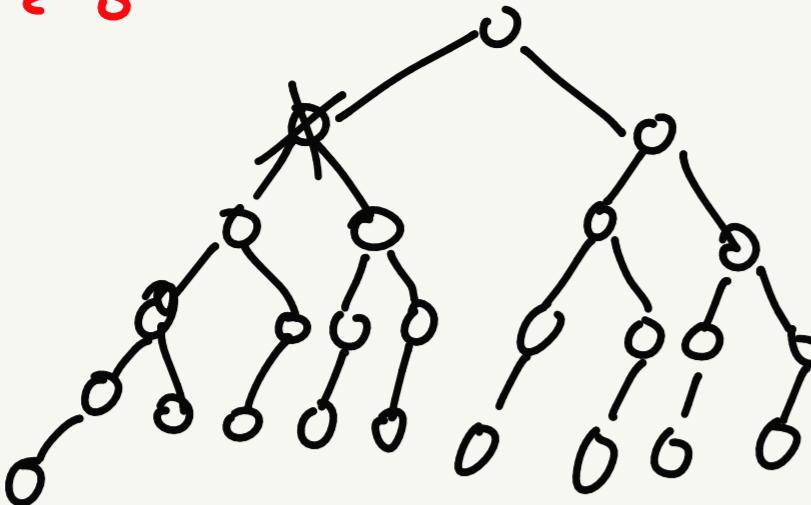
- (1) tree 데이터 갯수 내에서 삭제할 data 갯수 지정 (num)
- (2) 랜덤한수로 tree 데이터 무작위 지정
  - ① 랜덤 숫자가 tree 데이터와 일치하면 return 숫자
  - ② " " 블루치하려면 다시 랜덤한수 실행
- (3) 숫자 삭제
- (4) 삭제할 data 수만큼 반복

### \* idx 중복 방지

- random으로 뽑은 idx 값을 배열에 저장한다.
- 배열내 data 중 중복값을 확인한다.

- (1) 배열의 크기를 tree 데이터 수 내로 random 한수를 이용하여 정한다.
- (2) 반복문을 실행시키며 배열에 랜덤으로 idx 값을 저장한다.
- (3) 배열을 실행시키며 (idx 값) idx 배열에 저장된 값들과 중복되는지 check
- (4) 중복이 있으면 (2)로 강제로 Jmp

\* left\_maxZ 인 경우 흔적발생



40LL rotation

