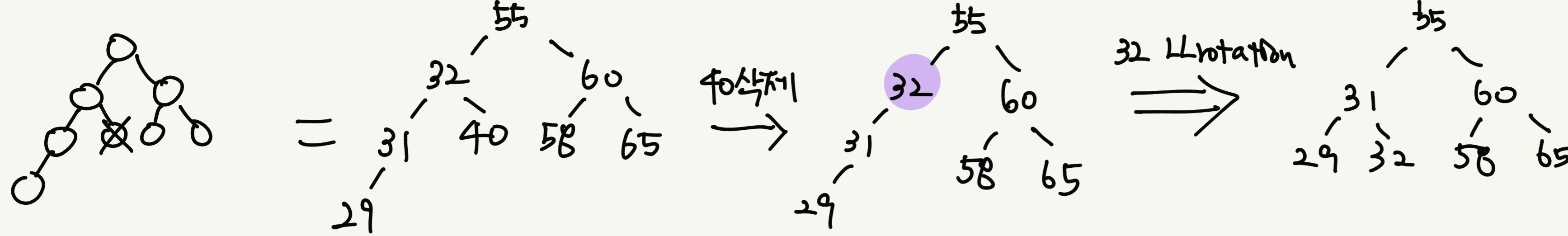


* del_node로 인해 불균형 발생



◦ random_delete_test 헤더 만들기

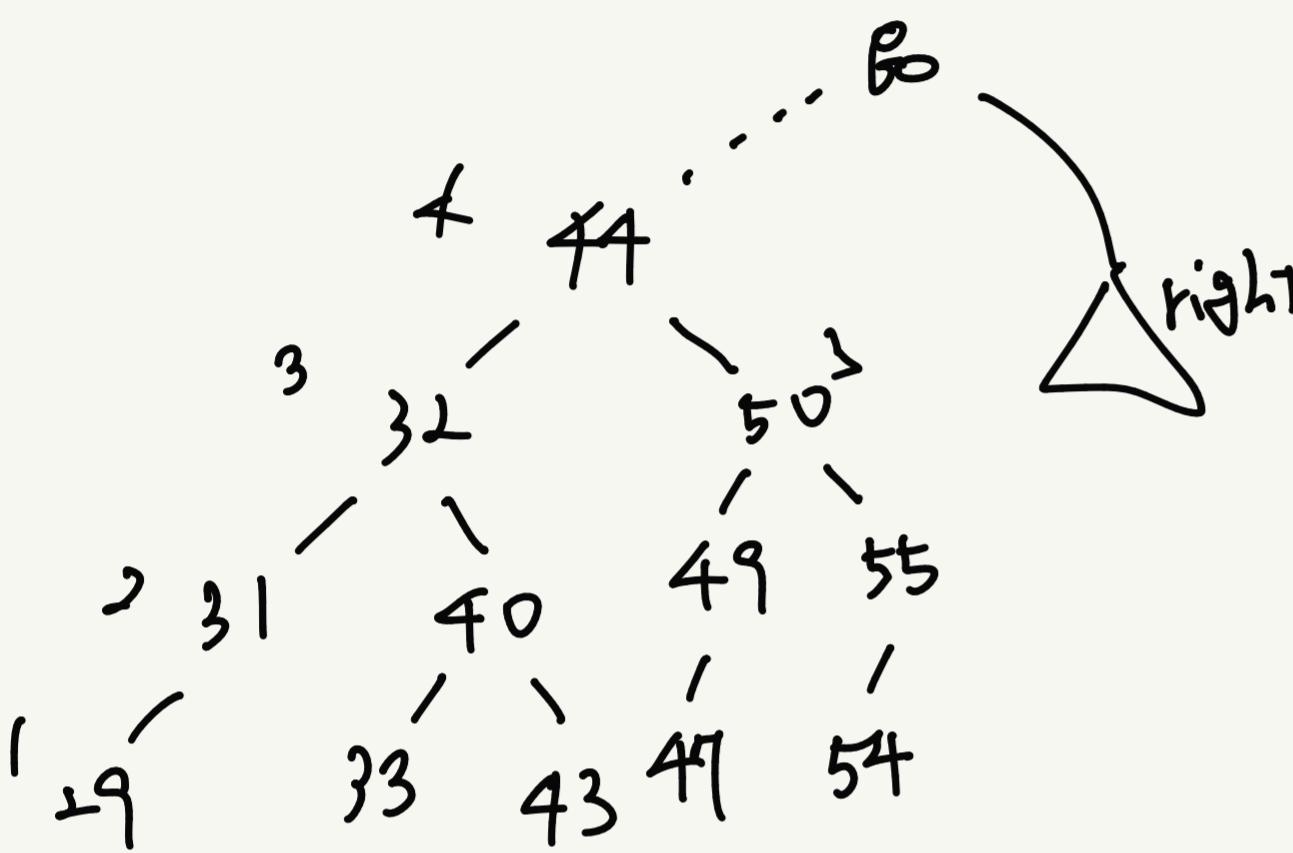
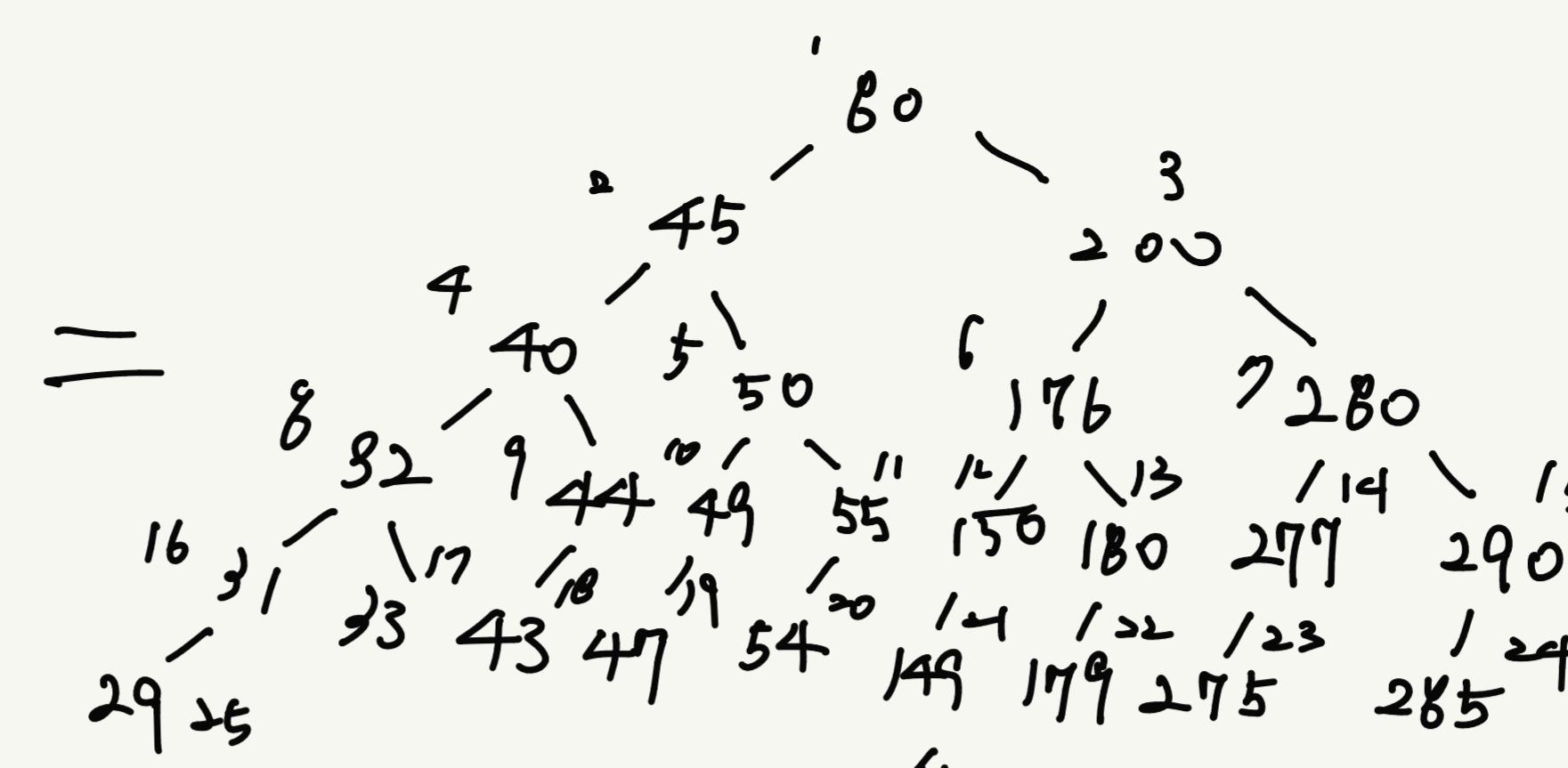
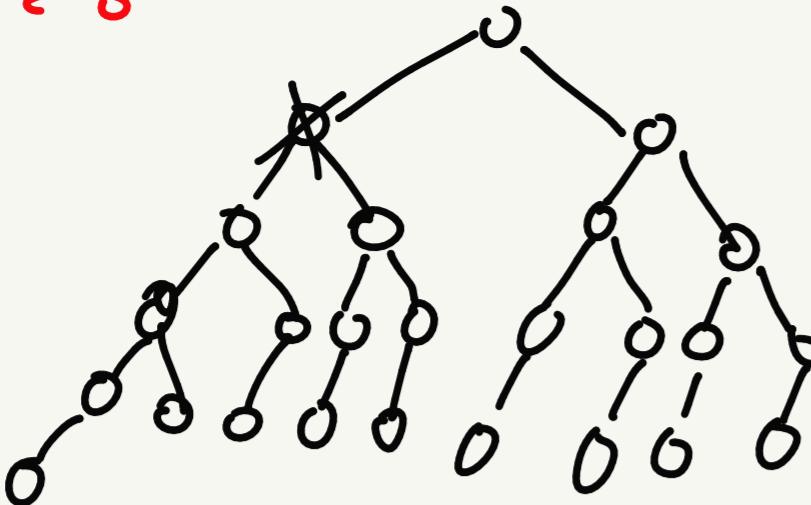
- (1) tree 데이터 갯수 내에서 삭제할 data 갯수 지정 (num)
- (2) 랜덤한수로 tree 데이터 무작위 지정
 - ① 랜덤 숫자가 tree 데이터와 일치하면 return 숫자
 - ② " " 블루치하려면 다시 랜덤한수 실행
- (3) 숫자 삭제
- (4) 삭제할 data 수만큼 반복

* idx 중복 방지

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

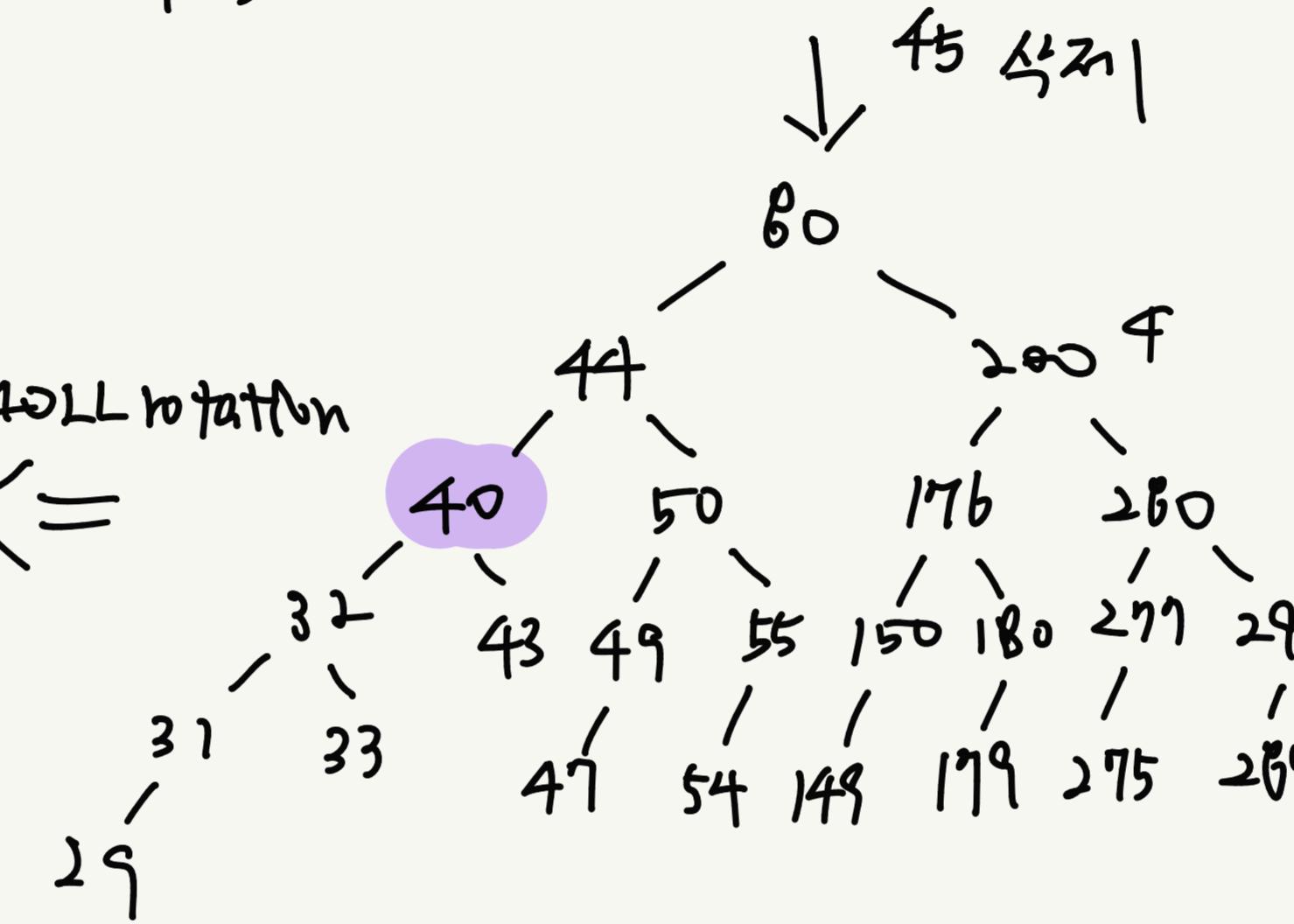
- (1) 배열의 크기를 tree 데이터 수 내로 random 한수를 이용하여 정한다.
- (2) 반복문을 실행시키며 배열에 랜덤으로 idx 값을 저장한다.
- (3) 배열을 실행시키며 (idx 값) idx 배열에 저장된 값들과 중복되는지 check
- (4) 중복이 있으면 (2)로 강제로 Jmp

* left_maxZ 인 경우 흔적발생



right

40LL rotation



Main —————

del-node —————
0x01 B
root data

A~L B - L
del-node left_max

~~find-node~~ $\xrightarrow{A-L}$

~~root~~ $\xrightarrow{\beta}$ data

~~while (root->data != data) # 1~~

~~root~~

~~while (root->data != data) # 2~~

~~A-L~~

~~root~~

— 조건에 자식노드가 있으므로 —

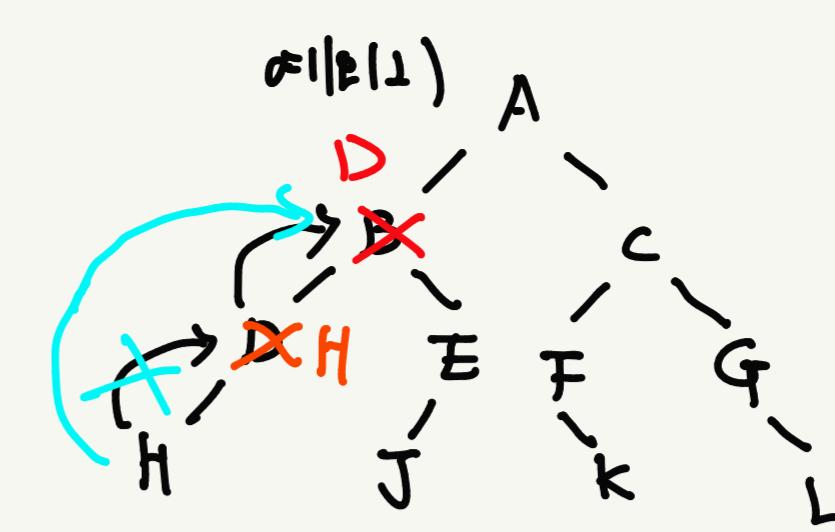
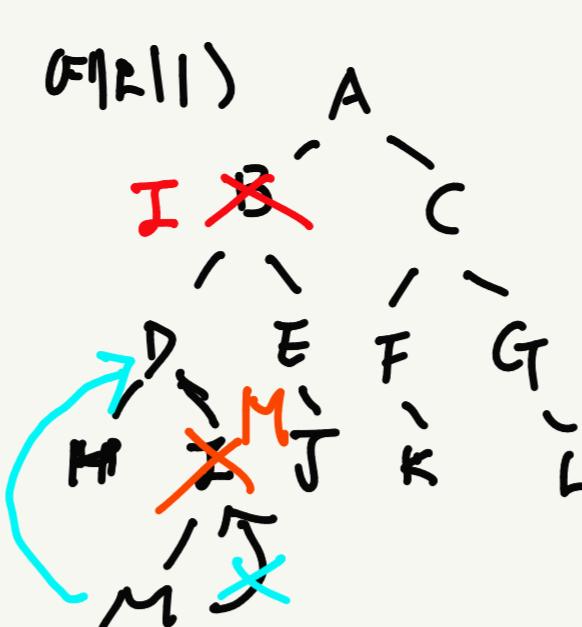
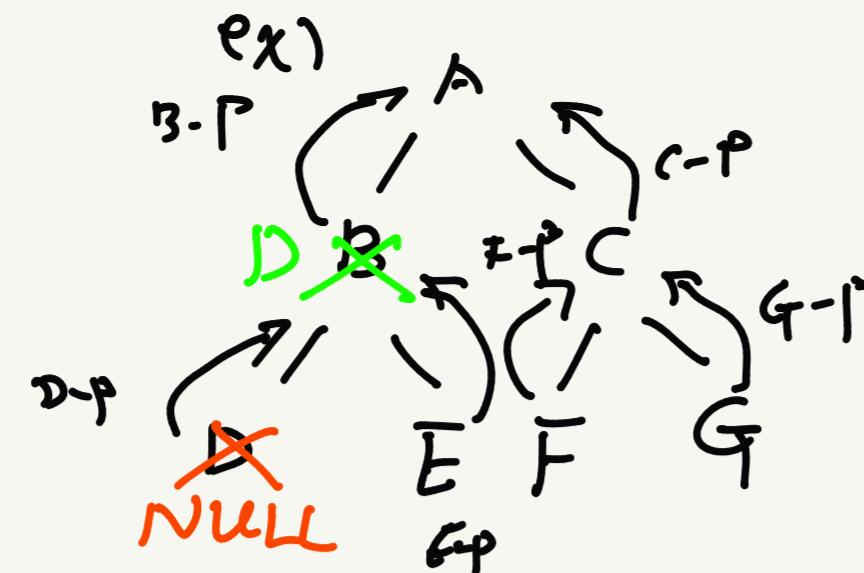
~~find_max~~ $\xrightarrow{D-L}$

~~root~~

~~while (root)~~

~~right 자식노드가 없으므로~~

~~loop 탈출~~



→ $\text{del_node} \rightarrow \text{data} = \text{left_max} \rightarrow \text{data}$

Chg-node —

- 1)의 자식노드가 없으므로 $root = NULL$

free(tmp^s)

chq_node ————— (예외!!)

~~XM~~ root ~~I~~ fmp

- I의 왼쪽자식노드가 있으므로 $\text{root} = \text{root} \rightarrow \text{left}$

$\text{root} \rightarrow \text{left} \rightarrow \text{parent} = \text{root} \rightarrow \text{parent}$

`free(fmp)`

chg_node ————— ↑ (≈1%2)

~~root~~ ~~tmp~~

- 예) 왼쪽 자식노드가 있으면 $host = root \rightarrow left$

$\text{root} \rightarrow \text{left} \rightarrow \text{parent} = \text{root} \rightarrow \text{parent}$

free (tmp)

→ chg-node →

if (root → left)
root → left → parent = root → parent
root = root → left

else if (root → right)
root → right → parent = root → parent
root = root → right

free (tmp)

return root;

* ex) 예외 1, 2에 대한 처리가 안됨

if (!root->left && !root->right)

free(tmp);

return null;

else if(!root->right)

else if (!root->left)

(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. 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

\downarrow
체인여부 판단 ($\text{factor} > 1$)

\downarrow
factor가 1보다 크면
회전시작

\downarrow

$\text{factor} > 0$ 이면 LL, LR 경우
 $\text{factor} < 0$ 이면 RR, RL 경우

\downarrow

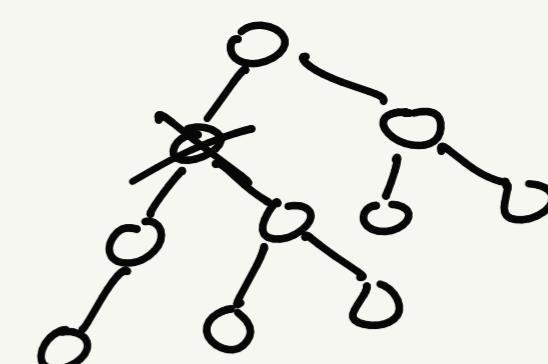
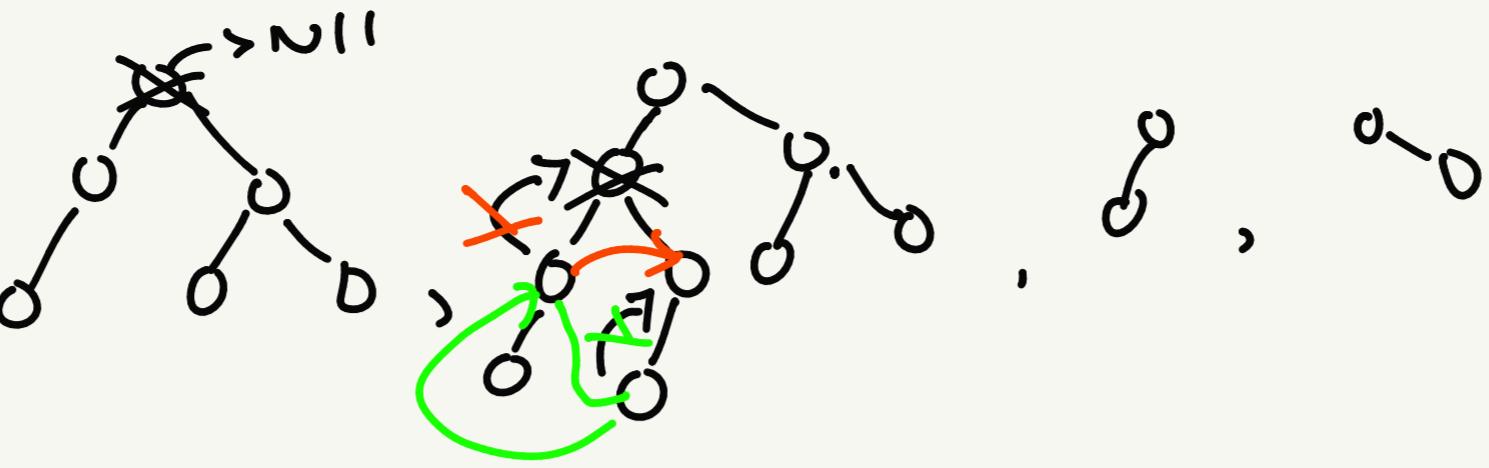
$\text{factor} > 0$ 이면 왼쪽 자식노드의
자식노드 여부 확인

\downarrow
왼쪽에 있으면 LL
오른쪽에 있으면 LR

$\text{root} \rightarrow \text{del_node}$ 인 경우는 고려하지
update까지 root ~ del_node까지

top-down 방식 (X)

- left_max, 삭제된 노드, root에 대한
level 재설정이 끝난 balancing



* 삭제 전략

- 삭제할 노드의 $data$ 가 자장된 노드를 찾고, 삭제 노드의 $parent$ 를 백업한다.
- 삭제 노드의 자식 노드가 있는지 확인 $\rightarrow del_node$

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

① 왼쪽 최대 값의 경우 : $left_max$

- $del_node \rightarrow data = left_max \rightarrow data$

- $left_max \rightarrow left \rightarrow$ 존재하면

$left_max \rightarrow left \rightarrow parent = left_max \rightarrow parent$

$left_max$ 노드와 $left_max \rightarrow left$ 노드 위치를 서로 바꾼다.

- $left_max$ 의 빠져나온 부모

② 오른쪽 최소값의 경우 : $right_min$

- $del_node \rightarrow data = right_min \rightarrow data$

- $right_min \rightarrow right \rightarrow$ 존재하면

$right_min \rightarrow right \rightarrow parent = del_node$

$right_min$ 노드와 $right_min \rightarrow right$ 노드 위치를 서로 바꾼다.

- $right_min$ 의 빠져나온 부모

- 삭제된 노드의 $parent$, $grand_parent$ 풀오

- Parent factor check ↗

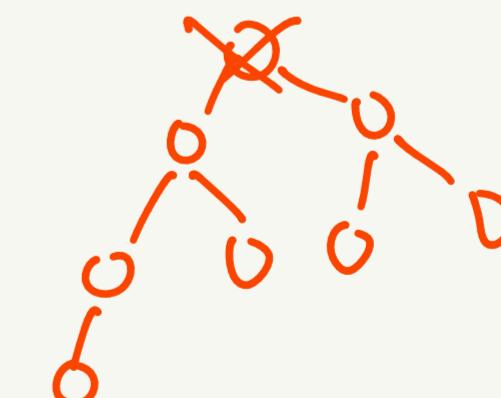
- Parent balancing ↘

- parent = parent \rightarrow parent

- parent \rightarrow null or not

반복 종료

o root가 삭제되었는가?



left_max의 data 정수가 같을 때