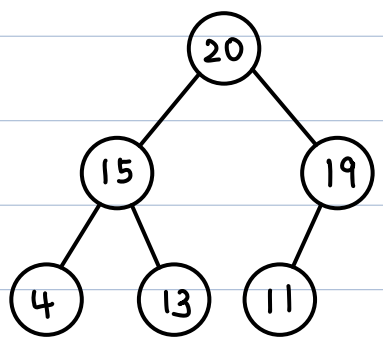
7. Heap

↳ PQ 구현시 사용!

: 모든 노드에 관해서 부모와 자식 간에 일정한 대소관계가 성립하는 완전 이진 트리



→ Max Heap : 부모 노드 키 값 > 자식 노드 키 값

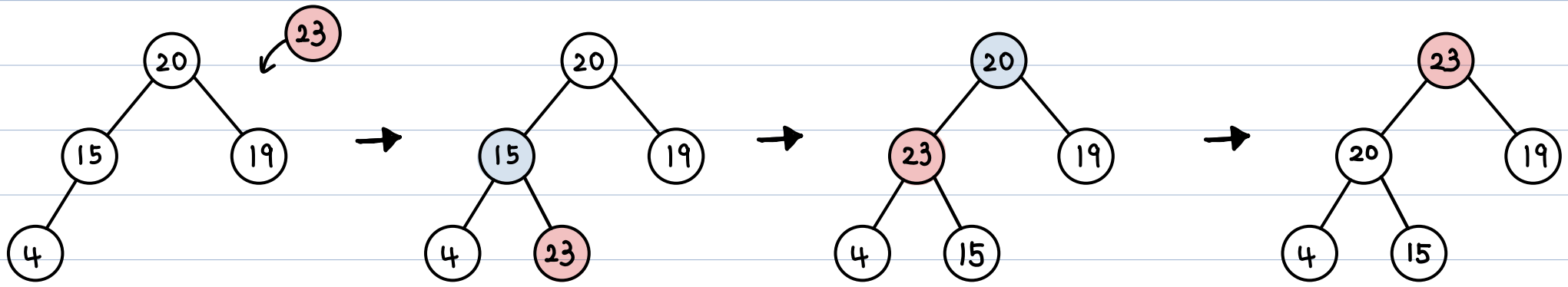
Min Heap : 부모 노드 키 값 < 자식 노드 키 값

- **삽입 연산**

1) 비어있는 자리에 값 추가

2) 부모-자식 간 우선순위 비교 → 위치 변경

3) 자리확정



- **삭제 연산**

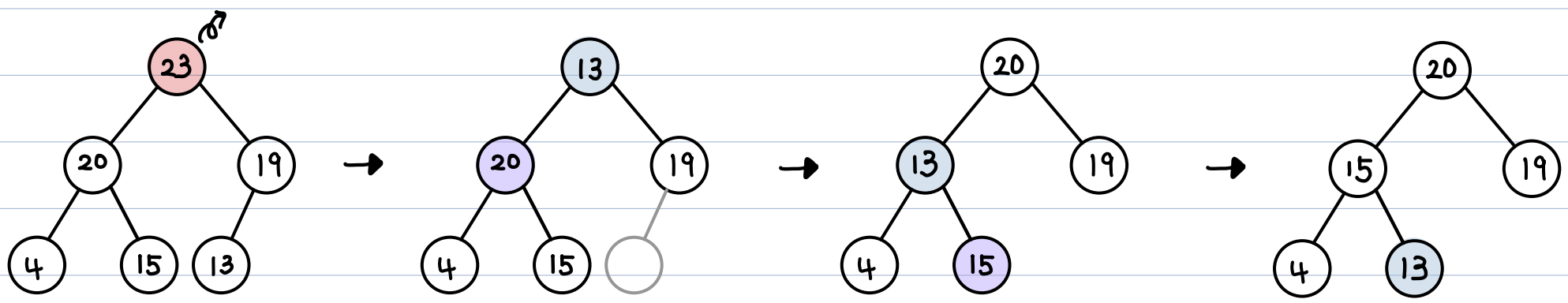
1) 루트 노드 삭제

2) 마지막 data를 루트에 저장

3) 부모-자식 간 우선순위 비교 → 위치 변경

4) 자리확정

→ if) 자식노드가 2개라면 더 큰 키값을 가지는 자식과 비교 (Max Heap 기준)

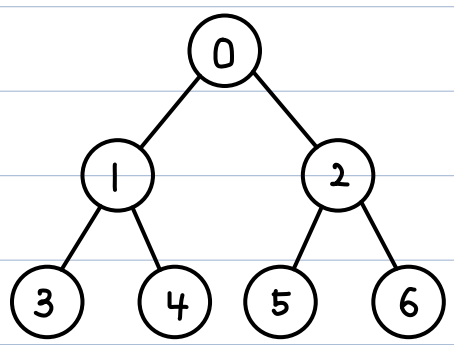


- Heap의 구현

1) 배열

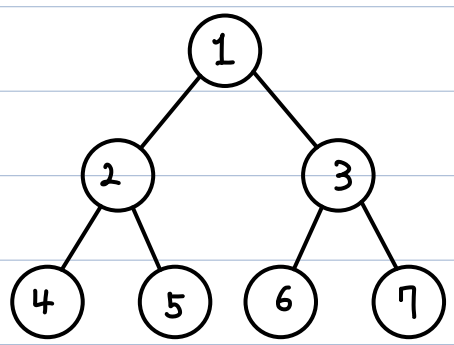
* 완전 이진트리의 index

루트노드의 index가 0 일 때



- 부모 = (자식-1)/2
- 왼쪽 자식 = 부모 x 2 + 1
- 오른쪽 자식 = 부모 x 2 + 2

루트노드의 index가 1일 때



- 부모 = 자식 / 2
- 왼쪽 자식 = 부모 x 2
- 오른쪽 자식 = 부모 x 2 + 1

→ 삽입·삭제 연산의 시간복잡도 = $O(\log_2 N)$

→ 연결 리스트로 구현 시, 새로운 노드를 마지막 위치에 추가하기 어려움

→ 배열이 실행속도 측면에서 효율적! (O(1)의 시간 복잡도로 마지막 위치 접근 가능)

2) Linked List

→ 마지막 노드 찾을 때 실행속도 증가

→ but 메모리 공간 문제 해결

- Priority Queue

- Max heap

```
PriorityQueue<Integer> pq = new PriorityQueue<>(Collections.reverseOrder());
```

- Min heap

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
```

- 활용

1) BFS 우선순위 큐 : 최솟값을 구하는 문제에서 실행시간을 단축시킬 수 있다.

2) 중간값구하기 : Max Heap, Min Heap 사용하여 구현

i. mid 보다 작은 값을 넣는 Max heap, mid보다 큰 값을 넣는 Min heap 선언

ii. 입력된 값을 mid와 비교해서 각각 heap 에 삽입

iii. 두 heap의 크기가 다르면... 크기가 작은 heap에 mid 값을 push 한후, (현재 mid 값이 진짜 mid가 아님.)
크기가 같아질 때 까지 크기가 더 큰 heap의 원소를 pop 해서 mid 값으로 선언