

# Concurrent skip list algorithm

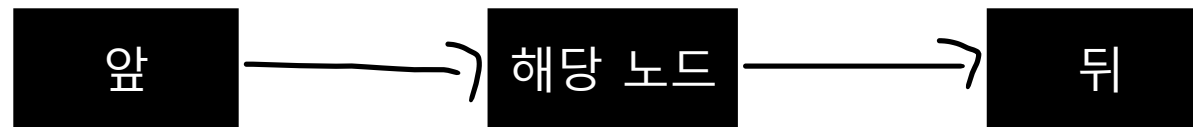
1. Create
2. Search
3. Insert
4. Delete

Presenter: Kim Kyung Min

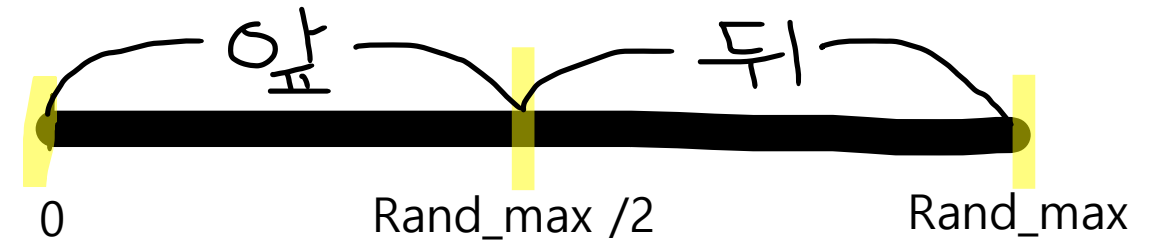
Reference: [Concurrent-Skip-list/README.md at main · shreyas-gopalakrishna/Concurrent-Skip-list \(github.com\)](https://github.com/shreyas-gopalakrishna/Concurrent-Skip-list)

Marked: 해당 노드가 삭제되고 있는지 여부를 판단하는 변수  
marked = 1 : 해당 노드가 삭제되고 있음  
marked = 0 : 해당 노드의 삭제가 이루어지지 않고 있음

Fully-linked: 해당 노드가 선행, 후속 노드와 연결이 되어있는지 여부를 판단하는 변수  
fully-linked = 0 : 선행, 후속 노드와의 연결이 되어있지 않음  
fully-linked = 1 : 선행, 후속 노드와 연결이 되어있음



# Create



- 맨 밑 단에는 전체 리스트가 정렬되어 있음
- 가장 앞과 뒤에는 -무한대, 무한대에 해당하는 값
- 한 층 올라갈 때마다 1/2의 확률로 각 원소가 위 층으로 올라갈 수 있음



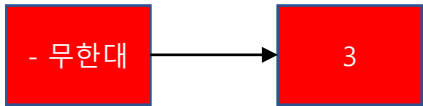
```
38 static int rand_level() {  
39     int level = 1;  
40     while (rand() < RAND_MAX / 2 && level < SKIPLIST_MAX_LEVEL)  
41         level++;  
42     return level;  
}
```

40번째 줄: level이 max level로 지정해 놓은 값보다 작고 rand()로 호출되는 값이 RAND\_MAX / 2 보다 작아야 함

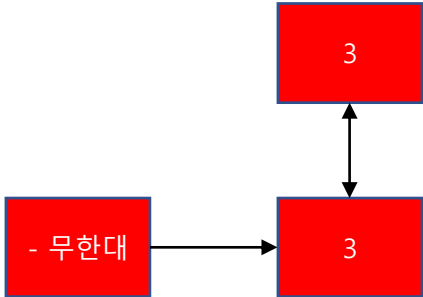


Skip list의 형상이 만들 때마다 달라질 수 있다??

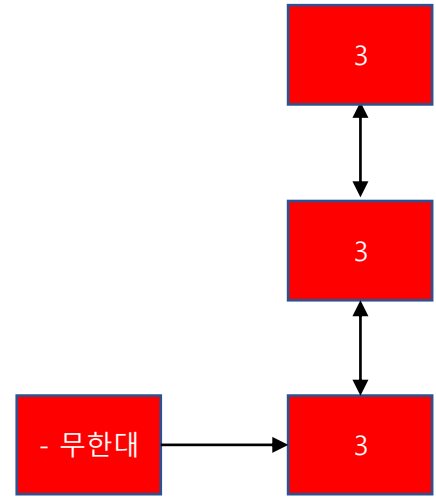




flip  
앞



flip  
앞



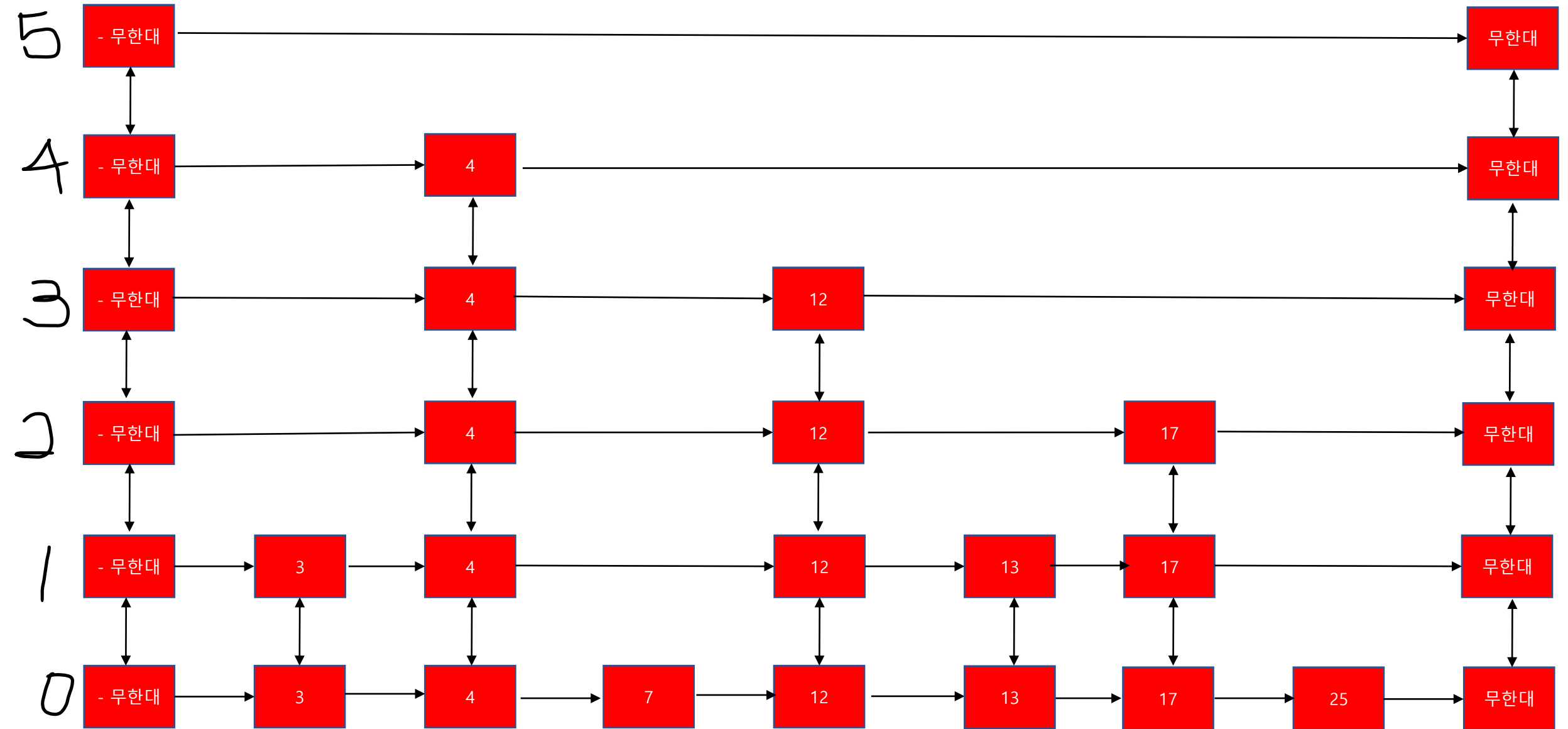
flip | 뒤  
↓

STOP

모든 삽입되는 노드들에  
대해서 동일한 과정 반복

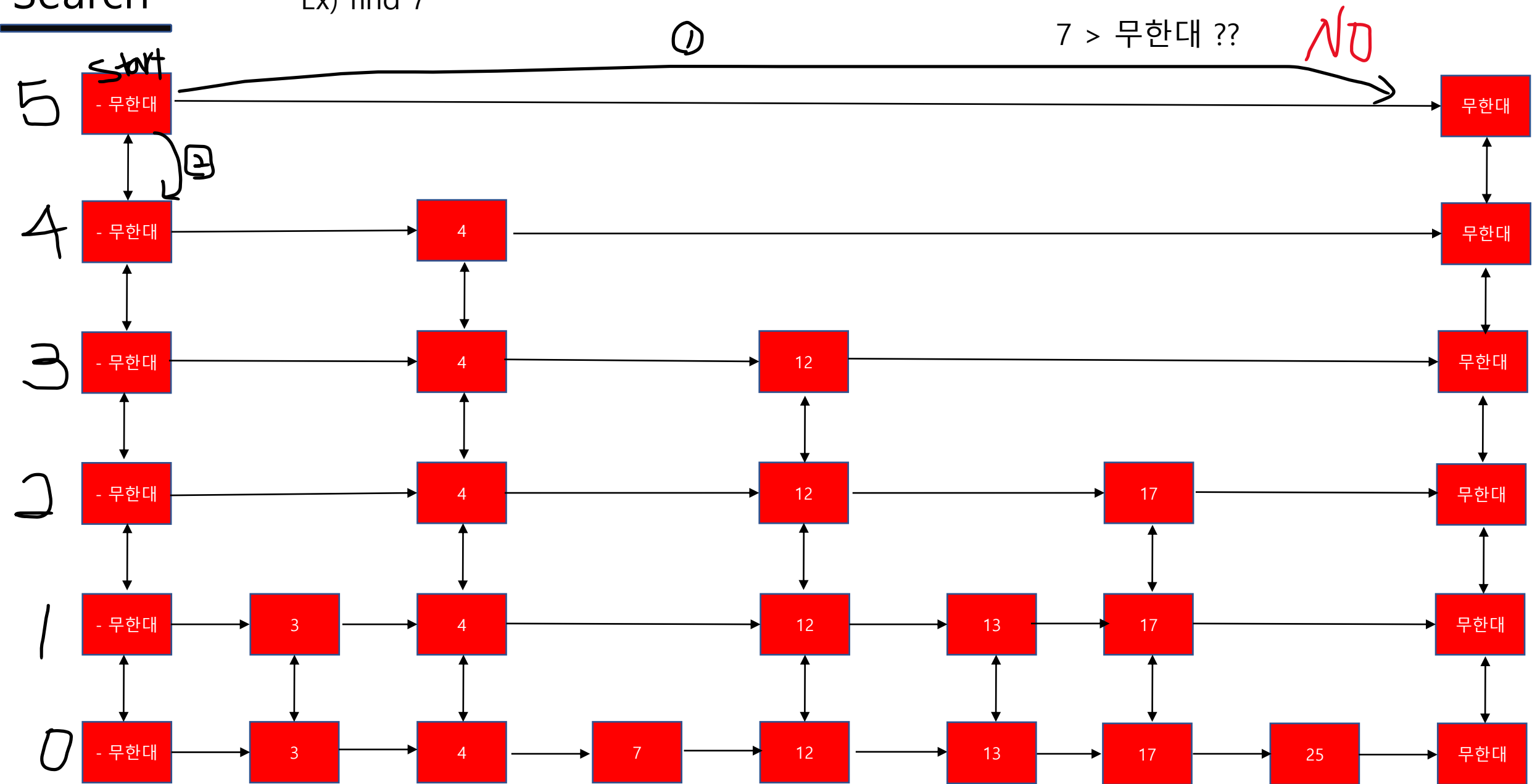


# Create

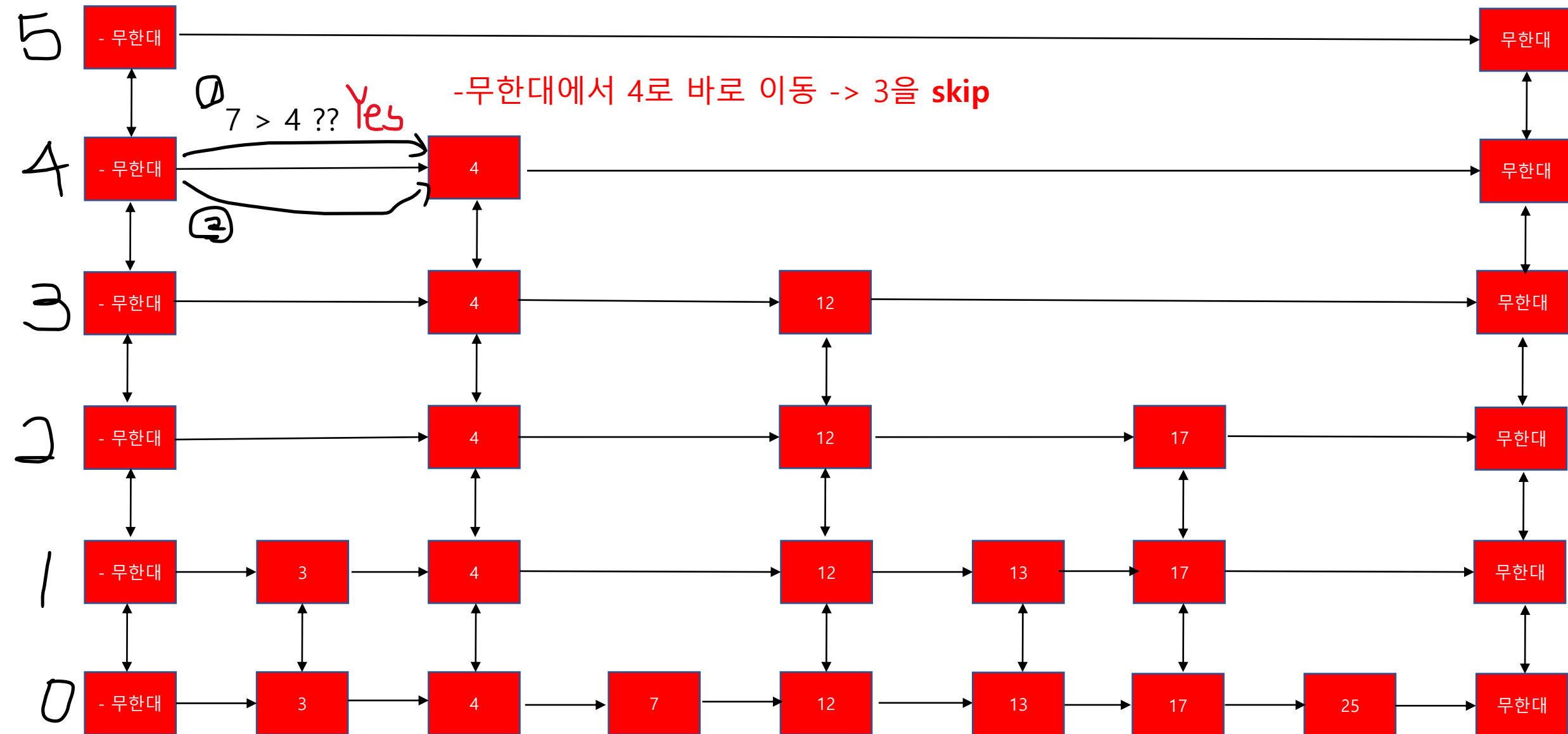


\_\_\_\_\_

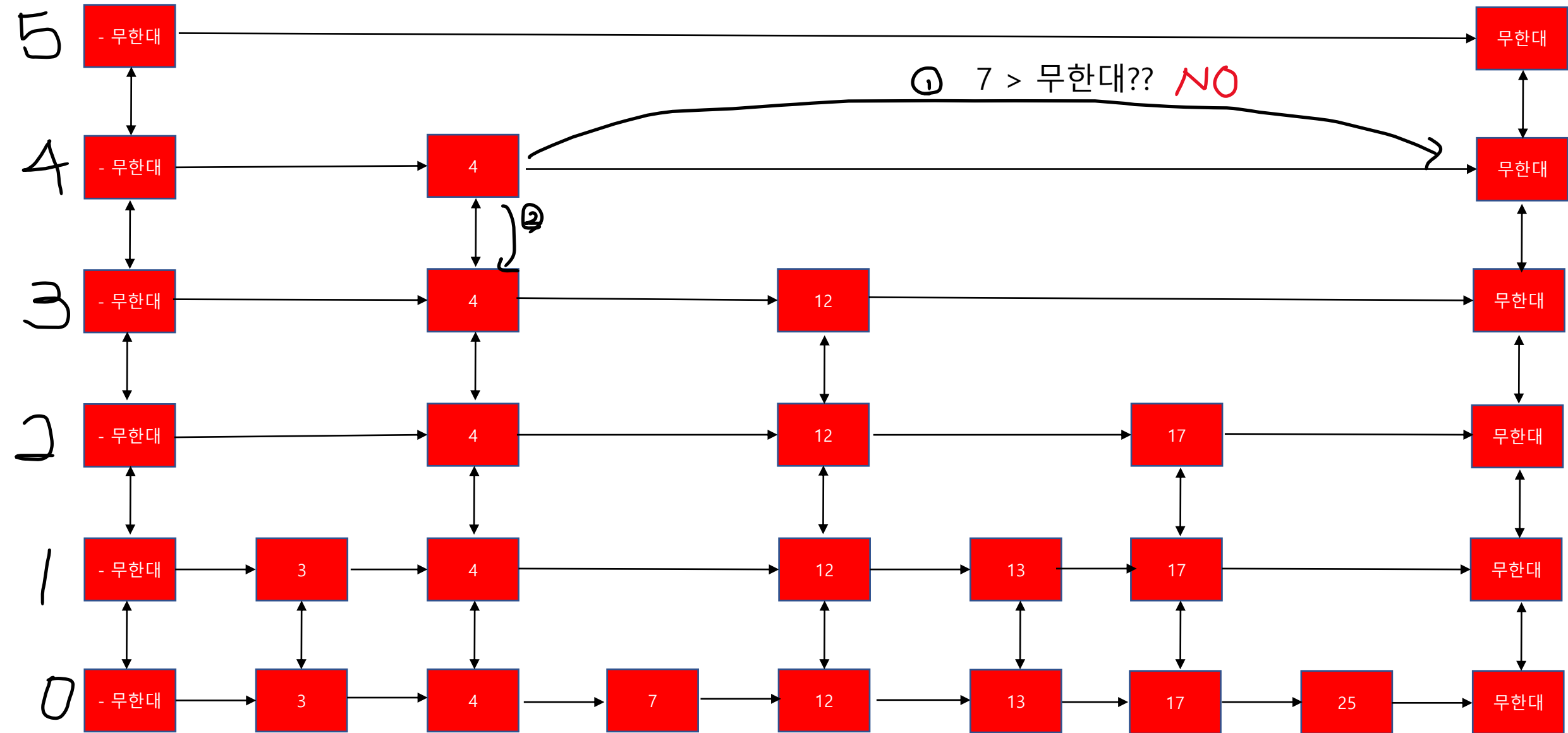
Ex) find 7



# Search

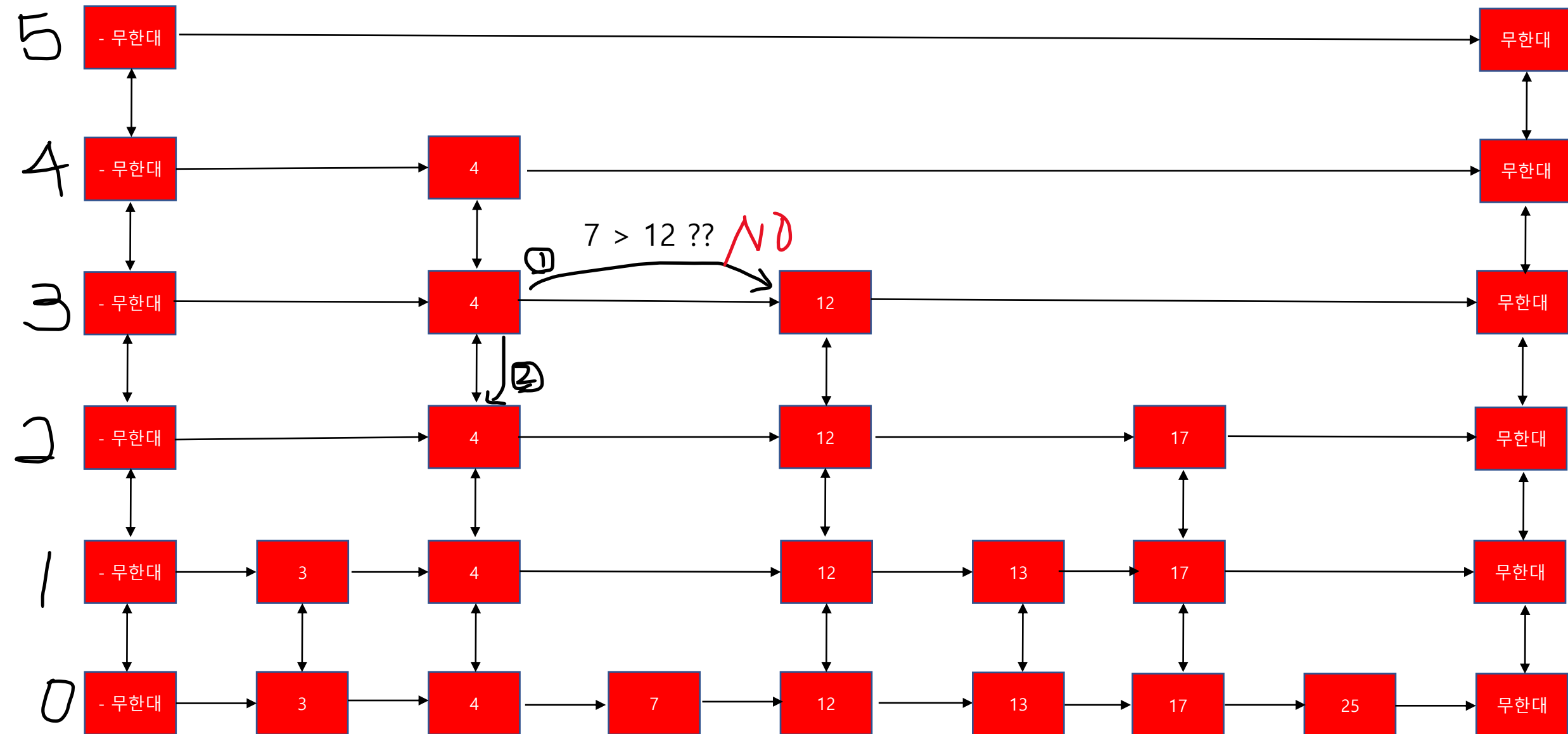


# Search

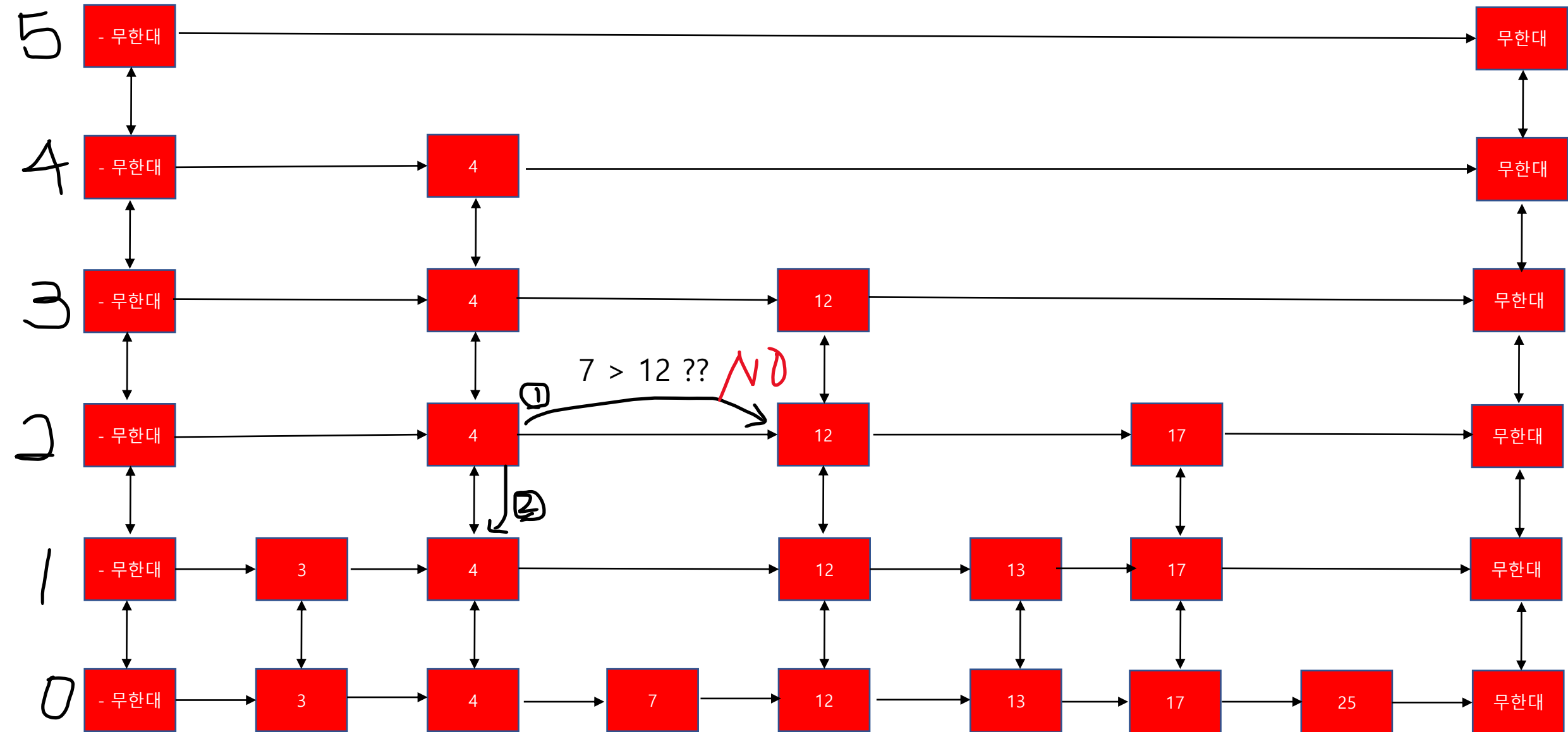




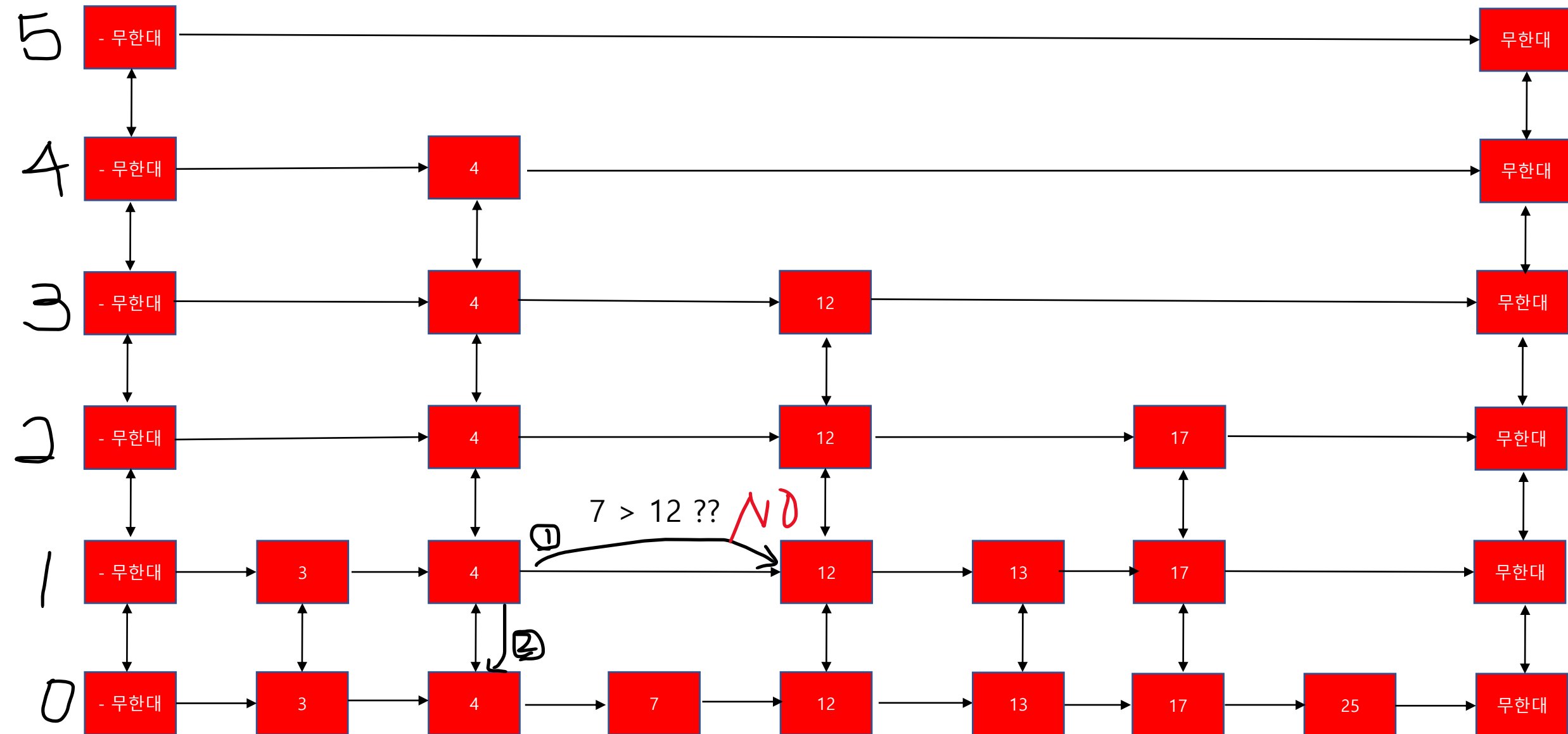
# Search



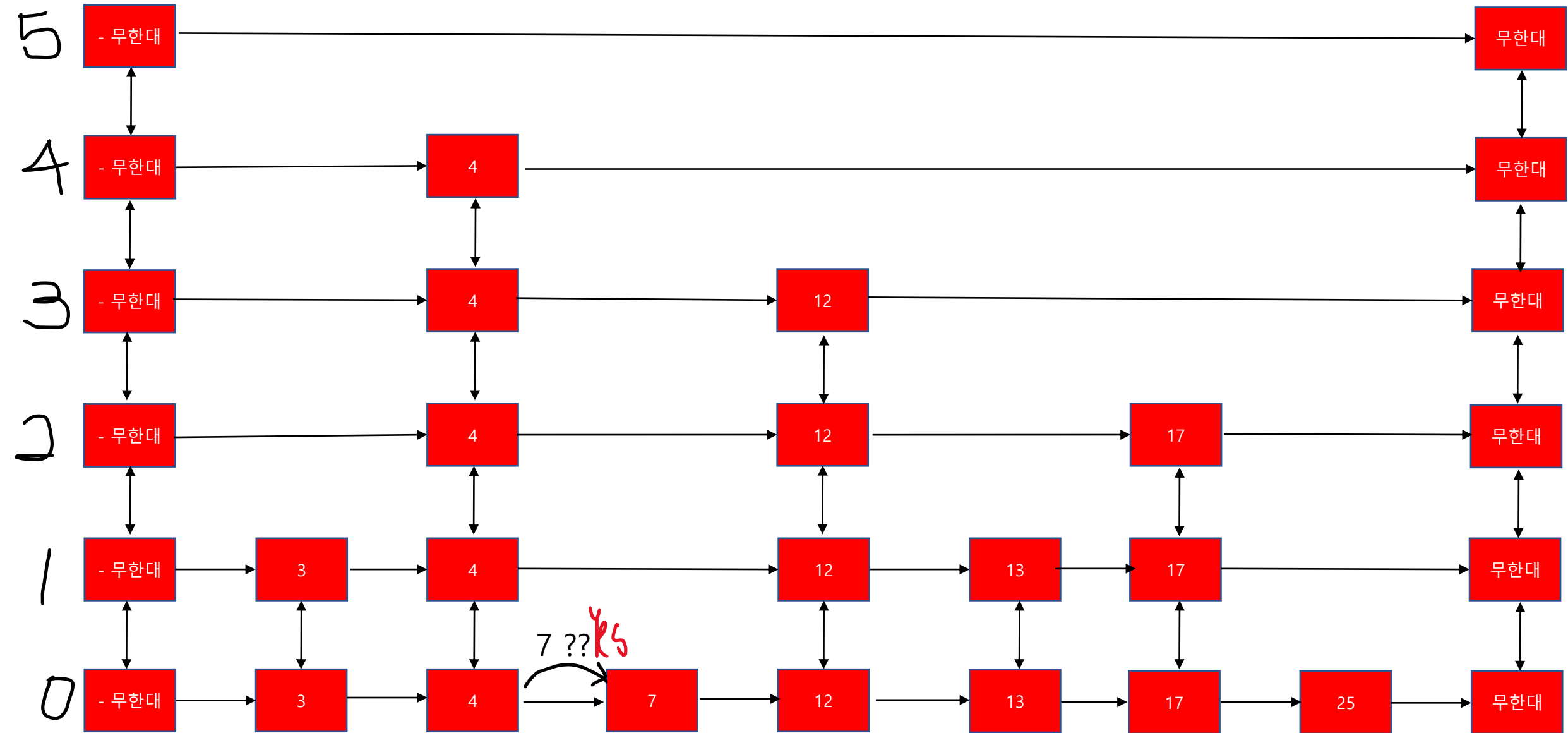
# Search



# Search



# Search



Find 7

1. 검색 키에 가까워질수록 '상위 -> 하위' 레벨로 떨어진다

2. 검색 키 찾았을 경우(원하는 노드 찾은 경우)

If(Marked = 0 and fully-linked = 1)

키와 관련된 값 반환

Else

false

3. 읽는(search) 요청은 병렬적으로 실행할 수 있도록 함 (lock free, non blocking)


특정 스레드가 insert, delete하는 도중에 다른 스레드가 search -> 내용의 일관성 깨짐

특정 스레드가 search하는 도중에 다른 스레드도 search -> 딱히 문제될 게 없어 보임

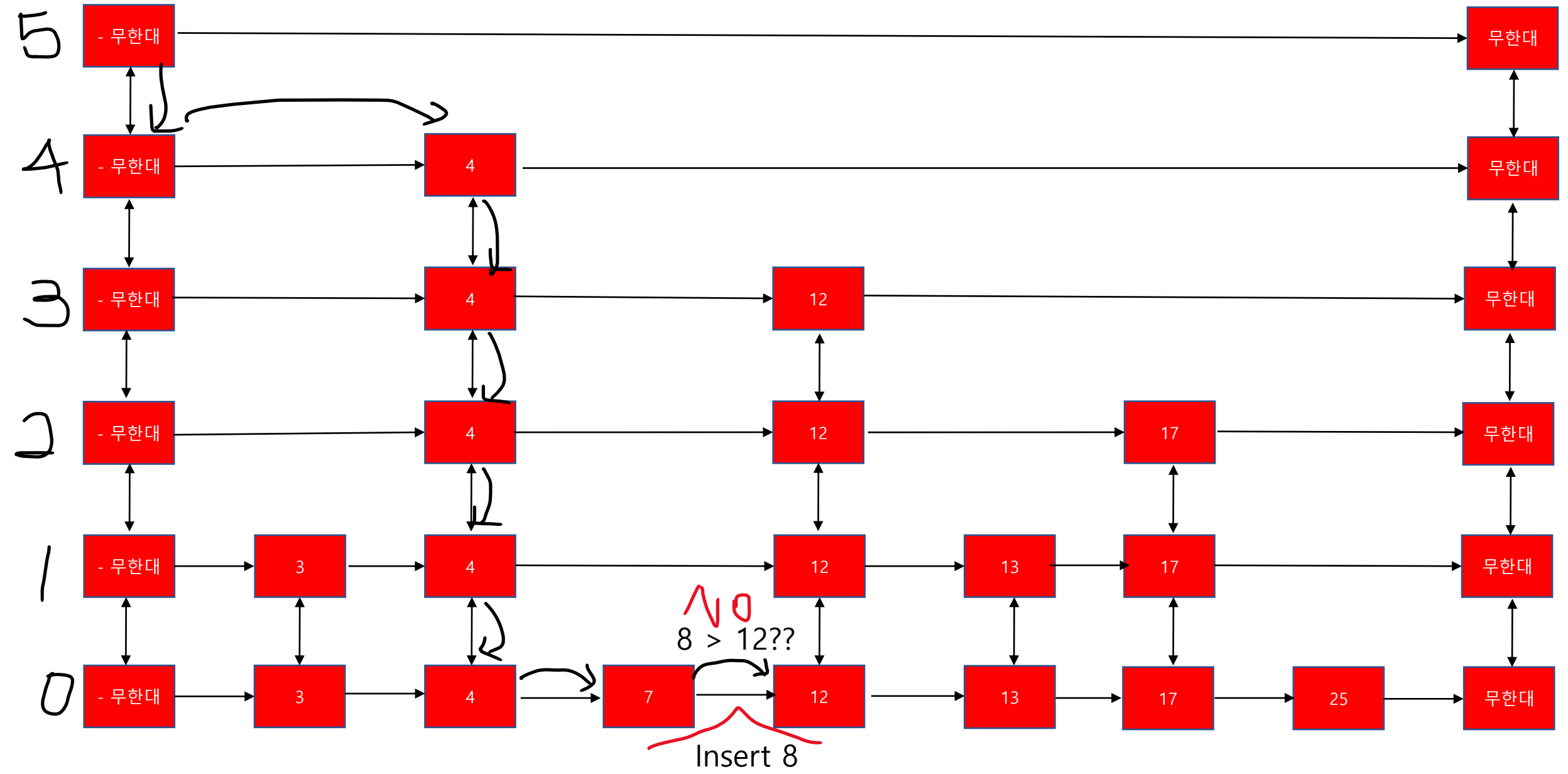
여러 스레드가 search를 동시에 할 수 있으면 lock을 효율적으로 사용할 수 있음

단, 특정 스레드의 search 도중 다른 스레드의 insert, delete가 없음이 보장되어야 함


# Insert

Ex) insert 8 

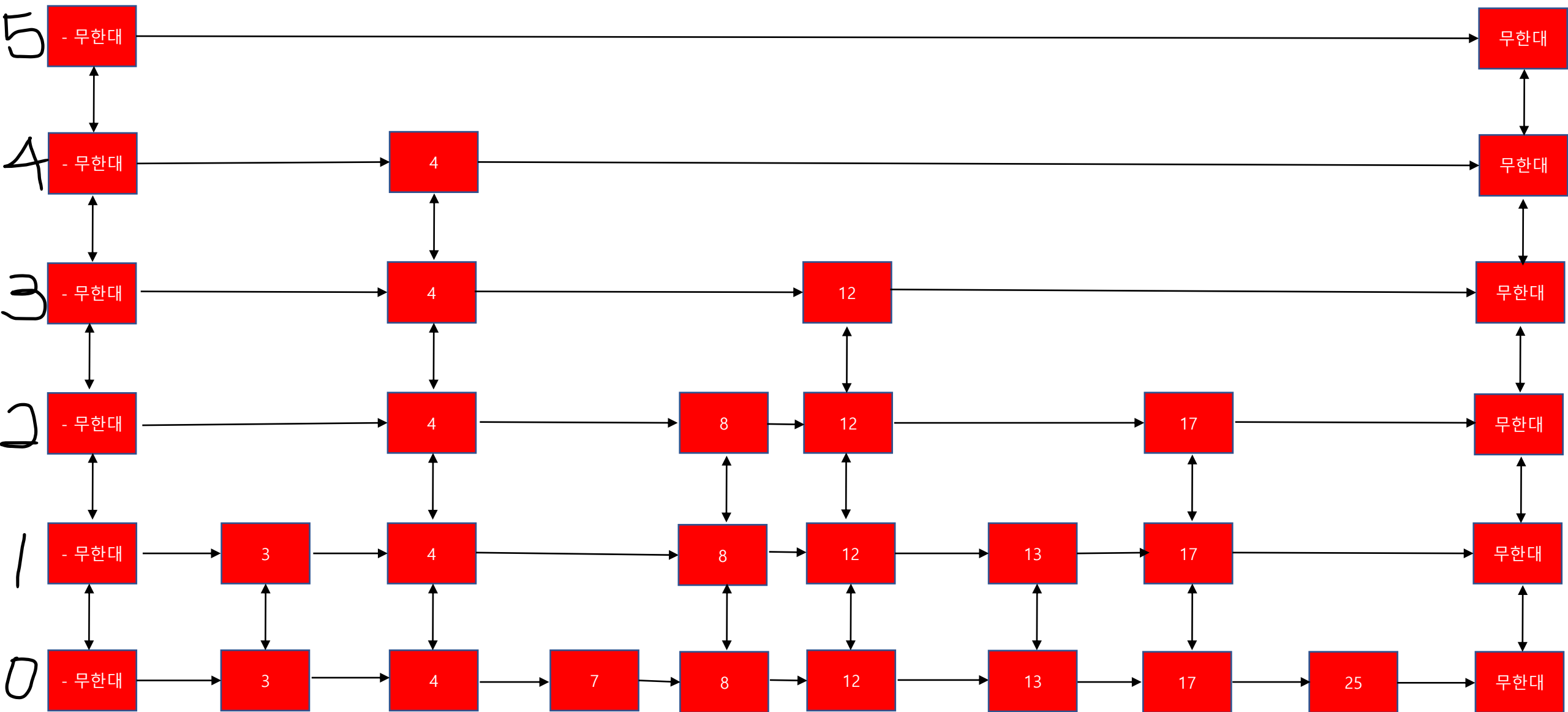
1. Find location
2. Insert(Flip & level)



# Insert

Ex) insert 8 

1. Find location
2. **Insert(Flip & level)**



(삽입 전 요소 유무 확인 및 mark, fully-linked 변수 확인)

1. 요소 이미 있고 노드가 표시되지 않음 -> skiplist에 이미 있으므로 삽입 x
2. 요소 있는데 완전히 연결되지 않은 경우 -> 완전히 연결될 때까지 기다림
3. 요소 존재하는데 노드가 표시되는 경우 -> 노드가 삭제되고 있으므로 기다렸다가 나중에 전체 삽입 알고리즘 재시도

1. 노드 삽입 위치의 선행과 후속에 대한 참조를 찾는다.
2. 선행, 후속 노드 모두 표시되어 있지 않고 선행 노드의 다음이 각 레벨에서 이어받을 것인지 확인할 필요 있음
3. 1, 2번 충족이 되지 않음 -> 기다렸다가 전체 삽입 알고리즘 재시도

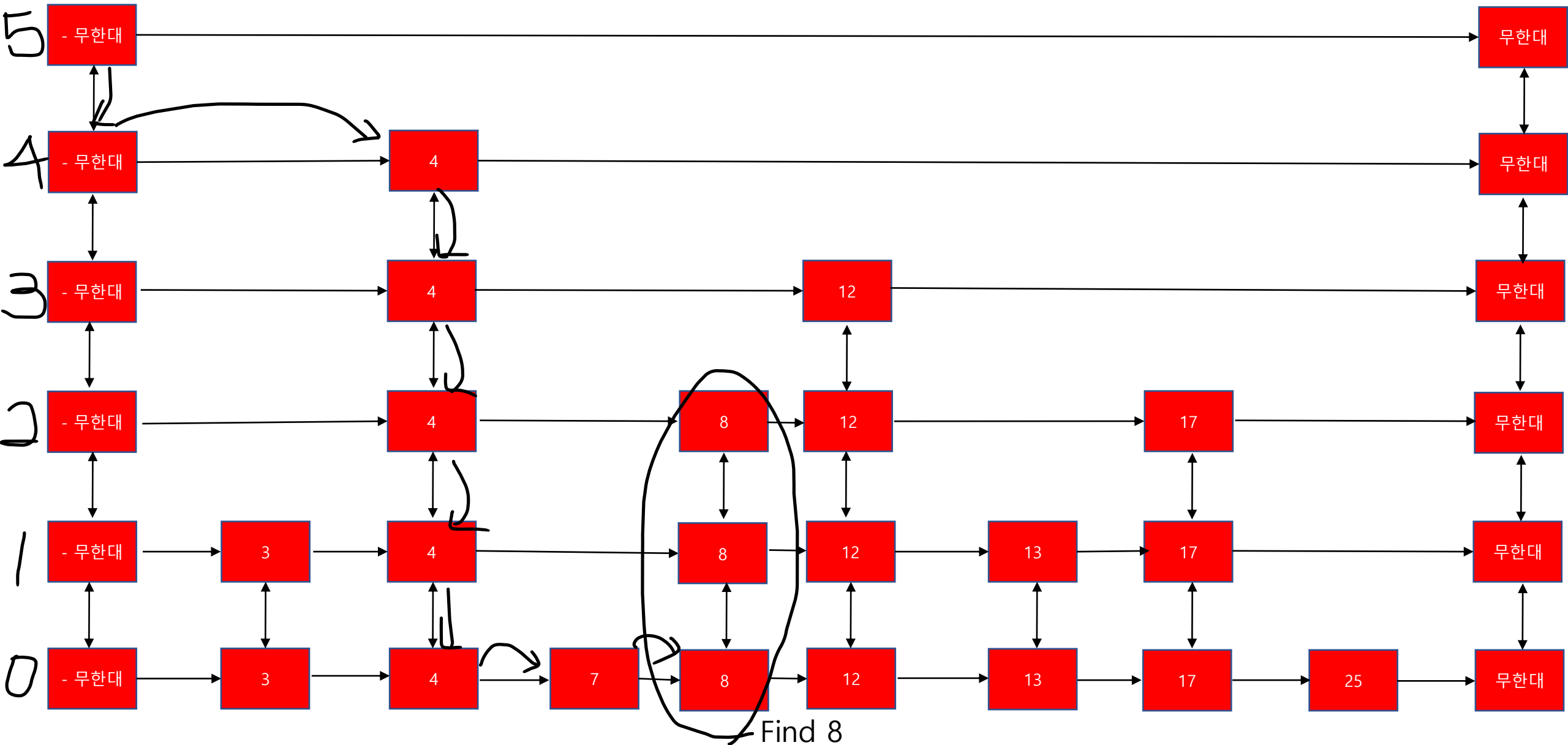
1. 위 조건이 모두 충족되면 선행 노드의 lock을 동시에 유지하기 시작 (후속 노드에 대한 lock은 필요 x)
2. 노드 삽입 전 새 노드의 level이 무작위로 정해짐
3. 선행, new, 후속 노드 연결
4. Fully-linked = 1 -> lock 모두 해제 -> 동시 삽입 완료



# Delete

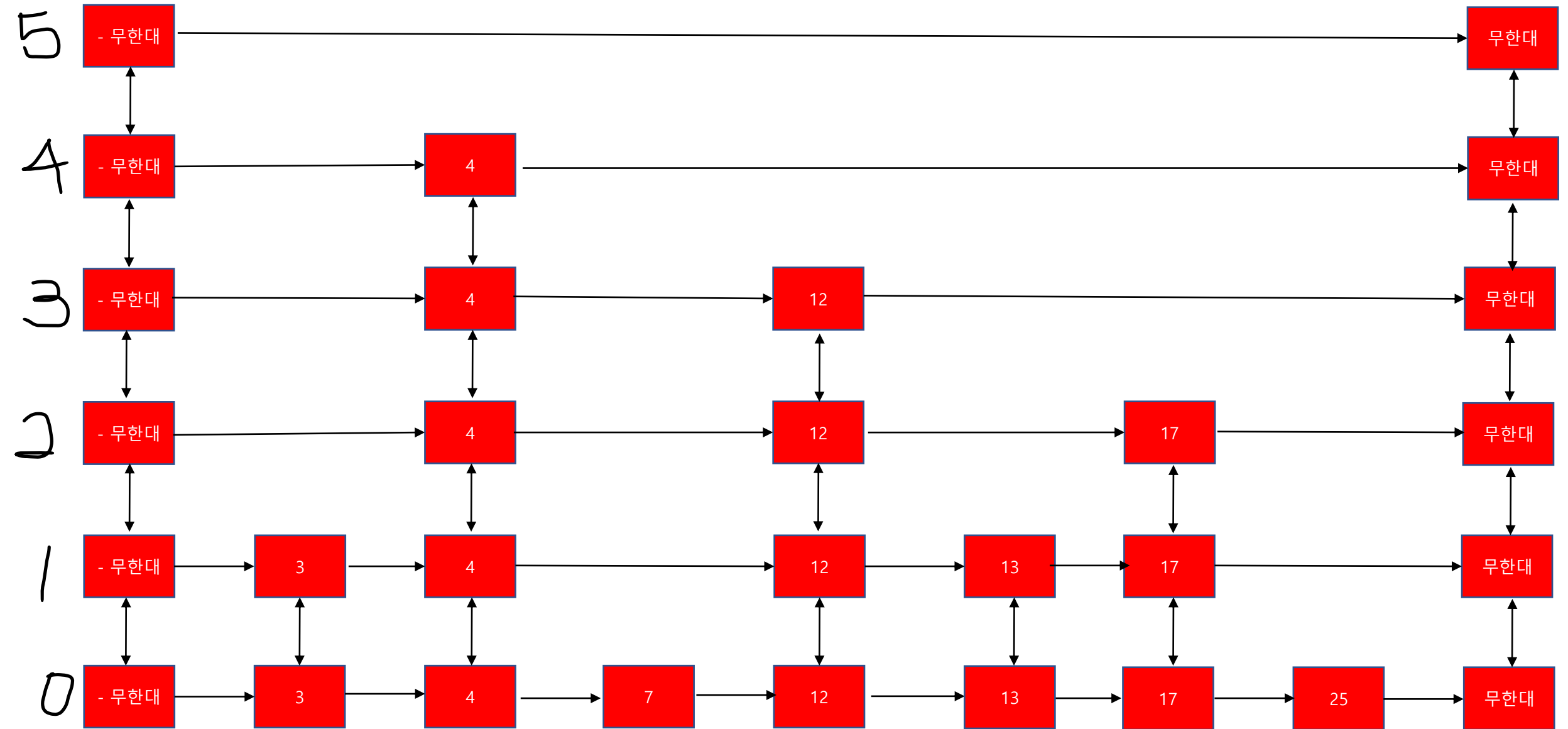
Ex) delete 8

1. Find location
2. Delete node



# Delete

Ex) delete 8 < 1. Find location  
2. **Delete node**



1. 지우려는 노드가 있는지 확인 -> 없으면 false
2. Fully-linked = 1, marked = 0 인지 확인 -> 조건 충족 안되는 경우 삭제 알고리즘 재시도

1. 선행, 후속 노드에 대한 참조를 찾음
2. 삭제할 노드의 lock 획득 -> 선행 노드의 lock 획득
3. 선행 노드가 marked = 0, '선행 노드 -> next = 삭제 노드' 확인
4. 3번 충족 안되면 모든 lock 해제 -> 삭제 알고리즘 재시도

1. 위의 조건 모두 충족 시 노드 삭제 및 선행 노드와 후속 노드 연결
2. Lock 해제하면 동시 삭제 완료

