동아대학교 DONG-A UNIVERSITY

# CSE 206 - 파일 처리론 (File Processing)

## 7 장. M-Way Search Tree, B-Tree

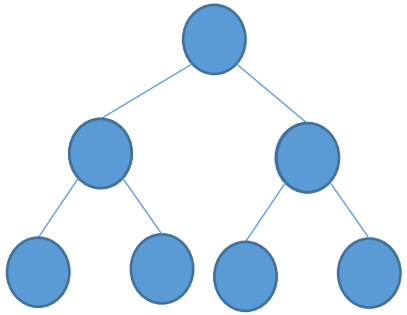
# Content

- 7.1. m-way Search Tree
  - Definition
  - Search
  - Insertion
- 7.2 B Tree
  - Definition
  - Search
  - Insertion
  - Deletion

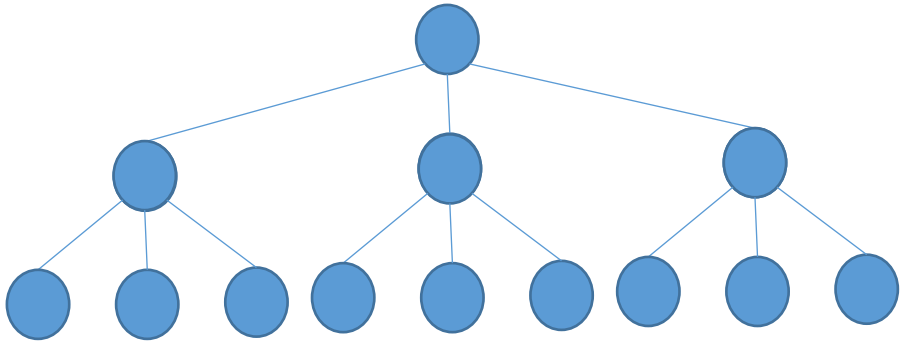
# 7.1 m-way Search Tree (m-원 탐색 트리)

# m-way search tree (m-원 탐색 트리)

- 각 Node에서의 분기가 최대 m개인 탐색 트리
  - 이원 탐색 트리의 2 보다 높은 분기율
  - **장점 :** 트리의 높이가 감소 (특정 Node의 탐색 시간 감소)
  - **단점 :** 삽입, 삭제 시 트리의 균형 유지를 위해 복잡한 연산이 필요



2원 트리



3원 트리



4

# m-way search tree (m-원 탐색 트리) (Cont'd)

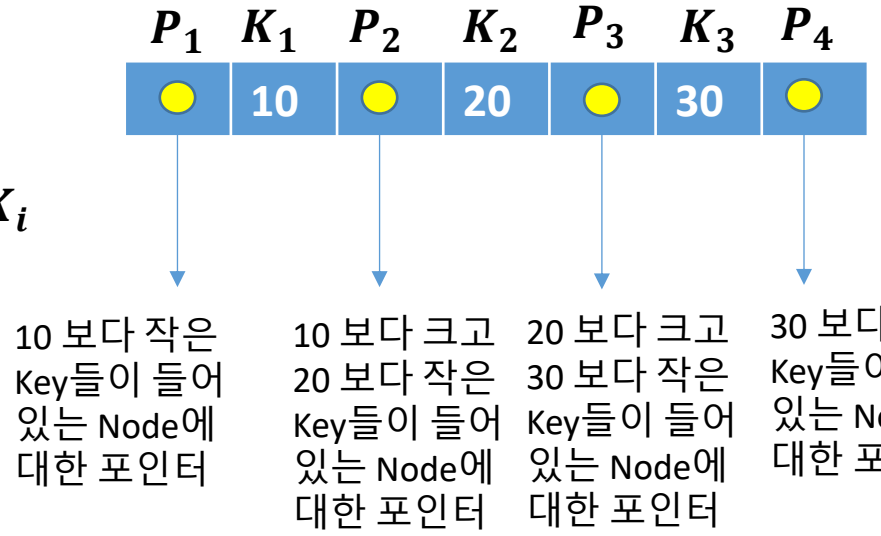
- **m-way search tree 의 성질**
  - **1. Node는 최소 1, 최대 m-1개의 Key를 가질 수 있다.**
    - **왜 m이 아닌, m-1인가?**
  - **2. Node 구조는**

    <n, <$P_1$, $K_1$, $P_2$, $K_2$, $P_3$, … , $P_n$, $K_n$, $P_{n+1}$> >
    - n : Key 개수 ($K_i$ 가 몇 개 있나), 1 ≤ n < m-1
      - = 하나의 Node는 최소 1, 최대 m-1 의 Key를 가질 수 있다.
    - $P_i$ : sub-tree에 대한 포인터 (**P**ointer),
    - $K_i$ : **K**ey 값

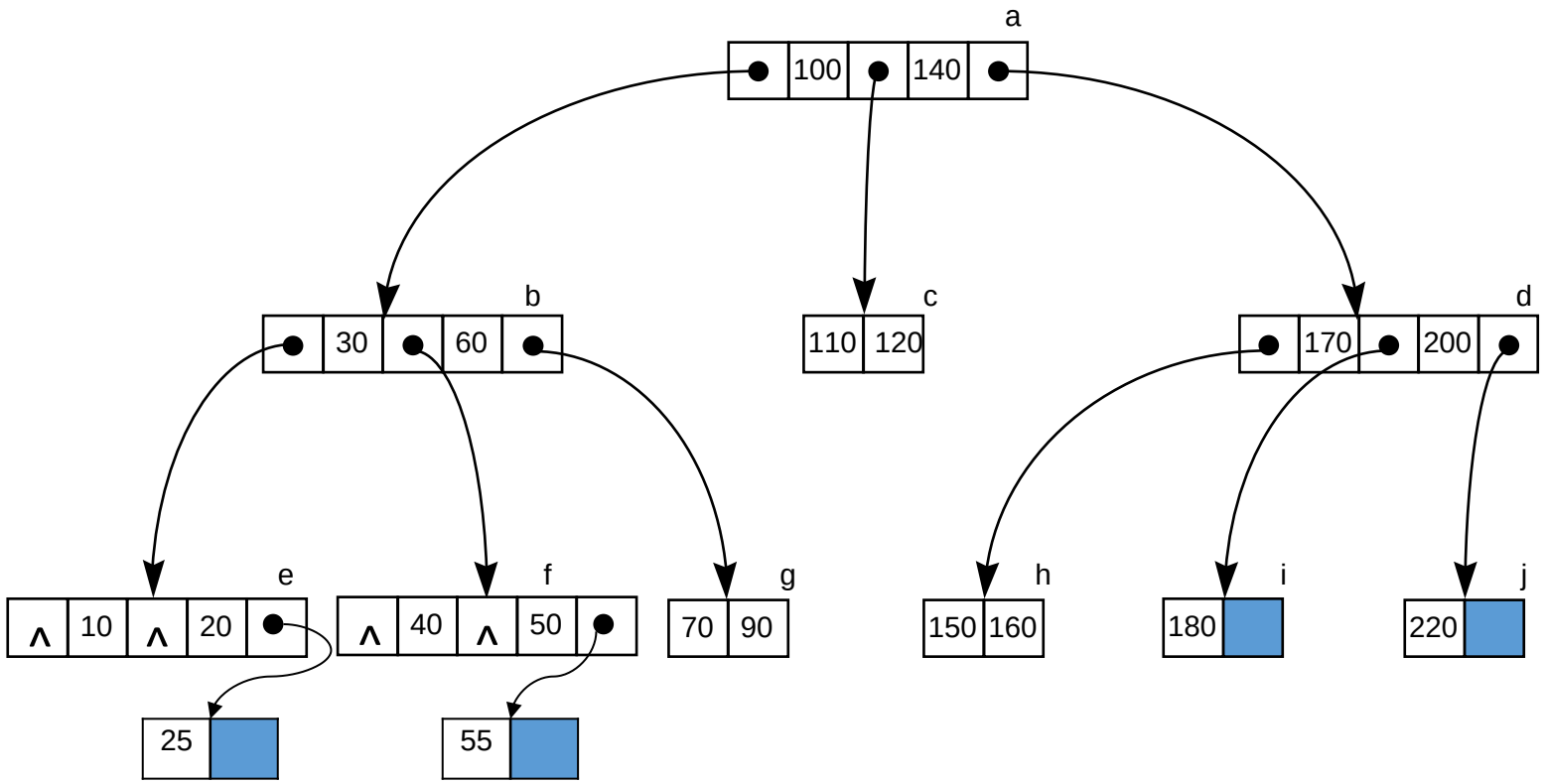
# m-way search tree (m-원 탐색 트리) (Cont'd)

- **m-way search tree 의 성질**
  - **3. 한 Node에 있는 Key 값들은 오름차순 으로 정렬됨**
  - <n, <$P_1$, $K_1$, $P_2$, $K_2$, $P_3$, … , $P_n$, $K_n$, $P_{n+1}$> >
    - $K_i < K_{i+1}$ , 1 ≤ i ≤ n-1
  - **4. "$P_i$ (0 ≤ i ≤ n)가 지시하는 sub-tree 모든 Node들의 Key 값" < $K_i$**
  - **5. "$P_{i+1}$ 이 지시하는 sub-tree 의 모든 Node들의 Key 값" > $K_i$**
  - **6. $P_i$ 가 지시하는 sub-tree 는 m-way search tree**

4원 탐색 트리 (최대 키 Key 3 (=4-1)을 가질 경우)



# 3-way search tree



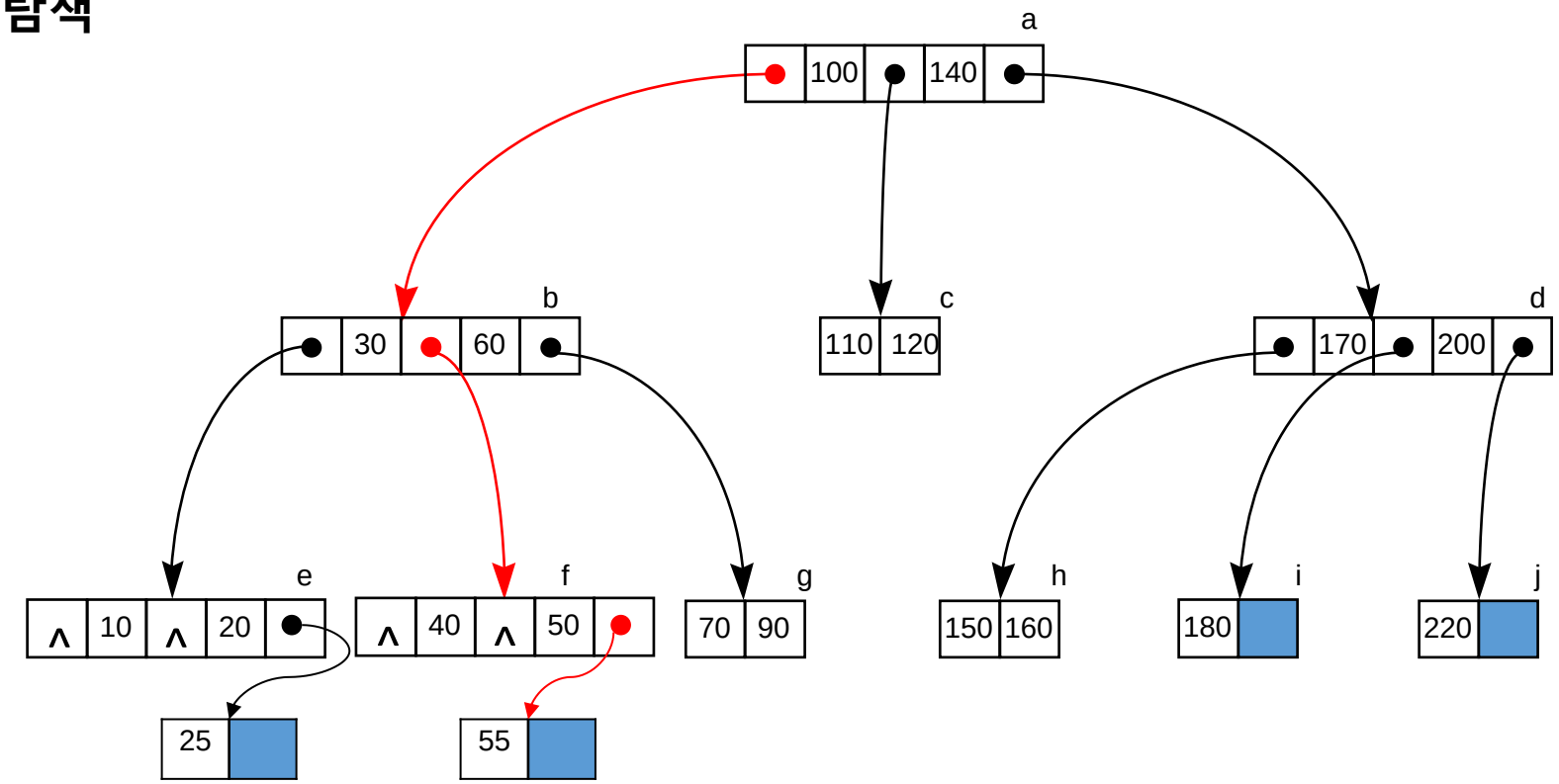

**키 값** $K_i$: ($K_i$, $A_i$)를 의미하고, $A_i$는 키 값 $K_i$ 를 포함하고 있는 Data Record의 주소

# m-way search tree 의 검색

- 탐색할 key를 K 라 하자.
- **Step 1.** Root node 부터 시작
- **Step 2.** Node의 정렬된 <$P_1$, $K_1$, $P_2$, $K_2$, $P_3$, … , $P_n$, $K_n$, $P_{n+1}$>의 $K_i$(키)를 왼쪽에서 부터(작은 쪽 부터) 비교해 나간다.
  - **Case1.** K = $K_i$ 이면:
    - Key를 찾았으므로 $K_i$ 를 반환하고 종료
  - **Case 2.** K < $K_i$ 가 되는 $K_i$ 가 있다면:
    - 바로 왼쪽의 Pointer $P_i$ 이 가리키는 Node로 이동하여 step 2를 수행.
      - $P_i$이 NULL 값이라면 Key 값이 존재 하지 않으므로 종료.
  - **Case 3.** 마지막 키 $K_n$ 보다 K가 크다면:
    - $K_n$ 바로 오른쪽의 $P_{n+1}$ 이 가리키는 Node로 이동하여 step 2를 수행.
      - $P_{n+1}$이 NULL 값이라면 Key 값이 존재 하지 않으므로 종료.

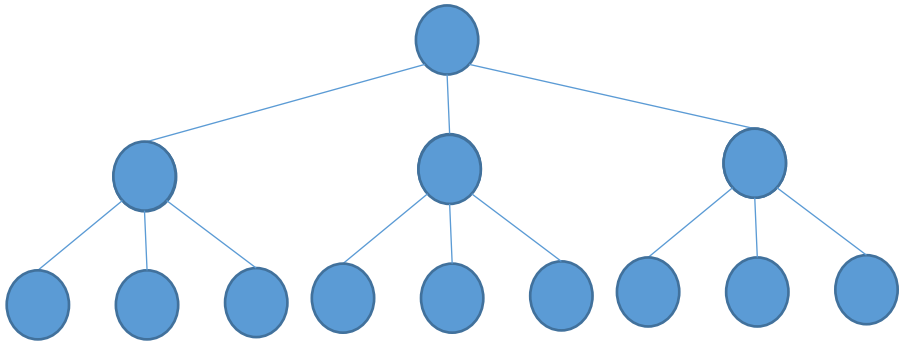
# m-way search tree의 검색 (m=3)

**예) Key 55 의 탐색**



# m-way search tree 분석

- **한 Node에 m-1개 Key 값을 저장하는 m-way search tree**
  - **높이 h 이면 $m^{h+1}-1$ 개의 키 값을 저장 가능**
    - 높이 h 이면 Tree에 최대 $\sum_{i=0}^{h} m^i$ 개 (= $\frac{m^{h+1}-1}{(m-1)}$ 개) 의 Node 저장.
    - 각 Node에 m-1 개의 키 값을 저장 가능하므로
      - 최대 $n = (m-1) * \sum_{i=0}^{h} m^i = (m-1) * \frac{m^{h+1}-1}{(m-1)} = m^{h+1}-1$ 개의 키 값을 저장 가능
      - (예) 3-원 탐색 트리: 높이가 2이면 n=2*(1+3+9)=26개의 키 값을 저장
      - (교과서에서는 $m^h-1$ 이라 하나 이는 잘못됨)



3원 트리

# m-way search tree 분석 (Cont'd)

**m-way search tree 탐색시간**: 탐색 경로 길이(높이)에 비례
- Node 수 N이 같을 경우 분기 수(m)가 커지면 트리의 높이가 낮아짐

- N개의 Node를 가진 m-원 탐색 트리
  - 최소 높이 h = $\lceil log_m((m-1)\cdot N+1)-1 \rceil$
- n 개의 키를 가진 m-원 탐색 트리
  - 최소 높이
    - $n=(m-1)\cdot N$ 이므로 $N=\frac{n}{(m-1)}$
    - $h=\lceil log_m((m-1)\cdot N+1)-1 \rceil$
    - $=\lceil log_m(n+1)-1 \rceil$
  - 평균 검색 시간
    - $=O(\lceil log_m(n+1)-1 \rceil)$
    - $=O(log_m n)$

**최소 높이 계산 과정 (참고)**

N = $\frac{m^{h+1}-1}{(m-1)}$ :높이 h의 m원 탐색 트리의 최대 Node 수.
이는 높이 h의 탐색 트리의 리프 제외한 모든 Node가 m개의 자식 Node를 가지고 있다는 뜻이다. 따라서 위 식에서의 높이 h가 N개의 Node를 가지는 트리 최소 높이가 된다.

$$N=\frac{m^{h+1}-1}{(m-1)}$$

<=> $(m-1)\cdot N+1=m^{h+1}$

<=> $h+1=log_m((m-1)\cdot N+1)$

<=> $h=log_m((m-1)\cdot N+1)-1$

=> $h=\lceil log_m((m-1)\cdot N+1)-1 \rceil$

# 7.2 B Tree

- Definition &  Search (정의 & 검색)

# B-Tree

- **1972년 Bayer & McCreight가 제안**

- **Balanced m-way Search Tree (균형 m-원 탐색 트리)**
  - 가장 많이 사용되는 인덱스 방법
  - 효율적인 균형 알고리즘을 제공

# B-Tree

- **차수(order)가 m인 B-tree의 특성**
  - 1. B-Tree는 공백이거나 높이가 0 이상인 m-way search tree
    - 교과서에서는 1 이라 되어 있으나, Tree 높이의 정의에 따르면 root만 있는 Tree는 높이가 0이라 0이 맞다.
  - 2. Root와 Leaf(단말)를 제외한 내부 Node (=중간 Node)
    - 최소⌈ m/2 ⌉, 최대 m개의 Sub-Tree
    - 적어도⌈ m/2 ⌉- 1개의 key 값이 존재 (Node의 반 이상이 채워짐)

# B-Tree (Cont'd)

- **차수(order)가 m인 B-tree의 특성 (Cont'd)**
  - 3. Root
    - Root는 자신이 Leaf가 아닌 이상 적어도 두 개의 Sub-Tree를 가짐
      - Root는 만약 자식이 있다면 적어도 두 개 자식이 있다.
  - 4. 모든 Leaf는 같은 레벨

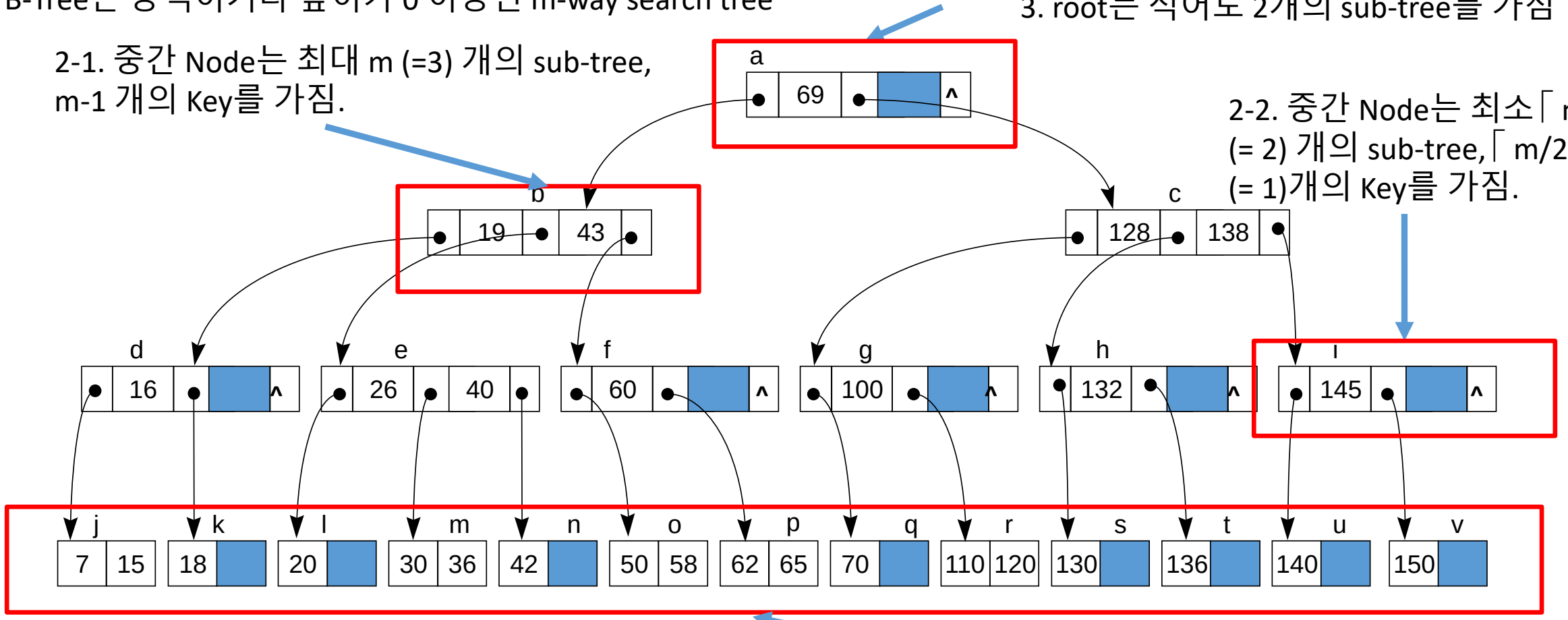
# 3 Order B-Tree 예

1. B-Tree는 공백이거나 높이가 0 이상인 m-way search tree

2-1. 중간 Node는 최대 m (=3) 개의 sub-tree, m-1 개의 Key를 가짐.

3. root는 적어도 2개의 sub-tree를 가짐

2-2. 중간 Node는 최소⌈ m/2 ⌉ (= 2) 개의 sub-tree,⌈ m/2 ⌉- 1 (= 1)개의 Key를 가짐.




4. 모든 Leaf는 같은 레벨

# m차 B-트리 Node 구조

- **Node 구조**

  < n, <$P_1, K_1, P_2, K_2, P_3, K_3, \dots, P_n, K_n, P_{n+1}$ >>

  - n : 키 값의 개수( 1 ≤ n < m-1),
  - $P_1, \dots, P_{n+1}$ : sub-tree에 대한 포인터
  - 각 키 값 $K_i$ 는 그 키 값을 가진 레코드에 대한 포인터 $A_i$ 를 포함

1. 각 Node의 Key 값들은 항상 오름차순(1 ≤ i ≤ n → $K_i < K_{i+1}$)을 유지
2. $P_i$ 가 지시하는 sub-tree의 Key 값들은 모두 $K_i$ 보다 작다.
3. $P_{n+1}$ 이 지시하는 sub-tree의 Key 값들은 모두 $K_n$ 보다 크다.
4. $P_i$ (0 ≤ I ≤ n+1)가 지시하는 sub-tree들은 모두 m-way search tree이다.

# m차 B-Tree Node 구조 (Cont'd)

- B-Tree의 장점
  - Key 삽입, 삭제 뒤에도 트리의 균형 상태를 유지
  - 저장장치의 효율적인 사용
    - 각 Node의 Key를 위해 확보한 공간의 반 이상은 항상 Key 값으로 채워짐

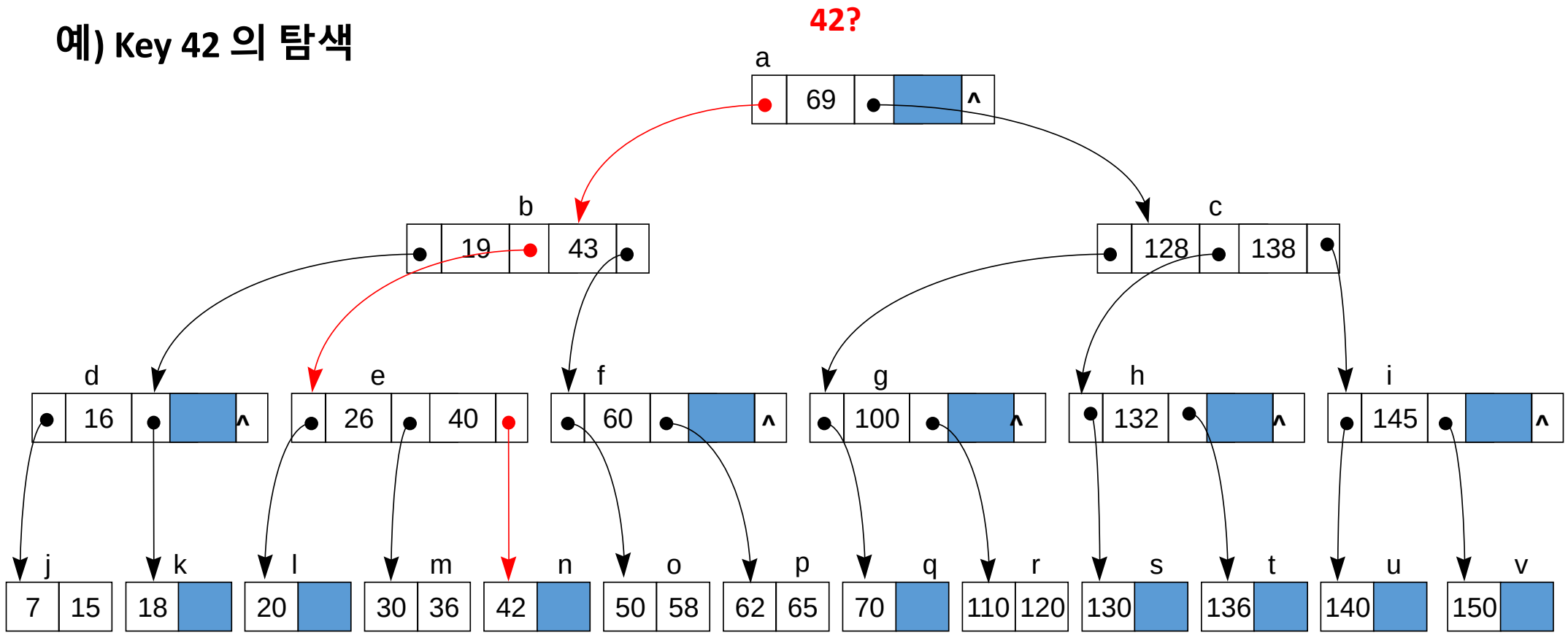
# B-Tree에서의 검색

- **검색 :** m-way search tree의 검색과 같은 과정
  - **직접 탐색 :** 검색 키 값과 비교하는 키의 크기에 따라 왼쪽 또는 오른쪽 sub-tree로 분기
    - 검색 키 < 비교하는 키 => 왼쪽 sub-tree
    - 검색 키 > 비교하는 키 => 오른쪽 sub-tree
  - **동일 Node 안에서의 키 검색은 순차 검색**

    예) 키 값 42 검색

# B-트리에서의 검색 예.

예) Key 42 의 탐색



# 7.2 B Tree

- Insertion (삽입)

# B-Tree에서의 Insertion

- 삽입 : **새로운 Key 값은 항상 Leaf Node에** 삽입
  - **Node에 공간이 있는 경우 :** 단순히 순서에만 맞게 삽입

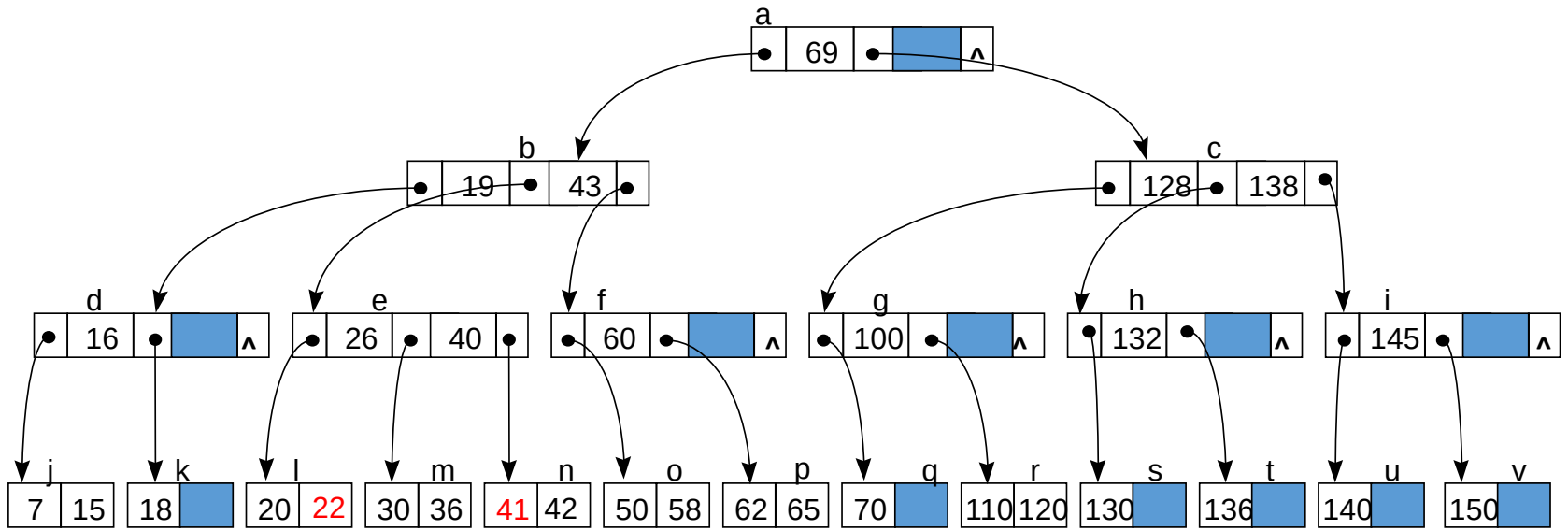
# B-Tree에서의 Insertion (Cont'd)

- **Node에 공간이 없는 경우 : overflow로 split** 발생
  - 해당 Node를 **두 개의 Node로 분할**
    - 해당 Node에 새로운 Key 값을 삽입했다고 가정
    - 중간(⌈**m/2**⌉번째) Key 값을 기준으로 왼쪽 작은 키 값들은 그대로 두고, 오른쪽 큰 키 값들은 새로운 Node에 저장
    - 중간 Key 값은 분할된 두 Node가 왼쪽 sub-tree, 오른쪽 sub-tree가 되도록 부모 Node에 삽입
    - 이 때, 다시 overflow가 발생하면 위와 같은 분할(split) 작업을 반복

# B-Tree에서의 삽입 예

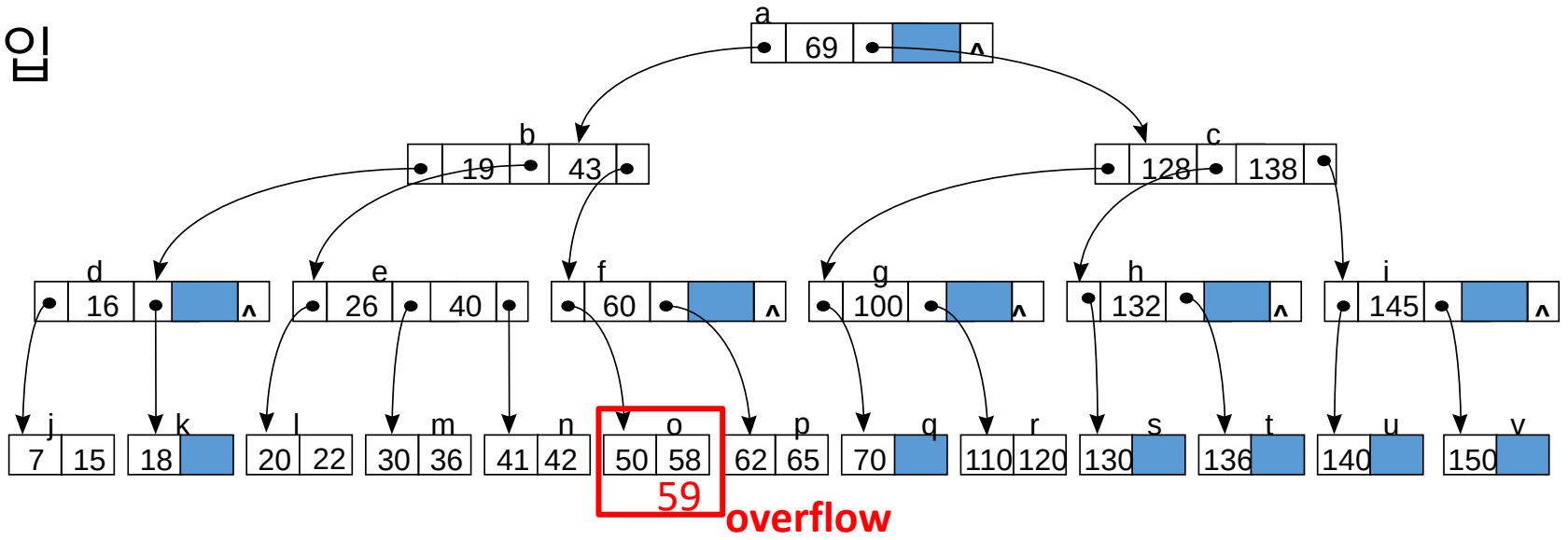
- 앞의 예의 3원 B-Tree에 새로운 키 값 22, 41, 59, 57, 54, 44, 75, 124, 122, 123 삽입

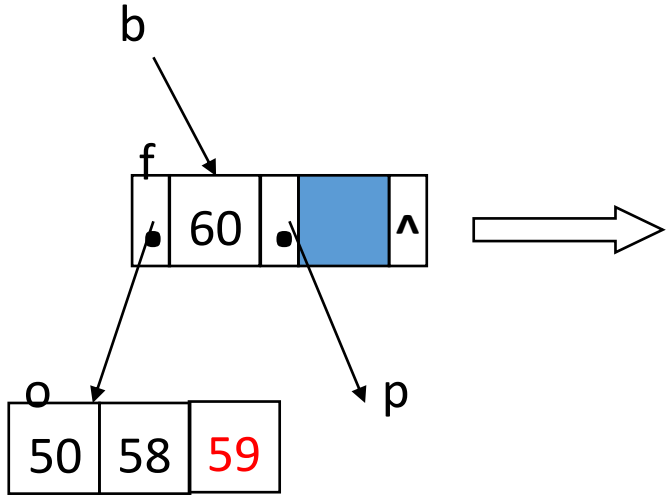

**(a) Node *l*에 키 22의 삽입** **(b) 리프 Node *n*에 키 42의 삽입**



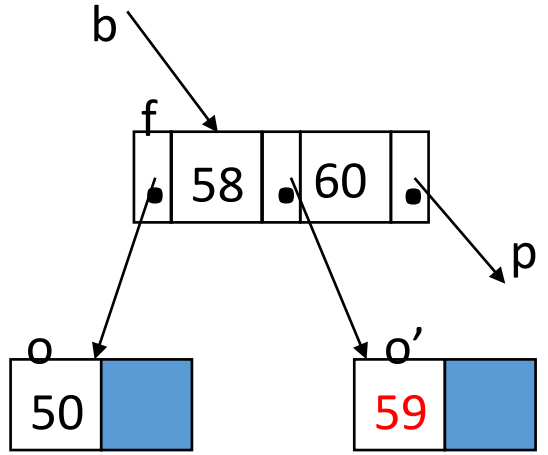
# B-Tree에서의 삽입 예 (Cont'd) : Node *o*에 key 59의 삽입

- 59 삽입

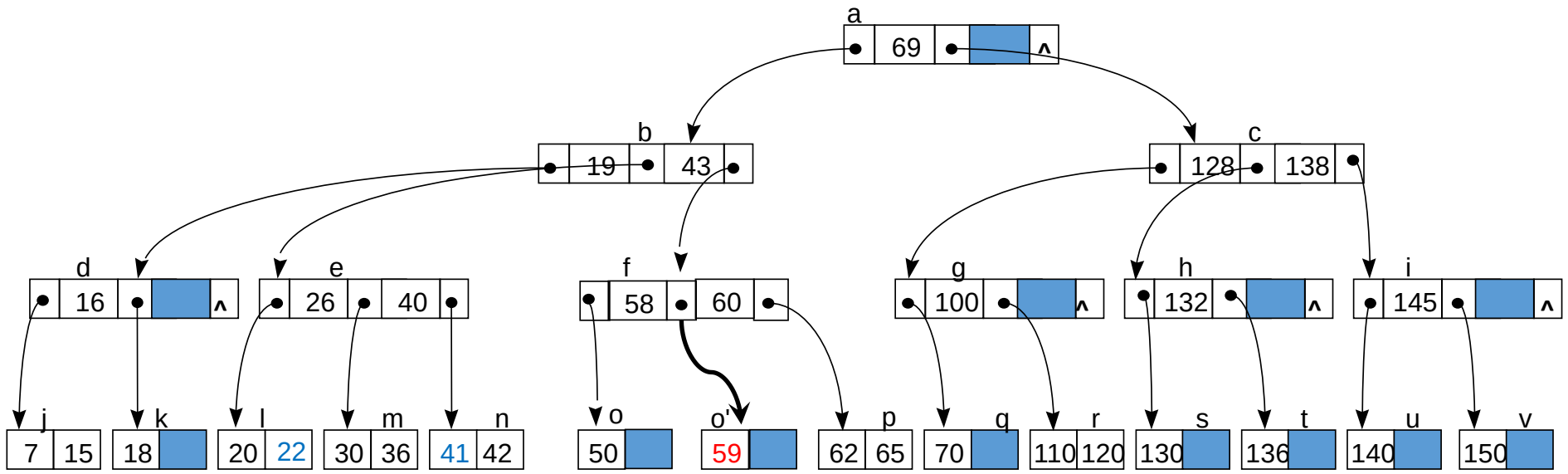


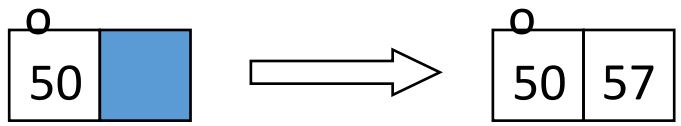

**1.** 59 가 Node o에 들어가서 Node의 key가 넘침. (3 > 2 (m-1))
**2.** 중간 key 58을 기준으로 작은 Key(50)들은 왼쪽 sub-tree, 큰 Key (59)들은 오른쪽 sub-tree가 되도록 함.
**3.** key 58이 부모키가 되고 (58은 60 보다 작으므로 60 앞에 위치) 58의 왼쪽/오른쪽 sub-tree를 설정.



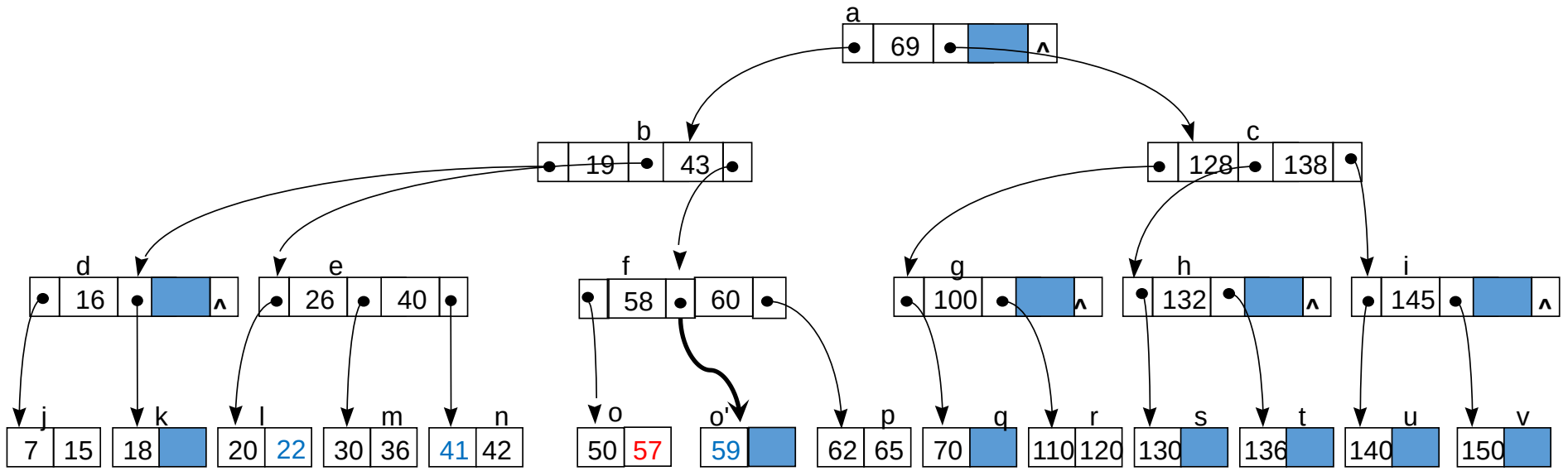
# B-Tree에서의 삽입 예 (Cont'd) : Node *o*에 Key 59의 삽입



# B-Tree에서의 삽입 예 (Cont'd)

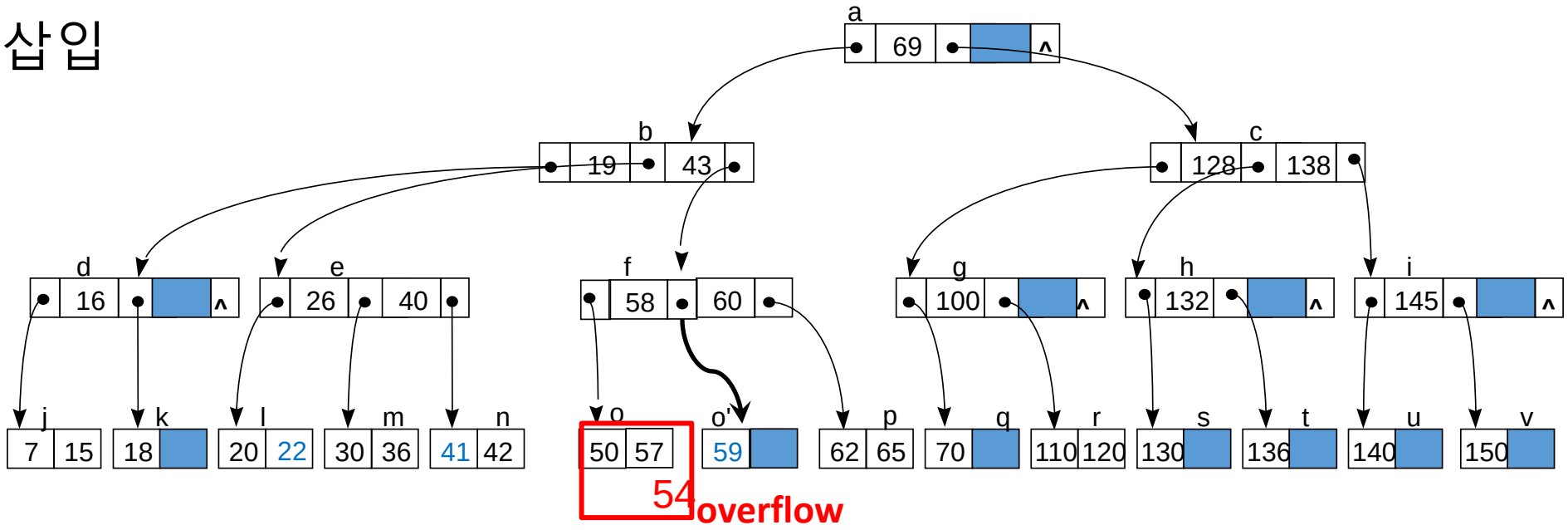


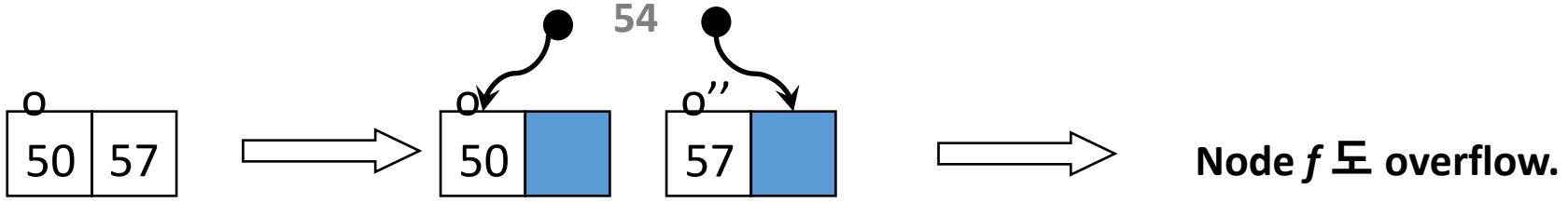

**(d) Node *o*에 key 57 삽입**



# B-Tree에서의 삽입 예 (Cont'd)

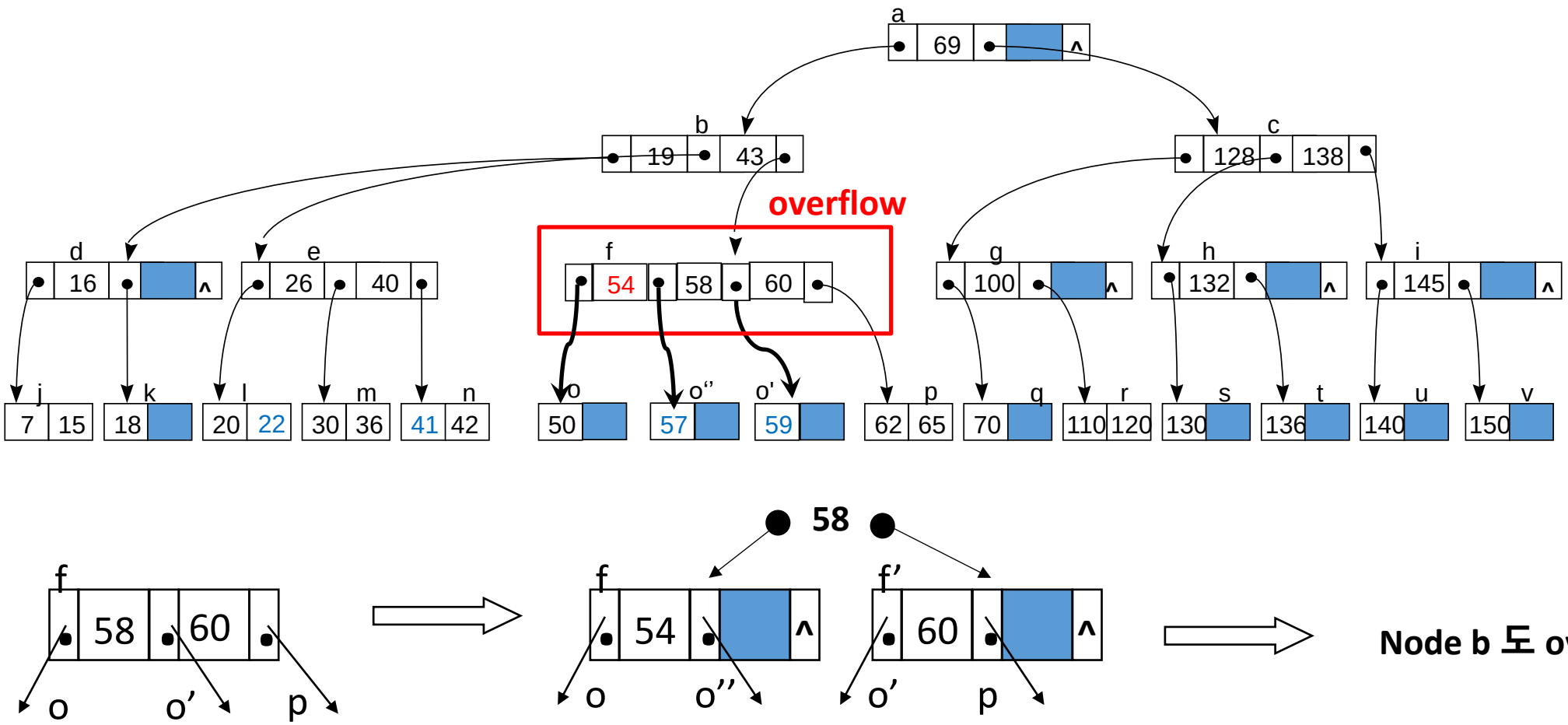
- 54 삽입



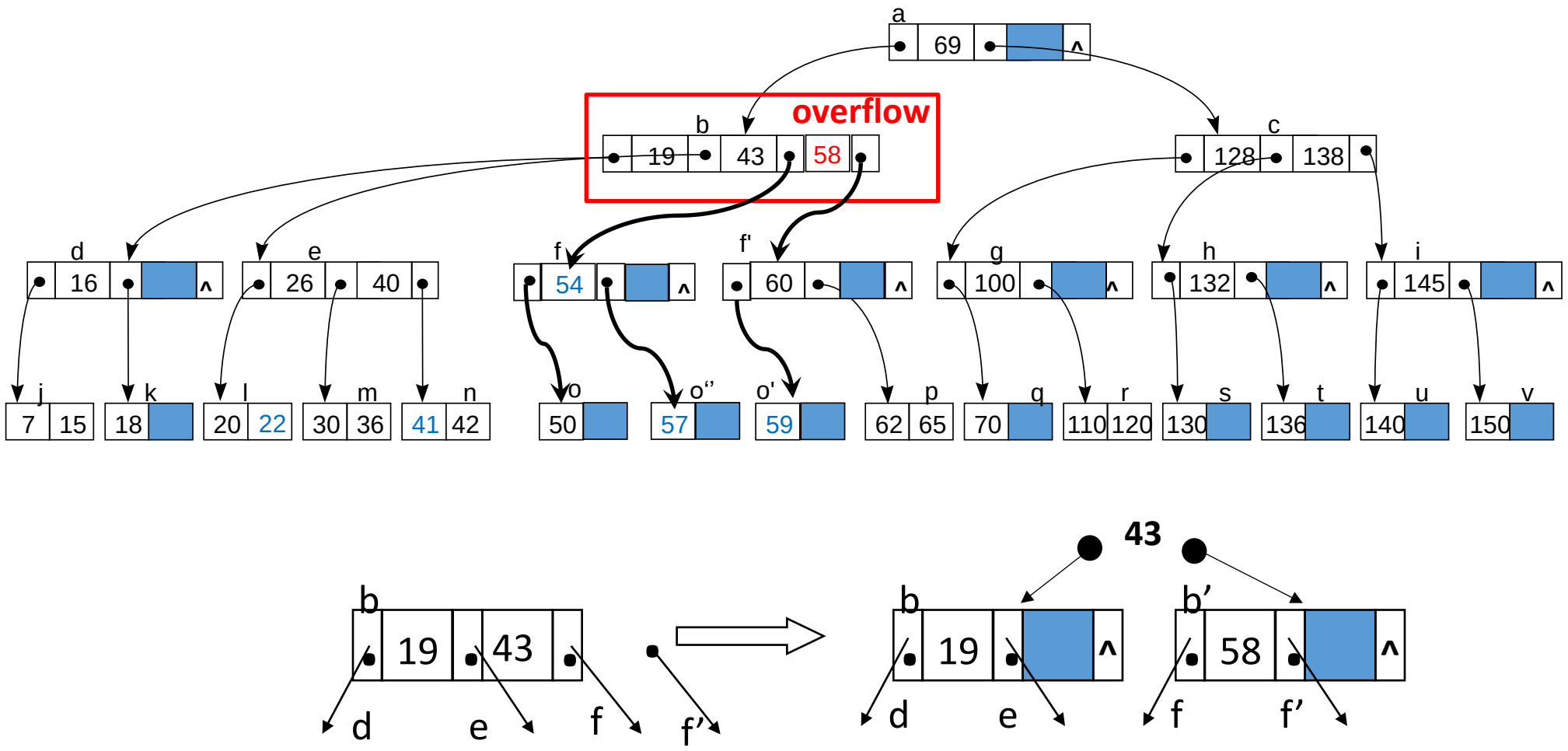

**(d) key 54의 삽입으로 Node *o*의 분할**
**(54는 부모 Node *f*로 이동)**

# B-Tree에서의 삽입 예 (Cont'd)



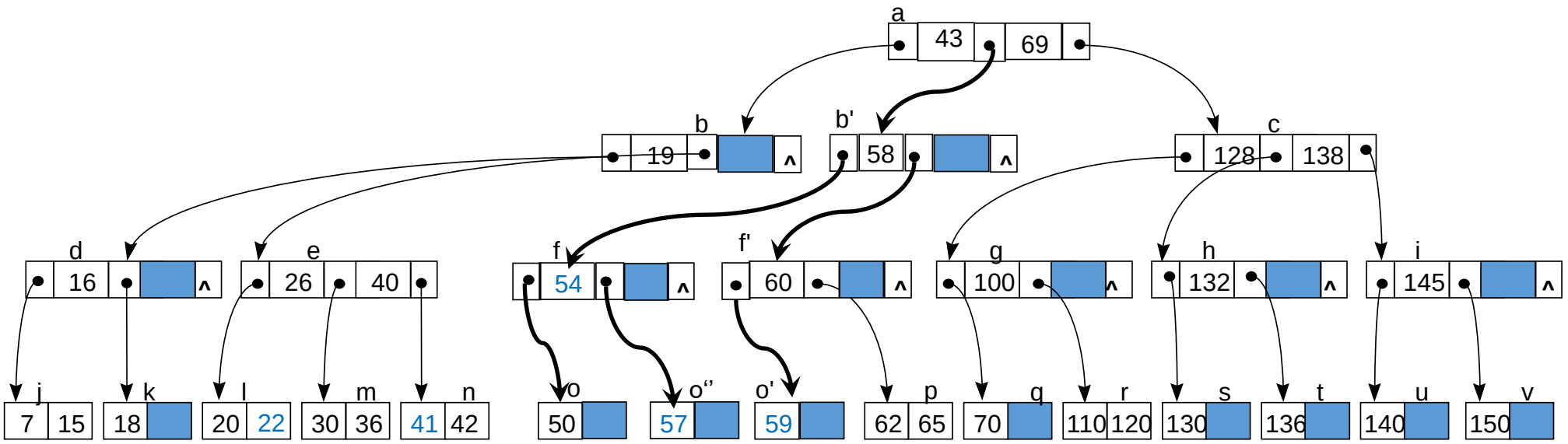

**(e) Node *f* 분할 :** *f* 에 key 54의 삽입, 58는 부모 Node *b*에 삽입, 60은 새 Node f'로

# B-Tree에서의 삽입 예 (Cont'd)




**(g) Node *b*에 키 58의 삽입 (43은 부모 Node *a*에 삽입)**

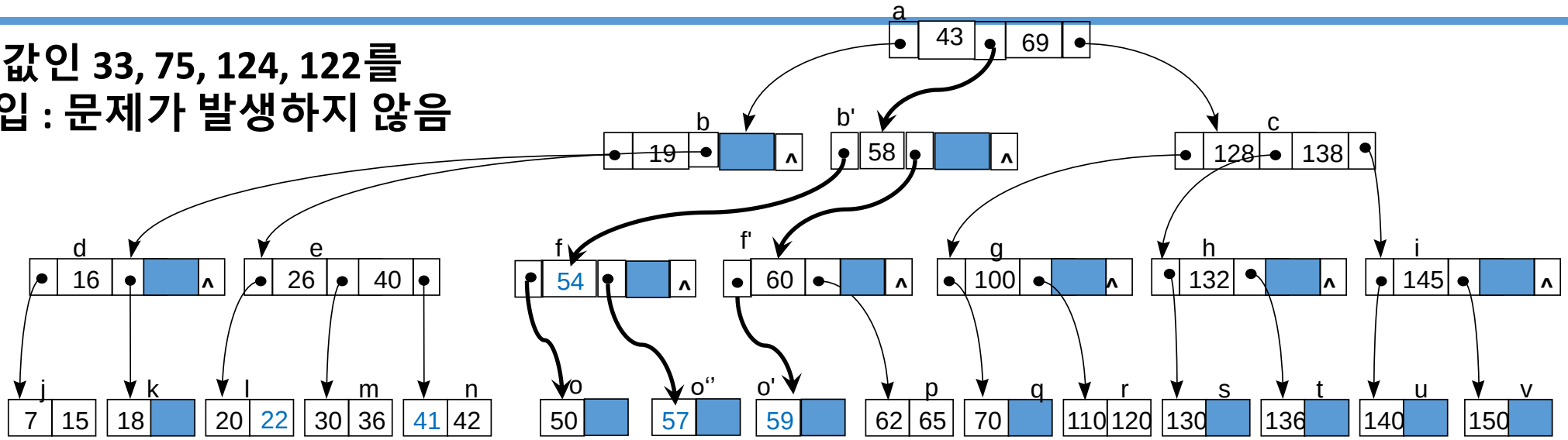
# B-Tree에서의 삽입 예 (Cont'd)



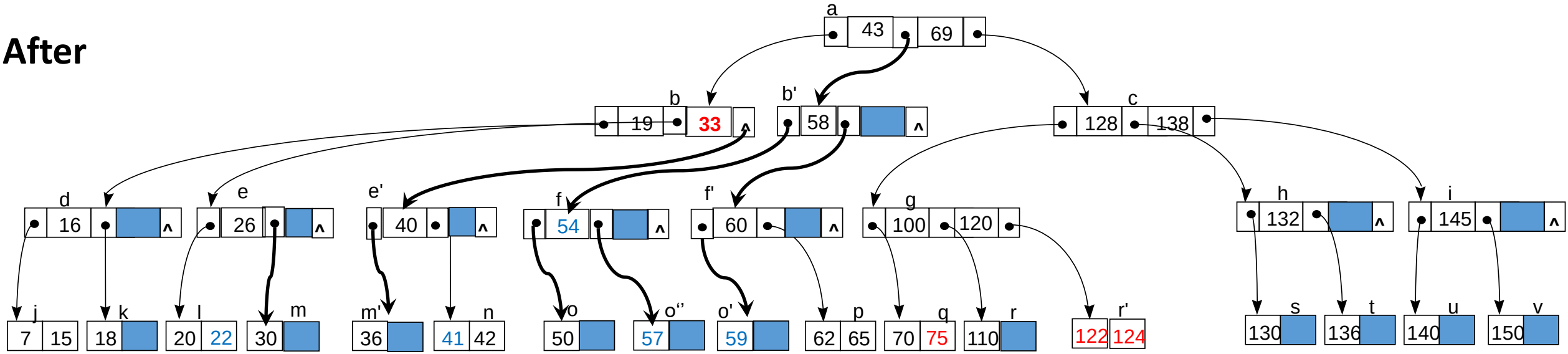
# B-Tree에서의 삽입 예 (Cont'd)

**나머지 키 값인 33, 75, 124, 122를 차례로 삽입 : 문제가 발생하지 않음**

**Before**

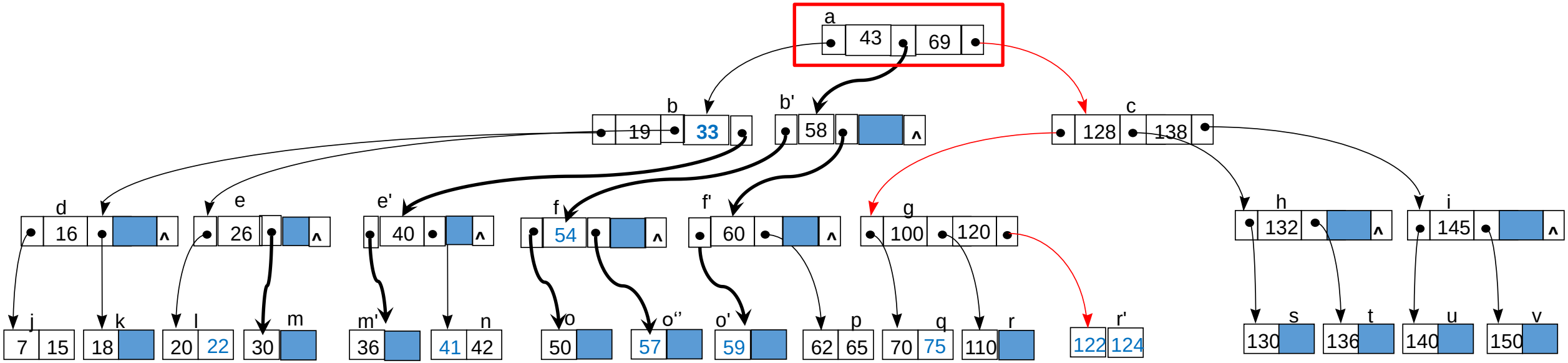



**After**



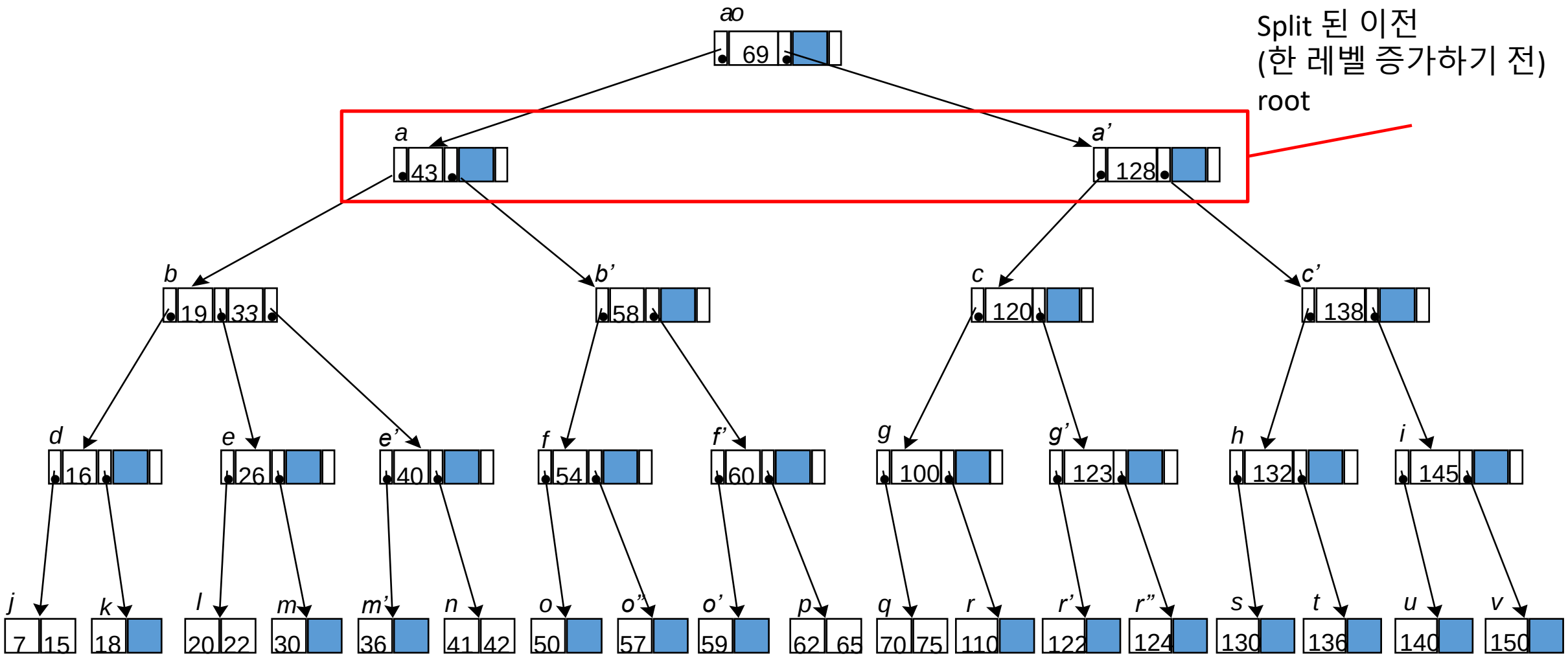
# B-Tree에서의 삽입 예 (Cont'd)

**마지막 키 값인 123을 삽입 : B-트리는 한 레벨 증가됨**



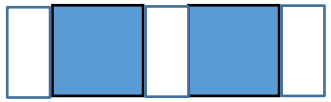

123 삽입으로 인해 r' 가 overflow되어 split하려 해도, Node r'에서 root 까지 경로의 모든 Node의 키가 가득 차 있음. => 부모 Node를 split 해 가며 root까지 도달 => **root도 split** => **B-트리는 한 레벨 증가.**

# B-Tree에서의 삽입 예 (Cont'd)

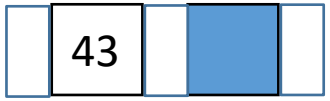


# 3원 B-Tree 생성 과정

- key 값 43, 69, 138, 19 순으로 삽입하여 생성



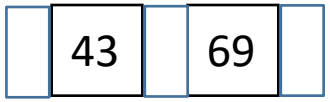
(a) 크기가 2인 공백 root Node



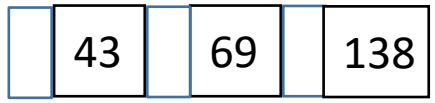

(b) 키 값 43의 삽입(Node 1개의 3차 B-트리)

# 3원 B-Tree 생성 과정 (Cont'd)

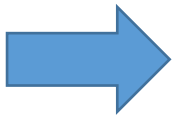
- key 값 43, 69, 138, 19 순으로 삽입하여 생성

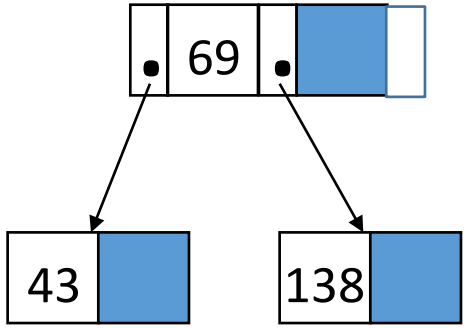



**(c) key 값 69의 삽입(Node 1개의 3차 B-트리)**

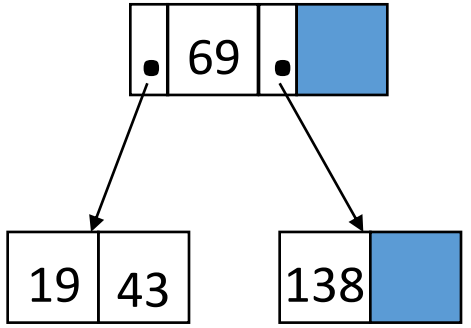



138 이 들어가 overflow



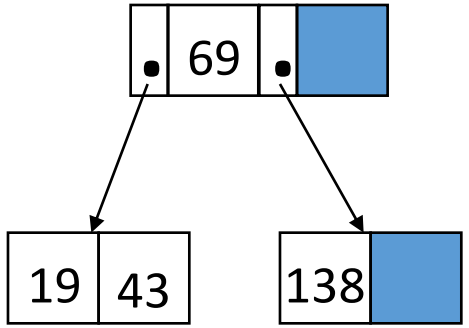

**(d) key 값 138의 삽입(Node 3개의 3차 B-트리)**




**(e) key 값 19의 삽입(Node 3개의 3차 B-트리)**

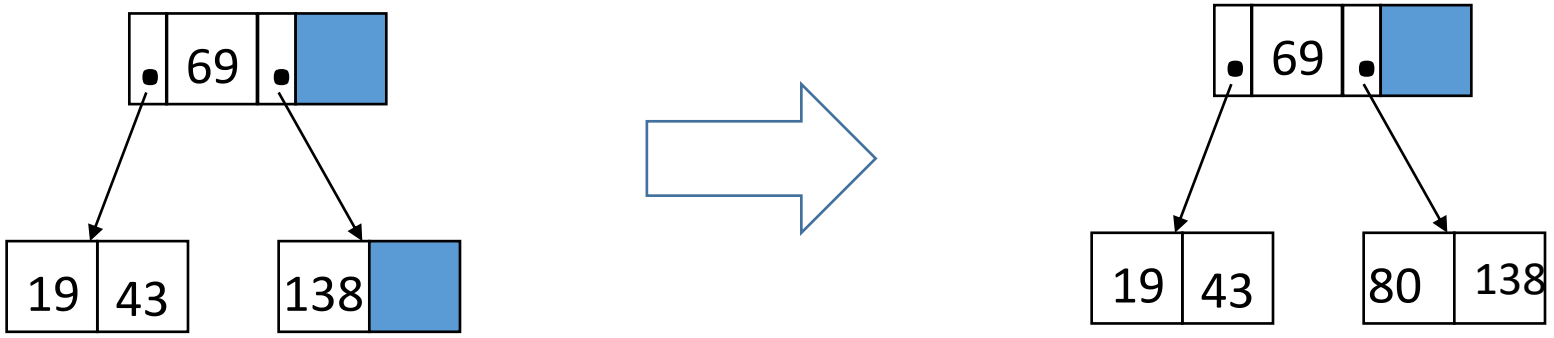
# B-Tree Key 삽입 연산의 특성

- Key 삽입 시 Node 분할이 일어나지 않는 한 Key는 Leaf Node에 저장된다.
- root Node가 분할되면 Tree 높이가 위로 하나 증가한다. (새 root 추가)
- => 모든 Leaf Node가 같은 레벨에 있다.

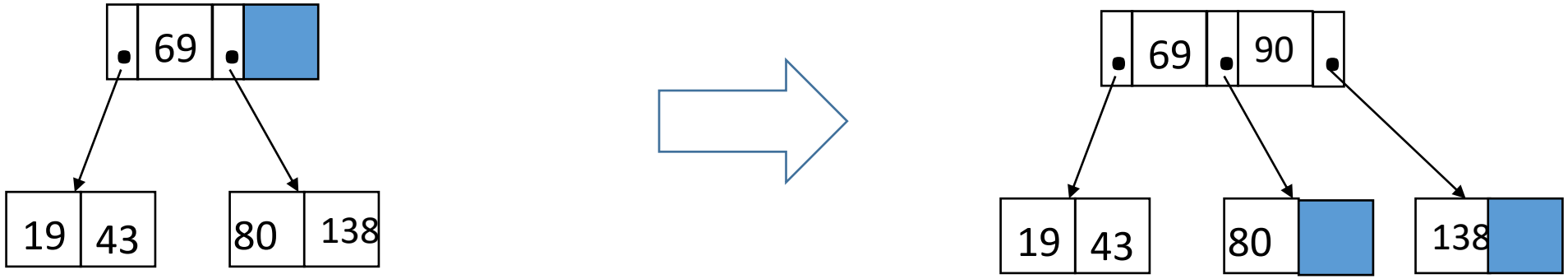


# 3원 B-Tree 생성 과정 (Cont'd)

- key 값 80 삽입

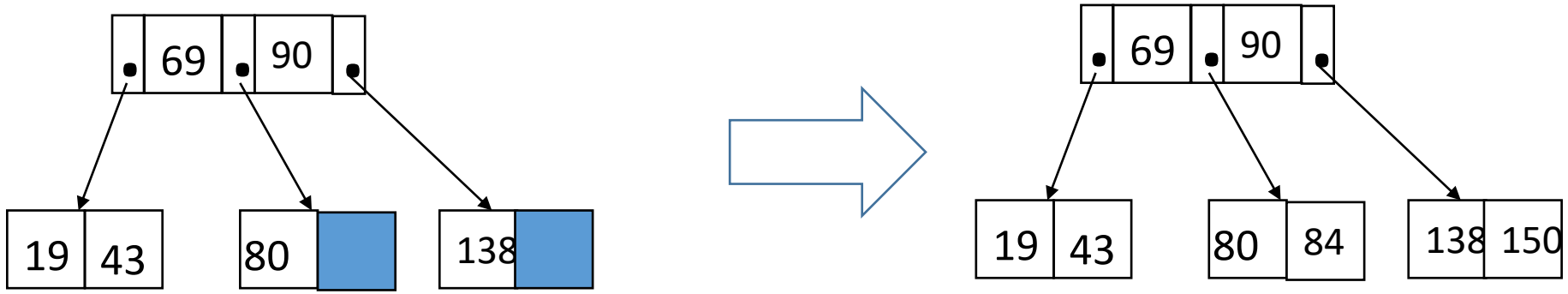



- key 값 90 삽입



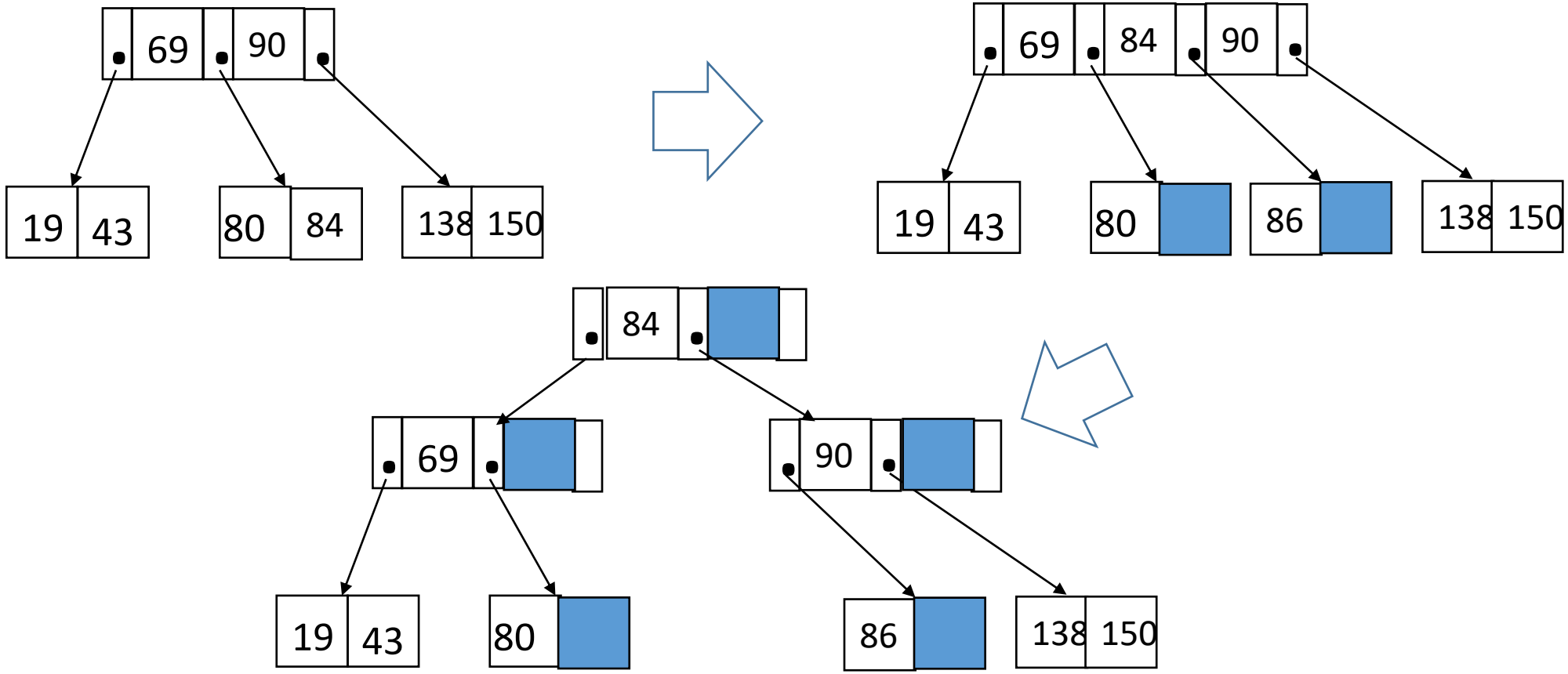
# 3원 B-Tree 생성 과정 (Cont'd)

- key 값 84, 150 삽입



# 3원 B-Tree 생성 과정 (Cont'd)

- key 값 86 삽입



# 7.2 B Tree

- Deletion (삭제)

# B-Tree에서의 삭제

- **삭제 알고리즘**
  - 0. 검색을 이용해 삭제할 Key 값 $K_{rm}$ 을 찾는다.
  - 1. $K_{rm}$ 이 Leaf Node에 저장되어 있는 경우
    - 그대로 삭제

# B-Tree에서의 삭제

- 2. $K_{rm}$ 이 내부 Node에 저장되어 있는 경우
  - => $K_{rm}$ 값의 후행Key를 찾아 ($K_{rm}$ 보다 큰 키 중 가장 작은 key) $K_{rm}$ 과 교환 후 Leaf Node에서 삭제
    - B-트리 특성상 이 **후행 key 값은 항상 Leaf Node에 있음**
    - Leaf Node에서의 삭제 연산이 더 간단
  - => 후행 key 값 대신 선행 key 값 ($K_{rm}$ 보다 작은 key 중 제일 큰 key 값) 을 사용할 수 있음

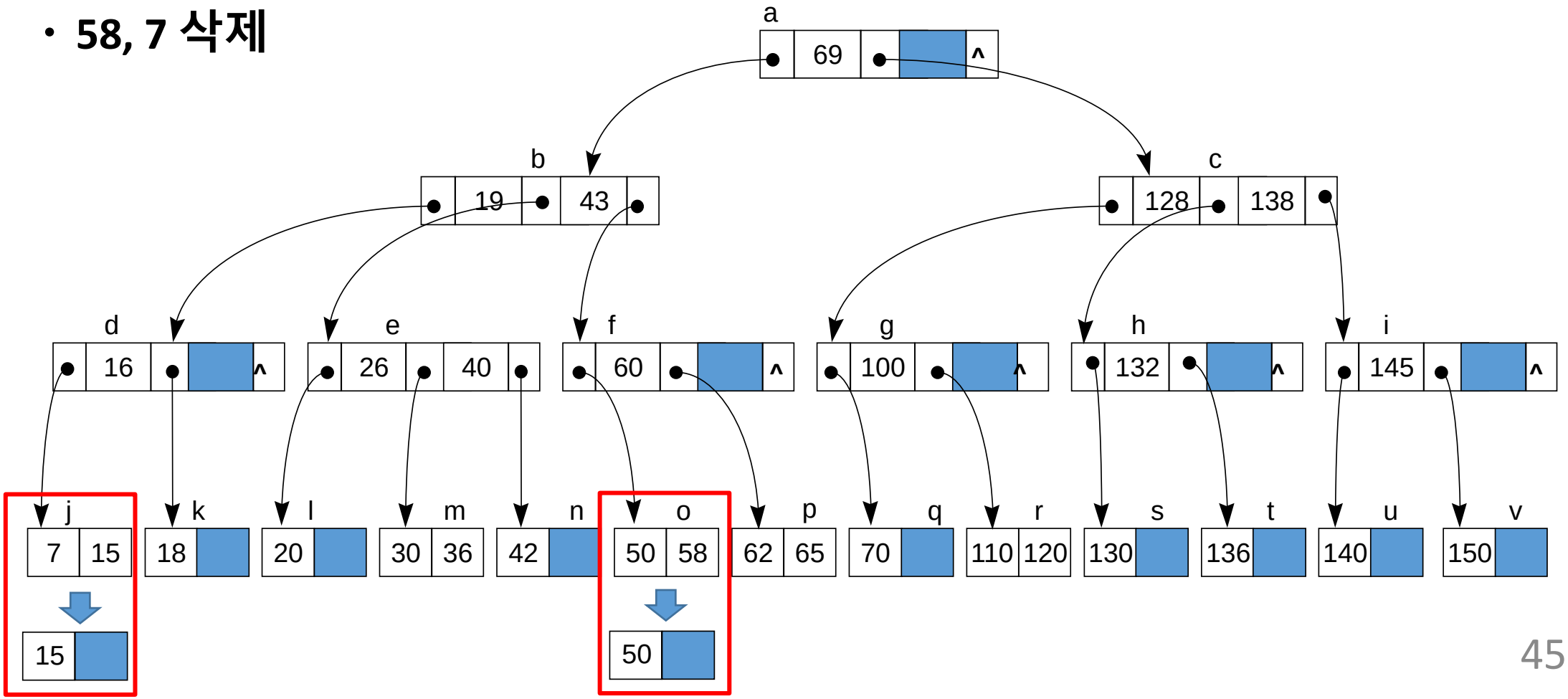
# B-Tree에서의 삭제

- 3.삭제 결과로 Node의 key 값 수가 B-트리의 최소 key 값 수 ($\lceil \frac{m}{2} \rceil$- 1) 보다 작게 되면**underflow**가 일어남
  - 재분배나 합병을 수행

# B-Tree에서의 삭제 예 : $K_{rm}$ 값이 Leaf Node에 저장되어 있는 경우

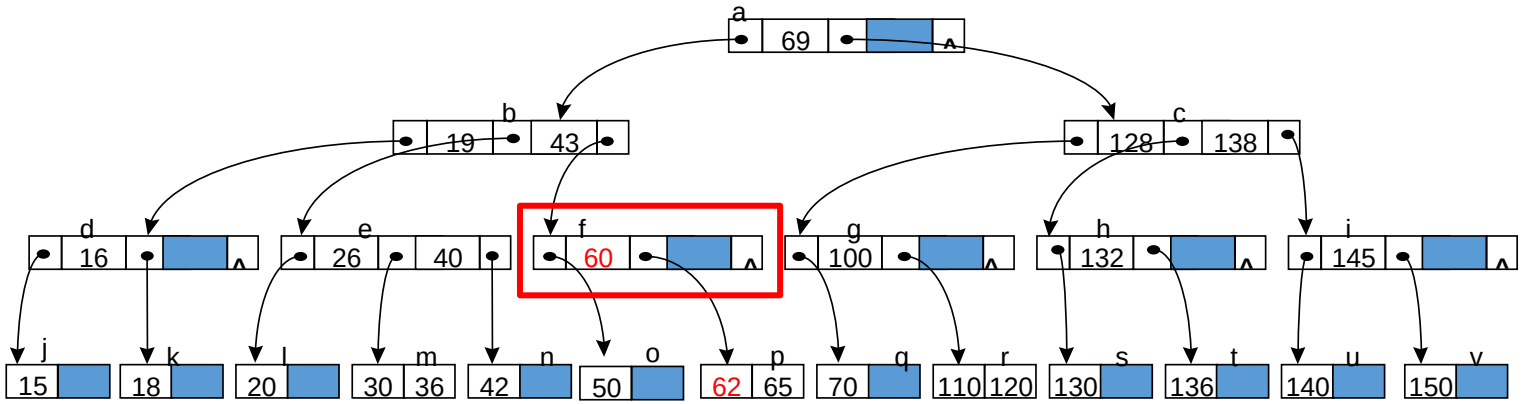
- 1. $K_{rm}$ 값이 Leaf Node에 저장되어 있는 경우 => 그대로 삭제
- (자식에의 Pointer도 없으므로 key 삭제만 필요)

- **58, 7 삭제**

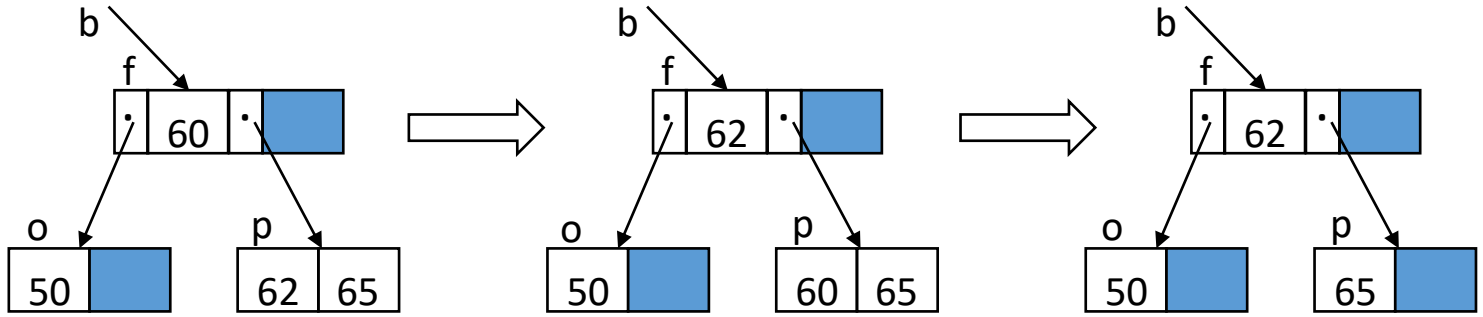


# B-Tree에서의 삭제 예 : $K_{rm}$ 값이 내부 Node에 저장되어 있는 경우

- 2. $K_{rm}$ 값이 내부 Node에 저장되어있는 경우
  - =>$K_{rm}$ 값의 후행 key 값 ($K_{rm}$보다 큰 자식 중 제일 작은 key 값)과 교환 후 Leaf Node에서 삭제
  - => 후행 key 값 대신 선행 key 값 ($K_{rm}$ 보다 작은 자식 중 제일 큰 key 값)을 사용할 수 있음

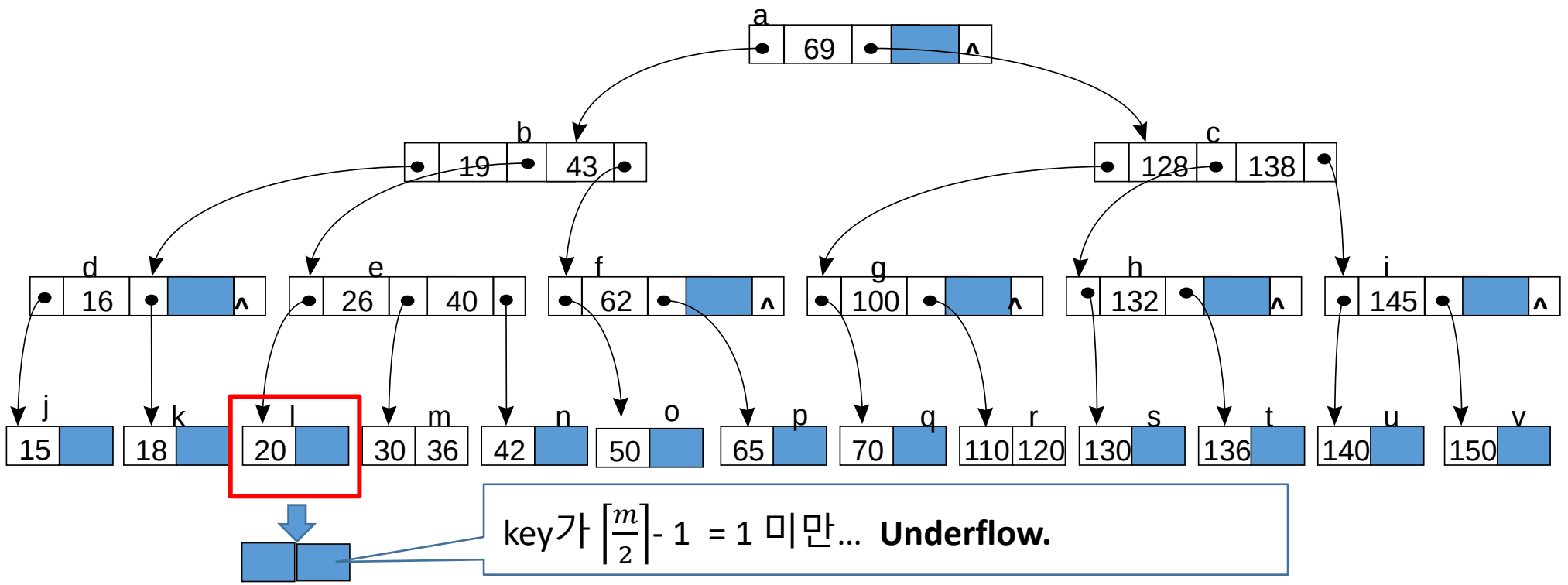



**Node $f$에서 key 값 60의 삭제**



# B-Tree에서의 삭제 예 (Cont'd)

- 3.삭제 결과로 Node의 key 값 수가 B-트리의 최소 key 값 수 ($\lceil \frac{m}{2} \rceil$ - 1) 보다 작게 되면 **underflow** 가 일어남 => **재분배**나 **합병**을 수행




**Node /에서 key 값 20의 삭제**

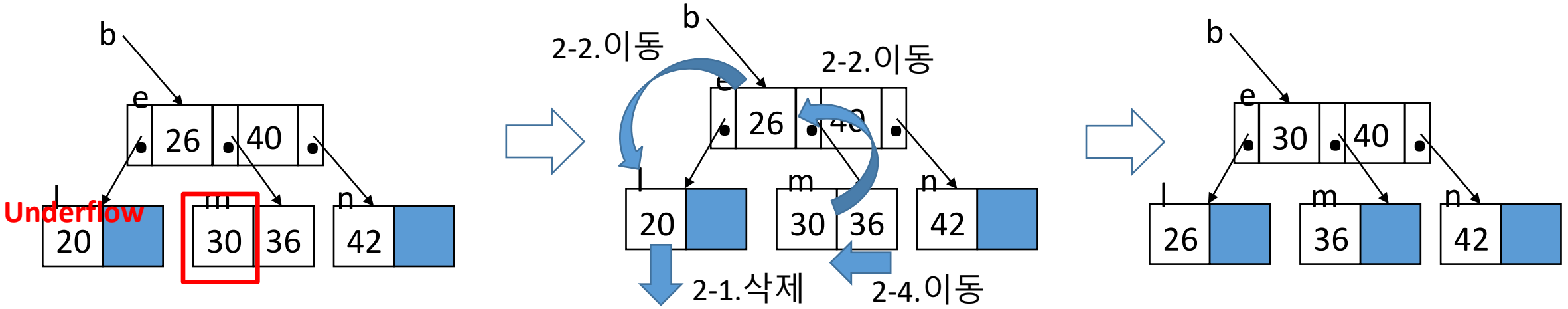
# B-Tree 에서의 삭제 : 재분배

- **재분배(redistribution)**
  - Step 1. Underflow가 일어난 Node의 오른쪽이나 왼쪽 형제 Node 중에서 최소 key 수 보다 많은 수의 key를 가진 Node에서 key 하나를 차출
  - Step 2. 부모 Node에 있는 분리key를 Under가 일어난 Node로 이동하고, 이 빈 자리로 차출된 key를 이동
    - 트리 구조가 변경되지 않음

# B-Tree에서의 삭제 예 : 재분배

## 재분배(redistribution)

## Node /에서 key 값 20의 삭제




1. 해당 Node의 오른쪽이나 왼쪽 **형제 Node** 중에서 최소 key 수보다 많은 수의 key를 가진 Node에서 key를 하나 차출.
(오른쪽 형제 Node에서 찾을 경우 Node 안의 key 중 가장 작은 key.
왼쪽 형제 Node면 그 반대.)

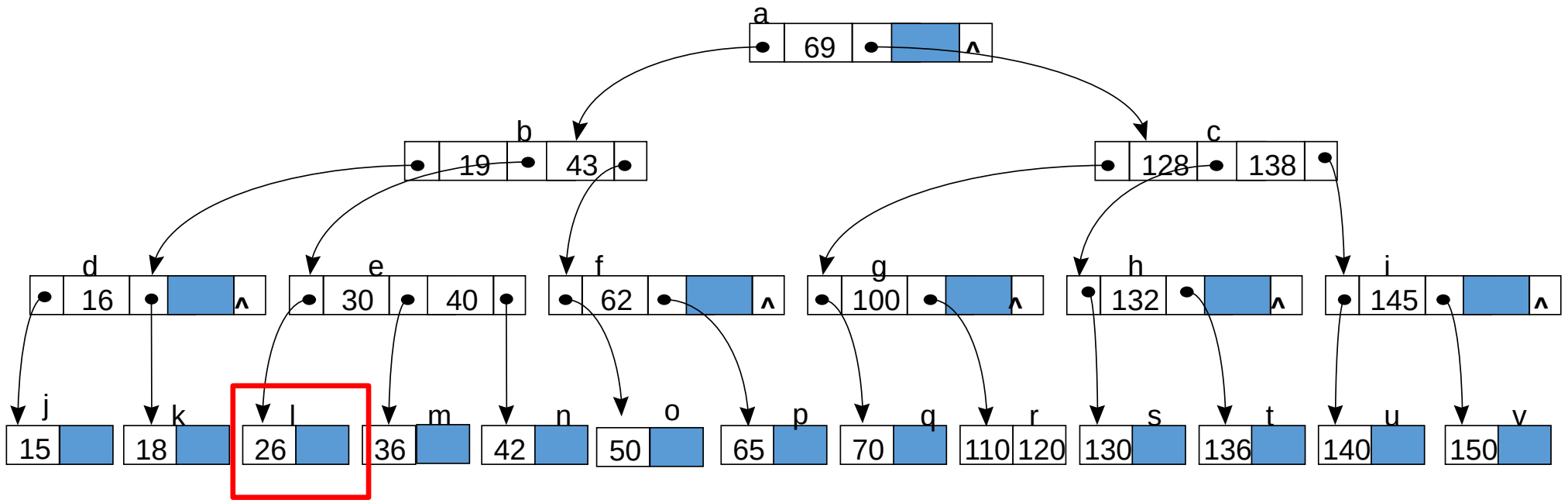
2. 부모 Node에 있는 key 를 언더플로가 일어난 Node로 이동하고, 이 빈 자리로 차출된 key를 이동
**(Key 이동은 삭제와 다르게 처리.
Key가 삭제될 경우 그 자리에 후행Key를 가져다 놓으나,
이동은 key만 이동. 결과 발생할 수 있는 고아 pointer(경로)는 따로 처리)**

# B-Tree 에서의 삭제 : 합병

- **합병(merge)**
  - 재분배가 불가능한 경우(두 형제 Node 모두가 최소의 key 수만을 가짐)에 적용
  - Step 1. Underflow가 된 Node의 오른쪽(또는 왼쪽) 형제 Node에 있는 key들과 이 두 Node를 분리시키는 부모 Node의 key를 합치고(이동시키고) 트리 구조를 조정.
    - **Key 이동은 삭제와 다르게 처리.**
    - **Key가 삭제될 경우 그 자리에 후행Key를 가져다 놓으나, 이동은 Key만 이동. 결과적으로 발생할 수 있는 고아 pointer(경로)는 따로 처리)**
  - Step 2. 합병으로 생긴 빈 Node를 제거
    - 트리 구조가 변경됨
  - 이 합병 작업은 **root Node까지 연쇄적으로 파급될 수 있음**. 이 경우에는 트리의 레벨이 하나 감소될 수도 있음.

# B-Tree 에서의 삭제 예: 합병

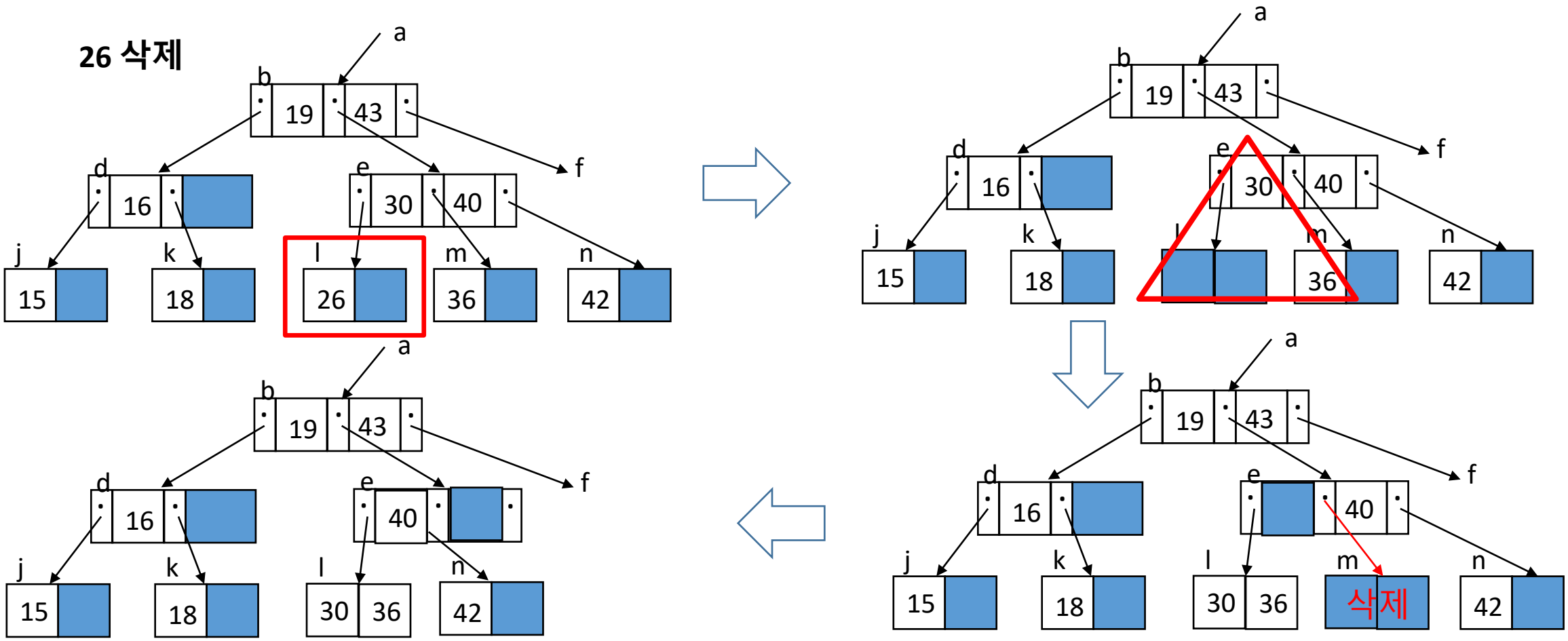
## Node l에서 key 값 26의 삭제




- 26를 삭제하면 node l 가 underflow.
- 형제 Node에서 최소 수보다 key 개수가 많은 형제 Node가 없다. => 재분배 불가능.

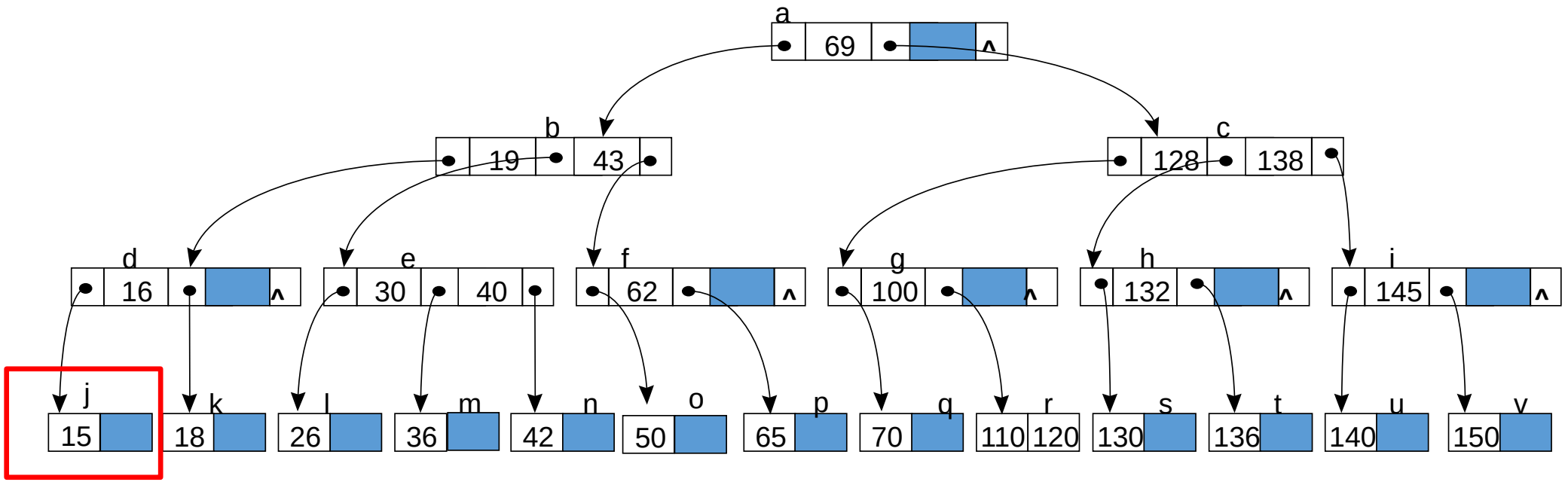
# B-Tree 에서의 삭제 예: 합병

- **합병(merge)**
  - Underflow가 된 Node의 오른쪽(또는 왼쪽) 형제 Node에 있는 key들과 이 두 Node를 분리시키는 부모 Node의 key를 합치고 트리 구조를 조정. 합병으로 생긴 빈 Node는 제거.



# B-Tree 에서의 삭제 예: 합병

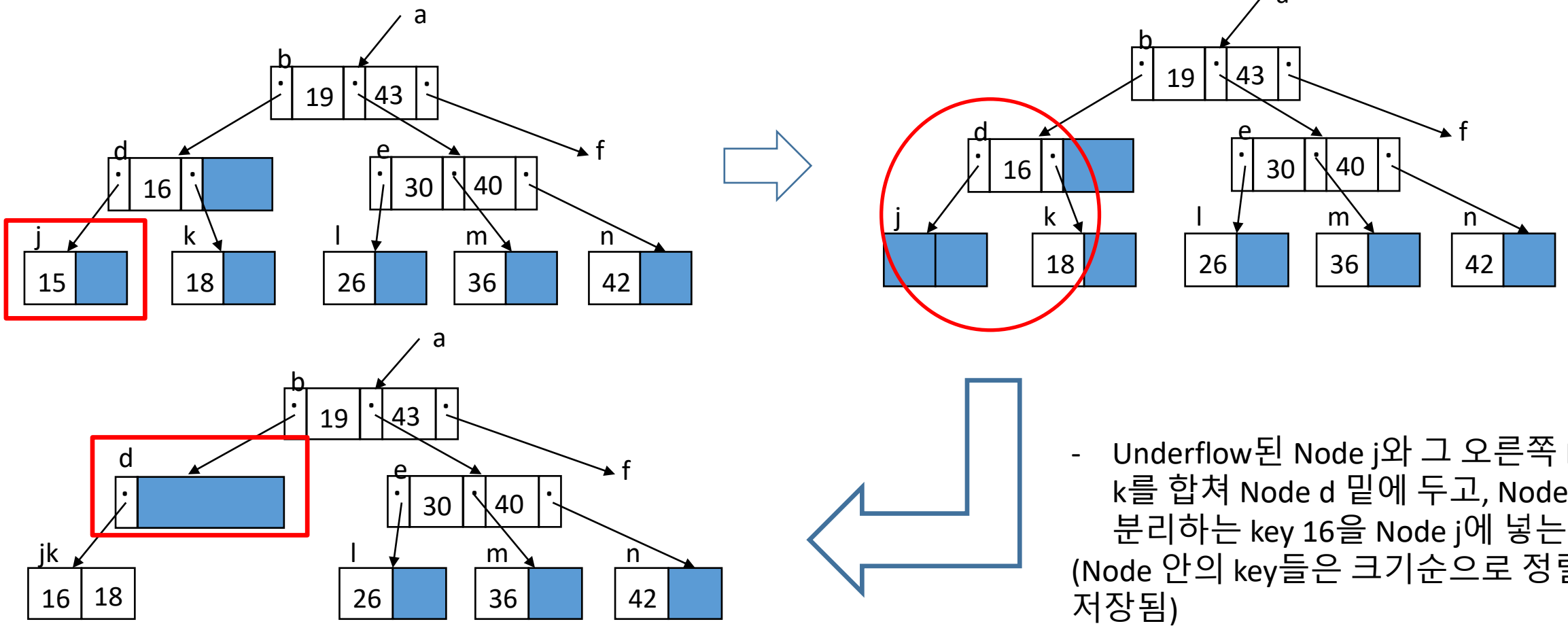
## Node *j*에서 key 값 15의 삭제




- 15를 삭제하면 node j 가 underflow.
- 형제 Node에서 최소 수보다 key 개수가 많은 형제 Node가 없다. => 재분배 불가능.

# B-Tree 에서의 삭제 예: 합병

- **합병(merge)**
  - Underflow 된 Node의 오른쪽(또는 왼쪽) 형제 Node에 있는 key들과 이 두 Node를 분리시키는 부모 Node의 key를 합치고 트리 구조를 조정. 합병으로 생긴 빈 Node는 제거.



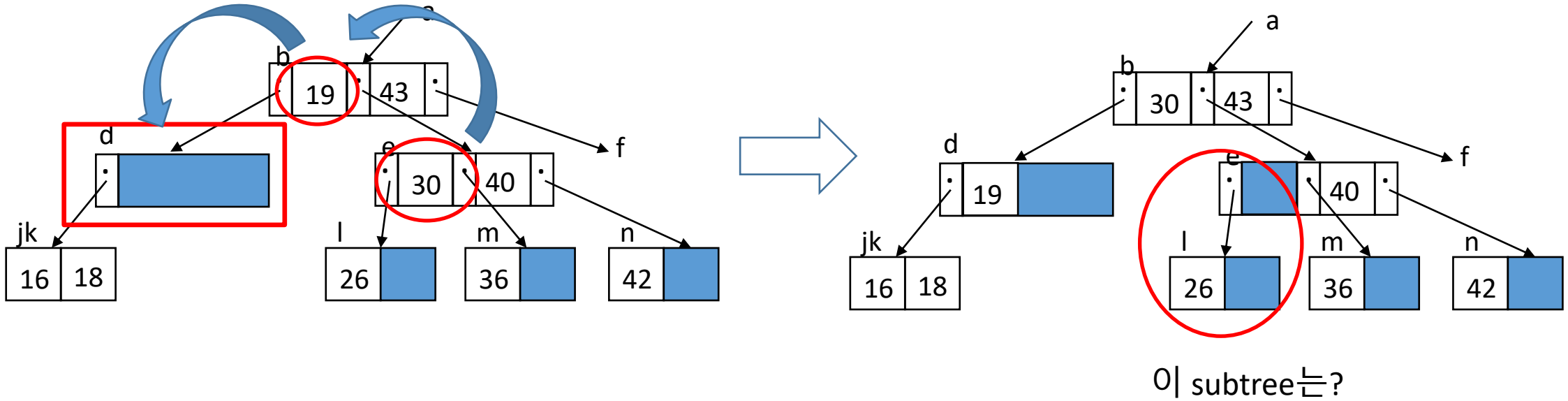

- Underflow된 Node j와 그 오른쪽 Node k를 합쳐 Node d 밑에 두고, Node j와 k를 분리하는 key 16을 Node j에 넣는다.

(Node 안의 key들은 크기순으로 정렬되어 저장됨)

# B-Tree에서의 삭제 예 : 합병 (Cont'd)

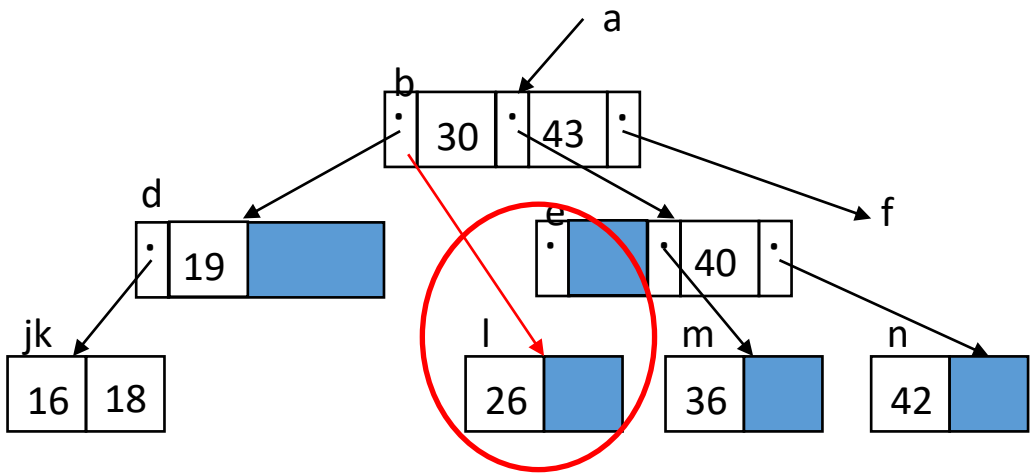
**Node d가 underflow이나 형제 Node e가 최소 key 수 보다 많은 key를 가지므로 중간 Node를 재분배 한다.**

- 해당 Node의 오른쪽이나 왼쪽 형제 Node 중에서 최소 key 수보다 많은 수의 key를 가진 Node에서 key 하나를 차출
- 부모 Node에 있는 분리key를 언더플로가 일어난 Node로 이동하고, 이 빈 자리로 차출된 key를 이동



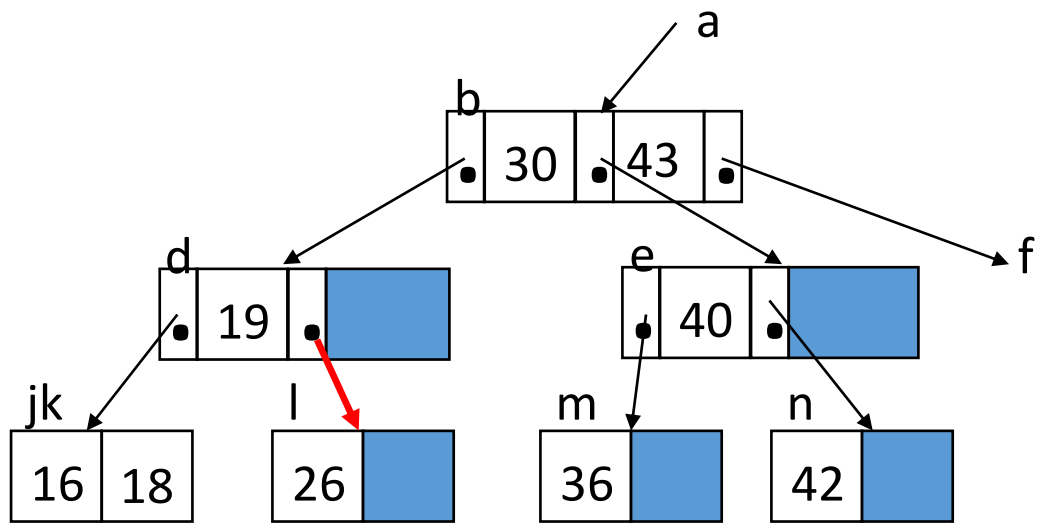

이 subtree는?

# B-Tree에서의 삭제 예 : 합병 (Cont'd)




이 subtree는?

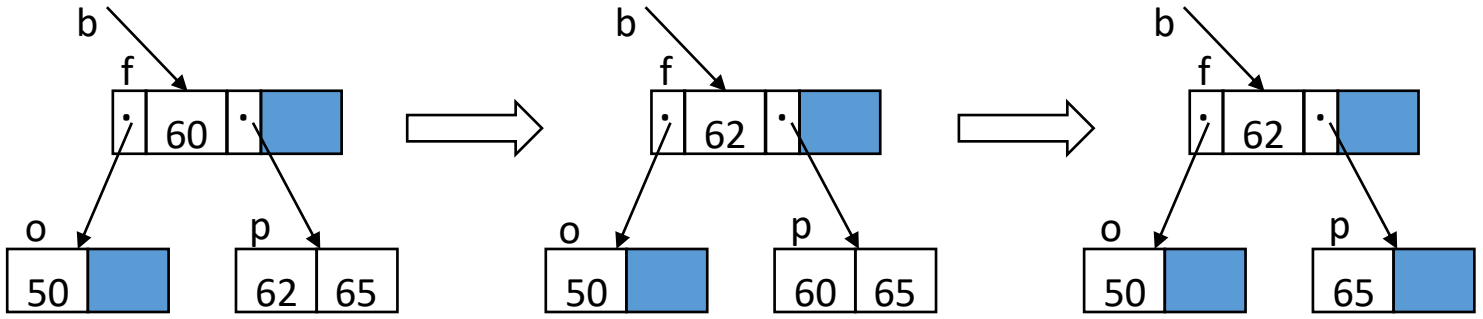
원래 30보다 작고, 19 보다 큰 데이터가 들어있는 sub-tree 이므로 Node 의 19 뒤에 sub-tree를 붙인다.
(30이 이동했으므로 빨간 화살표와 같은 상태,
30왼쪽에 화살표가 2개 있어서 하나를 이동해야 함.
이동한 30의 부모 Key였던 19보다는 큰 것이
확실하므로. 19의 오른쪽에 붙임)



# B-Tree key 삭제 연산의 특성

- key가 삭제 되어도 모든 Leaf Node는 트리의 같은 레벨에 위치한다.
- 바꾸어 말하면 root Node에서 모든 Leaf Node까지의 거리가 같아 균형 트리를 유지한다. (sub-tree의 높이차가 발생하지 않는다)

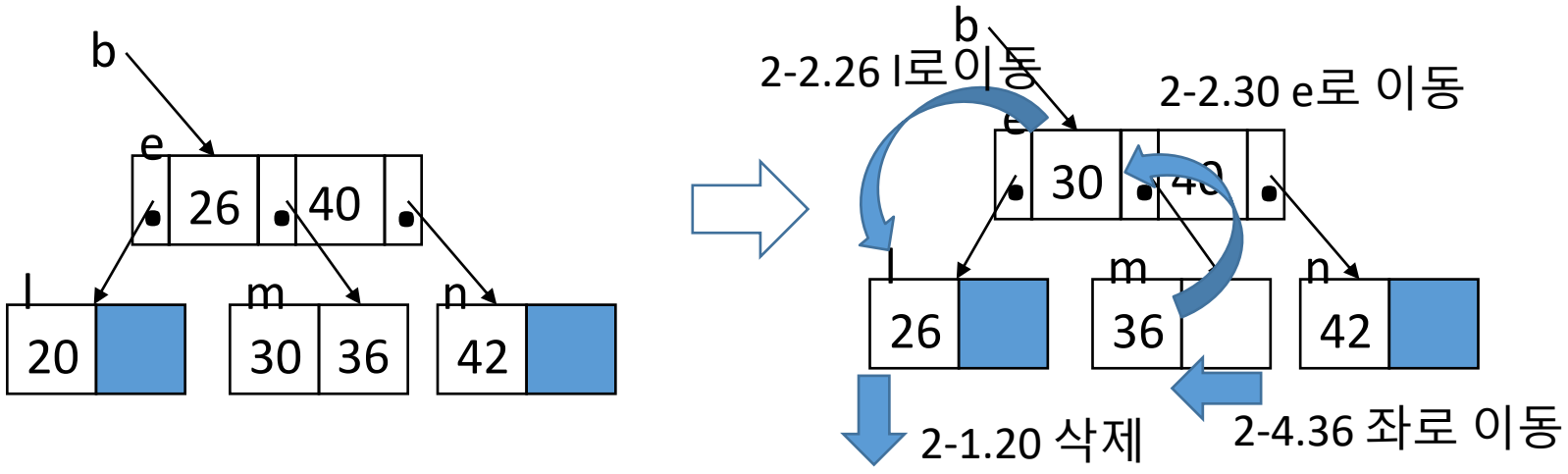
- 이유 :
  - 1. key 삭제 시 삭제되는 key는 Leaf Node에 있거나, Leaf Node로 이동하여 제거되므로 모든 Leaf Node는 트리의 같은 레벨에 위치한다.




**60 삭제 예**

# B-Tree key 삭제 연산의 특성 (Cont'd)

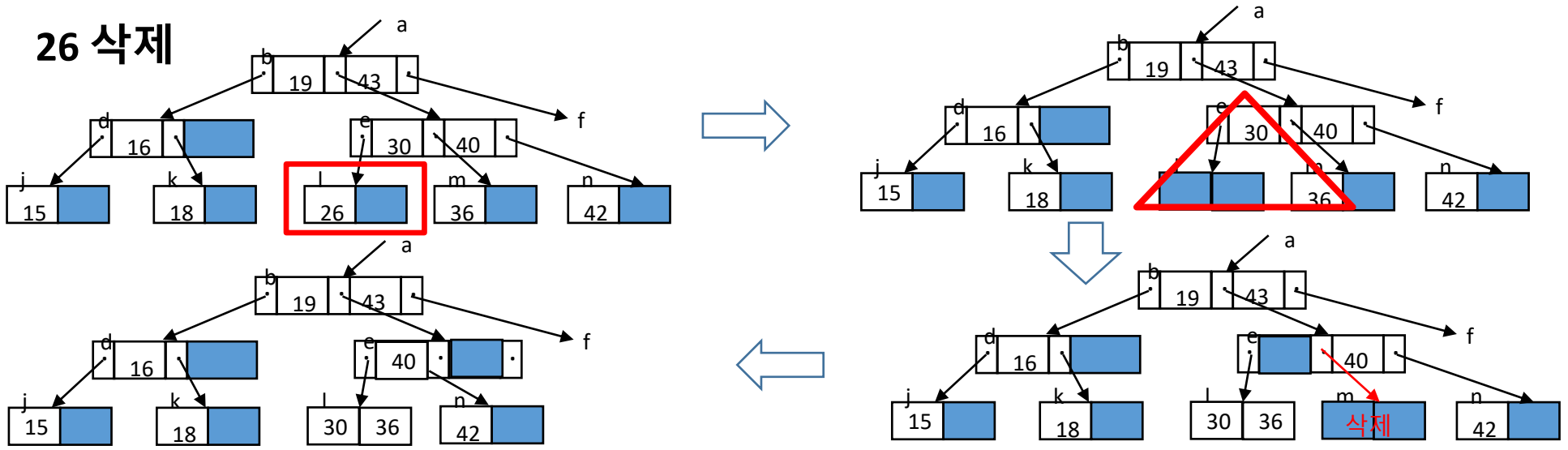
- 2.key 삭제 시 underflow가 발생하면 재분배 혹은 합병이 일어나는데 이 경우도 모든 Leaf Node는 트리의 같은 레벨에 위치한다.
  - **2.1.재분배의 경우**
    - 좌우 형제 Node 중 하나에서 차출된 key 하나와 underflowNode와 key가 차출된 형제 Node를 구분하는 부모 Node의 구분key가 이동한다.
    - 가끔 sub-tree가 다른 Node 밑으로 이동하기는 하나 트리의 높이가 바뀌지 않는 수평 이동이다 (형제 Node 사이의 이동).



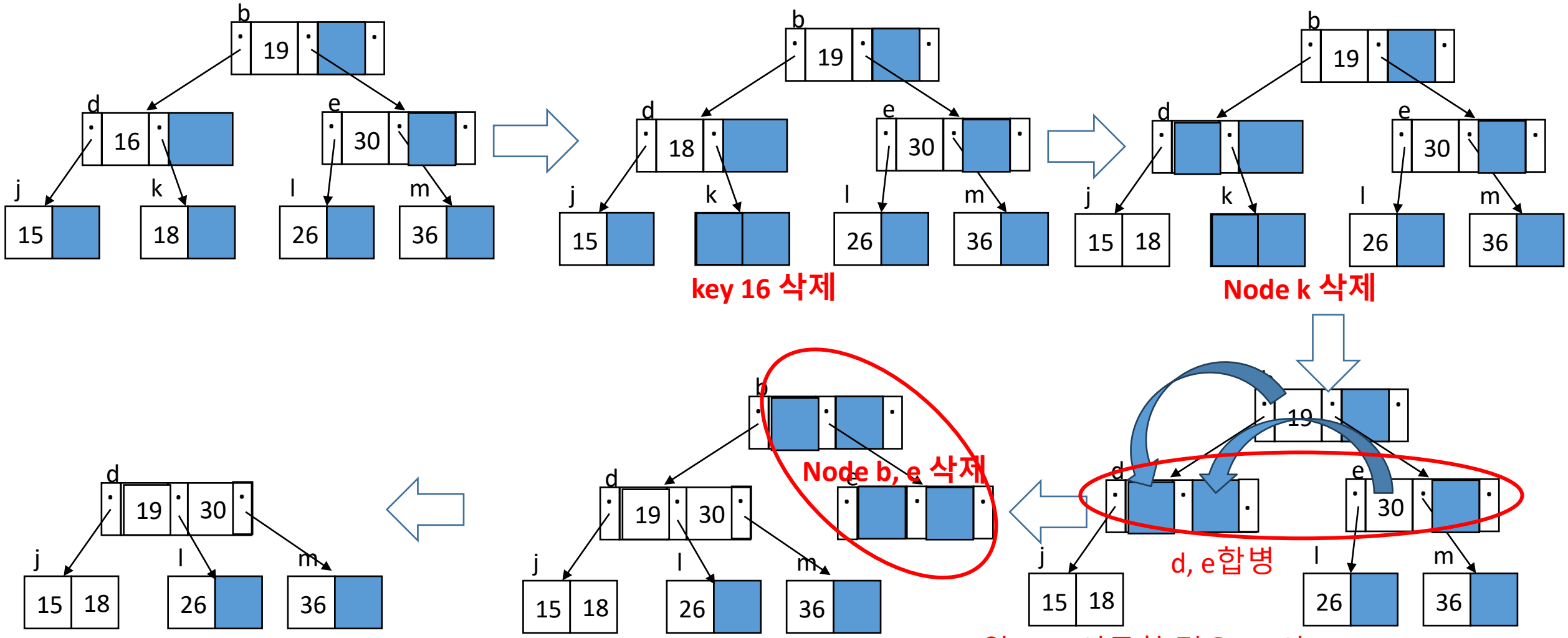

**20 삭제 예**

# B-Tree key 삭제 연산의 특성 (Cont'd)

- key 삭제 시 underflow가 발생하면 재분배 혹은 합병이 일어나는데 이 경우도 모든 Leaf Node는 트리의 같은 레벨에 위치한다.
  - **2.2.합병의 경우**
    - 두 개의 Node가 합병되어 하나의 Node가 생성되지만, 이 경우도 같은 레벨의 Node n 개가 n-1개로 하나 줄어드는 것일 뿐 트리의 높이가 변경되지는 않는다.
    - root 바로 아래의 두 개의 Node가 합병되어 root가 생성될 경우도 트리의 아래 레벨이 아닌 위 레벨이 하나 줄어드는 것이므로 모든 Leaf Node는 트리의 같은 레벨에 위치한다.



# B-Tree key 삭제 연산의 특성 (Cont'd)




**Leaf Node의 구조는 변화 없고** root Node가 사라짐.

# B-Tree 부록

**B-Tree에서, 임의의 중간 Node의 key의 후행 key는 항상 Leaf에 존재**

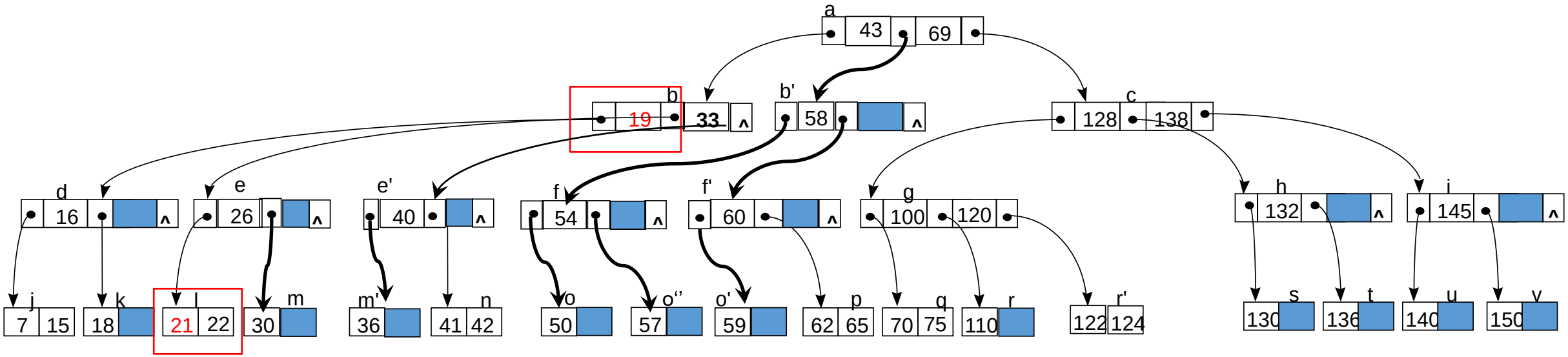
# B-Tree에서, 임의의 중간 Node의 key의 후행 key는 항상 Leaf에 존재

- 임의의 key $k_i$ 가 중간 Node에 존재할 경우, $k_i$ 의 후행 key ($k_i$보다 큰 key 중 제일 작은 key) $k_{i+1}$ 은 항상 Leaf Node에 존재한다.
  - $k_{i+1}$ 는 $k_i$ 의 바로 오른쪽 sub-tree에 존재하는 key 중에서 제일 작은 key

# B-Tree에서, 임의의 중간 Node의 key의 후행 key는 항상 Leaf에 존재 (Cont'd)

**Key 삽입의 경우 :**

- B트리의 Key는 항상 leaf에 삽입됨.
- Key가 중간 Node에 삽입되는 경우는, 분할(split)에서 하나의 Node를 두개의 Node로 분할하는 과정에서 분할 대상 Node의 중간 Key만이 부모 Node (중간 Node)에 삽입 가능.
- 부모 Node로 올라가는 key를 $k_i$ 라 하면 중간 Node에 저장되면
- $k_i$ 의 후행 Key인 $k_{i+1}$ 은 Leaf Node에 남아 있음.



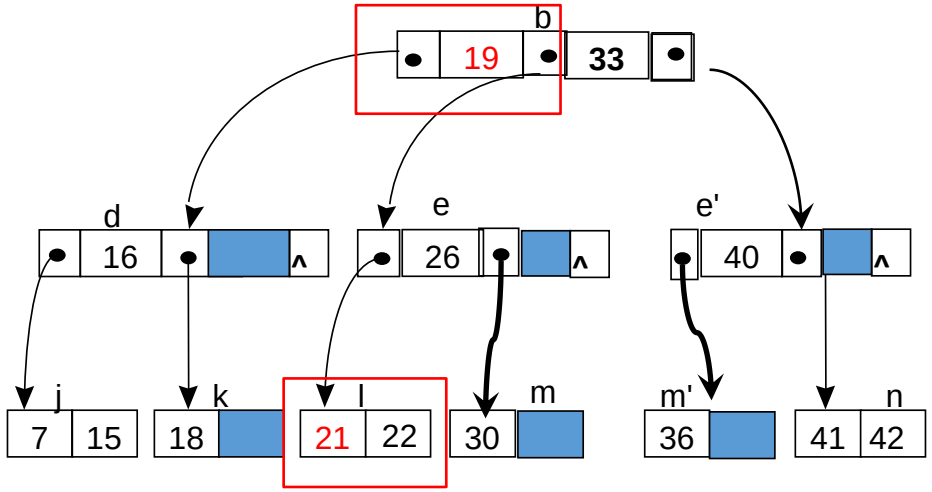

23 혹은 20 추가 되어 분할되어도 각각 21, 20은 Leaf에 남아 있다.

# B-Tree에서, 임의의 중간 Node의 key의 후행 key는 항상 Leaf에 존재 (Cont'd)

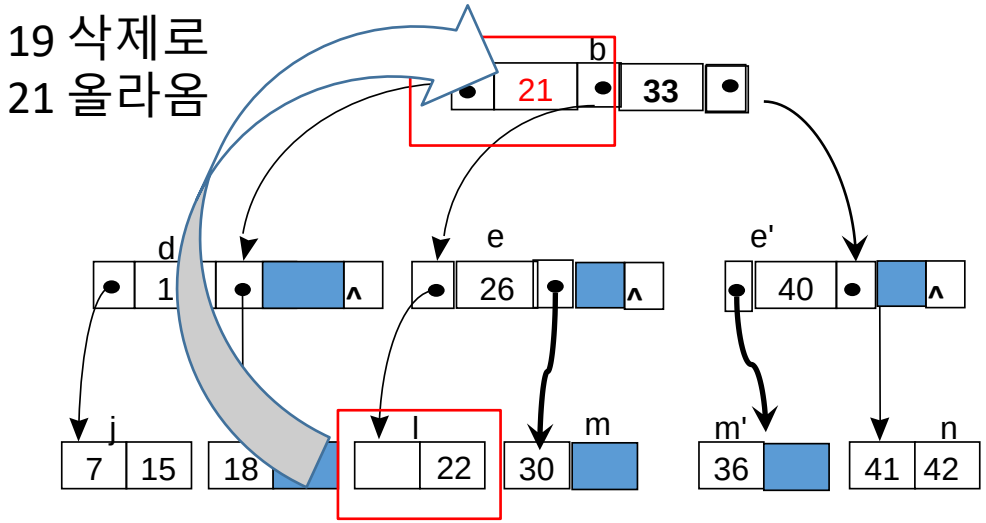
- 이 후 $k_{i+1}$ 이 새로운 삽입으로 발생한 분할로 중간 Node에 저장될 수 있나?
  => 저장될 수 있다. 그러나 $k_{i+1}$ 가 중간 Node에 올라갈 수 있다는 것은 $k_{i+1}$ 보다 작고, $k_i$ 보다 큰 $k_i$ 의 새로운 후행키가 존재하는 것을 뜻하며 이는 Leaf 에 남아있다.

# B-Tree에서, 임의의 중간 Node의 key의 후행 key는 항상 Leaf에 존재 (Cont'd)

- **Key 삭제의 경우:** 중간 Node의 어떤 key가 삭제되면서 해당 key의 후행 key인 $k_i$ 가 Leaf에서 중간 Node에 올라온 경우



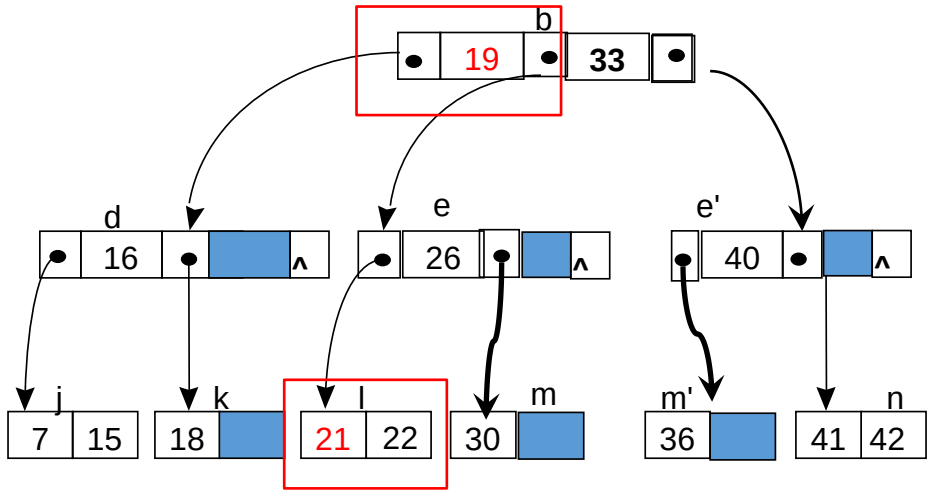

**트리에서 중간 Node의 key 19 삭제 연산이 일어남.**



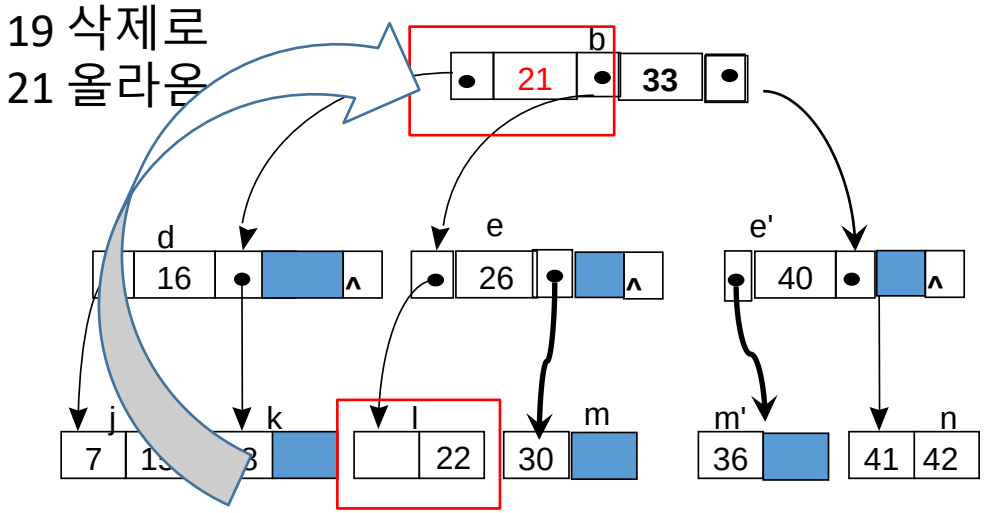

**이 경우 21의 후행key도 Leaf Node에 존재할까?**

# B-Tree에서, 임의의 중간 Node의 key의 후행 key는 항상 Leaf에 존재 (Cont'd)

- **Key 삭제의 경우:** 중간 Node의 어떤 key가 삭제되면서 해당 key의 후행 key인 $k_i$ 가 Leaf에서 중간 Node에 올라온 경우
  - $k_{i+1}$ 가 Leaf Node에 남아 있을 경우 => 문제 없음.



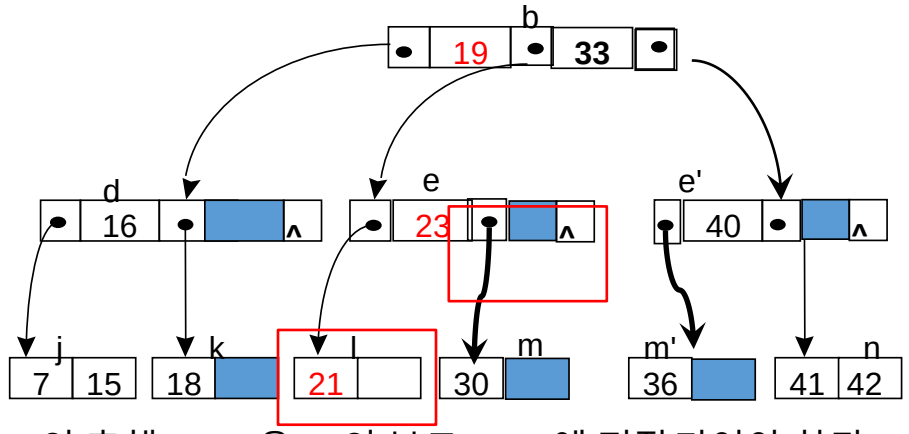

**트리에서 중간 Node의 key 19 삭제 연산이 일어남.**



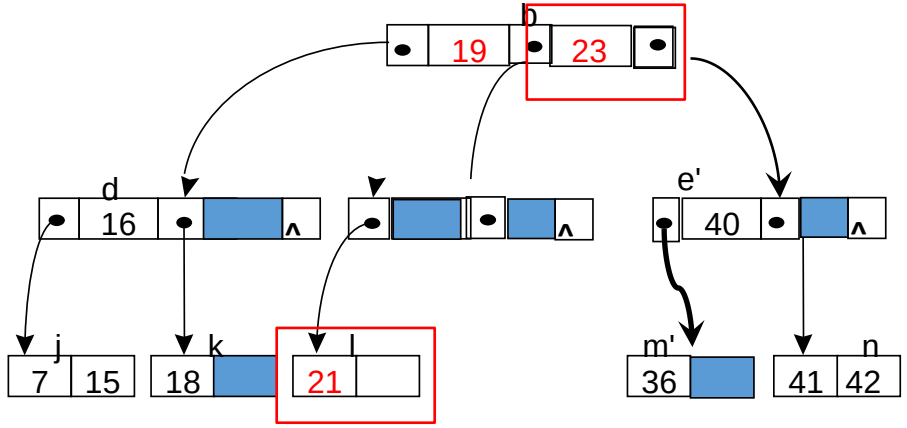

**중간 Node로 올라간 Key의 후행Key가 원래 Leaf Node에 있었을 경우 문제 없음.**

# B-Tree에서, 임의의 중간 Node의 key의 후행 key는 항상 Leaf에 존재 (Cont'd)

- **Key 삭제의 경우:** 중간 Node의 어떤 key가 삭제되면서 해당 key의 후행 key인 $k_i$ 가 Leaf에서 중간 Node에 올라온 경우
  - $k_{i+1}$ 가 sub-tree의 중간 Node에 존재 할 경우
    - $k_{i+1}$ 이 $k_i$ 바로 다음으로 큰 key이므로, 만약 $k_{i+1}$ 이 중간 node에 저장되어 있다면 $k_{i+1}$ 을 저장하는 Node는 $k_i$ 를 저장하고 있었던 Leaf Node의 부모 Node이다.
    - 즉 $k_i$는 $k_{i+1}$의 바로 왼쪽 자식 Node이다.




21의 후행key 23은 21의 부모 Node에 저장되어야 한다.

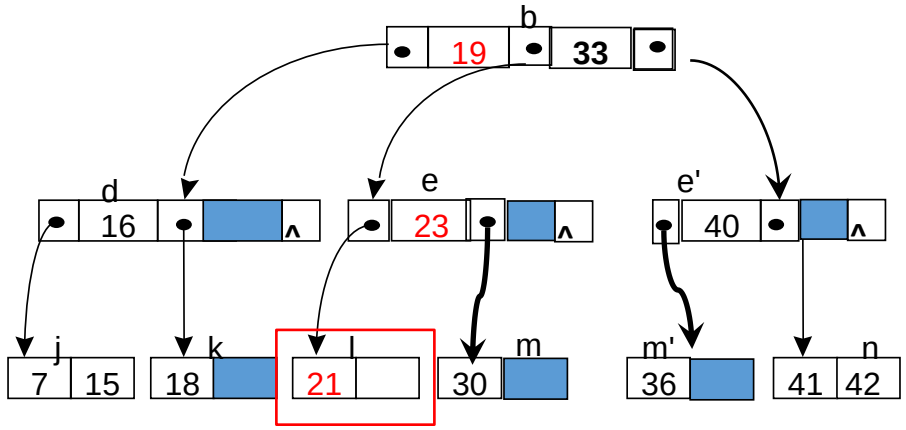



21의 후행key 23이 21의 부모 Node 보다 위에 저장되어 있다면, 23은 21의 후행 key가 될 수 없다.

**19 삭제전의 19의 후행 key 21 과 21의 후행 key 23의 예.**

# B-Tree에서, 임의의 중간 Node의 key의 후행 key는 항상 Leaf에 존재 (Cont'd)

- 후행key의 정의에 의해 $k_i$ 이 원래 삭제된 key의 오른쪽 sub-tree에서 가장 작은 key였으므로,
  - $k_{i+1}$ 의 왼쪽 자식 Node에는 $k_i$ 하나 밖에 존재할 수 없다.
  - 그런데 $k_i$ 가 더 상위의 중간 Node로 올라가 버리므로 해당 Node의 key 수는 0이 되어 재분배 혹은 합병이 일어난다. => $k_{i+1}$ 이 Leaf Node로 내려온다.




**19 삭제전의 19의 후행 key 21 과 21의 후행 key 23의 예.**

# B-Tree에서, 임의의 중간 Node의 key의 후행 key는 항상 Leaf에 존재 (Cont'd)