6장 인덱스 구조

Part B

❖ B-트리

- **Bayer & McCreight가 고안**

- **균형된 m-원 탐색 트리 (Balanced MST)**
  - 가장 많이 사용되는 인덱스 방법
  - 효율적인 균형 알고리즘을 제공

- **B-트리의 노드 구조 (MST와 동일)**

$<n, p_0, <K_1, A_1>, P_1, <K_2, A_2>, P_2, \ldots, P_{n-1}, <K_n, A_n>, P_n>$

  - n ($1 \leq n \leq m-1$): 한 노드 내의 키 값의 수
  - P_i ($0 \leq i \leq n$) : 서브트리에 대한 포인터
  - Ki ($1 \leq i \leq n$) : 키 값
  - Ai ($1 \leq i \leq n$) : 키 값으로 K_i 를 가진 레코드에 대한 포인터

▶ B-트리의 정의

◆ 차수 m인 B-트리의 정의

① B-트리는 공백이거나, 높이가 1 이상인 m-원 탐색 트리 (MST)

② 루트와 리프를 제외한 내부 노드

- ⌈ m/2 ⌉≤ 서브트리수 ≤ m
- Note: 따라서 내부노드는 적어도 ⌈ m/2 ⌉- 1개 이상의 키 값을 가짐. (노드의 반 이상 채워짐)
- Note: 리프는 서브트리가 없고, 적어도 ⌈ m/2 ⌉- 1개 이상의 키 값을 가짐.

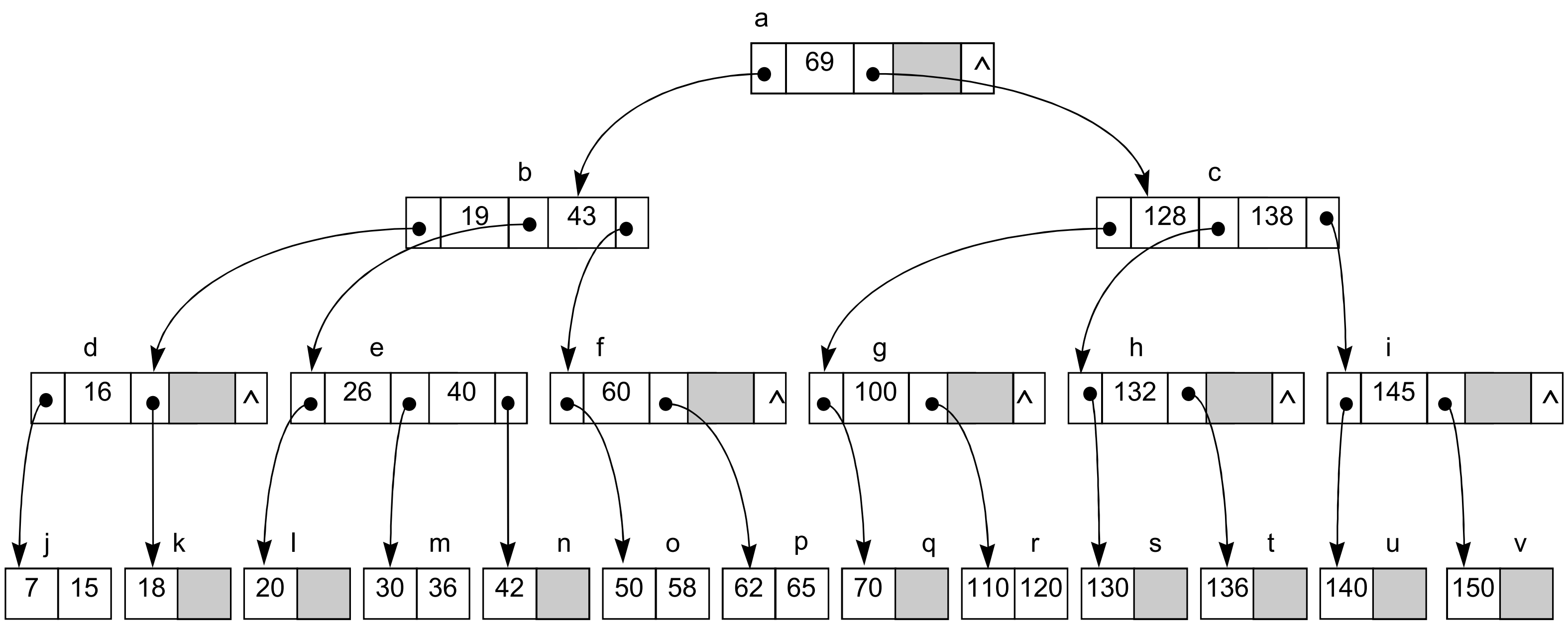
③ 루트 : 리프가 아니면 적어도 두 개의 서브트리 (한 개의 키 값)를 가짐

- 루트는 적어도 ⌈ m/2 ⌉- 1개 이상의 키 값을 가져야 한다는 제약이 없음 (m=5 경우). 루트가 꽉 차면 분기

④ 모든 리프는 같은 레벨

- -트리의 장점
  - 삽입, 삭제 뒤에도 균형 상태 유지 (재균형이 필요 없음)
  - 저장 장치의 효율성
    - 각 노드의 반 이상 키 값 저장
    - 최악 $O(\log_n(N+1))$

▶ 3-원 B-트리 구조



m=3

- 내부노드는 2 이상 3 이하의 서브트리를 가짐.
- 루트를 제외한 모든 노드는 1개 이상의 키 값을 가짐.

▶ B-트리에서의 연산: 검색

- 검색 : **m-**원 탐색 트리의 직접 검색과 같은 과정
  - 직접 탐색(random access)
    - 키 값에 의존한 분기
    - 예: 42 검색 시
  - 순차 탐색(sequencial access)
    - 중위 순회(inorder traversal)

▶ B-트리에서의 연산: 삽입

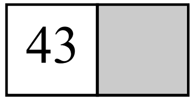
- 삽입 : 항상 리프 노드에 삽입
  ① 여유 공간이 있는 경우: overflow가 발생하지 않는 경우
  - : 단순히 순서에 맞게 삽입

  ② 여유 공간이 없는 경우 : overflow가 발생하는 경우
  - 해당 노드를 두 개의 노드로 **분할(split)**해야 한다.
  - 해당 노드의 키 값에 새로운 키 값 삽입했다고 가정
  - 중간 키 값을 중심으로 큰 키들은 새로운 노드에 저장
  - 중간 키 값 : 분할된 노드의 부모 노드로 올라가 “삽입”
  - 이 때, 다시 overflow 발생시 위와 같은 분할(split) 작업을 반복

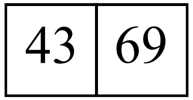
▶ 3차 B-트리 생성 과정

◆ 키 값 **43, 69, 138, 19** 순으로 삽입하여 생성

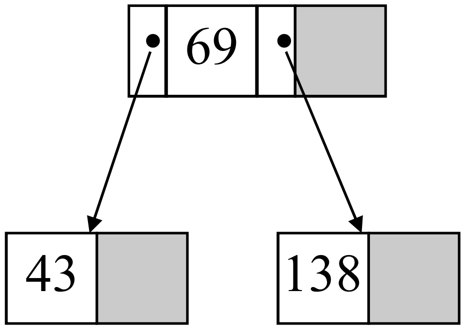
(a) 크기가 2인 공백 루트 노드



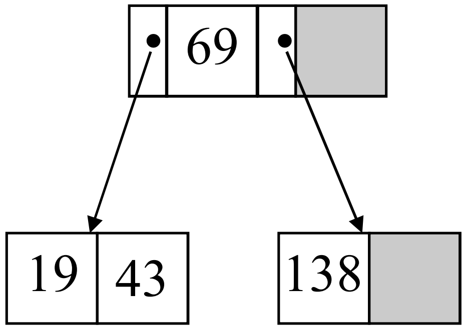
(b) 키 값 43의 삽입(노드 1개의 3차 B-트리)



(c) 키 값 69의 삽입(노드 1개의 3차 B-트리)



(d) 키 값 138의 삽입(노드 3개의 3차 B-트리)

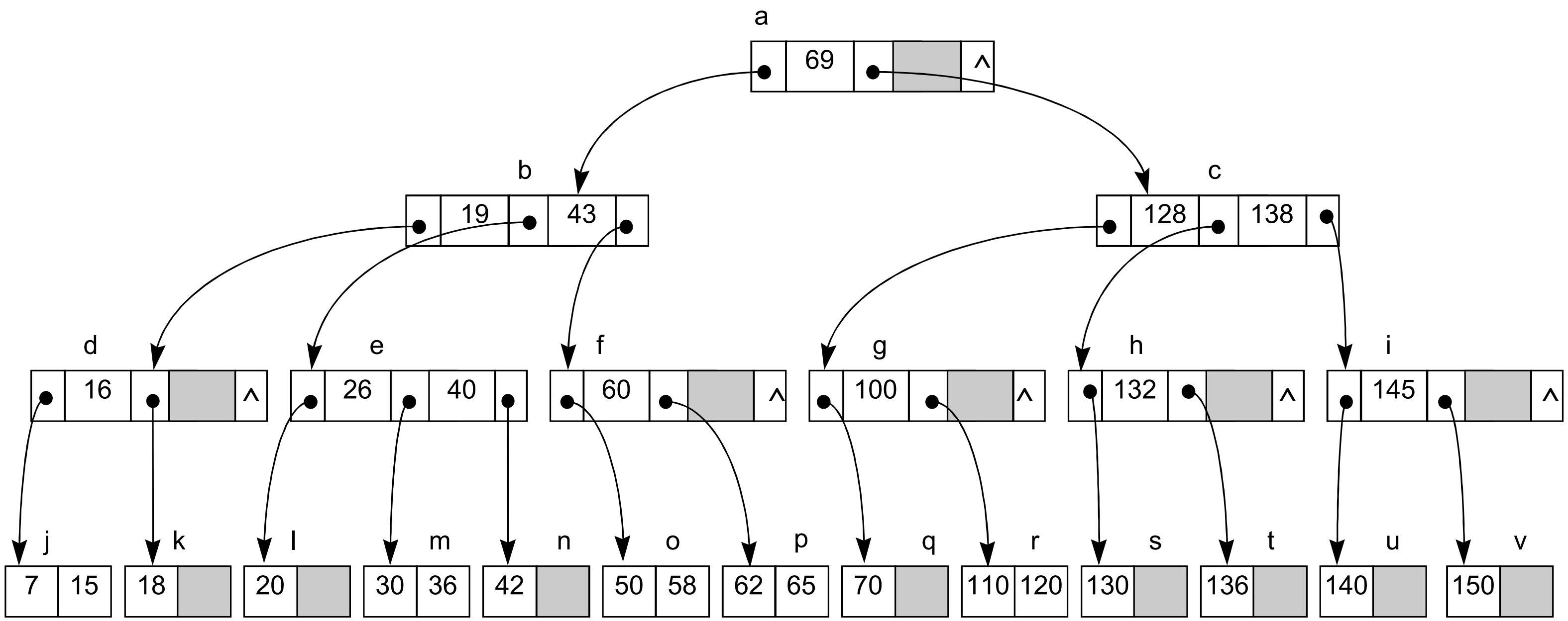


(e) 키 값 19의 삽입(노드 3개의 3차 B-트리)

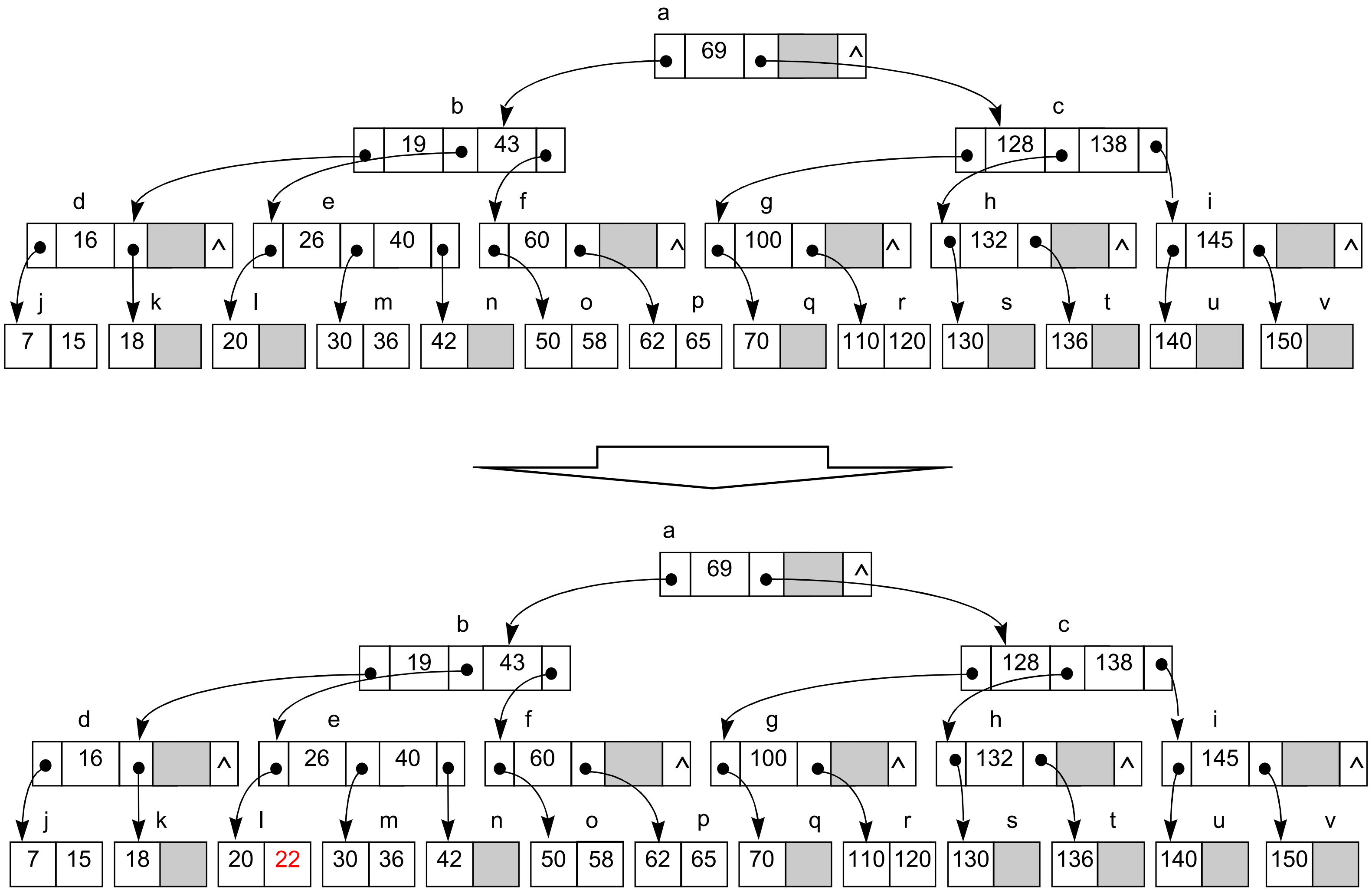
- 루트 노드의 키의 최소 개수
  - 루트를 제외한 모든 노드는 적어도 ⌈ m/2 ⌉- 1개 이상의 키 값을 가져야함.
  - 따라서 처음 루트 노드가 분할될 때, 생성되는 두 개의 리프 노드에는 각각⌈ m/2 ⌉- 1개의 키가 있어야함.
  - 따라서 루트 노드가 분할되기 직전에는, 루트에 최소 (⌈ m/2 ⌉- 1) x 2개의 키를 담고 있어야 함.
- 예
  - m=3, 루트 노드의 키의 최소 개수=2
  - m=4, 루트 노드의 키의 최소 개수=2 (3)
  - m=5, 루트 노드의 키의 최소 개수=4
  - m=6, 루트 노드의 키의 최소 개수=4 (5)
  - m=7, 루트 노드의 키의 최소 개수=6
  - m=8, 루트 노드의 키의 최소 개수=6 (7)

예제 1

- 다음 **B-**트리(그림 **6-18**)에 새로운 키 값 **22, 41, 59, 57, 54, 33, 75, 124, 122, 123** 삽입

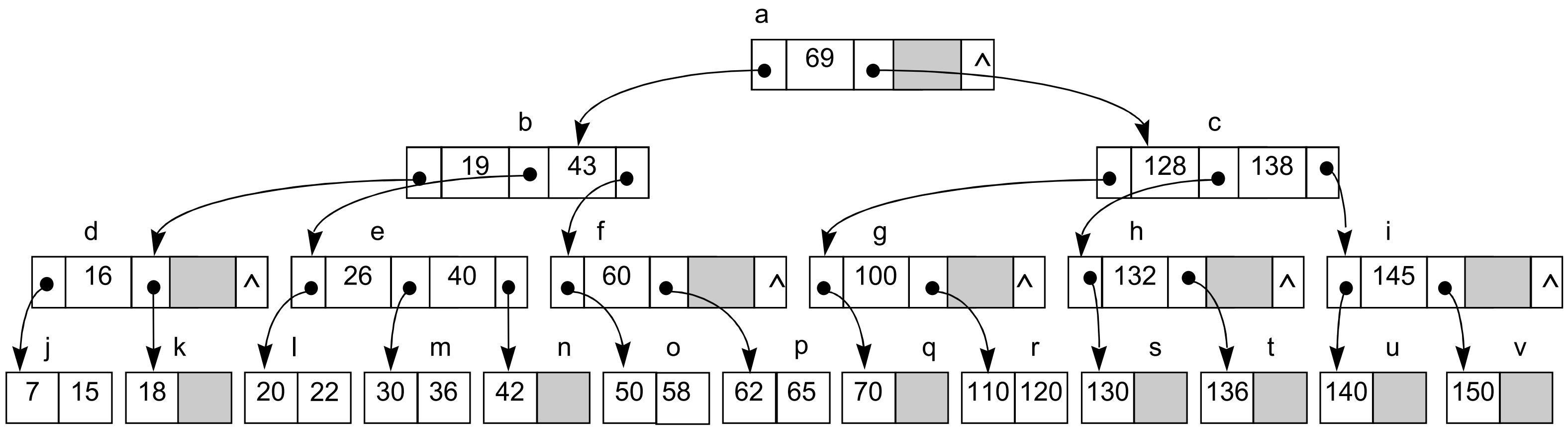


22 삽입

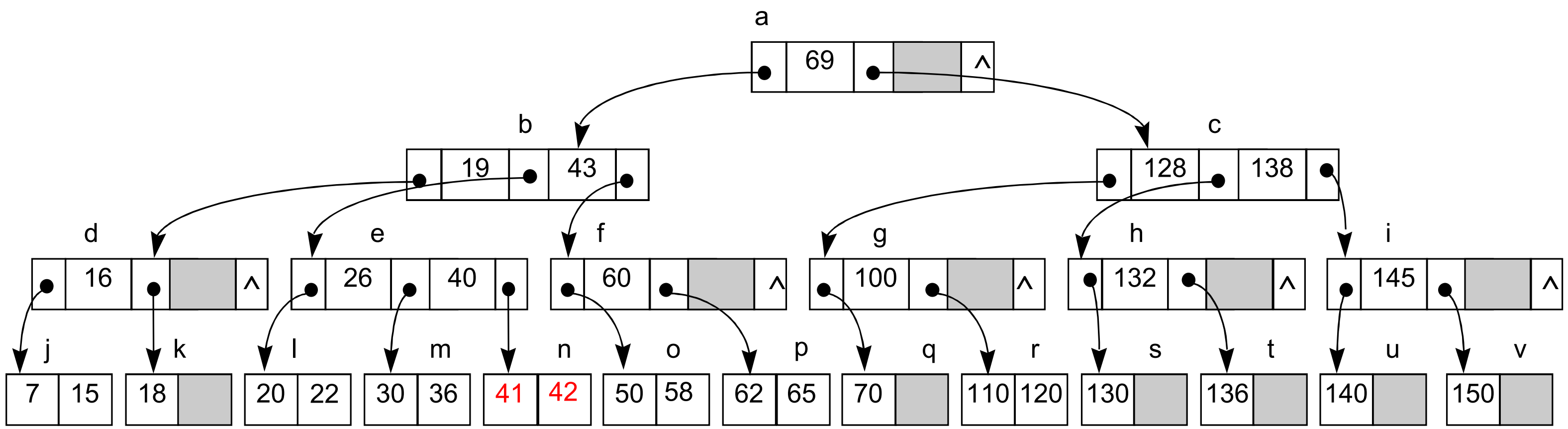


노드 o에서 키 값 58의 삭제

41 삽입

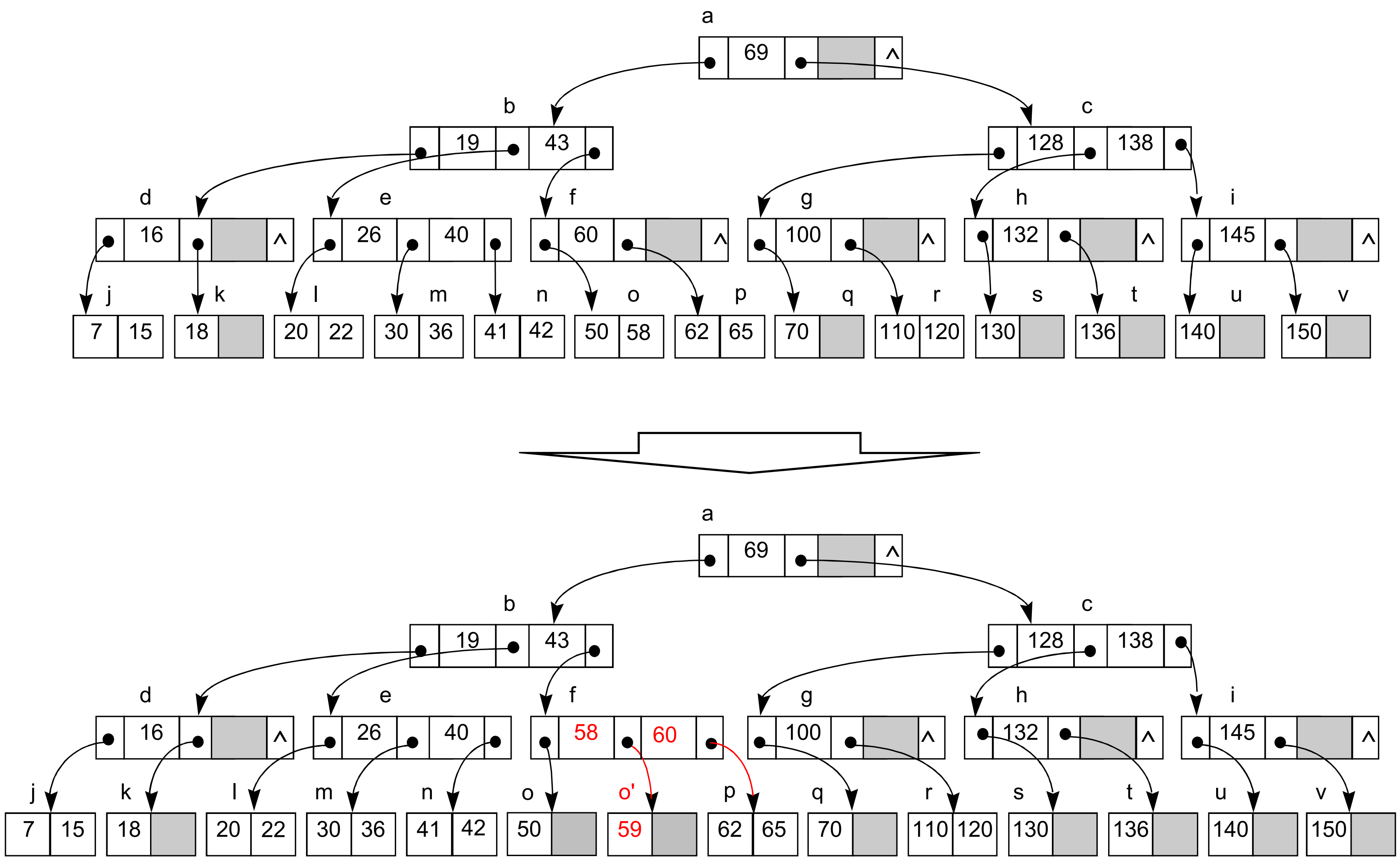


a
69
b
19
43
c
128
138
d
16
e
26
40
f
60
g
100
h
132
i
145
j
7 15
k
18
l
20 22
m
30 36
n
42
o
50 58
p
62 65
q
70
r
110 120
s
130
t
136
u
140
v
150

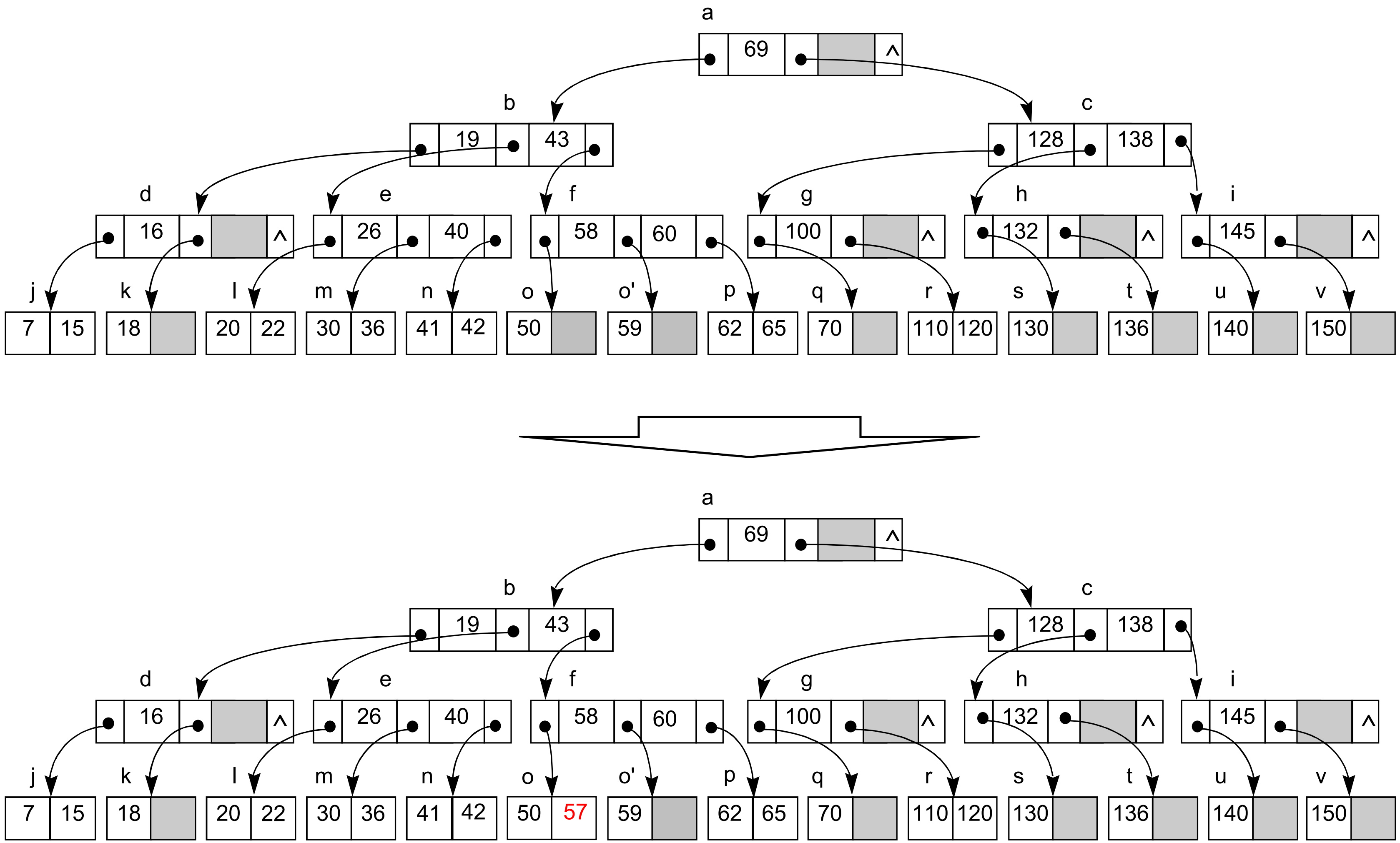


a
69
b
19
43
c
128
138
d
16
e
26
40
f
60
g
100
h
132
i
145
j
7 15
k
18
l
20 22
m
30 36
n
41 42
o
50 58
p
62 65
q
70
r
110 120
s
130
t
136
u
140
v
150

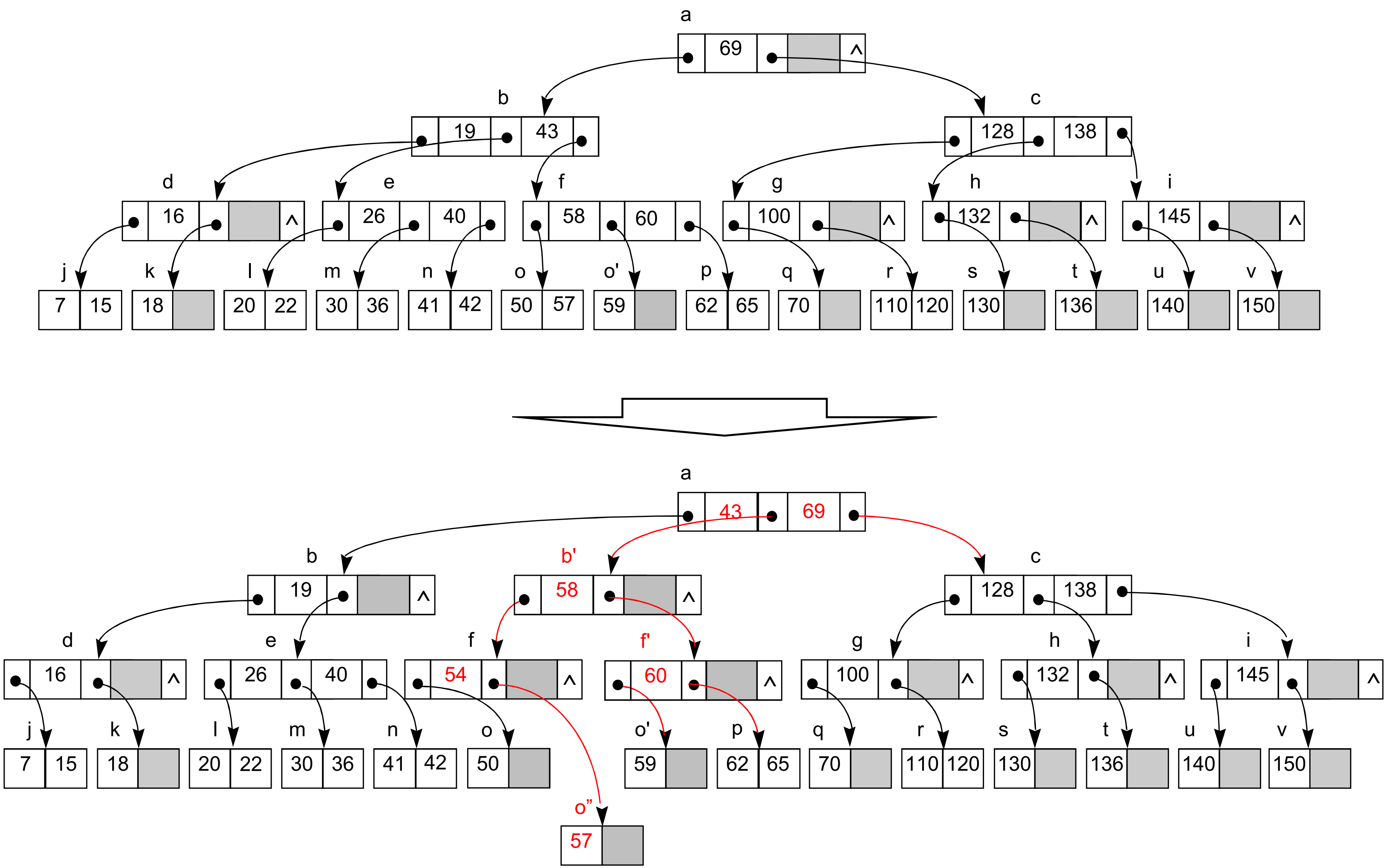
59 삽입


a
69
b
19
43
c
128
138
d
16
e
26
40
f
60
g
100
h
132
i
145
j
7
15
k
18
l
20
22
m
30
36
n
41
42
o
50
58
p
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150
a
69
b
19
43
c
128
138
d
16
e
26
40
f
58
60
g
100
h
132
i
145
j
7
15
k
18
l
20
22
m
30
36
n
41
42
o
50
o'
59
p
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150

57 삽입

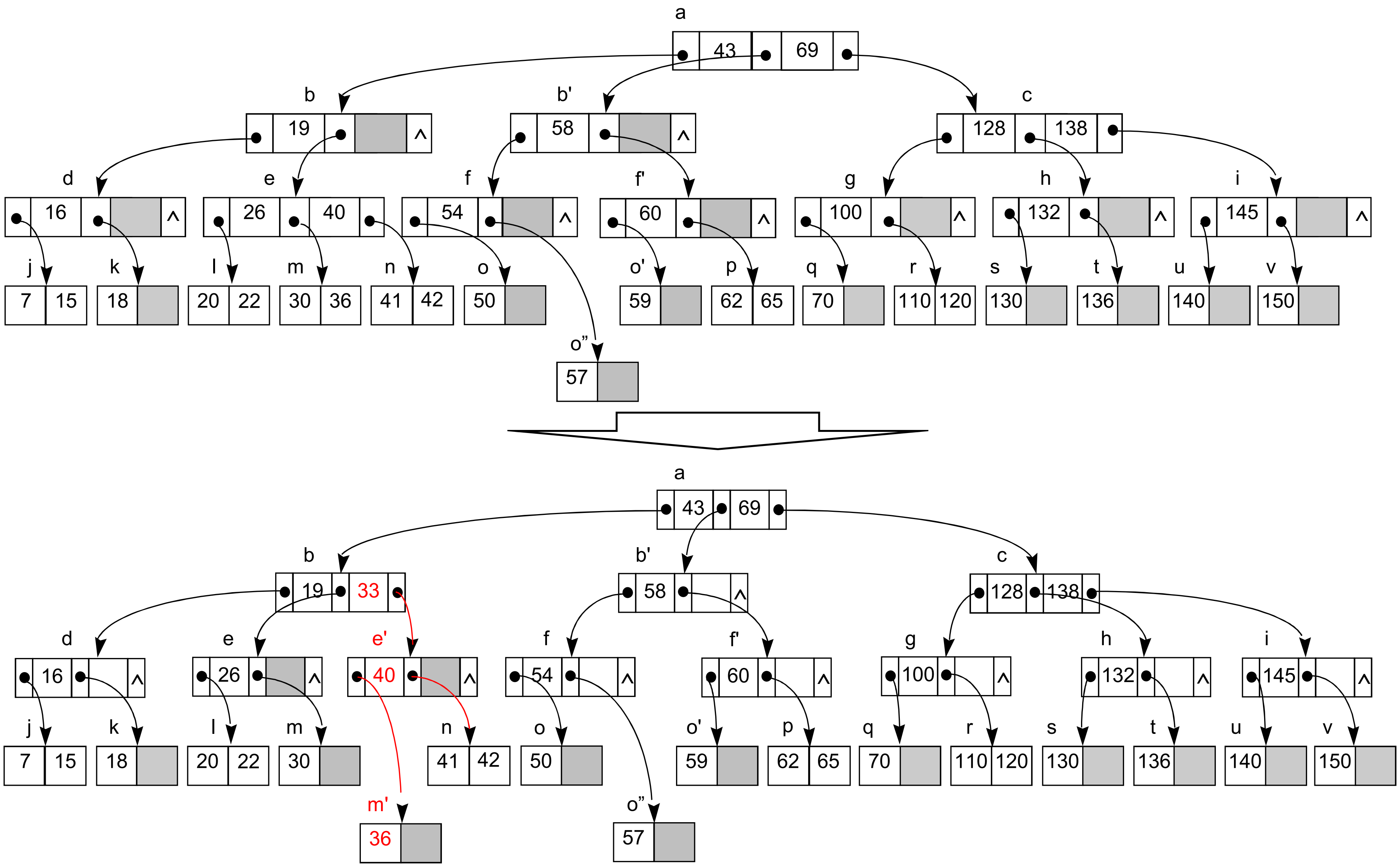

a
69
b
19
43
c
128
138
d
16
e
26
40
f
58
60
g
100
h
132
i
145
j
7
15
k
18
l
20
22
m
30
36
n
41
42
o
50
o'
59
p
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150
a
69
b
19
43
c
128
138
d
16
e
26
40
f
58
60
g
100
h
132
i
145
j
7
15
k
18
l
20
22
m
30
36
n
41
42
o
50
57
o'
59
p
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150

54 삽입

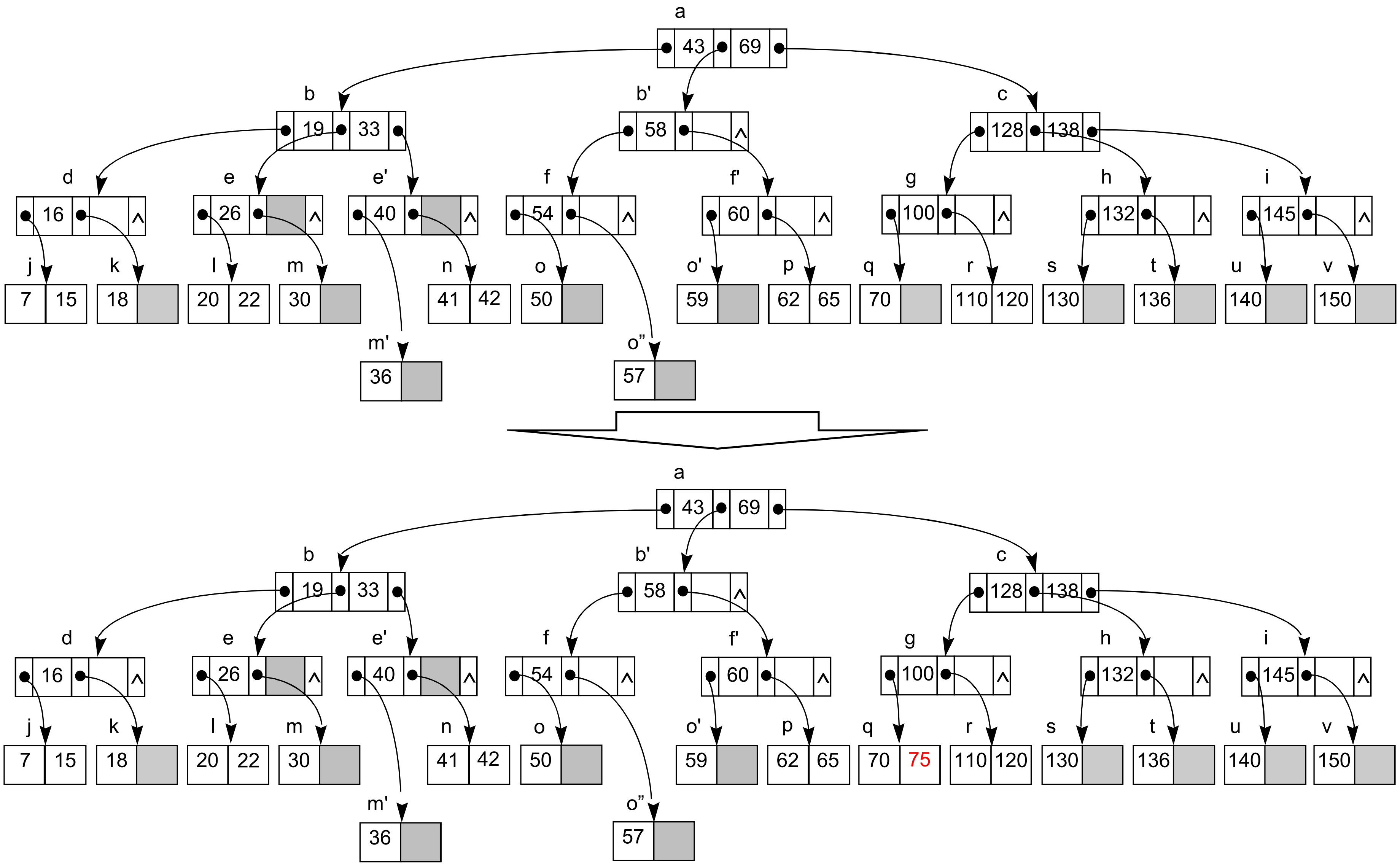


a
69
b
19
43
c
128
138
d
16
e
26
40
f
58
60
g
100
h
132
i
145
j
7
15
k
18
l
20
22
m
30
36
n
41
42
o
50
57
o'
59
p
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150
a
43
69
b
19
b'
58
c
128
138
d
16
e
26
40
f
54
f'
60
g
100
h
132
i
145
j
7
15
k
18
l
20
22
m
30
36
n
41
42
o
50
o'
59
p
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150
o”
57

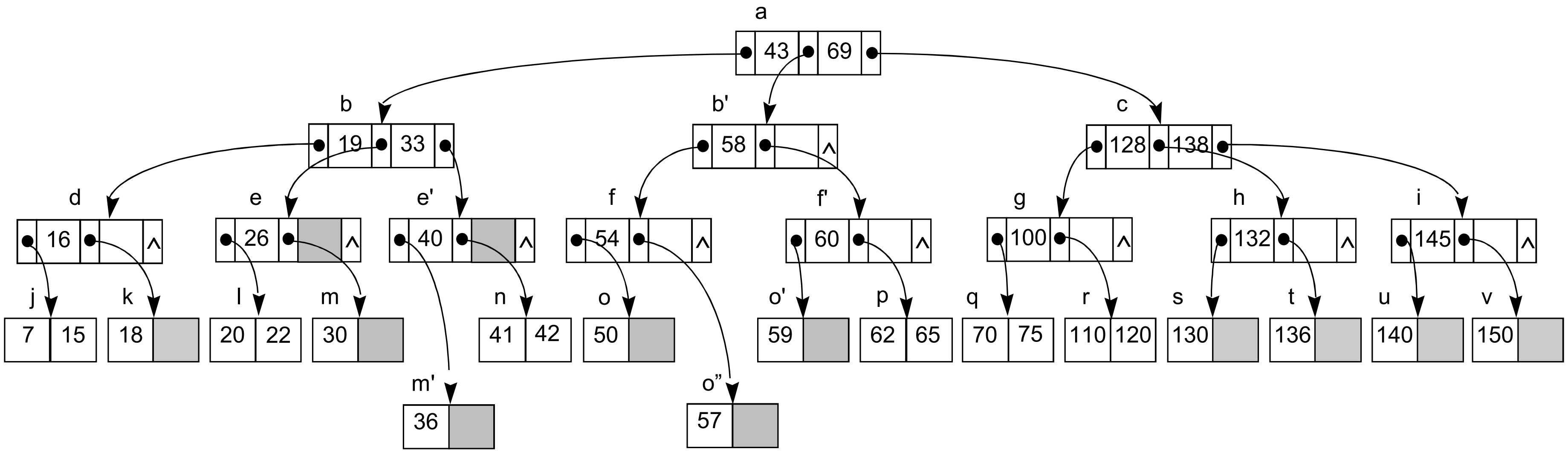
33 삽입


a
43
69
b
19
b'
58
c
128
138
d
16
e
26
40
f
54
f'
60
g
100
h
132
i
145
j
7
15
k
18
l
20
22
m
30
36
n
41
42
o
50
o'
59
p
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150
o"
57
a
43
69
b
19
33
b'
58
c
128
138
d
16
e
26
e'
40
f
54
f'
60
g
100
h
132
i
145
j
7
15
k
18
l
20
22
m
30
n
41
42
o
50
o'
59
p
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150
m'
36
o"
57

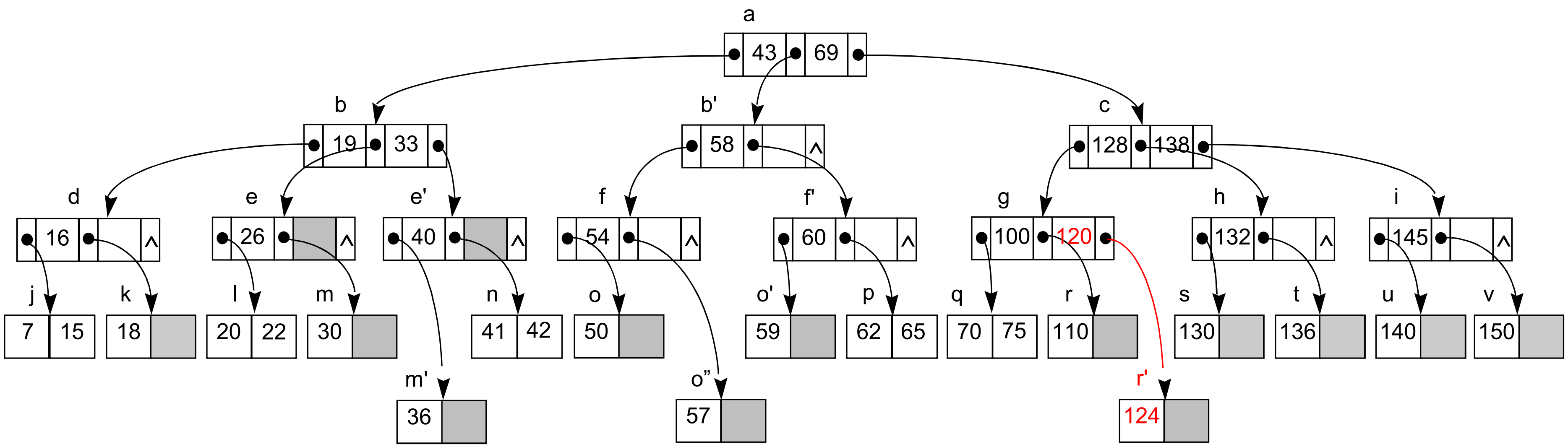
75 삽입


a
43 69
b
19 33
b'
58
c
128 138
d
16
e
26
e'
40
f
54
f'
60
g
100
h
132
i
145
j
7 15
k
18
l
20 22
m
30
n
41 42
o
50
o'
59
p
62 65
q
70
r
110 120
s
130
t
136
u
140
v
150
m'
36
o”
57
a
43 69
b
19 33
b'
58
c
128 138
d
16
e
26
e'
40
f
54
f'
60
g
100
h
132
i
145
j
7 15
k
18
l
20 22
m
30
n
41 42
o
50
o'
59
p
62 65
q
70 75
r
110 120
s
130
t
136
u
140
v
150
m'
36
o”
57

124 삽입

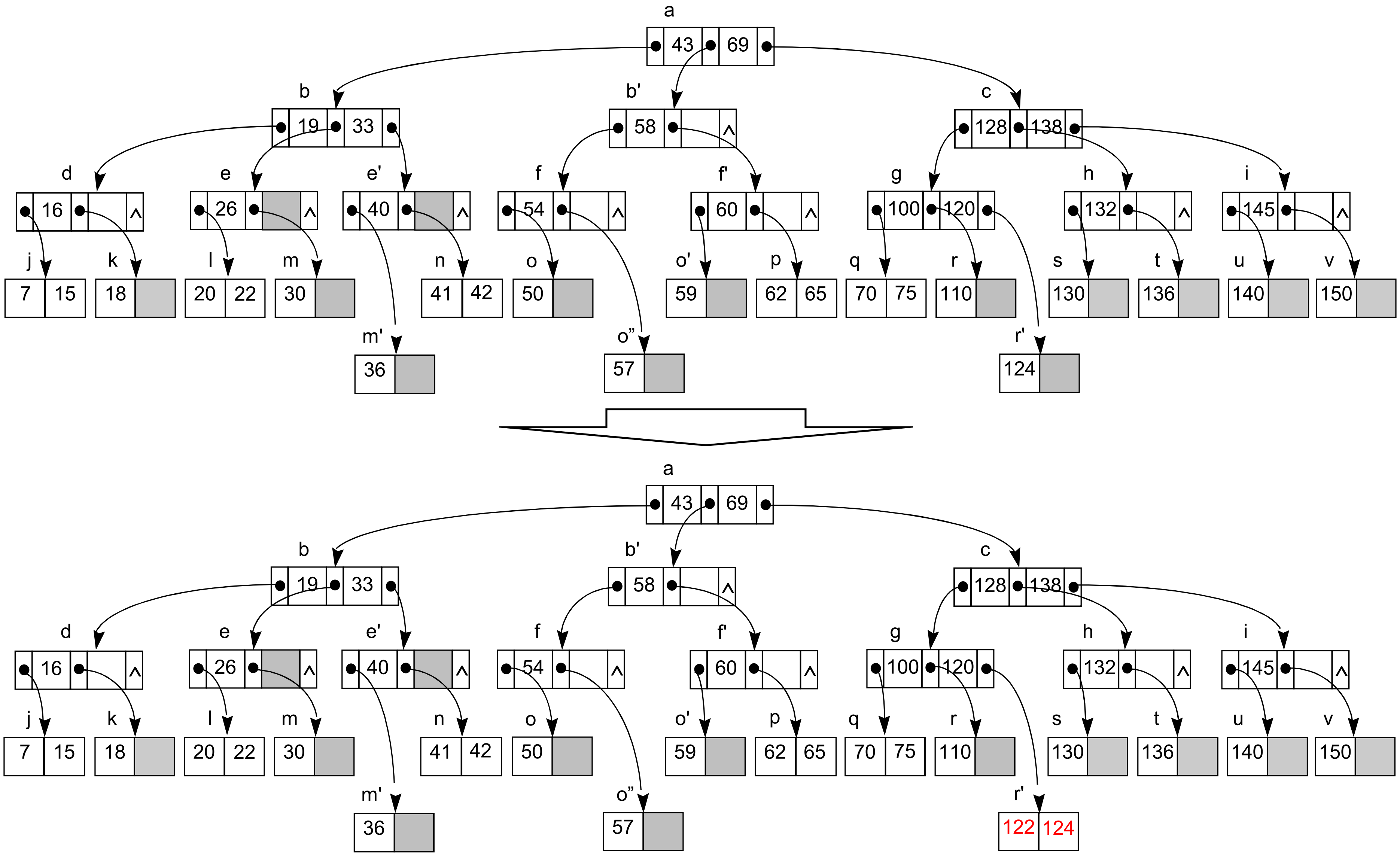


a
43 69
b
19 33
b'
58
c
128 138
d
16
e
26
e'
40
f
54
f'
60
g
100
h
132
i
145
j
7 15
k
18
l
20 22
m
30
n
41 42
o
50
o'
59
p
62 65
q
70 75
r
110 120
s
130
t
136
u
140
v
150
m'
36
o”
57



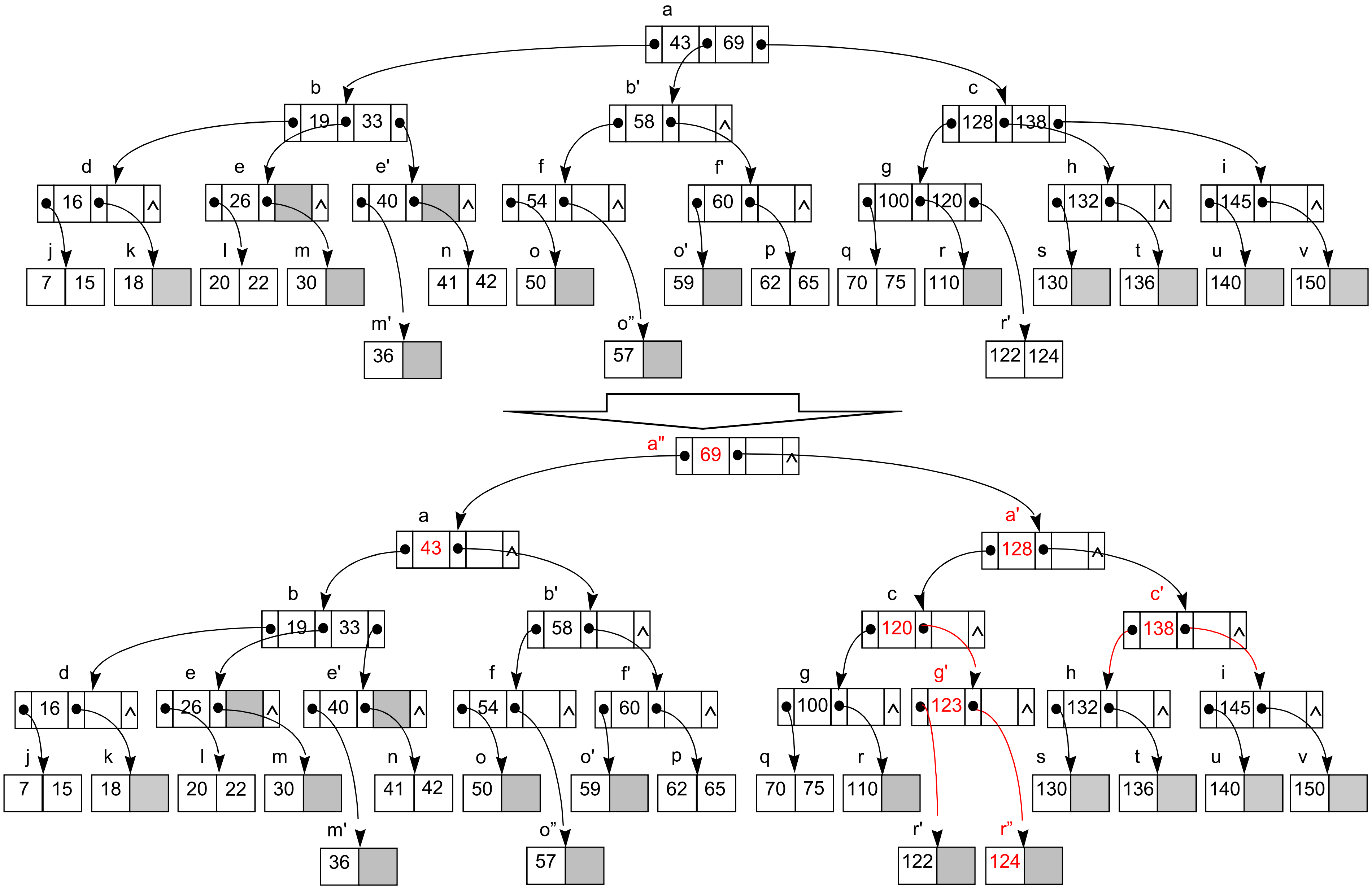
a
43 69
b
19 33
b'
58
c
128 138
d
16
e
26
e'
40
f
54
f'
60
g
100 120
h
132
i
145
j
7 15
k
18
l
20 22
m
30
n
41 42
o
50
o'
59
p
62 65
q
70 75
r
110
s
130
t
136
u
140
v
150
m'
36
o”
57
r'
124

122 삽입



a
43
69
b
19
33
b'
58
c
128
138
d
16
e
26
e'
40
f
54
f'
60
g
100
120
h
132
i
145
j
7
15
k
18
l
20
22
m
30
n
41
42
o
50
o'
59
p
62
65
q
70
75
r
110
s
130
t
136
u
140
v
150
m'
36
o”
57
r'
124
a
43
69
b
19
33
b'
58
c
128
138
d
16
e
26
e'
40
f
54
f'
60
g
100
120
h
132
i
145
j
7
15
k
18
l
20
22
m
30
n
41
42
o
50
o'
59
p
62
65
q
70
75
r
110
s
130
t
136
u
140
v
150
m'
36
o”
57
r'
122
124

123 삽입 (한 레벨 증가)


a
43
69
b
19
33
b'
58
c
128
138
d
16
e
26
e'
40
f
54
f'
60
g
100
120
h
132
i
145
j
7
15
k
18
l
20
22
m
30
n
41
42
o
50
o'
59
p
62
65
q
70
75
r
110
s
130
t
136
u
140
v
150
m'
36
o”
57
r'
122
124
a"
69
a
43
a'
128
b
19
33
b'
58
c
120
c'
138
d
16
e
26
e'
40
f
54
f'
60
g
100
g'
123
h
132
i
145
j
7
15
k
18
l
20
22
m
30
n
41
42
o
50
o'
59
p
62
65
q
70
75
r
110
s
130
t
136
u
140
v
150
m'
36
o”
57
r'
122
r”
124

▶ 삽입 알고리즘

```
/*  알고리즘에서 사용되는 변수는 다음과 같다.
In-key      :  B-트리에 삽입될 키
Finished  :  삽입이 완료되었음을 나타내는 플래그
Found      :  B-트리에서 레코드가 발견되었음을 나타내는 플래그
P             :  노드에 대한 포인터
TOOBIG              :  오버플로 노드를 위한 변수
N             :  키 카운터
*/
/*  노드의 주소를 스택에 저장하면서 In-key가 삽입될 위치를 탐색한다. */
Found  = false;
read root;
do {
      N  =  number of keys in current node;
      if (n-key == key in current node) found = true;
      else if (In-key < key1) P = Po;
      else if (In-key > keyN) P = PN;
      else P = Pi-1; /* for some i where keyi-1 < In-key < keyi */
      if (P != null)  {
         push onto stack address of current node;
         read node pointed to by P;
      }
} while (!Found && P is not null);
```

```
if (Found) report In-key already in tree;
else  { /* In-key를 B-트리에 삽입한다 */
      P = nil;
      Finished = false;
      do  {
            if (current node is not full) {
                put In-key in current node;
                /* 노드 안에서 키 순서를 유지하도록 키를 정렬한다 */
                Finished =  true;
            } else  {
                copy current node to TOOBIG;
                insert In-key and P into TOOBIG;
                In-key = center key of TOOBIG;
                current node = 1st half of TOOBIG;
                get space for new node, assign address to P;
                new node = 2nd half of TOOBIG;
                if (stack not empty) {
                    pop top of stack;
                    read node pointed to;
                } else  { /* 트리의 레벨이 하나 증가한다. */
                    get space for new node;
                    new node = pointer to old root, In-key and P;
                    Finished = true;
                }
            }
      }  while (!Finished);
}
```

▶ B-트리에서의 연산: 삭제

① 삭제될 키 값이 내부노드에 있는 경우: **underflow**가 발생하지 않는 경우

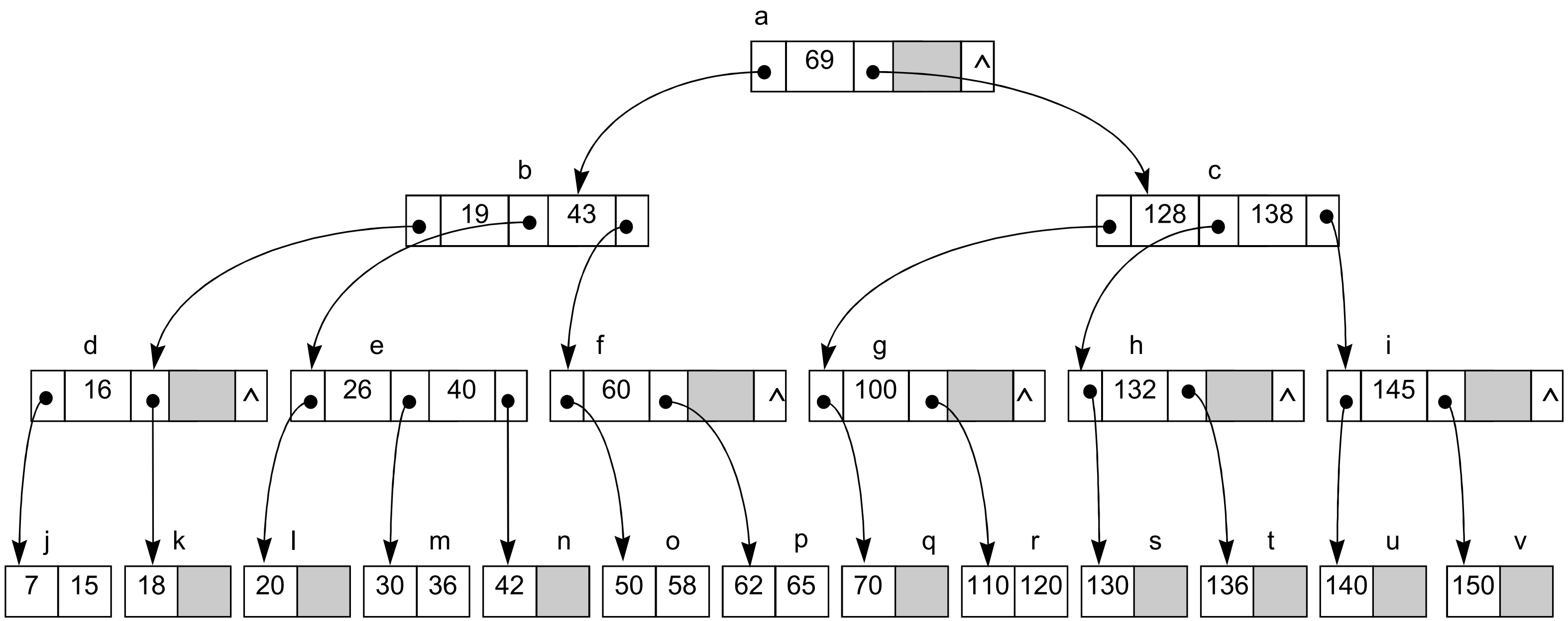
- 이 키 값의 "후행 키" 값과 교환 후 리프 노드에서 삭제
- 리프 노드에서의 삭제 연산이 더 간단
- 후행 키 값 대신 선행 키 값을 사용할 수 있다.

② 최소키 값 수($\lceil m/2 \rceil - 1$)보다 작은 경우 : **underflow** 발생

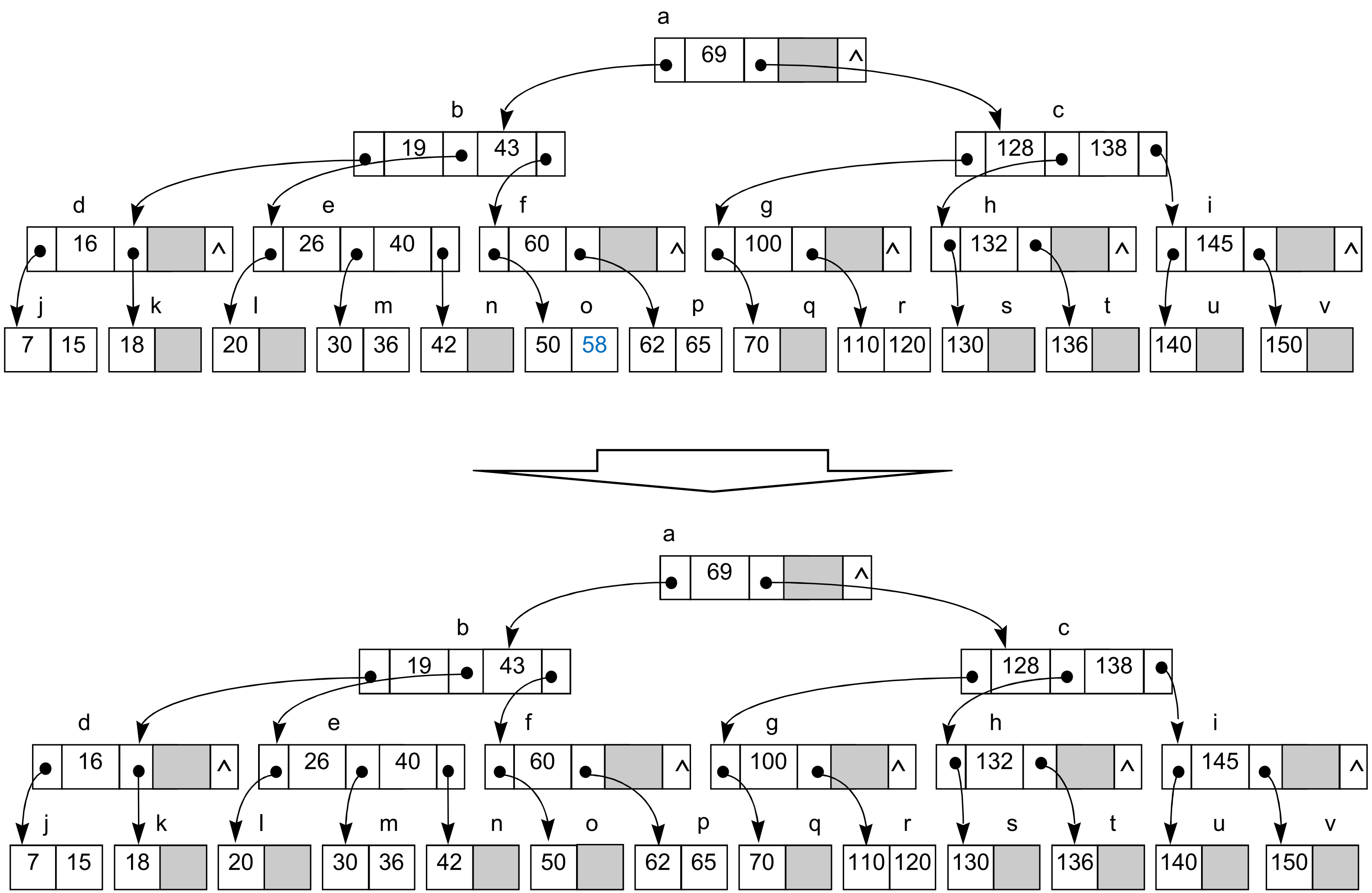
- 키 재분배(key redistribution)
  - 최소 키 값보다 많은 키를 가진 "형제 노드"에서 차출
  - 부모 노드에 있던 키 값을 해당 노드로 이동, 빈 자리에 차출된 형제 노드의 키 값을 이동
  - 트리 구조를 변경시키지 않음
- 노드 합병(node merge)
  - 재분배가 불가능한 경우에 적용
  - 형제 노드와 합치는 방법으로, 합병 결과 빈 노드는 제거
  - 이때 부모 노드의 키 값도 같이 합쳐짐
  - 트리 구조가 변경됨

예제 1

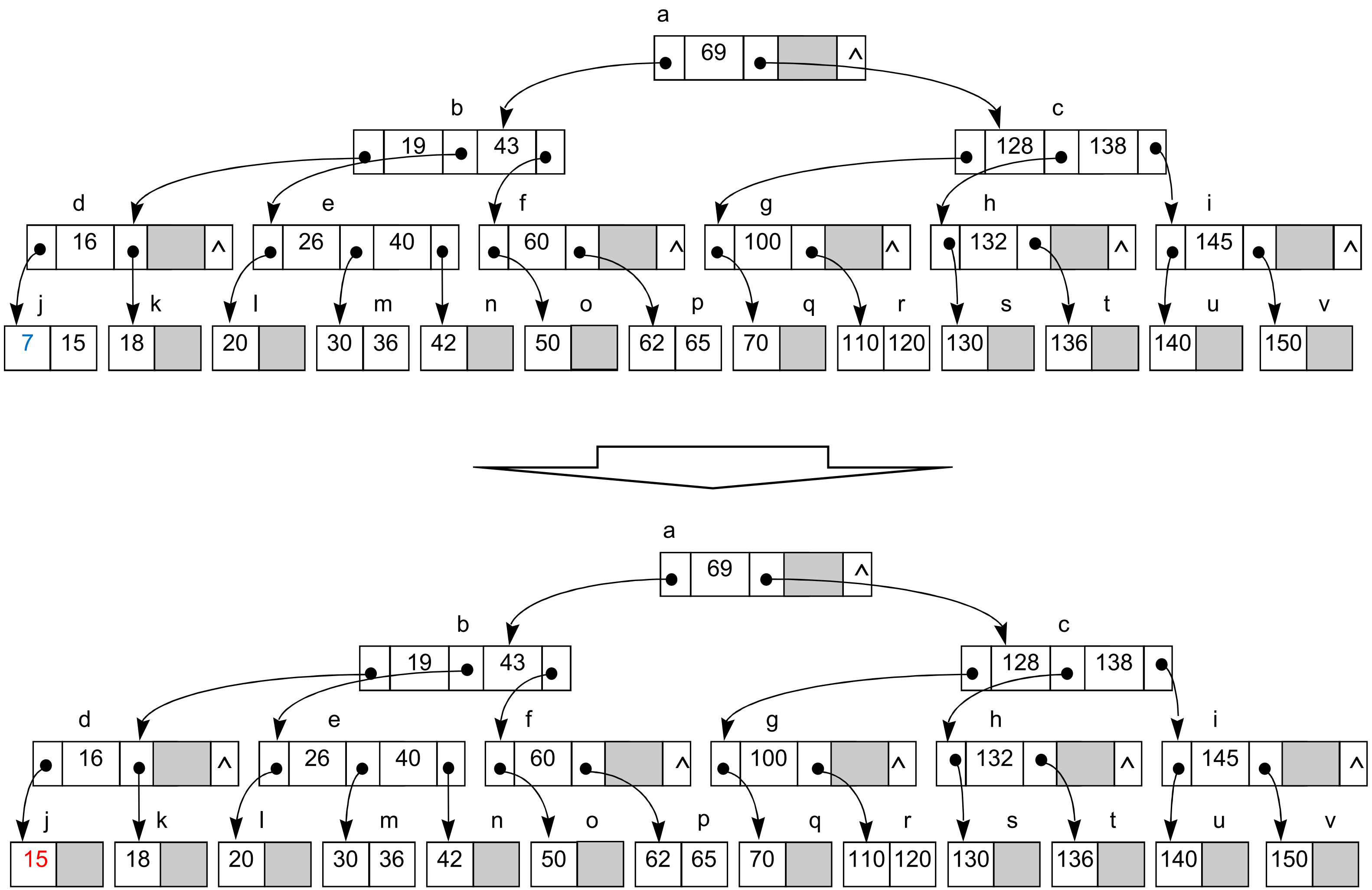
◆ 앞의 B-트리(그림 6-18)에서 키 값 58, 7, 60, 20, 15 , 36, 50, 16, 18, 130 삭제



58 삭제

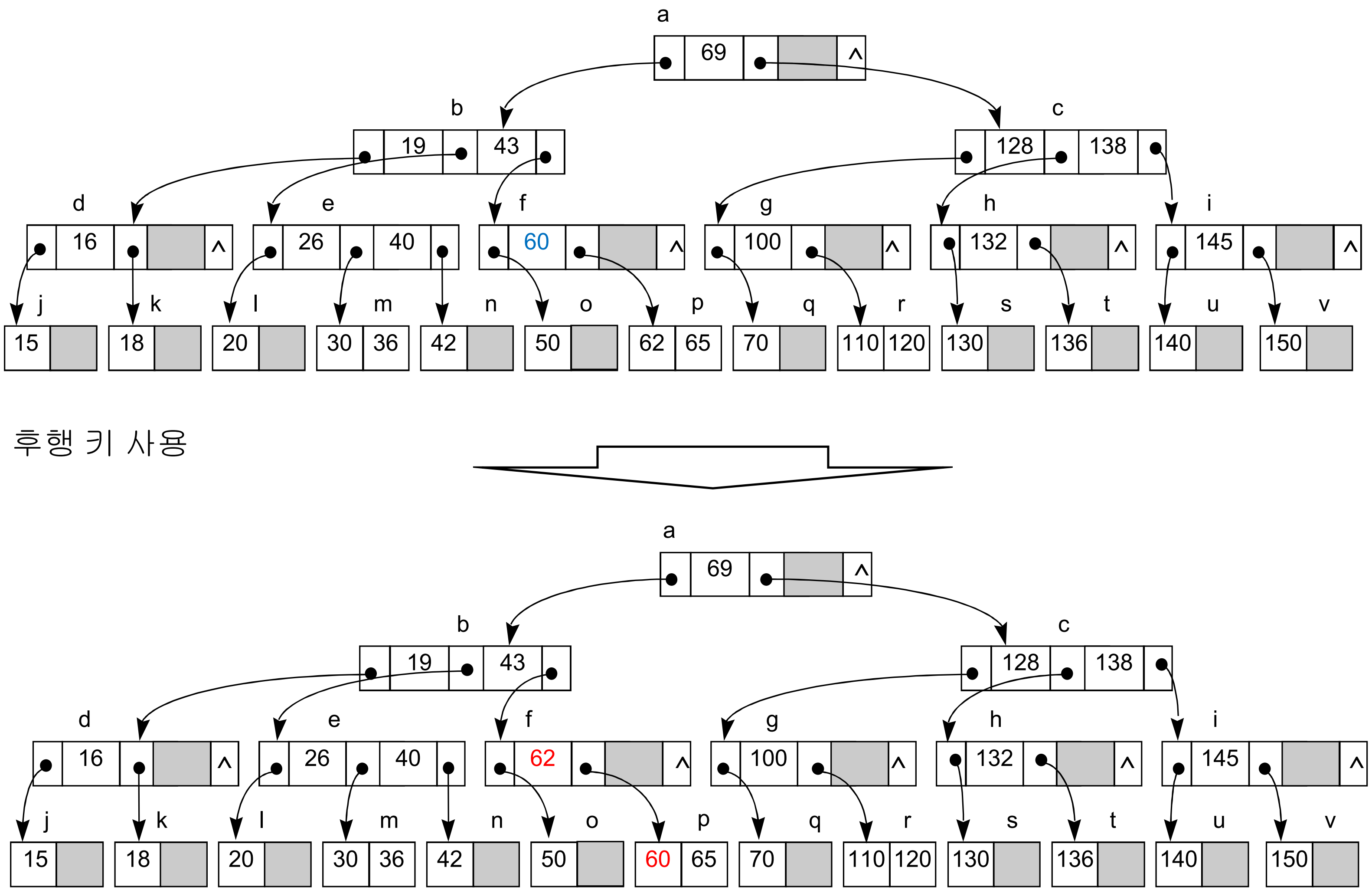

a
69
b
19
43
c
128
138
d
16
e
26
40
f
60
g
100
h
132
i
145
j
7
15
k
18
l
20
m
30
36
n
42
o
50
58
p
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150

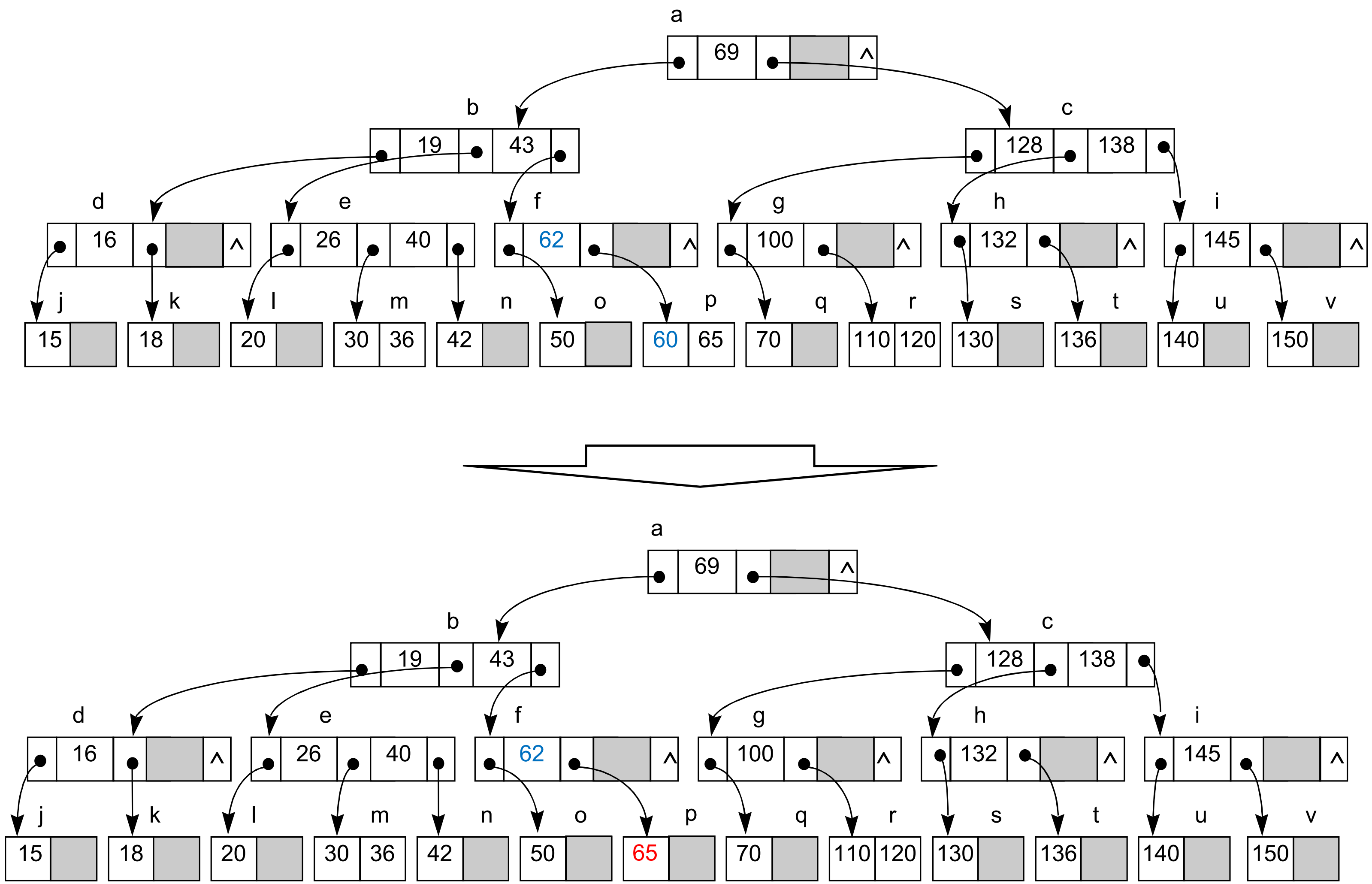
7 삭제



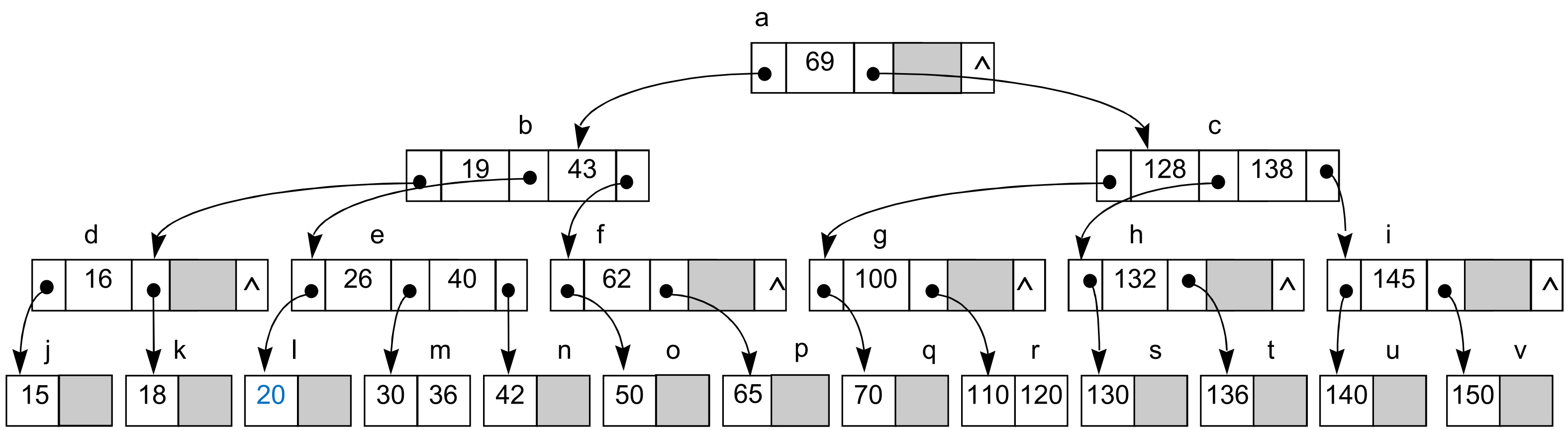
a
69
b
19 43
c
128 138
d e f g h i
16
26 40
60
100
132
145
j k l m n o p q r s t u v
7 15
18
20
30 36
42
50
62 65
70
110 120
130
136
140
150
a
69
b
19 43
c
128 138
d e f g h i
16
26 40
60
100
132
145
j k l m n o p q r s t u v
15
18
20
30 36
42
50
62 65
70
110 120
130
136
140
150

60 삭제

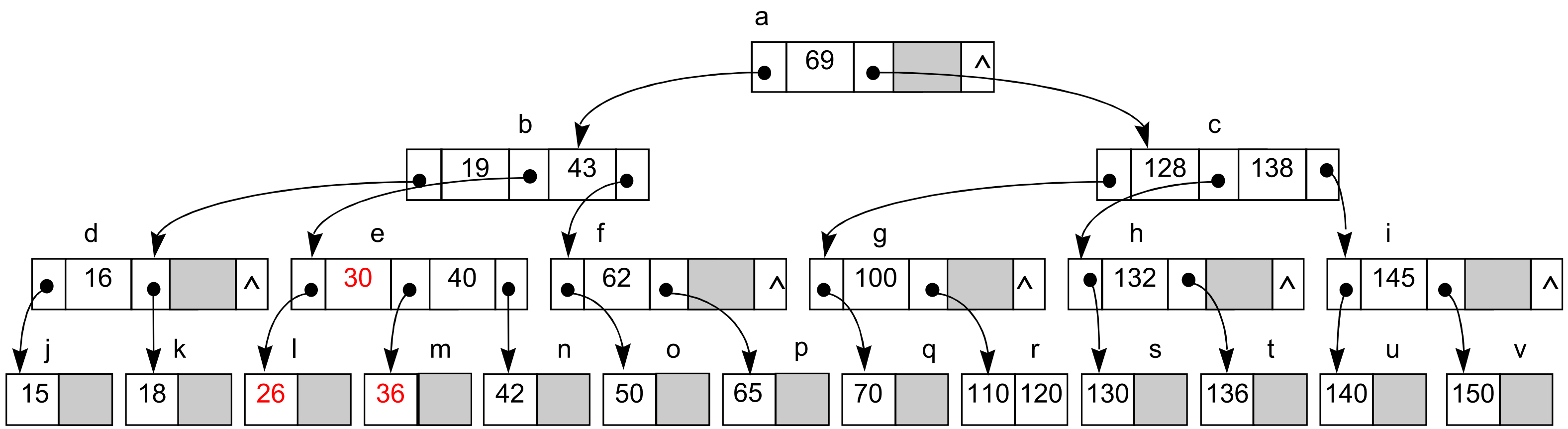

a
69
b
19
43
c
128
138
d
16
e
26
40
f
60
g
100
h
132
i
145
j
k
l
m
n
o
p
q
r
s
t
u
v
15
18
20
30
36
42
50
62
65
70
110
120
130
136
140
150
후행 키 사용
a
69
b
19
43
c
128
138
d
16
e
26
40
f
62
g
100
h
132
i
145
j
k
l
m
n
o
p
q
r
s
t
u
v
15
18
20
30
36
42
50
60
65
70
110
120
130
136
140
150


a
69
^
b
19
43
c
128
138
d
16
^
e
26
40
f
62
^
g
100
^
h
132
^
i
145
^
j
k
l
m
n
o
p
q
r
s
t
u
v
15
18
20
30
36
42
50
60
65
70
110
120
130
136
140
150
a
69
^
b
19
43
c
128
138
d
16
^
e
26
40
f
62
^
g
100
^
h
132
^
i
145
^
j
k
l
m
n
o
p
q
r
s
t
u
v
15
18
20
30
36
42
50
65
70
110
120
130
136
140
150

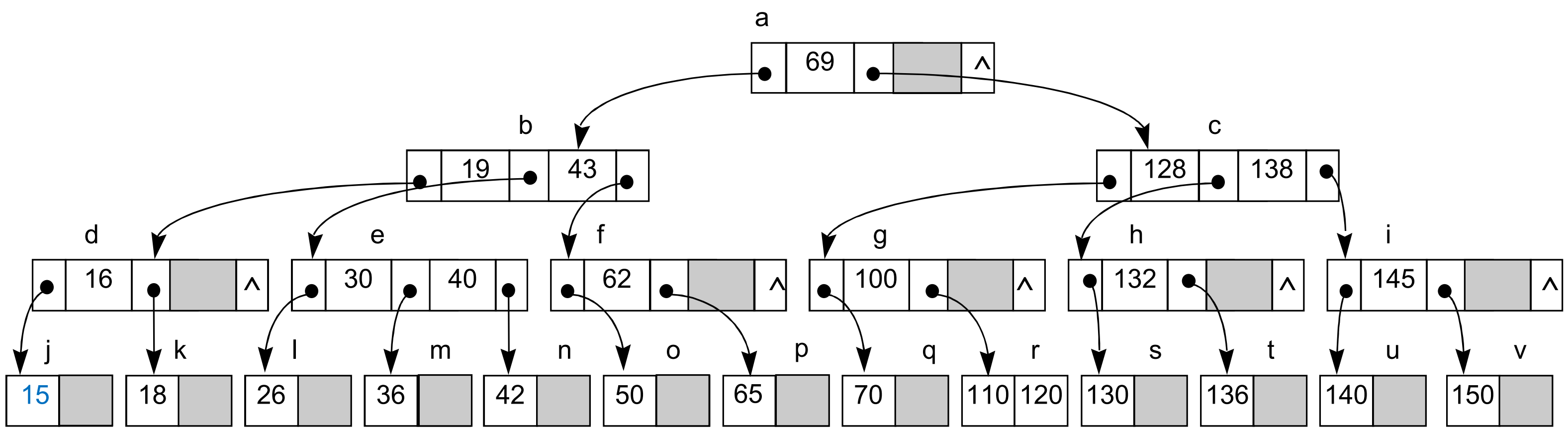
20 삭제



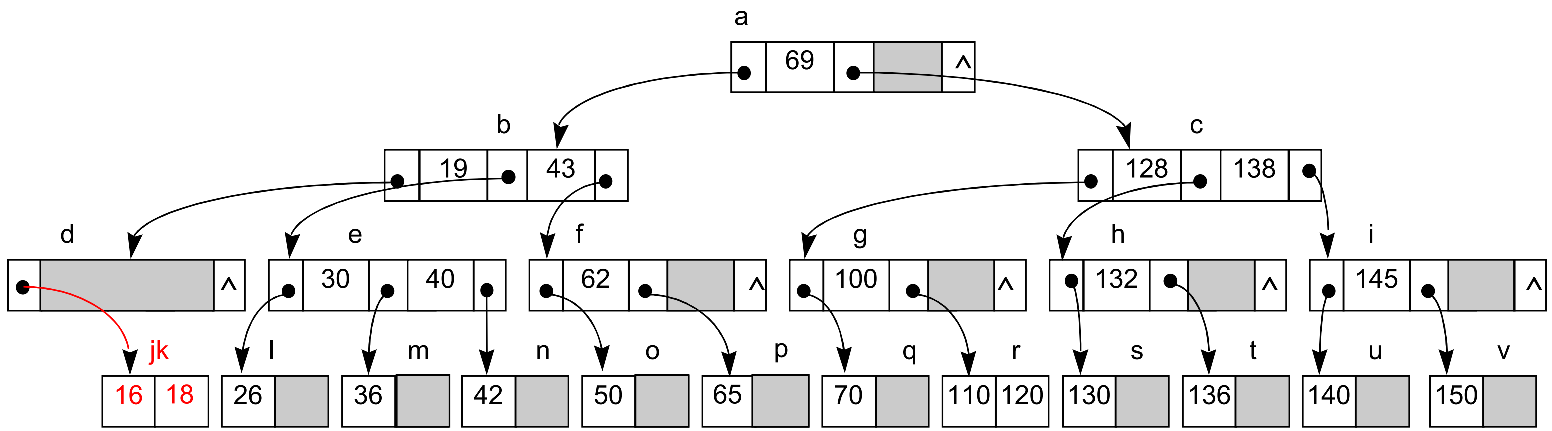
키 재분배(형제 노드 사용)

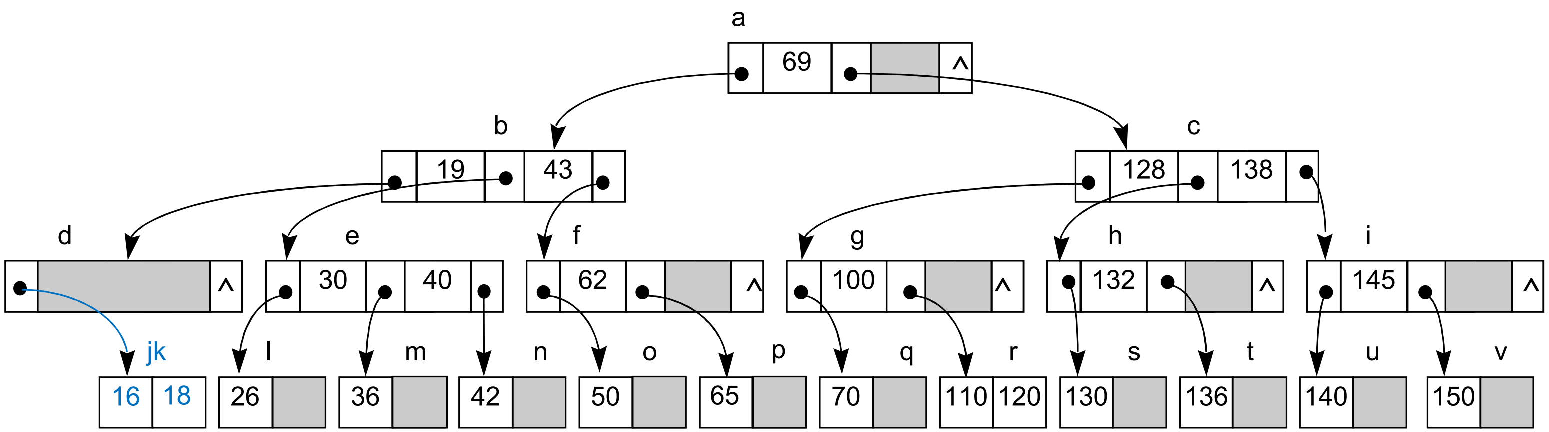


15 삭제

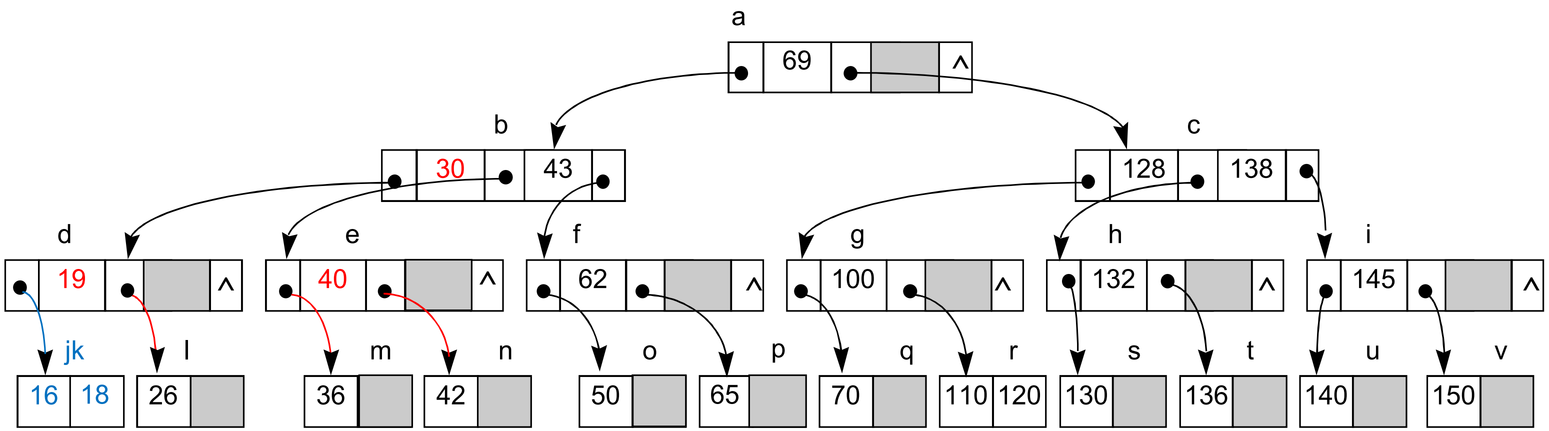


노드 합병: 후행키 및 형제 노드
모두 사용이 불가능함

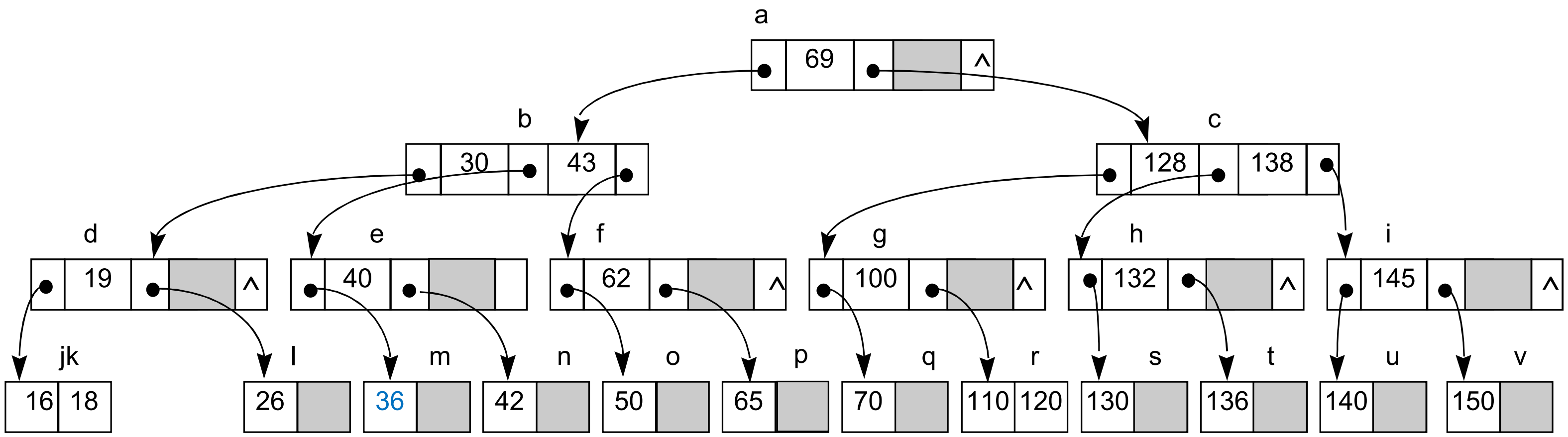



a
69
b
19
43
c
128
138
d
e
30
40
f
62
g
100
h
132
i
145
jk
16
18
l
26
m
36
n
42
o
50
p
65
q
70
r
110
120
s
130
t
136
u
140
v
150

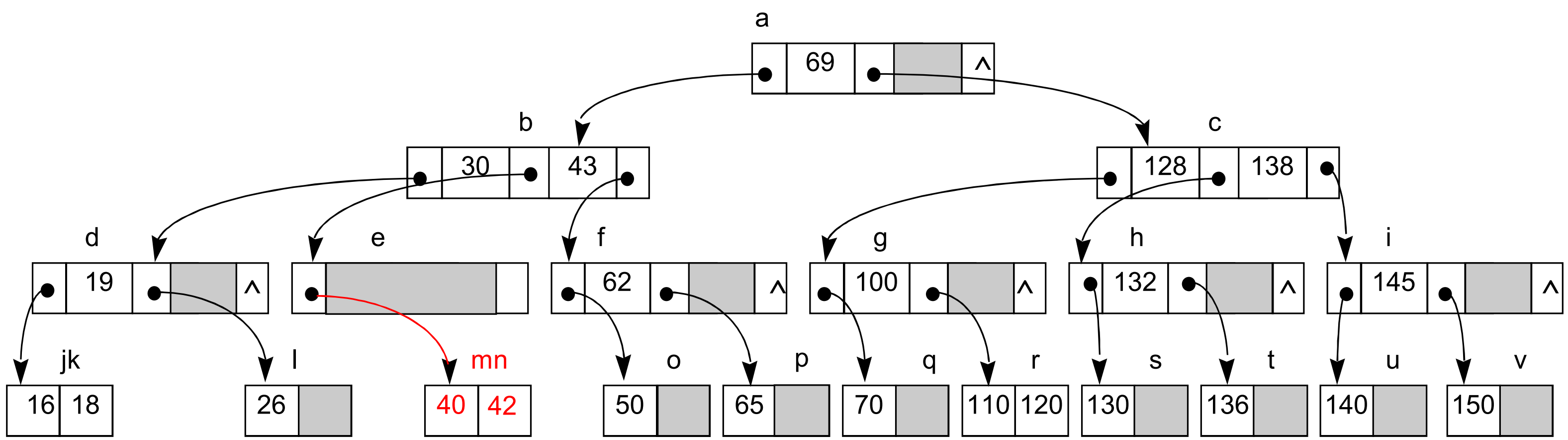
키 재분배(형제 노드 사용)

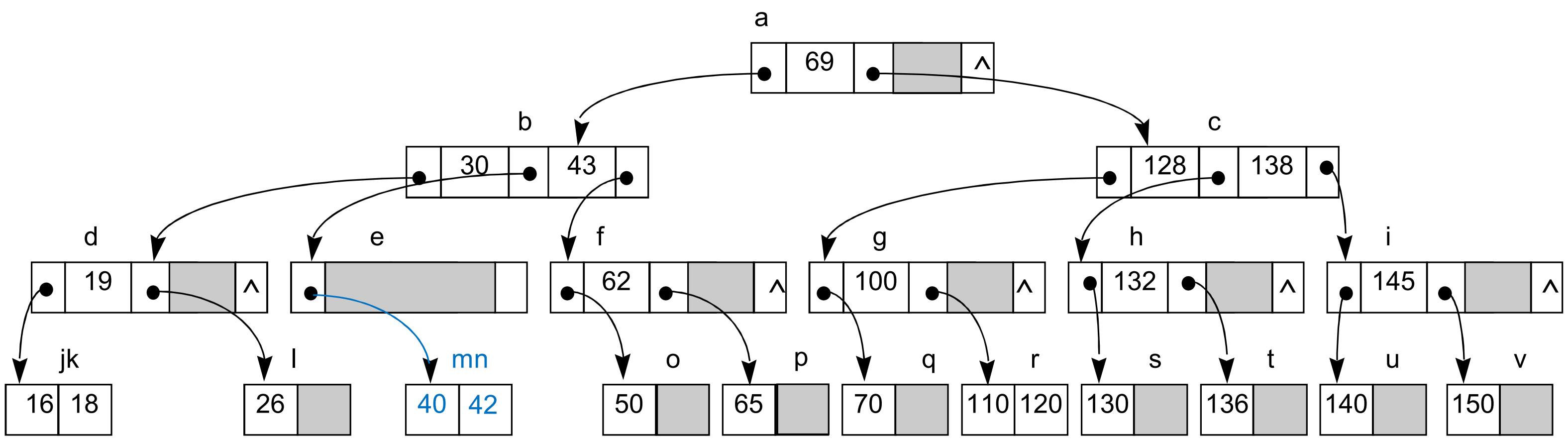

a
69
b
30
43
c
128
138
d
19
e
40
f
62
g
100
h
132
i
145
jk
16
18
l
26
m
36
n
42
o
50
p
65
q
70
r
110
120
s
130
t
136
u
140
v
150

36 삭제

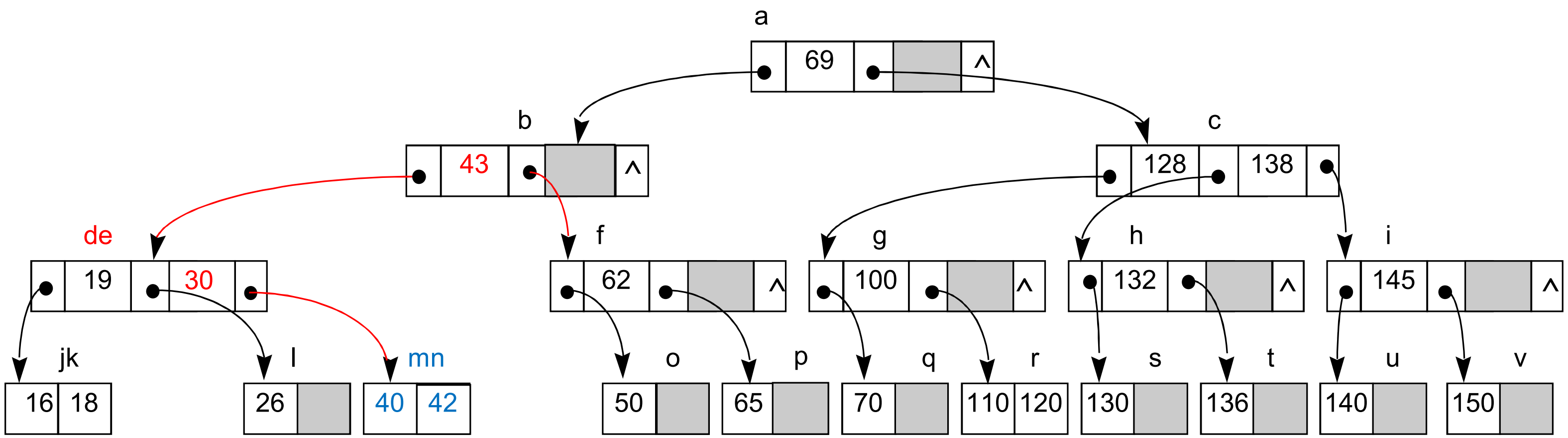


노드 합병: 후행키 및 형제 노드
모두 사용이 불가능함

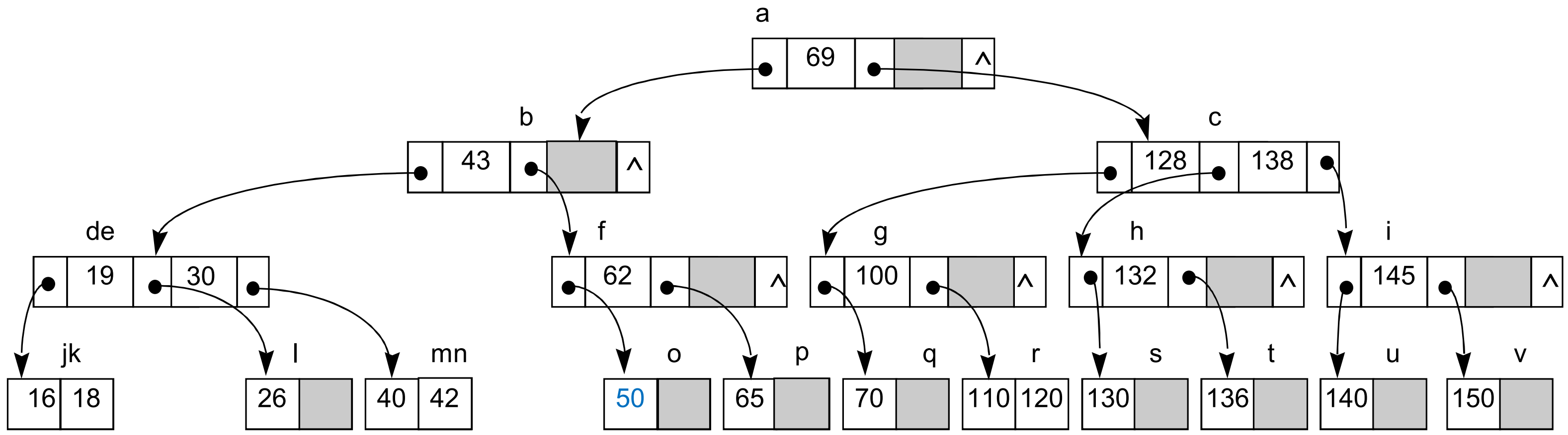




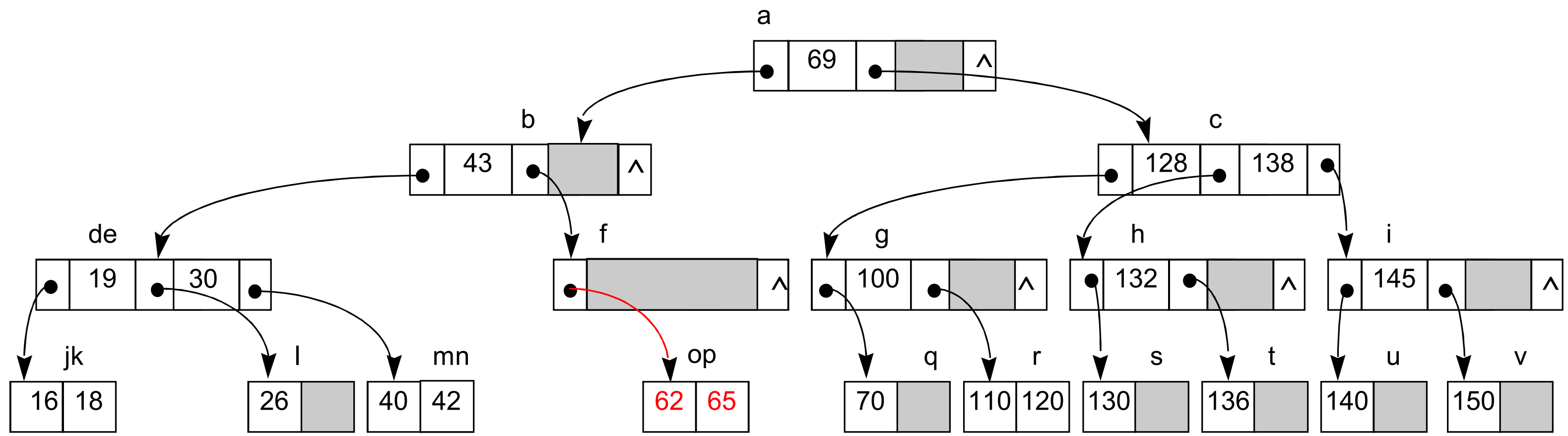
노드 합병: 후행키 및 형제 노드
모두 사용이 불가능함

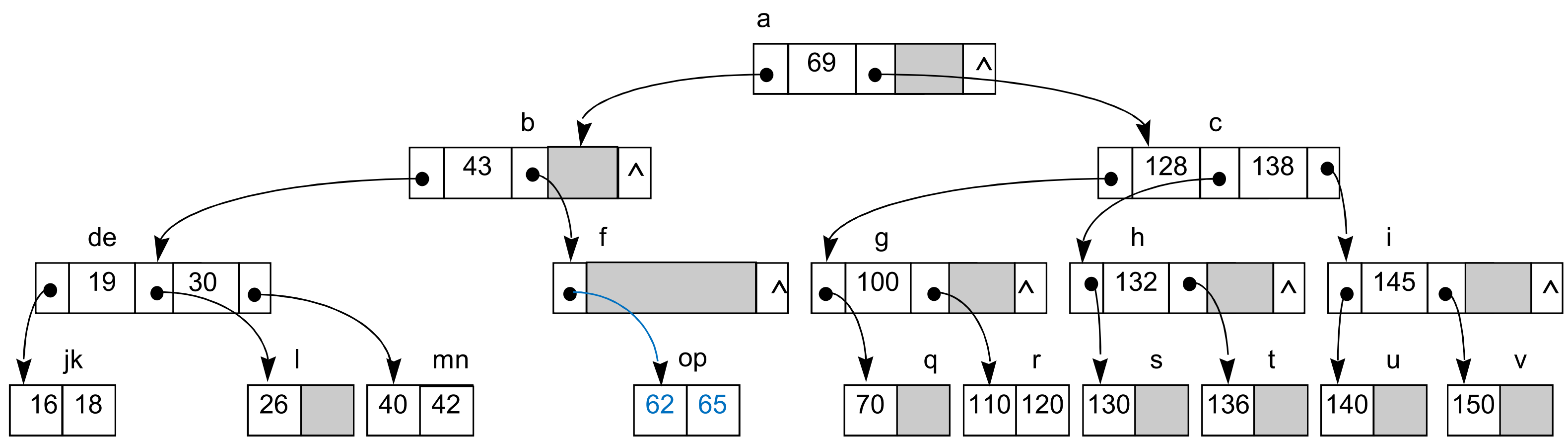


50 삭제

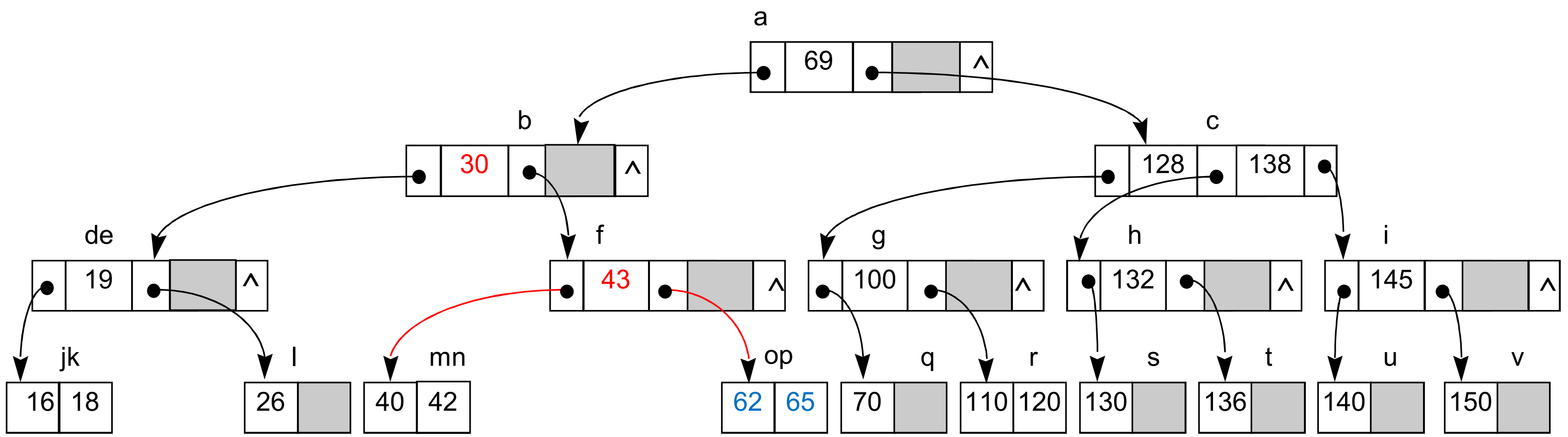


노드 합병: 후행키 및 형제 노드
모두 사용이 불가능함

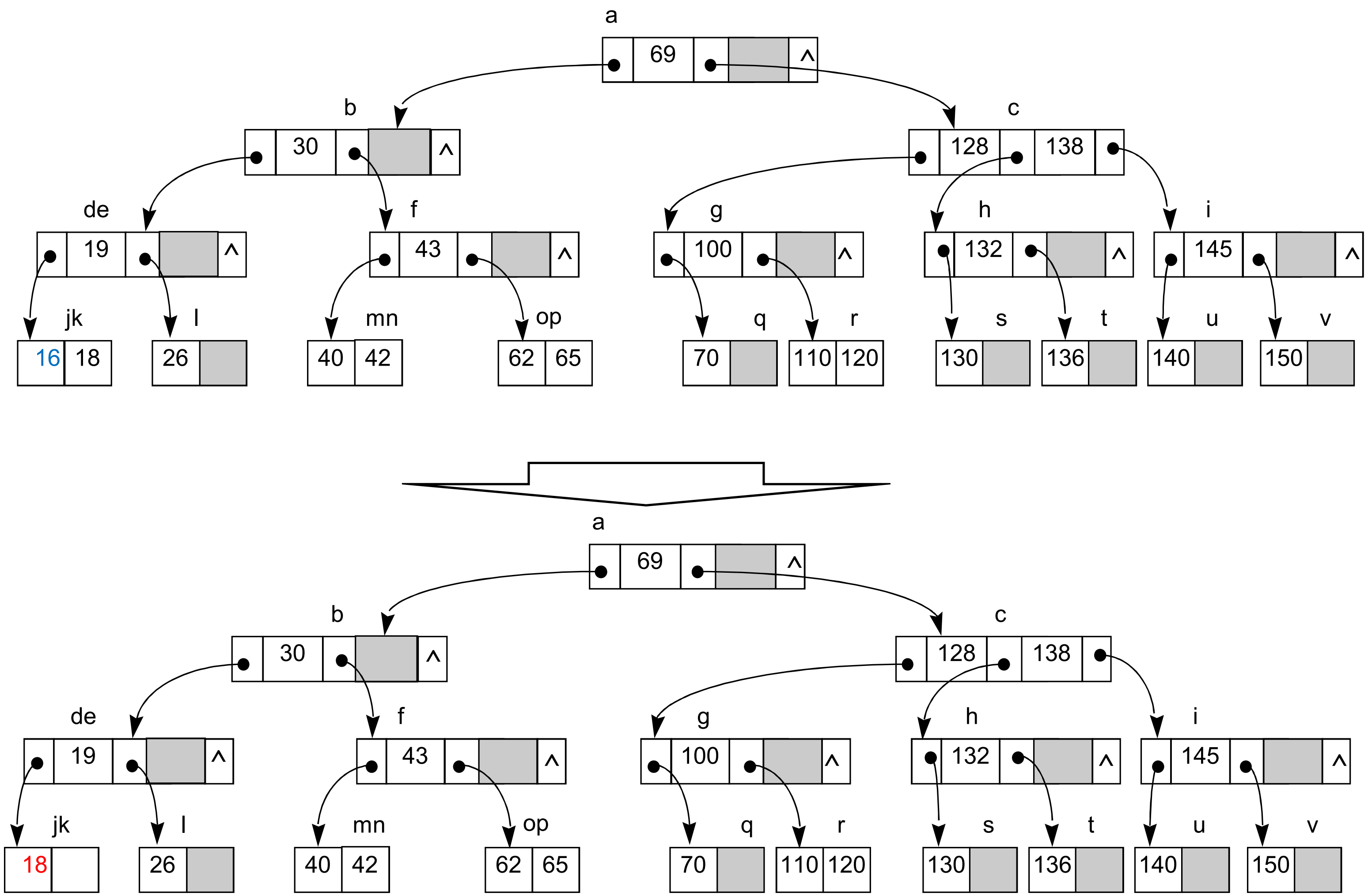



a
69
b
43
c
128
138
de
19
30
f
g
100
h
132
i
145
jk
16
18
l
26
mn
40
42
op
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150

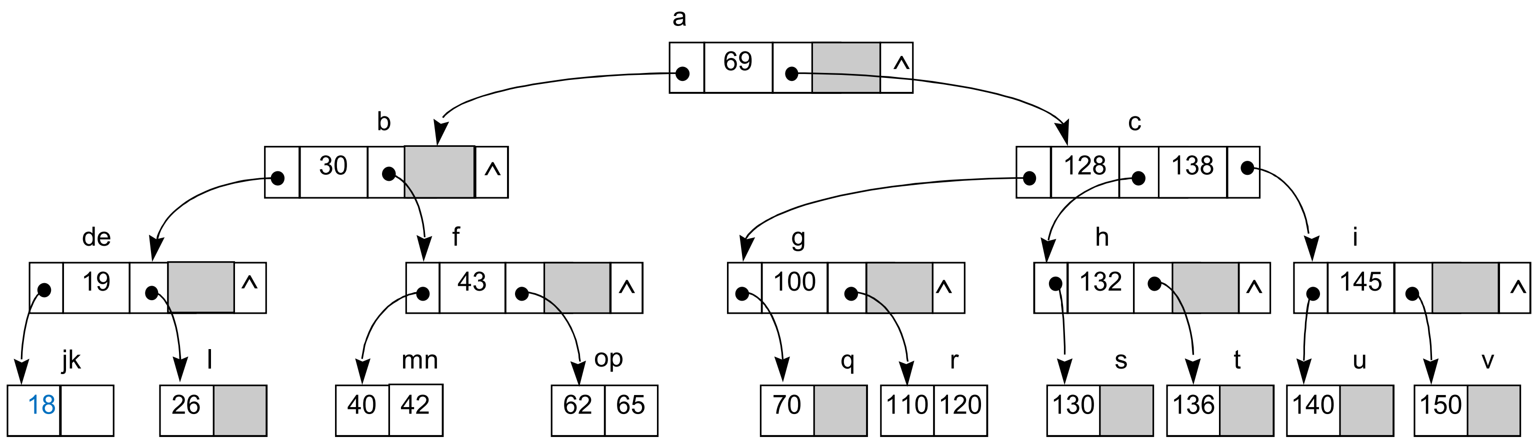
키 재분배(형제 노드 사용)


a
69
b
30
c
128
138
de
19
f
43
g
100
h
132
i
145
jk
16
18
l
26
mn
40
42
op
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150

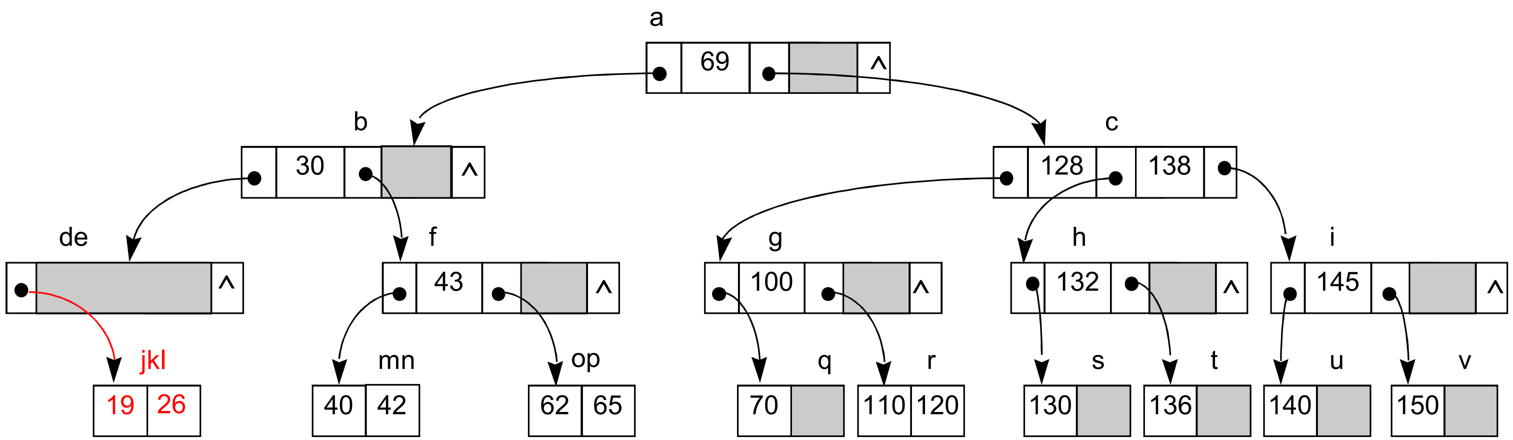
16 삭제

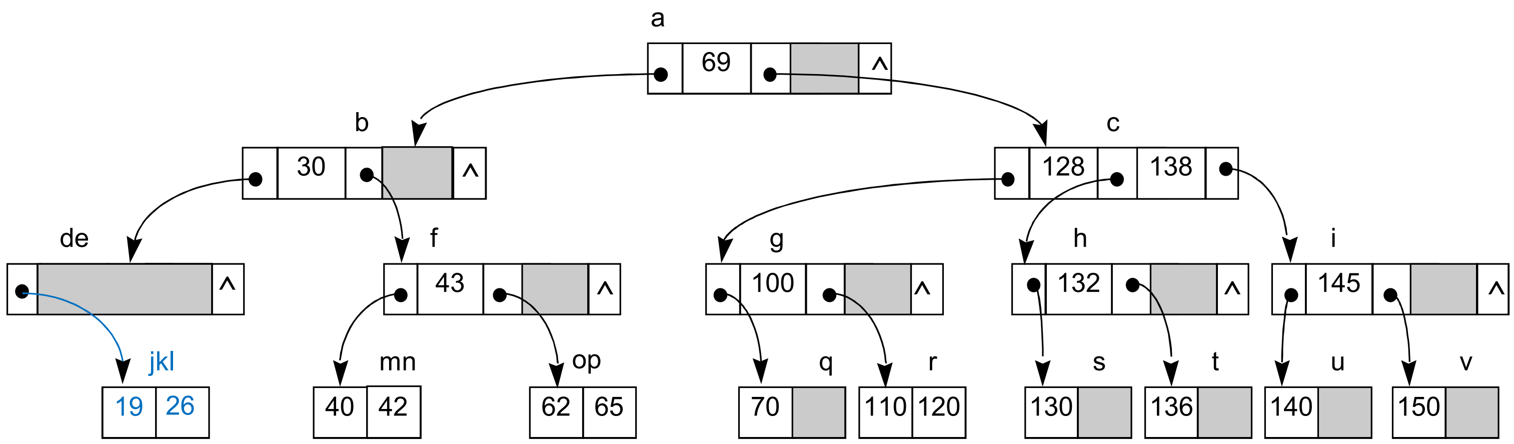

a
69
b
30
c
128
138
de
19
f
43
g
100
h
132
i
145
jk
16
18
l
26
mn
40
42
op
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150
a
69
b
30
c
128
138
de
19
f
43
g
100
h
132
i
145
jk
18
l
26
mn
40
42
op
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150

18 삭제

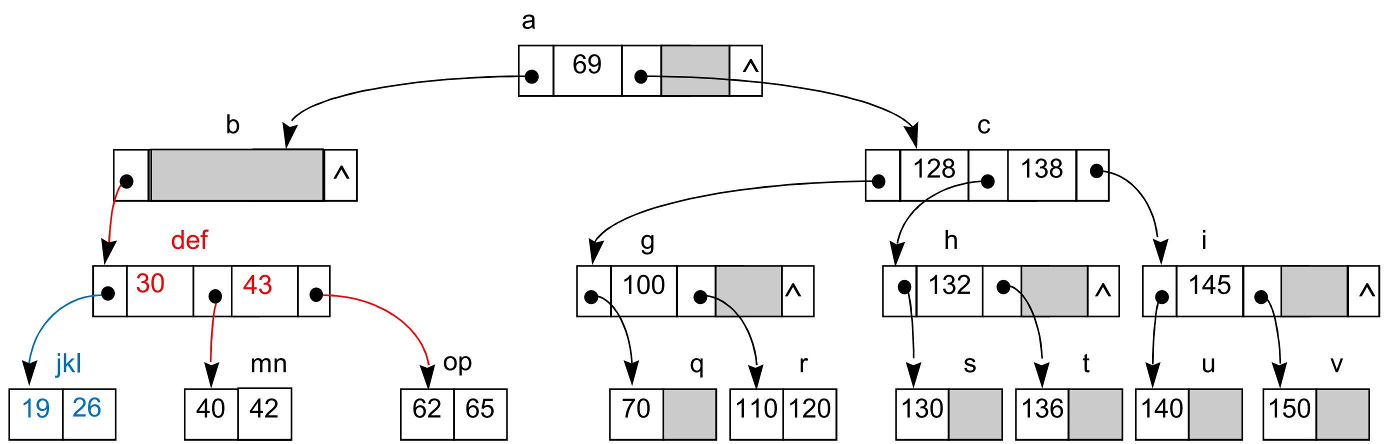


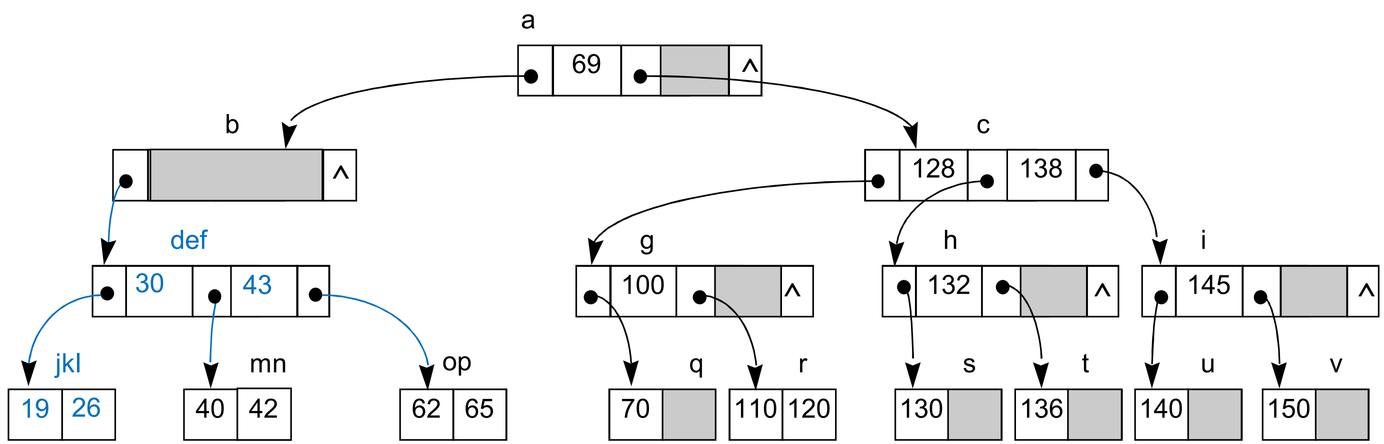
노드 합병: 후행키 및 형제 노드
모두 사용이 불가능함



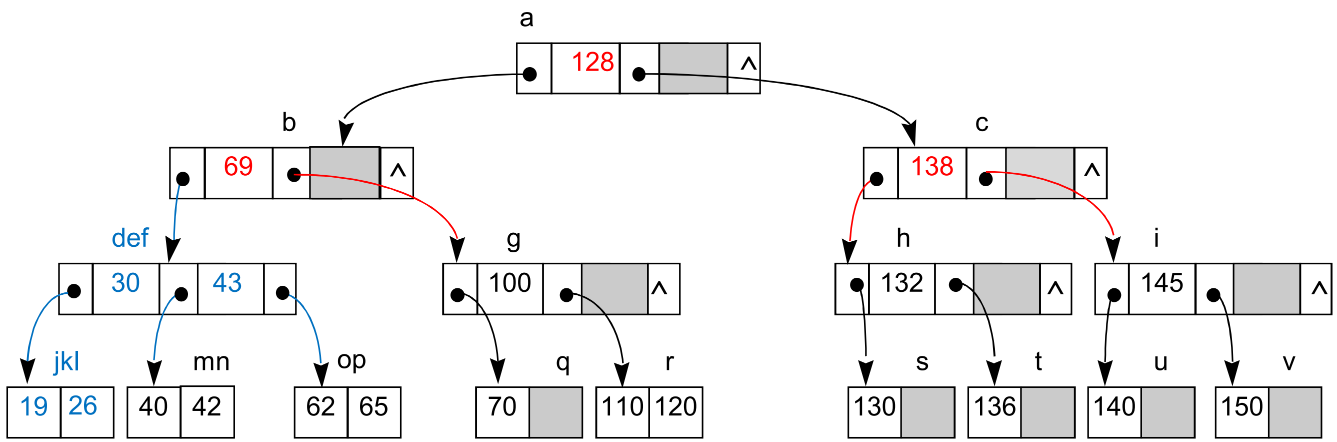


노드 합병: 후행키 및 형제 노드
모두 사용이 불가능함

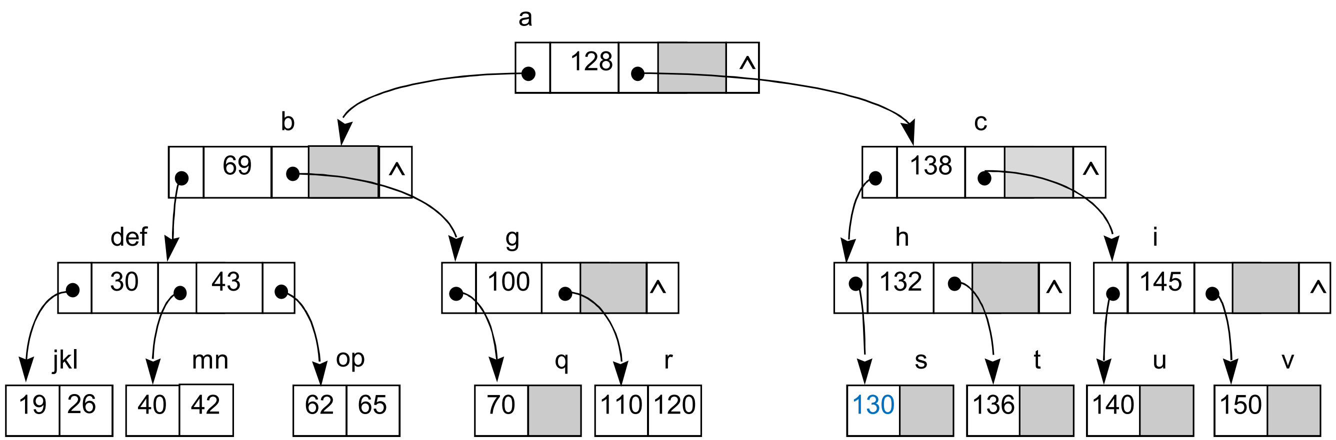



a
69
b
c
128
138
def
30
43
g
100
h
132
i
145
jkl
19
26
mn
40
42
op
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150

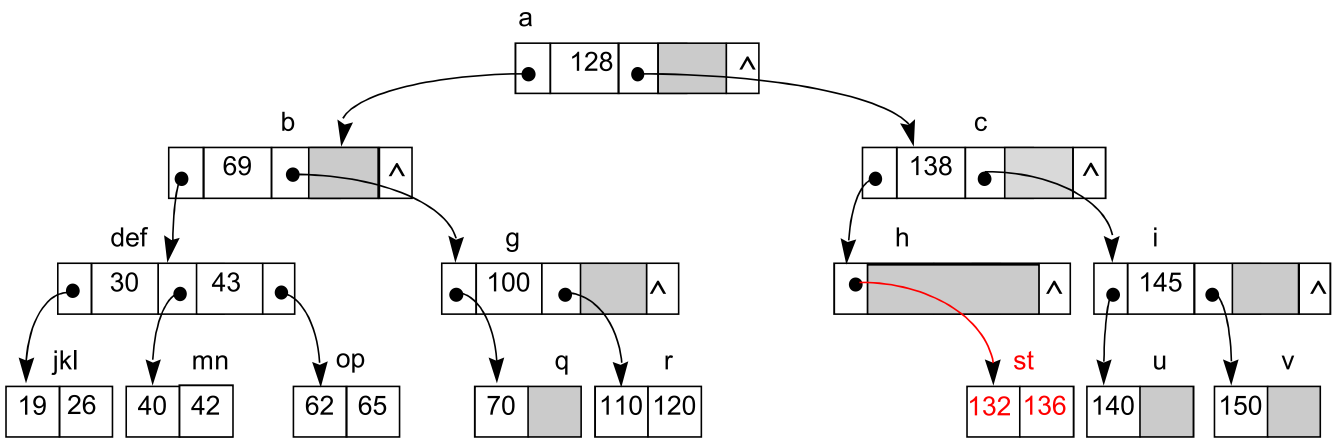
키 재분배(형제 노드 사용)

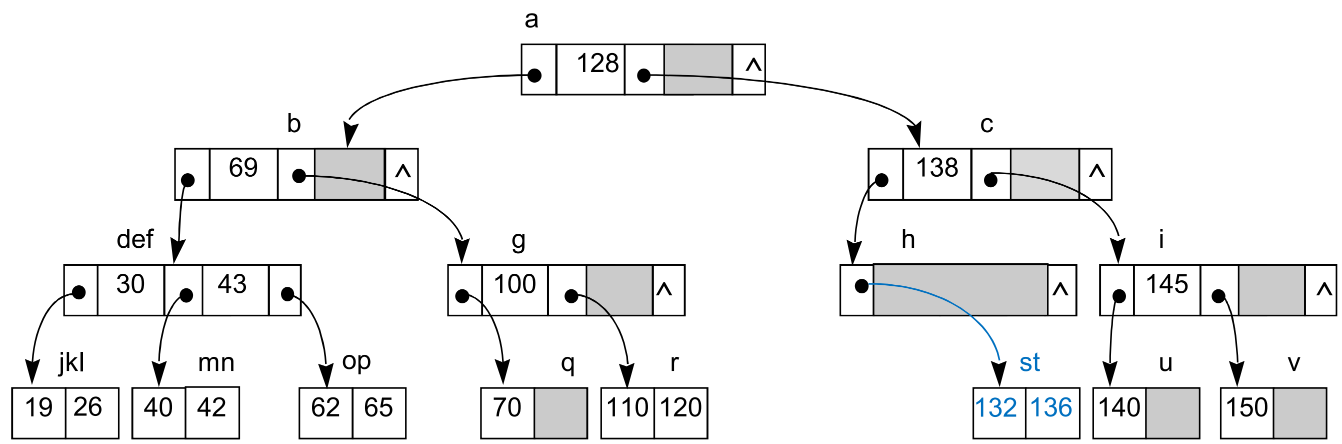

a
128
b
69
c
138
def
30
43
g
100
h
132
i
145
jkl
19
26
mn
40
42
op
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150

130 삭제

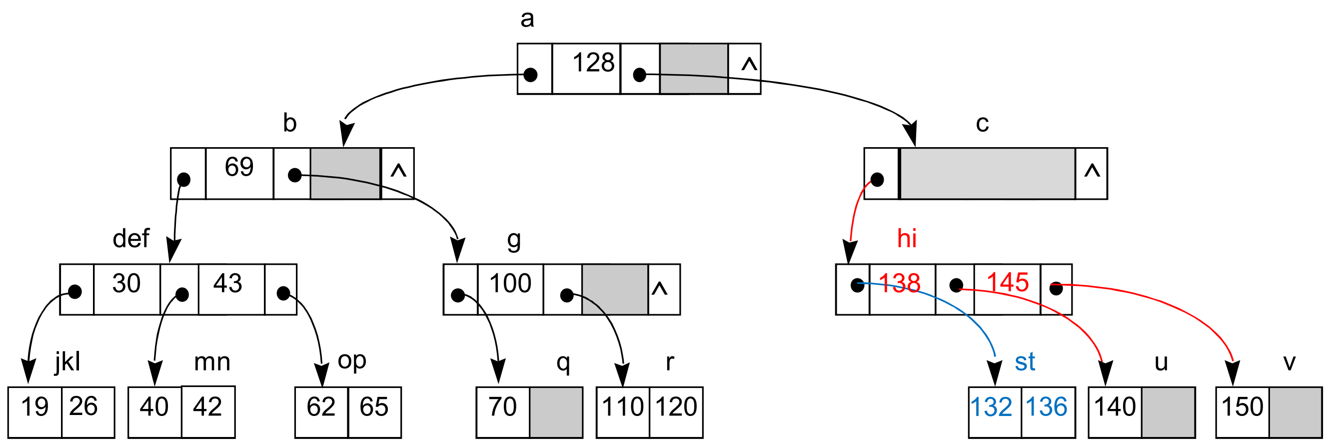


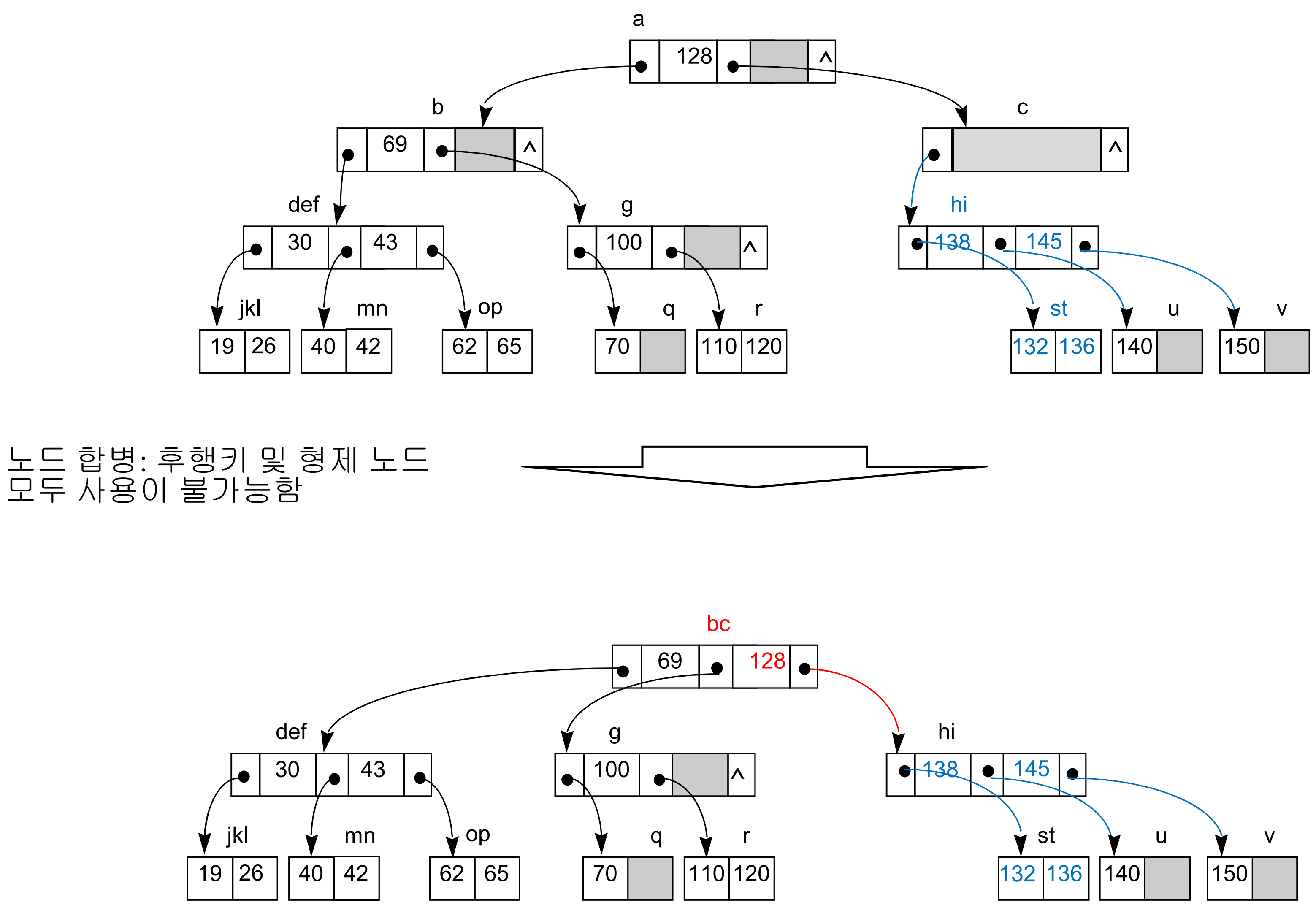
노드 합병: 후행키 및 형제 노드
모두 사용이 불가능함





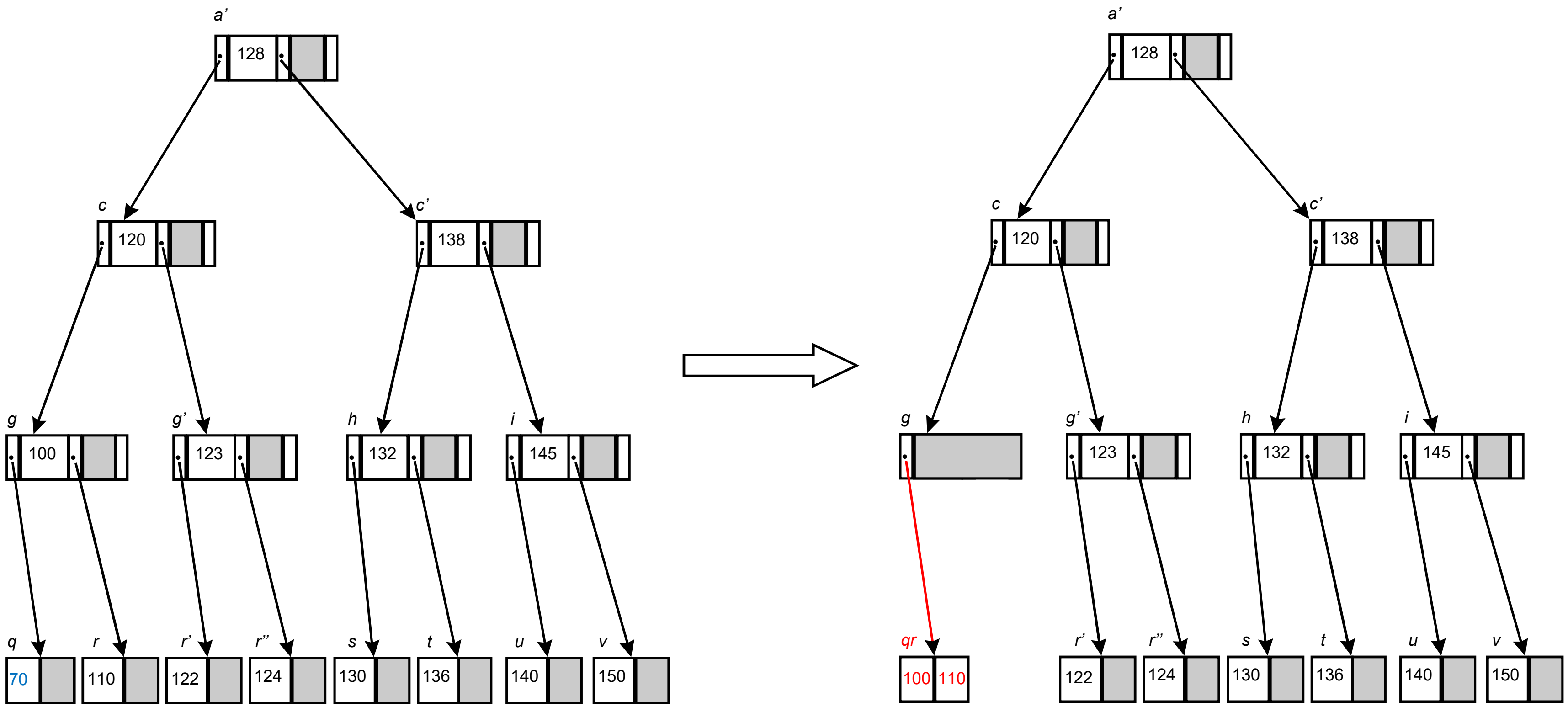
노드 합병: 후행키 및 형제 노드
모두 사용이 불가능함



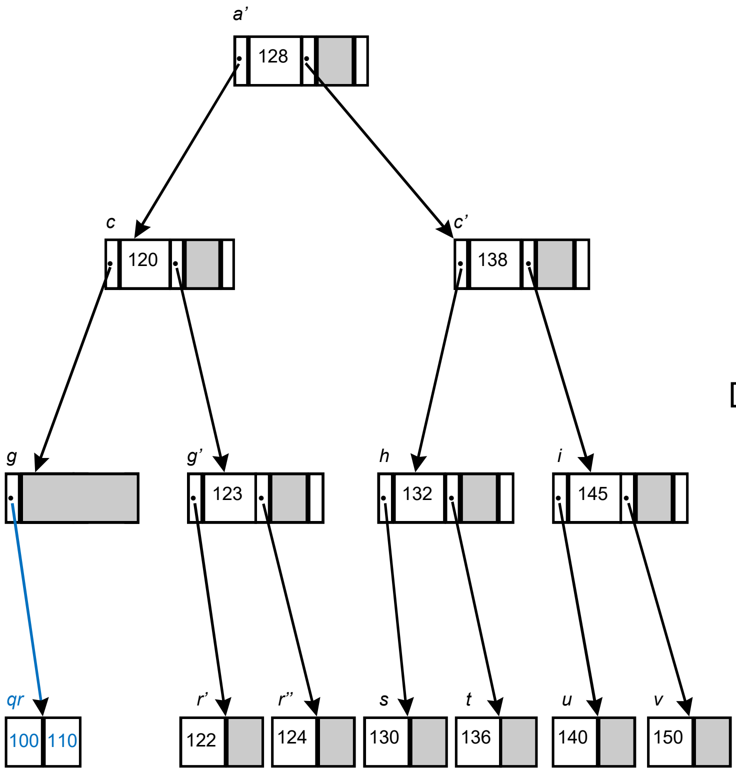


a
128
b
69
c
def
30
43
g
100
hi
138
145
jkl
19
26
mn
40
42
op
62
65
q
70
r
110
120
st
132
136
u
140
v
150
노드 합병: 후행키 및 형제 노드
모두 사용이 불가능함
bc
69
128
def
30
43
g
100
hi
138
145
jkl
19
26
mn
40
42
op
62
65
q
70
r
110
120
st
132
136
u
140
v
150

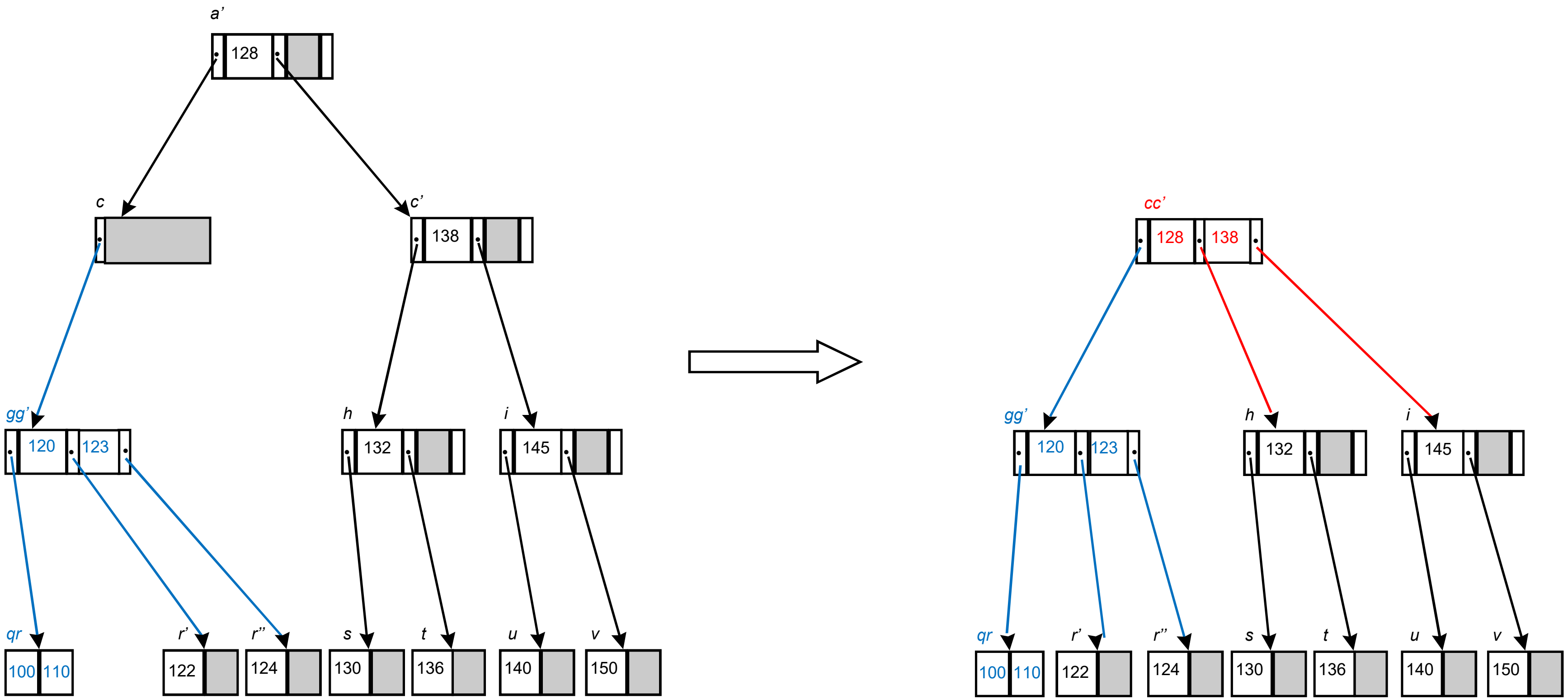
예제 2 : 70 삭제



노드 합병
(후행키, 형제노드 사용 불가능)



노드 합병
(후행키, 형제노드 사용 불가능)



노드 합병
(후행키, 형제노드 사용 불가능)

루트 레벨 하나 감소

▶ B-트리 삭제 알고리즘

```
/* 알고리즘에서 사용된 변수는 다음과 같다.
   Finished        : 삭제가 완료되었음을 나타내는 플래그
   TWOBNODE : 재분배를 위해 사용되는 정상 노드 보다 50% 큰
   노드
   A-sibling       : 인접 형제 노드
   Out-key         : B-트리에서 삭제될 키
*/

search tree for Out-key forming stack of node addresses;
/* 자세한 것은 그림 6.22의 삽입 알고리즘 참조 */
if (Out-key is not in terminal node) {
        search for successor key of Out-key at terminal level (stacking node
        addresses);
    copy successor over Out-key;
    terminal node successor now becomes the Out-key;
}

/* 키를 삭제하고 트리를 재조정한다. */
Finished = false;
```

```
do {
      remove Out-key
      if (current node is root or is not too small)
         Finished = true;
      else if (redistribution possible)  {
      /* A-sibling > minimum이 성립할 때 재분배 가능 */
         /* 재분배 실행 */
         copy "best" A-sibling, intermediate parent key, and
         current (too-small) node into TWOBNODE;
         copy keys and pointers from TWOBNODE to "best"
            A-sibling, parent, and current node so A-sibling and
            current node are roughly equal size;
         Finished = true;
      } else { /* 적당한 A-sibling과 합병한다 */
         choose best A-sibling to concatenate with;
         put in the leftmost of the current node and A-sibling the
            contents of both nodes and the intermediate key
            from the parent;
         discard rightmost of the two nodes;
         intermediate key in parent now becomes Out-key;
      }
} while (!Finished);
if (no keys in root) {
   /* 트리의 레벨이 하나 감소한다 */
   new root is the node pointed to by the current root;
   discard old root;
}
```

▶ m-원 B-트리의 성능

- 높이가 h인 m-원 B-트리
  - B-트리의 높이가 최대가 되는 때는 모든 노드의 분기율이 최소값인 $\left\lceil \frac{m}{2} \right\rceil$인 경우임. 따라서
  - B-트리의 높이: $\lceil \log_m(N+1) \rceil \le h \le \left\lceil \log_{\lceil \frac{m}{2} \rceil}(N+1) \right\rceil$

- **BST 및 MST 계열 트리의 높이 비교**
  - BST: $\lceil \log_2(N+1) \rceil \le h \le N$
  - AVL 트리: $\lceil \log_2(N+1) \rceil \le h \le \lceil 1.44 * \log_2(N+2) \rceil$
  - MST: $\lceil \log_m(N+1) \rceil \le h \le N$
  - B-트리: $\lceil \log_m(N+1) \rceil \le h \le \left\lceil \log_{\lceil \frac{m}{2} \rceil}(N+1) \right\rceil$

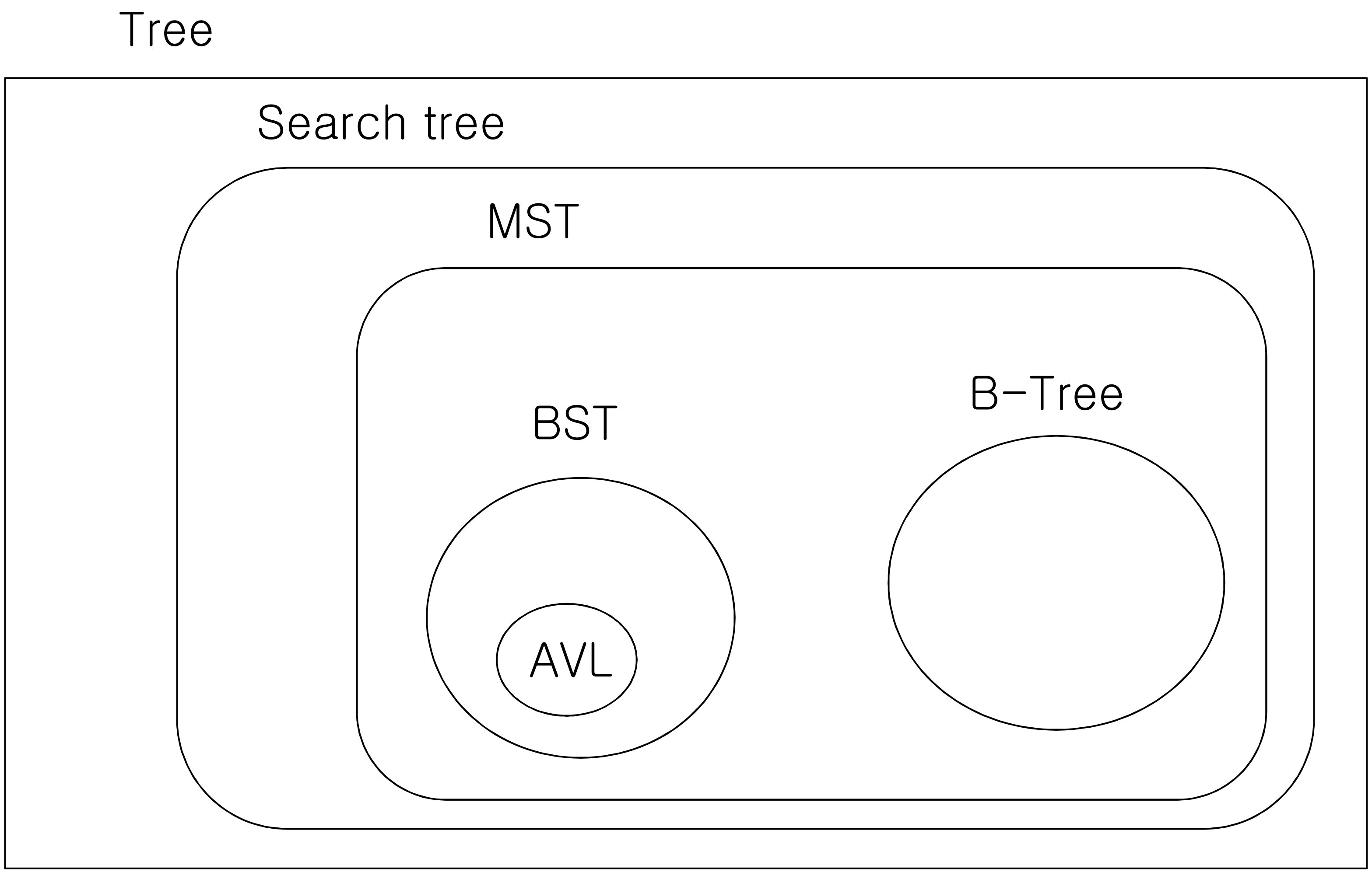
비교: N=1023 일때

- **BST (or 2-way search tree)**
  - 최소높이: $\lceil \log_2(N+1) \rceil = \lceil \log_2(1023+1) \rceil =$ 10
  - 최대높이: N = 1023
- **AVL (height-balanced BST)**
  - 최소높이: $\lceil \log_2(N+1) \rceil = \lceil \log_2(1023+1) \rceil =$ 10
  - 최대높이: $\lceil 1.4404\log_2(N+2)-0.328 \rceil =$ 15
- **MST**
  - m=3
    - 최소높이: $\lceil \log_3(N+1) \rceil = \lceil \log_3(1023+1) \rceil =$ 7
    - 최대높이: n = 1023
  - m=10
    - 최소높이: $\lceil \log_{10}(N+1) \rceil = \lceil \log_{10}(1023+1) \rceil =$ 4
    - 최대높이: n = 1023
- **B-Tree (balanced MST)**
  - m=3
    - 최소높이: $\lceil \log_3(N+1) \rceil = \lceil \log_3(1023+1) \rceil =$ 7
    - 최대높이: 모든 노드의 서브트리의 수가 최소일때, 즉 $\lceil m/2 \rceil$, 따라서 $\lceil \log_2(1023+1) \rceil =$ 10
  - m=10
    - 최소높이: $\lceil \log_{10}(N+1) \rceil = \lceil \log_{10}(1023+1) \rceil =$ 4
    - 최대높이: 모든 노드의 서브트리의 수가 최소일때, 즉 $\lceil m/2 \rceil$, 따라서 $\lceil \log_5(1023+1) \rceil =$ 5

비교: N=1,048,575 일 때

- **BST (or 2-way search tree)**
  - 최소높이: $\lceil \log_2(N+1) \rceil = 20$
  - 최대높이: n = 1,048,575
- **AVL (height-balanced BST)**
  - 최소높이: $\lfloor \log_2(N+1) \rceil = 20$
  - 최대높이: $\lceil 1.4404\log_2(N+2)-0.328 \rceil = 29$
- **MST**
  - m=3
    - 최소높이: $\lceil \log_3(N+1) \rceil = 13$
    - 최대높이: n = 1,048,575
  - m=10
    - 최소높이: $\lceil \log_{10}(N+1) \rceil = 7$
    - 최대높이: n = 1,048,575
- **B-Tree (balanced MST)**
  - m=3
    - 최소높이: $\lceil \log_3(N+1) \rceil = 13$
    - 최대높이: 모든 노드의 서브트리의 수가 최소일때, 즉 $\lceil m/2 \rceil$, 따라서 $\lceil \log_2(1023+1) \rceil = 20$
  - m=10
    - 최소높이: $\lceil \log_{10}(N+1) \rceil = 7$
    - 최대높이: 모든 노드의 서브트리의 수가 최소일때, 즉 $\lceil m/2 \rceil$, 따라서 $\lceil \log_5(1023+1) \rceil = 9$

비교



BST와 MST 계열 인덱스의 비교

- 메모리 적재용 인덱스 **(BST 계열)**
  - 키의 개수가 그리 크지 않을 때
  - 트리의 깊이가 다소 깊어도, 유지가 간단한 구조
  - BST, AVL 트리(height-balanced BST)

- 디스크 적재용 인덱스 **(MST 계열)**
  - 키의 개수가 매우 클 때 (대규모)
  - 유지가 힘들더라도, 트리의 깊이를 최소화
  - MST, B-트리(balanced MST)
  - 특히 B-트리의 경우
    - 모든 non-leaf는 메모리,
    - 모든 leaf는 디스크에 저장하여 성능을 향상.
  - N=1,000,000이고 m=100이면, $3 \le h \le 5$.
    N=1,000,000,000이고 m=200이면, $4 \le h \le 5$.

❖ B*-트리

- **B-트리의 문제점**
  - 구조 유지를 위해 추가적인 연산 필요
  - 삽입 : 노드의 분할, 삭제 : 노드의 합병 또는 재분배 필요

- **B*-트리**
  - 2/3 정도 찬 노드들을 가지는 B-트리
  - 삽입시 노드 분할(split)의 횟수 줄임

- **노드가 가득찬 경우 삽입**
  - B-트리: 즉시 분열
  - B*-트리: 분열 대신 인접한 형제 노드로 키를 분산(key redistribution)시킴

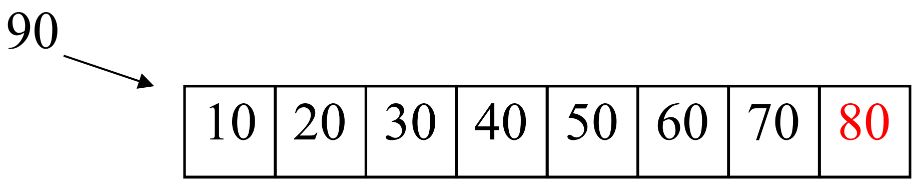
- **두 개의 이웃 노드가 모두 가득 찼을 때의 삽입**
  - 두 개의 노드를 세 개로 분할시킴
  - 2/3 가득 참

▶ B*-트리의 정의

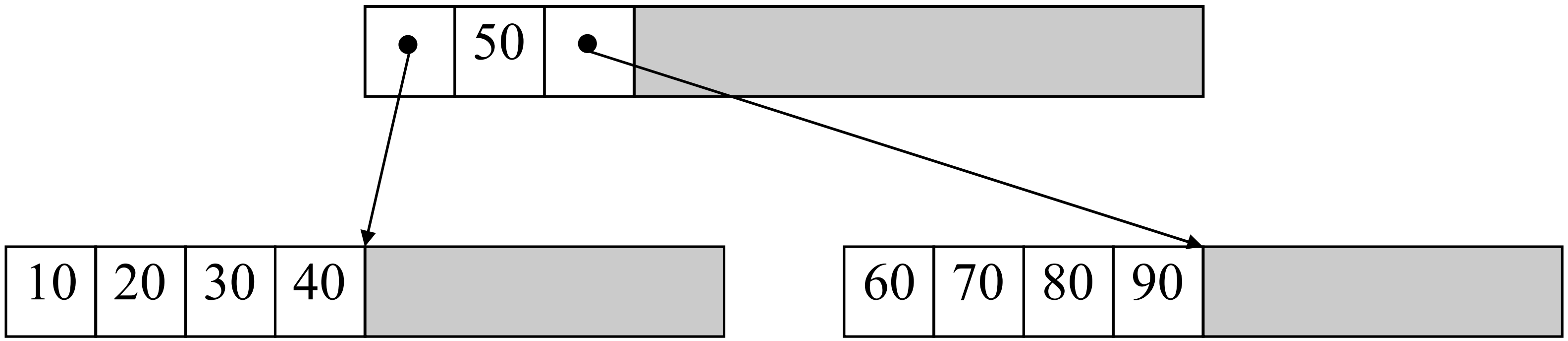
- **B*-트리 : Knuth가 제안한 B-트리의 변형**
  - ① B*-트리 : 공백이거나 높이가 1이상인 m-원 탐색 트리
  - ② 루트는 리프가 아닌 이상
    최소 2개, 최대 ⌊(2m-2)/3⌋+1 개의 서브트리를 갖는다.
  - ③ 루트, 리프를 제외한 모든 노드는
    적어도 ⌊(2m-2)/3⌋+1개의 서브트리, 즉 최소 ⌊(2m-2)/3⌋ 개의 키 값을 갖는다.
  - ④ 모든 리프는 같은 레벨에 있다.

- **특징**
  - 노드 분할의 빈도를 줄이기 위함
  - 각 내부 노드 : 2/3 이상 키 값으로 채워짐
  - B-트리보다 적은 수의 노드 필요

▶ 7차 B*-트리의 생성



(a) 8개의 키 값 삽입으로 루트 노드가 만원이 된 7차 B*-트리
(7차의 경우, 루트는 예외적으로 8개 키, 즉 9개 서브트리를 갖음)

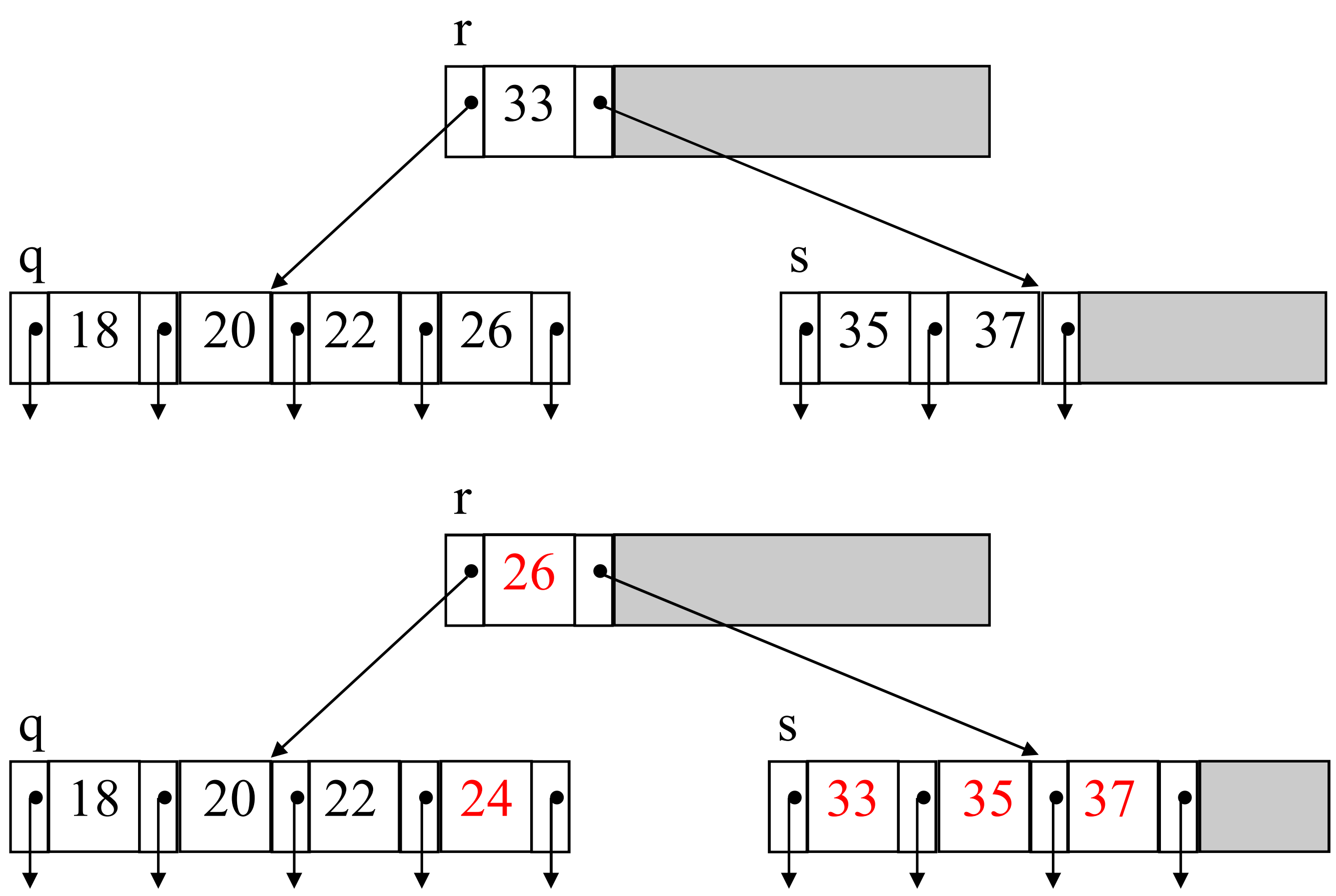


(b) 키 값 90의 삽입으로 루트 노드가 최초로 분할된 7차 B*-트리

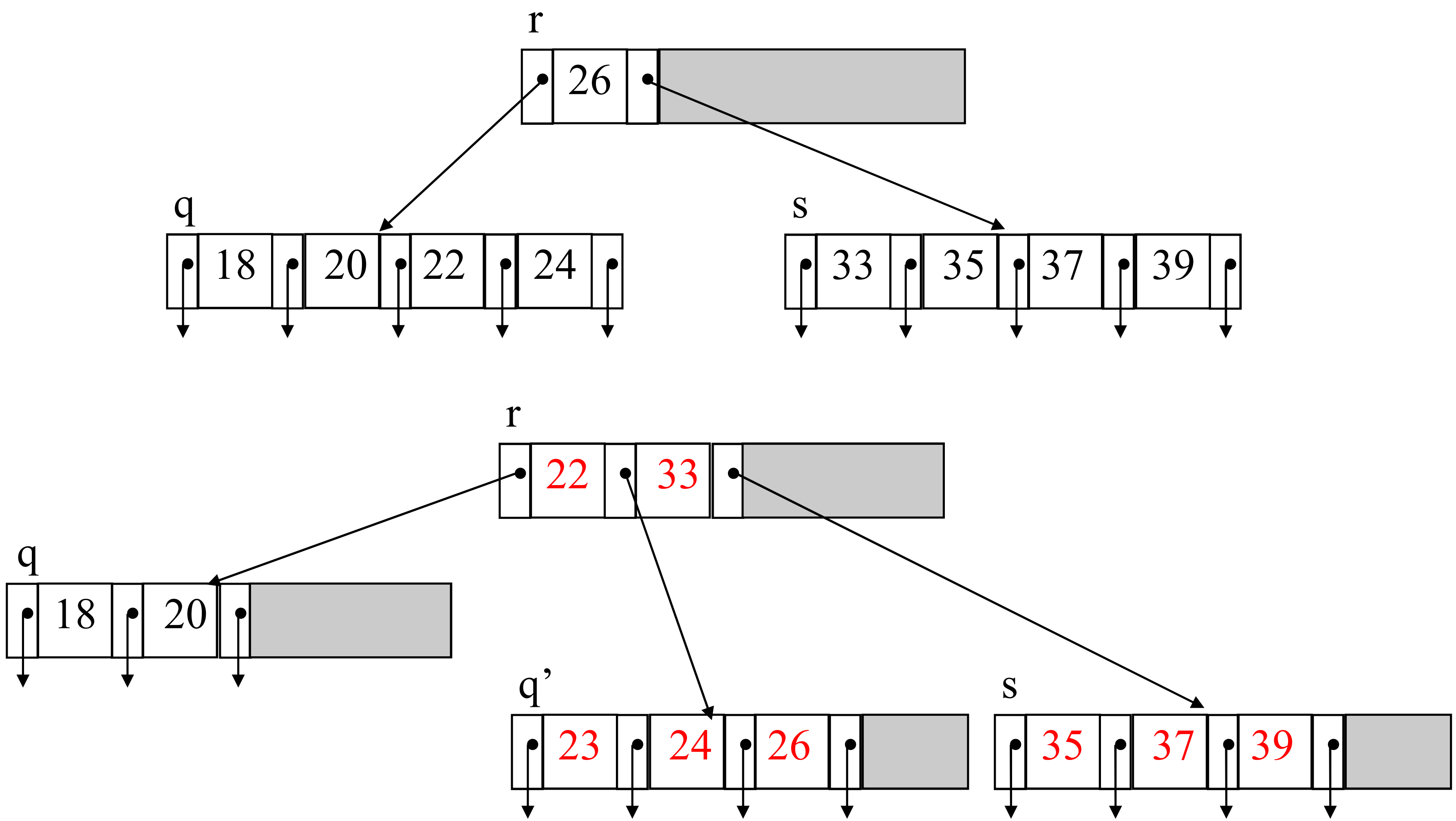
- 루트 노드의 키의 최소 개수
  - 루트를 제외한 모든 노드는 적어도 $\lfloor(2m-2)/3\rfloor$개 이상의 키 값을 가져야함.
  - 따라서 처음 루트 노드가 분할될 때, 생성되는 두 개의 리프 노드에는 각각 $\lfloor(2m-2)/3\rfloor$개의 키가 있어야함.
  - 따라서 루트 노드가 분할되기 직전에는, 루트에 최소 ($\lfloor(2m-2)/3\rfloor$) x 2개의 키를 담고 있어야 함.
- 예
  - m=5, 루트 노드의 키의 최소 개수=4
  - m=6, 루트 노드의 키의 최소 개수=6
  - m=7, 루트 노드의 키의 최소 개수=8
  - m=8, 루트 노드의 키의 최소 개수=8
  - m=9, 루트 노드의 키의 최소 개수=10
  - m=10, 루트 노드의 키의 최소 개수=12

▶ 5차 B*-트리에서의 삽입

◆ 재분배를 이용한 키 값 **24**의 삽입



◆ 노드 분할을 이용한 키 값 **23**의 삽입



❖ 트라이 (Trie)

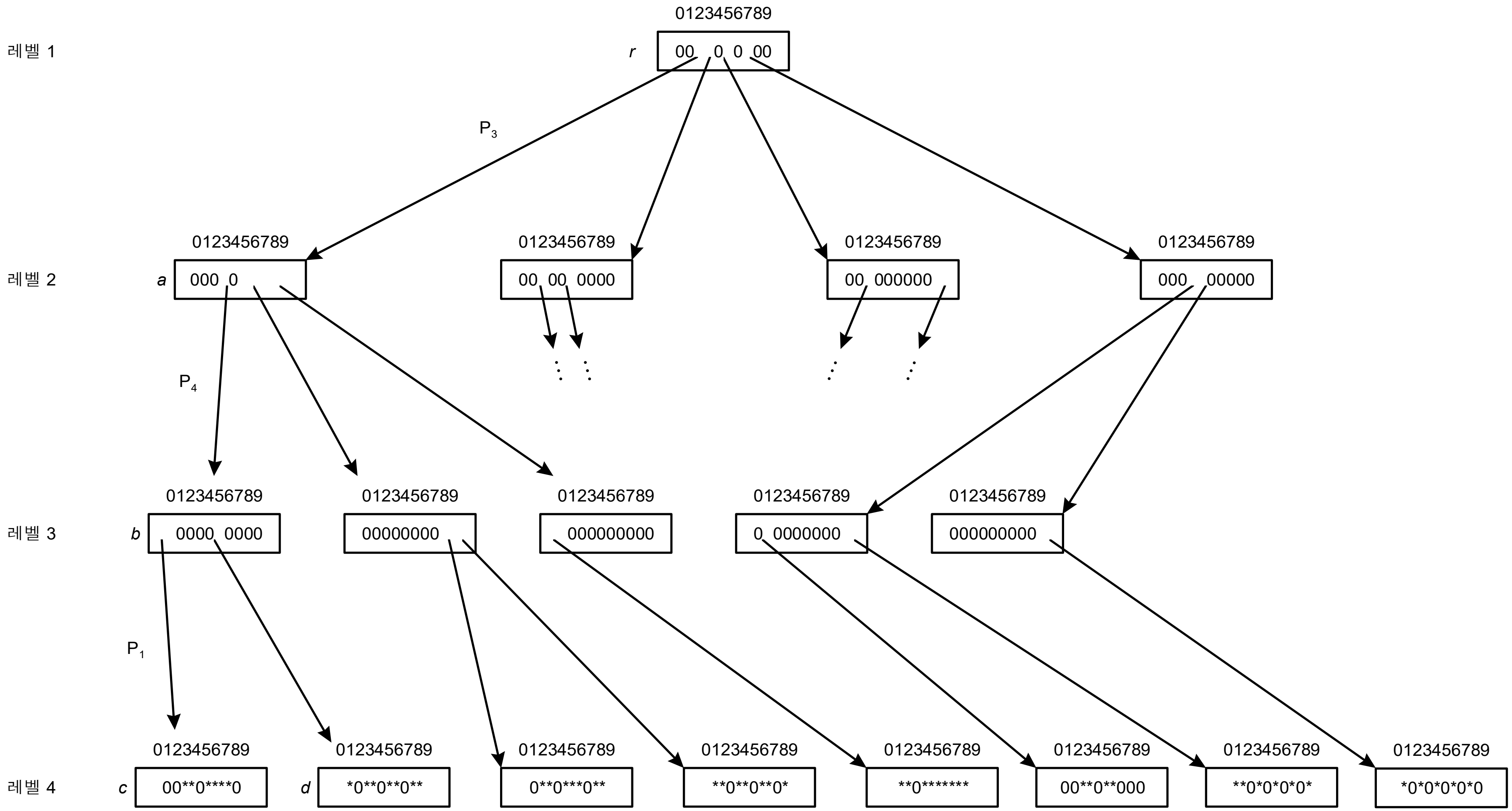
- **reTRIEval의 약자**
- 키를 구성하는 문자나 숫자의 순서로 키값을 표현한 구조
  - m-진 트리(m-ary tree), m-원 탐색 트리는 아님 : 키 값 배열순서 다름

- **m진 트라이(m-ary trie)**
  - 차수 m : 키 값을 표현하기 위해 사용하는 문자의 수, 즉 기수(radix)
  - 숫자 : 기수가 10이므로 m=10, 영문자 : m = 26
  - m진 트라이 : m개의 포인터를 표현하는 1차원 배열
  - 트라이의 높이 = 키 필드(스트링)의 길이

- **10진 트라이의 노드 구조**

0	1	2	3	4	5	6	7	8	9
P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}

▶ 높이가 4인 10진 트라이



0 : 널 포인터
* : 해당 키값을 가지고 있는 데이타 레코드의 주소

▶ m진 트라이 연산

- 탐색
  - 탐색 끝 : 리프 노드에서, 중간에 키 값이 없을 때
  - 탐색 속도 ≈ 키 필드의 길이 = 트라이의 높이
  - 최대 탐색 비용 ≤ 키 필드의 길이
  - 장점 : 균일한 탐색시간(단점 : 저장 공간이 크게 필요)
  - 선호하는 이유 : 없는 키에 대한 빠른 탐색 때문에

- 삽입
  - 리프 노드에 새 레코드의 주소나 마크를 삽입
  - 리프 노드 없을 때 : 새 리프 노드 생성, 중간 노드 첨가
  - 노드의 첨가나 삭제는 있으나 분열이나 병합은 없음

- 삭제
  - 노드와 원소들을 찾아서 널 값으로 변경
  - 노드의 원소 값들이 모두 널(공백노드) : 노드 삭제