Introduction to File Processing

6. 인덱스 구조 – Part B

Hyeokman Kim
School of Computer Science
Kookmin Univ.

File Processing Intro

B-트리

B-트리 (B-tree)

- Bayer & McCreight가 고안
  - 가장 많이 사용되는 인덱스 방법

- 균형된 m-원 탐색 트리 (Balanced MST)
  - 각 노드 마다 키 값이 최소 반 이상 찬 상태의 MST 트리.
    - 루트를 제외한 모든 노드는 최소 $\lceil m/2 \rceil - 1$ 개, 최대 $m-1$ 개의 키 값을 가짐.
    - 루트를 제외한 모든 내부 노드는 최소 $\lceil m/2 \rceil$ 개, 최대 m 개의 서브트리를 가짐.
  - 삽입/삭제 시 효율적인 균형 알고리즘을 제공

- B-트리의 노드 형식
  - MST와 동일
  - Note: 내부 노드와 단말 노드의 형식이 동일함.

B-트리의 노드 형식

- B-트리 노드의 형식 (MST와 동일)

  $<n, P_0, <K_1, A_1>, P_1, <K_2, A_2>, P_2, \cdots , P_{n-1}, <K_n, A_n>, P_n>$

  - n ($\lceil m/2 \rceil - 1 \le n \le m-1$): 노드 내의 키 값의 개수
  - P_i ($0 \le i \le n$) : 서브트리에 대한 포인터
  - K_i ($1 \le i \le n$) : 키 값
  - A_i ($1 \le i \le n$) : 키 값 K_i 를 가진 레코드에 대한 포인터

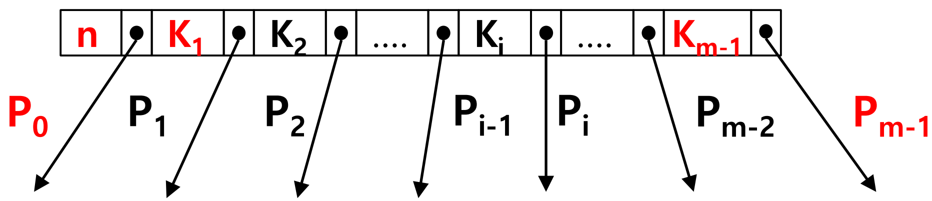
- Note
  - 내부 노드와 단말 노드의 형식이 같음.
  - 같은 키 값이 다른 노드에 존재할 수 없음.

B-트리의 제약조건

□ 제약조건

- K_i > (P_{i-1}가 가리키는 서브트리에 속한 모든 키)
- K_i < (P_i가 가리키는 서브트리에 속한 모든 키)

$\lceil m/2 \rceil - 1 \le$ n (키의 개수) $\le m-1$



$\lceil m/2 \rceil \le$ 서브트리의 개수 $\le m$

차수 m인 B-트리의 정의

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

② 루트는 "0" 또는 "2에서 m개 사이"의 서브트리를 가짐.

- 서브트리의 개수 = 0, 혹은 2 ≤ 서브트리의 개수 ≤ m

③ 루트를 제외한 모든 내부 노드는 최소 ⌈m/2⌉ 개, 최대 m 개의 서브트리를 가짐.

- ⌈m/2⌉ ≤ 서브트리의 개수 ≤ m
  (참고: m ≥ 3일 때, 항상 2 ≤ ⌈m/2⌉임)

④ 루트를 제외한 모든 내부 노드에 있는 키의 개수는 그 노드의 서브트리 개수 보다 하나 적음.

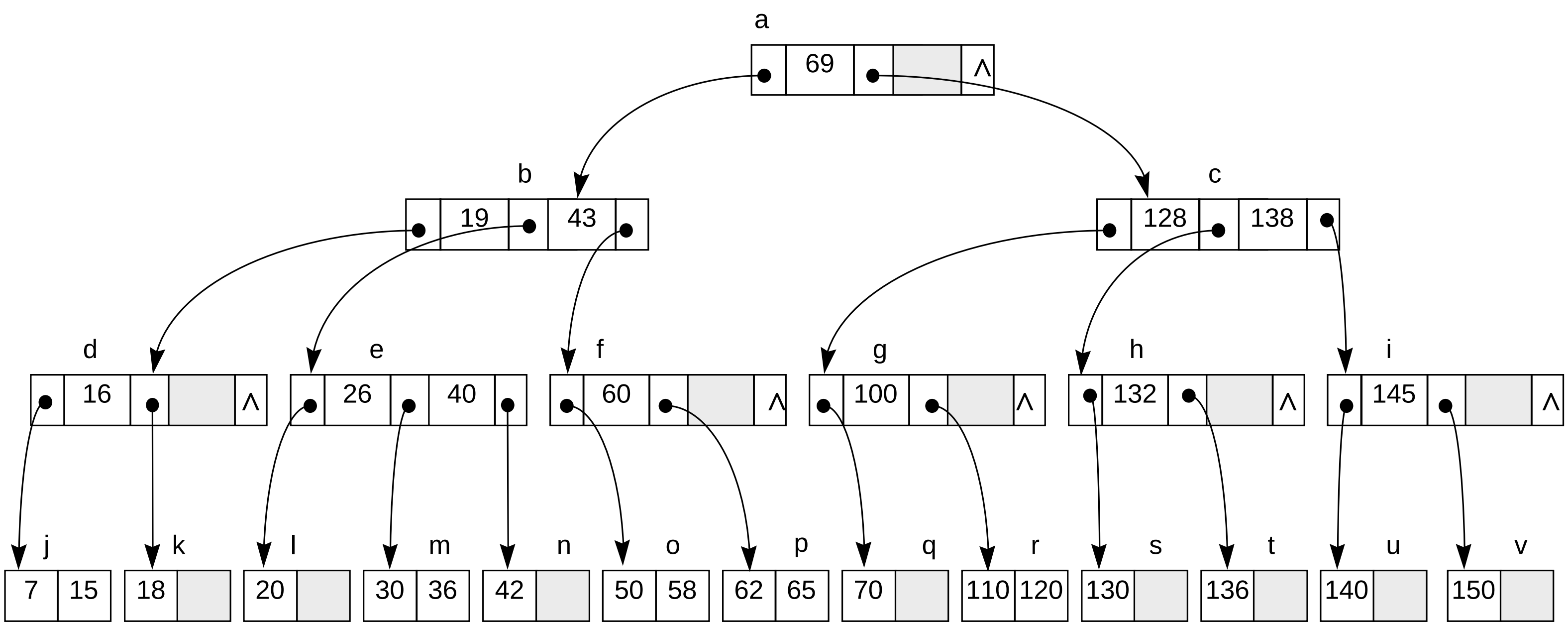
- ⌈m/2⌉-1 ≤ 키의 개수 ≤ m-1 (노드의 반 이상이 채워짐)
- Note: 루트의 경우, 적어도 ⌈ m/2 ⌉- 1개 이상의 키 값을 가져야 한다는 제약이 없음 (예, m=5). 루트가 꽉 차면 분기.

⑤ 모든 단말 노드는 같은 레벨

B-트리의 장점

- 삽입, 삭제 뒤에도 균형 상태 유지 (재균형이 필요 없음)
- 검색 성능: 최소 $O(\log_m(N+1))$

예: 3-원 B-트리



m=3

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

1. B-트리 연산: 검색

- 검색 : MST의 검색과 같은 과정
  - 임의 접근(random access)
    - m-원 탐색(m-way search) : 키 값에 의존한 분기 (예: 42 검색)
    - 시간 복잡도 : 최소 $O(\log_m N)$, 최대 $O(\log_{m/2} N)$
  - 순차 접근(sequential access)
    - 중위 순회(in-order traversal)
    - 순차 접근은 가능하나, 성능의 측면에서 의미는 없음.

2. B-트리 연산: 삽입

- 삽입 연산의 개요
  - 항상 단말 노드에 삽입, 단말에서 삽입이 위쪽으로 전파됨.

- 알고리즘

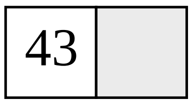
  ① 노드에 여유 공간이 있는 경우 (n < m-1)
  - 노드 내의 키 값 순서에 맞는 위치에 삽입.

  ② 노드가 차있는 경우 (n = m-1) : overflow가 발생함.
  - 해당 노드를 두 개의 노드로 **분할(split)**함.
  - 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 삽입. 이 때, 다시 overflow가 발생하면, 위의 과정(분할)을 반복 (삽입의 전파)

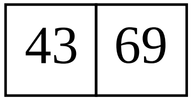
예: 3차 B-트리의 생성

- 키 값 43, 69, 138, 19 순으로 삽입

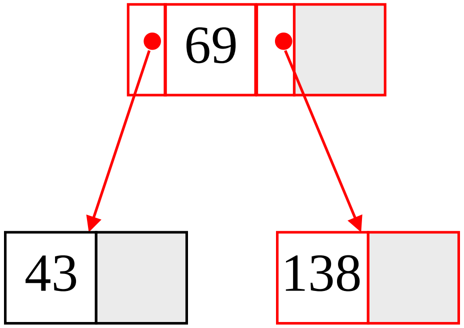
(a) 크기가 2인 공백 루트 노드



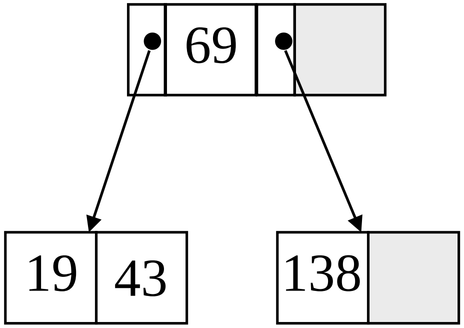
(b) 키 값 43의 삽입(노드 1개의 3차 B-트리)



(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)

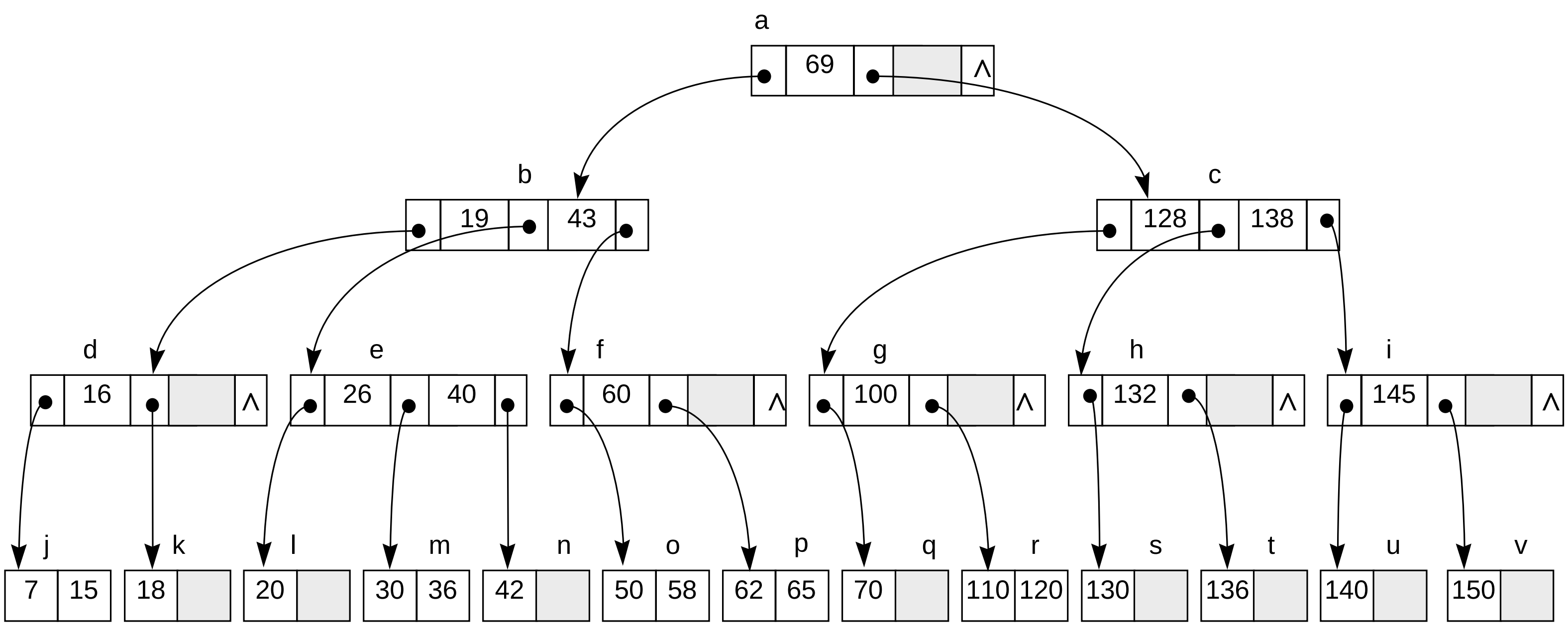
Note: B-트리에서 루트 노드의 분할

- 루트 노드의 키의 개수
  - 루트를 제외한 모든 노드는 적어도 $\lceil m/2 \rceil - 1$개 이상의 키 값을 가져야함.
  - 따라서 처음 루트 노드가 분할될 때, 생성되는 두 개의 단말 노드에는 각각 적어도 $\lceil m/2 \rceil - 1$개의 키가 있어야 함.
  - 분할할 때 최소 키 값의 개수 보다 많으면, 어느 한 쪽에 하나 더 많은 키를 분배함.

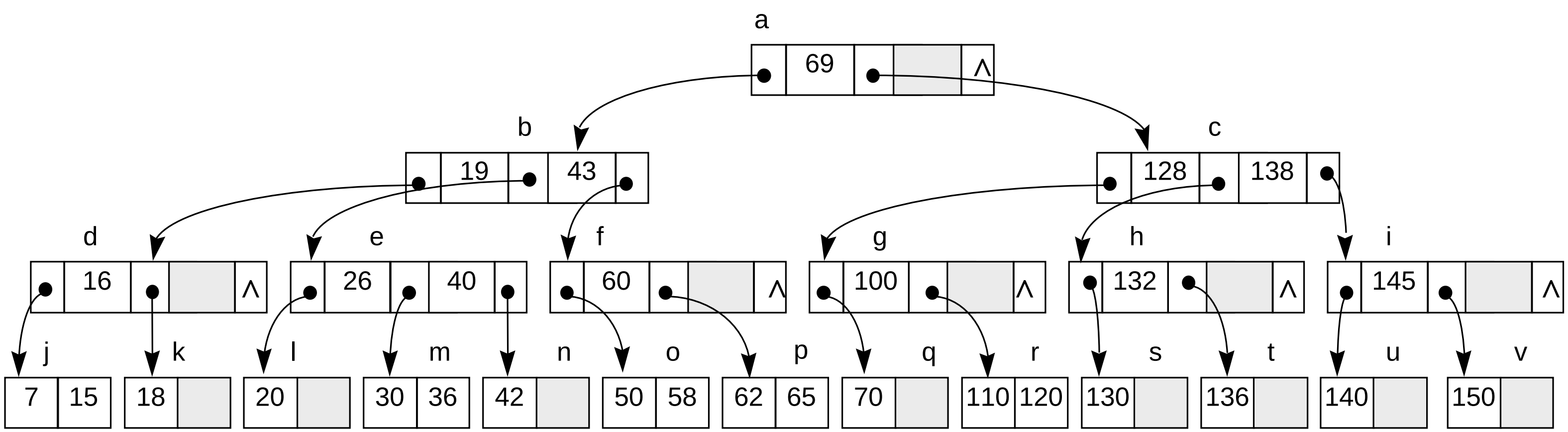
- 예
  - m=3, 루트 노드의 키의 개수=2 (1, 1개로 분할)
  - m=4, 루트 노드의 키의 개수=3 (2, 1개로 분할)
  - m=5, 루트 노드의 키의 개수=4 (2, 2개로 분할)
  - m=6, 루트 노드의 키의 개수=5 (3, 2개로 분할)
  - m=7, 루트 노드의 키의 개수=6 (3, 3개로 분할)
  - m=8, 루트 노드의 키의 개수=7 (4, 3개로 분할)

예제 1

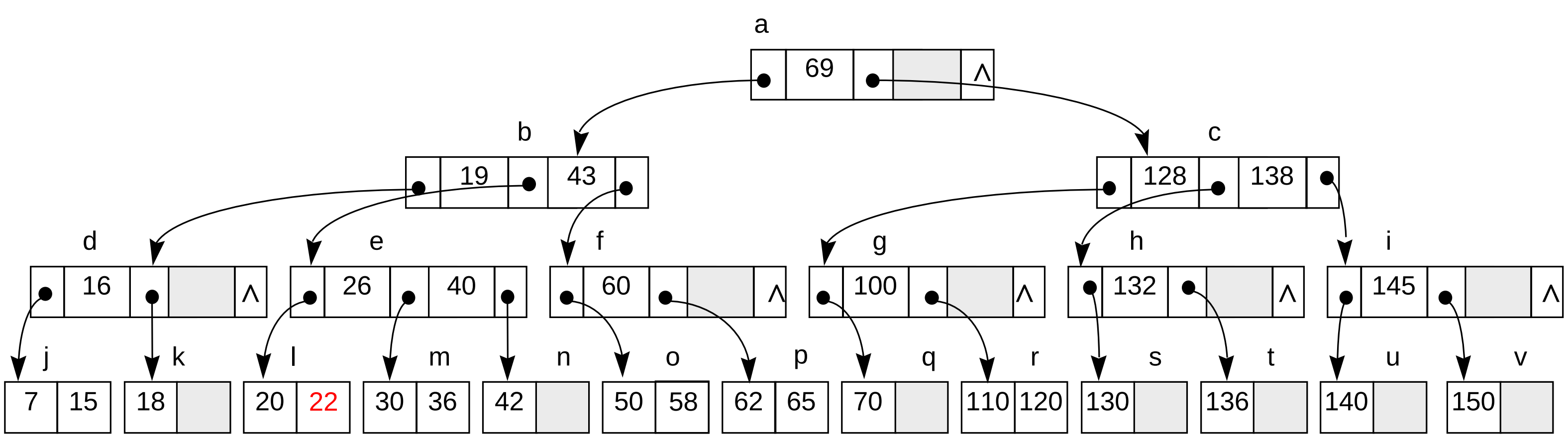
- 다음 B-트리(그림 6-18)에 새로운 키 값 22, 41, 59, 57, 54, 33, 75, 124, 122, 123 삽입



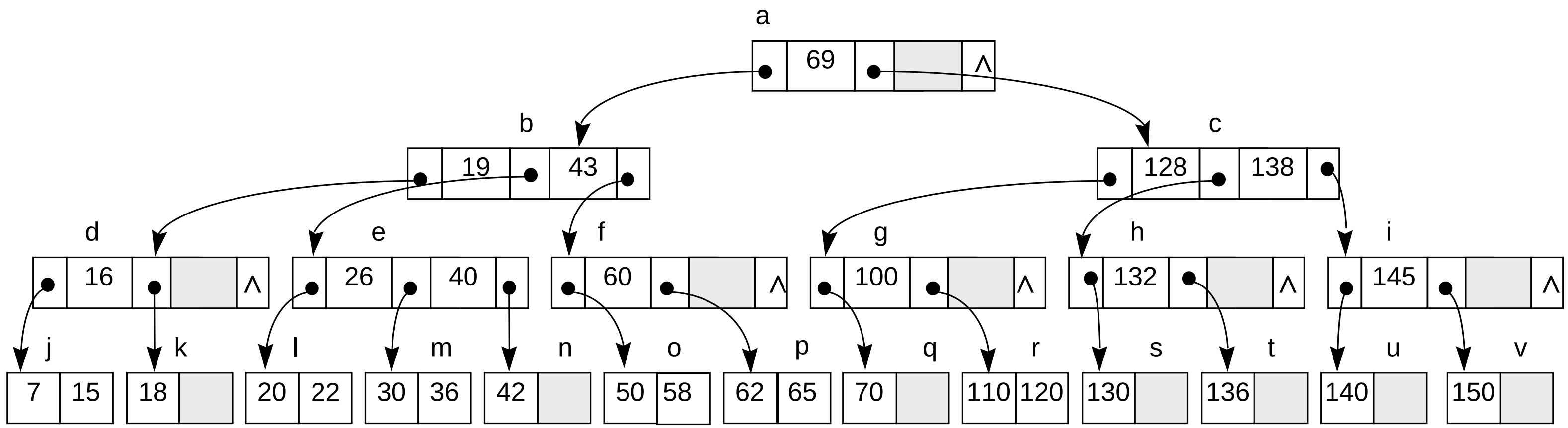
22 삽입


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

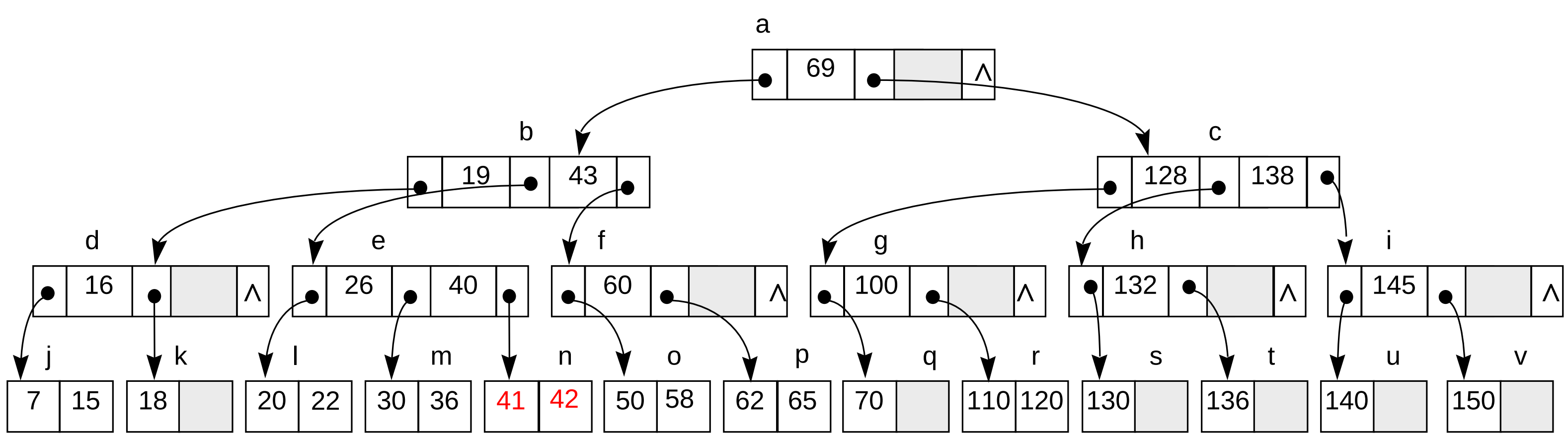
단말에서 삽입


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

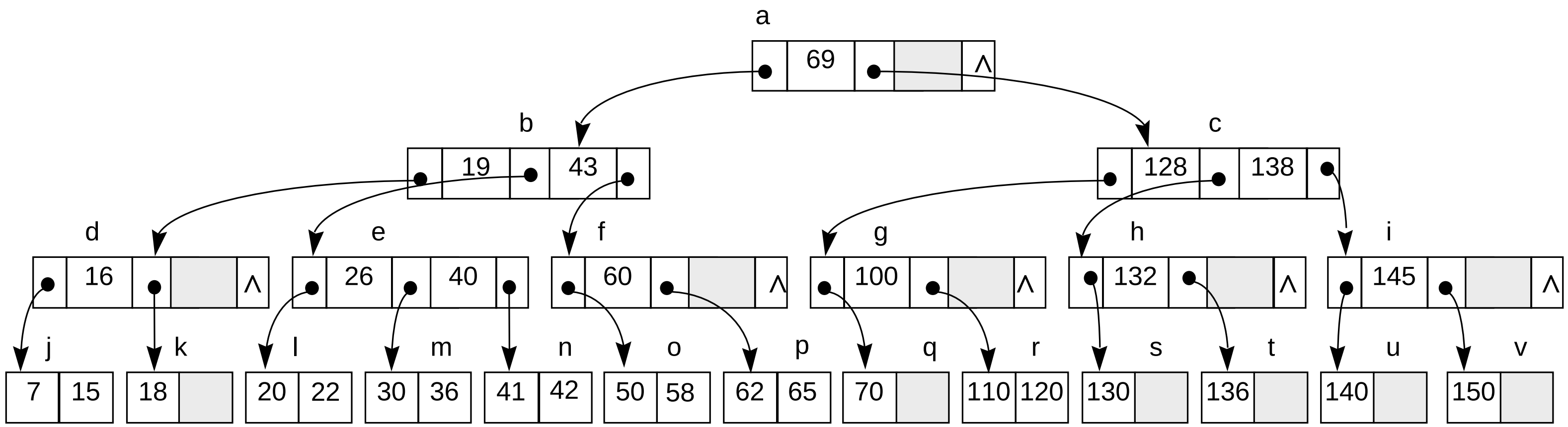
41 삽입


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
7
15
18
20
22
30
36
42
50
58
62
65
70
110
120
130
136
140
150

단말에서 삽입

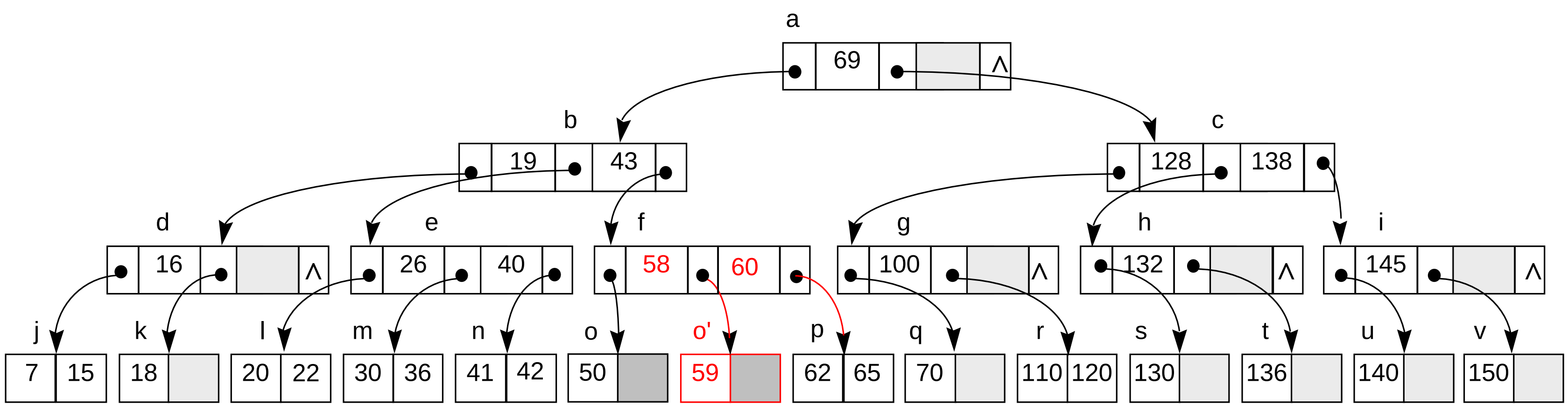

a
69
b
19
43
c
128
138
d
16
e
26
40
f
60
g
100
h
132
i
145
j
k
l
m
n
o
p
q
r
s
t
u
v
7
15
18
20
22
30
36
41
42
50
58
62
65
70
110
120
130
136
140
150

59 삽입



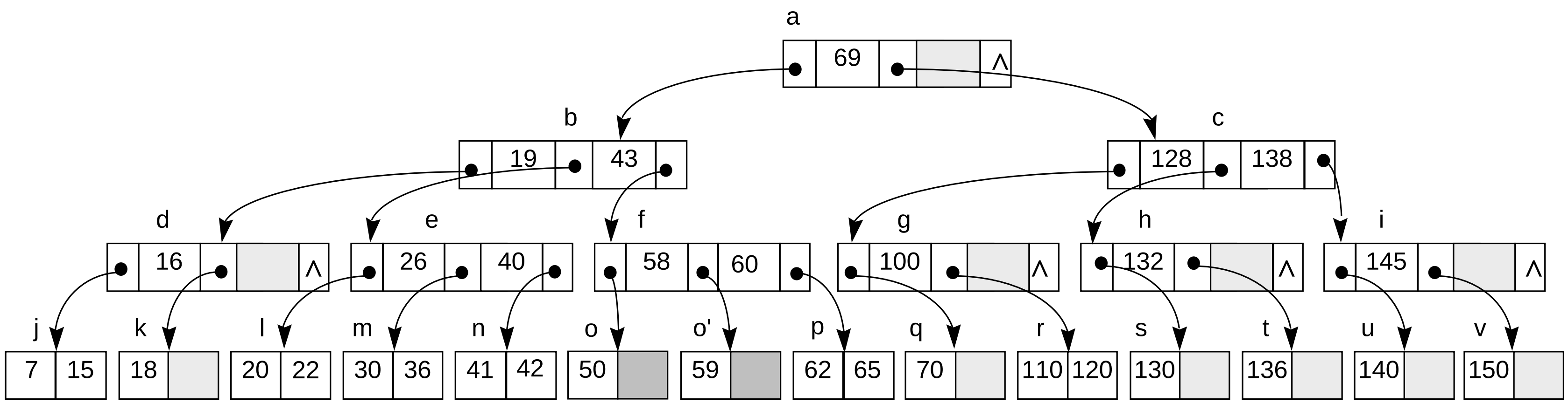
a
69
b
19
43
c
128
138
d
16
e
26
40
f
60
g
100
h
132
i
145
j
7
15
k
18
l
20
22
m
30
36
n
41
42
o
50
58
p
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150

노드 분할 (1회)

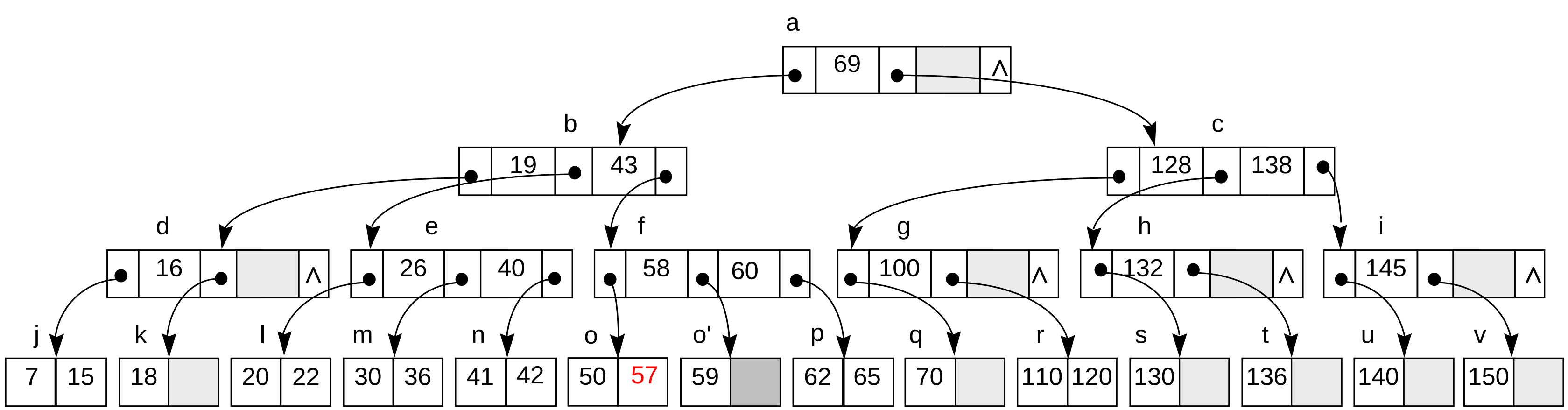


a
69
b
19
43
c
128
138
d
16
e
26
40
f
58
60
g
100
h
132
i
145
j
7
15
k
18
l
20
22
m
30
36
n
41
42
o
50
o'
59
p
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150

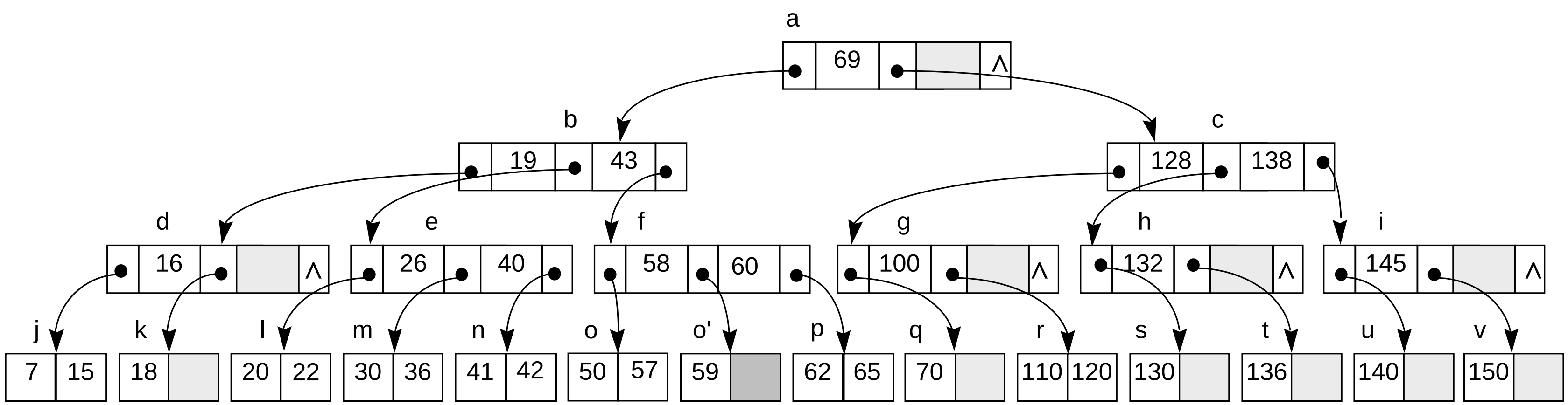
57 삽입


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

단말에서 삽입

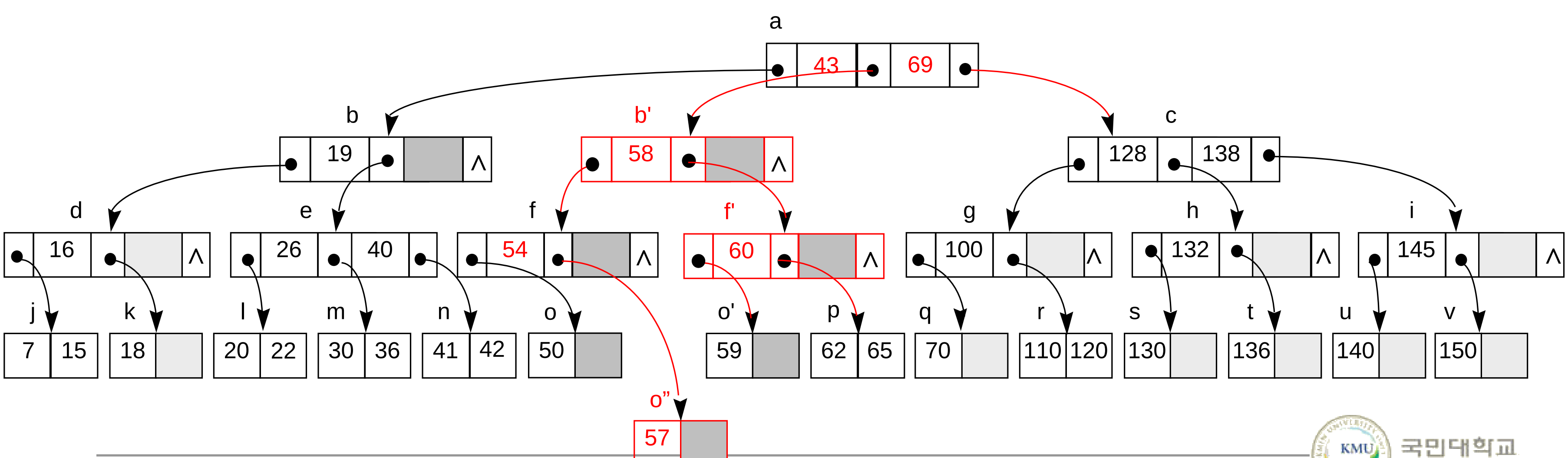

a
69
b
19
43
c
128
138
d
16
e
26
40
f
58
60
g
100
h
132
i
145
j
k
l
m
n
o
o'
p
q
r
s
t
u
v
7
15
18
20
22
30
36
41
42
50
57
59
62
65
70
110
120
130
136
140
150

54 삽입



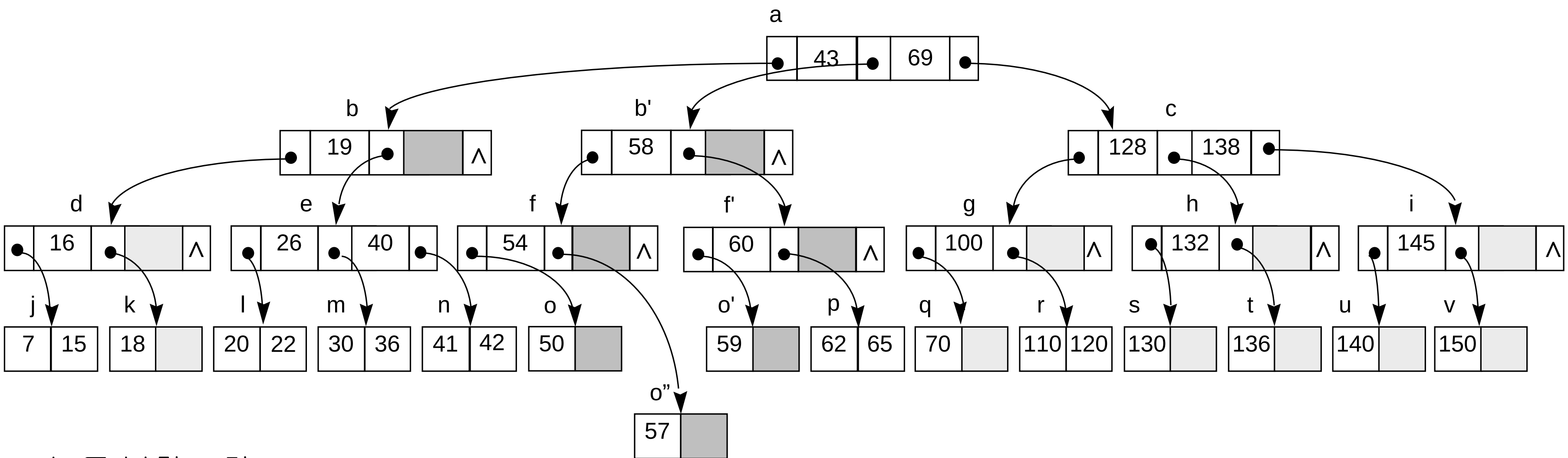
a
69 ∧
b
19 43
c
128 138
d
16 ∧
e
26 40
f
58 60
g
100 ∧
h
132 ∧
i
145 ∧
j
7 15
k
18
l
20 22
m
30 36
n
41 42
o
50 57
o'
59
p
62 65
q
70
r
110 120
s
130
t
136
u
140
v
150

노드 분할 (3회)

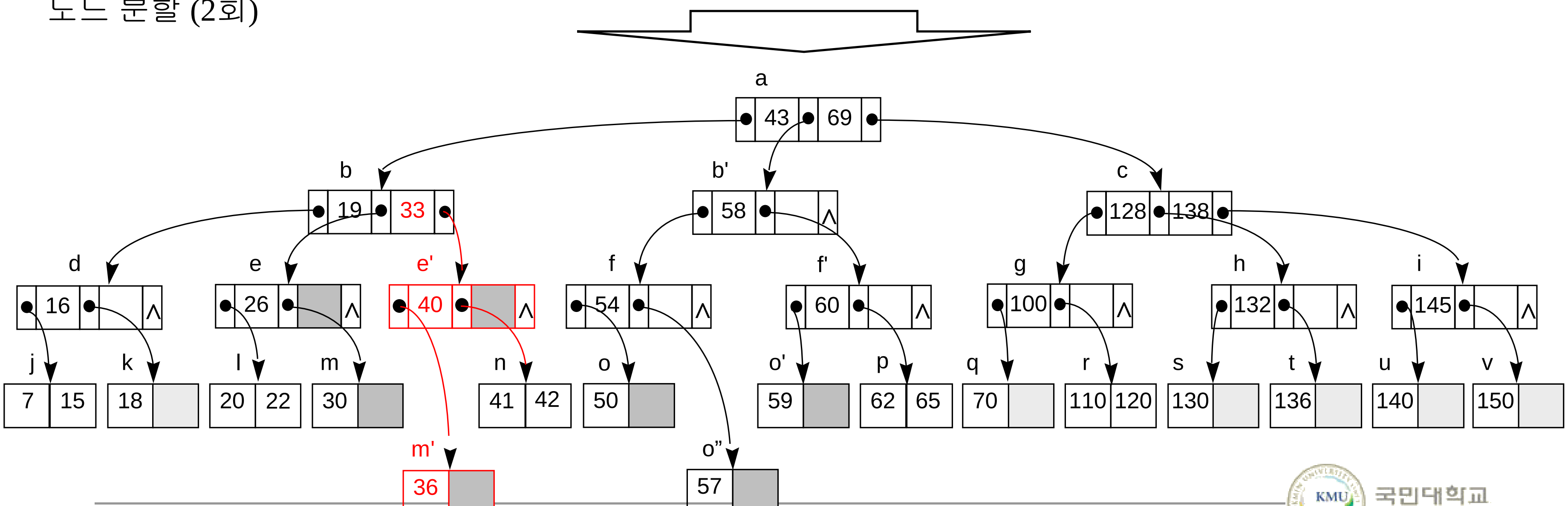


a
43 69
b
19 ∧
b'
58 ∧
c
128 138
d
16 ∧
e
26 40
f
54 ∧
f'
60 ∧
g
100 ∧
h
132 ∧
i
145 ∧
j
7 15
k
18
l
20 22
m
30 36
n
41 42
o
50
o'
59
p
62 65
q
70
r
110 120
s
130
t
136
u
140
v
150
o”
57

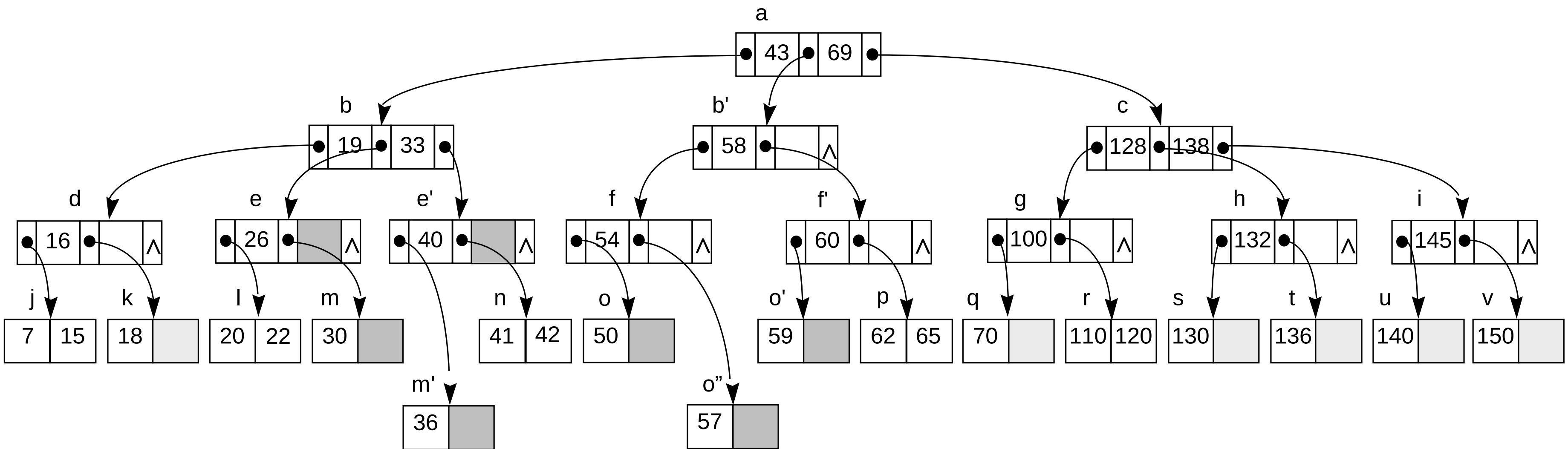
33 삽입


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

노드 분할 (2회)

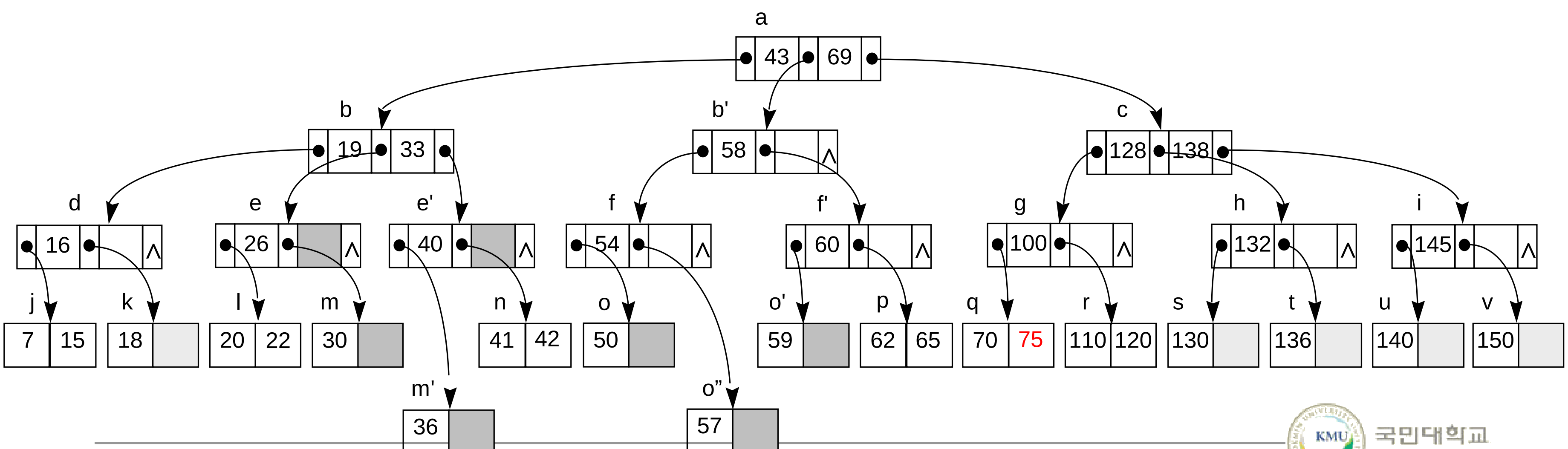

a
43
69
b
19
33
b'
58
c
128
138
d
16
e
26
e'
40
f
54
f'
60
g
100
h
132
i
145
j
7
15
k
18
l
20
22
m
30
n
41
42
o
50
o'
59
p
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150
m'
36
o”
57

75 삽입



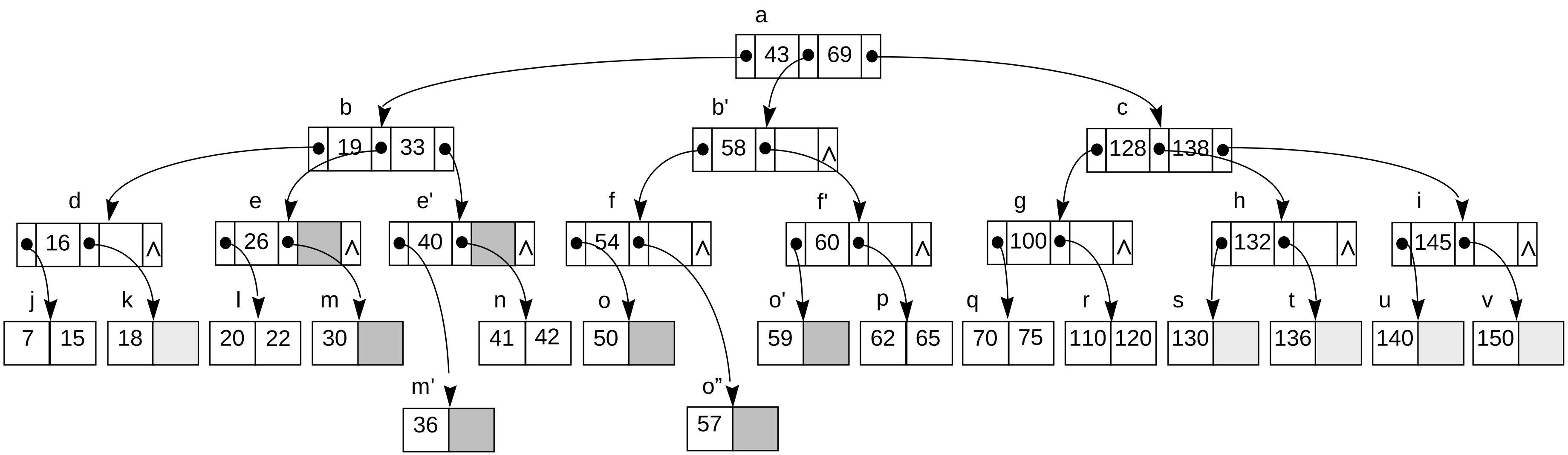
a
43
69
b
19
33
b'
58
c
128
138
d
16
e
26
e'
40
f
54
f'
60
g
100
h
132
i
145
j
7
15
k
18
l
20
22
m
30
n
41
42
o
50
o'
59
p
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150
m'
36
o"
57

단말에서 삽입

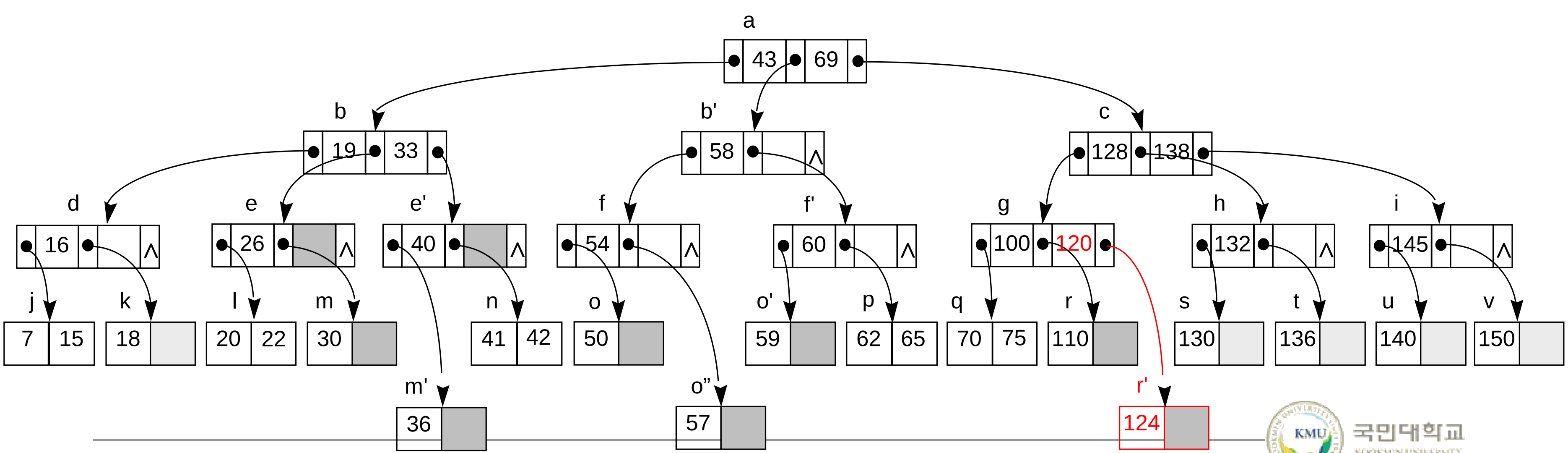


a
43
69
b
19
33
b'
58
c
128
138
d
16
e
26
e'
40
f
54
f'
60
g
100
h
132
i
145
j
7
15
k
18
l
20
22
m
30
n
41
42
o
50
o'
59
p
62
65
q
70
75
r
110
120
s
130
t
136
u
140
v
150
m'
36
o"
57

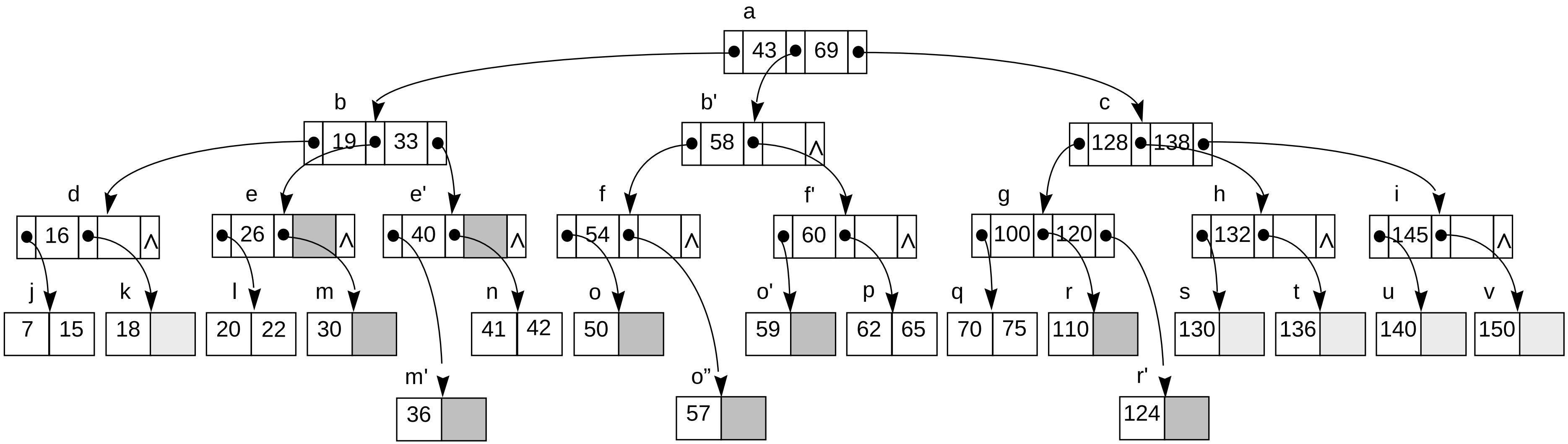
124 삽입



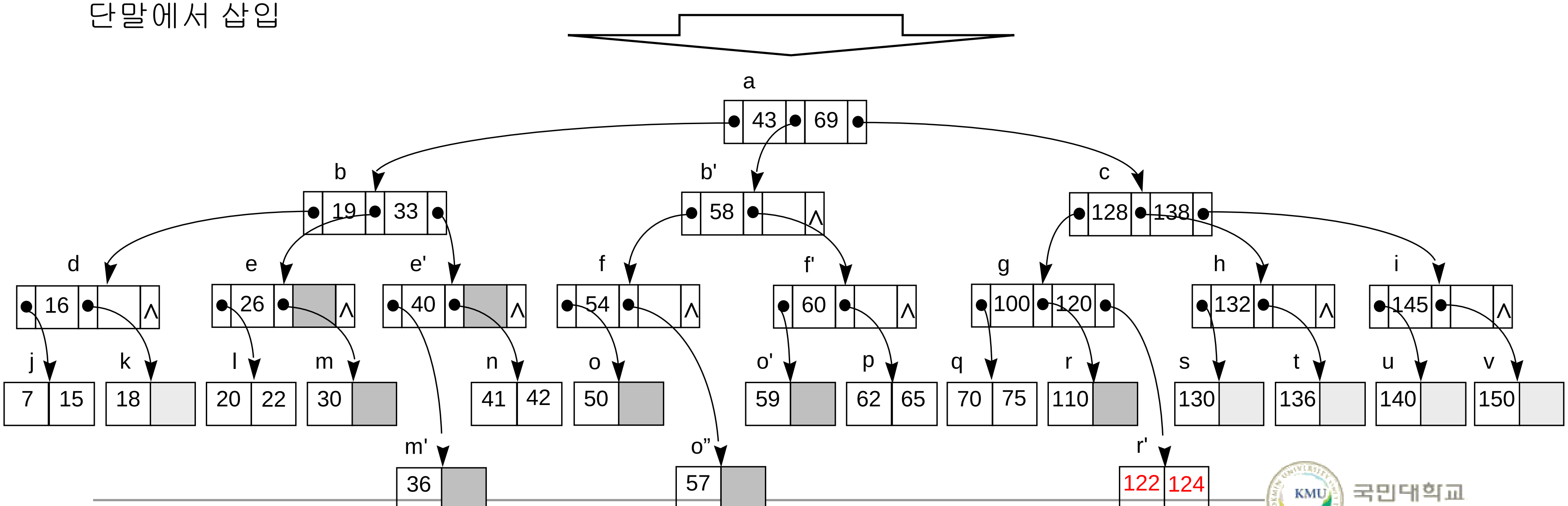
노드 분할 (1회)



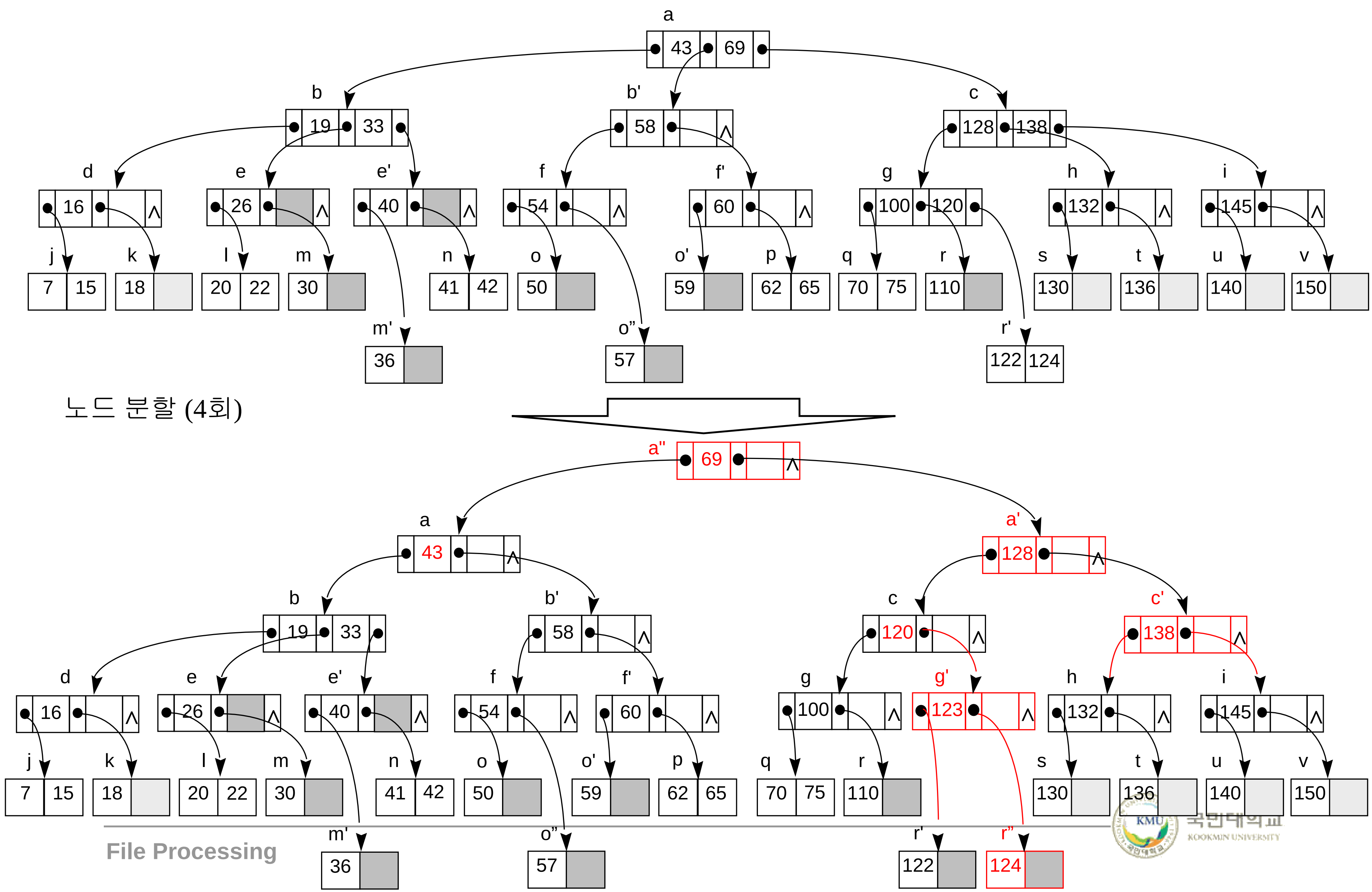
122 삽입



단말에서 삽입

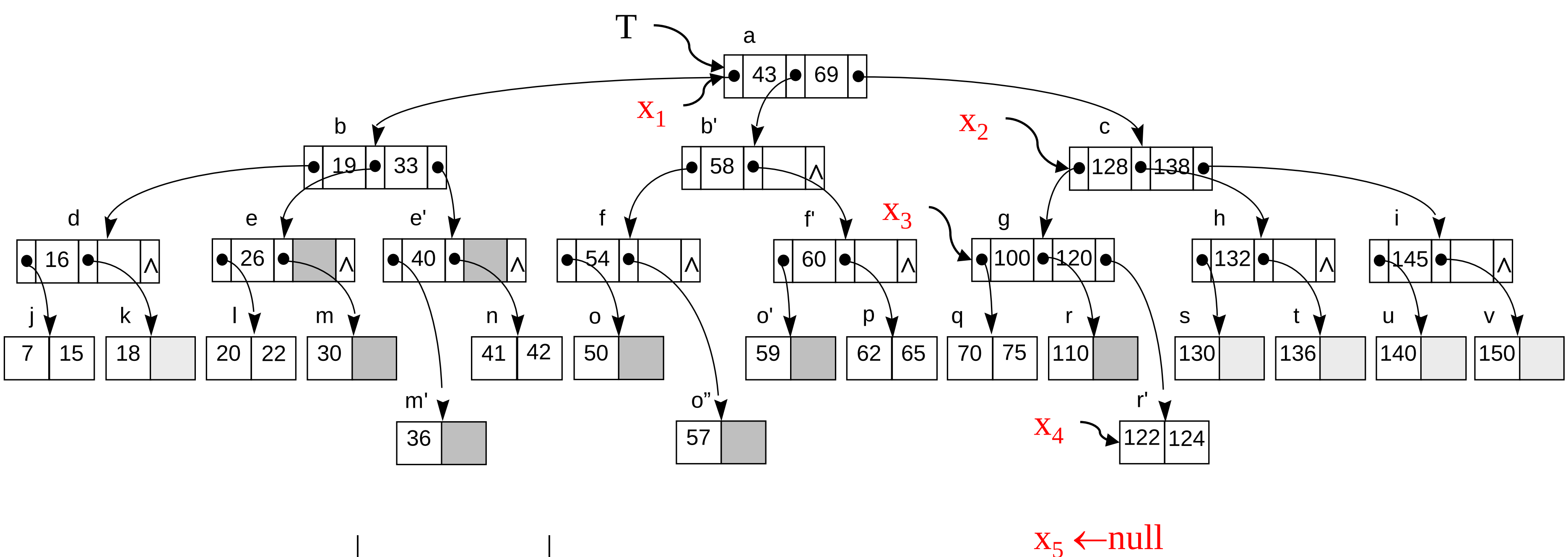


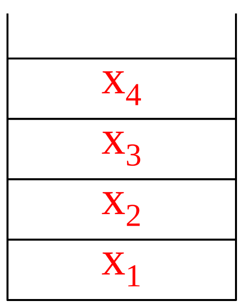
123 삽입 (한 레벨 증가)



B-트리 삽입 알고리즘의 개요

insertBT(T, 3, 123)





insertKey(3, x_4, 123)

insertKey(3, x_3, 123)

insertKey(3, x_2, 120)

insertKey(3, x_1, 128)

B-트리 삽입 알고리즘

```
insertBT(T, m, newKey)

    If (T = null) {              // 루트 노드를 생성한다.
        T ← getNode();           // T.n ← 0; T. Ki ← null; Pi ← null;
        T.K1 ← newKey;    T.n ← 1;
        return;
    }

    // x 노드의 주소를 스택에 저장하면서 newKey가 삽입될 위치를 탐색함.
    x ← T;
    do {
        i ← 1;
        while (i <= x.n && newKey > x.Ki )
                i ← i+1;
        if (i <= x.n && newKey = x.Ki)              // 삽입할 키를 발견함. 삽입 불가.
        then return;

        // for some i where Ki-1 < newKey < Ki, 삽입할 키를 아직 발견하지 못함.
        push x at the top of a stack;
    } while ( (x ← x.Pi-1) != null );

    // 삽입할 키가 없으므로, 아래 과정에서 newKey를 B-트리에 삽입. 이제 x는 null
```

```
// tempNode :  오버플로를 위한 노드 (x 노드와 newKey를 저장하기 위한 임시 노드)

finished = FALSE;
x ← pop a value from the top of the stack;

do {
        if (x.n < m-1) {insert newKey at an appropriate position of x;    finish = TRUE;}

        copy x node to tempNode;
        insert newKey at an appropriate position of tempNode;

        newKey ← center key of tempNode;
        copy 1st half of tempNode to x;
        y ← getNode();
        copy 2nd half of tempNode to y;

        if (stack is not empty) {
                x ← pop a value from the top of the stack;
        }
        else {                     // 트리의 레벨이 하나 증가한다.
                T ← getNode();              // T.n ← 0; T. K_i ← null; P_i ← null;
                T.K_1 ← newKey;
                T.P_0 ← x;
                T.P_1 ← y;
                finish = TRUE;
        }
} while (!finish);
```

3. B-트리 연산: 삭제

- 삭제 연산의 개요
  - 삭제할 키가 내부 노드에 있으면, 후행키와 교체. 따라서 삭제할 노드는 항상 단말 노드에 존재하게 됨.
  - 항상 단말 노드에서 삭제가 시작되며, 단말에서 삭제가 루트 쪽으로 전파됨.

- 알고리즘
  - Case 1: 삭제할 키가 내부 노드에 있을 때
    - 이 키 값의 "후행 키" 값과 교환 후 삭제
    - 후행 키 값은 언제나 단말 노드에 있으므로, 교환 후에는 내부 노드에 있던 삭제할 키 값이 단말 노드에 위치하게 됨. 따라서 아래 단계를 적용할 수 있음.
  - Case 2: 삭제할 키가 단말 노드에 있을 때
    - (다음 슬라이드)

- Case 2: 삭제할 키가 단말 노드에 있을 때

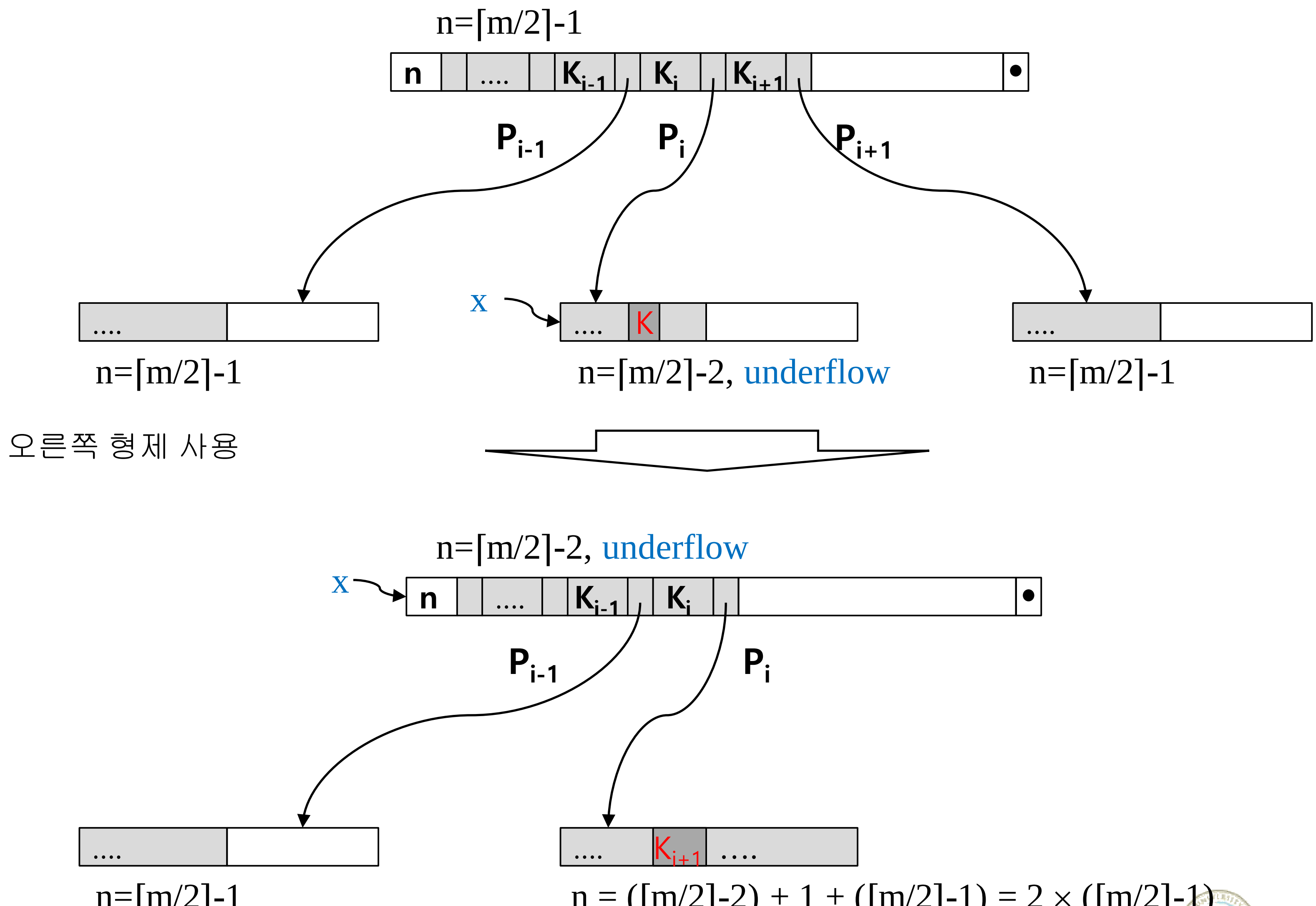
① 노드에 키 값이 반 보다 더 찬 경우 (n > ⌈m/2⌉−1)

- 키 삭제

② 노드에 키 값이 반만 찬 경우 (n = ⌈m/2⌉−1): underflow 발생

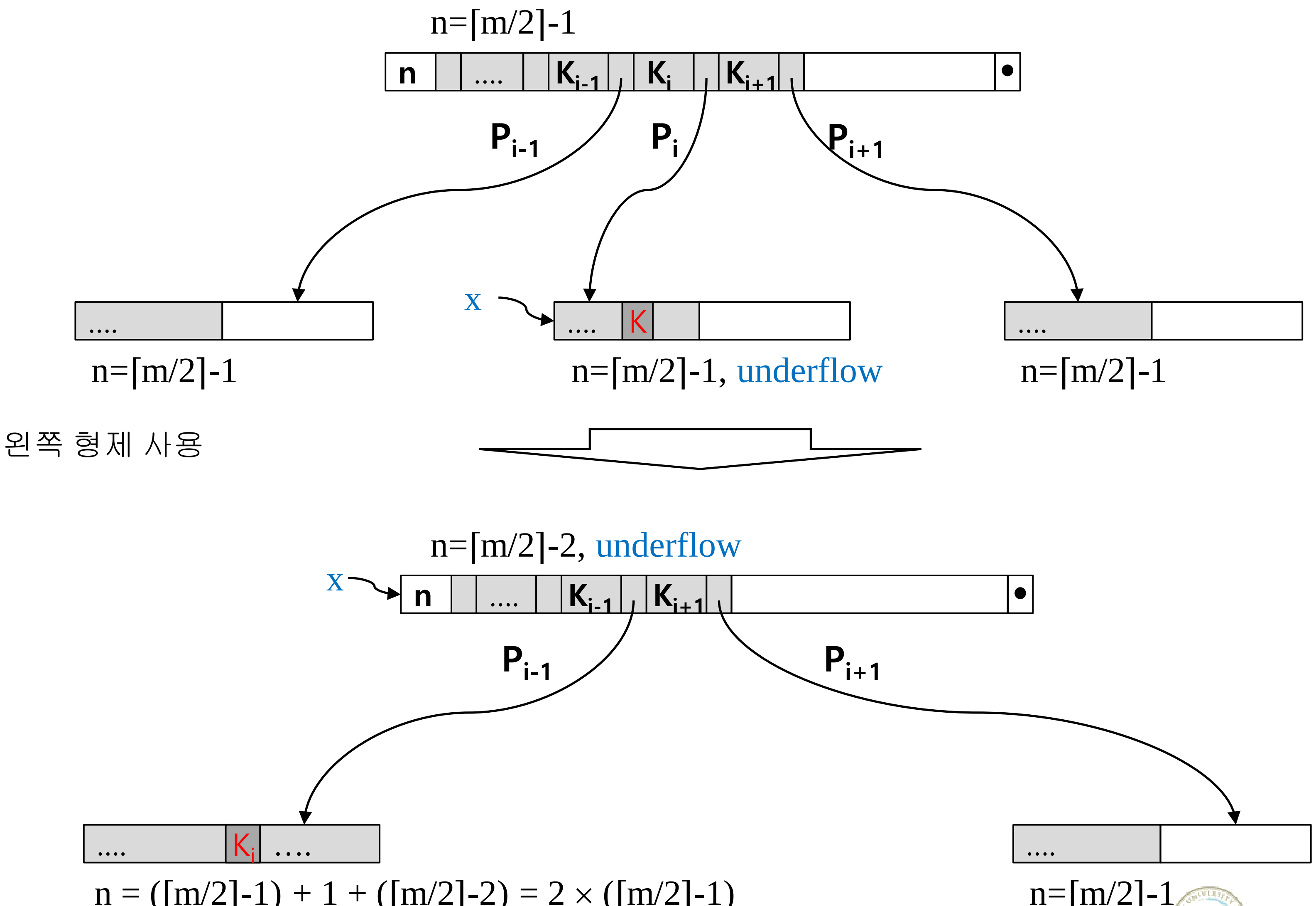
- 키 재분배(key redistribution) : 트리 구조가 변경되지 않음.
  - 최소 키 값보다 많은 키를 가진 "형제 노드"에서 키 값 차출. (왼쪽 먼저)
  - 부모 노드에 있던 키 값을 해당 노드로 이동, 빈 자리에 차출된 형제 노드의 키 값을 이동
- 노드 합병(node merge) : 트리 구조가 변경됨.
  - 키 재분배가 불가능한 경우에 적용
  - 형제 노드와 **합병(merge)**하며(왼쪽 먼저), 합병 결과 빈 노드는 제거함. 이때 부모 노드의 중간 키 값도 같이 합병됨. (부모 노드에서 삭제 발생).
  - 만일 부모 노드에서 삭제 중 다시 underflow가 발생하면, 위의 과정(합병)을 반복함. (underflow의 전파)

Underflow 발생시의 노드 합병 (Node merge) : 우측



국민대학교
KOOKMIN UNIVERSITY

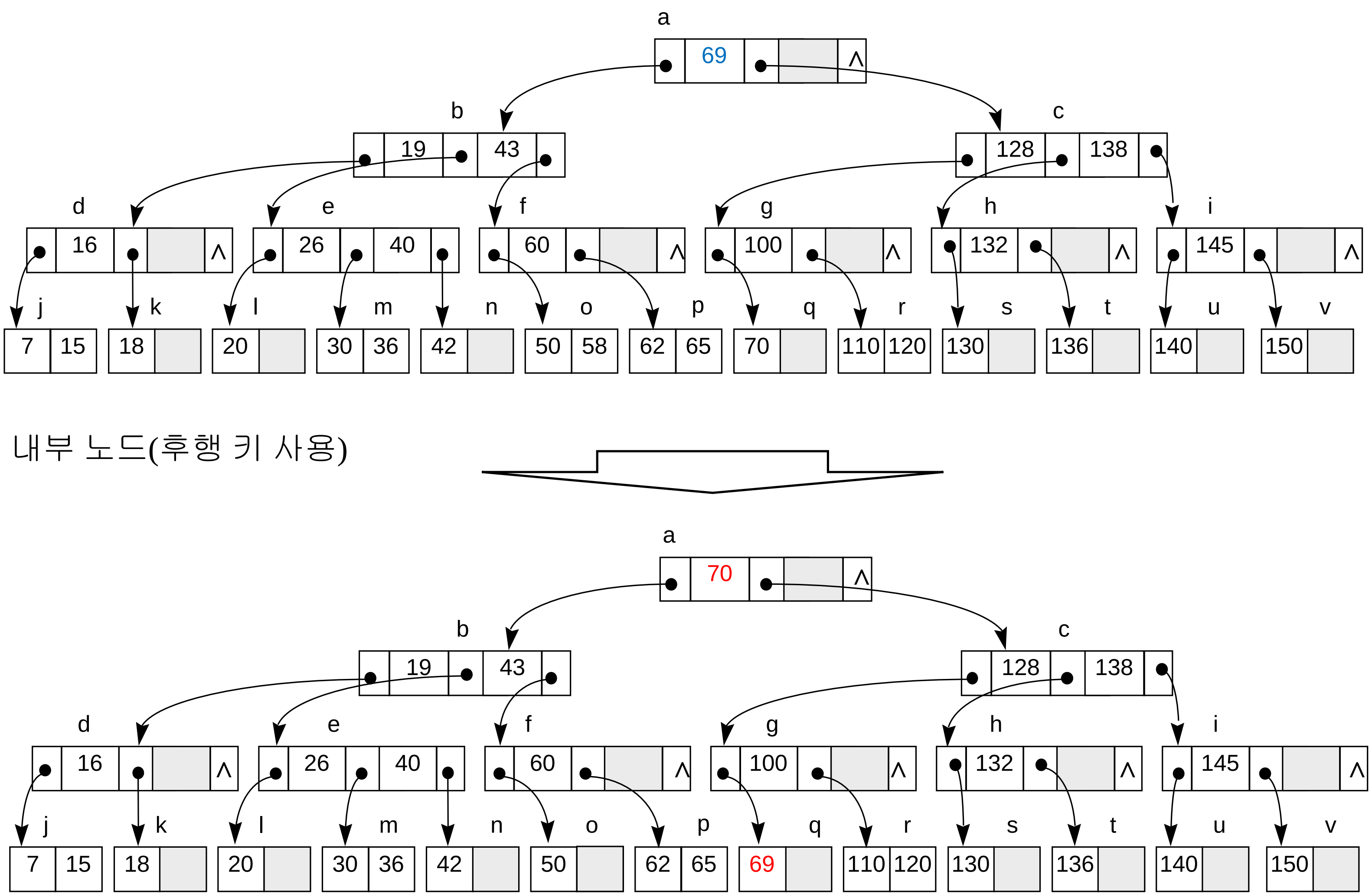
Underflow 발생시의 노드 합병 (Node merge) : 좌측

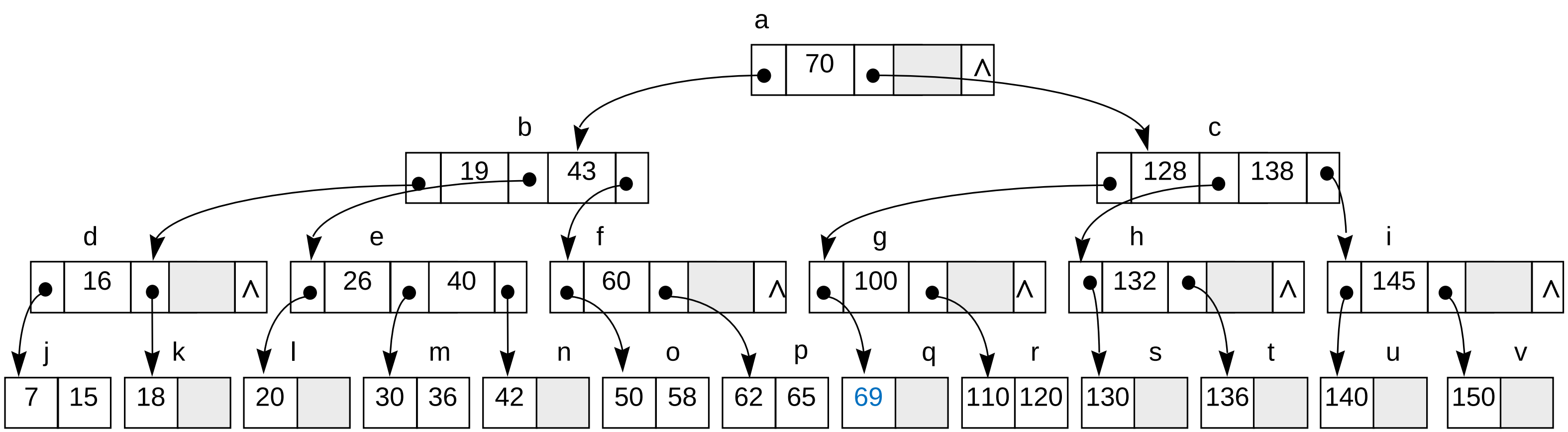


Best sibling의 선택

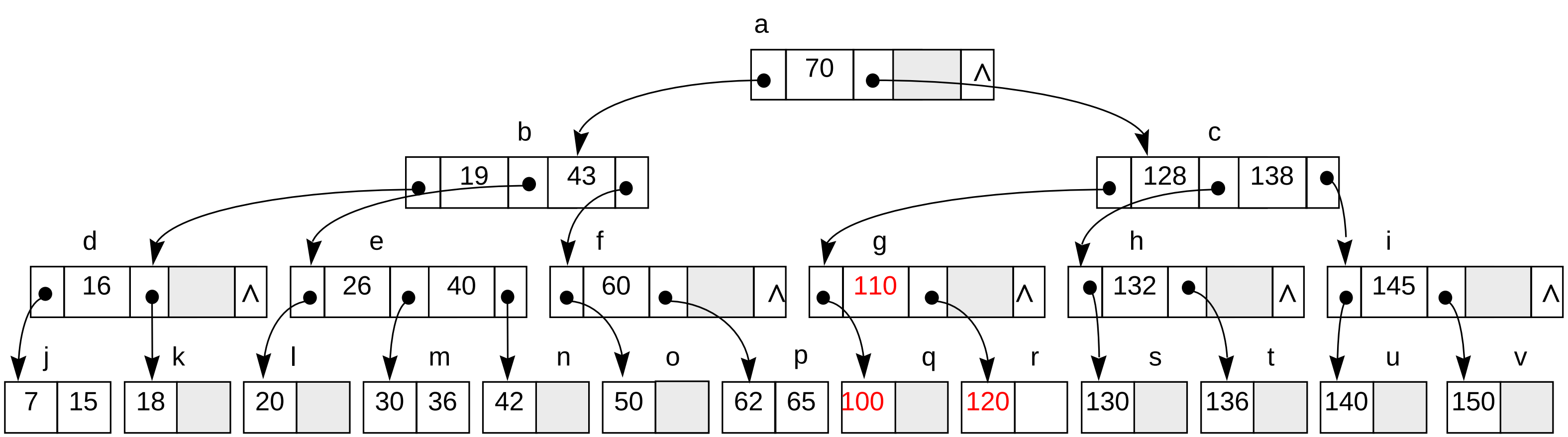
- o 오른쪽 및 왼쪽 형제 모두 사용 가능
  - – 가능하면 노드의 개수가 많은 쪽을 선택하는 것이 추후의 underflow 발생 회수를 줄이는데 도움이 됨.
- o 재분배의 경우
  - – Underflow가 아닌 쪽을 선택.
  - – 양쪽 모두 (거의) underflow 라면, 노드의 수가 많은 쪽을 선택
- o 노드 합병의 경우 (좌우가 모두 (거의) underflow 일 때)
  - – m이 홀수 : 좌우 모두 ⌈m/2⌉–1 이므로, 어느 것을 사용해도 무방
  - – m이 짝수 : 좌우가 각각 ⌈m/2⌉과 ⌈m/2⌉–1 이므로, ⌈m/2⌉인 쪽을 선택

예: 69 삭제

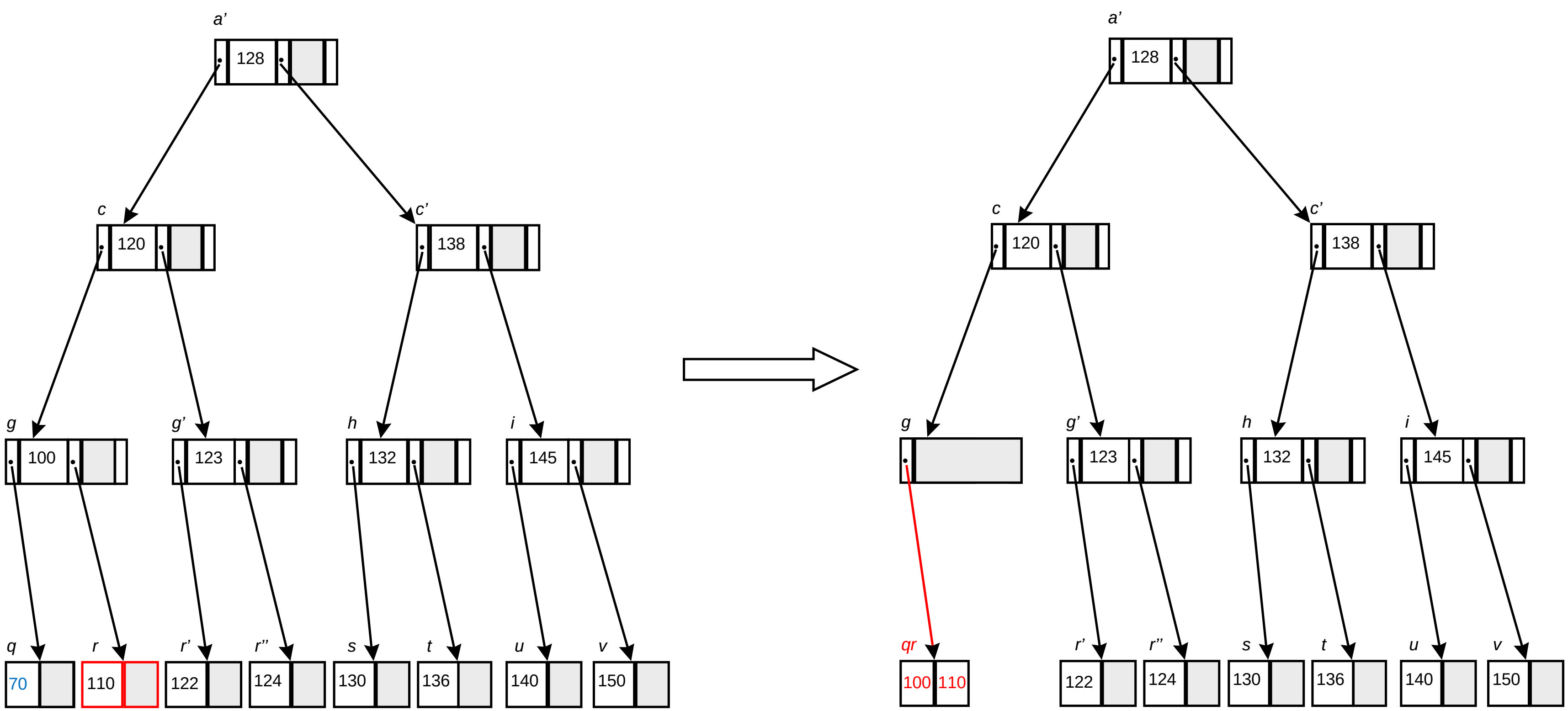

a
69
b
19
43
c
128
138
d
16
e
26
40
f
60
g
100
h
132
i
145
j
k
l
m
n
o
p
q
r
s
t
u
v
7
15
18
20
30
36
42
50
58
62
65
70
110
120
130
136
140
150
내부 노드(후행 키 사용)
a
70
b
19
43
c
128
138
d
16
e
26
40
f
60
g
100
h
132
i
145
j
k
l
m
n
o
p
q
r
s
t
u
v
7
15
18
20
30
36
42
50
62
65
69
110
120
130
136
140
150


a
70
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
7
15
18
20
30
36
42
50
58
62
65
69
110
120
130
136
140
150

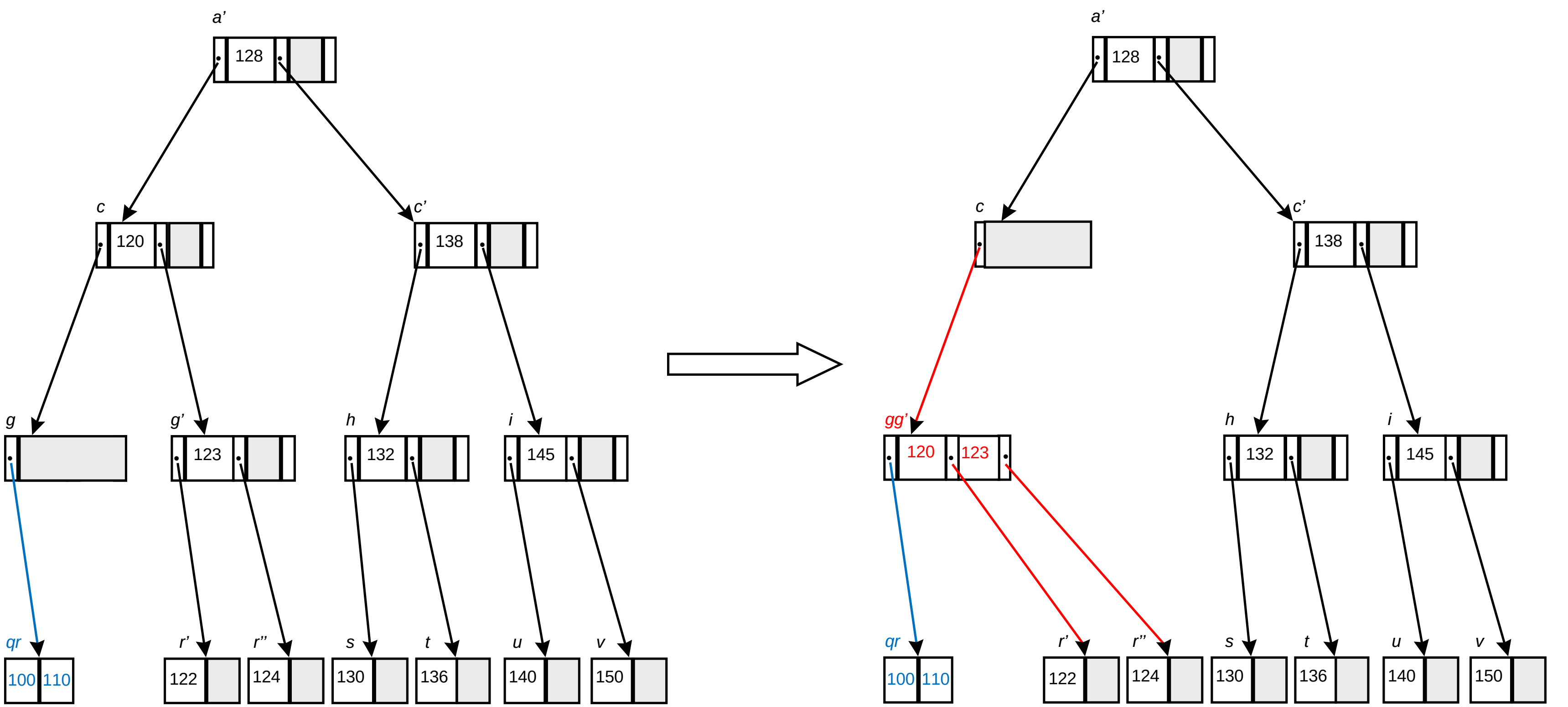
키 재분배(형제 노드 사용)


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

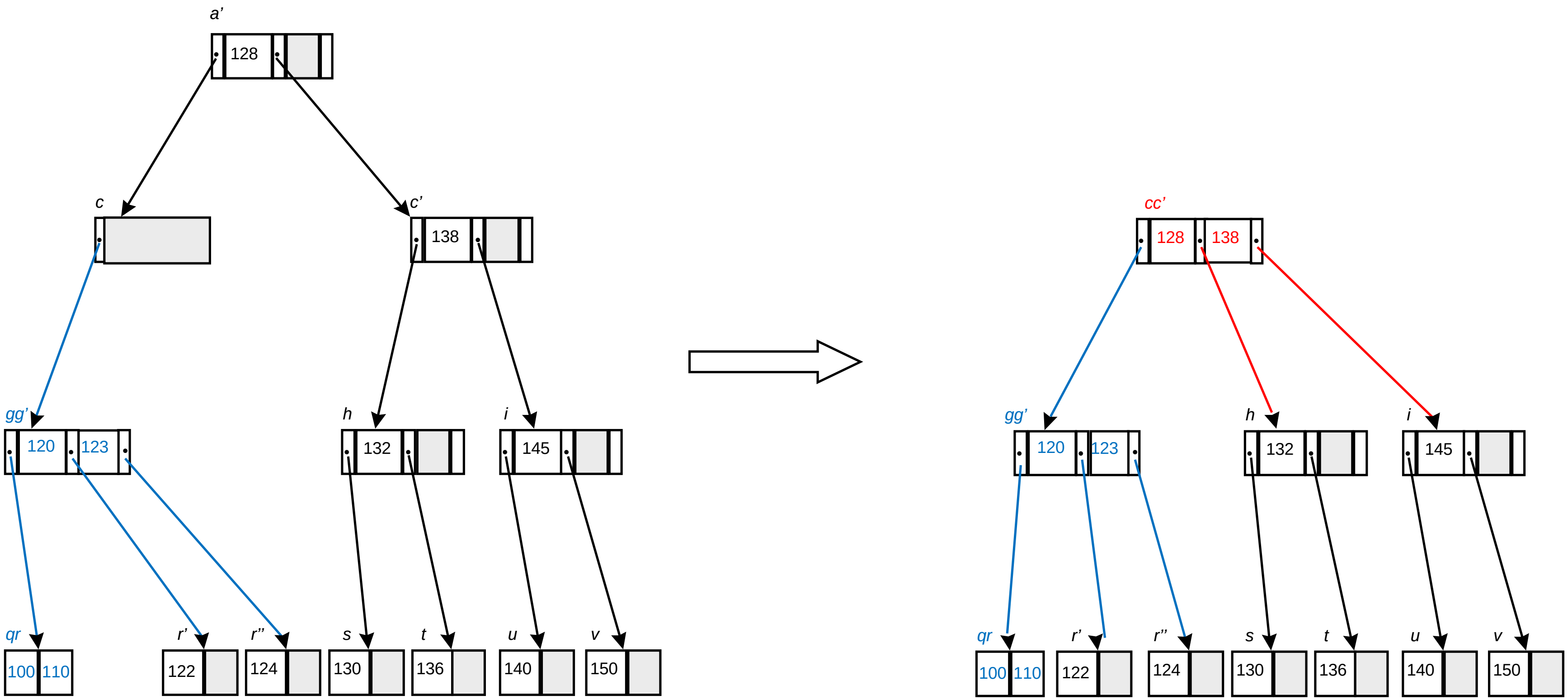
예: 70 삭제



노드 합병
(형제노드 사용 불가능)



노드 합병
(후행키, 형제노드 사용 불가능)

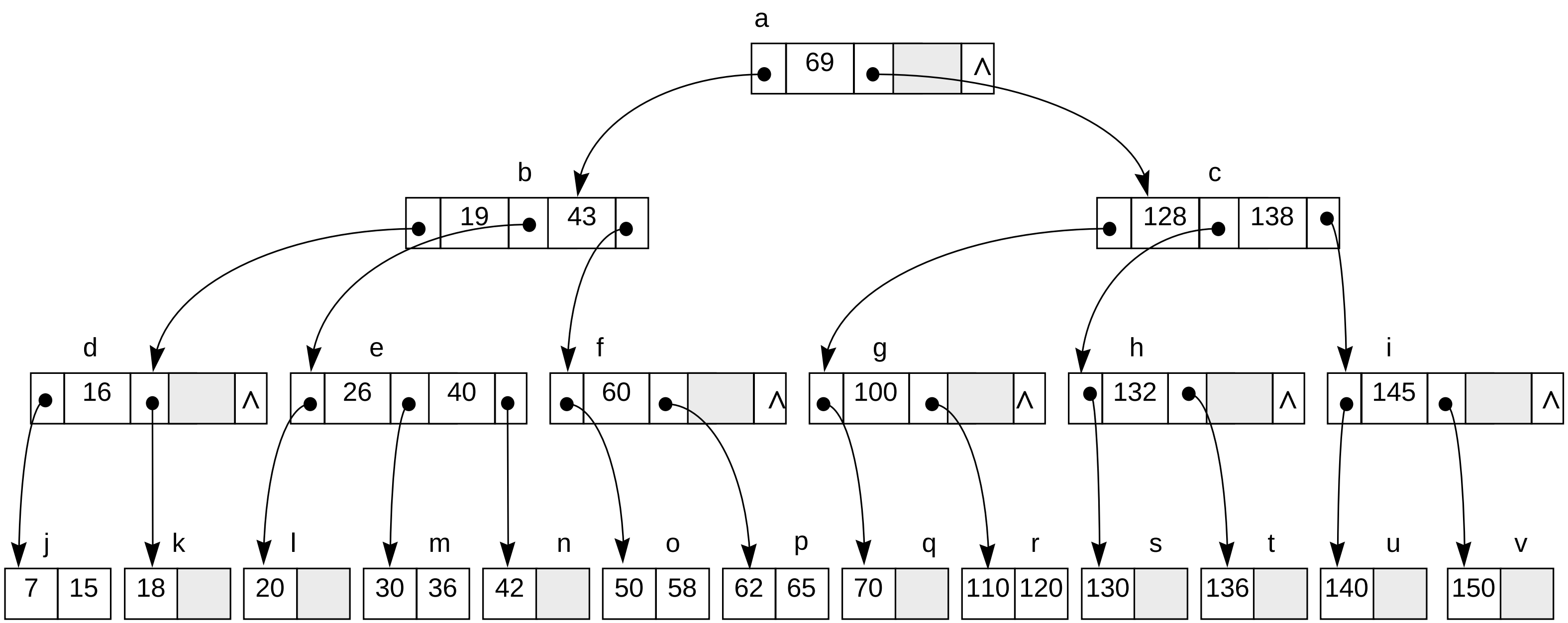


노드 합병
(후행키, 형제노드 사용 불가능)

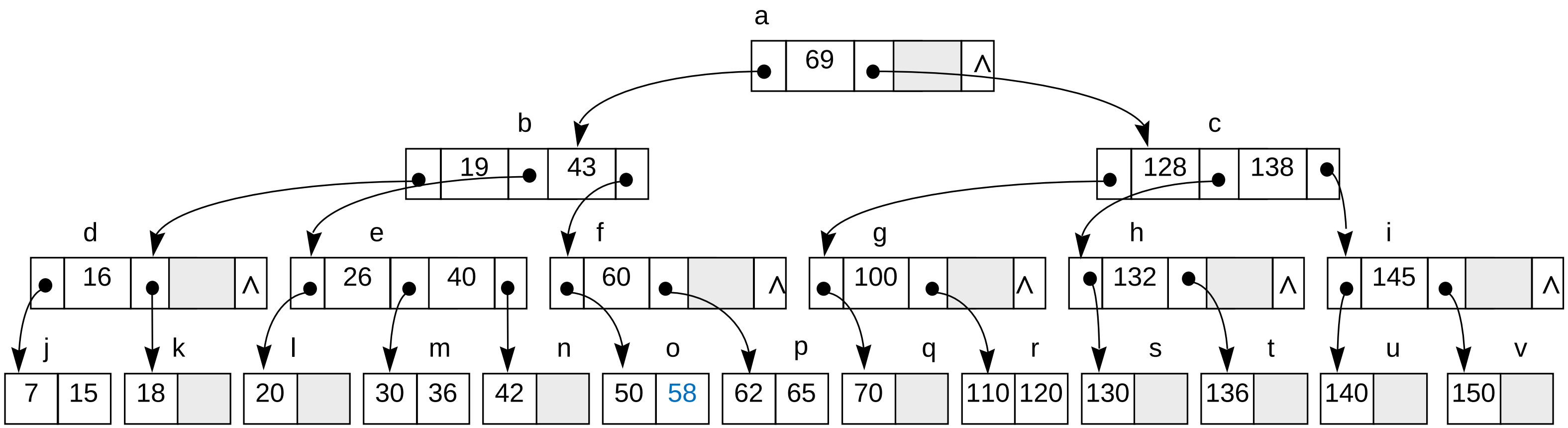
루트 레벨 하나 감소

예제 2

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

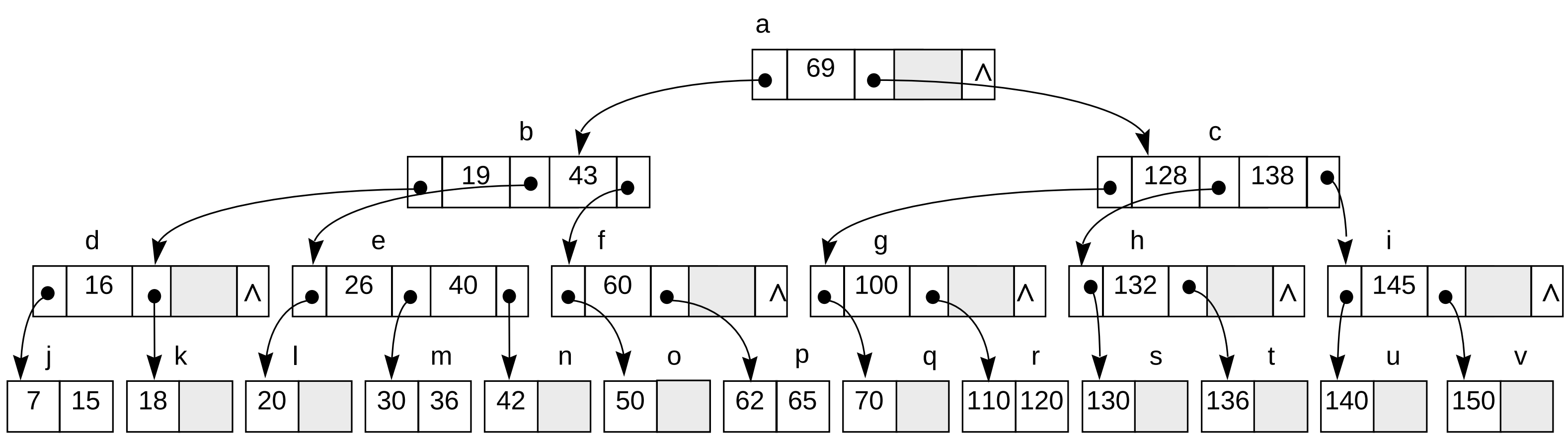


58 삭제



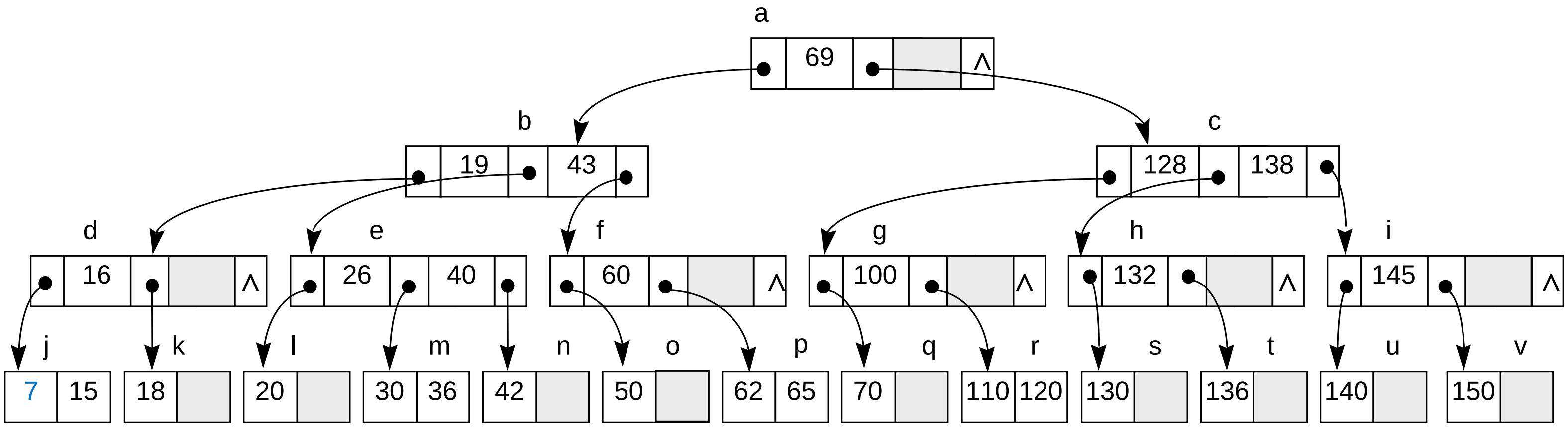
a
69
b
19
43
c
128
138
d
16
e
26
40
f
60
g
100
h
132
i
145
j
k
l
m
n
o
p
q
r
s
t
u
v
7
15
18
20
30
36
42
50
58
62
65
70
110
120
130
136
140
150

단말에서 삭제



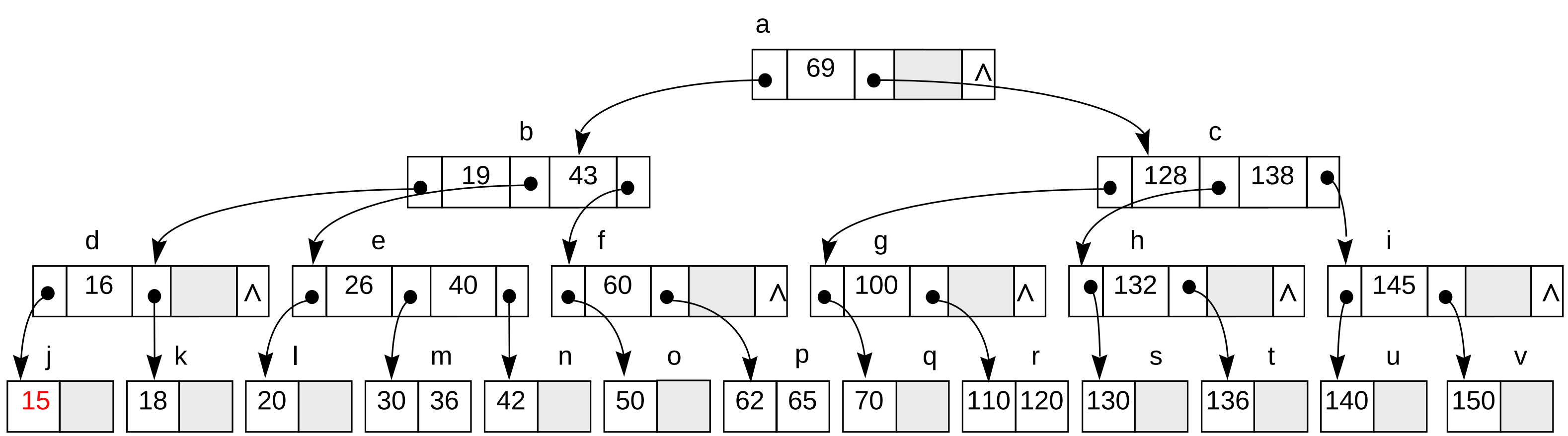
a
69
b
19
43
c
128
138
d
16
e
26
40
f
60
g
100
h
132
i
145
j
k
l
m
n
o
p
q
r
s
t
u
v
7
15
18
20
30
36
42
50
62
65
70
110
120
130
136
140
150

7 삭제



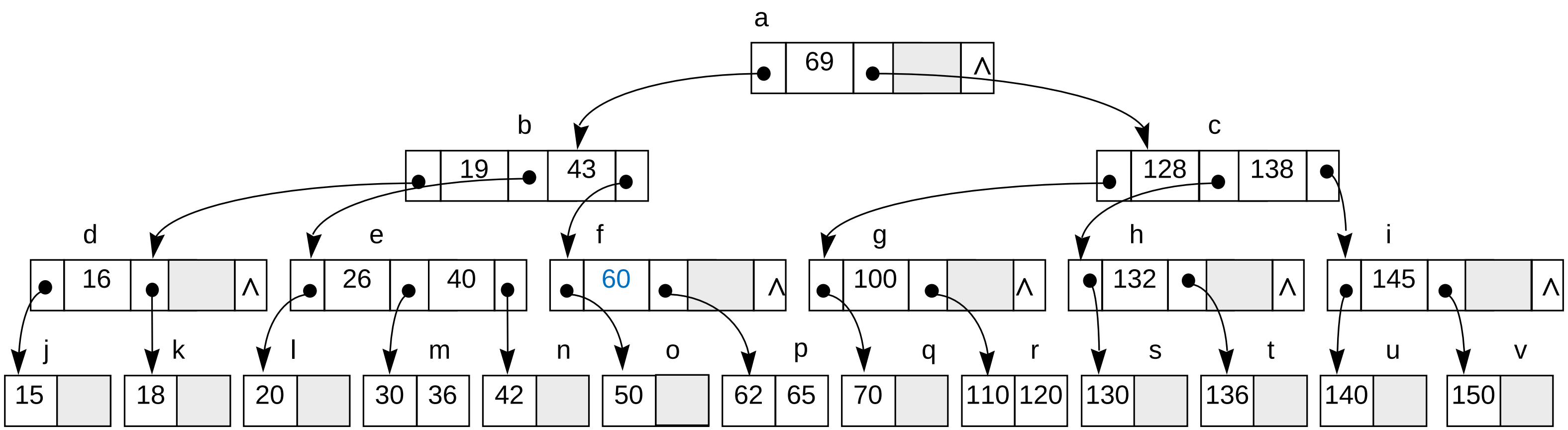
a
69
b
19
43
c
128
138
d
16
e
26
40
f
60
g
100
h
132
i
145
j
7
15
k
18
l
20
m
30
36
n
42
o
50
p
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150

단말에서 삭제

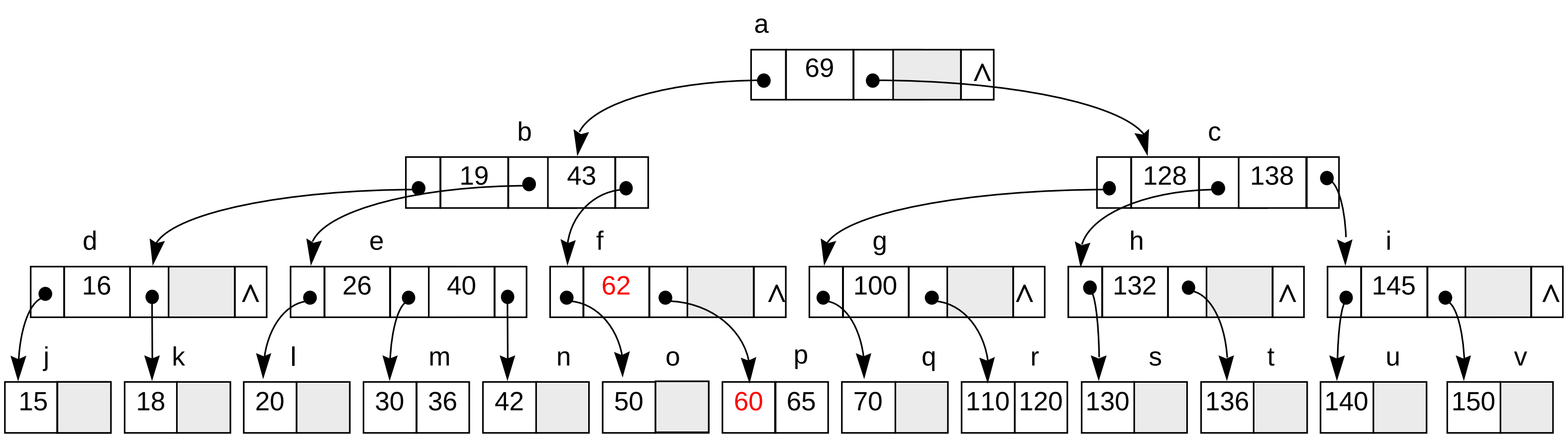


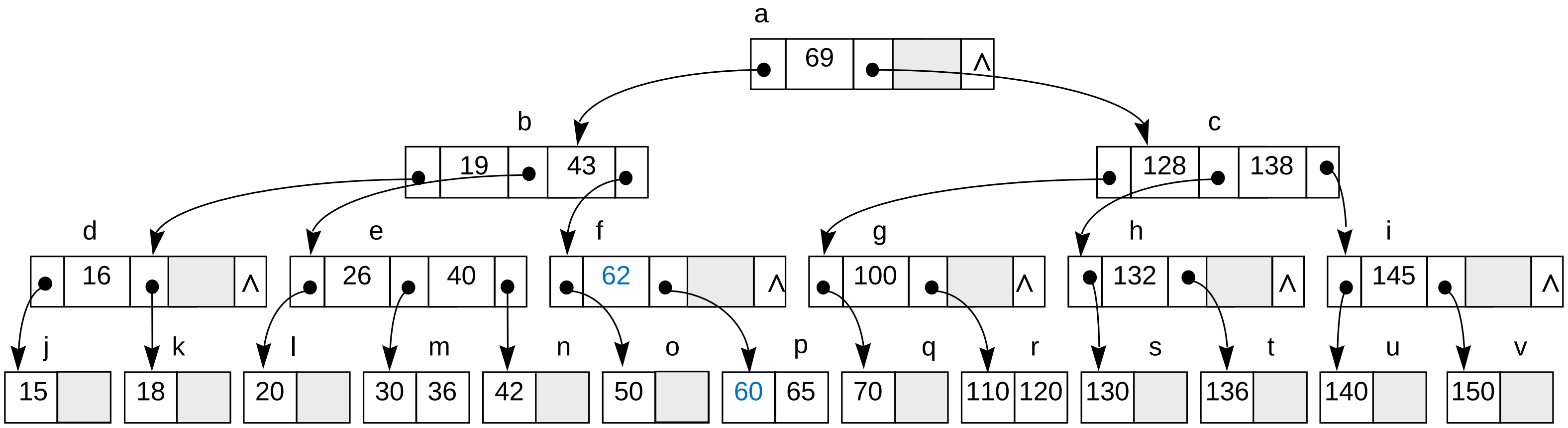
a
69
b
19
43
c
128
138
d
16
e
26
40
f
60
g
100
h
132
i
145
j
15
k
18
l
20
m
30
36
n
42
o
50
p
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150

60 삭제



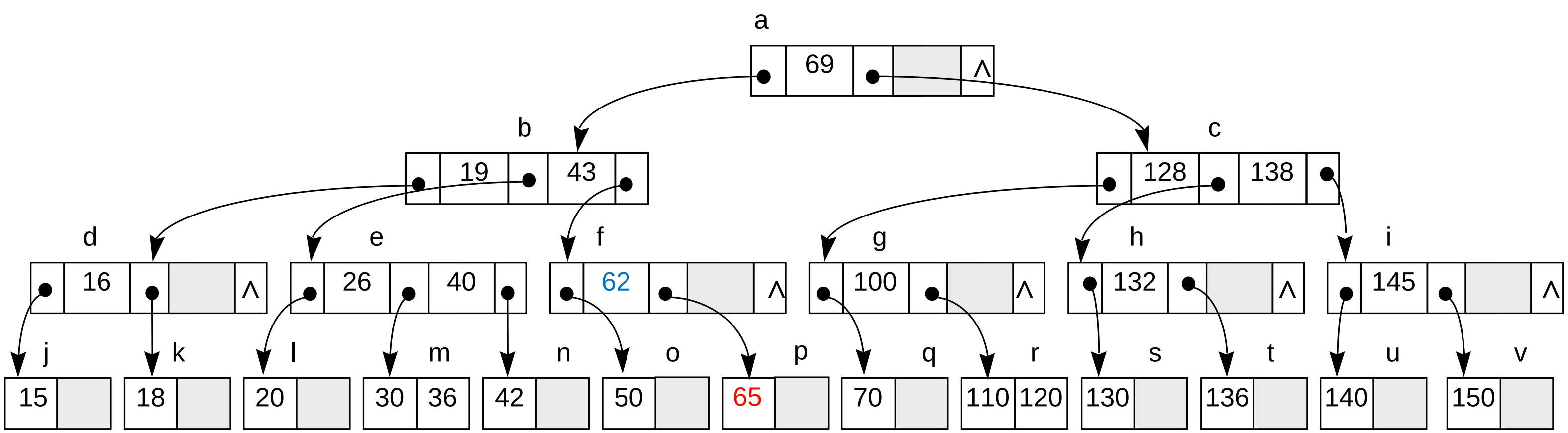
내부 노드(후행 키 사용)





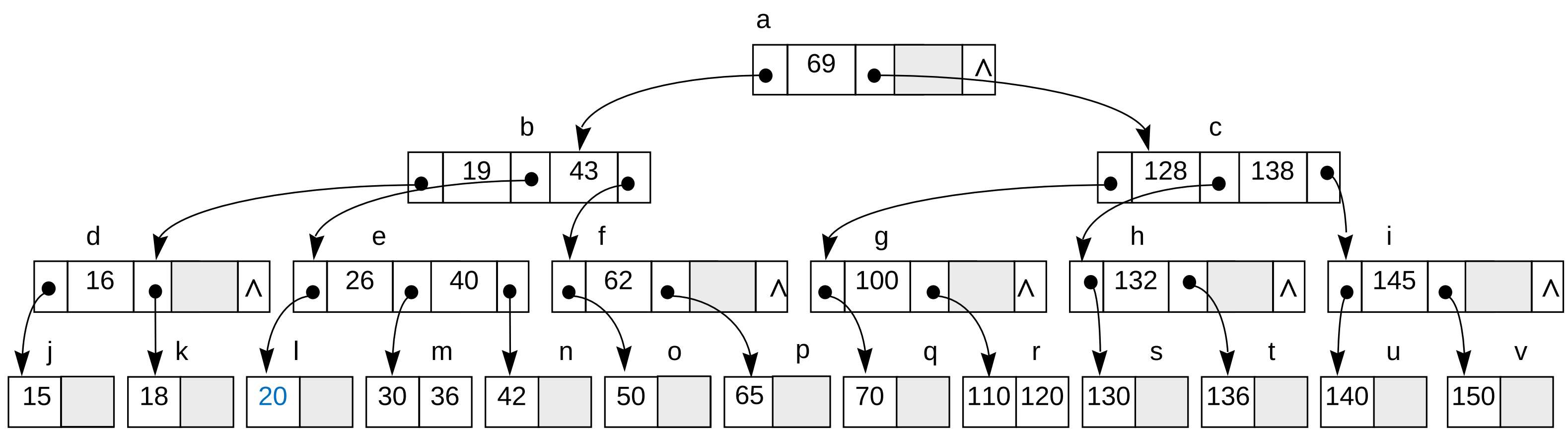
a
69
b
19
43
c
128
138
d
16
e
26
40
f
62
g
100
h
132
i
145
j
15
k
18
l
20
m
30
36
n
42
o
50
p
60
65
q
70
r
110
120
s
130
t
136
u
140
v
150

단말에서 삭제

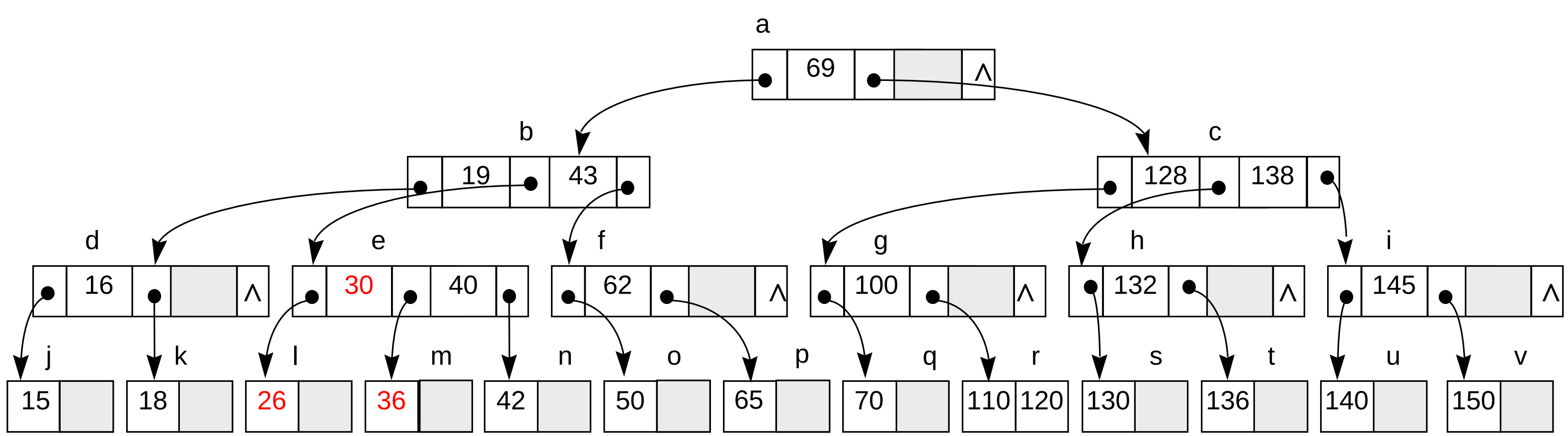


a
69
b
19
43
c
128
138
d
16
e
26
40
f
62
g
100
h
132
i
145
j
15
k
18
l
20
m
30
36
n
42
o
50
p
65
q
70
r
110
120
s
130
t
136
u
140
v
150

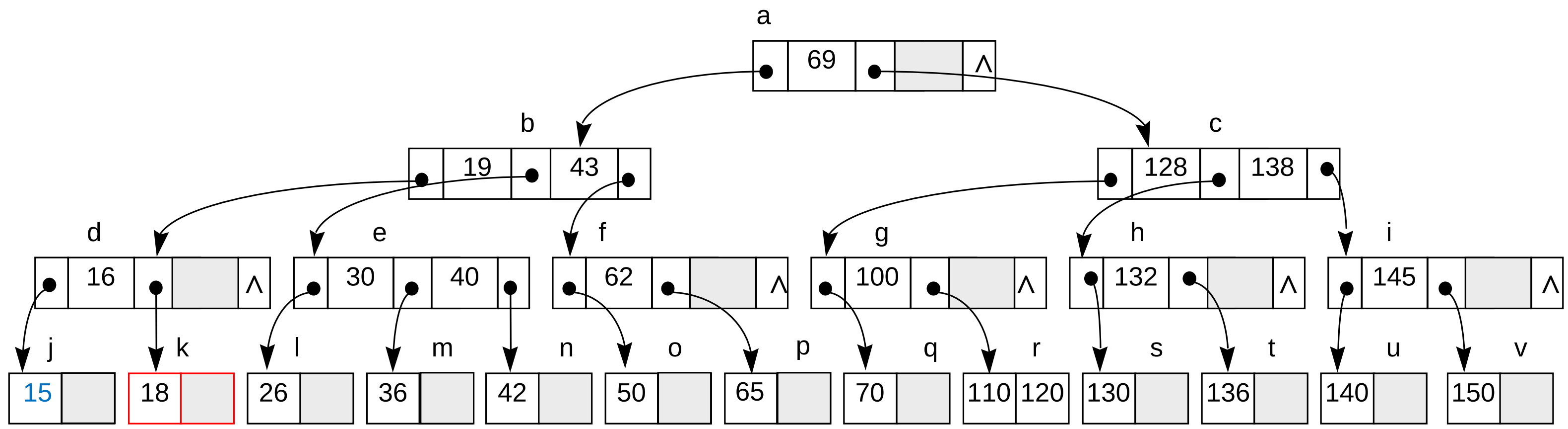
20 삭제



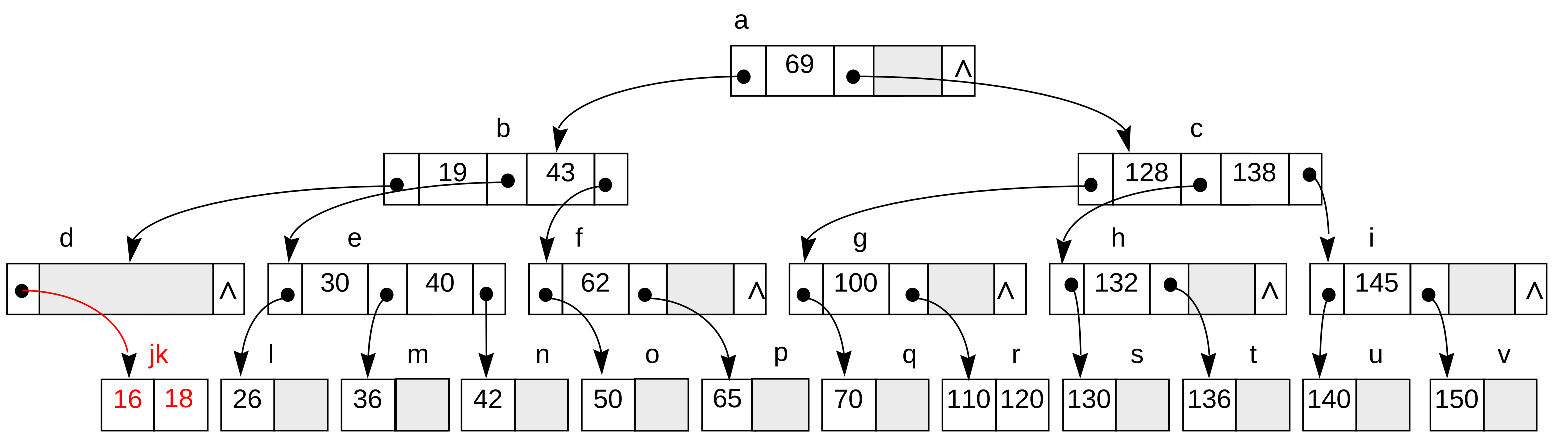
키 재분배(형제 노드 사용)

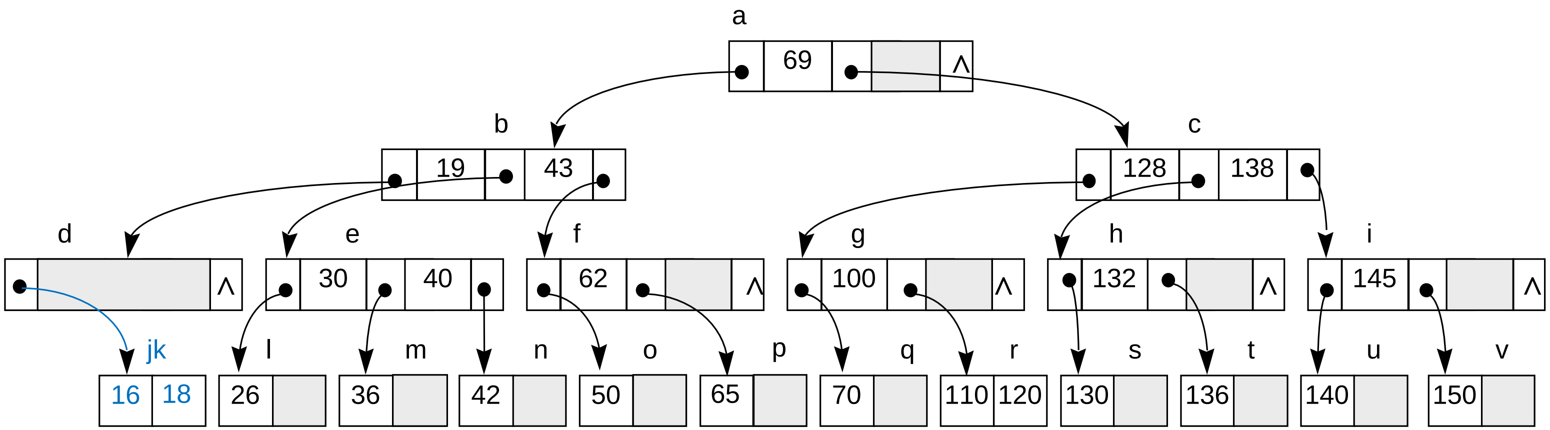


15 삭제



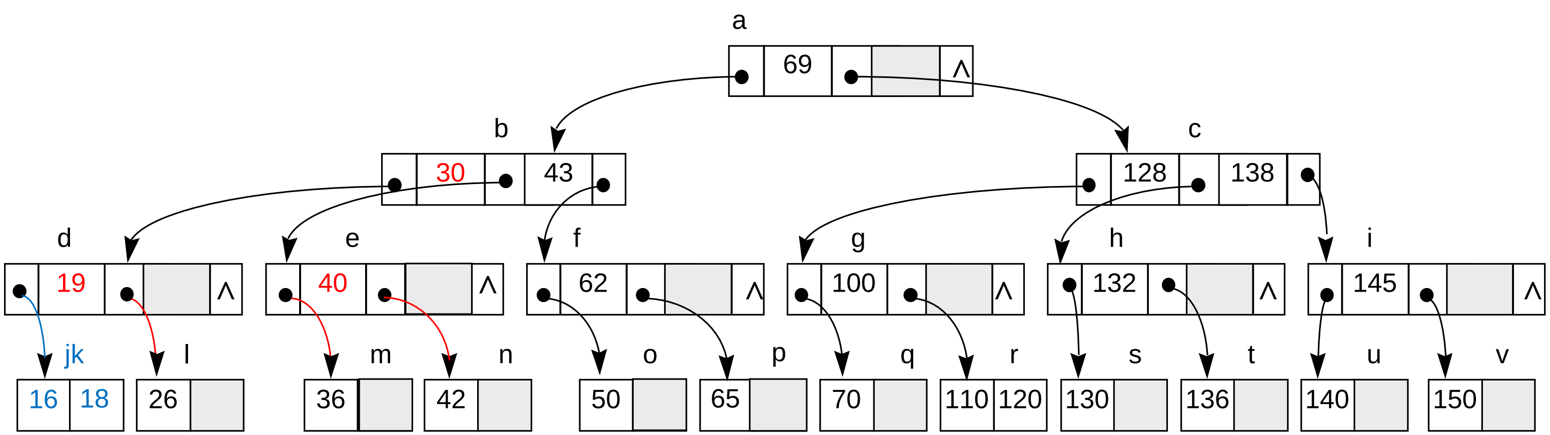
노드 합병 (형제 노드 사용 불가)





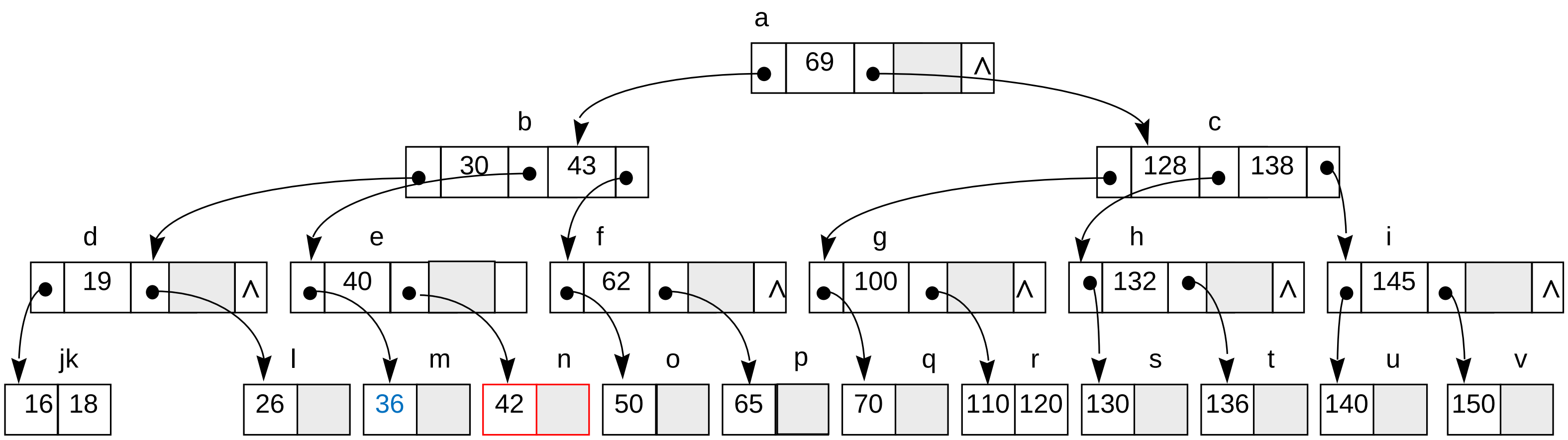
a
69
b
19
43
c
128
138
d
e
30
40
f
62
g
100
h
132
i
145
jk
16
18
l
26
m
36
n
42
o
50
p
65
q
70
r
110
120
s
130
t
136
u
140
v
150

키 재분배(형제 노드 사용)

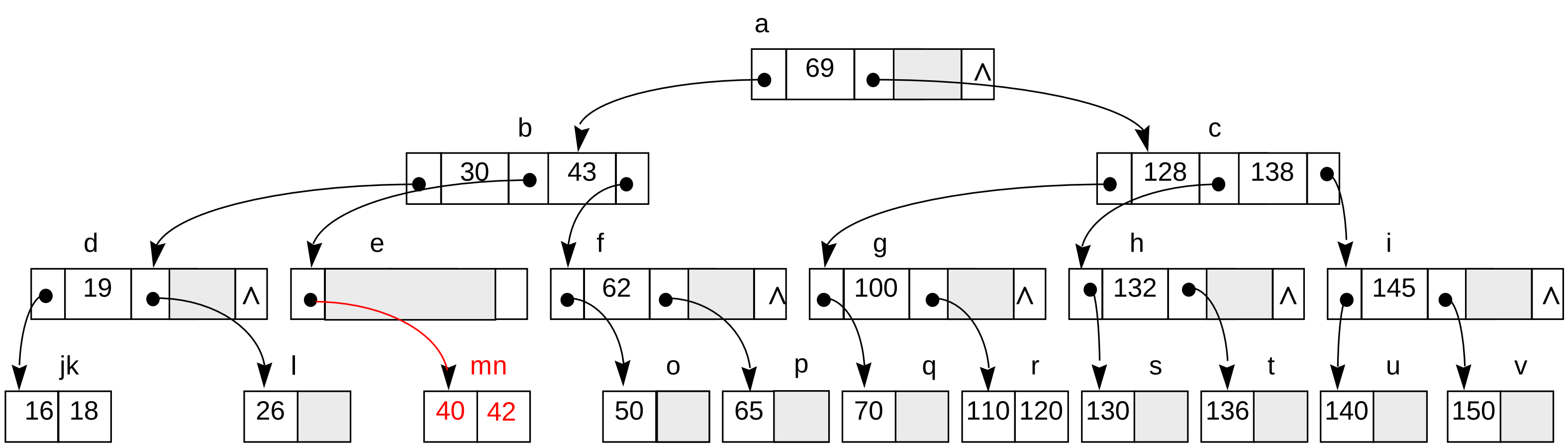


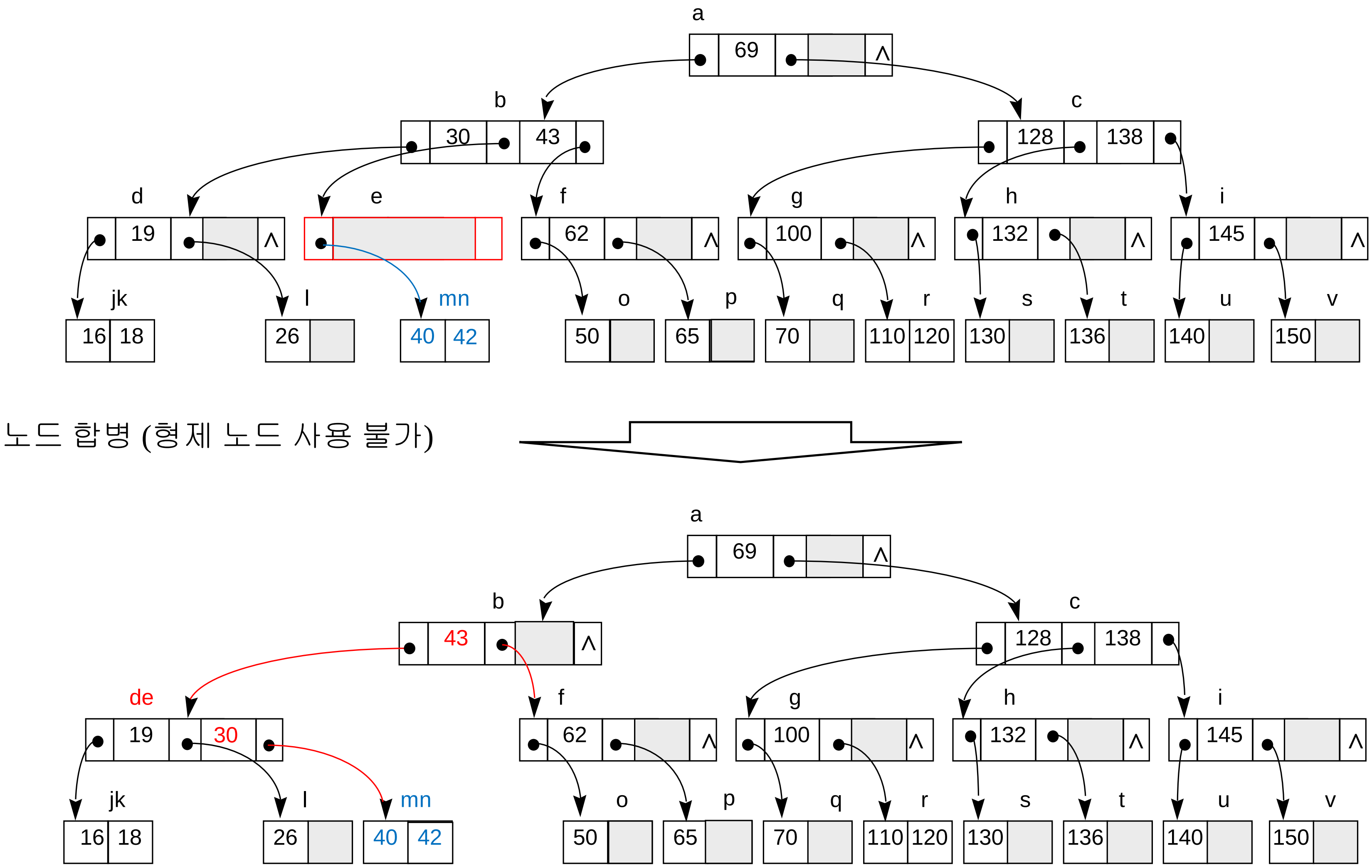
a
69
b
30
43
c
128
138
d
19
e
40
f
62
g
100
h
132
i
145
jk
16
18
l
26
m
36
n
42
o
50
p
65
q
70
r
110
120
s
130
t
136
u
140
v
150

36 삭제

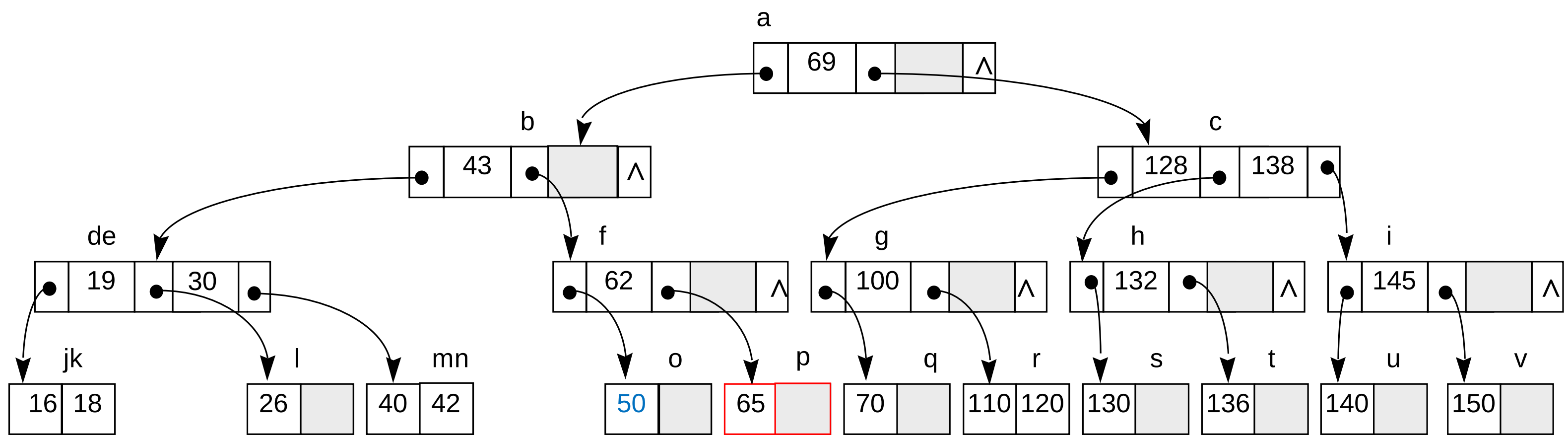


노드 합병 (형제 노드 사용 불가)

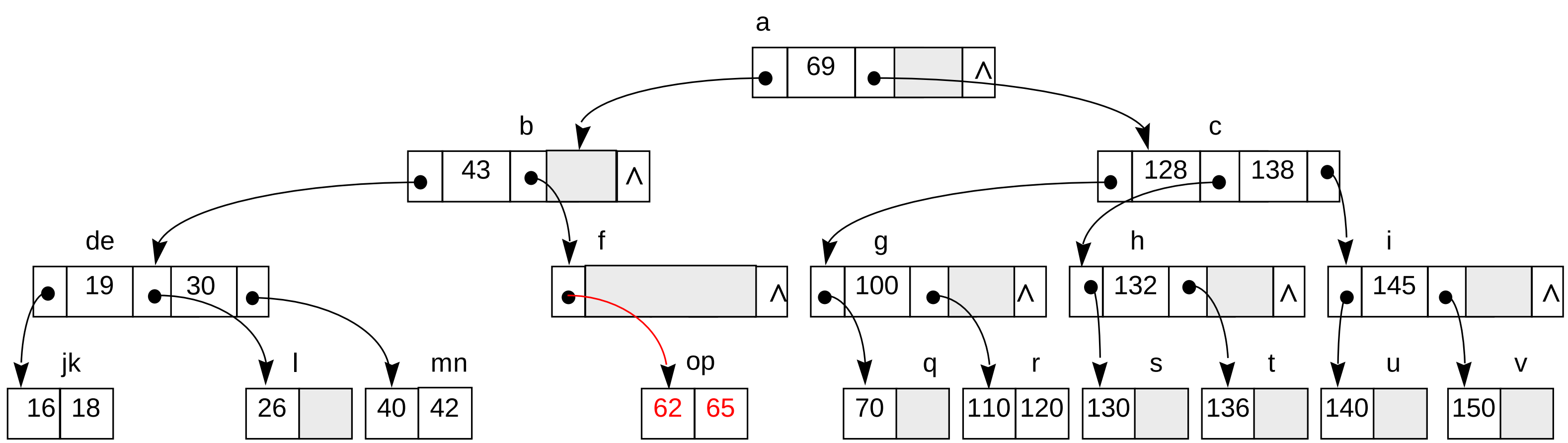


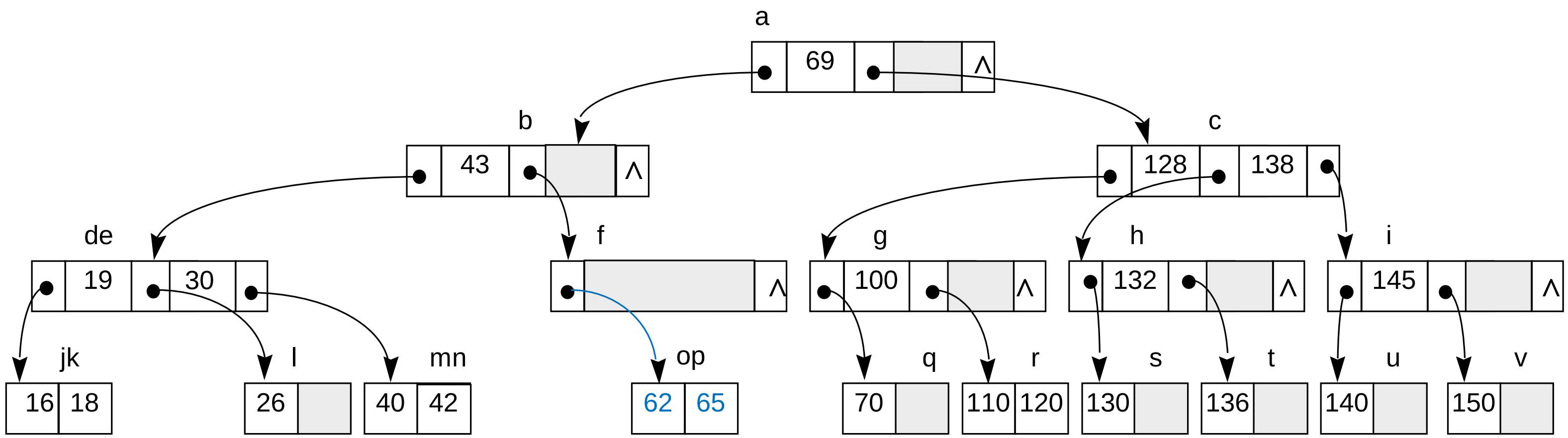

a
69
b
30
43
c
128
138
d
19
e
f
62
g
100
h
132
i
145
jk
16
18
l
26
mn
40
42
o
50
p
65
q
70
r
110
120
s
130
t
136
u
140
v
150
노드 합병 (형제 노드 사용 불가)
a
69
b
43
c
128
138
de
19
30
f
62
g
100
h
132
i
145
jk
16
18
l
26
mn
40
42
o
50
p
65
q
70
r
110
120
s
130
t
136
u
140
v
150

50 삭제

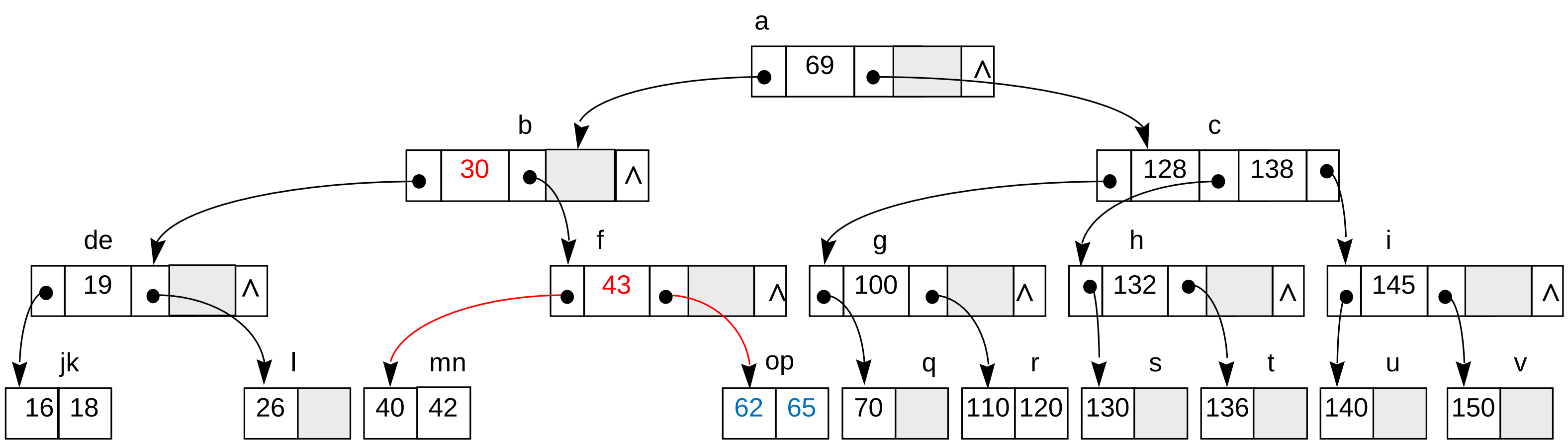


노드 합병 (형제 노드 사용 불가)

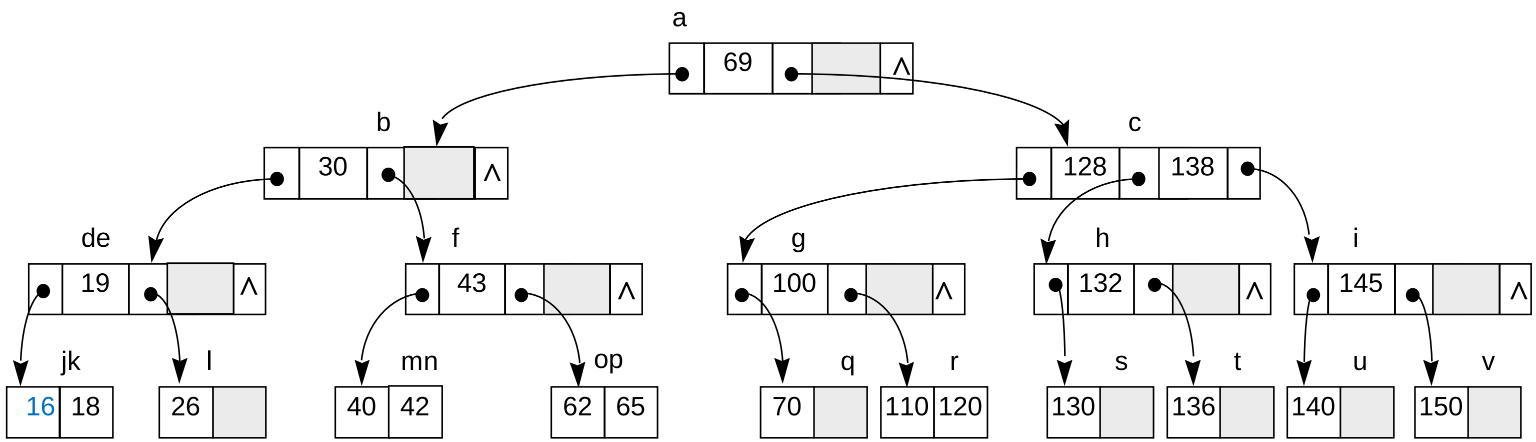



a
69
b
43
c
128 138
de
19 30
f
g
100
h
132
i
145
jk
16 18
l
26
mn
40 42
op
62 65
q
70
r
110 120
s
130
t
136
u
140
v
150

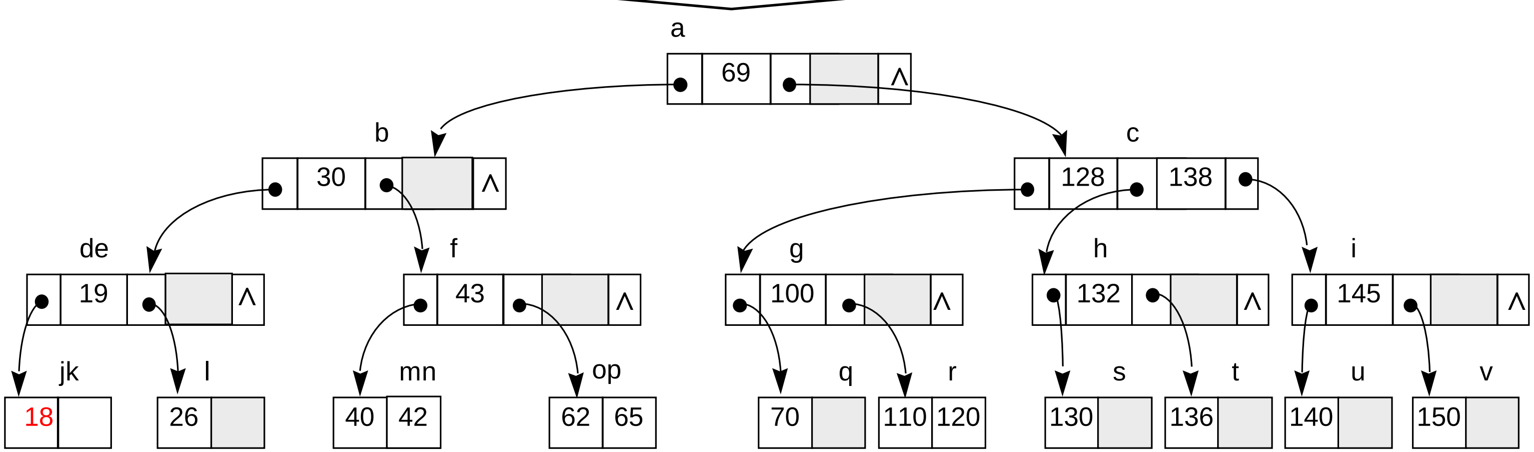
키 재분배(형제 노드 사용)


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

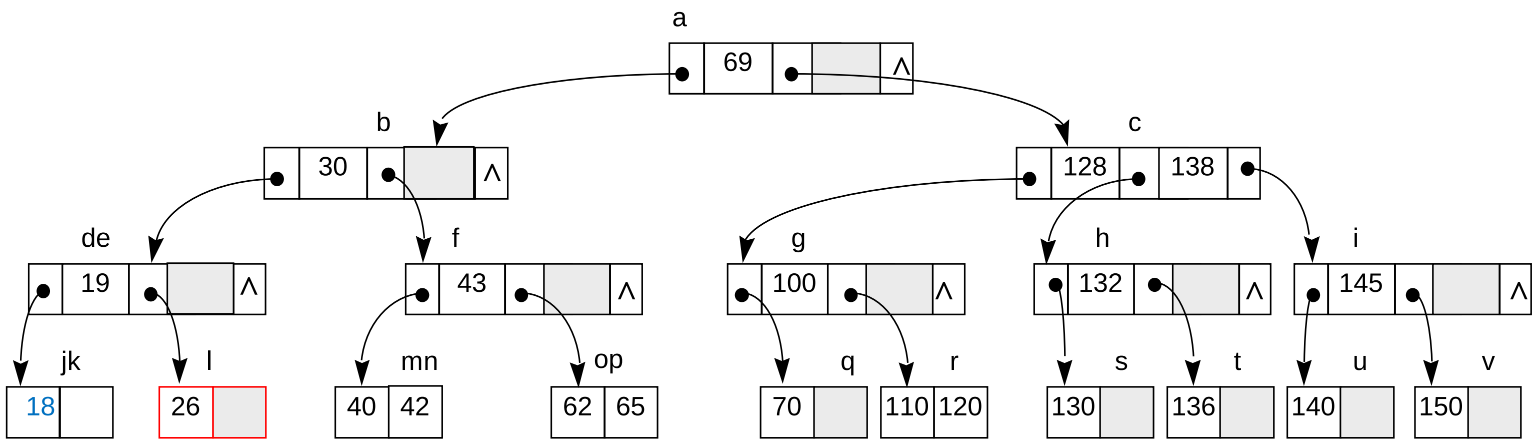
16 삭제


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

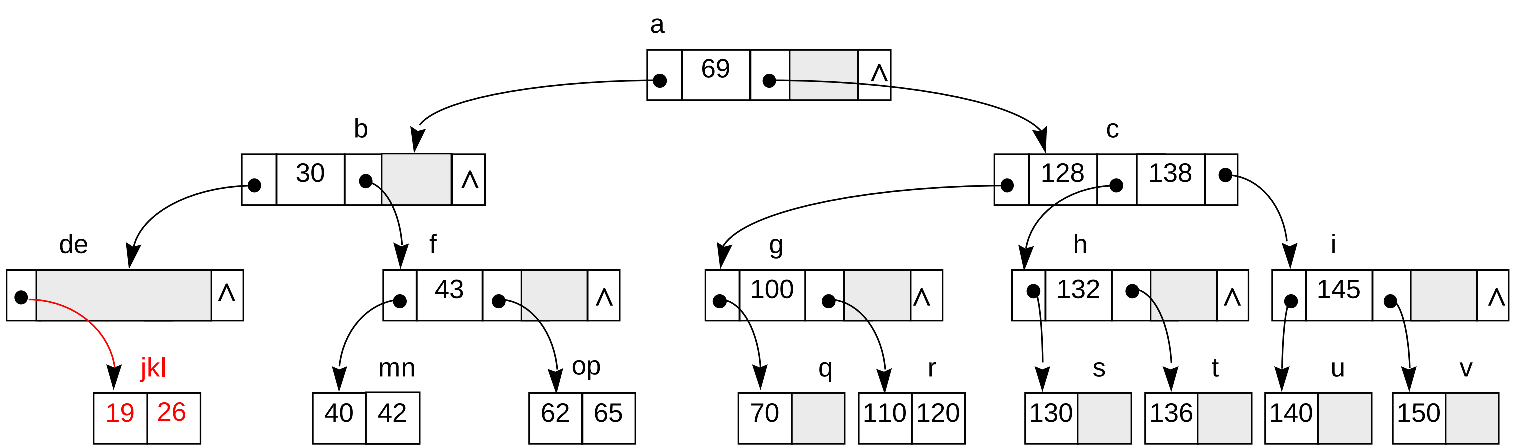
단말에서 삭제

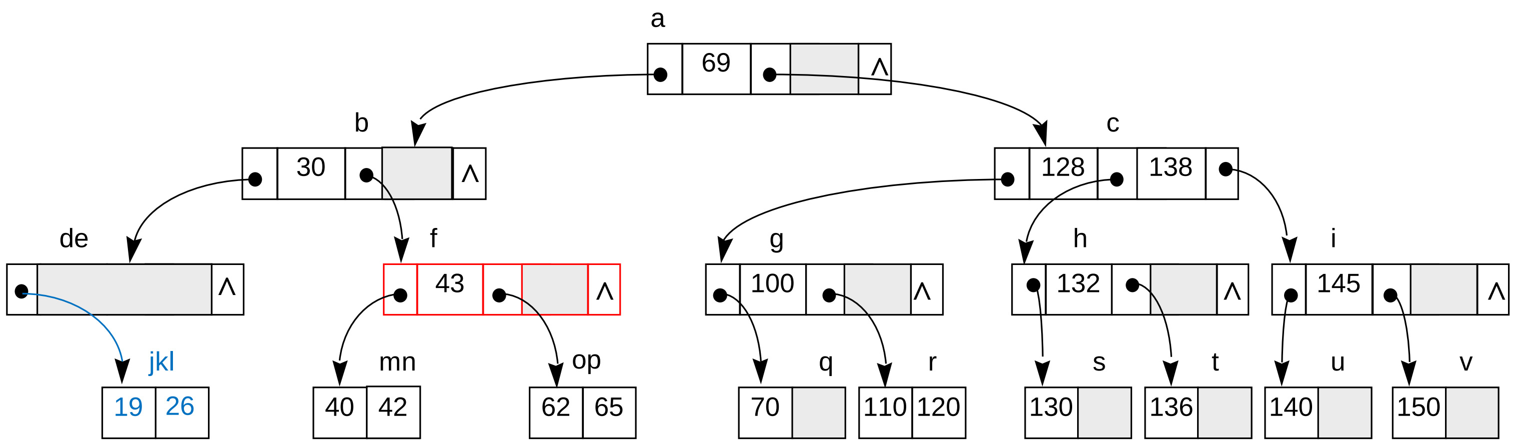

a
69
b
30
c
128
138
de
19
f
43
g
100
h
132
i
145
jk
18
l
26
mn
40
42
op
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150

18 삭제

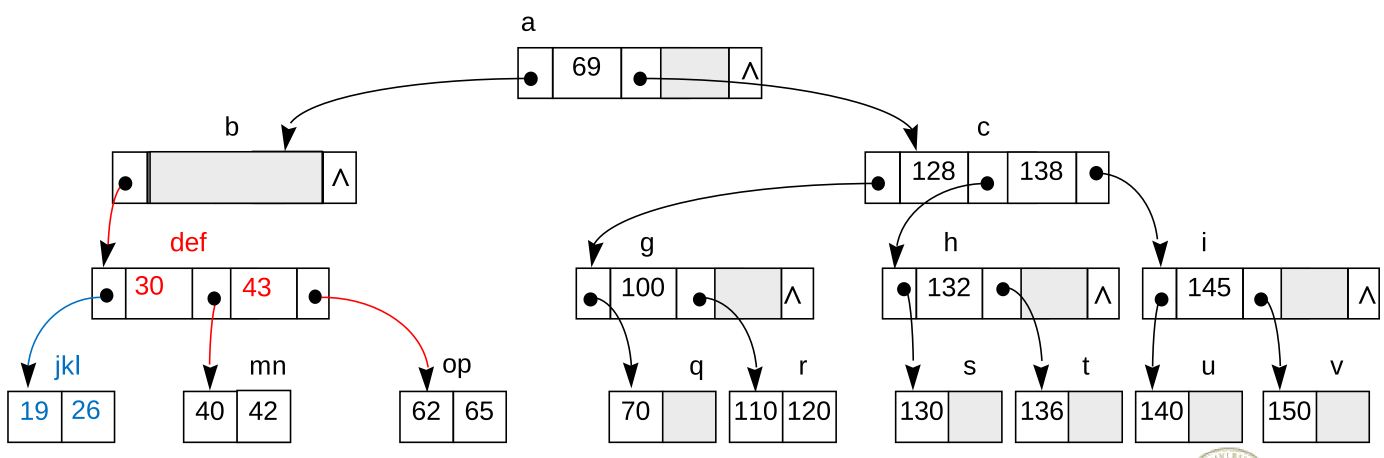


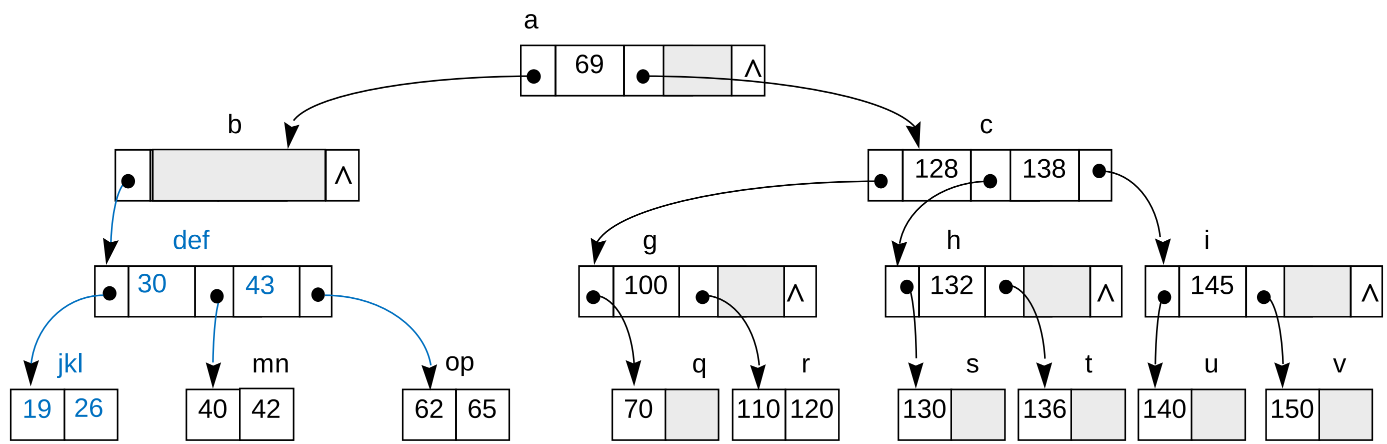
노드 합병 (형제 노드 사용 불가)



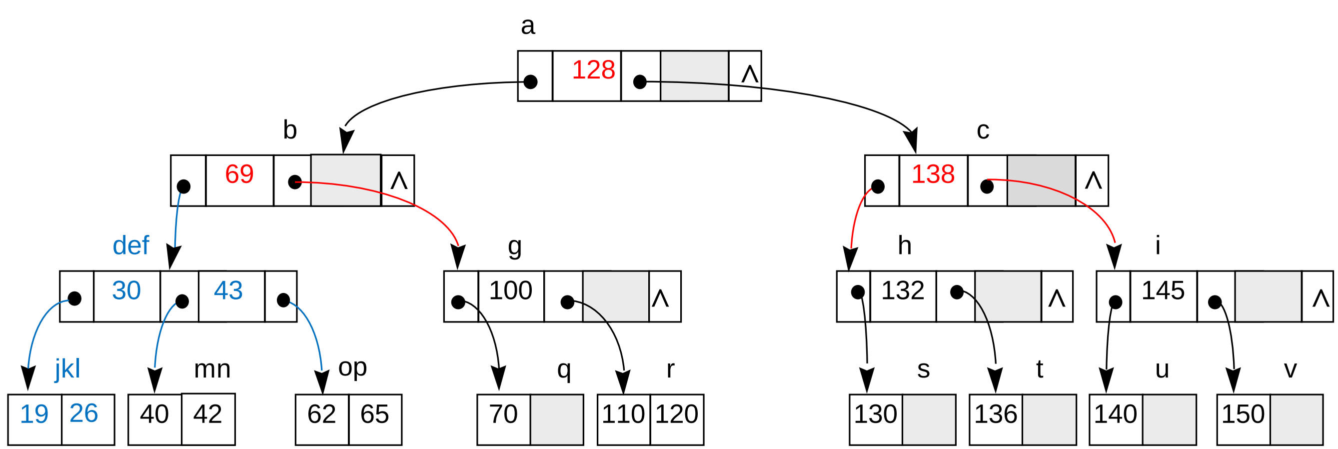


노드 합병 (형제 노드 사용 불가)

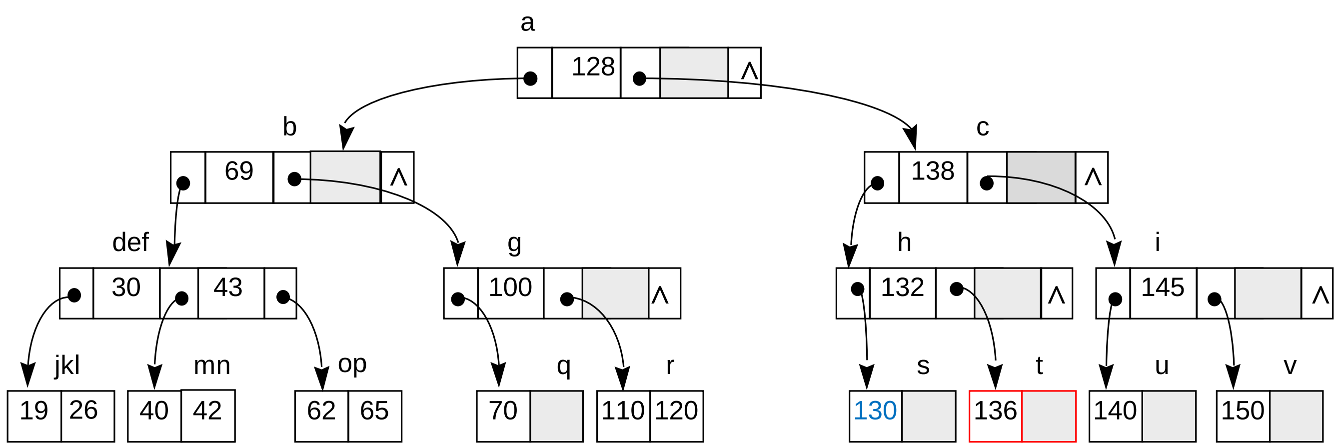



a
69
b
c
128
138
def
30
43
g
100
h
132
i
145
jkl
19
26
mn
40
42
op
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150

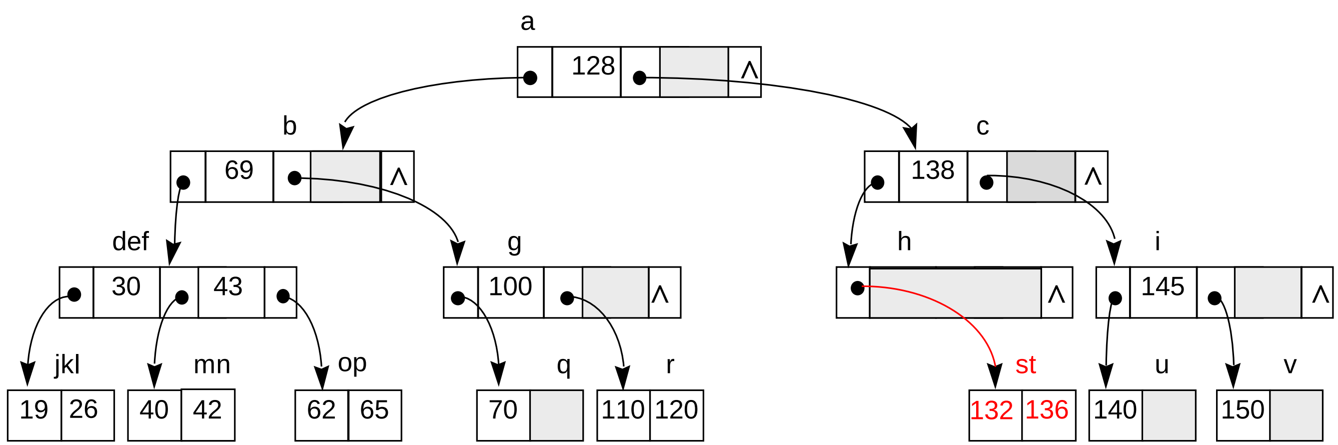
키 재분배(형제 노드 사용)

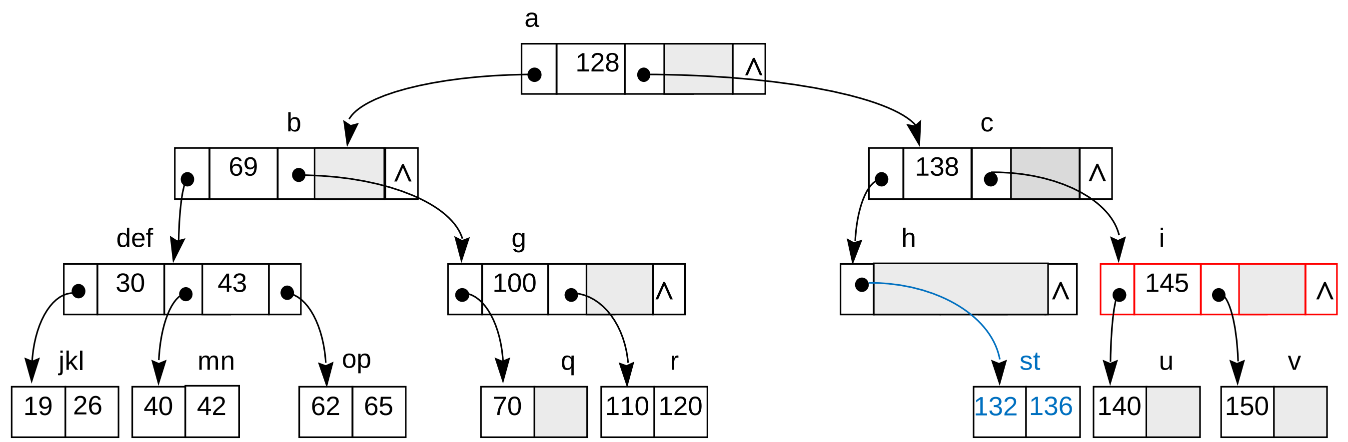

a
128
b
69
c
138
def
30
43
g
100
h
132
i
145
jkl
19
26
mn
40
42
op
62
65
q
70
r
110
120
s
130
t
136
u
140
v
150

130 삭제

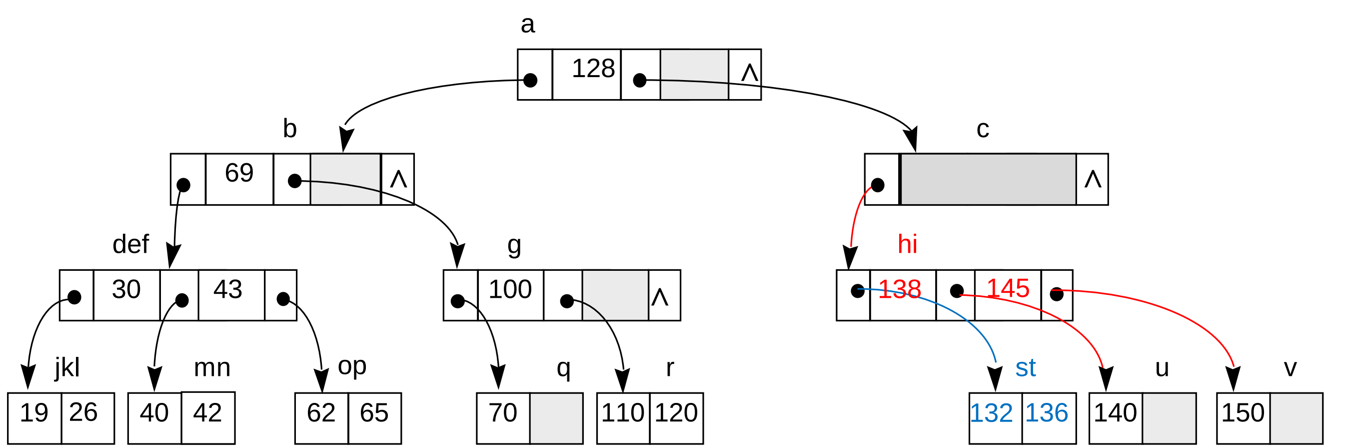


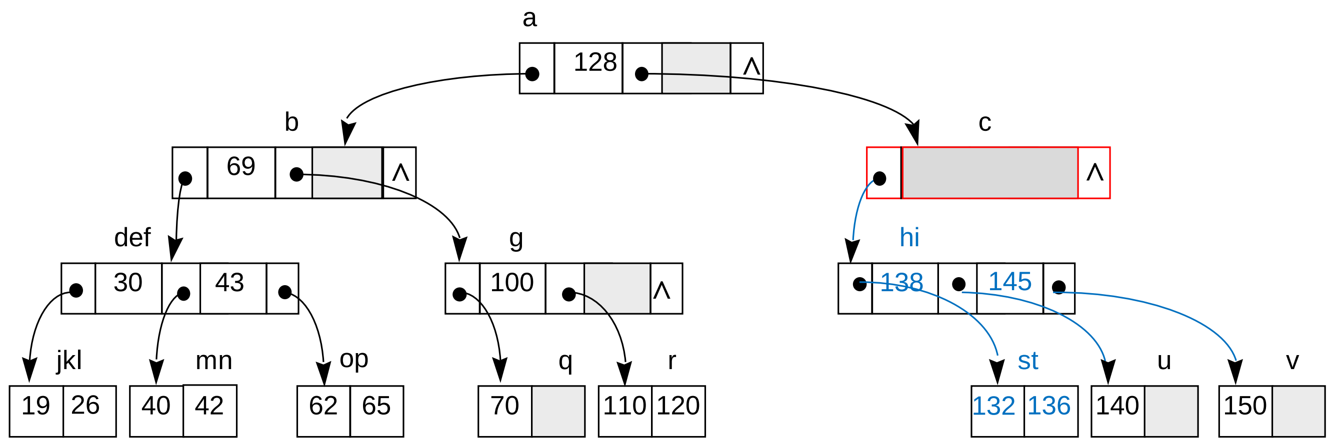
노드 합병 (형제 노드 사용 불가)



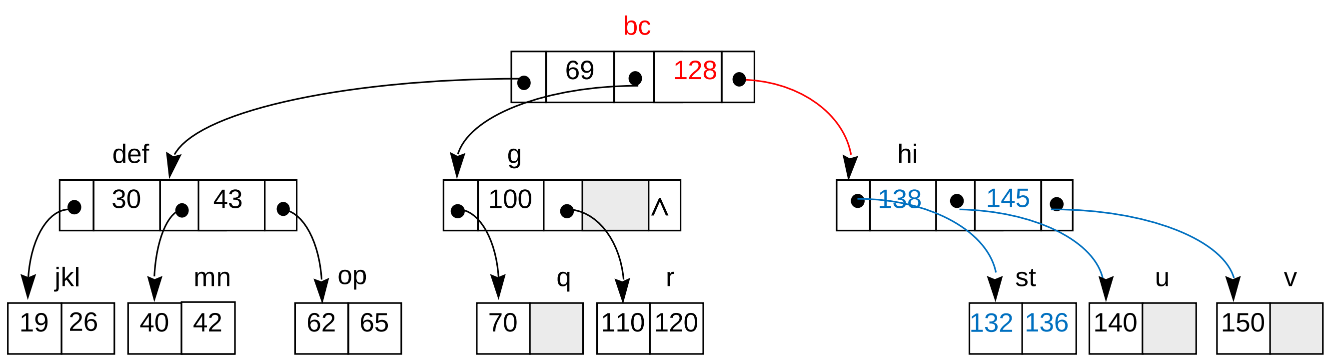


노드 합병 (형제 노드 사용 불가)



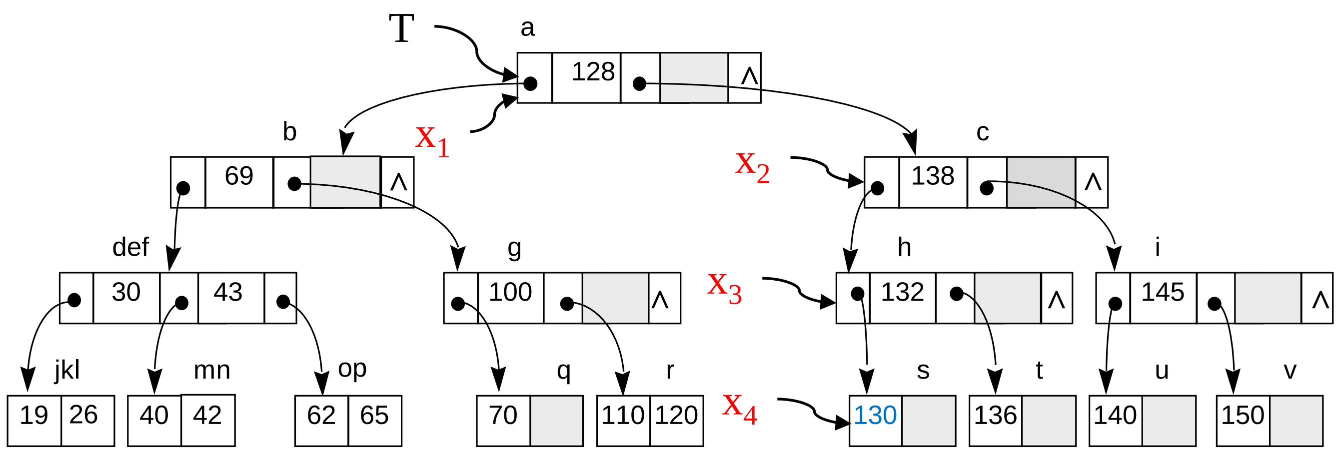


노드 합병 (형제 노드 사용 불가)

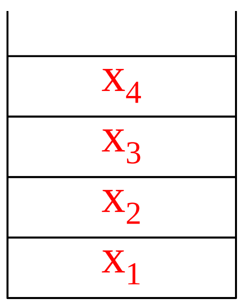


B-트리 삭제 알고리즘의 개요

deleteBT(T, 3, 130)



$x_5 \leftarrow$null (삭제할 키가 존재하지 않음)



deleteKey(3, x_4, 130)
deleteKey(3, x_3, 132)
deleteKey(3, x_2, 138)
deleteKey(3, x_1, 128)

B-트리 삭제 알고리즘

```
deleteBT(T, m, oldKey)

    x ← T;        //  x 노드의 주소를 스택에 저장하면서 oldKey의 위치를 탐색함.
    do {
        i ← 1;
        while( i <= x.n && oldKey > x.K_i )
                i ← i+1;
        if (i <= x.n && oldKey = x.K_i)             // 삭제할 키를 발견함.
                then exit;

        // for some i where K_i-1 < oldKey < K_i, 삭제할 키를 아직 발견하지 못함.
        push x at the top of a stack;
    } while ( (x ← x.P_i-1) != null );

    if (x is null) return;                         // 삭제할 키를 최종 발견 못하면 종료
```

```
if (oldKey is not in terminal node) {          // 키가 내부 노드에서 발견됨.
        internalNode ← x;
        push x at the top of a stack;
        x ← x.Pi;
        do {
                push x at the top of a stack;
        } while ( (x ← x.P0) != null );         // 이 경우, x의 최종 값은 널이 됨.
}

if (x is null) {          // 키가 내부노드에서 발견되면, 키와 후행키를 교환함.
        x ← pop a value from the top of the stack;          // x는 후행키가 있는 단말노드
        temp ← internalNode.Ki;    // 삭제할 키와 후행키를 교환함.
        internalNode.Ki ← x.P0;
        x.P0 ← temp;                            // 이제 x.P0 가 oldKey임.
}

finished = FALSE;
delete oldKey in x node;                        // 현재 x는 단말 노드
if (stack is not emty) {
        y ← pop a value from the top of the stack;          // y is a parent of x.
}
```

```
do {
    if (root(x) or !underflow(x))
        finished = TRUE;
    else if (existAvailableSibling(x))  {      // bestSibling과 재분배
        choose bestSibling of x node;
        // tempNode : 재분배를 위해 사용되는 정상 노드 보다 50% 큰 노드
        copy bestSibling, x, and intermediate key of y into tempNode;
        copy keys and pointers of tempNode to bestSibling, x, and intermediate key
                of y so bestSibling and x are roughly equal size;
        finished = TRUE;
    } else {                                   // bestSibling과 합병
        choose bestSibling of x node;
        put, in the leftmost of x or bestSibling, the contents of both nodes and
                the intermediate key of y;
        discard one of the x or bestSibling node which is empty now;
        x ← y;
        if (stack is not empty) {
            y ← pop a value from the top of the stack;
        }
        else finished = TRUE;
    }
} while (!finish);
```

```
if (no keys in y) {     // 트리의 레벨이 하나 감소한다.
    T ←  x;
    discard y node;    // old root를 삭제
}
```

4. 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: $\lceil \log_m(N+1) \rceil \le h \le \left\lceil \log_{\lceil \frac{m}{2} \rceil}(N+1) \right\rceil$

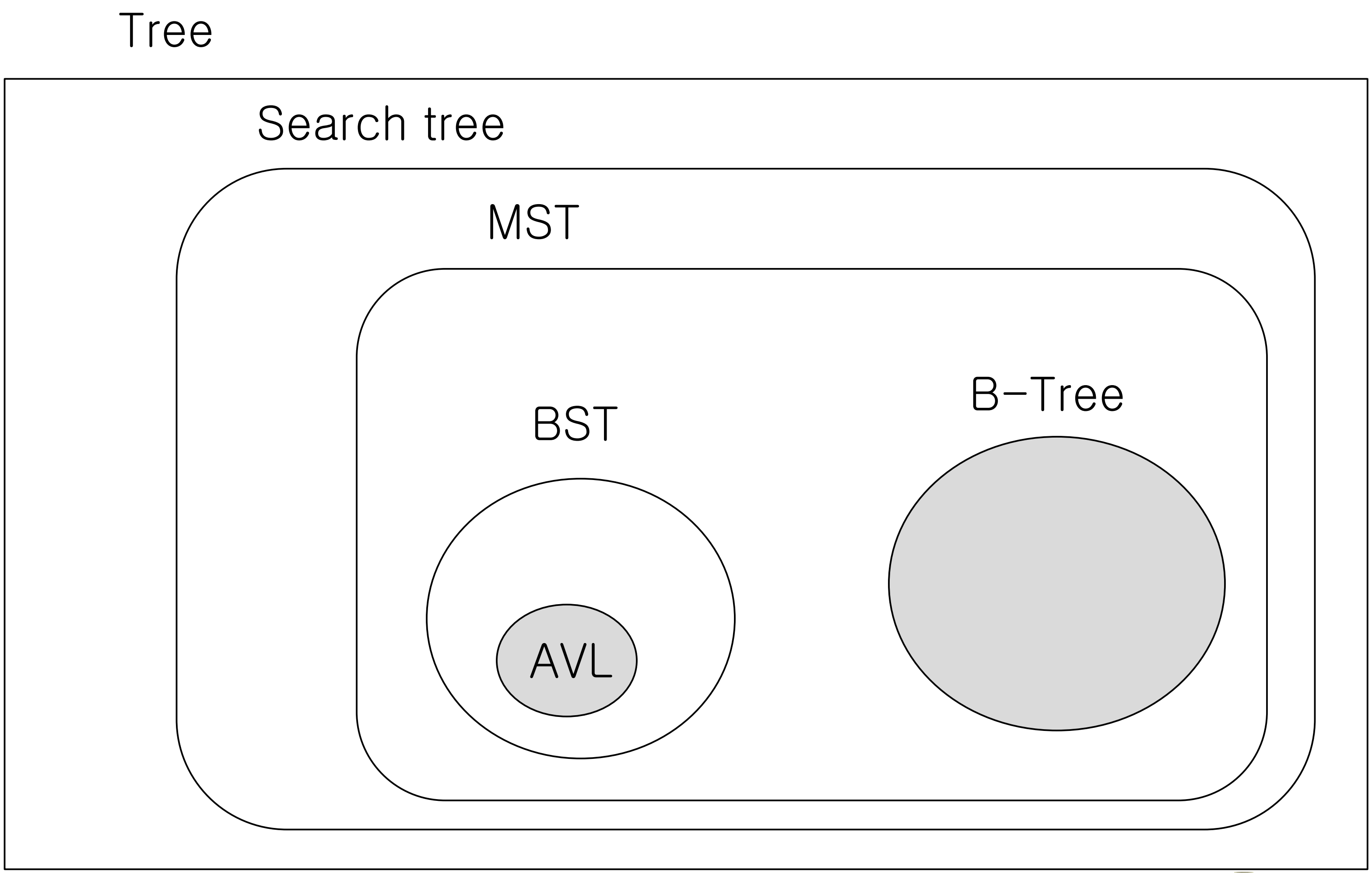
비교: N=1023 일때

- BST
  - 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 일때

- BST
  - 최소높이: $\lceil \log_2(N+1) \rceil = 20$
  - 최대높이: $n = 1{,}048{,}575$
- AVL (height-balanced BST)
  - 최소높이: $\lceil \log_2(N+1) \rceil = 20$
  - 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, $4 \le h \le 5$.

File Processing Intro

B*-트리

B*-트리

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

- B*-트리 : Knuth가 제안한 B-트리의 변형
  - 각 노드 마다 키 값이 최소 2/3 이상 찬 상태의 B-트리
  - 삽입시 노드 분할, 삭제시 노드 합병의 횟수 줄임.

B*-트리의 정의

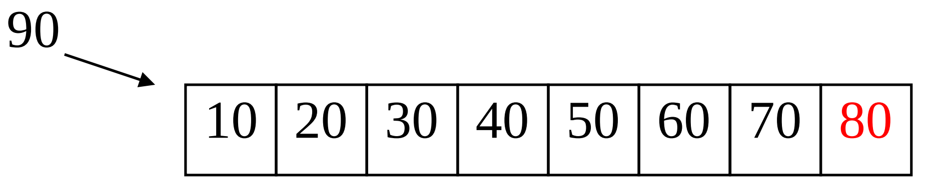
- B*-트리
  ① B*-트리 : 공백이거나 높이가 1 이상인 m-원 탐색 트리
  ② 루트는 단말 노드가 아닌 이상
  최소 2개, 최대 $\lfloor(2m-2)/3\rfloor+1$ 개의 서브트리를 갖는다.
  ③ 루트와 단말 노드를 제외한 모든 노드는 적어도 $\lfloor(2m-2)/3\rfloor+1$개의 서브트리, 즉 최소 $\lfloor(2m-2)/3\rfloor$ 개의 키 값을 갖는다.
  ④ 모든 리프는 같은 레벨에 있다.

- Note
  - B-트리보다 적은 수의 노드로 구성됨.

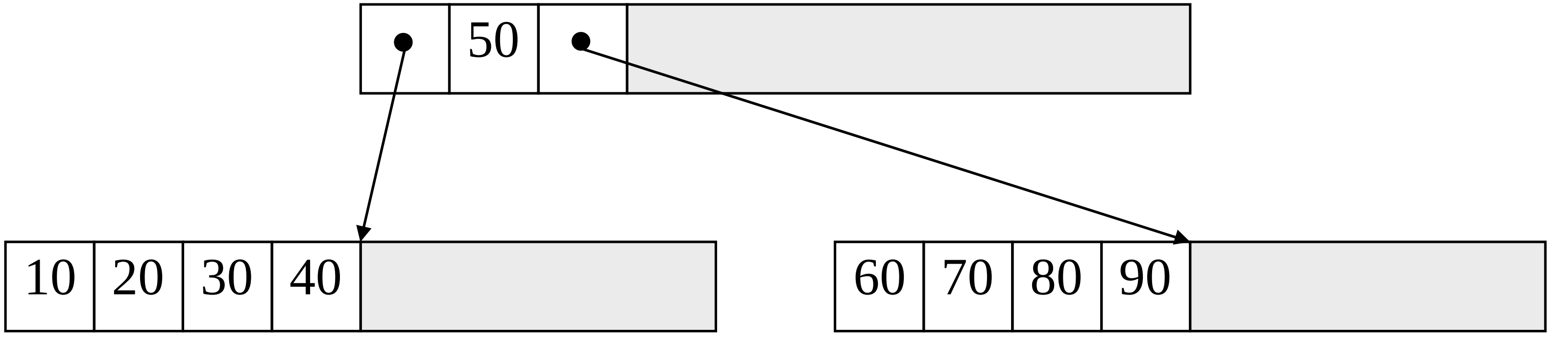
B*-트리 연산: 삽입

- 노드가 가득 찬 경우 삽입
  - B-트리: 즉시 분할
  - B*-트리: 분할 대신 인접한 형제 노드로 키를 재분배(key redistribution)시킴.
- 두 개의 이웃 노드가 모두 가득 찼을 때의 삽입
  - 두 개의 노드를 세 개로 분할시킴.
  - 분할된 세 개 노드는 각각 2/3만 참.

예: 7차 B*-트리의 생성



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



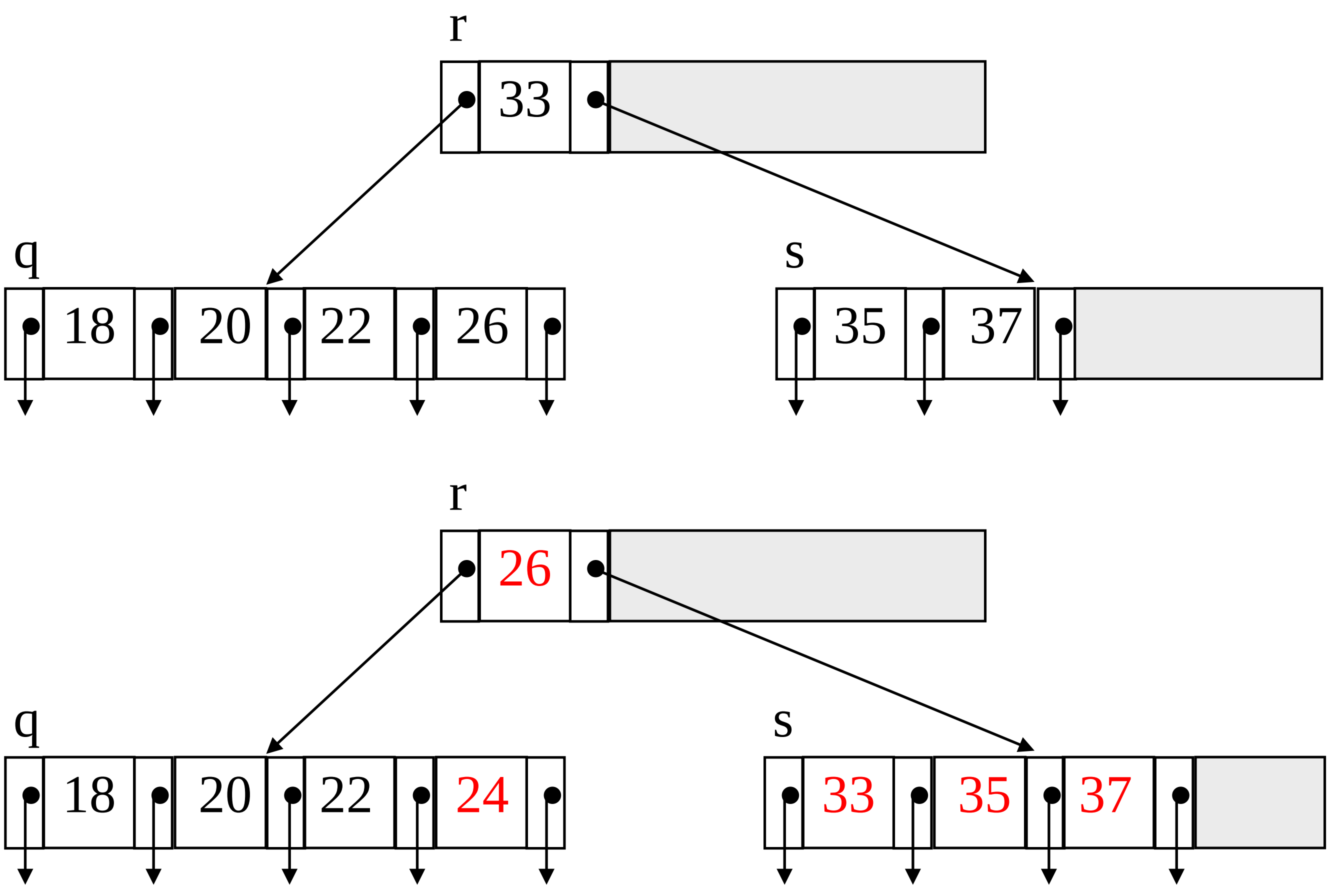
(b) 키 값 90의 삽입으로 루트 노드가 최초로 분할된 7차 B*-트리

Note: B*-트리에서 루트 노드의 분할

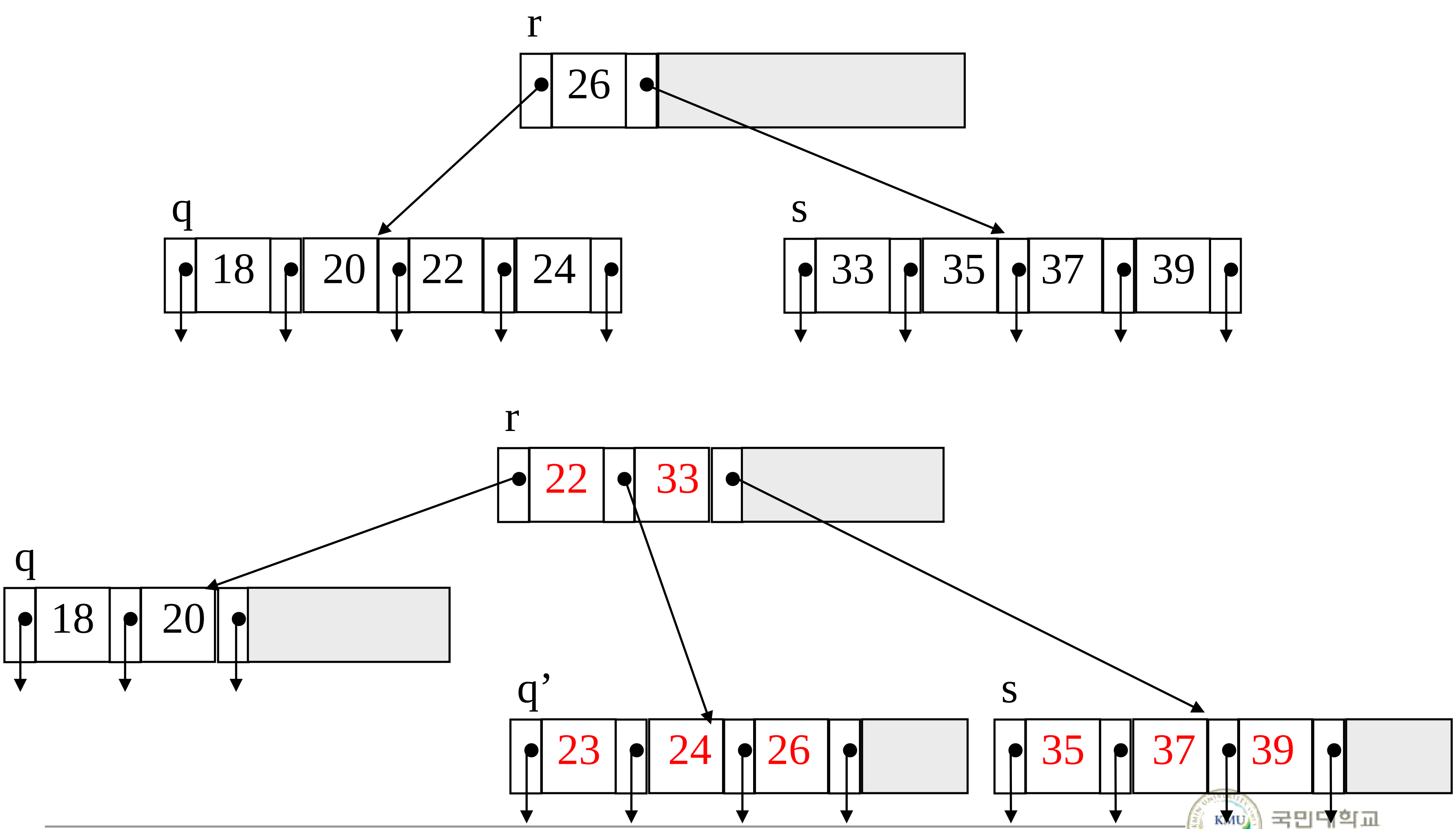
- 루트 노드의 키의 개수
  - 루트를 제외한 모든 노드는 적어도 $\lfloor(2m-2)/3\rfloor$ 개 이상의 키 값을 가져야함.
  - 따라서 처음 루트 노드가 분할될 때, 생성되는 두 개의 리프 노드에는 각각 $\lfloor(2m-2)/3\rfloor$ 개의 키가 있어야함.
  - 따라서 루트 노드가 분할되기 직전에는,
    루트에 ($\lfloor(2m-2)/3\rfloor$) x 2개의 키를 담고 있어야 함.
- 예
  - m=5, 키의 최소 개수=2, 루트 노드의 키의 개수=4
  - m=6, 키의 최소 개수=3,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

예제: 5차 B*-트리에서의 삽입

- 재분배를 이용한 키 값 24의 삽입



o 노드 분할을 이용한 키 값 23의 삽입



File Processing Intro

트라이

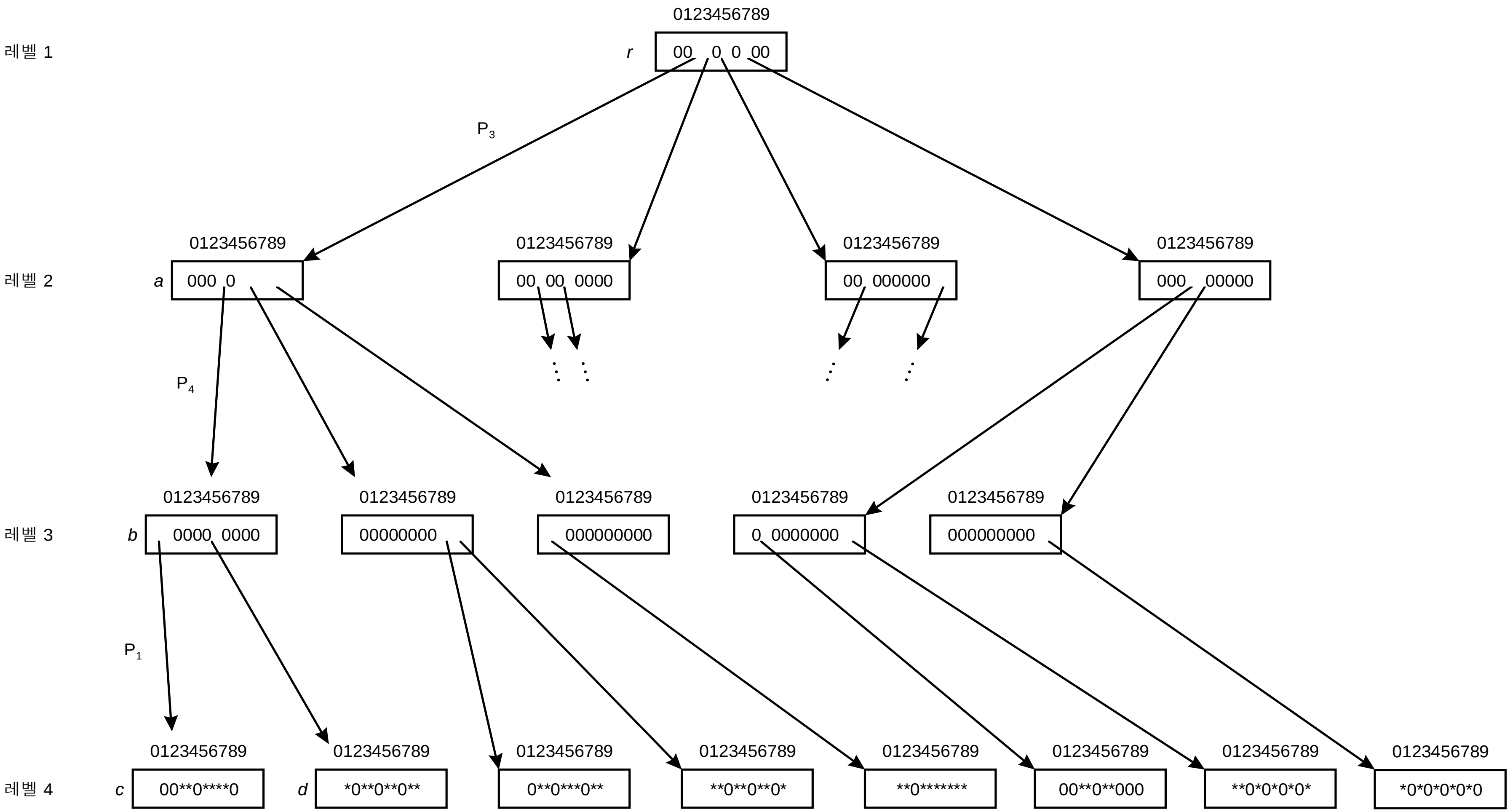
트라이 (Trie)

- reTRIEval의 약자
- 키를 구성하는 문자나 숫자의 순서로 키 값을 표현한 구조
- m진 트라이(m-ary trie)
  - 기수(radix) m : 키 값을 표현하기 위해 사용하는 문자의 수
    - ◆ 숫자 : 기수가 10이므로 m=10
    - ◆ 영문자 : m = 26
  - m진 트라이 : m개의 포인터를 표현하는 1차원 배열
    - ◆ 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}

  - 트라이의 높이 = 키 필드(스트링)의 길이

높이가 4인 10진 트라이



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

m진 트라이 연산

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

- 삽입
  - 단말 노드에 새 레코드의 주소나 마크를 삽입
  - 단말 노드 없을 때 : 새 단말 노드 생성, 중간 노드 첨가
  - 노드의 삽입이나 삭제는 있으나, 분할이나 병합은 없음.

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