6장 인덱스 구조

❖ 인덱스

- 키값과 해당 레코드 주소쌍의 체계적 모음
- 목적 : 레코드 접근 용이
- 밀집(dense) 인덱스 : 모든 레코드에 대한 키값-주소 쌍
- 희소(sparse) 인덱스

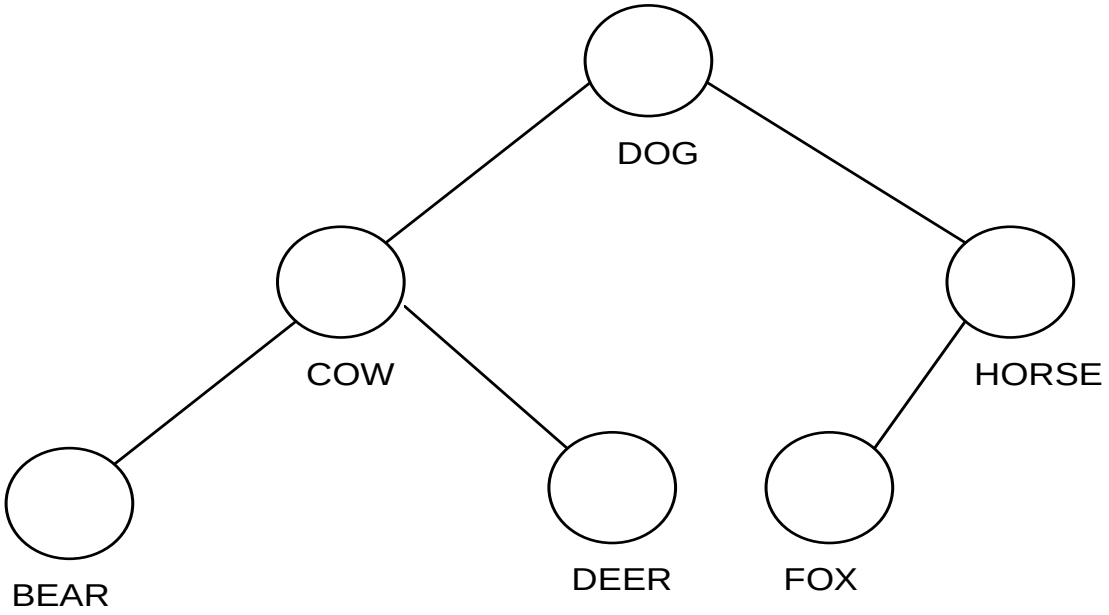
❖ 이원 탐색 트리 (binary search tree)

◆ 노드 Ni = (키 K, 주소)

① Nj ∈ RT(Ni) → Ki < Kj

② Nj ∈ LT(Ni) → Kj < Ki

이원 탐색 트리



▶ 탐색과 삽입

◆ 탐색

– 루트 Ni, 탐색키 K

① 공백 트리 : 탐색 실패, 종료

② K = Ki : Ni가 목표 노드

③ K < Ki : Ni ← ROOT(LT(Ni))

④ K > Ki : Ni ← ROOT(RT(Ni))

◆ 삽입

– 리프 노드의 위치에 삽입

– 루트 Ni, 삽입 레코드의 키 K

① 공백 트리 : K의 노드가 루트

② K = Ki : 삽입 실패

③ K < Ki : 왼쪽 서브트리 탐색

④ K > Ki : 오른쪽 서브트리 탐색

▶ 삭제와 성능

- **삭제**
  - 리프의 삭제 : 노드 제거
  - 리프가 아닌 경우 : 서브트리 유지
  - 삭제 표시

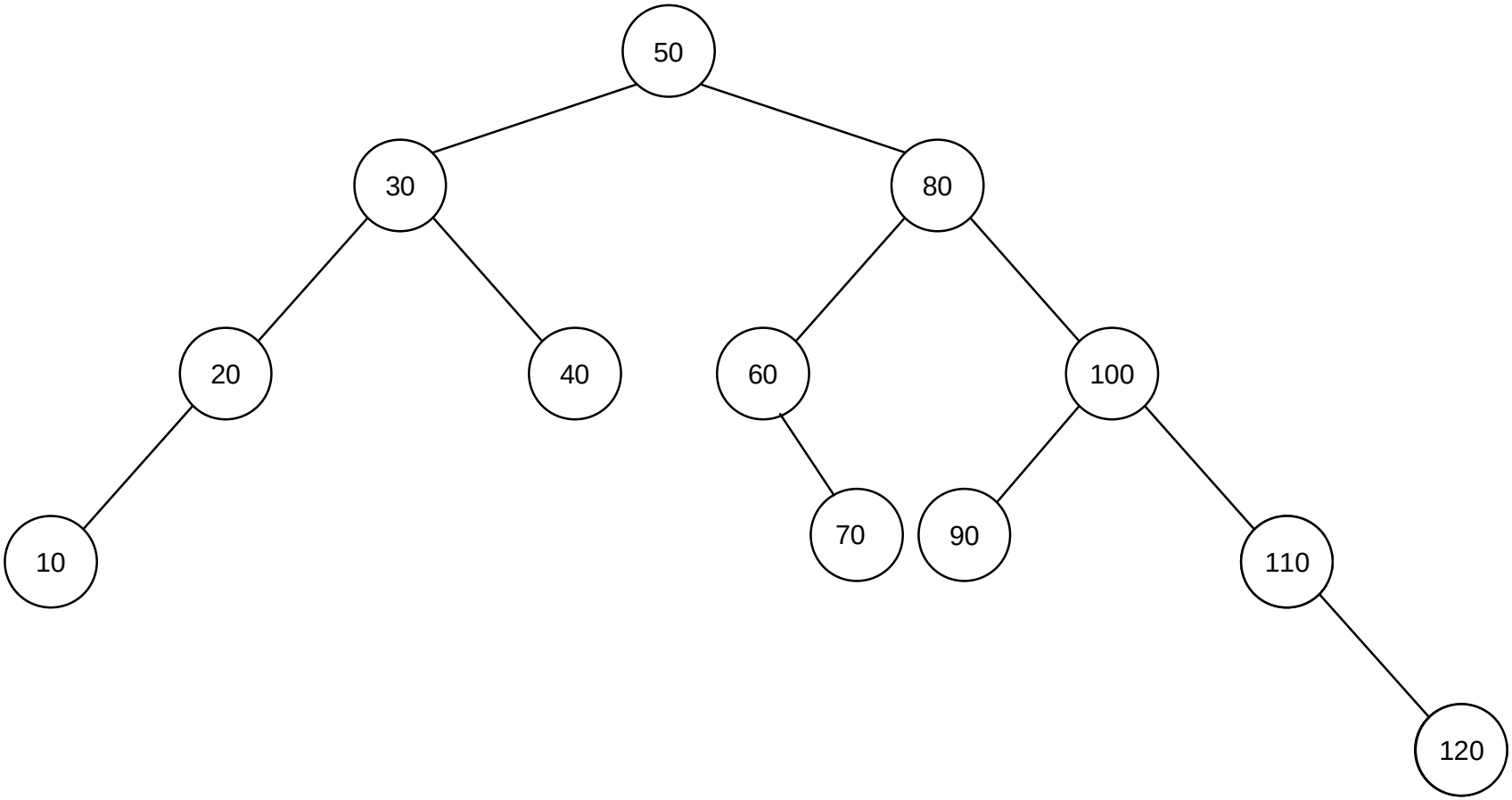
- **성능**
  - 자주 접근되는 노드 - 루트에 근접
  - 균형 트리(balanced tree)
    - 트리의 최대 경로 길이를 최소화
    - N 노드 : ⌊log2N⌋ + 1

❖ AVL 트리

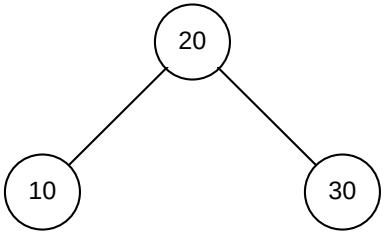
- 높이균형 이진트리(**height-balanced binary tree**)
- **Adelson-Velskii**와 **Landis**가 소개

- 정의
  - T = AVL 트리
  - 공백이 아닌 이진 트리
  - $h(LT(Ni)) - h(RT(Ni)) \leq 1 \quad Ni \in T$
  - N노드 AVL트리에서 임의 접근 : O(log N)

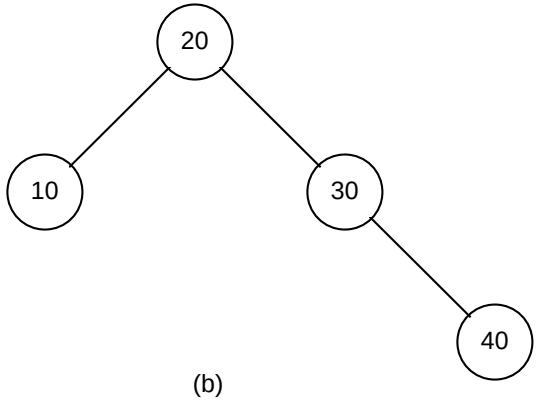
▶ AVL 트리 예



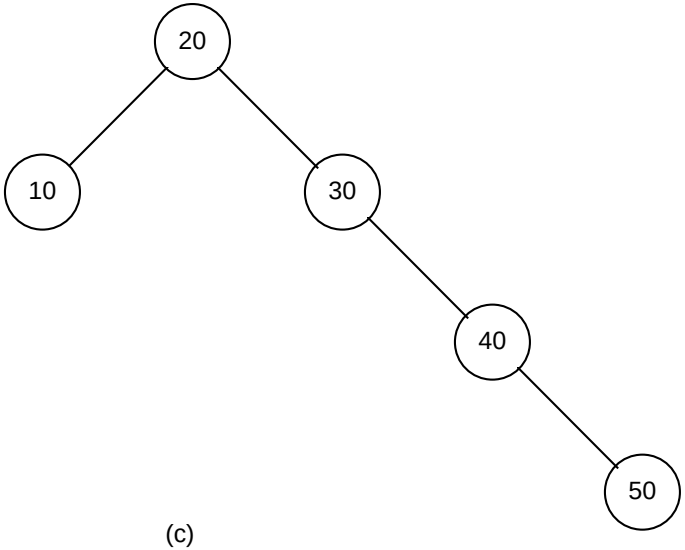
▶ AVL 트리의 재균형 (삽입시)



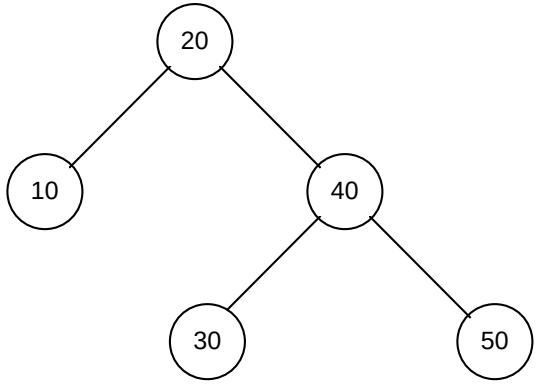
(a)



(b)



(c)



(d)

▶ AVL 트리의 높이

- **log2(N+1) ≤ h ≤ 1.4404log2(N+2)-0.328**
- **O(1.4 log N) : O(log N)**
- 부분 재균형 : 전체 트리의 재균형

❖ m-원 탐색 트리

- **(m-1) 키, m 서브트리**
  - 높이 감소 : 탐색 시간 감소
  - 삽입, 삭제 : 균형 유지 어려움

- **특성**

T : m-원 탐색 트리

① 노드 구조

n	P_1	K_1	P_2	K_2	P_3	…	P_{n-1}	K_{n-1}	P_n

(Ki : 키 , Pi : 서브트리에 대한 포인터 , $1 \le n \le m$)

② 키의 오름차순 : Ki < Ki+1 i=1,2,..,n-2

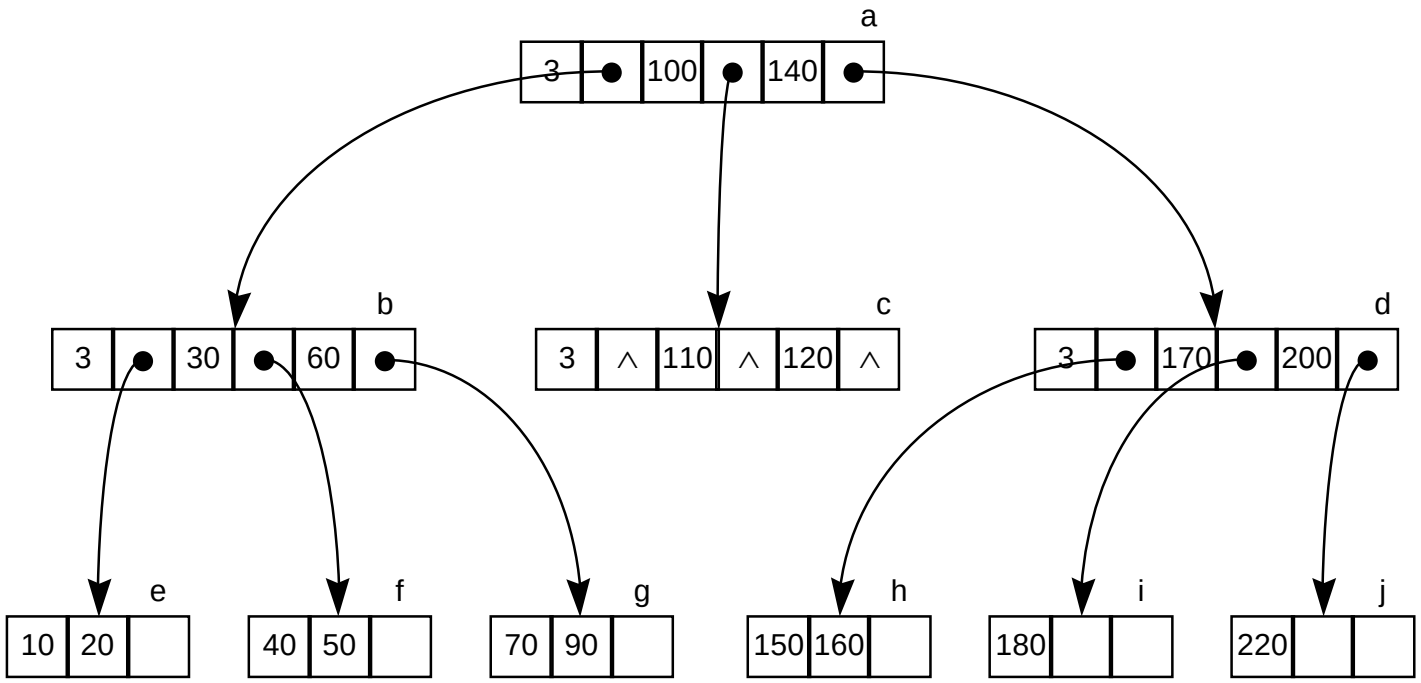
③ Pi가 지시하는 서브트리 내의 키값 < Ki

④ Pn이 지시하는 서브트리 내의 키값 > Kn-1

⑤ Pi가 지시하는 서브트리 : m-원 서브트리

-

▶ 3원 탐색 트리



★ Ki → (Ki, Ai) Ai : 데이타 레코드의 주소

▶ 성능

- **m-원 탐색트리 : m-1 키**
- **높이 h → (m^h-1)개의 키값**
  3 → $(3^3-1) = 26$
- **n 키 → 높이 : $\lceil \log_m(n+1) \rceil$**
  최대 탐색시간 : $O(\log_m(n+1))$
- **삽입, 삭제시 트리의 균형 유지**

❖ B-트리

- **Bayer & McCreight**
- 균형 **m**-원 탐색 트리

- 차수 **m**인 **B**-트리의 특성

① 루트와 리프를 제외한 노드의 서브트리 수
⌈ **m/2** ⌉≤ 개수 ≤ **m**

② 루트(리프가 아닌)의 서브트리의 수 **2**

③ 모든 리프는 같은 레벨

④ 키값의 수
리프 : ⌈ **m/2** ⌉**- 1 ~ (m-1)**
리프가 아닌 노드 : 서브트리수 **- 1**

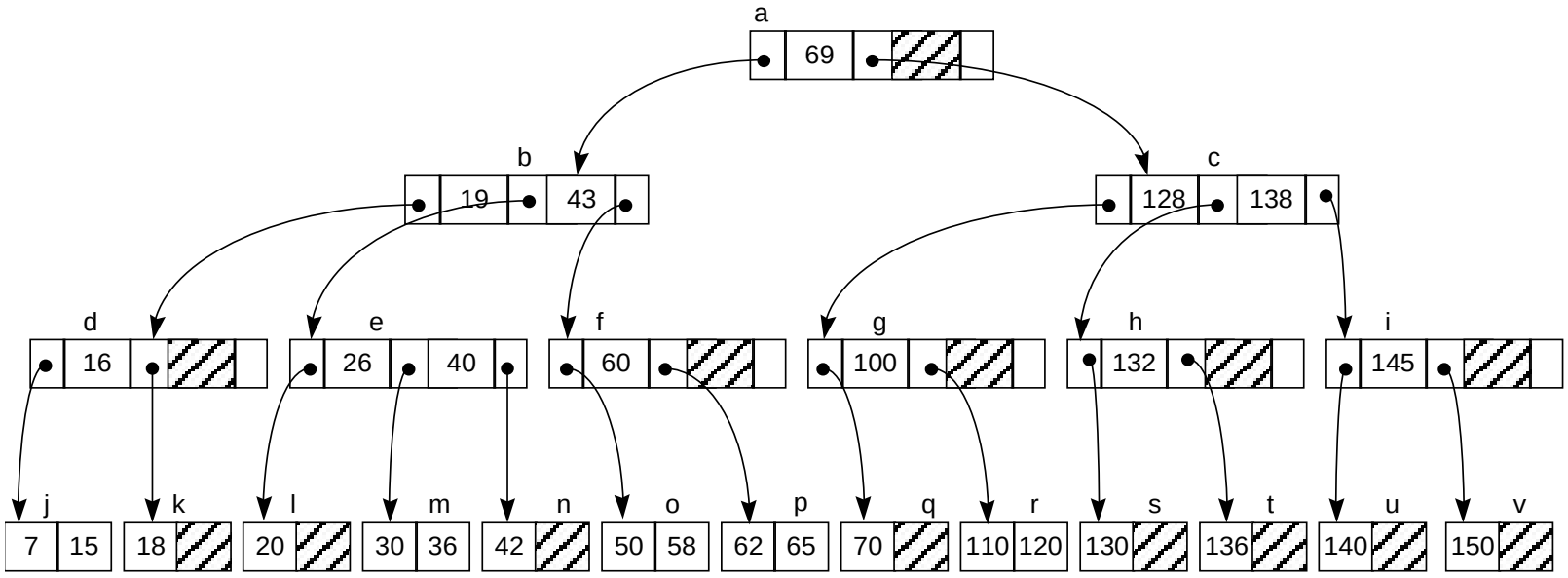
⑤ 한 노드 내의 키값 : 오름차순

▶ B-트리 구조

- 노드 구조
  - Ki → (Ki, Ai)

n	P_1	K_1	P_2	K_2	P_3	…	P_{n-1}	K_{n-1}	P_n

차수 3인 B-트리 구조

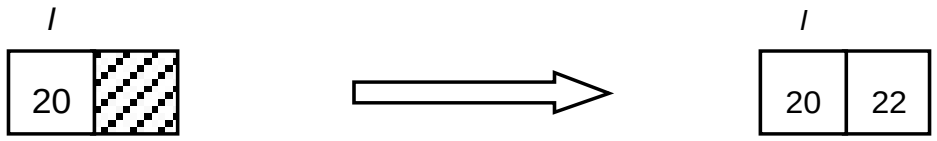


▶ B-트리 연산

- 연산
  - 직접 탐색 - 키 값에 의존한 분기
  - 순차 탐색 - 중위 순회
  - 삽입, 삭제 - 트리의 균형 유지
    - 분할 → 높이 증가
    - 합병 → 높이 감소

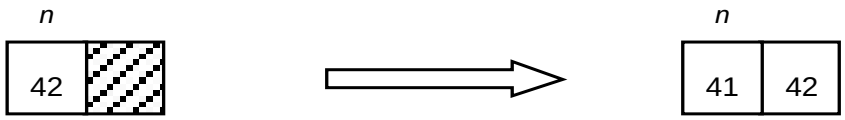
- 삽입

(a) 노드 I에 22 삽입

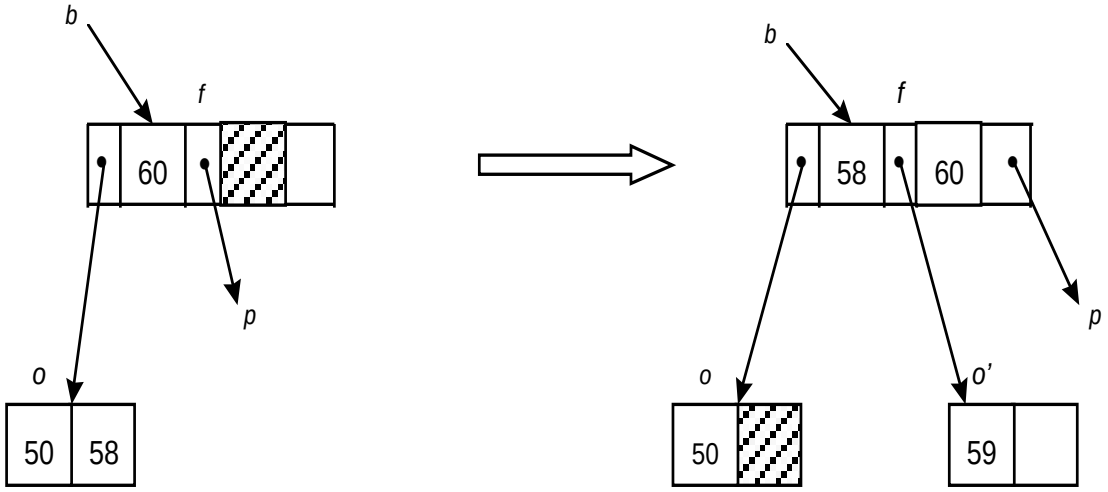


▶ B-트리 삽입

(b) 노드 n에 41 삽입

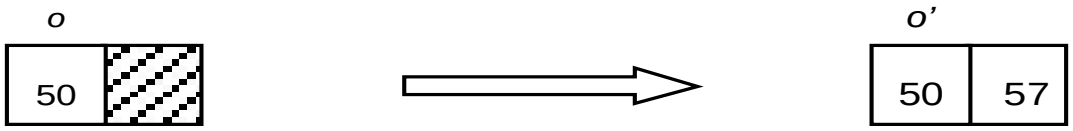


(c) 노드 o에 59 삽입

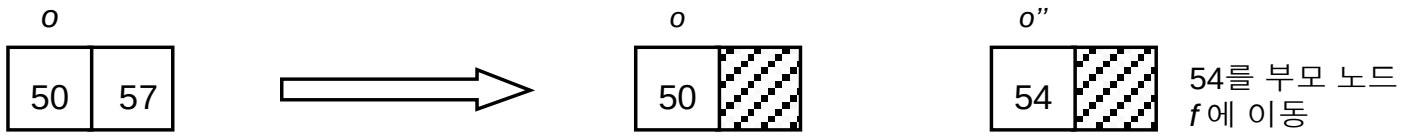


▶ B-트리 삽입

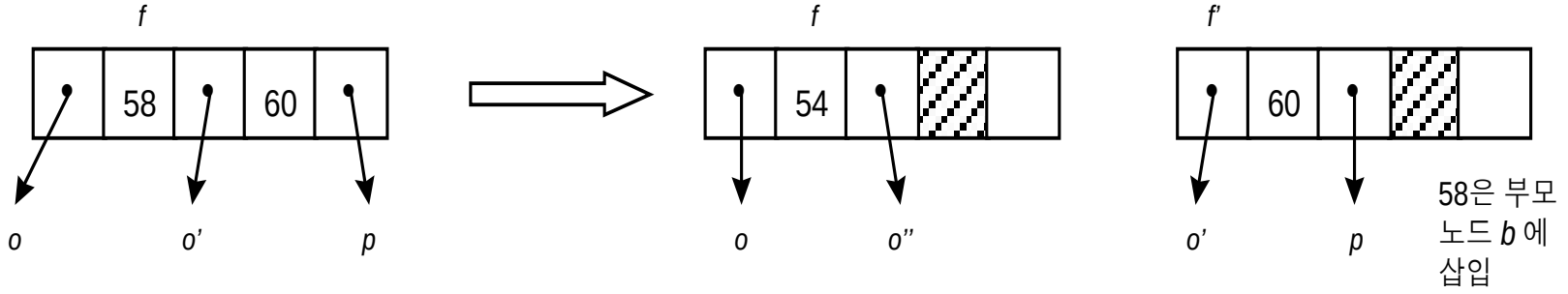
(d) 노드 o에 57 삽입



(e) 54의 삽입으로 노드 o의 분열

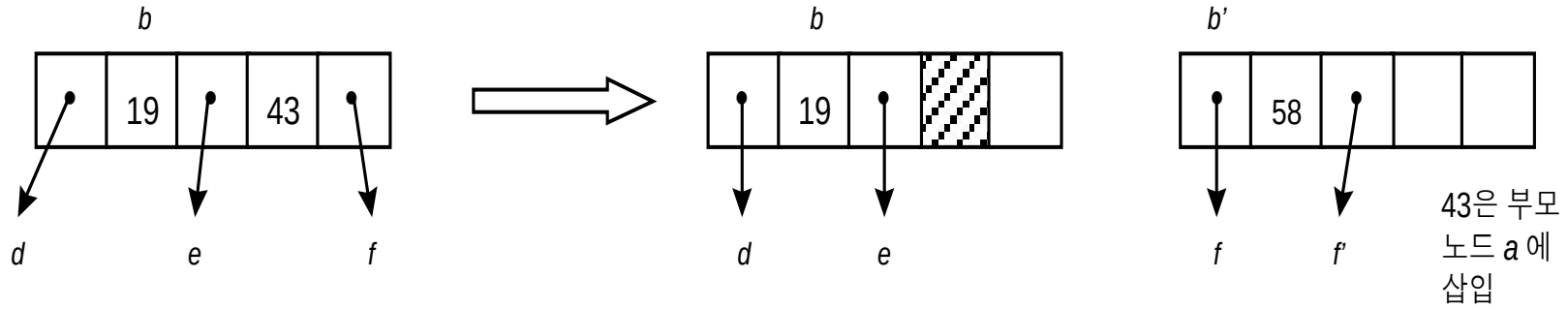


(f) 노드 f에 54 삽입

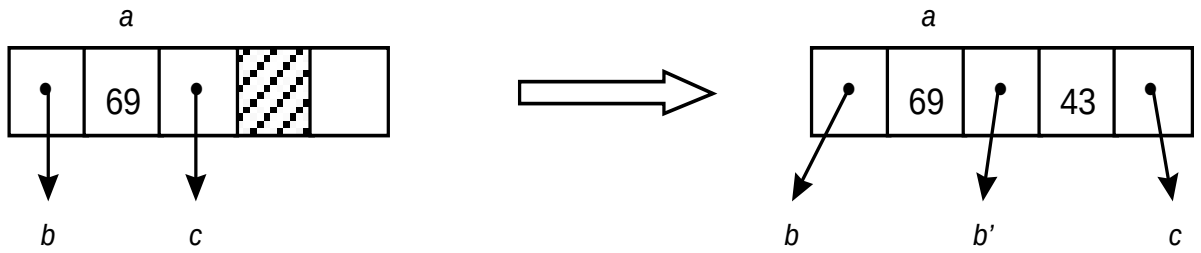


▶ B-트리 삽입

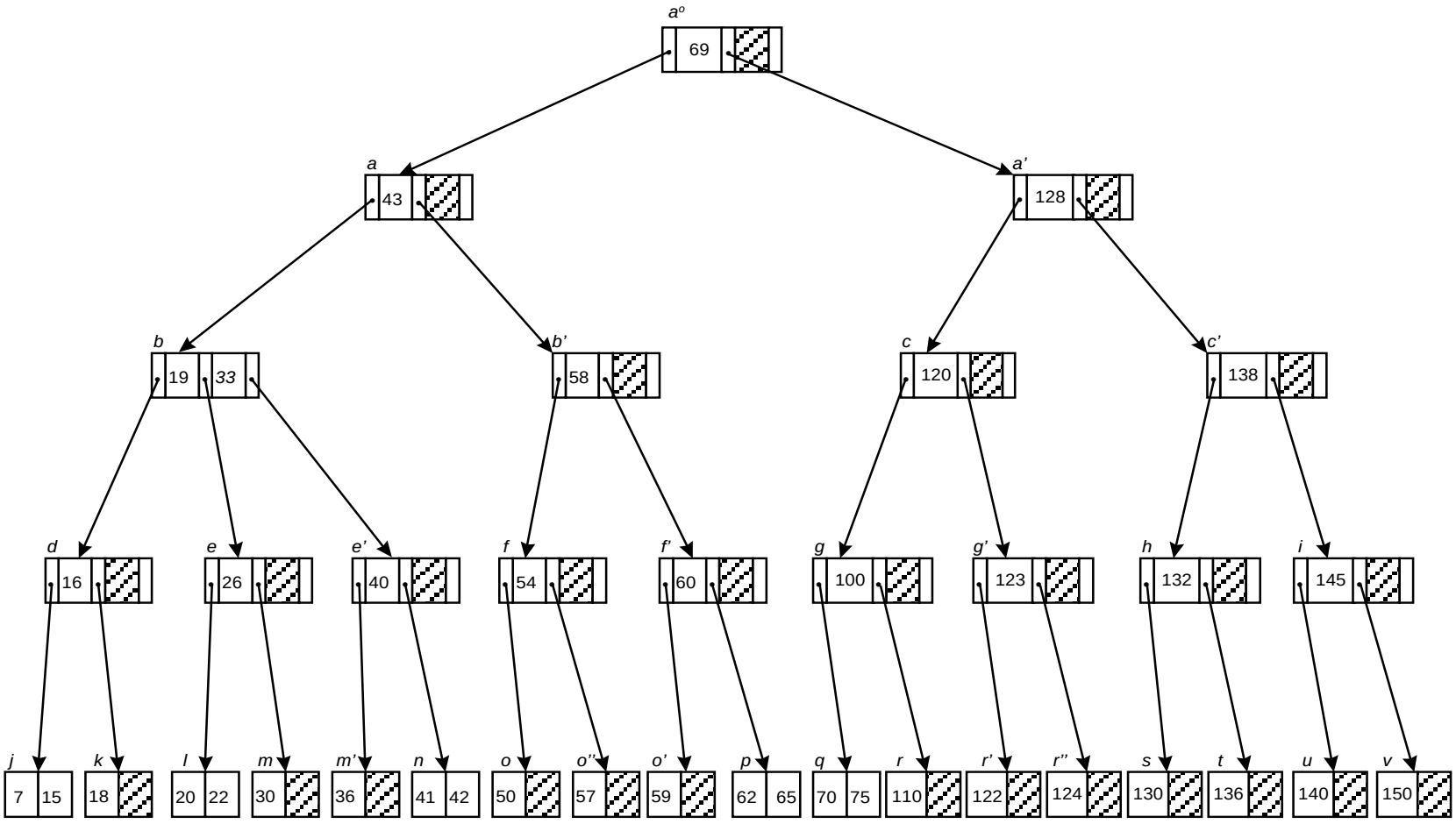
(g) 노드 b에 58 삽입



(h) 노드 a에 43 삽입



한 레벨 증가된 B-트리


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
m'
36
n
41
42
o
50
o''
57
o'
59
p
62
65
q
70
75
r
110
r'
122
r''
124
s
130
t
136
u
140
v
150

▶ 삽입 알고리즘

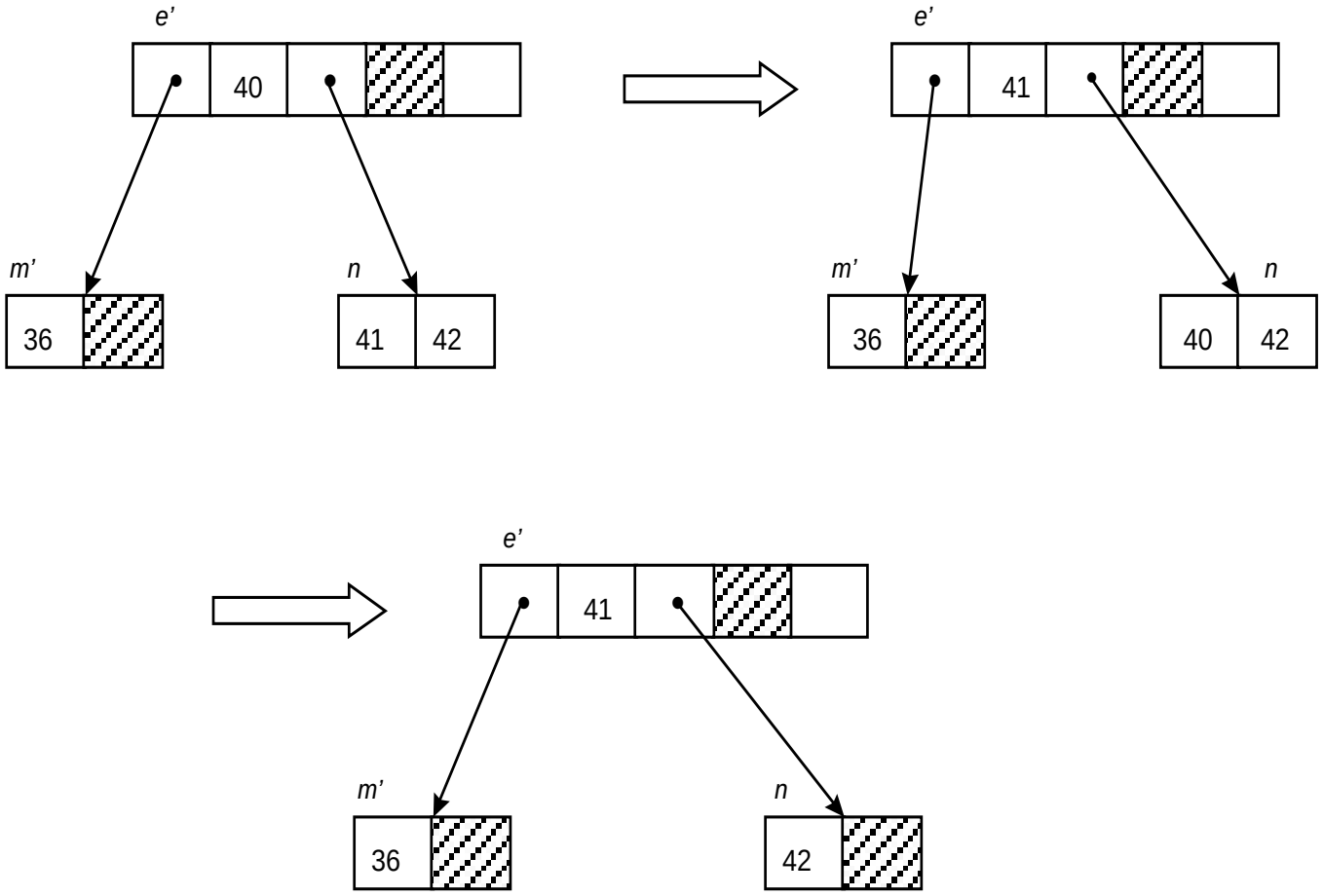
```
/*  알고리즘에서 사용되는 변수는 다음과 같다.
In-key     :  B-트리에 삽입될 키
Finished  :  삽입이 완료되었음을 나타내는 플래그
Found     :  B-트리에서 레코드가 발견되었음을 나타내는 플래그
P          :  노드에 대한 포인터
TOOBIG              :  오버플로 노드를 위한 변수
N          :  키 카운터
*/

/*  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);
```

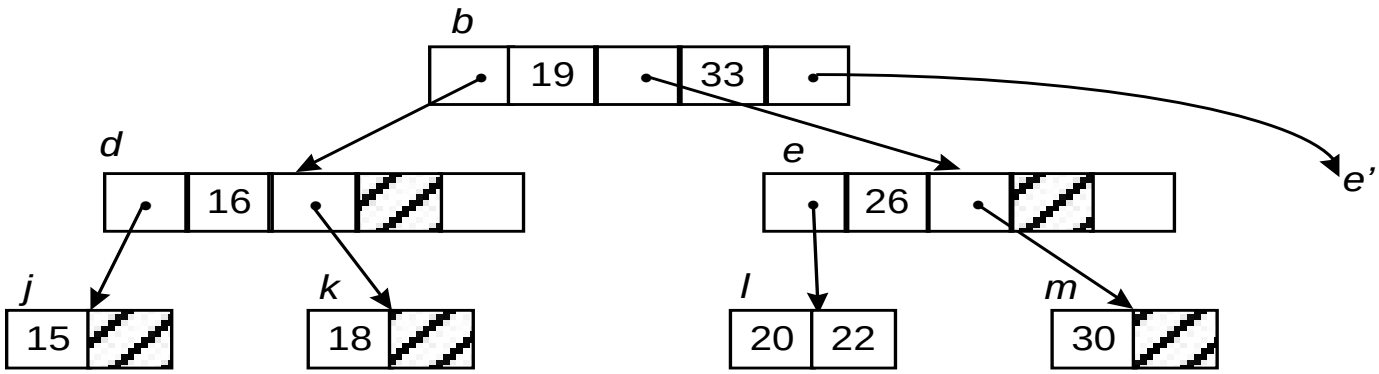
```
  if (Found) report In-key already in tree;
  else  { /* In-key를 B-트리에 삽입한다 */
        P = nil;
        Finished = false;
        do  {
              if (current node is not full) {
                  put In-key in current node;
                  /* 노드 안에서 키 순서를 유지하도록 키를 정렬한다 */
                  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-트리 삭제

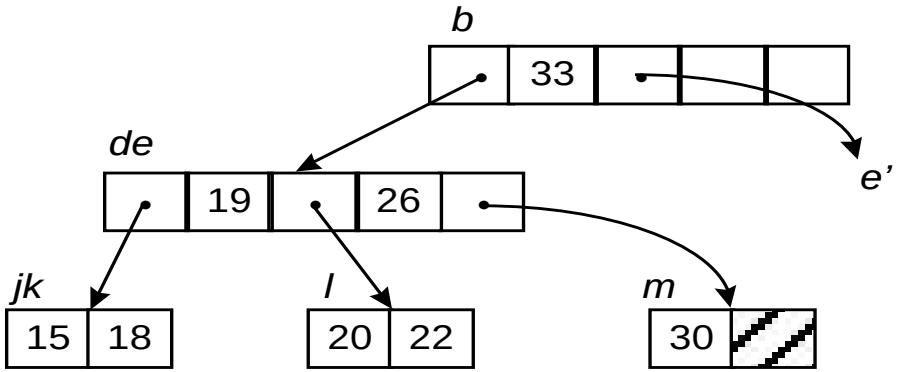
- 삭제
  - 노드 e'에서 키 값 40의 삭제



▶ 노드 d에서 키 값 16의 삭제



(a)



(b)

▶ 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 8.9의 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;
```

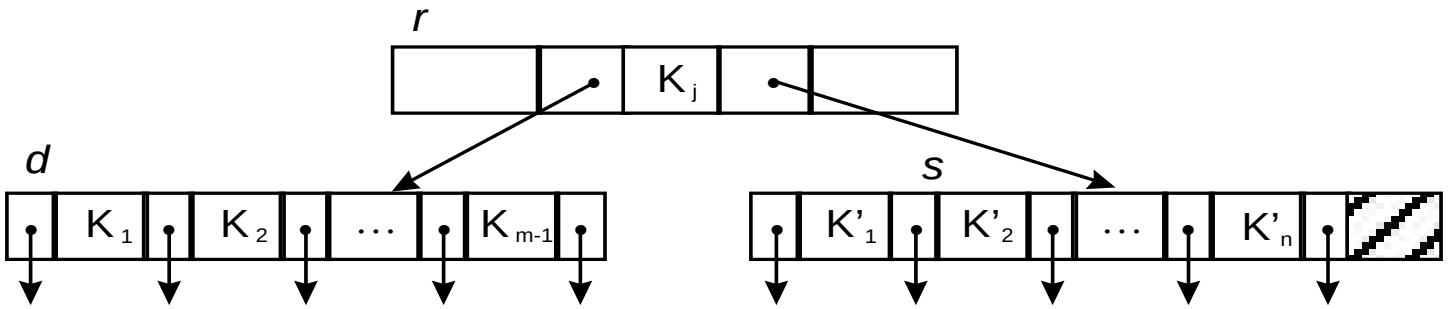
```
do {
      remove Out-key
      if (current node is root or is not too small)
         Finished = true;
      else if (redistribution possible)  {
      /* A-sibling > minimum이 성립할 때 재분배 가능 */
         /* 재분배 실행 */
         copy "best" A-sibling, intermediate parent key, and
         current (too-small) 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;
}
```

❖ B*-트리

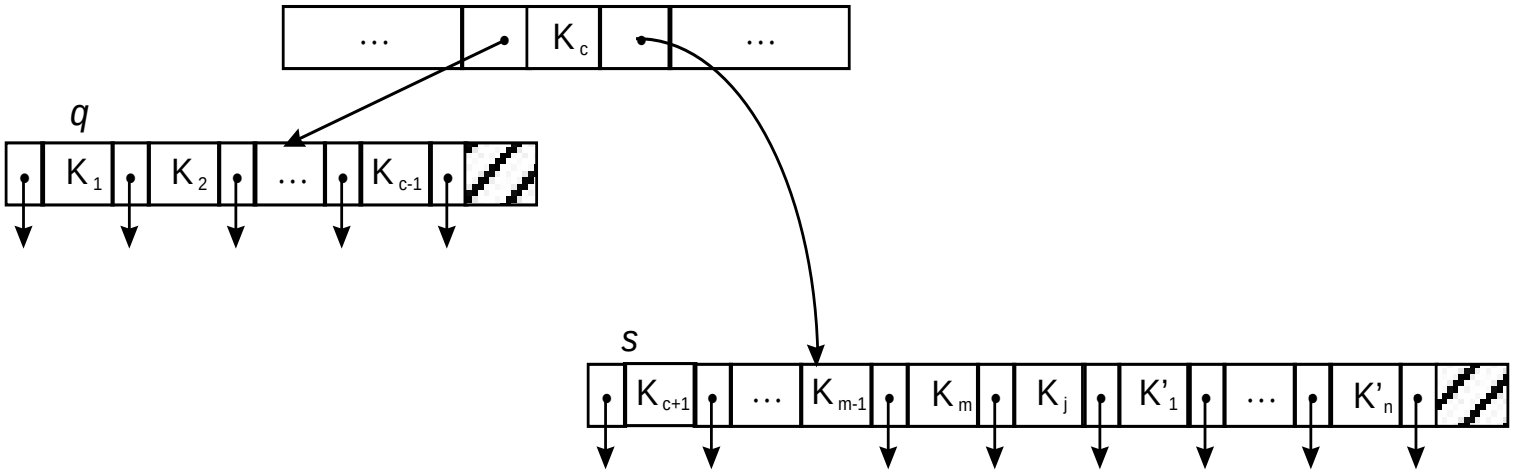
- 2/3 정도 찬 노드들을 가지는 B-트리
- 노드 분할의 횟수 줄임
- 노드가 가득참 → 형제노드로 분산 (이동)
- 두 개의 이웃노드가 모두 가득 찼을 때의 삽입
  - 두 개의 노드를 세 개로 분할시킴
  - 2/3 가득 참

▶ B*-트리 삽입

- 차수가 m인 B*-트리

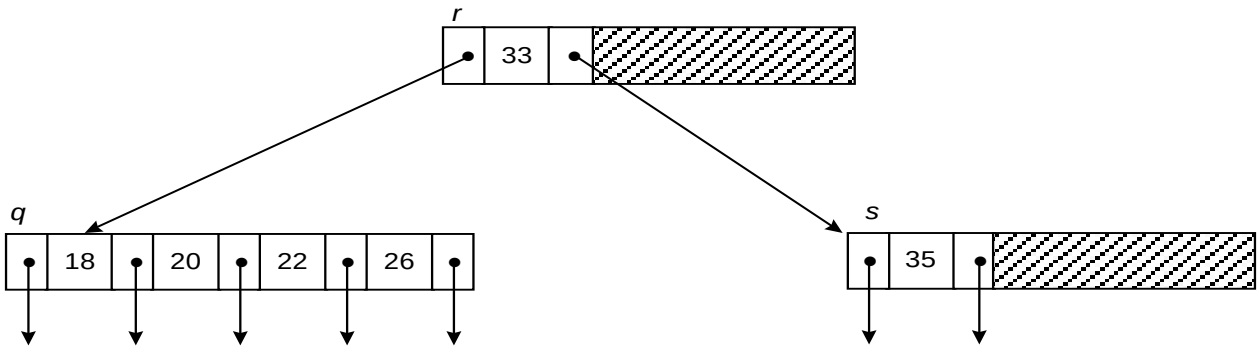


- 차수가 m인 B*-트리에서 재분배를 이용한 키 값 Km의 삽입

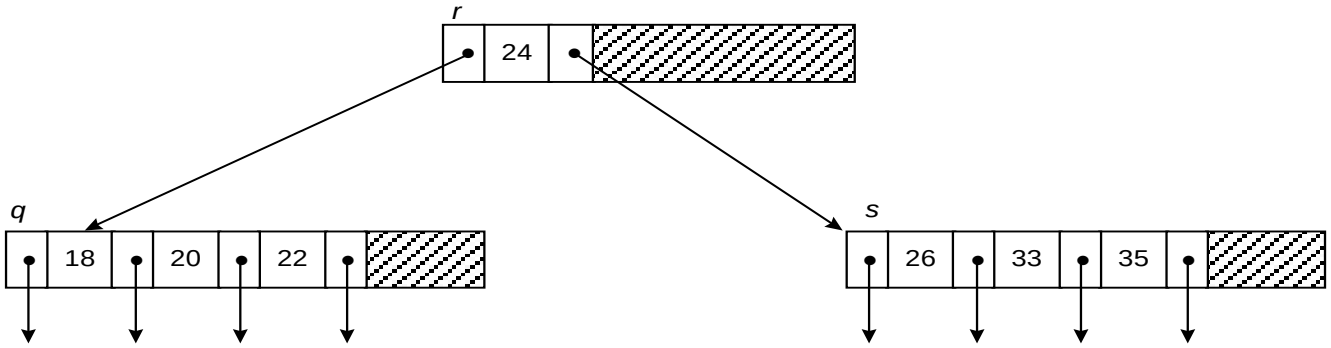


▶ 재분배를 이용한 삽입 예

- 차수가 5인 B*-트리에서 재분배를 이용한 키 값 24의 삽입

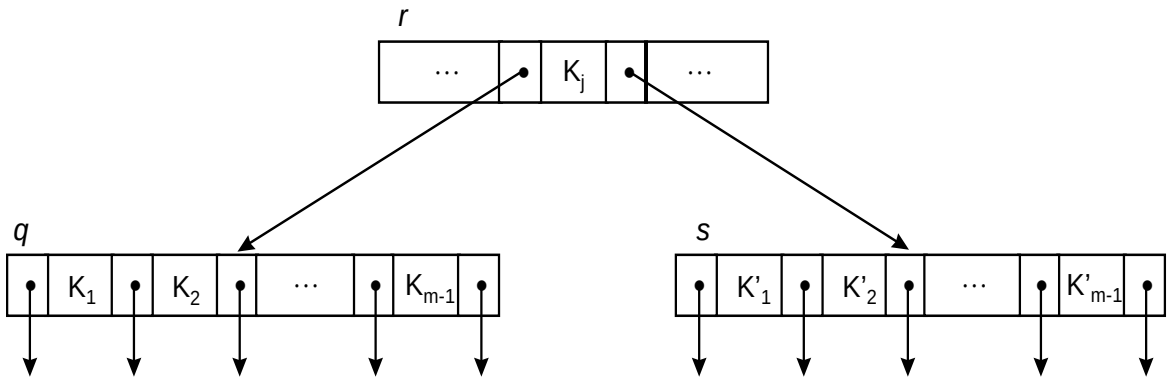


(a)

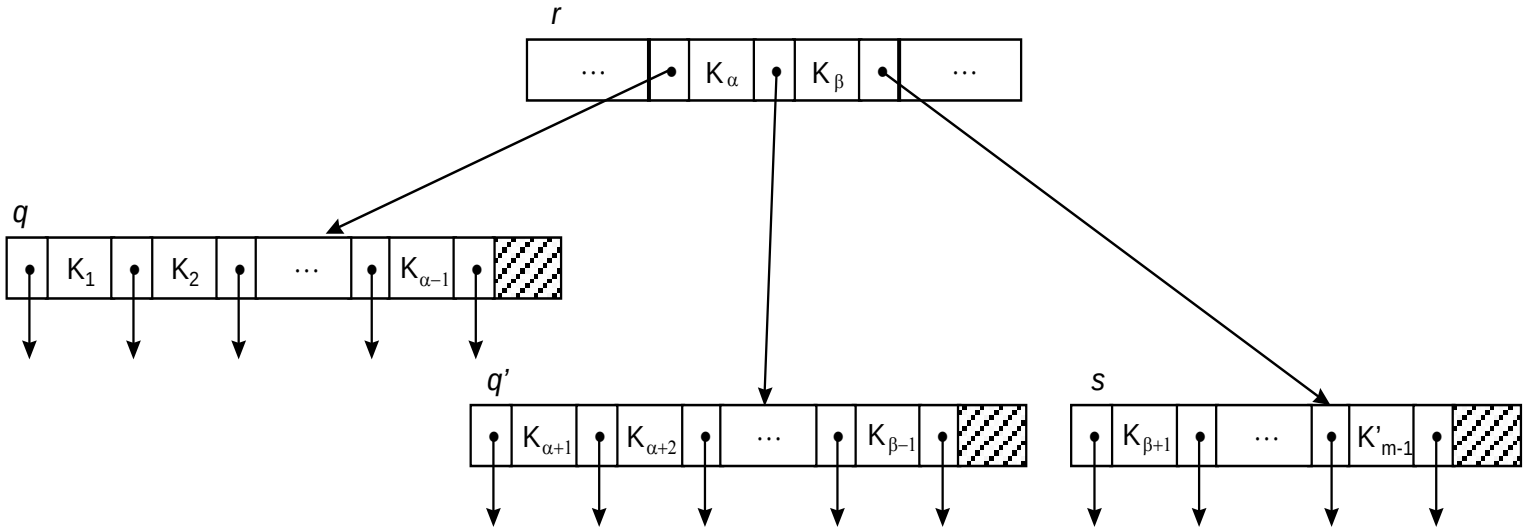


(b)

▶ 차수가 m인 B*-트리에서의 노드 분열



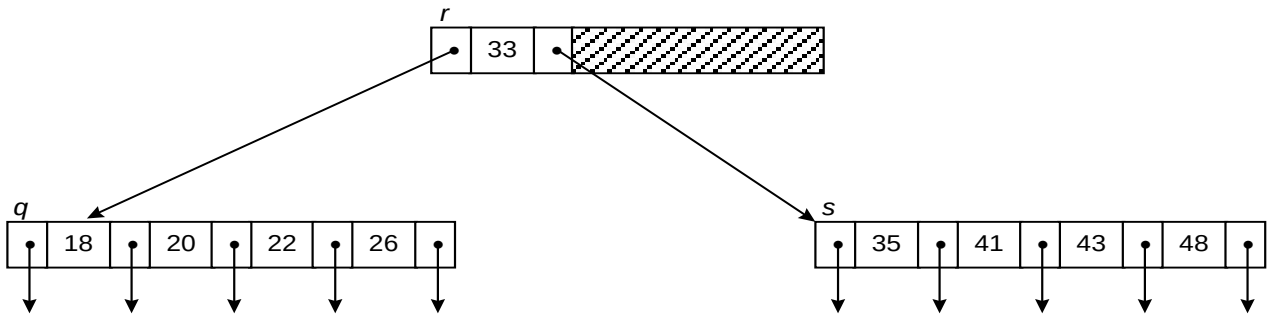
(a)



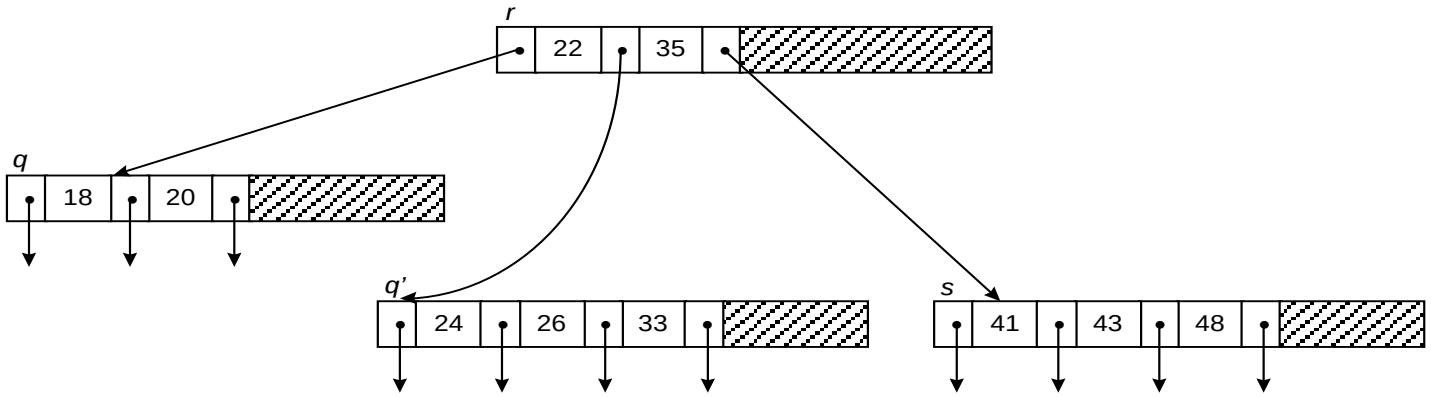
(b)

노드분열을 이용한 삽입

- 차수가 5인 B*-트리에서 노드 분열을 이용한 키 값 24의 삽입



(a)



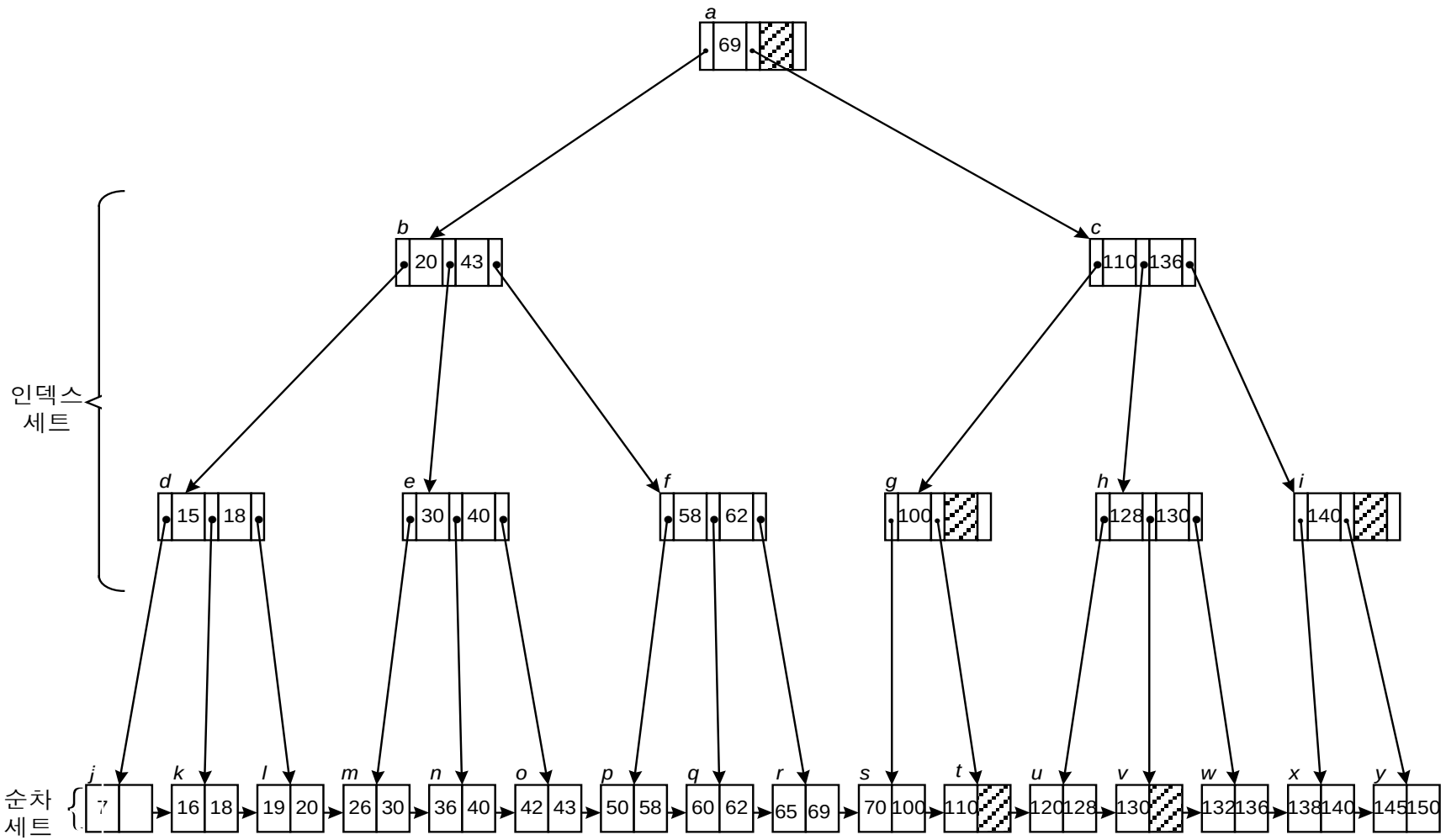
(b)

❖ B+-트리

- **인덱스 셑 (index set)**
  - 내부 노드
  - 리프에 있는 키들에 대한 경로정보 제공

- **순차 셑 (sequence set)**
  - 리프 노드
  - 모든 키 값들을 포함
  - 순차셑은 순차적으로 연결
    - 직접 또는 순차 접근
  - 내부 노드와 다른 구조

▶ 차수가 3인 B+-트리


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

▶ B+-트리 특성

① 루트의 서브트리 : 0, 2,⌈ m/2 ⌉ ~ m

② 노드의 서브트리 (루트,리프제외) : 2 ~ m

③ 모든 리프는 동일 레벨

④ 리프가 아닌 노드의 키값 수 : 서브트리수-1

⑤ 리프노드 : 데이타 화일의 순차셑 (리스트로 연결)

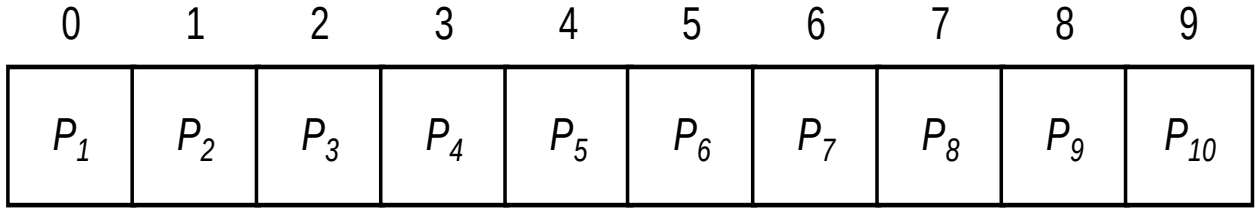
▶ B+-트리 연산

- 연산
  - 탐색
    - B+-트리의 인덱스 셑 = m-원 탐색 트리
    - 리프에서 검색
  - 삽입
    - B-트리와 유사
    - 오버플로우(분열) → 부모노드, 분열노드 모두에 키값 존재
  - 삭제
    - 리프에서만 삭제 (재분배, 합병 없는 경우)
    - 재분배시 : 인덱스의 키값도 삭제

❖ 트라이 (Trie)

- 글자나 숫자의 위치에 의해 키값을 나타내는 자료구조

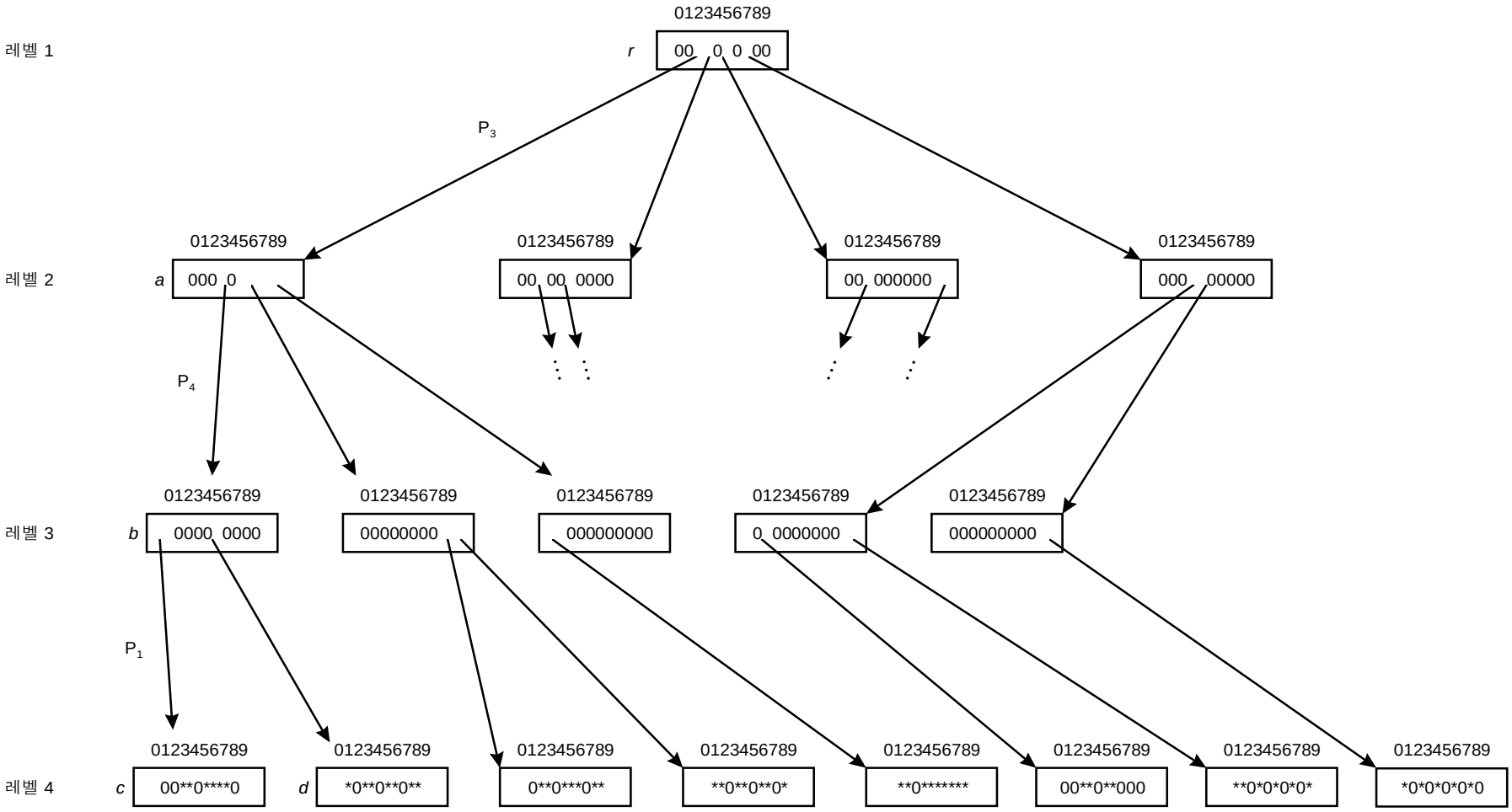
- **10원 트라이의 노드 구조**



▶ m-원 트라이

- **m진 트리**
  - m = 10 : 숫자
  - m = 26 : 글자
- **레벨 j, Pj : j 번째 값이 Pi인 모든 키값을 나타내는 서브트리를 가리킴**
- **높이 = 키 필드의 길이**
- **최대 탐색 비용 ≤ 키 값의 길이**
- **균일한 탐색시간(키내의 숫자나 글자수)**

▶ 높이가 4인 10원 트라이



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