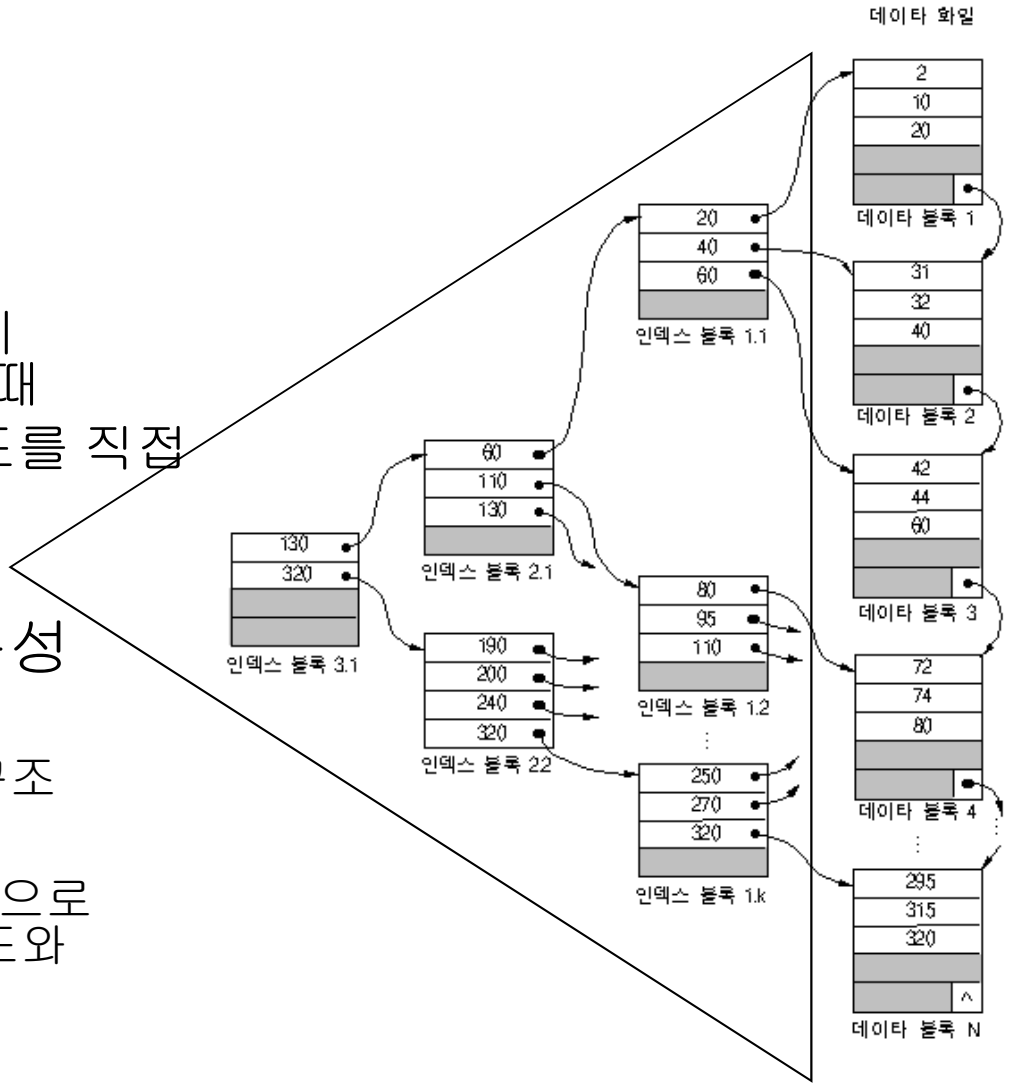
7장 인덱스된 순차 화일

- 순차 화일
  - 순차 검색에 특화
- 인덱스 파일 (AVL tree, B-tree)
  - 임의 검색에 특화
  - 순차 검색은 in-order로 가능은 하나, 키의 개수가 많아지면 stack overflow 발생. (실질적으로 순차 검색이 불가능)
- 인덱스된 순차 파일
  - 순차 검색, 임의 검색 모두 가능

❖ 인덱스된 순차 화일의 구조

- 구조
  - 순차 데이타 화일 (순차적으로 정렬)과 인덱스(화일에 대한 포인터)로 구성
    - 순차 데이타 화일 : 전체 레코드를 순차 접근할 때
    - 인덱스 : 한 특정 레코드를 직접 접근할 때
  - 각 화일은 블록으로 구성
    - 인덱스 화일
      - 인덱스 블록 : 트리 구조
    - 순차 데이타 화일
      - 데이타 블록 : 순차적으로 저장된 데이터 레코드와 자유공간

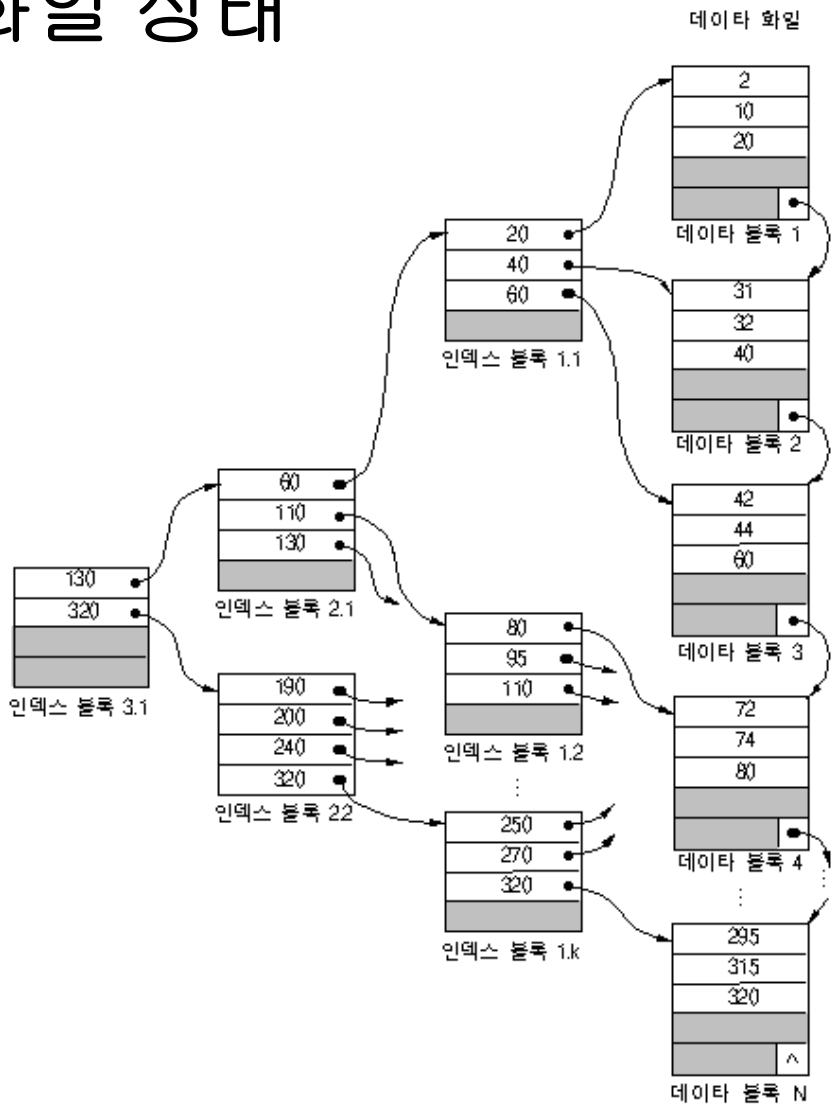


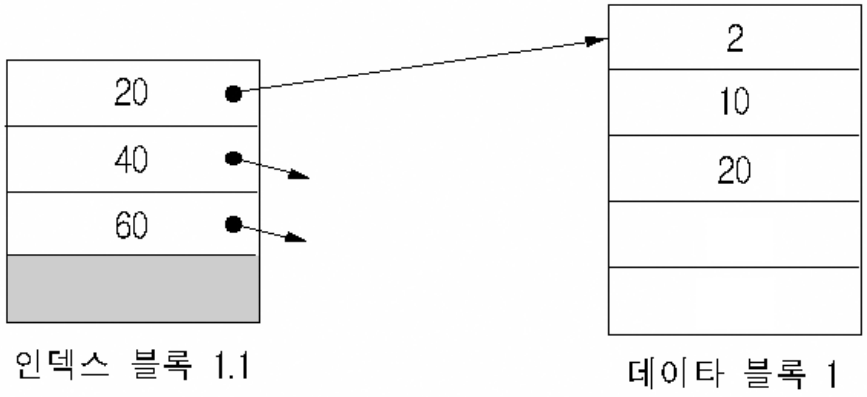
❖ 인덱스된 순차 화일의 구조

- ◆ 마스터 인덱스(**master index**)
  - 최상위 레벨의 인덱스 블록
- ◆ 인덱스 엔트리의 구성
  - <최대 키 값, 포인터>
- ◆ 자유 공간
  - 레코드의 추가를 위해 남겨둔 여분의 공간
  - 화일 생성시 각 블록에 만듦

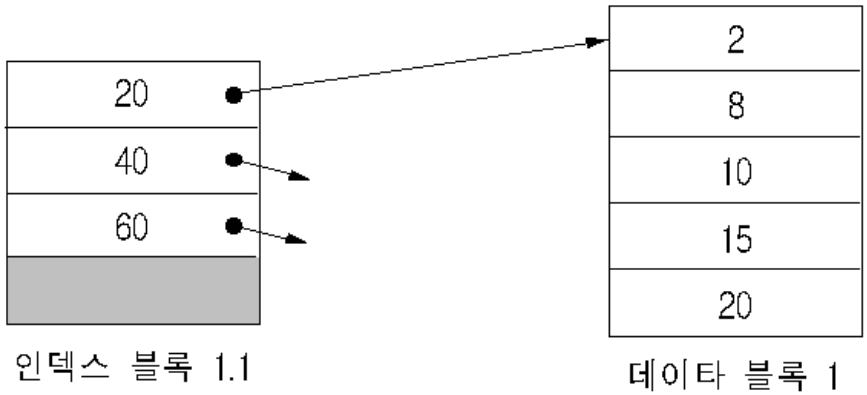
▶ 삽입

◆ 삽입 전의 화일 상태

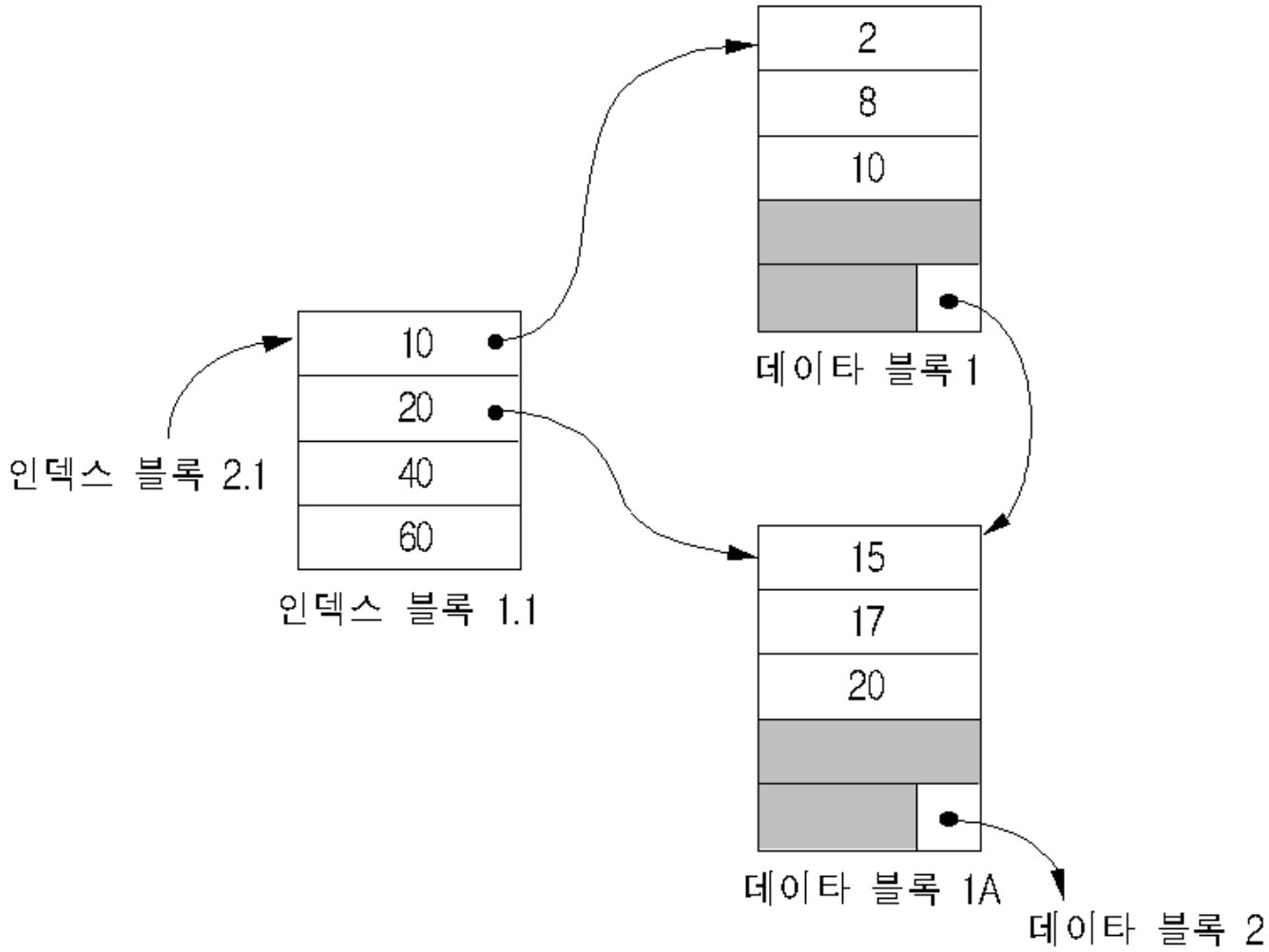

데이타 화일
2
10
20
데이타 블록 1
31
32
40
데이타 블록 2
42
44
60
데이타 블록 3
72
74
80
데이타 블록 4
295
315
320
데이타 블록 N
20
40
60
인덱스 블록 1.1
80
95
110
인덱스 블록 1.2
250
270
320
인덱스 블록 1.k
60
110
130
인덱스 블록 2.1
190
200
240
320
인덱스 블록 2.2
130
320
인덱스 블록 3.1


20
40
60
2
10
20
인덱스 블록 1.1
데이타 블록 1

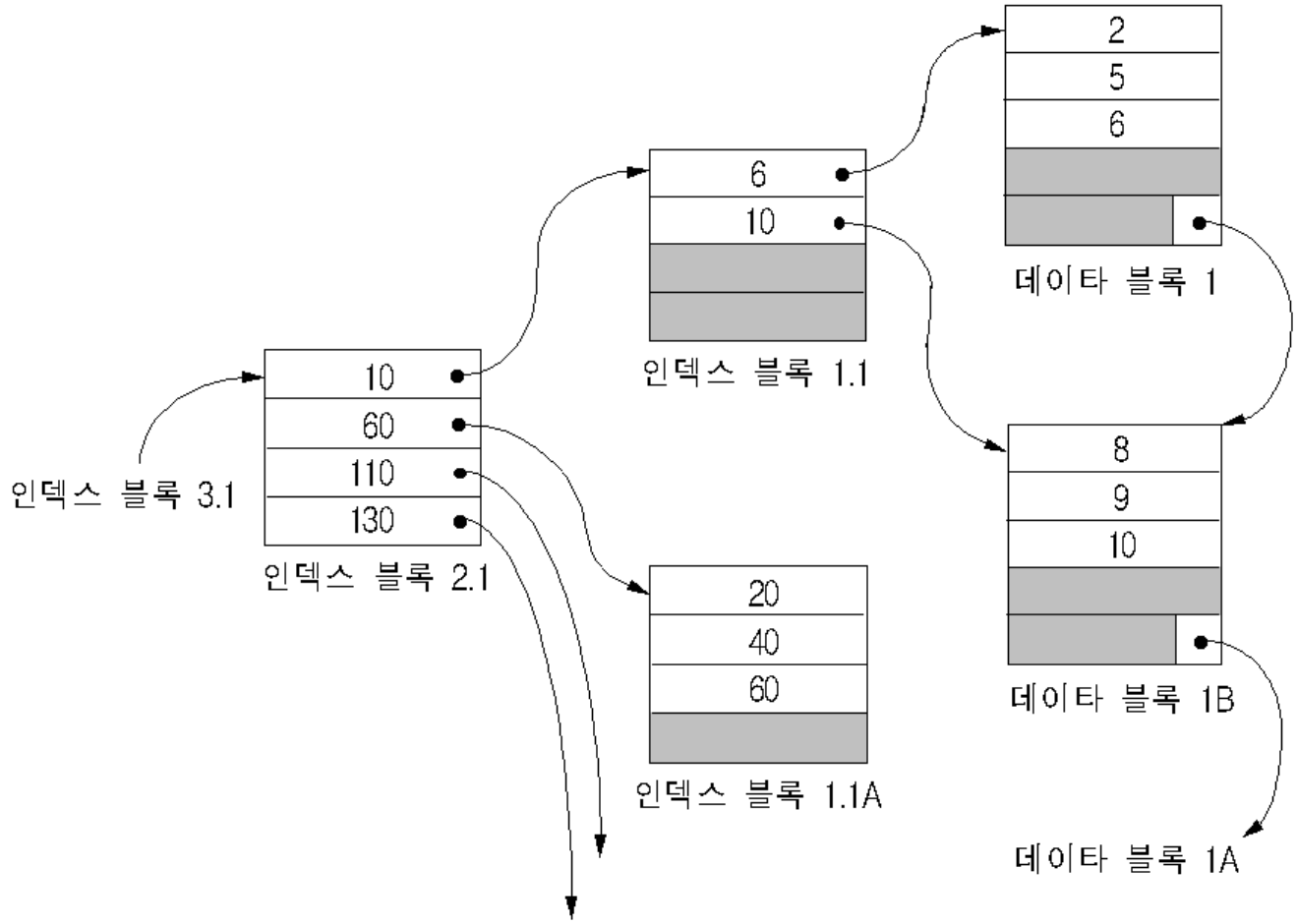
◆ **8, 15를 삽입**


20
40
60
2
8
10
15
20
인덱스 블록 1.1
데이타 블록 1

◆ 17을 삽입


2
8
10
데이타 블록 1
10
20
40
60
인덱스 블록 2.1
인덱스 블록 1.1
15
17
20
데이타 블록 1A
데이타 블록 2

◆ **6, 9, 5를 차례로 삽입**


2
5
6
6
10
데이타 블록 1
인덱스 블록 1.1
10
60
110
130
인덱스 블록 3.1
인덱스 블록 2.1
8
9
10
데이타 블록 1B
20
40
60
인덱스 블록 1.1A
데이타 블록 1A

▶ 인덱스된 순차 화일의 문제점

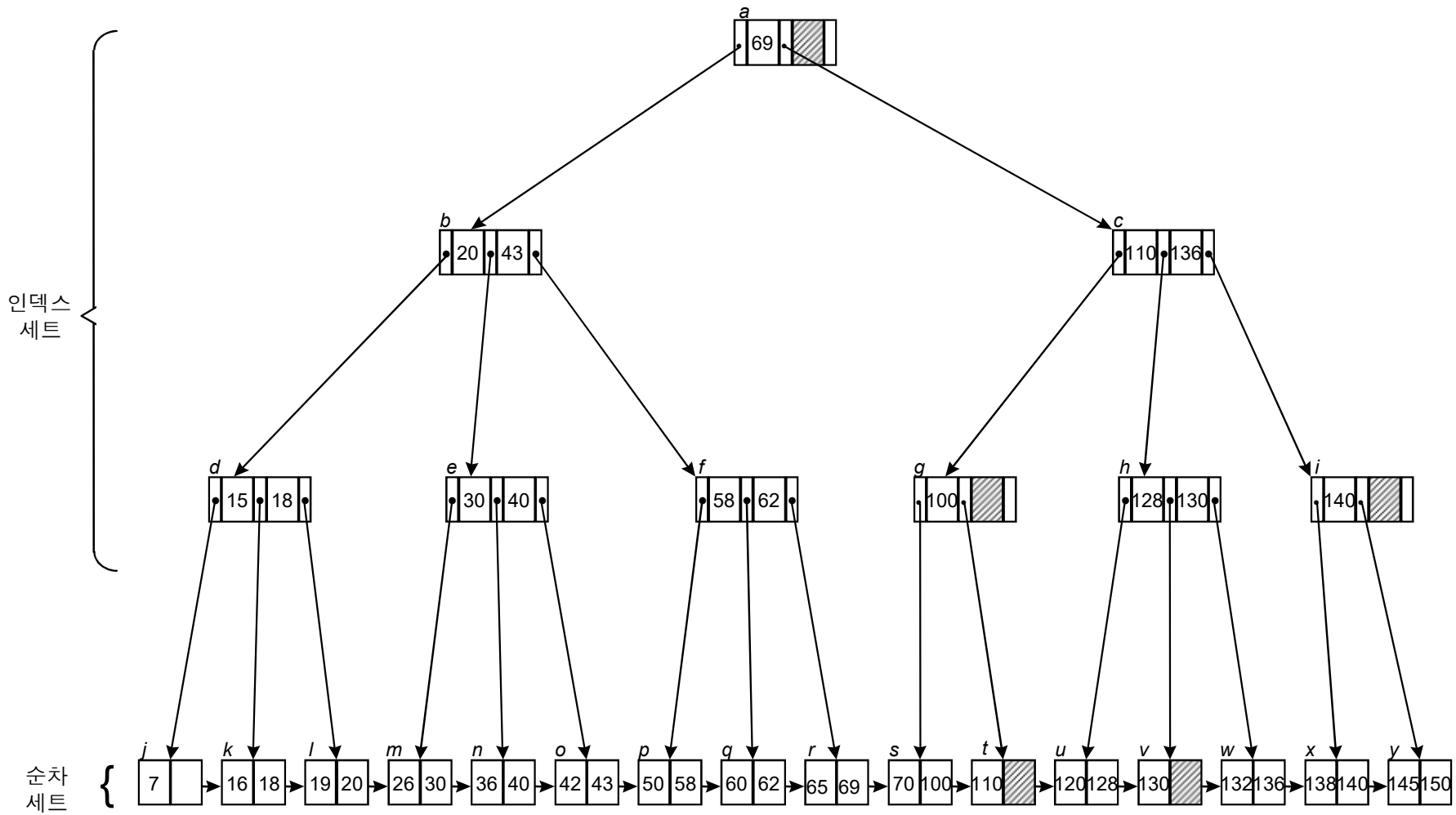
- 데이터 화일
  - 순차적으로 정렬되어 있어야 함.
  - 데이터 블럭들이 정렬되어 연결되어 있어야 함.

❖ B+-트리

- **B+-트리**
  - 직접 접근 및 순차 접근 모두의 속도 향상
    - 직접 접근 및 순차 접근 모두가 필요한 응용에 적합
  - Knuth

- 구조
  - 인덱스된 순차화일의 인덱스 화일이 두 부분으로 나뉨.
    - 인덱스 셑 (index set)
      - 내부 노드
      - 키값만 존재
      - 리프에 있는 키들에 대한 경로 정보만 제공
    - 순차 셑 (sequence set)
      - 리프 노드
      - 내부 노드와 다른 구조: 키값과 레코드 주소가 같이 존재
      - 인덱스 셑의 모든 키 값들을 포함
      - 순차셑은 순차적으로 연결
  - 데이터 화일이 정렬되어 연결되지 않음.
    - 레코드가 랜덤하게 저장됨.

▶ 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
7
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

▶ 차수가 m인 B+-트리의 노드 구조

- 내부 노드의 구조
  - $<n, p_0, K_1, P_1, K_2, P_2, \ldots, P_{n-1}, K_n, P_n>$
  - n ($\lceil m/2 \rceil - 1 \le n \le m-1$): 내부 노드 내의 키 값의 수
  - P_i ($0 \le i \le n$) : 서브트리에 대한 포인터
  - K_i ($1 \le i \le n$) : 키 값

- 리프 노드의 구조
  - $<q, <K_1, A_1>, <K_2, A_2>, \ldots, <K_q, A_q>, P_{next}>$
  - q ($\lceil m/2 \rceil \le q \le m$): 리프 노드 내의 키 값의 수
  - K_i ($1 \le i \le q$) : 키 값
  - A_i ($0 \le i \le q$) : 키 값으로 K_i를 가진 레코드에 대한 포인터
  - P_{next} : 순차 셑의 다음 블록에 대한 포인터

▶ 차수가 m인 B+-트리의 특성

<MST의 특성>

① 한 노드 안에 있는 키 값들은 오름차순으로 정렬
($1 \le i \le n-1$에 대해 $K_i < K_{i+1}$)

② P_i, $(0 \le i \le n-1)$가 지시하는 서브트리에 있는 노드들의 모든 키 값들은 키 값 K_{i+1}보다 작거나 같음

③ P_n이 지시하는 서브트리에 있는 노드들의 모든 키 값은 키 값 K_n보다 큼

<B-트리의 특성>

④ 루트는 “0” 또는 “2에서 m개 사이”의 서브트리를 가짐

⑤ 루트와 리프를 제외한 모든 내부 노드는 “최소 [m/2]개”, “최대 m개”의 서브트리를 가짐.

⑥ 리프가 아닌 노드에 있는 키 값의 수는 그 노드의 서브트리 수보다 하나 적음.

⑦ 모든 리프 노드는 같은 레벨

<B+-트리의 특성>

⑧ 리프 노드는 화일 레코드들의 순차 세트를 나타내며 모두 링크로 연결되어 있음

▶ B+-트리의 B-트리와의 차이

- 인덱스 세트와 순차 세트의 구분이 있으며 구조가 다름
  - 인덱스 셑은 키 값만 저장하고, 순차 셑에는 키 값과 그 키 값을 가진 레코드에 대한 포인터를 저장함.
    - 인덱스 셑은 데이터를 찾는 경로로만 사용
    - 순차 셑은 실제 데이터를 접근하는 데 이용.
  - Pnext는 인덱스 셑에는 없고, 순차 셑에만 존재.
    - 순차 셑의 모든 노드가 연결 리스트로 연결 (효율적인 순차 접근 가능)
  - 인덱스 셑에는 분기를 위한 포인터가 있고, 순차 셑에는 없음.
- 인덱스 셑에 있는 키 값은 순차 셑에 다시 나타남

▶ B+-트리 연산

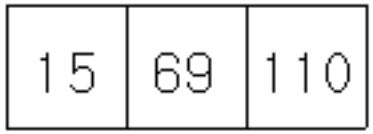
- 연산
  - 검색
    - Random access: 인덱스 세트 검색 = MST 검색 알고리즘
    - Sequential access: 순차 세트(리프) 검색
  - 삽입
    - B-트리와 유사
    - 오버플로우(분열) → 부모노드, 분열노드 모두에 키값 존재
  - 삭제
    - 재분배, 합병 없는 경우: 리프에서만 삭제
    - 재분배시 : 인덱스의 키값도 삭제

▶ B+-트리의 삽입

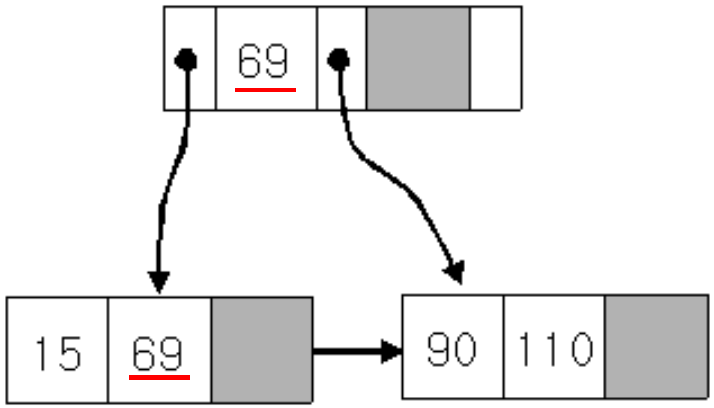
- **B-**트리의 리프 노드에 삽입하는 것과 유사
  - 리프 노드(순차 세트) 분할 시
    - 중간 키 값(분할된 왼쪽 노드에서 제일 큰 키 값)의 복사본이 부모(인덱스 노드)로 올라가 삽입됨
  - 인덱스 노드(인덱스 세트) 분할 시
    - B-트리와 똑같은 알고리즘을 수행하고, 중간 키는 부모 노드로 올라감

▶ 삽입 예 (m=3)

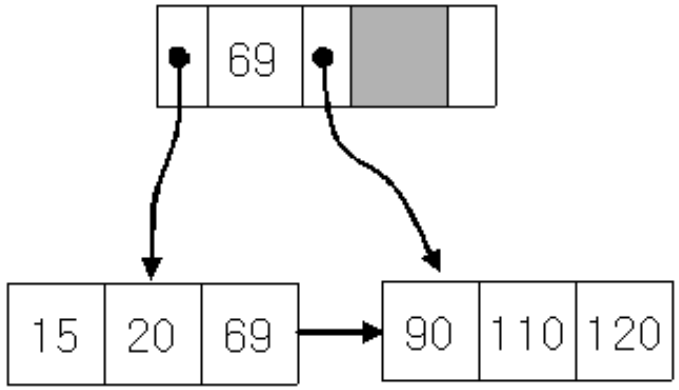
- 처음 키 값 **15, 69, 110** 삽입 (순차 세트의 첫번째 블록 생성됨)



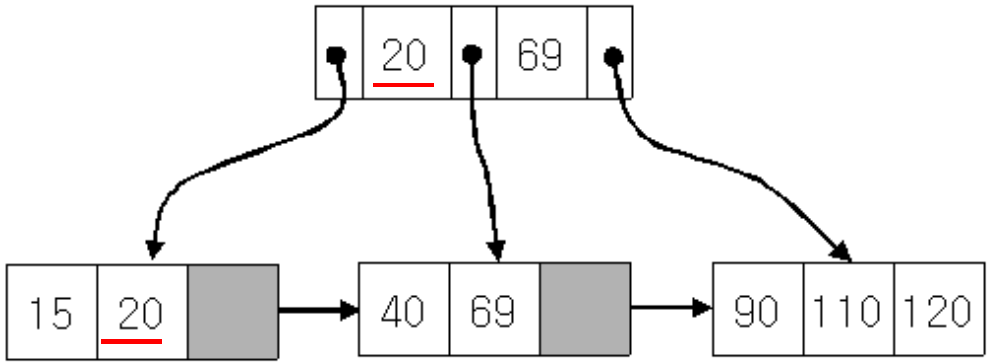
- 키 값 **90**의 삽입 (인덱스 세트의 첫번째 노드 생성됨)



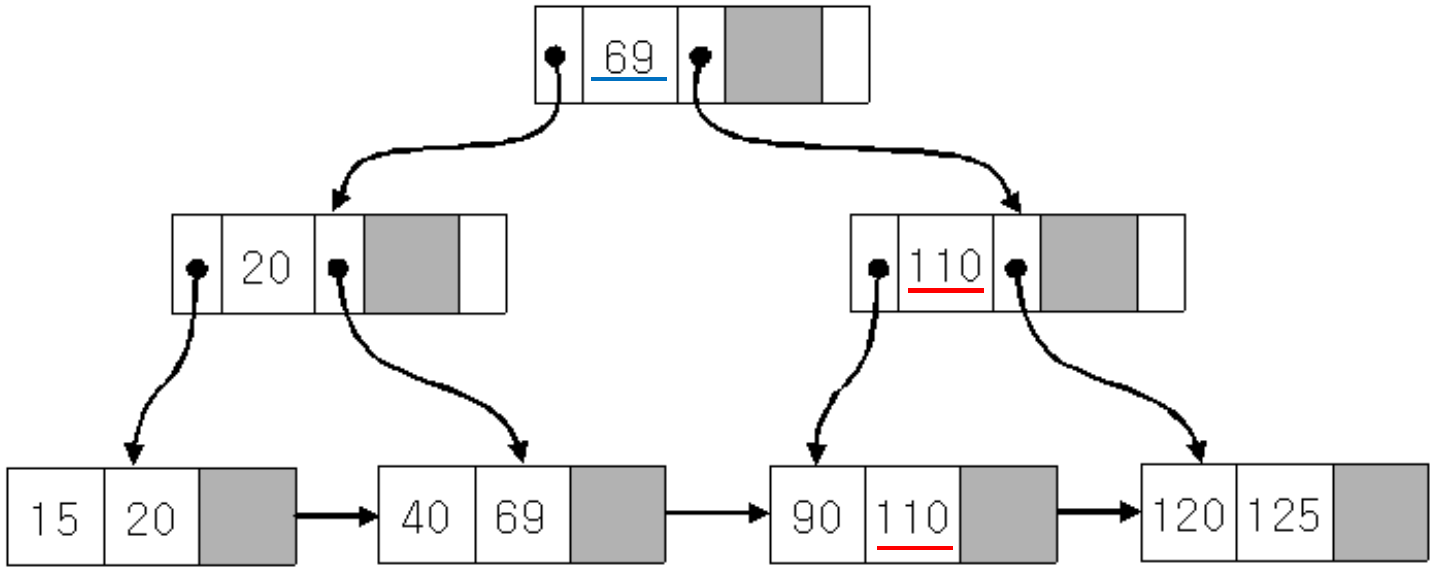
◆ 키 값 **20, 120**의 삽입



◆ 키 값 **40**의 삽입



◆ 키 값 **125**의 삽입



▶ B+-트리에서의 키 값의 삭제

◆ B-트리와 유사

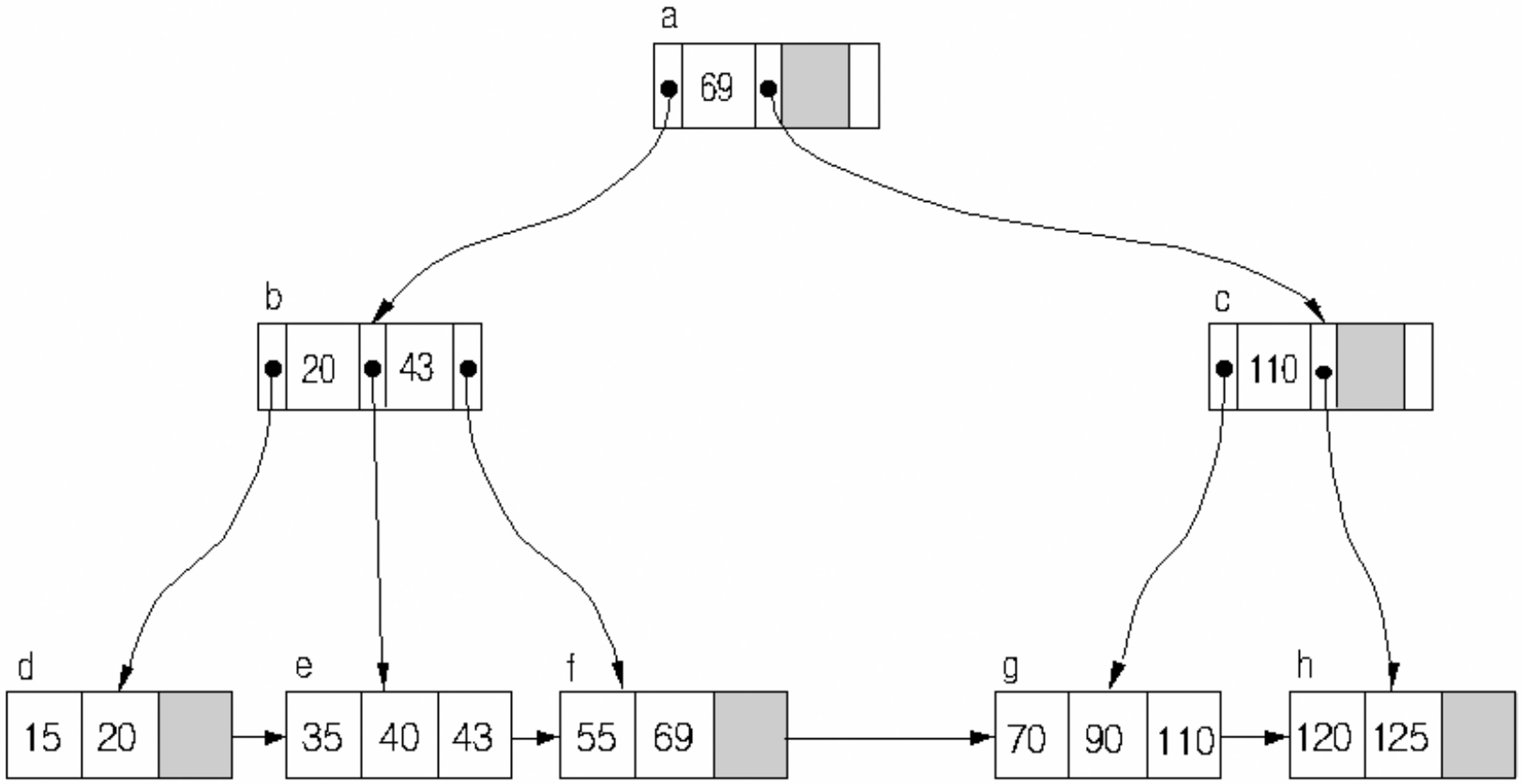
- 차이 : 키 값의 삭제는 리프노드에서만 수행

- 인덱스 세트의 키 값을 삭제할 필요가 있는 경우
  - 삭제하지 않고, 분리자(seperator)로 이용
    (다른 키 값을 탐색하는 데 이용)

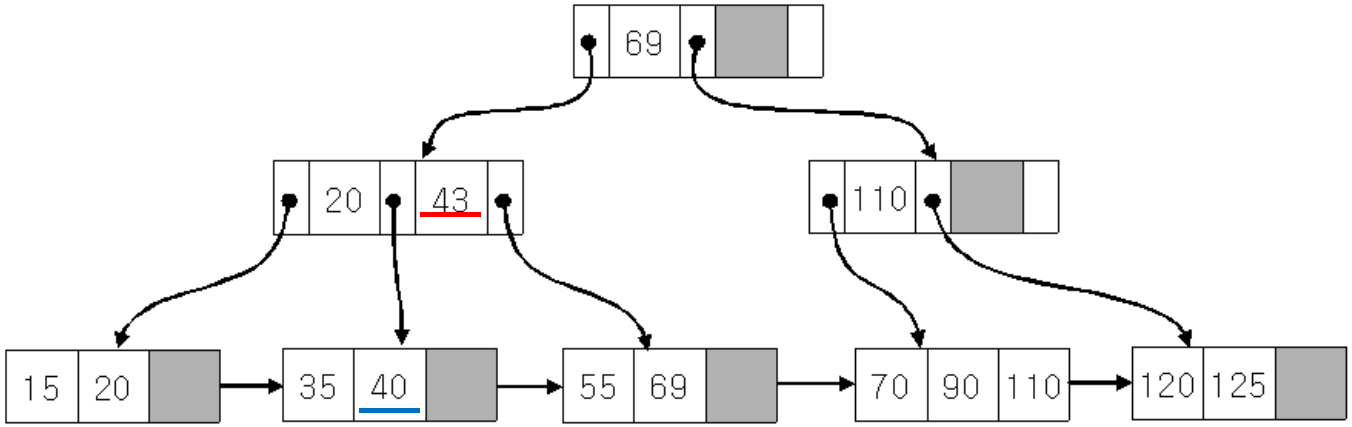
- Underflow가 되는 키 개수의 기준
  - 내부 노드 : $\lceil m/2 \rceil - 1 \leq n \leq m-1$
  - 리프 노드 : $\lceil m/2 \rceil \leq q \leq m$

▶ 삭제 예

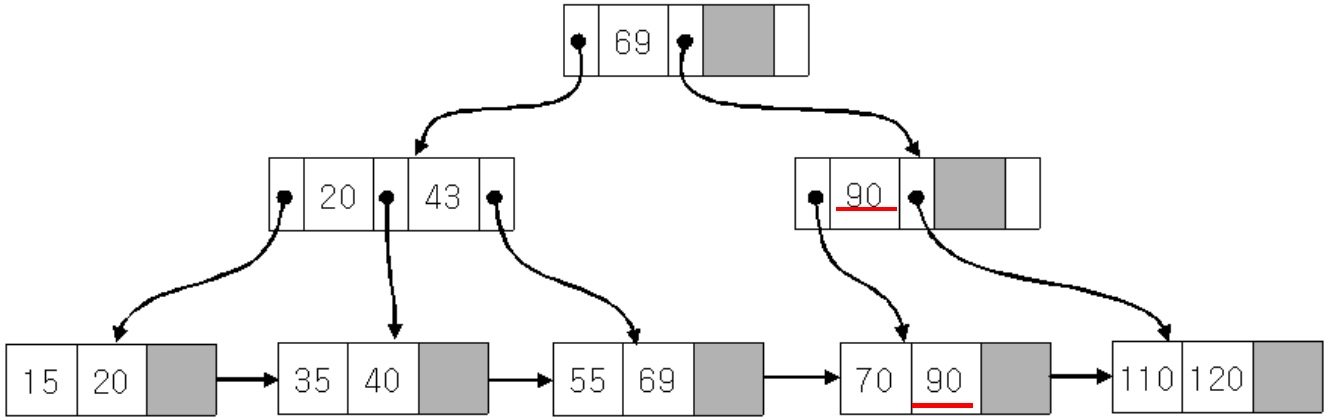
◆ 삭제 전 **B+-**트리



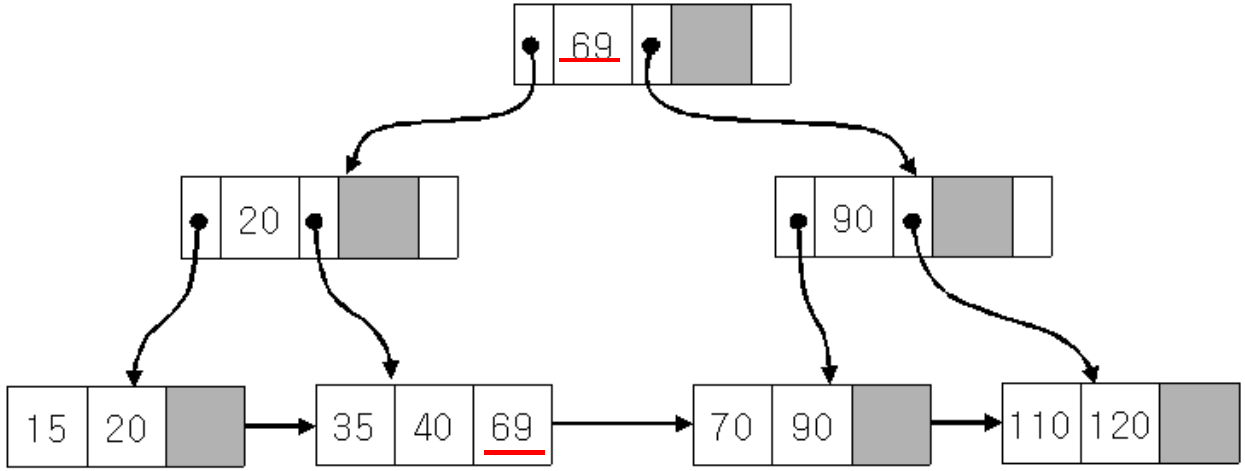
◆ 키 값 **43**의 삭제



◆ 키 값 **125**의 삭제



◆ 키 값 55의 삭제



▶ Range Search (범위 검색)

- 범위 검색
  - 임의 검색 + 순차 검색

▶ B+-트리의 성능 (B-트리와의 비교)

- 특정 키 값의 검색
  - 단점
    - 검색이 항상 리프 노드까지 내려가야만 종료
      - 검색하고자 하는 값이 인덱스 세트에서 발견되더라도 리프 노드까지 내려가야만 실제 레코드의 포인터를 알 수 있음
  - 장점
    - 인덱스 셑의 노드는 포인터를 저장하지 않으므로, 노드 내 공간이 절약됨.
- 순차 검색
  - 연결 리스트를 순회 → 효율적
- **B+-트리는 직접 처리와 순차 처리를 모두 필요한 응용에서 효율적**

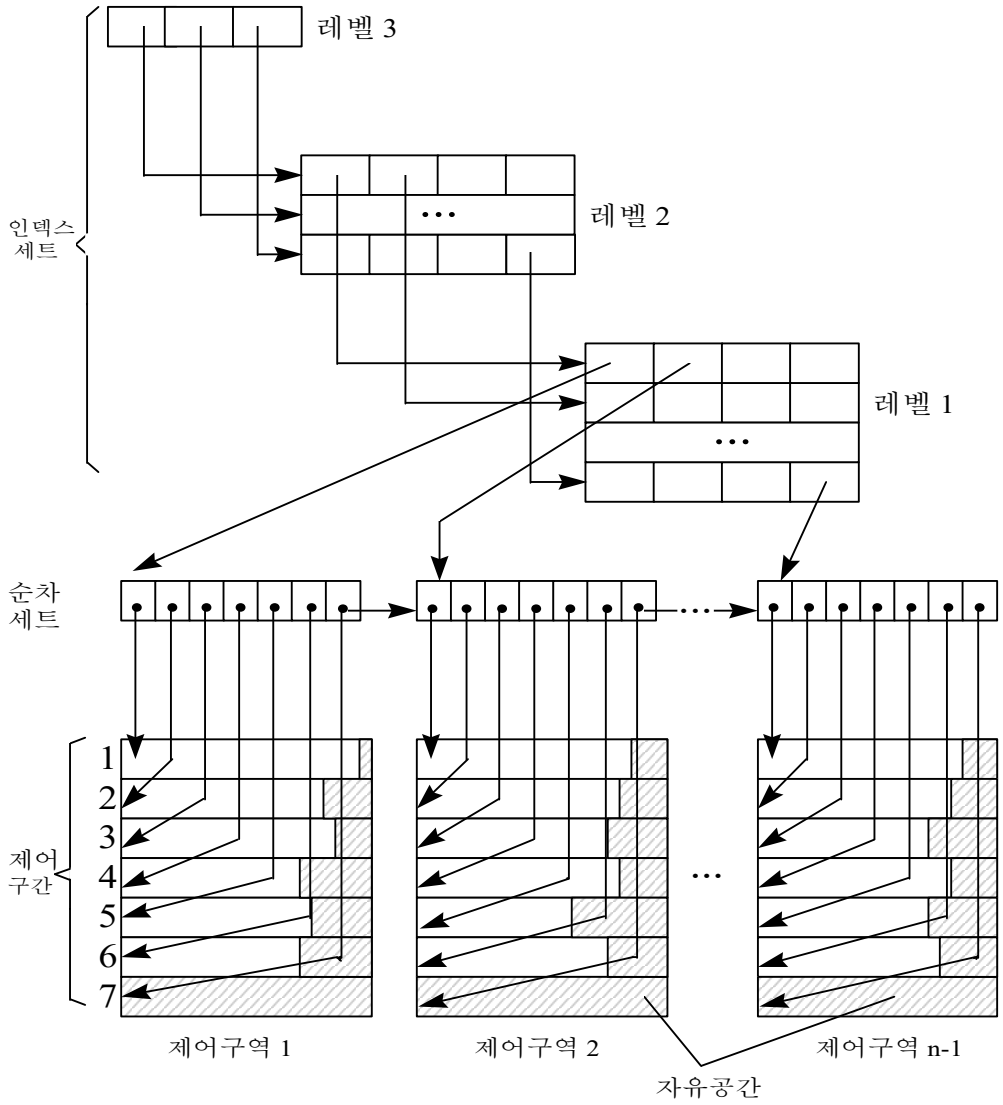
❖ VSAM 화일

- **VSAM : Virtual Storage Access Method**

- **B+-트리 인덱스 구조 기법을 이용하는 대표적인 인덱스된 순차 화일 구성 방법**

- **VSAM 화일의 구조**
  - 제어 구간(control interval)
    - 데이타 레코드 저장
  - 제어 구역(control area)
    - 제어 구간의 모임
  - 순차 세트(sequence set)
    - 제어 구역에 대한 인덱스 저장
  - 인덱스 세트(index set)
    - 순차 세트의 상위 인덱스
    - 여러 단계로 구성

(1) VSAM 화일 구조



▶ 제어 구간의 구조

- 제어 구간
  - 데이타 블록 : 하나 이상 레코드의 저장 공간
  - 자유 공간 (padding) 유지 : 추가 데이타 삽입 위해
  - 키 값에 따라 물리적 순차로 저장
  - 제어 정보(control information)
    - 데이타 레코드와 자유공간에 대한 정보
    - 제어 구간 뒤쪽부터 저장됨

  - 제어 구간의 구조

<table>
<tr><td>레코드 1</td><td>레코드 2</td><td>레코드 3</td><td>레코드 4</td></tr>
<tr><td>레코드 5</td><td>레코드 6</td><td colspan="2">자유 공간</td></tr>
<tr><td colspan="2">자유 공간</td><td>레코드
제어정보</td><td>자유공간
제어정보</td></tr>
</table>

▶ 제어 구역, 순차 세트, 인덱스 세트

- 제어 구역
  - 일정 수의 제어 구간의 그룹
  - 어느 한 제어 구간의 오버플로를 제어 구역 내에서 해결하려는 목적
  - 자유 공간 유지 : 제어 구역의 마지막 제어 구간

- 순차 세트
  - 인덱스 엔트리=(제어 구간의 최대키값, 주소)
  - 제어 구간의 물리적 순서는 필요 없음
    - 순차 세트에 의해 유지(체인으로 연결) : 순차 접근이 용이

- 인덱스 세트
  - 인덱스 블록으로 구성된 m-원 탐색 트리 구조
  - 엔트리= (차하위 레벨의 인덱스 블록의 최대키값, 인덱스 블록에 대한 주소)
  - 최하위 단계 인덱스 세트 : 순차 세트 지시
  - 키 값의 크기 순으로 저장

(2) VSAM 화일에서의 연산

- 키 순차 화일(key-sequenced file) 지원
  - 레코드들을 키 값 순으로 저장
    - 제어 구간내의 물리적 순서와 일치
    - 제어 구간내의 키 값 순서 : 순차 세트 엔트리로 유지
    - 순차 세트의 순서 : 인덱스 세트 엔트리로 유지
  - 분산 자유 공간 (distributed free space)
    - 제어 구간 내의 자유 공간
    - 각 제어 구역 끝의 빈 제어 구간을 할당

- 순차 접근
  - 화일의 첫 제어 구간을 식별하기 위해 인덱스 탐색
  - 연결된 순차 세트 체인을 따라가며 모든 제어 구간 접근

- 직접 접근
  - 인덱스 세트 -> 순차 세트 (B+-트리와 유사)

▶ 레코드의 삽입과 삭제

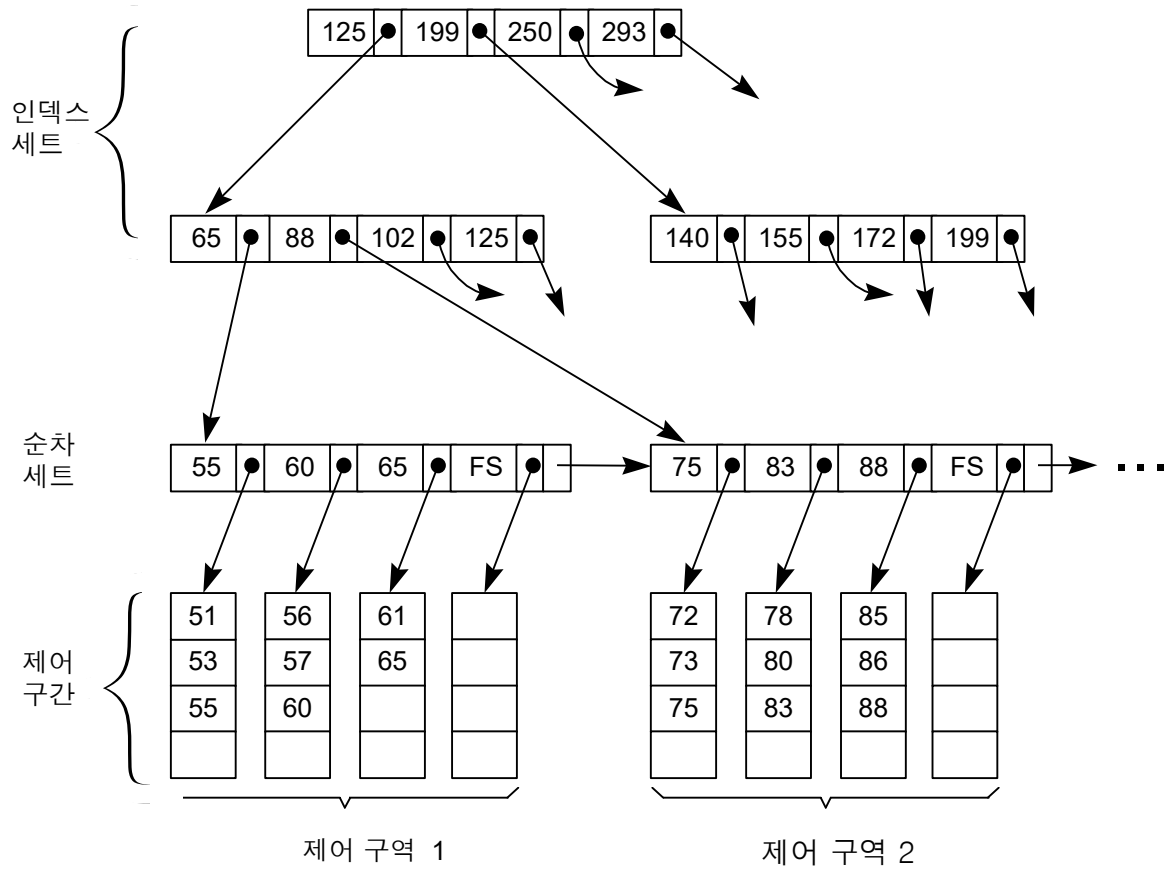
- 레코드의 삽입, 삭제
  - 제어 구간 내 레코드의 이동 필요
    ∵제어 구간 내의 물리적 순서 유지
  - 순차 세트 엔트리 변경 → 인덱스 세트까지 파급

- 제어 구간 분열(control interval split) : 제어 구간 내 자유 공간 없을 때
  - 만원이 된 제어 구간
    - 제어 구역 끝의 빈 제어 구간으로 레코드 절반 이동
    - 순차 세트 엔트리로 제어구간 순서 유지

- 제어 구역 분열(control area split) : 제어 구역 내에 빈 제어구간이 없을 때
  - 만원이 된 제어 구역
    - 화일 끝의 빈 제어구역으로 절반 이동
    - 순차 세트 체인을 조정해서 제어 구역 순서 유지

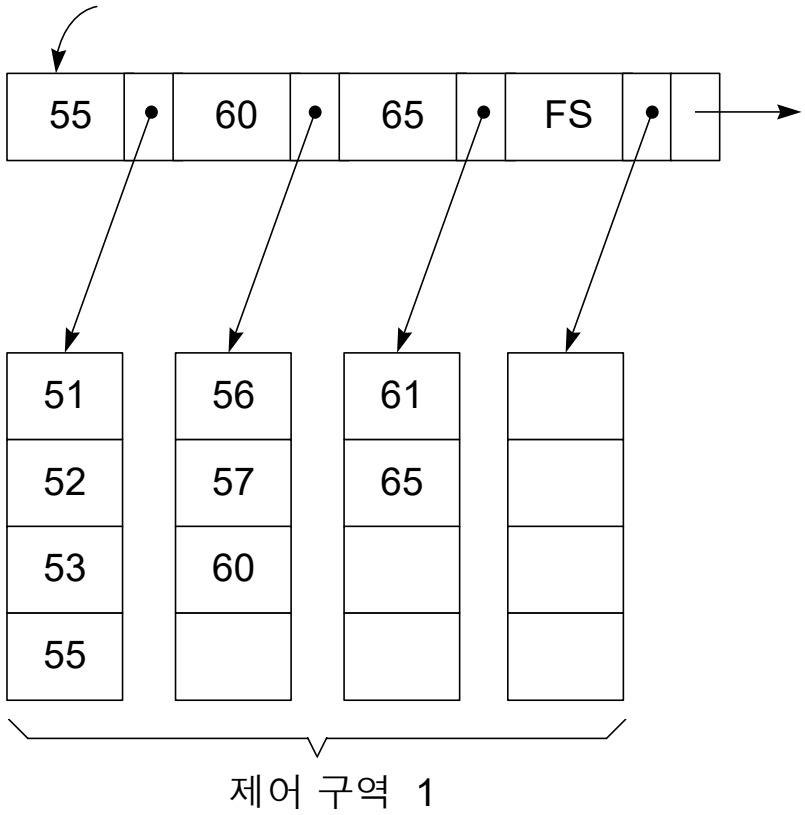
▶ 삽입 예(1)

◆ 삽입이 이루어지기 전의 **VSAM** 화일



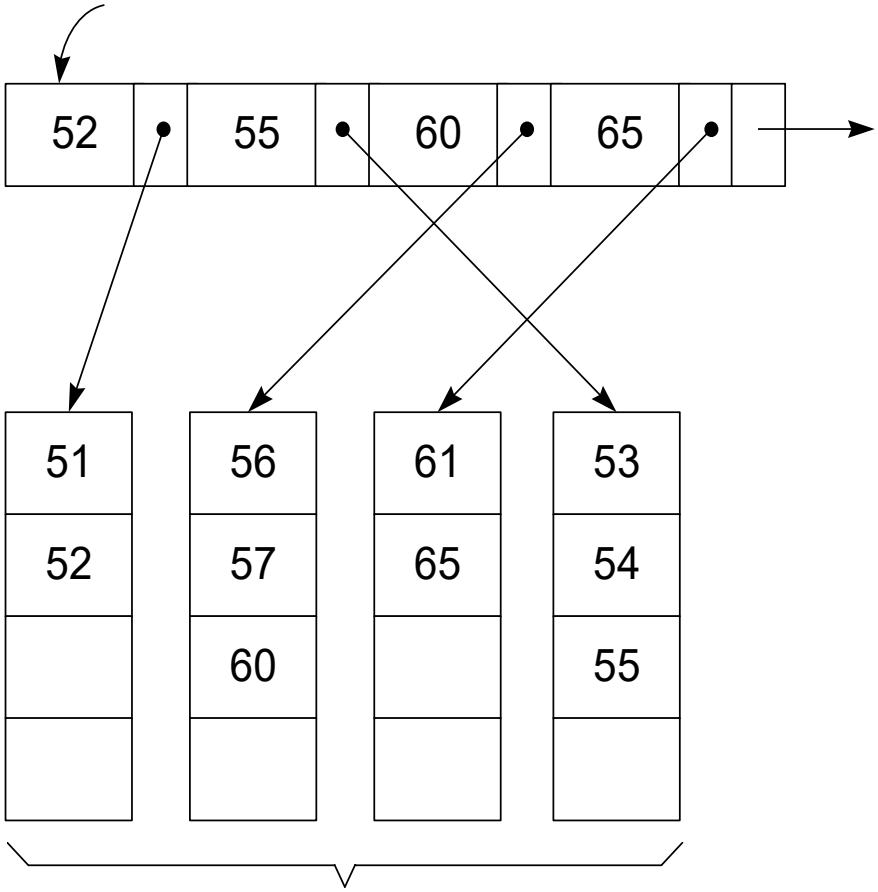
▶ 삽입 예(2)

◆ 키가 52인 레코드가 삽입된 뒤의 제어 구역



▶ 삽입 예(3)

- 키가 **54**인 레코드가 삽입되어 제어 구간 분열이 일어난 뒤의 제어 구역 **1**



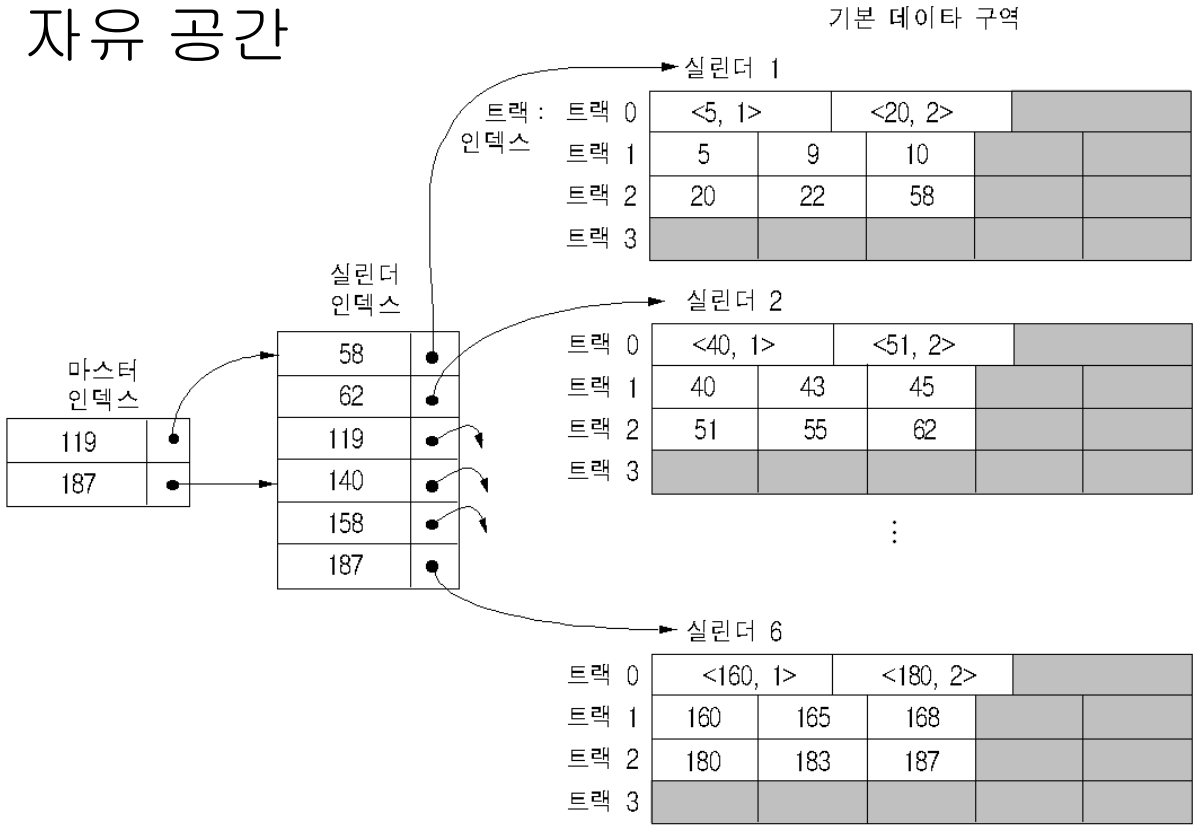
❖ ISAM 화일

- **IBM의 ISAM(Indexed Sequential Access Method)**

- 인덱스, 기본 데이타 구역**(prime data area)**과 오버플로 구역**(overflow area)**으로 구성
  - 인덱스 : 마스터 인덱스(master index), 실린더 인덱스(cylinder index), 트랙 인덱스(track index)로 구성

(1) ISAM 화일의 구조

- **ISAM 화일의 예**
  - 기본 데이터 구역은 6개 실린더로 구성
  - 실린더당 4 트랙(한 트랙은 오버플로 구역)
  - 트랙당 5개의 레코드 수용
  - 트랙당 40%의 자유 공간



▶ ISAM 화일의 인덱스 구조

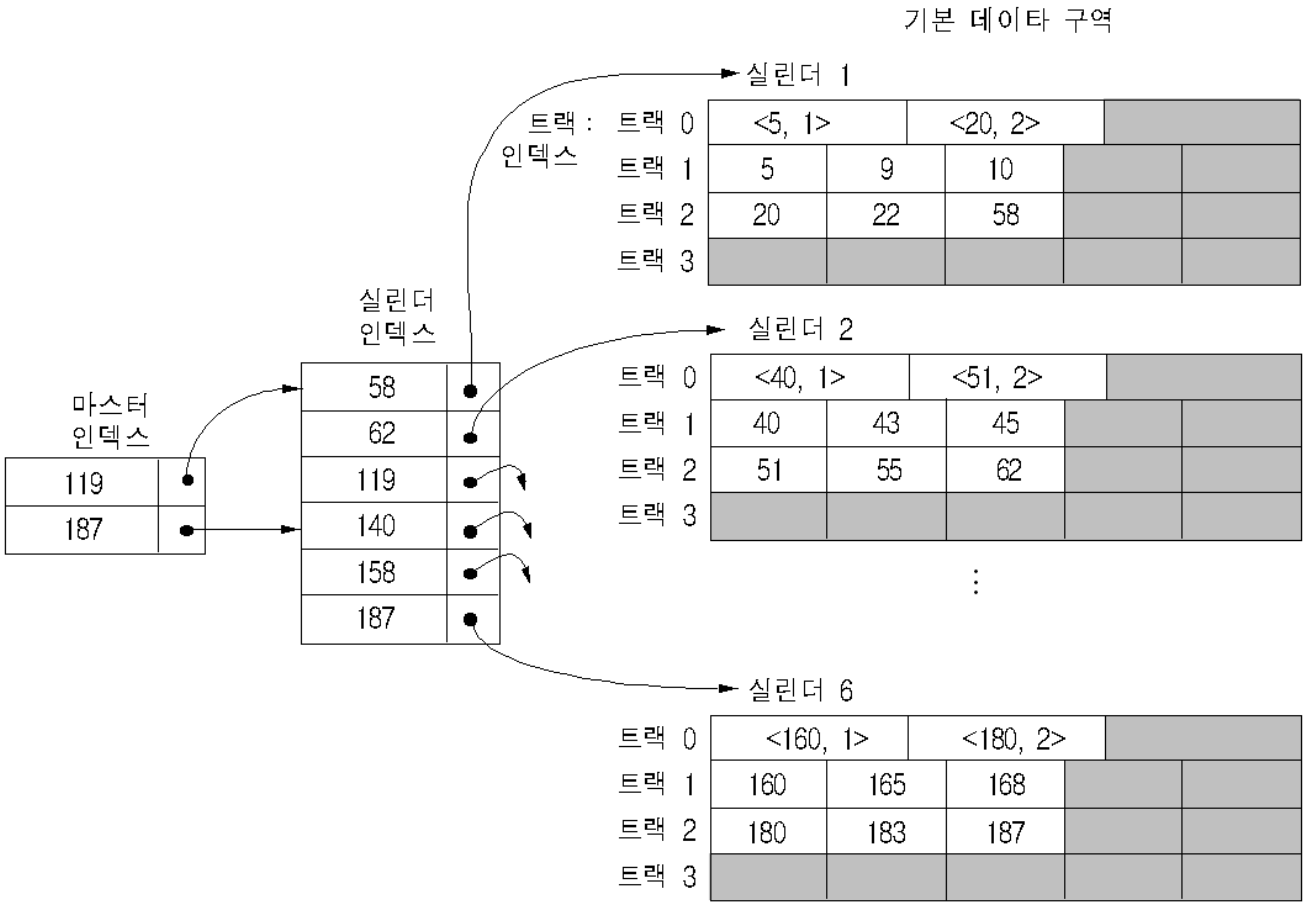
- **마스터 인덱스(master index)**
  - 최상위 레벨의 인덱스
  - 각 엔트리는 <최대 키 값, 포인터> 로 구성
    - 포인터는 해당 키 값을 가진 실린더 인덱스 엔트리를 가리킴
  - 메모리에 상주

- **실린더 인덱스(cylinder index)**
  - 기본 데이타 구역에 대한 포인터를 가짐
  - 각 엔트리는 <최대 키 값, 실린더 번호> 로 구성

- **트랙 인덱스(track index)**
  - 각 실린더의 첫번째 트랙(트랙 0)에 저장되어 있음
  - 각 엔트리는 <최소 키 값, 트랙 번호> 로 구성

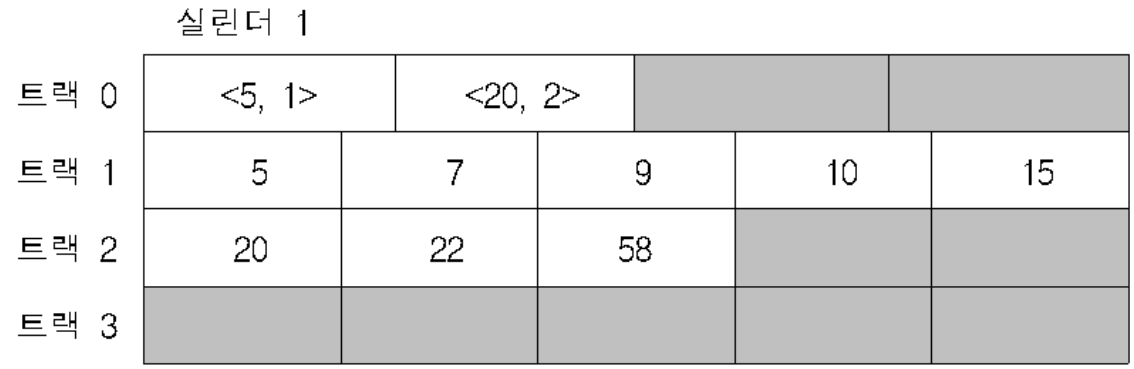
▶ ISAM 화일에서의 레코드 삽입(1)

◆ 삽입 전의 ISAM 화일



▶ ISAM 화일에서의 레코드 삽입(2)

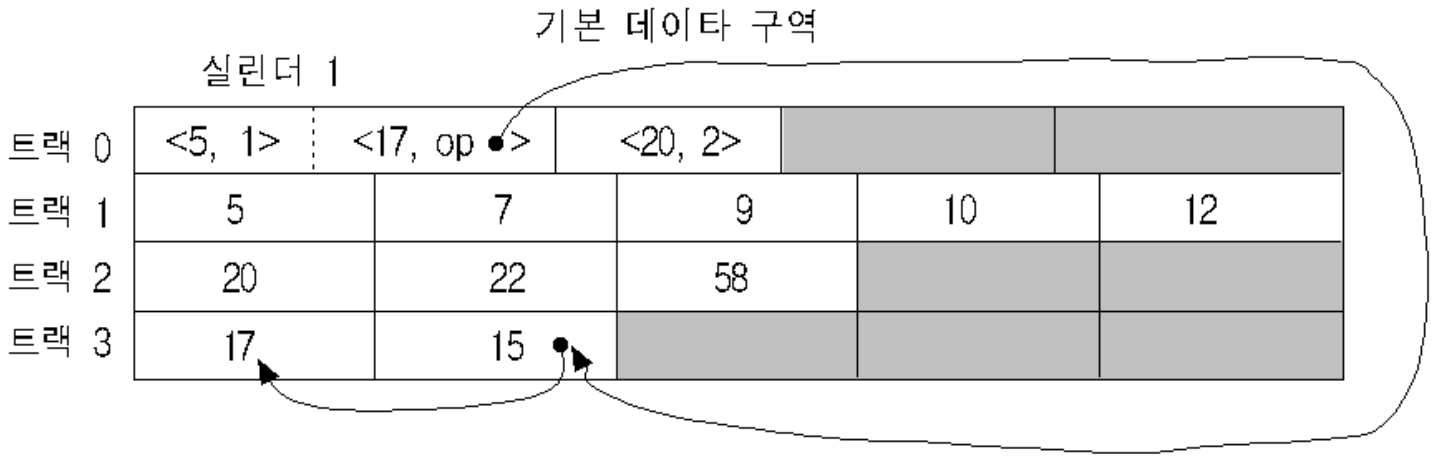
◆ 레코드 15와 7을 삽입



◆ 레코드 17을 삽입

▶ ISAM 화일에서의 레코드 삽입(3)

- 레코드 **12**를 삽입



- 레코드의 삽입이 많은 경우 오버플로 체인이 길어져 성능 저하 → 주기적인 화일 재구성 필요

▶ ISAM 화일에서의 레코드 삭제(1)

- **방법 1 : 삭제할 레코드를 물리적으로 삭제**
  - 삭제할 레코드가 기본 데이타 구역에 있을 경우
    - 물리적 순차를 지키기 위해 트랙 내 레코드 조정
    - 오버플로 트랙에 레코드가 있는 경우 기본 구역으로 가져옴
  - 삭제할 레코드가 오버플로 구역에 있을 경우
    - 연결 리스트를 적절히 조정

- **방법 2 : 삭제할 레코드를 물리적으로 삭제하지 않고 삭제 표시만 함**
  - 검색시에 삭제 표시된 레코드는 생략
  - 일정한 주기로 삭제 표시된 레코드 정리 필요
  - 단점
    - 공간의 낭비
    - 오버플로 레코드의 불필요한 발생

❖ 인덱스된 순차 화일의 설계

◆ 설계 시의 고려사항

① 레코드의 필드 수와 배치
② 레코드의 키 필드와 크기
- 고정 / 가변 길이 레코드를 모두 수용할 수 있도록
- 화일에 대한 직접 접근이 용이한 키 선택
③ 예상 레코드 추가 삽입 레코드 수
- 일반적으로 약 **40%**의 자유공간 할당
④ 화일의 구현 방법

▶ 설계 결정을 위한 매개 변수

① 데이타 블록의 크기
② 인덱스 블록의 크기
③ 초기 인덱스 레벨 수
④ 최대 인덱스 레벨

→ 실제 값은 시스템 본래의 블록 크기, 현재 또는 예상되는 데이타 레코드의 수, 화일의 용도 등에 따라 결정

★ 개별 레코드에 대한 직접 검색이 주로 요구되는 응용 : 밀집 인덱스 사용

순차 검색이 주로 요구되는 응용 : 데이타 블록을 크게 하거나 기본 구역과 블로킹 인수를 크게 하는 것이 유리

▶ 인덱스의 중요 매개 변수

◆ 인덱스 블록의 참조 능력 : 인덱스 분기율(fanout)

$$y = \lfloor B / (V+P) \rfloor$$

- y : 인덱스 분기율
- B : 블록 크기
- V : 키 값 길이
- P : 포인터 길이
- (V+P) : 인덱스 엔트리 크기

◆ 인덱스 레벨 vs 인덱스 분기율
- 인덱스 분기율이 크면 → 인덱스 레벨은 적고
  → 검색 속도는 빠름

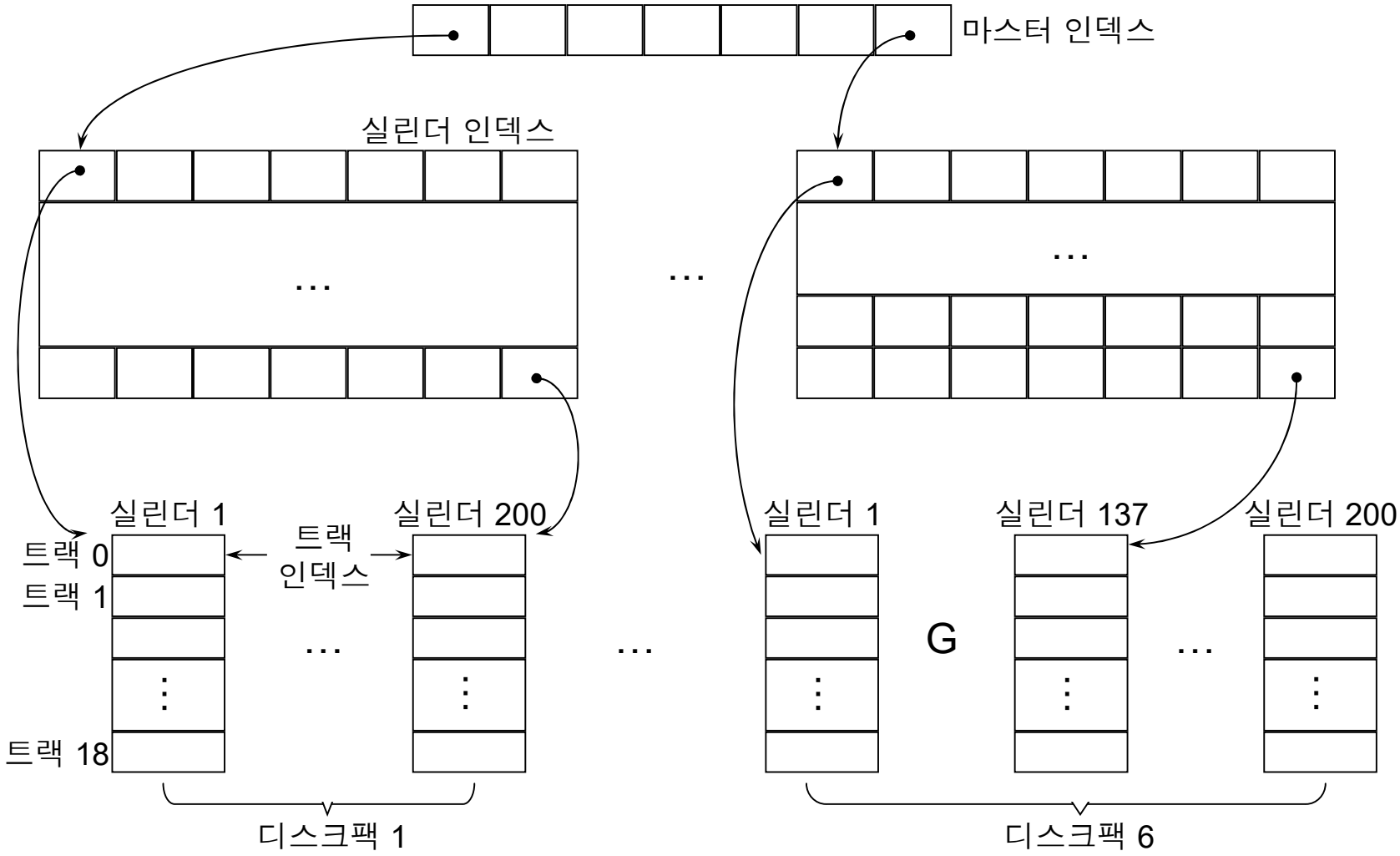
▶ 정적 인덱스의 설계 예

- 블록 크기가 **2000**바이트라 할 때, 다음과 같은 데이타를 저장할 인덱스된 순차 화일을 설계해보자
  - 키 길이 : 14바이트
  - 포인터의 크기 : 6바이트,
  - 레코드 길이 : 200바이트
  - 레코드 수 : 100만 개

- 풀이
  - 블로킹 인수 : 2000/200 = 10
  - 필요한 인덱스 엔트리 : 10만개
  - 인덱스 엔트리의 크기 : 14+6 = 20
  - 인덱스 블록의 분기율 : 2000/20 = 100
  - 최하위 레벨(레벨1)의 인덱스 엔트리 블록 수 : 10만/100 = 1000
  - 레벨 2의 인덱스 엔트리 블록 수 : 1000/100 = 10
  - 레벨 2의 인덱스 블록 10개는 메모리에 상주 시키기엔 너무 크므로 레벨 3의 인덱스를 만들어야 함(마스터 인덱스)

- 이 예에서, 레코드 접근시 총 **3**번의 디스크 접근 필요
  - 인덱스 접근 2번(마스터 인덱스는 메모리 상주) + 레코드 접근 1번

▶ 정적 인덱스 설계 예


마스터 인덱스
실린더 인덱스
실린더 1
실린더 200
실린더 1
실린더 137
실린더 200
트랙 0
트랙 1
트랙 18
트랙
인덱스
G
디스크팩 1
디스크팩 6