

## 데이터베이스 파일 인덱싱 기법, ~~B-Tree~~ 및 **B<sup>+</sup>-Tree**

- 단일-단계 순서 인덱스들의 유형
  - 기본 인덱스
  - 클러스터링 인덱스
  - 보조 인덱스
- 다단계 인덱스
- 해싱(Hashing)
- B-Tree와 B<sup>+</sup>-Tree를 사용하여 구현하는 동적 다단계 인덱스

### 인덱스의 생성 및 삭제

```
create index citiesx on customers (city);
```

1. without index – table scan (data file)이 필요
2. with index?

```
select * from customers where city = 'Boston' and discnt between 12 and 14;
```

- Dropping index:

```
drop index citiesx;
```

- Unique index :

```
create unique index cidx on customers (cid);
```

- cid가 unique함을 index가 보장함
- 일반적으로 unique나 PK 제약조건에 대해서 DBMS가 자동으로 unique index를 생성함

## Index - Access Path

- 인덱스 - 접근 경로
- 단일 단계 인덱스는 데이터 파일내의 레코드를 효과적으로 찾도록 도와주는 보조 파일임
- 인덱스는 보통 파일내의 한 필드에 대해 정의된다 (여러 필드에 대해 정의될 수도 있음)
- 인덱스는 <필드값, 레코드에 대한 주소>로 구성된 엔트리들을 저장한 파일이다.
- 인덱스 파일은 필드값에 따라 정렬되어 있다.
- 인덱스는 파일에 대한 접근 경로(access path)라고 불린다.
- 인덱스 엔트리는 실제 레코드 크기보다 훨씬 작기 때문에, 인덱스 파일은 데이터 파일보다 훨씬 적은 디스크 블록을 차지한다.
- 인덱스에 대한 이진 탐색으로 데이터 파일의 해당 레코드에 대한 주소를 얻을 수 있다.
- 인덱스는 밀집 또는 희소 인덱스가 될 수 있다.  
인덱스는 정렬되어 있는 순서 파일이다. |<sup>이</sup>2 경로 탐색으로 가능 원래는 트리를 따라 내려온다.
  - 밀집 (dense)인덱스는 데이터 파일내의 모든 탐색 키 값(즉, 모든 레코드)에 대한 인덱스 엔트리를 갖는다.
  - 희소 (sparse 또는 비밀집 nondense) 인덱스는 탐색 값의 일부에 대해서만 인덱스 엔트리를 갖는다.

## Example

- 주어진 데이터 파일: EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ... )
  - 데이터 파일에 대한 가정:
  - 레코드 크기  $R = 150$  바이트
  - 블록 크기  $B = 512$  바이트
  - 레코드 개수  $r = 30000$  레코드

$$Bfr = \frac{B}{R} = \frac{512}{150} = 3$$

기호나 표

얻을 수 있는 값:

- 블록킹 인수(Blocking Factor)  $Bfr = \lceil B \div R \rceil = \lceil 512 \div 150 \rceil = 3$  레코드/블록
- 화일의 블록 수  $b = \lceil r \div Bfr \rceil = \lceil 30000 \div 3 \rceil = 10000$  블록
- SSN 필드에 대한 인덱스에서, 필드 크기  $V_{SSN} = 9$  바이트라고 가정하고,
- 레코드 포인터 크기  $P_R = 7$  바이트라고 가정한다. 그러면:
- 인덱스 엔트리 크기  $R_I = (V_{SSN} + P_R) = (9 + 7) = 16$  바이트
- 인덱스 블록킹 인수(Index Blocking Factor)  $Bfr_I = \lfloor B \div R_I \rfloor = \lfloor 512 \div 16 \rfloor = 32$  엔트리/블록
- 인덱스 블록 수  $b_I = \lceil r \div Bfr_I \rceil = \lceil 30000 \div 32 \rceil = 938$  블록
- 이진 탐색시 접근할 블록 수  $\lceil \log_2 b_I \rceil = \lceil \log_2 938 \rceil = 10$  블록

$$\overline{\overline{30000}}_3$$

- 이 비용은 다음의 평균 선형 탐색 비용보다 훨씬 적은 비용임: 블록 접근수  $(b/2) = 10000/2 = 5000$
- 만약 파일의 레코드들이 정렬되어 있으면, 이에 대한 이진 탐색 비용은 다음과 같다: 블록 접근수  $\lceil \log_2 b \rceil = \lceil \log_2 10000 \rceil = 13.14$

$$\log_2 2^{13} \cdot 10 = 10 + \log_2 10 = 10 + 3 \dots$$

$$(10^6 = 10^3 \cdot 10^3 = 2^6 \cdot 2^{10})$$

## 단일 단계 인덱스의 유형

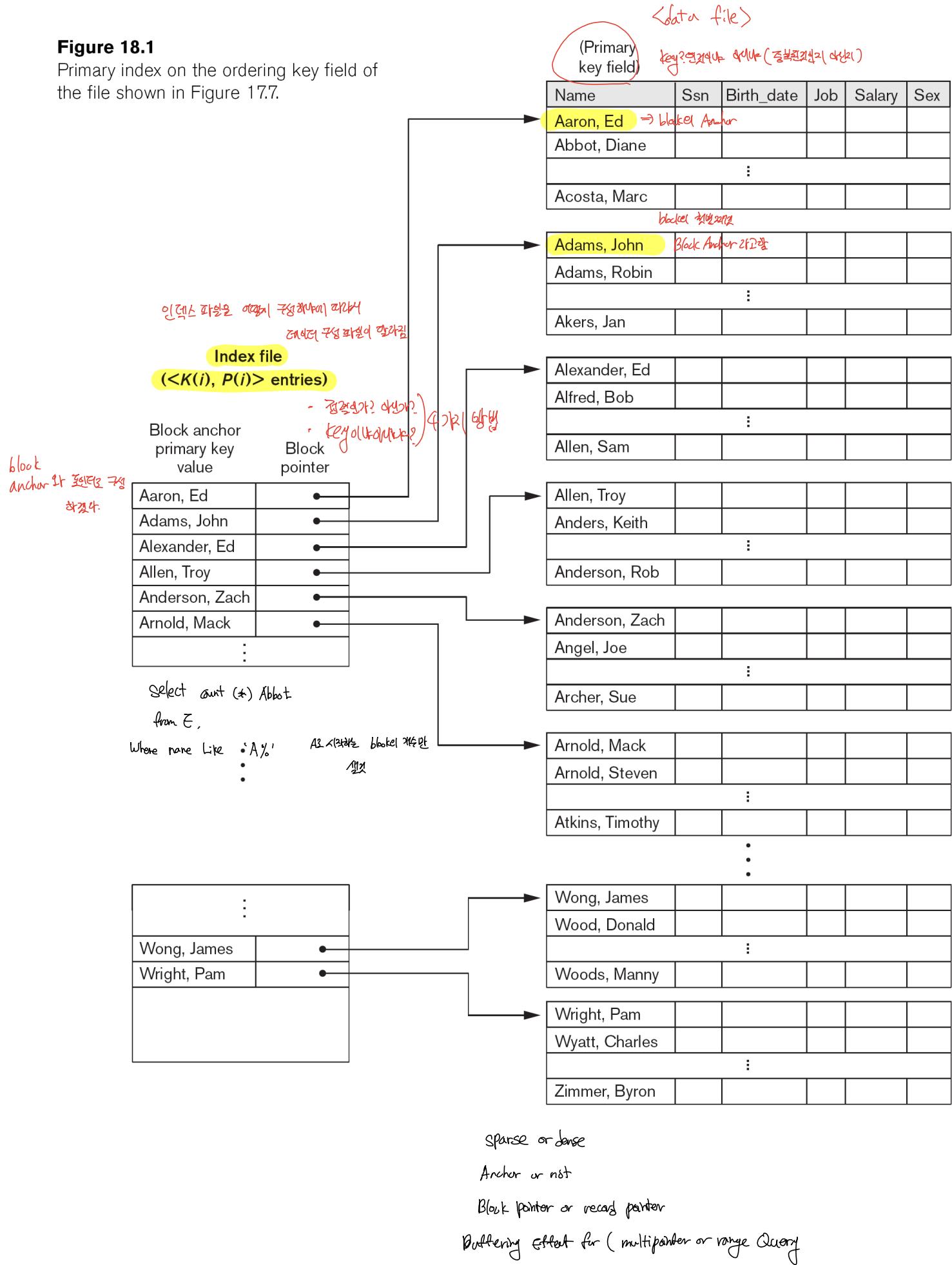
## 기본 인덱스(Primary Index)

- 순서화일(ordered data file)에 대해 정의할 수 있는 인덱스이다.
- 이 데이터 파일은 키 필드 (key field) 순으로 정렬돼 있어야 한다.
- 데이터 파일의 각 블록에 대해 하나의 엔트리를 가지며, 블록 앵커 (block anchor)라 불리는 각 블록의 첫 번째 레코드에 대한 키 필드 값을 엔트리로 갖는다.
- 각 블록의 마지막 레코드를 블록 앵커로 사용하는 방식을 이용할 수도 있다.
- 전체 탐색 값이 아니라 데이터 파일의 각 블록에 대해 하나의 엔트리 (즉, 블록의 앵커 레코드에 대한 키 값)을 가지므로, 기본 인덱스는 비밀집(희소) 인덱스이다.

sparse

**Figure 18.1**

Primary index on the ordering key field of the file shown in Figure 17.7.



## 클러스터링 인덱스(Clustering Index)

데이터 순서화 및 인덱스

nonkey field

- 순서화 파일에 대해 정의할 수 있다.
- 데이터 파일은 각 레코드에 대해 구별된 값을 갖지 않는 필드(즉, 키가 아닌 필드)에 따라 정렬된다.
- 해당 필드에 올 수 있는 각 값의 종류별 (each distinct value)로 하나의 인덱스 엔트리를 포함하며, 이 엔트리에는 그 필드 값을 가진 레코드들이 저장된 첫번째 블록에 대한 주소를 포함한다.
- 클러스터링 인덱스는 삽입과 삭제가 상대적으로 간단한 비밀집(nondense) 인덱스의 한 예이다.
- EMPLOYEE 파일의 키가 아닌 필드 DEPTNUMBER에 대한 클러스터링 인덱스

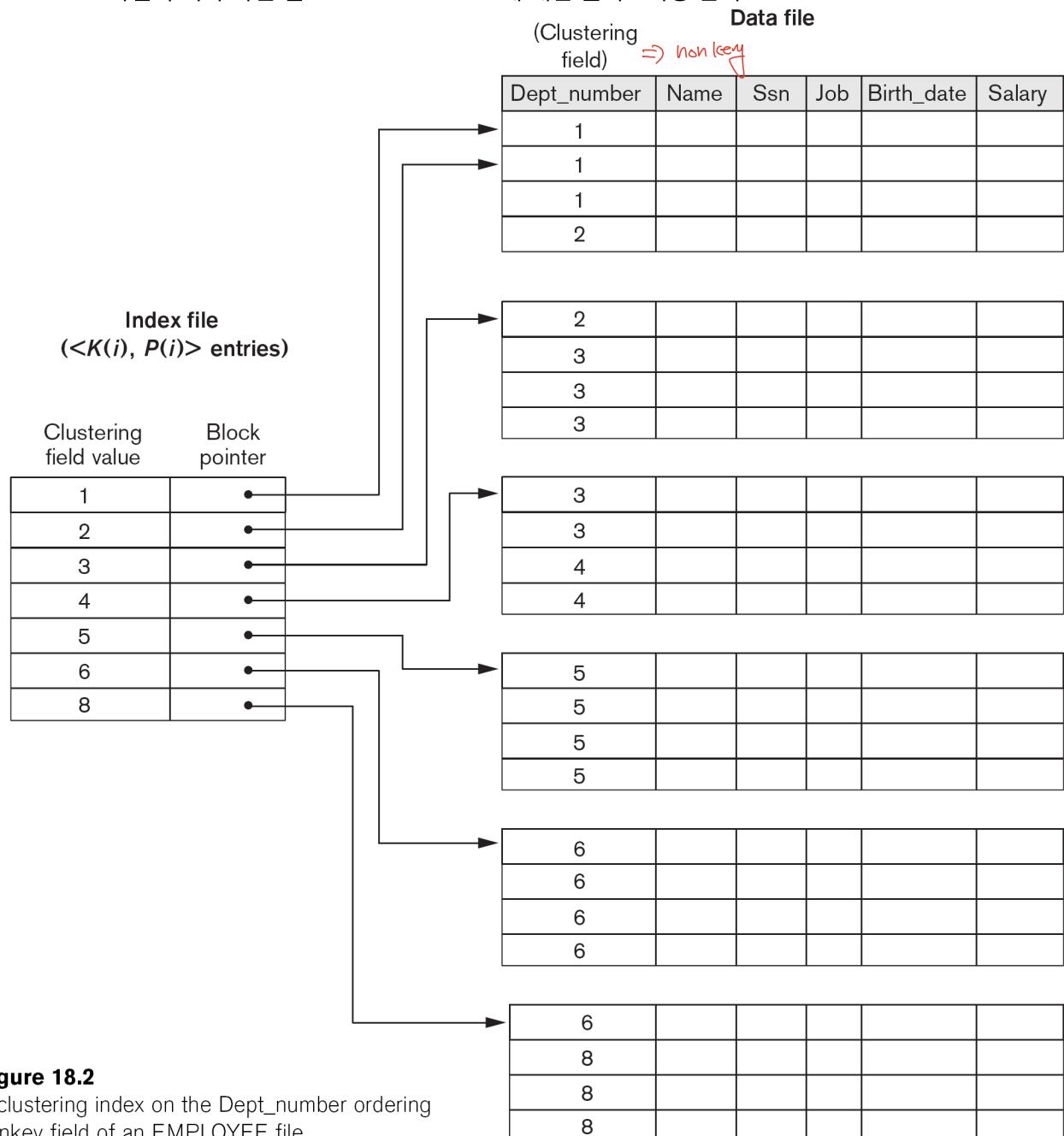


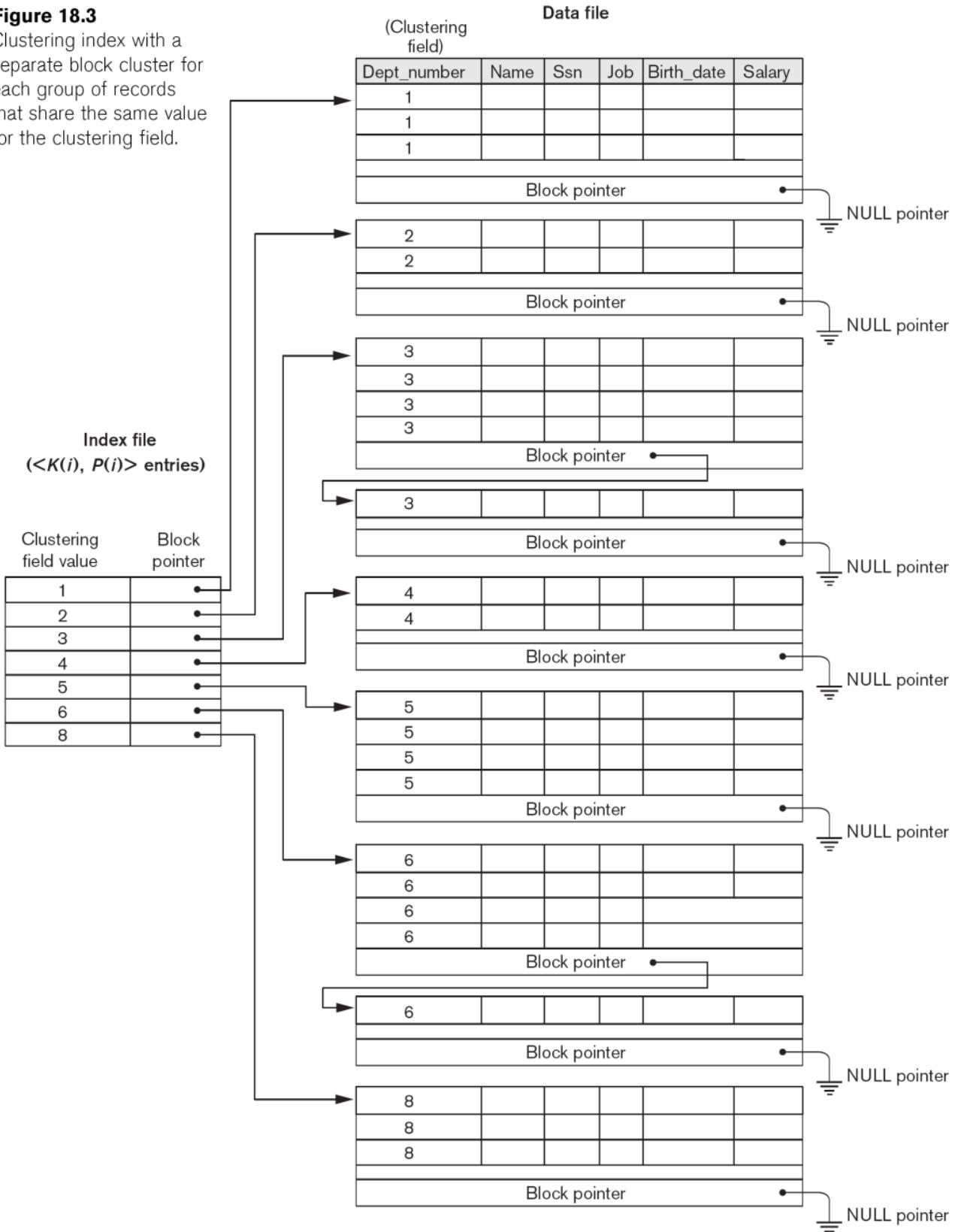
Figure 18.2

A clustering index on the `Dept_number` ordering nonkey field of an `EMPLOYEE` file.

- 같은 클러스터링 필드값을 갖는 레코드들의 각 그룹을 위해 별도의 블록 클러스터를 가지는 클러스터링 인덱스

**Figure 18.3**

Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

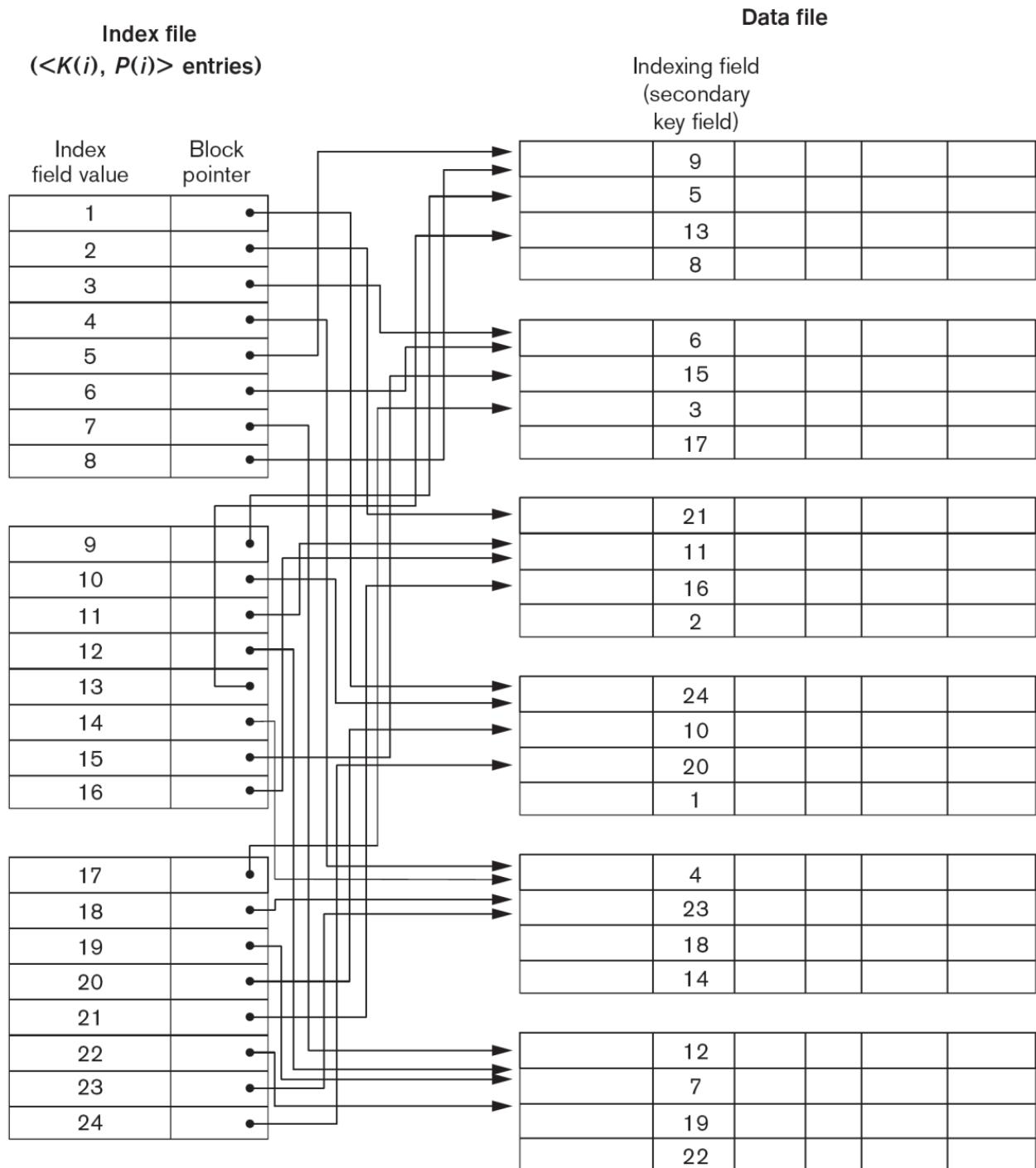


## 보조 인덱스(Secondary Index)

- 보조 인덱스는 기본 접근 방법이 이미 존재하는 파일을 접근하는 보조 수단을 제공한다.
- 보조 인덱스는 후보 키나 모든 레코드에 대해 유일한 값을 갖는 필드 또는 중복된 값을 갖는 키가 아닌 필드에 대해 만들 수 있다.
- 보조 인덱스는 두 개의 필드로 구성된 순서 파일이다.
  - 첫 번째 필드는 인덱스 필드인 데이터 파일의 비순서 필드와 같은 데이터 타입이다.
  - 두 번째 필드는 블록 포인터이거나 레코드 포인터이다.
- 같은 파일에 여러 개의 보조 인덱스가 존재할 수 있다.
- 엔트리에는 레코드에 대한 보조 키 값과 레코드가 저장되어 있는 블록 또는 레코드 자체에 대한 포인터가 있으므로, 키 필드에 대한 보조 인덱스는 밀집 인덱스이다.
- 파일의 비순서 키 필드에 대한 밀집 보조 인덱스(블록 포인터를 갖는 경우)

**Figure 18.4**

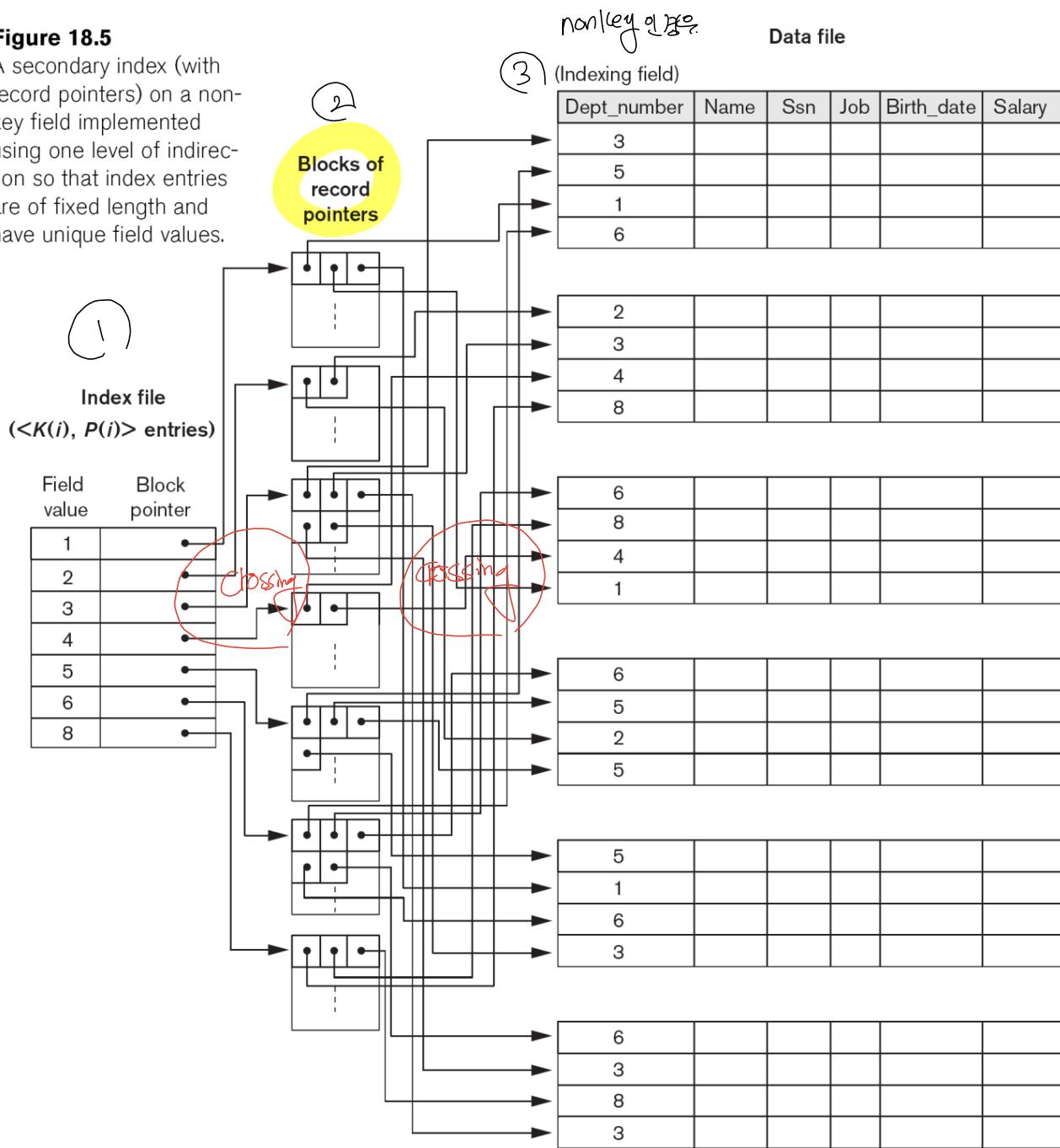
A dense secondary index (with block pointers) on a nonordering key field of a file.



- 인덱스 엔트리들이 고정 길이이고 유일한 필드값들을 갖도록 하나의 간접 단계를 이용하여 구현된, 키가 아닌 필드에 대한 보조 인덱스(레코드 포인터를 갖는 경우)

**Figure 18.5**

A secondary index (with record pointers) on a non-key field implemented using one level of indirection so that index entries are of fixed length and have unique field values.



## Summary

**Table 18.1** Types of Indexes Based on the Properties of the Indexing Field

	<b>Index Field Used for Physical Ordering of the File</b>	<b>Index Field Not Used for Physical Ordering of the File</b>
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

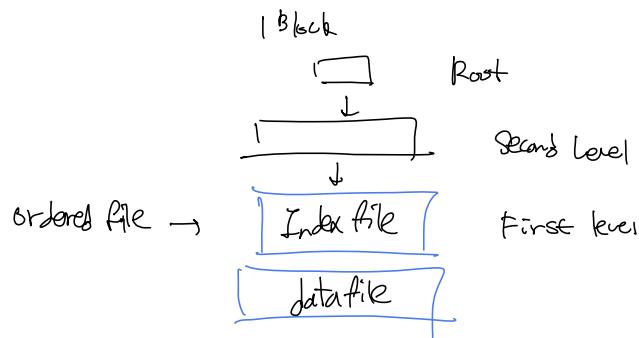
**Table 18.2** Properties of Index Types

Type of Index	Number of (First-level) Index Entries	Dense or Nondense (Sparse)	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no <sup>a</sup>
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records <sup>b</sup> or number of distinct index field values <sup>c</sup>	Dense or Nondense	No

## 다단계(Multi-Level) 인덱스

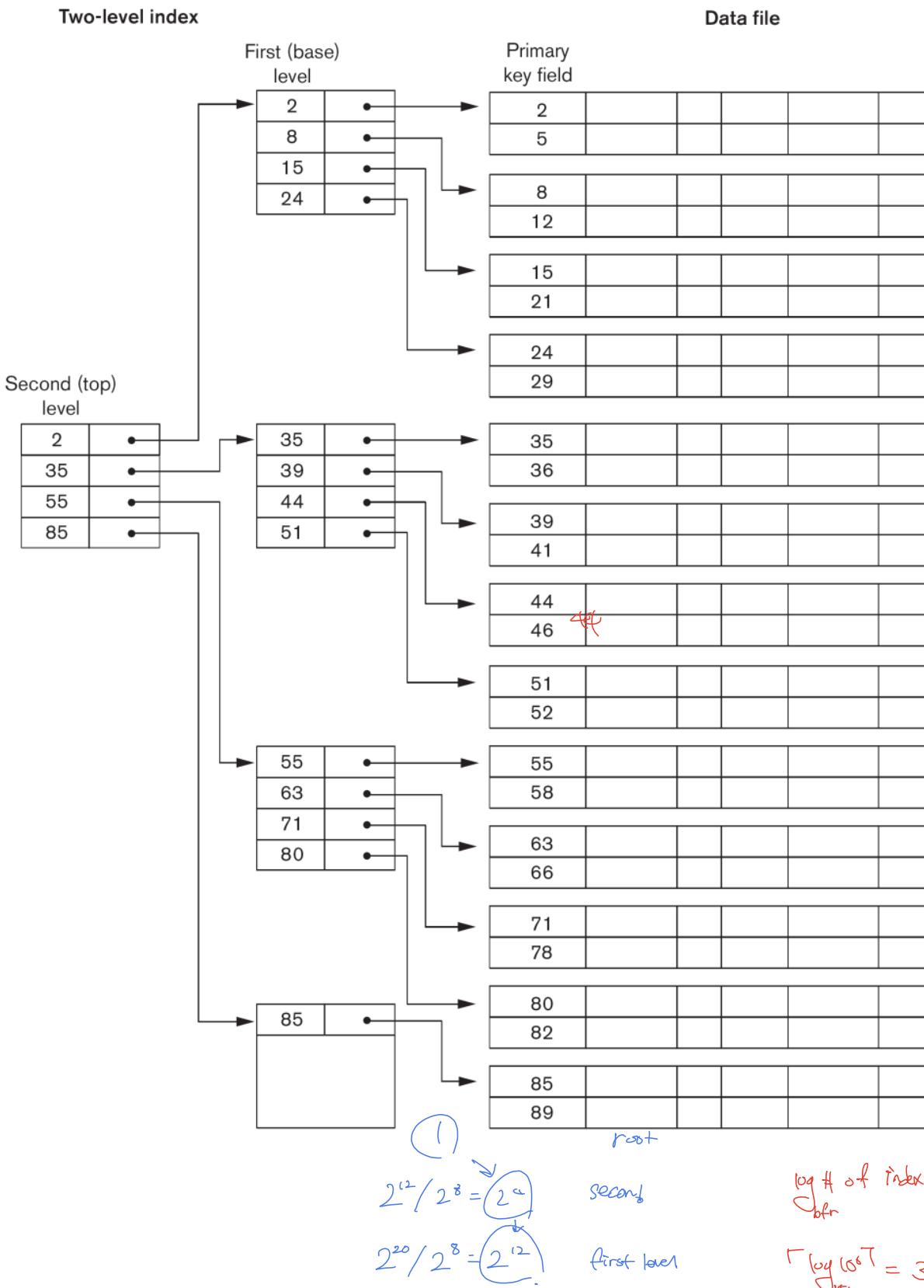
- 단일 단계 인덱스가 순서 화일이므로, 이 인덱스 자체에 대한 기본 인덱스를 만들 수 있다; 이 경우, 원래 인덱스 화일은 첫번째 단계 인덱스라 부르고 그 인덱스에 대한 인덱스는 두번째 단계 인덱스라 부른다.
- 위와 같은 과정을 반복하면 모든 엔트리를 한 블록에 저장할 수 있는 단계가 생기고, 이 단계의 블록을 최상위 단계라고 한다.
- 다단계 인덱스는 첫번째 단계 인덱스가 어떤 인덱스 유형(기본 인덱스, 클러스터링 인덱스, 보조 인덱스)이든지 사용할 수 있다.

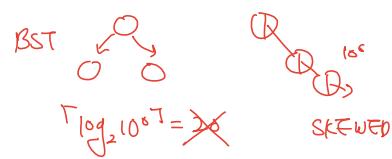
- Example: ISAM(Indexed Sequential Access Method) 조직과 유사한 2-단계 기본 인덱스



**Figure 18.6**

A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.





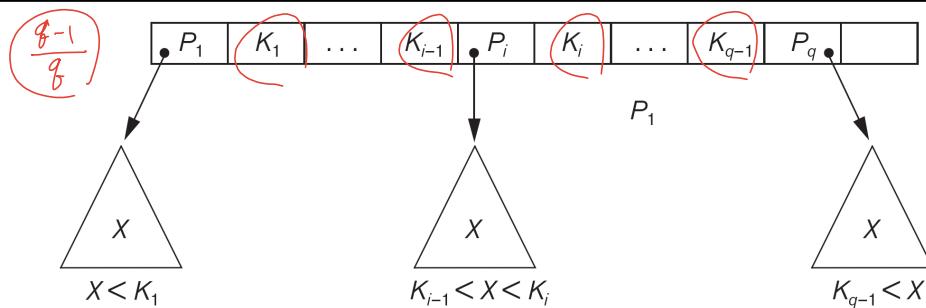
## Search Tree

- 다단계 인덱스는 탐색 트리 (Search Tree) 형태를 갖는다. 그러나 인덱스의 모든 단계가 순서 화일이므로, 새로운 인덱스 엔트리에 대한 삽입과 삭제가 매우 복잡하다는 문제점이 있다.

f-ary Search Tree

- 서브트리에 대한 포인터를 갖는 탐색 트리의 한 노드

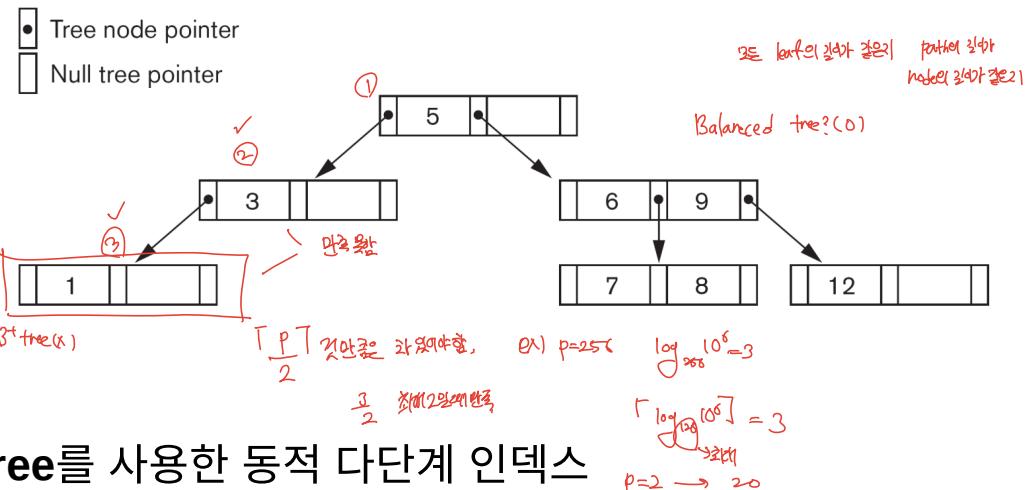
**Figure 18.8**  
A node in a search tree with pointers to subtrees below it.



$$\log_9 \rightarrow \text{트리 깊이 } \log_2(10^6)$$

- 차수가  $p = 3$ 인 탐색 트리

**Figure 18.9**  
A search tree of order  $p = 3$ .



## B-Tree와 B<sup>+</sup>-Tree를 사용한 동적 다단계 인덱스

- 삽입과 삭제 때문에, 대부분의 다단계 인덱스는 B-Tree나 B<sup>+</sup>-Tree 자료 구조를 사용한다. 이들 구조는 각 트리 노드 (즉 디스크 블록)에 새로운 인덱스 엔트리를 저장할 약간의 여유 공간을 남겨둔다.
- 이들 자료 구조들은 새로운 탐색 키의 삽입과 삭제를 효율적으로 처리할 수 있는 탐색 트리의 변형이다.
- B-Tree와 B<sup>+</sup>-Tree 자료 구조에서 각 노드는 하나의 디스크 블록을 할당하여 저장한다.
- 각 노드가 최소한 절반 이상 차 있도록 보장하여 저장 효율을 높일 수 있는 방식이다.
- 완전히 차 있지 않는 노드에 삽입하는 것은 간단히 처리될 수 있다; 만약 노드가 차 있으면 삽입을 위해 두 노드로 분할한다.
- 노드 분할은 트리의 다른 단계로 파급될 수 있다.
- 삭제시 절반이상 차 있게 되는 노드에 대한 삭제는 간단히 처리될 수 있다.
- 삭제시 노드가 절반이하로 차게되면 이 노드를 이웃 노드들과 합병해야 한다.

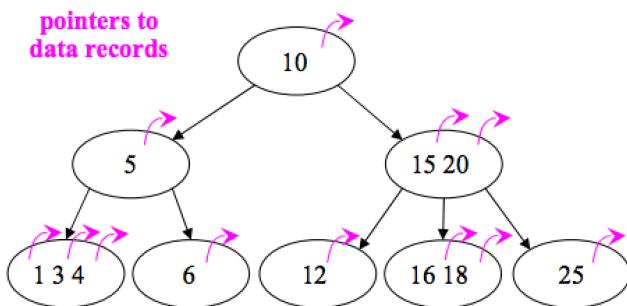
내려온다  
 100% 차지  
 50% 차지

## B-Tree와 B<sup>+</sup>-Tree의 차이점

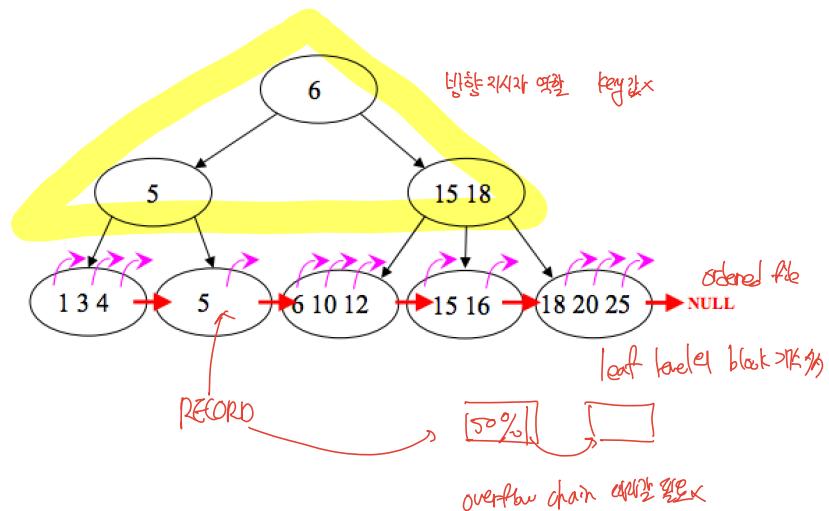
- B-Tree에서는 모든 단계의 노드들이 데이터 레코드들에 대한 포인터들을 갖는다.
- B<sup>+</sup>-Tree에서는 리프 단계의 노드들만 데이터 레코드들에 대한 포인터들을 갖는다.
- B<sup>+</sup>-Tree는 대응하는 B-Tree보다 더 적은 단계를 (더 많은 탐색 값을) 갖는다.
- 대부분의 DBMS는 B<sup>+</sup>-Tree를 사용한다. DBMS에서 B-Tree라고하면 실제로는 B<sup>+</sup>-Tree를 언급하고 있다고 보면된다. (B-Tree는 주로 정보검색과 같이 읽기가 주 작업인 경우에 사용된다.)

- B-tree vs B<sup>+</sup>-tree

B-tree of order 4

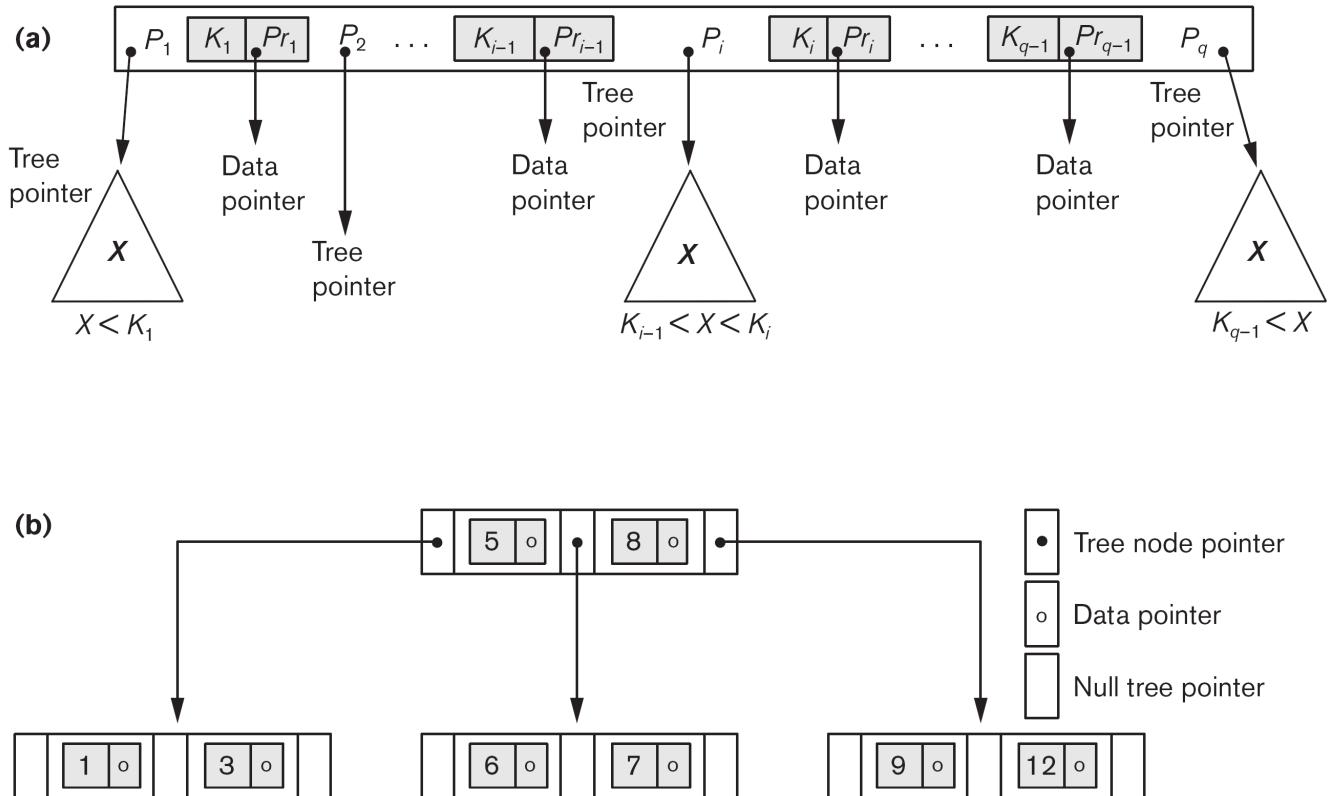


B<sup>+</sup>-tree of order 4



## B-Tree 구조

- (a)  $q - 1$ 개의 탐색값을 갖는 B-Tree의 한 노드
- (b) 차수  $p = 3$ 인 B-Tree(삽입 순서는 8, 5, 1, 7, 3, 12, 9, 6이다.)

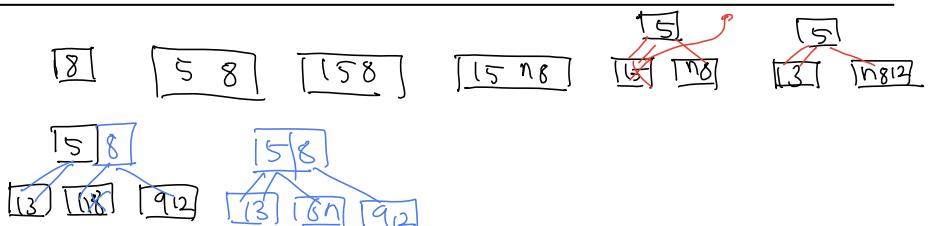


**Figure 18.10**

B-tree structures. (a) A node in a B-tree with  $q - 1$  search values. (b) A B-tree of order  $p = 3$ . The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

Btree는 5개의 탐색값  
B<sup>+</sup>tree는 5개의 인덱스

## B<sup>+</sup>-Tree



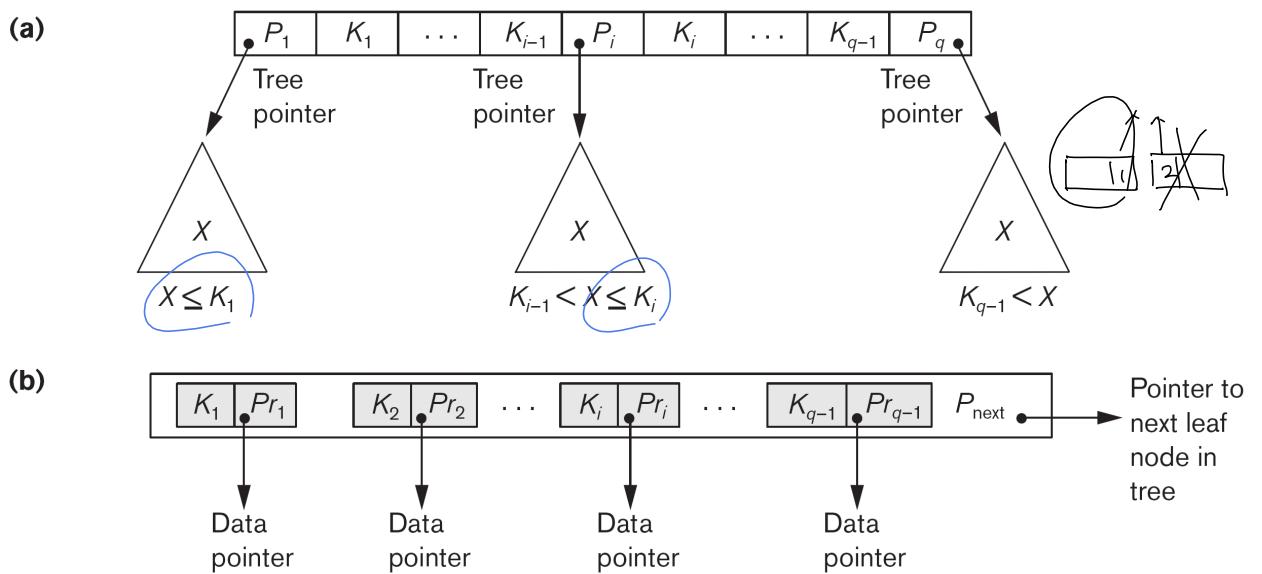
## B<sup>+</sup>-Tree of Order $p$

- (a)  $q - 1$ 개의 탐색값을 갖는 내부 노드
- (b)  $q - 1$ 의 탐색값과  $q - 1$ 의 데이터 포인터를 가지는 B<sup>+</sup>-Tree의 리프 노드

**Figure 18.11**

The nodes of a B<sup>+</sup>-tree. (a) Internal node of a B<sup>+</sup>-tree with  $q - 1$  search values.

(b) Leaf node of a B<sup>+</sup>-tree with  $q - 1$  search values and  $q - 1$  data pointers.



## Internal node structure

1. Each internal node is of the form  $\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$  where  $q \leq p$  and each  $P_i$  is a tree pointer.
2. Within each internal node,  $K_1 < K_2 < \dots < K_{q-1}$ .
3. For all search field values  $X$  in the subtree pointed at by  $P_i$ , we have  $K_{i-1} < X \leq K_i$  for  $1 < i < q$ ;  $X \leq K_i$  for  $i = 1$ ; and  $K_{i-1} < X$  for  $i = q$  (see Figure 18.11(a)). 10  
 $P_{\text{leaf}}$
4. Each internal node has at most  $p$  tree pointers.
5. Each internal node, except the root, has at least  $\lceil p/2 \rceil$  tree pointers. The root node has at least two tree pointers if it is an internal node.
6. An internal node with  $q$  pointers,  $q \leq p$ , has  $q - 1$  search field values.

## Leaf node structure

1. Each leaf node is of the form  $\langle \langle K_1, Pr_1 \rangle, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_{\text{next}} \rangle$  where  $q \leq p$ , each  $Pr_i$  is a data pointer, and  $P_{\text{next}}$  points to the next leaf node of the  $B^+$ -tree.
2. Within each leaf node,  $K_1 \leq K_2 \dots, K_{q-1}, q \leq p$ .
3. Each  $Pr_i$  is a data pointer that points to the record whose search field value is  $K_i$  or to a file block containing the record (or to a block of record pointers that point to records whose search field value is  $K_i$  if the search field is not a key).
4. Each leaf node has at least  $\lceil p/2 \rceil$  values.
5. All leaf nodes are at the same level.

## Searching in $B^+$ -Tree

**Algorithm 18.2.** Searching for a Record with Search Key Field Value  $K$ , Using a  $B^+$ -tree

```
n ← block containing root node of  $B^+$ -tree;  
read block n;  
while (n is not a leaf node of the  $B^+$ -tree) do  
    begin  
        q ← number of tree pointers in node n;  
        if  $K \leq n.K_1$  (* $n.K_i$  refers to the  $i$ th search field value in node n*)  
            then  $n \leftarrow n.P_1$  (* $n.P_i$  refers to the  $i$ th tree pointer in node n*)  
        else if  $K > n.K_{q-1}$   
            then  $n \leftarrow n.P_q$   
        else begin  
            search node n for an entry  $i$  such that  $n.K_{i-1} < K \leq n.K_i$   
             $n \leftarrow n.P_i$   
        end;  
    read block n  
end;  
search block n for entry  $(K_i, Pr_i)$  with  $K = K_i$ ; (* search leaf node *)  
if found  
    then read data file block with address  $Pr_i$  and retrieve record  
    else the record with search field value  $K$  is not in the data file;
```

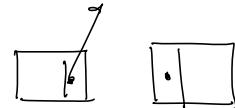
# Insertion in B<sup>+</sup>-Tree

- Full algorithm은 교재 참고

## Insertion

Perform a search to determine what bucket the new record should go into.

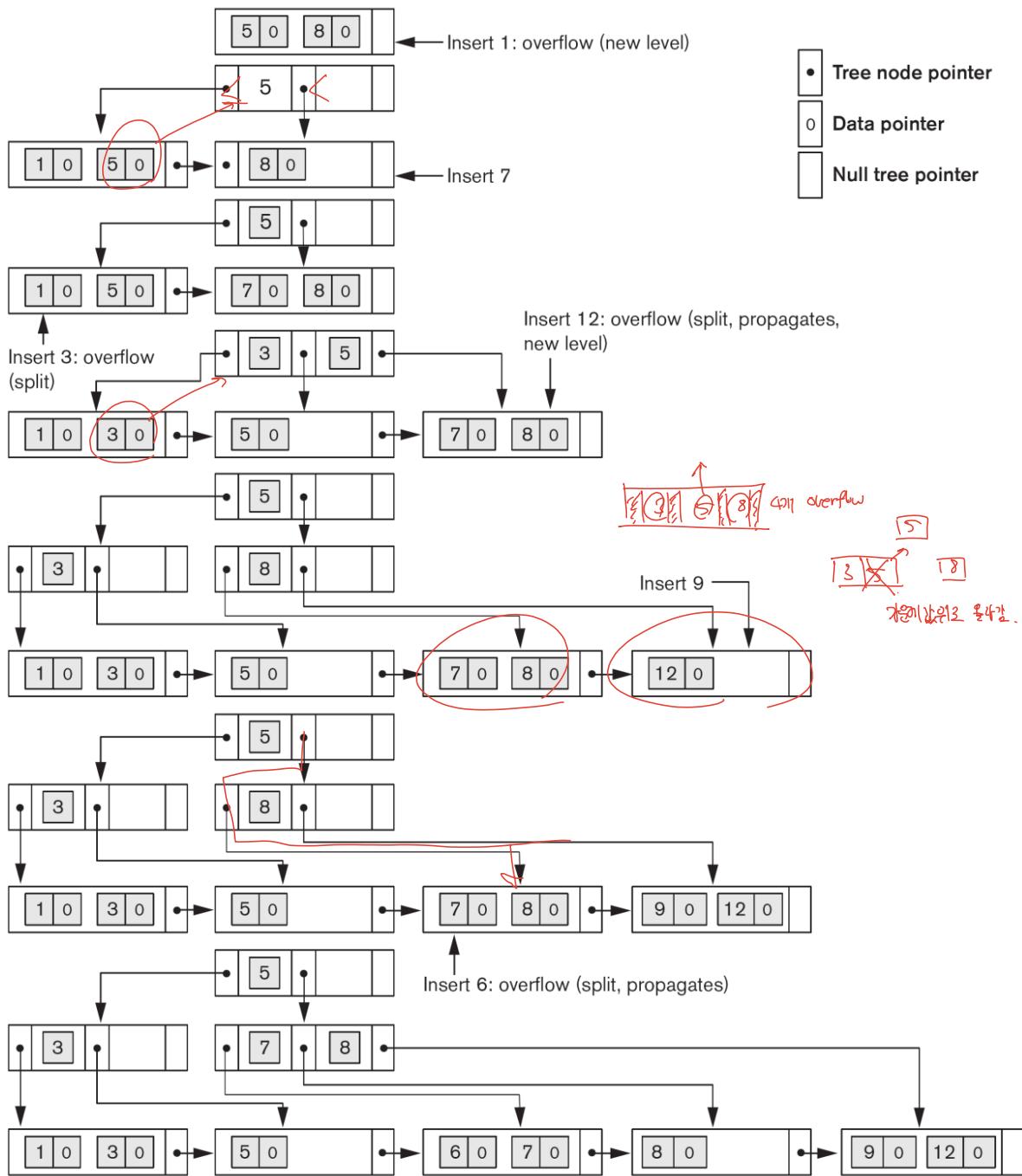
- If the bucket is not full (at most  $b - 1$  entries after the insertion), add the record.
- Otherwise, split the bucket.
  - Allocate new leaf and move half the bucket's elements to the new bucket.
  - Insert the ~~new leaf's smallest key~~ and address into the parent.
  - If the parent is full, split it too.
    - Add the middle key to the parent node.
    - Repeat until a parent is found that need not split.
- If the root splits, create a new root which has one key and two pointers.



- $p = 3$ 이고  $p_{leaf} = 2$ 인 B<sup>+</sup>-Tree에 대한 삽입의 예

$$\begin{array}{ll} p=3 & p_{leaf}=2 \\ 2,3 & 1,2 \end{array}$$

Insertion sequence: 8, 5, 1, 7, 3, 12, 9, 6



**Figure 18.12**

An example of insertion in a B<sup>+</sup>-tree with  $p = 3$  and  $p_{\text{leaf}} = 2$ .

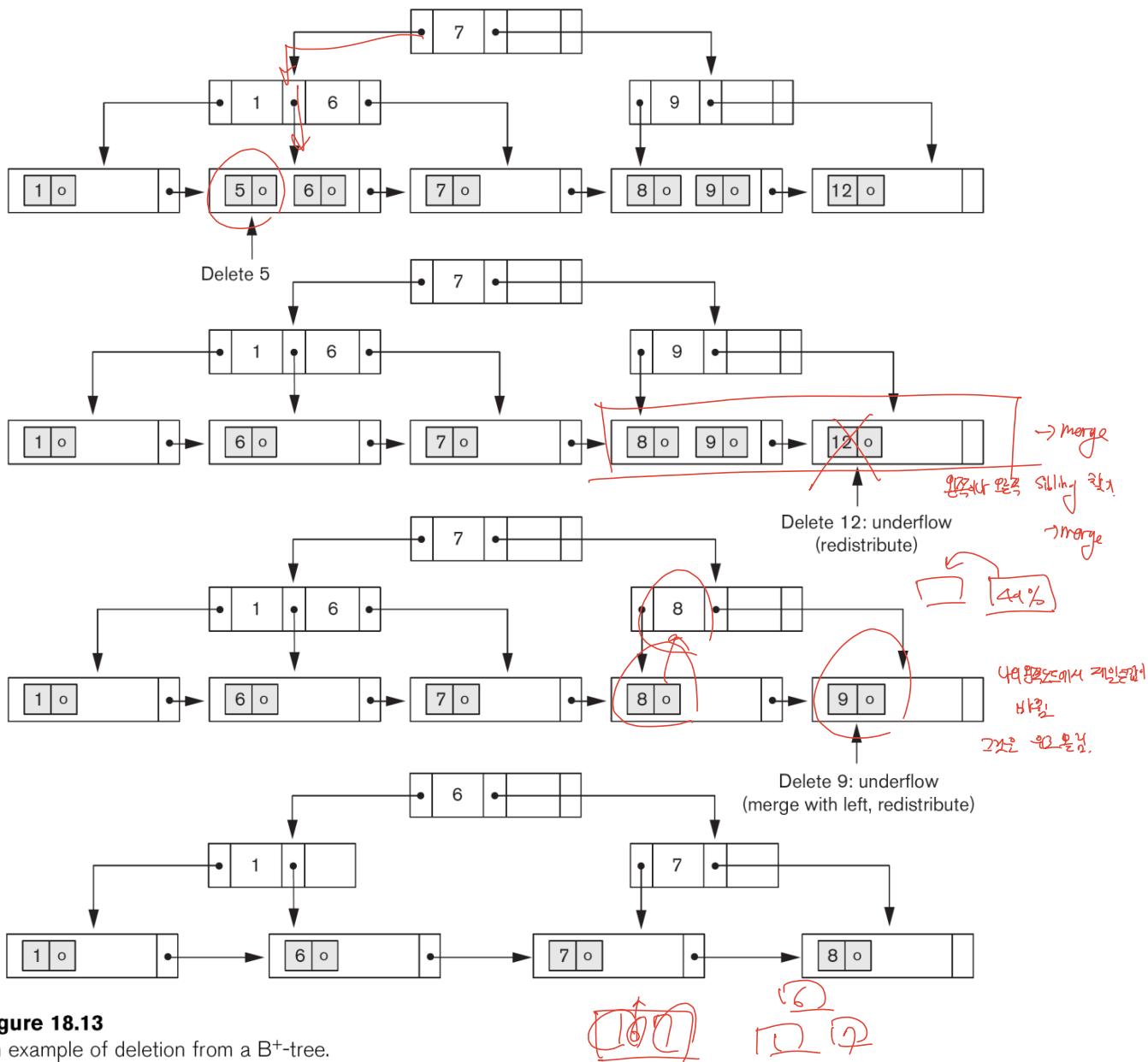
## **Deletion in B<sup>+</sup>-Tree**

- Full algorithm은 교재 참고

### **Deletion**

- Start at root, find leaf L where entry belongs.
  - Remove the entry.
    - If L is at least half-full, done!
    - If L has fewer entries than it should,
      - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
      - If re-distribution fails, merge L and sibling.
  - If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
  - Merge could propagate to root, decreasing height.
- 
- B<sup>+</sup>-Tree에 대한 삭제의 예

Deletion sequence: 5, 12, 9



**Figure 18.13**

An example of deletion from a B<sup>+</sup>-tree.

## Examples

## The order $p$ and $p_{leaf}$ of $B^+$ -tree

Assume

- the search key field is  $V = 9$  bytes long,
- the block size is  $B = 512$  bytes,
- a record pointer is  $P_r = 7$  bytes,
- and a block pointer is  $P = 6$  bytes.

Calculate the order  $p$  and  $p_{leaf}$  of  $B^+$ -tree

order  $p$

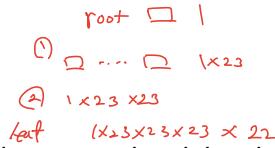
$$512 \geq p \times 6 + (p-1) \times 9$$

$$p = \left\lfloor \frac{512+9}{15} \right\rfloor = 34 \times 0.69 \\ = 23$$

$$p_{leaf} = \frac{512}{16} = 32 \geq (9+7)p_{leaf} \\ \left\lfloor \frac{512}{16} \right\rfloor = p_{leaf} = 32 \times 0.69 = 22$$

## Levels of $B^+$ -tree

Assume that each node is 69 percent full. Calculate the average number of entries at root level, level 1, level 2, leaf level:



## How many disk I/O for B+-tree indexing

$\log_f N$

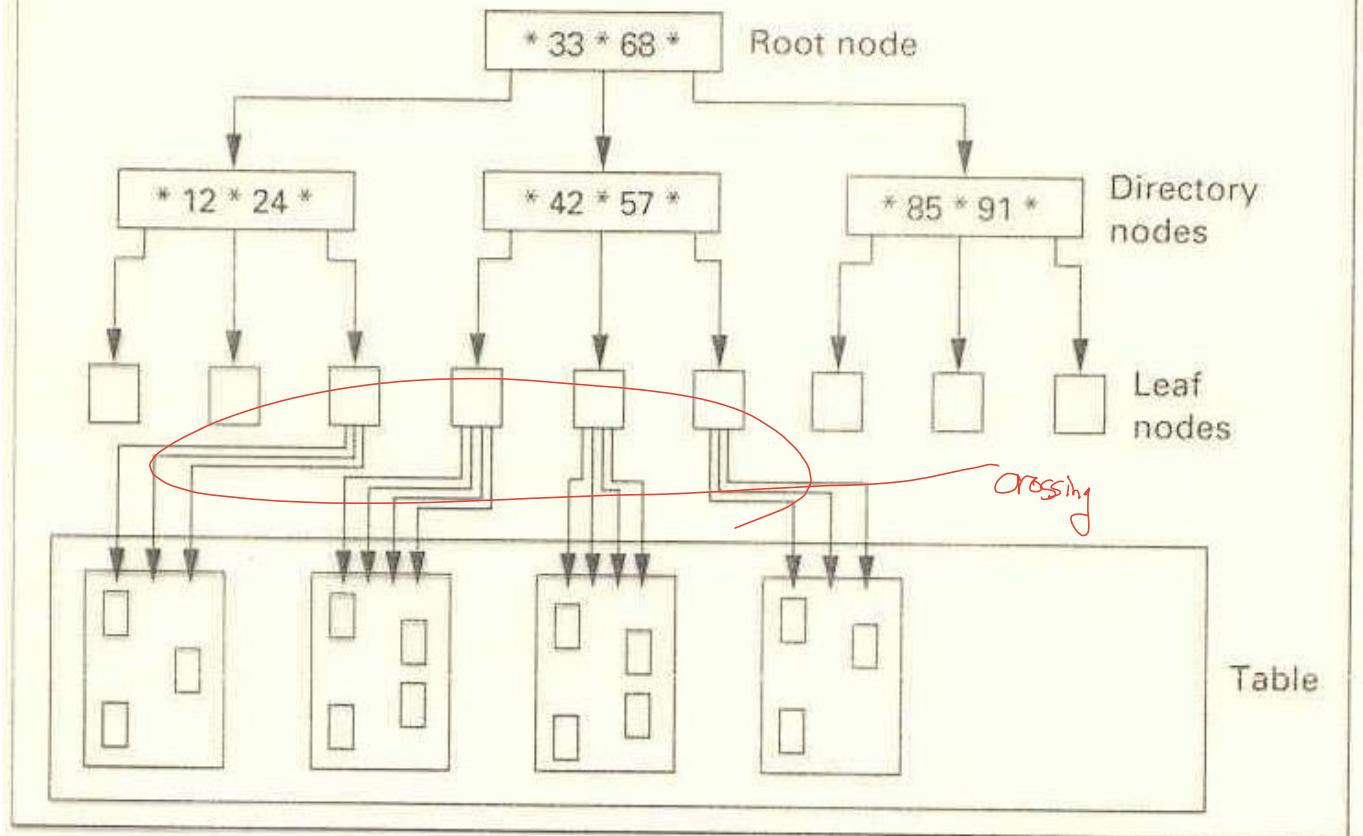
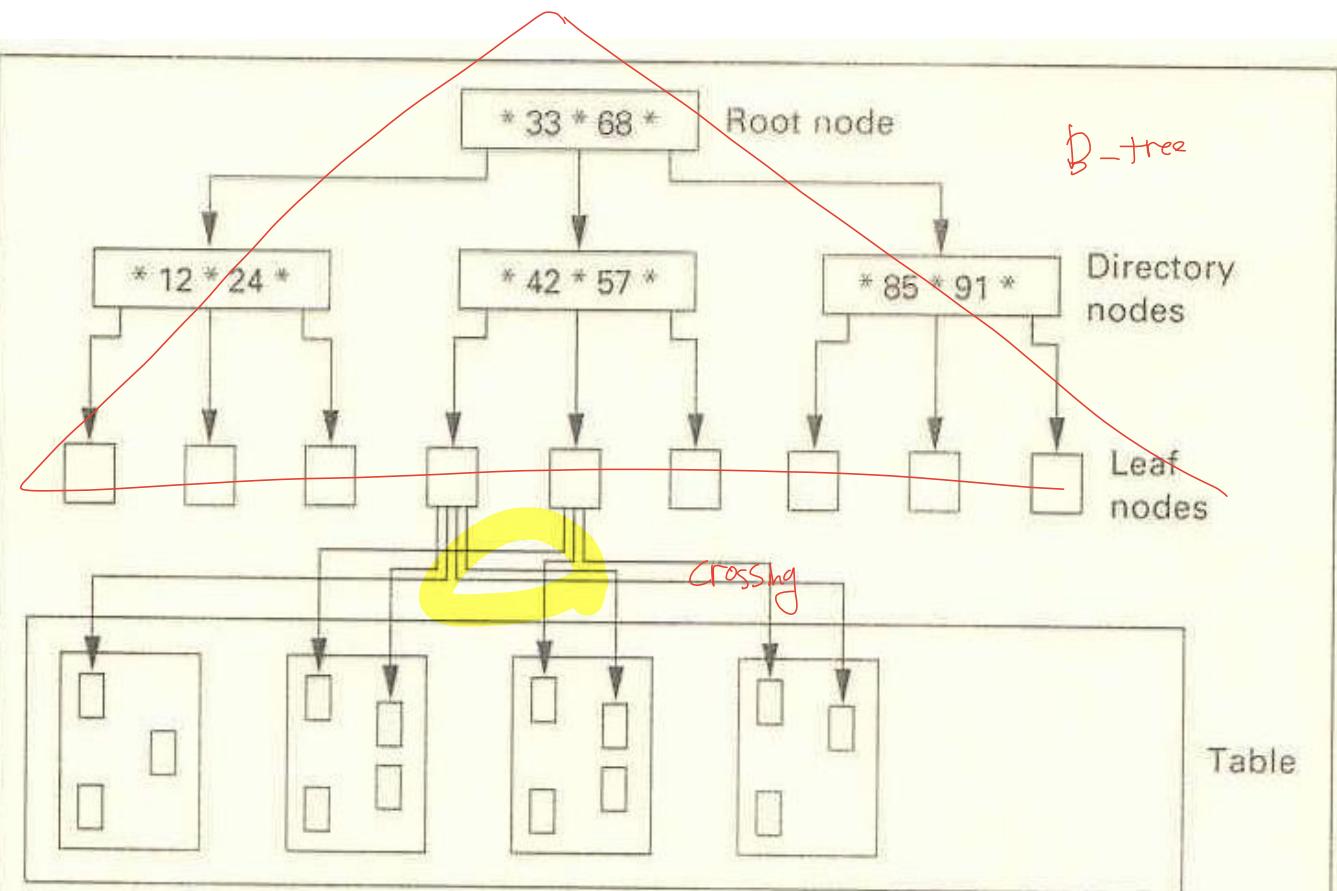
- In general,  $\lceil \log_f N \rceil$  probes are required.  $f$ : fanout,  $N$ : # of rows
- Disk page buffering: index pages may be buffered in main memory. If top 2 levels are buffered, only 1 I/O to find a ROWID is needed.
- Assume that # of rows =  $10^6$ , ROWID = 4 bytes, keyvalue = 4 bytes, diskpage = 2 KB
  - How many I/Os to find a ROWID?
  - How many I/Os to find a ROWID if binary search is used?

## Clustered vs Nonclustered

- Assume 10 million customers
- customers : row size 100 bytes on 2KB pages
- Attributes:
  - straddr
  - cityst
  - zipcode
  - age
  - incomeclass : 1 – 10 등급
  - hobby : 50 distinct values
  - major1dept : 고객이 가장 많이 시간을 보낸 곳
  - major2dept : 고객이 두 번째로 많이 시간을 보낸 곳
- mailing labels을 만들려고 함
- SQL Query:

```
select name, straddr, cityst, zipcode from customers where
city = 'Boston' and age between 18 and 50 and hobby in ('...');
```

- Clustering effect. Clustering의 효과는?
  - Assume clustered index on zipcode (Note that zipcode → city) :
  - Boston 지역은 전체 고객의 1/50
  - 200,000 rows
  - 10,000 disk pages
  - 10,000 disk I/O
  - 10,000/80 (초당 80 I/O) = 125 sec
- Assume non-clustered index :
  - where 절의 다른 조건을 만족하는 row가 전체의 1/5
  - 40,000 rows가 500,000 disk pages에 흩어져 있다.
  - 40,000 disk I/O / 80 (초당 80 I/O) = 500 sec



# Reorganizing Clustered pages

- Eventually, the index can lose much of its clustering property  $\Rightarrow \text{REORGANIZE data pages}$
- 언제 재조직이 가능한가?
- 24-hours up-time system인 경우는?

B<sup>+</sup> tree      Primary      Indexing

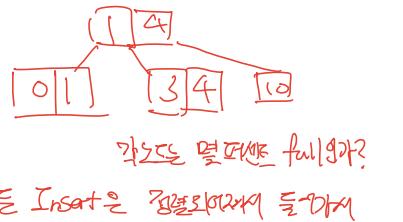
상당히 빠르게 데이터 검색하기

## Bulk Loading and Bottom-Up Build

- Inserting entries one-at-a-time into a B+-tree requires  $\geq 1$  IO per entry
  - assuming leaf level does not fit in memory
  - can be very inefficient for loading a large number of entries at a time (bulk loading)
- Efficient alternative 1:**
  - sort entries first (using efficient external-memory sort algorithms discussed later in Section 12.4)
  - insert in sorted order
    - insertion will go to existing page (or cause a split)
    - much improved IO performance, but most leaf nodes half full
- Efficient alternative 2: Bottom-up B+-tree construction**
  - As before sort entries
  - And then create tree layer-by-layer, starting with leaf level
  - Implemented as part of bulk-load utility by most database systems

한번 만들었는데

여기로 가시(들)기



각 노드는 몇 페스토 full인가?

모든 Insert는 정렬된 데이터를 가짐

