Databases

화일의 인덱스 구조

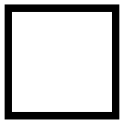
한국공학대학교
게임공학과
장 지 웅

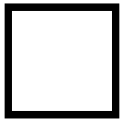
Contents

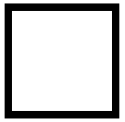
화일(file)

화일

이재영 2012096138 031-8041-0551 AI
장지웅 2013081194 031-8041-0554 DB
윤정현 2013083029 031-8041-0552 C++







필

레코드의 특징

이재영	2012096138	031-8041-0551	AI
장지웅	2013081194	031-8041-0554	DB
윤정현	2013083029	031-8041-0552	C++
		□	
		□	
		□	

1. 필드의 개수는 동일
2. 각 필드의 크기는 고정
3. 모든 레코드의 크기는 동일

기억을 되살려 봅시다.

자료구조에서 성능을 결정하는 가장 중요한 요소는 무엇이었나?

- 시간복잡도 측면에서?
- 공간복잡도 측면에서?

생각해 봅시다.

배열과 같은 구조로 데이터 화일을 만들 수 있을까?
문제점은 무엇인가?

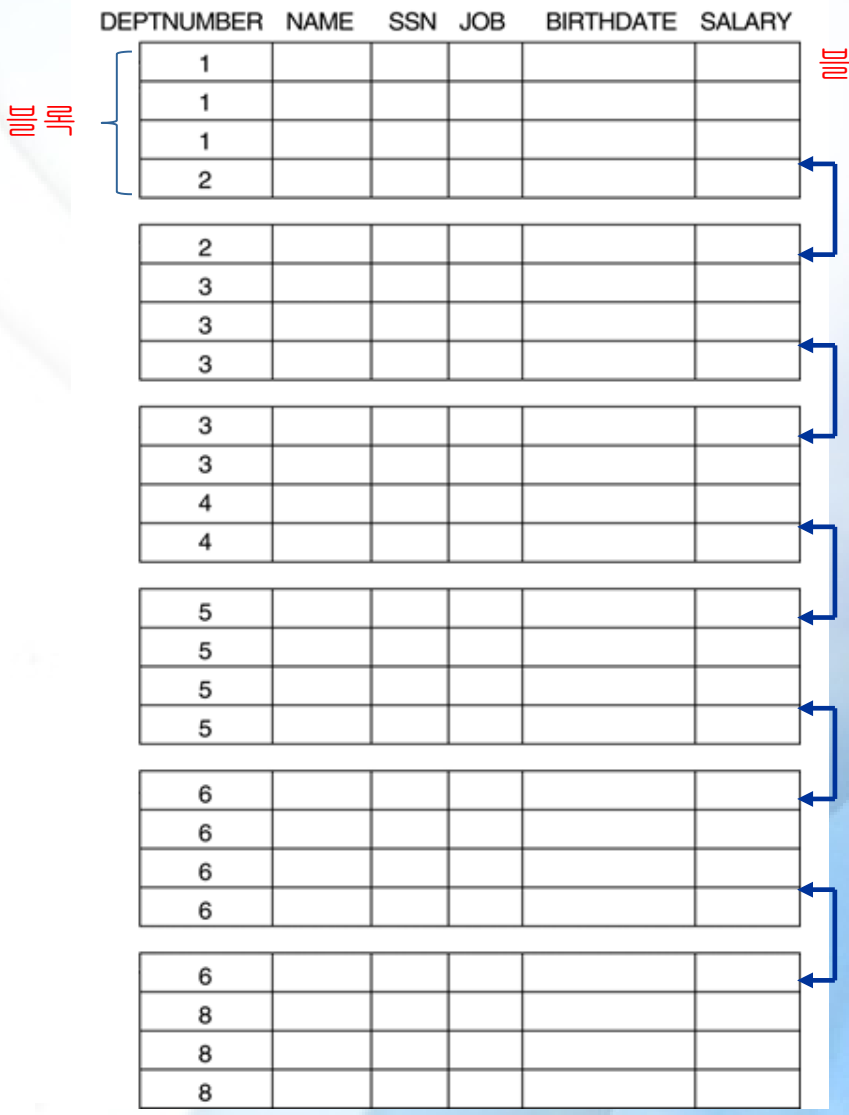
생각해 봅시다.

연결리스트 구조로 데이터 화일을 만들 수 있을까?
문제점은 무엇인가?

생각해 봅시다.

배열과 연결리스트의 단점을
보완할 수 있는 새로운 자료구조를
생각해보자

데이터 화일의 구조



문제가 해결되었는가?
- 레코드의 추가 삽입
- 하드디스크 액세스 횟수

블록의 크기는 어느 정도가 적당한가?

생각해 봅시다.

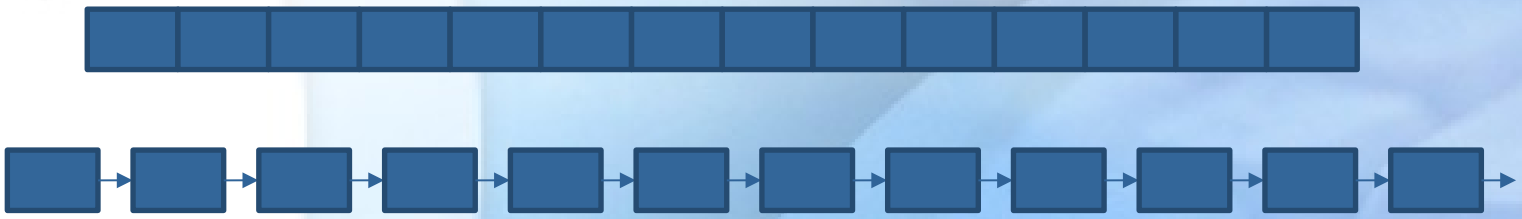
데이타 화일에서 원하는 레코드를 찾는 방법은 무엇인가?
이 때, 성능에 영향을 미치는 요인은 무엇인가?

기억나시나요?

순차검색 VS 이진검색

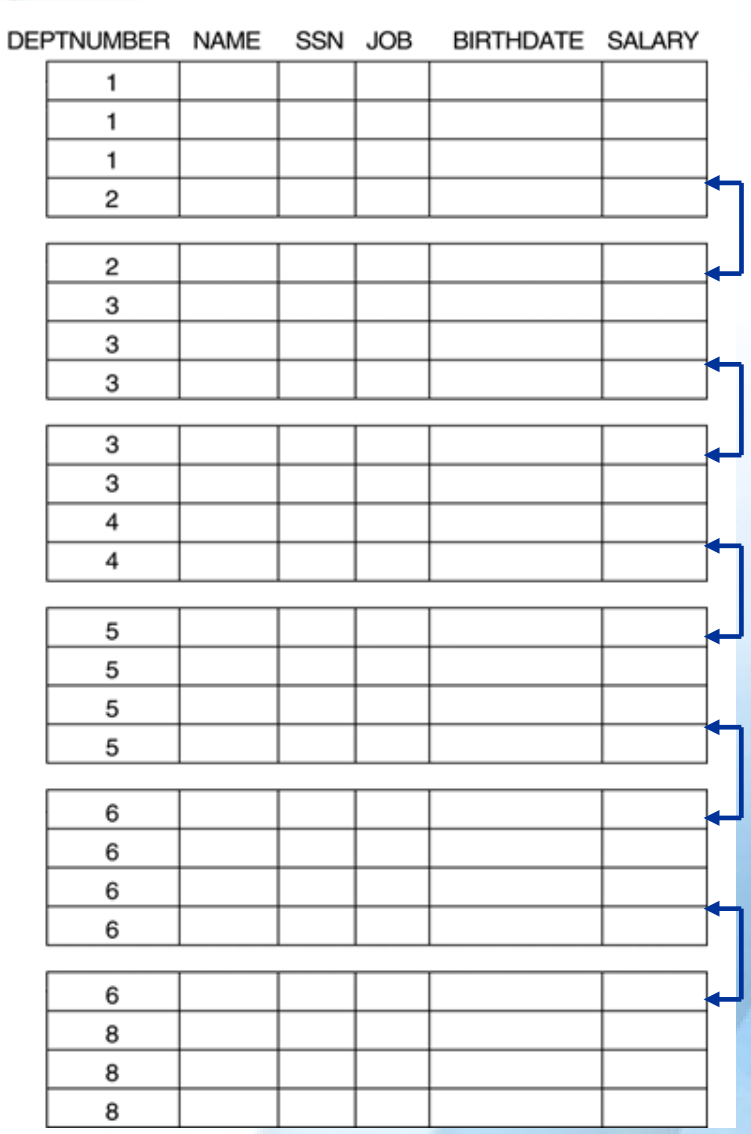
시간복잡도?

이진검색의 조건은?



순서화일에서의 검색

순서화일이란?
특정 필드 값의 크기 순으로
레코드를 저장한 화일



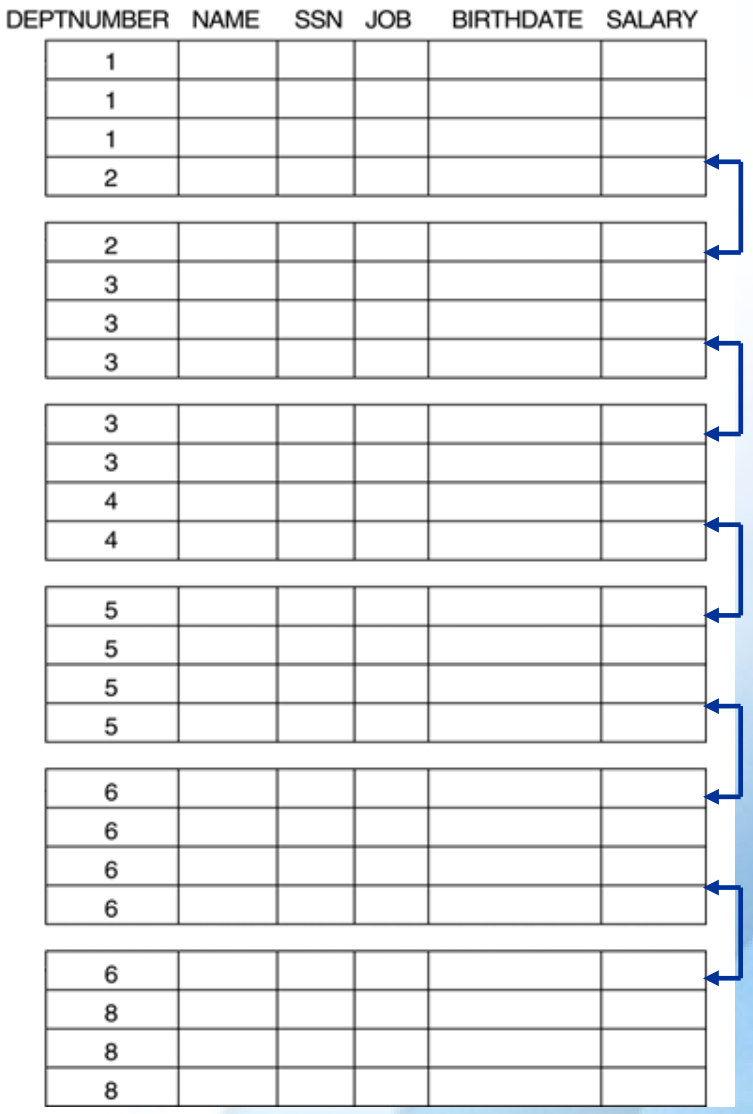
순차검색?

이진검색?

생각해 봅시다.

데이타 화일에 새로운 레코드를 삽입하는 방법은 무엇인가?
이 때, 성능에 영향을 미치는 요인은 무엇인가?

순서화일에서의 삽입



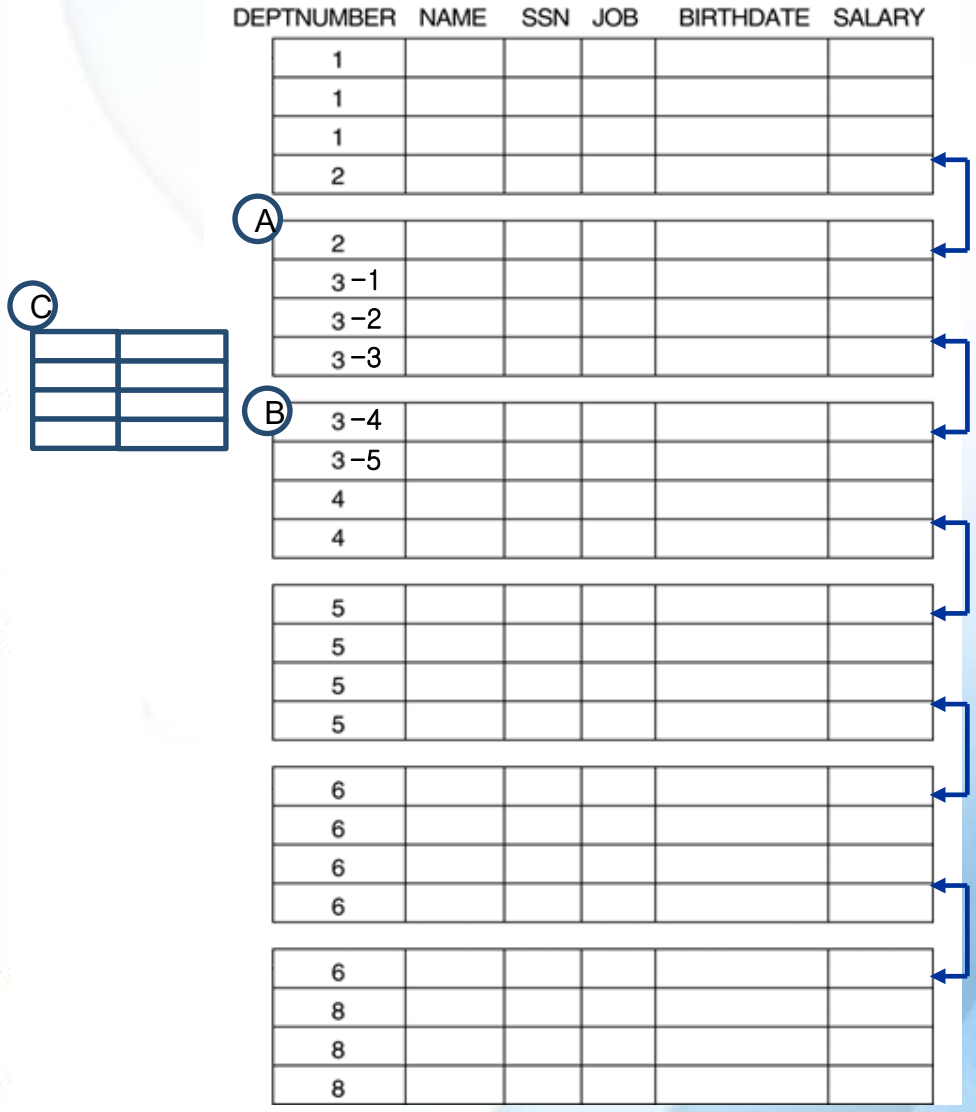
1. 삽입할 자리를 찾는다.
2. 삽입할 공간을 만든다.
3. 삽입한다.

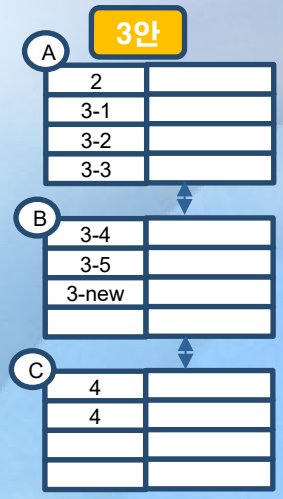
삽입할 공간이 없으면
어떻게 만드는가?

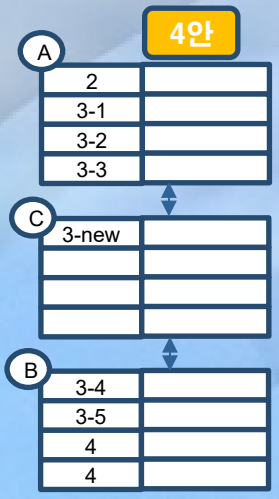
순서화일에서의 삽입

새로운 블록에 넣을 레코드는?

3-new

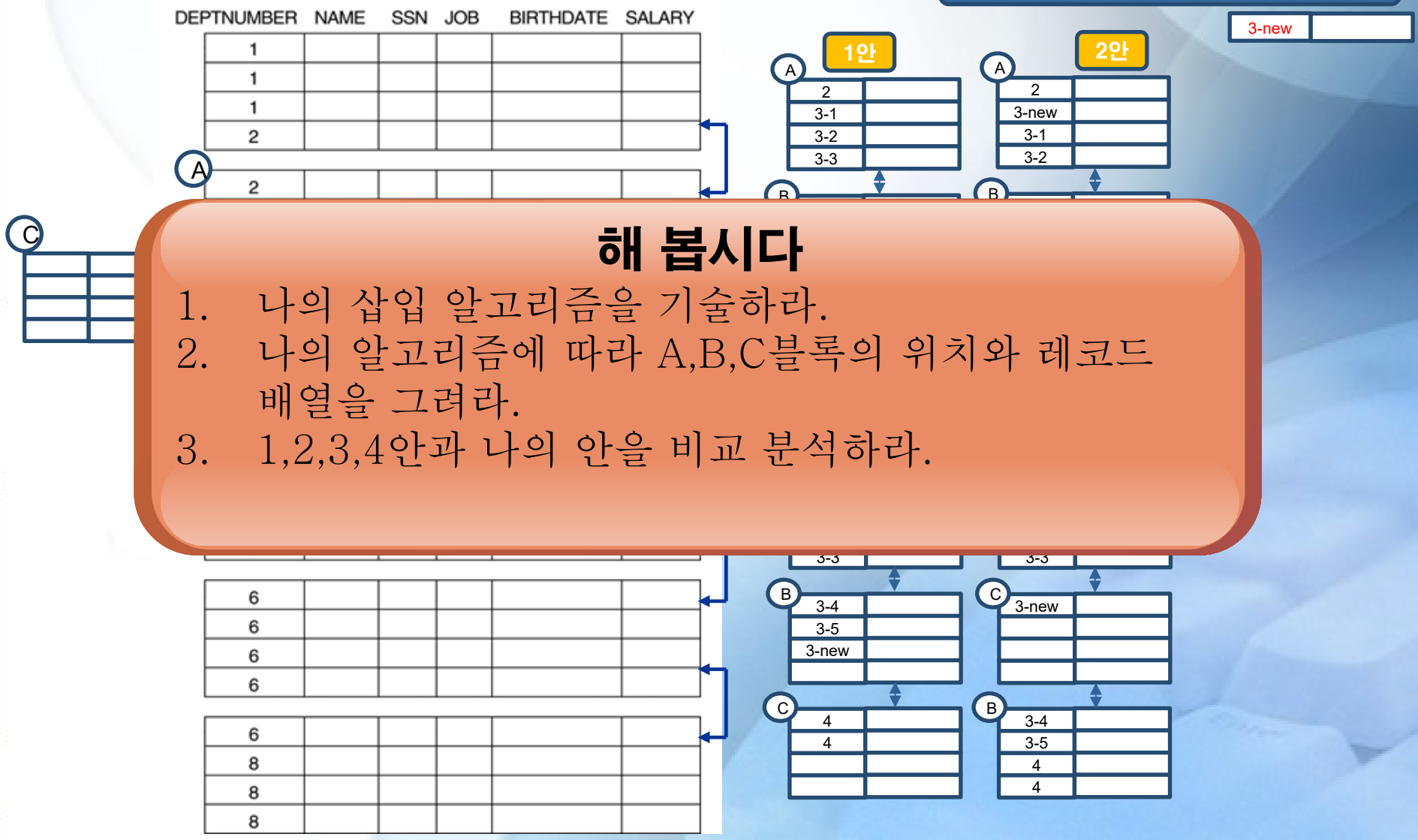






순서화일에서의 삽입

새로운 블록에 넣을 레코드는?

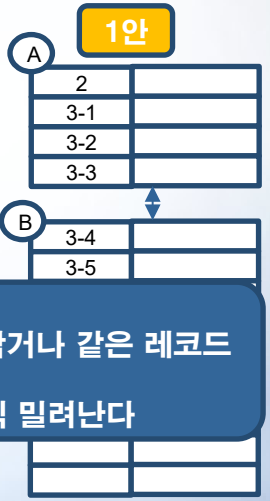


해 봅시다

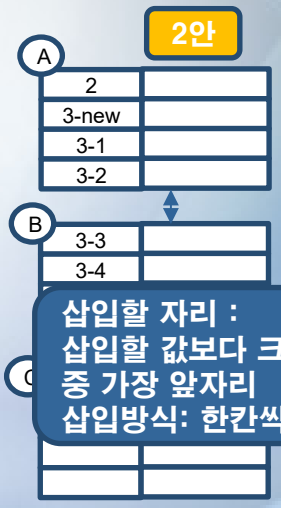
1. 나의 삽입 알고리즘을 기술하라.
2. 나의 알고리즘에 따라 A,B,C블록의 위치와 레코드 배열을 그려라.
3. 1,2,3,4안과 나의 안을 비교 분석하라.

순서화일에서의 삽입

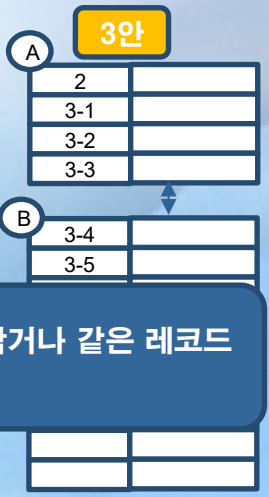
새로운 블록에 넣을 레코드



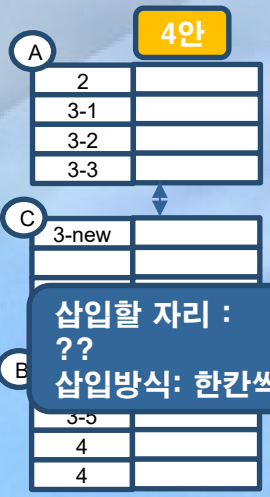
삽입할 자리 :
삽입할 값보다 작거나 같은 레코드 중 가장 뒷자리
삽입방식: 한칸씩 밀려난다



삽입할 자리 :
삽입할 값보다 크거나 같은 레코드 중 가장 앞자리
삽입방식: 한칸씩 밀려난다



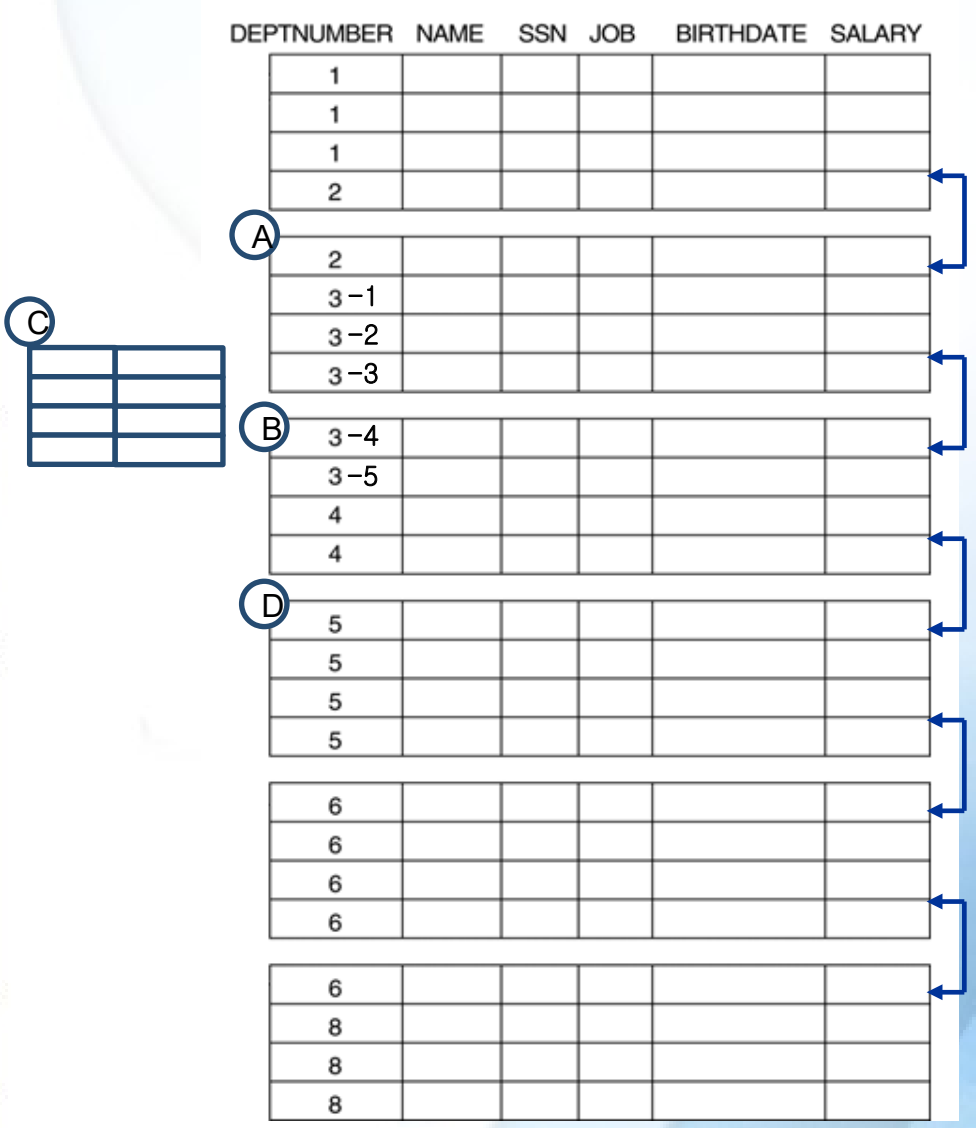
삽입할 자리 :
삽입할 값보다 작거나 같은 레코드 중 가장 뒷자리
삽입방식: ??



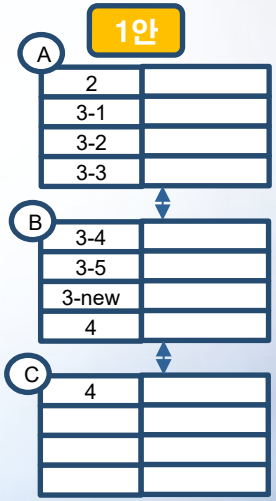
삽입할 자리 :
??
삽입방식: 한칸씩 밀려난다

1. 삽입할 자리를 찾는다.
2. 삽입할 공간을 만든다.
3. 삽입한다.

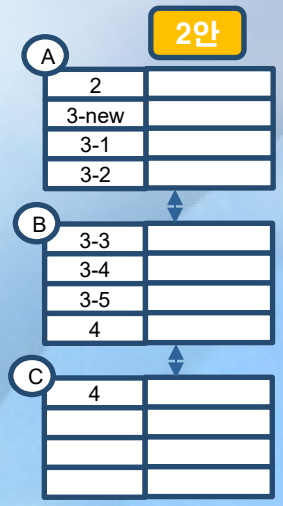
순서화일에서의 삽입안 분석1



디스크 액세스 횟수는?

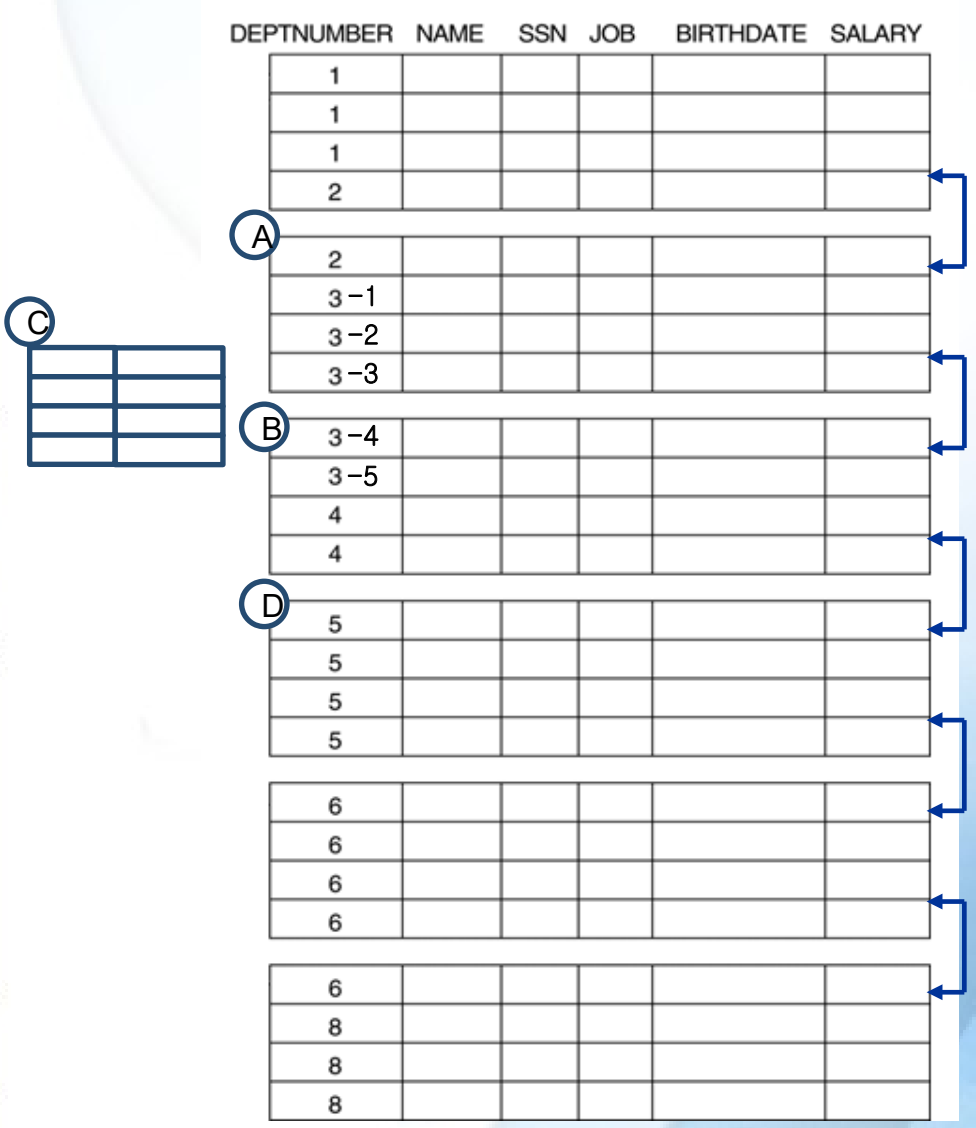


총 5회
삽입할 위치 검색 – 3회
B블록 수정(레코드, 링크) - 0회(위에 포함)
C 블록 수정 (레코드, 링크) – 1회
D블록 링크 수정 – 1회

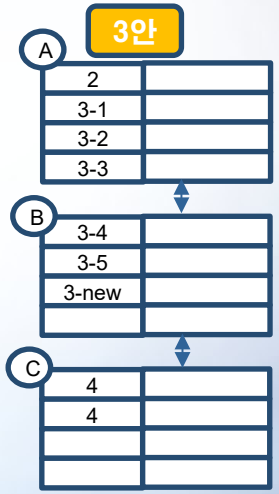


총 5회
삽입할 위치 검색 – 2회
A블록 수정 – 0회(위에 포함)
B블록 수정(레코드, 링크) - 1회
C 블록 수정 (레코드, 링크) – 1회
D블록 링크 수정 – 1회

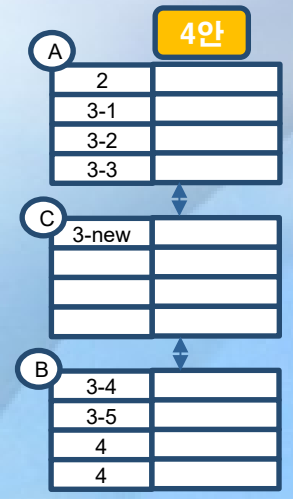
순서화일에서의 삽입분석 2



디스크 액세스 횟수는?



총 5회
삽입할 위치 검색 – 3회
B블록 수정(레코드, 링크) - 0회(위에 포함)
C 블록 수정 (레코드, 링크) – 1회
D블록 링크 수정 – 1회

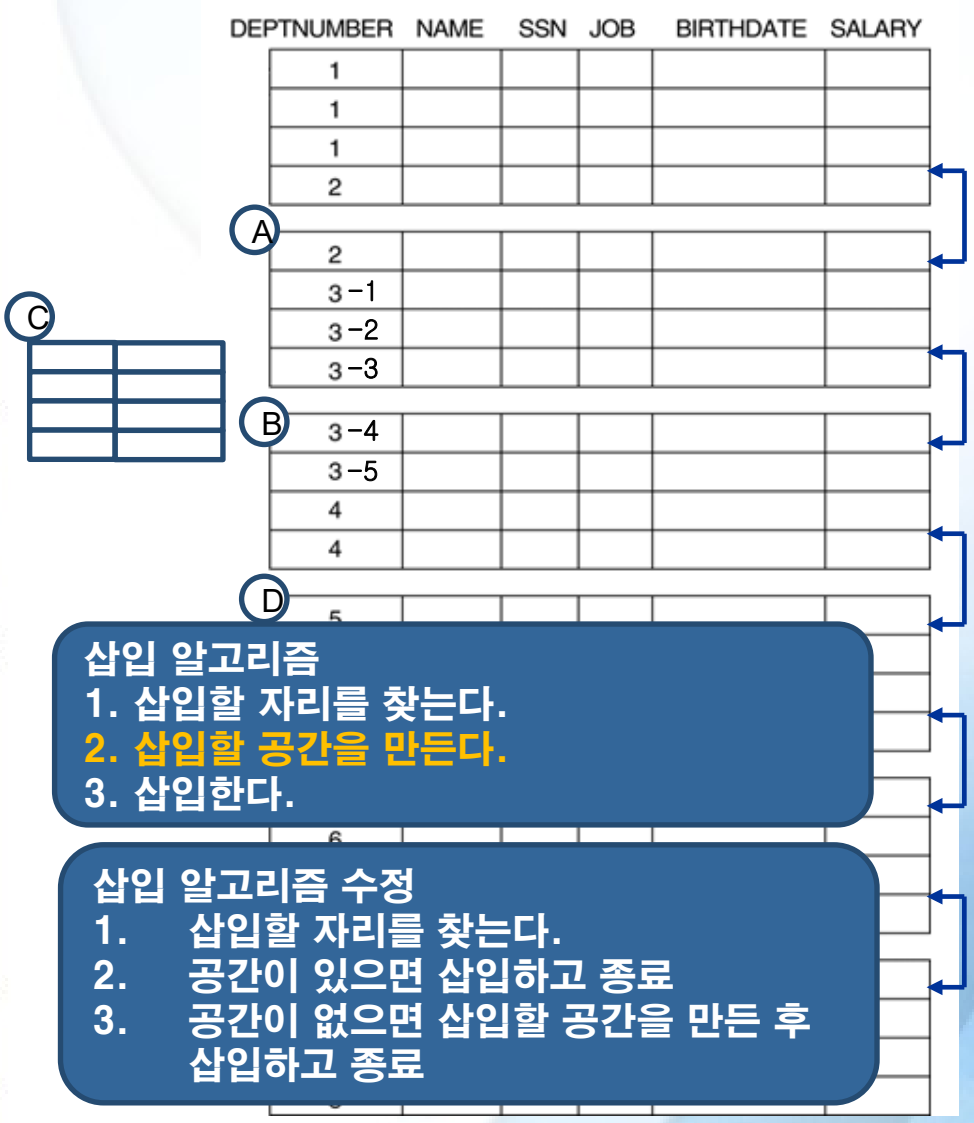


총 4회
삽입할 위치 검색 – 2회
A블록 수정 – 0회(위에 포함)
B블록 수정(레코드, 링크) - 1회
C 블록 수정 (레코드, 링크) – 1회
D블록 링크 수정 – 0회

순서화일에서의 삽입분석 3

이어서 삽입할 레코드는?

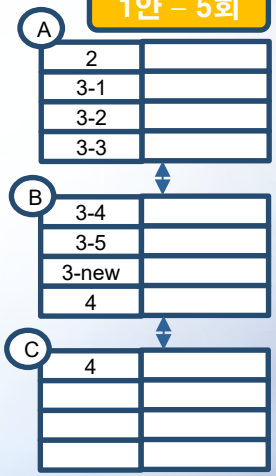
3-new2



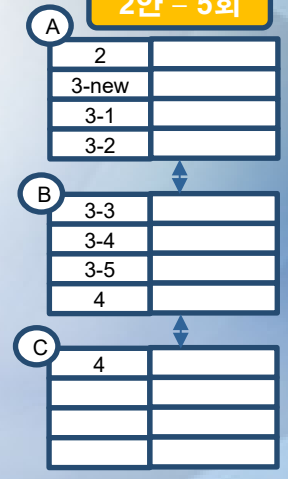
삽입 알고리즘
1. 삽입할 자리를 찾는다.
2. 삽입할 공간을 만든다.
3. 삽입한다.

삽입 알고리즘 수정
1. 삽입할 자리를 찾는다.
2. 공간이 있으면 삽입하고 종료
3. 공간이 없으면 삽입할 공간을 만든 후 삽입하고 종료

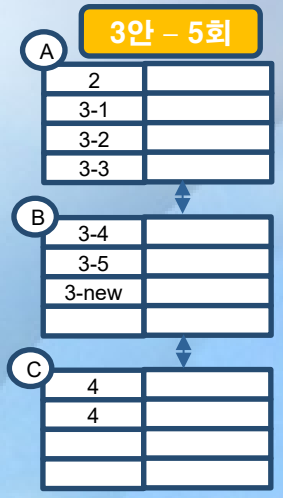
1안 – 5회



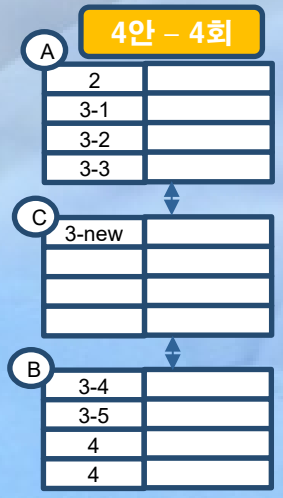
2안 – 5회



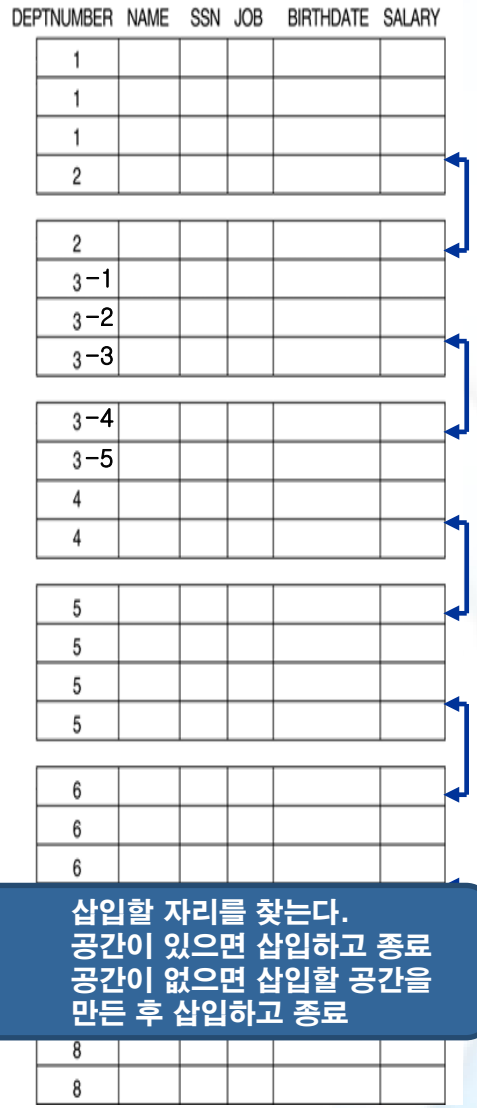
3안 – 5회



4안 – 4회



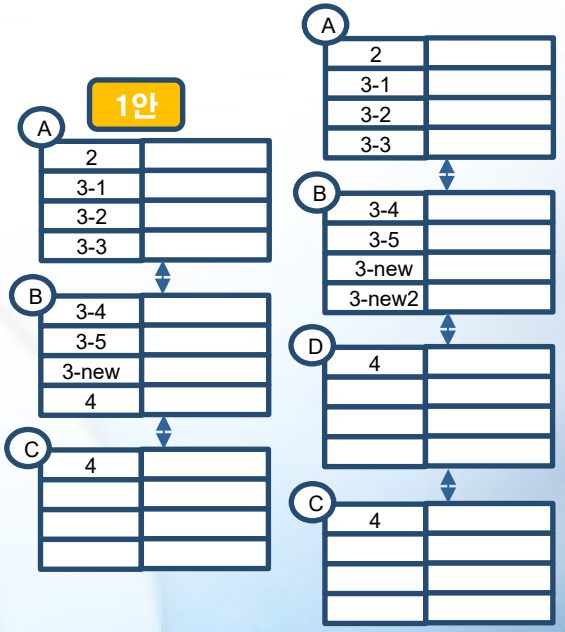
순서화일에서의 연속삽입1



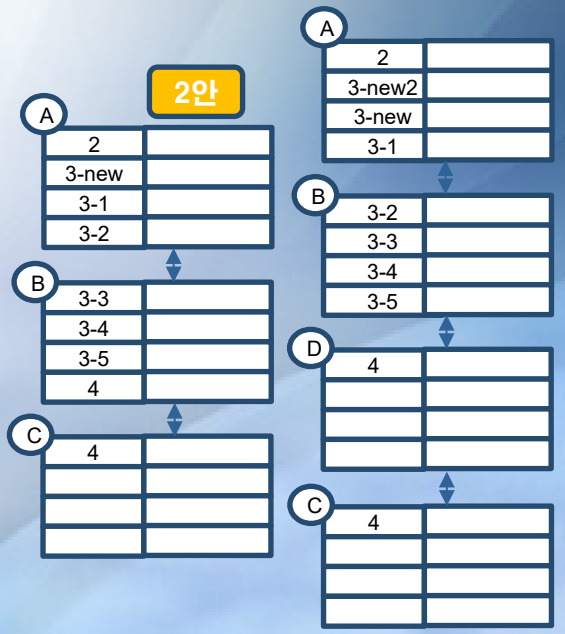
1. 삽입할 자리를 찾는다.
2. 공간이 있으면 삽입하고 종료
3. 공간이 없으면 삽입할 공간을 만든 후 삽입하고 종료

이어서 삽입할 레코드는?

3-new2

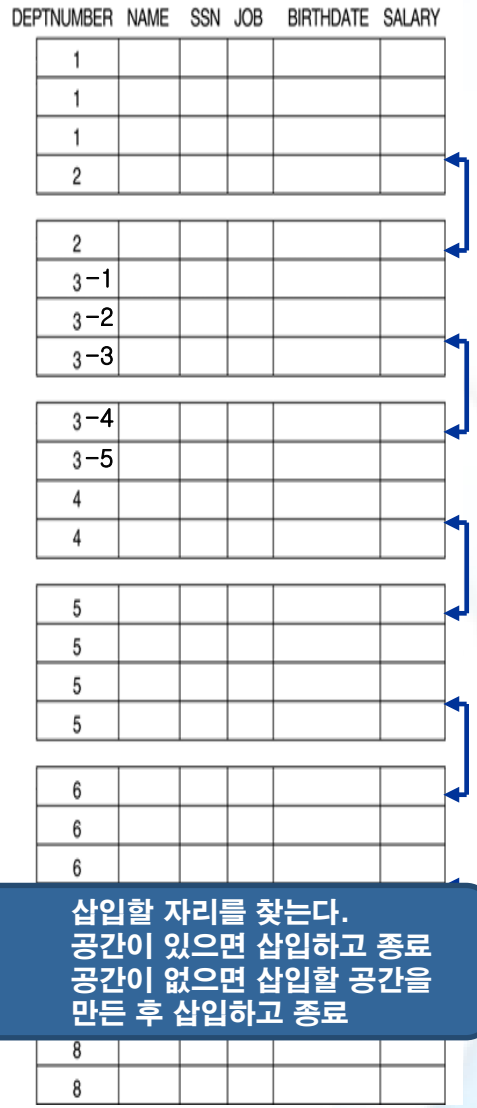


삽입할 자리 :
삽입할 값보다 큰 레코드 중 가장 앞자리
삽입방식: 한칸씩 밀려난다

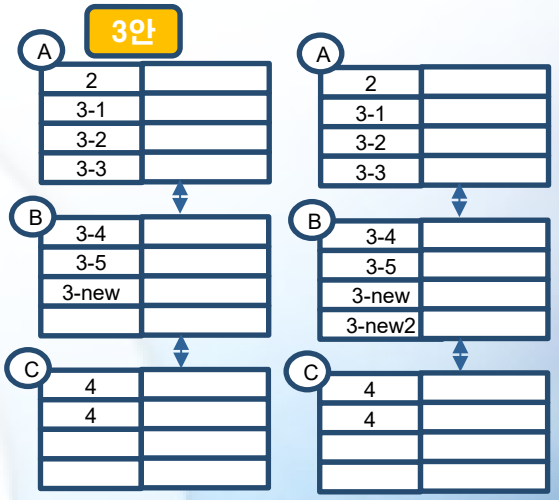


삽입할 자리 :
삽입할 값보다 크거나 같은 레코드 중 가장 앞자리
삽입방식: 한칸씩 밀려난다

순서화일에서의 연속삽입2



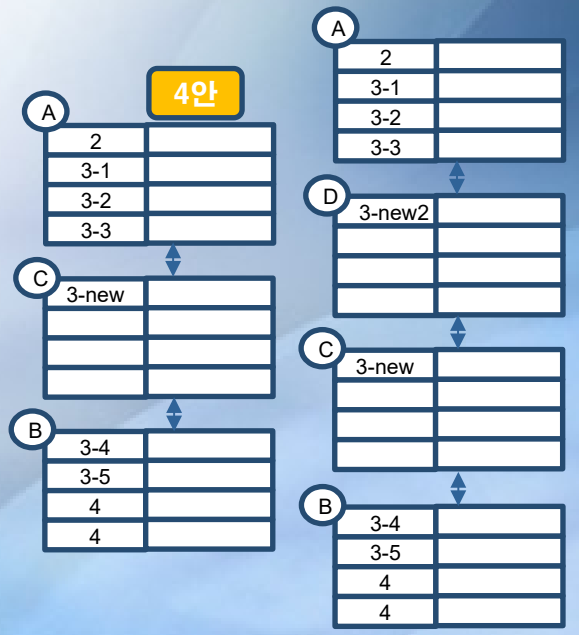
1. 삽입할 자리를 찾는다.
2. 공간이 있으면 삽입하고 종료
3. 공간이 없으면 삽입할 공간을 만든 후 삽입하고 종료



삽입할 자리 :
삽입할 값보다 작거나 같은 레코드 중 가장 뒷자리
삽입방식: ??

이어서 삽입할 레코드는?

3-new2



삽입할 자리 :
??
삽입방식: 한칸씩 밀려난다

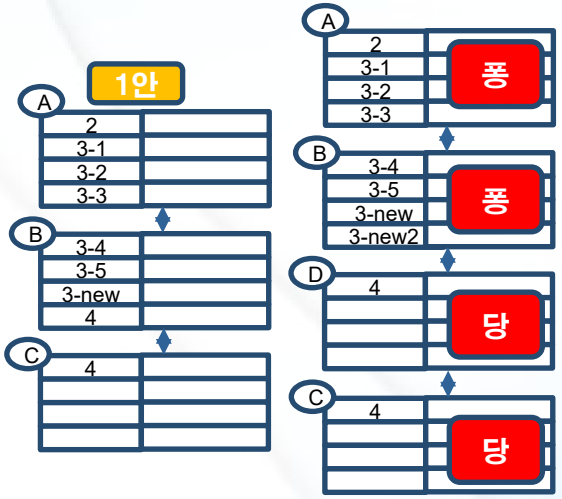
생각해봅시다

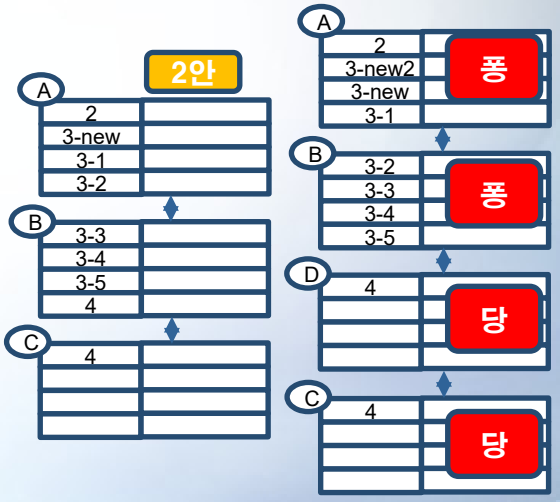
연속적인 삽입이 발생하는 경우
어떠한 일이 벌어지는가?

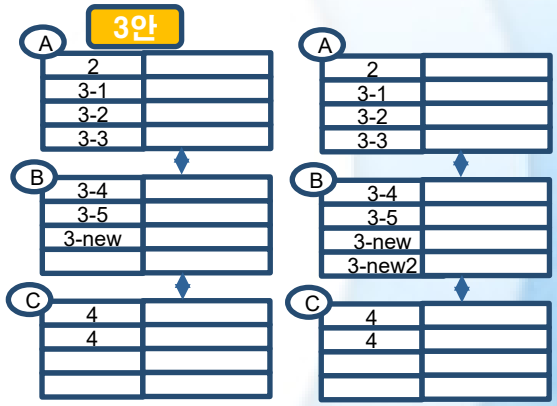
순서화일에서의 연속삽입

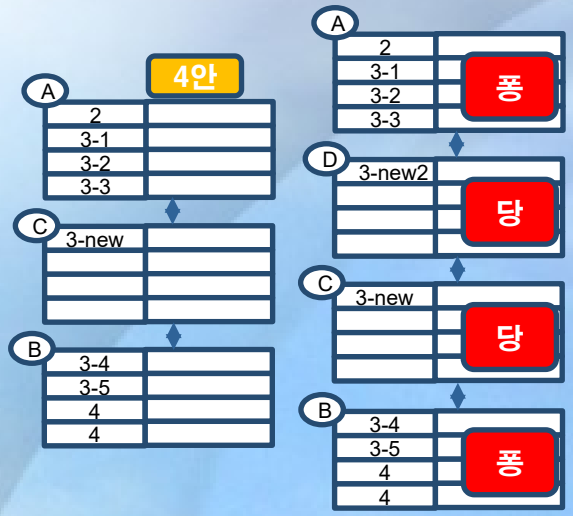
1. 삽입할 자리를 찾는다.
2. 공간이 있으면 삽입하고 종료
3. 공간이 없으면 삽입할 공간을 만든 후 삽입하고 종료

문제점
1. 저장 공간의 낭비 - 레코드 1개인 블록 증가
2. 성능 저하 - 블록의 개수 증가
   ➔ 3안에서 힌트를 얻자

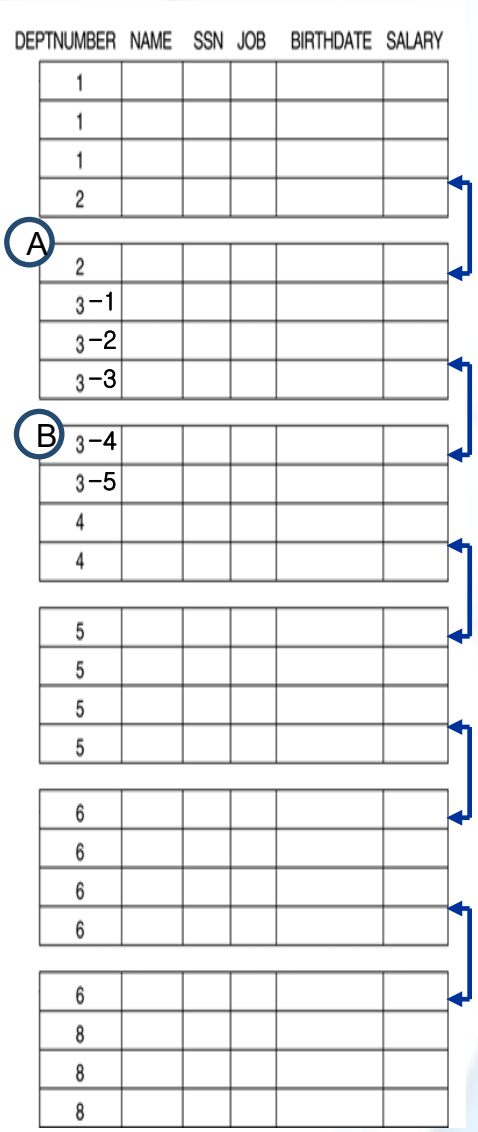




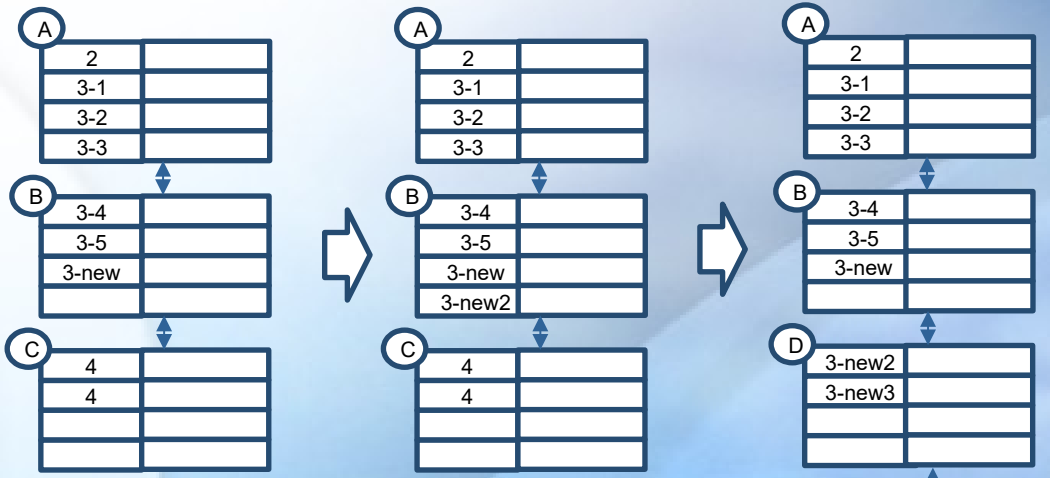




순서화일에서의 삽입 다시



1. 삽입할 자리를 찾는다.
2. 찾은 블록(A)에 공간이 있으면 삽입하고 종료
3. A에 공간이 없으면
   3-1. 새로운 블록(B)을 할당하여 A뒤에 연결한다.
   3-2. A에 저장되어 있던 레코드와 삽입할 레코드를 A와 B에 절반씩 나누어 저장하고 종료한다.

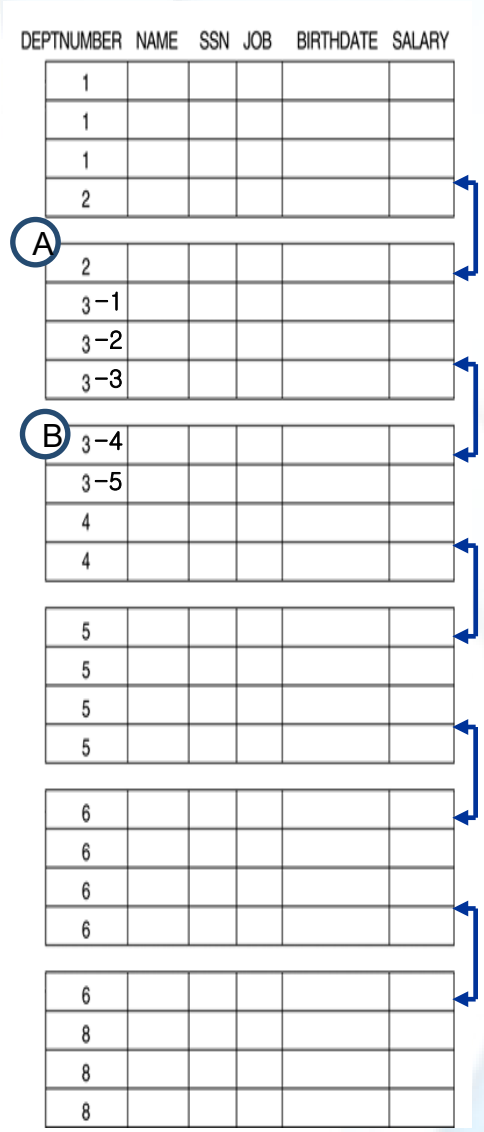


문제점은 무엇인가?

정리해봅시다

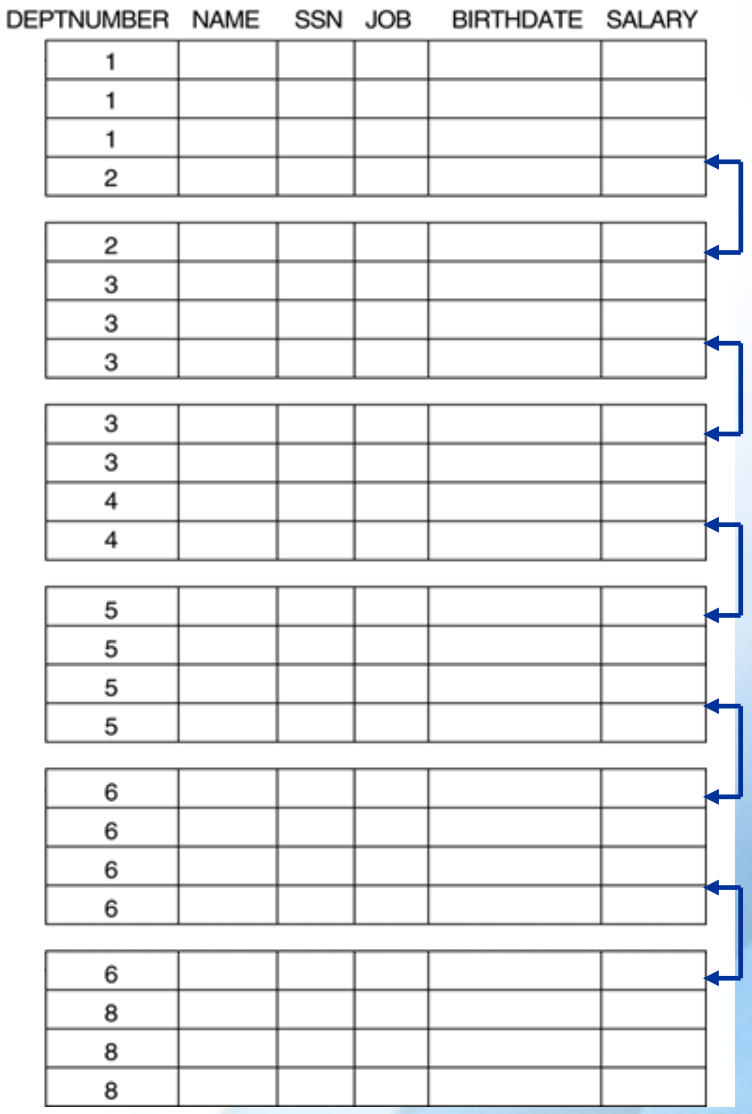
순서파일에서 검색 및 삽입
알고리즘을 정리해봅시다.

순서화일에서의 삽입



1. 삽입할 자리를 찾는다.
2. 찾은 블록(A)에 공간이 있으면 삽입하고 종료
3. A에 공간이 없으면
   3-1. 새로운 블록(B)을 할당하여 A뒤에 연결한다.
   3-2. A에 저장되어 있던 레코드와 삽입할 레코드를 A와 B에 절반씩 나누어 저장하고 종료한다.

순서화일에서의 삭제

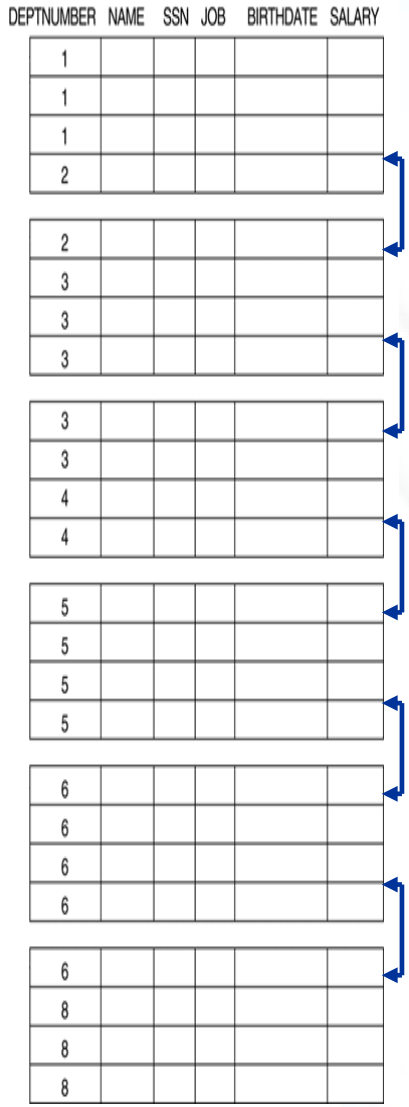


1. 삭제할 레코드를 찾는다.
2. 삭제한다.
3. 삭제한 빈 공간을 처리한다.

삭제한 빈 공간을 어떻게 처리할 것인가?

순서화일에서의 삭제

1. 삭제할 레코드를 찾는다.
2. 삭제한다.
3. 삭제한 빈 공간을 처리한다.

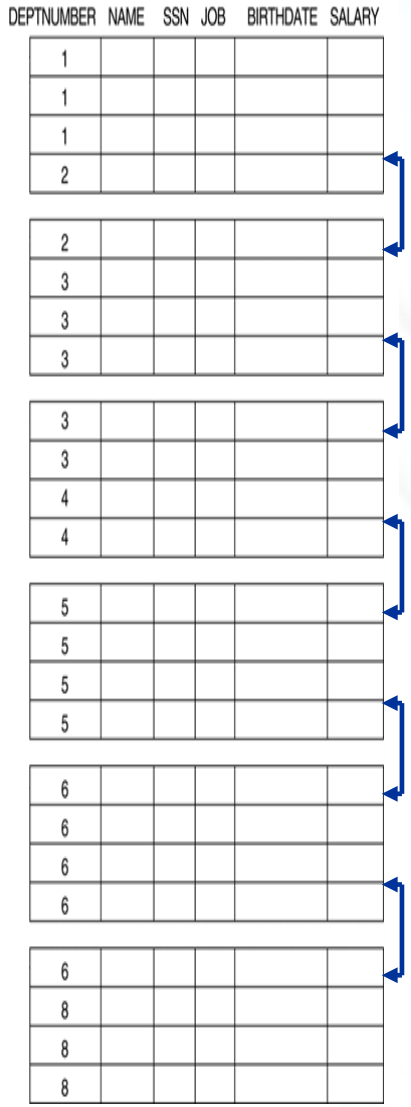


삭제한 빈 공간을 언제, 어떻게 처리할 것인가?

1. 그냥 둔다. 즉, 처리하지 않는다.
2. 블록(A) 내 레코드 개수가 0이 되면 블록을 삭제한다.
3. 블록(A) 내 레코드 개수가 X가 되면 앞뒤블록과 합친다.

순서화일에서의 삭제

1. 삭제할 레코드를 찾는다.
2. 삭제한다.
3. 삭제한 빈 공간을 처리한다.



삭제한 빈 공간을 언제, 어떻게 처리할 것인가?

1. 그냥 둔다. 즉, 처리하지 않는다.

2. 블록(A) 내 레코드 개수가 0이 되면 블록을 삭제한다.

3. 블록(A) 내 레코드 개수가 X가 되면 앞뒤블록과 합친다.

블록 사용률(a)이 50% 이하가 되면
3-1. 다음 블록(B)을 읽어 블록사용률(b)를 계산한다.
3-2. a+b <= 100%이면 합친다.
3-3. a+b > 100%이면 B에서 A로 일부 레코드를 이동하여 A,B 모두 50% 이상이 되도록 한다.

삽입, 삭제 정리해봅시다

삽입

1. 삽입할 자리를 찾는다.
2. 찾은 블록(A)에 공간이 있으면 삽입하고 종료
3. A에 공간이 없으면
   3-1. 새로운 블록(B)을 할당하여 A뒤에 연결한다.
   3-2. A에 저장되어 있던 레코드와 삽입할 레코드를 A와 B에 절반씩 나누어 저장하고 종료한다.

삭제

1. 삭제할 레코드를 찾는다. 찾은 블록을 A라한다.
2. 삭제한다.
3. 블록 사용률(a)이 50% 이하가 되면
   3-1. 다음 블록(B)을 읽어 블록사용률(b)를 계산한다.
   3-2. a+b <= 100%이면 합친다.
   3-3. a+b > 100%이면 B에서 A로 일부 레코드를 이동하여 A,B 모두 50% 이상이 되도록 한다.

이 방법의 장단점을 분석해보자.

저장공간 사용량의 관점에서?

하드디스크 액세스 횟수의 관점에서?

인덱스란?

레코드를 빠르게 찾을 수 있도록
도와주는 보조 화일

화일에 대한 또 다른 접근 경로

인덱스 엔트리

인덱스 화일의 레코드

인덱스 엔트리의 구조

[키 값들, 레코드 주소]

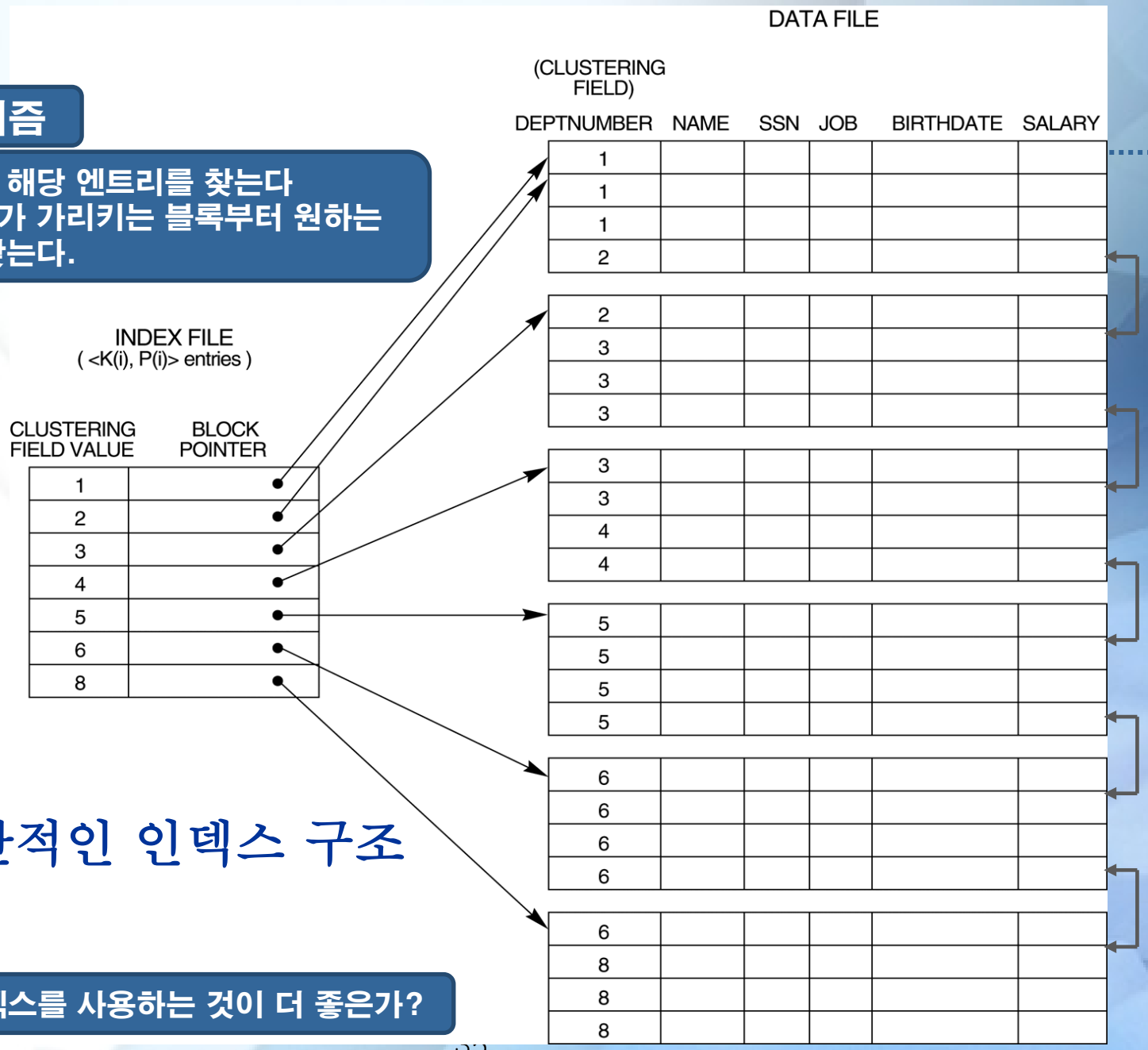
인덱스 엔트리의 구조

[키 값들, 레코드 주소]

일반적인 인덱스 구조

검색 알고리즘

1. 인덱스에서 해당 엔트리를 찾는다
2. 블록포인터가 가리키는 블록부터 원하는 레코드를 찾는다.



일반적인 인덱스 구조

검색할 때 인덱스를 사용하는 것이 더 좋은가?

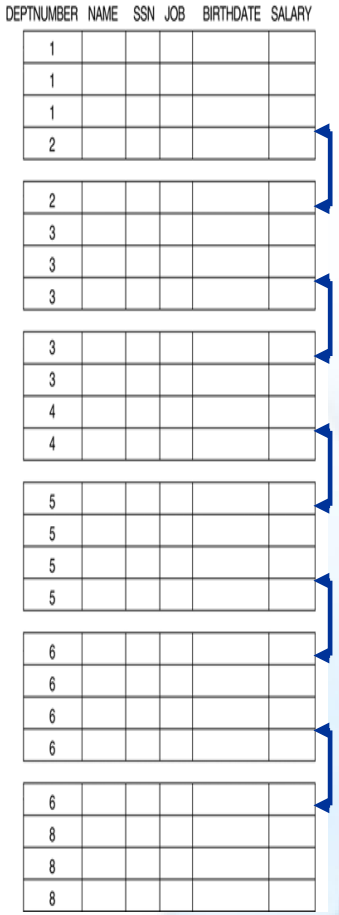
검색 : 순서화일 vs 인덱스

순서화일의 **처음부터** 순차검색 한다.

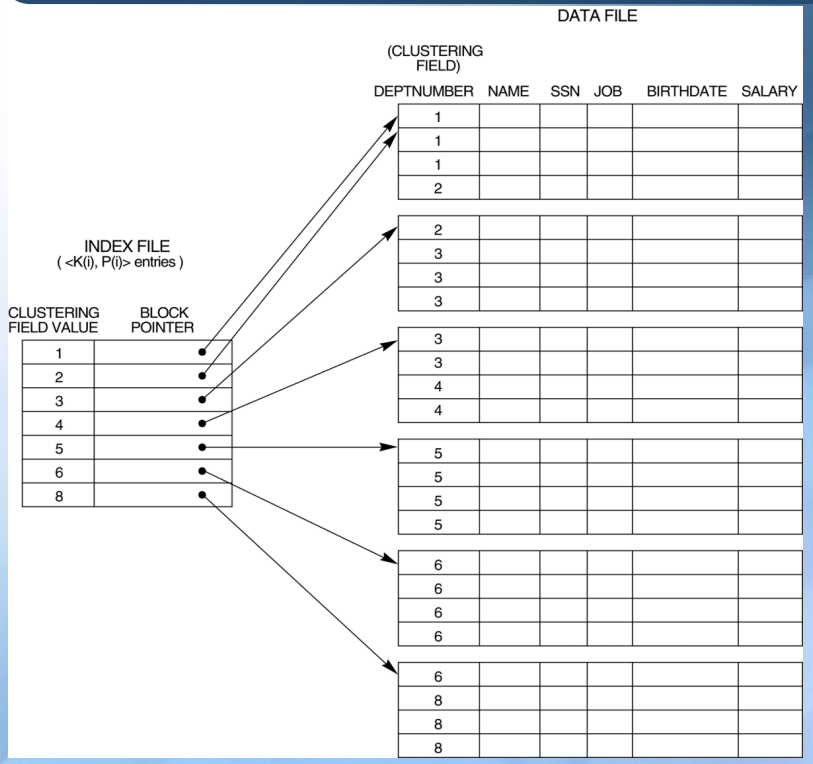
Search 1

Search 3

Search 8

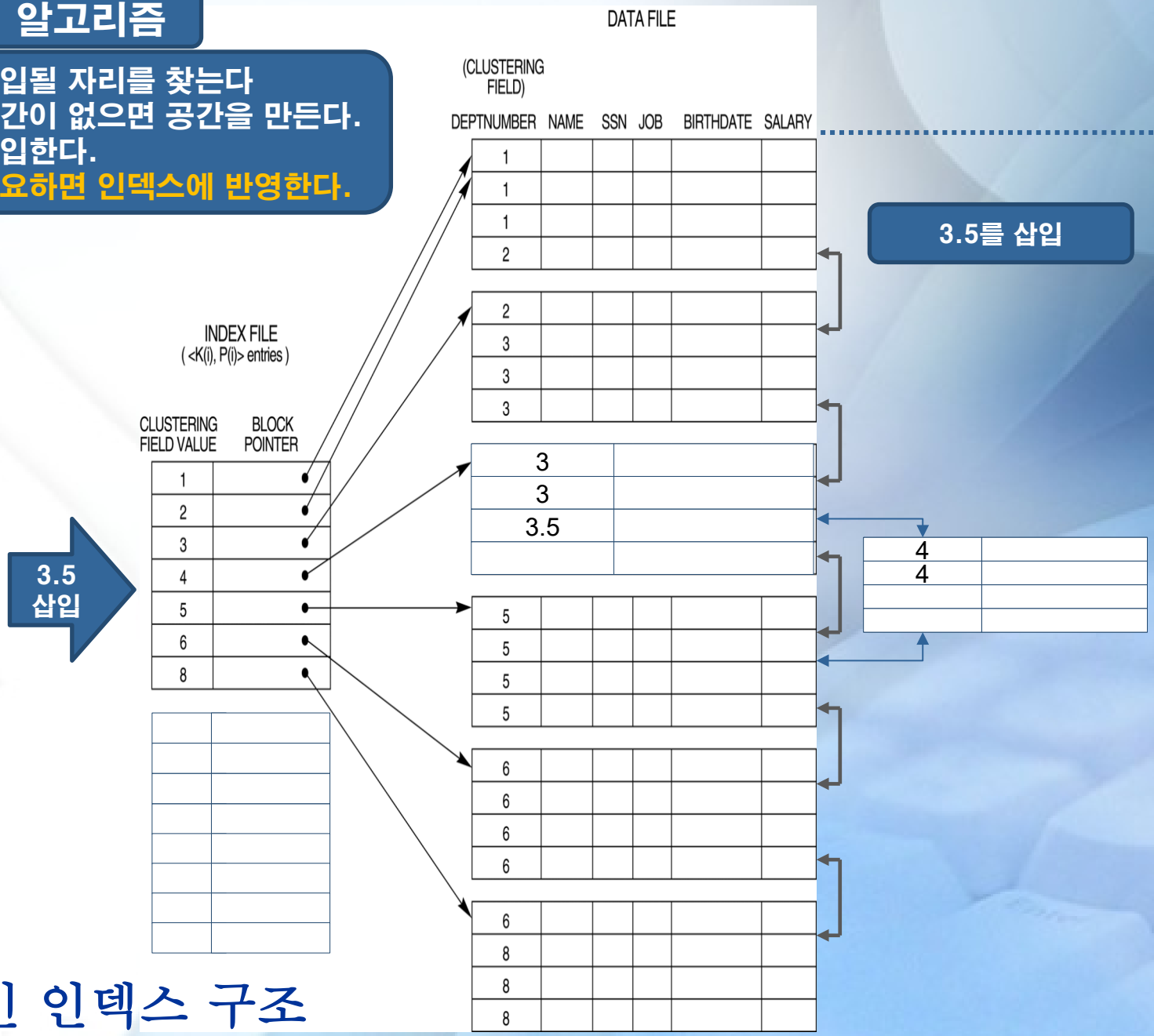


1. 인덱스에서 해당 엔트리를 찾는다
2. **블록포인터가 가리키는 블록부터** 순차검색 한다.



삽입 알고리즘

1. 삽입될 자리를 찾는다
2. 공간이 없으면 공간을 만든다.
3. 삽입한다.
4. 필요하면 인덱스에 반영한다.

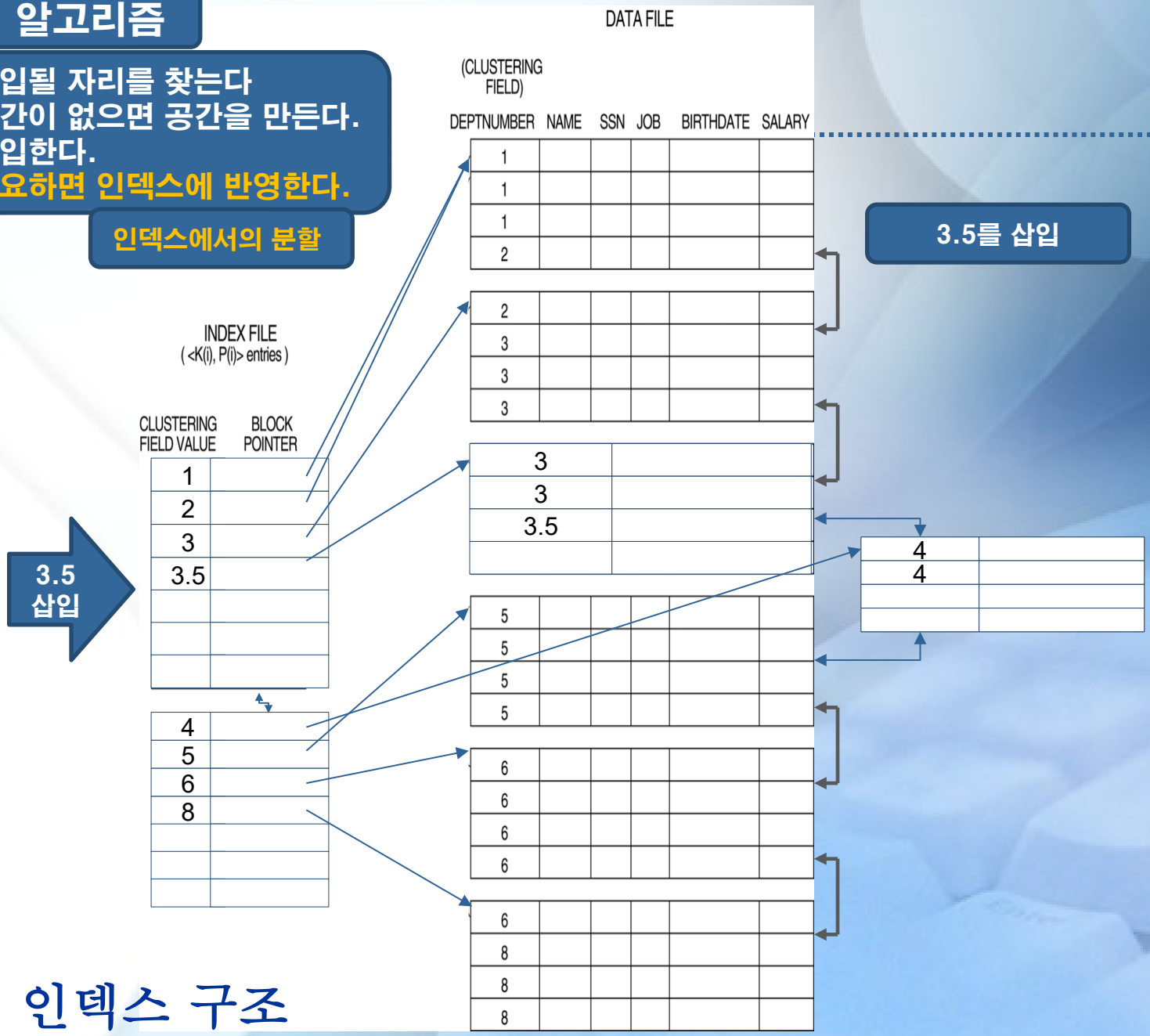


일반적인 인덱스 구조

삽입 알고리즘

1. 삽입될 자리를 찾는다
2. 공간이 없으면 공간을 만든다.
3. 삽입한다.
4. 필요하면 인덱스에 반영한다.

인덱스에서의 분할

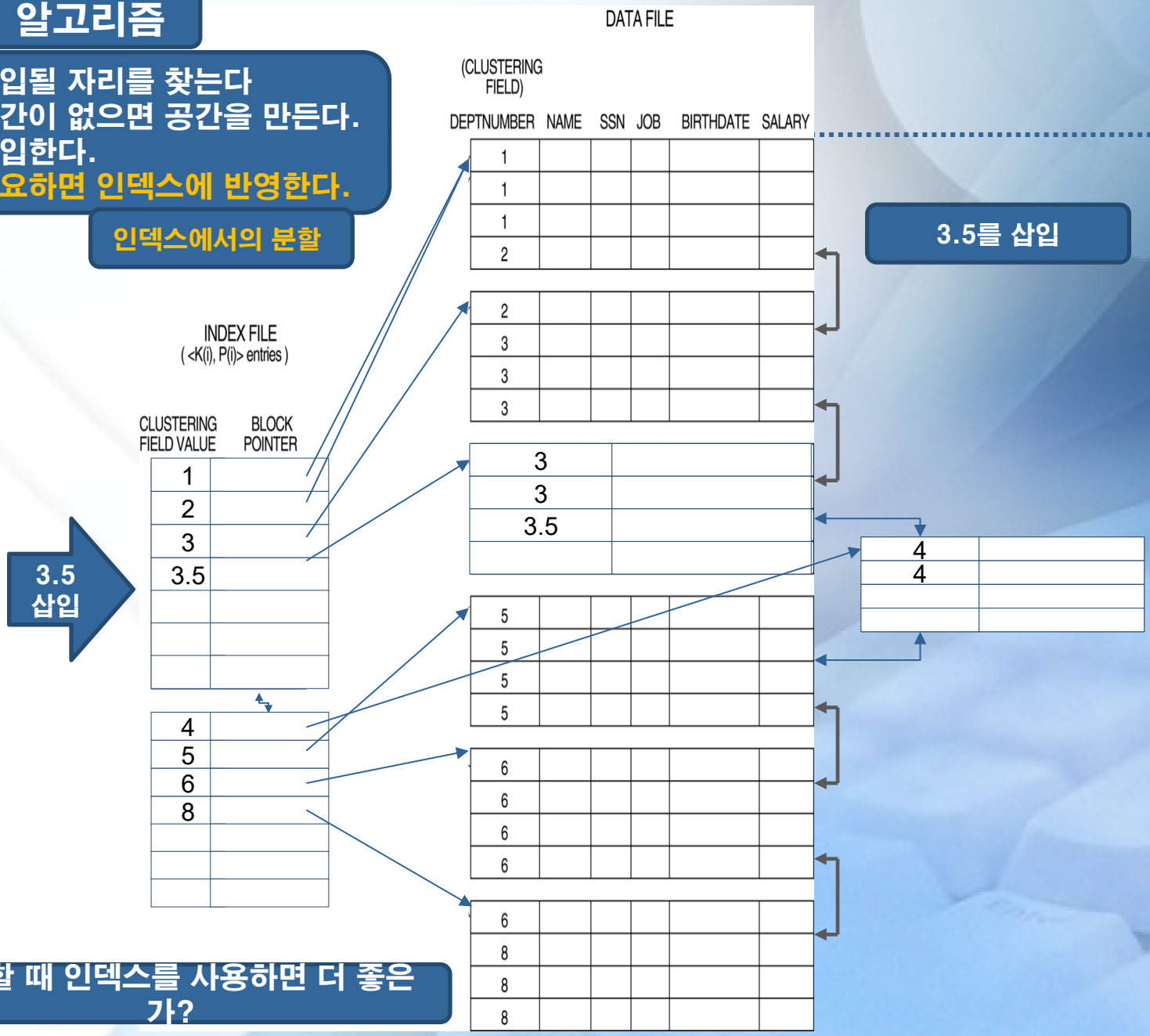


일반적인 인덱스 구조

삽입 알고리즘

1. 삽입될 자리를 찾는다
2. 공간이 없으면 공간을 만든다.
3. 삽입한다.
4. 필요하면 인덱스에 반영한다.

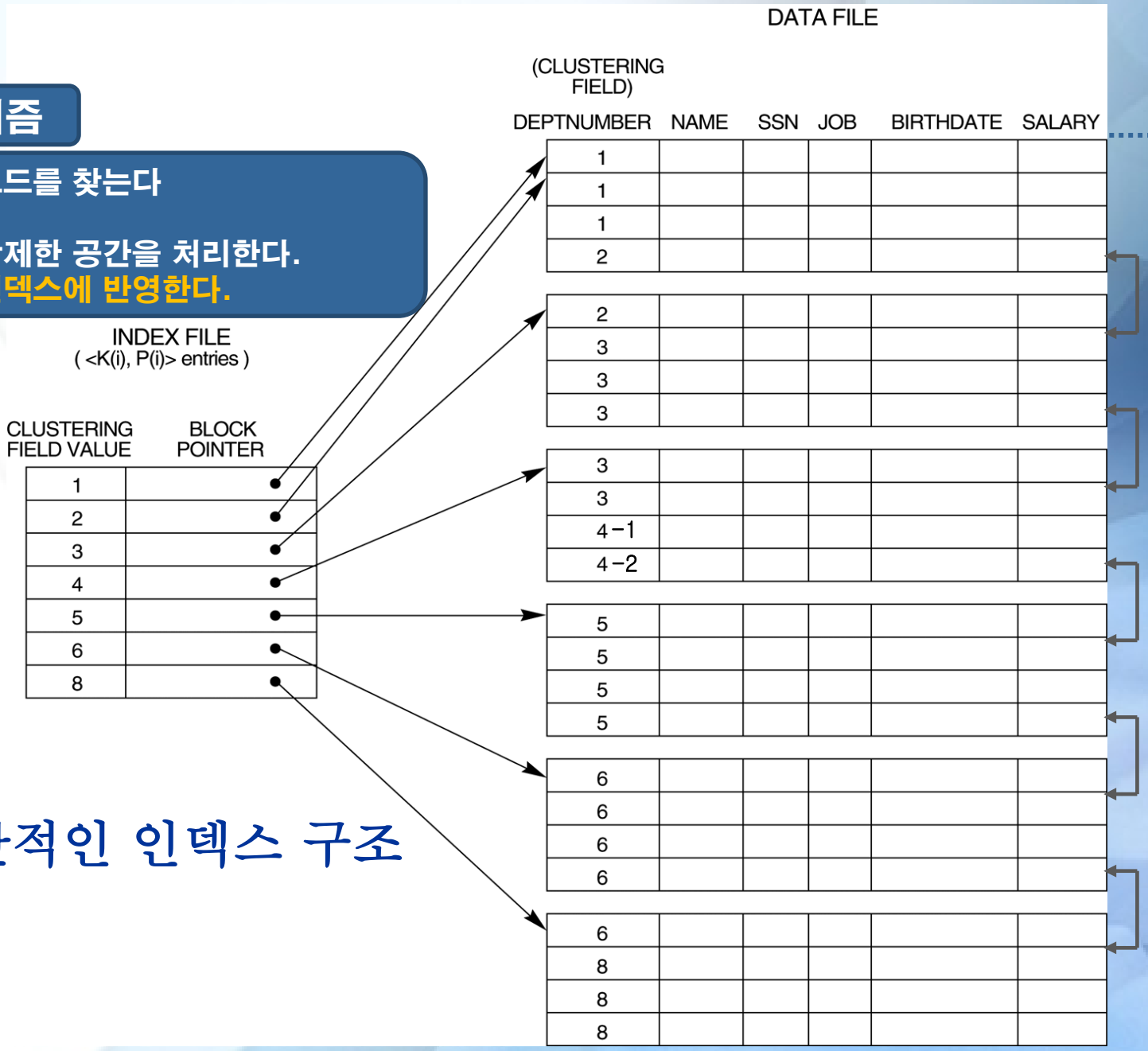
인덱스에서의 분할



삽입할 때 인덱스를 사용하면 더 좋은가?

삭제 알고리즘

1. 삭제할 레코드를 찾는다
2. 삭제한다.
3. 필요하면 삭제한 공간을 처리한다.
4. 필요하면 인덱스에 반영한다.

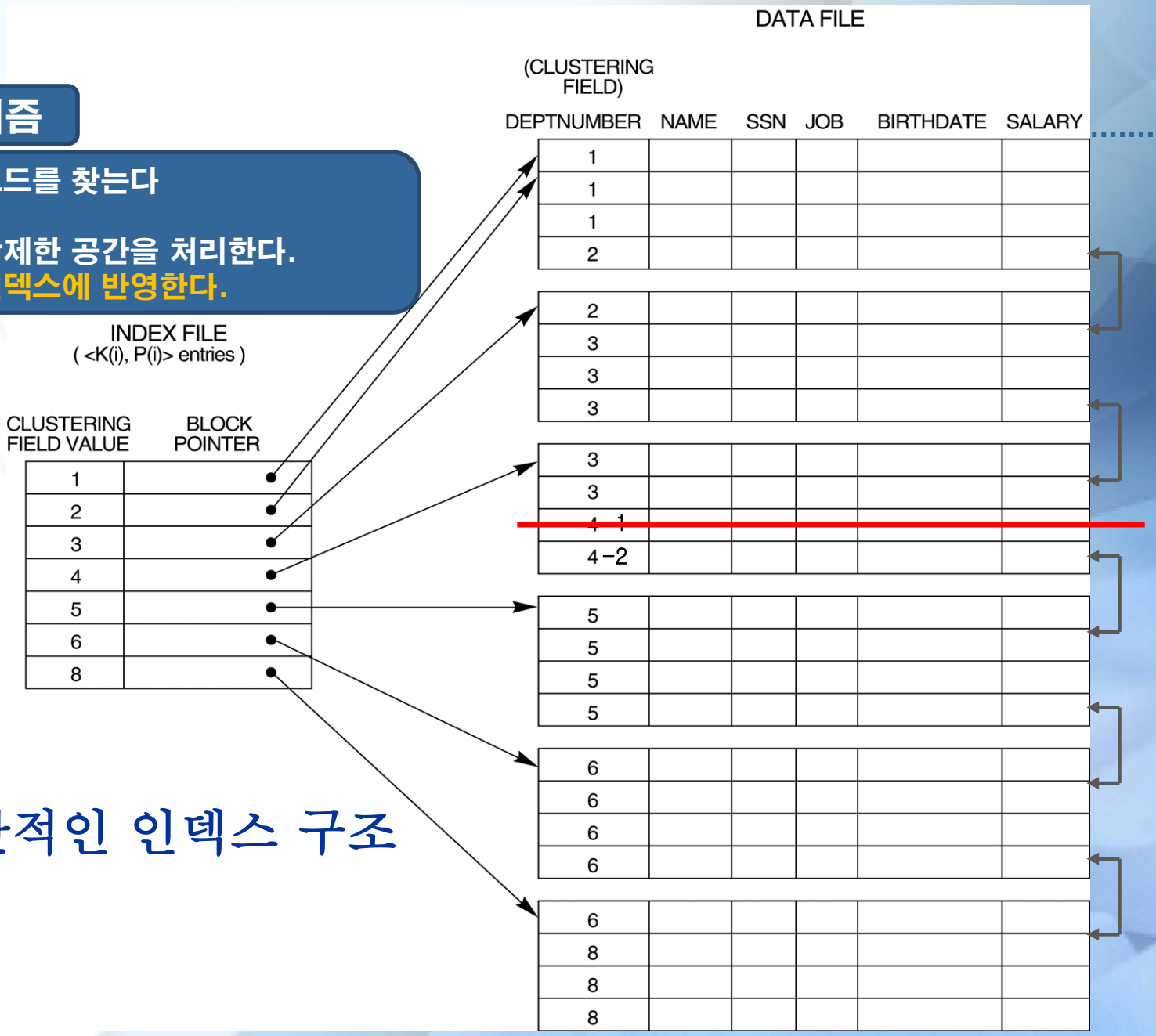


일반적인 인덱스 구조

삭제 알고리즘

1. 삭제할 레코드를 찾는다
2. 삭제한다.
3. 필요하면 삭제한 공간을 처리한다.
4. 필요하면 인덱스에 반영한다.

Delete 4-1



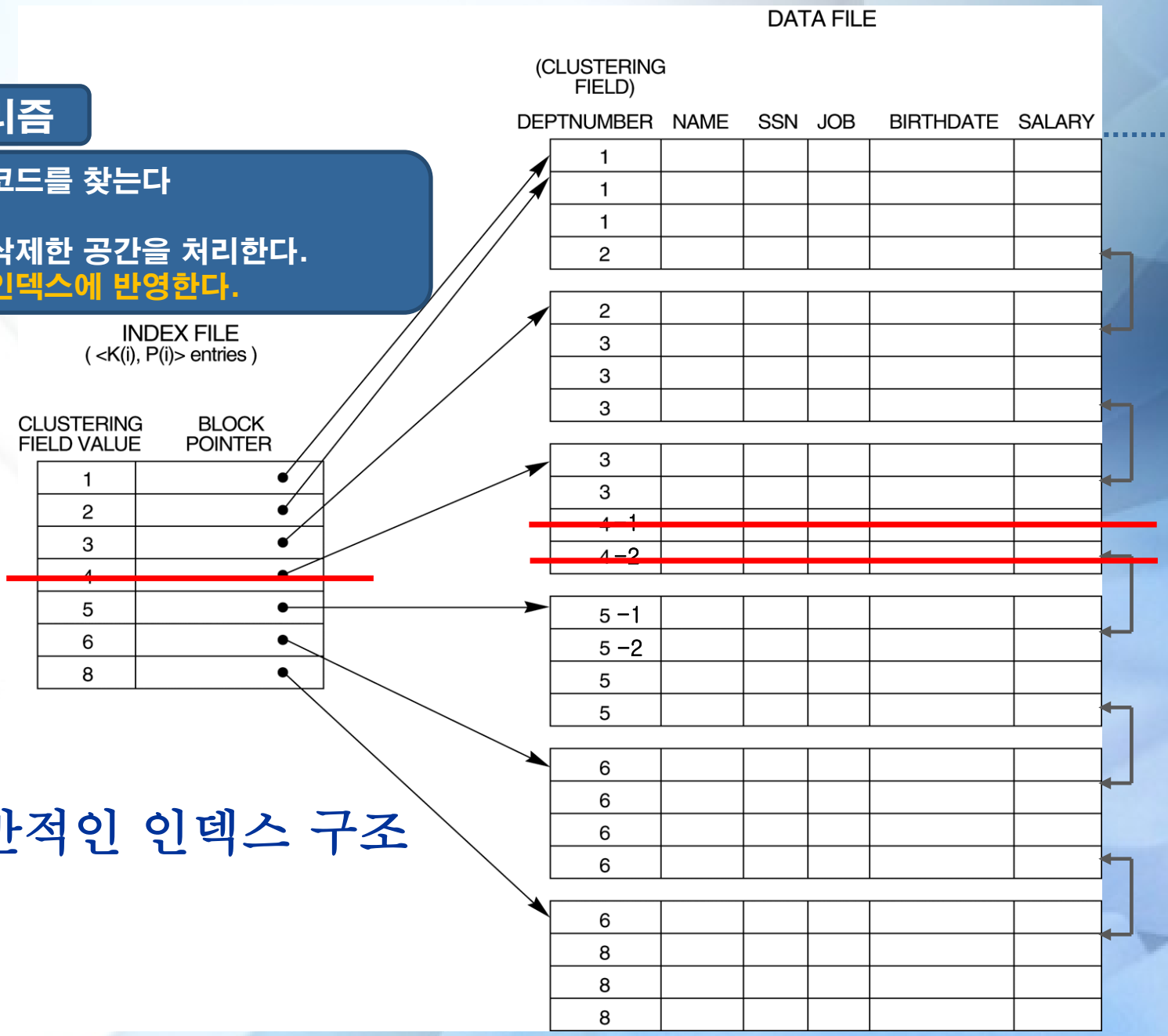
일반적인 인덱스 구조

삭제 알고리즘

1. 삭제할 레코드를 찾는다
2. 삭제한다.
3. 필요하면 삭제한 공간을 처리한다.
4. 필요하면 인덱스에 반영한다.

Delete 4-1

Delete 4-2



일반적인 인덱스 구조

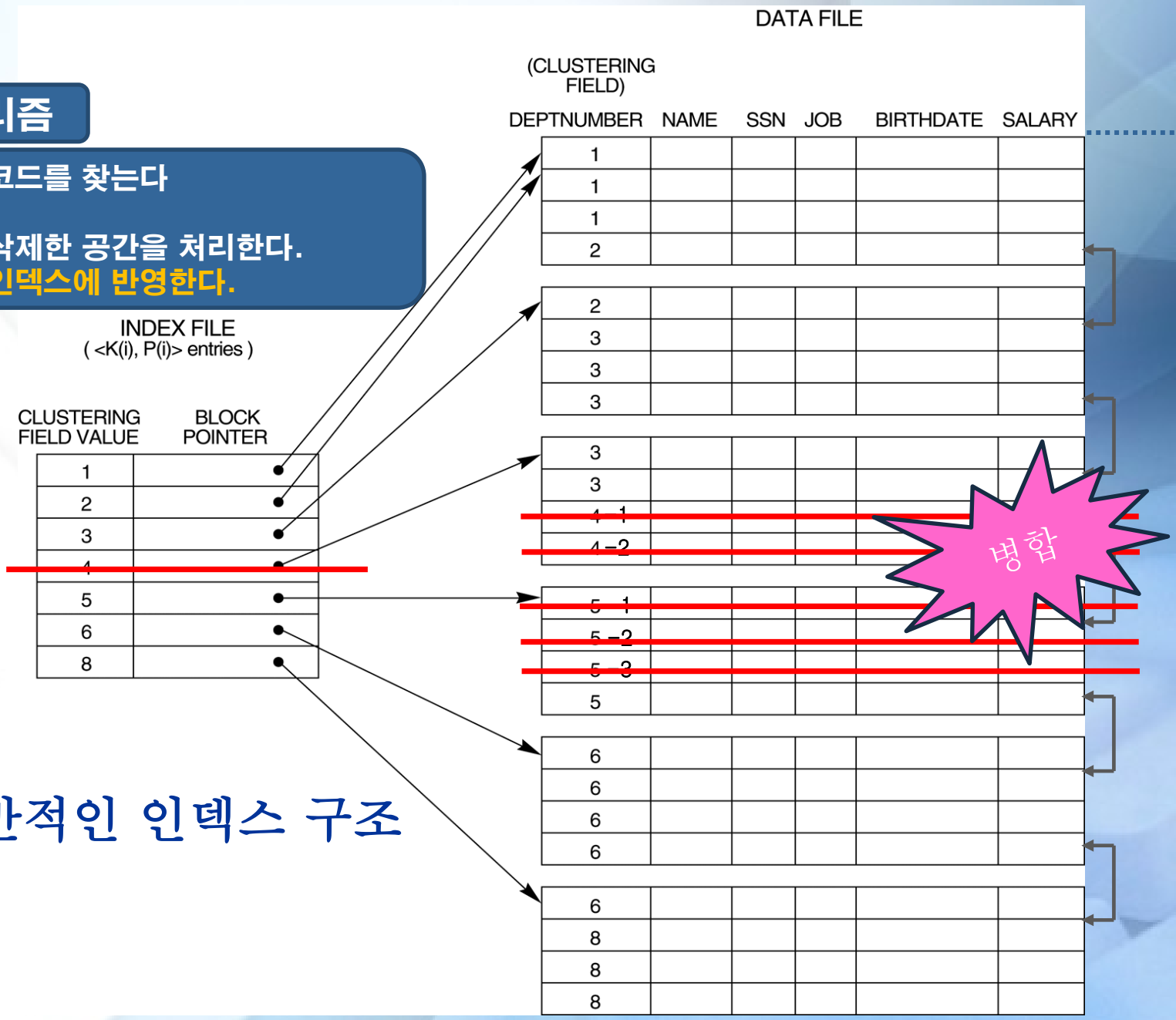
삭제 알고리즘

1. 삭제할 레코드를 찾는다
2. 삭제한다.
3. 필요하면 삭제한 공간을 처리한다.
4. 필요하면 인덱스에 반영한다.

Delete 4-1

Delete 4-2

Delete 5-1
Delete 5-2
Delete 5-3



일반적인 인덱스 구조

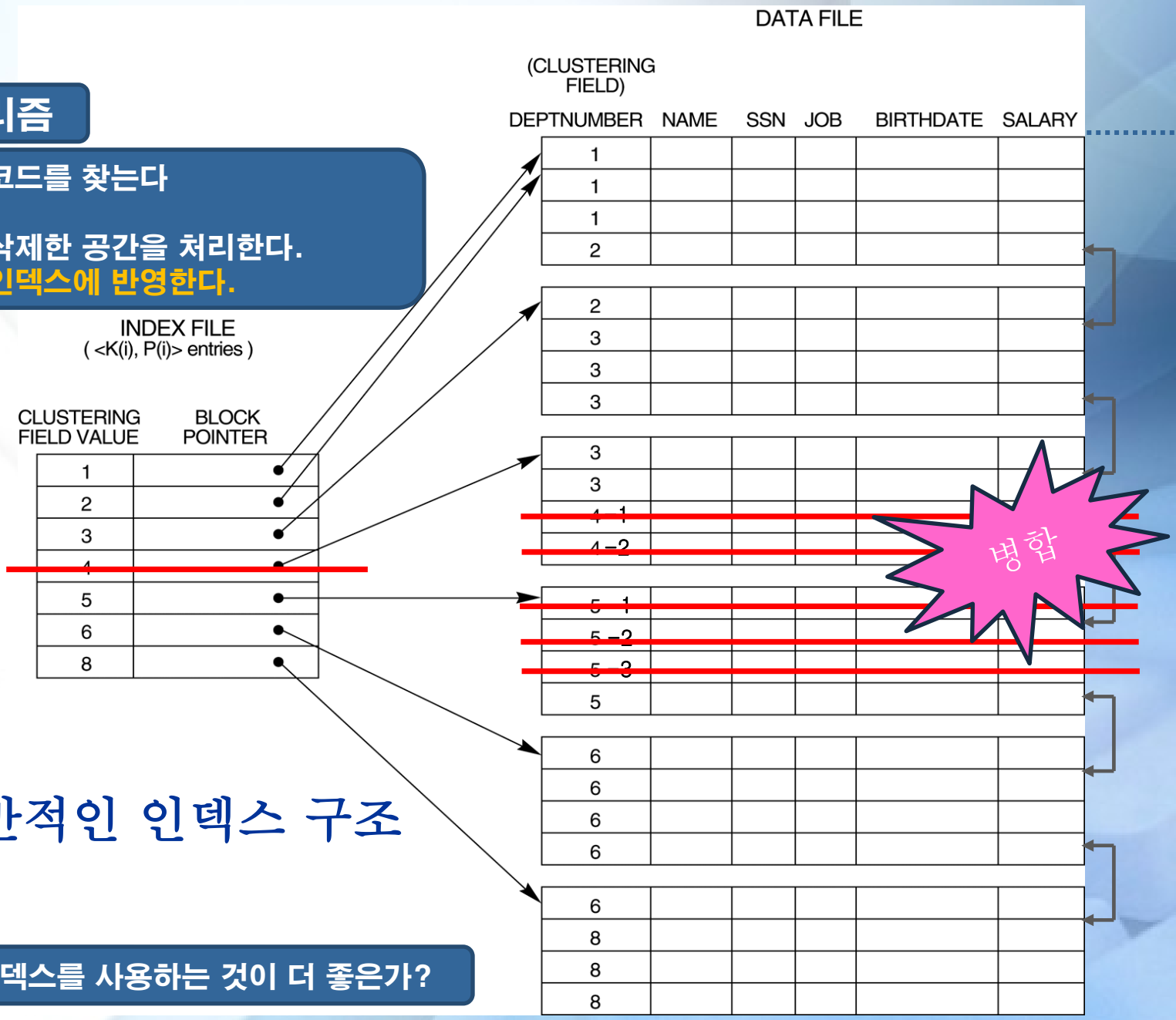
삭제 알고리즘

1. 삭제할 레코드를 찾는다
2. 삭제한다.
3. 필요하면 삭제한 공간을 처리한다.
4. 필요하면 인덱스에 반영한다.

Delete 4-1

Delete 4-2

Delete 5-1
Delete 5-2
Delete 5-3



일반적인 인덱스 구조

삭제할 때 인덱스를 사용하는 것이 더 좋은가?

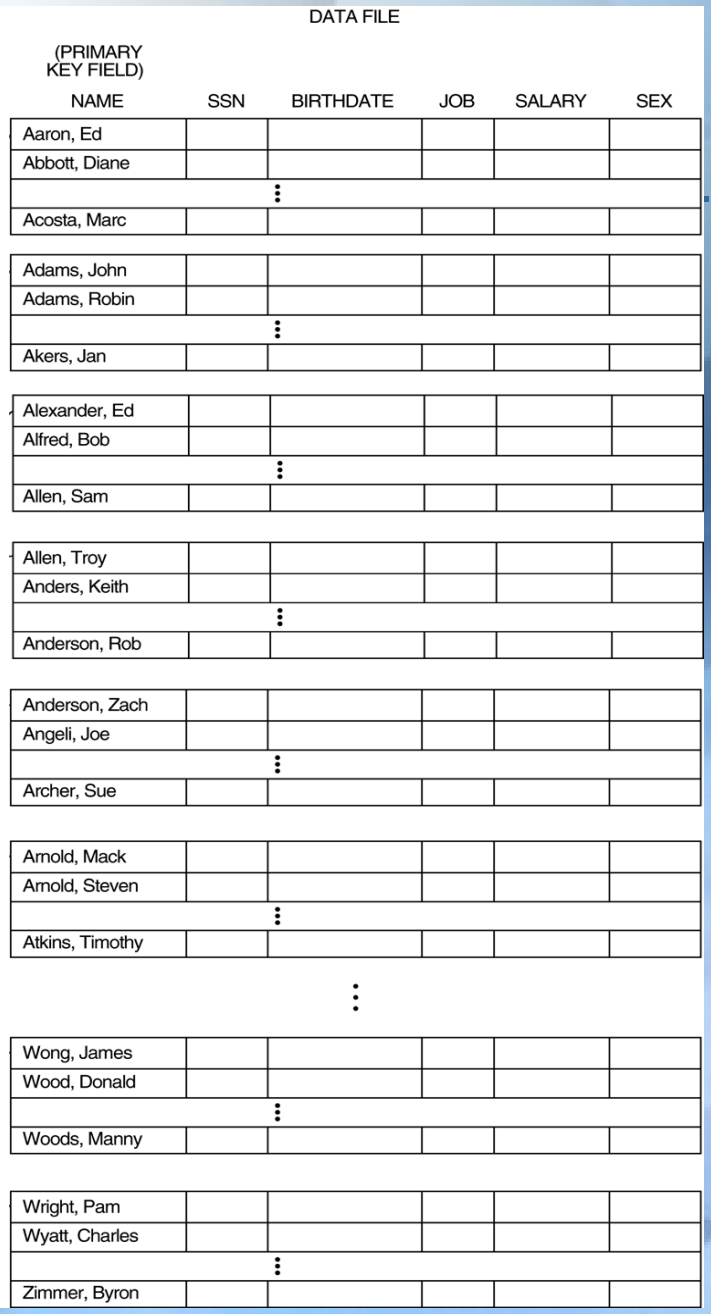
생각해 봅시다.

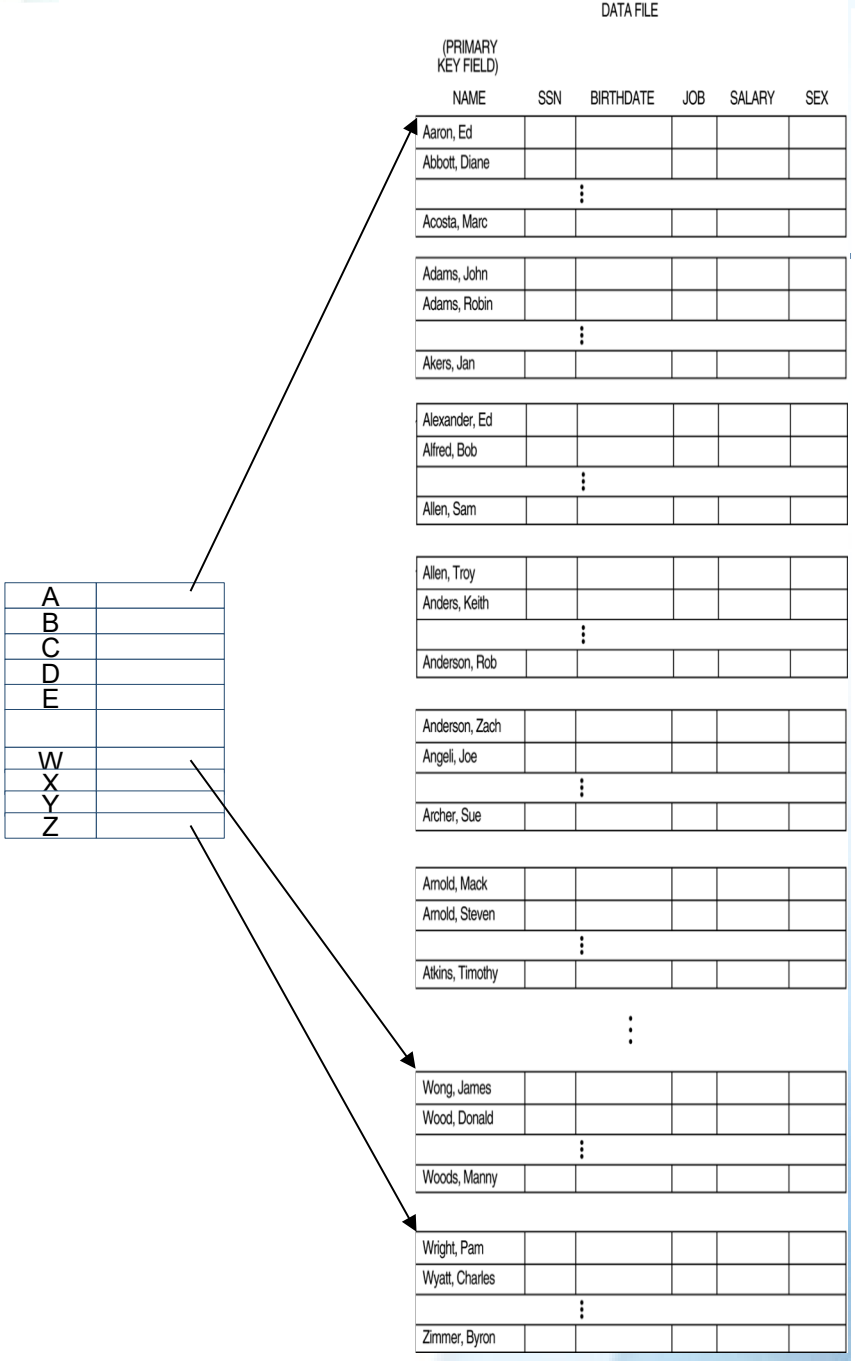
1. 인덱스 화일의 크기는 어느 정도 일까?
2. 인덱스 엔트리의 개수는 몇 개인가?
3. 하나의 화일에 몇 개의 인덱스를 만들수 있을까?

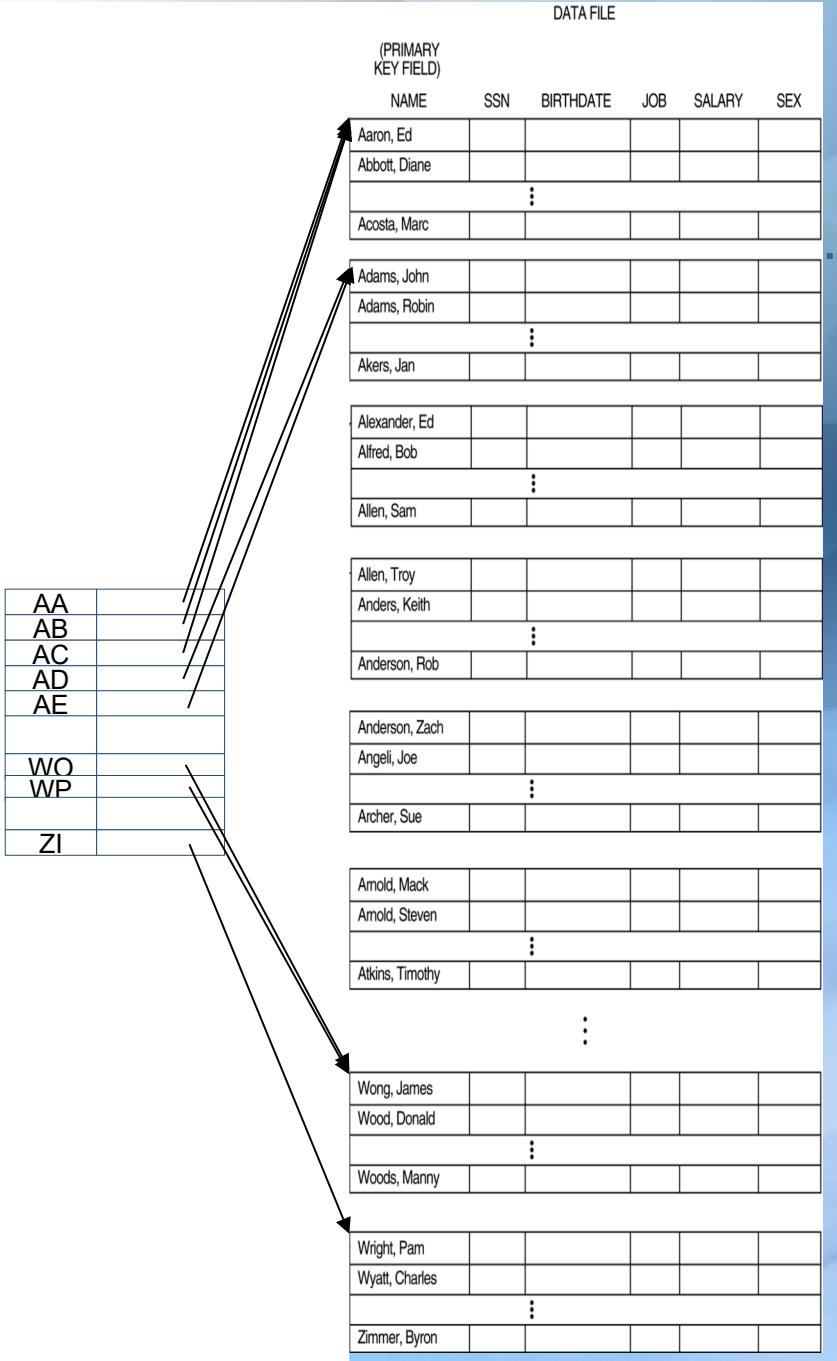
중복 없는 필드에 의한 순서화일을 위한 인덱스에 대하여 생각해 보자.

앞에서 배운 방식으로 만들면 될까?

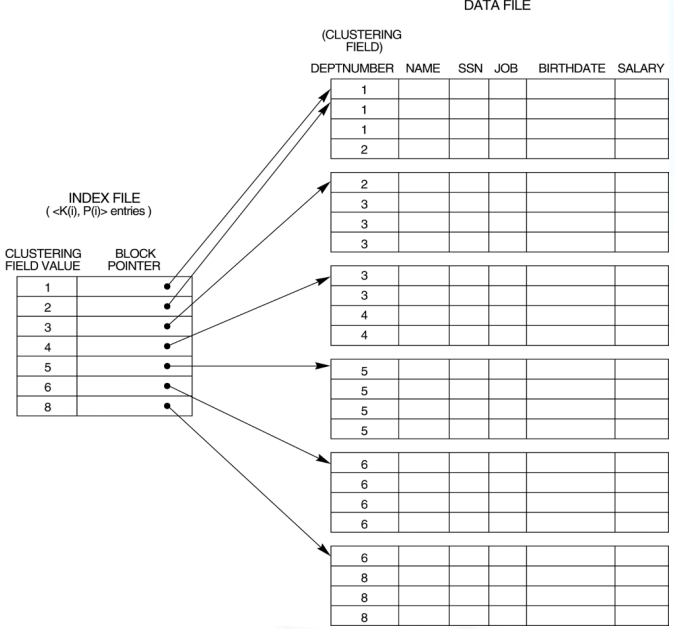
검색, 삽입, 삭제 연산을 생각해보자.

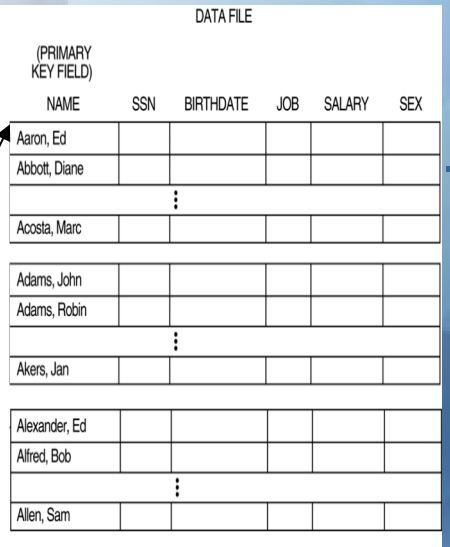



DATA FILE
(PRIMARY KEY FIELD)
NAME
SSN
BIRTHDATE
JOB
SALARY
SEX
A
B
C
D
E
W
X
Y
Z
Aaron, Ed
Abbott, Diane
Acosta, Marc
Adams, John
Adams, Robin
Akers, Jan
Alexander, Ed
Alfred, Bob
Allen, Sam
Allen, Troy
Anders, Keith
Anderson, Rob
Anderson, Zach
Angeli, Joe
Archer, Sue
Arnold, Mack
Arnold, Steven
Atkins, Timothy
Wong, James
Wood, Donald
Woods, Manny
Wright, Pam
Wyatt, Charles
Zimmer, Byron

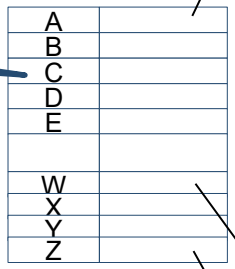

DATA FILE
(PRIMARY KEY FIELD)
NAME
SSN
BIRTHDATE
JOB
SALARY
SEX
AA
AB
AC
AD
AE
WO
WP
ZI
Aaron, Ed
Abbott, Diane
Acosta, Marc
Adams, John
Adams, Robin
Akers, Jan
Alexander, Ed
Alfred, Bob
Allen, Sam
Allen, Troy
Anders, Keith
Anderson, Rob
Anderson, Zach
Angeli, Joe
Archer, Sue
Arnold, Mack
Arnold, Steven
Atkins, Timothy
Wong, James
Wood, Donald
Woods, Manny
Wright, Pam
Wyatt, Charles
Zimmer, Byron


DATA FILE
(PRIMARY KEY FIELD)
NAME
SSN
BIRTHDATE
JOB
SALARY
SEX
Aaron, Ed
Abbott, Diane
Acosta, Marc
Adams, John
Akers, Jan
Alfred, Bob
Allen, Sam
Allen, Troy
Anders, Keith
Anderson, Rob
Anderson, Zach
Angeli, Joe
Archer, Sue
Arnold, Mack
Arnold, Steven
Atkins, Timothy
Wong, James
Wood, Donald
Woods, Manny
Wright, Pam
Wyatt, Charles
Zimmer, Byron
A
B
C
D
E
W
X
Y
Z
불필요한 인덱스 엔트리
데이터 파일에서의 블록 엑세스 증가
문제가 뭐지?
AA
AB
AC
AD
AE
WO
WP
ZI
DATA FILE
(PRIMARY KEY FIELD)
NAME
SSN
BIRTHDATE
JOB
SALARY
SEX
Aaron, Ed
Abbott, Diane
Acosta, Marc
Adams, John
Adams, Robin
Akers, Jan
Alexander, Ed
Alfred, Bob
Allen, Sam
Allen, Troy
Anders, Keith
Anderson, Rob
Anderson, Zach
Angeli, Joe
Archer, Sue
Arnold, Mack
Arnold, Steven
Atkins, Timothy
Wong, James
Wood, Donald
Woods, Manny
Wright, Pam
Wyatt, Charles
Zimmer, Byron

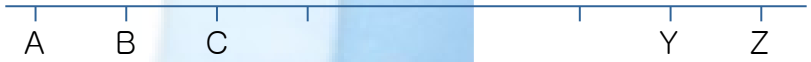




인덱스 엔트리에서 키 필드의 의미?



특정 값이 아니라 값의 범위이다



인덱스 엔트리에서 키 필드의 값은
값의 범위를 표현한 것

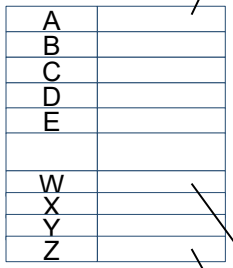
불필요한 인덱스 엔트리

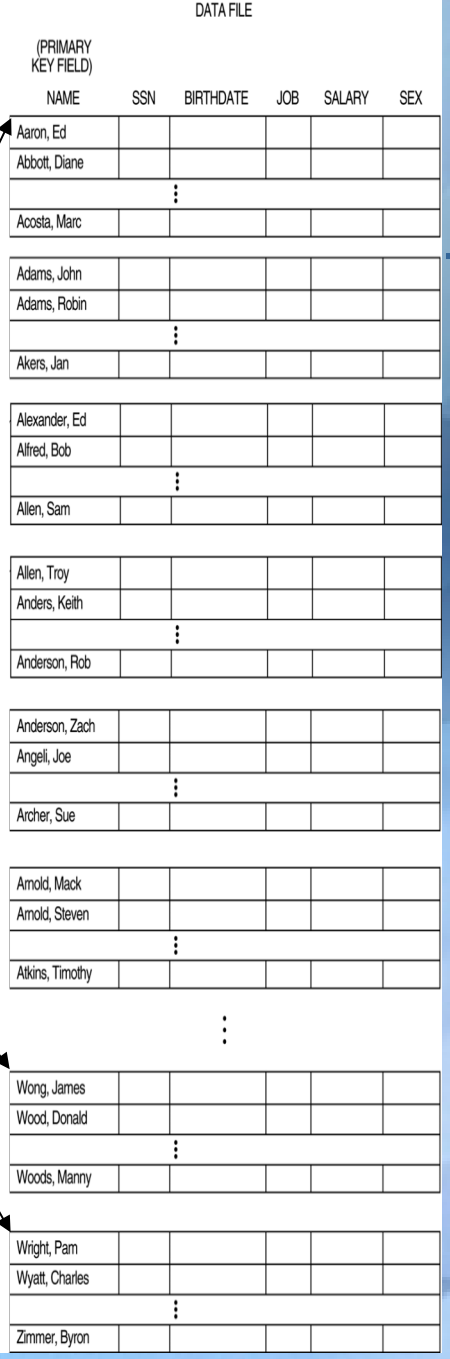
데이터 파일에서의 블록 엑세스 증가

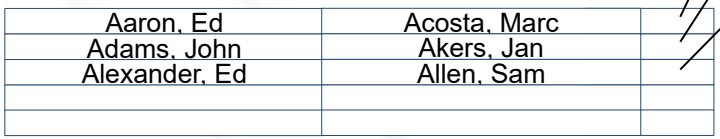
순서 필드 값의 분포 불균형 때문

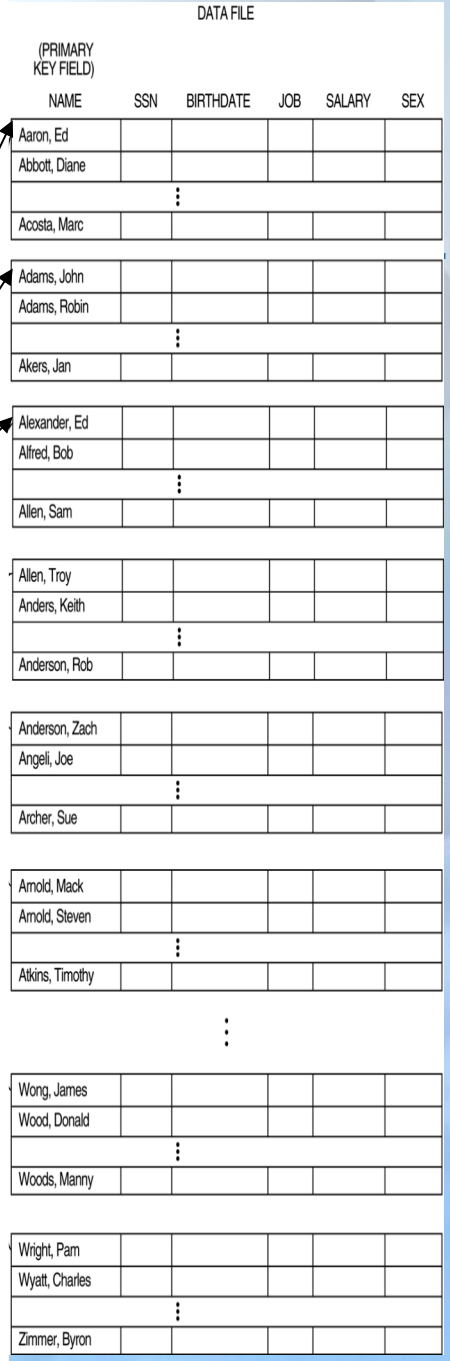
전체 키 범위가 균등하게 분할된 것이 문제

인덱스 엔트리가 표현하는 범위에 블록이
하나씩만 들어가도록 분할할 수 있을까?









문제가 해결되었는가?

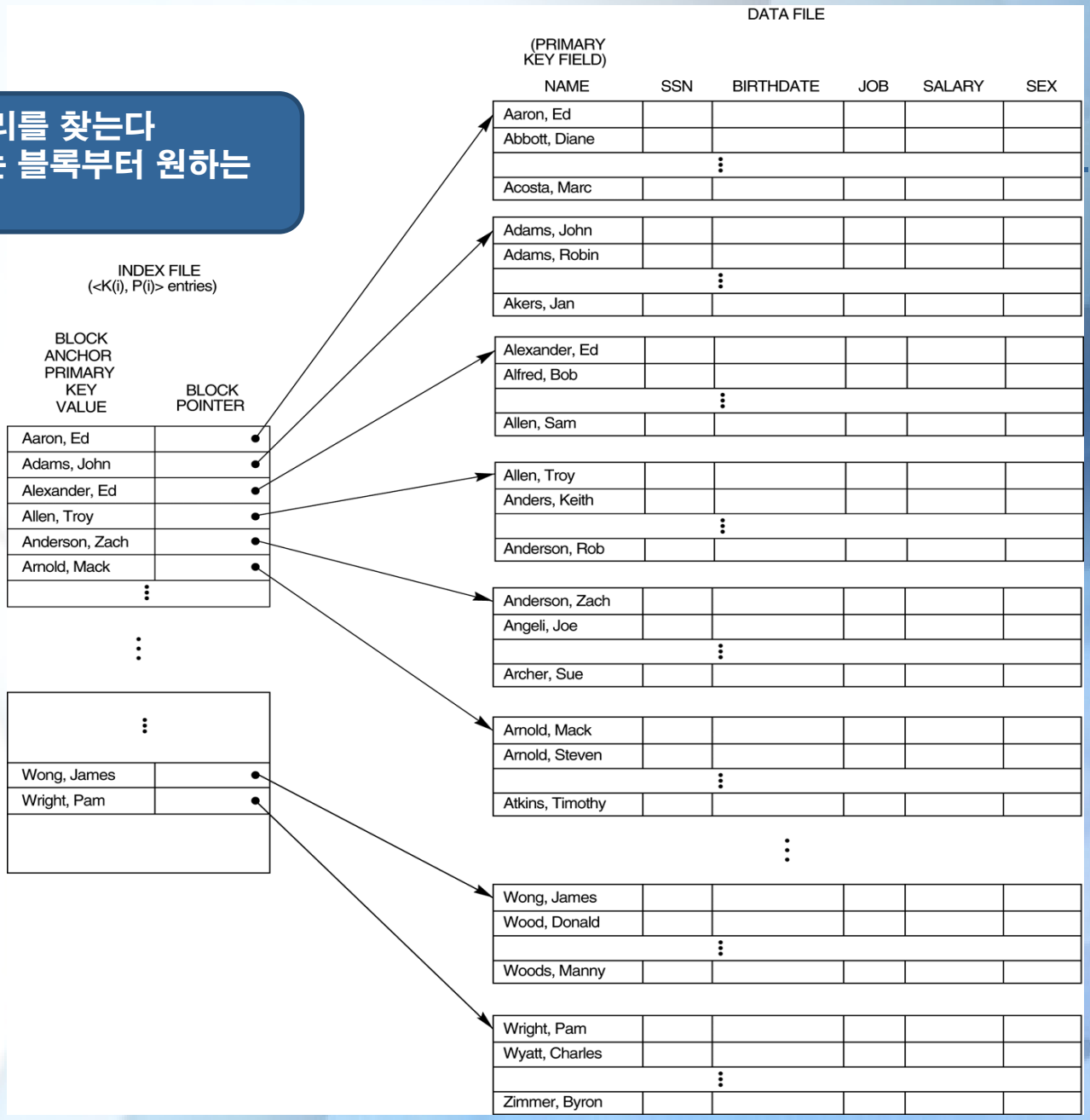
이 방법의 문제는?

인덱스 엔트리가
전 범위를 표현하지 못한다.

검색 알고리즘

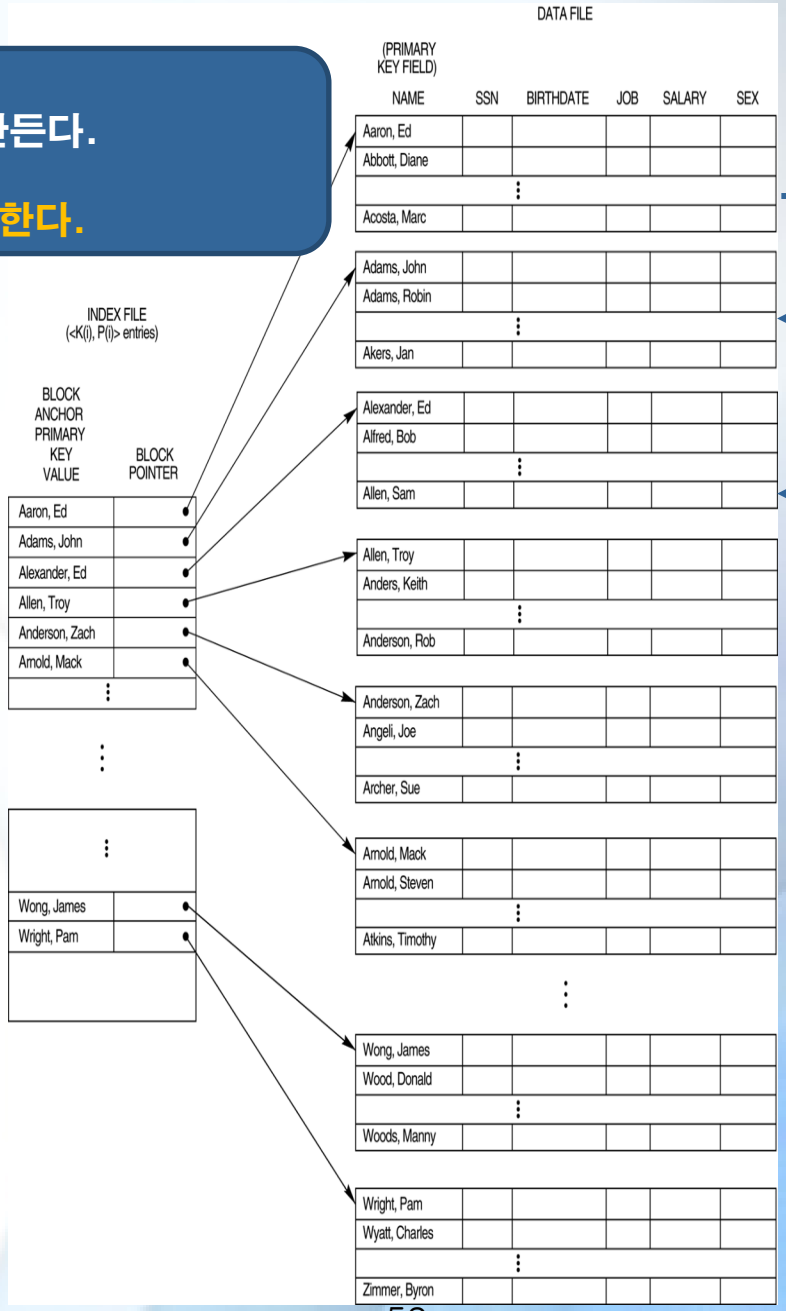
1. 인덱스에서 해당 엔트리를 찾는다
2. 블록포인터가 가리키는 블록부터 원하는 레코드를 찾는다.

순서화일의 순서키 필드에 대한 기본 인덱스



삽입 알고리즘

1. 삽입될 자리를 찾는다
2. 공간이 없으면 공간을 만든다.
3. 삽입한다.
4. 필요하면 인덱스에 반영한다.

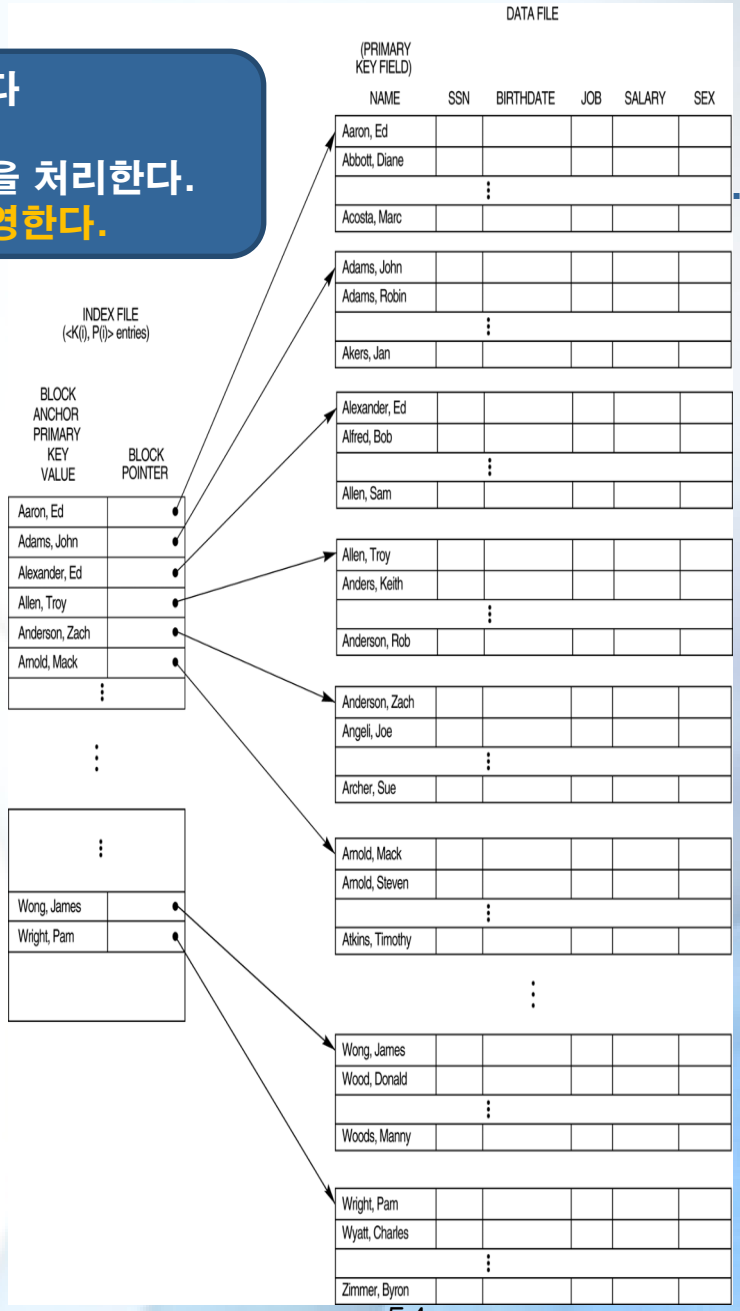


Adams, Sam 삽입

순서화일의 순서키 필드에 대한 기본 인덱스

삭제 알고리즘

1. 삭제할 레코드를 찾는다
2. 삭제한다.
3. 필요하면 삭제한 공간을 처리한다.
4. 필요하면 인덱스에 반영한다.



순서화일의 순서키 필드에 대한 기본 인덱스

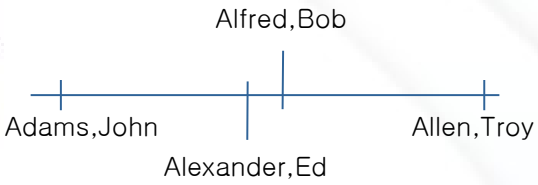
언제 인덱스에 반영하는가?

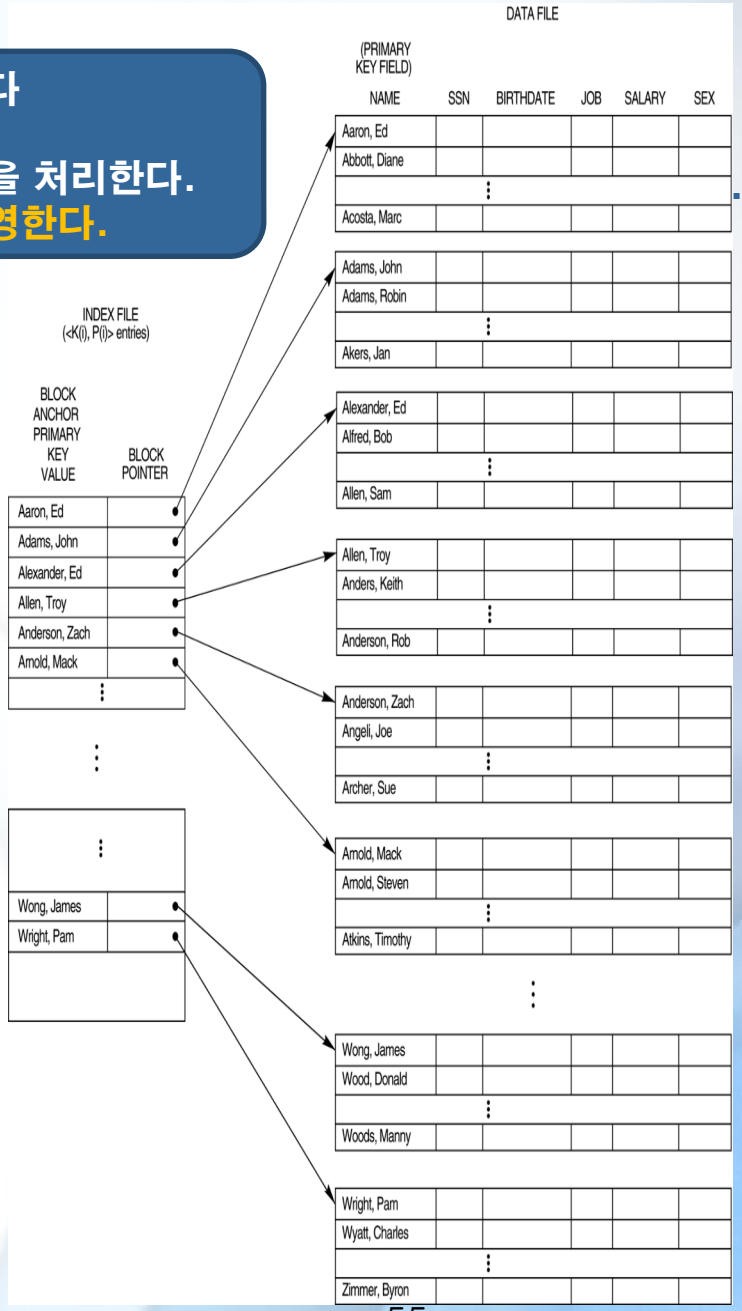
- 병합이 일어났을 때
- 재분배가 일어났을 때
- 블록의 첫번재 레코드가 삭제되었을 때

삭제 알고리즘

1. 삭제할 레코드를 찾는다
2. 삭제한다.
3. 필요하면 삭제한 공간을 처리한다.
4. 필요하면 인덱스에 반영한다.

블록의 첫번재 레코드가 삭제되었을 때??





순서화일의 순서키 필드에 대한 기본 인덱스

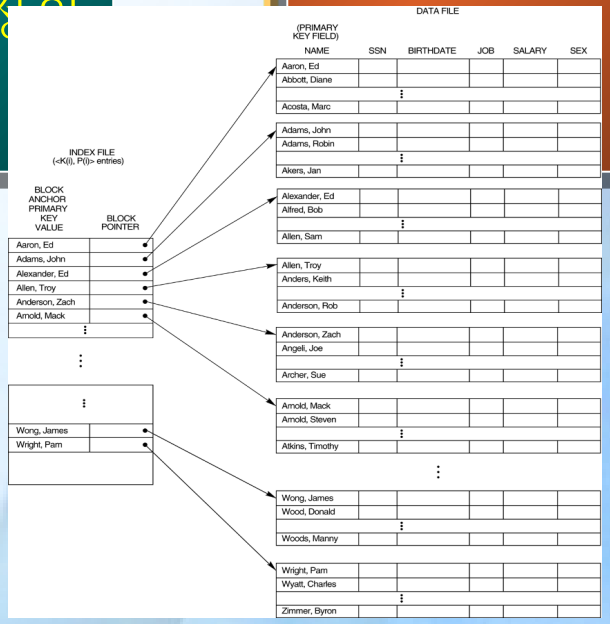
인덱스의 종류

밀집 인덱스

데이터 화일내의 *모든 탐색 키 값*(즉, 모든 레코드)에 대한 인덱스 엔트리를 정의

희소 인덱스

탐색 값의 일부에 대해서만 인덱스 엔트리를 정의



화일의 크기와 성능

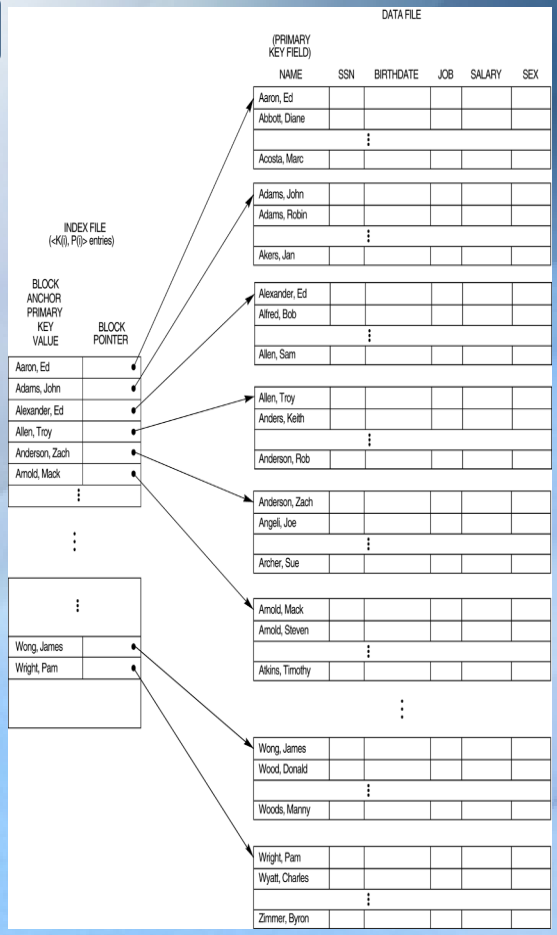
화일의 크기가 클 수록

저장공간 증가

순차검색 시 디스크 액세스 증가

?

밀집인덱스가 좋을까 희소인덱스가 좋을까?



화일의 크기 계산

가정: 각 블록은 레코드로 가득차 있다.

화일의 크기 = 블록의 개수 X 블록의 크기

블록의 개수 = ⌈레코드의 개수/bf⌉

※ bf(blocking factor) : 하나의 블록에 들어갈 수 있는 레코드의 최대 개수

bf = ⌊블록의 크기/레코드의 크기⌋

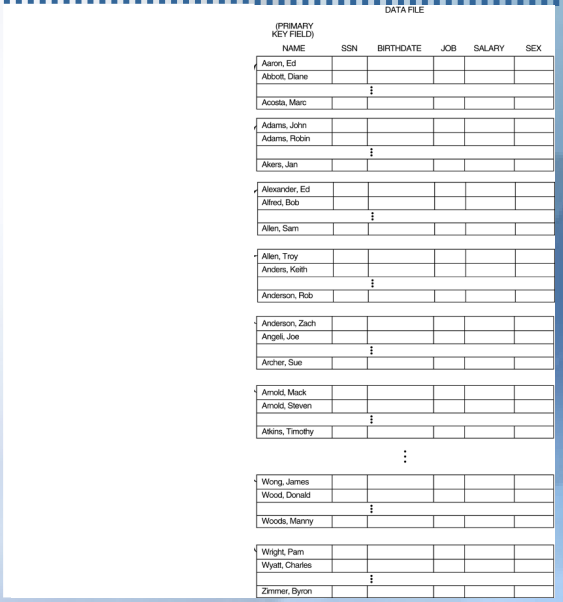
밀집인덱스를 사용한 접근 비용의 예

데이터 파일:

레코드 크기 R=150 바이트
블록 크기 B=512 바이트
레코드 개수 r=30000 레코드

인덱스 화일:

키 필드 크기 VSSN=9 바이트
레코드 포인터 크기 PR=7 바이트



1. 데이터 화일의 Blocking factor는? ⌊512/150⌋
2. 데이터 화일의 블록 갯수는? ⌈30000/1번답□
3. 인덱스 엔트리의 크기는? 9+7
4. 인덱스 화일의 Blocking factor는? ⌊512/3번답□
5. 인덱스 화일의 블록 갯수는? ⌈??/4번답□

단일 단계 인덱스의 유형

기본 인덱스

- 순서화일의 키 필드에 대하여 정의함
- 블록 당 하나의 인덱스엔트리

클러스터링 인덱스

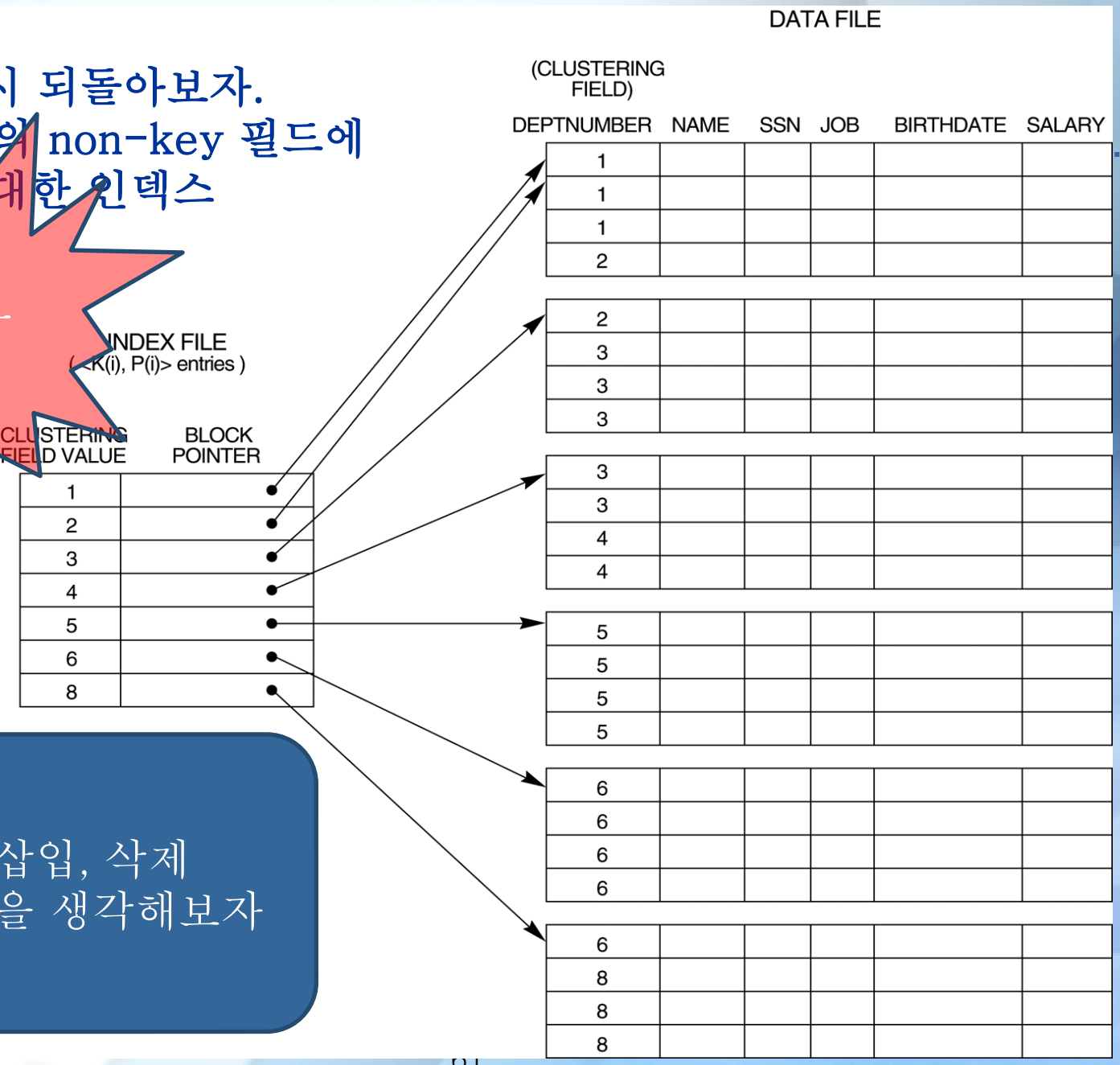
- 순서화일의 non-키 필드에 정의함
- 키 값당 하나의 인덱스 엔트리

보조 인덱스

- 순서화일의 순서 키가 아닌 키 필드에 대한 인덱스

다시 되돌아보자.
순서화일의 non-key 필드에 대한 인덱스

문제가 뭐지?



검색, 삽입, 삭제 알고리즘을 생각해보자

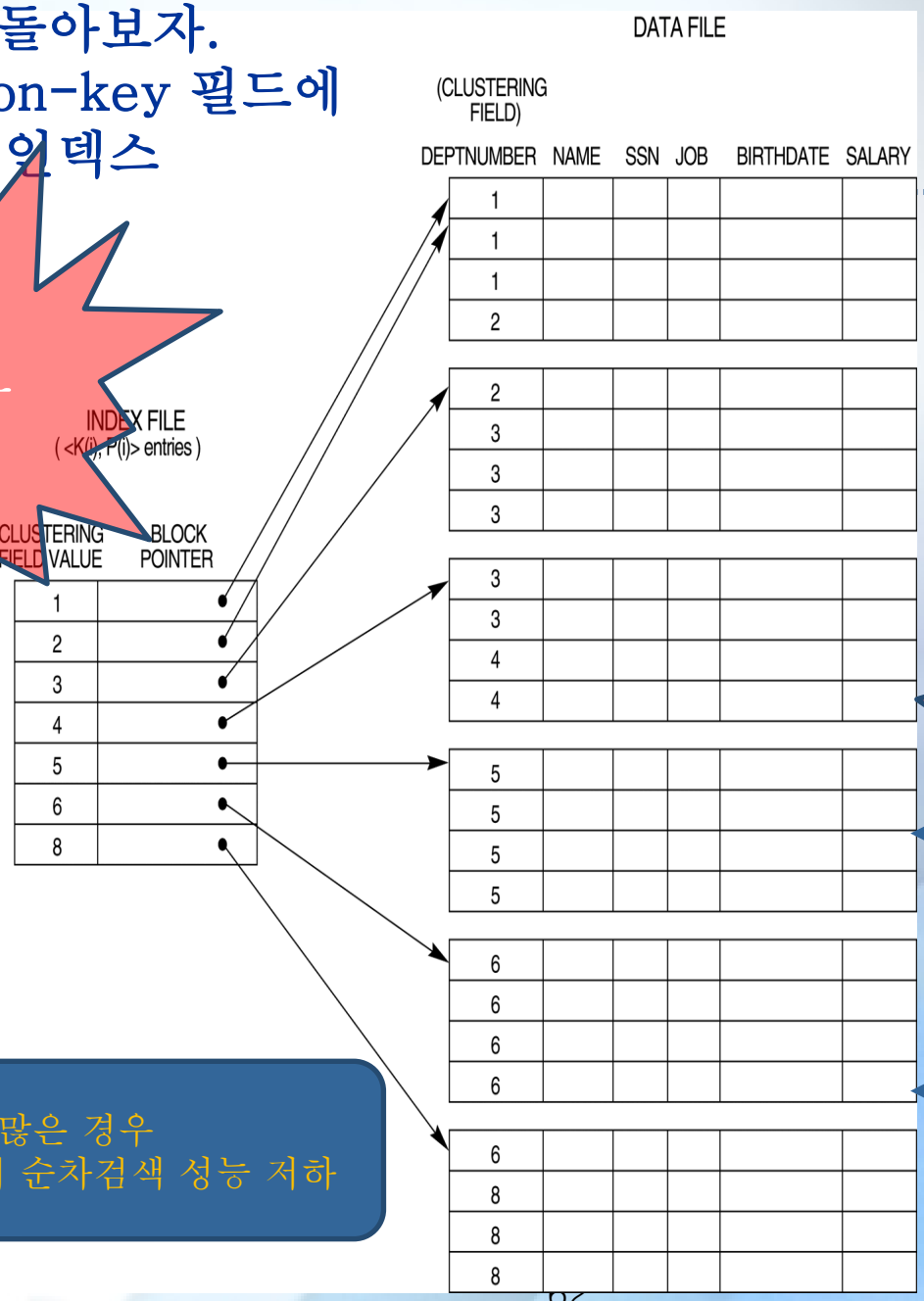
다시 되돌아보자.
순서화일의 non-key 필드에 대한 인덱스

문제가 뭐지?

5가 연속적으로 삽입되면?

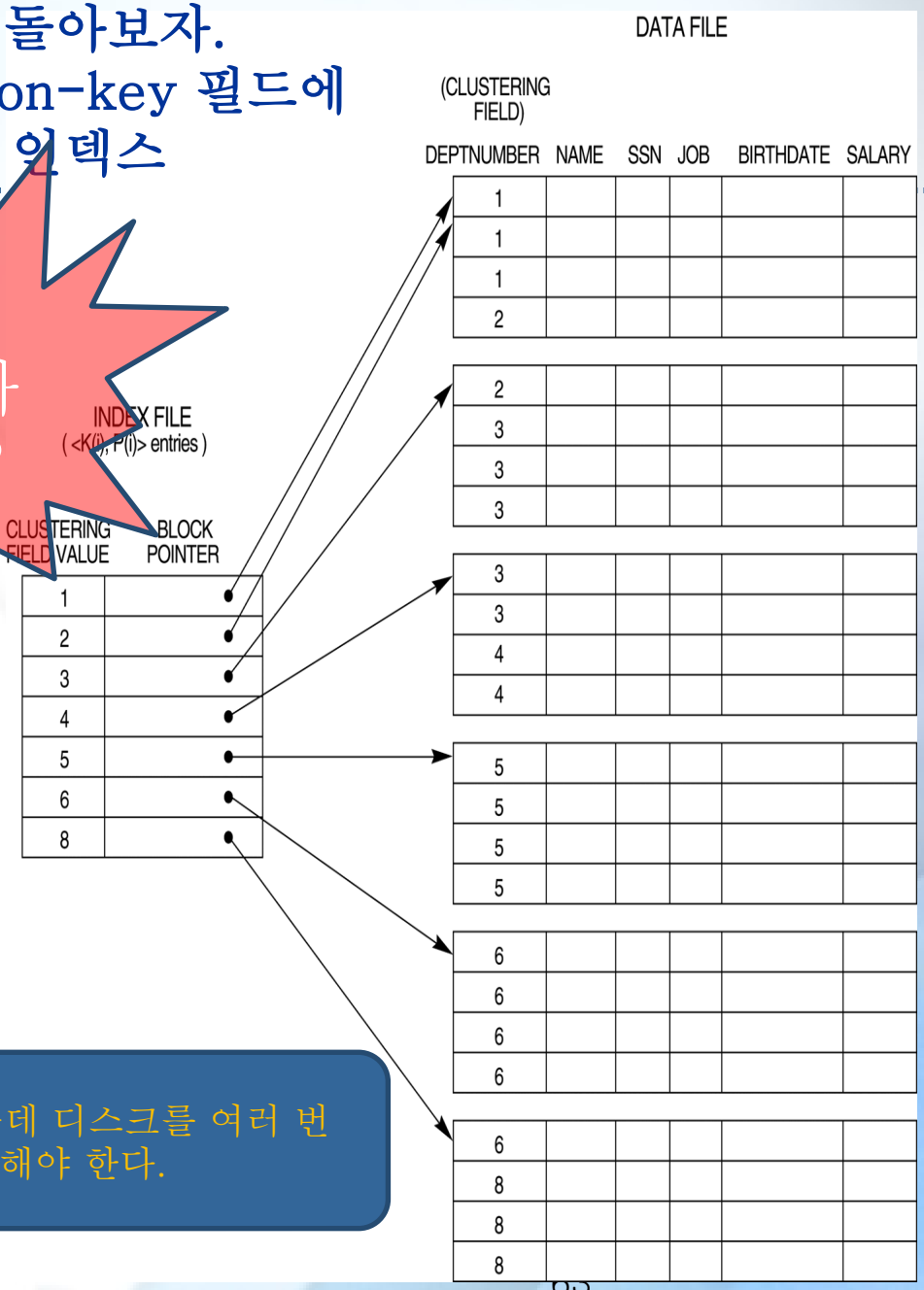
굳이 비워 두어야 하나?

중복이 많은 경우
데이터 화일에서의 순차검색 성능 저하



다시 되돌아보자.
순서화일의 non-key 필드에 대한 인덱스

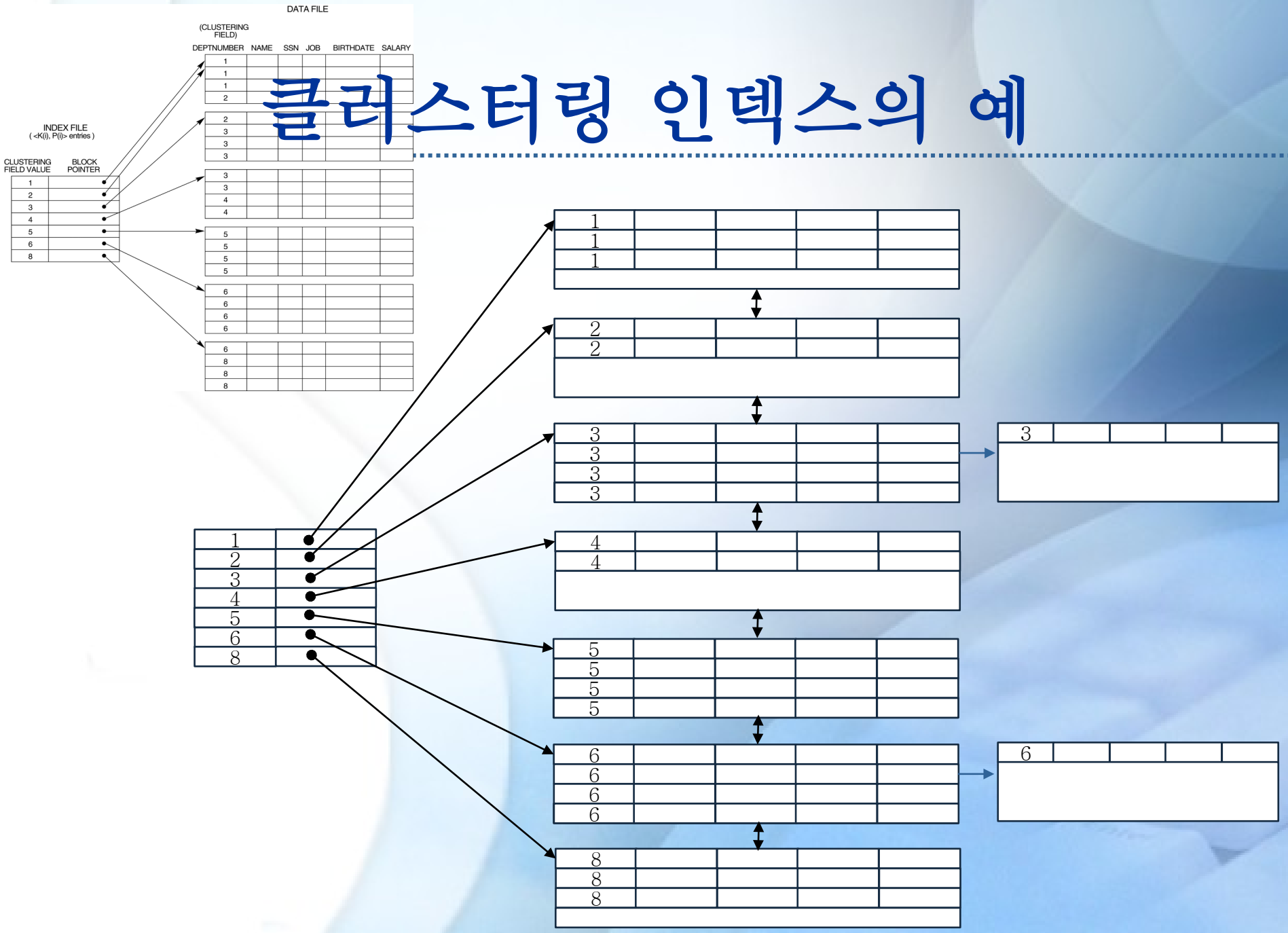
문제가 뭐지?



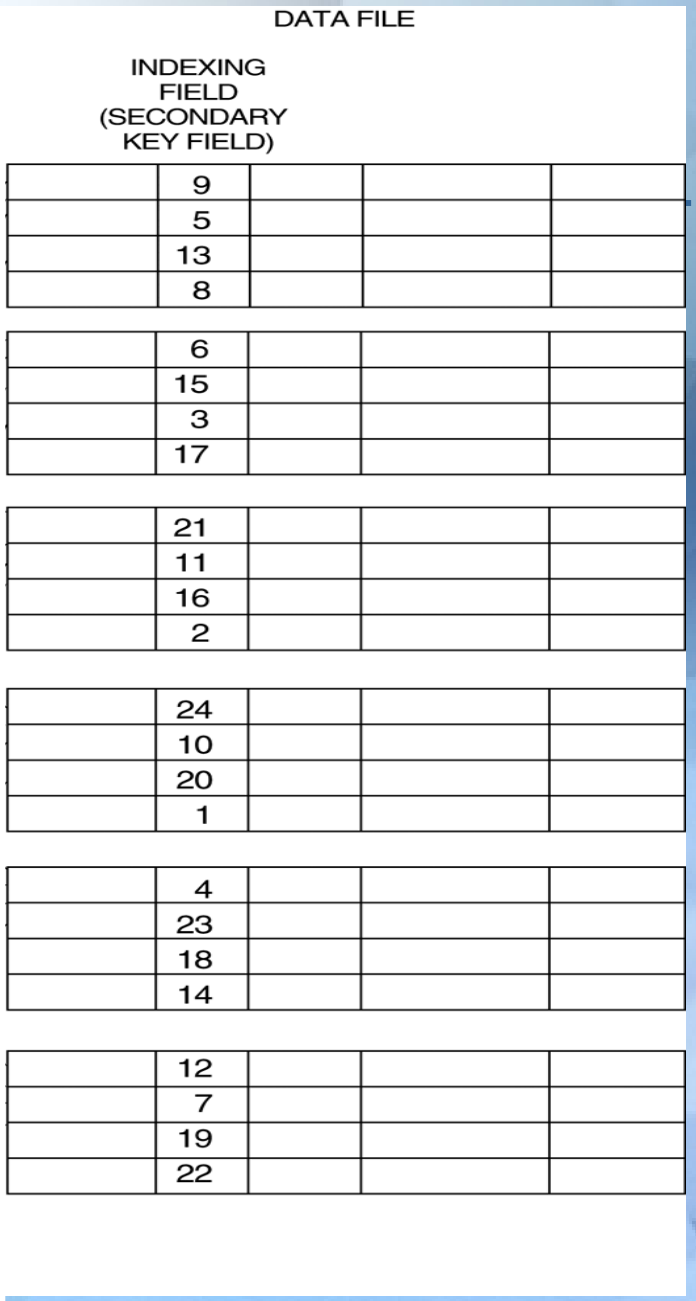
2를 검색하는 경우

중복이 별로 없는데 디스크를 여러 번 액세스해야 한다.

클러스터링 인덱스의 예


DATA FILE
(CLUSTERING FIELD)
DEPTNUMBER
NAME
SSN
JOB
BIRTHDATE
SALARY
INDEX FILE
(<K(i), P(i)> entries)
CLUSTERING FIELD VALUE
BLOCK POINTER

정렬되지 않고,
중복값이 없는 필드에 대한
인덱스를 생각해보자.

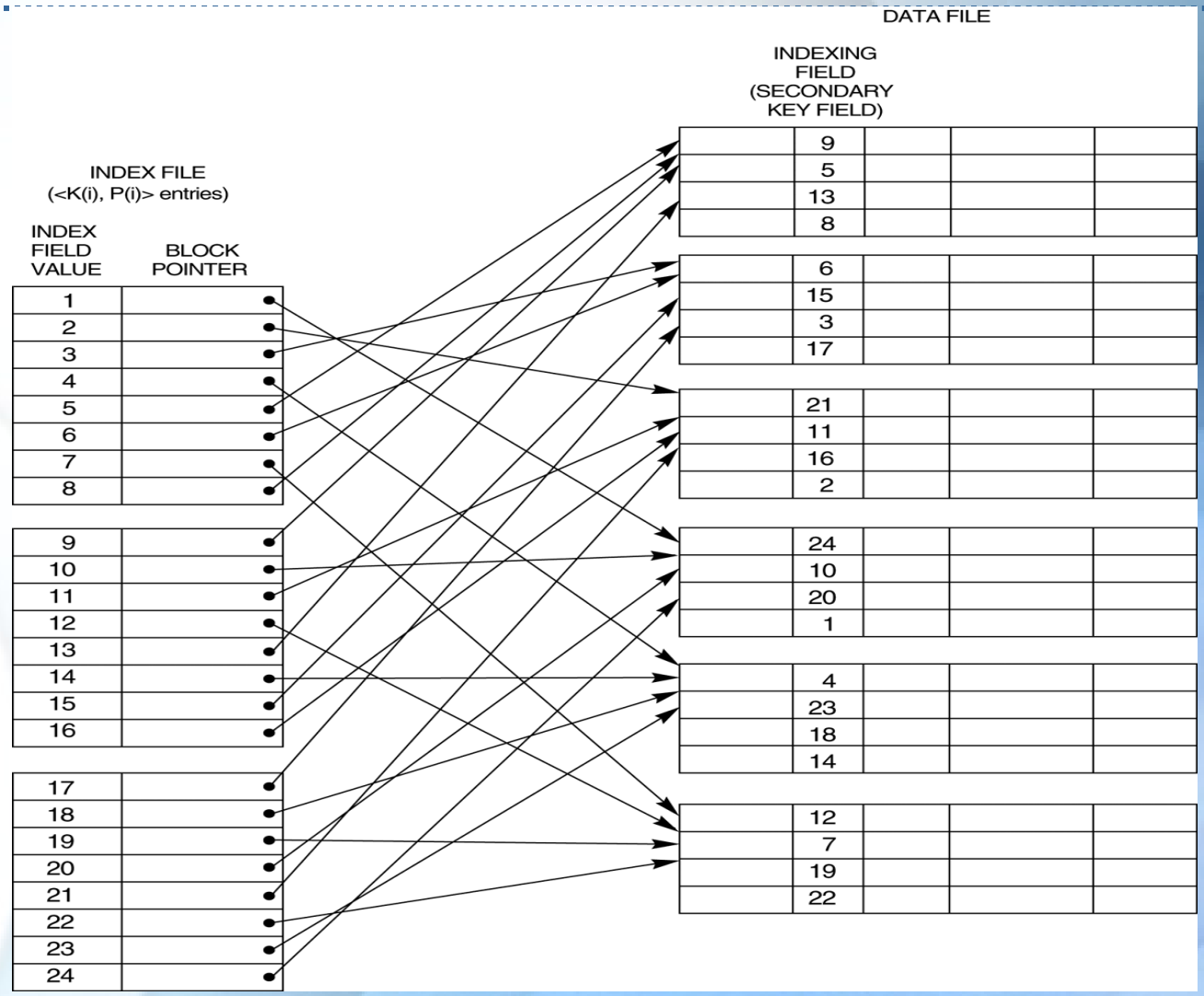


밀집 보조 인덱스의 예
(블록 포인터를 갖는 경우)

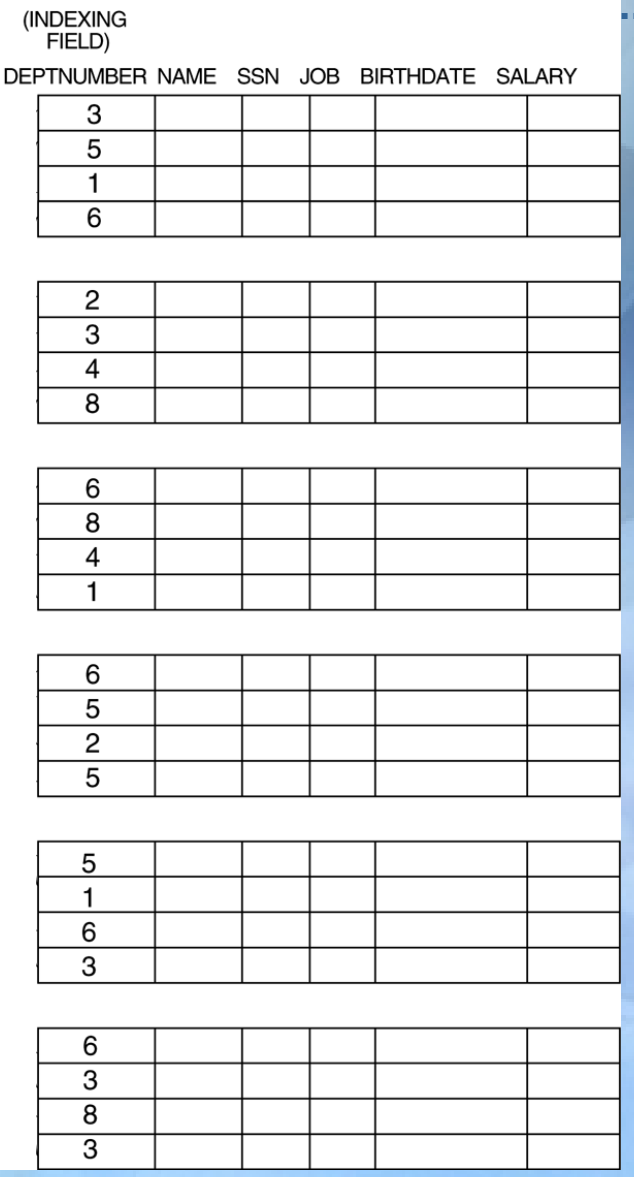
화일의 크기

레코드의 개수
인덱스화일 = 데이터화일

레코드의 크기
인덱스화일 < 데이터화일



정렬되지 않고
중복값을 가지는 필드에 대한
인덱스를 생각해보자.



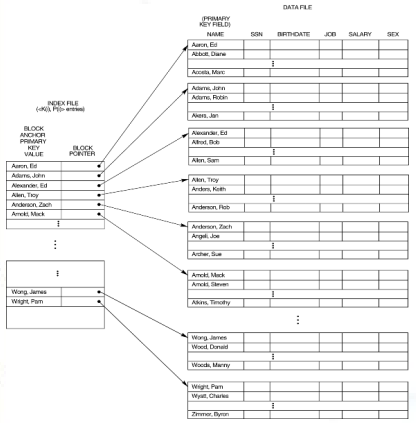
생각해 봅시다.

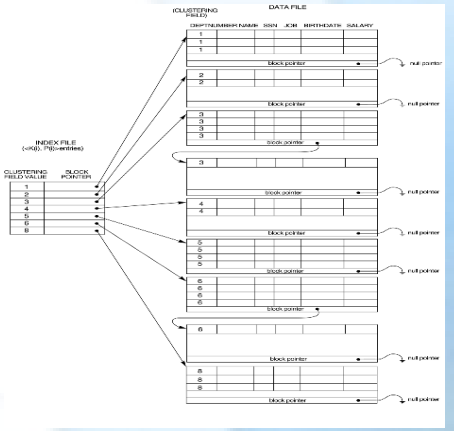
인덱스 엔트리를 보다 빨리
찾을 수 있는 방법은 없을까?

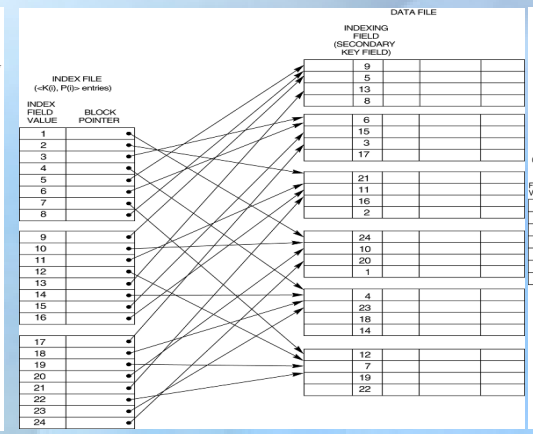
생각해 봅시다.

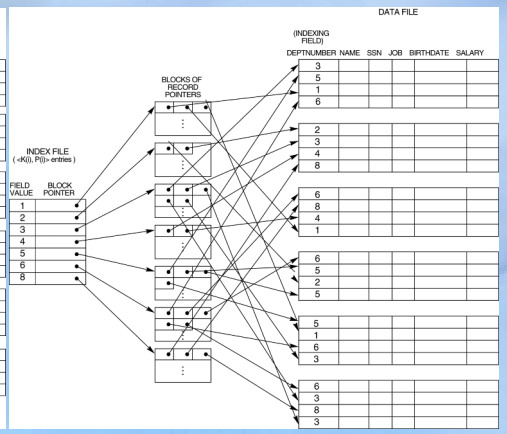
인덱스 엔트리를 보다 빨리
찾을 수 있는 방법은 없을까?

1. 인덱스 엔트리 검색
2. 데이터 레코드 검색









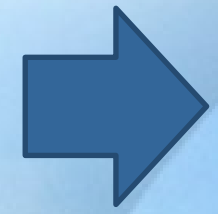
생각해 봅시다.

인덱스 엔트리를 보다 빨리
찾을 수 있는 방법은 없을까?

데이터 화일에서
레코드를 빨리 찾는 법

인덱스를 만든다

인덱스 화일에서
인덱스엔트리를 빨리 찾는 법



인덱스의 인덱스를 만든다

다단계 인덱스

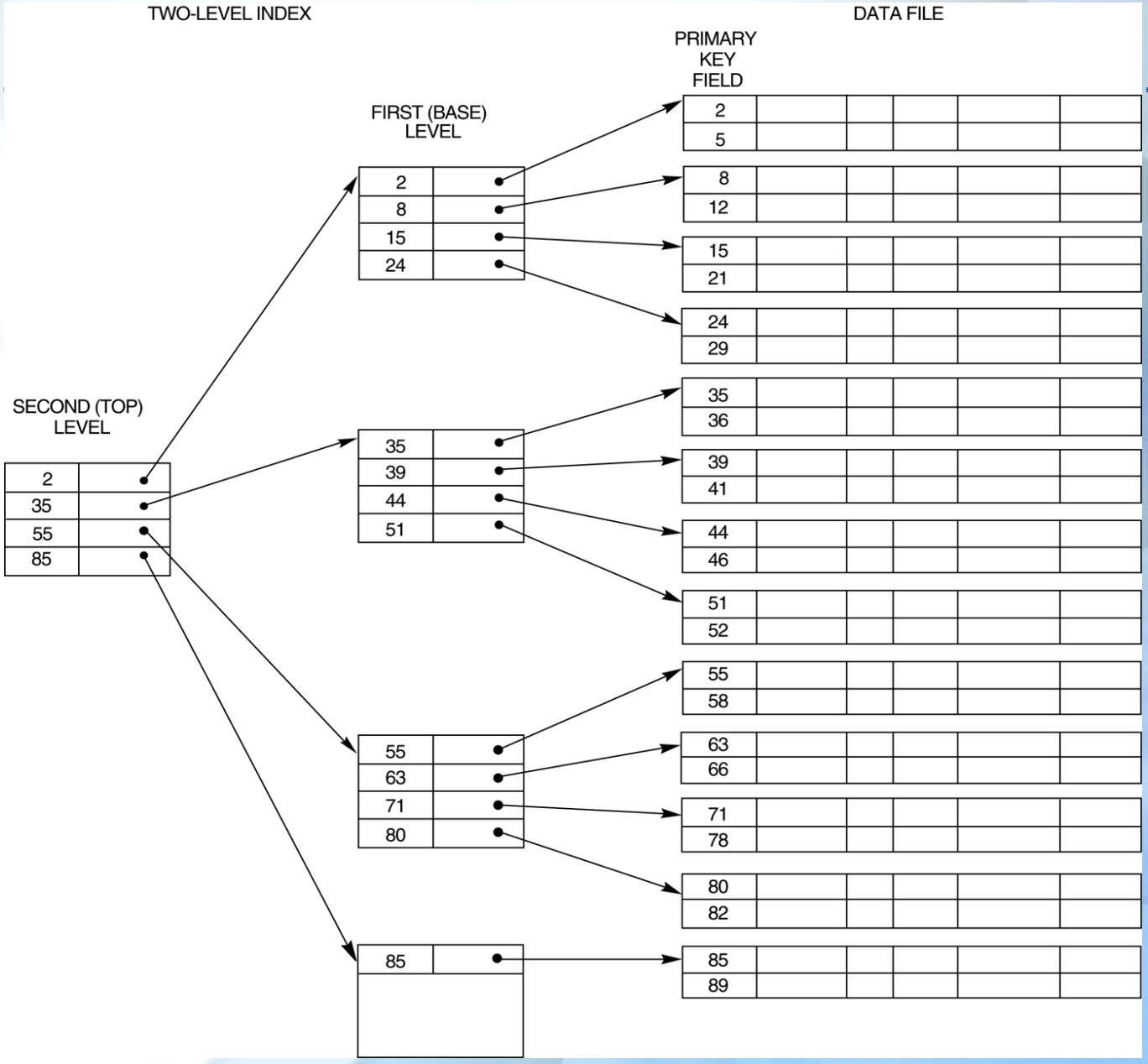
인덱스 화일에 대한 인덱스 화일을 여러 단계로 구성한 인덱스

- 첫번째 단계 인덱스
- 두번째 단계 인덱스
  :
- 최 상위 단계 인덱스

특징

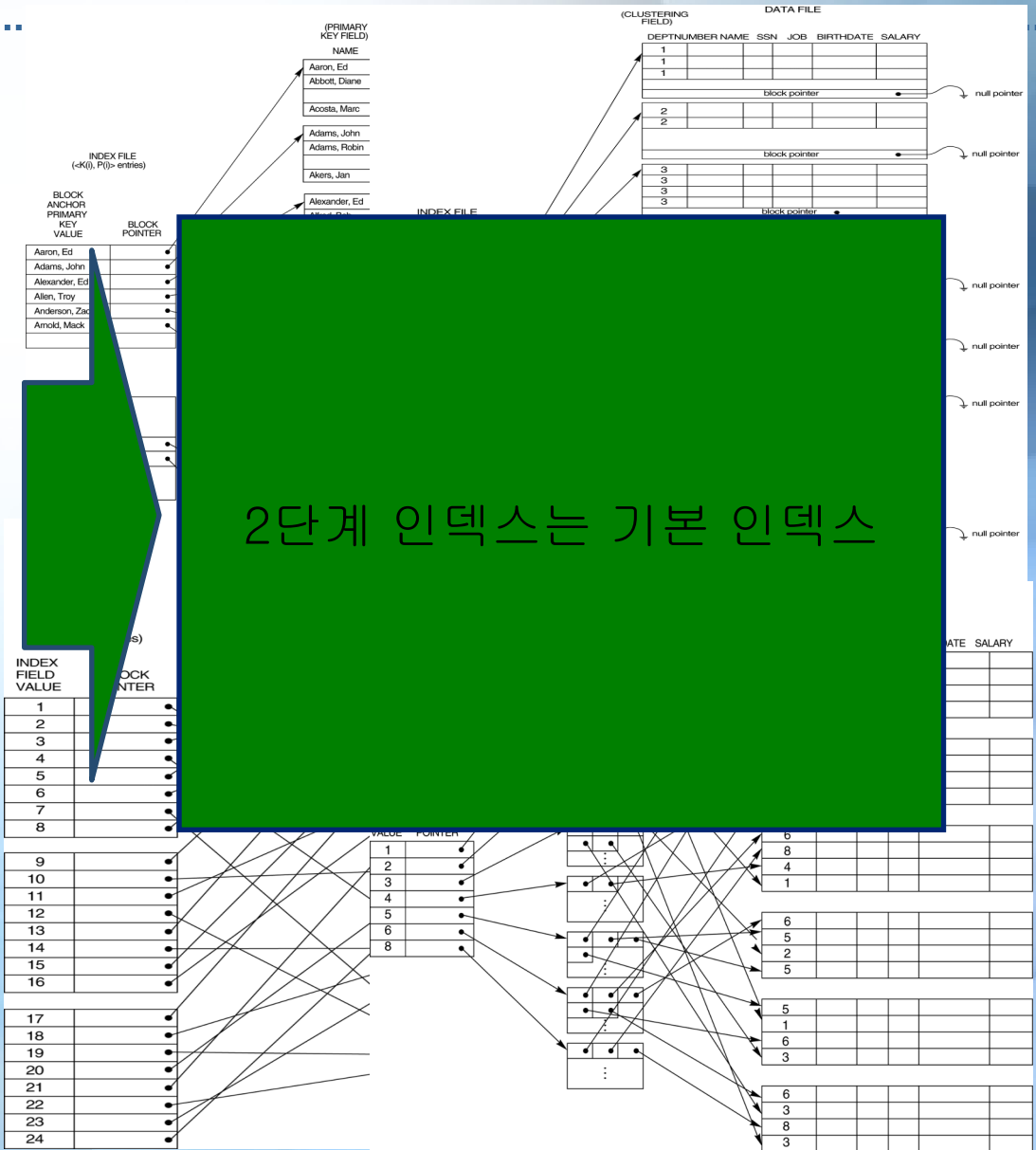
- 탐색트리 형태를 가짐

2-단계 기본 인덱스의 예


TWO-LEVEL INDEX
DATA FILE
PRIMARY KEY FIELD
FIRST (BASE) LEVEL
SECOND (TOP) LEVEL
2
35
55
85
2
8
15
24
35
39
44
51
55
63
71
80
85
2
5
8
12
15
21
24
29
35
36
39
41
44
46
51
52
55
58
63
66
71
78
80
82
85
89

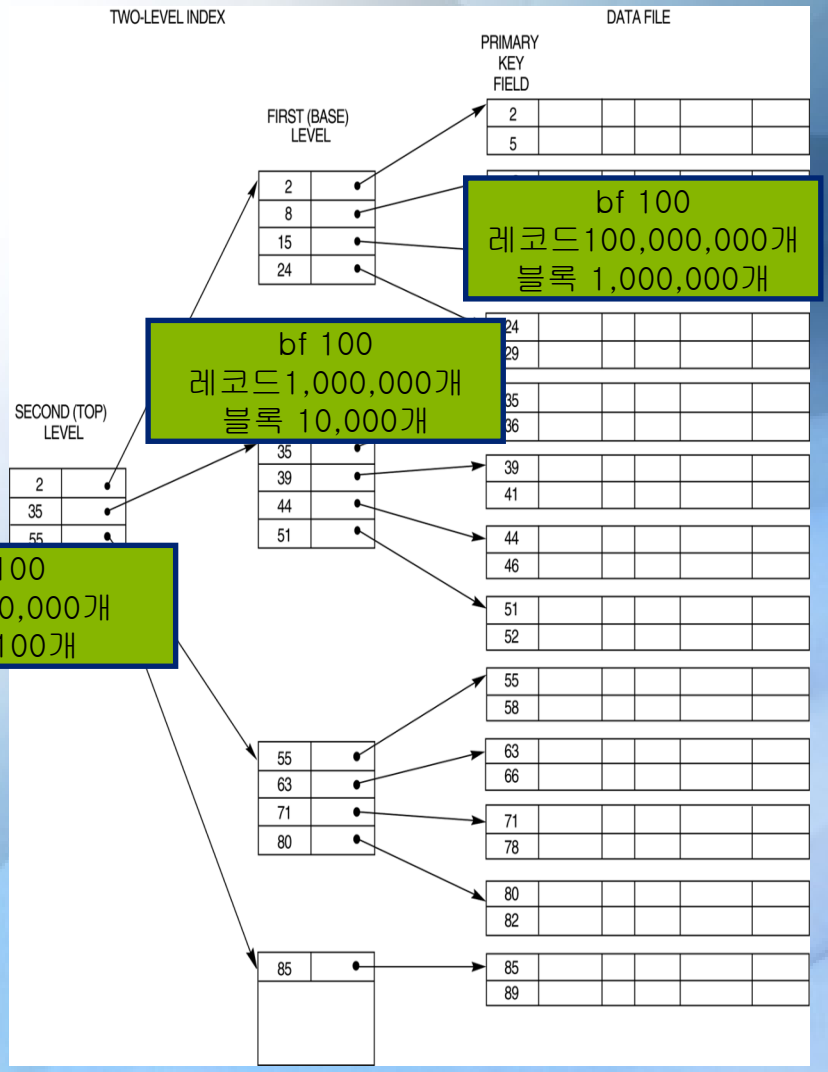
1단계 인덱스의 특징

기본 인덱스 화일	정렬O 중복X
클러스터링 인덱스 화일	정렬O 중복X
첫번째 보조인덱스 화일	정렬O 중복X
두번째 보조인덱스 화일	정렬O 중복X



생각해 봅시다.

다단계 인덱스의 성능은
어느 정도일까?



bf 100
레코드100,000,000개
블록 1,000,000개

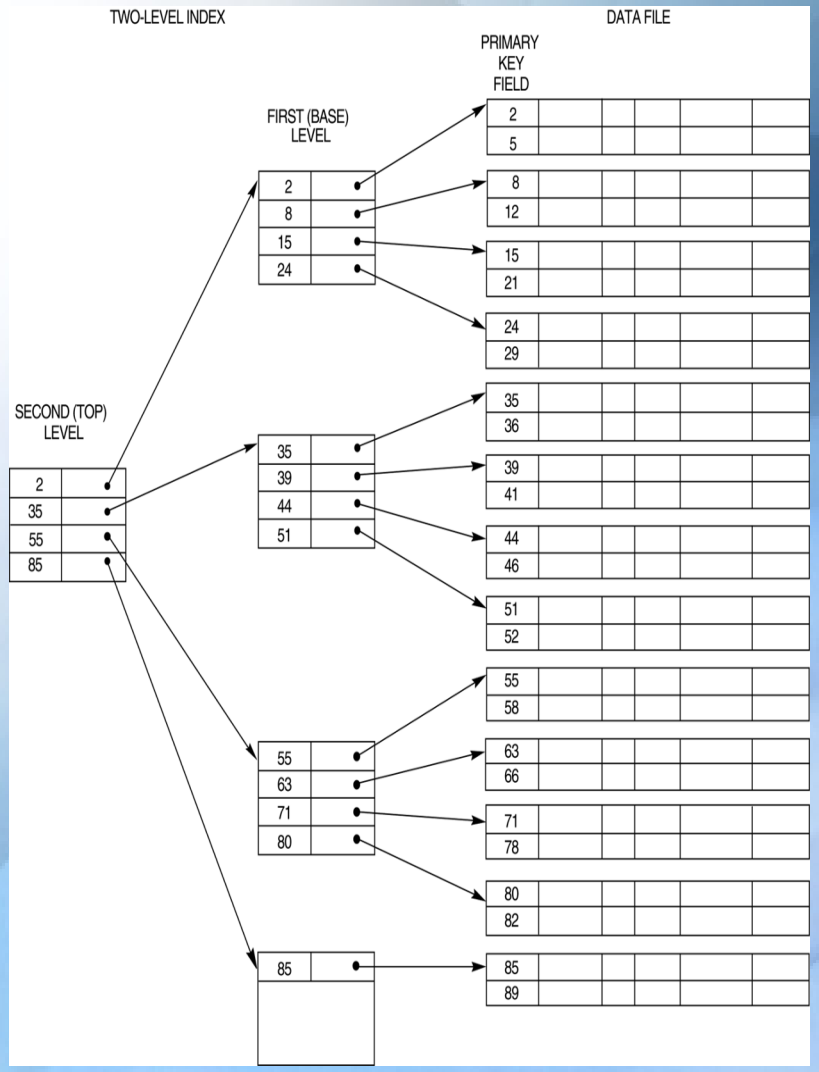
bf 100
레코드1,000,000개
블록 10,000개

bf 100
레코드10,000개
블록 100개

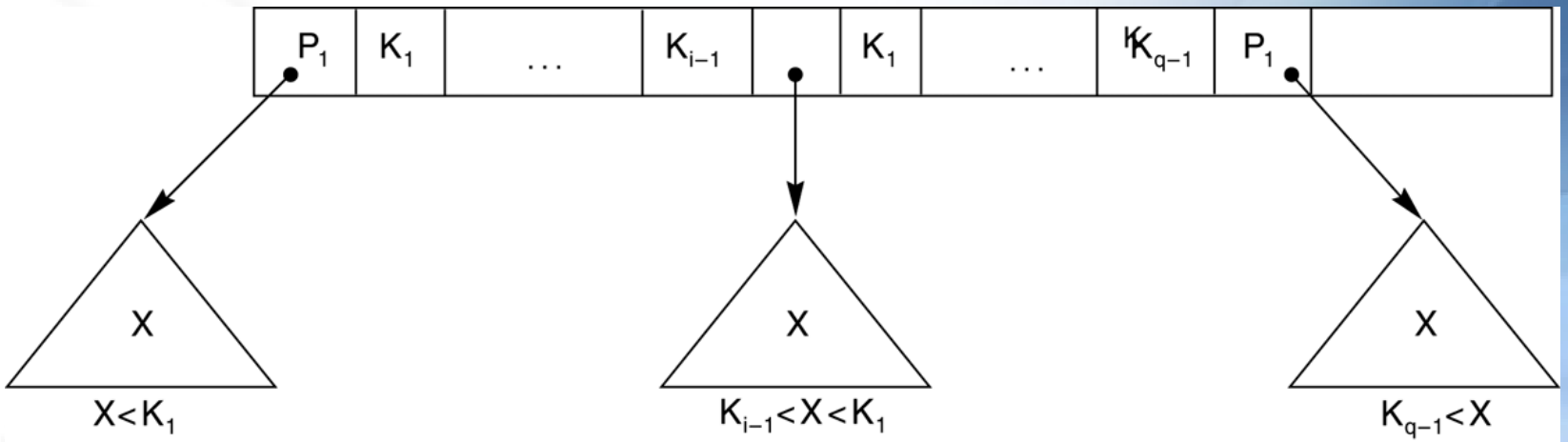
bf 100
레코드100개
블록 1개

생각해 봅시다.

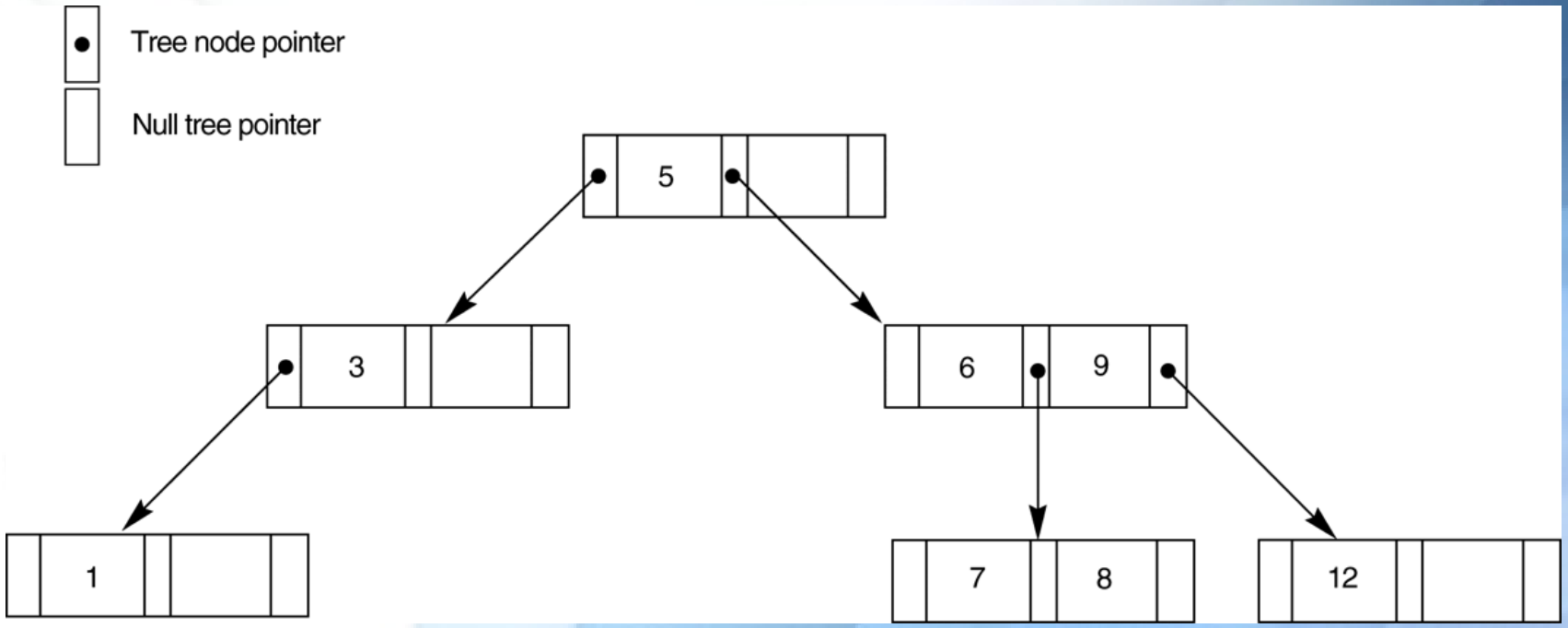
다단계 인덱스 화일에 대한
레코드 연산 방법을 생각해보자.
이 인덱스, 사용할 수 있는가?

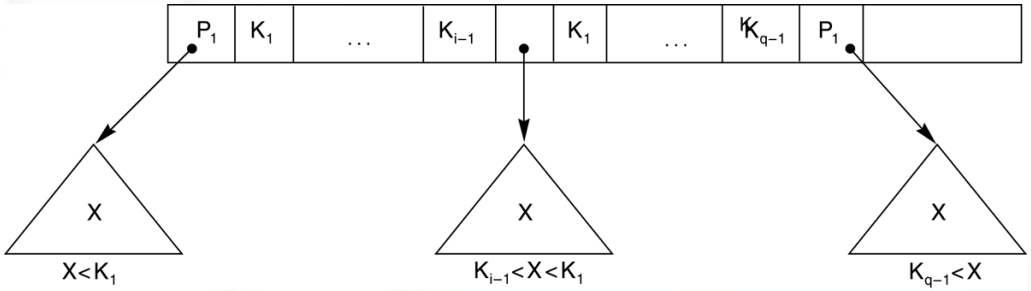


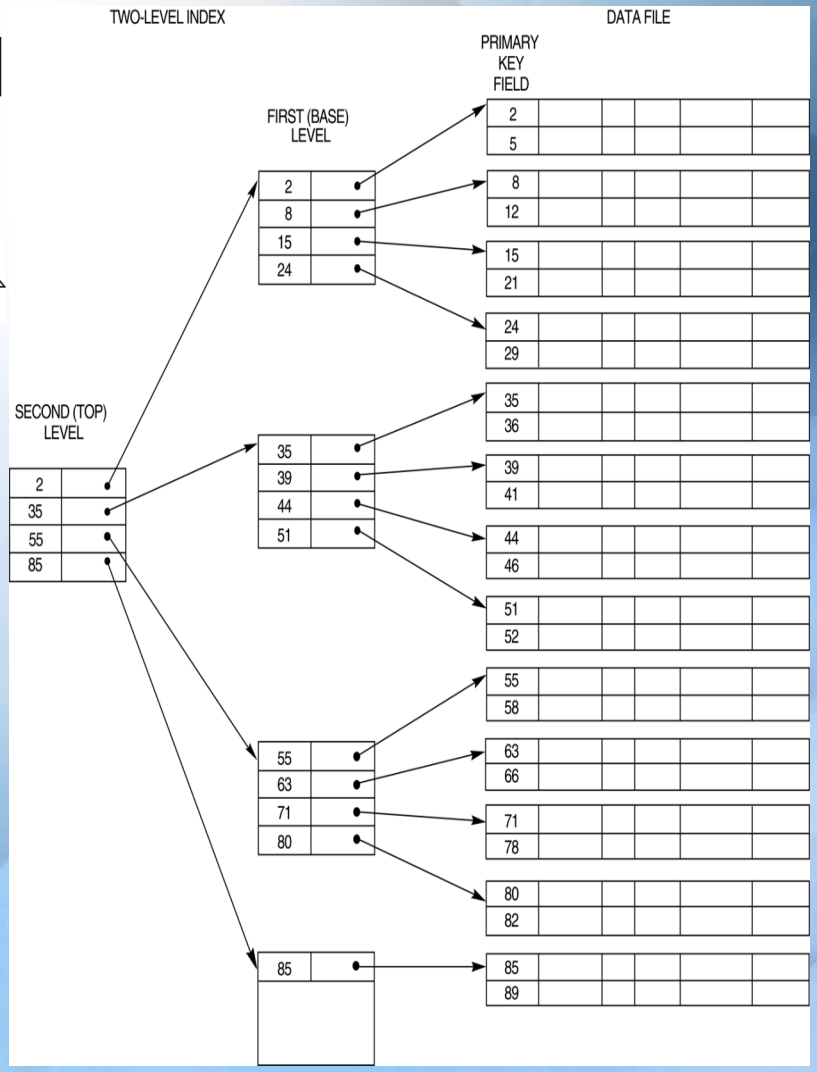
서브트리에 대한 포인터를 갖는 탐색 트리의 한 노드



차수가 p = 3인 탐색 트리




P1
K1
...
Ki−1
K1
...
Kq−1
P1
X
X
X
X<K1
Ki−1<X<K1
Kq−1<X


TWO-LEVEL INDEX
DATA FILE
PRIMARY KEY FIELD
FIRST (BASE) LEVEL
SECOND (TOP) LEVEL
2
35
55
85
2
8
15
24
35
39
44
51
55
63
71
80
85
2
5
8
12
15
21
24
29
35
36
39
41
44
46
51
52
55
58
63
66
71
78
80
82
85
89

B 트리 또는 B+ 트리

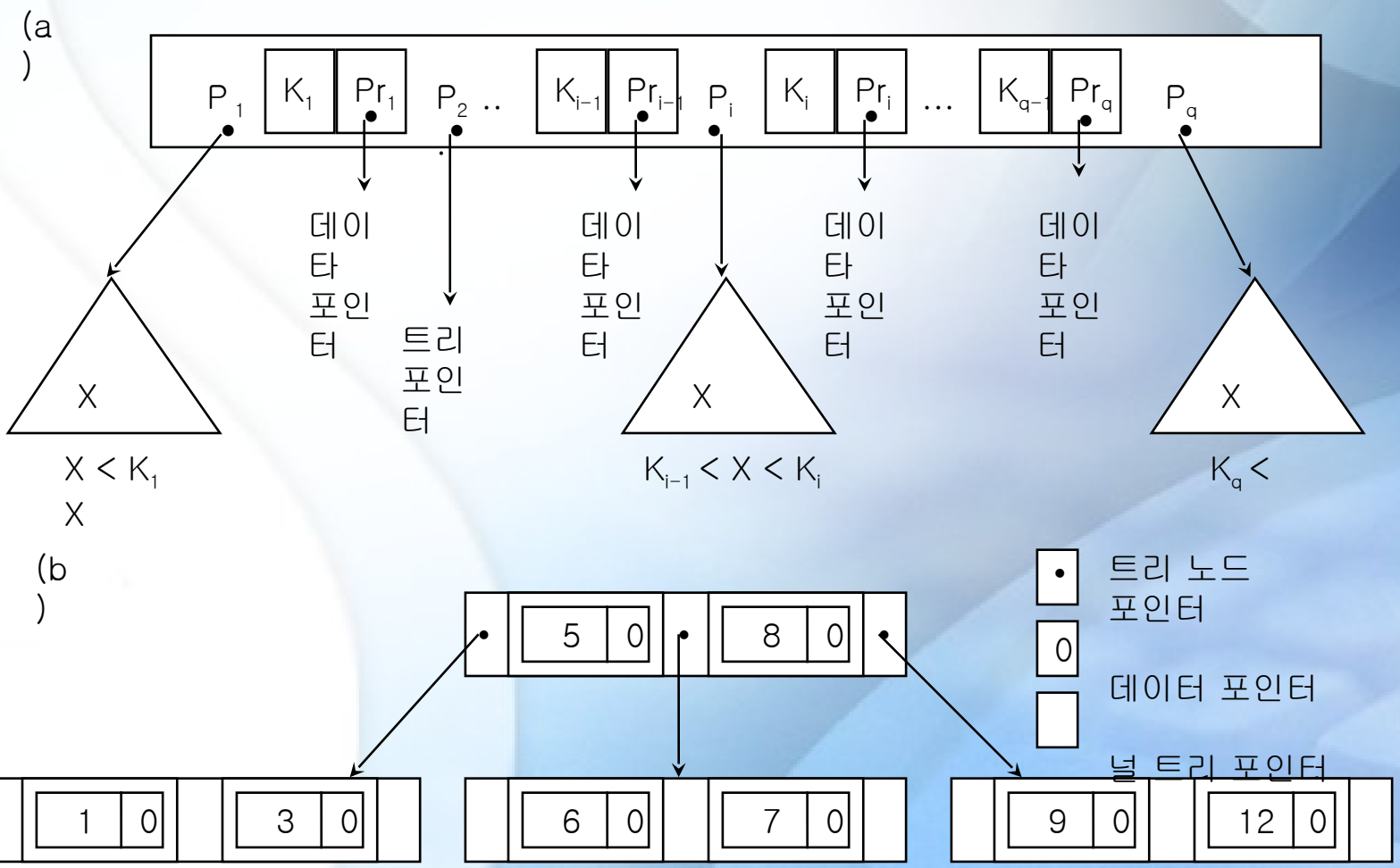
1. 노드에 여분의 공간을 허용(50%)
2. 삽입, 삭제 시 분할 및 병합 발생 가능
3. 분할, 병합은 상위 노드로 전파 가능
4. B+ 트리는 리프에만 데이터 포인터 존재

B 트리 알고리즘

- 트리는 단계 0에서 하나의 루트노드로 시작한다. 루트노드가 가득 찬 경우 트리에 다른 엔트리를 삽입하고자 하면 루트노드를 단계 1의 두개의 노드들로 분할한다. 중간값만이 루트노드에 남아 있고 다른 값들은 두 노드들에 균등하게 나누어 저장한다.
- 비루트 노드(nonroot node)가 가득 찬 경우 새로운 엔트리를 삽입하면, 그 노드는 같은 단계의 두 노드들로 분할하여 저장하고 중간값은 분할된 노드들에 대한 두개의 포인터들을 갖는 부모노드로 옮긴다.
- 이런 분할 과정은 루트노드까지 계속할 수 있으며, 루트노드를 분할하면 새로운 단계가 생성된다.
- 하나의 값을 삭제할 때 노드의 반 이상이 비게 되면 두 이웃노드(neighboring node)끼리 결합한다. 이러한 과정도 루트까지 계속할 수 있다.

B 트리 구조

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



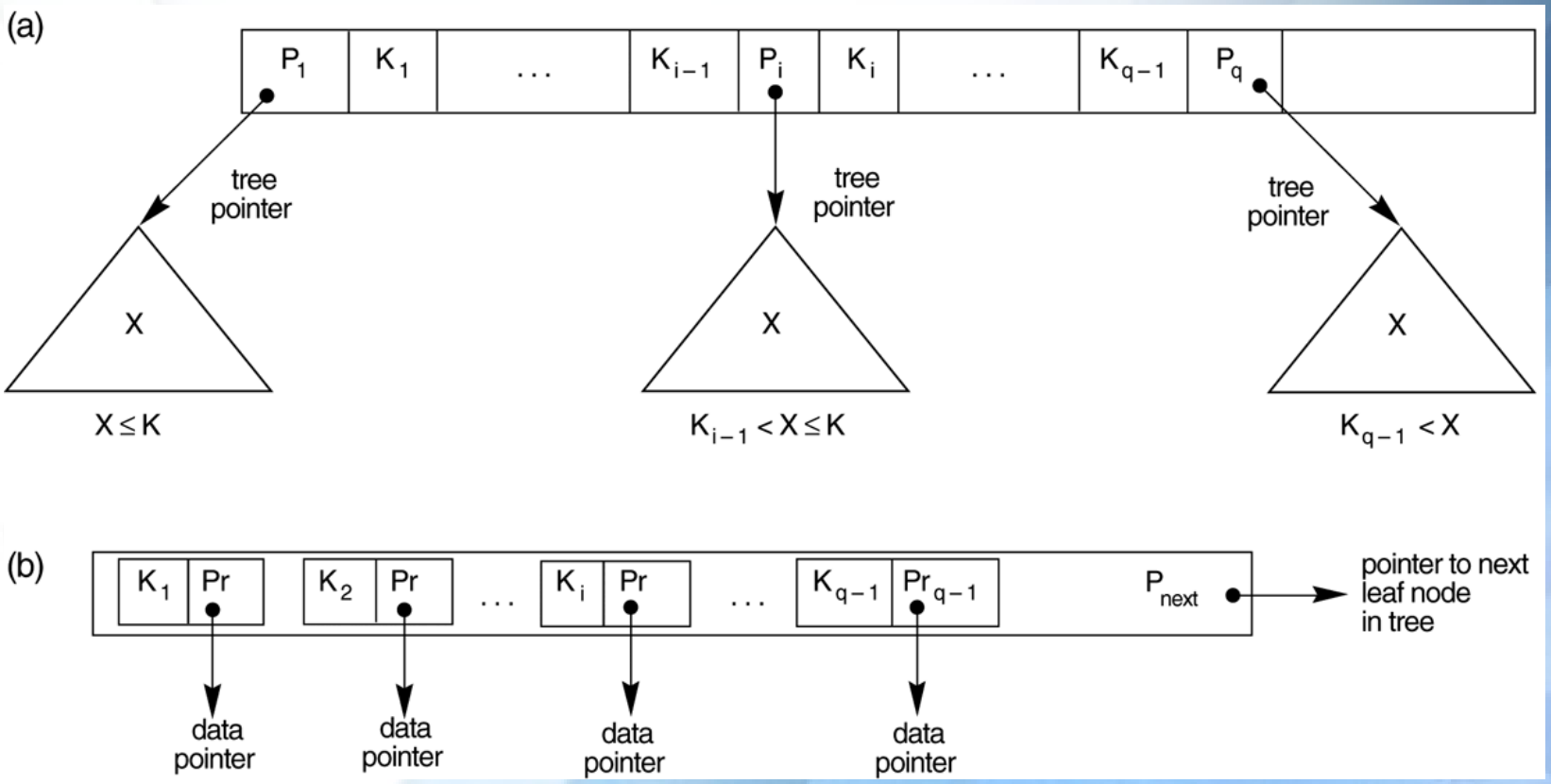
B+ 트리

1. 데이타화일에 대한 포인터가 단말노드에만 존재
2. 단말 노드들은 포인터로 연결

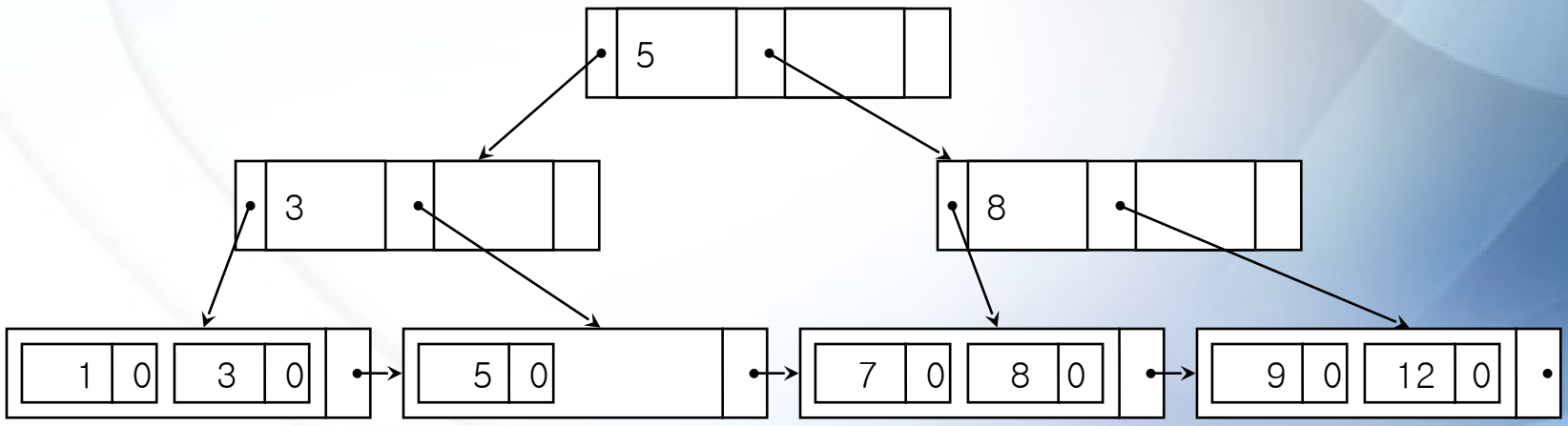
B+ 트리의 노드

(a) q – 1개의 탐색값을 갖는 내부 노드

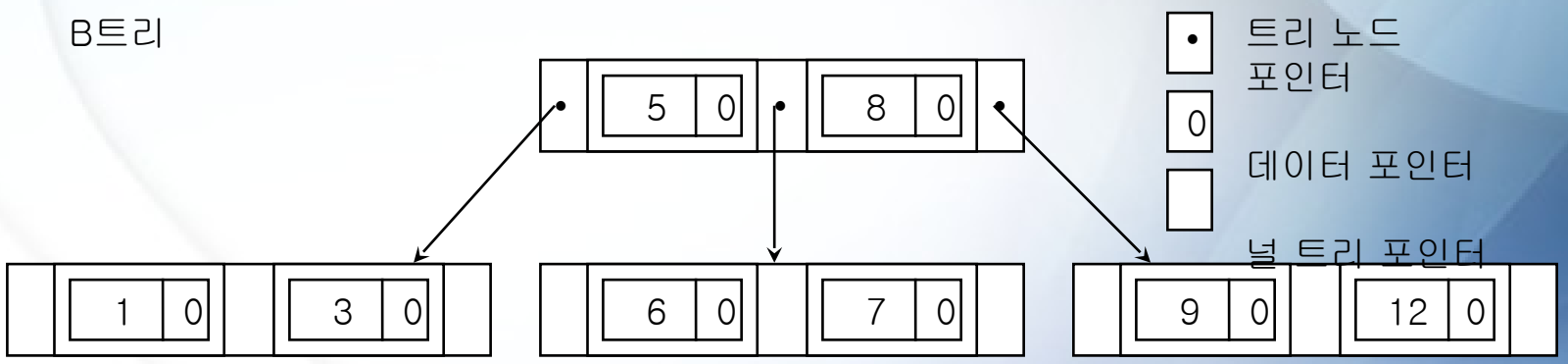
(b) q – 1의 탐색값과 q – 1의 데이터 포인터를 가지는 B+ –트리의 단말 노드

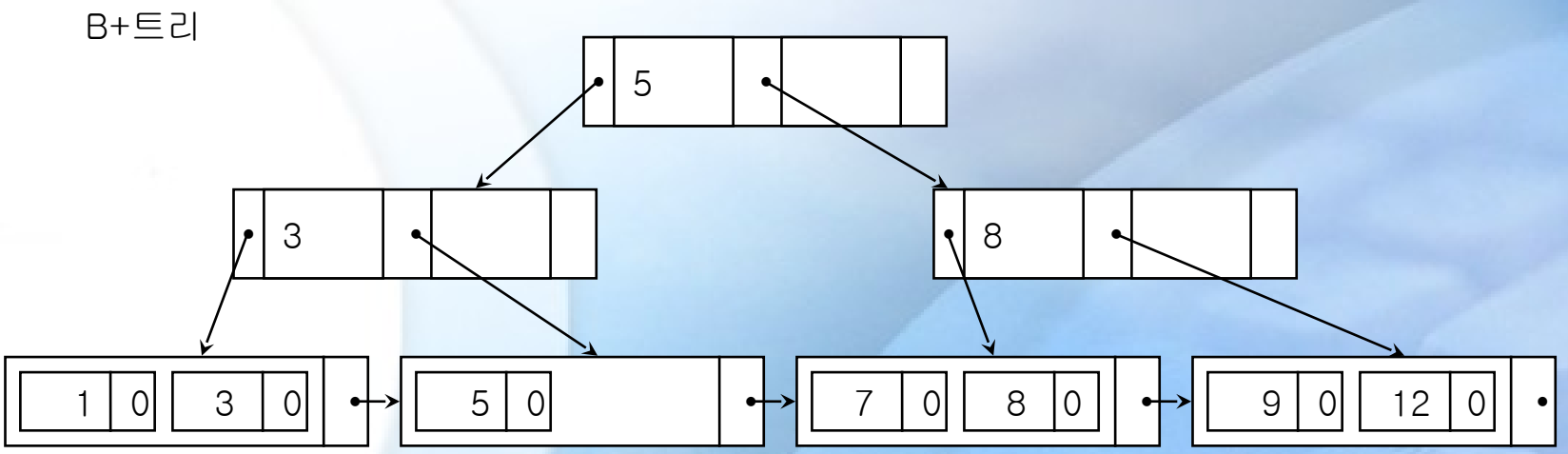


B+ 트리의 예



B 트리와 B+ 트리

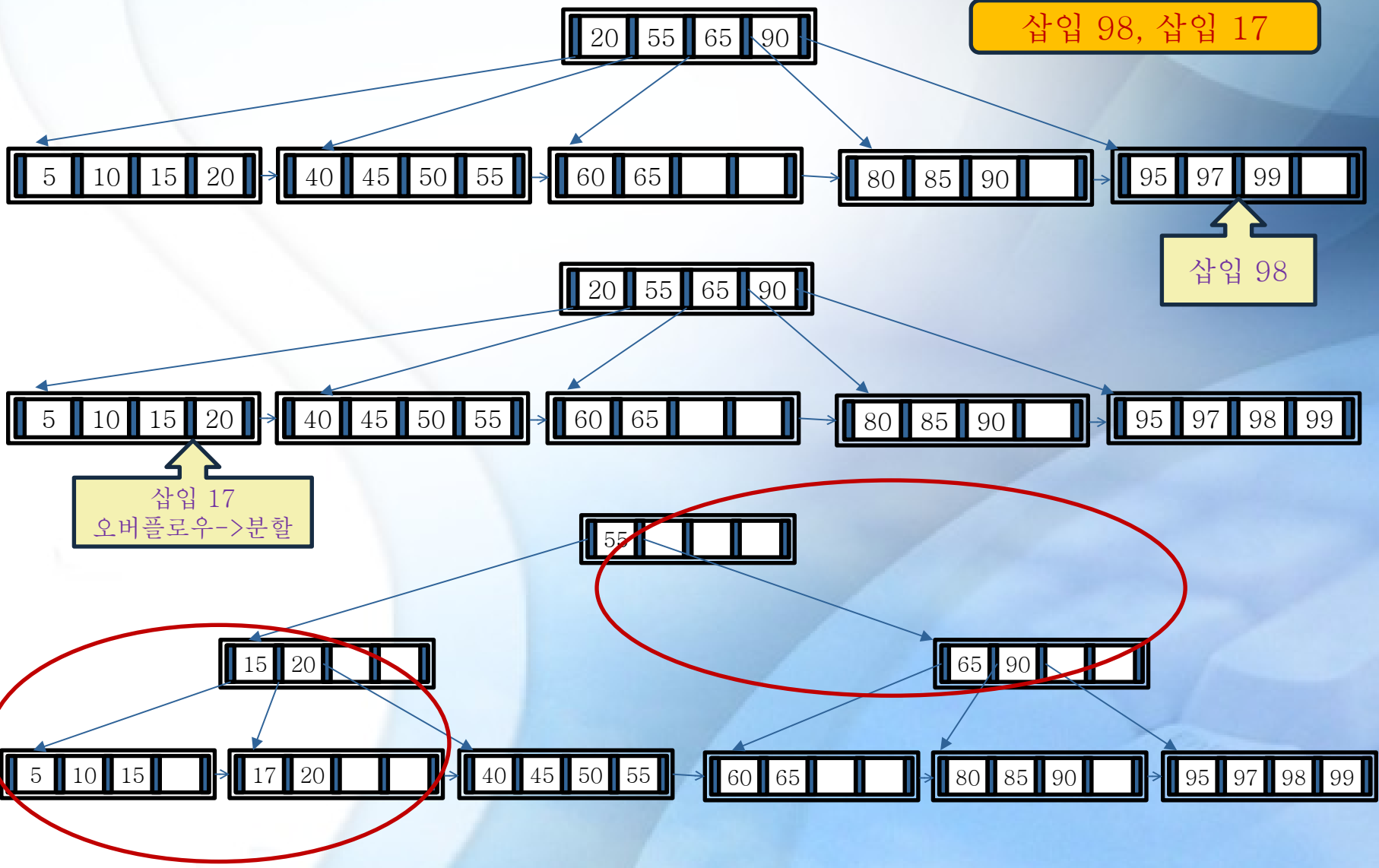

B트리
트리 노드
포인터
0
데이터 포인터
널 트리 포인터
5 0
8 0
1 0
3 0
6 0
7 0
9 0
12 0


B+트리
5
3
8
1 0
3 0
5 0
7 0
8 0
9 0
12 0

B+ 트리 삽입 알고리즘

- 가득 찬 단말노드에 새로운 엔트리를 삽입하고자 할 때, 그 노드를 두 노드로 분할한다.
- 먼저 원래 노드(original node)에 인덱스 엔트리들 중 절반을 새로운 단말노드로 옮긴다.
- 부모 내부노드에는 분할된 인덱스 엔트리 중 가운데 엔트리의 탐색값과 새로 생성된 노드에 대한 포인터의 쌍을 삽입한다.
- 이때 만약 부모 내부노드가 가득 차 있으면 그 노드도 분할해야 한다.
- 이러한 분할 과정은 새로운 루트노드를 만들 때까지 계속될 수도 있고,그렇게 되면 B^+ 트리의 새로운 단계가 생긴다.

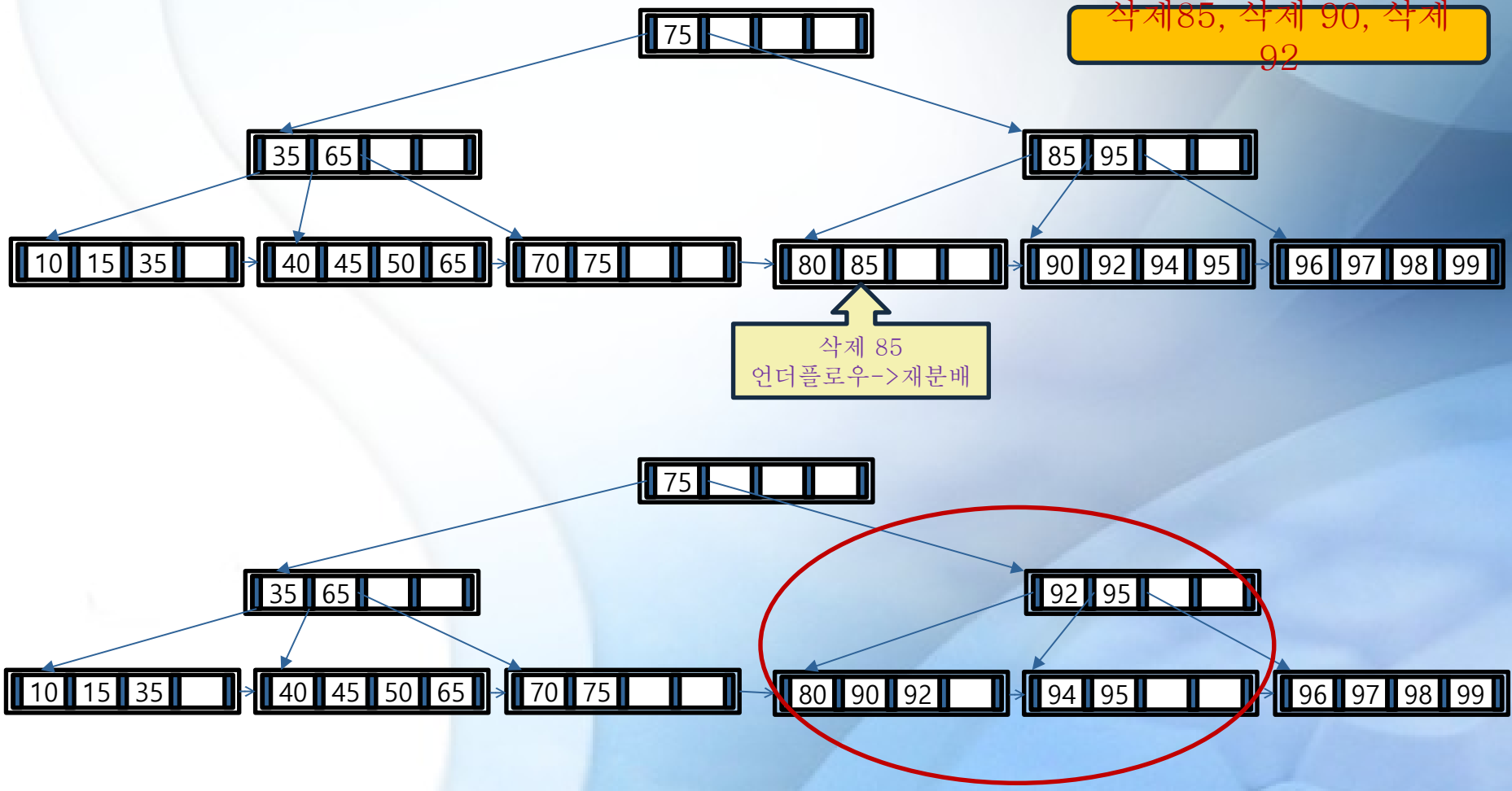
B+ 트리에 대한 삽입의 예

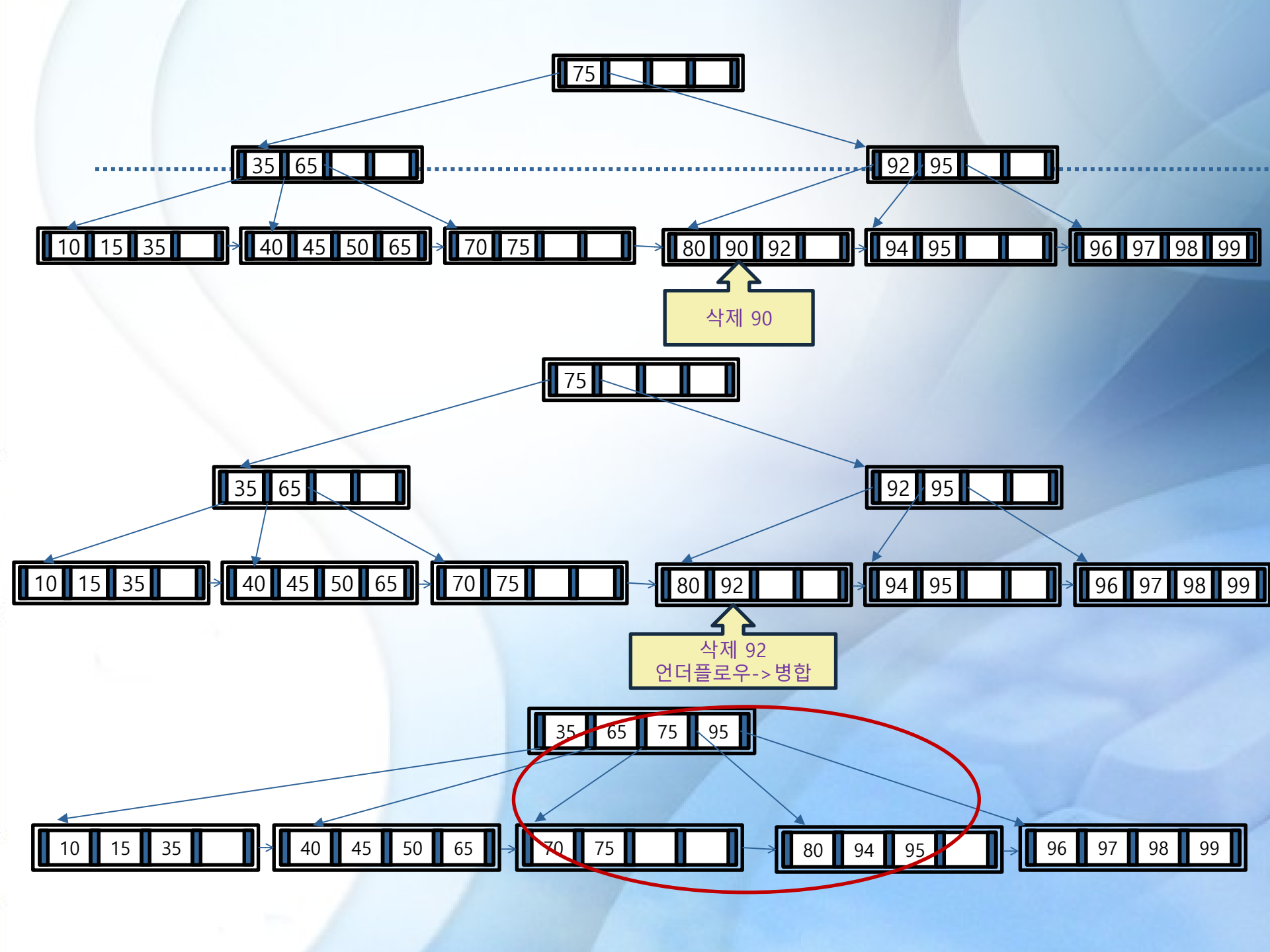

삽입 98, 삽입 17
20 55 65 90
5 10 15 20
40 45 50 55
60 65
80 85 90
95 97 99
삽입 98
20 55 65 90
5 10 15 20
40 45 50 55
60 65
80 85 90
95 97 98 99
삽입 17
오버플로우->분할
55
15 20
65 90
5 10 15
17 20
40 45 50 55
60 65
80 85 90
95 97 98 99

B+ 트리 삭제 알고리즘

- 삭제할 때는 항상 단말노드에서 삭제한다.
- 삭제로 인해 단말노드의 엔트리 수가 절반 이하로 줄어들면 형제 단말노드(sibling leaf node)와 엔트리를 재분배(redistribute)한다.
- 엔트리를 재분배할 때는 두 노드 모두 반 이상이 차도록 한다.
  만약 그렇게 할 수 없으면 형제노드와 병합하여 단말노드의 수를 줄인다.
- 이런 영향이 루트노드에까지 이르면 트리단계가 줄어든다.

B+ 트리에서 삭제하는 예


삭제85, 삭제 90, 삭제 92
75
35 65
85 95
10 15 35
40 45 50 65
70 75
80 85
90 92 94 95
96 97 98 99
삭제 85
언더플로우->재분배
75
35 65
92 95
10 15 35
40 45 50 65
70 75
80 90 92
94 95
96 97 98 99


75
35 65
92 95
10 15 35
40 45 50 65
70 75
80 90 92
94 95
96 97 98 99
삭제 90
75
35 65
92 95
10 15 35
40 45 50 65
70 75
80 92
94 95
96 97 98 99
삭제 92
언더플로우->병합
35 65 75 95
10 15 35
40 45 50 65
70 75
80 94 95
96 97 98 99

다중키 인덱스

여러 필드의 조합에 따라 순서화 된 인덱스

1. 여러 필드의 조합에 따라 어떻게 순서화 할까?
2. 다중키 인덱스는 어느 경우에 사용될까?