다원 탐색 트리
(Multiway Search Trees)

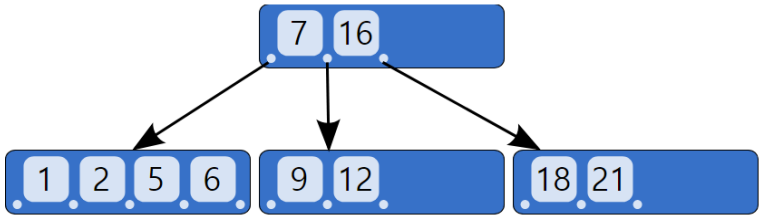
소프트웨어학과

이 의 종

Introduction

Why m-way Search Tree?

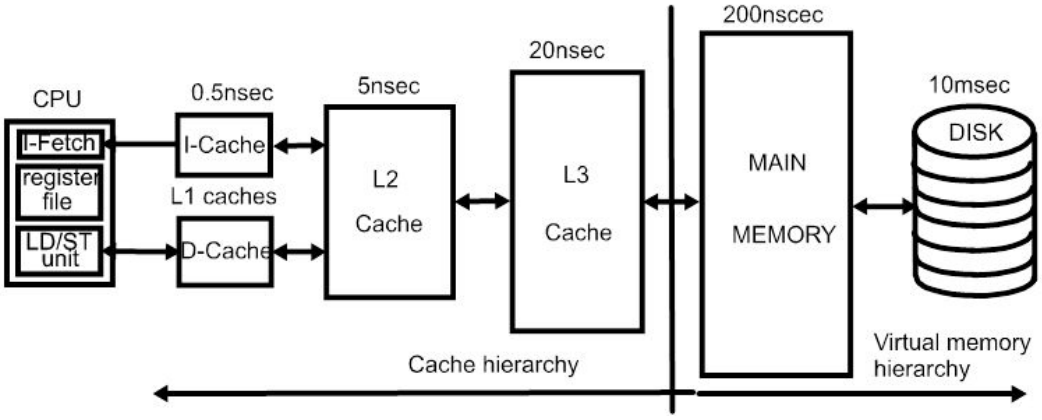
The m-way search trees are multi-way trees which are generalized versions of binary trees where each node contains multiple elements. In an m-Way tree of order m, each node contains a maximum of m – 1 elements and m children.



https://www.geeksforgeeks.org/m-way-search-trees-set-1-searching/

Overview

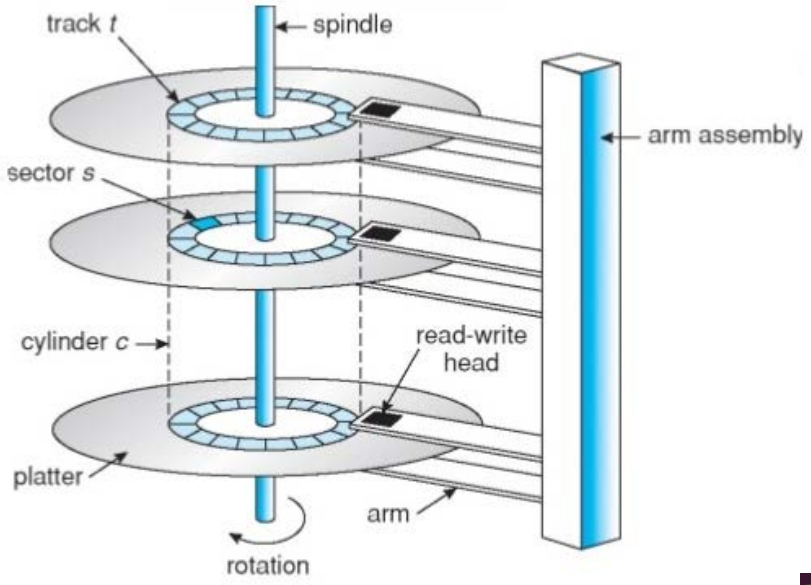
- **산술연산, 논리연산보다 메모리접근(RAM, Disk) 비용이 더 큼**
  - 메인 메모리 접근은 산술연산보다 대략 100정도 시간이 더 걸림
  - 디스크 접근은 산술연산보다 대략 10,000배 더 걸림
- **프로세서 속도와 메모리 접근시간의 차이로 캐시(cache) 사용**
- **디스크에서 메인 메모리로 블록(block) 단위로 데이터 전송**

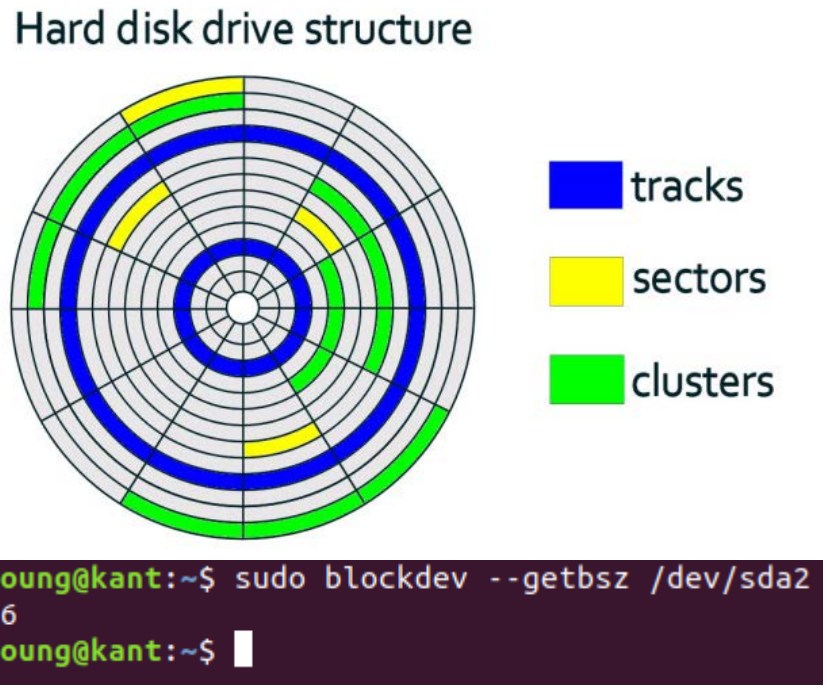


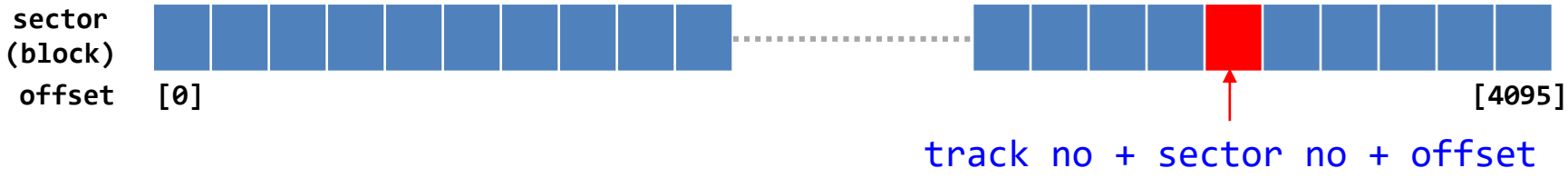

```
rsyoung@kant:~$ lscpu | grep "cache"
L1d cache:                       32K
L1i cache:                       32K
L2 cache:                        256K
L3 cache:                        8192K
rsyoung@kant:~$
```

```
rsyoung@kant:~$ sudo blockdev --getbsz /dev/sda2
4096
rsyoung@kant:~$
```

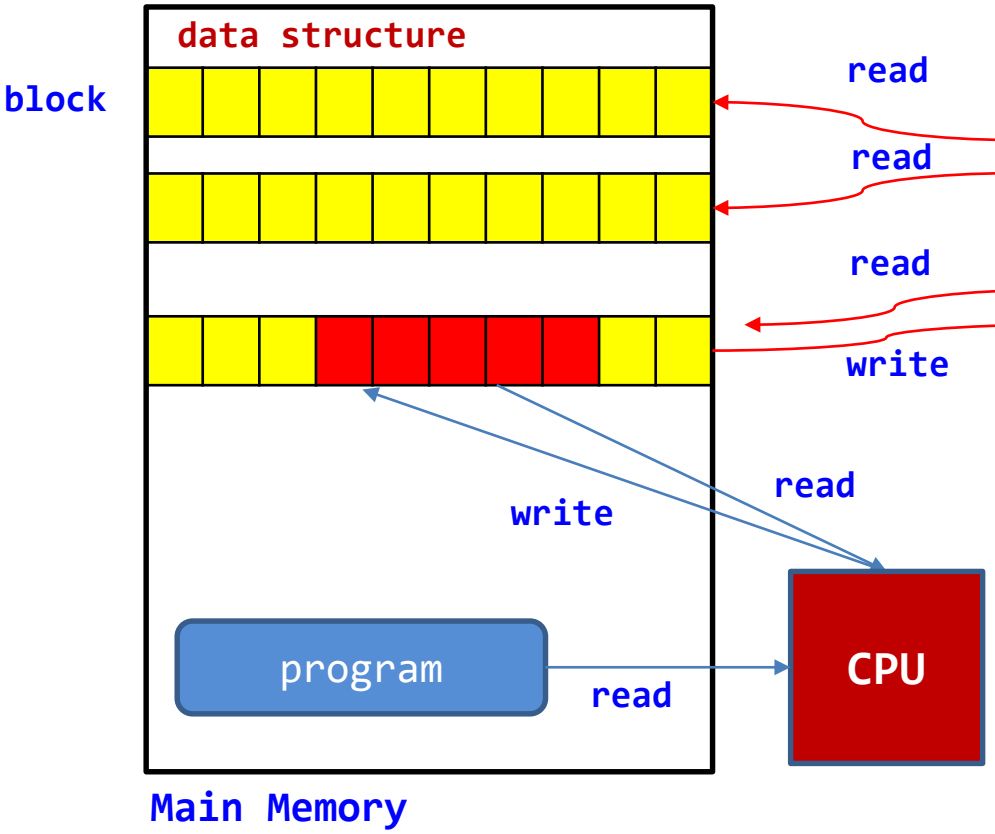
Disk Structure

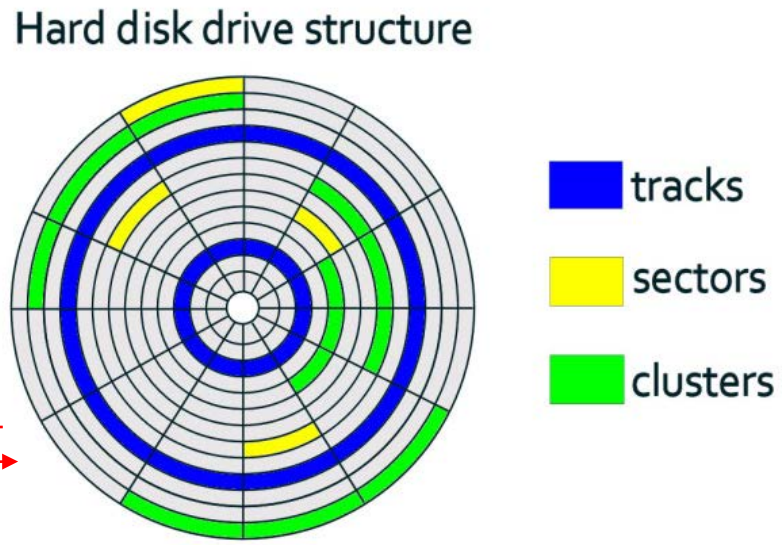




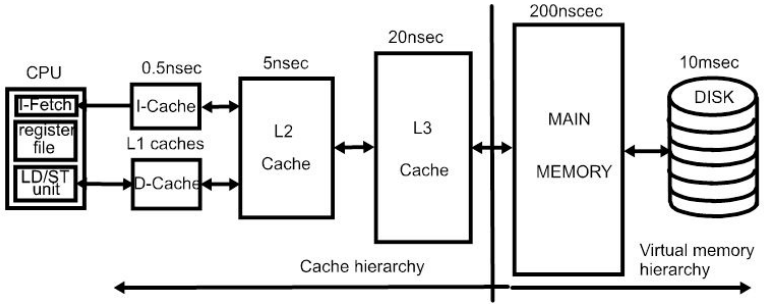


Data between Memory and Disk

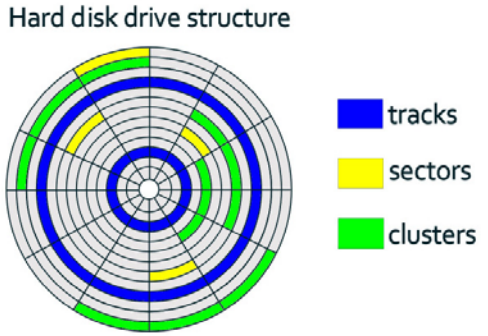




Database Management System (DBMS)



Data on Disk




```
rsyoung@kant:~$ sudo blockdev --getbsz /dev/sda2
4096
rsyoung@kant:~$
```

Block Size = 4096 Bytes = 4KB

size of record = 512 Bytes

# of records per block = 4096 / 512 = 8
8 records/ block

# of blocks for 120 records
= 120 records / 8 records / 1 block
= 15 blocks

1 – 8 --> block #1
9 – 16 --> block #2
17 – 36 --> block #3
...
112 – 200 --> block #15

record를 찾고자 한다면 15개의 block을 access

시간을 줄일 수 있는가?

ID	Name	Dept.	...	...
0001	Kang	Computer		
0002	Kwon	Math.		
0003	Kim	History		
...	...	...		
0120	Hwang	Chem.		

120 records

Data on Disk

시간을 줄일 수 있는가? Yes. Index

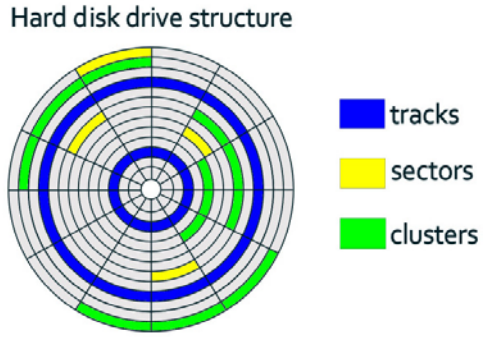
120 records

ID	pointer
0001	block#1
0002	block#1
0003	block#1
...	...
0120	block#15

120 index records

ID	Name	Dept.	...	...
0001	Kang	Computer		
0002	Kwon	Math.		
0003	Kim	History		
...	...	...		
0120	Hwang	Chem.		

index도 block 으로 Disk에 저장



size of index record = 16 Bytes

# of records per block = 4096 / 16 = 256

database record = 120

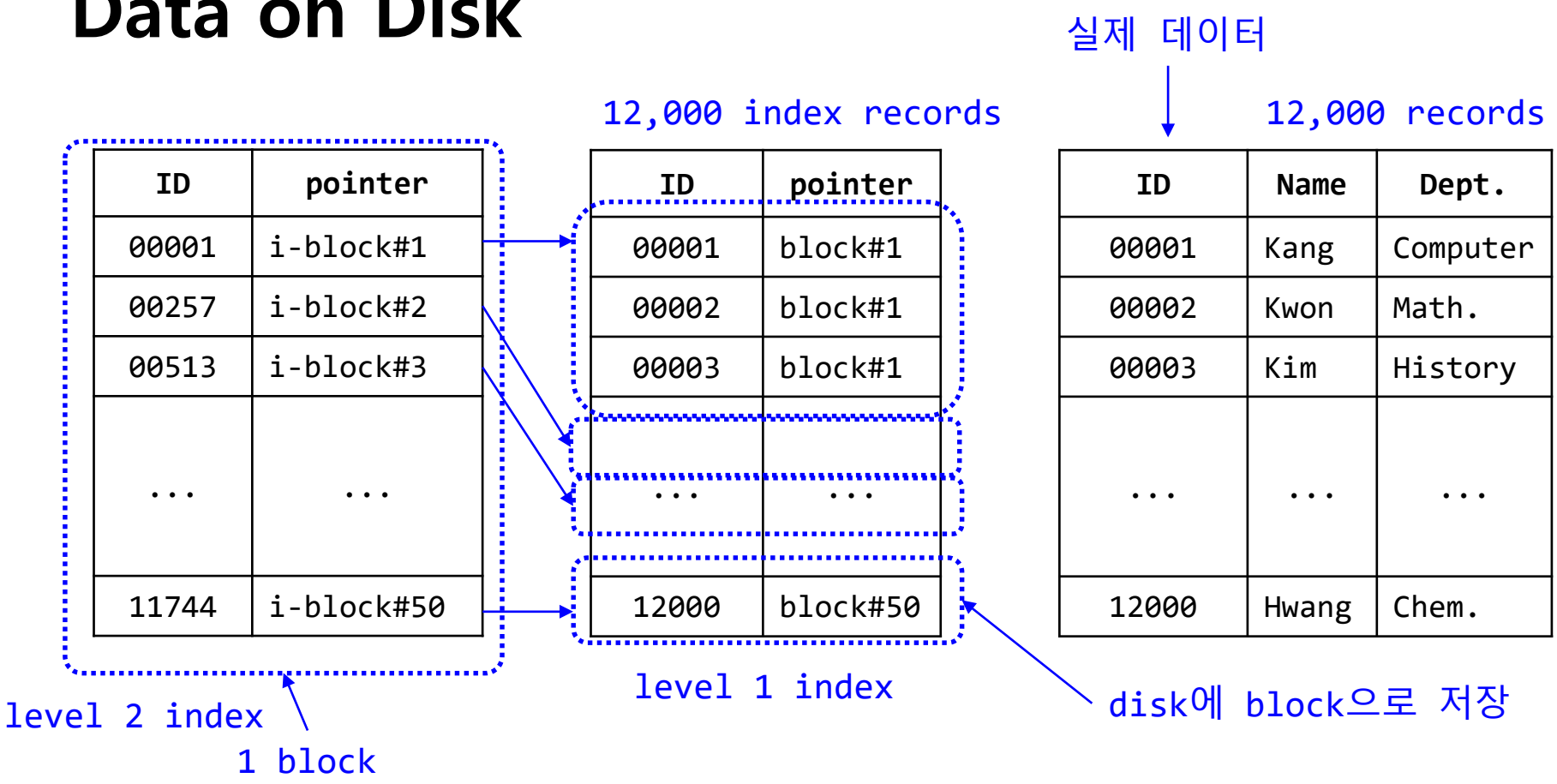
→ # of index block = 120/256 ~ 0.5 → 1 block

→ total 1 + 1 = 2 block access

records의 수가 120 → 12,000 ?

50 + 1 block access

Data on Disk

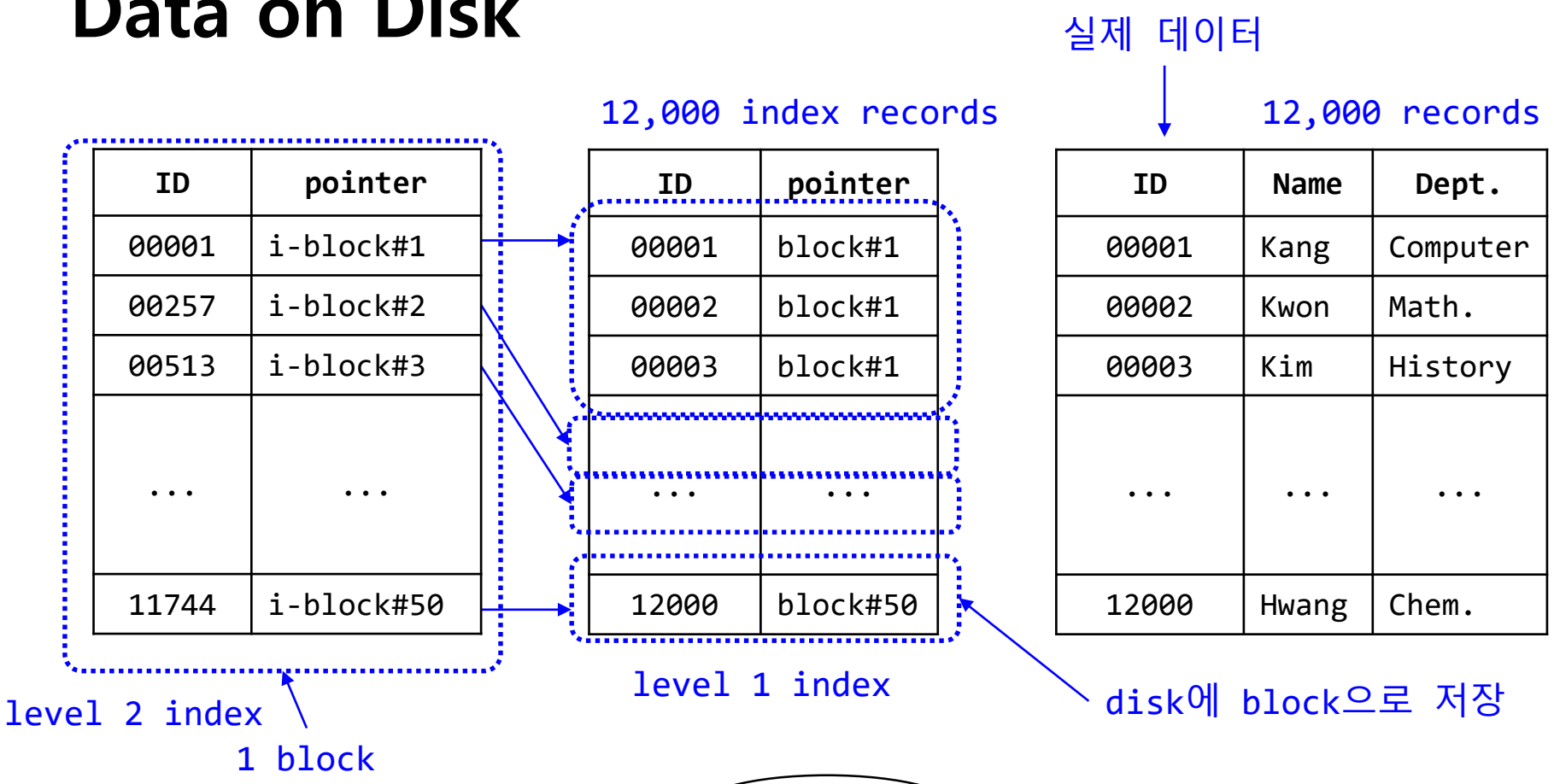


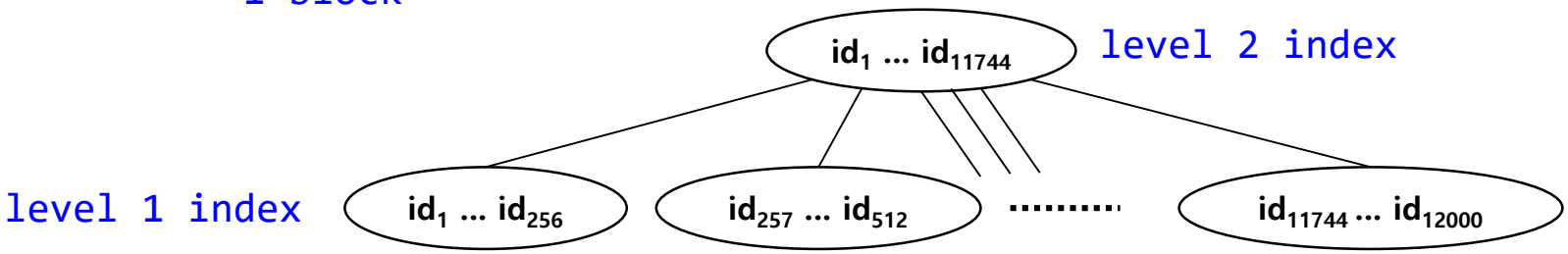
records의 수가 120 → 12,000 ?

level 1 index만 사용 → 50 + 1 block access

level 1, 2 index 사용 → 1 + 1 + 1 = 3 block access

Data on Disk

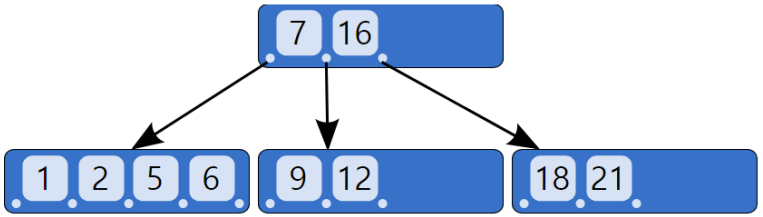




self-managed balanced search tree = m-way search tree

m-원 탐색 트리
(m-way Search Trees)

A binary search tree has one value in each node and two subtrees. This notion easily generalizes to an m-way search tree, which has (m-1) values per node and m subtrees.



https://webdocs.cs.ualberta.ca/~holte/T26/m-way-trees.html

Why? m-원 탐색 트리

- AVL, Red-Black Tree의 탐색 성능 = $O(\log_2 n)$
- n = 1,000,000인 AVL Tree = $\lfloor 1.44 \log_2(n+2) \rfloor = 28$

The height h of an AVL tree with n nodes lies in the interval:[8]

$$\log_2(n+1) \leq h < c \log_2(n+2) + b$$

with the golden ratio $\varphi := (1+\sqrt{5})/2 \approx 1.618$, $c := 1/\log_2 \varphi \approx 1.44$, and $b := 5/2 \log_2 5 - 2 \approx -0.328$.

https://en.wikipedia.org/wiki/AVL_tree

노드가 모든 다른 블록에 있는 경우(Worst Case) 28번의 메모리 접근 필요

탐색 시간 대부분을 메모리 접근에 소비

메모리 접근 횟수는 Tree의 높이와 밀접하게 연관 → degree를 2보다 큰 탐색 트리를 이용 → m-way search tree → node의 크기가 cacheline이나 disk block size 만큼 큰 차수(degree) 사용

정의와 성질(Definition and Properties)

- m-원 탐색트리는 공백(empty)이거나 다음 성질을 만족
  - (1) 루트는 최대 m개의 서브트리를 가진다.
    - 구조 : n, A_0, (E_1, A_1), (E_2, A_2), ... , (E_n, A_n) $(0 \le i < n < m)$
    - E_i는 원소(element)를 의미
    - 각 원소 E_i는 $E_i.K$ 키(key)를 가지고 있음
    - A_i는 서브트리에 대한 포인터
  - (2) $E_i.K < E_{i+1}.K$ $(1 \le i < n)$
  - (3) $E_0.K = -\infty$, $E_{n+1}.K = \infty$이다.
    - $E_i.K <$ 서브트리 A_i의 모든 키 $< E_{i+1}.K$ $(0 \le i \le n)$
  - (4) $0 \le i \le n$ 에 대하여 서브 트리 A_i 역시 m-원 탐색 트리이다.

n = # of elements

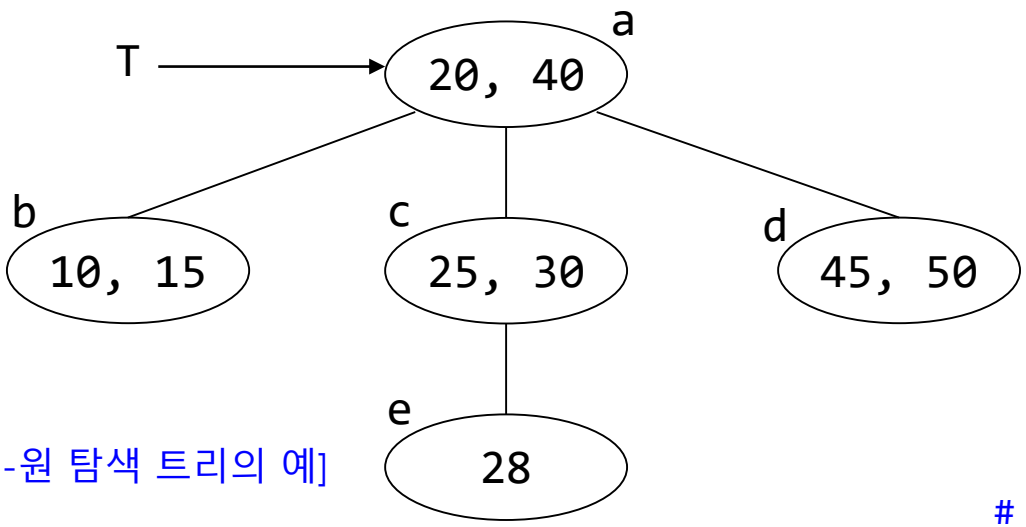
note:
element = key + data

$E_0.K$ = 왼쪽 끝
$E_{n+1}.K$ = 오른쪽 끝

실제 존재하지는 않고 편의성 때문에 정의

정의와 성질(Definition and Properties)

$n, A_0, (E_1, A_1), (E_2, A_2), \ldots, (E_n, A_n)$



[3-원 탐색 트리의 예]

node	schematic format
a	2, b, (20, c), (40, d)
b	2, 0, (10, 0), (15, 0)
c	2, 0, (25, e), (30, 0)
d	2, 0, (45, 0), (50, 0)
e	1, 0, (28, 0)

# of elements A_0 E_1 A_1

차수가 m이고 높이가 h인 트리에서 최대 노드 수(node)

$$\sum_{0 \le i \le h-1} m^i = (m^h - 1)/(m-1)$$

Element의 수

h=3, binary search tree
# of element = 7

h=3, 200-way search tree
# of element = $200^3 - 1 = 8 \times 10^6 - 1$

최적의 m-원 탐색 트리의 성능 → 트리가 반드시 균형을 이루어야 함

균형 m-원 탐색 트리의 특별한 형태: **B-Tree, B⁺-Tree**

m-원 탐색 트리에서의 탐색
(Searching an m-way Search Tree)

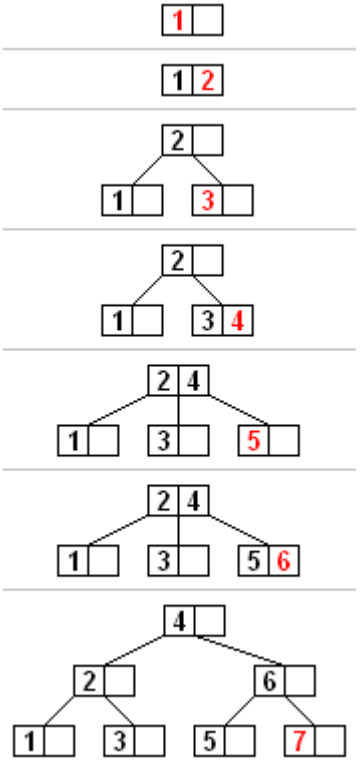
- 트리의 루트에서 시작
- $E_i.K \le x < E_{i+1}.K$를 만족하는 i를 찾는다.
- $x = E_i.K$이면 탐색 완료
- $x \neq E_i.K$이고 x가 존재한다면 서브트리 A_i를 루트로 탐색 반복

```
/* m-원 탐색 트리에서 키가 x인 원소를 찾는다.
  원소를 찾으면 그 원소를 반환한다. 그렇지 않으면 NULL을 반환한다. */
  E0.K = -MAXKEY;  /* E0.K = -∞, En+1.K = +∞ 라고 가정 */
  for(*p = root; p; p = Ai){
      p의 형식이 n, A0, (E1, A1), ..., (En, An)이라 하자;
      En+1.K = MAXKEY;
      Ei.K ≤ x < Ei+1.K와 같은 i를 결정;
      if(x == Ei.K) return Ei;
  }
  return NULL; /* x는 트리에 없다. */
```

m-원 탐색 트리에서 키 x를 가진 원소를 탐색

Rudolf Bayer (born 3 March 1939) is a German computer scientist.

B-트리 (B-Trees)



B tree of order of 3 (2-3 tree)

전산학에서 B-트리(B-tree)는 데이터베이스와 파일 시스템에서 널리 사용되는 트리 자료구조의 일종으로, 이진 트리를 확장해 하나의 노드가 가질 수 있는 자식 노드의 최대 숫자가 2보다 큰 트리 구조이다.

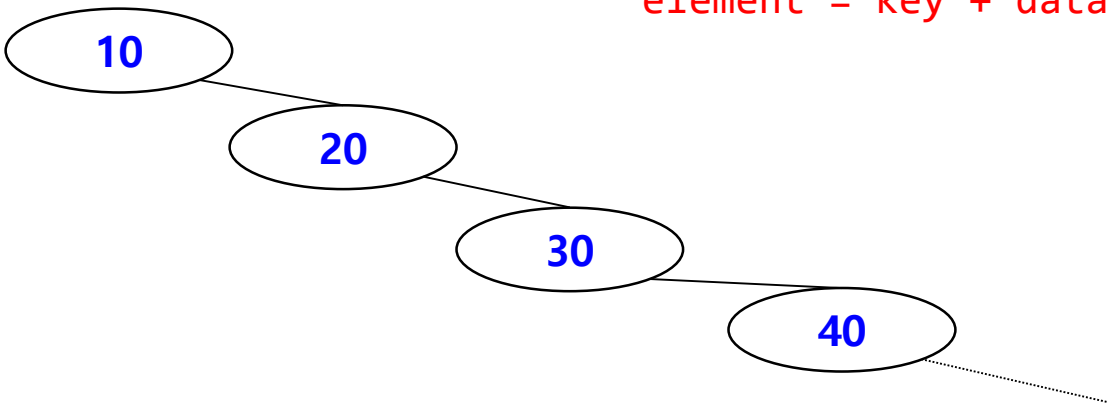
B-트리의 창시자인 루돌프 바이어는 'B'가 무엇을 의미하는지 따로 언급하지는 않았다. 가장 가능성 있는 대답은 리프 노드를 같은 높이에서 유지시켜주므로 균형잡혀있다(Balanced)는 뜻에서의 'B'라는 것이다. '바이어(Bayer)'의 'B'를 나타낸다는 의견도, 혹은 그가 일했던 보잉 과학 연구소(Boeing Scientific Research Labs)에서의 'B'를 나타낸다는 의견도 있다.

-- wikipedia

정의와 성질(Definition and Properties)

- **10-way search tree (9 elements)**
  - 10, 20, 30, 40, 50, 60, 70, 80, 90 ... 순차적으로 삽입

element = key + data (or data pointer)



균형(balance)을 위한 규칙(rule)이 필요

차수가 m인 B-트리
(B-tree of order m)

- **B-tree 정의**
  (1) 루트 노드는 적어도 2개의 자식을 갖는다.
  (2) 모든 노드는 적어도 $\lceil \frac{m}{2} \rceil$ 개의 자식을 갖는다. (루트, 외부노드 제외)
  (3) 모든 외부 노드(external node)들은 같은 레벨에 있다.

원소를 검색하지 못 했을 때 도달하는 노드

정의와 성질(Definition and Properties)

차수가 m인 B-트리(B-tree of order m)

2-3-4-5 Tree?

- **B-tree Guideline**
  (1) 루트 노드는 적어도 2개의 자식을 갖는다.
  (2) 모든 노드는 적어도 $\lceil \frac{m}{2} \rceil$ 개의 자식을 갖는다. (루트, 외부노드 제외)
  (3) 모든 외부 노드(external node)들은 같은 레벨에 있다.

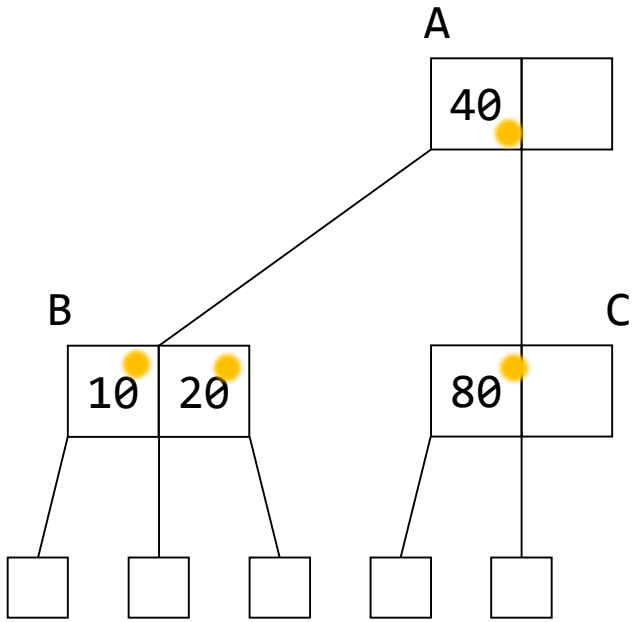
d = min. # of keys

d+1 – 2d+1

2-3 tree, 4-7 tree

B-Tree Example (2-3 Tree)

- 차수가 3인 B-트리 (B tree of order 3)



2-3 트리의 예

element =
key + data (or data pointer)

2-3 Tree =
max 2 element를 가질 수 있음

- **B-tree Guideline**
  (1) 루트 노드 = 적어도 2개의 자식을 가짐.
  (2) 모든 노드는 적어도 $\lceil \frac{m}{2} \rceil$ 개의 자식을 가짐.
  (3) 외부 노드(external node)들은 같은 레벨에 있음.

B-Tree Example (2-3-4 Tree)

- 차수가 4인 B-트리
  (B tree of order 4)

• **B-tree Guideline**

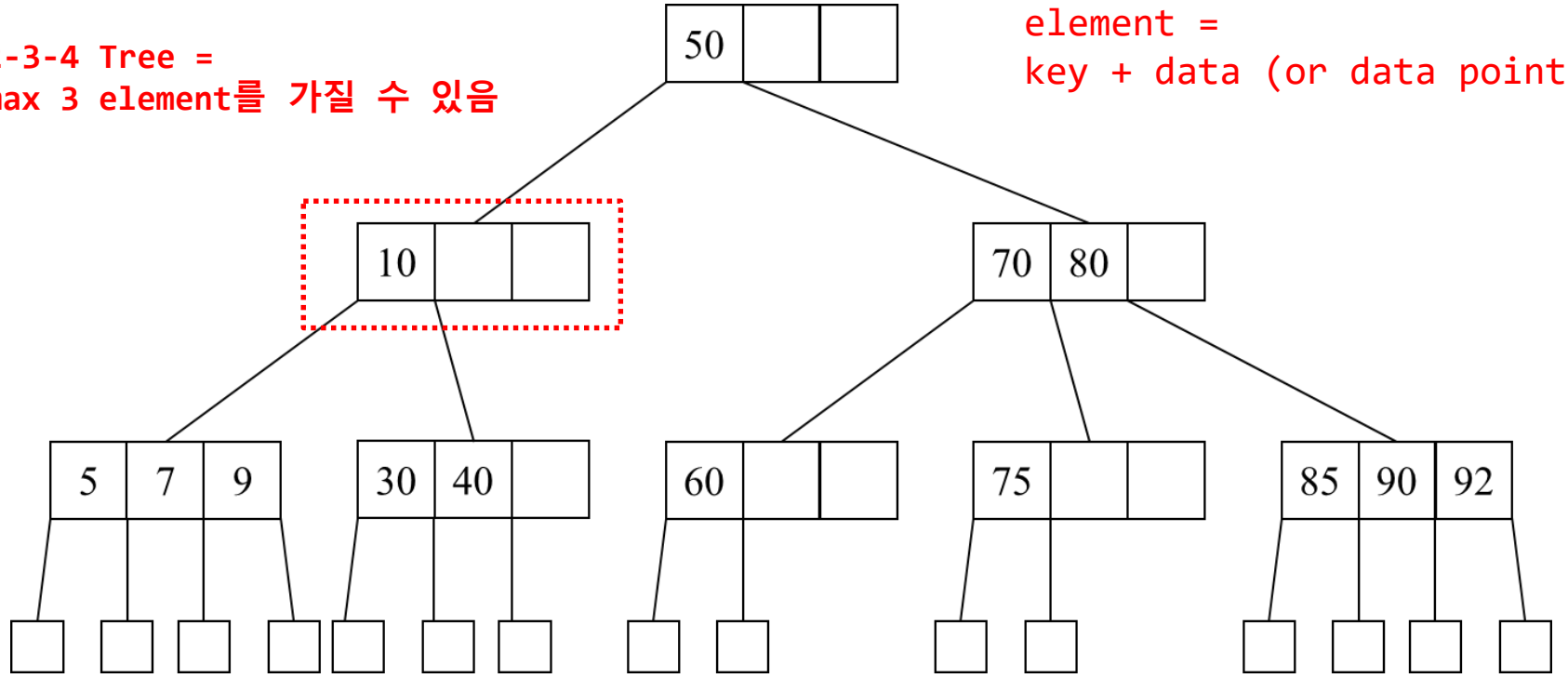
(1) 루트 노드 = 적어도 2개의 자식을 가짐.

(2) 모든 노드는 적어도 $\lceil \frac{m}{2} \rceil$ 개의 자식을 가짐.

(3) 외부 노드(external node)들은 같은 레벨에 있음.

2-3-4 Tree =
max 3 element를 가질 수 있음

element =
key + data (or data pointer)



2-3-4 트리의 예

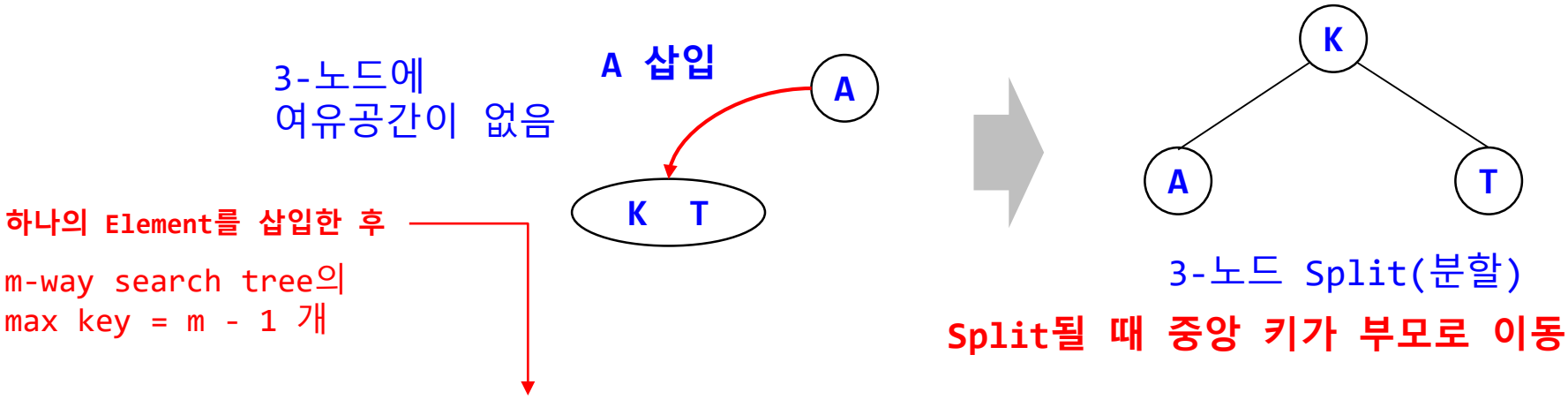
B-Tree의 원소 수(Number of Elements in a B-Tree)

- **모든 외부 노드의 레벨이 l + 1인 차수가 m인 B-트리**
  - 최대 m^l - 1개의 키를 가짐
  - 최소 원소 수 N은?
    - (level 1) root = 1개 원소, 2개의 자식 노드
    - (level 2) 2개의 노드는 최소 $\lceil \frac{m}{2} \rceil$개의 자식 노드 → $2\lceil \frac{m}{2} \rceil$ 자식 노드
    - (level l) $2(\lceil \frac{m}{2} \rceil)^{l-2}$
    - (level l+1, external node) $2(\lceil \frac{m}{2} \rceil)^{l-1}$
    - 트리에 있는 키들을 K_1, K_2,…, K_N ($K_i < K_{i+1}$, $1 \le i < N$)이라 할 때 외부 노드의 수(NULL node)는 N + 1개

$$N + 1 \geq 2(\lceil \frac{m}{2} \rceil)^{l-1}$$

B-Tree에서의 삽입(Insertion into a B-Tree)

- 2-3 Tree (B-Tree of order of 3) 삽입(Insertion), 분할(Split)



leaf node p = m개의 key를 가짐 → p split

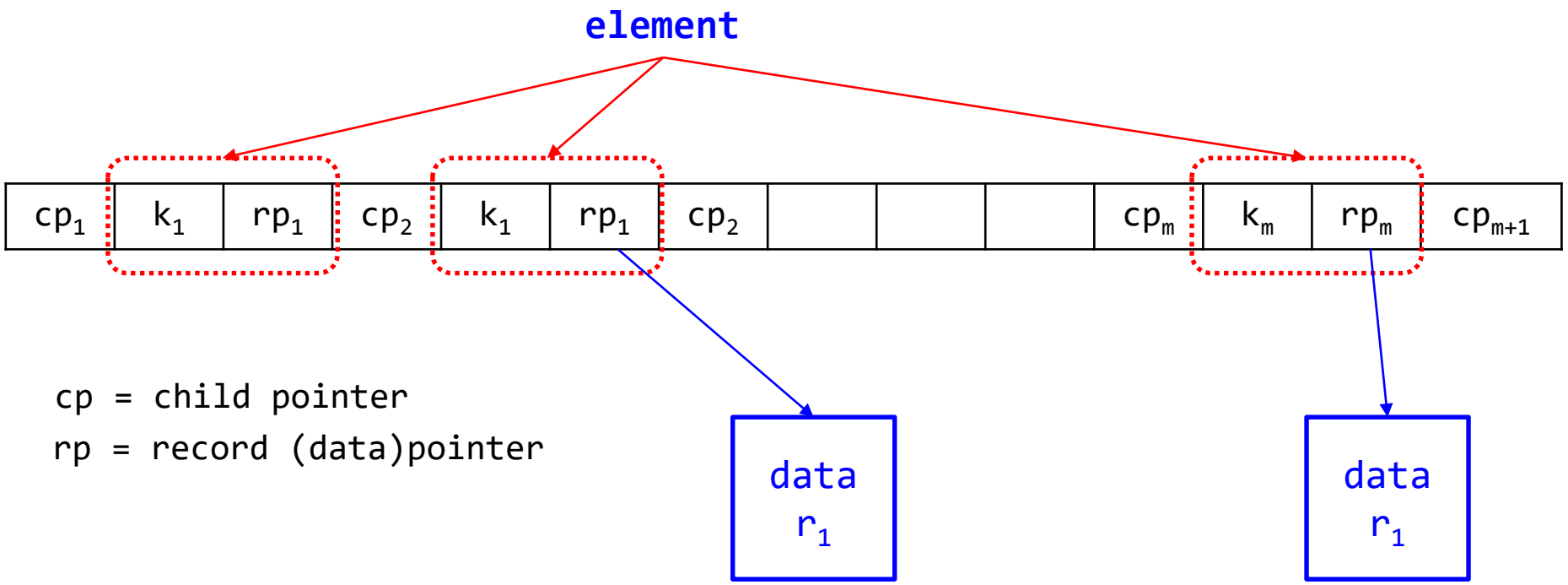
$m, A_0, (E_1, A_1), \ldots, (E_m, A_m), \quad E_i < E_{i+1}, 1 \le i < m$

node p : $\lceil m/2 \rceil - 1, A_0, (E_1, A_1), \ldots, (E_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1})$

node q : $m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, (E_{\lceil m/2 \rceil + 1}, A_{\lceil m/2 \rceil + 1}), \ldots, (E_m, A_m)$

$(E_{\lceil m/2 \rceil}, q)$는 p의 부모에 삽입

B-Tree의 노드 구조(node structure)

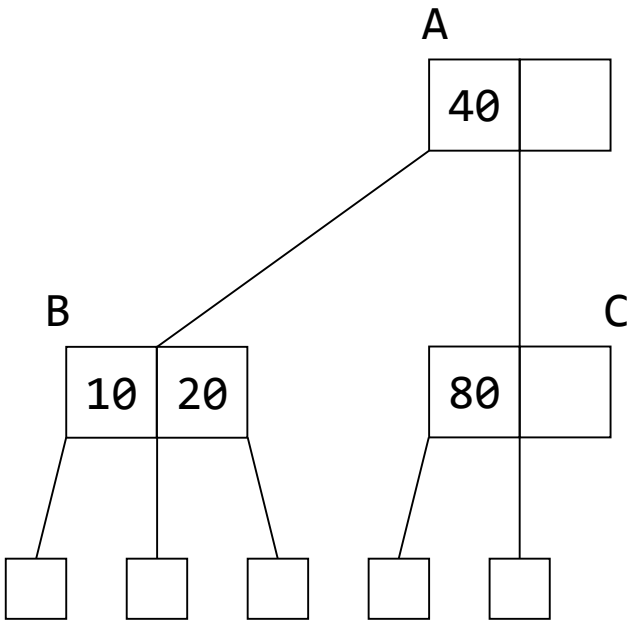


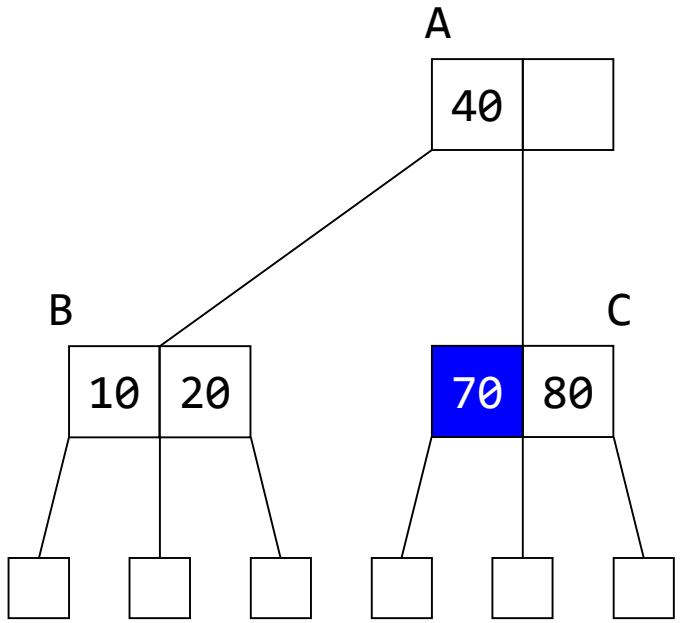
cp = child pointer
rp = record (data)pointer

rp위치에 데이터를 직접 저장할 수 도 있다.

element =
key + data (or data pointer)

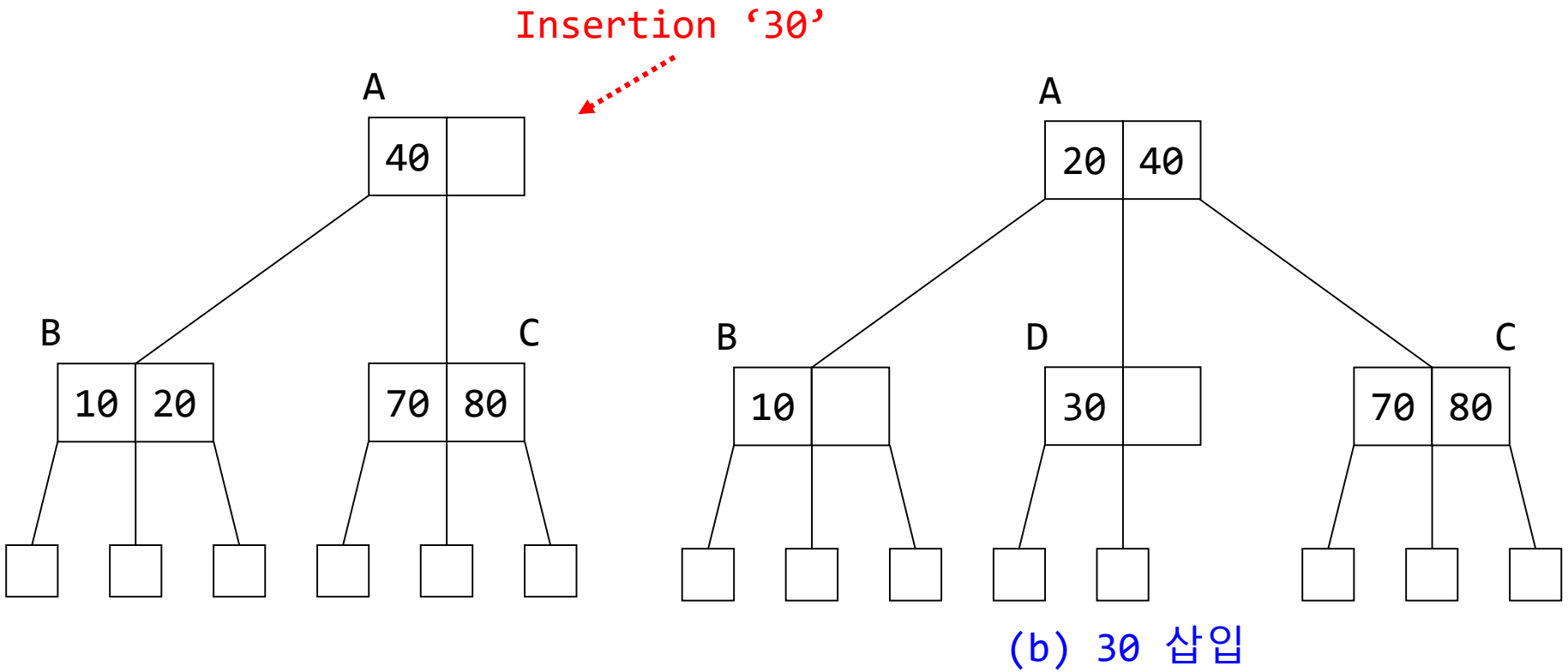
B-Tree에서의 삽입(Insertion into a B-Tree)





(a) 70 삽입

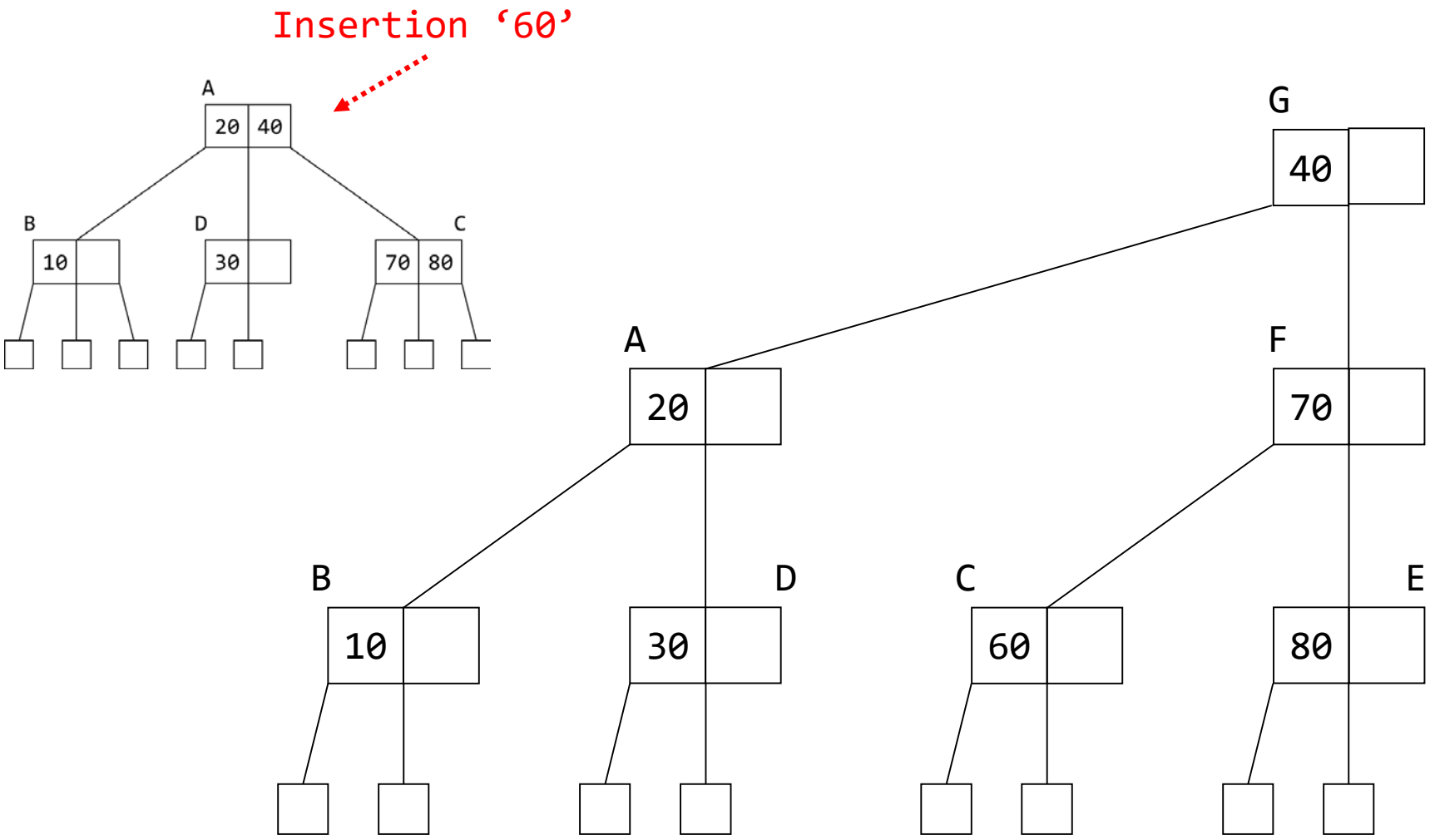
B-Tree에서의 삽입(Insertion into a B-Tree)



(b) 30 삽입

2-3 Tree의 Insertion과 동일

B-Tree에서의 삽입(Insertion into a B-Tree)



2-3 트리에 60을 삽입

B-Tree에서의 삭제(Deletion from a B-Tree)

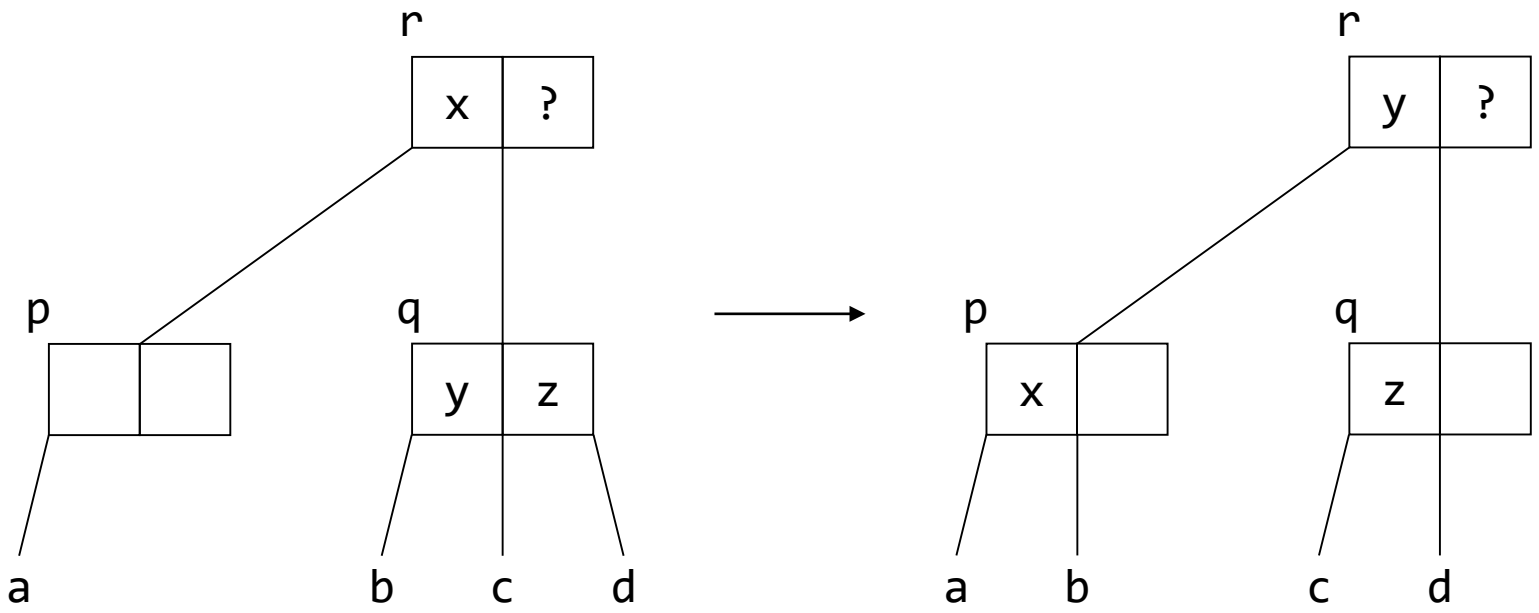
- **"삭제는 리프노드에서만 이루어짐"**

- **삭제할 아이템이 리프노드(leaf node)가 아니라면**
  - 삭제할 노드의 후속노드(successor)와 Swap한 후 노드 삭제

- **삭제할 아이템이 리프노드에 있다면**
  - 노드에 다른 아이템이 존재하면, 단순 삭제
  - 그렇지 않다면 형제노드(siblings)에서 아이템을 빌리고 삭제
  - 형제노드에서 빌릴 수 없다면 병합

B-Tree에서의 삭제(Deletion from a B-Tree)

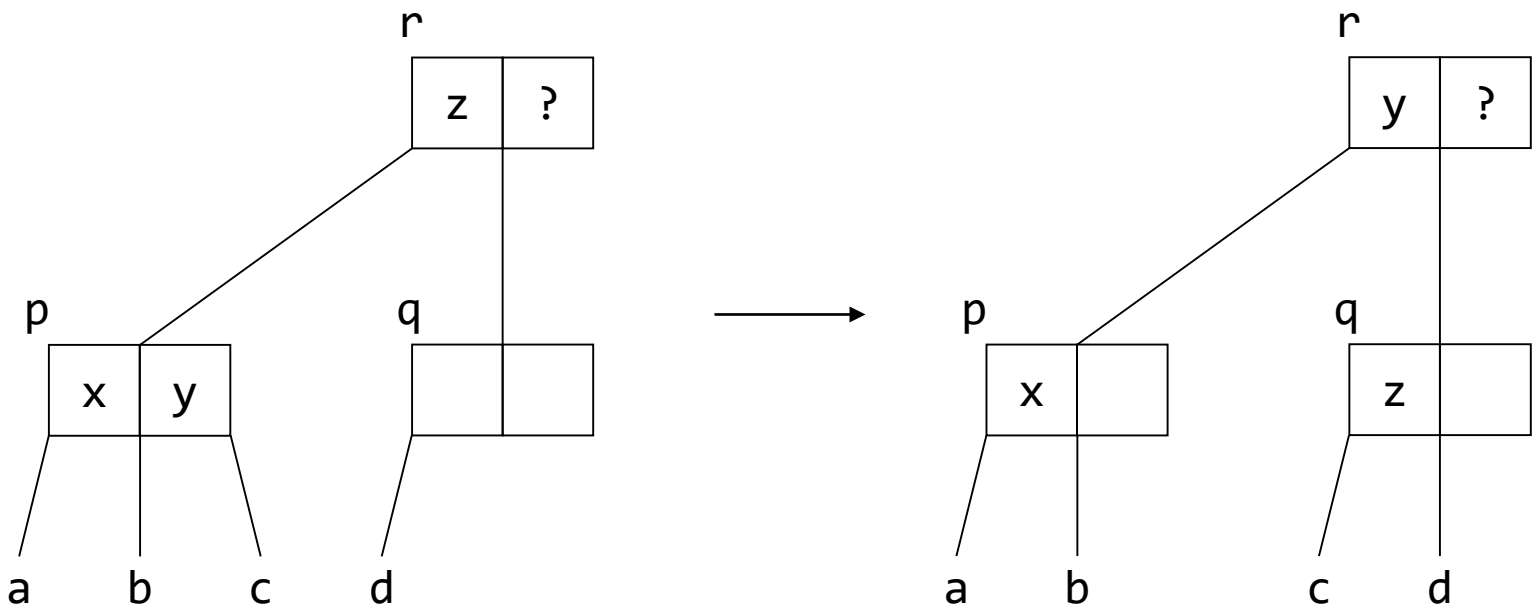
sibling에서 빌려 오기
빌릴 수 없으면 부모와 병합(merge)



(a) p는 r의 왼쪽 자식이다.

B-Tree에서의 삭제(Deletion from a B-Tree)

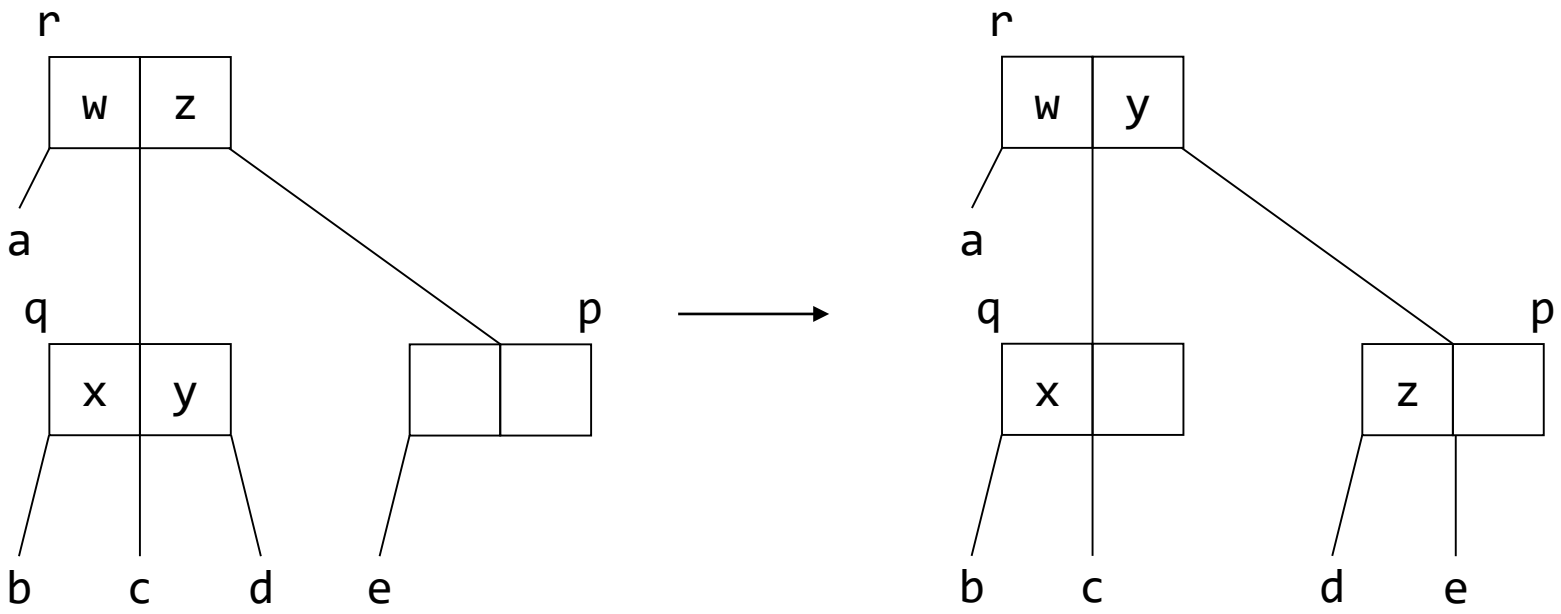
sibling에서 빌려 오기
빌릴 수 없으면 부모와 병합(merge)



(b) p는 r의 중간 자식이다.

B-Tree에서의 삭제(Deletion from a B-Tree)

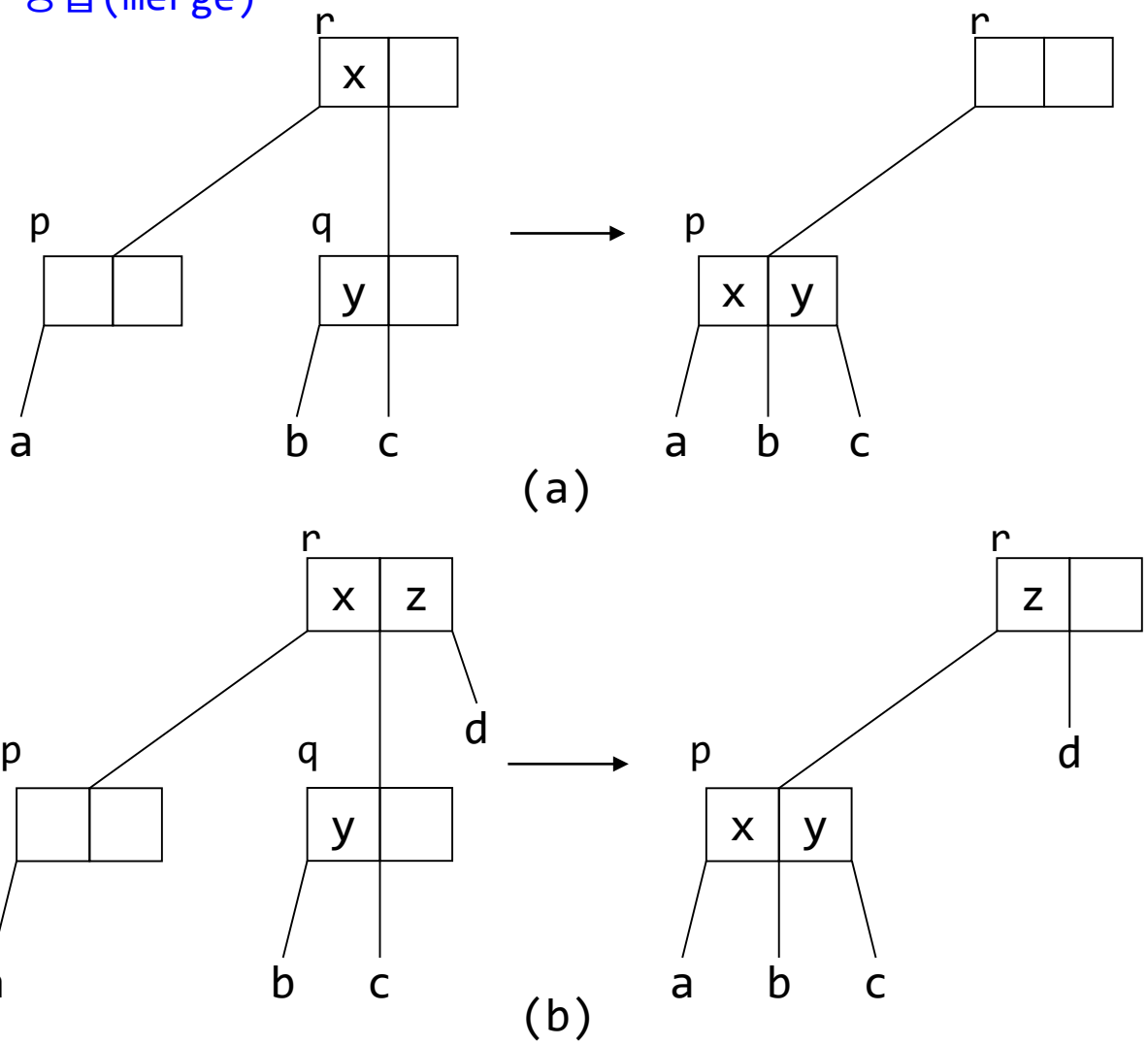
sibling에서 빌려 오기
빌릴 수 없으면 부모와 병합(merge)



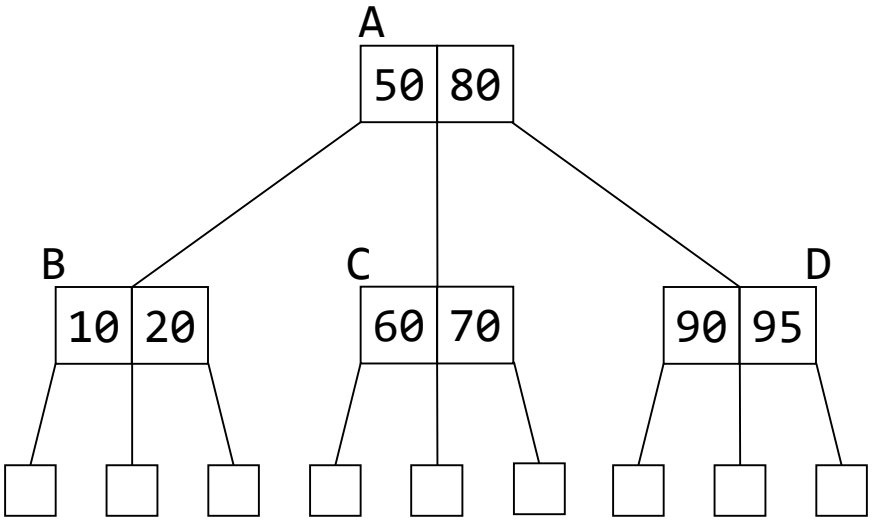
(c) p는 r의 오른쪽 자식이다.

B-Tree에서의 삭제(Deletion from a B-Tree)

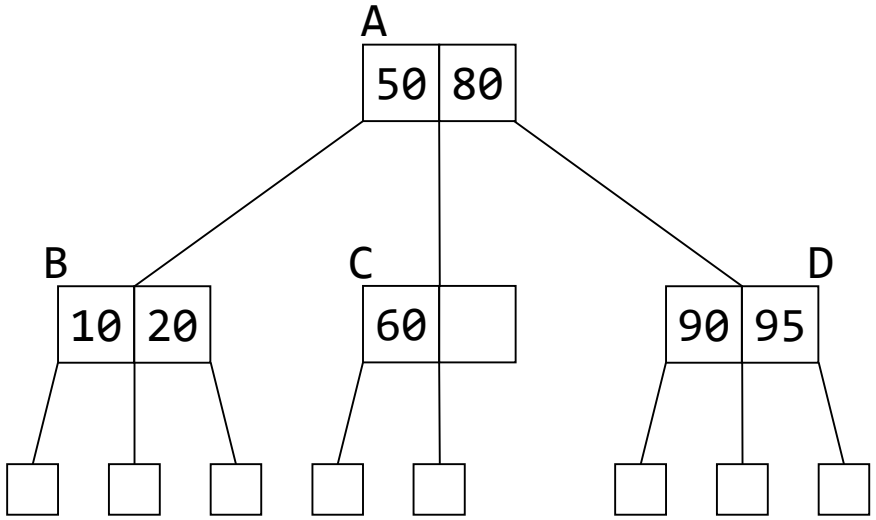
sibling에서 빌려 오기
빌릴 수 없으면 부모와 병합(merge)



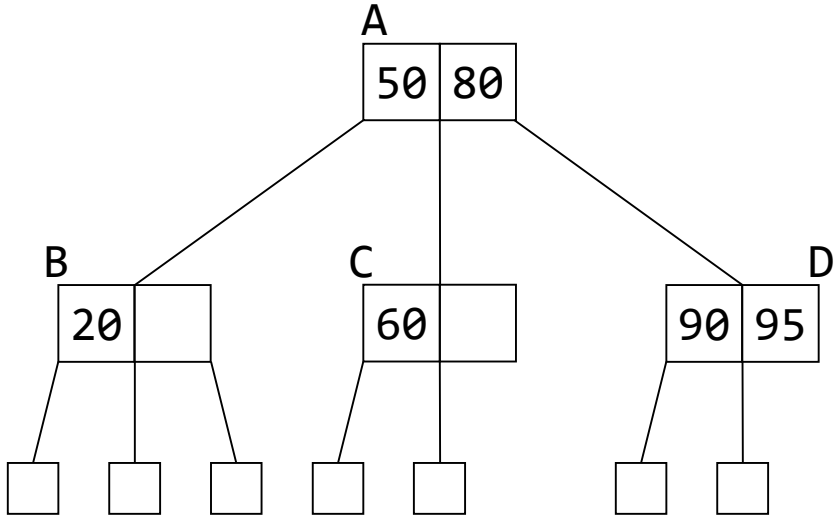
삭제 예제(Deletion Example)



(a) 2-3트리의 초기 상태

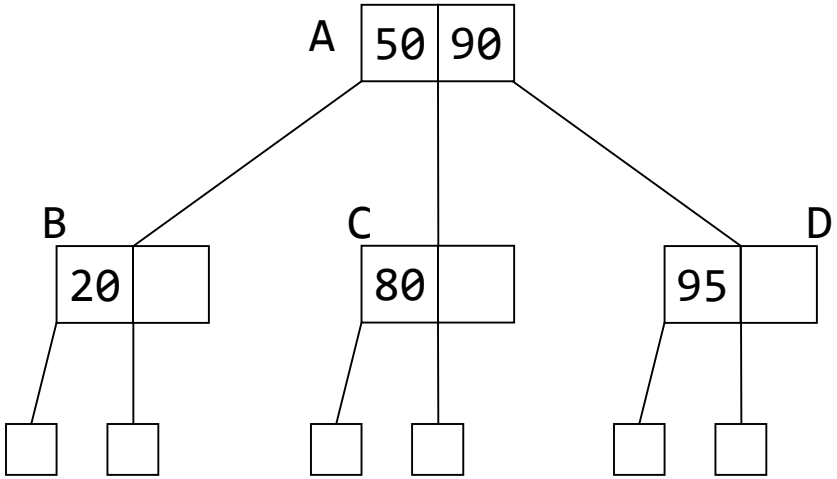


(b) 70이 삭제됨

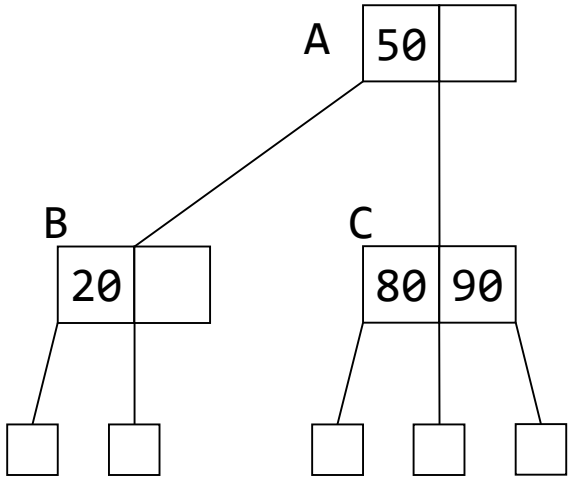


(c) 10이 삭제됨

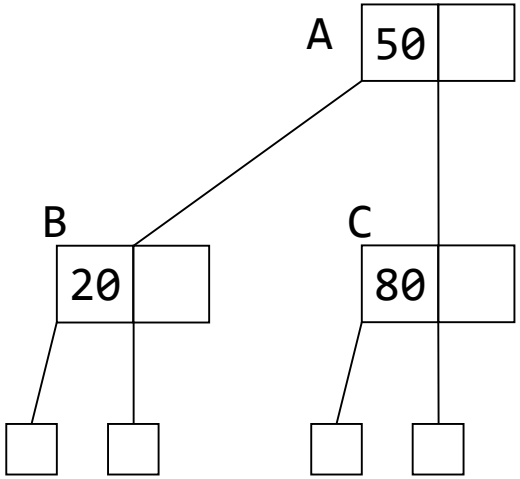
삭제 예제(Deletion Example)



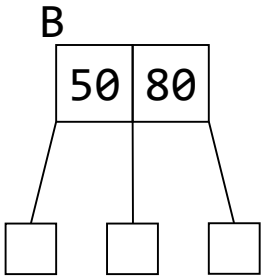
(d) 60이 삭제됨



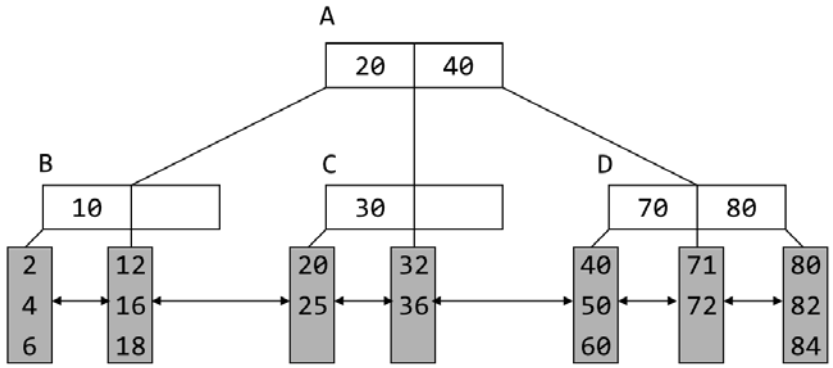
(e) 95가 삭제됨



(f) 90이 삭제됨



(g) 20이 삭제됨



B+ 트리
(B+ Trees)

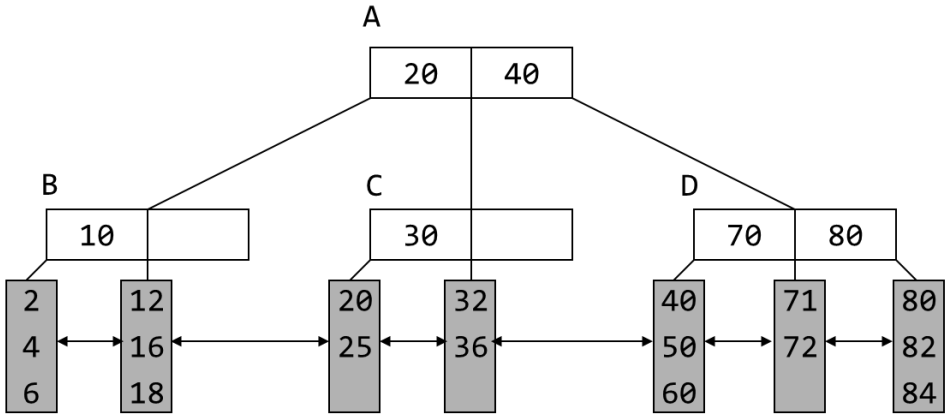
A B+ tree can be viewed as a B-tree in which each node contains only keys (not key–value pairs), and to which an additional level is added at the bottom with linked leaves.

The ReiserFS, NSS, XFS, JFS, ReFS, and BFS filesystems all use this type of tree for metadata indexing; BFS also uses B+ trees for storing directories. NTFS uses B+ trees for directory and security-related metadata indexing.

Relational database management systems such as IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and SQLite support this type of tree for table indices. -- wikipedia

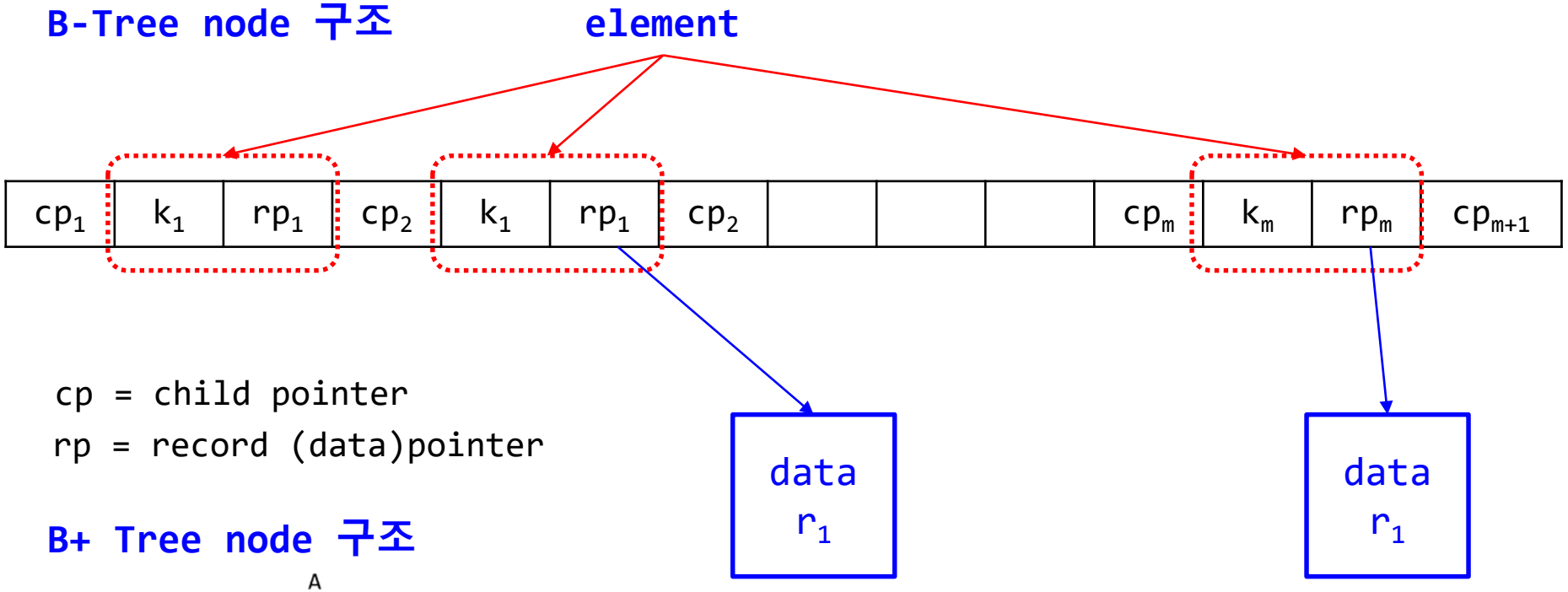
B-Tree와 차이점(Difference from B-Tree)

- (1) 인덱스(index) 노드와 데이타(data)노드의 분리
  - 인덱스 노드 : 키와 포인터를 저장 (element를 저장하지 않음)
  - 데이타 노드 : 키와 함께 데이터를 저장 (element)
- (2) 데이타 노드는 왼쪽에서 오른쪽 순서대로 서로 링크 되어 있고 이중 연결 리스트를 형성



B-Tree와 차이점(Difference from B-Tree)

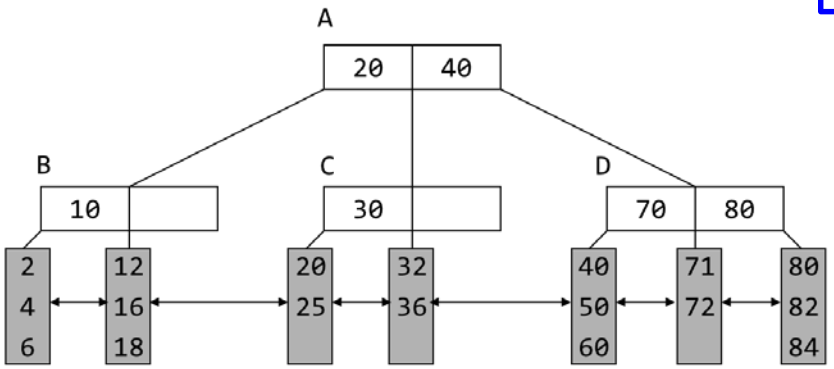
B-Tree node 구조



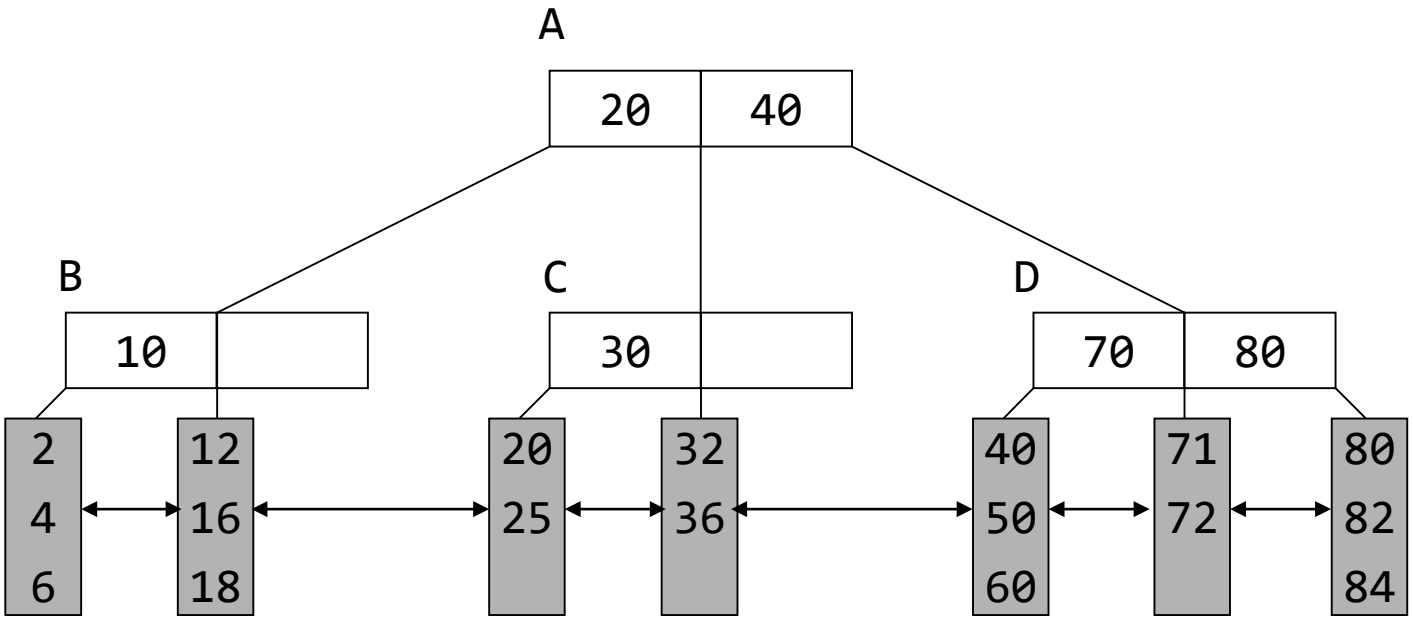
cp = child pointer

rp = record (data)pointer

B+ Tree node 구조



B+ Tree Example



- 데이터 노드 (회색)
- 인덱스 노드들은 높이 2인 2-3 트리를 형성하고 있음
- 데이터 노드 크기와 인덱스 노드 크기는 똑같지 않아도 된다

B+ Tree 정의(Definition)

차수가 m인 B^+ 트리(B^+ tree of order m)

(1) 모든 데이타 노드는 같은 레벨에 위치해있고, 리프 노드이다.

(2) 인덱스 노드는 차수가 m인 B-트리, 단 키만 있고 데이터를 갖지 않음.

(3) 인덱스 노드의 형식 : B^+-tree node

$n, A_i, (K_1,A_1), (K_2,A_2),..., (K_n, A_n)$

- $A_i (0 \le i \le n < m)$가 서브트리에 대한 포인터, $K_i (1 \le i \le n < m)$는 키
- $K_0 = -\infty$, $K_{n+1} = \infty$
- 서브트리 A_i의 모든 원소는 $0 \le i \le n$일 때, K_{i+1}보다 작고 K_i보다 크다

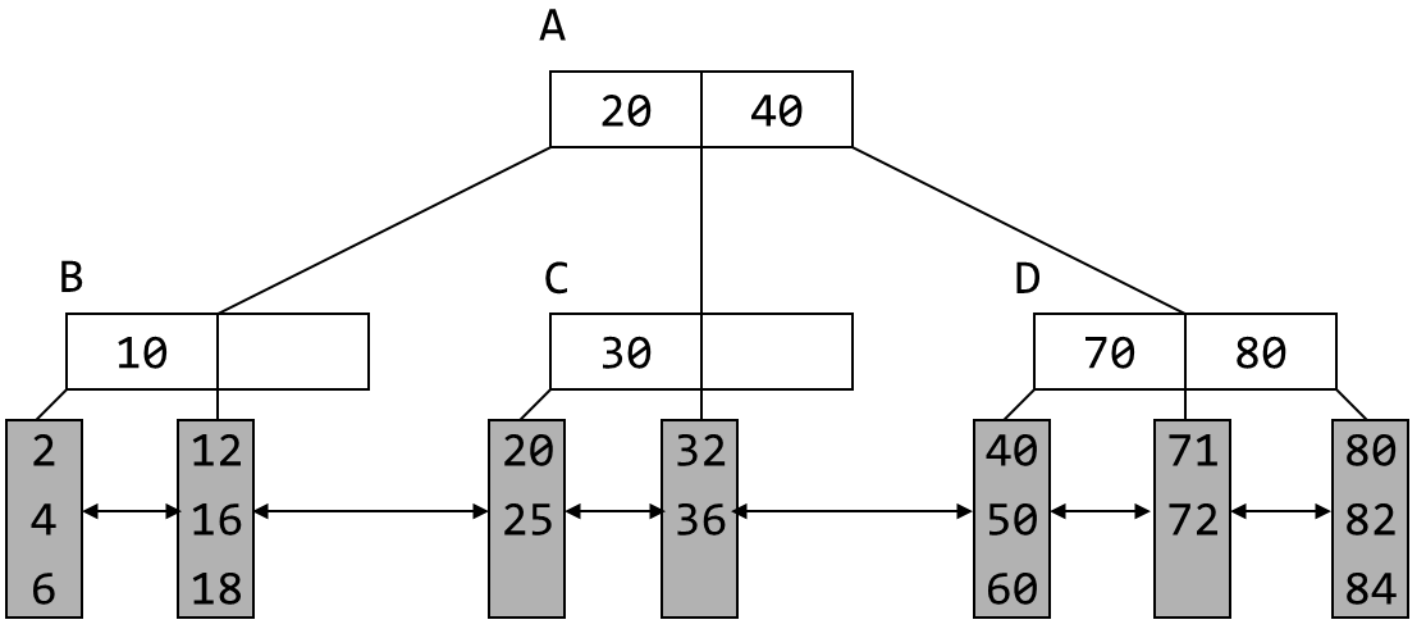
차이점 확인

B-tree node

$m, A_0, (E_1, A_1), \ldots, (E_m, A_m), \quad E_i < E_{i+1}, 1 \le i < m$

B+ Tree 에서의 탐색(Searching a B+ Tree)

- **두 가지 종류의 탐색이 가능**
  - 정확히 일치하는 값에 대한 검색
  - 범위 검색 (예: 20보다 작은 모든 데이터)

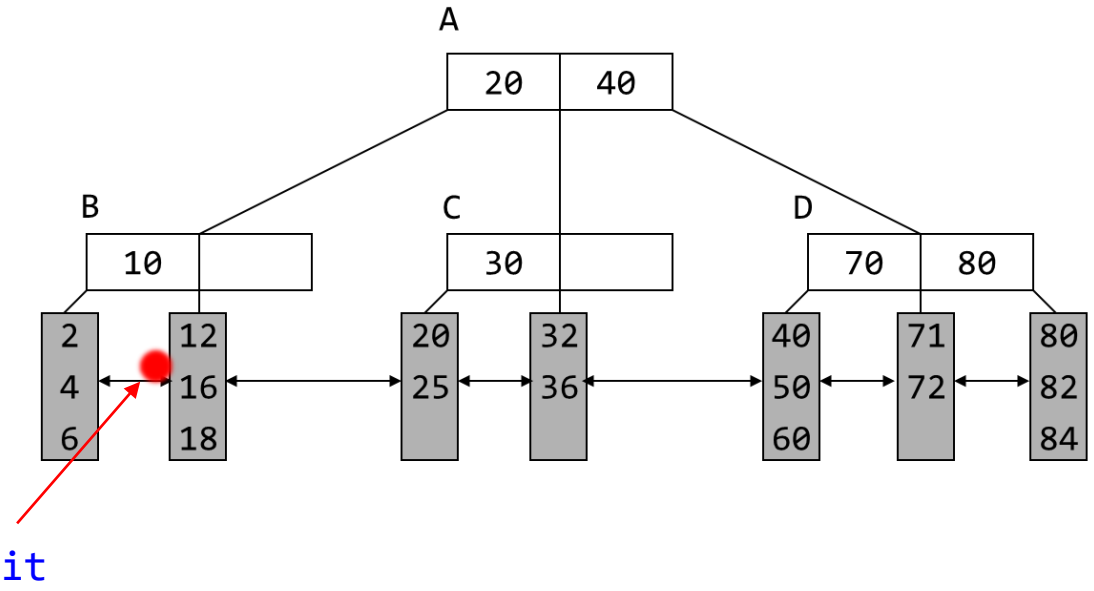


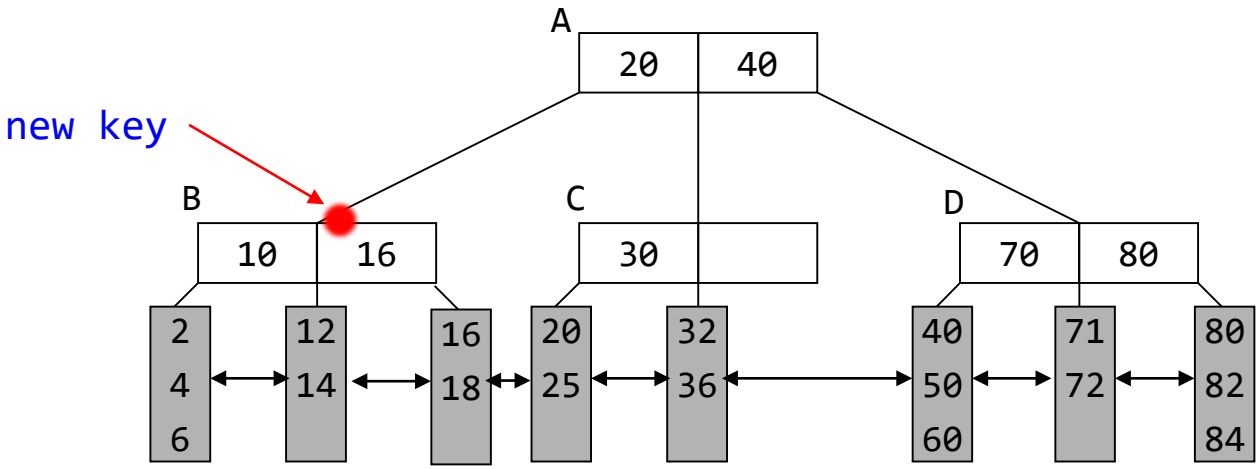
B+ Tree 에서의 삽입(Insertion into a B+ Tree)

- **데이터노드에 삽입 (분할) – B-Tree와 차이점**
  - 데이터 노드가 완전히 차면 가장 큰 키들을 가지고 있는 원소의 절반을 새로운 노드로 옮김
  - 분리된 새로운 노드에 대한 포인터와 가장 작은 키를 부모 인덱스 노드에 삽입
- **인덱스 노드 분할은 B-트리에서의 내부 노드 분할과 같음**

B+ Tree 에서의 삽입(Insertion into a B+ Tree)

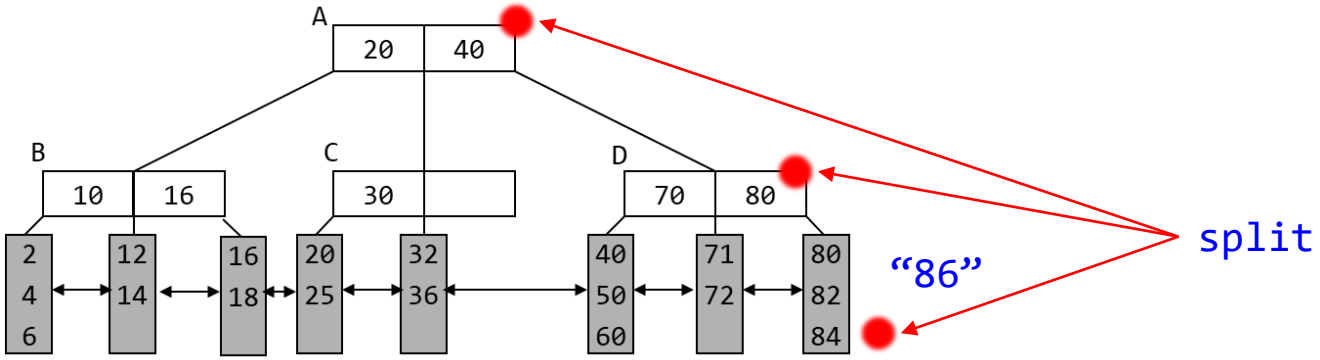
(a) “14” 삽입

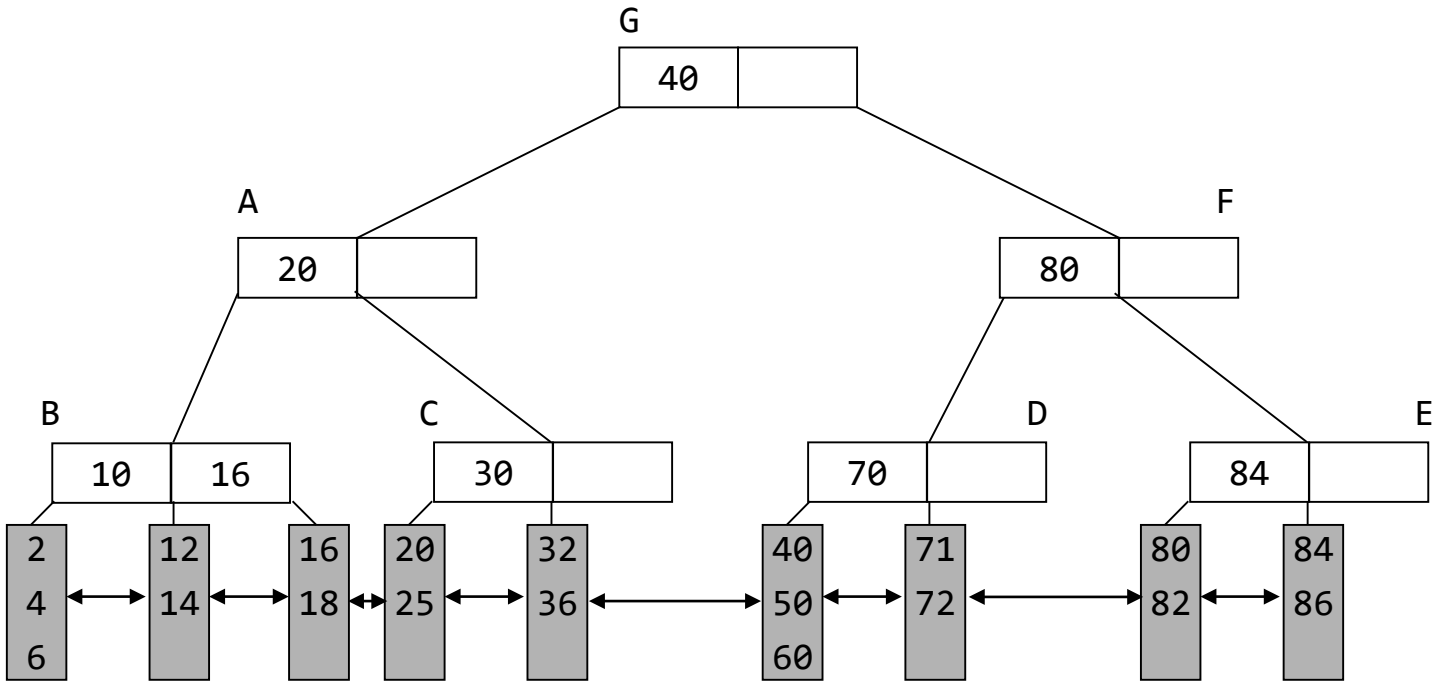




B+ Tree 에서의 삽입(Insertion into a B+ Tree)

(b) “86” 삽입



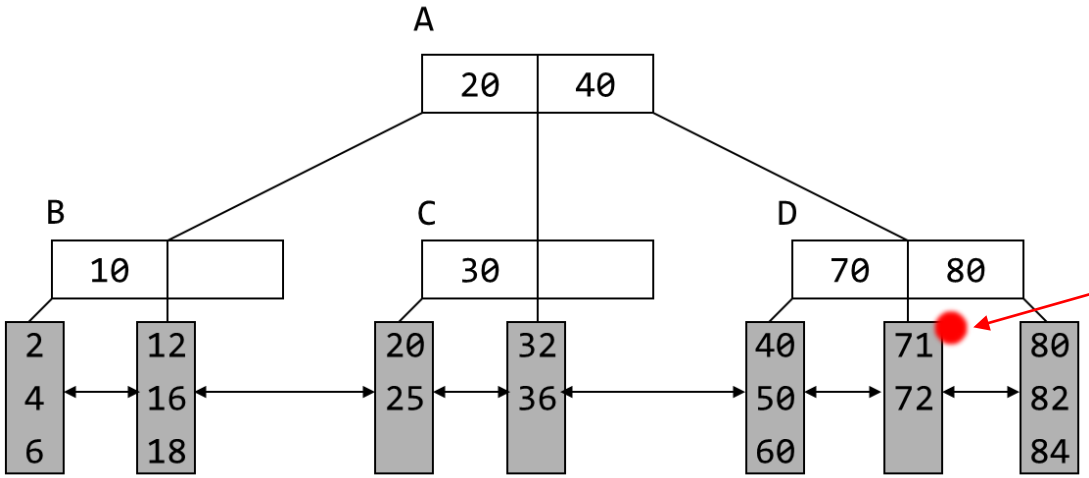


B+ Tree 에서의 삭제(Deletion from a B+ Tree)

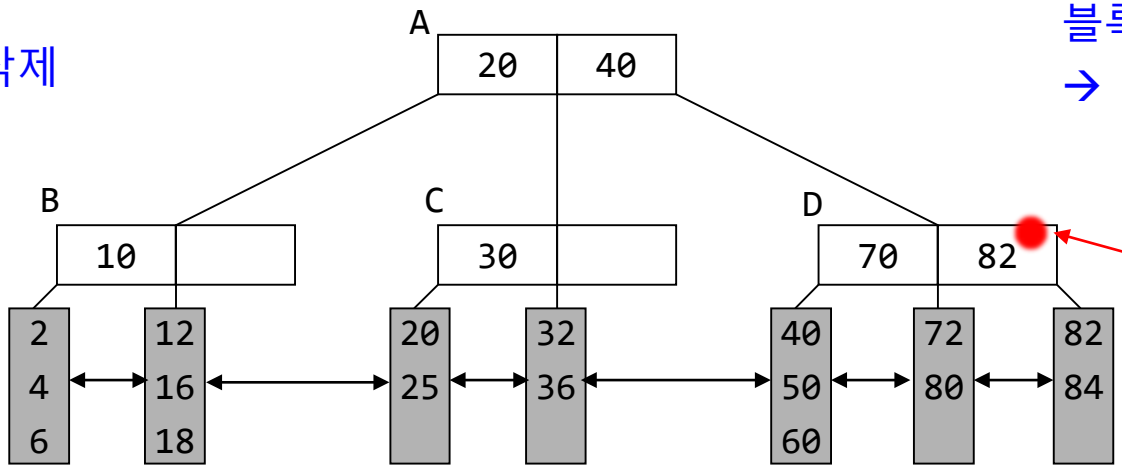
- **데이터 노드의 최소 원소 수가 부족하지 않은 경우**
  - 변경된 데이터 노드는 디스크에 기록
  - 인덱스 노드는 변경되지 않음

- **데이터 노드의 최소 원소 수가 부족한 경우**
  - 가까운 형제 노드에게 원소를 빌려옴
  - 그에 따른 부모 노드(인덱스 노드)의 해당 키 값을 변경
  - 형제 노드가 원소를 빌려줄 수 없는 경우 부모와 병합

B+ Tree 에서의 삭제(Deletion from a B+ Tree)

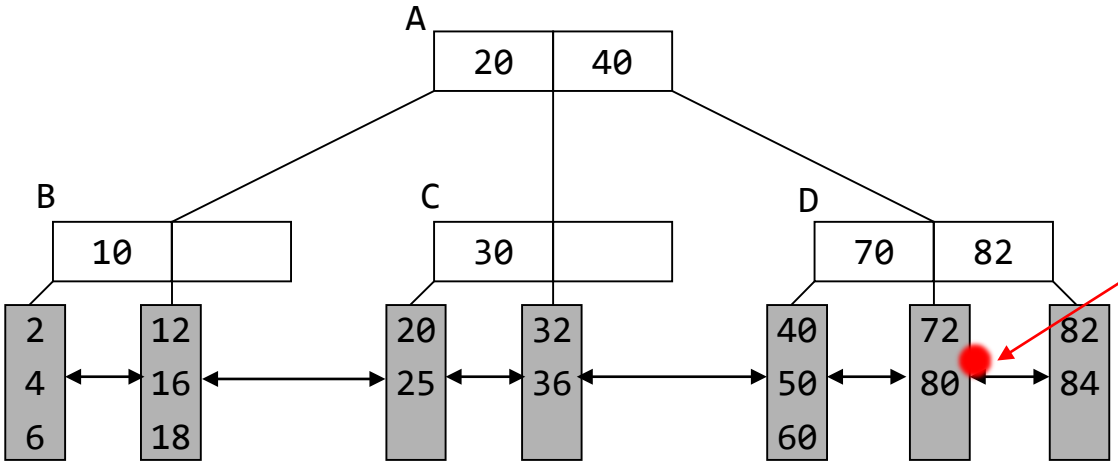


(a) "71" 삭제



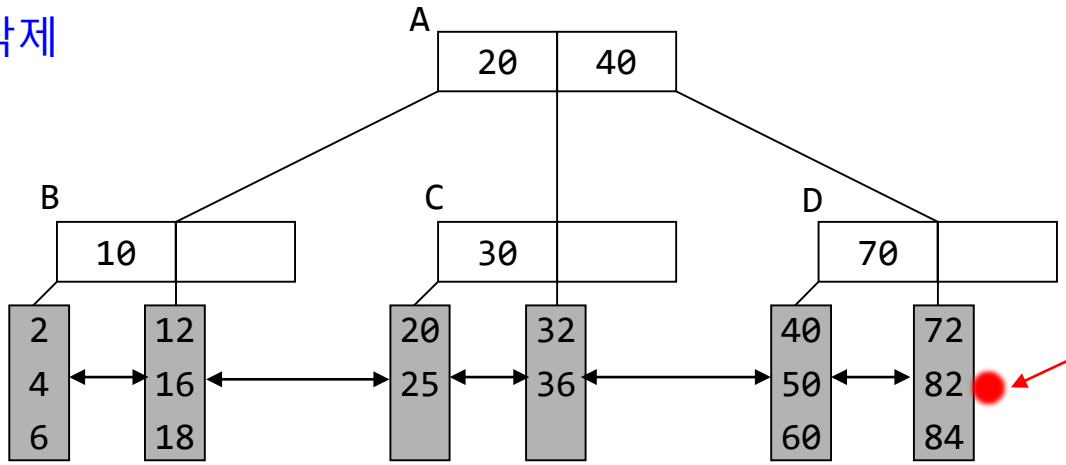
블록 원소 부족
→ sibling에서 빌림
오른쪽에서 빌린다고 가정

B+ Tree 에서의 삭제(Deletion from a B+ Tree)



"삭제"

(b) "80" 삭제

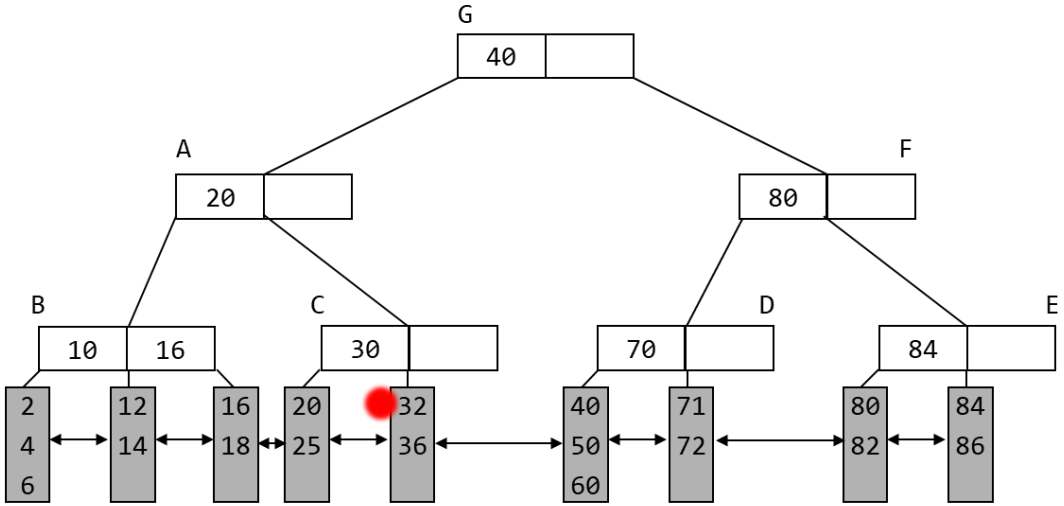


블록 원소 부족
→ sibling에서 빌림
→ 원소 부족
→ 부모와 병합

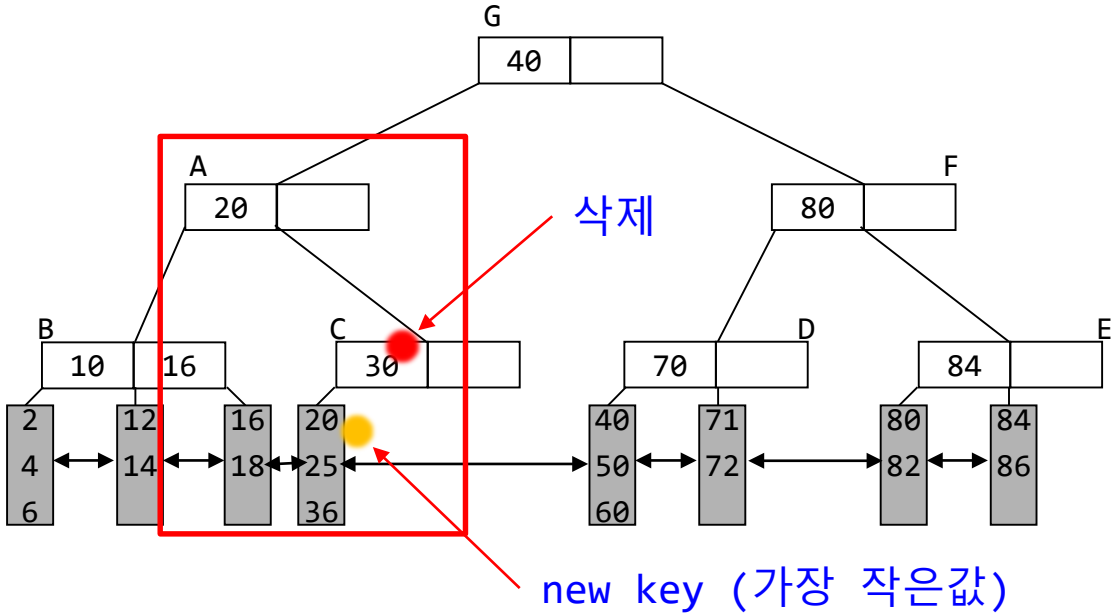
"부모와 병합"

삭제 예제(Deletion Example)

"32" 삭제"



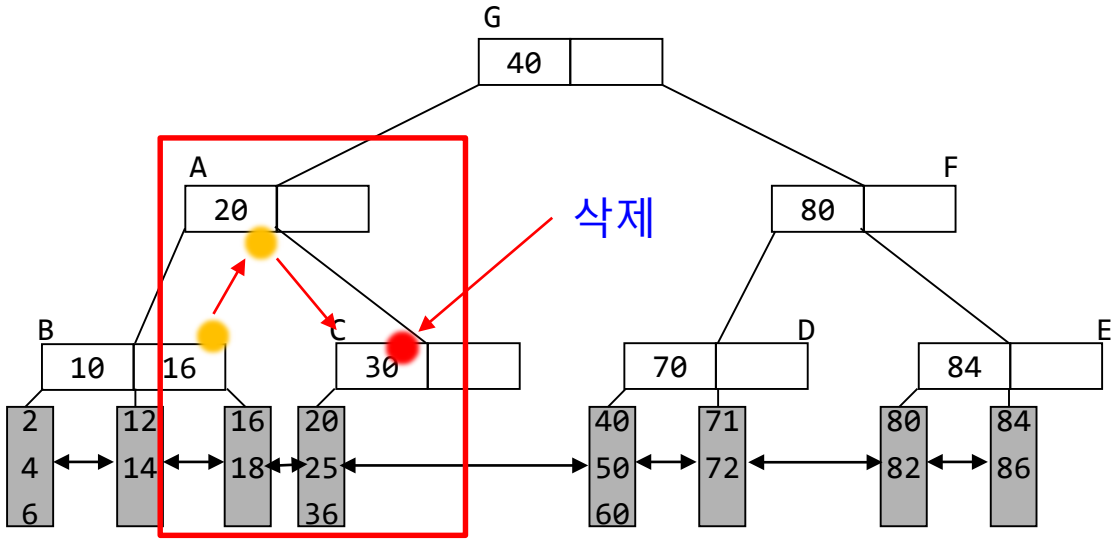
merge

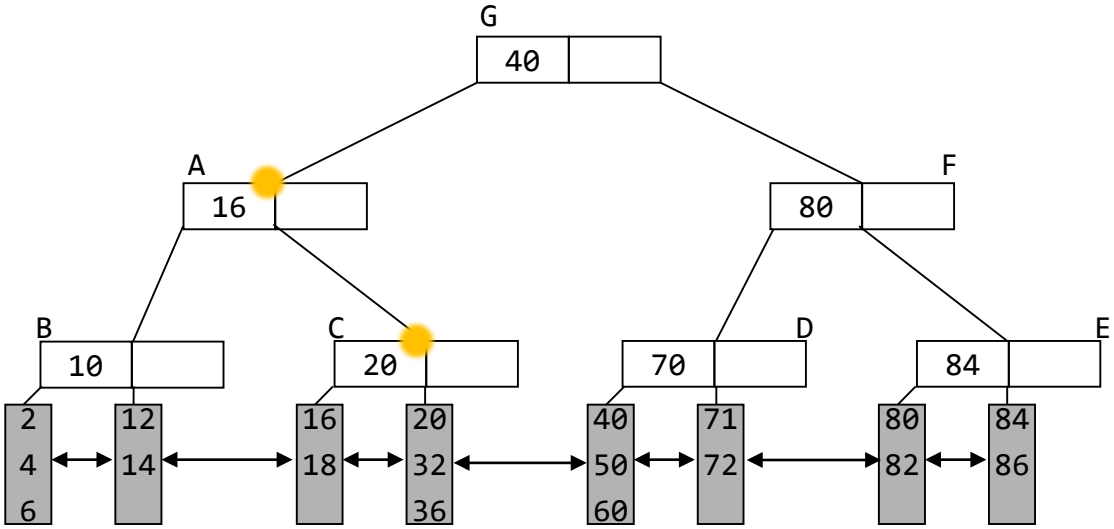


삭제 예제(Deletion Example)

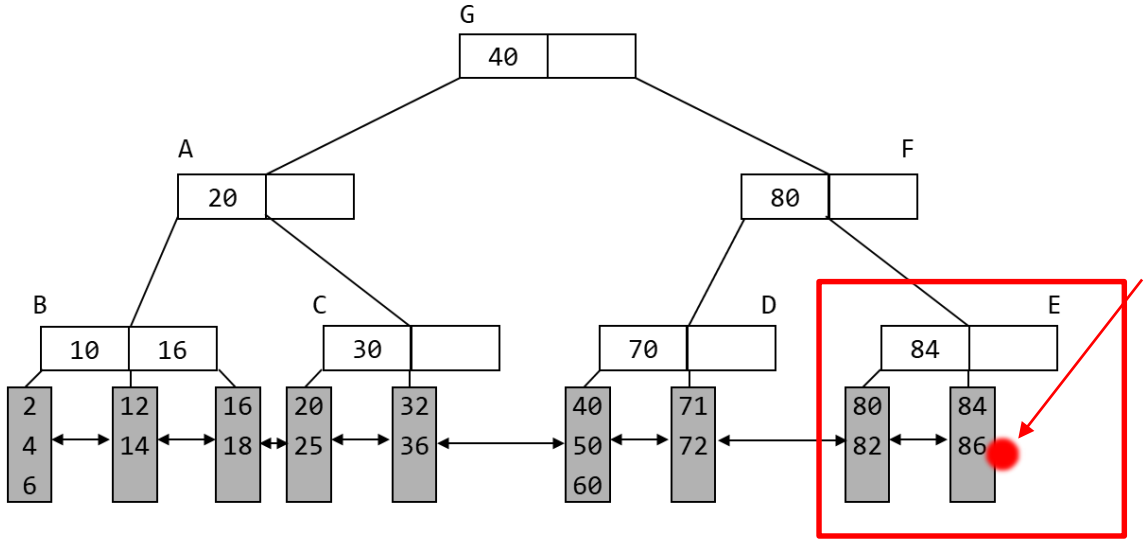
merge

이동



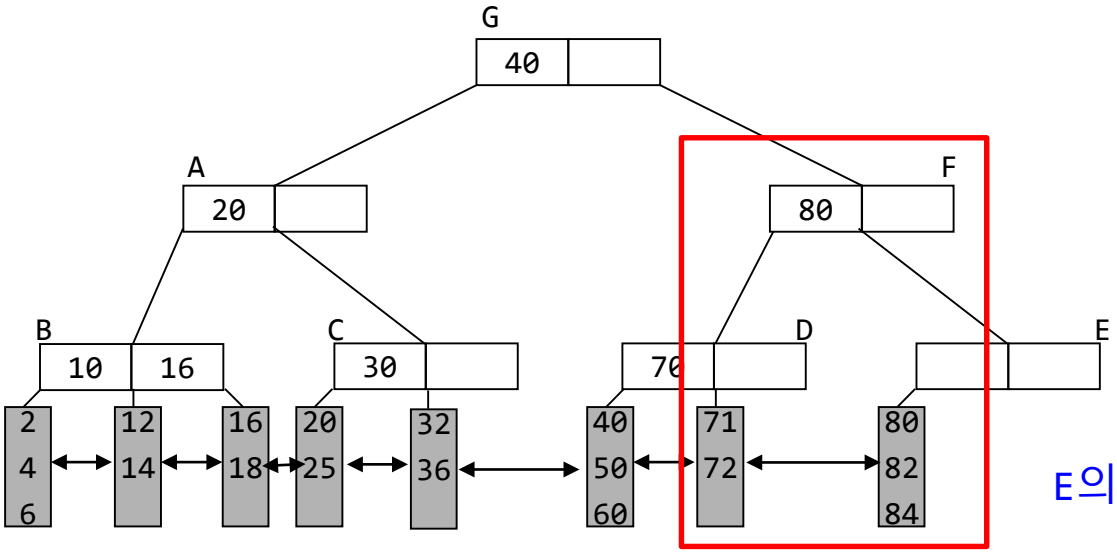


삭제 예제(Deletion Example)



"86" 삭제"

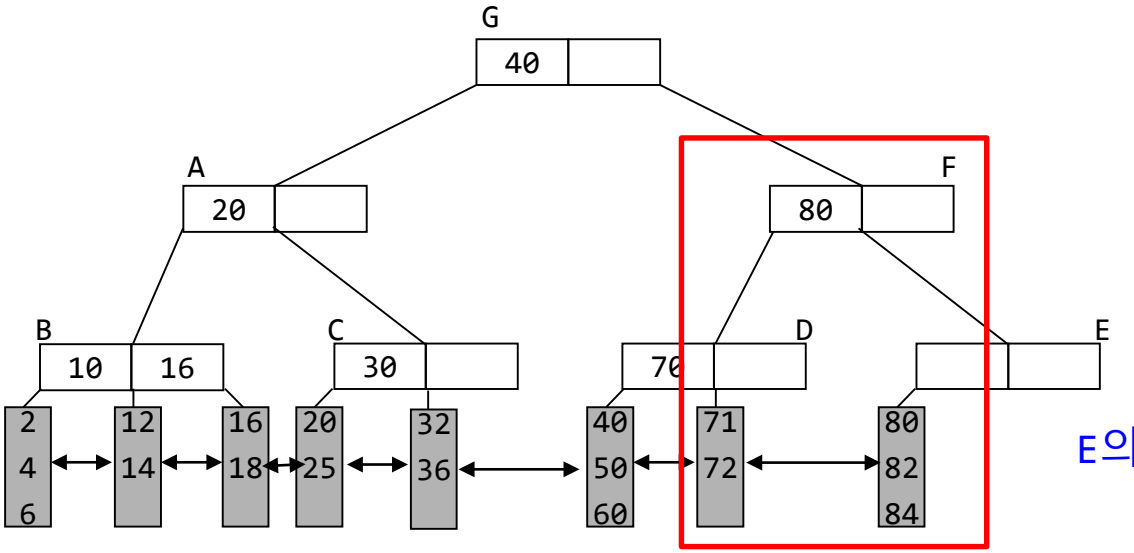
merge



merge

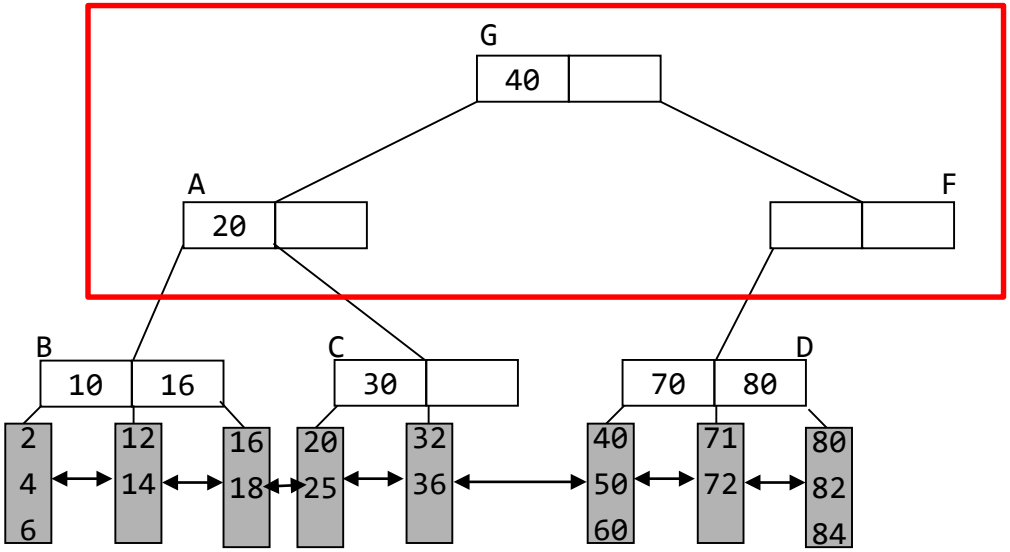
E의 원소가 부족하게 됨

삭제 예제(Deletion Example)



merge

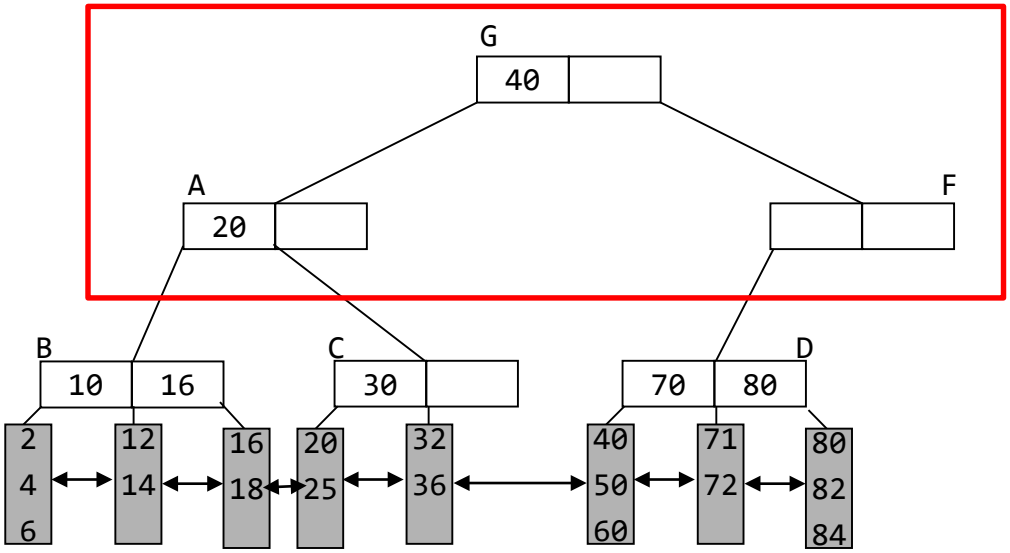
E의 원소가 부족하게 됨



F의 원소가 부족하게 됨

merge

삭제 예제(Deletion Example)



F의 원소가 부족

merge