

Database Systems

Lecture12 – Chapter 14: Indexing

Beomseok Nam (남범석)
bnam@skku.edu

Indexing

- Indexing is used to speed up access to data.
Primary key 이면 키와 다른
- **Search Key** – a set of attributes used to look up records
- **Index file** consists of **index entries** of the form

search-key	pointer (or value)
------------	-----------------------

- Index is typically much smaller than the original table
- Two basic kinds of indices:
 - **Ordered indices:** keys are stored in sorted order
 - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.

$lb < key < ub \Rightarrow$ Range query ; ordered
To cover bound
 $key = v \Rightarrow$ point query ; Hash

Index Evaluation Metrics

- Access types supported efficiently. E.g.,
 - Point query (i.e., $\text{key} = \text{value}$)
 - Range query (i.e., $\text{low} < \text{key} < \text{high}$)
- Access time *Search*
- Insertion time
- Deletion time
- Space overhead

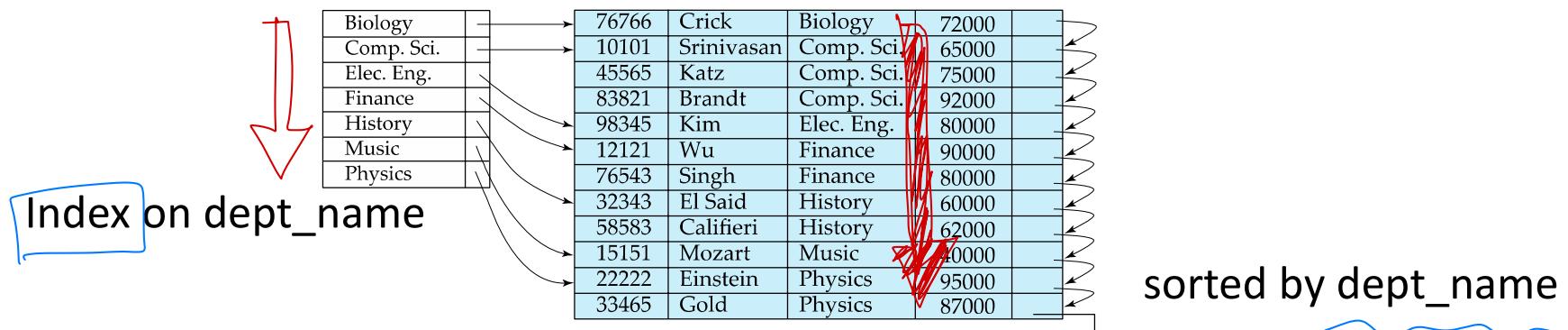
Ordered Indices

- In an **ordered index**, keys are stored sorted

- Primary index (clustered index)**

파일의 row 순서와 같음

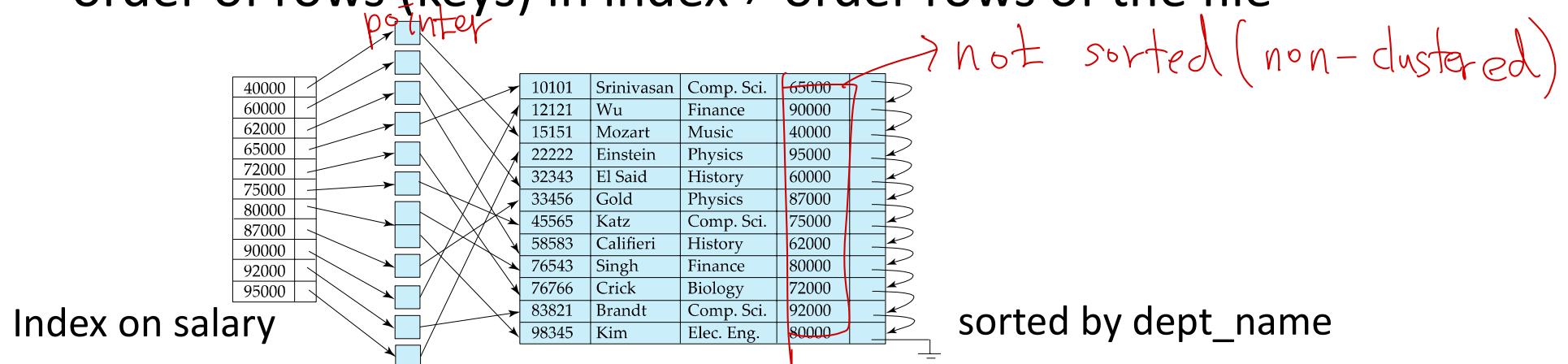
- order of rows (keys) in index = order rows of the file



- Secondary index (non-clustered index)**

pointer³

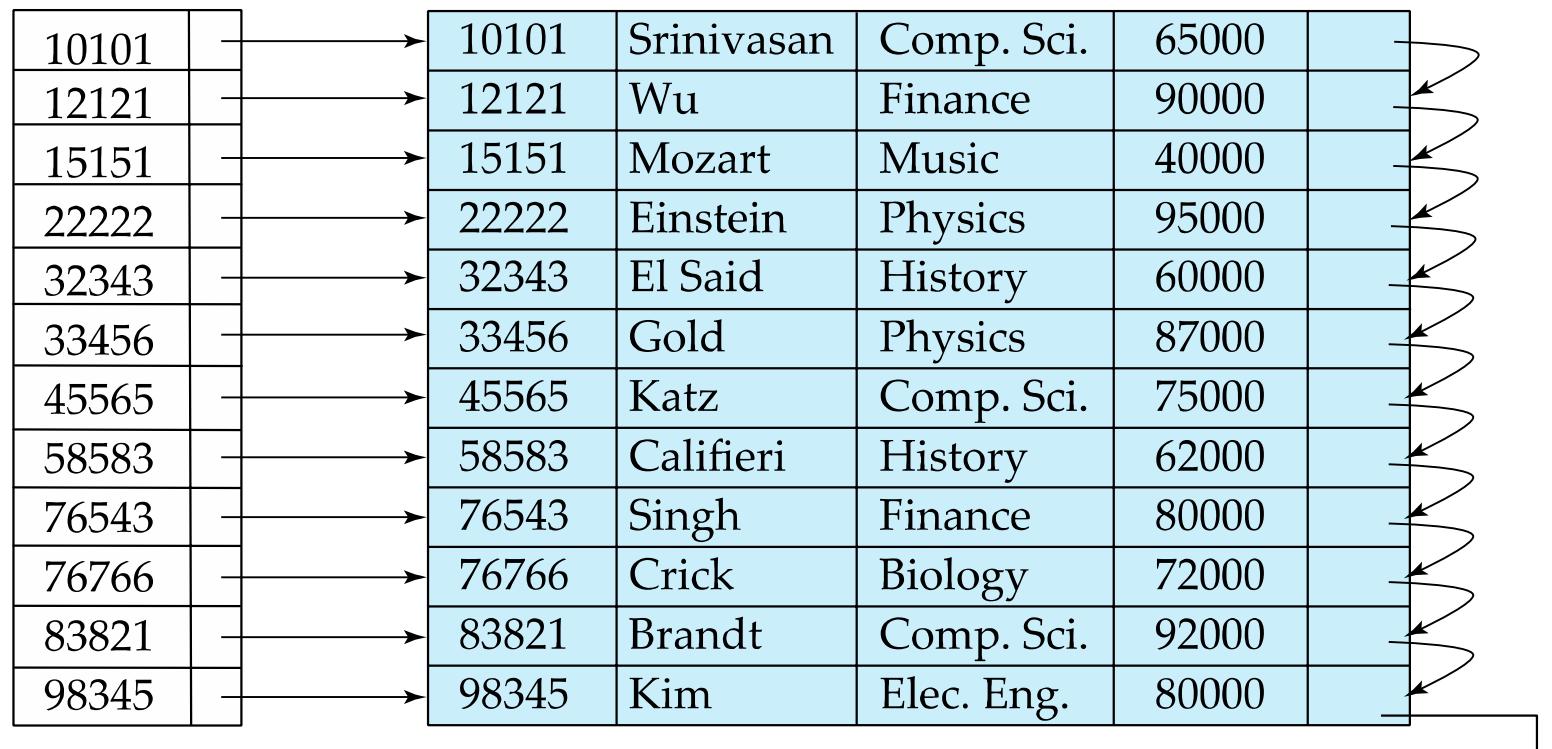
- order of rows (keys) in index ≠ order rows of the file



Dense Index Files

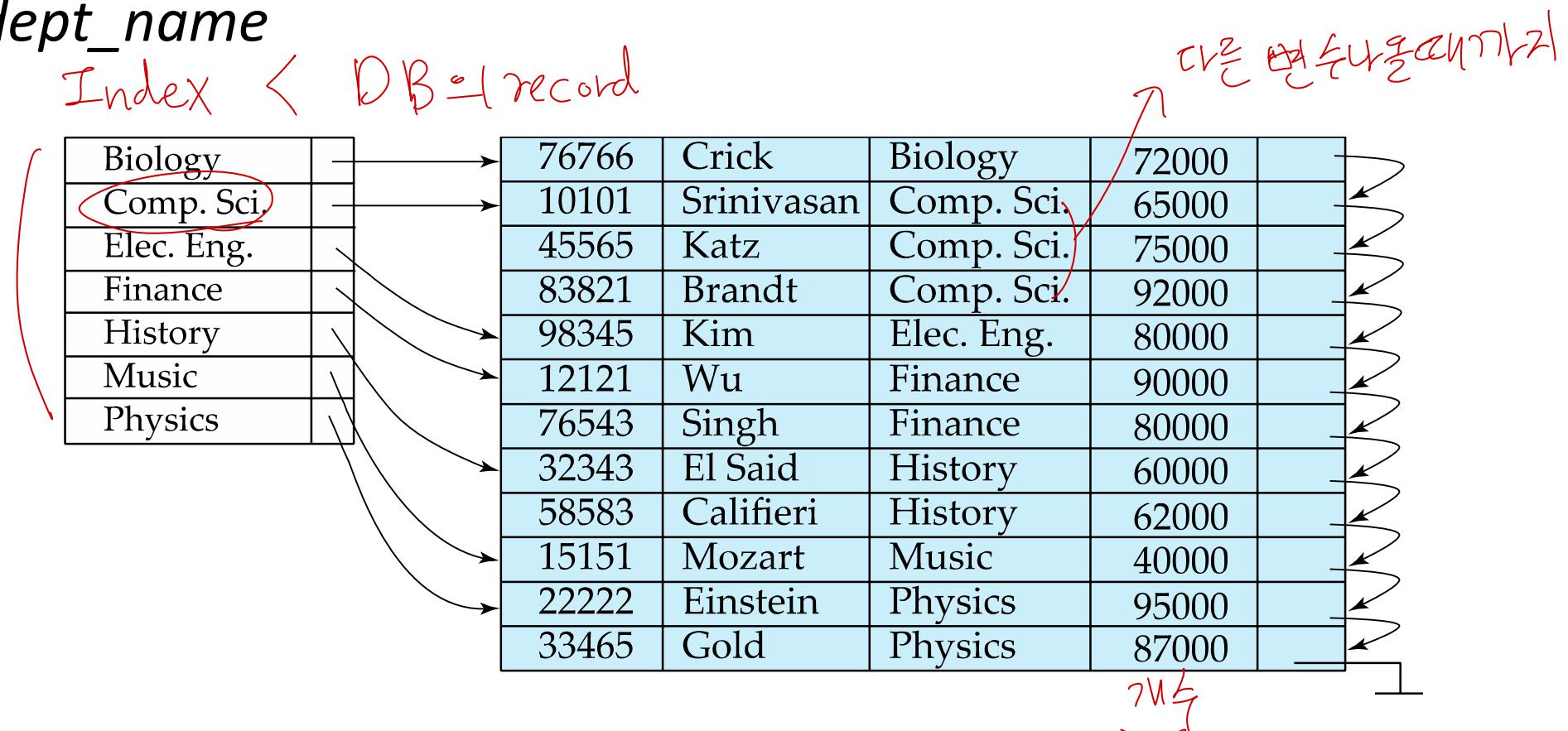
- **Dense index** — Index for every search key in the data file
- E.g. index on *ID*

백색함



Dense Index Files (Cont.)

- **Dense index** — Index for every search key in the data file
- Eg.) Dense index on *dept_name*, with records sorted on *dept_name*

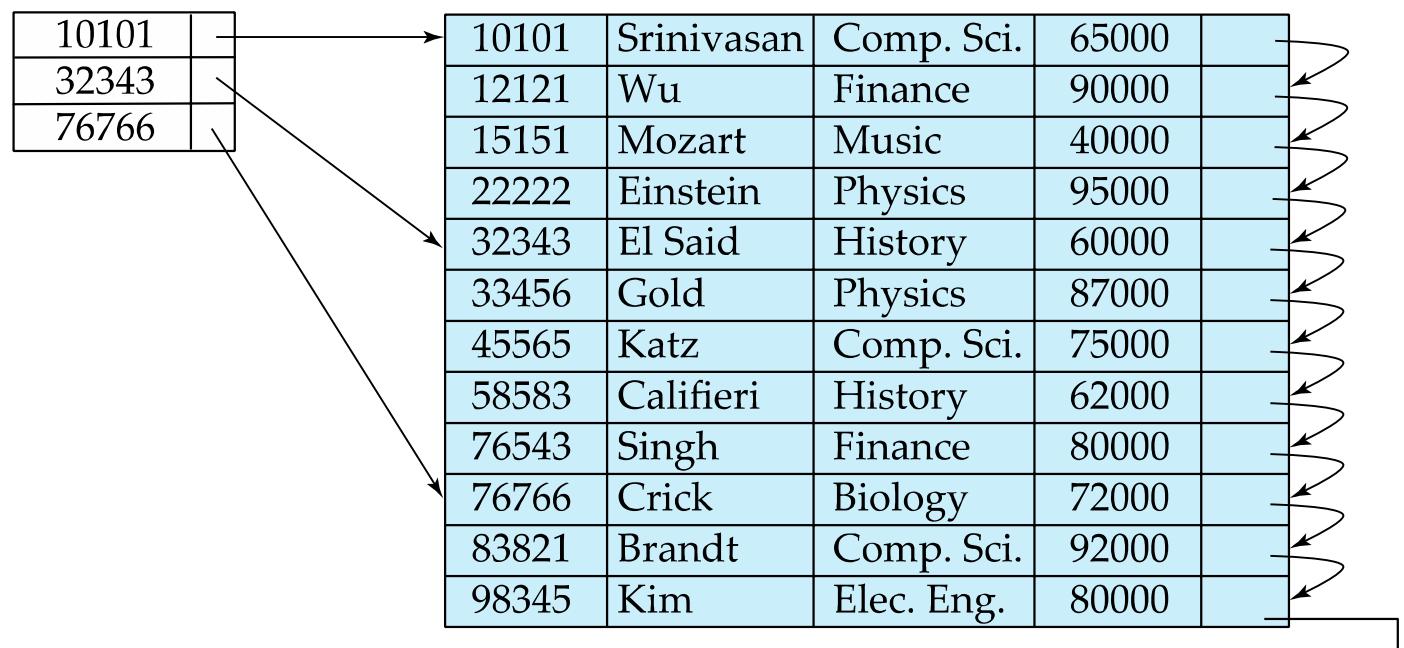


clustered (primary) Index

non-clustered (secondary) Index = Table
record

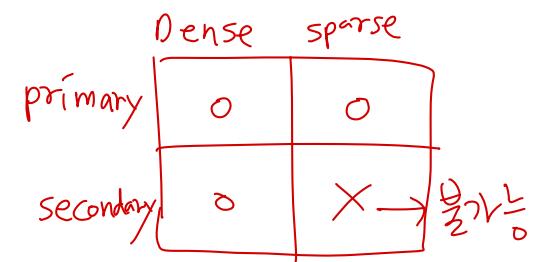
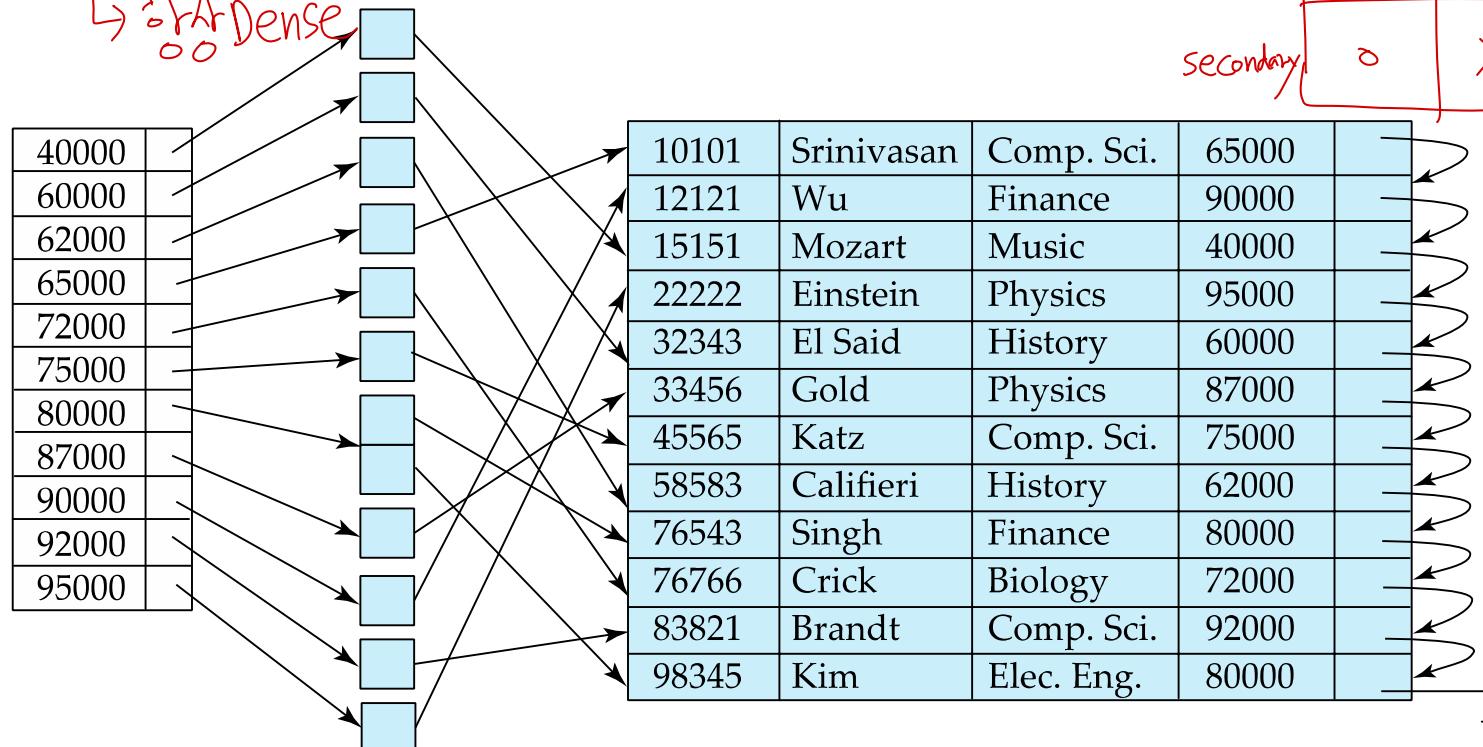
Sparse Index Files

- **Sparse Index:** index only some search-key values.
• Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value K ,
 - Find index record with largest search-key value $< K$
 - Scan file starting at the record to which the index record points



Secondary Indices Example

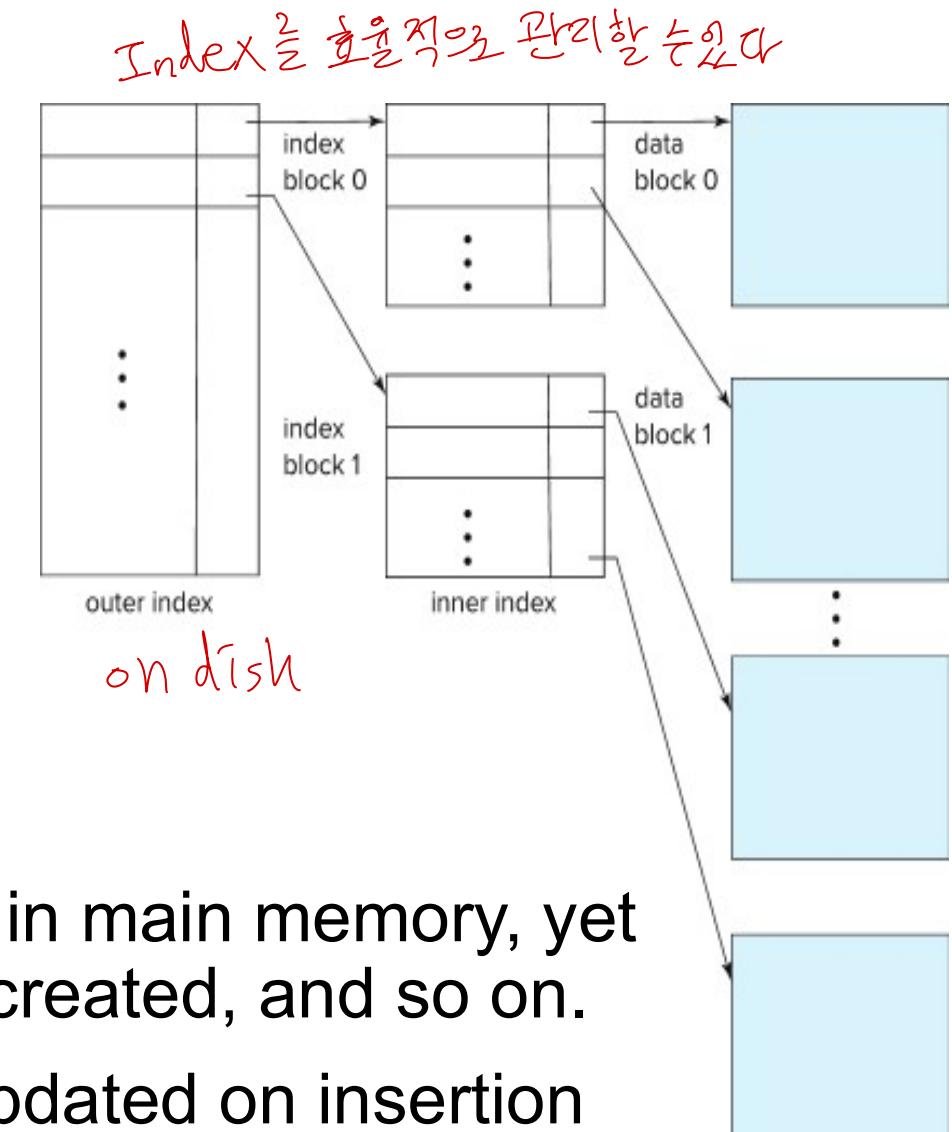
- Secondary index on salary field of instructor



- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

Multilevel Index

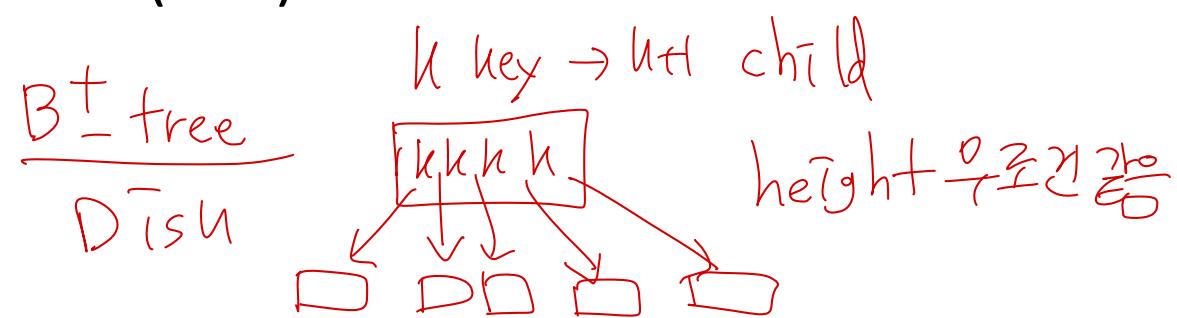
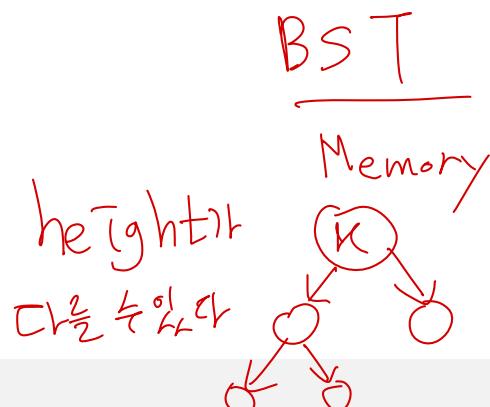
- If index does not fit in DRAM, search becomes expensive.
- Solution: Multilevel Indexing
 - DRAM-resident top-level index (outer index)
 - Disk-resident lower-level index (inner index)
- If outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.



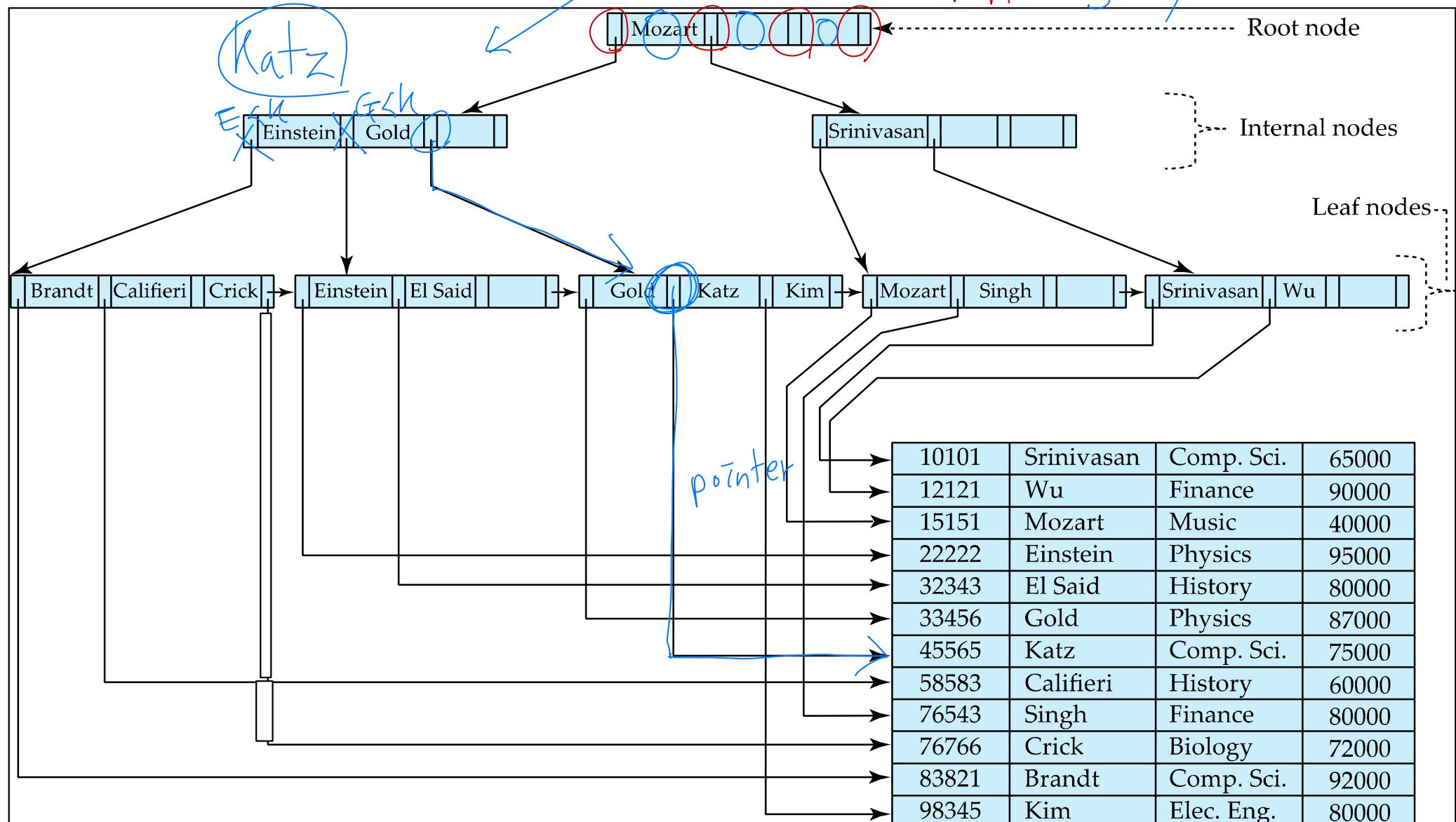
B+-Tree Index

- B⁺-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
 - **Balanced tree**
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children. ($n=$ order, degree, branch factor)
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- Special cases:
 - If the root is not a leaf, it has at least 2 children.
 - If the root is a leaf (i.e., there are no other nodes), it can have between 0 and $(n-1)$ values.



B⁺-Tree



B⁺-Tree Node Structure

- Node structure pivot key : 가운데 되는 키

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	-----	-----------	-----------	-------

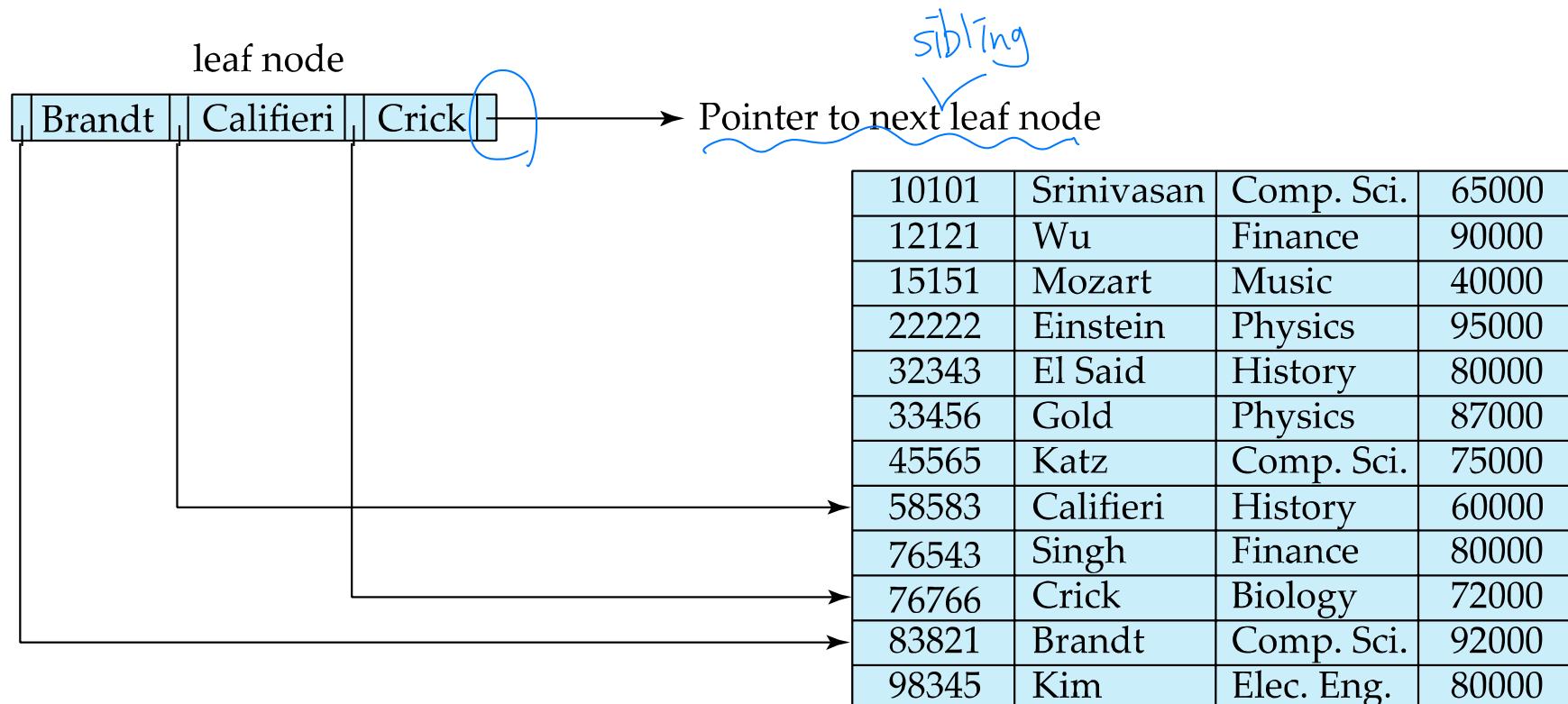
- K_i are the search-key values
- P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \dots < K_{n-1}$$

Note: Let's assume no duplicate keys

Leaf Nodes in B⁺-Trees

- Properties of a leaf node:
- Pointer P_i points to a record with search-key value K_i ($0 < i < n$)
- If L_i, L_j are leaf nodes and $i < j$, L_i 's search-key values are less than or equal to L_j 's search-key values
- P_n points to next leaf node in search-key order



Internal Nodes in B⁺-Trees

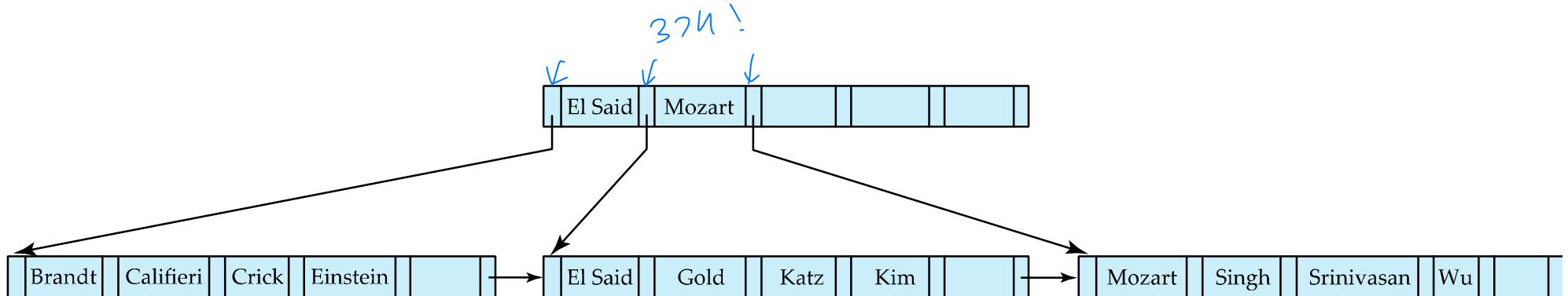
- Non leaf nodes form a multi-level sparse index on the leaf nodes.
- For a non-leaf node with m pointers:
 - All the keys in the subtree to which P_1 points are less than K_1
 - For $2 \leq i \leq n - 1$, all the keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
 - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}
 - General structure

P_1	K_1	P_2	\dots	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	---------	-----------	-----------	-------

Example of B⁺-tree

- B⁺-tree for *instructor* file ($n = 6$)

$$\lceil \frac{n}{2} \rceil = 3$$



- Leaf nodes must have between 3 and 5 values ($\lceil (n-1)/2 \rceil$ and $n - 1$, with $n = 6$).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil (n/2) \rceil$ and n with $n = 6$).
- Root must have at least 2 children.**

Observations about B⁺-trees

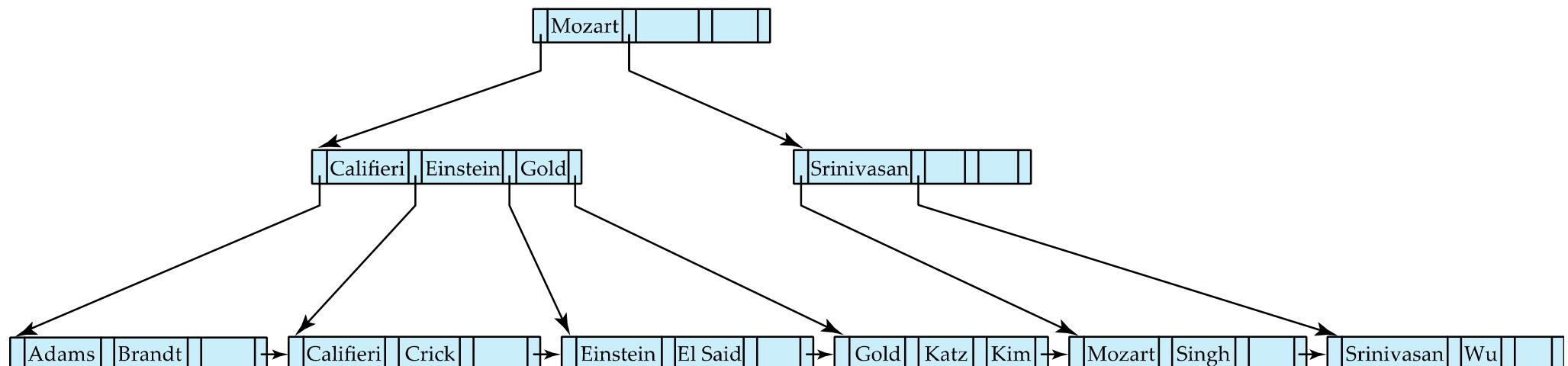
HDD 디자인의 원칙 (random access)

- Since nodes are connected by pointers, “logically” close blocks need not be “physically” close.
디스크의 계산하는 위치와 실제 위치와는 다르다
- The non-leaf levels of the B⁺-tree form a hierarchy of sparse indices.
- The B⁺-tree contains a relatively small number of levels
 - Level below root has at least $2 \lceil n/2 \rceil$ values
 - Next level has at least $2 \lceil n/2 \rceil * \lceil n/2 \rceil$ values
 - .. etc. *전체 record 개수*
- If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
노드에 있는 자식의 최소값
- Efficient search: search cost = tree height = $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
(P)

Queries on B⁺-Trees

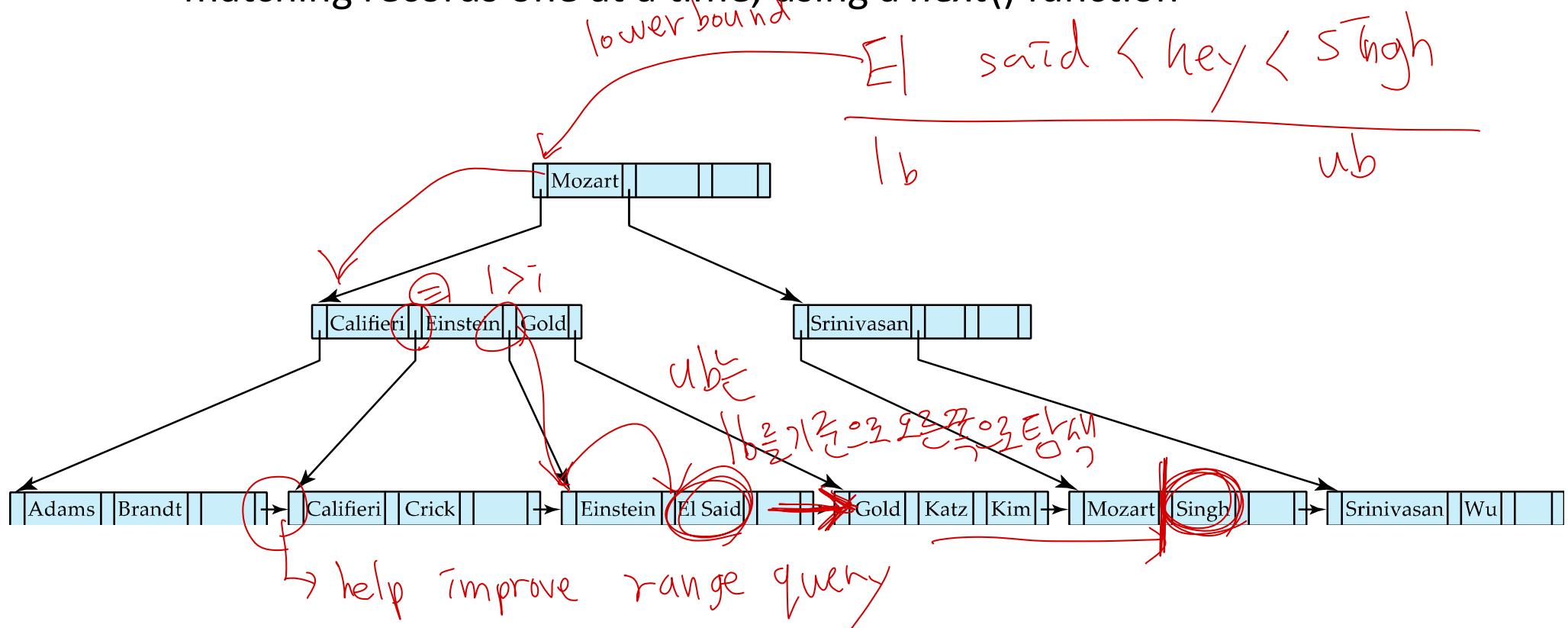
function *find(v)*

1. *C=root*
2. **while** (*C* is not a leaf node)
 1. Let *i* be least number s.t. $V \leq K_i$
 2. **if** there is no such number *i* **then**
 3. Set *C = last non-null pointer in C*
 4. **else if** (*v = C.K_i*) Set *C = P_{i+1}*
 5. **else set** *C = C.P_i*
3. **if** for some *i*, $K_i = V$ **then** return *C.P_i*
4. **else** return null /* no record with search-key value *v* exists. */



Queries on B+-Trees (Cont.)

- **Range queries** find all records with search key values in a given range
 - Read textbook for details of **function** $\text{findRange}(lb, ub)$ which returns set of all such records
 - Real implementations usually provide an iterator interface to fetch matching records one at a time, using a $\text{next}()$ function

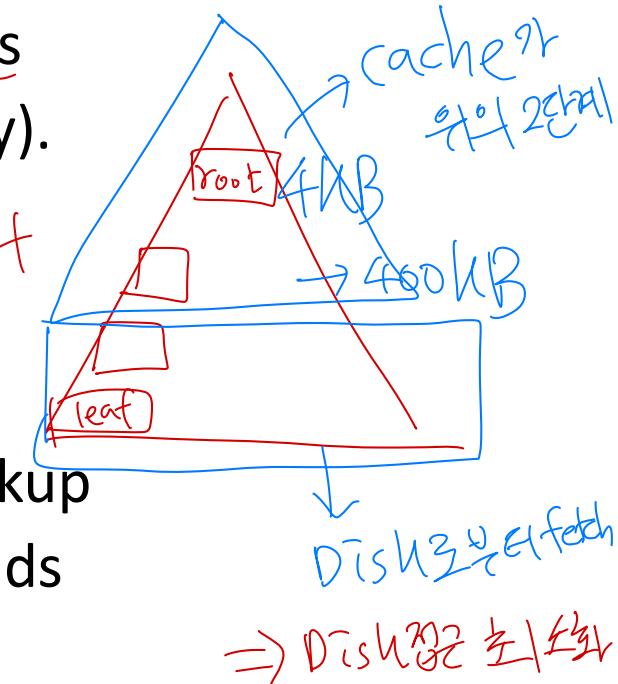


Queries on B+-Trees (Cont.)

- B+Tree node size = disk block size, typically 4 kilobytes
 - n is typically around 100 (40 bytes per index entry).

- With 1 million records and $n = 100$
 - Tree height = $\lceil \log_{n/2}(K) \rceil \rightarrow \log_{50}(1,000,000) = 4$
 - i.e., at most 4 B+Tree nodes are accessed in a lookup
 - Every node access (disk I/O) incurs ~20 milliseconds
 \rightarrow ~80 milliseconds

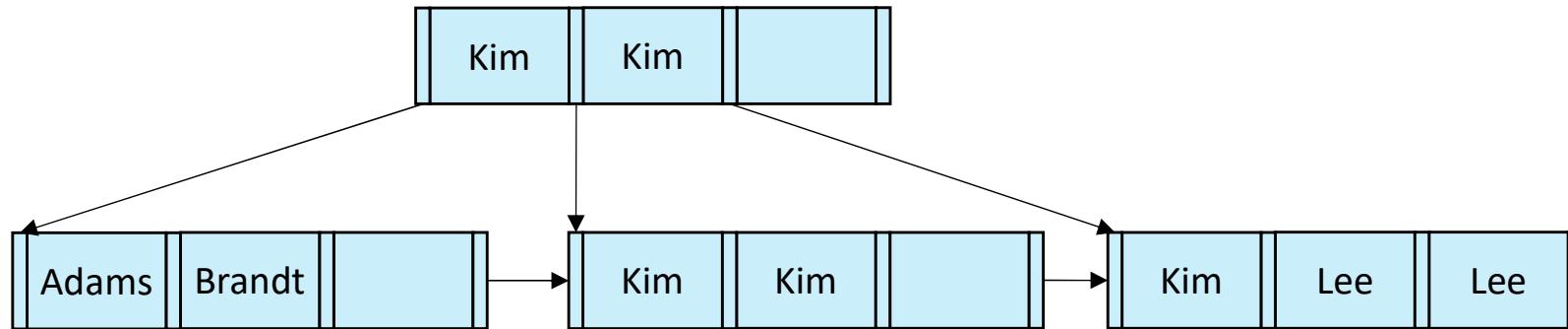
- Contrast this with a balanced binary search tree with 1 million key values
 - \rightarrow around 20 nodes are accessed in a lookup
 - \rightarrow ~400 milliseconds



Non-Unique Keys

- Ignore duplicates

중복 신경안해도 됨 \Rightarrow 앞에서 잘 처리함



- Alternatively, if a search key a_i is not unique, create an index on a composite key (a_i, A_p) to make the key unique
 - A_p could be a primary key, record ID, or any other attribute that guarantees uniqueness
 - Search for $a_i = v$ can be implemented by a range search on composite key, with range $(v, -\infty)$ to $(v, +\infty)$

row id가 같아 record의 Index가 같은
ID를 통해 중복 데이터 찾기

Updates on B⁺-Trees: Insertion

Assume record already added to the file. Let

- ptr be pointer to the record
- and let k be the search key of the record

1. Find the leaf node in which the search-key would appear
 1. If there is room in the leaf node, insert (k, ptr) pair
 2. Otherwise, split the node as discussed in the next slide, and propagate updates to parent nodes.

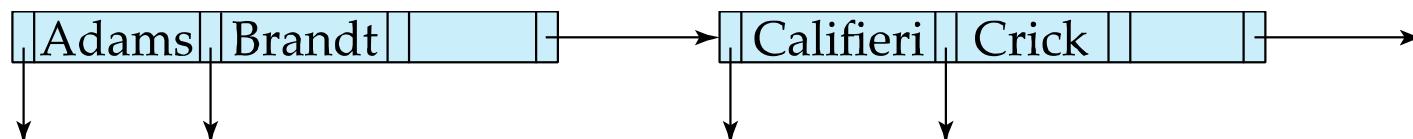
Updates on B⁺-Trees: Insertion (Cont.)

■ Splitting a leaf node L :

- Take the n (search-key, pointer) pairs in sorted order.
- Create a sibling node S .
- Move the second $\lceil n/2 \rceil$ pairs to S .
- Let k be the least key value in p . Insert (k, S) in the parent node P .
- If the parent is full, split it and **propagate** the split further up.

■ Splitting proceeds upwards till a node that is not full is found.

- In the worst case the root node may be split increasing the height of the tree by 1.



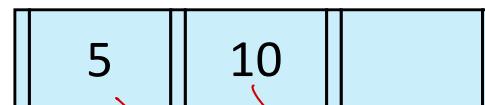
Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri, pointer-to-new-node) into parent

B⁺-Tree Insertion

- Insert 5



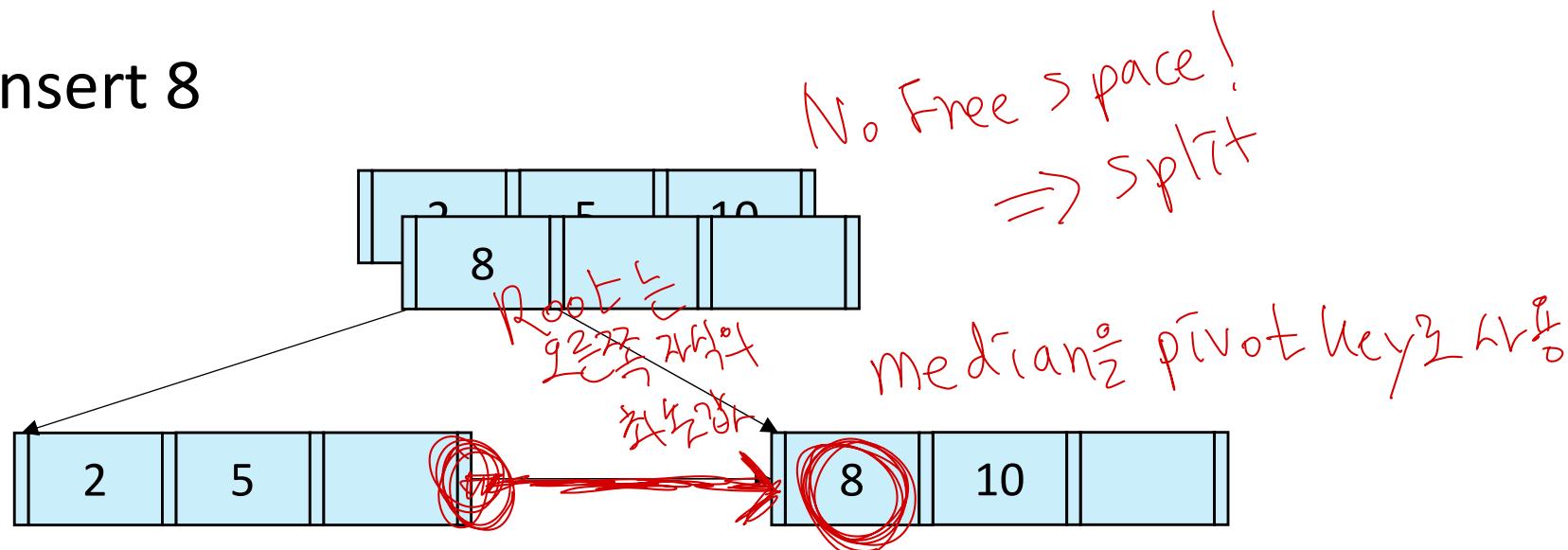
- Insert 10



- Insert 2

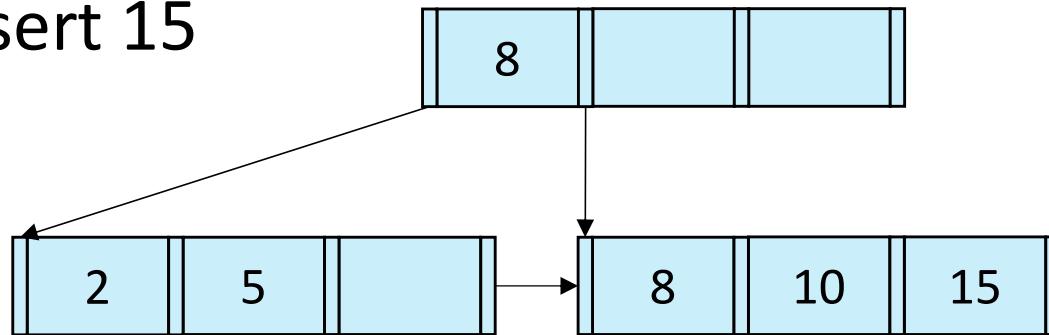


- Insert 8

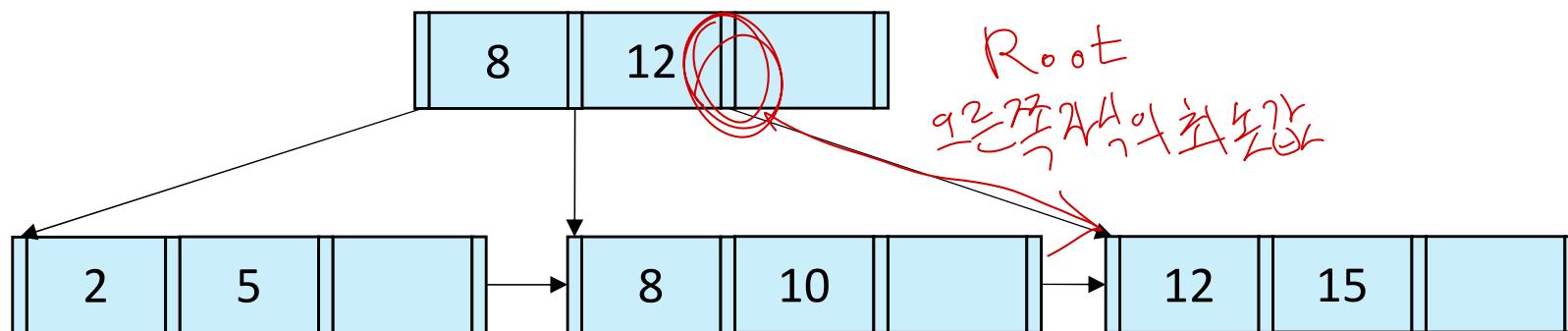


B⁺-Tree Insertion

- Insert 15

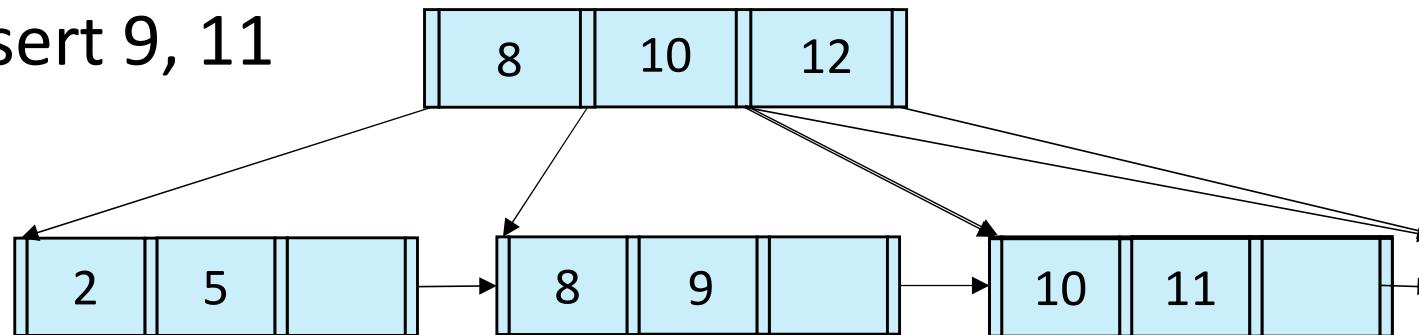


- Insert 12

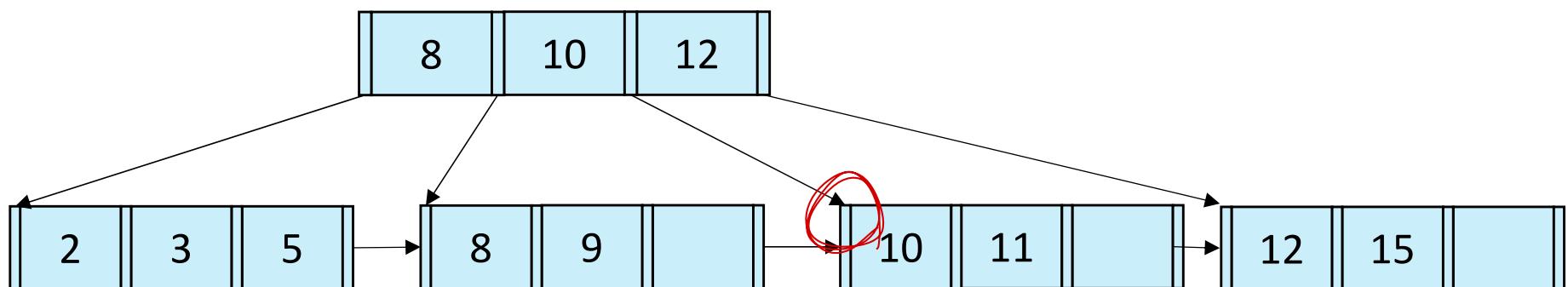


B⁺-Tree Insertion

- Insert 9, 11

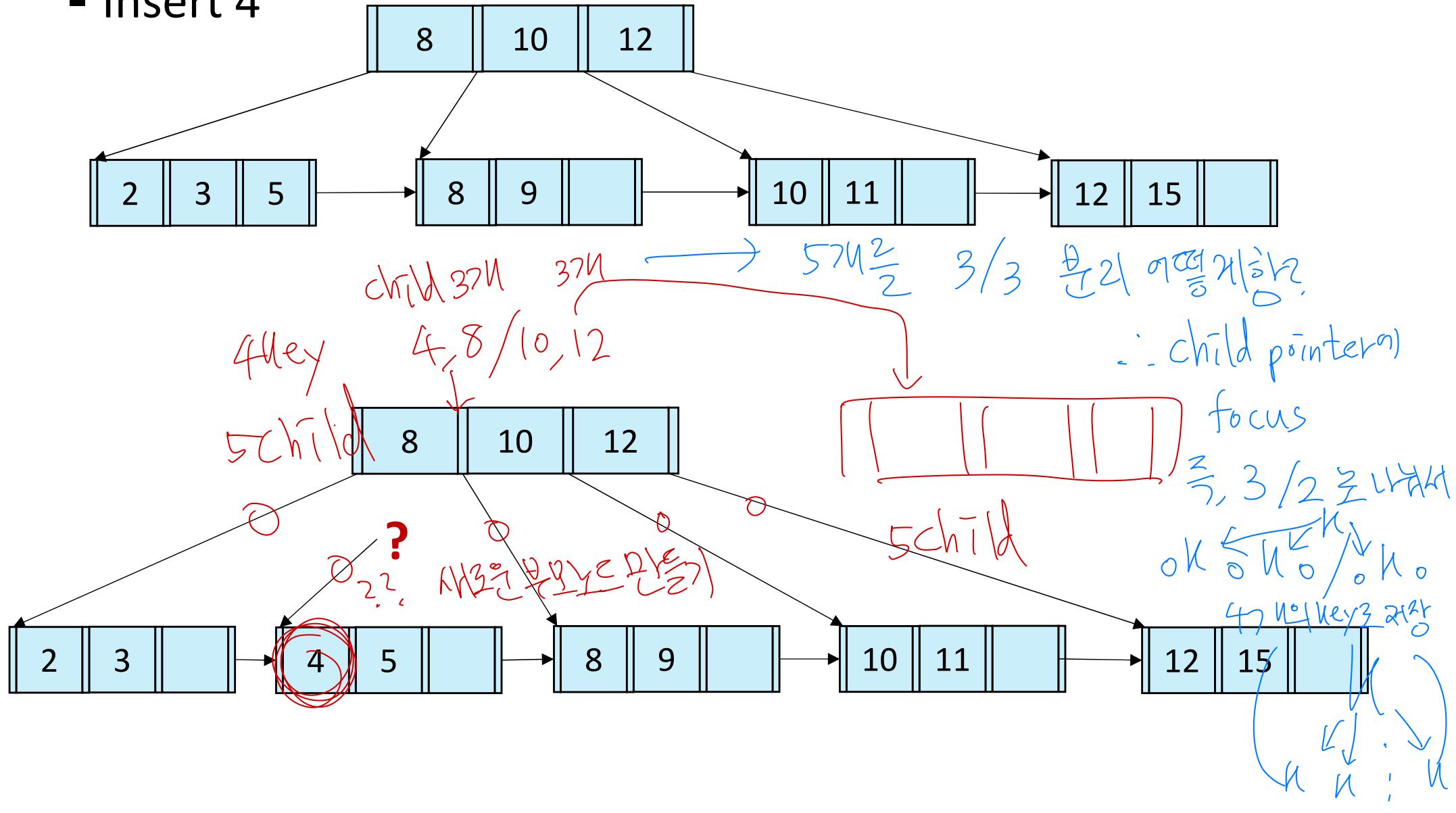


- Insert 3



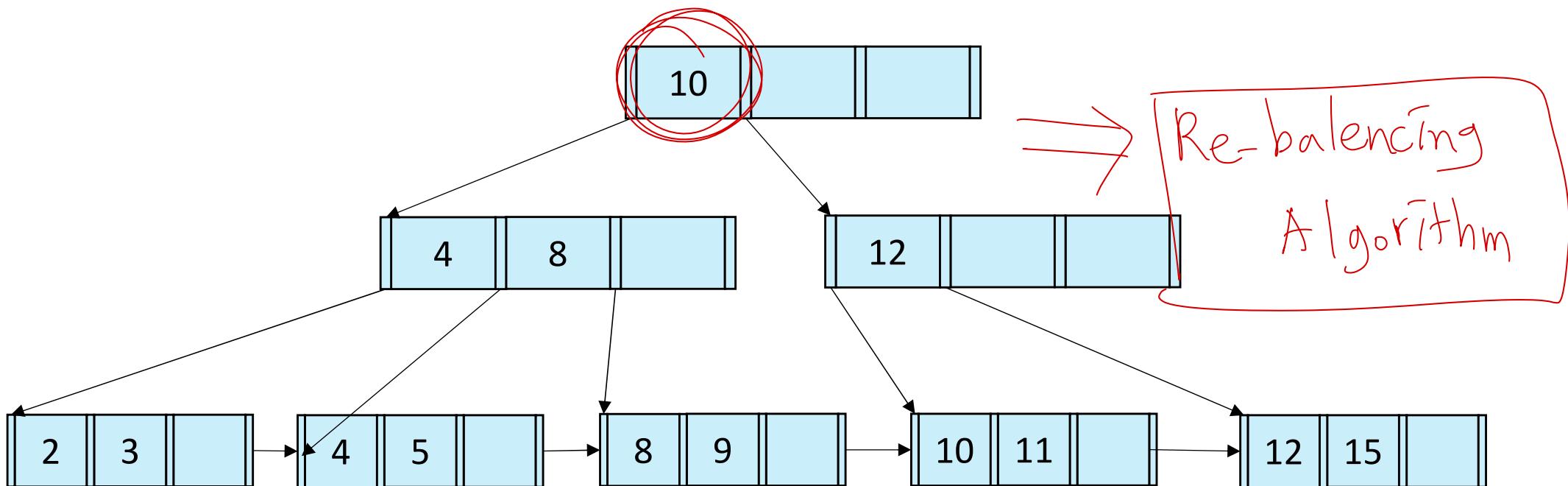
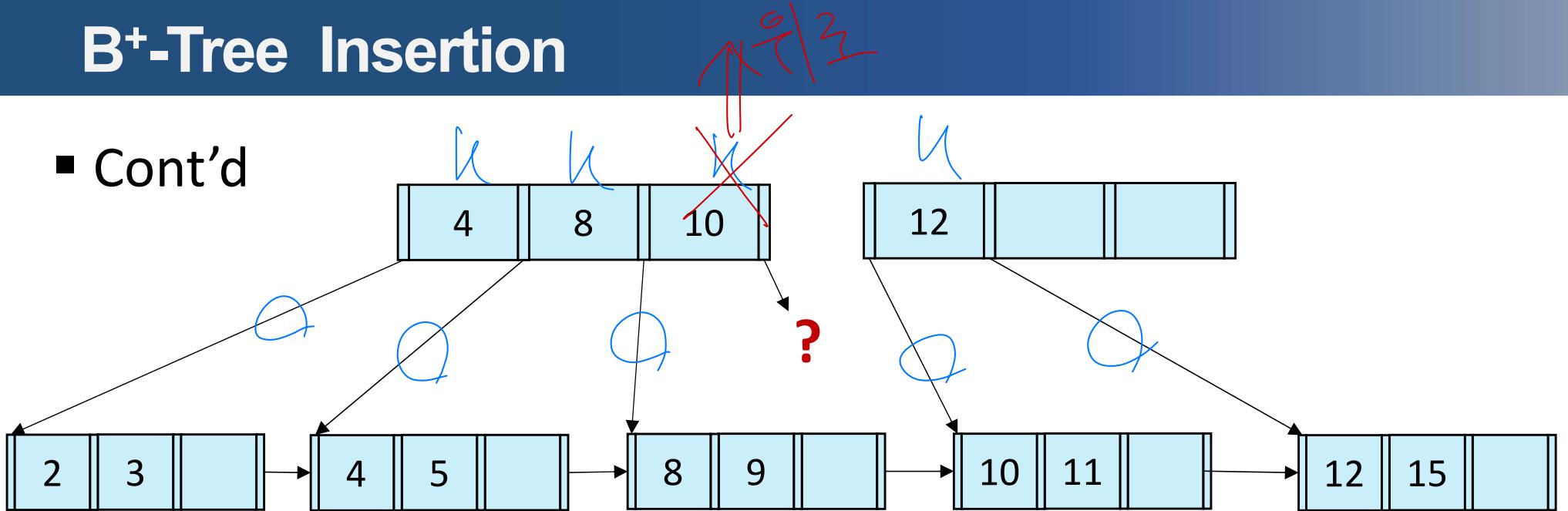
B+-Tree Insertion

■ Insert 4



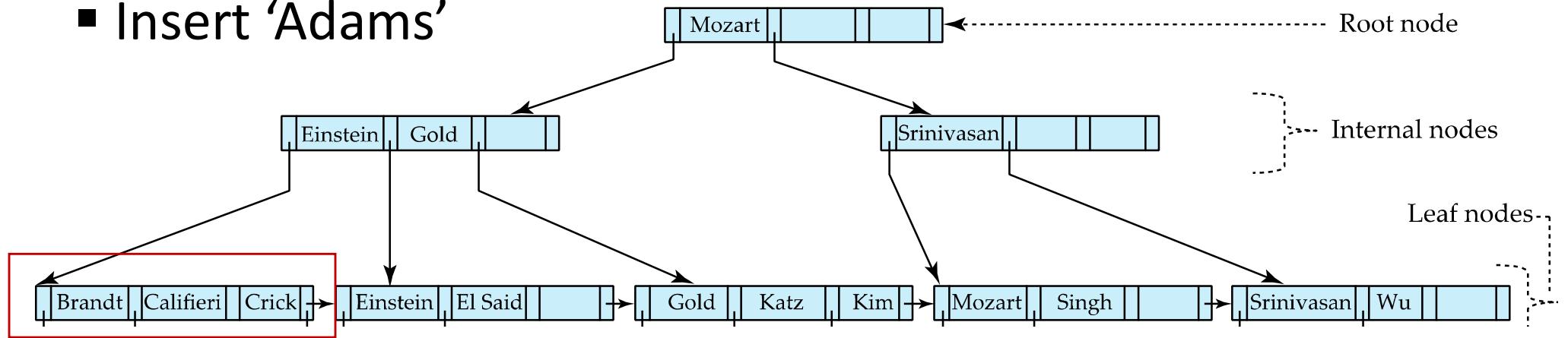
B⁺-Tree Insertion

- Cont'd

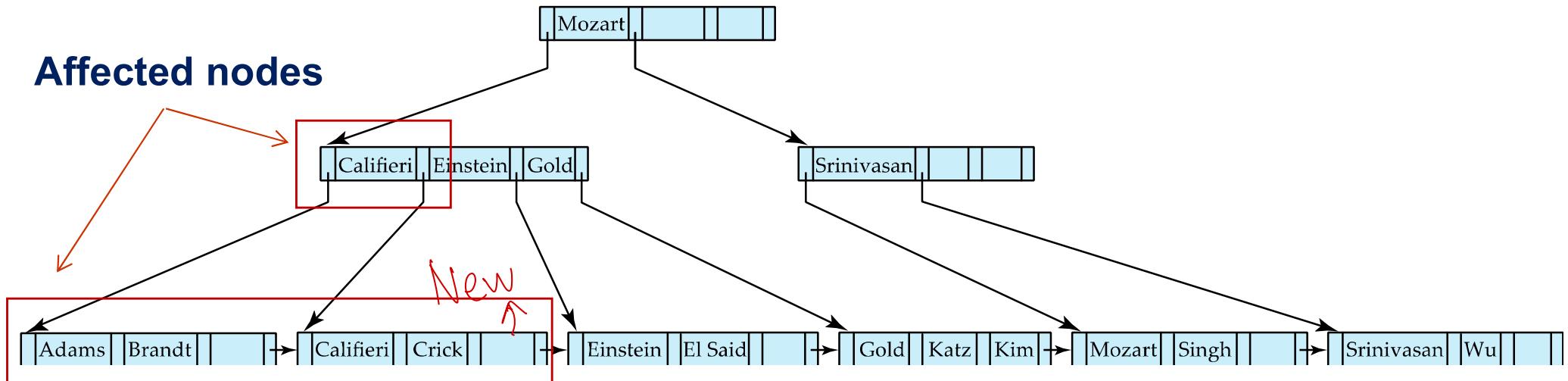


B⁺-Tree Insertion (Examples from Textbook)

■ Insert 'Adams'



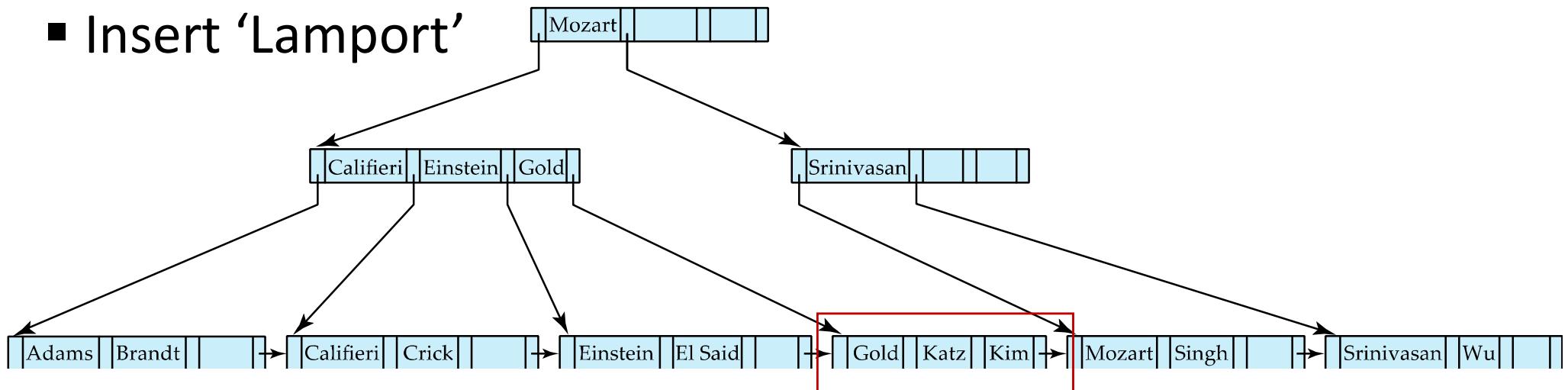
Affected nodes



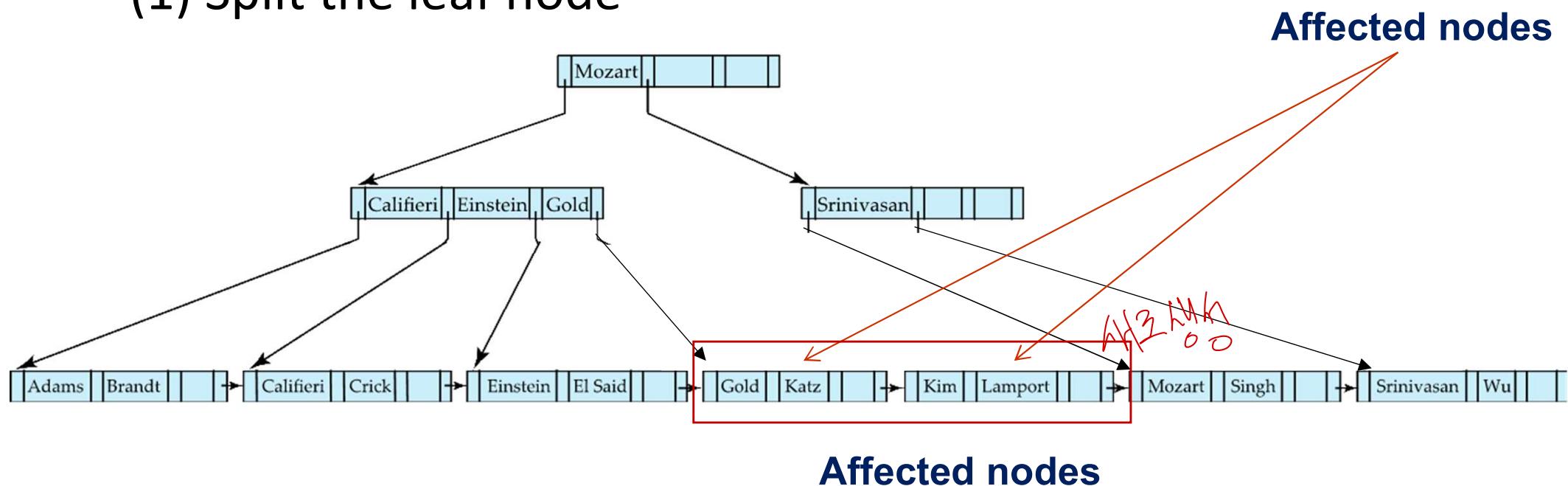
B⁺-Tree before and after insertion of “Adams”

B⁺-Tree Insertion

- Insert 'Lamport'

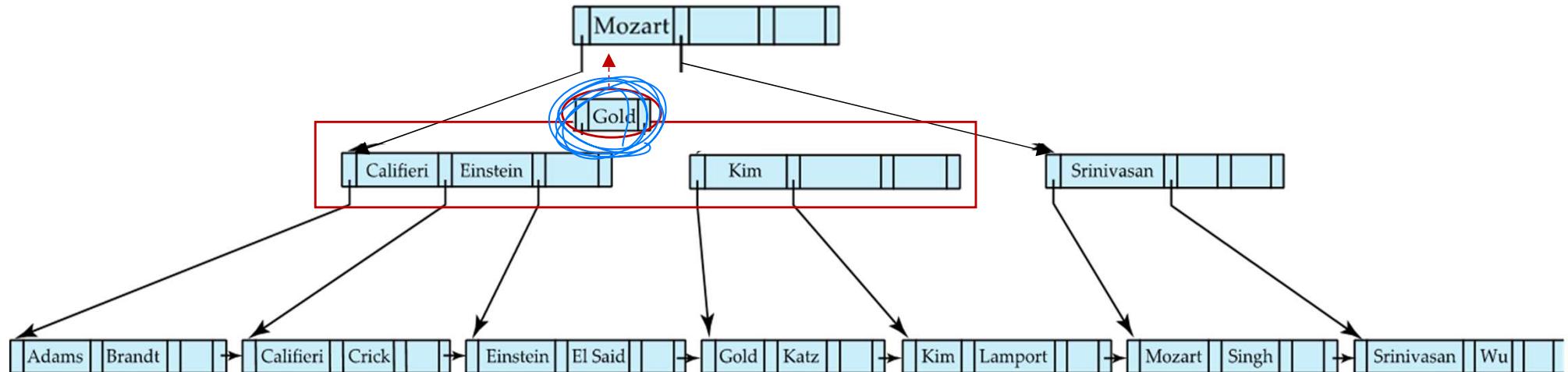


- (1) Split the leaf node

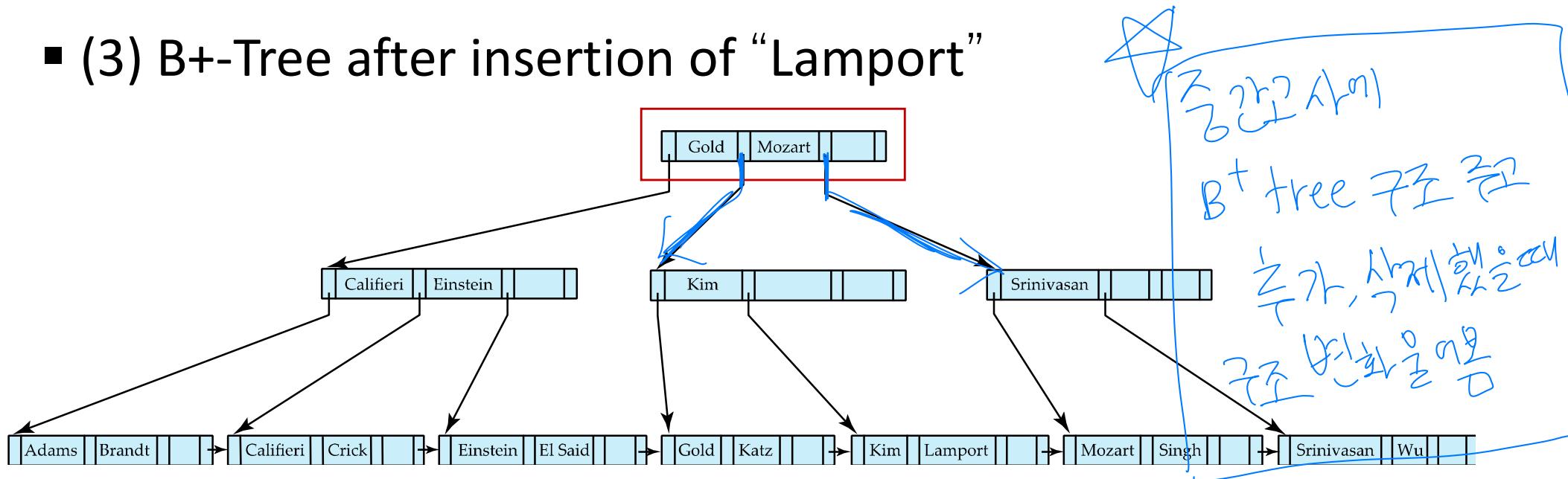


B⁺-Tree Insertion

- (2) Split non-leaf nodes



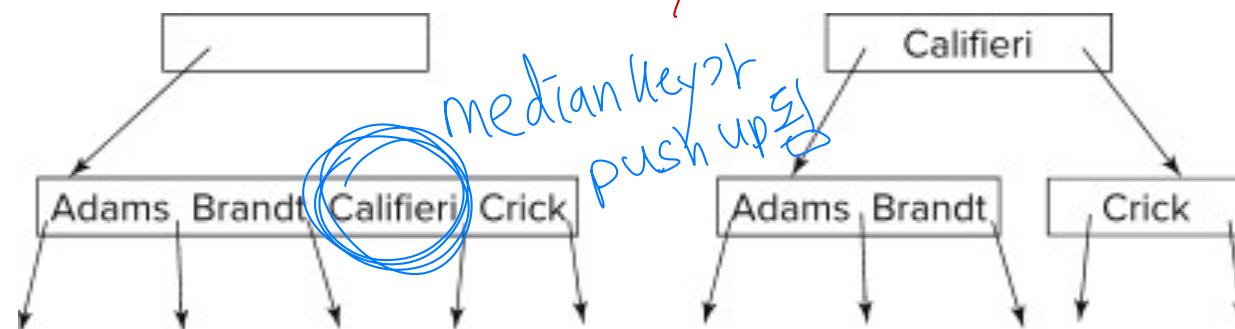
- (3) B⁺-Tree after insertion of “Lamport”



Insertion in B⁺-Trees (Cont.)

- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
 - Copy N to an in-memory temporary buffer space M
 - Insert (k,p) into M
 - Write $P_1, K_1, \dots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N on disks
 - Write $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \dots, K_n, P_{n+1}$ from M into newly allocated node N'
 - Insert $(K_{\lceil n/2 \rceil}, N')$ into parent N

- Example



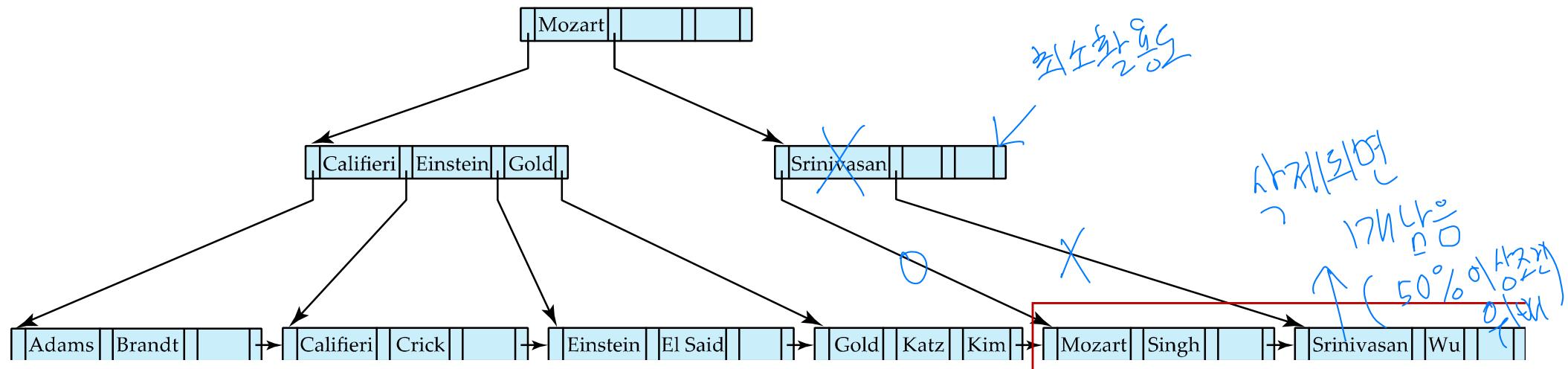
2(프) 노드 차지와 노드들 만 허당 (Internal Node?)

↗

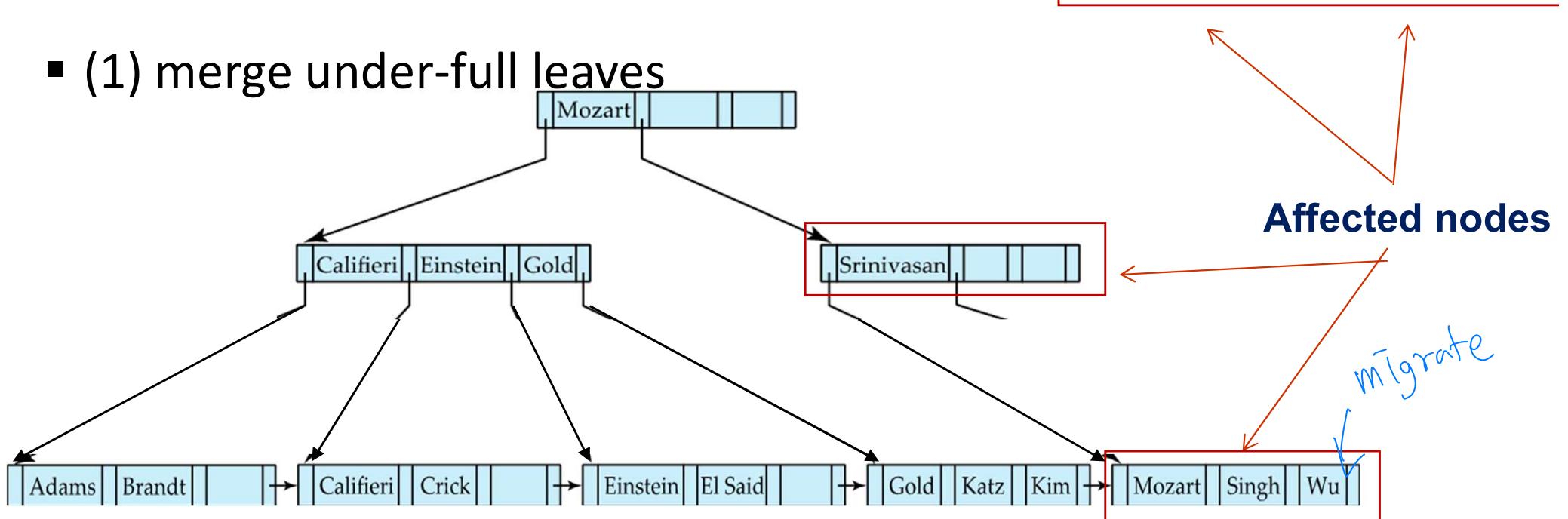
- Read pseudocode in textbook!

Examples of B⁺-Tree Deletion

- Deleting “Srinivasan” causes merging of under-full leaves



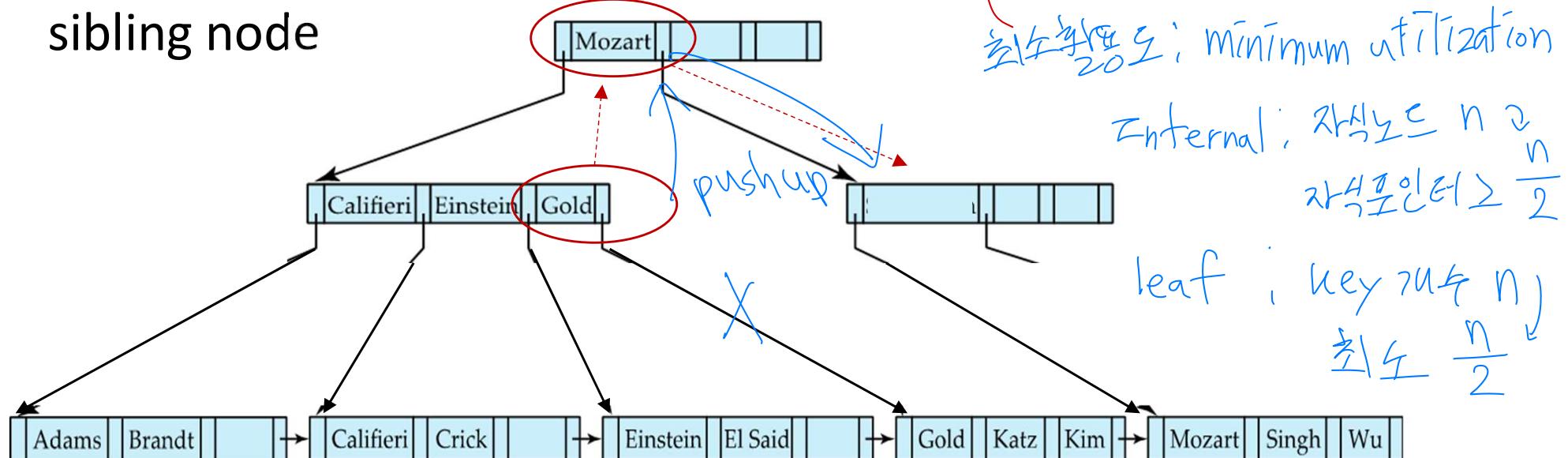
- (1) merge under-full leaves



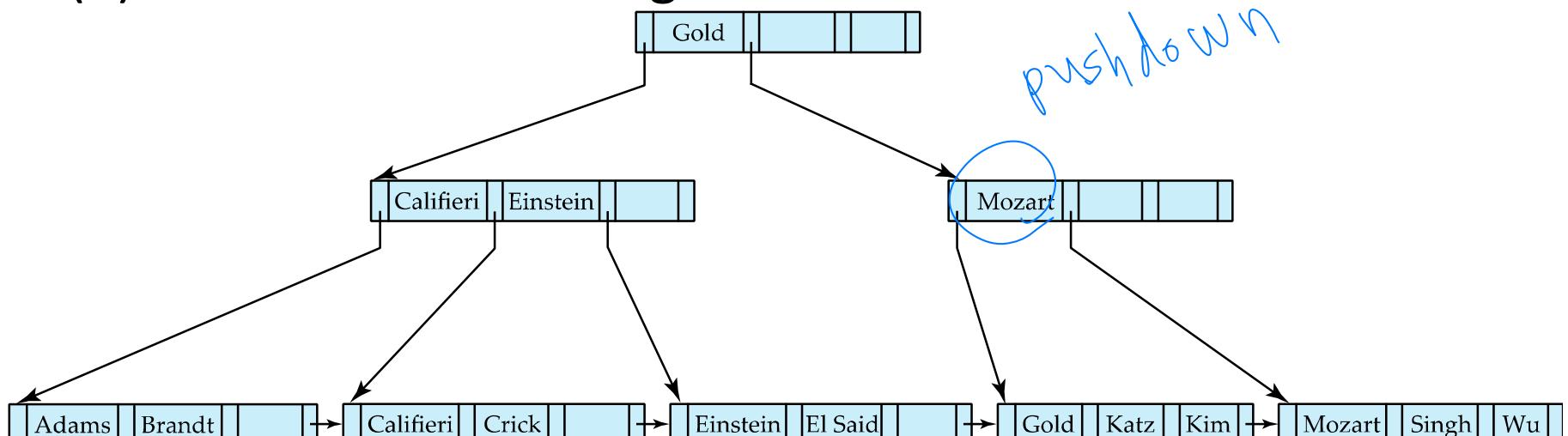
Examples of B⁺-Tree Deletion

부록 모든 노드 높이 같음

- (2) merge under-full non-leaf nodes, or borrow from non-leaf sibling node

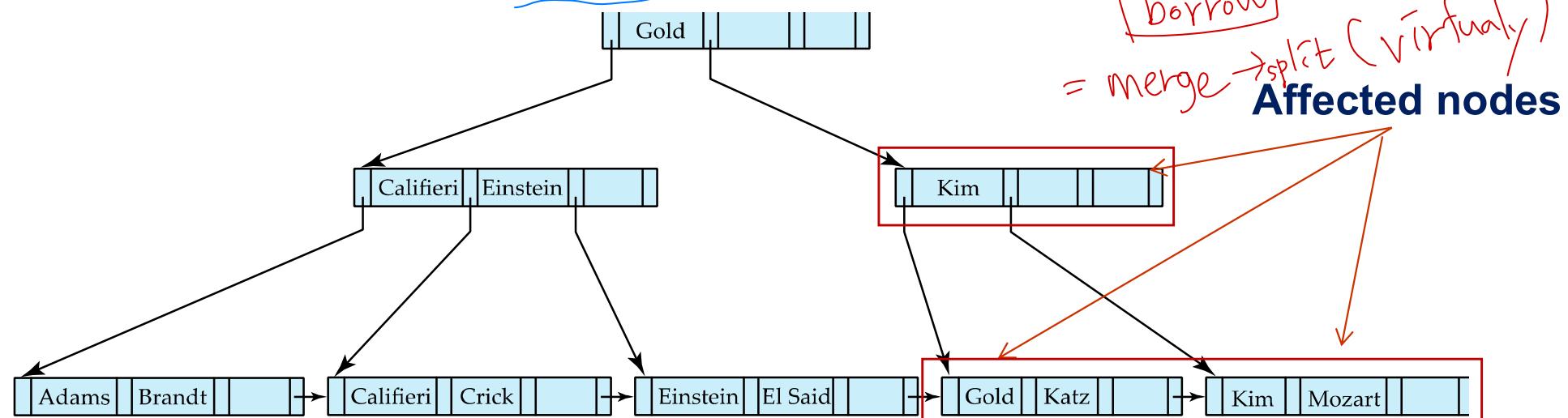
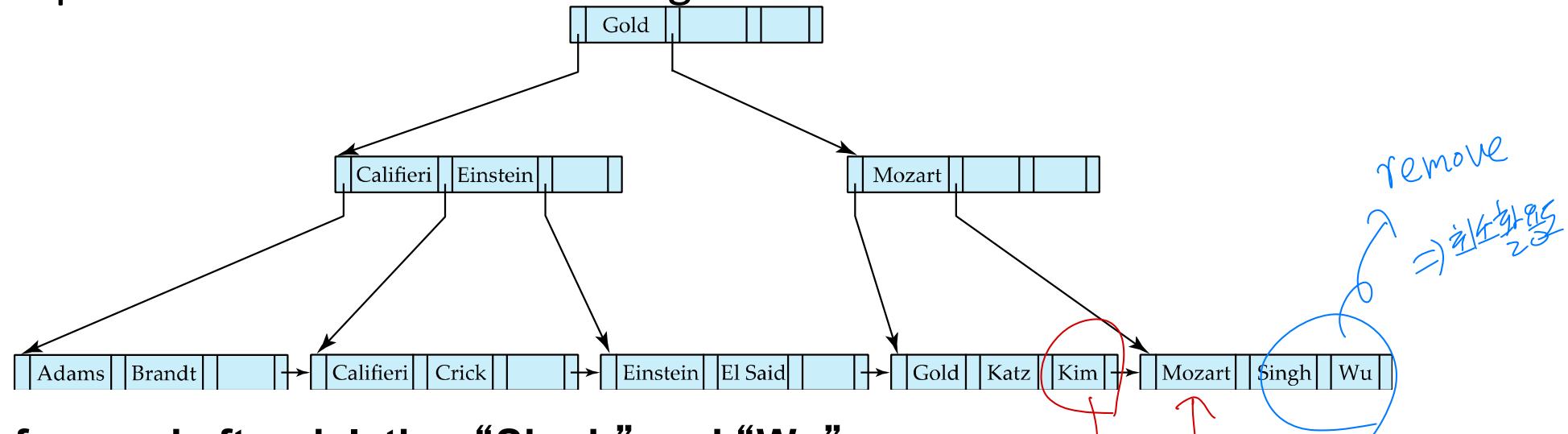


- (3) B+-tree after deleting “Srinivasan”



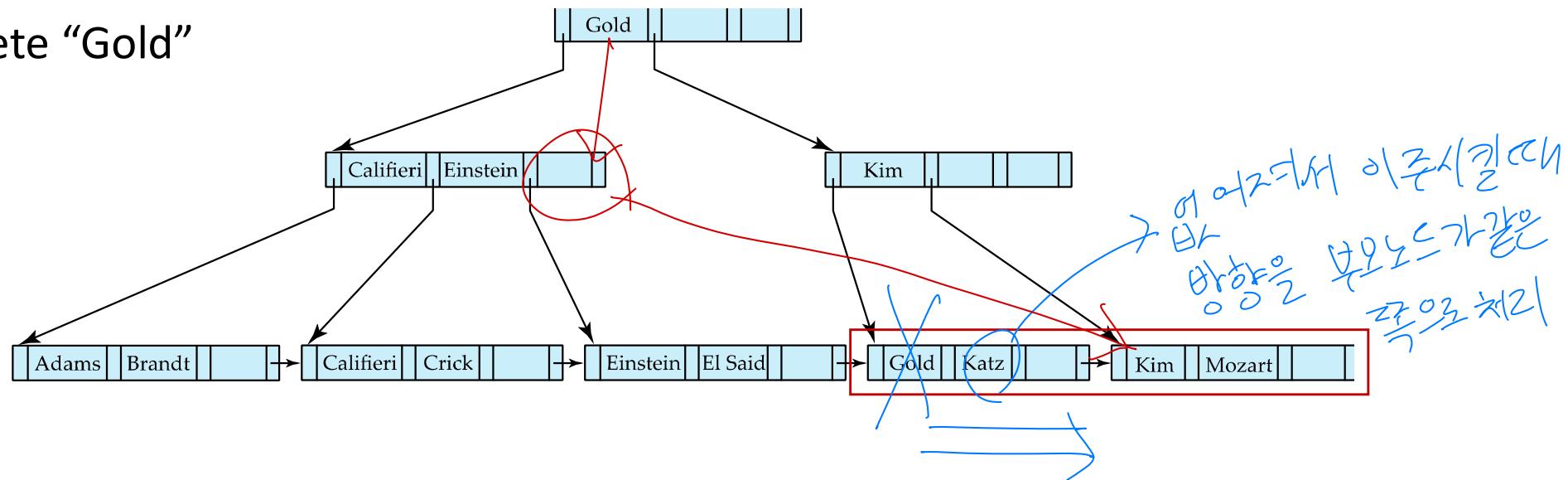
Redistribution between sibling nodes

- If a sibling node is not under-full, **borrow a (key,ptr)** from its sibling
- The parent node need to be changed as a result

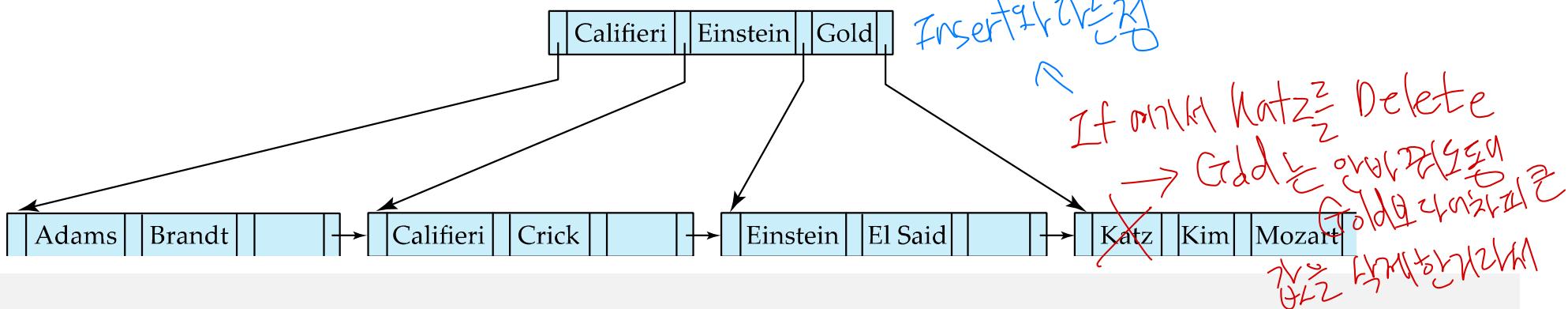


Example of B⁺-tree Deletion (Cont.)

- Delete “Gold”



- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged/redistributed with its sibling
 - Value separating two nodes is removed from the parent node
- If the root node has only one child, it is deleted



Updates on B⁺-Trees: Deletion

Assume record already deleted from file. Let K be the key of the record, and Ptr be the pointer to the record.

- Remove (K, Ptr) from the leaf node
 - If the node has too few entries and its sibling has enough free space, then **merge siblings**:
 - Insert all the key values in the two nodes into a single node (the one on the left), and delete the other node.
 - Delete the pair (K_{i-1}, P_i) , where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.
- record 삭제하면
Index도 삭제 해야 함
- 충분하지 않다면
merge 가능
⇒ redistribute

Updates on B⁺-Trees: Deletion

0/1/2/3/4/5/6/7

- If a node has too few entries due to the removal but cannot be merged into a sibling, then **redistribute pointers**:
 - Redistribute the entries from a sibling such that both have more than the minimum number of entries.
 - Update the corresponding key in the parent node.
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.
- *Note: Redistribution can be performed to postpone node splits as well.*

Pivotkey

연결하여
증가하는

재분배 → 각각 포인터를 증가시킨다

Complexity of Updates

29 ~ 30% 5/24/2023

- Cost (in terms of number of I/O operations) of insertion and deletion of a single entry proportional to height of the tree
 - With K entries and maximum fanout of n , worst case complexity of insert/delete of an entry is $O(\log_{\lceil n/2 \rceil}(K))$
- In practice, number of I/O operations is less:
 - Internal nodes tend to be in buffer
 - Splits/merges are rare, most insert/delete operations only affect a leaf node
- Average node occupancy depends on insertion order
 - 2/3rds with random, 1/2 with insertion in sorted order

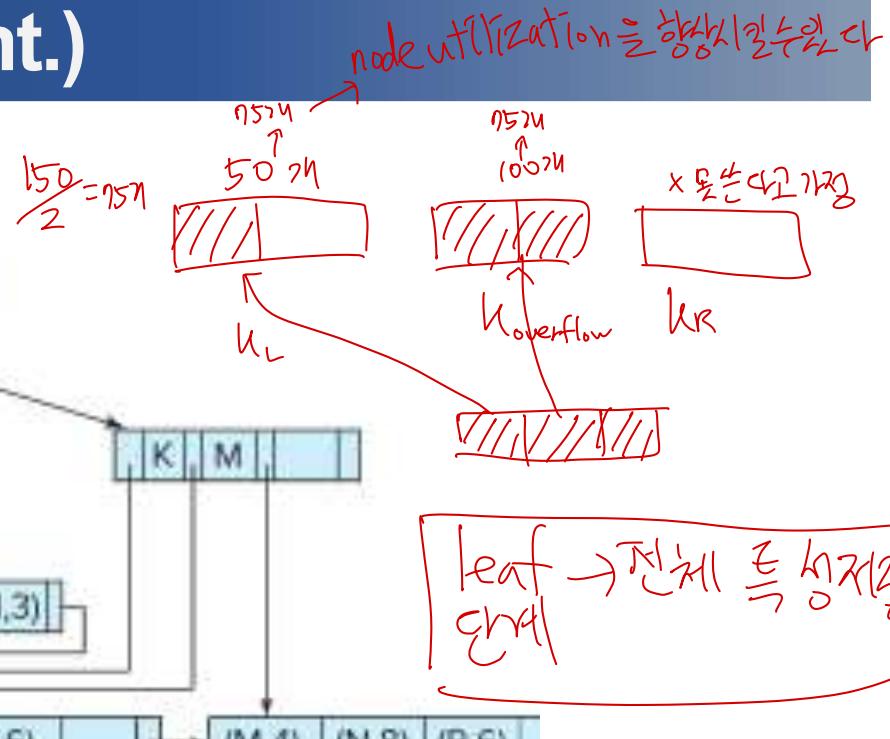
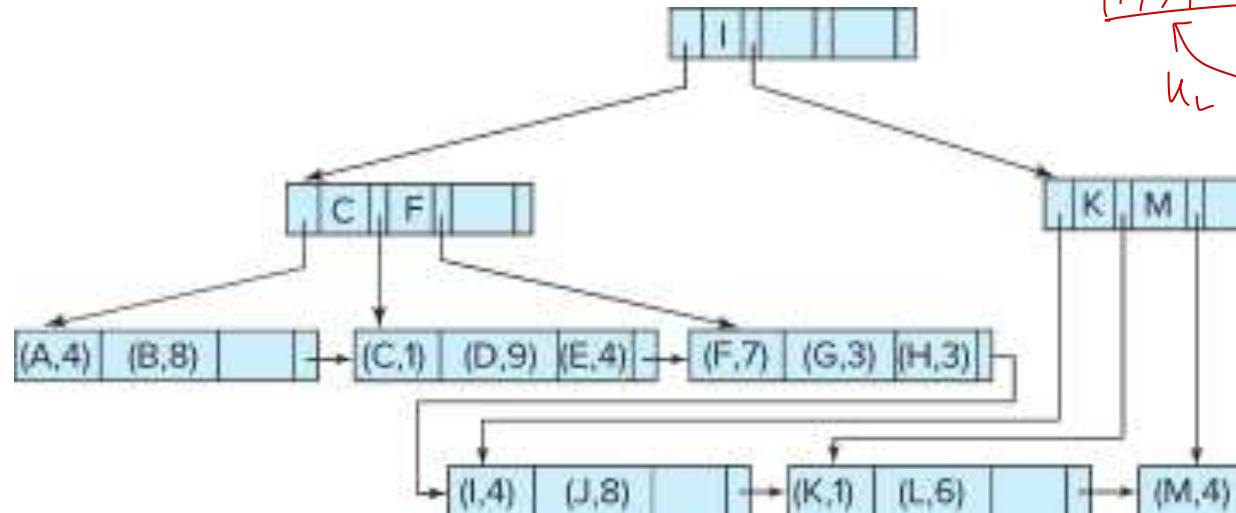
B⁺-Tree File Organization

- B⁺-Tree File Organization:
 - Leaf nodes in a B⁺-tree file organization store records, instead of pointers
 - Helps keep data records clustered even when there are insertions/deletions/updates
- Leaf nodes are still required to be half full
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B⁺-tree index.

original table $\xrightarrow{\text{B}^+ \text{-tree}} \text{행식의 표}$

B+-Tree File Organization (Cont.)

■ Example of B+-tree File Organization



- Good space utilization is important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
 - Involving both left and right siblings in redistribution
 - This results in each node having at least $\lfloor 2n/3 \rfloor$ entries

Handwritten notes at the bottom right:
만일 형제노드로부터 받으면 $\rightarrow 66\%$ 가 2가 될 수 있음
utilization을

Other Issues in Indexing

~~Record relocation and secondary indices~~

- If a record moves, all secondary indices that store record pointers have to be updated
- Node splits in B⁺-tree file organizations become very expensive
- *Solution:* use search key of B⁺-tree file organization instead of record pointer in secondary index
 - Add record-id if B⁺-tree file organization search key is non-unique
 - Extra traversal of file organization to locate record
 - Higher cost for queries, but node splits are cheap

Not
Important

Indexing Strings

■ Variable length strings as keys

- Variable fanout
- Use space utilization as criterion for splitting, not number of pointers

연결노드 가장 작은값 = pivotkey로 선택되는

자장 \rightarrow string일 경우 key길이가 제한된다면

실제 저장하면 X

"Mozart" \rightarrow "M" or "Mo"로 압축

압축 시 가능해!

■ Prefix compression

- Key values at internal nodes can be prefixes of full key
 - Keep enough characters to distinguish entries in the subtrees separated by the key value
 - E.g., "Silas" and "Silberschatz" can be separated by "Silb"
- Keys in leaf node can be compressed by sharing common prefixes

Bulk Loading and Bottom-Up Build

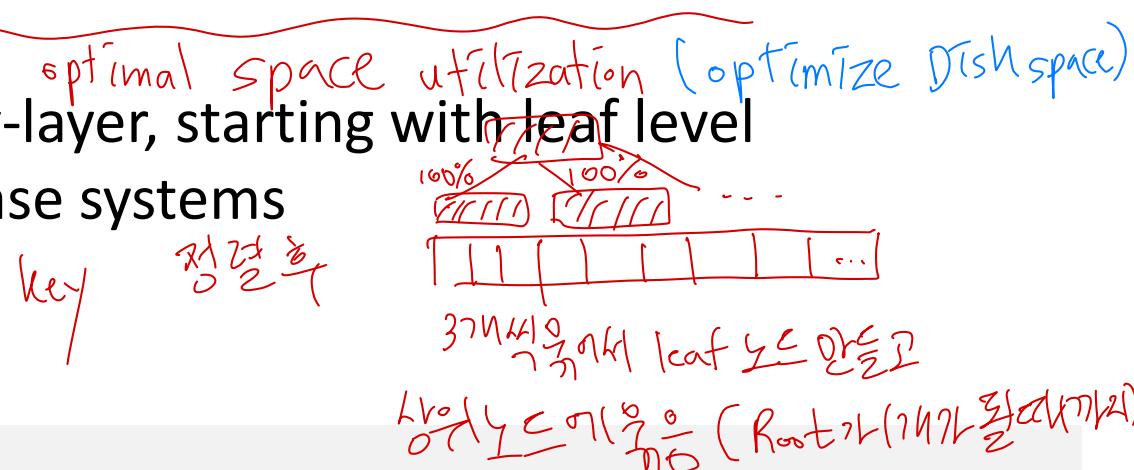
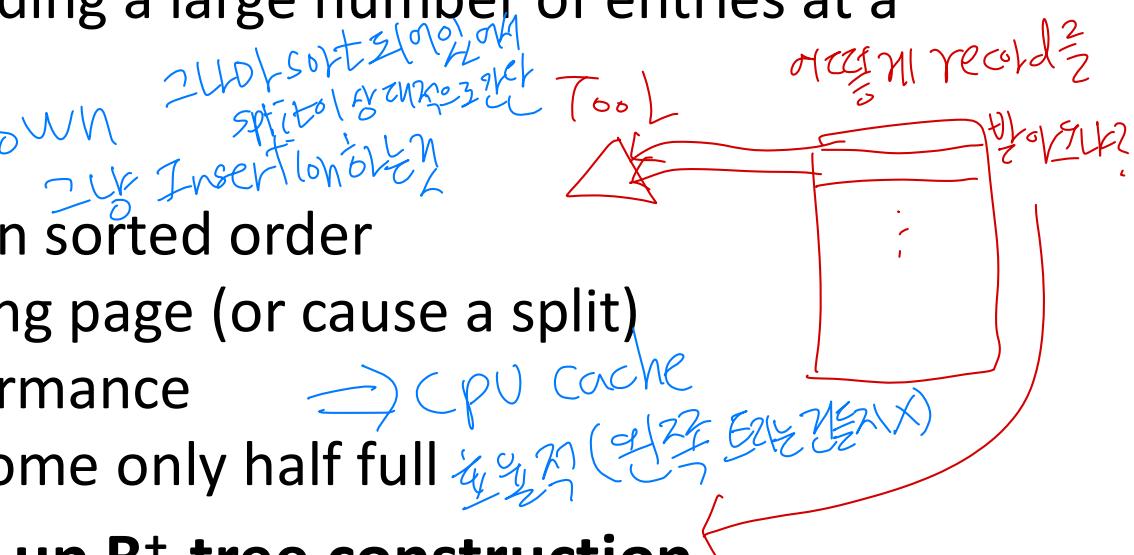
- Inserting entries one-at-a-time into a B⁺-tree requires ≥ 1 IO per entry
 - assuming leaf level does not fit in memory
 - can be very inefficient for loading a large number of entries at a time (**bulk loading**)

- Efficient alternative 1: *Top-down*

- sort entries and insert them in sorted order
 - insertion will go to existing page (or cause a split)
 - much improved IO performance
 - but most leaf nodes become only 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 by most database systems



B-Tree Index Files

- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node

P_1	K_1	P_2	\dots	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	---------	-----------	-----------	-------

(a)

P_1	B_1	K_1	P_2	B_2	K_2	\dots	P_{m-1}	B_{m-1}	K_{m-1}	P_m
-------	-------	-------	-------	-------	-------	---------	-----------	-----------	-----------	-------

(b)

- Nonleaf node – pointers B_i are the bucket or file record pointers.

B-Tree Index File Example

- B-tree (above) and B+-tree (below) on same data**

