Chapter 11: Indexing and Hashing

Database System Concepts, 6th Ed.

Chapter 11: Indexing and Hashing

- 11.1 Basic Concepts
- 11.2 Ordered Indices
- 11.3 B^+-Tree Index Files
- 11.4 B^+-Tree Extensions
- 11.5 Multiple-Key Access
- 11.6 Static Hashing
- 11.7 Dynamic Hashing
- 11.8 Comparison of Ordered Indexing and Hashing
- 11.9 Bitmap Indices
- 11.10 Index Definition in SQL

Tree 기반

Hashing 기반

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., Author catalog in library, Term index at the end of a book, Google search

- An **index file** consists of index records (also called **index entries**) of the form

search-key	pointer

  - **Search Key** - set of attributes used to look up records in a file
  - **Pointer –** pointer to a data record

- Index files are typically much smaller than the original data file

- Two main streams of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”

Index Evaluation Metrics

- Access types supported efficiently
  - records with a specified value in the attribute (point query)
  - records with an value in a specified range of values. (range query)

- Index Evaluation Metric
  - Time overhead
    - Access time
    - Insertion time
    - Deletion time
  - Space overhead

Chapter 11: Indexing and Hashing

- 11.1 Basic Concepts
- 11.2 Ordered Indices
- 11.3 B$^+$-Tree Index Files
- 11.4 B$^+$-Tree Extensions
- 11.5 Multiple-Key Access
- 11.6 Static Hashing
- 11.7 Dynamic Hashing
- 11.8 Comparison of Ordered Indexing and Hashing
- 11.9 Bitmap Indices
- 11.10 Index Definition in SQL

Tree 기반

Hashing 기반

Figure 11.01: Sequential file for instructor records

Instructor_ID,	name,	department,	salary,	next-record
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Update들이 많이 발생하면 record들의 배열이 뒤죽박죽되거나
아니면 Instructor_ID순서로 file을 유지하려면 할일이 많음!!!

Query Processing 할때에 sorted 되어있는 Instructor_ID에 binary search하거나
다른 attribute 들에 대해서 linear search를 하는 방법밖에 없는데…
그것도 disk 에 있는 record 들을 가져와서 처리해야 하는 부담이 있다….

Ordered Indices

- In an **ordered index,** index entries are stored sorted on the search key value
  E.g., author catalog in library.

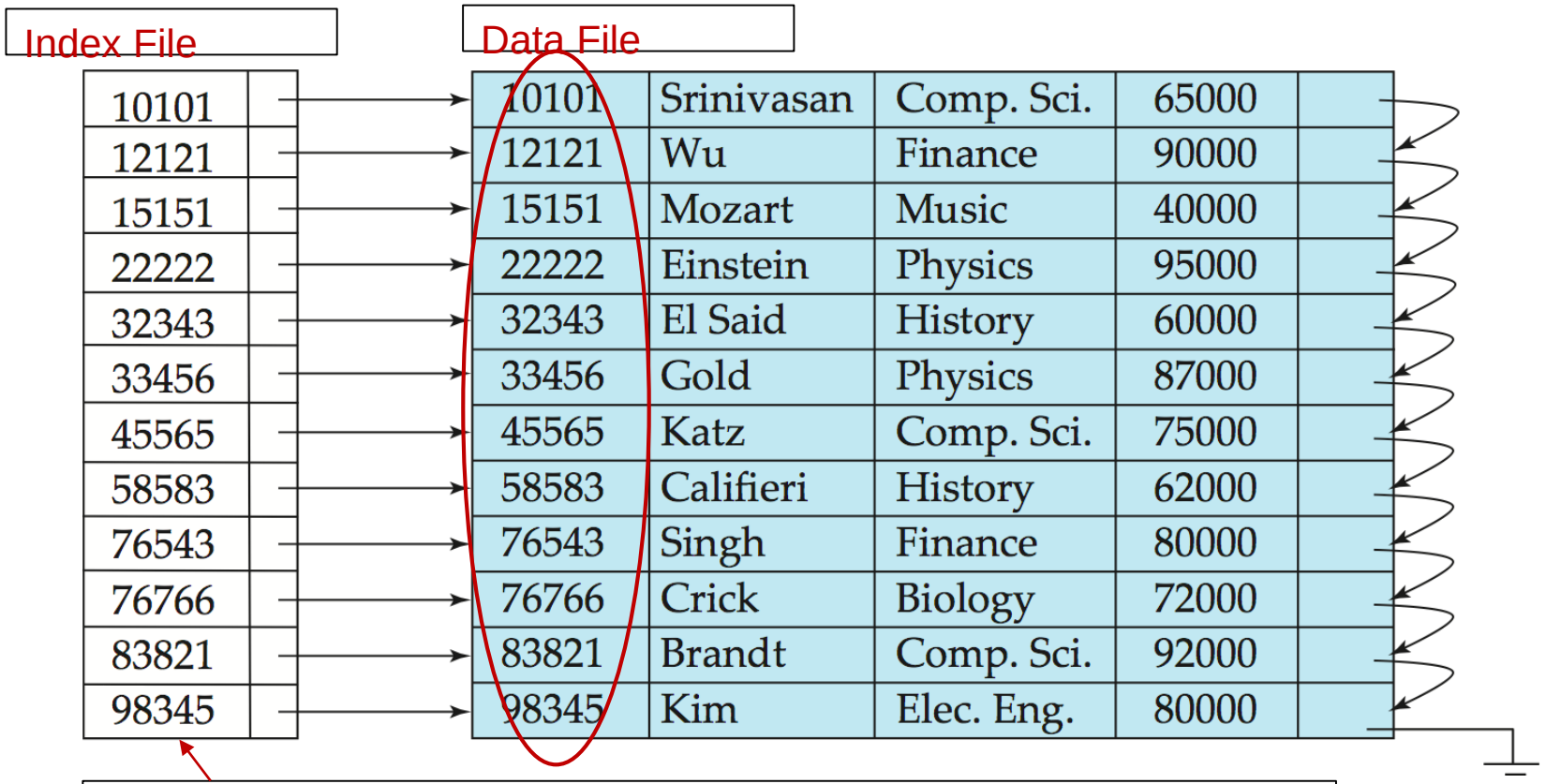
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index**.**

- Index-sequential file**:** ordered sequential file with a primary index.

- Dense Index vs Sparse Index

Dense Index Files [1/2]

- **Dense index** — Index record appears for every search-key value in the file.
  - E.g. A dense index on *ID* attribute of *instructor* relation

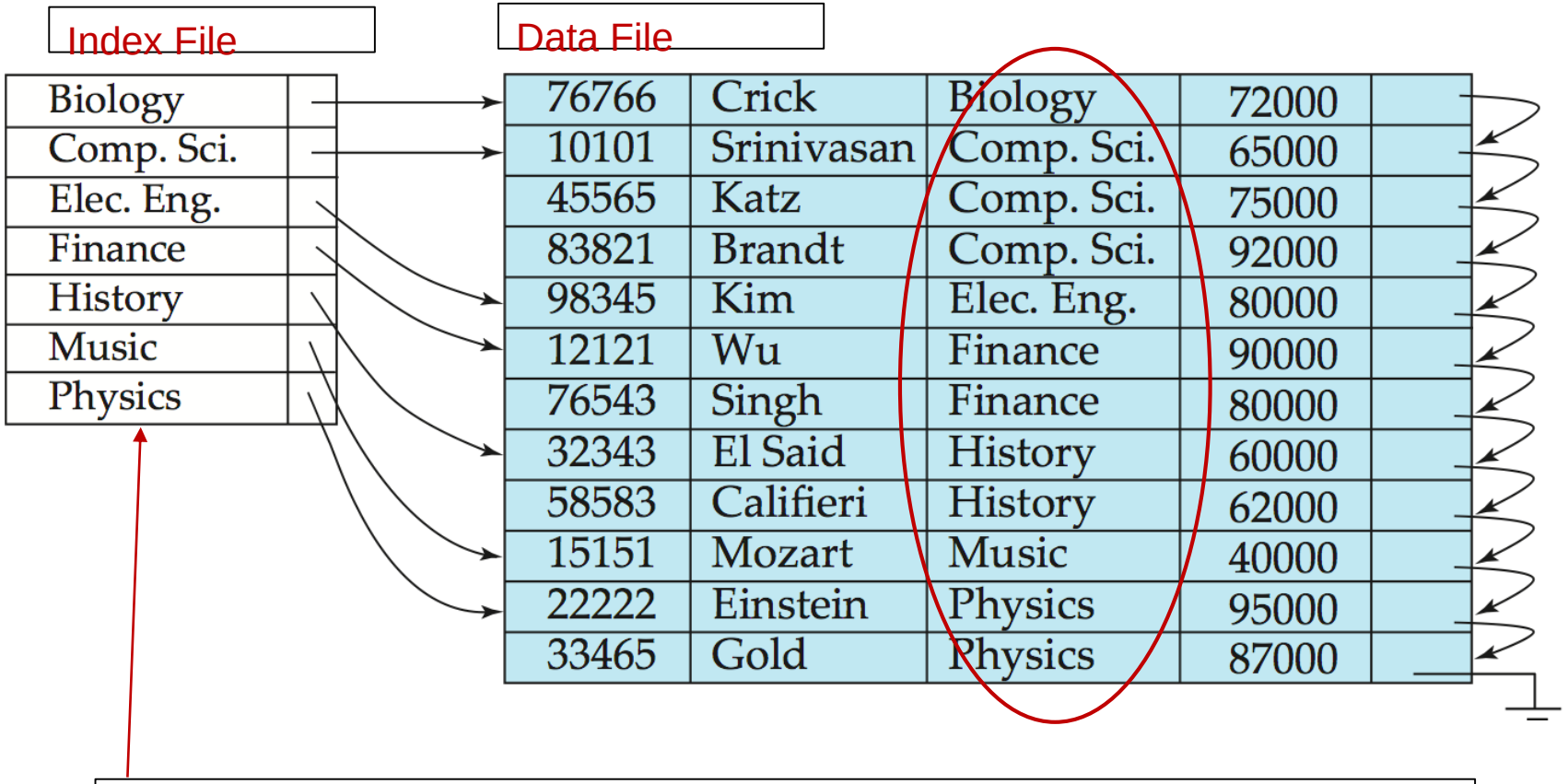


Index file is made by every search key values of ID of the data file !

Query Processing 할때에 Index File 이 memory에 있으면 disk에 있는 record들을 Memory로 가져오는 부담을 크게 줄일수 있다…

Dense Index Files [2/2]

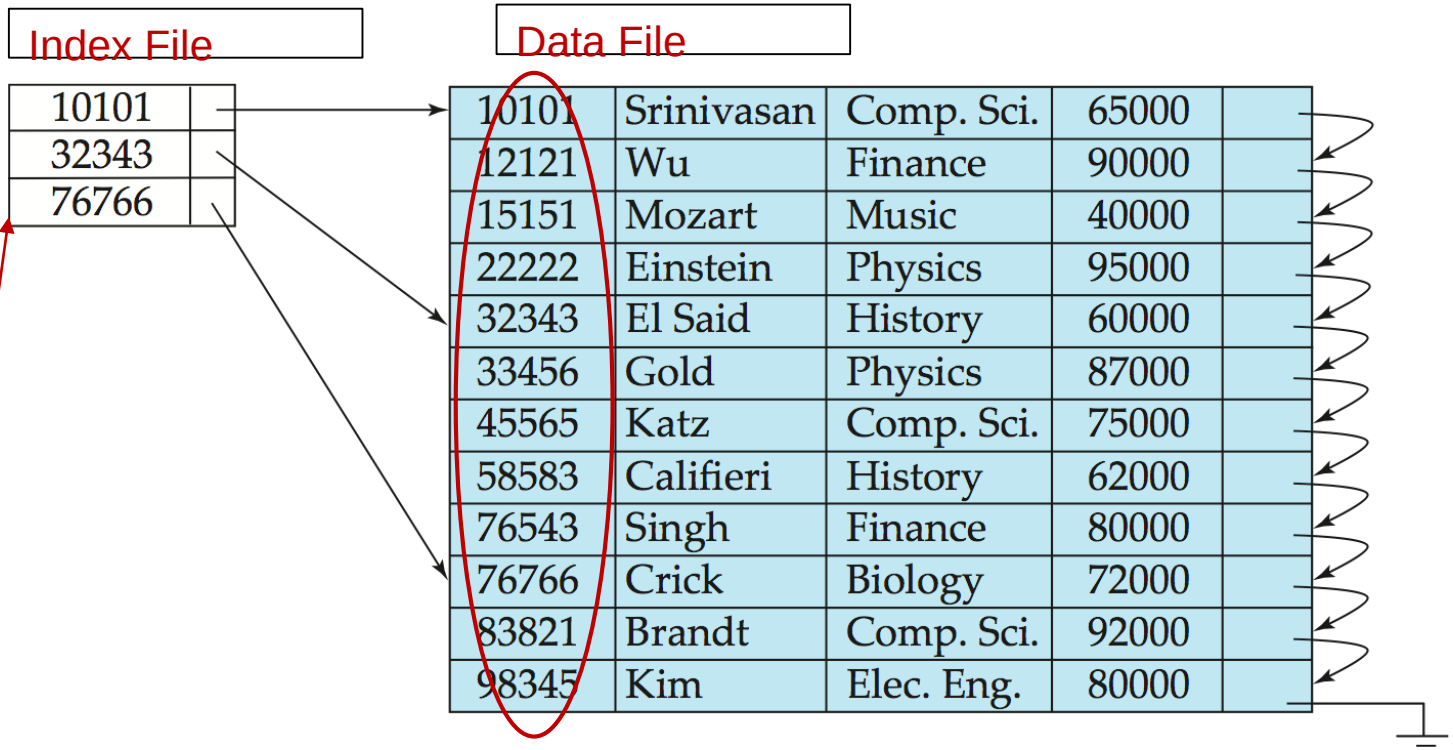
- A dense index on *dept_name*, with *instructor* file sorted on *dept_name*



Index file is made by every search key values of Department of the data file !

Sparse Index Files [1/2]

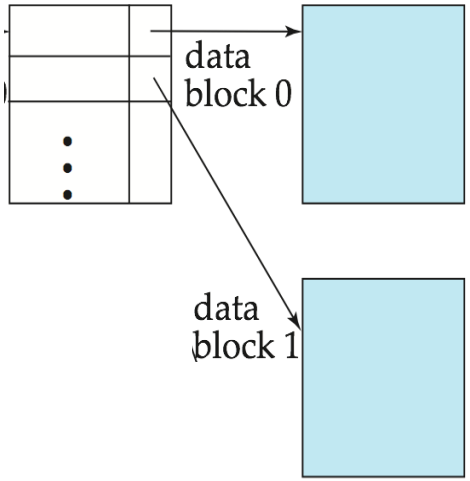
- **Sparse Index**: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K*
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to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



Index file is made by only some search key values of Department of the data file !

Sparse Index Files [2/2]

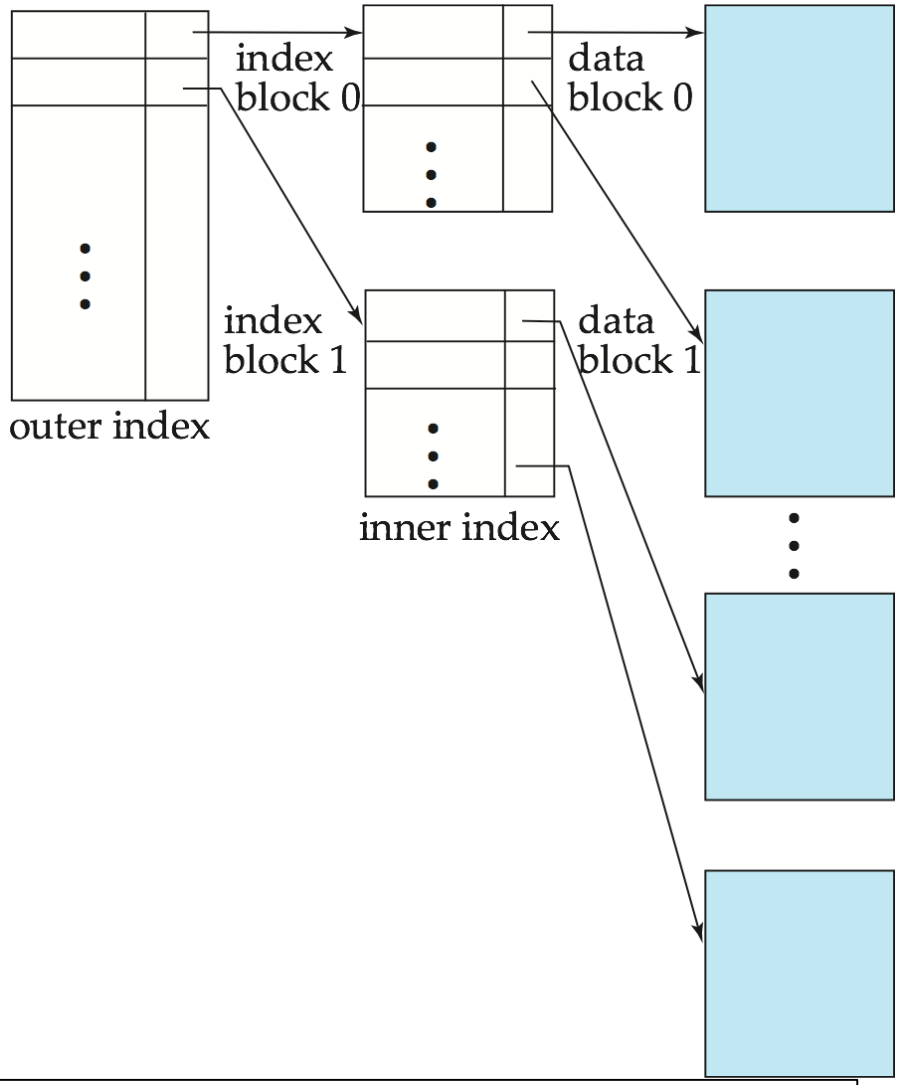
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions
  - Generally slower than dense index for locating records
- **Good tradeoff**: sparse index with an index entry for every block in file, corresponding to least search-key value in the block



- Index file은 query처리에 유익한 방법이지만… maintenance cost도 생각해야!!!

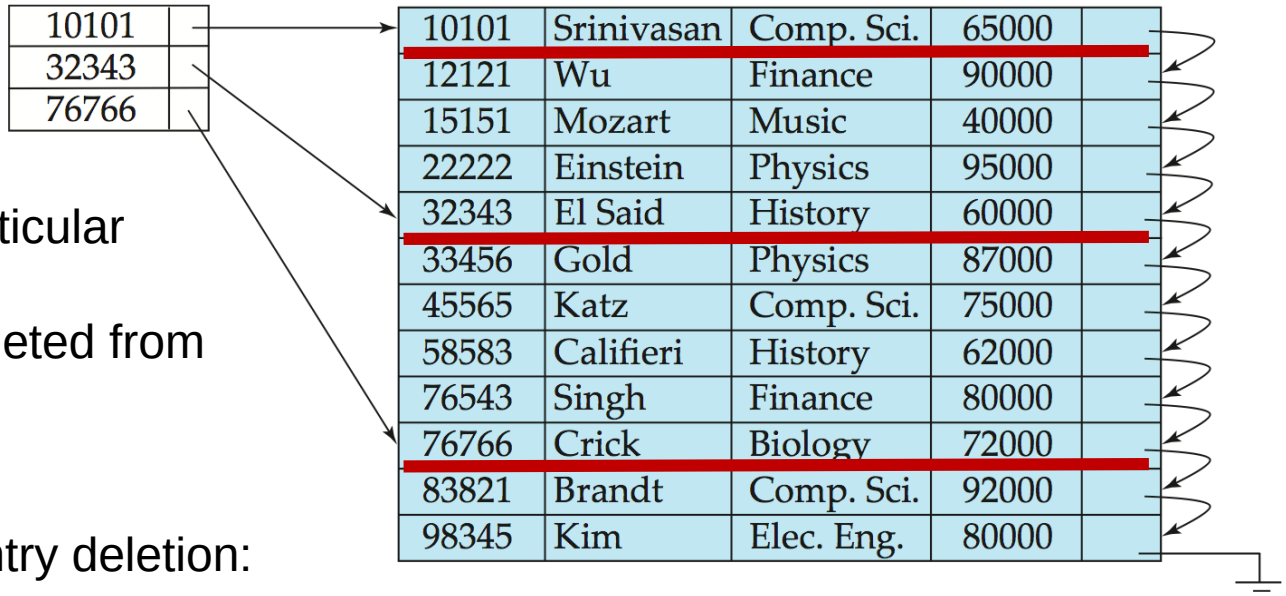
Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even 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.



책뒤에 있는 term index를 보면 “A”, “B”, “C”…. 순서로 중간중간에 표시가 있어서 term search가 빨리 진행될수 있는것과 유사!

Index Update: Deletion



- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

- Single-level index entry deletion:
  - **Dense indices** – deletion of search-key is similar to file record deletion.
  - **Sparse indices** –
    - If an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order). Ex: delete the 76766 record in data file ➔ index file에서 76766를 83821로 교체
    - If the next search-key value already has an index entry, the entry is deleted instead of being replaced. Ex: 위의 경우에 83821 record를 가르키는 index entry가 있다면 76766을 83821로 교체가 아니라 delete

Index Update: Insertion

- **Single-level index insertion:**
  - Perform a lookup using the search-key value appearing in the record to be inserted.
  - **Dense indices** – if the search-key value does not appear in the index, insert it.
  - **Sparse indices** – Assume the index stores an entry for each block of the file.
    - No change needs to be made to the index unless a new block is created.
    - If a new block is created, the first search-key value appearing in the new block is inserted into the index.

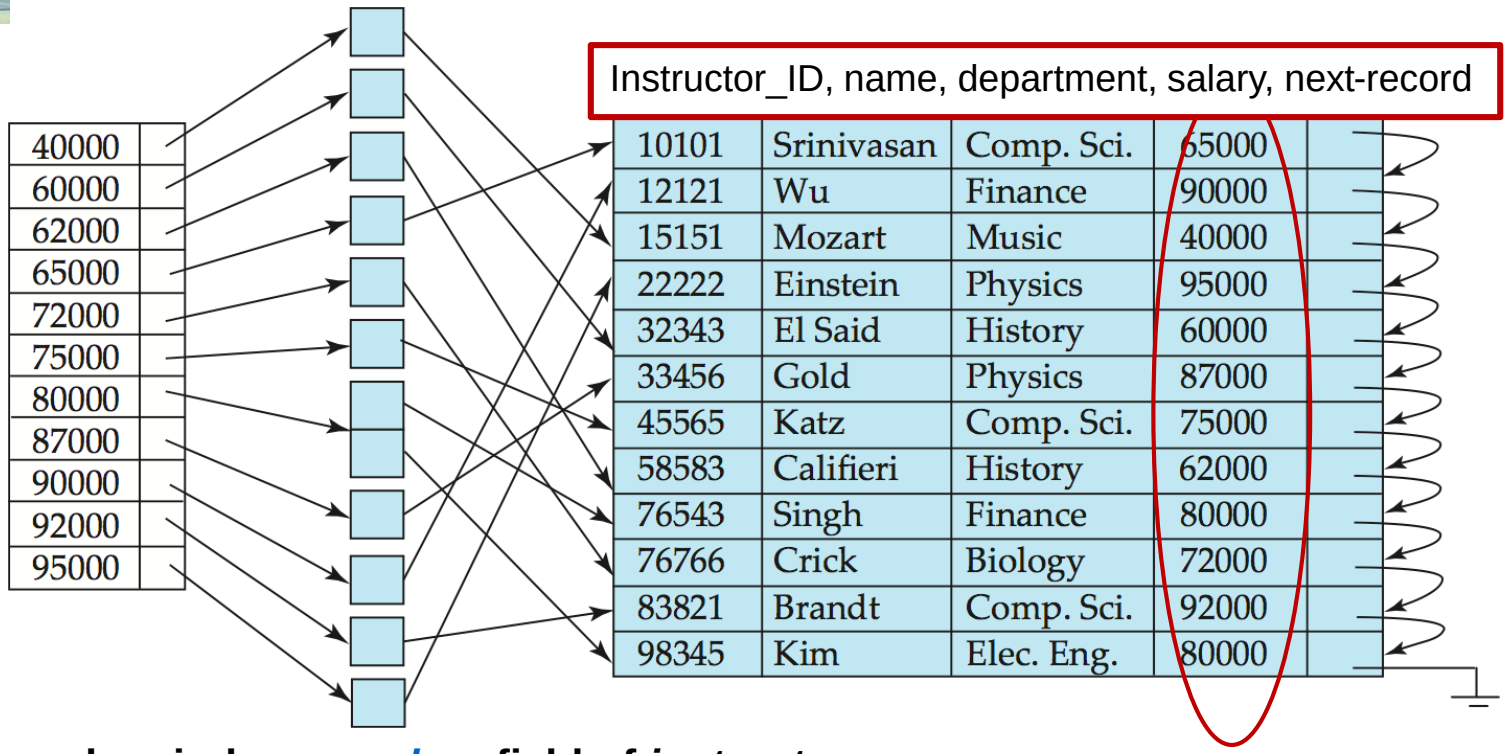
  (new block을 가리키는 index record가 만들어지고 index file에 반영됨)

- **Multilevel insertion and deletion:** algorithms are simple extensions of the single-level algorithms

Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
  - Example 1: In the *instructor* relation stored sequentially by ID, we may want to find all instructors in a particular department
  - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values
- We can have a secondary index with an index record for each search-key value
- Example1 query would need only one primary index on department attribute
- Example2 query would need a secondary index on salary as well as primary index
- Index file은 필요한 만큼 여러 개의 attribute에 만들수 있고, 그중에 대표를 primary index라고 하고, 나머지 index들은 secondary index라고 부른다!

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
- Salary에 대한 Secondary index를 만들면 query가 efficient하게 처리 ➔ maintenance cost가 추가!

Primary and Secondary Indices

- Indices offer substantial benefits when searching for records
- When a file is modified, every index on the file must be updated
  - Updating indices imposes overhead on database modification
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive (may cause many disk block IOs)
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access

- So far, we have talked about "Indexed Sequential Files"
- Sometimes called, ISAM "Indexed Sequential Access Method"

ISAM도 기본적으로 Sequential File이므로, Update (Insert, Delete)들이 많이 발생하면
Record들의 배열이 뒤죽박죽되므로 성능이 저하되고
성능이 저하되는것을 막으려면 Data Reorganization도 필요 !!!
즉 Dynamic Data Environment에는 바람직하지 않다!!

Chapter 11: Indexing and Hashing

- 11.1 Basic Concepts
- 11.2 Ordered Indices
- 11.3 B+-Tree Index Files
- 11.4 B+-Tree Extensions
- 11.5 Multiple-Key Access
- 11.6 Static Hashing
- 11.7 Dynamic Hashing
- 11.8 Comparison of Ordered Indexing and Hashing
- 11.9 Bitmap Indices
- 11.10 Index Definition in SQL

Tree 기반

Hashing 기반

참고자료

Introduction: The Invention of the B-tree

- 1972 Acta Infomatica : R. Bayer and E. McCreight (at Boeing Corporation), "Organization and Maintenance of Large Ordered Indexes"

- 1979 : D. Comer, "The Ubiquitous B-tree", ACM Computing Survey
  - 'de facto' standard for database index

- Why the name B-tree?
  - Balanced, Bushy, Broad, Boeing, Bayer

- Retrieval time, Insertion time, Deletion time

  = $\log_K I$ (I : no of indexes in file, K : no of indexes in a page)

- Excellent for dynamically changing random access files

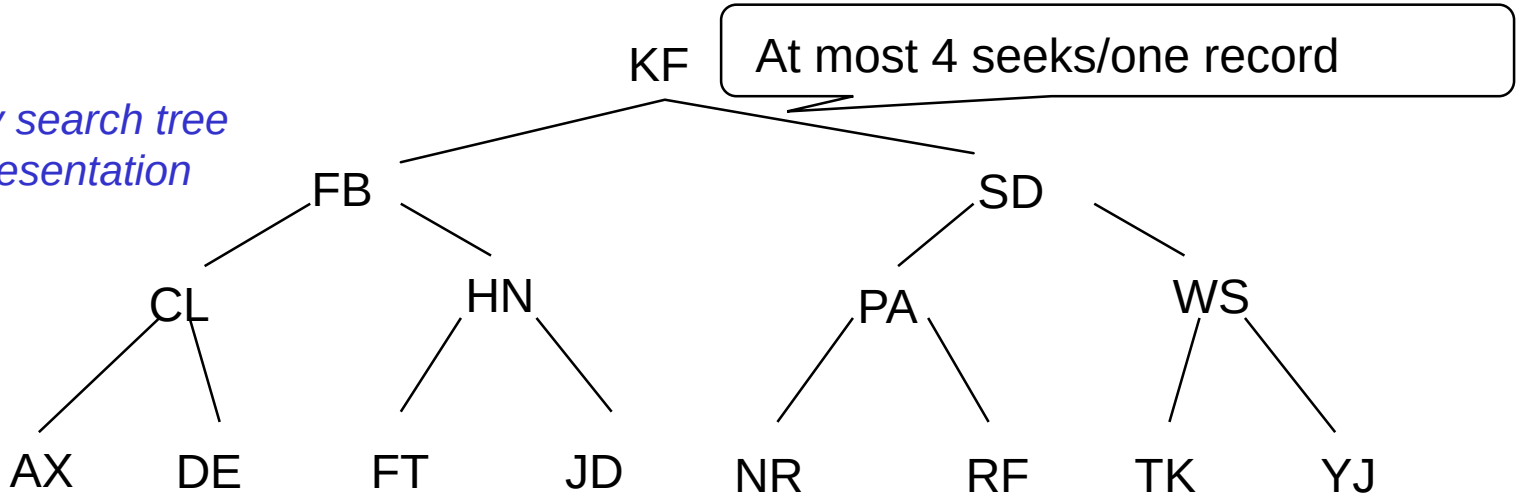
Binary Search Tree

참고자료

- Advantages
  - Data may not be physically sorted
  - Good performance on balanced tree
  - Insert cost = search cost
- Disadvantages
  - In out-of-balance binary tree, more seeks are required

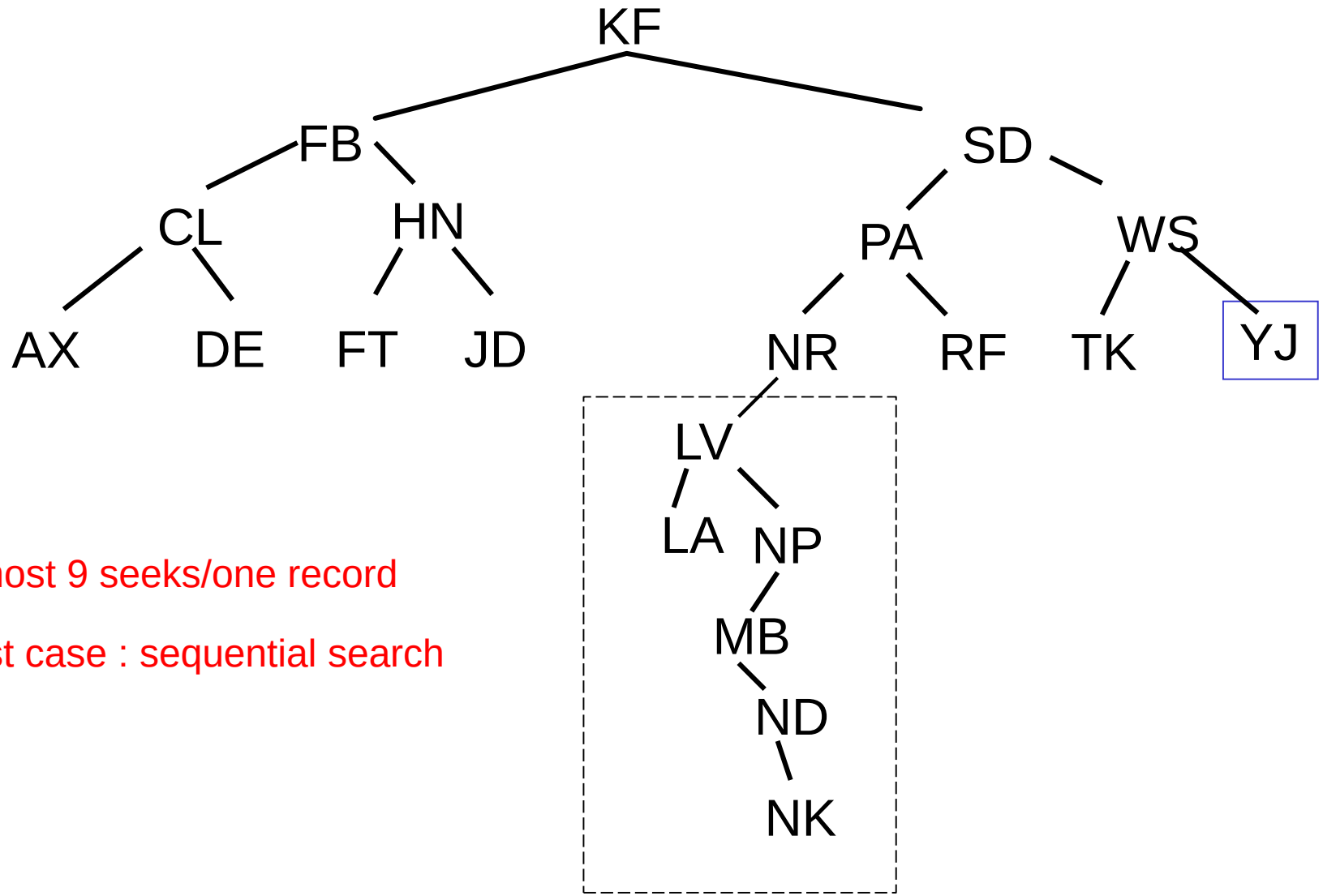
- Sorted list of keys : **AX, CL, DE, FB, FT, HN, JD, KF, NR, PA, RF, SD, TK, YJ**

Binary search tree representation



참고자료

Unbalanced Binary Tree



- At most 9 seeks/one record

- Worst case : sequential search

AVL Tree Concept [1/3]

참고자료

- AVL Tree : Balanced Binary Search Tree with respect to the height of subtree
  - G.M. **A**del'son **V**el'ski and E.M. **L**andis *(1962). ""An algorithm for the organization of information"". 《Proceedings of the USSR Academy of Sciences》* (Russian)
  - Maintenance overhead for balancing is needed
  - Completely balanced AVL tree : worst-case search ➔ $\log_2(N+1)$

- Definition
  - An empty tree is AVL Tree
  - If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then

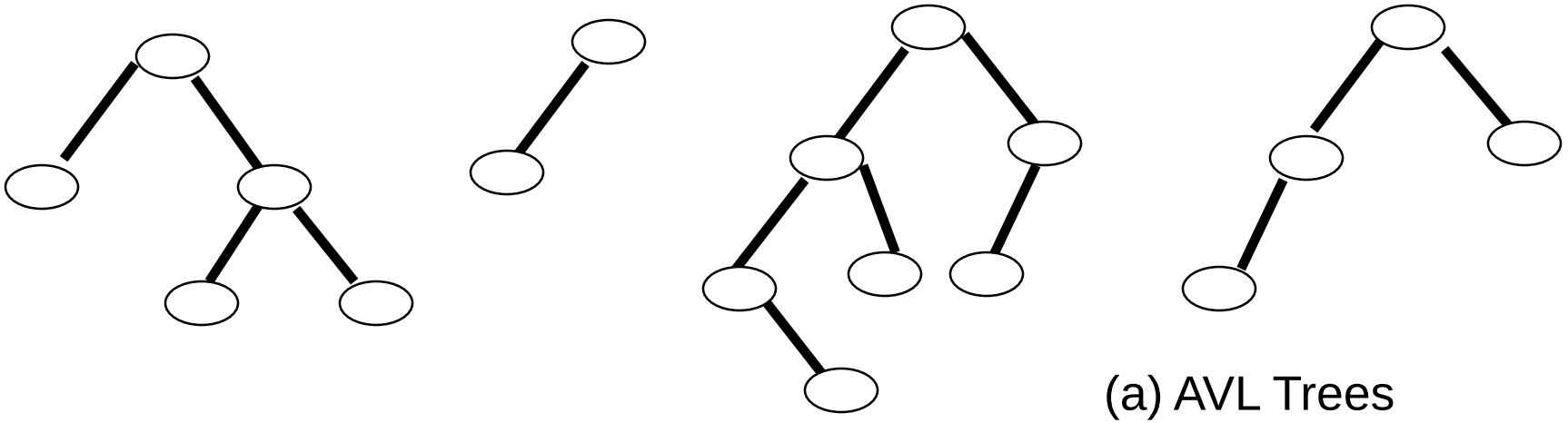
    T is AVL Tree

    IFF

    (1) T_L and T_R are height balanced and (2) $|h_L - h_R| < 1$ where h_L and h_R are the heights of T_L and T_R, respectively

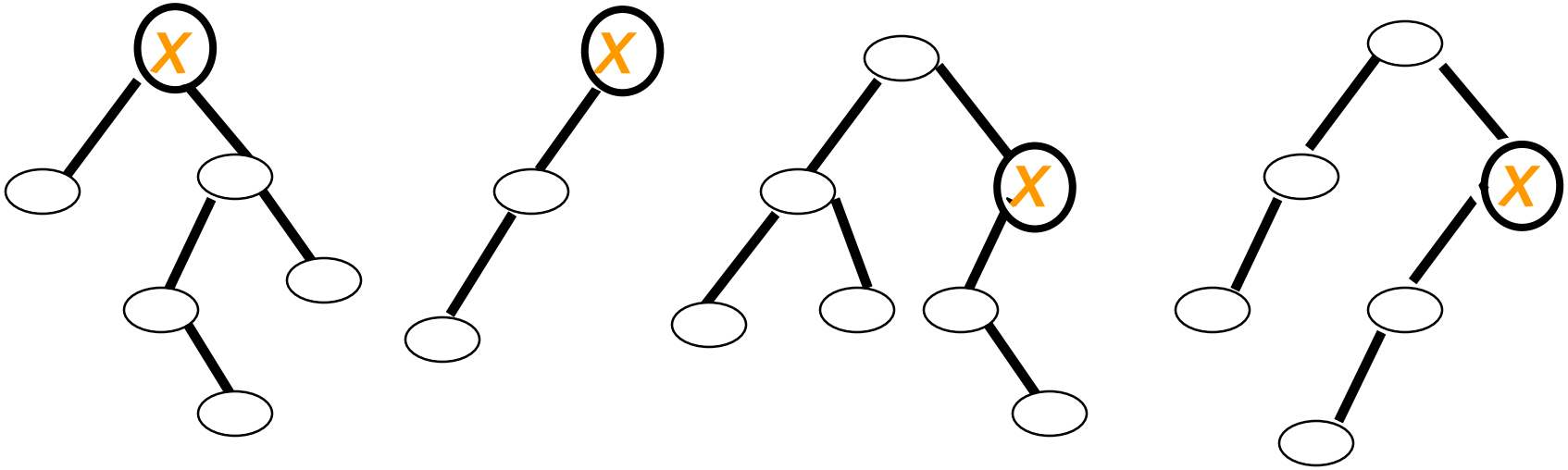
AVL Tree Concept [2/3]

참고자료



(a) AVL Trees

(b) Non - AVL Trees



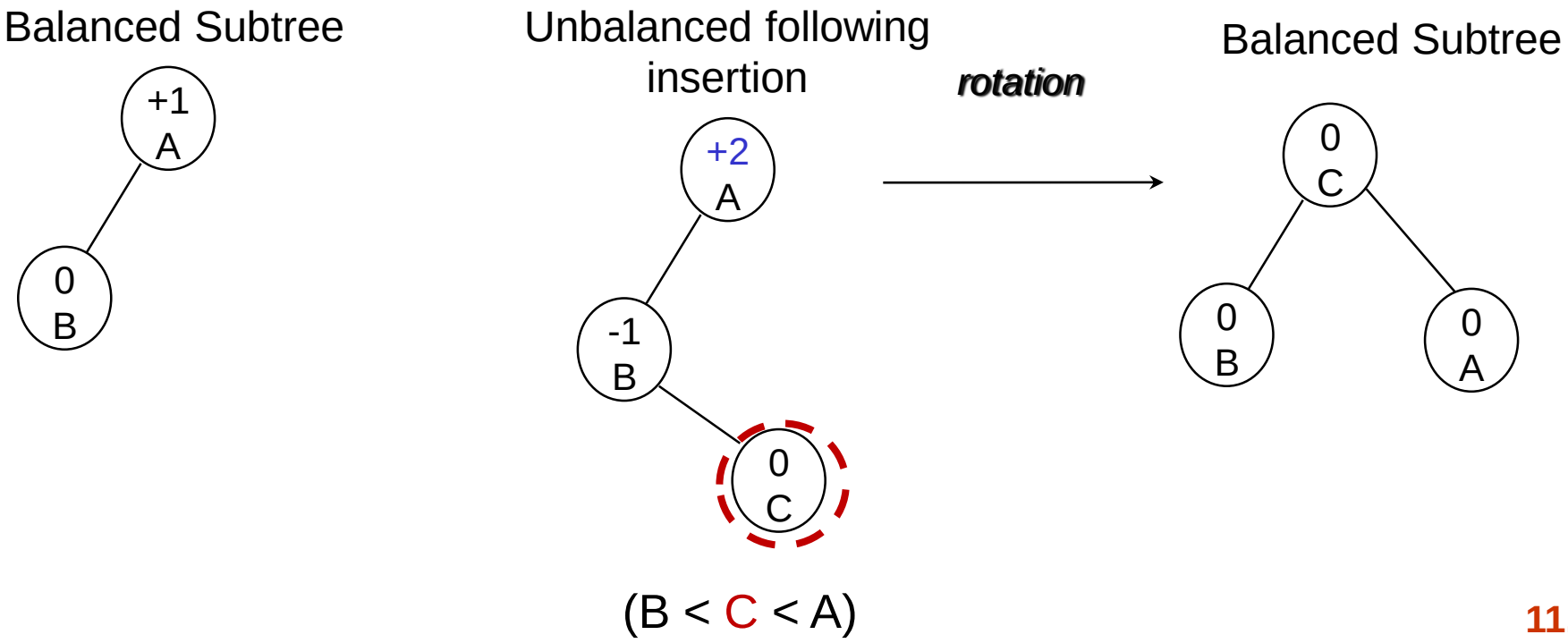
AVL Tree Concept [3/3]

참고자료

- *BalanceFactor*, BF(T), of a node T in a binary tree is h_L-h_R where h_L and h_R are the height of the left and right subtree of T

- For any node in tree T in AVL tree, BF(T) should be one of " -1, 0, 1"

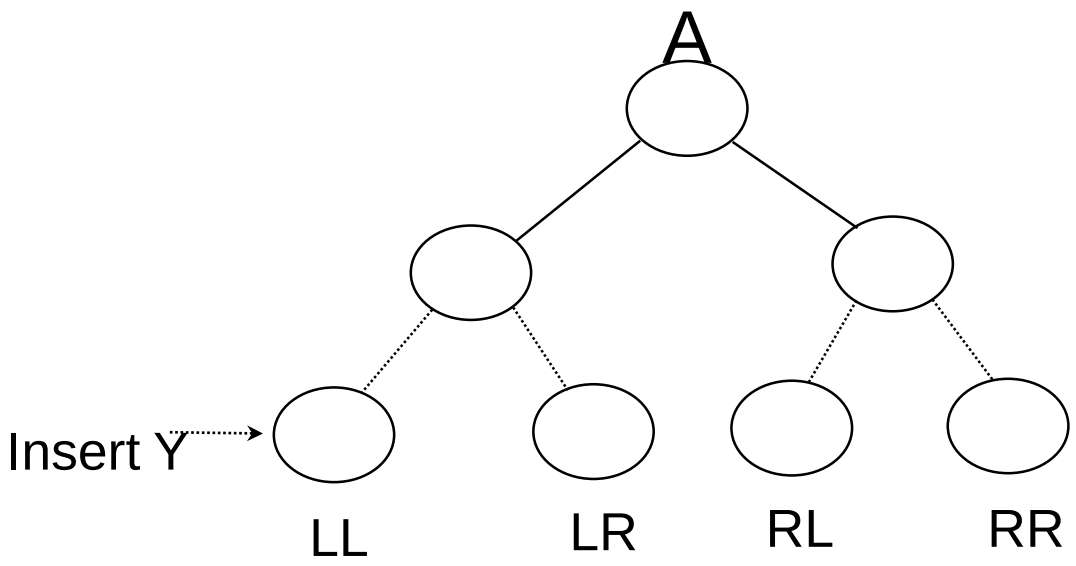
- If BF(T) is -2 or 2, then proper rotation is carried out in order to get balance



AVL Tree : Rebalancing Cases

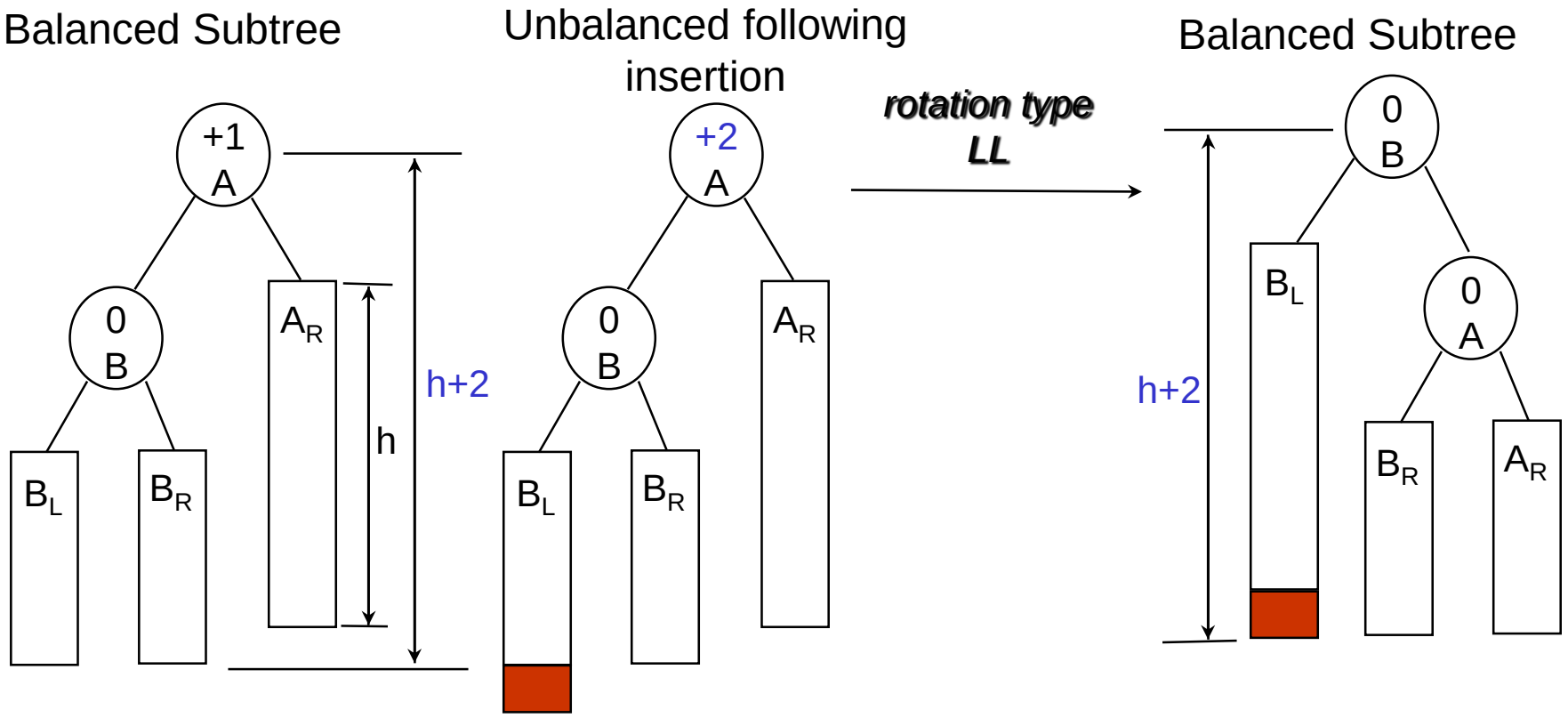
참고자료

- Rebalancing is carried out using four different kinds of rotations
  - LL case when new node Y is inserted in the left subtree of the left subtree of A
  - LR case when new node Y is inserted in the right subtree of the left subtree of A
  - RL case when new node Y is inserted in the left subtree of the right subtree of A
  - RR case when new node Y is inserted in the right subtree of the right subtree of A



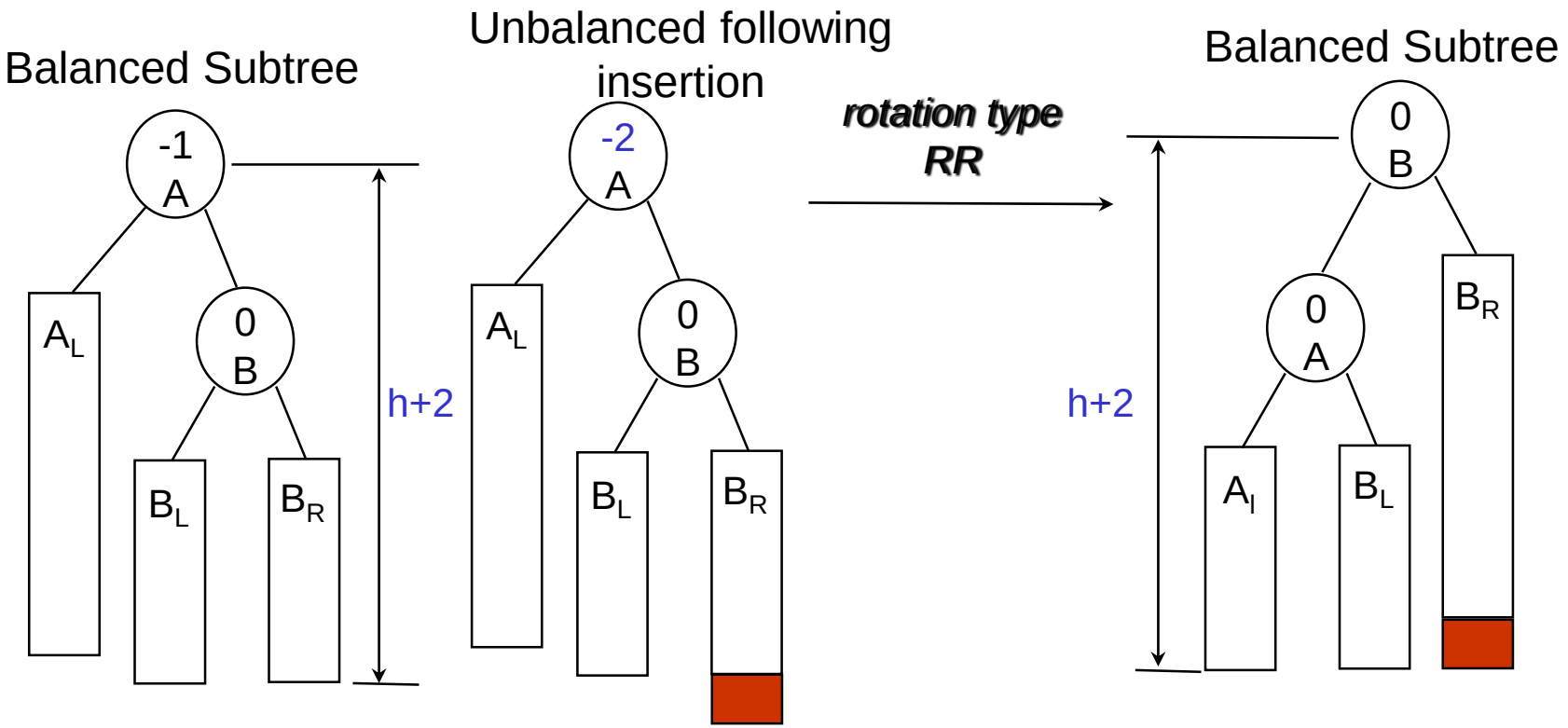
AVL Tree : Rebalancing (LL case)

참고자료



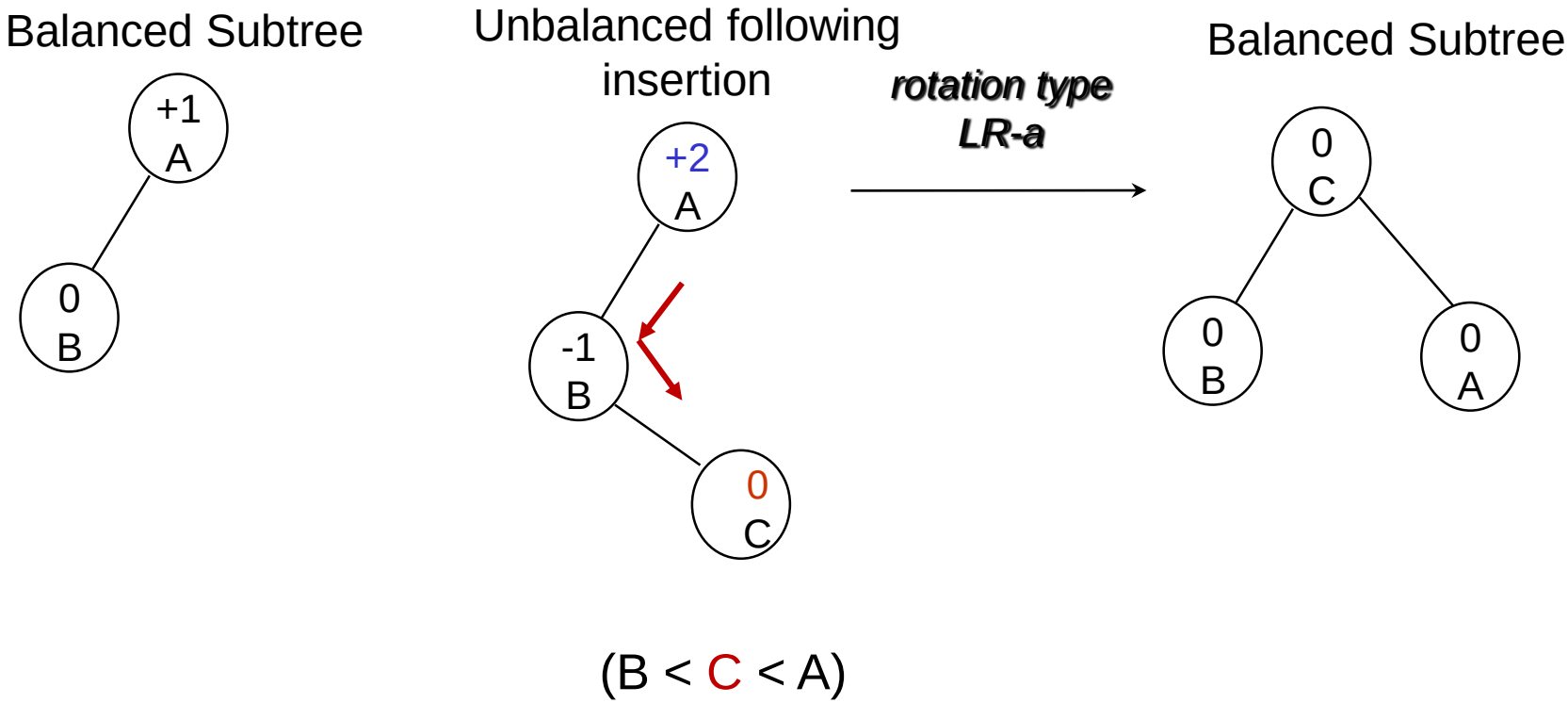
Height of B_L increase to h+1
(BL < B < BR < A < AR)

AVL Tree : Rebalancing (RR case)



Height of B_R increase to h+1
(AL < A < BL < B < BR)

AVL Tree : Rebalancing (LR-a case)

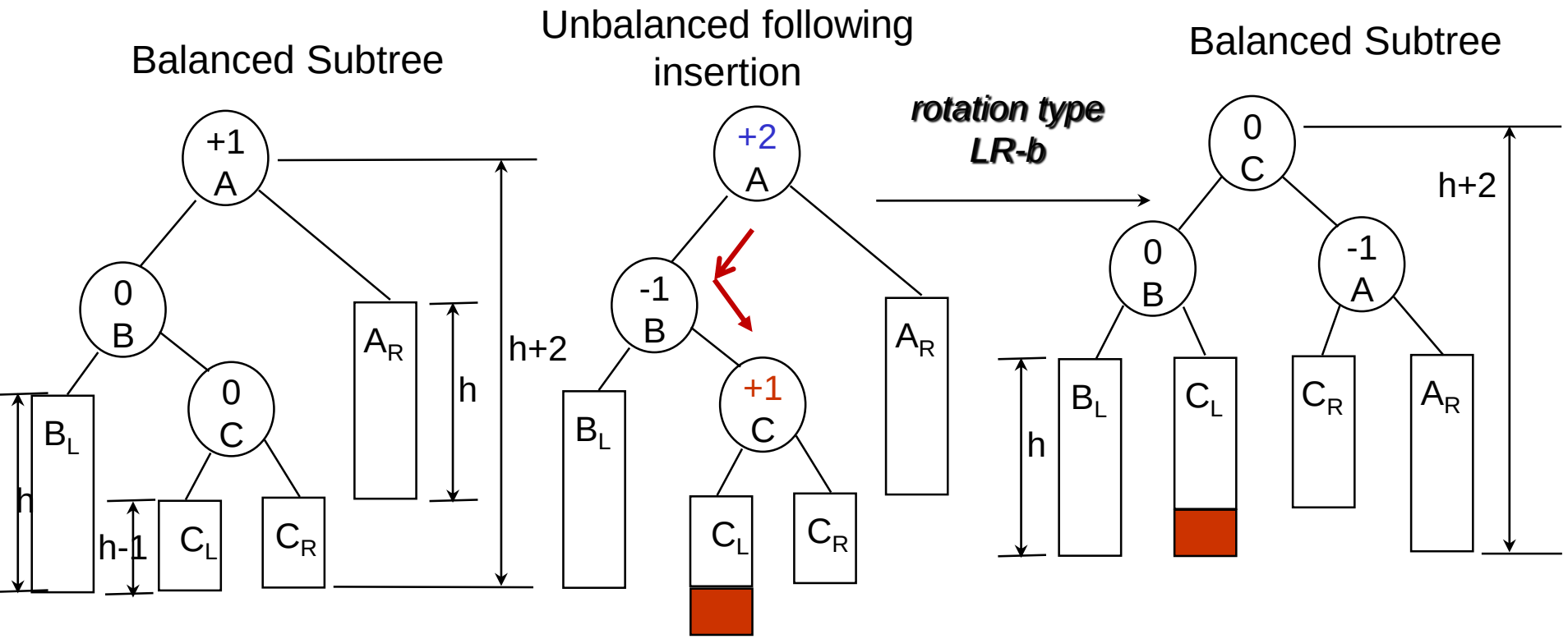


(B < C < A)

BF가 +2가 된 node를 기준으로 봐서 LR에 위치한 node의 BF가 0인 경우

AVL Tree : Rebalancing (LR-b case)

참고자료

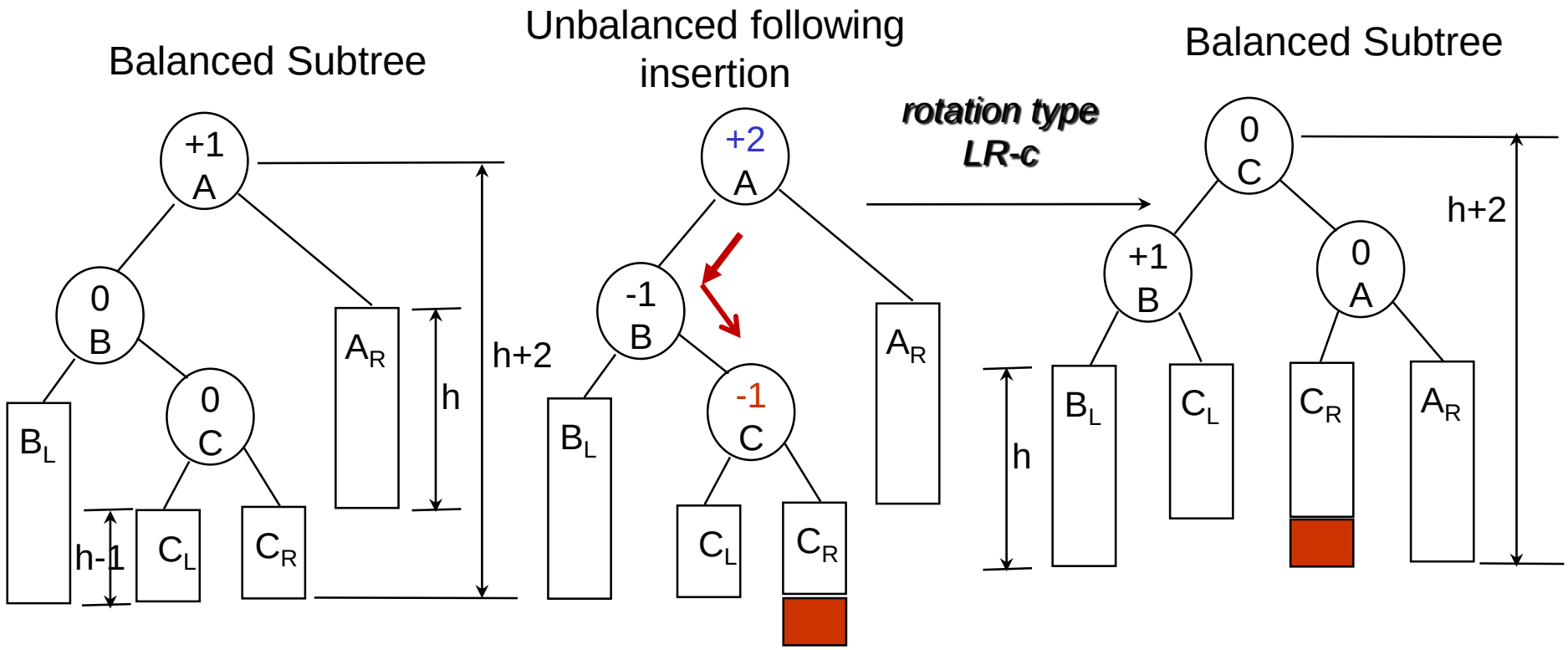


(BL < B < CL < C < CR < A < AR)

BF가 +2가 된 node를 기준으로 봐서 LR에 위치한 node의 BF가 +1인 경우

AVL Tree : Rebalancing (LR-c case)

참고자료



(BL < B < CL < C < CR < A < AR)

BF가 +2가 된 node를 기준으로 봐서 LR에 위치한 node의 BF가 -1인 경우

RL-a, RL-b and RL-c are symmetric to LR-a, LR-b and LR-c respectively

AVL Tree [1/11]

참고자료

[MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN, OCT, SEP]

New Identifier

MAR

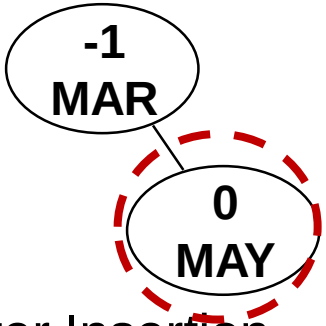
After Insertion

No Rebalancing needed

New Identifier

MAY

After Insertion

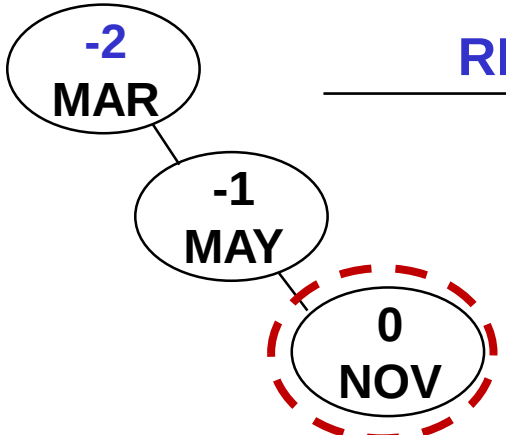


No Rebalancing needed

New Identifier

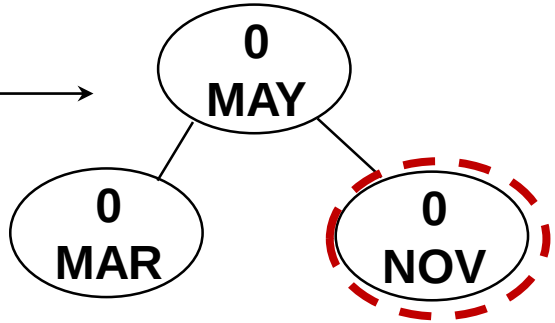
NOV

After Insertion



RR

After Rebalancing



AVL Tree [2/11]

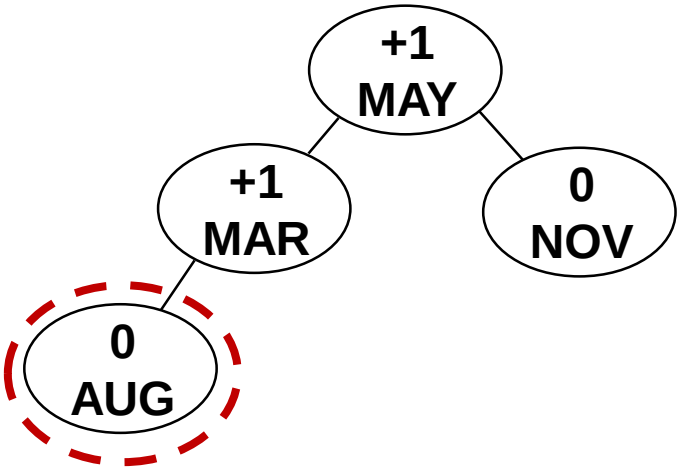
참고자료

[MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN, OCT, SEP]

New Identifier

AUG

After Insertion

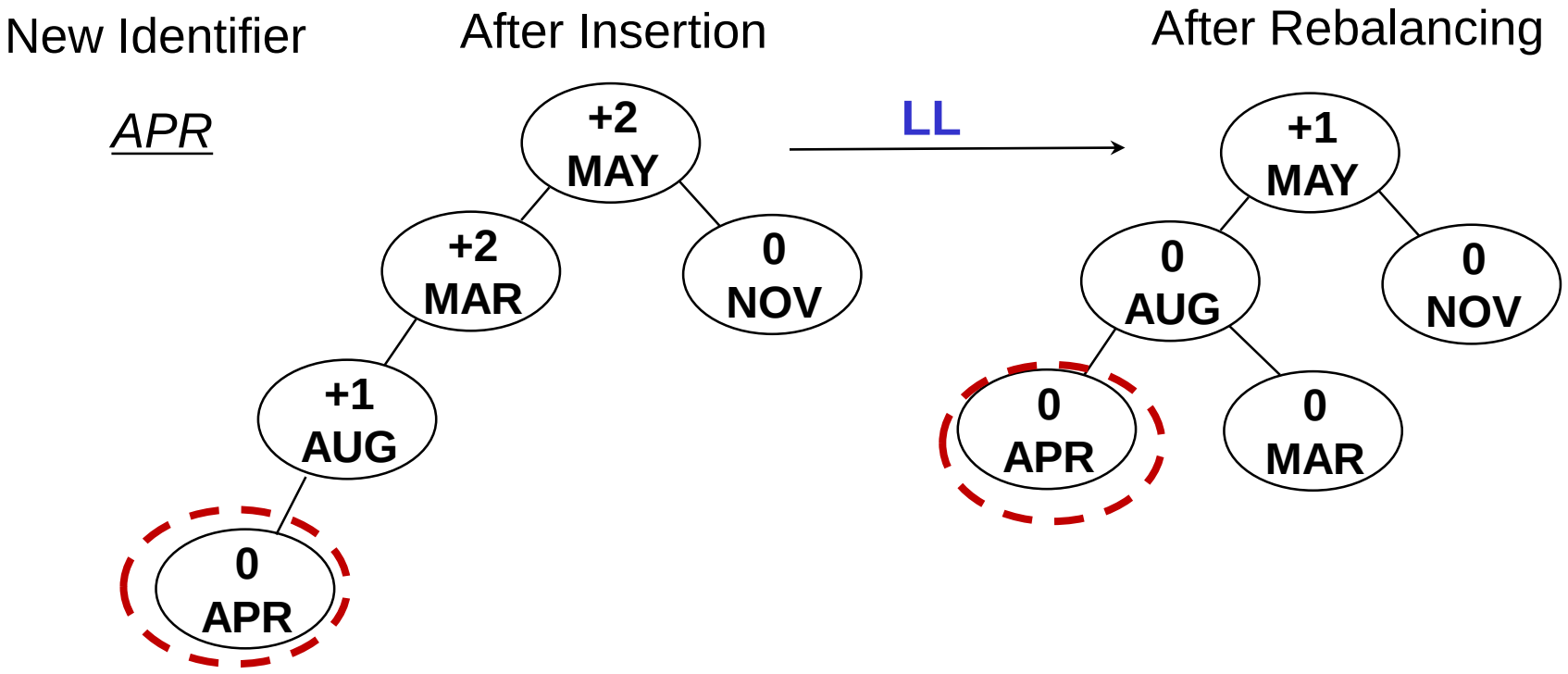


No Rebalancing needed

AVL Tree [3/11]

참고자료

[MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN, OCT, SEP]



AVL Tree [4/11]

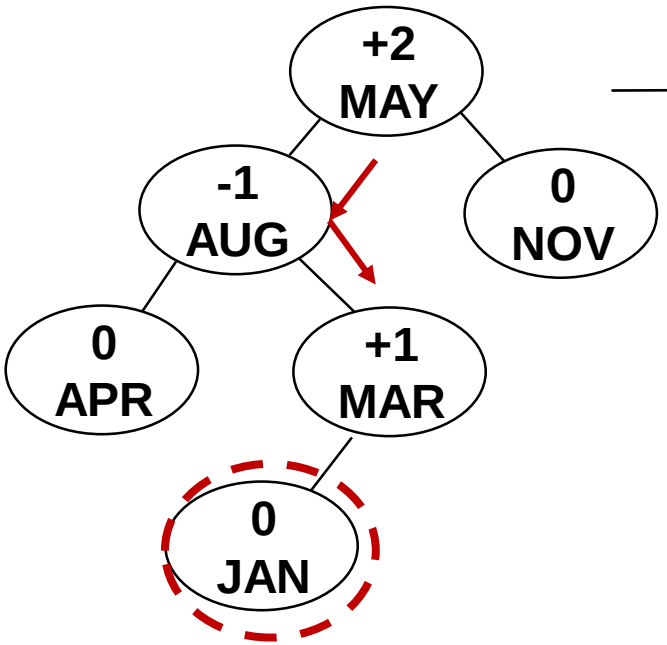
참고자료

[MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN, OCT, SEP]

New Identifier

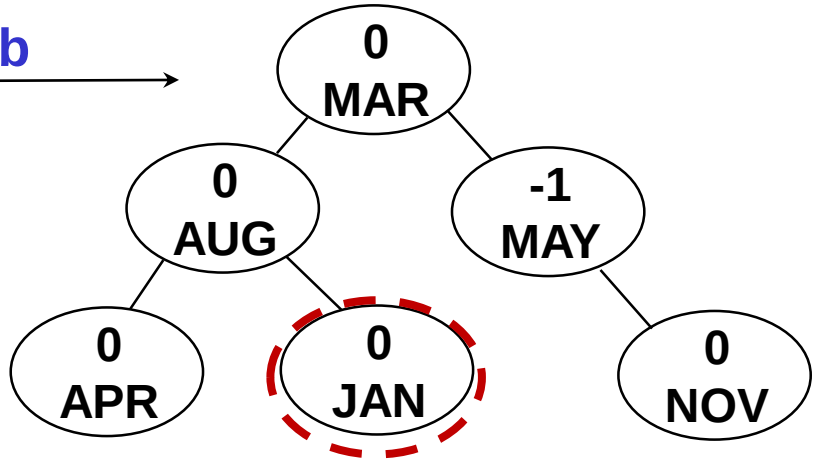
JAN

After Insertion



LR-b

After Rebalancing



AVL Tree [5/11]

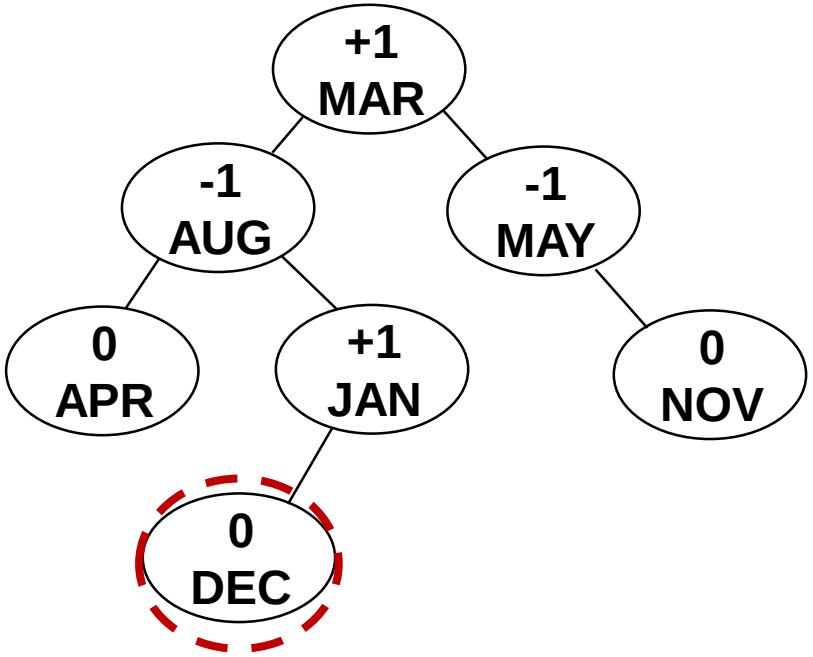
참고자료

[MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN, OCT, SEP]

New Identifier

DEC

After Insertion



No Rebalancing needed

AVL Tree [6/11]

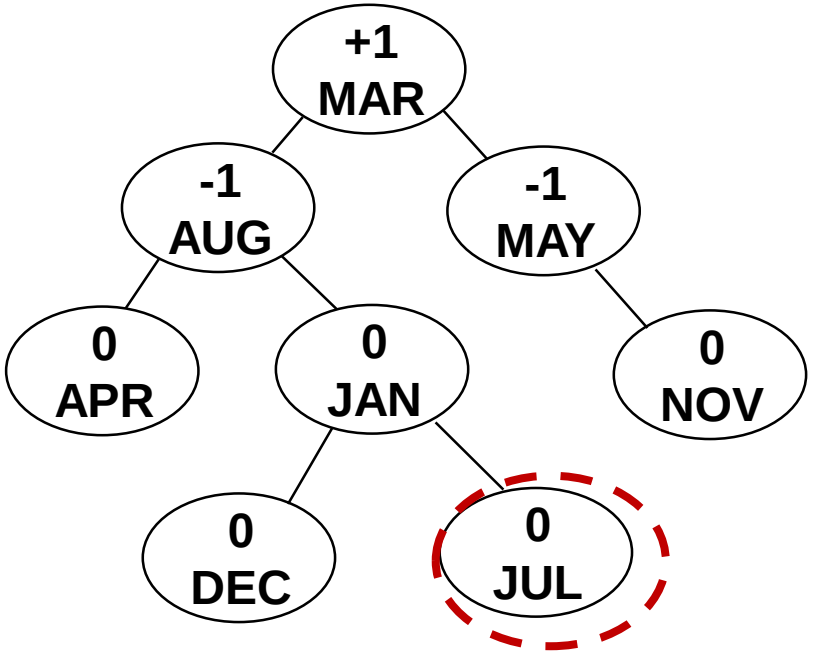
참고자료

[MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN, OCT, SEP]

New Identifier

JUL

After Insertion

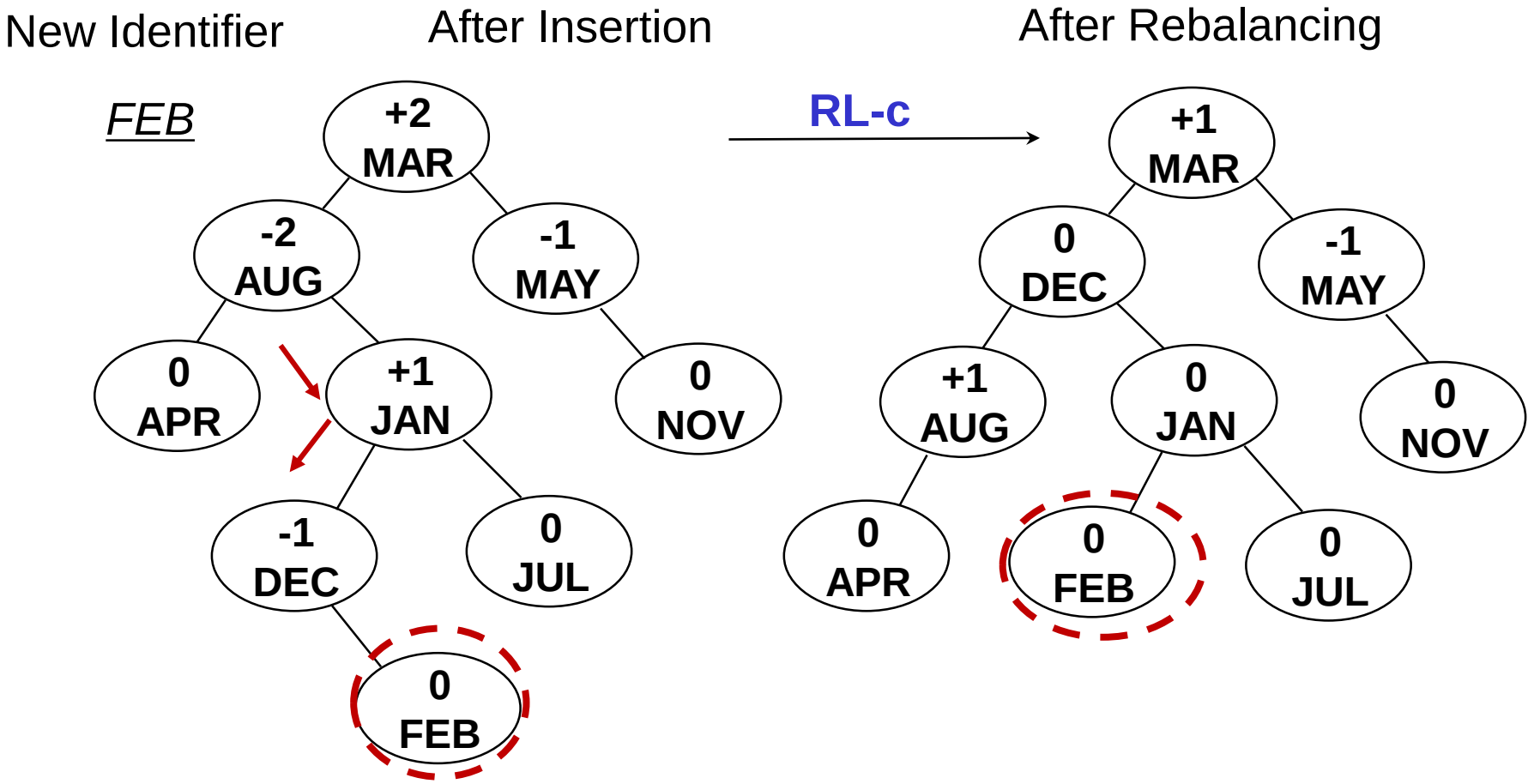


No Rebalancing needed

AVL Tree [7/11]

참고자료

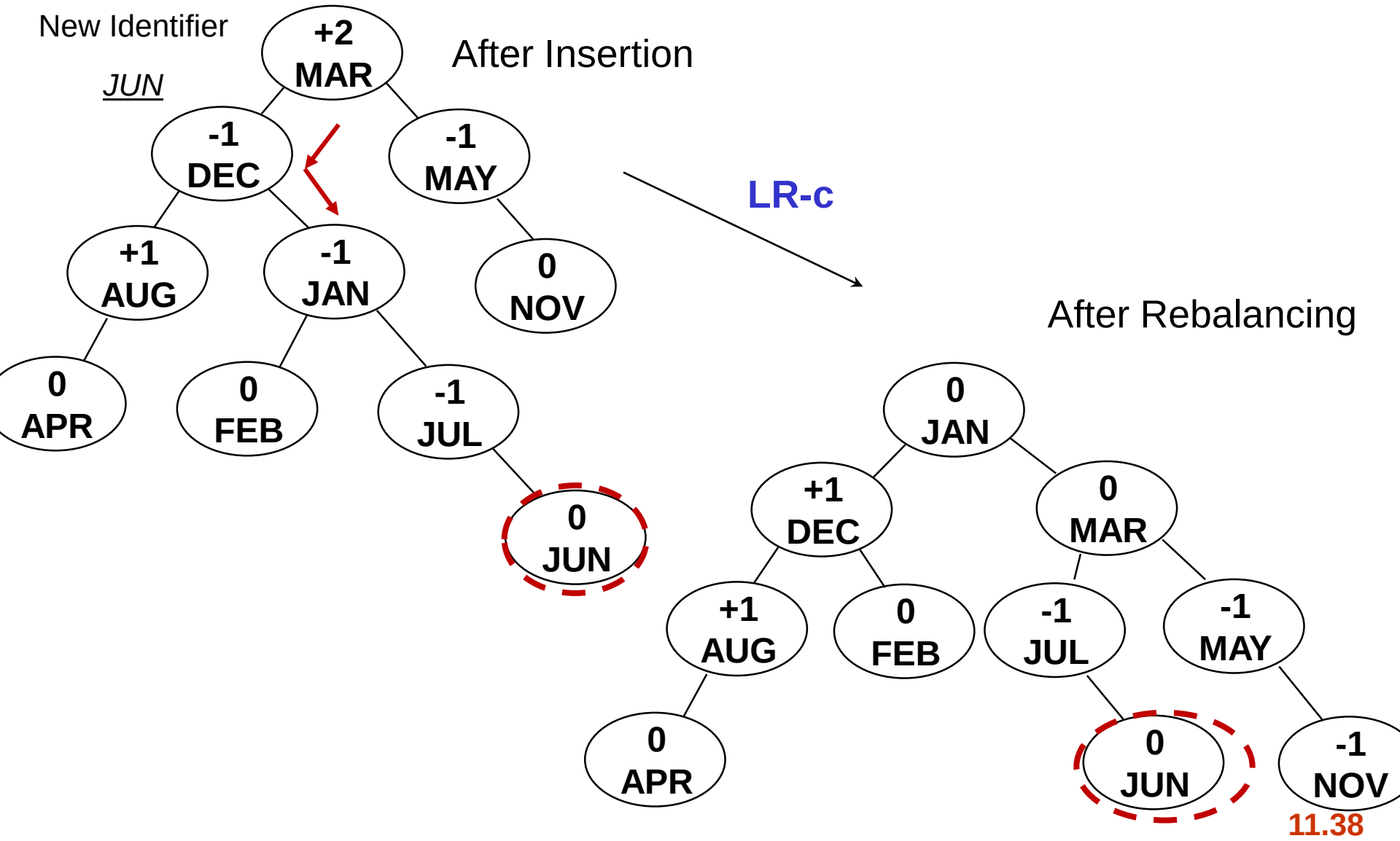
[MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN, OCT, SEP]



AVL Tree [8/11]

참고자료

[MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN, OCT, SEP]



AVL Tree [9/11]

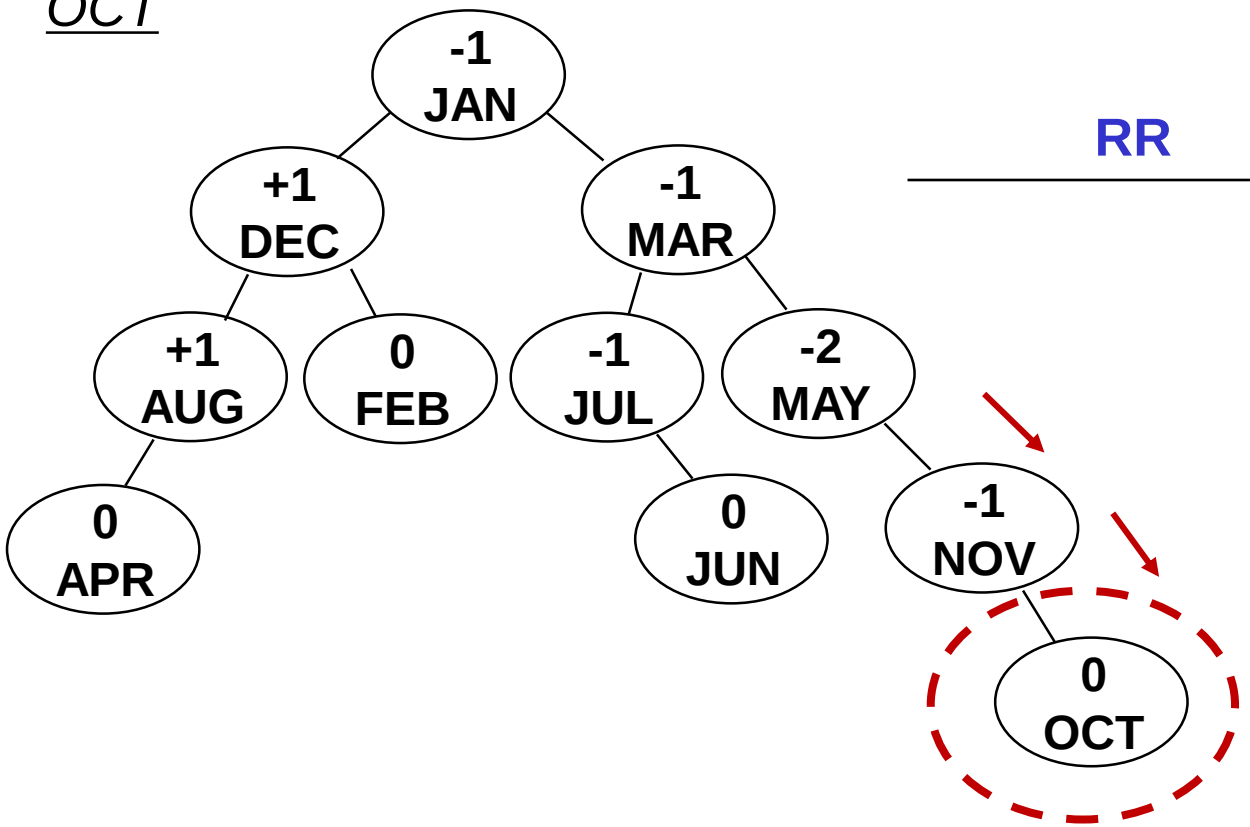
참고자료

[MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN, OCT, SEP]

New Identifier

OCT

After Insertion



RR

After Rebalancing

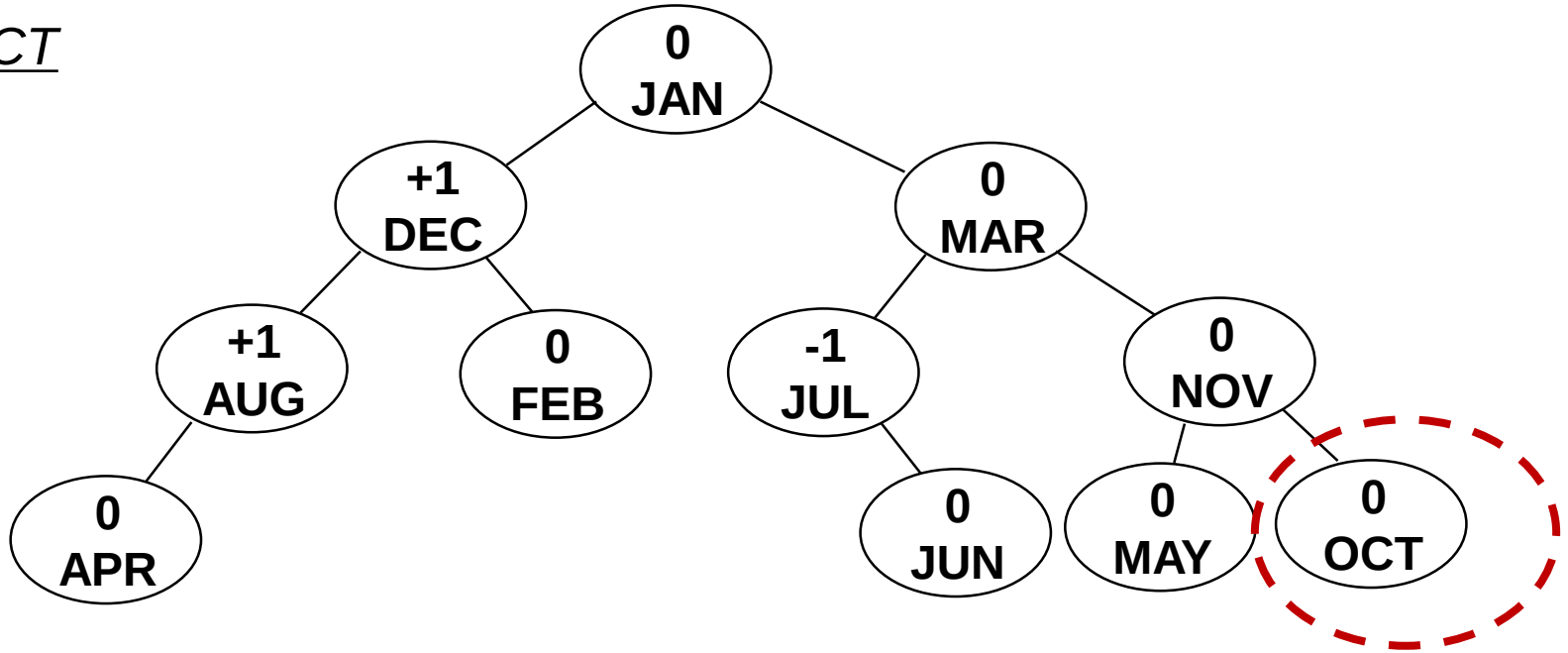
AVL Tree [10/11]

참고자료

[MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN, OCT, SEP]

New Identifier

OCT



AVL Tree [11/11]

참고자료

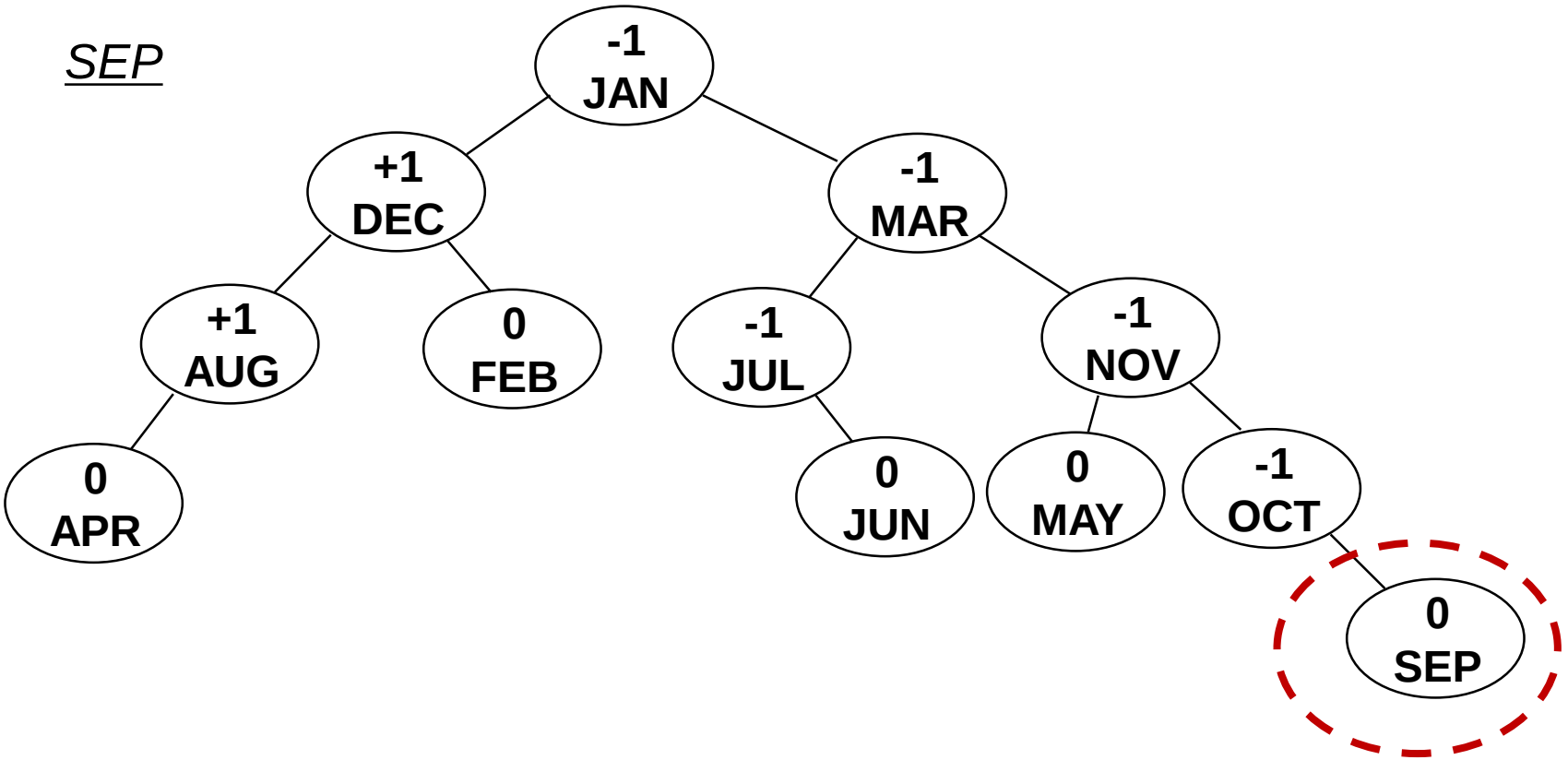
[MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN, OCT, SEP]

New Identifier

SEP

After Insertion

No Rebalancing needed

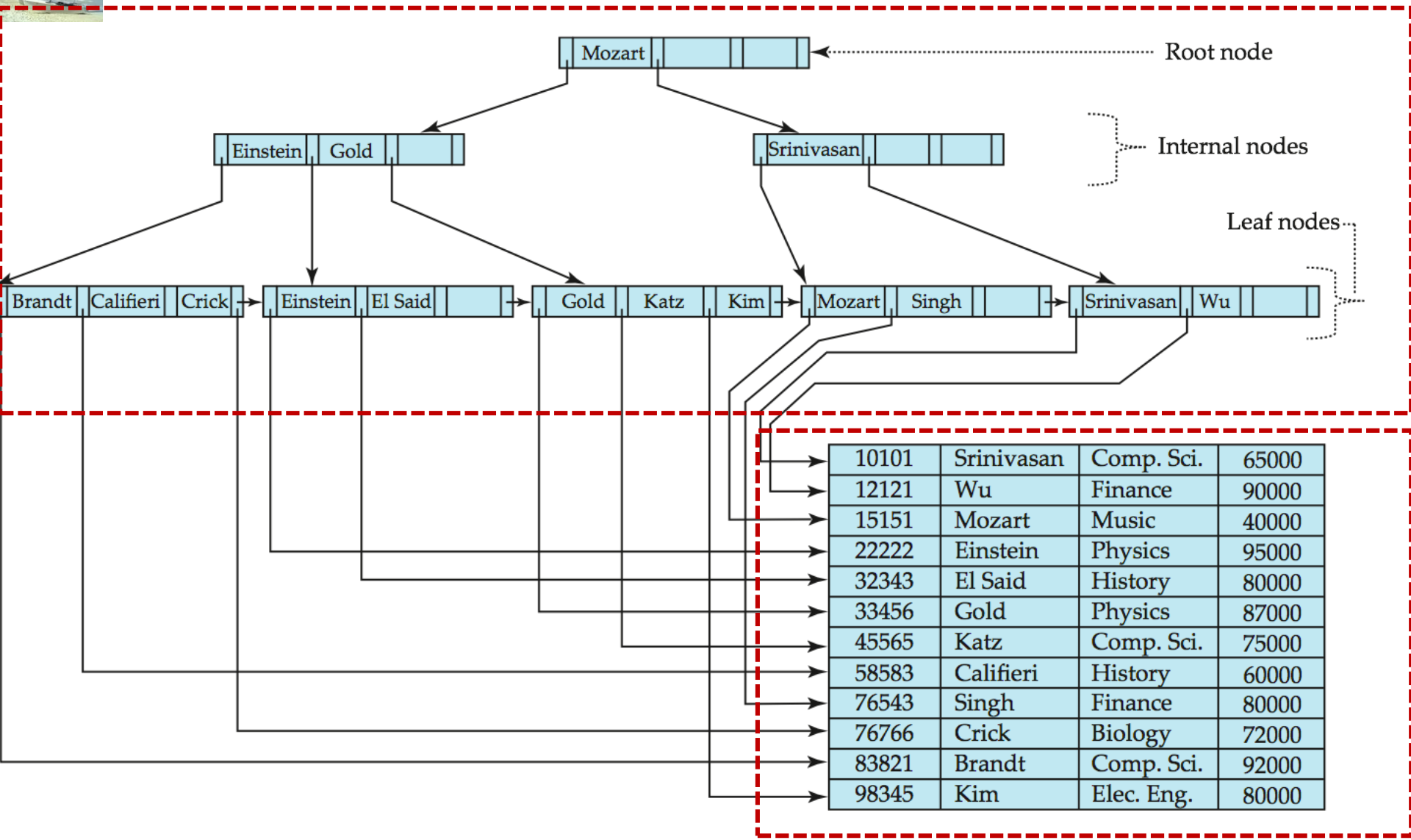


B^+-Tree Index Files

B^+-tree indices are an alternative to indexed-sequential files (ISAM files).
"B" means "Balanced"!

- Disadvantage of indexed-sequential files
  - performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.
- Advantage of B^+-tree index files:
  - automatically reorganizes itself with small local changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B^+-trees:
  - Implementation is rather complicated! (But not a problem any more!)
  - extra insertion and deletion overhead, space overhead
- Advantages of B^+-trees outweigh disadvantages
  - B^+-trees are used extensively

Example of B+-Tree [1/2]



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

B^+-Tree Index Files [2/2]

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

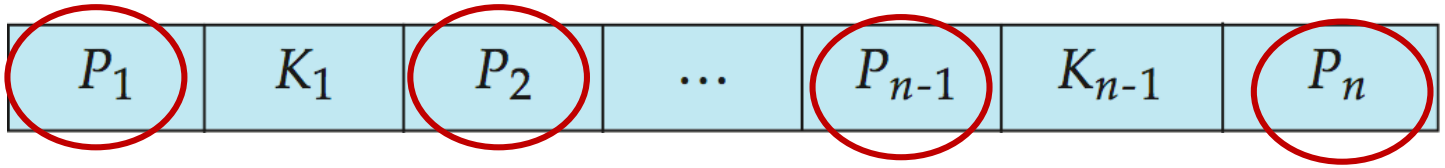
- All paths from root to leaf are of the same length (remember "Balanced"!)
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children (number of pointers in the non-root or non-leaf node)
- A leaf node has between $\lceil (n–1)/2 \rceil$ and $n–1$ values
- Special cases:
  - If the root is not a leaf, it has at least 2 children
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n–1)$ values

- For example, n = 5 인 B+-tree Index에서는
  - Node에 key 값이 최소 2개이상, child에 대한 pointer가 3개이상

n: node에서 child node들에 대해서 pointer를 n개까지 가질수 있다!
➔ n-ary tree

B^+-Tree Node Structure

- Typical node (non-leaf node)



  - 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

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

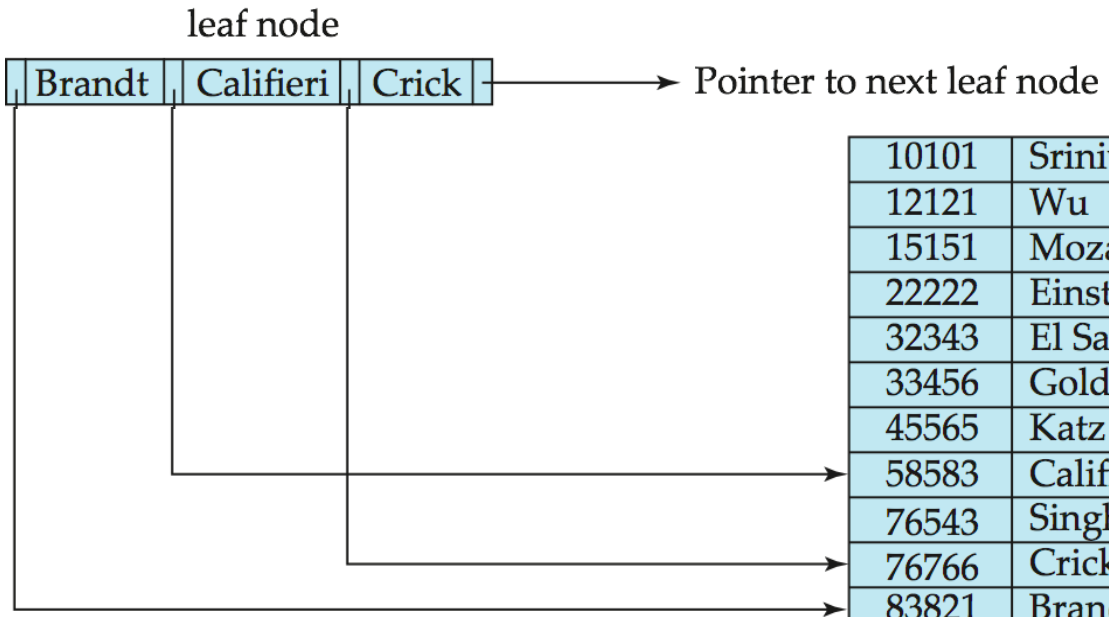
(Initially assume no duplicate keys, address duplicates later)

n: node에서 child node들에 대해서 pointer를 n개까지 가질수 있다!
➔ n-ary tree

B^+-Trees Leaf Nodes

Properties of a leaf node:

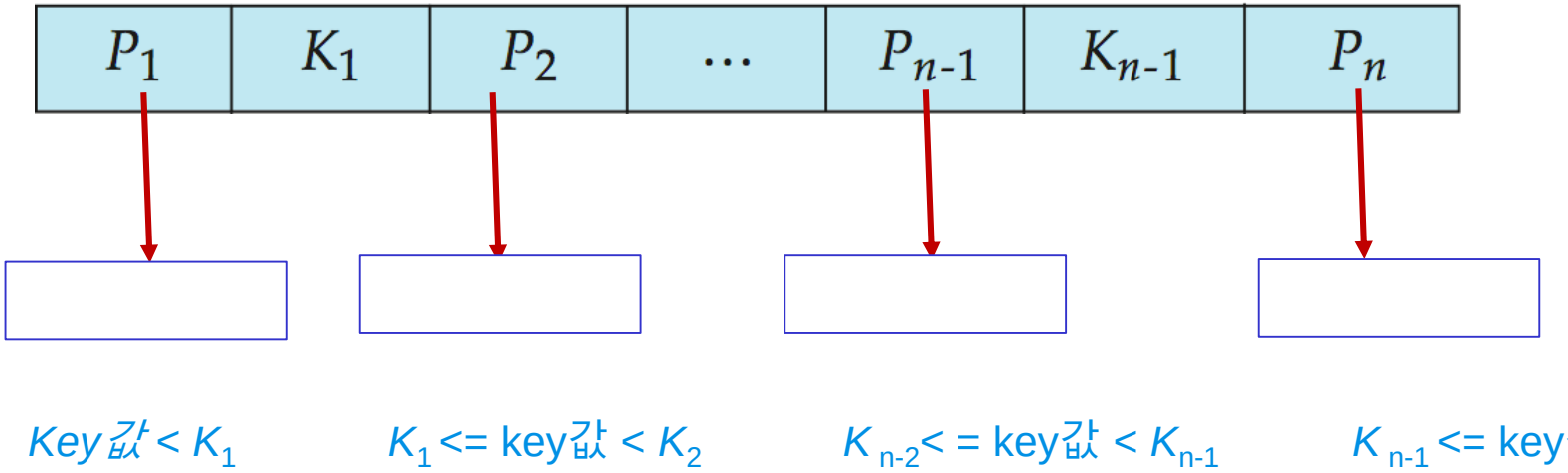
- For $i = 1, 2, \ldots, n-1$, pointer P_i either points to a file record with search-key value K_i, or to a bucket of pointers to file records, each record having search-key value K_i.
  - Only need bucket structure if search-key does not form a primary key.
- If L_i, L_j are leaf nodes and $i < j$, 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



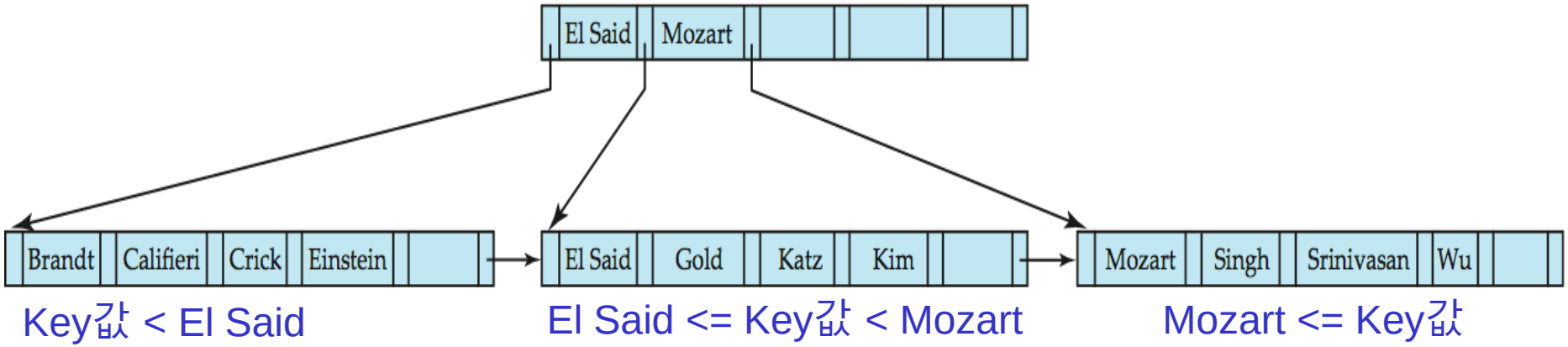
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

B+-tree Non-Leaf Nodes

- Non leaf nodes form a multi-level sparse index on the leaf nodes.
- For a non-leaf node with *m* pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search-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}



Example of B+-tree with n = 6



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

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

Observations about B+-trees

- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close
- 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.
  - If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  - thus searches can be conducted efficiently
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see)

Queries on B$^+$-Trees [1/2]

- Find record with search-key value *V*

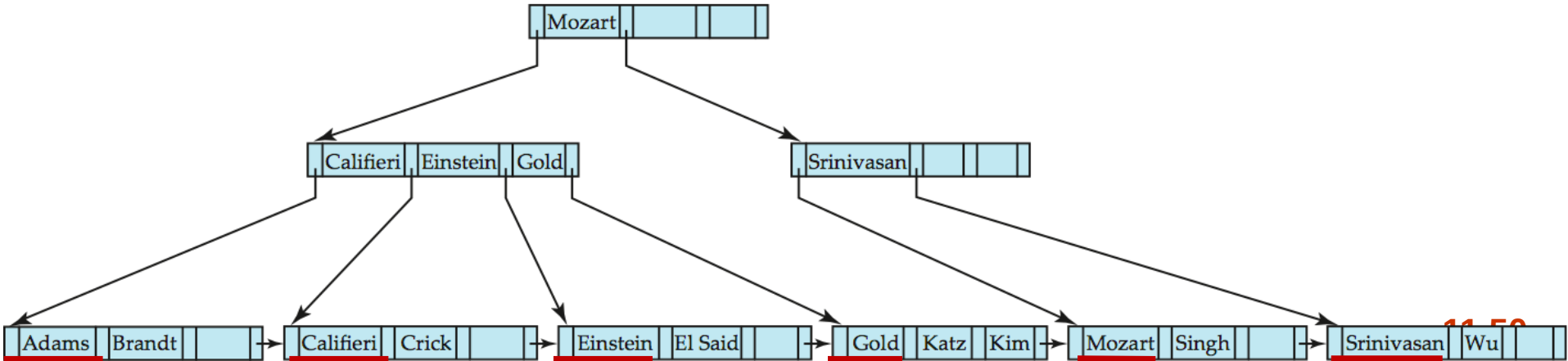
function find(value V)

1. *C=root*
2. While C is not a leaf node {
   1. Let *i* be the least value such that $V \leq K_i$.
   2. If no such i exists,

      then set *C = last non-null pointer in C*

      else { if (V= K_i) then Set C = P_{i+1} else set *C* = P_i }

   }
3. Let *i* be the least value such that $K_i = V$
4. If there is such a value *i, then* follow pointer P_i to the desired record else no record with search-key value *k* exists.



Queries on B^{+}-Trees [2/2]

- In processing a query, a path is traversed in the tree from the root to some leaf node

- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  - A node is generally the same size as a disk block, typically 4 kB (kilobytes)
    - and n is typically around 100 (40 bytes per index entry).
  - With 1 million search key values and n = 100
    - at most $log_{50}(1{,}000{,}000)$ = 4 nodes are accessed in a lookup

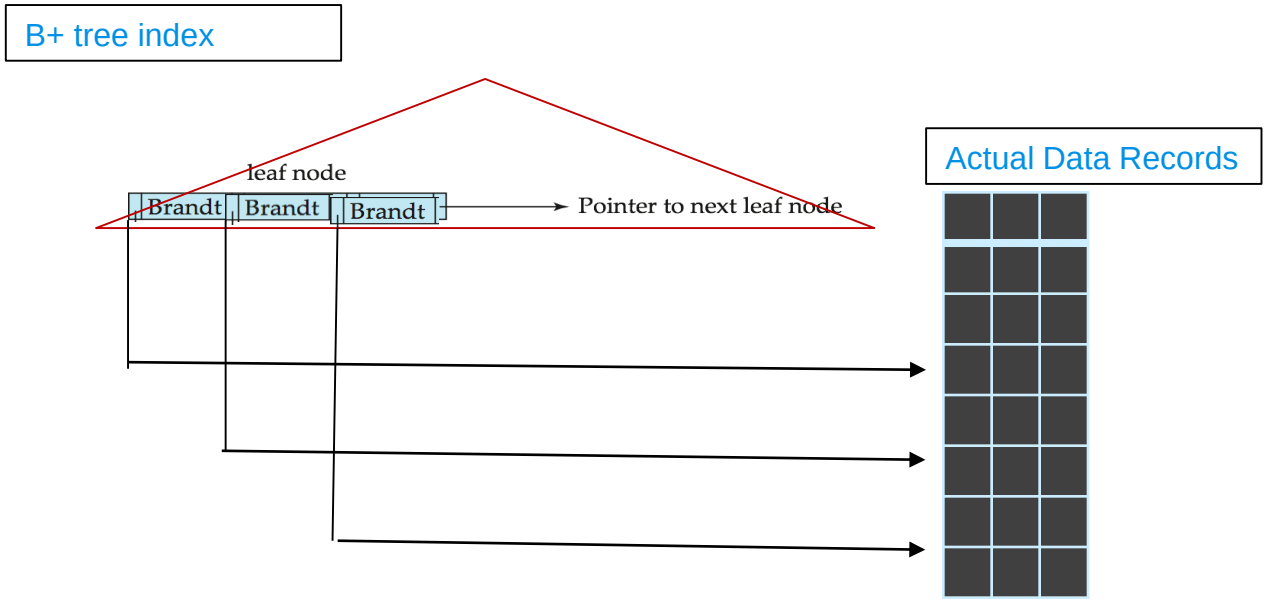
- Contrast this with a balanced binary tree with 1 million search key values
  - around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Handling Duplicate Search Keys [1/2]

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

- If duplicate search keys are allowed
  - In both leaf and internal nodes,
    - We can guarantee $K_1 \leq K_2 \leq K_3 \leq \ldots \leq K_{n-1}$
  - Search-keys in the subtree to which P_i points
    - are $\leq K_i$, but not necessarily $< K_i$
    - To see why, suppose same search key value V is present in two leaf node L_i and L_{i+1}. Then in parent node K_i must be equal to V
- We modify *find(value V)* as follows
  - traverse P_i even if $V = K_i$
  - As soon as we reach a leaf node *C,* check if *C* has only search key values less than *V*
    - if so, then set *C* = right sibling of *C* before checking whether *C* contains *V*
- Procedure *printAll(value V)*
  - uses modified find *function(value V)* to find first occurrence of *V*
  - Traverse through consecutive leaves to find all occurrences of *V*

Handling Duplicate Search Keys [2/2]

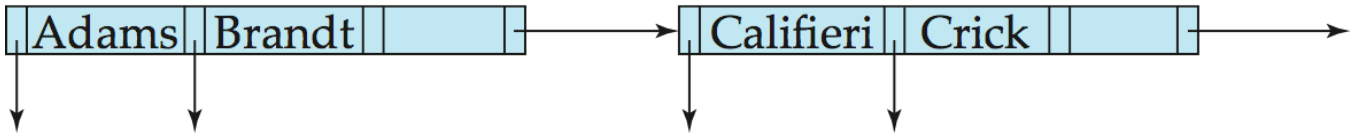


Insertion in B$^+$-Trees [1/2]

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
    1. Add record to the file
    2. If necessary, then add a pointer to the bucket.
3. If the search-key value is not present, then
    1. add the record to the main file (and create a bucket if necessary)
    2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
    3. Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide.

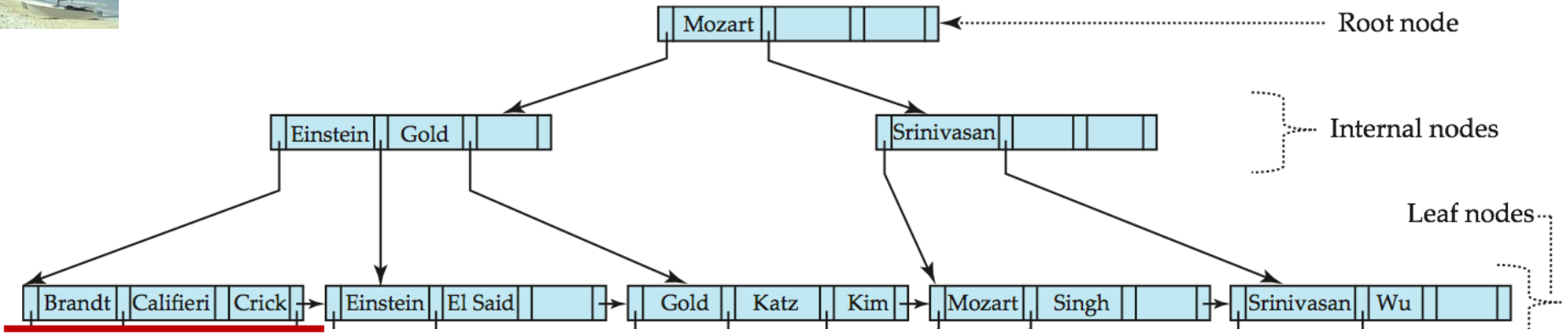
Insertion in B$^+$-Trees [2/2]

- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order.
    - Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be *p,* and let *k* be the least key value in *p.*
    - Insert (*k,p*) in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.
- Splitting of nodes 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

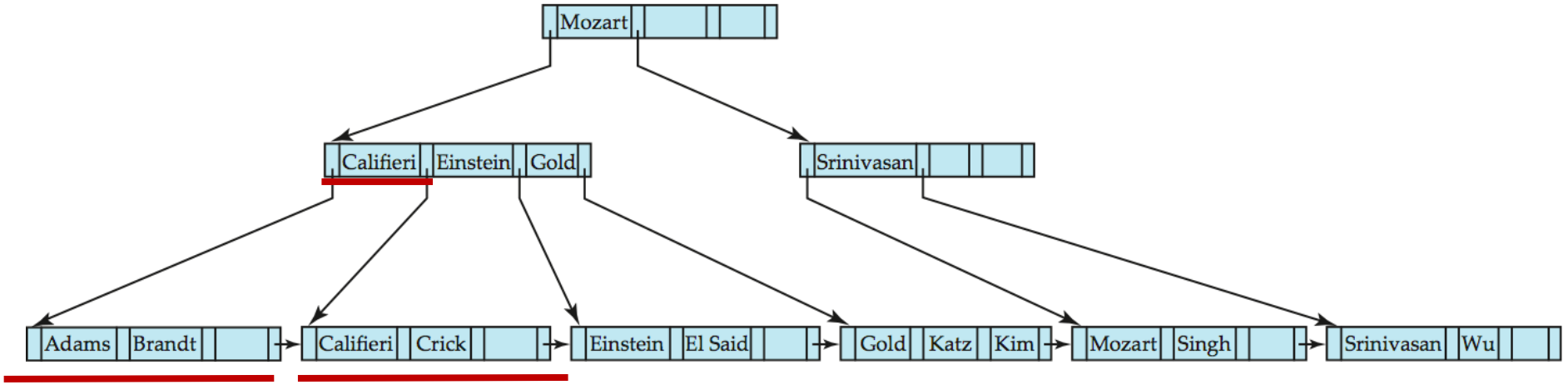


Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri,pointer-to-new-node) into parent

B+-Tree Insertion Example [1/3]

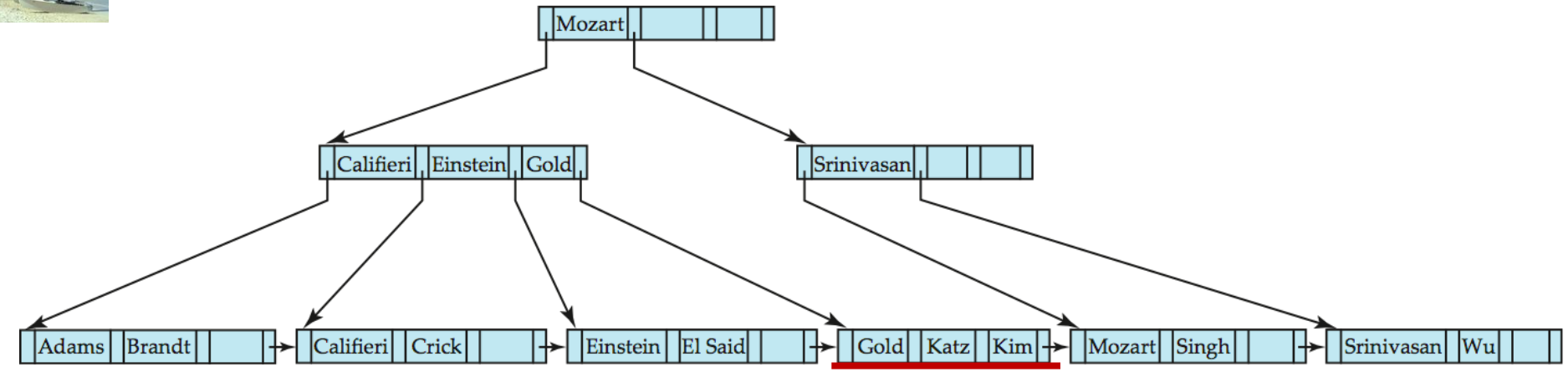


***** B+-Tree before and after insertion of "Adams" *****

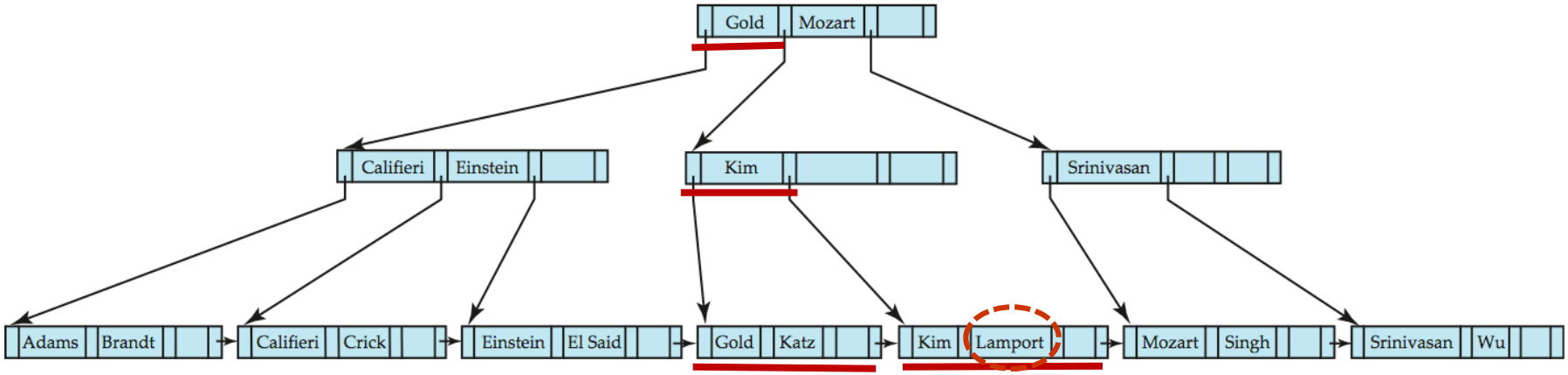


New leaf node is created ➔ new index record in the upper node created

B+-Tree Insertion Example [2/3]



*** B+-Tree before and after insertion of “Lamport” ***

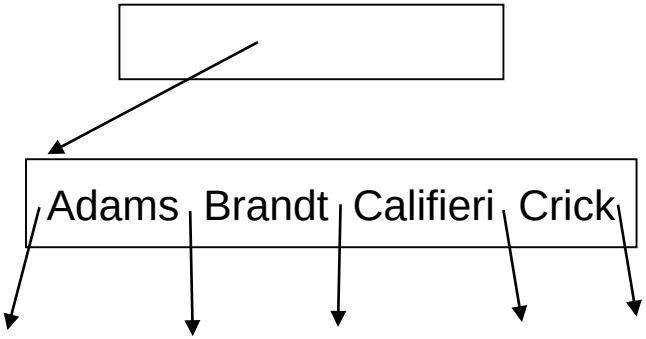


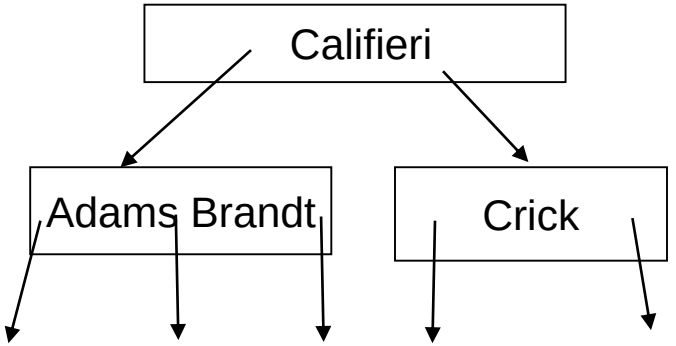
New leaf node is created ➔ new index record in the upper layer created

➔ new node in the upper layer is created ➔ new index record in the root node created

Insertion in B$^+$-Trees Example [3/3]

- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy $P_1,K_1, \ldots, K_{\lceil n/2 \rceil-1},P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil n/2 \rceil+1},K_{\lceil n/2 \rceil+1},\ldots,K_n,P_{n+1}$ from M into newly allocated node N'
  - Insert $(K_{\lceil n/2 \rceil},N')$ into parent N



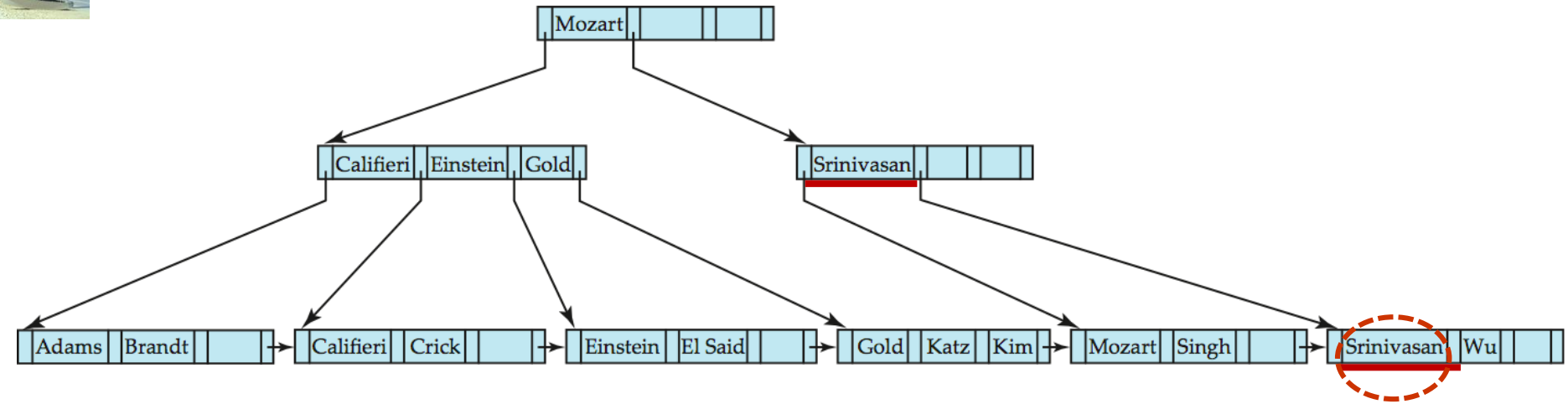


- **Read pseudocode Figure 11.5 in book!**

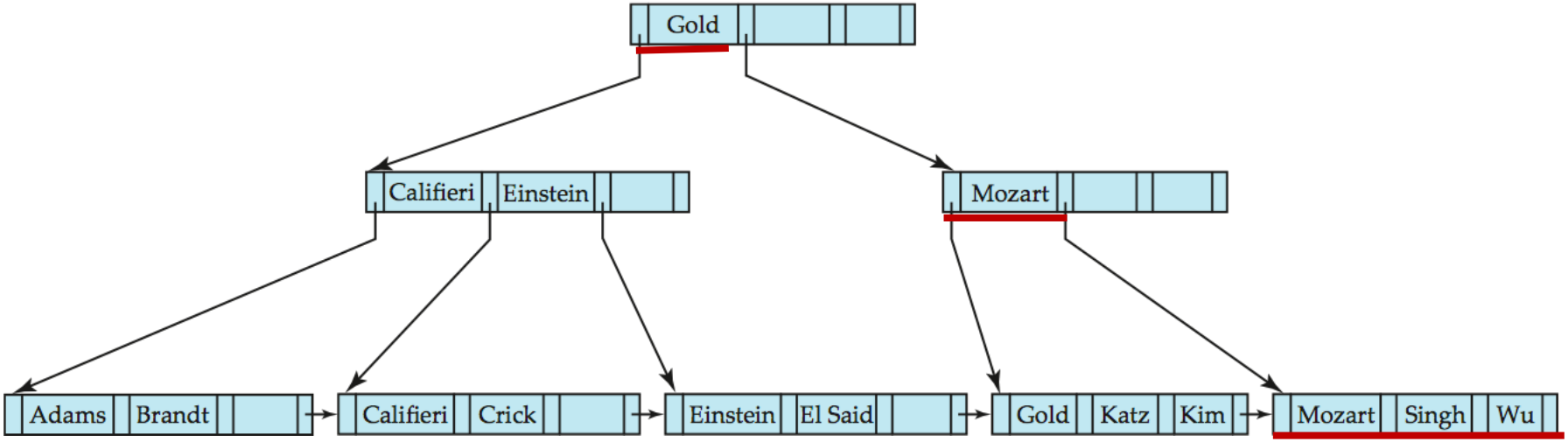
Deletion in B$^+$-Trees

- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings*:**
  - Insert all the search-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.
- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the 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 ➔ Shorten the height of B+ tree

Examples of B+-Tree Deletion [1/3]

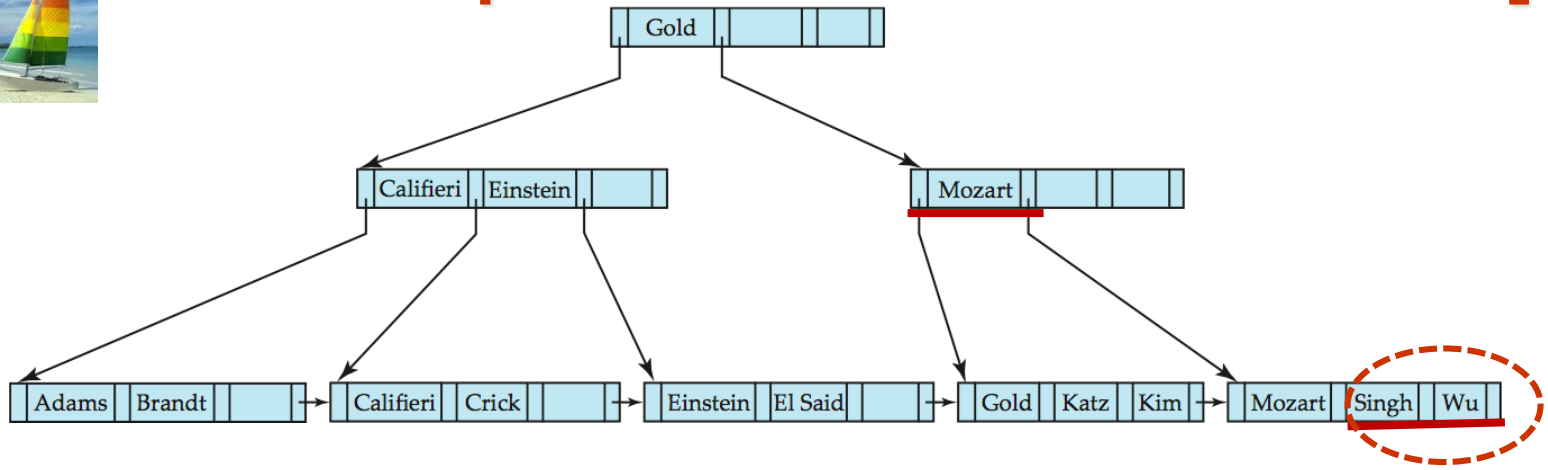


*** Before and after deleting “Srinivasan” ****

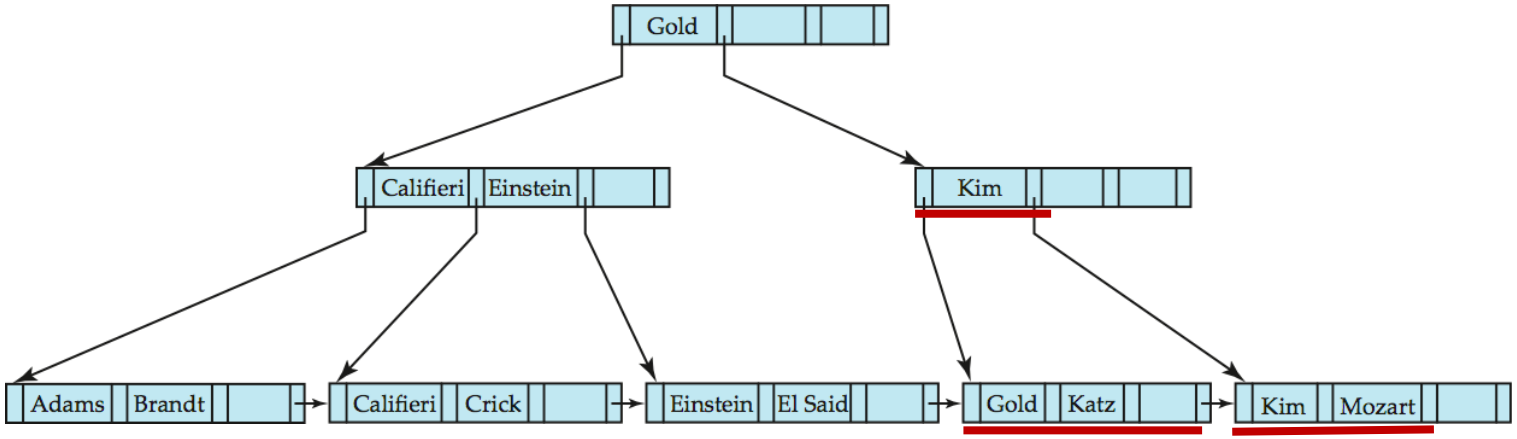


- Deleting “Srinivasan” causes merging of under-full leaves

Examples of B+-Tree Deletion [2/3]

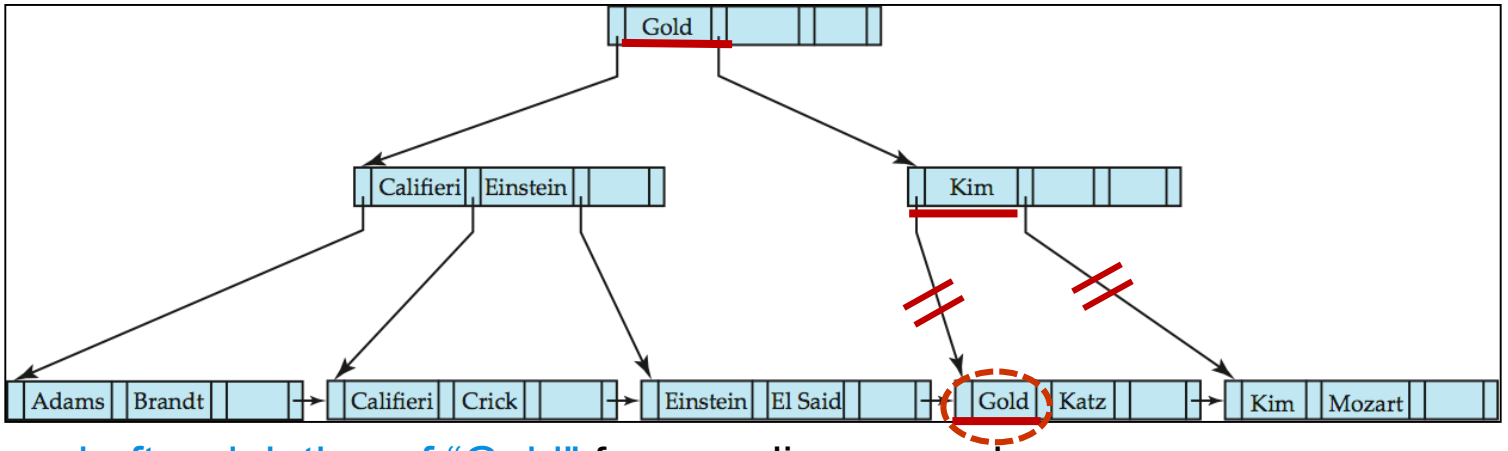


*** Before and After of Deleting “Singh” and “Wu” ***

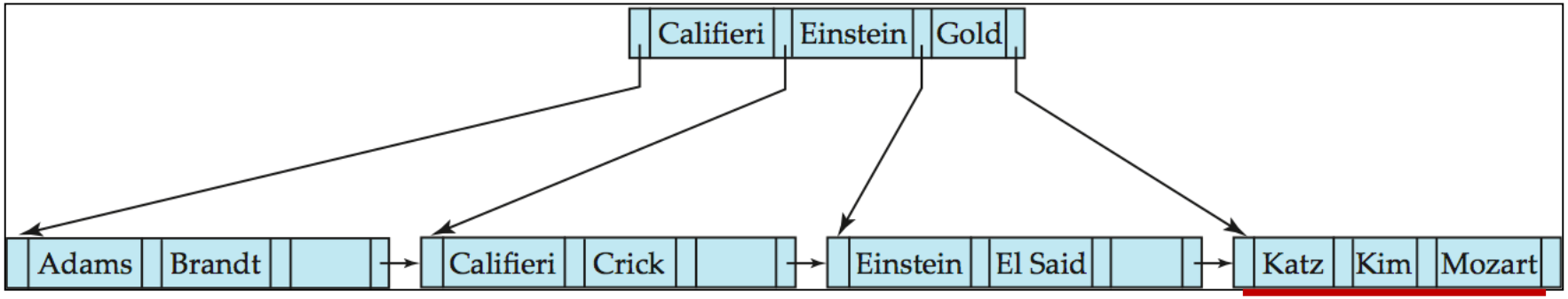


- Leaf containing Singh and Wu became underfull, and borrowed a value Kim from its left sibling
- Search-key value in the parent changes as a result

Example of B+-tree Deletion [3/3]



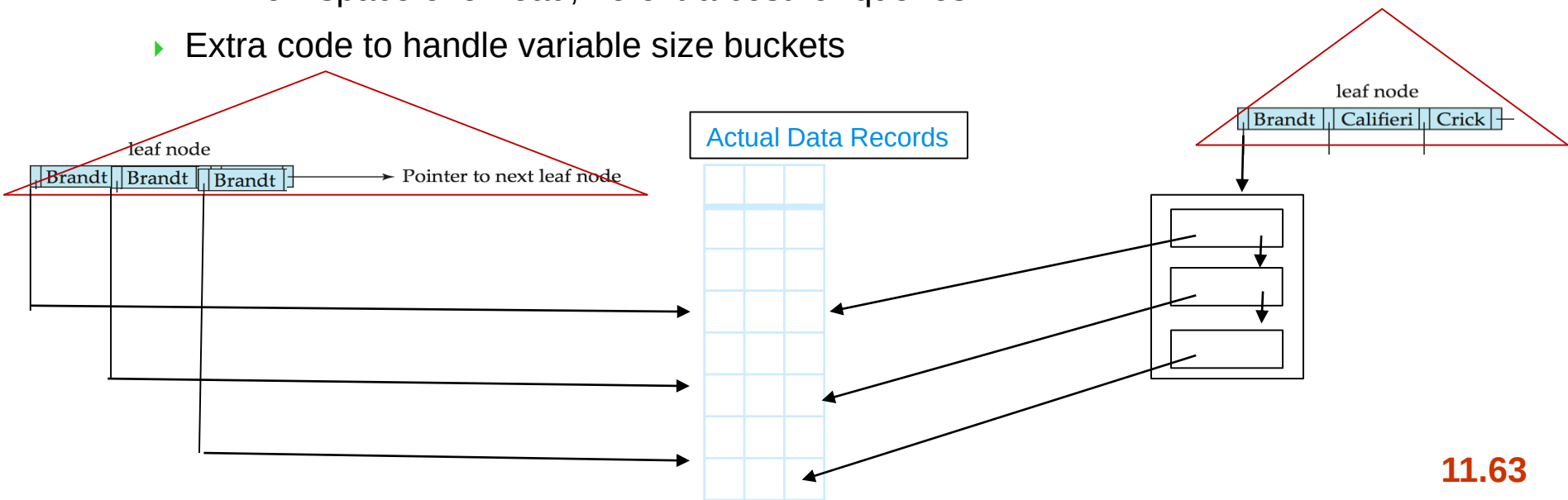
Before and after deletion of "Gold" from earlier example



- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
  - Value separating two nodes (at the parent) is pulled down when merging
  - "Gold index record" in the root is not changed because "Katz" is greater than "Gold"
- Root node then has only one child, and the root should be deleted
- The B+ tree needs to be shrinked!

Non-Unique Search Keys

- What if more than one index record contain the same search key?
  - Store duplicate index records in the leaf nodes of B+ tree
    - Deletion of a tuple can be expensive if there are many duplicates
  - Make search key of an index record unique by adding a record-identifier
    - Uniquifier attribute (Eg, EmplyeeName + BirthDate)
    - Extra storage overhead for keys
    - Simpler code for insertion/deletion (Widely used method)
  - Maintain a bucket of record pointers with a search key value
    - Store each key value only once
      - Low space overhead, no extra cost for queries
    - Extra code to handle variable size buckets



Chapter 11: Indexing and Hashing

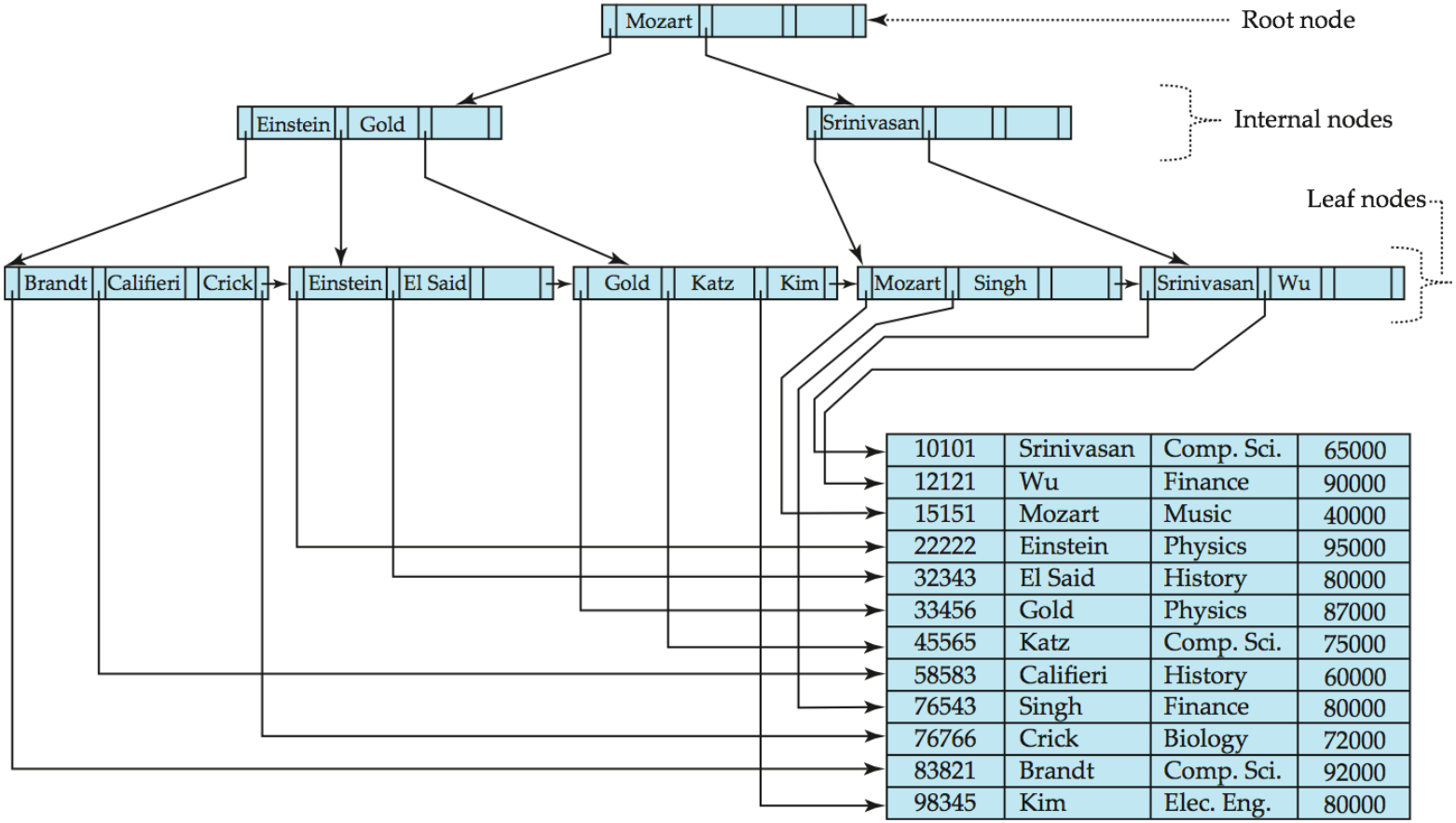
- 11.1 Basic Concepts
- 11.2 Ordered Indices
- 11.3 B^+-Tree Index Files
- 11.4 B^+-Tree Extensions
- 11.5 Multiple-Key Access
- 11.6 Static Hashing
- 11.7 Dynamic Hashing
- 11.8 Comparison of Ordered Indexing and Hashing
- 11.9 Bitmap Indices
- 11.10 Index Definition in SQL

Tree 기반

Hashing 기반

B^+-Tree File Organization [1/2]

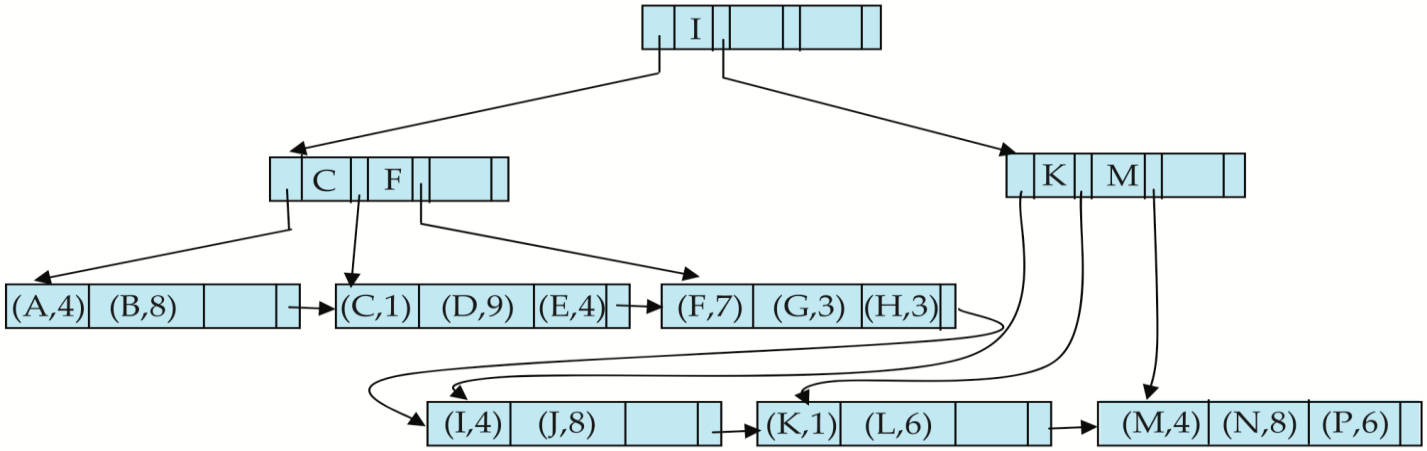
- Previously B+ tree for Index Records and Sequential File for Data Records



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

- B+ tree of index records are dynamically managed, but sequential file of data records may look ugly after intensive insertions and deletions ➔ Data file degradation problem
- B+ tree index file may include data records ➔ B+-tree file organization

B+-Tree File Organization [2/2]



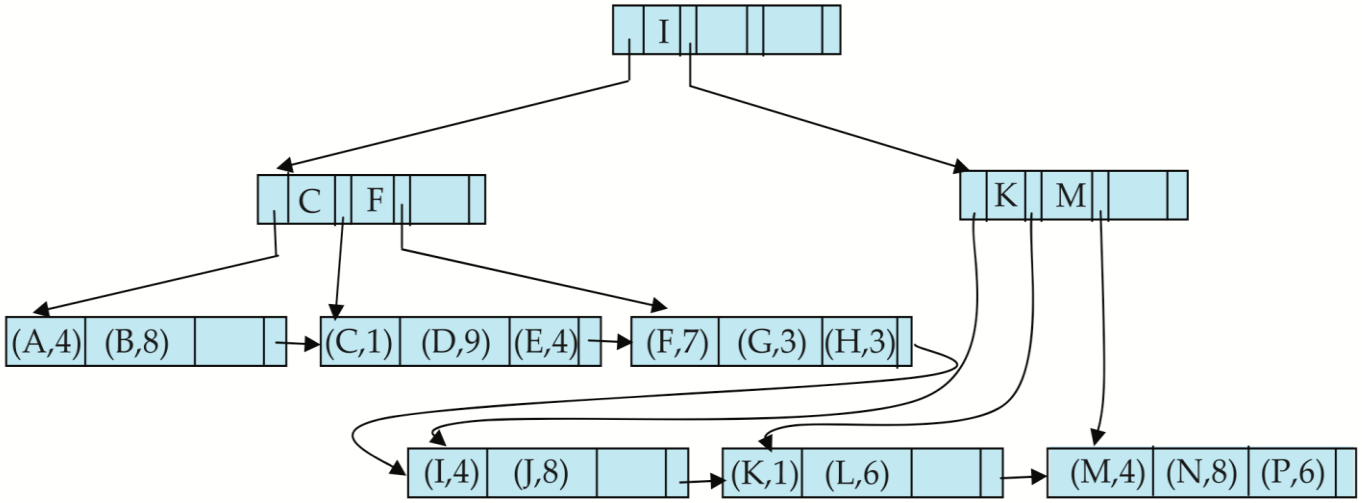
- Data file degradation problem in the previous page is solved by using B+-Tree File Organization
  - The leaf nodes in a B+-tree file organization store records, instead of pointers.
  - Leaf nodes are still required to be half full
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B+-tree index.
- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries

Record Relocation and Secondary Indices
in B+-tree file organizations

- In B+-tree file organizations, location of data records can be changed because of B+ tree property
- If a data record is relocated in B+ -tree file organization, all secondary indices that store record pointers have to be updated
- Therefore, node splits in B+-tree file organizations become very expensive
  - Extra care must be exercised
- *Solution*: use primary-index search key instead of record pointer in secondary index

Secondary key value	Pointer
Biology	pointer to a data record
EE	pointer to a data record

Secondary key value	Pointer
Biology	K
EE	F



Indexing Strings in B+ tree Index

- If B+ tree index uses variable length strings as keys
  - The node-fanout would be variable
  - May use space utilization as criterion for splitting, not number of pointers
- **Prefix compression** for better fanout of nodes
  - Key values at internal nodes can be prefixes of full key
    - Keep minimum enough characters to distinguish entries in the subtrees separated by the key value
      - E.g. “Silas” and “Silberschatz” can be separated by “Silb”
  - Keys of index records in leaf node can be compressed by sharing common prefixes
  - Leaf node에서의 index record들의 key 값에는 full key를 사용
- The fanout of node: 한 node에 저장하는 key의 갯수

B+-tree Bulk Loading and Bottom-Up Build

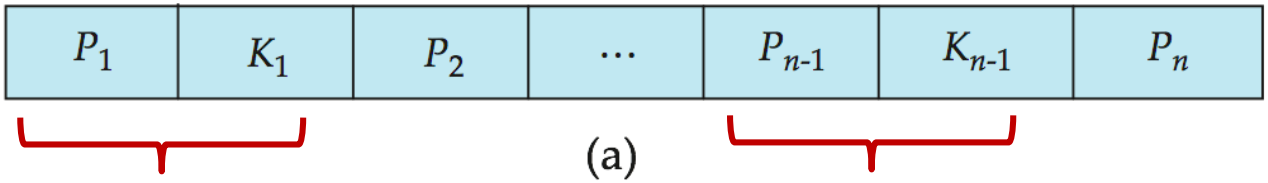
- Inserting entries one-at-a-time into a B^+-tree requires ≥ 1 IO per entry
  - assuming leaf level does not fit in memory
  - can be very inefficient for loading a large number of entries at a time (**bulk loading**)

- Efficient alternative 1:
  - sort entries first (using efficient external-memory sort algorithms discussed later in Section 12.4)
  - insert in sorted order
    - insertion will go to existing page (or cause a split)
    - much improved IO performance, but most leaf nodes half full

- Efficient alternative 2: **Bottom-up B^+-tree construction**
  - As before, sort entries first
  - And then create tree layer-by-layer, starting with leaf level
    - details as an exercise
  - Implemented as part of bulk-load utility by most database systems

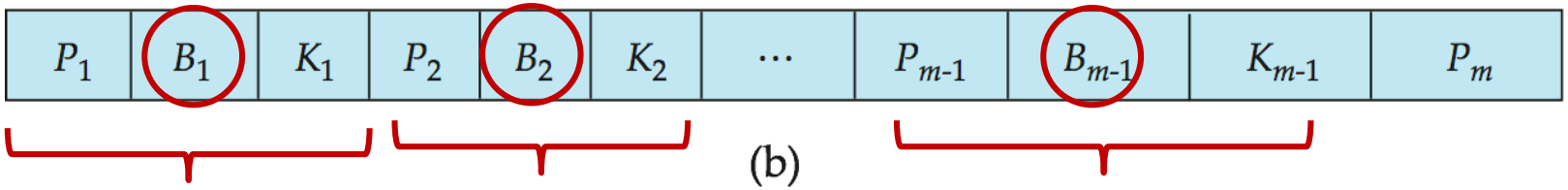
B-Tree Index Files [1/3]

- 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



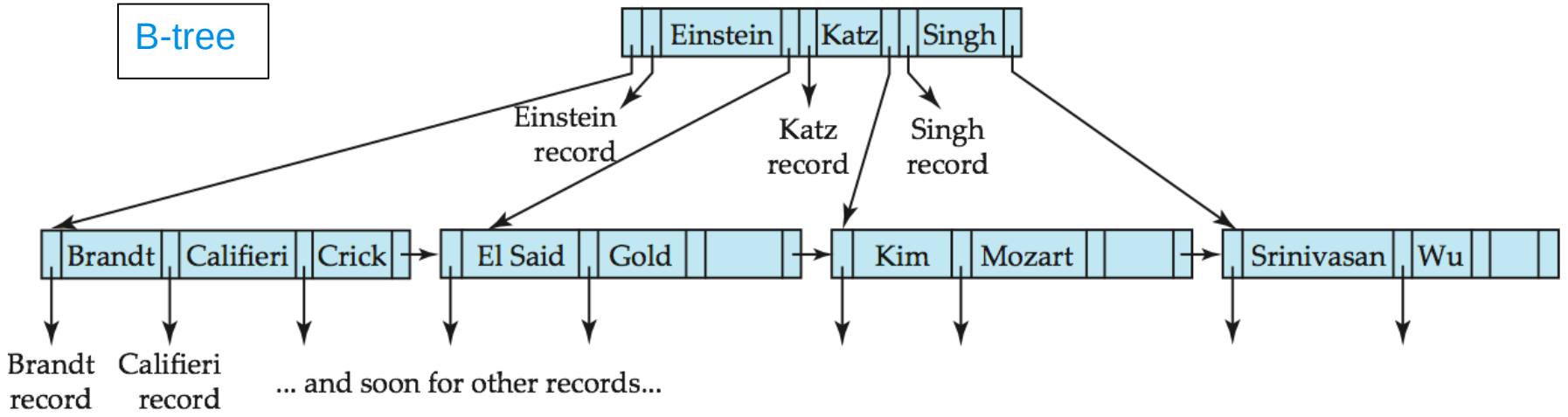
(a)

- Generalized B-tree Nonleaf node
  - pointers Bi are the bucket or file record pointers.

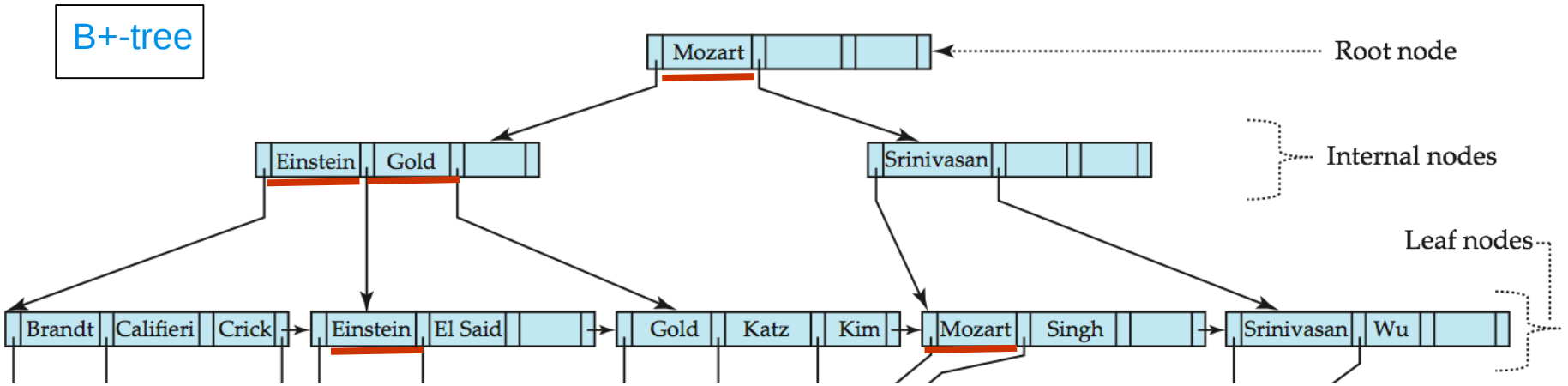


(b)

B-Tree Index File: Example [2/3]



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



B-Tree Index Files [3/3]

- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B^+-Tree
  - Sometimes possible to find search-key value before reaching leaf node
- Disadvantages of B-Tree indices:
  - Only small fraction of all search-key values are found early
  - Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B^+-Tree
  - Insertion and deletion more complicated than in B^+-Trees
  - Implementation is harder than B^+-Trees
- Typically, advantages of B-Trees do not out weigh disadvantages

Chapter 11: Indexing and Hashing

- 11.1 Basic Concepts
- 11.2 Ordered Indices
- 11.3 B^+-Tree Index Files
- 11.4 B^+-Tree Extensions
- 11.5 Multiple-Key Access
- 11.6 Static Hashing
- 11.7 Dynamic Hashing
- 11.8 Comparison of Ordered Indexing and Hashing
- 11.9 Bitmap Indices
- 11.10 Index Definition in SQL

Tree 기반

Hashing 기반

Multiple-Key Access [1/2]

- **Simple Method:** Multiple Single-Key Indices
- Example:

  select *ID*

  from *instructor*

  where *dept_name* = “Finance” **and** *salary* = 80000

- Possible strategies for processing query using indices on single attributes:
  1. Use index on *dept_name* to find instructors with department name Finance; test *salary = 80000*
  2. Use index on *salary* to find instructors with a salary of $80000; test *dept_name* = “Finance”.
  3. Use *dept_name* index to find pointers to all records pertaining to the “Finance” department. Similarly use index on *salary*.
     - Take intersection of both sets of pointers obtained.

Multiple Keys Access [2/2]

- **Better Method: B+ Index on multiple attributes**
- **Composite search keys** are search keys containing more than one attribute
  - E.g. (*dept_name, salary*)
- Lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either
  - $a_1 < b_1$ or $a_1 = b_1$ and $a_2 < b_2$
- Suppose we have an B+ index on combined search-key (*dept_name, salary*) and a query with **where** *dept_name* = "Finance" **and** *salary* = 80000
  - The B+ index on (*dept_name, salary*) can be used to fetch only records that satisfy both conditions
  - Using separate B+ indices in less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions
- Can also efficiently handle

  where *dept_name* = "Finance" **and** *salary* < 80000
- But cannot efficiently handle

  where *dept_name* < "Finance" **and** salary < 80000
  - May fetch many records that satisfy the first but not the second condition

Other Features

- **Covering indices**
  - Add extra attributes to index records at leaf nodes, so (some) queries can avoid fetching the actual records
    - Particularly useful for secondary indices
- **Index records in leaf nodes** have the values of some attributes
  - E.g. Index records using Instructor_ID can have the value of salary attribute

(instructor 327, pointer-to-actual record)

(instructor 432, pointer-to-actual record)

VS.

(instructor 327, $90000, pointer-to-actual record)

(instructor 432, $55000, pointer-to-actual record)

  - Query: Find the salary of “Instructor 327”
    - Quick processing ➔ No need to visit actual data record

Instructor의 salary를 묻는 query가 아주 frequent하다면 의미있는 시도

Chapter 11: Indexing and Hashing

- 11.1 Basic Concepts
- 11.2 Ordered Indices
- 11.3 B$^+$-Tree Index Files
- 11.4 B$^+$-Tree Extensions
- 11.5 Multiple-Key Access
- 11.6 Static Hashing
- 11.7 Dynamic Hashing
- 11.8 Comparison of Ordered Indexing and Hashing
- 11.9 Bitmap Indices
- 11.10 Index Definition in SQL

Tree 기반

Hashing 기반

Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block)
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B*
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record
  - Bucket overflow problem

- In most cases, one disk access is enough for search or update!
  - All you need to do is computation for hashing

Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key

- There are 8 buckets,
- The binary representation of the *i*th character is assumed to be the integer *i*
- The hash function returns the sum of the binary representations of the characters modulo 8
  - E.g. h(Music) = 1 h(History) = 2 h(Physics) = 3 h(Elec. Eng.) = 3

Hashing based on Dept_name

Id	name	Dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash Functions

- Worst hash function maps all search-key values to the same bucket
  - this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**
  - i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**
  - so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned. .

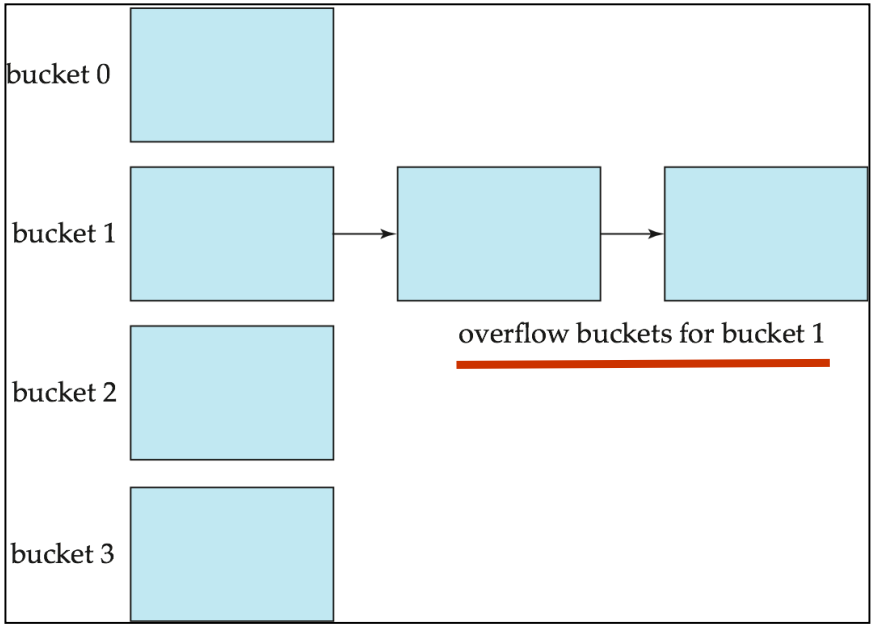
Handling of Bucket Overflows [1/2]

- Collision: Return values of a hash function may collide
  - When more than proper number of records are assigned into a bucket
- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using ***overflow buckets***
- We need Collision Resolution Techniques

Handling of Bucket Overflows [2/2]

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list (This scheme is called **closed hashing)**

- **There are alternative techniques, called open hashing**
  - Do not use overflow buckets, are not suitable for database applications
    - Linear probing
    - Rehashing

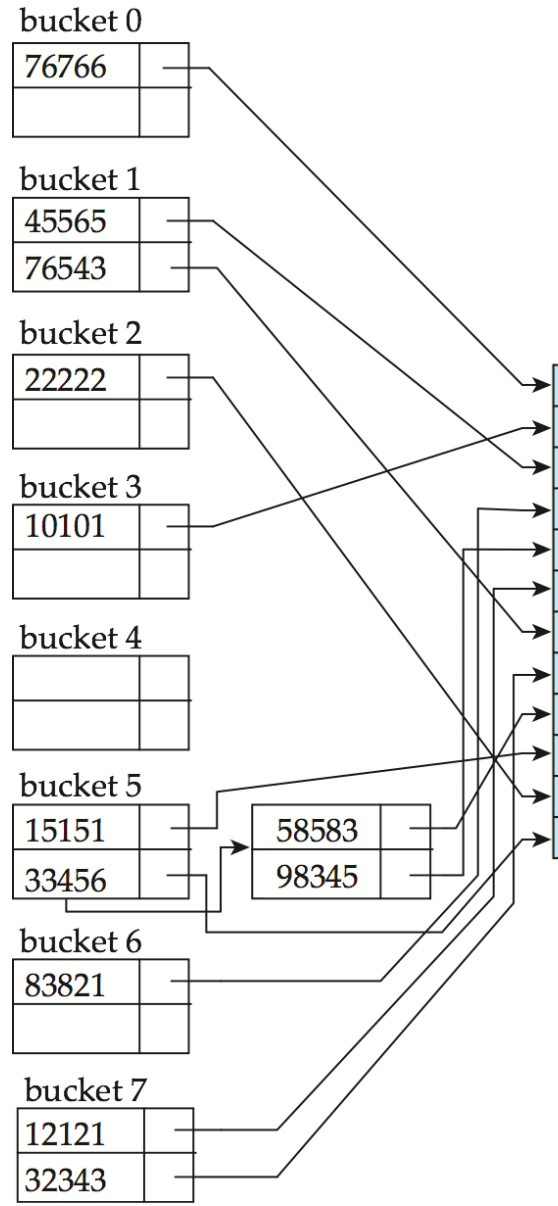


Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure
- Strictly speaking, hash indices are always secondary indices
  - If the file itself is organized using hashing, then a separate primary hash index on it using the same search-key is unnecessary
  - However, we use the term hash index to refer to both secondary index structures and hash organized files

- Hash File Organization 이 아니라 Hash Index (using Hash Table)

Example of Hash Index



Id	name	Dept_name	salary
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
33465	Gold	Physics	87000

The hash index on attribute *ID* of the *instructor* file

Deficiencies of Static Hashing

- Static hashing means "pre-allocation of data buckets"
- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses
  - Databases may grow or shrink with time
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflow buckets
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull)
  - If database shrinks, again space will be wasted
- One solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically

Static Hashing에서는 공간의 utilization도 떨어지고, Update (Insert, Delete)들이 많이 발생하면 Overflow에 의존하므로 성능이 저하되고, 결국 성능이 저하되면 Data Reorganization도 필요!
즉 Dynamic Data Environment에는 바람직하지 않다!!

Chapter 11: Indexing and Hashing

- 11.1 Basic Concepts
- 11.2 Ordered Indices
- 11.3 B$^+$-Tree Index Files
- 11.4 B$^+$-Tree Extensions
- 11.5 Multiple-Key Access
- 11.6 Static Hashing
- 11.7 Dynamic Hashing
- 11.8 Comparison of Ordered Indexing and Hashing
- 11.9 Bitmap Indices
- 11.10 Index Definition in SQL

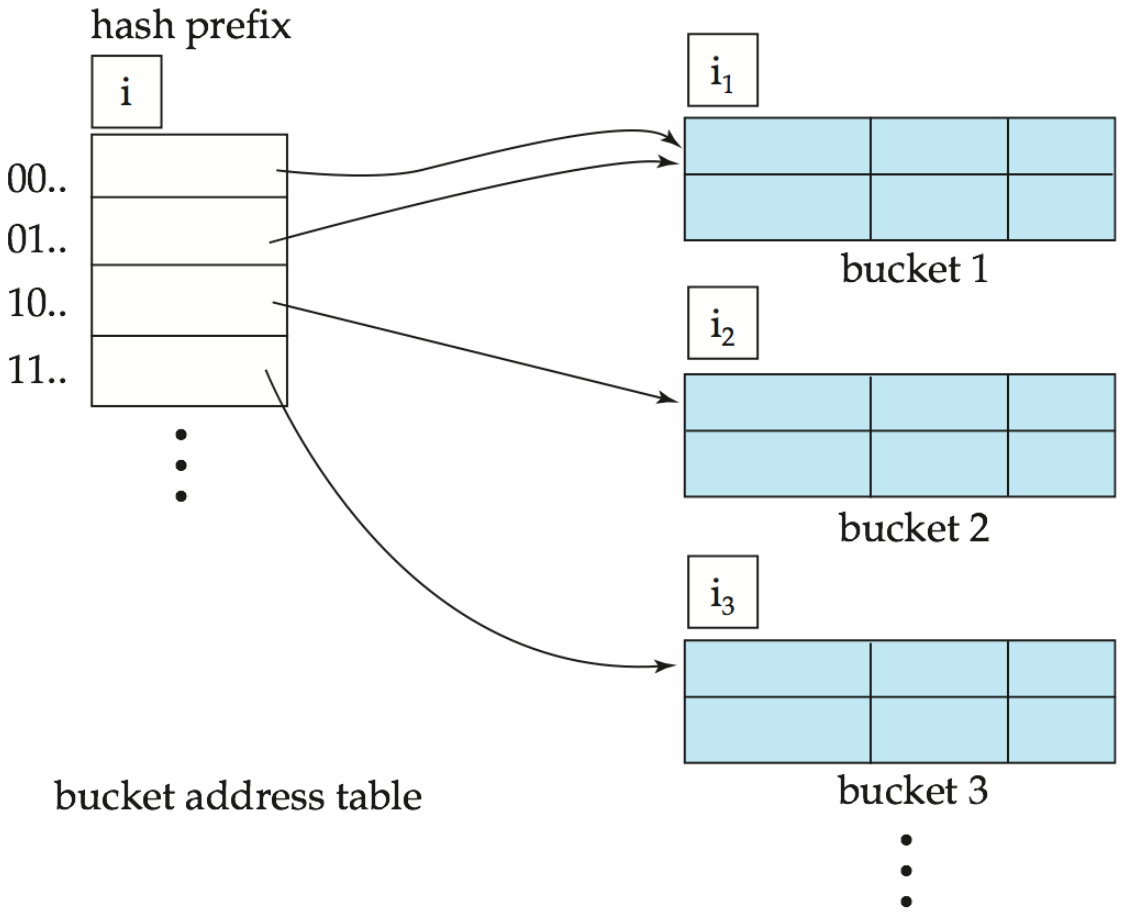
Tree 기반

Hashing 기반

Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically!
- **Extendable hashing** – one form of dynamic hashing
  - 3 components: Bucket address table, Data Bucket, Hash Prefix
  - Hash function generates values over a large range
    - typically *b*-bit integers, with $b = 32$
  - At any time use only a prefix of the hash function to index into a table of bucket addresses
  - Let the length of the prefix be *i* bits, $0 \leq i \leq 32$
    - Bucket address table size = 2^i (Initially $i = 0$)
    - Value of *i* grows and shrinks as the size of the database grows and shrinks
  - Multiple entries in the bucket address table may point to a bucket (why?)
  - Thus, actual number of buckets is $< 2^i$
    - The number of buckets also changes dynamically due to coalescing and splitting of buckets

General Extendable Hash Structure



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$ (see next slide for details)

Use of Extendable Hash Structure

- Each bucket *j* stores a value i_j
  - All the entries that point to the same bucket have the same values on the first i_j bits
- To locate the bucket containing search-key K_j:
  1. Compute $h(K_j) = X$
  2. Use the first *i* high order bits of *X* as a displacement into bucket address table, and follow the pointer to appropriate bucket
- To insert a record with search-key value K_j
  - follow same procedure as look-up and locate the bucket, say *j*
  - If there is room in the bucket *j* insert record in the bucket
  - Else the bucket must be split and insertion re-attempted (next slide)
    - Overflow buckets used instead in some cases (will see shortly)

Insertion in Extendable Hash Structure

To split a bucket j when inserting record with search-key value K_j:

- If $i > i_j$ (more than one pointer to bucket j)
  - allocate a new bucket z, and set $i_j = i_z = (i_j + 1)$
  - Update the second half of the bucket address table entries originally pointing to j, to point to z
  - remove each record in bucket j and reinsert (in j or z)
  - recompute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full)
- If $i = i_j$ (only one pointer to bucket j)
  - If i reaches some limit b, or too many splits have happened in this insertion, create an overflow bucket
  - Else
    - increment i and double the size of the bucket address table
    - replace each entry in the table by two entries that point to the same bucket
    - recompute new bucket address table entry for K_j
      Now $i > i_j$ so use the first case above

Deletion in Extendable Hash Structure

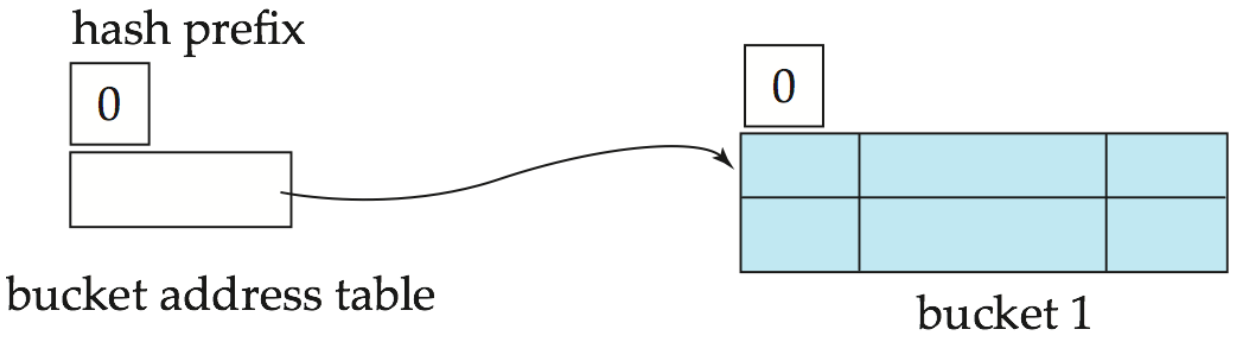
- To delete a key value,
  - locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  - Coalescing of buckets can be done (can coalesce only with a "*buddy*" bucket having same value of i_j and same i_j –1 prefix, if it is present)
  - Decreasing bucket address table size is also possible
    - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

Use of Extendable Hash Structure: Example

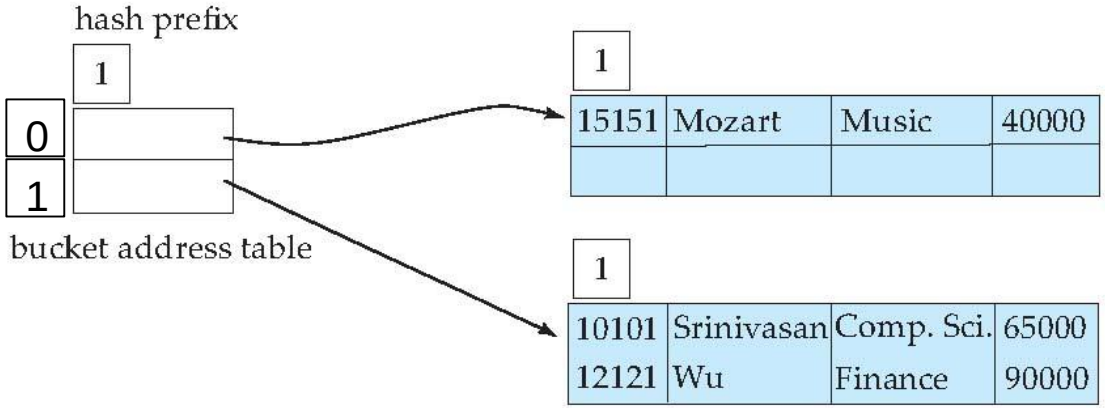
dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

Extendable Hash Example [1/6]

- Initial Hash structure; bucket size = 2



- Hash structure after insertion of "Mozart", "Srinivasan", and "Wu" records



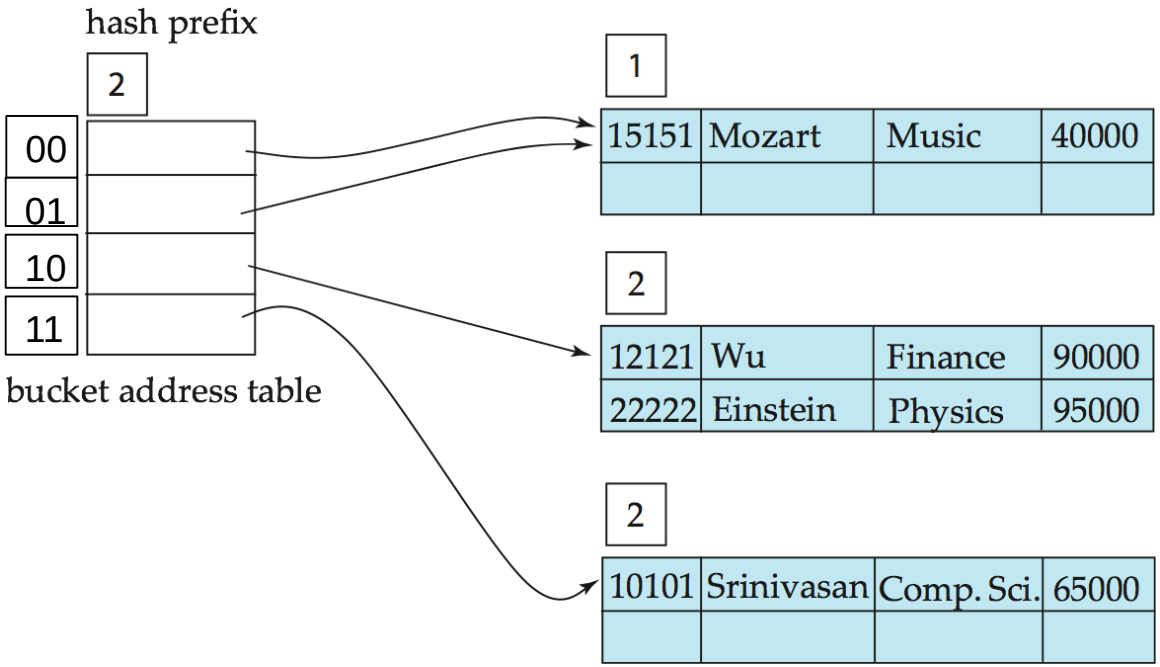
dept_name

Biology	0010
Comp. Sci.	1111
Elec. Eng.	0100
Finance	1010
History	1100
Music	0011
Physics	1001

"Music" 의 hash value의 1st bit
"Comp Sci" 의 hash value의 1st bit
"Finance" 의 hash value의 1st bit

Extendable Hash Example [2/6]

- Hash structure after insertion of Einstein record

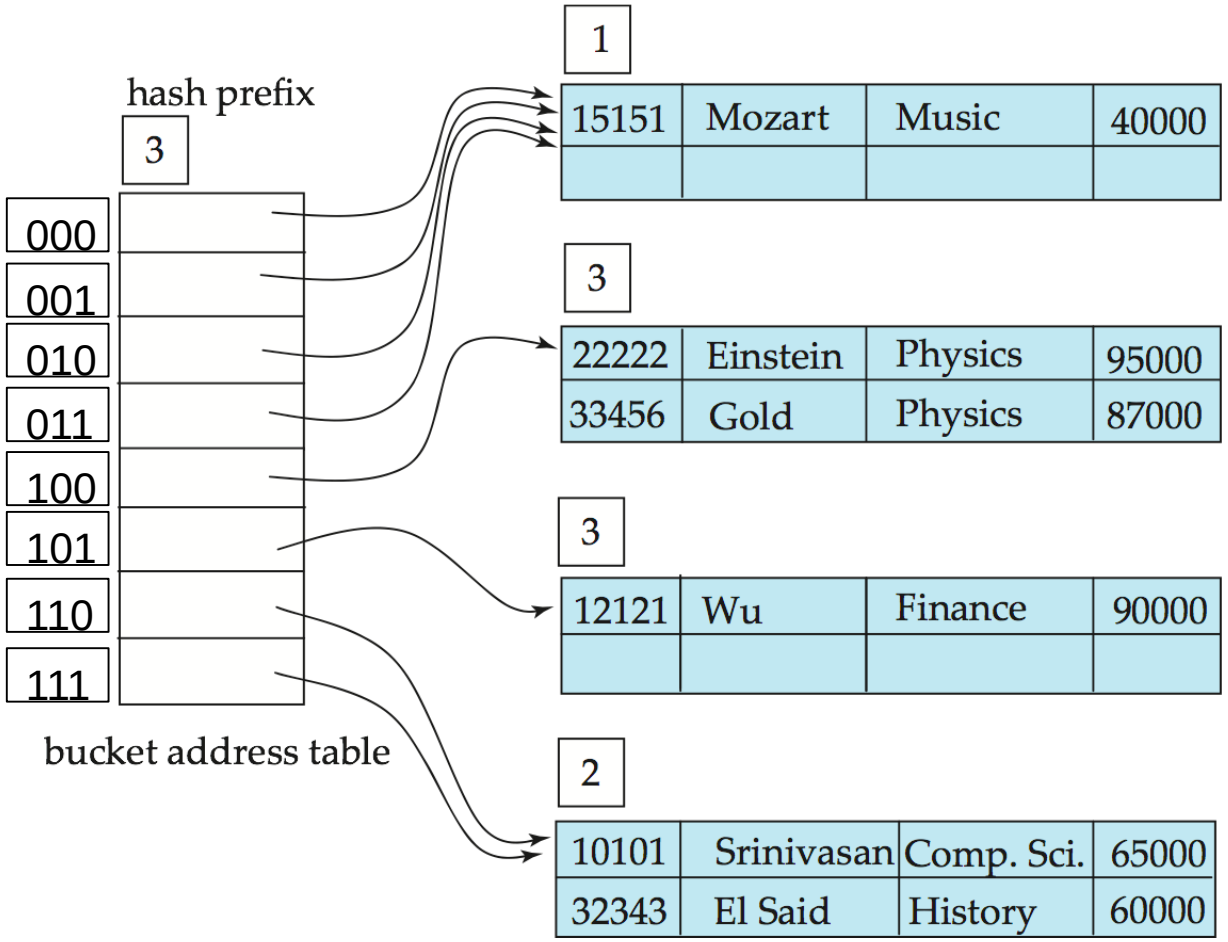


dept_name	
Biology	0010
Comp. Sci.	1111
Elec. Eng.	0100
Finance	1010
History	1100
Music	0011
Physics	1001

"Music" 의 hash value의 1st bit
"Finance" 의 hash value의 1st & 2nd bit
"Einstein" 의 hash value의 1st & 2nd bit
"Srinivasan" 의 hash value의 1st & 2nd bit

Extendable Hash Example [3/6]

- Hash structure after insertion of Gold and El Said records

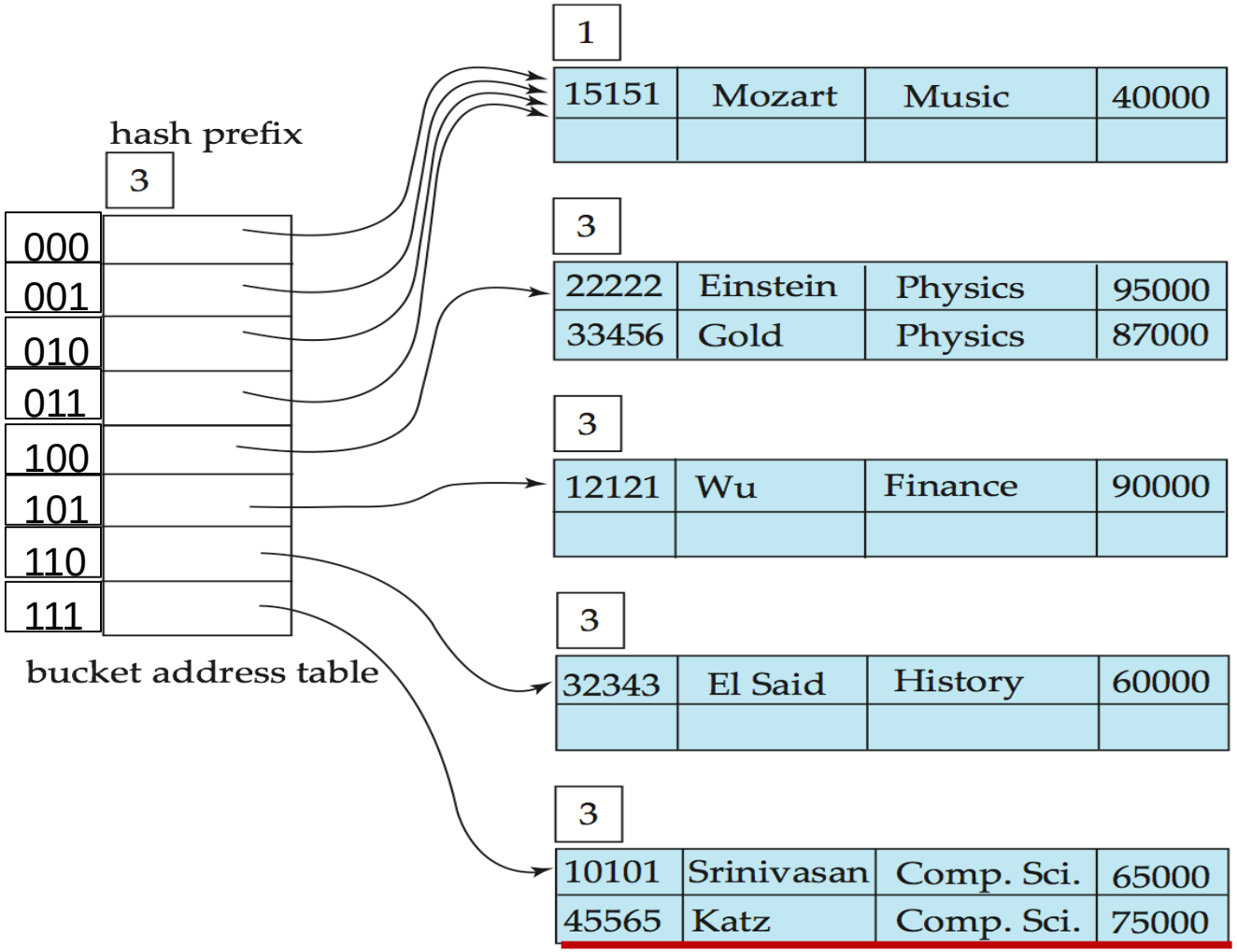


dept_name

Biology	0010
Comp. Sci.	1111
Elec. Eng.	0100
Finance	1010
History	1100
Music	0011
Physics	1001

Extendable Hash Example [4/6]

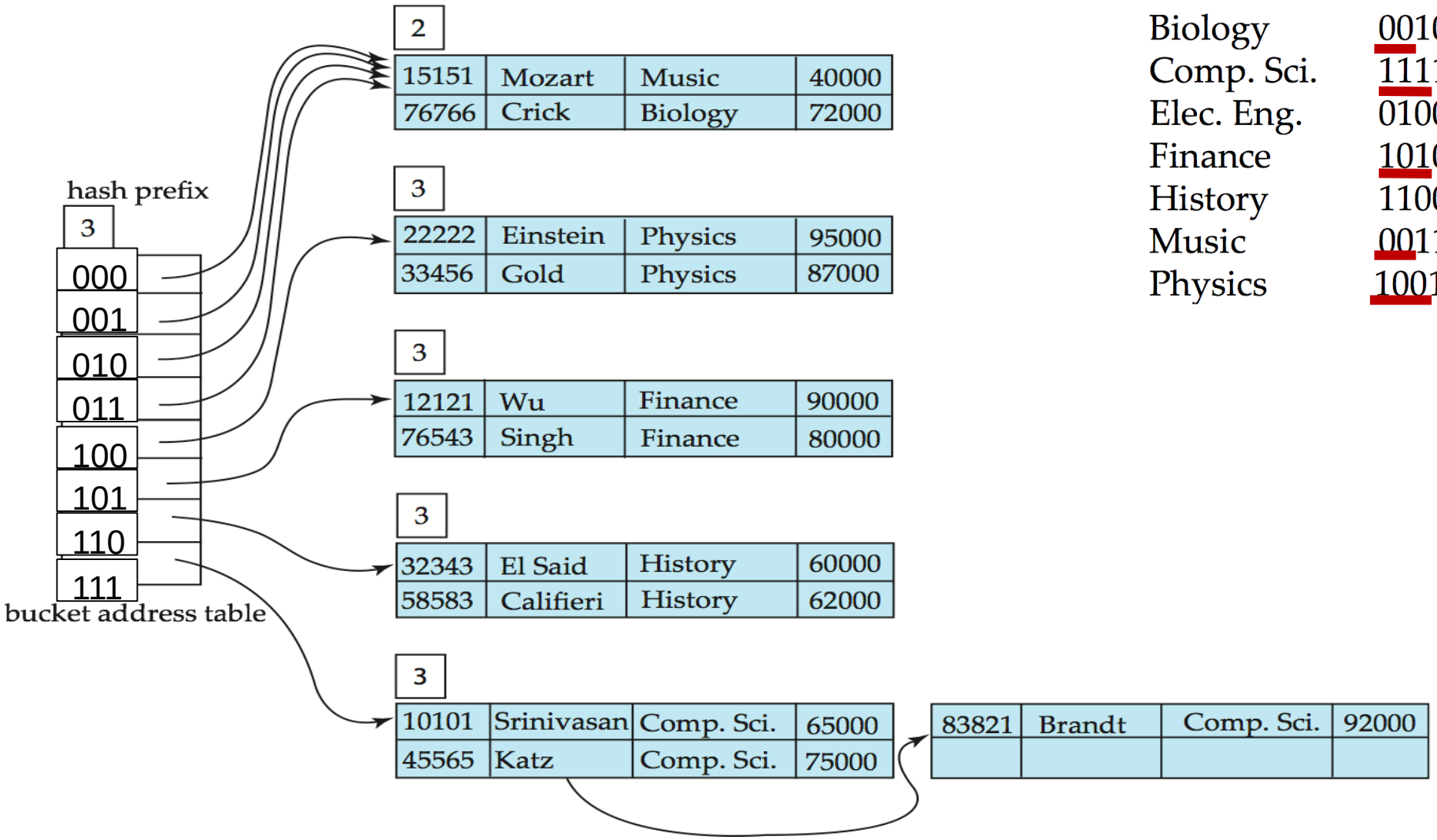
- Hash structure after insertion of Katz record



dept_name	
Biology	0010
Comp. Sci.	1111
Elec. Eng.	0100
Finance	1010
History	1100
Music	0011
Physics	1001

Extendable Hash Example [5/6]

** And after insertion of eleven records



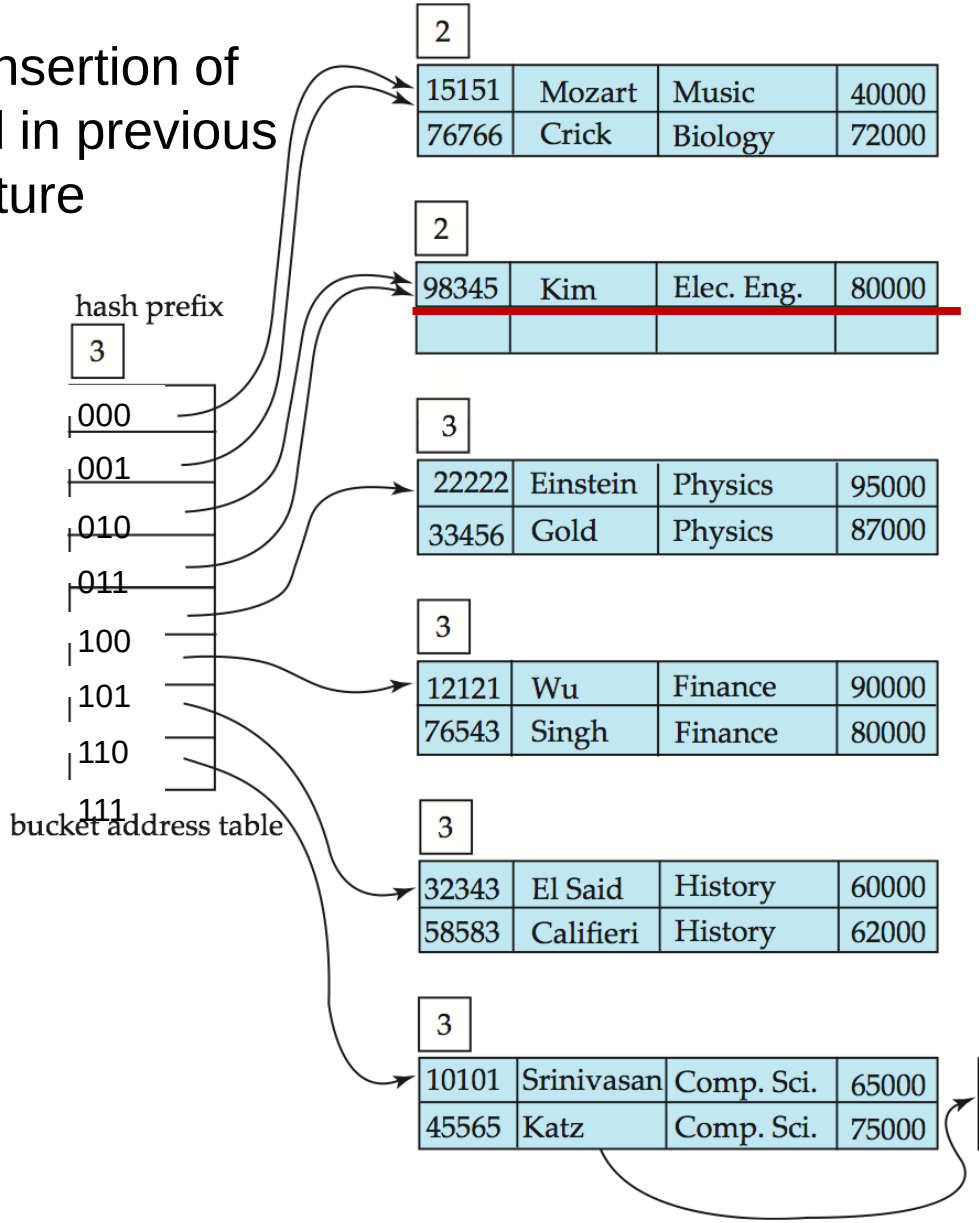
dept_name

Biology	0010
Comp. Sci.	1111
Elec. Eng.	0100
Finance	1010
History	1100
Music	0011
Physics	1001

Comp. Sci.를 가진 records 가 2개이상이면 Overflow를 이용해야!

Extendable Hash Example [6/6]

And after insertion of Kim record in previous hash structure



dept_name

Biology	0010
Comp. Sci.	1111
Elec. Eng.	0100
Finance	1010
History	1100
Music	0011
Physics	1001

Dept_name이 총 7개이면 7개 main data bucket를 만든후에는 overflow bucket을 이용해야!

Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Cannot allocate very large contiguous areas on disk either
    - Solution: B^+-tree structure to locate desired record in bucket address table
  - Changing size of bucket address table is an expensive operation

- Linear hashing is an alternative mechanism
  - Allows incremental growth of its directory (equivalent to bucket address table)
  - At the cost of more bucket overflows

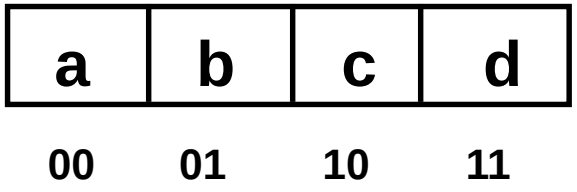
참고자료

Linear Hashing

- The actual address space is extended one bucket at a time as buckets overflow
- Because the extension of the address space does not necessarily correspond to the bucket that is overflowing,

  linear hashing necessarily involves the use of overflow buckets, even as the address space expands
- No directories: Avoid additional seek resulting from additional layer
- Use more bits of hashed value
  - $h_d(k)$: depth *d* hashing function (using function make_address)

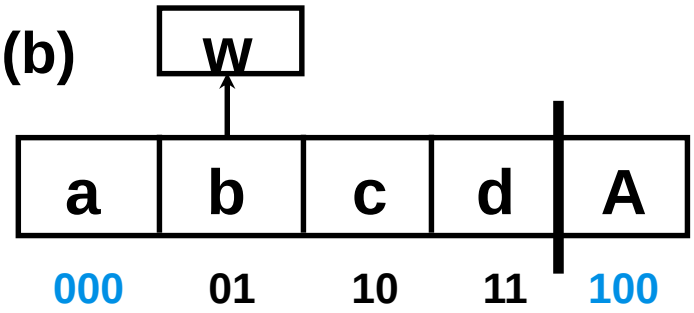
(a)



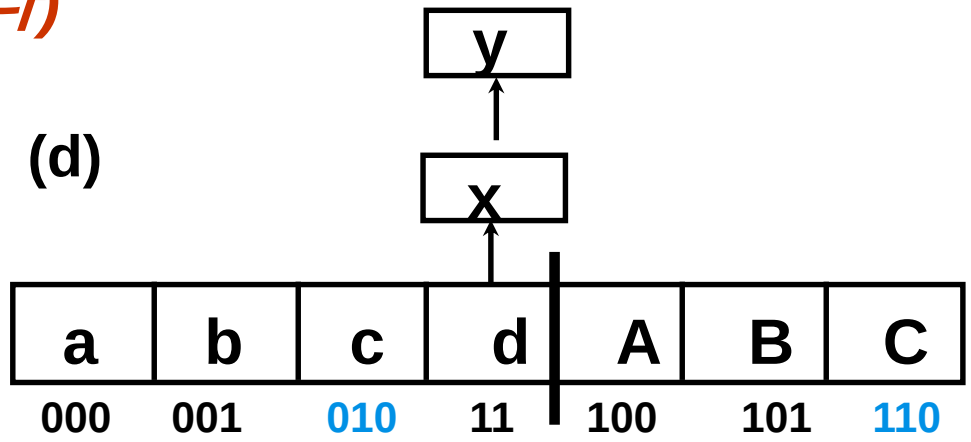
The growth of address space in linear hashing (overflow 발생할때마다)

참고자료

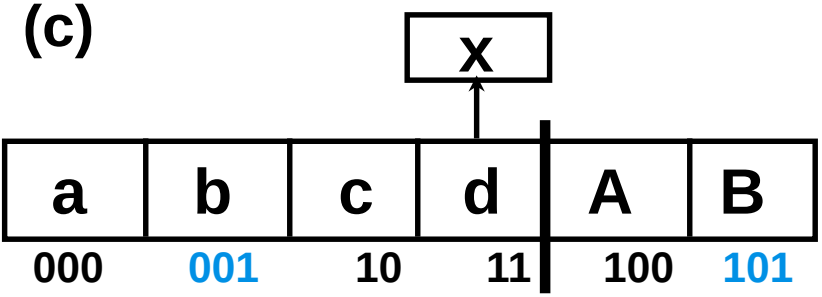
(b)



(d)

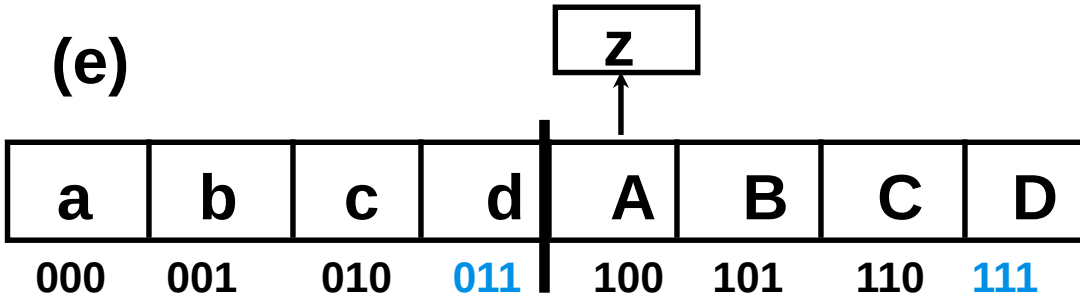


(c)



Now, W in the overflow area is relocated

(e)



Now, X and Y in the overflow area are relocated

Chapter 11: Indexing and Hashing

- 11.1 Basic Concepts
- 11.2 Ordered Indices
- 11.3 B^+-Tree Index Files
- 11.4 B^+-Tree Extensions
- 11.5 Multiple-Key Access
- 11.6 Static Hashing
- 11.7 Dynamic Hashing
- 11.8 Comparison of Ordered Indexing and Hashing
- 11.9 Bitmap Indices
- 11.10 Index Definition in SQL

Tree 기반

Hashing 기반

Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key (point query)
  - If range queries are common, ordered indices are to be preferred
- In practice:
  - Most DBMS supports B+ trees
  - Hashing
    - PostgreSQL supports hash indices, but discourages use due to poor performance
    - Oracle supports static hash organization, but not hash indices
    - SQLServer does not support Hash index

Chapter 11: Indexing and Hashing

- 11.1 Basic Concepts
- 11.2 Ordered Indices
- 11.3 B^+-Tree Index Files
- 11.4 B^+-Tree Extensions
- 11.5 Multiple-Key Access
- 11.6 Static Hashing
- 11.7 Dynamic Hashing
- 11.8 Comparison of Ordered Indexing and Hashing
- 11.9 Bitmap Indices
- 11.10 Index Definition in SQL

Tree 기반

Hashing 기반

Bitmap Indices [1/5]

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
  - Given a number *n*, it must be easy to retrieve record *n*
    - Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
  - E.g. gender, country, state, …
  - E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits

Bitmap Indices [2/5]

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
  - Bitmap has as many bits as records!
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise
  - Bitmap record 1개의 길이는 record갯수와 같다

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Income level이 5개만 있다면

Bitmap Indices [3/5]

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (AND)
  - Union (OR)
  - Complementation (NOT)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - E.g. 100110 AND 110011 = 100010

    100110 OR 110011 = 110111

    NOT 100110 = 011001
  - Males with income level L1
    - bitmap for m AND bitmap for L1 ➔ 10010 AND 10100 = 10000
    - Can then retrieve tuples whose value is 1 in the result bitmap
    - Counting number of matching tuples is even faster

Bitmap Indices [4/5]

- Bitmap indices generally very small compared with relation size
  - The space occupied by a single bitmap is usually less than 1 percent of the space occupied by the relation
  - In the example below, gender attribute has 4 byte data and income_level attribute has 8 byte data
    - The relation spends 12 bytes X 5 records ➔ 60 bytes
    - The bitmap representation spends 5 bits X 7 ➔ 35 bits ➔ 4.3 bytes

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices [5/5]

- *bitmap-A-v: A attribute 가 v값을 가지는 record들의 bitmap*
- *NOT bitmap-A-v: A attribute 가 v값을 가지지 않는 record들의 bitmap*
- **Existence bitmap** to note if there is a valid record at a record location
- 만약 NOT(A = v) "A값이 v가 아닌 record" 를 처리하려면
- Deleted 된 record를 고려하려면
  - *(NOT bitmap-A-v) AND ExistenceBitmap*
- A attribute에 Null값이 들어 있는 record도 무시하려면
  - *(NOT bitmap-A-v) AND ExistenceBitmap* AND *(NOT bitmap-A-Null)*

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

	ExistenceBitmap
초기	11111
After Delete R3	11101

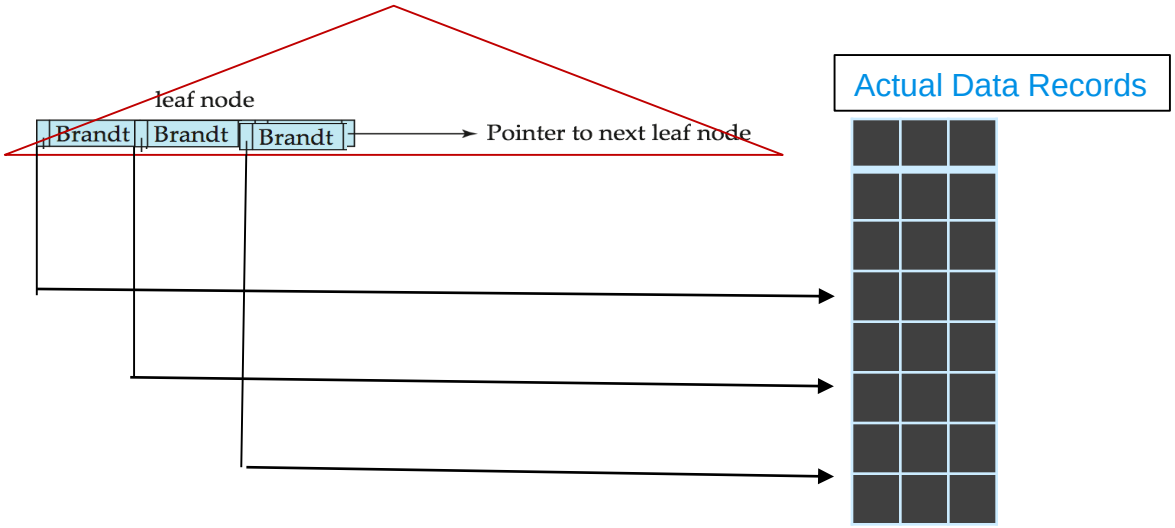
Efficient Implementation of Bitmap Operations

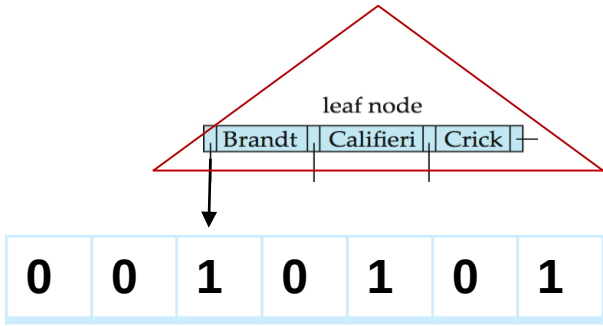
- Bitmaps are packed into words; a "*single word and operation*" (a basic CPU instruction) computes *AND* of 32 (or 64) bits at once
  - E.g. In 32bit computer, 1-million-bit maps can be AND-ed with just 31,250 instruction (1million-bit / 32bit ➔ 31,250)
  - Relation에서 attribute A가 특정값 V을 가지는 record들의 수를 알려고 한다면…
- 모든 record들을 retrieve해서 비교연산 (value(A) == V) 를 수행
- With Bitmap, Just Count number of 1s (can be done fast by a trick)
  - Use each byte (8 bits) to index into a precomputed array of 256 entries where the ith entry stores the count of 1s in the binary representation of i

8 bit 표현	count
00000001	1
00000010	1
00000100	1
……	
11111101	7
11111110	7
11111111	8

Bitmaps and B+-trees

- B+ tree index를 만들었는데, index attribute의 어떤값들이 여러번 반복적으로 발생하는경우에
- Bitmaps can be used instead of Tuple-ID lists at leaf levels of B^+-trees, for values that have a large number of matching records
  - Let N be the number of records in the relation and a record has 64 bit record number
  - Suppose a particular value *Vi* occurs in 1/16 of the records in a relation
    - List representation needs 64 * (N/16) = 4N bits
    - Bitmap needs only N bits (총 N개의 record가 있고, 어떤 attribute가 어떤값을 가지는지를 보여주는 Bitmap)
- Above technique merges benefits of bitmap and B^+-tree indices





List representation

Bitmap representation

Chapter 11: Indexing and Hashing

- 11.1 Basic Concepts
- 11.2 Ordered Indices
- 11.3 B^+-Tree Index Files
- 11.4 B^+-Tree Extensions
- 11.5 Multiple-Key Access
- 11.6 Static Hashing
- 11.7 Dynamic Hashing
- 11.8 Comparison of Ordered Indexing and Hashing
- 11.9 Bitmap Indices
- 11.10 Index Definition in SQL

Tree 기반

Hashing 기반

Index Definition in SQL

- To create an index

  create index <index-name> **on** <relation-name> (<attribute-list>)

  E.g.: **create index** *dept_index* **on** *instructor (dept_name)*

- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported

- To drop an index

  drop index <index-name>

- Most database systems allow specification of type of index, and clustering.

Chapter 11: Indexing and Hashing

- 11.1 Basic Concepts
- 11.2 Ordered Indices
- 11.3 B^+-Tree Index Files
- 11.4 B^+-Tree Extensions
- 11.5 Multiple-Key Access
- 11.6 Static Hashing
- 11.7 Dynamic Hashing
- 11.8 Comparison of Ordered Indexing and Hashing
- 11.9 Bitmap Indices
- 11.10 Index Definition in SQL
- 참고자료: Multi-Dimensional Index

Tree 기반

Hashing 기반

참고자료

Multi-Dimensional Indexing and Hashing

- **Best but Complex Method!**
- **Multi-Dimensional Data: Spatial Data, Geographic Data, Temporal Data**
- **R-tree : ~~ B+ tree for rectangles with (x1, y1, x2, y2)**
- **Grid File : ~~ Multi-Dimensional Extendible Hashing**
- **Chap 25: Spatial and Temporal Data and Mobility**

참고자료

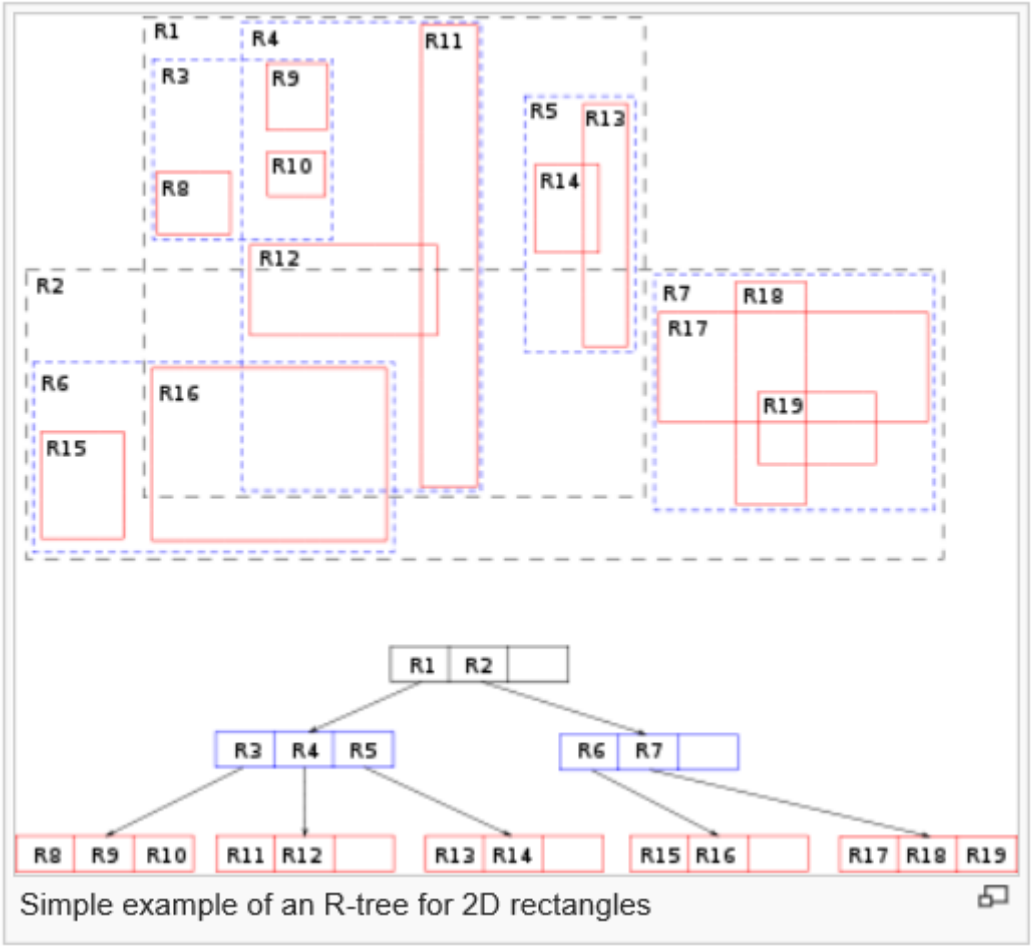
R-Tree [1/2]

R-tree

Type	Tree
Invented	1984
Invented by	Antonin Guttman

Time complexity in big O notation

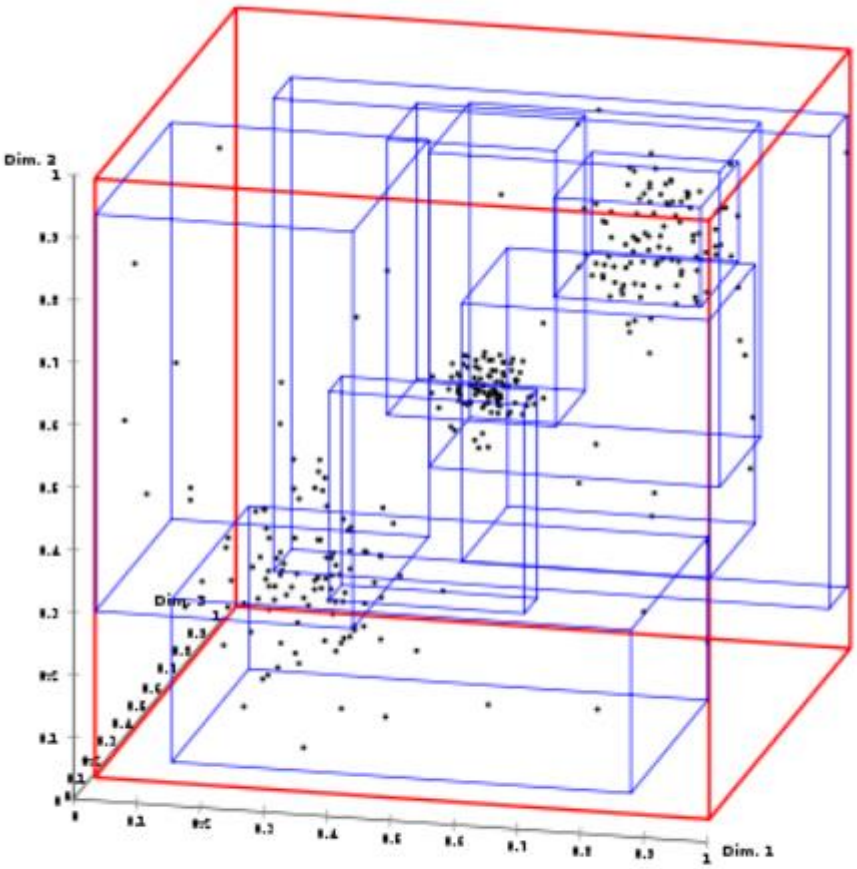
	Average	Worst case
Space		
Search	$O(log_M n)$	
Insert		$O(n)$
Delete		



Simple example of an R-tree for 2D rectangles

참고자료

R-Tree [2/2]



Visualization of an R*-tree for 3D points using ELKI (the cubes are directory pages)

참고자료

Grid File [1/4]

- Structure used to speed the processing of general multiple search-key queries involving one or more comparison operators.
- The grid file has a single grid array and one linear scale for each search-key attribute. The grid array has number of dimensions equal to number of search-key attributes.
- Multiple cells of grid array can point to same bucket
- To find the bucket for a search-key value, locate the row and column of its cell using the linear scales and follow pointer

The Grid File: An Adaptable, Symmetric Multikey File Structure

J. NIEVERGELT, H. HINTERBERGER
Institut für Informatik, ETH
AND K. C. SEVCIK
University of Toronto

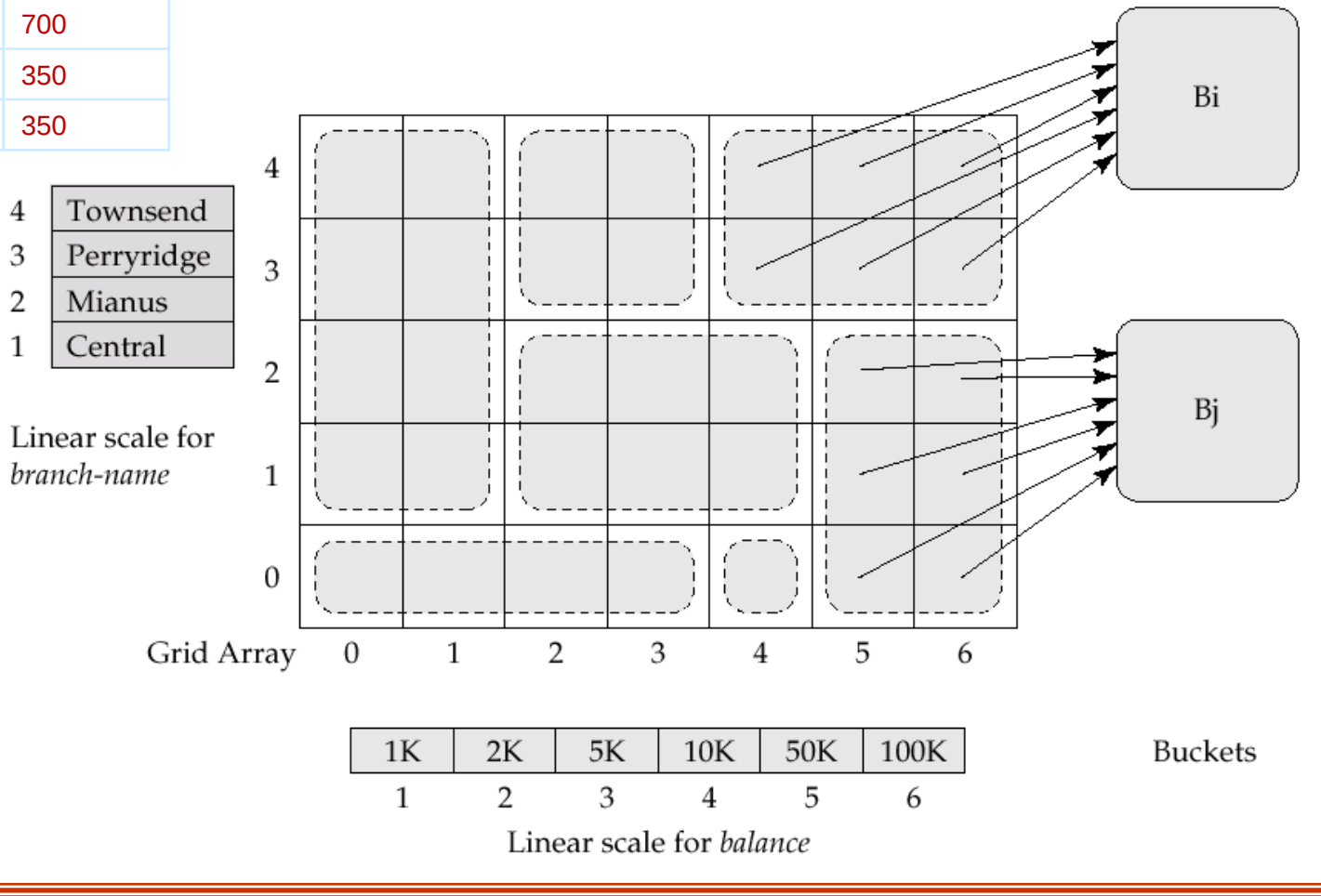
ACM Transactions on Database Systems, Vol. 9, No. 1, March 1984, Pages 38–71.

Grid File [2/4]

Example for *Account*

참고자료

account_num	branch_name	balance
A-101	Downtown	500
A-110	Downtown	600
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-305	Round Hill	350
A-215	Mianus	350



Grid File: Query [3/4]

- A grid file on two attributes *A* and *B* can handle queries of all following forms with reasonable efficiency
  - $(a_1 \leq A \leq a_2)$
  - $(b_1 \leq B \leq b_2)$
  - $(a_1 \leq A \leq a_2) \wedge (b_1 \leq B \leq b_2)$

- E.g., To answer $((a_1 \leq A \leq a_2) \wedge (b_1 \leq B \leq b_2))$, use linear scales to find corresponding candidate grid array cells, and look up all the buckets pointed to from those cells.

참고자료

Grid File: Update [4/4]

- During insertion, if a bucket becomes full, new bucket can be created if more than one cell points to it
  - Idea similar to extendable hashing, but on multiple dimensions
  - If only one cell points to it, either an overflow bucket must be created or the grid size must be increased
- Linear scales must be chosen to uniformly distribute records across cells.
  - Otherwise there will be too many overflow buckets
- Periodic re-organization to increase grid size will help
  - But reorganization can be very expensive
- Space overhead of grid array can be high