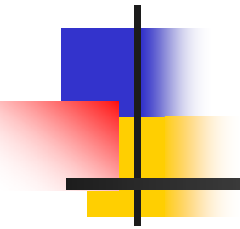
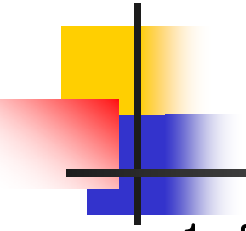


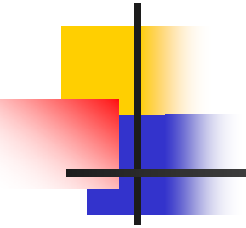
File Management & Physical Database Implementation

Ki-Joon Han
School of Computer Science & Engineering
Konkuk University
kjhan@db.konkuk.ac.kr



Contents

1. File Organization
2. Record Representation
3. Indexing and B^+-Tree
4. REQUIEM
   - File Organization
   - Implementation of B^+-tree Structure
   - Cache Buffering Scheme
5. Access Path Implementation
   - B^+-tree Management
   - File Management
   - Access Path Implementation



File Organization

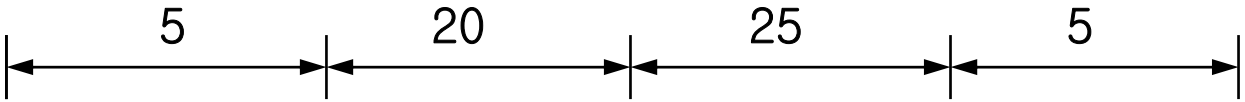
- File Organization
  - the way in which records are stored in a data file
  - the data structures used for organizing data

- Access Types
  - Depend on the organization of the file and the storage media
  - Sequential
  - Random

Record Representation (1/4)

- Search through a fixed length file, deleting all records satisfying some easily checked criterion

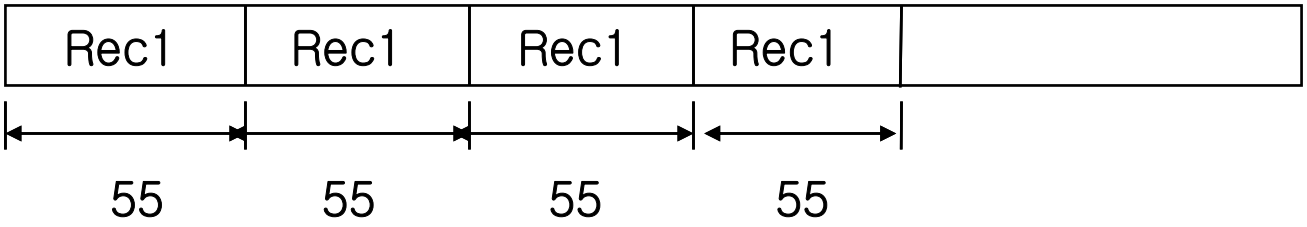
Supplier Data File



	SUP_NO	SUP_NAME	SUP_ADDRESS	PROD_NO
Rec1	125	Jones	London	21
Rec2	128	Black	NewYork	25
Rec3	135	Gray	Paris	17
Rec4	127	Smith	Zurich	12

Record Representation (2/4)

- Normal approach



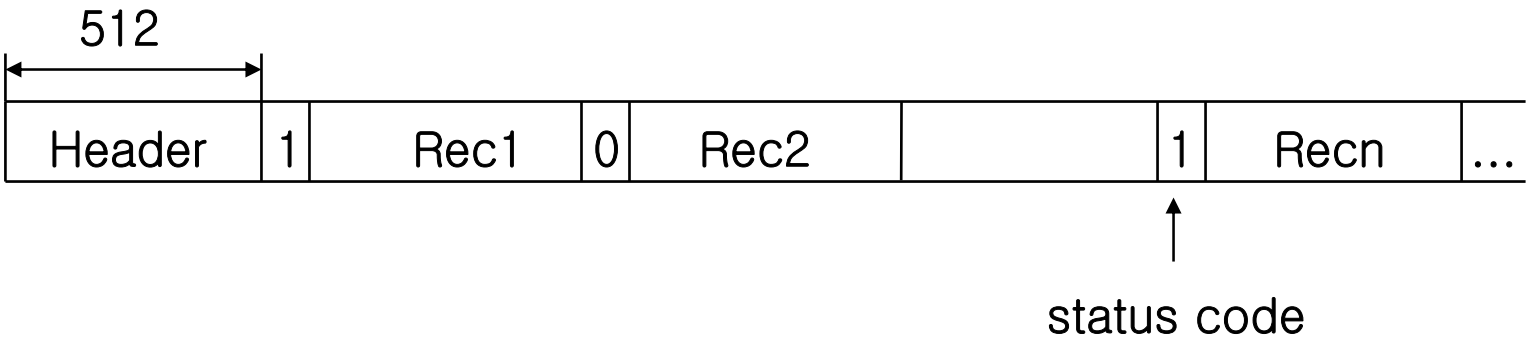
  - Difficult to delete a record
  - Record compaction : time consuming
  - Simple delete mark on a deleted record
    - searching procedure to locate the available space

Record Representation (3/4)

- File header (File Control block)
  - Include information about the nature of the file (record length , address of first data record, # of records,…)
  - When a file is opened, the header information is brought into memory
  - File headers have a fixed size (ex. 512 bytes)
  - deleted records have a DELETE character as their first character (use status character)

Record Representation (4/4)

- If you know the size of the data file header
  - open the data file
  - call LSEEK() to get past the data file header
  - call the C function READ() to read each record, using the size of the record
  - check table status character



Indexing (1/2)

- logical reorganization of records of a file in terms of certain field values
- fast direct access to a specific stored record on the basis of a given value (ex. primary key)

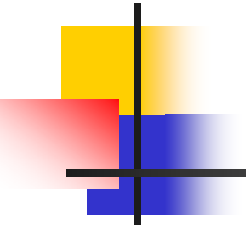
Index File IX

SUP_NO	REC_ADDRESS
S125	622
S128	677
S135	512
S137	567
S142	732

Data File Supplier

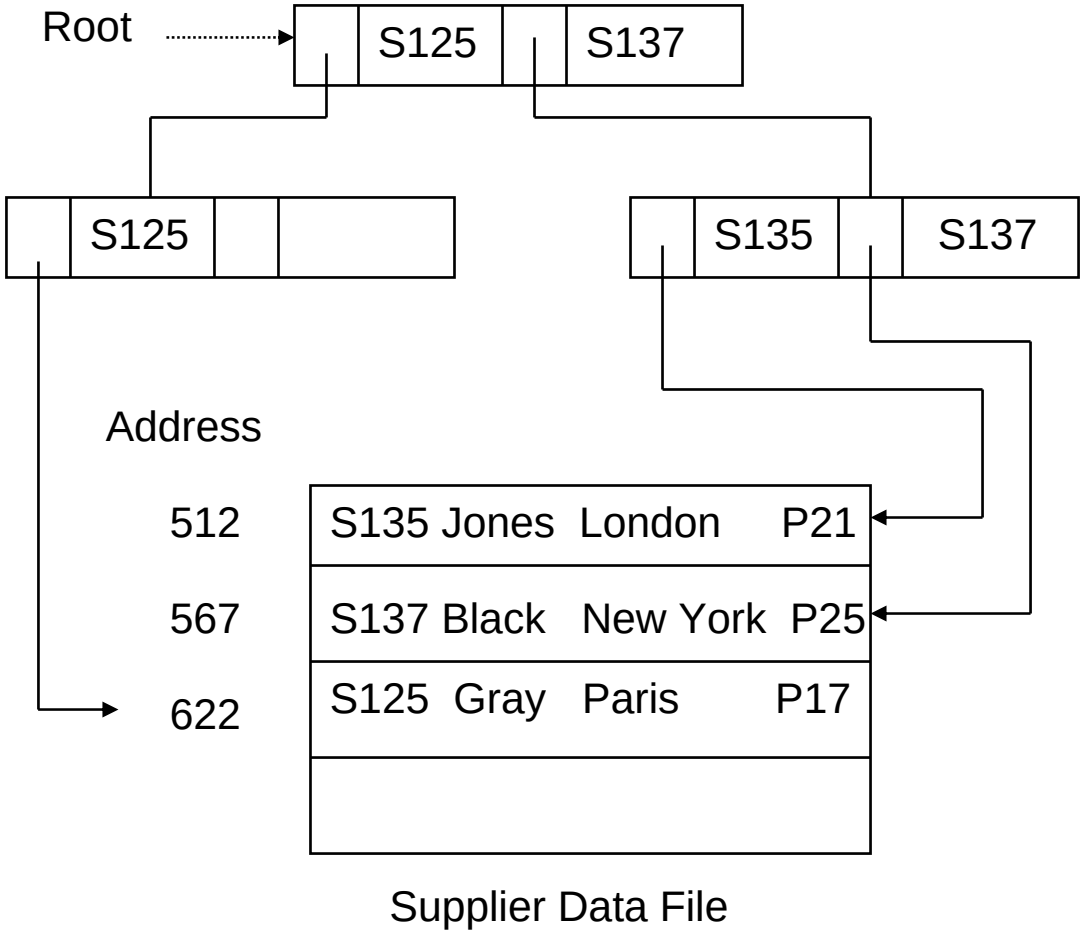
	SUP_NO	SUP_NAME	SUP_ADDRESS	PROD_NO
512	S135	JONES	LONDON	P21
567	S137	BLACK	NEW YORK	P25
622	S125	GRAY	PARIS	P17
677	S128	SMITH	ZURICH	P12
732	S142	ROGERS	SYNDEY	P21

Indexing (2/2)



- The index file is much smaller than the data file
  - faster than a sequential scan of the data file
- Indices provide both direct and sequential access to the indexed data

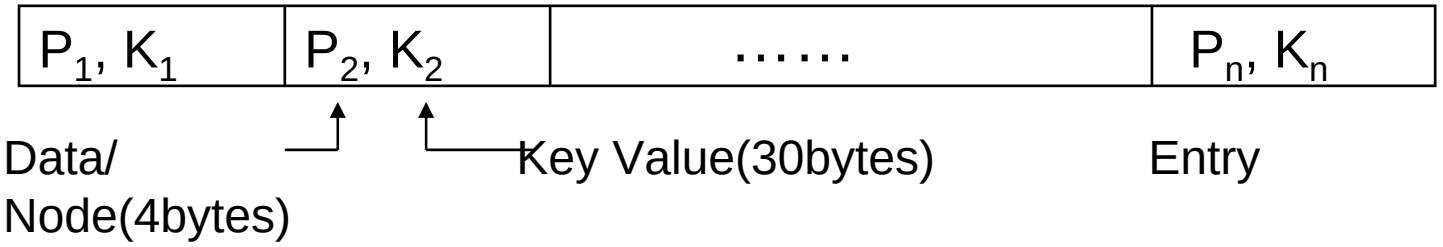
B+–tree (1/2)



[Diagram of B+ -tree of order 2]

B+-tree (2/2)

- Typical node of the B+-tree(1)



- Typical node of the B+-tree(2)

Ki : the search-key values ($K1 < K2 < K3 < \ldots < Kn-1$)
Pi : pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes, Pn : sibling node pointer).

B^+-tree Indexed File (1/2)

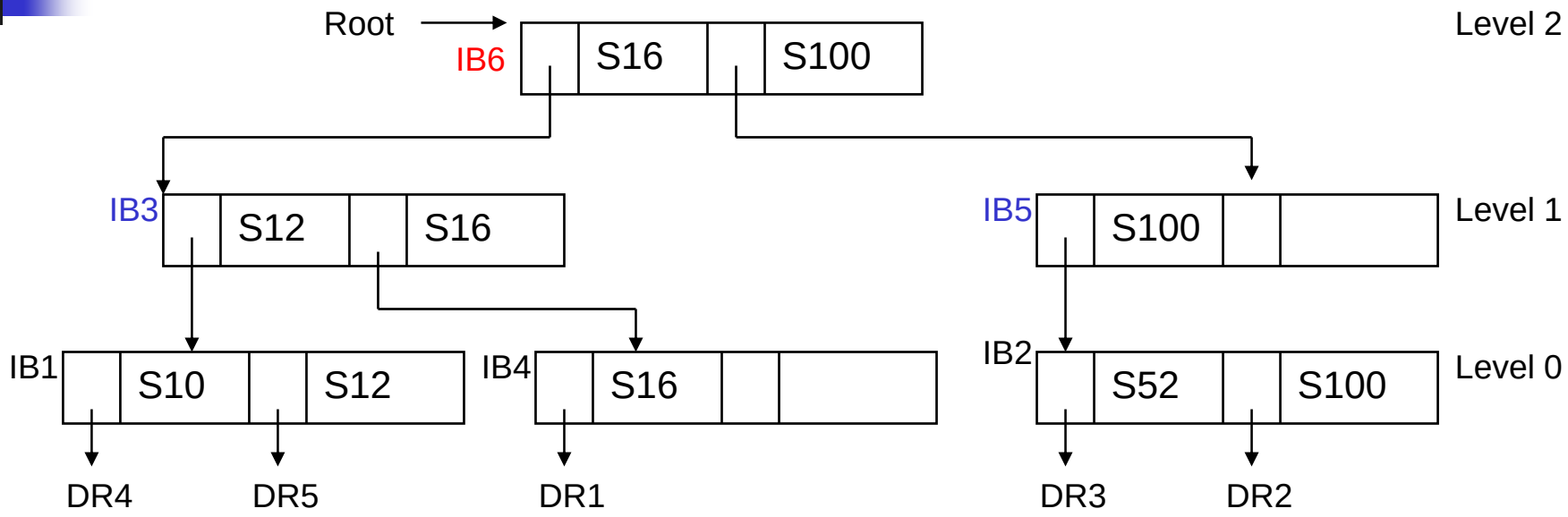
- Disadvantages of the indexed file organization
  - performance is degraded as the file grows
- The most popular indexing methods
  - the B^+-tree indexing organization
- B^+-tree structure
  - All keys and associated data pointers reside in the leaf nodes and are accessed via a multilevel index
  - Branch nodes contain key values and pointers to lower levels
  - Leaf nodes contain addresses for records in data file, and all keys appear in the leaf nodes

B^+-tree Indexed File (2/2)

- In B^+-tree, more than one index key can be built for the same file, if more than one key exists for a given record

  => multivalued index B^+-tree

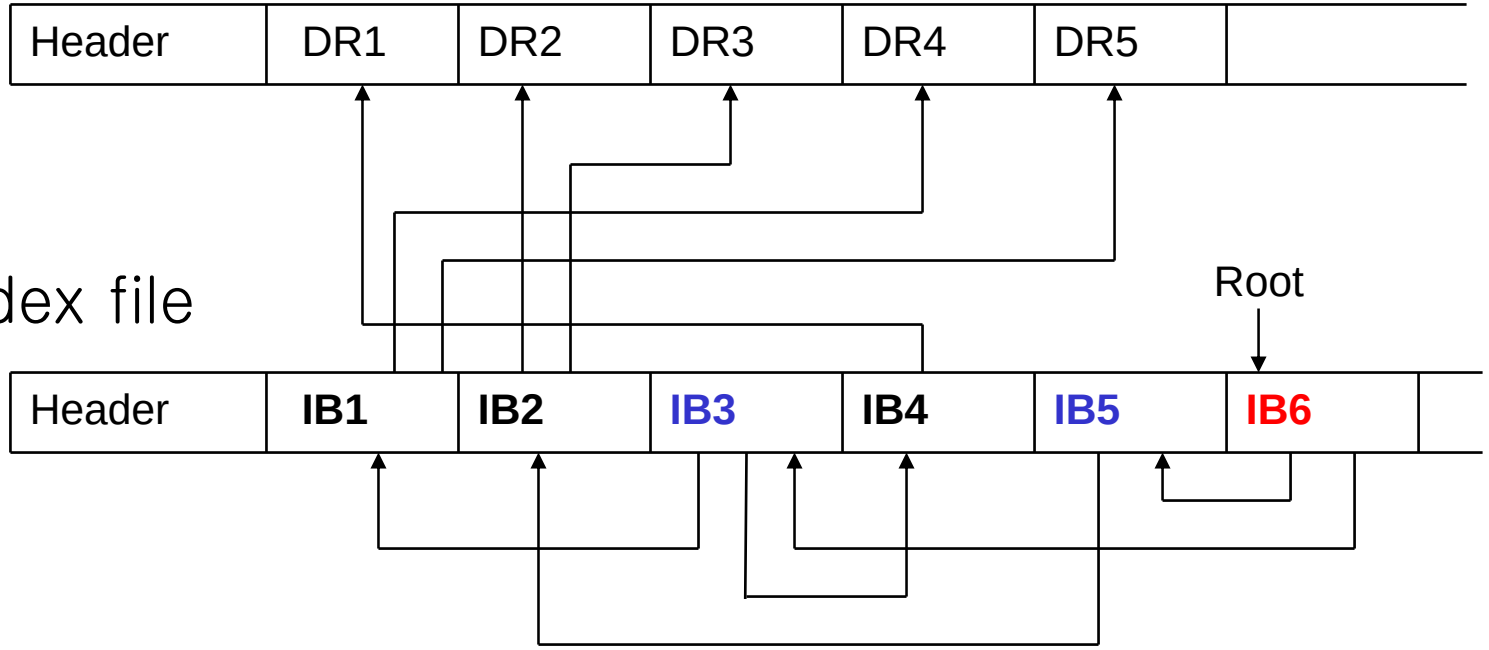
Diagram of Index Block and Data Record Structure (1/2)



- Dri : data records/blocks
  i denotes the physical location of data records/blocks
- IBj : index blocks
  j indicates the physical location of the index blocks in the index file

Diagram of Index Block and Data Record Structure (2/2)

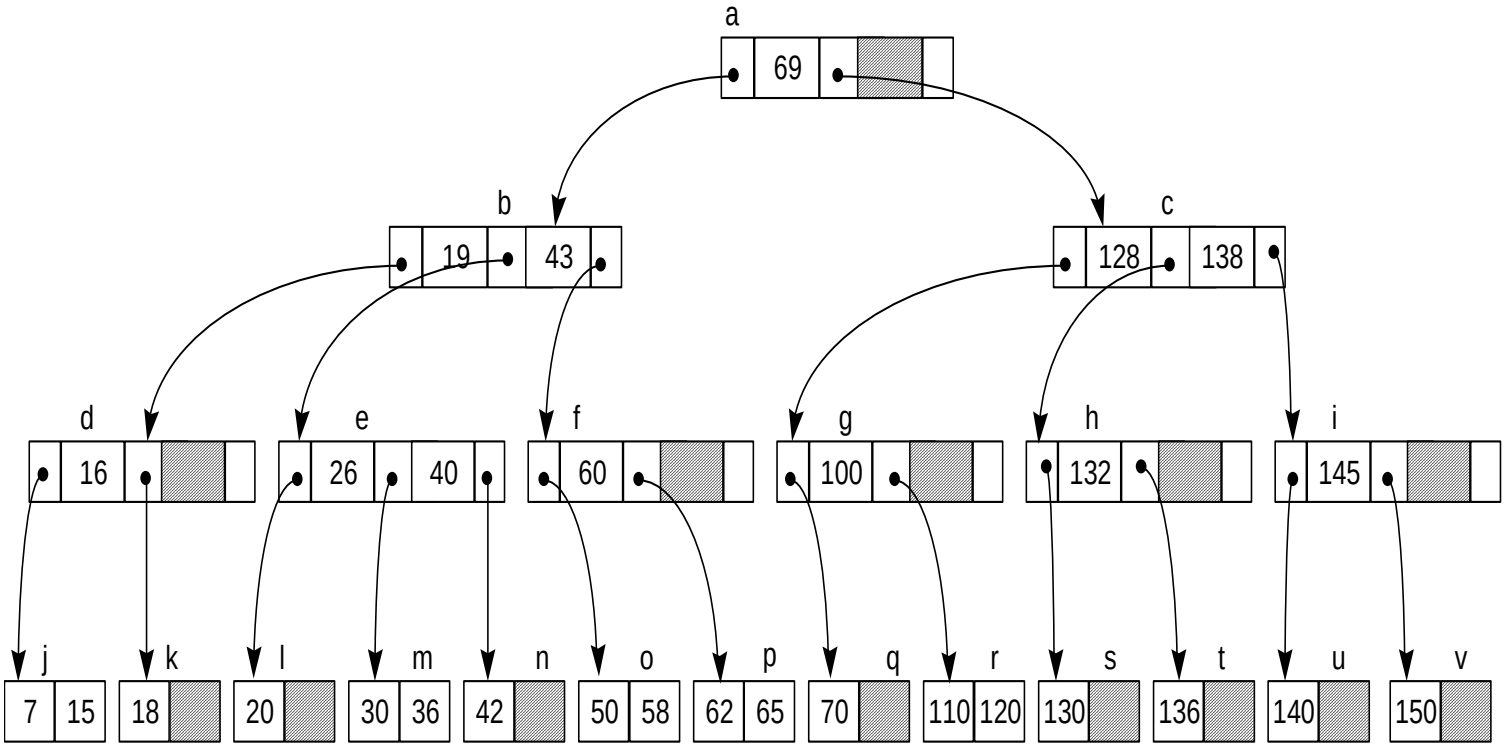
- Data file
- Index file



B-tree (1/2)

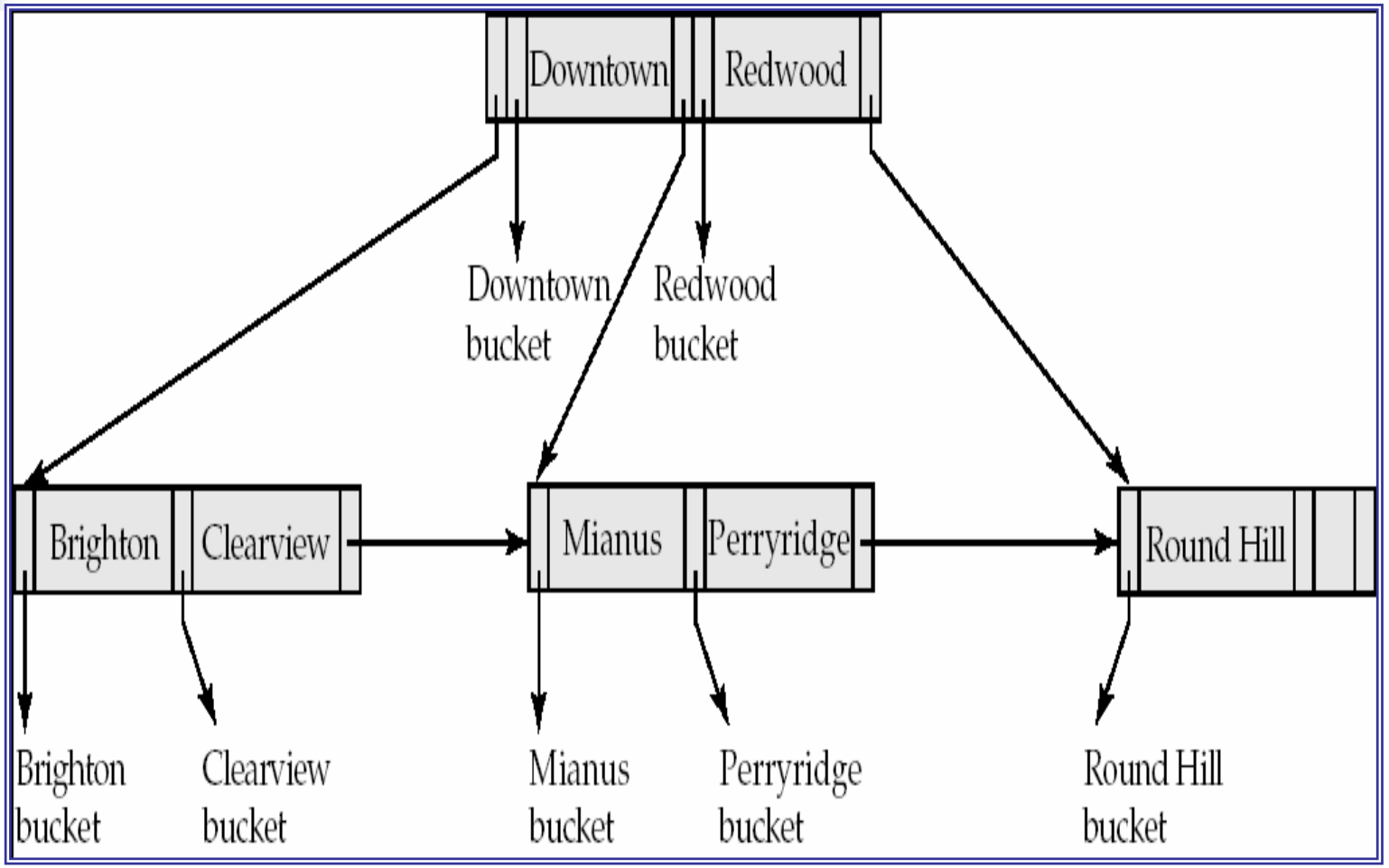
- B-트리는 인덱스를 구성하는 방법으로 많이 사용되는 균형된 m-원 검색 트리이다.
- 키값과 레코드를 가리키는 포인터들이 트리의 노드에 오름차순에 따라 차례대로 저장된다.
- B-트리에서의 순차 검색은 중위순회 방식(inorder traversal)으로 가능하다.
- 키의 삽입과 삭제시 노드의 분열과 합병이 발생할 수 있다.
- 차수 m인 B-트리의 특징
  - 모든 노드는 최대 m개의 서브트리를 가진다.
  - 루트(Root) 노드와 단말 노드를 제외한 모든 노드는 최소 $\lceil m/2 \rceil$개, 최대 m개의 서브트리를 가진다.
  - 루트 노드는 단말 노드가 아닌 이상 적어도 두 개의 서브트리를 가진다.
  - 리프가 아닌 노드에 있는 키 값의 수는 그 노드의 서브트리 수보다 하나 작다.
  - 모든 단말 노드는 같은 레벨(Level)에 있다.
  - 한 노드안에 있는 키 값들은 오름차순을 유지한다.

B-tree (2/3)



<차수가 3인 B-트리 예>

B-tree (3/3)


Downtown
Redwood
Downtown bucket
Redwood bucket
Brighton
Clearview
Mianus
Perryridge
Round Hill
Brighton bucket
Clearview bucket
Mianus bucket
Perryridge bucket
Round Hill bucket

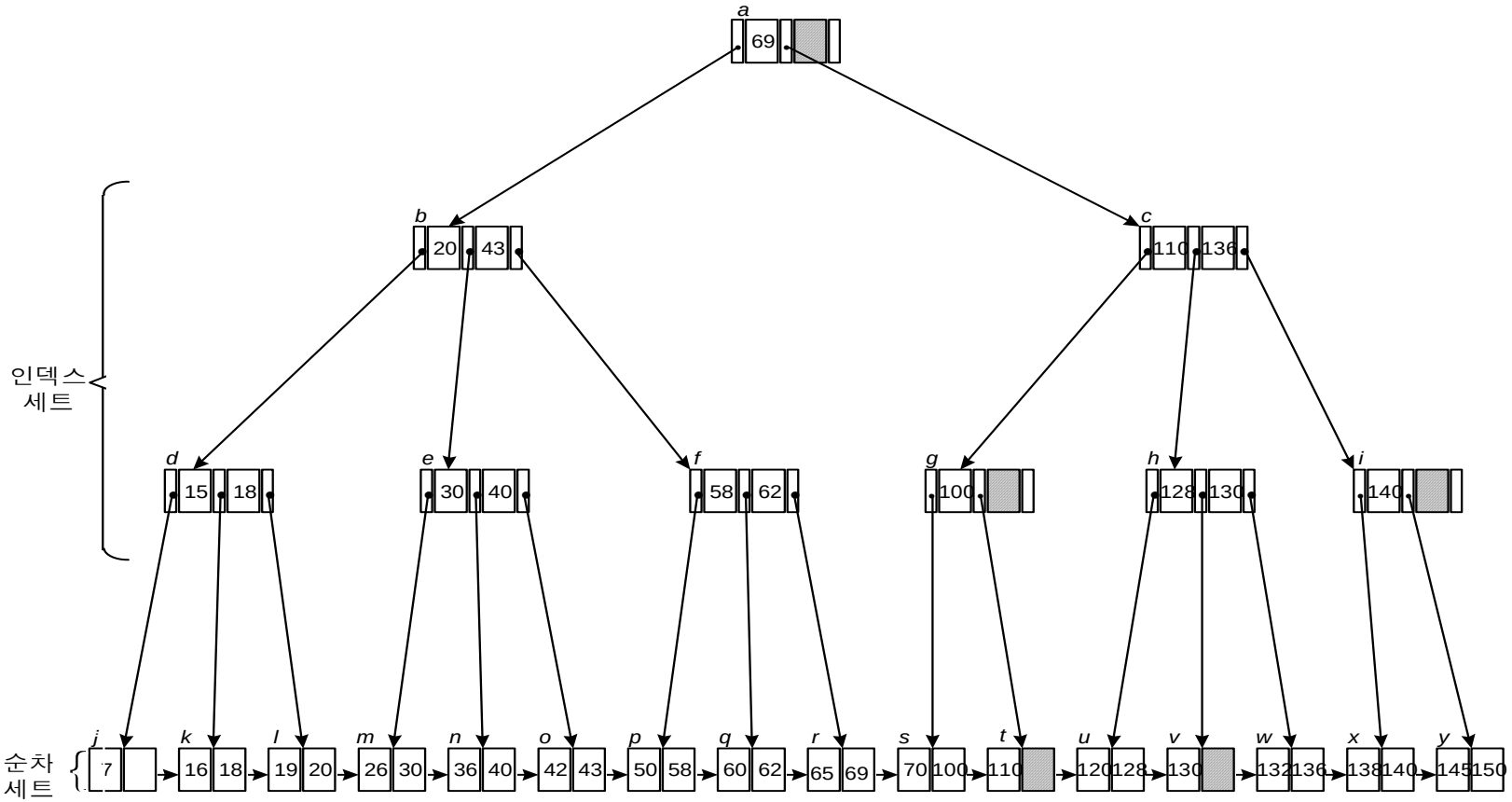
B*-tree

- B*-트리는 B-트리의 문제점인 빈번한 노드의 분할을 줄이는 목적으로 제시된 B-트리의 변형이다.
- B*-트리에서는 각 노드가 가능한 최소한 2/3가 채워지도록 한 것으로 특징이다.
- B*-트리에서의 순차 검색은 중위순회 방식(inorder traversal)으로 가능하다.
- 차수 m인 B*-트리의 특징
  - 루트 노드를 제외한 모든 노드는 적어도 ⌈(2m-2)/3⌉개, 최대 m개의 서브트리를 가진다.
  - 루트 노드는 그 자체가 단말이 아닌 경우 적어도 2개의 서브트리를 갖는다.
  - 리프가 아닌 노드에 있는 키 값의 수는 그 노드의 서브트리 수보다 하나 작다.
  - 모든 단말 노드는 같은 레벨에 있다. 즉, 루트로부터 같은 거리에 있다.
  - 한 노드안에 있는 키 값들은 오름차순을 유지한다.

B^+-tree (1/3)

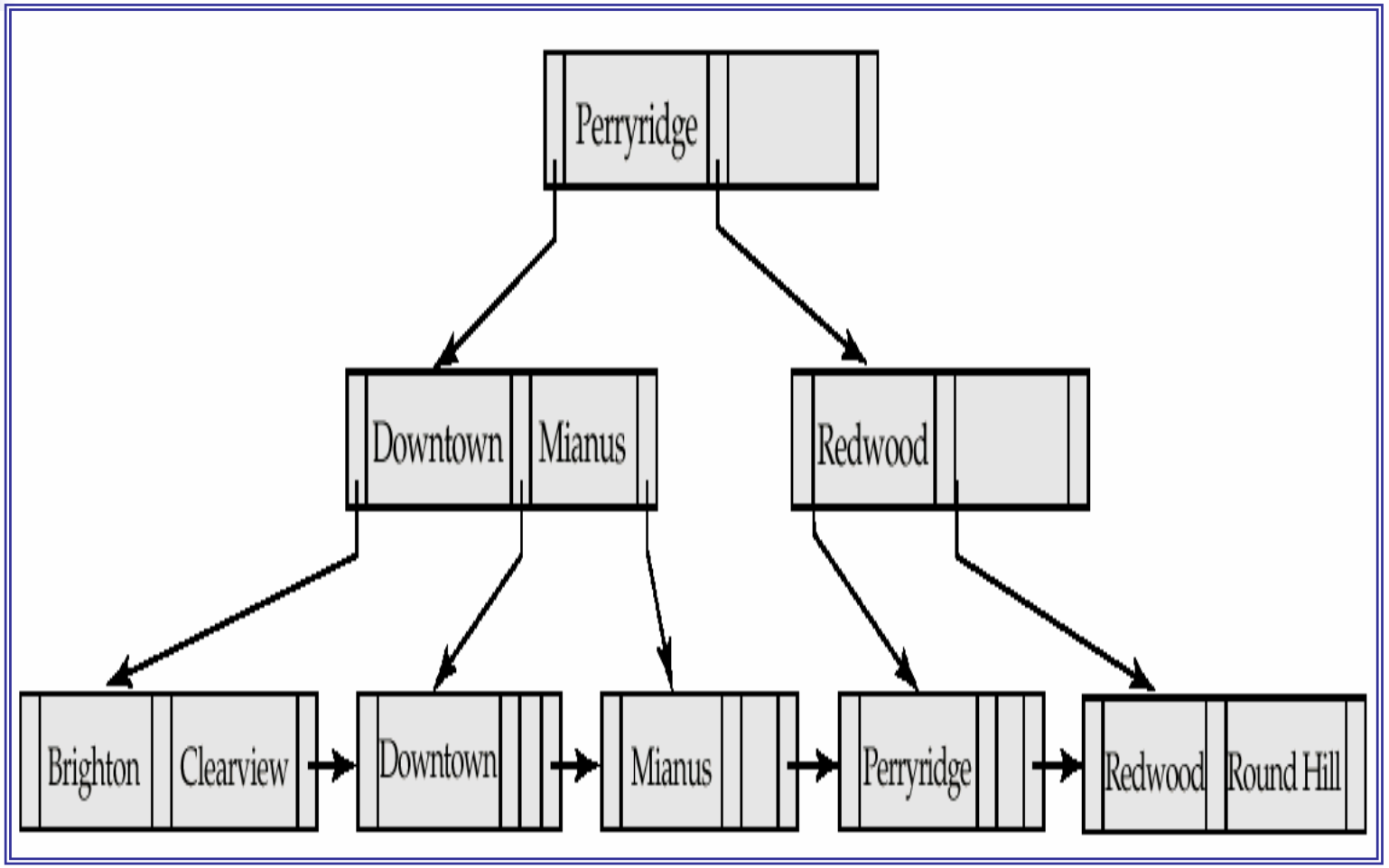
- B^+-트리는 B-트리의 변형으로 단말 노드가 아닌 노드로 구성된 인덱스 세트(Index Set)와 단말 노드로만 구성된 순차 세트(Sequence Set)로 구분된다.
- 인덱스 세트에 있는 노드들은 단말 노드에 있는 키 값을 찾아갈 수 있는 경로로만 제공되며, 순차 세트에 있는 단말 노드가 해당 데이터 레코드를 가리키고 있다. 또한, 모든 키값은 단말 노드에 나타난다.
- B^+-트리에서의 직접접근에는 인덱스 세트가 이용되고, 순차접근에는 순차 세트가 이용된다.
- 차수 m인 B^+-트리의 특징
  - 루트 노드와 단말 노드를 제외한 모든 노드는 최소 $\lceil m/2 \rceil$개, 최대 m개의 서브트리를 가진다.
  - 루트 노드는 0 또는 2에서 m개 사이의 서브트리를 가진다.
  - 리프가 아닌 노드에 있는 키 값의 수는 그 노드의 서브트리 수보다 하나 작다.
  - 모든 단말 노드는 같은 레벨에 있다. 즉 루트로부터 같은 거리에 있다.
  - 한 노드 안에 있는 키 값들은 오름차순을 유지하고, 순차 세트내의 단말 노드들은 모두 링크로 연결되어 있다.

B+-tree (2/3)



<차수가 3인 B+-트리 예>

B^+–tree (3/3)

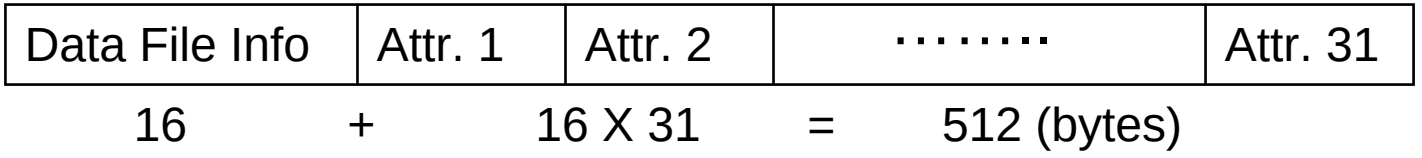

Perryridge
Downtown
Mianus
Redwood
Brighton
Clearview
Downtown
Mianus
Perryridge
Redwood
Round Hill

REQUIEM File Organization

- Each relation is stored as a combination of a data and index file
  - data file (rel_name.db) : File header + Data records
  - index file (rel_name.indx): File header + Index blocks
- Records in the stored data file represent tables of the base relations
- Each record is stored as a byte string of fixed length, corresponding to the maximum length of a tuple
- Fixed length storage space
  - Consumes more space
  - Faster
  - Simple
  - File manager does not have to save tuple identifiers (including tuple lengths, field lengths, …)

Data File Header

- Structure HEADER
  - describe the contents of a relation data file
  - hd_size : the size of each data tuple in bytes
  - hd_tcnt : the number of tuples
  - hd_available : the address of the first free record(linked list)
  - hd-tmax : relative position of last tuple
  - Structure ATTRIBUTE hd_attrs [NATTRS]
- Structure ATTRIBUTE
  - implement relation attribute
  - at_name : attribute name
  - at_size : attribute size
  - at_type : data type
  - at_key : index type

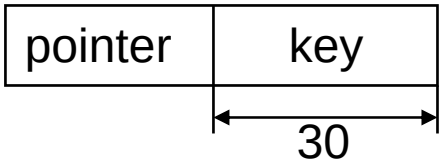


Implementation of B+-tree Structures (1/4)

- all keys stored in a B+-tree are character strings
- supports multiple keys (a secondary key)
- dummy entry (0xffffffff) which hold highest possible key value
- Initial B+-tree structure comprises one leaf level block containing the dummy entry
- Four index levels are allowed
  - index block of 1 Kbytes (1024bytes)

Implementation of B+-tree Structures (2/4)

- Entry structure
  - store keys and addresses of index or data records in the B+-tree



- Index Block structure

blk_pos	blk_offset	lvl	entries (entr1, entr2, …, entr n)

  - blk_pos : index file location of block
    or location of next free block
  - blk_offset : first unused location in block
  - lvl : record level no. -1 for free block
  - entries[IXBLK_SPACE] : space for entries (1016 byte)
  - total length : 1024 bytes

Implementation of B+-tree Structures (3/4)

- Index File Header

nl	root_bk	first_fbk	indx_attrs	dum_entr

  - nl : no. of index levels
  - root_bk : location of root block in file
  - first_fbk : address of first free block
  - indx_attrs : indexed attribute entry
  - dum_entr : dummy entry

- Index attributes

ref key1	ref key2	prim1	prim2	secnd1	secnd2	ix_rel	ix_foregin

  - ix_ref key$_i$: referenced attributes
  - ix_prim$_j$: primary attributes
  - ix_secnd$_k$: secondary attributes
  - ix_rel : relation name
  - ix_foregin : indicates a foreign key

Implementation of B+-tree structures (4/4)

- String PO* always prefixes primary key values to distinguish them from secondary keys values which are prefixed by SO * and SI * .

622, PO * 125	677, PO * 128	...	732, PO * 142	622, SO * Gray	...

- Size of an index block
  - Blocks : units in which the OS allocates disk spaces to files
  - Pages : units of transfer between disk and memory space
  - Index block should correspond to page size, otherwise I/O performance degrades
  - Optimal block size for index blocks for most Unix systems is 1 Kbytes

Cache Buffering Scheme (1/2)

- reduce the number of actual disk reads
- Cache buffers
  - a collection of blocks
  - kept in memory space to improve performance
  - Use LRU (Least recently used) techniques
- read requests can be satisfied without a disk access
- any updates or deletes are performed simultaneously on the contents of the cache blocks and on corresponding index file blocks held on disk

Cache Buffering Scheme (2/2)

- REQUIEM
  - allocate space for one block at each index level and keep the current block in the cache
  - every open index file obtains its own set of cache buffers

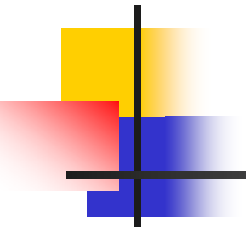
B^+-tree Management (1/2)

- Most of the functions managing the B^+-tree index files take two arguments
  - address of an index file header
  - structure of the type ENTRY
- B^+-tree searching function yields data record addresses to retrieve the data file records
  - Function FIND_IX() : locate the first entry that contains a key value
  - Function GET_CURRENT() : preserve the entry at the current position in the index
  - Function GET_NEXT() : advance the current position
  - Function GET_PREVIOUS() : move the current position backward in the index file

B^+-tree Management (2/2)

- Function FIND_INS()
  - to insert a new record into B^+-tree
  - call the function FIND_EXACT() to verify that the current entry is not already present in the index file
  - call function INSERT_IX() to add its entry argument in front of the current position

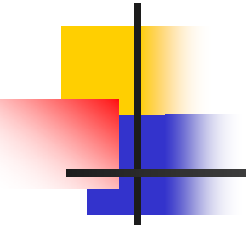
- Function FIND_DEL()
  - to delete an entry from B^+-tree
  - call FIND_EXACT() to locate the desired entry in the B^+-tree
  - call DEL_IX() to remove the current entry from the index block



File Management

- Functions for file access
  - DB_ROPEN(rname) : open a relation file
  - DB_RCLOSE(sptr) : close a relation file

- Functions for tuple access
  - FETCH_TUPLE(sptr): fetch the tuple
  - STORE_TUPLE(sptr, key_buf): store the tuple
  - DELETE_TUPLE(sptr, del_buf): delete the tuple
  - UPDATE_TUPLE(sptr, key_buf, del_buf): update the tuple



Access Path Implementation

- Access path modules (APMs) handle all actual data accessing in the stored relations

- Functions implementing the access path code
  - Index creation routines
  - Key searching functions
  - Key insertion routines
  - Key updating functions