Database Management Systems 2010/11

– Chapter 3: Indexing and Hashing –

J. Gamper

- Basic Concepts
- Ordered indices
- B+-Tree Index Files
- B-Tree Files
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Index Definition in SQL
- Multiple-Key Access

These slides were developed by:
– Michael Böhlen, University of Zurich, Switzerland
– Johann Gamper, University of Bozen-Bolzano, Italy

Basic Concepts

- Indexing mechanism are used to speed up access to data
  - e.g., author catalog in library, book index
- **Index file:** Consists of records (called **index entries**) of the form (*search-key,pointer*) where
  - **search-key** is an attribute or set of attributes used to look up records in a data file
  - **pointer** is a pointer to a record (database tuple)in a data file
- Duplicates in an index file are allowed
- Index files are typically much smaller than the original file
- Two basic kinds of indices
  - **Ordered indices:** Search keys are stored in sorted order
  - **Hash indices:** Search keys are distributed uniformly across "buckets" using a "hash function"

Basic Concepts . . .

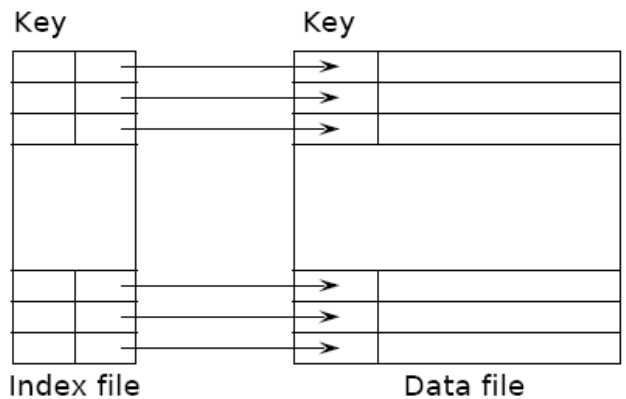
- **Evaluation of an index** must include
  - Access Time
  - Insertion Time
  - Deletion Time
  - Space overhead
  - Access Type supported efficiently, e.g.,
    - Records with a **specific value** in the attribute
    - Reocrds with an attribute value falling in a **specific range of values**

Ordered Indices

- **Ordered Index** Index entries are stored sorted on the search-key value
  - e.g., author catalog in library
- Different forms of ordered indices
  - Primary index (clustring index)
    - Dense index
    - Sparse index
  - Secondary index (non-clustring index)
    - Must be a dense index

Primary Index

- **Primary index (clustering index):** Index whose search-key order corresponds to the sequential order of the data file (table rows).
  - The search key of a primary index is usually but not necessarily the primary key

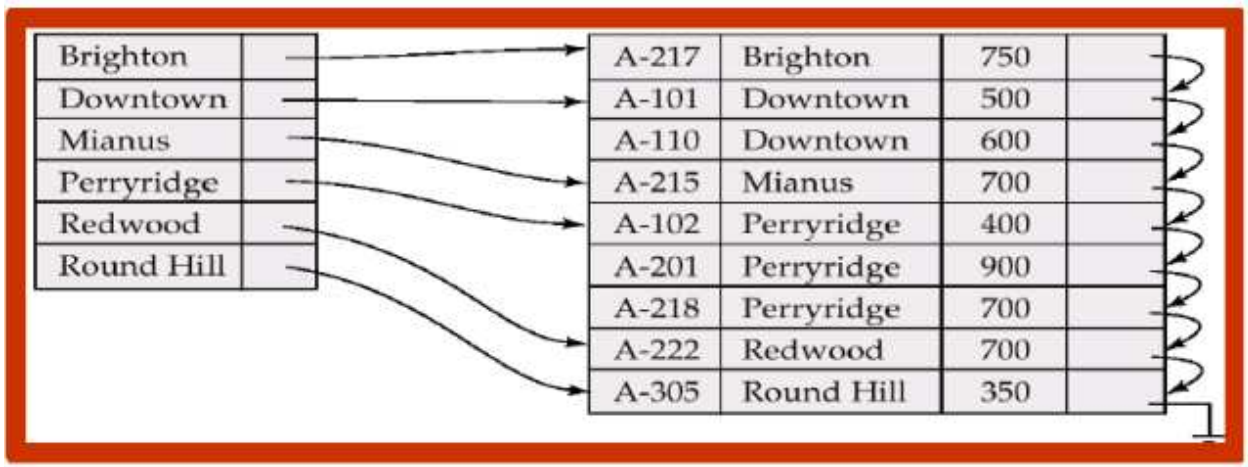


Primary Index ...

- **Index-sequential file:** Ordered sequential file with a primary index
  - Both index and data are stored on sequential files (index file and data file)
  - Designed for both efficient sequential access and random access
  - One of the oldest indexing techniques in DB

Dense Index

- **Dense index:** Index record appears for every search-key value in the file.
  - Index record contains a pointer to the first data record with that search-key value
  - The rest of data records with that search-key are stored sequentially
  - Alternative: an index record for each data record



Dense Index ...

- **Lookup** a record with search-key value K:
  1. Find index record with search-key value $= K$
  2. Search data file sequentially starting at the record to which the index record points
- Lookup with dense index is efficient for several reasons
  - Number of index blocks is usually small compared to the number of data blocks
  - Complete index might fit in main memory: no expensive block I/O
  - Otherwise, since keys are sorted, binary search can be used to find K: $\log_2 n$ blocks are accessed

Dense Index ...

- **Index update: Deletion** of a record with search-key value K
  1. Look up the record to be deleted
  2. If deleted record was the only record in the file with this search-key value, K is deleted from the index
  3. Otherwise, if the deleted record was the first record with search-key value K, the index record is updated to point to the next data record

- If the index stores pointers to all data records with the same search-key value, step 3 needs only to delete the pointer from the index entry.

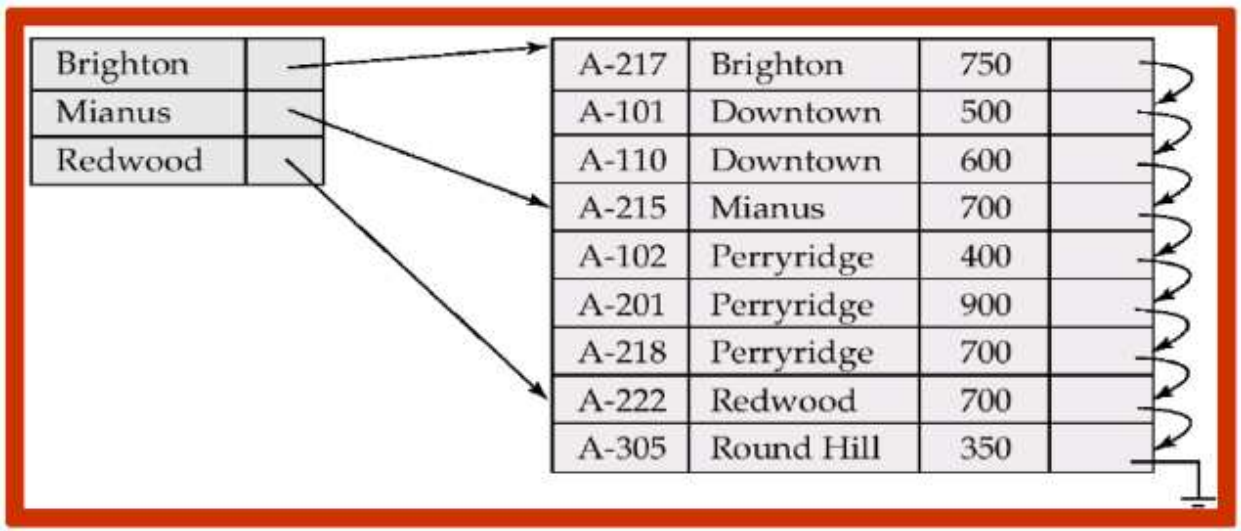
Dense Index ...

- **Index update: Insertion** of a record with search-key value K
  1. Perform a lookup with search-key value K
  2. If K does not appear in the index, an index record with the search-key K is inserted at the appropriate position
  3. Otherwise, the index needs not to be updated; the new data record is placed after the other records with search-key K

- If the index stores pointers to all data records with the same search-key value, step 3 adds to the index entry a pointer to the new data record.

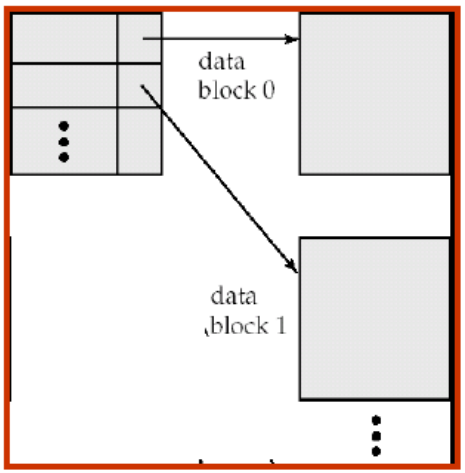
Sparse Index

- **Sparse index:** Contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key (i.e., primary index)



Sparse Index ...

- Compared to dense indices
  - Less space and maintenance overhead for insertion and deletion
  - Generally slower than dense index for locating records.
- Good trade-off
  - Sparse index with an index entry for every block in file
  - Index entry stores least search-key value of the block it points to



Sparse Index ...

- **Lookup** a record with search-key value K
  1. Find index record with largest search-key value $<= K$
  2. Search file sequentially starting at the record to which the index record points

Sparse Index ...

- **Index update: Deletion** of record with search-key K
  1. Look up the record to be deleted
  2. If the index does not contain an entry with search-key value K,nothing needs to be done on the index
  3. Otherwise, the following actions are taken
     - If the deleted record was the only one with search-key K, the corresponding index entry is replaced by an index entry for the next search-key value (in search-key order)
       If the next search-key value already has an index entry, the index entry for K is deleted instead of being replaced
     - If the index entry for K points to the record being deleted (the first record with that index), the index entry is updated to point to the next record with search-key K

Sparse Index ...

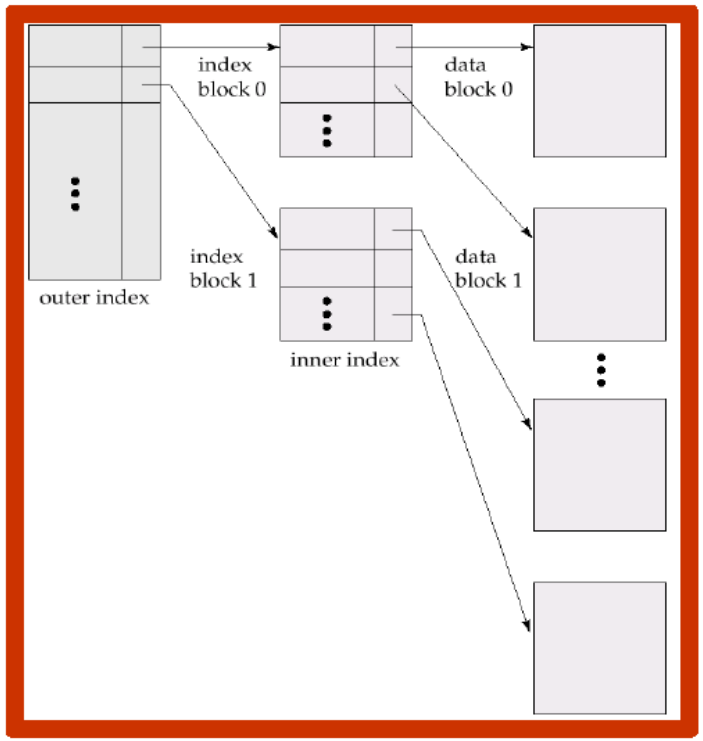
- **Index update: Insertion** of a record with search-key value K (assume the index stores an entry for each data block)
  1. Perform a lookup with search-key value K
  2. If a new block is being created, the least search-key value appearing in the new block is inserted into the index
  3. Otherwise, if the new record has the least search-key value in its block, the index entry pointing to that block is updated; if not, no changes to the index are required

Multilevel Index

- If the primary index does not fit in memory, access becomes more expensive
  - A search for a data record requires several disk block reads from the index file
  - Binary search might be used on index file
    - $\log_2 b$ disk block reads, where b is the total number of index blocks
  - If overflow blocks are used in the index file, binary search is not applicable, and sequential scan is required
    - b disk block reads are required
- To reduce the number of index block I/Os, treat primary index kept on disk as a sequential file and construct a sparse index on it $\Rightarrow$ **multilevel index**

Multilevel Index ...

- **Multilevel index**
  - Inner index: The primary index file on the data
  - Outer index: A sparse index on the primary index (i.e., inner index)
- If even the outer index is too large to fit in main memory, yet another level of index can be created, etc.



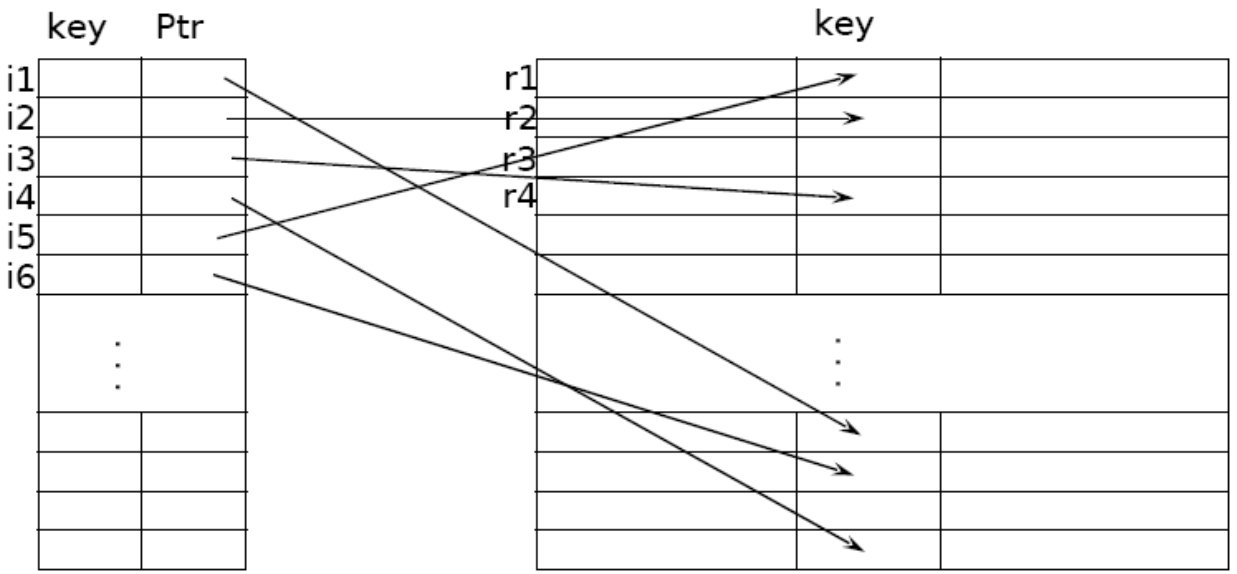
- **Index update: Deletion and insertion**
  - Indices at all levels must be updated on insertion and deletion in the data file
  - Update starts with the inner index
  - Algorithms are simple extensions of the single-level algorithms

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:** Consider an account relation that is stored sequentially by account number
  - Find all accounts in a particular branch
  - Find all accounts with a specified balance or range of balances
- An additional index is needed to answer such queries efficiently
  $\Rightarrow$ **secondary index**

Secondary Indices ...

- **Secondary index (non-clustering index):** Index whose search key specifies an order **different** from the sequential order of the file.

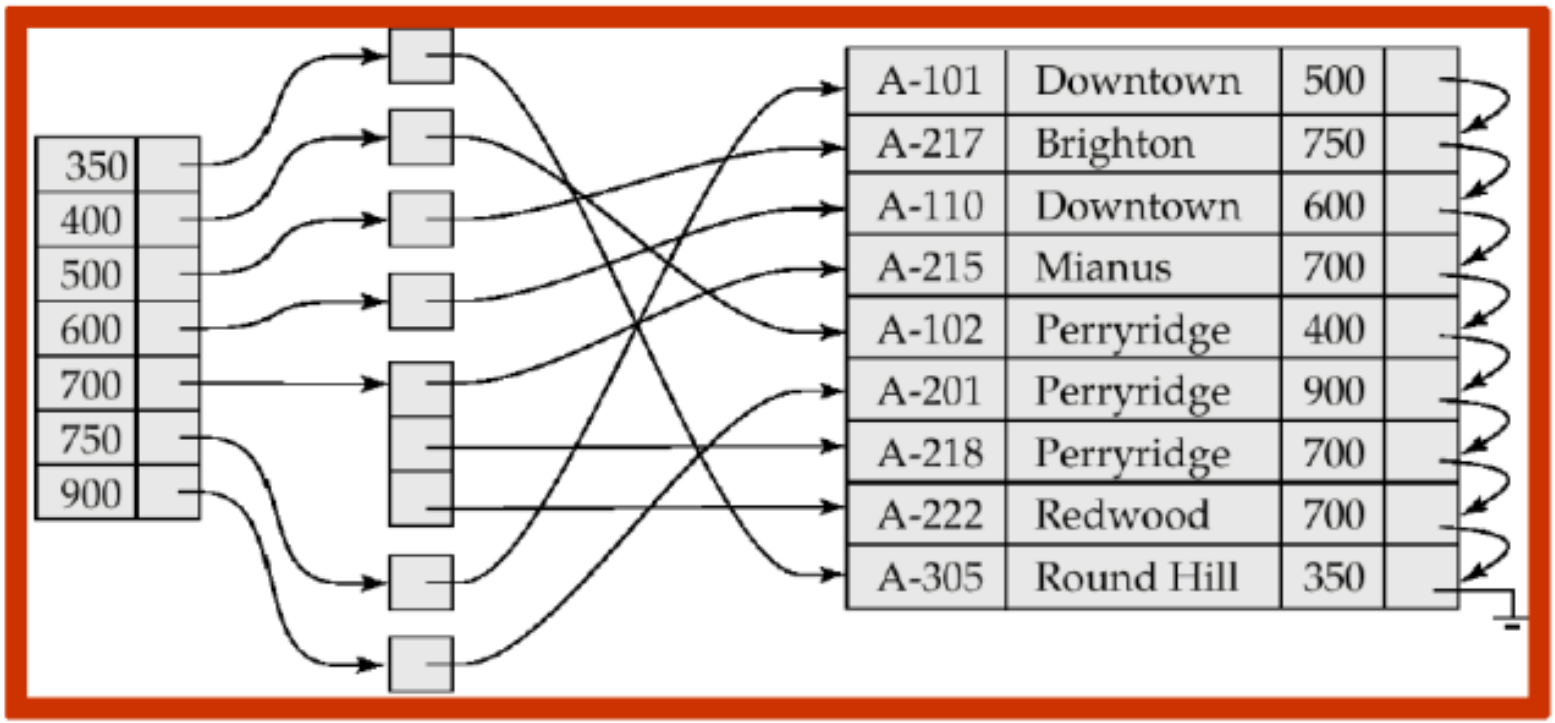


Secondary Indices ...

- Secondary indices **must be dense** with an index entry for every search-key value and a pointer to every record in the data file.
  - With a sparse index, records with intermediate search-key values (or multiple records with the same search-key) may be anywhere in the file
- Two options for data pointers
  - **Duplicate index entries**: an index record for every data record
  - **Buckets**: An index record for each search-key value; index record points to a bucket that contains pointers to all the actual records with that particular search-key value

Secondary Indices ...

- **Example:** Secondary index on the balance field of the account relation using buckets



Primary vs. Secondary Indices

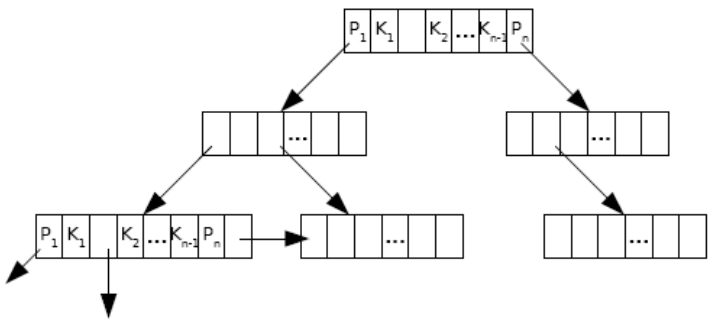
- Indices offer substantial benefits for lookups
- BUT: Updating indices imposes overhead on DB modifications
  - When data are modified, every index must be updated, too
- Primary indices can be dense or sparse
- Secondary indices must be dense
- Sequential scan using primary index is efficient
- BUT: A sequential scan using a secondary index is expensive
  - Each record access may fetch a new block from disk

B^+-Tree Index Files

- Main problem of index-sequential file organization
  - Performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required
- **B^+-tree index** is a multi-level index and is an alternative to index-sequential files
  - Advantage of B+-tree index files
    - Automatically maintains as many levels of index as appropriate
    - Automatically reorganizes itself with small, local changes, in the face of insertions and deletions; reorganization of entire file is not required to maintain performance
  - Disadvantage of B+-tree
    - Extra insertion and deletion overhead as well as space overhead
- Advantages of B+-trees outweigh disadvantages, and they are used extensively

B^+-Tree Index Files ...

- **B^+-tree:** a rooted tree with the following properties
  - Balanced tree, i.e., all paths from root to leaf are of the same length
    - at most $\lceil log_{\lceil n/2 \rceil}(K) \rceil$ for K search-key values
  - A node contains up to $n-1$ search-key values and n pointers
  - The search-key values in a node are sorted
- Nodes are between half and completely full
  - Internal nodes have between $\lceil n/2 \rceil$ and n children
  - Leaf nodes have between $\lceil (n-1)/2 \rceil$ and $n-1$ search-key values
  - Root node: If it is a leaf, it can have between 0 and $(n-1)$ search-key values; otherwise, it has at least 2 children
- Data pointers are stored at leaf nodes only
- Leaf nodes are linked together



B^+-Tree Node Structure

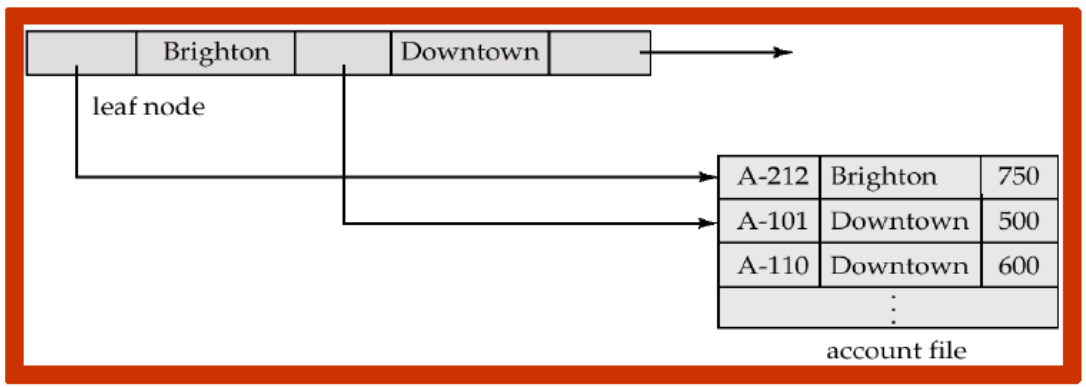
- **General node structure**

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

  - $K_1, \ldots, K_{n-1}$ are the search-key values
  - $P_1, \ldots, P_n$ 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: $K_1 < K_2 < K_3 < \cdots < K_{n-1}$

B^+-Tree Node Structure ...

- **Leaf node**
  - $P_1, \ldots, P_{n-1}$ either point to a file record with search-key value K_i or to a bucket of pointers to file records with search-key value K_i
    - Bucket structure is only needed if search-key does not form a primary key
  - Pointer P_n points to next leaf node in search-key order
  - For $i < j$, the search-key values in node i are less than the search-key values in node j



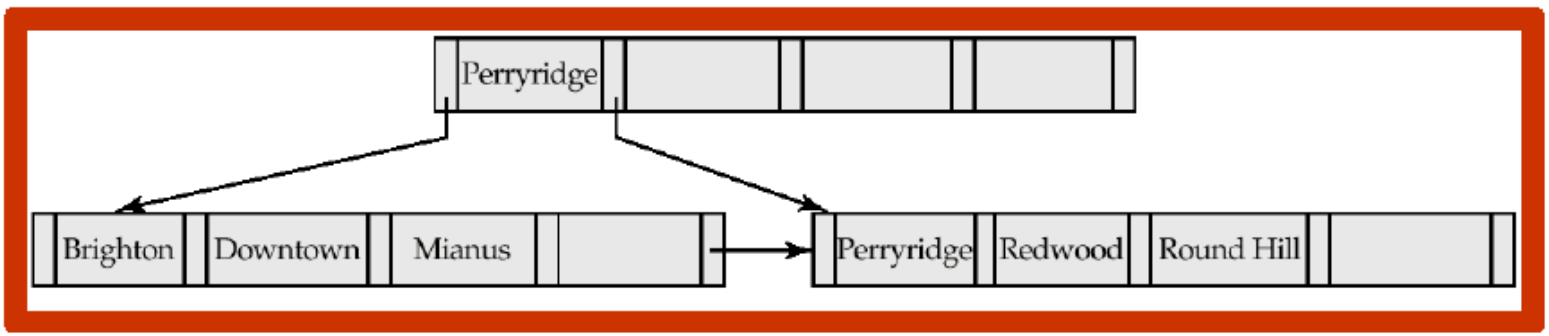
B^+-Tree Node Structure

- **Non-leaf nodes**
  - Form a multi-level sparse index on the leaf nodes
  - For a non-leaf node with n pointers
    - P_1 points to the subtree where all search-key values are less than K_1
    - For $2 \leq i \leq n-1$: Pointer P_i points to the subtree where all search-key values are greater than or equal to K_{i-1} and less than K_i
    - Pointer P_n points to the subtree where all search-key values are greater than or equal to K_{n-1}

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

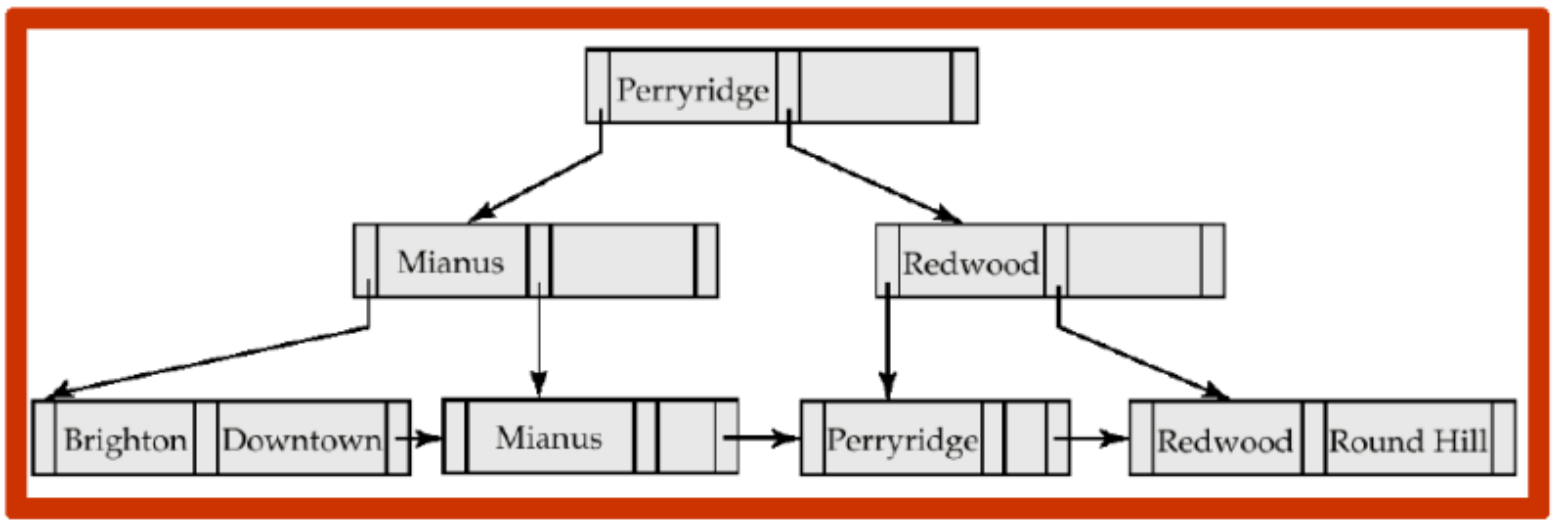
Example of B$^+$-tree

- **Example:** B$^+$-tree for account file with $n = 5$ pointers per node
  - Leaf nodes: between 2 and 4 search-key values (i.e., $\lceil (n-1)/2 \rceil$ and $n-1$)
  - Non-leaf nodes other than root: between 3 and 5 children (i.e., $\lceil n/2 \rceil$ and n)
  - Root node: at least 2 children



Example of B$^+$-tree ...

- **Example:** B$^+$-tree for account file with $n = 3$ pointers per node
  - Leaf nodes: between 1 and 2 search-key values (i.e., $\lceil (n-1)/2 \rceil$ and $n-1$)
  - Non-leaf nodes other than root: between 2 and 3 children (i.e., $\lceil n/2 \rceil$ and n)
  - Root node: at least 2 children



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 (= multilevel index on leaf nodes)
- The B^+-tree contains a relatively small number of levels
  - $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$ for K search-key values in the file
- Search is efficient, since only a small number of index blocks need to be read
  - Compare to the $\log_2 b$ disk block reads for binary search in index-sequential files
  - Typically the root node and perhaps the first level nodes are kept in main memory, which further reduces the disk block reads.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time

Queries on B^+-Trees

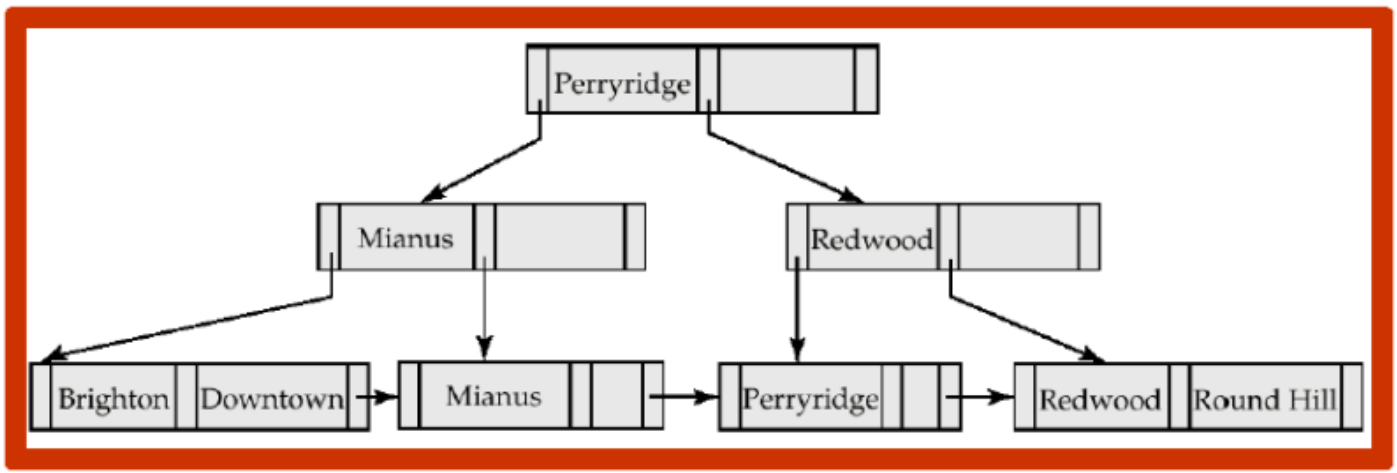
- Find all records with a search-key value of k

Algorithm: *FIND*(k)

Set C = root node;
while C *is not a leaf node* **do**
  Examine C for the smallest search-key value $> k$;
  if *such a value exists* **then**
    Assume it is K_i;
    Set C = the node pointed to by P_i;
  else
    $k \geq K_{m-1}$, where m = the number of pointers in the node;
    Set C = the node pointed to by P_m;
if *there is a key value* K_i *in* C *such that* $K_i = k$ **then**
  Follow pointer P_i to the desired record or bucket;
else
  No record with search-key value k exists;

Queries on B^+-Trees ...

- **Example:** Find all records with a search-key value equal to "Mianus"
  - Start from the root node
  - "Perryridge" is the smallest search-key $>$ "Mianus", thus follow first pointer
  - No search-key $>$ "Mianus" exists, thus follow last pointer
  - Search-key $=$ "Mianus" exists, thus follow the data pointer



Queries on B^+-Trees ...

- In processing a query, a path is traversed from the root to some leaf node
- For K search-key values in the data file, the path length is at most $\lceil log_{\lceil n/2 \rceil}(K) \rceil$
- A node generally corresponds to a disk block, typically 4KB, and n is typically ≈ 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 (or binary search) with 1 million search key values: around 20 nodes are accessed in a lookup
  - This difference is significant since every node access may need a disk I/O, costing around 20 milliseconds!

Updates on B$^+$-Trees: Insertion

- **Insert** a record r with search-key value of k

Algorithm: *INSERT*(r, k)

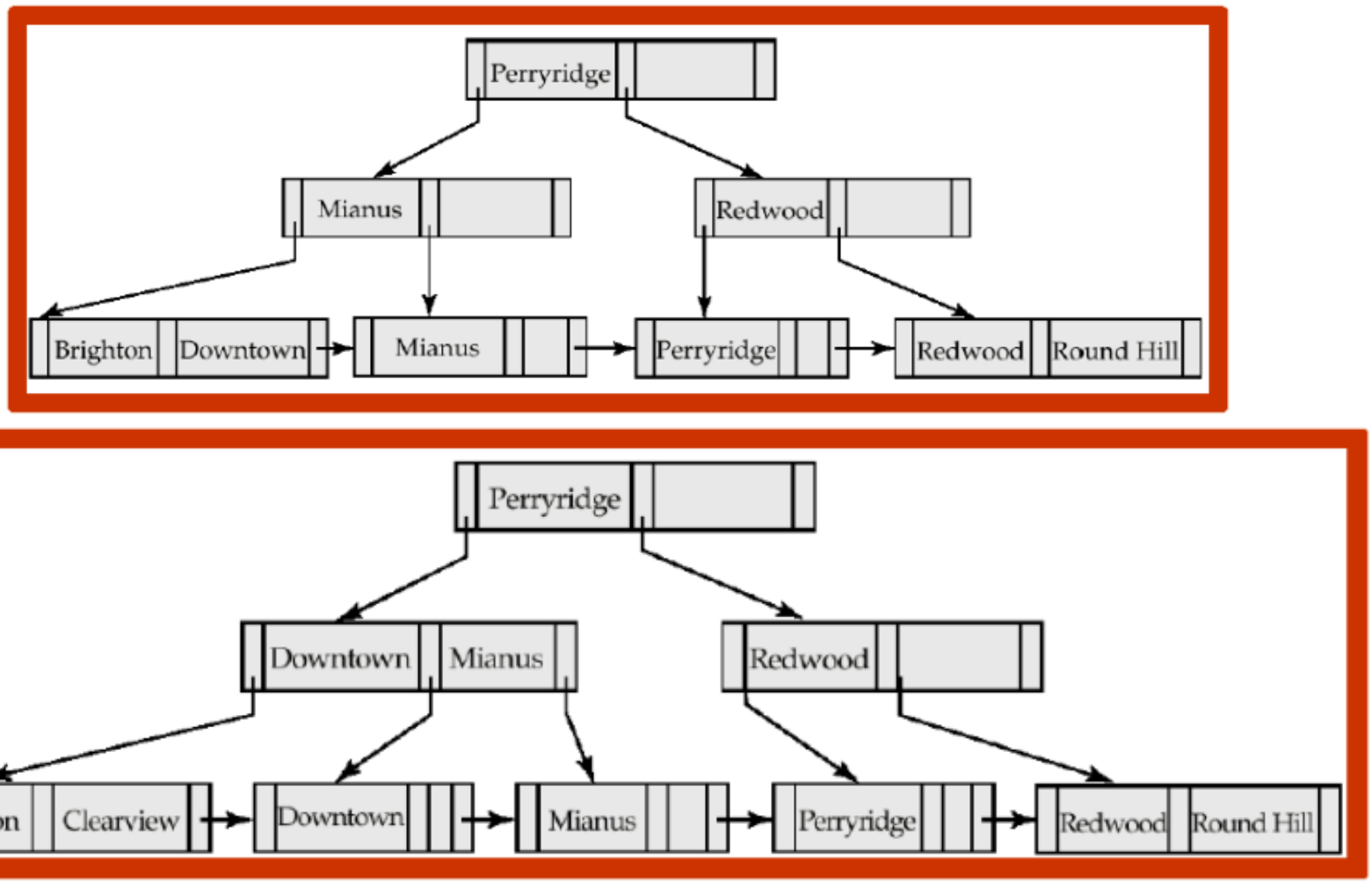
Find the leaf node in which k value would appear;
if *k is already there* **then**
    Add r to the data file;
    Insert a pointer into the bucket if necessary;
else
    Add r to the data file (and create a bucket if necessary);
    if *there is room in the leaf node* **then**
        Insert (*key-value*, *pointer*) pair in the leaf node such that the search keys are still in order
    else
        Split the node (along with new (*key-value*, *pointer*) entry);

Updates on B^+-Trees: Insertion ...

- **Splitting** a node
  1. Take the n (*search-keyvalue*, *pointer*) pairs (including the new 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
  2. Insert (k, p) in the parent of the node being split
  3. If the parent is full, split it and propagate the split further up
     - Splitting proceeds upwards till a node that is not full is found
     - In the worst case the root node may be split, increasing the height of the tree by 1

Updates on B$^+$-Trees: Insertion ...

- **Example:** B$^+$-tree before and after insertion of "Clearview"



Updates on B$^+$-Trees: Deletion

- **Deletion** of a record with search-key k

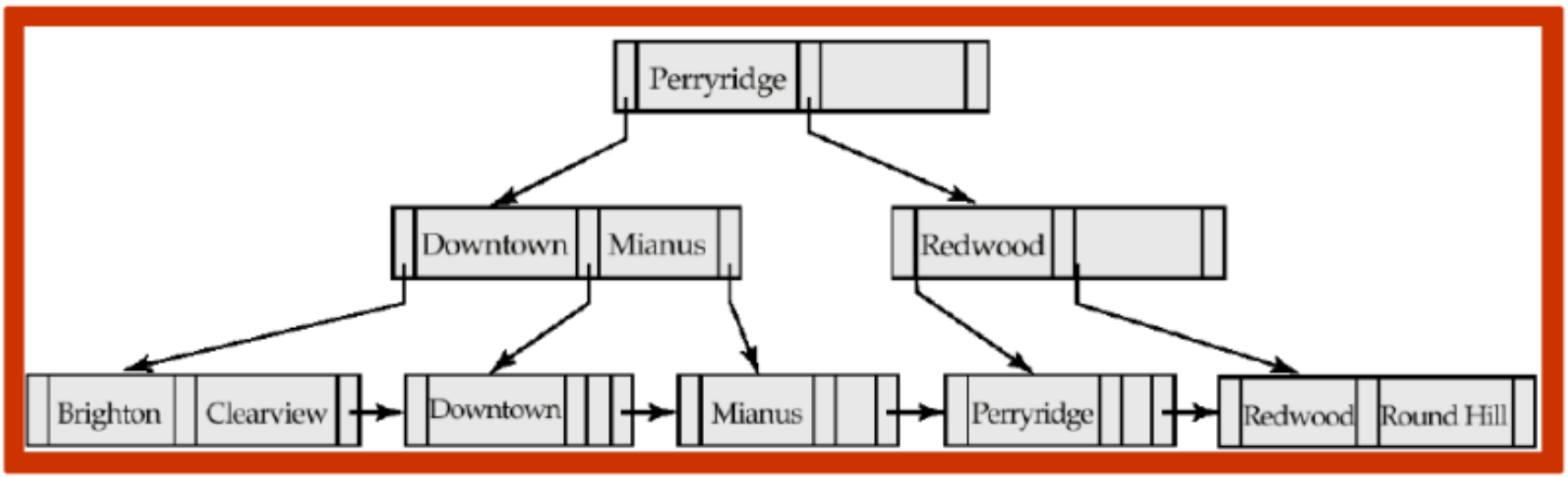
  Algorithm: $DELETE(k)$

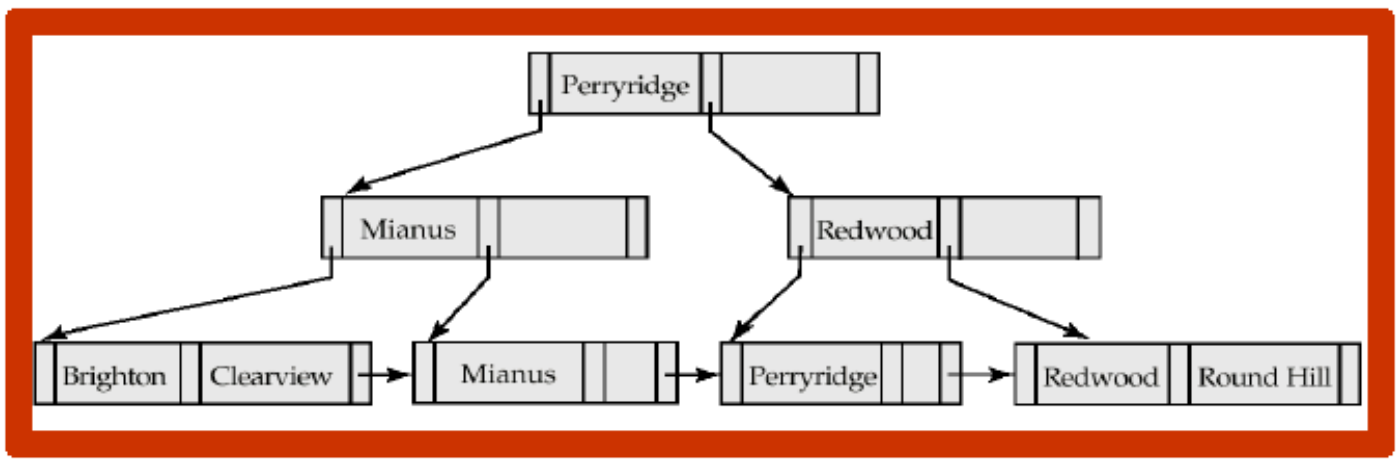
  Find the record to be deleted and remove it from data file (and bucket);
  if *there is no bucket or bucket has become empty* **then**
    Remove (*search-key*, *pointer*) pair from the leaf node;
  if *node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node* **then**
    Merge siblings, i.e., insert all search-key values in the two nodes into a single nod
    (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;
  if *node has too few entries due to the removal, and the entries in the node and a sibling fit not into a single node* **then**
    Redistribute the pointers between the node and a sibling such that both have mor
    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

Updates on B$^+$-Trees: Insertion ...

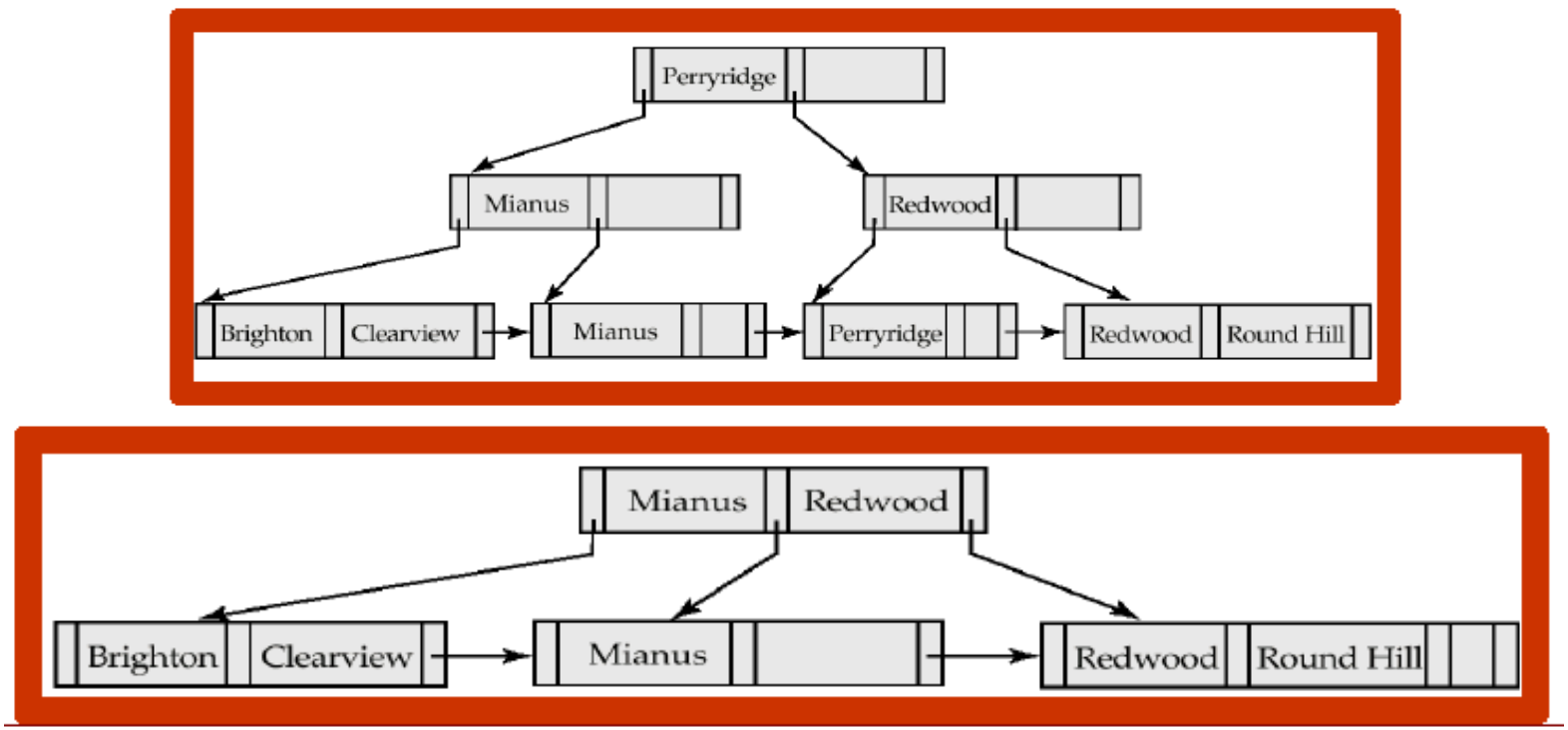
- **Example:** Before and after the deletion of "Downtown"
  - The removal of the leaf node containing Downtown did not result in its parent having too little pointers. So the cascaded deletions stopped with the deleted leaf nodes parent.





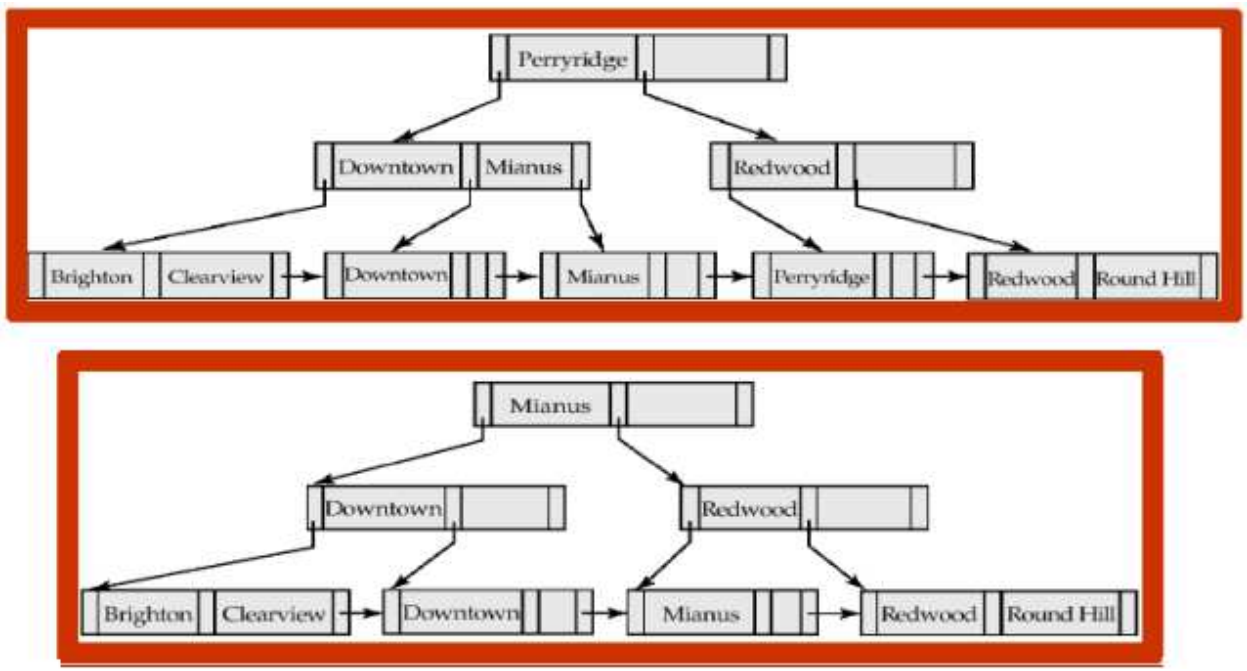
Updates on B$^+$-Trees: Insertion ...

- **Example:** Before and after deleting "Perryridge"
  - Node with "Perryridge" becomes underfull and is merged with its sibling
  - As a result "Perryridge" node's parent becomes underfull, and is merged with its sibling (and an entry is deleted from their parent)
  - Root node then has only one child and is deleted



Updates on B$^+$-Trees: Insertion ...

- **Example:** Before and after deleting "Perryridge" from an earlier example
  - Parent of leaf containing "Perryridge" became underfull and borrowed a pointer from its left sibling
  - Search-key value in the parent's parent changes as a result
  - Root node then has only one child and is deleted

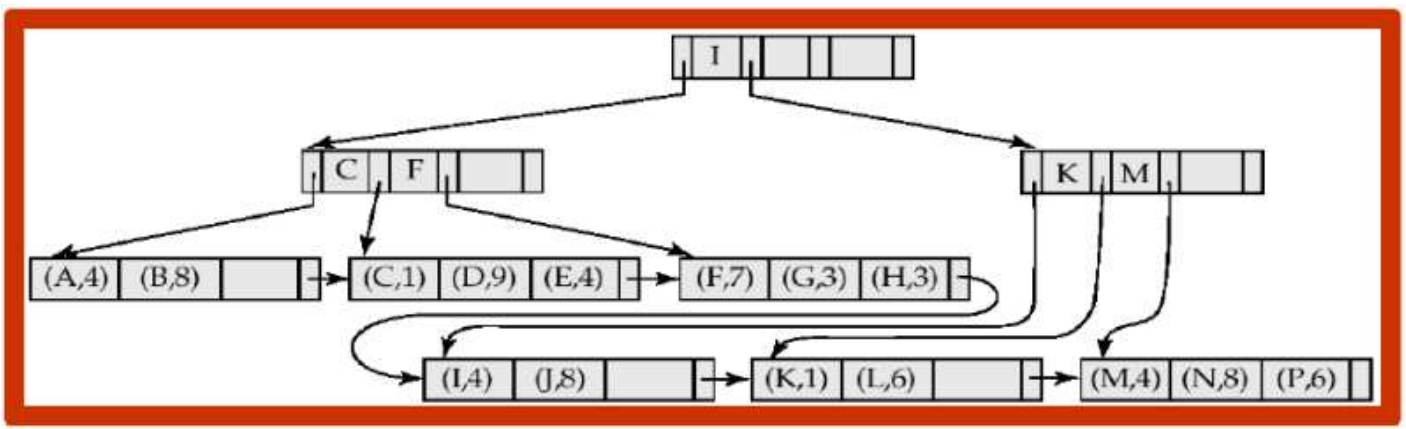


B^+-Tree File Organization

- Index file degradation problem is solved by using B^+-Tree indices
- Data file degradation problem is solved by using B^+-Tree file organization
- **B^+-Tree File Organization:** The leaf nodes in a B^+-tree file organization store records (instead of pointers)
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a non-leaf node
  - 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

B^+-Tree File Organization ...

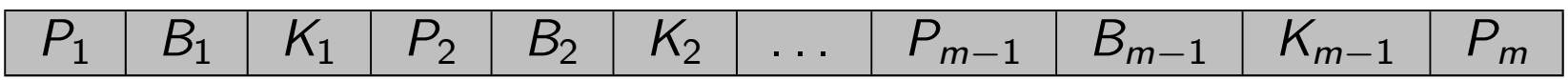
- Good space utilization is important since records use more space than pointers
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - During insertion, involving 2 siblings in redistribution results in each node having at least $\lfloor 2n/3 \rfloor$ entries
- **Example:** B^+- tree file organization



B-Tree Index File

- **B-tree:** Similar to B^+-tree with the following modifcations
  - Search-key values are allowed to appear only once
    - Eliminates redundant storage of search-keys.
    - Search-keys in non-leaf nodes appear nowwhere else in the B-tree
  - Leaf node: The same as for B^+-tree (Fig. a)
  - Non-leaf node: An additional pointer b_i for each search-key must be included, pointing to bucket or file record (Fig.b)

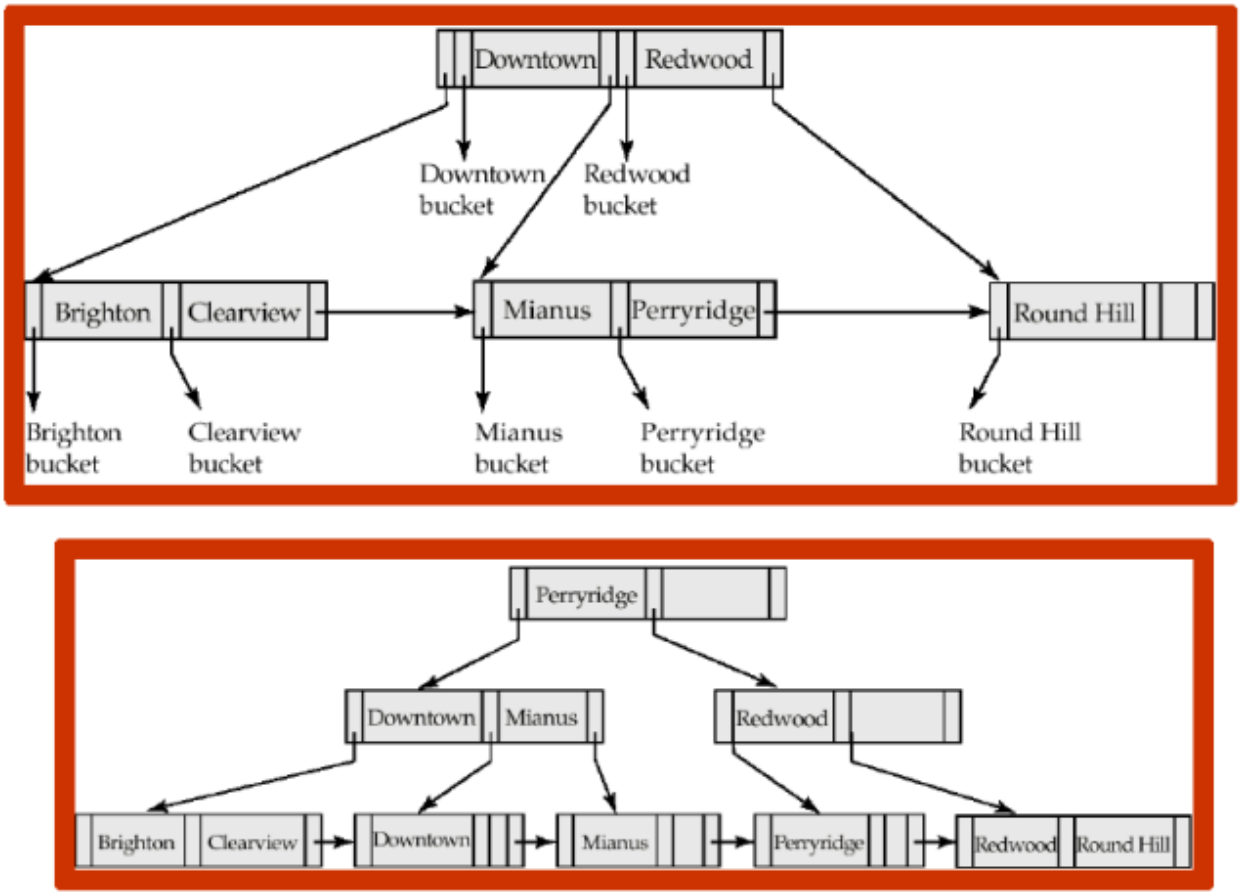
(a)



(b)

B-Tree Index File ...

- **Example:** B-tree and B^+-tree on the same data



B-Tree Index File ...

- **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 are more complicated than in B^+-trees
  - Implementation is harder than B^+-trees.
- Typically, advantages of B-trees do not out weigh the disadvantages.

Static Hashing

- Disadvantage of sequential and B^+-tree index file organization
  - Index structure must be accessed to locate data
  - Or binary search on sequential data file or on a large index file might be required
  - This yields additional block IO
- **Hashing**
  - provides a way to avoid index structures and to access data directly
  - provides also a way of constructing indices
- A **bucket** is a unit of storage containing one or more records (typically a disk block).

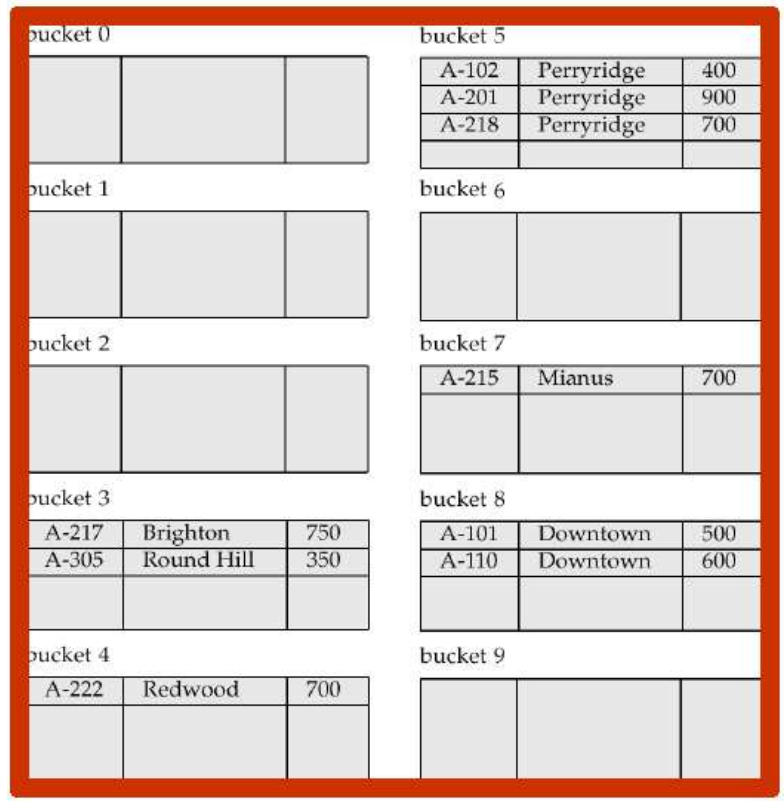
Static Hashing ...

- **Hash file organization**
  - We obtain the bucket of a record directly from its search-key value using a hash function.
    - Constant access time
    - Avoids the use of an index
  - **Hash function h:** A function from the set of all search-key values K to the set of all bucket addresses B.
  - Function h is used to locate records for access, insertion, and deletion.
  - Records with different search-key values may map to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Multilevel Index . . .

- **Example:** Hash file organization of account file, using branch-name as key

- 10 buckets
- Binary representation of the ith character is assumed to be i, e.g. $binary(B) = 2$
- Hash function h
  - Sum of the binary representations of the characters modulo 10, e.g.,
  - $h(Perryridge) = 5$
  - $h(RoundHill) = 3$
  - $h(Brighton) = 3$



Hash Functions

- Worst hash function maps all search-key values to the same bucket
  - This makes access time proportional to the number of searchkey values in the file.
- An **ideal hash function** has the following properties:
  - The distribution is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of all possible values.
  - The distribution is **random**, so in the average case each bucket will have the same number of records assigned to it irrespective of the actual distribution of search-key values in the file.

Hash Functions ...

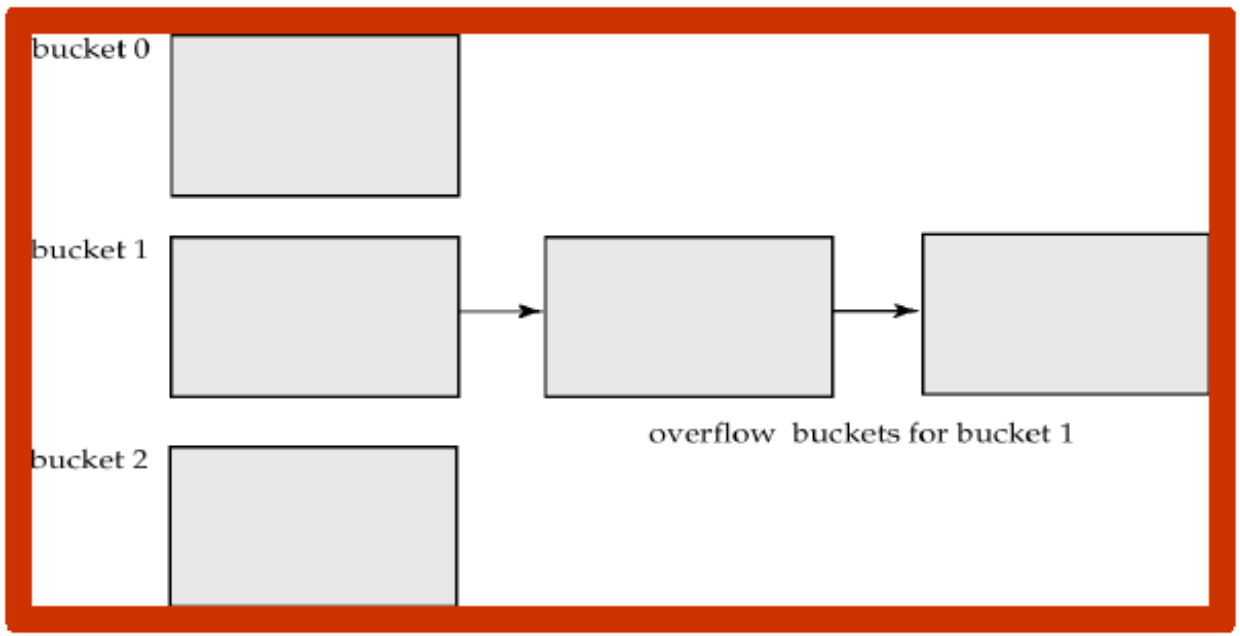
- **Example:** 26 buckets and a hash function that maps branch names beginning with the i-th letter of the alphabet to the i-th bucket
  - Simple, but not a uniform distribution, since we expect more branch names to begin, e.g., with B and R than Q and X.
- **Example:** Hash function on the search-key balance by splitting the balance into equal ranges: 1–10000, 10001–20000, etc.
  - Uniform but not random distribution
- **Typical hash function:** Perform computation on the internal binary representation of the search-key.
  - e.g., for a string search-key, add the binary representations of all characters in the string and return the sum modulo the number of buckets

Bucket Overflow

- **Bucket overflow:** If a bucket has not enough space, a bucket overflow occurs; two reasons for bucket overflow
  - Insufficient buckets: the number of buckets n_B must be chosen to be $n_B > n/f$, where n = total number of records and f = number of records in bucket
  - **Skew in distribution of records:** A bucket may overflow even when other buckets still have space. This can occur due to two reasons:
    - multiple records have same search-key value
    - hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it **cannot** be eliminated!
  - Handled by using overflow buckets

Bucket Overflow ...

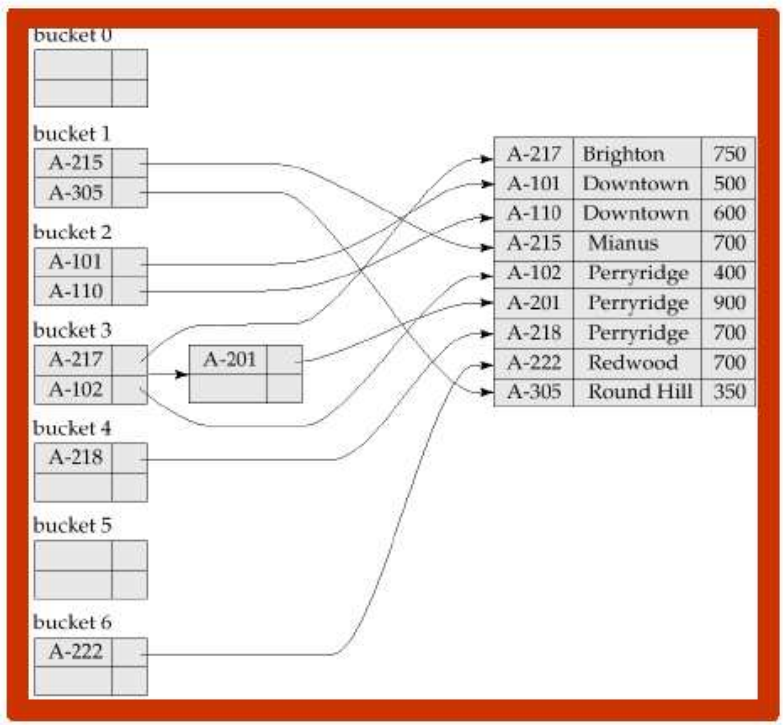
- **Overflow chaining (closed hashing)**
  - If a record is inserted into bucket b, and b is already full, an **overflow bucket** is provided, where the record is inserted
    - The overflow buckets of a given bucket are chained together in a list.



- **Open hashing:** An alternative, which does not use overflow buckets; not suitable for DB applications.

Hash Indices

- **Hash index:** organizes the search-key values with their associated record pointers into a hash file structure.
  - Buckets contain search-keys and pointers to the data records
  - Multiple (search-key, pointer)-pairs might be required (different from index-sequential file)
  - Secondary index (never needed as primary index)
- **Example:** Index on account; h = sum of digits in account-num modulo 7



- Term hash index is used to refer to both, secondary hash indices and hash file structures

Deficiencies of Static Hashing

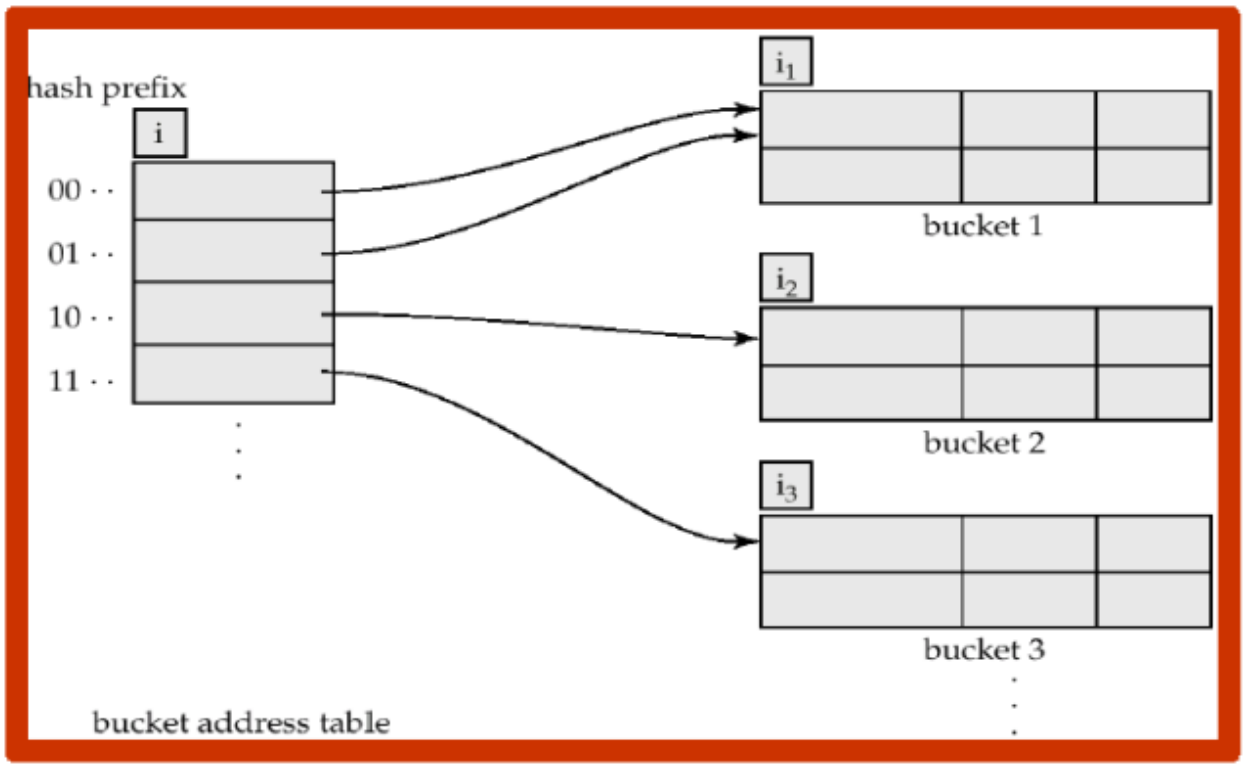
- In static hashing, the **fixed** set B of bucket addresses presents a serious problem
  - Databases grow and shrink with time
  - If initial number of buckets is too small, performance will degrade due to too much overflows.
  - If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space is wasted initially.
  - If database shrinks, again space will be wasted
  - One option is periodic re-organization of the file with a new hash function, but it is very expensive.
- These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically $\Rightarrow$ **dynamic hashing**

Dynamic Hashing

- **Dynamic hashing:** Allows the hash function to be modified dynamically.
- **Extendable hashing:** one form of dynamic hashing
  - Hash function h generates values over a large range — typically b-bit integers, with $b = 32$.
  - At any time use only a prefix of h to index into the bucket address table
  - Let the size of the prefix be i bits, $0 \leq i \leq 32$
    - Bucket address table has $size = 2^i$
    - Value of i grows and shrinks as the size of DB grows and shrinks; initially $i = 0$
  - The actual number of buckets is $\leq 2^i$
    - Multiple entries in the bucket address table may point to the same bucket.
    - All such entries have a common hash prefix, $i_j \leq i$, which is stored with each bucket j
    - The number of buckets changes dynamically due to coalescing and splitting of buckets.

Extendable Hashing

- General structure of **extendable hashing**
  - In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$ (thus, several entries point to bucket 1)



Lookup in Extendable Hashing

- **Lookup:** Locate the bucket containing search-key value K_j
  1. Compute $h(K_j) = X$
  2. Use the first i (hash prefix) high order bits of X as a displacement into the bucket address table, and follow the pointer to the appropriate bucket

Updates in Extendable Hashing

Insertion of a record with search-key value K_j

1. Use lookup to locate the bucket, say bucket j
2. **If** there is room in bucket j **then**
   - Insert the record in the bucket.
3. **Else** //The bucket j is split and insertion re-attempted
   - **If** $i > i_j$ (more than one pointer to bucket j) **then**
     - Allocate a new bucket z, and set i_j and i_z to the old $i_j + 1$.
     - Make the second half of bucket address table entries pointing to j point to z
     - Remove and reinsert each record in bucket j.
     - 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) **then**
     - 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.

- Overflow buckets needed instead of splitting (or in addition) in some cases, e.g., too many records with same hash value.

Updates in Extendable Hashing ...

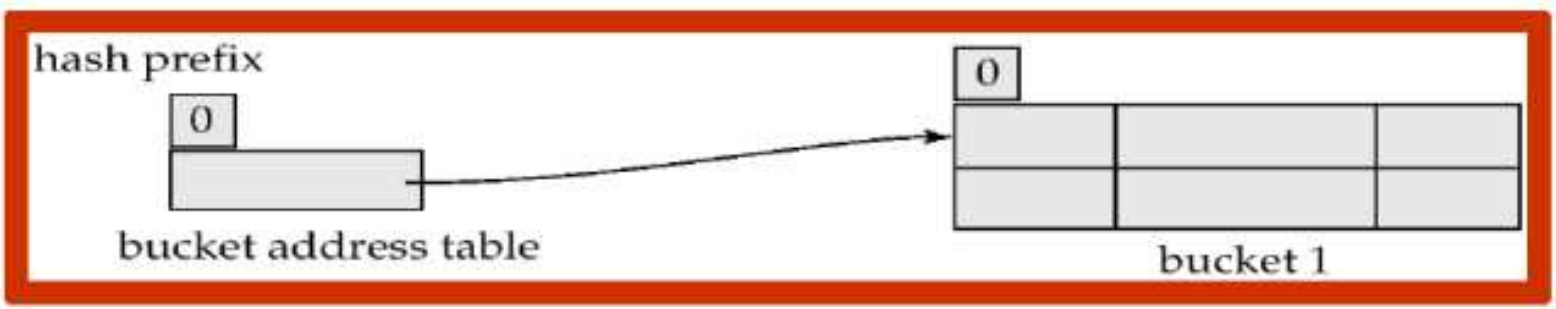
- **Deletion** of a key value K
  1. Locate K in its bucket and remove it (search-key from bucket and record from the file).
  2. The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  3. Coalescing of buckets can be done
     - Can coalesce only with a "buddy" bucket having the same value of i_j and the same $i_j - 1$ prefix, if it is present
  4. 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 able

Extendable Hashing: Example

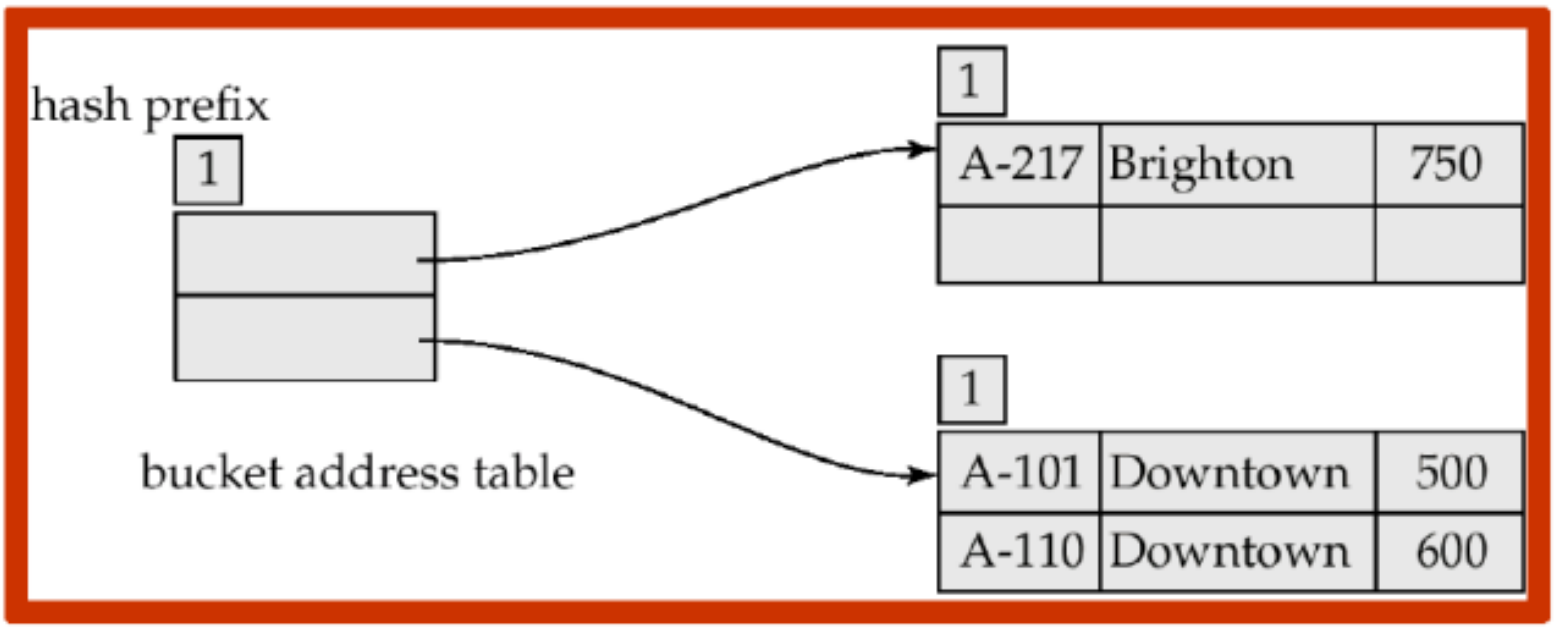
- Initial hash structure with a bucket size = 2 and bucket address table size of 1 ($i = 0$)

branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



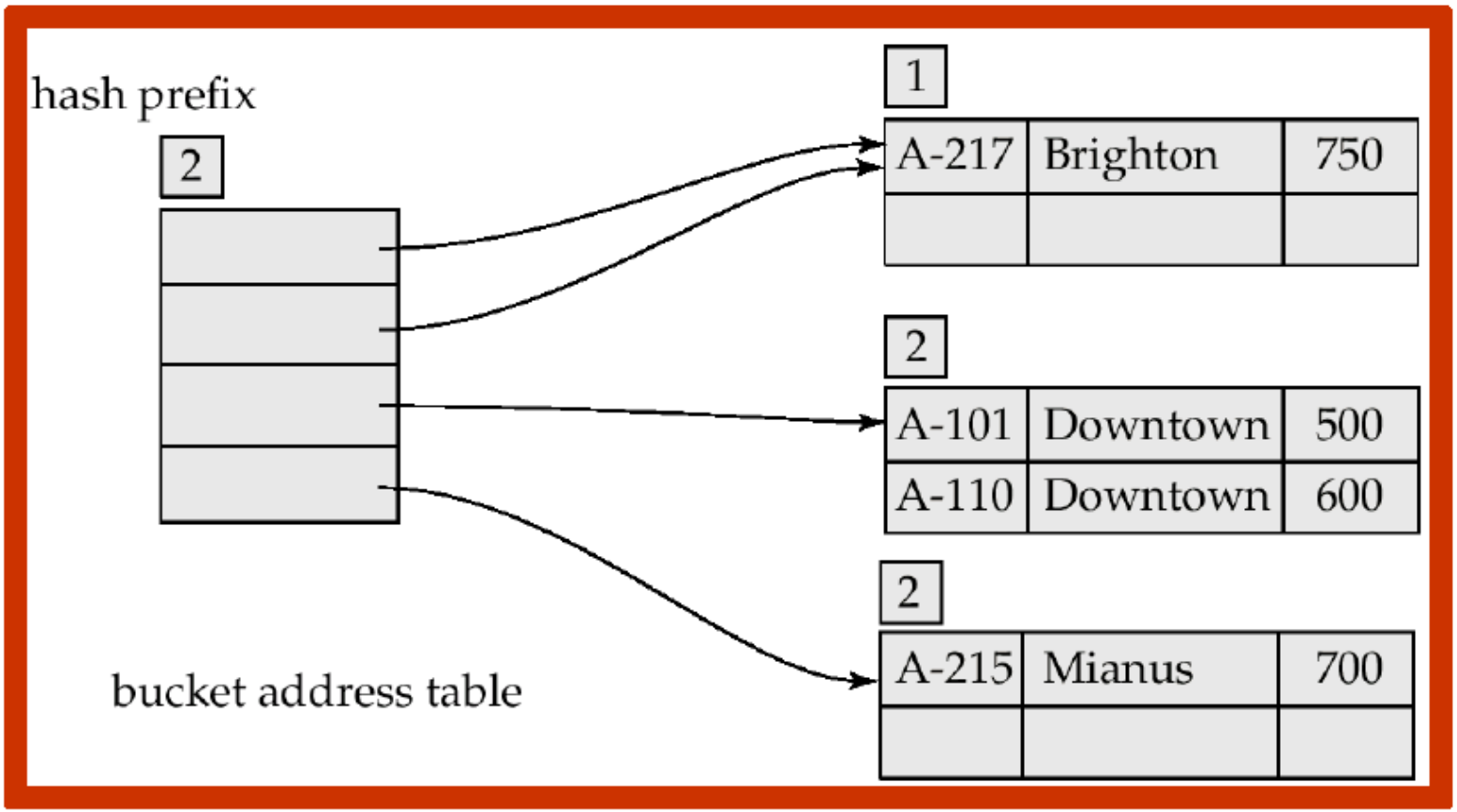
Extendable Hashing: Example . . .

- Hash structure after insertion of one "Brighton" and two "Downtown" records.



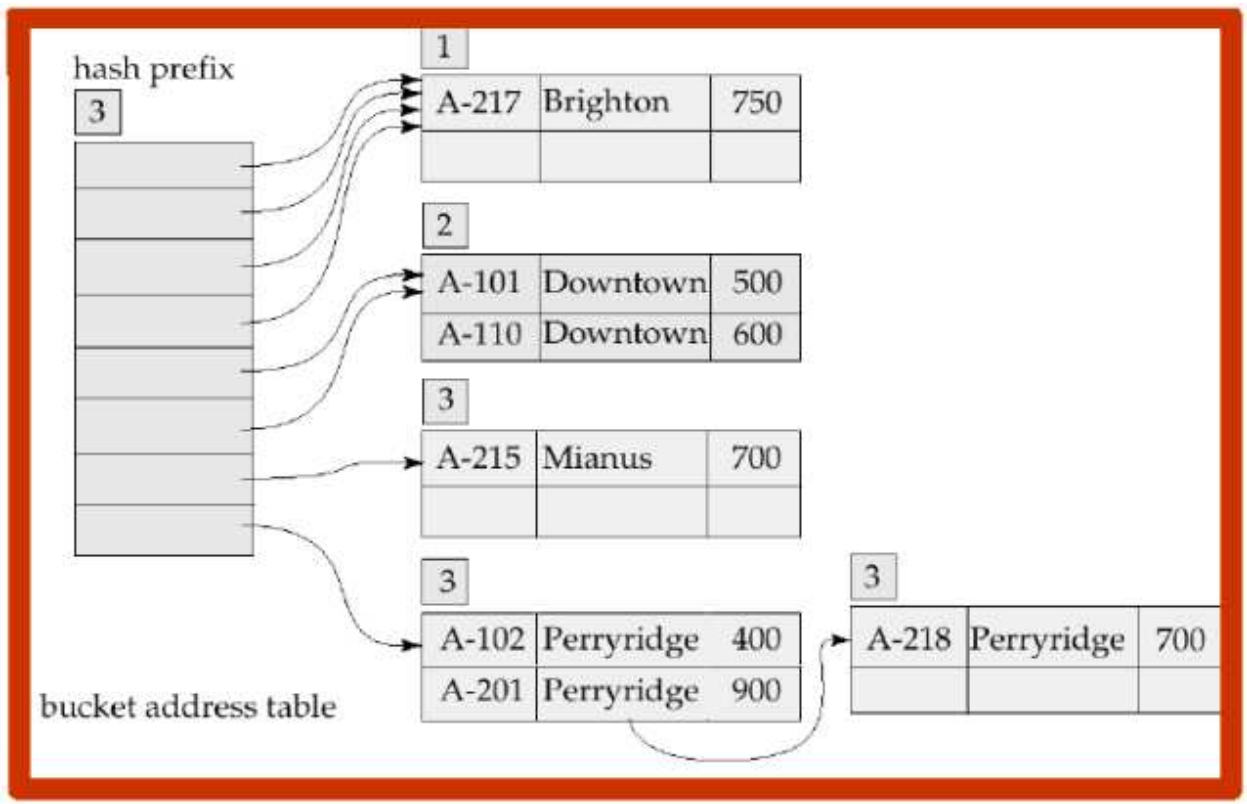
Extendable Hashing: Example ...

- Hash structure after insertion of "Mianus" record



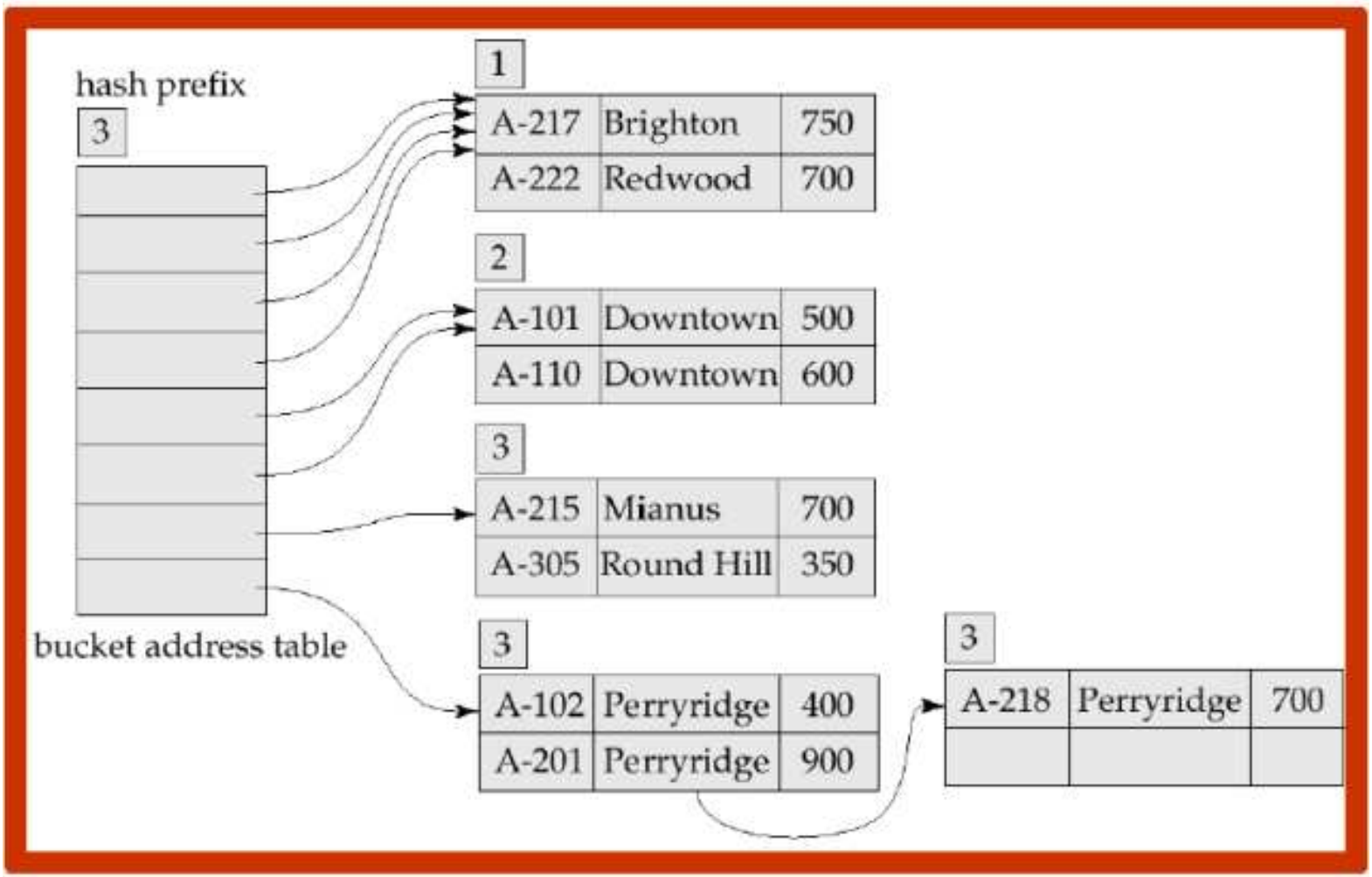
Extendable Hashing: Example ...

- Hash structure after insertion of three "Perryridge" records.
  - Cannot be handled by increasing the number of bits, since the three records have the same hash value; overflow bucket is needed.



Extendable Hashing: Example ...

- Hash structure after insertion of "Redwood" and "Round Hill" records.



Extendable Hashing: Discussion

- **Benefits** of extendable hashing
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
  - No buckets are reserved for future growth, but are allocated dynamically.
- **Disadvantages** of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Need a tree structure to locate desired record in the structure!
  - Changing size of bucket address table is expensive
- **Linear hashing** is an alternative mechanism
  - Extends the address table one slot at a time and avoids these disadvantages at the possible cost of more bucket overflows

Ordered Indexing vs. 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.
  - If range queries are common, ordered indices are to be preferred
    - There is no ordering in hash organization, and hence there is no notion of "next record in sort order".

Multiple-Key Access

- Use multiple indices for certain types of queries.
- **Example:**

```
SELECT account-number
FROM   account
WHERE  branch-name = "Perryridge" AND balance = 1000
```

- Possible strategies to process query by using **multiple single-key indices:**
  1. Use index on branch-name to find accounts with branch-name = "Perryridge"; test balances = \$1000.
  2. Use index on balance to find accounts with balance = \$1000; test branch-name = "Perryridge".
  3. Use branch-name index to find pointers to all records pertaining to the Perryridge branch. Similarly use index on balance. Take intersection of both sets of pointers obtained.

Multiple-Key Access ...

- Only the third strategy takes advantage of the existence of multiple single-key indices
- Even this strategy may be a poor choice
  - There are many records pertaining to the "Perryridge" branch
  - There are many records pertaining to accounts with a balance of $1000
  - There are only a few records pertaining to both the Perryridge branch and accounts with a balance of $1000
- More efficient access methods exist
  - (Traditional) indices on **combined search-keys**, i.e., multiple attributes
  - Special index structures that support **multiple keys**
    - e.g., grid files, bitmap index

Indices on Multiple Attributes

- Consider an ordered index on the **combined search-key** (*branch-name*, *balance*)
- Can efficiently handle

```
WHERE branch-name = "Perryridge" AND balance = 1000
```

  - The index on the combined search-key will fetch only records that satisfy both conditions
  - Using separate indices is less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.

- Can also efficiently handle

```
WHERE branch-name = "Perryridge" AND balance < 1000
```

- But **cannot** efficiently handle

```
WHERE branch-name < "Perryridge" AND balance = 1000
```

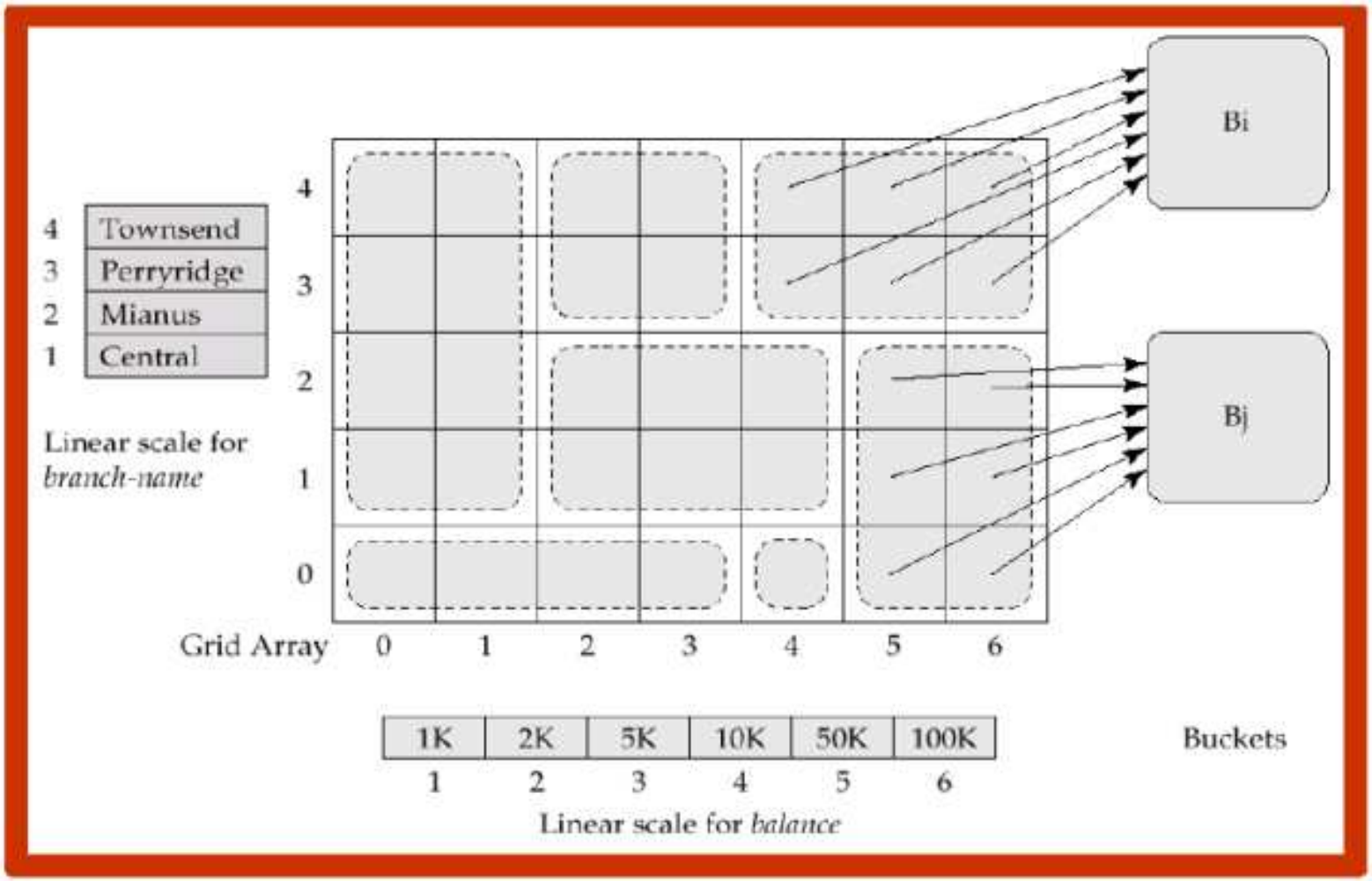
  - May fetch many records that satisfy the first but not the second condition

Grid Files

- Structure used to speed up the processing of multiple search-key queries involving one or more comparison operators.
- **Grid file**
  - Consists of a single grid array.
  - One linear scale for each search-key attribute.
  - The number of dimensions of grid array is equal to the number of search-key attributes.
  - Each cell in the grid array has a pointer to a bucket that contains the search-key values and pointers to records.
    - Multiple cells of grid array can point to same bucket

Grid Files . . .

- **Example**: Grid file for account.



Grid Files ...

- **Queries** in a grid file
  - To find the bucket for a search-key value, locate the row and column of its cell using the linear scales and follow the pointers.
- 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 $(a_1 \leq A \leq a_2 \wedge b_1 \leq \mathrm{B} \leq b_2)$, use linear scales to find corresponding candidate grid array cells, and look up all the buckets pointed to from those cells.

Grid Files ...

- 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.
- R-trees are an alternative.

Bitmap Indices

- **Bitmap index:** A special type of index designed for efficient querying on multiple keys. Simplest form:
  - A bitmap is an array of bits and has as many bits as records in the file.
  - A bitmap is required for each different attribute value v of the search-key attribute A.
  - 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.

record number	*name*	*gender*	*address*	*income-level*
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income-level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices ...

- Records in a relation are assumed to be numbered sequentially
  - 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, ...
  - income-level broken up into a small number of levels such as (0-9999, 10000-19999, 20000-50000, 50000-infinity)
- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Other forms of bitmap indices are possible.

Bitmap Indices ...

- Queries can be answered efficiently 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
- **Example:** Males with income level L1: 10010 AND 10100 = 10000
  - Tuple 1 is the only result tuple.
  - Counting number of matching tuples is even faster

Bitmap Indices ...

- Bitmap indices are in general very small compared to the size of the data relation
  - e.g., if record is 100 bytes, space for a single bitmap is 1/800 of the space used by relation
  - if then the number of distinct attribute values is 8, bitmap is only 1% of relation size
- Deletion needs to be handled properly
  - **Existence bitmap** to note if there is a valid record at a record location
  - Needed for complementation, e.g.,
    - not(A=v): (NOT bitmap-A-v) AND ExistenceBitmap
- Should keep bitmaps for all values, even null value
  - To correctly handle SQL NULL semantics for NOT(A=v):
    - Intersect the above result with (NOT bitmap-A-Null)

Bitmap Indices: Implementation

- Bitmap are packed into words; a single word computes the AND of 32 or 64 bits at once (a basic CPU instruction)
  - e.g., 1-million-bit maps can be anded with just 31,250 instructions
- Counting number of 1s can be done fast by a trick:
  - Use each byte of the bitmap to index into a precomputed array of 256 elements, each storing the count of 1s in the binary representation
    - Can use pairs of bytes to speed up further at a higher memory cost
  - Add up the retrieved counts
- Bitmaps can be used instead of pointer lists at leaf levels of B^+-tress, for values with a large number of matching records
  - List representation needs typically 64bits for each records pointer; bit representation needs 1 bit.
  - Worthwhile if $> 1/64$ of the records have that value.
  - Above technique merges benefits of bitmap and B^+-tree indices.

Index Definition in SQL

- SQL-92 does not define syntax for indices because these are not considered part of the logical data model
- All DBMSs (must) provide support for indices
- Create an index:

  `CREATE INDEX <index-name> ON <relation-name> (<attribute-list>)`

  - e.g, `CREATE INDEX b-index ON branch(branch-name)`

- **Create unique index** to indirectly specify and enforce the condition that the search key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index: `DROP INDEX <index-name>`
  - e.g, `DROP INDEX b-index`

Indices in Oracle

- B^+-tree indices in Oracle

```
CREATE [UNIQUE] INDEX <name> ON <table_name>
  "(" col [DESC] {"," col [DESC]} ")" PCTFREE n] [...]
```

  - PCTFREE specifies how many percent of a index page are left unfilled initially (dafult to 10%)
  - In index definitions UNIQUE should not be used because it is a logical concept.
  - Oracle creates a B^+-Tree index for each unique (and primary key) declaration.

- **Example**

```
CREATE TABLE book (ISBN INTEGER, Author VARCHAR2 (30), ...);
CREATE INDEX book_auth ON book(Author);
```