

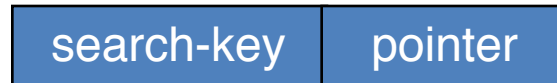
Indexing and Hashing

Overview

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

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
 - E.g., author catalog in library
- Search Key - attribute to set of attributes used to look up records in a file.
- An index file consists of records (called index entries) of the form



- 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”.

Index Evaluation Metrics

- Access types supported efficiently. E.g.,
 - records with a specified value in the attribute
 - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

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.

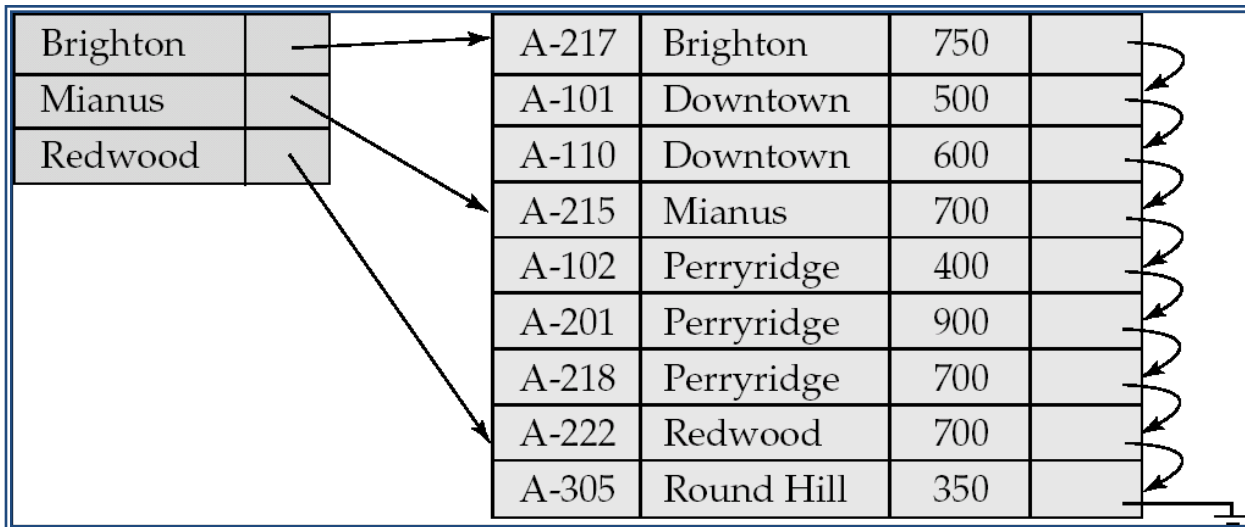
Dense Index Files

- Dense index — Index record appears for every search-key value in the file.

Brighton		→	A-217	Brighton	750		↙
Downtown		→	A-101	Downtown	500		↙
Mianus		→	A-110	Downtown	600		↙
Perryridge		→	A-215	Mianus	700		↙
Redwood		→	A-102	Perryridge	400		↙
Round Hill		→	A-201	Perryridge	900		↙
			A-218	Perryridge	700		↙
			A-222	Redwood	700		↙
			A-305	Round Hill	350		↙

Sparse Index Files

- 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 we:
 - Find index record with largest search-key value $< K$
 - Search file sequentially starting at the record to which the index record points

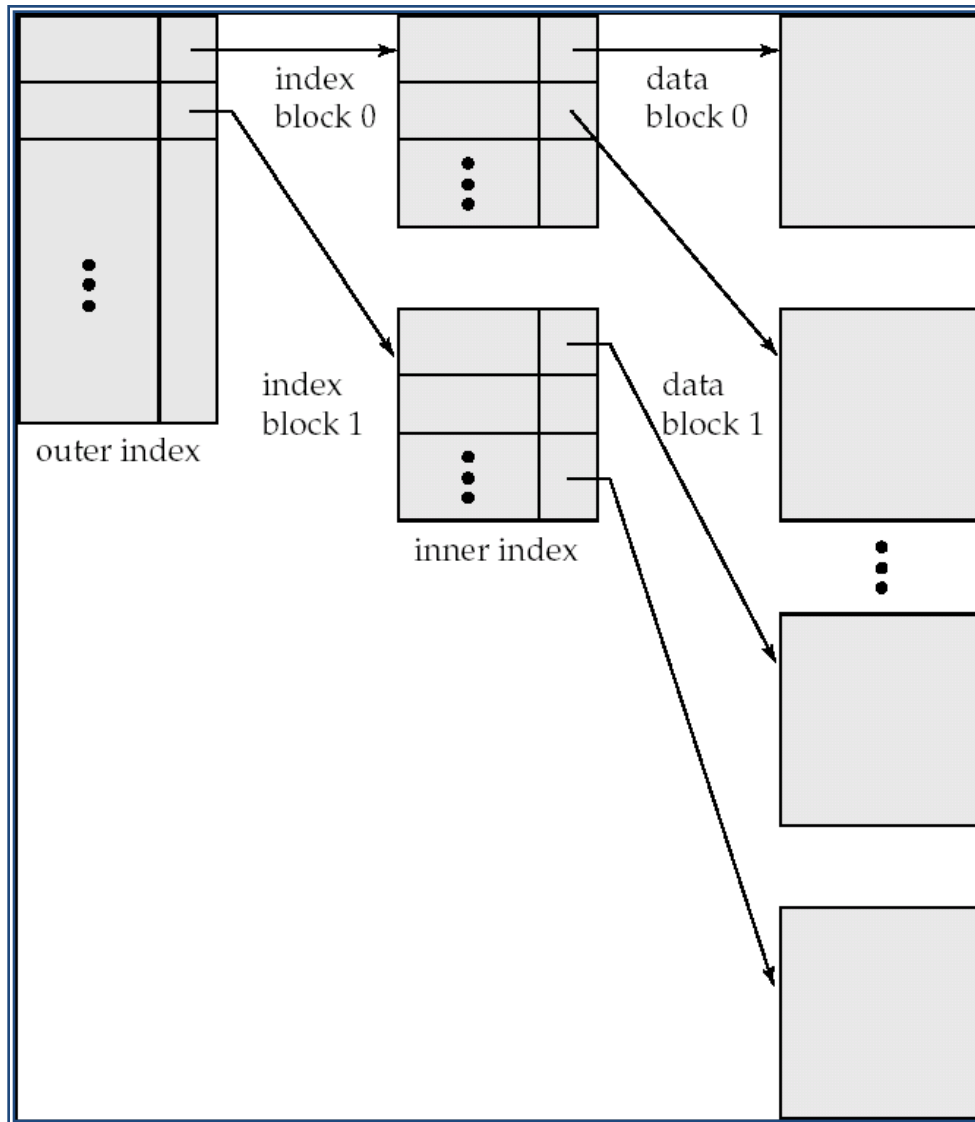


Sparse Index Files (Cont.)

- 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.

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.



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 deletion:
 - Dense indices – deletion of search-key: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).
 - If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

Index Update: Insertion

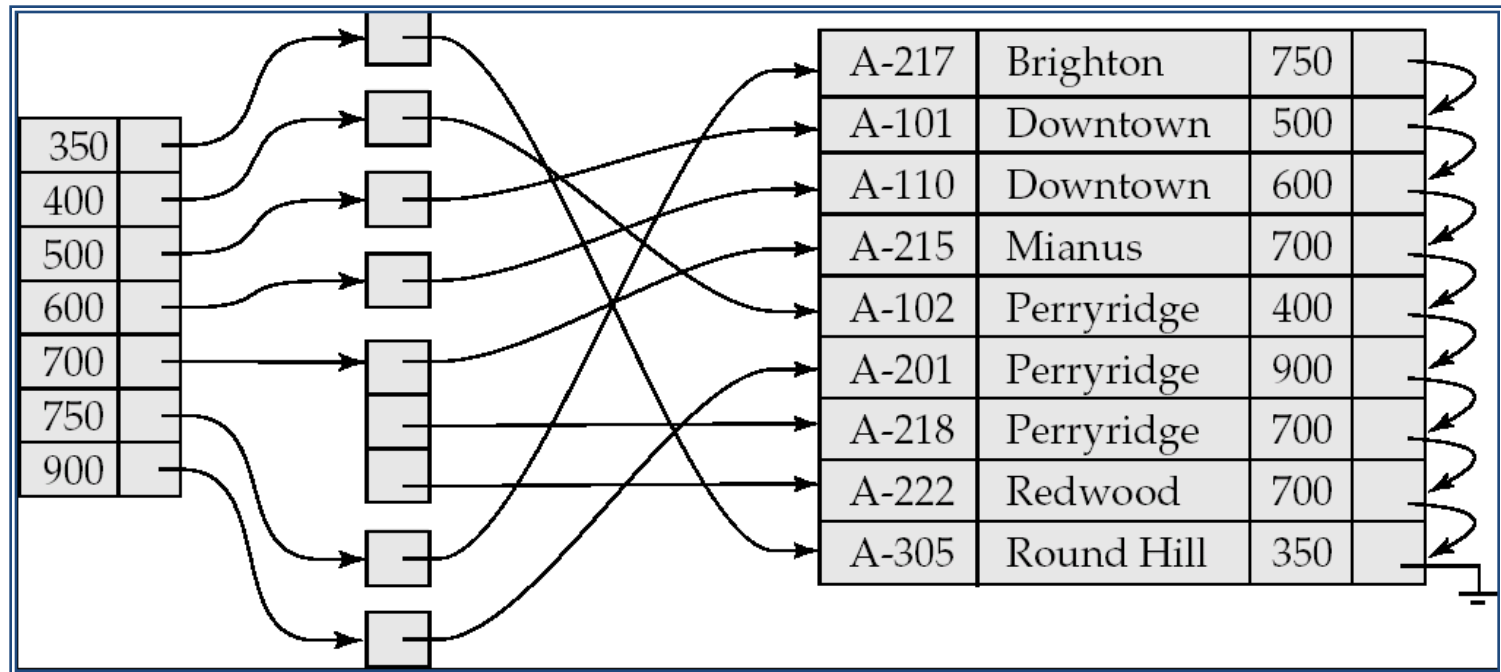
- Single-level index insertion:
 - Perform a lookup using the search-key value appearing in the record to be inserted.
 - Dense indices – if the search-key value does not appear in the index, insert it.
 - Sparse indices – if index stores an entry for each block of the file, no change needs to be made 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.
- Multilevel insertion (as well as 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 account relation stored sequentially by account number, we may want to find all accounts in a particular branch
 - Example 2: as above, but where we want to find all accounts with a specified balance or range of balances
- We can have a secondary index with an index record for each search-key value

Secondary Indices Example

- 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



Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.
- BUT: Updating indices imposes overhead on database modification --when a file is modified, every index on the file must be updated,
- 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
 - Block fetch requires about 5 to 10 milliseconds
 - versus about 100 nanoseconds for memory access

B+-Tree Index Files

B+-tree indices are an alternative to indexed-sequential files.

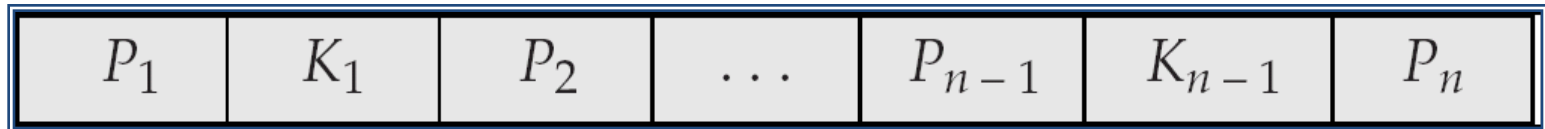
- Disadvantage of indexed-sequential files
 - performance degrades as file grows, since many overflow blocks get created.
 - Periodic reorganization of entire file is required.
- Advantage of B+-tree index files:
 - automatically reorganizes itself with small, local, changes, in the face of insertions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

B+-Tree Index Files (Cont.)

- A B+-tree is a rooted tree satisfying the following properties:
 - All paths from root to leaf are of the same length
 - Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
 - 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.

B+-Tree Node Structure

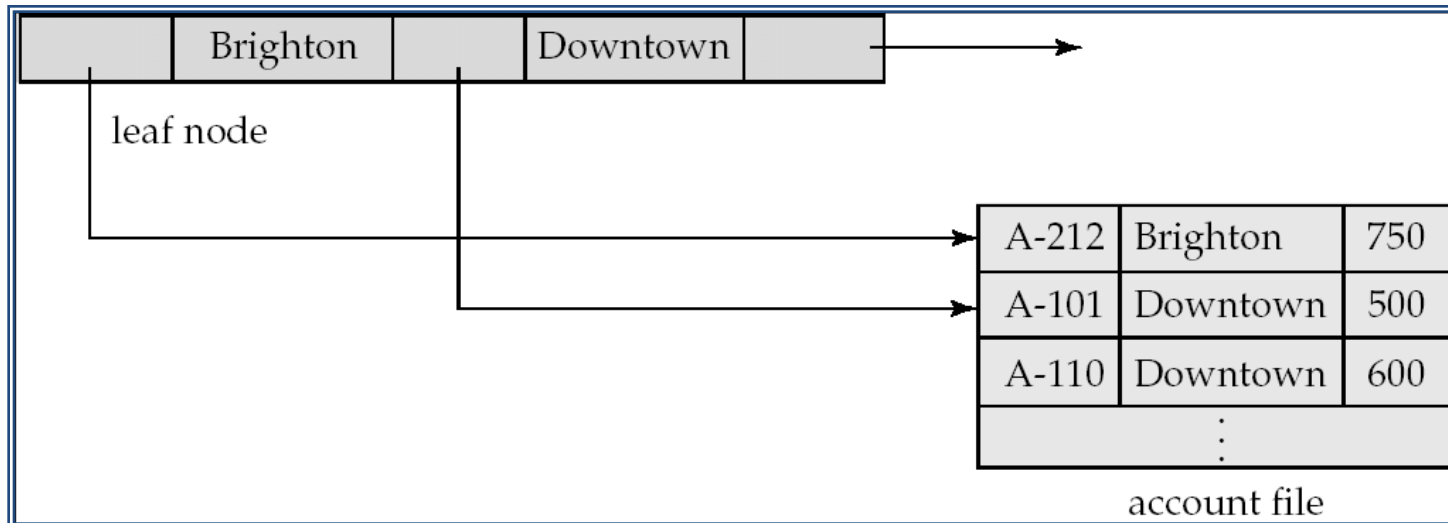
- Typical node



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

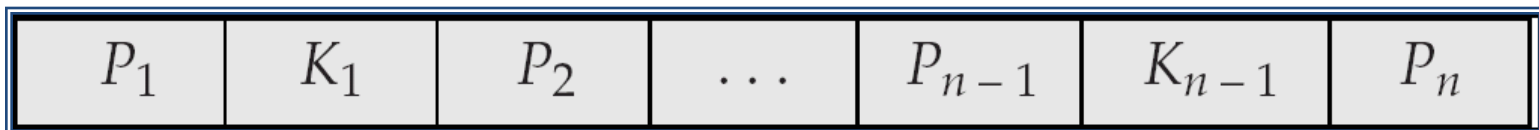
Leaf Nodes in B+-Trees

- For $i = 1, 2, \dots, n-1$, 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 L_j 's search-key values
- P_n points to next leaf node in search-key order

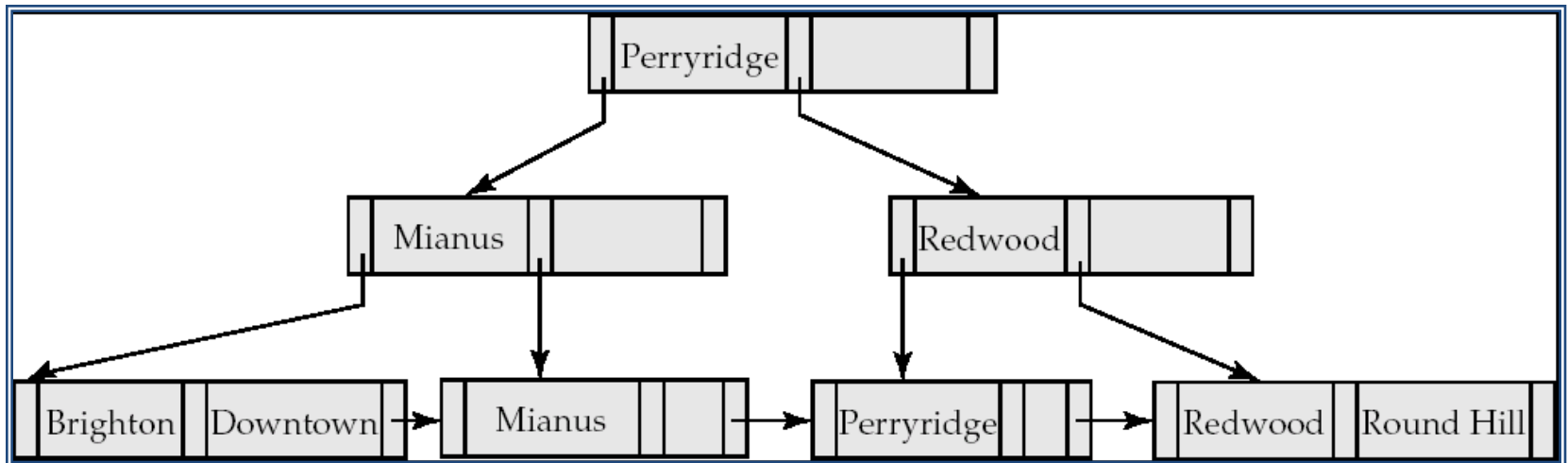


Non-Leaf Nodes in B+-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:
 - All the 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}

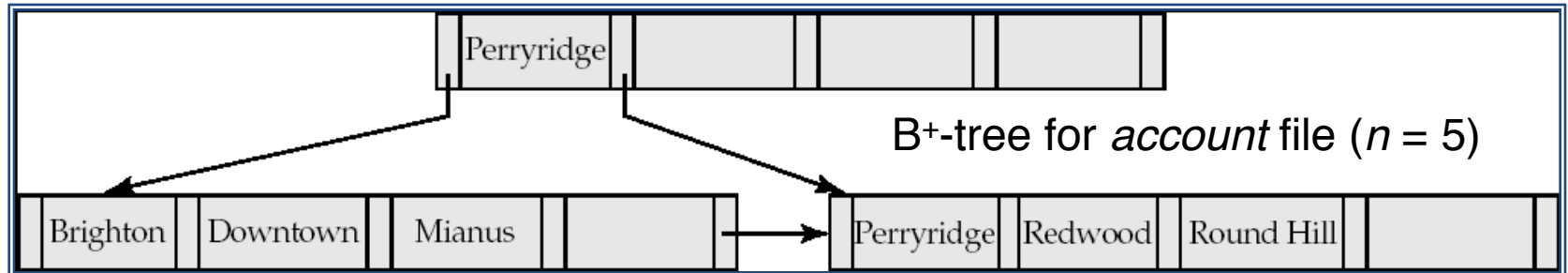


Example of a B+-tree



B+-tree for *account* file ($n = 3$)

Example of B+-tree



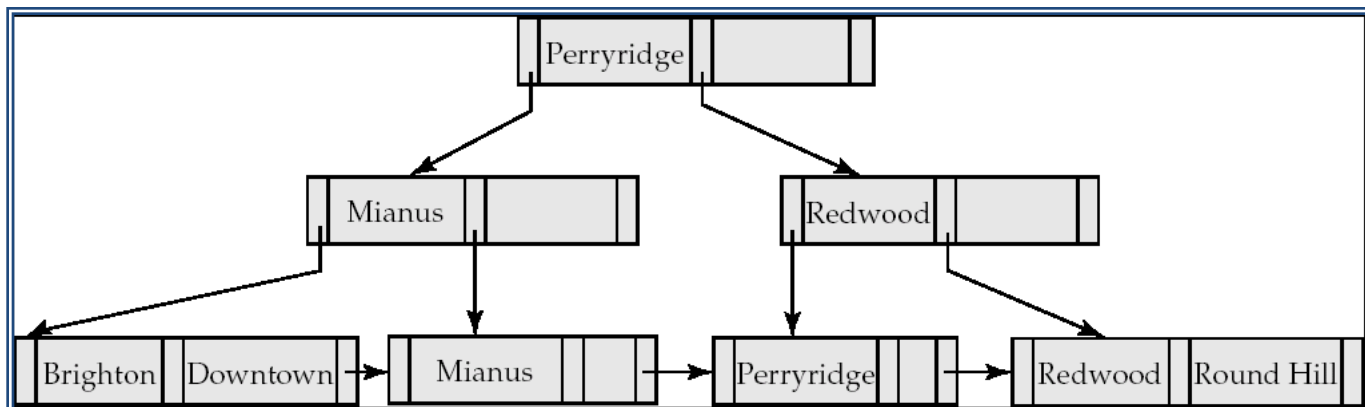
- Leaf nodes must have between 2 and 4 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 5$).
- Non-leaf nodes other than root must have between 3 and 5 children ($\lceil n/2 \rceil$ and n with $n = 5$).
- Root must have at least 2 children.

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.
- 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
 - 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

- Find all records with a search-key value of k .
 - $N = \text{root}$
 - Repeat
 - Examine N for the smallest search-key value $> k$.
 - If such a value exists, assume it is K_i . Then set $N = P_i$
 - Otherwise $k \geq K_{n-1}$. Set $N = P_n$
 - Until N is a leaf node
 - If for some i , key $K_i = k$ follow pointer P_i to the desired record or bucket.
 - Else no record with search-key value k exists.



Queries on B+-Trees (Cont.)

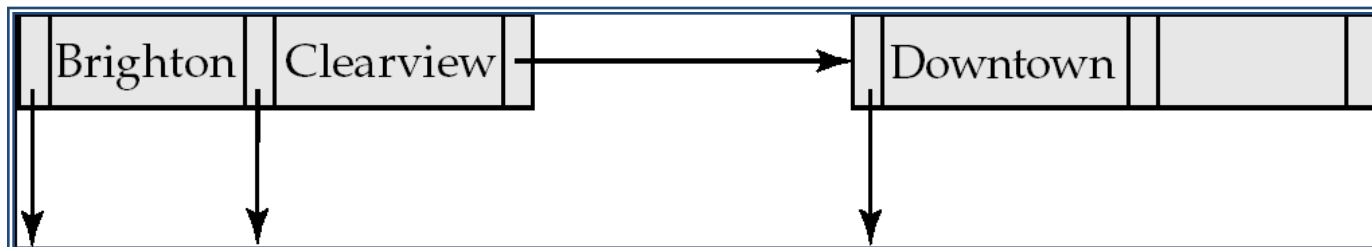
- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
 - and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values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

Updates on B+-Trees: Insertion

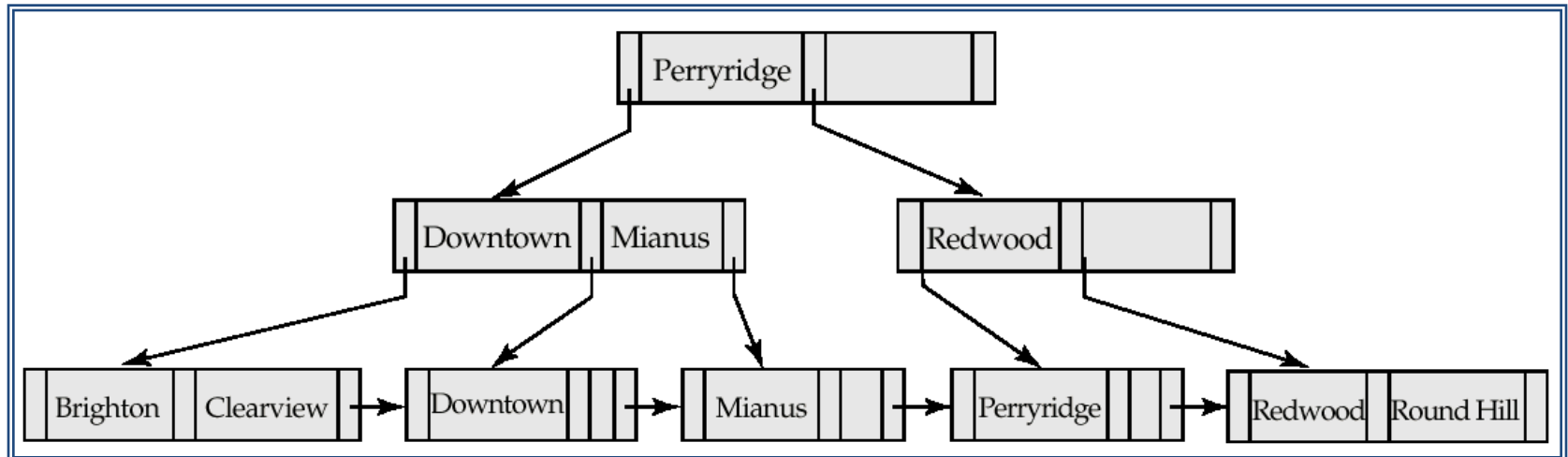
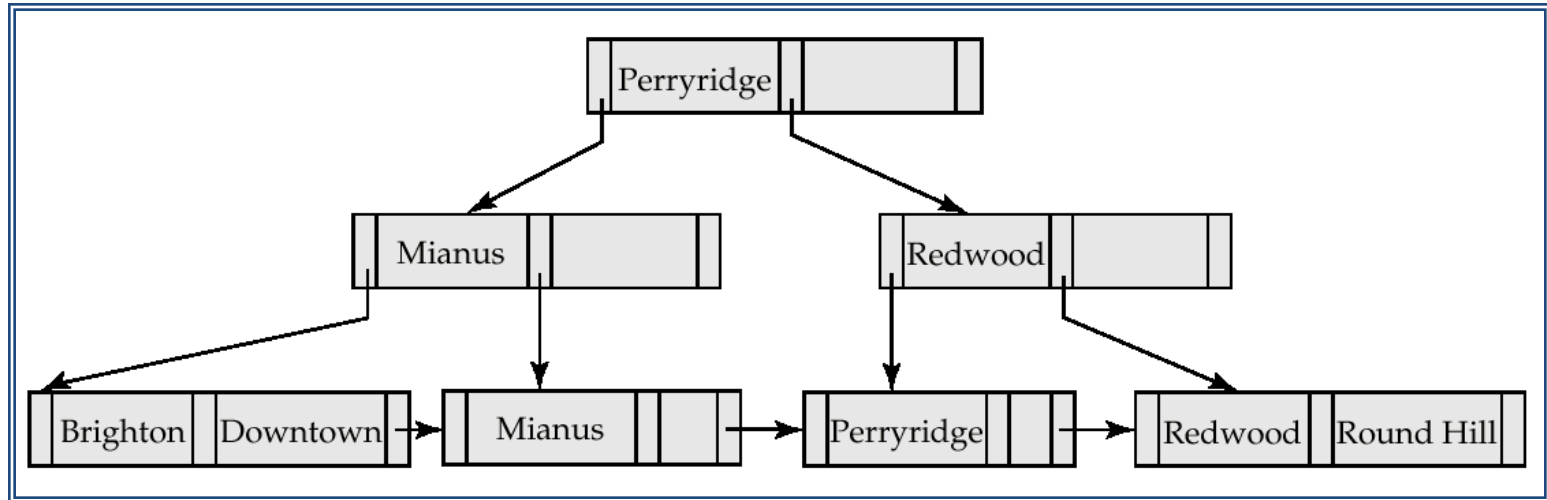
- Find the leaf node in which the search-key value would appear
- If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
- Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide.

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

- 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.



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



B+-Tree before and after insertion of "Clearview"

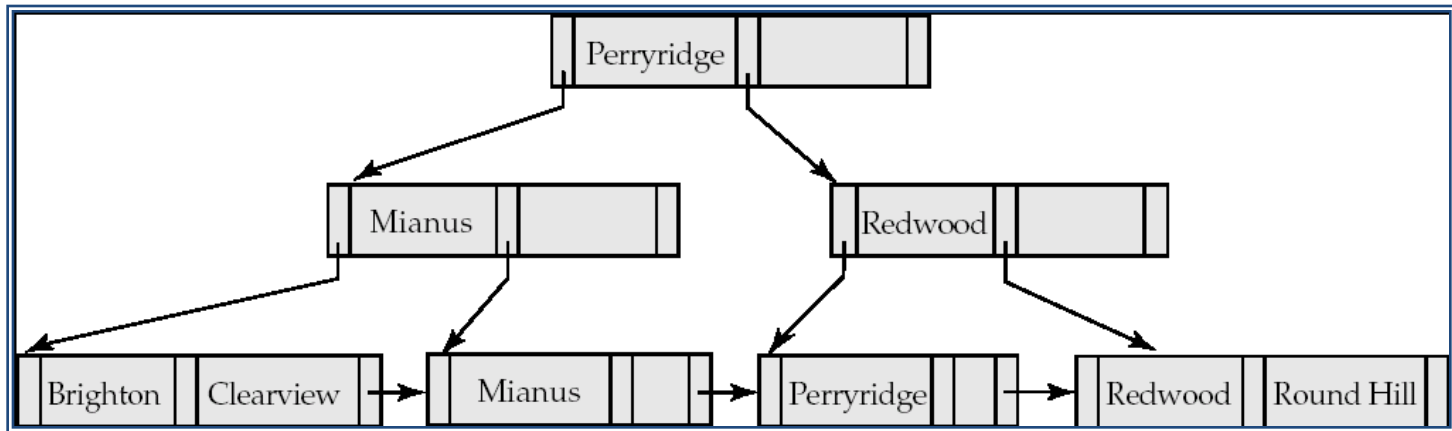
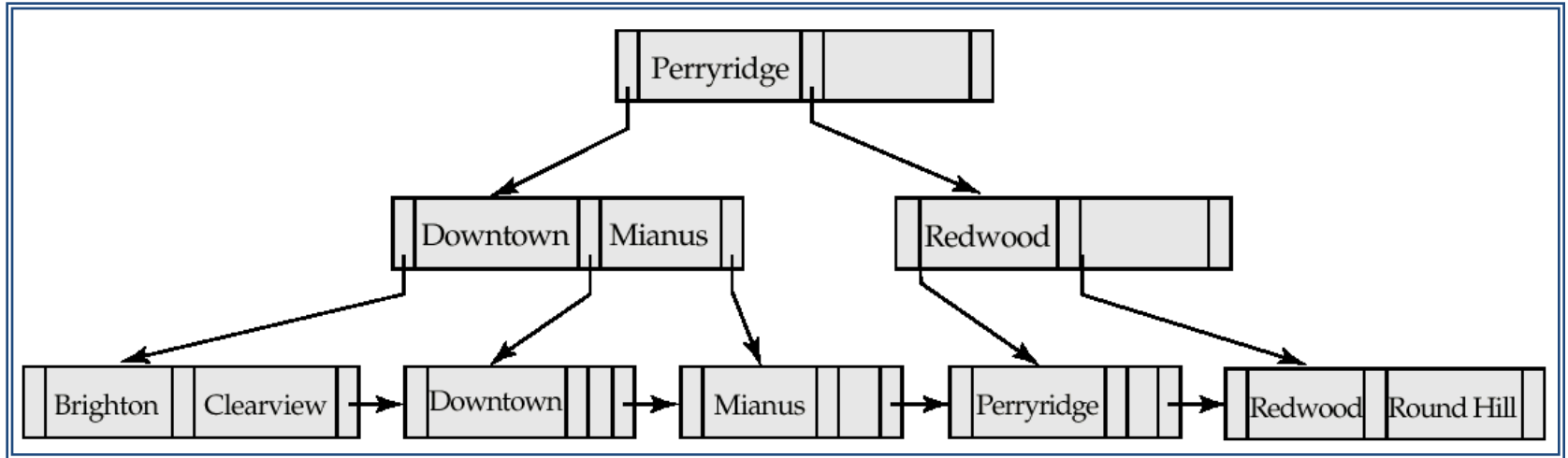
Updates on B+-Trees: Deletion

- 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

Updates on B+-Trees: Deletion

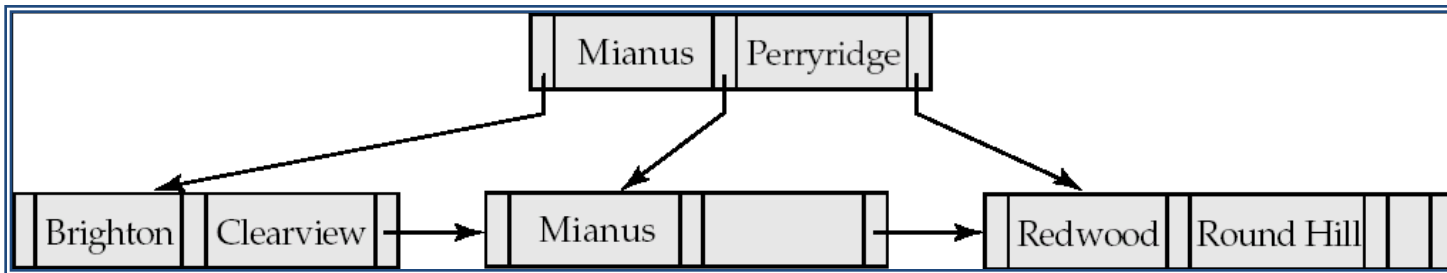
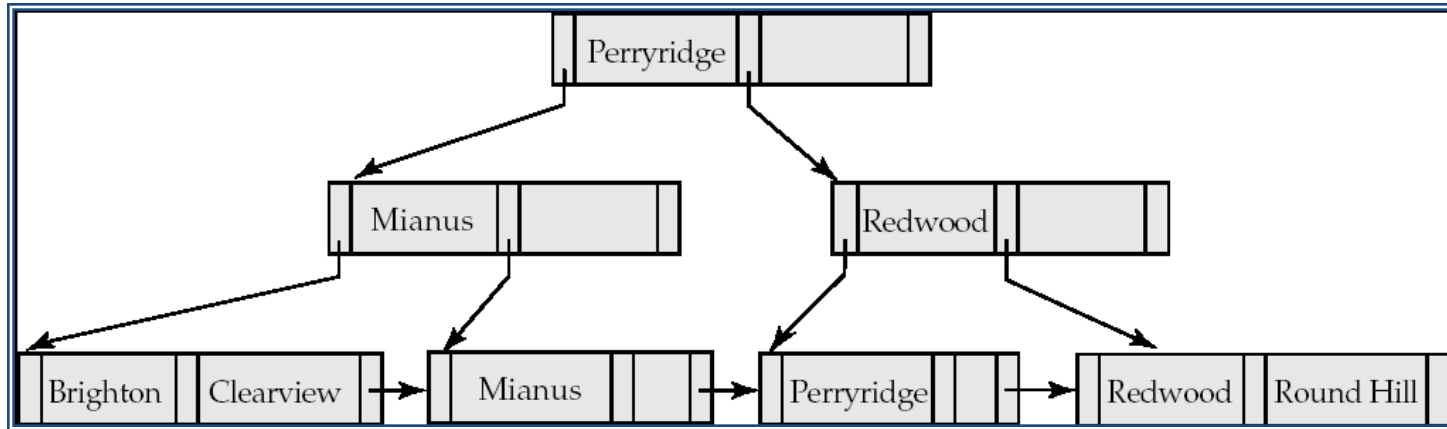
- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then redistribute pointers:
 - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
 - Update the 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.

Examples of B+-Tree Deletion



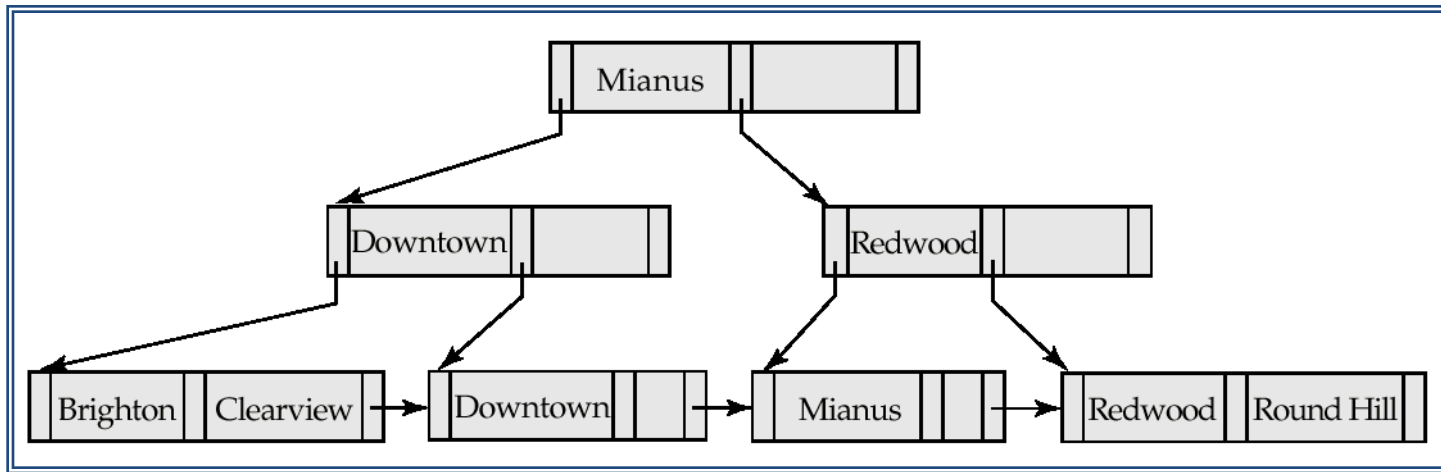
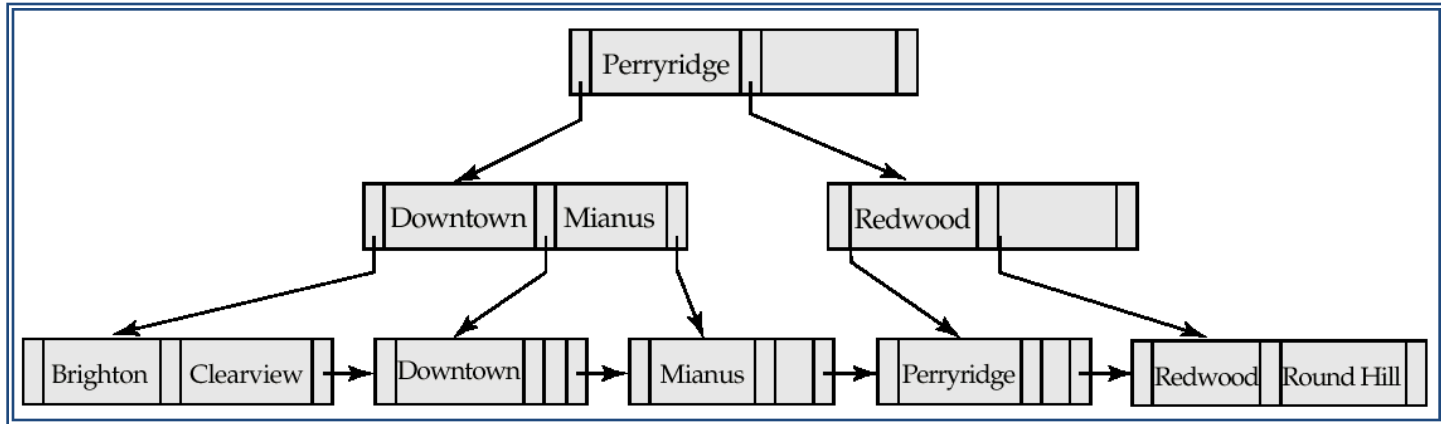
Before and after deleting "Downtown"

Examples of B+-Tree Deletion



Deletion of "Perryridge" from result of previous example

Example of B+-tree Deletion



Before and after deletion of "Perryridge" from earlier example

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 for processing query using indices on single attributes:
 1. Use index on branch_name to find accounts with branch name Perryridge; test balance = 1000
 2. Use index on balance to find accounts with balances of \$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.

Indices on Multiple Keys

- Composite search keys are search keys containing more than one attribute
 - E.g. (branch_name, balance)
- Lexicographic ordering: $(a1, a2) < (b1, b2)$ if either
 - $a1 < b1$, or
 - $a1=b1$ and $a2 < b2$
- Can also efficiently handle
 - where branch_name = “Perryridge” and balance < 1000
- But cannot efficiently handle
 - where branch_name < “Perryridge” and balance = 1000

Non-Unique Search Keys

- Alternatives:
 - Make search key unique by adding a record-identifier
 - Extra storage overhead for keys
 - Simpler code for insertion/deletion
 - Widely used

Other Issues in Indexing

- Covering indices
 - Add extra attributes to index so (some) queries can avoid fetching the actual records
 - Particularly useful for secondary indices
 - Can store extra attributes only at leaf
- Record relocation and secondary indices
 - If a record moves, all secondary indices that store record pointers have to be updated
 - Node splits in B+-tree file organizations become very expensive
 - Solution: use primary-index search key instead of record pointer in secondary index
 - Extra traversal of primary index to locate record
 - Higher cost for queries, but node splits are cheap
 - Add record-id if primary-index search key is non-unique

Hashing

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.

Example of Hash File Organization

- Hash file organization of *account* file, using *branch_name* as key
 - There are 10 buckets,
 - E.g. $h(\text{Perryridge}) = 5$
 $h(\text{Round Hill}) = 3$
 $h(\text{Brighton}) = 3$

bucket 0			bucket 5		
			A-102	Perryridge	400
			A-201	Perryridge	900
			A-218	Perryridge	700
bucket 1			bucket 6		
bucket 2			bucket 7		
			A-215	Mianus	700
bucket 3			bucket 8		
A-217	Brighton	750	A-101	Downtown	500
A-305	Round Hill	350	A-110	Downtown	600
bucket 4			bucket 9		
A-222	Redwood	700			

Hash Functions

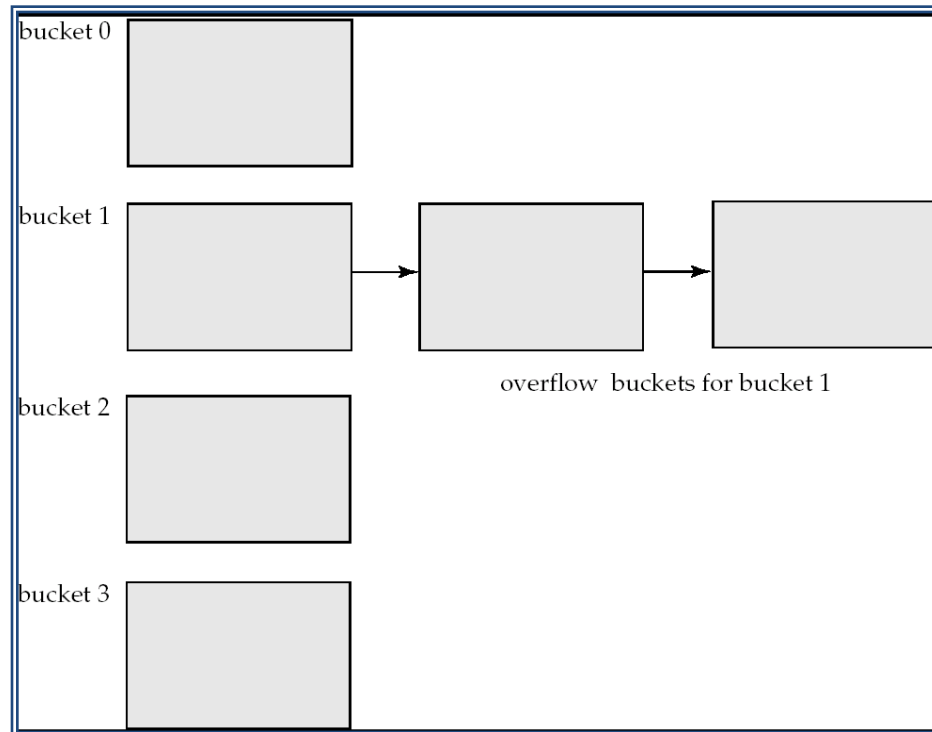
- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is uniform, i.e., each bucket is assigned the same number of search-key values from the set of all possible values.
- Ideal 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.

Handling of Bucket Overflows

- Bucket overflow can occur because of
 - Insufficient buckets
 - Skew in distribution of records. This can occur due to two reasons:
 - 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.

Handling of Bucket Overflows (Cont.)

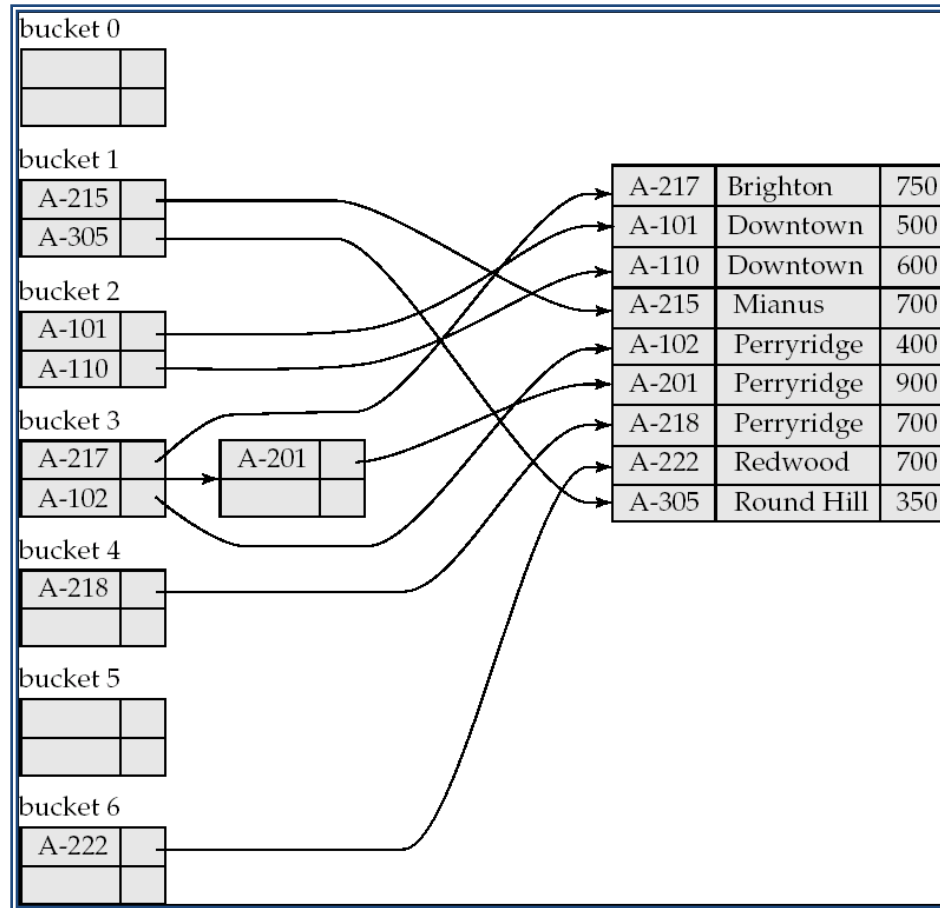
- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.



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, 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.

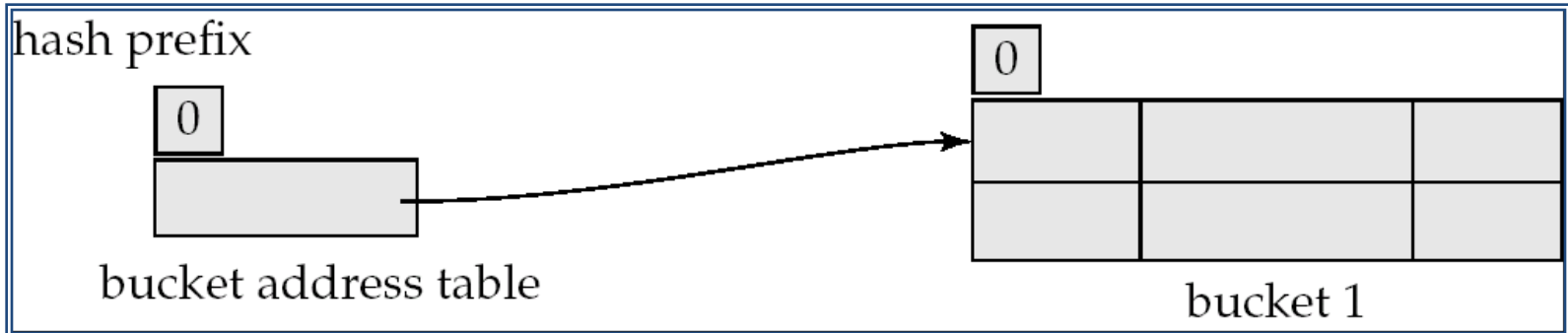
Example of Hash Index



Deficiencies of Static Hashing

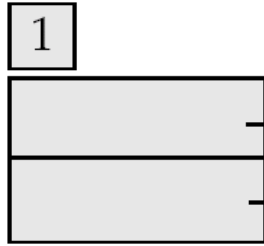
- In static hashing, function h maps search-key values to a fixed set of B of bucket addresses. Databases grow or shrink with time.
 - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
 - If space is allocated for anticipated growth, a significant 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.

<i>branch_name</i>	$h(\text{branch_name})$
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



Initial Hash structure, bucket size = 2

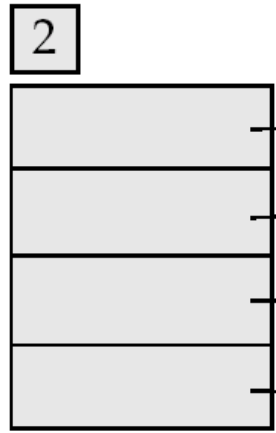
hash prefix



bucket address table

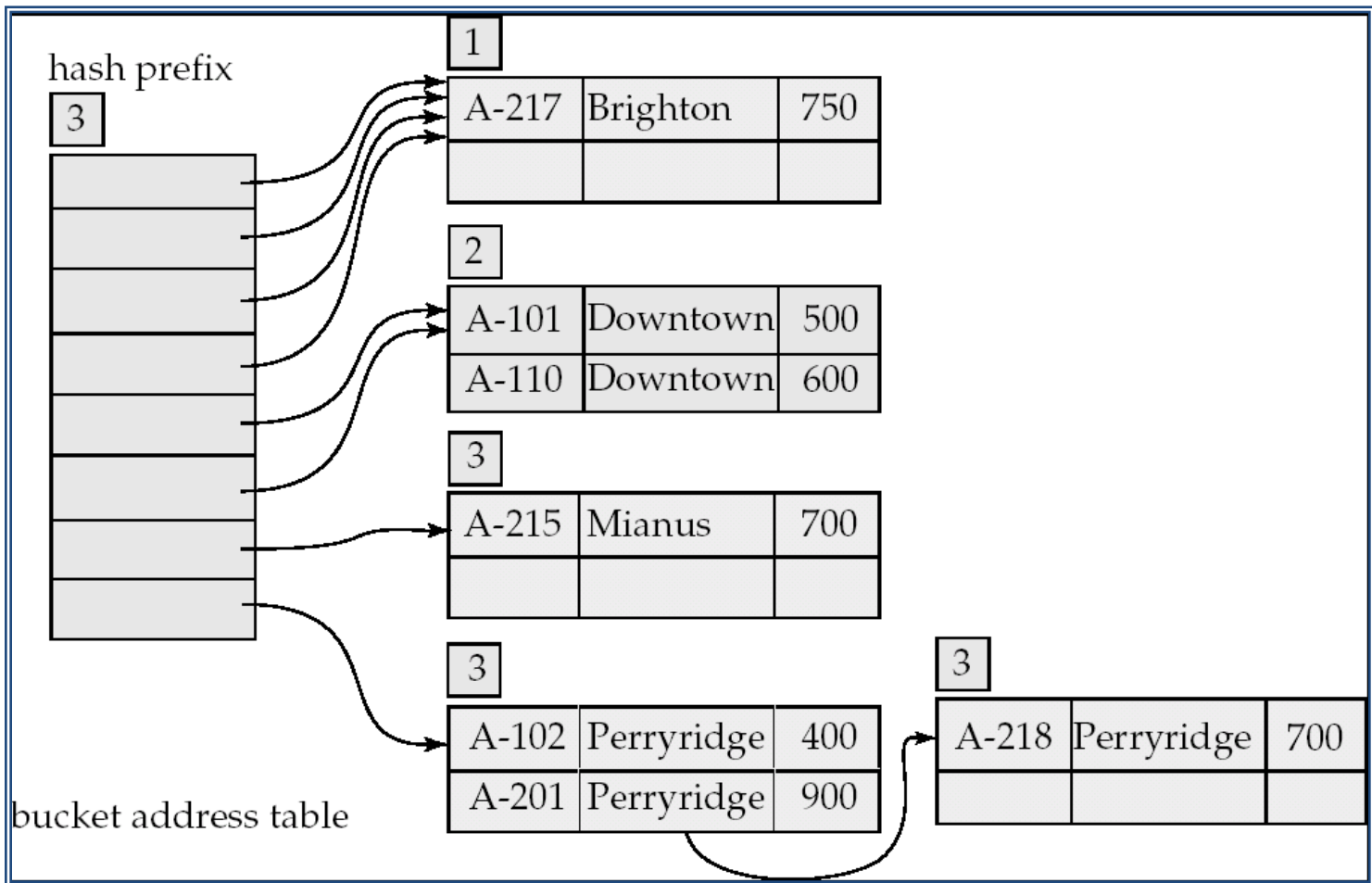
1		
A-217	Brighton	750
1		
A-101	Downtown	500
A-110	Downtown	600

hash prefix



bucket address table

1		
A-217	Brighton	750
2		
A-101	Downtown	500
A-110	Downtown	600
2		
A-215	Mianus	700



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

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.
 - If range queries are common, ordered indices are to be preferred
- In practice:
 - PostgreSQL supports hash indices, but discourages use due to poor performance
 - Oracle supports static hash organization, but not hash indices
 - SQLServer supports only B+-trees

Bitmap Indices

- 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 (Cont.)

- 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

record number	<i>name</i>	<i>gender</i>	<i>address</i>	<i>income_level</i>	Bitmaps for <i>gender</i>		Bitmaps for <i>income_level</i>	
					m	10010		
					f	01101	L1	10100
0	John	m	Perryridge	L1			L2	01000
1	Diana	f	Brooklyn	L2			L3	00001
2	Mary	f	Jonestown	L1			L4	00010
3	Peter	m	Brooklyn	L4			L5	00000
4	Kathy	f	Perryridge	L3				

Bitmap Indices (Cont.)

- 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. $100110 \text{ AND } 110011 = 100010$
 - $100110 \text{ OR } 110011 = 110111$
 - $\text{NOT } 100110 = 011001$
 - Males with income level L1: $10010 \text{ AND } 10100 = 10000$
 - Can then retrieve required tuples.
 - Counting number of matching tuples is even faster

Bitmap Indices (Cont.)

- Bitmap indices generally very small compared with relation size
 - E.g. if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.
 - If 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
 - $\text{not}(A=v)$: $(\text{NOT bitmap-}A\text{-}v) \text{ AND ExistenceBitmap}$
- Should keep bitmaps for all values, even null value
 - To correctly handle SQL null semantics for $\text{NOT}(A=v)$:
 - intersect above result with $(\text{NOT bitmap-}A\text{-Null})$

Efficient Implementation of Bitmap Operations

- Bitmaps are packed into words; a single word and (a basic CPU instruction) computes and of 32 or 64 bits at once
 - E.g. 1-million-bit maps can be and-ed with just 31,250 instruction
- Counting number of 1s can be done fast by a trick:
 - Use each byte 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 Tuple-ID lists at leaf levels of B+-trees, for values that have a large number of matching records
 - Worthwhile if $> 1/64$ of the records have that value, assuming a tuple-id is 64 bits
 - Above technique merges benefits of bitmap and B+-tree indices

Index Definition in SQL

- Create an index
 - create index <index-name> on <relation-name>
(<attribute-list>)
 - E.g.: create index b-index on branch(branch_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.

End of Chapter