

Chapter 12: Indexing and Hashing (Cnt.)

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

Queries on B^+ -Trees



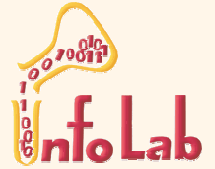
- ❖ Find all records with a search-key value of k .
 1. Start with the root node
 1. Examine the node for the smallest search-key value $> k$.
 2. If such a value exists, assume it is K_j . Then follow P_j to the child node
 3. Otherwise $k \geq K_{m-1}$, where there are m pointers in the node. Then follow P_m to the child node.
 2. If the node reached by following the pointer above is not a leaf node, repeat the above procedure on the node, and follow the corresponding pointer.
 3. Eventually reach 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.)



- ❖ 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 path is no longer 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 the search-key value is already there in the leaf node, record is added to file and if necessary a pointer is inserted into the bucket.
- ❖ If the search-key value is not there, then add the record to the main file and create a bucket if necessary. Then:
 - 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.

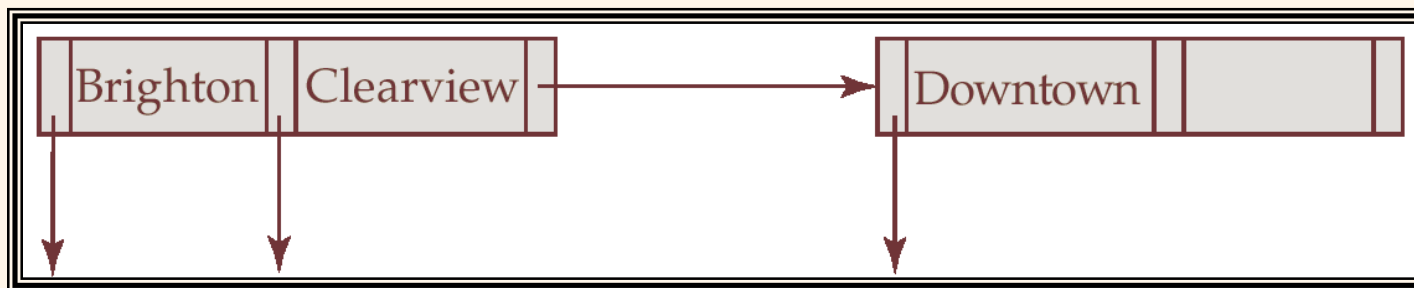
Updates on B^+ -Trees: Insertion (Cont)



❖ Splitting a node:

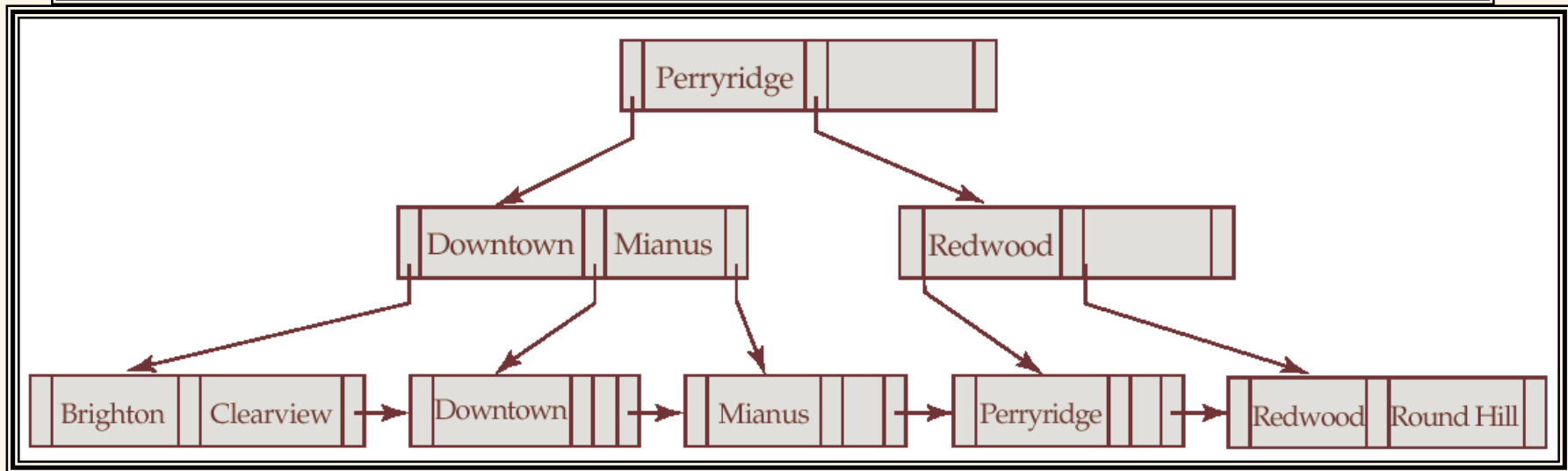
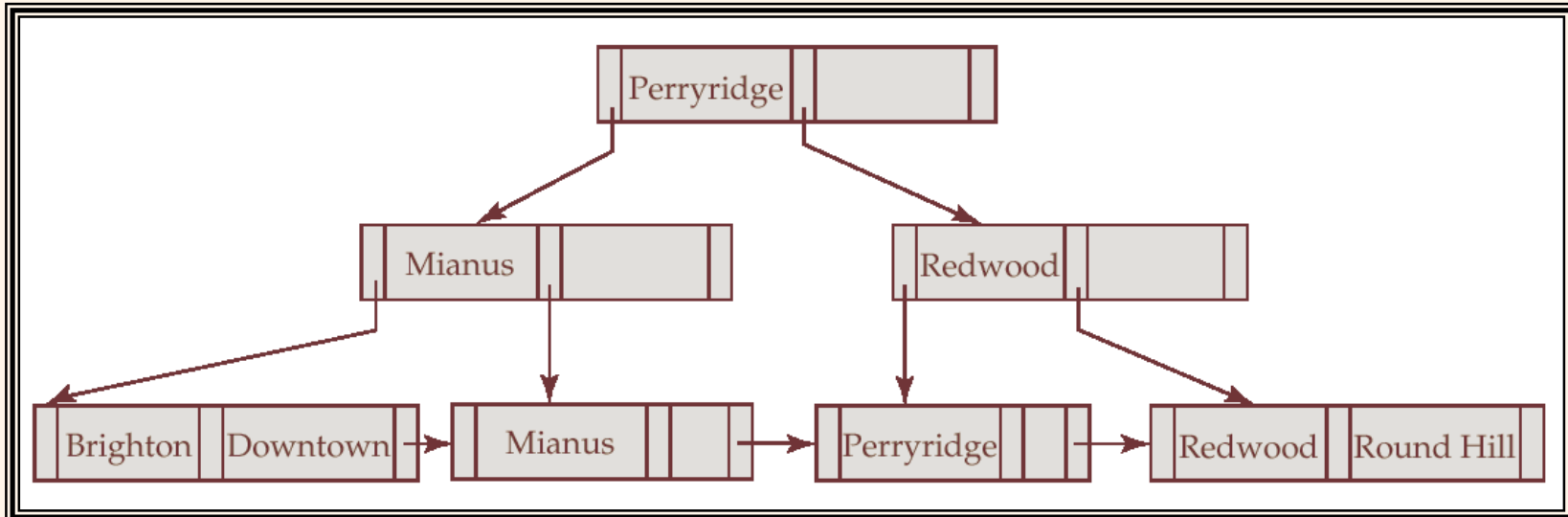
- take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
- let the new node be p , and let k be the least key value in p . Insert (k,p) in the parent of the node being split.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

- ## ❖ The 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.



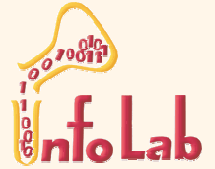
Result of splitting node containing Brighton and Downtown on inserting Clearview

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



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

Updates on B^+ -Trees: Deletion

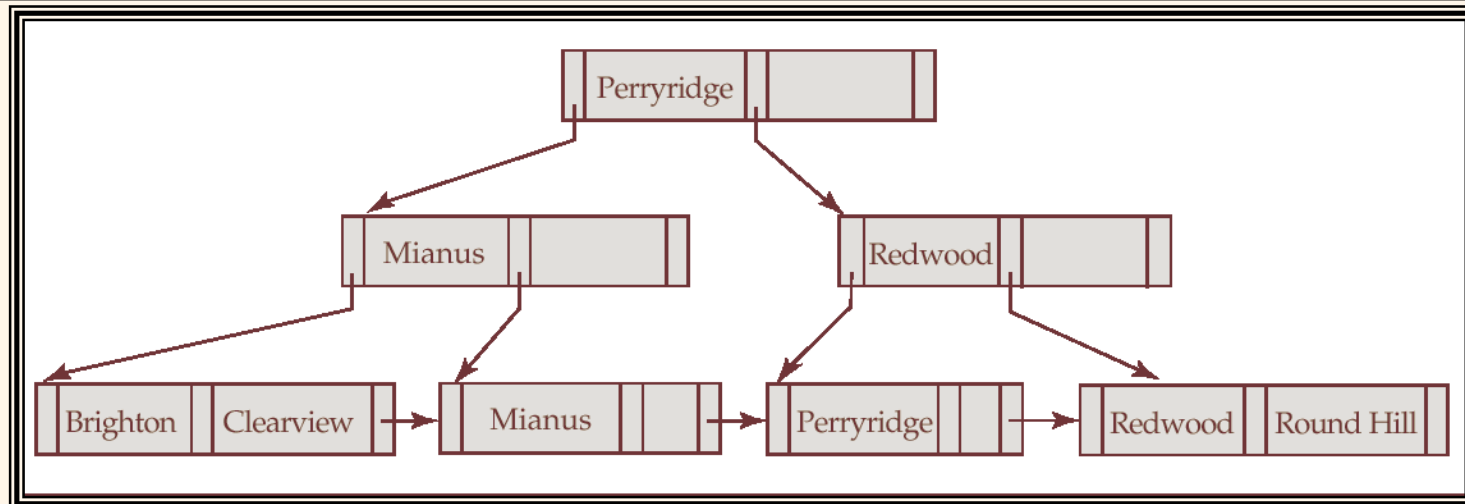
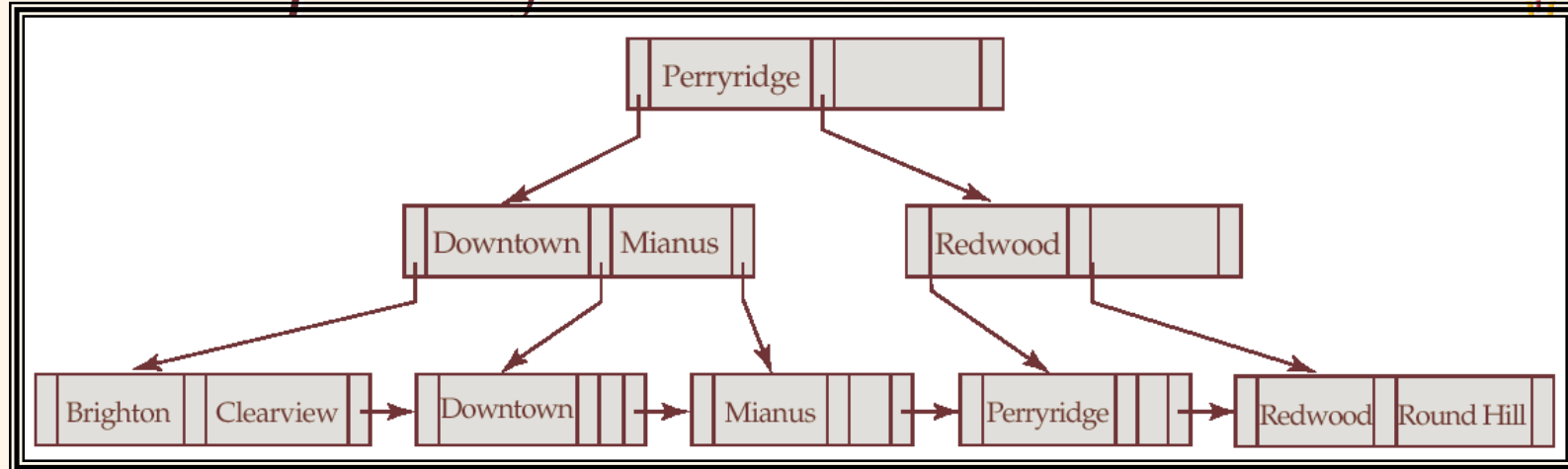


- ❖ Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- ❖ Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- ❖ If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then
 - 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.

Updates on B^+ -Trees: Deletion

- ❖ Otherwise, if the node has too few entries due to the removal, and the entries in the node and a sibling does not fit into a single node, then
 - 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.

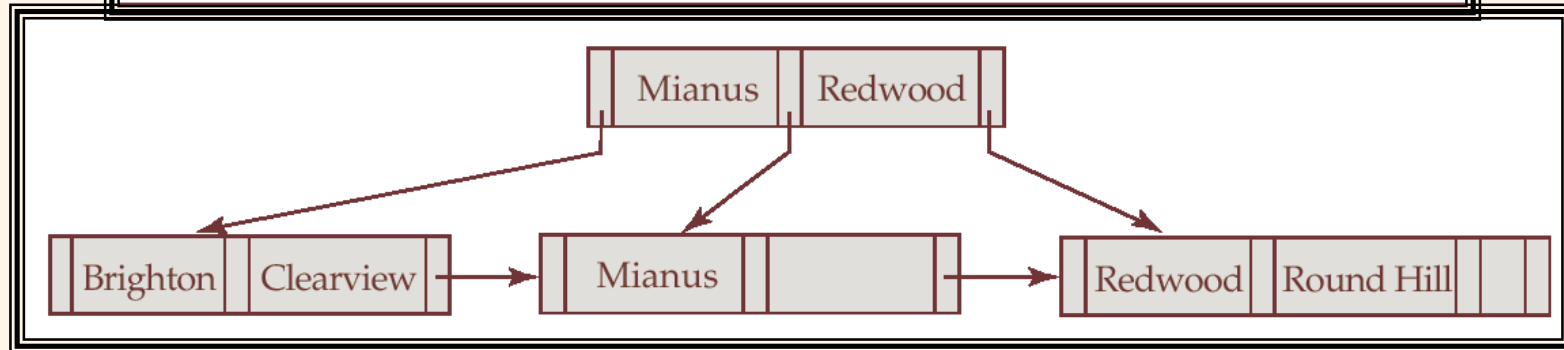
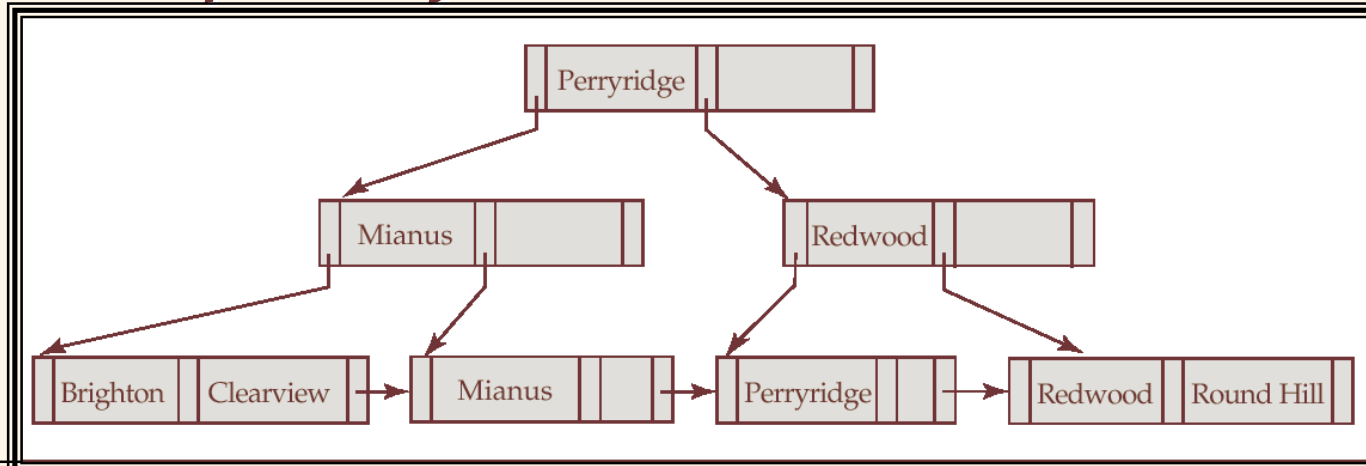
Examples of B⁺-Tree Deletion



Before and after deleting "Downtown"

- ❖ 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 node's parent.

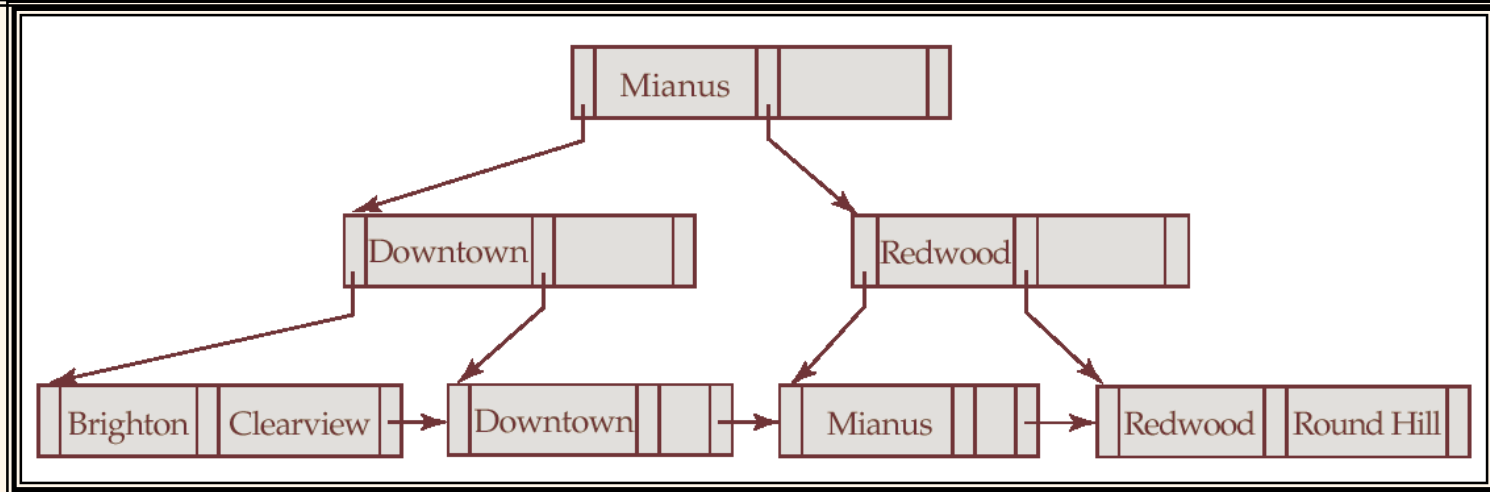
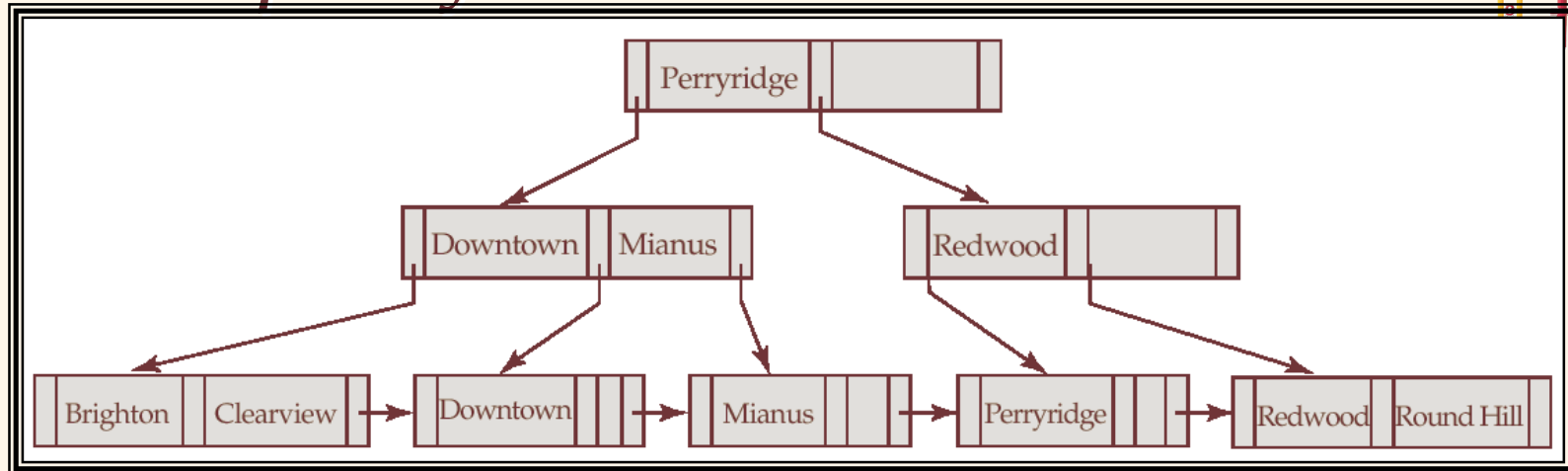
Examples of B⁺-Tree Deletion (Cont.)



Deletion of "Perryridge" from result of previous example

- ❖ Node with "Perryridge" becomes underfull (actually empty, in this special case) and merged with its sibling.
- ❖ As a result "Perryridge" node's parent became underfull, and was merged with its sibling (and an entry was deleted from their parent)
- ❖ Root node then had only one child, and was deleted and its child became the new root node

Example of B⁺-tree Deletion (Cont.)



Before and after deletion of “Perryridge” from earlier example

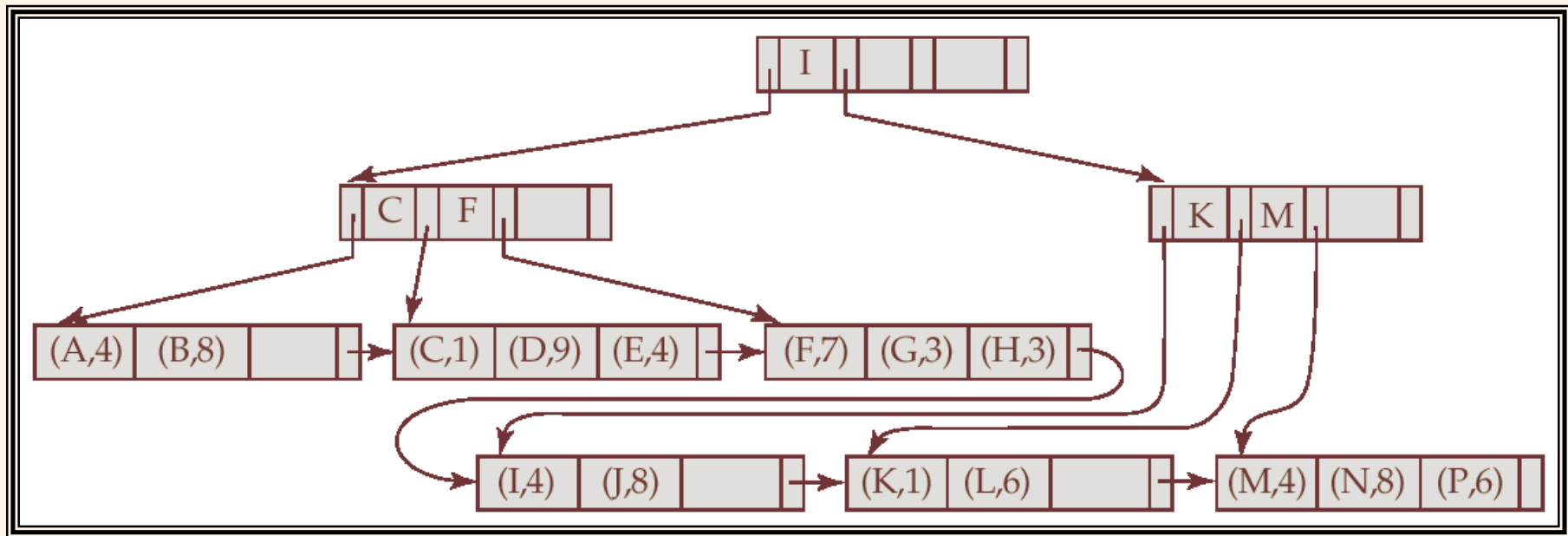
- ❖ Parent of leaf containing Perryridge became underfull, and borrowed a pointer from its left sibling
- ❖ Search-key value in the parent’s parent changes as a result

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.
- ❖ 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 nonleaf 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 (Cont.)

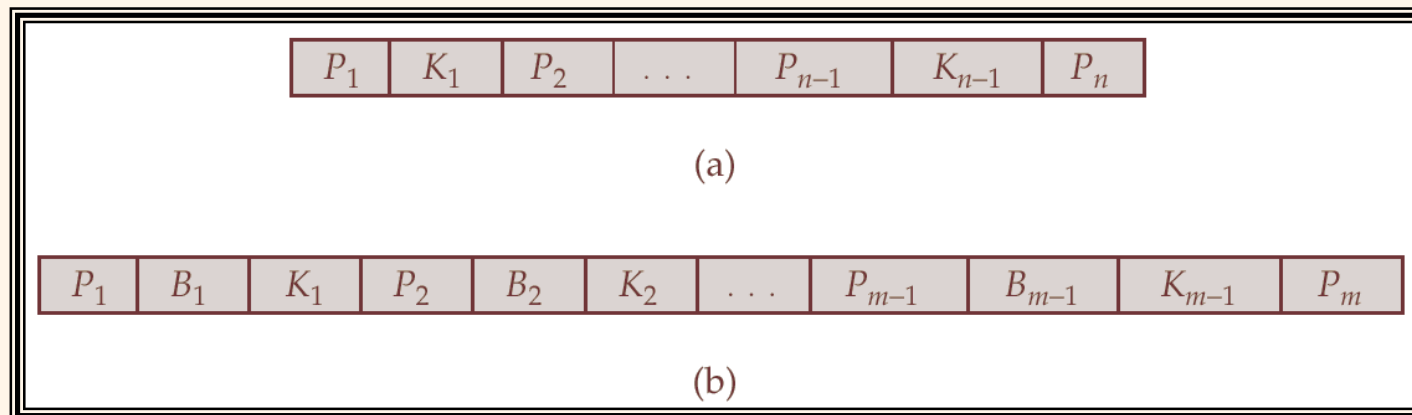


Example of B⁺-tree File Organization

- ❖ 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

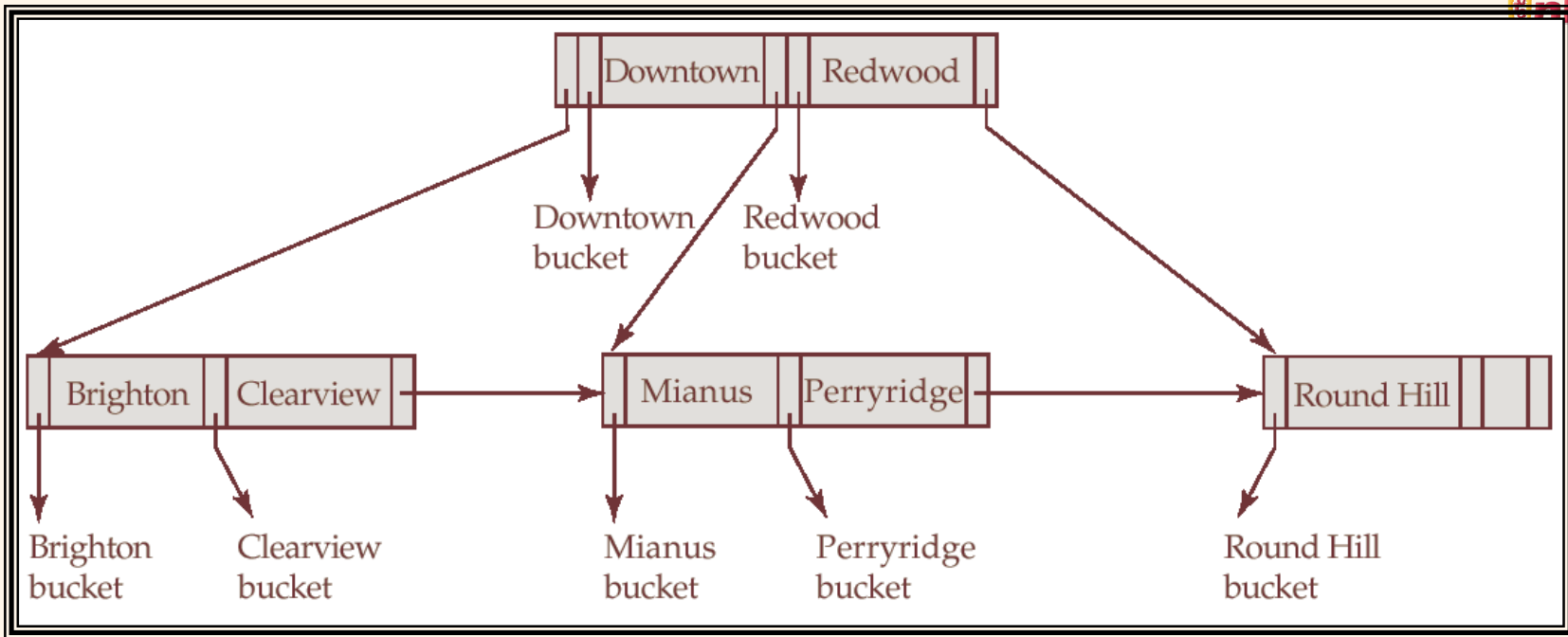
B-Tree Index Files

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

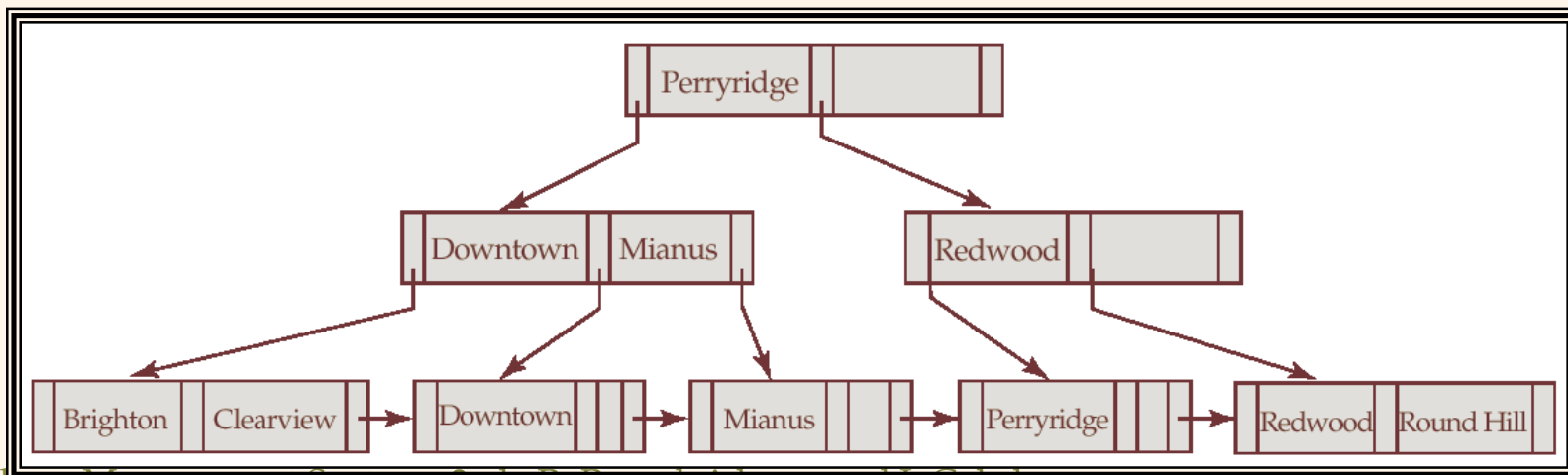


- Nonleaf node – pointers B_i are the bucket or file record pointers.

B-Tree Index File Example



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



B-Tree Index Files (Cont.)

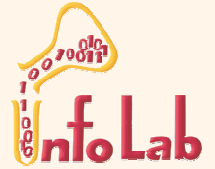
- ❖ Advantages of B-Tree indices:
 - May use less tree nodes than a corresponding B⁺-Tree.
 - Sometimes possible to find search-key value before reaching leaf node.
- ❖ Disadvantages of B-Tree indices:
 - Only small fraction of all search-key values are found early
 - Non-leaf nodes are larger, so fan-out 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 outweigh disadvantages.

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 (Cont.)



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

(See figure in next slide.)

- ❖ There are 10 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 10
 - E.g. $h(\text{Perryridge}) = 5$ $h(\text{Round Hill}) = 3$ $h(\text{Brighton}) = 3$

Example of Hash File Organization



Hash file organization of *account* file, using *branch-name* as key
(see previous slide for details).

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



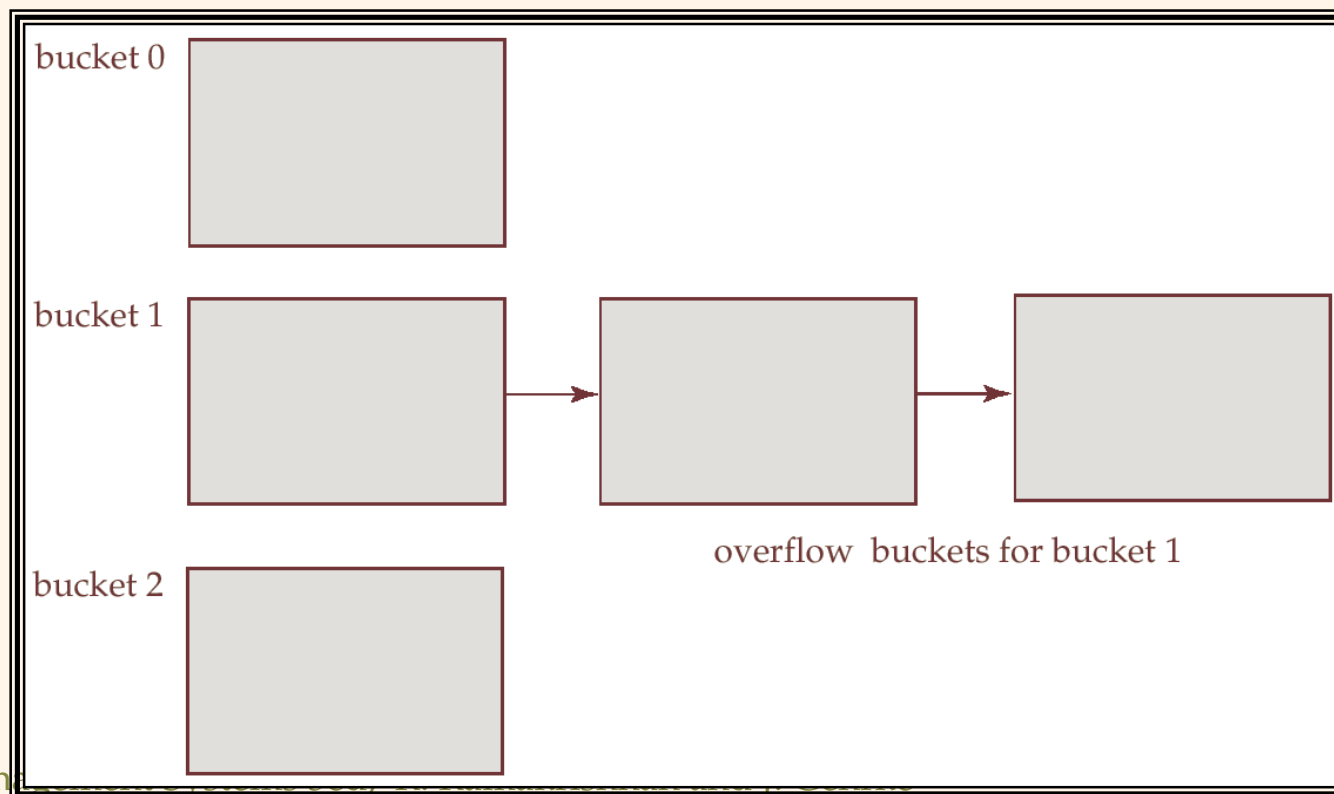
- ❖ Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- ❖ An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.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

- ❖ 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.)

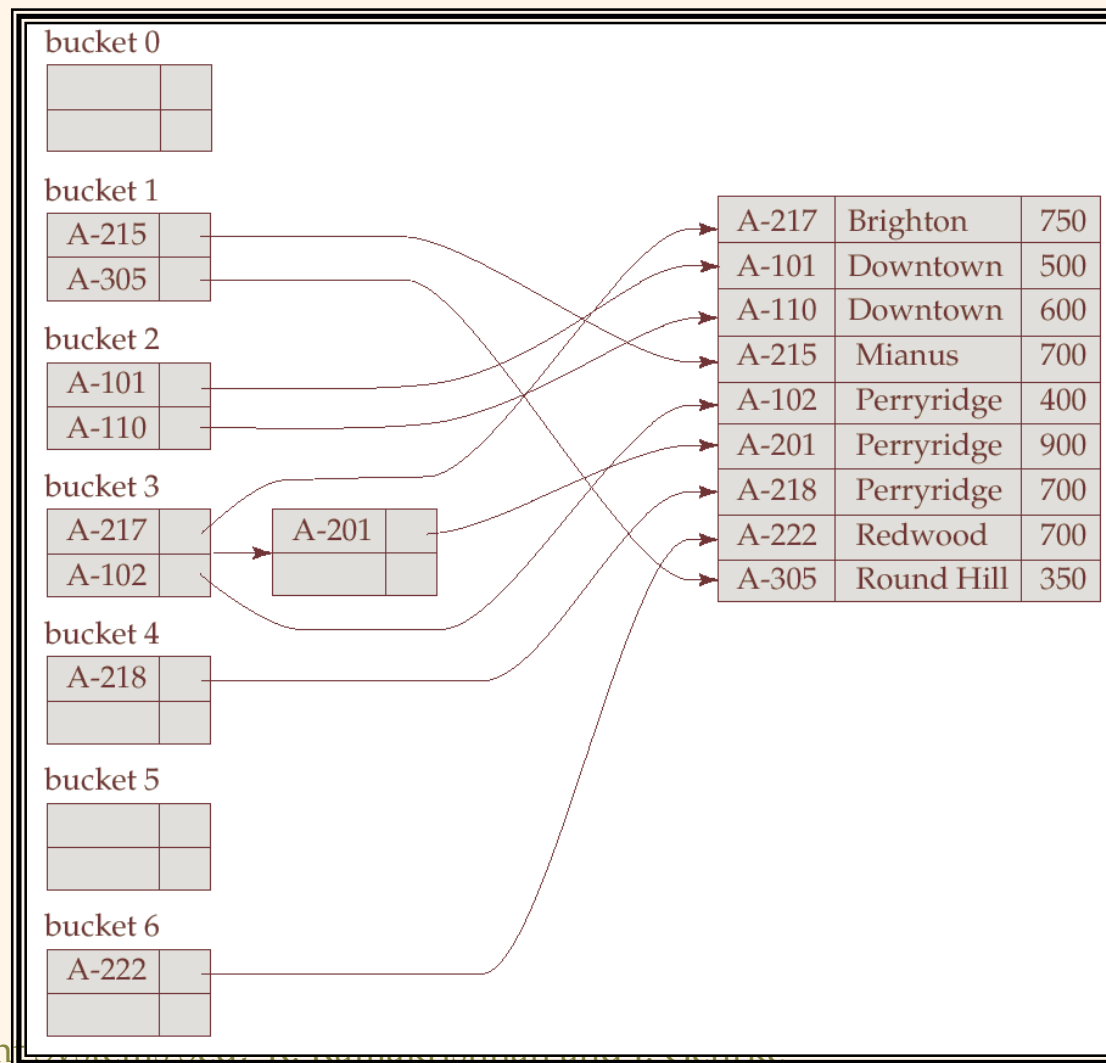
- ❖ Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.
- ❖ Above scheme is called closed hashing.
 - An alternative, called open hashing, which does not use overflow buckets, is not suitable for database applications.



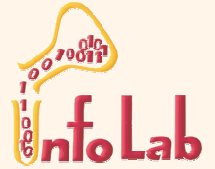
Hash Indices

- ❖ 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 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 will be 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.