COP 4710: Database Systems
Spring 2007

Chapters 10 and 11 – Indexing

Instructor : Mark Llewellyn
markl@cs.ucf.edu
ENG3 236, 823-2790
http://www.cs.ucf.edu/courses/cop4710/spr2007

School of Electrical Engineering and Computer Science
University of Central Florida

Basic Concepts Behind Indexing

- 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

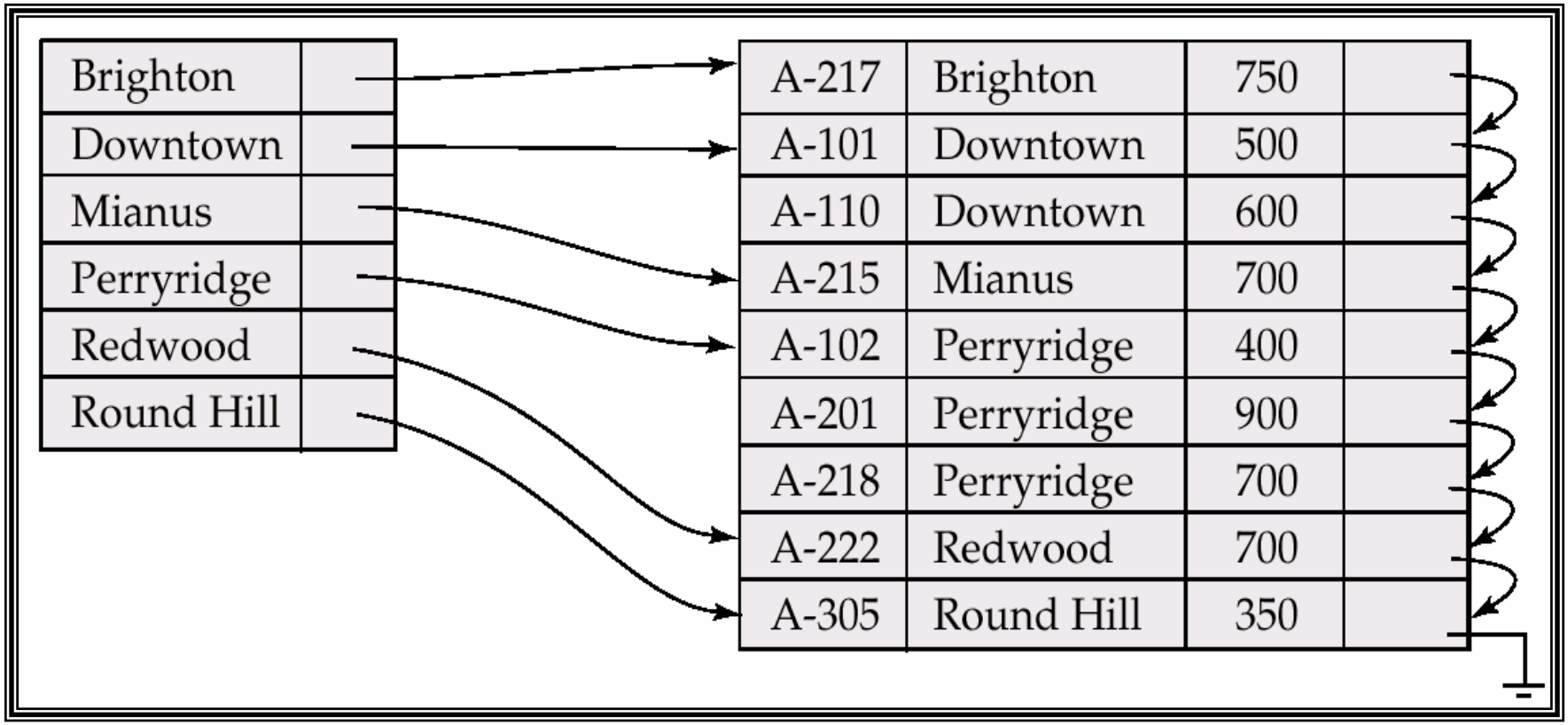
- Access types: The types of access that are efficiently supported.
  - Finding records with a specified attribute value.
  - Finding records with an attribute value falling in a specified range of values.
- Access time: The time required to find a particular data item, or set of items.
- Insertion time: The time it takes to insert a new data item. This value includes the time required to find the correct place to insert, as well as the time required to update the index structure.
- Deletion time: The time it takes to delete a data item. This value includes the time required to find the item, as well as the time required to update the index structure.
- Space overhead: The additional space required by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

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.
- One of the oldest index schemes used in database systems. Designed for applications that require both sequential processing of entire files as well as random access to individual records.

Dense Index Files

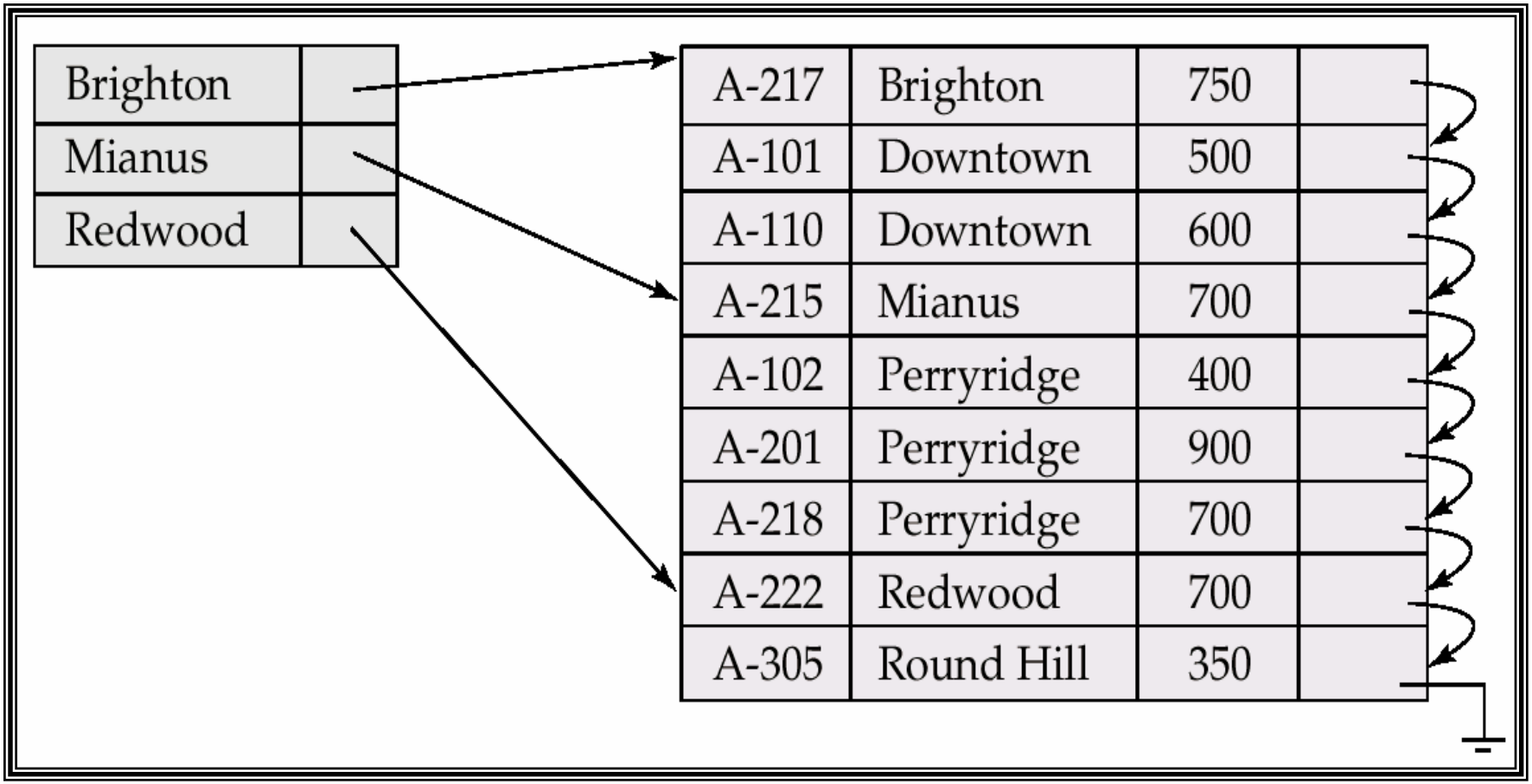
- Dense index — Index record appears for every search-key value in the file.



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
- 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.

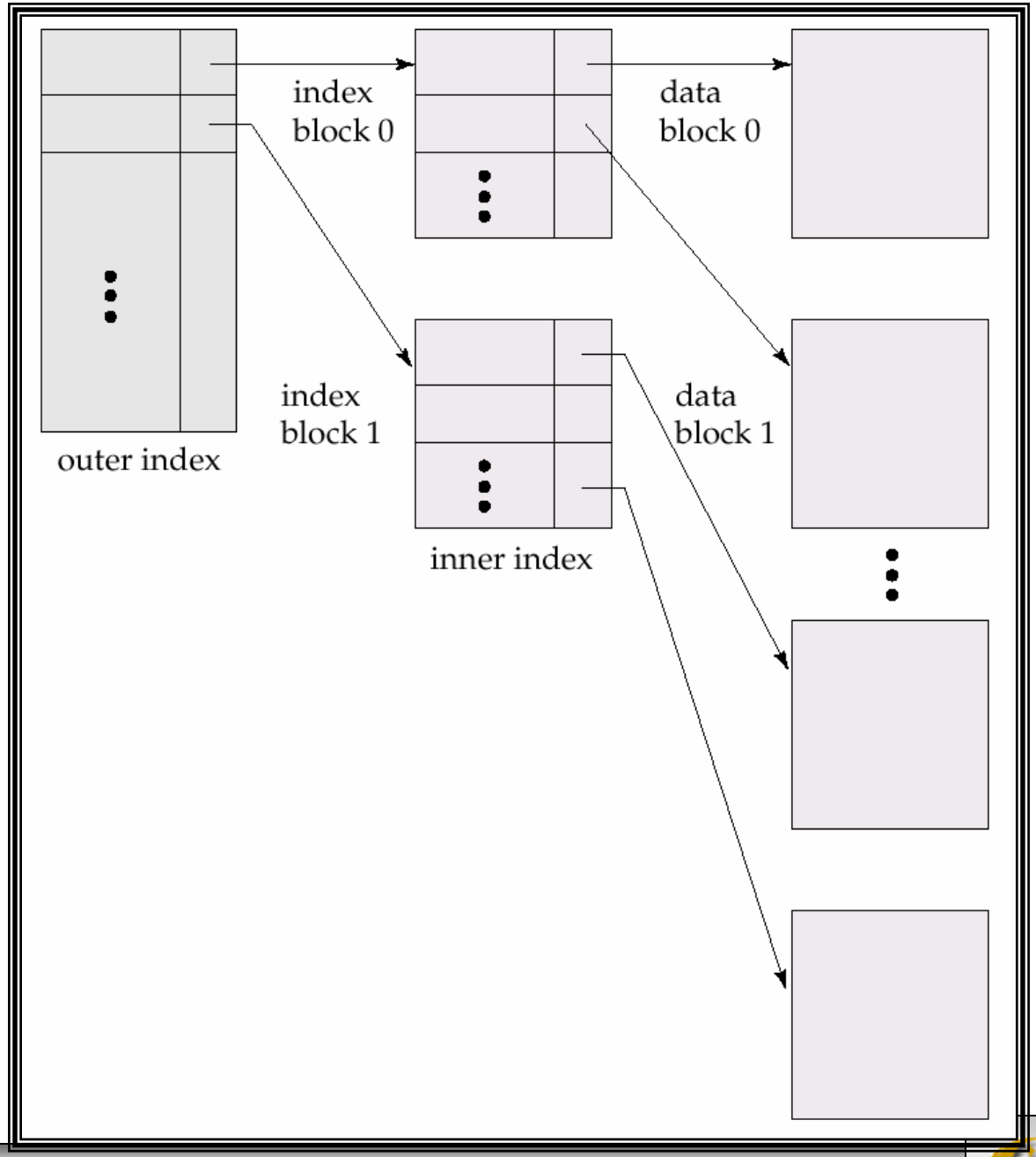
Example of Sparse Index Files



Multi-level Indexing

- If primary index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses to index records, 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.

Example of Multi-level Indexing



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 is similar to file record deletion.
  - Sparse indices – if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

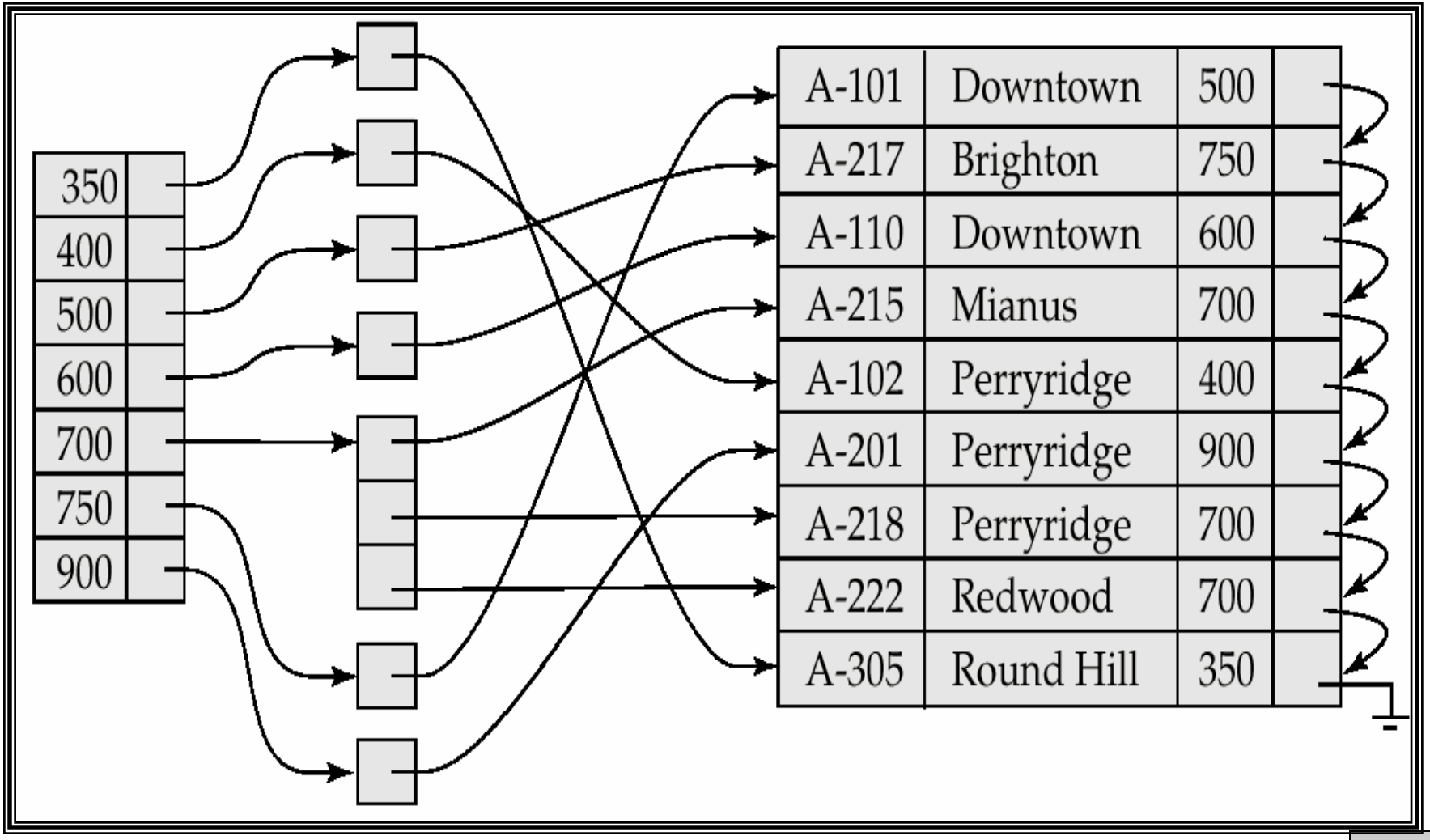
Index Update: Insertion

- 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. In this case, 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) that satisfy some condition.
  - Example 1: In the *account* database 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; index record points to a bucket that contains pointers to all the actual records with that particular search-key value.

Secondary Index on ***balance*** field of ***account***



Primary and Secondary Indices

- Secondary indices must be dense.
- Indices offer substantial benefits when searching for records.
- When a file is modified, every index on the file must be updated, Updating indices imposes overhead on database modification.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - each record access may fetch a new block from disk

B^+-Tree Index Files

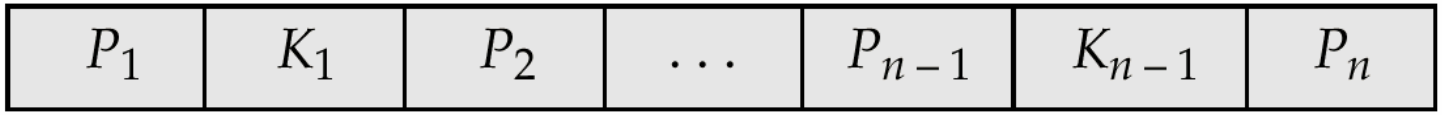
- B+-tree indices are an alternative to indexed-sequential files.
- Disadvantage of indexed-sequential files: performance degrades as file grows, since many overflow blocks get created. Periodic reorganization of entire file is required.
- Advantage of B^+-tree index files: automatically reorganizes itself with small, local, changes, in the face of insertions and deletions. Reorganization of entire file is not required to maintain performance.
- Disadvantage of B^+-trees: extra insertion/deletion overhead and space overhead.
- Advantages of B^+-trees outweigh disadvantages, and they 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 (i.e., all leaves are on the same level).
  - Each node that is not a root or a leaf holds *k-1* keys and *k* references to subtrees where $\lceil n/2 \rceil \leq k \leq n$
  - A leaf node holds *k–1* keys where $\lceil n/2 \rceil \leq k \leq n$
  - 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 (*k–1*) values.

B+-Tree Node Structure

- Typical node

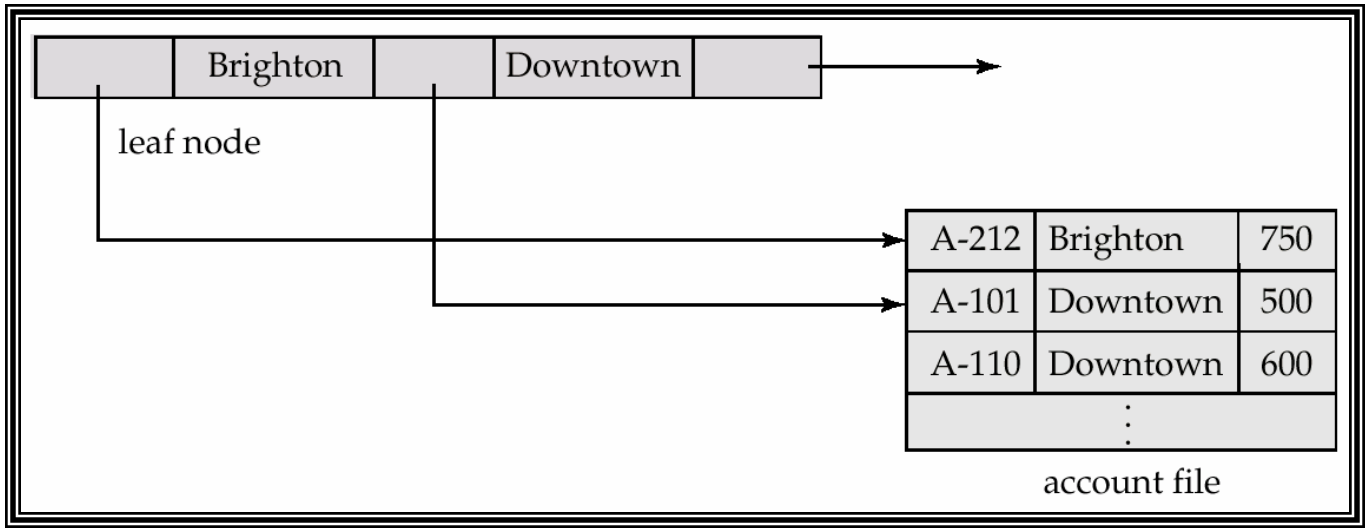


  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

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

Leaf Nodes in B+-Trees

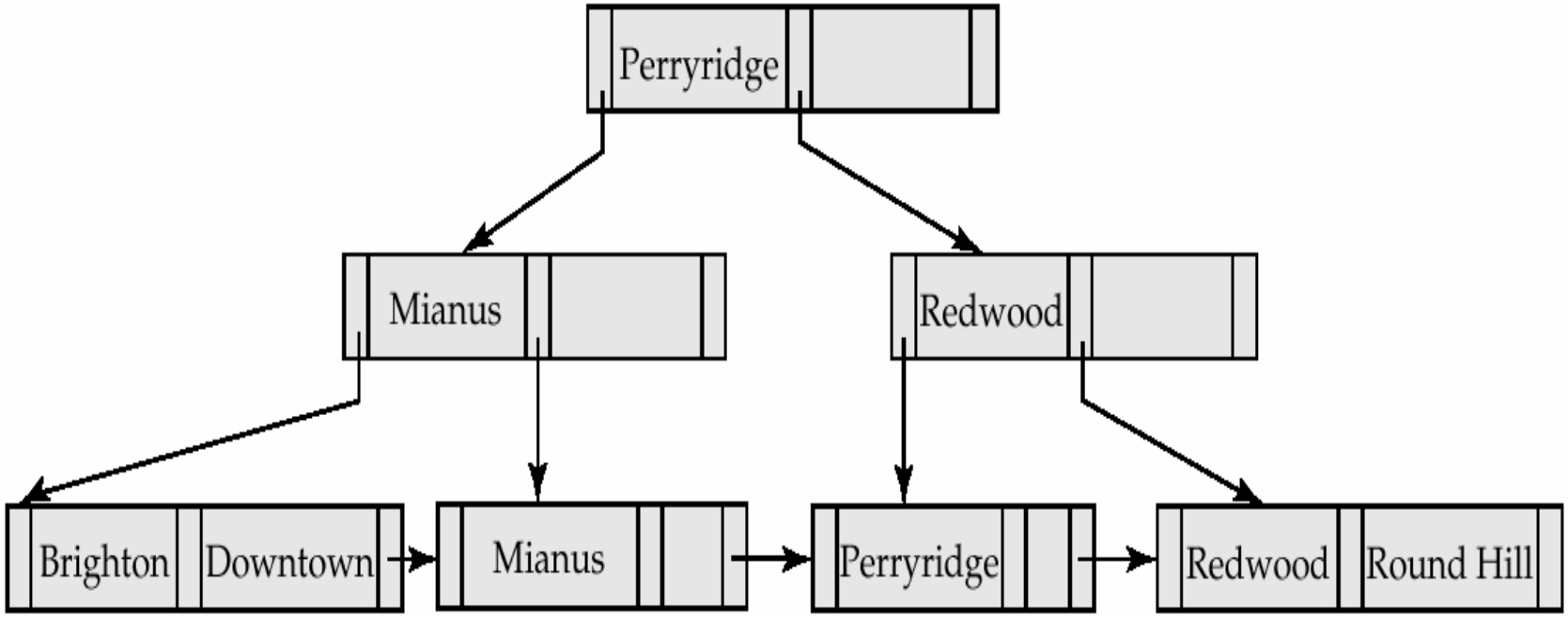
- For $i = 1, 2, \ldots, n-1$, pointer P_i either points to a file record with search-key value K_i, or to a bucket of pointers to file records, each record having search-key value K_i. Only need bucket structure if search-key does not form a primary key.
- If L_i, L_j are leaf nodes and $i < j$, L_i's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



Non-Leaf Nodes in B$^+$-Trees

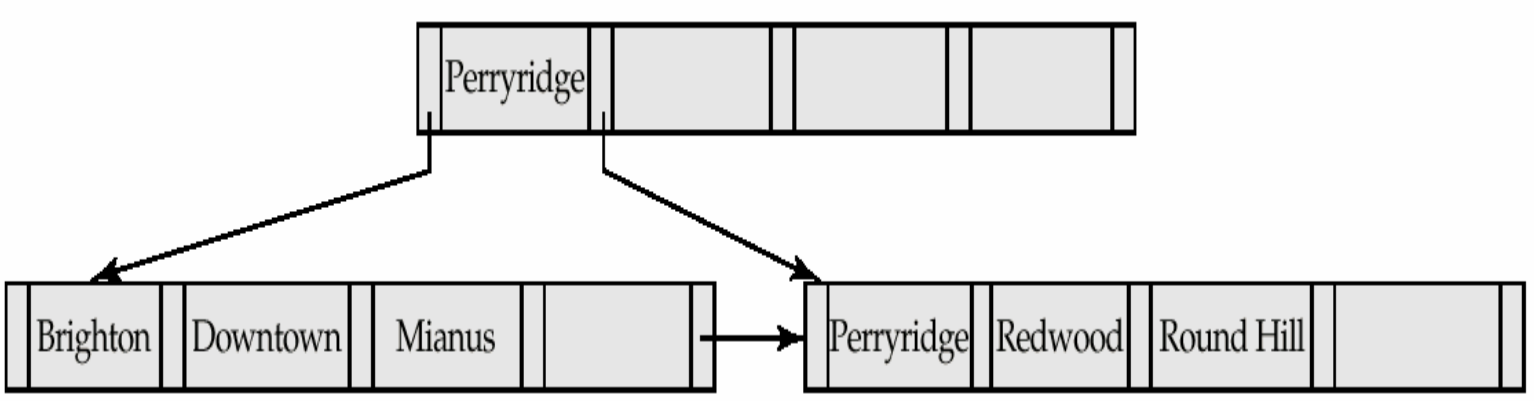
- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with *m* pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal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

Example of a B+-tree



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

Example of B+-tree



B+-tree for *account* file (*n* - 5)

- 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 (logarithmic in the size of the main file), 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.

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_i to the child node
    3. Otherwise $k \geq K_{n-1}$, where there are n pointers in the node. Then follow P_n 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.)

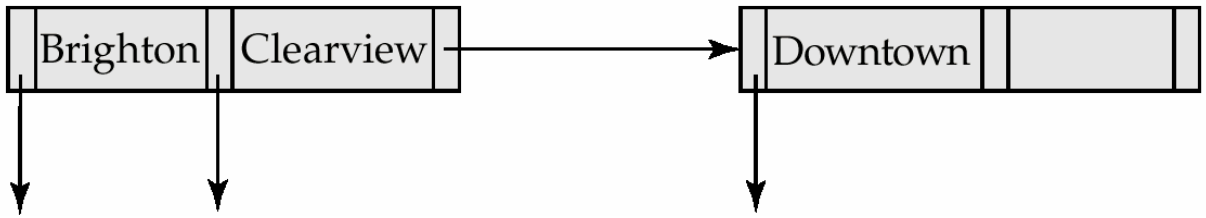
- In processing a query, a path is traversed in the tree from the root to some leaf node.
- If there are K search-key values in the file, the path is no longer than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 KB, and n is typically around 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.

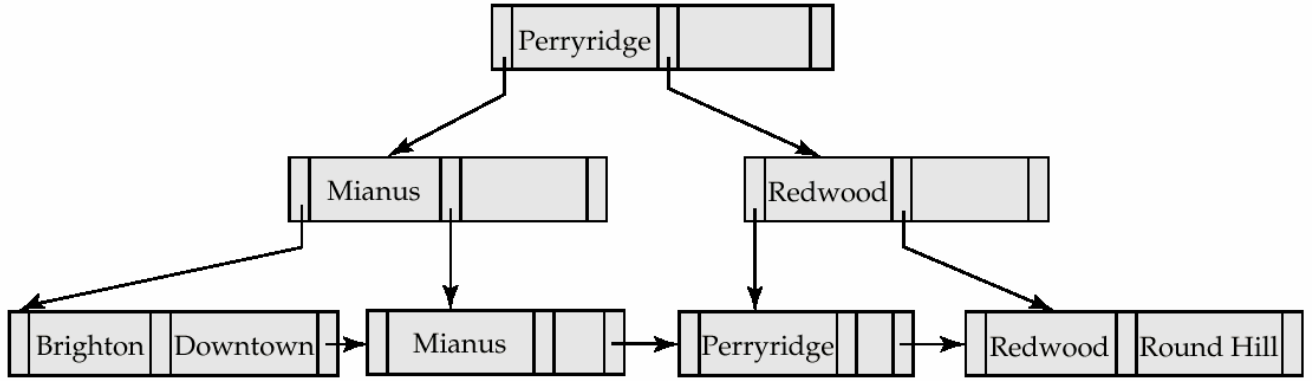
Updates on B+-Trees: Insertion (cont.)

- 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.

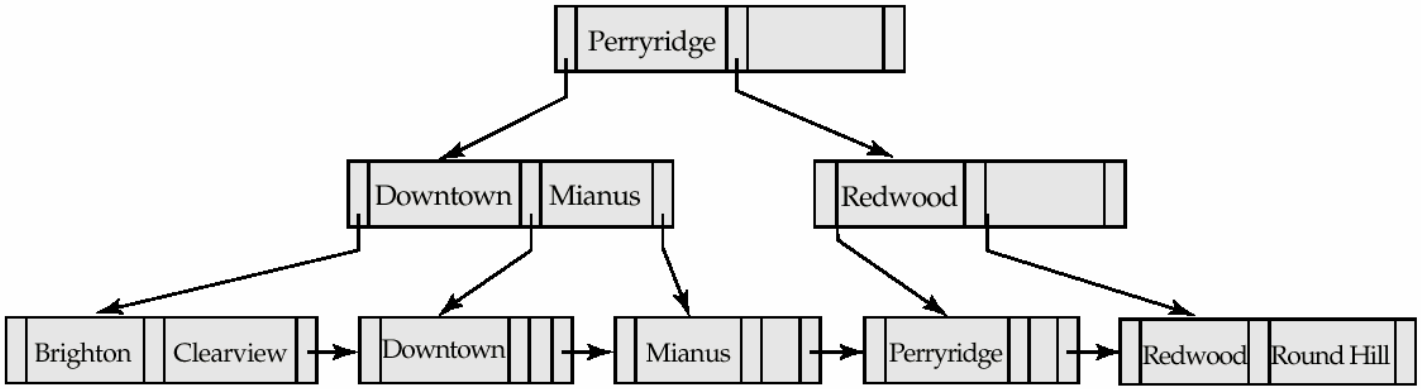


Result of splitting node containing Brighton and Downtown on inserting Clearview into the B+-tree shown on page 20 (n = 3)

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



before



after

Result of splitting node containing Brighton and Downtown on inserting Clearview

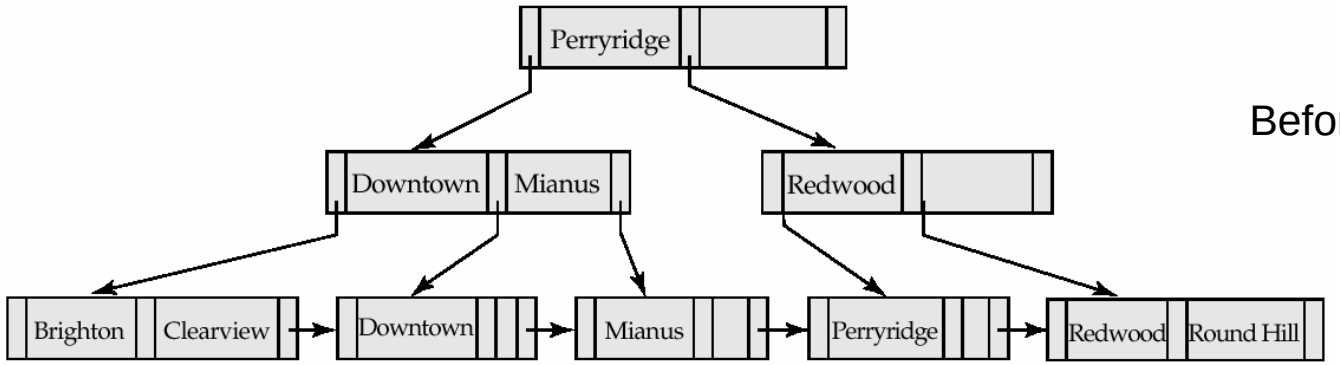
Updates on B$^+$-Trees: Deletion

- 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.

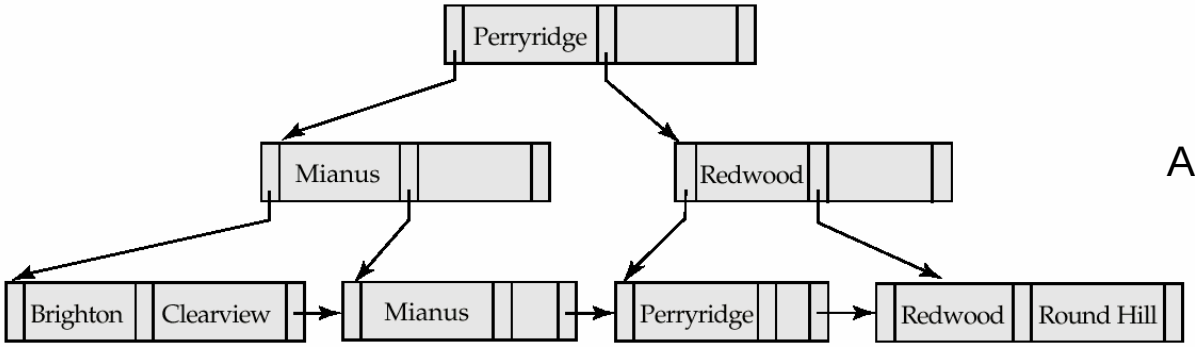
Updates on B$^+$-Trees: Deletion (cont.)

- Otherwise, 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
  - 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



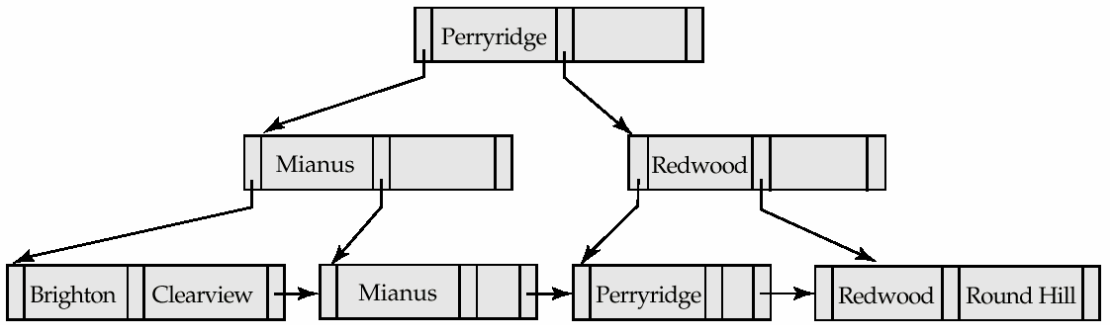
Before



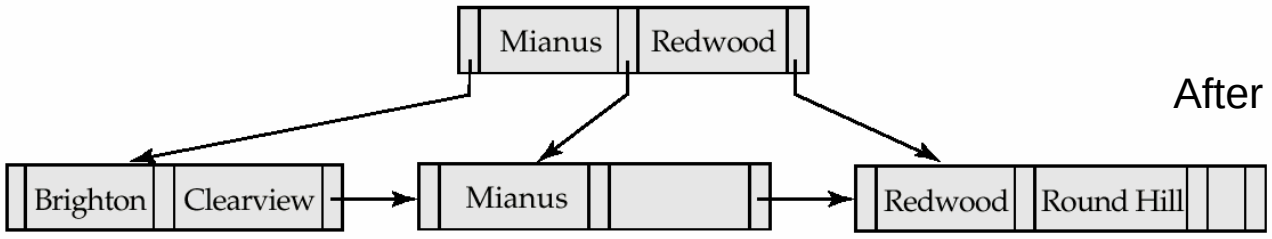
After deleting “Downtown”

The removal of the leaf node containing “Downtown” did not result in its parent having too few pointers. So the cascaded deletions stopped with the deleted leaf node’s parent.

Examples of B+-Tree Deletion (cont.)



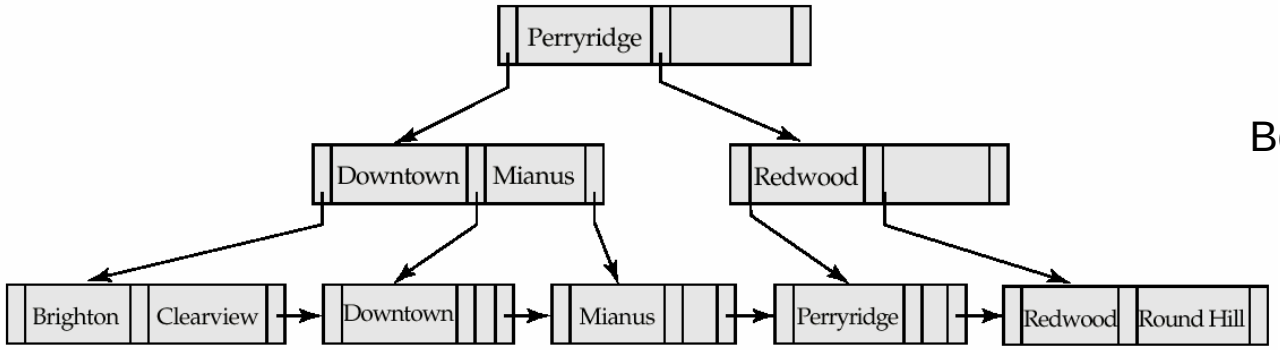
Before



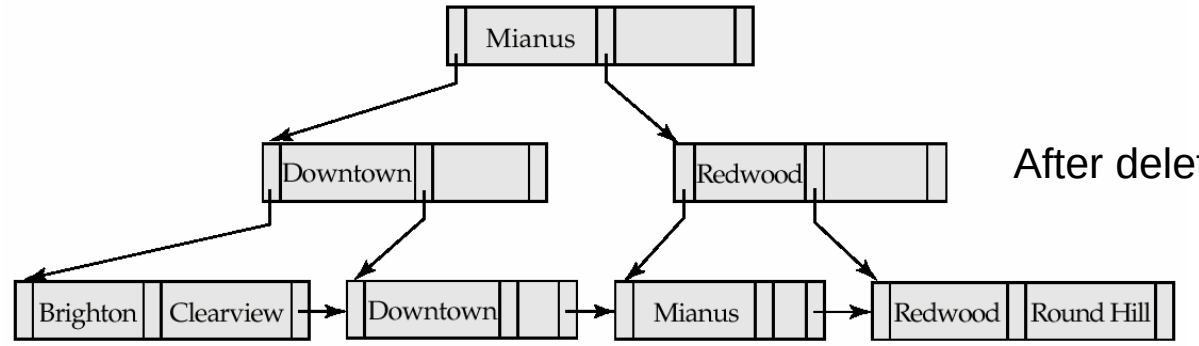
After deletion of "Perryridge"

- 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



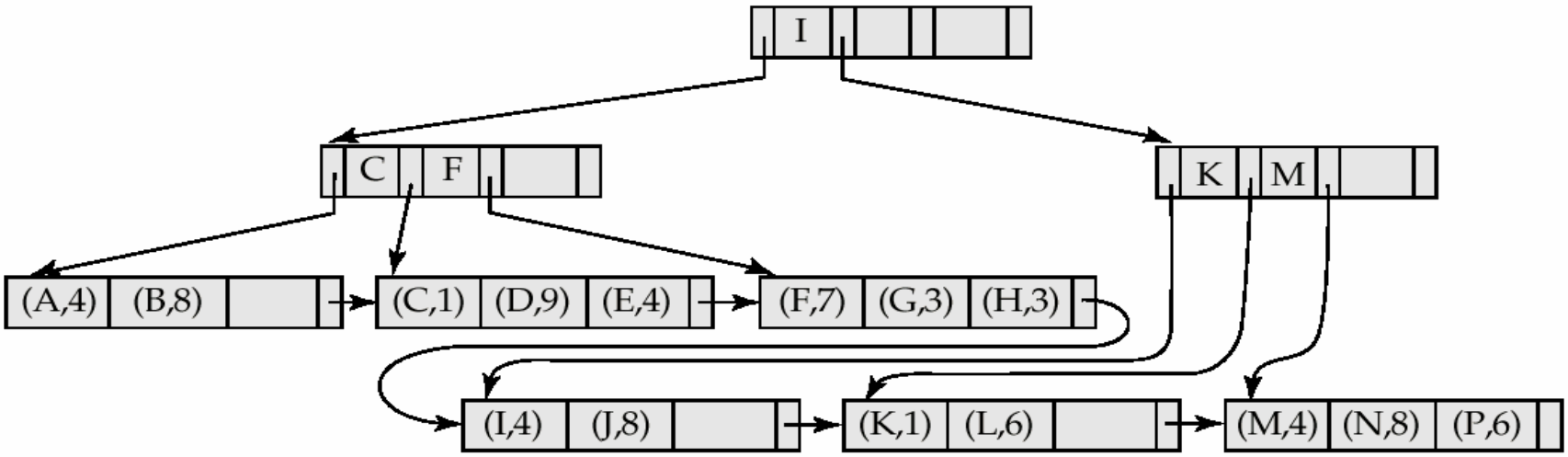
After deletion of "Perryridge"

- 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.

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 (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(Perryridge) = 5 h(Round Hill) = 3 h(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 1

bucket 2

bucket 3

A-217	Brighton	750
A-305	Round Hill	350

bucket 4

A-222	Redwood	700

bucket 5

A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700

bucket 6

bucket 7

A-215	Mianus	700

bucket 8

A-101	Downtown	500
A-110	Downtown	600

bucket 9

Hash Functions

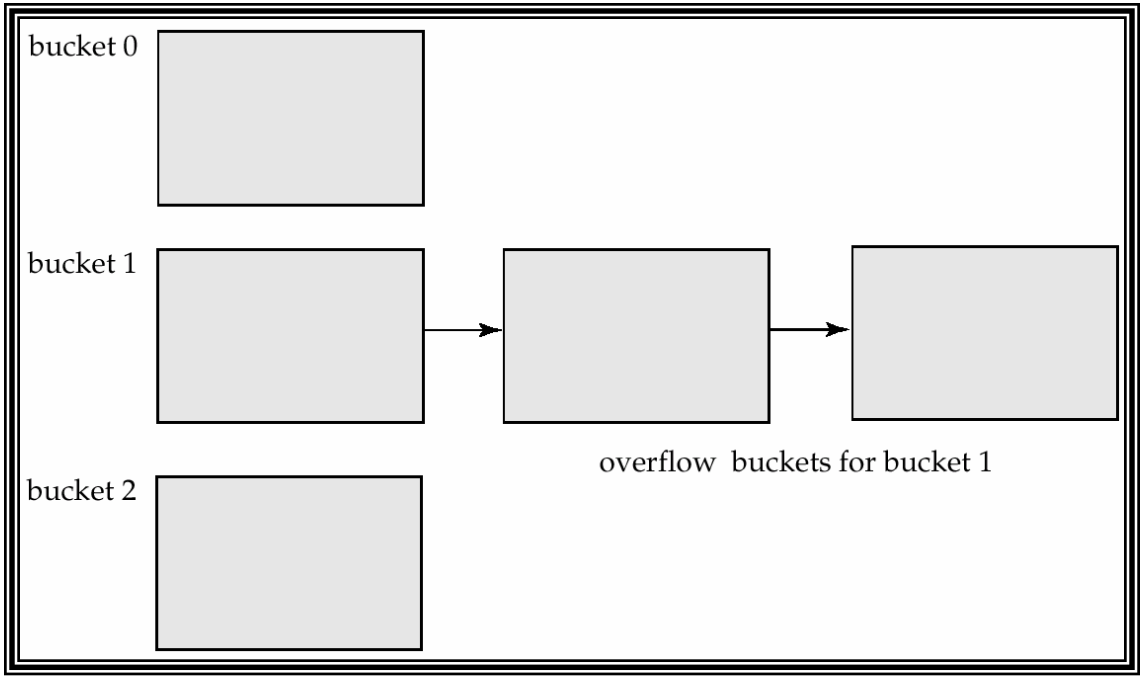
- Worst has function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
-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

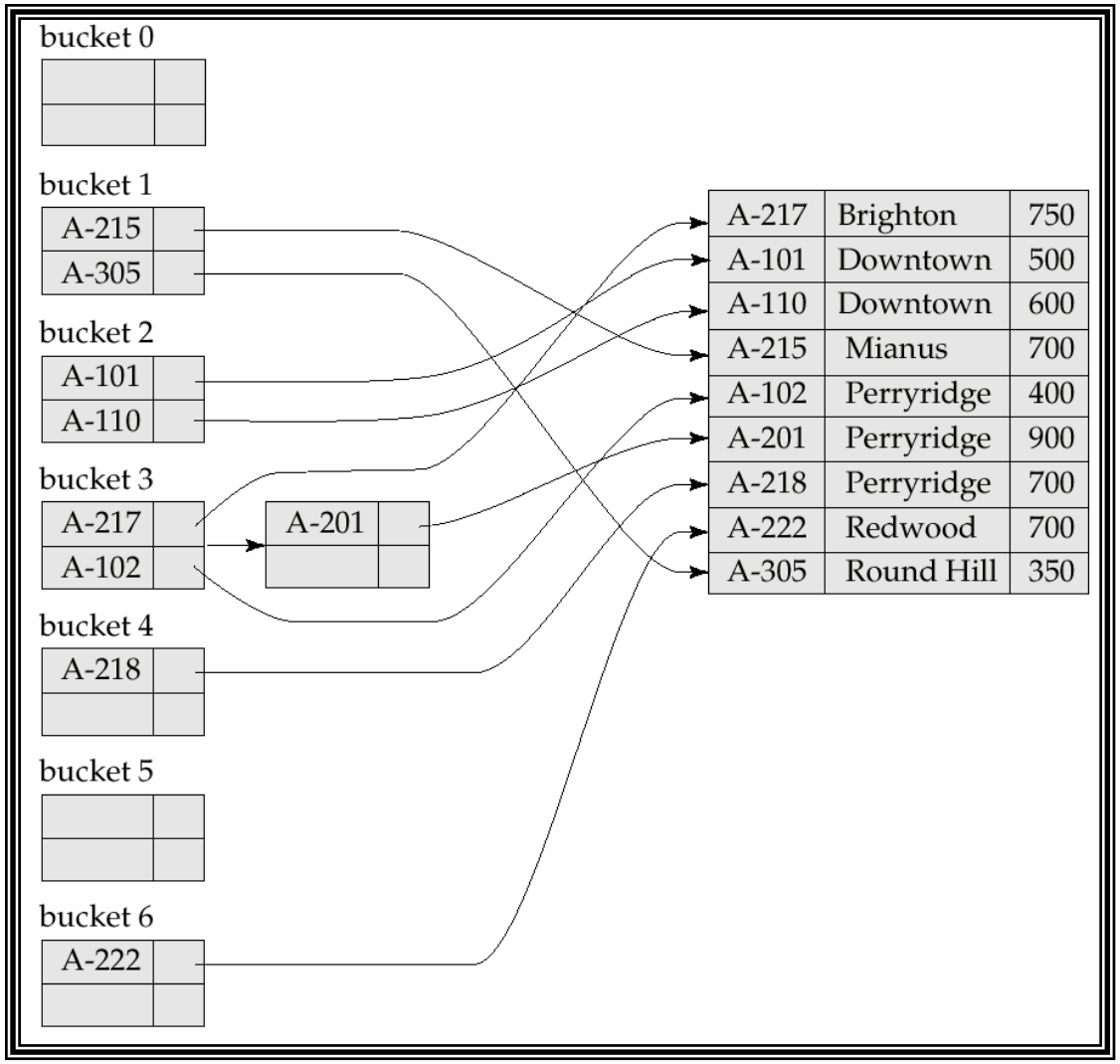
- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called closed hashing.
  - An alternative, called open hashing, which does not use overflow buckets, is not suitable for database applications.



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



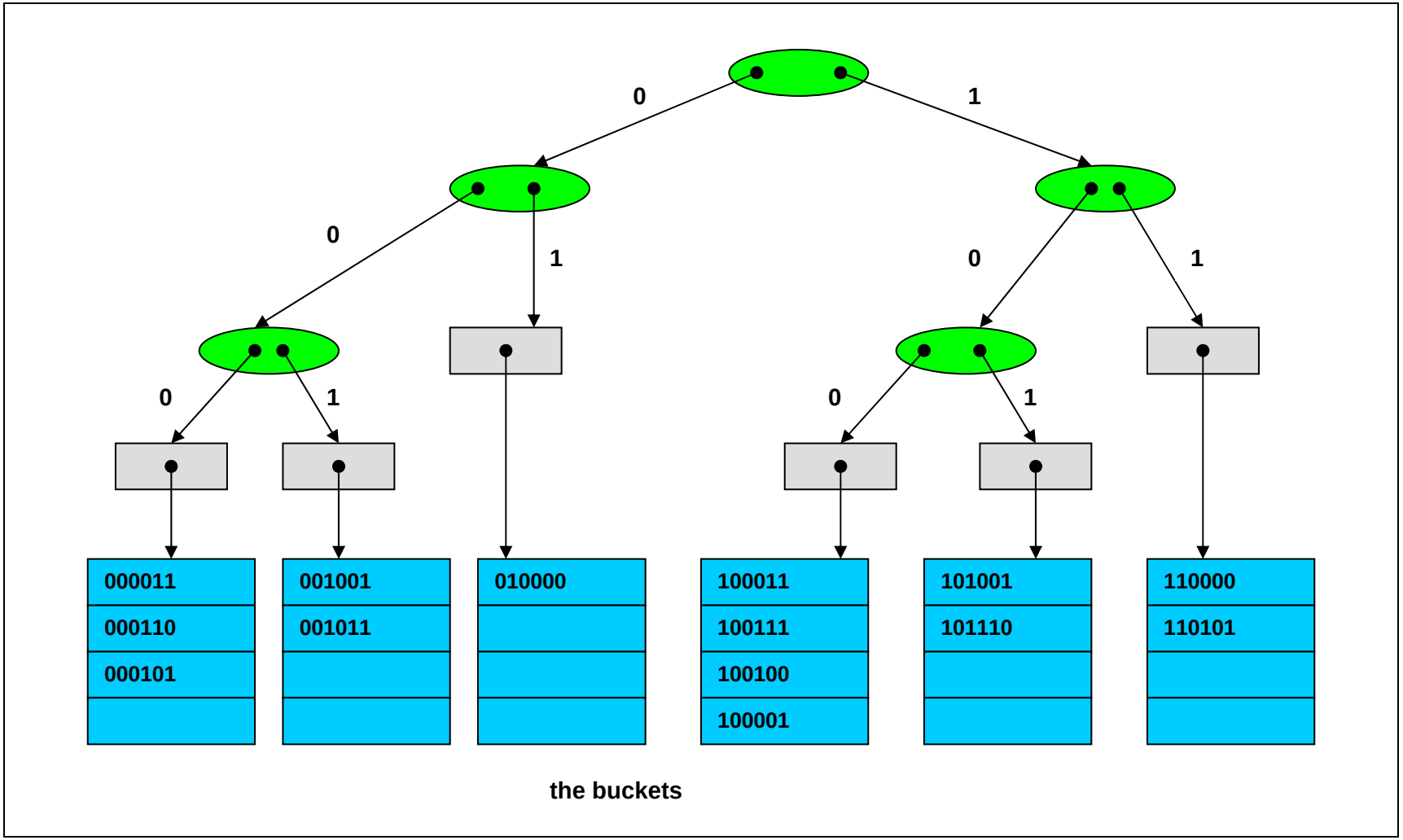
Deficiencies of Static Hashing

- 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.

Dynamic Hashing

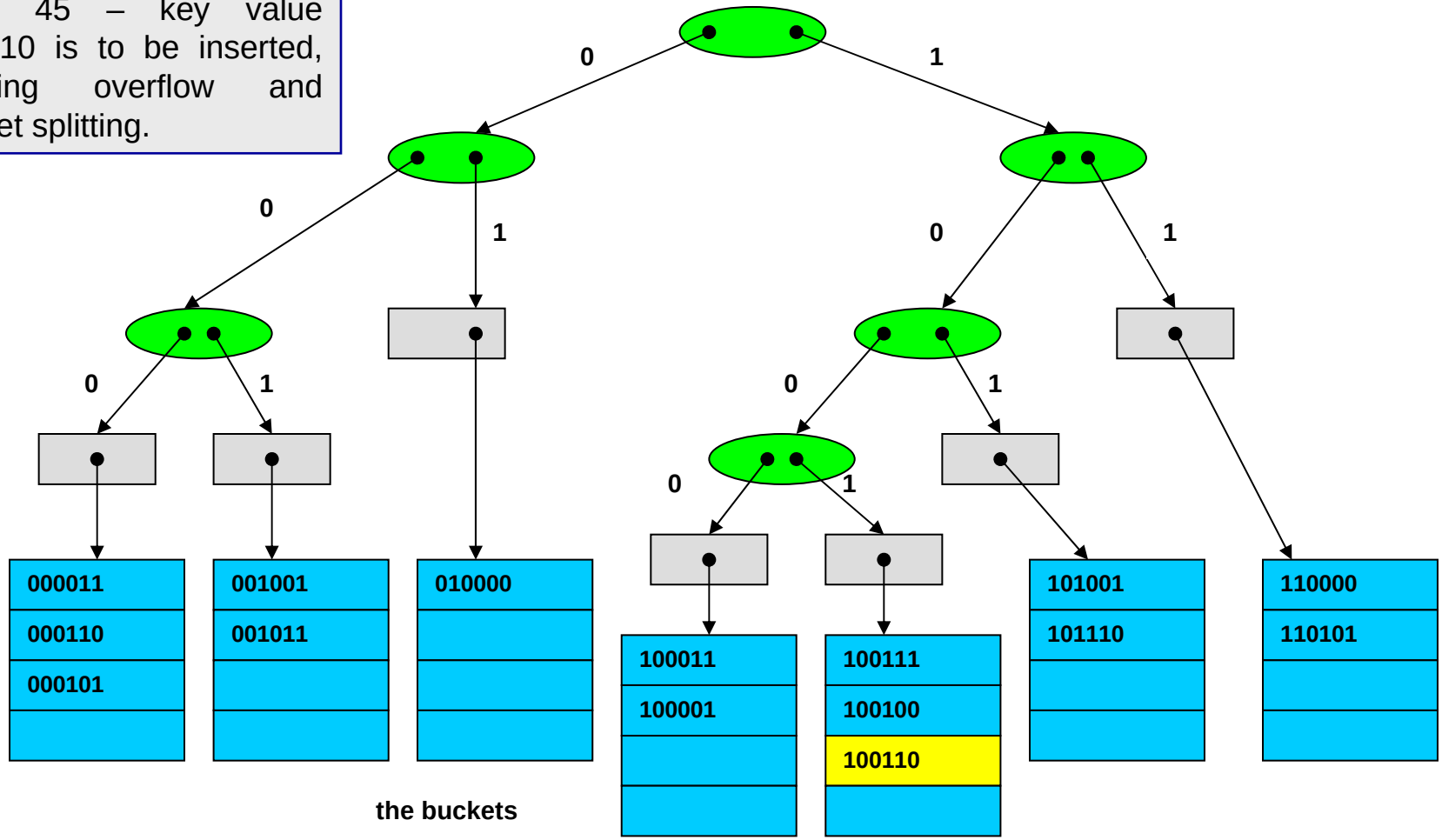
- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
  - Hash function generates values over a large range — typically *b*-bit integers, with $b = 32$.
  - At any time use only a prefix of the hash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

Dynamic Hashing Example



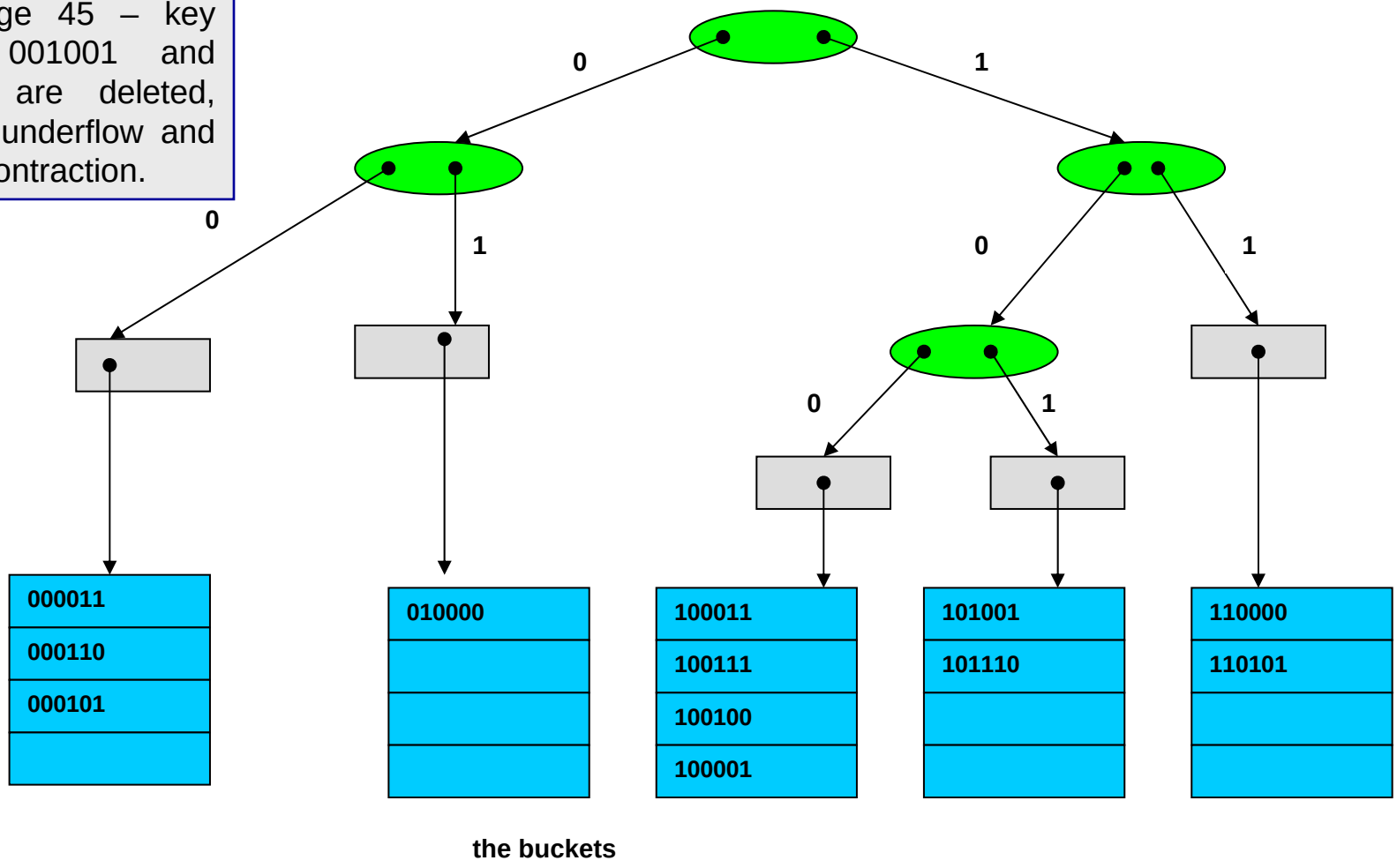
Example: Insert Key Value 100110

Starting with structure on page 45 – key value 100110 is to be inserted, causing overflow and bucket splitting.

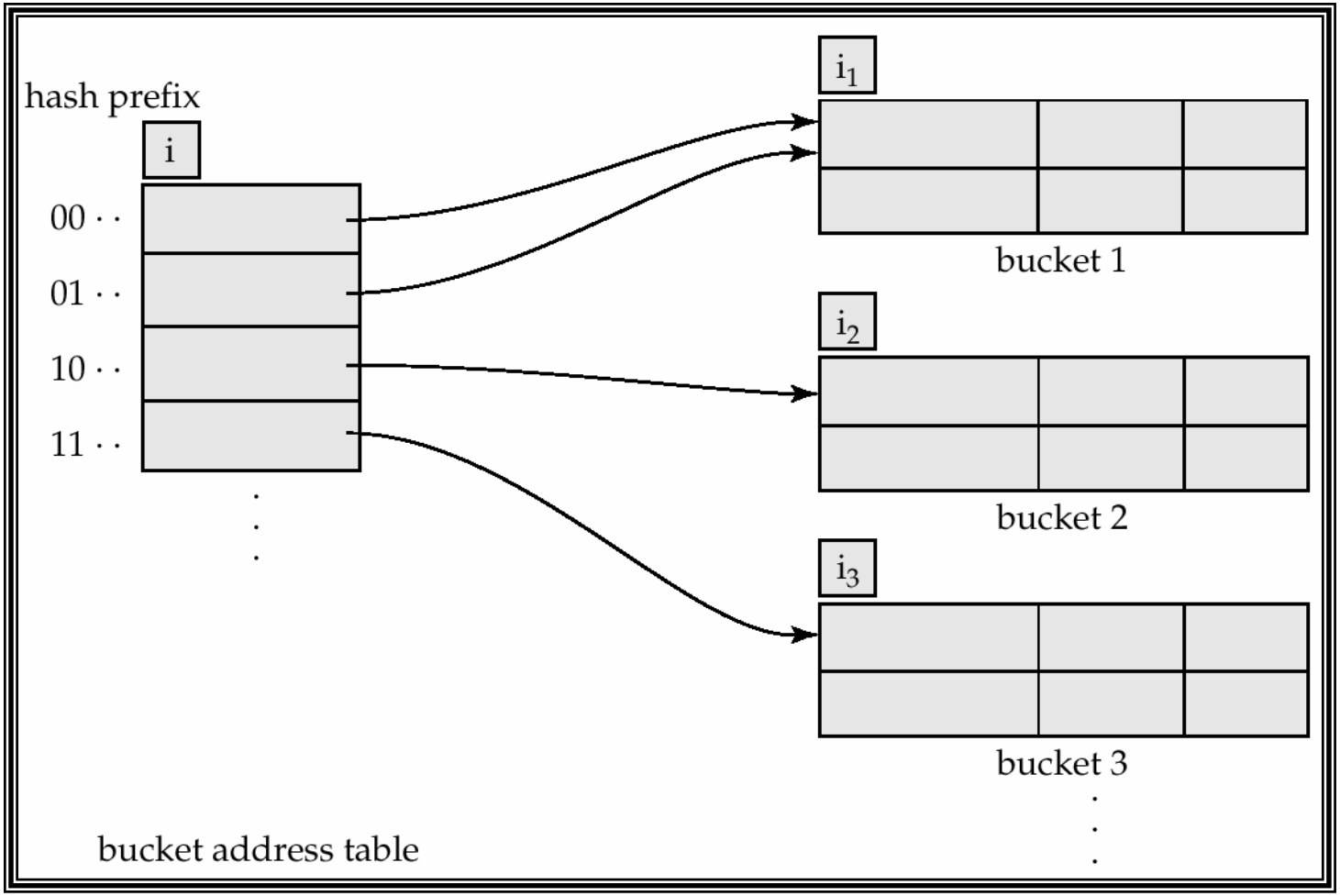


Example: Deletion of Key Values 001001 and 001011

Starting with structure on page 45 – key values 001001 and 001011 are deleted, causing underflow and bucket contraction.



General Extendable Hash Structure



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

Use of Extendable Hash Structure

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

Updates in Extendable Hash Structure

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

- If $i > i_j$ (more than one pointer to bucket j)
  - allocate a new bucket z, and set i_j and i_z to the old i_j -+ 1.
  - make the second half of the bucket address table entries pointing to j to 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)
  - 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.

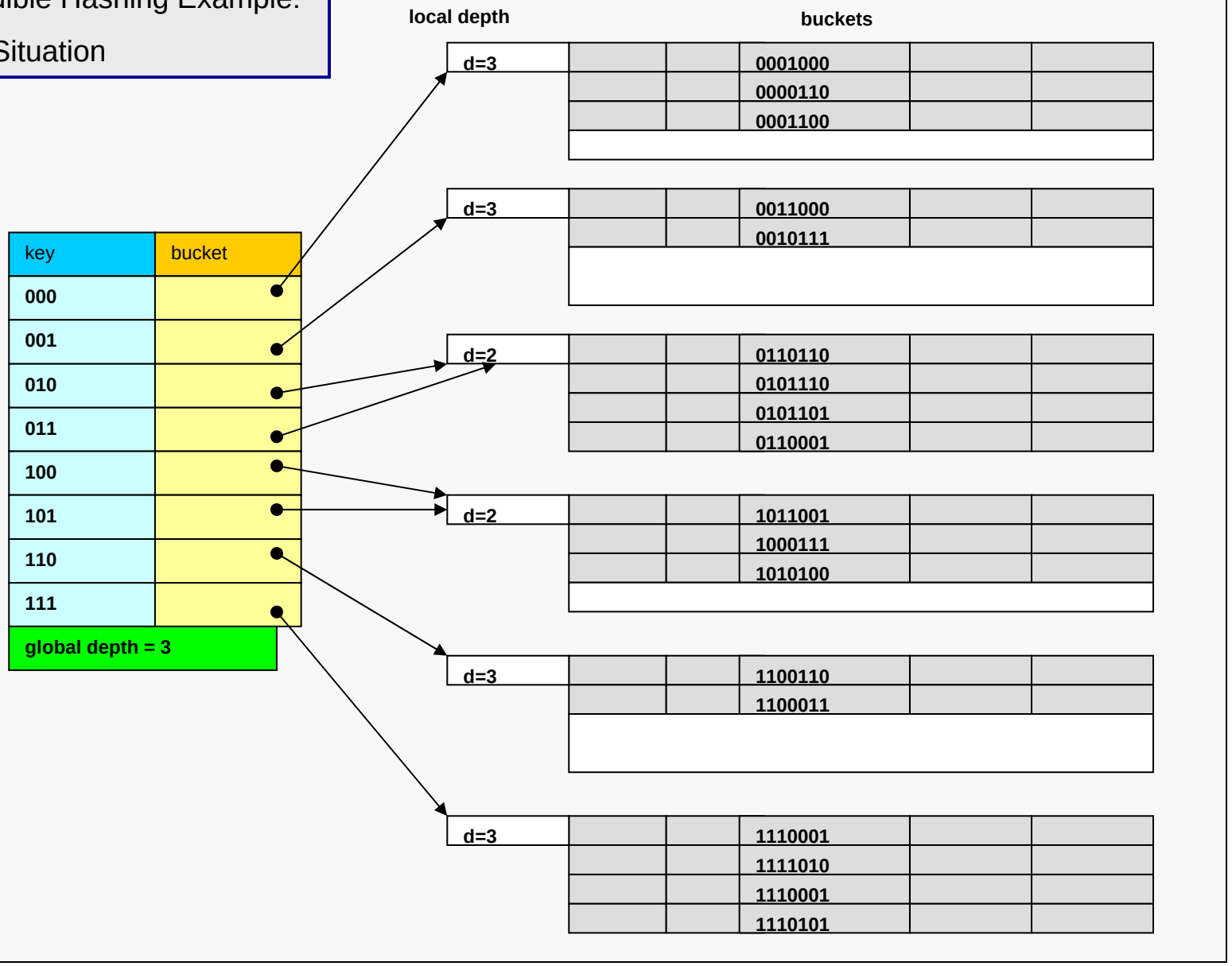
Updates in Extendable Hash Structure

(cont.)

- When inserting a value, if the bucket is full after several splits (that is, *i* reaches some limit *b*) create an overflow bucket instead of splitting bucket entry table further.
- To delete a key value,
  - locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  - Coalescing of buckets can be done (can coalesce only with a "buddy" bucket having same value of i_j and same $i_j - 1$ prefix, if it is present)
  - Decreasing bucket address table size is also possible
    - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

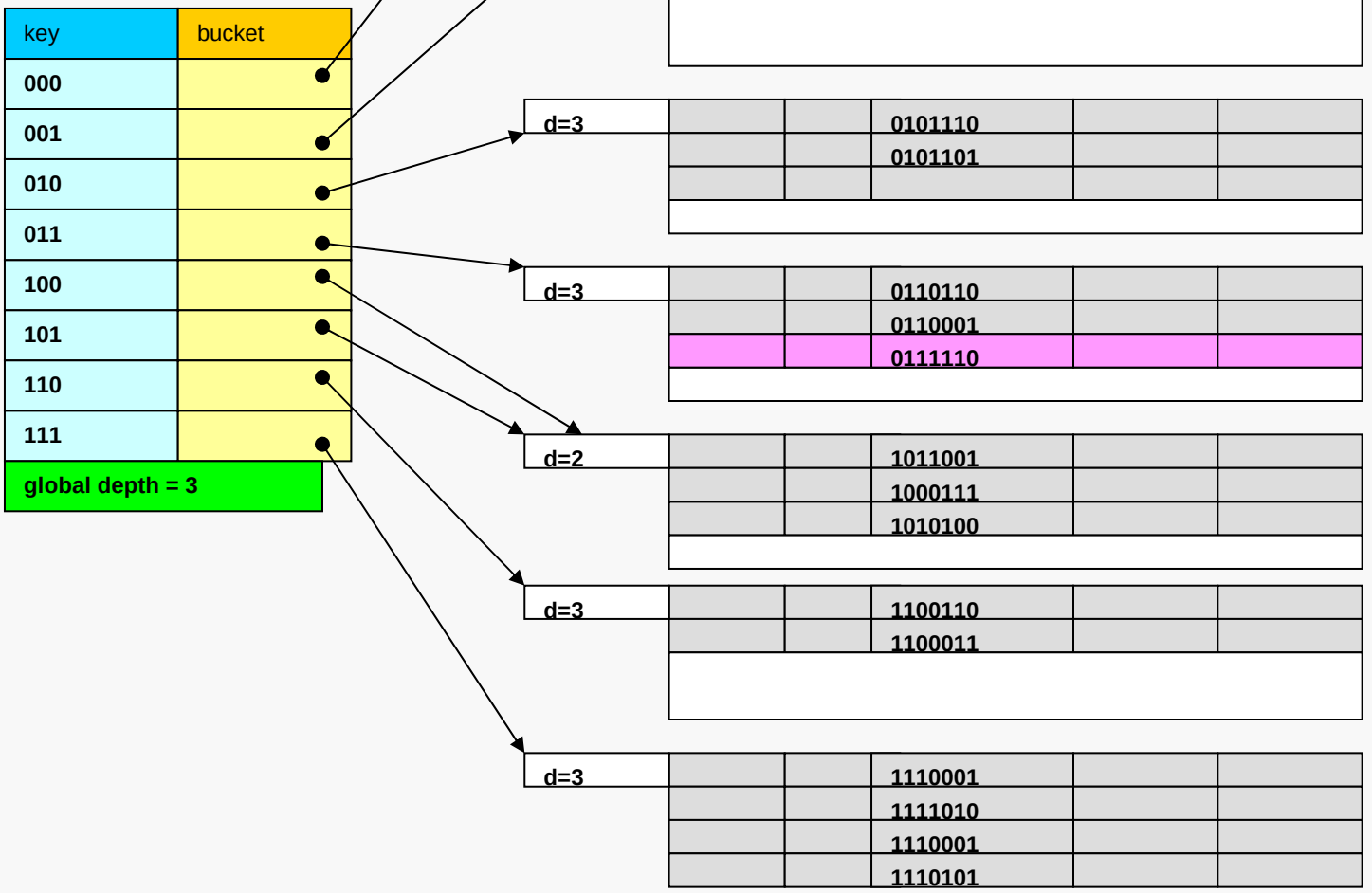
Extendible Hashing Example:

Initial Situation



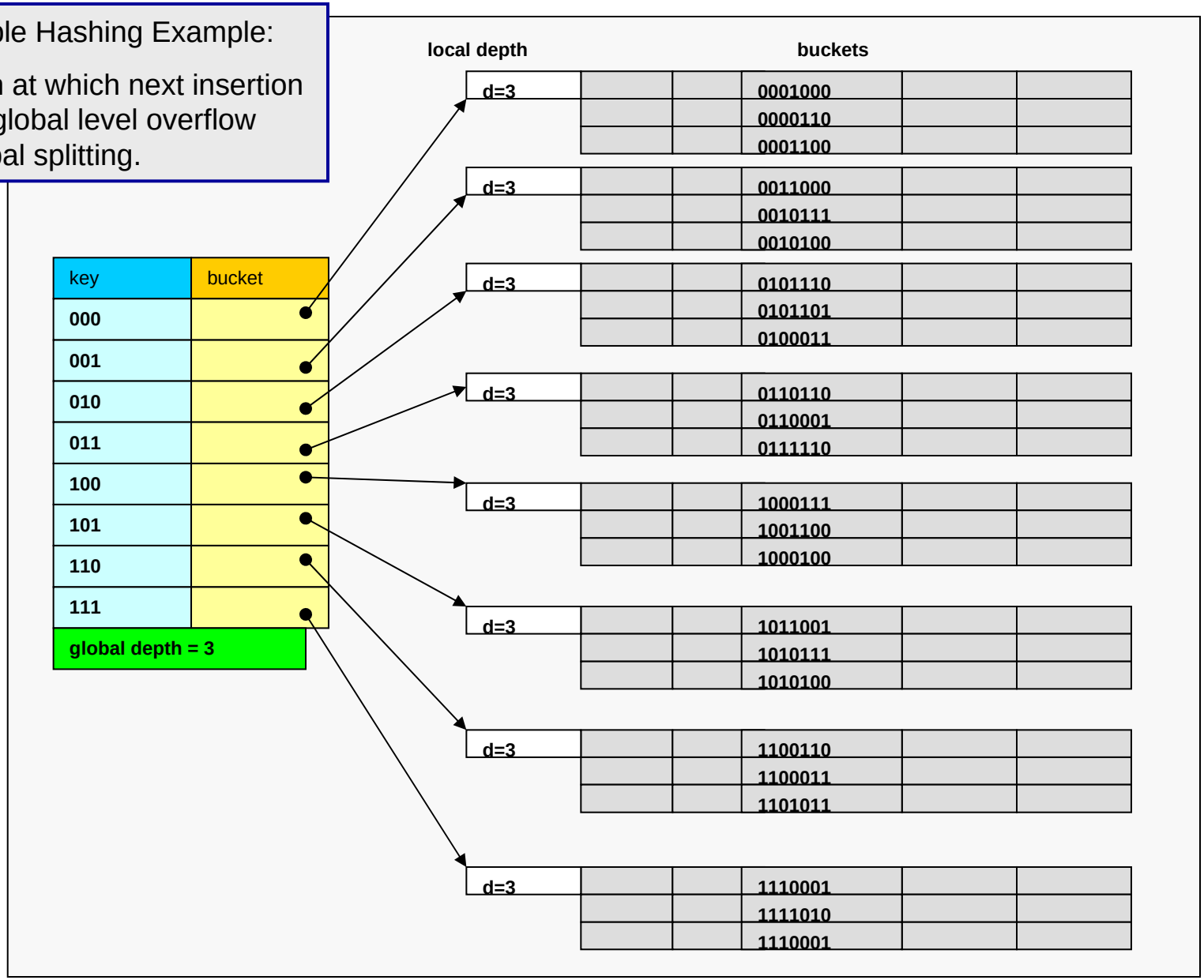
Extendible Hashing Example:

Insertion of key value 0111110 causing overflow and splitting



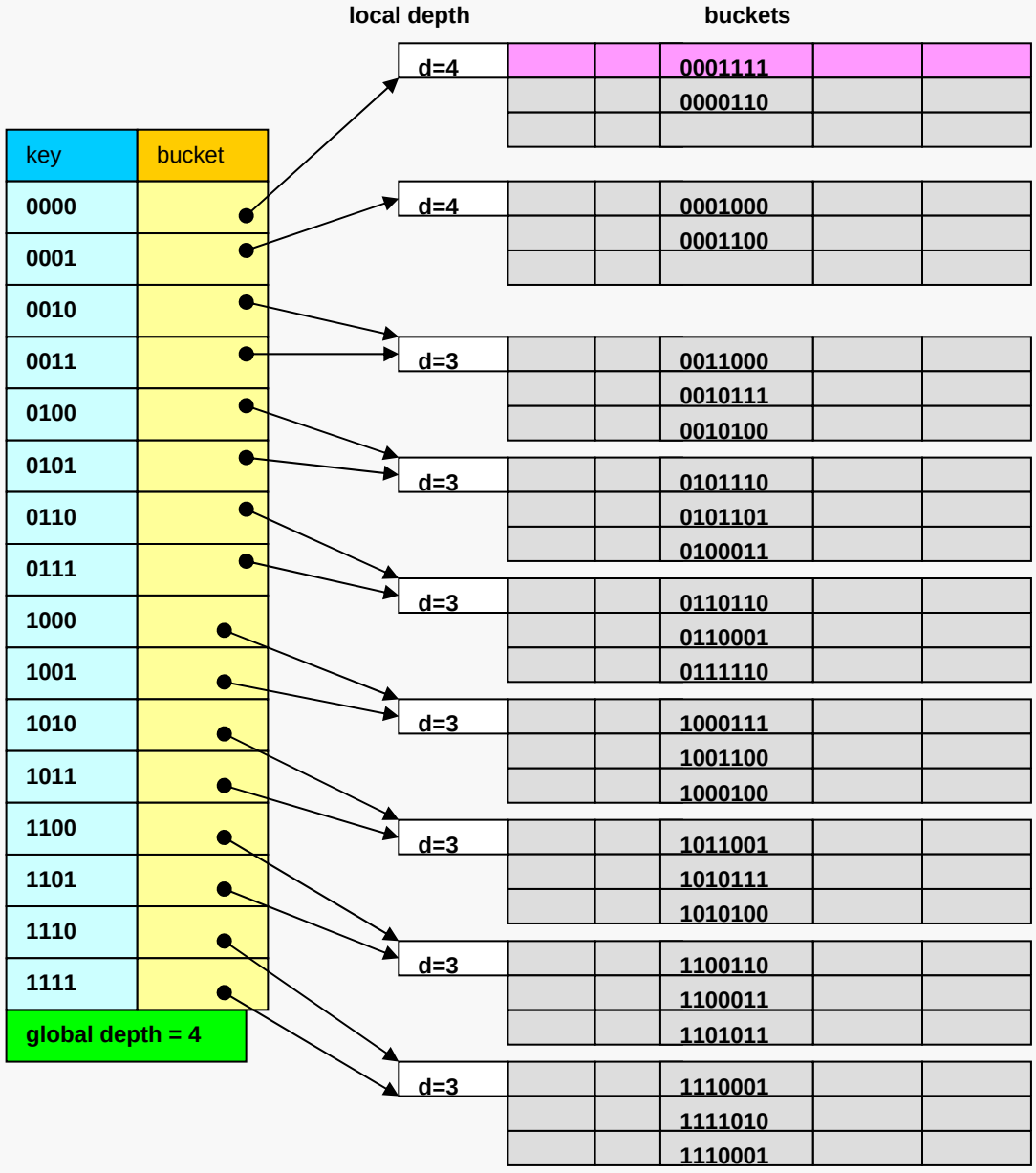
Extendible Hashing Example:

Situation at which next insertion causes global level overflow and global splitting.



Extendible Hashing Example:

Insertion of key value 00001111 into situation shown on page 59, causes global level overflow and global splitting.



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)
    - Need a tree structure to locate desired record in the structure!
  - Changing size of bucket address table is an expensive operation
- Linear hashing is an alternative mechanism which avoids these disadvantages at the possible cost of more bucket overflows

Linear Hashing

- The basic idea behind linear hashing is to provide dynamic expansion and contraction of the hash file address space without requiring the overhead of a directory structure.
- This is accomplished with the overhead of a single integer and a slightly modified search algorithm.
- Suppose that the address space starts with *M* buckets numbered 0, 1, 2, …, M-1 and uses a simple modulo hash function h(K) = K mod M, this hash function is called the initial hash function h0.

Linear Hashing

- Collisions are still resolved using chaining. However, when a collision occurs which leads to an overflow in *any* bucket, the *first* bucket in the file, bucket 0, is split into two buckets, the original bucket 0 and a new bucket *M* at the end of the file space. The records originally in bucket 0 are redistributed between bucket 0 and bucket *M* based upon a new hashing function h1(K) = K mod (2M).
- A requirement of the new hash function h1 is that any record that hashed to bucket 0 on hash function h0 must hash to either bucket 0 or bucket *M* on hash function h1.

Linear Hashing

- As further collisions leading to overflow records occur, additional buckets are split in the *linear* order 1, 2, 3, …
- If enough overflow occurs, eventually all the file buckets will be split, so the records in overflow are redistributed into regular buckets using the h1 hash function via a *delayed split* of their buckets.
- In this manner we don't need a directory structure – only a value *n* to determine how many buckets have been split. For retrieving a record with hash key *K*, first apply the function h0 to *K*; if h0(*K*) < *n*, use function h1 on *K* because this indicates that the first bucket has already been split and the records from the first bucket were redistributed between bucket 0 and bucket *M* by the h1 hash function. Initially, n = 0, indicating that the hash function h0 applies to all buckets; *n* grows linearly as buckets are split.

Linear Hashing

- When *n* = *M*, all the original buckets have been split and the hash function h1 applies to all the records in the file. At this point *n* is reset to 0, and any new collisions causing bucket overflow lead to the use of a new hashing function h2 where h2(K) = K mod (4M). In general, a sequence of hashing functions hj(K) = K mod (2j M) is used where j = 0, 1, 2, „„; a new hashing function hj+1is needed whenever all the buckets 0, 1, …, (2j M)-1 have been split and *n* is reset to 0.
- The search algorithm required for the linear hashing technique is given below:

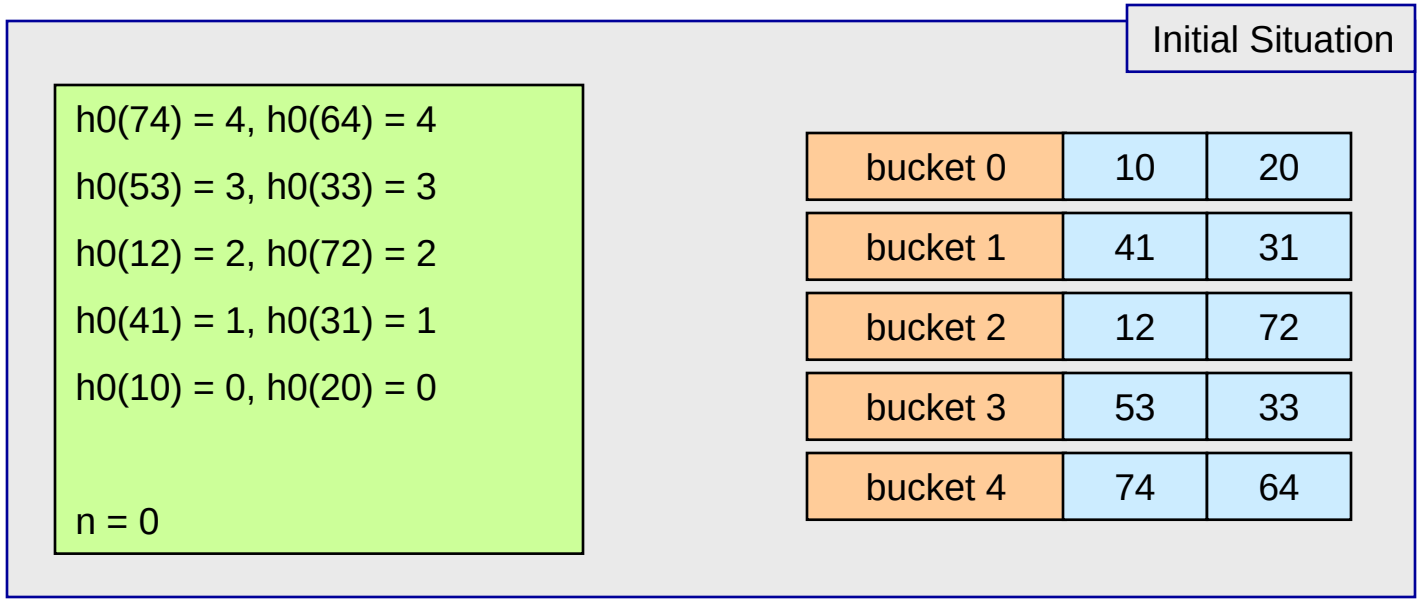
```
if n = 0
     then m ← hj(K)  //m is the hash value of record with key K
    else
    {          m ← hj(K);
               if m < n then m ← hj+1(K)
    }
search the bucket whose hash value is m (and its overflow, if any);
```

- The following example will clarify the operation of linear hashing.

Linear Hashing Example: Initial Situation

Assume that we have a 5 bucket address space with each bucket capable of holding two records.

Assume that our sequence of hash functions is just a modulo operation and that all keys are simply integer values.



Linear Hashing Example: Insert Key 63

Since this key value maps to bucket 3 and this bucket is full, a collision occurs with the new key value record being placed into an overflow chain. In addition, the first bucket is split into two buckets, bucket 0 and bucket M with record redistribution occurring and n is incremented to 1. This is shown below:

h0(74) = 4, h0(64) = 4

h0(53) = 3, h0(33) = 3, h0(63) = 3

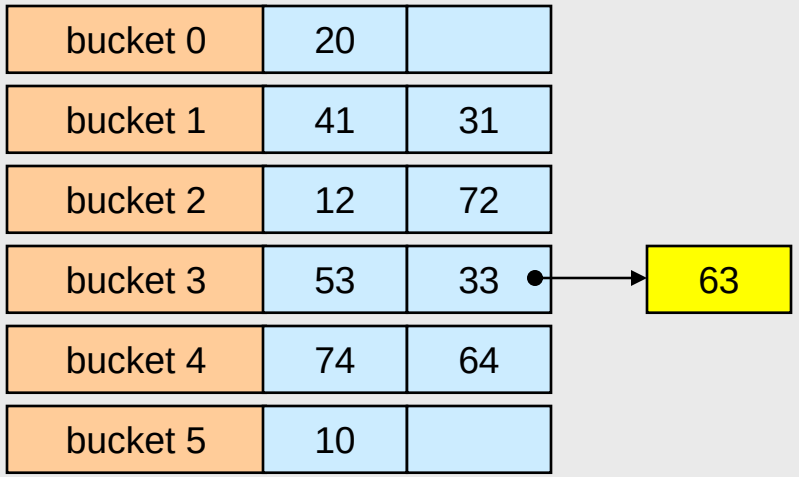
h0(12) = 2, h0(72) = 2

h0(41) = 1, h0(31) = 1

h1(10) = 5, h1(20) = 0

n = 1, 1 bucket (#0) has split.

After insert of 63



Linear Hashing Example: Insert Keys 40 and 52

A subsequent insertion of the key value 52 (hash to bucket 2), will cause an overflow from bucket 2 and a splitting of bucket 1 as shown below. Notice that the insertion of key value 40 did not cause any overflow.

After insert of 40 and 52

h0(74) = 4, h0(64) = 4

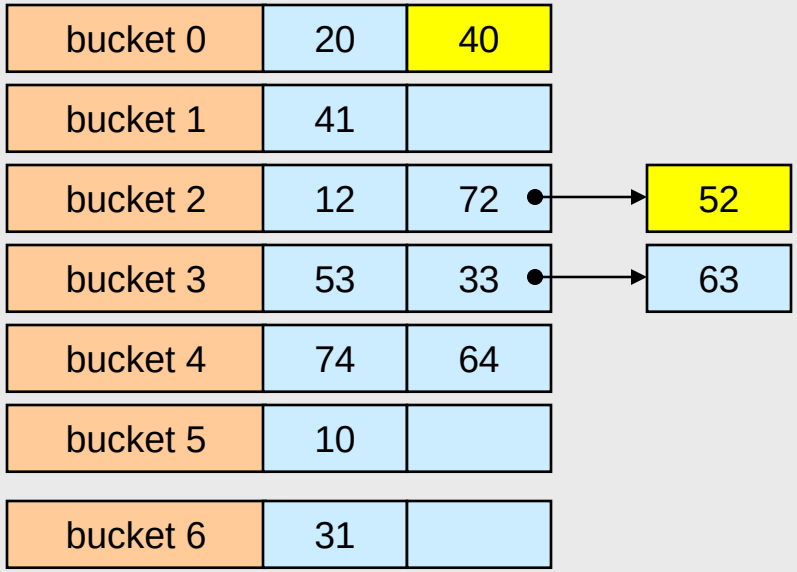
h0(53) = 3, h0(33) = 3, h0(63) = 3

h0(12) = 2, h0(72) = 2, h0(52) = 2

h1(41) = 1, h1(31) = 6

h1(10) = 5, h1(20) = 0, h1(40) = 0

n = 2, 2 buckets (#0 and #1) have split.



Linear Hashing Example

Notice at this point that although two buckets have split, neither have been buckets to which an insertion occurred causing an overflow.

The overflowing records which caused buckets 0 and 1 to split are still in their respective overflow chains.

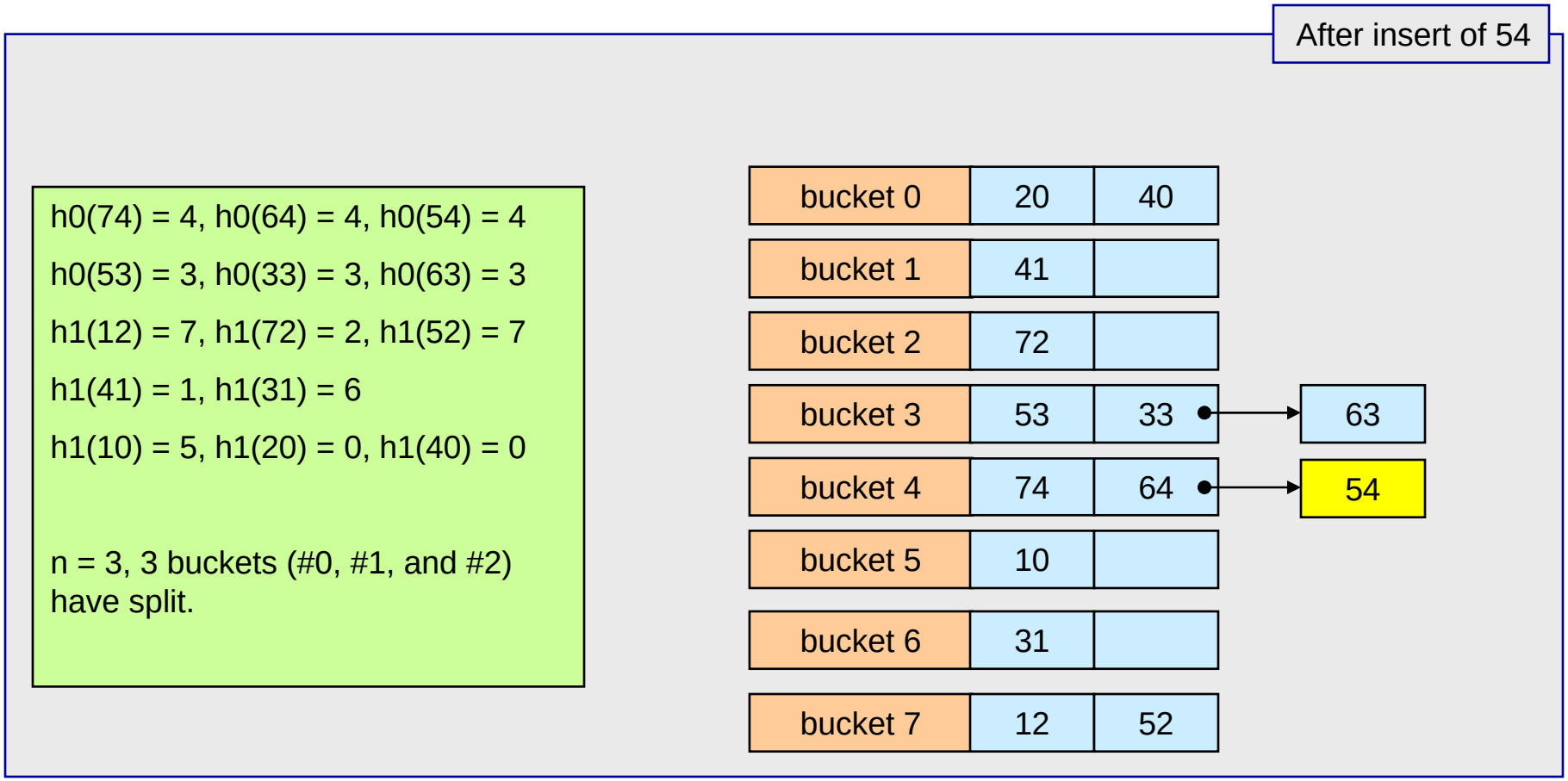
Notice too, that the insertion of key value 40 did not cause an overflow and thus no splitting of another bucket.

The next insertion that occurs which causes an overflow (notice that this insertion would not be to buckets 0, 1, 5 or 6) will cause the redistribution of records from bucket 2 including those in its overflow chain.

This is shown in the next diagram when key value 54 is inserted.

Linear Hashing Example: Insert Key 54

A subsequent insertion of the key value 54 (hash to bucket 4), will cause an overflow from bucket 2 and a redistribution of its records, including those in its overflow chain.



Linear Hashing Example:

Now let's assume that time has passed and more insertions have occurred to the file so that all of the original M buckets (0-4) have split. At this point every record in the file has been hashed according to hash function h1 and there are a total of 2M buckets in the file (0-2M-1 or 0-9). This situation is shown below.

After some period of time

h1(74) = 9, h1(64) = 4, h1(54) = 4, h1(84) = 9

h1(53) = 3, h1(33) = 3, h1(63) = 3

h1(12) = 7, h1(72) = 2, h1(52) = 7, h1(22) = 7

h1(41) = 1, h1(31) = 6

h1(10) = 5, h1(20) = 0, h1(40) = 0

n = 5, all five original buckets have split.

bucket 0	20	40
bucket 1	41	
bucket 2	72	
bucket 3	53	33
bucket 4	64	54
bucket 5	10	
bucket 6	31	
bucket 7	12	52
bucket 8	63	
bucket 9	74	84

Linear Hashing Example:

At this point, the file is twice as large (in terms of buckets) as it was initially and the value of n = M = 5. The hash function h1 applies to every record in the file and thus *n* is reset to 0 and the next insertion to cause an overflow will result in the next hash function h2 being used to hash the records from bucket 0 into two buckets, 0 and 2M. This is shown below on the insert of key value 23 to bucket 3 which causes an overflow.

h1(74) = 9, h1(64) = 4, h1(54) = 4, h1(84) = 9

h1(53) = 3, h1(33) = 3, h1(63) = 3

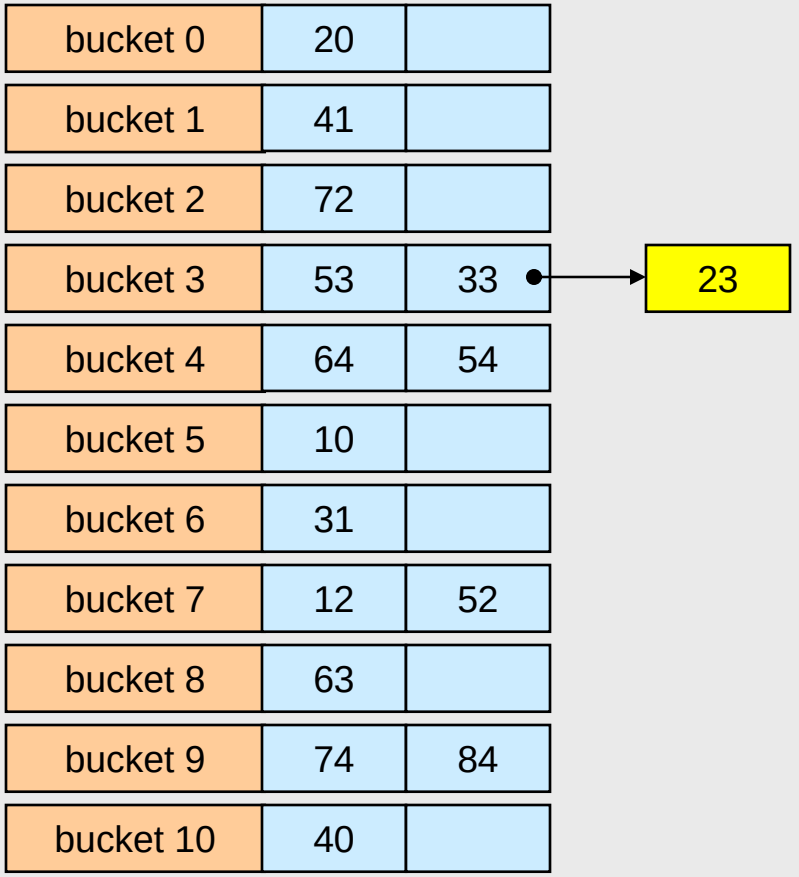
h1(12) = 7, h1(72) = 2, h1(52) = 7, h1(22) = 7

h1(41) = 1, h1(31) = 6

h1(10) = 5, h1(20) = 0, h1(40) = 10

n = 1, 1 bucket (#0) has split.

After some period of time



Linear Hashing

Buckets that have been split can also be merged back together if the loading of the file falls below a certain threshold. In general, the file load *L* can be defined as:

$$L = \frac{r}{bfr \times N}$$

where *r* is the current number of file records, *bfr* is the maximum number of records that can fit into a single bucket, and *N* is the current number of file buckets.

Blocks are combined linearly and *n* is decremented appropriately. In fact, the file load is typically used to trigger both splitting and contraction. Using this technique the file load can be kept within a desired range. Splits are triggered when the load exceeds a certain threshold, say 0.9, and contraction is triggered when the file load falls below a certain threshold, say 0.7.

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

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 balances of $1000; test *branch-name* = "Perryridge".
  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 Attributes

- Suppose we have an index on combined search-key (*branch-name*, *balance*).
- With the **where** clause
  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.