DATABASE DESIGN II - 1DL400

Spring 2014

A course on modern database systems

http://www.it.uu.se/research/group/udbl/kurser/DBII_VT14/indexes.pdf

Tore Risch
Uppsala Database Laboratory
Department of Information Technology, Uppsala University,
Uppsala, Sweden

Introduction to Indexing

Elmasri/Navathe ch 16 and 17
Padron-McCarthy/Risch ch 21 and 22

Tore Risch
Uppsala Database Laboratory
Department of Information Technology, Uppsala University,
Uppsala, Sweden

Database Design

- **Physical** Database Design:

  E.g by *indexes*:

  - Permit *fast* matching of records in table satisfying certain *search conditions* (predicates).
  - Critical for *scalable* access

PROBLEM:

- New applications may require data and index structures that are not supported by the DBMS.

  E.g. calendars, numerical arrays, geographical data, text, etc.

⇒ Extensible DBMSs needed where user can plug-in own indexing and search algorithms

Scalability

- DBMS is designed to handle *very large* amounts of data.

  => 10000 elements is considered small.

- DBMS should *scale*:

  => Performance should *not degrade* when the database *grows*.

- How to get scalability:

  => Need *index structures* that are maintained when database is updated.

  => Run on many *parallel* nodes.

Faster *response* but **more** resources often required!

Content

- Files of records
  - Operations on files
- Scans
  - Operations on scans
- Ordered and unordered files
  - Index sequential and hash files
- Index concepts
  - Types of indexes
  - Clustering indexes
  - Ordered indexes, B-trees, B+ trees
  - Unordered indexes, hash indexes
  - Index properties and evaluation metrics

Files of records

- A **file** managed by a DBMS is a *sequence* of records, where each record is a collection of data values representing a *tuple*.
- A **file descriptor** (or **file header**) includes
  - Meta-information that describes the file and the records it stores, such as the *attribute names* and *data types* of the *fields* in the records.
  - The records in a file are usually *uniform*, i.e. they are of the same size and contain the same kind of data values in each position.
  - File records are grouped into *blocks of records*. The file descriptor contains disk the addresses of the file blocks.
- The **blocking factor bfr** (or block size) for a file is the (average) number of file records stored in a disk block.

Operations on disk files

- Typical file operations include:
  - **OPEN**: Readies the file for access, and associates a pointer called a **cursor** that will refer to a **current** file record representing a current **tuple**.
  - **FIND**: Searches for the **first** record in a file that satisfies a certain condition, and makes it the cursor position.
  - **FINDNEXT**: Searches for the **next** record from the current cursor position that satisfies a certain condition, and makes it the current cursor position.
  - **READ**: Reads the record in the current cursor position into program variables.
  - **DELETE**: Removes the record at the current cursor position from the file, usually by marking the record to indicate that it is no longer valid.
  - **MODIFY**: Changes the values of some fields in the record of the current cursor position, i.e. copies values from program variables to the current record.
  - **INSERT**: Inserts a new record into the file and makes it the current cursor position., i.e. copies values from program variables into a new record and inserts it at the current cursor position.
  - **CLOSE**: Terminates access to the file.
  - **REORGANIZE**: Reorganizes the file records.
    - For example, the records marked deleted are physically removed from the file or batches of new records are merged into the file.

Streamed access to database operators

- The result of internal database **operators** (e.g. a join) is usually represented as **scans** (~streams) of tuples
  - The **cursors** represent positions in scans.
  - Cursors can be moved iteratively forward over the scans until end-of-scan is reached
- SQL queries are translated by the query optimizer into programs called **execution plans**.
- Execution plans call **physical relational algebra operators** that are programs that iterate over scans of tuples and iteratively produce new scans of tuples as results.
- Scans can be defined over
  - file records
  - index records
  - *Reverse scans* over files and indexes are also possible where scans move backwards
  - records produced (emitted) by some physical relational algebra operator.

Streamed access to database operators

- SCAN operators
  The following the three basic operators are defined over scans:
  - **OPEN**: Opens the scan for reading tuples and sets the cursor to the first tuple.
  - **NEXT**: reads the next tuple in the scan a into some program variables and makes it the current cursors position of the scan
  - **EOS**: true is there are no more tuples in the scan
  - **CLOSE**: Closes the scan and releases all its resources.
- Scans over physical files is represented by file pointers where a new read record is read for each NEXT call.
- Scans over the result of a physical operator is usually represented as iterator objects with *next*, *eos*, and *close* methods.
- Intermediate results may either be *materialized* as a list of blocks of tuples or *generated* by some code (e.g. performing a join) when *next* is called

Scan-based database operators

- Examples of physical algebra operators (code) using and producing scans:
  - **FINDALL**: emit all tuples in the result of a query. Such a scan operator call is e.g. produced by the query processor to iteratively emit the result of a query. Execution plans usually contain many FINDALL operators.
  - **STOPAFTER *n***.: iteratively emit the first n tuples in another scan
  - **DISTINCT:** iteratively emit the tuples of another scan where duplicate tuples are removed
  - **SORT**: emit the tuples of another in sorted orders
  - **INDEXSCAN**: Iterative emitting the tuples matching the key of an index
  - **REVESEINDEXSCAN**: Iteratively emitting matching index tuples in reverse order
  - **MATERIALIZE**: Store each tuple in a scan in a file.

Unordered files

- Also called a **heap** file.
- New records are inserted at the end of the file.
- A **linear search** through the file records is necessary to search for a record.
  - This requires reading and searching half the file blocks on the average, and is hence does not scale as the file grows.
- Record insertion is quite efficient.
- Reading the records in order of a particular field requires sorting the file records, which does not scale.

Ordered files

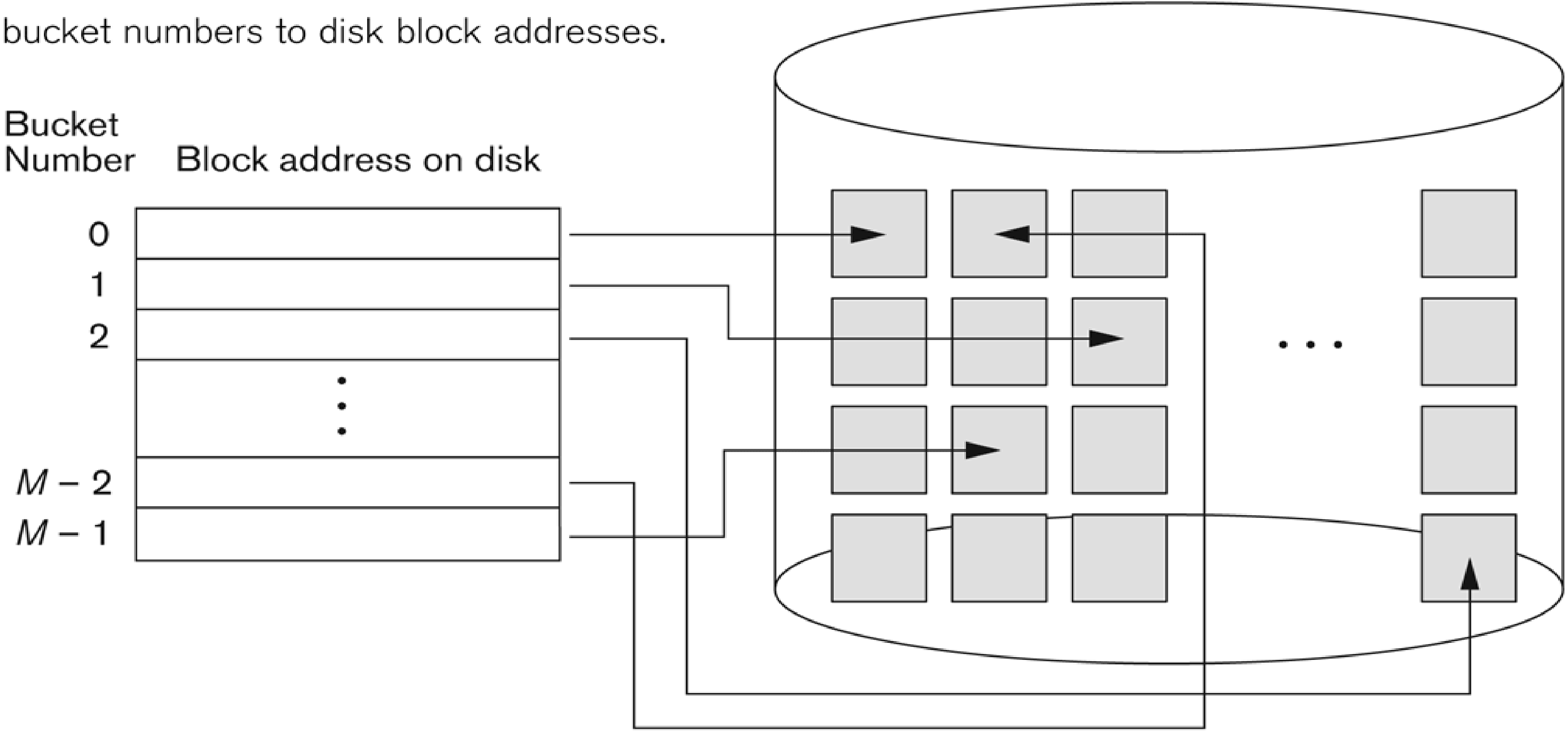
- Also called a **index sequential (ISAM)** file.
- File records are kept sorted by the values of an **ordering field**.
- Insertion is rather expensive: records must be inserted in the correct order.
  - It is common to keep a separate unordered **overflow** (or **transaction**) file for **batching** new records to improve insertion efficiency; this is periodically merged with the main ordered file.
- A **binary search** can be used to search for a record on its ordering field value.
  - This requires reading and searching $\log_2$ of the file blocks on the average, an improvement over linear search.
- Reading the records in order of the ordering field is quite efficient.
- Efficiency is measured in # of read disk blocks.

Hash files

- Hashing for disk files is called **External Hashing**
- The file blocks are divided into M equal-sized **buckets**, numbered bucket_0, bucket_1, ..., bucket_{M-1}.
  - Typically, a bucket corresponds to one disk block in a file.
  - Each bucket represents one or several records (i.e. tuples)
- One of the fields of the records is designated to be the **hash key** of the file.
- The record with hash key value K is stored in bucket i, where $i = h(K)$, and h is the **hashing function**.
- Search and update is very efficient on equality of the hash key.
- **Collisions** occur when a new record hashes to a bucket that is already full.
  - An overflow file is kept for storing such records.
  - Overflow records that hash to each bucket can be linked together.
- Main **disadvantages** of static external hashing:
  - *Fixed* number of buckets M is a *problem* if the number of records in the file grows or shrinks.
  - *Ordered* access on the hash key is quite *inefficient* (requires sorting the records).

Hash files (contd.)

Figure 13.9
Matching bucket numbers to disk block addresses.



Hash files - overflow handling

- Methods for collision resolution include **chaining**:
  - Overflow locations are kept, usually by extending the bucket with **overflow positions**.
  - In addition, a pointer to a chain of **overflow records** is added to each bucket.
  - A **collision** is resolved by placing the new record and its key in an unused overflow location

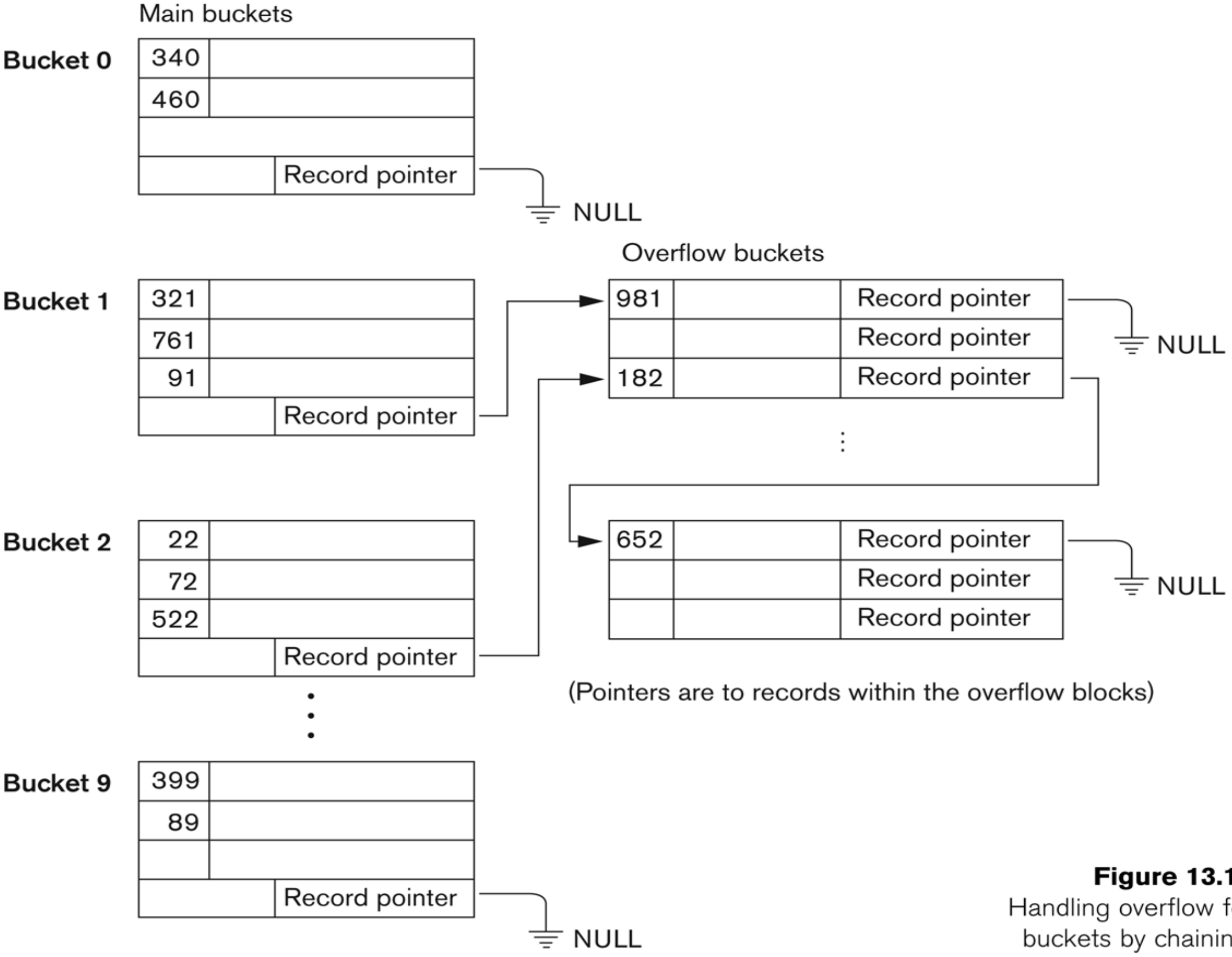


Figure 13.10
Handling overflow for buckets by chaining.

Basic index concepts

- **Indexes** are data structures used to speed up access to sets of records stored in a database (on disk or in main memory).
  - E.g., author catalog in library
- An **index** consists of records, called **index entries**, of the form

key	value

- The **key** of an index is the attribute(s) of the indexed set of tuples used to look up records, e.g. SSN, ISBN.
  - The key is a record of one or several key values, which are usually stored directly in the index entry (e.g. SSN + ACCOUNT#).
- The **value** is a tuple that stores the corresponding data values
  - The value field is usually a pointer to a data record storing the values
- Two basic kinds of indices:
  - **Ordered indexes:** search keys are stored in sorted order
  - **Hash indexes:** search keys are distributed randomly across “buckets” using a “hash function”.

Types of indexes

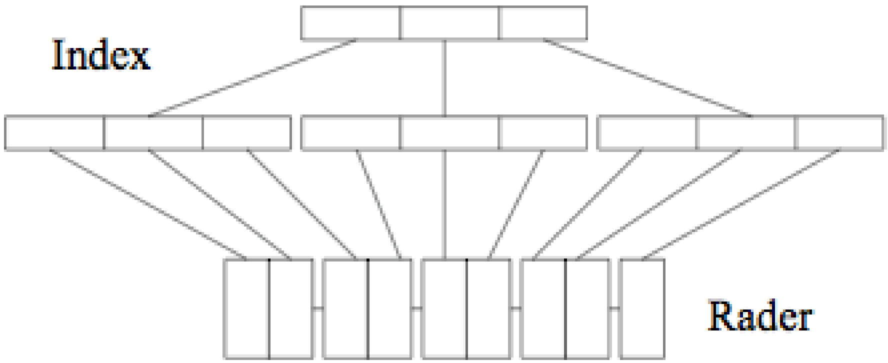
- **Primary Index**
  - Defined on an ordered data file
  - The data file is ordered on one ore several **key field(s)**
  - Includes one index entry **for each block** in the data file; the index entry has the key field value for the **first record** in the block, which is called the **block anchor**
    - This makes primary indexes very **compact**
- **Clustering Index**
  - Defined on an ordered data file
  - The data file is ordered on one or several **non-key field(s)** unlike the primary index, which requires that the ordering field of the data file has a distinct value **for each distinct value** of the index.
  - The index value for each distinct value of the search key points to the first data block that contains records with that field value.
    - For example, think on an index of four character string used to index a files ordered on long strings.

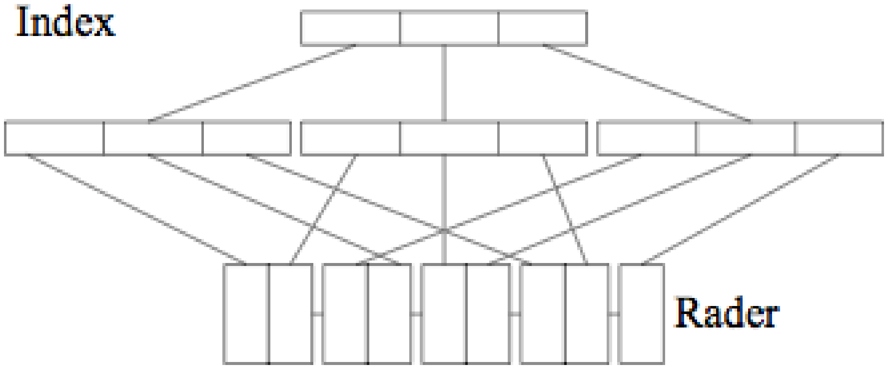
Types of indexes (cont.)

- **Secondary Index**
  - A secondary index provides a secondary means of accessing a file for which some primary access already exists.
  - It is a non-clustering index since the indexed records are not ordered by the index keys
  - Retrieving all records pointed to from a secondary index can be very slow if the table is large.
- **Unique index**
  - A **unique** index contains key thus having a single unique value for each key
  - A **multiple** index indexes a non-key position and has a set of values for each key value.

Clustered indexes

- clustered index
- Non-clustered index





More index concepts

- The index is often specified on one attribute of the indexed tuples, but can also be specified on several attributes.
- The index is sometimes called an **access path** to the indexed attribute.
- A search over an index yields a **scan** whose cursor points to file records
- **Scans** over ordered indexes are usually represented data structures representing upper and lower limits of key values in a search along with the cursor
- Indexes can also be characterized as dense or sparse:
  - A **dense index** has an index entry for every search key value (and hence every record) in the data file. A secondary index is usually dense.
  - A **sparse (or nondense) index**, on the other hand, has index entries for only some of the search values. A primary index is usually sparse.

Ordered indexes

- Most ordered indexes use the highly scalable *B-tree* data structure (Bayer, Acta Informatica 1(2), 1972)
- B-trees are automatically *rebalanced* trees with *many* children for each node (large fan-out)
- Each B-tree node occupies one disk block.
  - One disk block at the time is read into main memory by the DBMS
  - The DBMS maintains a **pool** of disk blocks in main memory
  - When pool is full disk blocks are **flushed** to disk
- In main memory each B-tree node should have a size close to the **cache line** size used, to avoid memory cache misses.

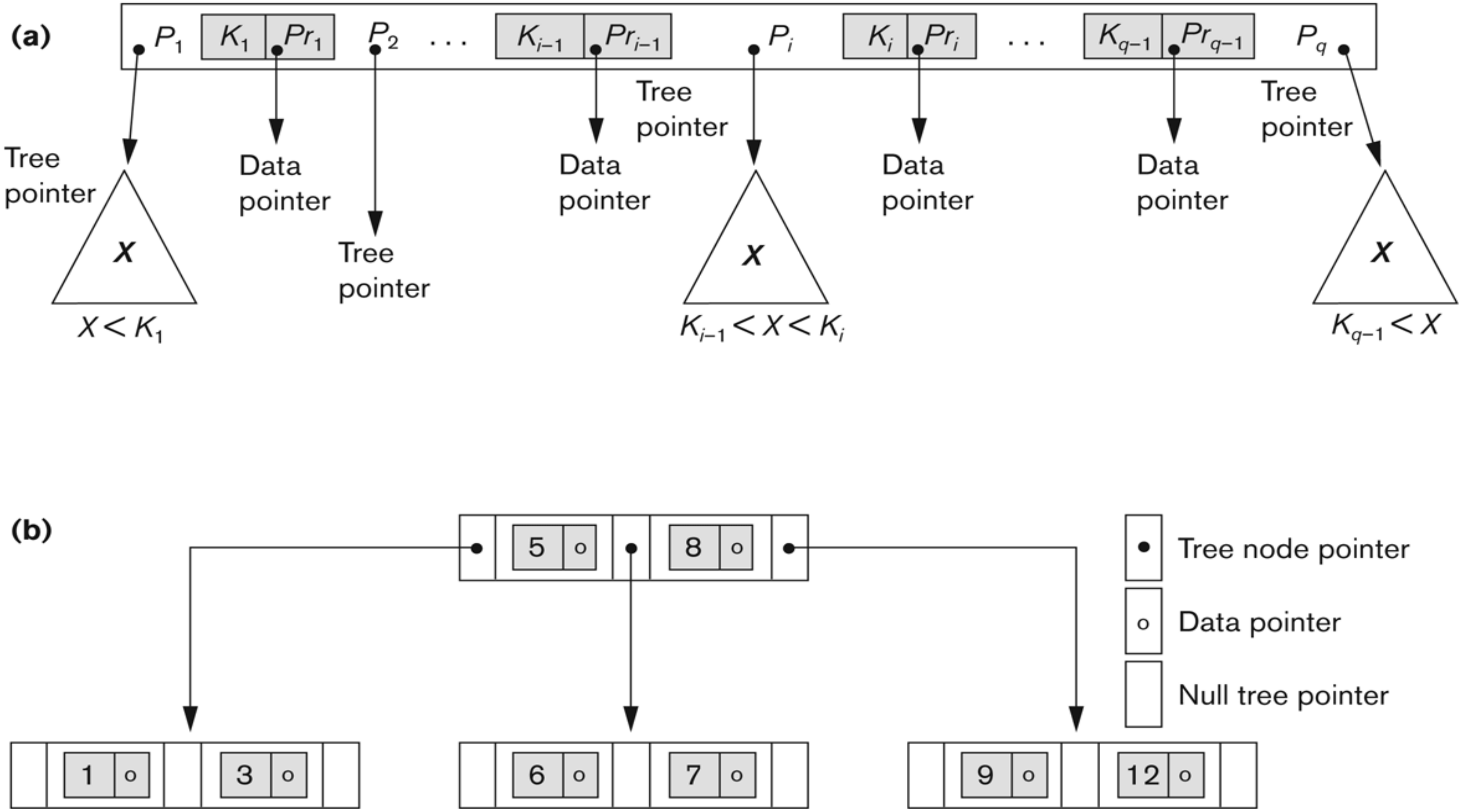
B-tree indexes

- Each node is kept between half-full and completely full
- An insertion into a node that is not full is quite efficient
  - Just fill an empty key/value slot in the node
- If a node is full the insertion causes a split into two nodes
- Splitting may propagate to neighboring tree levels
- A deletion is quite efficient if a node does not become less than half full
- If a deletion causes a node to become less than half full, it must be merged with neighboring nodes
- On the average the nodes are 63% full
- B-trees also shown excellent in modern main memories with big differences in speed between data in caches and in the rest of the memory (G. Graefe & P-Å. Larsson, *B-tree Indexes and CPU Caches*, ICDE 2001).

B-tree Structures

Figure 14.10

B-Tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.



B-trees

- Nodes in a B-tree are **uniform**:

  ((key, value, subtree) …)
- Each node in a B-tree contains an array of key/value pairs and pointers to subtrees.
- There is no difference between leaf nodes and intermediate nodes.
- Assume there are 10^8 tuples in the index
- Assume a block size of 8192 and that each B-tree node is on the average 63% full.
- Assume each (key,value,subtree) triple uses 12 bytes.

  => $8192*0.63/12 = 430$ = triples per node

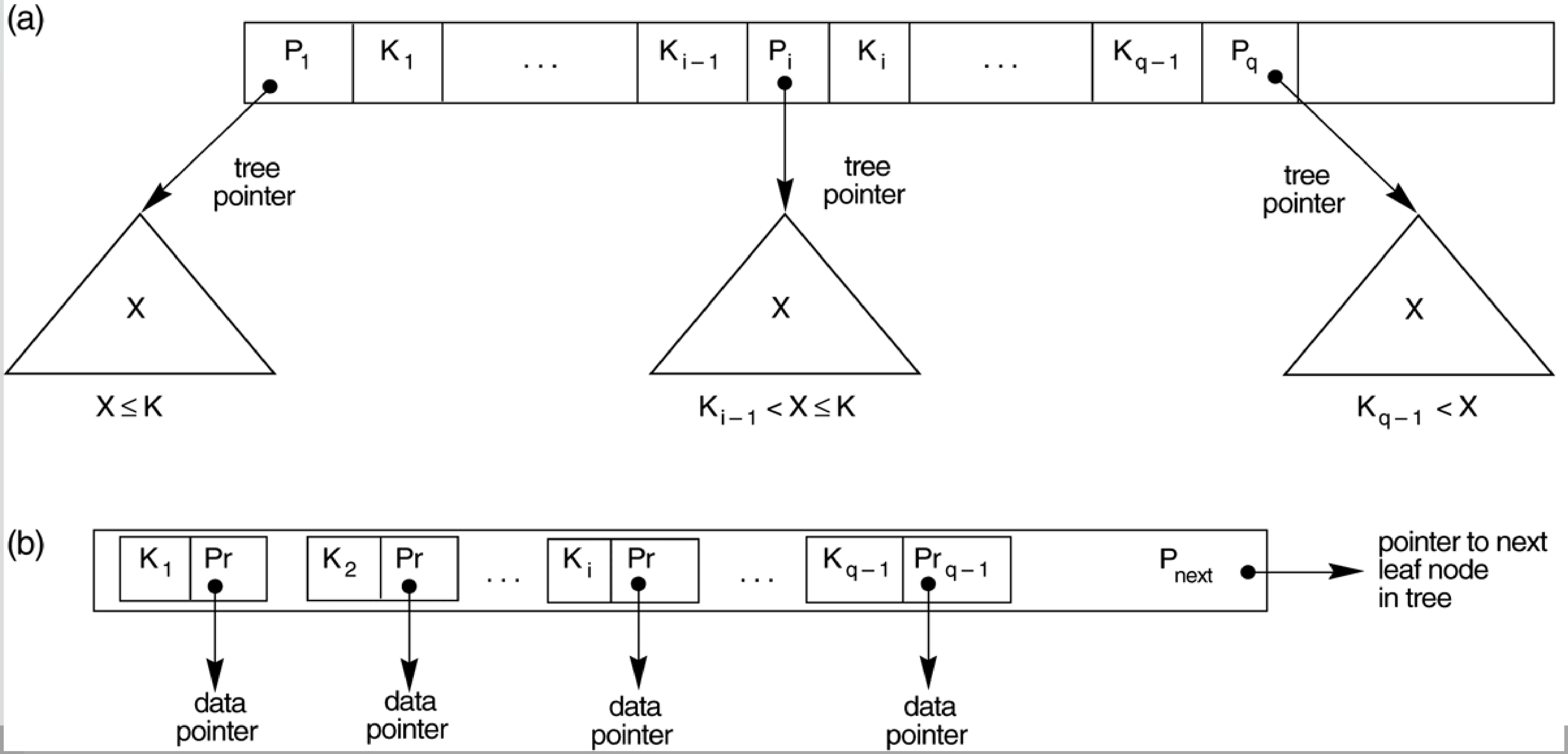
  => The average depth of the B-tree will be $\log_{430}(10^8)=3.0$

  => A key/value pair can be accessed with 3 block reads (disk accesses) on the average

  Root node kept in main memory => 2 blocks to read

The Nodes of a B+-tree

- FIGURE 14.11 The nodes of a B+-tree
    - (a) Internal node of a B+-tree with q –1 search values.
    - (b) Leaf node of a B+-tree with q – 1 search values and q – 1 data pointers.



B^+-trees

- In a B^+-tree, **intermediate nodes** contain keys + pointers to subtrees (no values)

  Intermediate nodes: ((key, subtree) …)
- In a B^+-tree, **only** the leaf-level nodes contain pointers to data records:

  Leaf nodes: ((key, value) ….).

  The leaf nodes are linked to enable *fast scans*.
- Because there are fewer data pointers in B+-trees, the fan-out (average number of children) becomes larger with B^+-trees than with B-trees.
- Assume a block size of 8193 and that each B-tree node is on the average 63% full.
- Assume each (key,value) or (key,suptree) pairs uses 8 bytes.

  => 8192*0.63/8 = 645 = pairs per node

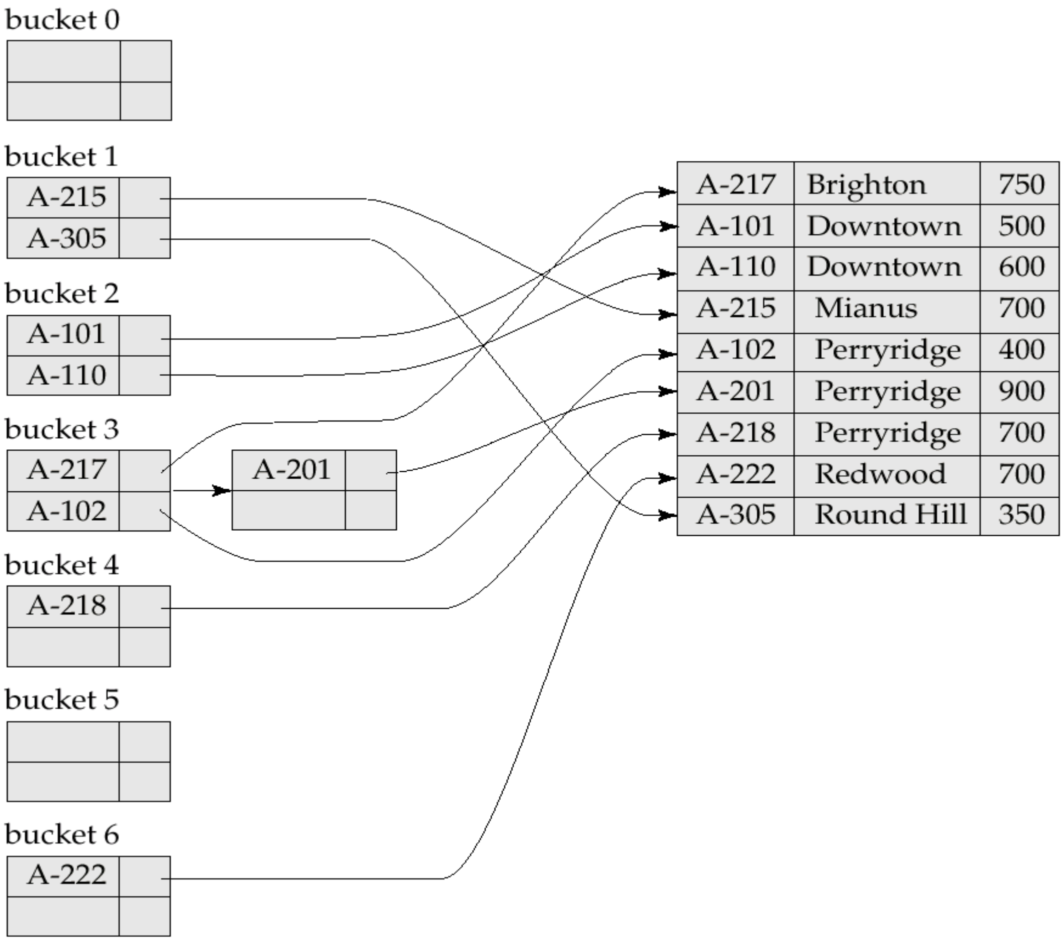
  => The average depth of the B-tree will be $\log_{645}(10^8)=2.8$

  => A key/value pair can be accessed with 2.8-1=1.8 block reads on the average

Hash indexes

- **Hash indexes** organizes the keys with their associated value pointers, into a hash table structure.
- Hash indexes are very fast for accessing a value (or a set of values) for a given key.
- Hash indexes do **not** support range searches.
- Hash indexes require **dynamic hash tables** that **gracefully** grow or schrink without significant delays as the database is updated
- Regular hashing problematic when files grow and shrink dynamically
  - Dynamic and graceful **scale-up** and **scale-down** of tables needed in DBMSs
  - Special hashing techniques have been developed to allow in incremental and dynamic growth and shrinking of the number of file records in hash files.
  - The most used one is called **LH, Linear Hashing** (Litwin, VLDB 1980)
  - Linear Hashing is also shown to be preferable when storing dynamic hash tables in main memory (P-Å Larson, *Dynamic Hash Tables*, CACM 31(4), 1988).

Example of a hash index



Overview of LH

- Dynamic hash algorithm
- A hash file has buckets with some capacity $b >> 1$
- Not one hash functions but a *series* of hash functions $h_i (k)$ of keys k as the hash file evolves.
- Typically hash by division $h_i (k) = k \bmod 2^i$, $i=1,2,3,4,\ldots.$
- Buckets split through the replacement of h_i with h_{i+1} ; $i = 0,1,..$ as the file grows
- As the file grows and the load (number of keys) of the hash table thereby increases beyond some threshold, buckets are successively split. For split buckets *b/2* keys move towards new buckets.
- Shrinking the files is similarly possible by joining buckets when the table load decreases below some threshold.

LH File Evolution


35

12

7

15

24

0

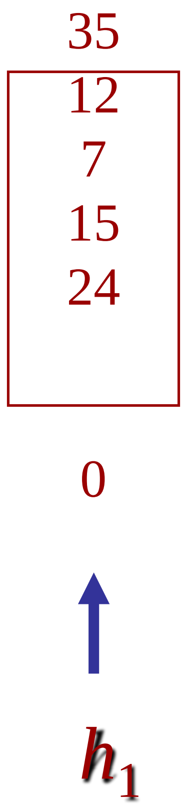
$b = 4$

$i = 0$

$p = 0$

$h_0(k) = k \bmod 2^0$

LH File Evolution



35

12

7

15

24

0

h_1

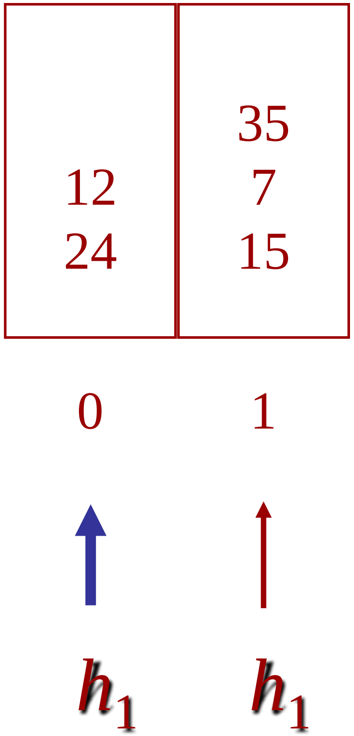
$b = 4$

$i = 0$

$p=0$

$h_0(k)=k \bmod 2^0$

LH File Evolution



12
24

35
7
15

0 1

h_1 h_1

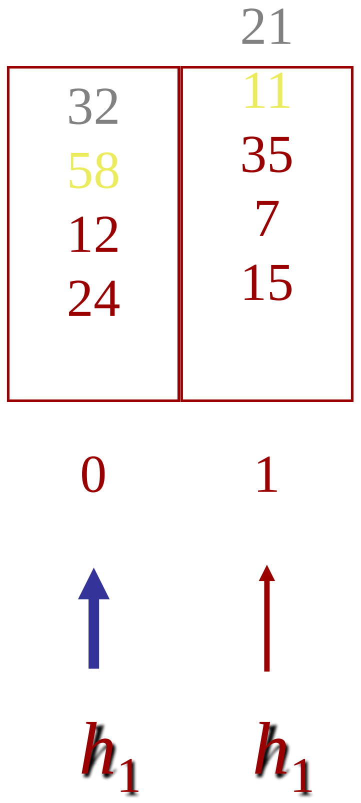
$b = 4$

$i = 1$

$p=0$

$h_1(k)=k \bmod 2^1$

LH File Evolution



21

32
58
12
24

11
35
7
15

0 1

h_1 h_1

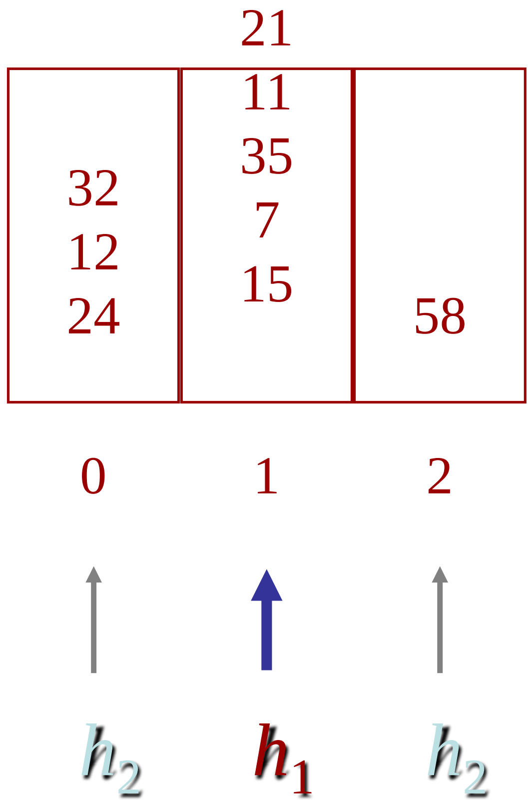
$b = 4$

$i = 1$

$p=0$

$h_1(k)=k \bmod 2^1$

LH File Evolution



21

32 12 24

11 35 7 15

58

0 1 2

h_2 h_1 h_2

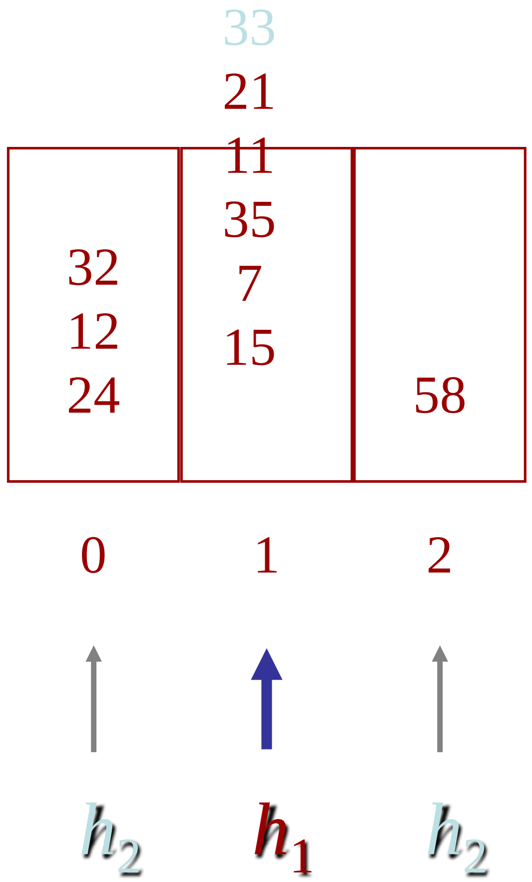
$b = 4$

$i = 1$

$p=1$

$h_2(k)=k \bmod 2^2$

LH File Evolution


33

21

11

35

7

15

32

12

24

58

0 1 2

h_2 h_1 h_2

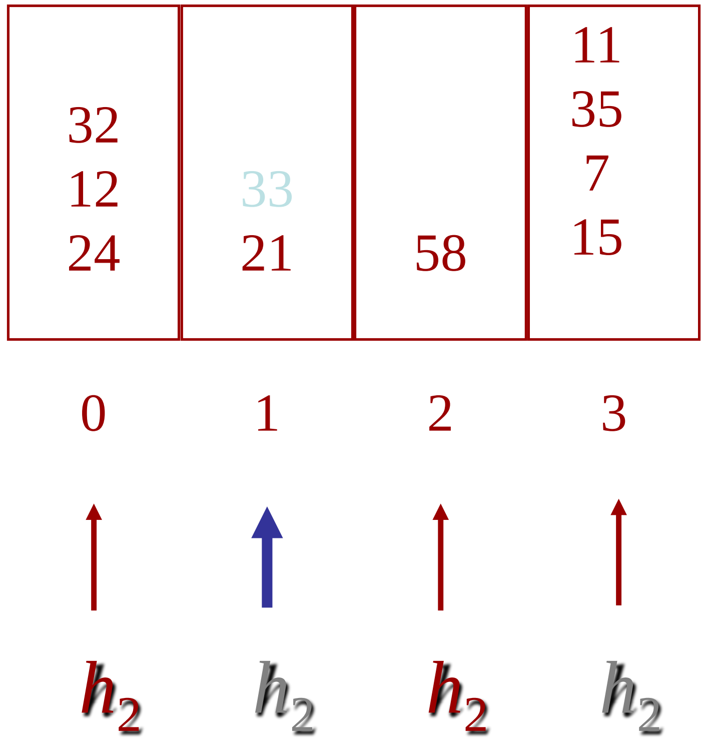
$b = 4$

$i = 1$

$p=1$

$h_2(k)=k \bmod 2^2$

LH File Evolution



0	1	2	3
32 12 24	33 21	58	11 35 7 15

h_2 h_2 h_2 h_2

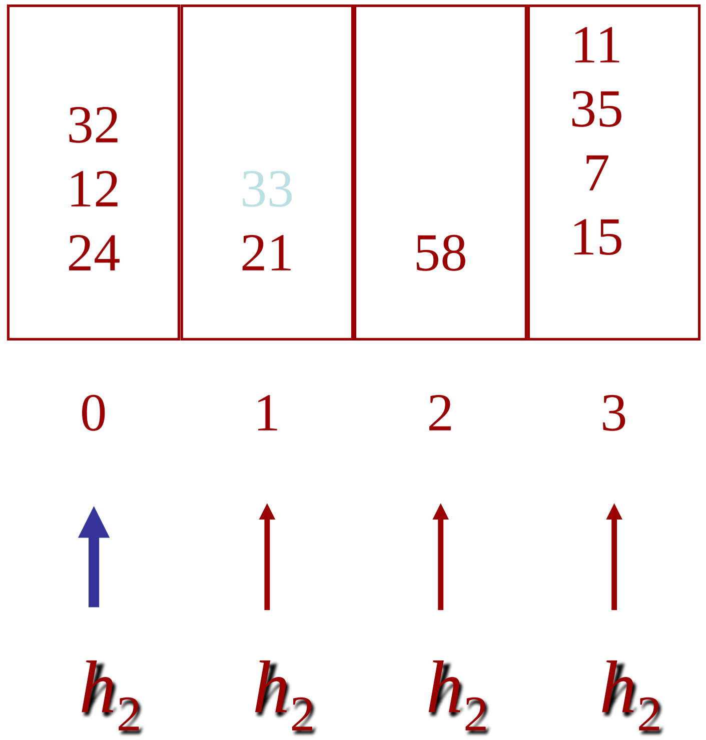
$b = 4$

$i = 1$

$p=1$

$h_2(k)=k \bmod 2^2$

LH File Evolution



0	1	2	3
32 12 24	33 21	58	11 35 7 15
h_2	h_2	h_2	h_2

$b = 4$

$i = 2$

p=0

$h_2(k)=k \bmod 2^2$

Main-memory LH

- LH shown excellent also for main memory hash tables (P-Å Larson 1988).
- No need for specifying size when hash table allocated
- Dynamic grow and shrink
- No buckets but just overflow chains possible in main memory (i.e. b=1).
  - As for B-tree a bucket size close to cache line size is preferred.
- When say 200% full (twice as many keys as size of table) after insert move p forward and add bucket
- When say 50% full after delete move p backwards and delete bucket
- Problem: Need dynamically extensible array to hold hash table

Index evaluation metrics

- **Access operations** supported efficiently. e.g.,
  - Records with an explicitly specified value in the attribute (efficient get/put)
  - Records with an attribute value falling in a specified range of values (range search).
- **Access time** as the number of records grow
- **Insertion time** as the number of records grow
- **Deletion time** as the number of records grow
- **Space overhead** of representing the index
- **Scalable dynamic behavior**: The index should not make the DBMS behave unpredicably, such as stopping for reorganization as the number of records grows or shrinks.