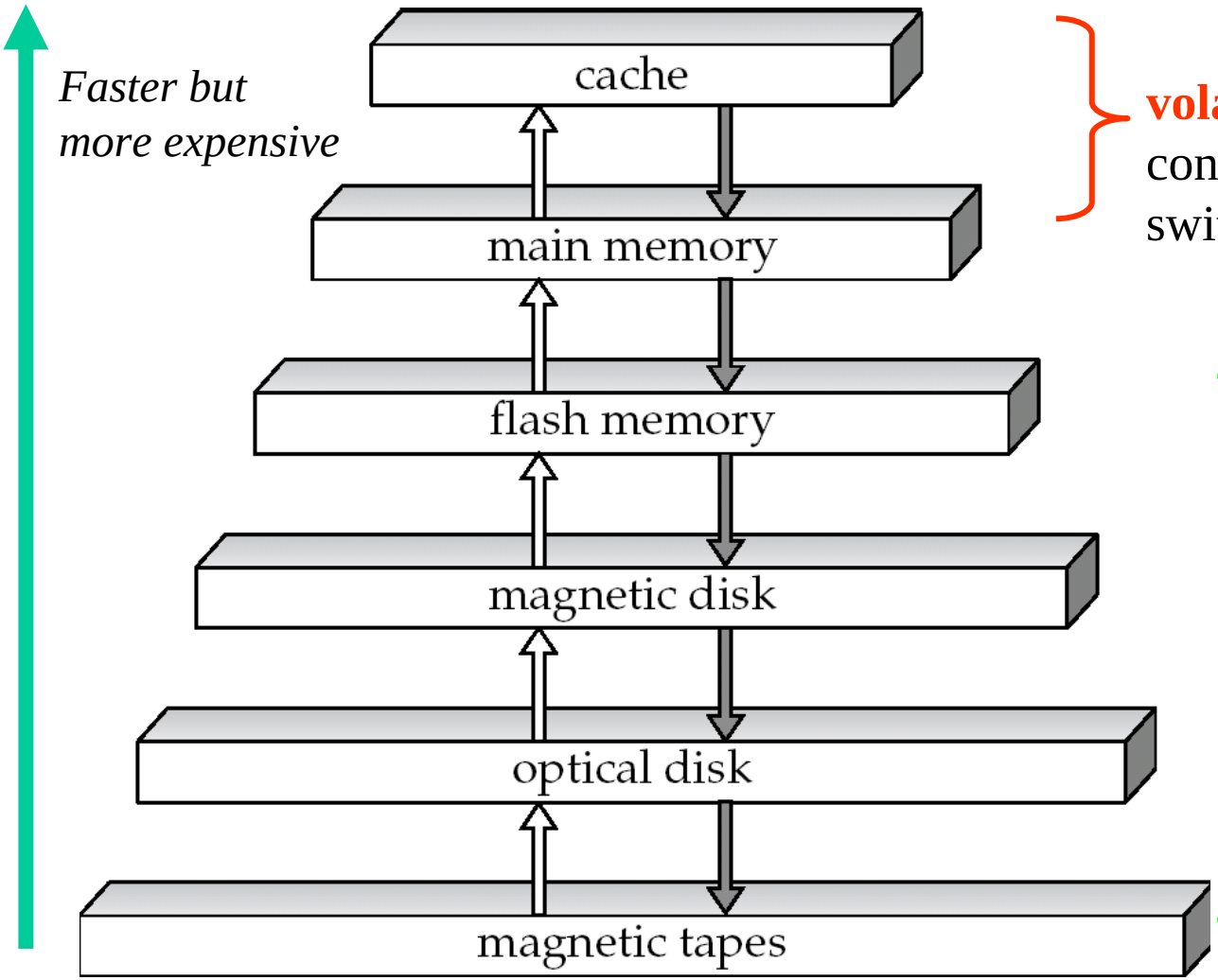
Physical Database Design: Outline

- File Organization
  - Fixed size records
  - Variable size records
- Mapping Records to Files
  - Heap
  - Sequentially
  - Hashing
  - Clustered
- Buffer Management
- Indexes (Trees and Hashing)
  - Single-level versus multi-level
  - B^+-Trees
  - Static Hashing
- Tuning Physical Design

Physical Database Design

- File Organization
  - Fixed size records
  - Variable size records
- Mapping Records to Files
- Buffer Management
- Indexes (Trees and Hashing)
- Tuning Physical Design

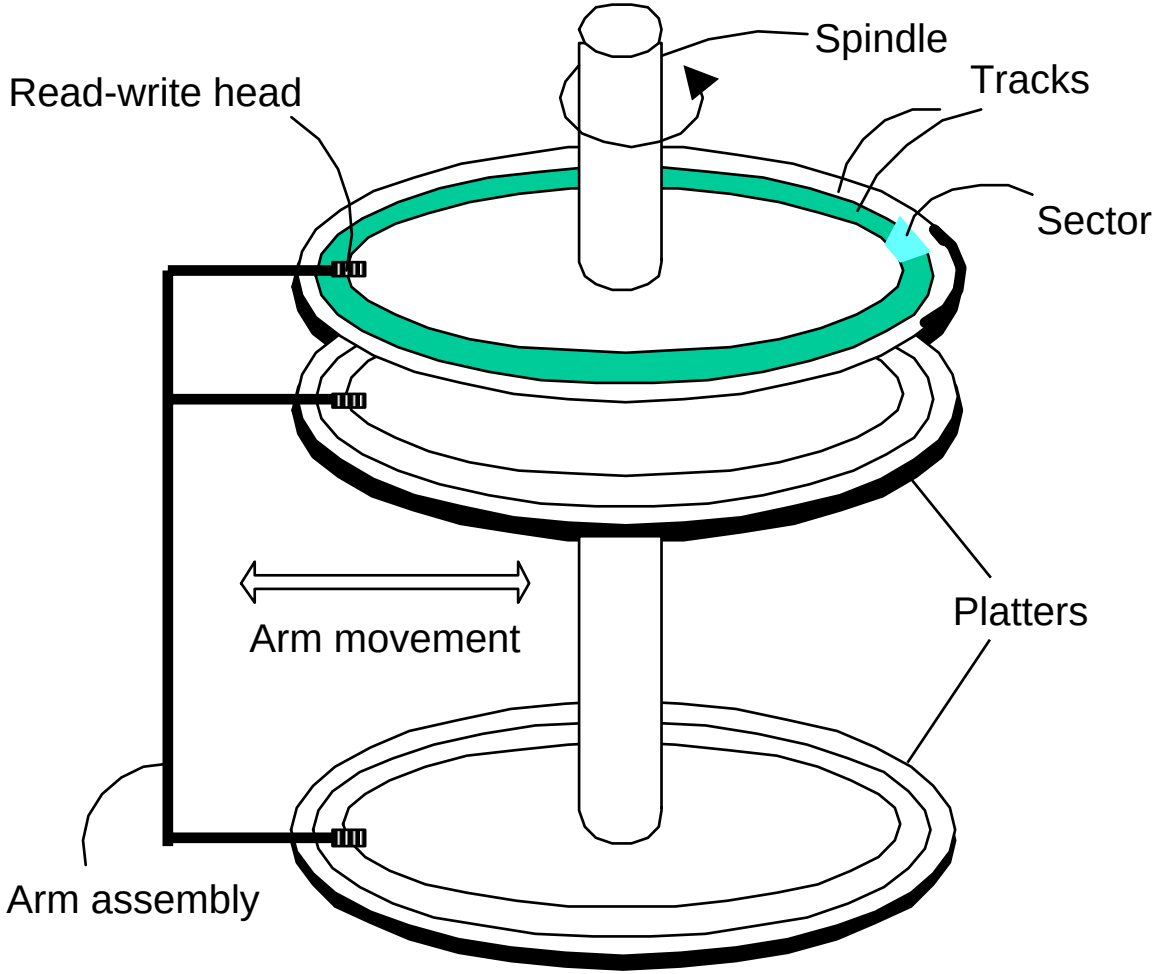
Storage Hierarchy



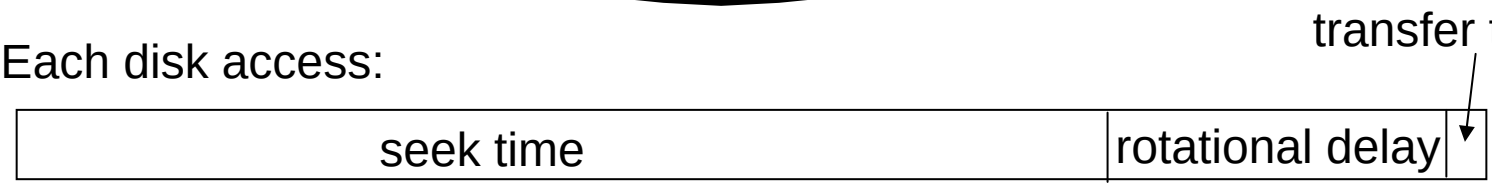
volatile storage: loses contents when power is switched off

non-volatile storage:
Contents persist even when power is switched off.

Magnetic Disk Access



Each disk access:



Optimization of Disk Block Access

- Each track is divided into **sectors.**
  - A sector is the smallest unit of data that can be read or written.
- **Block** – a contiguous sequence of sectors from a single track
  - data is transferred between disk and memory in blocks (I/O)

- Optimization of disk block access
  - Optimize block access time by organizing the blocks to correspond to how data will be accessed
  - E.g. Store related information on the same or nearby cylinders.
    - To reduce the time-consuming track seeking

The Problem of Physical Design

- Conceptual relations are sequences of bits on disk.
- Functionality Requirements
  - Be able to sequentially process records.
  - Be able to search for key-values efficiently.
  - Be able to insert and delete records.
- Performance Objectives
  - Achieve a high packing density (little wasted space).
  - Achieve fast response time
  - Support a high volume of transactions.

File Organization

- The database is stored as a fixed collection of *files*. Each file is a set of *records*.
- A record is a sequence of *fields*.

- Record size
  - Fixed
  - Variable

- Files reside on mass storage, usually a disk.
  - Fast random access
  - Non-volatile storage

Fixed Size Records

- Record access simple: to access record *i*, get record at location $n \times i$.
- Deletion of record *i*
  - Move records $i+1, \ldots, n$ to $i, \ldots, n-1$.
    - This is costly in time.
    - Need to worry about auxiliary pointers to the moved records.
  - Move record *n* to *i*.
    - This is fast.
    - Need to worry only about pointers to record *n*.
- Link all free records on a free-list.

Variable Size Records

- Example: *Film* and *Performers*

Toy Story	Tom	Tim	#		
Lion King	Simba	Mufasa	Scar	Timon	#
True Lies	Arnold	Jamie	#		

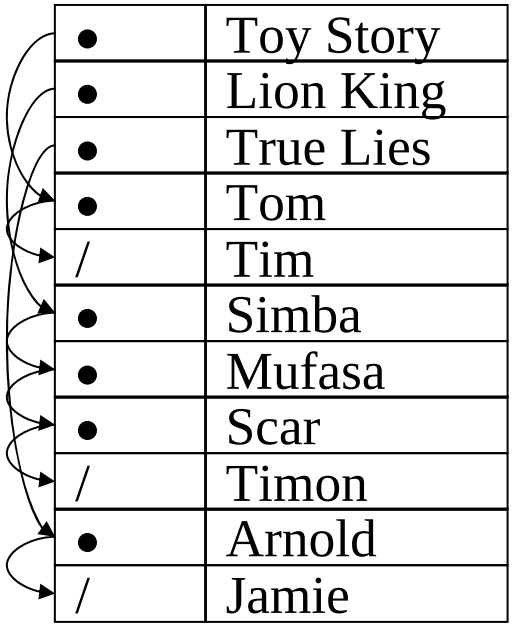
- Record access is difficult.
- Implementation alternatives
  - Can implement variable-size records with one byte record size, using “end-of-record” control character.
    - Difficulty with deletion.
    - Difficulty with growth, e.g., another performer was added to the Lion King

Variable Size Records, cont.

- Use fixed size records
  - If maximum size is known (at most four performers per film)

Toy Story	Tom	Tim	-	-
Lion King	Simba	Mufasa	Scar	Timon
True Lies	Arnold	Jamie	-	-

  - If maximum size is not known



Physical Database Design

- File Organization
- Mapping Records to Files
  - Heap
  - Sequentially
  - Hashing
  - Clustered
- Buffer Management
- Indexes (Trees and Hashing)
- Tuning Physical Design

Mapping Records to Files

- Ways to organize records in a file
  - Heap (unordered)
    - A record can be placed anywhere in the file where there is space
  - Sequential (sorted on one or more attributes)
    - Store records in sequential order, based on the value of the search key of each record
  - Hash (hash the record to a block based on a hash key)
    - A hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed
  - Clustering (Store one or more tables based on a cluster key)
    - Records with the same or similar key values are stored close to each other

Heap Example

- Heap Example
  - *Customer*(*Name*, *Film*, *ResDate*)
- Search by scanning the file
- Insertion of (‘Hans’, ‘ET’, 2006-04-22) is straight forward
  - To find a free slot

Merrie	True Lies	2006-03-29
Melanie	True Lies	2006-04-19
Eric	Lion King	2006-04-18
Melanie	Lion King	2006-02-21
Eric	Toy Story	2006-04-19

Sequential Examples

- Sorted on one or more attributes
  - *Customer*(*Name*, *Film*, *ResDate*)
  - Search on *Name*: log_2 (size)
  - Insertion of (‘Hans’, ‘ET’, 2006-04-22) cause the file to be reorganized.

Eric	Toy Story	2006-04-19
Eric	Lion King	2006-04-18
Melanie	True Lies	2006-04-19
Melanie	Lion King	2006-02-21
Merrie	True Lies	2006-03-29

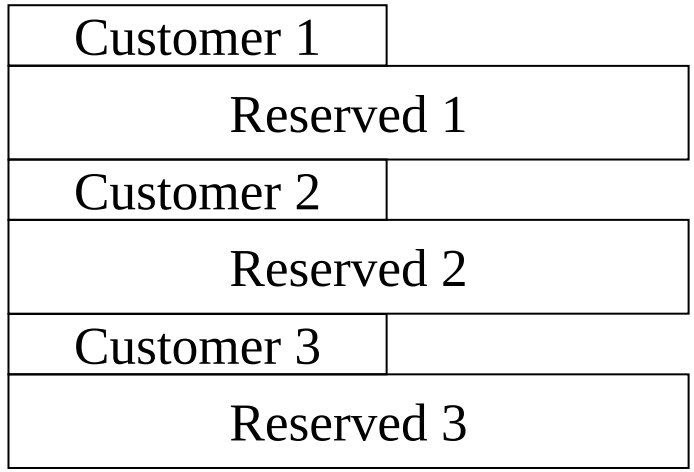
Clustering Examples

- *Customer*(*Name*, *Film*, *ResDate*)
- Keep all information concerning a customer and her reservations in the same block (or chain of blocks).
- In case a block is too small to hold all accounts, a new block is allocated and chained to the previous appropriate blocks.

Melanie	
True Lies	2006-04-19
Lion King	2006-02-21
Eric	
Toy Story	2006-04-19
Lion King	2006-04-18
Merrie	
True Lies	2006-03-29

Mapping Relations to Files

- Store each relation in a separate file.
  - Good for queries in which all attributes in a tuple need to be examined.

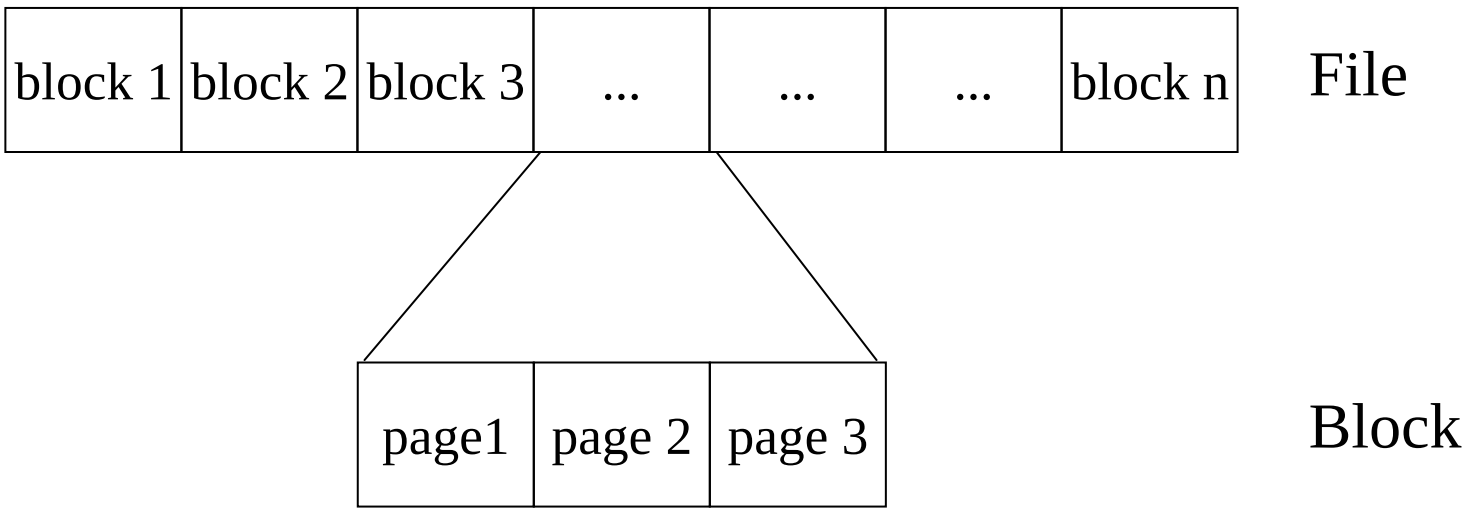


- Store several relations in one file.
  - Good for
    - queries involving *Customer* ⋈ *Reserved*
    - queries involving a single customer and her reservations.
  - Bad for queries involving only customers.
  - Variable size records complicate storage management.

Packing Records Into Blocks

- *Goal*: minimize I/O transfer in processing queries.

- *Idea*: pack the information that is likely to be used in a single query in the same block.



Physical Database Design

- File Organization
- Mapping Records to Files
- Buffer Management
- Indexes (Trees and Hashing)
- Tuning Physical Design

Buffer Management

- **Buffer –** portion of main memory available to store copies of disk blocks. It can reduce I/O transfer.
- **Buffer manager –** subsystem responsible for allocating buffer space in main memory.
- Programs call on the buffer manager when they need a block.
  1. If the block is already in the buffer, buffer manager returns the address of the block in main memory
  2. If the block is not in the buffer, the buffer manager
     1. Allocates space in the buffer for the block
        1. Replacing (throwing out) some other block, if required, to make space for the new block.
        2. Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
     2. Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester.

Buffer-Replacement Policies

- Most systems replace the block **least recently used** (LRU strategy)
- Idea behind LRU – use past pattern of block references as a predictor of future references
- Queries have well-defined access patterns (such as sequential scans), and a database system can use the information in a user's query to predict future references
  - LRU can be a bad strategy for certain access patterns involving repeated scans of data
    - For example: when computing the join of 2 relations r and s by a nested loops
      for each tuple *tr* of *r* do
        for each tuple *ts* of *s* do
          if the tuples *tr* and *ts* match …
  - Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable

Buffer-Replacement Policies, cont.

- **Pinned block** – memory block that is not allowed to be written back to disk at this point of time.
- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
- **Most recently used** (MRU) strategy – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.
- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
  - E.g., the data dictionary is frequently accessed. Heuristic: keep data-dictionary blocks in main memory buffer
- Buffer managers also support forced output of blocks for the purpose of recovery
  - force a memory block to be written back to disk.

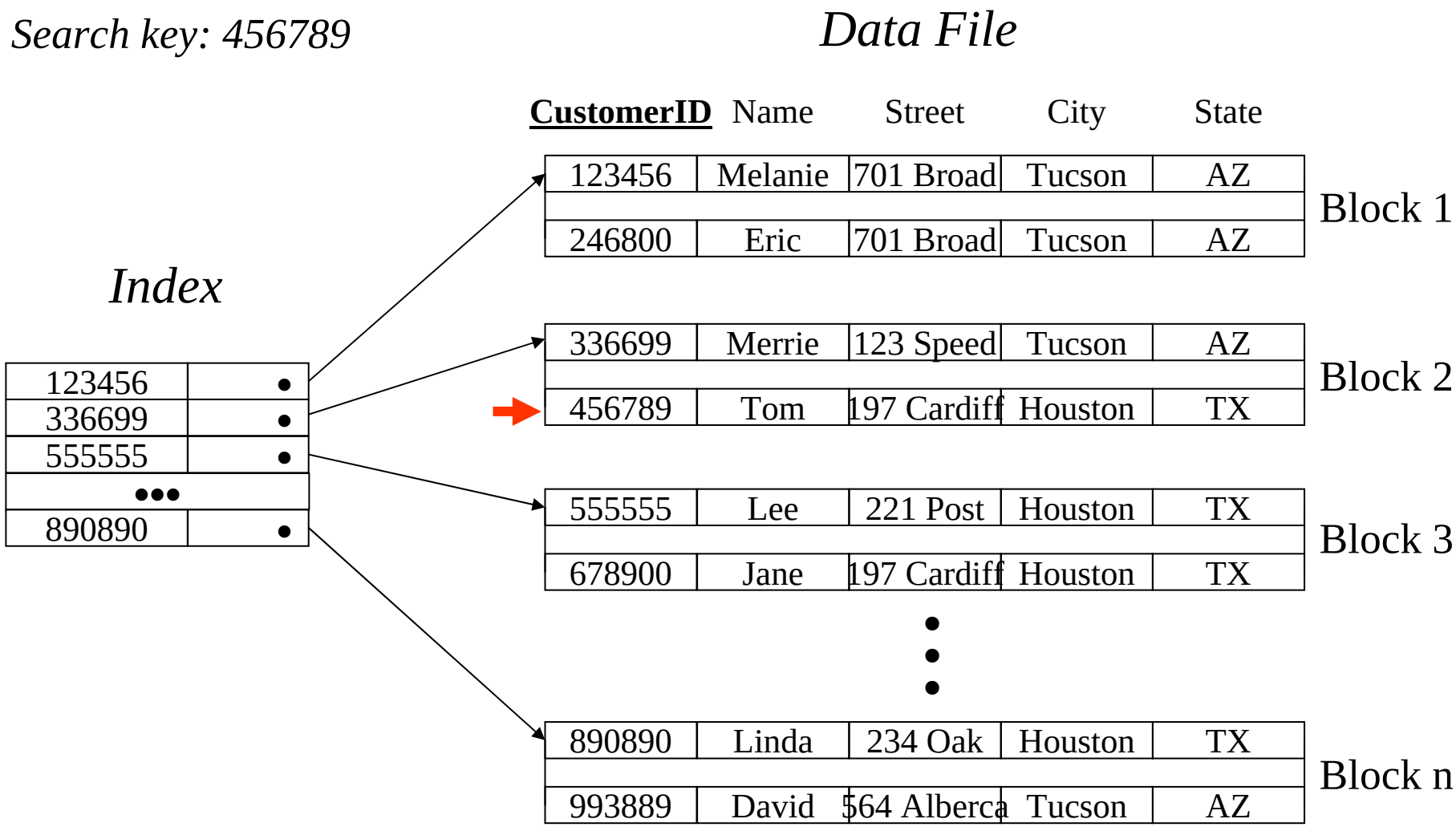
Physical Database Design

- File Organization
- Mapping Records to Files
- Buffer Management
- Indexes (Trees and Hashing)
  - Single-level versus multi-level
  - B^+-Trees
  - Static Hashing
- Tuning Physical Design

Types of Single-Level Indexes

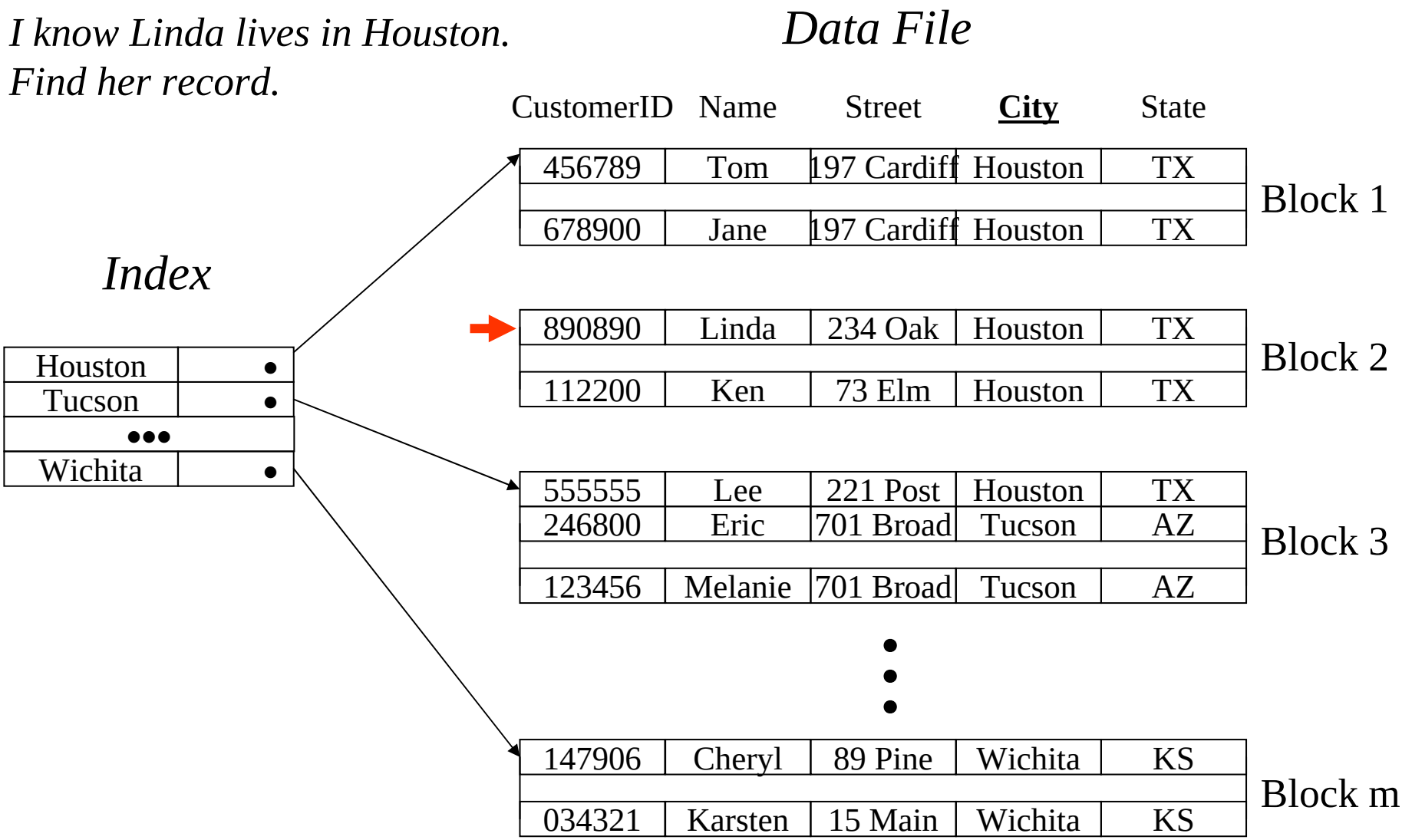
- *Primary Index*
  - Defined on a data file ordered on the *primary key*.
  - Includes one index entry for *each block* in the data file.
    - The index entry has the key field value for the *first record* in the block, which is called the *block anchor*.
- *Clustering Index*
  - Defined on a data file ordered on a *non-key field*.
  - Includes one index entry for each *distinct value* of the field.
    - The index entry points to the *first data block* that contains records with that field value.
- *Secondary Index*
  - Defined on a data file not ordered on the index key.
  - Includes one entry for *each record* in the data file: termed a *dense index*.

Example: Primary Index



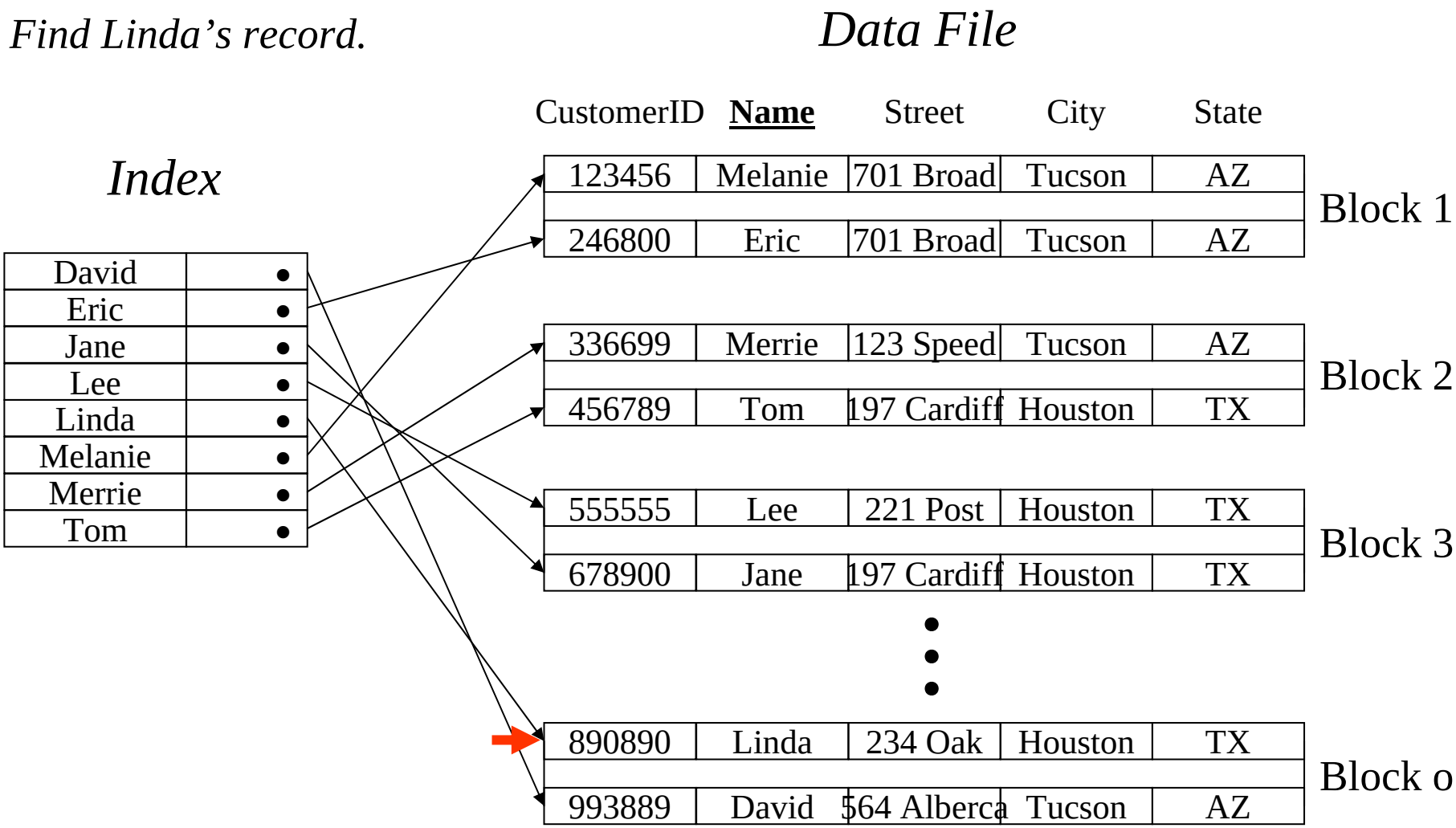
Example: Clustering Index

I know Linda lives in Houston.
Find her record.



Example: Secondary Index

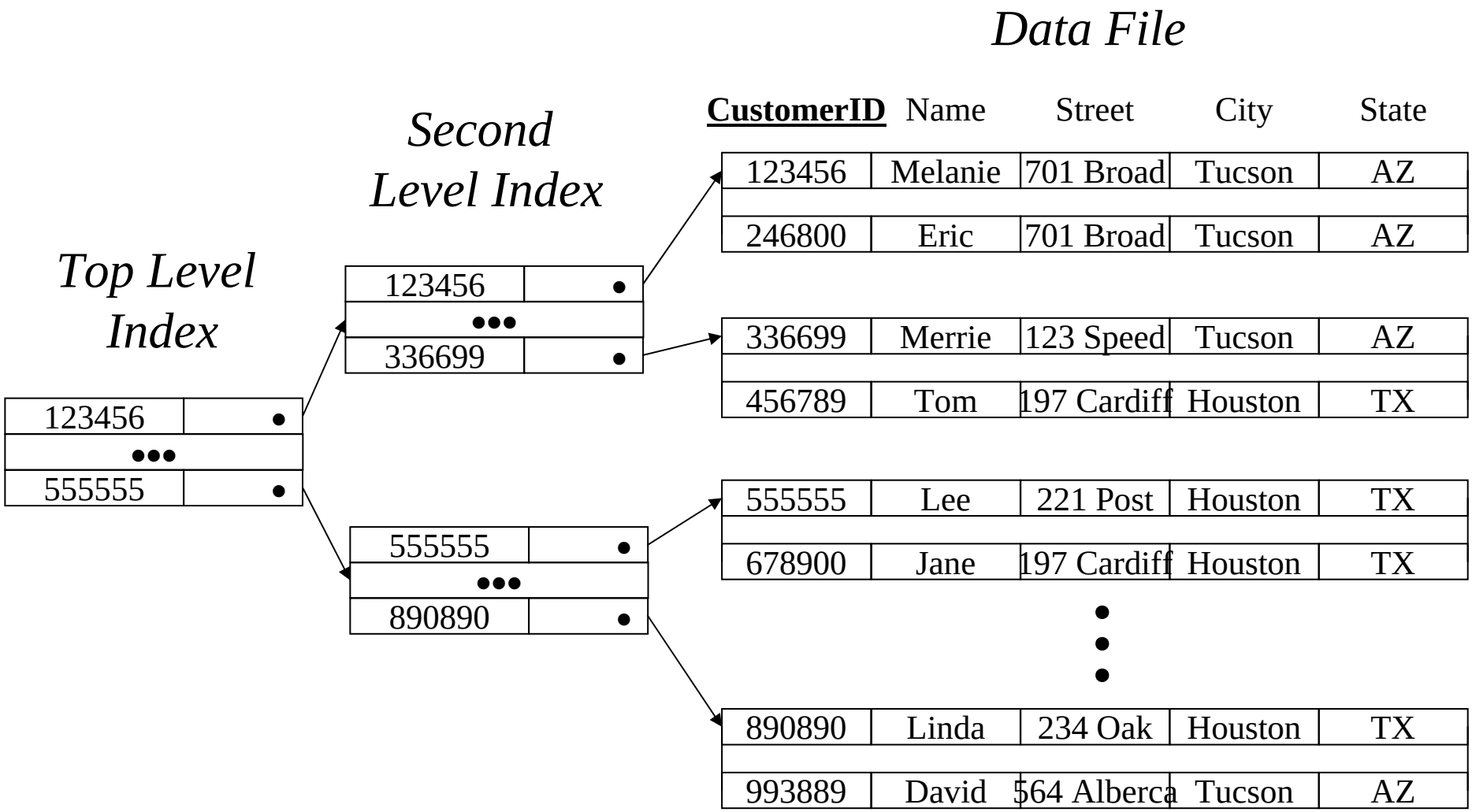
Find Linda's record.



Multi-Level Indexes

- Because a single-level index is an ordered file, we can create a primary index to the index itself.
  - The original index file is called the *second-level index*.
  - The index to the index is called the *(top-) first-level index*.
- We can repeat the process, until all entries of the top level index fit in one disk block, which is pinned in main memory.

Multi-level Index Example



Limitations of Multi-Level Indexes

- Suppose we have a number of levels
- Insertion and deletion of new index entries is a severe problem, because every level of the index is an ordered file.
- Most multi-level indexes use B-tree or B^+-tree data structures, which leave space in each disk block to allow for new index entries.

- B^+-trees can be used as an efficient tool for maintaining a hierarchy of indices. They are, in effect, balanced search trees.

B^+-Tree

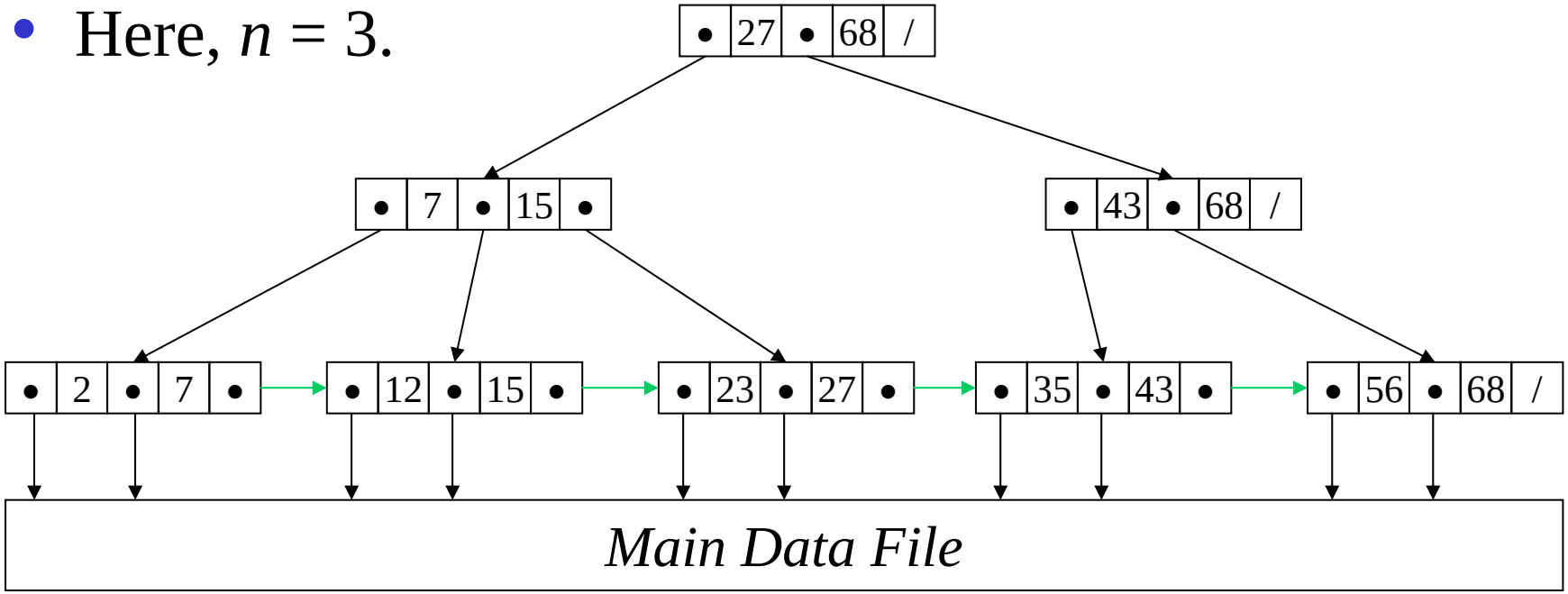
- A *B^+-tree* is a balanced tree in which every path from the root of the tree to a leaf of the tree is of the same length.
- Typical node: having *n-1* keys and *n* pointers.

  - 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).
  - *n* is often called *fan-out*.
- The search-keys in a node are ordered

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

Example of a B$^+$-tree

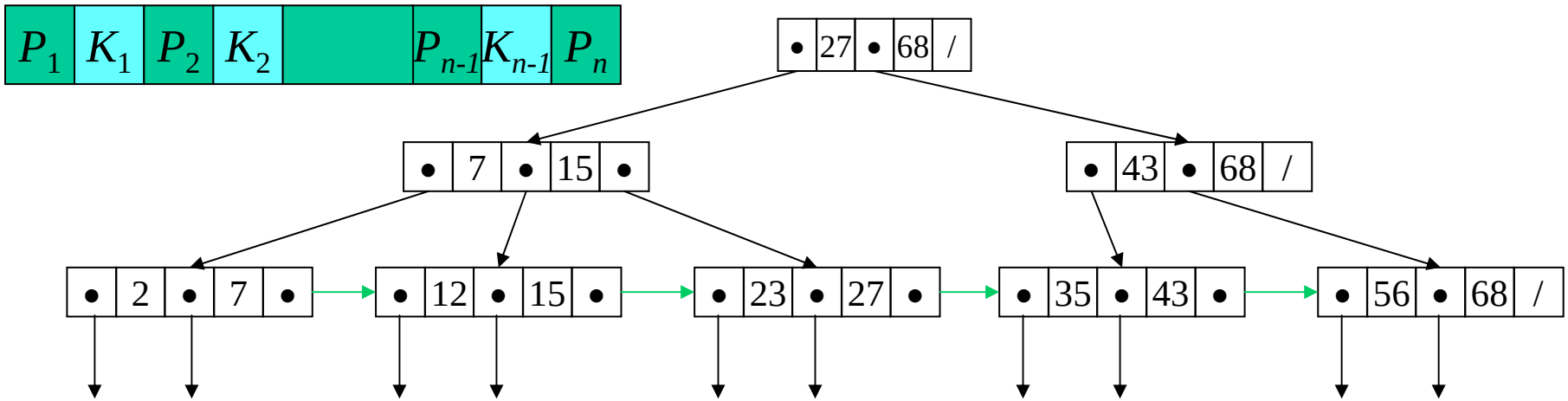
- Here, $n = 3$.



- A B$^+$-tree of size n 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 has between $\lceil (n-1)/2 \rceil$ and $n-1$ children.

B$^+$-tree Properties

- All the keys in the subtree to which P_1 points are $\leq K_1$
- For $1 < i < n$ all the keys in the subtree to which P_i points have values $K_{i-1} < X \leq K_i$
- All the keys in the subtree to which P_n points $> K_n$
- In the leaf nodes P_i points to the record with key value K_i
- The last pointer in each leaf is not needed for pointing into the main file. It is used instead to link all the leaf nodes to form a sequential index. (sometimes double link)

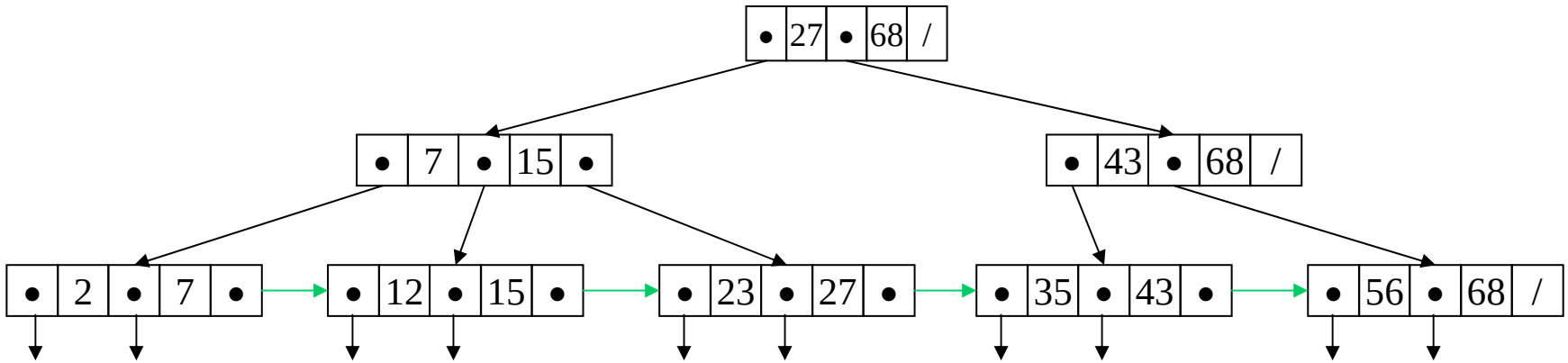


B^+-tree in Practice

- Each B^+-tree node has the size of one I/O block of data.
- The B^+-tree contains a rather small number of levels, usually logarithmic in the size of the main file.
  - minimize tree height and thus searches can be conducted efficiently.
- First one or two levels of the tree are stored in main memory to speed up searching
- Most internal nodes have less than (n-1) searching keys most of the time
  - huge space wastage for main memory, but not a big problem for disk

More B$^+$-tree Properties

- Note that here is no assumption that in the B$^+$-tree the "logically" close blocks are "physically" close, as the parent/child connections are done by pointers.
- The non-leaf levels of the B$^+$-tree form a hierarchy of *sparse indices*.
- Insertions and deletions to the main file can be handled efficiently, as the index can be reconstructed in logarithmic time as well.

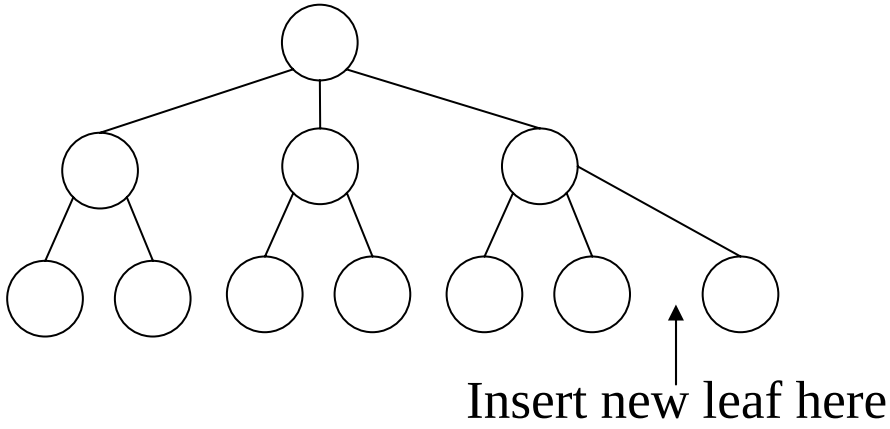


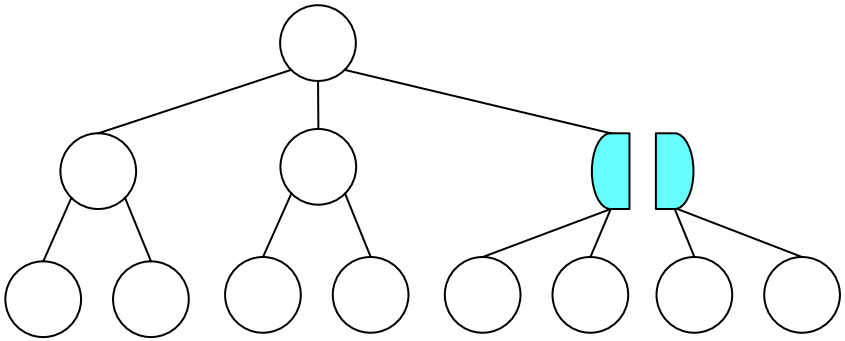
B^+-tree Insertions

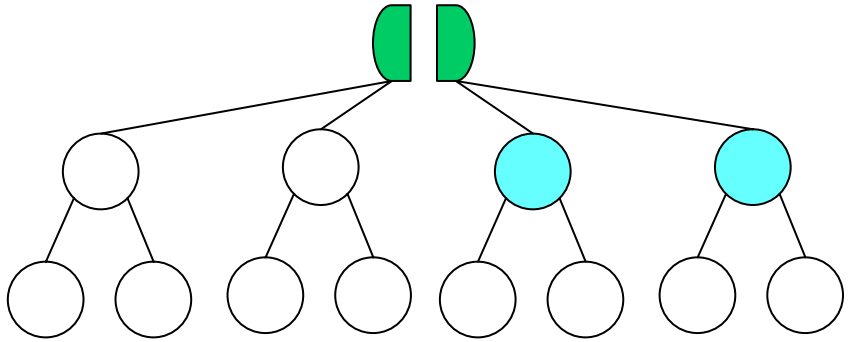
- Find the leaf node in which the search-key value would appear.
- If the search-key value is already there, add data to bucket.
- If the search-key value is not there:
  - if there is room in the node record, add the search-key value in appropriate order, and add data to bucket.
  - if there is no room in record split the record.
  - Splitting can go up until the root, which if necessary will also split and a new root will be created.

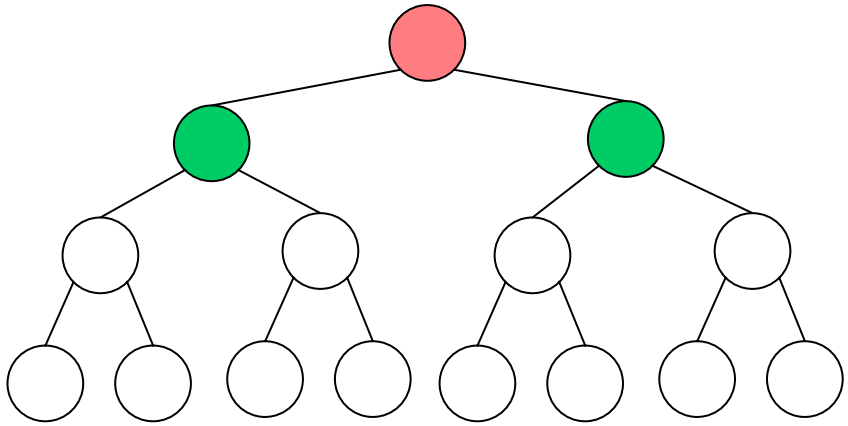
B$^+$-tree Insertions, cont.

- Example: (n = 3)



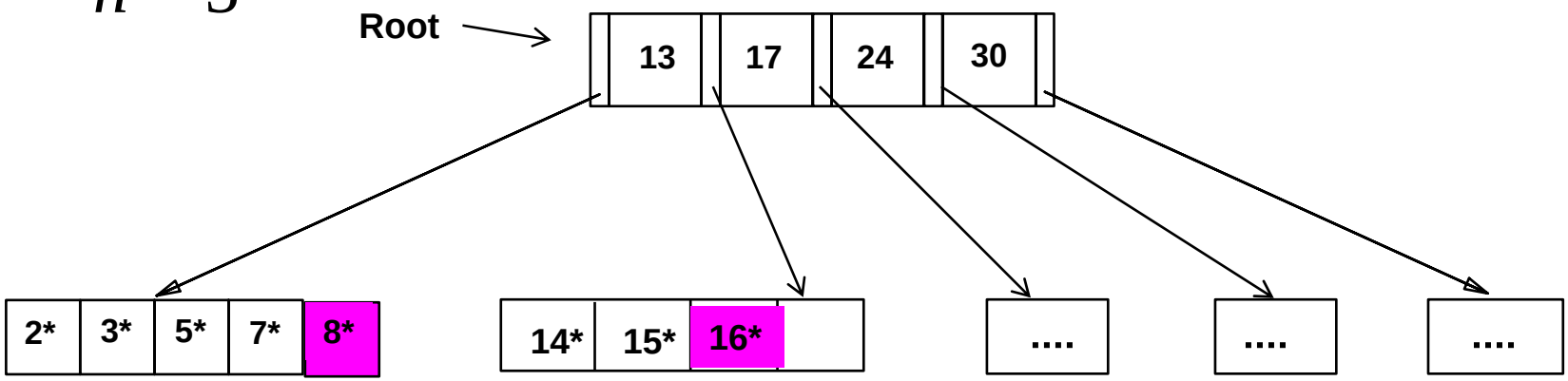






Inserting 16*, 8* into Example B+ tree

- $n = 5$

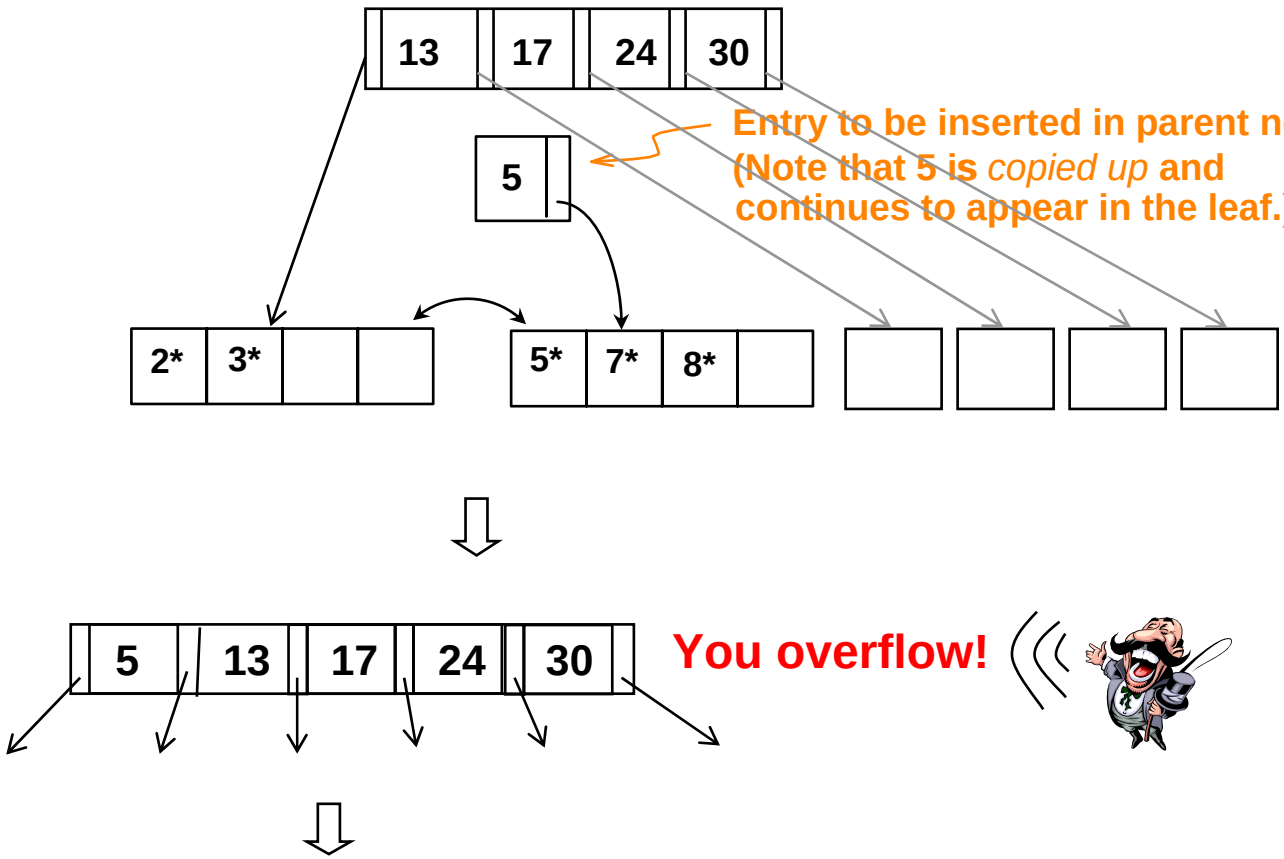


You overflow

One new child (leaf node) generated; must add one more pointer to its parent, thus one more key value as well.

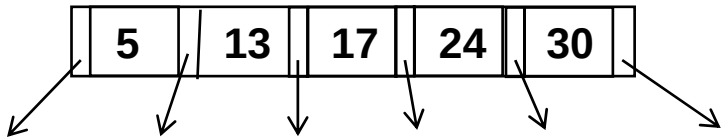
Inserting 8*, cont.

- Copy up the middle value (leaf split)



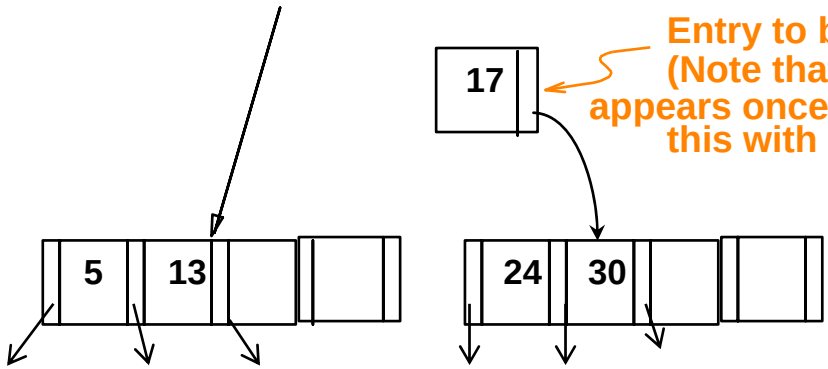
Inserting 8*, cont.

- Understand difference between copy-up and push-up
- Observe how minimum occupancy is guaranteed in both leaf and non-leaf splits.

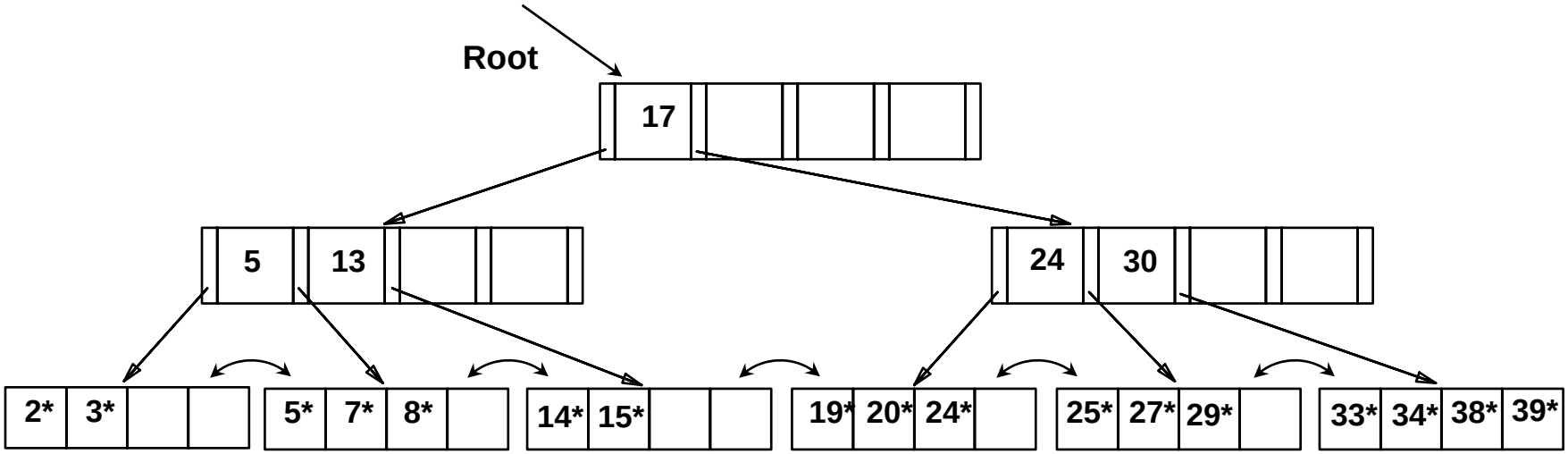


We split this non-leaf node, redistribute entries evenly, and push up middle key.

⇓



Example B+ Tree After Inserting 8*

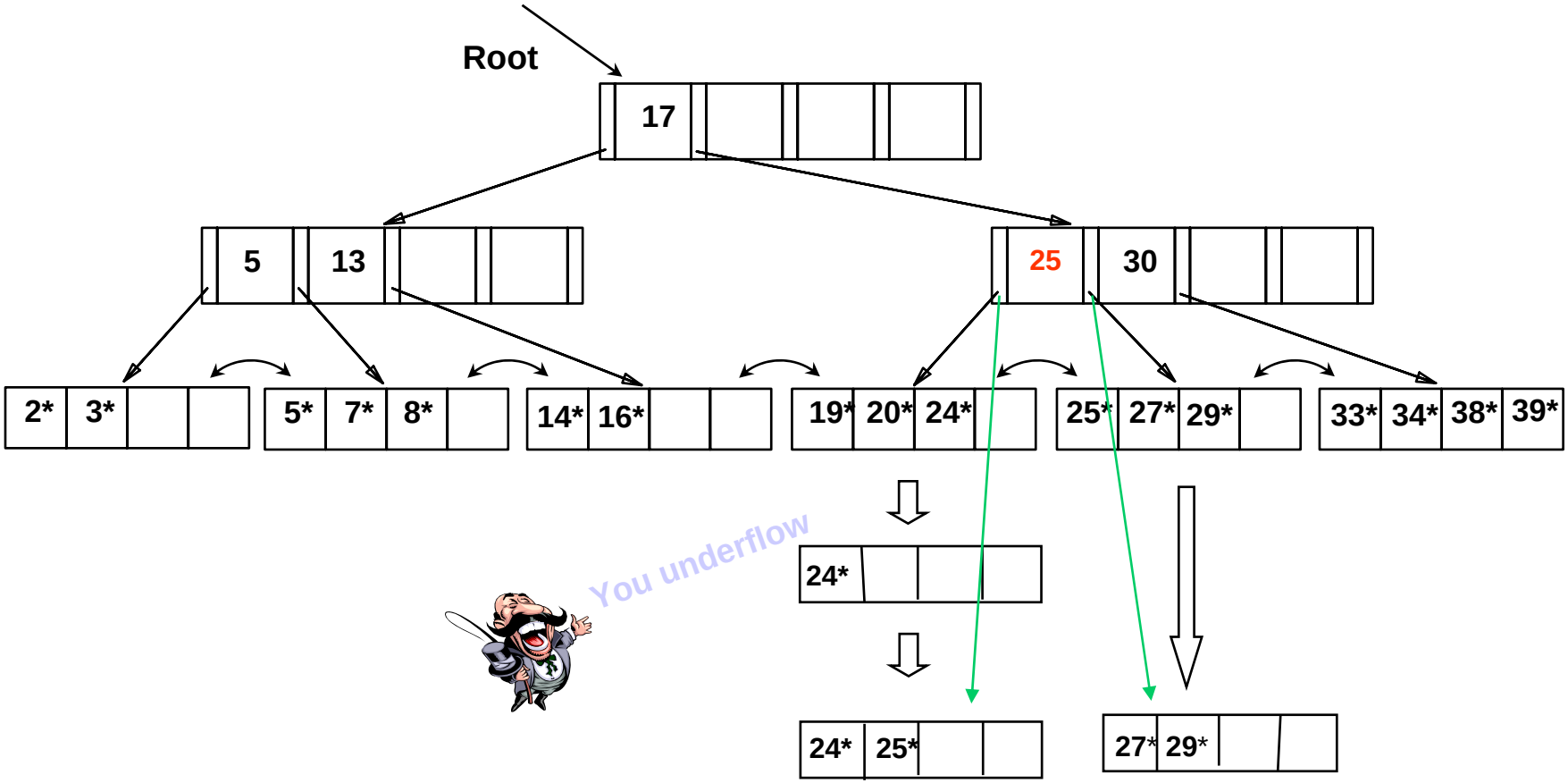


Notice that root was split, leading to increase in height.

B^+-tree Deletions

- Find the record to be deleted, and remove it from the bucket.
- If the bucket becomes empty, remove the search-key from the leaf node.
- Adjust the tree, if necessary.
  - Merge leaf nodes if underflow.
  - Recurse up the tree if necessary, to ensure that all nodes are not underfull.
- For both insertions and deletions, the probability that reorganization will be necessary is low, ensuring logarithmic average case performance.

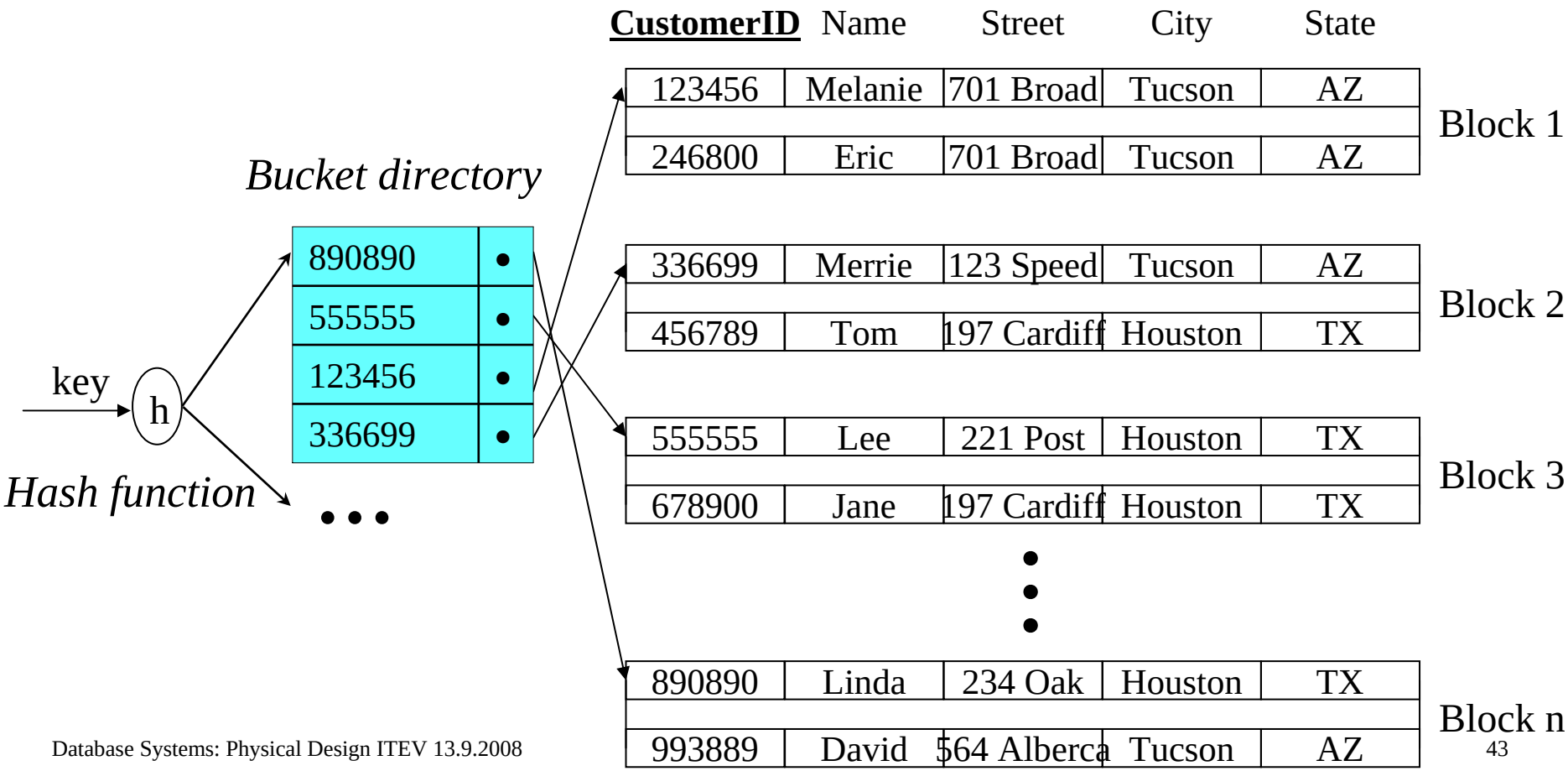
Delete 19* and 20*



- Notice how 25 is *copied up*.

Static Hashing Index

- In a static hash index, the keys are not stored in sequential order.
- Rather, the key is stored in a bucket computed by applying a hash function to the key.



Static Hashing Index, cont.

- In *direct hashing*, the bucket directory points to blocks of the data file.
- In *indirect hashing*, the bucket directory points to blocks of index entries, which themselves point to the blocks of the data file.
- *Bucket overflow* (when too many different key values hash to the same bucket) is handled with overflow chains.
- Lookup: 2 accesses, one to the bucket and one to the data file.
- Good performance depends on a good hash function.

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 underflow).
  - 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.
  - Dynamic hashing (not covered here)

Physical Database Design

- File Organization
- Mapping Records to Files
- Buffer Management
- Indexes (Trees and Hashing)
- Tuning Physical Design
  - Design decisions concerning indexes

Tuning Physical Design

- Design decisions
  - Ordered indexes vs. Hashing
  - Sparse vs. dense index
  - Clustering vs. Non-clustering indexes
  - Joins and indexes
- When is the index not used?

Ordered Indexes vs. Hashing

- In an **ordered index,** index entries are stored sorted on the search key value. E.g., author catalog in library.
  - Primary index and secondary index are ordered indexes
  - Hashing is not
- 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

Sparse vs. Dense Index

- #pointers in dense index =
  #pointers in sparse index · #records per block
- For large records, dense and sparse indexes are about the same size
- Sparse index usually has one level less
  - Recall in B^+-tree all non-leaf levels form a hierarchy of sparse indexes
- Sparse index is better for all updates and most queries
- IF query retrieves index attribute (e.g. count queries) only
  THEN use dense index
  ELSE use sparse index

Clustering vs. Non-clustering

- Good news for clustering
  - Clustering index may be sparse (small)
  - Good for range queries (e.g., 20000 < *E.sal* < 40000) and prefix queries (e.g., *E.Name*='Sm%')
- Bad news for clustering
  - Inserts tend to be placed in the middle of the table. This causes overflows, destroying the benefits of clustering
  - Similarly for updates
- Use low block utilisation when clustering
- Good news for non-clustering
  - Dense, so some queries can be answered without access to table (Example on the next slide)
  - Good for point queries and multi-point queries

Joins & Indexes

- An index on `R.B` or `S.C` will help:

```
SELECT      R.A, R.D
FROM        R, S
WHERE       R.B = S.C
```

- An index on `R.B` or `S.C` will not help (selection is done first).

```
SELECT      R.A, R.D
FROM        R, S
WHERE       R.B = S.C AND S.D = 5 AND R.E = 6
```

- A dense index on S.C is better than sparse clustering index (semijoin; no access to S needed).

```
SELECT      R.A, R.D
FROM        R, S
WHERE       R.B = S.C
```

Joins & Indexes, cont.

- A non-clustering index on `S.C` will help (but every match gives a logical block access).

```
SELECT      R.A, R.D, S.E
FROM        R, S
WHERE       R.B = S.C
```

- A clustering index on `S.C` will also help (but every match gives a logical block access).

```
SELECT      R.A, R.D, S.E
FROM        R, S
WHERE       R.B = S.C
```

- B^+-tree can be used for <, <=, =, >=, > joins.
- Hash index can only be used for equijoins.

Composite Index

- Phone book: Composite index on (lastname, firstname)
- Advantages:
  - If dense, then query may be answered from index attributes only
  - Composite index more selective, i.e. returns fewer tuples
  - Sometimes good for multi-dimensional queries, e.g.

```
SELECT phonenum FROM phonebook
WHERE lastname='Smith' AND'Jason'<firstname<'John'
```

- Disadvantages:
  - They tend to have large keys (key compression may help)
  - An update to any of the key attributes causes an index update
  - Sometimes bad for multi-dimensional queries, e.g.

```
SELECT phonenum FROM phonebook
WHERE firstname='Jim'AND'Jamison'<lastname<'Johnson'
```

When Isn't the Index Used?

- Catalogue out of date
  - optimizer may think table is small
- “Good” and “bad” query examples:
  - `SELECT * FROM EMP WHERE salary/12 > 4000`
  - `SELECT * FROM EMP WHERE salary > 48000`
  - `SELECT * FROM EMP WHERE SUBSTR(name, 1, 1) = 'G’`
  - `SELECT * FROM EMP WHERE name = 'G%’`
  - `SELECT * FROM EMP WHERE salary IS NULL`
- Nested sub-query
- Selection by negation
- Queries with **`OR`**

Summary

- Variable length records complicate storage management.
- Requiring the main file to be sorted slows down insertions and deletions.
- B^+-trees support logarithmic insertion, deletion, and lookup, without excessive space costs.
- Query processing should take file structure and presence of indexes into account.
- Tuning the physical design requires knowing query and modification mix.