Storage, I/O complexity, & external sorting

Lecture 2
2ID35, Spring 2015

George Fletcher

Faculteit Wiskunde & Informatica
Technische Universiteit Eindhoven

24 April 2015

Agenda

Last time: course overview and background

Agenda

Last time: course overview and background

Today

- Admin: results of 5-minute paper
- Admin: project update
- Finish background review: FO queries

Agenda

Last time: course overview and background

Today

- Admin: results of 5-minute paper
- Admin: project update
- Finish background review: FO queries
- Topic 1: Storage on disk & I/O model of computing
- Topic 2: External sorting
- Topic 3: Indexing, part 1 – ordered indexes

Admin: results of 5-minute papers

Interests range from foundations to applications

- "Realising an efficient database implementation."
- "How research in database technology is developing."
- "I would like to see some real-world examples, not only the theoretical viewpoint."

Admin: results of 5-minute papers

Interests range from foundations to applications

- "Realising an efficient database implementation."
- "How research in database technology is developing."
- "I would like to see some real-world examples, not only the theoretical viewpoint."

many papers: NoSQL technologies (JSON stores, Graph DBs)

- many non-relational technologies are explored in the team projects
- let's also shift lecture of 5 June to further highlight these topics

Admin: results of 5-minute papers, cont.

Also, several papers: obtain knowledge to be able to make effective data management decisions in practice

- "How to make use of in-depth knowledge about the inner workings of current existing popular DB systems, while working in industry."
- "How are the technologies discussed used in a business setting?"
- "To know more about the different technologies used in DBs so I can make informed decisions on which existing DB software to use in my applications."

Admin: results of 5-minute papers, cont.

Also, several papers: obtain knowledge to be able to make effective data management decisions in practice

- "How to make use of in-depth knowledge about the inner workings of current existing popular DB systems, while working in industry."
- "How are the technologies discussed used in a business setting?"
- "To know more about the different technologies used in DBs so I can make informed decisions on which existing DB software to use in my applications."

core goal of 2ID35: developing the solid technical foundations for making such decisions, for both state-of-the-art and emerging technologies

Funny Dilbert comic strip removed. See
http://search.dilbert.com/search?w=mauve+database

Admin

Project part 1 update:

- Please finish your submissions ASAP, if you haven't already
  - I will give contact via email
  - ~5 members per team
  - each team focuses on one paper
- Next step: carefully read and understand team paper. Full details to follow on course website.

FO review

author(<u>authorID</u>, name, birthdate, language)
book(<u>bookID</u>, title, authorID, publisher, language, year)
store(<u>storeID</u>, address, phone)
sells(<u>storeID</u>, <u>bookID</u>)

- List all (authorID, language) pairs where the given author has not published a book in the given language.

Agenda

- Topic 1: Storage on disk & I/O model of computing
- Topic 2: External sorting
- Topic 3: Indexing, part 1 – ordered indexes

Topic 1: Storage

Storage

Assumption: database may grow too large to maintain in volatile memory and/or we'd like to park it on stable storage

Storage

Assumption: database may grow too large to maintain in volatile memory and/or we'd like to park it on stable storage

- the memory hierarchy
- mapping relations to disk
- the structure of tuples/relations on disk
- the buffer manager
- I/O computing

Storage: the memory hierarchy

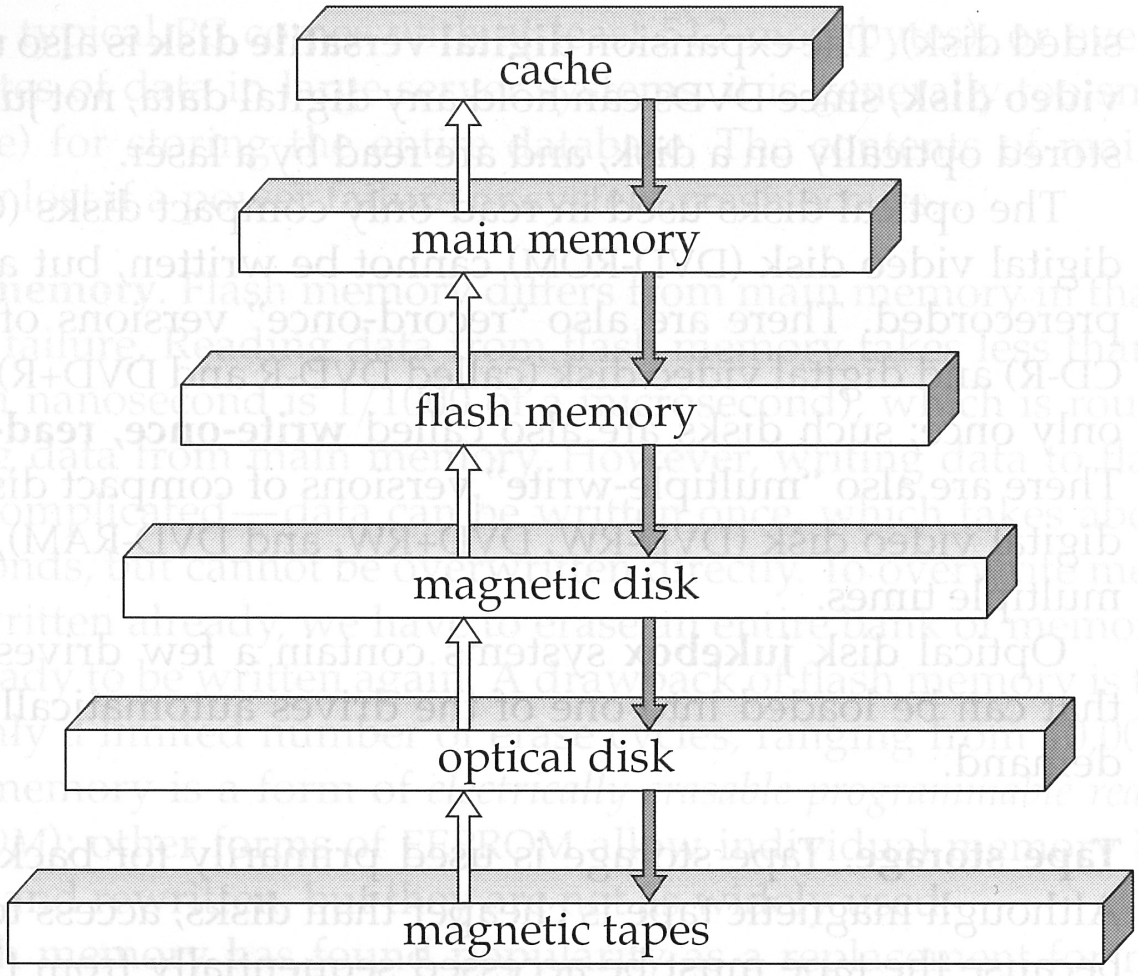
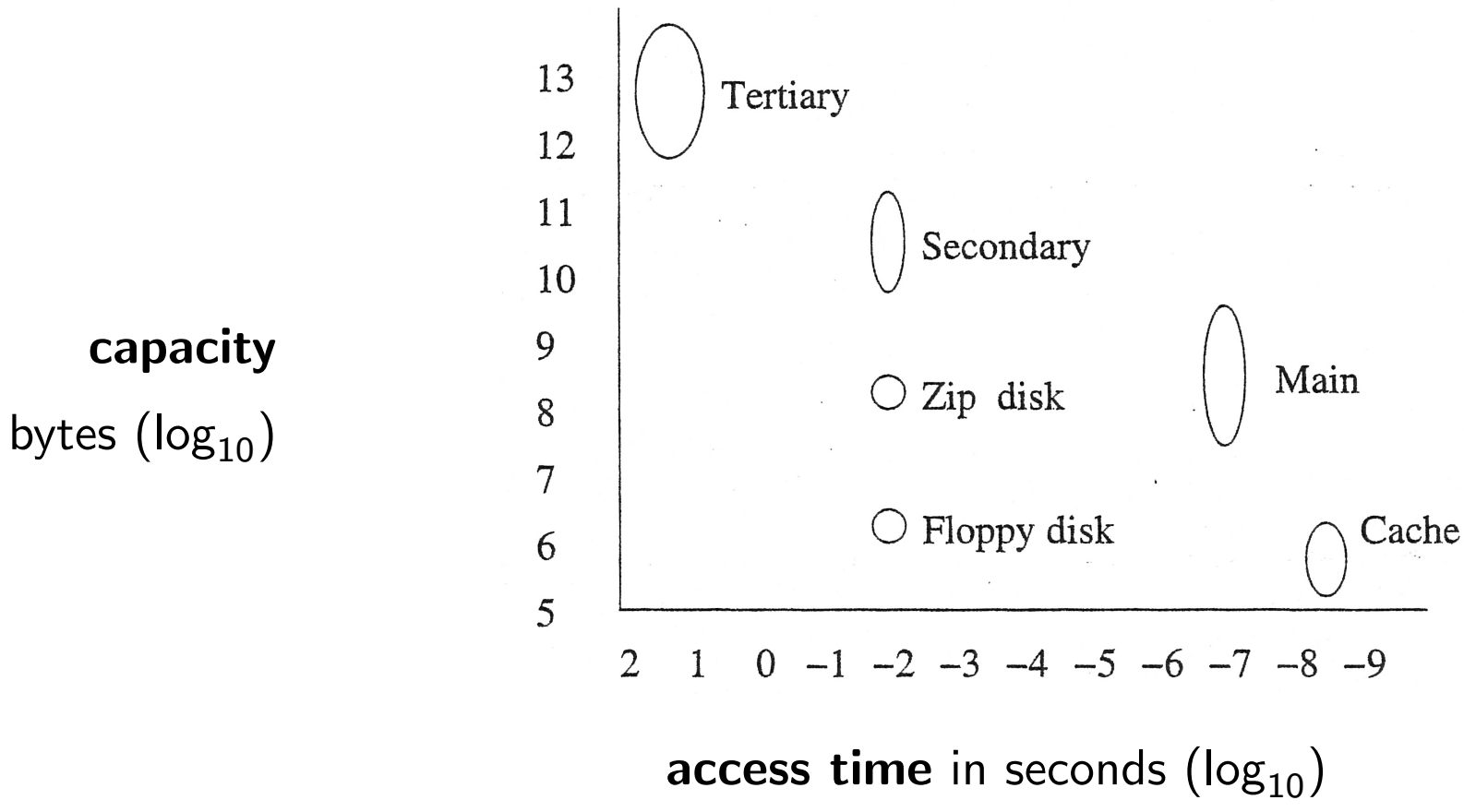
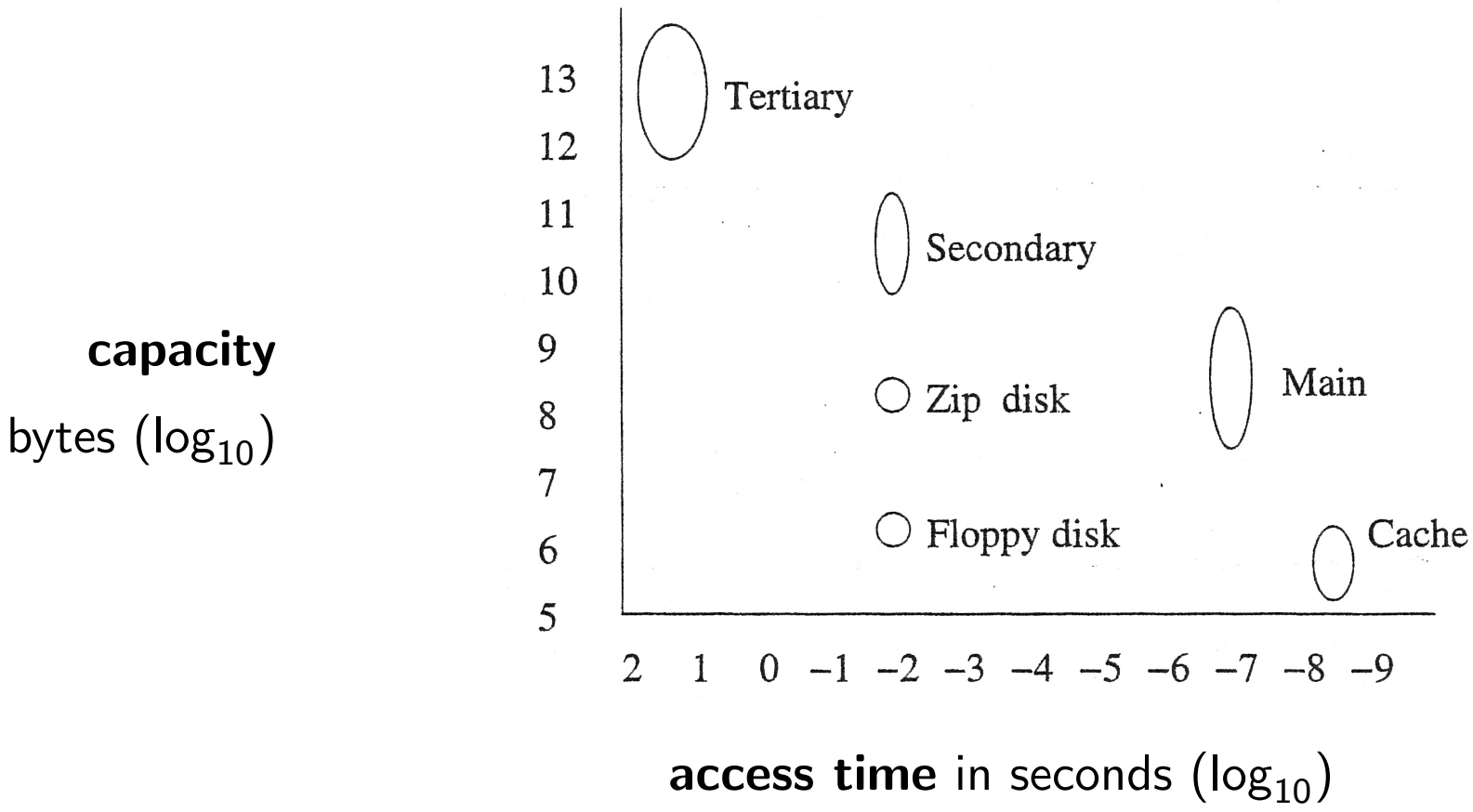


Figure 11.1 Storage device hierarchy.

Storage: the memory hierarchy



Storage: the memory hierarchy



$\Rightarrow$ main memory is $\approx 10^5$ times faster and $\frac{1}{100}$ the size of secondary memory

- SSDs narrow the speed gap by $\approx 10^2$

Storage: the memory hierarchy

As a comparison, if a book in your hand represents primary memory and a book in the library is like secondary memory, and it takes 20 seconds to locate an item in the book at hand....

Storage: the memory hierarchy

As a comparison, if a book in your hand represents primary memory and a book in the library is like secondary memory, and it takes 20 seconds to locate an item in the book at hand....

... it would take over 23 days to locate an item in a book in the library ...

Storage: the memory hierarchy

As a comparison, if a book in your hand represents primary memory and a book in the library is like secondary memory, and it takes 20 seconds to locate an item in the book at hand....

... it would take over 23 days to locate an item in a book in the library ...

... not a very good library ...

Storage: the memory hierarchy

As another comparison, if retrieving something from main memory is like retrieving an object from Amsterdam (126 km), then retrieving something from disk is like ...

Storage: the memory hierarchy

As another comparison, if retrieving something from main memory is like retrieving an object from Amsterdam (126 km), then retrieving something from disk is like ...

traveling to New York city?

Storage: the memory hierarchy

As another comparison, if retrieving something from main memory is like retrieving an object from Amsterdam (126 km), then retrieving something from disk is like ...

traveling to New York city?

no, that is only 6,000 km away ....

Storage: the memory hierarchy

As another comparison, if retrieving something from main memory is like retrieving an object from Amsterdam (126 km), then retrieving something from disk is like ...

traveling to New York city?

no, that is only 6,000 km away ....

traveling to the moon?

Storage: the memory hierarchy

As another comparison, if retrieving something from main memory is like retrieving an object from Amsterdam (126 km), then retrieving something from disk is like ...

traveling to New York city?

no, that is only 6,000 km away ....

traveling to the moon?

no, that is only 385,000 km away ....

Storage: the memory hierarchy

As another comparison, if retrieving something from main memory is like retrieving an object from Amsterdam (126 km), then retrieving something from disk is like ...

traveling to New York city?

no, that is only 6,000 km away ....

traveling to the moon?

no, that is only 385,000 km away ....

it would be like traveling to New York 2100 times (or, around the earth 315 times, or only 33 times to the moon)!

Storage: the memory hierarchy

moral: we don't want to travel to the moon, unless we absolutely must

Storage: secondary memory

- tuple $\longrightarrow$ record
- relation (collection of records) $\longrightarrow$ file

Storage: secondary memory

- tuple $\longrightarrow$ record
- relation (collection of records) $\longrightarrow$ file
- file $\longrightarrow$ collection of pages (disk blocks)
- block size typically in the range 4K to 56K

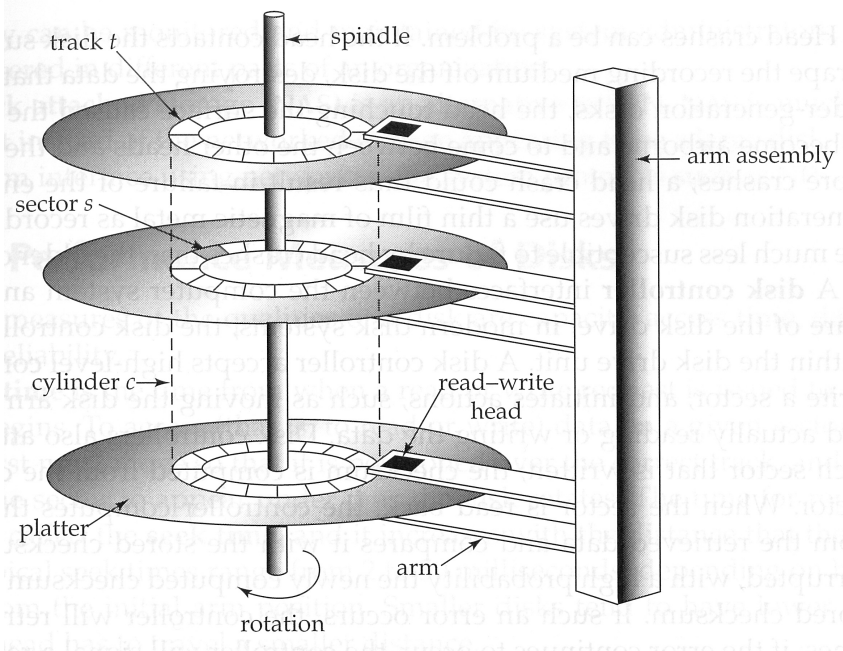
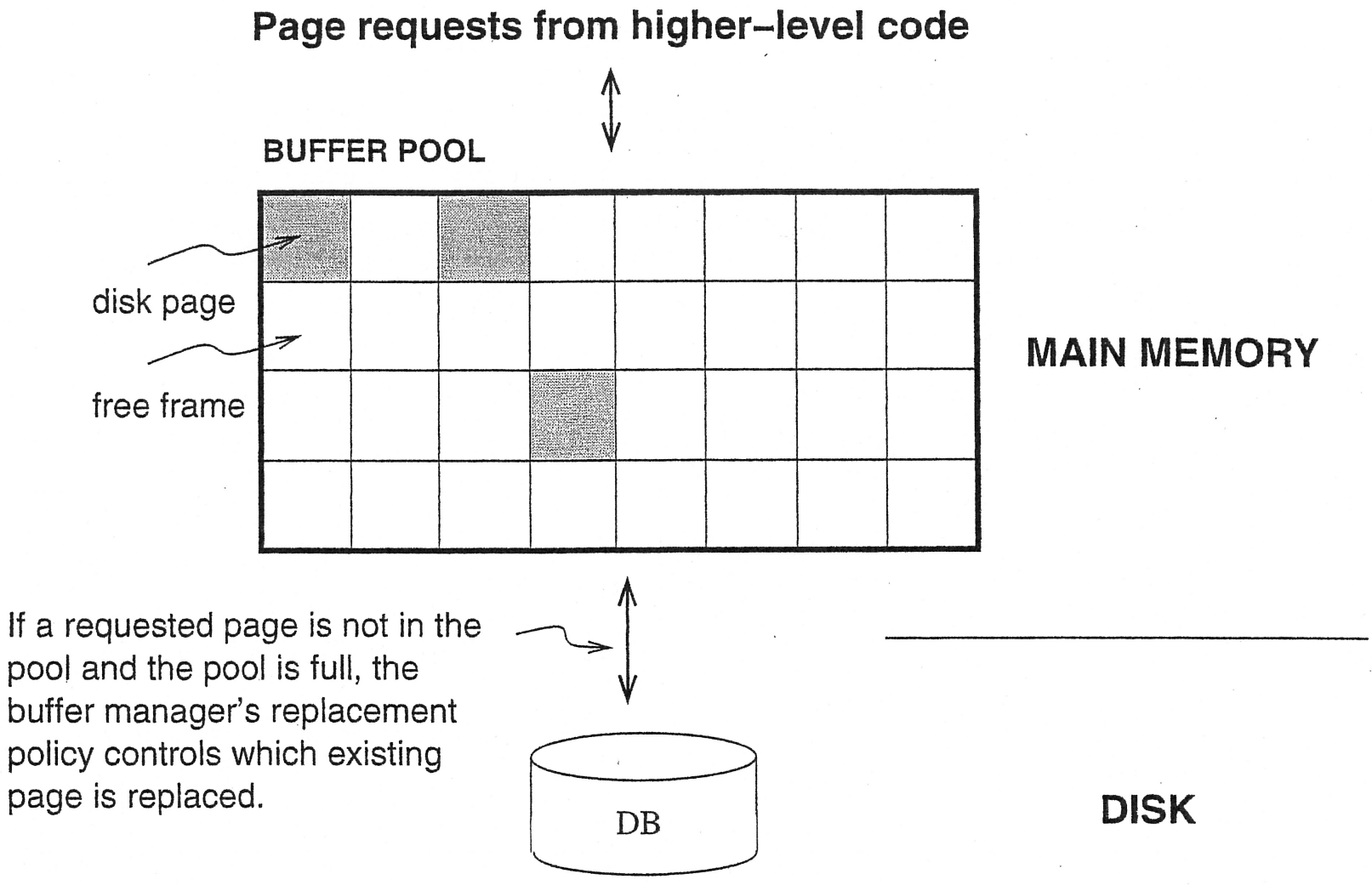


Figure 11.2 Moving head disk mechanism.

Storage: the buffer manager

We must bring data into lowest memory level (i.e., CPU registers) to perform work.

Storage: the buffer manager



The Buffer Pool

I/O model of computation

- disk block is unit of I/O
- model is defined by dominance of I/O cost over CPU cost
- parameters:
  - B = number of blocks in a file
  - D = average time to read a block
  - R = number of records per block
  - C = average time to process a record

I/O model of computation

- disk block is unit of I/O
- model is defined by dominance of I/O cost over CPU cost
- parameters:
  - B = number of blocks in a file
  - D = average time to read a block
  - R = number of records per block
  - C = average time to process a record

Typically $D = 15$ milliseconds $(10^{-3}s)$ and $C = 100$ nanoseconds $(10^{-9}s)$

$\Rightarrow$150,000 record ops per block access!!

I/O model of computation

- disk block is unit of I/O
- model is defined by dominance of I/O cost over CPU cost
- parameters:
  - B = number of blocks in a file
  - D = average time to read a block
  - R = number of records per block
  - C = average time to process a record

Suppose 16KB blocks and 160B records

$\Rightarrow R = 100$ records per block

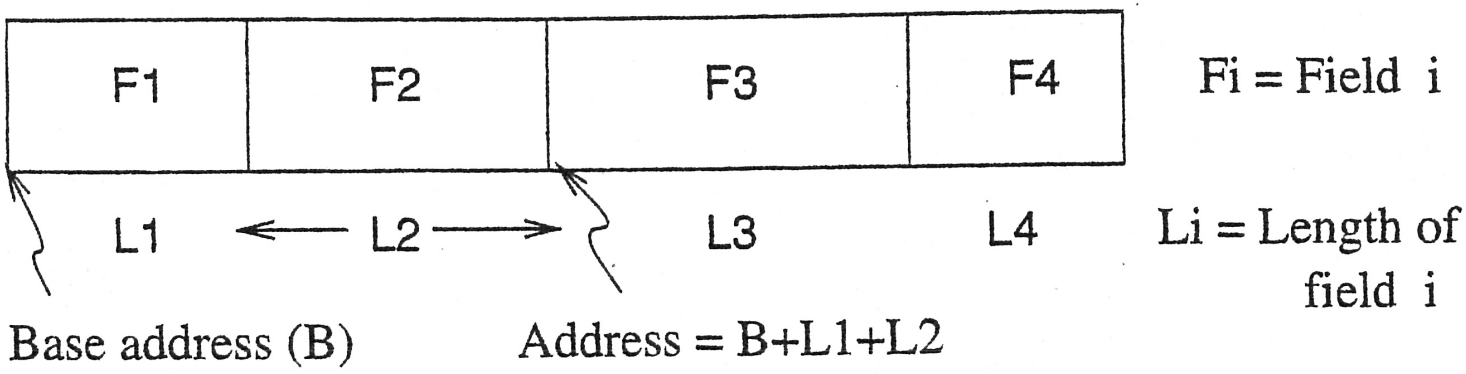
$\Rightarrow$ even a linear scan is negligible wrt block access!

Storage: record structure

- a file is organized logically as a sequence of records, and these records are mapped onto disk blocks
- two basic record types: fixed length vs. variable length

Storage: record structure

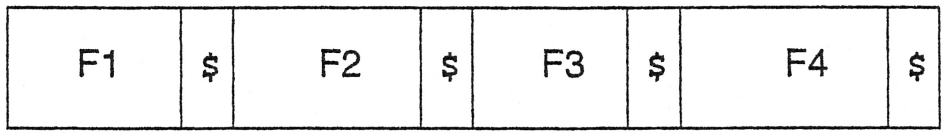
- a file is organized logically as a sequence of records, and these records are mapped onto disk blocks
- two basic record types: fixed length vs. variable length



Organization of Records with Fixed-Length Fields

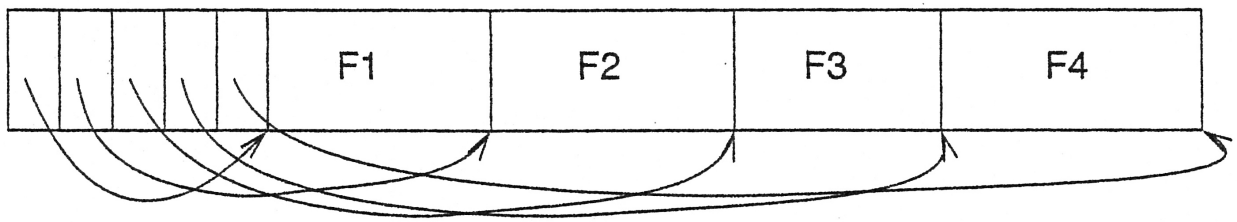
Storage: record structure

- a file is organized logically as a sequence of records, and these records are mapped onto disk blocks
- two basic record types: fixed length vs. variable length



Fi = Field i

Fields delimited by special symbol $



Array of field offsets

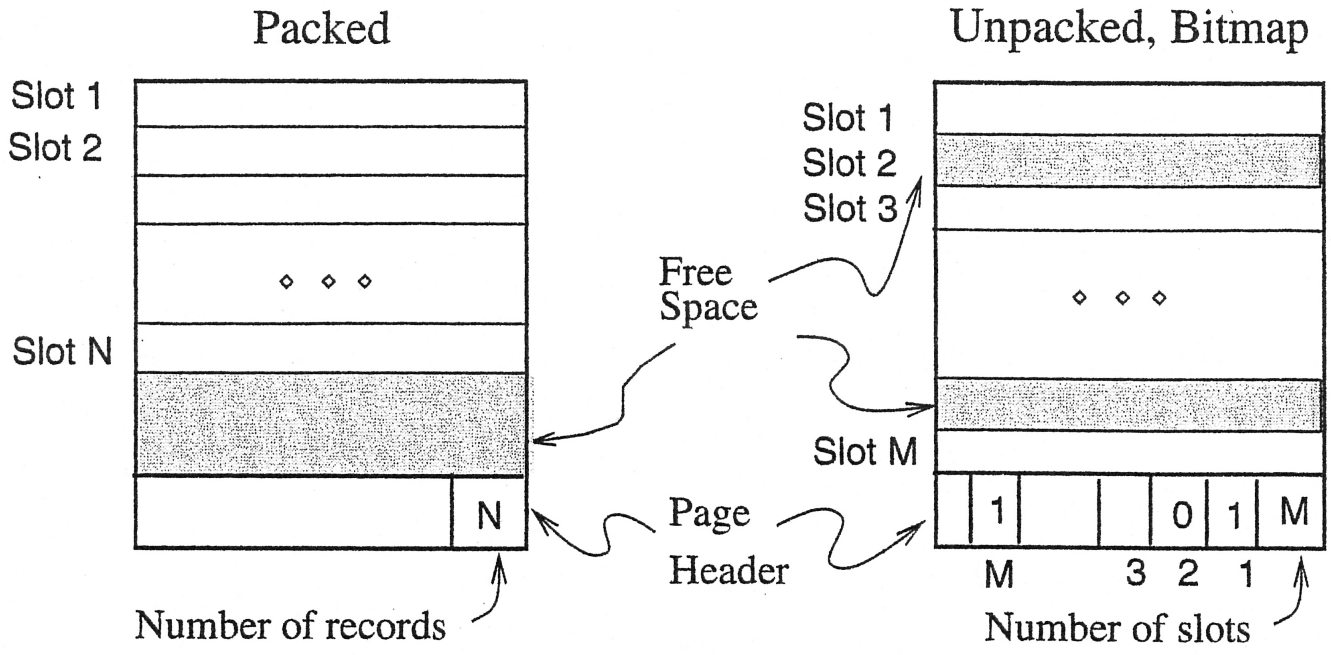
Alternative Record Organizations for Variable-Length Fields

Storage: page structure

Hence, we have two basic types of page structure

Storage: page structure

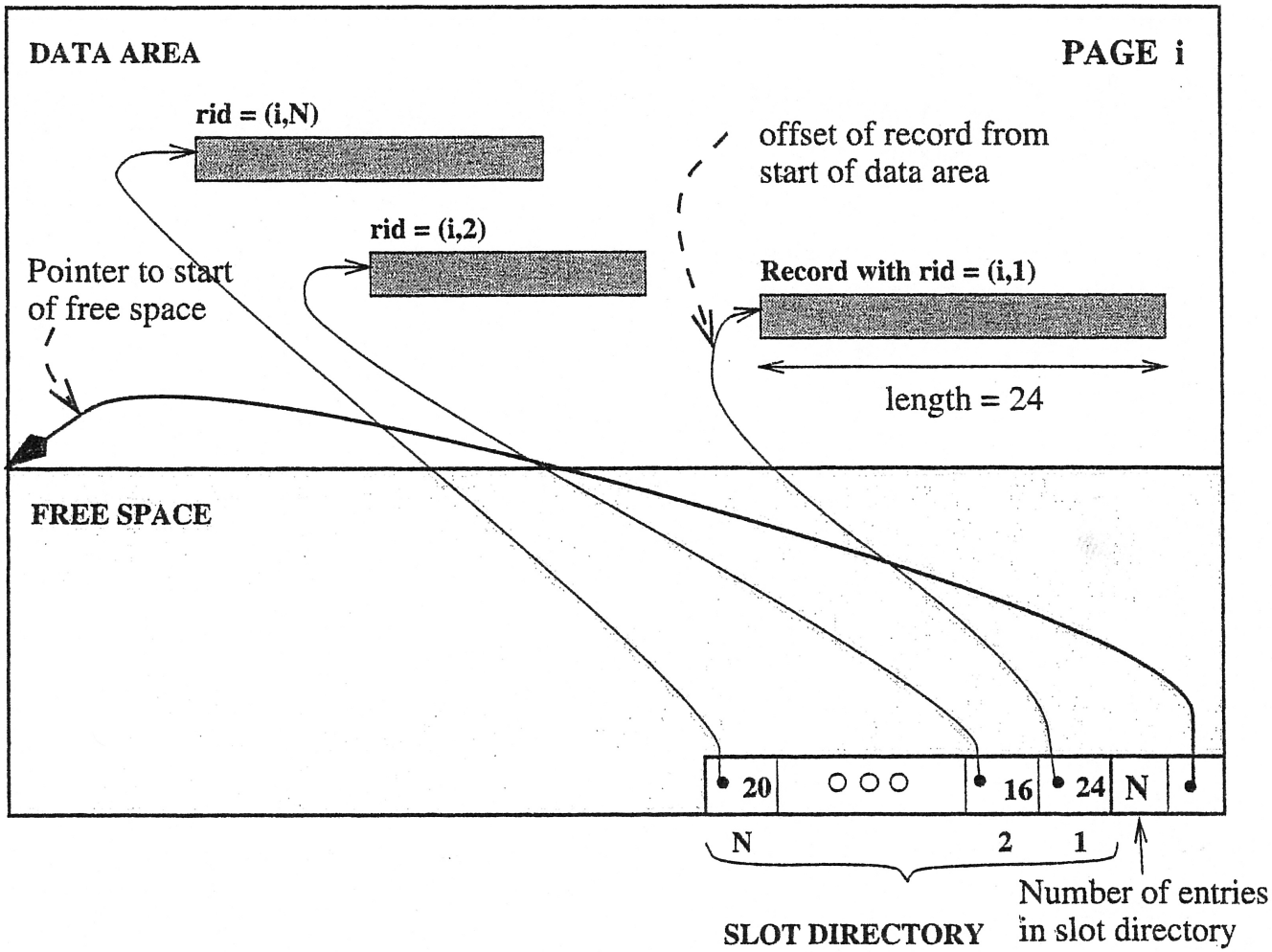
Hence, we have two basic types of page structure



Alternative Page Organizations for Fixed-Length Records

Storage: page structure

Hence, we have two basic types of page structure



Page Organization for Variable-Length Records

Storage: file structure

Within a file, records can be maintained

- *sequentially*, i.e., ordered on some field(s); as a

Storage: file structure

Within a file, records can be maintained

- *sequentially*, i.e., ordered on some field(s); as a
- *heap*, i.e., unordered; or, as a

Storage: file structure

Within a file, records can be maintained

- *sequentially*, i.e., ordered on some field(s); as a
- *heap*, i.e., unordered; or, as a
- *hash* file (which we'll study in some detail in a few lectures).

Storage: file structure

Within a file, records can be maintained

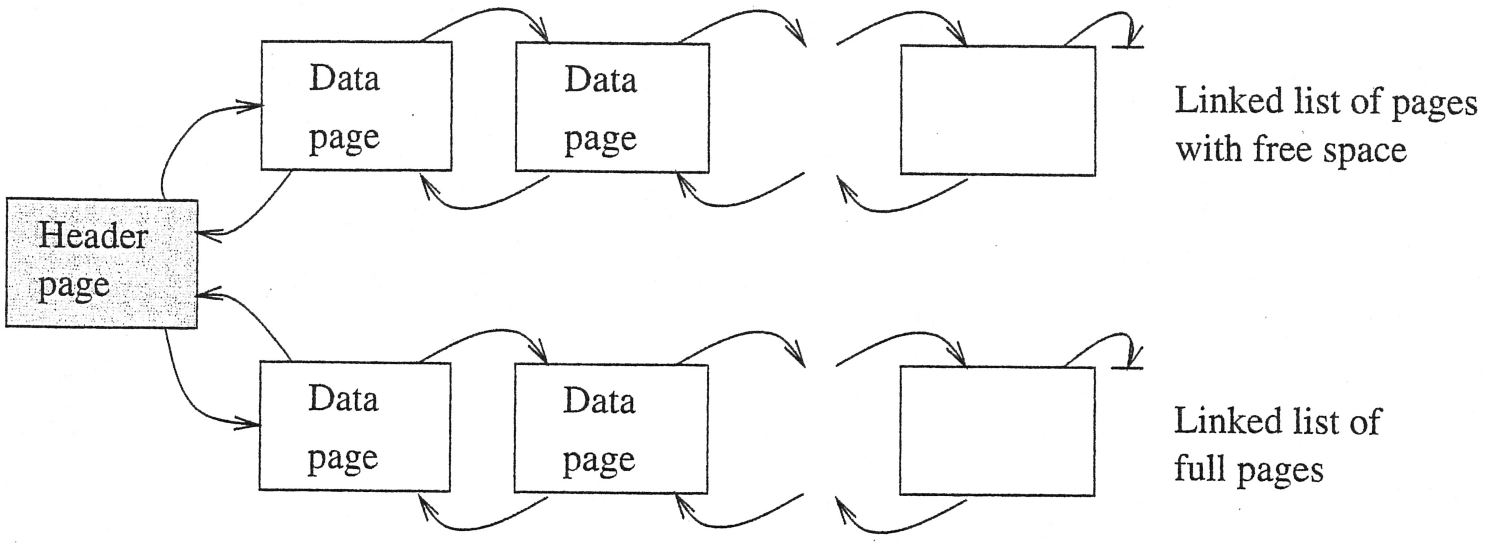
- *sequentially*, i.e., ordered on some field(s); as a
- *heap*, i.e., unordered; or, as a
- *hash* file (which we'll study in some detail in a few lectures).

Sorted data is very valuable, for a variety of reasons as we'll see, yet requires some work to maintain.

We'll investigate *indexing* as a means for maintaining a sorted file, in the next lecture.

Storage: heap file structure

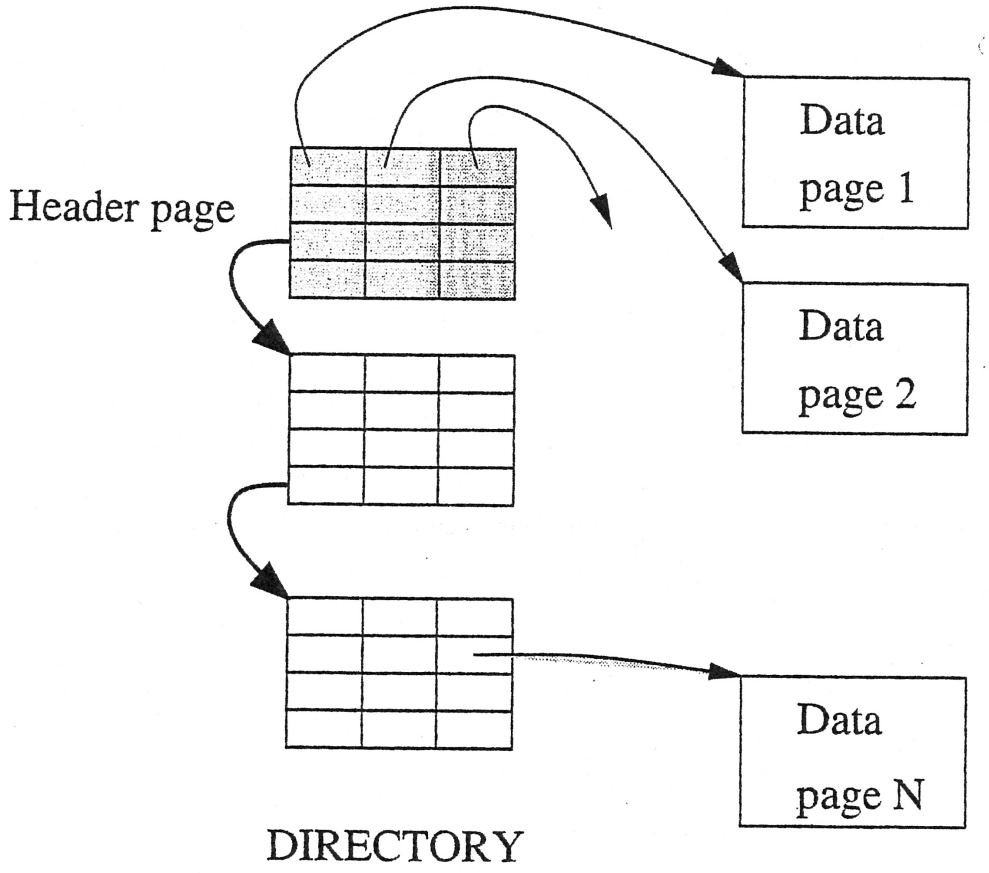
There a several natural ways to maintain a heap: as a linked list.



Heap File Organization with a Linked List

Storage: heap file structure

There a several natural ways to maintain a heap: using a directory.



Heap File Organization with a Directory

Searching in a heap

Recall our scenario of 16KB blocks and 160B records (i.e., 100 records/block).

Suppose our relation has 10,000,000 tuples, and therefore occupies 100,000 blocks.

Searching in a heap

Recall our scenario of 16KB blocks and 160B records (i.e., 100 records/block).

Suppose our relation has 10,000,000 tuples, and therefore occupies 100,000 blocks.

Now, suppose we are looking for records with a given value in a field (say, authors named Jan).

Since searching in a heap is exhaustive, the delay for this search is

$$\begin{aligned} B \times D &= 100,000 \times 0.015s \\ &= 1500s \\ &= 25 \text{ minutes} \end{aligned}$$

Searching in a heap

Clearly, this is unacceptable .... a sequential file, sorted on author names would be nice ...

Searching in a heap

Clearly, this is unacceptable .... a sequential file, sorted on author names would be nice ...

some other uses of sorted data:

- presentation of results,
- computing aggregates,
- duplicate elimination,
- algorithms for RA operations (e.g., join and projection)

Searching in a heap

Clearly, this is unacceptable .... a sequential file, sorted on author names would be nice ...

some other uses of sorted data:

- presentation of results,
- computing aggregates,
- duplicate elimination,
- algorithms for RA operations (e.g., join and projection)

So, let's consider how to efficiently sort in external memory

Topic 2: External sorting

Sorting: two-way external merge sort

Suppose we have 3 buffer frames available, and 2^k pages in the file to be sorted.

Sorting: two-way external merge sort

Suppose we have 3 buffer frames available, and 2^k pages in the file to be sorted.

(0) read each page in, sort in main-memory, and write it out. result: 2^k sorted runs of length 1.

Sorting: two-way external merge sort

Suppose we have 3 buffer frames available, and 2^k pages in the file to be sorted.

(0) read each page in, sort in main-memory, and write it out. result: 2^k sorted runs of length 1.

(1) read in two runs, merge sort, and write out. result: 2^{k-1} sorted runs of length 2.

Sorting: two-way external merge sort

Suppose we have 3 buffer frames available, and 2^k pages in the file to be sorted.

(0) read each page in, sort in main-memory, and write it out. result: 2^k sorted runs of length 1.

(1) read in two runs, merge sort, and write out. result: 2^{k-1} sorted runs of length 2.

(2) read in two runs, merge sort, and write out. result: 2^{k-2} sorted runs of length 4.

Sorting: two-way external merge sort

Suppose we have 3 buffer frames available, and 2^k pages in the file to be sorted.

(0) read each page in, sort in main-memory, and write it out. result: 2^k sorted runs of length 1.

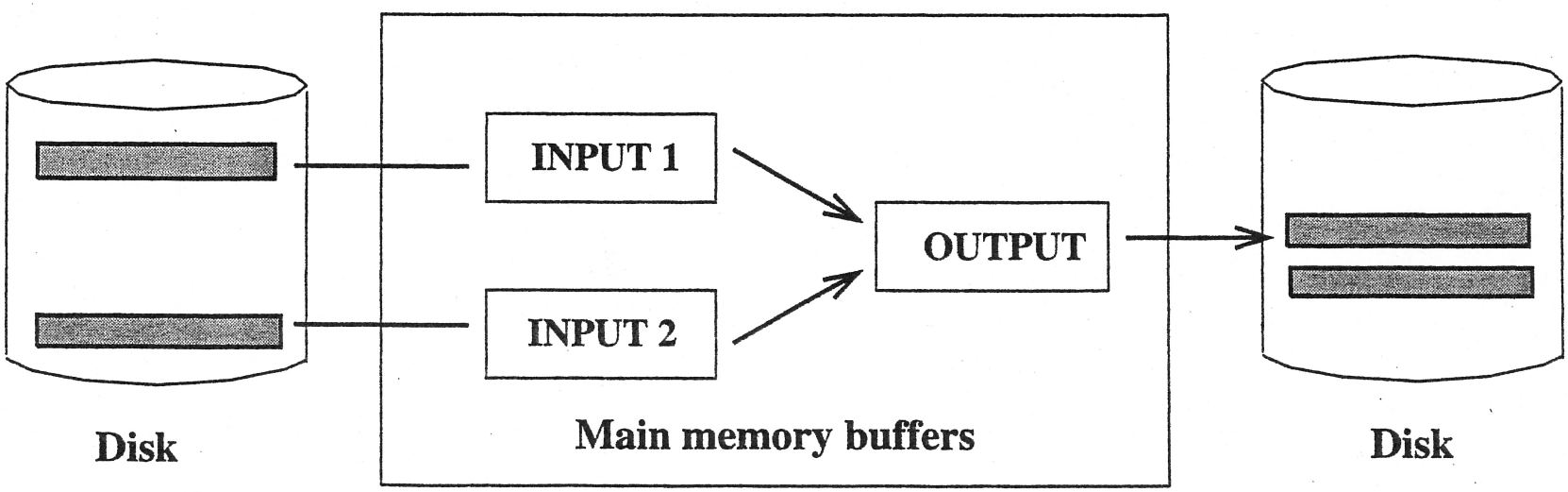
(1) read in two runs, merge sort, and write out. result: 2^{k-1} sorted runs of length 2.

(2) read in two runs, merge sort, and write out. result: 2^{k-2} sorted runs of length 4.

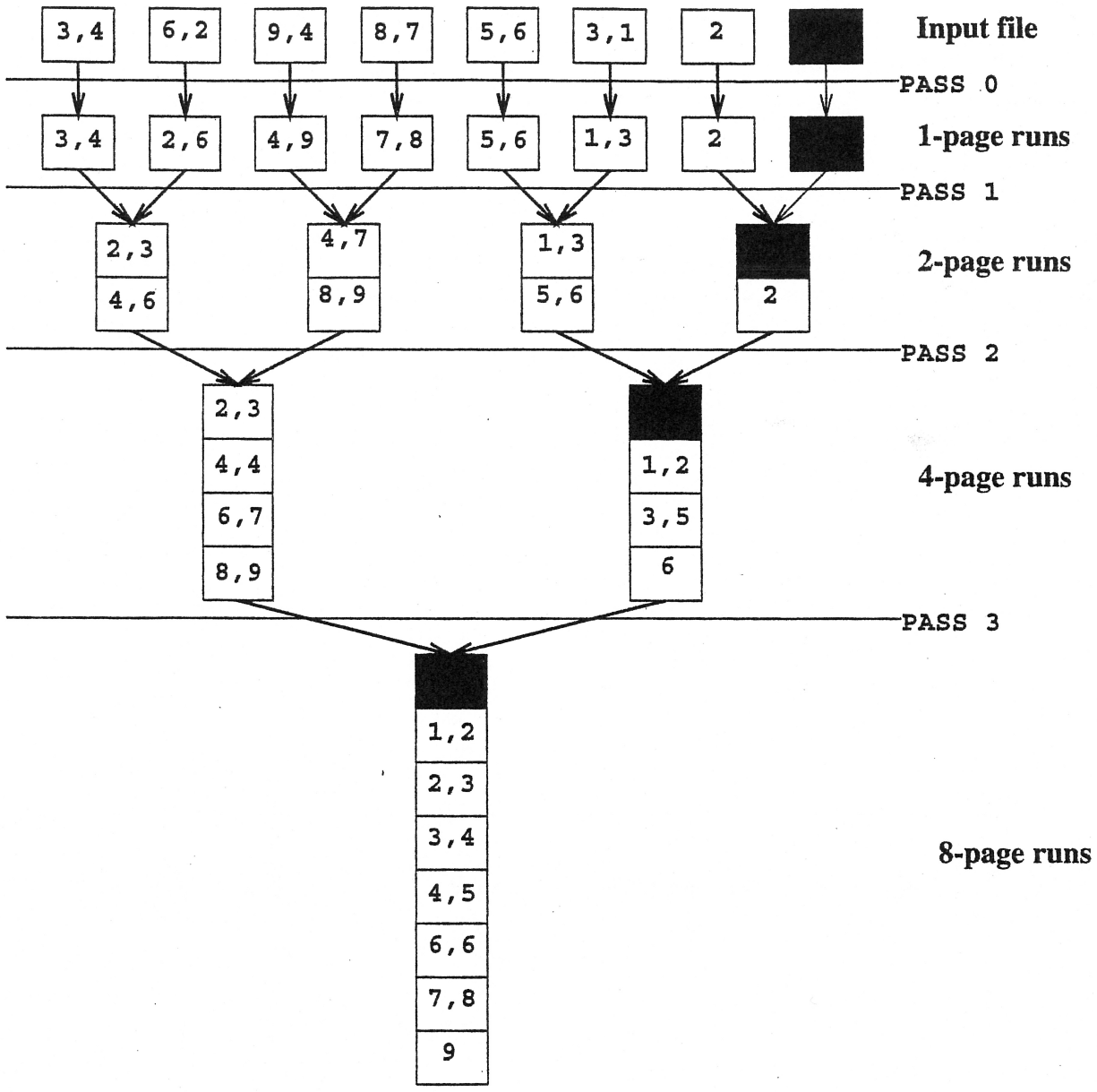
$\vdots$

(k) read in two runs, merge sort, and write out. result: 2^0 sorted runs of length 2^k.

Sorting: two-way external merge sort



Sorting: two-way external merge sort



Sorting: two-way external merge sort

So, with 2 I/Os per page, per pass, we have

$$2B(\lceil \log B \rceil + 1)$$

I/Os for two-way merge sort

Sorting: two-way external merge sort

So, with 2 I/Os per page, per pass, we have

$$2B(\lceil \log B \rceil + 1)$$

I/Os for two-way merge sort

In our example, one scan was 25 minutes, giving us 50 minutes per pass, and therefore

$$\begin{aligned} 50 \times (\lceil \log 100,000 \rceil + 1) &= 50 \times 18 \\ &= 900 \text{ minutes} \\ &= 15 \text{ hours} \end{aligned}$$

Sorting: two-way external merge sort

So, with 2 I/Os per page, per pass, we have

$$2B(\lceil \log B \rceil + 1)$$

I/Os for two-way merge sort

In our example, one scan was 25 minutes, giving us 50 minutes per pass, and therefore

$$\begin{aligned} 50 \times (\lceil \log 100,000 \rceil + 1) &= 50 \times 18 \\ &= 900 \text{ minutes} \\ &= 15 \text{ hours} \end{aligned}$$

a disaster.

Sorting

A disaster indeed ....

But what if we have N buffer pages available (for some $N > 3$)?

We can do an $(N - 1)$-way merge sort. Perhaps this will help?

Sorting

A disaster indeed ....

But what if we have N buffer pages available (for some $N > 3$)?

We can do an $(N - 1)$-way merge sort. Perhaps this will help?

Why $N - 1$?

Sorting: external merge sort

Suppose we have N buffer frames available.

- (pass 0) read in N pages at a time, sort in main-memory, and write out. result: $\lceil \frac{B}{N} \rceil$ sorted runs of length N.

Sorting: external merge sort

Suppose we have N buffer frames available.

- (pass 0) read in N pages at a time, sort in main-memory, and write out. result: $\lceil \frac{B}{N} \rceil$ sorted runs of length N.
- (pass $1, 2, \ldots$) Use $N - 1$ buffer frames to do an $N - 1$ way merge sort

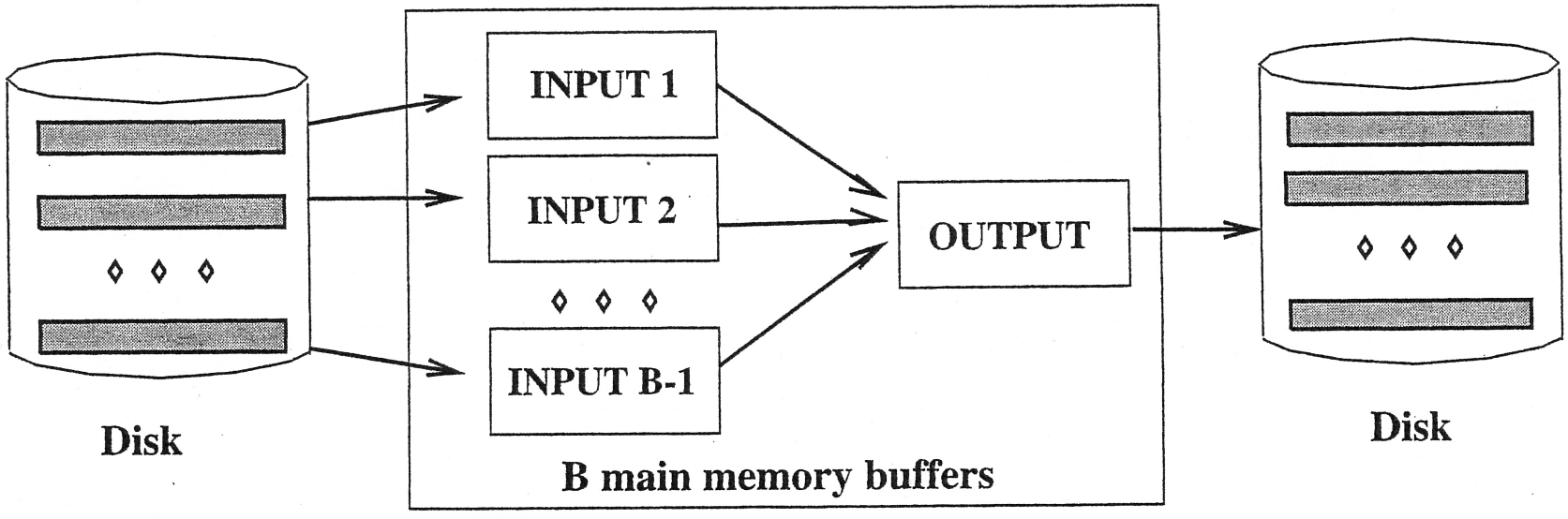
Sorting: external merge sort

Suppose we have N buffer frames available.

- (pass 0) read in N pages at a time, sort in main-memory, and write out. result: $\lceil \frac{B}{N} \rceil$ sorted runs of length N.
- (pass $1, 2, \ldots$) Use $N - 1$ buffer frames to do an $N - 1$ way merge sort

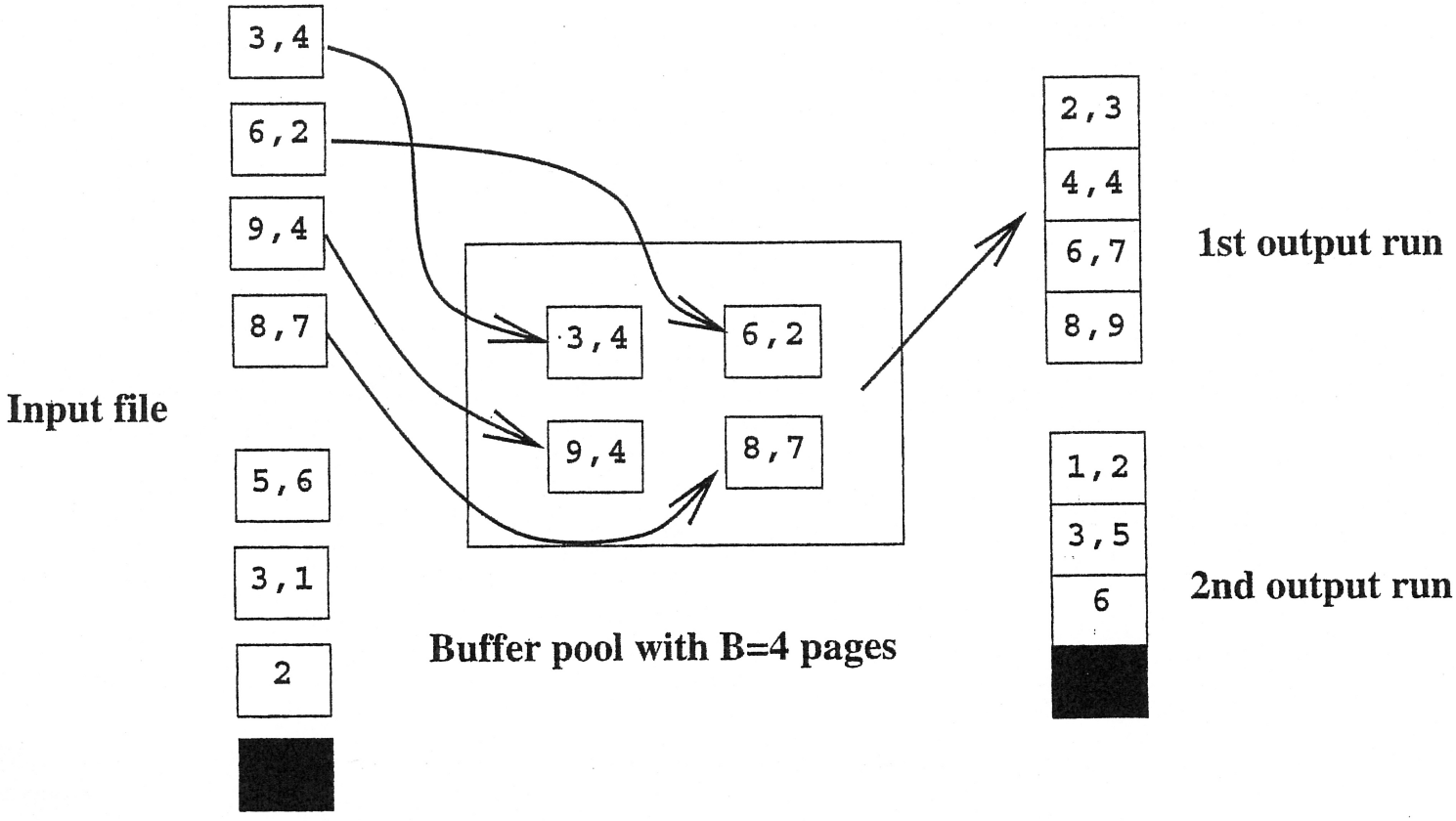
Number of passes: $\lceil \log_{N-1} \lceil \frac{B}{N} \rceil \rceil + 1$

Sorting: external merge sort



External Merge Sort with B Buffer Pages: Pass $i > 0$

Sorting: external merge sort



External Merge Sort with B Buffer Pages: Pass 0

Sorting: external merge sort

Suppose $N = 101$, and let's revisit our example.

$$\begin{aligned} 50 \times (\lceil \log_{100} \left\lceil \frac{100,000}{101} \right\rceil \rceil + 1) &= 50 \times (\lceil 1.5 \rceil + 1) \\ &= 150 \text{ minutes} \\ &= 2.5 \text{ hours} \end{aligned}$$

Not too shabby

Sorting: external merge sort

B	$N = 3$	$N = 5$	$N = 9$	$N = 17$	$N = 129$	$N = 257$
100	7	4	3	2	1	1
1000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Number of passes of external merge sort

Exercise

Suppose we have $N = 3$ buffer frames available and at most two records fit on a block. Given the file

$$\langle [9, 8], [2, 1], [7, 6], [4, 3], [5, 10], [5, 1], [4, 11], [12] \rangle$$

show each step of performing an external merge sort on the file.

Topic 3: Ordered indexes

Indexing

- key: a collection of attributes (of a relation)

Indexing

- key: a collection of attributes (of a relation)
- index:
  - data structure for searching (i.e., mapping key values to data)

Indexing

- key: a collection of attributes (of a relation)
- index:
  - data structure for searching (i.e., mapping key values to data)
  - i.e., collection of data entries $k*$ for search key k

Indexing

we are of course primarily interested in *external* search

key value $\rightarrow$ index
$\rightarrow$ blocks holding records
$\rightarrow$ matching records

Indexing

we are of course primarily interested in *external* search

key value $\rightarrow$ index
$\rightarrow$ blocks holding records
$\rightarrow$ matching records

Example. Find all bookstore tuples with "*City* = `Eindhoven`"

Indexing

- *basic operations:*
  - search(k)
  - insert(k)
  - delete(k)

Indexing

- *basic operations:*
  - search(k)
  - insert(k)
  - delete(k)
- also known as: dictionaries or symbol tables
  - *ex:* linked-list, binary search tree, AVL tree, skip list, hash table

Index entries

- index = collection of data entries $k*$ for search key k
- three main alternatives for $k*$

Index entries

- index = collection of data entries $k*$ for search key k
- three main alternatives for $k*$
  1. $k*$ is actual data record for k

Index entries

- index = collection of data entries $k*$ for search key k
- three main alternatives for $k*$
  1. $k*$ is actual data record for k
  2. $k* = \langle k, rid \rangle$, where *rid* is a record-id of a data record with key k

Index entries

- index = collection of data entries $k*$ for search key k
- three main alternatives for $k*$
  1. $k*$ is actual data record for k
  2. $k* = \langle k, rid \rangle$, where *rid* is a record-id of a data record with key k
  3. $k* = \langle k, ridList \rangle$, where *ridList* is a list of record-id's of data records with key k

Index entries

- index = collection of data entries $k*$ for search key k
- three main alternatives for $k*$
  1. $k*$ is actual data record for k
  2. $k* = \langle k, rid \rangle$, where *rid* is a record-id of a data record with key k
  3. $k* = \langle k, ridList \rangle$, where *ridList* is a list of record-id's of data records with key k
- alternative (1) is called an indexed file organization
- alternatives (2) and (3) are independent of the underlying file's organization

Indexing

Evaluation criteria

- construction cost

Indexing

Evaluation criteria

- construction cost
- update cost: insertion/deletion

Indexing

Evaluation criteria

- construction cost
- update cost: insertion/deletion
- space overhead

Indexing

Evaluation criteria

- construction cost
- update cost: insertion/deletion
- space overhead
- access types

Indexing

Evaluation criteria

- construction cost
- update cost: insertion/deletion
- space overhead
- access types
- access cost

Indexing

Two basic types of indexes

- Ordered indexes. based on a sorted ordering of the values

Indexing

Two basic types of indexes

- Ordered indexes. based on a sorted ordering of the values
- Hash indexes. based on a uniform distribution of values across a range of "buckets." The bucket to which a value is assigned is determined by a hash function. (e.g., integer value of key, modulo index size)

Ordered indexes

Based on a sorted ordering of the values, just like the index of a book

Ordered indexes

Based on a sorted ordering of the values, just like the index of a book

- clustering (clustered, primary): search key also defines the sequential order of the file
  - i.e., data records in file are sorted in the same order as in the index (on the search key fields)

Ordered indexes

Based on a sorted ordering of the values, just like the index of a book

- clustering (clustered, primary): search key also defines the sequential order of the file
  - i.e., data records in file are sorted in the same order as in the index (on the search key fields)
- nonclustering (nonclustered, secondary): search key specifies an order different from the sequential order of the file

Ordered indexes

two types:

Ordered indexes

two types:

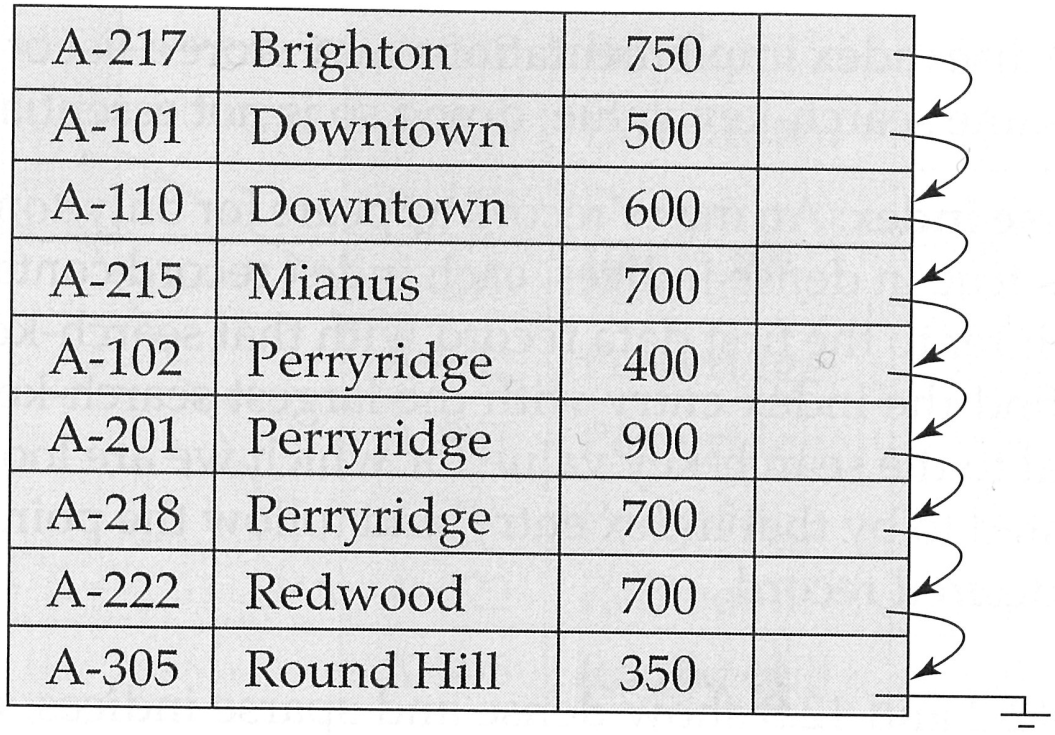
- dense: an index record appears for *every* search-key value in the file

Ordered indexes

two types:

- dense: an index record appears for *every* search-key value in the file
- sparse: an index record appears for only *some* of the search-key values in the file

Ordered indexes



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

Figure 12.1 Sequential file for *account* records.

Ordered indexes: dense clustering index

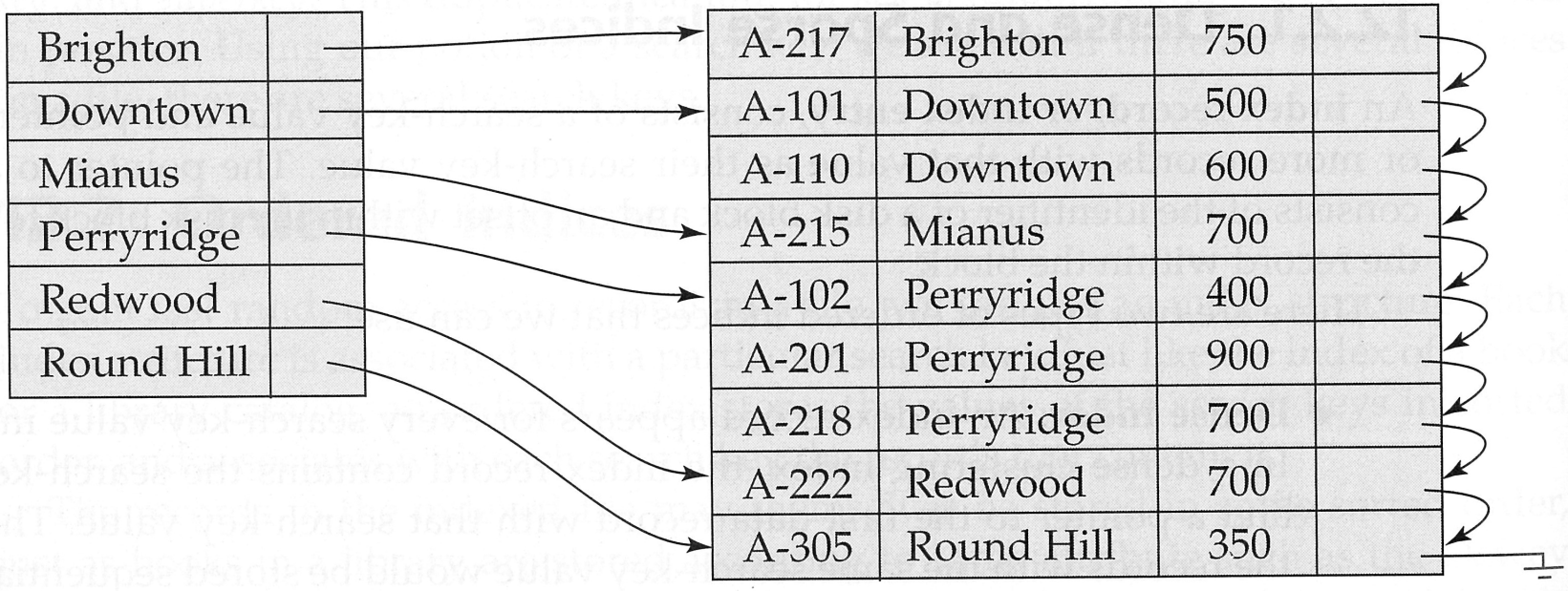


Figure 12.2 Dense index.

Ordered indexes: sparse clustering index

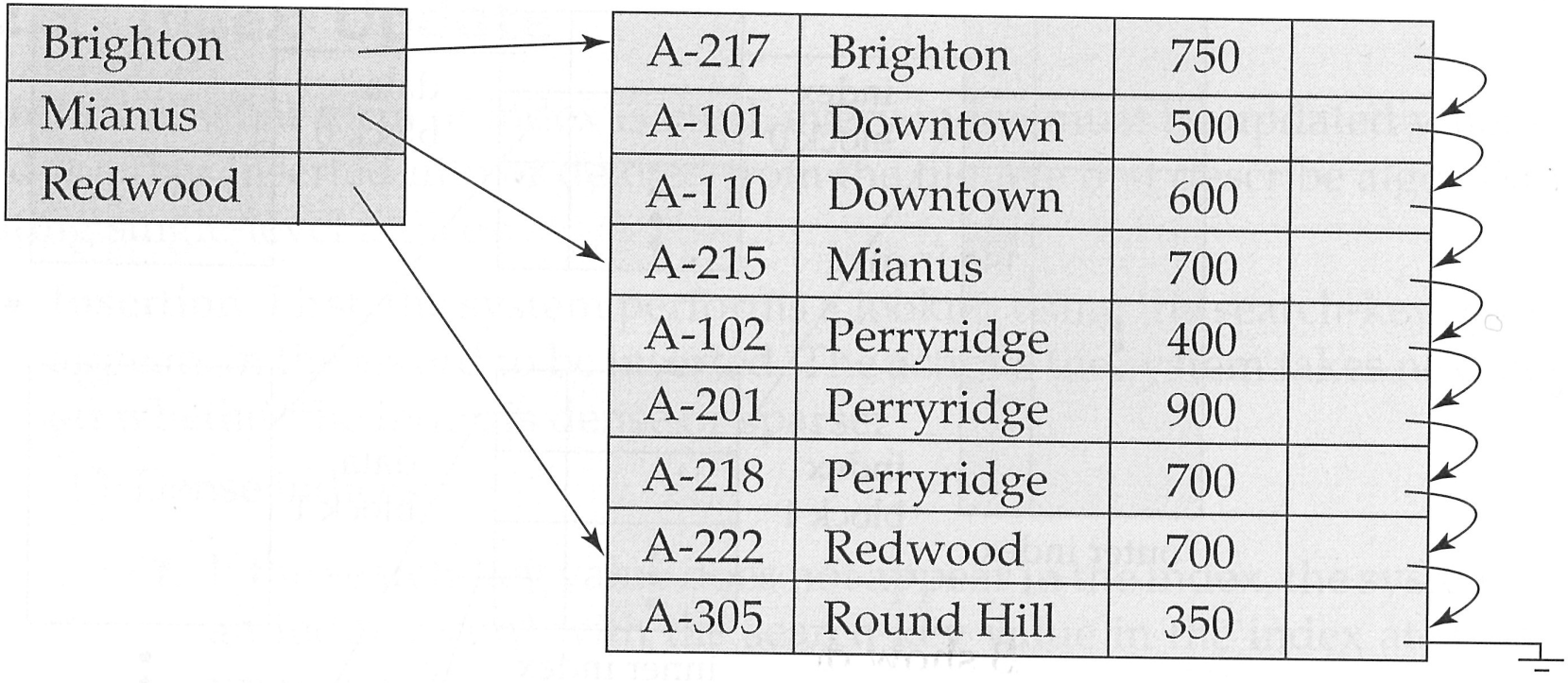


Figure 12.3 Sparse index.

Nonclustering index

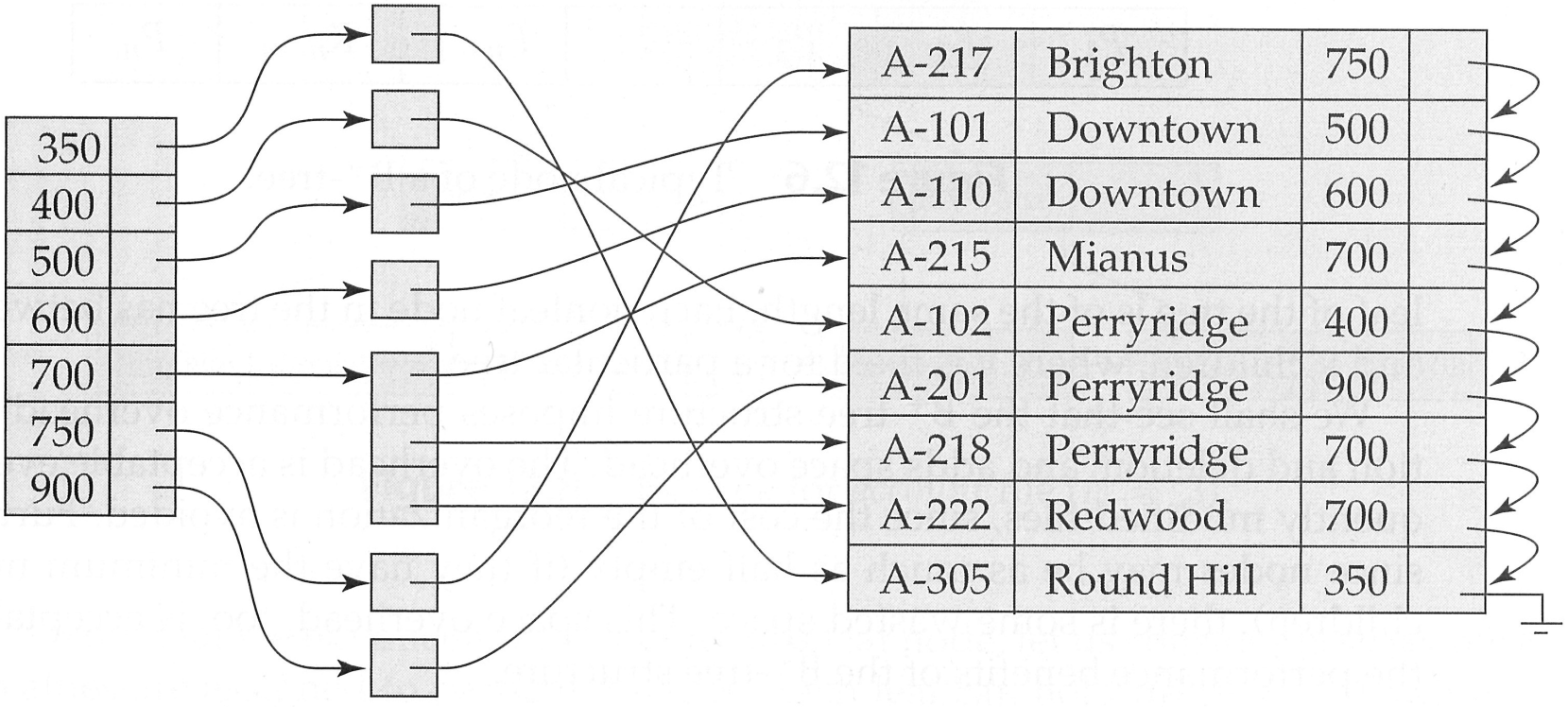


Figure 12.5 Secondary index on *account* file, on noncandidate key *balance*.

Nonclustering index

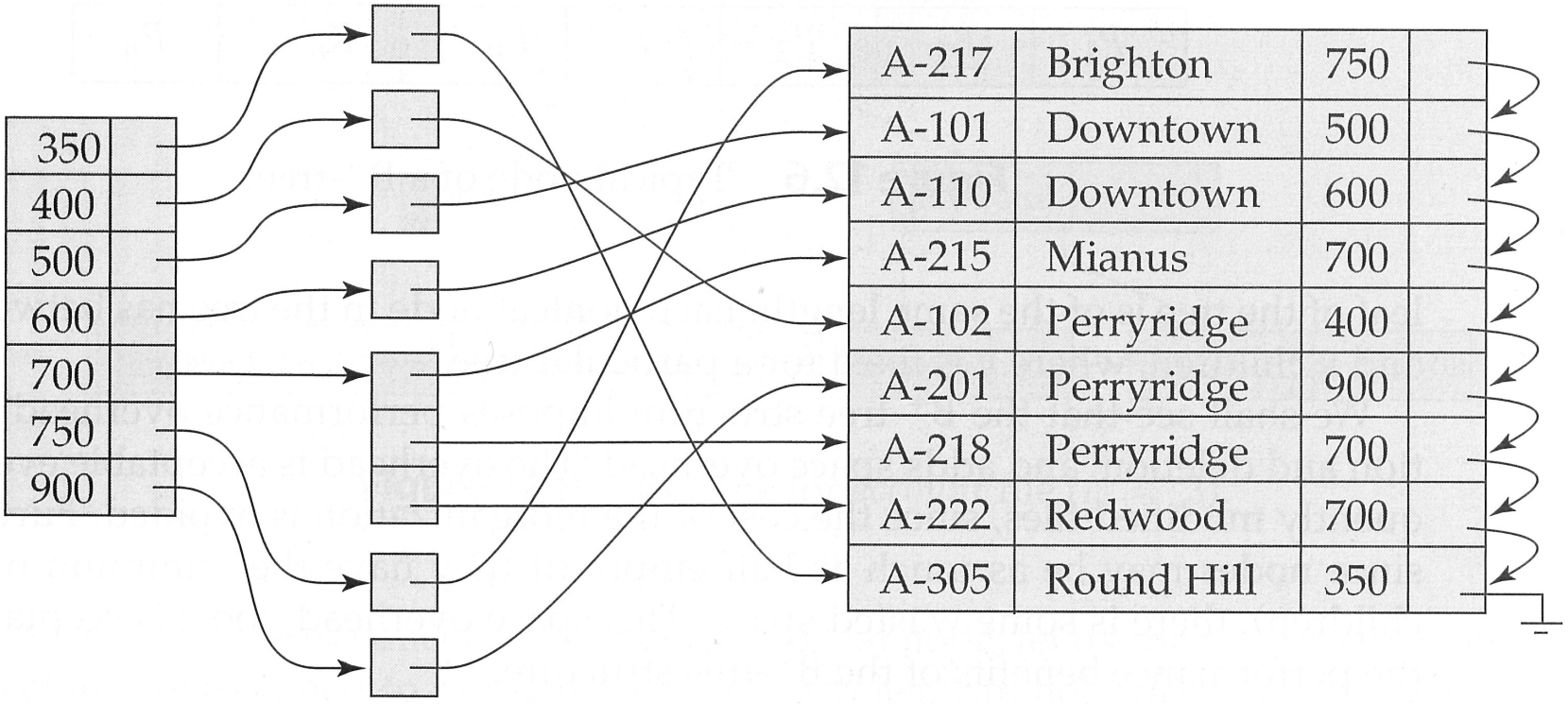


Figure 12.5 Secondary index on *account* file, on noncandidate key *balance*.

Must be dense (why?)

Two-level sparse index

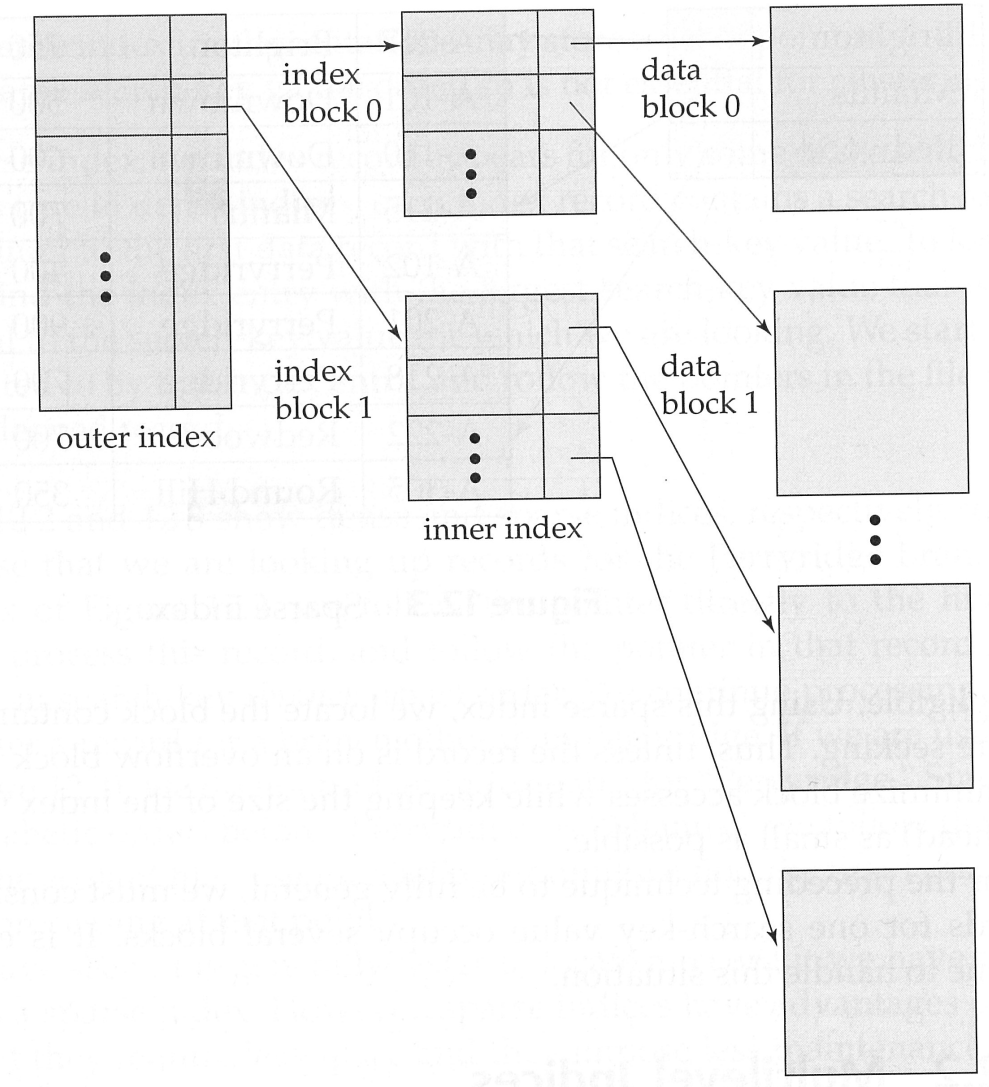


Figure 12.4 Two-level sparse index.

Ordered indexing

Evaluation criteria

- access types
- access cost
- update cost: insertion/deletion
- space overhead

Ordered indexing

- performance of index-sequential file organization degrades as the file grows, necessitating reorganization

Ordered indexing

- performance of index-sequential file organization degrades as the file grows, necessitating reorganization
- multi-level indexing as indexing an index ....
- isn't this a search tree?

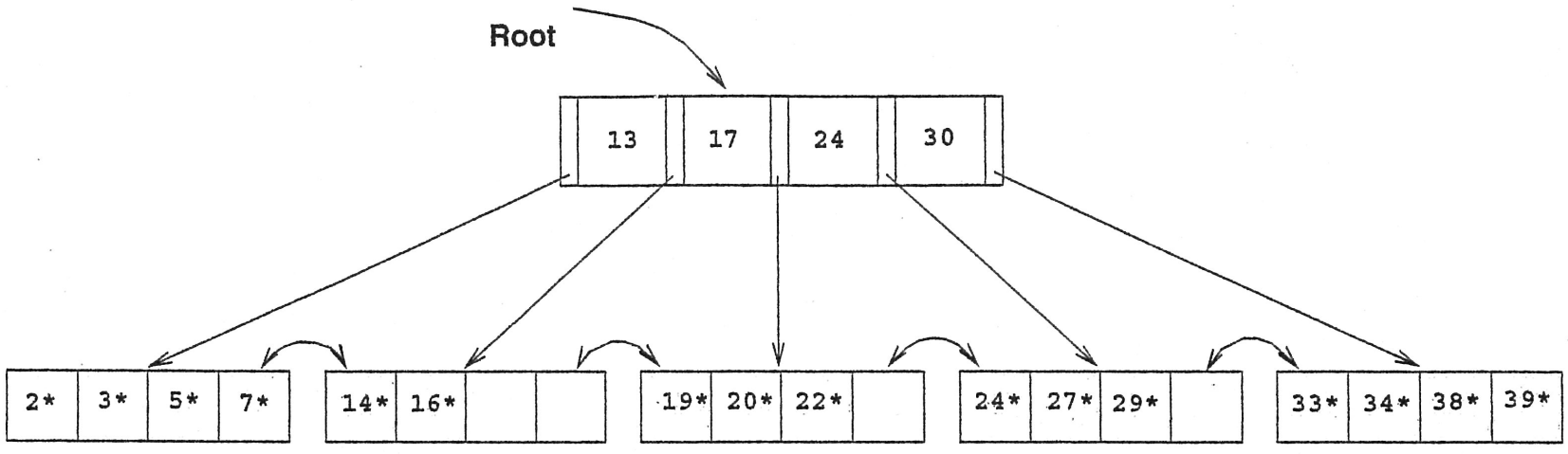
Ordered indexing: B+ trees

- height-balanced, dynamic search tree
- all $k*$ at the leaf level
- generalizes multi-level sparse index structure

Ordered indexing: B+ trees

- height-balanced, dynamic search tree
- all $k*$ at the leaf level
- generalizes multi-level sparse index structure
- leaf level can be sparse or dense
- designed to minimize I/O
- efficient equality and range search

Ordered indexing: B+ trees



B+ trees: structure

- node structure:
  $[P_1, K_1, P_2, K_2, \cdots, P_{n-1}, K_{n-1}, P_n]$
- $n - 1$ search-key values K_i
- n pointers P_i
- if $i < j$, then $K_i < K_j$

B+ trees: structure

- node structure:
  $[P_1, K_1, P_2, K_2, \cdots, P_{n-1}, K_{n-1}, P_n]$
- $n-1$ search-key values K_i
- n pointers P_i
- if $i < j$, then $K_i < K_j$

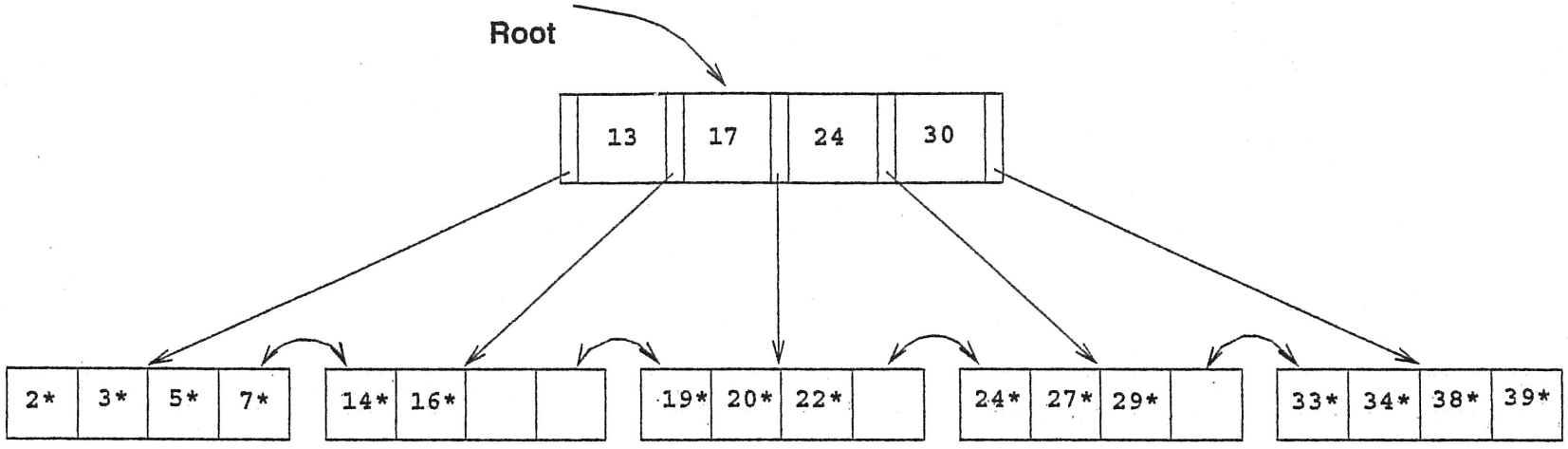
- leaf nodes:
  - P_i points to records with key value K_i
  - P_n points to next sibling leaf, forming *sequence set*, supporting sequential search
  - must contain at least $\lceil (n-1)/2 \rceil$ search-key values

B+ trees: structure

- node structure:
  $[P_1, K_1, P_2, K_2, \cdots, P_{n-1}, K_{n-1}, P_n]$
- $n - 1$ search-key values K_i
- n pointers P_i
- if $i < j$, then $K_i < K_j$

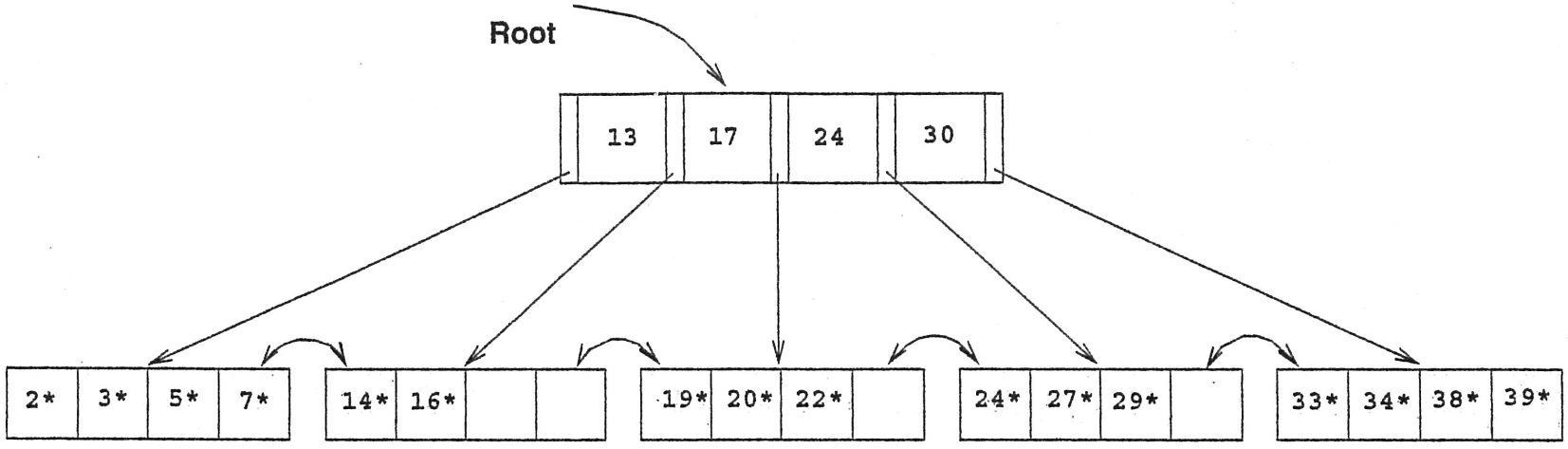
- internal nodes:
  - P_i points to records with key value less than K_i
  - P_{i+1} points to records with key value greater than or equal to K_i
  - must contain at least $\lceil n/2 \rceil$ pointers (i.e., fanout)
    - except the root, which can have fewer, with a minimum of two pointers

B+ trees: structure



order $n = 5$

B+ trees: structure



order $n = 5$

– Internal nodes support random access
– Sequence set supports ordered access

B+ trees: search

Search(key k, node N)

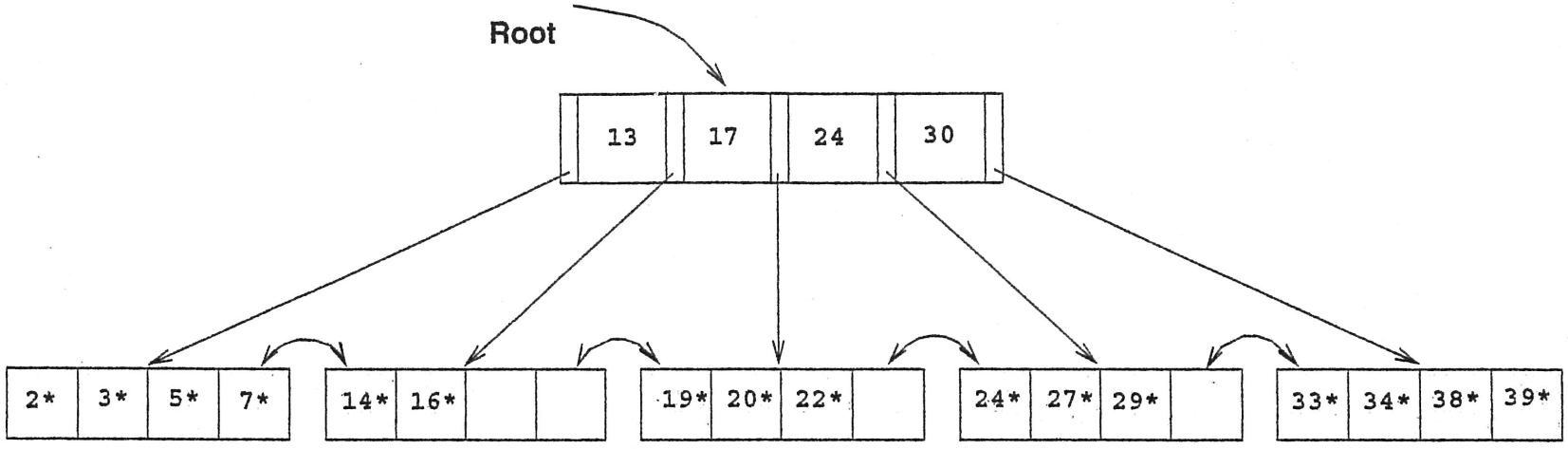
- if $k = K_i \in N$
  - if N is a leaf, return P_i
  - else return Search(k, P_{i+1})

B+ trees: search

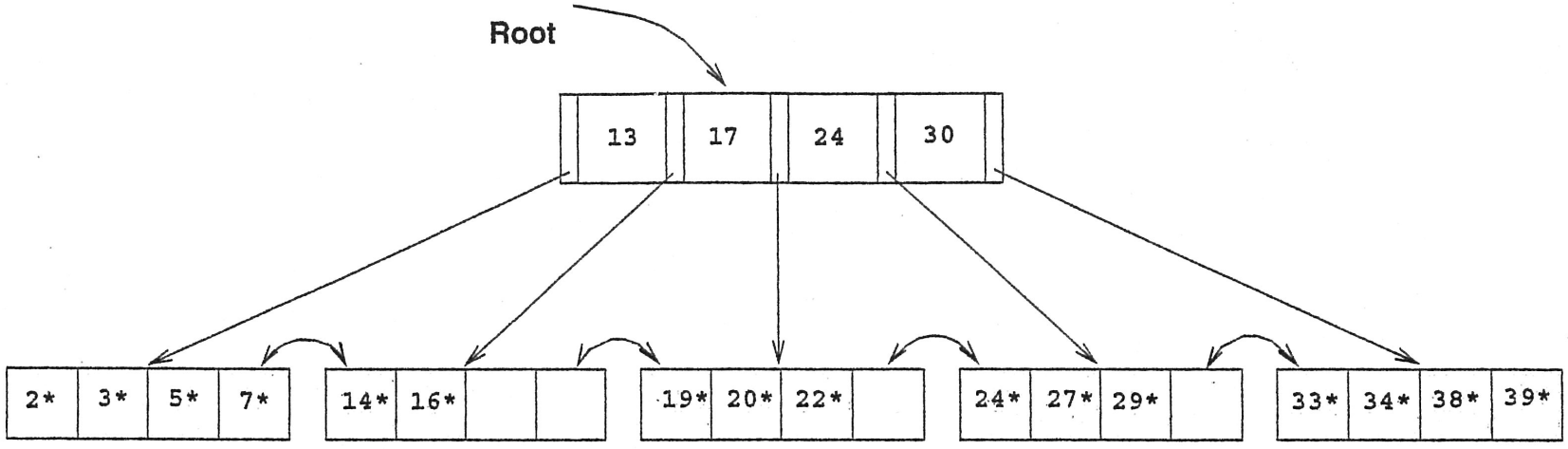
Search(key k, node N)

- if $k = K_i \in N$
  - if N is a leaf, return P_i
  - else return Search(k, P_{i+1})
- if $k \notin N$
  - if N is a leaf, return FAIL
  - else, find $K_i \in N$ such that K_i is minimal in node satisfying $k < K_i$, and return Search(k, P_i)

B+ trees: search

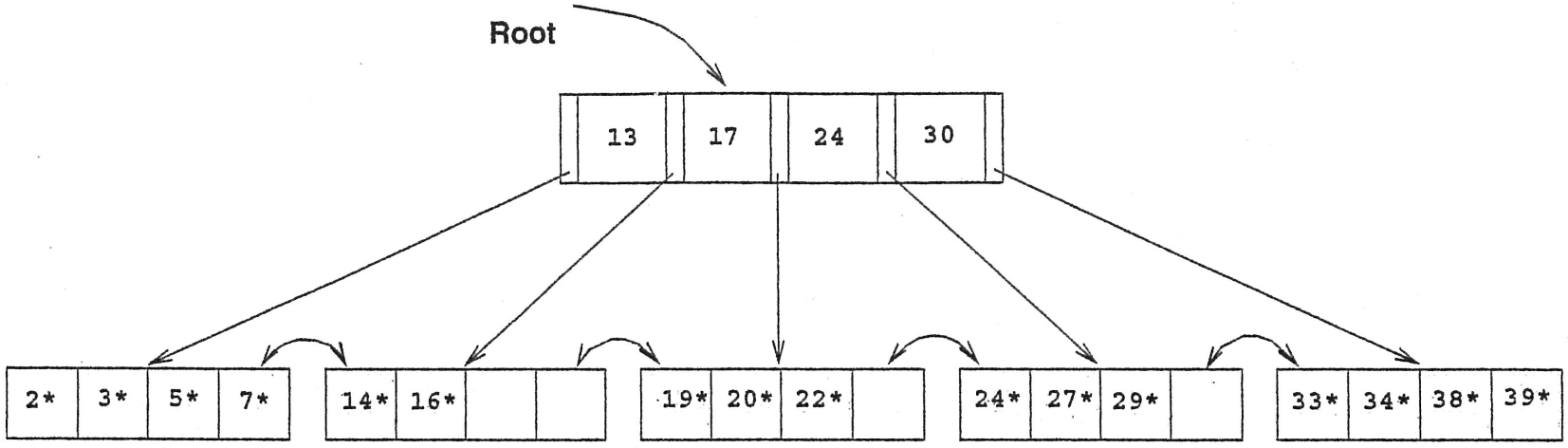


B+ trees: search



Search in $\mathcal{O}(\log_{\lceil n/2 \rceil} N)$, for tree of N search-key values.

B+ trees: search



Search in $\mathcal{O}(\log_{\lceil n/2 \rceil} N)$, for tree of N search-key values.

So, if $n = 100$ (a conservative estimate in practice), then locating an item in an index of $N = 1,000,000$ entries requires $\lceil \log_{50} 1,000,000 \rceil = 4$ I/Os.

B+ trees: insertion

Insert(key k, node N, record r)

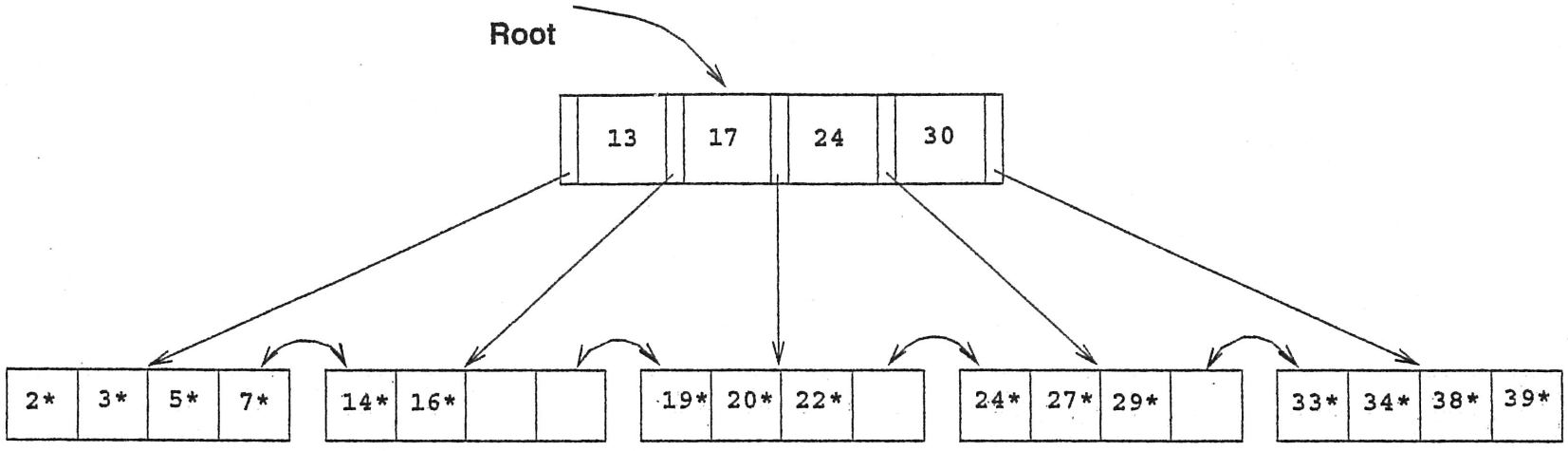
- if $k = K_i \in N$
  - if N is a leaf, return FAIL (no duplicates allowed)
  - else return Insert(k, P_{i+1}, r)

B+ trees: insertion

Insert(key k, node N, record r)

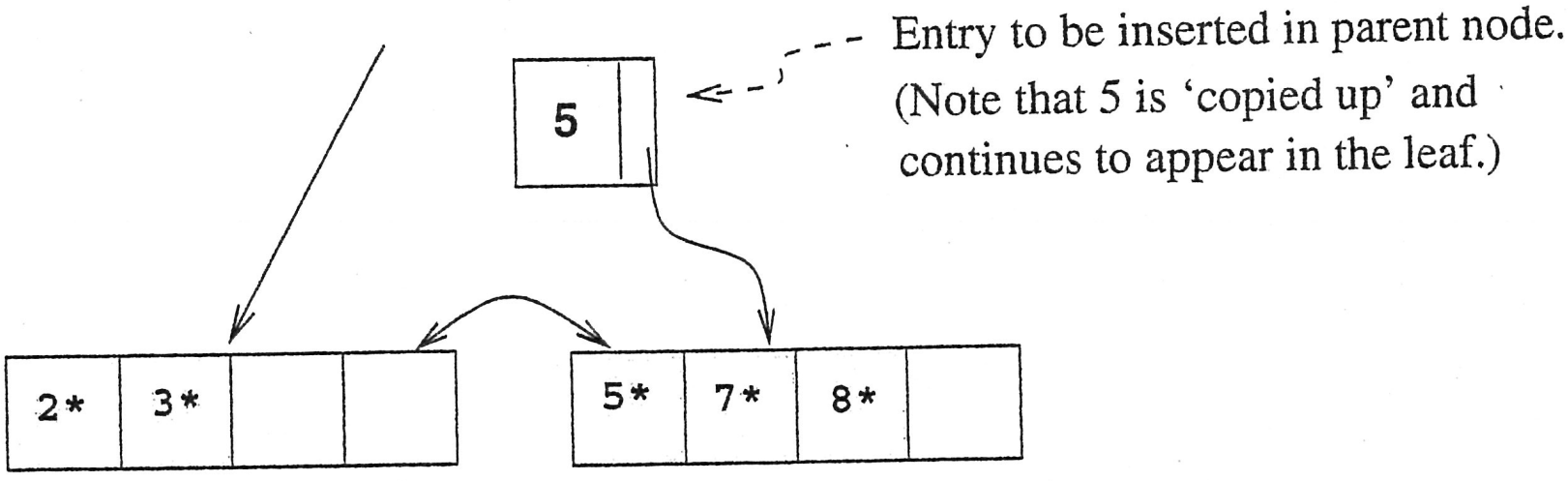
- if $k = K_i \in N$
  - if N is a leaf, return FAIL (no duplicates allowed)
  - else return Insert(k, P_{i+1}, r)
- if $k \notin N$
  - if N is a leaf
    - if there is space, insert k and r
    - else, split N into N and N', redistribute entries evenly between them, and insert index entry for N' into parent of N
  - else, find $K_i \in N$ such that K_i is minimal in node satisfying $k < K_i$, and return Insert(k, P_i, r)

B+ trees: insertion



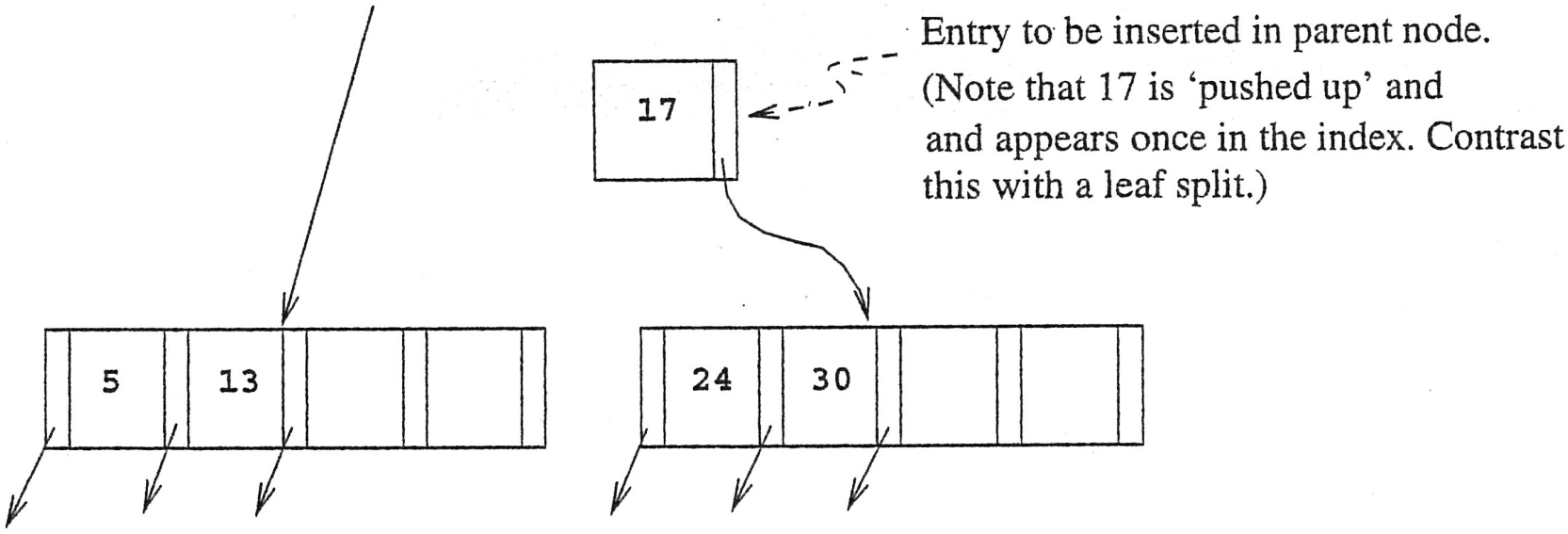
lets insert 8*

B+ trees: insertion



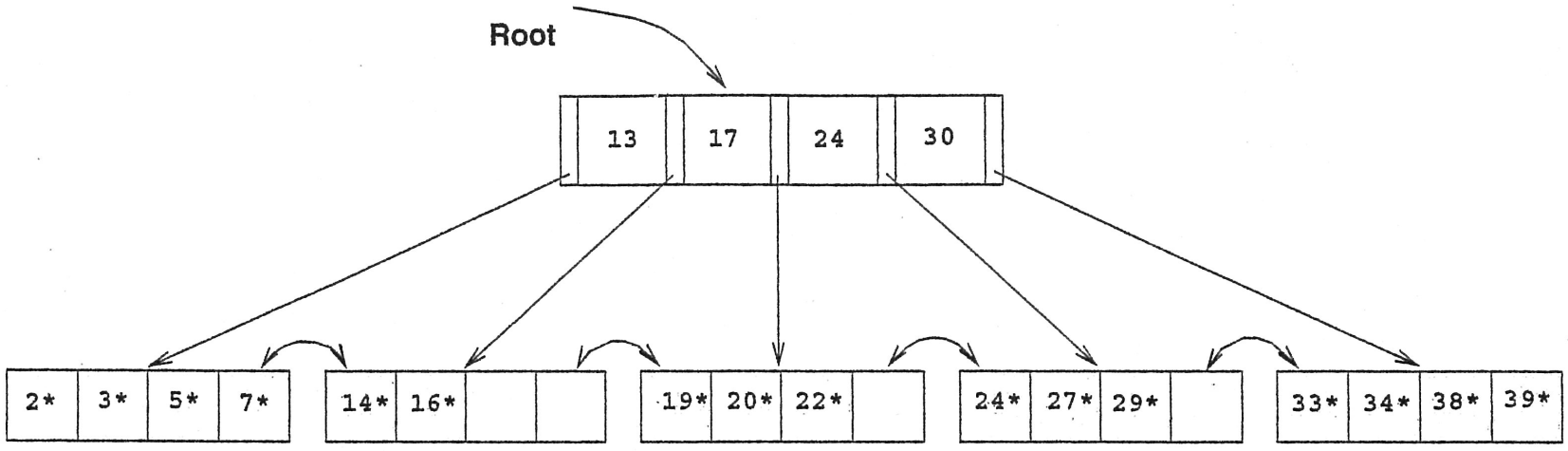
Split leaf pages during insertion of 8*

B+ trees: insertion



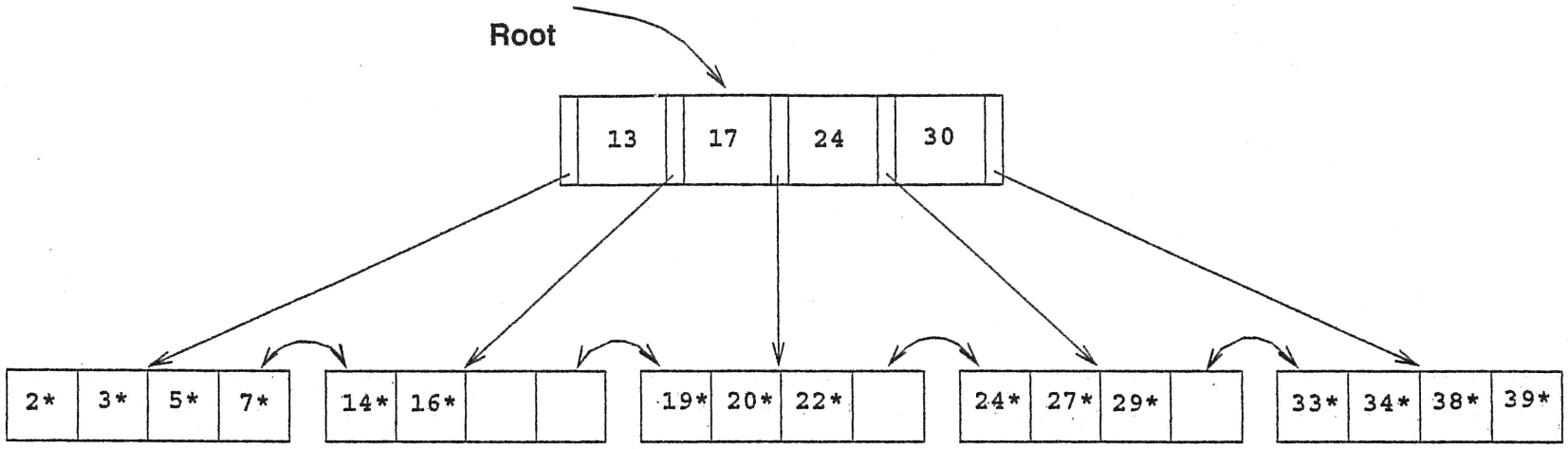
Split index pages during insertion of 8*

B+ trees: insertion

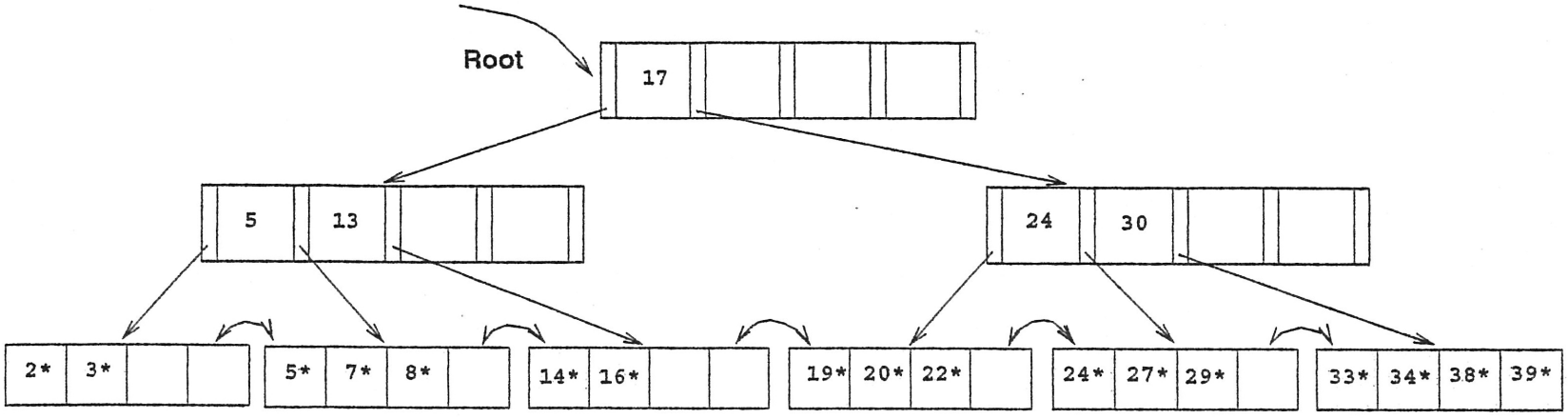


before ...

B+ trees: insertion



before ...

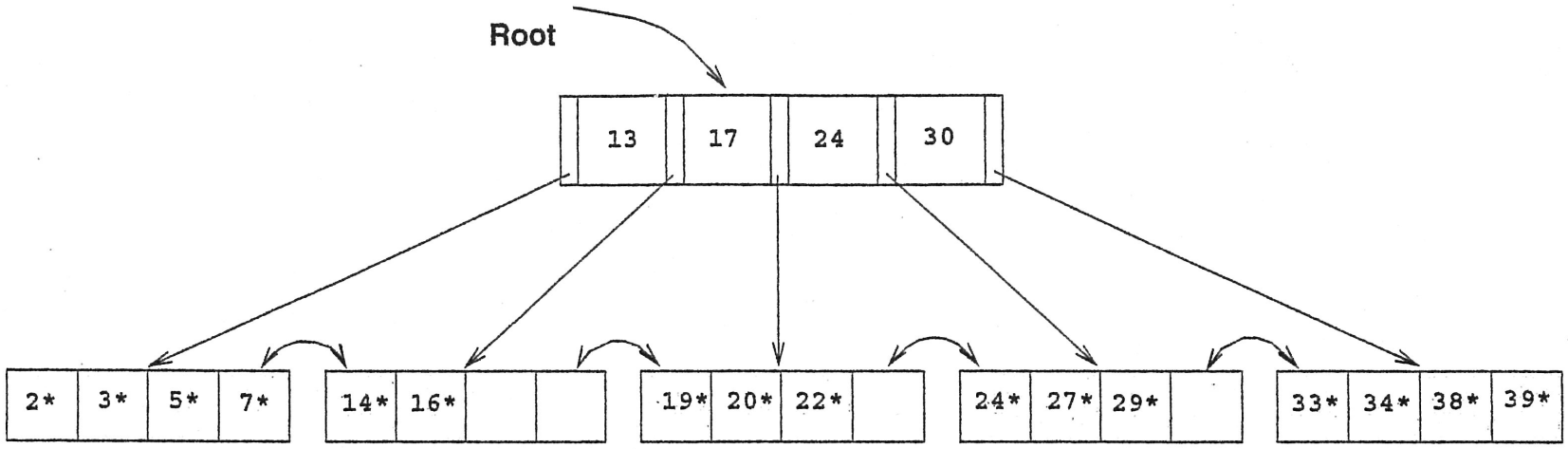


... and after inserting 8*

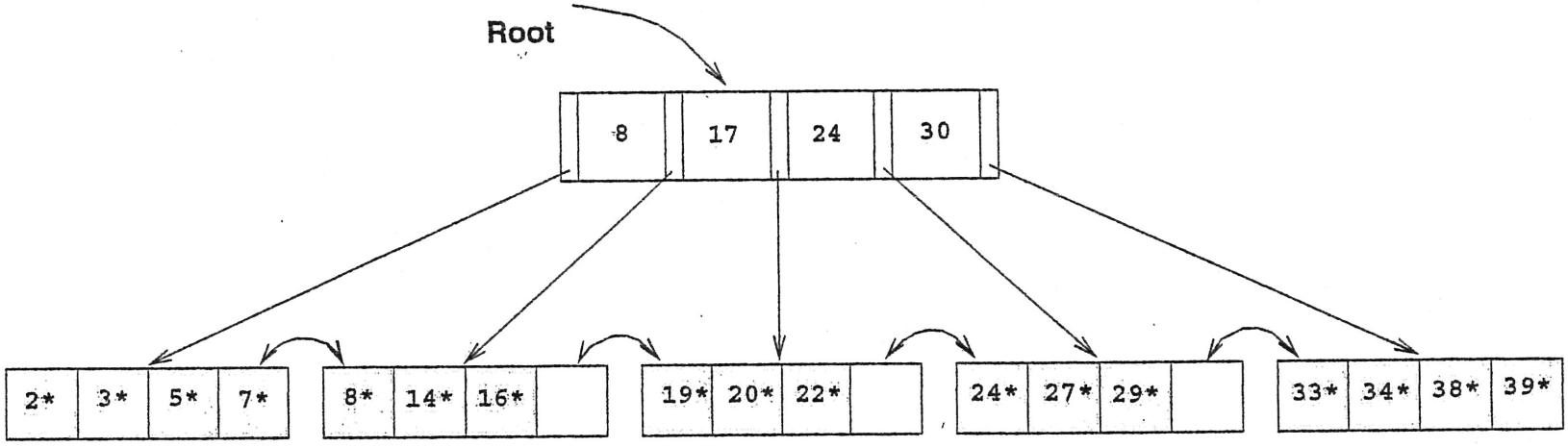
B+ trees: insertion

If space is available, we can also redistribute overflow to siblings, delaying (recursive) splitting

B+ trees: insertion



before ...



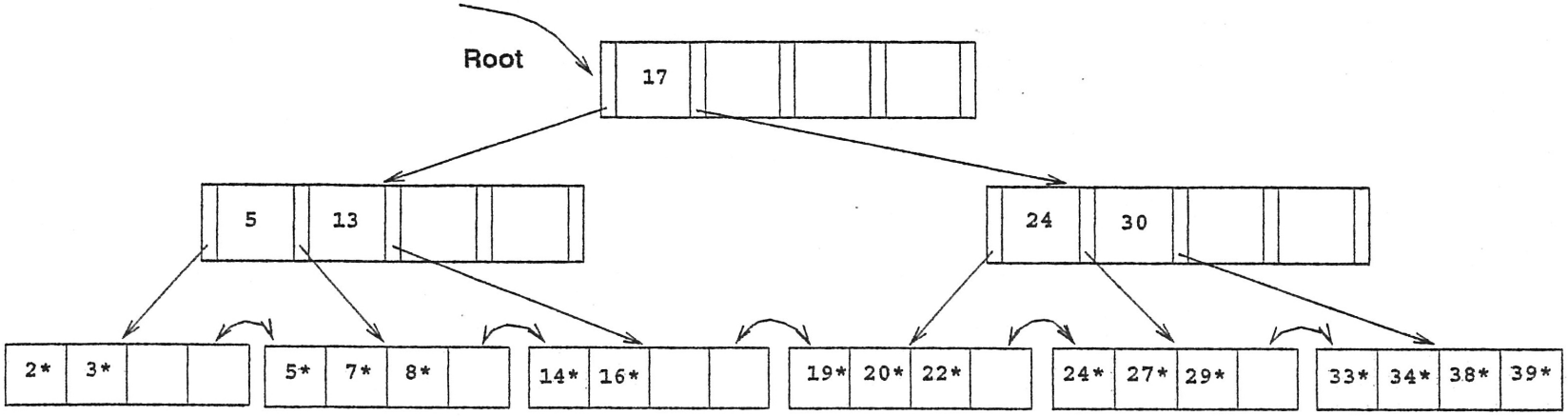
... and after inserting 8* using redistribution

B+ trees: deletion

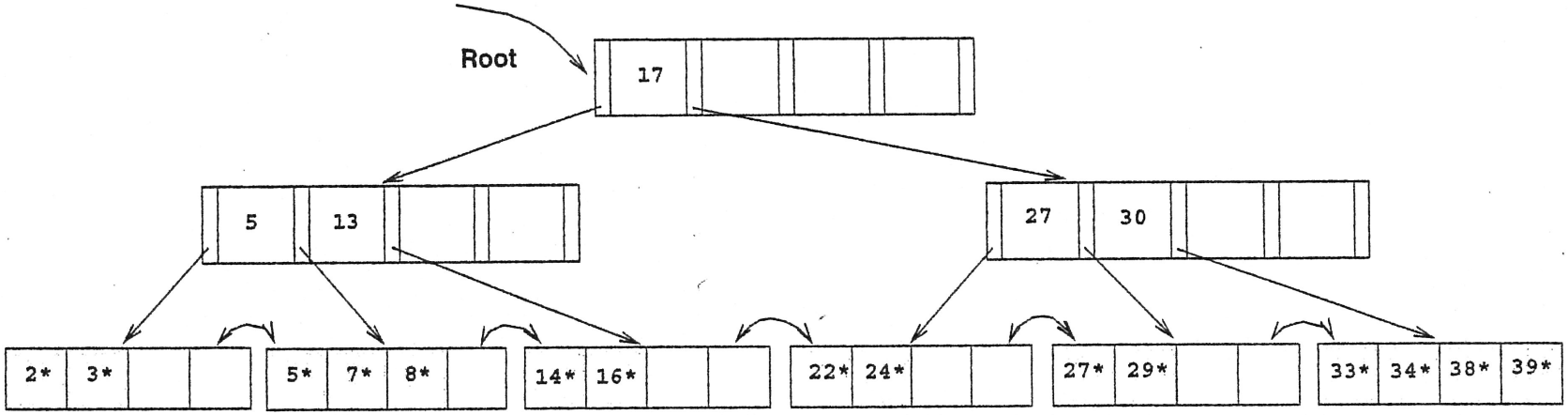
Delete(key k, node N)

- find the leaf node L where k belongs
  - remove entry
  - if L is still at least half full, return SUCCESS
  - else if possible, borrow entry from adjacent sibling
  - else, merge L and sibling, and delete entry for sibling from parent

B+ trees: deletion

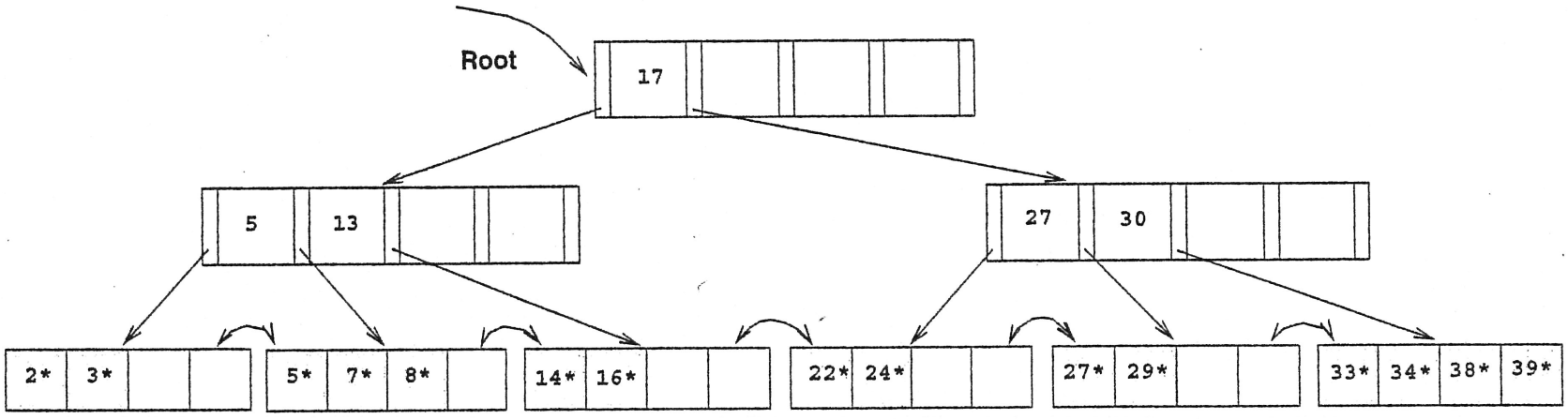


Before ....



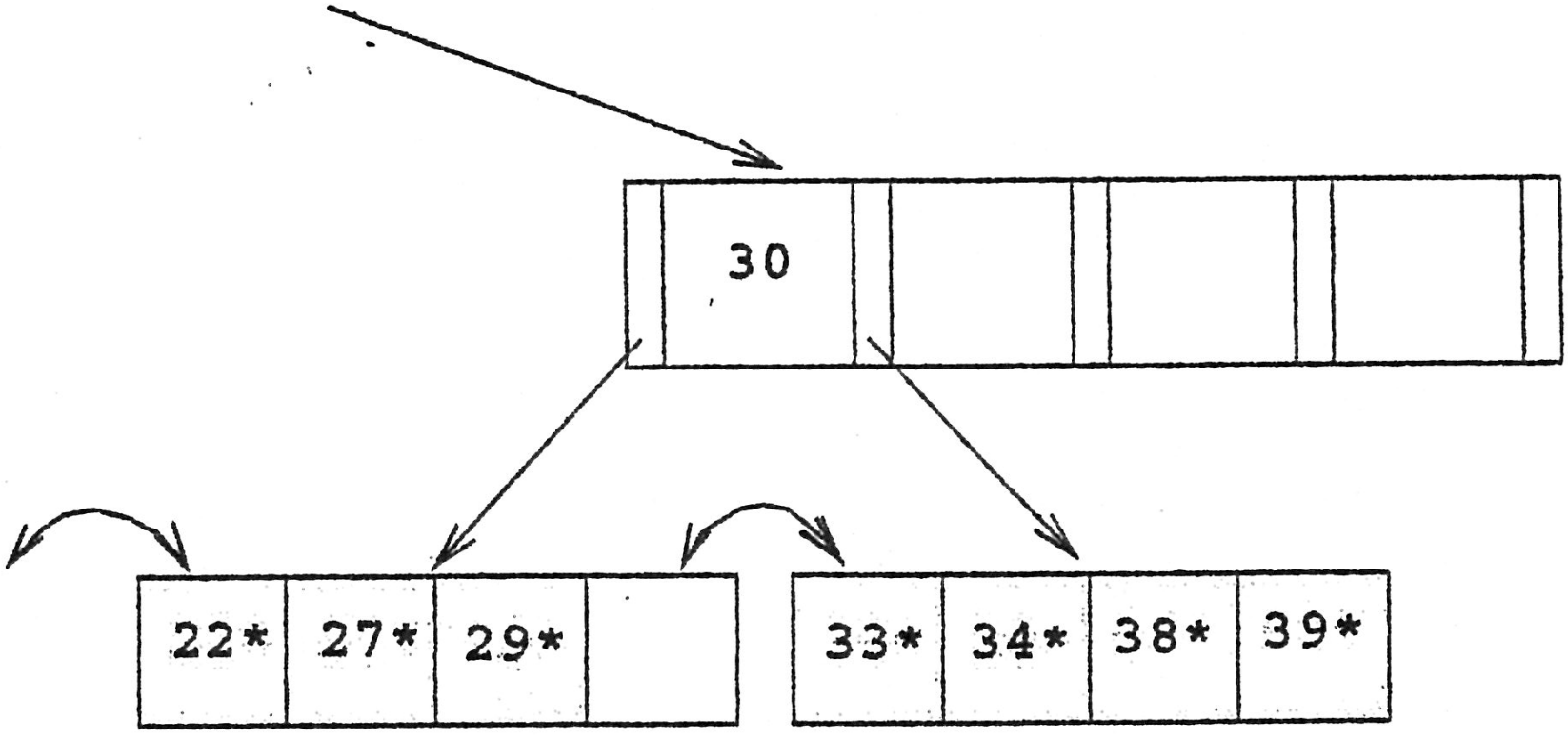
... and after deleting 19* and 20*

B+ trees: deletion



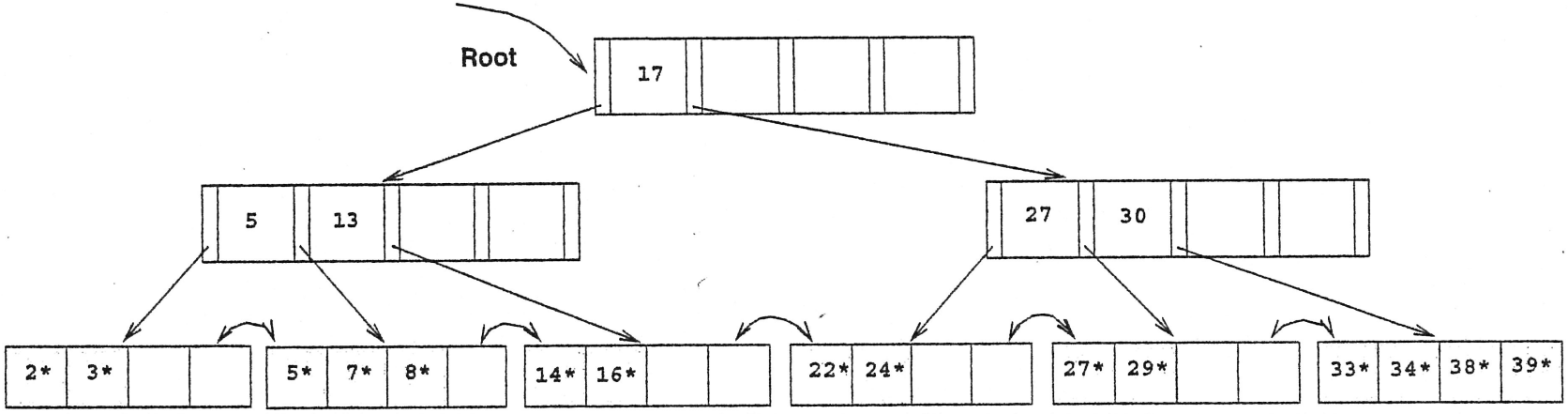
suppose we next delete 24*

B+ trees: deletion

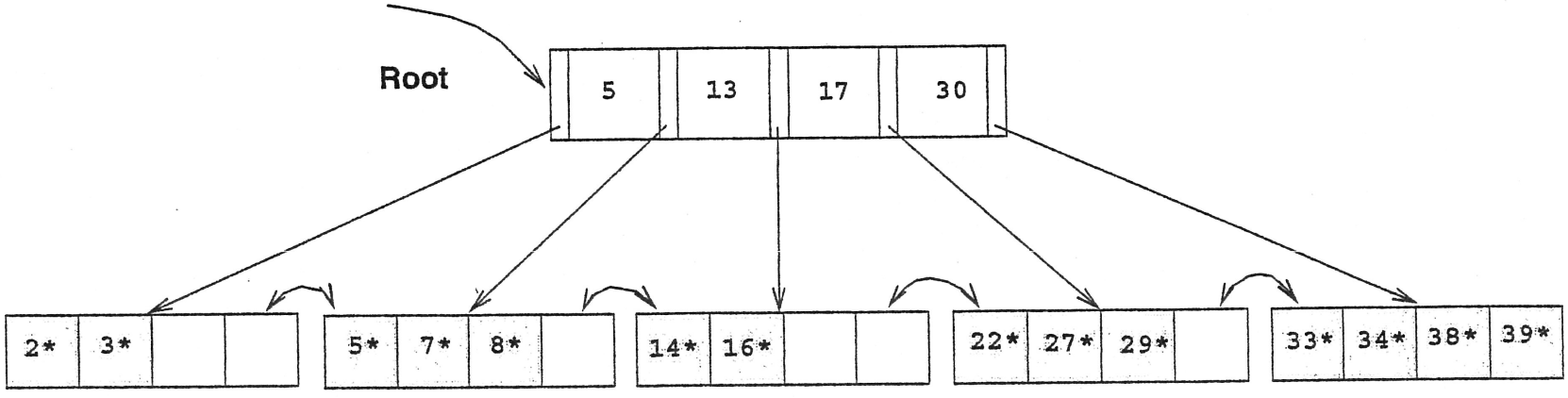


During deletion of 24*

B+ trees: deletion

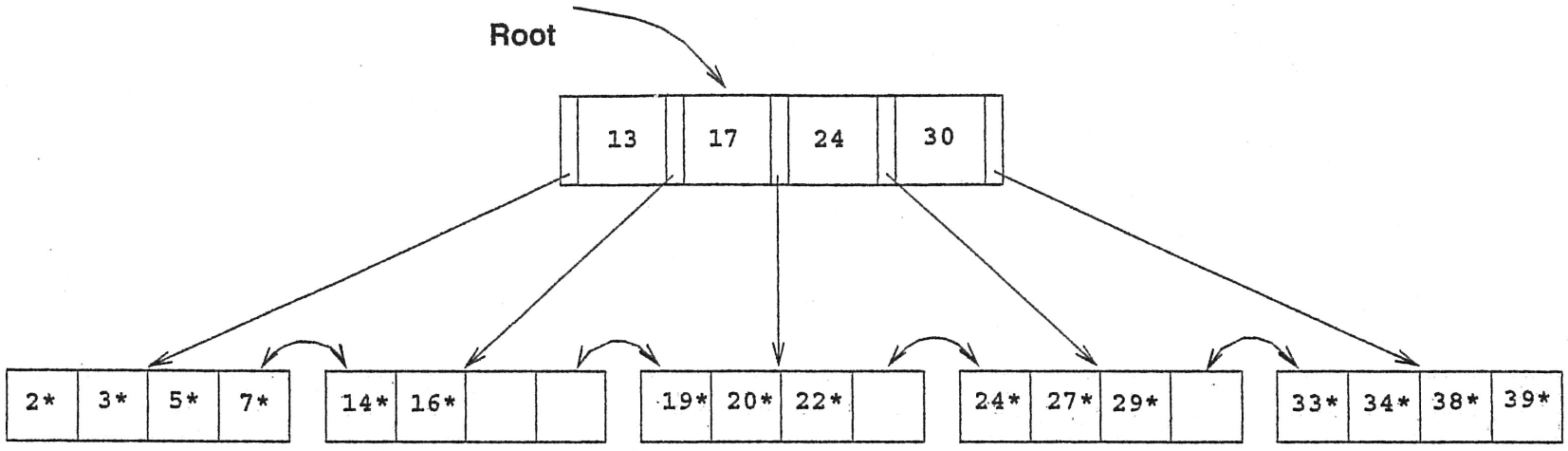


before ...



and after deletion of 24*

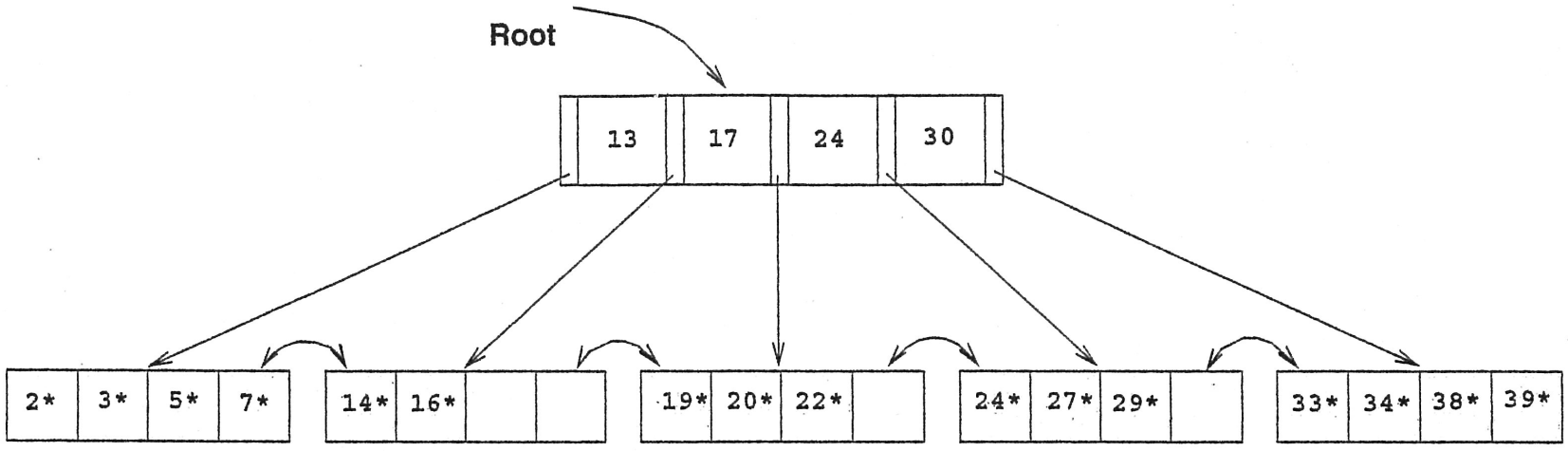
B+ trees



order $n = 5$

Exercise: Show the full tree that would result from inserting an item with key 6, with sibling redistribution.

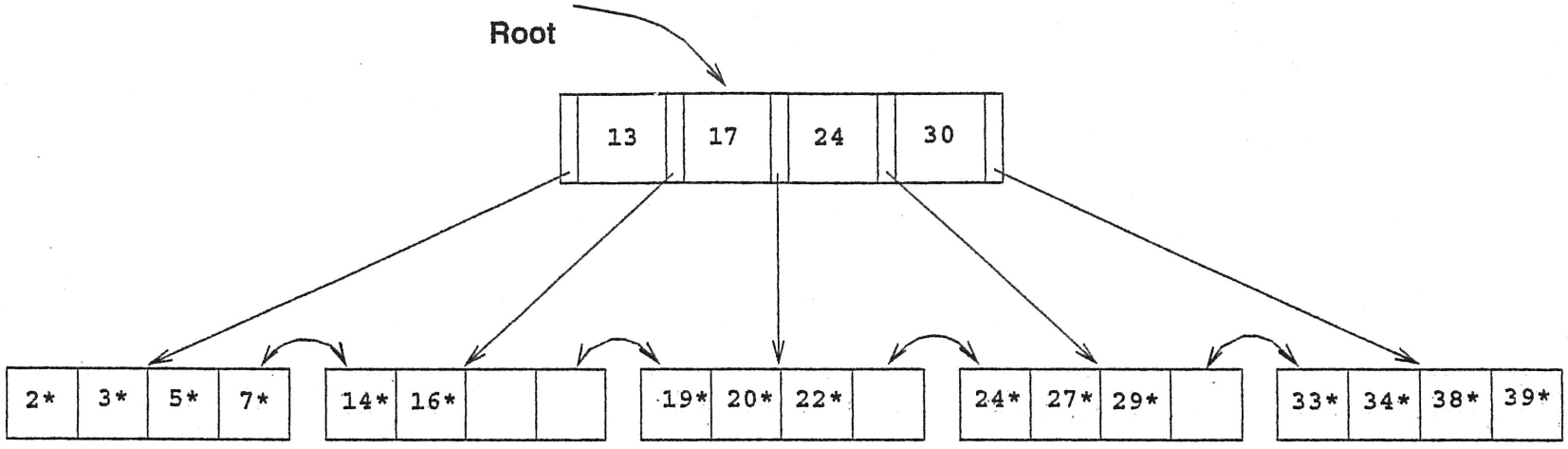
B+ trees



order $n = 5$

Exercise: Show the full tree that would result from inserting an item with key 6, without sibling redistribution.

B+ trees



order $n = 5$

Exercise: Show the full tree that would result from deleting item with key 14.

Recap

- Topic 1: Storage on disk & I/O model of computing
- Topic 2: External sorting
- Topic 3: Indexing, part 1 – ordered indexes

Recap

- Topic 1: Storage on disk & I/O model of computing
- Topic 2: External sorting
- Topic 3: Indexing, part 1 – ordered indexes

Next Wednesday: Ordered indexes, cont.

Next Friday: hash-based index data structures.

Please submit your paper selections now, if you haven't already done so.

Figure credits

- Our textbook (Silberschatz *et al.*, 2011)
- Ramakrishnan & Gehrke, 2003
- Garcia-Molina *et al.*, 2002