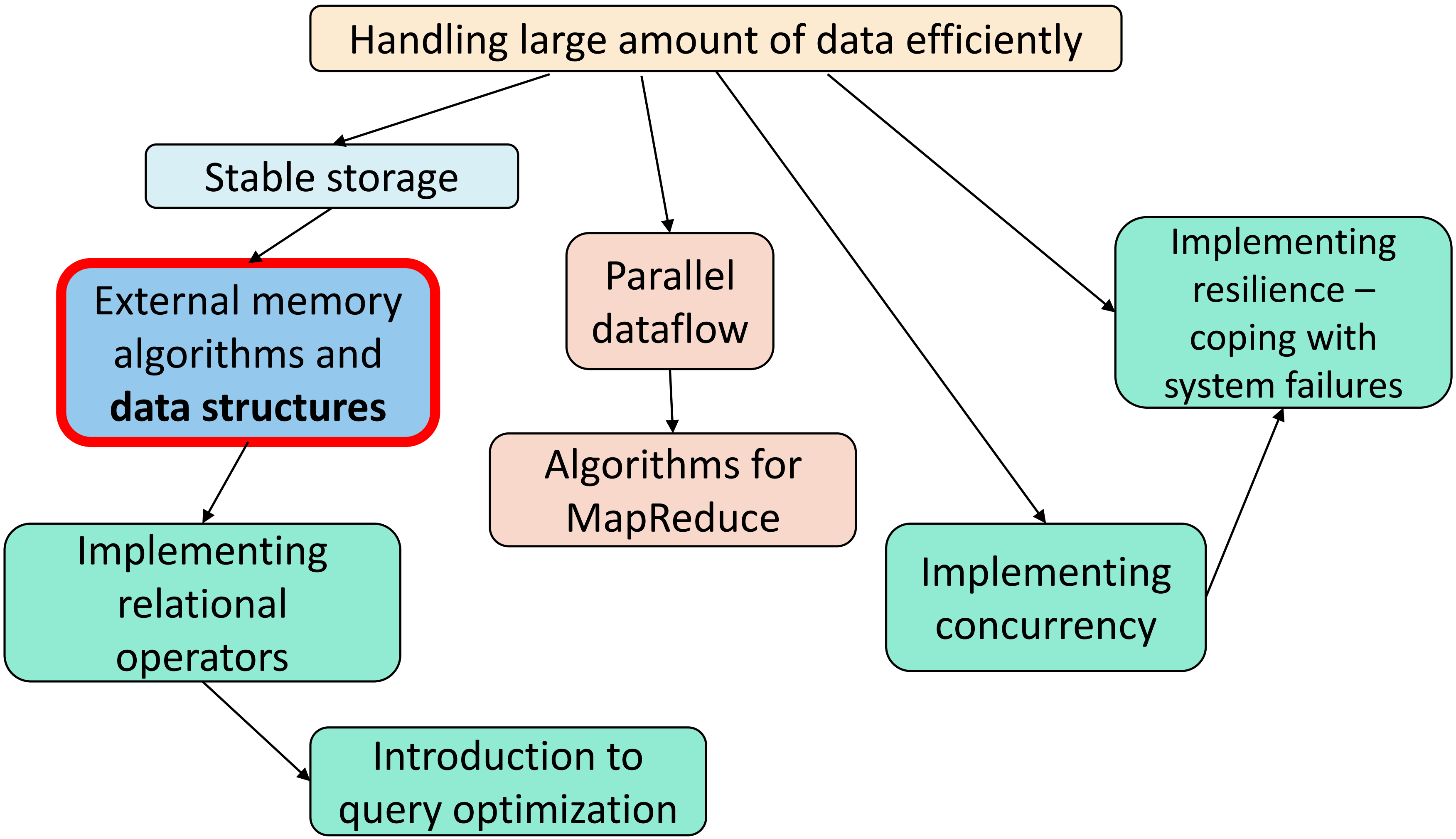
Roadmap



Handling large amount of data efficiently
Stable storage
External memory algorithms and **data structures**
Parallel dataflow
Algorithms for MapReduce
Implementing resilience – coping with system failures
Implementing relational operators
Implementing concurrency
Introduction to query optimization

Lecture 02.04

Disk data structures: Static indexes

By Marina Barsky
Winter 2017, University of Toronto

Combining pages (blocks) into files

Which block of a file should a record go to ?

I. Anywhere ? *"Heap"* organization
   - How to search for "SID= 123" ?

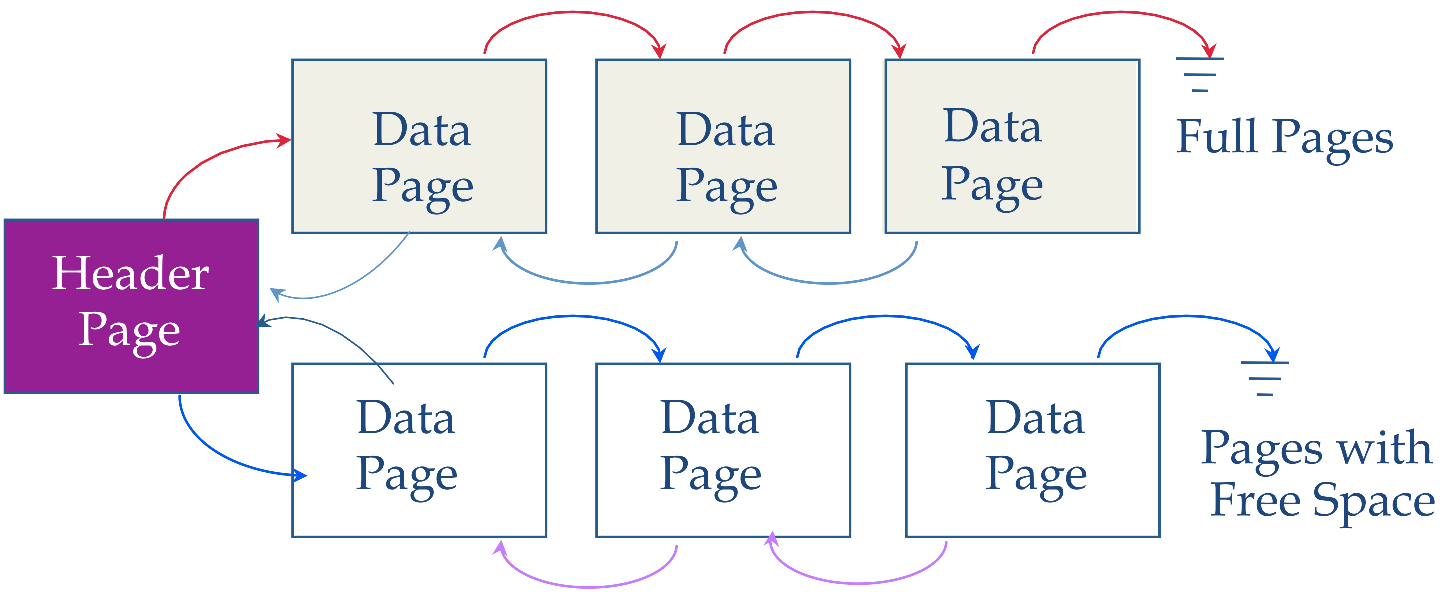
II. Sorted by some key ? *"Sequential"* organization
   - Keeping it sorted could be painful

III. Based on a "hash" key? *"Hashing"* organization
   - Store the record with SID = x in the block number h(x)%1000

I. Heap files

- *Heap files* – unordered set of disk blocks – simplest file structure. Contains blocks in no particular order
- As file grows and shrinks, disk blocks are allocated and de-allocated
- We must:
  - Keep track of all the *pages* in a file
  - Keep track of pages with *free space* on them

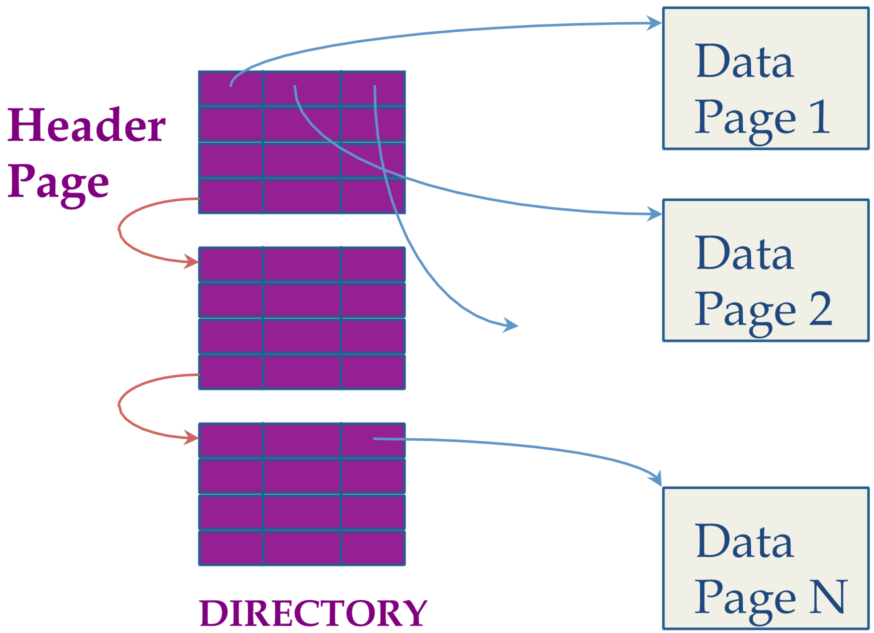
Example: Heap implemented as doubly-linked list



- The header page id and Heap file name must be stored someplace.
- The header page contains 2 ‘pointers’ – block IDs
- Each page contains 2 `pointers’ in addition to page header and data

Example: Heap with page directory

- The entry for a page can include the number of free bytes on the page.
- The directory is a collection of pages by itself: linked list implementation is just one alternative.
  - *Much smaller than linked list of all data pages*!



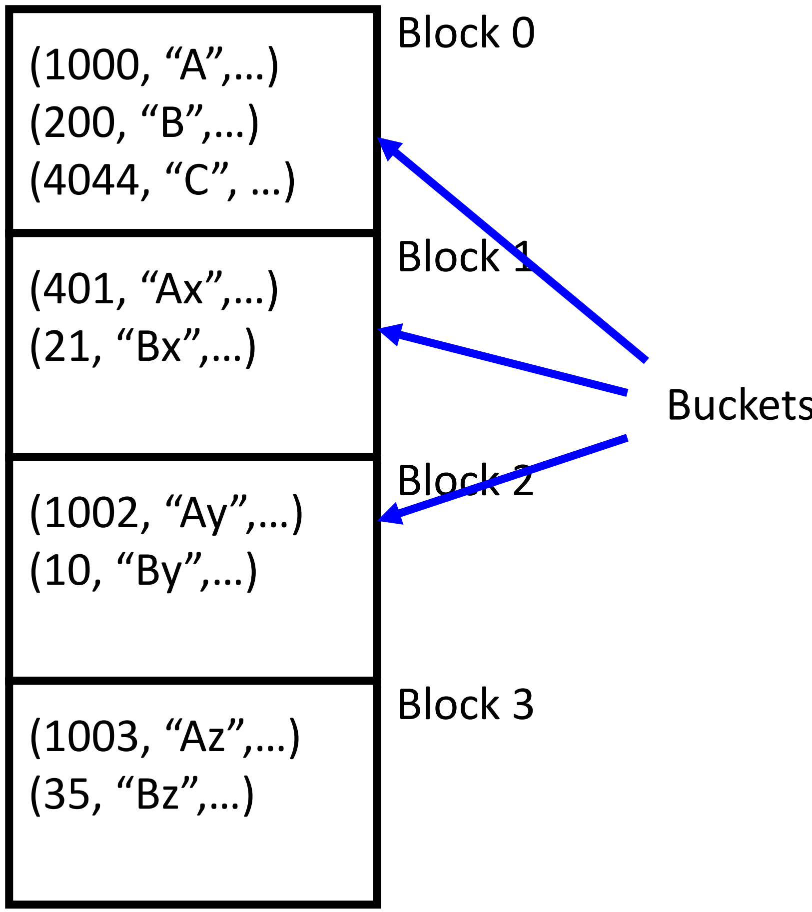
II. Sequential (sorted) file organization

- Keep pages sorted by some key in the records
- Insertion
  - Find the block in which the tuple should be
  - If there is free space, insert it
  - Otherwise, create an overflow page, and link from the corresponding page Can create a long list of overflow pages
- Deletions
  - Delete and keep the record of a free space
  - Databases tend to be insert-heavy, so free space gets used fast

  Can become *fragmented:* must reorganize once in a while

III. Hash-based file organization

- Allocate file with 4 pages
- Store record with search key *k* in block number *h(k)* % 4, where h(k) is a hash function
- *Blocks* are called "*buckets*"
- What if the bucket becomes full ?
  Overflow pages. As file grows, the search becomes inefficient



File manager component of DBMS

- The file manager component takes care of file manipulations
- It interacts directly with the disk blocks, bypassing operating system
- It is a complex piece of software whose detailed implementation is outside the scope of this course

Comparing efficiency of file organizations

- Operations to compare:
  - Scan: fetch all records from disk
  - Equality search: find record with key = *k*
  - Range selection: find all records where key is between *a* and *b*
  - Insert a record
  - Delete a record

Cost Model for Our Analysis

N – total number of data pages (file blocks)

- We calculate the average number of disk I/Os per operation
- We **ignore** CPU costs in our model

Note 1: Measuring number of page I/O's ignores differences between random and sequential I/Os

Note 2: Average-case analysis; based on several simplistic assumptions

Good enough to show the overall trends!

Assumptions

- Sorted Files:
  Files compacted after deletions, no overflow pages
- Heap Files:
  Equality selection on key - exactly one match
- Hash:
  No overflow buckets

Cost of operations for different file organizations (in disk I/Os)

N - number of data pages

	Scan	Equality	Range	Insert	Delete
(1) Heap					
(2) Sorted					
(3) Hashed					

Cost of operations for different file organization (in disk I/Os)

N - number of data pages

	Scan	Equality	Range	Insert	Delete
(1) Heap	N	0.5N	N	2	0.5N +1
(2) Sorted	N	$\log_2 N$	$\log_2 N$ + output	$\log_2 N + N$	$\log_2 N + N$
(3) Hashed	N	1	N	2	2

* *Several assumptions underlie these (rough) estimates!*

Cost of operations for different file organization (in disk I/Os)

N - number of data pages

	Scan	Equality	Range	Insert	Delete
(1) Heap	N	0.5N	N	2	0.5N +1
(2) Sorted	N	$\log_2 N$	$\log_2 N$ + output	$\log_2 N + N$	$\log_2 N + N$
(3) Hashed	N	1	N	2	2

Why 2?
Always insert at the end of the file:
1 I/O to read, 1 to write back

Cost of operations for different file organization (in disk I/Os)

N - number of data pages

	Scan	Equality	Range	Insert	Delete
(1) Heap	N	0.5N	N	2	0.5N +1
(2) Sorted	N	$\log_2 N$	$\log_2 N$ + output	$\log_2 N + N$	$\log_2 N + N$
(3) Hashed	N	1	N	2	2

Why +1?
Find the page (average equality search), mark record as deleted and write back

Cost of operations for different file organization (in disk I/Os)

N - number of data pages

	Scan	Equality	Range	Insert	Delete
(1) Heap	N	0.5N	N	2	0.5N +1
(2) Sorted	N	$\log_2 N$	$\log_2 N$ + output	$\log_2 N + N$	$\log_2 N + N$
(3) Hashed	N	1	N	2	2

Why +N?
Find the page, insert record, shift all the pages, assuming there are no empty slots

Not very efficient: example - search

- Find an *Account info for SIN = 123*
- Sequential file: *log(N)* disk accesses - Random accesses!
  - For N = 1,000,000,000   log(N) = 30
  - Each random access ≈10 ms
  - 300 ms to find just one account information!
  - < 4 requests satisfied per second

Conclusion

- Heap, sequential, and hash-based file organizations are not very efficient in most cases
- We need more sophisticated data structures

Introducing Indexes

- *Index* - a data structure for efficient search through large databases (Think - library index/catalogue)
- Goal: quickly locate the record given a key
- Two key ideas:
  - The records are mapped to the disk blocks in specific ways
  - Auxiliary data structures allow quick search

Key ideas

- Idea 1:
  - The records are mapped to the disk blocks in specific ways: we deduce the disk location from a key, because record is in the block which is a hash of a key
- Idea 2:
  - Store records in a pile (heap or sorted)
  - Provide auxiliary data structures guiding the search, which are significantly smaller than the data itself

Flat indexes

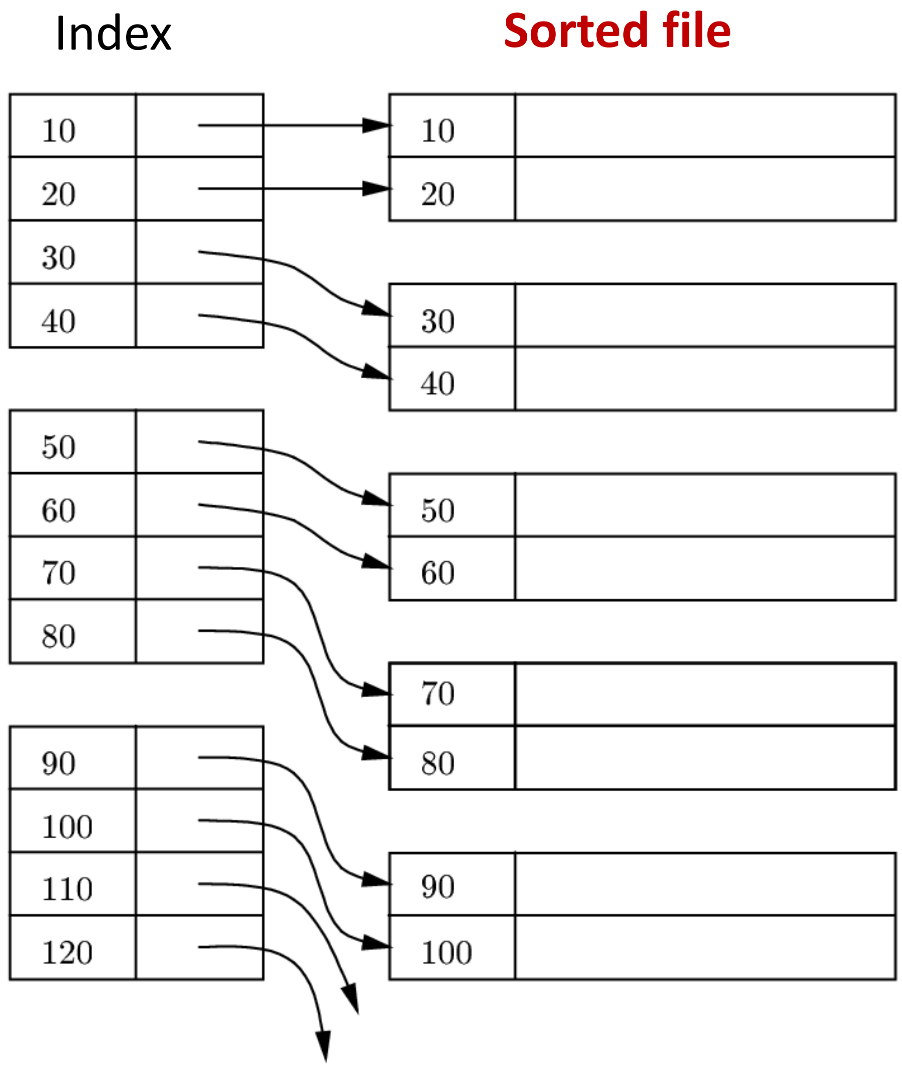
- Have a catalog of search keys which is smaller than the entire table and can be searched more efficiently (in RAM or with less disk I/Os)

- Inside the index each value of a key is associated with a unique, system-generated physical address of a corresponding tuple on disk: ***RID*** (file number, block number, slot within the data block)

Dense indexes

- ***Dense index*** – each record has its representative inside index
- If the table has multiple fields, the index stores only key-RID pair and is much smaller – may fit into RAM
- The keys in the index are sorted: use binary search, buffer guiding pointers at 1/2N, 1/4N, 3/4N, 1/8N, 3/8N, 5/8N, 7/8N –th positions to save disk I/Os

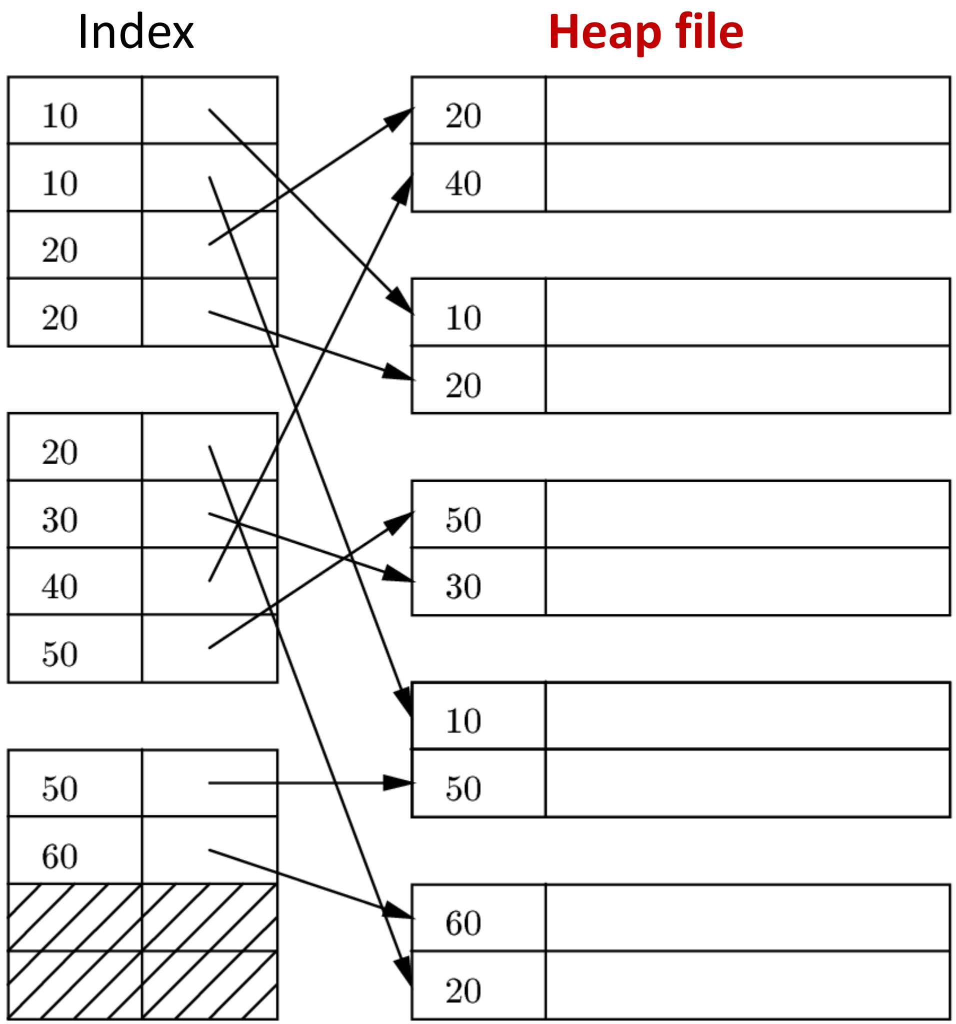
Example:
dense index for sorted file



Additional structure on top

Data records

Example: dense index for heap file



Can answer if the record exists even without accessing it on disk

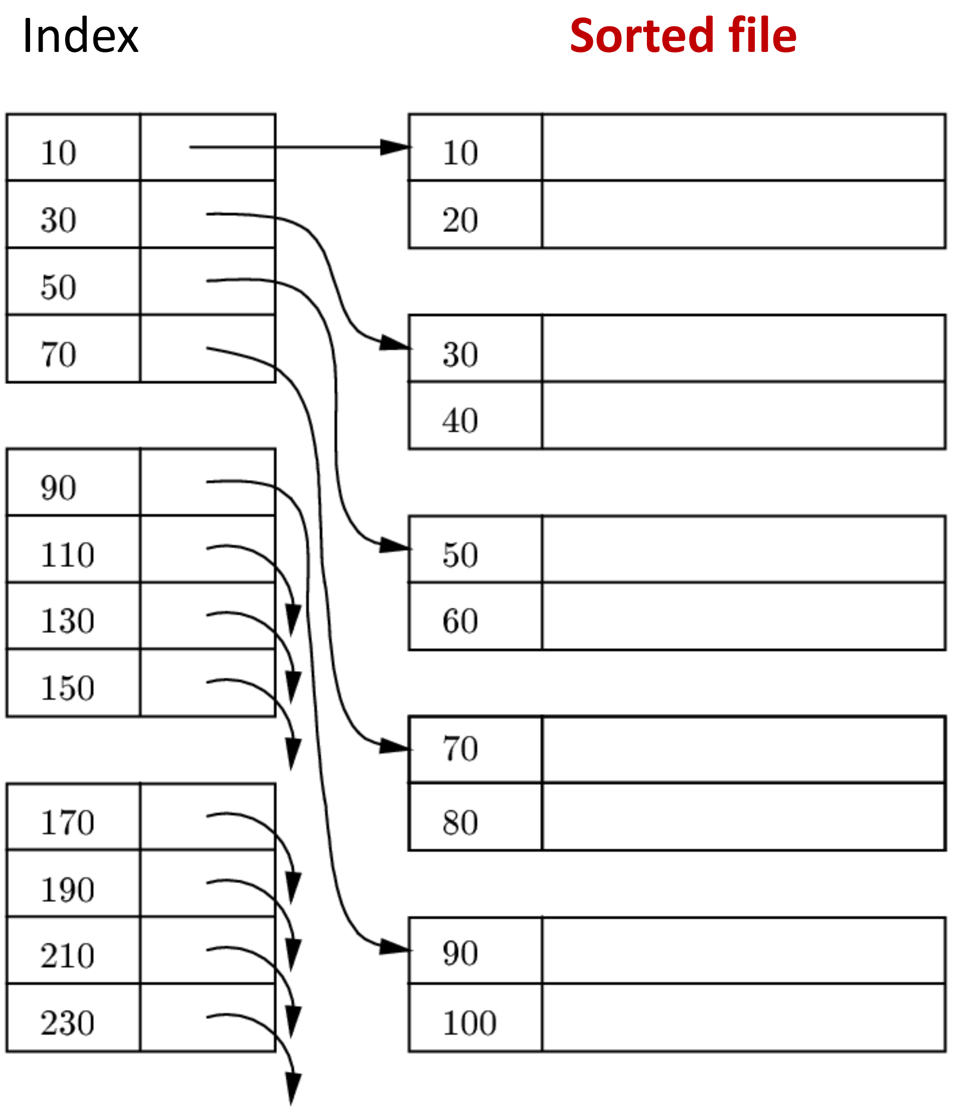
Sparse indexes

- ***Sparse index*** – contains key-RID pairs for only a subset of records, typically first in each block.
- Works only with sequential (sorted) files

Why?

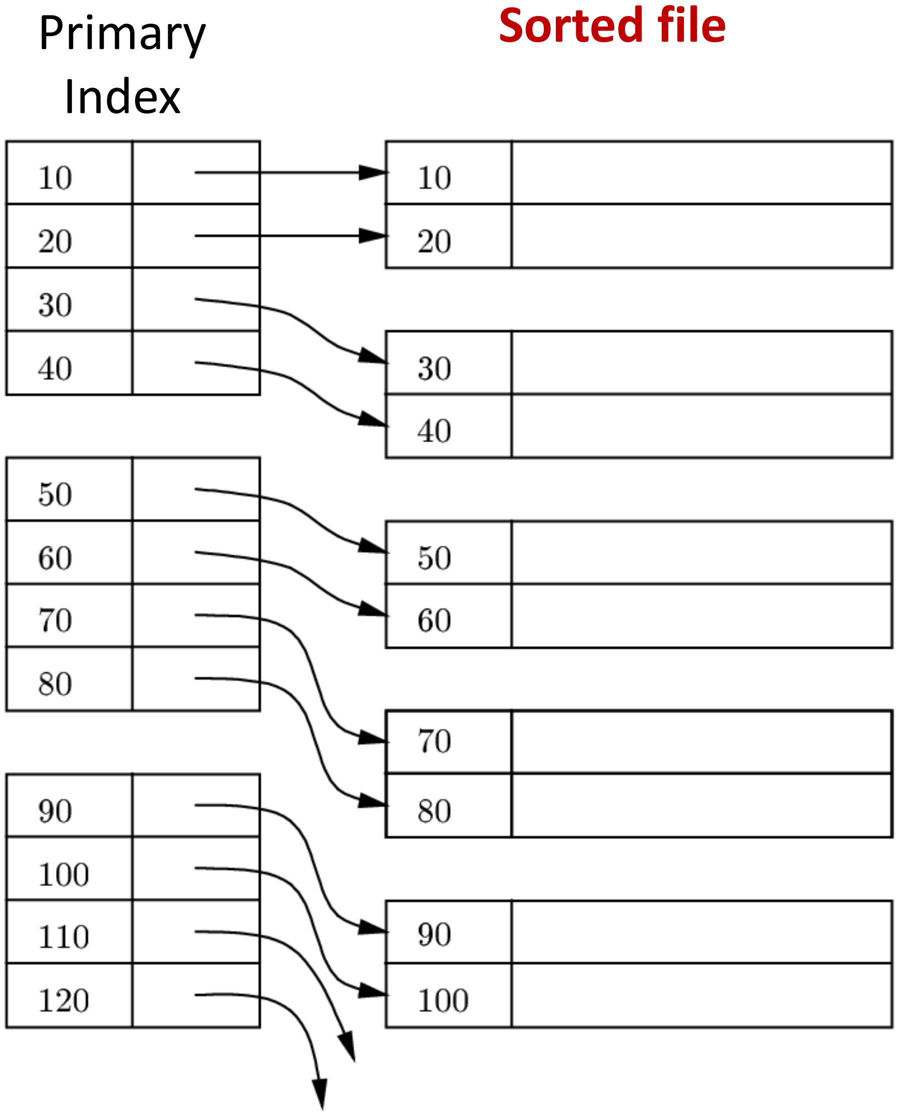
- Allows for very small indexes - better chance of fitting in memory
- Tradeoff: *must* access the relation file even if the record is not present

Example: sparse index



Primary indexes

- ***Primary index*** – index on a sorted file for the sorting attribute
- Only **one primary index per relation** – otherwise needs to maintain several sorted copies of the same data

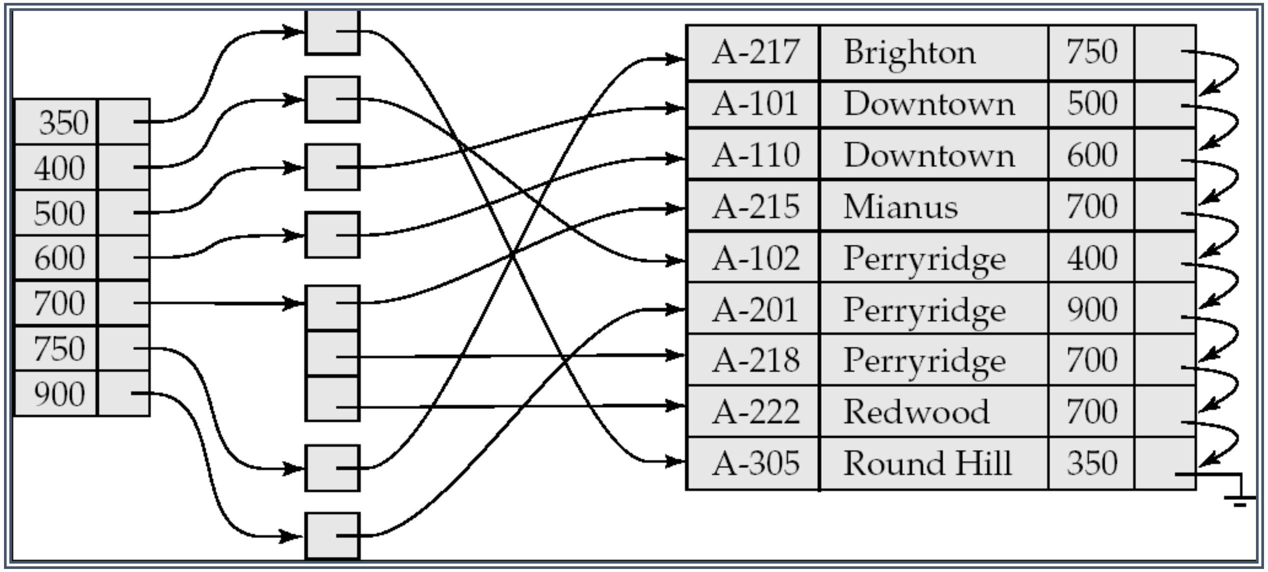


Secondary indexes

- ***Secondary index*** – index on any other attribute, does not "control placement."

Example:

- Relation sorted on *branch*
- But we want an index on *balance*



- Secondary index **must be dense**

Why?

What if a flat index is too big?

Example:

- Relation of size: N = 500 GB = $5*10^{11}$ bytes
- 100 tuples per block: $5*10^{9}$ blocks to index
- Each key-RID pair is at least 16 bytes
- So, even keeping one entry per page (sparse index) takes too much space - 8 GB

Solution: build an index on the index itself!

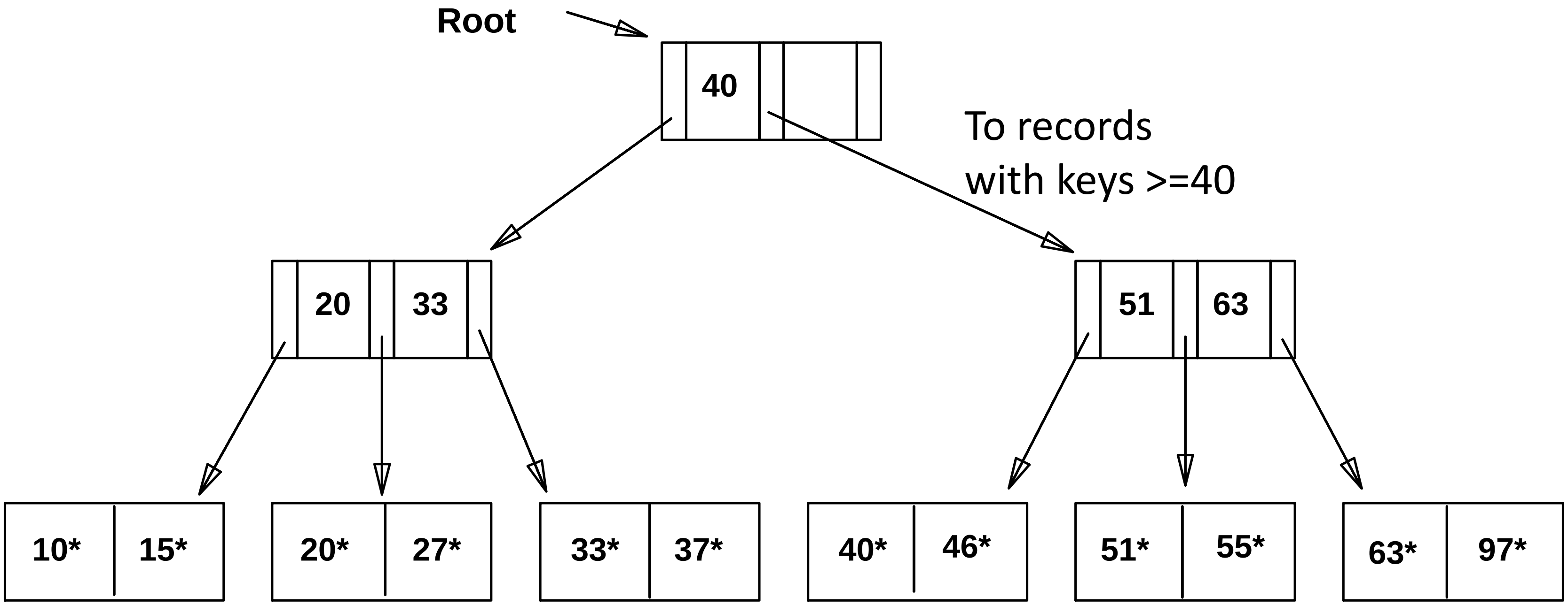
Multi-level indexes - static trees

- If distribution of keys is not very skewed and we know the range of keys in advance, we can allocate data pages to keep records sorted, and build on top a tree of search-guiding dividers

- This tree is *static*, and is never modified

Example: static trees

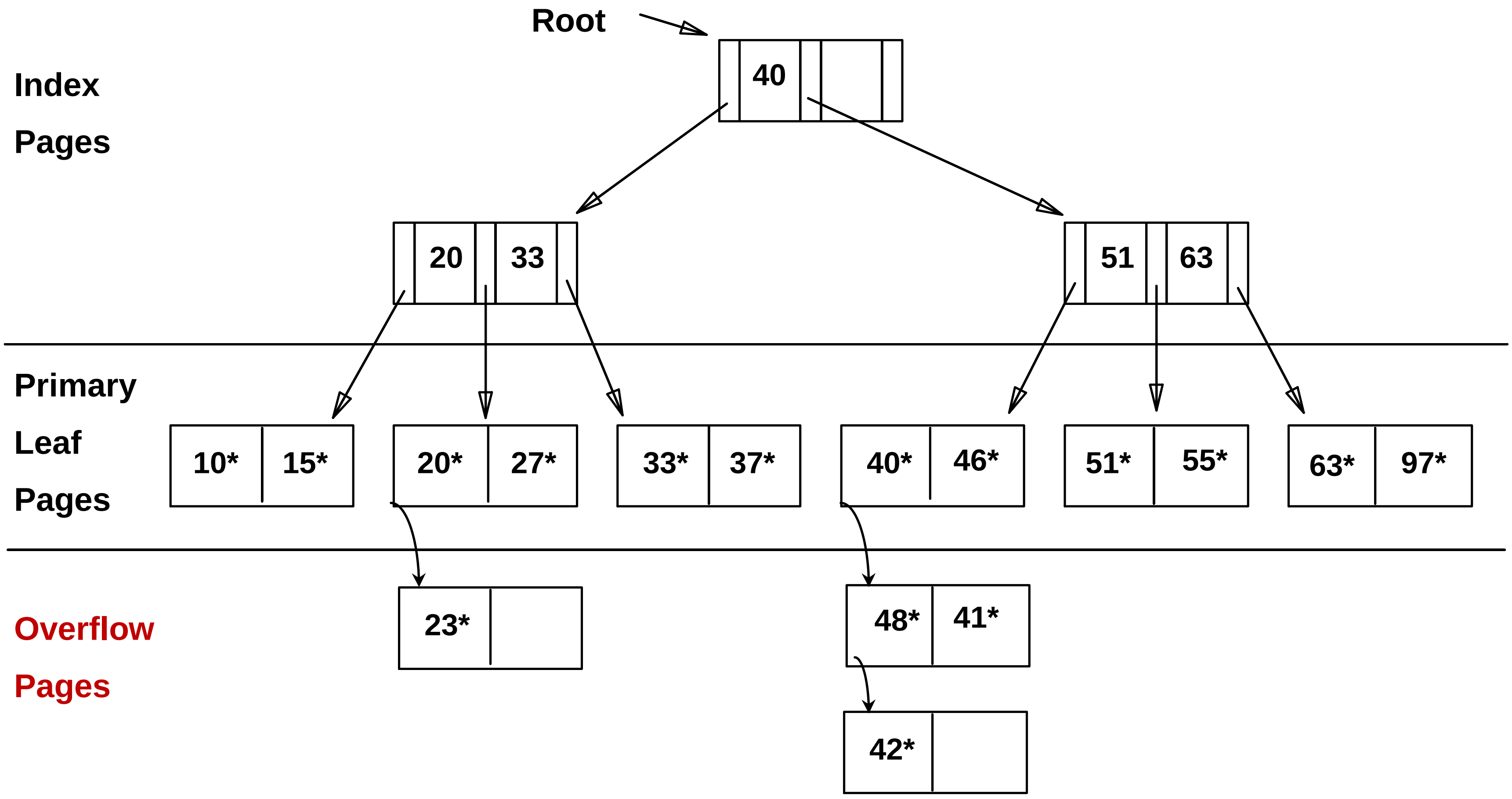
2-level search-guiding ternary tree



Data pages (sorted)

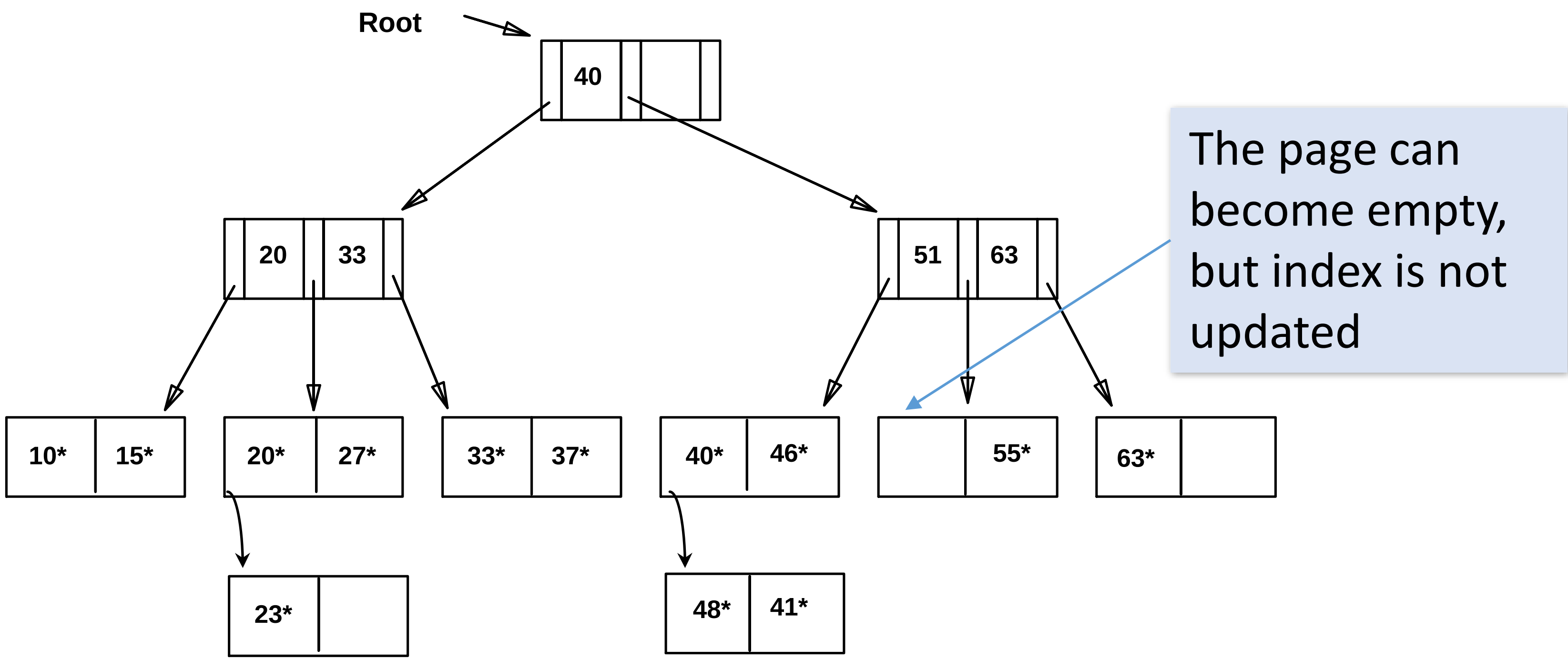
Static tree: insertion

- Inserting 23*, 48*, 41*, 42*



Static trees: deletion

- Deleting 42*, 51*, 97*



Note that 51 appears in index levels, but not in leaf!*

Static trees: pro and contra

- To build ISAM index, the sorted data is distributed among pages, leaving each data page half-full to accommodate future insertions
- The index tree is built on top of the sorted file, and **is never modified**

Good:

- Updates affect only the level of data pages – no need to worry of modifying by multiple users – no locking of the index pages

Bad:

- If data file grows (especially with skewed keys) the number of overflow pages becomes too big for efficient search
- Skewed deletions may leave a lot of unused empty space, which never gets filled with new records

Dynamic indexes

We need a dynamic data structure, which will guarantee an efficient search in any case and will accommodate database modifications

2 main indexing data structures:

- Dynamic Trees
- Dynamic Hashes