Tree-Structured Indexes
BTrees -- ISAM

Chapter 10 – Ramakrishnan & Gehrke
(Sections 10.1-10.2)

What will I learn from this lecture?

- Basics of indexes
- What are data entries?
- Basics of tree-structured indexing
  - ISAM (Indexed Sequential Access Method).
  - and its limitations.
- ISAM – precursor for B-trees.

Motivation

- On the one hand, user wants to keep as much data as possible, e.g., set of songs and reviews, 100 million records, …
- On the other hand, user also wants *performance* e.g., searching 100 million records in real time
- Key: *indexing*
- Two main categories to be considered
  - Tree-structured
  - Hashing

Basics – Data Entries

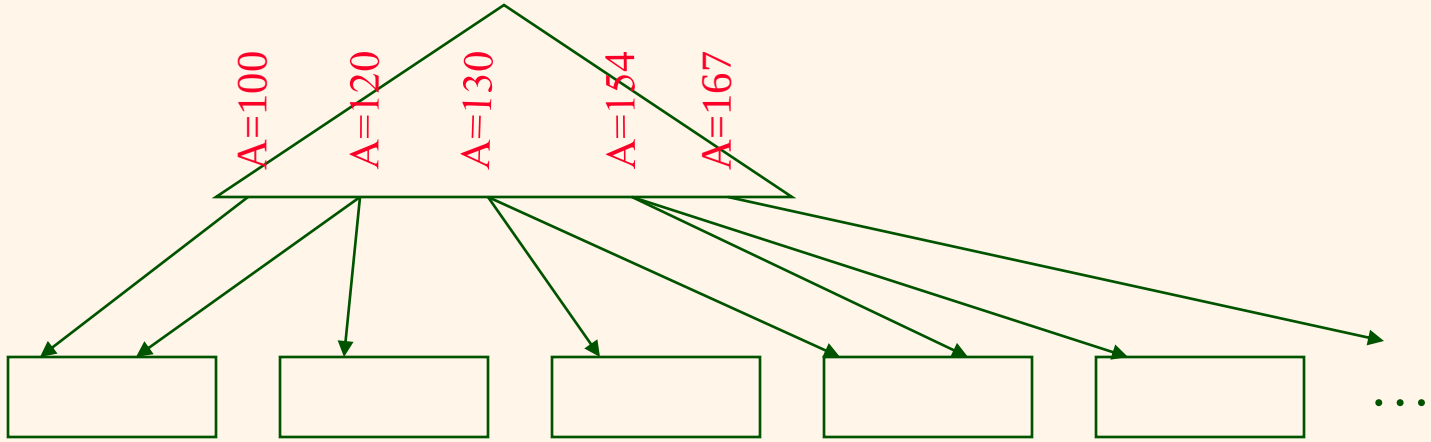
- *Example table: reviews*(revid:**integer**, sname:**string**, rating:**integer**, time:**time**)
- *Field op value ----Index--→ data entries.*
- *For any index, there are 3 alternatives for data entries* **k***:
  - Data record with key value **k**
  - <**k**, rid of data record with search key value **k**> e.g., <*revid, sname*> (typically, primary key)
  - <**k**, list of rids of data records with search key **k**> e.g., *rating* or time (typically secondary key)
- Choice is orthogonal to the *indexing technique* used to locate data entries **k***.
- When would you choose which option?

Basics – Clustered Index

- What about the physical address of the records?
- Strictly speaking, pairs <rid, addr> but omit addr for simplicity
- If the data records are *physically sorted* on indexed attr *A*, we say *A* has a *clustered* index
- Otherwise, we call the index *non-clustered (or unclustered)*
  - Implication: records not contiguously stored, may need one disk access (i.e., random I/O) per record in the worst case
  - Remember, random I/O is much more expensive than sequential I/O.

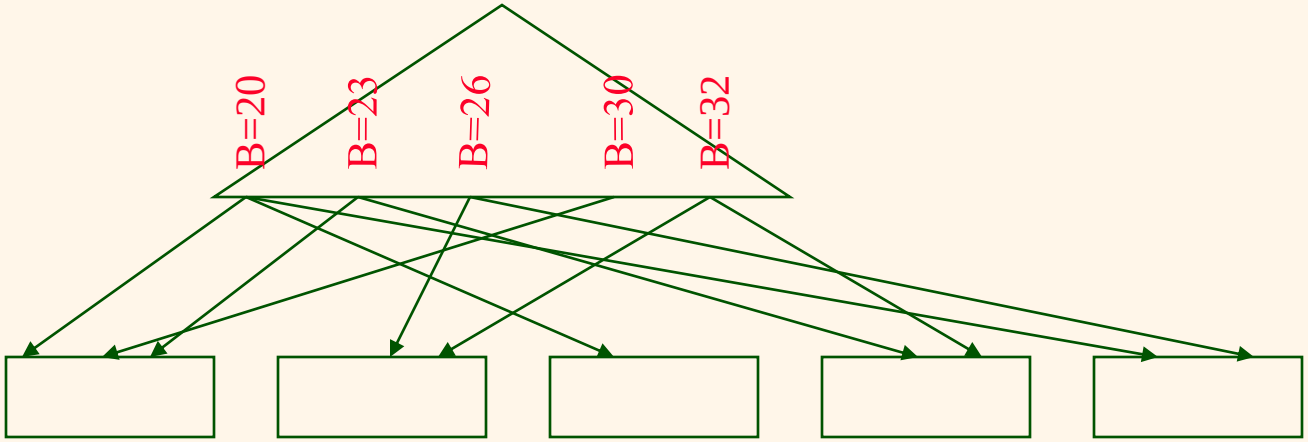
A schematic view of a clustered index

Consider a file of records with fields A, B, …, sorted on A.
Here is a clustered index on A.



A schematic view of an unclustered index

If file is sorted on A, it is not sorted on B, in general.



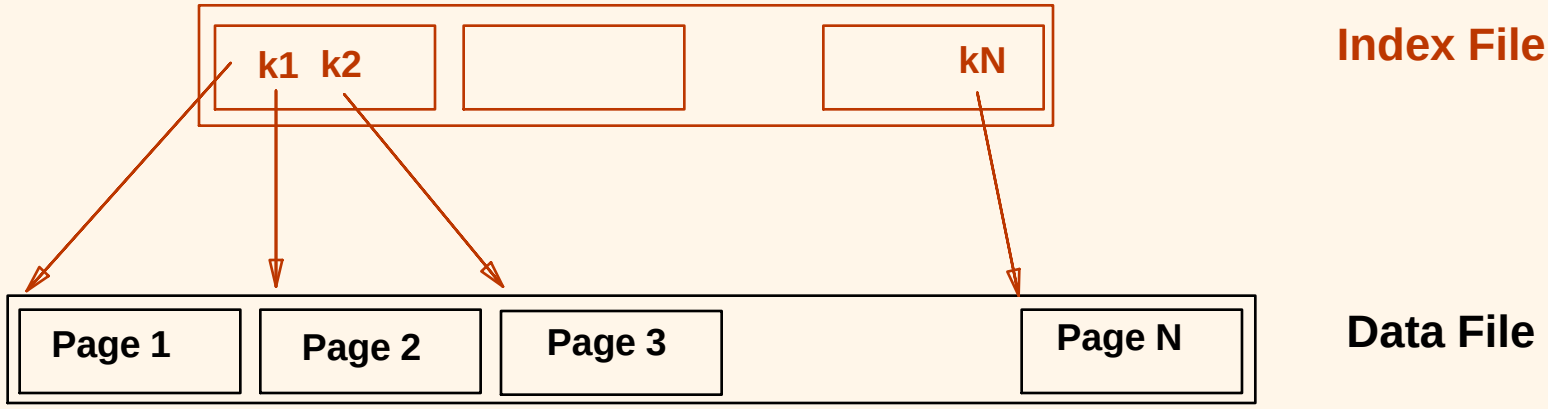
These schematics are for illustrative purposes only. Actual structures may vary depending on details of implementation.

Basics – Tree-structured Indexing

- Tree-structured indexing techniques support both *range searches* and *equality searches*
  - range: e.g., find all songs with rating >= 8
  - equality:
    - ordered domains: degenerate case of a range
    - unordered domains: e.g., sname="International love"
  - But, for unordered domains, hierarchies may induce natural ranges: e.g., song_genre ="hiphop"; B-trees don't handle that!
- *ISAM*: static structure; *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.

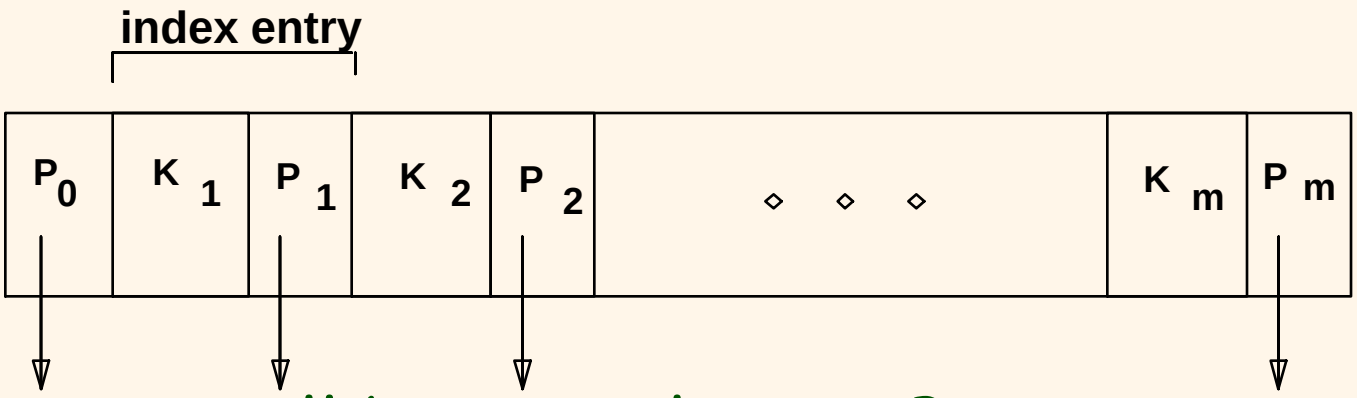
Range Searches

- ``*Find all songs with at least one rating >=8'*
  - If data is sorted on rating, do binary search to find first such song, then scan to find others.
  - Cost of binary search can be quite high. (Why?)
- Simple idea: Create an `index' file.

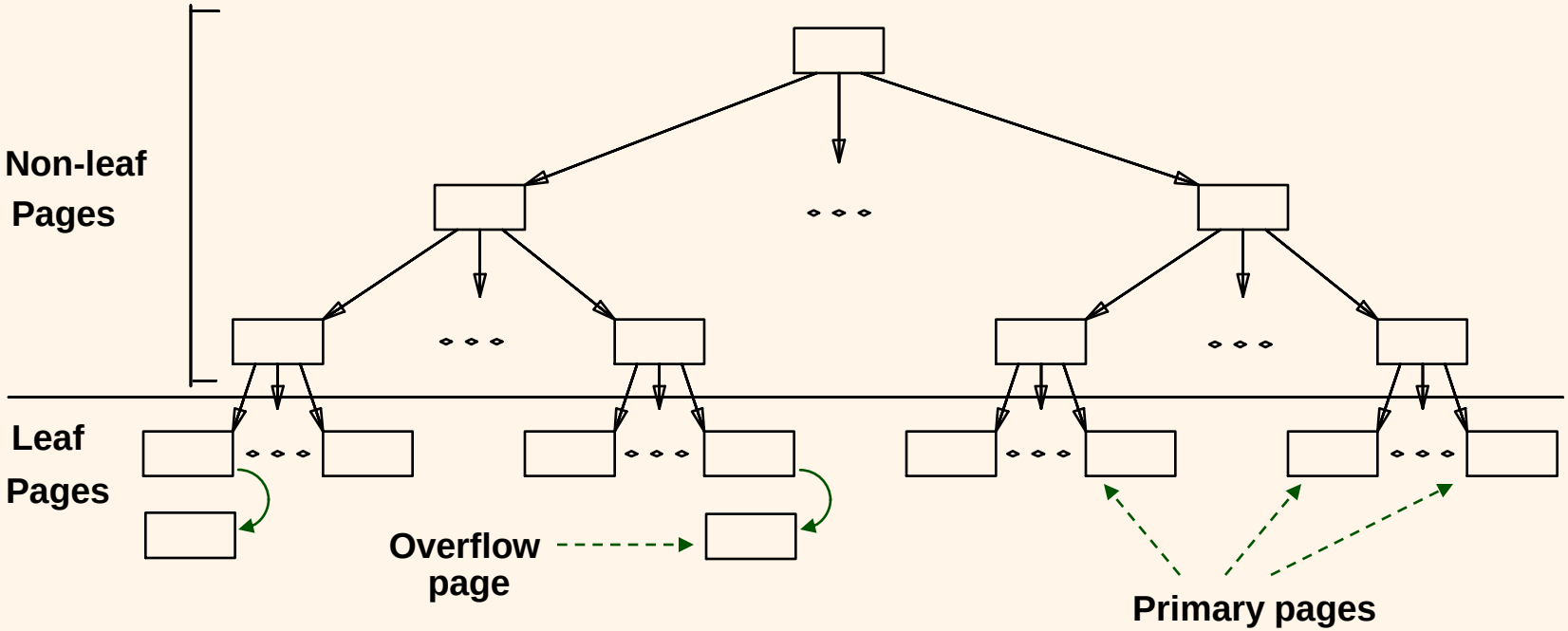


✉ *Can do binary search on (smaller) index file!*

ISAM

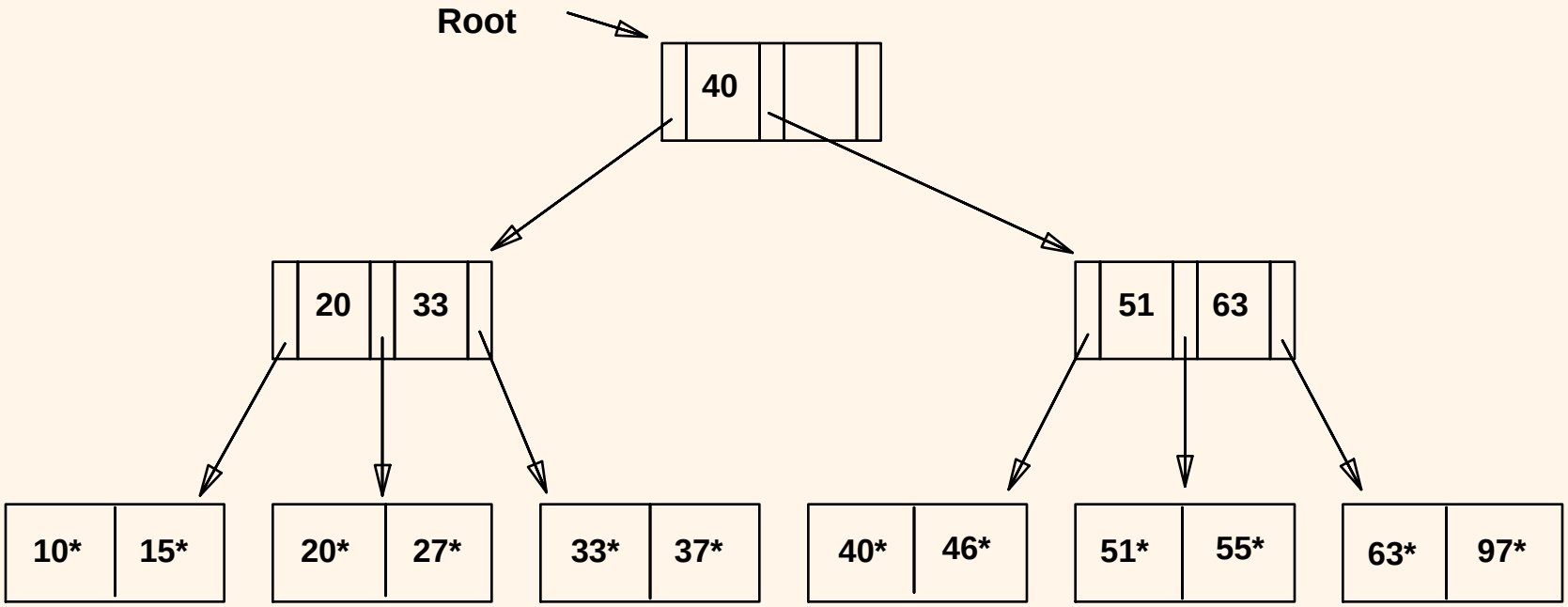


- Index file may still be quite large. But we can apply the idea repeatedly!

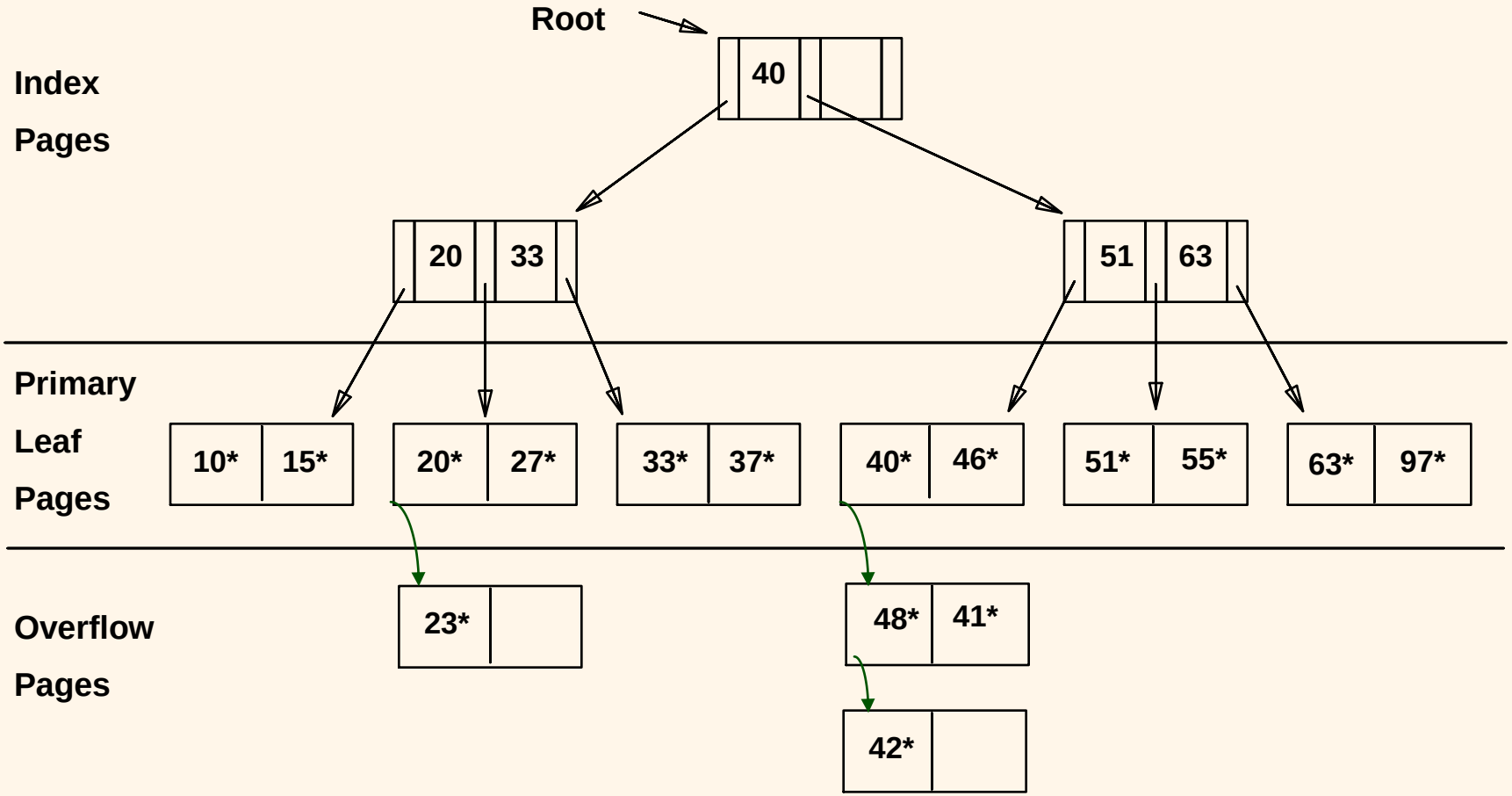


✉ *Leaf pages contain data entries.*

Example ISAM Tree

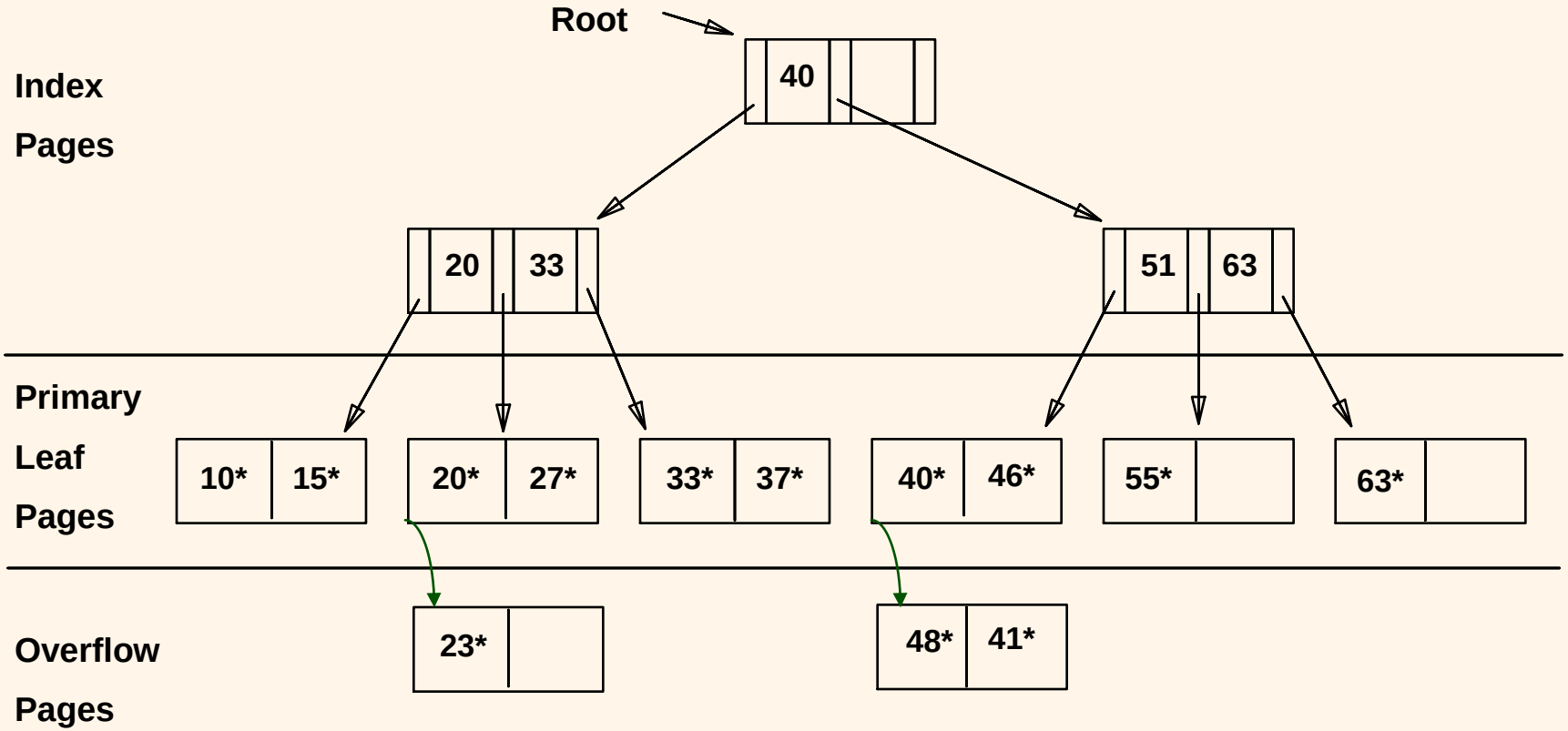


After Inserting 23*, 48*, 41*, 42* ...



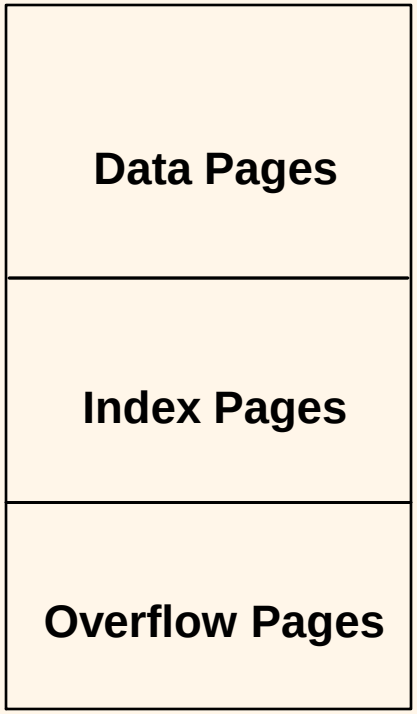
Suppose we now delete 42*, 51*, 97*.

...Then Deleting 42*, 51*, 97*



note that 51 still appears in the index page!

Comments on ISAM



- *File creation*: Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages.
- *Index entries*: <search key value, page id>; they `direct' search for *data entries*, which are in leaf pages.
- *Search*: Start at root; use key comparisons to go to leaf. Cost $\propto \log_F N$; F = # pointers/index pg, N = # leaf pgs
- *Insert*: Find leaf that data entry belongs to, and put it there, which may be in the primary or overflow area.
- *Delete*: Find and remove from leaf; if empty overflow page, de-allocate.

✉ **Static tree structure**: *inserts/deletes affect only leaf pages.*

Evaluation of ISAM

- is a static indexing structure
- works well for certain applications
  - few updates, e.g., dictionary
- frequent updates may cause the structure to degrade
  - index pages never change
  - some range of values may have too many overflow pages
  - e.g., inserting many values between 40 and 51.

Lead up to B-Trees

- ISAM done right!
- Improve upon ISAM idea using lessons learned.
- Don't leave index structure static.
- Adapt it to changes (i.e., updates to underlying data).
- ISAM – almost certainly won't be used for any current apps.