CS222/CS122C: Principles of Data Management

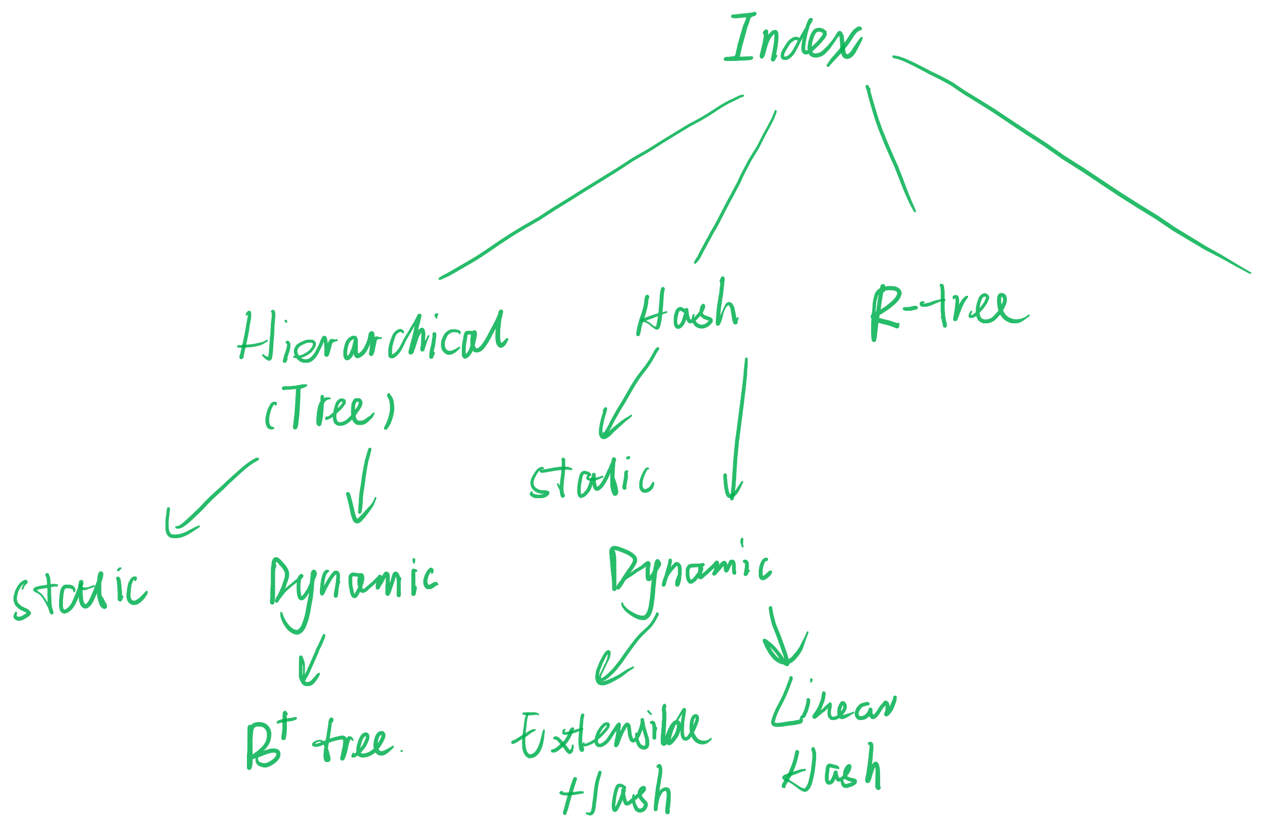
UCI, Fall 2019
Notes #05

Index Overview and ISAM Tree Index

Instructor: Chen Li

Index families

- Tree-based indexes
  - ISAM: static structure (Indexed Sequential Access Method)
  - B+ tree: dynamic, adjusts gracefully under inserts and deletes.
- Hash tables
  - Static hashing
  - Dynamic hashing



Index
Hierarchical (Tree)
Hash
R-tree
static
Dynamic
B+ tree
static
Dynamic
Extensible Hash
Linear Hash

Indexes

- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Any subset of the fields of a relation can serve as the search key for an index on the relation.
  - *Search key* is **not** the same as a *key* (a minimal set of fields that uniquely identify a record in a relation).

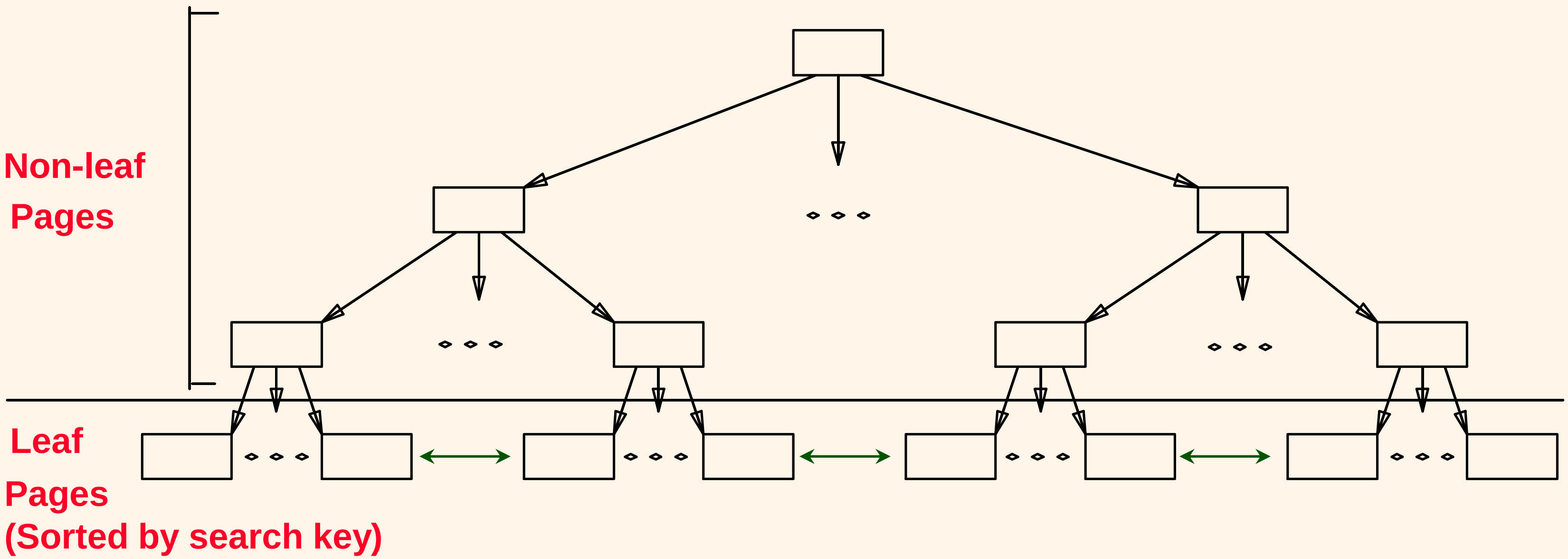
Allow same Search key

~~Allow duplicate~~

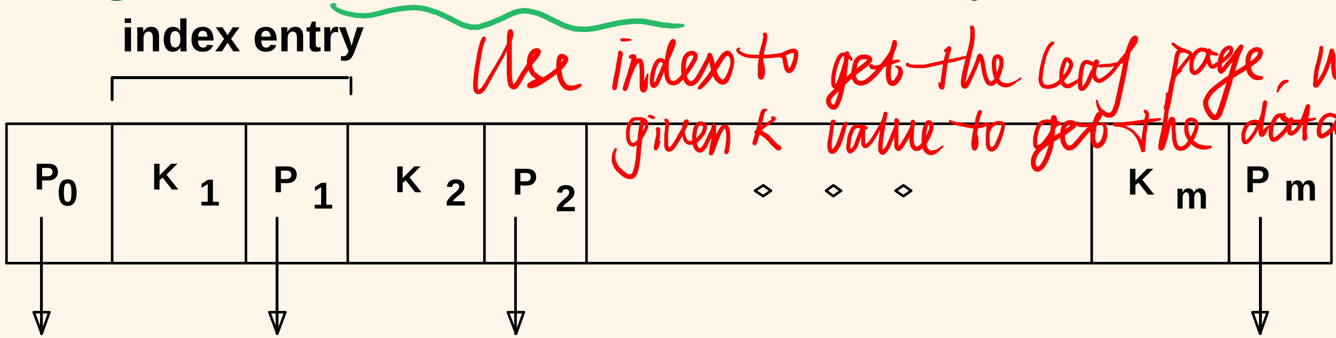
Data Entries

- An index contains a collection of *data entries*, and supports efficient retrieval of all data entries **k*** with a given key value **k**.
  - Given data entry k*, we can find one record with key k with just more disk I/O.

E.g., Tree index



- Leaf pages contain *data entries*, and are chained (prev & next)
- Non-leaf pages have *index entries;* used only to direct searches:



Use index to get the leaf page, use the given k value to get the data entry

Alternatives for Data Entry ***k**** *in Index*

clustered tree

- In a data entry k* we can store:
  - Actual data record with key value **k**, or
  - <**k**, rid of data record with search key value **k**>, or
  - <**k**, list of rids of data records with search key **k**>
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value **k**.
  - Examples of indexing techniques: B+ trees, hash-based structures, R trees, …
  - Index's job: direct searches to desired data entries
  - Rid alternative in secondary indexes: primary key

Each Rid is unique → primary key

Alternatives for Data Entries (Contd.)

- Alternative 1: Data Records Live in Index
  - If this is used, index structure is actually a file organization for the data records (instead of a Heap file or a sorted or hashed file).
  - At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency. Right?)

    否则，有几个树就有几份拷贝

  - If data records are very large, # of (leaf) pages containing data entries is high. Implies size of auxiliary information in the index is also large, typically.

Alternatives for Data Entries (Contd.)

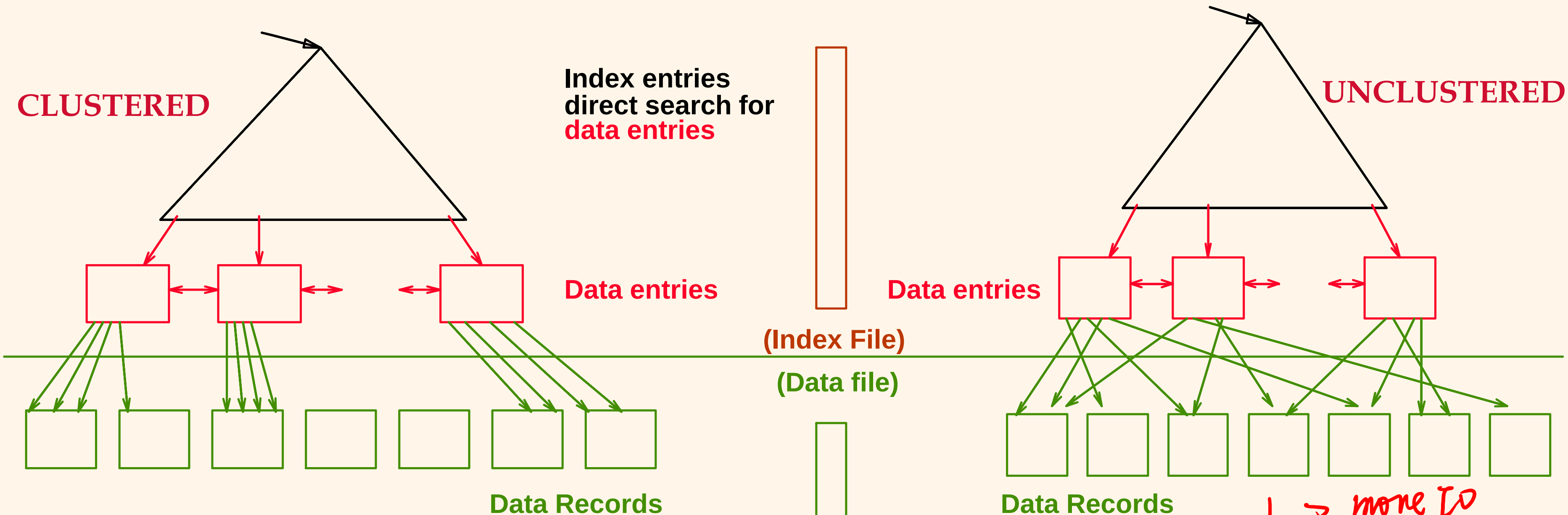
- Alternatives 2 and 3: Key/Rid or Key/RidList
  - Data entries typically much smaller than data records. (Portion of index structure used to direct searches, which depends on size of data entries, is much smaller than with Alternative 1.)
  - Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if the search keys are of fixed length.
  - Can treat Key/Rid pair in a composite key-like fashion in higher levels of index to handle case where a (big) RidList could overflow a leaf page.

Index Classification

- *Primary* vs. *secondary*: If search key contains the primary key, then called the primary index.
  - *Unique* index: Search key contains a *candidate* key.
- *Clustered* vs. *unclustered*: If order of data records is the same as, or `close to', the order of stored data entries, then called a clustered index.
  - Alternative 1 implies clustered; in practice, clustered also implies Alternative 1 (sorted files are rare).
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records via index varies *greatly* based on whether index is clustered or not!

Clustered vs. Unclustered Index

- Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.
  - To build a clustered index, first sort the Heap file (with some free space left on each page for future inserts).
  - Overflow pages may be needed for inserts. (Thus, order of data recs is `close to', but not identical to, the sort order.)



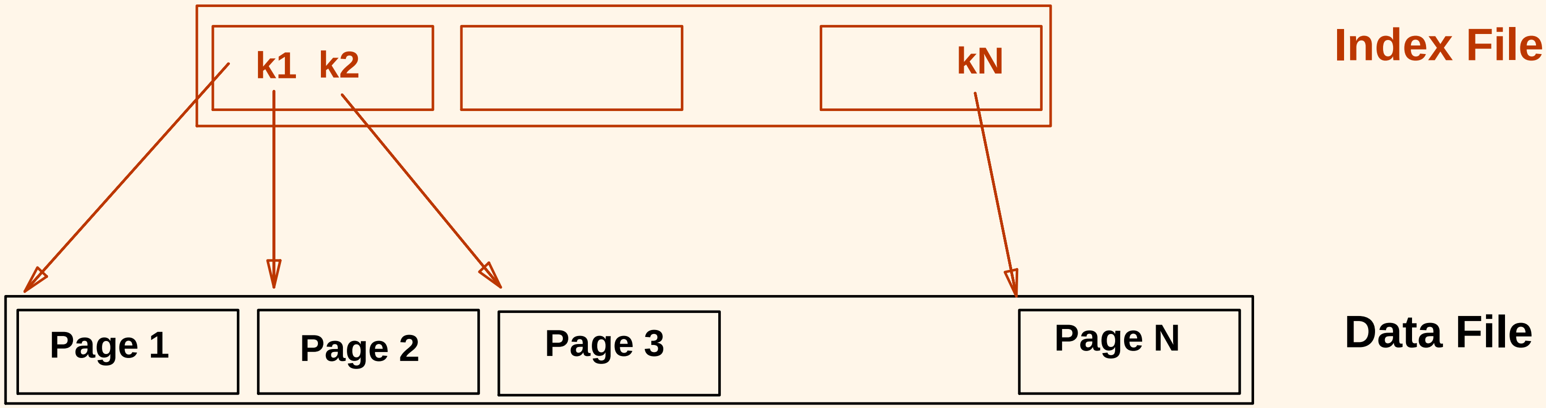
Do range search → more IO

Next: Tree Indexes

- *ISAM*: static structure (Indexed Sequential Access Method)
- *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.

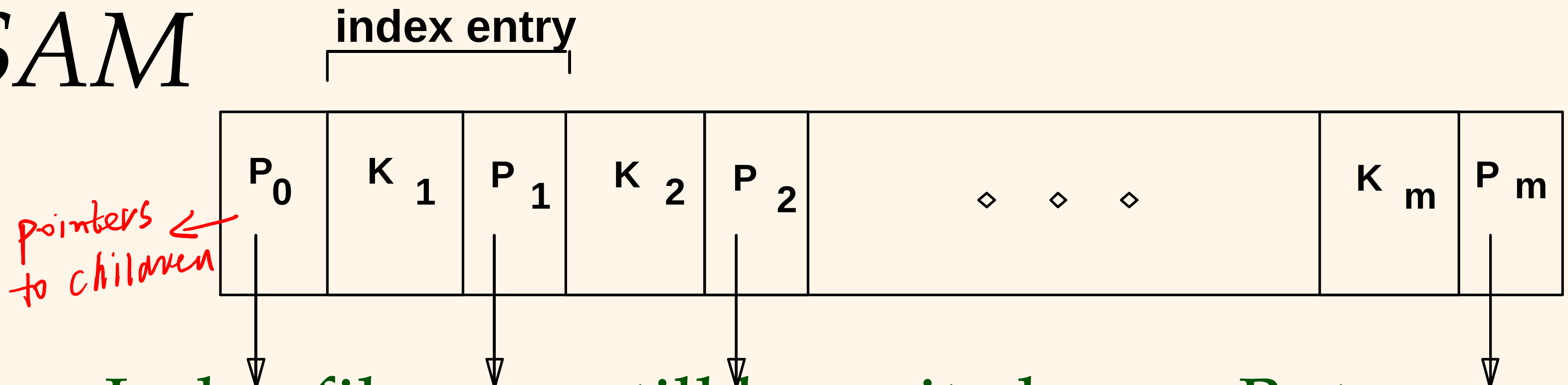
Range Searches

- *``Find all students with gpa > 3.0''*
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search can be quite high.
- Simple idea: Create an `index' file.

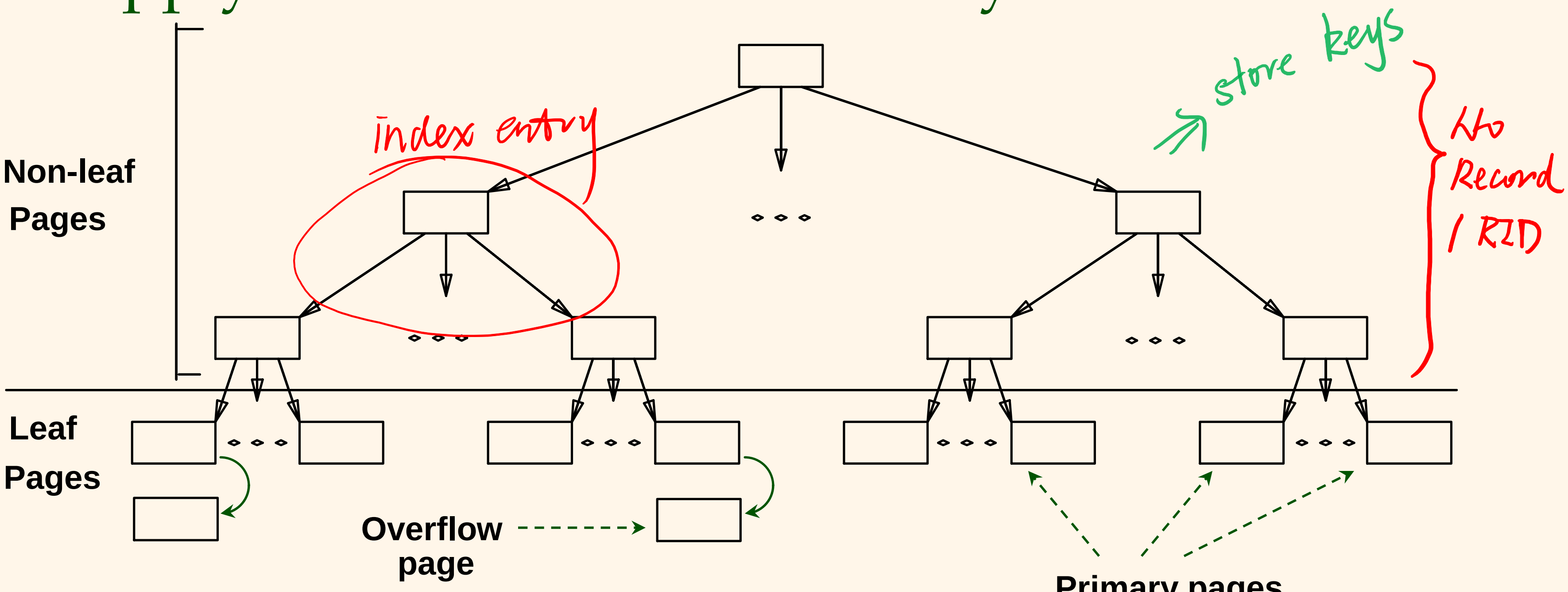


Can now do the binary search on the (smaller) index file!

ISAM



- Index file may still be quite large. But, we can apply the same idea recursively to address that!



Leaf pages contain data entries.

Index key attribute: price

k^* →

1) Record itself : clusterd Index → then tree is the data home, the usually only have one copy

2) RID for k : much smaller

(k, RID) → heap file is the data home, could have multiple index tree.

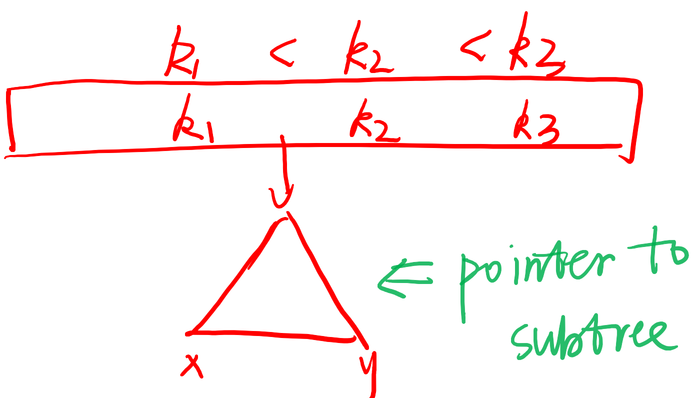
3) List of RIDs → for the records which have same key

↓

Duplicate

↓

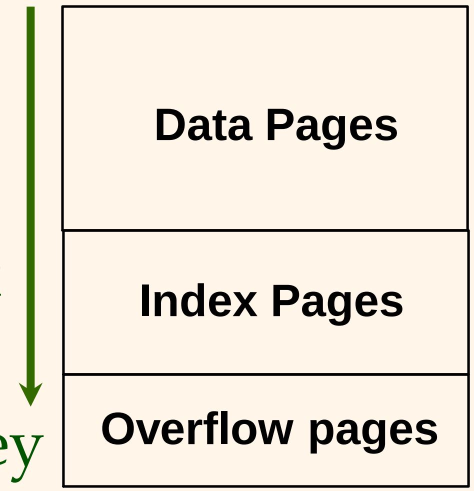
key entry



$k_1 \le x < y < k_2$

只有穿边可以equal

Comments on ISAM

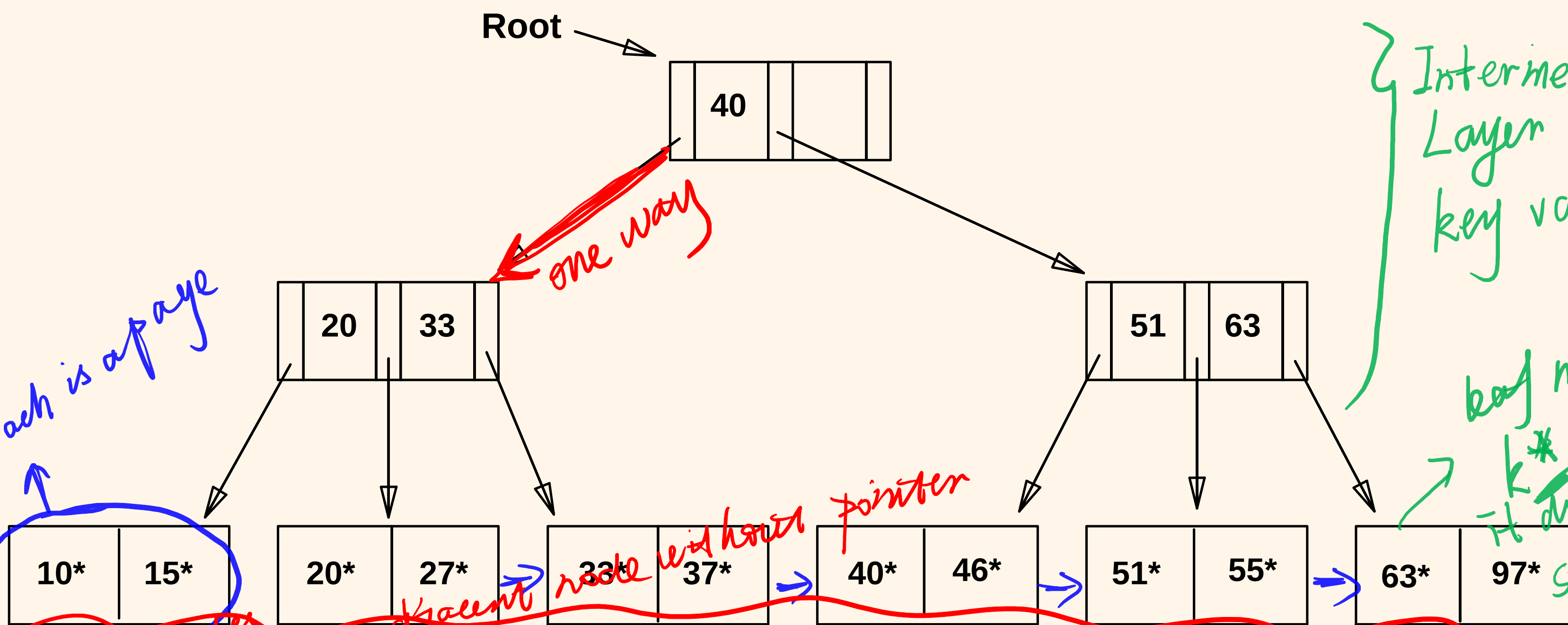


- *File creation*: Leaf (data) pages first allocated sequentially, sorted by search key; then index pages allocated, and then overflow pages.
- *Index entries*: <search key value, page id>; they "direct" searches for *data entries*, which are in leaf pages.
- *Search*: Start at root; use key comparisons to go to leaf. I/O cost $\propto \log_F N$; F = # entries/index pg, N = # leaf pgs
- *Insert*: Find leaf data entry belongs to, and put it there.
- *Delete*: Find and remove from leaf; if empty overflow page, de-allocate.

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

Example ISAM Tree

❖ Suppose each node can hold 2 entries (really more like 200 since the nodes are disk pages)



Intermediate Layer stores key value.

one way

Each is a page

leaf node

k*

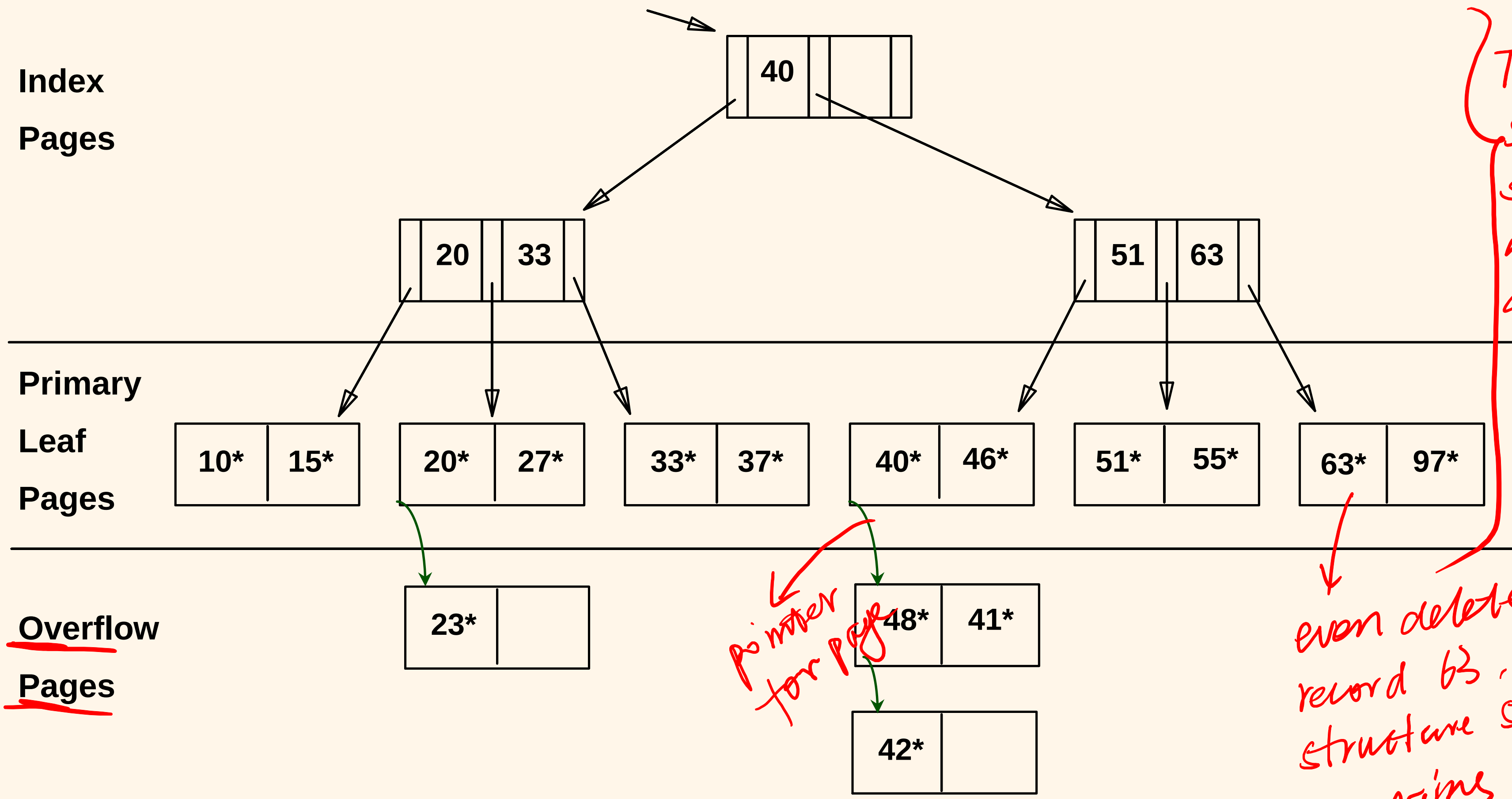
It doesn't store the key value directly

get adjacent node without pointer

store leaf nodes compactly.

There is a implicit link between leaf nodes.

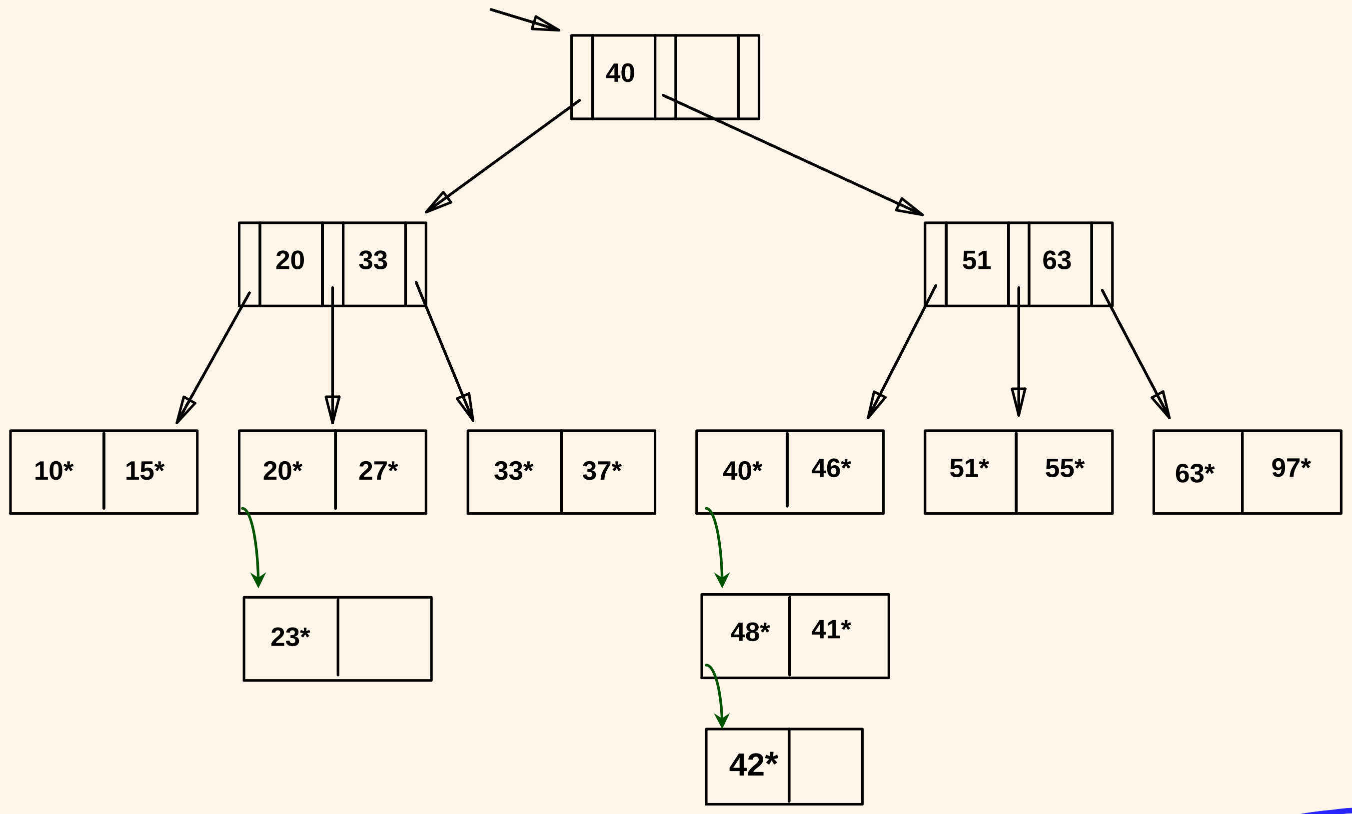
Let's Insert 23*, 48*, 41*, 42* ...



This structure should not change

even delete record 63, structure still remains

... Then Delete 42*, 51*, 97*



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