Indexes

What is Index?

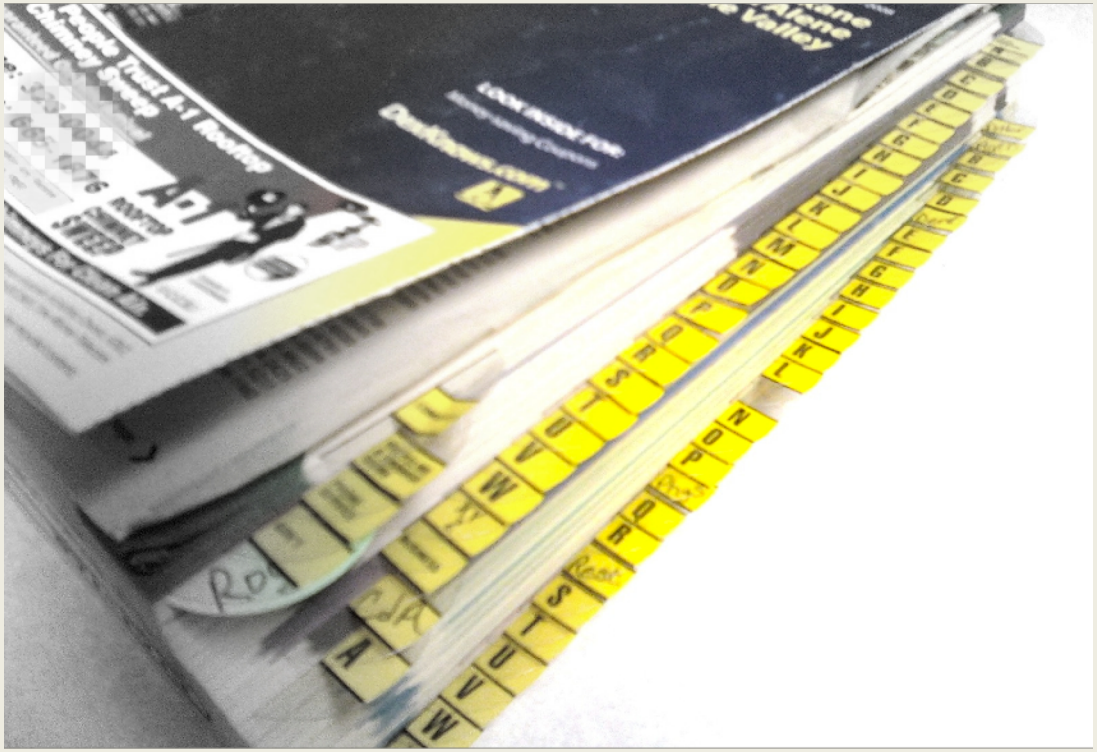
Book has been arranged via categories, subjects.

Same categories will be stored in the same area.

Book is data

Catalogue is the index

What is Index?



Phone number has been arranged in Alphabets, groups (work, friends, …)

Phone number is the data

The Alphabets and groups are the index.

What is Index?

- Auxiliary data
- Properly organised (data structure)
- To facilitate data search

Indexes

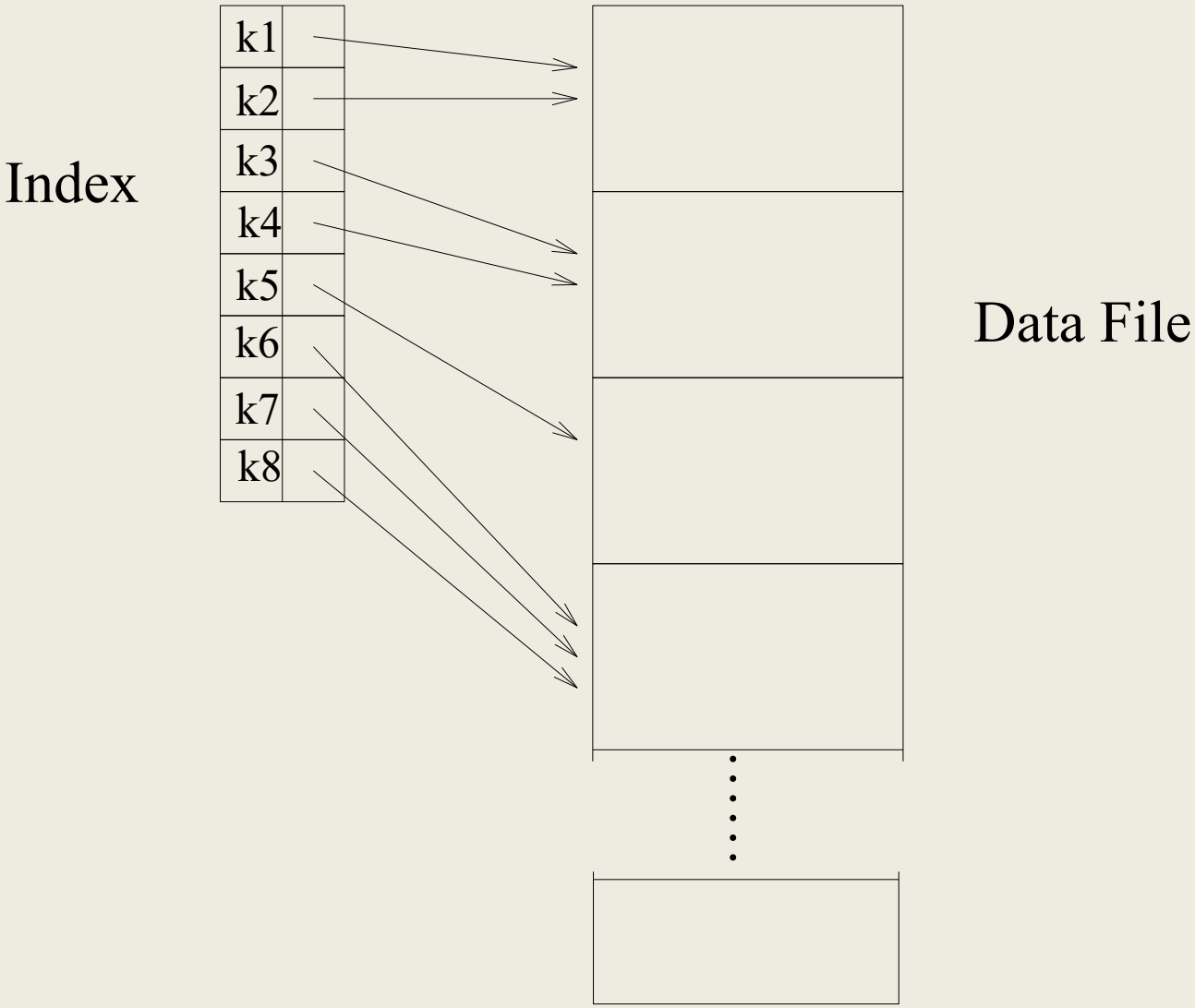
Word indexes in a book:

INDEXES

aardvark 25,36	lion 18
bat 12	llama 17,21,22
cat 1,5,12	sloth 18
dog 3	tiger 18
elephant . . 17	wombat . . . 27
emu 28	zebra 19

- A table of key values, where each entry gives places where key is used.
- Aim: efficient access to records via key.

Indexing Structure



Indexing Structure

- Index is collection of data entries k∗.
  - Each data entry k∗ contains enough information to retrieve (one or more) records with search key value k.

- Indexing:
  - How are data entries organized in order to support efficient retrieval of data entries with a given search key value?
  - Exactly what is stored as a data entry?

Alternatives for Data Entries in an Index

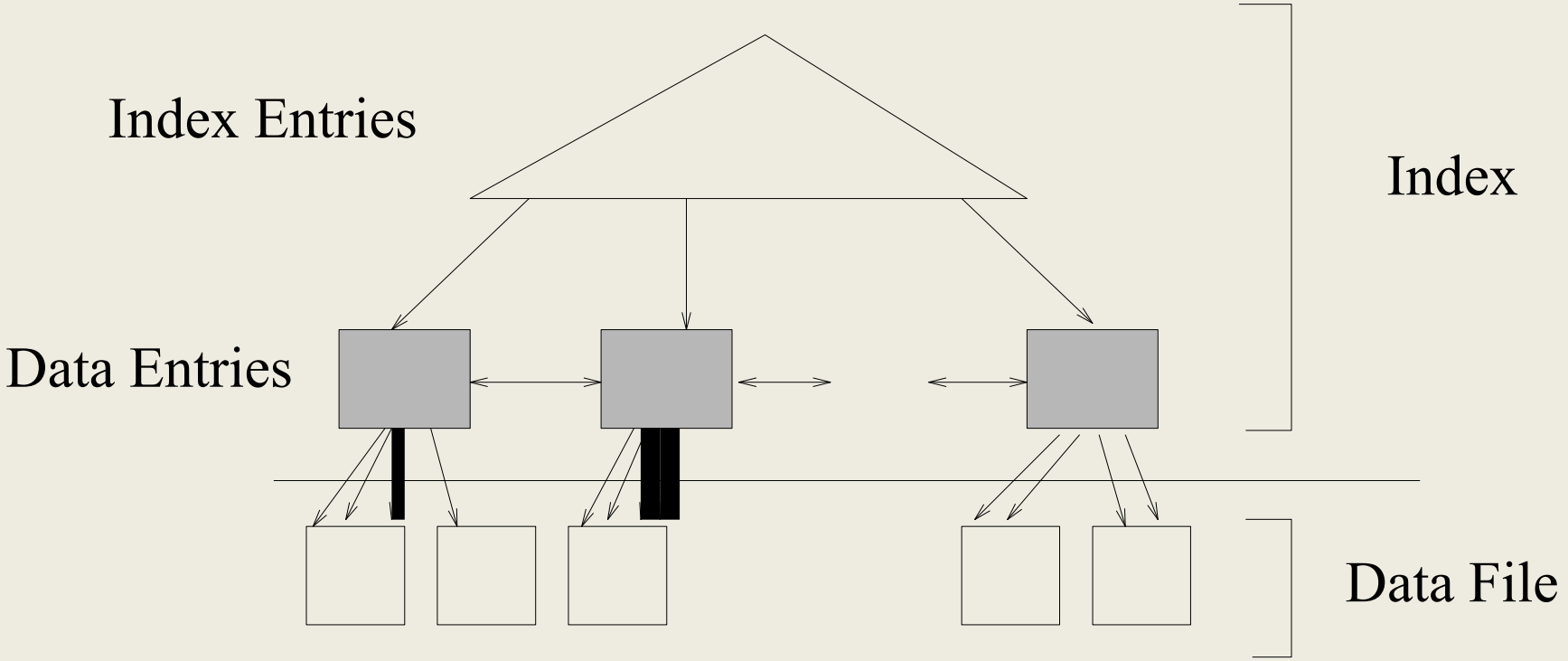
- A data entry k∗ is an actual data record (with search key value k).

- A data entry is (k, rid) pair (rid is the record id of a data record with search key value k).
  - E.g. Example: (Xuemin Lin, page 12), (Xuemin Lin, page 100)

- A data entry is a (k, rid − list) pair (rid − list is the list of record ids of data records with search key value k).
  - E.g. (Xuemin Lin, page 12, page 100)

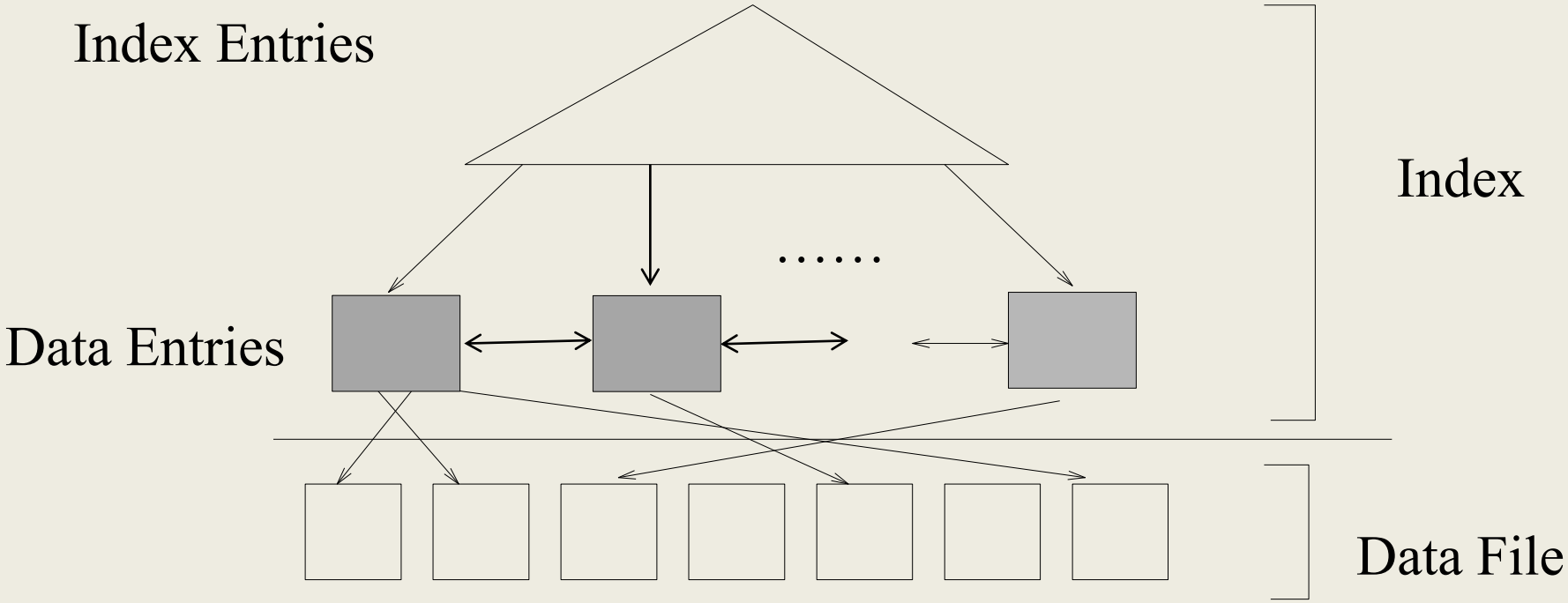
Clustered Index

- Clustered: a file is organized of data records is the same as or close to the ordering of data entries in some index.
- Typically, the search key of file is the same as the search key of index.



Unclustered Index

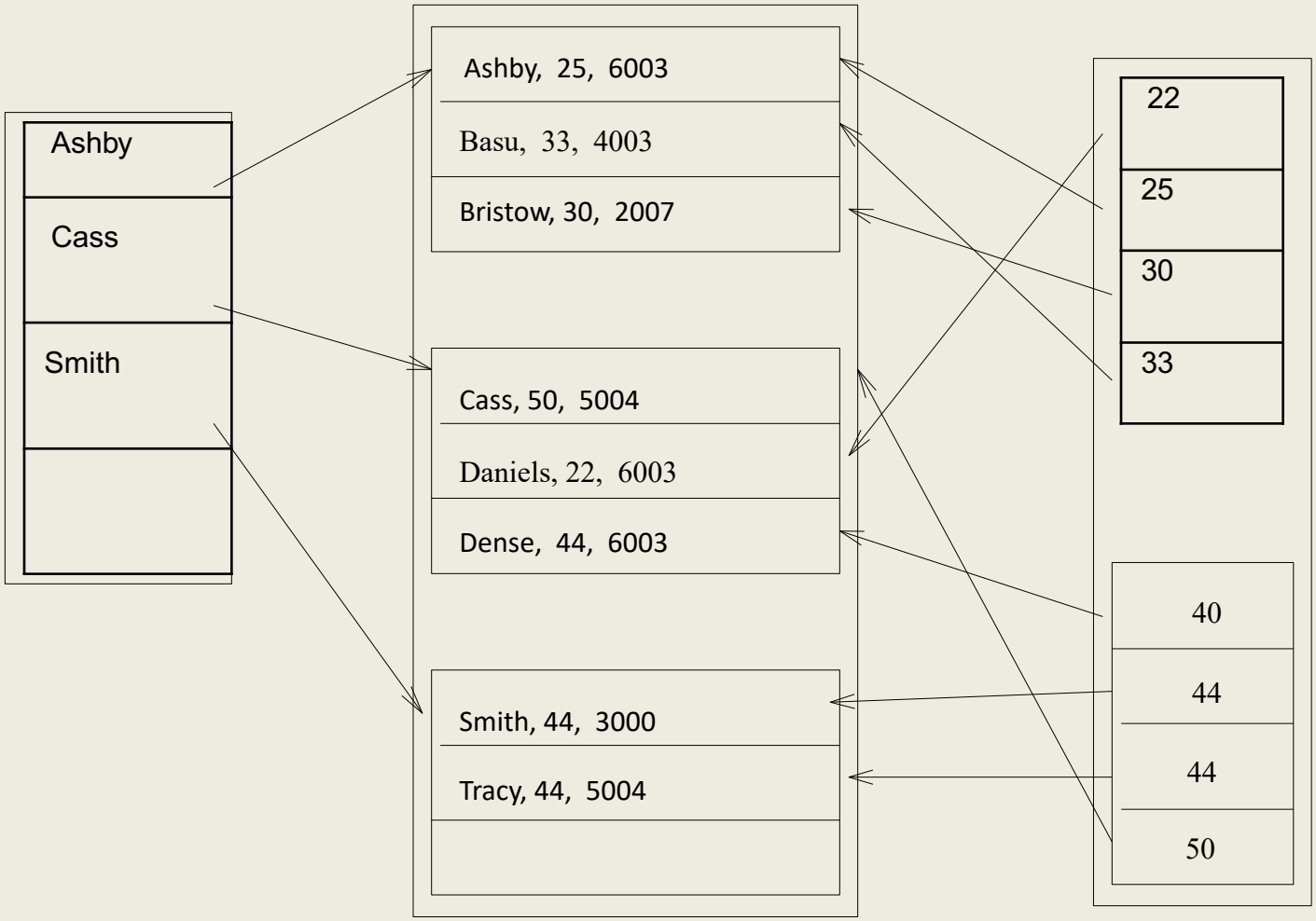
- Clustered indexes are relatively expensive to maintain.
- A data file can be clustered on at most one search key.



Dense VS Sparse Indexes

- Dense Index and Sparse Index
  - Dense Index contains (**at least**) one data entry for every search key value.
  - Sparse Index may note and one search key can points to a set of data entries

Q: Can we build a sparse index that is not clustered?



Sparse Index VS Dense Index

Primary and Secondary Indexes

- Primary: Indexing fields include **primary key**.
  - There can be at most one primary index for a table
- Secondary: otherwise.
  - Composite search keys: search key contains several fields.

Introduction Indexing Techniques

As for any index, 3 alternatives for data entries **k***:

- Data record with key value **k**
- <**k**, rid of data record with search key value **k**>
- <**k**, list of rids of data records with search key **k**>

Choice is orthogonal to the *indexing technique* used to locate data entries **k***.

- Tree-Structured indexes are best for **sorted access** and **range queries**.
- Hash-based indexes are best for **equality selections**. **Cannot** support range searches.

We will focus on these two types of indexes.

Tree-Structured Indexes

- Tree-structured indexing techniques support both ***range searches*** and ***equality searches***.

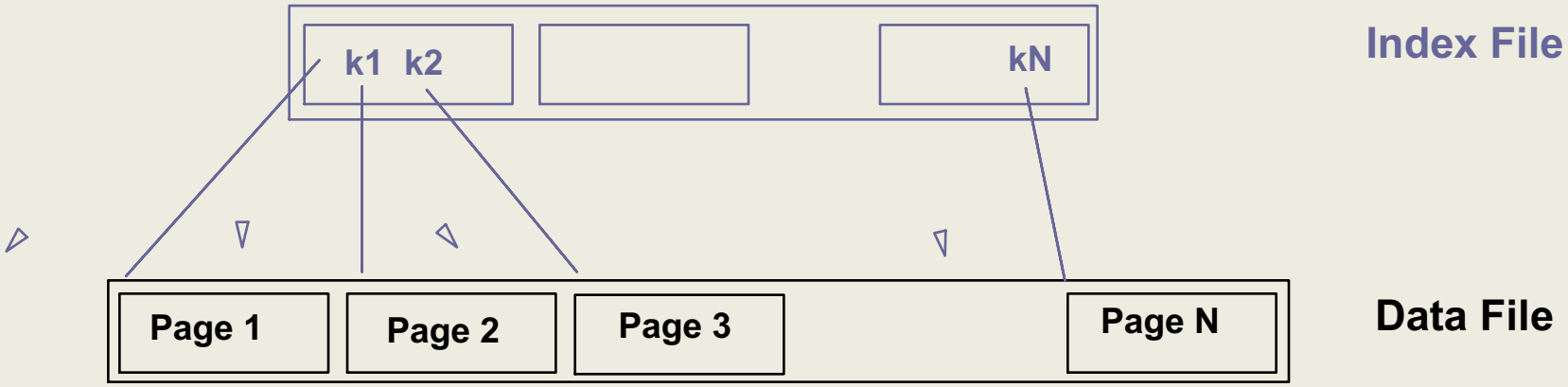
  - ***ISAM*****:** static structure;
  - ***B+ tree*****:** dynamic, adjusts gracefully under inserts and deletes.

Range Searches

"Find all students with gpa > 3.0"

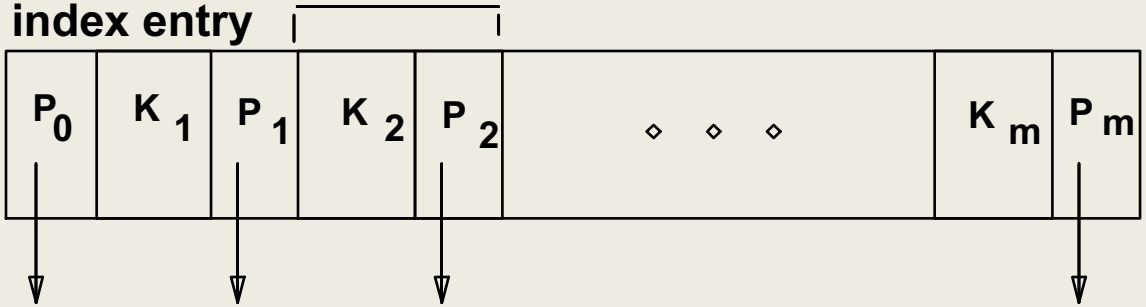
- If data is in sorted file, do binary search to find first such student, then scan to find others.
- Cost of binary search can be quite high.

Simple idea: Create an 'index' file.

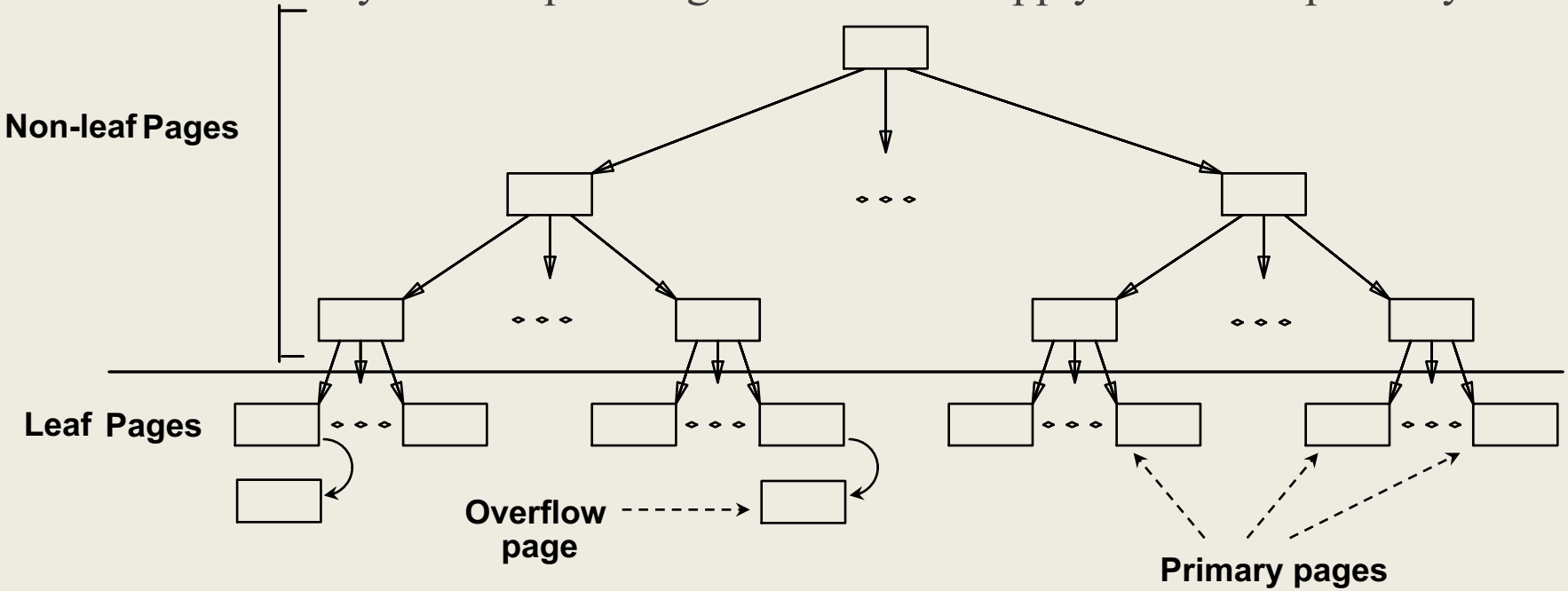


☛ *Can do binary search on (smaller) index file!*

ISAM



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



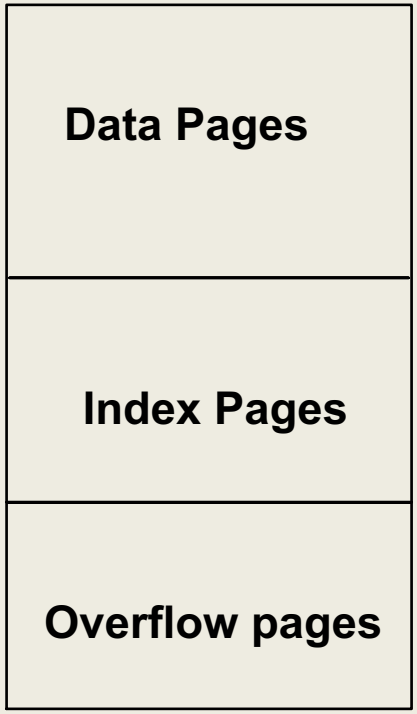
☛ *Leaf pages contain* ***data entries.***

Comments on ISAM

- *File creation*: Leaf (data) pages allocated sequentially, sorted by each 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 $\log_F N$;

  F = # entries/index pg, N = # leaf pgs

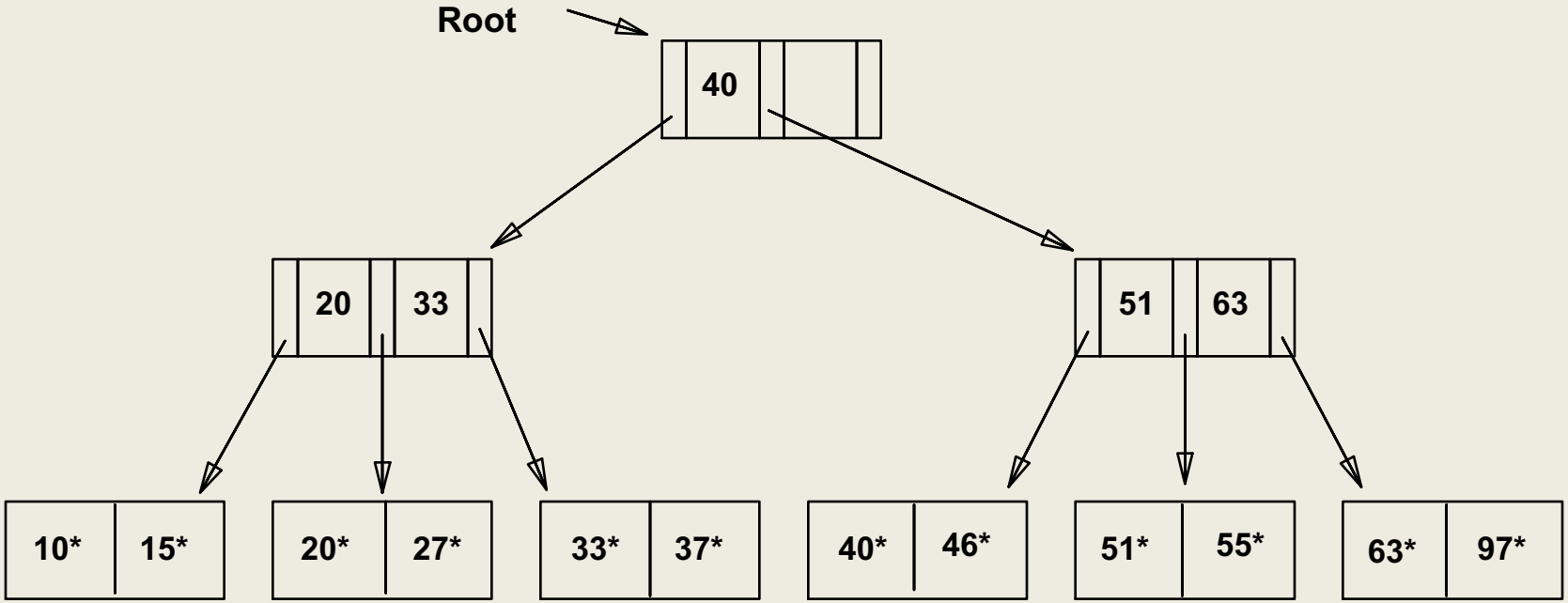
- *Insert*: Find leaf data entry belongs to, and put it there if space is available, else allocate an overflow page, put it there, and link it in.
- *Delete*: Find and remove from leaf; if empty overflow page, de-allocate.



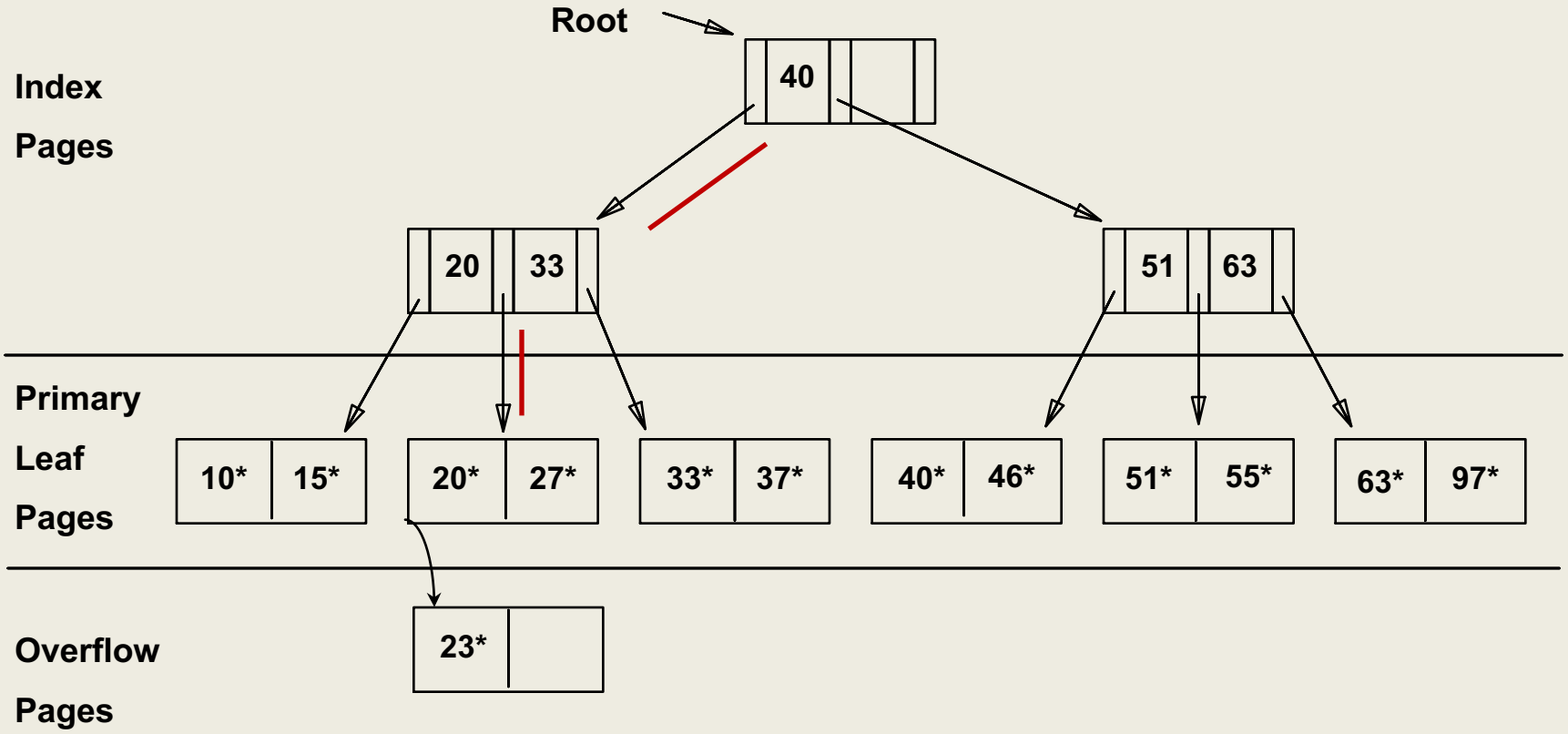
☛ **Static tree structure**: *inserts/deletes affect only leaf pages.*

Example ISAM Tree

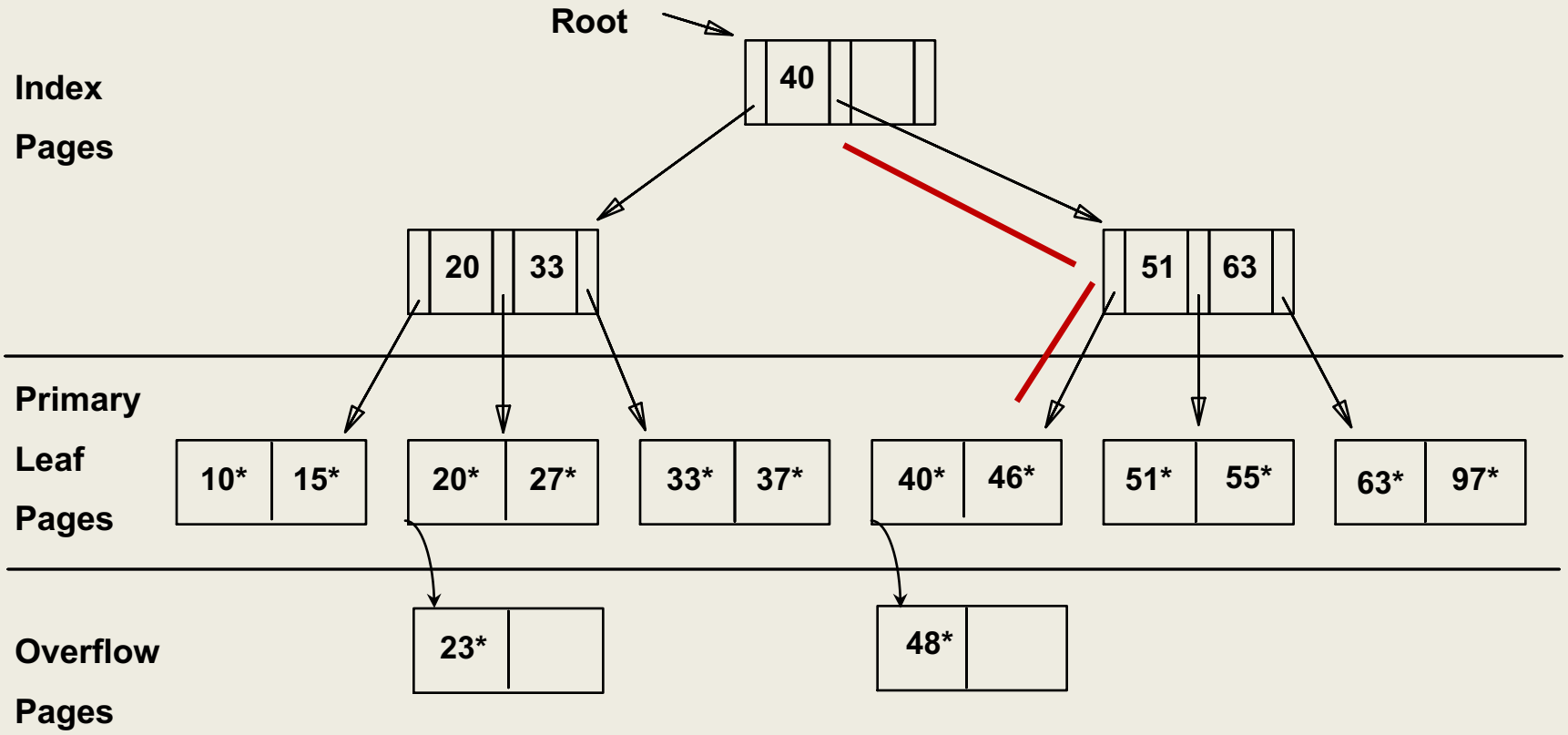
Each node can hold 2 entries; no need for ‘next-leaf-page’ pointers. (Why?)



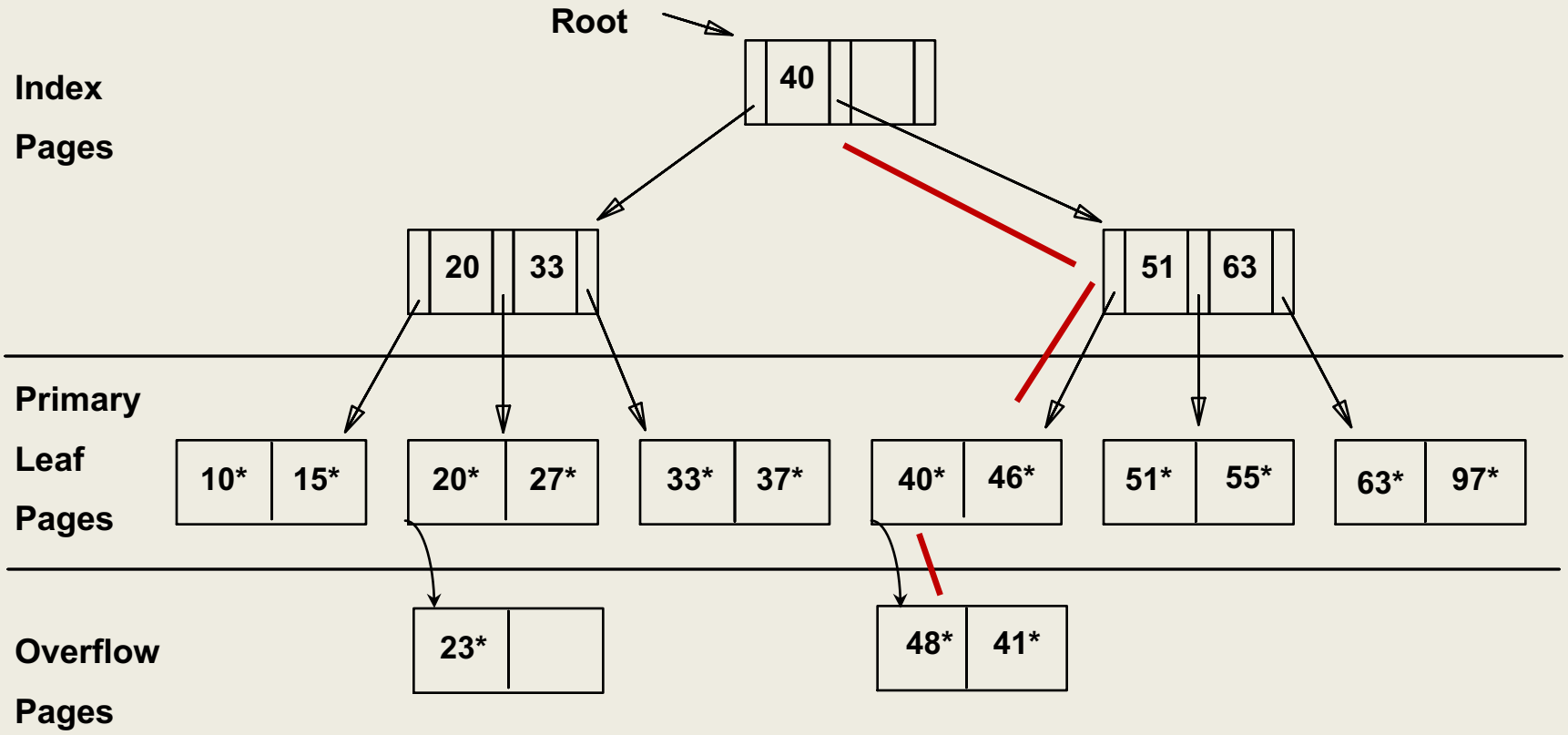
Inserting 23*



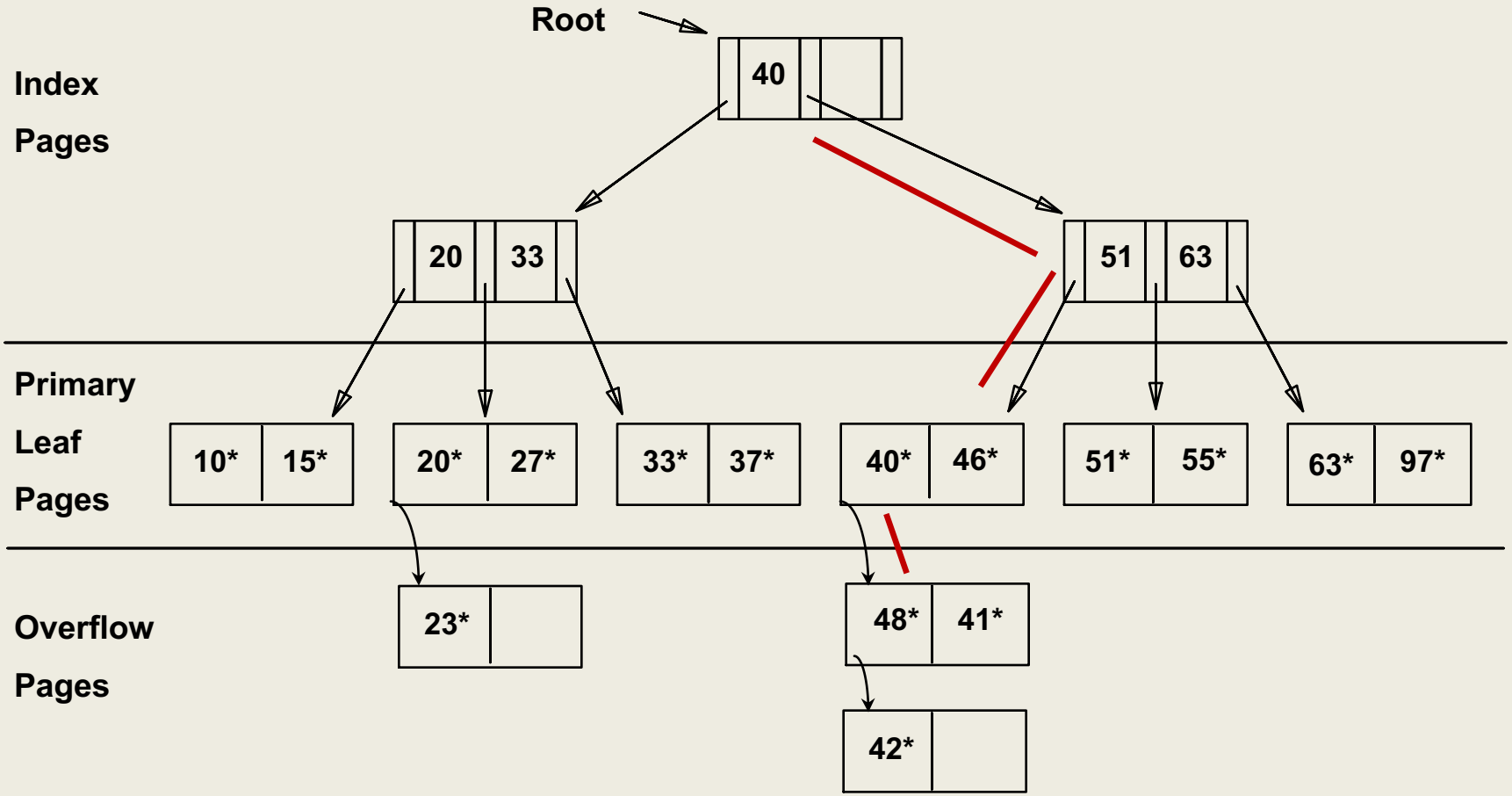
Inserting 48*



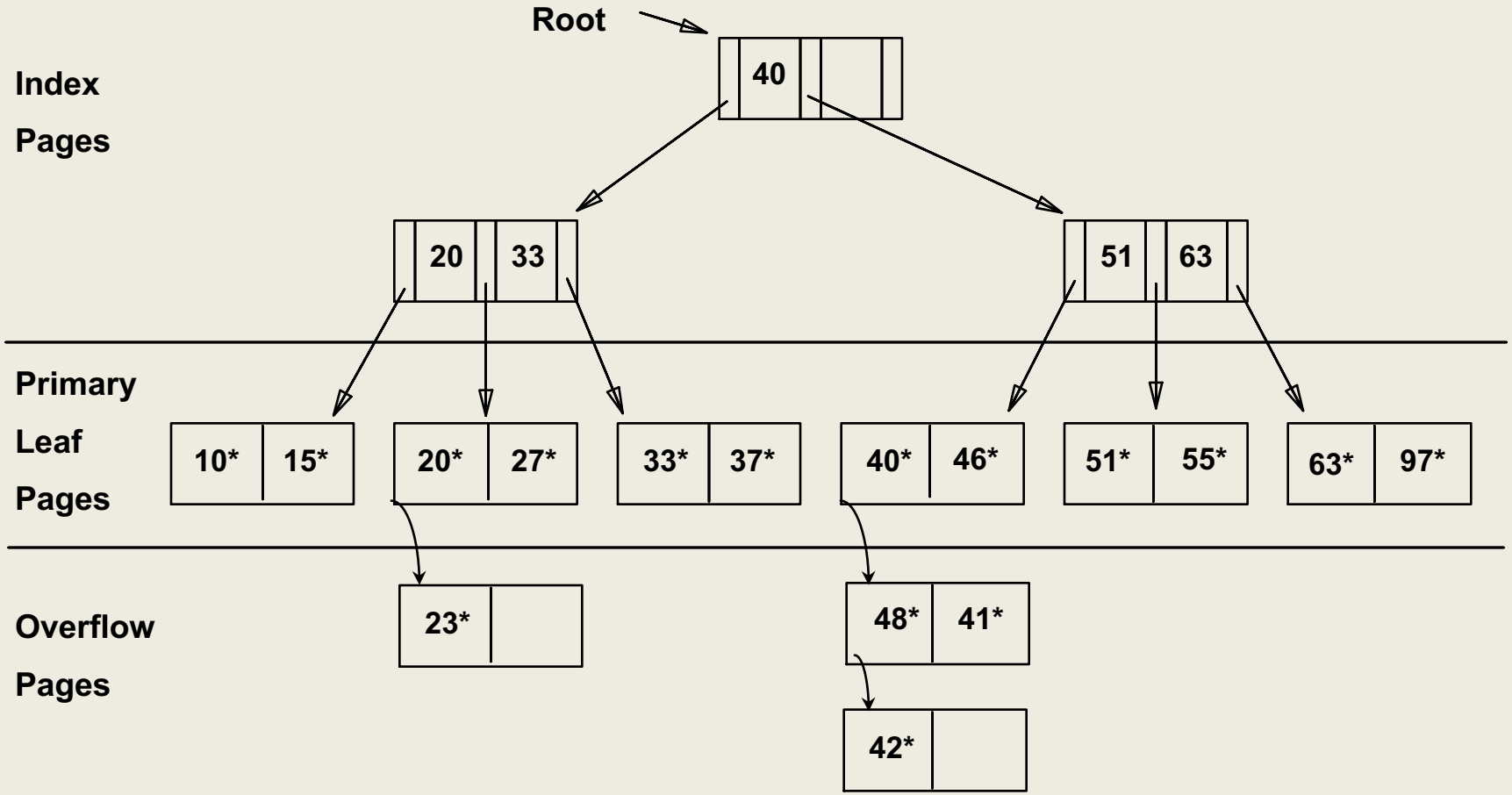
Inserting 41*



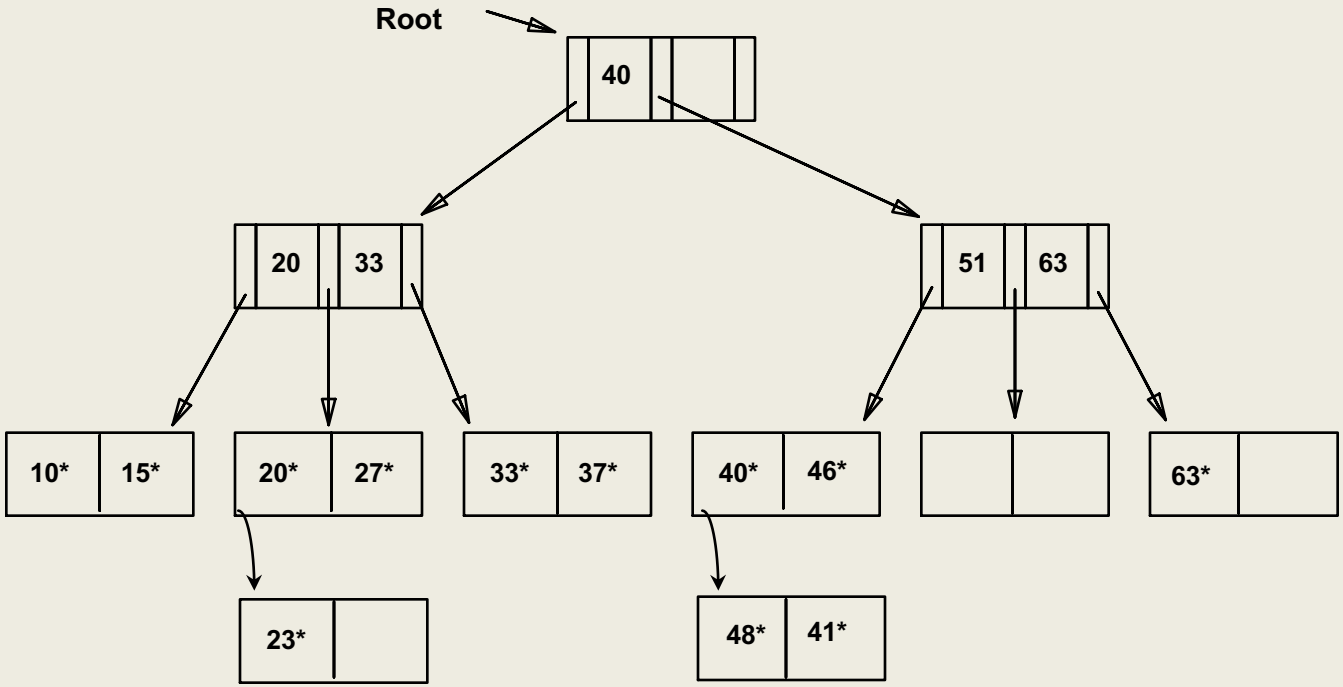
Inserting 42*



Then Deleting 42*, 51*, 97*,55*



... After Deleting 42*, 51*, 97*, 55*



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

B+ Tree: Most Widely Used Index

- Insert/delete at $\log_F N$ cost; keep tree ***height-balanced.***

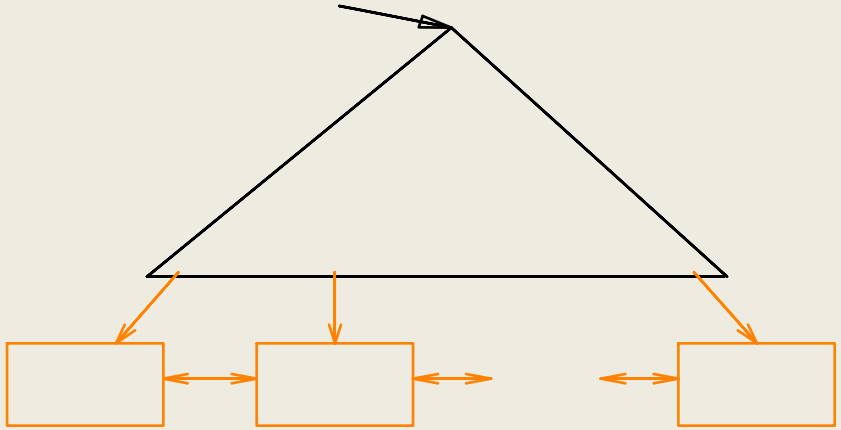
  (F = fanout, N = # leaf pages)

- Minimum 50% occupancy (except for root).

  Each node contains $\lceil (\mathbf{p}/2) \rceil <= \underline{m} <= \mathbf{p}$ entries.

- The parameter **p** is called the *order* of the tree.

- Supports equality and range-searches efficiently.



Index Entries
(Direct search)

Data Entries
("Sequence set")

B+ Tree Structure

The structure of the ***internal nodes*** of a B+- tree of order p:

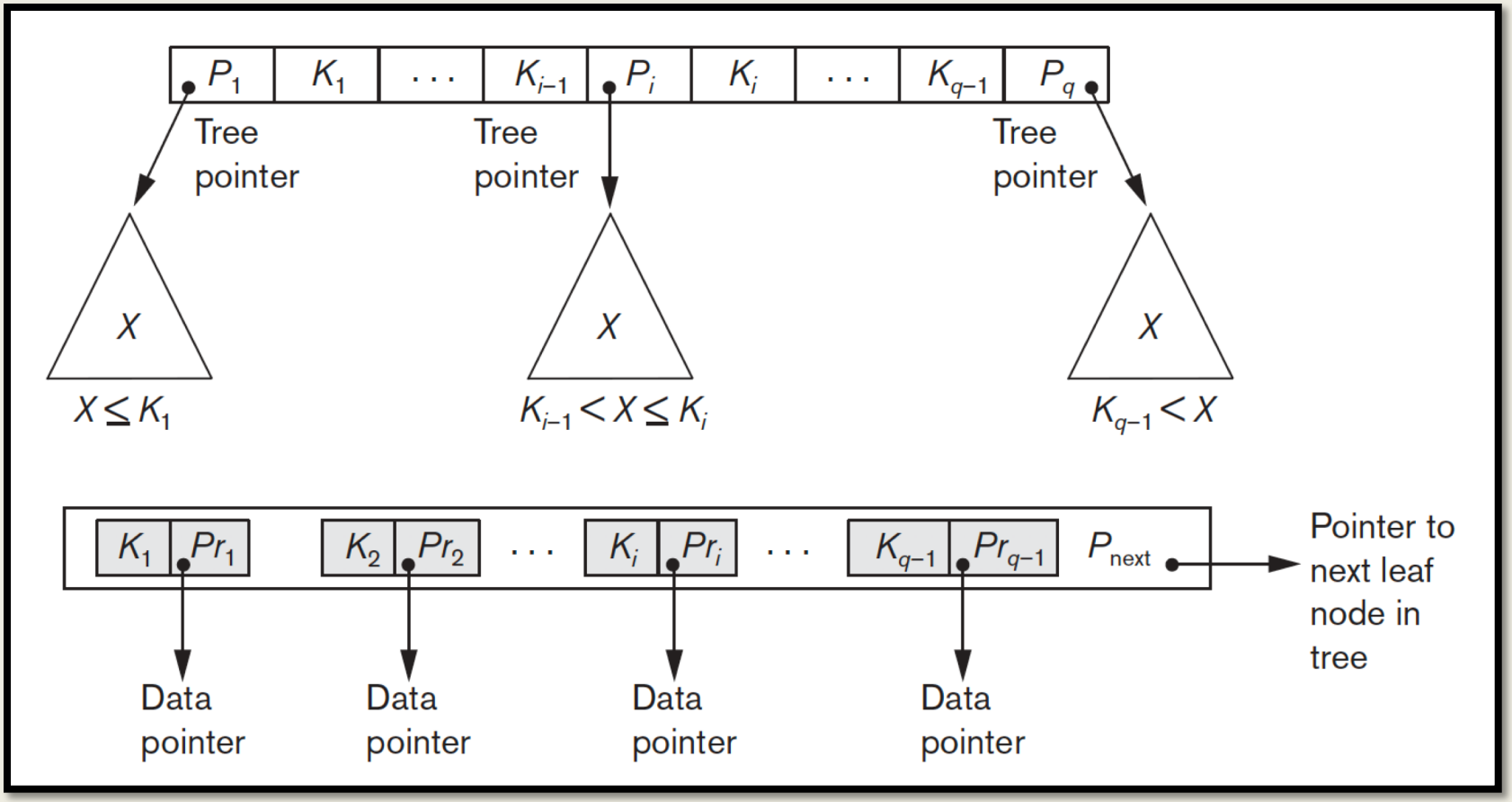
1. Each internal node is of the form $<P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q>$ where $q \leq p$ and each P_i is a tree pointer.
2. Within each internal node, $K_1 < K_2 < ... < K_{q-1}$.
3. For all search field values X in the subtree pointed at by P_i, we have $K_{i-1} < X \leq K_i$ for $1 < i < q$; $X \leq K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$.
4. Each internal node has at most p tree pointers.
5. Each internal node, except the root, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at least two tree pointers if it is an internal node.
6. An internal node with q pointers, $q \leq p$, has $q - 1$ search field values.

B+ Tree Structure

The structure of the ***leaf nodes*** of a B+-tree of order p:

1. Each leaf node is of the form $<<K_1, Pr_1>, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_{next}>$ where $q \leq p$, each Pr_i is a data pointer, and P_{next} points to the next leaf node of the B+-tree.
2. Within each leaf node, $K_1 \leq K_2 ... , K_{q-1}$, $q \leq p$.
3. Each Pr_i is a data pointer that points to the record whose search field value is K_i or to a file block containing the record (or to a block of record pointers that point to records whose search field value is K_i if the search field is not a key).
4. Each leaf node has at least $\lceil (p/2) \rceil$ values.
5. All leaf nodes are at the same level.

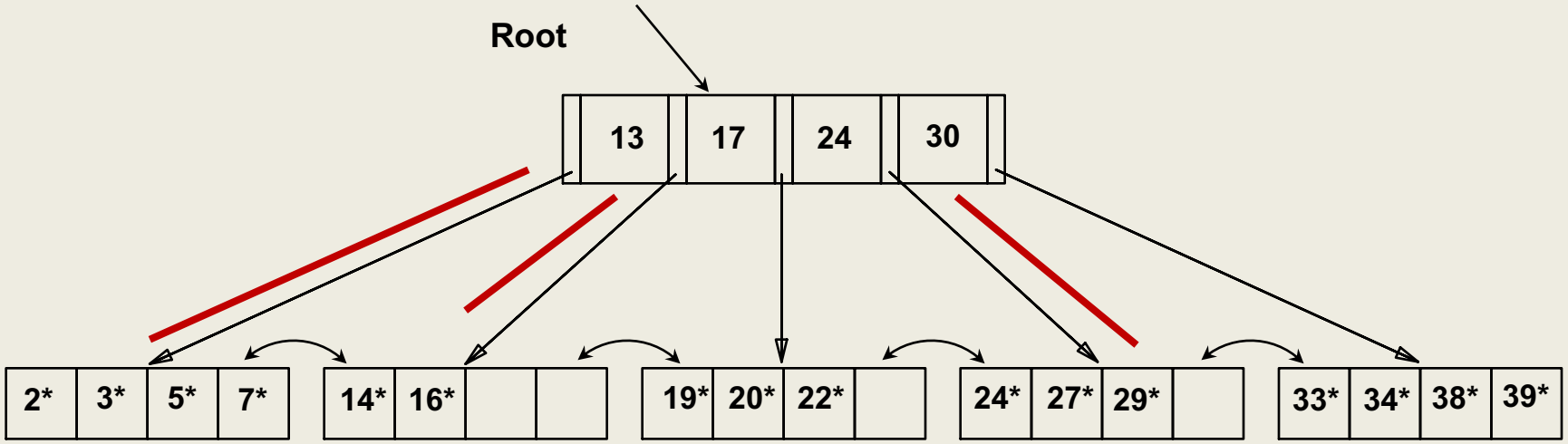
B+ Tree Structure



Example B+ Tree

Search begins at root, and key comparisons direct it to a leaf.

Search for 5*, 15*, all data entries >= 24* ...



☛ ***Based on the search for 15*, we know it is not in the tree!***

B+ Trees in Practice

Typical order: 100. Typical fill-factor: 67%.

- average fanout = 133

Typical capacities:

- Height 4: 133^4 = 312,900,700 records
- Height 3: 133^3 = 2,352,637 records

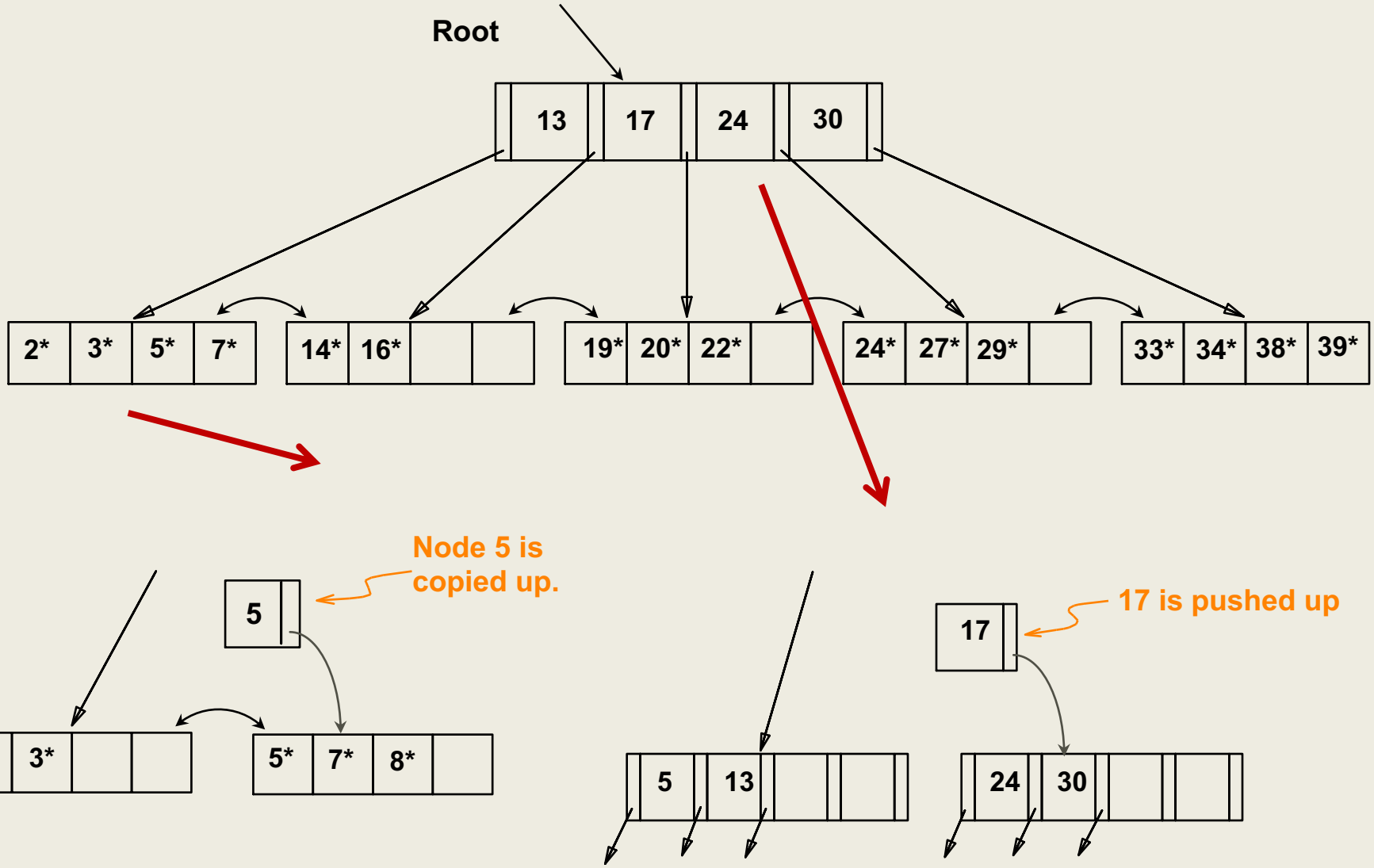
Can often hold top levels in buffer pool:

- Level 1 = 1 page = 8 Kbytes
- Level 2 = 133 pages = 1 Mbyte
- Level 3 = 17,689 pages = 133 MBytes

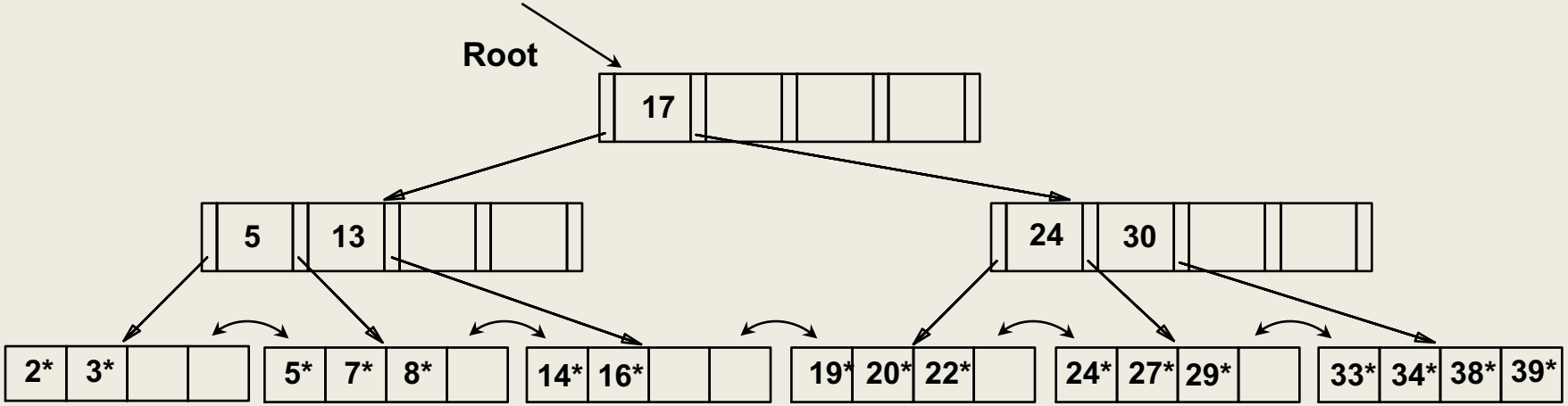
Inserting a Data Entry into a B+ Tree

1. Find correct leaf *L*.
2. Put data entry onto *L*.
   - If *L* has enough space, *done*!
   - Else, must ***split*** *L (into L and a new node L2)*
     - ➢Redistribute entries evenly, **copy up** middle key.
     - ➢Insert index entry pointing to *L2* into parent of *L*.
3. This can happen recursively
   - **To split index node**, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
4. Splits 'grow' tree; root split increases height.
   - Tree growth: gets ***wider*** or ***one level taller at top.***

Inserting 8*


Root
13
17
24
30
2*
3*
5*
7*
14*
16*
19*
20*
22*
24*
27*
29*
33*
34*
38*
39*
Node 5 is copied up.
5
2*
3*
5*
7*
8*
5
13
17 is pushed up
17
24
30

After Inserting 8*

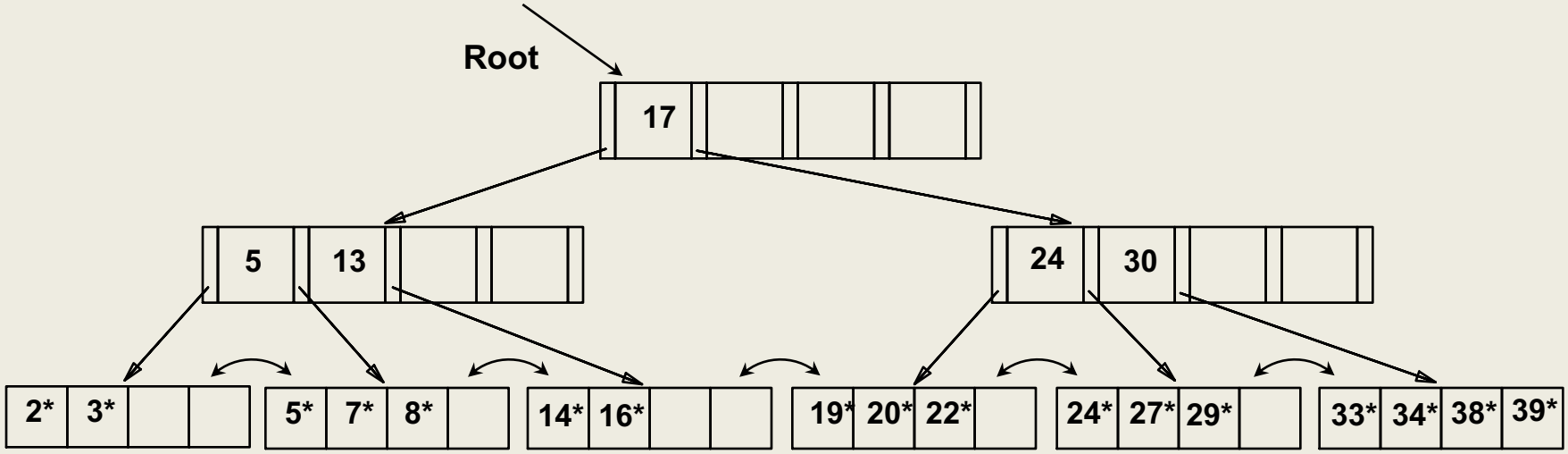


- In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.
  - Redistributing I/O costs at not smaller than those of splitting.
  - It has a chance that redistributing does not work; thus costs for exploring redistribution are wasted.

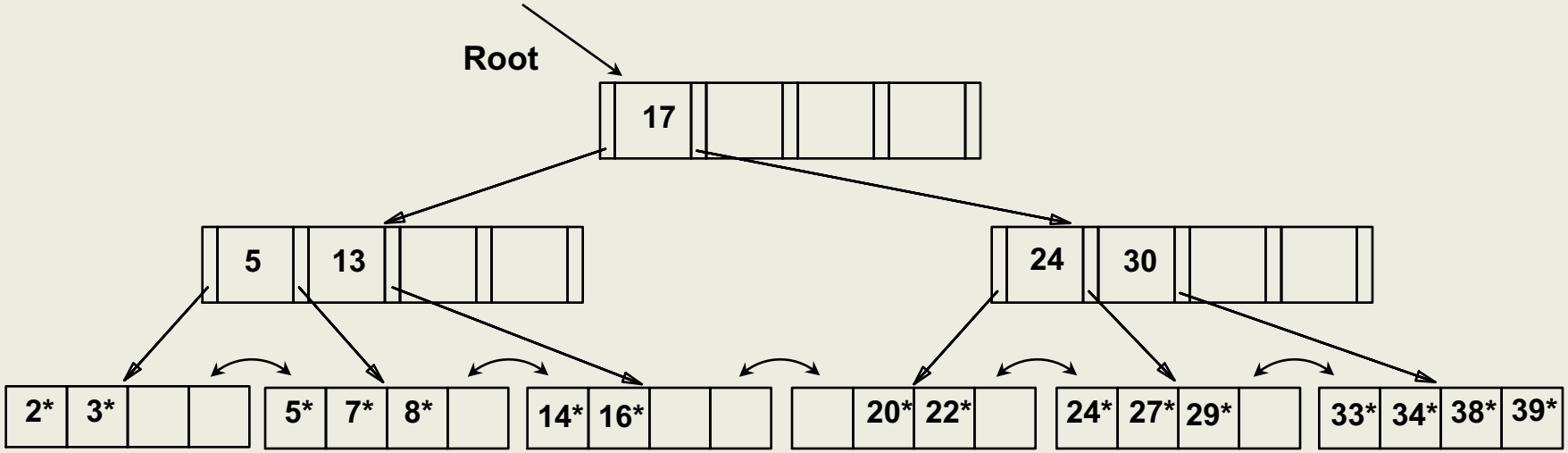
Deleting a Data Entry from a B+ Tree

1. Start at root, find leaf *L* where entry belongs.
2. Remove the entry.
   - If L is at least half-full, *done!*
   - If L has only **p-1** entries,
     - Try to **re-distribute**, borrowing from *sibling (adjacent node with same parent as L)*.
     - If re-distribution fails, ***merge*** *L* and sibling.
3. If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
4. Merge could propagate to root, decreasing height.

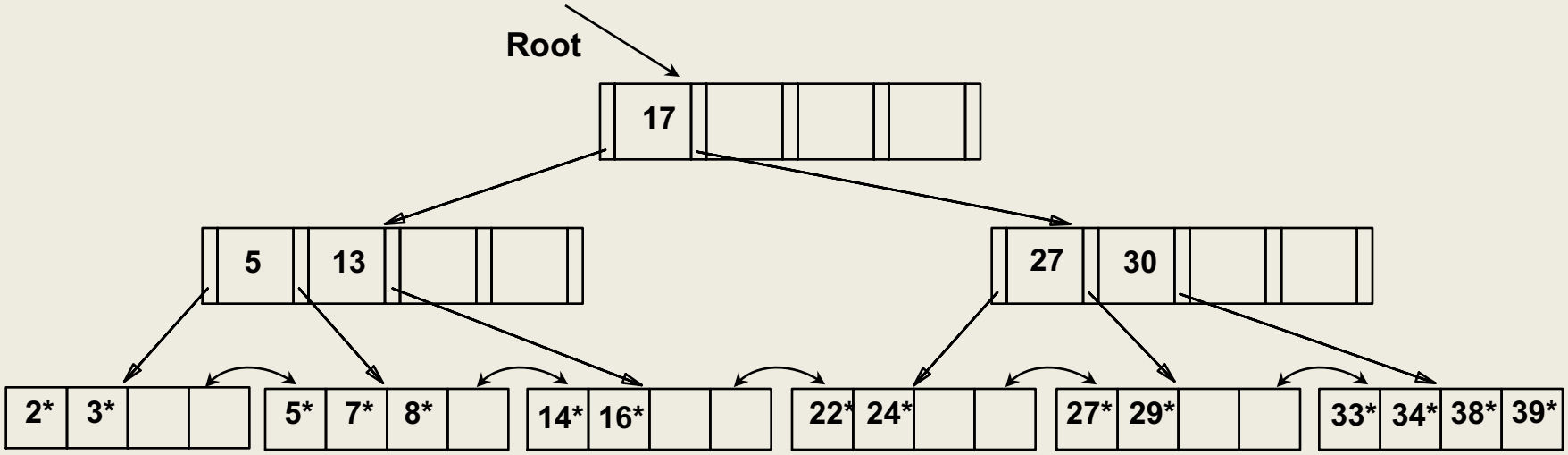
Deleting 19*



Deleting 20*

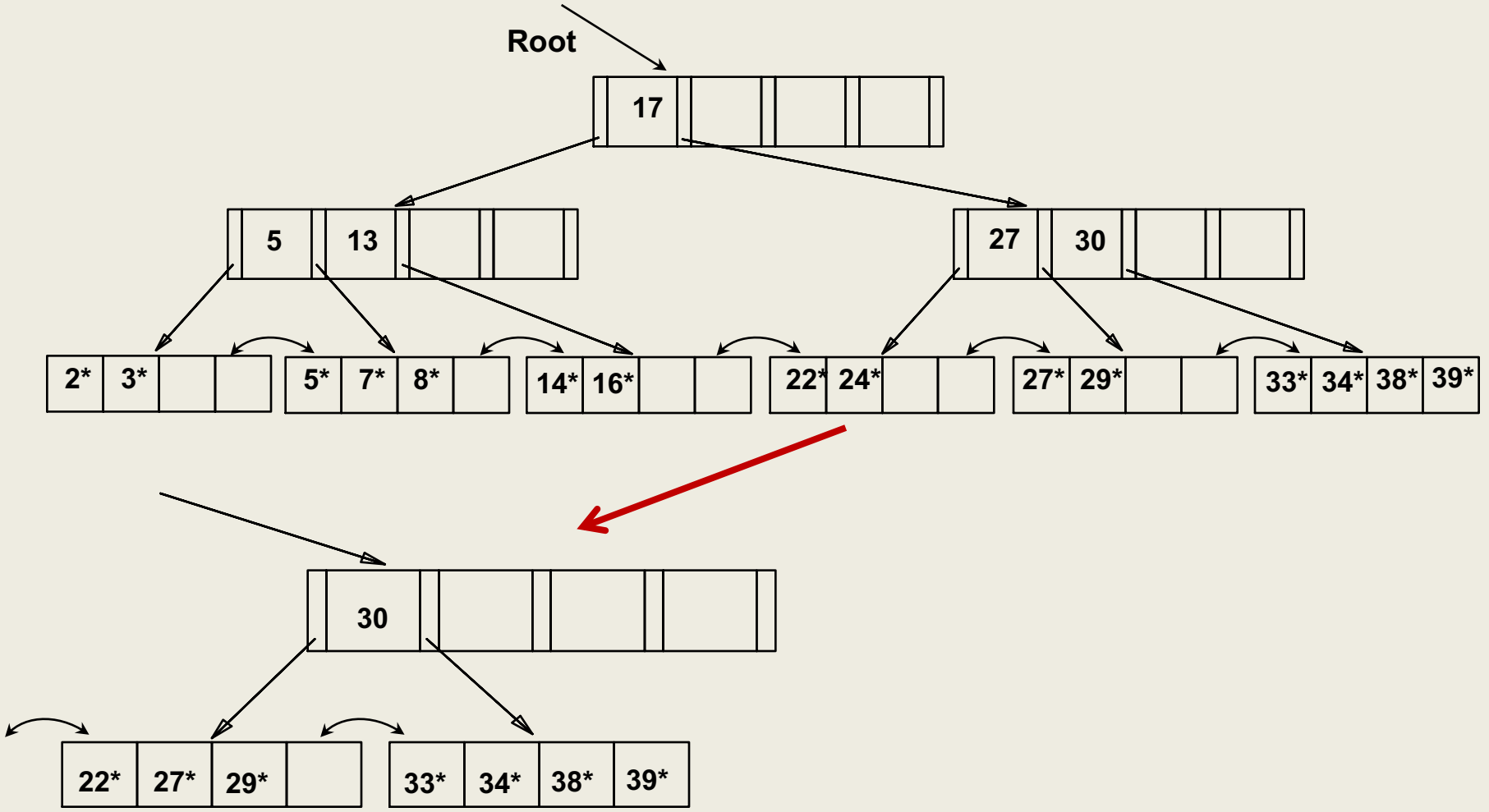


After Deleting 20* ...

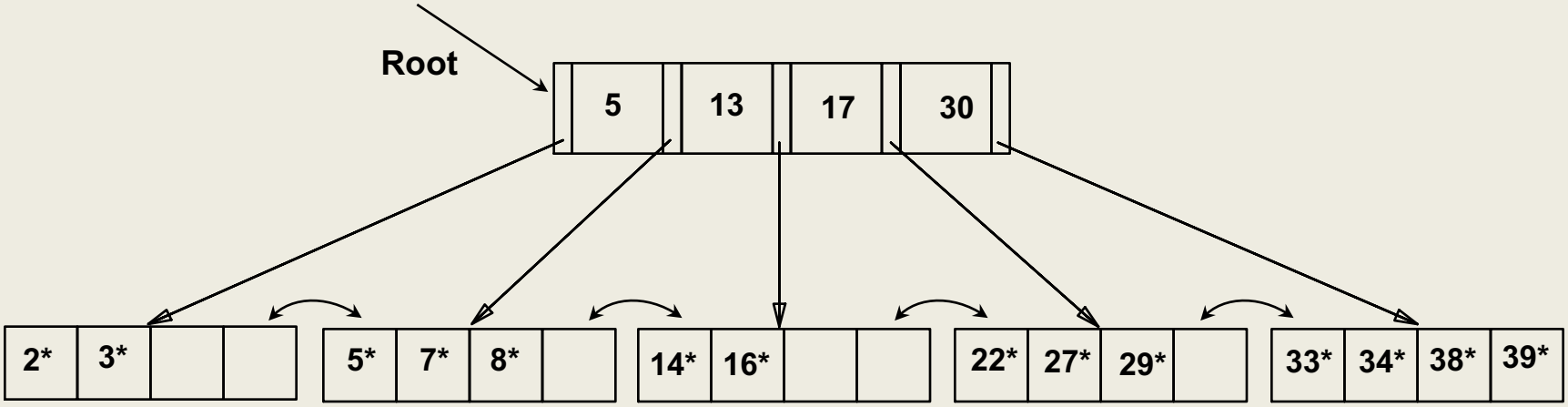


Deleting 20* is done with re-distribution. Notice how middle key is ***copied up***.

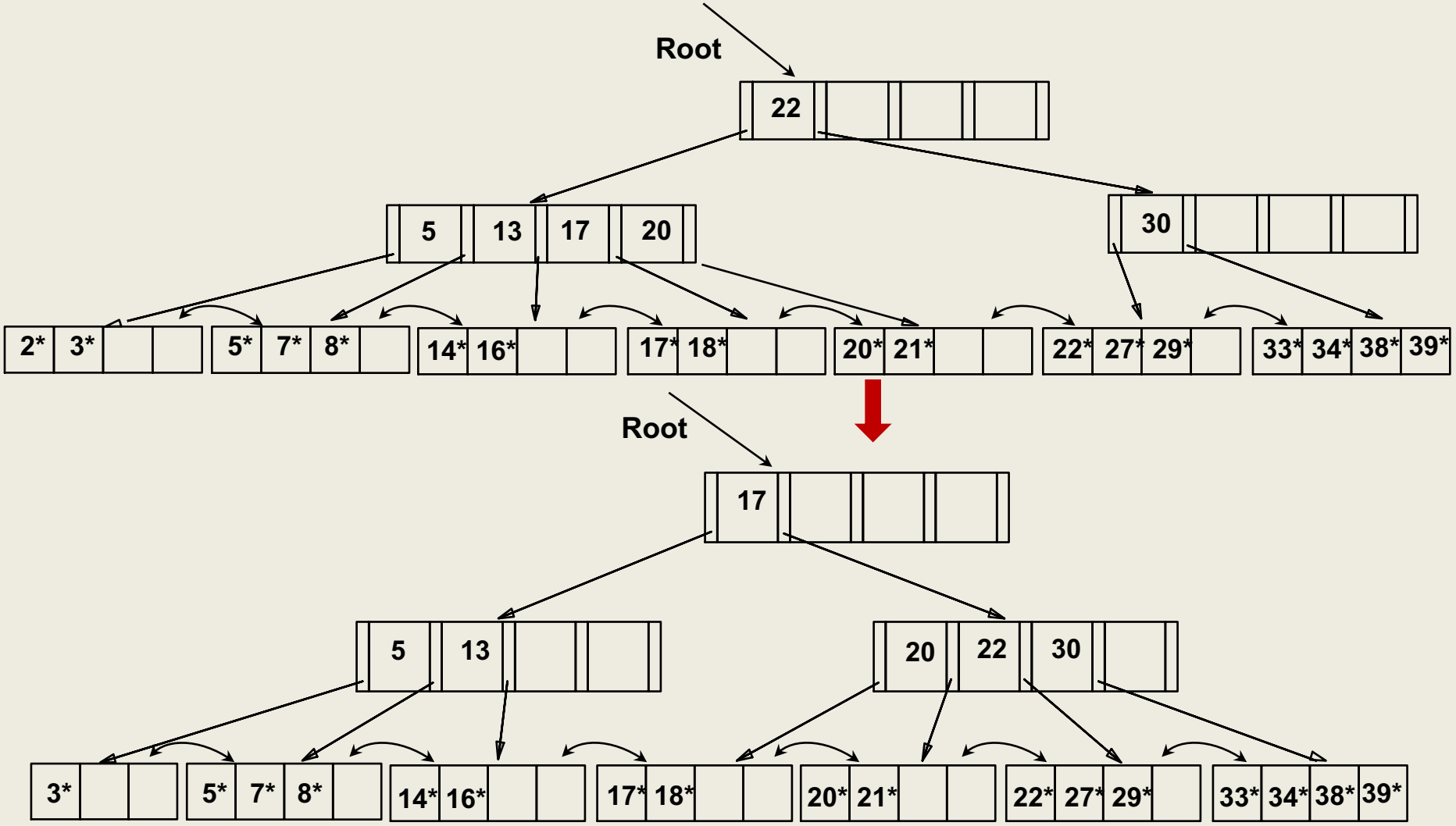
Deleting 24* ...


Root
17
5
13
27
30
2*
3*
5*
7*
8*
14*
16*
22*
24*
27*
29*
33*
34*
38*
39*
30
22*
27*
29*
33*
34*
38*
39*

After Deleting 24*

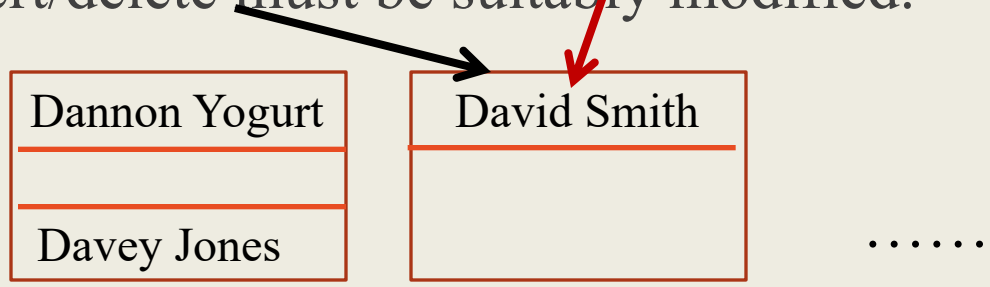


Example of Non-leaf Re-distribution



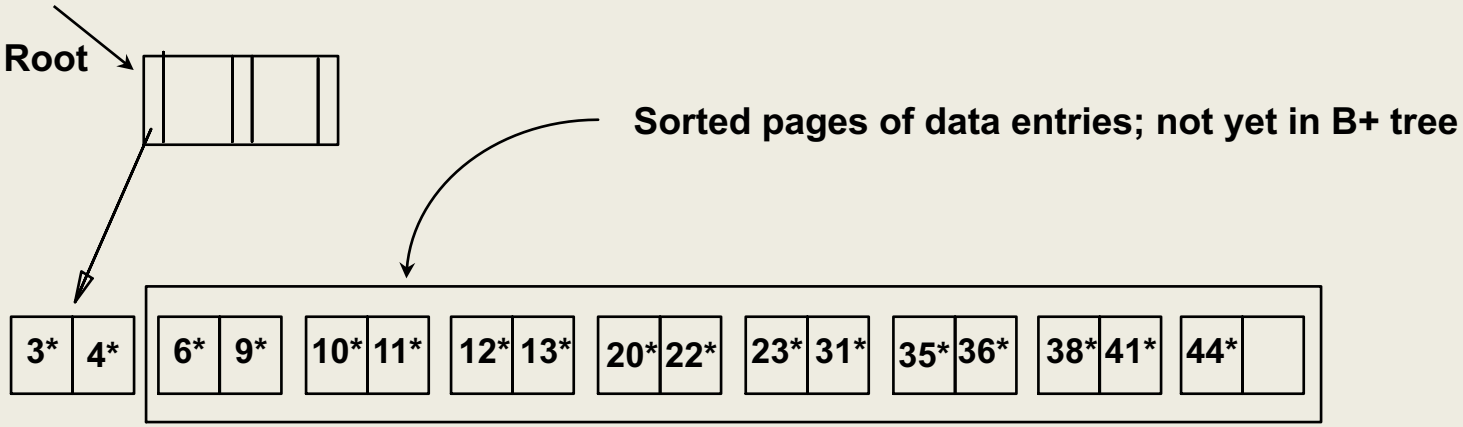
Prefix Key Compression

- Important to increase fan-out. (Why?)
- Key values in index entries only 'direct traffic'; can often compress them.
  - E.g., If we have adjacent index entries with search key values *Dannon Yogurt*, *David Smith* and *Devarakonda Murthy*, we can abbreviate *David Smith* to *Dav*. (The other keys can be compressed too ...)
    - Is this correct? Not quite! What if there is a data entry *Davey Jones*? (Can only compress *David Smith* to *Davi*)
    - In general, while compressing, must leave each index entry greater than every key value (in any subtree) to its left.
- Insert/delete must be suitably modified.



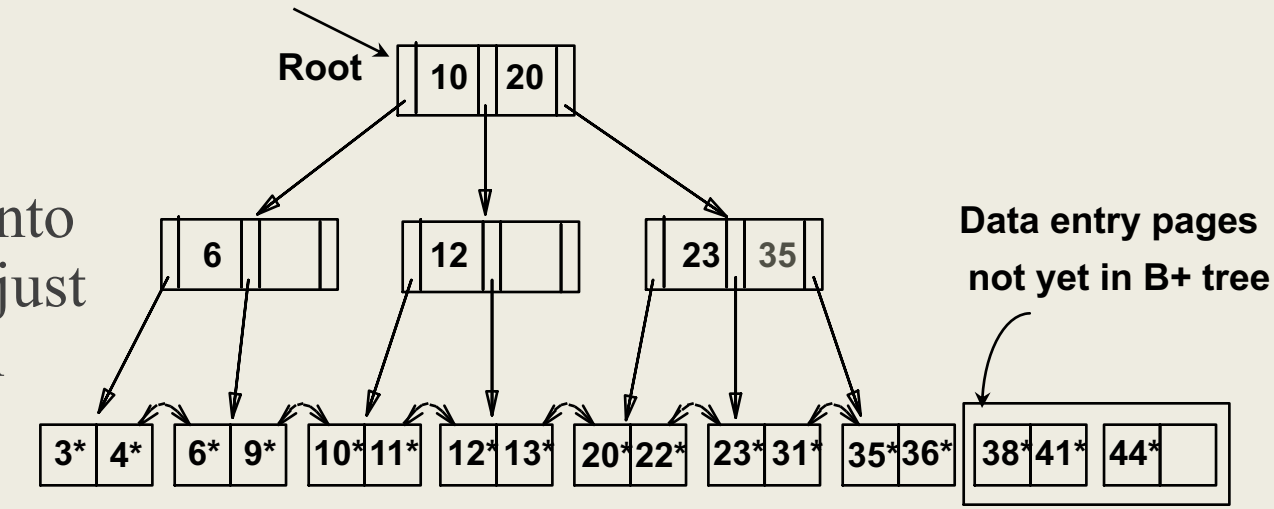
Bulk Loading of a B+ Tree

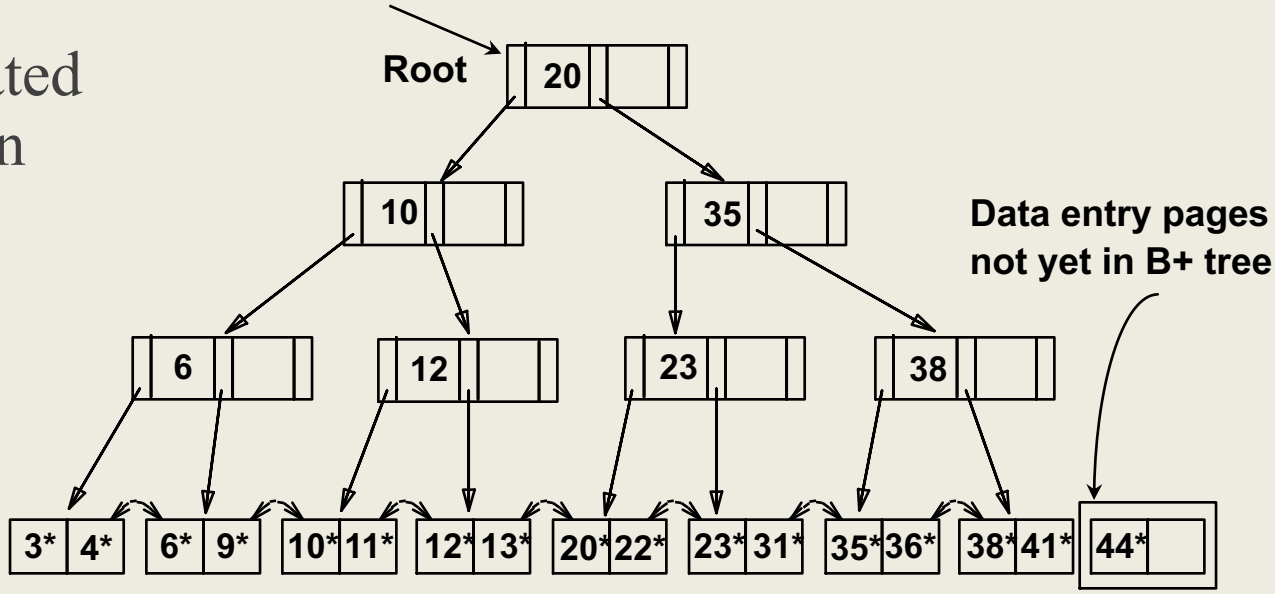
- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
- ***Bulk Loading*** can be done much more efficiently.
- *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)

Much faster than repeated inserts, especially when one considers locking!





Summary of Bulk Loading

Option 1: multiple inserts.

- Slow.
- Does not give sequential storage of leaves.

Option 2: ***Bulk Loading***

- Has advantages for concurrency control.
- Fewer I/Os during build.
- Leaves will be stored sequentially (and linked, of course).
- Can control ‘fill factor’ on pages.

A Note on ‘Order’

Order (**p**) concept replaced by physical space criterion in practice (‘*at least half-full’*).

- Index pages can typically hold many more entries than leaf pages.
- Variable sized records and search keys mean differnt nodes will contain different numbers of entries.
- Even with fixed length fields, multiple records with the same search key value (*duplicates*) can lead to variable-sized data entries (if we use Alternative (3)).

Summary of Tree-structured Indexes

Tree-structured indexes are ideal for range-searches, also good for equality searches.

ISAM is a static structure.

- Only leaf pages modified; overflow pages needed.
- Overflow chains can degrade performance unless size of data set and data distribution stay constant.

B+ tree is a dynamic structure.

- Inserts/deletes leave tree height-balanced; $\log_F N$ cost.
- High fanout (**F**) means depth rarely more than 3 or 4.
- Almost always better than maintaining a sorted file.

Summary of Tree-structured Indexes

- Typically, **67%** occupancy on average.
- Usually preferable to ISAM, modulo ***locking*** considerations; adjusts to growth gracefully.
- If data entries are data records, splits can change rids!

➢Key compression increases fanout, reduces height.

➢Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set.

➢Most widely used index in database management systems because of its versatility. One of the most optimized components of a DBMS.

Hash-Based Indexes

Hash-based indexes are best for ***equality selections***. ***Cannot*** support range searches.

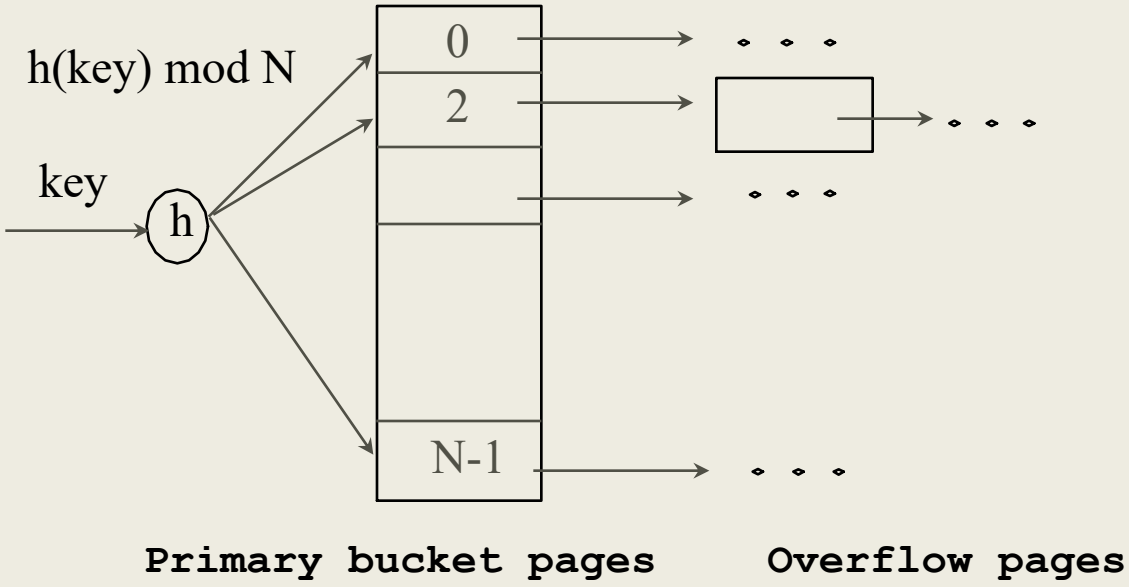
Static and dynamic hashing techniques exist;

trade-offs similar to ISAM vs. B+ trees.

Static Hashing

primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.

h(*k*) mod M = bucket to which data entry with key *k* belongs. (M = # of buckets)



Static Hashing (Contd.)

Buckets contain *data entries*.

Hash fn works on *search key* field of record *r*. Must distribute values over range 0 ... M-1.

- **h**(*key*) = (a * *key* + b) usually works well.
- a and b are constants; lots known about how to tune **h**.

Long overflow chains can develop and degrade performance.

- ***Extendible*** and ***Linear Hashing***: Dynamic techniques to fix this problem.

Extendible Hashing

Situation: Bucket (primary page) becomes full. Why not re-organize file by *doubling* # of buckets?

- Reading and writing all pages is expensive!
- *Idea*: Use ***directory of pointers to buckets***, double # of buckets by *doubling the directory,* splitting just the bucket that overflowed!
- Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split. ***No overflow page!***
- Trick lies in how hash function is adjusted!

Example

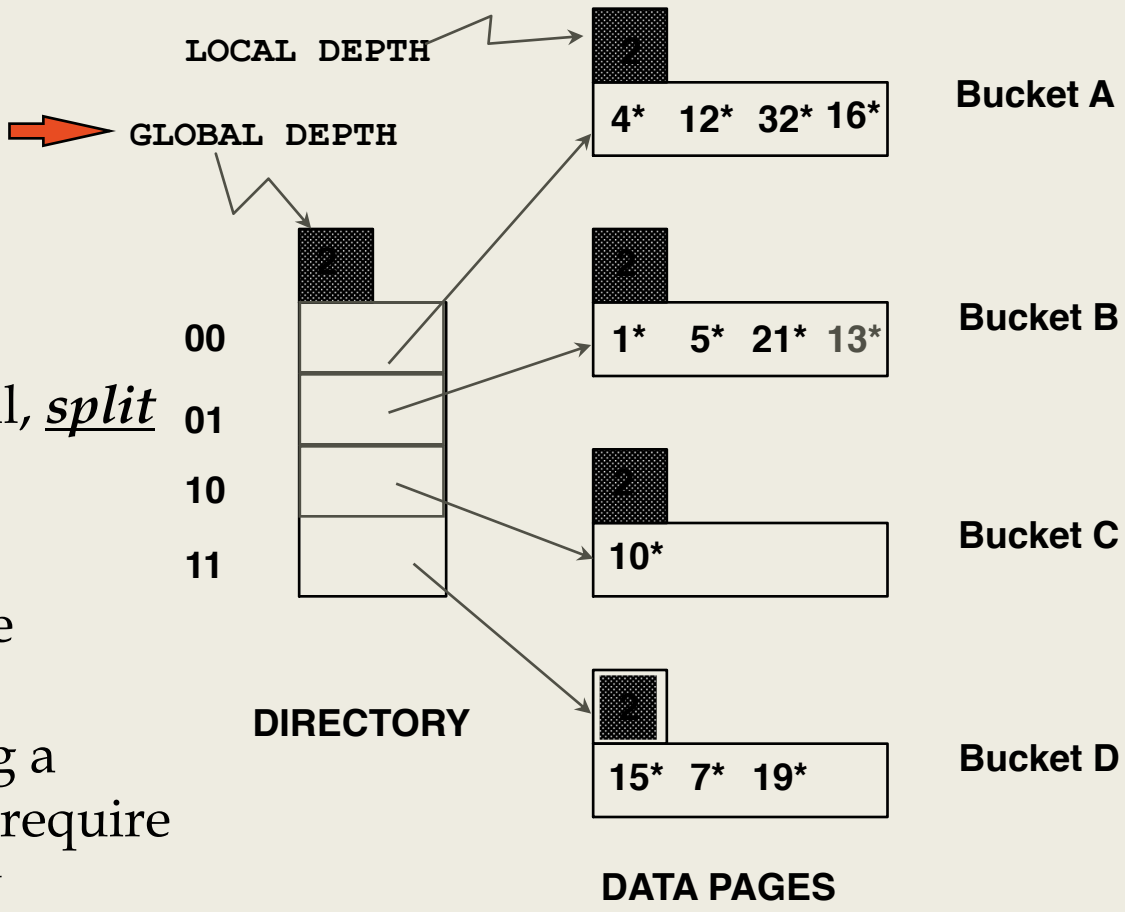
Directory is array of size 4.

To find bucket for *r*, take last `***global depth***' # bits of **h**(*r*); we denote *r* by **h**(*r*).

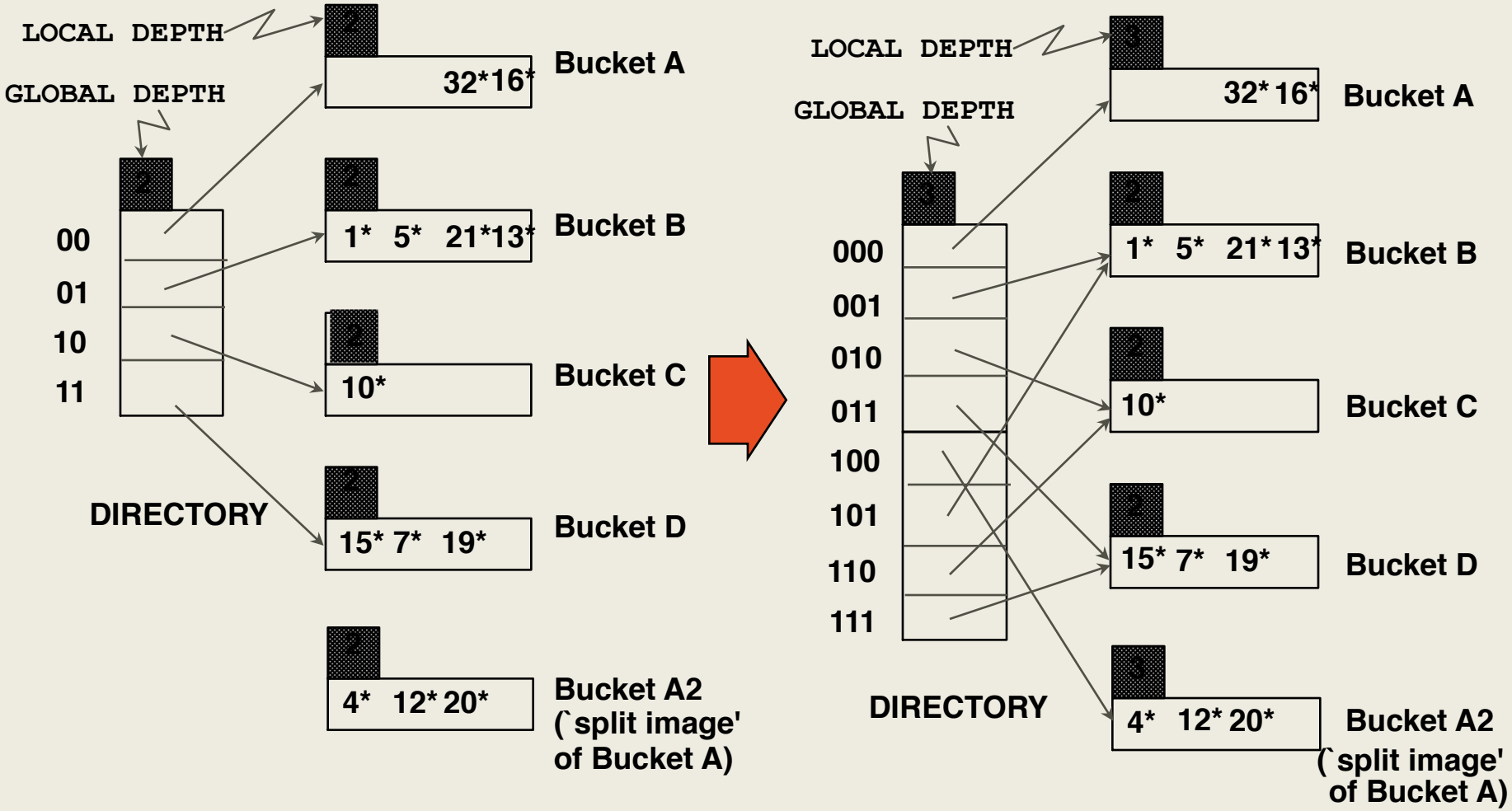
- If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.

❖ **Insert**: If bucket is full, ***split*** it (*allocate new page, re-distribute*).

❖ *If necessary*, double the directory.
(As we will see, splitting a bucket does not always require doubling; we can tell by comparing ***global depth*** with ***local depth*** for the split bucket.)



Insert **h**(r)=20 (Causes Doubling)



Points to Note

20 = binary 10100. Last **2** bits (00) tell us *r* belongs in A or A2. Last **3** bits needed to tell which.

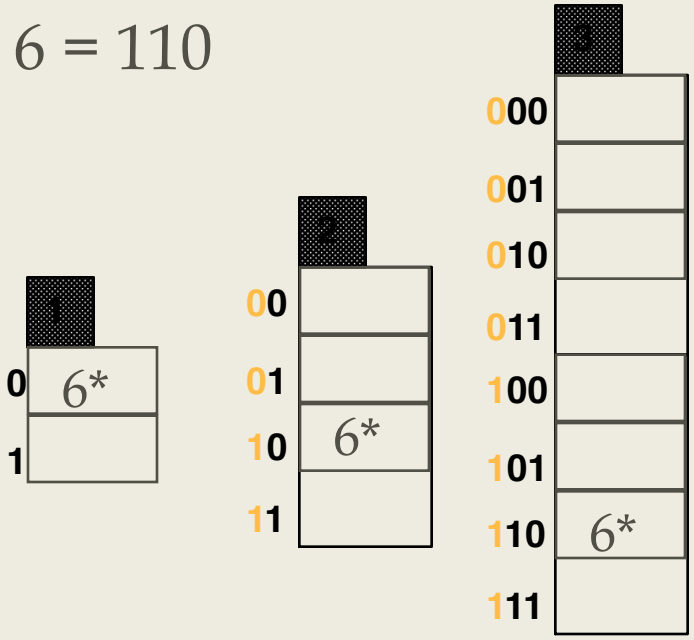
- ***Global depth of directory*:** Max # of bits needed to tell which bucket an entry belongs to.
- ***Local depth of a bucket*:** # of bits used to determine if an entry belongs to this bucket.

When does bucket split cause directory doubling?

- Before insert, *local depth* of bucket = *global depth*. Insert causes *local depth* to become > *global depth*; directory is doubled by ***copying it over*** and `fixing' pointer to split image page. (Use of least significant bits enables efficient doubling via copying of directory!)

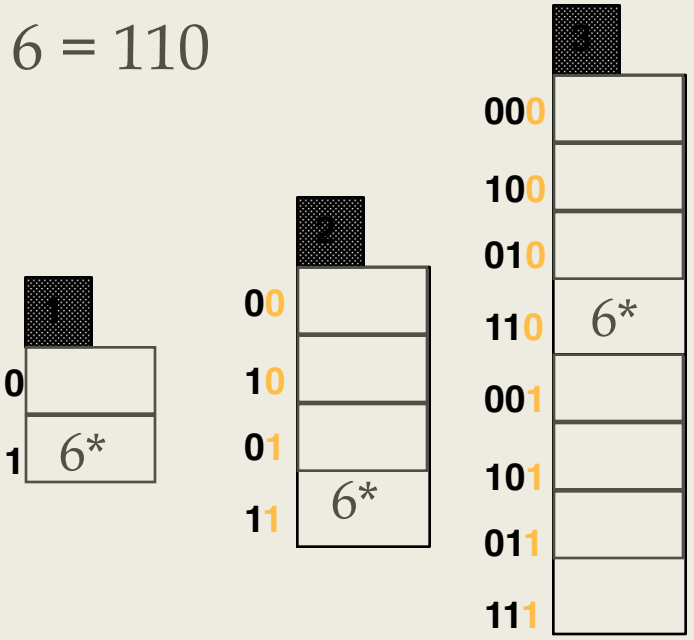
Directory Doubling

Why use least significant bits in directory?
⇔ Allows for doubling via copying!



Least Significant

vs.



Most Significant

Comments on Extendible Hashing

If directory fits in memory, equality search answered with one disk access; else two.

- 100MB file, 100 bytes/rec, 4K pages contains 1,000,000 records (as data entries) and 25,000 directory elements; chances are high that directory will fit in memory.
- Directory grows in spurts, and, if the distribution *of hash values* is skewed, directory can grow large.
- Multiple entries with same hash value cause problems!

Delete: If removal of data entry makes bucket empty, can be merged with `split image'. If each directory element points to same bucket as its split image, can halve directory.

Linear Hashing

This is another dynamic hashing scheme, an alternative to Extendible Hashing.

LH handles the problem of long overflow chains without using a directory, and handles duplicates.

Idea: Use a family of hash functions $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, ...

- $\mathbf{h_i(}$***key***$\mathbf{) = h(}$***key***$\mathbf{) \bmod (2^i N)}$; N = initial # buckets
- **h** is some hash function (range is *not* 0 to N-1)
- If $N = 2^{d0}$, for some *d0*, $\mathbf{h}_i$ consists of applying **h** and looking at the last *di* bits, where $di = d0 + i$.
- $\mathbf{h}_{i+1}$ doubles the range of $\mathbf{h}_i$ (similar to directory doubling)

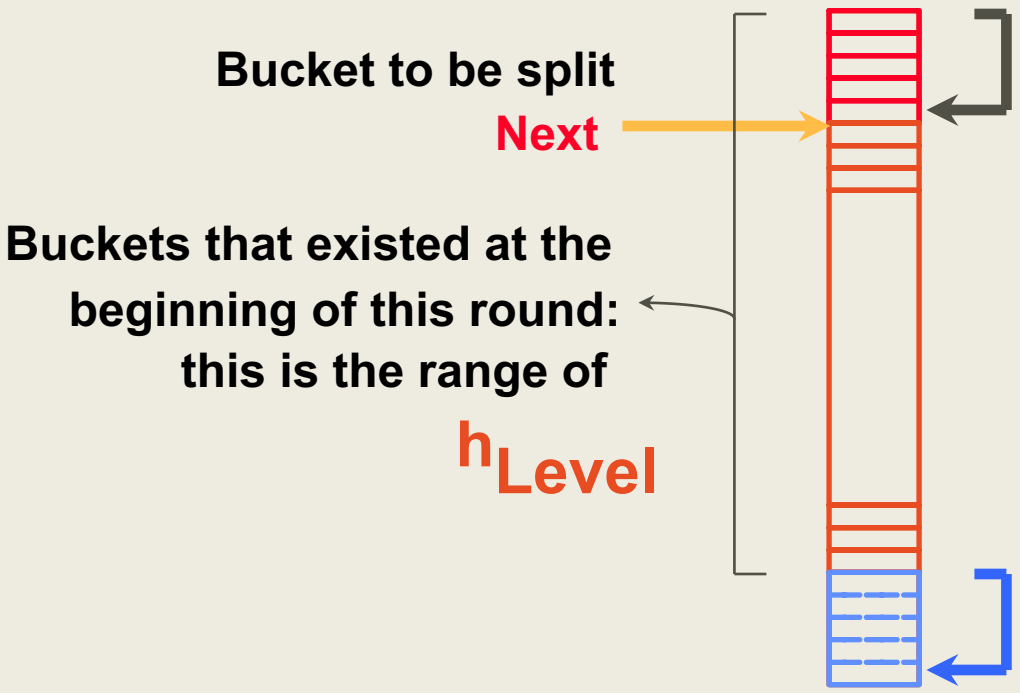
Linear Hashing (Contd.)

Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin.

- **Splitting proceeds in `rounds'.** Round ends when all N_R initial (for round R) buckets are split. Buckets 0 to *Next-1* have been split; *Next* to N_R yet to be split.
- **Current round number is *Level*.**
- **Search:** To find bucket for data entry *r,* find $\mathbf{h}_{Level}(r)$:
  - If $\mathbf{h}_{Level}(r)$ in range `*Next* to N_R', *r* belongs here.
  - Else, r could belong to bucket $\mathbf{h}_{Level}(r)$ or bucket $\mathbf{h}_{Level}(r) + N_R$; must apply $\mathbf{h}_{Level+1}(r)$ to find out.

Overview of LH File

In the middle of a round.



Buckets split in this round:
If h_{Level} (search key value)
is in this range, must use
$h_{Level+1}$ (search key value)
to decide if entry is in
`split image' bucket.

`split image' buckets:
created (through splitting
of other buckets) in this round

Linear Hashing (Contd.)

Insert: Find bucket by applying $\mathbf{h}_{Level}$ / $\mathbf{h}_{Level+1}$:

- If bucket to insert into is full:
  - Add overflow page and insert data entry.
  - (*Maybe*) Split *Next* bucket and increment *Next*.

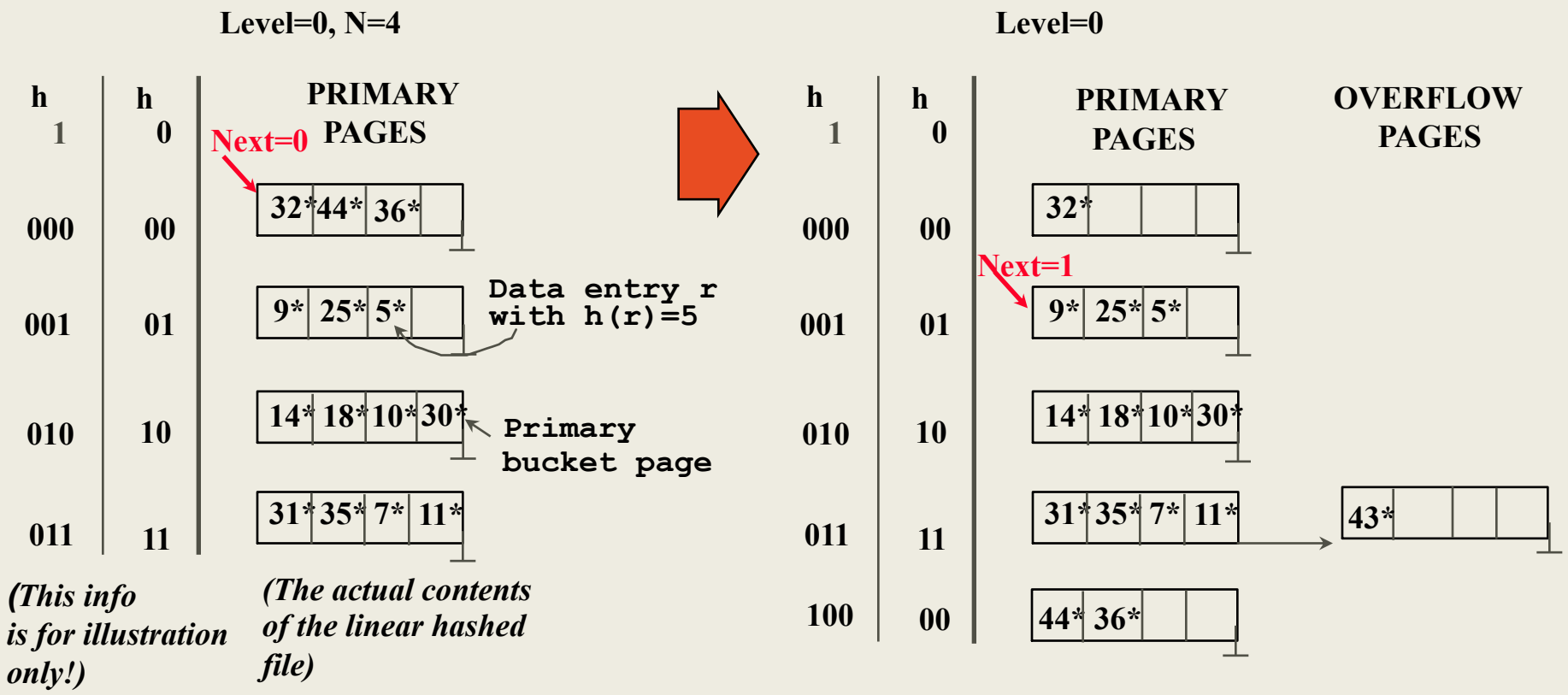
Can choose any criterion to `trigger' split.

Since buckets are split round-robin, long overflow chains don't develop!

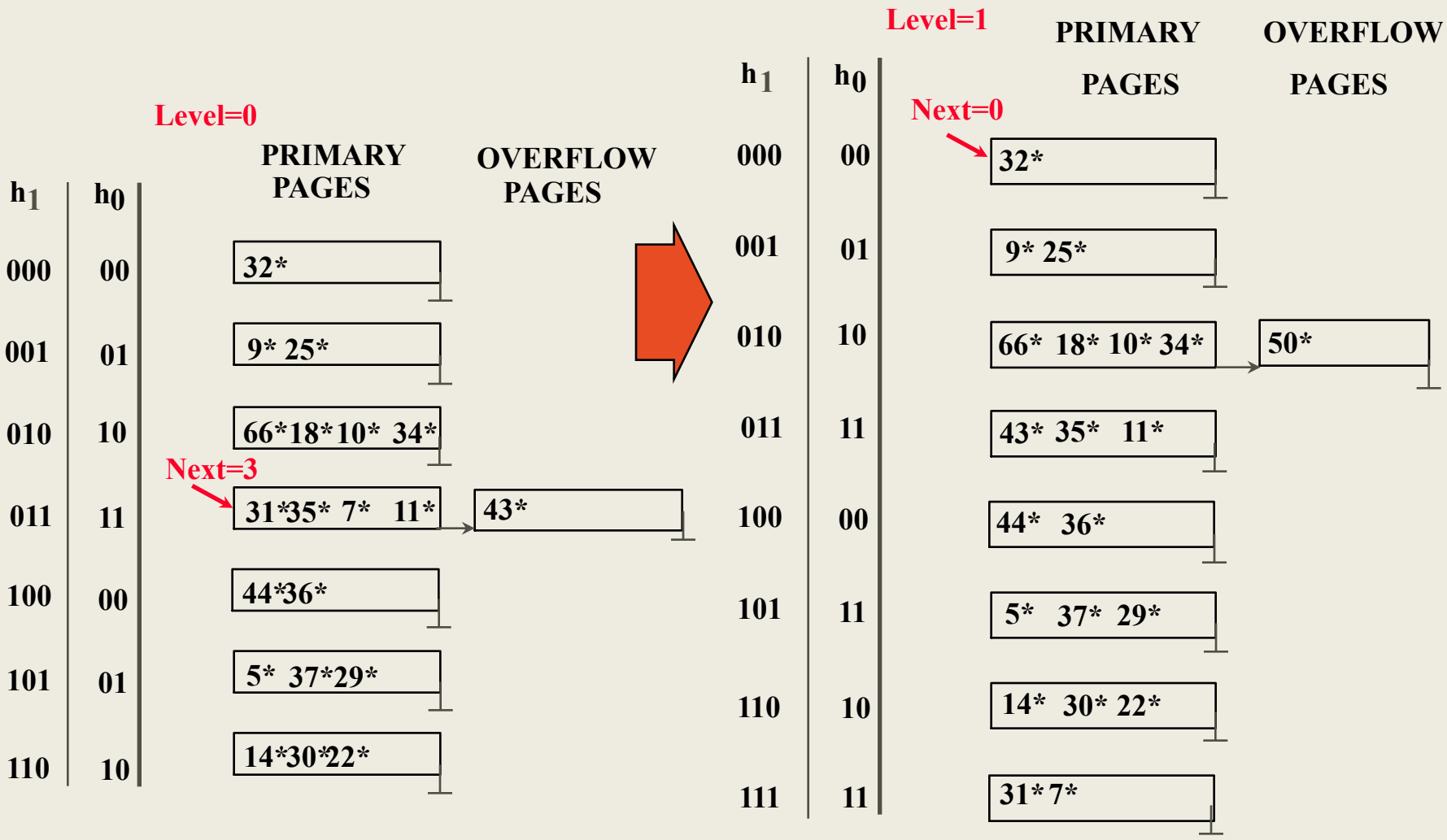
Doubling of directory in Extendible Hashing is similar; switching of hash functions is *implicit* in how the # of bits examined is increased.

Example of Linear Hashing

On **split, $h_{Level+1}$** is used to **re-distribute** entries.



Example: End of a Round



LH Described as a Variant of EH

The two schemes are actually quite similar:

- Begin with an EH index where directory has *N* elements.
- Use overflow pages, split buckets round-robin.
- First split is at bucket 0. (Imagine directory being doubled at this point.) But elements <1,*N*+1>, <2,*N*+2>, ... are the same. So, need only create directory element *N*, which differs from 0, now.
- When bucket 1 splits, create directory element *N*+1, etc.

So, directory can double gradually. Also, primary bucket pages are created in order. If they are *allocated* in sequence too (so that finding i'th is easy), we actually don't need a directory! Voila, LH.

Summary of Hash-based Indexes

Hash-based indexes: best for equality searches, cannot support range searches.

Static Hashing can lead to long overflow chains.

Extendible Hashing avoids overflow pages by splitting a full bucket when a new data entry is to be added to it. ***(Duplicates may require overflow pages.)***

- Directory to keep track of buckets, doubles periodically.
- Can get large with skewed data; additional I/O if this does not fit in main memory.

Summary of Hash-based Indexes

Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages.

- Overflow pages not likely to be long.
- Duplicates handled easily.
- Space utilization could be lower than Extendible Hashing, since splits not concentrated on `dense' data areas.
  - ❖Can tune criterion for triggering splits to trade-off slightly longer chains for better space utilization.

For hash-based indexes, a *skewed* data distribution is one in which the *hash values* of data entries are not uniformly distributed!