Dynamic Hash Indexes & Tree-Structured Indexes

Kathleen Durant PhD
Northeastern University
Lecture 16

Index Concept

- Main idea: ***A separate data structure used to locate records***
- Most generally, index is a list of value/address pairs
  - Each pair is an index “entry”
  - Value is the index “key”
  - Address will point to a data record, or to a data page
    - There might be many records on a page
  - The assumption is that the value/address pair will be much smaller in size than the full record
- If index is small, a copy can be maintained in memory
  - Permanent disk copy is still needed

Indexing Pitfalls

- Index itself is a file
  - Occupies disk space
  - Must worry about maintenance, consistency, recovery, etc.
- Large indices won't fit in memory
  - May require multiple seeks to locate record entry

Essential for Multilevel Indexes

- Should support efficient random access
  - Should also support efficient sequential access, if possible
- Should have low height
- Should be efficiently updatable
- Should be storage-efficient
- Top level(s) should fit in memory

Hashing Index

Hash index record

- *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**>

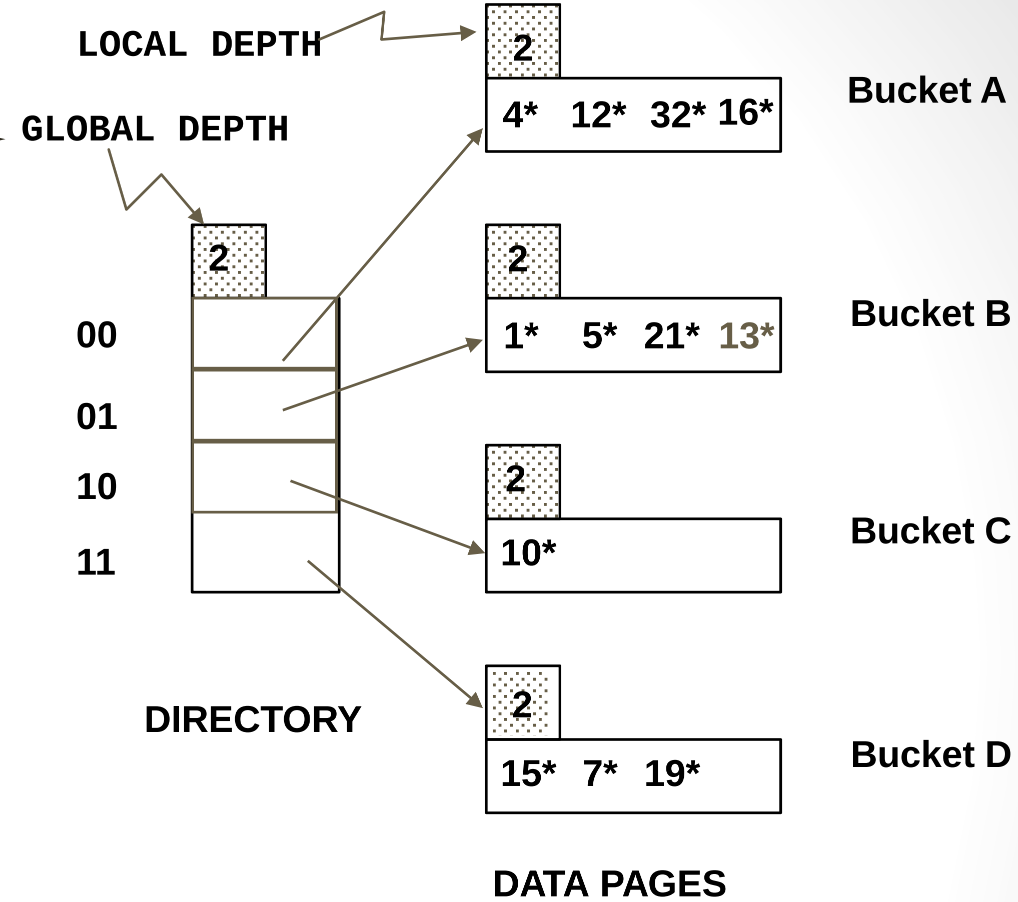
Hashing mechanism

- Your index is a collection of *buckets* (bucket = page)
- Define a hash function, *h*, that maps a key to a bucket.
- Store the corresponding data in that bucket.
- Collisions
  - Multiple keys hash to the same bucket.
  - Store multiple keys in the same bucket.
- What do you do when buckets fill?
  - Chaining: link new pages(overflow pages) off the bucket.

Extendible Hashing

- **Main Idea:** Use a directory of (logical) pointers to bucket pages
- 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

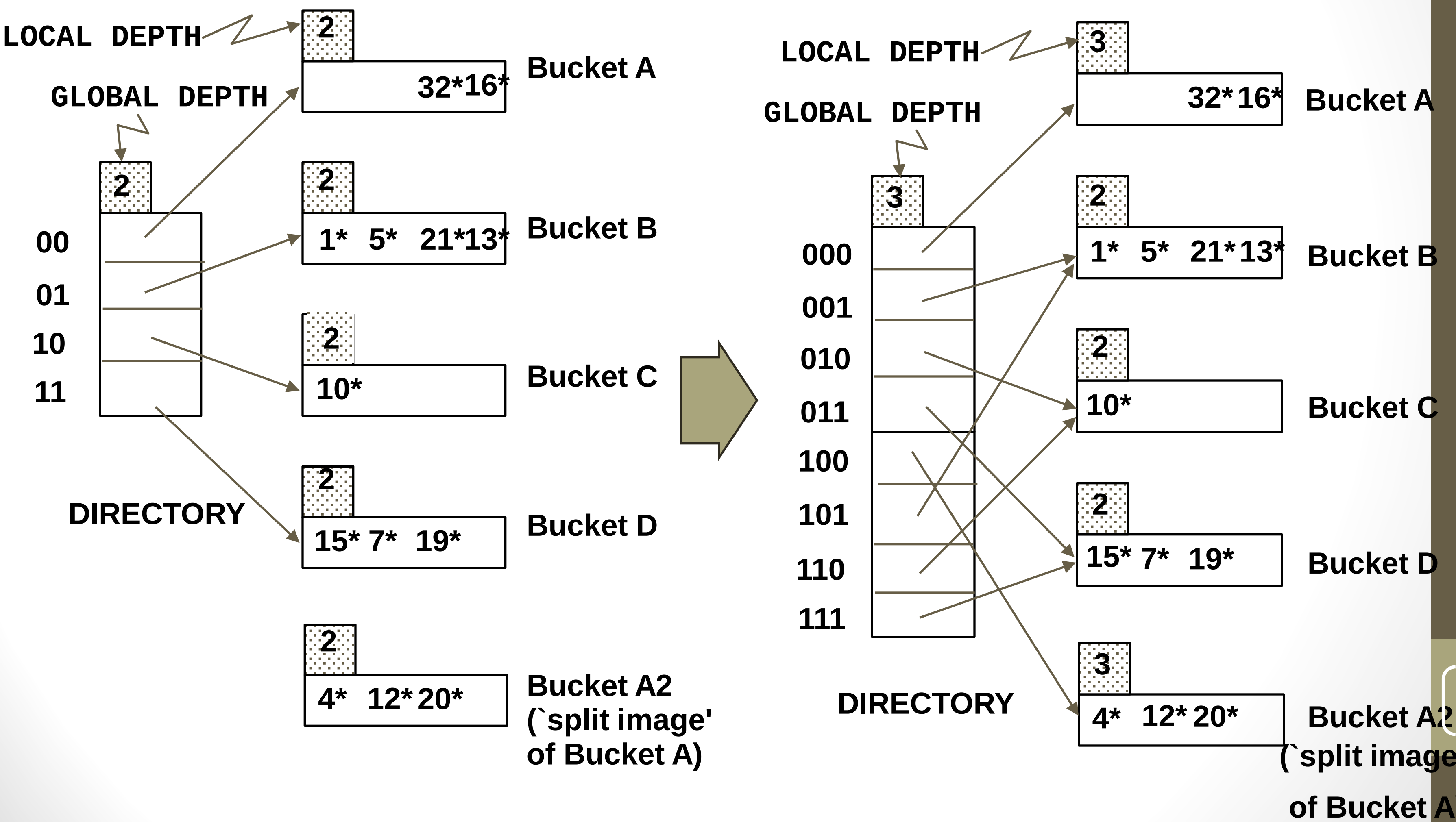


- Directory is array of size 4.
- To find bucket for *r*, take last \`*global depth*' # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.

❖ **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)

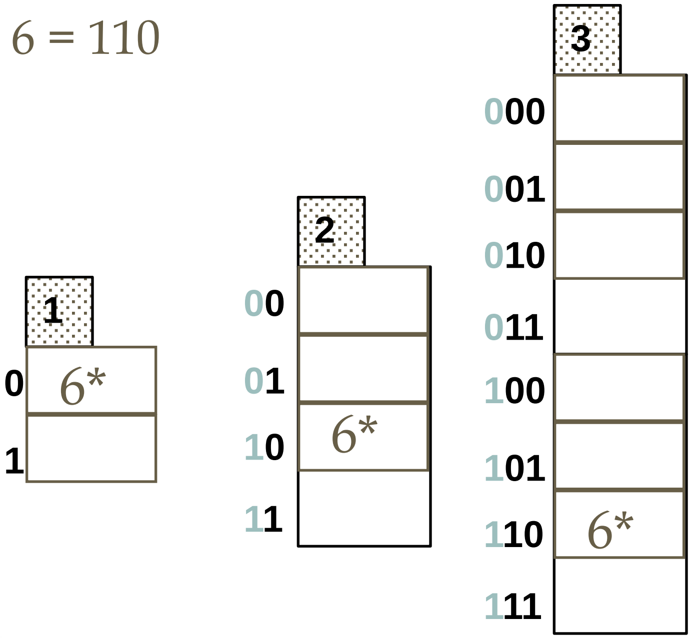


Points to Note

- 20 = binary 10100. Last **2** bits (00) tell us *r* belongs in A or A2. Last <u>**3**</u> 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!)

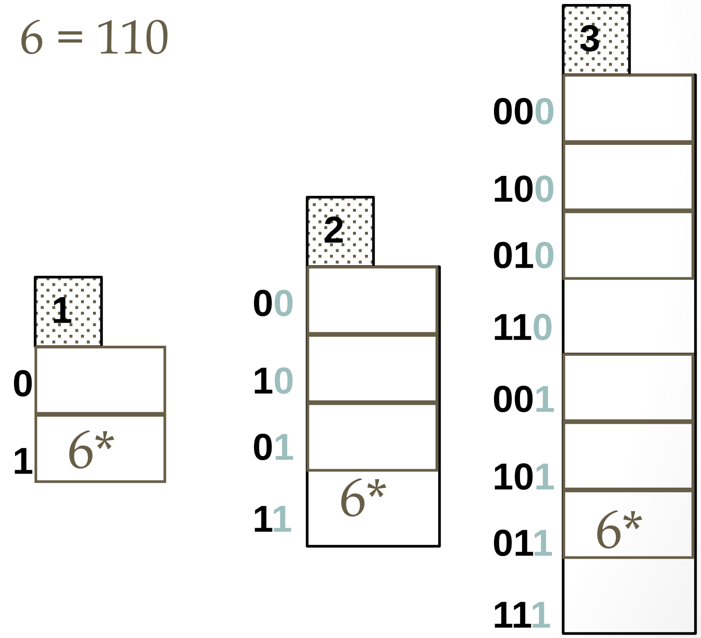
Directory Doubling

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
    - Need a decent hash function
- **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 $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, ...
  - $\mathbf{h}_i(key) = \mathbf{h}(key) \bmod (2^i N)$; 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)
- Duplicates extendible hash without the directory – since extendible hash always adds 1 bit to the bucket's address

Linear Hashing Details

- 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.

Extendible Hashing vs. Linear Hashing

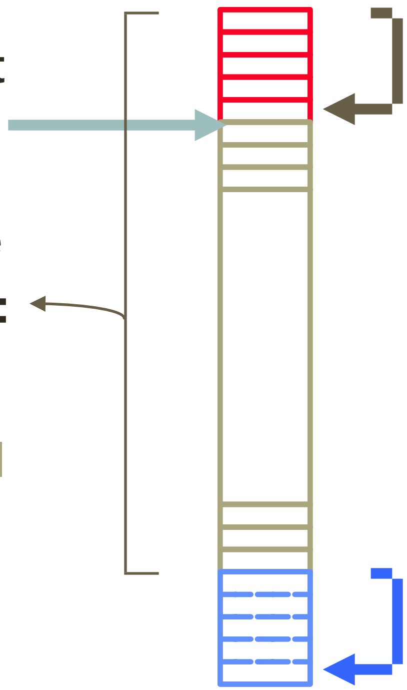
- Dynamic **Extendible Hashing**
  - Periodically double the size of the database directory.
  - Rehash every key.
- Dynamic **Linear Hashing** (Litwin)
  - Grow table one bucket at a time.
  - Split buckets sequentially; rehash just the splitting bucket.
  - Maintain overflow buckets as necessary.
  - Keep track of max bucket to identify the correct number of bits to consider in the hash value

Overview of LH File

- In the middle of a round.

Bucket to be split

Next



Buckets that existed at the beginning of this round: this is the range of h_{Level}

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

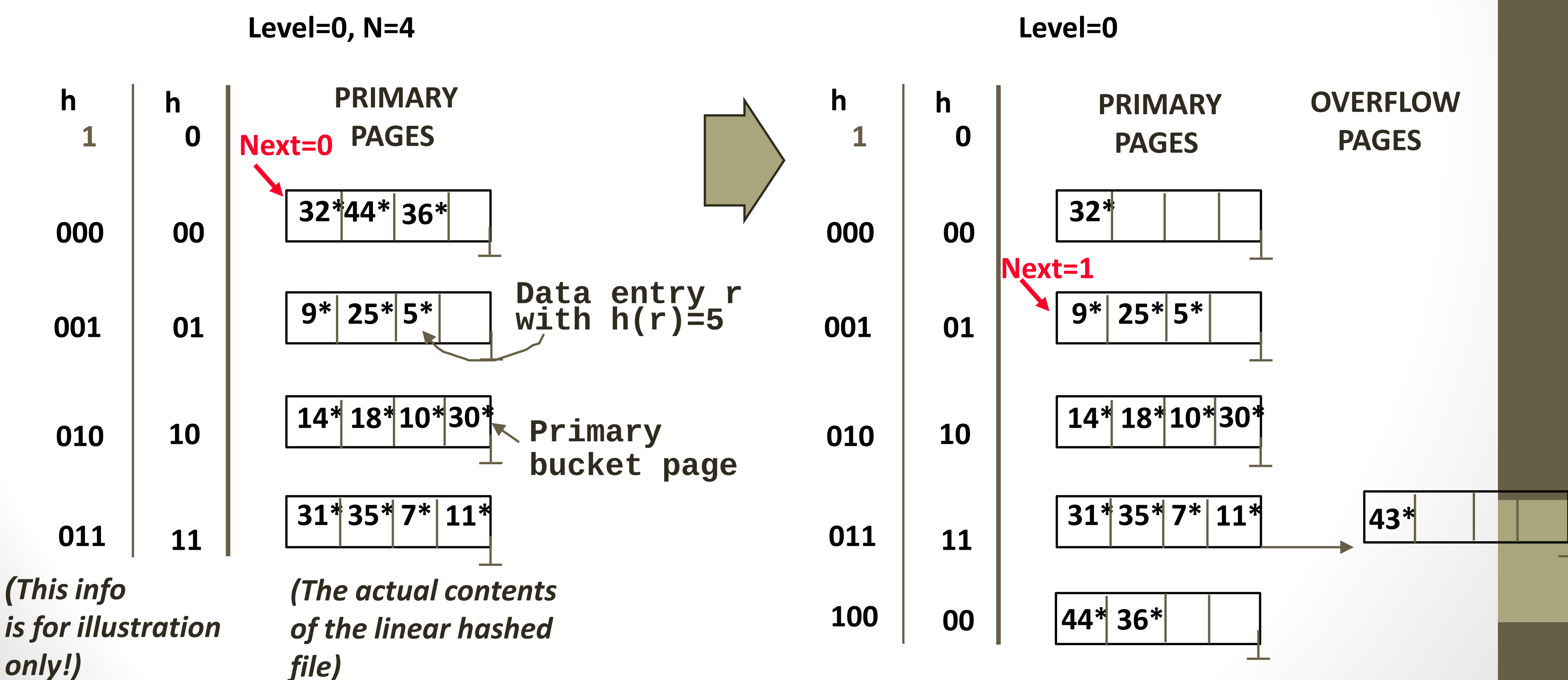
Linear Hashing (Contd.)

- 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.

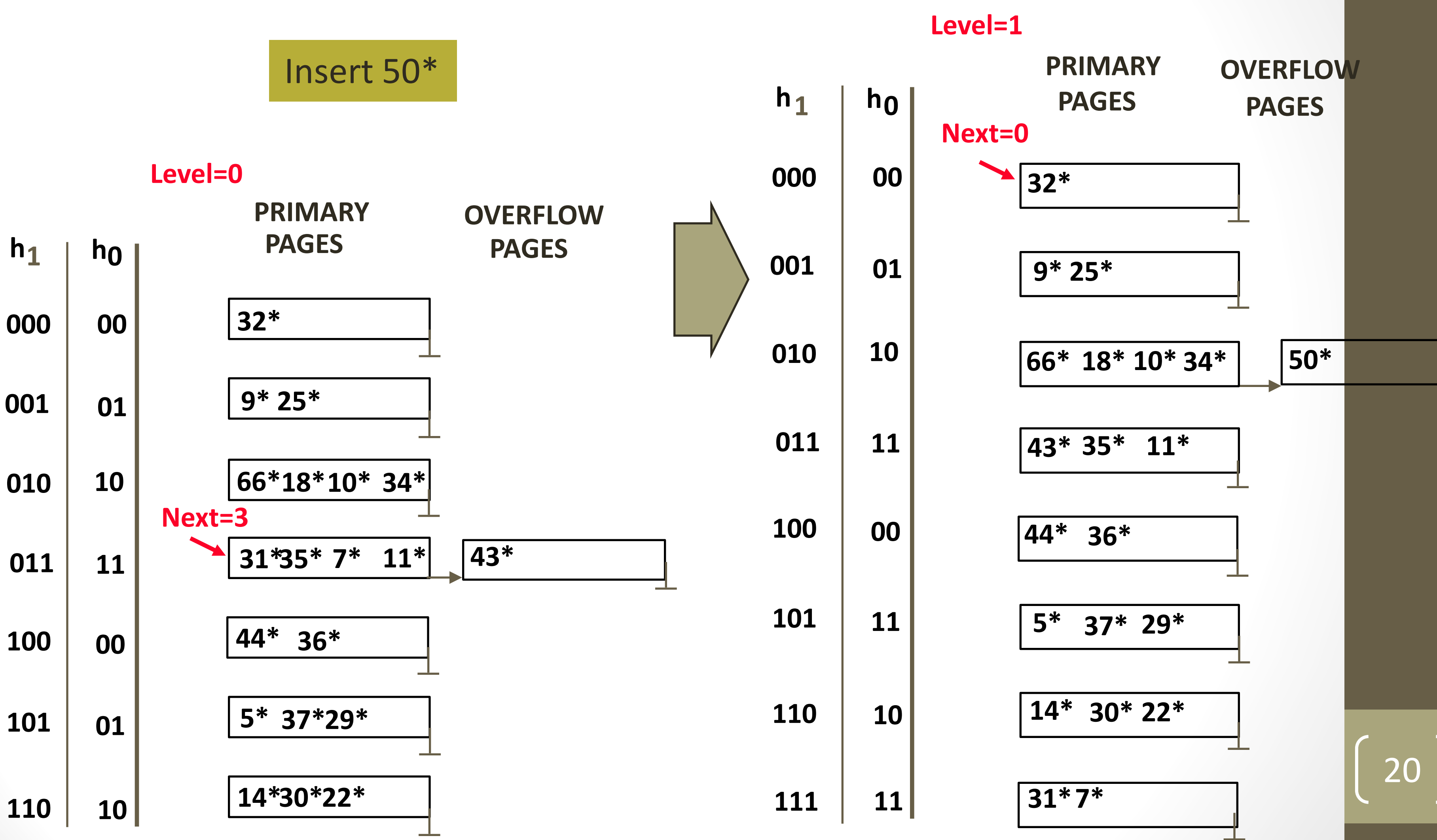
Example of Linear Hashing

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

Insert 43*



Example: End of a Round



Summary: 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: Linear hashing

- 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

Tree Structured Indexes

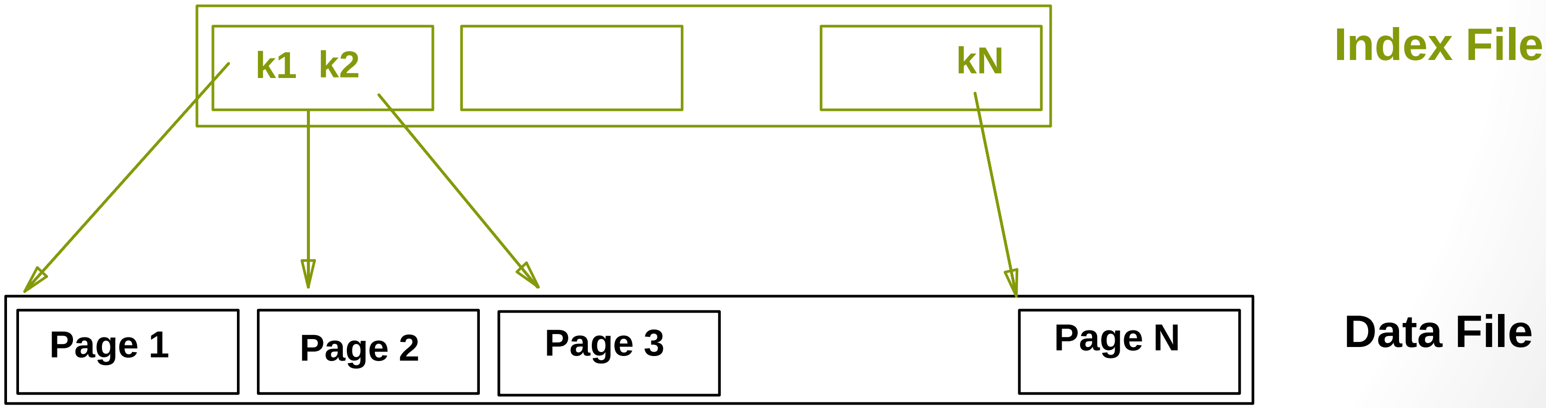
- Tree-structured indexing techniques support both *range searches* and *equality searches*.
- Tree structures with search keys on *value-based domains*
  - *ISAM*: static structure
  - *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.
- Tree structures with the search key on *multi-dimensional objects*
  - R-tree, R*-tree representation of spatial data

Introduction

- *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 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 a 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

- = Indexed Sequential Access Method
  - IBM terminology
  - “Indexed Sequential” more general term (non-IBM)
  - ISAM as described in textbook is very close to B+ tree
    - simpler versions exist
- Main idea: ***maintain sequential ordered file but give it an index***
  - Sequentiality for efficient “batch” processing
  - Index for random record access

ISAM Technique

- Build a dense index of the pages (1st level index)
  - Sparse from a record viewpoint
- Then build an index of the 1st level index (2nd level index)
- Continue recursively until top level index fits on 1 page
- Some implementations may stop after a fixed # of levels27

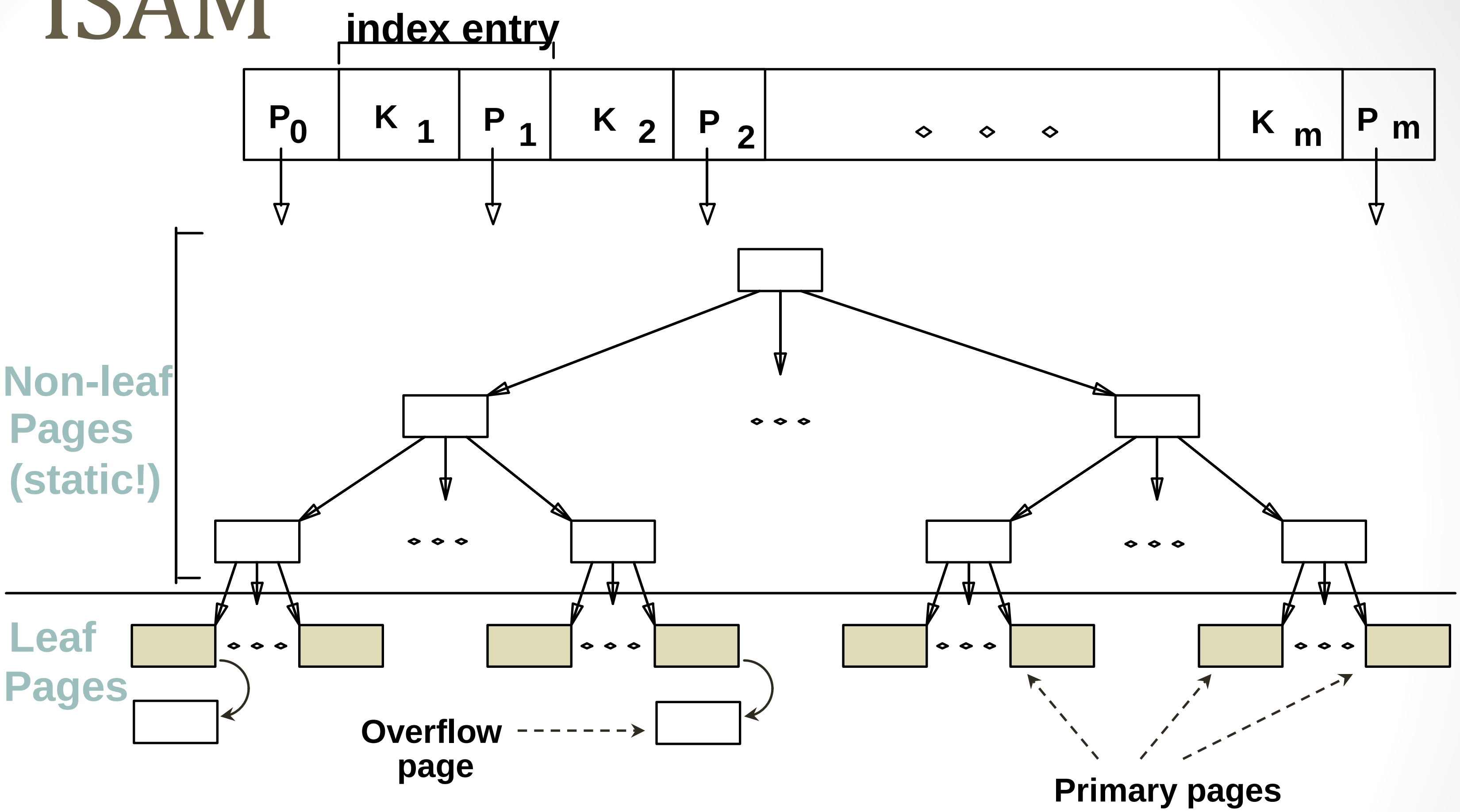
Updating an ISAM File

- Data set must be kept sequential
  - So that it can be processed without the index
  - May have to rewrite entire file to add records
  - Could use overflow pages
    - chained together or in fixed locations (overflow area)
- Index is usually NOT updated as records are added or deleted
- Once in a while the whole thing is "reorganized"
  - Data pages recopied to eliminate overflows
  - Index recreated

ISAM Pros, Cons

- Pro
  - Relatively simple
  - Great for true sequential access
- Cons
  - Not very dynamic
  - Inefficient if lots of overflow pages
  - Can only be one ISAM index per file

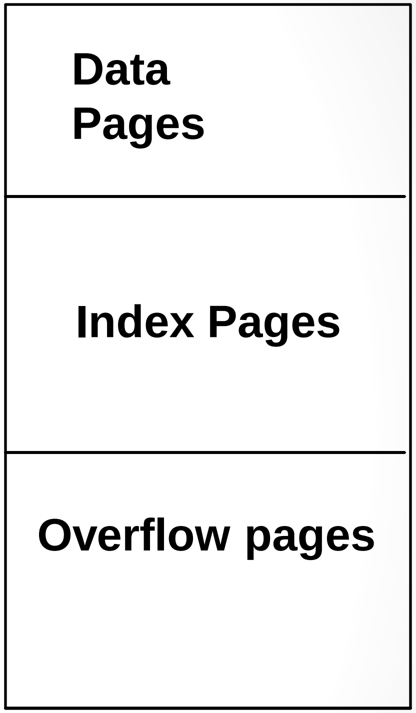
ISAM



- Leaf pages contain sorted data records (e.g., Alt 1 index).
- Non-leaf part directs searches to the data records; static once built
- Inserts/deletes: use overflow pages, bad for frequent inserts.

Comments on ISAM

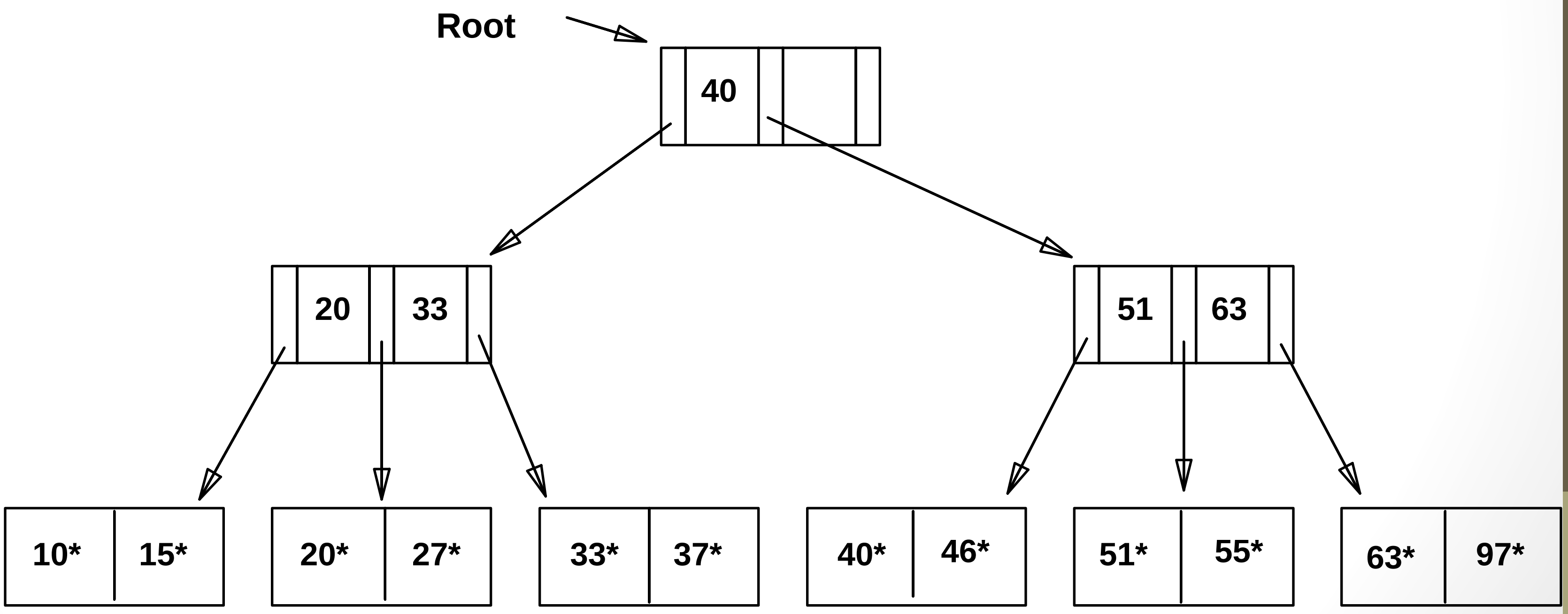
- *File creation*: Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages.
- *Index entries*: <search key value, page id>; they `direct' search for *data entries*, which are in leaf pages.
- <u>*Search*</u>: Start at root; use key comparisons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



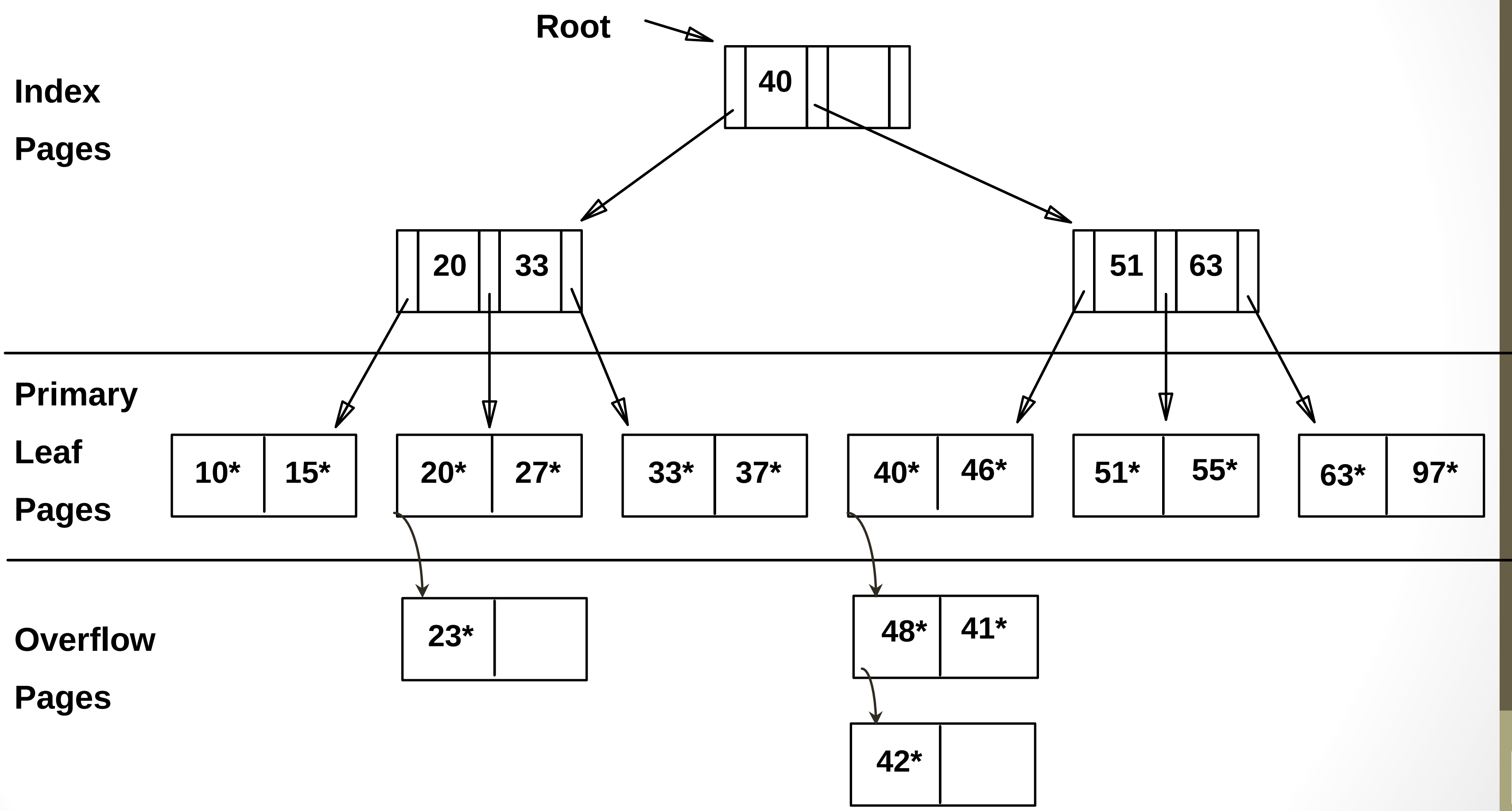
* **Static tree structure**: *inserts/deletes affect only leaf pages.*

Example ISAM Tree

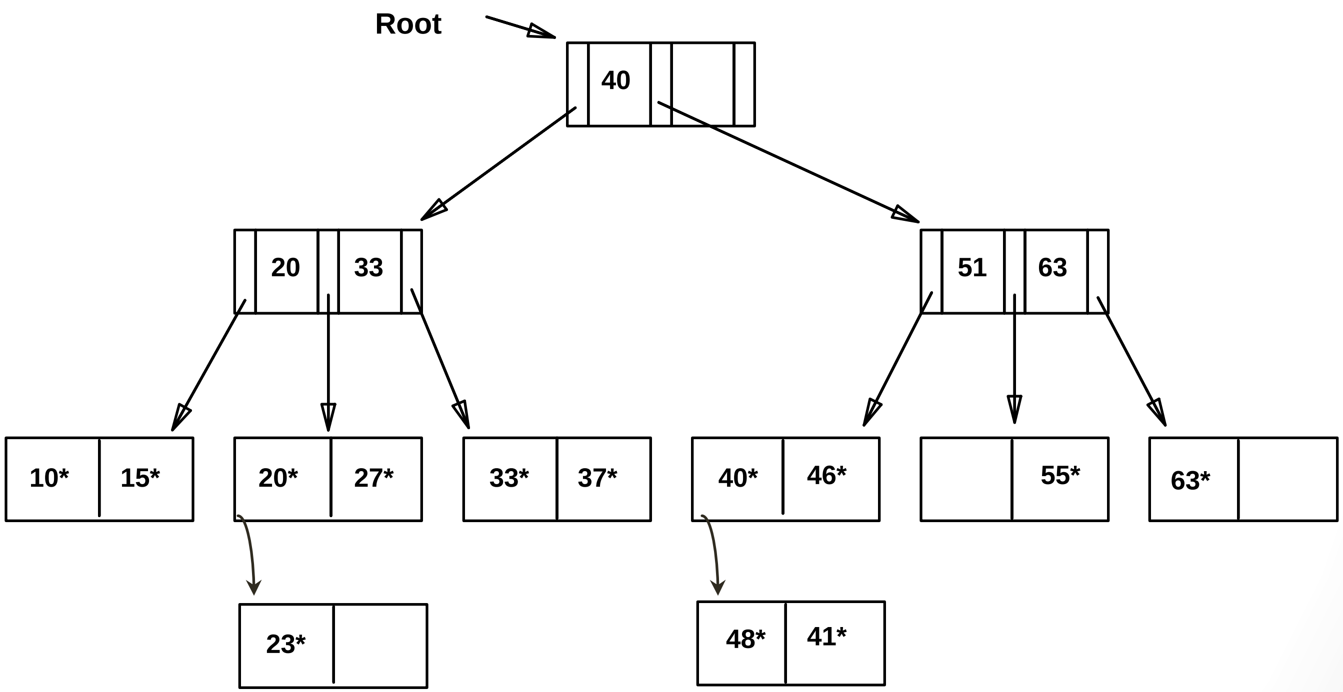
- Each node can hold 2 entries; no need for `next-leaf-page' pointers. (Why?)



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



… Then Deleting 42*, 51*, 97*



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

Comments on ISAM

- Main problem
  - *Long overflow chains* after many inserts, high I/O cost for retrieval.
- Advantages
  - Simple when updates are rare.
  - Leaf pages are allocated in sequence, leading to *sequential I/O*.
  - **Non-leaf pages are static; for *concurrent access*, no need to lock non-leaf pages**
- Good performance for frequent updates?

  B+tree!

B trees Introduction

- A B-tree is a keyed index structure, comparable to a number of memory resident keyed lookup structures
  - Balanced binary tree, AVL tree, and the 2-3 tree.
- Difference B-tree is meant to reside on disk
  - Can be partially memory-resident when entries in the structure are accessed.
- The B-tree structure is the most common used index type in databases today.
  - It is provided by **ORACLE, DB2**, and **INGRES**.

B-tree Organization

A B-tree helps minimize access to the index / directory

A B-tree is a tree where:

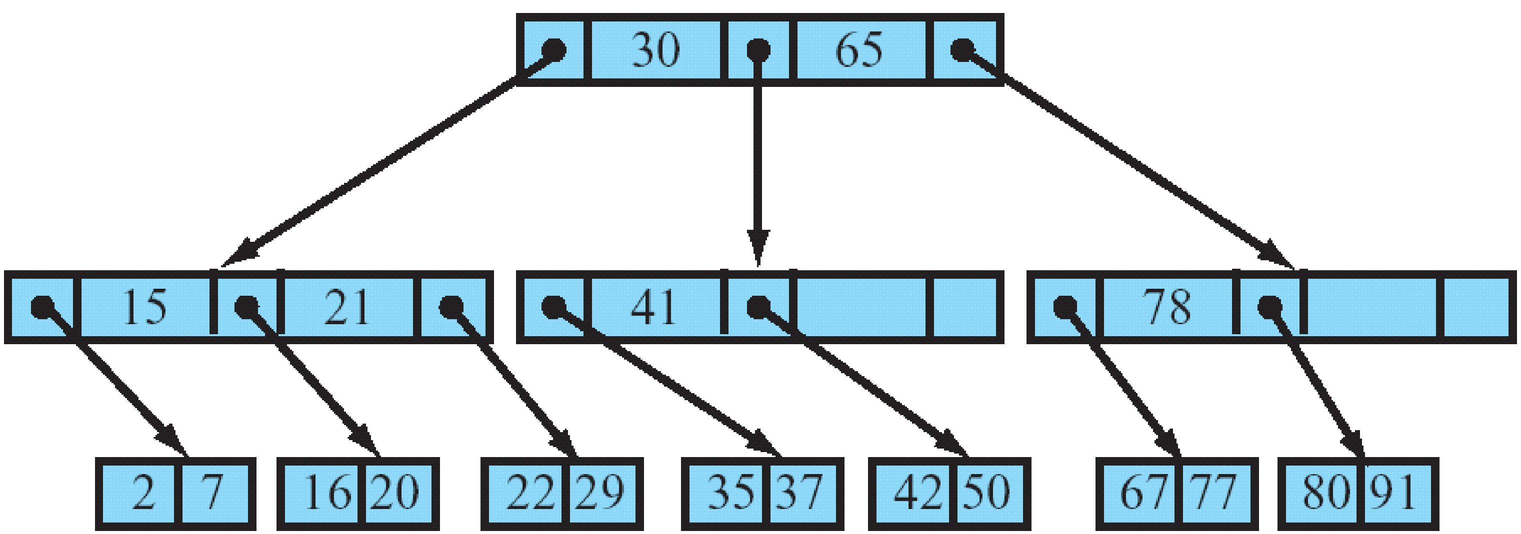
- Each node contains s slots for a index record and s + 1 pointers
- Each node is always at least ½ full

Order: the maximum number of keys in a non-leaf node

Fanout of a node: the number of pointers out of the node

- It is a type of Multi-way tree

Example B-Tree

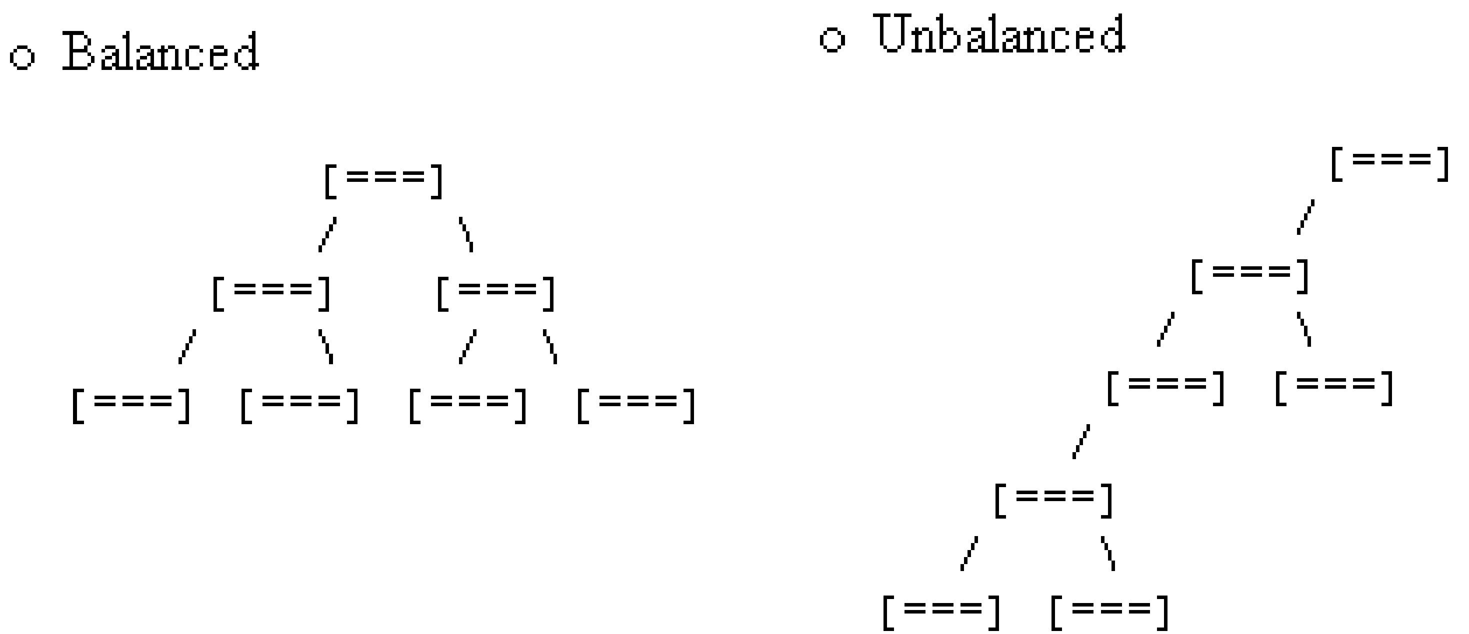


The B-Tree Shape

- A B-tree is built upside down with the root at the top and the leaves at the bottom.
- All nodes above the leaf level, including the root, are called **directory nodes** or **index nodes**.
- **Directory nodes** below the root are called **internal nodes**.
- The root node is known as **level 1** of the B-tree and successively lower levels are given successively larger level numbers with the leaf nodes at the lowest level.
- The total number of levels is called the **depth** of the B-tree.

•Balanced and Unbalanced Trees

Trees can be *balanced* or *unbalanced*.

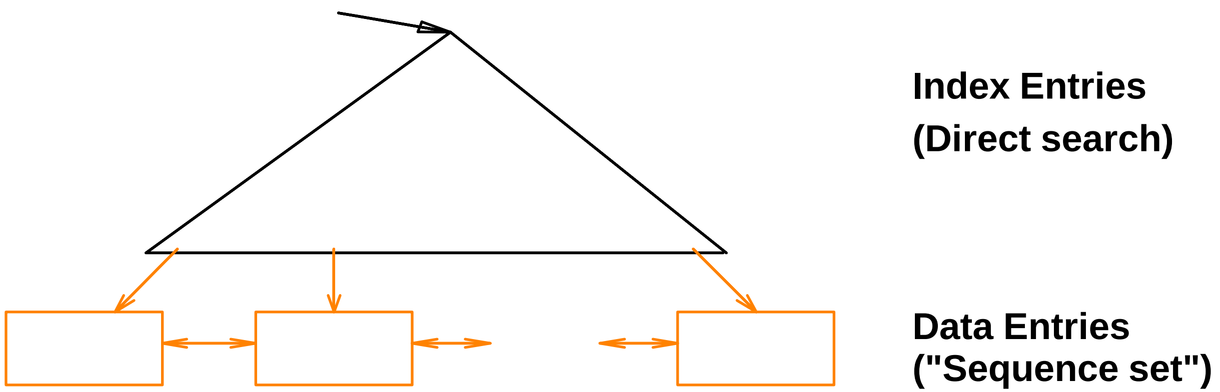


In a balanced tree, every path from the route to a leaf node is the same length.

A tree that is balanced has at most $\boldsymbol{log_{order}\ n}$ levels. This is desirable for an index.

B+ Tree: Most Widely Used Index

- Search for a record requires a tranversal from the root to the appropriate leaf
- Height-balanced given arbitrary inserts/deletes.
  - F = fanout, N = # leaf pages, Height = $\text{Log}_F N$.
- Minimum 50% occupancy (except for root).
  - Each non-root node contains [$\lceil n/2 \rceil$, n] entries, where n is the max # of keys in a node, called *order* of the tree.
  - Root node can have [1, n] entries.

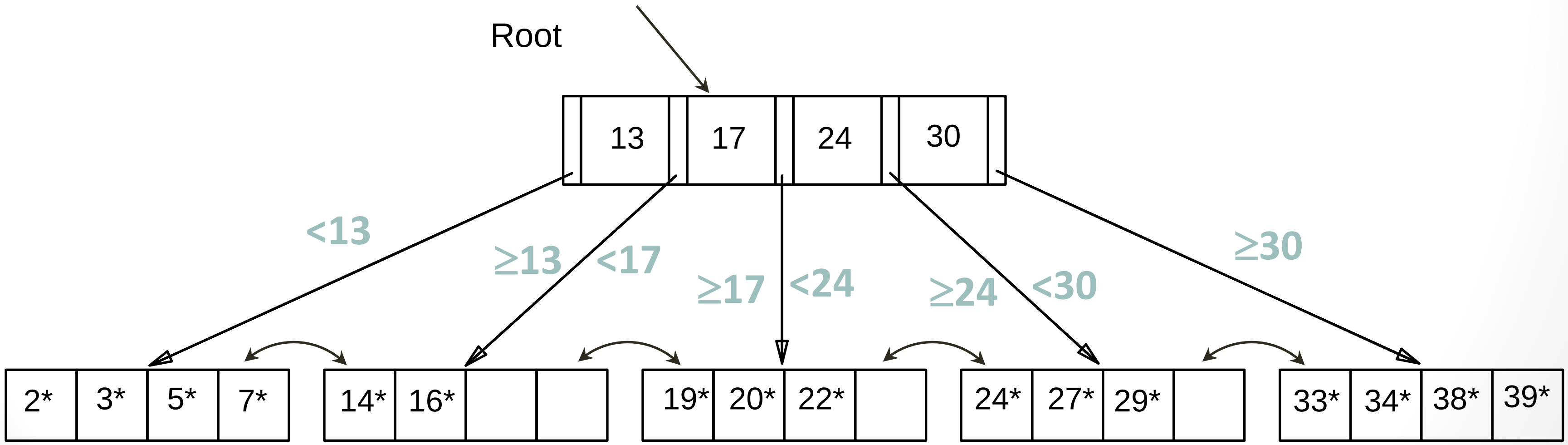


Definition of B+ tree

- A B-tree of order *n* is a height-balanced tree , where each node may have up to *n* children, and in which:
  - All leaves (leaf nodes) are on the same level
  - No node can contain more than *n* children
  - All nodes except the root have at least n/2 children
  - The root is either a leaf node, or it has at least n/2 children

Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...

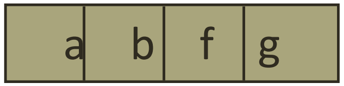


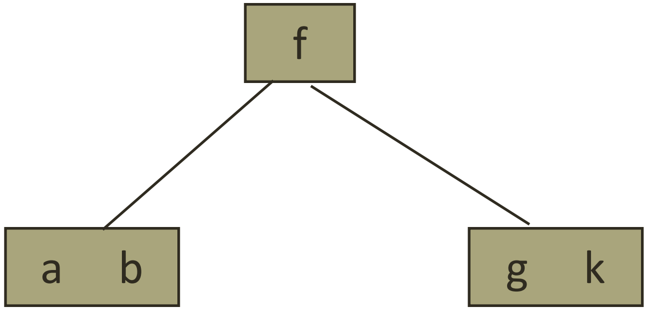
B+ Trees in Practice

- Typical order: 200. Typical fill-factor: 67%.
  - Average fan-out for internal nodes = 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

Insertion in B-Tree

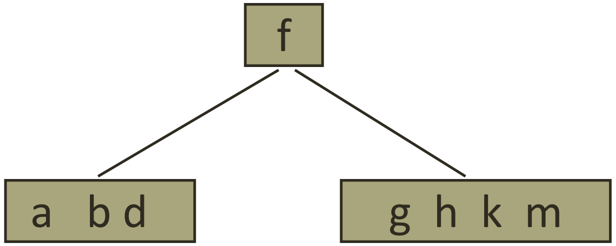
- 1. 2.
- a, g, f,b: k:





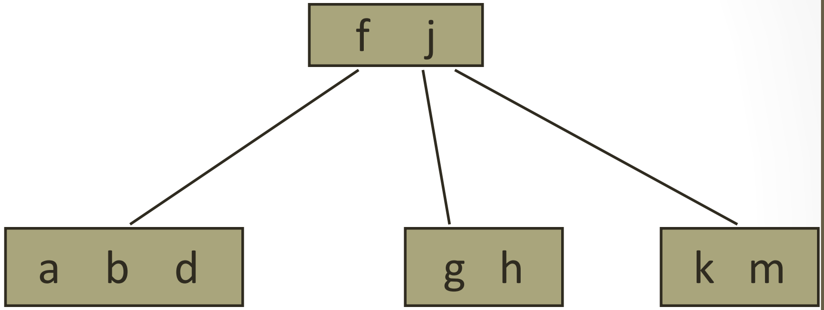
Insertion (cont.)

- 3.
- d, h, m:

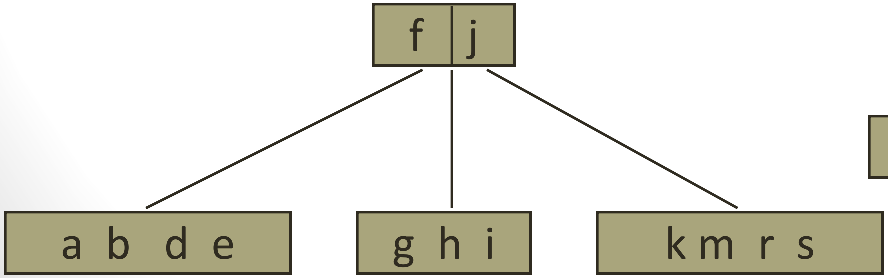


4.

j:

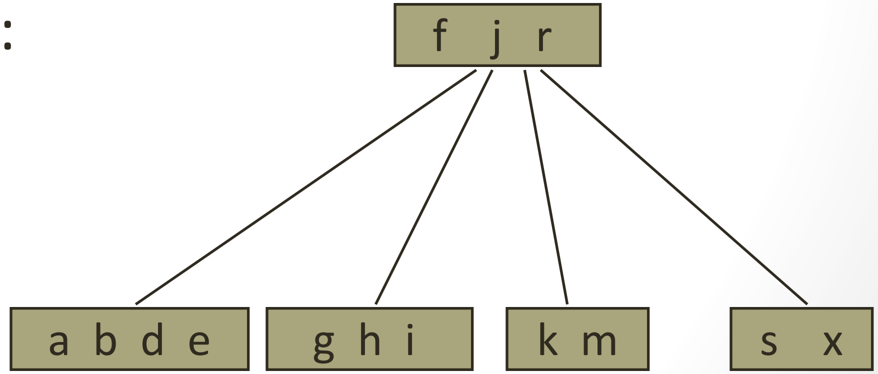


- 5.
- e, s, i, r:
-



6.

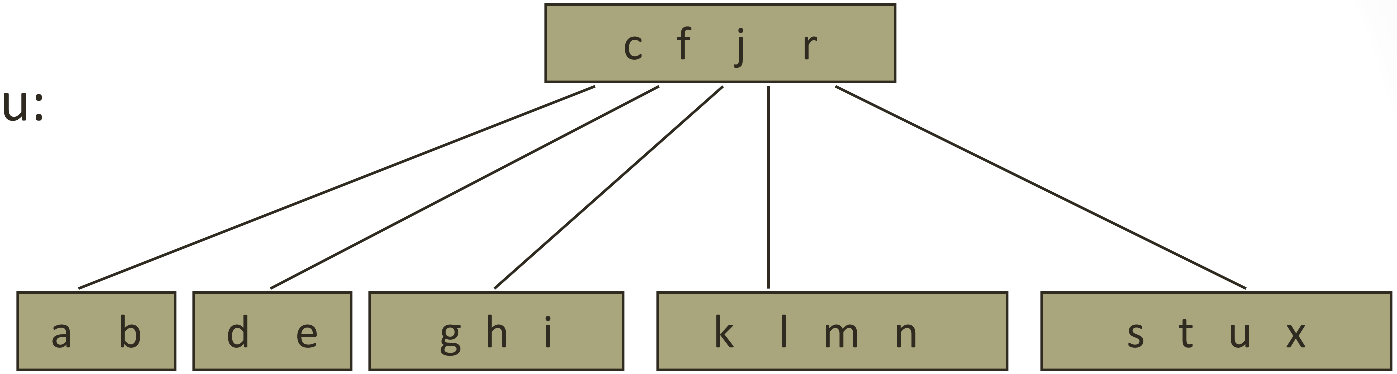
x:



Insertion (cont.)

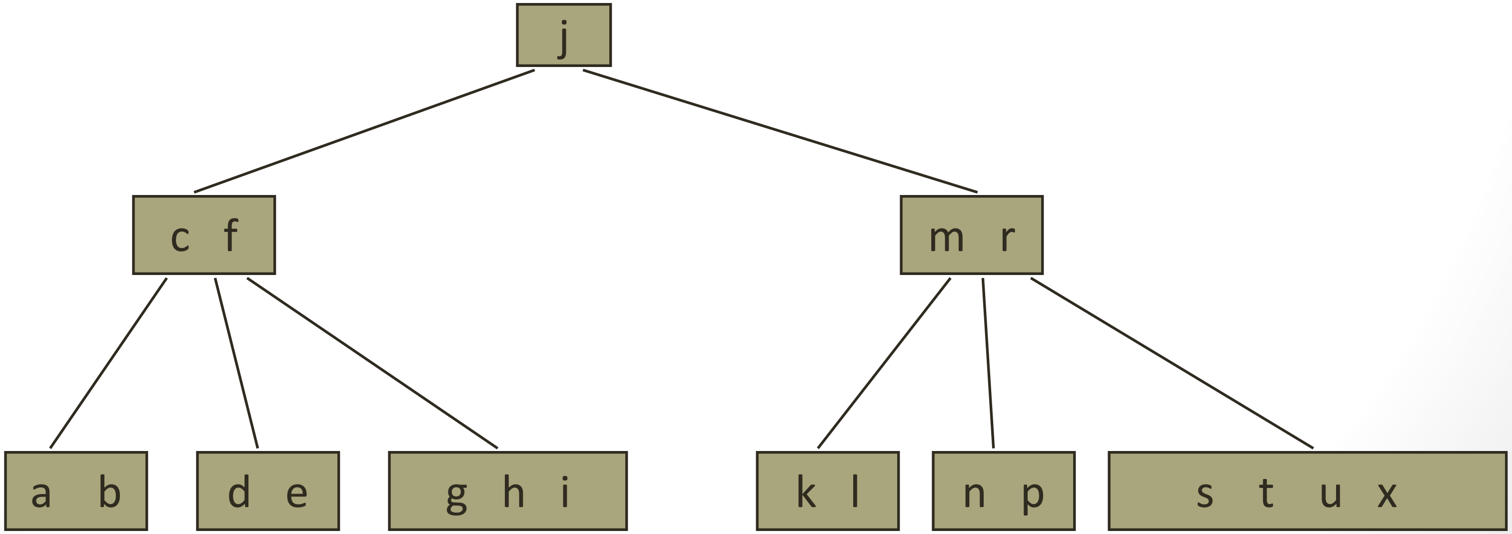
7\.

c, l, n, t, u:



8\.

p:

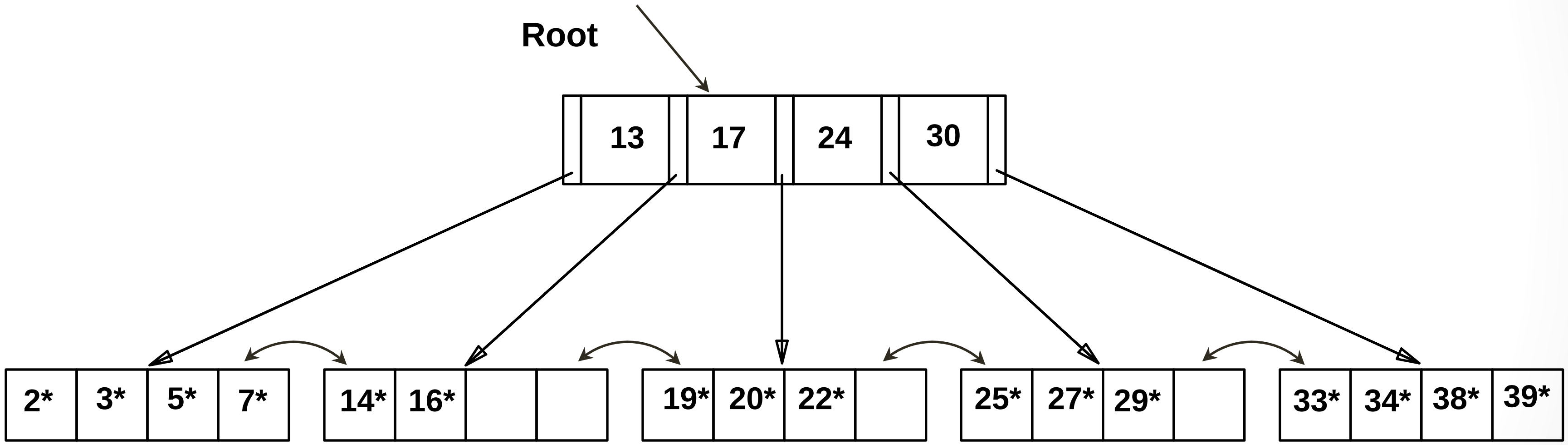


Inserting a Data Entry into a B+ Tree

- Find correct leaf *L.*
- 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*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but ***push up*** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

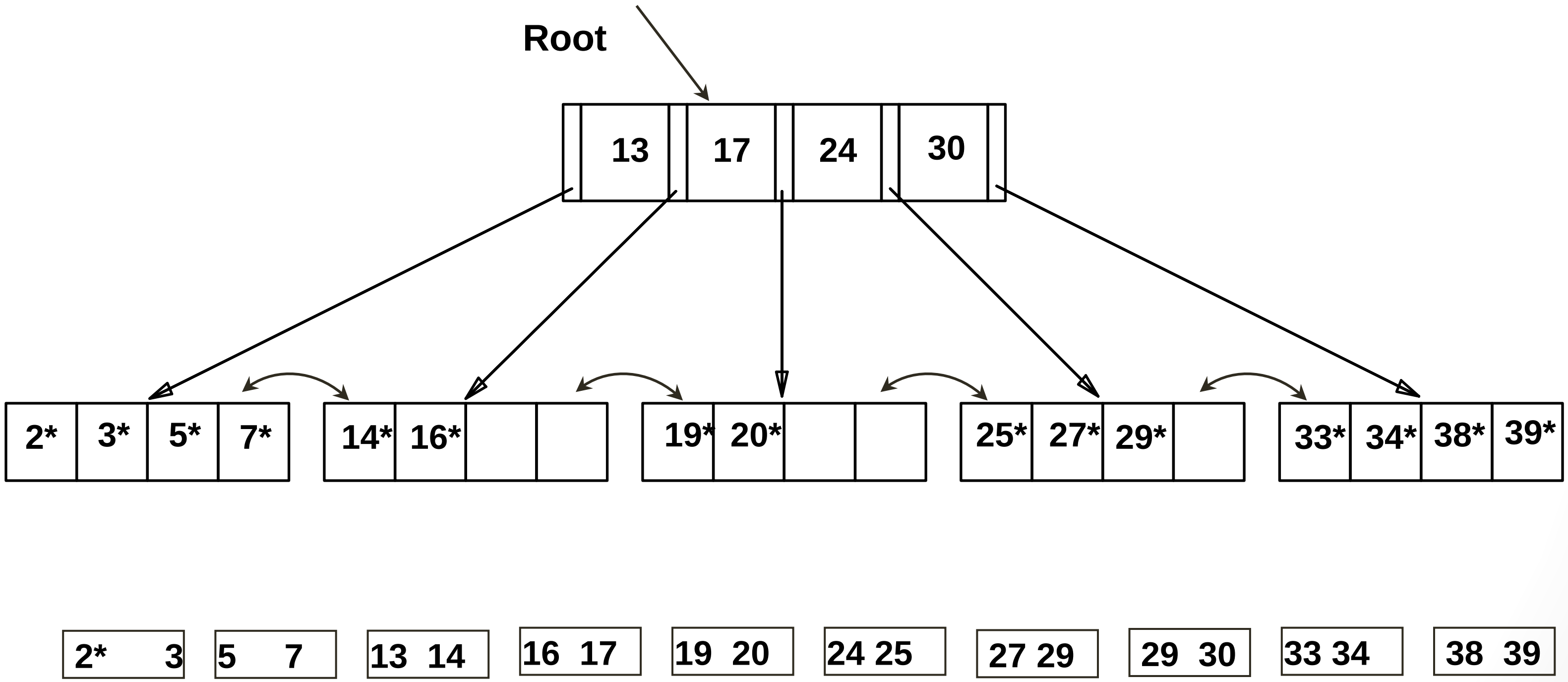
Previous B+ Tree Example

Inserting 8*



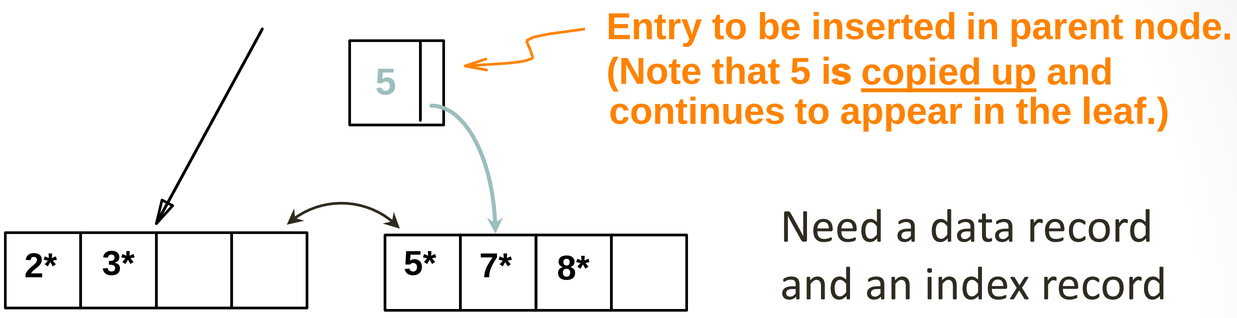
Previous B+ Tree Example

Inserting 8*

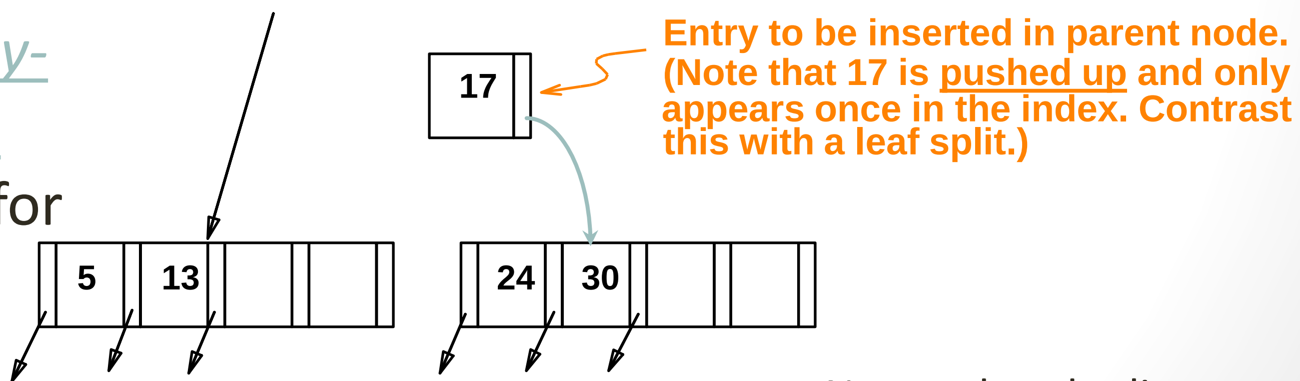


Inserting 8* into Example B+ Tree

- *Minimum occupancy* is guaranteed in both leaf and index pg splits.
- Note difference between copy-up and push-up. Reasons for this?

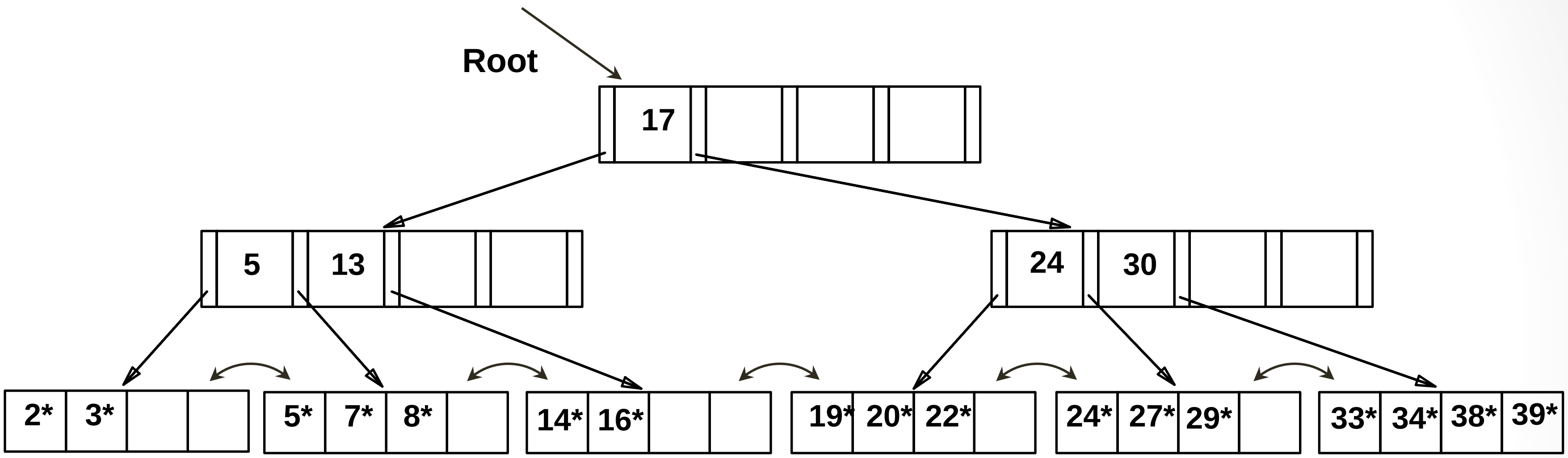


Need a data record and an index record



No need to duplicate Index record in index

Example B+ Tree After Inserting 8*



❖ Notice that root was split, leading to increase in height.

❖ In this example, we can avoid split by *re-distributing* entries between siblings; but not usually done in practice.

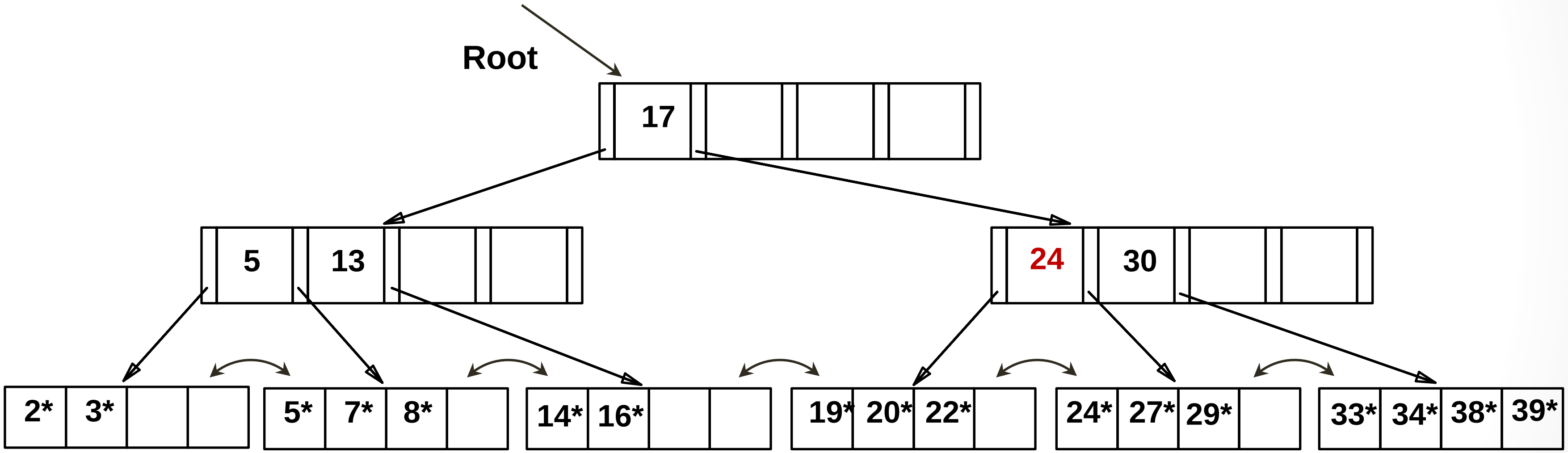
Deleting a Data Entry from a B+ Tree

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

Current B+ Tree

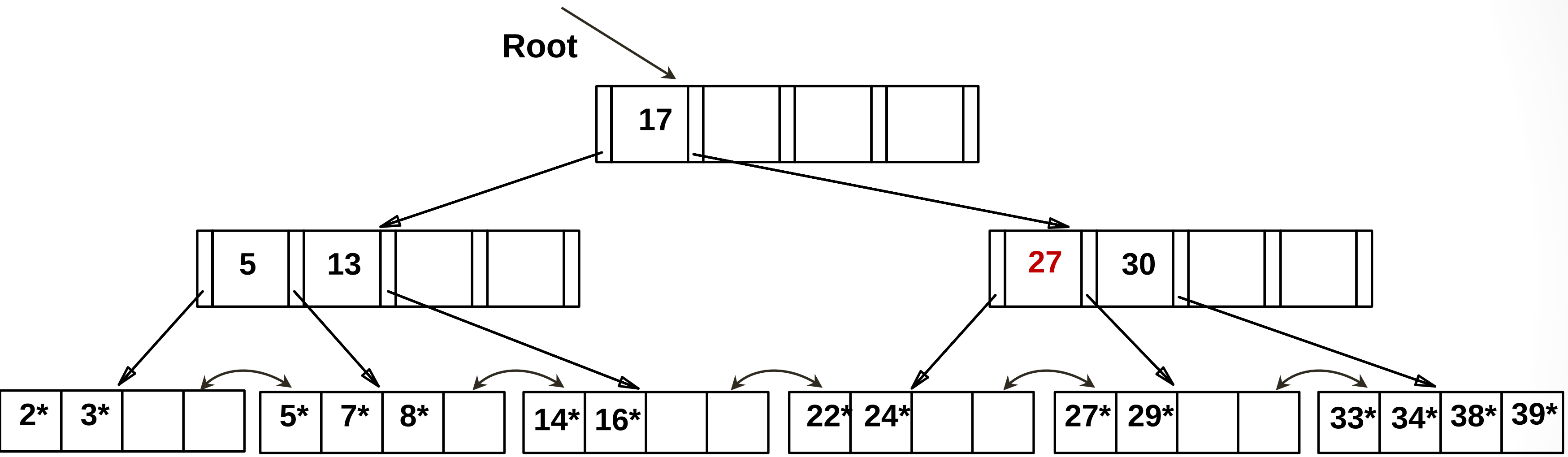
Delete 19*

Delete 20*



- One record on page after deletions
- Move records over from sibling page
  - Record 24 to the left

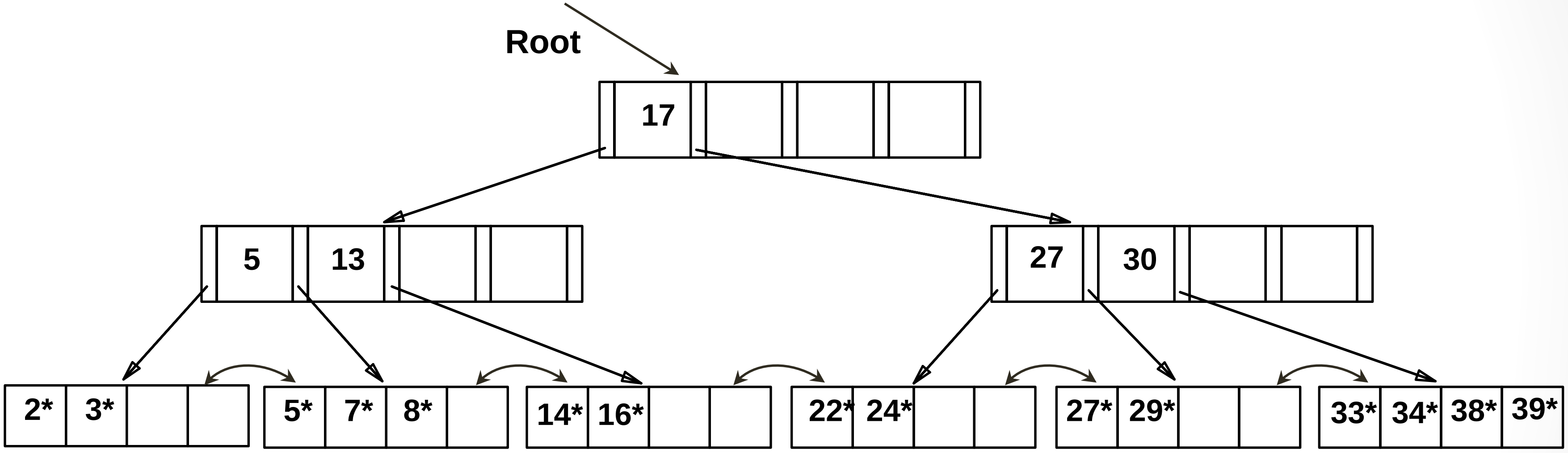
Example Tree After Deleting 19* and 20* ...



- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

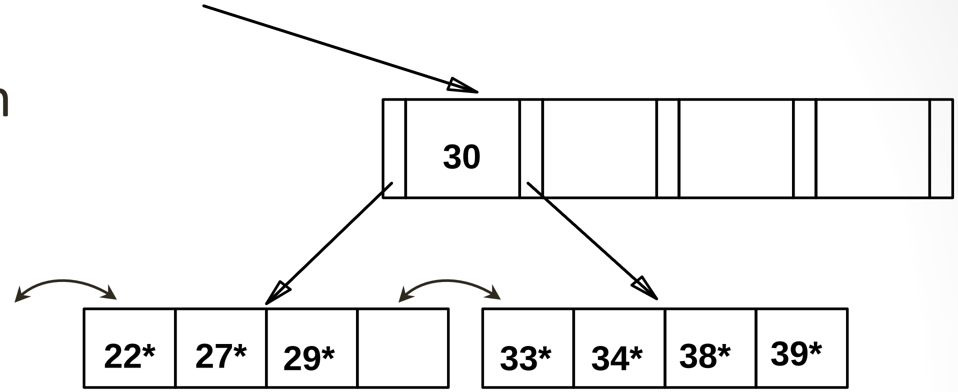
New B+ Tree …

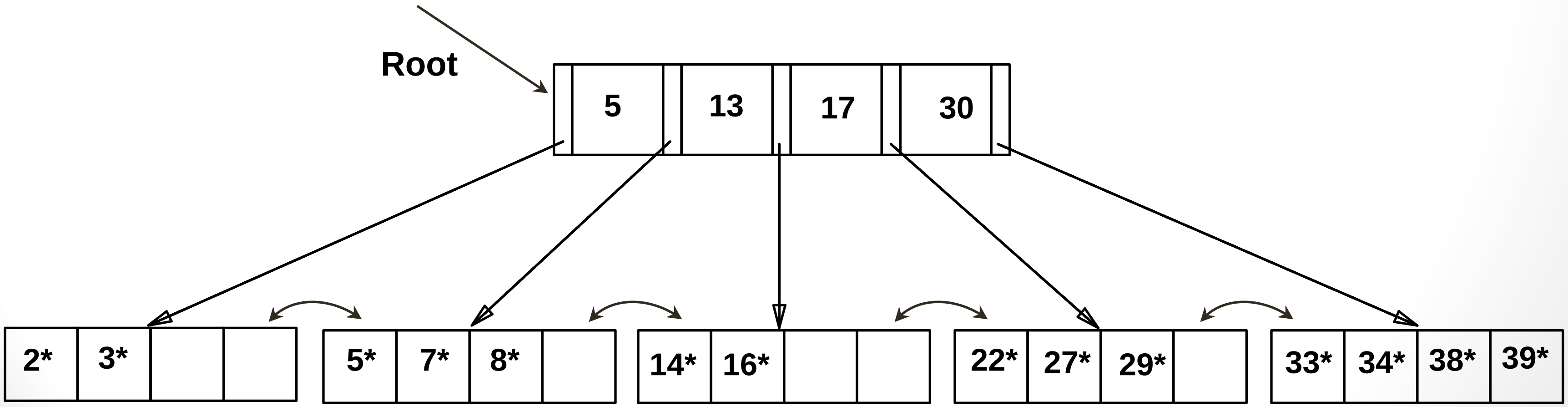
Delete 24*



… And Then Deleting 24*

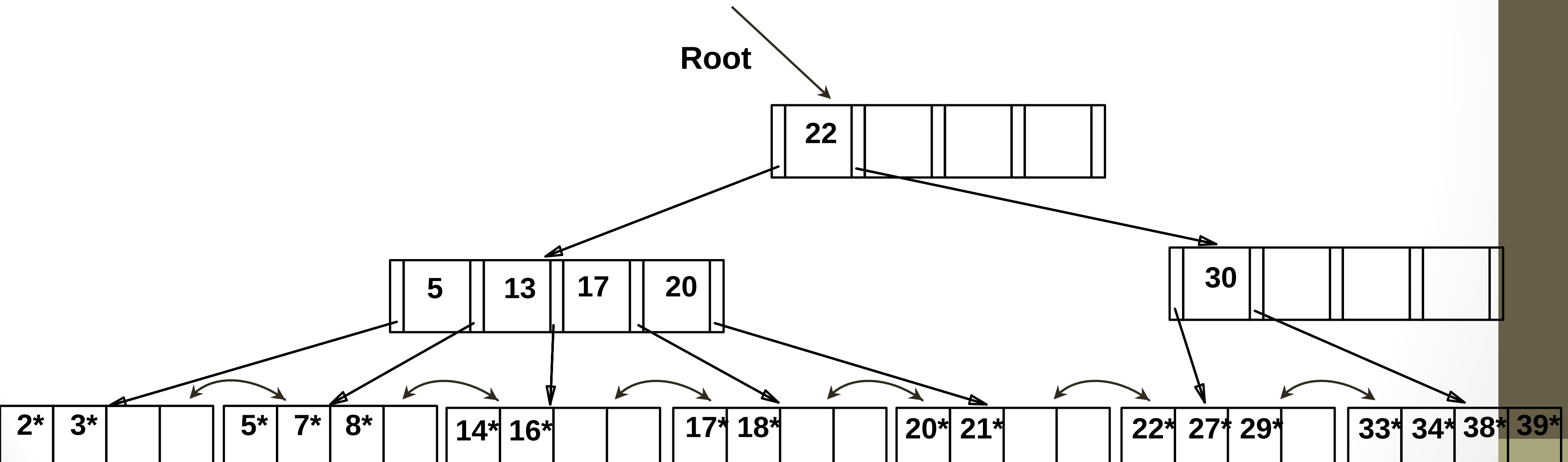
- Merge the two leaves to form [22,27,29].
- Observe *toss* of index entry ([30]on right), and *pull down* of index entry (below [17]).





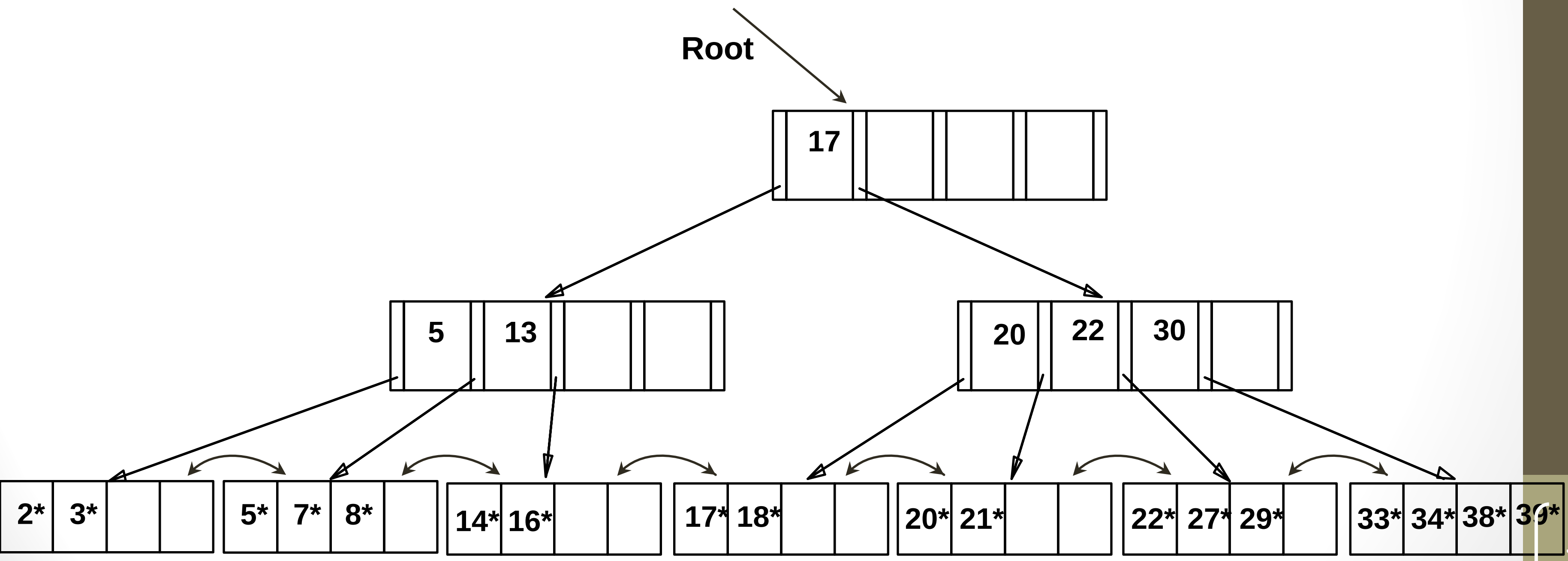
Example of Non-leaf Re-distribution

- Tree is shown below *during deletion* of 24*.
- In contrast to previous example, can re-distribute entry from left child of root to right child.



After Re-distribution

- Intuitively, entries are re-distributed by *`pushing through'* the splitting entry in the parent node.
- It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.



B+Tree Growth and Change

The big idea: When a node is full, it splits.

- middle value is propagated upward
  - If we're lucky, there's room for it in the level above
- two new nodes are at same level as original node

- Height of tree increases *only* when the root splits
  - A very nice property
  - This is what keeps the tree perfectly balanced
- Recommended: split only "on the way down"
- On deletion: two adjacent nodes recombine if both are < half full

Duplicate records

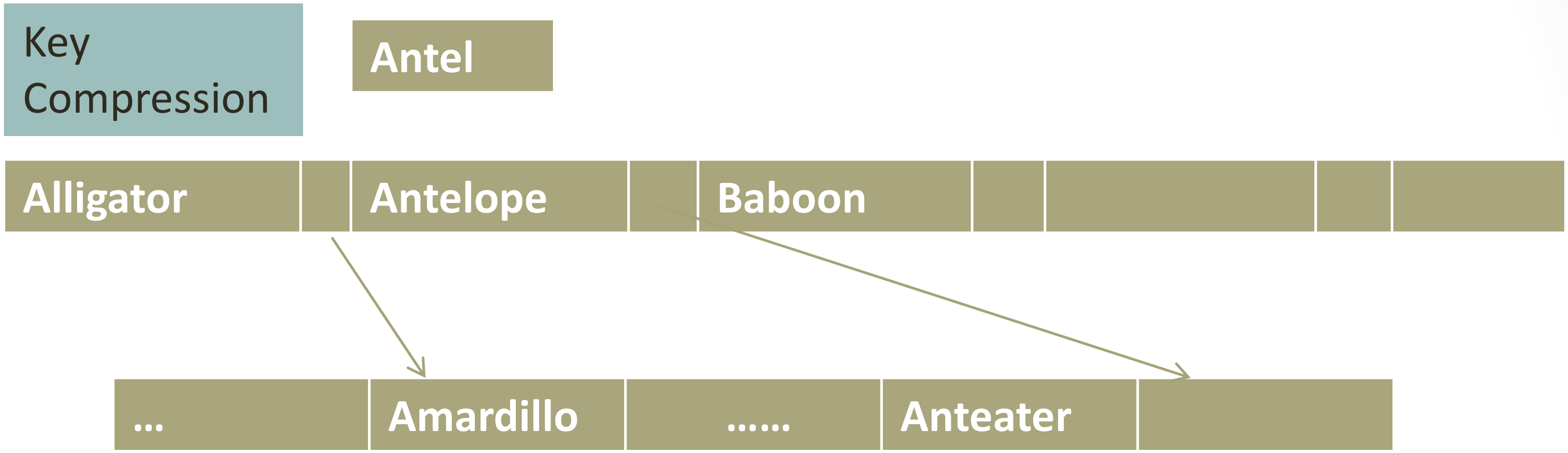
- Up to now we have considered 1 record for each key
- How do we handle duplicate records?
  - Search – find first page with the given value then retrieve more 'next' leaf page until the criterion fails
  - Delete – How do we identify which record to delete?
    - Treat the search key as including the record id – since it makes the record unique

Prefix Key Compression

- Height of a B+ tree depends on the number of data entries and the size of index entries
  - Size of index entries determines the number of index entries that will fit on a page – and therefore the fan-out of the tree.
- Key Compression can increase fan-out. (Why?)
- Key values in index entries only `direct traffic'; can often compress them.
  - E.g., adjacent index entries with search key values

    [*Dave Jones*, *David Smith* and *Devarakonda Murthy*]
  - Can we abbreviate *David Smith* to *Dav*?
    - Not correct! Can only compress *David Smith* to *Davi*.
    - In general, while compressing, must <u>leave each index entry greater than every key value (in any subtree) to its left</u>.
- Insert/delete must be suitably modified.

Prefix Key compression

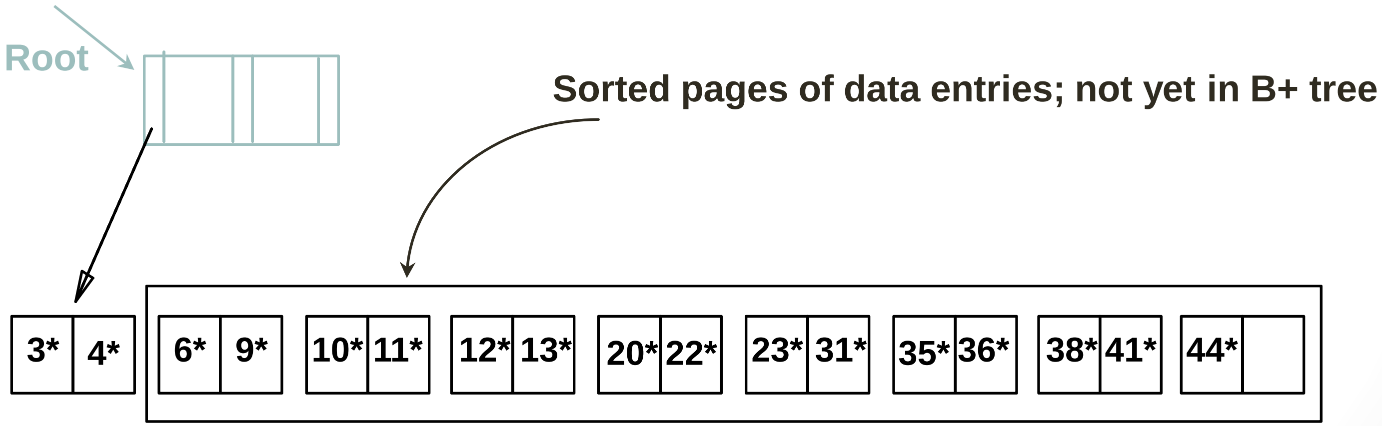


Bulk Loading of a B+ Tree

- Have a large collection of records, and want to create a B+ tree on some field. Doing so by repeatedly inserting records?
  - Slow due to repeated traversals and splits
  - Significant locking overhead.
  - Not necessarily the optimal structure. An example?
  - Low storage utility. An example?
- *Bulk Loading* can be done much more efficiently!

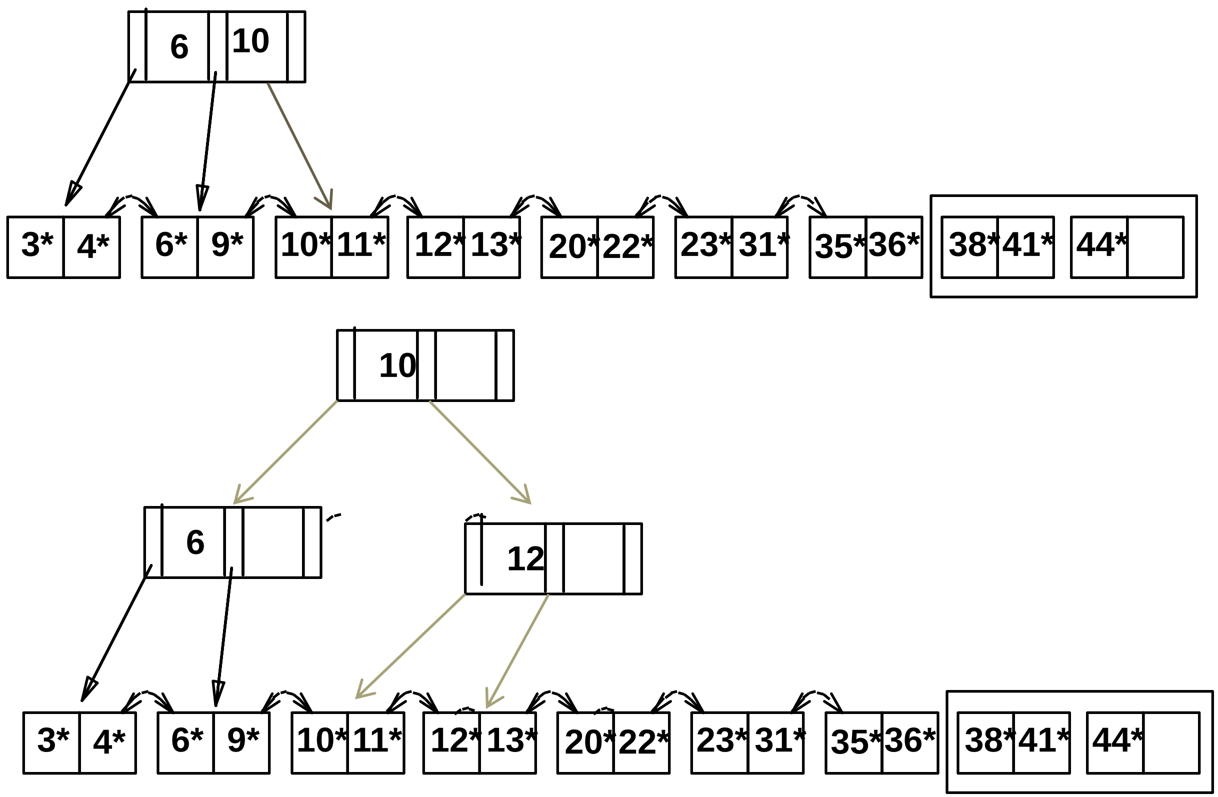
Bulk Loading Algorithm

- *Initialization*:
  - Sort all data entries
  - Insert pointer to the first (leaf) page into a new (root) page.



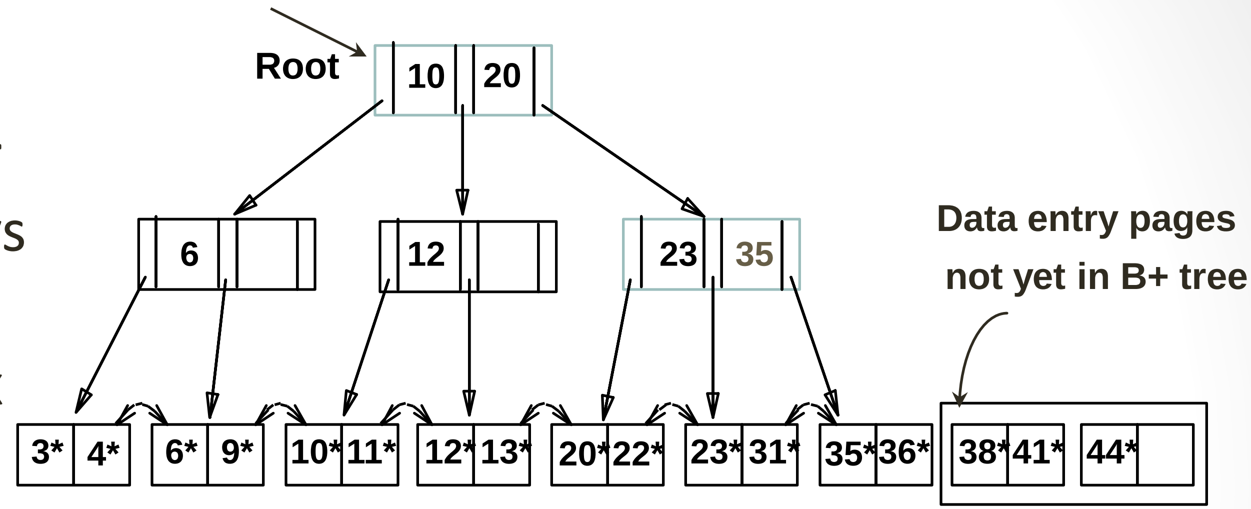
Page splitting during bulk insert

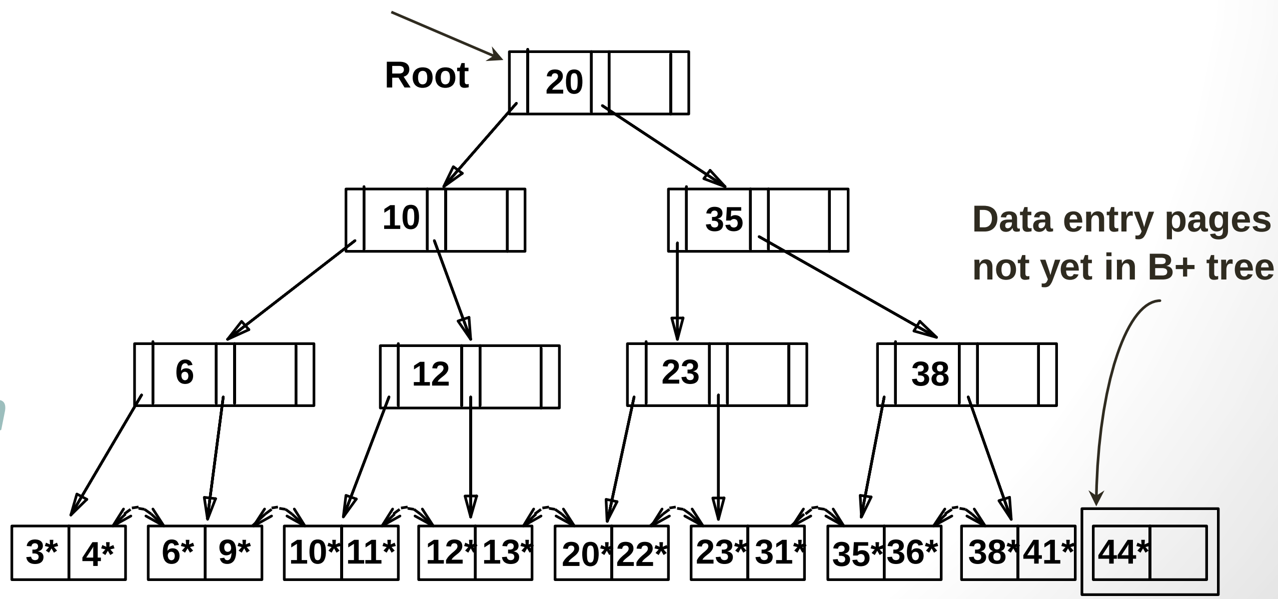
- Insert the minimal key value for each page
- Continue until all data pages processed



Bulk Loading Algorithm (Contd.)

- Index entries for leaf pages always enter into r*, right-most index page just above leaf level.
- When the r* node fills up, it splits.
- Split may go up *right-most path to the root*.





Summary of Bulk Loading

- Option 1: multiple inserts.
  - Slow due to I/O cost and locking overhead.
  - Does not provide sequential storage of leaves.
  - Sometimes low storage utility.
- Option 2: <u>*Bulk Loading*</u>
  - 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* (**n**) 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 means different nodes will contain different number 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: Tree-based Index

- 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: B+ trees

- 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 fan-out, 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.