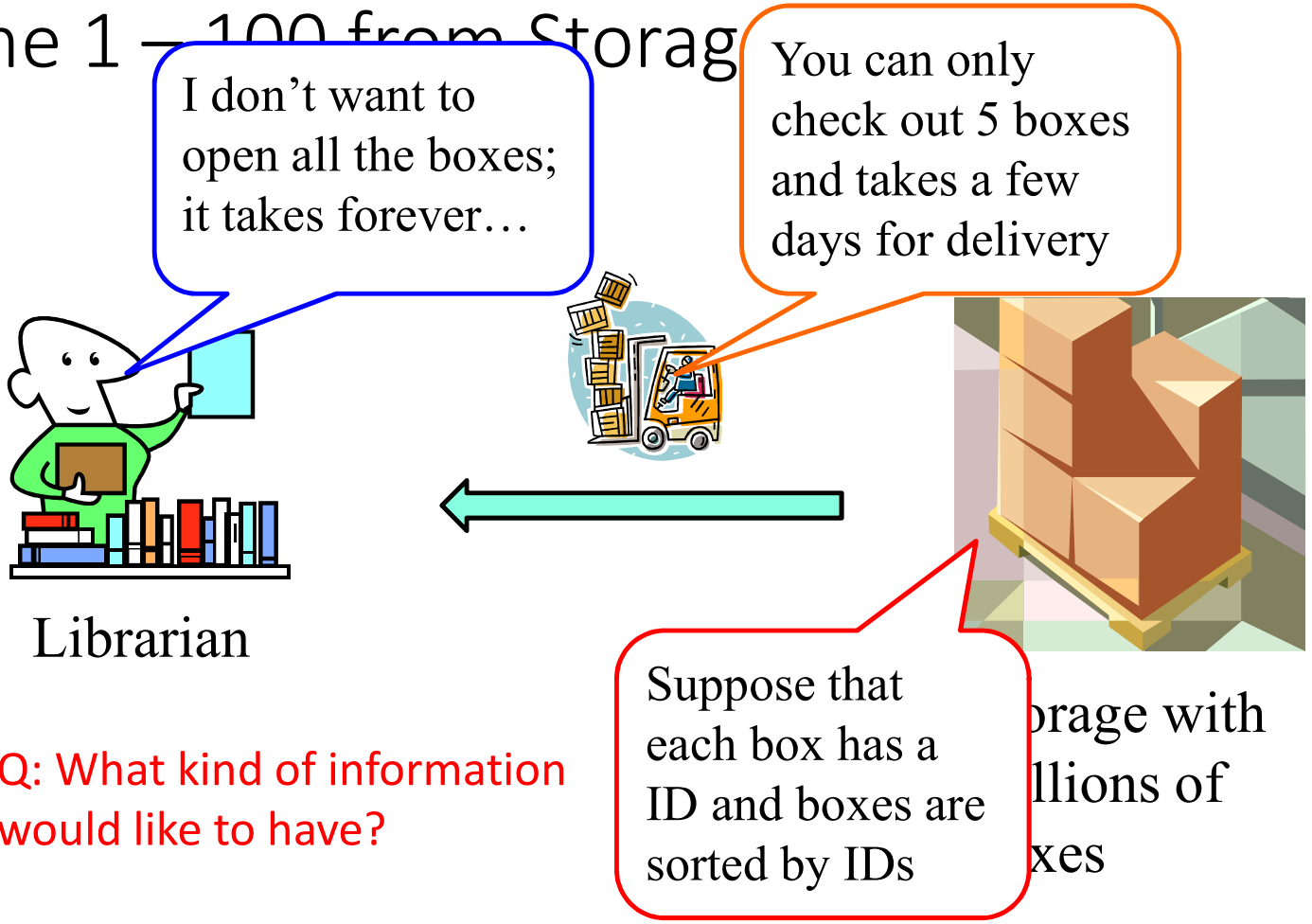
ICCS240 Database Management

Indexing

How to find boxes containing “History of Bangkok” Volume 1 – 100 from Storag



Q: What kind of information would like to have?

Probably, what you want is a table (i.e., index)

Book title	Box ID
History of BKK vol. 1	925
History of BKK vol. 2	925
History of BKK vol. 3	926
History of BKK vol. 4	928
History of BKK vol. 5	928
History of BKK vol. 6	928
History of BKK vol. 7	1001
History of BKK vol. 8	1002
History of BKK vol. 9	1002
History of BKK vol. 10	1003

Q: do you think that we solved the problem?

Q: how do you find the entry for “History of BKK” in this table?

Q: What if this table is so big and is stored in boxes in the storage?

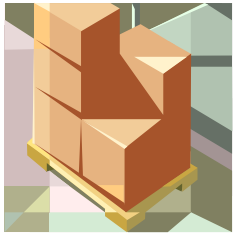
This is the exact problem we need to address in DBMS

How to first find boxes containing Index for “History of Bangkok” from Storage?

Need to retrieve index blocks first

Book title	Box ID
History of BKK vol. 1	925
History of BKK vol. 2	925
History of BKK vol. 3	926
History of BKK vol. 4	928
History of BKK vol. 5	928
History of BKK vol. 6	928
History of BKK vol. 7	1001
History of BKK vol. 8	1002
History of BKK vol. 9	1002
History of BKK vol. 10	1003

Librarian



Boxes for index

Storage with millions of boxes

Hashing & Sorting

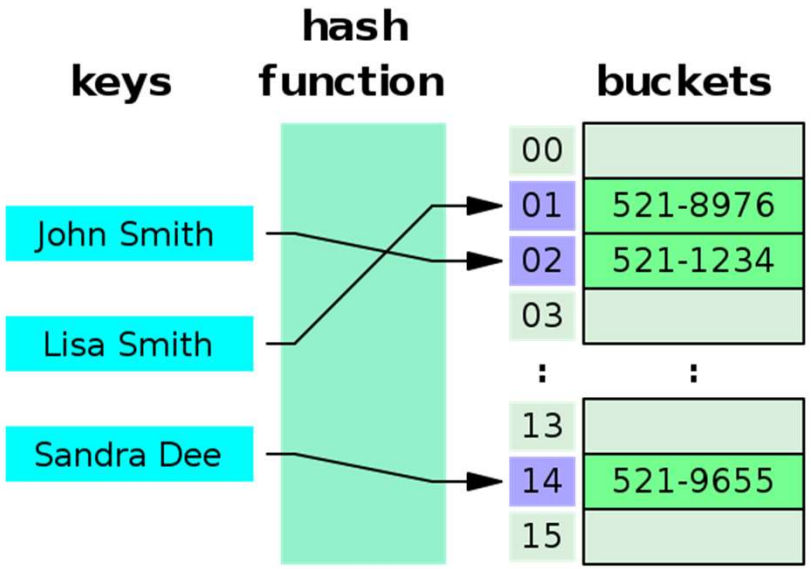
Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  E.g., author catalog in library, shelf/box of books in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices**: search keys are stored in sorted order
  - **Hash indices**: search keys are distributed uniformly across *buckets* using a *hash function*.

Static Hashing

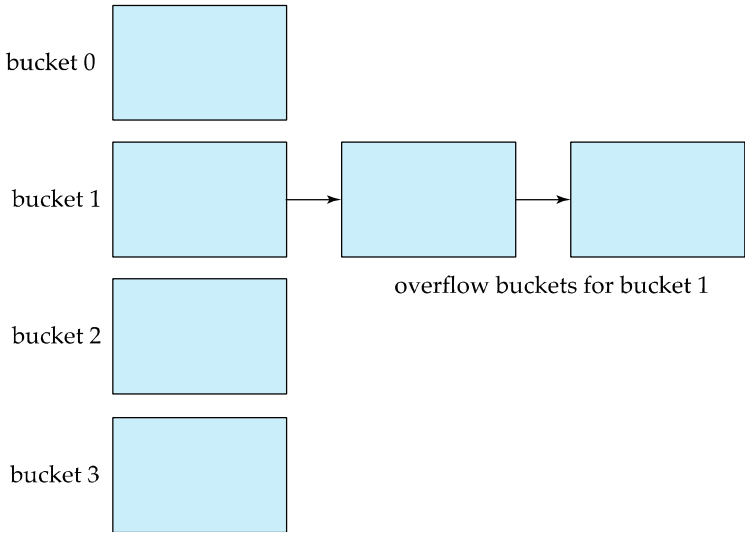
- A **bucket** (typically a disk block) stores one or more entries.
- **Hash function h** is a function from the set of all search-key values K to the set of all bucket addresses B.
- Hash function is used to locate entries for access, insertion as well as deletion.



Entries with different search-key values may be mapped to the same bucket. Thus, entire bucket has to be searched sequentially to locate an entry.

Static Hashing will overflow

- Bucket overflow can occur because:
  - Insufficient buckets
  - Skew in distribution of records.
    - Multiple records have same search-key value
    - Chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated.
  - ✓*Handled by using **overflow buckets***



Static Hashing (cont.)

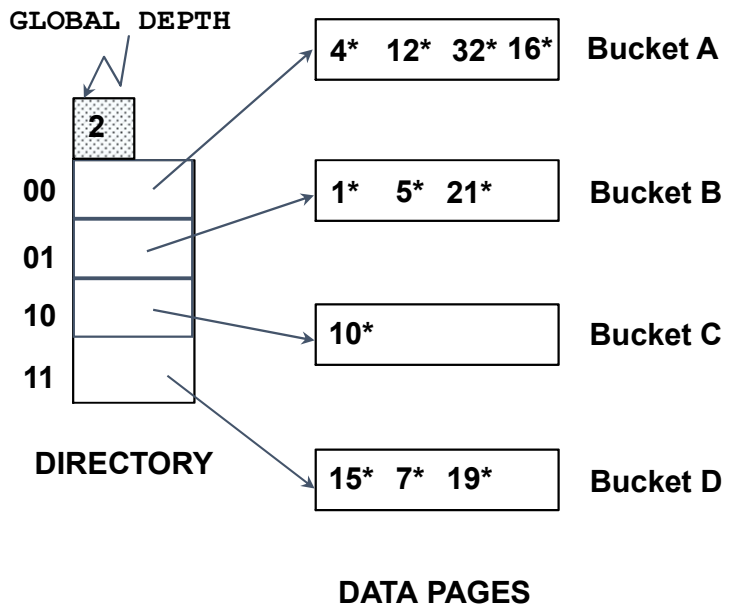
- Search
  - Identify the correct bucket using h
  - Search the bucket for the data entry
- Insert
  - Identify the correct bucket using h
  - If no space in the bucket, allocate new overflow page to the overflow chain of the bucket, put the data entry on this bucket.
- Delete
  - Identify the correct bucket using h
  - Search the bucket for the data entry, remove it.
  - If it is the last in an overflow bucket, remove the bucket from the chain and de-allocate the page of the bucket.

Dynamic Hashing
– adjust the hash table as situation changes

- Periodic Re-Hashing
  - If #entries in a hash table becomes (say) 1.5 times size of hash table,
    - create new hash table of size (say) 2 times the size of the previous hash table
    - Re-hash all entries to new table
  - But, `READ` and `WRITE` all pages is *costly*!
- Extendable Hashing
  - Tailored to disk-based hashing, with buckets shared by multiple hash values
  - Doubling of # of entries in hash table, without doubling # of buckets
- Linear Hashing
  - Do rehashing in an incremental manner

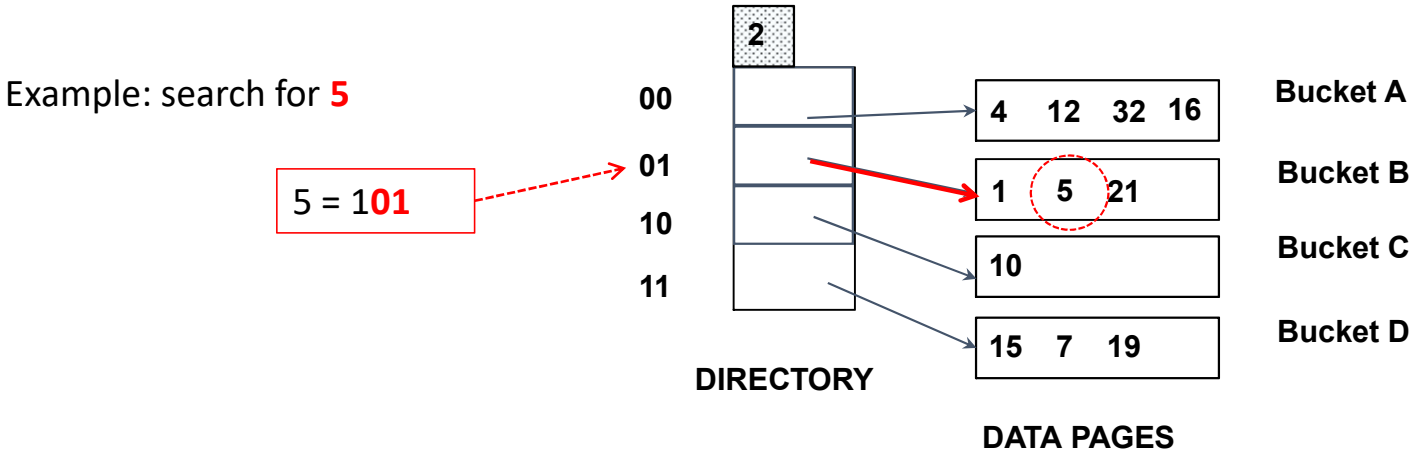
Extendible Hashing

- Extendible Hashing uses a directory of pointers to buckets
- The result of applying a hash function h is treated as a *binary number* and the last d bits are interpreted as an offset into the directory
  - d is referred to as the *global depth* of the hash file and is kept as part of the header of the file



Extendible Hashing: Search

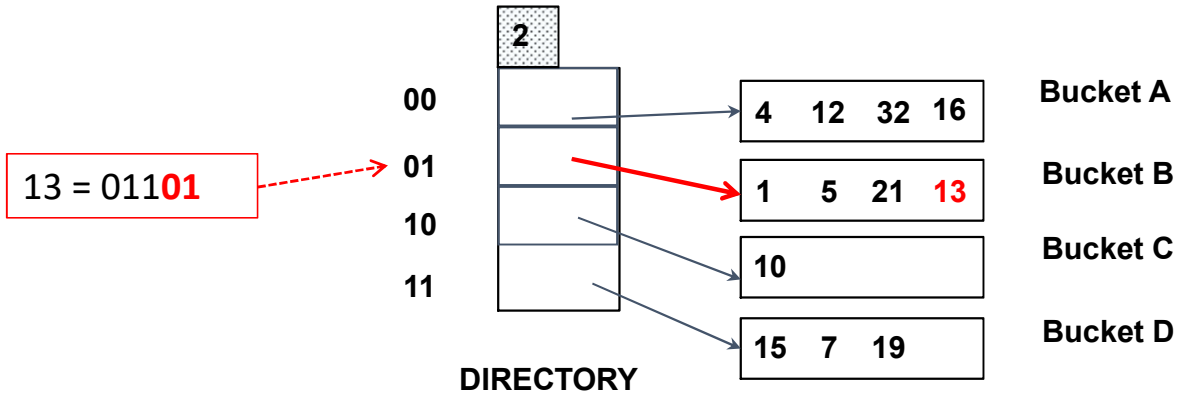
- To search for a data entry, apply a hash function h to the key and take the last d bits of its binary representation to get the bucket #.



Extendible Hashing: Insertion

- Find the appropriate bucket (as in search), insert the given entry.

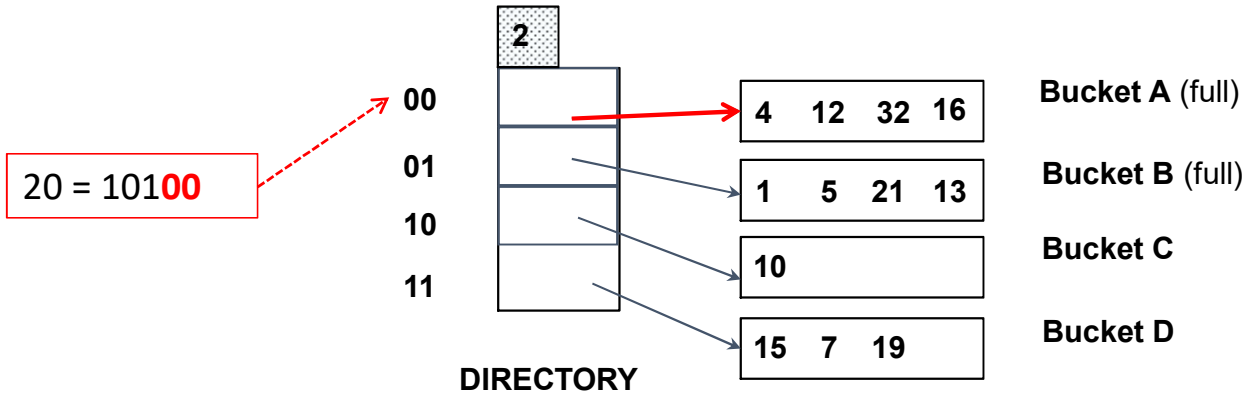
Example: insert **13**



Extendible Hashing: Insertion

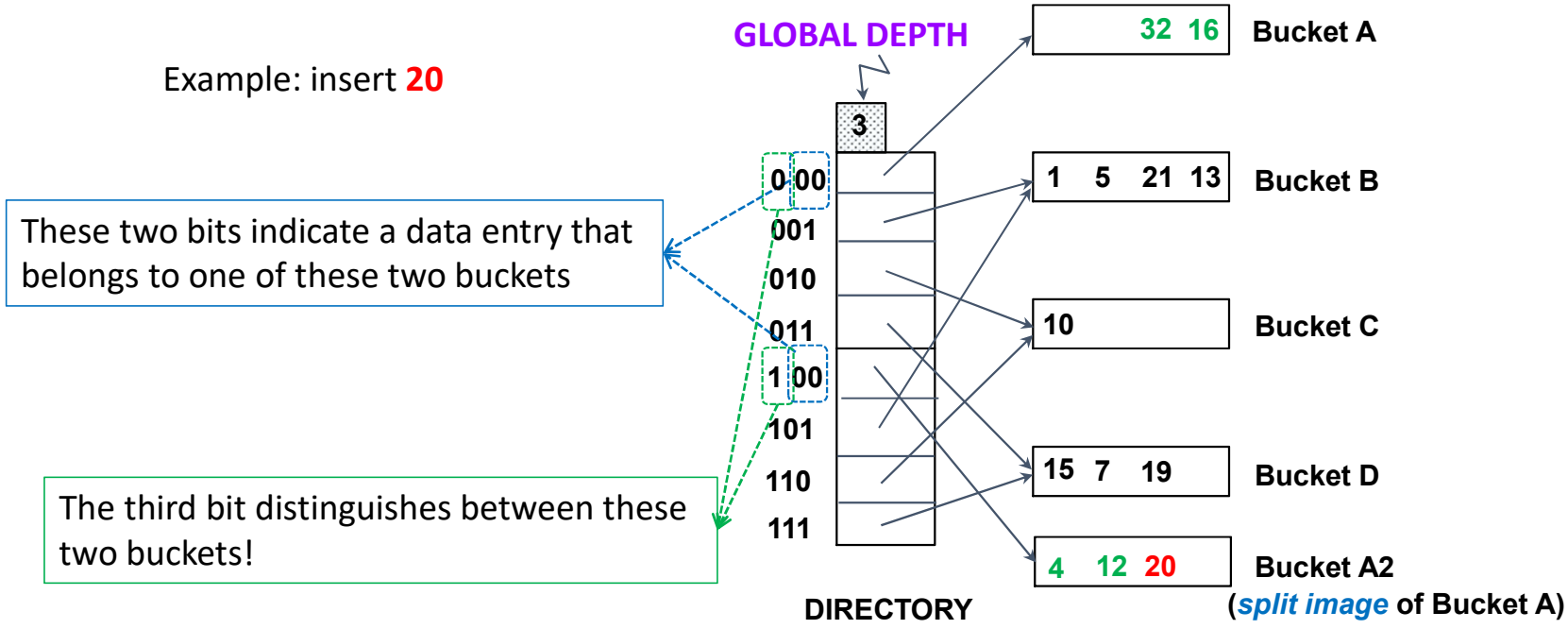
- Find the appropriate bucket (as in search), split the bucket if FULL, double the directory if necessary and insert the given entry.

Example: insert **20**



Extendible Hashing: Insertion

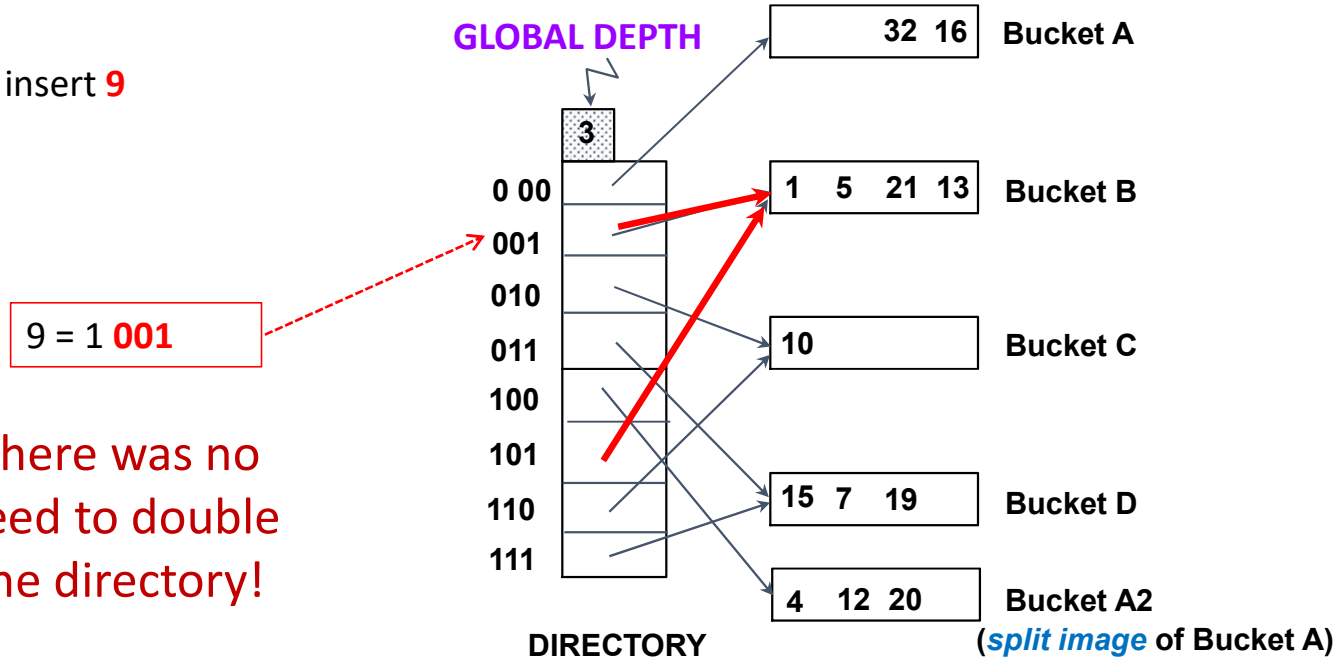
- Find the appropriate bucket (as in search), split the bucket if FULL, double the directory if necessary and insert the given entry.



Extendible Hashing: Insertion

- Find the appropriate bucket (as in search), split the bucket if FULL, double the directory if necessary and insert the given entry.

Example: insert **9**

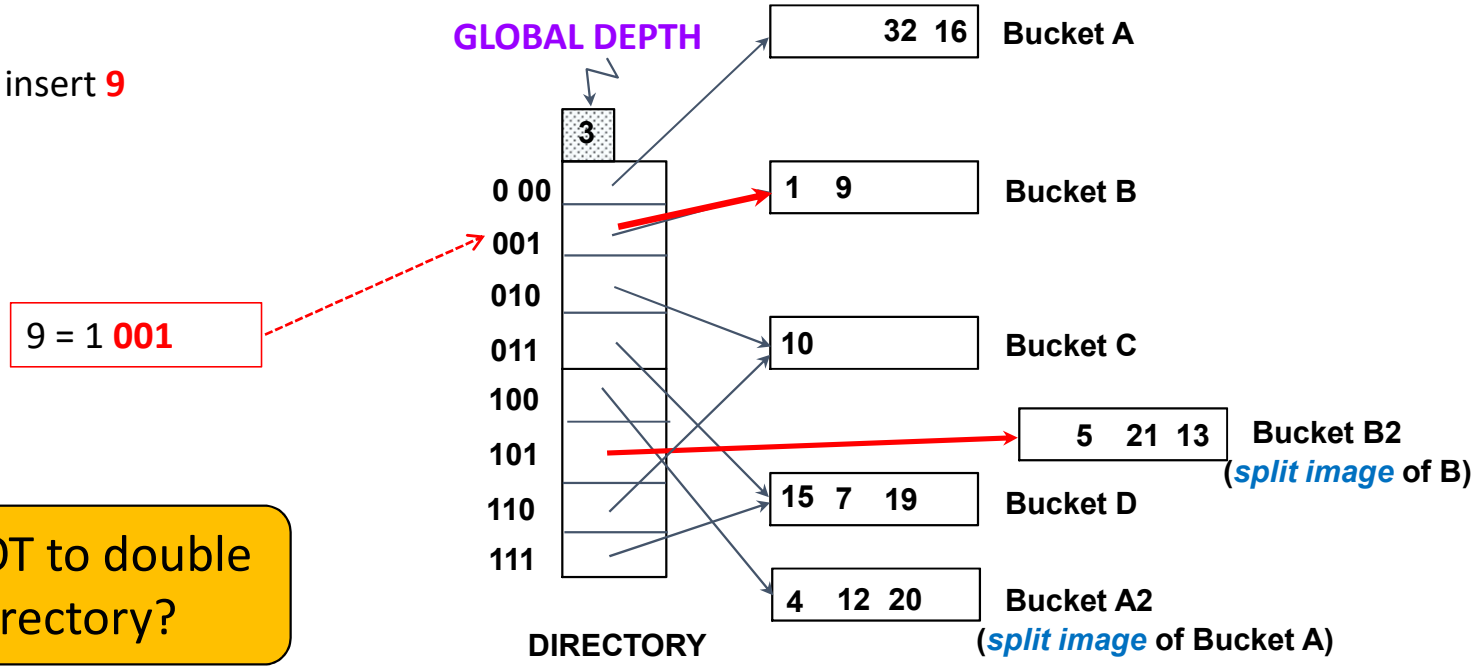


There was no need to double the directory!

Extendible Hashing: Insertion

- Find the appropriate bucket (as in search), split the bucket if FULL, double the directory if necessary and insert the given entry.

Example: insert **9**

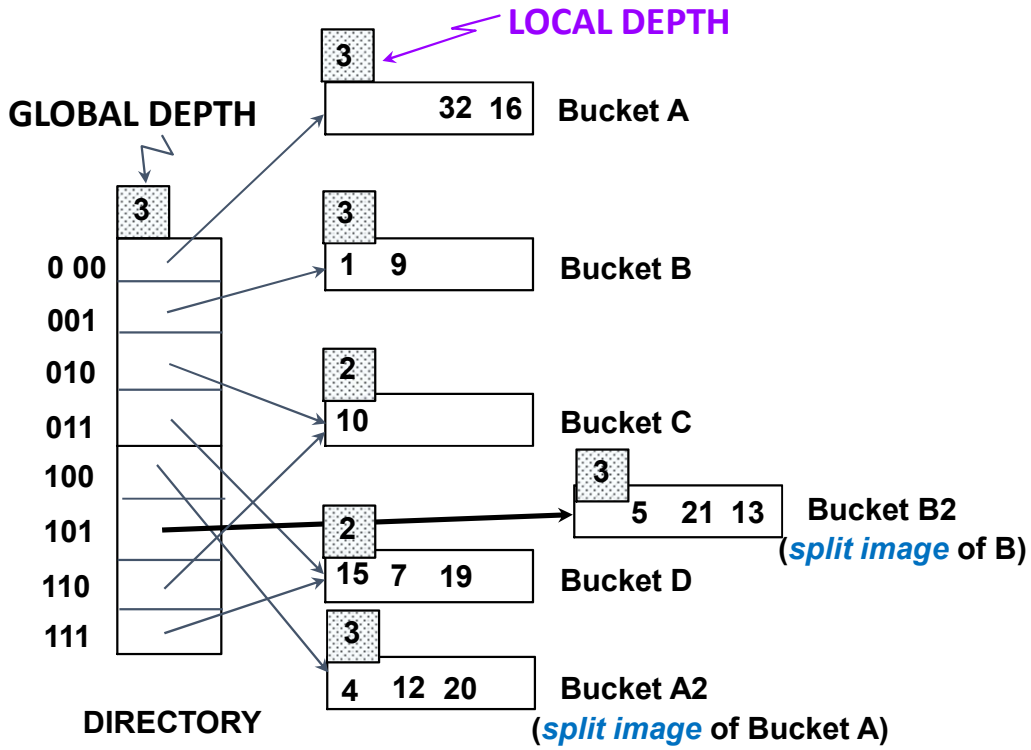


When NOT to double the directory?

Extendible Hashing: Local Depth

Local depth = #of bits used to determine if an entry belong to this image of bucket

If a bucket whose **local depth equals to the global depth** is split, the directory *must* be doubled



Extendible Hashing: Deletion

- Find the appropriate bucket (as in search)
- Remove the data entry

(unnecessary steps)

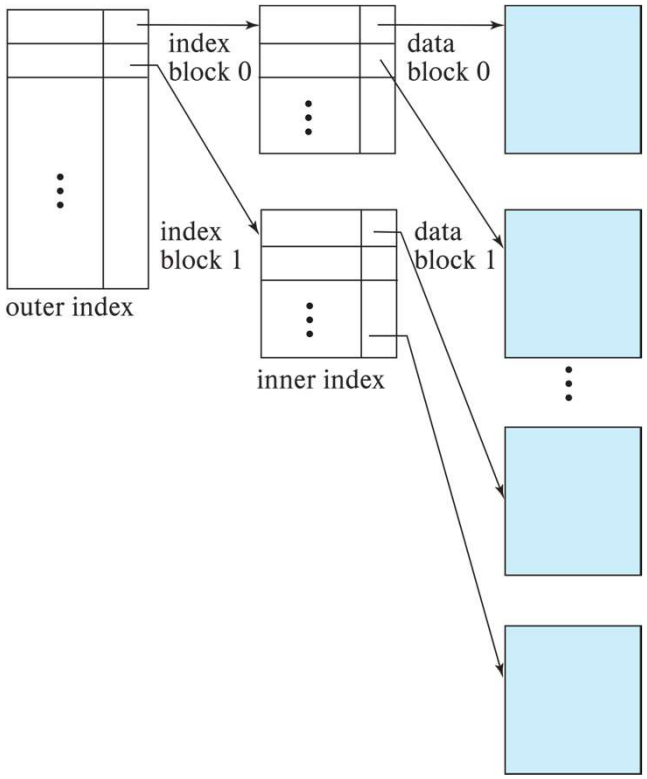
- If the removal makes the bucket empty, it depth is decreased.
- If the split image is empty, merge the buckets
- If each directory element points to same bucket as its split image, the directory can be halved → *global depth is decreased*

Extendible Hashing: complexity

- I/O cost of equality search
  - If the directory fits in memory, equality search can be answered with one disk access (and maybe one more access to retrieve the actual record).
  - Otherwise, two disk accesses (or more?)

Multi-level Index
– What if your index can't fit in RAM?

- If index does not fit in memory, access becomes expensive.
- Solution: Treat index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of the basic index
  - inner index – the basic index file
- If even outer index is too large to fit in main memory,
  yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.



Linear Hashing

- An alternative to extendible Hashing
- Idea: use a family of hash functions $h_0, h_1, h_2, \ldots$
  - N = initial # of buckets
  - h is some hash function
  - d_0 = number of bits to represent N
  - If $N = 2^{d_0}$, h_i consists of applying h and looking at the last d_i bits, where $d_i = d_0 + i$.
  - $h_i = h(key) \bmod 2^i N$
  - Note: h_{i+1} doubles the range of h_i (similar to directory doubling)

Linear Hashing

Linear Hashing avoids directory by using overflow pages and choosing bucket to split *round-robin*

- Splitting proceeds in “rounds”.
  Current round number is called $level$ (initially 0).
- Buckets to split is denoted by $Next$.

  $Next$ is initialized to 0 when a round begins, and increased by 1 after *splitting*.
  E.g., Buckets 0 to $Next$-1 have been split; buckets $Next$ to N_{level} have not.
- Splitting is triggered when an overflow page is added,
  and $h_{level+1}$ redistributes entries between this bucket and its split image.
- The round $level$ ends when all N_{level} initial buckets are split.

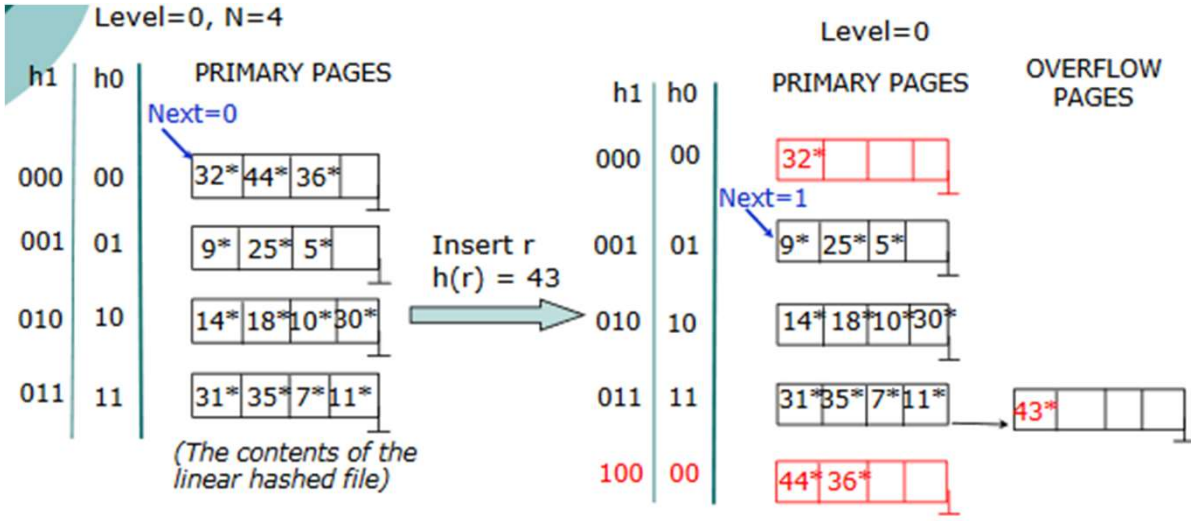
Linear Hashing: Search

To search for data entry r

- Find the "bucket" by first calculating $h_{level}(r)$
- If $h_{level}(r)$ is in the range ($Next \sim N_{level}$),
  r belongs to that bucket.
- Otherwise, r could belong to
  bucket $h_{level}(r)$ or bucket $h_{level}(r) + N_{level}$
  So use h_{lev} (r) to find out.

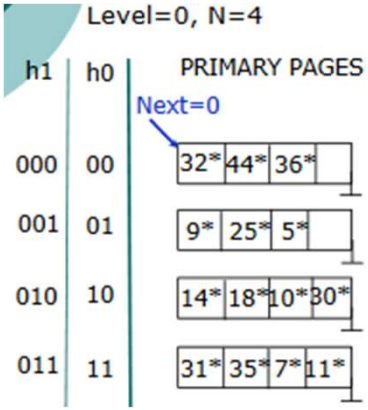
Linear Hashing: Insertion

- Find appropriate bucket of data r (as in search)
- If bucket to insert into is full,
  - Add overflow page and insert the data entry
  - Split $Next$ bucket (its entries are redistributed by h_{lev}) and increment $Next$.

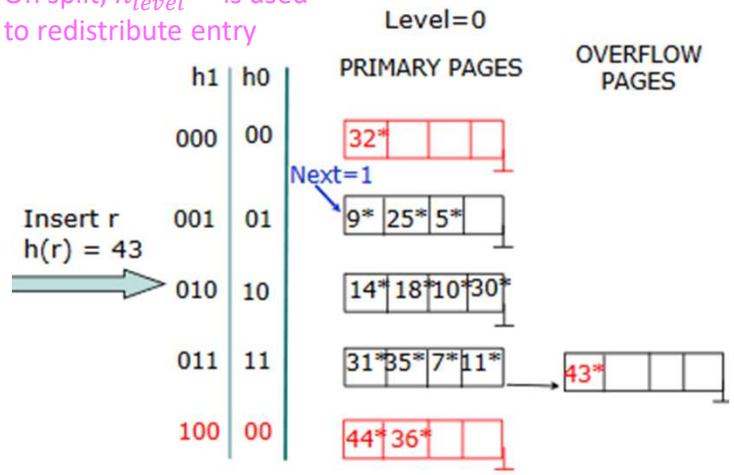


Source: Linda Wu, Stanford U

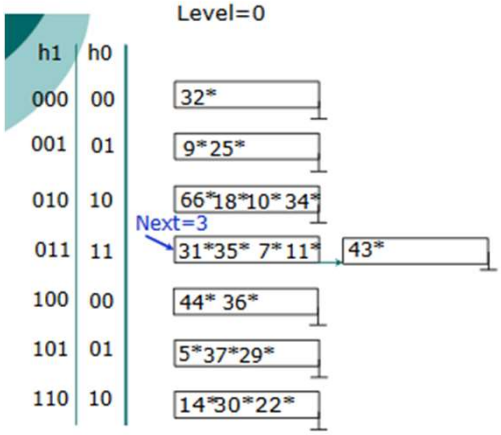
Example

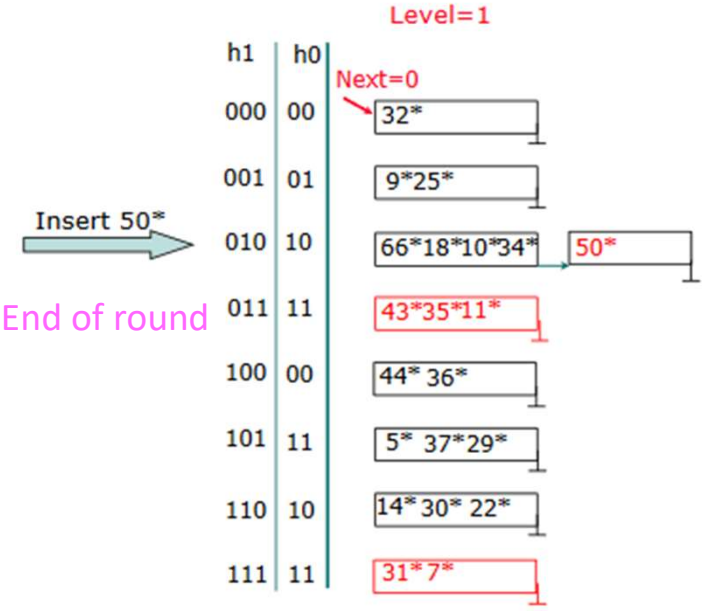


On split, h_{level} is used to redistribute entry



...





Source: Linda Wu, Stanford U

Extendible Hashing vs. Linear Hashing

- The two schemes are quite similar
  - Begin with an imaginary directory in LH with N buckets
  - LH split first at bucket 0
  - The directory is doubled gradually.
- Moving from h_i to h_{i+1} in LH corresponds to doubling the directory in EH.

  For EH:
  - Directory is doubled in a single step
  - Always splitting the appropriate bucket (reduced # of splits and a higher bucket occupancy)

  For LH:
  - Directory is doubled gradually over the course of a round

(External) Sorting

When does a DBMS sort data?

- Users may want answers in some order

```
SELECT FROM Student ORDER BY Name;

SELECT M.Rating, MIN(M.Age) FROM Movies M GROUP BY
M.Rating;
```

- Bulk loading a B+ tree index involves sorting
- Sorting is useful in eliminating duplicate records

In-Memory vs. External Sorting

Assume we want to sort 60GB of data on a machine with only 8GB of RAM!

- In-memory sort (e.g., quicksort)?
  - Yes, but data do not fit in memory
  - What about relying on virtual memory?

In this case, **External Sorting** is needed.

- In-memory sorting is ORTHOGONAL to external sorting!

A Simple Two-Way Merge Sort

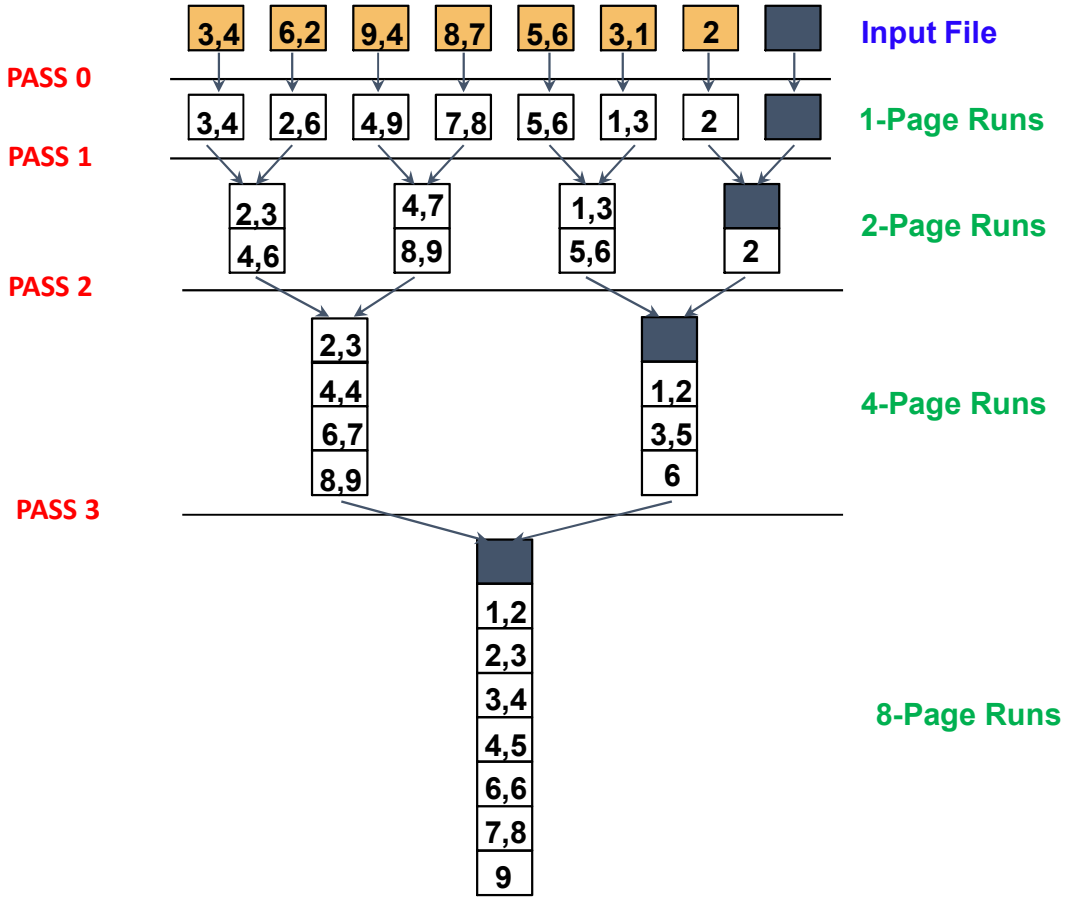
- Idea: sort sub-files that can fit in memory and merge

- Let's refer to each *sorted sub-file* as a **RUN**.

- Algorithm:
  - Pass 1: Read a page into memory, sort it, write it

    1-Page runs are produced.
  - Passes 2, 3, … : Merge **pair**s (hence, 2-way) of runs to produce longer runs until only one run is left.

Example


3,4
6,2
9,4
8,7
5,6
3,1
2
Input File
PASS 0
3,4
2,6
4,9
7,8
5,6
1,3
2
1-Page Runs
PASS 1
2,3
4,6
4,7
8,9
1,3
5,6
2
2-Page Runs
PASS 2
2,3
4,4
6,7
8,9
1,2
3,5
6
4-Page Runs
PASS 3
1,2
2,3
3,4
4,5
6,6
7,8
9
8-Page Runs

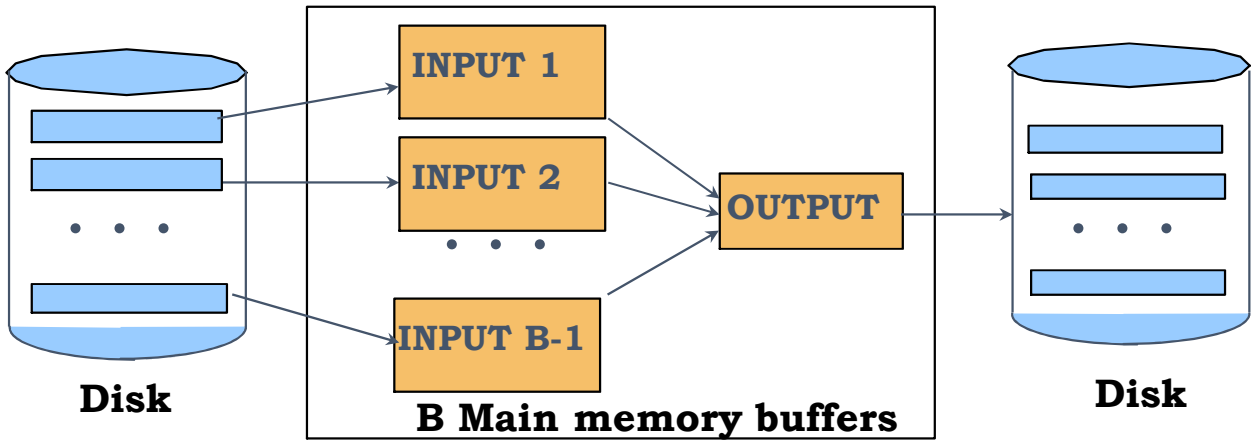
Two-Way Merge Sort – I/O Cost Analysis

Suppose the number of pages in the input file is 2^k

- How many runs are produced in Pass 0?
- How many runs are produced in Pass 1?
- How many runs are produced in Pass i?
- How many runs are produced in Pass k?
- For N number of pages, how many passes are incurred?
- How many pages do we read and write in each pass?
- What is the overall cost?

B-Way Merge Sort

- Sort a file with N pages using B buffer pages
  - Pass 0: use B buffer pages and sort internally
    - This will produce $[N/B]$ sorted B-page runs
  - Passes 1, 2, … : use $B - 1$ buffer pages for input and the remaining page for output; do $(B - 1)$-way merge in each run



B-Way Merge Sort – I/O Cost Analysis

- I/O Cost =
- Number of passes =

- Assume the previous example (i.e., 8 pages), but using 5 buffer pages (instead of 2)
  - Then, I/O cost =

- Therefore, increasing the number of buffer pages minimizes the number of passes and accordingly the I/O cost!

Number of passes of B-way mergesort

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

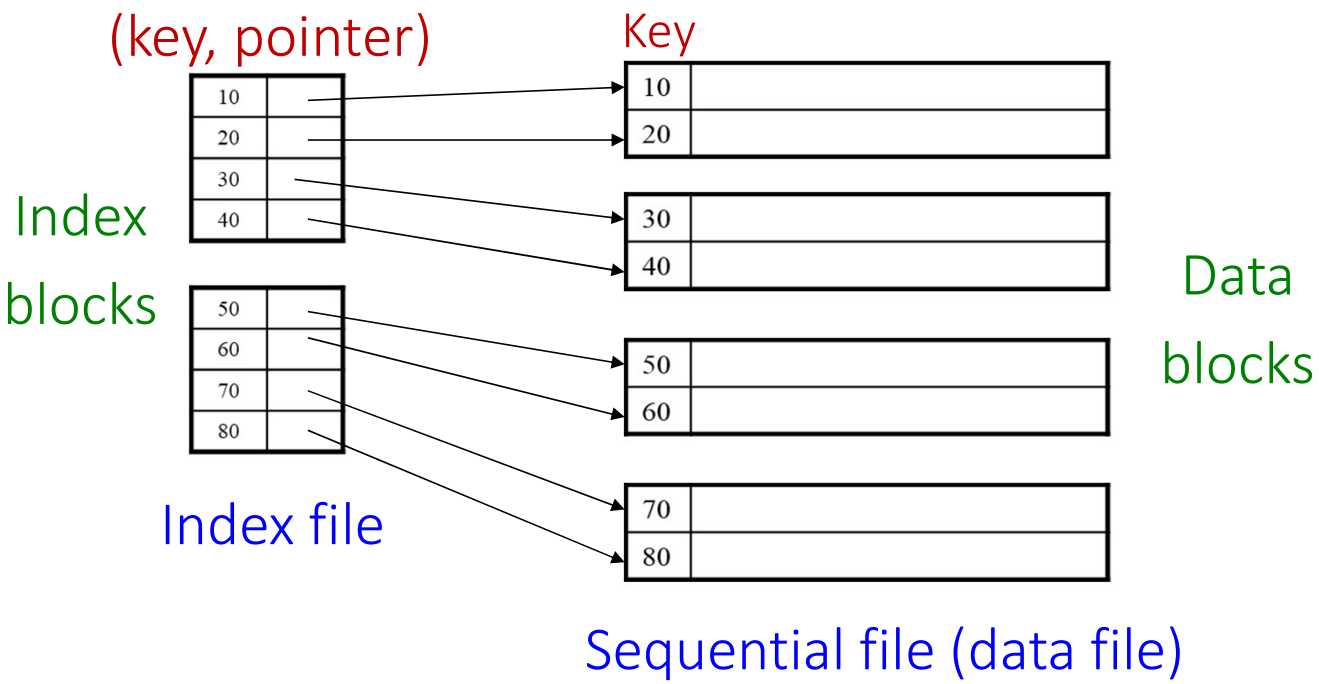
Indexing (intro)

Several different kinds of indexes used in DBMS

- Clustered / Unclustered
  - Clustered = records sorted in the search key order
  - Unclustered = no
- Dense / Sparse
  - Dense = each record has an entry in the index
  - Sparse = only some records have
- Primary / Secondary
  - Primary = on the primary key(s)
  - Secondary = on any specified key(s)

Dense Indexes on a Sequential Data File

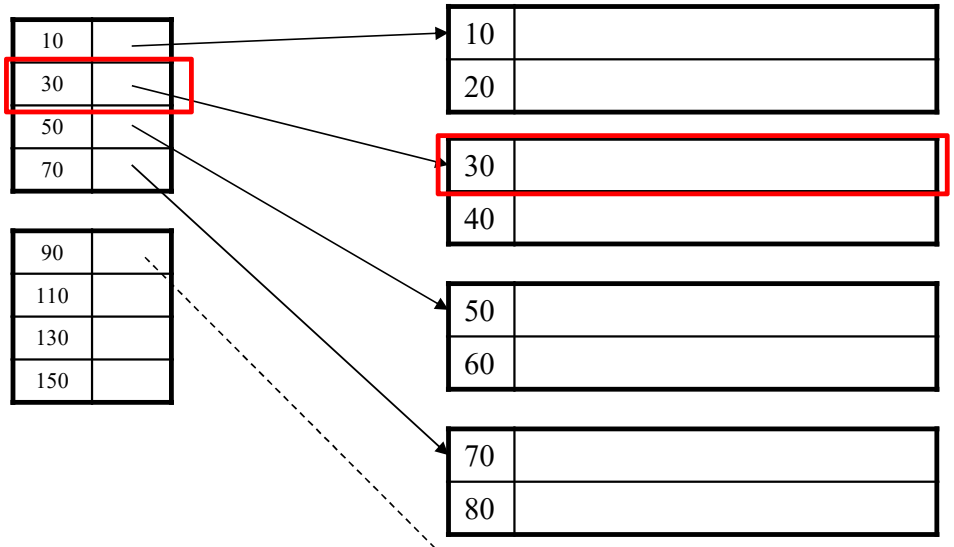
- $(key, pointer)$ pair for every record
- File is sorted by the primary key



Sparse Indexes on a Sequential Data File

- **Sparse** index: one key per data block
- Use *less* space, but takes *more* time for search
- Only work with sequential files

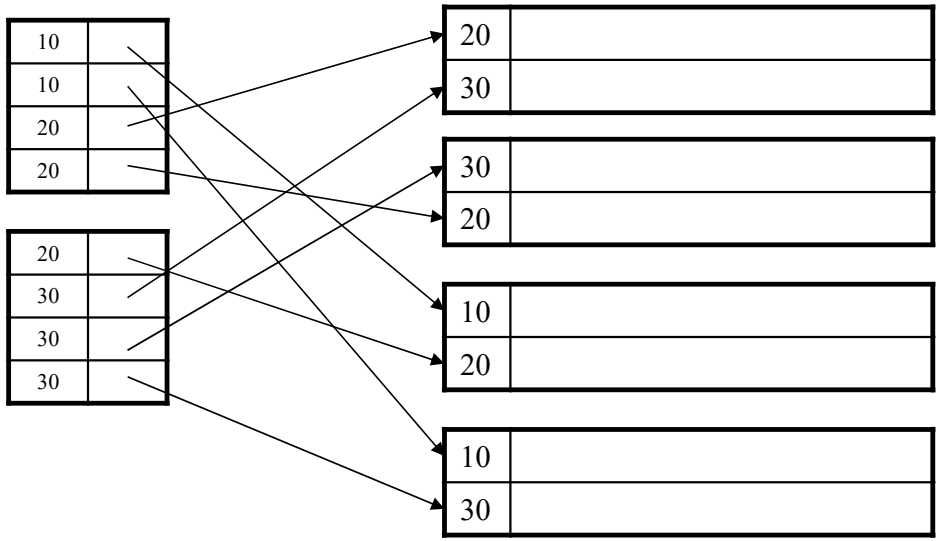
Search the sparse index for the largest key less than or equal to K



Sequential file (data file)

Unclustered Indexes

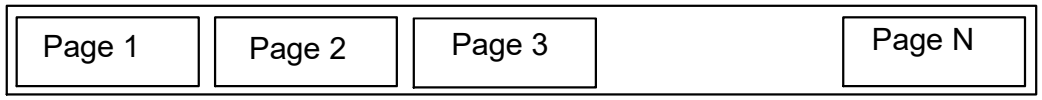
- Pointers of one index block can go to many different data blocks, in stead of a few consecutive blocks.
- Typically used to index other attributes than primary keys.



Tree Structured Index

Issues and Alternatives

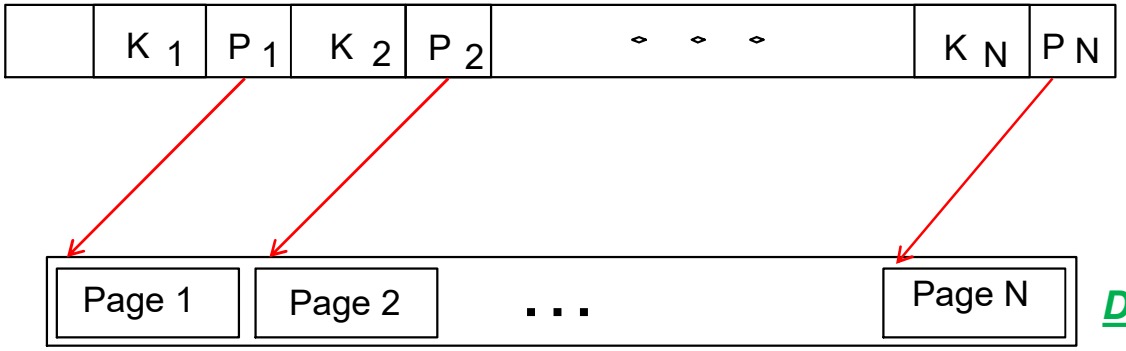
- How can we answer a ***range query***?
  E.g., *"List all students with a GPA ≥ 3.0"*



Data File

Say data is already sorted by GPA

- What about doing a *binary search* followed by a *scan*?
- What about creating an **index file** and do binary search there?



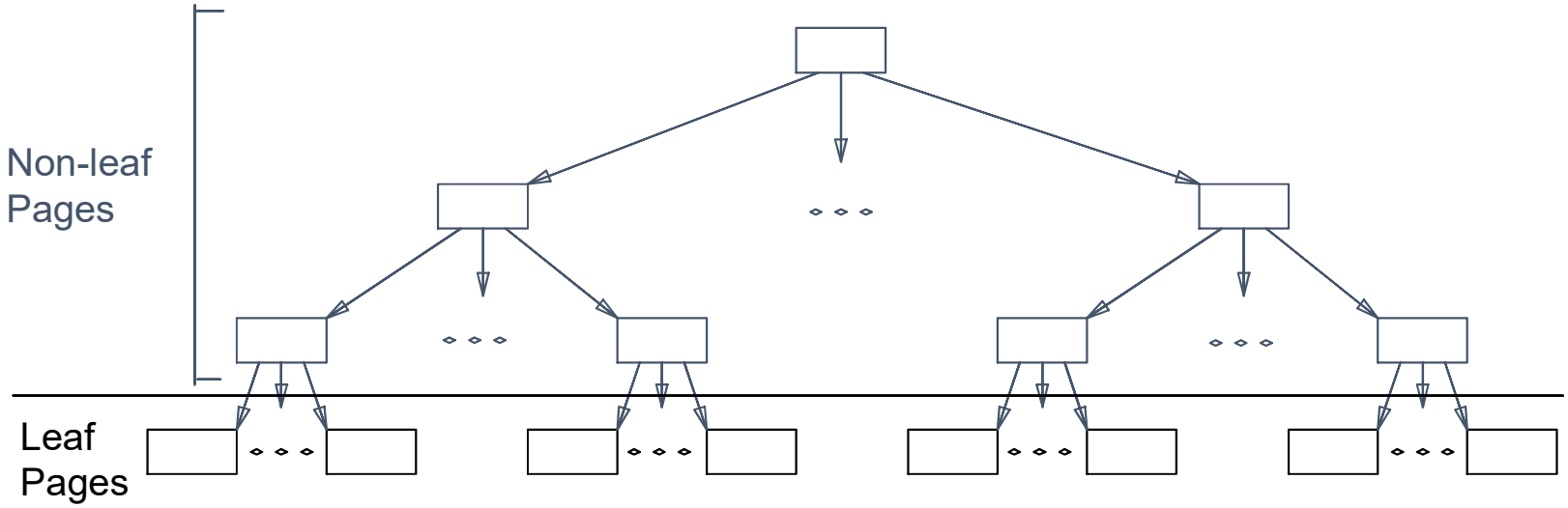
Index File

Problem: This index can still be very large.

Data File

General Strategic Plan

- Index of indices. Recursion to our rescue!



Where to store data records?

- In general, 3 alternatives for "data records" (each referred to as K^*) can be pursued:

Alternative (1): Leaf pages contain the actual data (i.e., the data records)

Alternative (2): Leaf pages contain the $\langle key, rid \rangle$ pairs and actual data records are stored in a separate file

Alternative (3): Leaf pages contain the $\langle key, rid{-}\text{list} \rangle$ pairs and actual data records are stored in a separate file

The choice among these alternatives is orthogonal to the *indexing technique*

Clustered VS. Unclustered Indexes

- Is an index that uses Alternative (1) clustered or un-clustered?

  Clustered

- Is an index that uses Alternative (2) or (3) clustered or un-clustered?

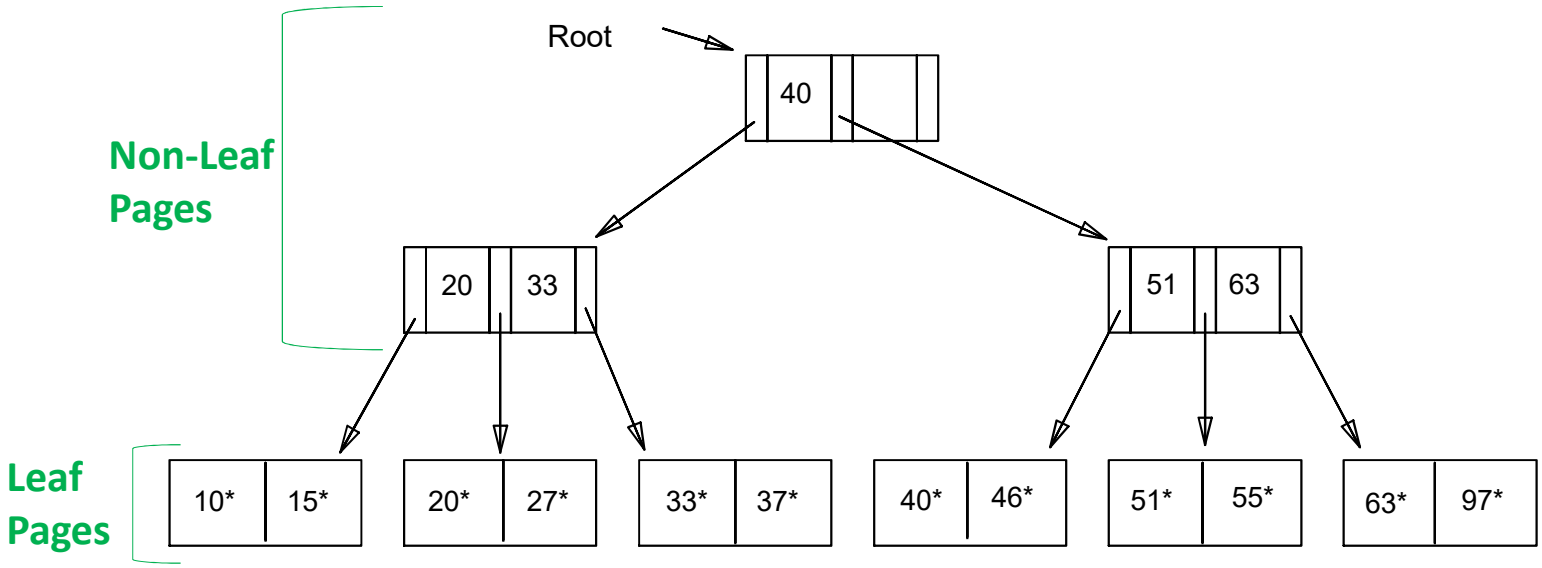
  Clustered “only” if data records are sorted on the search key field

- In practice:
  - A clustered index is an index that uses Alternative (1)
  - Indexes that use Alternatives (2) or (3) are un-clustered

Static Trees

ISAM Tree

- **Indexed Sequential Access Method** (ISAM) trees are *static*



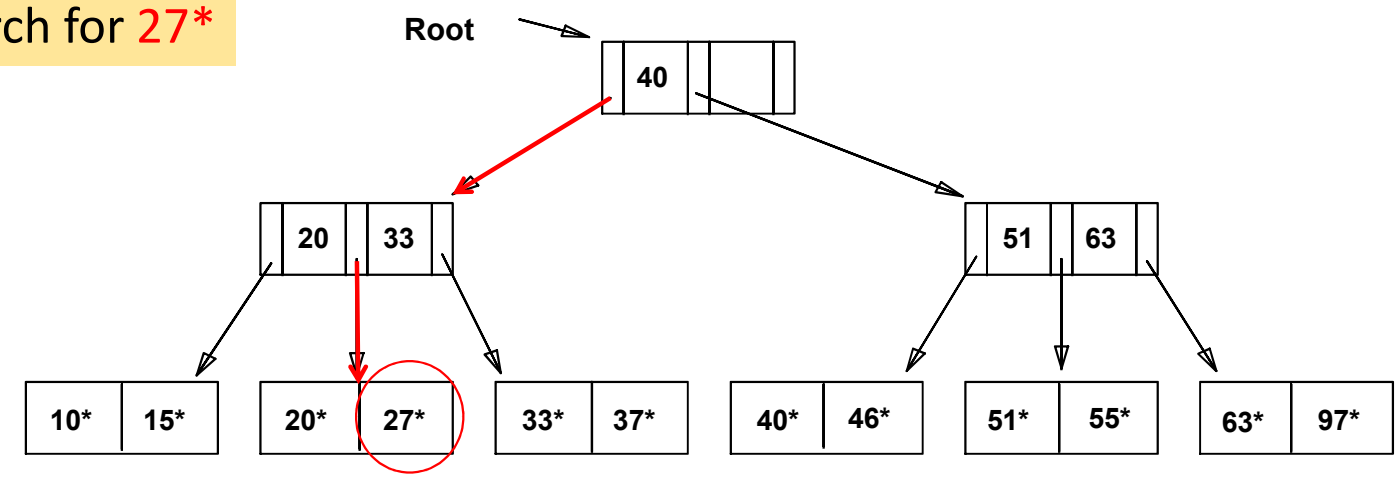
E.g., 2 Entries Per Page

Data pages are allocated *sequentially* and *sorted* on the search key value

ISAM Trees: Search

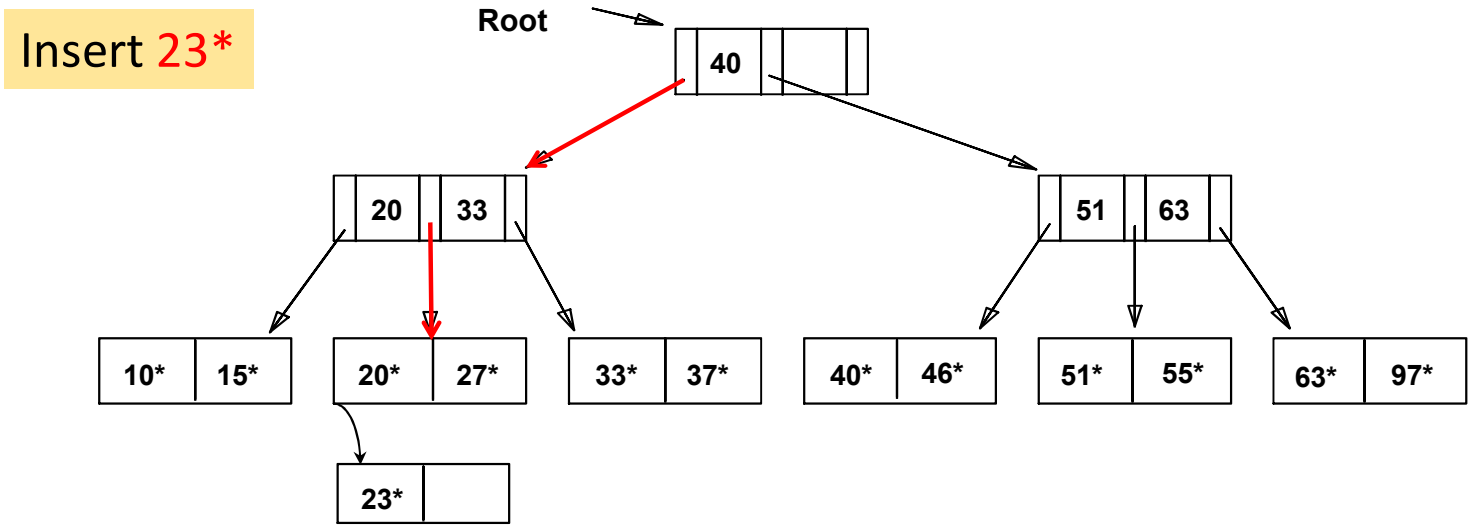
- Search begins at root, and key comparisons direct it to a leaf

Search for 27*



ISAM Trees: Insert

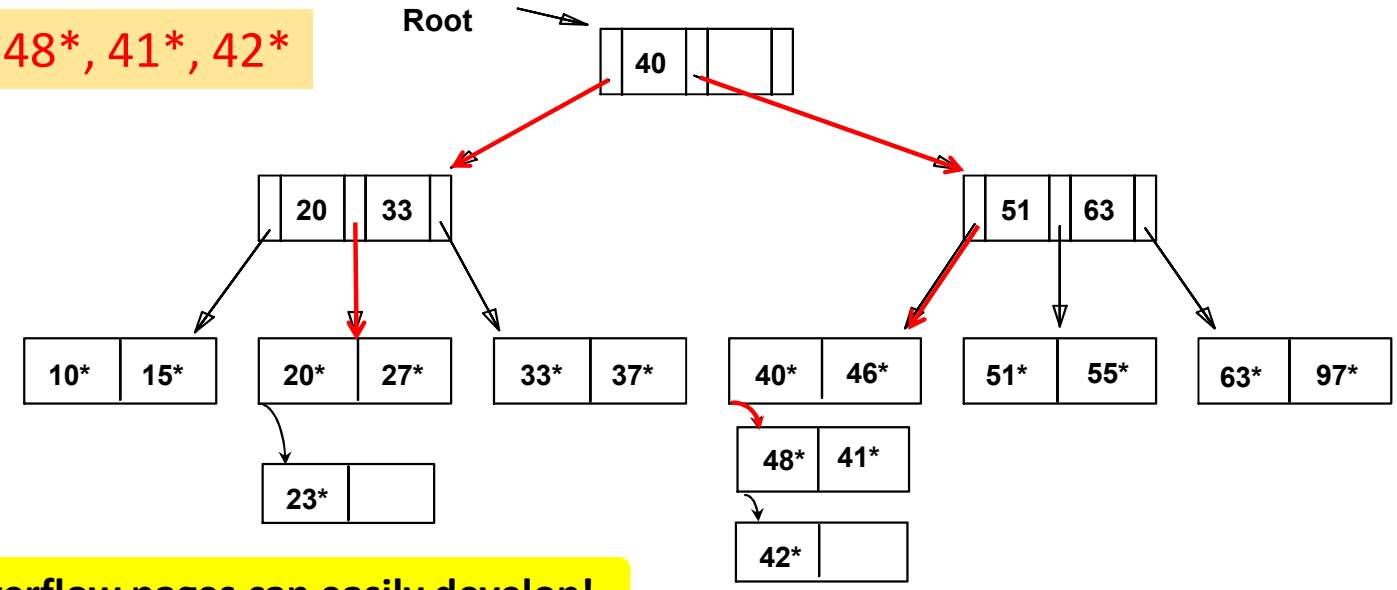
- The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)



ISAM Trees: Insert

- The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

Insert 48*, 41*, 42*

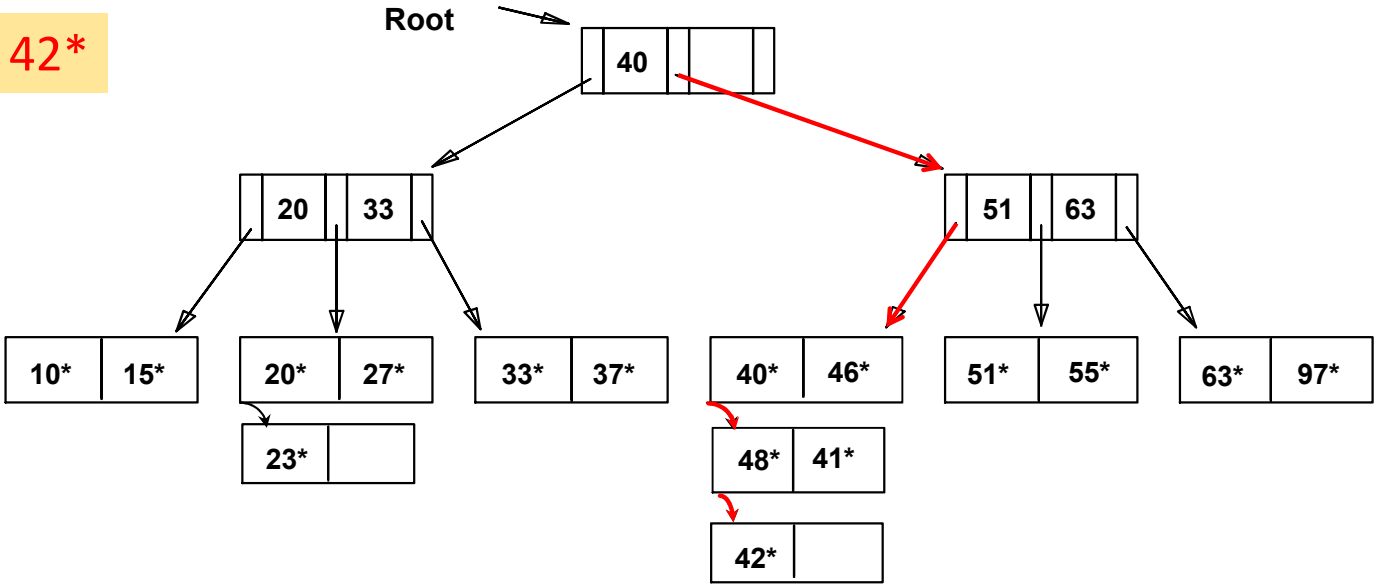


Chains of overflow pages can easily develop!

ISAM Trees: Delete

- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

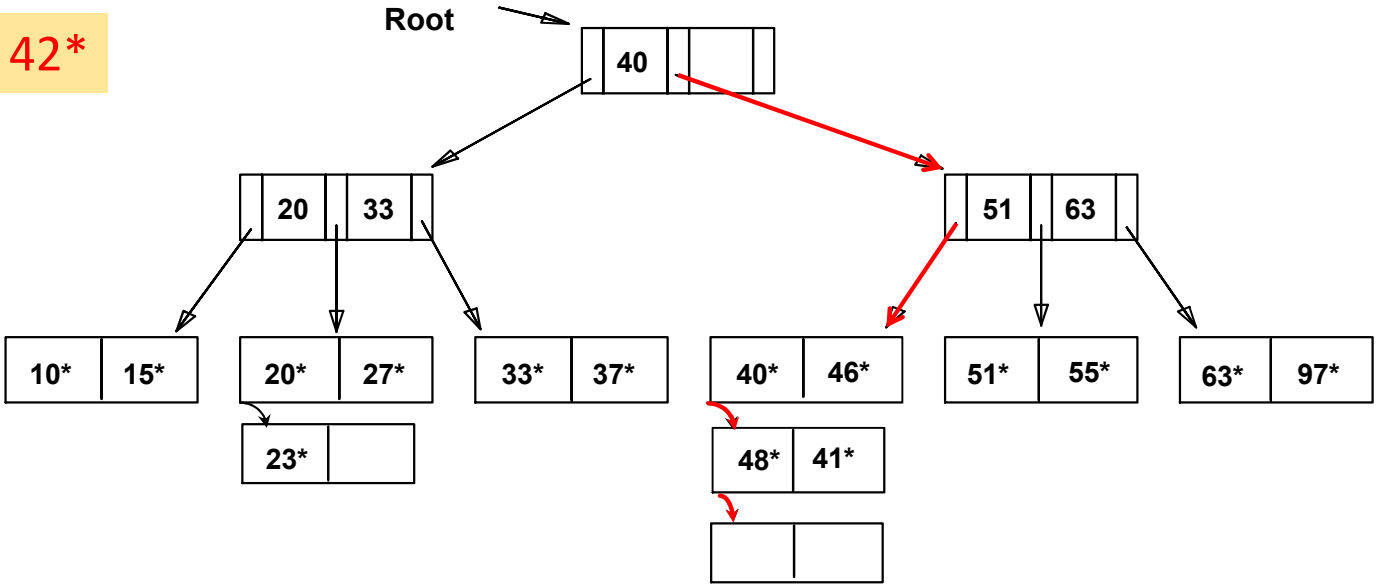
Delete 42*



ISAM Trees: Delete

- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

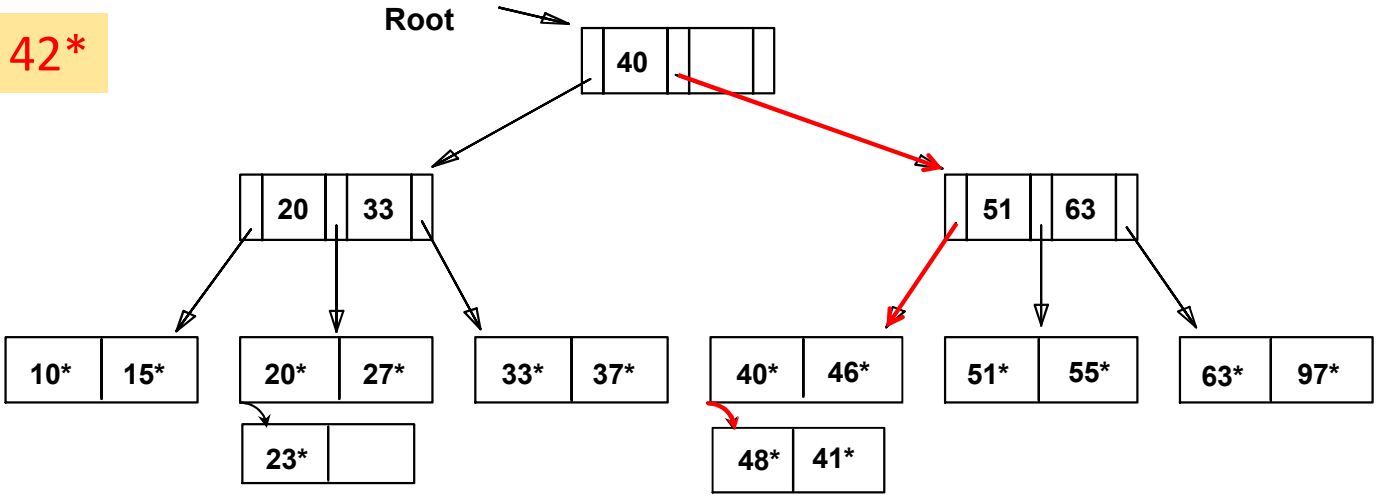
Delete 42*



ISAM Trees: Delete

- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

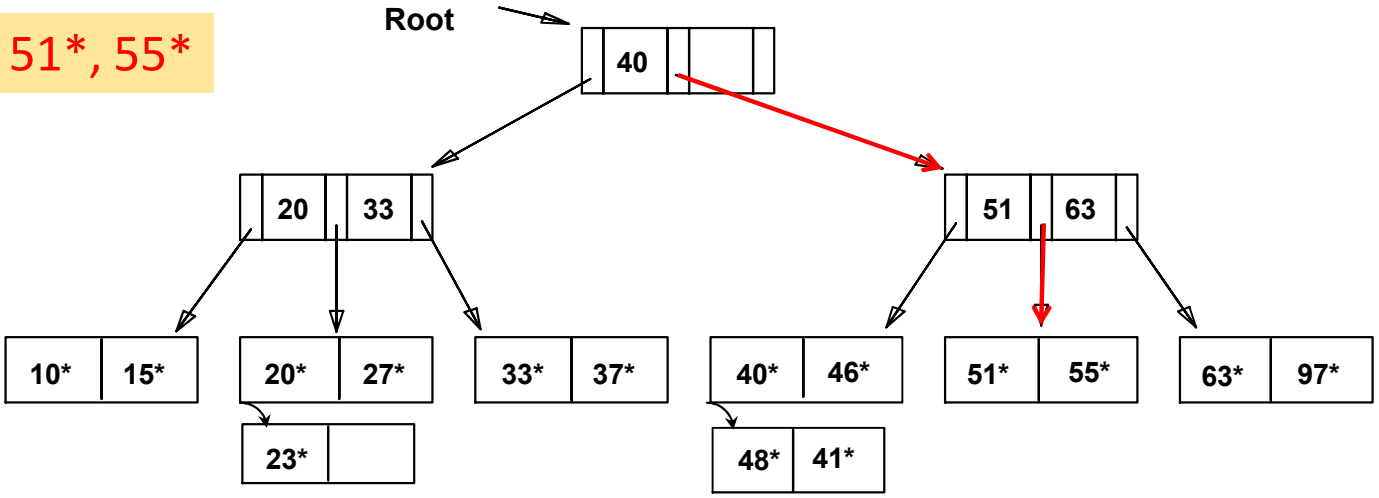
Delete 42*



ISAM Trees: Delete

- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

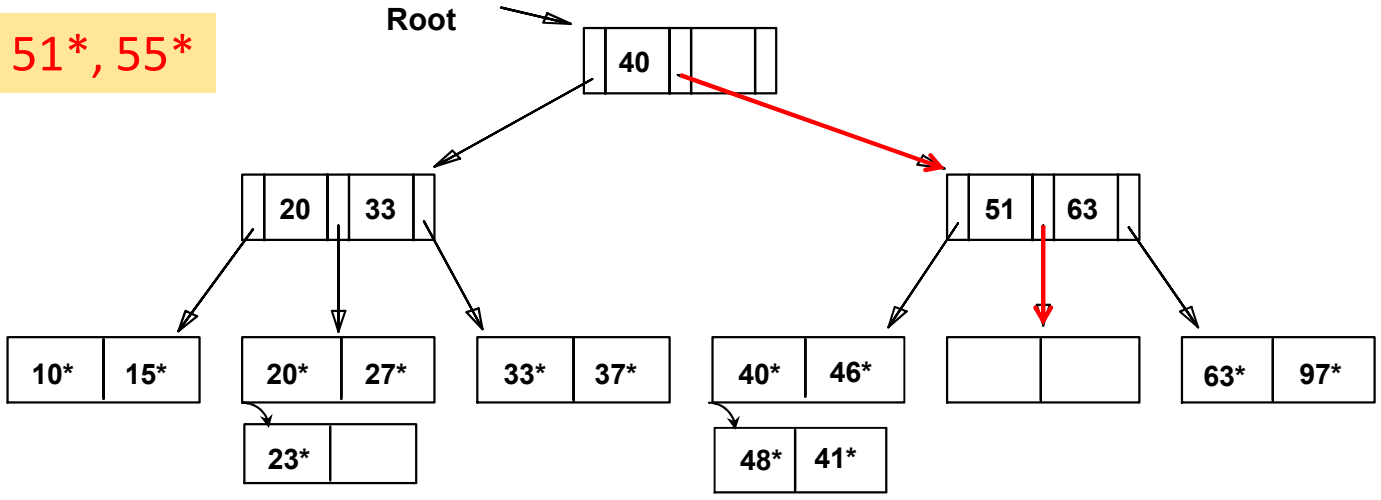
Delete 51*, 55*



ISAM Trees: Delete

- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

Delete 51*, 55*



Primary pages are NOT removed, even if they become empty!

ISAM: pros & cons

- Once an ISAM file is created, insertions and deletions affect *only* the contents of leaf pages (i.e., **ISAM is a static structure!**)
- Since index-level pages are never modified,
  there is no need to lock them during insertions/deletions
  - Critical for concurrency!
- Long overflow chains can develop easily
  - The tree can be initially set so that ~20% of each page is free
- If the data distribution and size are relatively static,
  ISAM might be a good choice to pursue!

Dynamic Trees

More Dynamic Structure

- ISAM indices are static

  Long overflow chains can develop as the file grows, leading to poor performance

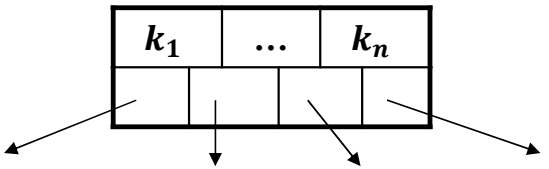
- This calls for more flexible, *dynamic* indices that adjust gracefully to insertions and deletions

  No need to allocate the leaf pages sequentially as in ISAM

- Among the *most successful* dynamic index schemes is **B-tree**/**B+ tree**

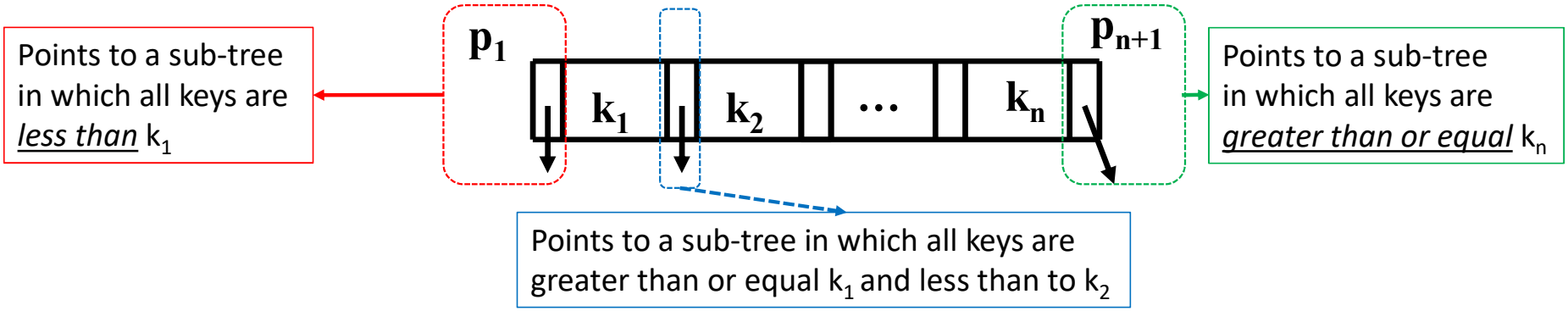
B+ Tree – Basics

Another popular picture of B+Tree nodes:



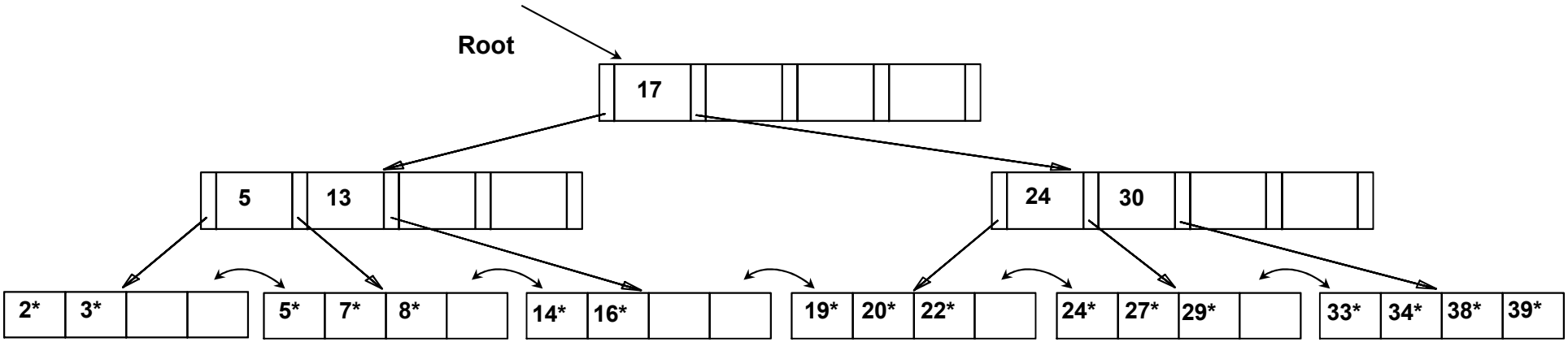
Each node in a B+ tree of order d

- d is a measure of the capacity of a tree
- has $\lceil d/2 \rceil \leq k \leq d$ keys (except root which may have just 1 key)
- has exactly $k + 1$ pointers
- all leaves are on the same level



- Each leaf also has *another pointer* that point to **next leaf** in the tree.

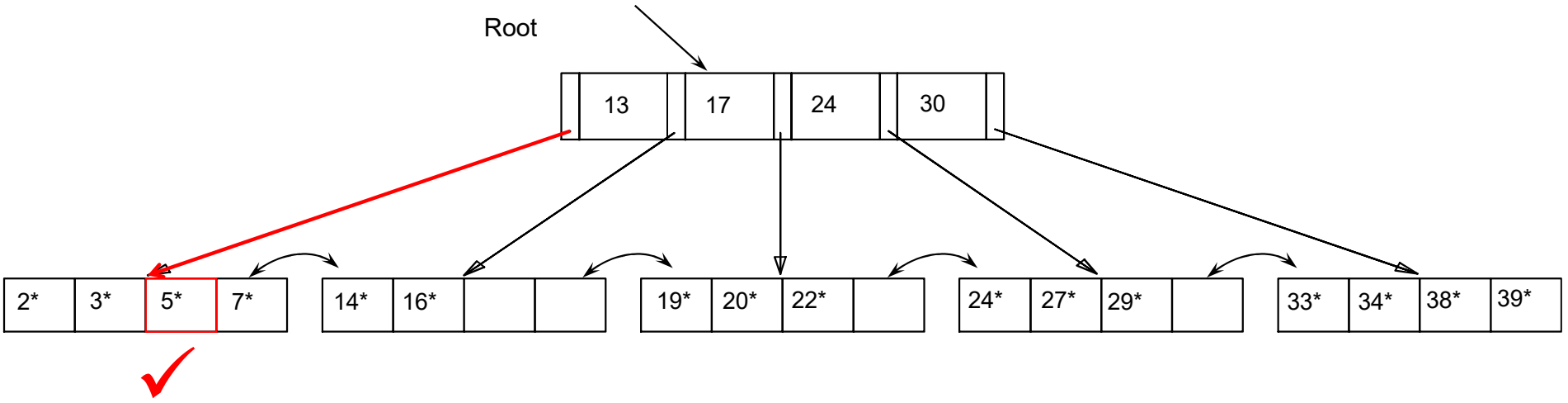
What B+Tree may look like …



B+ Tree: Searching for Entries

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

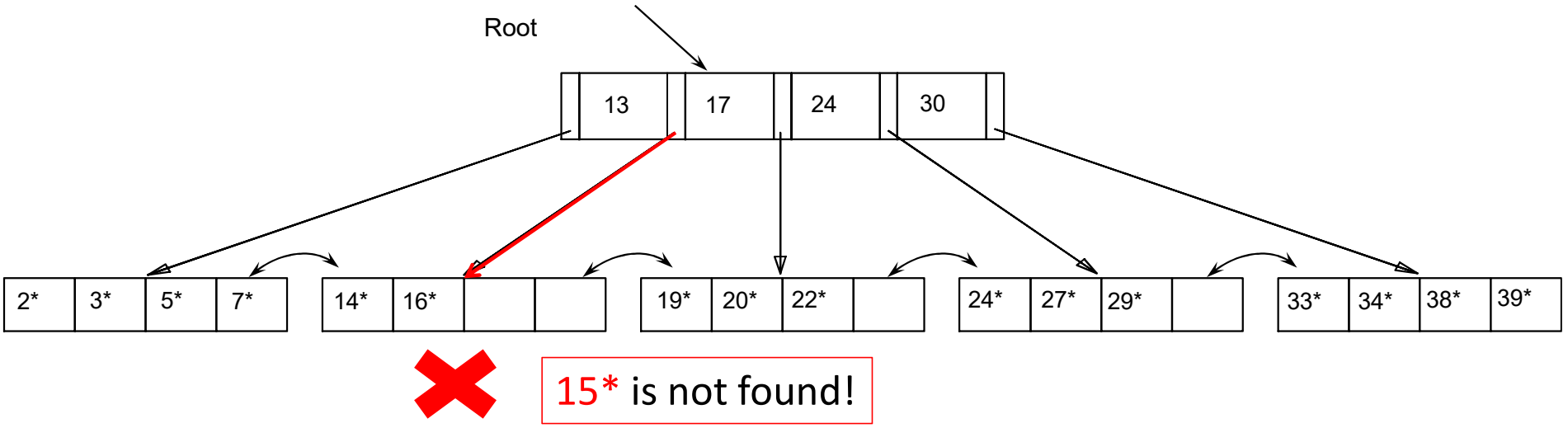
Example 1: Search for entry 5*



B+ Tree: Searching for Entries

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

Example 2: Search for entry 15*



Queries on a B+Tree of order k

- If there are n search-key values in the file,
  the height of the tree is no more than $\lceil \log_k n \rceil$.
- A node is generally the same size as a disk block, typically 4 kB
  and k is typically around 100 (40B/index entry).
- With 1 million search key values and $k = 100$,
  at most $\log_{100} 10^6 = 3$ nodes are accessed in a lookup traversal from root to leaf.
- Contrast this with a balanced binary tree with 1 million search key values
  where 20 nodes are accessed in a lookup

This difference is significant since every node access may need a disk I/O.

B+ Trees: Inserting Entries

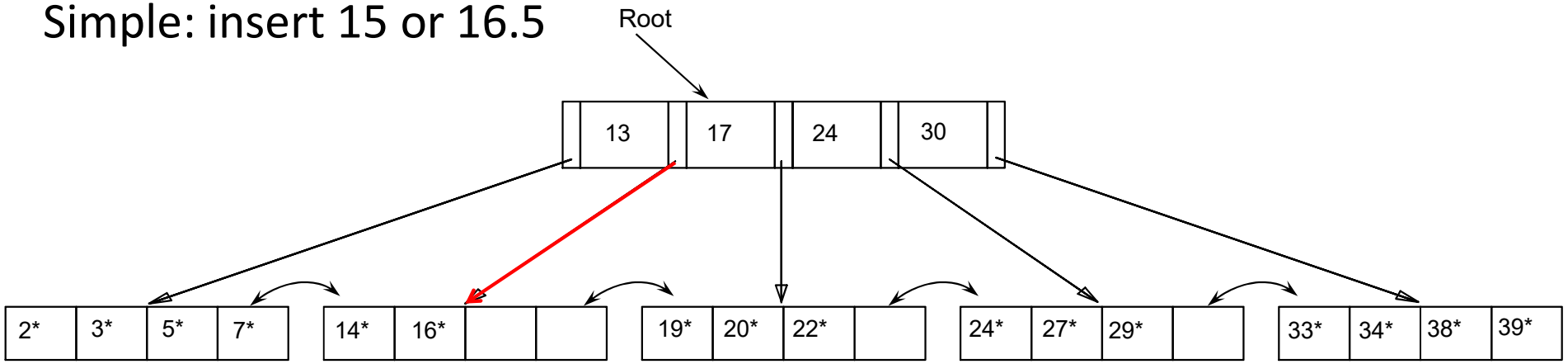
- Find correct leaf ***L***

- Put data entry onto (leaf) ***L***
  - If ***L*** has enough space, *done*!
  - Else, ***split*** ***L*** *into* ***L*** *and a new node* L_2
    - Re-partition entries *evenly*, ***copying up*** the middle key

- Internal/Parent node may *overflow*
  - ***Push up*** middle key (splits "grow" trees; a root split increases the height of the tree)

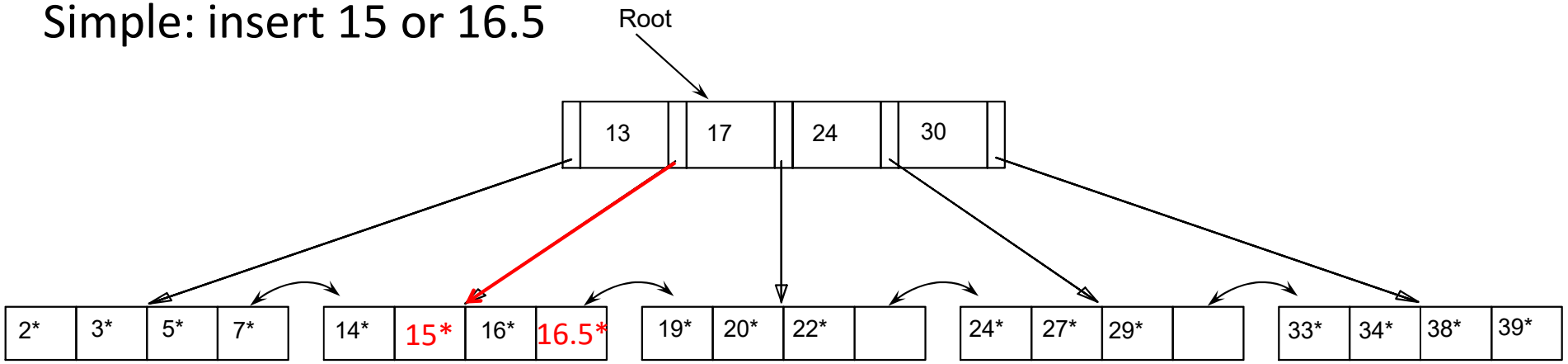
B+ Tree: Examples of Insertions

Simple: insert 15 or 16.5



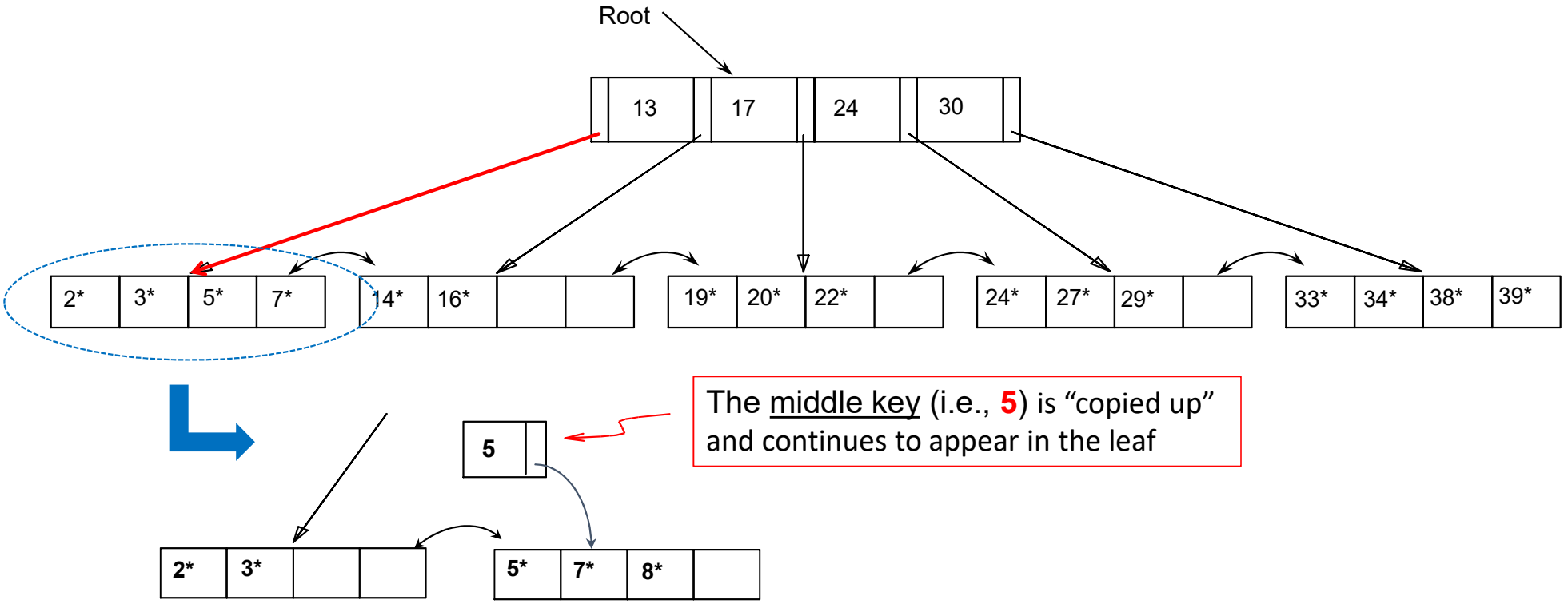
B+ Tree: Examples of Insertions

Simple: insert 15 or 16.5



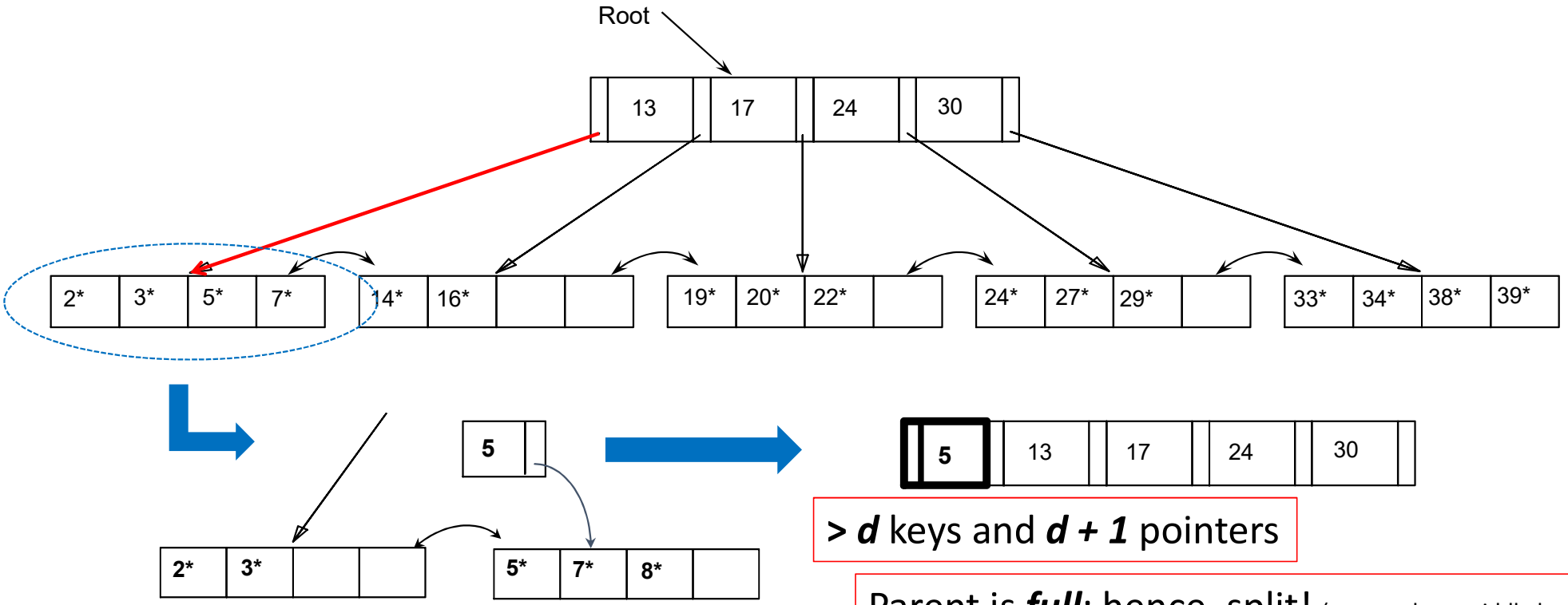
B+ Tree: Examples of Insertions

More interesting, insert entry 8*



B+ Tree: Examples of Insertions

More interesting, insert entry 8*



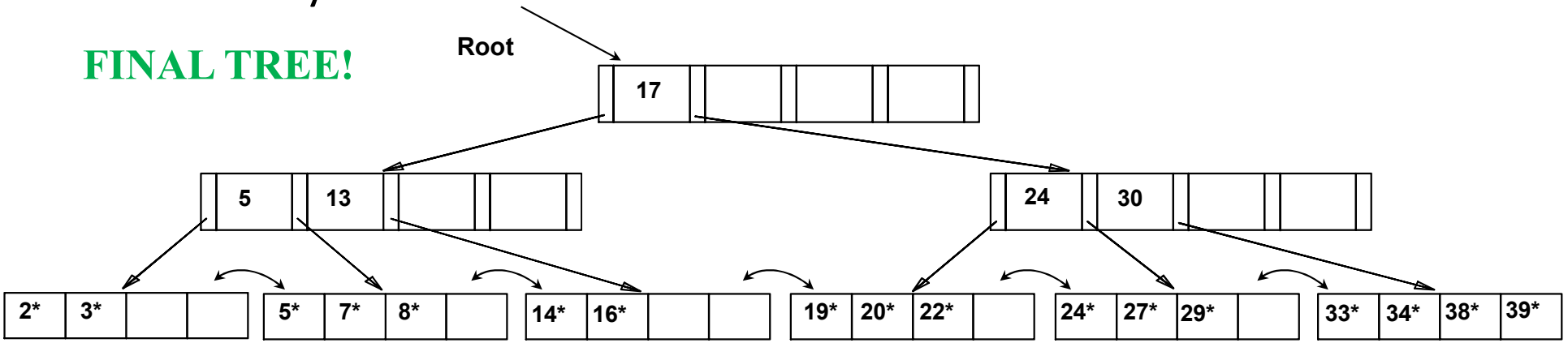
> *d* keys and ***d + 1*** pointers

Parent is ***full***; hence, split! (e.g., push up middle key)

B+ Tree: Examples of Insertions

- Insert entry 8*

FINAL TREE!



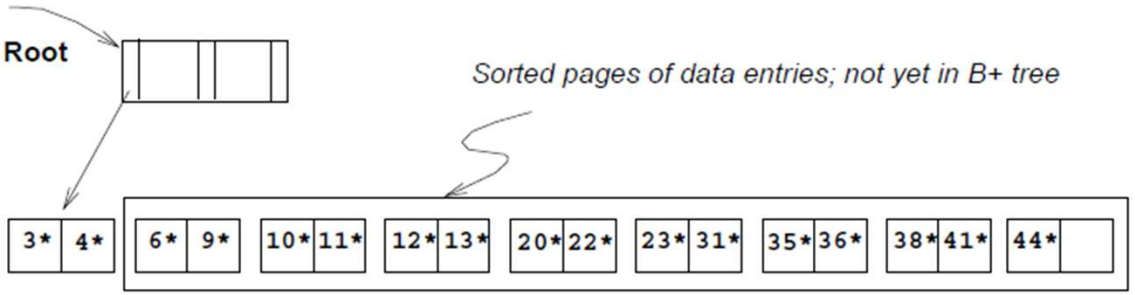
Splitting the root lead to an increase of height by 1!

B+ Trees: Deleting Entries

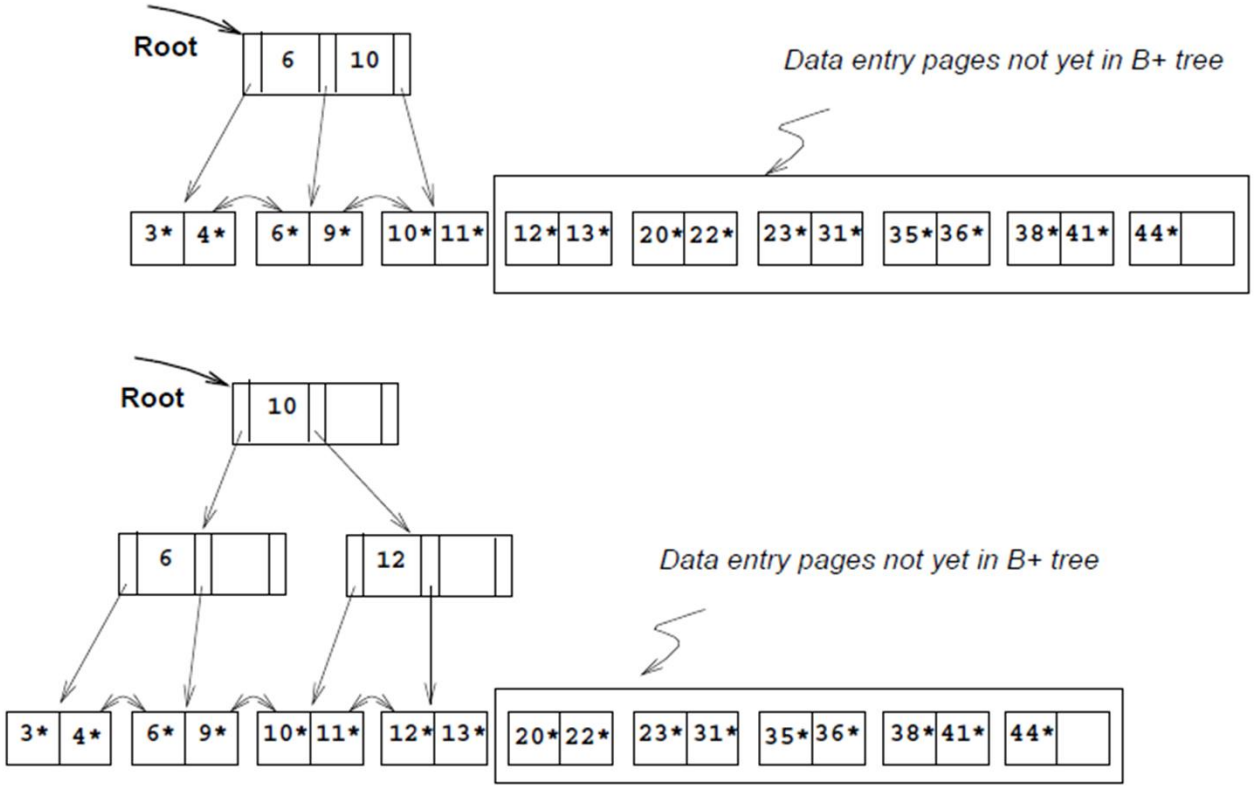
- Start at root, find leaf ***L*** where entry belongs
- Remove the entry
  - If ***L*** is at least half-full, *done!*
  - If ***L*** *underflows*
    - Try to re-distribute (i.e., borrow from a "rich sibling" and "copy up" its *lowest key*)
    - If re-distribution fails, *merge* ***L*** and a "poor sibling"
      - Update parent
      - And possibly merge, recursively

Bulk Loading of a B+Tree

- If we have a large collection of records and we want to create a B+tree on some filed, doing so by repeatedly inserting records is *very slow*! - HW2.1???
- **Bulk loading** can be done much more efficiently
- *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



Bulk Loading of a B+Tree (cont.)

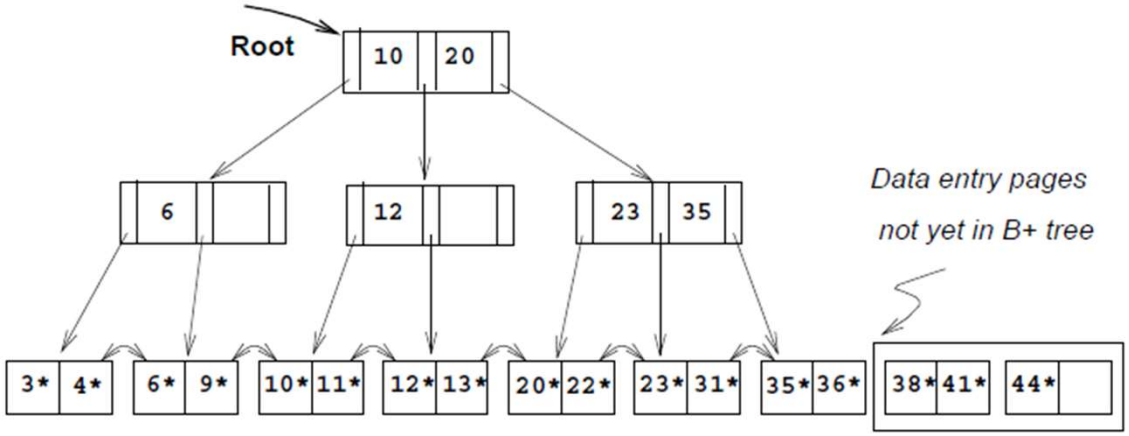


Bulk Loading of a B+Tree (cont.)

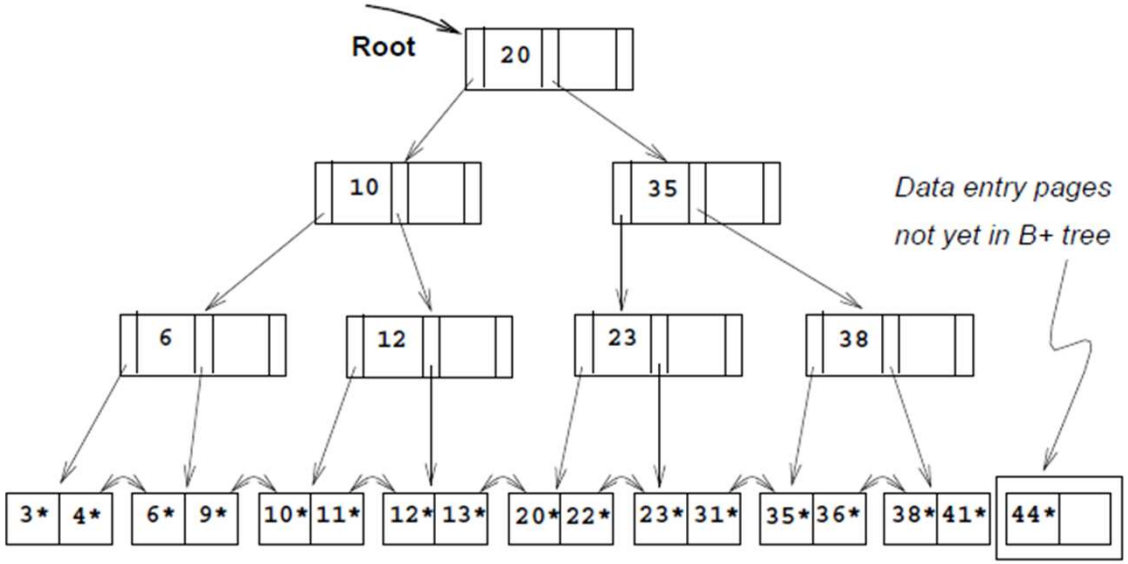
- Index entries for leaf pages will enter into right-most index page just above leaf level.
  When this fills up, it splits.

- Much faster than repeated inserts.

Bulk Loading of a B+Tree (cont.)



Bulk Loading of a B+Tree (cont.)



Advantage of Bulk Loading

- Recall: Multiple insertions:
  - Slow
  - May not give sequential store of leaves

- **Bulk Loading**:
  - Fewer I/O operations during build (insert entire page at a time is possible)
  - Leaves will be stored sequentially
  - Can control page occupancy factor