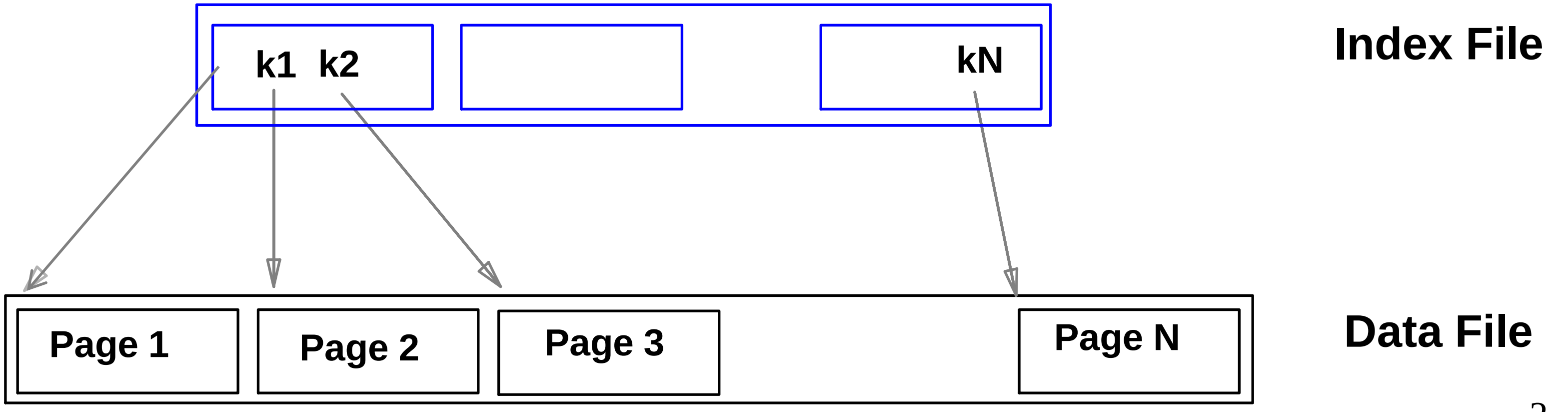
Tree-Structured Indexing

Michael Yu

2024-2025 T1

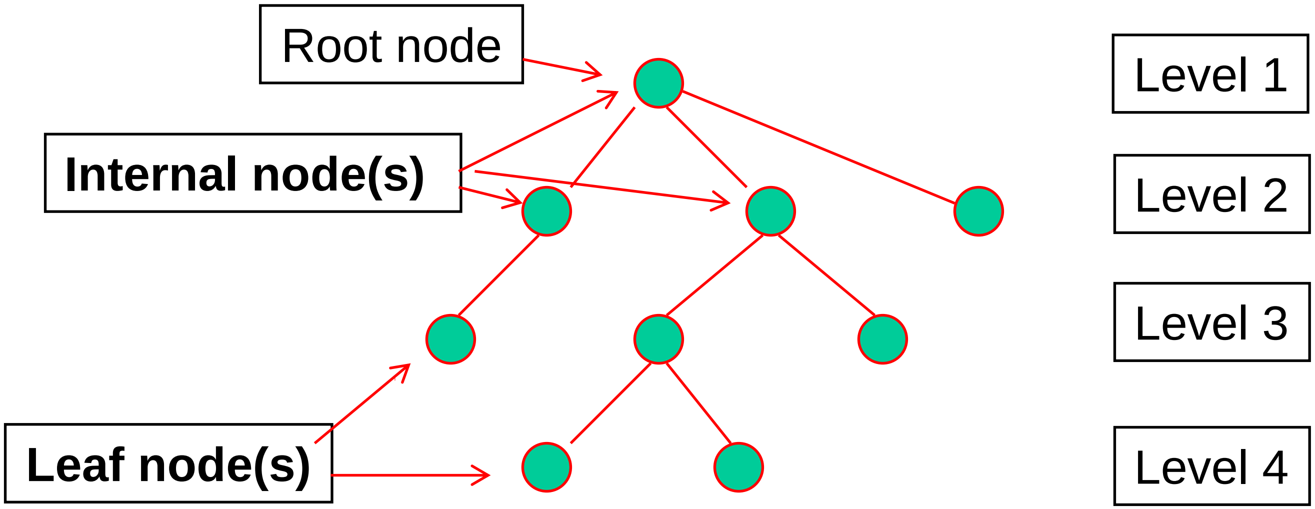
Range Searches

- "Find all students with 3.0 < gpa < 3.5"
  - If records are sorted via gpa…
  - Do a binary search to find first such student, then scan to find others.
- Limitation: Cost of binary search can be quite high
  - Motivation idea: Create an 'index' file, and *do binary search on (much smaller) index file!*



B+ Tree: Most Widely Used Index

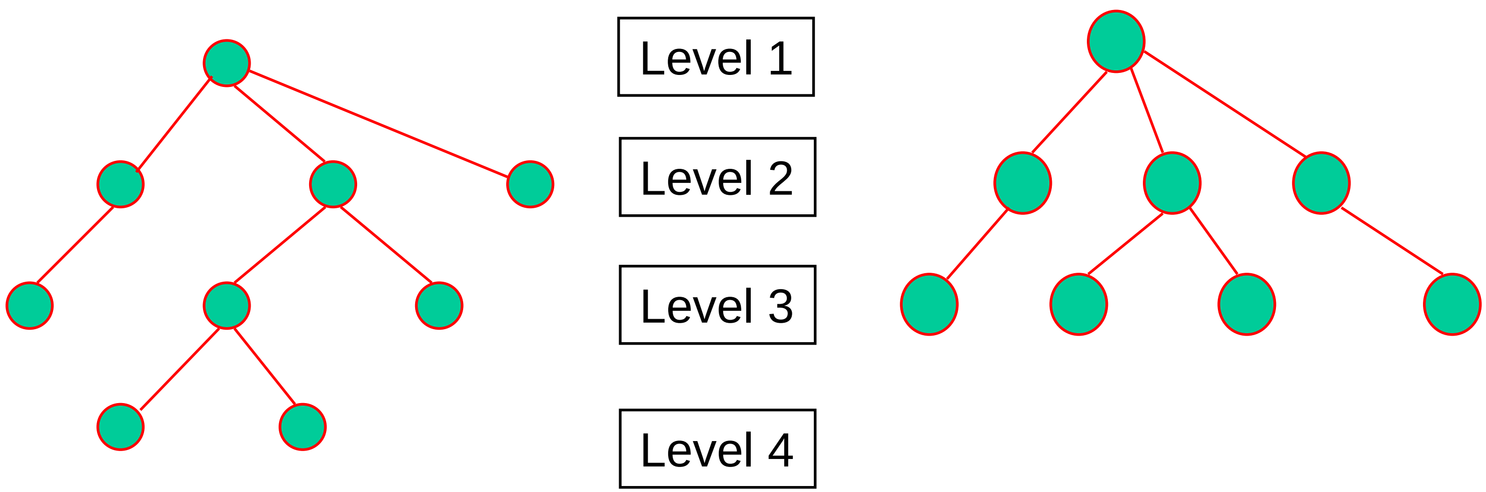
- B+ Tree is a tree (See below for concept)



If a higher level node is connected to a lower level node, then the higher level node is called a parent (grandparent, ancestor, etc.) of the lower level node

B+ Tree: Most Widely Used Index

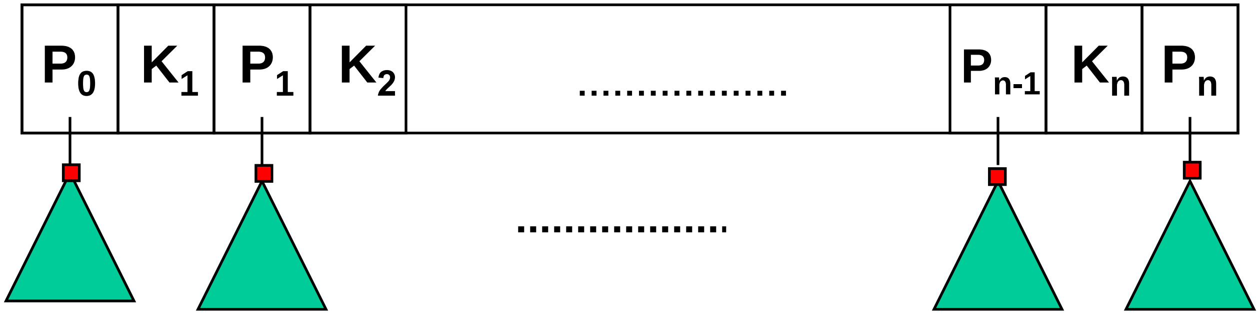
- B+ trees are all Balanced trees: all leaves at the same **level/height**
  - Height of a node: its distance to the root
  - Level of a node: height + 1



B+ Tree: Most Widely Used Index

- Structure of a B+ tree:
  - It is ***balanced:*** all leaf nodes at the same level
  - It's nodes has a special structure
    - ***Internal nodes (includes root node)***
    - ***Leaf nodes***

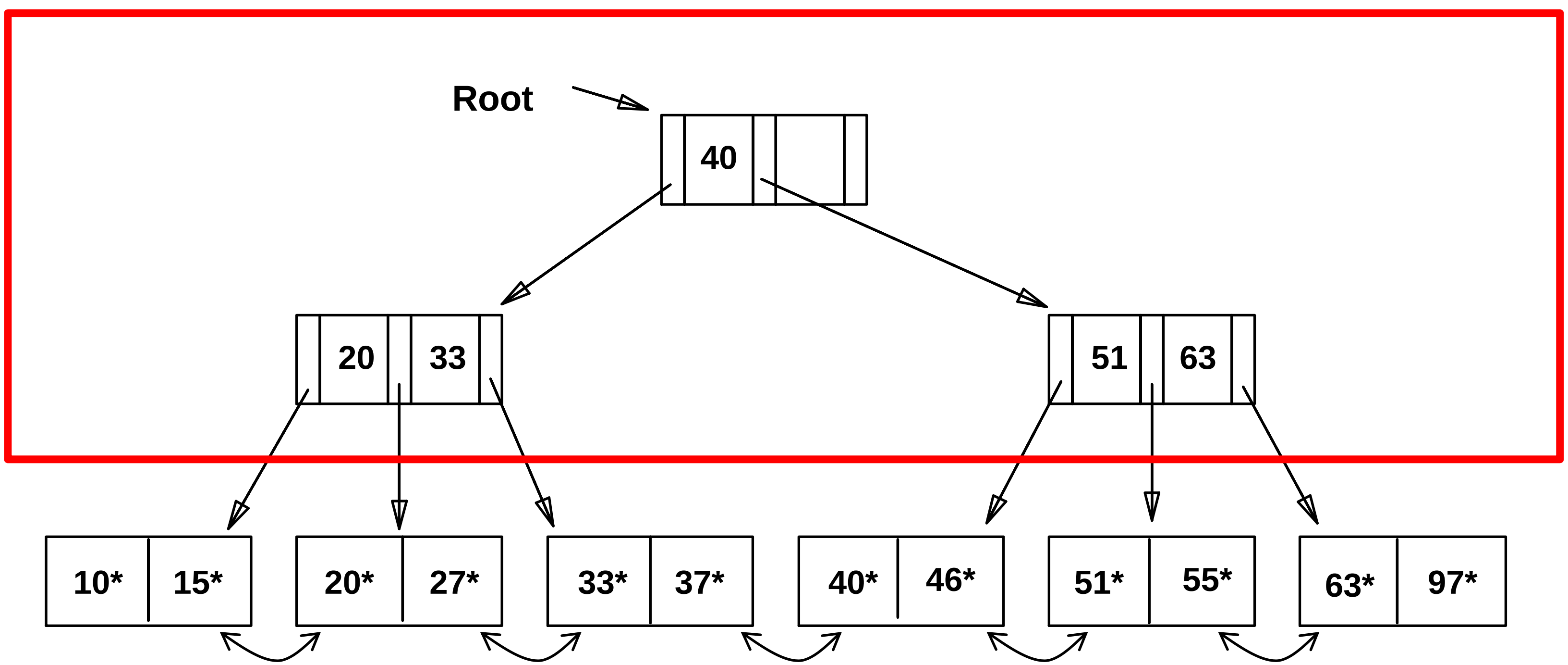
B+ tree: Internal nodes



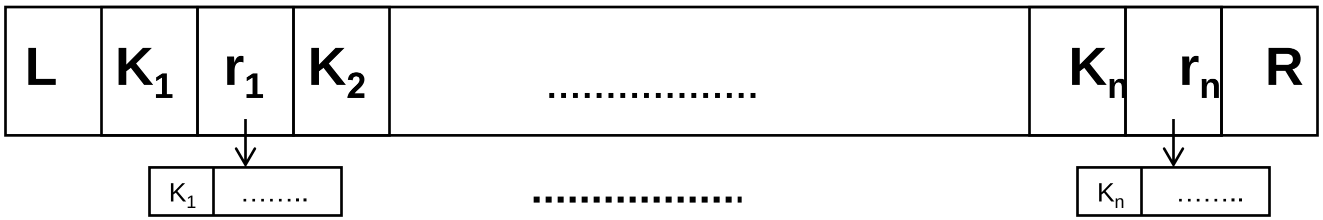
Each P_i is a pointer to a child node, each K_i is a search key value
Pointers outnumber search key values by *exactly one.*

- **Requirements:**
  - $K_1 < K_2 < \ldots < K_n$
  - For any search key value K in the subtree pointed by P_i,
    - If $P_i = P_0$, we require $K < K_1$
    - If $P_i = P_1, \ldots, P_{n-1}$, we require $K_i \leq K < K_{i+1}$
    - If $P_i = P_n$, we require $K_n \leq K$

B+ tree: Internal nodes

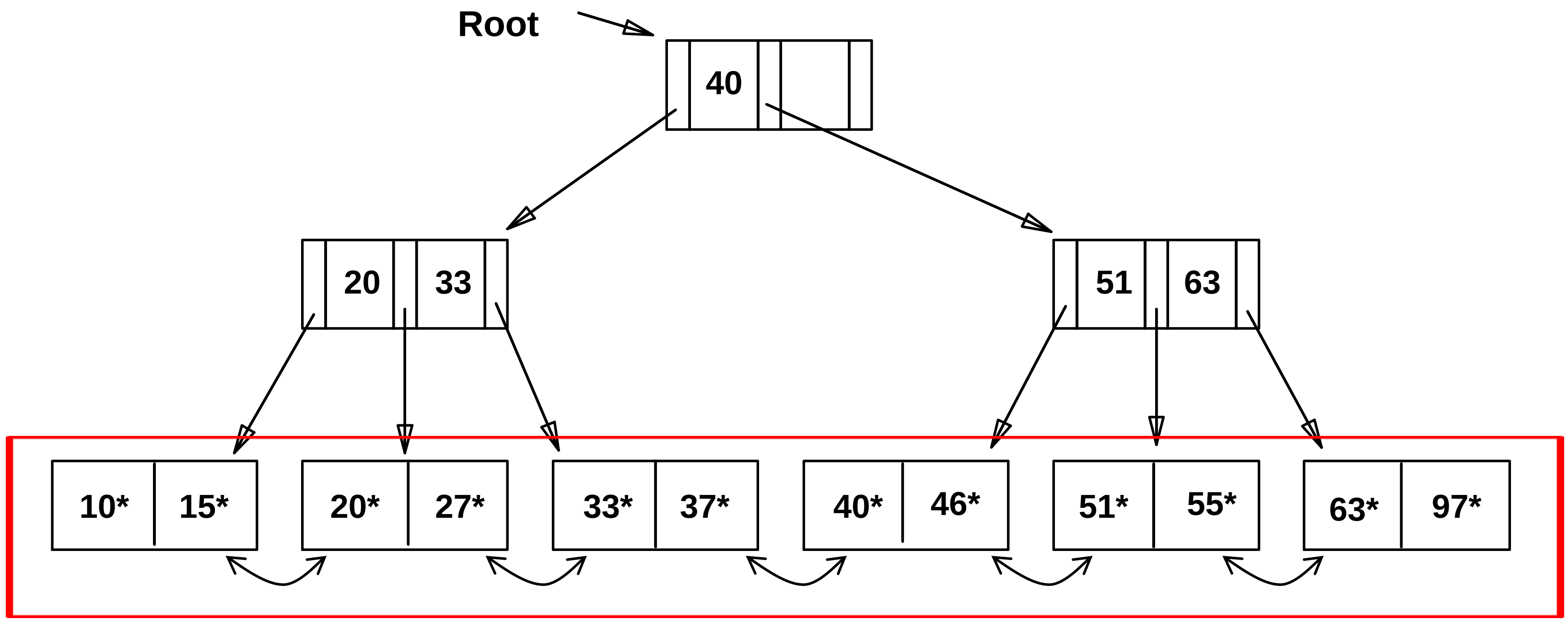


B+ tree: Leaf node



- Each r_i is a pointer to a **record** that contains search key value K_i ...
- L points to the left neighbor, and R points to the right neighbor
- $K_1 < K_2 < \ldots < K_n$
- We require $d \leq n \leq 2d$ where d is the order of this B+ tree
- We will use K_i* for the pair K_i, r_i and omit L and R for simplicity

B+ tree: <u>Leaf</u> node

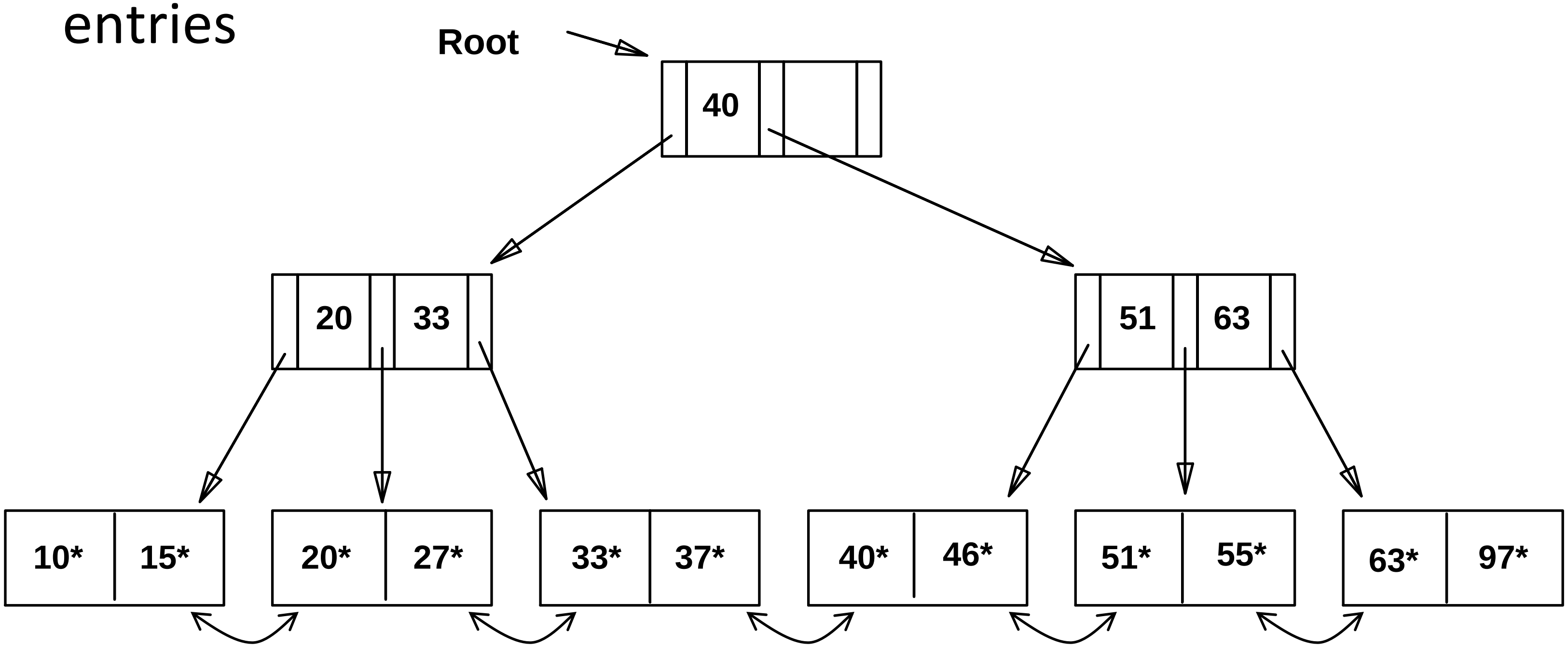


Order of B+ Tree

- **The number d is the order of a B+ tree**.
  - If the node is not the root, we require $d \leq n \leq 2d$ where d is a pre-determined value for this B+ tree, called its ***order***
  - If the node is the root, we require $1 \leq n \leq 2d$
  - However, it's technically possible for leaf nodes to temporarily end up with < d entries <u>immediately</u> after you delete data, we do not consider this specific case for now.

Example: B+ tree with Order of 1

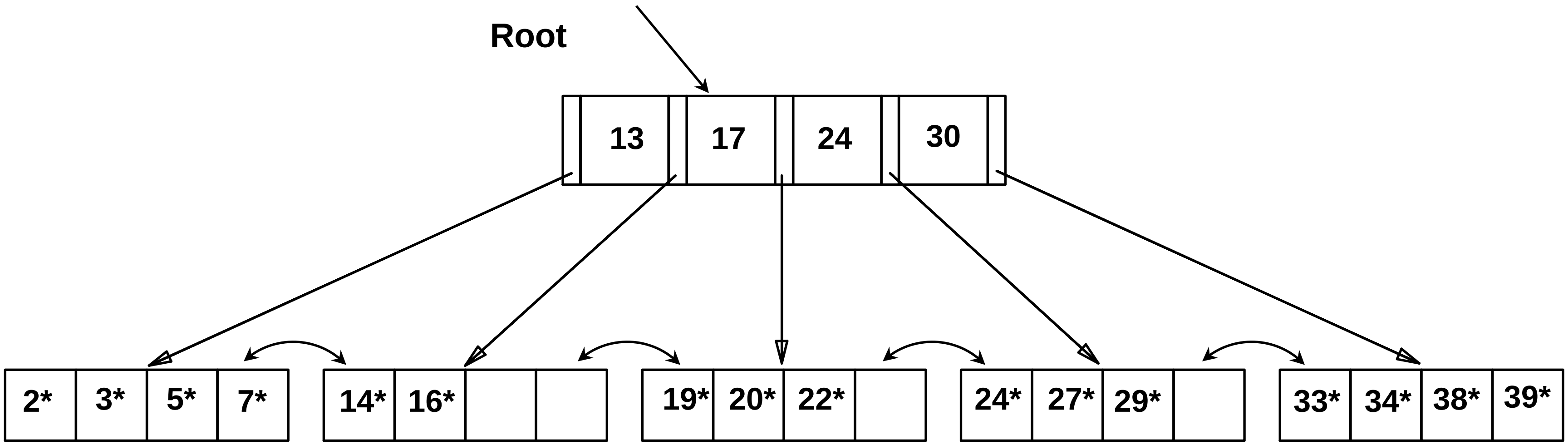
- Each node must hold at least 1 entry, and at most 2 entries



- Given search key values 27, 51, 64, how to find the rids?
  - Search begins at the root, and key comparisons direct it to a leaf

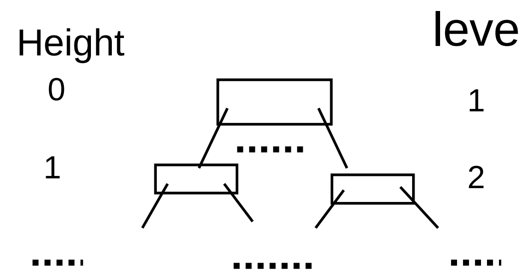
Example: B+ tree with Order 2

- Search for 5*, 15*, all data entries >= 24* ...
- The last one is a range search, we need to do the sequential scan, starting from the first leaf containing a value >= 24.



Searching a value in B+ tree - Cost

- In general nodes are **pages**
- Let **H** be the height of the B+ tree: need to read **H+1** pages to reach a leaf node
- Let **F** be the (average) number of pointers in a node (for internal node, called *fanout factor*)
  - Level 1 = 1 page = F^0 page
  - Level 2 = F pages = F^1 pages
  - Level 3 = Fx F pages = F^2 pages
  - Level H+1 = …….. = F^H pages (i.e., leaf nodes)
  - Suppose there are D data entries. So there are D/(F-1) leaf nodes
  - D/(F-1) = F^H. That is, H = $\log_F(\frac{D}{F-1})$



If the fanout factor is F, we usually assume a leaf node stores F-1 data entries.

B+ Trees in Practice

- Typically, a node is a page
- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133 (i.e, # of pointers in internal node)
- 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
- Suppose there are 1,000,000,000 data entries.
  - $H = \log_{133}(1000000000/132) < 4$
  - The cost is 5 pages read

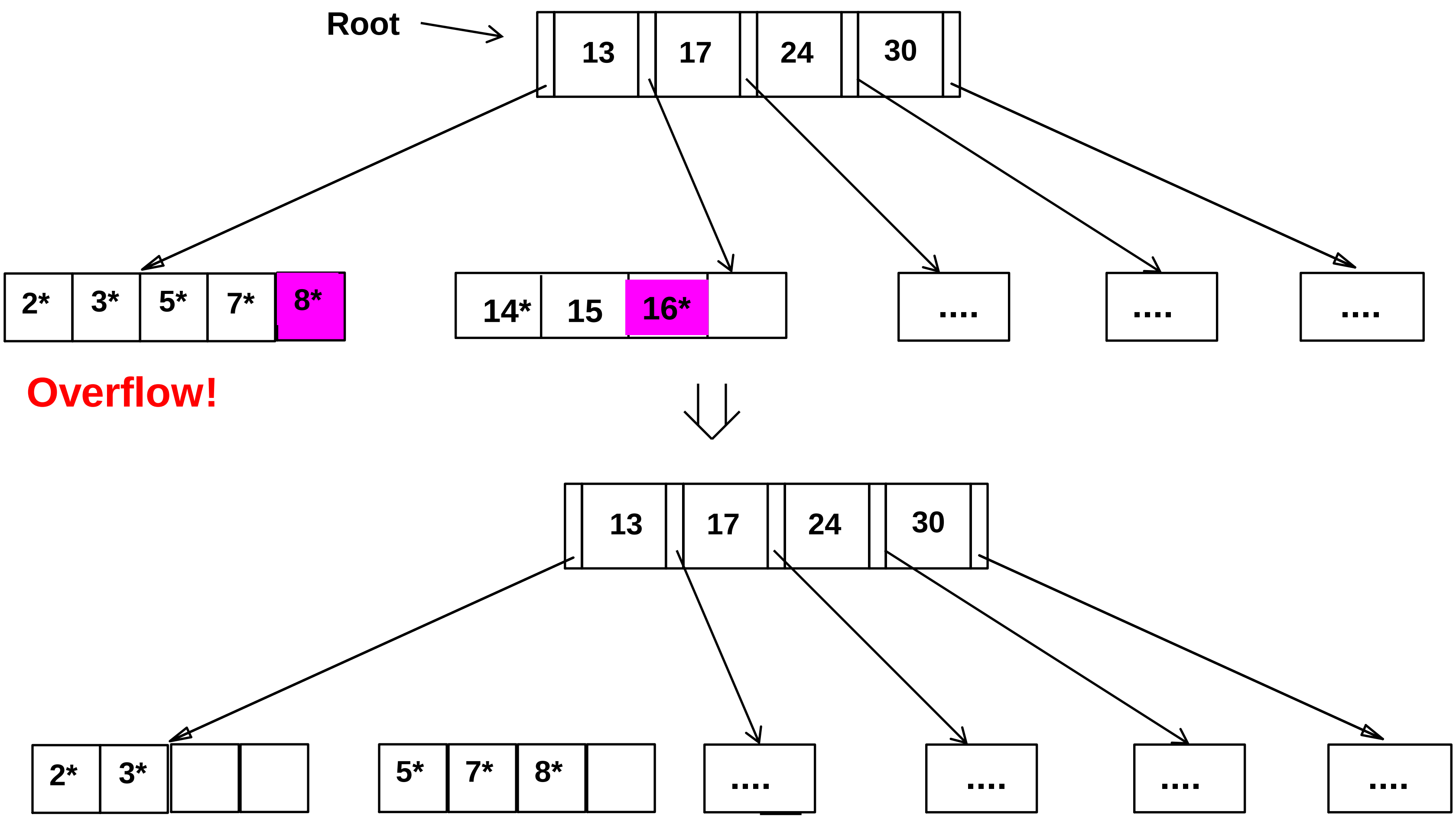
Next

- Inserting Data into B+ Tree
- Deleting from B+ Tree

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, put middle key in L2
    - **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split an internal 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.*

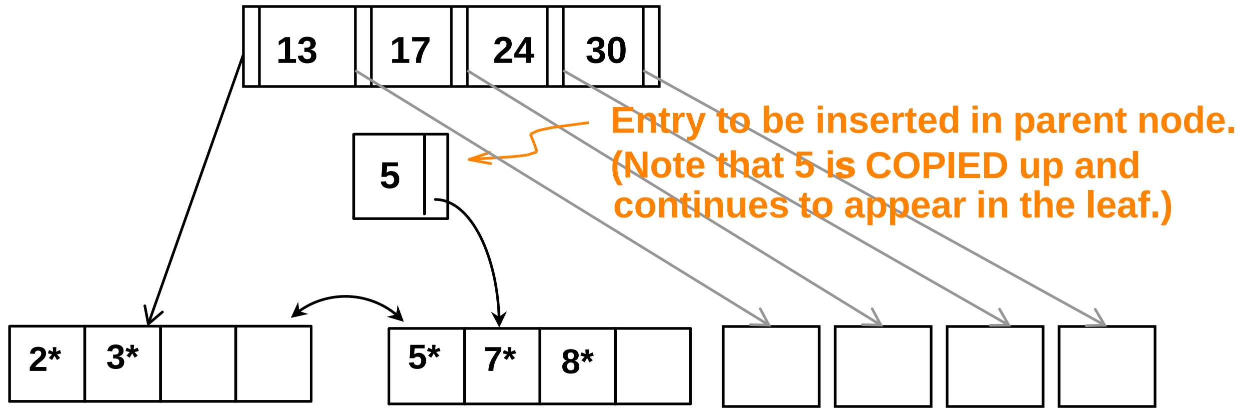
Inserting 16*, 8* into Example B+ tree



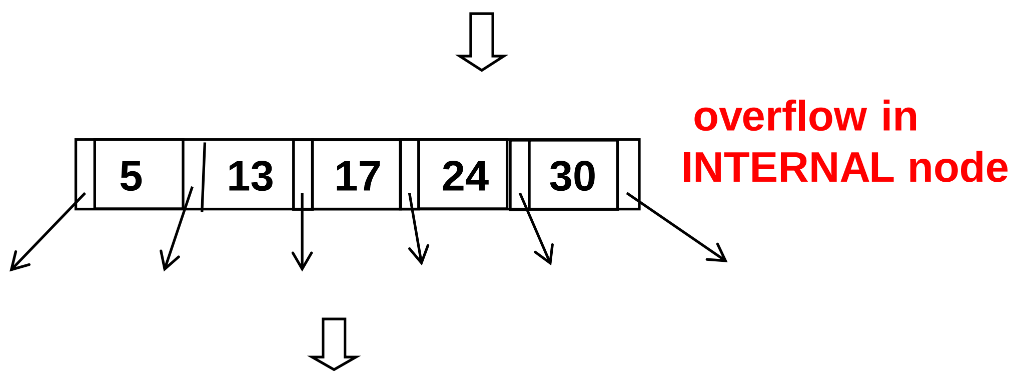
One more child generated, must add one more pointer to its parent, thus one more key value as well.

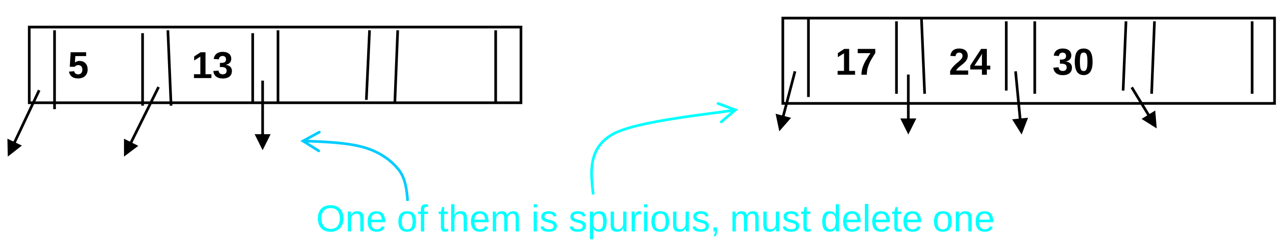
Inserting 8* into Example B+ Tree (order 2)

CASE: For splitting leaf node, we copy the middle value up.



CASE: For splitting internal node, do we also copy the middle value up?

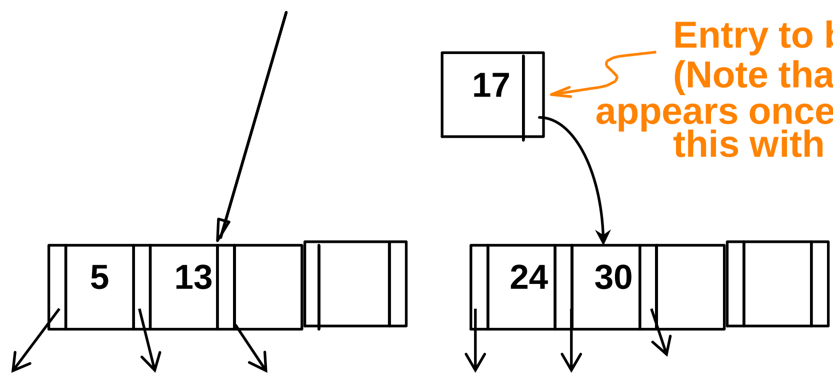




Insertion into B+ tree (cont.)

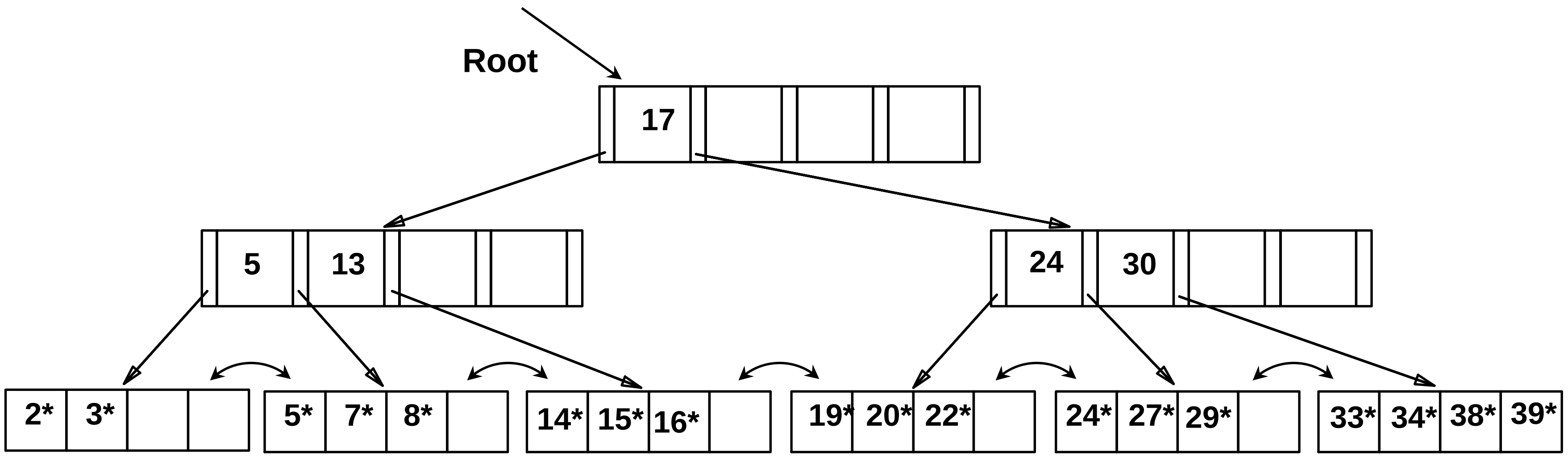
- We delete this pointer!
- But then we should also delete 17
- On the other hand, a value must be inserted into its parent.
- Therefore, we insert 17 to its parent

- **This explains why we must push up the middle entry, instead of copying it up, when we split an internal node.**



Entry to be inserted in parent node. (Note that 17 is pushed up and only appears once in the index. Contrast this with a leaf split.)

Example B+ Tree After Inserting 8*



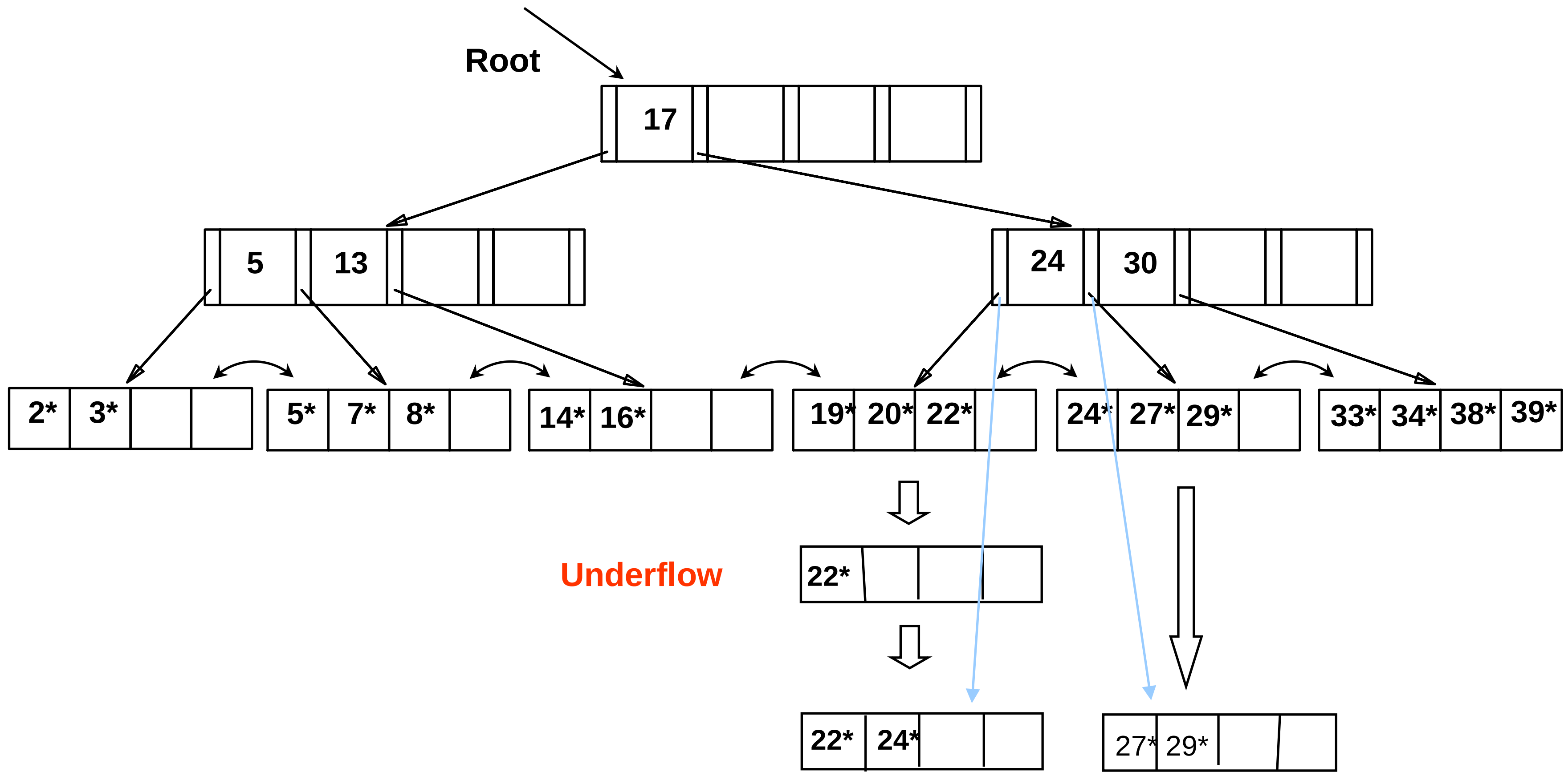
- Notice that root was split, leading to increase in height.

- In this example, we can avoid splitting by re-distributing entries; however, this is usually not done in practice.

Deleting a <u>Data Entry</u> from a B+ Tree

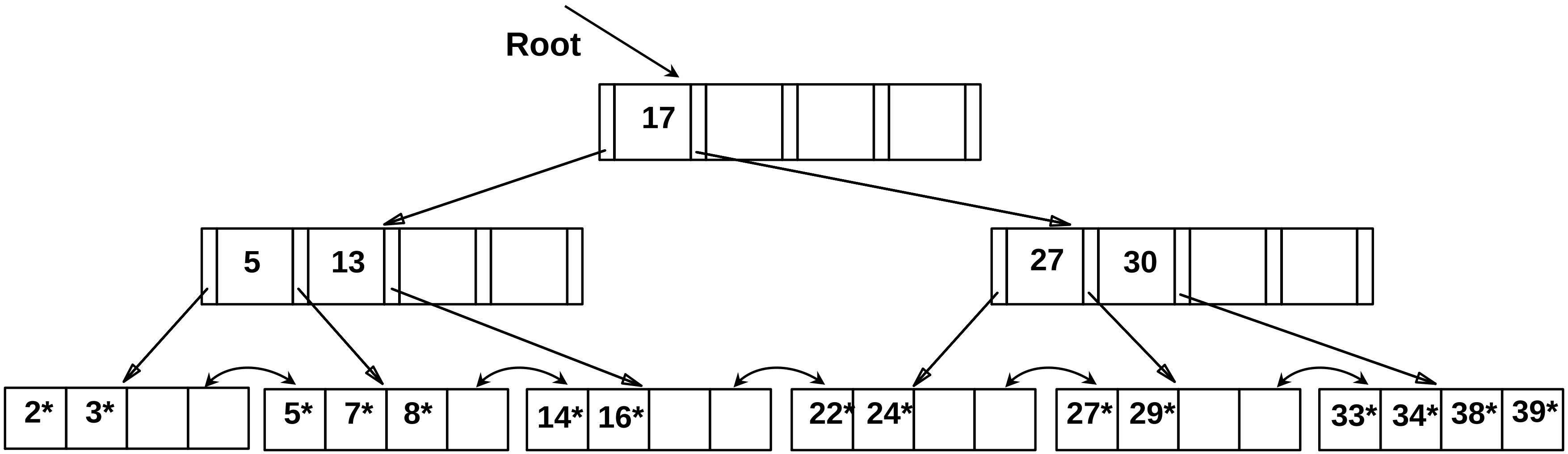
1. From **root**, go to **leaf node** L wt. target entry
2. Remove the target entry from the node
3. Count the number of entries left in L
   - [case 1] L is at least half-full
     1) done! end of operation
   - [case 2] L now has only **d-1** entries
     1) Try to re-distribute by borrowing from <u>sibling</u> (any adjacent node with same parent as L)
     2) If re-distribution fails, <u>merge</u> L and sibling
        I. Delete entry (pointing to L or sibling) from parent of L
        II. Merge could propagate to root, decreasing height

Delete 19* and 20*

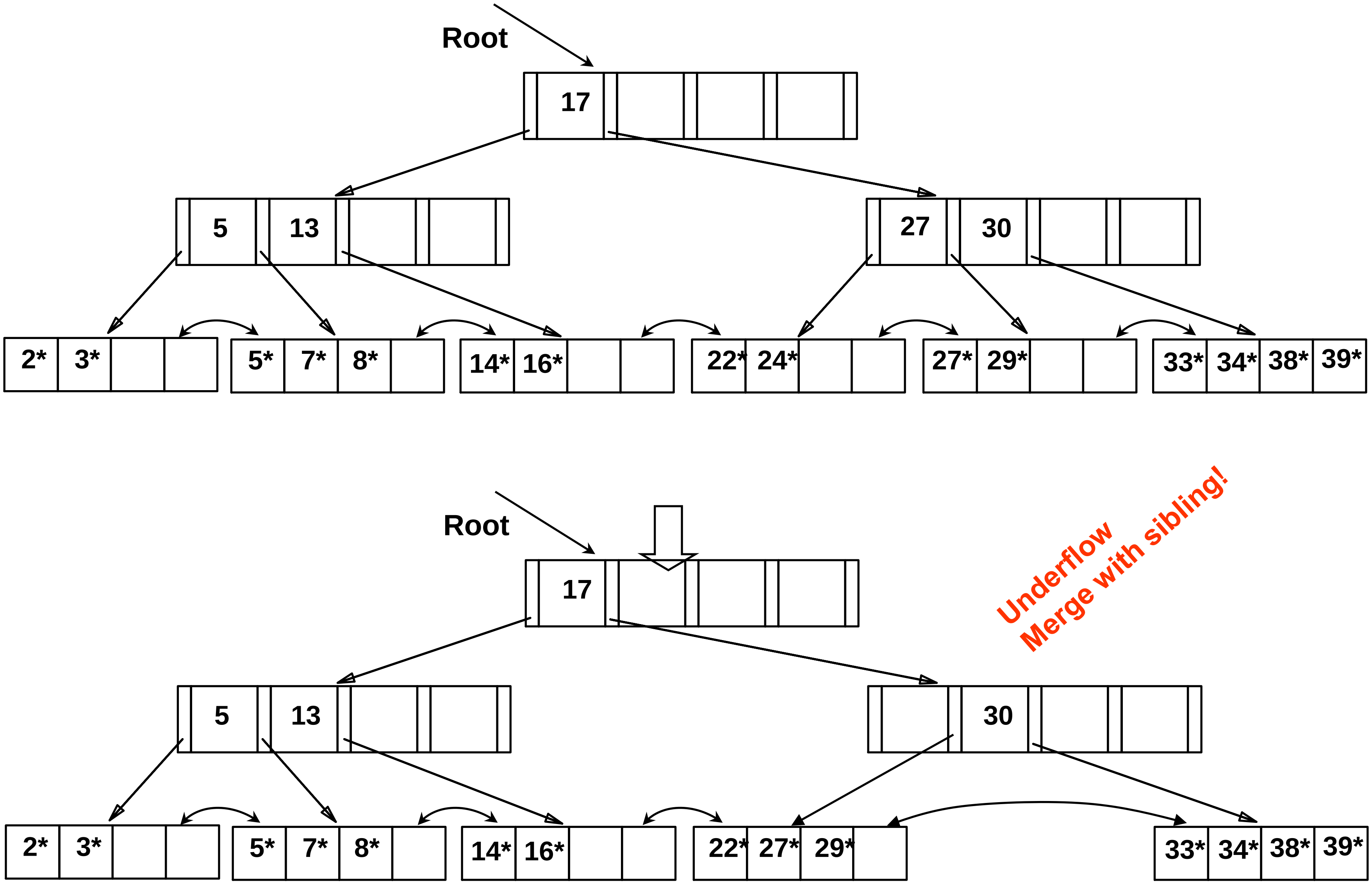


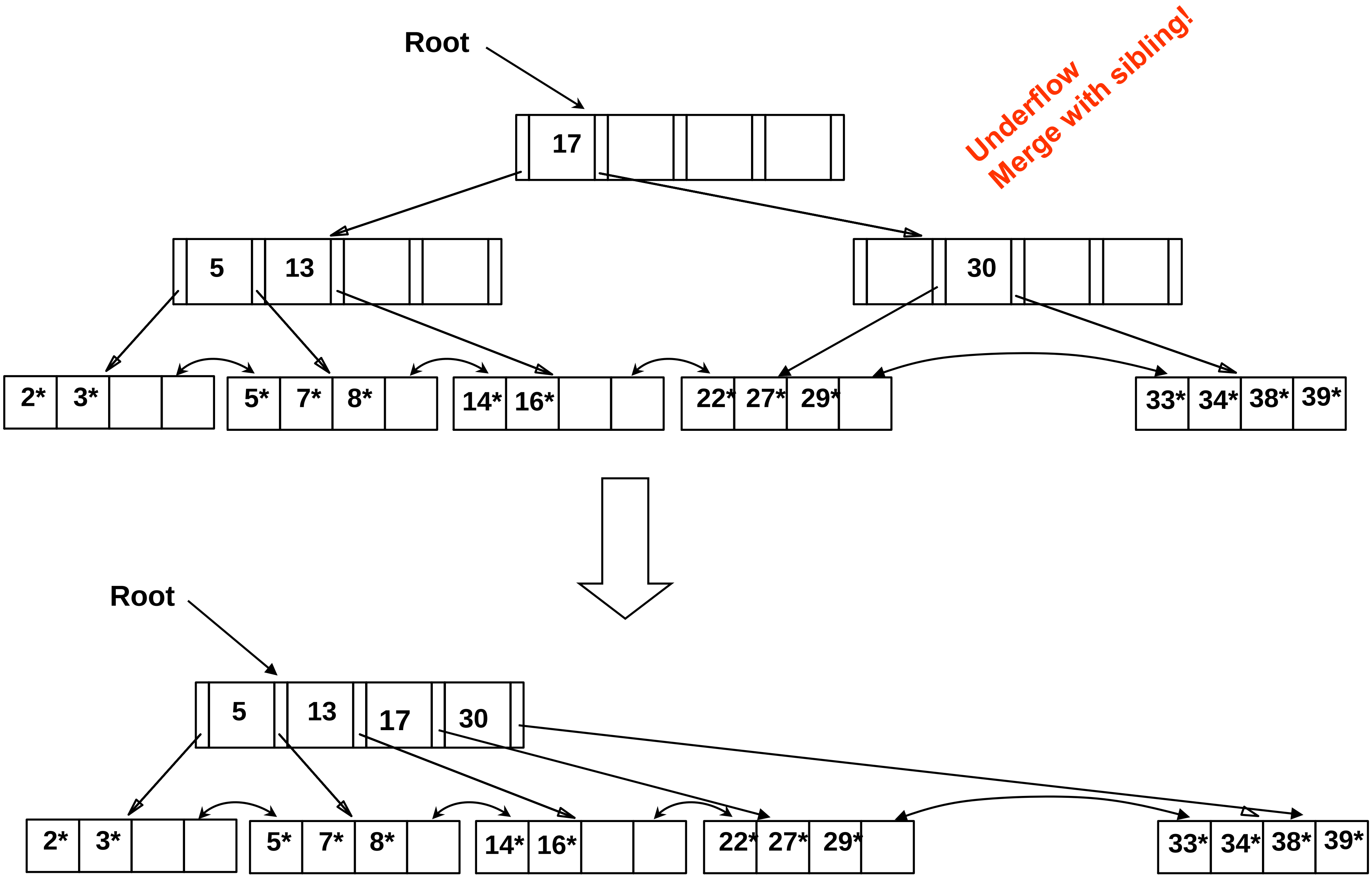
Psst… We forgot something

Deleting 19* and 20* (cont.)



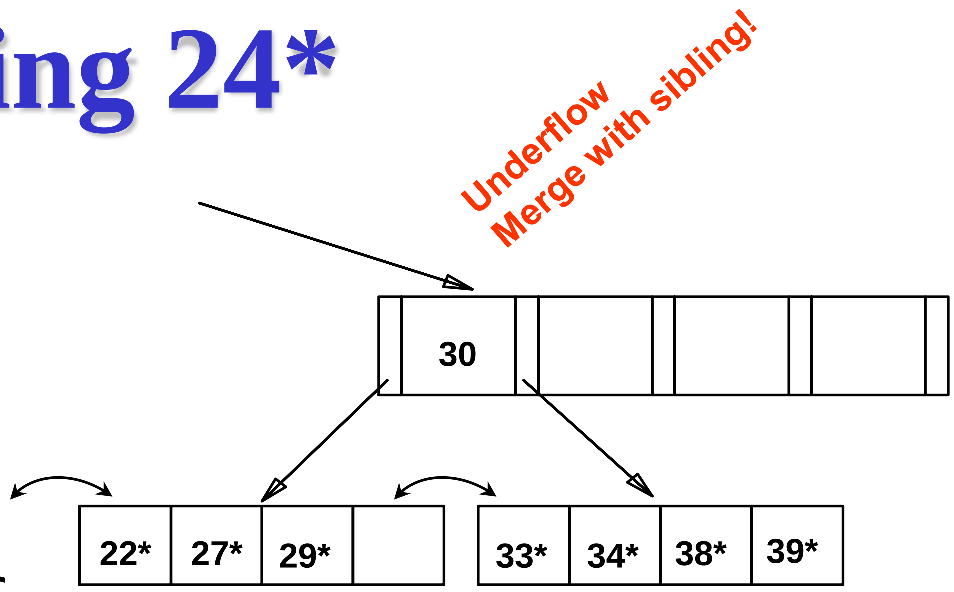
- Due to the redistribution, we must **update** some internal nodes
  – We can do this by copying 27 up to it's parent node.
- Suppose we now want to delete 24..
  – Underflow again! But can we redistribute this time?

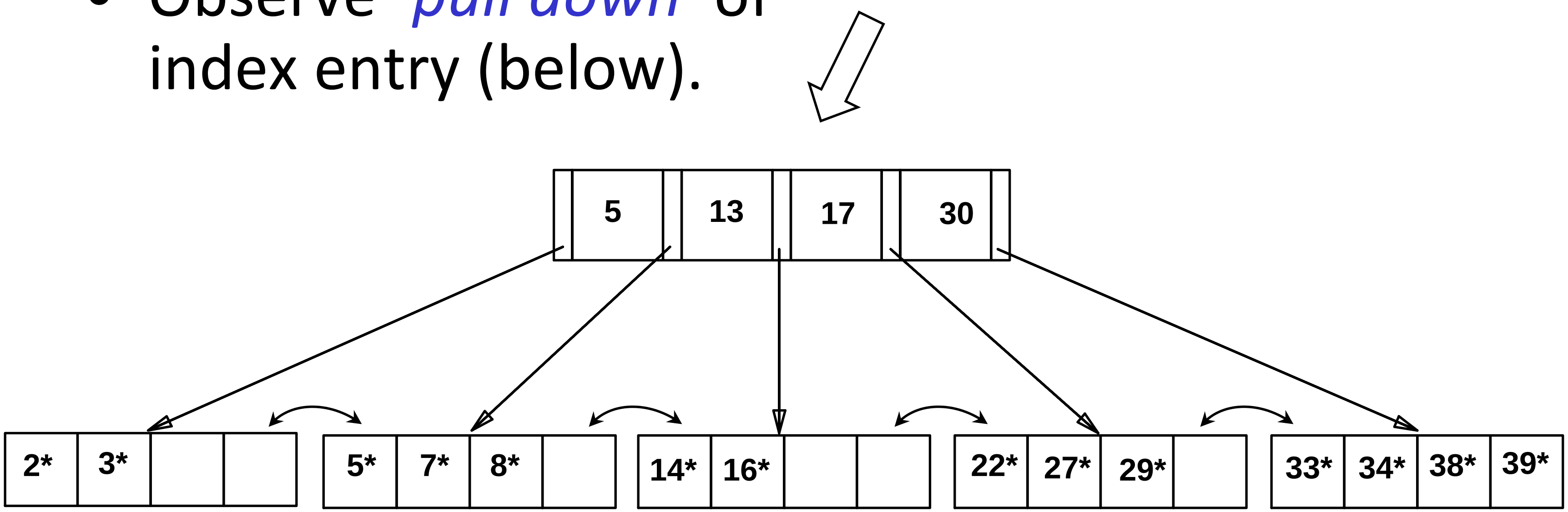

Root
17
5
13
27
30
2*
3*
5*
7*
8*
14*
16*
22*
24*
27*
29*
33*
34*
38*
39*
Root
17
5
13
30
2*
3*
5*
7*
8*
14*
16*
22*
27*
29*
33*
34*
38*
39*
Underflow
Merge with sibling!


Root
17
Underflow
Merge with sibling!
5
13
30
2*
3*
5*
7*
8*
14*
16*
22*
27*
29*
33*
34*
38*
39*
Root
5
13
17
30
2*
3*
5*
7*
8*
14*
16*
22*
27*
29*
33*
34*
38*
39*

Deleting 24*

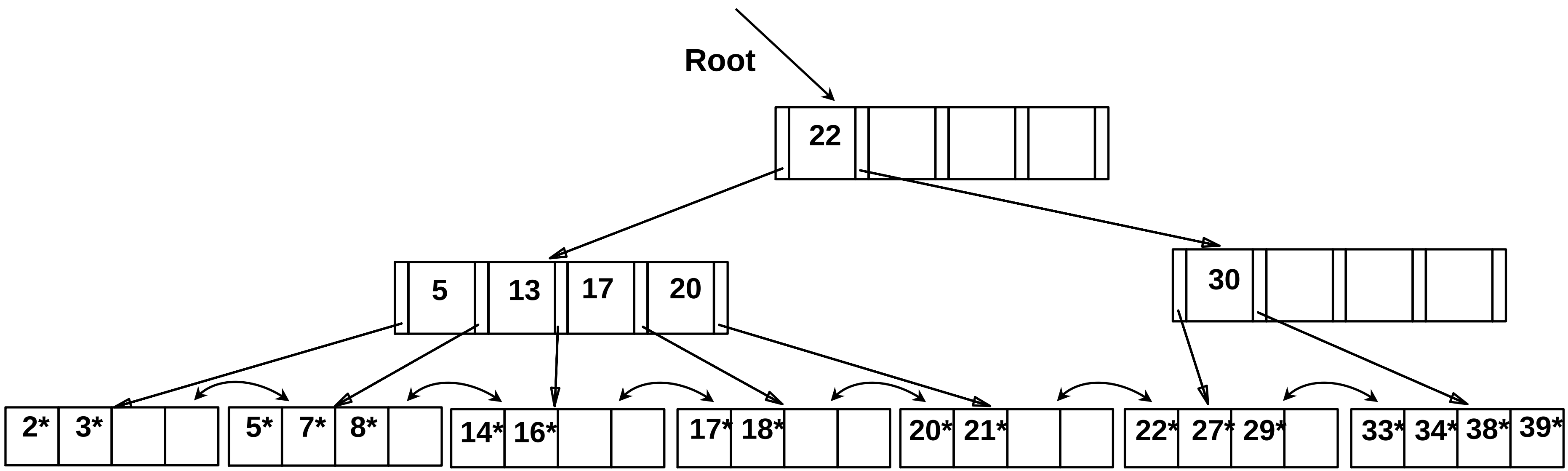
- Two leaf nodes are merged, and 27 is discarded from their parent.
- Observe *`pull down'* of index entry (below).





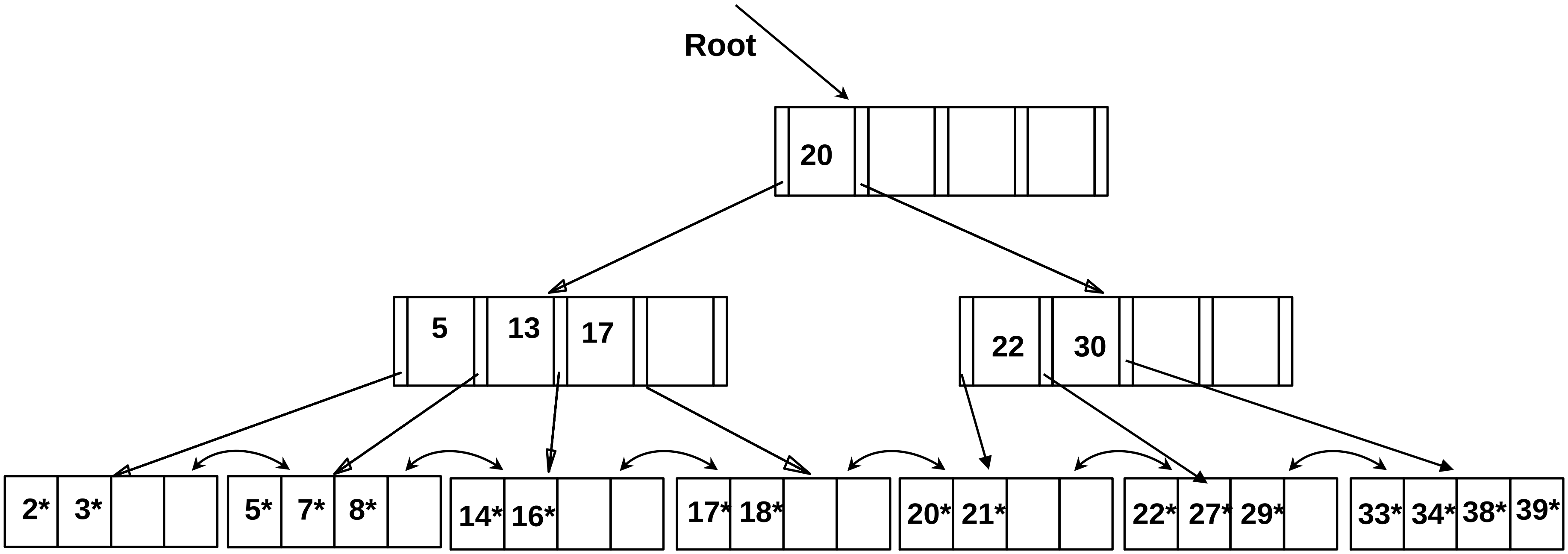
Example of Non-leaf Re-distribution

- Tree is shown below *during deletion* of 24*. (What could be a possible initial tree?)
- In contrast to previous example, we may also opt to 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.



Hash-based Indexes

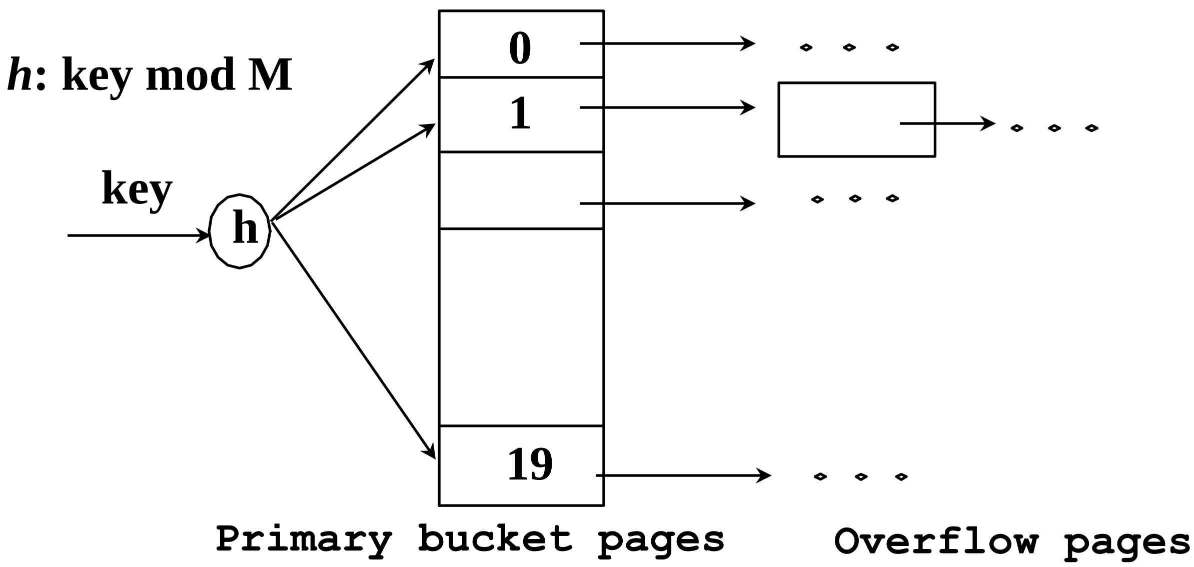
Introduction

- *As for any index, 3 alternatives for data entries* **k***
  1. Data entries are kept in ***buckets*** (an abstract term)
  2. Each bucket is a collection of one primary page and zero or more overflow pages
  3. Given a search key value, k, we can find the bucket where the data entry k* is stored as follows:
     - Use a function, called *hash function,* denoted as *h*
     - The value of *h*(k) is the address for the desired bucket (i.e, the address of a bucket is represented by the address of its primary page)
     - *h*(k) should distribute the search key values uniformly over the collection of buckets

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

Static Hashing

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- A simple hash function can be: **h**(k) = f(k) mod M where M = # of **buckets** and f(k) = a × k + b
- Example: f(k) = k. Let M = 20. Thus h(k) = k mod 20
  - Assume each page contains two entries
  - For k = 1, 21, 41, one of them must go to overflow page



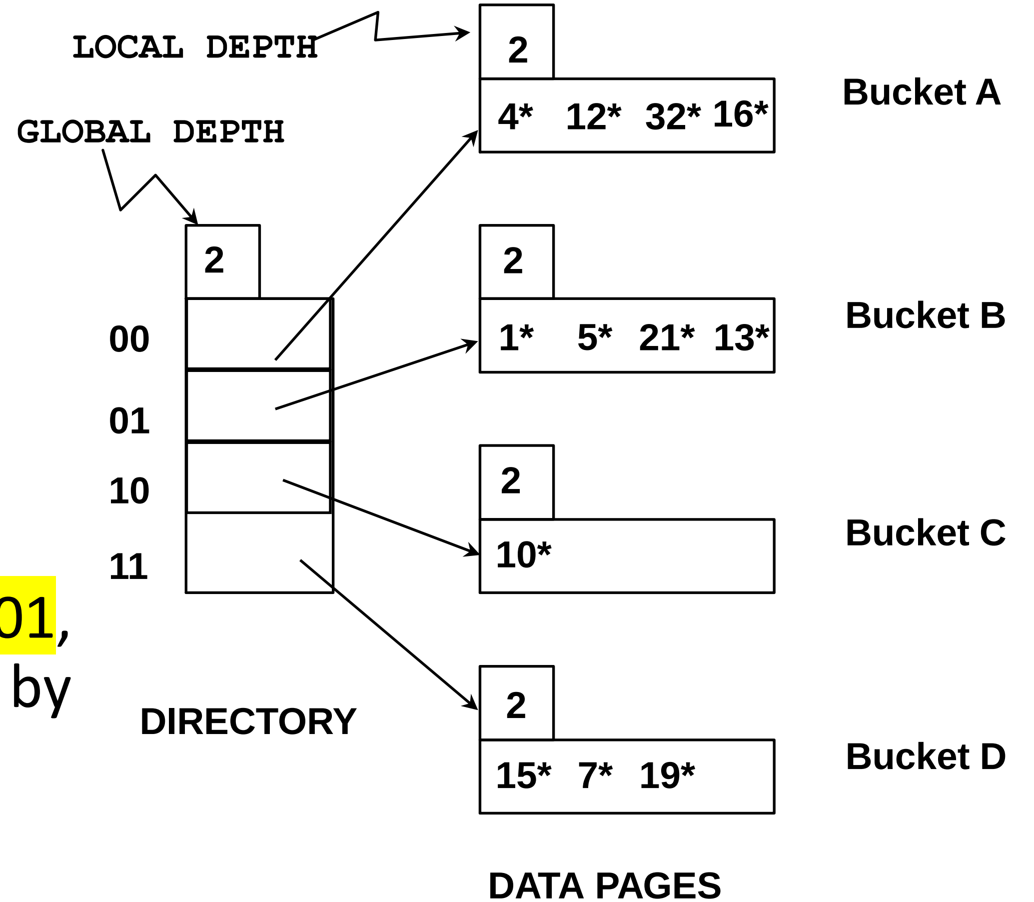
Static Hashing (Exp.)

- Buckets contain *data entries*.
- Hash function must distribute values over range of 0 ... M-1.
- It **should** be random and uniform.
  - If search key values greatly outnumber M, then many different key values may be hashed to the same bucket
  - Consider a possible scenario if M = 20 and there are 1000 different search key values:
    - Suppose at least one bucket contains 50 values.
    - If the size of a page is 2, then that bucket contains 25 pages: 1 primary and 24 overflow pages
  - Therefore, long overflow chains can develop and degrade performance.
  - *Extendible hashing*: Dynamic techniques to fix this problem.

Extendible Hashing

- Situation: Bucket (primary page) becomes full. Why not re-organize file by *doubling* # of buckets?
  - Must re-hash all data entries to the right buckets
  - Example: assume hash function h(k) = k mod M
    - For M = 4, entries 3* and 7* both in bucket 3 (3 mod 4 = 7 mod 4 = 3)
    - But for M = 8, entry 7* will be in bucket 7
  - Can we only re-hash those values that have changed addresses?
    - Difficulties: without re-hashing all the values, we don't know which values keep the old addresses and which get new addresses
  - Reading and writing all pages are expensive!
  - Question: how do we add more buckets, but only re-hash a few data entries?
  - Answer: use a level of indirection, ***directory of pointers to buckets***

Example

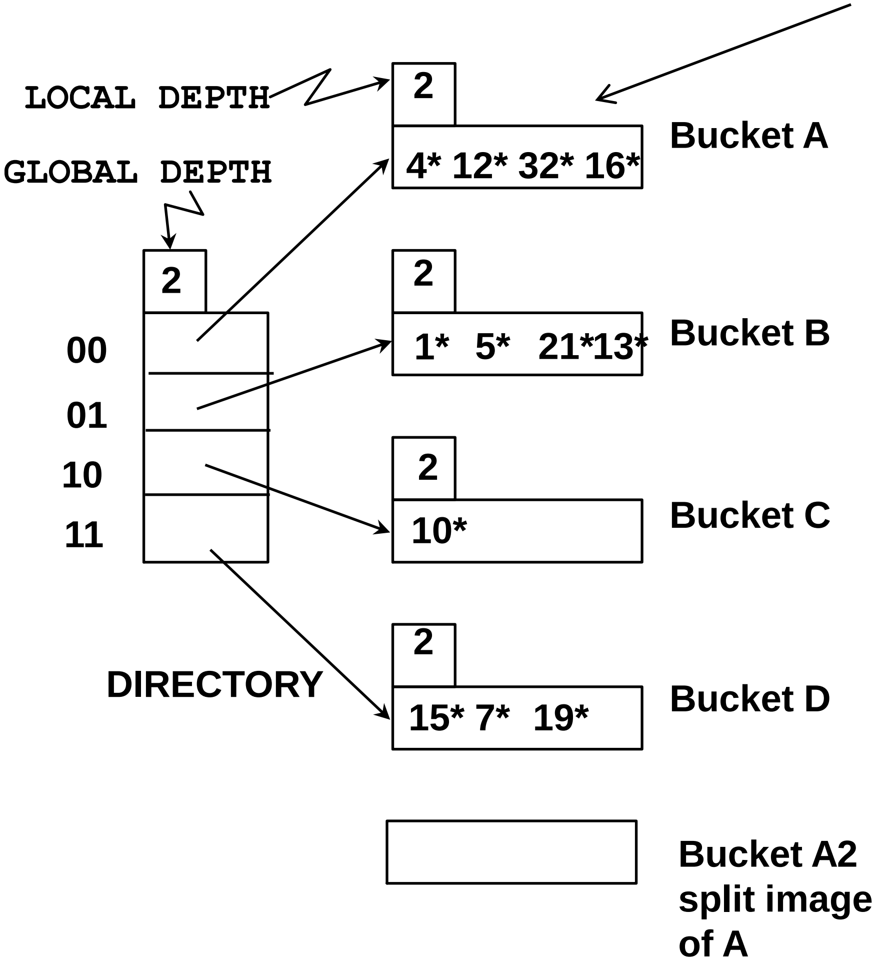


- **h**(*r*) = *r mod 32*
- Directory is array of size 4.
- To find bucket for *r*, take last \`*global depth*' # bits of **h**(*r*)
  - If *r* = 5, **h**(*r*) = 5 = binary 101, 5* is in bucket pointed to by 01.

❖ **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.)**

Example (cont.): Insert 20*



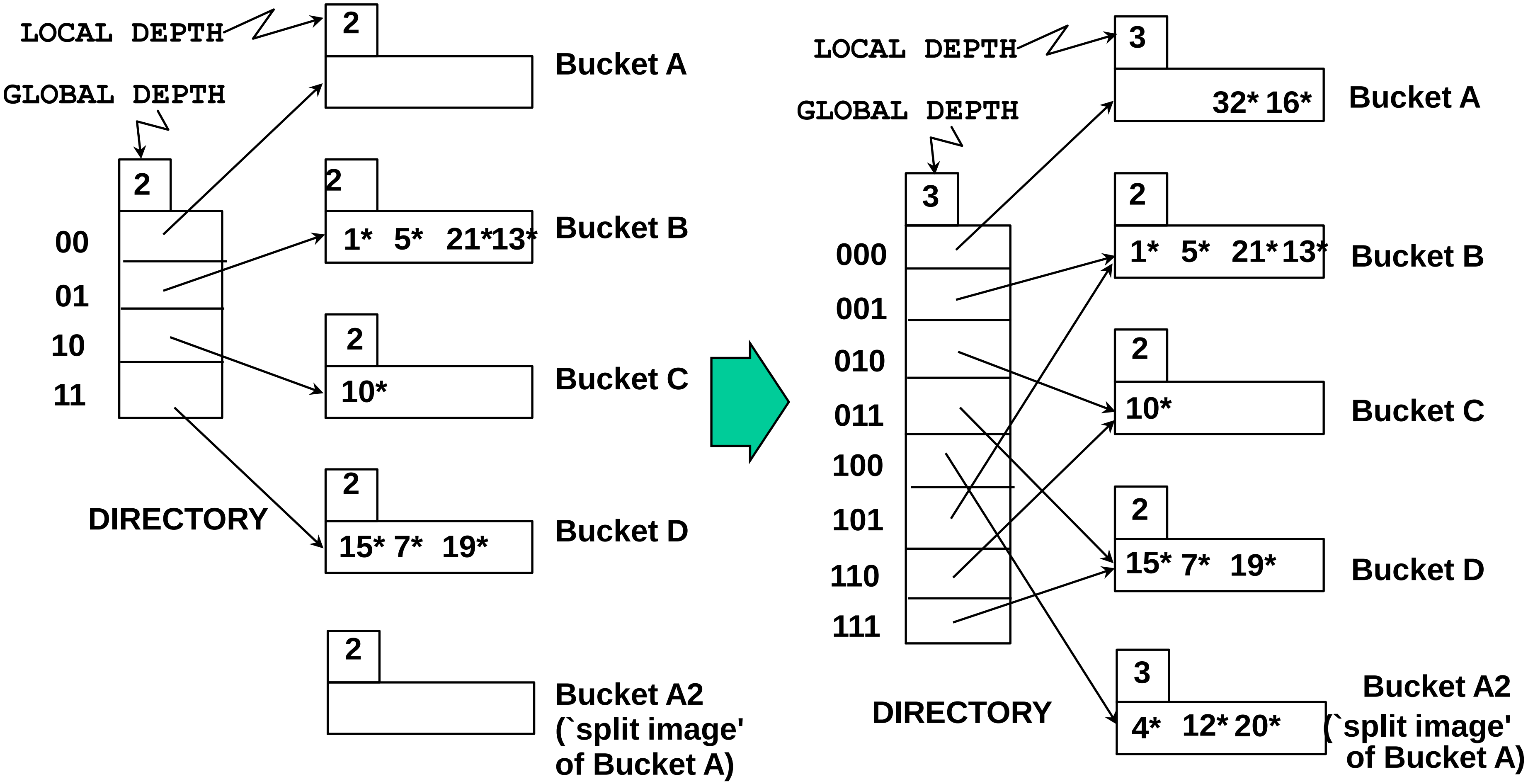
Insert 20*, causing overflow, we do the following:

- Split bucket A to A &A2
- for the five data entries, 4*, 12*,32*,16*,20*, if for any r* its 3rd bit in h(r) is 1, then move it to A2:

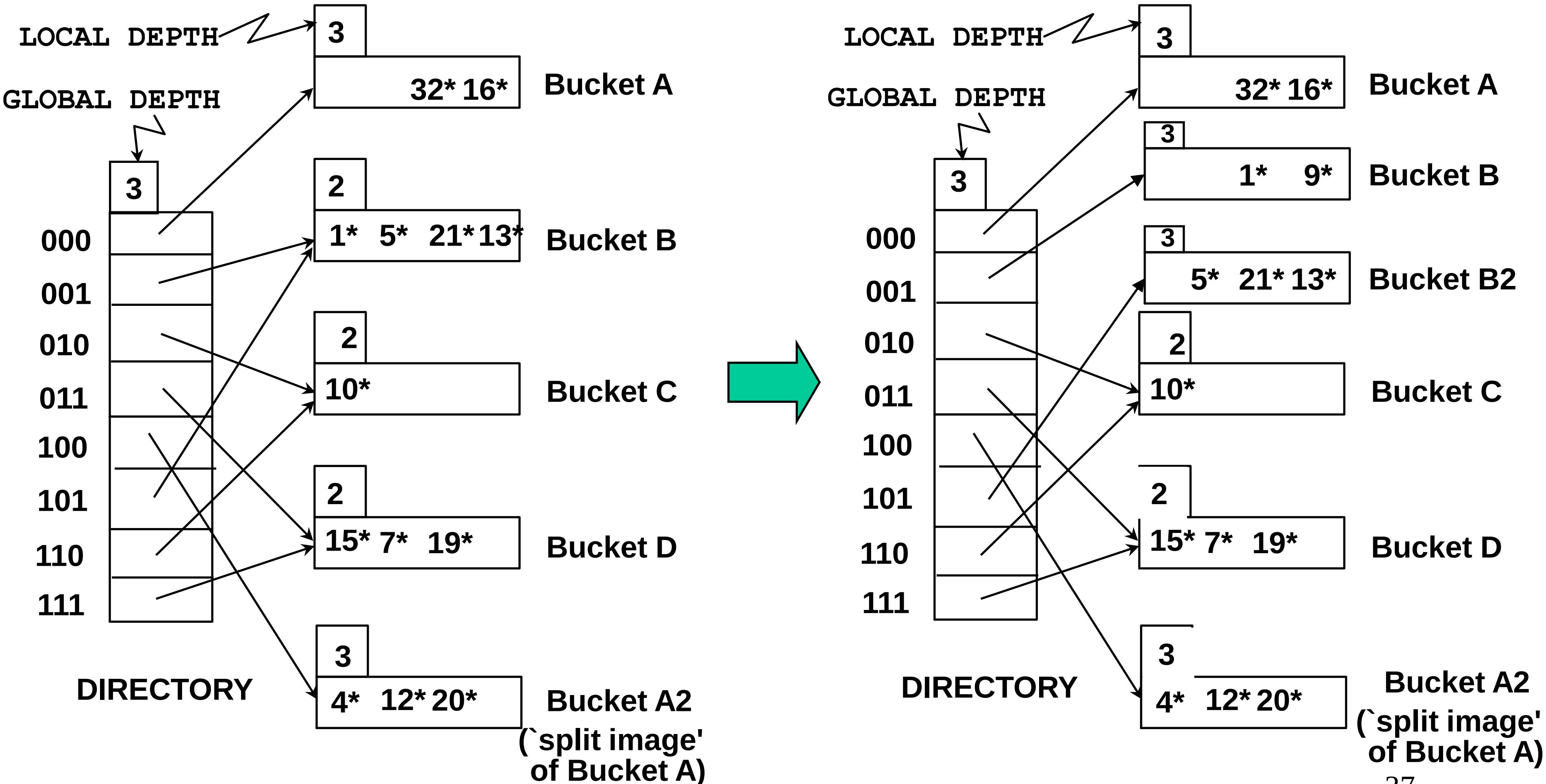
  h(4) = 000100
  h(12) = 001100
  h(32) = 100000
  h(16) = 010000
  h(20) = 010100

Insert h(r)=20 (Causes Doubling)

h(16)=010000
h(32)=000000
h(20)=010100
h(12)=001100
h(4) =000100



Insert 9 (Does Not Cause Doubling): h(9) = 01001

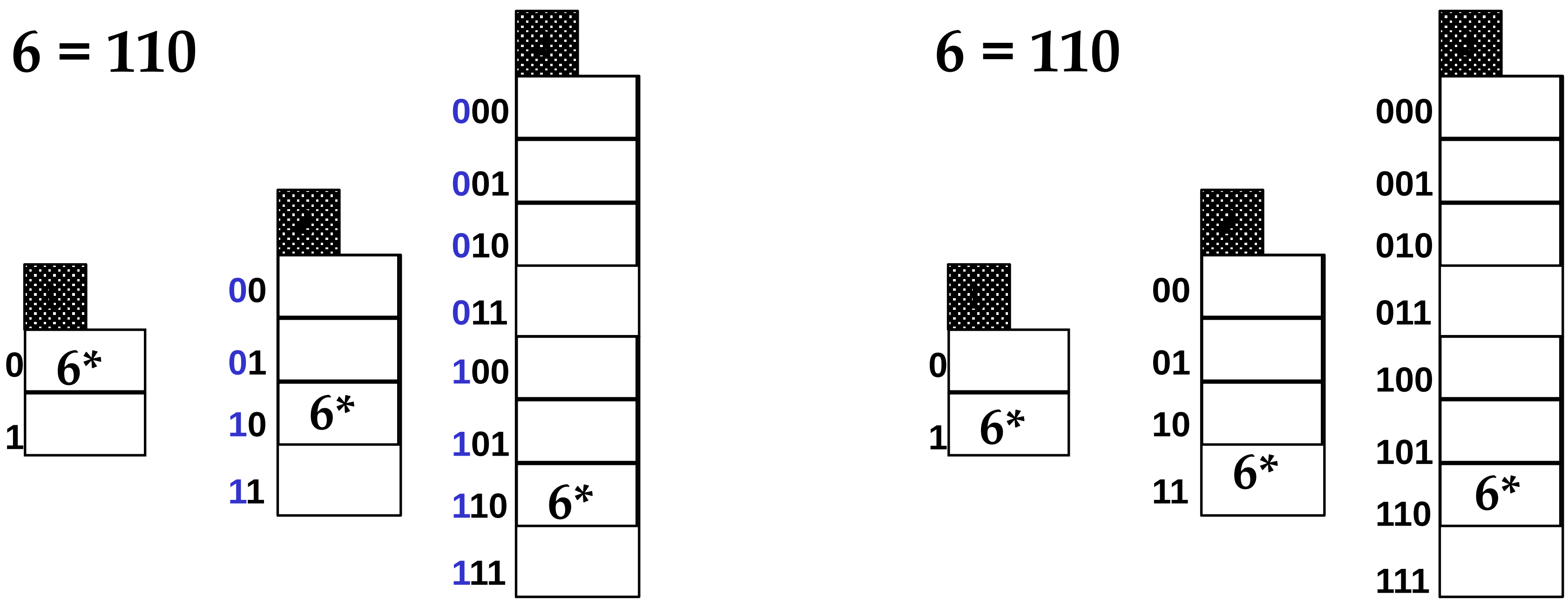


Points to Note

- h(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 the directory need doubling. How?
- When does bucket split cause directory doubling?
  - Before insertion, *local depth* of bucket <= *global depth*.
  - After insertion, if overflow, generate split image, and increment the *local depth*
  - If this causes *local depth* > *global depth*, then directory is doubled, and at the same time increment *global depth*
  - Doubling directory is done 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.
  - Directory grows in spurts, and, if the distribution *of hash values* is skewed (e.g., a large number of search key values all are hashed to the same bucket), 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 the same bucket as its split image, can halve directory.

Summary

- 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.