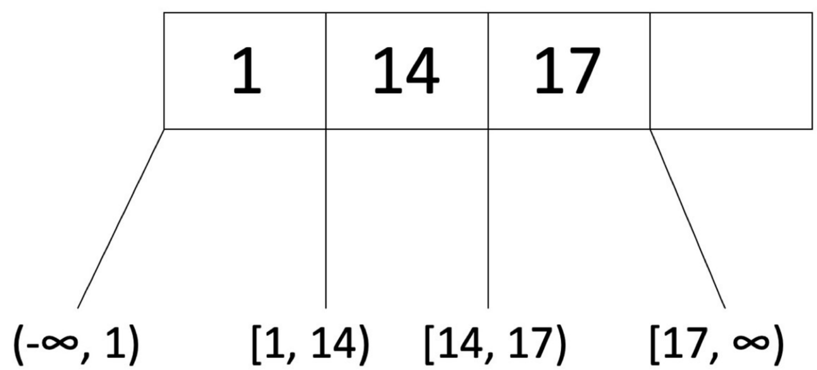
# B+ Tree: Most Widely Used Index



- 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 **d** <= $\underline{m}$ <= 2**d** entries. The parameter **d** is called the *order* of the tree.
- Node with order d = 2,

e.g., 2 <= m <= 4



# B+ Trees in Practice



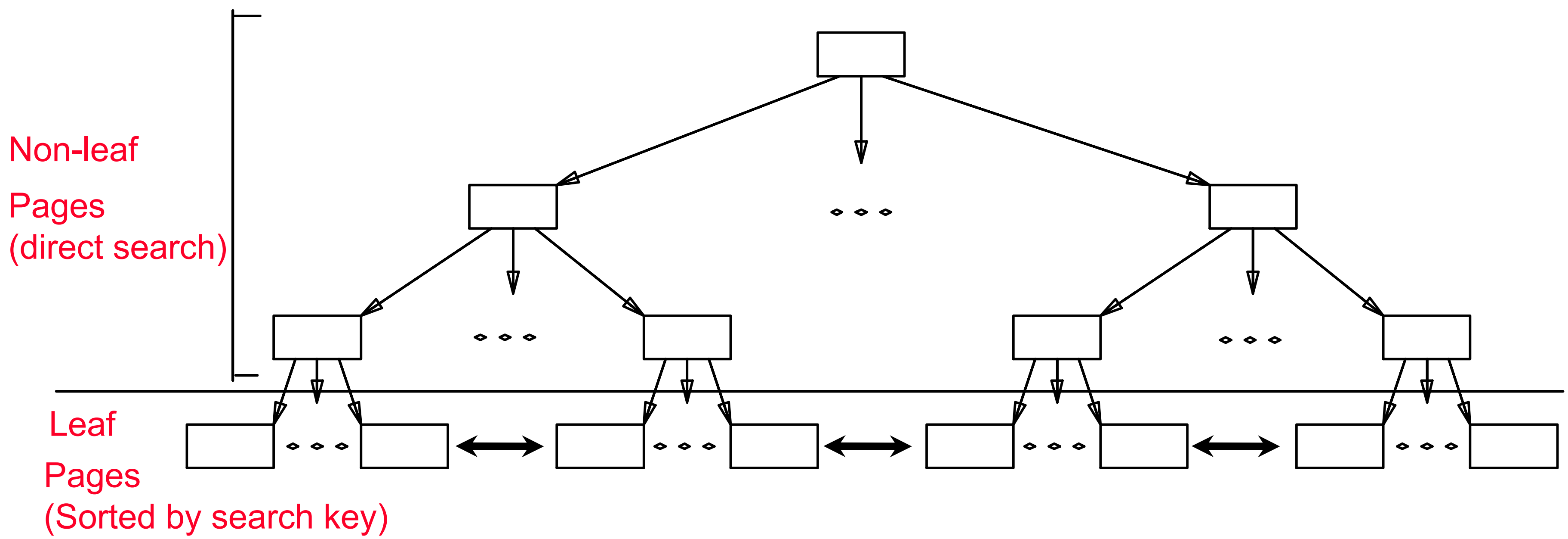
- Typical order: 100.
- Typical fill-factor: ln 2 = 66.5%  (approx)
  - average fanout = 2 x 100 x 66.5% = 133
- Typical capacities:
  - Height 4: $133^4$ = 312,900,721 pages
  - Height 3: $133^3$ = 2,352,637 pages

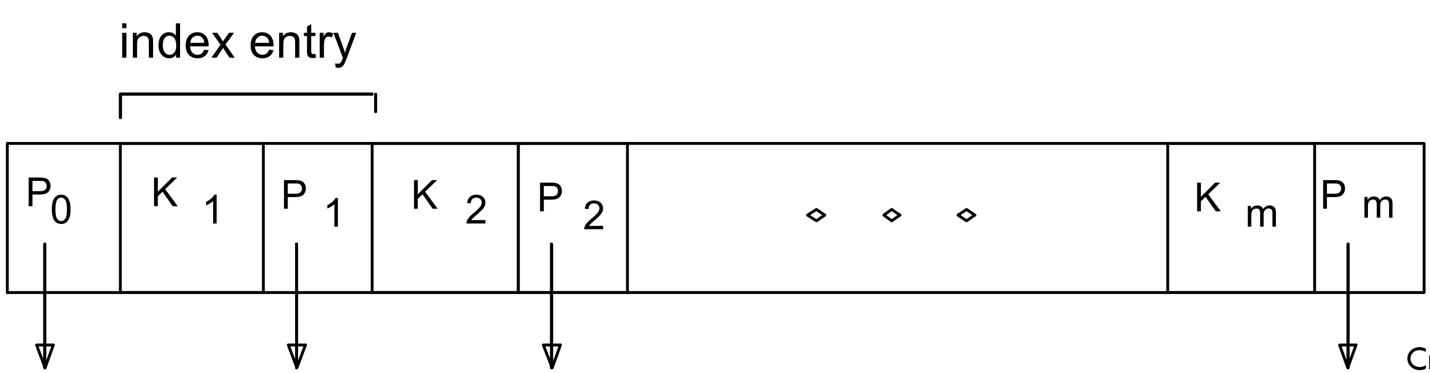
- For typical orders (d ~ 100-200), a shallow B+ tree can accommodate very large files.

# B+ Tree Index



Supports equality and range-searches efficiently

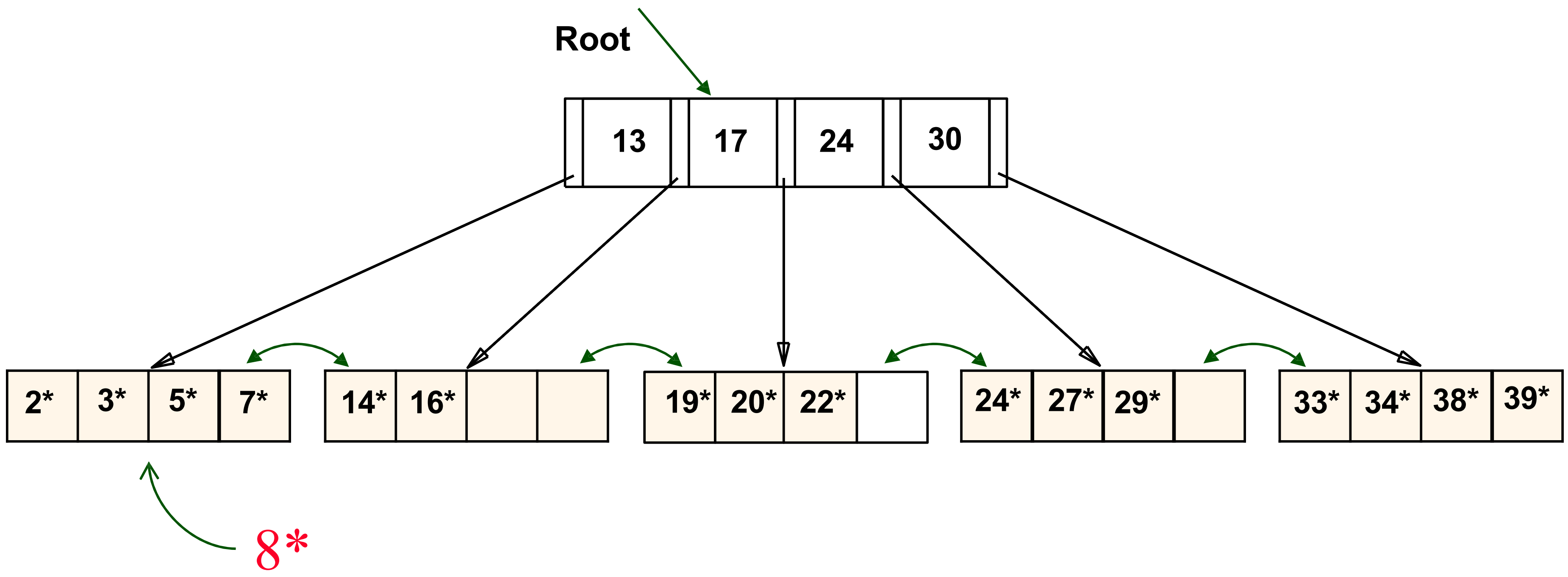



Credit: Renee Miller

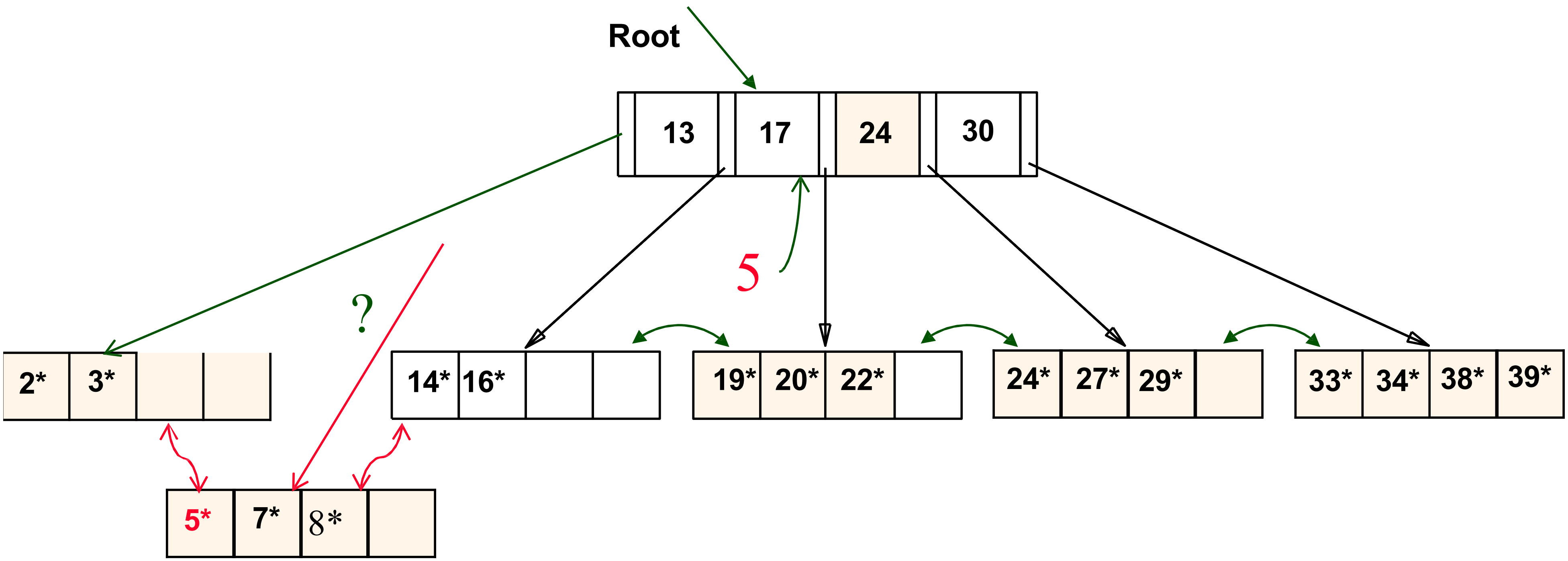
# Insertion Example





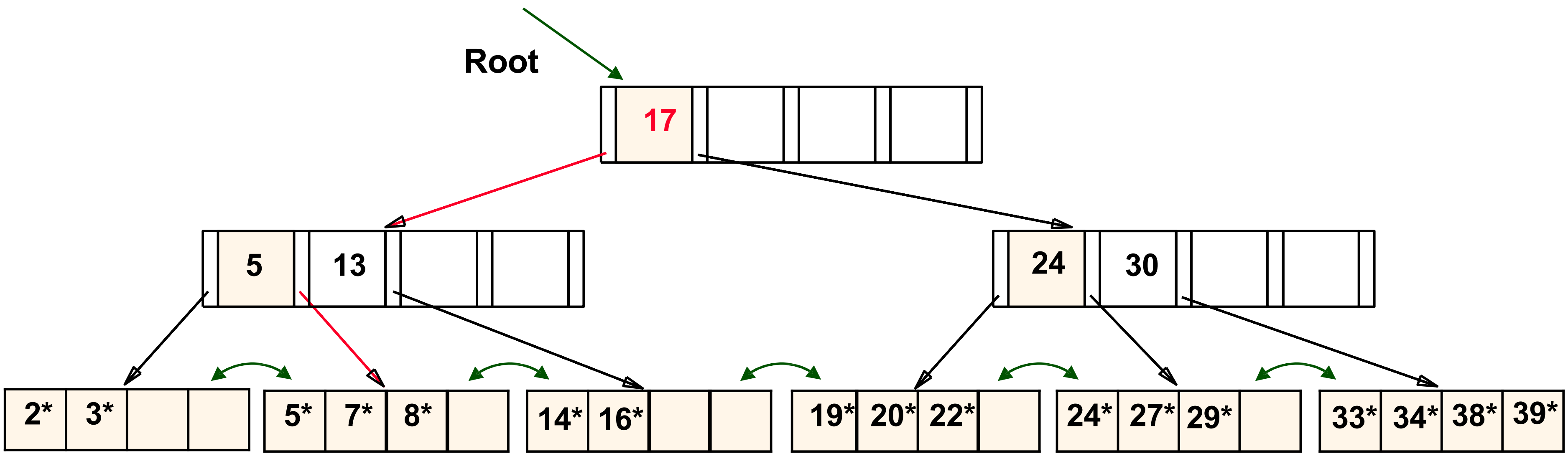
# Insertion Example





# After Inserting 8*



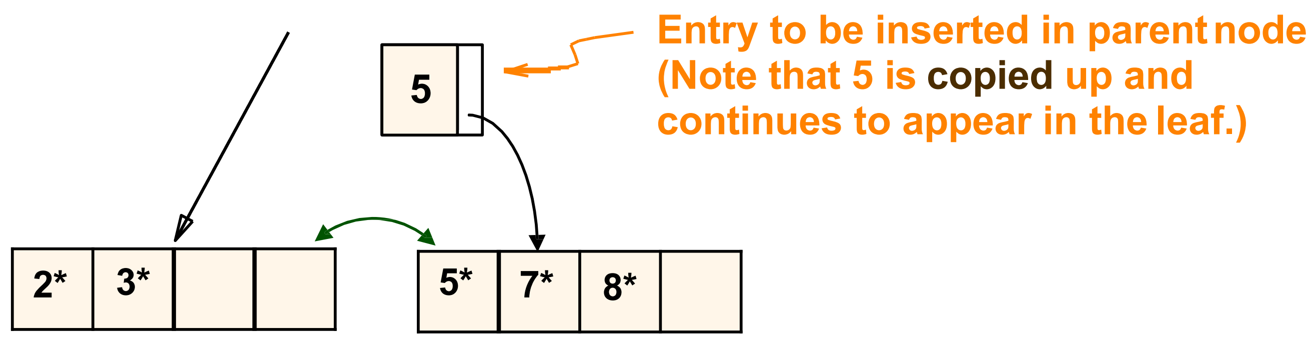


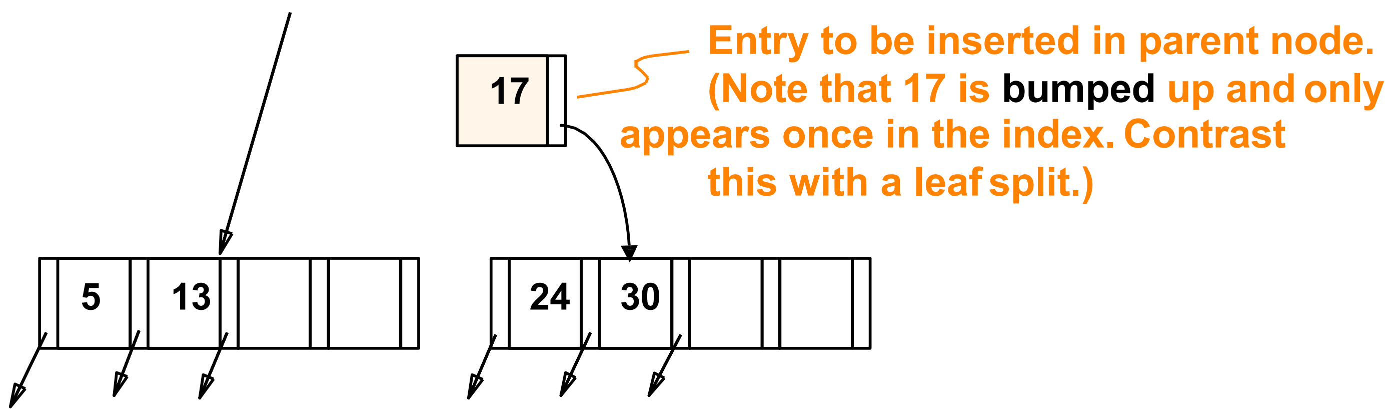

❖ Notice that root was split, leading to increase in height.

# Copy-Up vs. Bump-Up



- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Note difference between *copy-up* and *bump-up;* Why do we handle leaf page split and index page split differently?

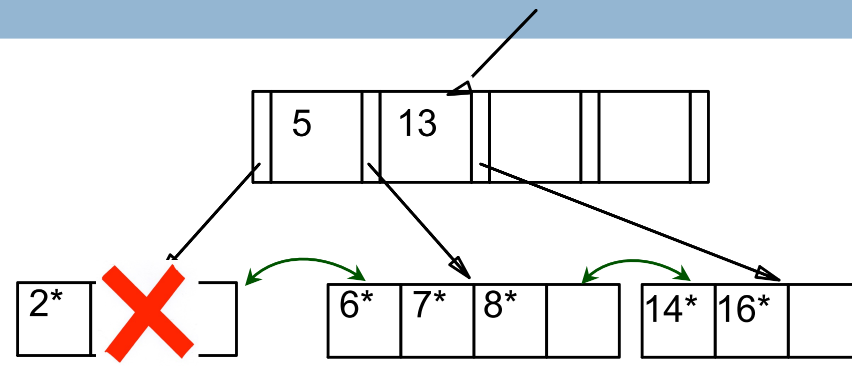


# Deleting a Data Entry

Delete value 3

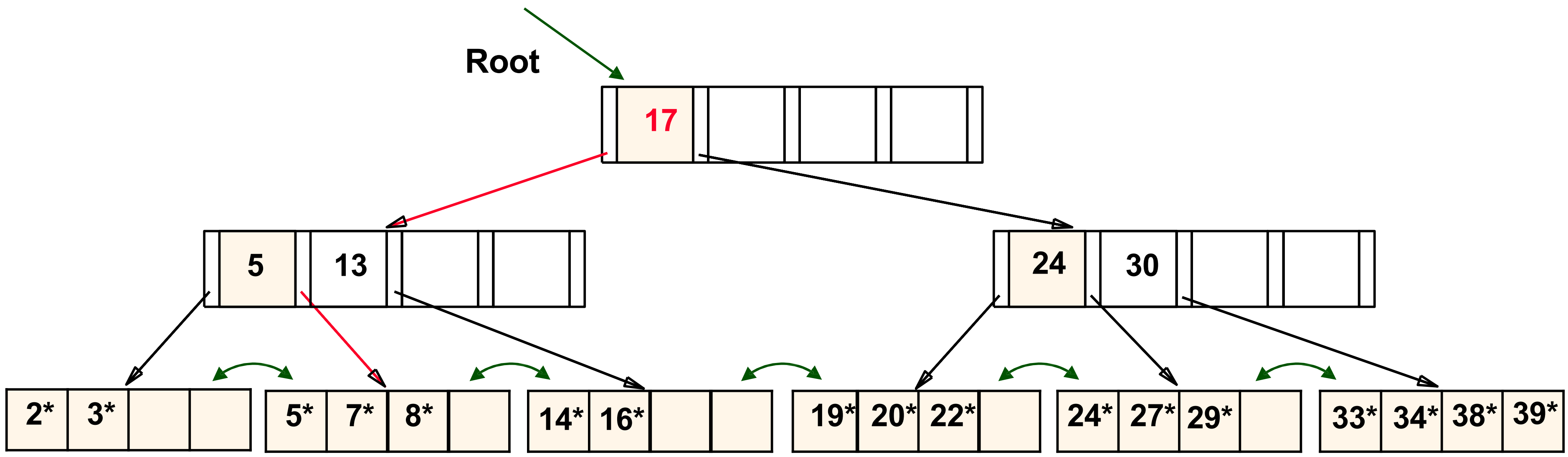





- Start at root, find leaf L where entry belongs.
- Remove the entry.
  - If L is at least half-full, done!
  - If not,
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
    - If re-distribution fails, merge L and sibling.
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- Merge could propagate to root, decreasing height.

# Deleting 19* is Straightforward

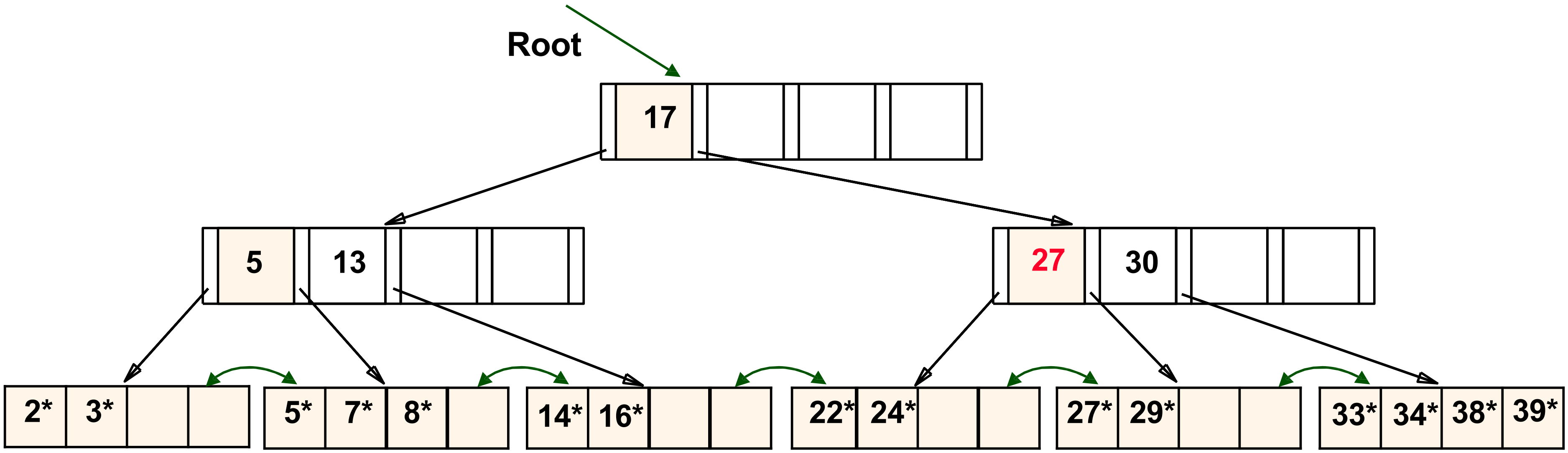





❖ What happens if we delete 20* next?

# Example Tree: Deleting 19* and 20*



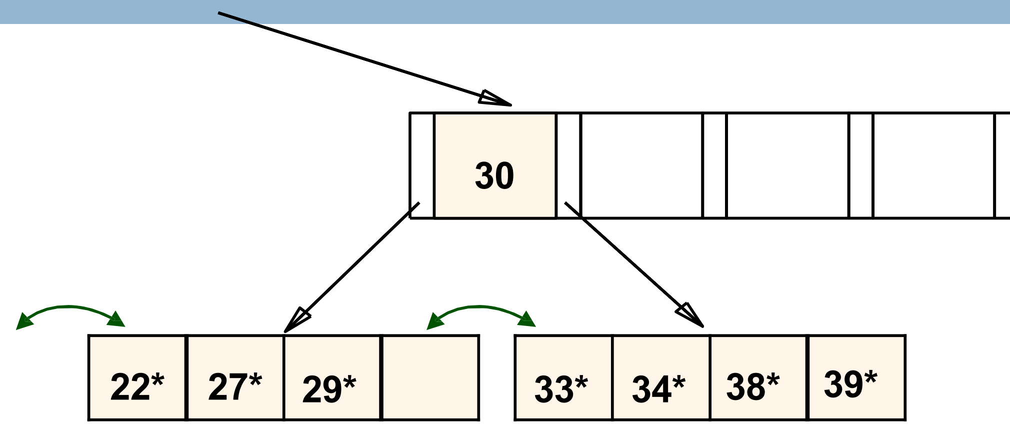


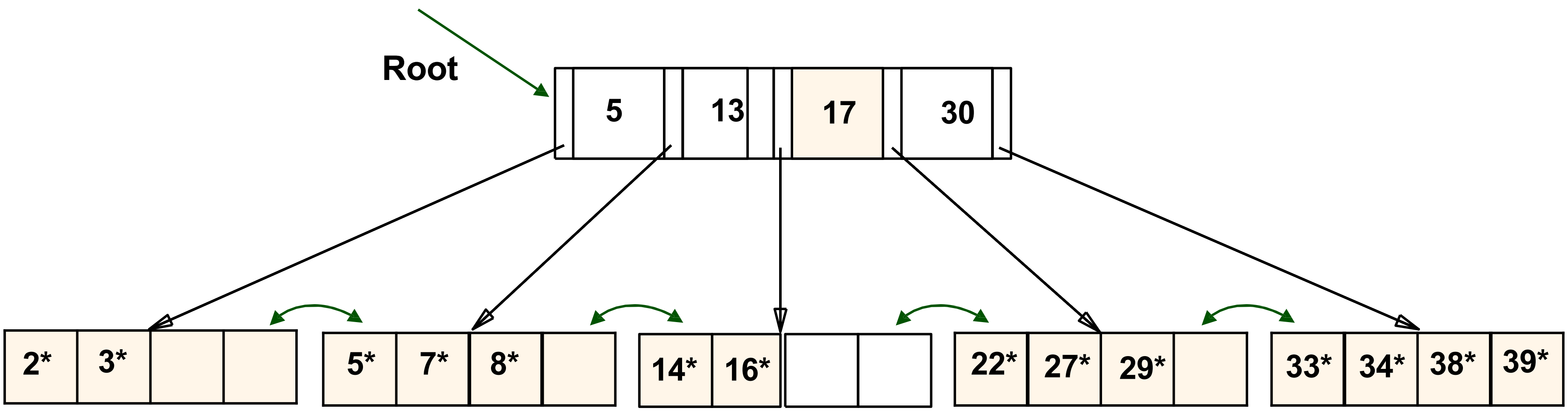

- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how *new* middle key is *copied up*.
- What happens if we delete 24* now?

# Deleting 24* ...



- Must merge.
- Observe *`toss'* of index entry (on right), and *`pull down'* of index entry (below).



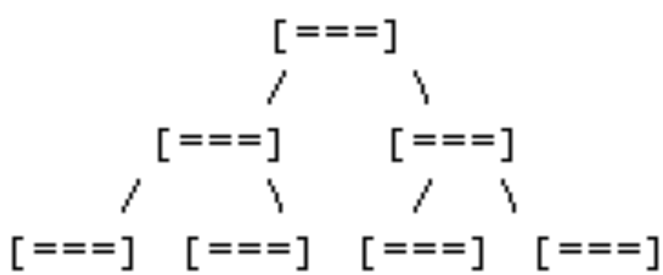



# Balanced vs. Unbalanced Trees

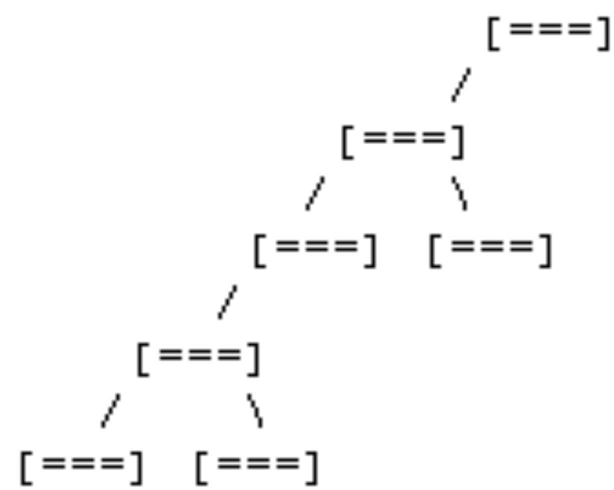


- In a balanced tree, every path from the root to a leaf node is the same length.

- Balanced



- Unbalanced



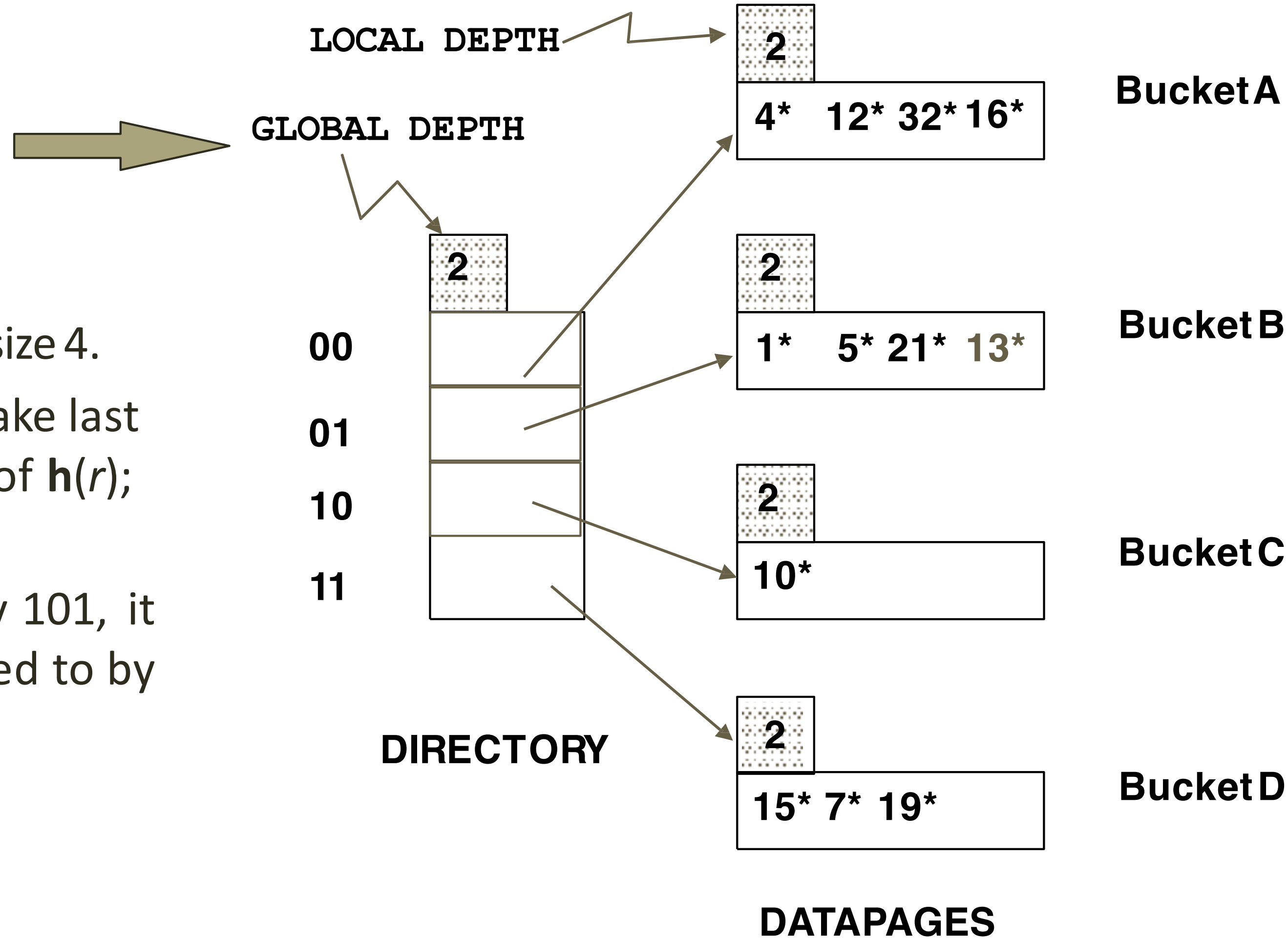
Credit: S. Lee

# Hash Based Indexes



- Good for equality searches
- 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.

# 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.)

# Three basic alternatives for data entries in any index



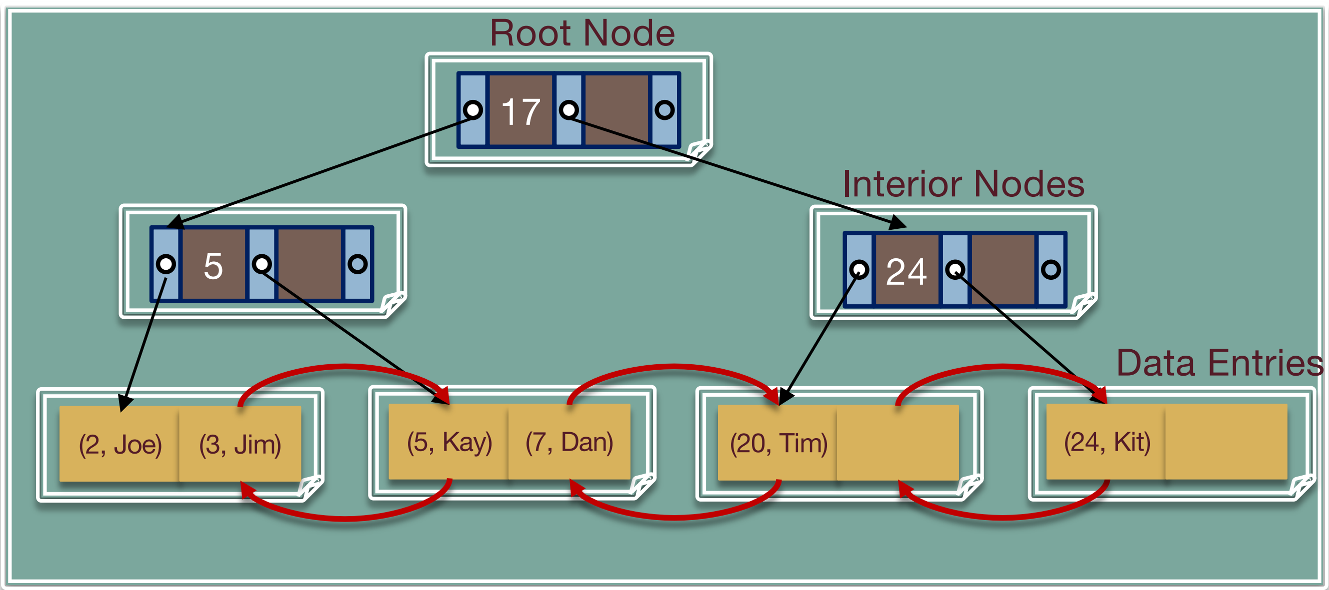
- Three basic alternatives for data entries in any index
  - Alternative 1: By Value
  - Alternative 2: By Reference
  - Alternative 3: By List of references

# Alternative 1 Index (B+ Tree)



| uid | name |
|---|---|
| 2 | Joe |
| 3 | Jim |
| 5 | Kay |
| 7 | Dan |
| 20 | Tim |
| 24 | Kit |

- Record contents are stored in the index file
  - No need to follow pointers

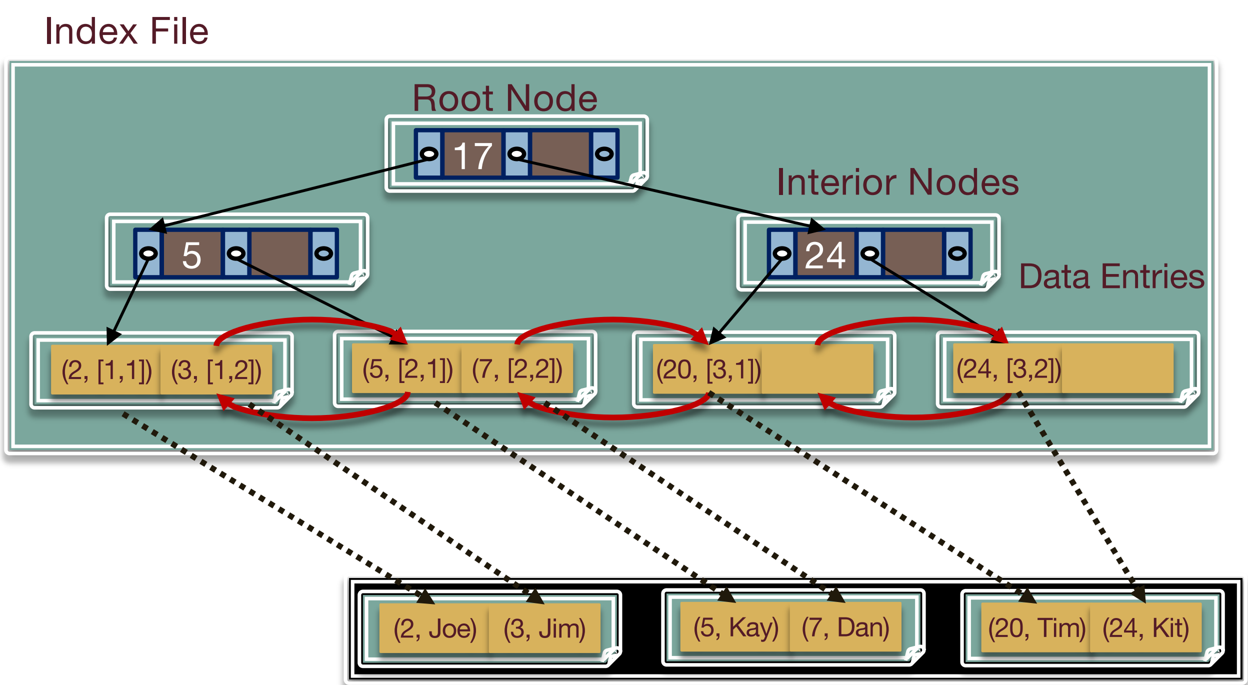


# Alternative 2 Index



| uid | name |
|---|---|
| 2 | Joe |
| 3 | Jim |
| 5 | Kay |
| 7 | Dan |
| 20 | Tim |
| 24 | Kit |

- Alternative 2: **By Reference, <k,** rid of matching data record>

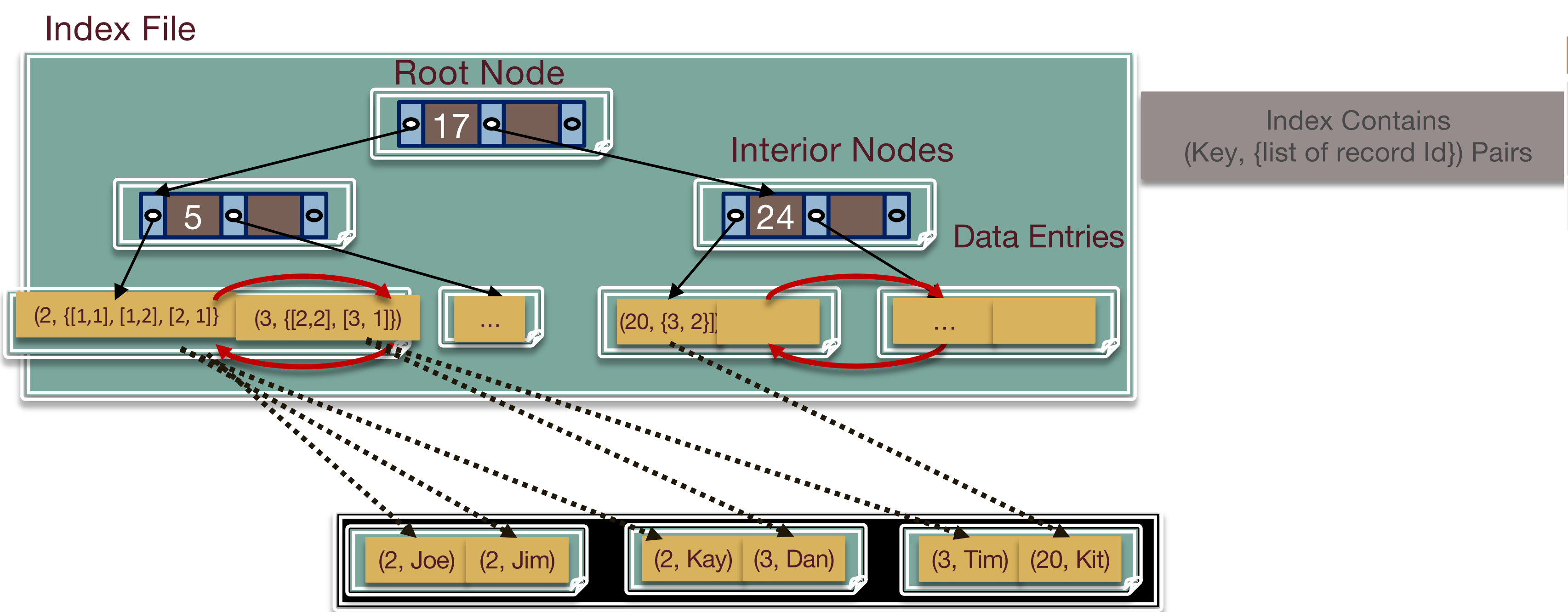



Index Contains (Key, Record Id) Pairs

# Alternative 3 Index



- Alternative 3: **By List of references,** <**k**, list of rids of matching data records>
  - Alternative 3 more compact than alternative 2
    - For very large rid lists, single data entry spans multiple blocks




| Key | Record Id |
|---|---|
| 2 | {[1,1], [1,2], [1,3]} |
| 3 | 4 |

# Indexing By Reference



- Both Alternative 2 and Alternative 3 index data *by reference*
- By-reference is *required* to support multiple indexes per table
  - Otherwise, we would be replicating entire tuples
  - Replicating data leads to complexity when we're doing updates, so it's something we want to avoid

# Alternative 2 vs Alternative 3 Table Illustration



Alternative 2
Index data entries

| Key | Record Id |
|---|---|
| Gonzalez | [3, 1] |
| Gonzalez | [3, 2] |
| Gonzalez | [3, 3] |
| Hong | [3, 4] |

| SSN | Last Name | First Name | Salary |
|---|---|---|---|
| 123 | Gonzalez | Amanda | $400 |
| 443 | Gonzalez | Joey | $300 |
| 244 | Gonzalez | Jose | $140 |
| 134 | Hong | Sue | $400 |

Alternative 3
Index data entries

| Key | Record Id |
|---|---|
| Gonzalez | [3, {1, 2, 3}] |
| Hong | [3,4] |