

Tree Indexes

Alvin Cheung
Fall 2022
Reading: R & G Chapter 10



Simple Idea?

Input
Heap
File



- **Step 1:** Sort heap file & leave some space
 - Pages physically stored in logical order (sequential access)
 - Maintenance as new records are added/deleted is a pain, can lead to B updates in the worst case (move everything down or up)



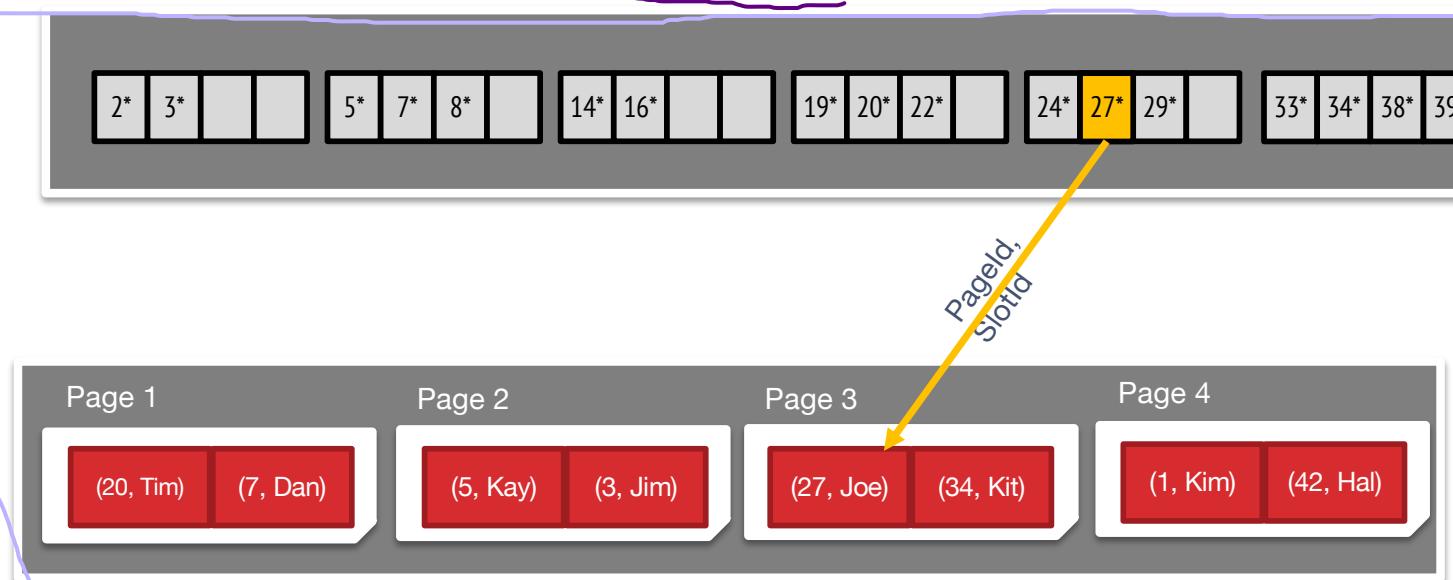
One sorted
lookup page

- **Step 2:** Use binary search on this sorted heap file: $\log_2(B)$ pages read
 - Fan-out of 2 → deep tree → lots of I/Os
 - Examine entire records just to read key during search: would prefer $\log_2(K)$ where K is number of pages to store keys $\ll B$

Let's fix these assumptions

- Idea: Keep **separate (compact) key lookup pages**, laid out sequentially
 - Maintain **key → recordID mapping** [We'll revisit this later]
- No need to sort heap file anymore! Just sort key lookup pages
- Use binary search on **lookup pages** as opposed to on all of the data pages
 - Still have a deep tree due to fan-out of 2 → **lots of I/Os**
- Also, maintenance of the key lookup pages is a pain! Worst case K updates

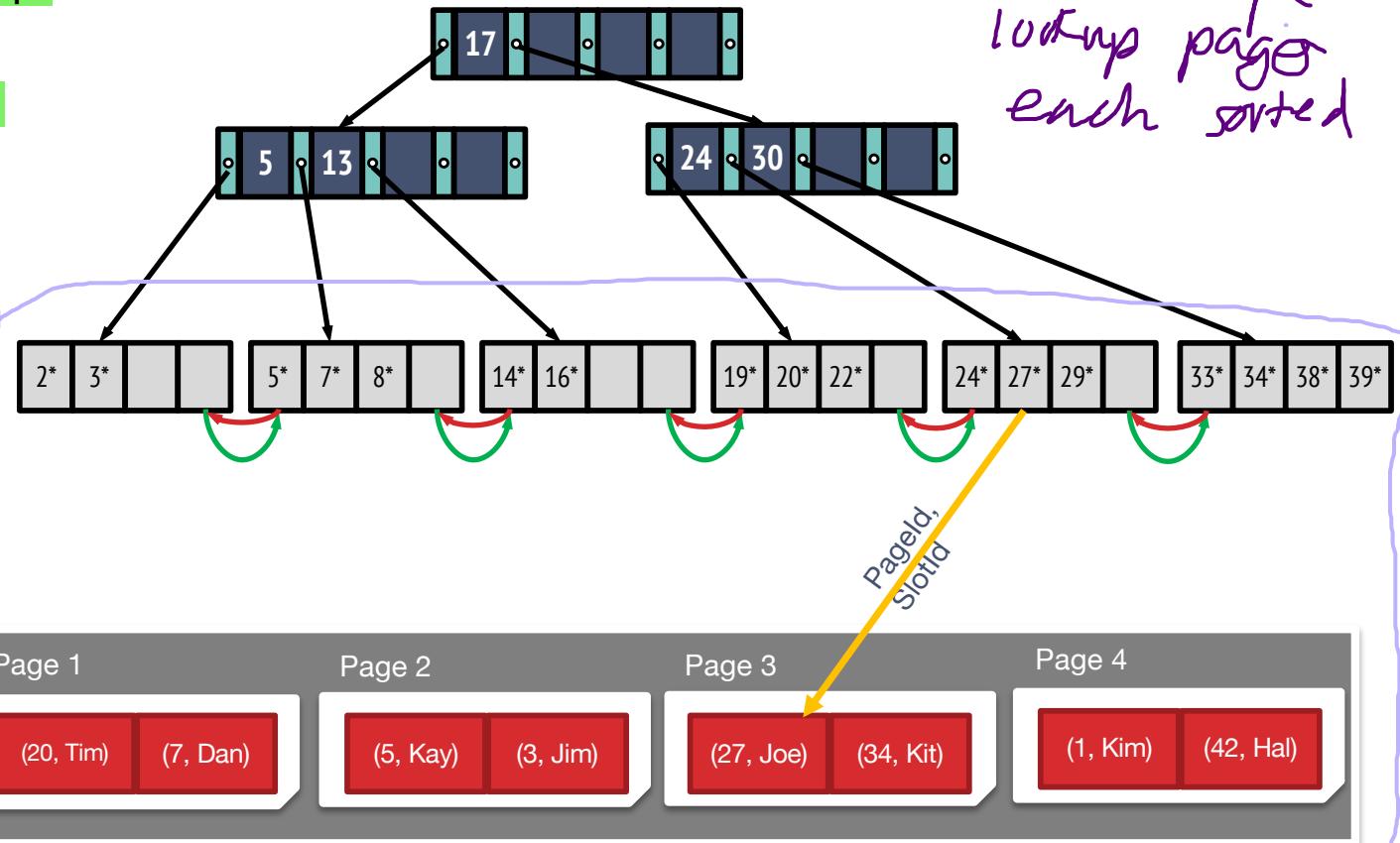
Multiple
lookup pages
each
sorted



Let's fix these assumptions, take 2

- **Idea:** repeat the process!
- **Lookup pages for the lookup pages**
- And then **lookup pages for the lookup pages for the lookup pages,**
- Let's set fanout to be $>> 2$
- That is essentially the idea behind B+ Trees ...
- We'll find out why the pointers are helpful later

one lookup page,
for multiple
lookup pages,
for multiple
lookup pages
each sorted



Enter the B+ Tree, More Formally

- **Dynamic Tree Index**
 - Always **Balanced**
 - **High fanout**
 - Support **efficient insertion & deletion**
 - Grows at root not leaves!
- “+”? B-tree that stores data entries in leaves only
 - Helps with range search

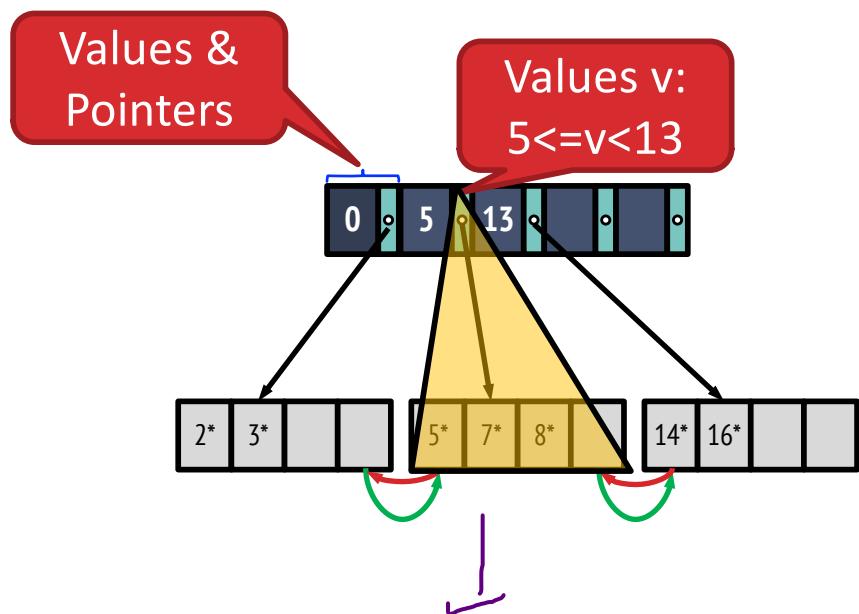
B+ tree = B-tree, but data entries in
leaves only

B+ Trees: How to Read an Interior Node

- $\text{Node}[\dots, (K_L, P_L), (K_R, P_R)\dots]$

means that

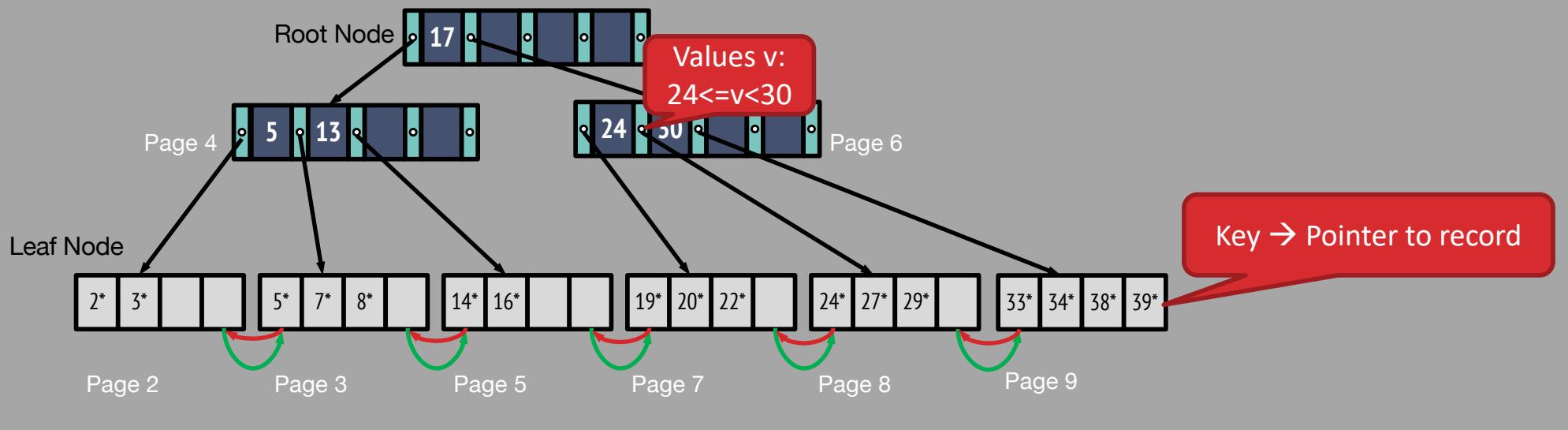
All tuples in range
 $K_L \leq K < K_R$ are in tree P_L



$$k_L \leq k < k_R$$

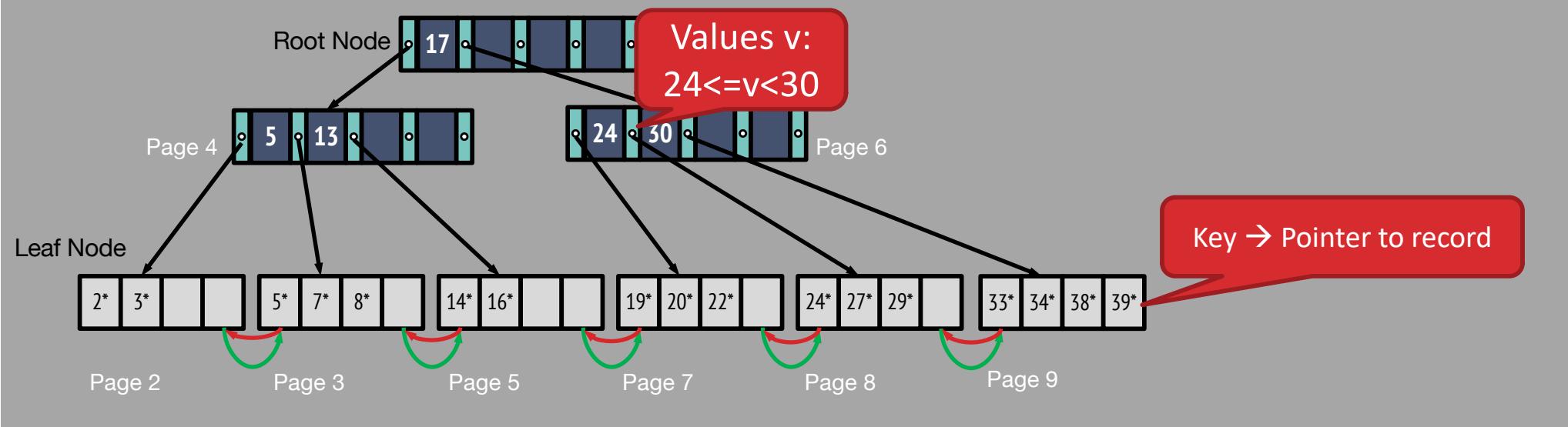
$$5 \leq 5, 7, 8 < 13$$

Example of a B+ Tree



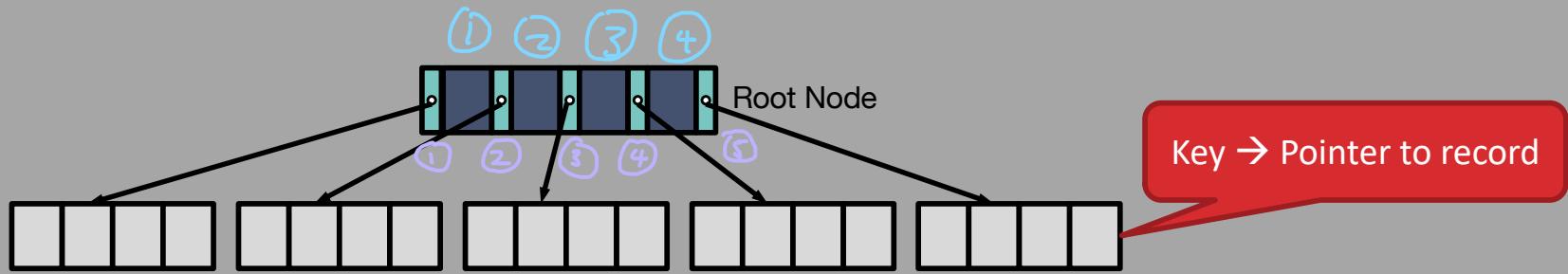
- Property 1: Nodes in a B+ tree must obey an **occupancy invariant**
 - Guarantees that **lookup costs are bounded**
 - Invariant:** each interior node is full beyond a certain minimum: in this case [and typically], **at least half full**
 - This minimum, d , is called the **order of the tree**
 - Here, **max # of entries = 4**. Thus $d = 2$.
 - Guarantee:** $d \leq \# \text{ entries} \leq 2d$. In this tree, $2 \leq \# \text{ entries} \leq 4$
 - Root doesn't need to obey this invariant
 - Same invariant holds for leaf nodes: at least half full (d may differ, here it is the same)

Example of a B+ Tree



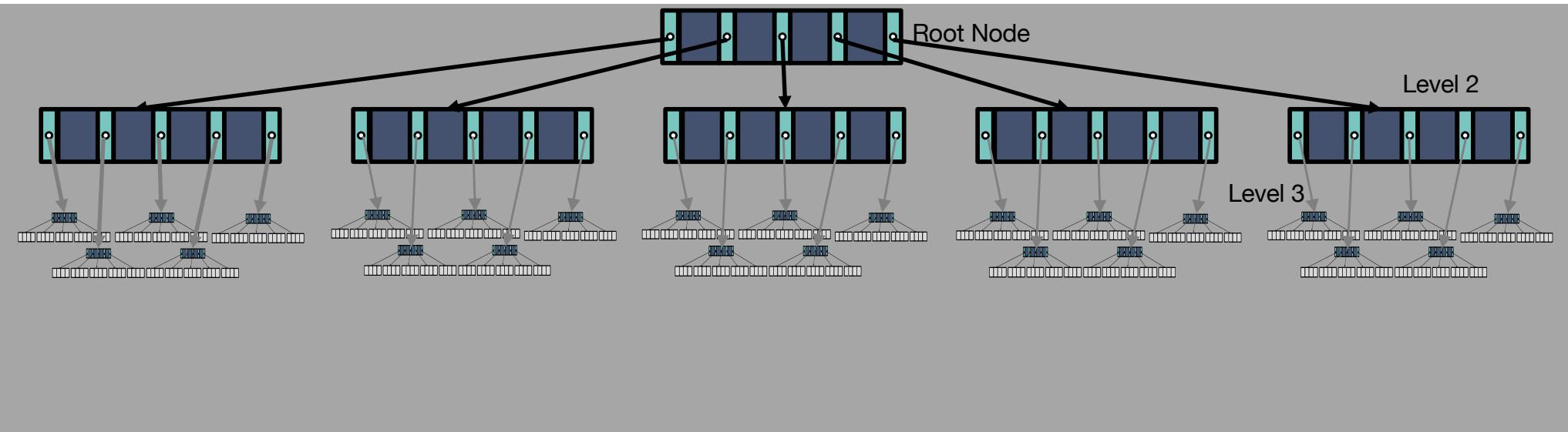
- **Property 1:** Nodes in a B+ tree must obey an **occupancy invariant**
 - Each interior/leaf node is full beyond a certain minimum d
- **Property 2:** Leaves need not be stored in sorted order (but often are)
 - Next and prev. pointers help examining them in sequence [useful as we will see soon]

B+ Trees and Scale



- How many records can this height 1 B+ tree index?
 - Max entries = 4; Fan-out (# of pointers) = 5
 - Height 1: 5 (pointers from root) x 4 (slots in leaves) = 20 Records

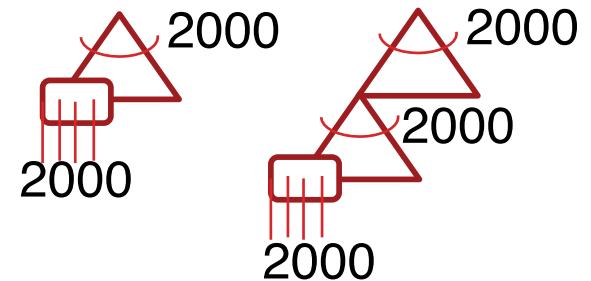
B+ Trees and Scale Part 2



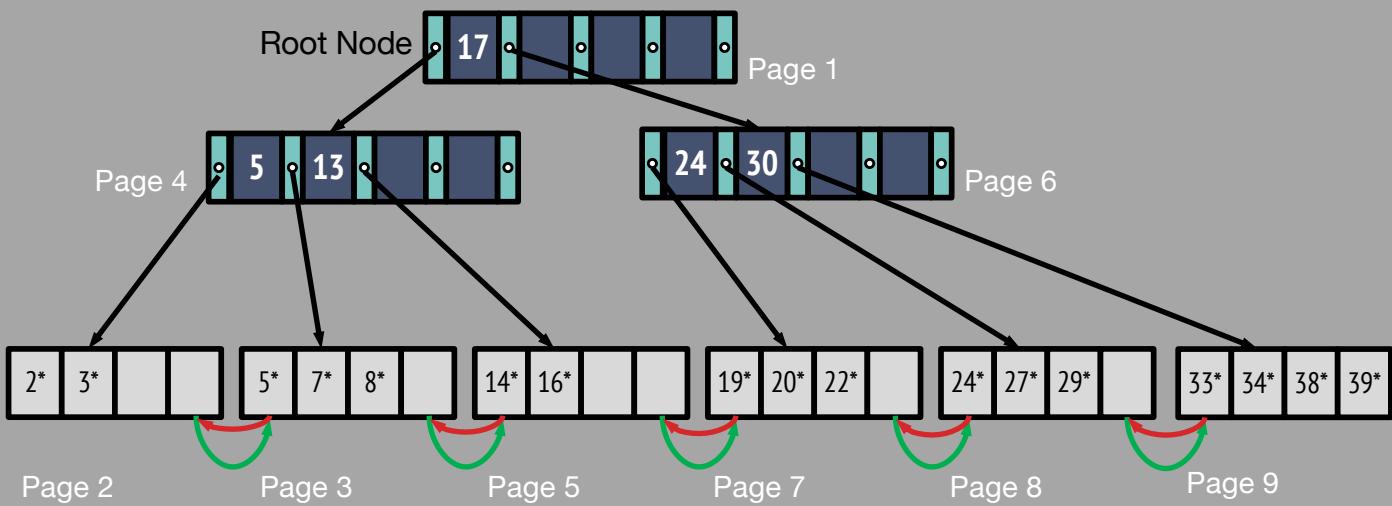
- How many records can this height 3 B+ tree index?
 - Fan-out = 5; Max entries = 4
 - **Height 3:** $5 \text{ (root)} \times 5 \text{ (level 2)} \times 5 \text{ (level 3)} \times 4 \text{ (leaves)} = 5^3 \times 4 = 500 \text{ Records}$

Extending this: B+ Trees in Practice

- (Warning: Sloppy back-of-the-envelope calculation!)
- Say 128KB pages, with around 40B per (val, ptr) pair
 - Max entries = roughly $128\text{KB}/40\text{B} = \text{approx. } 3000$
 - Max fanout = $3000+1 = \text{approx. } 3000$
 - Say 2/3 are filled on average
 - Average fan-out/entries = approx. 2000
- At these capacities
 - Height 1: $2000 \text{ (pointers from root)} \times 2000 \text{ (entries per leaf)} = 2000^2 = \mathbf{4,000,000}$
 - Height 2: $2000 \text{ (pointers from root)} \times 2000 \text{ (pointers from level 2)} \times 2000 \text{ (entries per leaf)} = 2000^3 = \mathbf{8,000,000,000 \text{ records!!}}$
- **Core takeaway: Even depths of 3 allow us to index a massive # of records!**



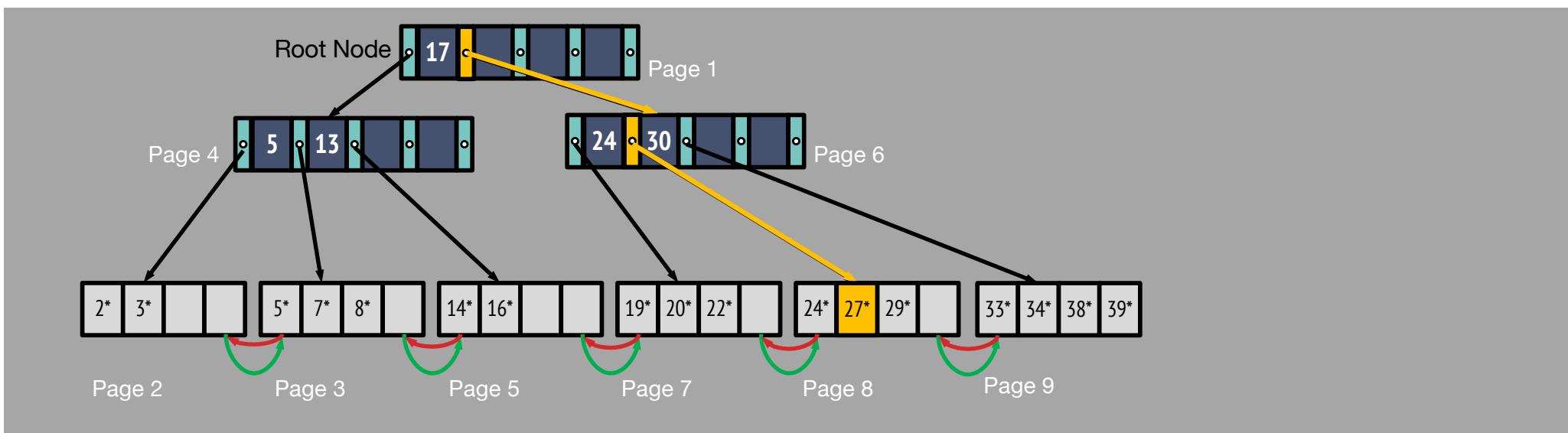
Searching the B+ Tree



- Procedure:
 - Find split on each node (Binary Search)
 - Follow pointer to next node
- { Repeat until find leaf node*

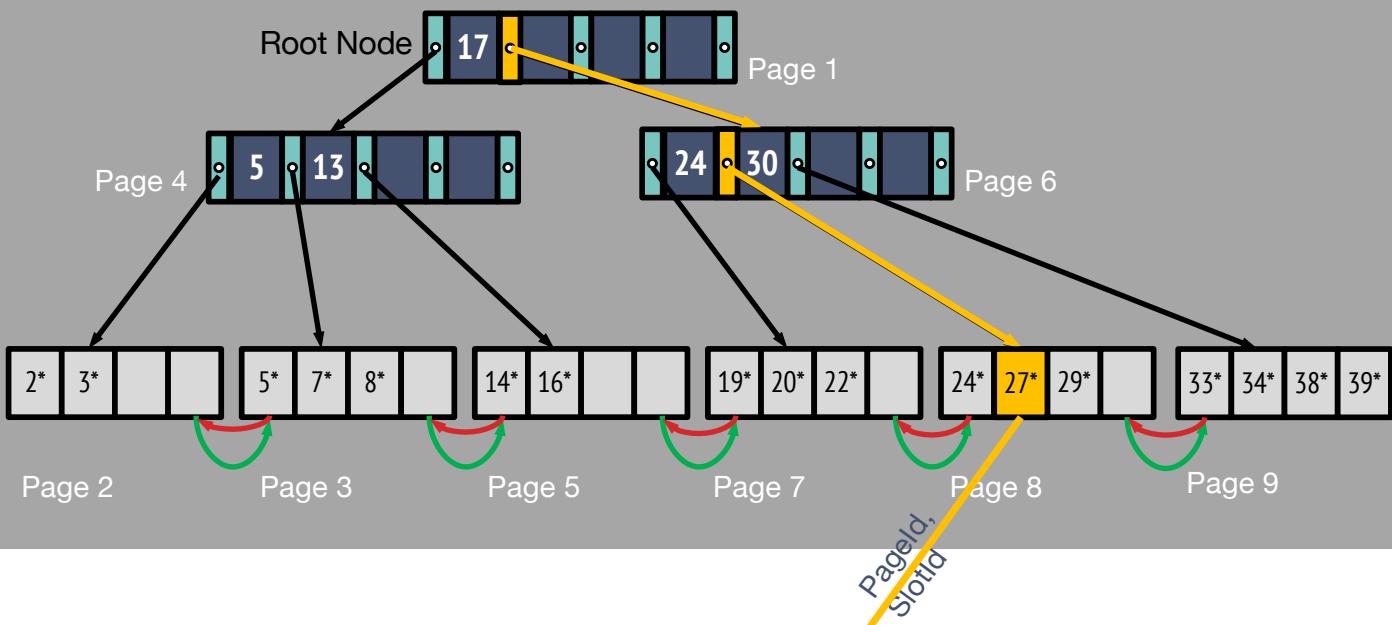
Search example

Searching the B+ Tree: Find 27



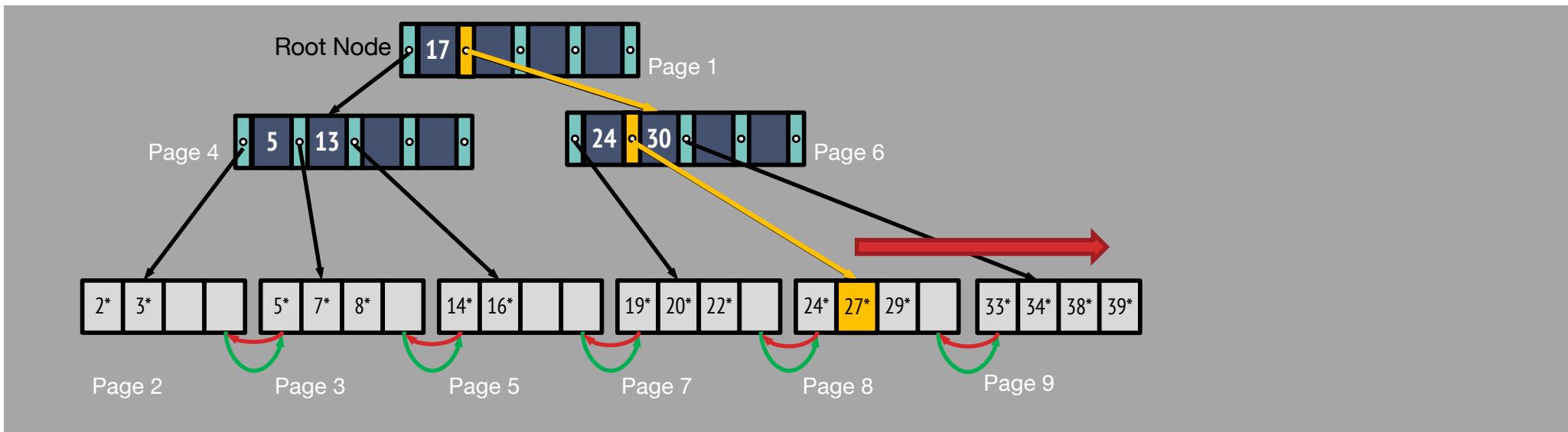
- Find key = 27
 - Find split on each node (Binary Search)
 - Follow pointer to next node

Searching the B+ Tree: Fetch Data



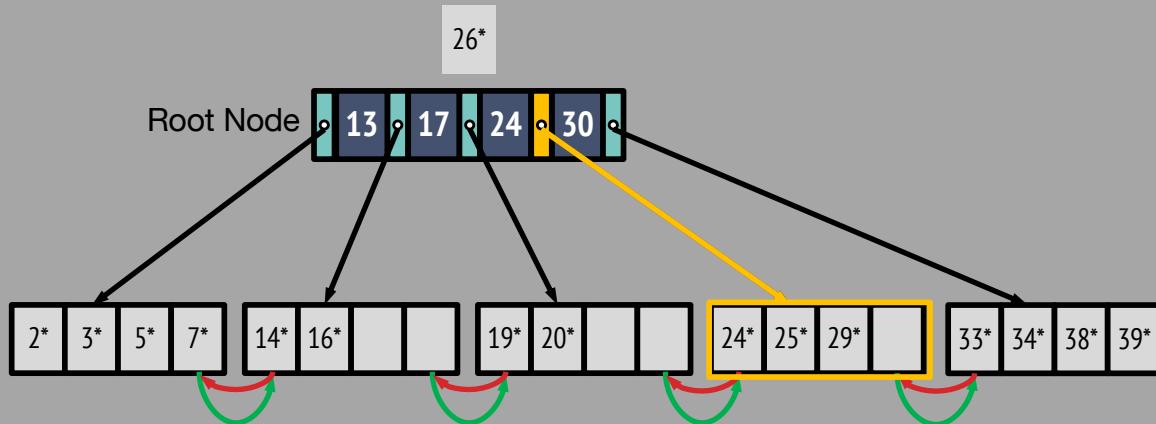
Range Search example

Searching the B+ Tree: Find 27 and up



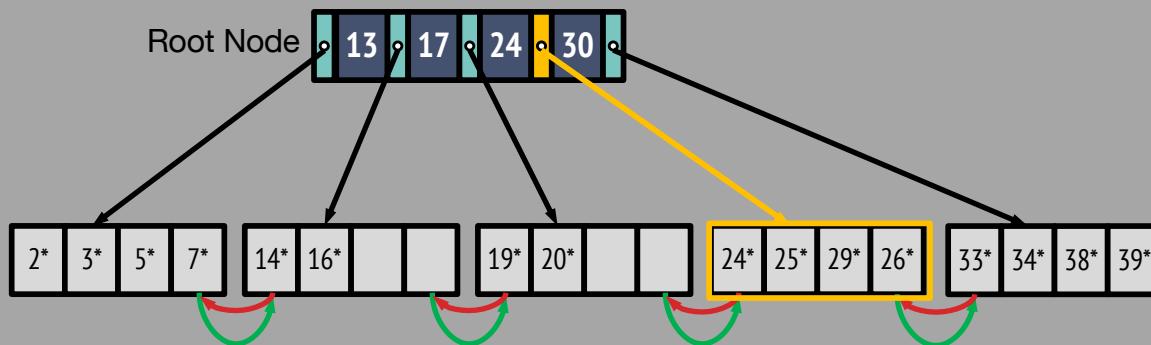
- Find keys ≥ 27
 - Find 27 first, then traverse leaves following “next” pointers in leaf
 - This is an example of a *range scan*: find all values in $[a, b]$
 - Benefit: no need to go back up the tree! Saves I/Os

Inserting 26* into a B+ Tree Part 1



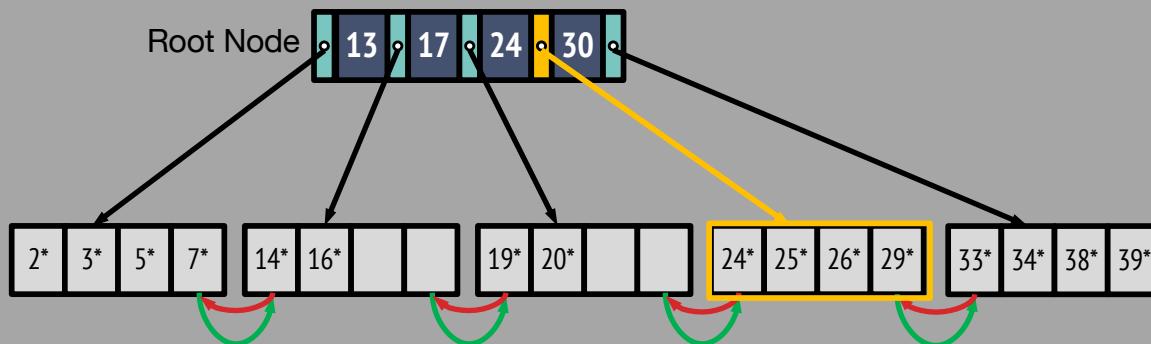
- Find the correct leaf

Inserting 26* into a B+ Tree Part 2



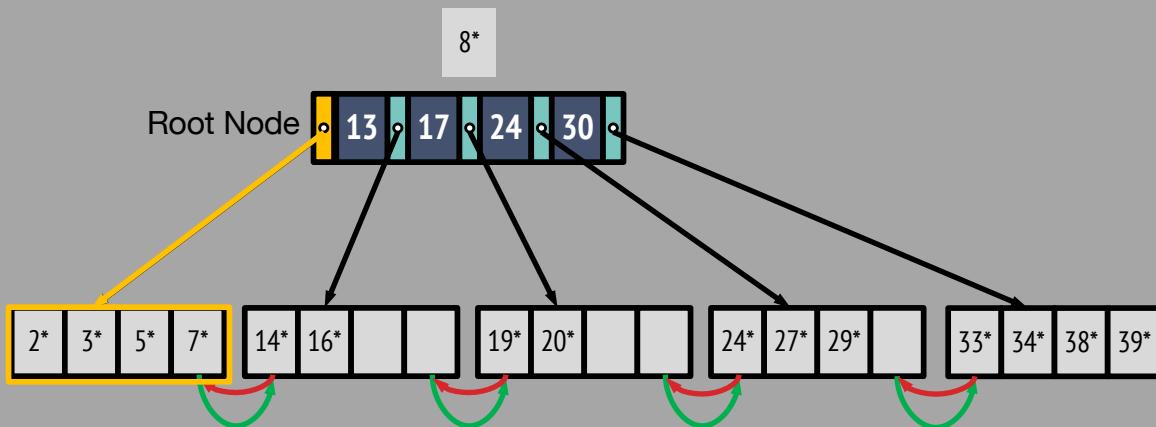
- Find the correct leaf
- If there is room in the leaf just add the entry

Inserting 26* into a B+ Tree Part 3



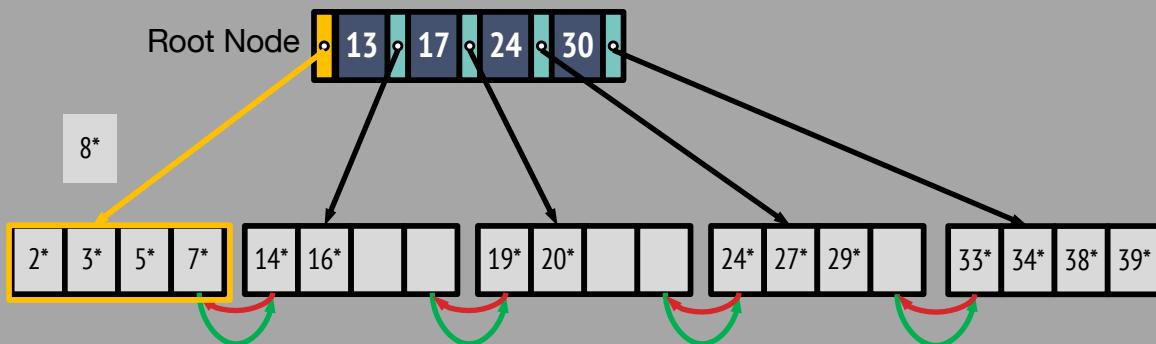
- Find the correct leaf
- If there is room in the leaf just add the entry
 - Sort the leaf page by key

Inserting 8* into a B+ Tree: Find Leaf



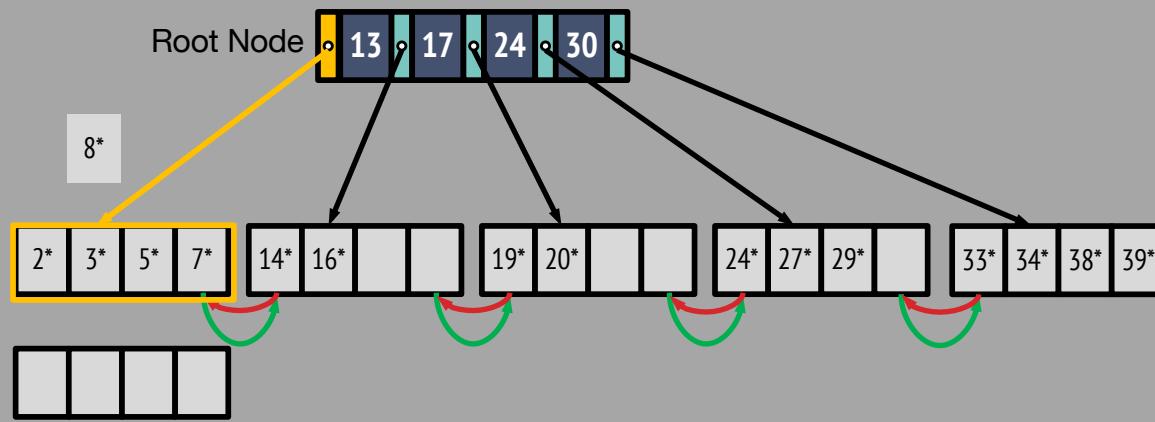
- Find the correct leaf

Inserting 8* into a B+ Tree: Insert



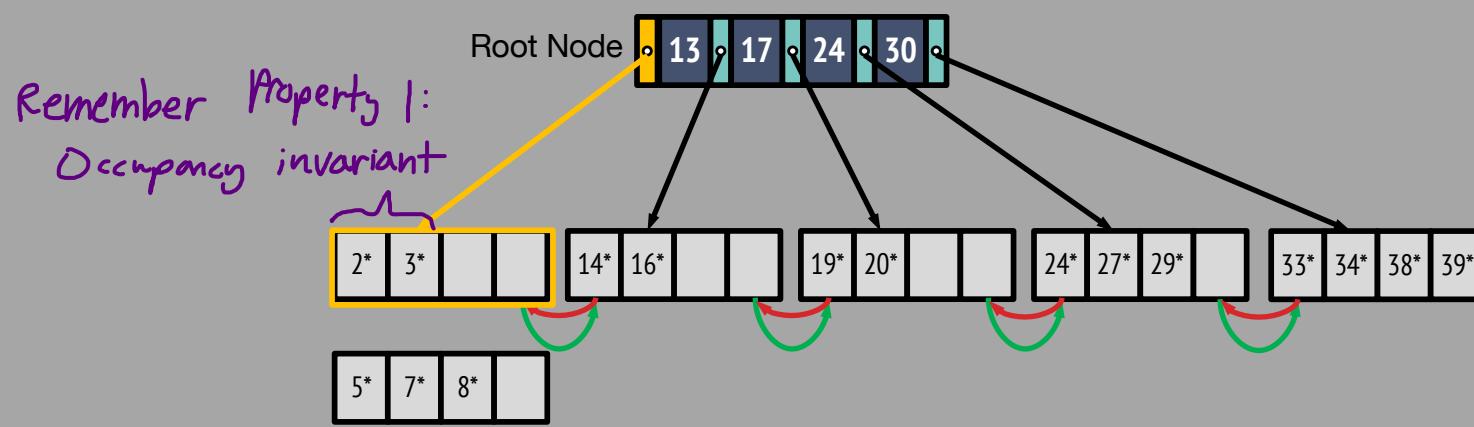
- Find the correct leaf
 - Split leaf if there is not enough room

Inserting 8* into a B+ Tree: Split Leaf



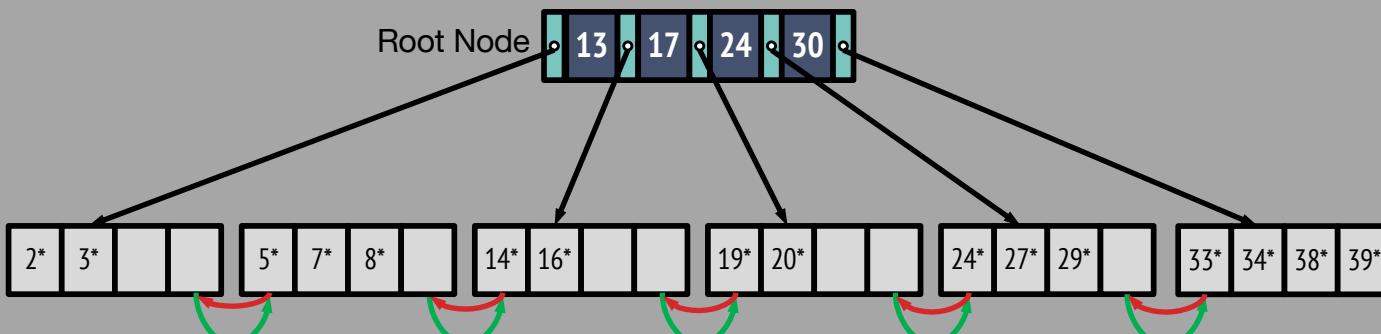
- Find the correct leaf
 - Split leaf if there is not enough room
 - Redistribute entries evenly

Inserting 8* into a B+ Tree: Split Leaf, cont



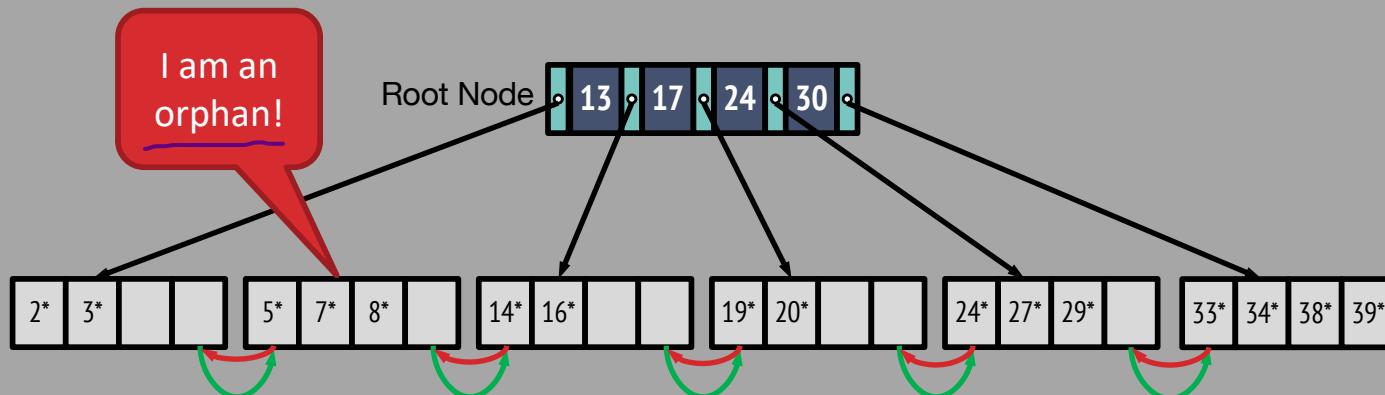
- Find the correct leaf
 - Split leaf if there is not enough room
 - Redistribute entries evenly
 - Fix next/prev pointers

Inserting 8* into a B+ Tree: Fix Pointers



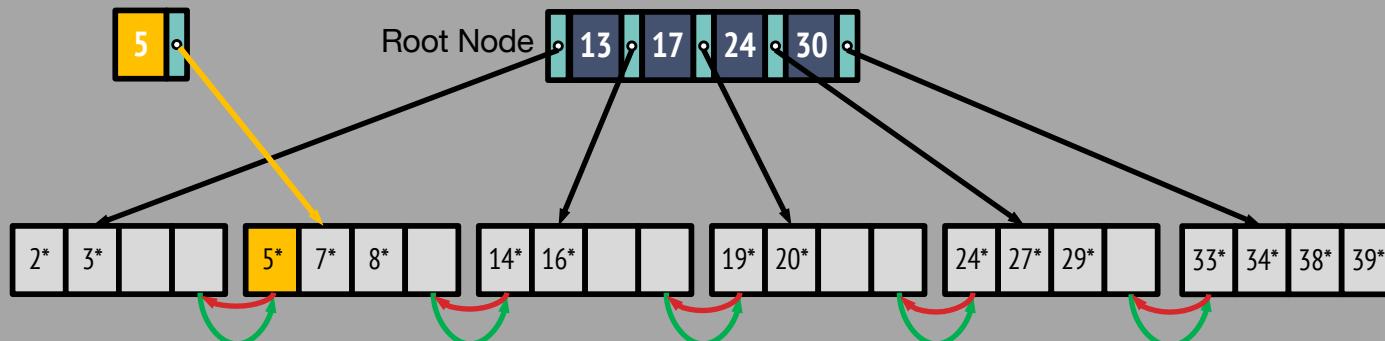
- Find the correct leaf
 - Split leaf if there is not enough room
 - Redistribute entries evenly
 - Fix next/prev pointers

Inserting 8* into a B+ Tree: Mid-Flight



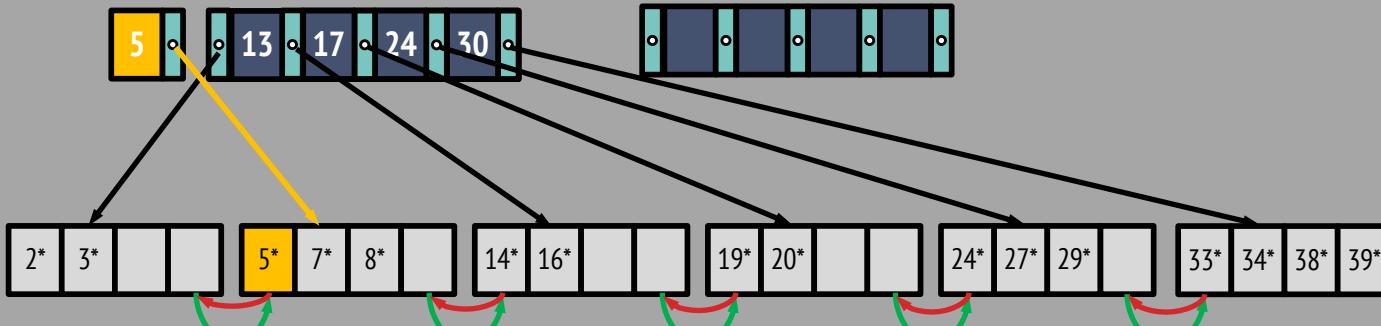
- Something is still wrong!

Inserting 8* into a B+ Tree: Copy Middle Key



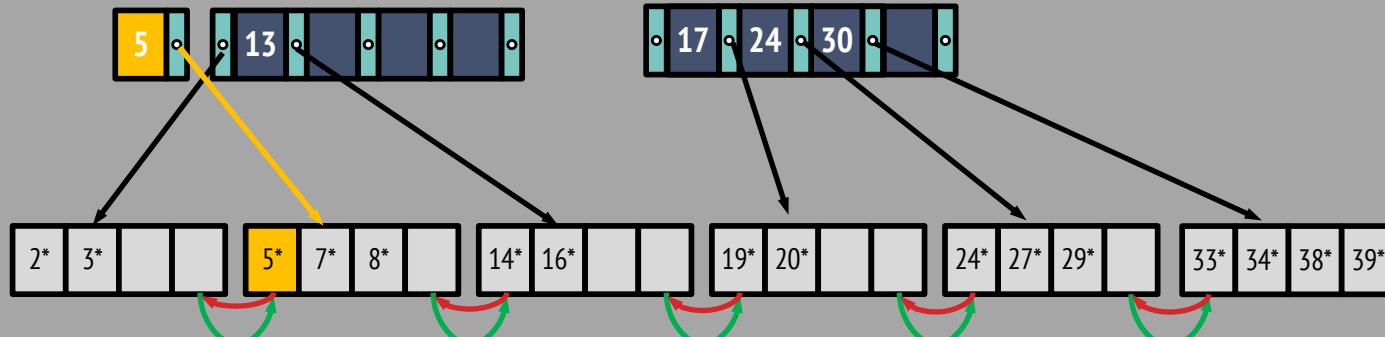
- **Copy up from leaf the middle key and pointer to the orphan leaf**
 - This is what we need to access it

Inserting 8* into a B+ Tree: Split Parent, Part 1



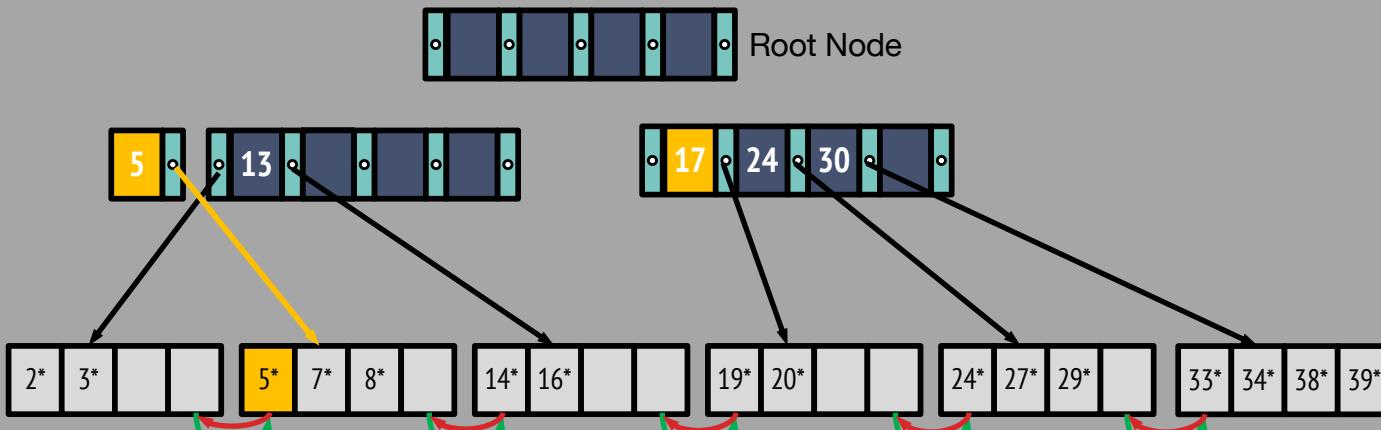
- Copy up from leaf the middle key and pointer to the orphan leaf
- No room in parent? (Parent now has $2d+1$ instead of $2d$)
 - Recursively split index nodes
 - Redistribute the rightmost $d+1$ keys

Inserting 8* into a B+ Tree: Split Parent, Part 2



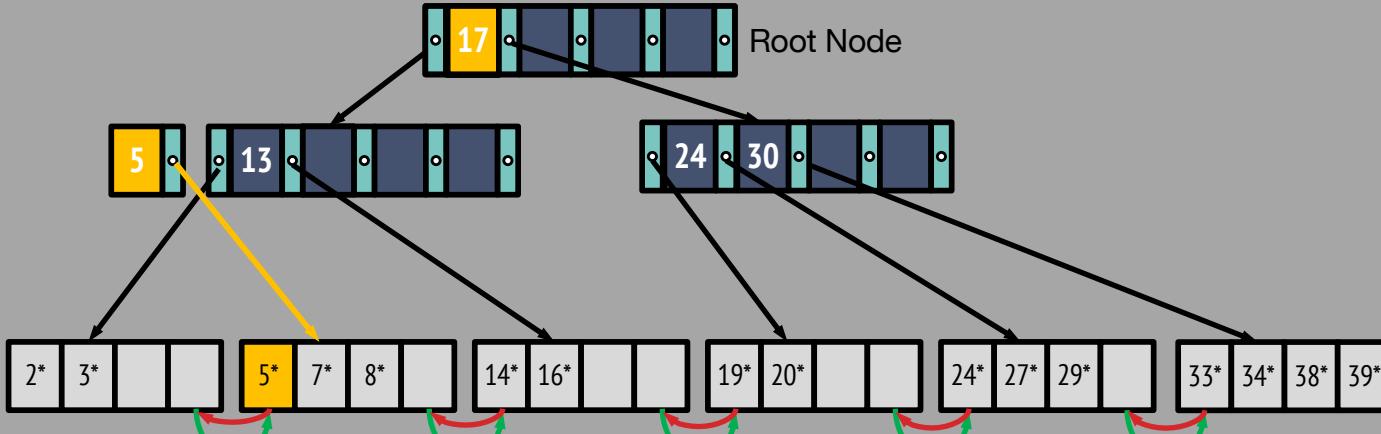
- Copy up from leaf the middle key and pointer to the orphan leaf
- No room in parent? Recursively split index nodes
 - Redistribute the rightmost $d+1$ keys
 - Not enough: we now have two roots!

Inserting 8* into a B+ Tree: Root Grows Up



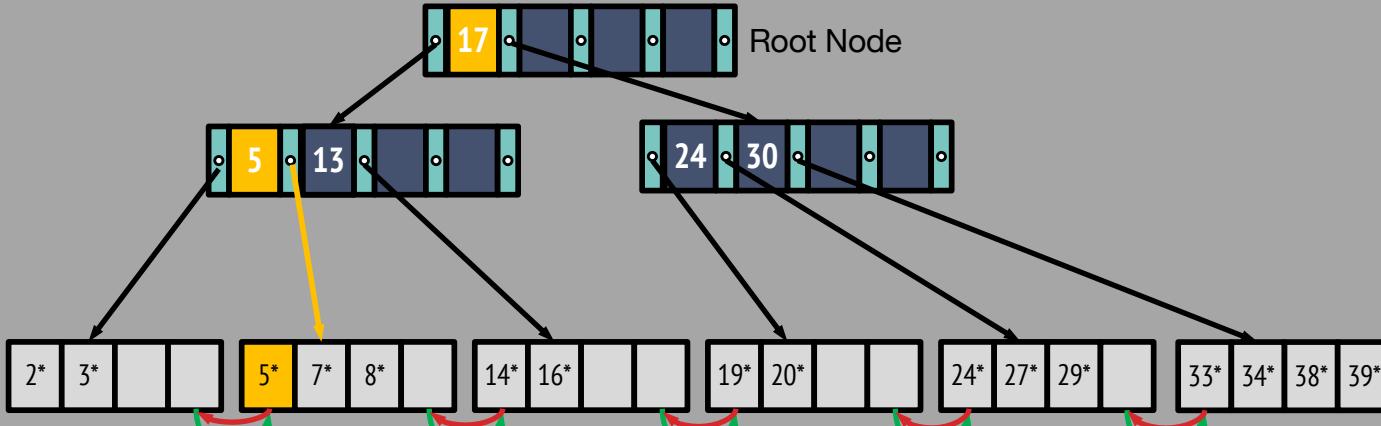
- No room in parent? Recursively split index nodes
 - Redistribute the rightmost $d+1$ keys
- **To fix, create a new root:**
 - **Push up from interior node** the middle key (and assoc. pointer)

Inserting 8* into a B+ Tree: Root Grows Up, Pt 2



- Net effect
 - d keys on the left and right => invariant satisfied!
 - middle key pushed up
- Consolidate 5* into left node

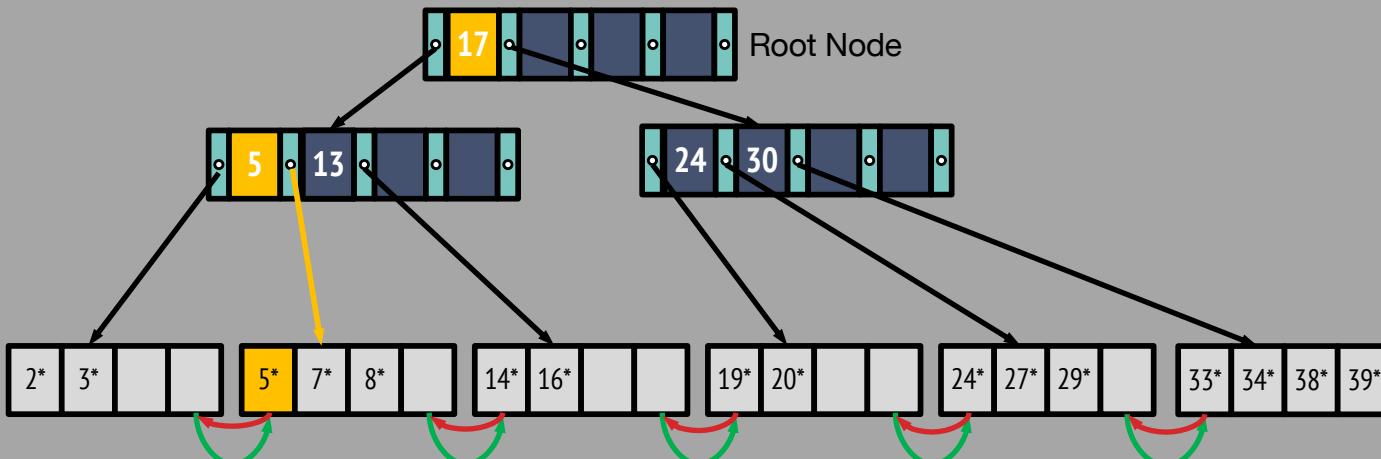
Inserting 8* into a B+ Tree: Root Grows Up, Pt 3



- Net effect
 - d keys on the left and right
 - middle key pushed up
- **Here, we ended up creating a new root and increasing depth => rare**

B_iggest difference between B and
B+ trees

Copy up vs Push up!



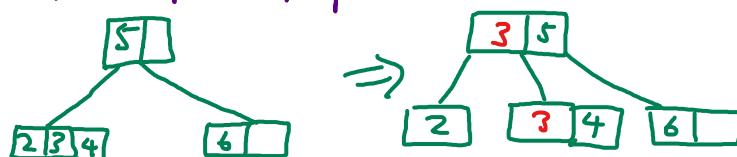
The leaf entry (5) was **copied** up

We can't lose the original key: all keys must be in leaves

The index entry (17) was **pushed** up

We don't need it any more for routing => convince yourself!

Rule: when leaf copied up, preserve leaf as so:

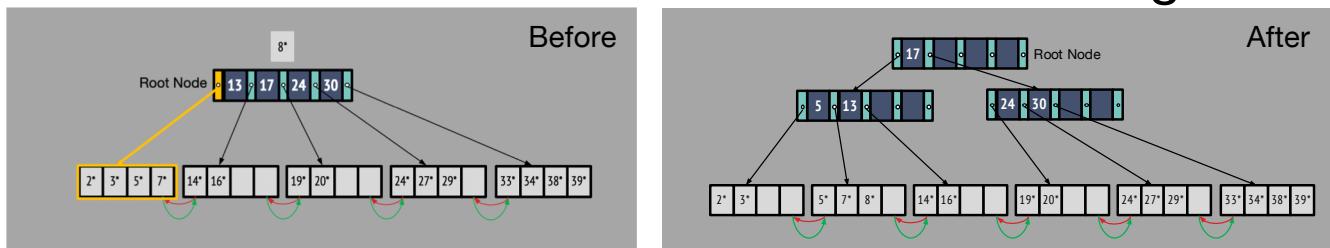


B+ Tree Insert: Algorithm Sketch

1. Find the correct leaf L.
2. Put data entry onto L.
 - If L has enough space, done!
 - Else, must split L (into L and a new node L2)
 - Redistribute entries evenly, copy up middle key (and ptr to L2)
 - Insert index entry pointing to L2 into parent of L.

B+ Tree Insert: Algorithm Sketch Part 2

- Step 2 can happen recursively
 - To split index node, redistribute entries evenly, but push up middle key (and ptr to new index node). (Contrast with leaf splits)
- Splits “grow” tree
 - Tree growth: gets wider if possible from bottom up
 - Worst case, adds another level with a new root
 - Ensures balance & therefore the logarithmic guarantee



We will skip deletion

- In practice, occupancy invariant often not enforced during deletion
- Just delete leaf entries and leave space
 - If new inserts come, great
 - This is common
- If page becomes completely empty, can delete
 - Parent may become underfull
 - That's OK too
- No need to delete inner pages even if empty
 - Only used for lookups
- Guarantees still attractive: $\log_F(\text{total number of inserts})$
- Textbook describes algorithm for rebalancing and merging on deletes

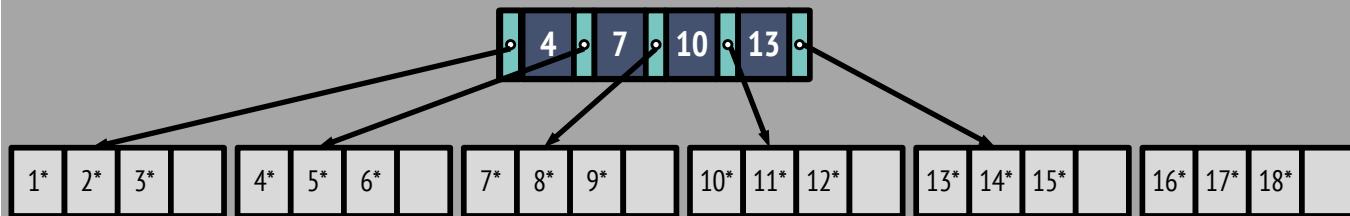
BULK LOADING B+-TREES

*pass in many values in
short amount of time*

Bulk Loading of B+ Tree

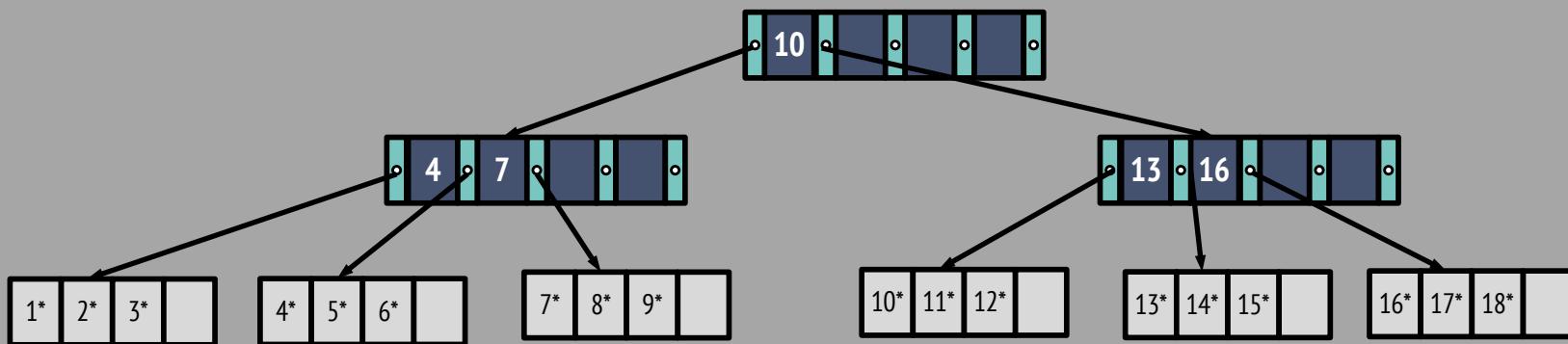
- Suppose we want to build an index on a large table from scratch
- Would it be efficient to just call insert repeatedly
 - Q: No ... Why not?
 - Constantly need to search from root
 - Modifying random pages: poor cache efficiency
 - Leaves poorly utilized (typically half-empty)

Smarter Bulk Loading a B+ Tree



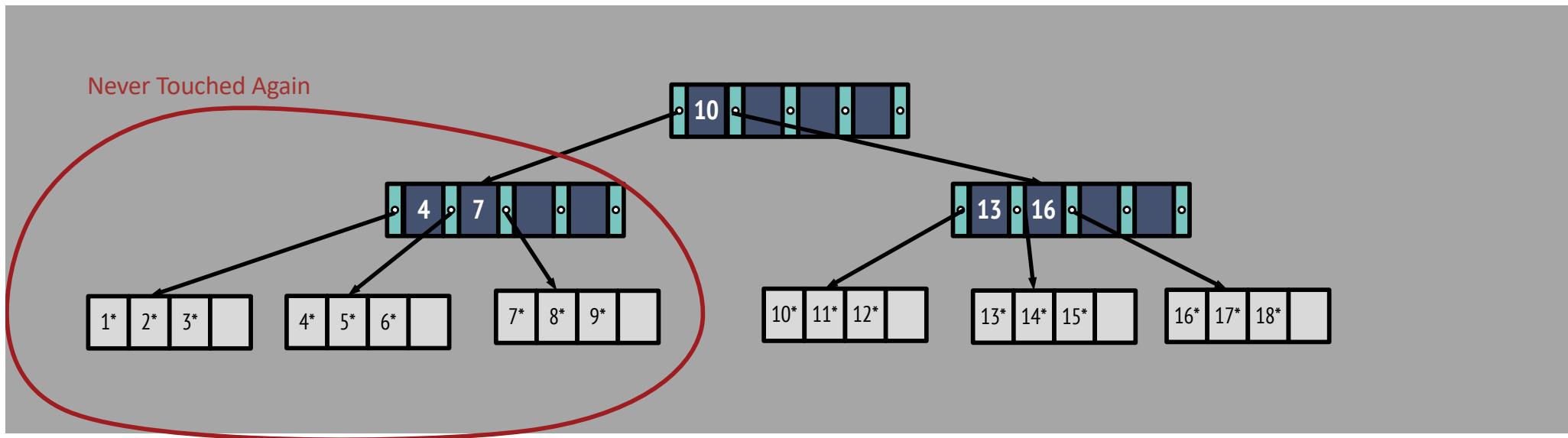
- Sort the input records by key:
 - 1*, 2*, 3*, 4*, ...
 - We'll learn a good disk-based sort algorithm soon!
- Fill leaf pages to some fill factor > d (e.g. $\frac{3}{4}$)
 - Updating parent until full

Smarter Bulk Loading a B+ Tree Part 2



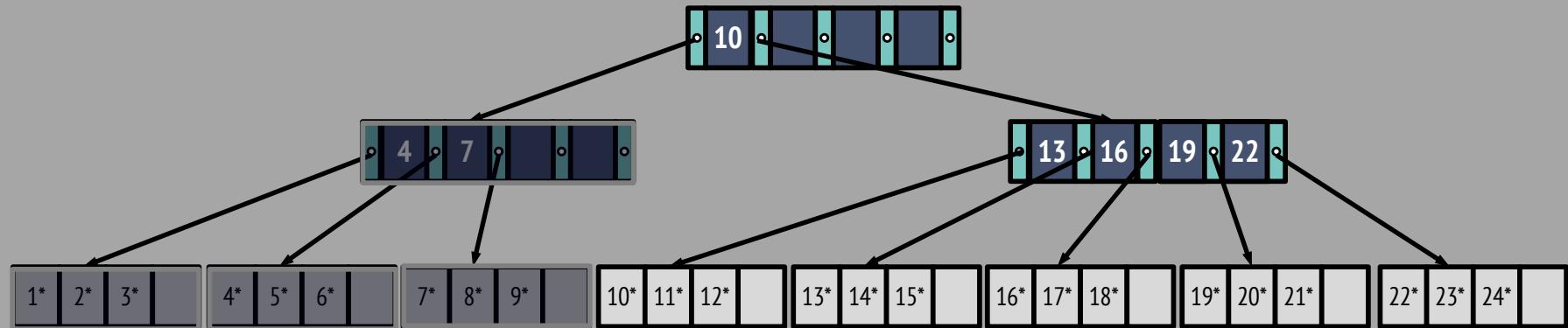
- Sort the input records by key:
 - 1*, 2*, 3*, 4*, ...
- Fill leaf pages to some fill factor > d (e.g. $\frac{3}{4}$)
 - Update parent until full
 - Then create new sibling and copy over half: same as in index node splits for insertion

Smarter Bulk Loading a B+ Tree Part 3



- Lower left part of the tree is never touched again
- Occupancy invariant maintained:
 - leaves filled beyond d, rest of the nodes via insertion split procedure

Smarter Bulk Loading a B+ Tree Part 4



- Benefits: Better
 - Cache utilization than insertion into random locations
 - Utilization of leaf nodes (and therefore shallower tree)
 - Layout of leaf pages (more sequential)

Summary

- B+ Tree is a powerful dynamic indexing structure
 - Inserts/deletes leave tree height-balanced; $\log_F N$ cost
 - High fanout (F) means height rarely more than 3 or 4.
 - Higher levels stay in cache, avoiding expensive disk I/O
 - Almost always better than maintaining a sorted file.
 - Widely used in DBMSs!
- Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set.