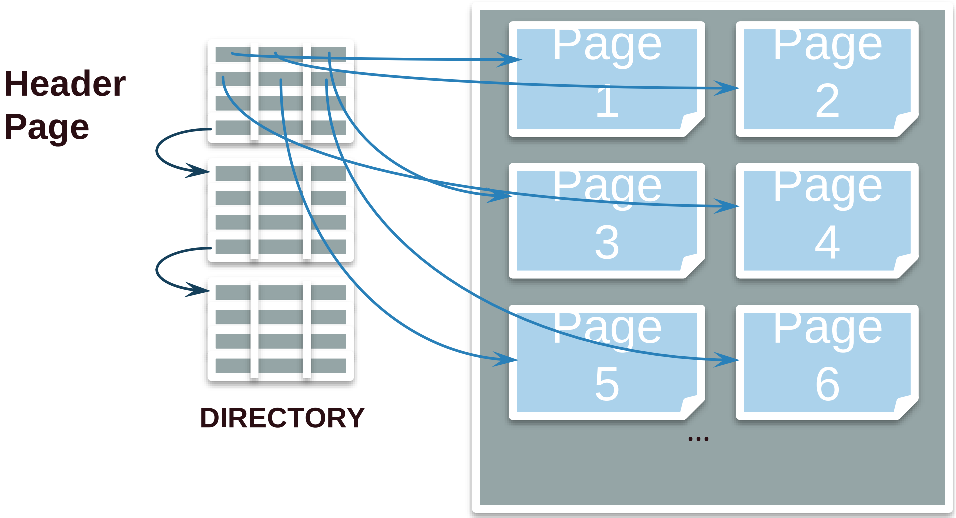
Tree Indexes

R & G - Chapter 10

Reminder on Heap Files

- Two access APIs:
  - fetch by recordId (pageId, slotId)
  - scan (starting from some page)



Wouldn’t it be nice…

- …if we could look things up by value?
- Toward a Declarative access API
- But … efficiency?

“If you don't find it in the index, look very carefully through the entire catalog. ”

—Sears, Roebuck, and Co., Consumers' Guide, 1897

We’ve seen this before

- Data structures … in RAM:
  - Search trees (Binary, AVL, Red-Black, …)
  - Hash tables

- Needed: disk-based data structures
  - “paginated”: made up of disk pages!

Index

An **index** is data structure that enables fast **lookup** and **modification** of **data entries** by **search key**

- **Lookup**: may support many different operations
  - **Equality**, 1-d range, 2-d region, …
- **Search Key:** any subset of columns in the relation
  - Do not need to be unique
    - —e.g. (firstname) or (firstname, lastname)

Index Part 2

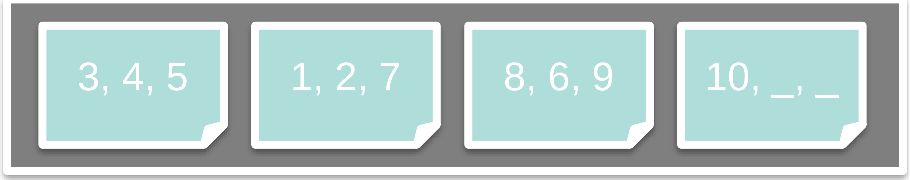
An **index** is data structure that enables fast **lookup** and **modification** of **data entries** by **search key**

- **Data Entries:** items stored in the index
  - Assume for today: a pair (**k**, recordId) …
    - Pointers to records in Heap Files!
    - Easy to generalize later
- **Modification:** want to support fast insert and delete

Many Types of indexes exist: B+-Tree, Hash, R-Tree, GiST, ...

Simple Idea?

Input
Heap
File

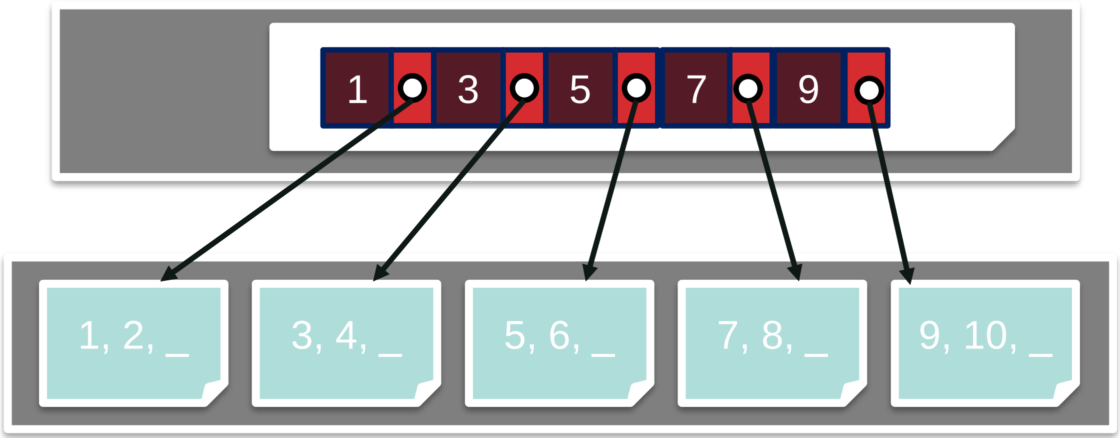


- **Step 1:** Sort heap file & leave some space
  - Pages physically stored in logical order (sequential access)
  - Do we need “next” pointers to link pages?
    - No. Pages are physically sorted in logical order

- **Step 2**: Build the index data structure over this…
  - Why not just use binary search in this heap file?
    - Fan-out of 2 → deep tree → lots of I/Os
    - Examine entire records just to read key during search

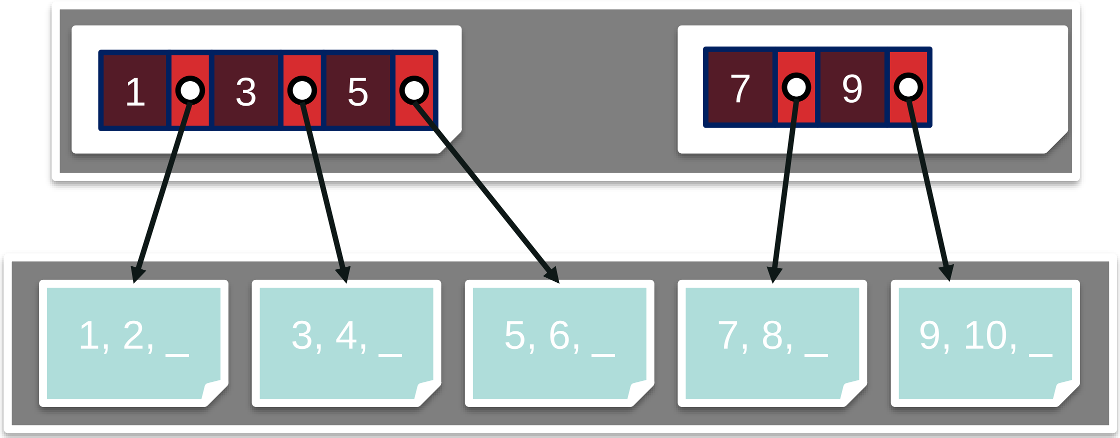
Build a high fan-out search tree

- Start simple: *Sorted (key, page id) file*
  - No record data
  - Binary search in the key file. Better!
  - **Forgot:** Need to break across pages!



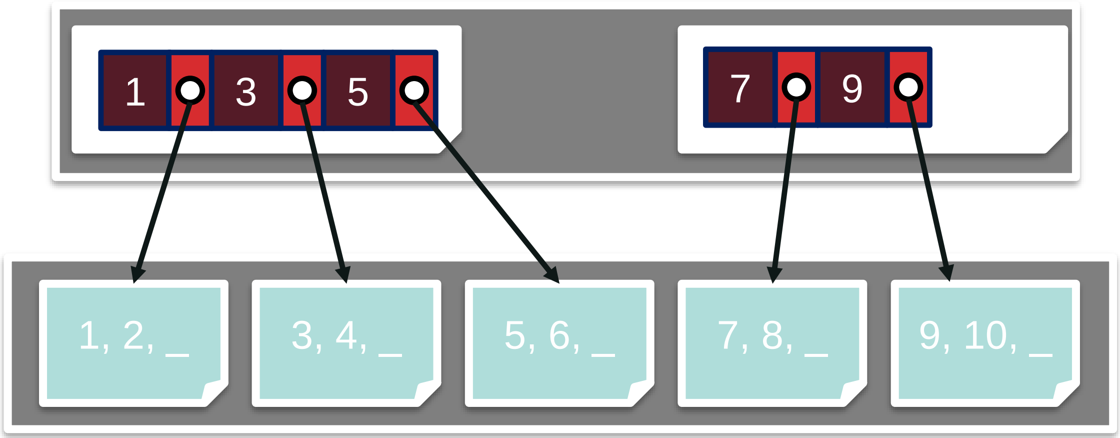
Build a high fan-out search tree

- Start simple: *Sorted (key, page id) file*
  - No record data
  - Binary search in the key file. Better!
  - **Forgot:** Need to break across pages!



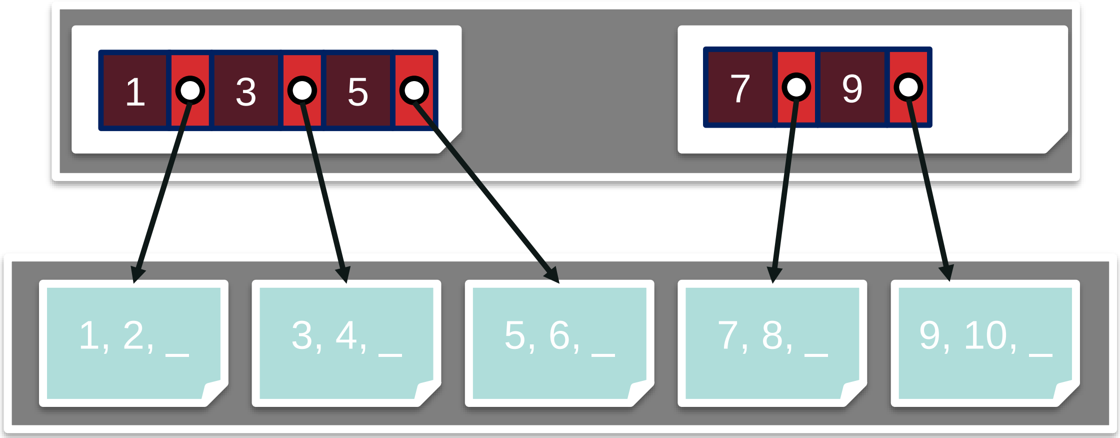
Build a high fan-out search tree Part 2

- Start simple: *Sorted (key, page id) file*
  - No record data
  - Binary search in the key file. Better!
  - **Complexity?**



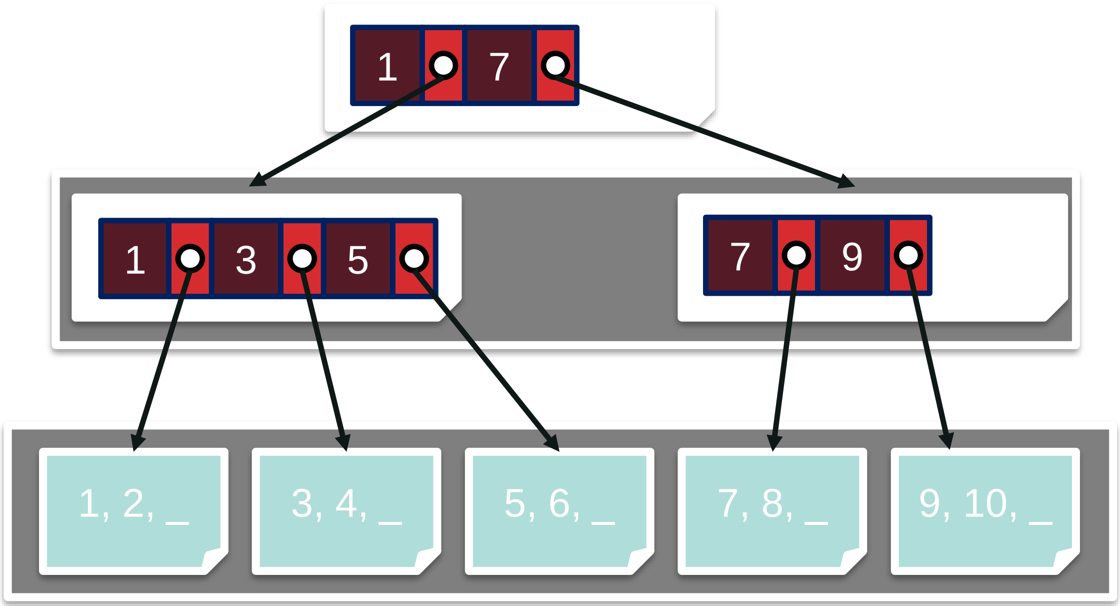
Build a high fan-out search tree Part 3

- Start simple: *Sorted (key, page id) file*
  - No record data
  - Binary search in the key file. Better!
  - **Complexity**: Still binary search, just a constant factor smaller input



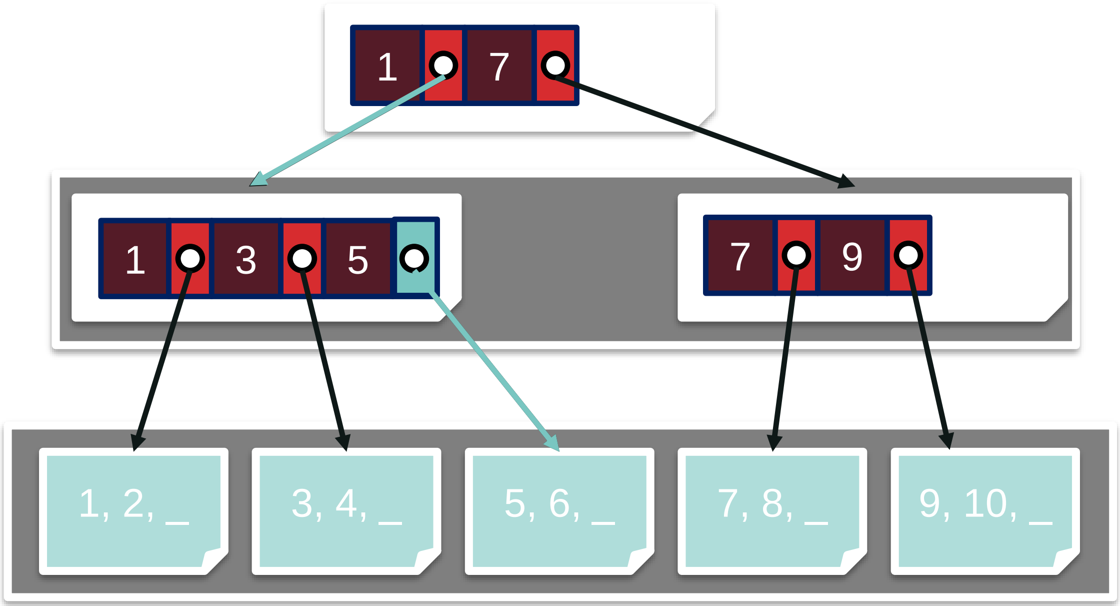
Build a high fan-out search tree Part 4

- Recursively “index” key file
- **Key Invariant:**
  - Node [..., (K_L, P_L), (K_R, P_R), ...] → All tuples in range $K_L <= K < K_R$ are in tree P_L



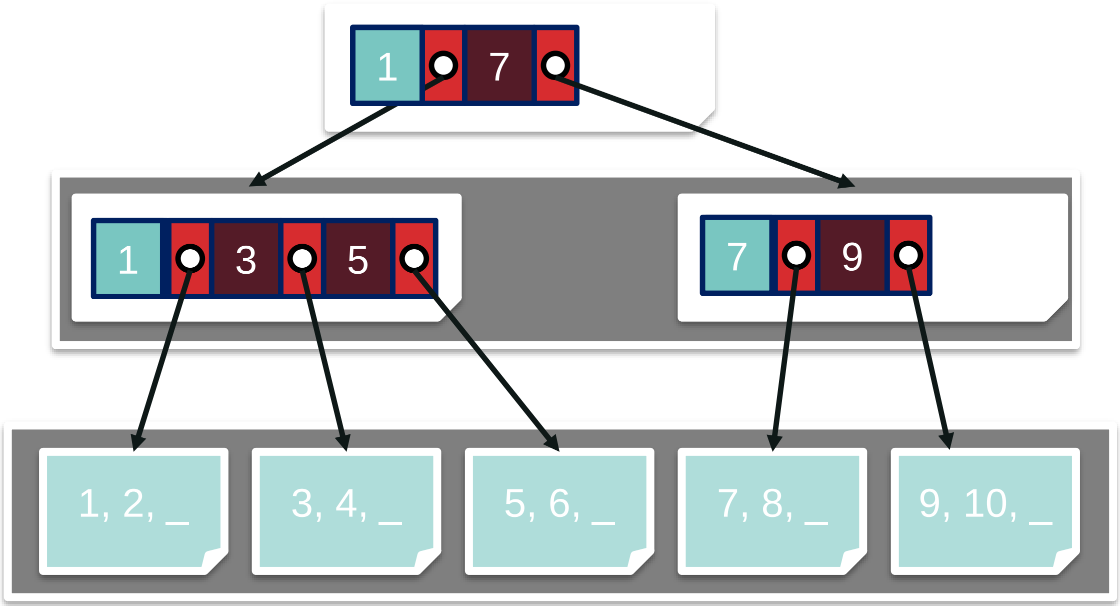
Search a high fan-out search tree

- Searching for **5**?
  - Binary Search each node (page) starting at root
  - Follow pointers to next level of search tree
- Complexity? $O(\log_F(\#Pages))$



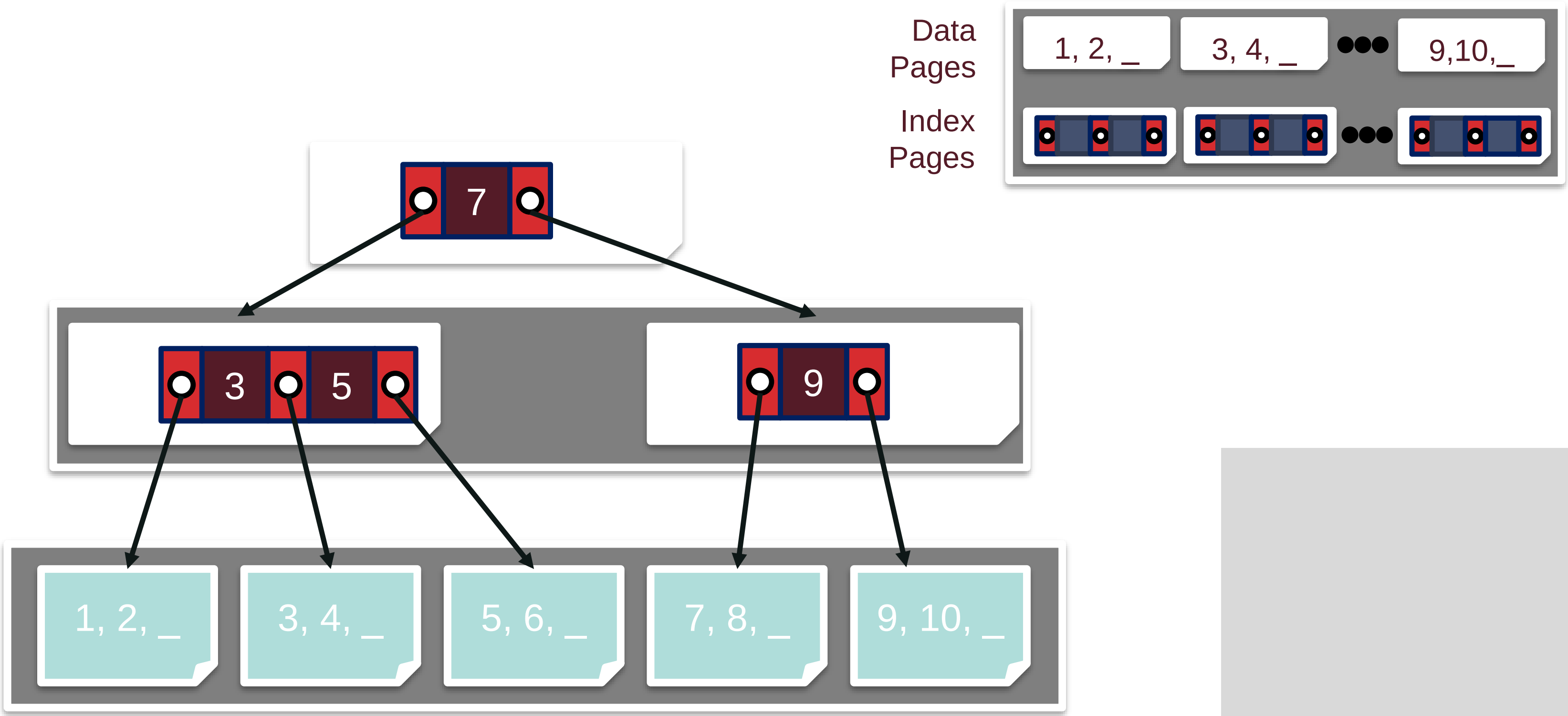
Left Key Optimization?

- Optimization
  - Do we need the left most key?



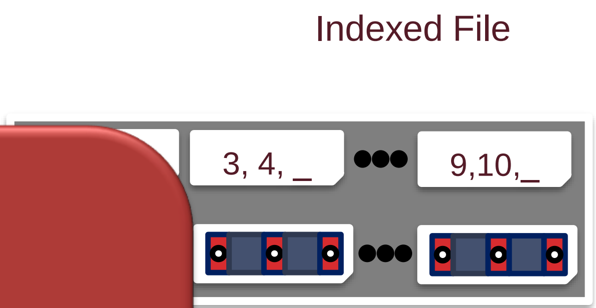
Build a high fan-out search tree

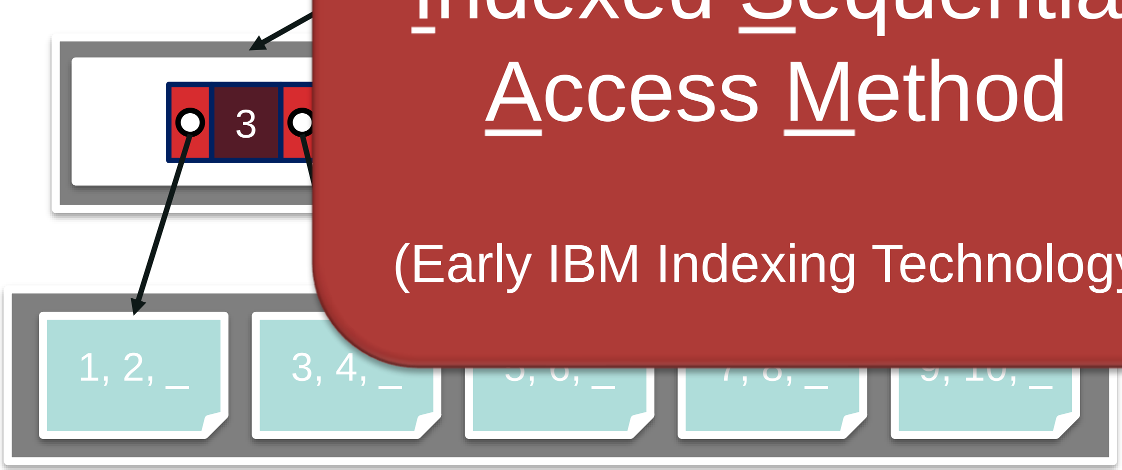
- Disk Layout? All in a single file, Data Pages first.



Status Check

- Some design goals:
  - Fast sequential scan?
  - High Fan-o
  - Support ins



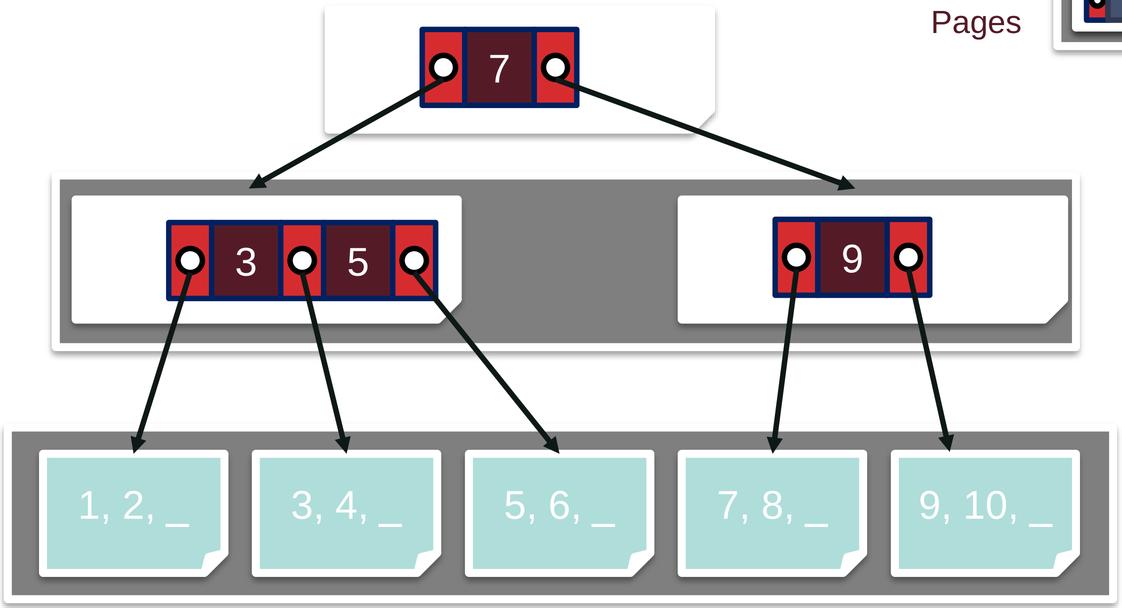


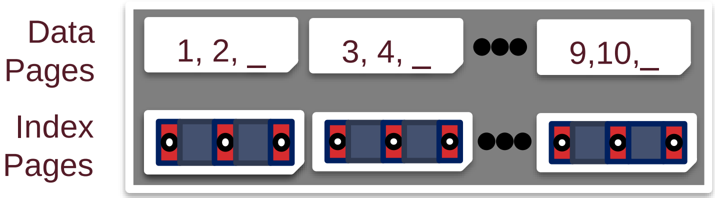
ISAM

Indexed Sequential Access Method

(Early IBM Indexing Technology)

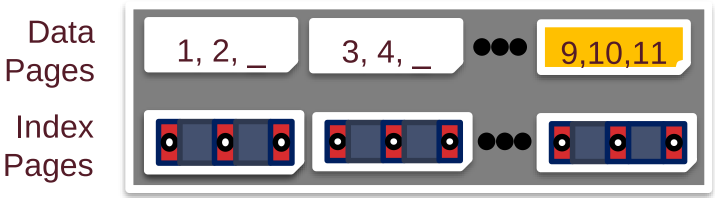
Insert 11, Before

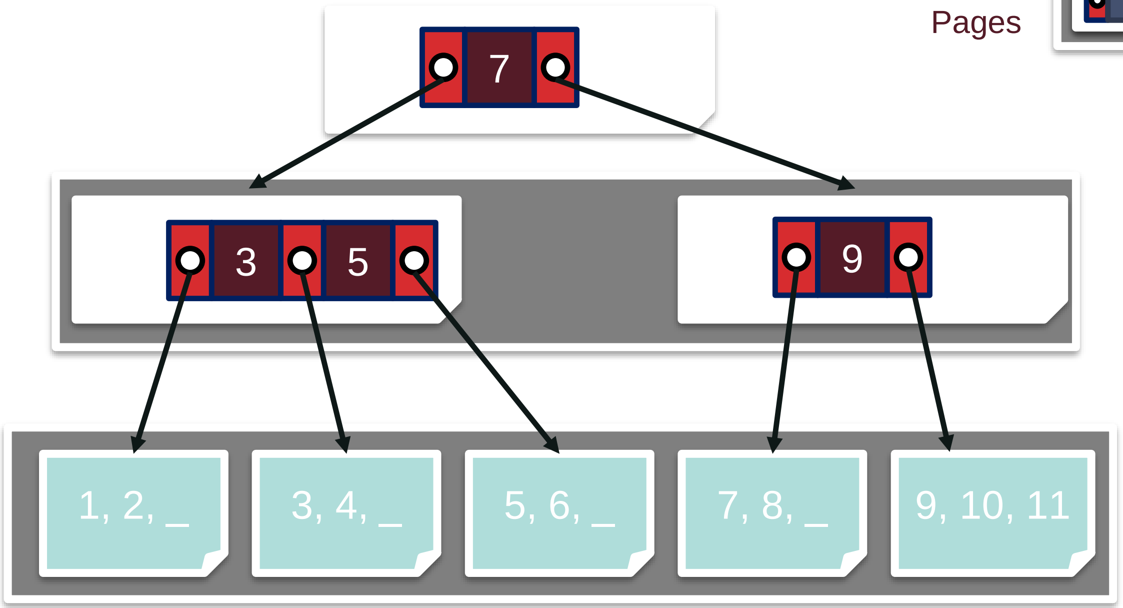




Insert 11, After

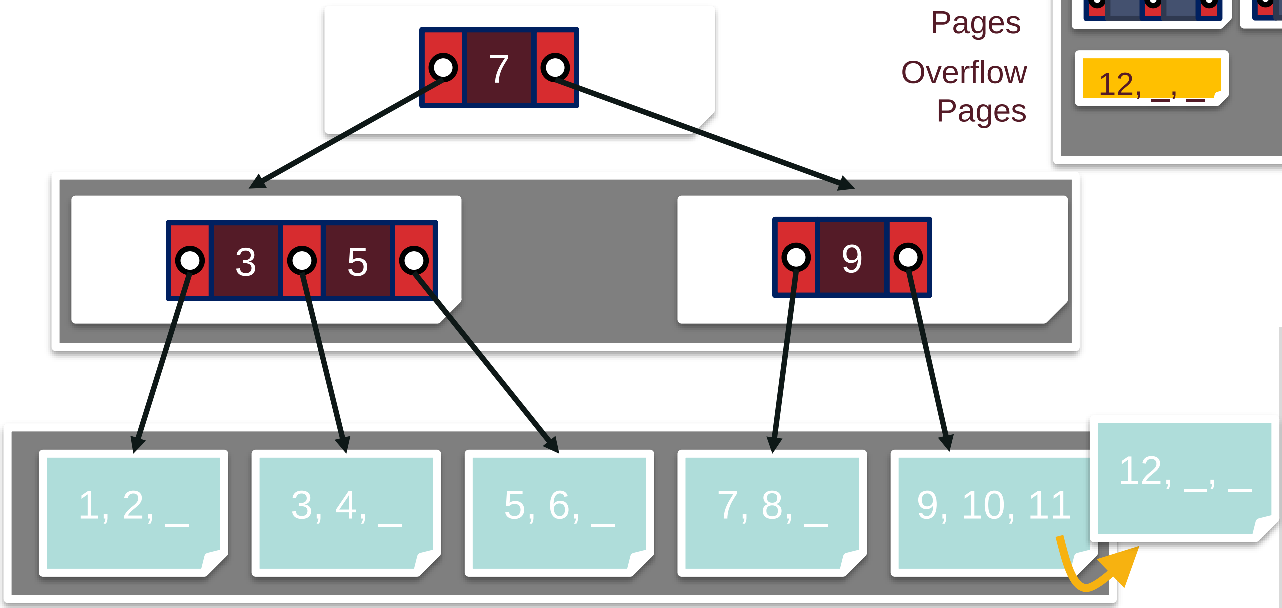
- Find location
- Place in data page
  - Re-sort page …

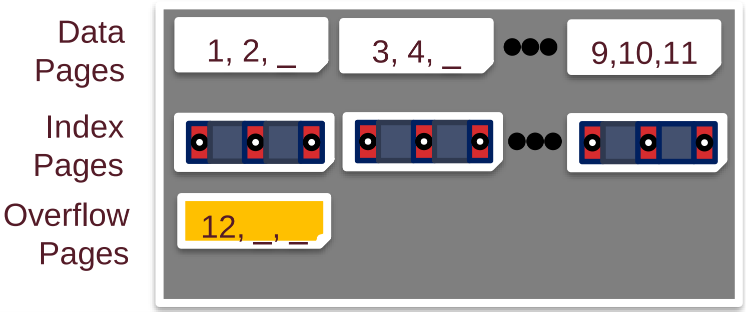




Insert 12?

- Find location
- Place in data page
- Add overflow page if necessary …





Recap: ISAM

- Data entries in sorted heap file
- High fan-out static tree index
- Fast search + good locality
  - Assuming nothing changes
- Insert into overflow pages

A Note of Caution

- ISAM is an old-fashioned idea
  - Introduced by IBM in 1960s
  - B+ trees are usually better, as we'll see
    - Though not always (← we'll come back to this)

- But, it's a good place to start
  - Simpler than B+ tree, many of the same ideas

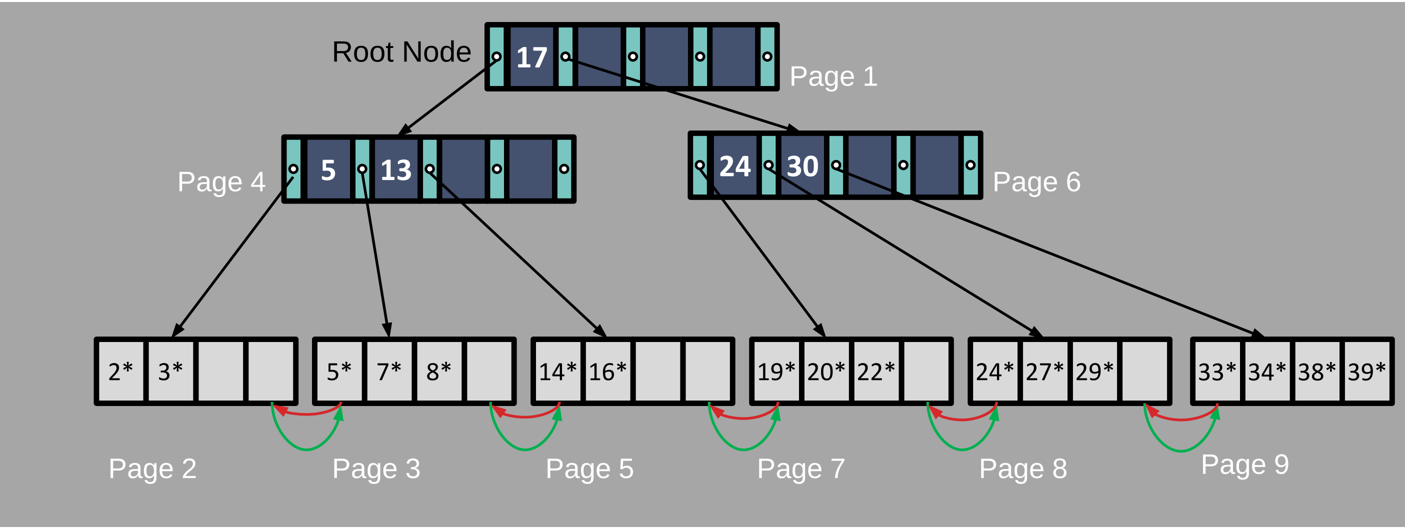
- Upshot
  - Don't brag about ISAM on your resume
  - Do understand ISAM, and tradeoffs with B+ trees

B+-TREE

Enter the B+ Tree

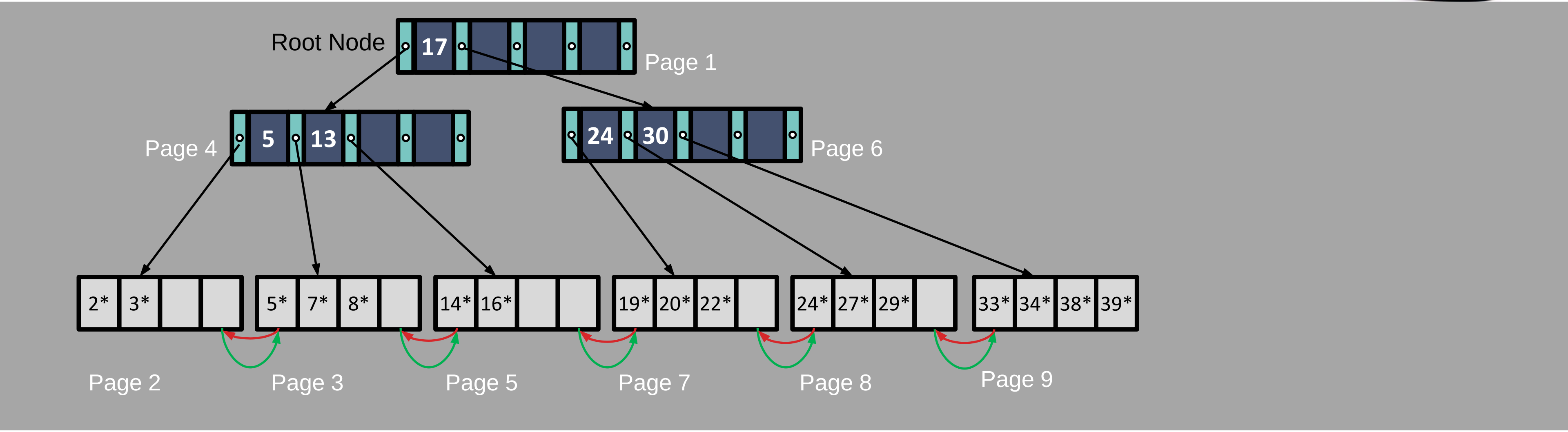
- Similar to ISAM
  - Same interior node structure
    - <Key, Page Ptr> pairs with same key invariant
  - Same search routine as before
- **Dynamic Tree Index**
  - Always Balanced
  - Support efficient insertion & deletion
    - Grows at root not leaves!
- “+”? B-tree that stores data entries in leaves only

Example of a B+ Tree



- Occupancy Invariant
  - Each interior node is at least partially full:
    - **d <= #entries <= 2d**
    - **d: order of the tree (max fan-out = 2d + 1)**
- Data pages at bottom need not be stored in logical order
  - Next and prev pointers

Sanity Check



What is the value of d?

2

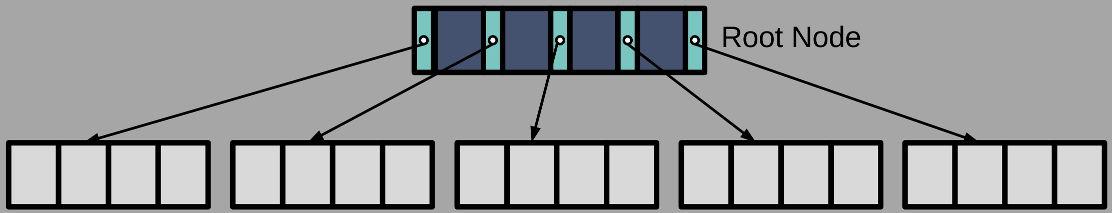
What about the root?

The root is special

Why not in sequential order?

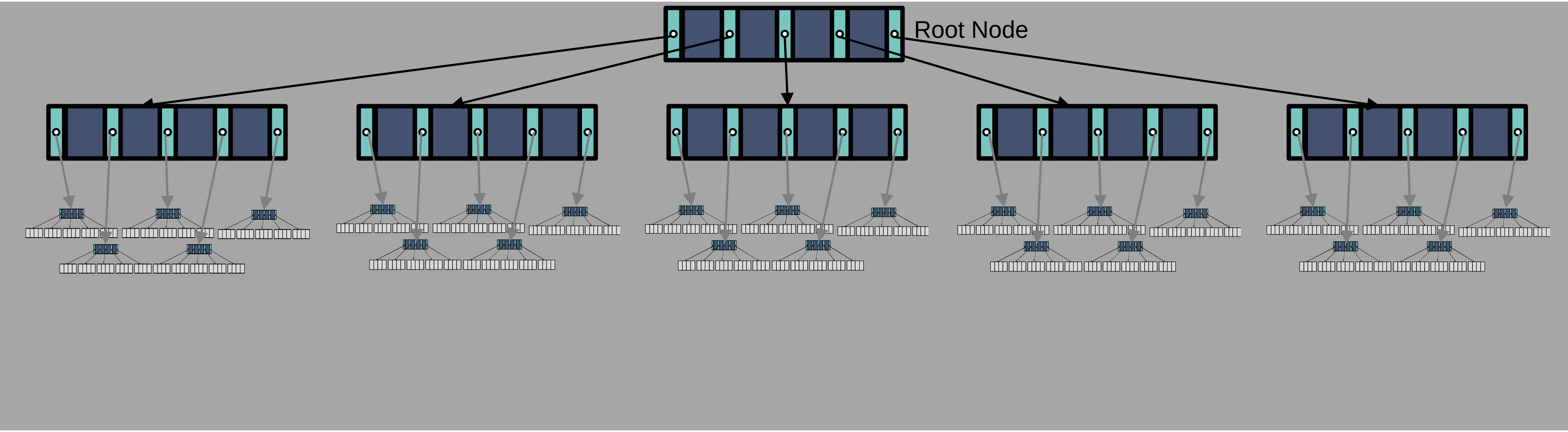
Data pages allocated dynamically

B+ Trees and Scale



- How big is a height 1 B+ tree
  - d = 2 → Fan-out?
  - Fan-out = 2d + 1 = 5
  - **Height 1:** 5 x 4 = 20 Records

B+ Trees and Scale Part 2

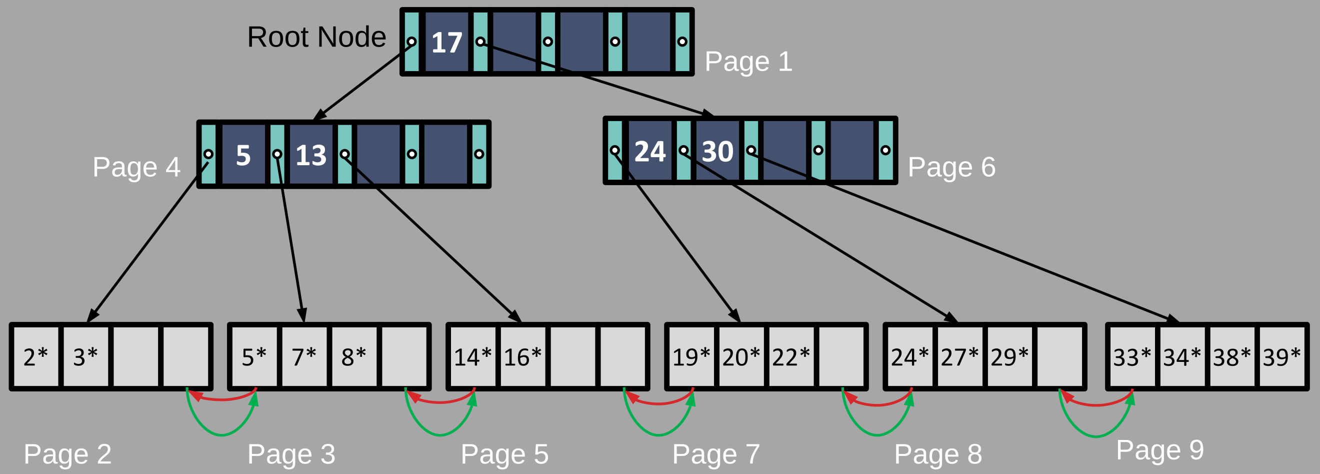


- How big is a height 3 B+ tree
  - d = 2 → Fan-out?
  - Fan-out = 2d + 1 = 5
  - **Height 3:** 5^3 x 4= 500 Records

B+ Trees in Practice

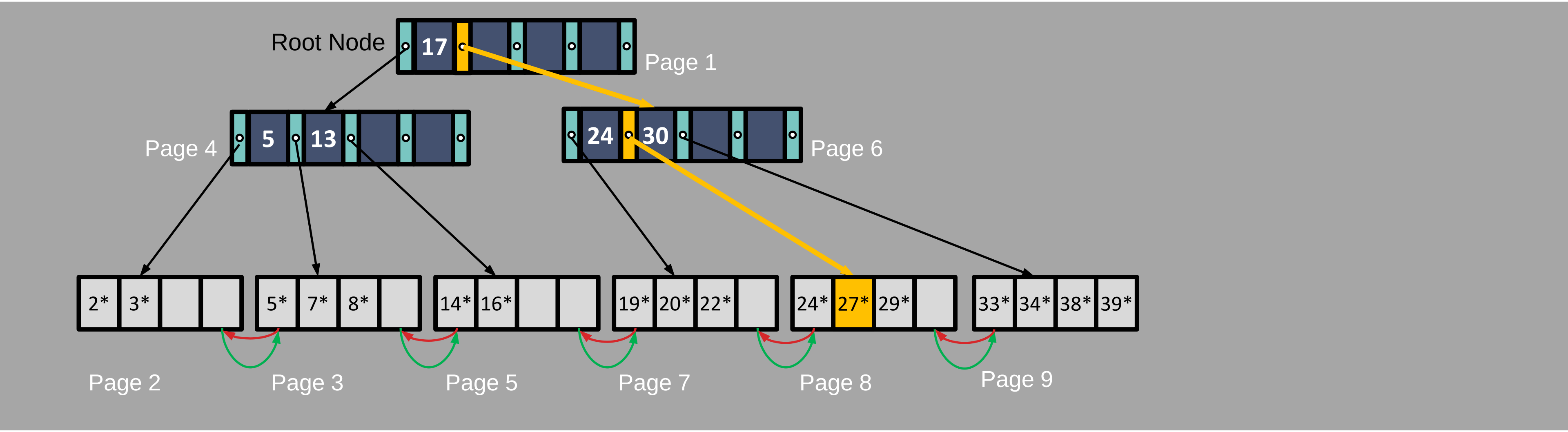
- Typical order: 1600. Typical fill-factor: 67%.
  - average fan-out = 2144
  - (assuming 128 Kbytes pages at 40Bytes per record)
- At typical capacities
  - Height 1: 2144^2 = **4,596,736 records**
  - Height 2: 2144^3 = **9,855,401,984 records**

Searching the B+ Tree



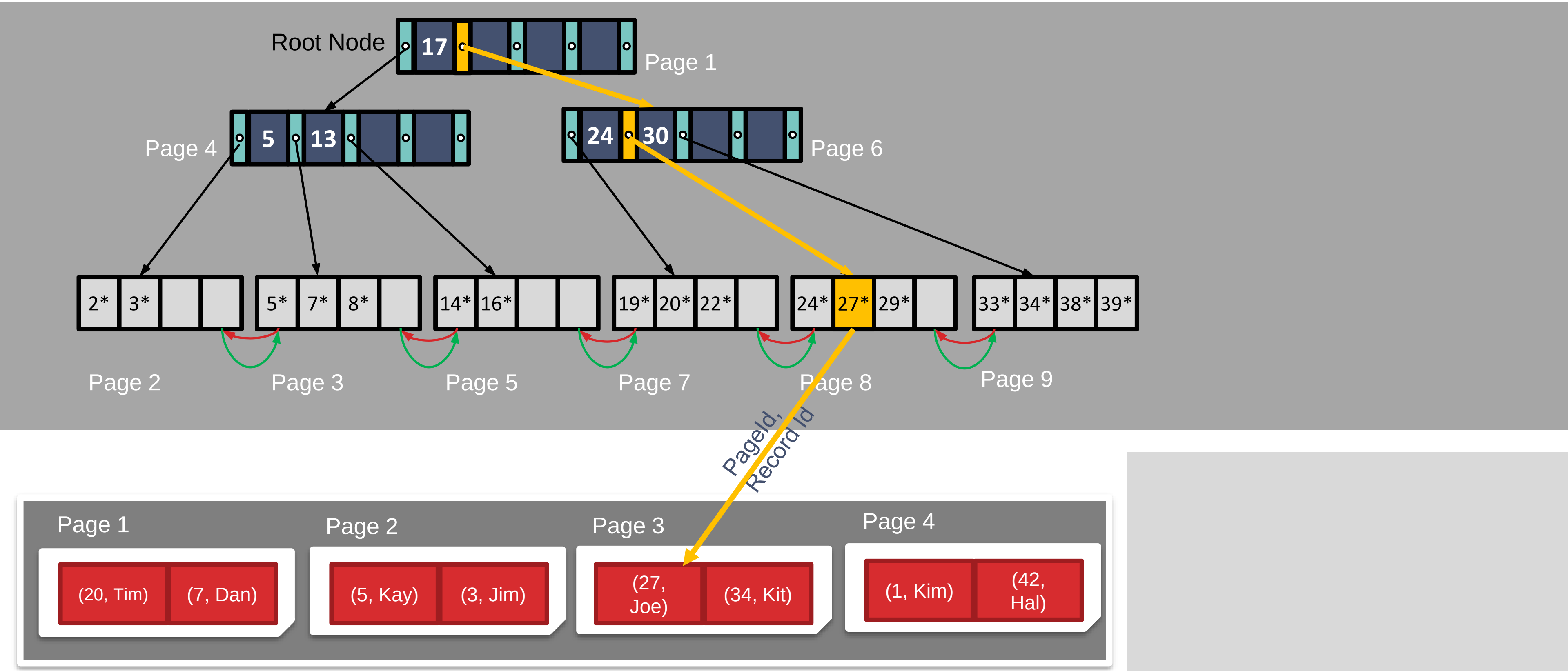
- Same as ISAM
- Find key = 27
  - Find split on each node (Binary Search)
  - Follow pointer to next node

Searching the B+ Tree: Find 27

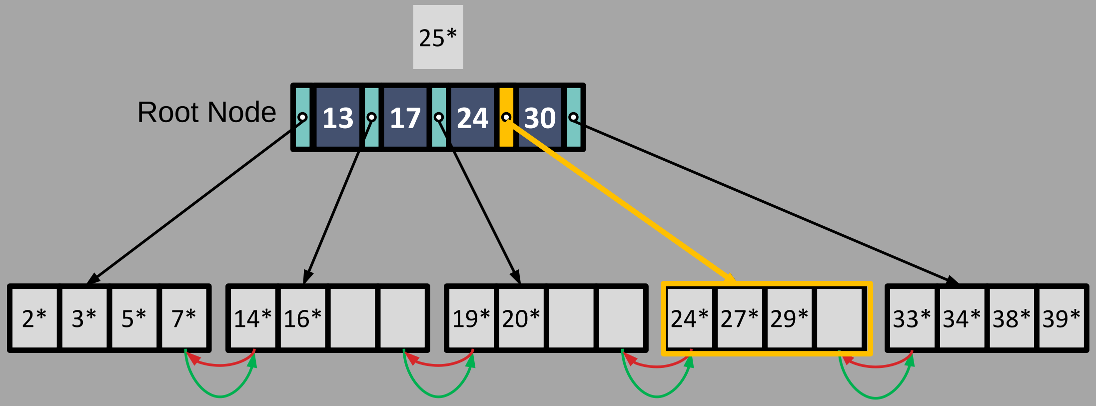


- Same as ISAM
- Find key = 27
  - Find split on each node (Binary Search)
  - Follow pointer to next node

Searching the B+ Tree: Fetch Data

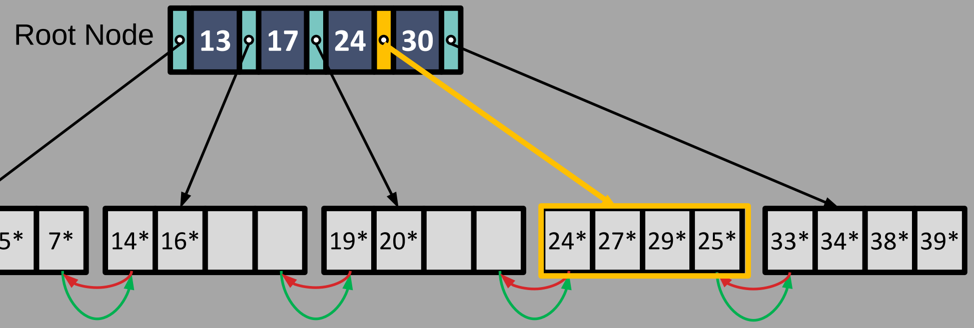


Inserting 25* into a B+ Tree Part 1



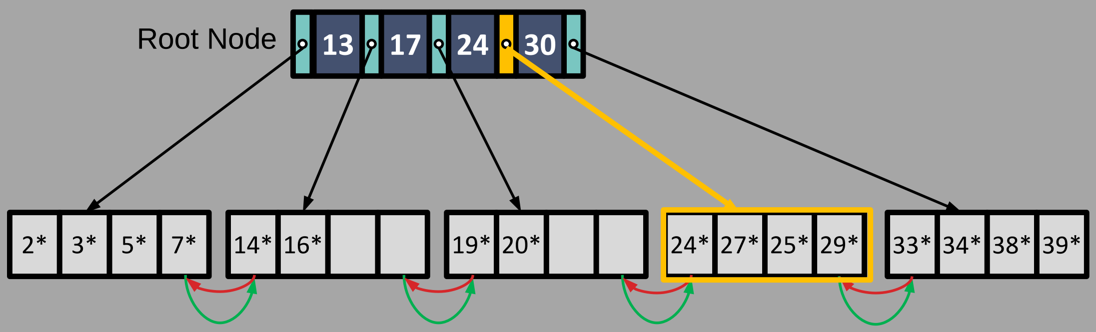
- Find the correct leaf

Inserting 25* into a B+ Tree Part 2



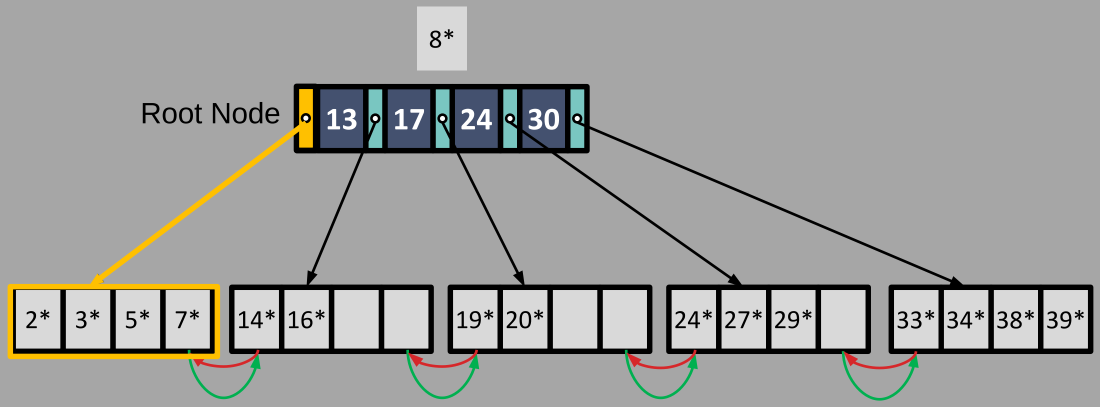
- Find the correct leaf
- If there is room in the leaf just add the entry

Inserting 25* into a B+ Tree Part 3



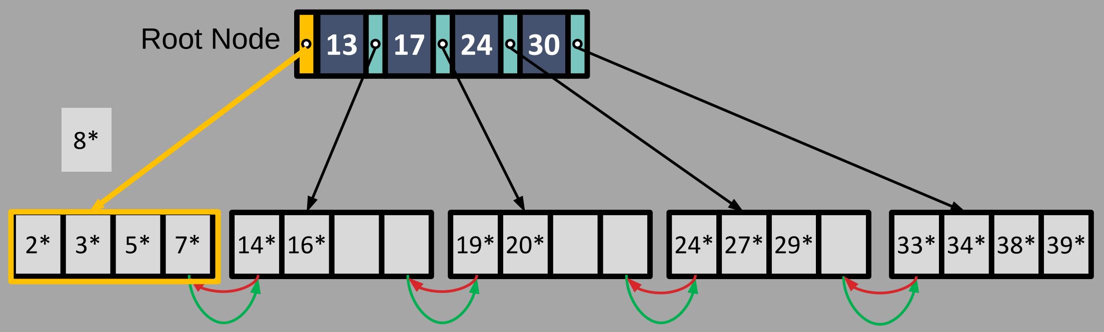
- Find the correct leaf
- If there is room in the leaf just add the entry
  - Sort the leaf page by key

Inserting 8* into a B+ Tree: Find Leaf



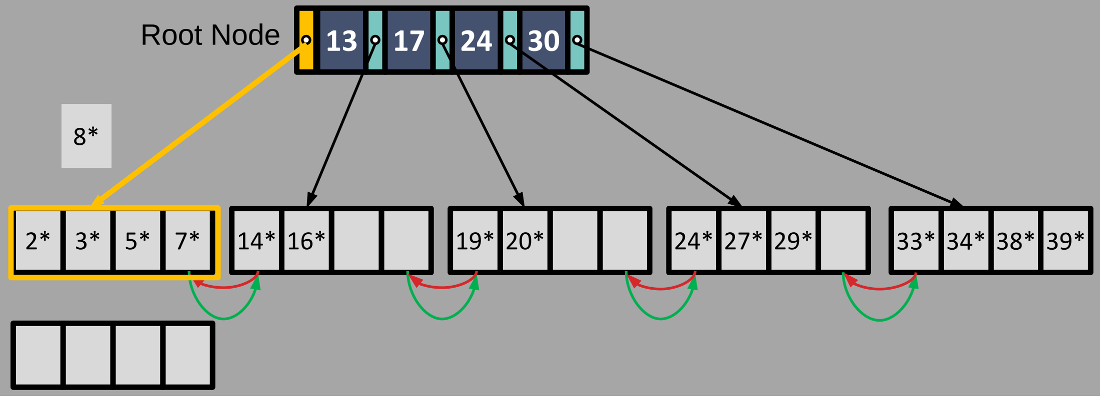
- Find the correct leaf

Inserting 8* into a B+ Tree: Insert



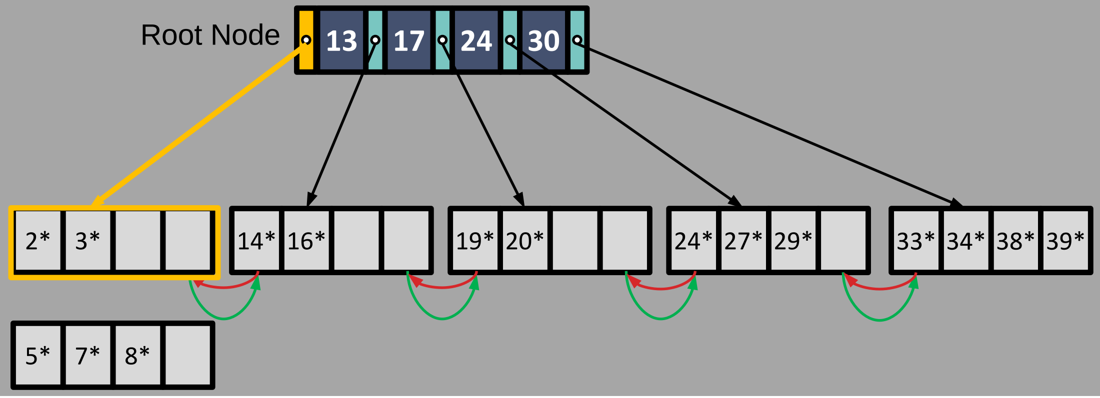
- Find the correct leaf
  - Split leaf if there is not enough room

Inserting 8* into a B+ Tree: Split Leaf



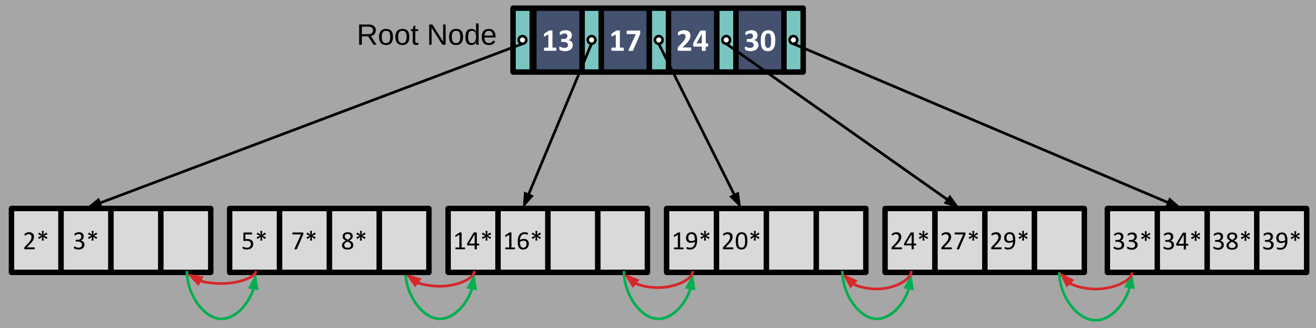
- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly

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



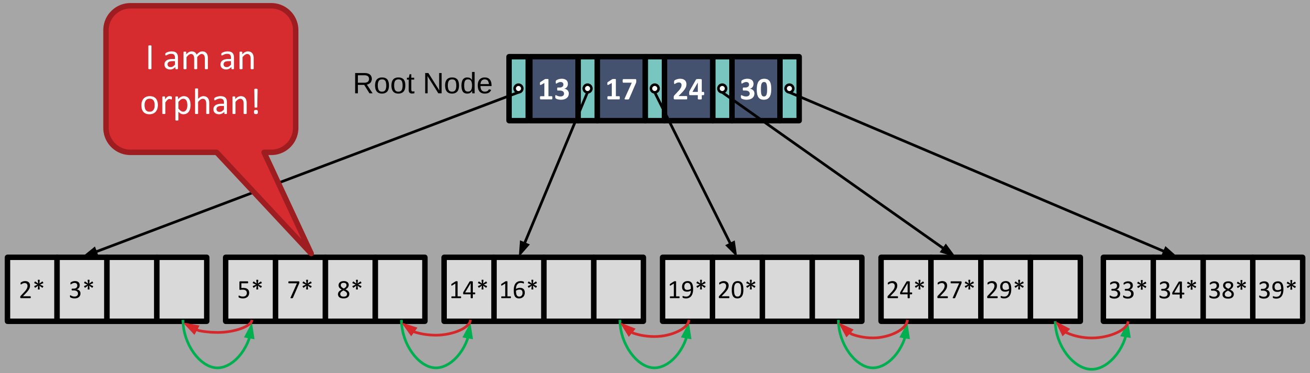
- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - Fix next/prev pointers

Inserting 8* into a B+ Tree: Fix Pointers



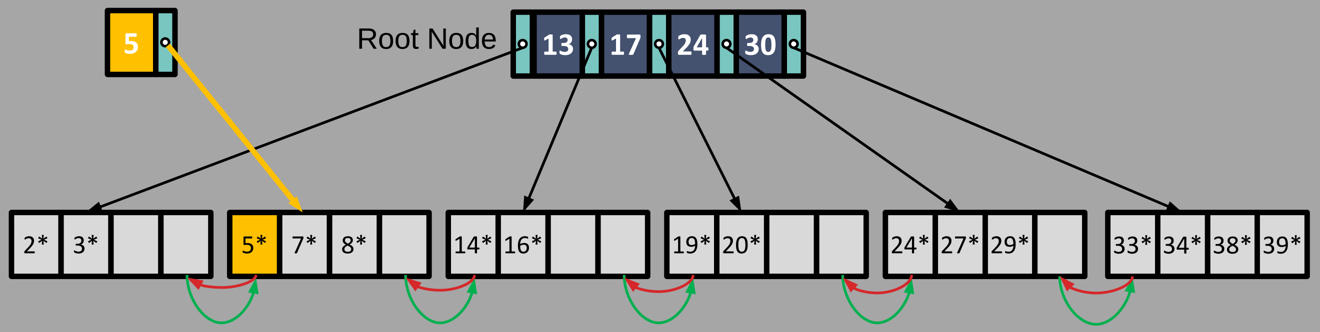
- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - Fix next/prev pointers

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



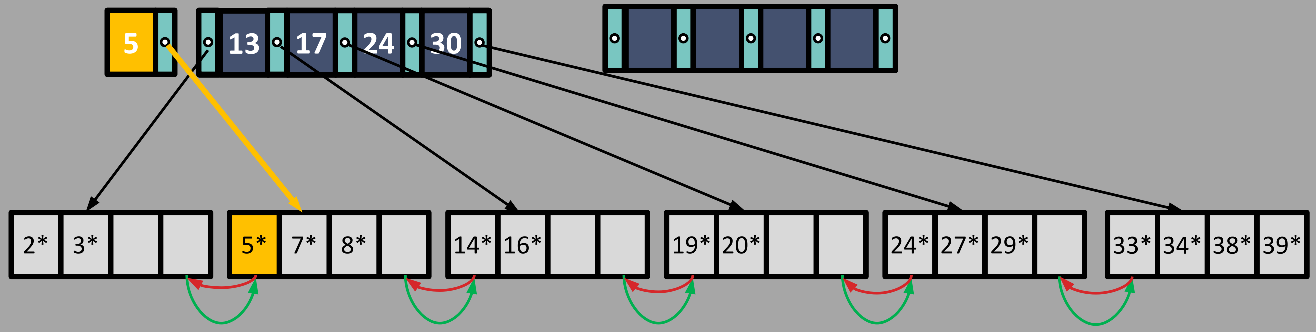
- Something is still wrong!

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



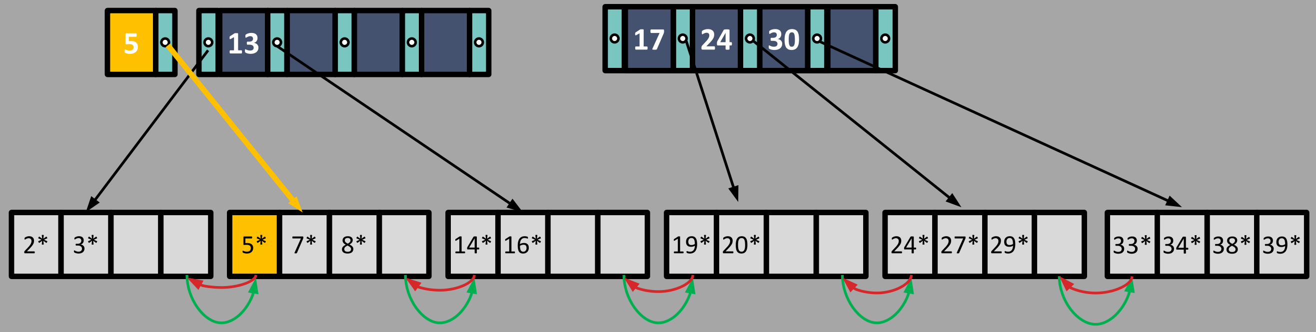
- **Copy up from leaf** the middle key
- No room in parent? Recursively split index nodes

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



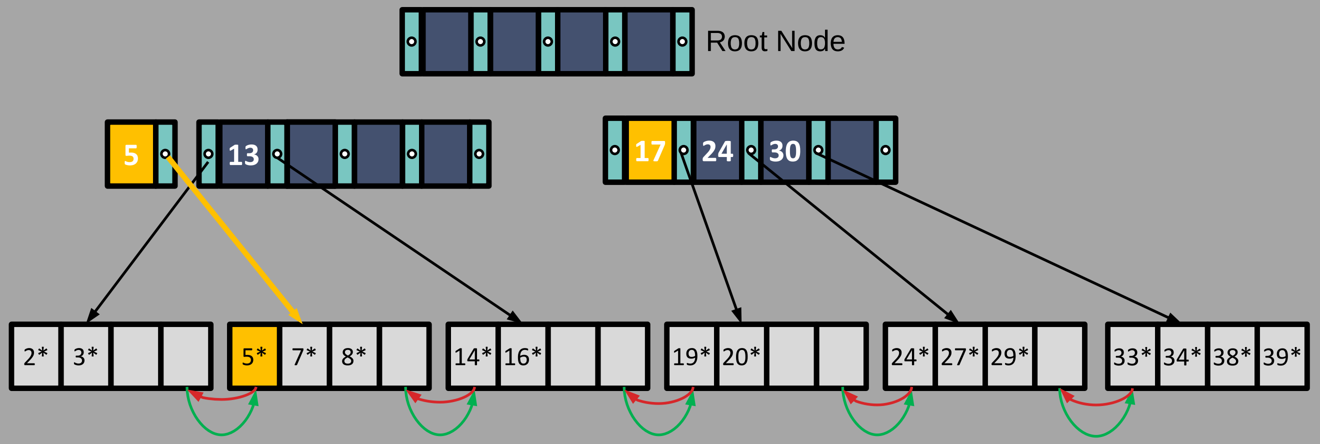
- **Copy up from leaf** the middle key
- No room in parent? Recursively split index nodes
  - Redistribute the rightmost d keys

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



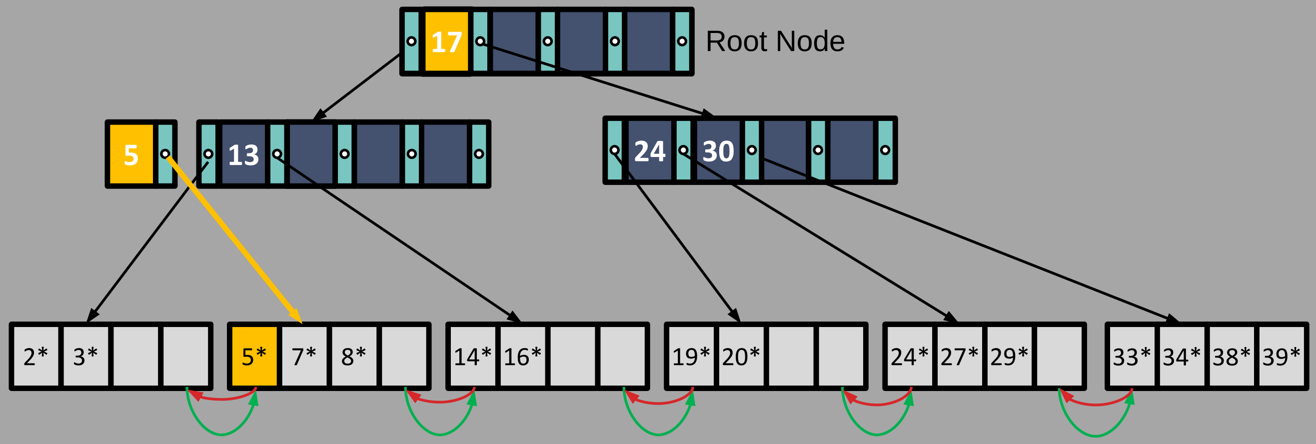
- **Copy up from leaf** the middle key
- No room in parent? Recursively split index nodes
  - Redistribute the rightmost d keys

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



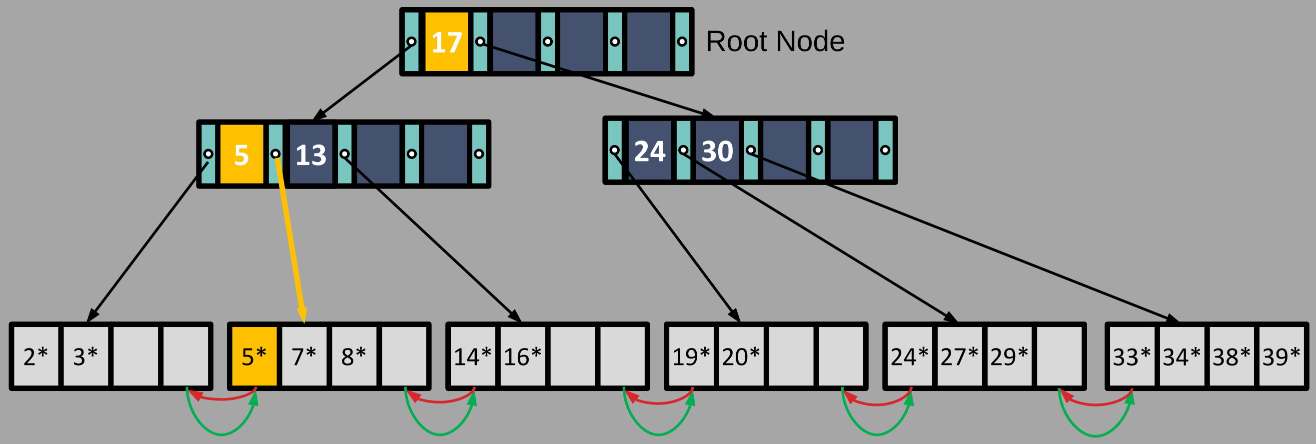
- **Push up from interior node** the middle key
  - Now the last key on left
- No room in parent? Recursively split index nodes
  - Redistribute the rightmost d keys

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



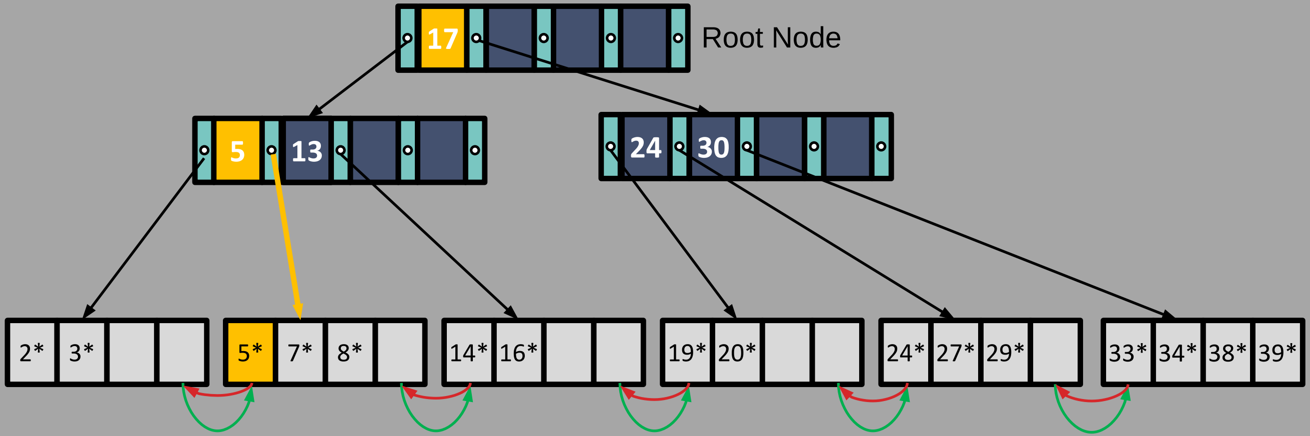
- Recursively split index nodes
  - Redistribute right d keys
  - **Push** up middle key

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



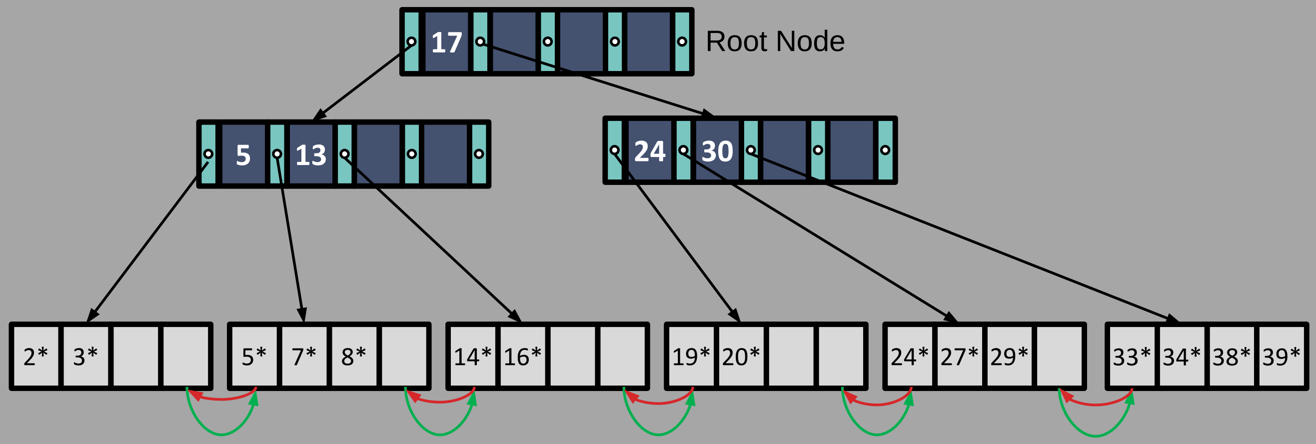
- Recursively split index nodes
  - Redistribute right d keys
  - **Push** up middle key

Copy up vs Push up!



- Notice:
  - The **leaf** entry (5) was **copied** up
  - The **index** entry (17) was **pushed** up

Inserting 8* into a B+ Tree: Final



- Check invariants
- **Key Invariant:**
  - Node[…, (K_L, P_L), …] ➔ K_L<= K for all K in P_L Sub-tree
- **Occupancy Invariant:**
  - d <= # entries <= 2d

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
     - Insert index entry pointing to L2 into parent of L.

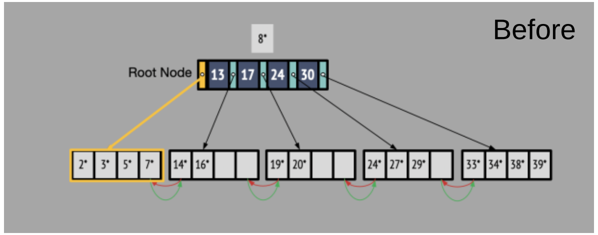
B+ Tree Insert: Algorithm Sketch Part 2

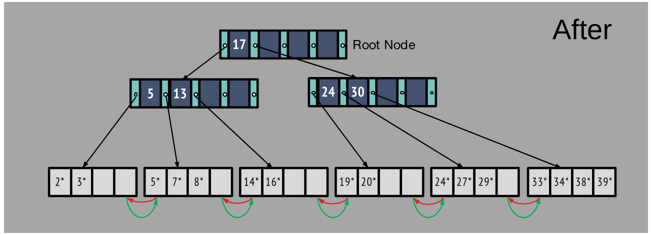
- Step 2 can happen recursively
  - To split index node, redistribute entries evenly, but push up middle key. (Contrast with leaf splits)
- Splits “grow” tree; root split increases height.
  - Tree growth: gets wider or one level taller at top.

Before and After Observations

- Notice that the root was split to increase the height
  - Grow from the root not the leaves
  - All paths from root to leaves are equal lengths

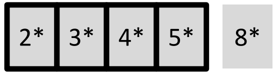
- Does the occupancy invariant hold?
  - Yes! All nodes (except root) are at least half full
  - Proof?



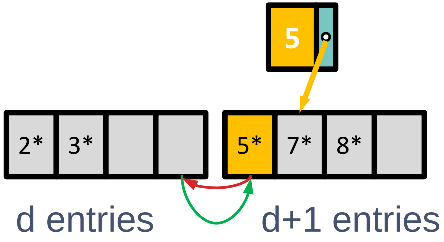


Splitting a Leaf

- Start with full leaf (2d) entries (let d = 2)
  - Add a 2d + 1 entry (8*)

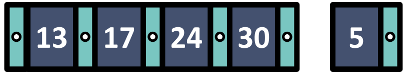


- Split into leaves with (d, d+1) entries
  - Copy key up to parent
- Why copy key and not push key up to parent?



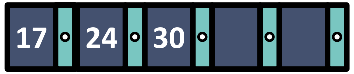
Splitting an Inner Node

- Start with full interior node (2d) entries: (let d = 2)
  - Add a 2d + 1 entry



- Split into nodes with (d, d+1) entries
  - **Push** key up to parent

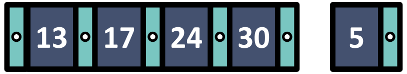
d entries



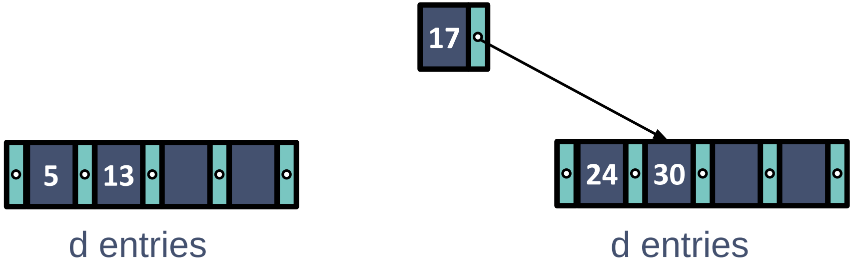
d+1 entries

Splitting an Inner Node Pt 2

- Start with full interior node (2d) entries: (let d = 2)
    - Add a 2d + 1 entry

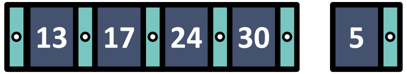


- Split into nodes with (d, d) entries
    - **Push** key up to parent

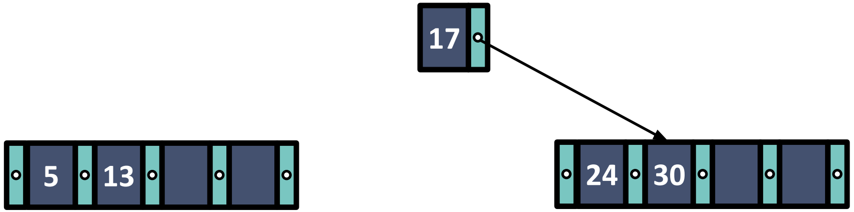


Splitting an Inner Node Pt 3

- Start with full interior node (2d) entries: (let d = 2)
  - Add a 2d + 1 entry



- Split into nodes with (d, d) entries
  - **Push** key up to parent



d entries

d entries

Why push not copy?

- Routing key not needed in child

Occupancy invariant holds after split

Nice Animation Online

- Great animation online of B+ Trees
- One small difference to note
  - Upon deletion of leftmost value in a node, it updates the parent index entry
  - Incurs unnecessary extra writes

B+-TREE DELETION

We will skip deletion

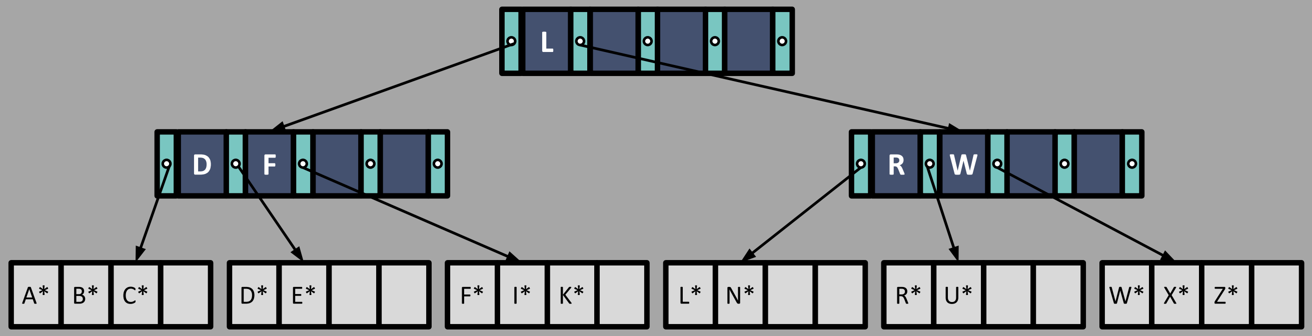
- In practice, occupancy invariant often not enforced
- Just delete leaf entries and leave space
- If new inserts come, great
  - This is common
- If page becomes completely empty, can delete
  - Parent may become underfull
  - That's OK too
- Guarantees still attractive: $\log_F$(max size of tree)

BULK LOADING B+-TREES

Bulk Loading of B+ Tree Part 1

- Suppose we want to build an index on a large table
- Would it be efficient to just call insert repeatedly
  - No … Why not?
  - Random Order: CLZARNDXEKFWIUB. Order 2.
  - Try it: Interactive demo

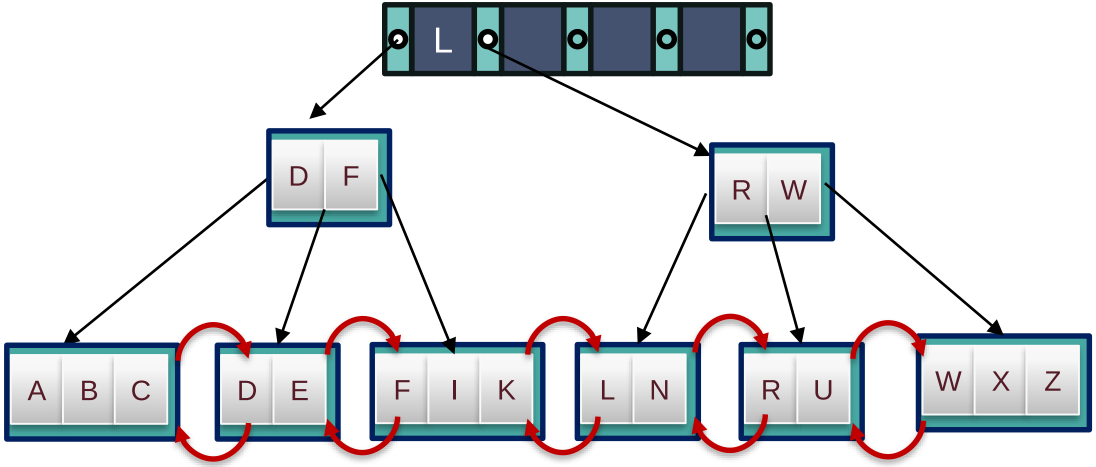
Bulk Loading of B+ Tree Part 2



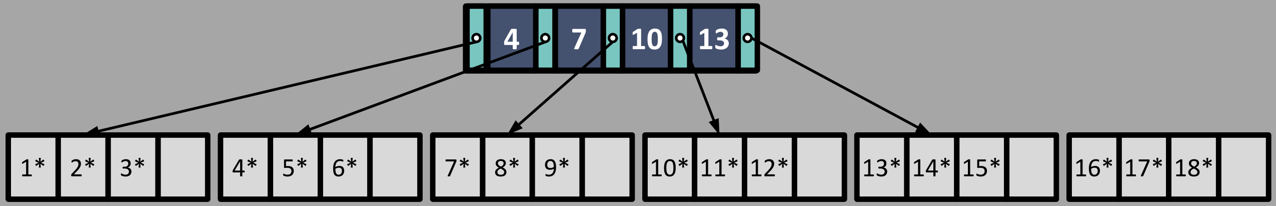
- Constantly need to search from root
- Leaves and internal nodes mostly half-empty
- **Modifying random pages:**
  poor cache efficiency

Bulk Loading of B+ Tree Part 2

- Constantly need to search from leaf
- Leaves and nodes are mostly half full
- **Modifying random pages -> poor cache efficiency**

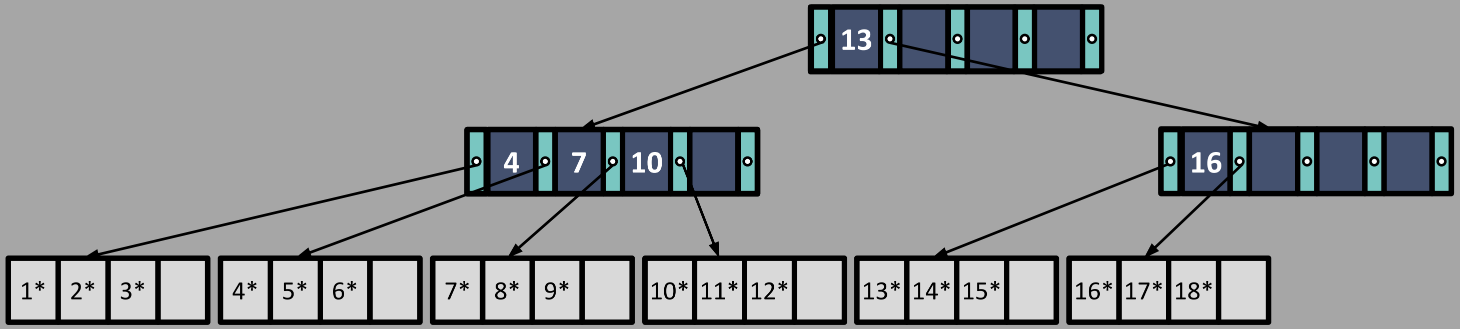


Smarter Bulk Loading a B+ Tree



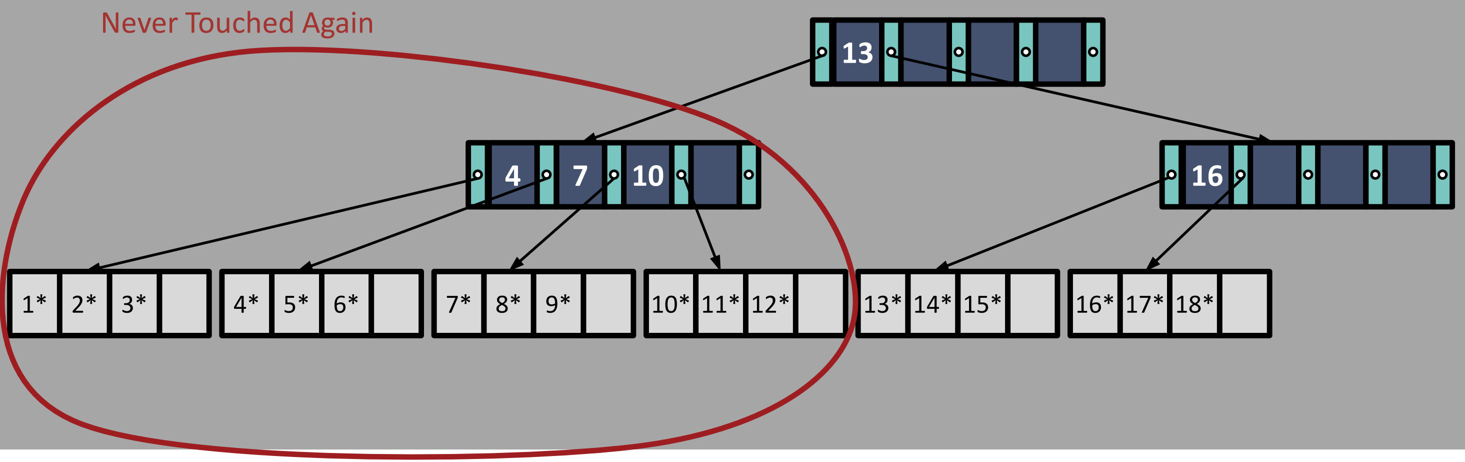
- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
  - We'll learn a good disk-based sort algorithm soon!
- Fill leaf pages to some fill factor (e.g. ¾)
  - Updating parent until full

Smarter Bulk Loading a B+ Tree Part 2



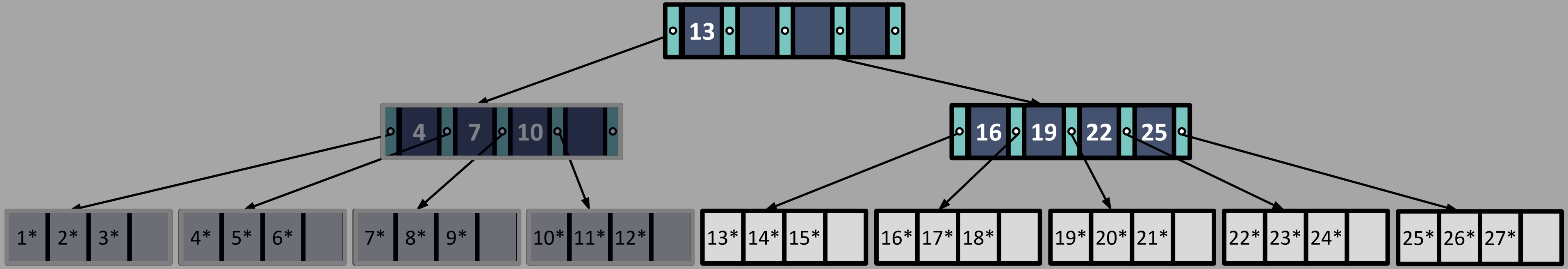
- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
- Fill leaf pages to some fill factor (e.g. ¾)
  - Update parent until full
  - Then split parent and copy to sibling to achieve fill factor

Smarter Bulk Loading a B+ Tree Part 3



- Lower left part of the tree is never touched again
- Occupancy invariant maintained

Smarter Bulk Loading a B+ Tree Part 4



- Sort the input records by key:
  - 1*, 2*, 3*, 4*, …
- Fill leaf pages to some fill factor (e.g. ¾)
  - Update parent until full
  - Then split parent

Summary of Bulk Loading

- Option 1: Multiple inserts
  - **Slow**
  - Does not give sequential storage of leaves

- Option 2: Bulk Loading
  - Fewer I/Os during build. (Why?)
  - Leaves will be stored sequentially (and linked, of course)
  - Can control “fill factor” on pages.

Summary

- 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.
  - Typically, 67% occupancy on average
  - Usually preferable to ISAM; adjusts to growth gracefully.

Summary Cont.

- Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set.
- B+ tree widely used because of its versatility
  - One of the most optimized components of a DBMS.
  - Concurrent Updates
  - In-memory efficiency

Graphic Components