CS 4604: Introduction to Database Management Systems

Storing Data and Indexes

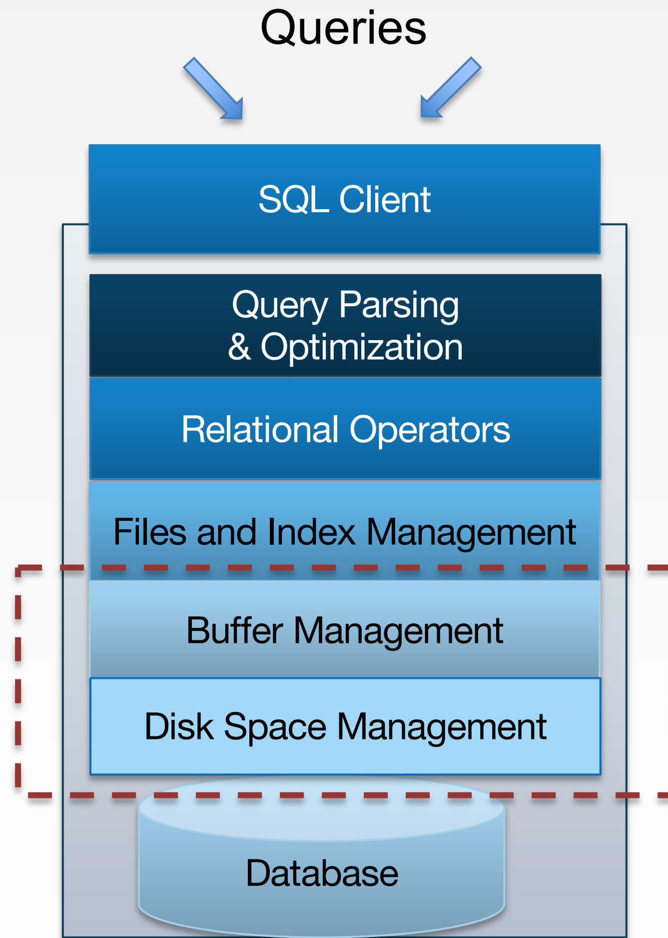
Virginia Tech CS 4604 Sprint 2021

Instructor: Yinlin Chen

Today’s Topics

- Storing data
- Indexes

DBMS Layers



File of Records

- Collections of **data records** are stored in **pages**, and collections of pages make up a **file**
- Think of “file” as an abstraction referring to a collection of pages, not necessarily a single physical construct
- Generally, a record is row-oriented, storing all the attributes from one row in a table
- Records might be of other types
  - Index data entries
  - Catalog metadata
  - Etc.

Page

- A page is the basic unit of storage.
- All pages are same size, usually 4-16KB
- Page stores related records (e.g., tuples from one particular table)
- Typical implementation is for pages to have a directory to identify the offset of records.
  - Variable record size
  - Allows for stable Record Id (pointer)
  - Record Id = <page#, slot#>
  - Very useful with indexing

Disks and Files

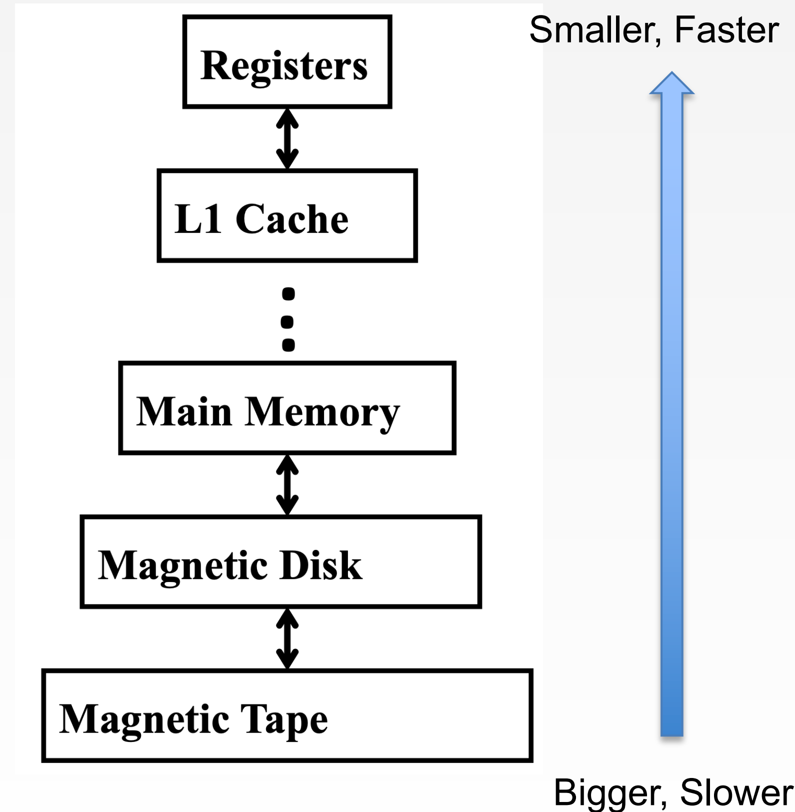
- DBMS stores information on disks
  – But disks are relatively slow
- Major implications for DBMS design
  – READ: disk -> main memory (RAM)
  – WRITE: RAM -> disk (reverse)
  – Both are high-cost operations, relative to in-memory operations, so must be planned carefully

In-memory Database

- Redis, Memcached, etc.
- Amazon ElastiCache for Redis can scale up to 250 nodes and enables a maximum in-memory data size of 170.6 TB
  - 42597.10 * 12 = 511165.2 per year
- Databases today in the 10-150+ TB range
- Amazon sold $10.4 billion worth of goods on Prime Day (2 days!)
- How about petabyte database?

The Storage Hierarchy

- Main memory (RAM) for currently used data
- Disk for the main database (secondary storage)
- Tapes for archiving older versions of the data (tertiary storage)

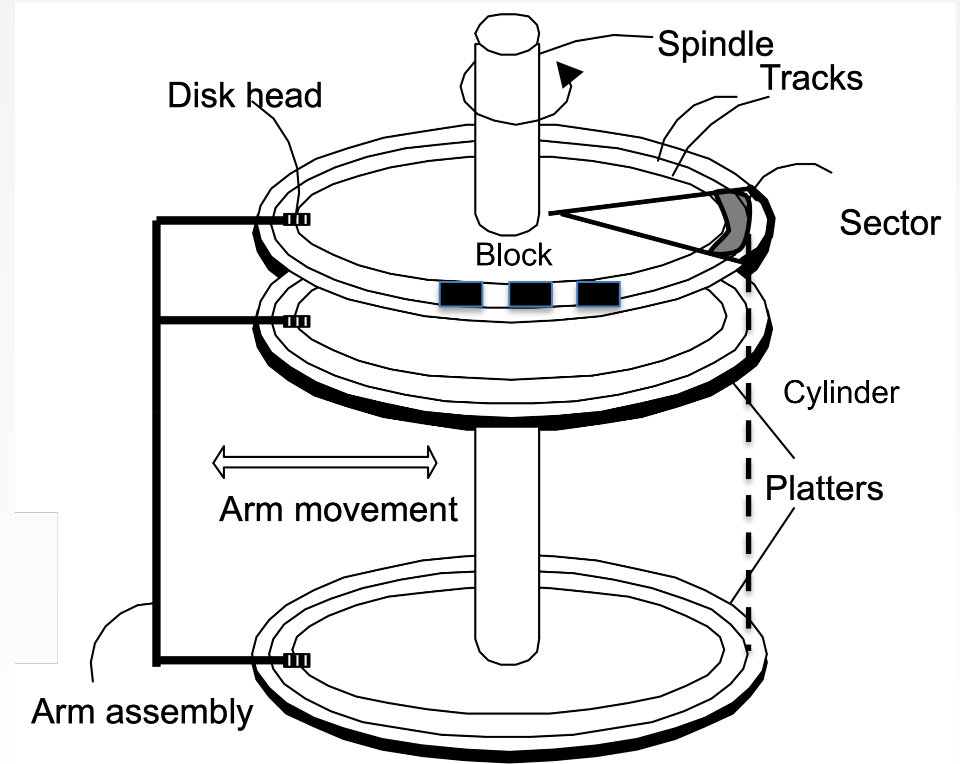


Disks

- Secondary storage device of choice
- Main advantage over tapes: **random access** vs. **sequential**
- Data is stored and retrieved in units called disk **blocks** or **pages**
- Unlike RAM, time to retrieve a disk page varies depending upon **location** on disk
  - relative placement of pages on disk is important!

Anatomy of a Disk

- Sector
- Track
- Cylinder
- Platter
- Block size = multiple of sector size (which is fixed)



Accessing a Disk Page

- Time to access (read/write) a disk block:
  - seek time: moving arms to position disk head on track (about 1 to 20msec)
  - rotational delay: waiting for block to rotate under head (0 to 10msec)
  - transfer time: actually moving data to/from disk surface (< 1msec per 4KB page)
- Key to lower I/O cost: reduce seek/rotation delays
- For shared disks, much time spent waiting in queue for access to arm/controller

Arranging Pages on Disk

- “Next” block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder
- Accessing ‘next’ block is cheap
- A useful optimization: Prefetching
  - Prefetch pages

Memory and Disk

- A unit of (disk) I/O refers to this movement of a page between secondary storage and the Buffer Pool (memory)
- Memory access much faster than disk I/O (~ 1000x)
- “Sequential” I/O faster than “random” I/O (~ 10x)
- Sorting
  - Pack info in blocks
  - Try to fetch nearby blocks (sequentially)
  - Index: used to sort, or group, the rows in the table

Buffer Manager

- Managing pages in memory
- Receives page requests from the file and index manager
- Communicates with the disk space manager to perform the required disk operations:
  - When pages are evicted from memory
  - When new pages are read into memory
- Higher levels of the DBMS need only ask the Buffer Manager for a page
  - Regardless of whether the page is on disk or in Buffer Pool

Buffer Management Levels of Abstraction

Files and Index Management

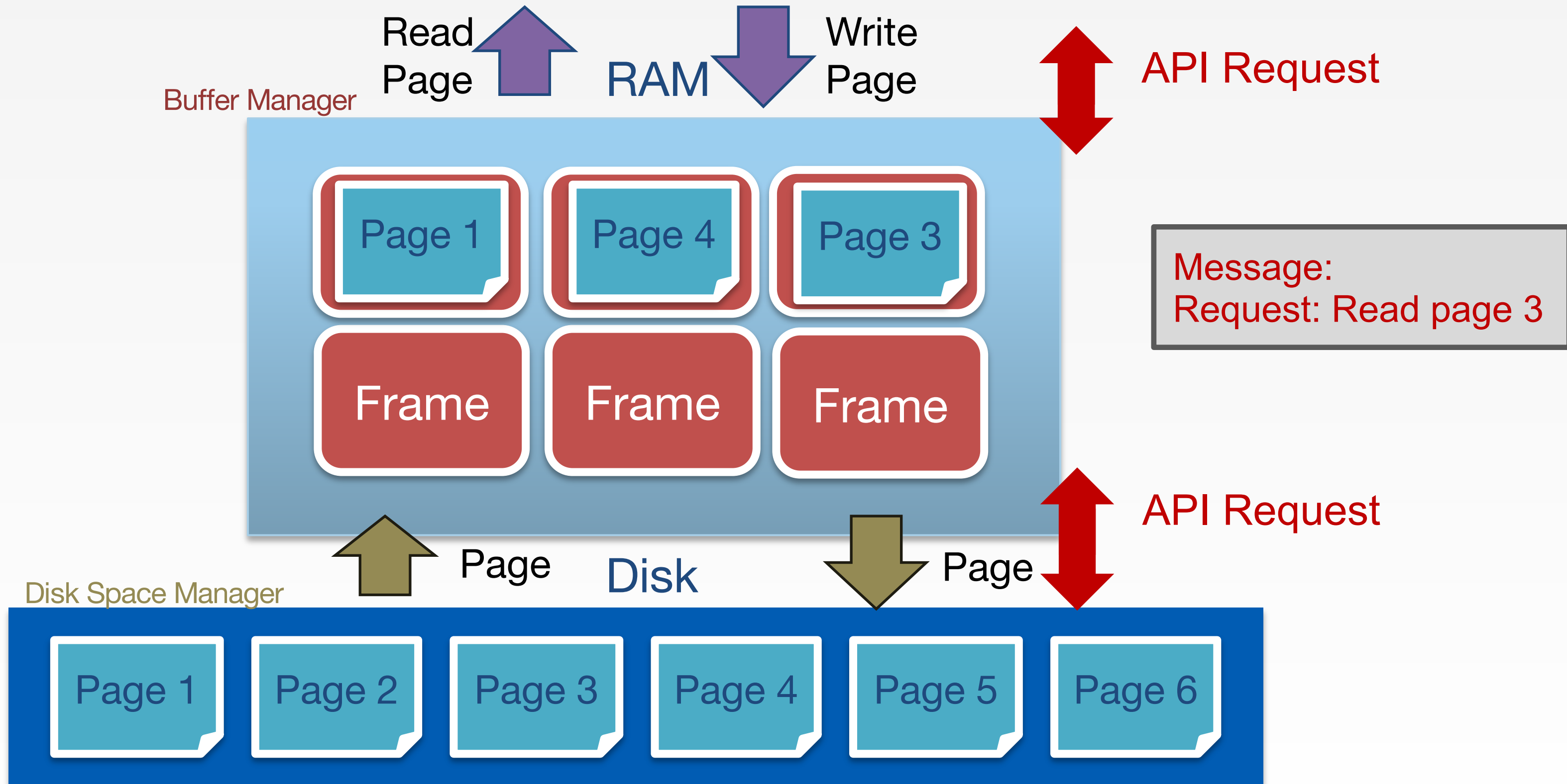
RAM

Buffer Management

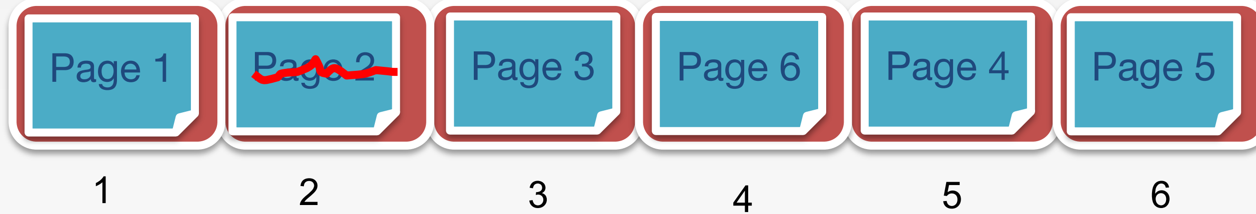
Disk

Disk Space Management

Buffer Management



Buffer Pool



FrameId	PageId	Dirty Bit	Pin Count
1	1	0	0
2	2	1	1
3	3	0	0
4	6	0	2
5	4	0	0
6	5	0	0

- **Frame ID** is uniquely associated with a memory address
- **Page ID** for determining which page a frame currently contains
- **Dirty Bit** is set if the page was modified
- **Pin Count** is incremented while a thread needs to access the page

Page Replacement

1. If requested page is not in pool:
    a. Choose an **un-pinned** (pin_count = 0) frame for replacement, using a **replacement policy**.
    b. If the "dirty bit" for the replacement frame is on, write current page to disk, mark "clean"
    c. Read requested page into the replacement frame
2. Pin the page and return the (main memory) address to the requester

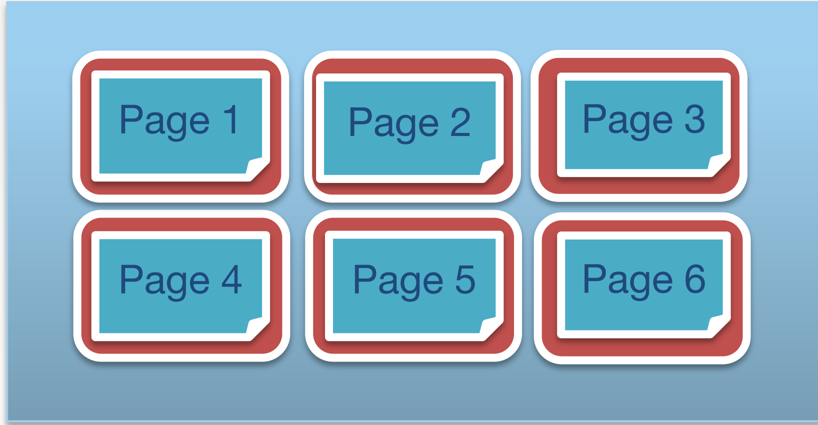
- If requests can be predicted (e.g., sequential scans) pages can be pre-fetched (several pages at a time)

Page Replacement Policy

- Page is chosen for replacement by a replacement policy:
  - Least-recently-used (LRU), Clock
  - Most-recently-used (MRU)

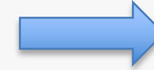
- Policy can have big impact on #I/Os
  - Depends on the access pattern.

LRU vs MRU



LRU
- Cache Hits: 0
- Attempts 6

MRU
- Cache Hits: 0
- Attempts 6



LRU
- Cache Hits: 0
- Attempts 14

MRU
- Cache Hits: 6
- Attempts 14

Disk Space Manager

Summary

- LRU wins for random access
- MRU wins for repeated sequential
- Attempts to minimize "cache misses"
  - By replacing pages unlikely to be referenced
  - By prefetching pages likely to be referenced
- Hybrids are not uncommon in modern DBMSs
- Machine / Deep learning on replacement policy

DB File Structures

- Unordered Heap Files
  - Records placed arbitrarily across pages
- Clustered Heap Files
  - Records and pages are grouped
- Sorted Files
  - Pages and records are in sorted order
- Index Files
  - B+ Trees, Linear Hashing, …
  - May contain records or point to records in other files

Indexes

- An index is a data structure that helps speed up reads on a specific key

```
CREATE INDEX StudentsInd ON Students(ID);

CREATE INDEX CoursesInd ON
    Courses(Number, DeptName);
```

Types of Indexes

- **Primary:** index that includes the primary key
  - Used to enforce constraints
- **Secondary:** index on non-key attribute
- **Clustering:** order of the rows in the data pages correspond to the order of the rows in the index
  - Only one clustered index can exist in a given table
  - Useful for range predicates
- **Non-clustering:** physical order not the same as index order

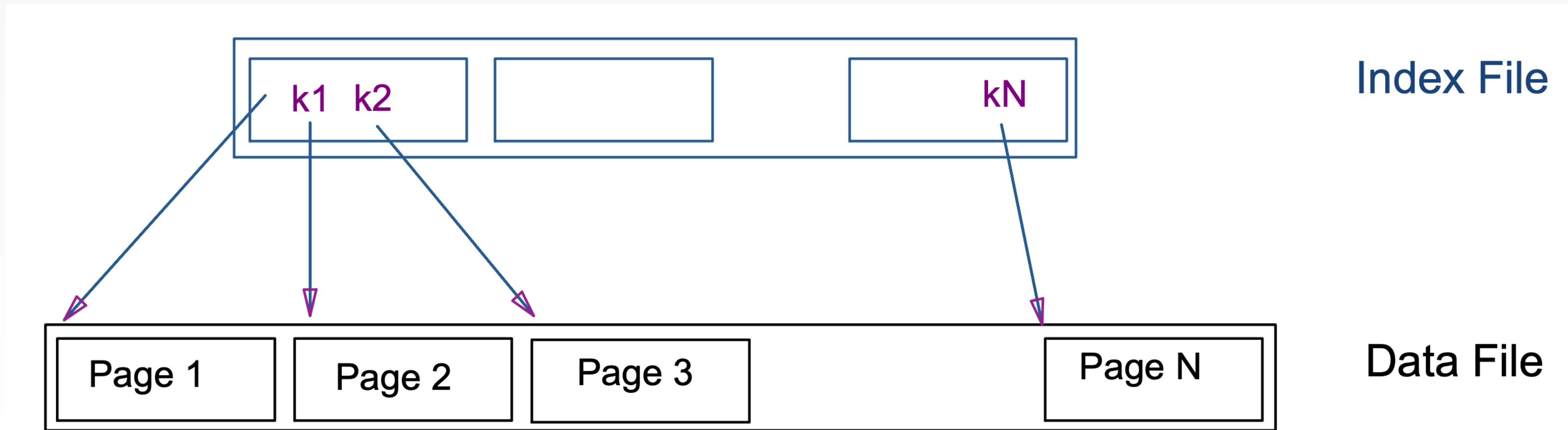
Using Indexes: Equality Searches

- Given a value *v*, the index takes us to only those tuples that have *v* in the attribute(s) of the index.
- E.g. (use CourseInd index)

```
SELECT Enrollment FROM Courses
    WHERE Number = "4604" and
          DeptName = "CS"
```

Using Indexes (2): Range Searches

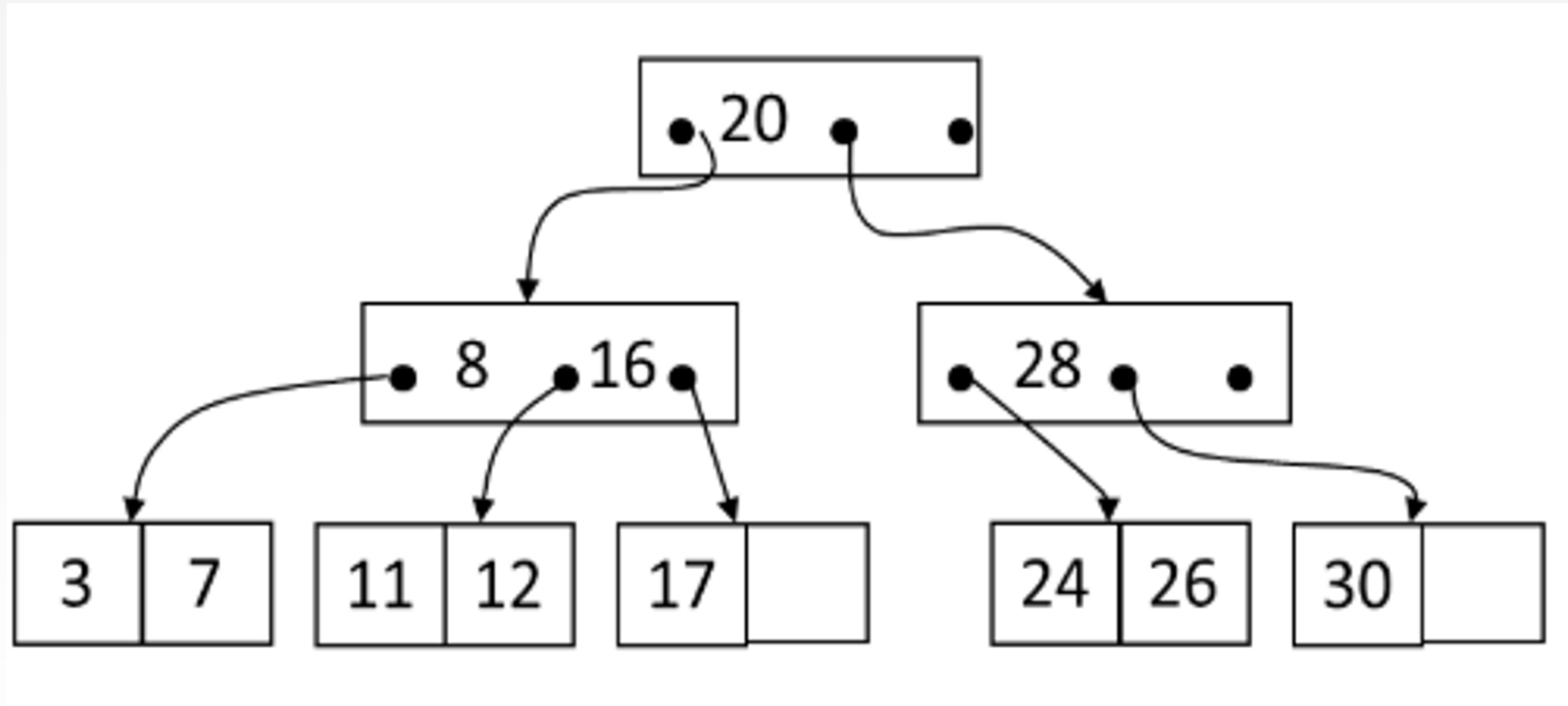
- *Find all students with gpa > 3.0*
- May be slow, even on sorted file
- Solution: Create an “index” file



B-Tree Index

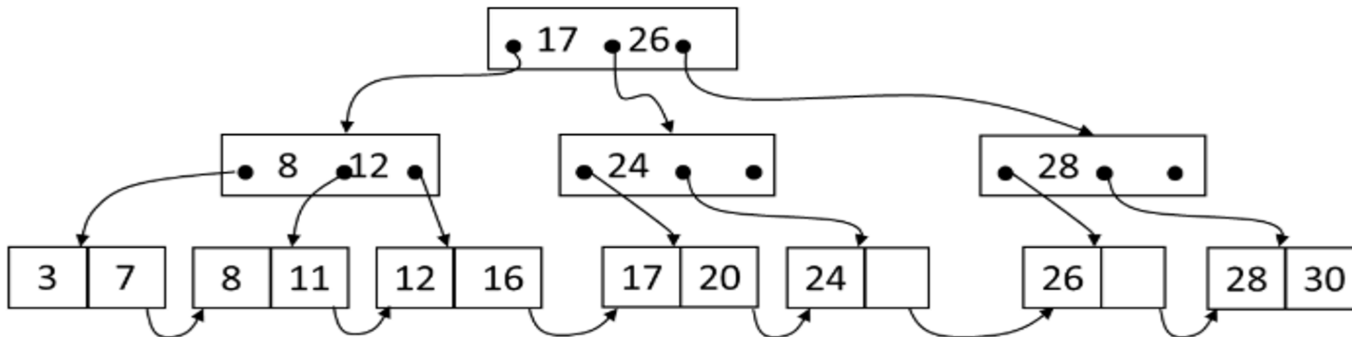
- B-Trees is the most successful family of index schemes (B-trees, B+-trees, B*-trees) used in DBMSs. It’s what you’ll get with a basic create index statement
- Can be used for primary/secondary, clustering/non-clustering index.
- B-Tree indexes are balanced, meaning all the leaf nodes have the same path length from the root node

B-Tree Index

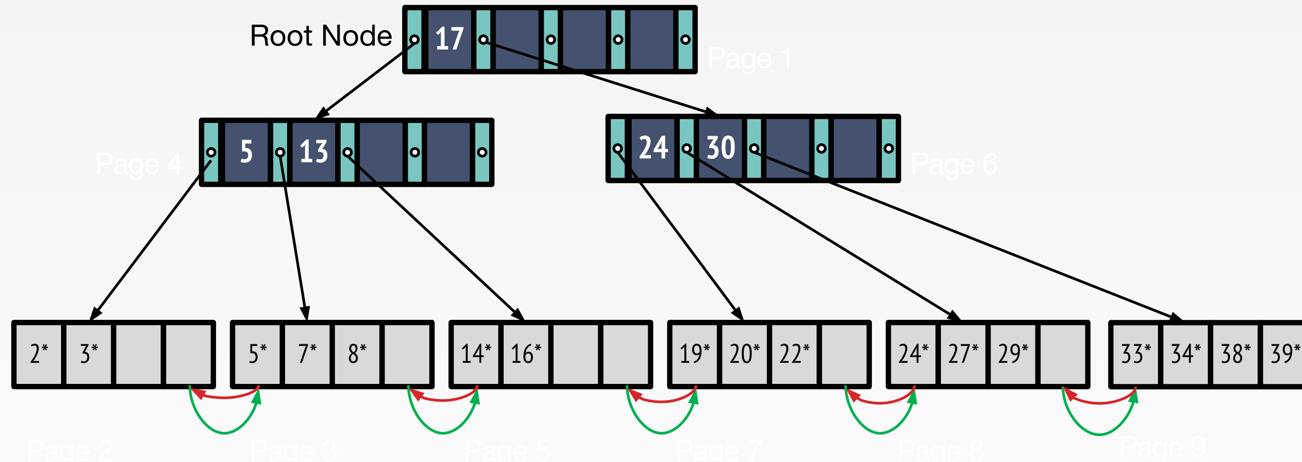


B^+ Tree

- In a variation called B^+ Tree only leaf nodes can have data records, and leaf nodes are linked with siblings
  - Search cost is more predictable
  - Tree can have smaller height
  - Enable sequential scanning of leaf nodes



Example: B+ Tree



- Each interior node is at least partially full:
  - d <= #entries <= 2d (* root: 1<= #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
- Height: the length of a path from the root to a leaf

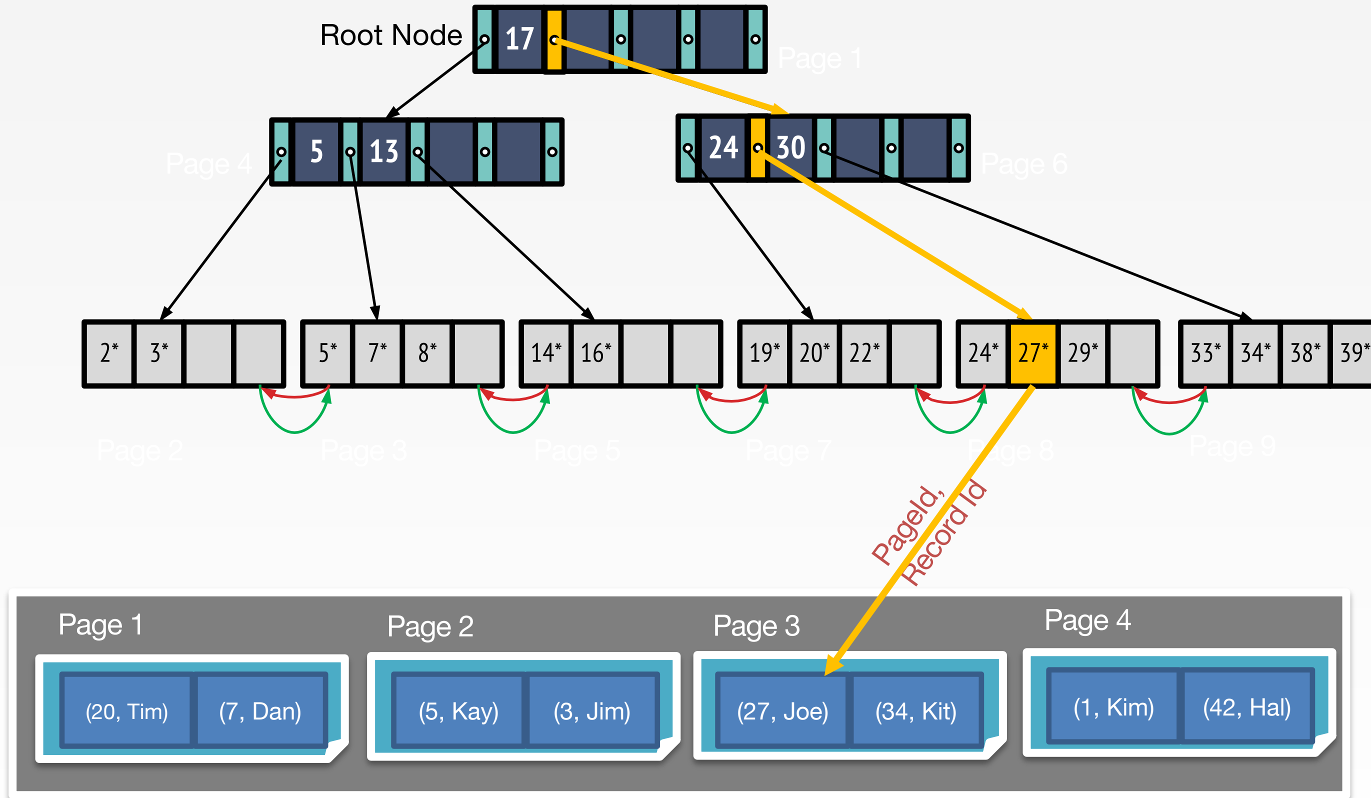
B+ Trees and Scale

- The height of a B+ tree is rarely more than 3 or 4
- How big is a B+ tree
  - d = 2
    - Fan-out = 5
    - Height 1: 5 x 4 = 20 Records
    - Height 3: 5^3 x 4 = 500 Records
  - d = 50
  - d = 100

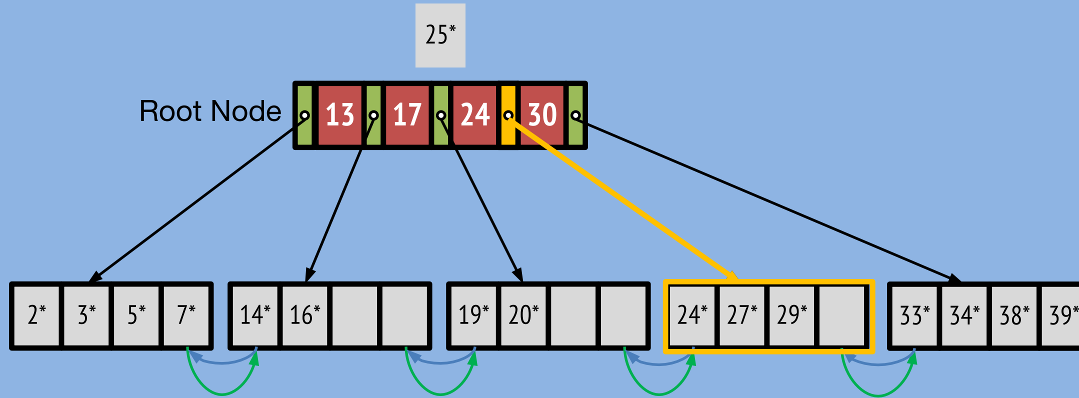
B+ Trees in Practice

- 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: Find 27

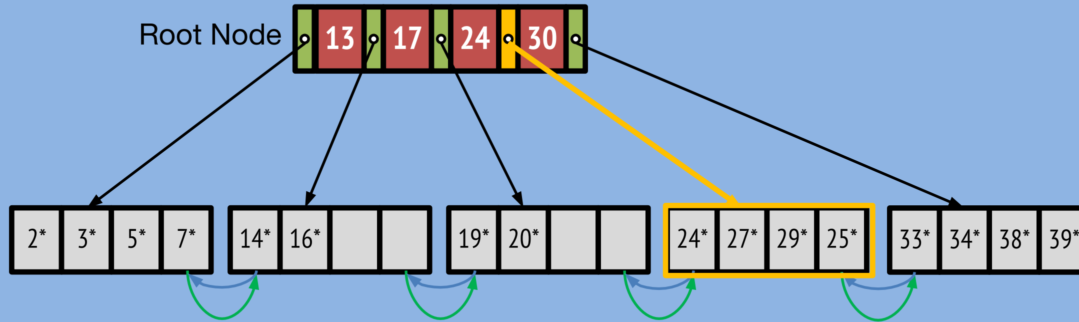


Inserting 25* into a B+ Tree Part 1



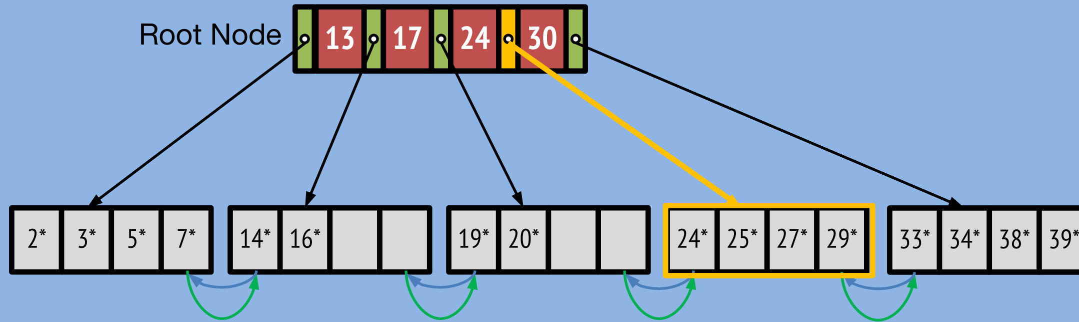
- Find the correct leaf

Inserting 25* into a B+ Tree Part 2



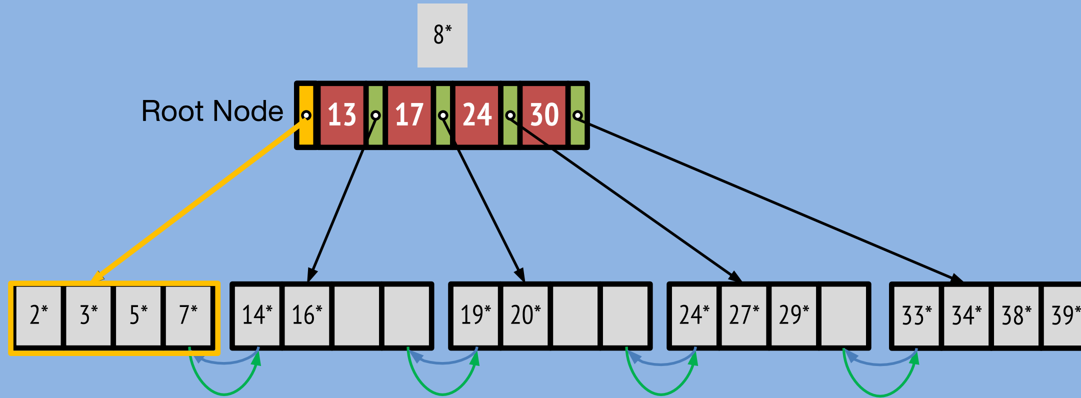
- Find the correct leaf
- If there is room in the leaf just add the entry

Inserting 25* into a B+ Tree Part 3



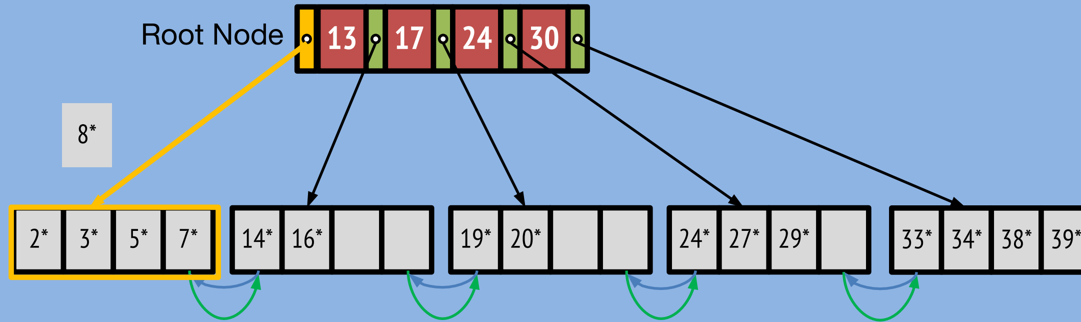
- Find the correct leaf
- If there is room in the leaf just add the entry
  - Sort the leaf page by key

Inserting 8* into a B+ Tree: Find Leaf



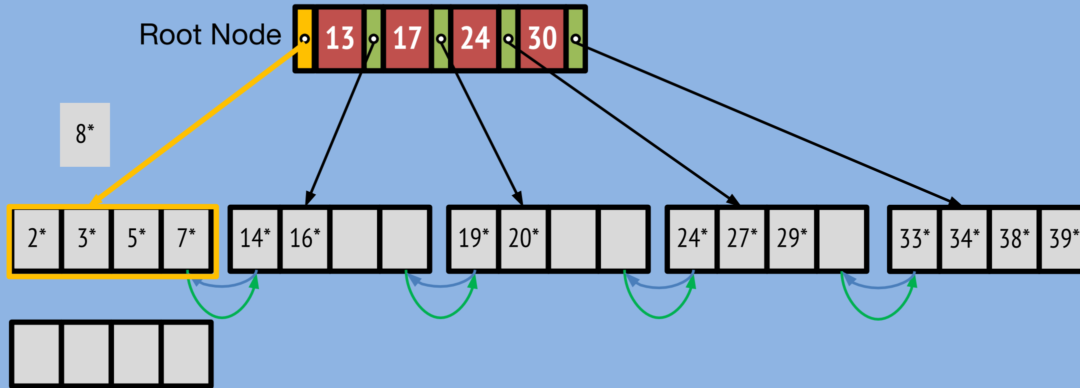
- Find the correct leaf

Inserting 8* into a B+ Tree: Insert



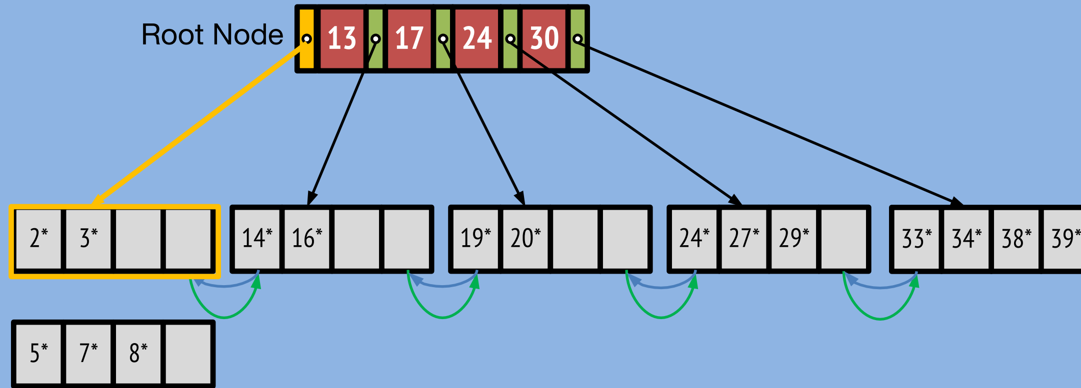
- Find the correct leaf
  - Split leaf if there is not enough room

Inserting 8* into a B+ Tree: Split Leaf



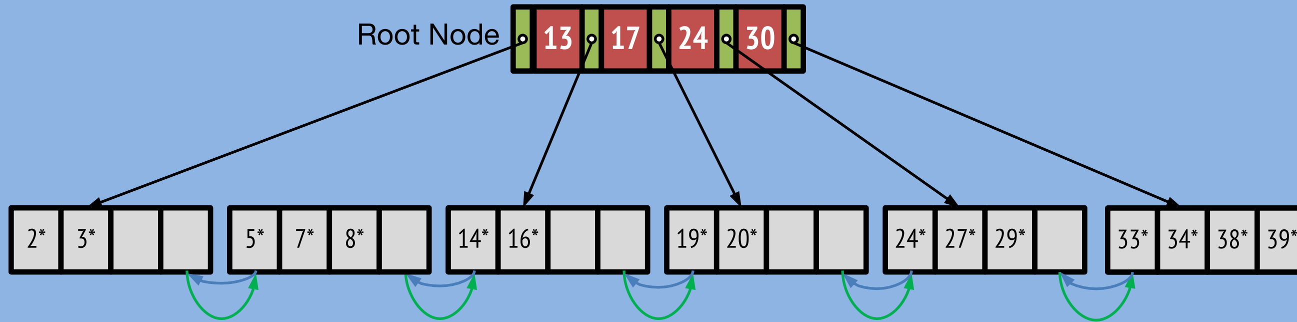
- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly

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



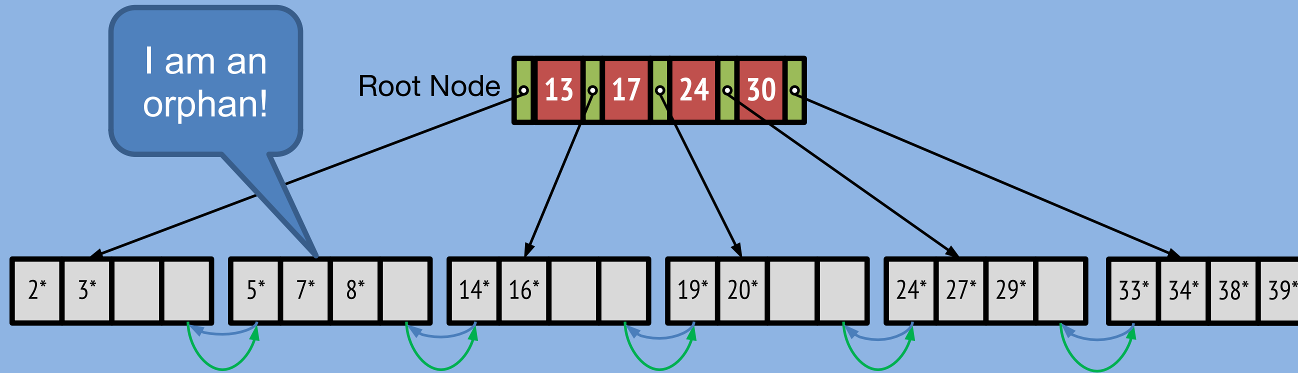
- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - Fix next/prev pointers

Inserting 8* into a B+ Tree: Fix Pointers



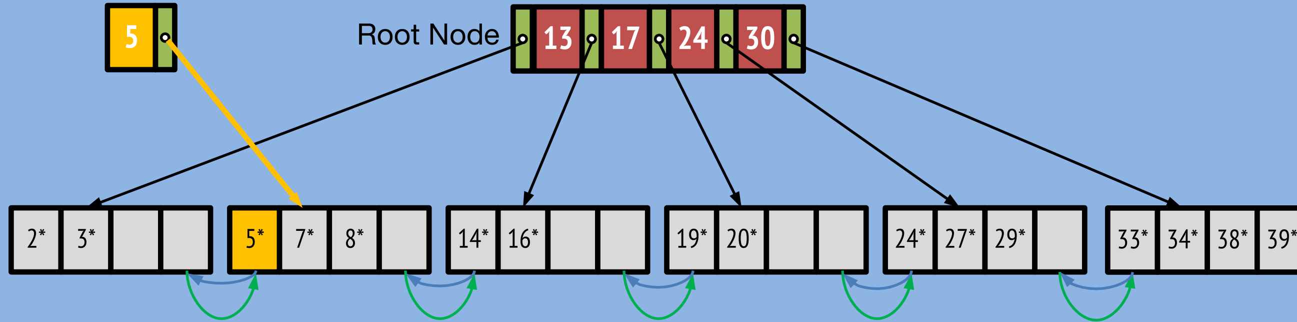
- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - Fix next/prev pointers

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



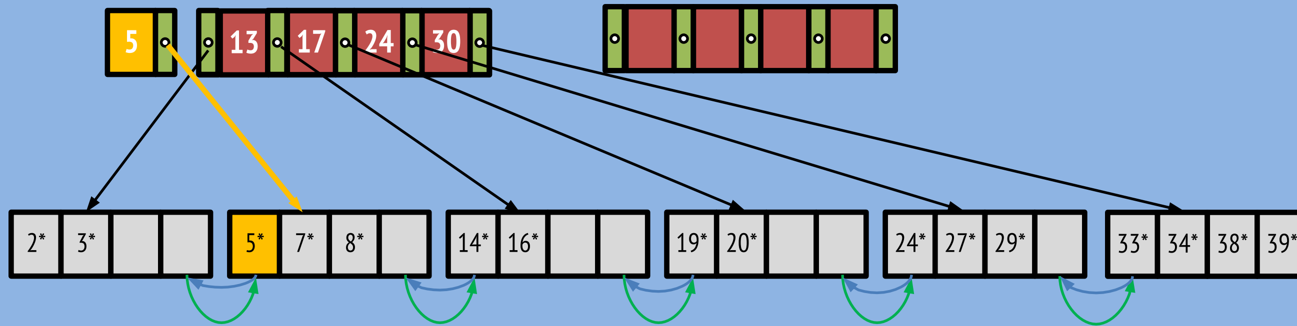
- Something is still wrong!

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



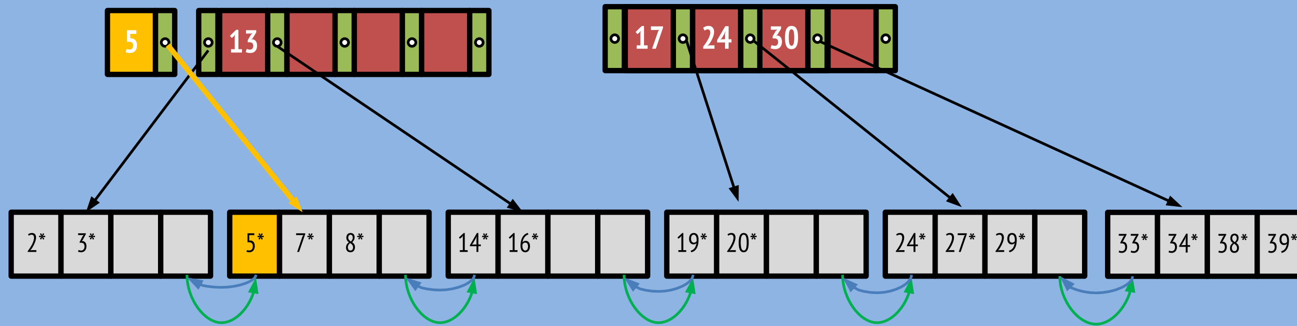
- **Copy up from leaf** the middle key
- No room in parent? Recursively split index nodes

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

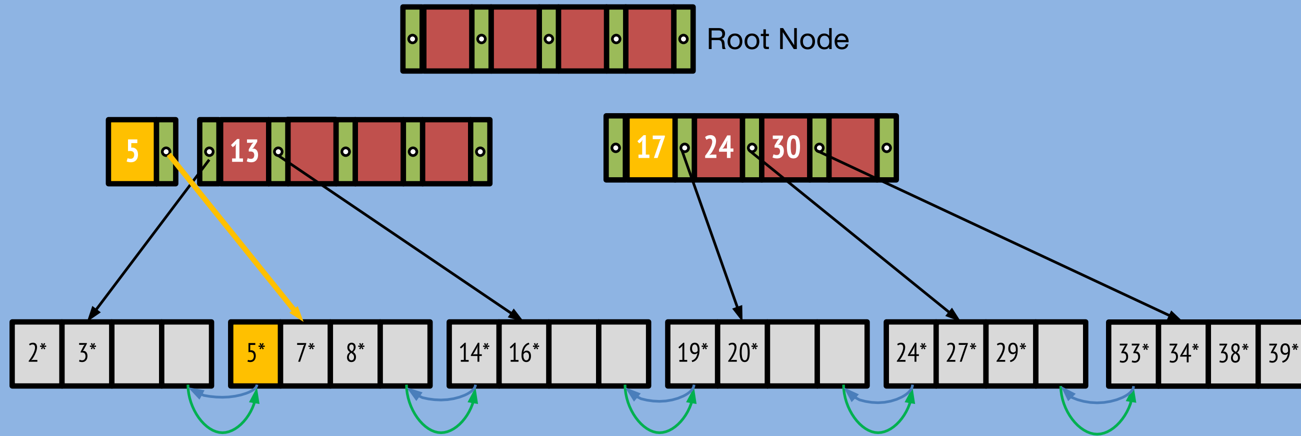
Inserting 8* into a B+ Tree: Split Parent, Part 2



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

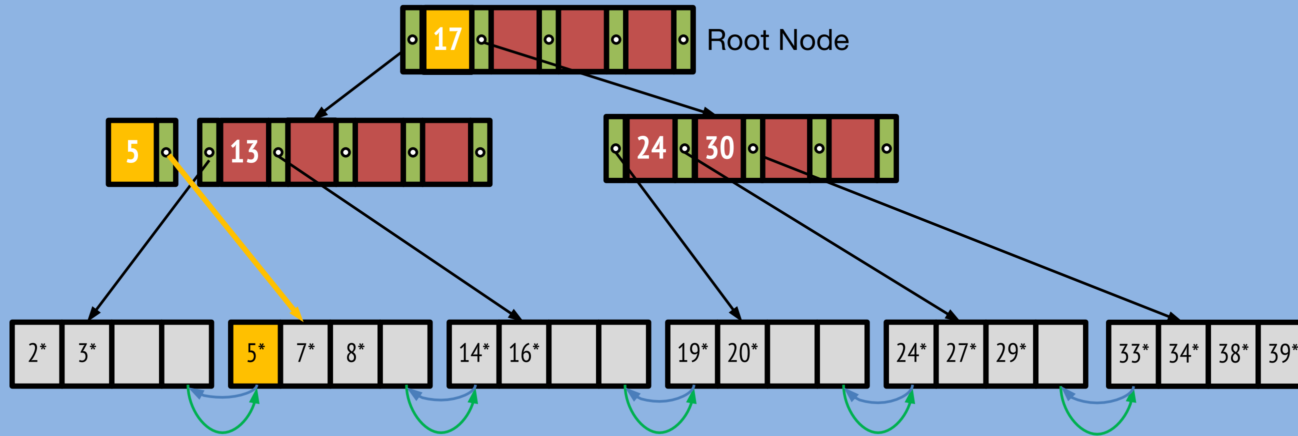
VIRGINIA TECH.

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



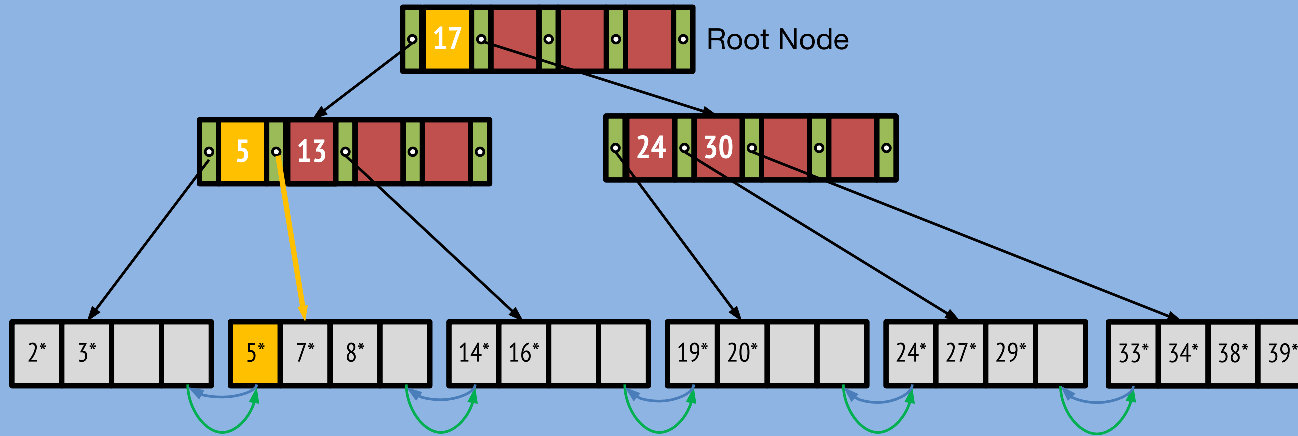
- **Push up from interior node** the middle key
  - Now the last key on left
- No room in parent? Recursively split index nodes
  - Redistribute the rightmost d keys

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



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

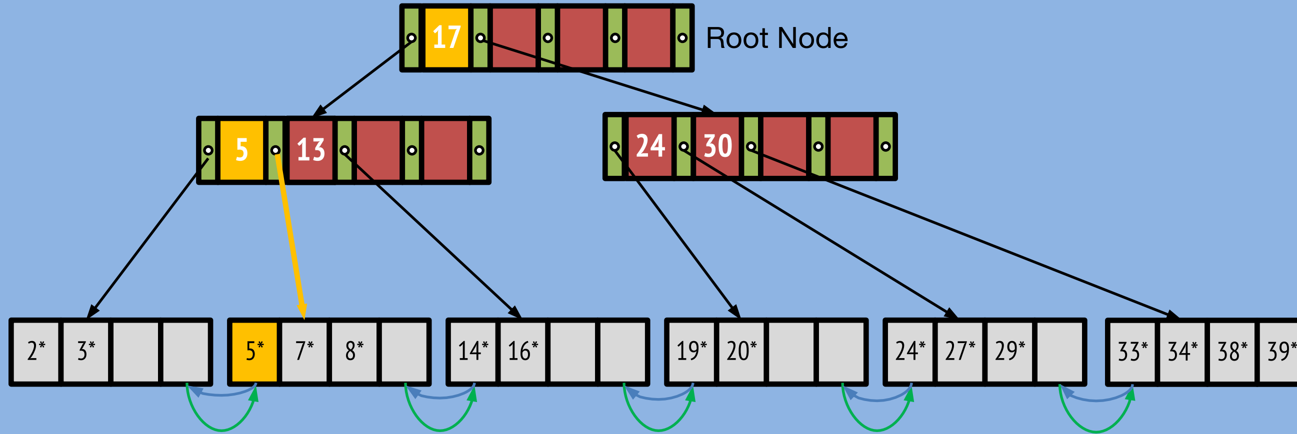
Inserting 8* into a B+ Tree: Root Grows Up, Pt 3



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

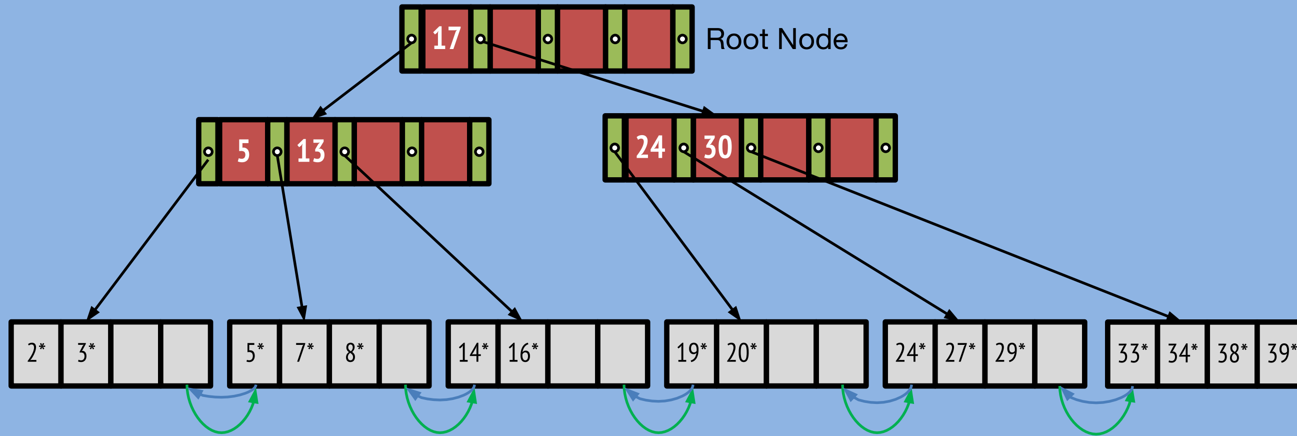
VIRGINIA TECH™

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

VIRGINIA TECH™

B+ Tree Insert: Algorithm Sketch

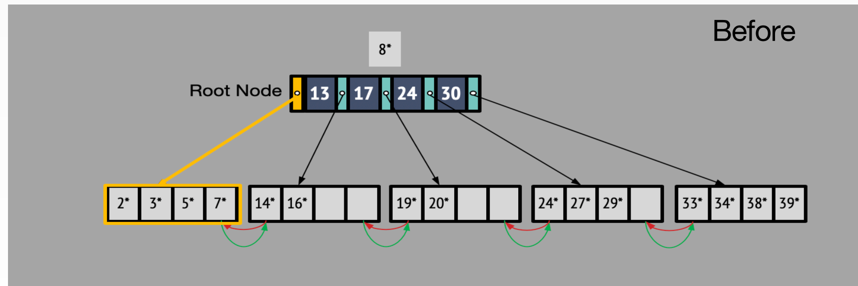
1. Find the correct leaf L.
2. Put data entry onto L.
   - If L has enough space, done!
   - Else, must split L (into L and a new node L2)
     - Redistribute entries evenly, copy up middle key
     - Insert index entry pointing to L2 into parent of L.

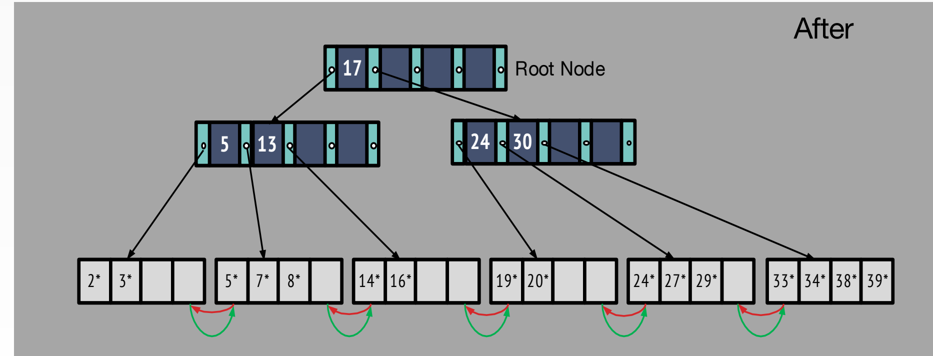
B+ Tree Insert: Algorithm Sketch Part 2

- 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

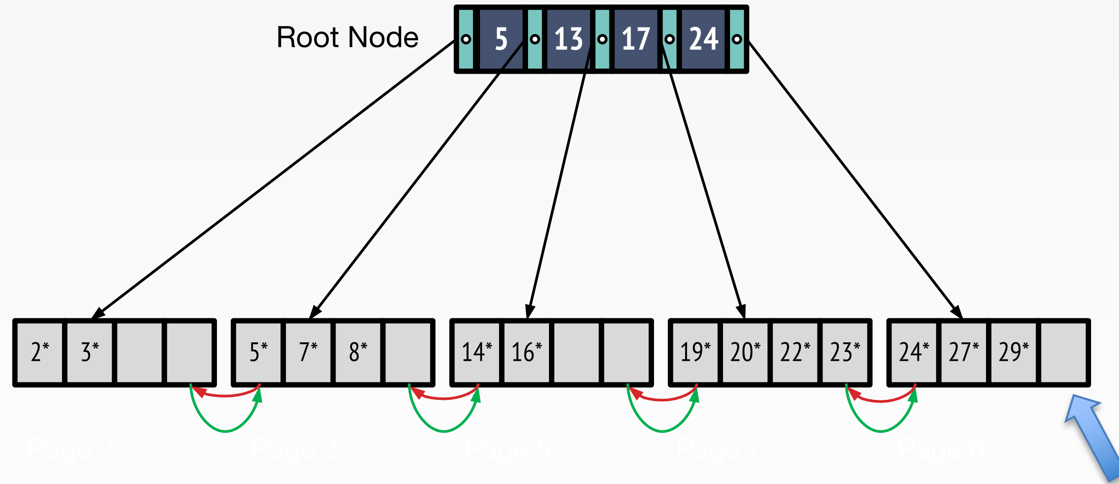
- Notice that the root was split to increase the height
  - Grow from the root not the leaves
  - All paths from root to leaves are equal lengths
- Does the occupancy invariant hold?
  - Yes! All nodes (except root) are at least half full





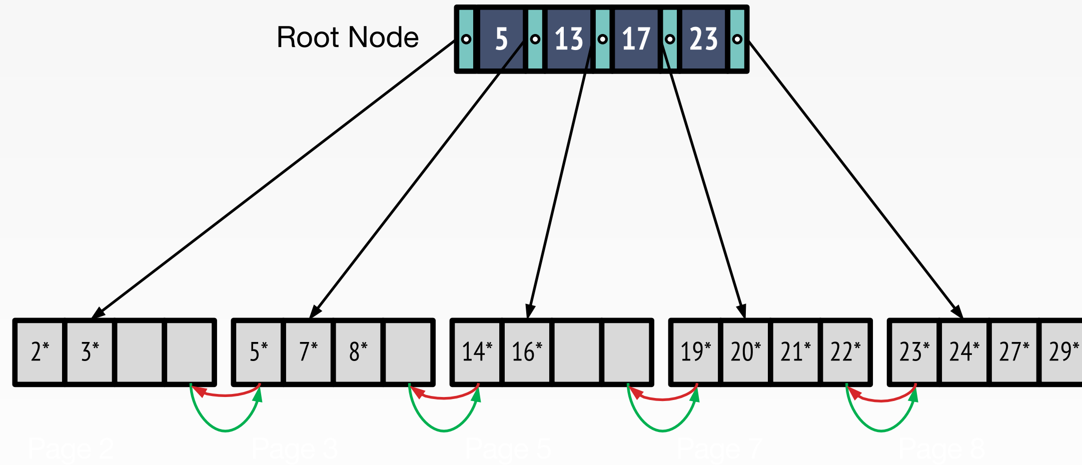
Insert: The Deferred Split

Insert 21



Insert: The Deferred Split

Done Insert 21

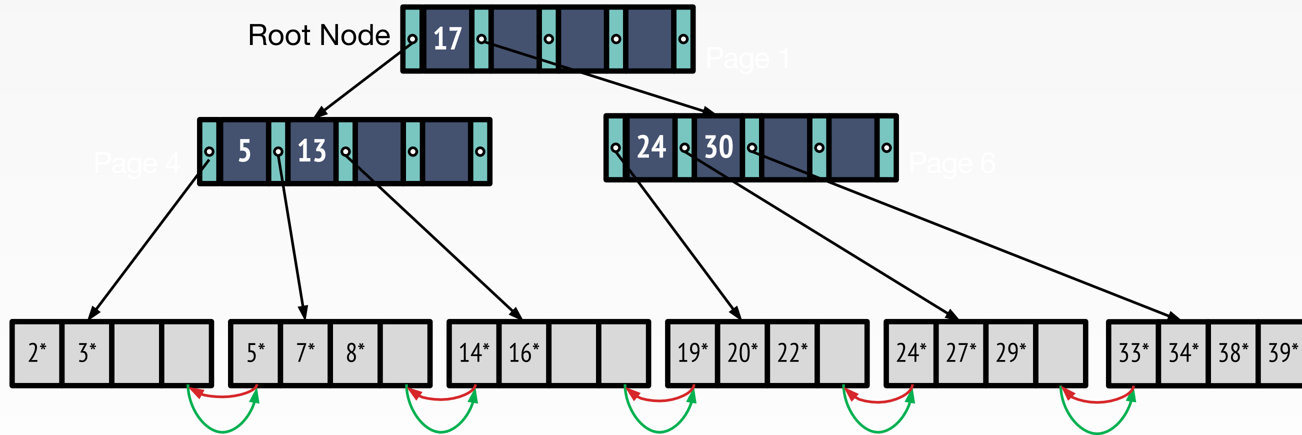


Deleting a Data Entry from a B+ Tree

- 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, borrowing from sibling (adjacent node with same parent as L)
    - If re-distribution fails, merge L and sibling.
      - update parent
      - and possibly merge, recursively

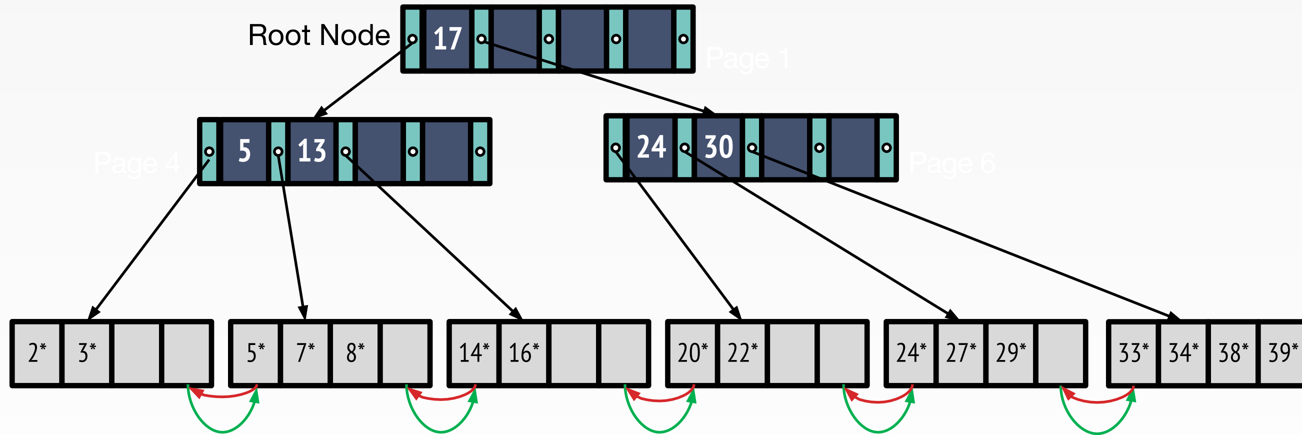
Deletion from B+Tree

Delete 19



Deletion from B+Tree

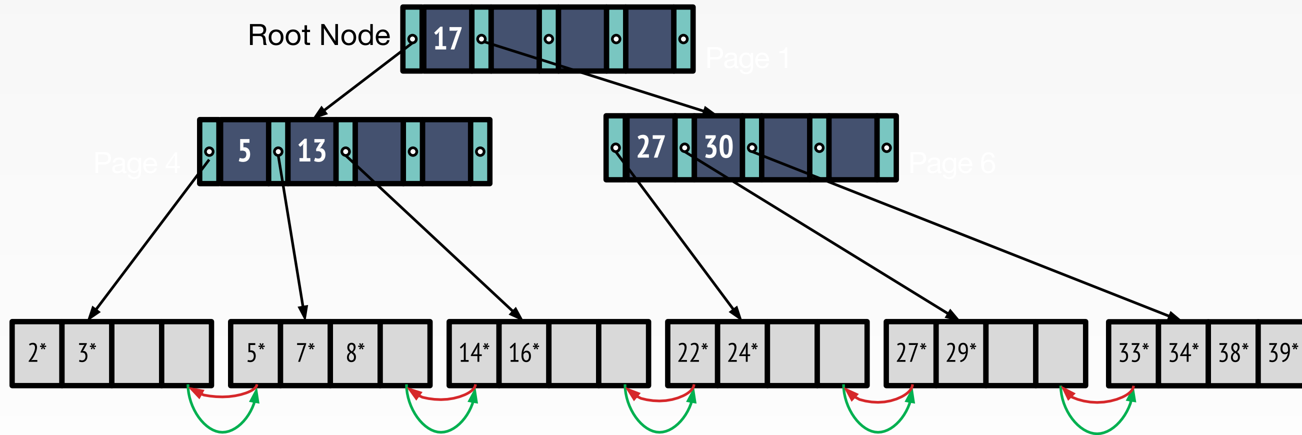
Done Delete 19
Next: Delete 20



VIRGINIA TECH™

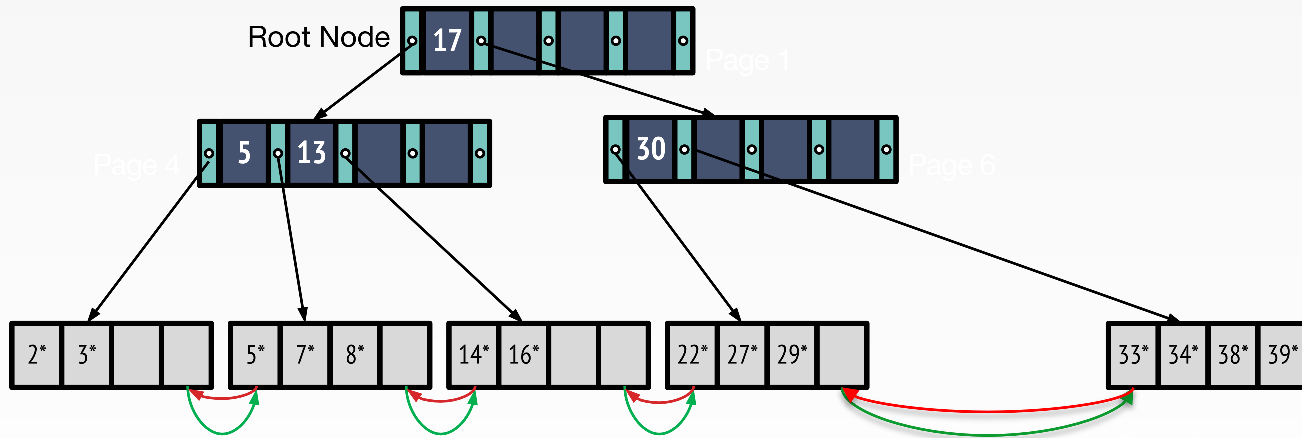
Deletion from B+Tree

Done Delete 20
Next: Delete 24



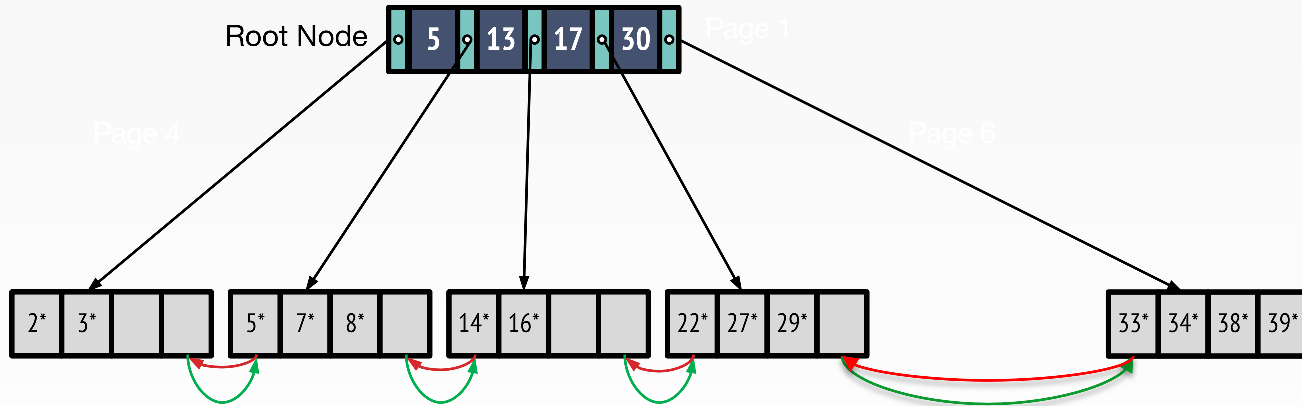
Deletion from B+Tree

Delete 24 – step 1



Deletion from B+Tree

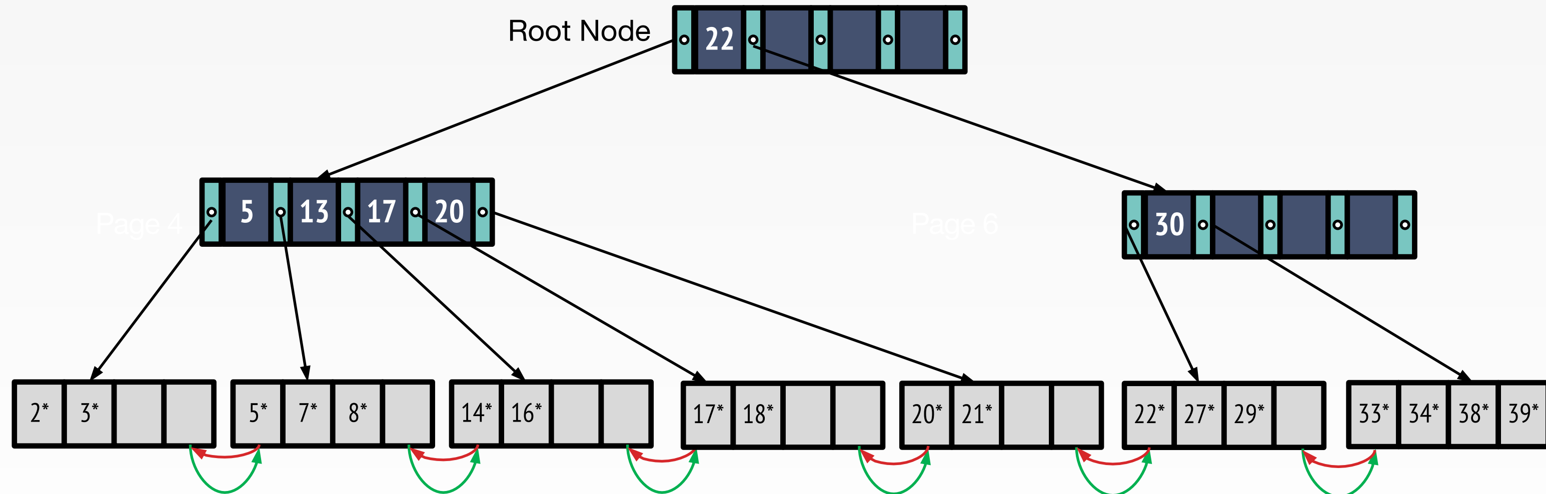
Delete 24 – step 2 Done



VIRGINIA TECH

Deletion from B+Tree: New Example

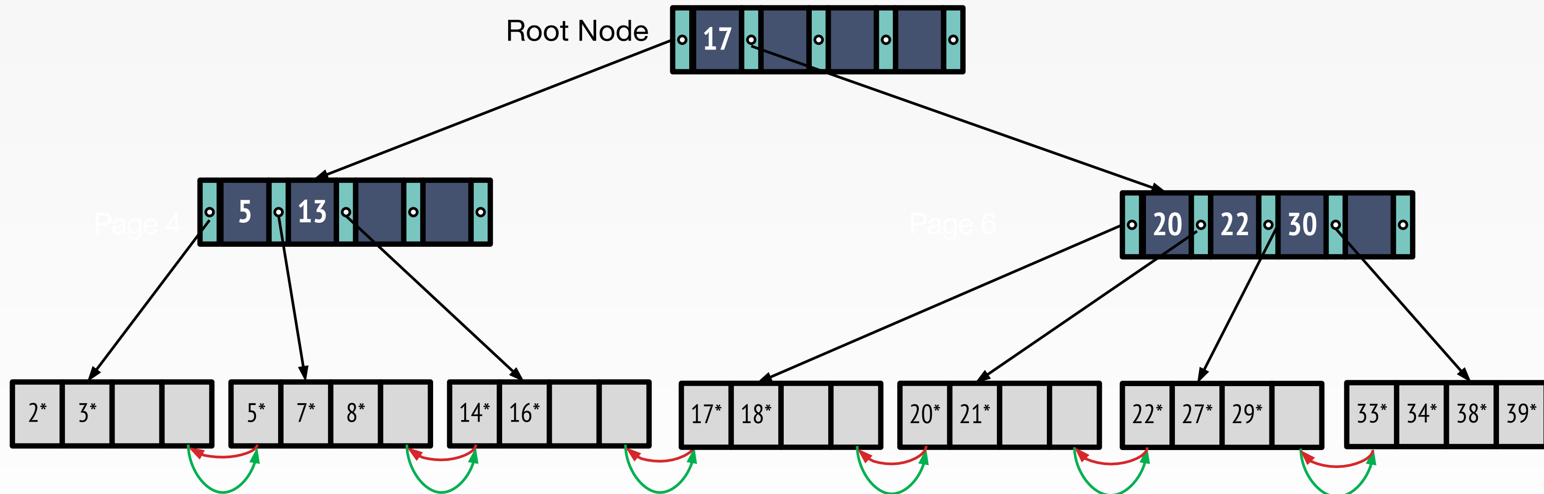
During deletion of 24 – step 1



VIRGINIA TECH

Deletion from B+Tree: New Example

During deletion of 24 – step 2

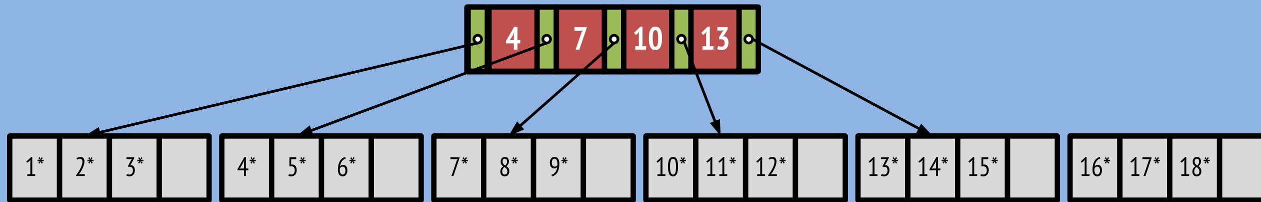


VIRGINIA TECH

Bulk Loading of a B+ Tree

- Suppose we want to build an index on a large table
  - Insert repeatedly? Too slow!
- Constantly need to search from leaf
- Leaves and internal nodes mostly half full
- **Modifying random pages → Poor cache efficiency**

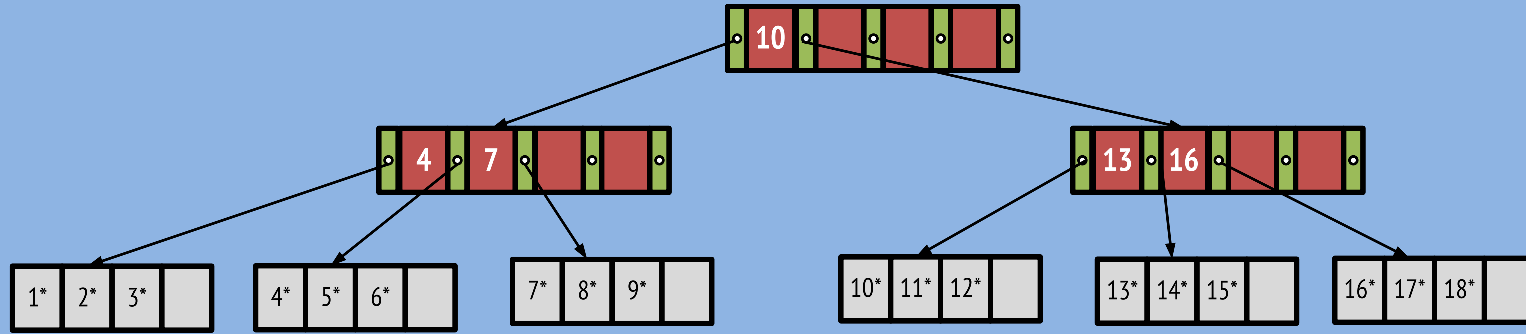
Smarter Bulk Loading a B+ Tree



- 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

VT VIRGINIA TECH™

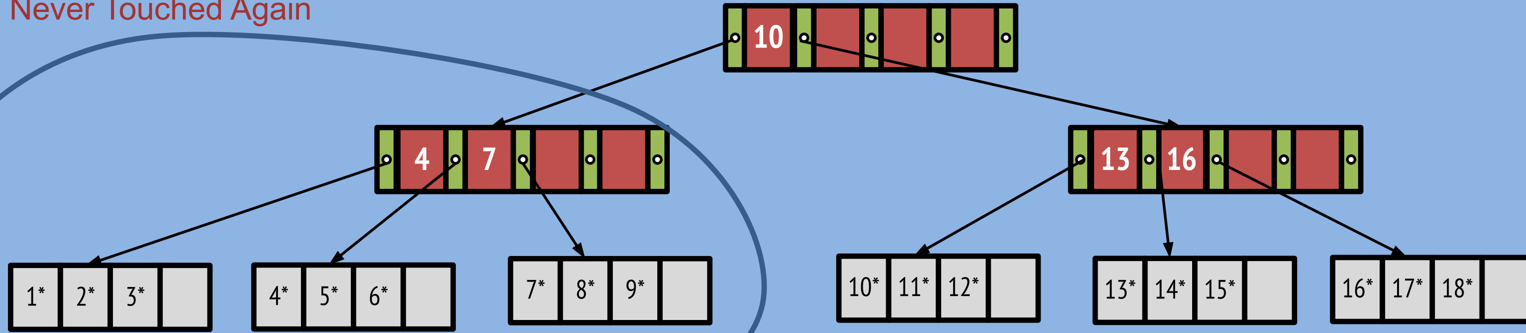
Smarter Bulk Loading a B+ Tree Part 2



- 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 (50/50) and copy to sibling

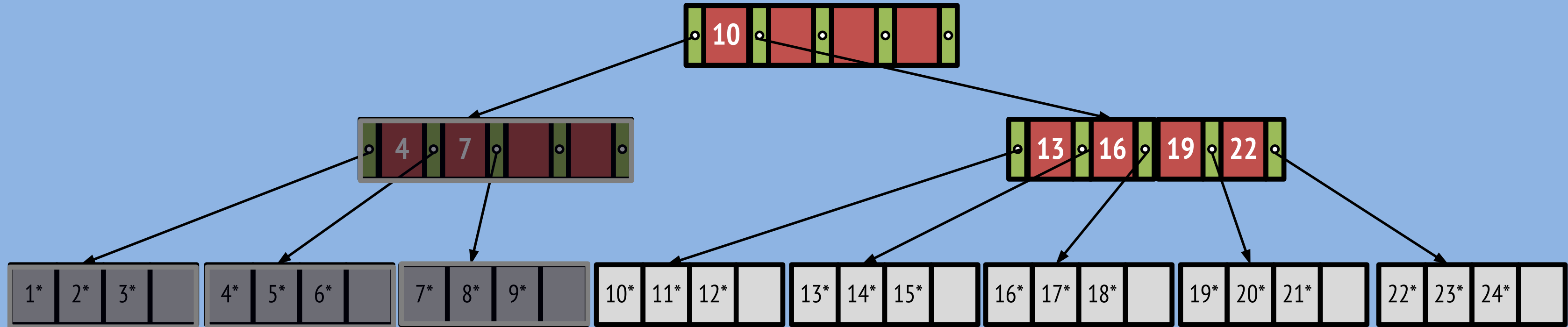
VIRGINIA TECH™

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

VIRGINIA TECH

Bulk Loading Summary

- Option 1: Multiple inserts
  - **Slow**
  - Does not give sequential storage of leaves
- Option 2: Bulk Loading
  - Fewer I/Os during build.
  - Leaves will be stored sequentially (and linked, of course)
  - Can control “fill factor” on pages.

B+ Tree Summary

- **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.
- 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

B+ Tree Animation

- Great animation online of B+ Trees

Summary

- The unit of (disk) I/O is a page.
- A page includes multiple records.
- The buffer manager moves pages between secondary storage and main memory
- There are different methods to physically organize data records.
- There are also different methods to organize pointers to data records in order to speed up searches without affecting the underlying record organization

Reading and Next Class

- Storing Data and Indexes
  - Ch 8.1, 8.2
  - Ch 9.1, 9.4
  - Ch 10.3 - 10.8
- Next: Hashing and Sorting
  - Ch11
  - Ch 13

Credits

- The animation Page 33-50 and some slides are adopted from UC Berkeley CS W 186.