Query execution

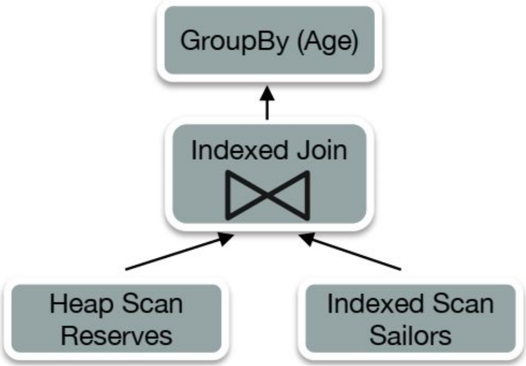
Cost estimation

Source: https://cs186berkeley.net/

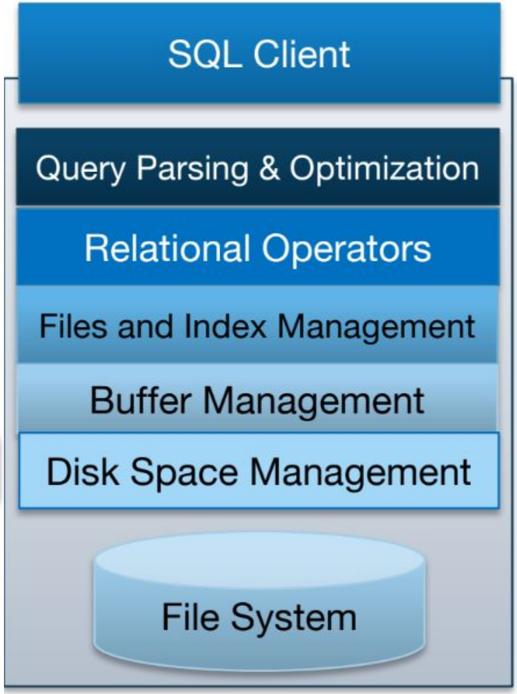
Introduction (recall)

Parsing & Optimization: Parse and check SQL and translate into an efficient relational query plan

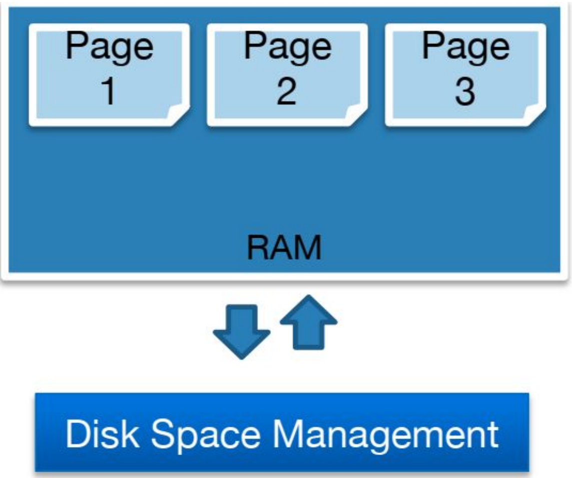
Relational Operators; Execute query by operating on records and files



Files and Index Management
Organize tables and Records as groups of pages in logical (in buffer memory) file



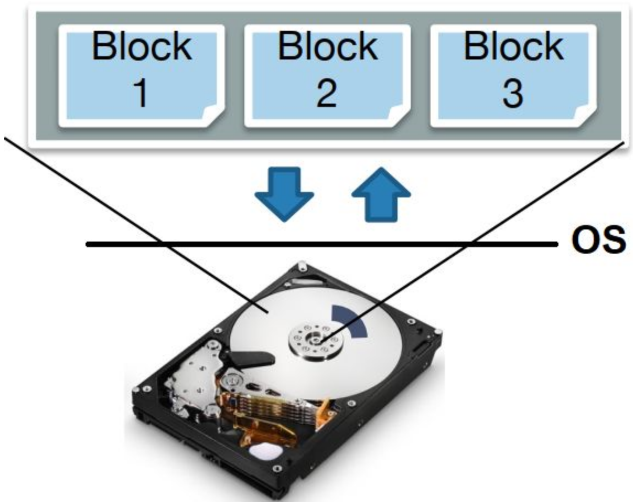
Buffer Management: provide the illusion of operating in memory



Disk Space Management:

Purpose

- Map pages to locations on disk
- Load pages from disk to memory
- Save pages back to disk & ensuring writes



Higher levels call upon this layer to:

- Read/write a page
- Allocate/de-allocate logical pages

Heap Files: is a physical layer for data storage ifa table, it is structured as unordered collection of records (tuples)

To access a heap file for querying or managing data, API for higher layers of the DBMS: can only READ and WRITE pages.

Clustered Heap Files: Records and pages are grouped in some meaningful way

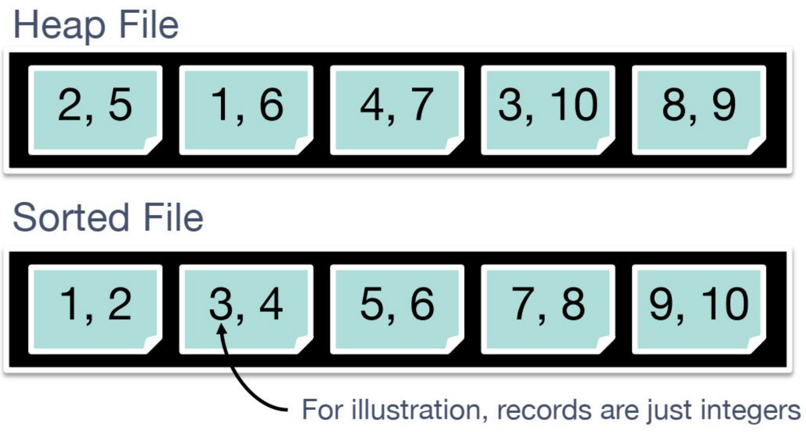
Sorted Files: Pages and records are in strict sorted order

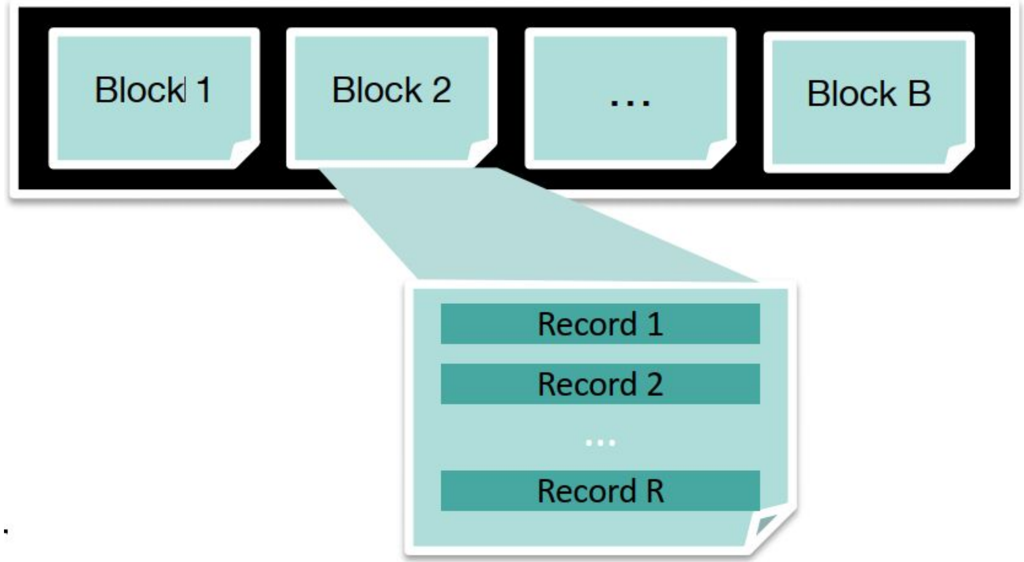
Index Files: contain pointer to records in other files

Question of this course: “How? At what cost?” to:

- Insert/delete/modify record
- Fetch a particular record
- Scan all records
  - Possibly with some conditions on the records to be retrieved

Heap Files & Sorted Files





B: Number of data blocks = 5

R: Number of records per block = 2

D: (Average) time to read/write disk block = 5ms

Cost of Operations: Scan?

	Heap File	Sorted File
Scan all tuples (records)	**B * D**	**B * D**
Equality search	**½ B * D**	**(Log_2 B) * D**
Range search	**B * D**	**((Log_2 B)+#pages) * D**
Insert	**2 * D**	**((log2 B)+B)*D**
Delete	**(B/2 + 1) * D**	**((log2 B) + B) * D**

B: Number of data blocks (pages) = 5

R: Number of records per block (page) = 2

D: (Average) time to read/write disk block (page) = 5ms

Equality selection

Heap File

Assume for equality selection => exactly one match

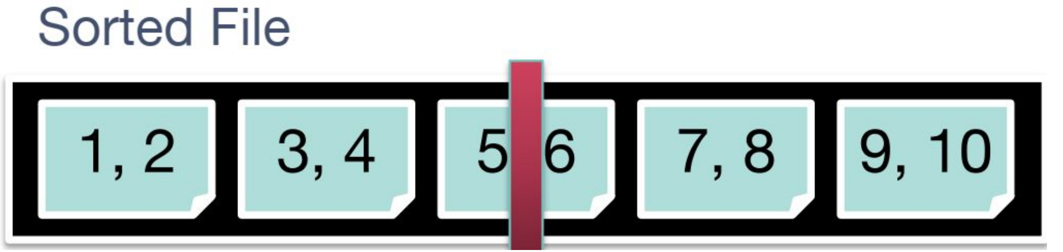
Find the tuple having the key 8 in a **heap file**

- P(i): Probability that key is on page i is 1/B
- T(i): Number of pages touched if the selected key is in page i

Therefore the expected number of pages touched is:

[for i from 1 to B] (cost of reading page i) * (prob that key is on page i)

$$\sum_{i=1}^{B} \mathrm{T}(i)\mathbf{P}(i) = \sum_{i=1}^{B} i\frac{1}{B} = \frac{B(B+1)}{2B} \approx \frac{B}{2}$$



Find the tuple having the key 8 in a **sorted file**

The used method for searching the page containing the searched tuple **is binary search**

Similar to binary search in an array but here each access is a page read

Pages touched in binary search **$Log_2(B)$** (worst case ≈ average case)

Range Search

Find tuples their values Between 7 and 9: Heap File

Always touch all blocks. Why?

=> to get all tuples that belongs to the range (for real number, strings or integers with duplicates in the searched value)

Find Records Between 7 and 9: sorted file

- Find beginning of range
- Scan right or left page after page until the end of the range

Cost = Log2(B) + #pages containing the range

Insertion

Assuming Single record for insert

Ex. Insert 4.5: Heap File

- Stick at end of file
- Cost = 2*D (read last page, append, write the page)

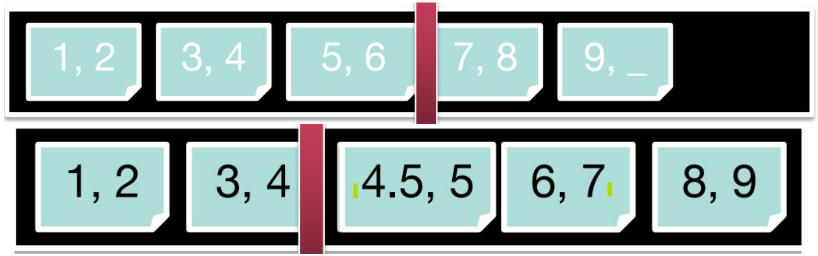
Insert 4.5: Sorted File

Find location for record. Cost = (log2B) * D

Insert and shift rest of file. Average cost = (B/2) * 2 * D = B * D

Total: find cost + insert and shift cost = (log2B) * D + B * D = ((log2B) + B) * D

Heap File



Deletion

Heap file

- Find the record: average case to find the record: B/2 reads
- Delete the record from the found page page
- Write the page
- Cost = (B/2 + 1) * D (why +1? => for W)

Sorted File

- Find location for record. Cost = log2B
- Delete record in page à Gap
- Read the rest into memory, shift by 1 record, and write back: 2 * (B/2) = B
- Total: find cost + delete and shift cost = (log2B) * D + B * D = ((log2 B) + B) * D

Can we do better?

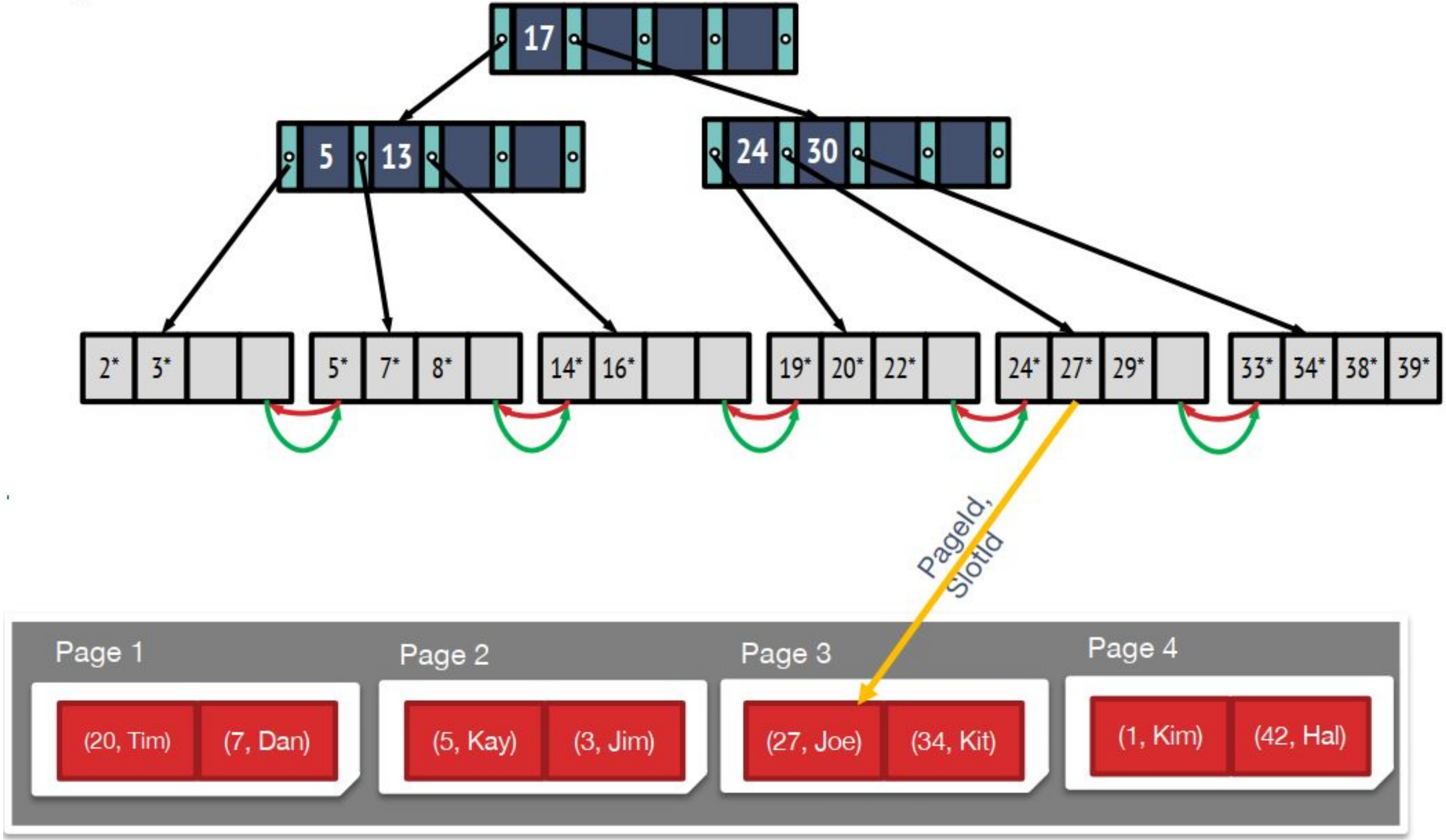
- Indexes!

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)
- **Data Entries**: items stored in the index
  - Contain a way access a record: a pair (key recordId) …
    - Pointers to records in Heap Files; assume a pair (key, recordId)

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


17
5
13
24
30
2*
3*
5*
7*
8*
14*
16*
19*
20*
22*
24*
27*
29*
33*
34*
38*
39*
PageId,
SlotId
Page 1
(20, Tim)
(7, Dan)
Page 2
(5, Kay)
(3, Jim)
Page 3
(27, Joe)
(34, Kit)
Page 4
(1, Kim)
(42, Hal)

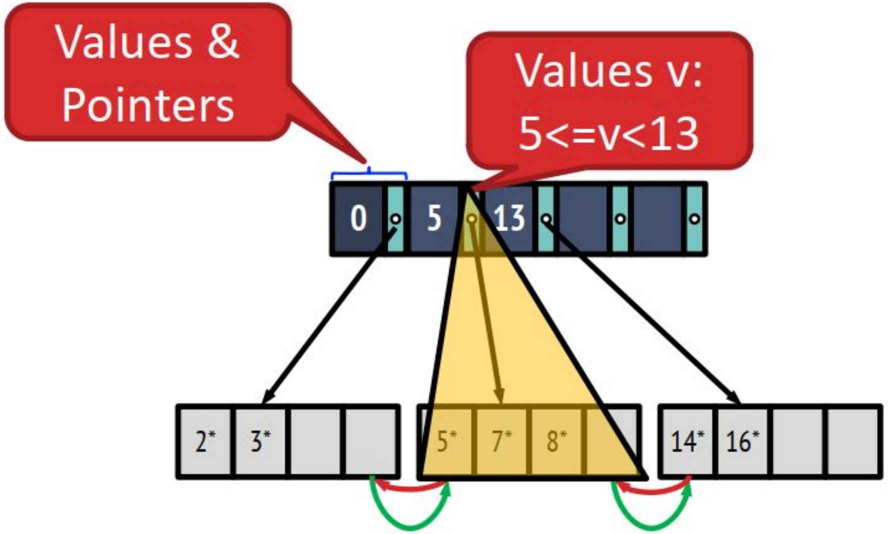
Dynamic Tree Index

- 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

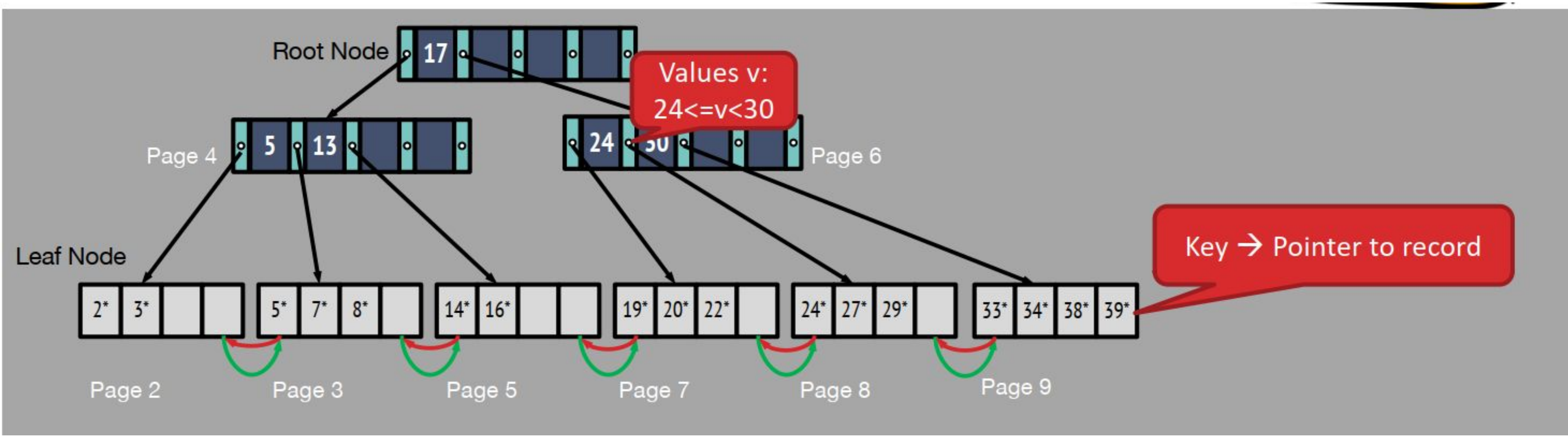
- Node[…, (K_L, P_L), (K_R, P_R)…]

  means that

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



Example of a B+ Tree



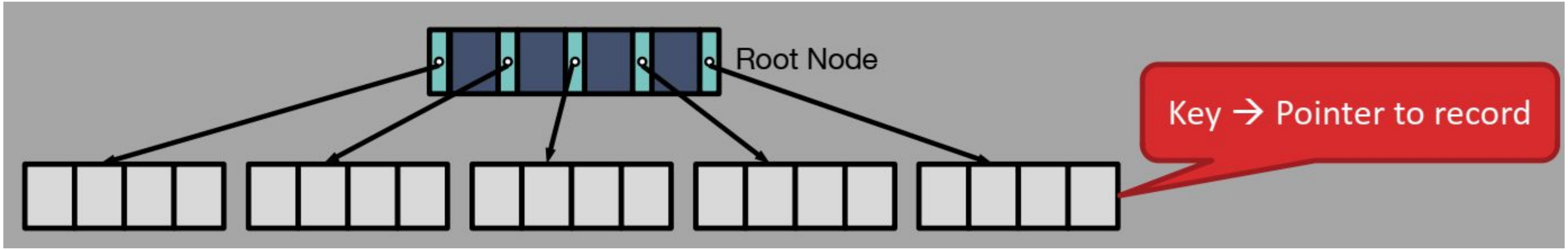
Property 1: Nodes in a B+ tree must obey an occupancy invariant

- Guarantees that lookup costs are bounded
- Invariant: each interior node is full beyond a certain minimum: typically, at least half full
    - This minimum, d, is called the order of the tree
    - Guarantee: d <= # entries <= 2d. Eg d=2, 2 <= # entries <= 4
- Root doesn’t need to obey this invariant

Property 2: Leaves need not be stored in sorted order

- Next and prev. pointers help examining them in sequence

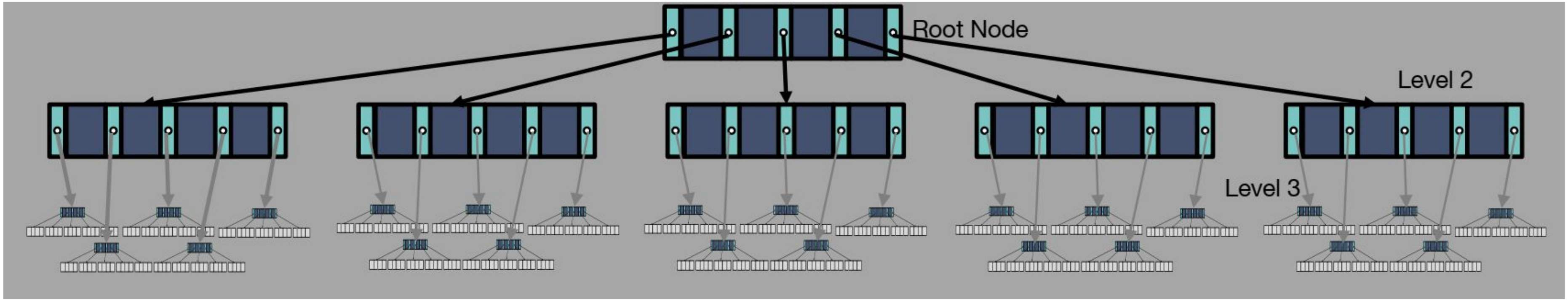
B+ Trees and Scale



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

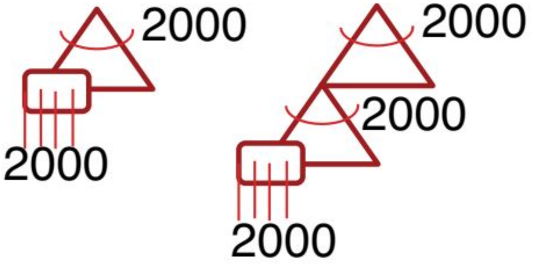


How many records can this height 3 B+ tree index?

- Fan-out = 5; Max entries = 4
- Height 3: 5 (root) x 5 (level 2) x 5 (level 3) x 4 (leaves) = 5^3 x 4 = 500 Records

B+ Trees in Practice

- Say 128KB pages, with around 40B per (val, ptr) pair
  - Max entries = roughly 128KB/40B = approx. 3000
  - Max fanout = 3000+1 = approx. 3000
  - Say 2/3 are filled on average
    - Average fan-out/entries = approx. 2000

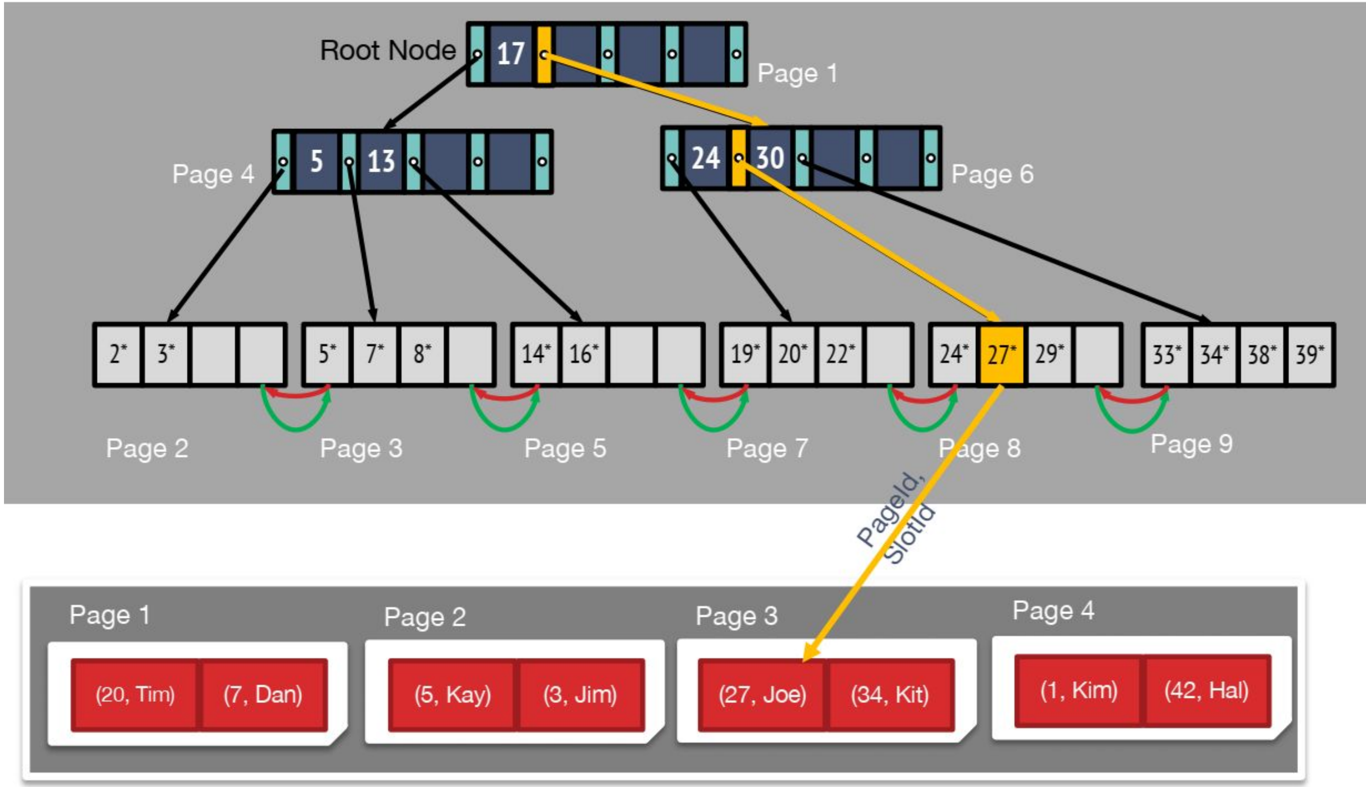


- At these capacities
  - Height 1: 2000 (pointers from root) x 2000 (entries per leaf) = 2000^2 = **4,000,000**
  - Height 2: 2000 (pointers from root) x 2000 (pointers from level 2) x 2000 (entries per leaf) = 2000^3 = **8,000,000,000 records!!**
- **Core takeaway: Even depths of 3 allow us to index a massive # of records!**

Searching the B+ Tree

Find key = 27

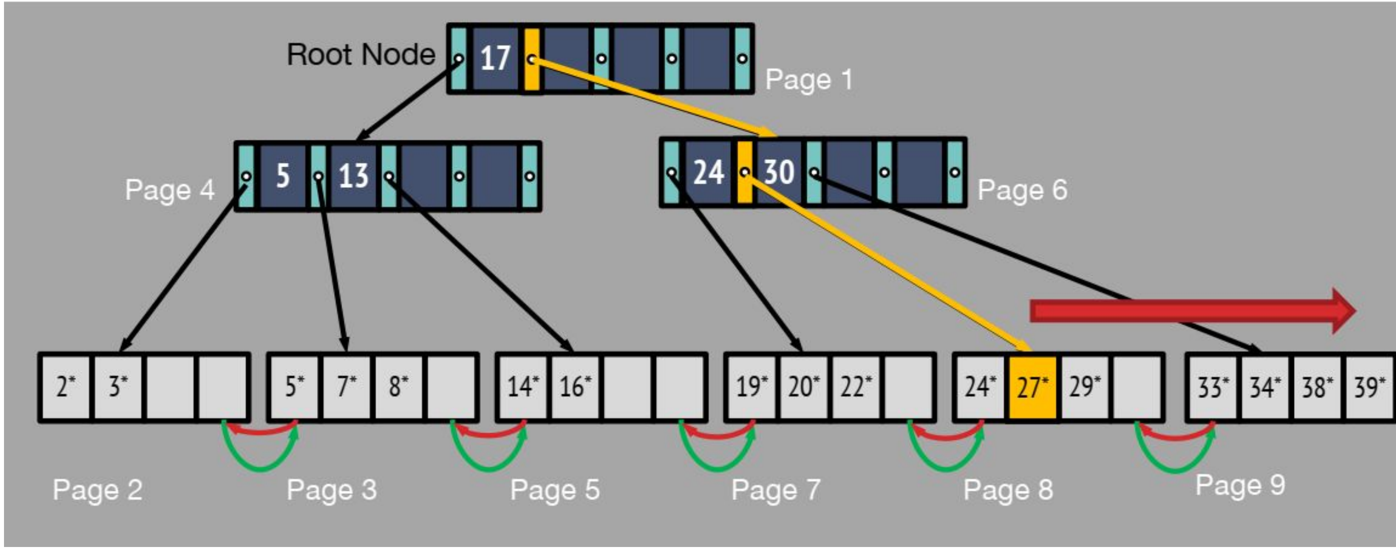
- Find split on each node (Binary Search)
- Follow pointer to next node



Searching the B+ Tree

Find keys >=27

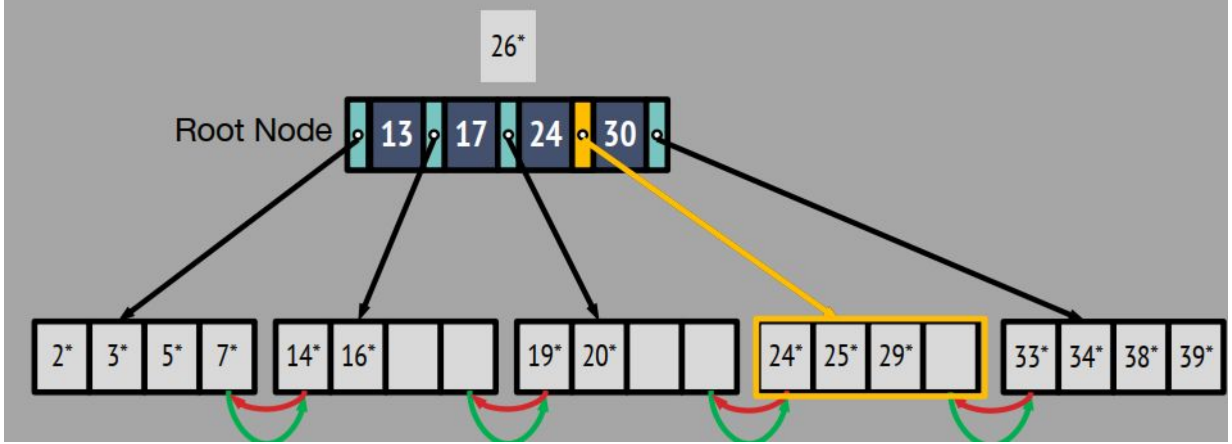
- Find 27 first, then traverse leaves following "next" pointers in leaf
- This is an example of a range scan: find all values in [a, b]
- Benefit: no need to go back up the tree! Saves I/Os



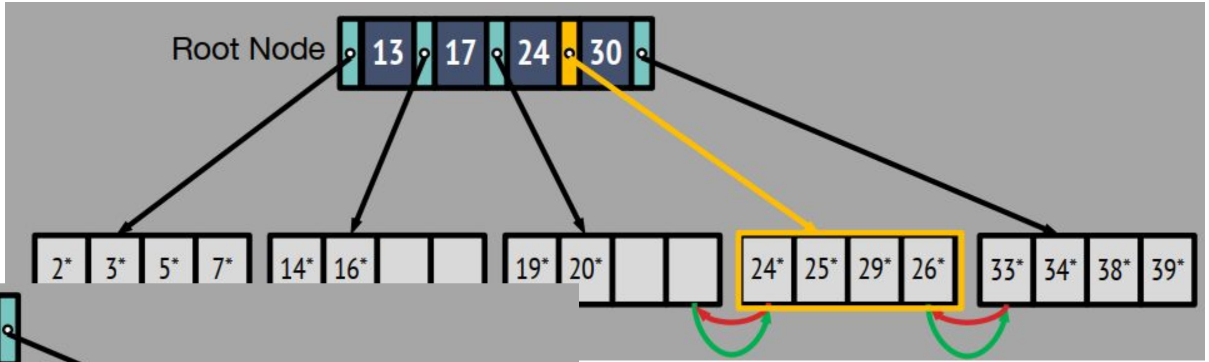
Insertion into a B+ Tree

Inserting 26*

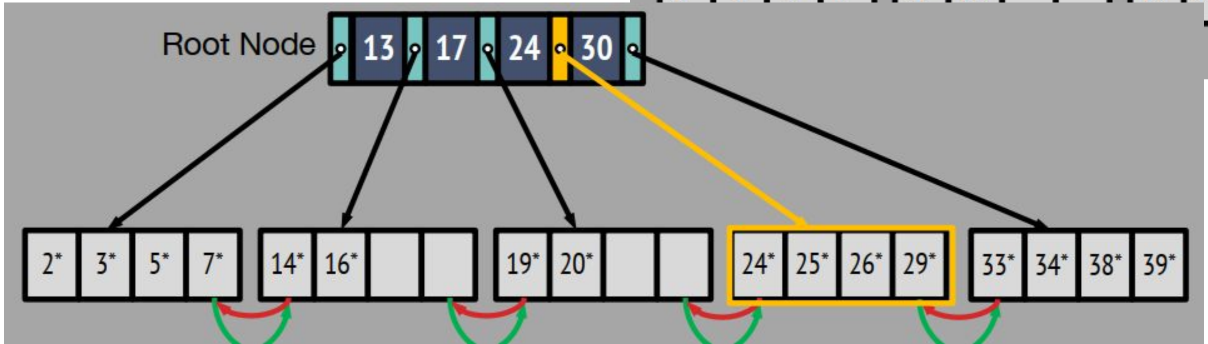
- Find the correct leaf

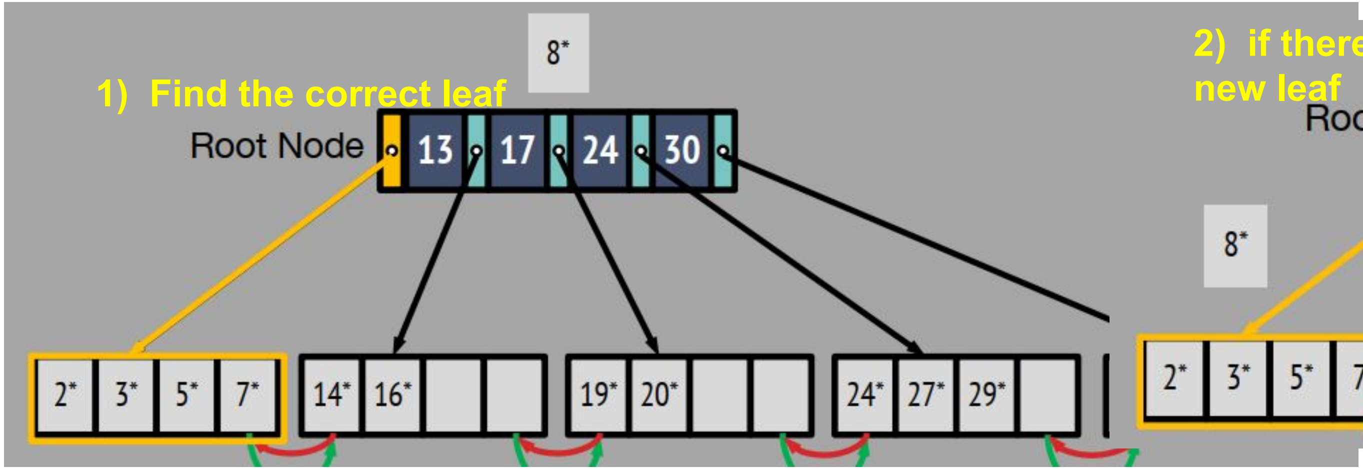


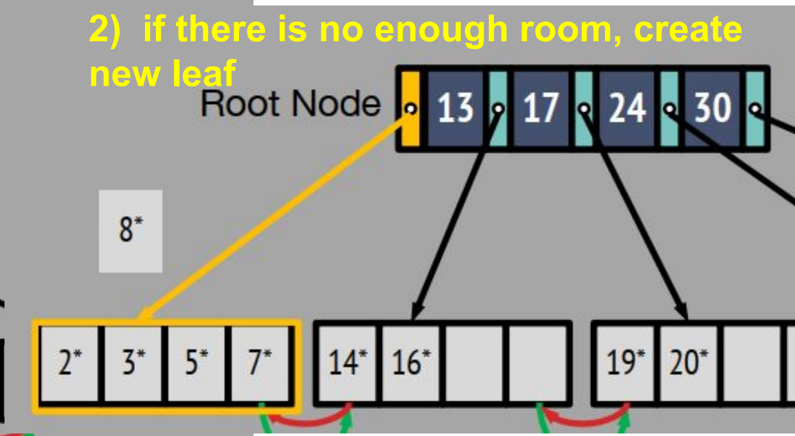
- If there is a room in the leaf, just add the entry

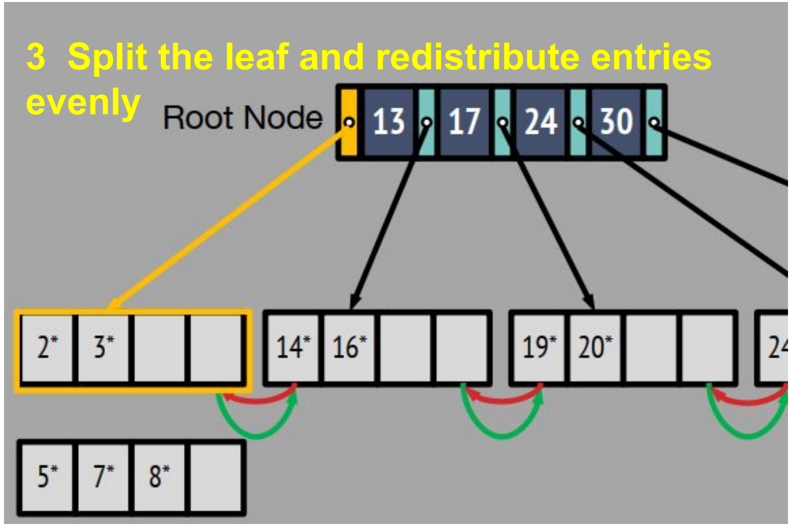


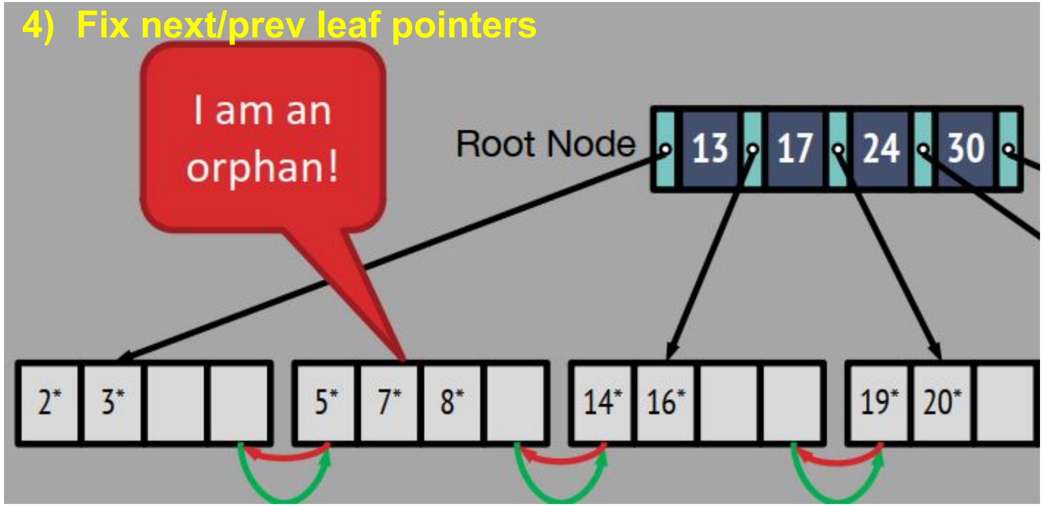
- Sort the leaf page by key



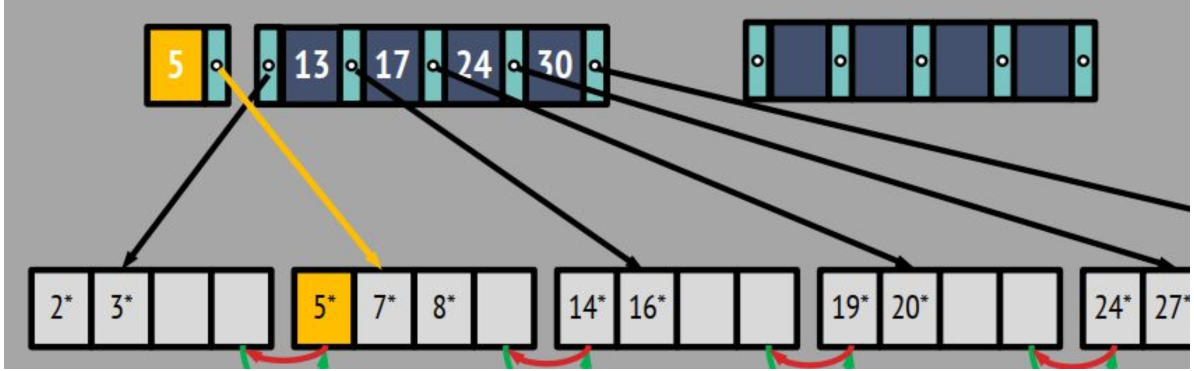

1) Find the correct leaf
8*
Root Node
13
17
24
30
2*
3*
5*
7*
14*
16*
19*
20*
24*
27*
29*

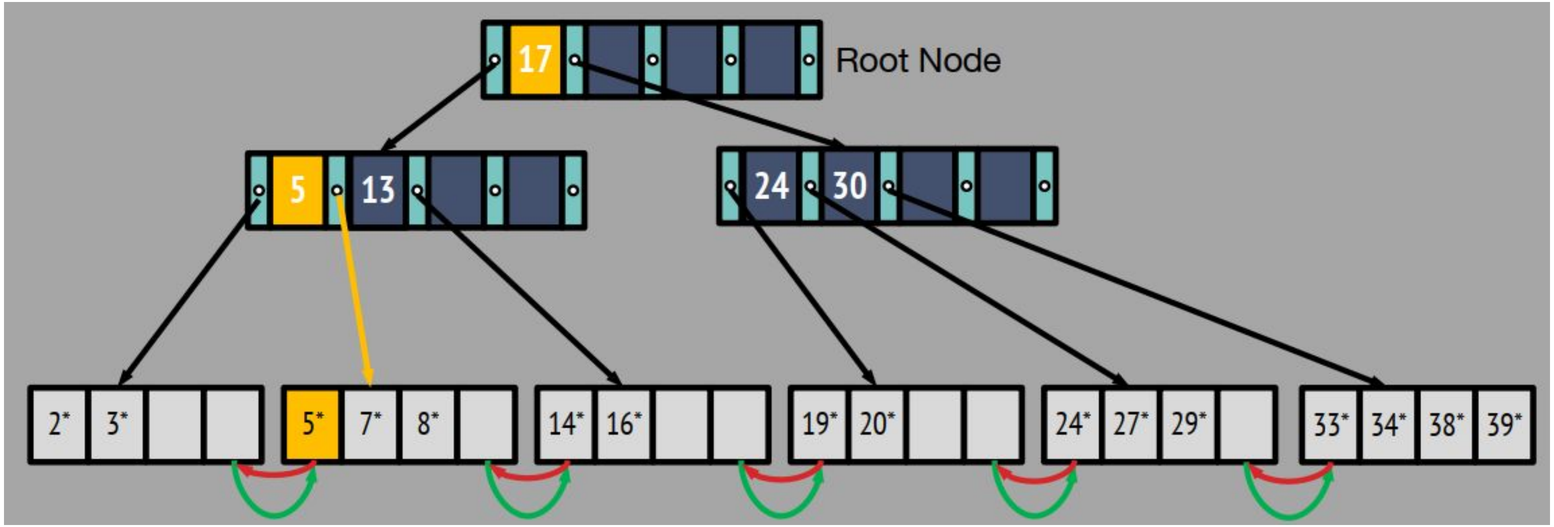

2) if there is no enough room, create new leaf
Root Node
13
17
24
30
8*
2*
3*
5*
7*
14*
16*
19*
20*


3 Split the leaf and redistribute entries evenly
Root Node
13
17
24
30
2*
3*
14*
16*
19*
20*
5*
7*
8*


4) Fix next/prev leaf pointers
I am an orphan!
Root Node
13
17
24
30
2*
3*
5*
7*
8*
14*
16*
19*
20*

5) No room in parent, create new indew node





Sorting

Algorithms and Costs

Why Sort?

- Eliminating duplicates (DISTINCT)
- Grouping for summarization (GROUP BY)
- Upcoming sort-merge join algorithm
    - Explicitly requested: ordering
    - For ordered outputs (ORDER BY)
    - First step in bulk-loading tree indexes

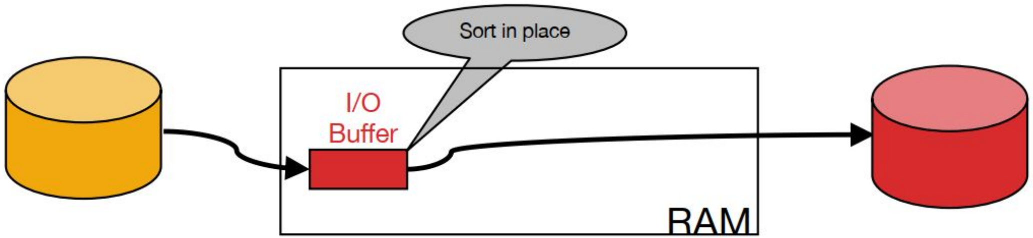
Problem: sort 100GB of data with 8GB of RAM.

- why not virtual memory?

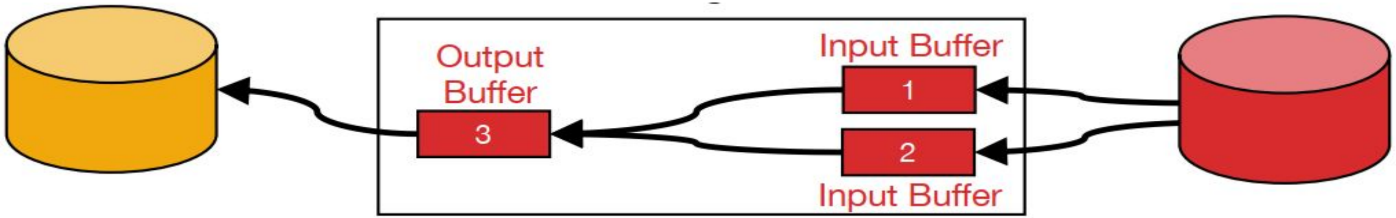
- Two themes
    1. Single-pass streaming data through RAM
    2. Divide (into RAM-sized chunks) and Conquer

Sorting: Two-Way

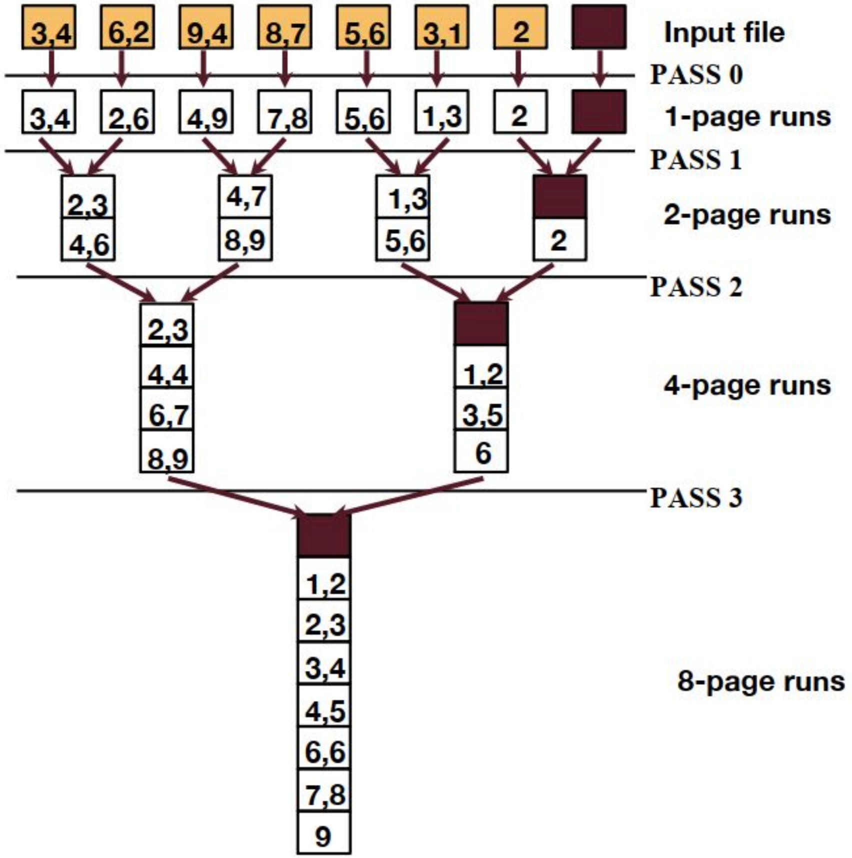
- Pass 0 (conquer a batch):
  - read a page, sort it, write it.
  - only one buffer page is used
  - a repeated “batch job”
  - results in N sorted blocks



- Pass 1, 2, 3, …, etc. (merge via streaming):
  - requires 3 buffer pages
  - merge pairs of runs into runs twice as long
  - a streaming algorithm
    - Drain/fill buffers as the data streams through them



Two-Way External Merge Sort



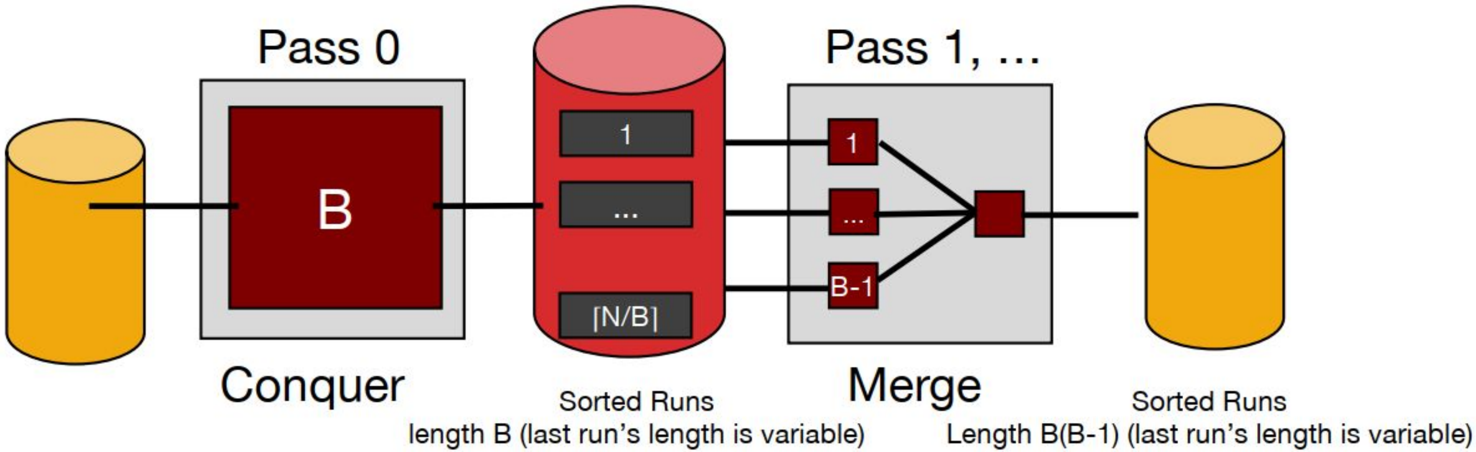
- Conquer and Merge:
  - sort subfiles and merge
- Each pass we read + write each page in file (2N)
- N pages in the file.
  - So, the number of passes is: $(\text{Log}_2 N) + 1$
- So total cost is: 2N((Log2 N) + 1)

General External Merge Sort

We got more than 3 buffer pages. How can we utilize them?

Big batches in pass 0, many streams in merge passes

- To sort a file with N pages using B buffer pages:
  - Pass 0: use B buffer pages. Produce (N/B) sorted runs of B pages each.
  - Pass 1, 2, …, etc.: merge B-1 runs at a time.



Cost of External Merge Sort

Number of passes: $1+(\log_{B-1}(N/B))$

Total I/Os = (I/Os per pass) * (# of passes) = $2*N*1+(\log_{B-1}(N/B))$

E.g., with 5 buffer pages, to sort 108 page file:

Pass 0: 108/5 = 22 sorted runs of 5 pages each (last run is only 3 pages).

Pass 1: 22/4 = 6 sorted runs of 20 pages each (last run is only 8 pages)

Pass 2: 6/4=2 sorted runs, 80 pages and 28 pages

Pass 3: Sorted file of 108 pages

Formula check: $1+\log_4 22 = 1+3$ => 4 passes √

of Passes of External Sort

(Total I/O is 2N * # of passes)

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Few runs can already sort large amounts of data!

Memory Requirement for External Sorting

How big of a table can we sort in exactly two passes?

- Each “sorted run” after Phase 0 is of size B
- Can merge up to B-1 sorted runs in Phase 1

nswer: B(B-1) ~ B2 data in two passes, using size B space

Sort X amount of data in about B =$\sqrt{x}$ buffer space (if we run only 2 passes)