File Organization

Cost Models

Intro to indexes

Alvin Cheung

Aditya Parameswaran

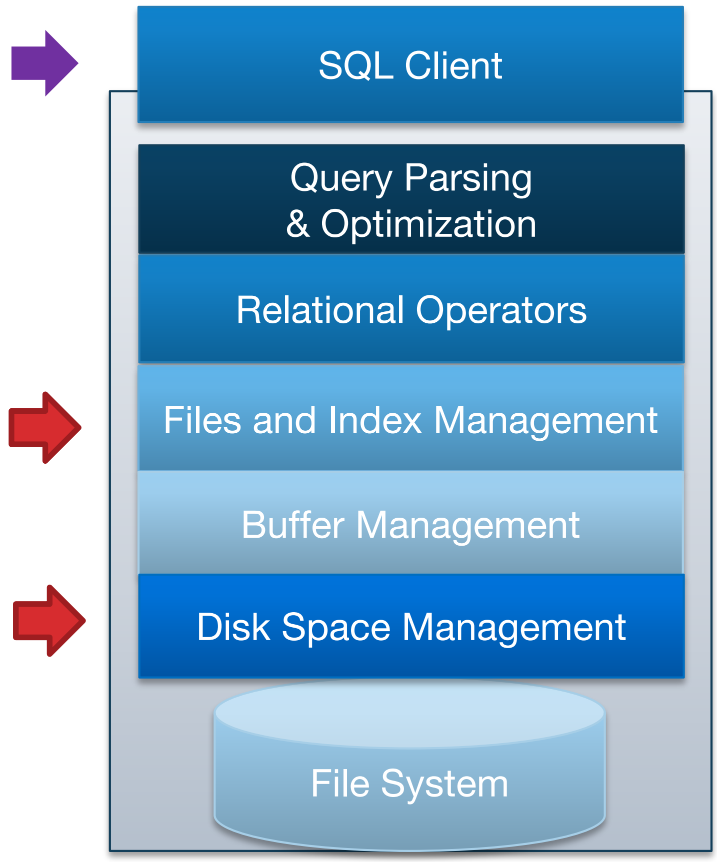
Reading: R & G Chapter 9

Architecture of a DBMS

Completed

And We'll Visit

You are Here



Recall: Heap Files

- Unordered collection of records
- Recall API for higher layers of the DBMS: only `READ` and `WRITE`!
- Today we'll ask: "How? At what cost?"
  - Insert/delete/modify record
  - Fetch a particular record by ***record id*** …
    - Record id is a pointer encoding pair of (**pageID, location** within page)
  - Scan all records
    - Possibly with some conditions on the records to be retrieved

Recall: Multiple File Organizations

- Many alternatives exist, each good in some situations and less so in others.
  - This is a theme in DB systems work!

- **Heap Files:** Suitable when typical access is a full scan of all records
- **Sorted Files:** Best for retrieval in order, or when a range of records is needed
- **Clustered Files & Indexes:** Group data into blocks to enable fast lookup *and* efficient modifications.
  - More on this soon …

- Want a way to quantitatively compare the cost of accessing data
  - Goal: given a query workload, find the best way to store data for optimal performance

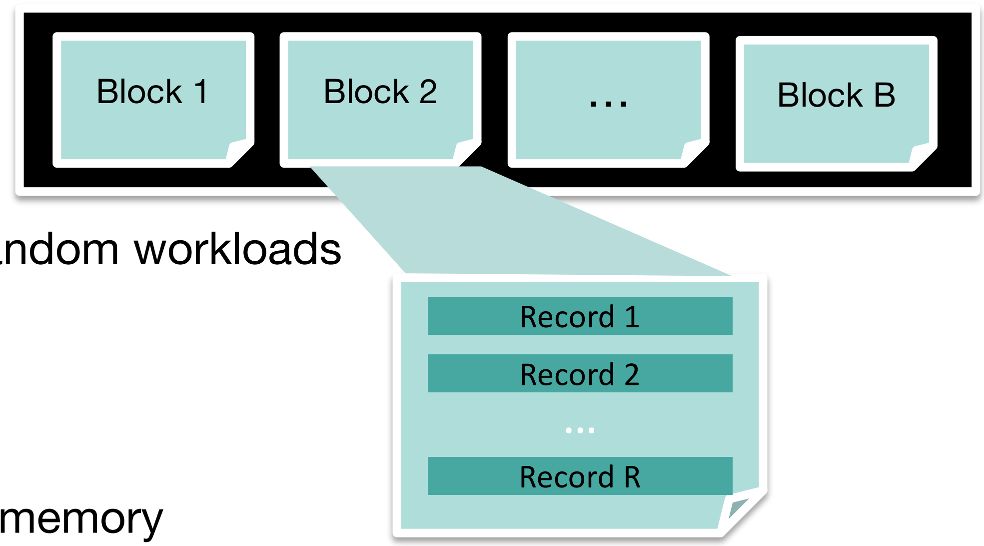
Cost Model Overview

- We want “big picture” estimates for data access
  - We’ll (overly) simplify performance models to provide insight, not to get perfect performance
  - Still, a bit of discipline:
    - **Clearly identify assumptions up front**
    - **Then estimate cost in a principled way**

- Foundation for query optimization
  - Can’t choose the fastest scheme without a speed estimate!

Cost Model for Analysis

- **B**: Number of data blocks in the file
- **R**: Number of records per block
- **D**: (Average) time to read/write disk block



- Focus: Average case analysis for uniform random workloads
- **Assumptions:** For now, we will ignore
  - Sequential vs Random I/O
  - Pre-fetching and cache eviction costs
  - Any CPU costs after fetching data into memory
  - Reading/writing of header pages for heap files
- Will assume data need to be brought into memory before operated on (and potentially written back to disk afterwards)
  - Both will cost I/O!
- Good enough to show the overall trends

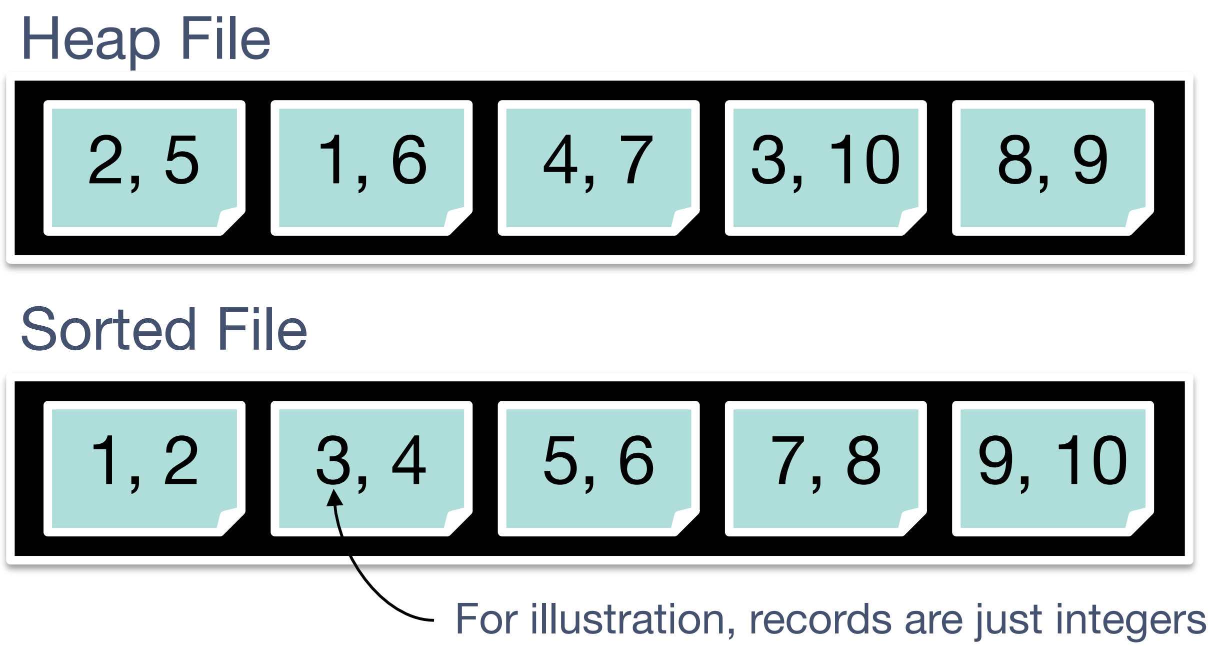
More Assumptions

- **Single record** insert and delete
- Equality selection – **exactly one match**
- For Heap Files:
  - Insert always **appends to end of file.**
- For Sorted Files:
  - **Packed**: Files compacted after deletions (i.e., no holes)
  - Sorted according to search key

Extra Challenge

- After understanding these slides …
  - You should question all these assumptions and rework
  - Good exercise to study for tests, and generate ideas

Heap Files & Sorted Files



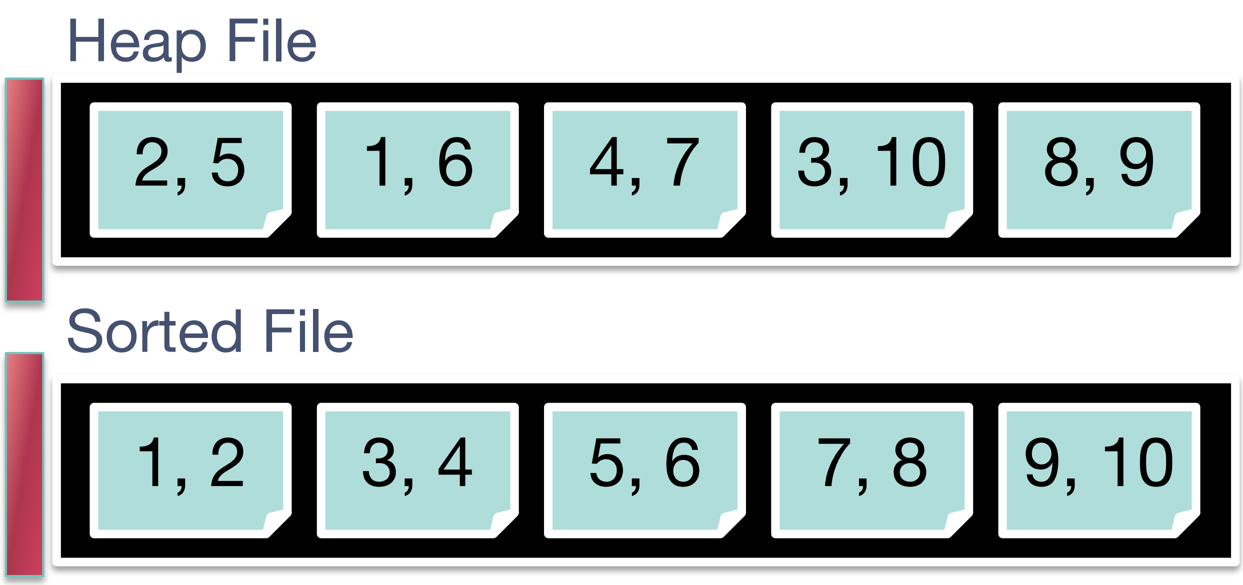
- **B:** Number of data blocks = 5
- **R:** Number of records per block = 2
- **D:** (Average) time to read/write disk block = 5ms

Cost of Operations: Scan?

	Heap File	Sorted File
Scan all records		
Equality Search		
Range Search		
Insert		
Delete		

- **B:** Number of data blocks = 5
- **R:** Number of records per block = 2
- **D:** (Average) time to read/write disk block = 5ms

Scan All Records



- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block
- **Pages touched: ?**
- **Time to read the record: ?**

Cost of Operations: Scan Cost

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search		
Range Search		
Insert		
Delete		

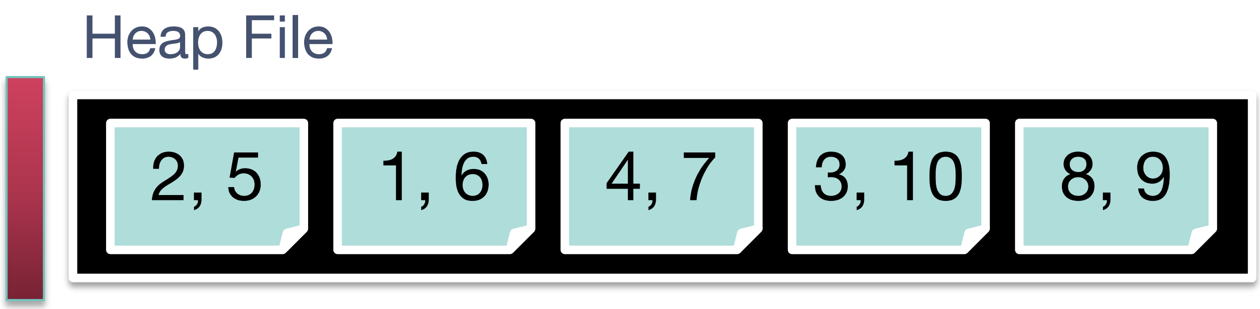
- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Cost of Operations: Equality Search?

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search		
Range Search		
Insert		
Delete		

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

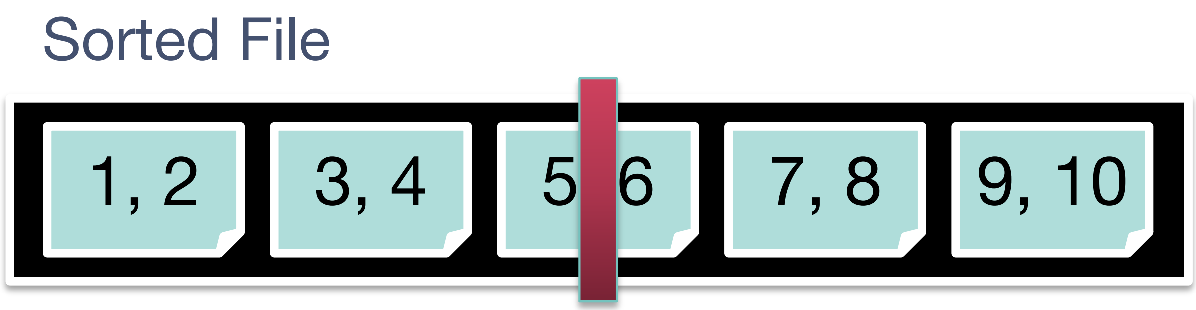
Find Record 8: Heap File



- **P(i):** Probability that key is on page ***i*** is **1/B**
- **T(i):** Number of pages touched if key on page ***i*** is **i**
- Therefore the expected number of pages touched is:

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

Find Record 8: Sorted File

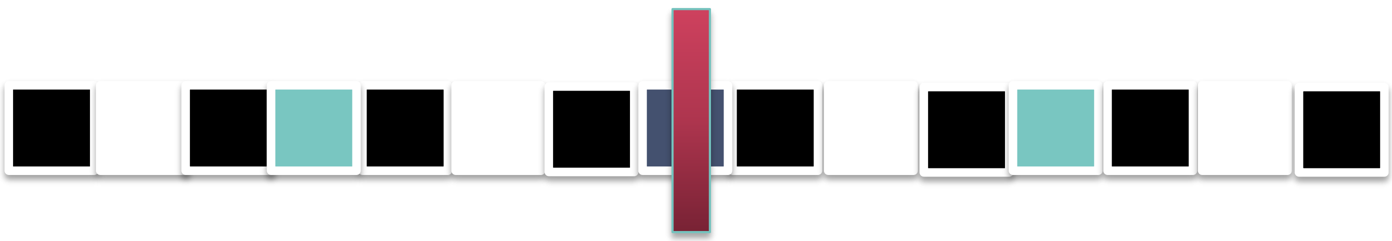


- **Worst-case:** Pages touched in binary search
  - **$\log_2 B$**
- **Average-case:** Pages touched in binary search
  - **$\log_2 B$?**

Average Case Binary Search

B: The number of data blocks

Expected Number of Reads: 1 (1 / B) + 2 (2 / B) + 3 (4 / B) + 4 (8 / B)



Average Case Binary Search cont

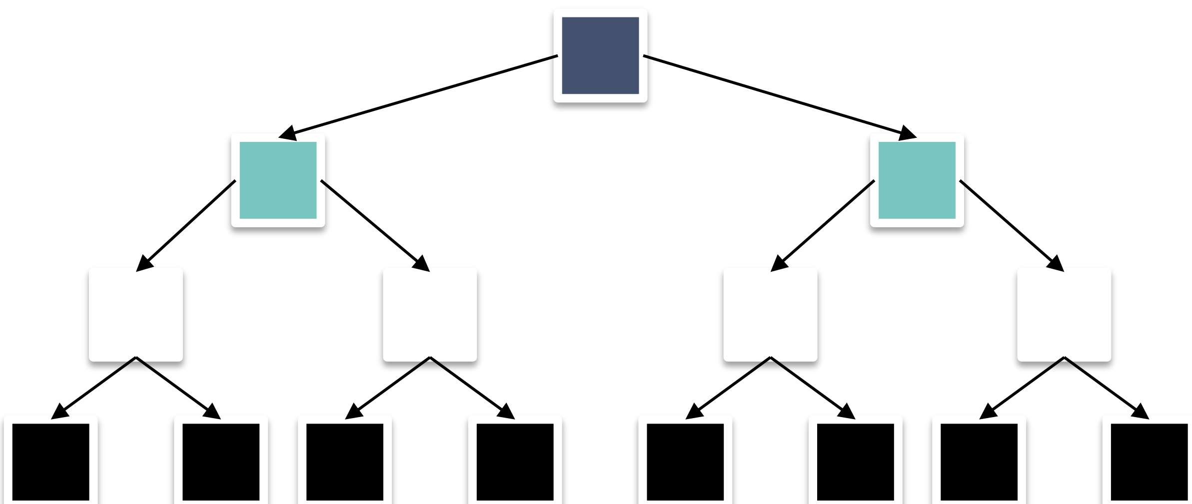
Expected Number of Reads: 1 (1 / B) + 2 (2 / B) + 3 (4 / B) + 4 (8 / B)

1 IO

2 IOs

3 IOs

4 IOs



$$\sum_{i=1}^{\log_2 B} i\frac{2^{i-1}}{B} = \frac{1}{B}\sum_{i=1}^{\log_2 B} i2^{i-1} = \log_2 B - \frac{B-1}{B}$$

Cost of Operations: Equation Search Cost

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search	0.5*B*D	$(\log_2 B)$*D
Range Search		
Insert		
Delete		

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Cost of Operations: Range Search?

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search	0.5*B*D	$(\log_2 B)$*D
Range Search		
Insert		
Delete		

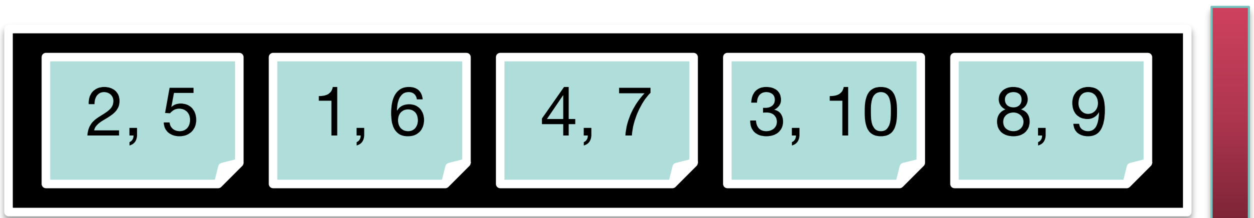
- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Find Records Between 7 and 9: Heap File

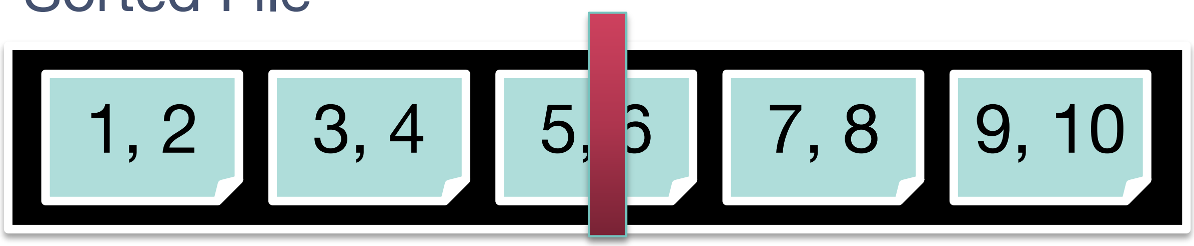
- Always touch all blocks. Why?

Find Records Between 7 and 9: Comparison

Heap File



Sorted File



- Find beginning of range
- Scan right

Cost of Operations: Range Search Cost

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search	0.5*B*D	$(\log_2 B)$*D
Range Search	B*D	$((\log_2 B)$+pages)* D
Insert		
Delete		

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Cost of Operations: Insert?

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search	0.5*B*D	$(\log_2 B)$*D
Range Search	B*D	$((\log_2 B)$+pages)* D
Insert		
Delete		

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Insert 4.5: Heap File

- Stick at end of file
- Cost = 2*D
- Why 2?

Insert 4.5: Heap Vs Sorted File

Heap File

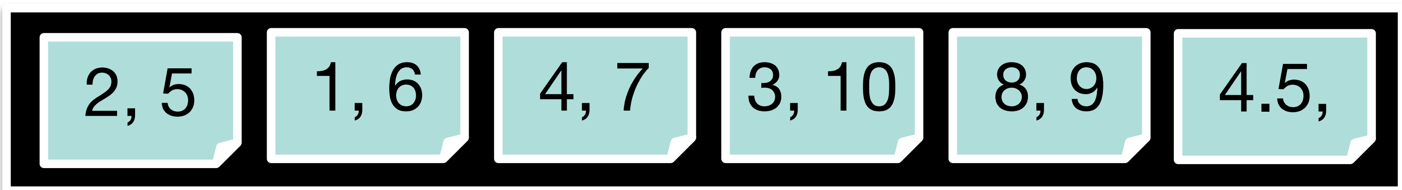
- Read last page, append, write. Total cost = 2*D

Sorted File

- Find location for record. Cost = $(\log_2 B) * D$

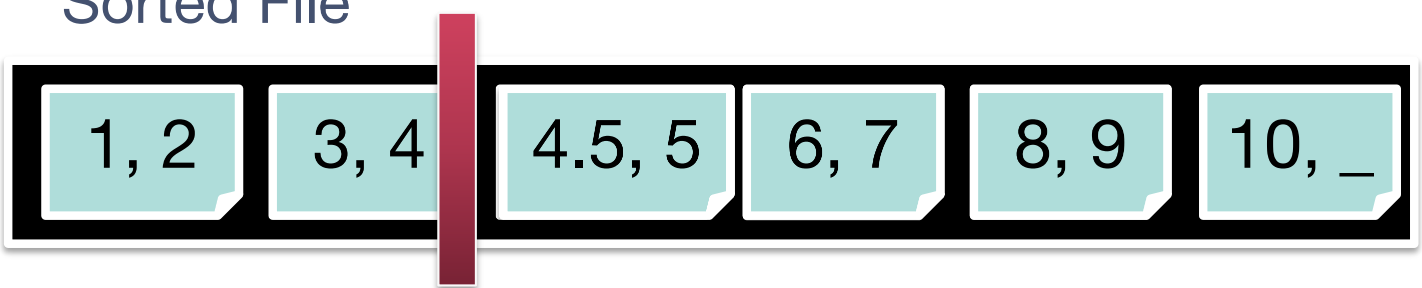
Insert 4.5: Heap Vs Sorted Pt 2

Heap File



- Read last page, append, write. Total cost = 2*D

Sorted File



- Find location for record. Cost = $(\log_2 B) * D$
- Insert and shift rest of file. Cost = $(B/2) * D * 2 = B * D$
- Total: find cost + insert and shift cost = $(\log_2 B) * D + B * D = ((\log_2 B) + B) * D$

Cost of Operations: Insert Cost

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search	0.5*B*D	$(\log_2 B)$*D
Range Search	B*D	$((\log_2 B)$+pages)* D
Insert	2*D	$((\log_2 B)$+B)*D
Delete		

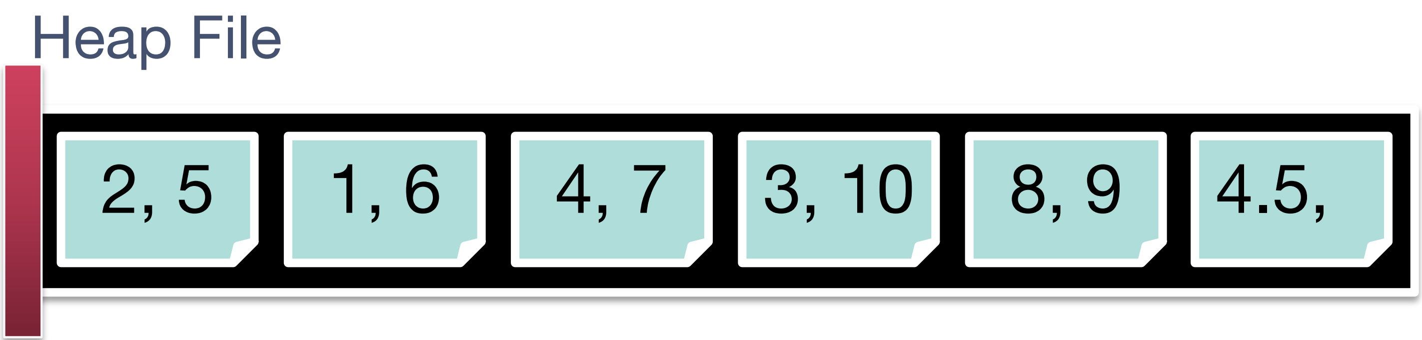
- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Cost of Operations: Delete?

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search	0.5*B*D	$(\log_2 B)$*D
Range Search	B*D	$((\log_2 B)$+pages)* D
Insert	2*D	$((\log_2 B)$+B)*D
Delete		

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

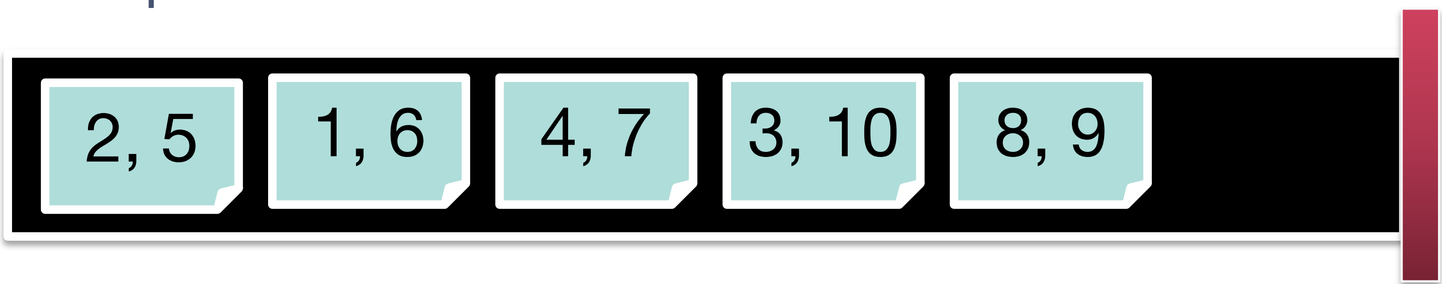
Delete 4.5: Heap File



- Average case to find the record: **B/2 reads**
- Delete record from page
- Cost = (B/2 + 1) * D
  - Why + 1?

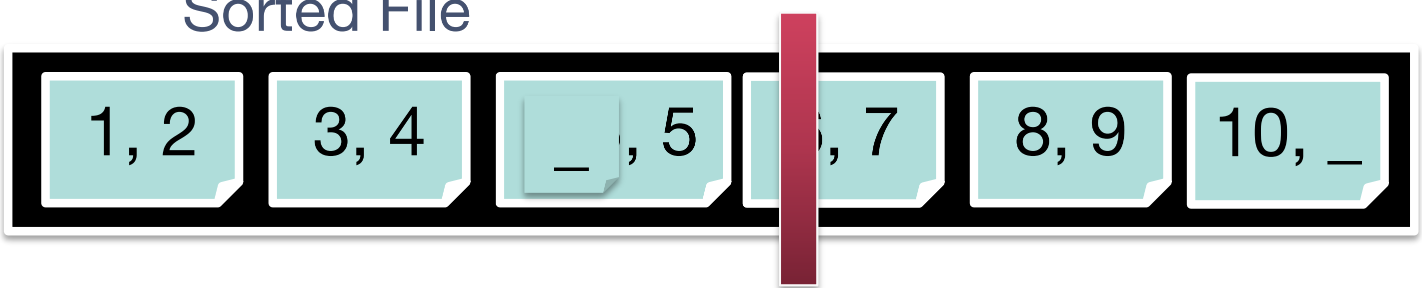
Delete 4.5: Heap File Vs Sorted File

Heap File



- Average case runtime: **(B/2+1) * D**

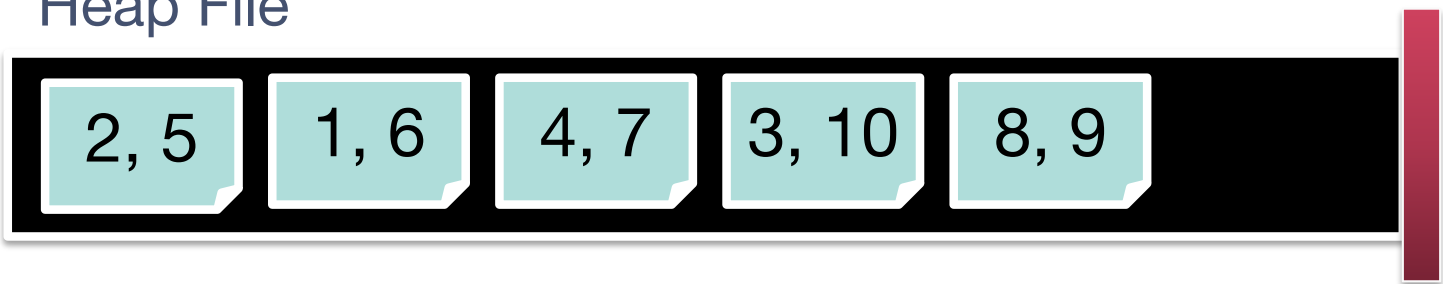
Sorted File



- Find location for record. Cost = $\mathbf{log_2 B}$
- Delete record in page → Gap

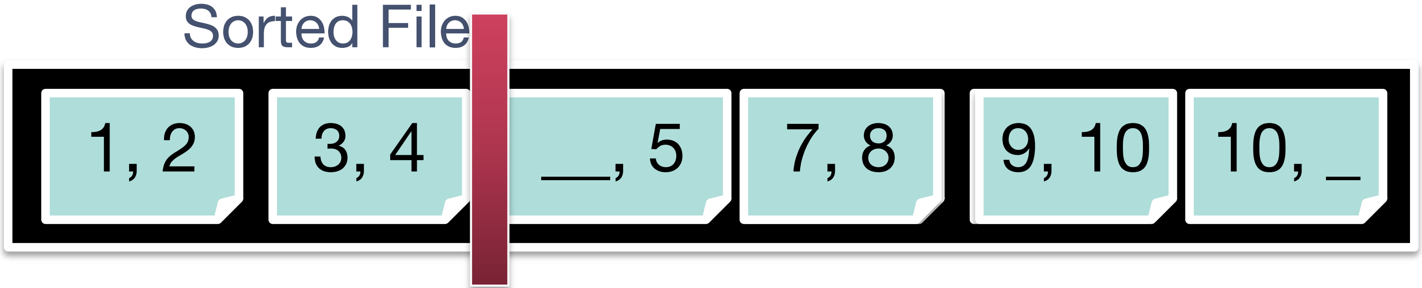
Delete 4.5: Heap File Vs Sorted File Pt 2

Heap File



- Average case runtime: (B/2+1) * D

Sorted File



- Find location for record. Cost = $\log_2 B$
- Read the rest into memory, shift by 1 record, and write back: 2 * (B/2) = B
- Total: find cost + delete and shift cost = $(\log_2 B) * D + B * D = ((\log_2 B) + B) * D$

Complete Cost of Operations

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search	0.5*B*D	$(\log_2 B)$*D
Range Search	B*D	$((\log_2 B)$+pages)*D
Insert	2*D	$((\log_2 B)$+B)*D
Delete	(0.5*B+1)*D	$((\log_2 B)$+B)*D

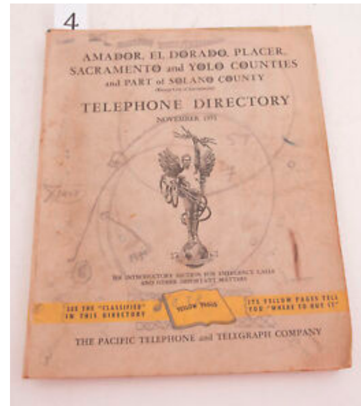
- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block

Complete Cost of Operations Pt 2

	Heap File	Sorted File
Scan all records	B*D	B*D
Equality Search	0.5*B*D	$(\log_2 B)$*D
Range Search	B*D	$((\log_2 B)$+pages)* D
Insert	2*D	$((\log_2 B)$+B)*D
Delete	(0.5*B+1)*D	$((\log_2 B)$+B)*D

- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block
- Can we do better?
  - Indexes!

Wouldn’t it be nice…



- …if we could look things up by value?
- 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 memory:
  - Search trees (Binary, AVL, Red-Black, …)
  - Hash tables
  - Recall cs61b!

- But we need disk-based data structures
  - “paginated”: made up of disk pages!

We've seen this before

```
CREATE TABLE Sailors (
      sid INTEGER,
      sname CHAR(20),
      rating INTEGER,
      age FLOAT,
      PRIMARY KEY (sid));
```

sid	**sname**	**rating**	**age**
1	Fred	7	22
2	Jim	2	39
3	Nancy	8	27

- How to store data in `Sailors`?
  - Use an index!

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, ...