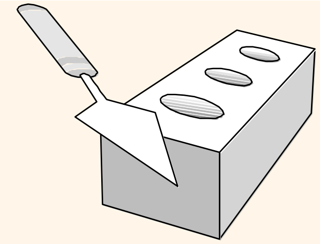
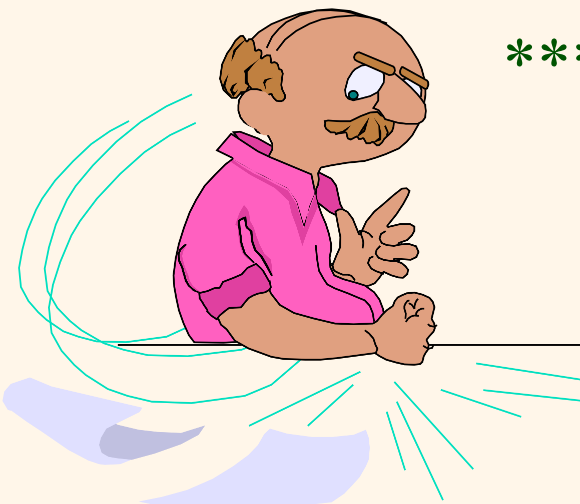


Introduction to Data Management

**** The "Flipped" Edition ****



Lecture #18 (Storage & Indexing I)

Instructor: Mike Carey
mjcarey@ics.uci.edu

Announcements

❖ Roadmap check:

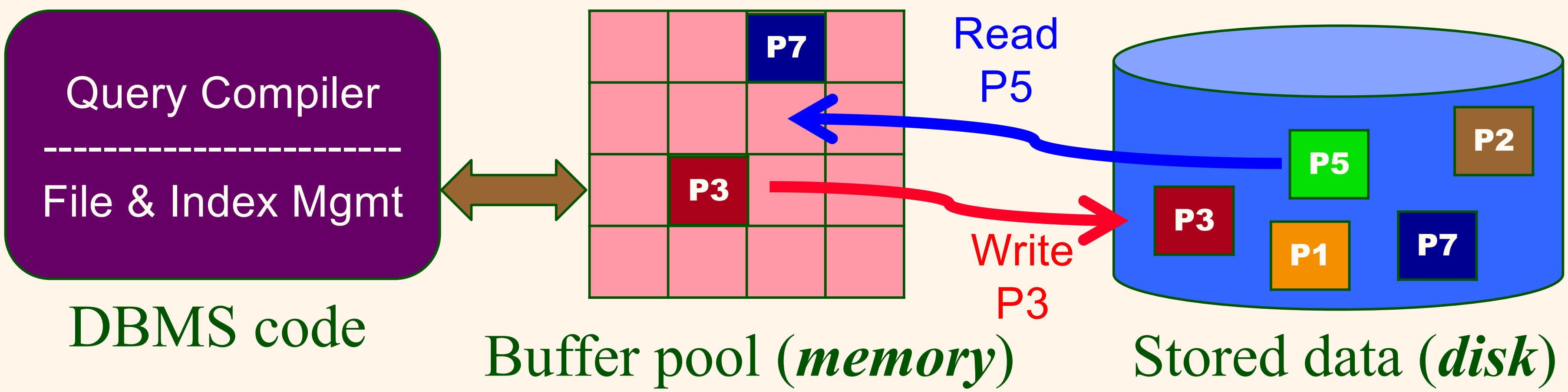
Midterm Exam 2	***Mon, Nov 15*** (during lecture time)
Storage	Ch. 12.1-12.4, 12.6-12.7
Indexing	Ch. 14.1-14.4, 14.5
Physical DB Design	Ch. 14.6-14.7, 15.1-15.3, 15.5.3
Semistructured Data Management (*a.k.a.* NoSQL)	Ch. 8.1, AsterixDB SQL++ Primer, Couchbase SQL++ Book
Data Science 1: Advanced SQL Analytics	Ch. 5.5, 11.3
Data Science 2: Notebooks, Dataframes, and Python/Pandas	Lecture notes and Jupyter notebook
Basics of Transactions	Ch. 4.3, Ch. 17
Endterm Exam	***Fri, Dec 3*** (during lecture time)

❖ SQL HW assignments (in "*Friday 6PM mode*")
  - HW #5 (Basic SQL) is due on Friday
  - HW #6 (Advanced SQL) will come out on Friday

❖ Midterm #2 (relational languages) is *1.5 weeks away (!)*
  - Same in-class Gradescope + cheat sheet logistics as Midterm #1

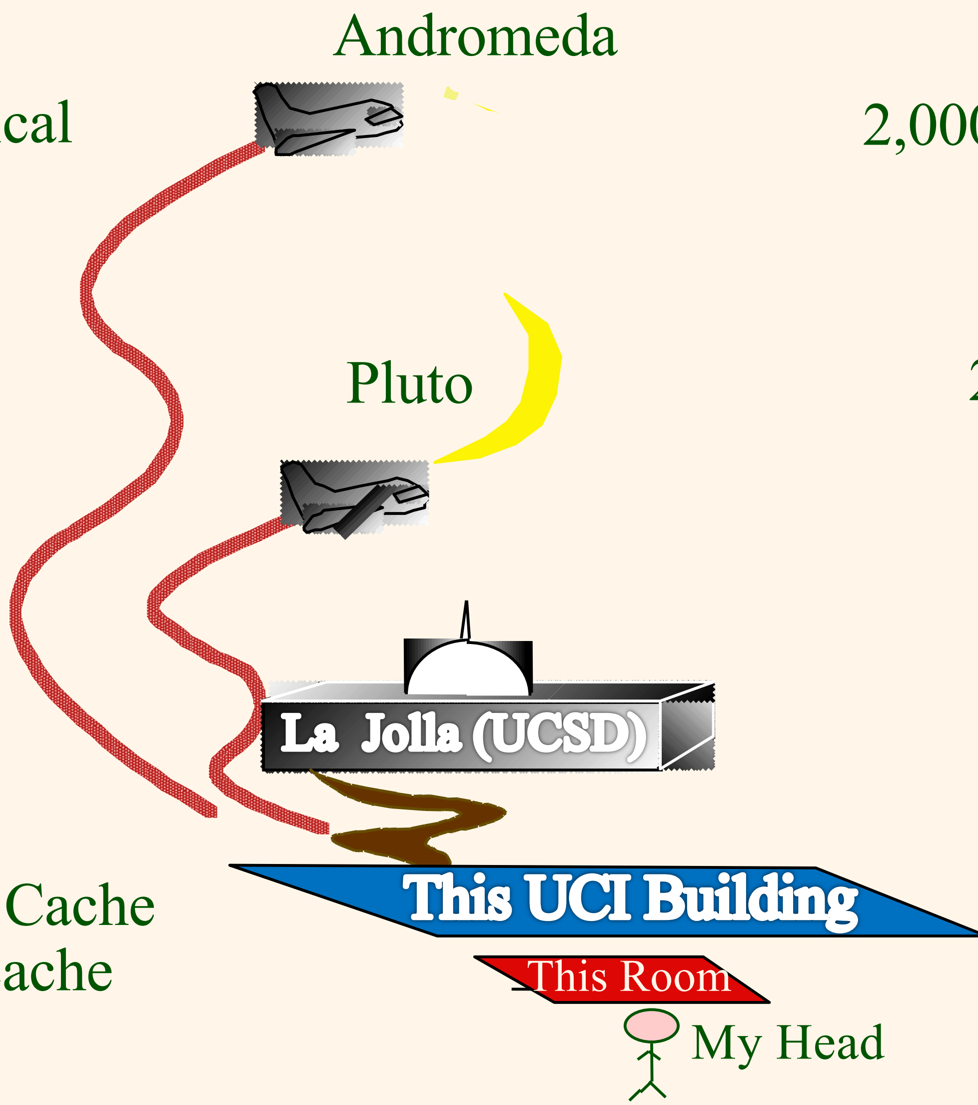
Disks and Files

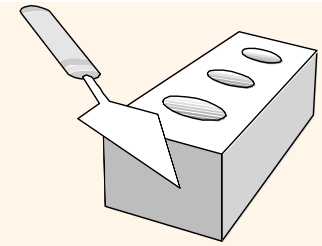
- DBMSs store data on *secondary storage*.
- This has major implications for DBMS design!
  - READ: xfer data from ***disk*** to ***memory*** (RAM).
  - WRITE: xfer data from RAM to disk (HDD or SSD).
  - Both are high-cost operations, relative to in-memory operations, so must be considered carefully!



Storage Hierarchy & Latency (Jim Gray in the mid 1990's): How Far Away is my Data?

10^9	Tape /Optical Robot	2,000 years
10^6	Disk	2 years
100	Memory	1.5 hr
10	On Board Cache	10 min
2	On Chip Cache	
1	Registers	1 min



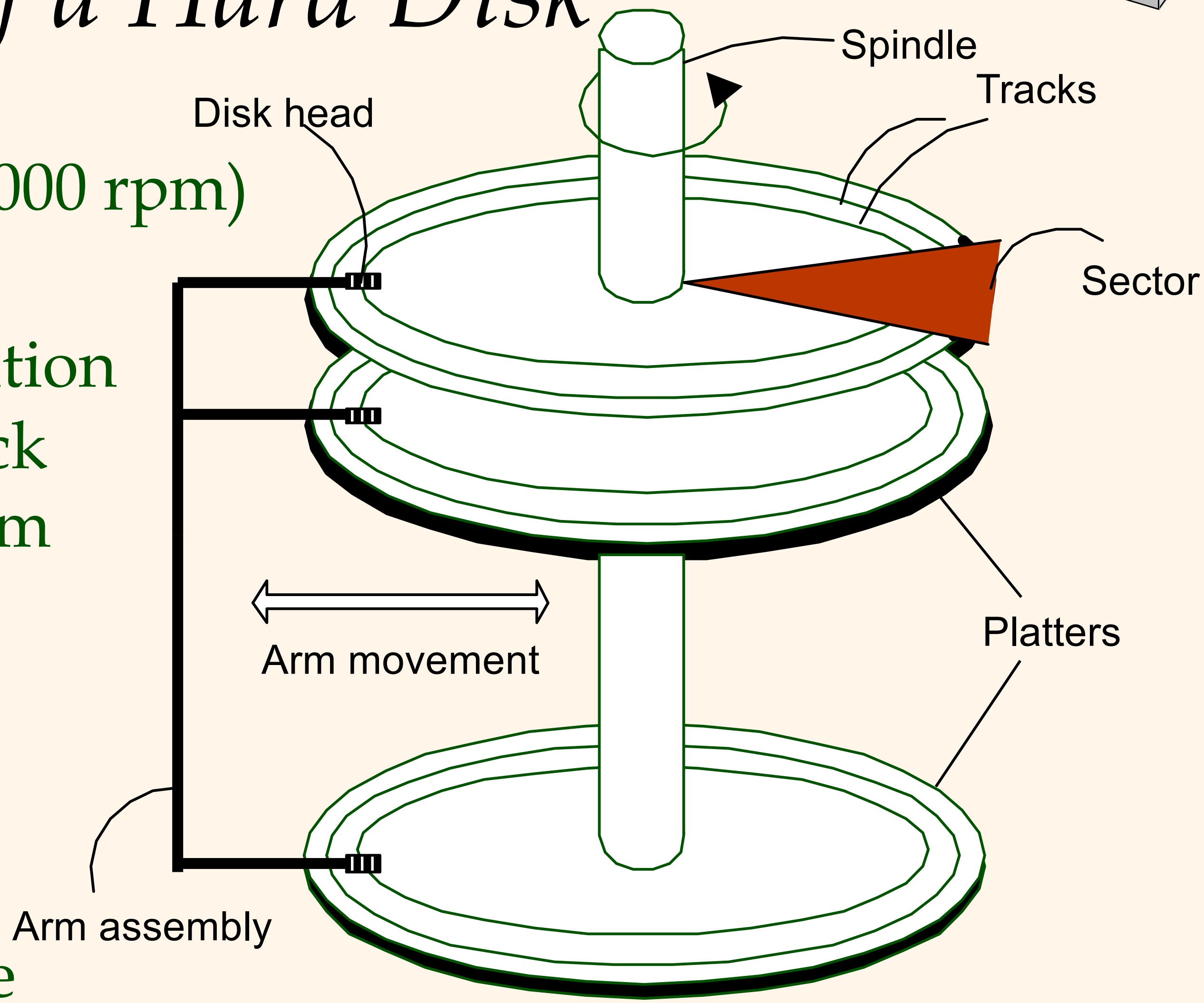


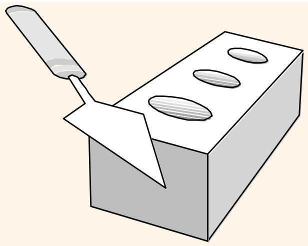
Why Not Store Data in Main Memory?

- *Main memory (RAM) costs too much! Roughly:*
  - RAM: $22/GB [Dell @ 05/2019]
  - SSD: $1/GB (22x cheaper than RAM)
  - Disk: $0.16/GB (138x cheaper than RAM, 5.5x *vs.* SSD)
- *Main memory is **volatile**.* We want our data to be saved between runs. (Obviously…!)
- Your typical (basic) storage hierarchy:
  - Main memory (RAM) for currently used data
  - Disk (HDD, or increasingly SSD) for the main database (secondary storage)
  - Tapes for archiving the data (tertiary storage)

Components of a Hard Disk

- The platters spin (10,000 rpm)
- The arm assembly is moved in or out to position a head on a desired track Tracks under heads form a *cylinder* (imaginary!)
- Only one head reads/writes at any one time.
- *Block size* is a multiple of *sector size* (which is fixed)



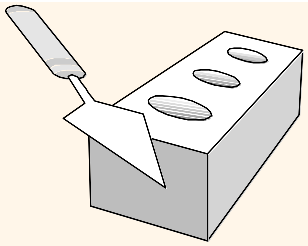


Accessing a Disk Page

- Time to access (read/write) a disk block:
  - *Seek time* (moving arms to position disk head on track)
  - *Rotational delay* (waiting for block to rotate under head)
  - *Transfer time* (actually moving data to/from disk surface)
- Seek time and rotational delay dominate!
  - Seek time varies from about 1 to 20msec
  - Rotational delay varies from 0 to 10msec
  - Transfer rate is < 1 msec per 4KB page
  - Key to I/O cost: Reduce seek/rotation delays!

  → **Bottom line: *Random vs. sequential I/O***

What About SSD Storage?

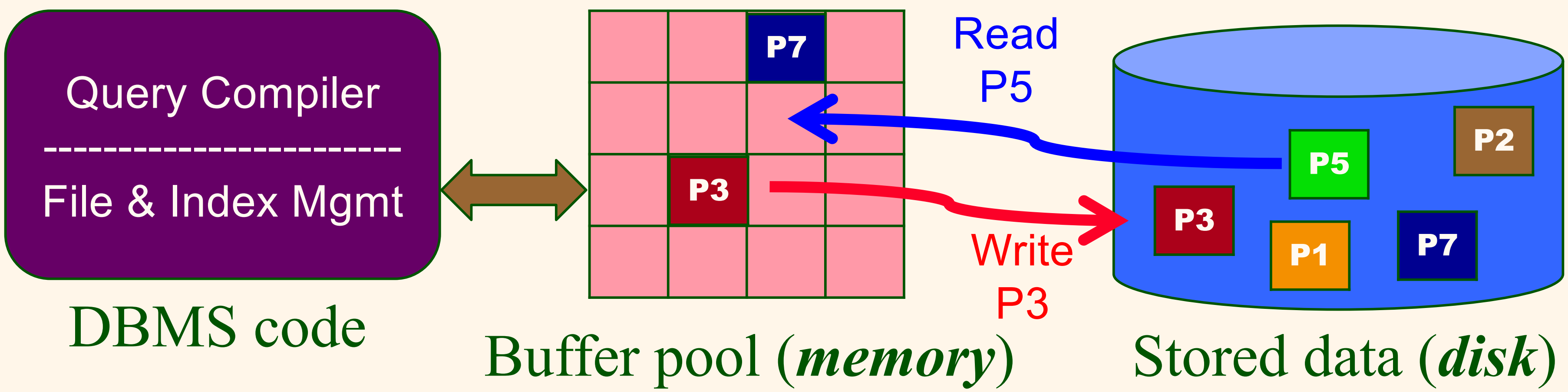


- NAND Flash: ***block-oriented*** interface like HDD
  - *Random reads:* 20-100 usec (vs. 5-10 msec for HDD)
  - *Transfer rate:* 500-3000 GB/sec (vs. 200 for HDD)
  - *Read performance:* 10,000+ IOPS (random reads/sec), but up to 10x with parallel accesses (unlike HDD)
- Writes are more complicated than HDD
  - *Random writes:* ~100 usec (first time in empty spot)
  - *Write performance:* 10,000+ IOPS (random reads/sec)
  - Limited life, wear leveling, flash translation layer, ...

  → **Bottom line: *No seeks, but still block-oriented***

So: *Disks and Files (Reminder!)*

- DBMSs store persistent data on *secondary storage*.
- This has major implications for DBMS design!
  - READ: xfer data block from ***disk*** to ***memory*** (RAM).
  - WRITE: xfer data block from RAM to disk.
  - Both are ***high-cost*** operations, relative to in-memory operations, so must be considered carefully!



Ex: Emp(<u>eid</u>, ename, sal, deptid)

P1	1	111	Smith	3K	1
	2	222	Lee	100K	3
	3	333	Carey	80K	1
	4	444	Smith	12K	7
P2	1	555	Smith	18K	3
	2	666	Jones	90K	5
	3	777	Smith	23K	4
	4	888	Krishan	60K	8
	...				
P10000	1	...	...	...	...
	2	...	...	...	...
	3	...	...	...	...
	4	9999999	Smith	18K	11

← Record id (RID) is (P2,3)

Underlying *Emp* file pages

Processing a Query

- Suppose someone asks a simple SQL query:

  SELECT ∗ **FROM** Emp **WHERE** eid = 12345;

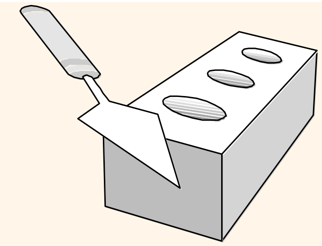
- Some processing options would include:
  - *Option 1*: Sequentially scan the data file (and stop, if we know eid is a key) → 5000 page reads (*avg.*)
  - *Option 2*: Binary search the data file (and stop, if we know eid is a key) → $\log_2(10{,}000) \approx 15$ page reads (*avg.*)
  - Even though Option 2 is ≈ **300x** faster, we'd like to do ***even better*** (especially for large data sets!)

Indexing is the Answer!

Key value *k* or key range (k_1, k_2) → **Index on *k*** → **I(*k*) *or* I(k_1), ..., I(k_2)**

777 → **(P2,3)**

- ***Index*** maps from ***keys*** to associated ***info*** **I**
  - I(k) can be the ***data record itself*** with key k, or
  - I(k) can be the *RID* of the data record with key k, or
  - I(k) can be a *list of RIDs* of data records with key k!
  - Alternatively, we could map from data field values to the *PK value(s)* of the associated record(s) – SQL Server and AsterixDB do this, for example.



Indexes

- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Any subset of the fields of a relation can serve as the search key for an index on the relation.
  - *Search key* is **not** the same as a *"key"* (i.e., it's not the primary key, it's just a field we're very interested in).
- An index contains a collection of *data entries*, and it supports efficient retrieval of *all* data entries ***k**** with a given key value *k*.
  - Given a **data** entry *k**, we can find **1st** actual **record** with key *k* with ~1 more disk I/O. (Details soon…)

Ex: *Emp (eid, ename, sal, deptid)*

- One simple approach would be to have one more file, sorted *on k*, for each *k* that we want to index
  - *Hundreds* of (key, RID) entries will fit on a *single page*
  - Index is thus *much* smaller than the data file
  - Less data (*fewer reads!*) to search to locate the RIDs of interest

eid (RID)

eid	RID
111	(P1,1)
222	(P1,2)
333	(P1,3)
444	(P1,4)
555	(P2,1)
666	(P2,2)
...	...

...

sal ~~(RID)~~

sal	RID
3K	(P1,1)
12K	(P1,4)
18K	(P2,1)
18K	(P10000,1)
23K	(P2,3)
60K	(P2,4)
...	...

...

Note:
*Can have multiple **unclustered** indexes!*

Indexed Tables Under the Hood

```
SELECT *
FROM Emp
WHERE eid = 555;
```

eid	RID
111	(P1,1)
222	(P1,2)
333	(P1,3)
444	(P1,4)
555	(P2,1)
666	(P2,2)
…	…

…

```
SELECT *
FROM Emp
WHERE sal >= 10K
    AND sal < 25K;
```

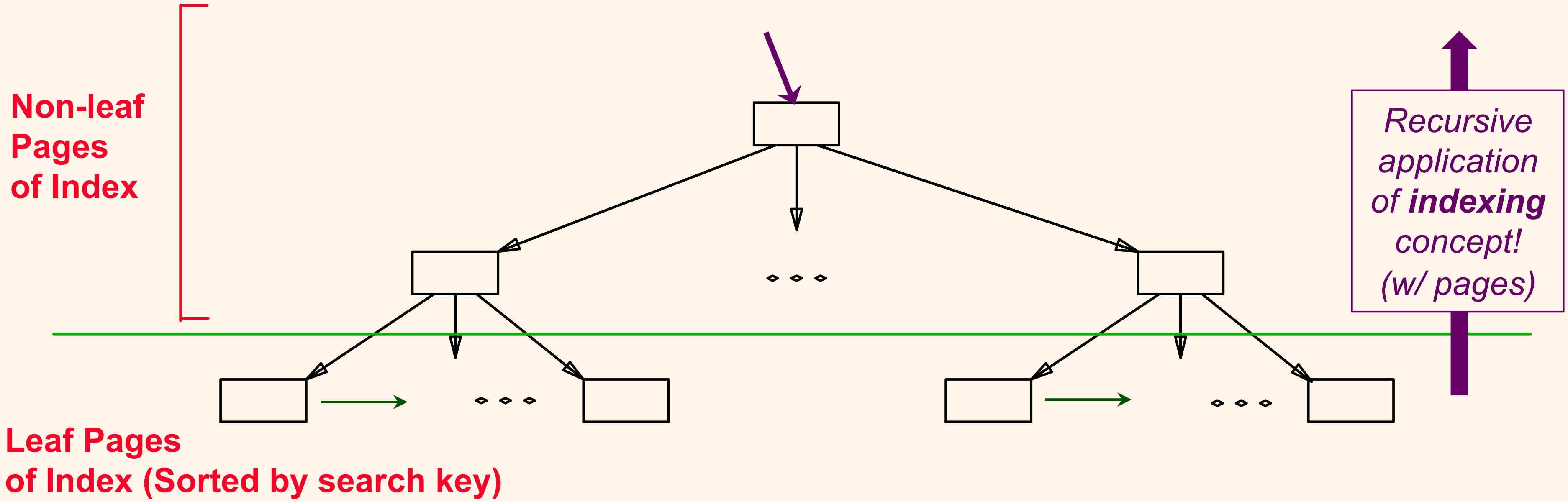
sal	RID
3K	(P1,1)
12K	(P1,4)
18K	(P2,1)
18K	(P10000,1)
23K	(P2,3)
60K	(P2,4)
…	…

…

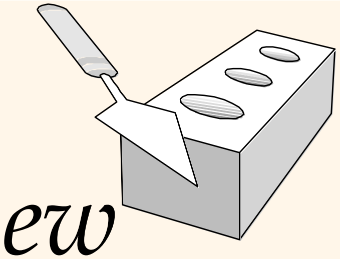
Emp(eid, ename, sal, deptid)

		eid	ename	sal	deptid
P_1	1	111	Smith	3K	1
	2	222	Lee	100K	3
	3	333	Carey	80K	1
	4	444	Smith	12K	7
P_2	1	555	Smith	18K	3
	2	666	Jones	90K	5
	3	777	Smith	23K	4
	4	888	Krishan	60K	8
		…			
P_{10000}	1	…	…	…	…
	2	…	…	…	…
	3	…	…	…	…
	4	9999999	Smith	18K	11

Even Better: Tree Indexes!



- Leaf pages contain *data entries,* and are chained
- Non-leaf pages have *index entries;* role is to guide searches
- Query processing steps become:
  1. Choose a good index to use (if one is available)
  2. Search the index to determine the interesting RID(s)
  3. Use the RID(s) to fetch the corresponding record(s)

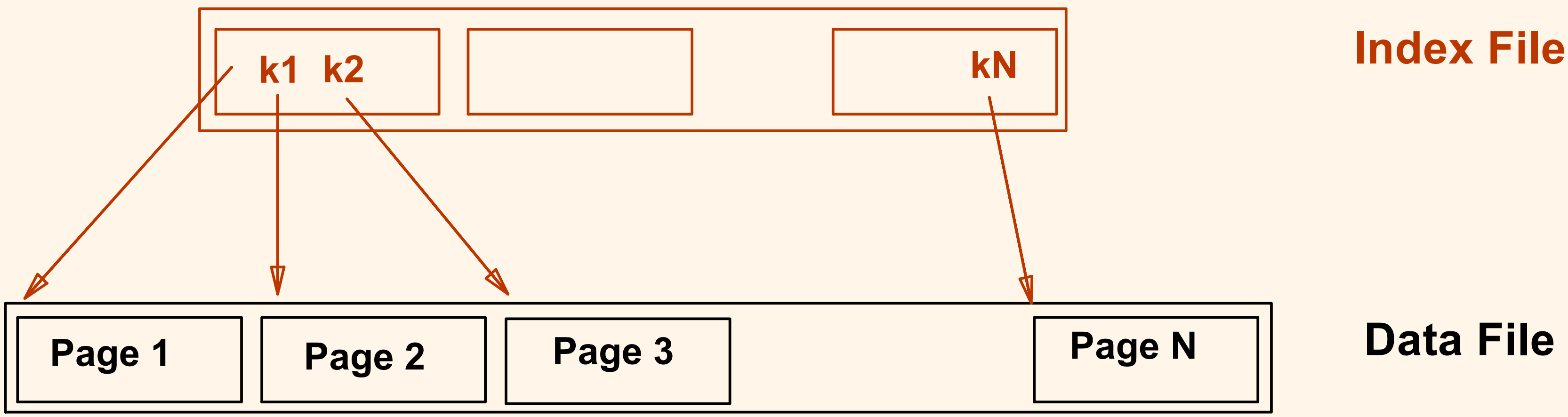


Tree-*Structured Indexes: Over(re)view*

- *As for **any** index, 3 alternatives for data entries **k***:*
  - Data record with key value **k**
  - <**k**, RID of data record with search key value **k**>
  - <**k**, list of RIDs of data records with search key **k**>
- This data entry choice is orthogonal to the *indexing technique* used to locate data entries **k***.
- Tree-structured indexing techniques support both *range searches* and *equality searches*.
- *ISAM*: static structure; *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.

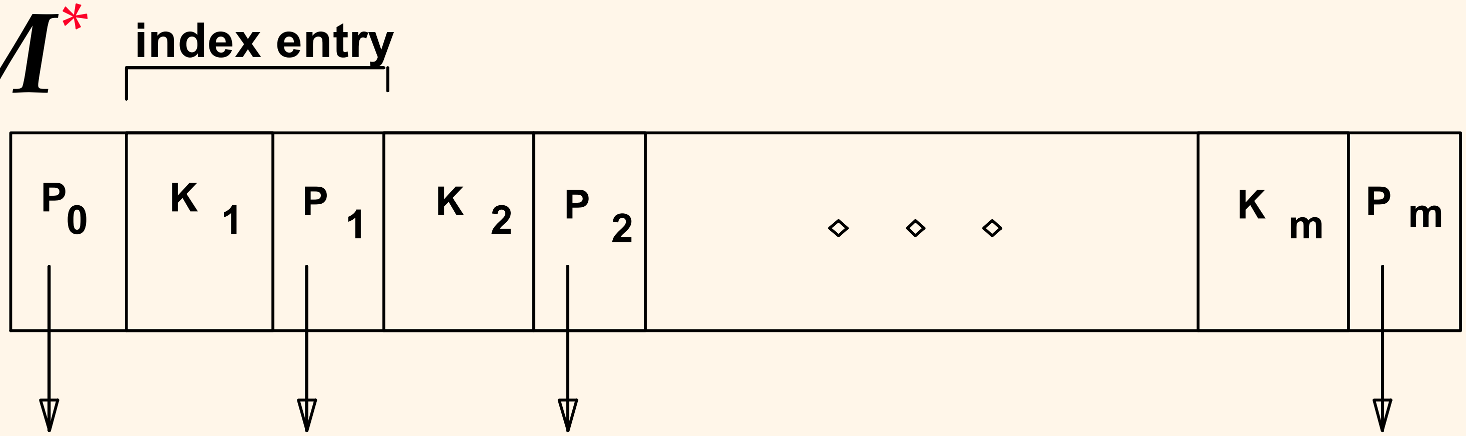
Range Searches (Review)

- *"Find all students with gpa > 3.0"*
- If records are in a sorted file, do binary search to find first such student, then scan to find others.
  - Cost of file binary search can be quite high.
- Simple idea to do better: add an "index" file.

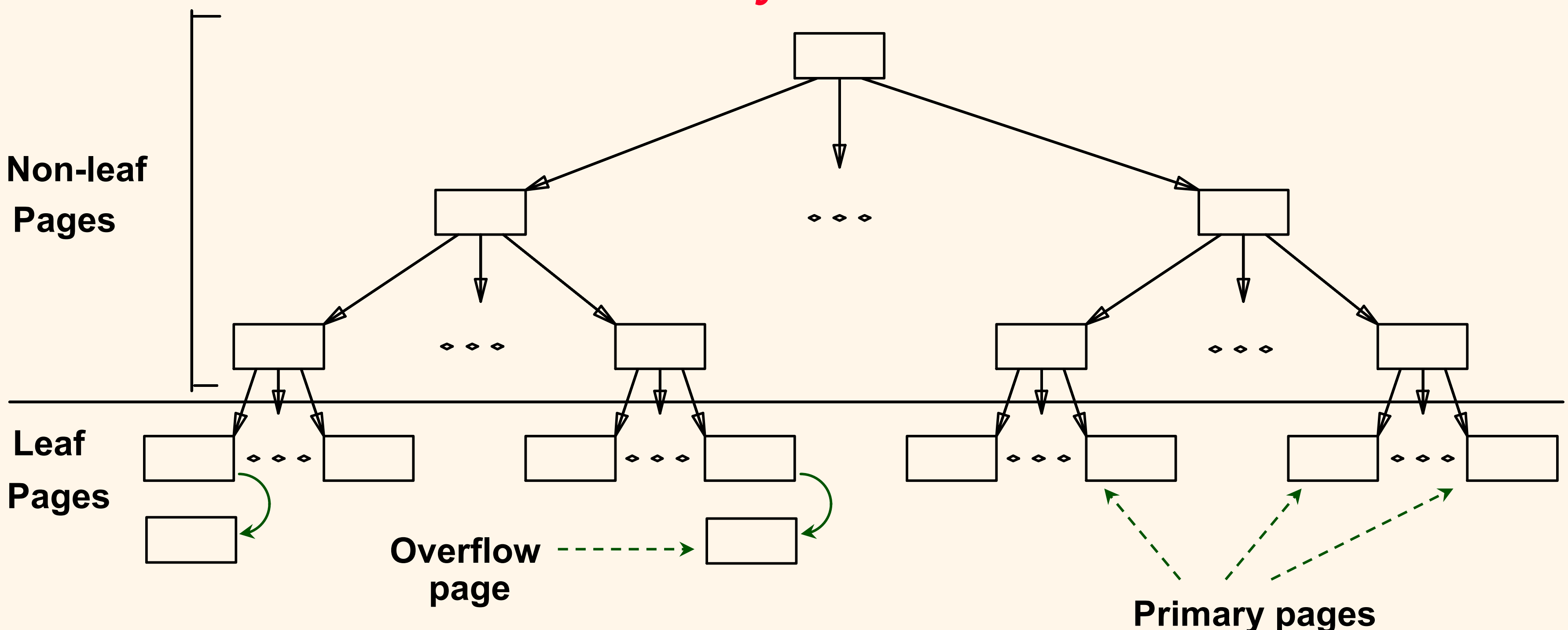


☛ *Can now binary search the (smaller) index file!*

ISAM*



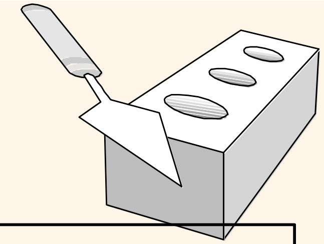
- Index file may still be quite large. But, we can apply the same idea **recursively** to address that!

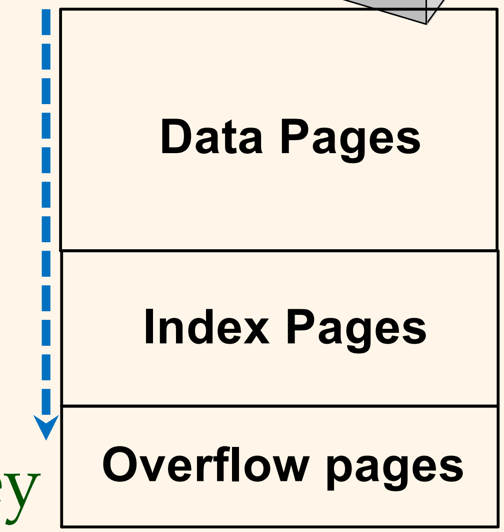


☛ *Leaf pages contain the data entries.*

**ISAM: Indexed Sequential Access Method*

ISAM Operation(s)

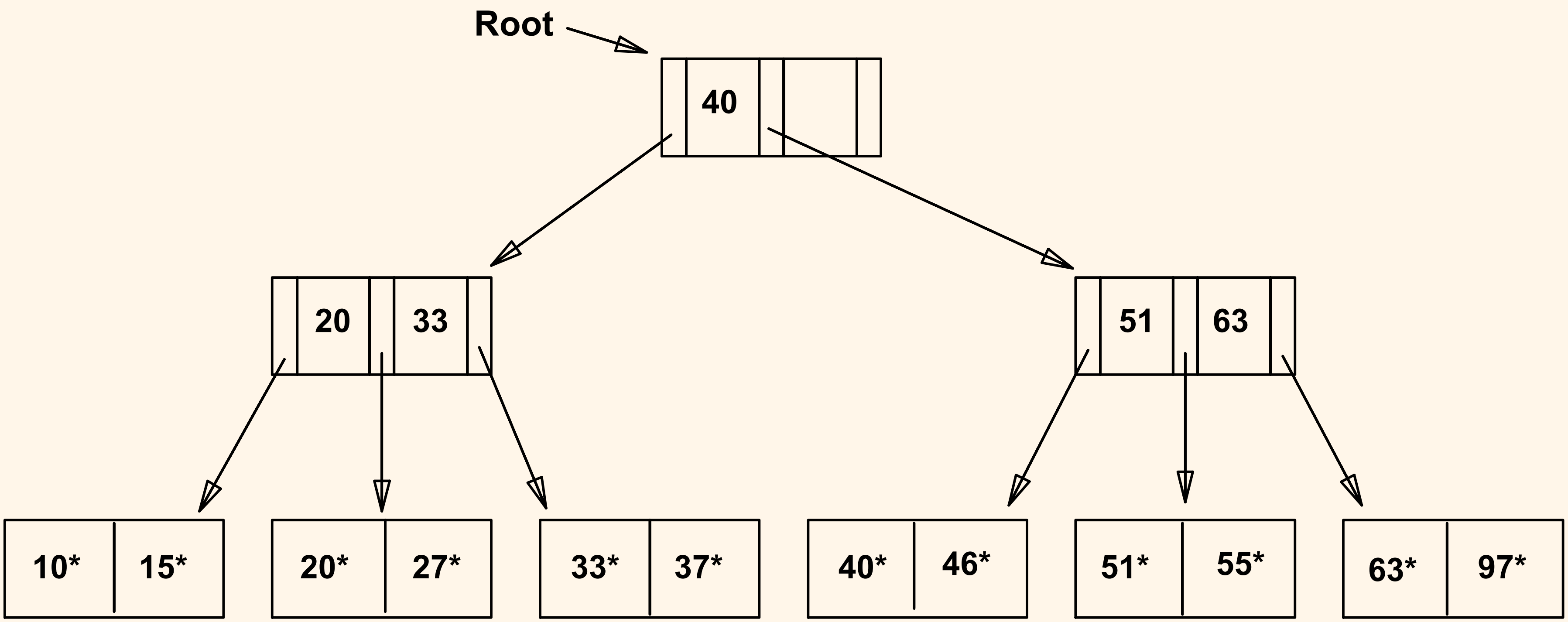




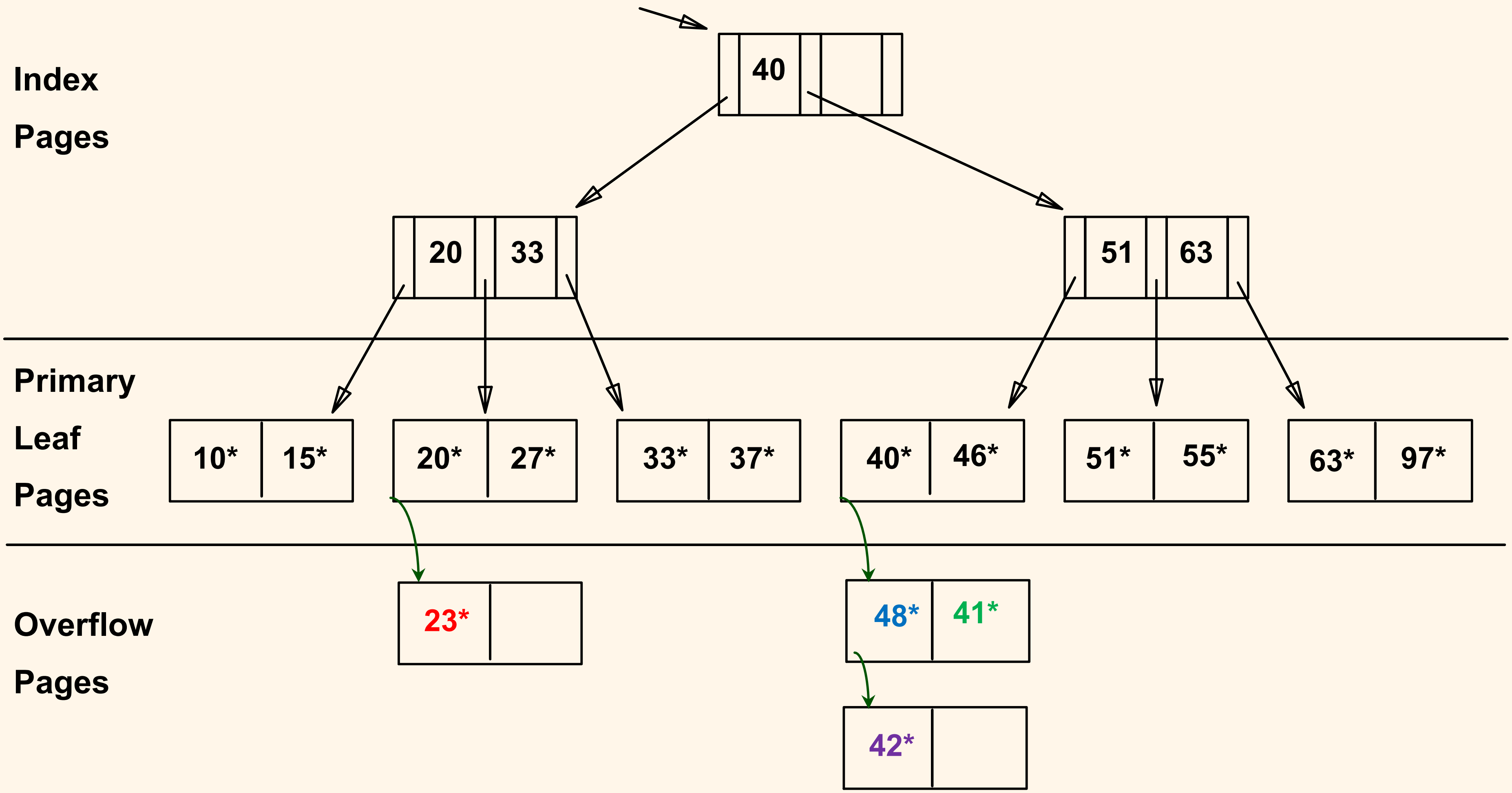
- *File creation*: Leaf (data) pages first allocated sequentially, sorted by search key; then index pages allocated, and then overflow pages.
- *Index entries*: <search key value, page id>; they "direct" searches for *data entries*, which are in leaf pages.
- Search: Start at root; use key comparisons to go to leaf. I/O cost $\propto$ **$\log_F N$**; F = #entries/index pg, N = # leaf pgs
- *Insert*: Find leaf data entry belongs to, put entry there.
- *Delete*: Find and remove entry from leaf; if overflow page is empty now, de-allocate it.
  - **Static tree structure**: *inserts/deletes affect only leaf pages*.

Example ISAM Tree

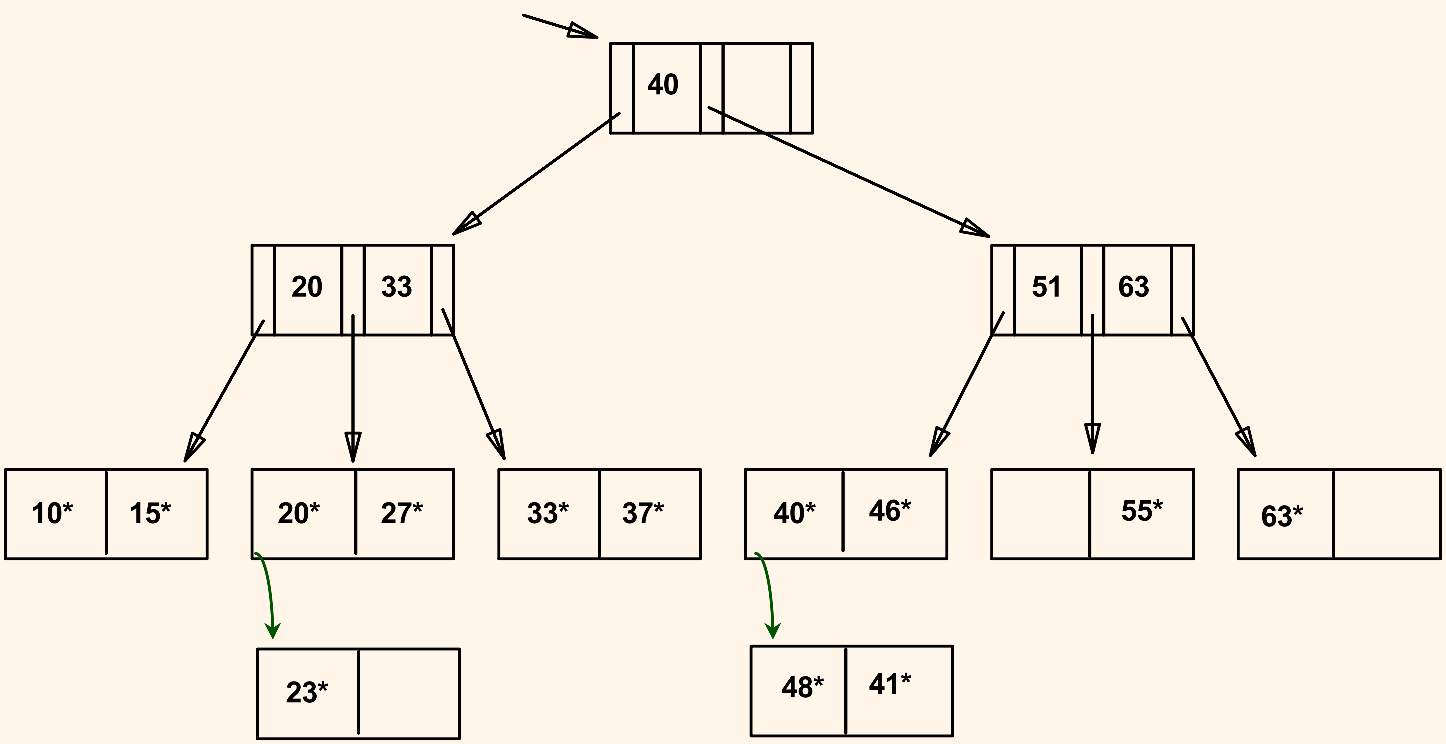
- Suppose each node can hold 2 entries (really more like 200, since nodes are disk pages!)



After Inserting 23, 48*, 41*, 42* ...*



... Then Deleting 42*, 51*, 97*



☛ *Note that 51* still appears in index levels, but **not** in leaf!*

To Be Continued…