VirginiaTech

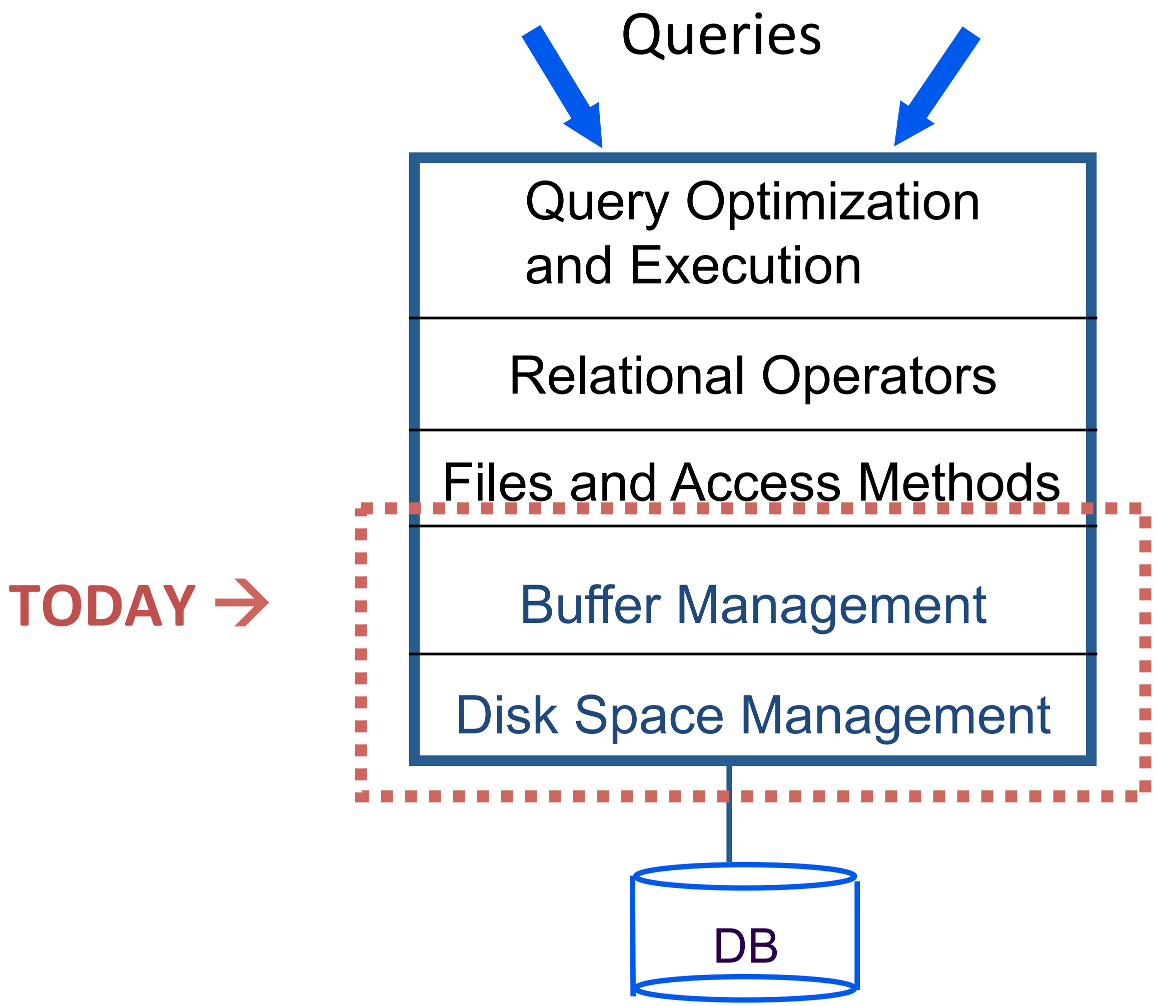
CS 4604: Introduction to Database Management Systems

B. Aditya Prakash

Lecture #8: Storing Data and Indexes

STORING DATA

DBMS Layers:


Queries
Query Optimization
and Execution
Relational Operators
Files and Access Methods
TODAY →
Buffer Management
Disk Space Management
DB

Leverage OS for disk/file management?

- Layers of abstraction are good … but:

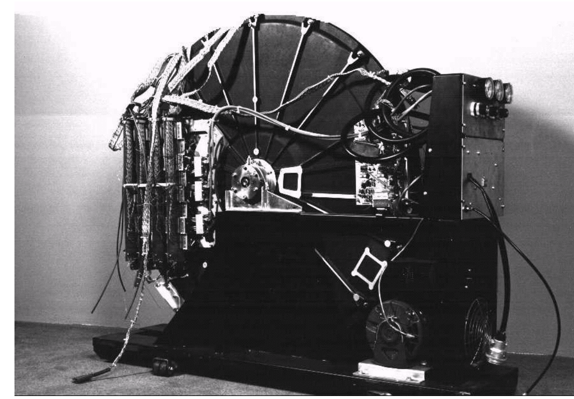
Leverage OS for disk/file management?

- Layers of abstraction are good … but:
  - Unfortunately, OS often gets in the way of DBMS

Leverage OS for disk/file management?

- DBMS wants/needs to do things "its own way"
  – Specialized prefetching
  – Control over buffer replacement policy
    - LRU not always best (sometimes worst!!)
  – Control over thread/process scheduling
    - "Convoy problem"
      – Arises when OS scheduling conflicts with DBMS locking
  – Control over flushing data to disk
    - WAL protocol requires flushing log entries to disk

Disks and Files



- DBMS stores information on disks.
  - but: disks are (relatively) VERY slow!
- Major implications for DBMS design!

Disks and Files

- Major implications for DBMS design:
  – READ: disk -> main memory (RAM).
  – WRITE: reverse
  – Both are high-cost operations, relative to in-memory operations, so must be planned carefully!

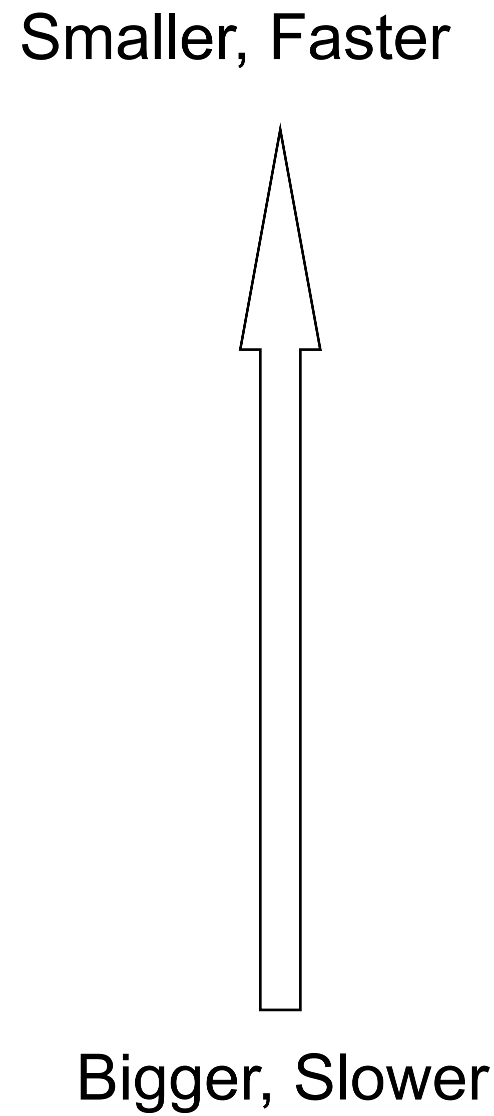
Why Not Store It All in Main Memory?

Why Not Store It All in Main Memory?

- *Costs too much.*
  - disk: ~$1/Gb; memory: ~$100/Gb
  - High-end Databases today in the 10-100 TB range.
  - Approx 60% of the cost of a production system is in the disks.
- *Main memory is volatile.*
- *Note*: some specialized systems do store entire database in main memory.

VirginiaTech

The Storage Hierarchy

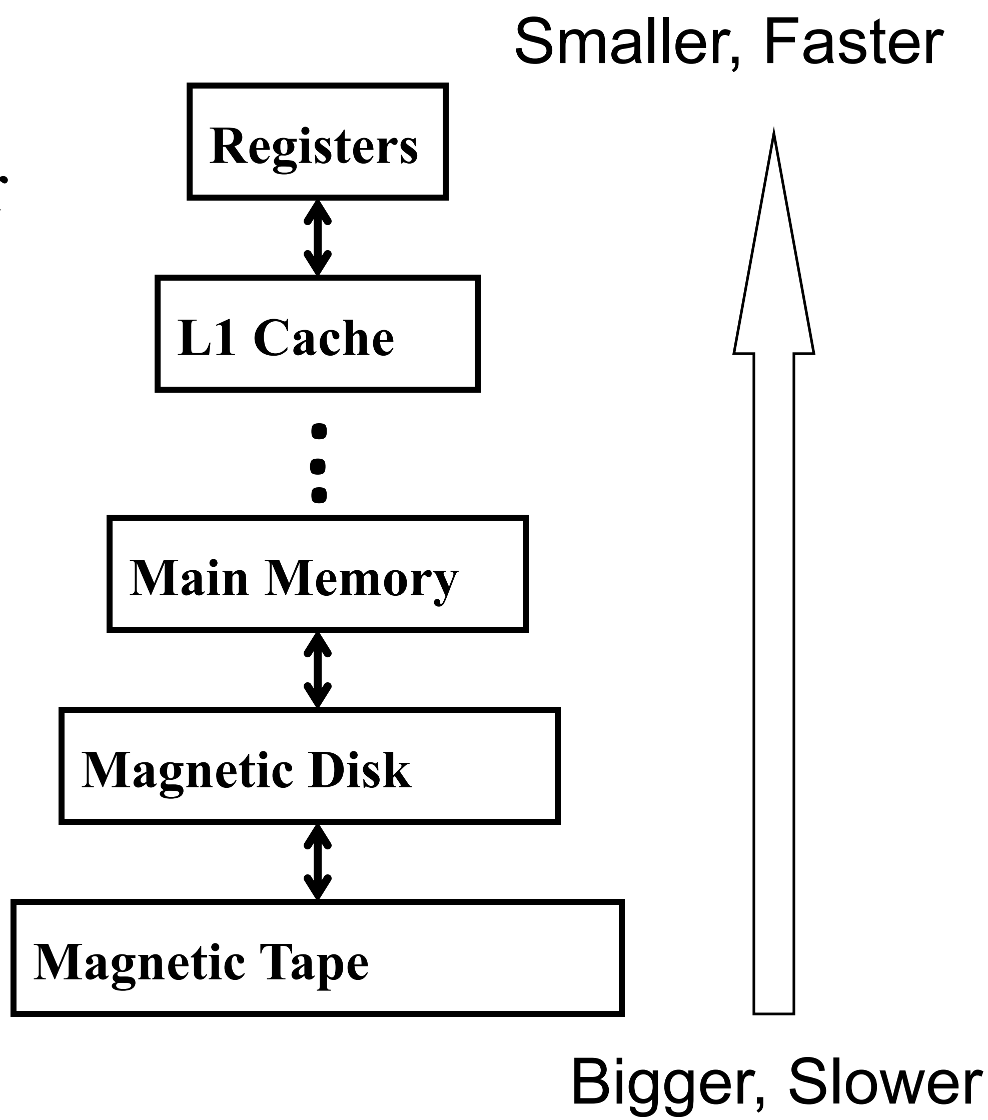


The Storage Hierarchy

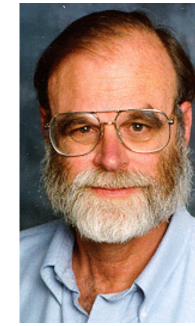
–Main memory (RAM) for currently used data.

–Disk for the main database (secondary storage).

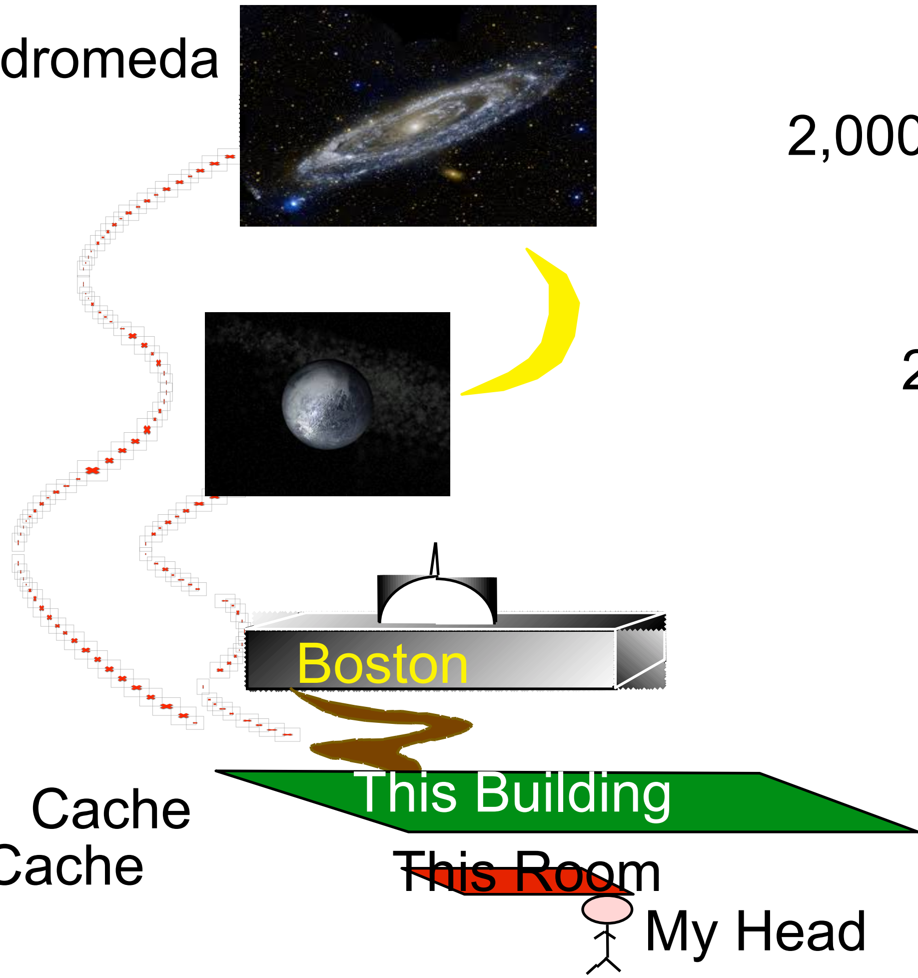
–Tapes for archiving older versions of the data (tertiary storage).



Jim Gray's Storage Latency Analogy: How Far Away is the Data?



10^9	Tape	Andromeda	2,000 Years
10^6	Disk		2 Years
100	Memory	Boston	1.5 hr
10	On Board Cache	This Building	10 min
2	On Chip Cache	This Room	
1	Registers	My Head	1 min

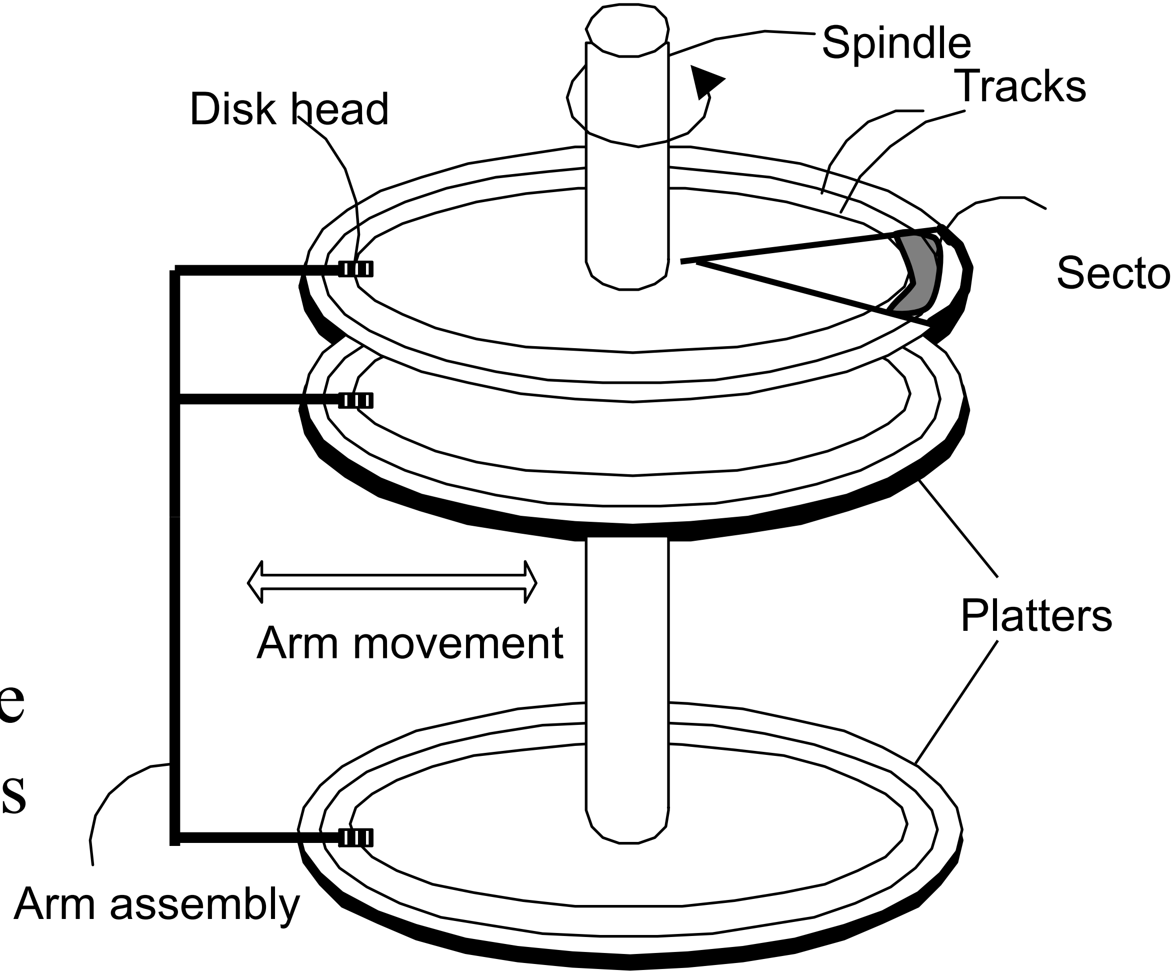


Disks

- 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:
  - .
  - .
  - .

Accessing a Disk Page

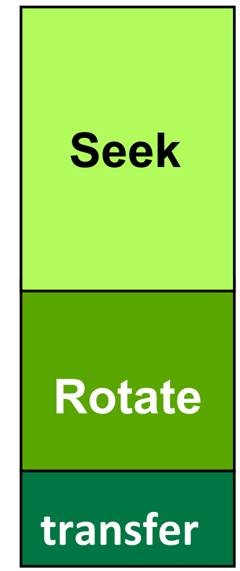
- Time to access (read/write) a disk block:
  - *seek time:* moving arms to position disk head on track
  - *rotational delay:* waiting for block to rotate under head
  - *transfer time:* actually moving data to/from disk surface

Accessing a Disk Page

- Relative times?
  - *seek time:*
  - *rotational delay:*
  - *transfer time:*

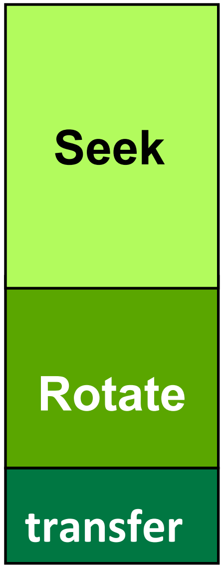
Accessing a Disk Page

- Relative times?
  - *seek time:* about 1 to 20msec
  - *rotational delay:* 0 to 10msec
  - *transfer time:* < 1msec per 4KB page



Seek time & rotational delay dominate

- Key to lower I/O cost:
  reduce seek/rotation delays!
- Also note: 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
- Accesing 'next' block is cheap
- A useful optimization: pre-fetching
  - See textbook page 323

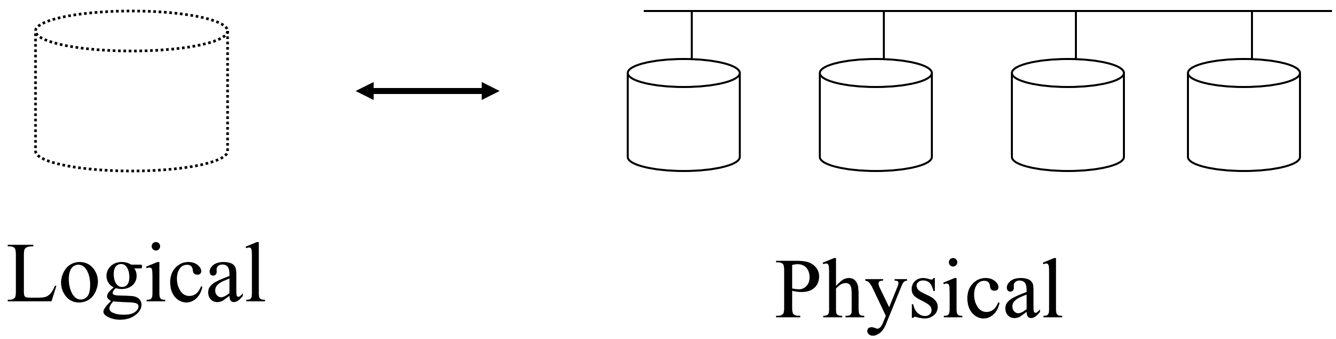
Rules of thumb...

1. Memory access much faster than disk I/O (~ 1000x)

- "Sequential" I/O faster than "random" I/O (~ 10x)

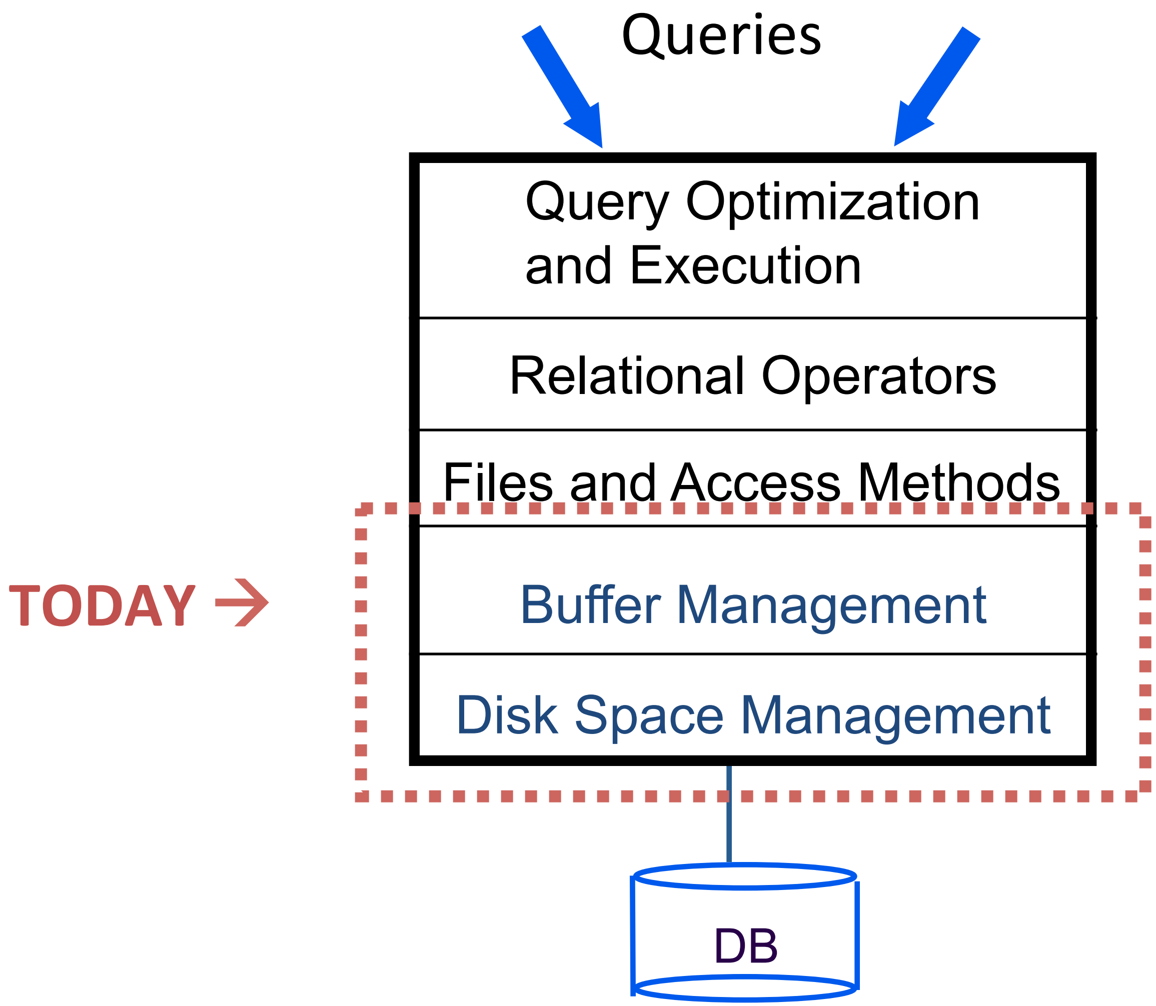
Disk Arrays: RAID

Just FYI



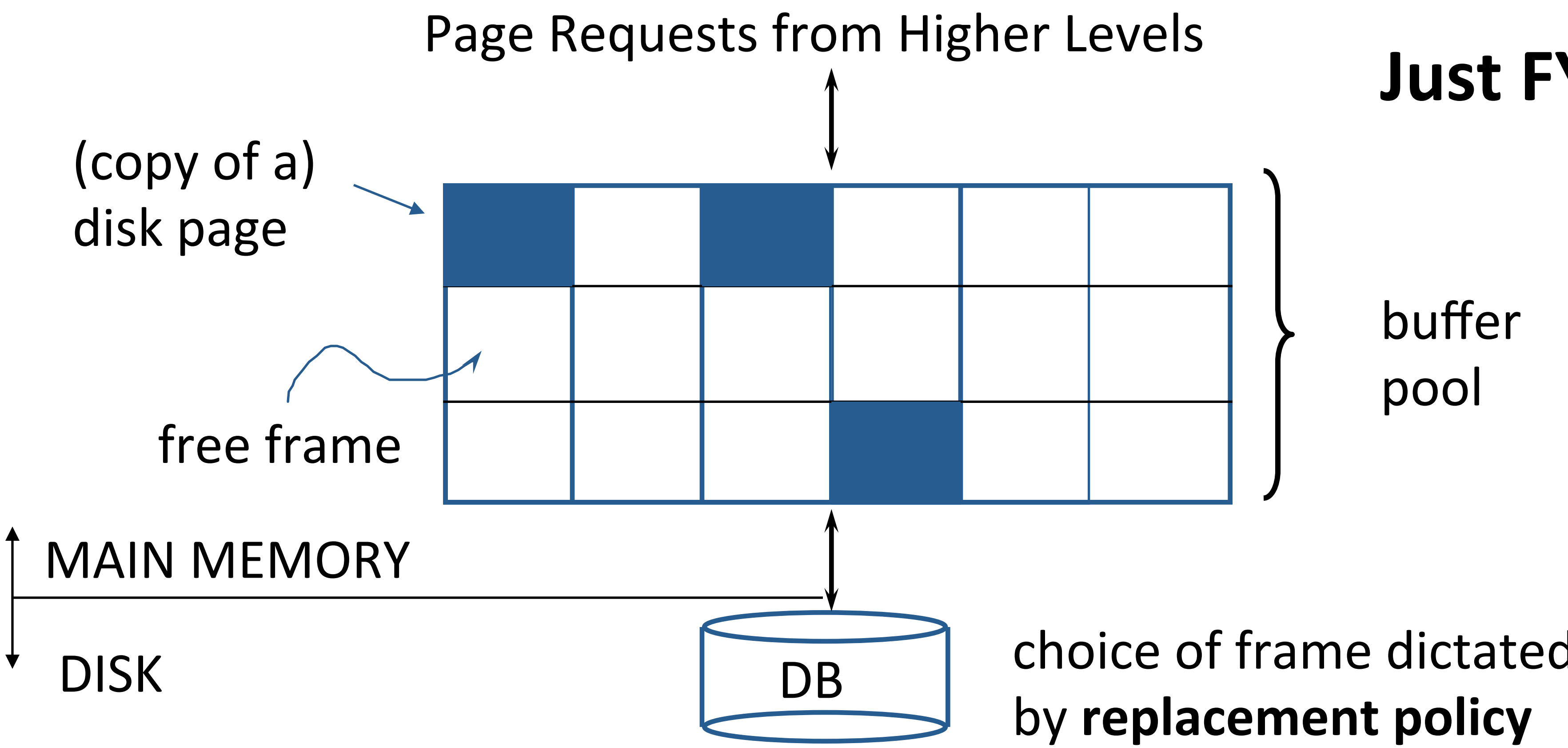
- Benefits:
  - Higher throughput (via data "striping")
  - Longer MTTF (via redundancy)

Recall: DBMS Layers



Buffer Management in a DBMS

Just FYI



Files

- FILE: A collection of pages, each containing a collection of records.
- Must support:
  - insert/delete/modify record
  - read a particular record (specified using *record id*)
  - scan all records (possibly with some conditions on the records to be retrieved)

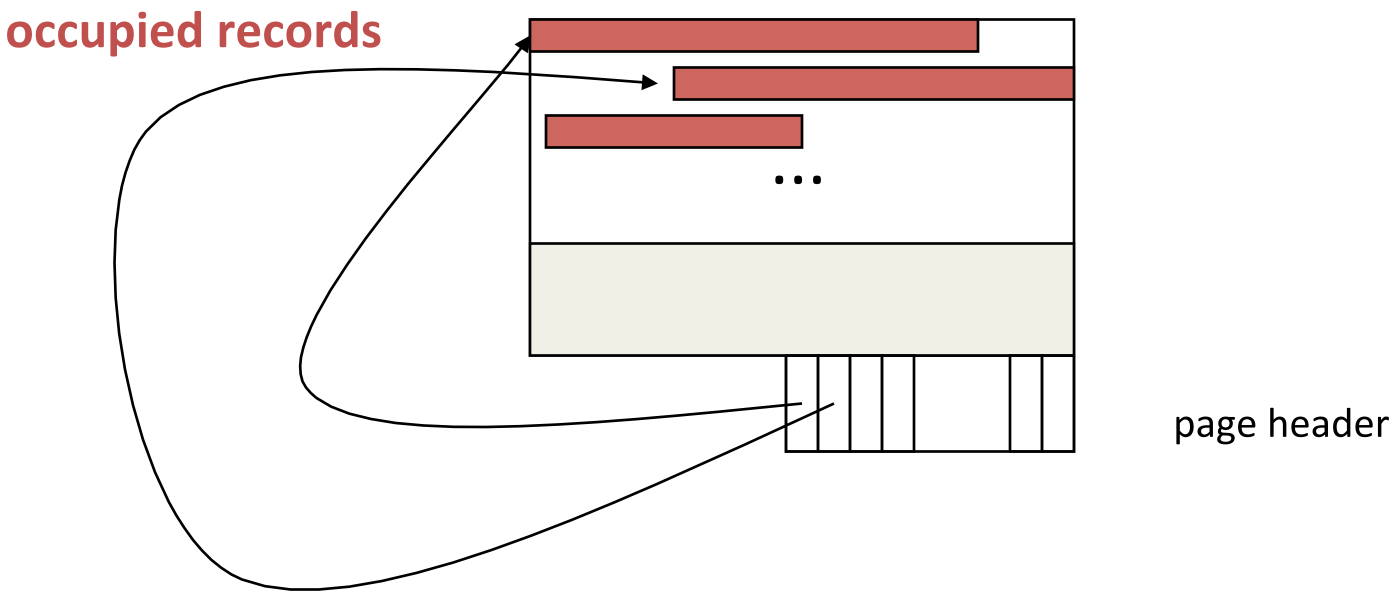
Alternative File Organizations

Several alternatives (w/ trade-offs):

- Heap files: Suitable when typical access is a file scan retrieving all records.
- Sorted Files:
- Index File Organizations:

Variable length records

- SLOTTED PAGE organization - popular.



Conclusions---Storing

- Memory hierarchy
- Disks: (>1000x slower) - thus
  - pack info in blocks
  - try to fetch nearby blocks (sequentially)
- Record organization: Slotted page

TREE INDEXES

Declaring Indexes

- No standard!
- Typical syntax:

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

Types of Indexes

- **Primary:** index on a 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 (1): 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"
```

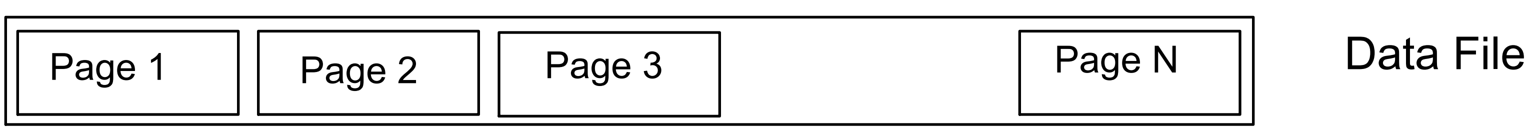
Using Indexes (1): Equality Searches

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

- Can use Hashes, but see next

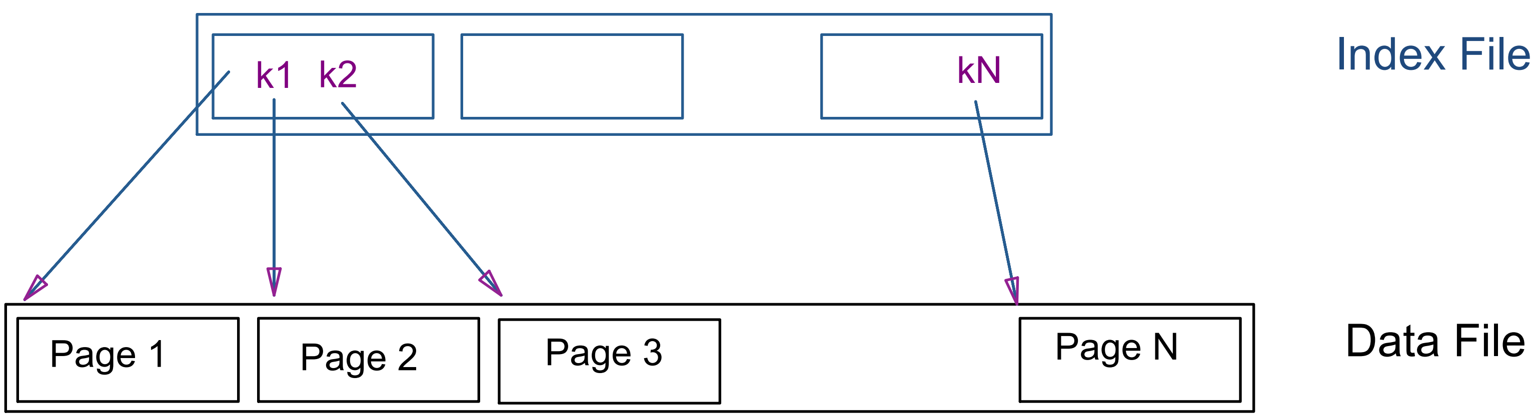
Using Indexes (2): Range Searches

- ``Find all students with gpa > 3.0’’
- may be slow, even on sorted file
- Hashes not a good idea!
- What to do?



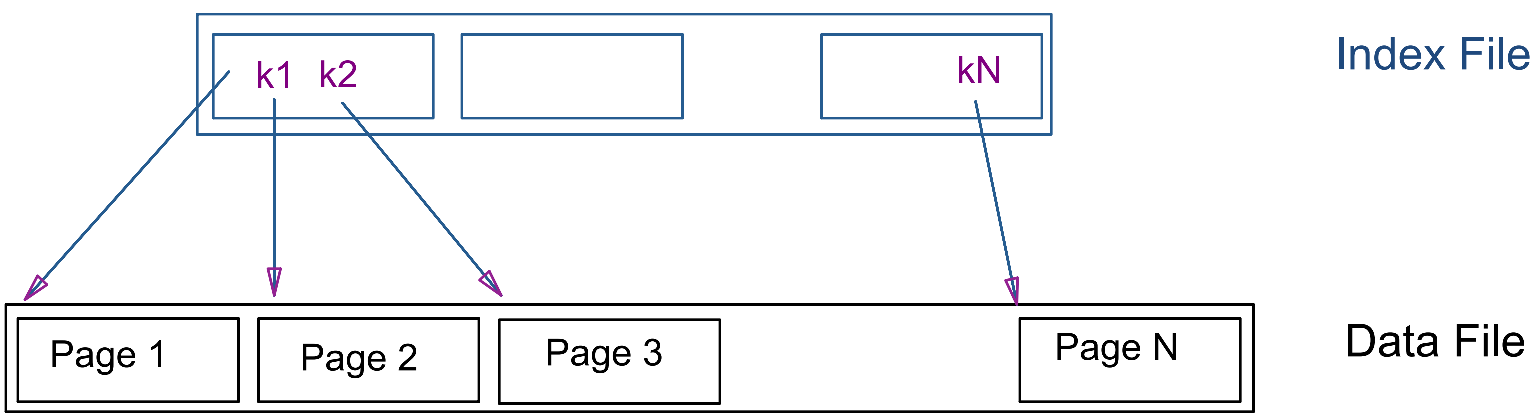
Range Searches

- *``Find all students with gpa > 3.0''*
- may be slow, even on sorted file
- Solution: Create an `index' file.



Range Searches

- More details:
- if index file is small, do binary search there
- Otherwise??

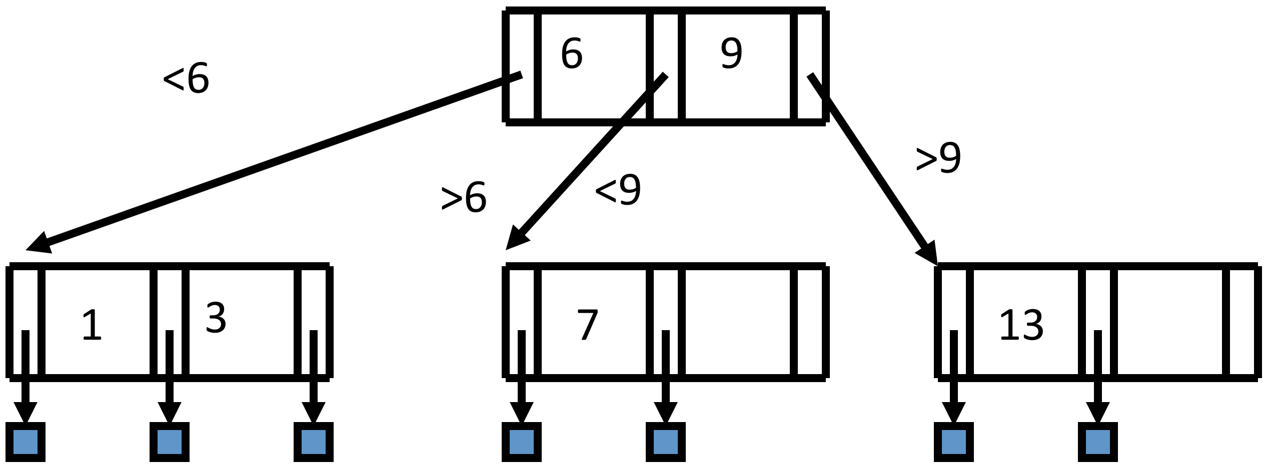


B-trees

- the most successful family of index schemes (B-trees, B+-trees, B*-trees)
- Can be used for primary/secondary, clustering/non-clustering index.
- balanced "n-way" search trees
- Original Paper: Rudolf Bayer and McCreight, E. M. Organization and Maintenance of Large Ordered Indexes. Acta Informatica 1, 173-189, 1972.

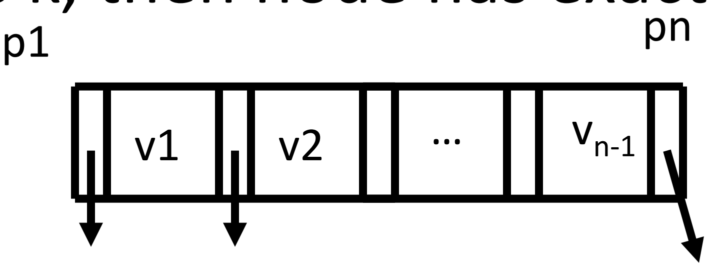
B-trees

- Eg., B-tree of order d=1:



B - tree properties:

- each node, in a B-tree of order d:
  - Key order
  - at most n=2d keys
  - at least d keys (except root, which may have just 1 key)
  - all leaves at the same level
  - if number of pointers is k, then node has exactly k-1 keys
  - (leaves are empty)

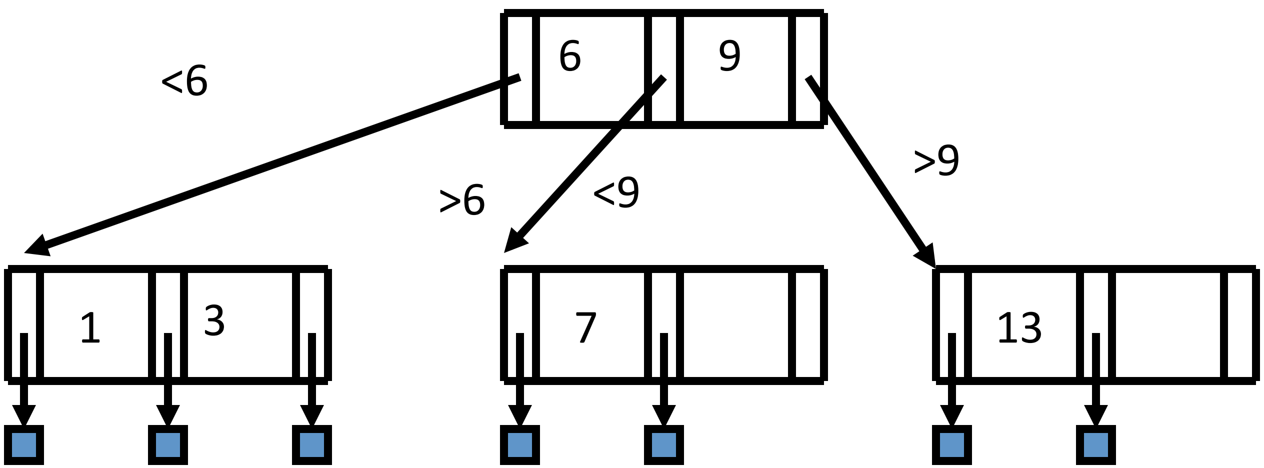


Properties

- "block aware" nodes: each node is a disk page
- O(log (N)) for everything! (ins/del/search)
- typically, if d = 50 - 100, then 2 - 3 levels
- utilization >= 50%, guaranteed; on average 69%

Queries

- Algo for exact match query? (eg., ssn=8?)

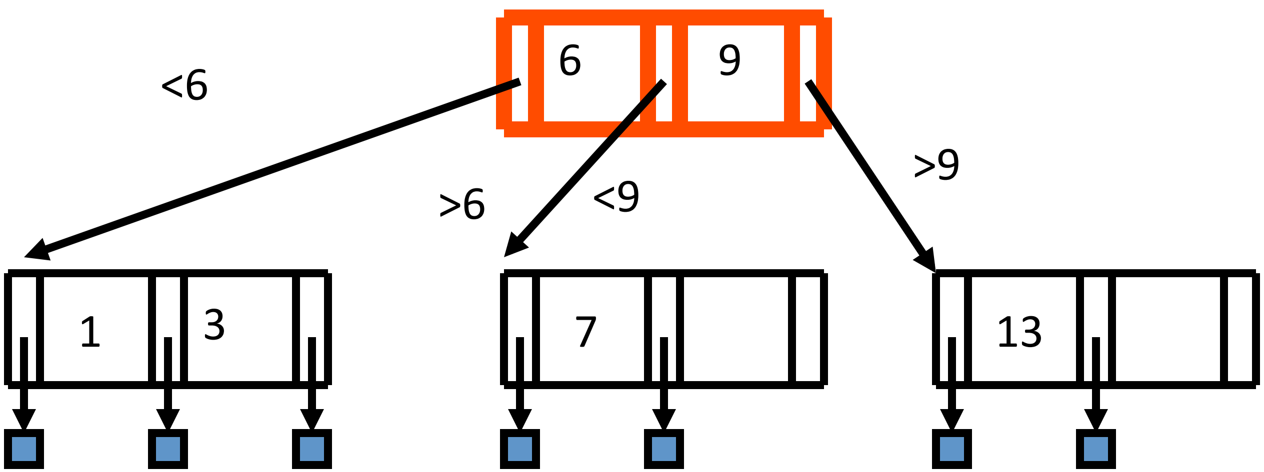


JAVA animation

- http://slady.net/java/bt/

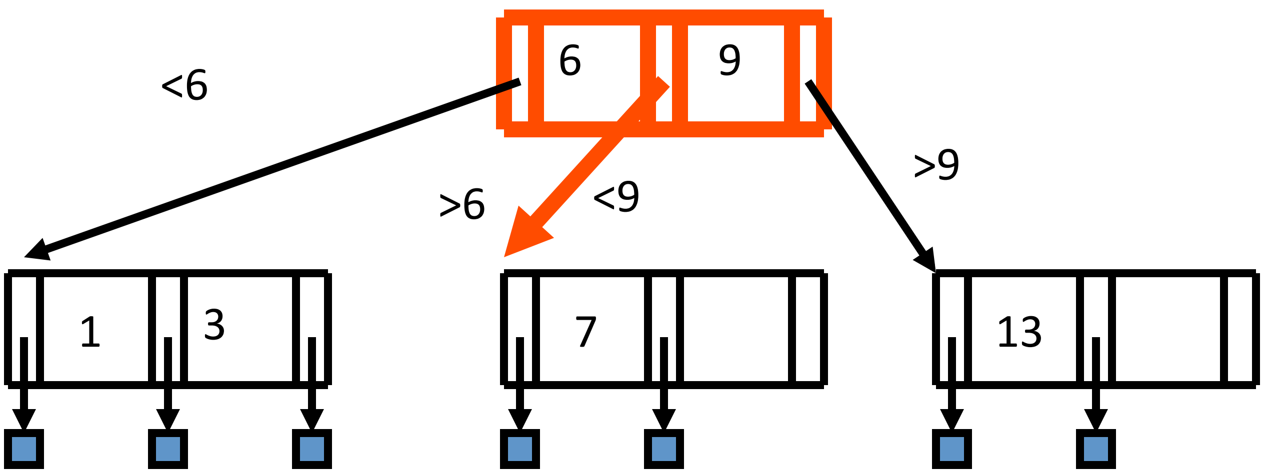
Queries

- Algo for exact match query? (eg., ssn=8?)



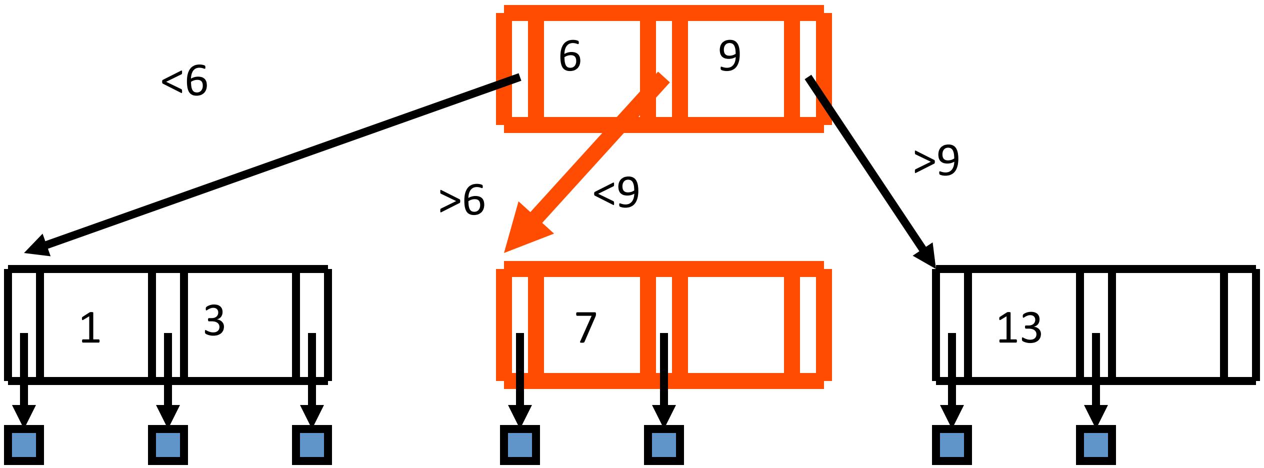
Queries

- Algo for exact match query? (eg., ssn=8?)



Queries

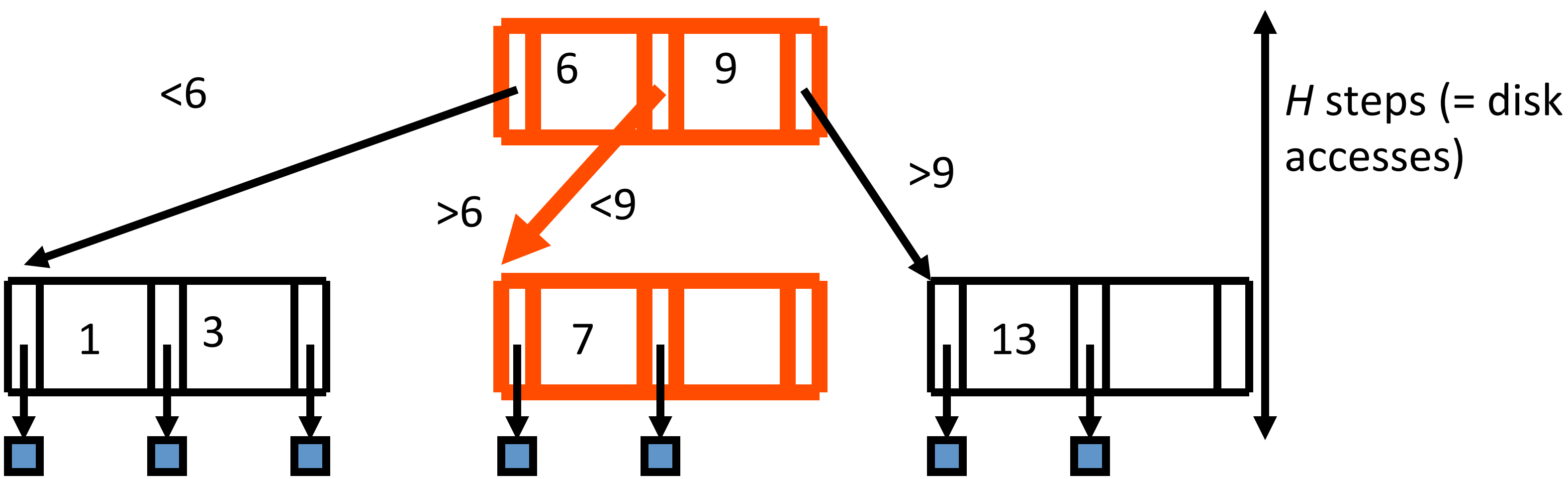
- Algo for exact match query? (eg., ssn=8?)



Queries

Queries

- Algo for exact match query? (eg., ssn=8?)

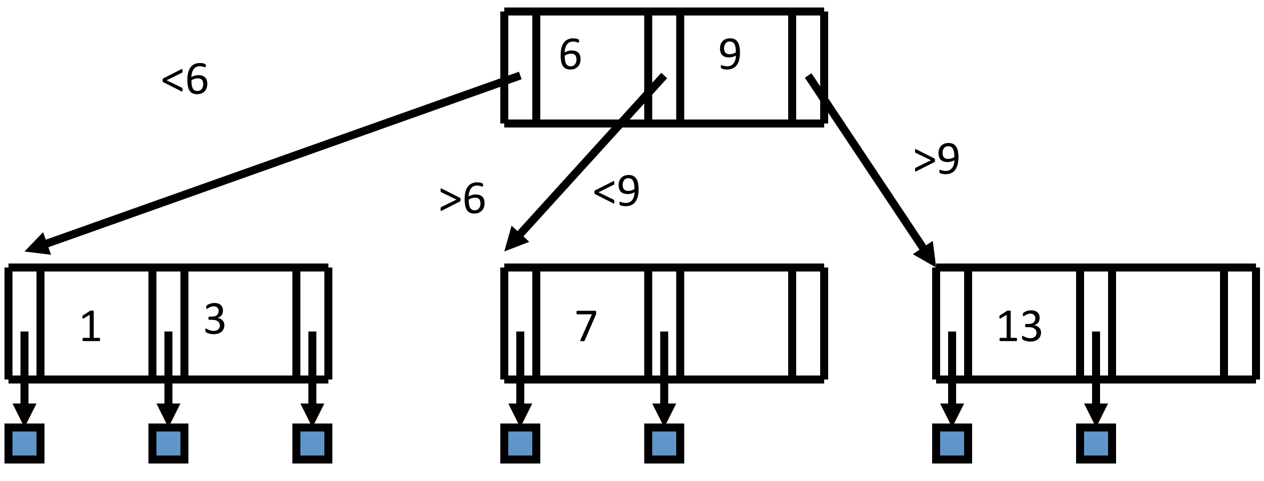


Queries

- what about range queries? (eg., 5<salary<8)
- Proximity/ nearest neighbor searches? (eg., salary ~ 8)

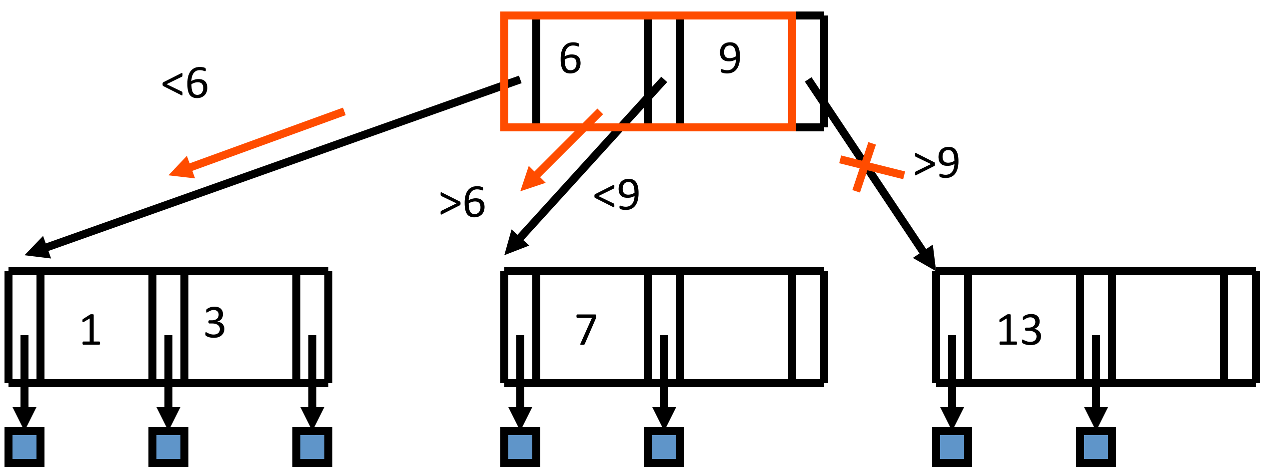
Queries

- what about range queries? (eg., 5<salary<8)
- Proximity/ nearest neighbor searches? (eg., salary ~ 8)



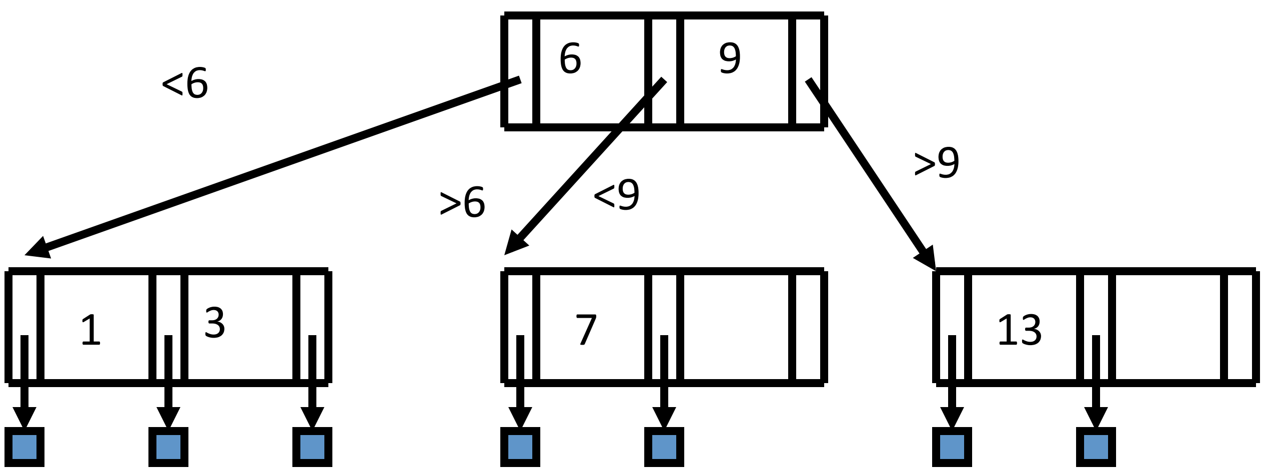
Queries

- **what about range queries? (eg., 5<salary<8)**
- Proximity/ nearest neighbor searches? (eg., salary ~ 8)



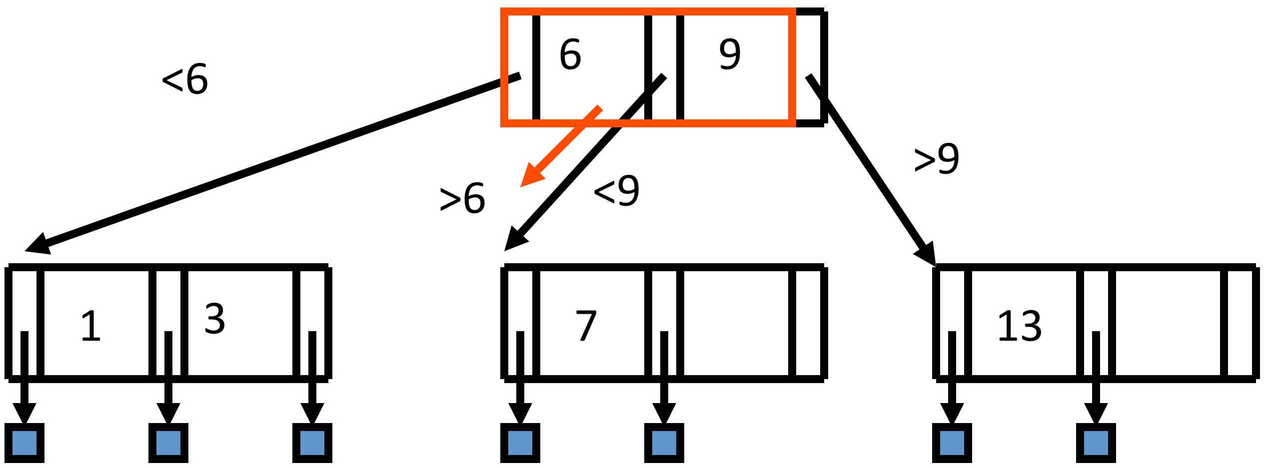
Queries

- what about range queries? (eg., 5<salary<8)
- **Proximity/ nearest neighbor searches? (eg., salary ~ 8)**



Queries

- what about range queries? (eg., 5<salary<8)
- **Proximity/ nearest neighbor searches? (eg., salary ~ 8)**

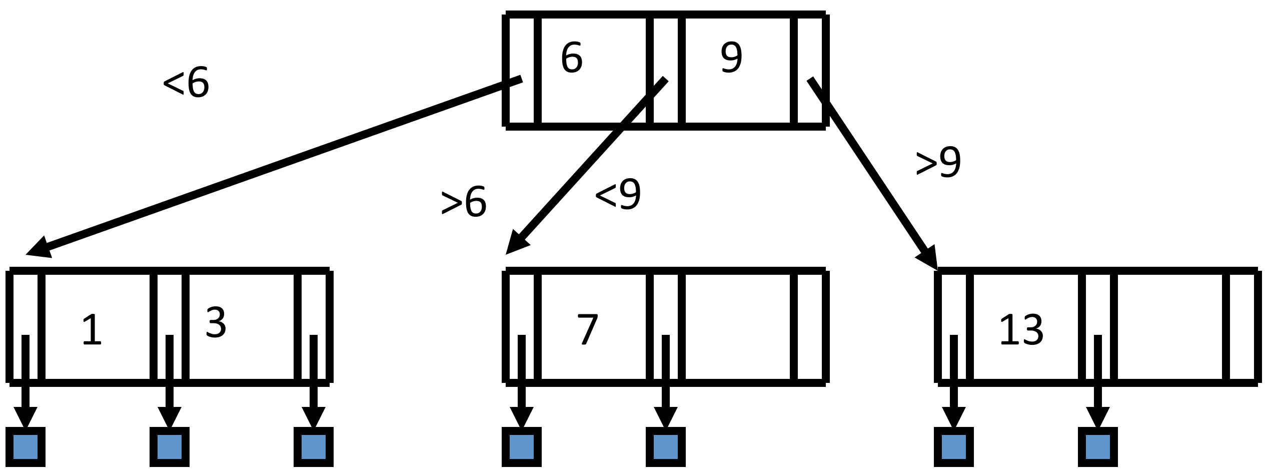


Variations

- How could we do even better than the B-trees above?

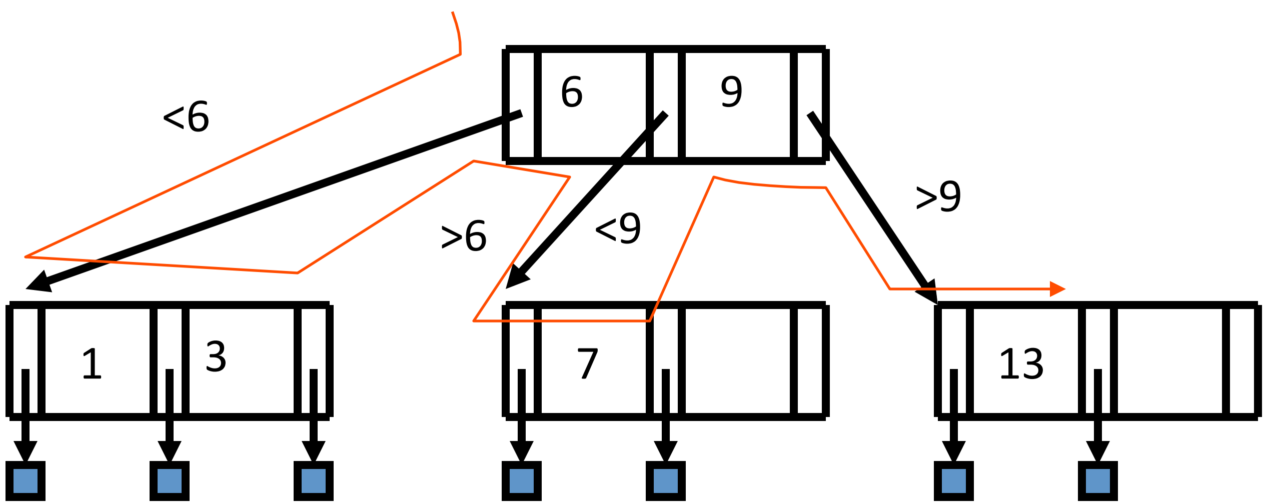
B+ trees - Motivation

- B-tree – print keys in sorted order:



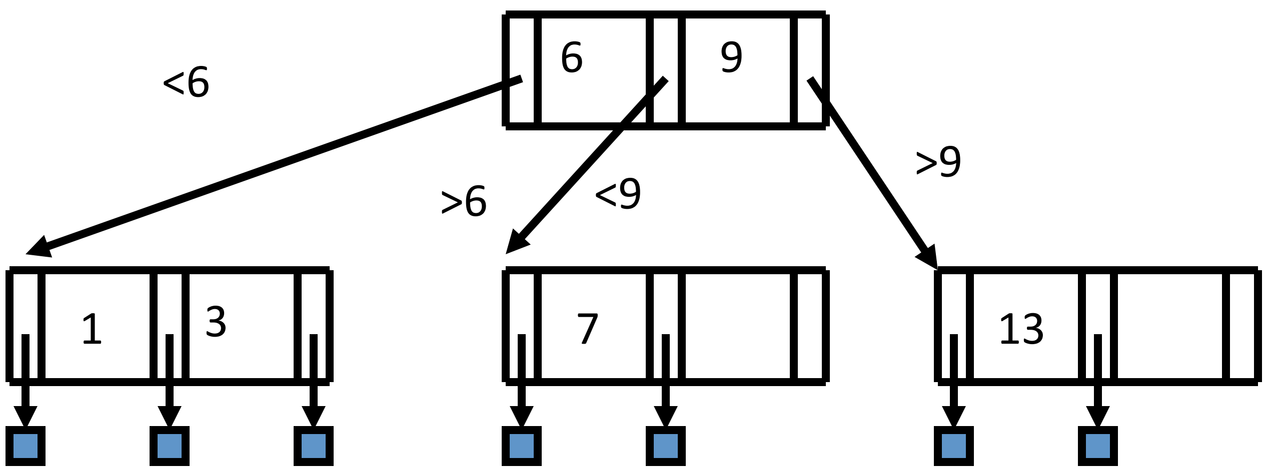
B+ trees - Motivation

- B-tree needs back-tracking – how to avoid it?



B+ trees - Motivation

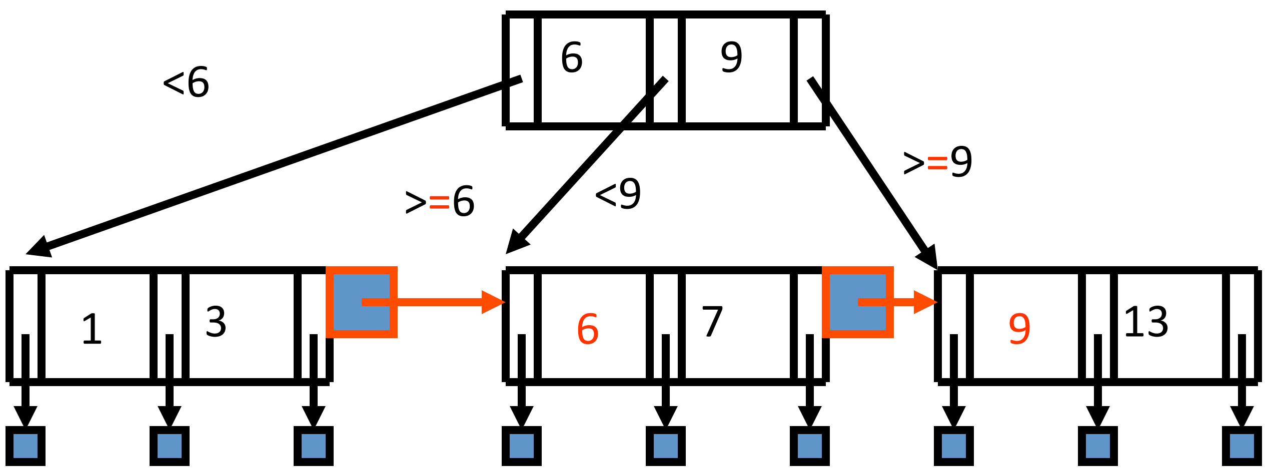
- Stronger reason: for clustering index, data records are scattered:



Solution: B+ - trees

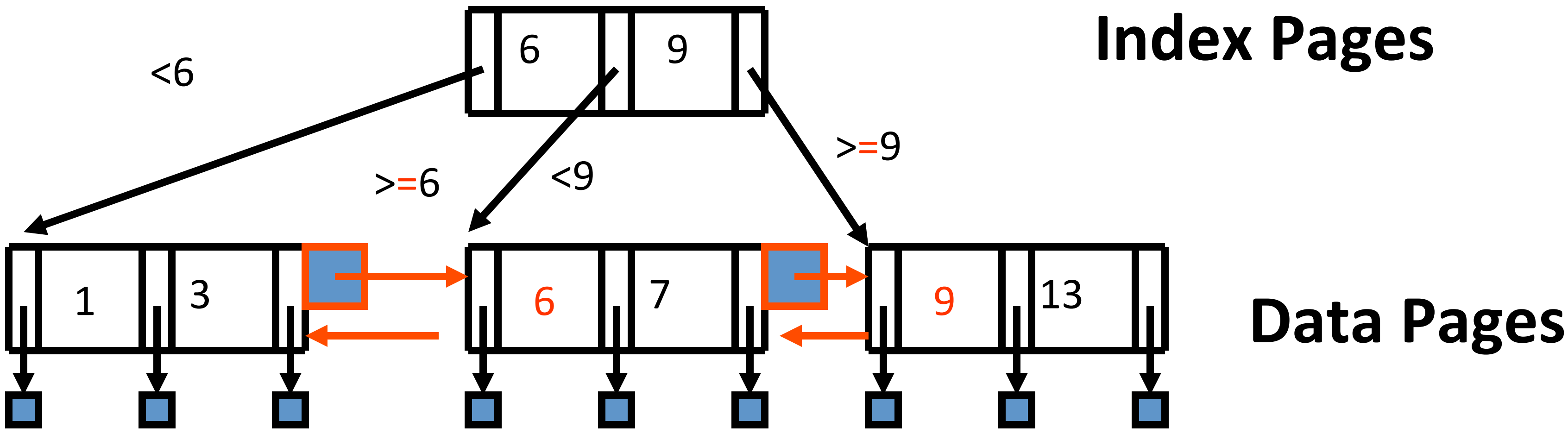
- facilitate sequential ops
- They string all leaf nodes together
- AND
- replicate keys from non-leaf nodes, to make sure every key appears at the leaf level
- (vital, for clustering index!)

B+ trees



VirginiaTech

B+ trees

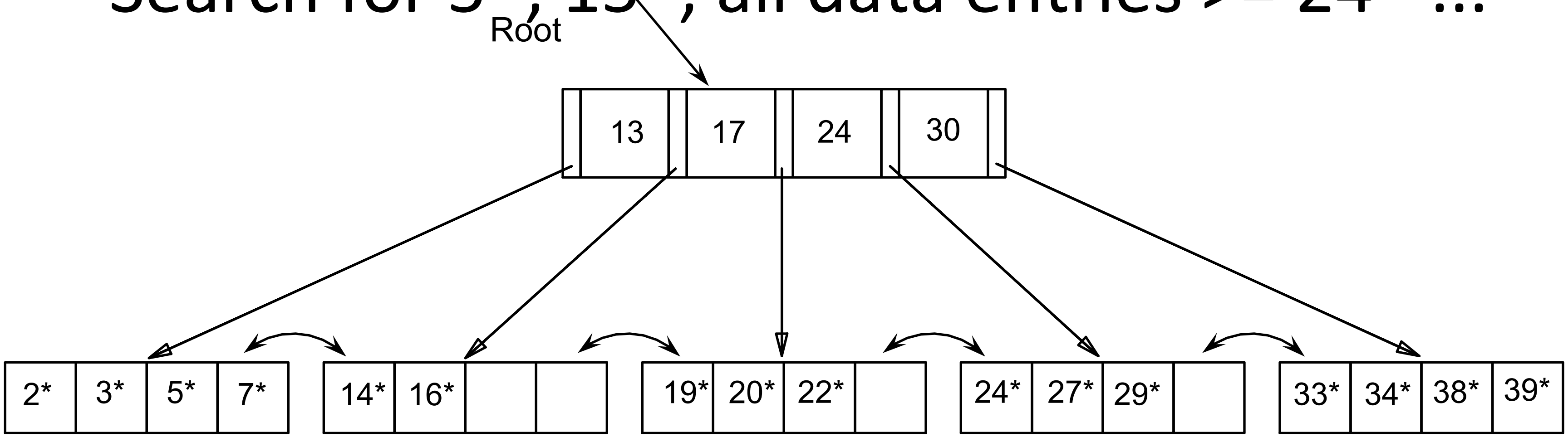


B+ trees

- More details: next (and textbook)
- In short: on split
  - at leaf level: COPY middle key upstairs
  - at non-leaf level: push middle key upstairs (as in plain B-tree)

Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf
- Search for 5*, 15*, all data entries >= 24* ...

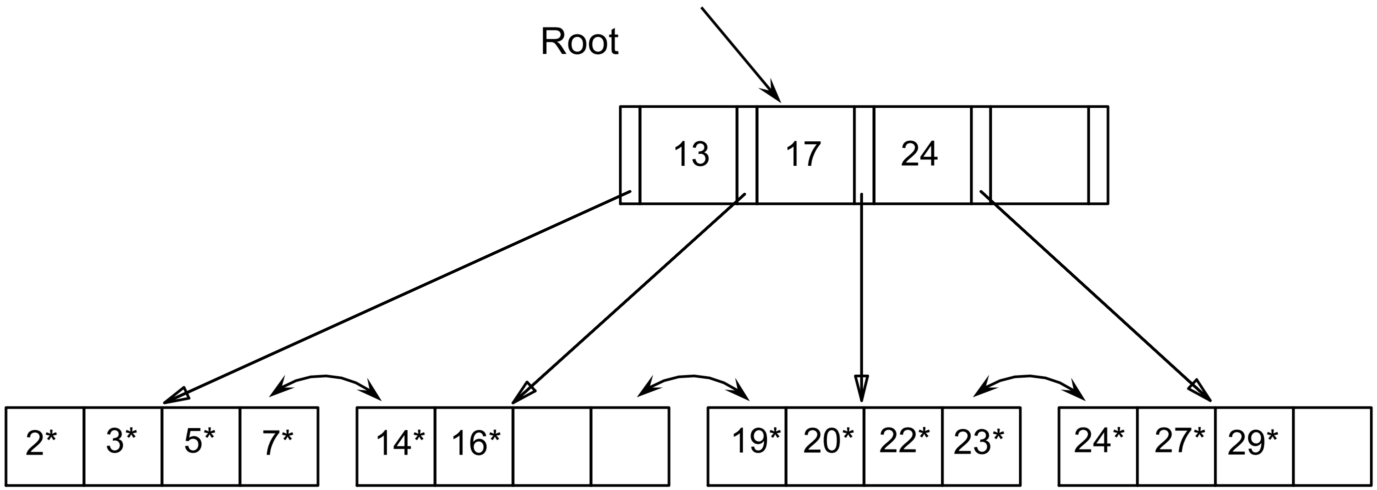


Based on the search for 15, we know it is not in the tree!*

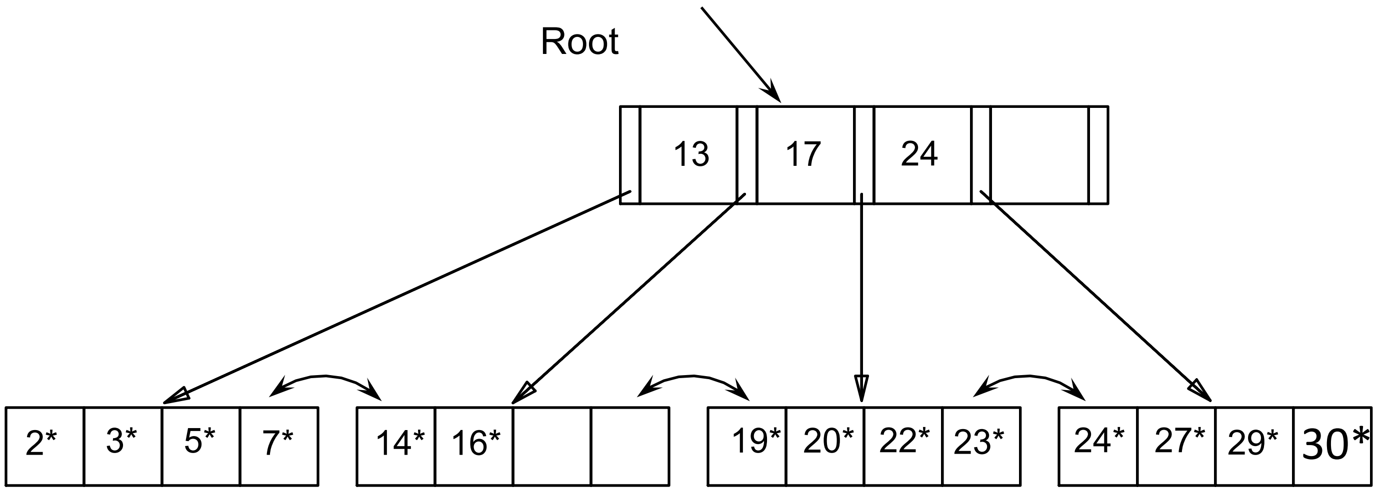
Inserting a Data Entry into a B+ Tree

- Find correct leaf L.
- 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.
- parent node may overflow
  - but then: push up middle key. Splits "grow" tree; root split increases height.

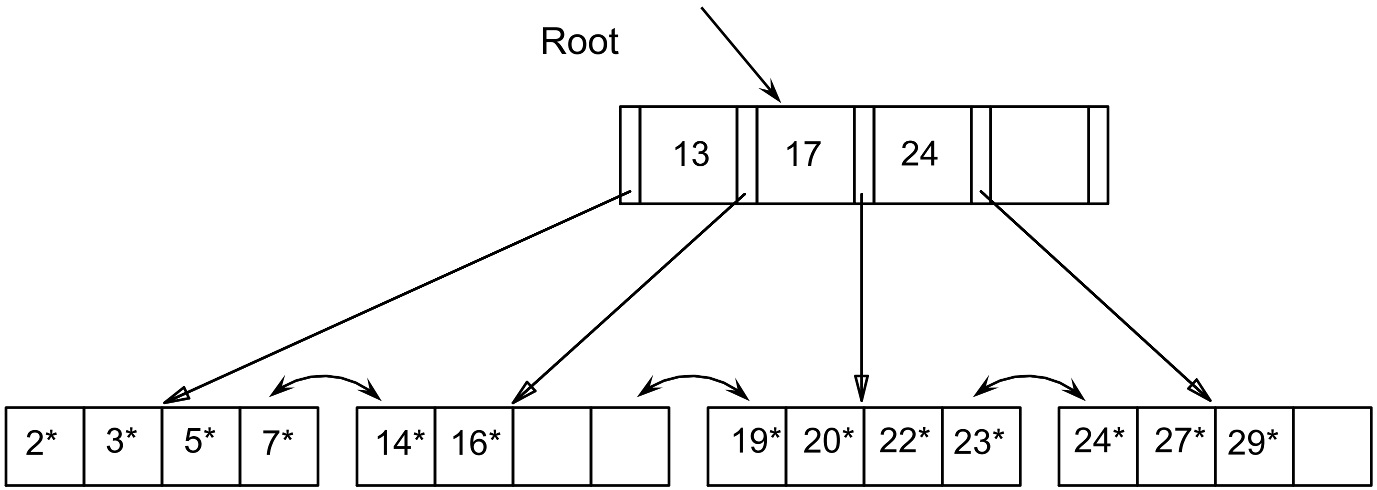
Example B+ Tree – Inserting 30*



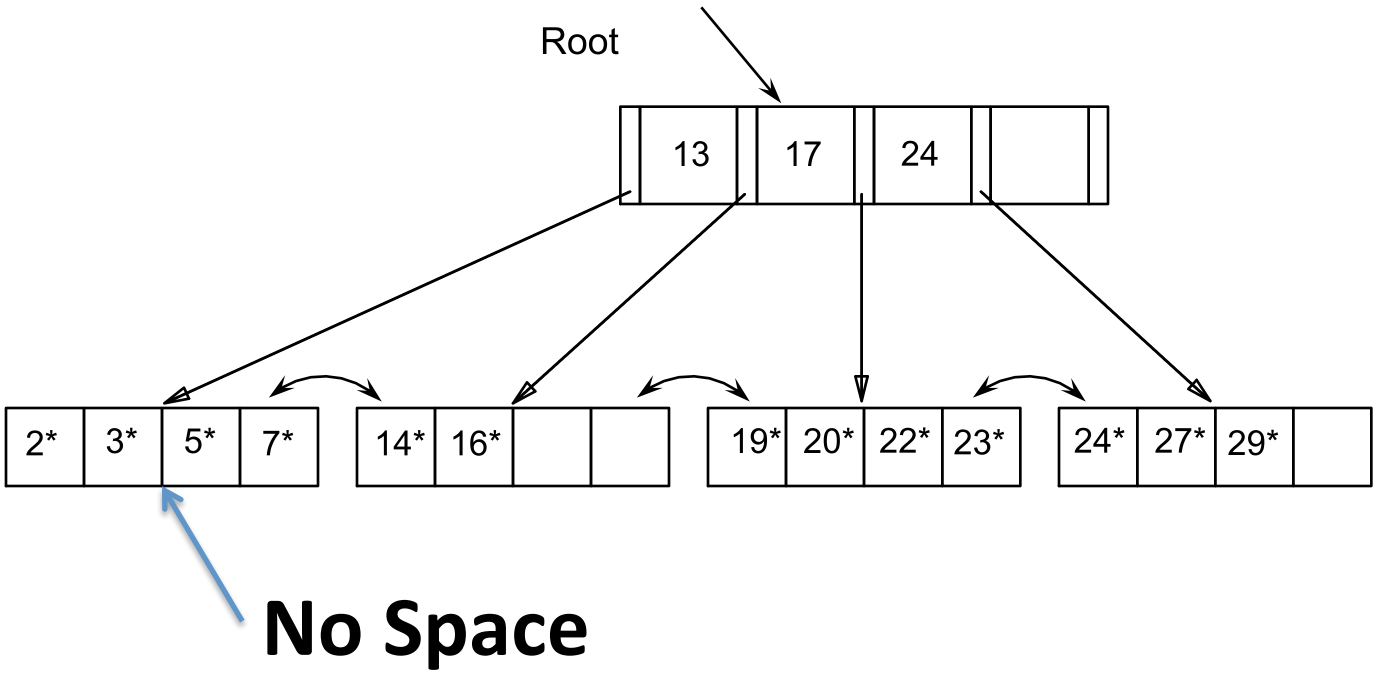
Example B+ Tree – Inserting 30*



Example B+ Tree - Inserting 8*

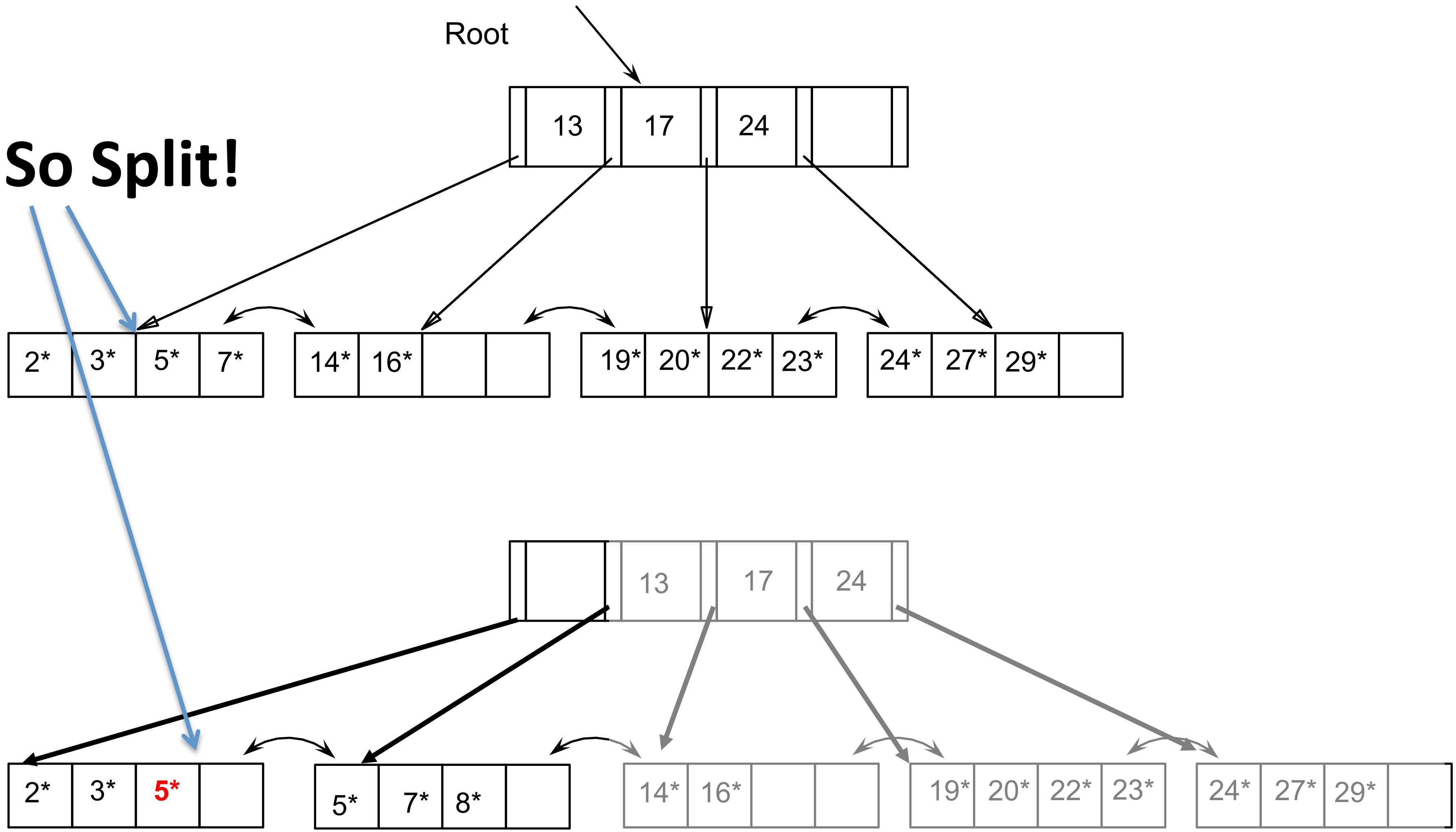


Example B+ Tree - Inserting 8*



Example B+ Tree - Inserting 8*

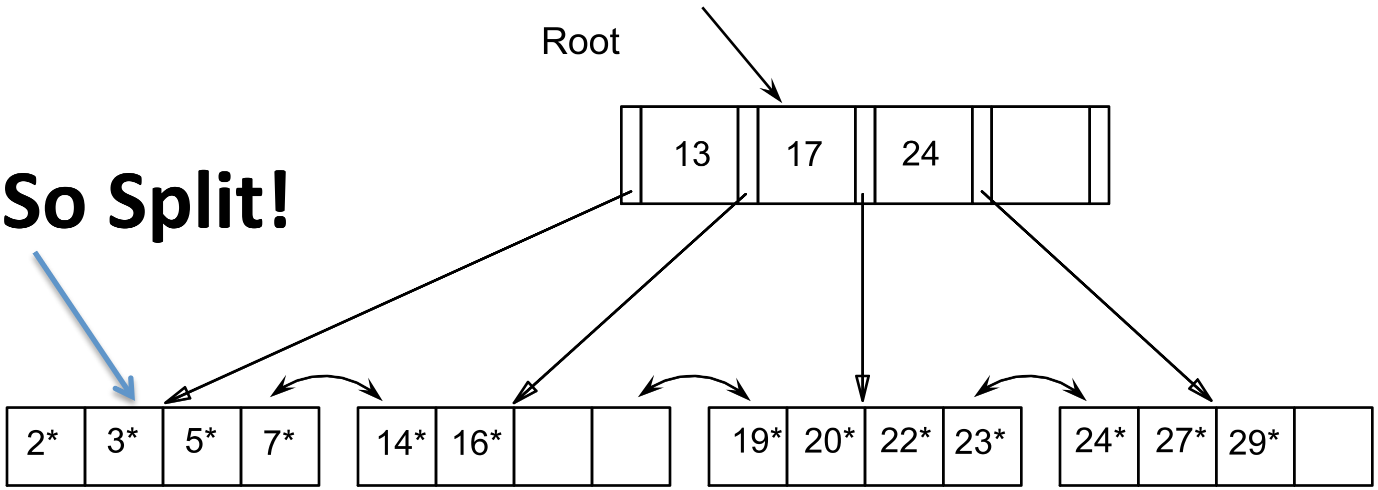
So Split!



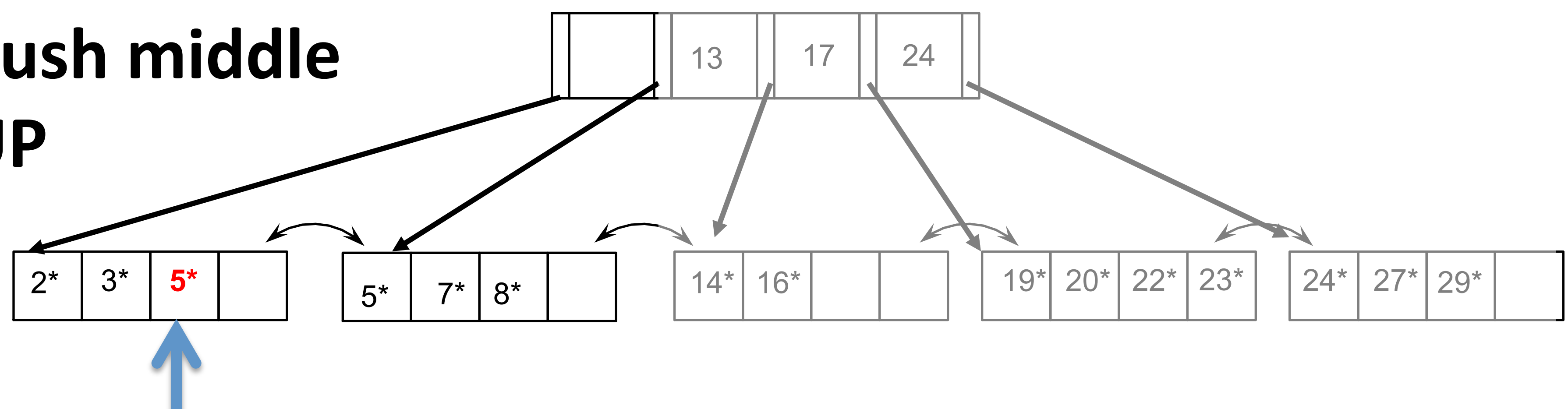
VirginiaTech

Example B+ Tree - Inserting 8*

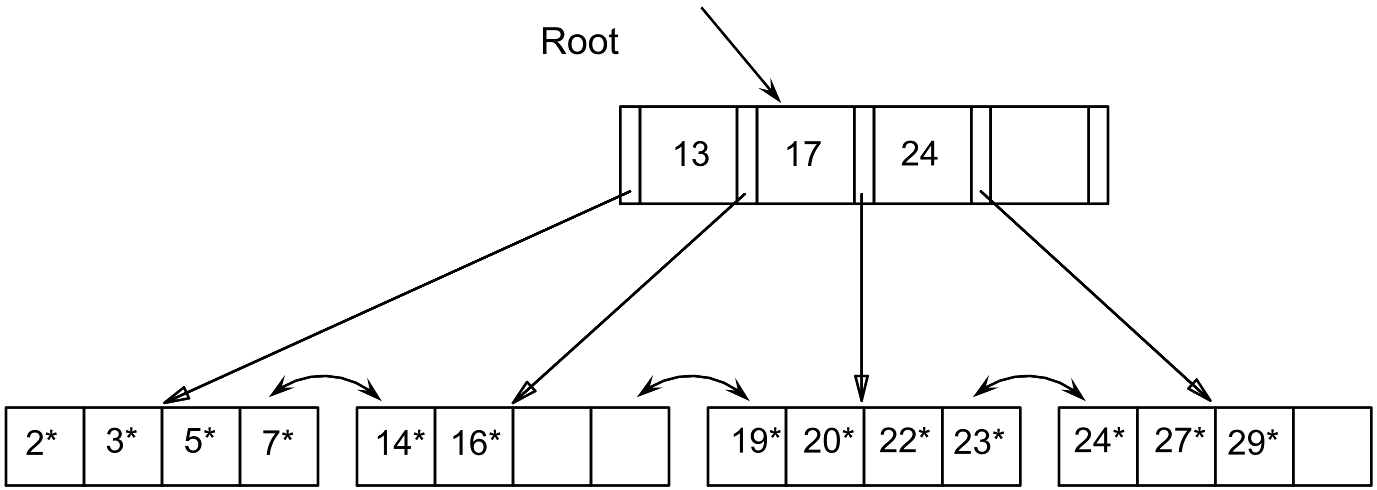
So Split!



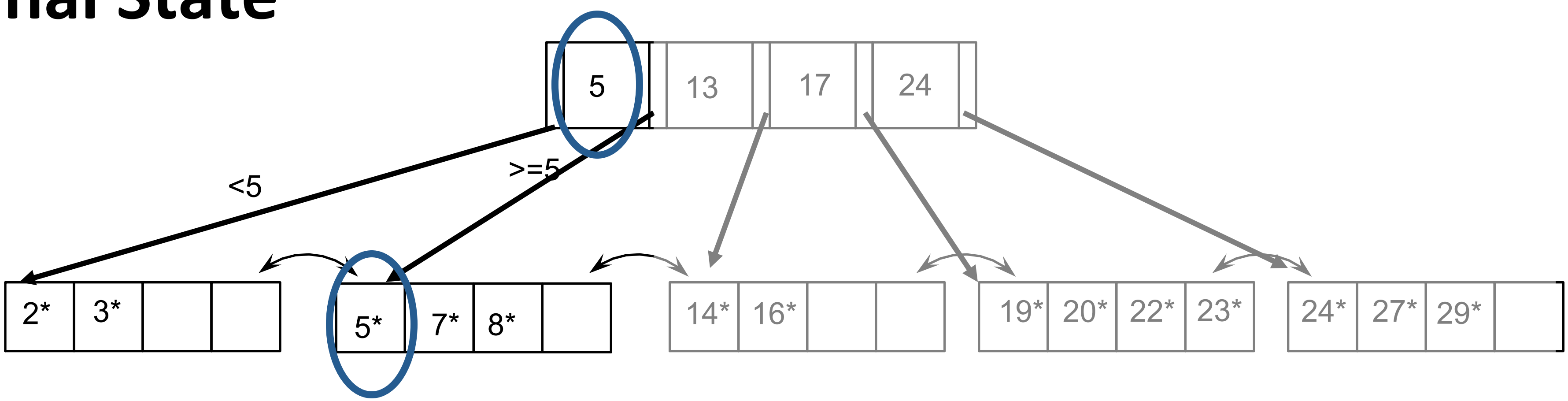
And then push middle UP



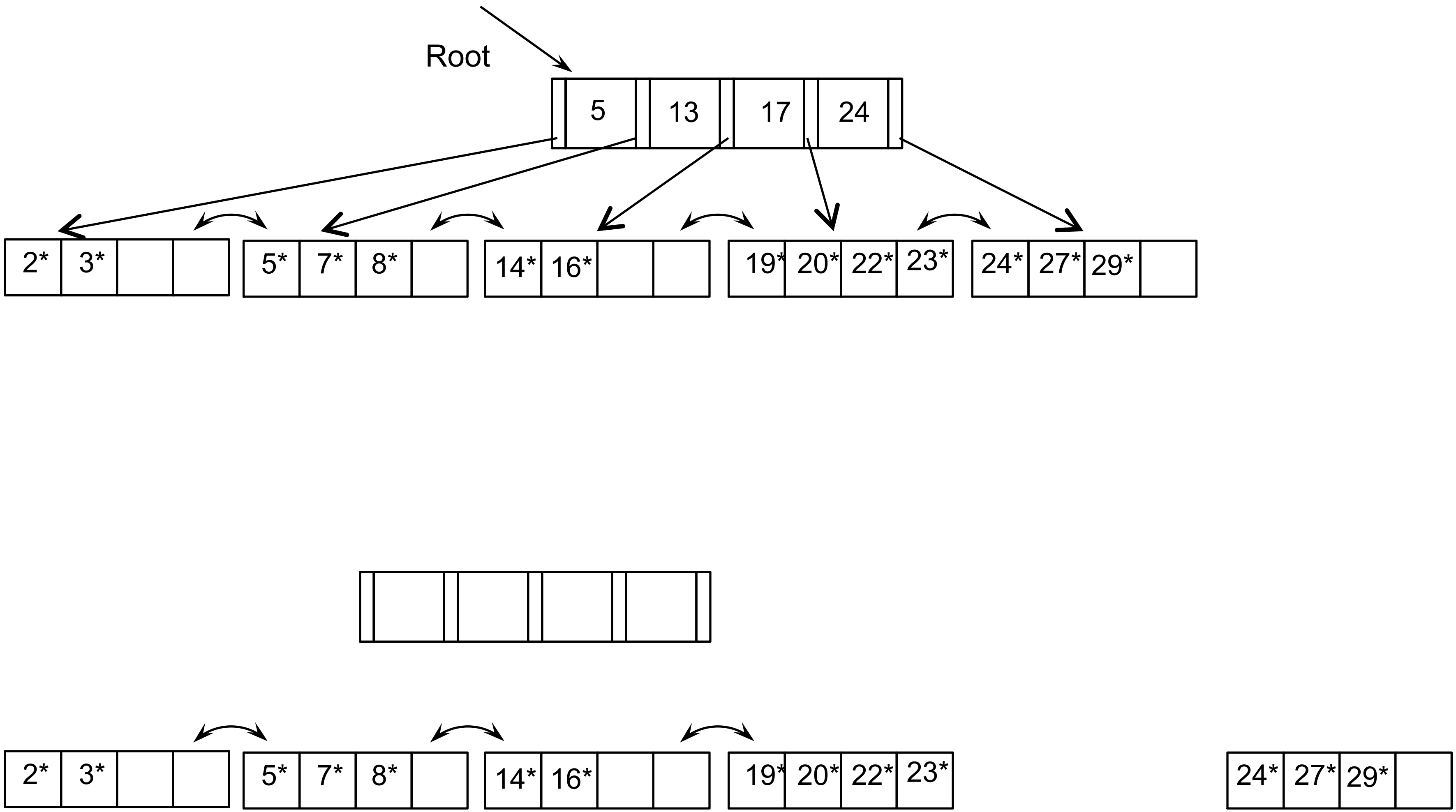
Example B+ Tree - Inserting 8*



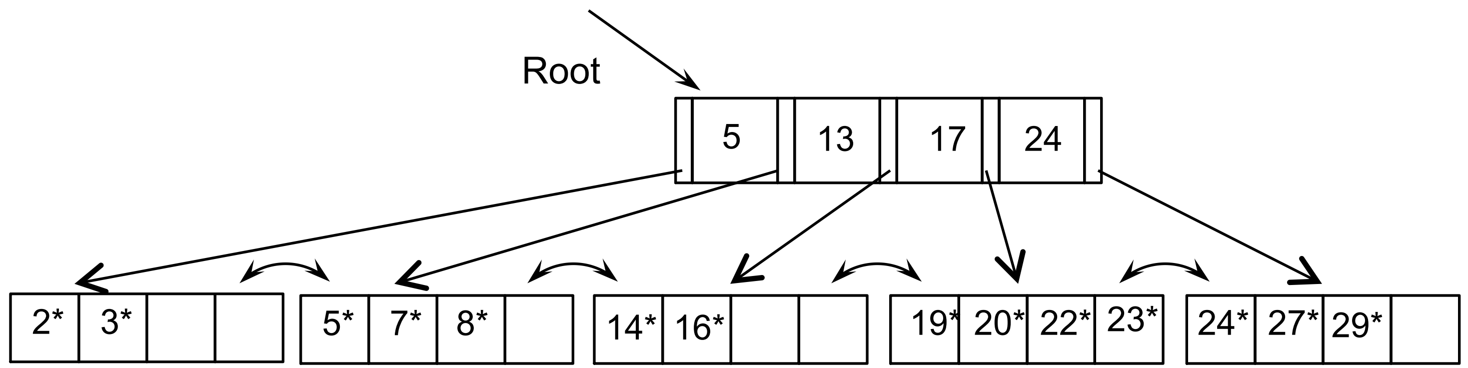
Final State

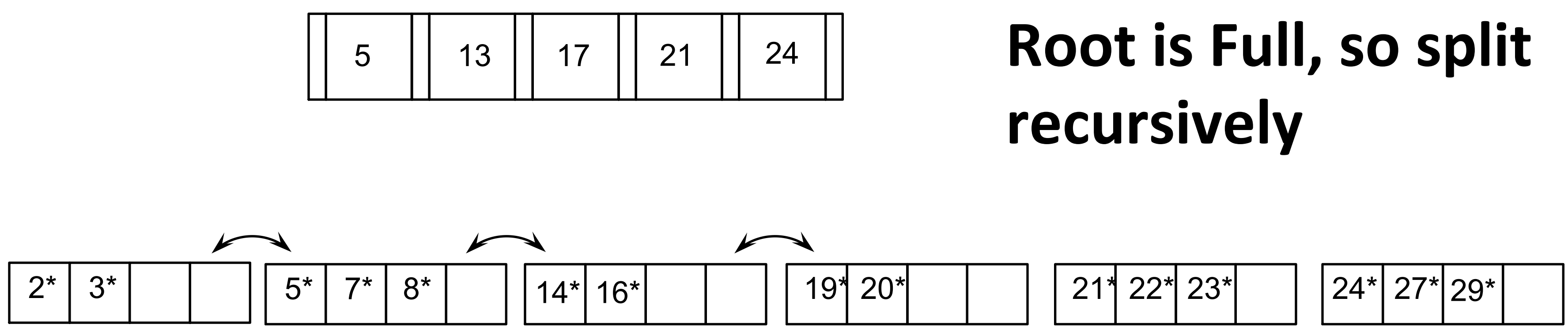


Example B+ Tree - Inserting 21*



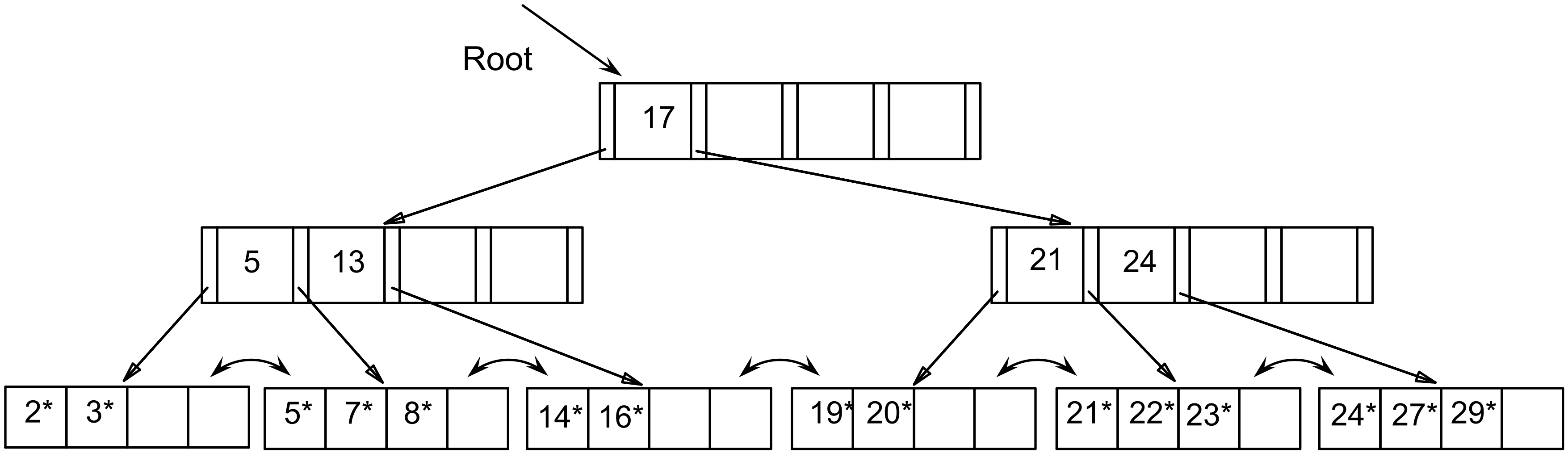
Example B+ Tree - Inserting 21*





Root is Full, so split recursively

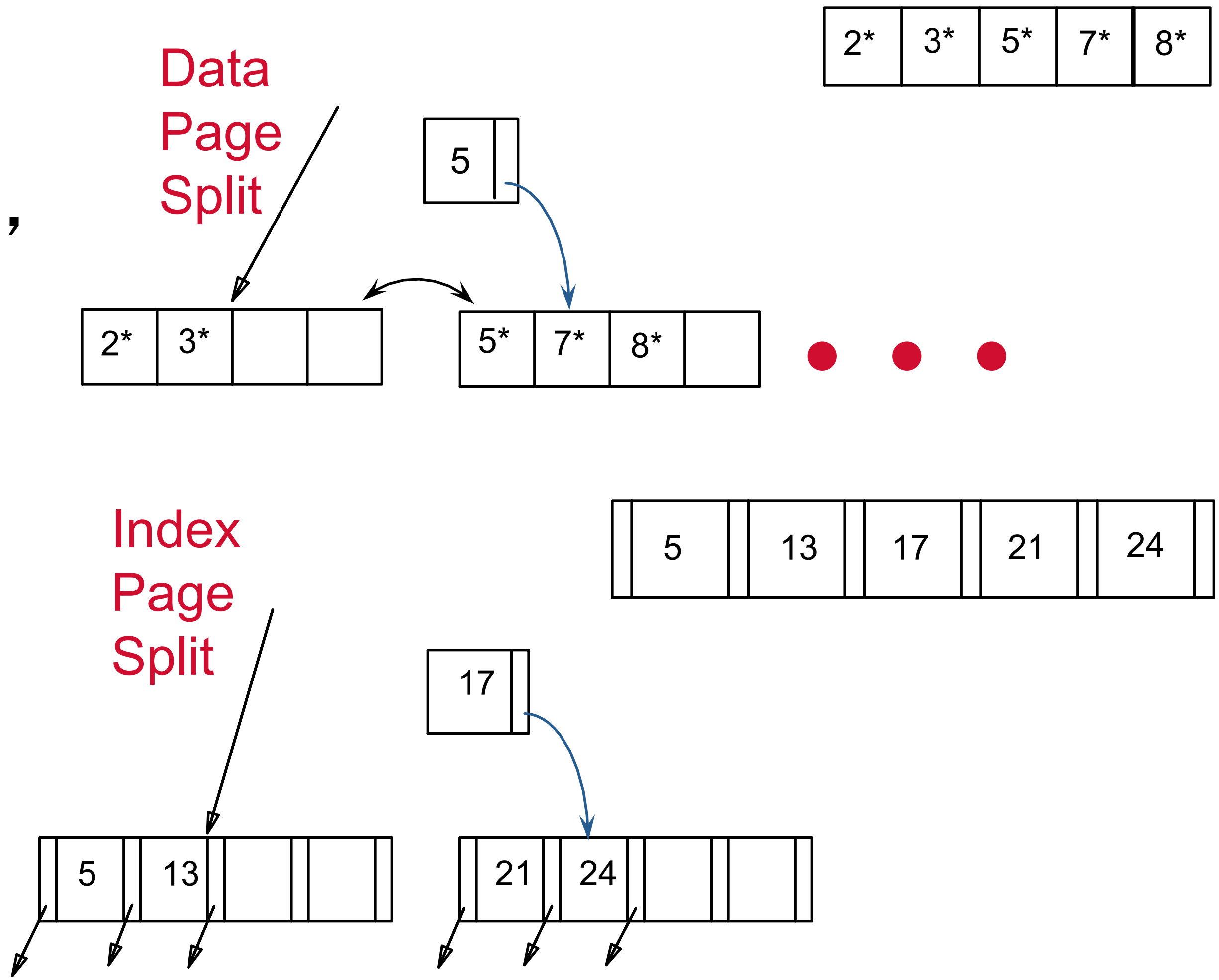
Example B+ Tree: Recursive split



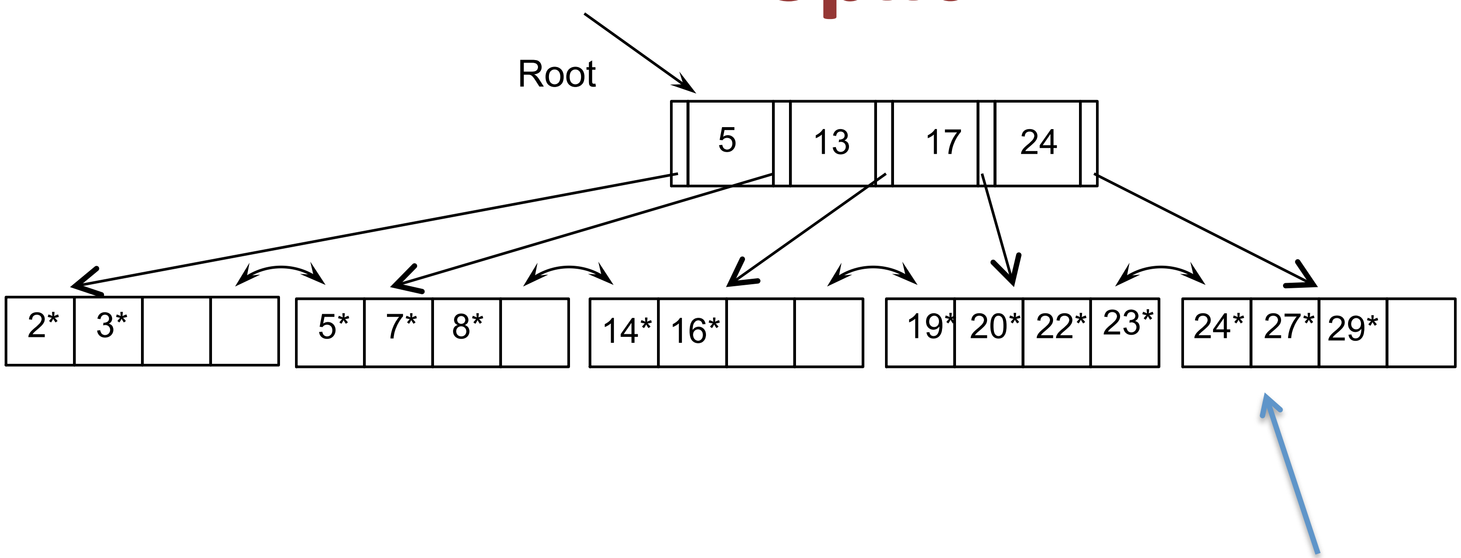
- Notice that root was also split, increasing height.

Example: Data vs. Index Page Split

- leaf: ‘copy’
- non-leaf: ‘push’
- why not ‘copy’ @ non-leaves?

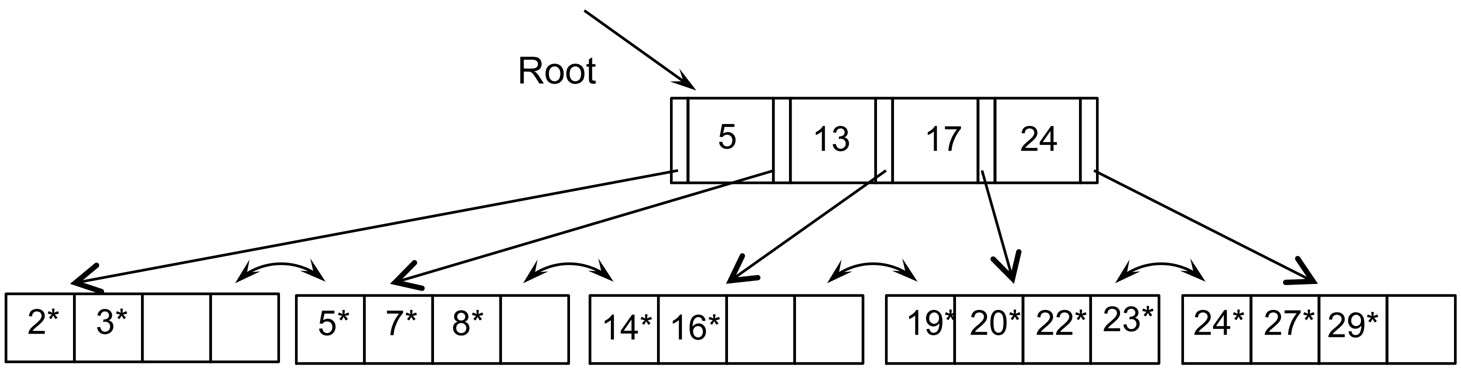


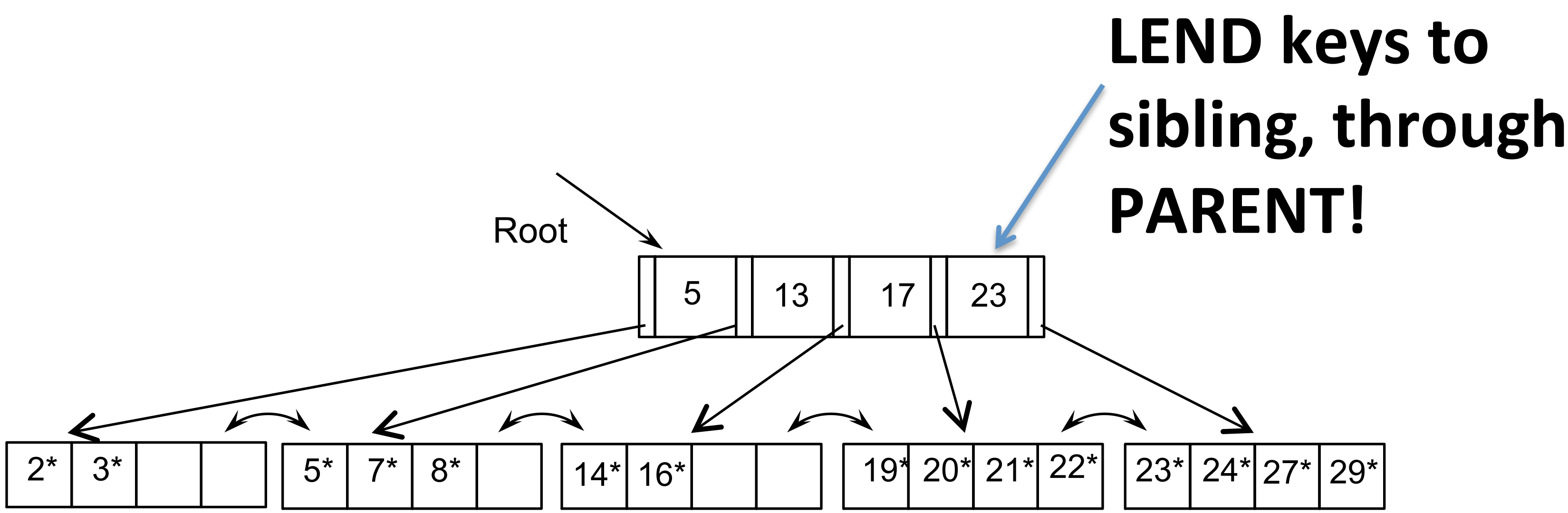
Same Inserting 21*: The Deferred Split



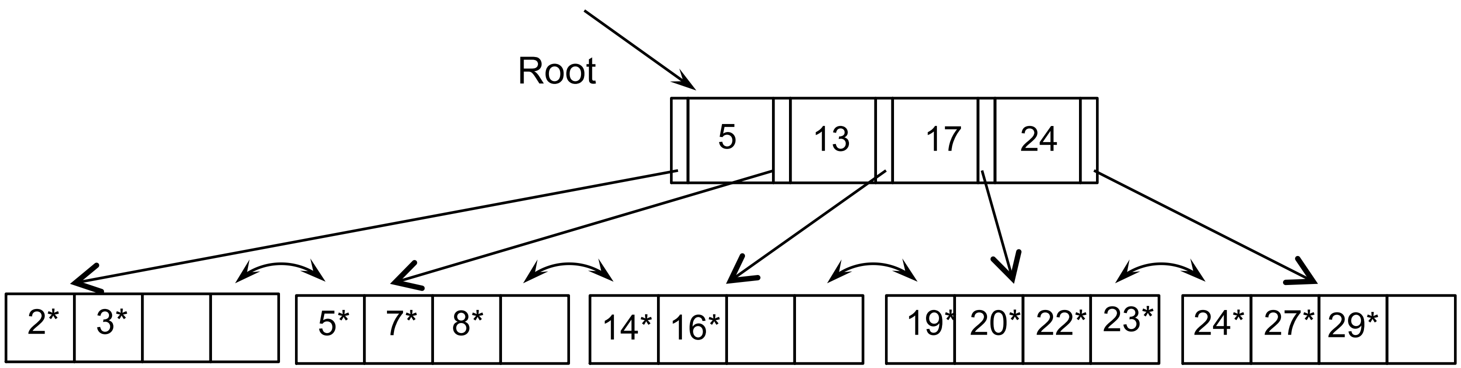
Note this has free space. So…

Inserting 21*: The Deferred Split

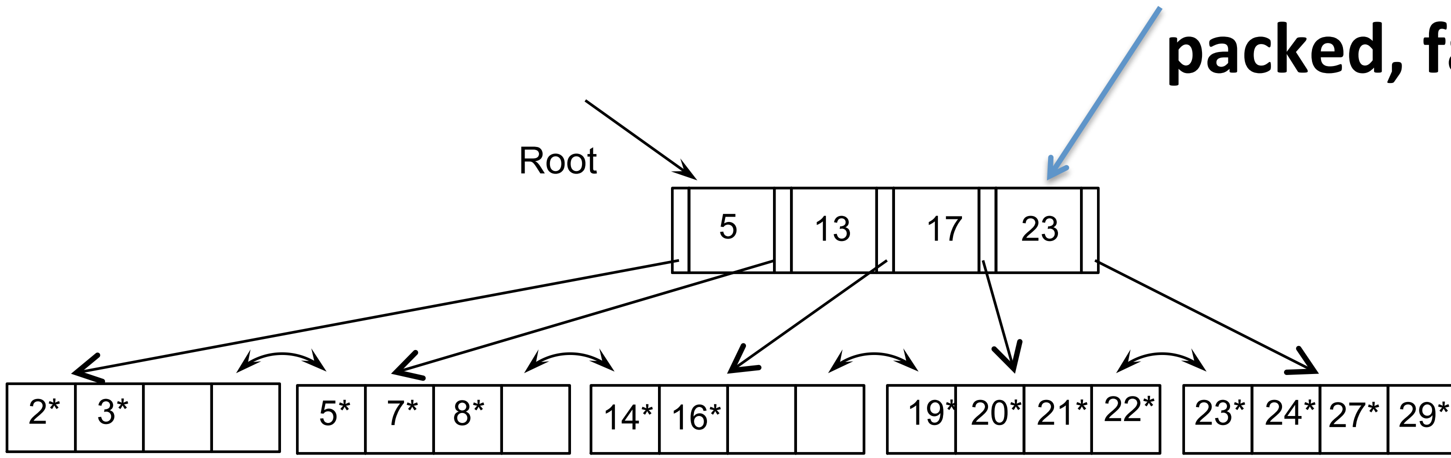




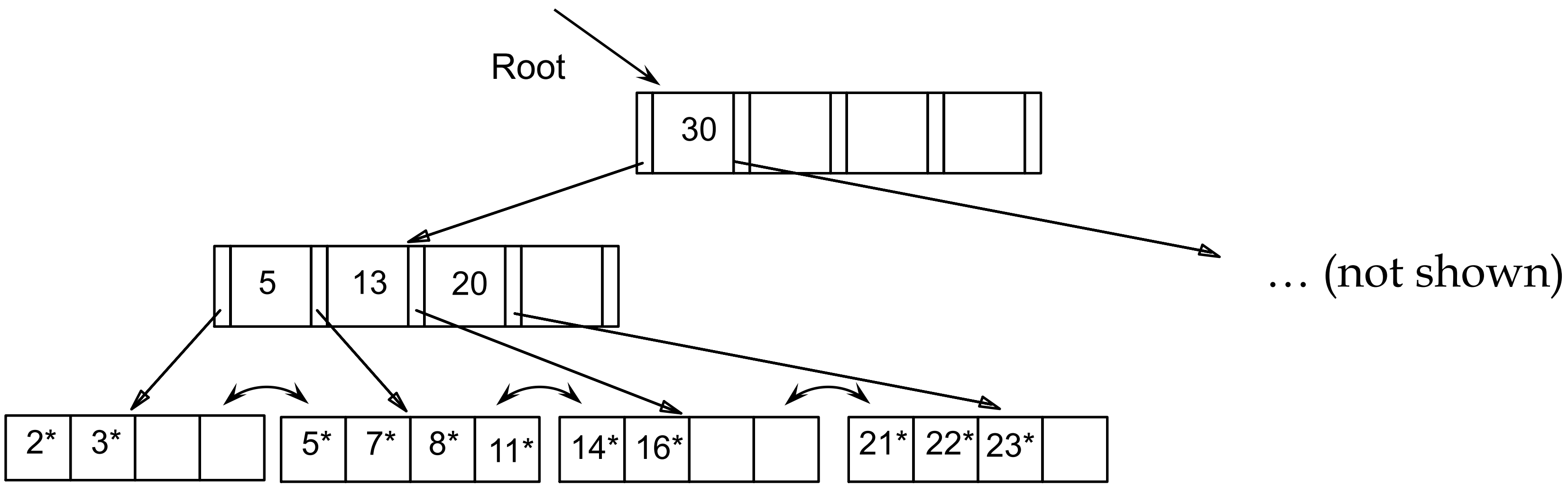
Inserting 21*: The Deferred Split



Shorter, more packed, faster tree



Insertion examples for you to try

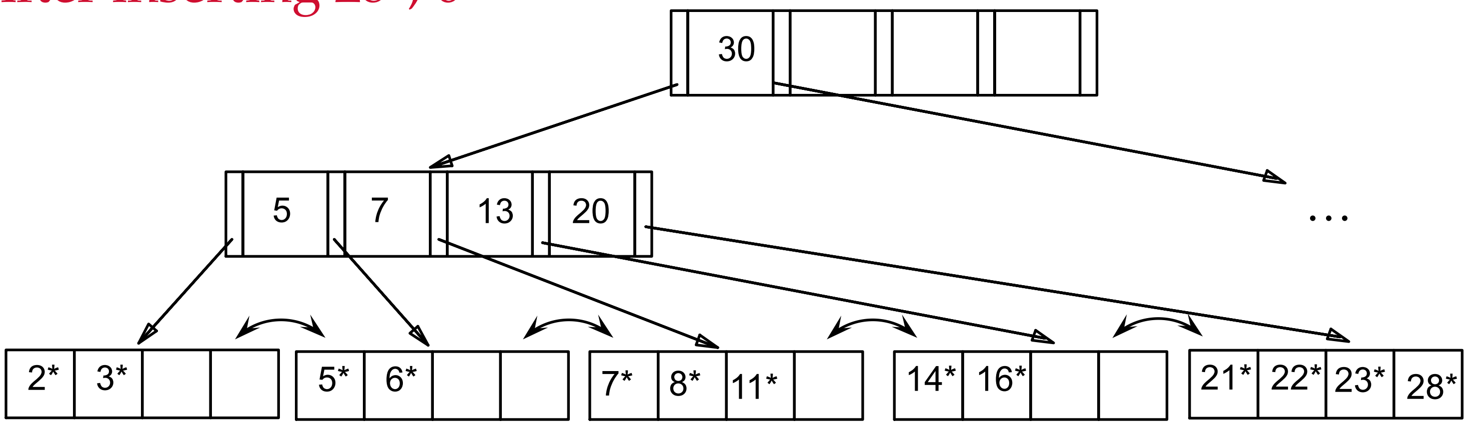


Insert the following data entries (in order): 28*, 6*, 25*

![VirginiaTech]

Answer…

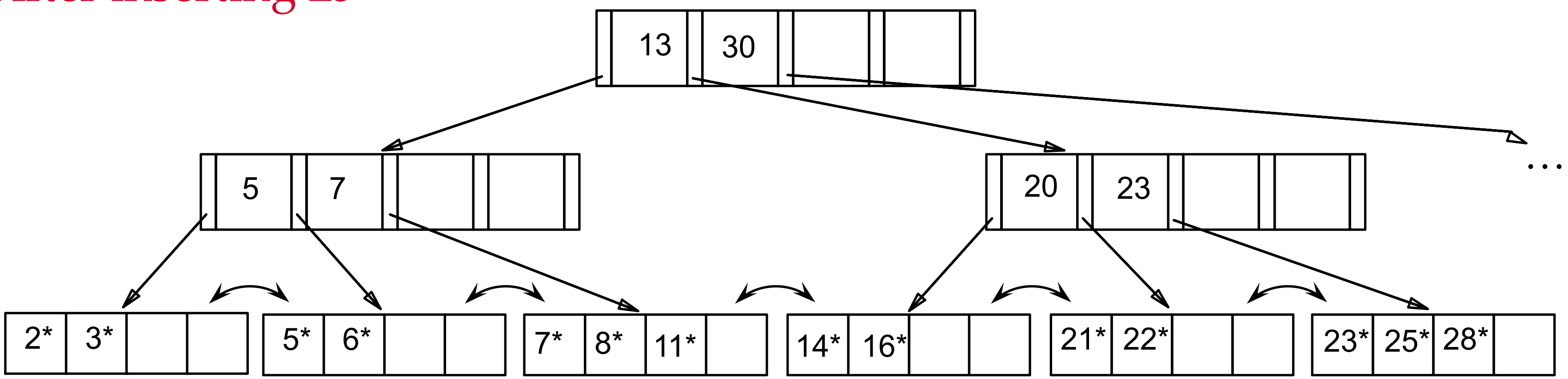
After inserting 28*, 6*



After inserting 25*

Answer…

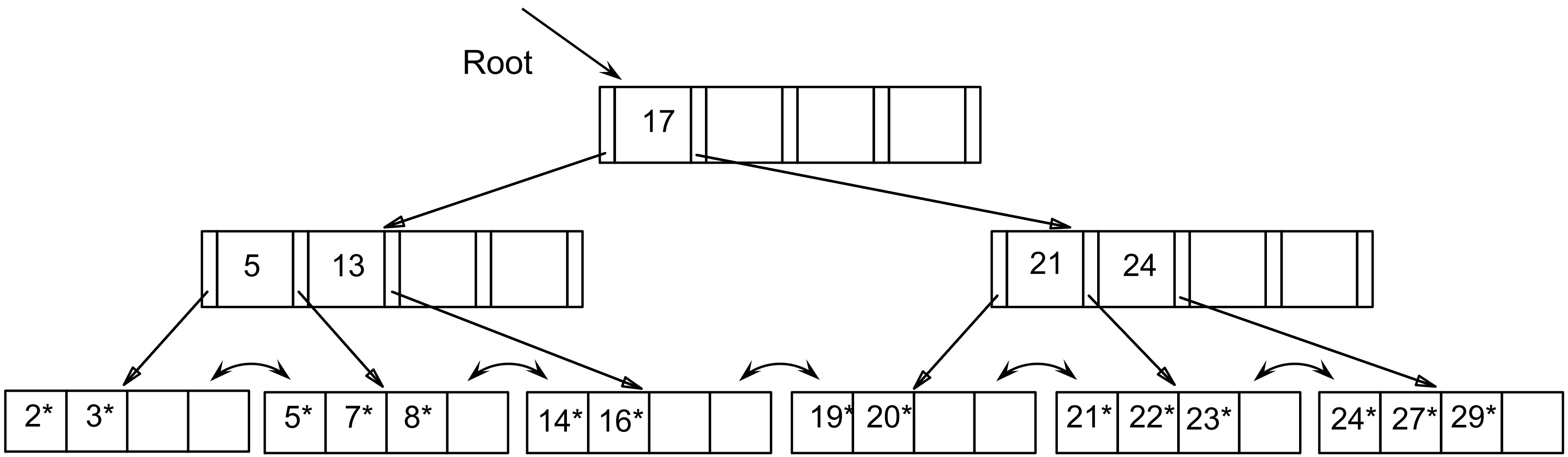
After inserting 25*



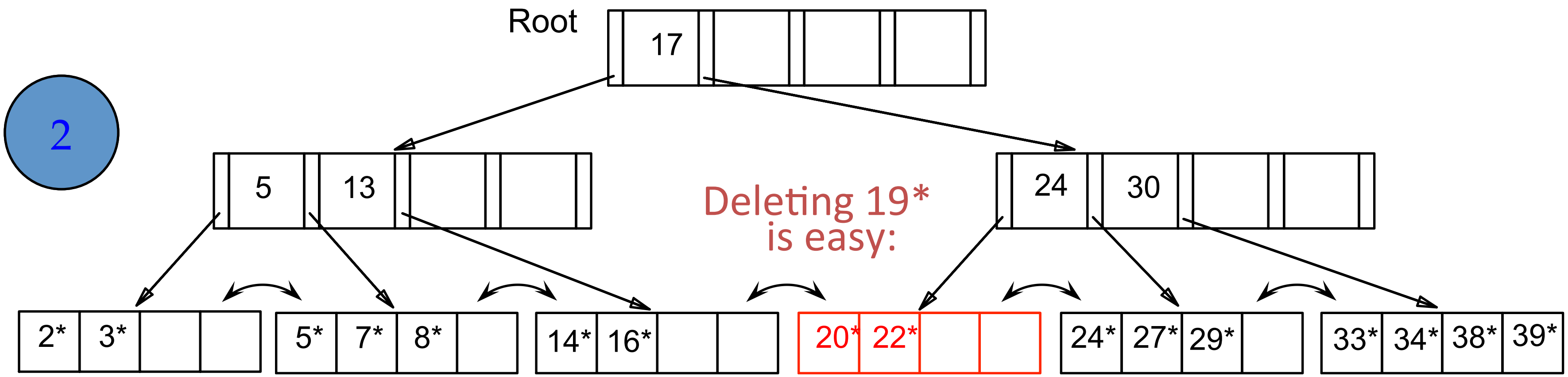
Deleting a Data Entry from a B+ Tree

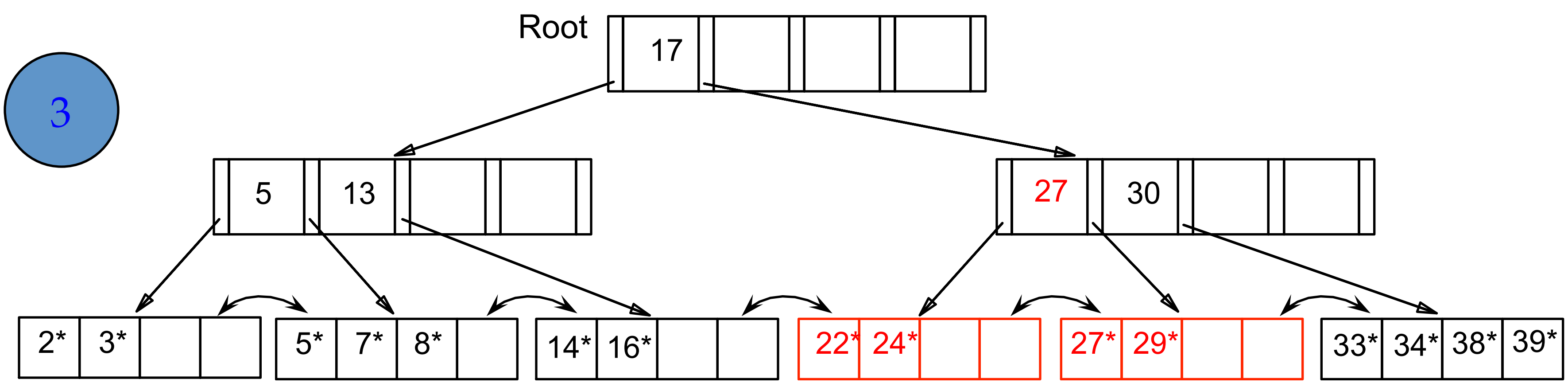
- Start at root, find leaf L where entry belongs.
- Remove the entry.
  - If L is at least half-full, done!
  - If L underflows
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
    - If re-distribution fails, merge L and sibling.
      - update parent
      - and possibly merge, recursively

Deletion from B+Tree



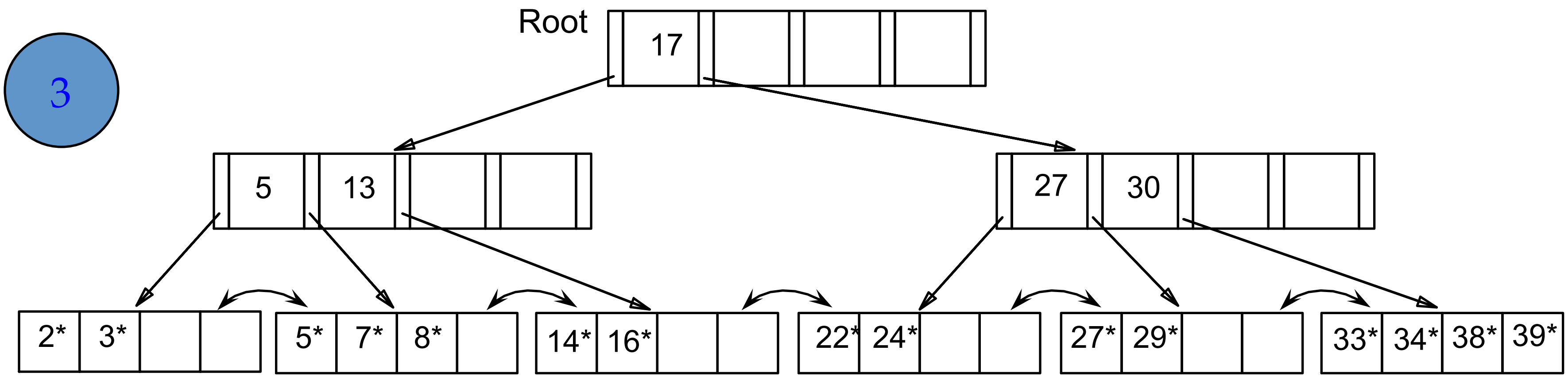
Example: Delete 19* & 20*

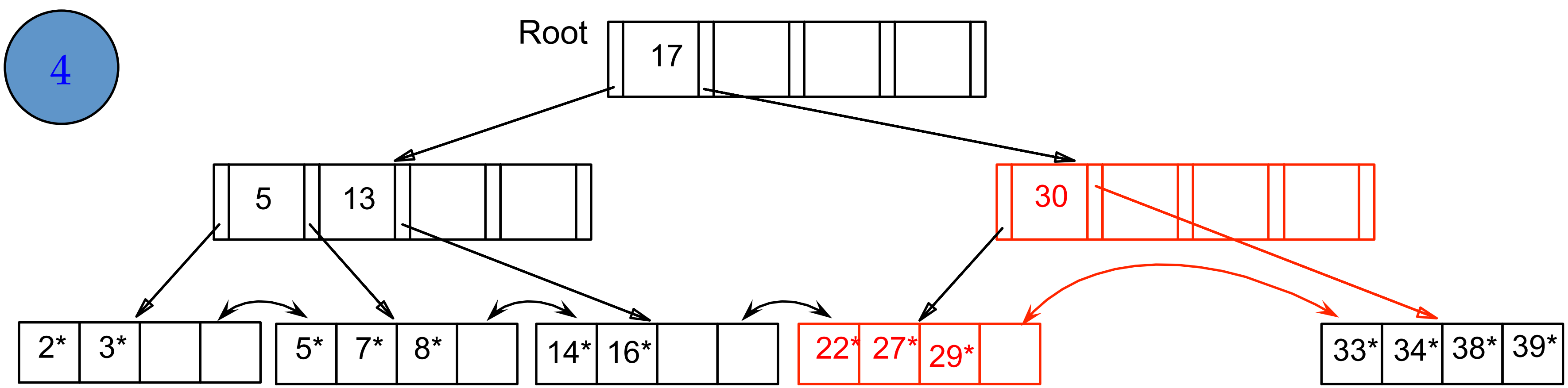




- Deleting 20* -> re-distribution (notice: 27 copied up)

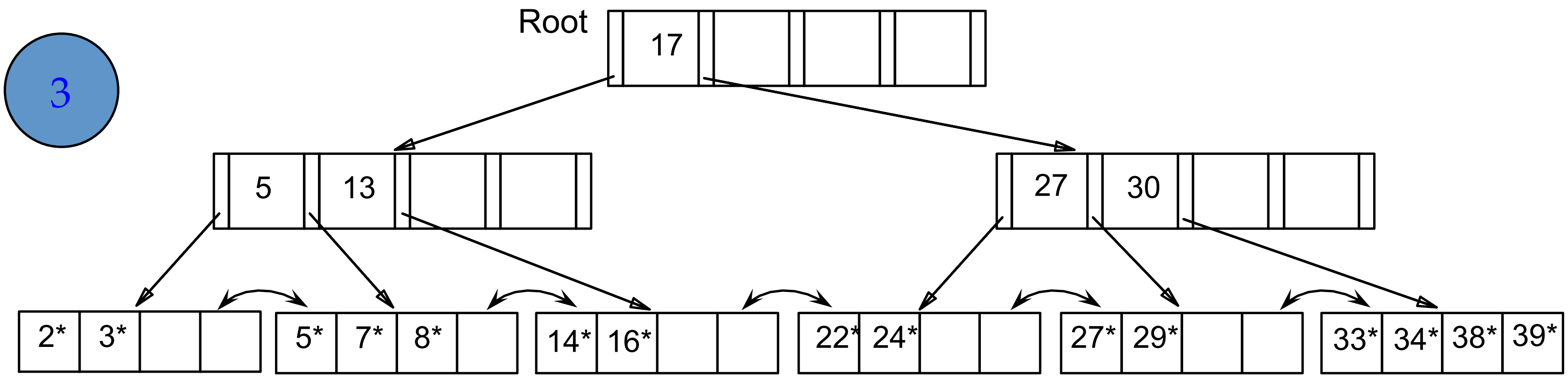
... And Then Deleting 24*

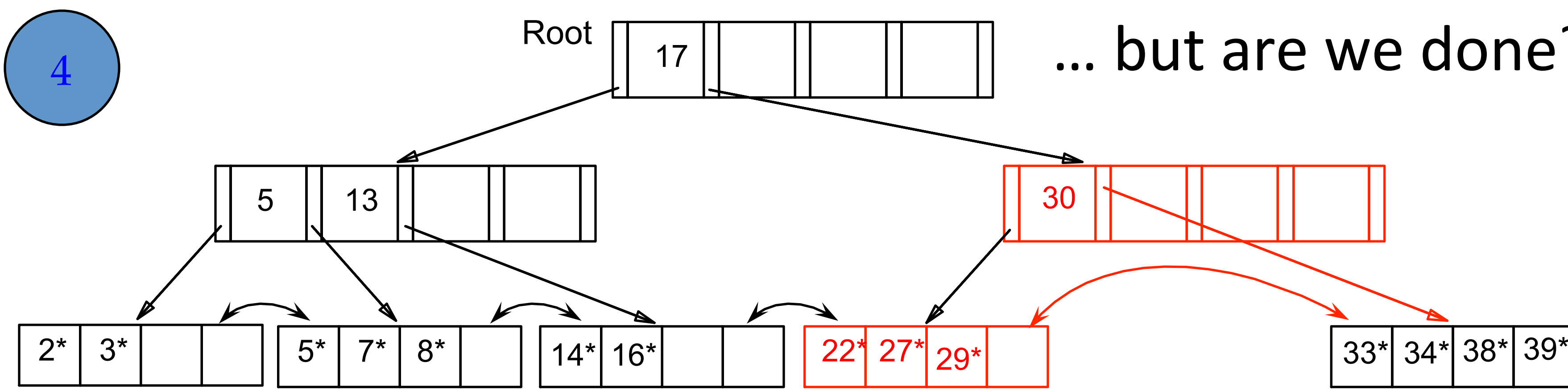




- Must merge leaves: OPPOSITE of insert

... And Then Deleting 24*

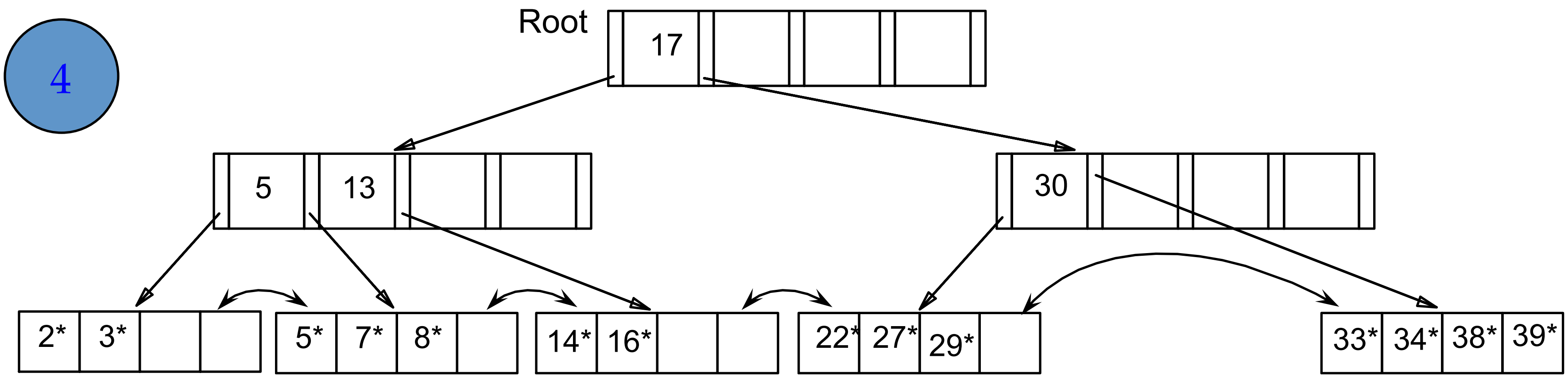


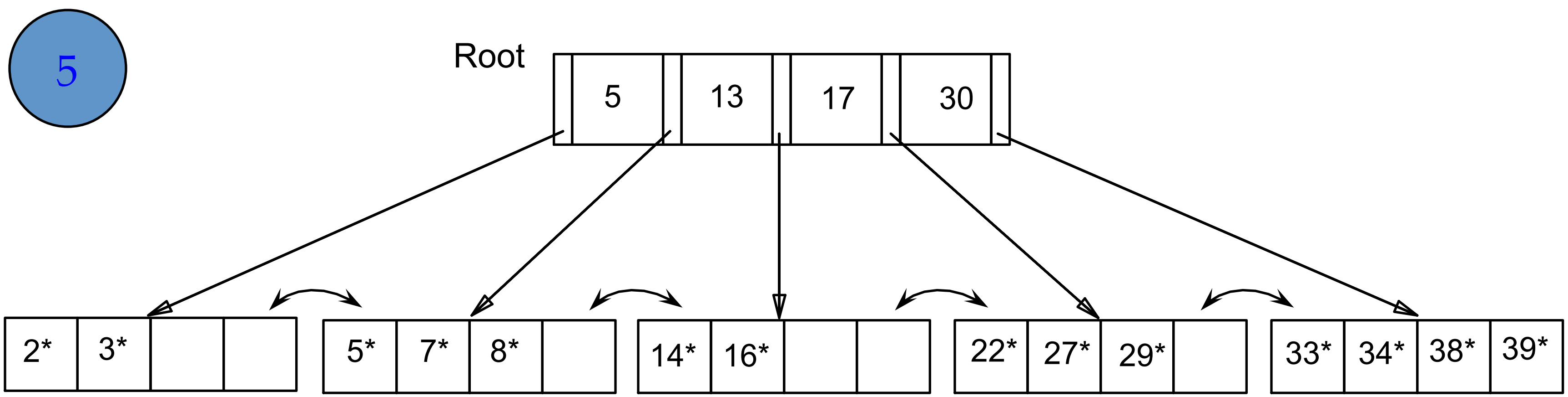


... but are we done??

- Must merge leaves: OPPOSITE of insert

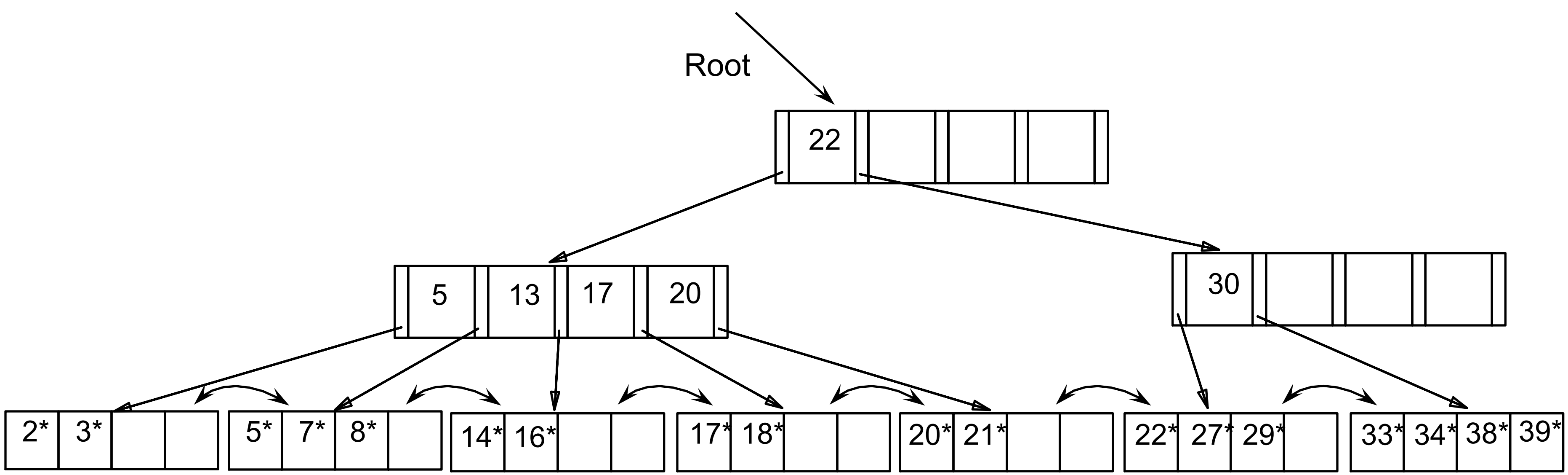
... Merge Non-Leaf Nodes, Shrink Tree





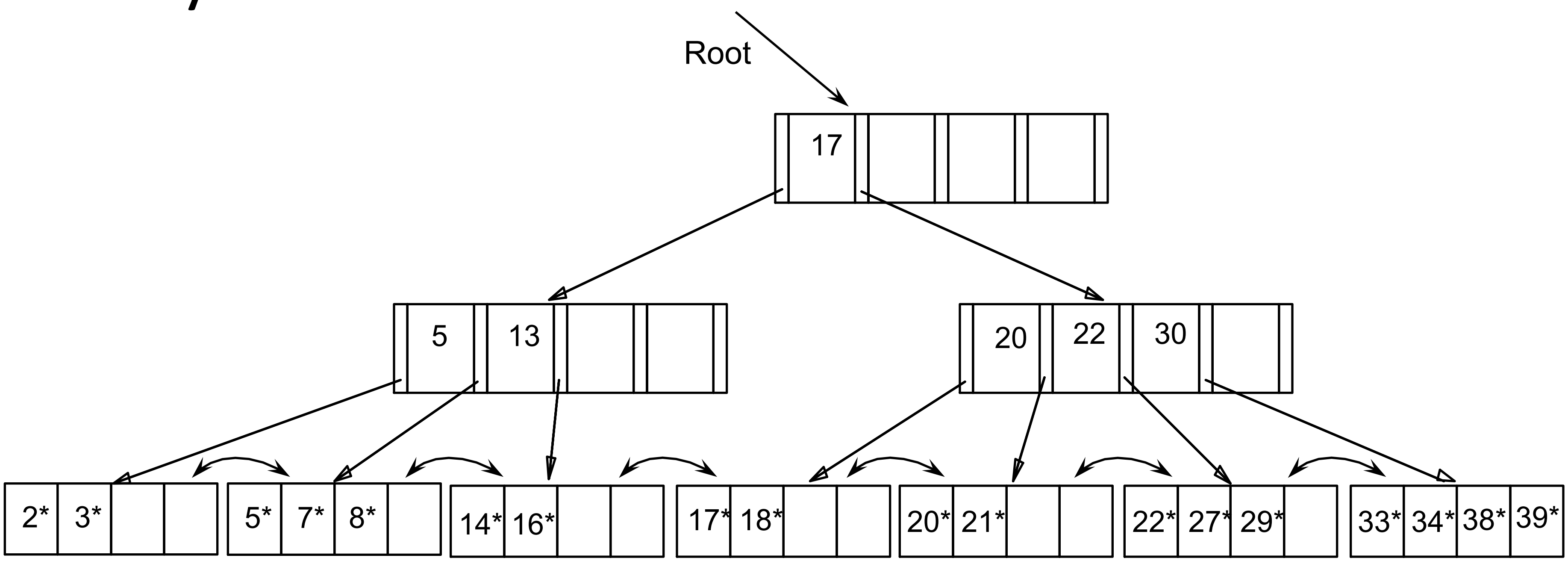
Example of Non-leaf Re-distribution

- Tree is shown below during deletion of 24*.
- Now, we can re-distribute keys



After Re-distribution

- need only re-distribute ‘20’; did ‘17’, too
- why would we want to re-distribute more keys?



VirginiaTech

Main observations for deletion

- If a key value appears twice (leaf + nonleaf), the above algorithms delete it from the leaf, only
- why not non-leaf, too?

VirginiaTech

Main observations for deletion

- If a key value appears twice (leaf + nonleaf), the above algorithms delete it from the leaf, only
- why not non-leaf, too?
- 'lazy deletions' - in fact, some vendors just mark entries as deleted (~ underflow),
  - and reorganize/compact later

Recap: main ideas

- on overflow, split (and 'push', or 'copy')
  - or consider deferred split
- on underflow, borrow keys; or merge
  - or let it underflow...

B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 2*100*0.67 = 134
- Typical capacities:
  - Height 4: 1334 = 312,900,721 entries
  - Height 3: 1333 =   2,406,104 entries

B+ Trees in Practice

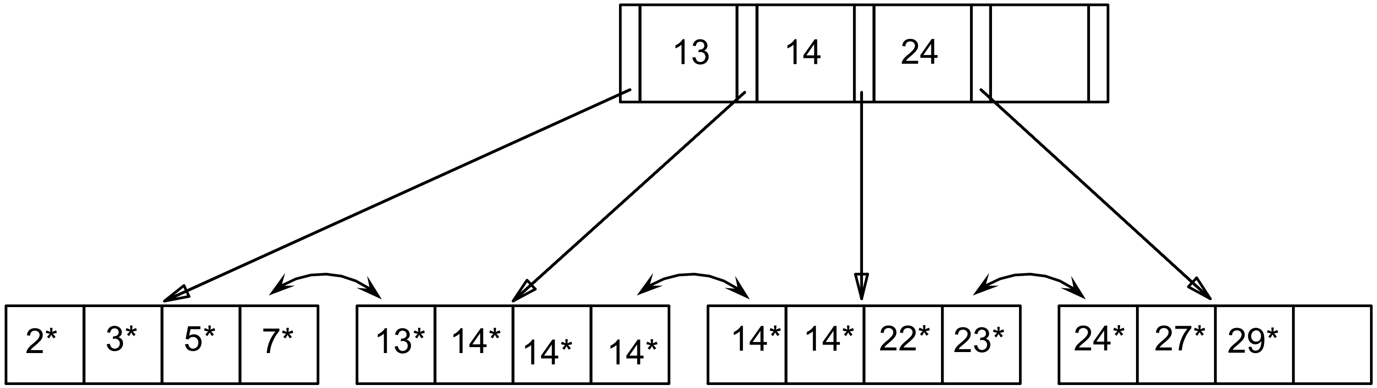
- Can often keep top levels in buffer pool:
  - Level 1 = 1 page = 8 KB
  - Level 2 = 134 pages = 1 MB
  - Level 3 = 17,956 pages = 140 MB

B+ trees with duplicates

- Everything so far: assumed unique key values
- How to extend B+-trees for duplicates?
  - Alt. 2: <key, rid>
  - Alt. 3: <key, {rid list}>
- 2 approaches, roughly equivalent

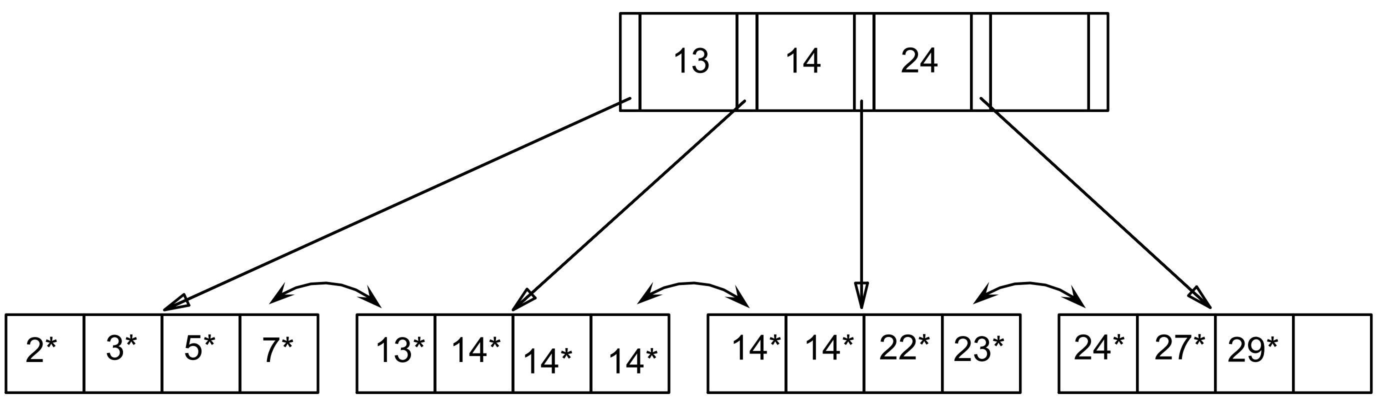
B+ trees with duplicates

- approach#1: repeat the key values, and extend B+ tree algo’s appropriately - eg. many ‘14’s



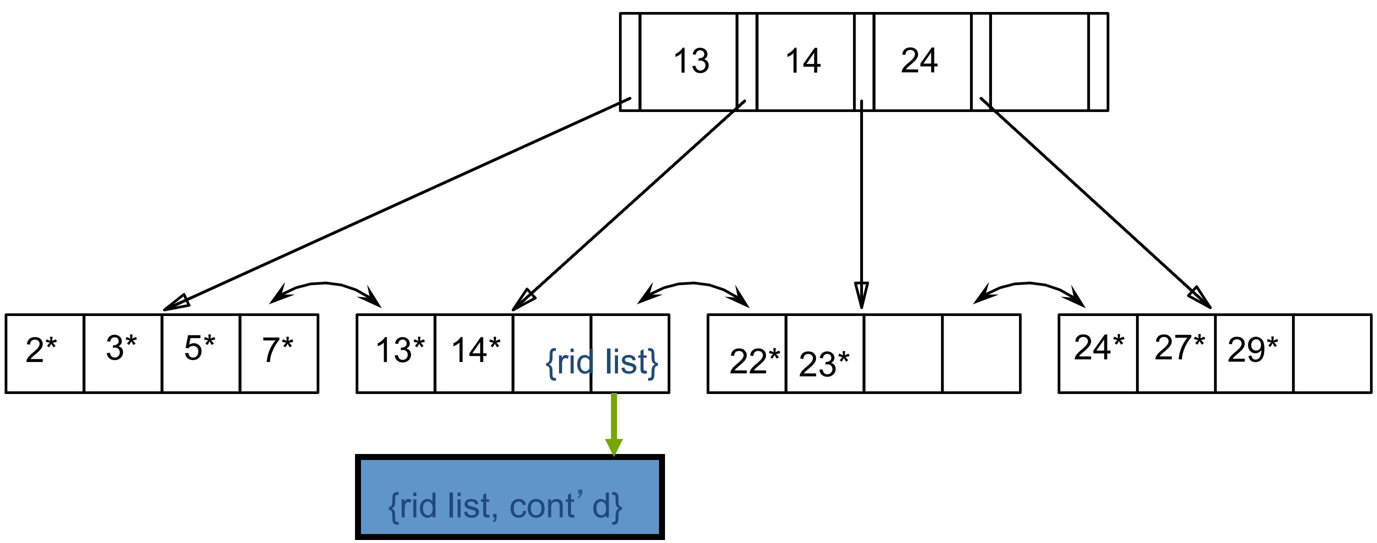
B+ trees with duplicates

- approach#1: subtle problem with deletion:
- treat rid as part of the key, thus making it unique



B+ trees with duplicates

- approach#2: store each key value: once
- but store the {rid list} as variable-length field (and use overflow pages, if needed)

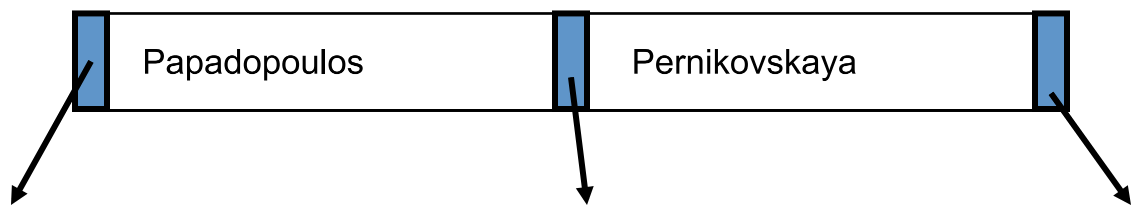


B+trees in Practice

- prefix compression;
- bulk-loading;
- ‘order’

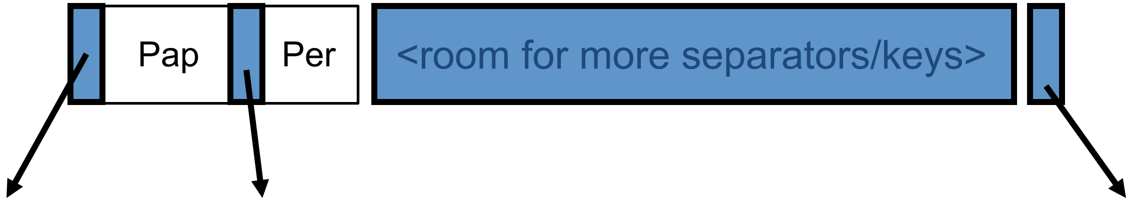
Prefix Key Compression

- Important to increase fan-out. (Why?)
- Key values in index entries only `direct traffic' ; can often compress them.



Prefix Key Compression

- Important to increase fan-out. (Why?)
- Key values in index entries only `direct traffic' ; can often compress them.

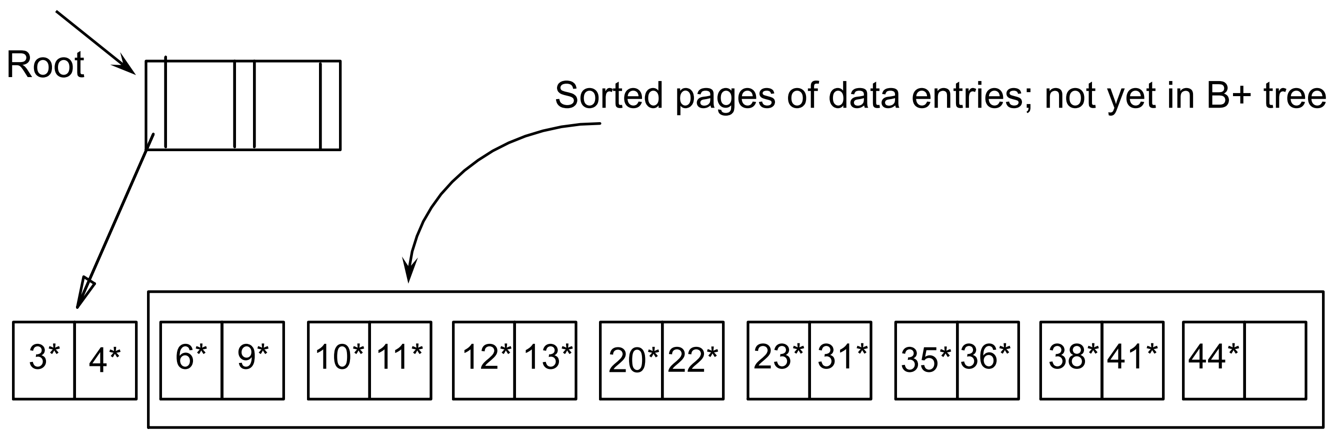


Bulk Loading of a B+ Tree

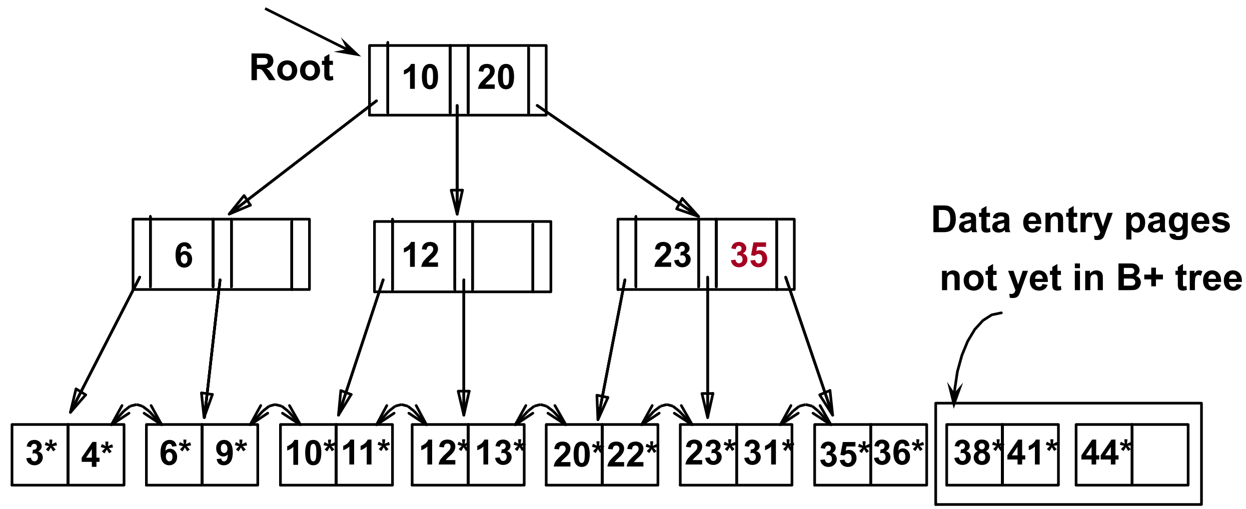
- In an empty tree, insert many keys
- Why not one-at-a-time?
  - – Too slow!

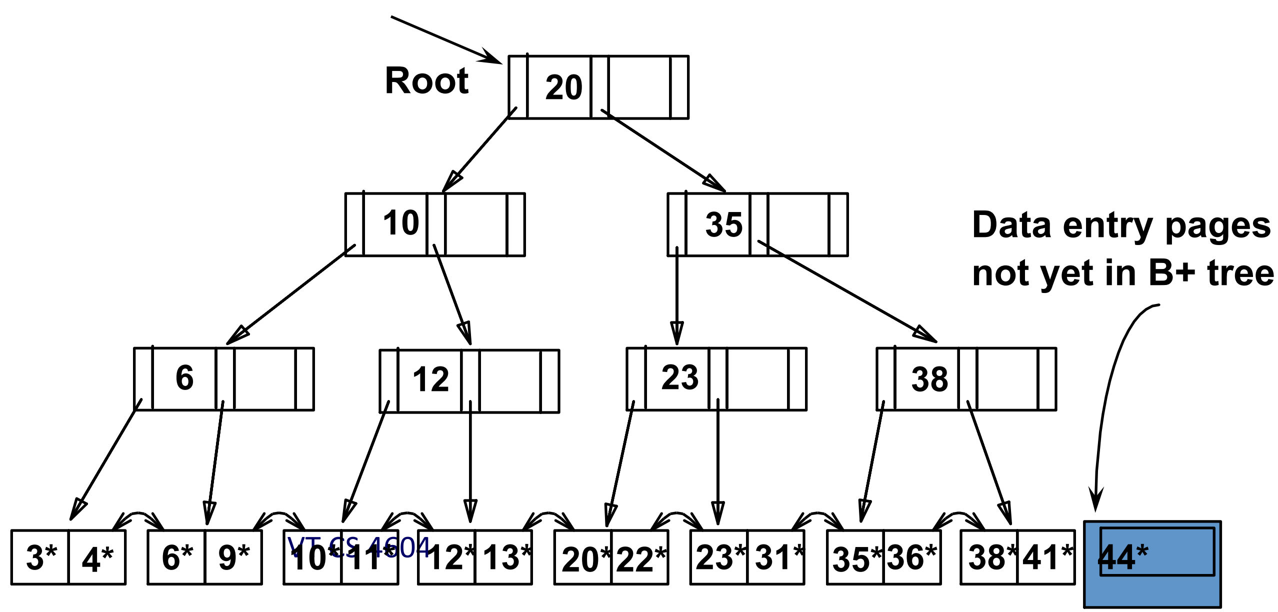
Bulk Loading of a B+ Tree

- Initialization: Sort all data entries
- scan list; whenever enough for a page, pack
- <repeat for upper level>



Bulk Loading of a B+ Tree





A Note on `Order'

- Order (d) concept replaced by physical space criterion in practice (`at least half-full').
- Why do we need it?
  - Index pages can typically hold many more entries than leaf pages.
  - Variable sized records and search keys mean different nodes will contain different numbers of entries.
  - Even with fixed length fields, multiple records with the same search key value (duplicates) can lead to variable-sized data entries (if we use Alternative (3)).

A Note on `Order’

- Many real systems are even sloppier than this: they allow underflow, and only reclaim space when a page is completely empty.
- (what are the benefits of such ‘slopiness’?)

Conclusions

- B+tree is the prevailing indexing method
- Excellent, O(logN) worst-case performance for ins/del/search; (~3-4 disk accesses in practice)
- guaranteed 50% space utilization; avg 69%

Conclusions

- Can be used for any type of index: primary/secondary, sparse (clustering), or dense (non-clustering)
- Several fine-extensions on the basic algorithm
  - deferred split; prefix compression; (underflows)
  - bulk-loading
  - duplicate handling