CS150: Database & Datamining
Lecture 11: Files&Indexes

ShanghaiTech-SIST

Spring 2019

Acknowledgement: Slides are adopted from the Berkeley course CS186 by Joey Gonzalez and Joe Hellerstein, Stanford CS145 by Peter Bailis.

Announcements

1. Homework 2 be released this week
2. Mid-term: to be announced next time

Today’s Lecture

1. Indexes: Motivations & Basics

2. B+ Trees

1. Indexes: Motivations & Basics

Index Motivation

Person(name, age)

- Suppose we want to search for people of a specific age

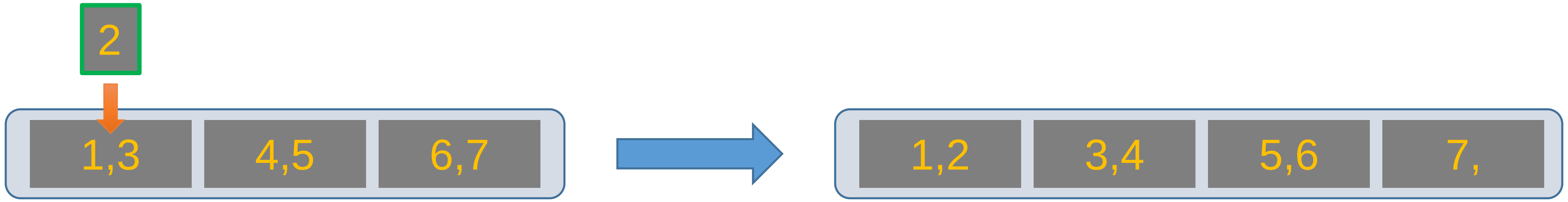
- ***First idea:*** Sort the records by age… we know how to do this fast!

- How many IO operations to search over ***N=B*R sorted*** records?
  - Simple scan: ***O(BD)***
  - Binary search: ***O($\log_2 B$*D)***

Could we get even cheaper search? E.g. go from $\log_2 B$ $\rightarrow \log_{200} B$?

Index Motivation

- What about if we want to **insert** a new person, but keep the list sorted?



- We would have to potentially shift ***N*** records, requiring up to **~ 2*B** IO operations!
  - We could leave some "slack" in the pages…

Could we get faster insertions?

Index Motivation

- What about if we want to be able to search quickly along multiple attributes (e.g. not just age)?
  - We could keep multiple copies of the records, each sorted by one attribute set… this would take a lot of space

Can we get fast search over multiple attribute (sets) without taking too much space?

We'll create separate data structures called *indexes* to address all these points

Further Motivation for Indexes: NoSQL!

- NoSQL engines are (basically) ***just indexes!***
  - A lot more is left to the user in NoSQL… one of the primary remaining functions of the DBMS is still to provide index over the data records, for the reasons we just saw!
  - Sometimes use B+ Trees (covered next), sometimes hash indexes (not covered here)

Indexes are critical across all DBMS types

Indexes: High-level

- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Search key properties
    - Any subset of fields
    - is **not** the same as *key of a relation*

- *Example:*

Product(name, maker, price)

On which attributes would you build indexes?

More precisely

- An *index* is a **data structure** mapping search keys to sets of rows in a database table
    - Provides efficient lookup & retrieval by search key value- usually much faster than searching through all the rows of the database table
- An index can store the full rows it points to (*primary index*) or pointers to those rows (*secondary index*)

Operations on an Index

- Search: Quickly find all records which meet some *condition on the search key attributes*
  - More sophisticated variants as well. Why?

- Insert / Remove entries
  - Bulk Load / Delete. Why?

Indexing is one the most important features provided by a database for performance

Kinds of Lookups Supported?

- Basic Selection: <key> <op> <constant>
  - Equality selections (op is =)?
  - Range selections (op is one of <, >, <=, >=, BETWEEN)

- More exotic selections:
  - 2-dimensional ranges ("east of Berkeley and west of Truckee and North of Fresno and South of Eureka")
  - 2-dimensional radii ("within 2 miles of Soda Hall")
  - Common **n-dimensional index**: *R-tree, KD-Tree*
    - Beware of the ***curse of dimensionality***
  - Ranking queries ("10 restaurants closest to Berkeley")
  - Regular expression matches, genome string matches, etc.
  - See http://en.wikipedia.org/wiki/GiST for more

Search Key: *Any* Subset of Columns

- Search key needn't be a key of the relation
  - Recall: key of a relation must be unique (e.g., SSN)
  - Search keys don't have to be unique
- **Composite Keys**: more than one column
  - Think: Phone Book <Last Name, First>
  - **Lexicographic order**
  - <Age, Salary>:
    - ✓ Age = 31 & Sal = 400
    - ✓ Age = 55 & Sal > 200
    - ✗ Age > 31 & Sal = 400
    - ✓ Age = 31
    - ✓ Age > 31
    - ✗ Sal = 300

SSN	Last Name	First Name	Age	Salary
123	Adams	Elmo	31	$400
443	Grouch	Oscar	32	$300
244	Oz	Bert	55	$140
134	Sanders	Ernie	55	$400

✗ Means that the index is unable to exclude all entries that are not in the result set.

Data Entries: How are they stored?

- What is the representation of data in the index?
  - Actual data or pointer(s) to the data
- How is the data stored in the data file?
  - Clustered or unclusted with respect to the indexed
- Big Impact on Performance

How is data stored in the index

- Three alternatives:
  1. **By Value:** actual data record (with key value **k**)
  2. **By Reference:** <**k**, rid of matching data record>
  3. **By List of Refs.:** <**k**, list of rids of *all* matching data records>

- Choice is orthogonal to the indexing technique.
  - B+ trees, hash-based structures, R trees, GiSTs, …

- Can have multiple (different) indexes per file.
  - E.g. file sorted by *age*, with a hash index on *salary* and a B+tree index on *name*.

Alternatives for Data Entries (Contd.)

Alternative 1 (By Value):
Actual data record (with key value ***k***)

- Index as a file organization for records
  - Alongside heap files or sorted files
- No “**pointer lookups**” to get data records
  - Following record ids
- Could a single relation have multiple indexes of this form?
  - Probably but it would be a bad idea. Why?

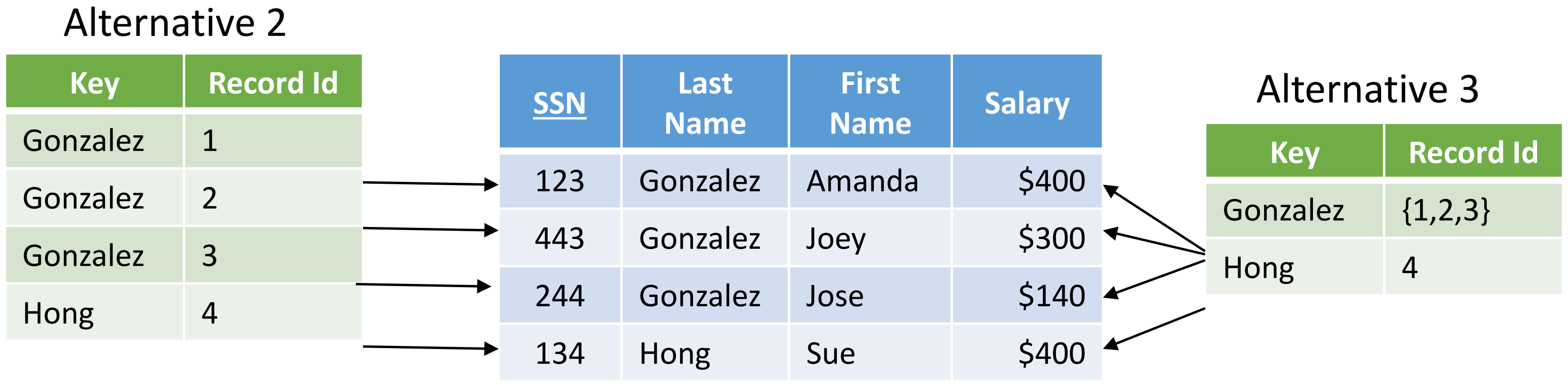
Alternatives for Data Entries (Contd.)

Alternative 2 (By Reference)

<**k**, rid of matching data record>

and Alternative 3 (By List of References)

<**k**, list of rids of matching data records>



Alternative 2

Key	Record Id
Gonzalez	1
Gonzalez	2
Gonzalez	3
Hong	4

SSN	Last Name	First Name	Salary
123	Gonzalez	Amanda	$400
443	Gonzalez	Joey	$300
244	Gonzalez	Jose	$140
134	Hong	Sue	$400

Alternative 3

Key	Record Id
Gonzalez	{1,2,3}
Hong	4

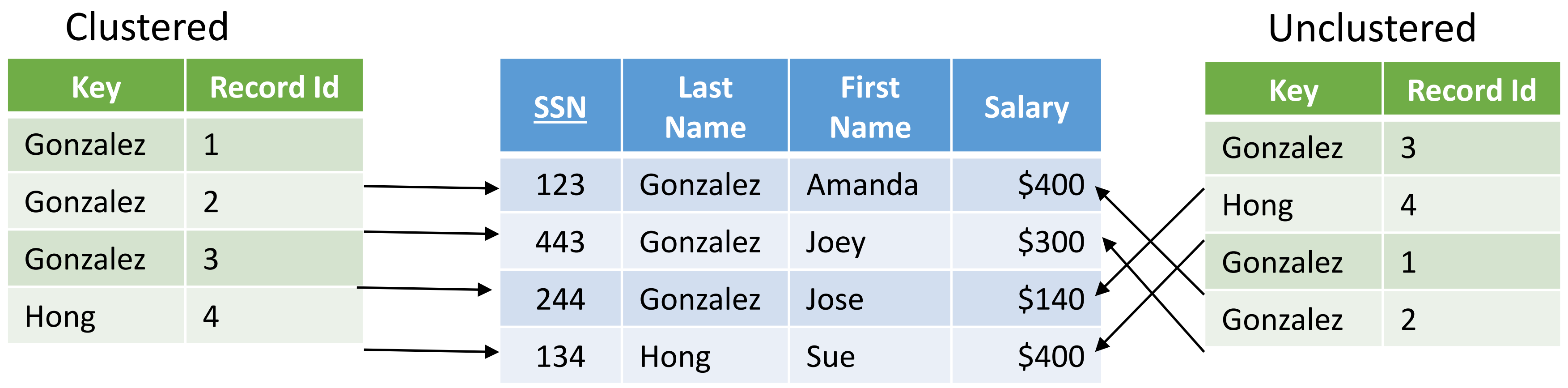
- Alts. 2 or 3 needed to support multiple indexes per relation!
- Alt. 3 more compact than Alt. 2
- For very large rid lists, single data entry spans multiple blocks

Clustered vs. Unclustered Index

- In a clustered index:
    - index data entries are stored in (approximate) order by value of search keys in data records

Clustered vs. Unclustered Index

- In a clustered index:
  - index data entries are stored in (approximate) order by value of search keys in data records



Clustered

Key	Record Id
Gonzalez	1
Gonzalez	2
Gonzalez	3
Hong	4

SSN	Last Name	First Name	Salary
123	Gonzalez	Amanda	$400
443	Gonzalez	Joey	$300
244	Gonzalez	Jose	$140
134	Hong	Sue	$400

Unclustered

Key	Record Id
Gonzalez	3
Hong	4
Gonzalez	1
Gonzalez	2

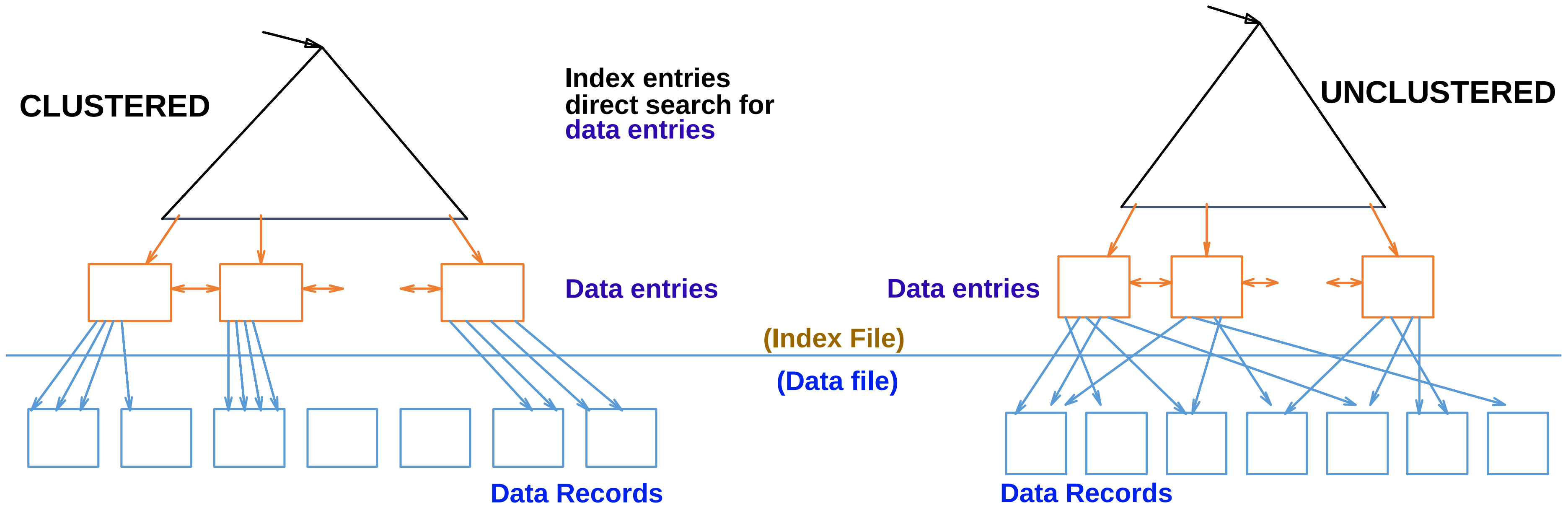
Clustered vs. Unclustered Index

- In a clustered index:
  - index data entries are stored in (approximate) order by value of search keys in data records
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!
  - Can Alternative 1 be un-clustered?
    - No, but clustered does not imply alt. 1 (earlier figure)
- Note: there is another definition of "clustering"
  - **Data Mining/AI**: grouping similar items in n-space

Clustered vs. Unclustered Index

Alternative 2 (by ref) data entries, data records in a Heap file.

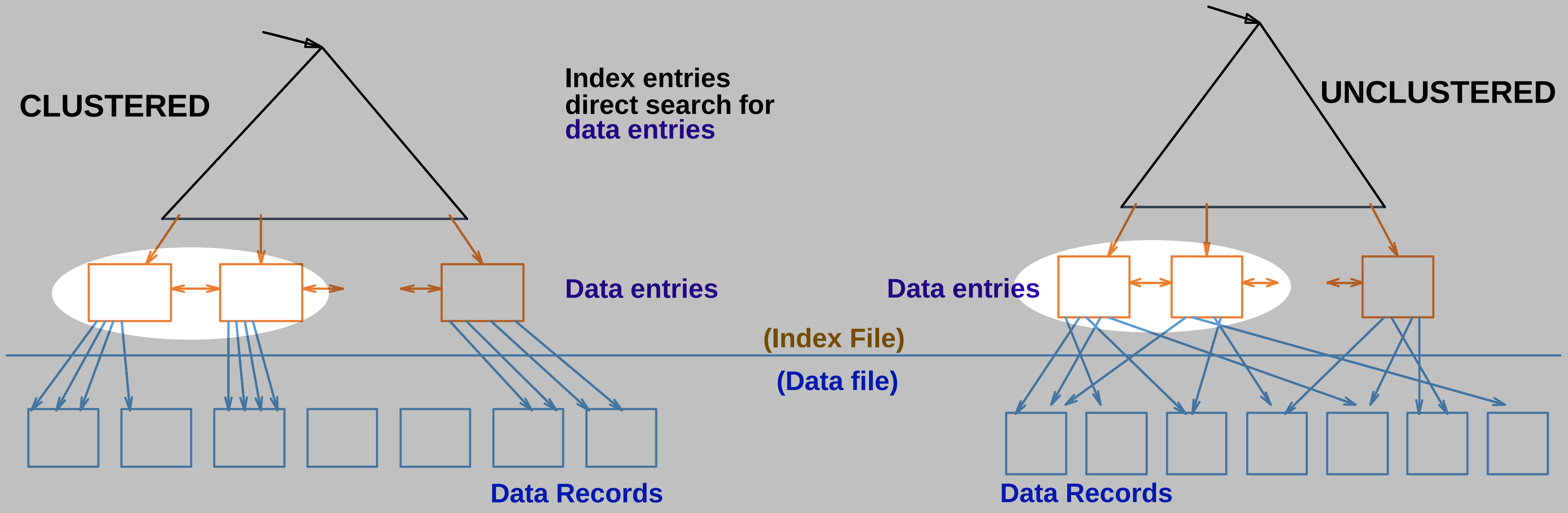
- To build clustered index, first sort the heap file
    - Leave some free space on each block for future inserts
- Overflow blocks may be needed for inserts.
    - Thus, order of data recs is "close to", but not identical to, the sort order.



Clustered vs. Unclustered Index

Alternative 2 (by ref) data entries, data records in a Heap file.

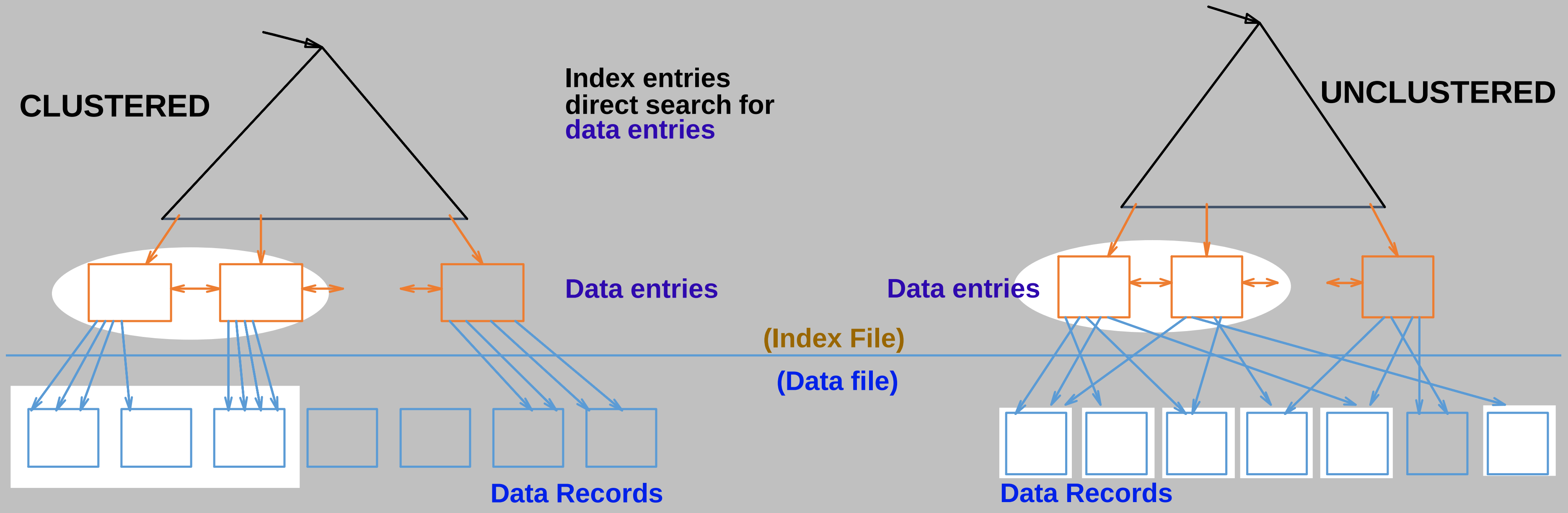
- To build clustered index, first sort the Heap file
  - Leave some free space on each block for future inserts
- Overflow blocks may be needed for inserts.
  - Thus, order of data recs is "close to", but not identical to, the sort order.



Clustered vs. Unclustered Index

Alternative 2 (by ref) data entries, data records in a Heap file.

- To build clustered index, first sort the Heap file
  - Leave some free space on each block for future inserts
- Overflow blocks may be needed for inserts.
  - Thus, order of data recs is "close to", but not identical to, the sort order.



Clustered vs. Unclustered Index

- Recall that for a disk with block access, **sequential IO is much faster than random IO**
- For exact search, no difference between clustered / unclustered
- For range search over N=B*R values: difference between **1 random IO + B sequential IO**, and **B random IO**:
  - A random IO costs ~ 10ms (sequential much much faster)
  - For B = 100,000 - **difference between ~10ms and ~17min!**

Unclustered vs. Clustered Indexes

- Clustered Pros
  - Efficient for range searches
  - Potential locality benefits?
    - Sequential disk access, prefetching, etc.
  - Support certain types of compression
    - More soon on this topic

- Clustered Cons
  - More expensive to maintain
    - Need to update index data structure
    - Solution: on the fly or “lazily” via reorgs
  - Heap file usually only **packed to 2/3** to accommodate inserts

High-level Categories of Index Types

- B-Trees *(covered next)*
  - Very good for range queries, sorted data
  - Some old databases only implemented B-Trees
  - *We will look at a variant called **B+ Trees***

- Hash Tables *(not covered)*
  - There are variants of this basic structure to deal with IO
  - Called ***linear*** or ***extendible hashing-*** IO aware!

The data structures we present here are “IO aware”

Real difference between structures: costs of ops *determines which index you pick and why*

2. B+ Trees

What you will learn about in this section

1. B+ Trees: Basics

2. B+ Trees: Design & Cost

3. Clustered Indexes

B+ Trees

- Search trees
  - B does not mean binary!

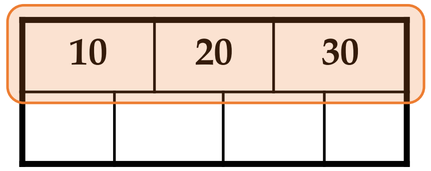
- Idea in B Trees:
  - make 1 node = 1 physical page
  - Balanced, height adjusted tree (not the B either)

- Idea in B+ Trees:
  - Make leaves into a linked list (for range queries)

Why B-trees?

- After a talk at CPM 2013 (24th Annual Symposium on Combinatorial Pattern Matching, Bad Herrenalb, Germany, June 17–19, 2013), Ed McCreight answered a question on B-tree's name by Martin Farach-Colton saying: "Bayer and I were in a lunch time where we get to think a name. And we were, so, B, we were thinking… B is, you know… We were working for Boeing at the time, we couldn't use the name without talking to lawyers. So, there is a B. It has to do with balance, another B. Bayer was the senior author, who did have several years older than I am and had many more publications than I did. So there is another B. And so, at the lunch table we never did resolve whether there was one of those that made more sense than the rest. What really lives to say is: the more you think about what the B in B-trees means, the better you understand B-trees."[3]

B+ Tree Basics

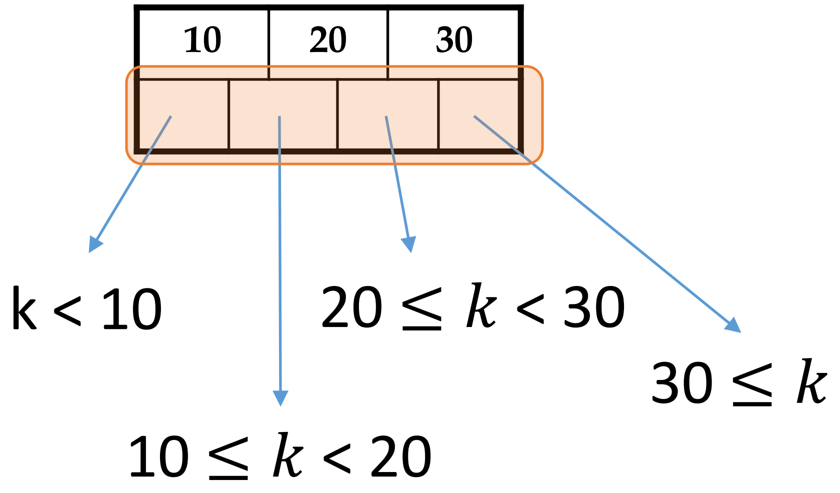


Parameter ***d*** = the degree

Each *non-leaf ("interior")* ***node*** has ≥ d and ≤ 2d ***keys****

except for root node, which can have between **1 and 2d keys*

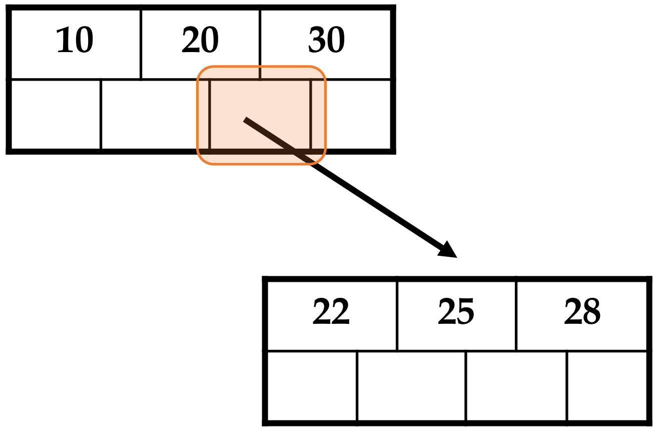
B+ Tree Basics



The *n* keys in a node define *n+1* ranges

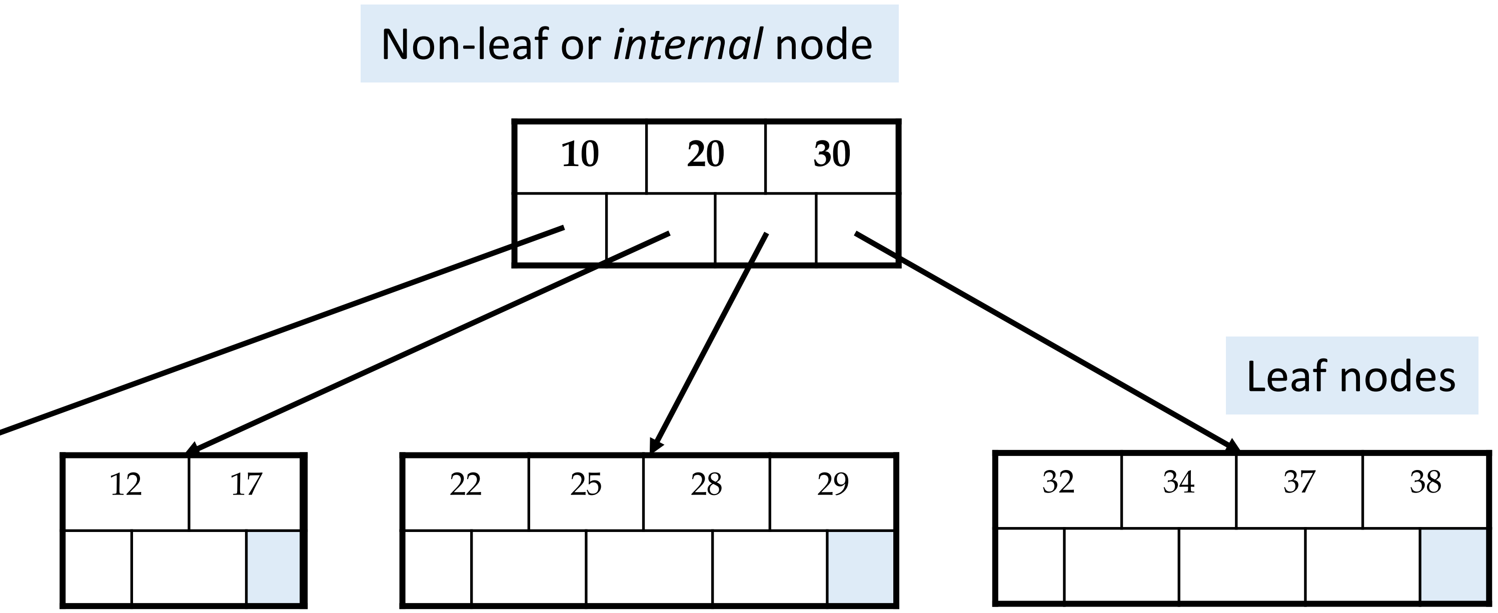
B+ Tree Basics

Non-leaf or *internal* node



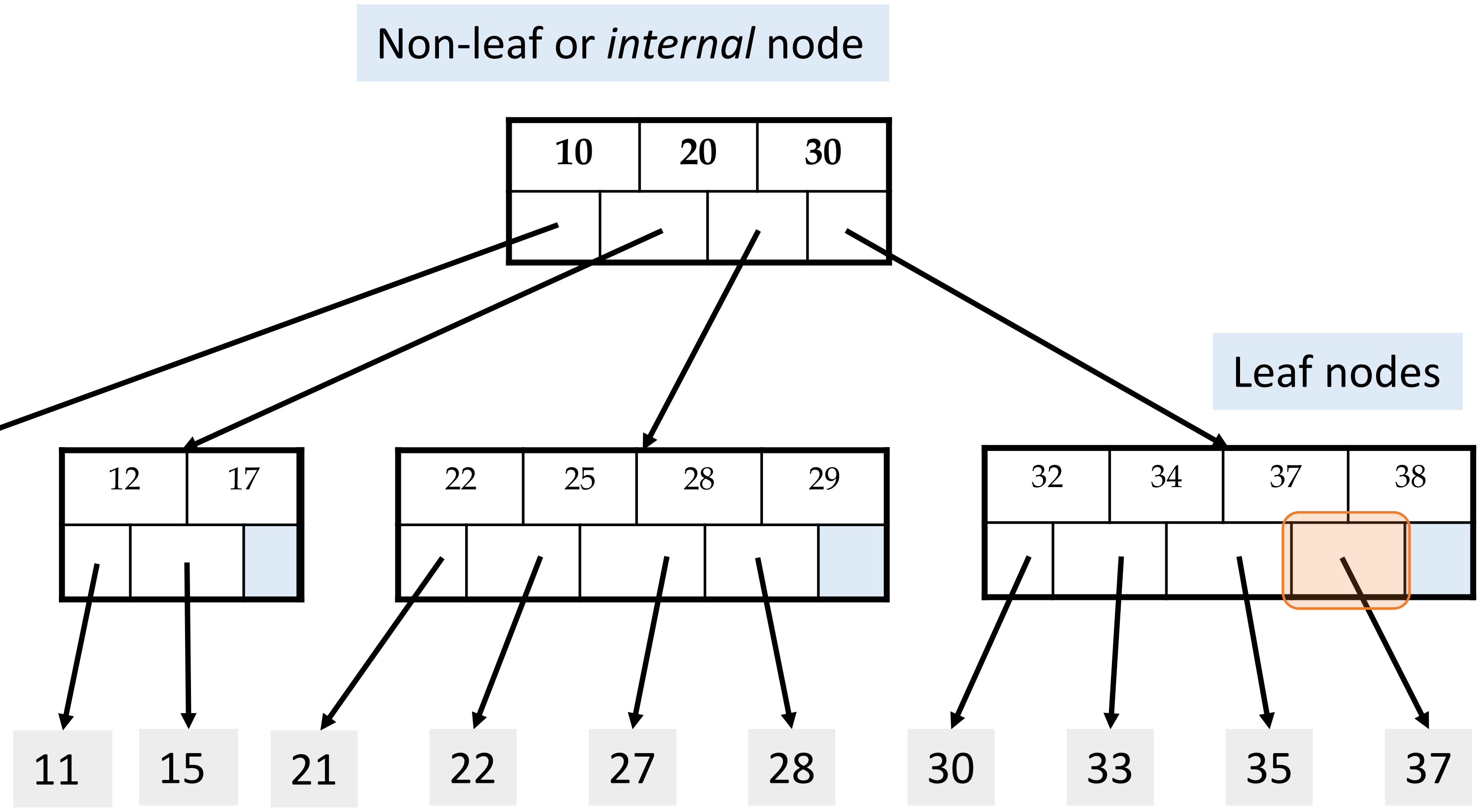
For each range, in a *non-leaf* node, there is a **pointer** to another node with keys in that range

B+ Tree Basics



Leaf nodes also have between *d* and *2d* keys, and are different in that:

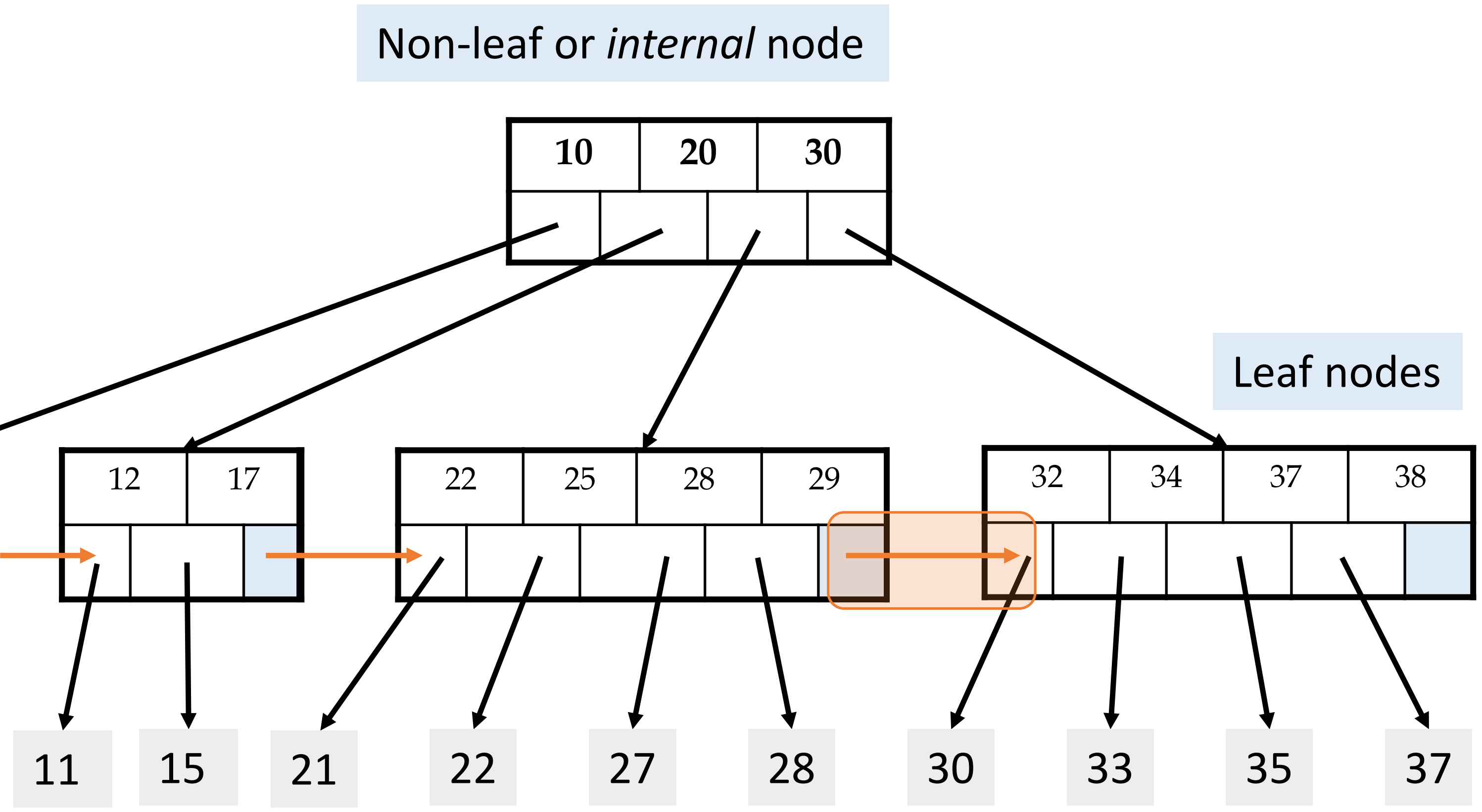
B+ Tree Basics



Leaf nodes also have between *d* and *2d* keys, and are different in that:

Their key slots contain pointers to data records

B+ Tree Basics

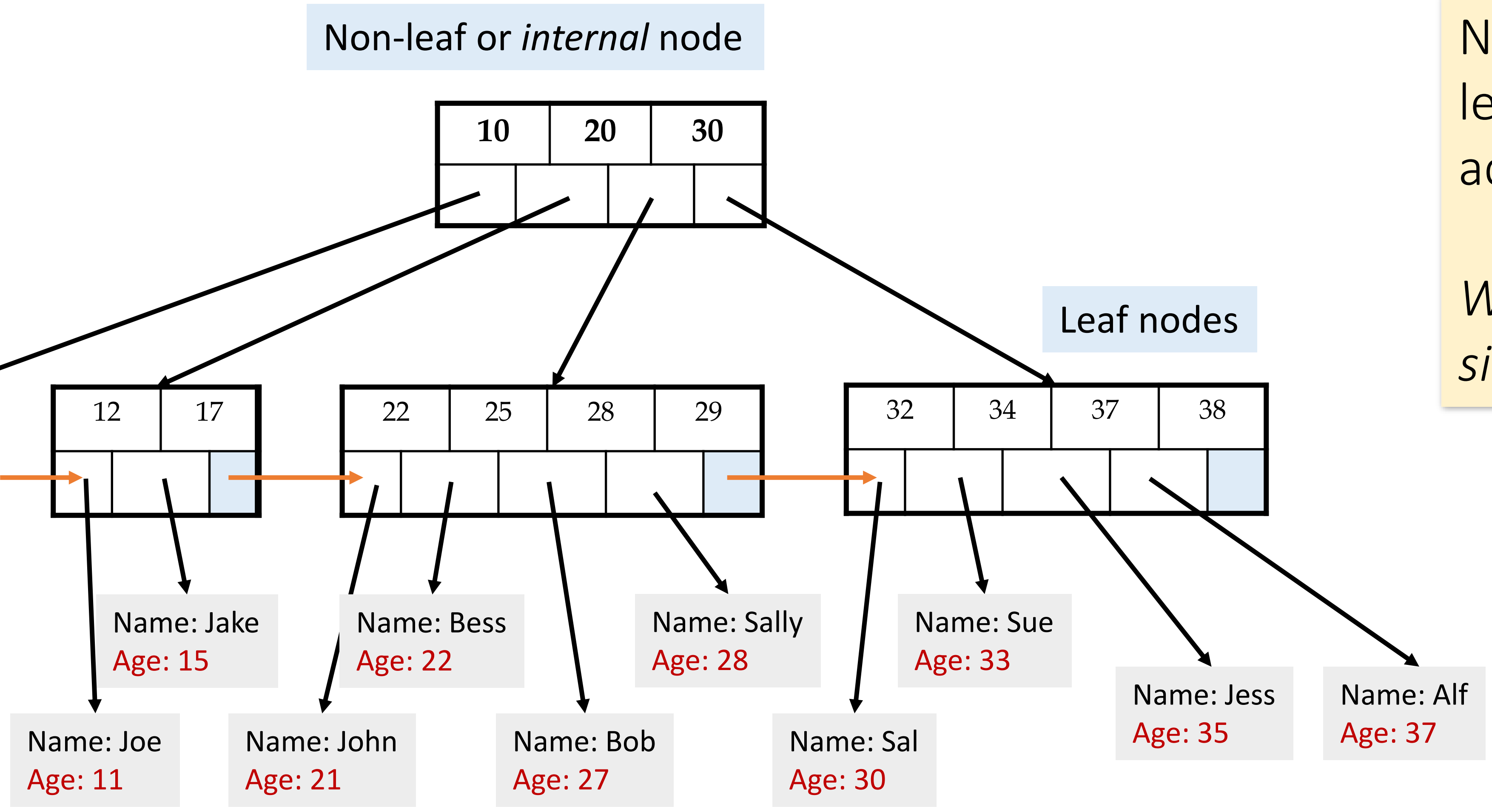


Leaf nodes also have between *d* and *2d* keys, and are different in that:

Their key slots contain pointers to data records

They contain a pointer to the next leaf node as well, ***for faster sequential traversal***

B+ Tree Basics



Note that the pointers at the leaf level will be to the actual data records (rows).

We might truncate these for simpler display (as before)…

Some finer points of B+ Trees

Searching a B+ Tree

- For exact key values:
  - Start at the root
  - Proceed down, to the leaf

```
SELECT name
FROM   people
WHERE  age = 25
```

- For range queries:
  - As above
  - *Then sequential traversal*

```
SELECT name
FROM   people
WHERE  20 <= age
  AND  age <= 30
```

B+ Tree Exact Search Animation

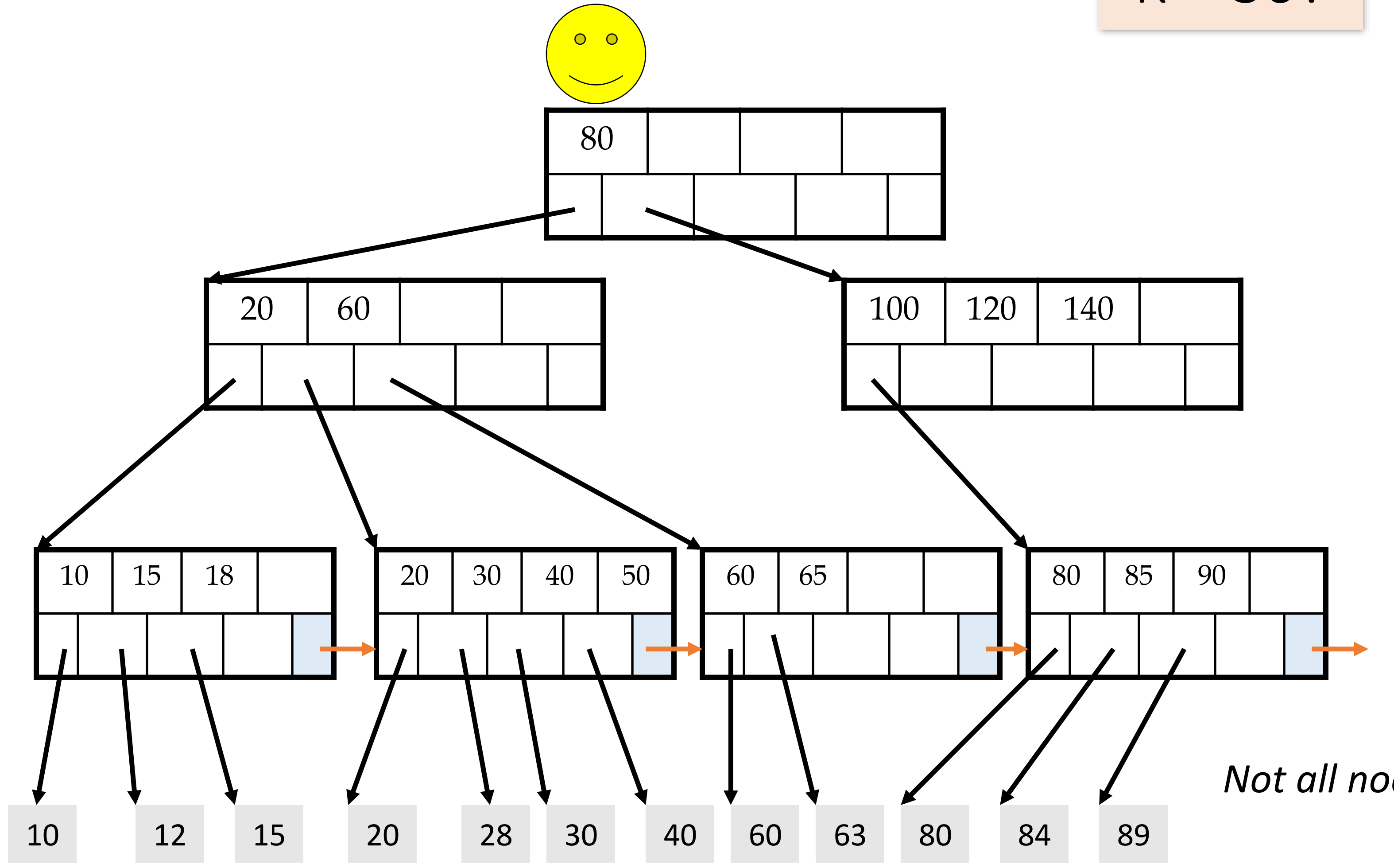
K = 30?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



Not all nodes pictured

B+ Tree Range Search Animation

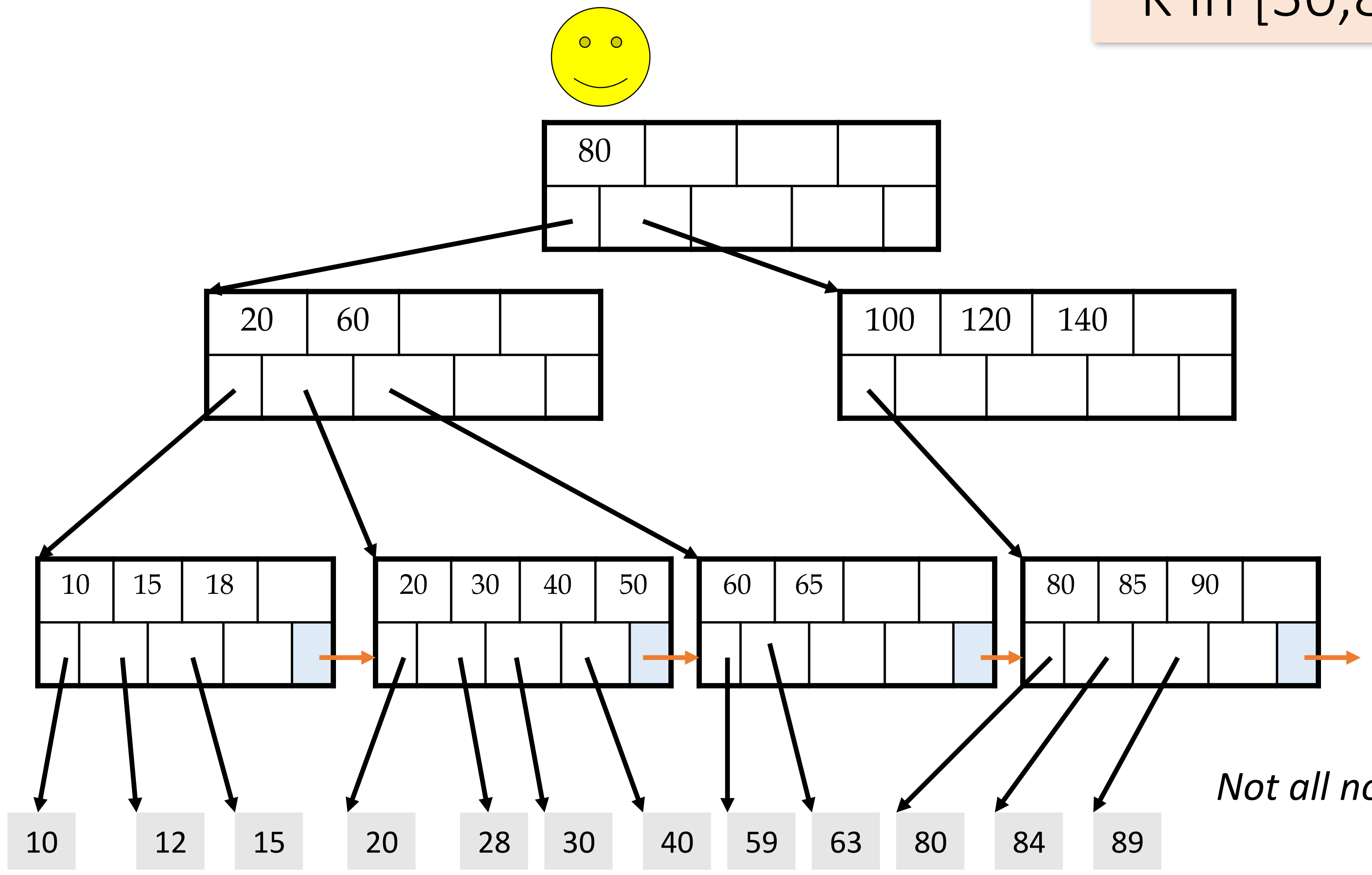
K in [30,85]?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



Not all nodes pictured

B+ Tree Design

- How large is ***d***?

- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes

NB: Oracle allows 64K = 2^16 byte blocks
→ d <= 2730

- We want each *node* to fit on a single *block/page*
  - 2d x 4 + (2d+1) x 8 <= 4096 → ***d <= 170***

B+ Tree: High Fanout = Smaller & Lower IO

- As compared to e.g. binary search trees, B+ Trees have **high *fanout*** (***between d+1 and 2d+1***)
- This means that the **depth of the tree is small** → getting to any element requires very few IO operations!
  - Also can often store most or all of the B+ Tree in main memory!
- A TiB = 2^{40} Bytes. What is the height of a B+ Tree (with fill-factor = 1) that indexes it (with 64K pages)?
  - $(2*2730 + 1)^h = 2^{40}$ → ***h = 4***

The fanout is defined as the number of pointers to child nodes coming out of a node

Note that fanout is dynamic- we'll often assume it's constant just to come up with approximate eqns!

The known universe contains ~10^{80} particles… what is the height of a B+ Tree that indexes these?

B+ Trees in Practice

- Typical order: d=100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Top levels of tree sit *in the buffer pool*:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Fill-factor is the percent of available slots in the B+ Tree that are filled; is usually < 1 to leave slack for (quicker) insertions

Typically, only pay for one IO!

Simple Cost Model for Search

- Let:
  - ***f*** = fanout, which is **in [d+1, 2d+1]** ***(we'll assume it's constant for our cost model…)***
  - ***N*** = the total number of *pages* we need to index
  - ***F*** = fill-factor (usually ~= 2/3)

- Our B+ Tree needs to have room to index ***N / F*** pages!
  - We have the fill factor in order to leave some open slots for faster insertions

- What height (*h*) does our B+ Tree need to be?
  - h=1 → Just the root node- room to index f pages
  - h=2 → f leaf nodes- room to index f^2 pages
  - h=3 → f^2 leaf nodes- room to index f^3 pages
  - …
  - h → f^{h-1} leaf nodes- room to index f^h pages!

→ We need a B+ Tree of height h = $\left\lceil \log_f \frac{N}{F} \right\rceil$!

Simple Cost Model for Search

- Note that if we have ***B*** available buffer pages, by the same logic:
  - We can store $\boldsymbol{L_B}$ levels of the B+ Tree in memory
  - where $\boldsymbol{L_B}$ ***is the number of levels such that the sum of all the levels' nodes fit in the buffer:***
    - $B \geq 1 + f + \cdots + f^{L_B - 1} = \sum_{l=0}^{L_B - 1} f^l$

- In summary: to do exact search:
  - We read in one page per level of the tree
  - However, levels that we can fit in buffer are free!
  - Finally we read in the actual record

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + 1$

where $B \geq \sum_{l=0}^{L_B - 1} f^l$

Simple Cost Model for Search

- To do range search, we just follow the horizontal pointers

- The IO cost is that of loading additional leaf nodes we need to access + the IO cost of loading each ***page*** of the results- we phrase this as "Cost(OUT)"

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + Cost(OUT)$

where $B \geq \sum_{l=0}^{L_B - 1} f^l$

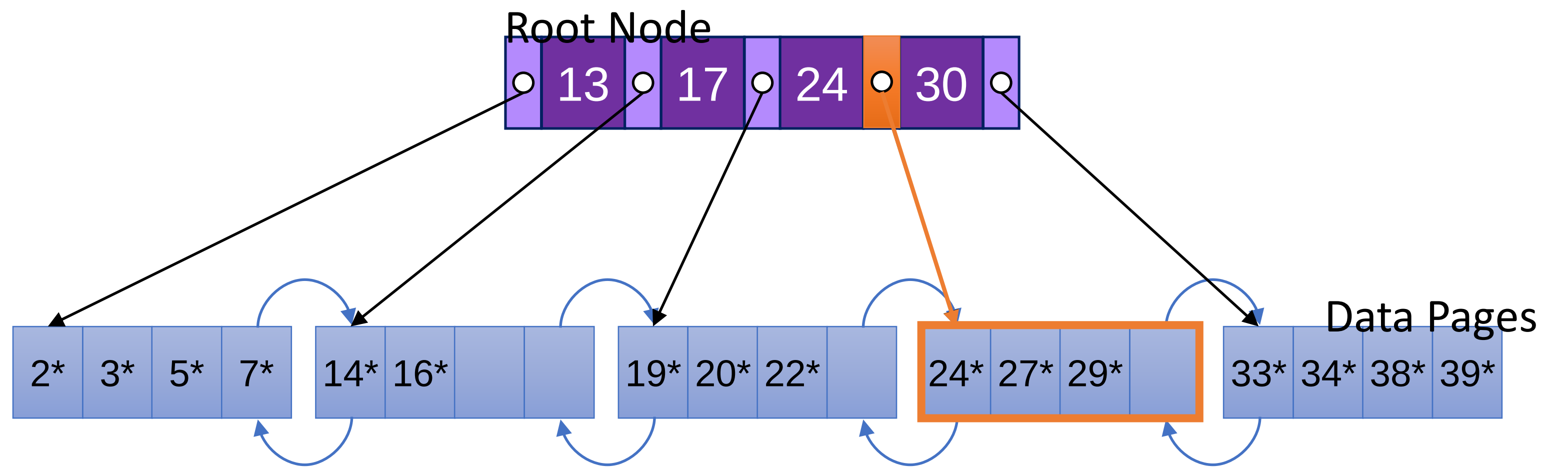
Fast Insertions & Self-Balancing

- B+ Tree insertion algorithm has several attractive qualities:
  - **~ Same cost as exact search**
  - ***Self-balancing:*** B+ Tree remains **balanced** (with respect to height) even after insert

B+ Trees also (relatively) fast for single insertions!
However, can become bottleneck if many insertions (if fill-factor slack is used up…)

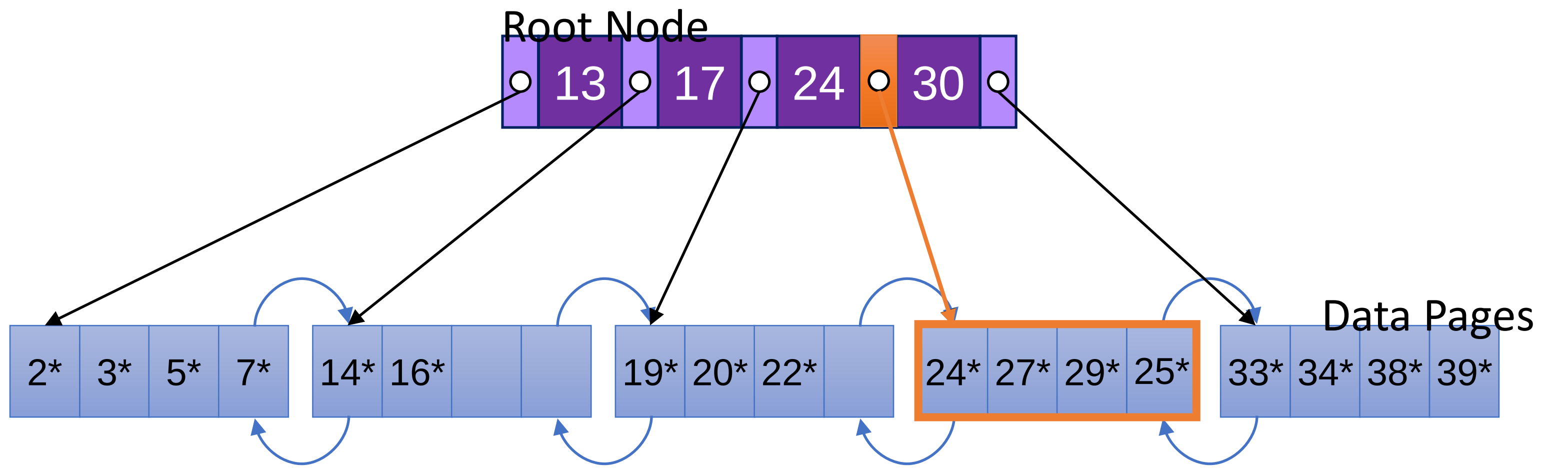
Inserting 25* into a B+-Tree

- Find the correct leaf



Inserting 25* into a B+-Tree

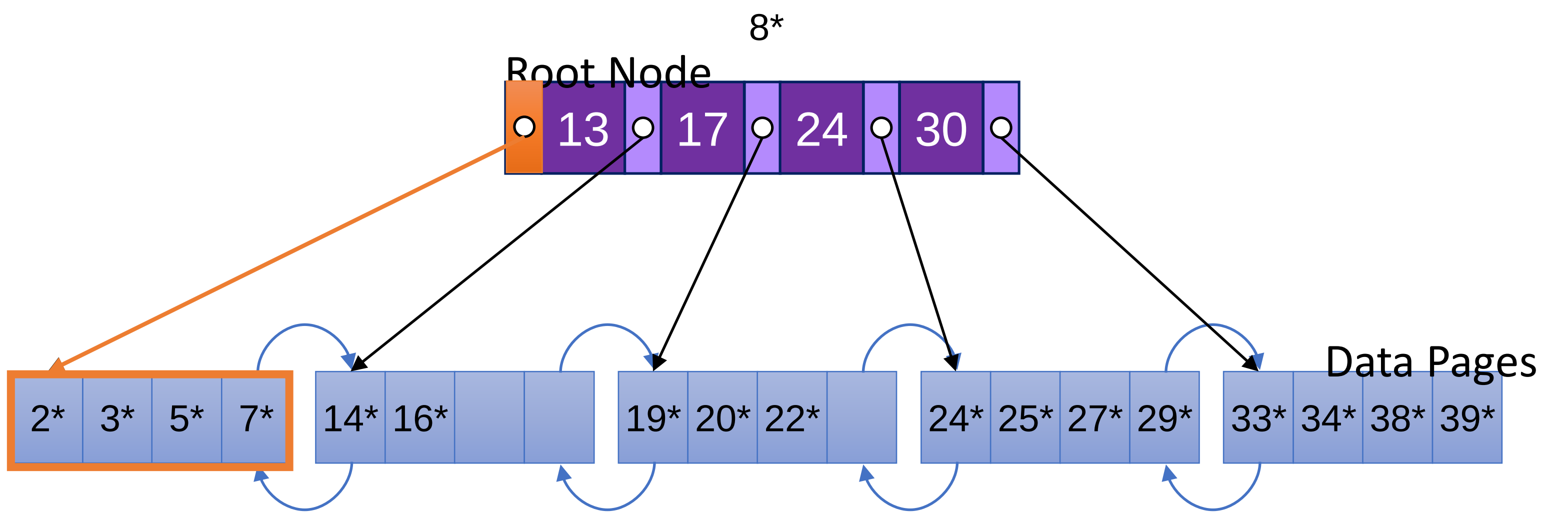
- Find the correct leaf
- If there is room in the leaf just add the entry
  - Sort the page leaf page by key



That was easy!
(it gets harder)

Inserting 8* into a B+-Tree

- Find the correct leaf



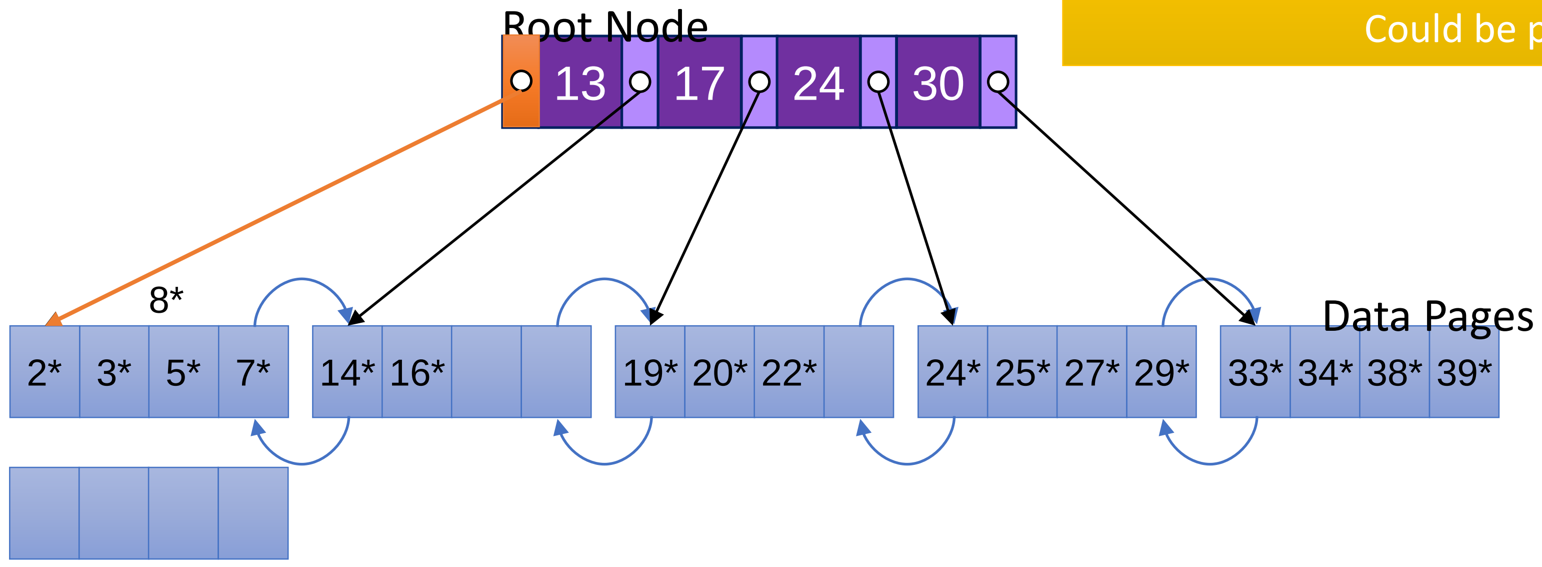
Inserting 8* into a B+-Tree

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

What happens to record ids if this is an Alternative 1 index (store data by value)?

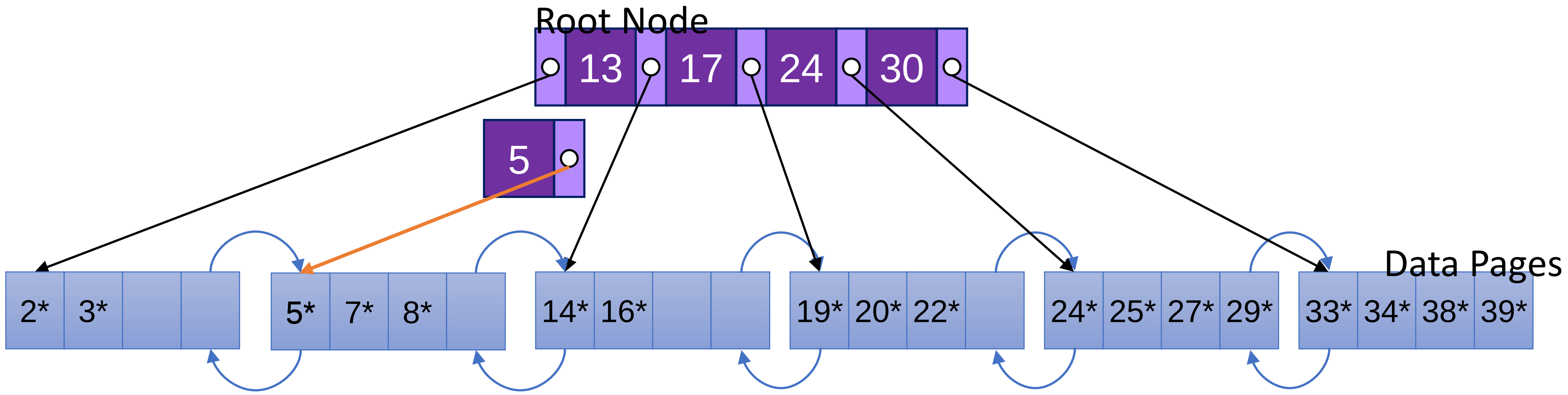
Careful:
Moving records to new pages changes records id

Could be pricey!!!



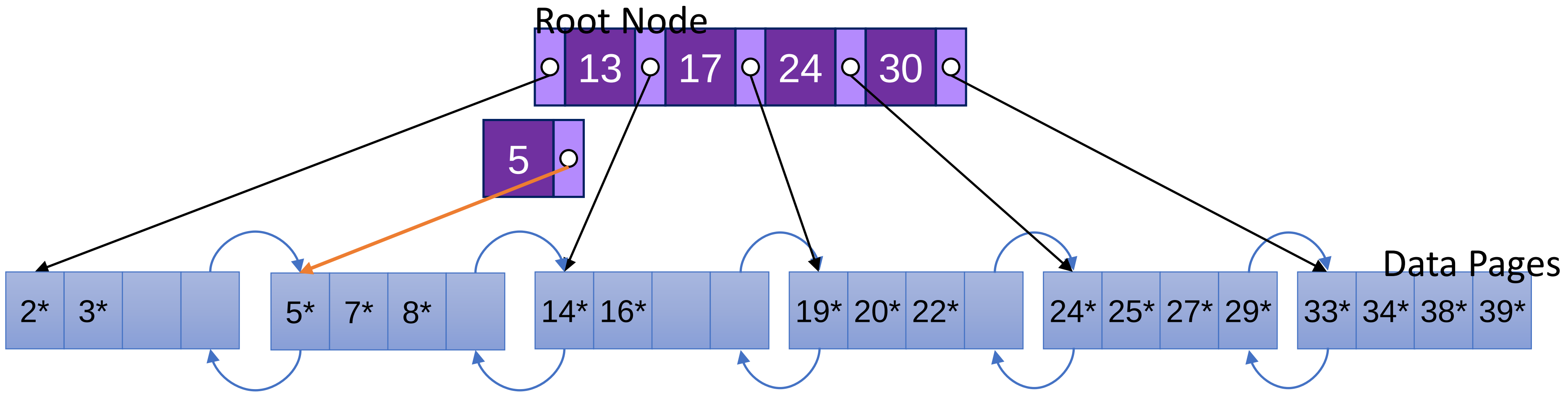
Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key



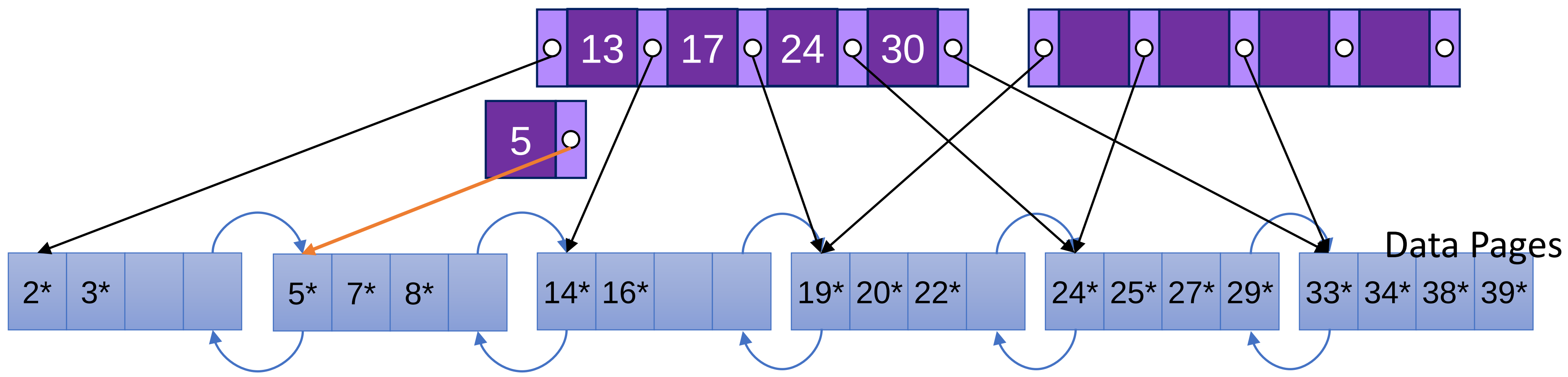
Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key



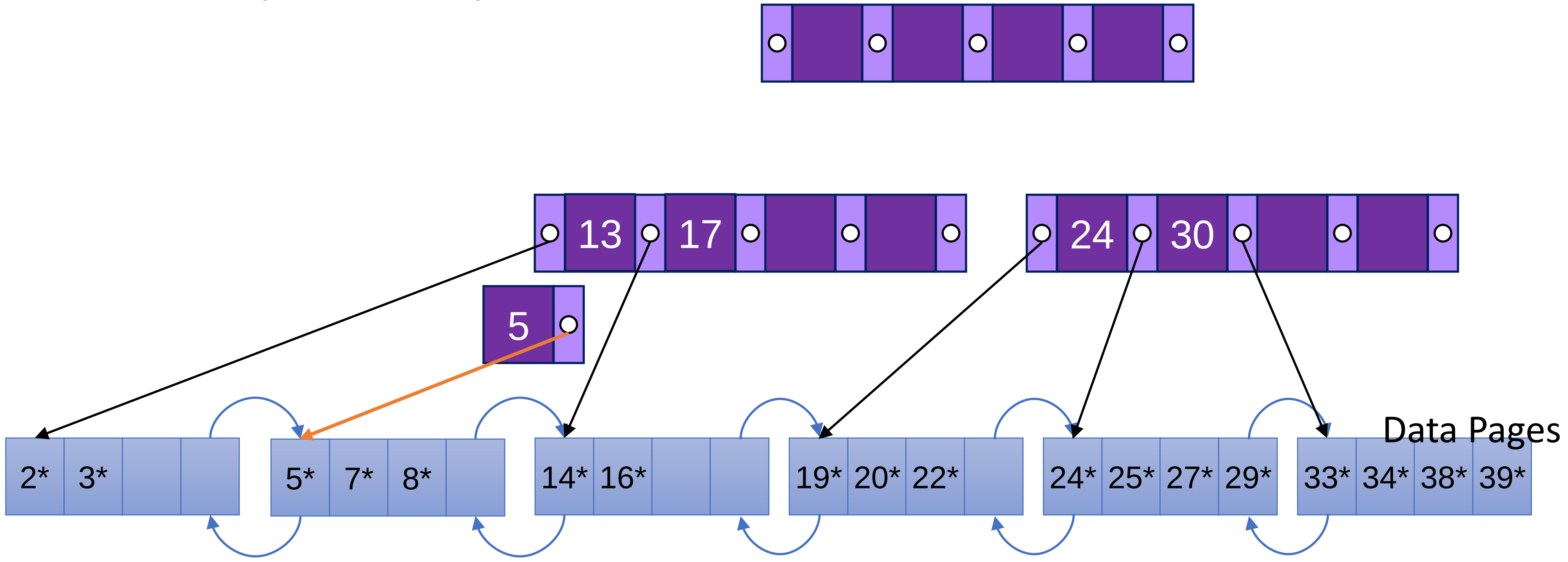
Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key
- Recursively split index nodes
  - Redistribute right d keys



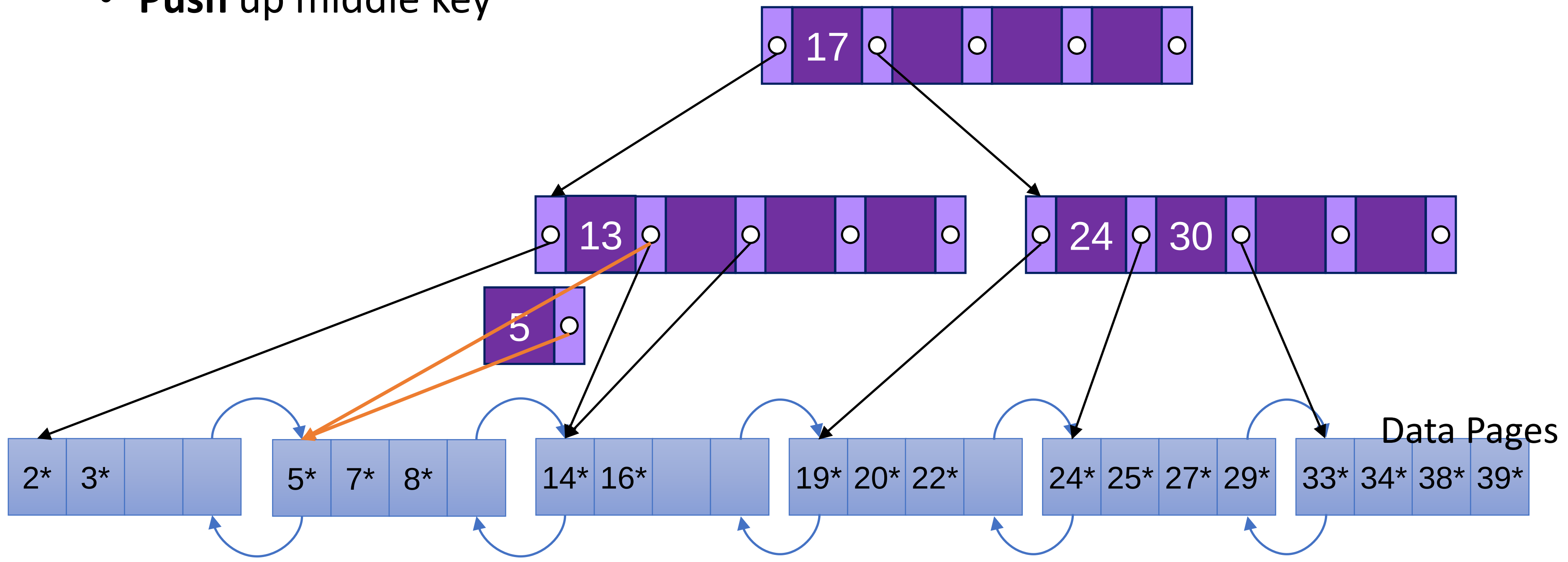
Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key
- Recursively split index nodes
  - Redistribute right d keys
  - **Push** up middle key



Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key
- Recursively split index nodes
  - Redistribute right d keys
  - **Push** up middle key



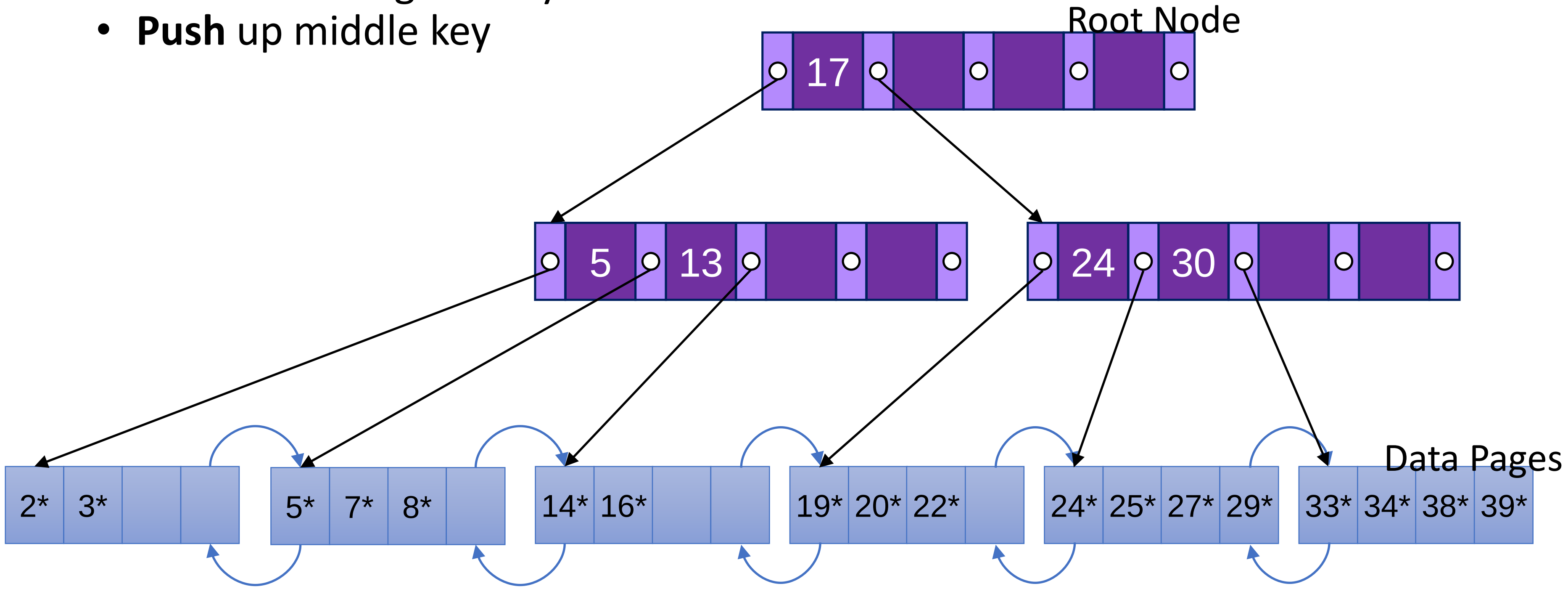
Inserting 8* into a B+-Tree

- Find the correct leaf
  - Split leaf if there is not enough room
  - Redistribute entries evenly
  - **Copy** up middle key
- Recursively split index nodes
  - Redistribute right d keys
  - **Push** up middle key

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



Inserting a Data Entry into a B+ Tree

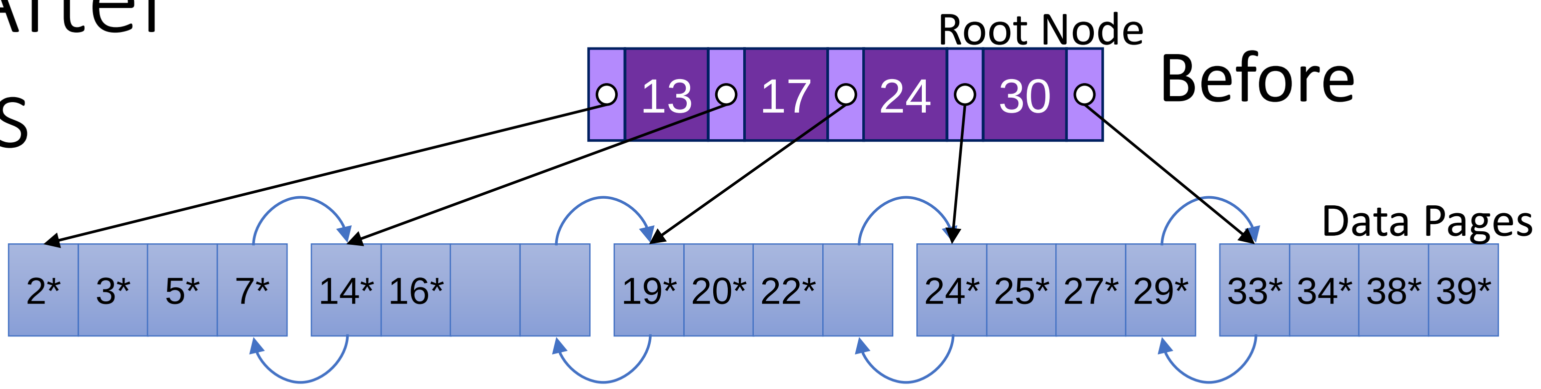
- Find correct leaf *L.*

- 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.
    - Insert index entry pointing to *L2* into parent of *L*.

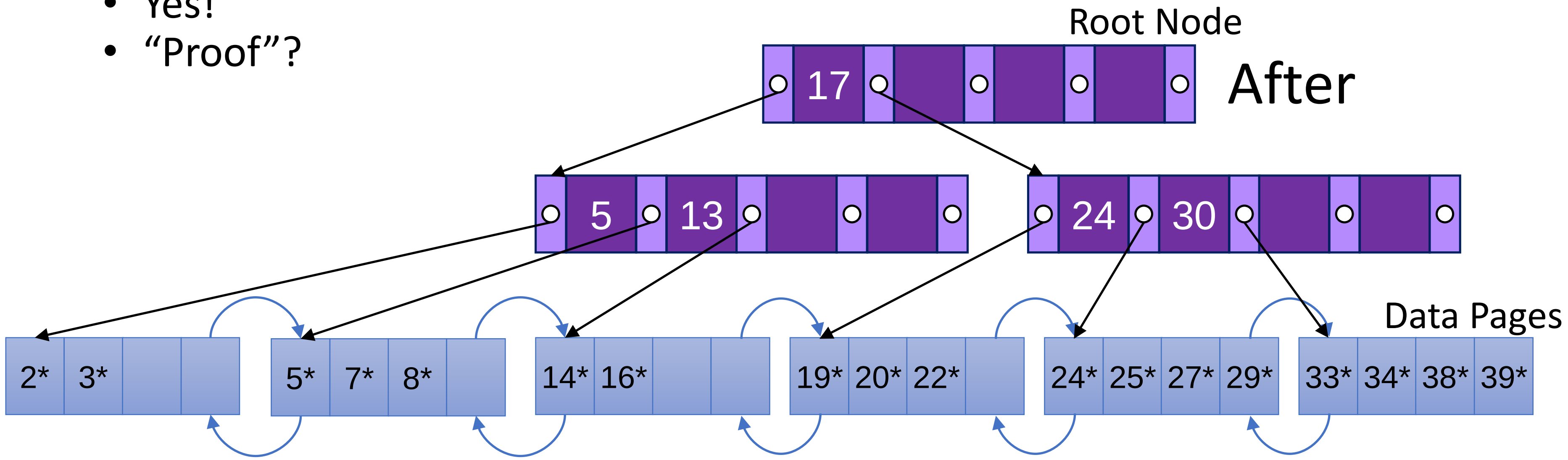
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)

- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

Before and After Observations



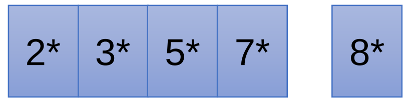
- Notice that the **root** was split to increase the height
  - *Grow from the root not the leaves*
  - ➔ All paths from root to leaves are equal lengths
- Does the occupancy invariant hold?
  - All nodes (except root) are at least half full
  - Yes!
  - "Proof"?



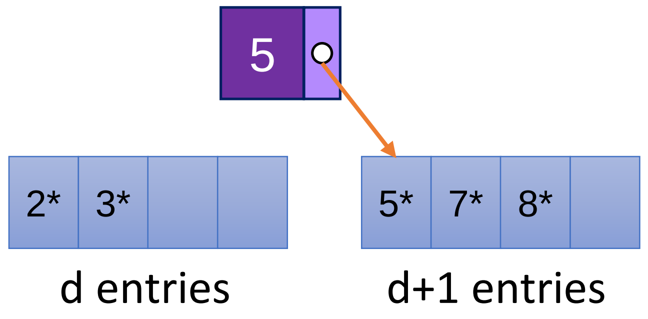
Splitting a Leaf

d = 2

- Start with full leaf (2d) entries:
  - Add a 2d +1 entry (8*)



- Split into leaves with (d, d+1) entries
  - **Copy** key up to parent



Why copy key and not push key up to parent?

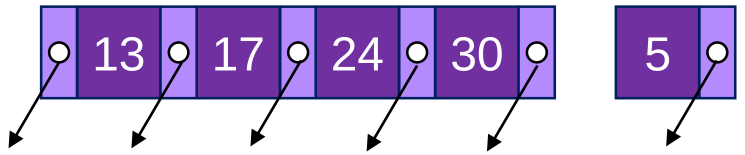
Key has value attached (5*)

Occupancy invariant holds after split.

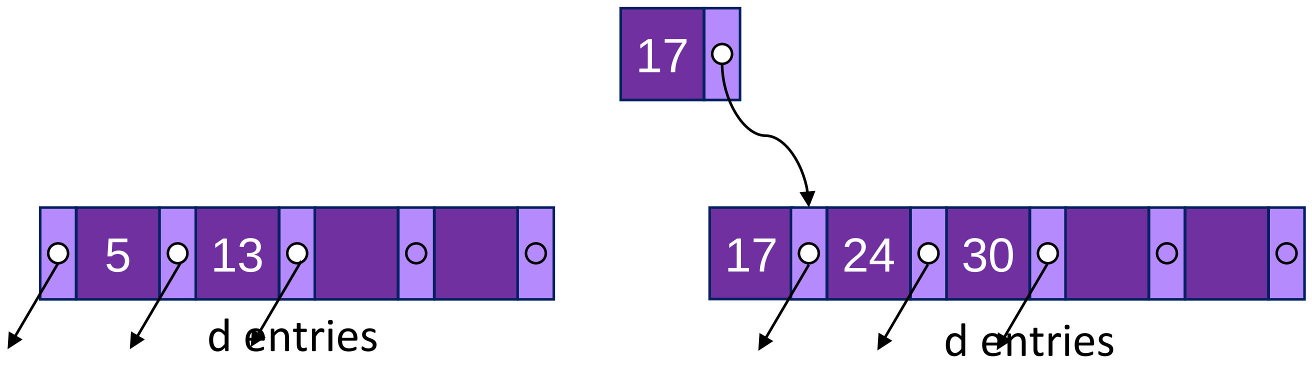
Splitting an interior node

d = 2

- Start with full interior node (2d) entries:
  - Add a 2d +1 entry



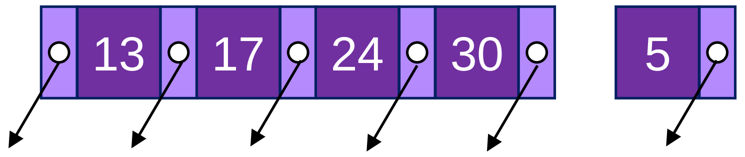
- Split into nodes with (d, d) entries
  - **Push** key up to parent



Splitting an interior node

d = 2

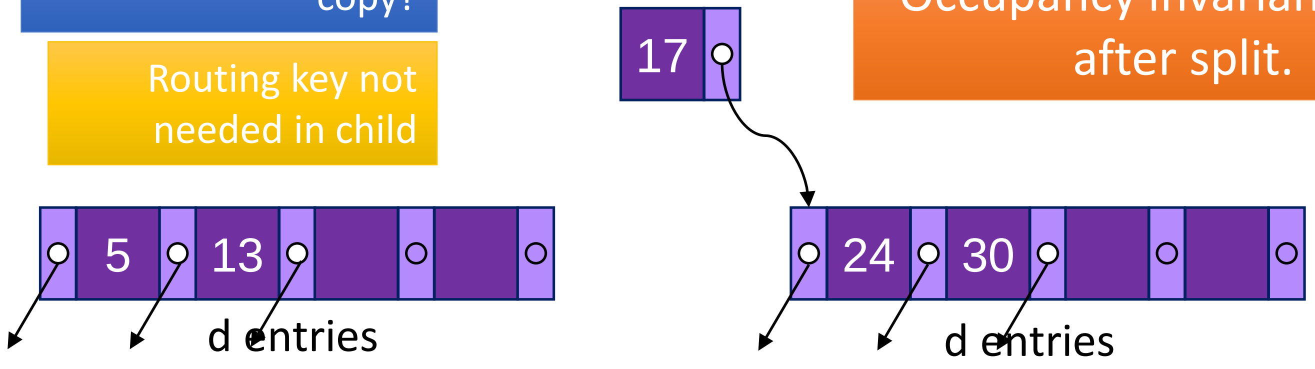
- Start with full interior node (2d) entries:
  - Add a 2d +1 entry



- Split into nodes with (d, d) entries
  - **Push** key up to parent

Why push and not copy?

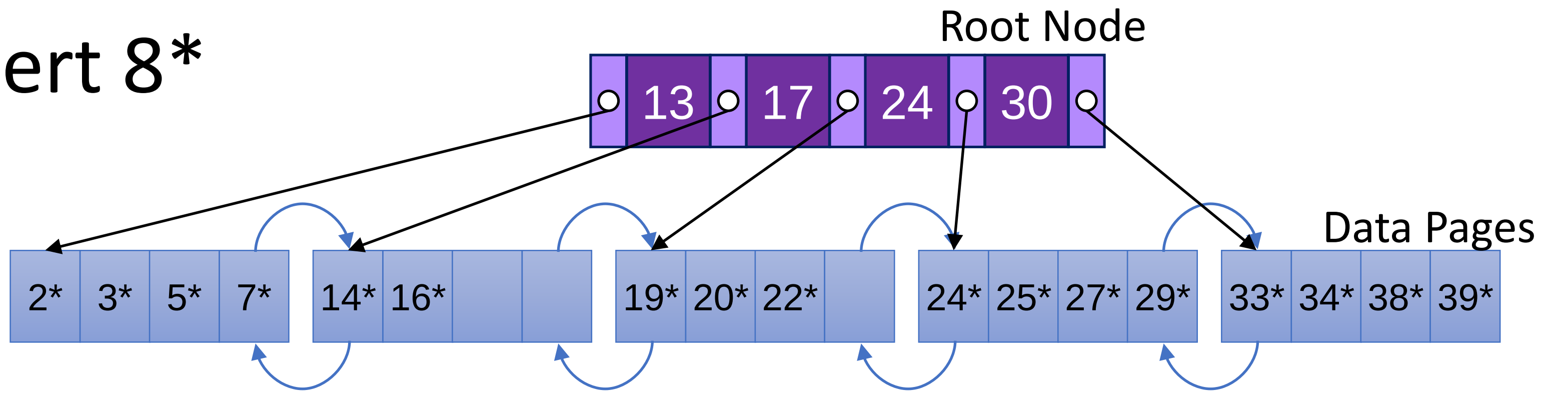
Routing key not needed in child



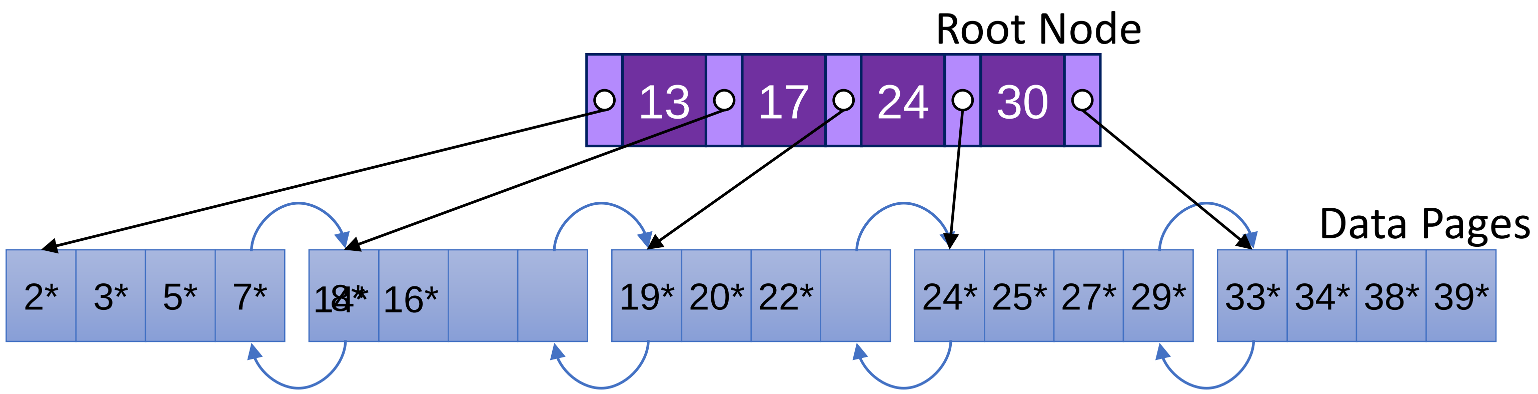
Occupancy invariant holds after split.

Did we have to split?

Insert 8*

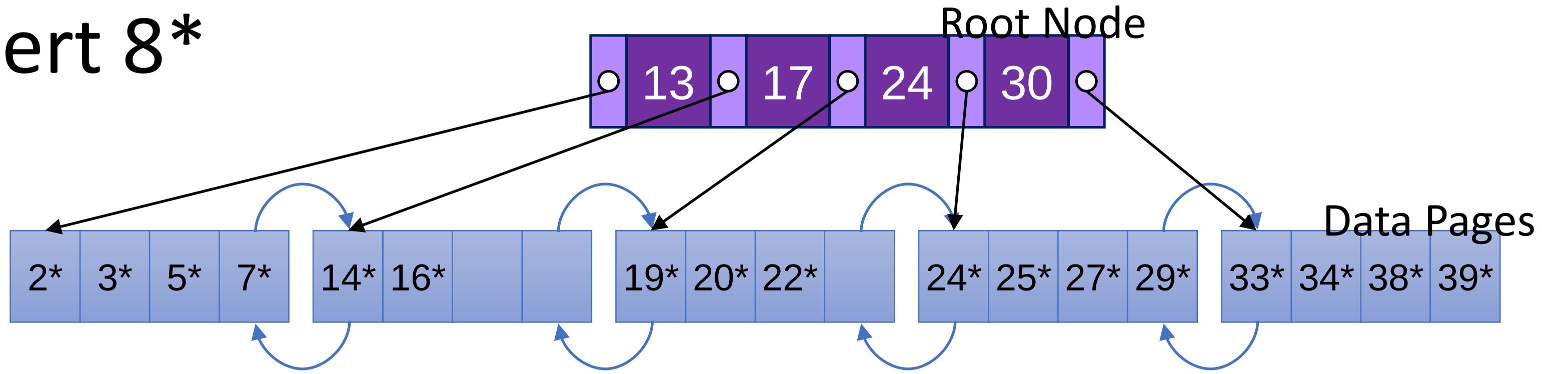


- What else could we do?
  - Redistribute keys?



Did we have to split?

Insert 8*

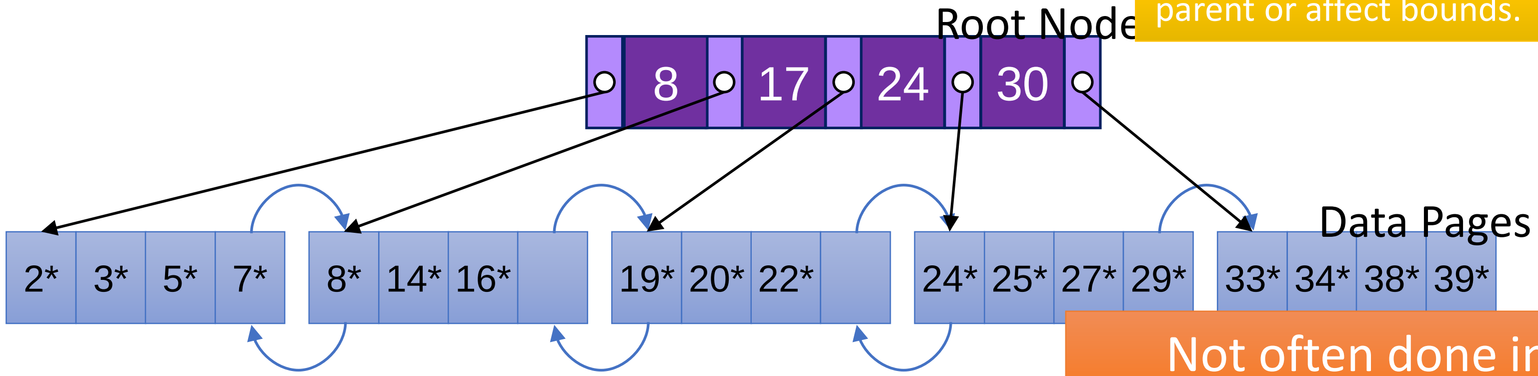


Do we need to recurse?

- What else could we do?
  - Redistribute keys?

No. Why?

Change does not split parent or affect bounds.



Not often done in practice

Summary

- We covered an algorithm + some optimizations for sorting larger-than-memory files efficiently
  - An ***IO aware*** algorithm!

- We create **indexes** over tables in order to support ***fast (exact and range) search*** and ***insertion*** over ***multiple search keys***

- **B+ Trees** are one index data structure which support very fast exact and range search & insertion via ***high fanout***
  - ***Clustered vs. unclustered*** makes a big difference for range queries too