CS 250: Foundations of Computer Systems [Data Structures for Storage]

Indexes to the rescue!
Have you a surfeit of on-disk data?
Needing searches by the sweat of your brow
Maintain indexes alongside thy data
Separate and apart
Update during ingestion or writes
A friend to consult when you search
Their raison d'etre
To speed up what you seek
Without many a disk seek
The search load lightened

Shrideep Pallickara
Computer Science
Colorado State University

1

Frequently asked questions from the previous class survey

- Complexity of operations
- Does the depth of a BST have performance implications?
- Are B-Trees a better choice?

2

Deletion of nodes with children in BST

- We looked at how you can splice out the successor
- You can also splice out the predecessor
  - This has the same complexity as splicing the successor: O(h) where h is the height of the tree
- For better empirical performance, some implementations
  - Alternate (or give equal priority to) splicing out the successor and predecessor

Predecessor → Maximum value in its left subtree
Successor → Minimum value in its right subtree

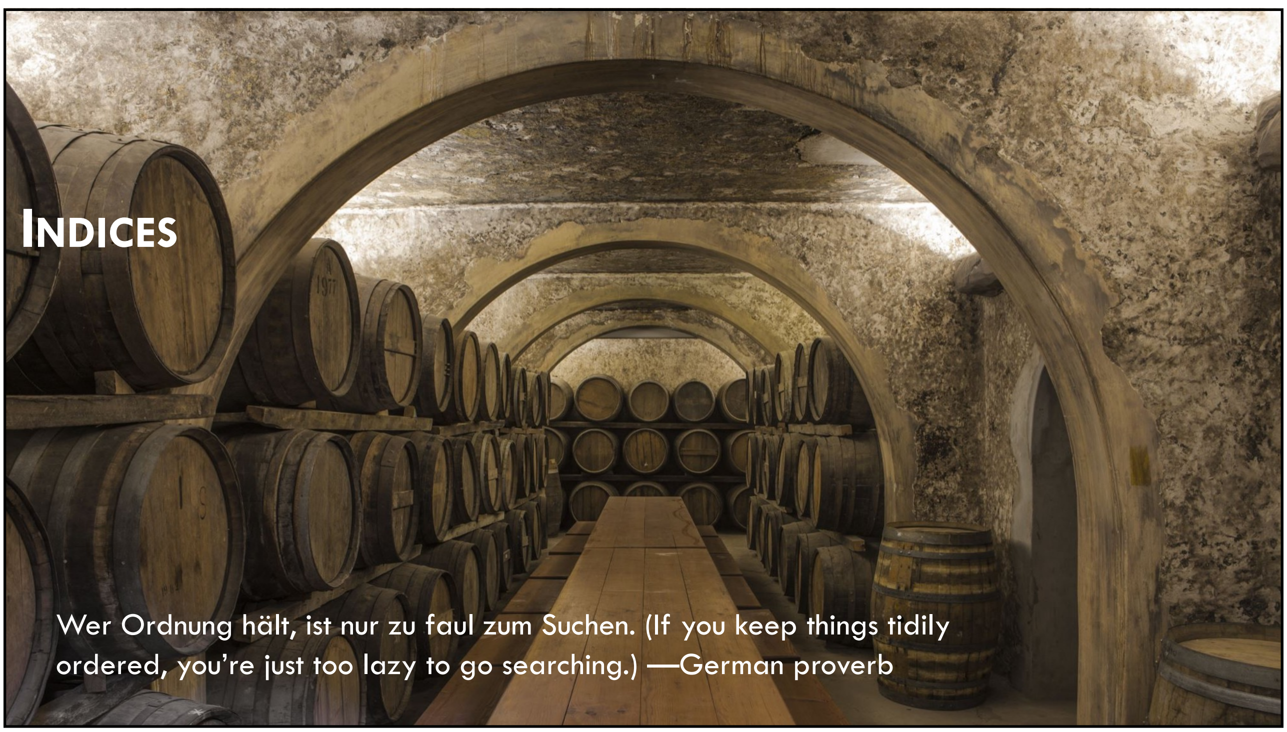
"Look at me!
Look at me!
Look at me NOW!
It is fun to have fun
But you have to know how.

SELF BALANCING TREES

Topics covered in this lecture

- B-Trees

5



6

Databases and indices

- In order to efficiently find the value for a particular key in the database, we *need* a different data structure: an **index**
- Key enabling idea
  - Keep some additional *metadata* on the side
  - **Index acts as a signpost** and helps you to locate the data you want
- If you want to search records in *several different* ways?
  - You may need several different **indexes on different parts** of the data

An index is an additional structure that is **derived** from the primary data

- Many databases allow you to add and remove indexes
  - This doesn't affect the contents of the database; it only affects the **performance of queries**
- Maintaining additional structures **incurs overheads during writes**
  - For writes, it's hard to beat the performance of simply appending to a file, because that's the simplest possible write operation
- Any kind of index usually slows down writes
  - Because the index also needs to be *updated every time data is written*

Trade-off in storage systems

- Well-chosen indexes *speed up* **read queries**
  - but every index slows down **writes**
- For this reason, databases **don't usually index everything** by default
  - Require you to *choose indexes manually*
    - Using your knowledge of the application's typical query patterns
  - You can then choose the indexes that give your application the greatest benefit, without introducing more overhead than necessary

The most widely used indexing structure?

- The **B-tree**
- Introduced in 1970 and called "ubiquitous" less than 10 years later
  - Inventors: Rudolf Bayer & Edward M. McCreight while @ Boeing Research Labs
  - "B" was not defined: Could be for "balanced", "broad", "Boeing"?
- B-trees have stood the test of time very well
- They **remain the standard index implementation** in *almost all* relational databases, and many nonrelational databases use them too

Nodes in B-Trees

- Are usually *also* referred to as **pages**
- Very closely aligned with block-sizes on storage devices

B-Tree Nodes (or pages)

- The node contains several keys and references to child nodes
- Each child node is responsible for a **continuous range of keys**
  - The keys indicate where the boundaries between those ranges lie
- Most databases can fit into a B-tree that is **three or four levels deep**
  - So, you don't need to follow many references to find the page you are looking for
  - A four-level tree of 4 KB pages with a branching factor (or fanout) of 500 can store up to 250 TB

13

How data is read from stable storage

- HDDs and SSDs address **blocks** rather than individual bytes
- Most operating systems have a block device abstraction
- When we're reading a single word from an HDD or an SSD?
    - The whole block containing it is read

COLORADO STATE UNIVERSITY　Professor: SHRIDEEP PALLICKARA　COMPUTER SCIENCE DEPARTMENT　DATA STRUCTURES FOR STORAGE　L26.14

14

Primary limitation and design consideration for building efficient on-disk structures

- The **cost of disk access** itself
- The smallest unit of disk operation is a **block**
- To follow a pointer to the specific location within the block, we have to **fetch an entire block**

If we must always read a block?

- Why don't we **change the layout of the data structure** to take advantage of it?
- Creating long dependency chains in on-disk structures greatly increases code and structure complexity
  - Much better to keep the number of pointers and their spans to a minimum

On-disk structures optimize for target storage specifics and optimize for fewer disk accesses

- Improving locality
- Optimizing the internal representation of the structure
- Reducing the number of out-of-page pointers

B-Trees combine these aforementioned ideas

- Account for storage characteristics (esp. the block construct)
- Increase node fanout
- Reduce tree height
- Reduce the number of node pointers
- Reduce frequency of balancing operations

B-Tree analogy: vast catalog room in the library

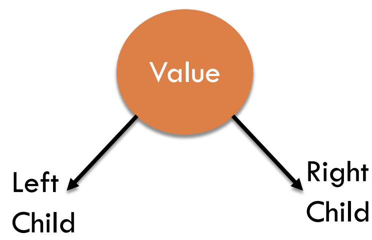
- You first have to pick the correct cabinet
- Then the correct shelf in that cabinet
- Then the correct drawer on the shelf, and
- Then browse through the cards in the drawer to find the one you're searching for

- Similarly, a B-Tree **builds a hierarchy** that helps to *navigate* and *locate* the searched items quickly

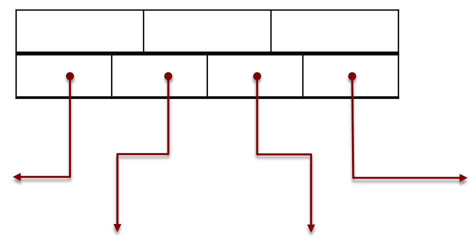
Depicting nodes in BSTs versus B-Trees [1/2]

- In most of the literature, **binary tree nodes are drawn as circles**
  - Each node is responsible for just one key and splits the range into two parts
    - This level of detail is sufficient and intuitive
- B-Tree nodes are often drawn as **rectangles**
  - Pointer blocks are also shown explicitly to highlight the relationship between child nodes and separator keys

Depicting nodes in BSTs versus B-Trees [2/2]



BST Node



B-Tree Node (or Page)

Keys inside the B-Tree nodes

- B-Trees are **sorted**
  - Keys inside the B-Tree nodes are stored in order
  - We can use an algorithm like *binary search* to locate a searched key
- This also implies that lookups in B-Trees have *logarithmic complexity*

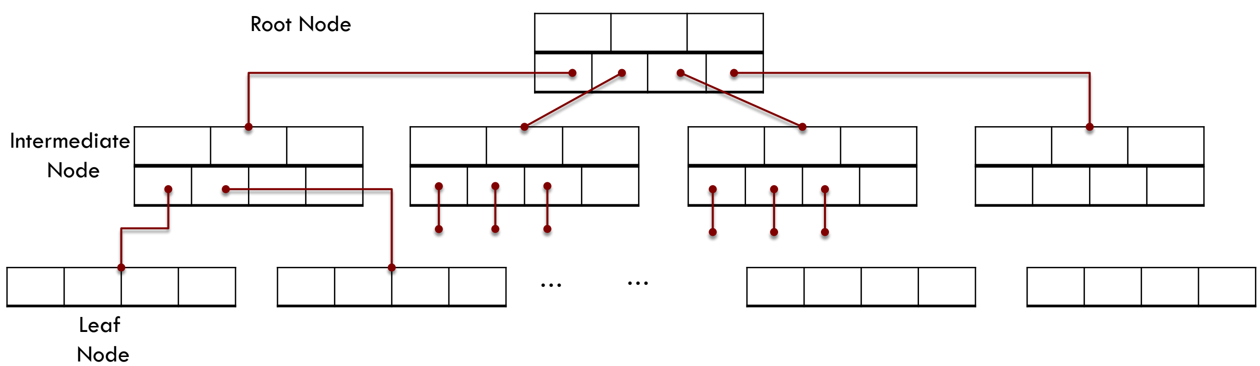
Using B-Trees, we can efficiently execute both point and range queries

- **Point queries** locate a single item
  - Expressed by the equality (**=**) predicate in most query languages
- **Range queries** are used to query multiple data items in order
  - Expressed by comparison (**<, >, ≤,** and **≥**) predicates

The B-Tree Hierarchy comprises multiple nodes

- Each node holds up to N keys
  - And N + 1 pointers to the child nodes
- Nodes are logically grouped into three groups:
  - **Root** node, which is the top of the tree
  - **Leaf** nodes: Bottommost layer nodes that have no child nodes
  - **Internal** nodes These are all other nodes with leaves
    - There is usually more than one level of internal nodes

B-Tree node hierarchy



Nomenclature

- Since B-Trees are a page organization technique
  - i.e., they are used to organize and navigate fixed-size pages
  - We often use terms node and page interchangeably
- The relation between the **node capacity** and the number of keys it actually holds is called **occupancy**

B-Trees are characterized by their fanout

- Fanout refers to the number of keys stored in each node
- **Higher fanout** helps to:
  - **Amortize** the cost of **structural changes** required to keep tree balanced
  - **Reduces the number of seeks** by storing keys and pointers to child nodes in a single block or multiple consecutive consecutive blocks
- Balancing operations (namely, splits and merges) are triggered when the nodes are full or nearly empty

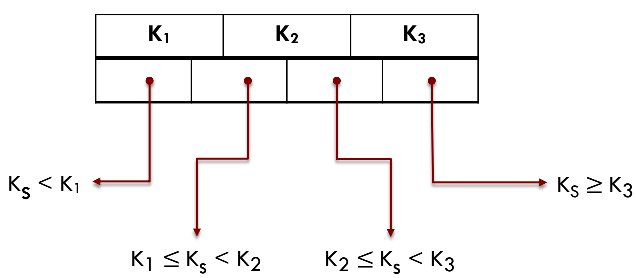
Separator Keys

- Keys stored in B-Tree nodes are called index entries, separator keys, or divider cells
- **Split the tree into subtrees** (also called branches or subranges), holding corresponding key ranges
  - Keys are stored in sorted order to allow binary search
- A subtree is found by locating a key and following a corresponding pointer from the higher-level to the lower-level

About pointers in a node

- The first pointer in the node?
  - Points to the subtree holding items less than the first key
- The last pointer in the node?
  - Points to the subtree holding items greater than or equal to the last key
- Other pointers?
  - Reference subtrees **between the two keys**: $K_{i-1} \leq K_s < K_i$, where K is a set of keys, and K_s is a key that belongs to the subtree.

Separator keys splitting a tree into subtrees



B-tree construction

- Rather than being built from top-to-bottom (as in BSTs), B-Trees are from **bottom-to-top**
- The number of leaf nodes grows, which increases the number of internal nodes and tree height
- B-Trees **reserve extra space** inside nodes for future insertions and updates
  - Tree storage utilization can get as low as 50%, but is usually much higher
- Higher occupancy does not influence B-Tree performance negatively

B-Tree lookup complexity can be viewed from two standpoints

- The number of block transfers
- The number of comparisons done during the lookup

B-Tree lookup complexity: Number of block transfers

- In terms of number of transfers, the logarithm base is N (number of keys per node)
  - There are N times more nodes on each new level
    - Following a child pointer reduces the search space by the factor of N
  - During lookup, at most $\log_N M$ pages are addressed to find a target key
    - M is the total number of items in the B-Tree
  - The number of child pointers that have to be followed on the root-to-leaf pass is also equal to the number of levels
    - In other words, the height h of the tree

B-Tree lookup complexity: Number of block transfers

- B-Tree lookup complexity is generally referenced as log M
- Logarithm base is generally not used in complexity analysis
  - Changing the base simply **adds a constant factor**
  - Multiplication by a constant factor does not change complexity
  - For example, given the nonzero constant factor c, $O(|c| \times n) == O(n)$

B-Tree lookup complexity: Number of comparisons

- From the perspective of number of comparisons within a node
  - The logarithm base is 2
  - Since searching a key inside each node is done using binary search
  - Every comparison halves the search space
- Complexity is log M

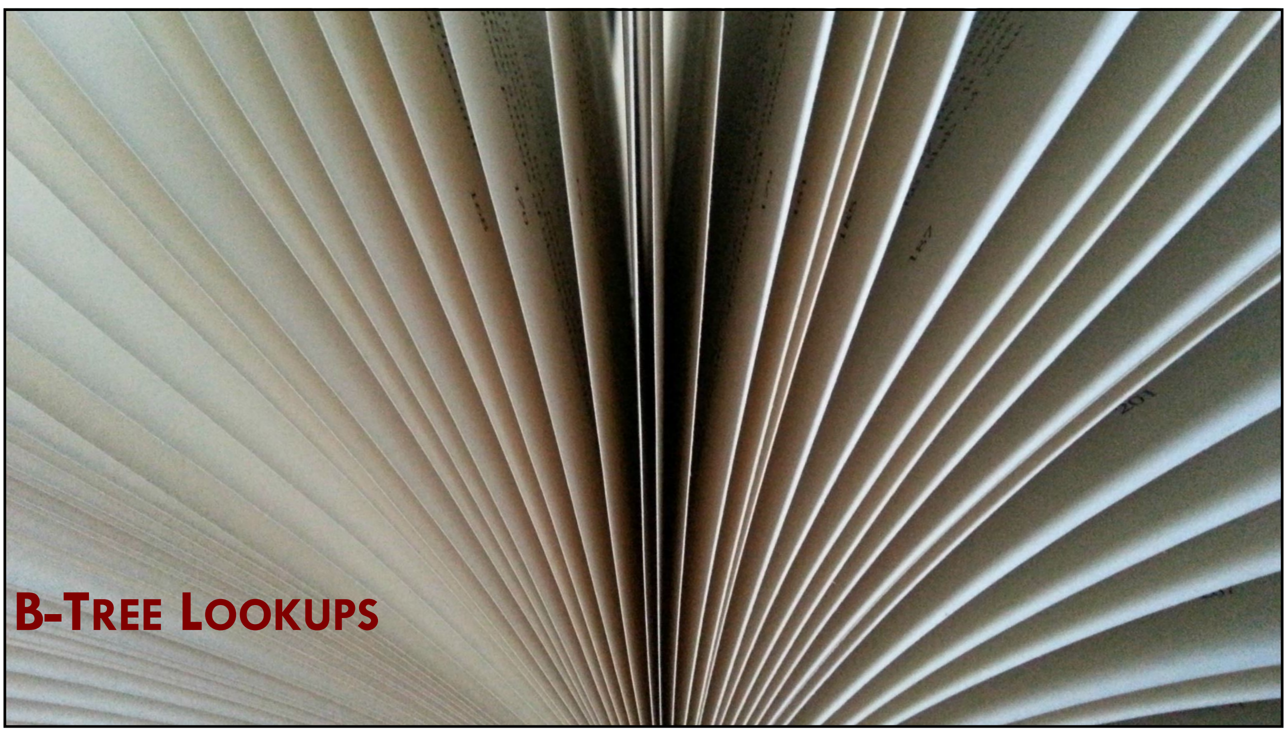
Different ways to describe key and child offset counts [1/2]

- The original paper refers to device-dependent natural number k
  - Nodes, in this case, can hold between k and 2k keys, but can be partially filled
  - Hold at least k + 1 and at most 2k + 1 pointers to child nodes
- The root page can hold between 1 and 2k keys
  - Later, a number l is introduced, and it is said that any nonleaf page can have l + 1 keys

Different ways to describe key and child offset counts [2/2]

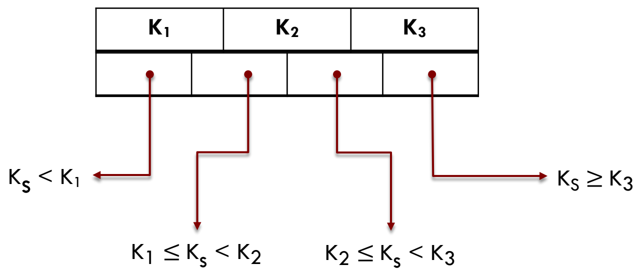
- Other sources, describe nodes that can hold up to N separator keys and N + 1 pointers, with otherwise similar semantics and invariants
- Both approaches bring us to the same result
  - Differences are only used to emphasize the contents of each source
  - We stick to N for clarity

37

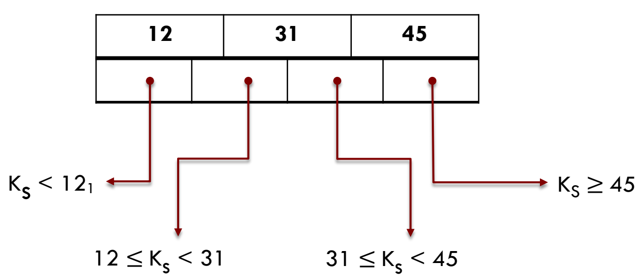


38

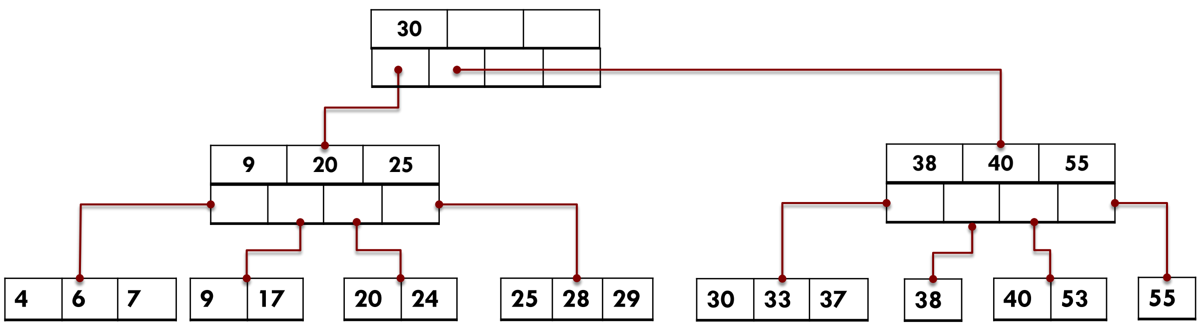
Separator keys splitting a tree into subtrees



Separator keys splitting a tree into subtrees



B-Tree: Example



B-tree Lookup Algorithm: [1/4]

- To find an item in a B-Tree, we perform a single **traversal from root to leaf**
- The objective of this search is to find the key or its predecessor
  - Finding an exact match is used for point queries, updates, and deletions
  - Finding its predecessor is useful for range scans and inserts

B-tree Lookup Algorithm: [2/4]

- Index keys split the tree into **subtrees**
  - With boundaries between two neighboring keys
- The algorithm starts from the root and performs a binary search
  - This locates a subtree
- As soon as we find the subtree?
  - Follow the pointer that corresponds to it
  - Repeat search

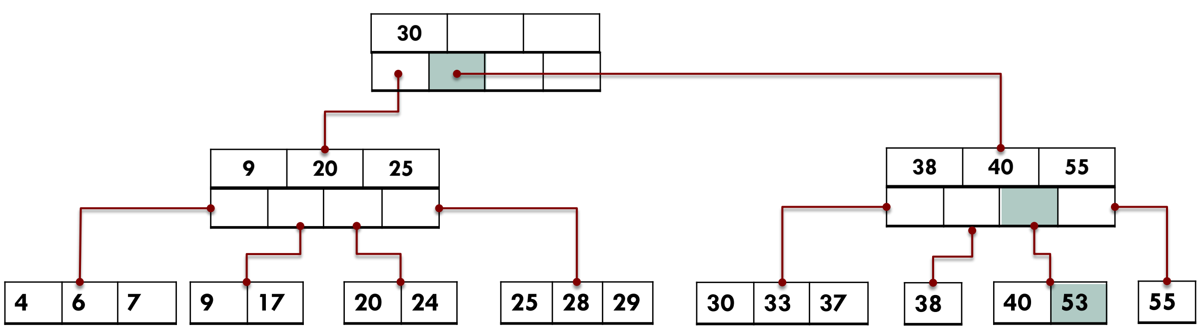
B-tree Lookup Algorithm: [3/4]

- On each level, we get a more detailed view of the tree:
  - We start on the most coarse-grained level (the root of the tree)
  - Descend to the next level where keys represent more precise, detailed ranges
  - Until we finally reach **leaves**, where the data records are located

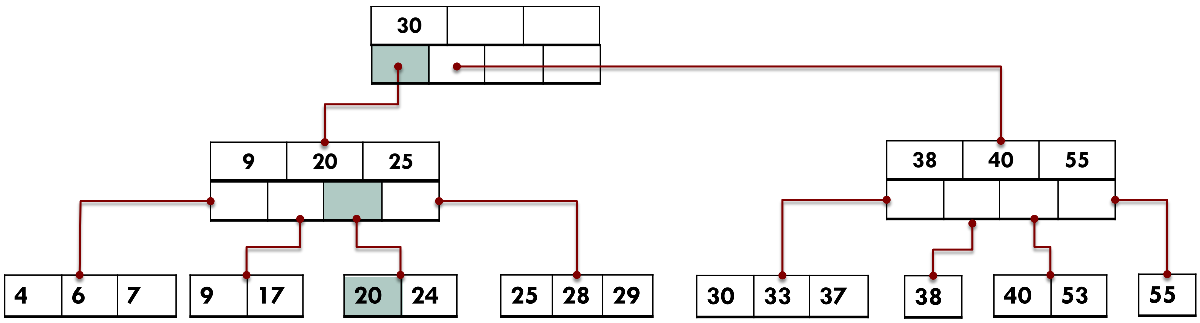
B-tree Lookup Algorithm: [4/4]

- During the point query
  - Search completes after finding (or failing to find) the target key
- During the range scan
  - Sibling pointers are followed until the end of the range is reached or the range predicate is exhausted

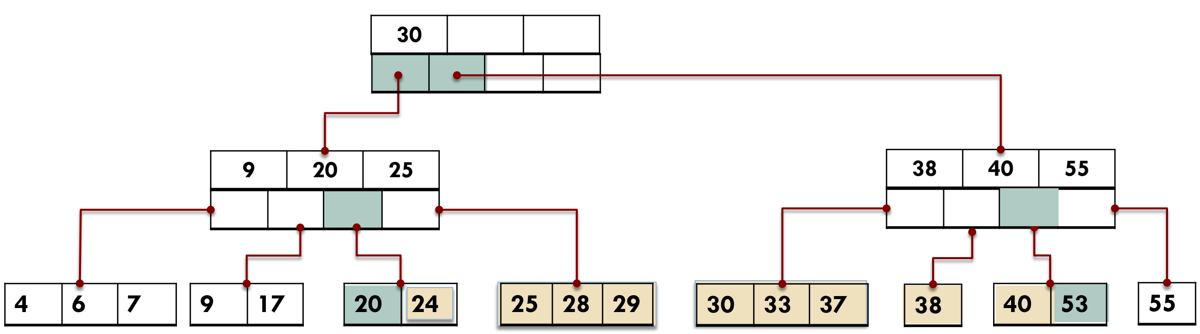
B-Tree: Example: Looking for 53



B-Tree: Example: Looking for 20



B-Tree: Example: Looking for 20 < x < 53



The contents of this slide-set are based on the following references

- Alex Petrov. *Database Internals*. ISBN-10/13: 1492040347/978-1492040347 O'Reilly Media. [Chapters 2,4]
- Martin Kleppmann. Designing Data-Intensive Applications. ISBN-10/13: 1449373321/ 978-1449373320. O'Reilly Media. [Chapter 3]