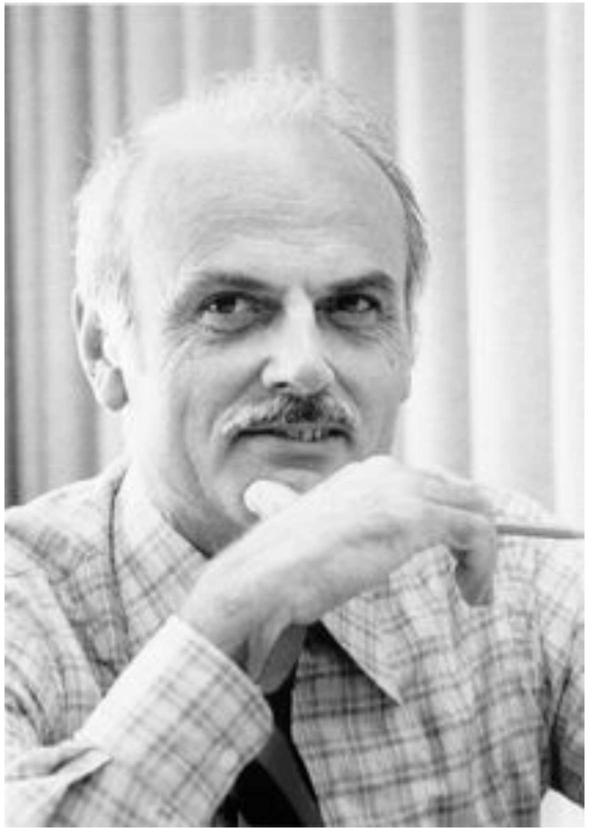
MONTANA
STATE UNIVERSITY

B Trees

Database Implementation

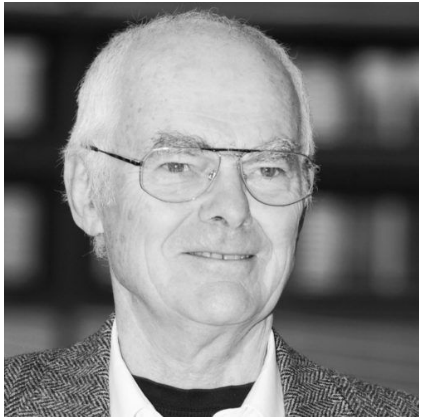
Last Lecture

- In the last lecture we looked at the theoretical basis for relational databases
- In this lecture we will look at the primary data structure used for the *implementation* of databases: the *B Tree*



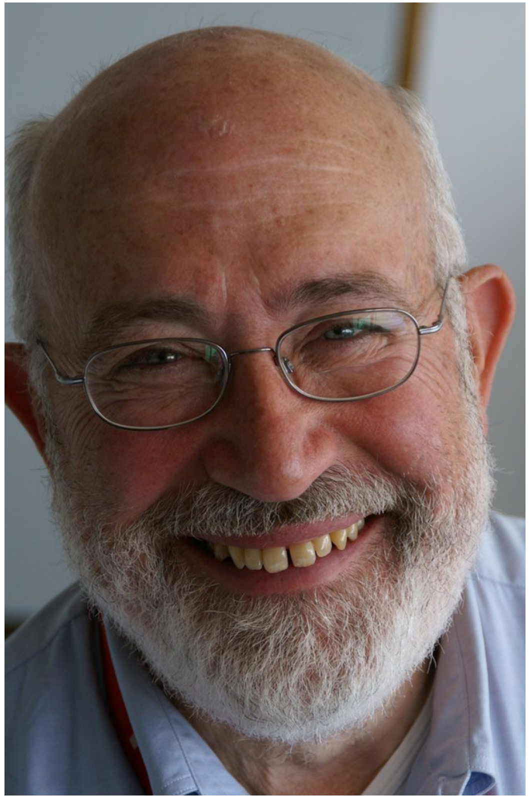
B Tree History

- Invented by Rudolf Bayer & Edward McCreight at Boeing
- Rudolf Bayer
  - German professor
  - Also invented the UB-Tree and the red-black tree algorithms
- Edward McCreight
  - US computer scientist
  - Also co-designed the Xerox Alto



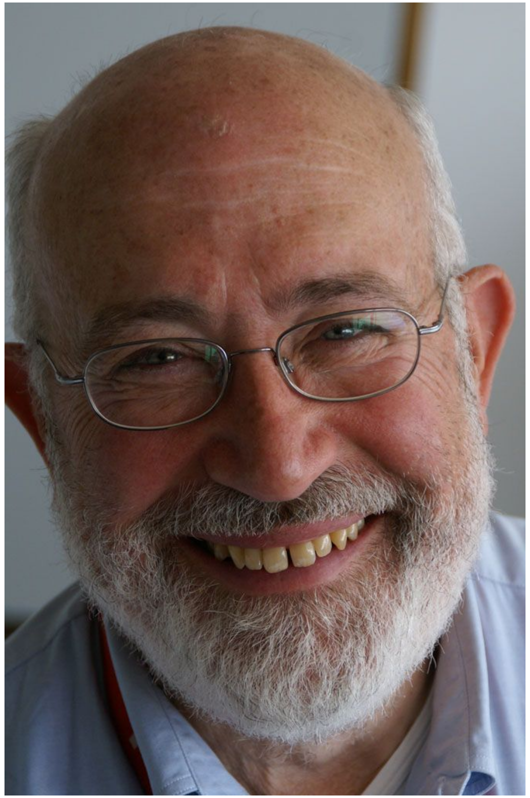
Why “B” Tree?

- What does the B stand for?
- Binary? No, B tree nodes can have more than 2 children…
- Some other Suggestions
    - Boeing
    - Balanced
    - Broad
    - Bushy
    - Bayer



Why B Tree?

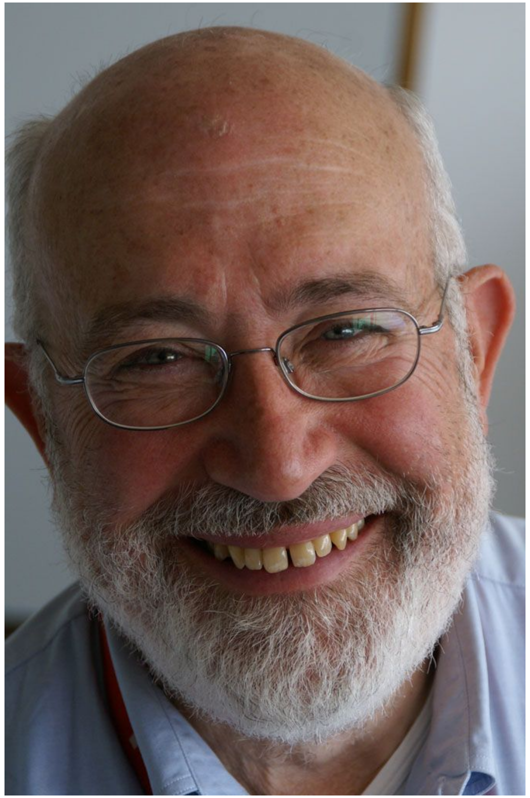
“I am occasionally asked what the B in B-Tree means. I recall it as a lunchtime discussion that you never in your wildest dreams imagine will one day have deep historical significance. We wanted the name to be short, quick to type, easy to remember. It honored our employer, Boeing Airplane Company, but we wouldn't have to request permission to use the name. It suggested Balance. Rudolf Bayer was the senior researcher of the two of us....



Why B Tree?

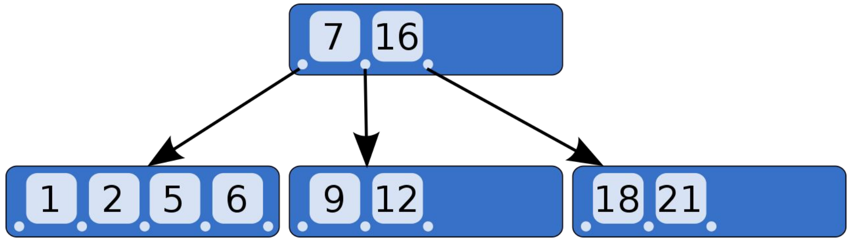
… We had been admiring the elegant natural balance of AVL Trees, but for reasons clear to American English speakers, the name BM Tree was a non-starter. I don't recall one meaning standing out above the others that day. Rudolf is fond of saying that the more you think about what the B could mean, the more you learn about B-Trees, and that is good.”

--E McCreight



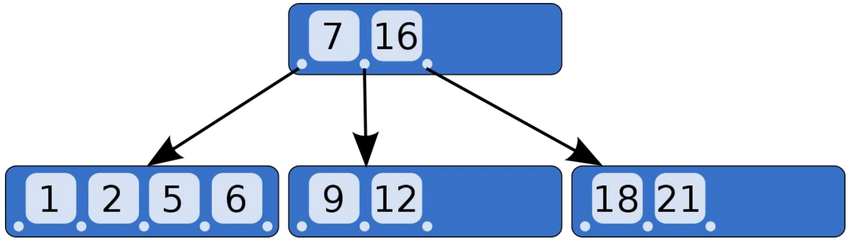
What is a B-Tree?

- Well, first off, it is obviously a *Tree*
  - A collection of hierarchically arranged data
- It is also *self balancing*
  - No branch can get too deep compared with other branches
- As a tree, it supports *logarithmic time operations*
  - search, insert, delete



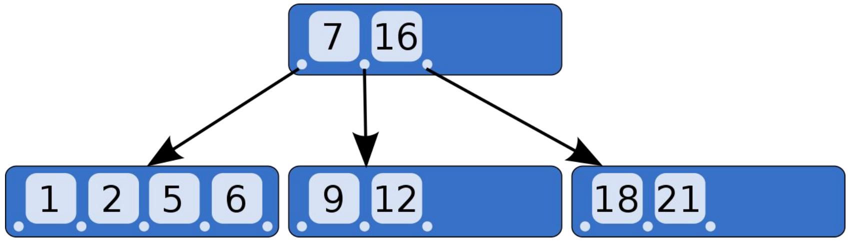
What is a B-Tree?

- B-Trees are particularly well suited for *block storage*
  - Block storage is a storage system that consists of large chunks, or blocks, of data
  - E.g. Hard drives
- Because of this advantage, B-Trees are commonly used in databases, file systems, etc.



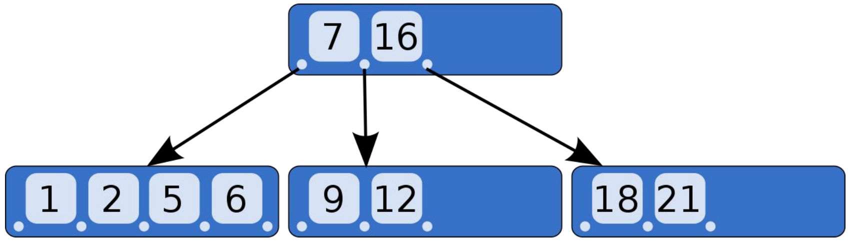
What is a B-Tree?

- Pictured at right is a simple B Tree
- The B Tree consists of *Nodes*
- Nodes hold *Keys* (values) and *Pointers*
- The top node is the *Root Node*
- The bottom nodes are all *Leaf Nodes*



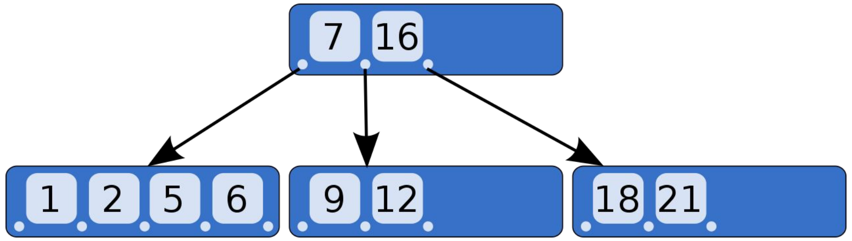
What is a B-Tree?

- Note that a node in this B tree has room for four values (keys) and five pointers
- The *Branching Factor* of this tree is 5
    - Note that the number of values available is always (*Branching Factor* - 1)
- The Branching Factor is sometimes called the *Degree* of the B Tree



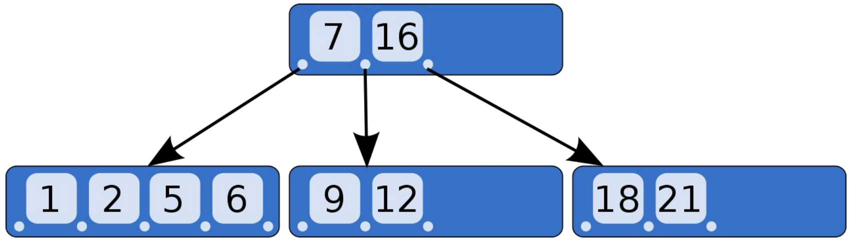
What is a B-Tree?

- Because of this relationship, you will sometimes see specific B Trees described as "*N-M B Trees*"
  - N = number of keys
  - M = number of pointers
- In our example we have a 4-5 B-Tree



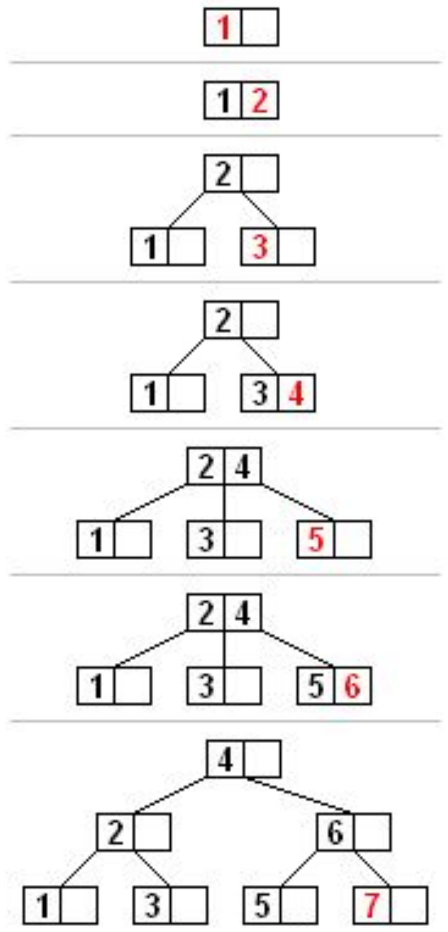
B Tree Operations - Search

- Search in a B Tree is similar to other tree search operations
- Search for value 12
  - Begin in root
  - Scan numbers
  - 12 is < 16, so follow pointer
  - Scan leaf
  - Find value at second position
- O(log(n)) - n = number of keys



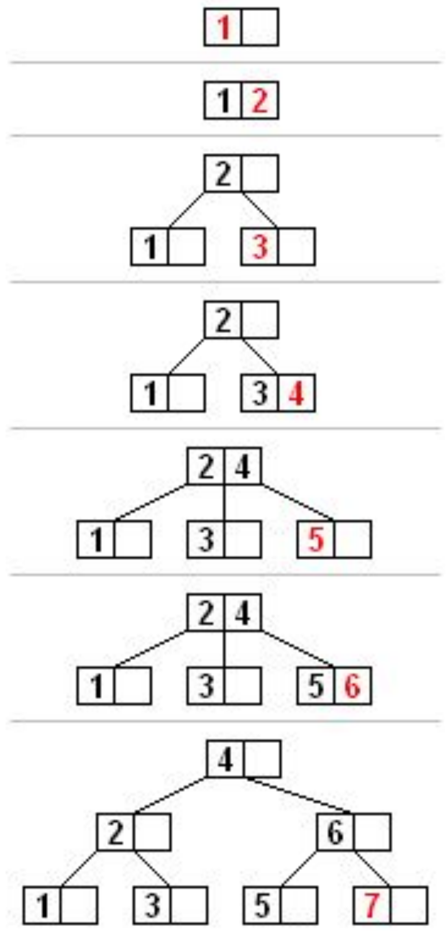
B Tree Operations - Insert

- Start at root node
- Find leaf where value is to be inserted
    - If node contains fewer than max values, insert it
    - Otherwise, split into two nodes
        - Push median value to parent
        - Less than median to right
        - Greater than median to left



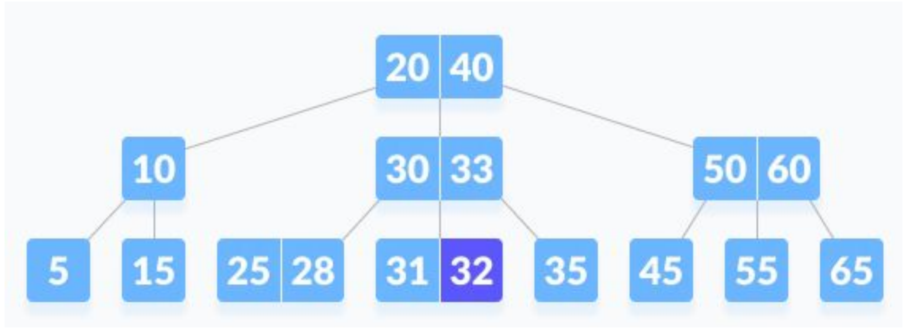
B Tree Operations - Insert

- Here we have a 2-3 B Tree
  - 2 keys
  - 3 pointers
- Insert values 1-7
- On each insert, we either add the key OR split
- Note on last insert, we recurse on splitting, creating a new root node



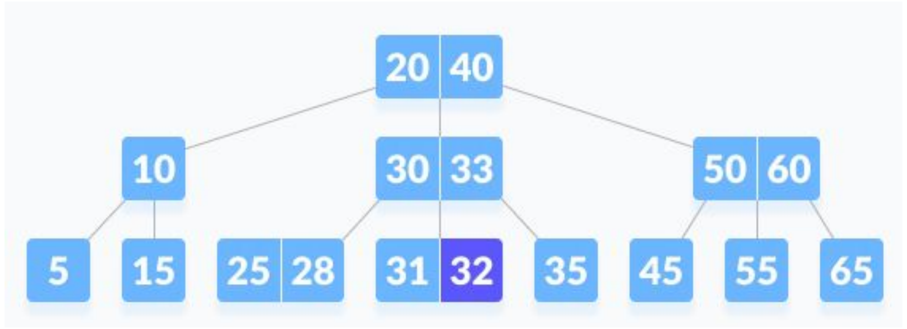
B Tree Operations - Deletion

- Deletion is a more complex algorithm
- The first concept to know is a *fill factor*
    - This is the *fewest* number of elements a node is allowed to hold
- Here we have a 2-3 b-tree with a fill factor of 0
    - A node cannot have 0 elements



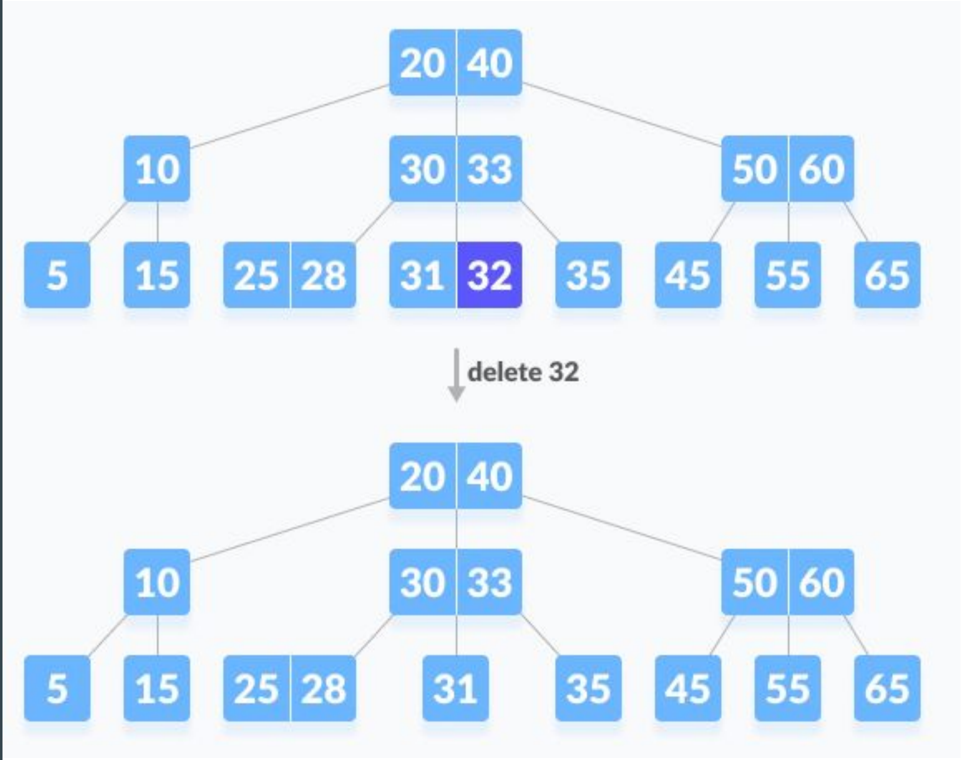
B Tree Operations - Deletion

- Deletion is a more complex algorithm
- The first concept to know is a *fill factor*
    - This is the *fewest* number of elements a node is allowed to hold
- Here we have a 2-3 b-tree with a fill factor of 0
    - A node cannot have 0 elements



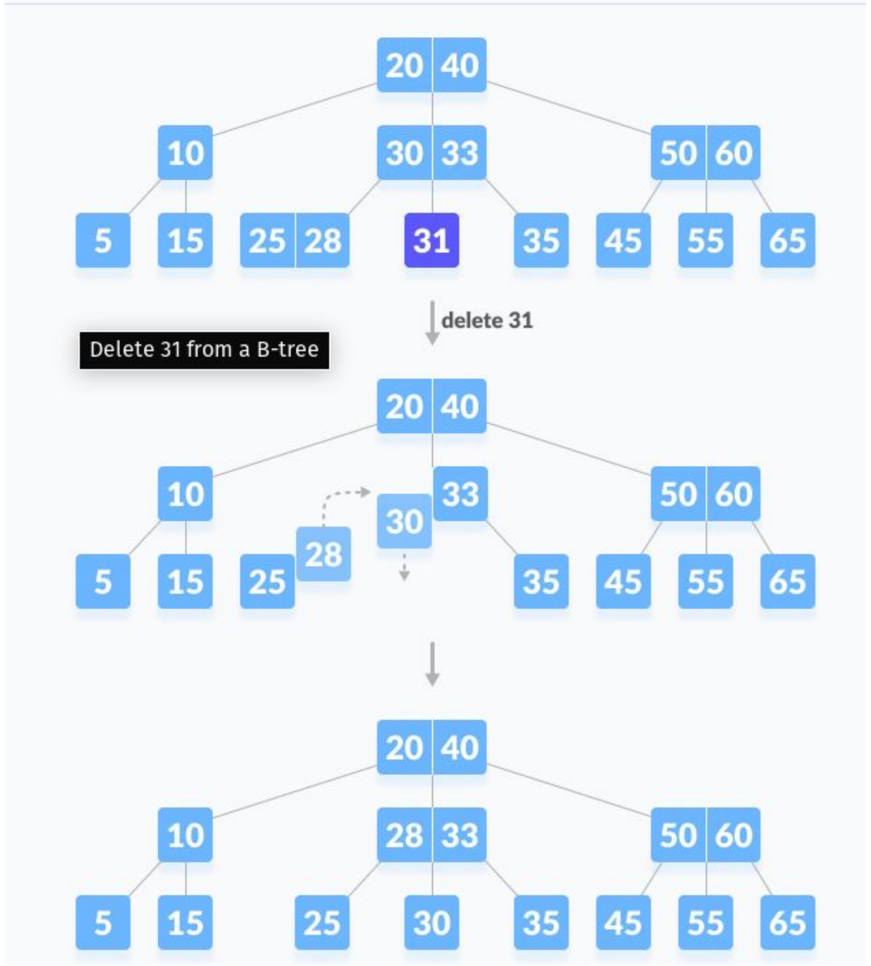
B Tree Operations - Deletion

- Case 1: Deleting a leaf value when there is no fill factor violation
- Consider deleting the number 32
- This is easy: simply remove the value



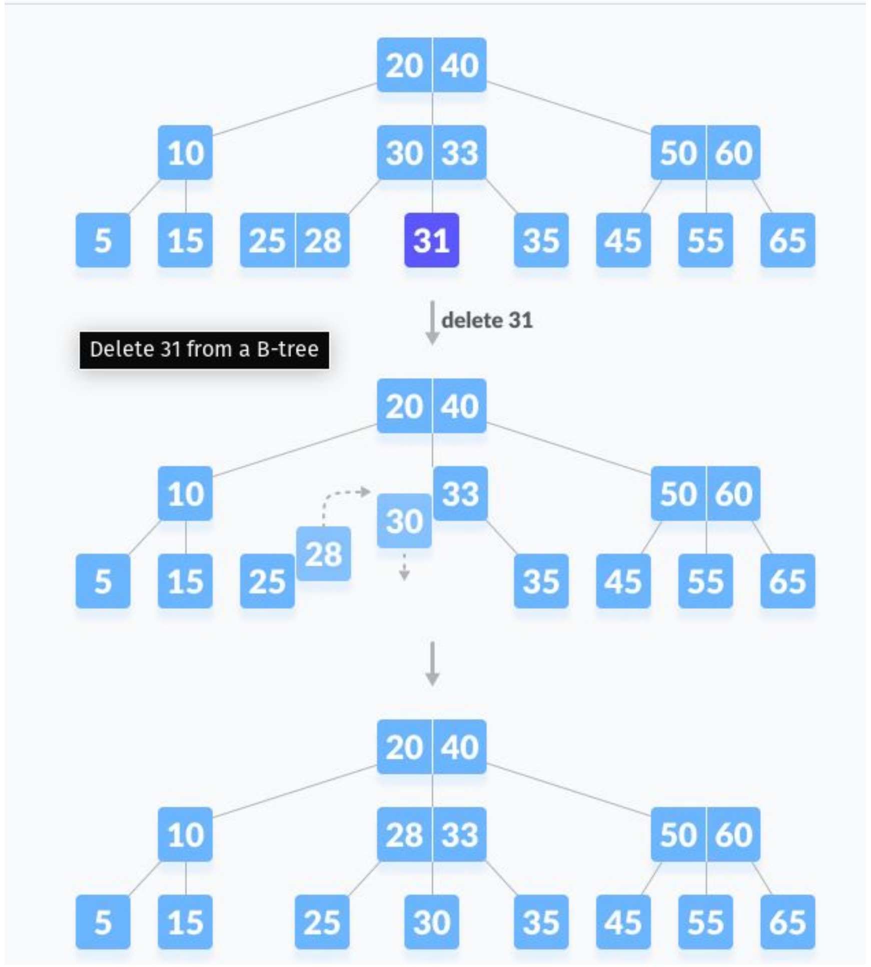
B Tree Operations - Deletion

- Case 2: Deleting a leaf value when there is a fill factor violation
- Consider deleting the number 31
- Now we have a violation: the node that held 31 has no values in it



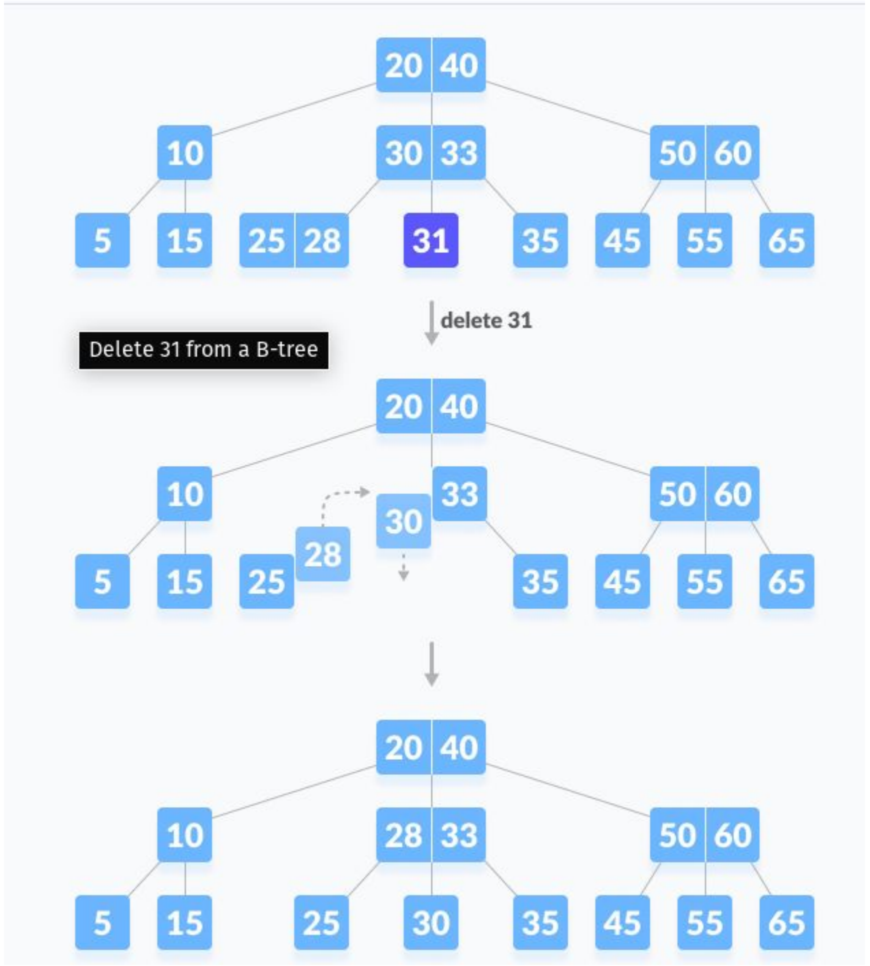
B Tree Operations - Deletion

- We *borrow* a value from the left sibling of the node
- Note the left sibling has the values 25 and 28
- We don’t want to just grab the value 28 and move it over, since it is less than 30
- So we *rotate* the value 28 up to the parent and bring the parents value, 30, down



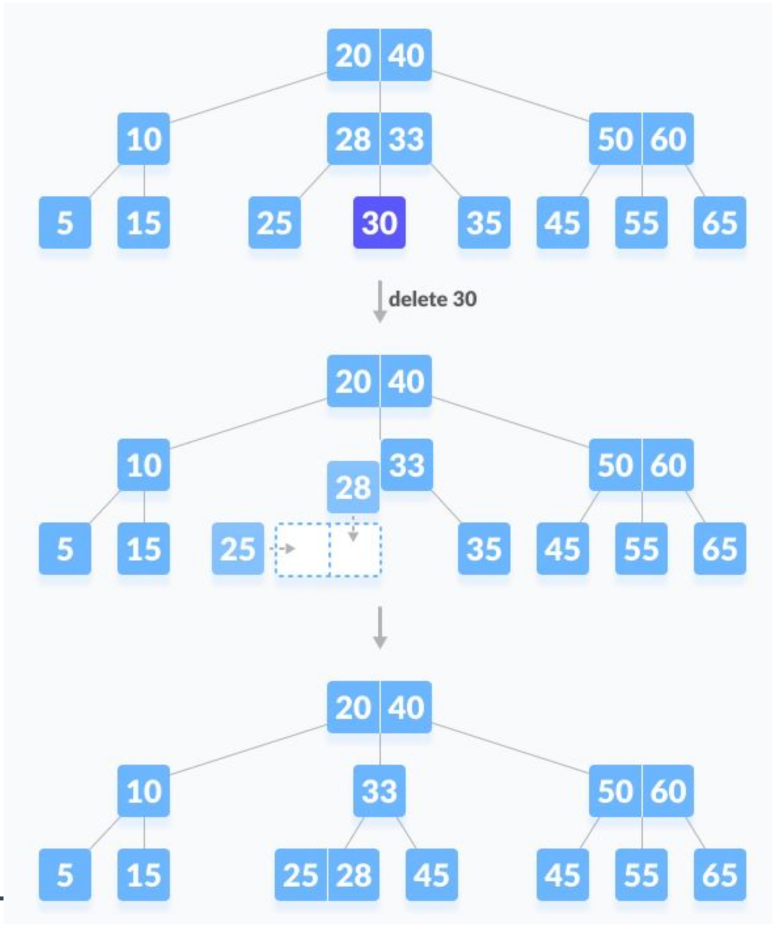
B Tree Operations - Deletion

- If the left sibling node does not have enough numbers to borrow from, we check the right sibling and do a *right rotation*



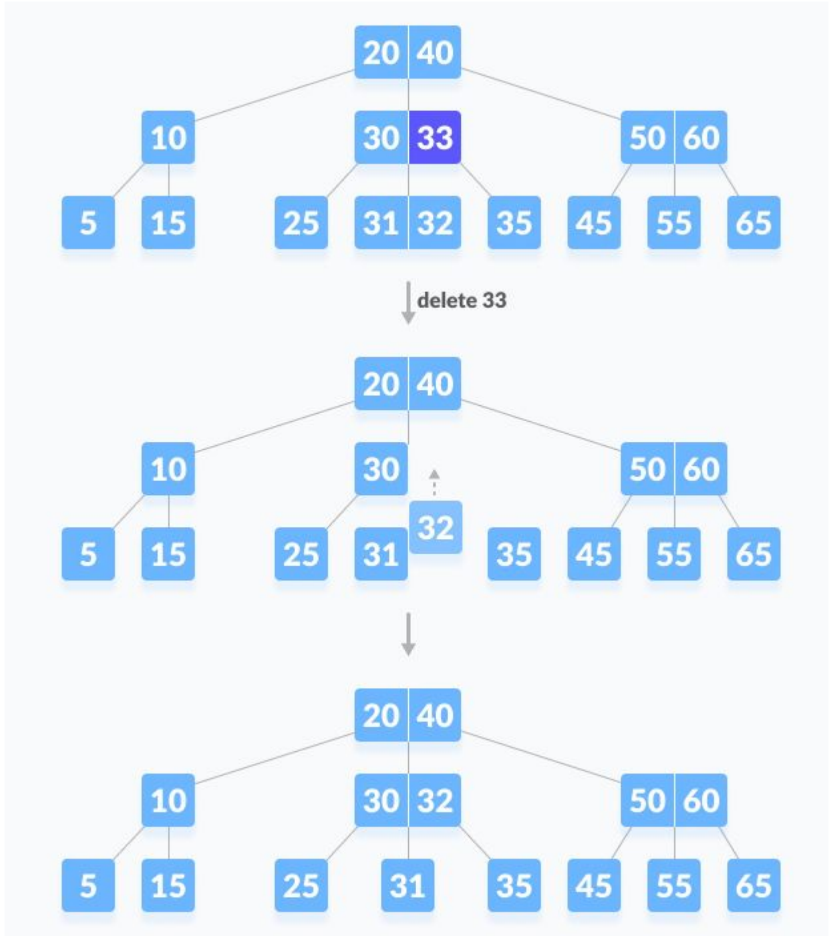
B Tree Operations - Deletion

- What if neither sibling has enough values to borrow from?
- We *merge* the siblings through the parent node
- If we delete 30 in this case, we *merge* 28 from the parent down with 25 from the left sibling



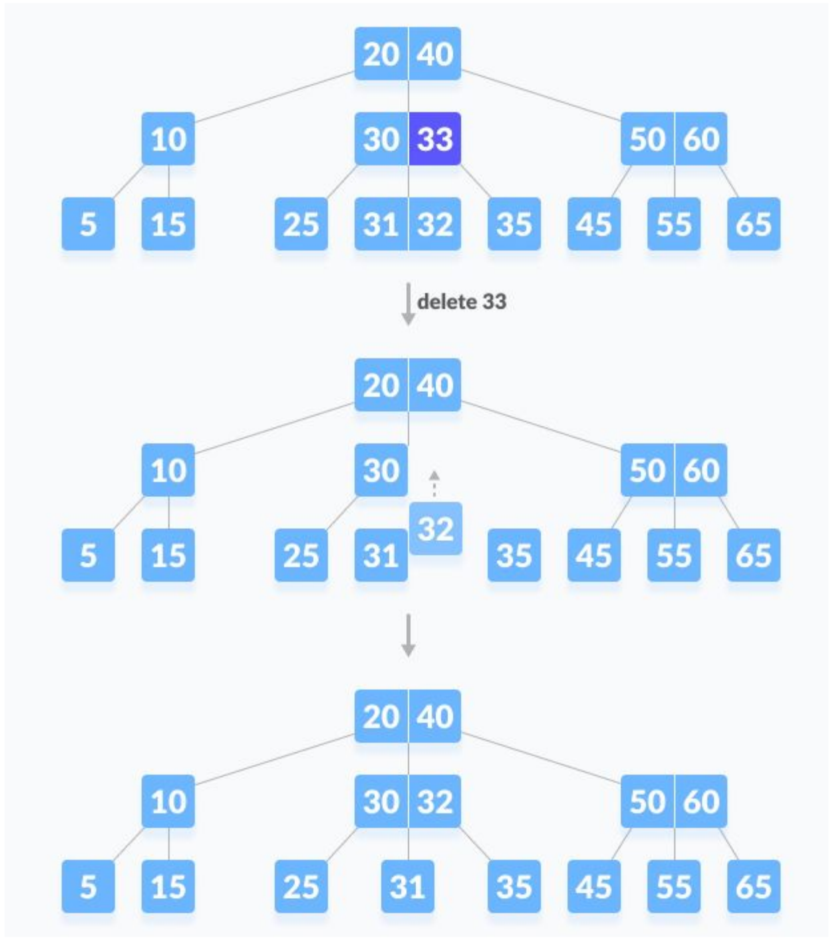
B Tree Operations - Deletion

- What about internal values?
- Consider the easy case: the child node *to the left of the deleted node* has enough values to donate
- Simply move the right most number from the left child up



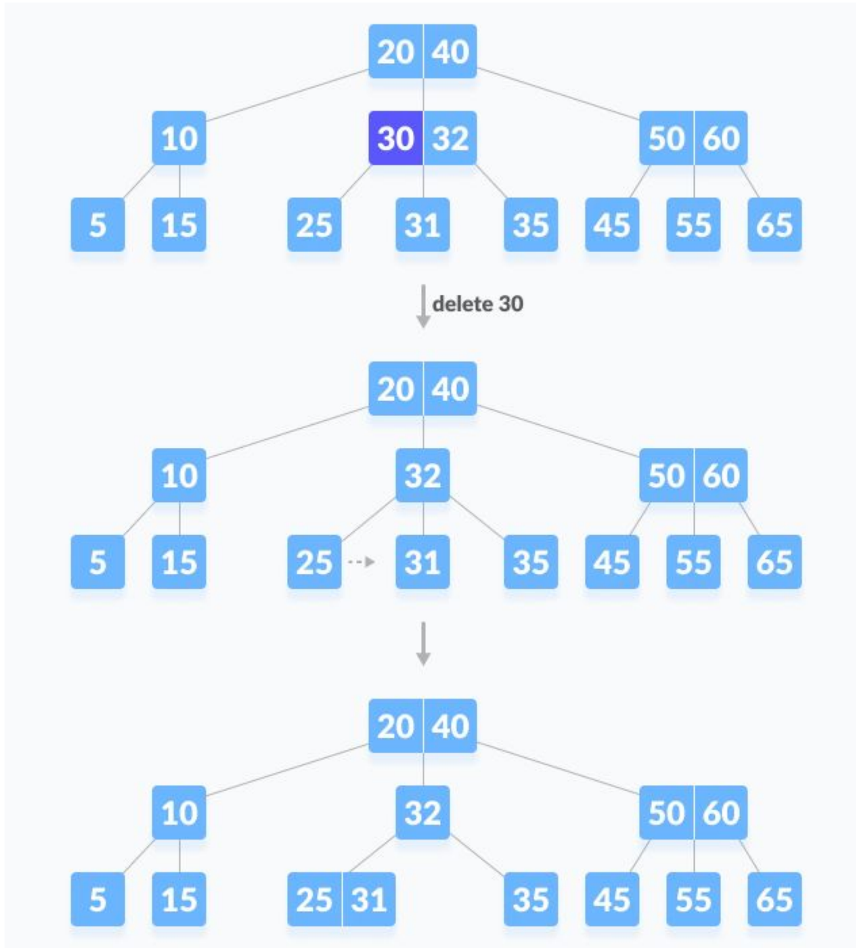
B Tree Operations - Deletion

- What about internal values?
- Consider the easy case: the child node *to the left of the deleted node* has enough values to donate
- Simply move the right most number from the left child up
  - Or vice-versa with the right child



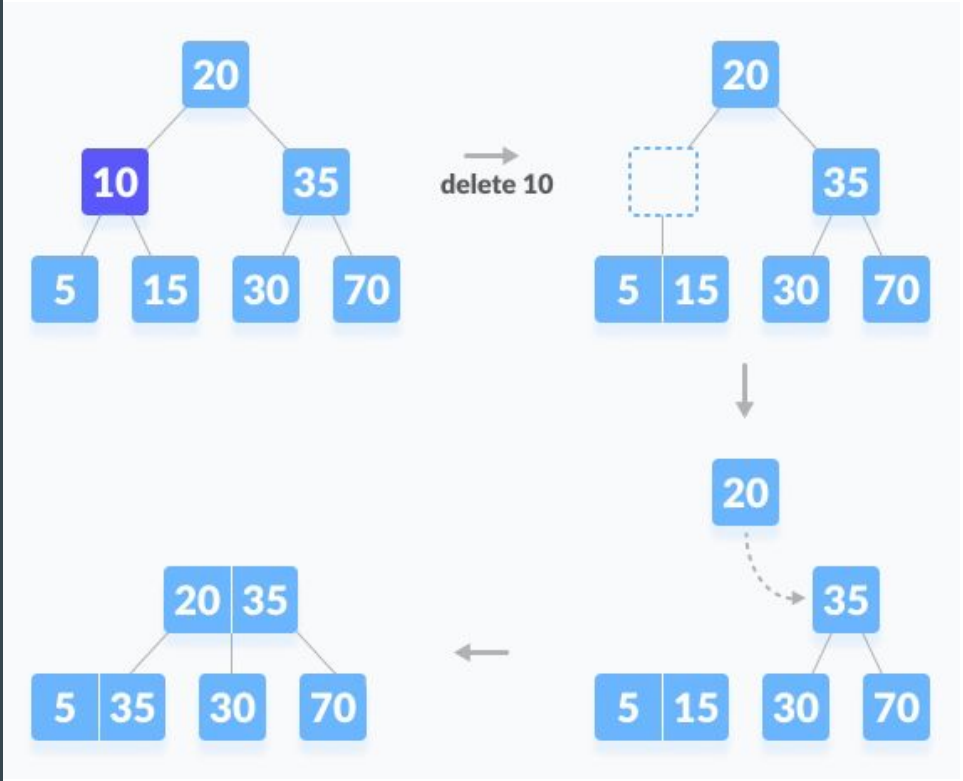
B Tree Operations - Deletion

- If both children have the minimum values possible, *merge* the two children
- Here, when deleting 30, the left and right child only have one value
- We merge then together as a single node to the left of 32



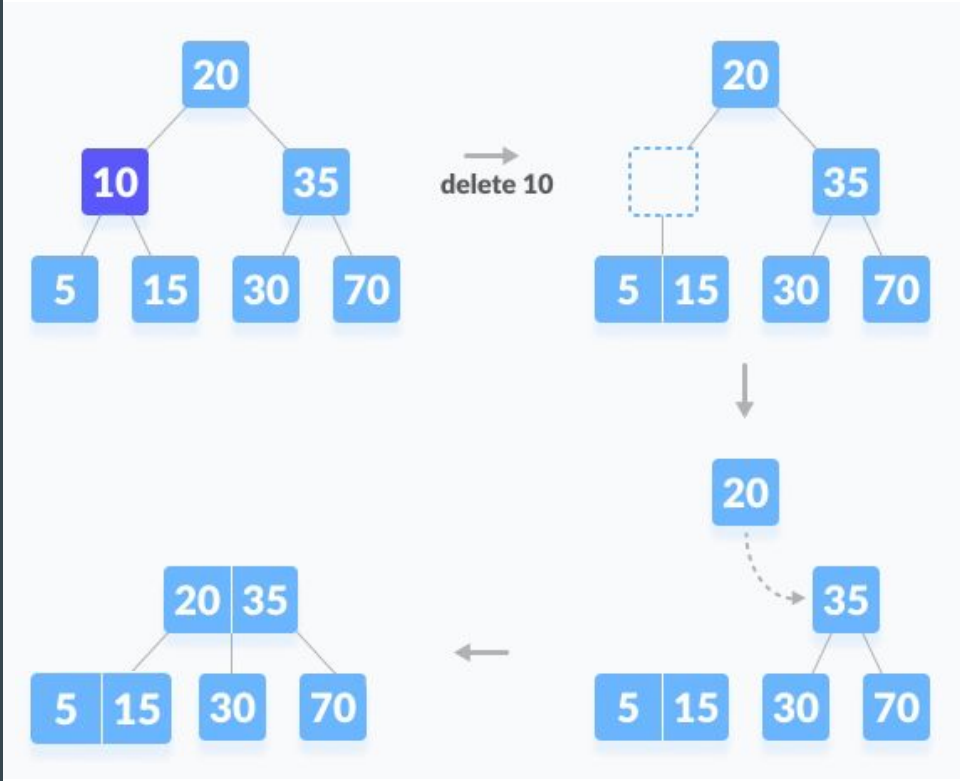
B Tree Operations - Deletion

- Final case to consider: what if an element is deleted from a node that drops it below the load factor and *no values* are available to borrow
- Here we have to shrink the tree



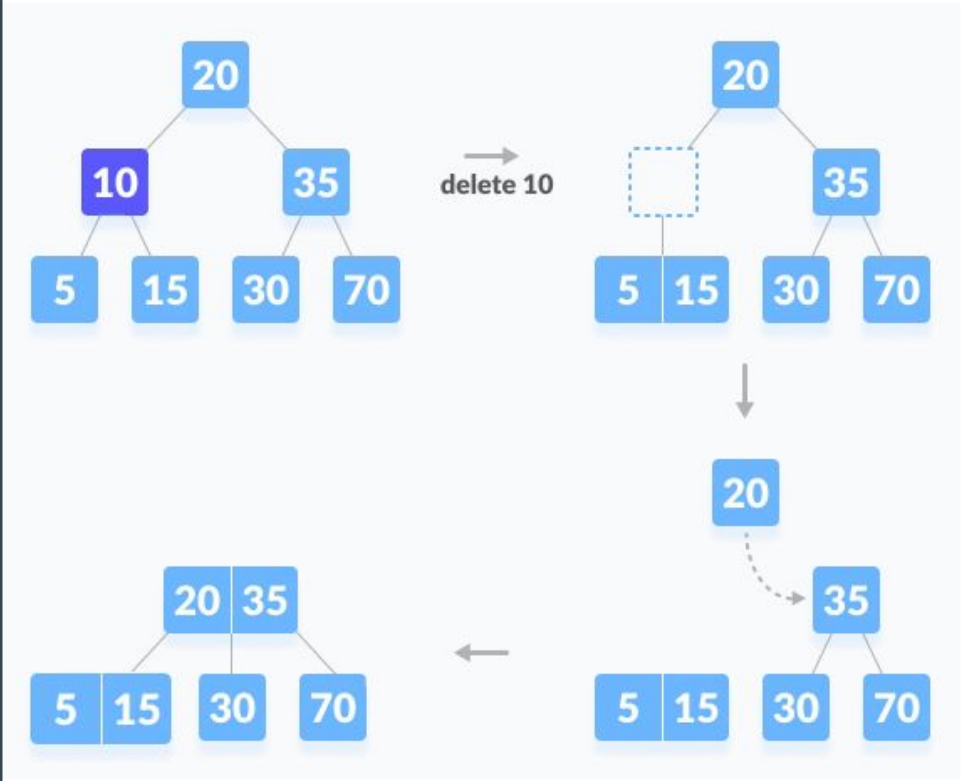
B Tree Operations - Deletion

- Consider deleting 10 from the tree to the right
- We can't simply move 5 or 15 up
- We also can't rotate the right sibling over
- We must instead merge 20 and 35 into a new root node



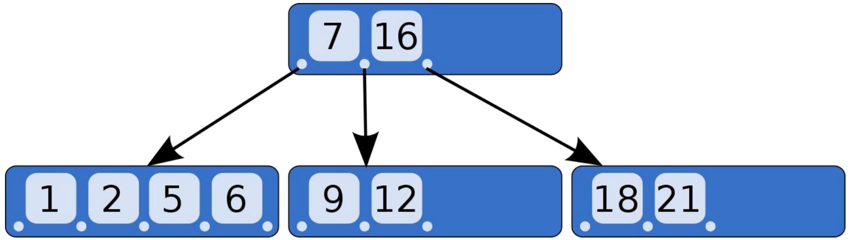
B Tree Operations - Deletion

- The original children become the left-most child of the newly formed root node
- Note that the tree shrunk in height
- This is the self-balancing nature of B-Trees



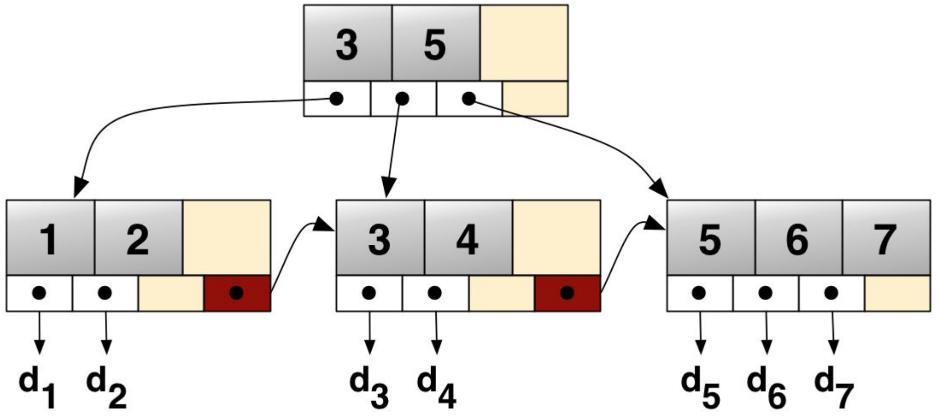
B Tree Operations - Search

- B Trees are particularly well suited for disk drives, which typically return a block of data at a time
    - Access to any sector is expensive
    - But once it is loaded it is cheap in relative terms to process
- In databases, B Trees are typically sized to disk blocks



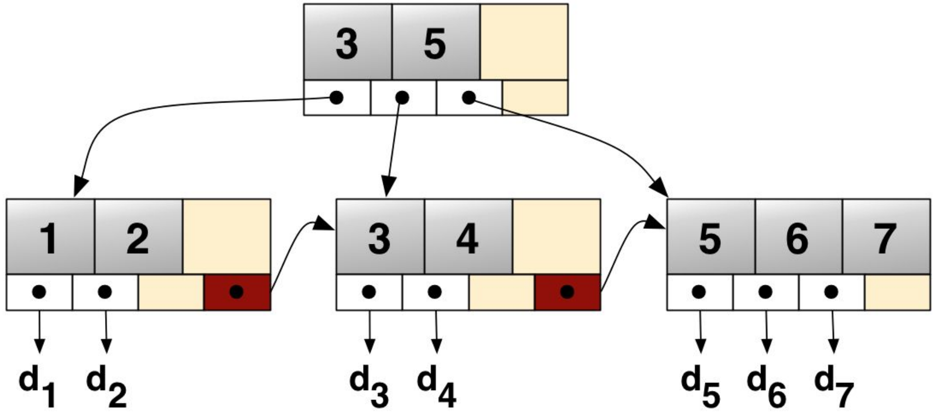
B+ Tree

- A B+ Tree is a variant of the B Tree
  - Includes all key values in leaf nodes
  - Contains a next pointer to point to the next node at a given level
  - Typically have very high degree (100+) when compared with B-Trees



B+ Tree

- Note that the next pointer makes it very fast to do an ordered iteration over all keys
    - No need to visit parent nodes, just head to the first leaf node and let it rip
- Databases typically use B+ Trees
    - In fact they are so common, people will say B Tree when they mean B+ Tree

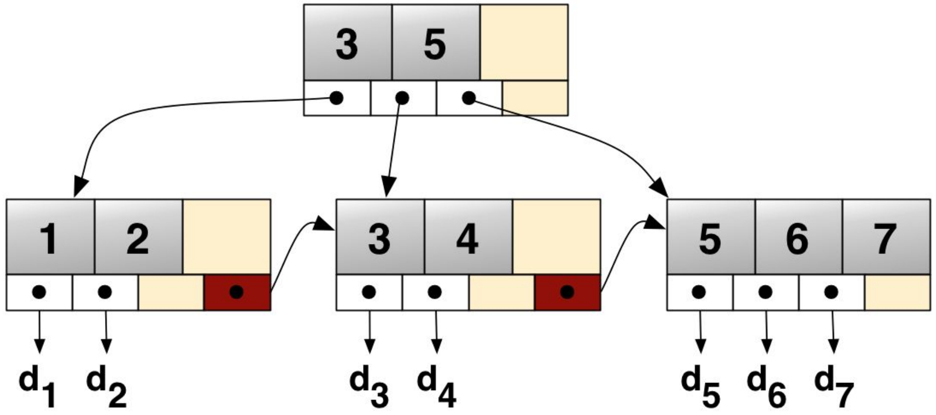


B+ Tree

- B+ Tree Simulator

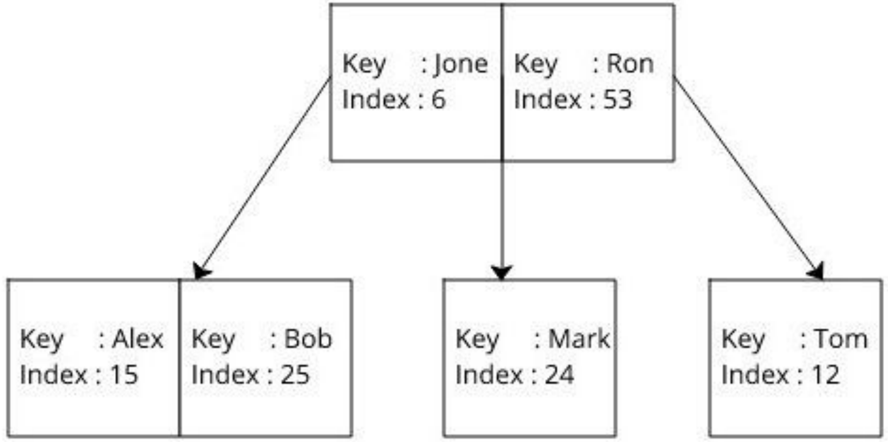
  https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

- DEMO TIME



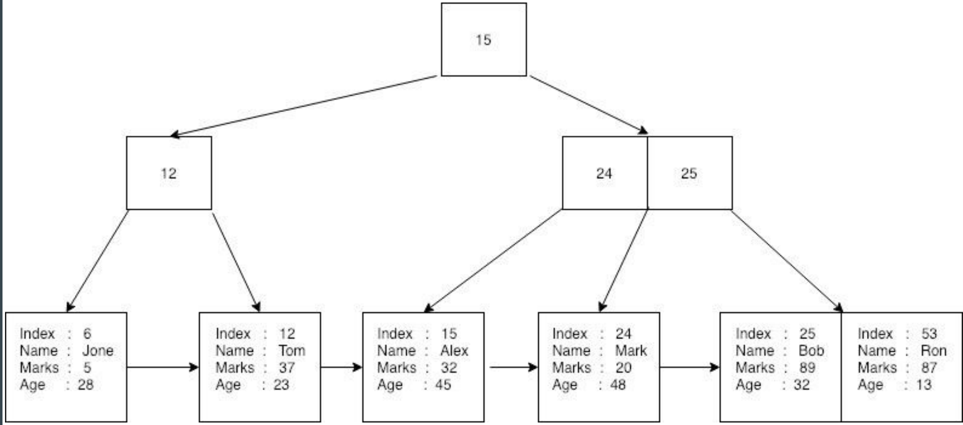
Database Usage

- What are B+ Trees used for in databases?
  - Pretty much everything!
  - Tables are laid out in B+ Trees
    - Rowid is the key
  - Indexes are smaller B+ Trees
  - Views can be implemented with B Trees



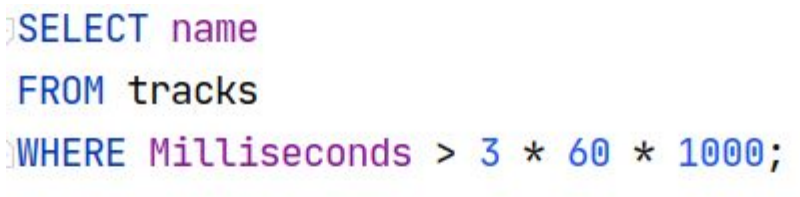
Database Usage

- Here is a table laid out on disk with a B+ Tree based on the rowid of the entry
- An index on a given value (e.g. Age) would look similar
- Note that in reality, multiple rows in the DB would be in a single node



Database Usage

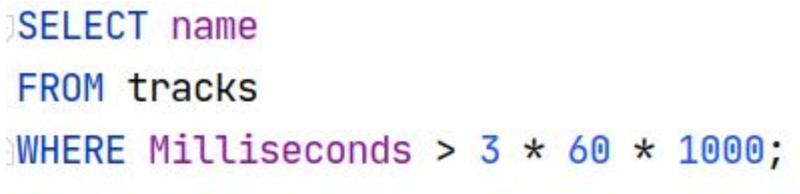
- Consider this query
  - With no index, we must scan the entire table testing the condition
  - O(n) time to compute this



```
SELECT name
FROM tracks
WHERE Milliseconds > 3 * 60 * 1000;
```

Database Usage

- If we have an index (B+ Tree) on Milliseconds, how can it be implemented?
  - Using B+ Tree, search for first leaf entry > 18000
  - From there, follow the B+ tree pointers to end
  - O(log(n)) time
  - MUCH FASTER
  - This is how indexes work



```sql
SELECT name
FROM tracks
WHERE Milliseconds > 3 * 60 * 1000;
```

B Trees

- Today we looked at how B (and B+) Trees work
- B Trees are automatically balancing trees that offer O(log(n)) operations like search, insert, delete
    - Particularly well suited for block storage technology
- We looked at B+ Trees
    - Include all values in the leaves
    - Include “Next” pointers to make scanning fast
- We looked at how B Trees can be used to implement database operations

MONTANA
STATE UNIVERSITY