B-Trees

CSE 332 Spring 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Annie Mao	Howard Xiao	Lucy Jiang
Chris Choi	Jade Watkins	Nathan Lipiarski
Ethan Knutson	Khushi Chaudhari	Richard Jiang

Announcements

- Quiz released today, due Friday!

Learning Objectives

- Describe the computer's memory hierarchy at a high level and describe how it affects AVL Trees and B-Trees differently

- Be able to implement B+ Tree find()/contains() and add() operations

Lecture Outline

- **Review: Memory Hierarchy and Data Structure Design**

- B-Trees
  - Goals and Design
  - B+ Tree Structure
  - B+ Tree Implementation: Find
  - B+ Tree Implementation: Add

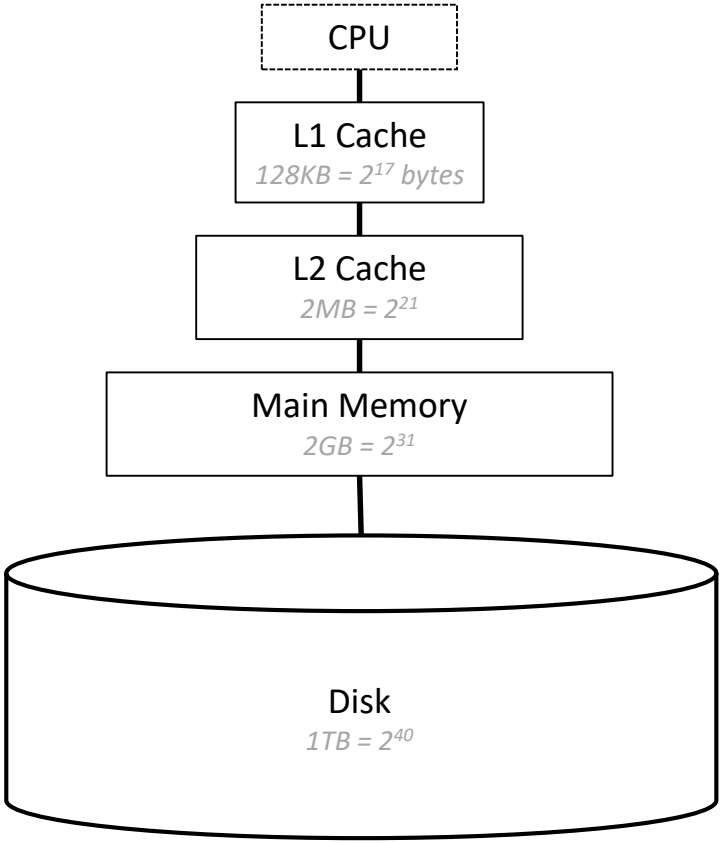
Why Does the Memory Hierarchy Matter?

- Asymptotic analysis supposedly helps us reason about large inputs. Why doesn't it work for B-trees?
  - We assumed "every memory access has an unimportant *O(1)* cost"
  - Learn more in CSE351/333/471; focus here on relevance to data structures and efficiency

```
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k) {
  for (int i=0; i < arr.length; ++i)
    if (arr[i] == k)
      return true;
  return false;
}
```

We claimed these two operations were approximately equal!

A Typical Memory Hierarchy



instructions (e.g., addition): 2^{30}/sec

fetch data in L1: 2^{29}/sec = 2 instructions

fetch data in L2: 2^{25}/sec = 30 instructions

fetch data in main memory: 2^{22}/sec = 250 instructions

fetch data from "new place" on disk: 2^{7}/sec = 8,000,000 instructions

Said In Another Way …

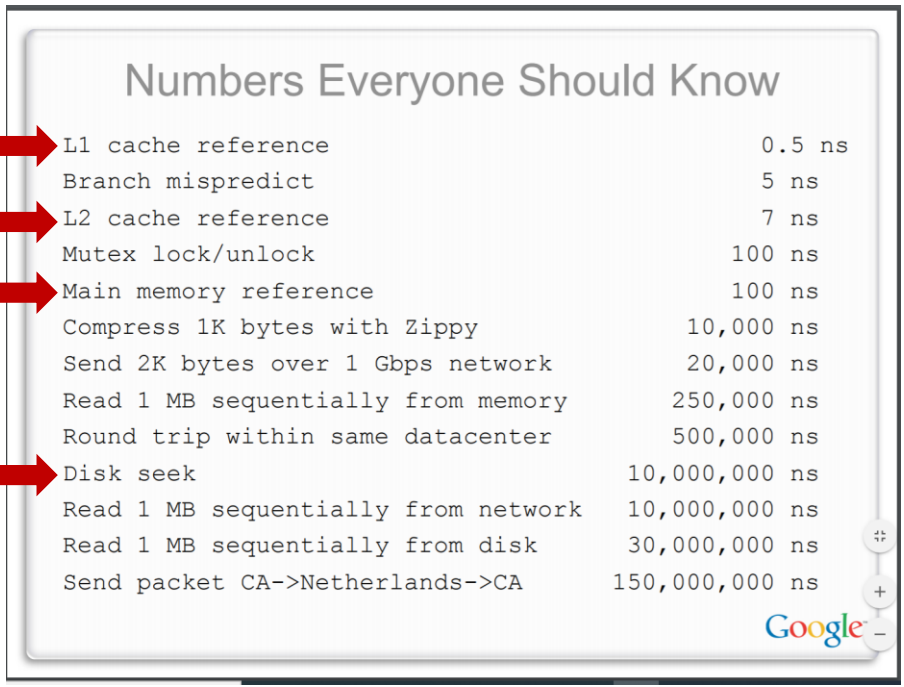
- Jeff Dean's "Numbers Everyone Should Know" (LADIS '09)

Sticky note on monitor

Asking your CSE roommate

Finding answer in textbook

Retaking 311 and then retaking 332



Hardware and OS Support

- The hardware and OS work together to automatically move data into and out of successive levels for you!
  - Replace data currently in memory/L2/L1
  - Data structures and algorithms faster if "fits in cache" (it often does)

- Most code "just works" most of the time
  - … but sometimes designing data structures and algorithms with knowledge of memory hierarchy is worth it
  - And when you do, you often need to know one more thing …

How Data Moves Around the Hierarchy

Spatial Locality

- Hardware/OS often fetches a chunk of data instead of a byte
  - Moving data up the hierarchy is slow because of the *lower level's latency* (think: distance-to-travel)
  - However, the latency is the same regardless if your program requests one byte or one chunk (think: carpool)
  - So a single fetch often causes the hardware/OS to send *nearby memory* because it's easy and likely to be asked for soon (think: object fields or arrays)

Temporal Locality

- Once a value has moved up the hierarchy, keep it around
  - A *particular value* is more likely to be accessed again in the near future than some random other value

Spatial Locality: Arrays vs. Linked Lists (1 of 3)

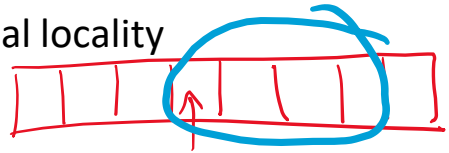
- Which has the potential to take advantage of **spatial locality**?

- Terminology:
  - The amount of data moved from **disk** into **memory** is called the "block" size or the "page" size
  - The amount of data moved from **memory** into **cache** is called the cache "line" size

- Reminder:
  - Neither the movement nor the sizes are under programmer control!

Spatial Locality: Arrays vs. Linked Lists (2 of 3)

- An array benefits more than a linked list from spatial locality
  - Language (e.g., Java) implementation can put LL nodes anywhere, whereas an array is typically implemented as contiguous memory
  - Contiguous memory benefits from spatial locality



- Suppose 2^{23} items of 2^{7} bytes each. They are stored on disk and the block size is 2^{10} bytes
  - An **array** needs 2^{20} disk accesses
    - If "perfectly streamed", > 4 seconds
    - If "random places on disk", 8000 seconds (> 2 hours)
  - A **linked list** *in the worst case* needs 2^{23} disk accesses
    - Assuming "random" placement around disk, >16 hours

Spatial Locality: Arrays vs. Linked Lists (3 of 3)

- However! "Array" doesn't necessarily mean "good"
  - Binary heaps "make big jumps" to percolate
  - Constantly loading/unloading different blocks from disk

What About BSTs? (1 of 2)

- Operations on balanced BSTs are $O(\log n)$
  - Even for $n = 2^{39}$ (512 GB) we "should be ok"
- Still, number of disk accesses matters:
  - Pretend for a minute we had an AVL tree of height 55
  - The total number of nodes could be: $1.62^{55} - 2^{55}$ (min/spindliest AVL; perfect/full AVL)
  - Most of the nodes will be on disk: the tree is shallow, but it is still many gigabytes big so the entire *tree* cannot fit in memory
    - Even if memory holds the first 25 nodes on our path, we still potentially need 30 disk accesses if we are traversing the entire height of the tree.

What about BSTs? (2 of 2)

If your data structure is mostly on disk, minimize disk accesses!

- In this scenario, a better data structure would exploit the block size and (relatively) fast memory access to ***avoid disk accesses***

Lecture Outline

- Review: Memory Hierarchy and Data Structure Design

- B-Trees
  - **Goals and Design**
  - B+ Tree Structure
  - B+ Tree Implementation: Find
  - B+ Tree Implementation: Add

Our B-Tree Goal

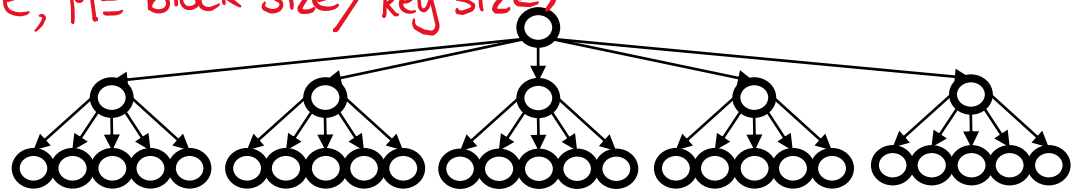
- **Problem**: A dictionary with so many items *most of it is on disk*
- **Desire**: A balanced tree (logarithmic height) that minimizes disk accesses and exploits disk-block size
- **An idea**: Increase the branching factor of our tree

Reminder: a dictionary maps *keys* to *values*; an *item* or *data* refers to the (key, value) pair

M-ary Search Tree

- A search tree with branching factor M (instead of 2)
  - Has a sorted array of children: `Node[]`
  - Choose M to fit into a disk block; only 1 disk access for entire array!

(ie, M= block size / key size)



- Perfect tree of height h has $(M^{h+1}-1)/(M-1)$ nodes
  - Weiss, page 4

M-ary vs Binary: Performance Comparison

- # hops for `find`/`contains`?
  - If we have a **balanced** trees: $\log_M n$ hops (M-ary) vs $\log_2 n$ (binary)
  - Eg: M = 256 (=2^8) and n = 2^{40}, M-ary makes 5 hops vs 40 hops

- For each internal node, how do we decide which branch/which child to take?
  - Binary tree: Less than vs greater than node's key? 1 comparison
  - M-ary: In range 1? In range 2? In range 3?... In range M?
    - Linear search the Node[]: M comparisons
    - Binary search the Node[]: $\log_2 n$ comparisons

- Runtime of M-ary `find`/`contains`: O($\log_2 M \log_M n$)
  - Remember: we're assuming a balanced M-ary tree!

Remaining Considerations for a B-Tree (1 of 2)

- Assuming a balanced M-ary tree, we can make a reasonably cache-aware B-Tree out of an M-ary tree Yay!

- Other design questions to answer:
  - What should the **order** property be?
    - M-way extension of a BST's 2-way ordering property: the subtree **between** keys ***a*** and ***b*** contains the keys between them; ie ***a ≤ k < b***
    - Note: only need M-1 keys to represent M ranges/subtrees
  - What should the **structure** propert(ies) be?
    - How would you **rebalance** (ideally without more disk accesses)?
    - Where should we store the key's **value?**

Remaining Considerations for a B-Tree (2 of 2)

- Remember that a Dictionary ADT stores key->value pairs. Where should we store a key's **value?**
  - A BST stores the value alongside the key at every node
  - In contrast, our B-Tree will need to load the entire node from disk, even though we are usually just "passing through" it on the way to the key we're looking for
  - Having a B-Tree store keys and values in the same node means loading values from disk *even though we won't use them!!*

Lecture Outline

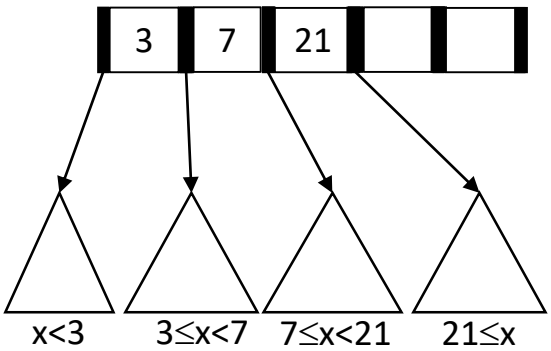
- Review: Memory Hierarchy and Data Structure Design

- B-Trees
  - Goals and Design
  - **B+ Tree Structure**
  - B+ Tree Implementation: Find
  - B+ Tree Implementation: Add

B+ Tree Node Structure

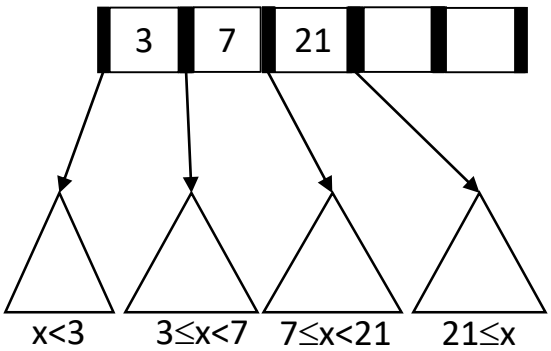
Both the textbook and we refer to "B+ Trees" as "B-Trees", but "B-Trees" actually encompass several variants

- Two types of nodes: **internal** and **leaf**
  - Each **internal node** contains up to *M-1* keys (for up to *M* children)
    - Does not store values, only keys
    - Function as "signposts"
  - Each **leaf node** contains up to *L* items
    - Stores (key, value) pairs
    - As usual, we'll ignore the "along for the ride" value in our examples



B+ Tree Parameters

- Two parameters, one for each type of node:
  - ***M***, the number of keys in an **internal** node
    - Recommend M* ≈ diskBlockSize/keySize
  - ***L***, the number of items in a **leaf** node
    - Recommend L = diskBlockSize/(keySize + valueSize)
- Any *M* > 2 and *L* will technically work, but picking *M* and *L* based on disk-block size maximizes B+ Tree's efficiency



** More precisely, we recommend*
M = (diskBlockSize + keySize)/(keySize + pointerSize)

B+ Tree Structure

- **Internal nodes**
  - Have between $\lceil M/2 \rceil$ and M children; i.e., at least half full
  - Reminder: no values, just keys
- **Leaf nodes**
  - All leaves at the same depth
  - Have between $\lceil L/2 \rceil$ and L items; i.e., at least half full
  - Reminder: keys *and* values
- **Root node** – A Special Case!
  - If tree has $\leq L$ items, root is a **leaf node**
    - Only occurs when starting up; otherwise unusual
  - Else, root is an **internal node** and has between 2 and M children
    - i.e., the "at least half full" condition does not apply

B+ Trees are Balanced (Enough)

- Not hard to show height *h* is logarithmic in number of items *n*
  - Let $M > 2$ (if $M = 2$, then a "linked list tree" is legal – no good!)
  - Because all nodes are at least half full *(except possibly the root)* and all leaves are at the same level, the minimum number of items *n* for a height *h>0* tree is

$$n \geq 2\lceil M/2 \rceil^{h-1} \lceil L/2 \rceil$$

minimum number of leaves

minimum items per leaf

Example: B+ Tree is Shallower Than AVL Tree

- Suppose we have 100,000,000 items

- Maximum height of AVL tree?
  - Recall S(h) = 1 + S(h-1) + S(h-2)
  - So: **37**

- Maximum height of B+ Tree with M=128 and L=64?
  - Recall $n \geq 2\lceil M/2 \rceil^{h-1} \lceil L/2 \rceil$
  - So: **5** (and 4 is more likely)

Are B+ Trees Disk Friendly?

Yes!!!

- Many keys stored in one **internal node**
  - *If we pick M wisely*, entire node can be loaded into memory with one disk access
  - Binary search over *M*-1 keys insignificant compared to disk access

- **Internal nodes** only contain keys
  - Any `find`/`contains` only wants *one value*; no need to load "incorrect" values into memory
  - Only need one disk access to bring (the single correct) value into memory: when we find the correct **leaf node**
  - If sizeof(keyType) << sizeof(valueType), can hold significantly more B+ Tree-style nodes in memory than BST-style nodes (which co-locates keys and potentially-very-large values)

Lecture Outline

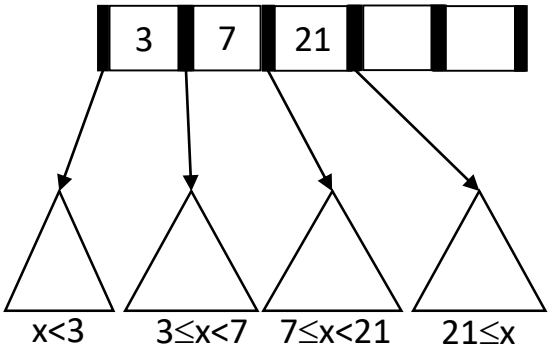
- Review: Memory Hierarchy and Data Structure Design

- B-Trees
  - Goals and Design
  - B+ Tree Structure
  - **B+ Tree Implementation: Find**
  - B+ Tree Implementation: Add

B+ Tree Find/Contains

- M-way extension of a BST's root-to-leaf recursive algorithm
  - At each internal node, do binary search on (up to) *M*-1 keys to find the branch to take
  - At the leaf node, do binary search on the (up to) *L* items

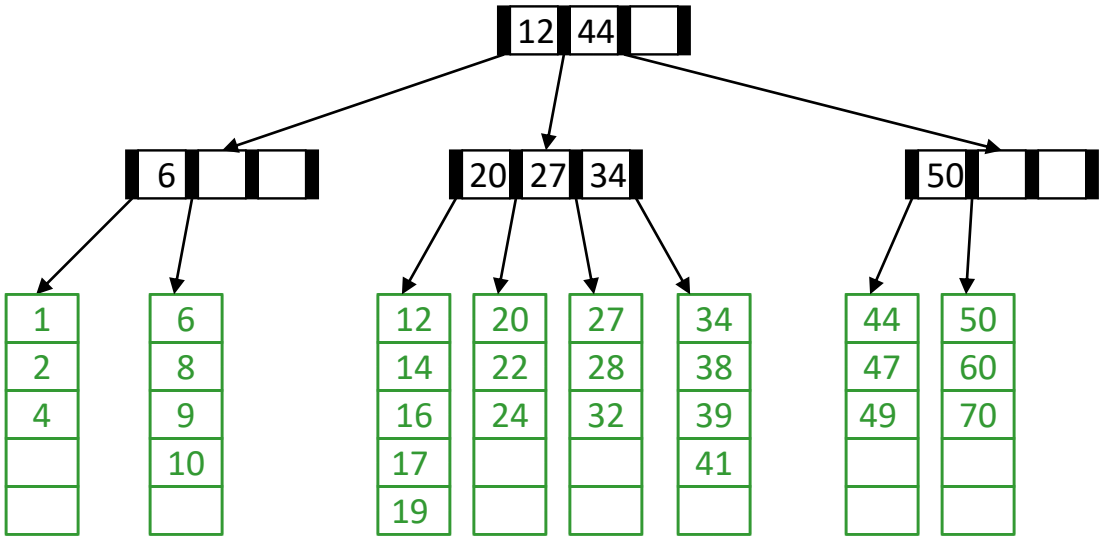
- Difference:
  - We *don't store value at internal nodes*, so there is no "best case" of finding our value at the root node; must always traverse to the bottom of B+ Tree



Find/Contains Example

Notation:
- Internal nodes drawn horizontally
- Leaf nodes drawn vertically
- All nodes include empty cells

- Tree with *M*=4 (max #pointers in **internal node**) and *L*=5 (max #items in **leaf node**)
  - All **internal nodes** must have ≥2 children
  - All **leaf nodes** must have ≥3 items (but we are only showing keys)



Maintaining B+ Tree Balance When Mutating …

- So this seems like a great data structure (and it is)
- But we haven't discussed other Dictionary operations yet:
  - `add`
  - `remove`
- As with AVL trees, maintaining structure is *hard*!
  - E.g., for `add`, there might not be room at the correct leaf

Lecture Outline

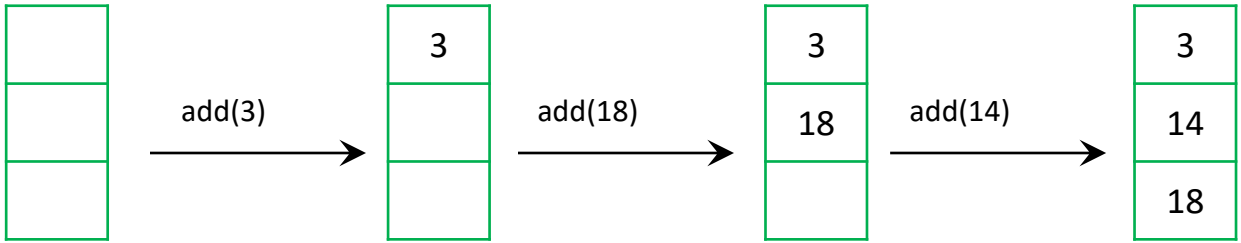
- Memory Hierarchy and Data Structure Design

- B-Trees
  - Goals and Design
  - B+ Tree Structure
  - B+ Tree Implementation: Find
  - **B+ Tree Implementation: Add**

Add Example:

- Add 3, 18, 14, 30, 32, 36, 15, 16, 12, 40, 45, 38
- M=3, L=3

Add Example: Answer (1 of 7)

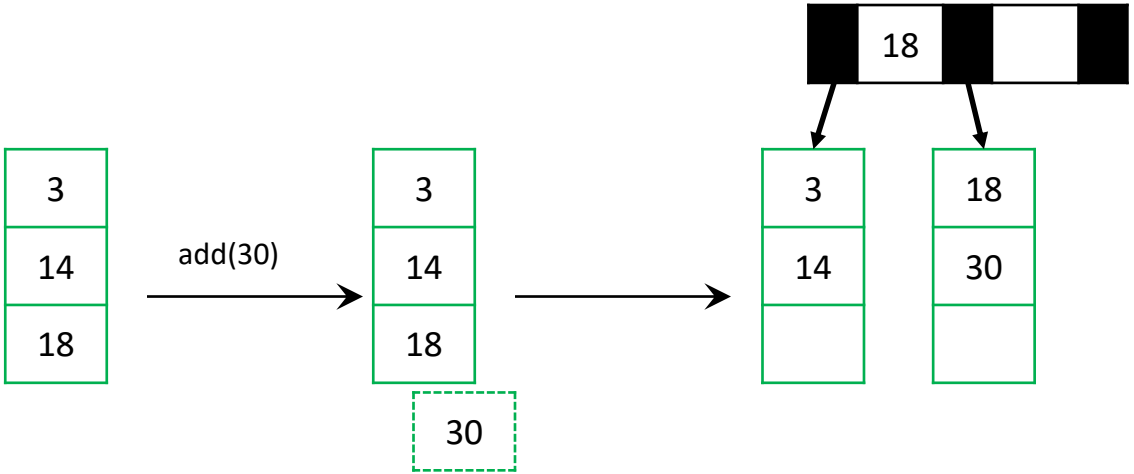


Special case: the
root is a **leaf node**

Values (omitted) are sorted in key order

Add 3, 18, 14, 30, 32, 36, 15, 16, 12, 40, 45, 38
M=3, L=3

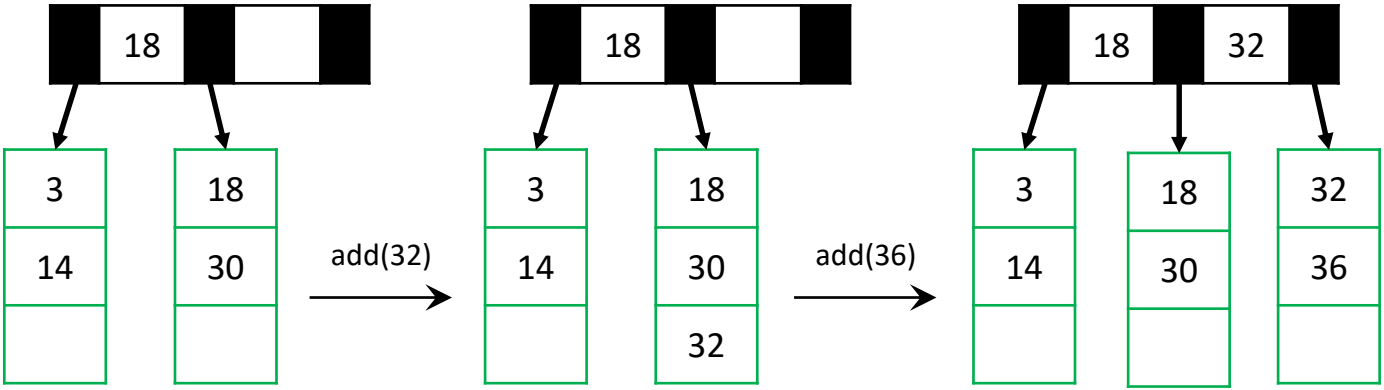
Add Example: Answer (2 of 7)



- When we "overflow" a leaf, it is split and the parent gains another key (to select between the two leaves)
- Parent's new key is the smallest element in the right child
- If there is no parent, create one

Add ~~3, 18, 14,~~ 30, 32, 36, 15, 16, 12, 40, 45, 38
M=3, L=3

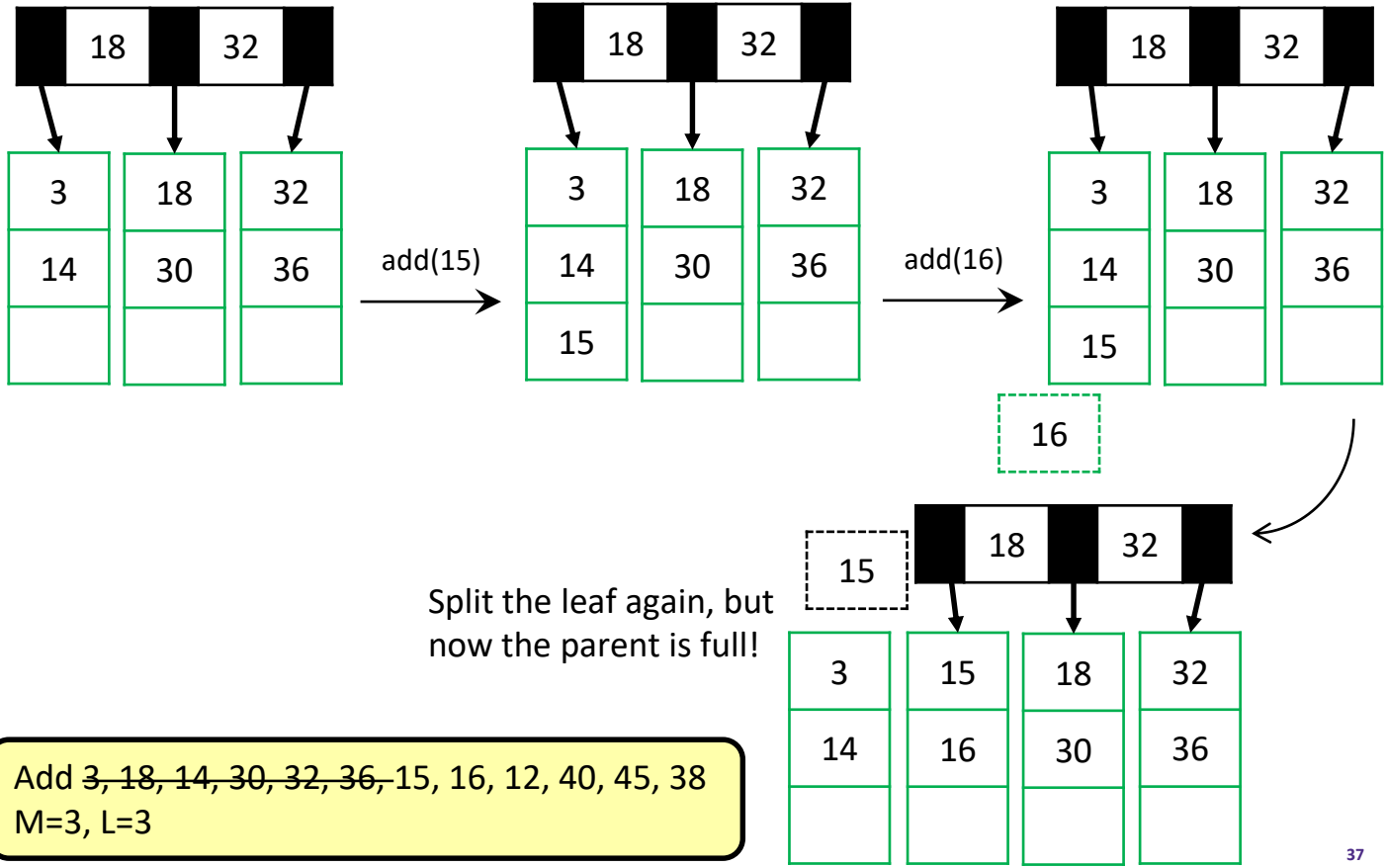
Add Example: Answer (3 of 7)



Split the leaf again

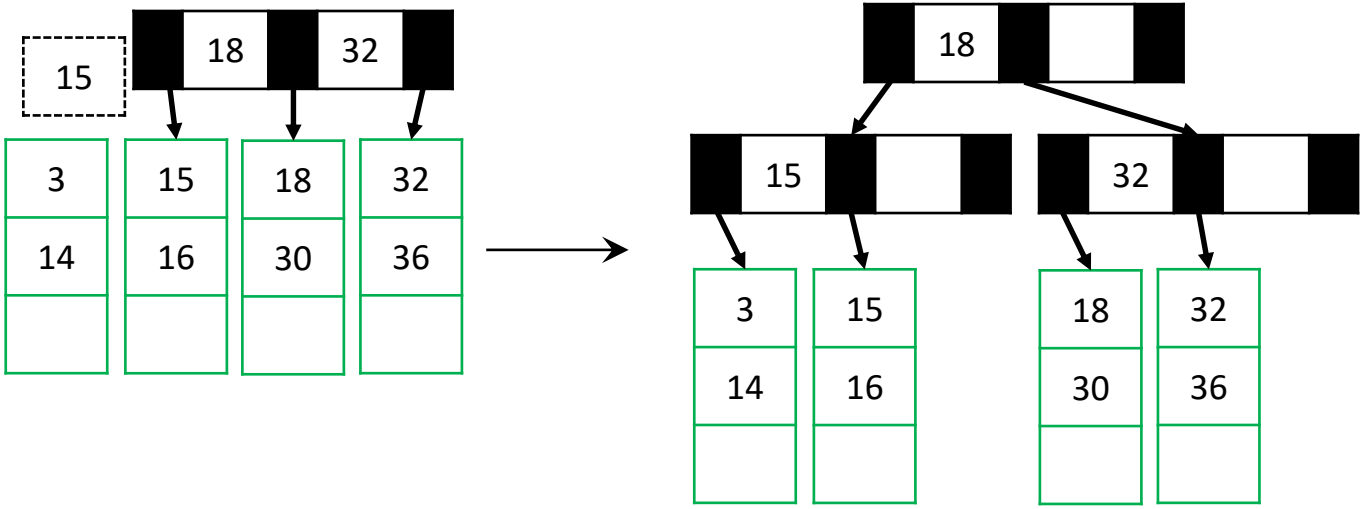
Add ~~3, 18, 14, 30,~~ 32, 36, 15, 16, 12, 40, 45, 38
M=3, L=3

Add Example: Answer (4 of 7)



Add ~~3, 18, 14, 30, 32, 36,~~ 15, 16, 12, 40, 45, 38
M=3, L=3

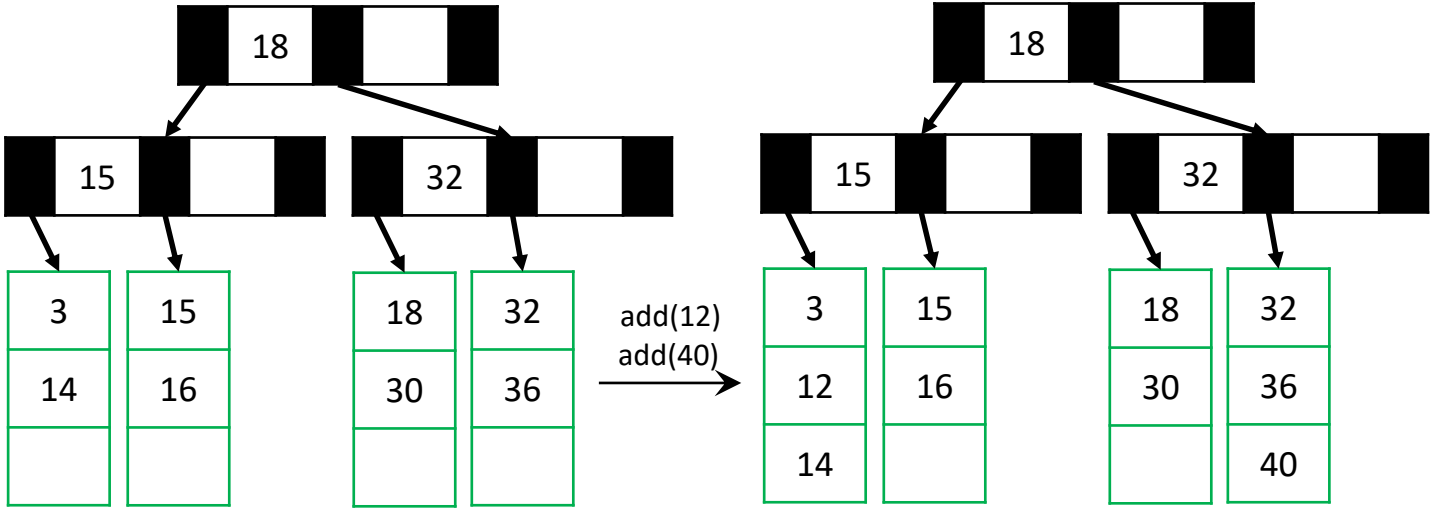
Add Example: Answer (5 of 7)



Split the parent (in this case, the root)

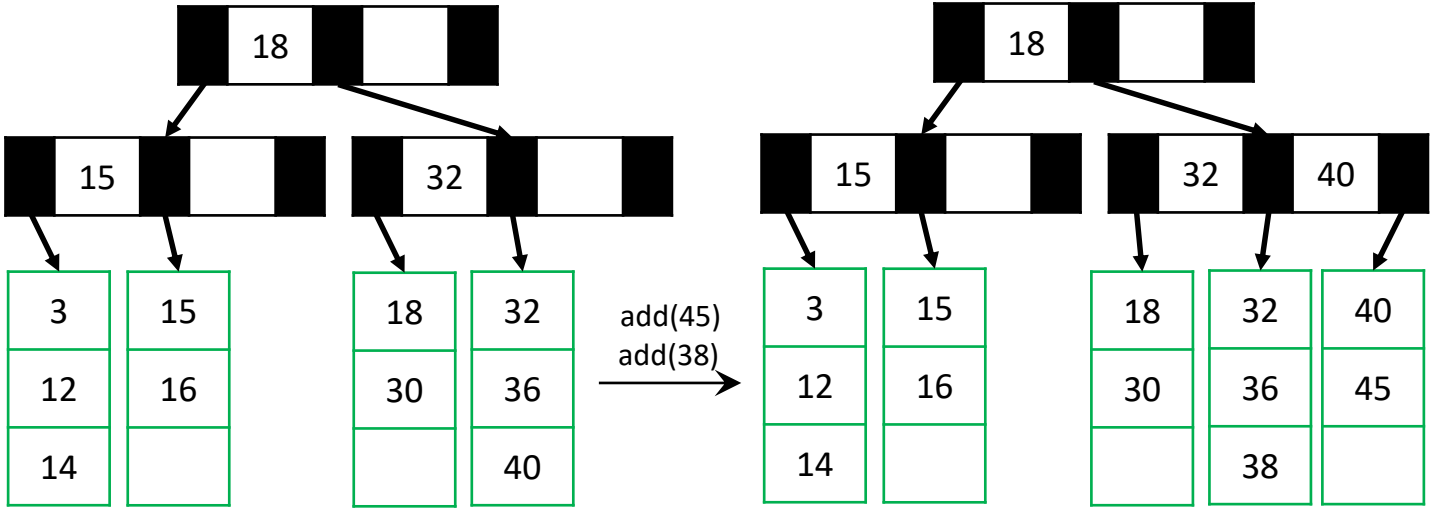
Add ~~3, 18, 14, 30, 32, 36, 15,~~ 16, 12, 40, 45, 38
M=3, L=3

Add Example: Answer (6 of 7)



Add ~~3, 18, 14, 30, 32, 36, 15, 16,~~ 12, 40, 45, 38
M=3, L=3

Add Example: Answer (7 of 7)



Split the leaf again

Add ~~3, 18, 14, 30, 32, 36, 15, 16, 12, 40,~~ 45, 38
M=3, L=3

B+ Tree Add Algorithm (1 of 2)

1. Add the value to its **leaf** in key-sorted order

2. If the **leaf** now has L+1 items, *overflow:*
   - Split the **leaf** into two leaves:
     - Original **leaf** with $\lceil (L+1)/2 \rceil$ smaller items
     - New **leaf** with $\lfloor (L+1)/2 \rfloor = \lceil L/2 \rceil$ larger items
   - Attach the new **leaf** to its parent
     - Add a new key (smallest key in new leaf) to parent in sorted order

3. If step (2) caused the parent to have M+1 children, …

B+ Tree Add Algorithm (2 of 2)

3. If step (2) caused an **internal node** to have M+1 children
   - Split the **internal node** into two nodes
     - Original **node** with $\lceil (M+1)/2 \rceil$ smaller keys
     - New **node** with $\lfloor (M+1)/2 \rfloor = \lceil M/2 \rceil$ larger keys
   - Attach the new **internal node** to its parent
     - Add a new key (smallest key in new node) to parent in sorted order
   - If step (3) caused the parent to have M+1 children, repeat step (3) on the parent
     - If the **root** overflows, make a new **root** with two children
     - This is the only case that increases the tree height

B+ Tree Add: Efficiency (1 of 2)

- Find correct leaf: $O(\log_2 M \log_M n)$
- Add (key, value) pair to leaf: $O(L)$
  - Why? shift items to maintain sorted order
- Possibly split leaf: $O(L)$
  - Why? Copy $L/2$ items to new leaf
- Possibly split parents all the way up to root: $O(M \log_M n)$
  - Why? Each split: copy $M/2$ keys
    Number splits: $\log_M n$ height
- Total: $O(L + M \log_M n)$

B+ Tree Add: Efficiency (2 of 2)

- Worst-case runtime is $O(L + M \log_M n)$!

- But the worst-case isn't that common!
  - Splits are uncommon
    - Only required when a node is *full*
    - M and L are likely to be large and, after a split, nodes will be half empty
  - Splitting the **root** is extremely rare
  - Remember that our goal is minimizing disk accesses! Disk accesses are still bound by $O(\log_M n)$