AVL Height Proof; B+ Trees

CSE 332 Summer 2021

Instructor: Kristofer Wong

Teaching Assistants:

Alena Dickmann	Arya GJ	Finn Johnson
Joon Chong	Kimi Locke	Peyton Rapo
Rahul Misal	Winston Jodjana	

Announcements

- P2 has been released! Please take a look at the spec online
- Repos for P2 created this morning!
- Grading is coming along – we're working on this as fast as we can ☺
- Sorry for actual lecture delay! I'll get it uploaded as soon as I possibly can.

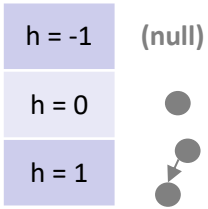
Lecture Outline

- **AVL Tree Height Bound**

- Memory Hierarchy Basics
  - What is the Memory Hierarchy?
  - How does it impact data structure design?

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

Height of an AVL Tree? (1 of 2)

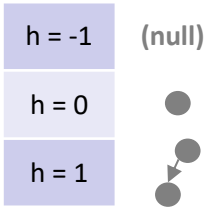


- The "best case" AVL tree is a perfect tree

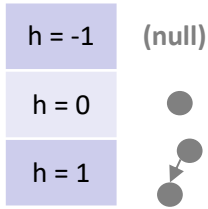
- What does the "worst case" AVL tree look like?
  - i.e. the tree with the fewest nodes

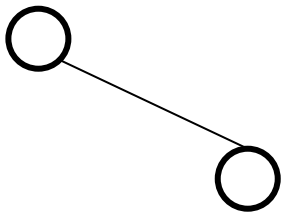
- Let `S(h)` = minimum # of nodes in an AVL tree of height *h*
  - And also `S(-1) = 0, S(0) = 1`
  - … so what is the expression for `S(h)`?

Minimal AVL Tree (height = 0)

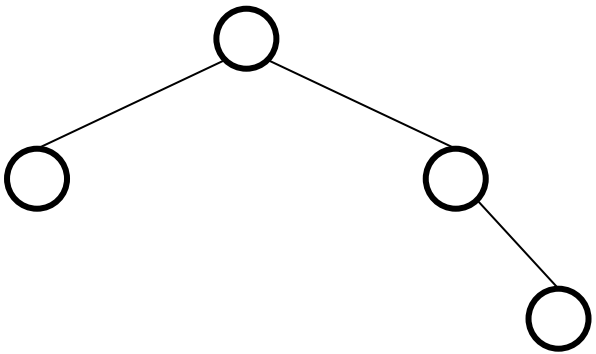


Minimal AVL Tree (height = 1)

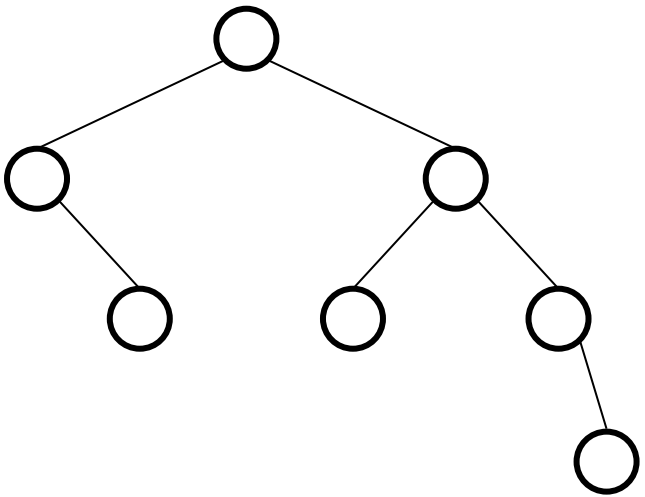




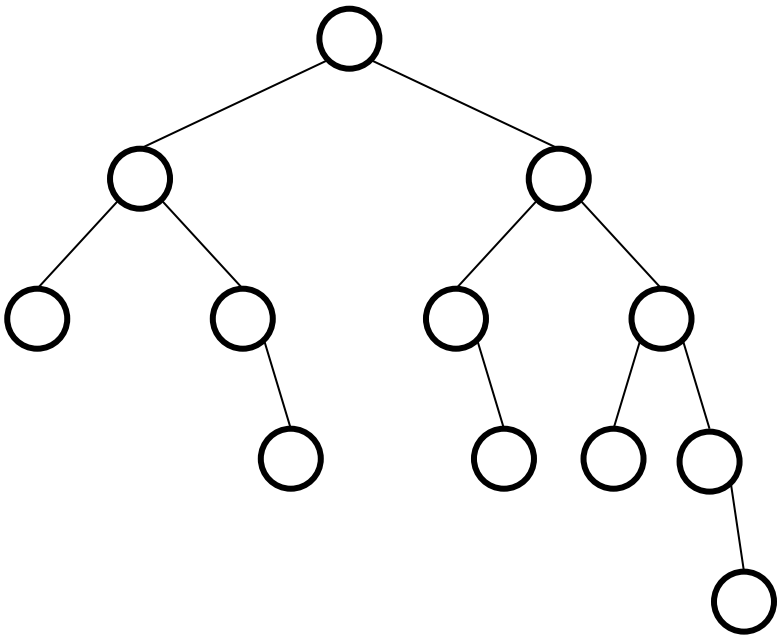
Minimal AVL Tree (height = 2)



Minimal AVL Tree (height = 3)



Minimal AVL Tree (height = 4)



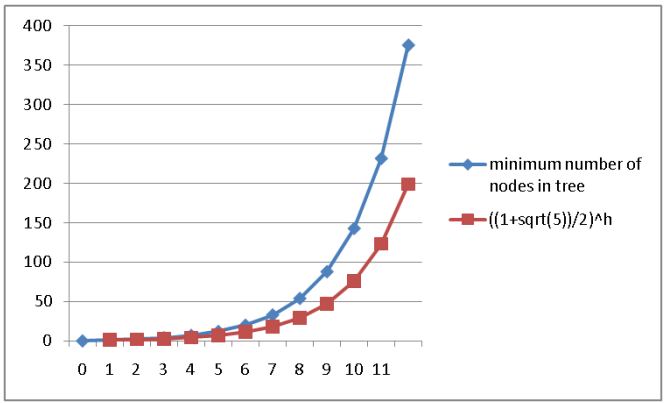
Height of an AVL Tree? (2 of 2)

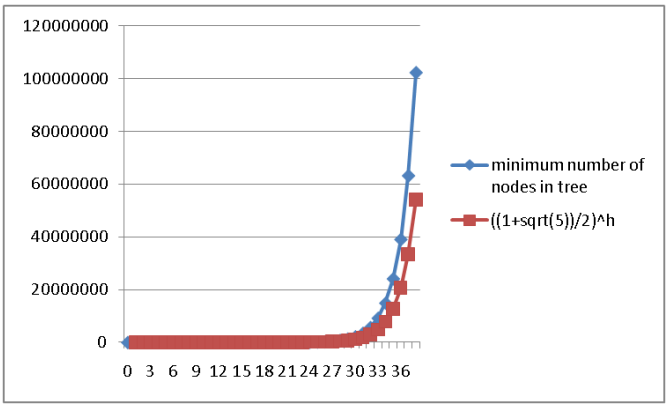
- Let $S(h)$ = minimum # of nodes in an AVL tree of height h
  - And also $S(-1) = 0$, $S(0) = 1$
  - … what is the expression for $S(h)$?
  - $S(h) = S(h-1) + S(h-2) + 1$

- Solution of Recurrence: $S(h) \approx 1.62^h$

Getting an intuition from graphs..

- Good intuition from plots comparing:
  1. $S(h)$ computed directly from the definition
  2. $((1+\sqrt{5})/2)^h \approx 1.62^h$
- $S(h)$ is always bigger, up to trees with huge # of nodes
  - Graphs aren't proofs, so let's prove it



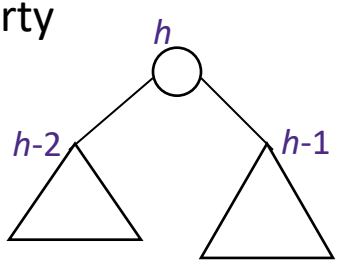


The Proof Outline

Let $S(h)$ = the min # of nodes in an AVL tree of height h

- If we can prove that $S(h)$ grows exponentially in h, then a tree with n nodes has a logarithmic height

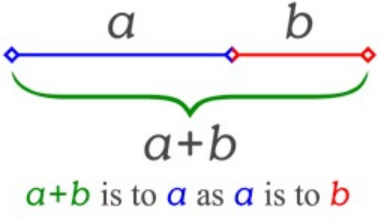
- Step 1: Define $S(h)$ inductively using AVL property
  - $S(-1)=0,\ S(0)=1,\ S(1)=2$
  - $S(h) = 1 + S(h-1) + S(h-2)$ for $h \geq 1$



- Step 2: Show this recurrence grows really fast
  - Similar to Fibonacci numbers
  - Can prove for all h, $S(h) > \phi^h - 1$ where ϕ is the golden ratio, $(1+\sqrt{5})/2 \approx 1.62$
  - Growing faster than 1.62^h is "plenty exponential"

Interlude: The Golden Ratio

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.62$$



This is a special number

- Aside: Since the Renaissance, many artists and architects have proportioned their work (e.g., length:height) to approximate the *golden ratio*: If `(a+b)/a = a/b`, then `a = ϕb`

- We will need one special arithmetic fact about ϕ :

$$\begin{aligned}
\phi^2 &= ((1+5^{1/2})/2)^2 \\
&= (1 + 2*5^{1/2} + 5)/4 \\
&= (6 + 2*5^{1/2})/4 \\
&= (3 + 5^{1/2})/2 \\
&= 1 + (1 + 5^{1/2})/2 \\
&= 1 + \phi
\end{aligned}$$

The Proof (1 of 2)

$S(-1)=0, \quad S(0)=1, \quad S(1)=2$
$S(h)=1 + S(h-1) + S(h-2) \quad \text{for } h \geq 1$

Theorem: For all $h \geq 0$, $S(h) > \phi^h - 1$

Proof: By induction on h

Base cases:

$S(0) = 1 > \phi^0 - 1 = 0$

$S(1) = 2 > \phi^1 - 1 \approx 0.62$

The Proof (2 of 2)

$S(-1)=0, \quad S(0)=1, \quad S(1)=2$
$S(h)=1 + S(h-1) + S(h-2) \quad \text{for } h \geq 1$

Theorem: For all $h \geq 0$, $S(h) > \phi^h - 1$

Proof: By induction on h

Inductive case ($k > 1$):

Show that $S(k+1) > \phi^{k+1}-1$, assuming $S(k) > \phi^k-1$ and $S(k-1) > \phi^{k-1} - 1$

$S(k+1)$	$= 1 + S(k) + S(k-1)$	*by definition of S*
	$> 1 + (\phi^k - 1) + (\phi^{k-1} - 1)$	*by induction*
	$= \phi^k + \phi^{k-1} - 1$	*by arithmetic (1-1=0)*
	$= \phi^{k-1}(\phi + 1) - 1$	*by arithmetic (factor ϕ^{k-1})*
	$= \phi^{k-1}\phi^2 - 1$	*by special property of ϕ*
	$= \phi^{k+1} - 1$	*by arithmetic (add exponents)*

Lecture Outline

- AVL Tree Height Bound
- Memory Hierarchy Basics
  - **What is the Memory Hierarchy?**
  - How does it impact data structure design?
- B-Trees
  - Goals and Design
  - B+ Tree Structure
  - B+ Tree Implementation: Find
  - B+ Tree Implementation: Add

gradescope

gradescope.com/courses/275833

- Suppose we have 100,000,000 items. What is the maximum height of:
  - A perfectly-balanced BST?
  - A perfectly-balanced octonary search tree?
    - Like a BST, but with <= 8 children instead of 2
  - An AVL tree?

Let's talk about space!

- We have a simple and elegant data structure for the Dictionary ADT: the Binary Search Tree
  - But its worst-case behavior isn't great

- We can guarantee worst-case $O(\log n)$ with an AVL tree
  - At the cost of increased implementation complexity and space
  - One of several interesting/fantastic balanced-tree approaches!

- We will learn another balanced-tree approach: B-trees
  - It performs really well on large dictionaries (>1GB = 2^{30} bytes)
  - But to understand why, we need some ***memory-hierarchy basics***

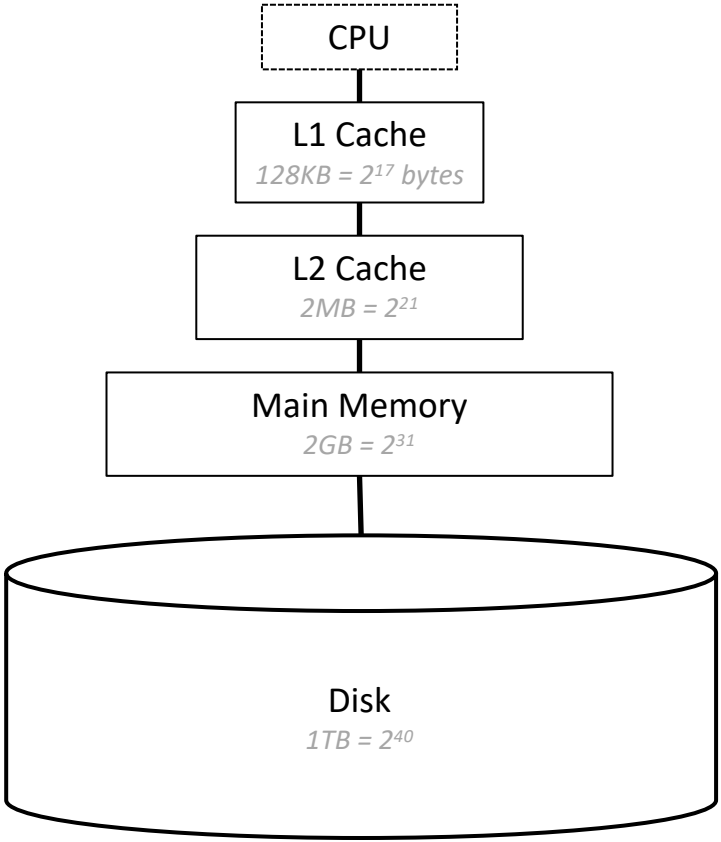
Why Does the Memory Hierarchy Matter?

- We said "every memory access has an unimportant *O(1)* cost"
  - Learn more in CSE 351/333/471

```
// Requires arr to be sorted
// Returns whether k is in array
boolean findSorted(int[] arr, int k) {
  for(int i=0; i < arr.length; ++i) {
    if(arr[i] == k)
      return true;
    else if(arr[i] > k)
      return false;
  }
  return false;
}
```

We claimed these two operations were approximately equal!

A Typical Real-World 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 HDD: 2^{7}/sec = 8,000,000 instructions

(immaterial difference with SSD)

Said In Another Way …

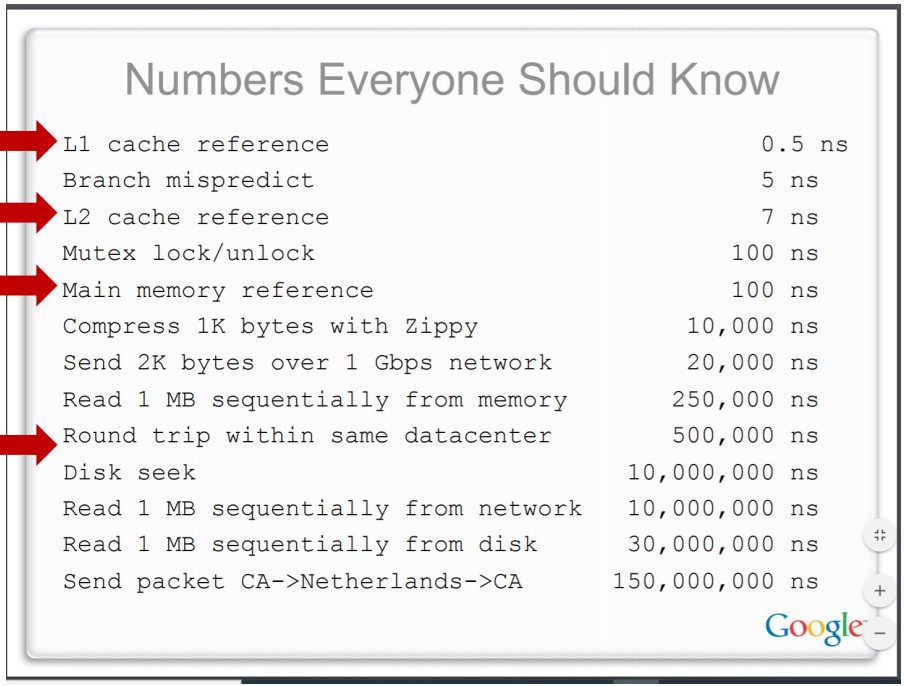
- Jeff Dean's "Numbers Everyone Should Know" (LADIS '09)
- Consider: I want to eat an avocado

I have an avocado, so I cut it and eat it

My roommate gives me their avocado

I go to the store for an avocado

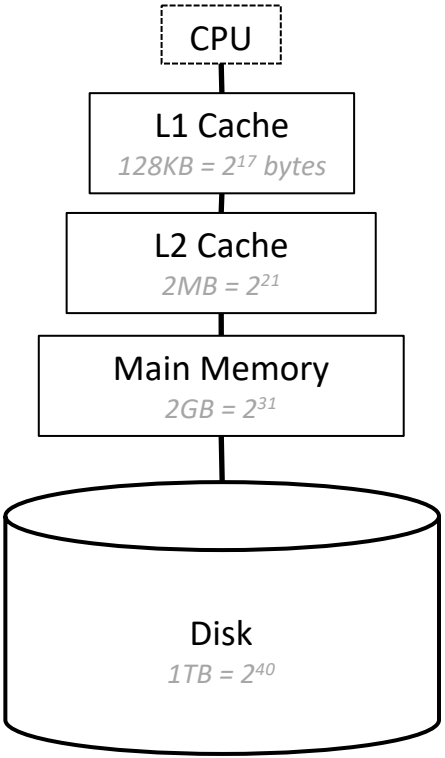
I move to Mexico, plant my own avocado tree, and it starts producing fruit in 30 years



Hardware and OS Support (1 of 2)

- The hardware and OS work together to automatically move data into and out of successive levels for you!
  - Replaces items currently in memory/L2/L1
  - Data structures and algorithms are faster if "fits in cache"

- Terminology:
  - Data moved from **disk** into **memory** is in "block" or "page" size
  - Data moved from **memory** into L1/L2 **cache** is in cache "line" size



Hardware and OS Support (2 of 2)

- Terminology:
  - Data moved from **disk** into **memory** is in "block" or "page" size
  - Data moved from **memory** into L1/L2 **cache** is in cache "line" size

- Neither movement nor sizes are under programmer control!

- 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 design memory-aware software, 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 data has moved up the hierarchy, keep it around
  - A *particular piece of data* is more likely to be accessed again in the near future than some random other piece of data

Locality Principles, in Detail

- **Spatial Locality** (locality in space)
  - If an address is referenced, ***addresses that are close by*** tend to be referenced soon

- **Temporal Locality** (locality in time)
  - If an address is referenced, ***that same address*** tends to be referenced again soon

Lecture Outline

- AVL Tree Height Bound
- Memory Hierarchy Basics
  - What is the Memory Hierarchy?
  - **How does it impact data structure design?**
- B+ Trees
  - Goals and Design
  - B+ Tree Structure
  - B+ Tree Implementation: Find
  - B+ Tree Implementation: Add

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

- Which has the potential to take advantage of **spatial locality**? Array vs Linked List?
  - As a simplification, assume each object allocated via Java's uses contiguous space

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 just for keys), isn't this ok?
- Big-Oh is a good start, but # disk accesses still matters:
  - Pretend those 2^{39} nodes were in an AVL tree of height 55
  - Most of the nodes will be on disk
    - Tree is shallow, but it is still many gigabytes big
    - 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

- AVL Tree Height Bound
- Memory Hierarchy Basics
  - What is the Memory Hierarchy?
  - How does it impact data structure design?
- B+ Trees
  - **Goals and Design**
  - B+ Tree Structure
  - B+ Tree Implementation: Find
  - B+ Tree Implementation: Add

Goal of the B+ Tree

- **Problem**: A dictionary with so many items *most of it is on disk*
- **Goal**: A balanced tree (logarithmic height) that minimizes disk accesses
- **Concept**: Increase the branching factor of our tree
  - Minimize number of nodes to traverse
- **Disclaimer:** You will not have to implement this structure!!
  - Requires more control over memory than Java allows

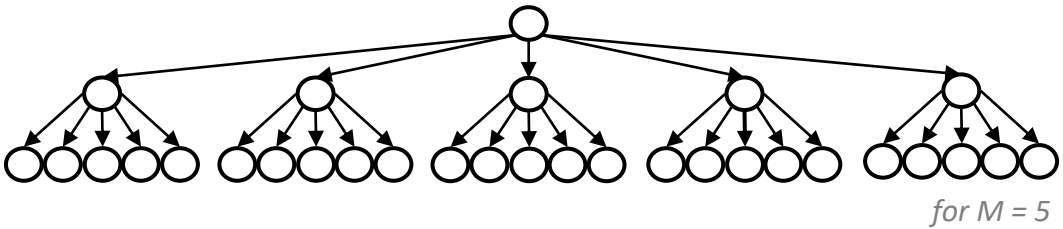
How do we minimize disk accesses?

- Increase size of each node in our tree
  - … to the size of a full disk block
  - For a dictionary, this would mean many key/value pairs and pointers per node

- Worst case number of nodes that we look at for a find in any tree will be bounded by height
  - So let's try to maximize how efficiently we use the height
  - Higher branching factor than 2

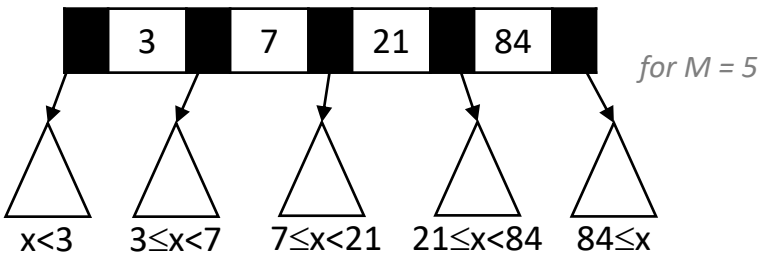
Increasing efficiency: M-ary Search Tree

- A search tree with branching factor M (instead of 2)
  - Each node has a key-sorted array of M children: `Node[]`



for M = 5

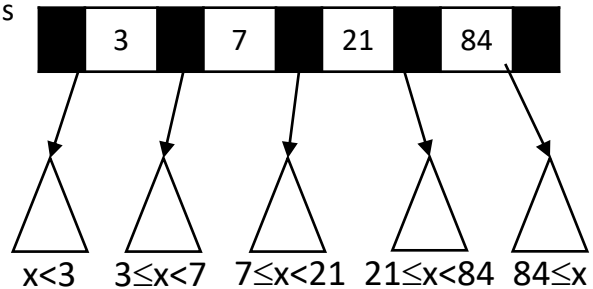
  - Ordering property similar to BST



  - M-1 keys define the M subtrees (ie, ranges) that we search through
- Choose M such that the node size = disk block size

M-ary Performance?

- Runtime for `find` = NumHops * WorkPerHop
  - **Balanced** tree height is: $\log_M n$ (M-ary) vs $\log_2 n$ (binary)
    - Eg: M = 256 (=2^8) and n = 2^{40}, M-ary makes 5 hops vs binary makes 40 hops
  - For each internal node, how to decide which child to take?
    - Binary: Less than vs greater than node's single 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



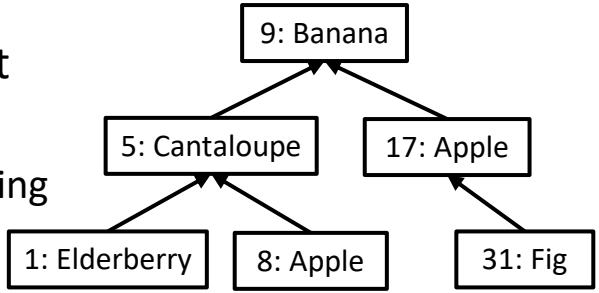
- Runtime for M-ary `find`:
  - $O(\log_2 M \log_M n)$

- Note: a "hop" here means following a pointer to another node

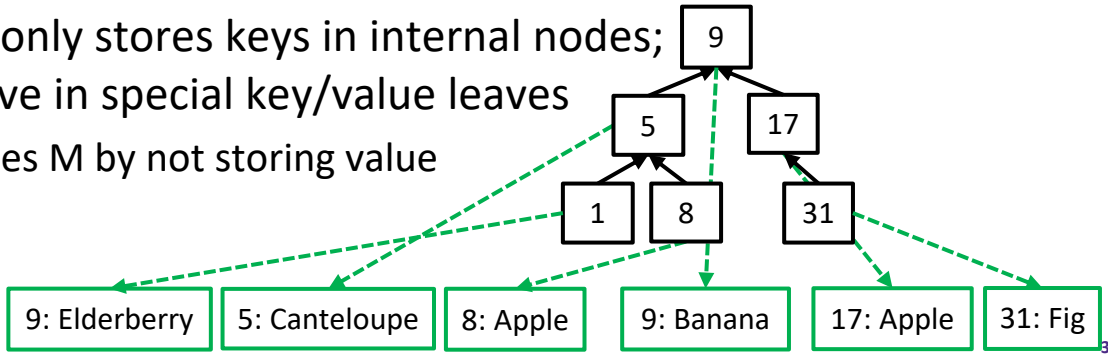
Design Decision: Key-only Internal Nodes

- A Dictionary ADT stores key->value pairs; where should we store a key's value?

- BST stores value alongside the key at every node
  - Loads entire node even if we are "passing through" to find a different key



- B+ Tree only stores keys in internal nodes; values live in special key/value leaves
  - Increases M by not storing value



Lecture Outline

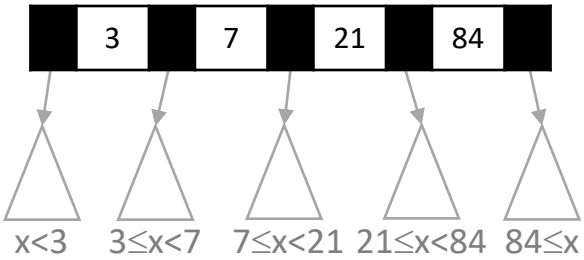
- AVL Tree Height Bound
- Memory Hierarchy Basics
  - What is the Memory Hierarchy?
  - How does it impact data structure design?
- B-Trees
  - Goals and Design
  - **B+ Tree Structure**
  - B+ Tree Implementation: Find
  - B+ Tree Implementation: Add

B+ Tree Node Structure

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

- Two node types: **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"

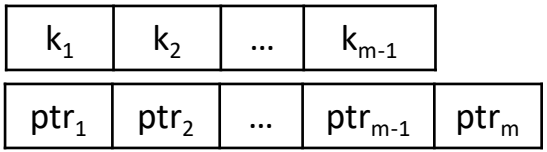


- Each **leaf node** contains up to *L* items
  - Stores (key, value) pairs
  - As usual, we'll ignore the "along for the ride" value in our examples

3	"cat"
7	"apple"
21	"purple"
84	"ideas"

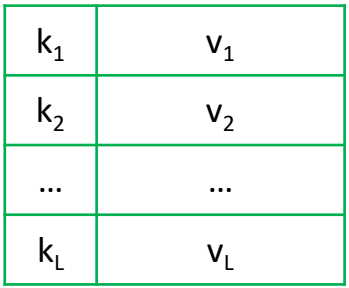
B+ Tree Parameters

- Two parameters, one for each type of node:
  - ***M*** = # of children in an **internal** node
    - The ranges are defined by M-1 keys
  - ***L*** = # of items in a **leaf** node



(sorted by key)

- Picking *M* and *L* based on disk-block size maximizes B+ Tree's efficiency
  - Recommend M* ≈ diskBlockSize/keySize
  - Recommend L = diskBlockSize/(keySize + valueSize)
  - In practice, ***M*** ≫ ***L***
    - Since typically sizeof(key) ≫ sizeof(value)



(sorted by key)

** 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**
    - Unusual; only occurs when starting up
  - 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

B+ Trees are Shallower than AVL Trees

- 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)

B+ Trees are Disk Friendly (1 of 2)

- Reduces number of disk accesses during `find`
  - Large *M* = shallower tree = potentially fewer accesses
  - Requires that *we pick M wisely*
    - Too large: multiple disk accesses to load a single **internal** node
    - Too small: tree could've been shallower
  - Binary search over *M*-1 keys insignificant compared to disk access

- Reduces unnecessary data transferred from disk
  - `find` wants *one value*; doesn't load "incorrect" values into memory
  - Only one disk access to bring (the single correct) value into memory: when we find the correct **leaf node**

B+ Trees are Disk Friendly (2 of 2)

- Maximizes temporal locality
  - Since typically sizeof(key) ≫ sizeof(value), can hold significantly more B+ Tree-style **internal** nodes in memory than BST-style nodes
  - B+ Tree-style **internal** nodes are used more often (they differentiate between a larger fraction of keys) than BST-style nodes, and therefore are more likely to be held in memory by the OS

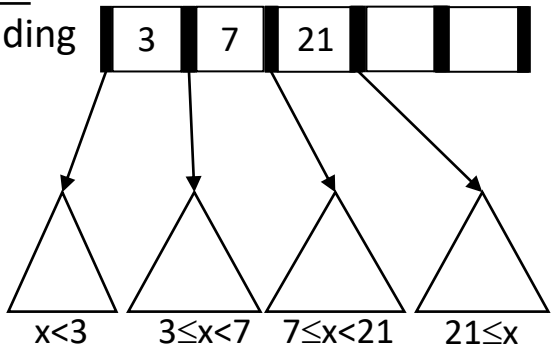
Lecture Outline

- AVL Tree Height Bound
- Memory Hierarchy Basics
  - What is the Memory Hierarchy?
  - How does it impact 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 determine which branch to take
  - At the **leaf** node, do binary search on the (up to) *L* items
  - *Requires that keys are sorted in both **internal** and **leaf** nodes!*

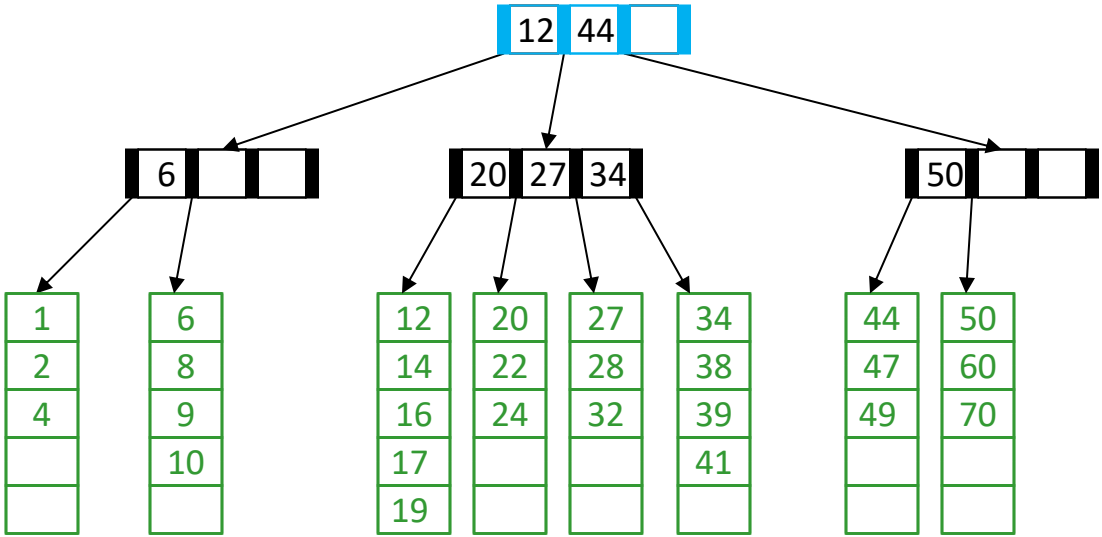
- Difference:
  - Since we *don't store value at internal nodes*, there is no "best case" of finding our value at the root node; must always traverse to the bottom of B+ Tree



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)



Lecture Outline

- AVL Tree Height Bound
- Memory Hierarchy Basics
  - What is the Memory Hierarchy?
  - How does it impact data structure design?
- B+ Trees
  - Goals and Design
  - B+ Tree Structure
  - B+ Tree Implementation: Find
  - **B+ Tree Implementation: Add**

B+ Tree Add Algorithm (1 of 3)

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 $\lceil L/2 \rceil$ smaller items
     - New **leaf** with $\lfloor L/2 \rfloor = \lceil L/2 \rceil$ larger items
   - Attach the new **leaf** to its parent
     - Add a new key (smallest key in new leaf) to parent in sorted order

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

B+ Tree Add Algorithm (2 of 3)

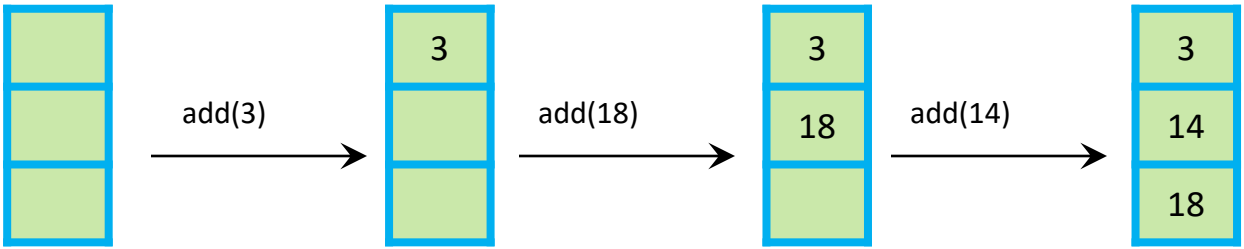
3. If step (2) caused an **internal node** to have M+1 children
   - Split the **internal node** into two nodes
     - Original **node** with $\lceil (M+1)/2 \rceil$ smaller keys
     - New **node** with $\lfloor (M+1)/2 \rfloor = \lceil M/2 \rceil$ larger keys
   - Attach the new **internal node** to its parent
     - Move the median 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

4. If step (3) caused the **root** to have M+1 children
   - Split the old root into two **internal nodes**, then add them to a newly-created **root** as described in step (3)
   - *This is the only case that increases the tree height!*

Add Example:

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

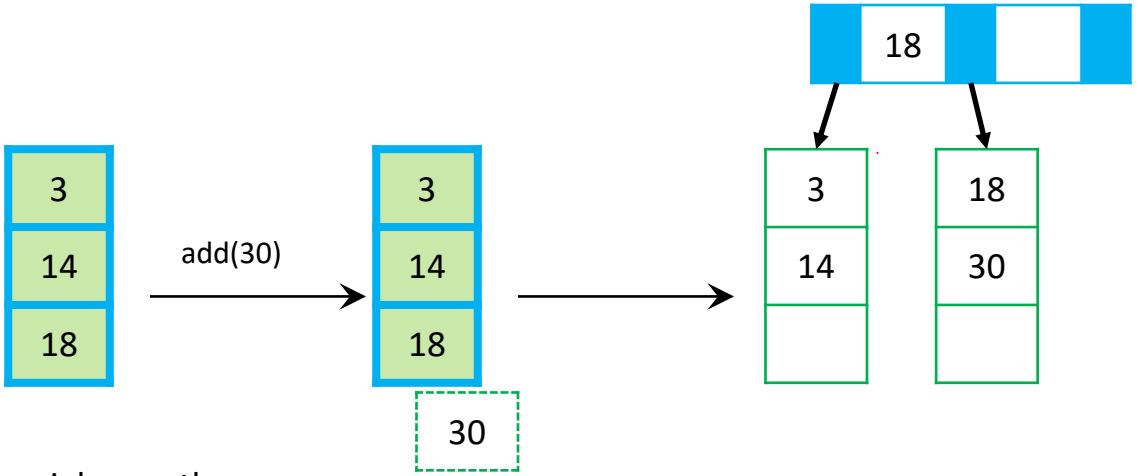
Add Example: Answer (1 of 7)



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

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

Add Example: Answer (2 of 7)

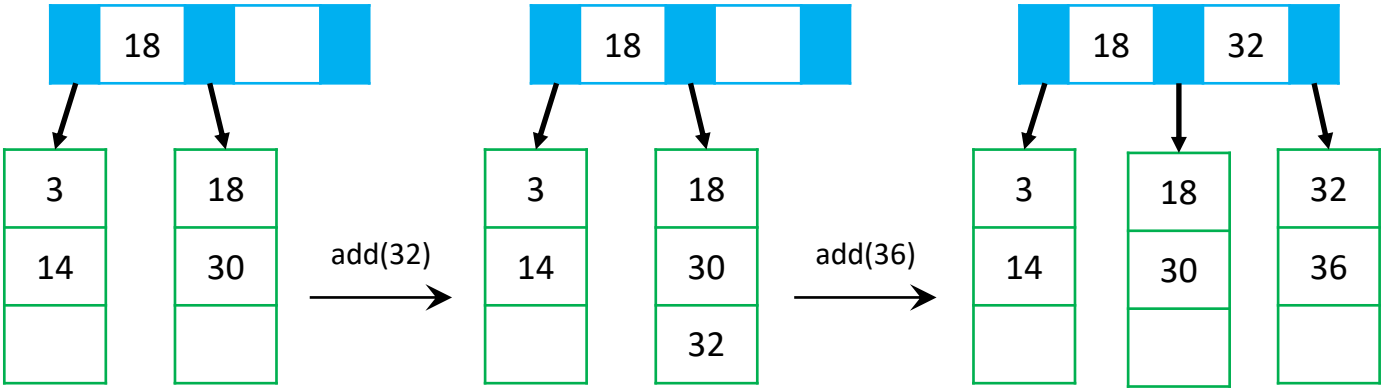


Special case: the **root** is a **leaf node**

- 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

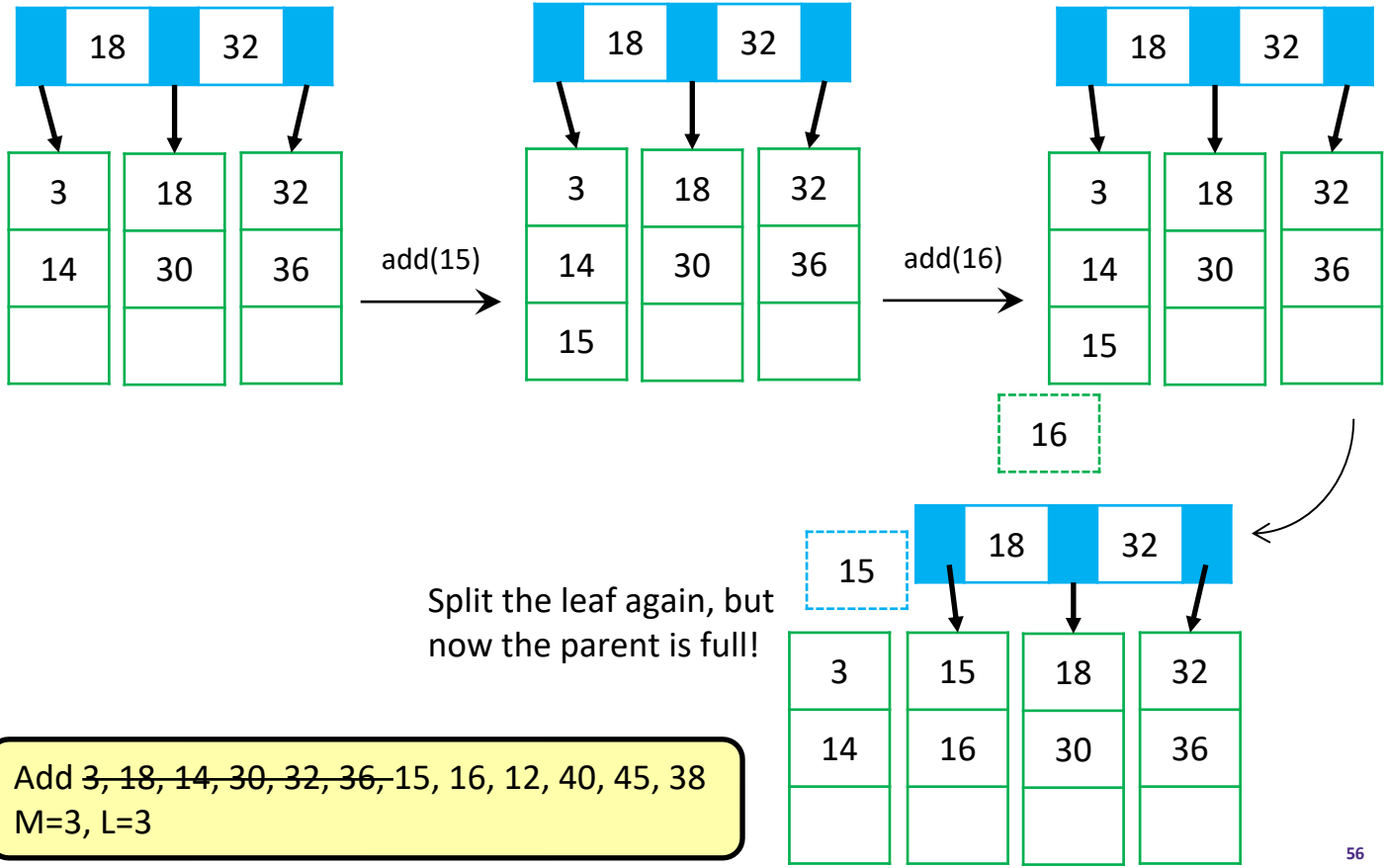
Add Example: Answer (3 of 7)



Split the leaf again

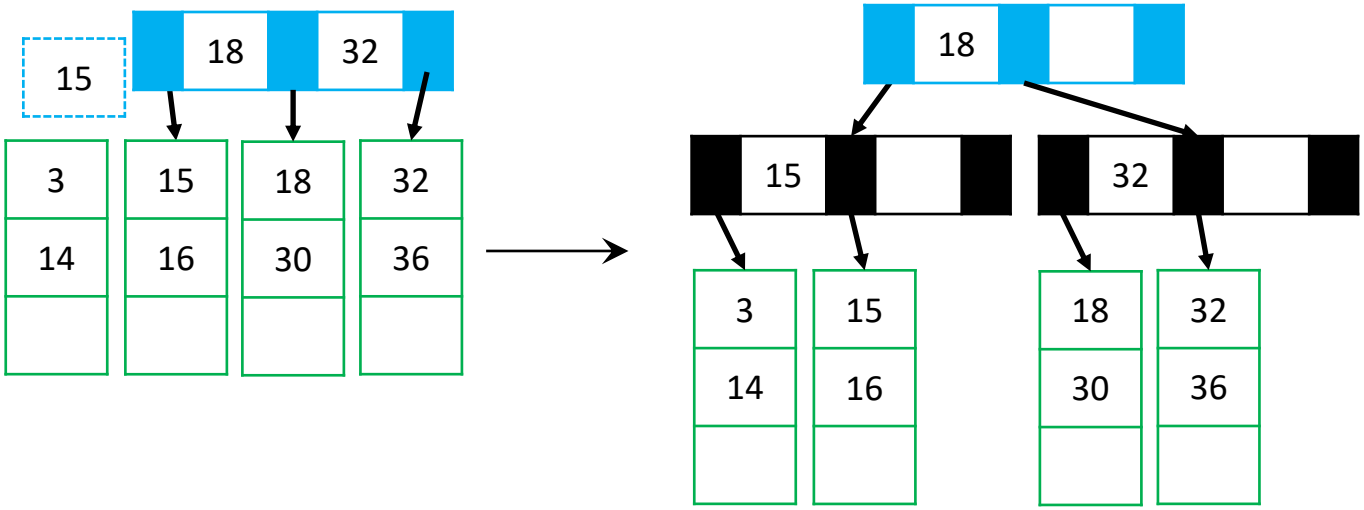
Add ~~3, 18, 14, 30,~~ 32, 36, 15, 16, 12, 40, 45, 38
M=3, L=3

Add Example: Answer (4 of 7)



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

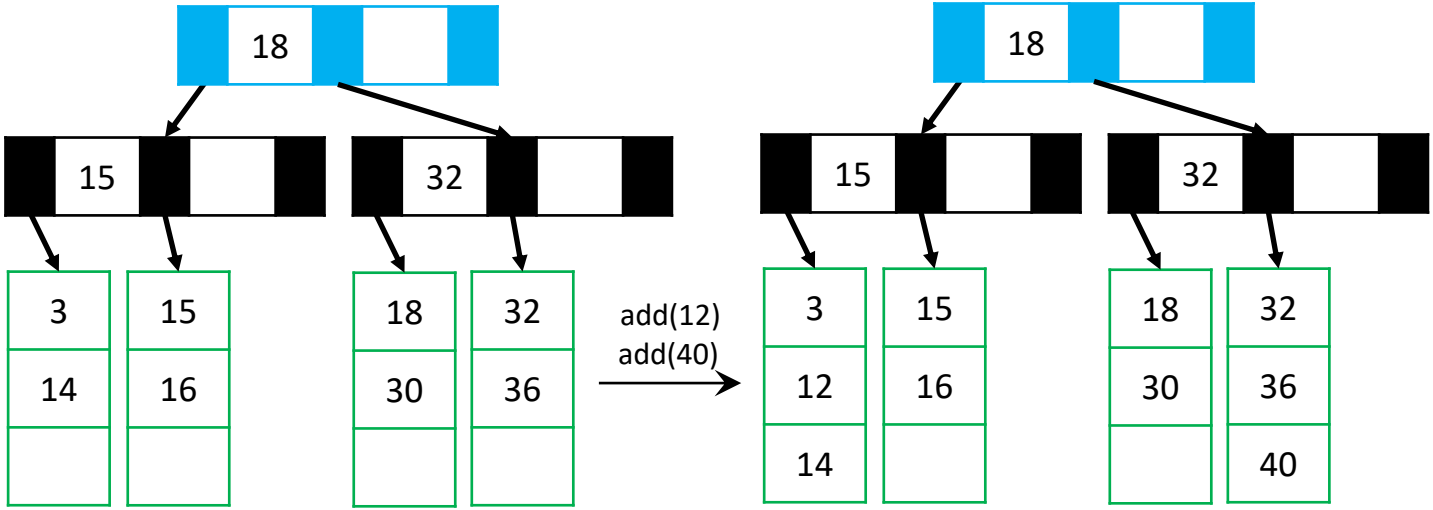
Add Example: Answer (5 of 7)



Split the parent (in this case, the root). Note that the median key **moves** into the parent (vs being copied)

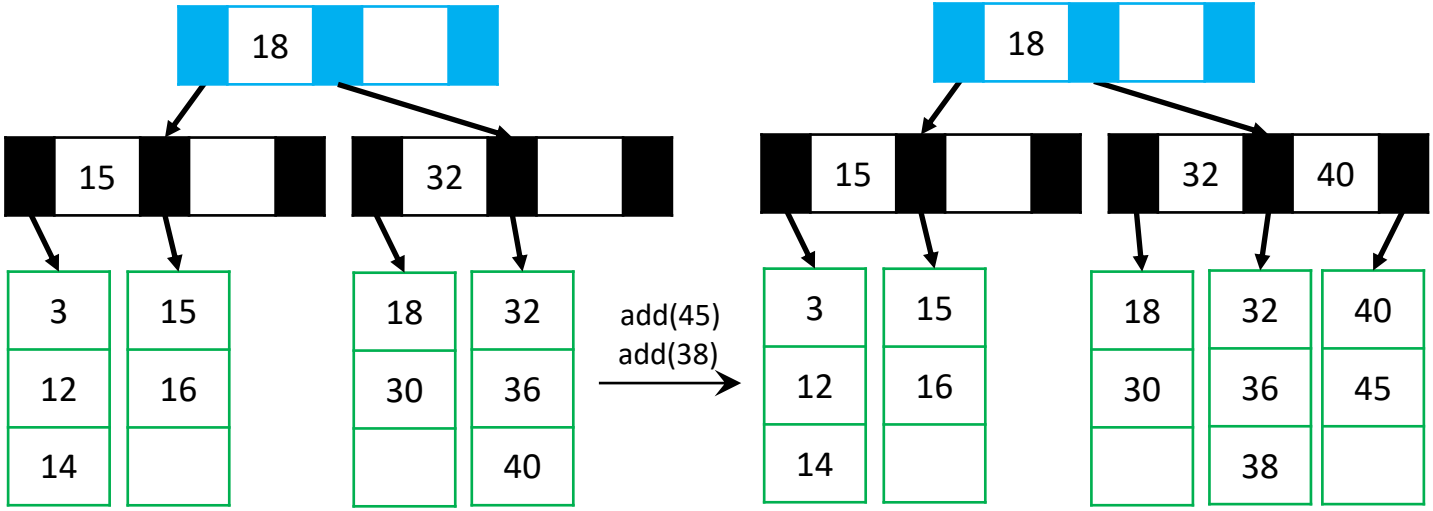
Add ~~3, 18, 14, 30, 32, 36, 15,~~ 16, 12, 40, 45, 38
M=3, L=3

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 (3 of 3)

- Note the similarities between the overflow steps:

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 the parent in sorted order

Split the **internal node** into two leaves:
- Original **node** with $\lceil (M+1)/2 \rceil$ smaller items
- New **node** with $\lfloor (M+1)/2 \rfloor = \lceil M/2 \rceil$ larger items

Attach the new **internal node** to its parent
- Move the median key (smallest key in new **node**) to the parent in sorted order

- But also the difference when overflowing a root:

Split the **root** into two **internal nodes**:
- Left **node** with $\lceil (M+1)/2 \rceil$ smaller items
- Right **node** with $\lfloor (M+1)/2 \rfloor = \lceil M/2 \rceil$ larger items

Attach the **internal nodes** to the new **root**
- Move the median key (smallest key in new right **node**) to the **root**

gradescope

gradescope.com/courses/275833

- When splitting nodes in a B+ Tree, why do we need to *copy* keys out of leaves but *move* keys out of internal nodes?

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?
- Possibly split **leaf**: $O(L)$
  - Why?
- Possibly split parents all the way up to **root**: $O(M \log_M n)$
  - Why?

- 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 $O(\log_M n)$