Memory Hierarchy; B-Trees

CSE 332 Spring 2021

Instructor: Hannah C. Tang

Teaching Assistants:

Aayushi Modi	Khushi Chaudhari	Patrick Murphy
Aashna Sheth	Kris Wong	Richard Jiang
Frederick Huyan	Logan Milandin	Winston Jodjana
Hamsa Shankar	Nachiket Karmarkar	

gradescope

gradescope.com/courses/256241

- 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?

Announcements

- P2 has been released!
- Project and quiz deadlines will never overlap again <3
- Expected turnaround time for quiz and project grading: ~1.5w

Lecture Outline

- 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

And Now for Something Completely Different…

- 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 $O(\texttt{log}\ n)$ with an AVL tree
  - … but 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 (eg >1GB = 2^{30} bytes)
  - But to understand why, we need some ***memory-hierarchy basics***

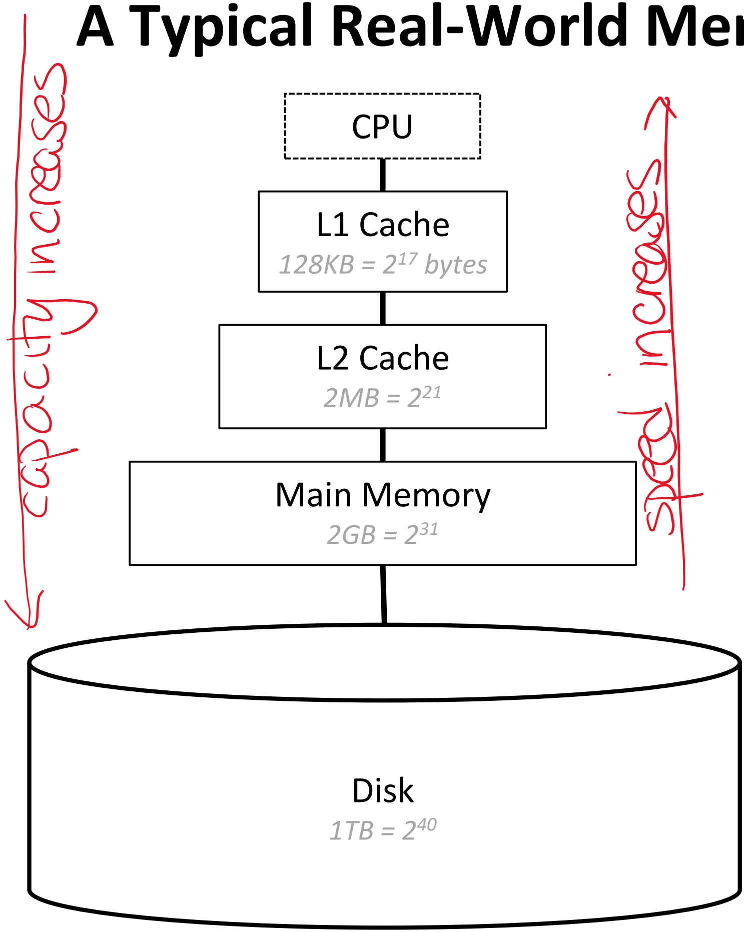
Why Does the Memory Hierarchy Matter?

- We said "every memory access has an unimportant *O(1)* cost"
  - Learn more in CSE 351/333/471
  - Focus here is on relevance to data structures and efficiency

```
// 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)

Numbers Everyone Should Know

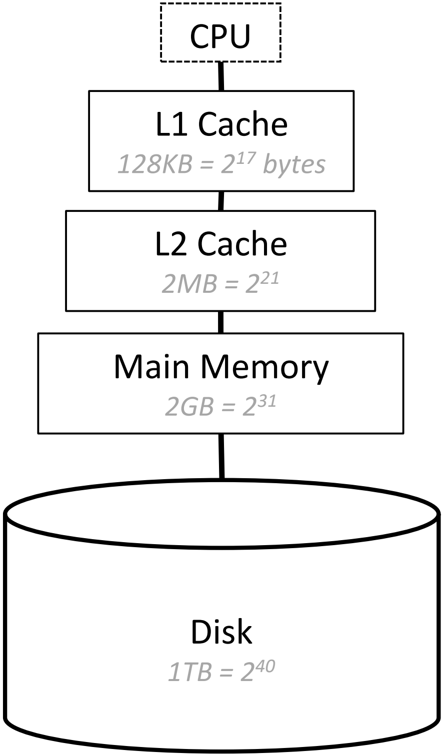
Sticky note on monitor	L1 cache reference	0.5 ns
	Branch mispredict	5 ns
Yelling for your roommate	L2 cache reference	7 ns
	Mutex lock/unlock	100 ns
Flipping through textbook	Main memory reference	100 ns
	Compress 1K bytes with Zippy	10,000 ns
	Send 2K bytes over 1 Gbps network	20,000 ns
	Read 1 MB sequentially from memory	250,000 ns
	Round trip within same datacenter	500,000 ns
Retaking 311 and then retaking 332 😱	Disk seek	10,000,000 ns
	Read 1 MB sequentially from network	10,000,000 ns
	Read 1 MB sequentially from disk	30,000,000 ns
	Send packet CA->Netherlands->CA	150,000,000 ns

Google

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

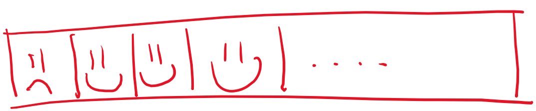
- Memory Hierarchy Basics
  - What is the Memory Hierarchy?
  - **How does it impact data structure design?**

- 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)

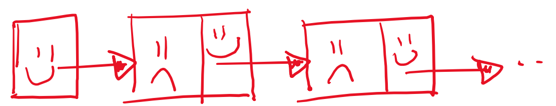
- Which has the potential to take advantage of **spatial locality**? Array vs Linked List?
  - As a simplification, assume each object allocated via Java's uses contiguous space

Node[] arr = new Node[100]



Node head = new Node
head.next = new Node
head.next.next = new Node

class Node {
  Node next
  Value v.
}



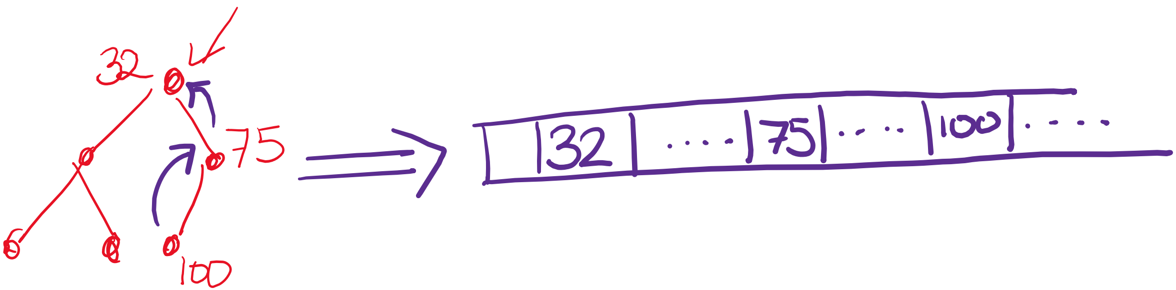
Spatial Locality: Arrays vs. Linked Lists (2 of 3)

- 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 $O(\texttt{log}\ n)$
  - 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

- 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

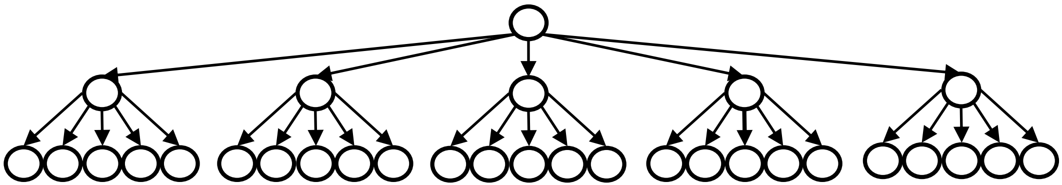
Our B-Tree Goal

- **Problem**: A dictionary with so many items *most of it is on disk*
- **Goal**: A balanced tree (logarithmic height) that minimizes disk accesses
- Let's look at two design decisions that'll get us there
- **Let's An idea**: Increase the branching factor of our tree
  - Each node

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

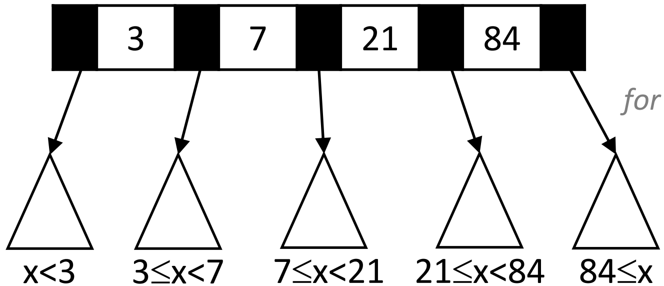
Decision #1: M-ary Search Tree

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



for M = 5

  - M-1 keys define the M subtrees (ie, ranges) that we search through

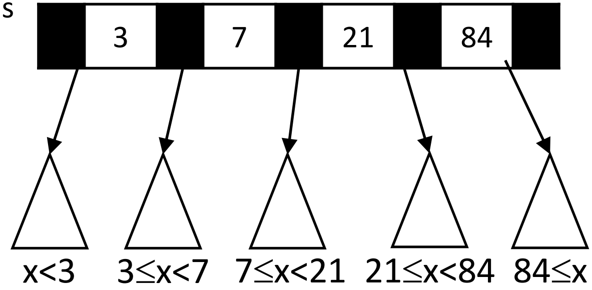


for M = 5

  - Choose M to fit into a disk block: only 1 disk access for entire array!

Decision #1: 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`:
  - $O(\log_2 M \log_M n)$

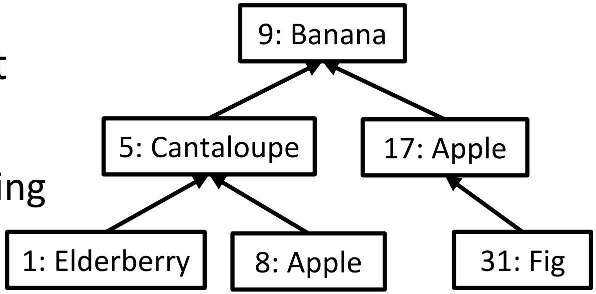


Decision #1: M-ary Order Property

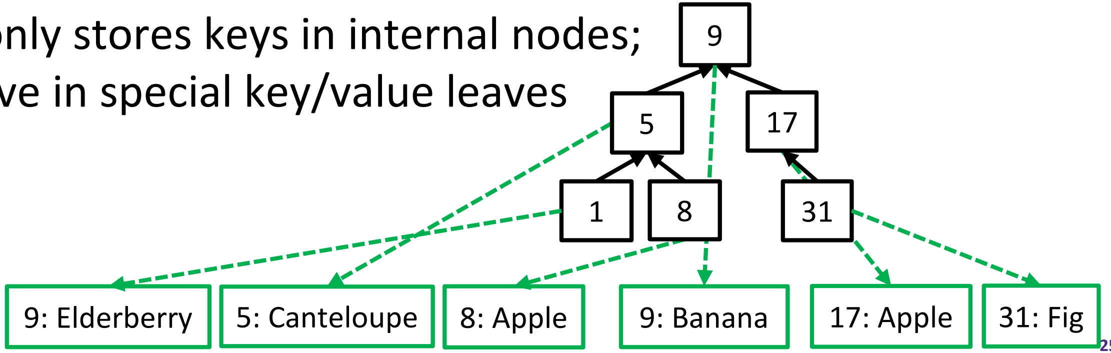
- M-ary search tree's order property is the M-way extension of a BST's 2-way ordering property
  - Subtree **between** keys ***a*** and ***b*** contains the keys between them
  - Ie, $a \leq k < b$

Decision #2: 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



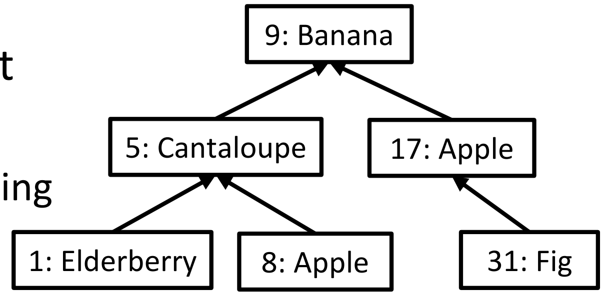
- B-Tree only stores keys in internal nodes; values live in special key/value leaves



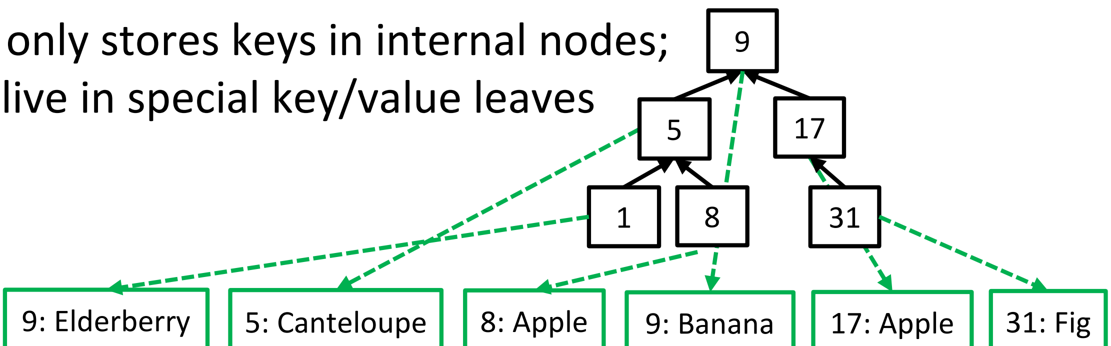
Decision #2: Key-only Internal Nodes

- 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



Lecture Outline

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

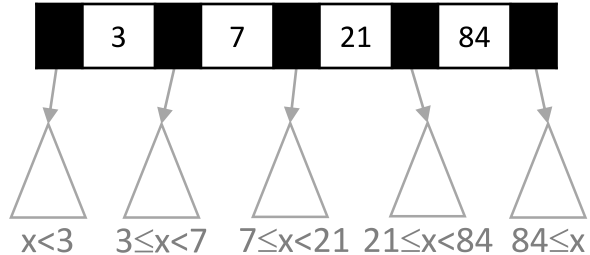
- B-Trees
  - Goals and Design
  - **B+ Tree Structure**
  - B+ Tree Implementation: Find
  - B+ Tree Implementation: Add

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

k_1	k_2	…	k_{m-1}	
ptr_1	ptr_2	…	ptr_{m-1}	ptr_m

(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)

k_1	v_1
k_2	v_2
…	…
k_L	v_L

(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

$$n \geq \underbrace{2\lceil M/2 \rceil^{h-1}}_{\text{minimum number of leaves}} \underbrace{\lceil L/2 \rceil}_{\text{minimum items per leaf}}$$

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

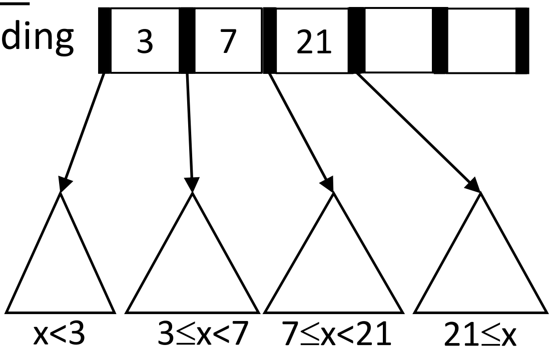
- Memory Hierarchy Basics
  - What is the Memory Hierarchy?
  - How does it impact data structure design?

- 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!*

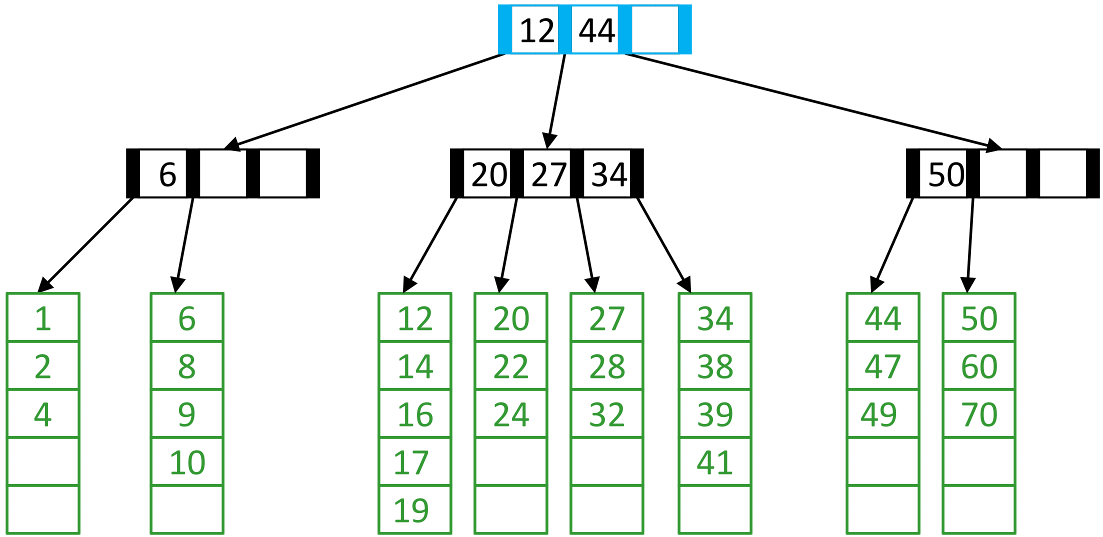
- Difference:
  - Since we *<u>don't store value at internal nodes</u>*, there is no "best case" of finding our value at the root node; must always traverse to the bottom of B+ Tree



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

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

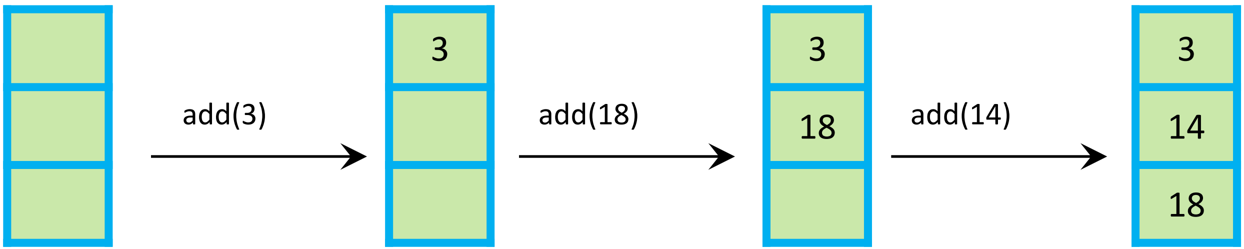
- B-Trees
  - Goals and Design
  - B+ Tree Structure
  - B+ Tree Implementation: Find
  - **B+ Tree Implementation: Add**

Add Example:

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

Min/Max slots in root: 2,3
Min/Max slots in interior node: 2,3
Min/Max slots in leaf: 2,3

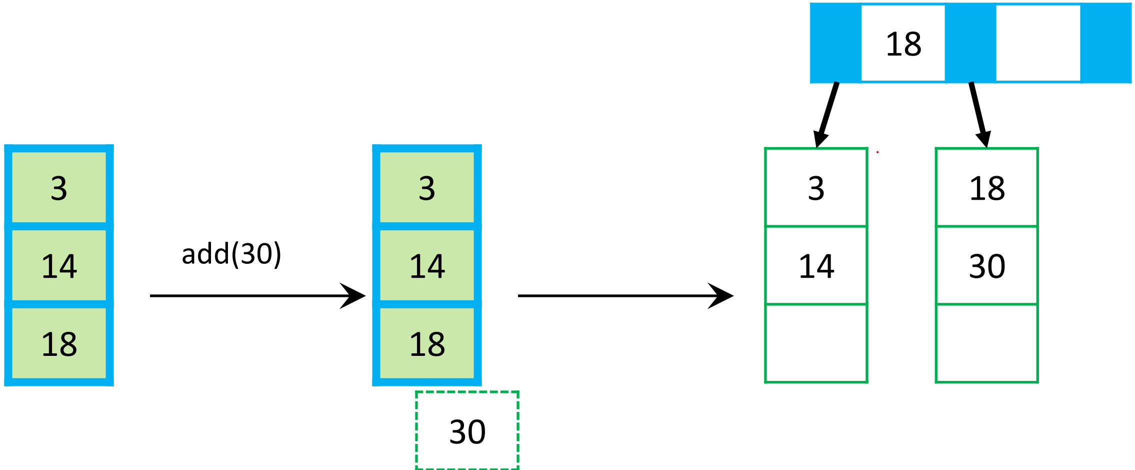
Add Example: Answer (1 of 7)



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)

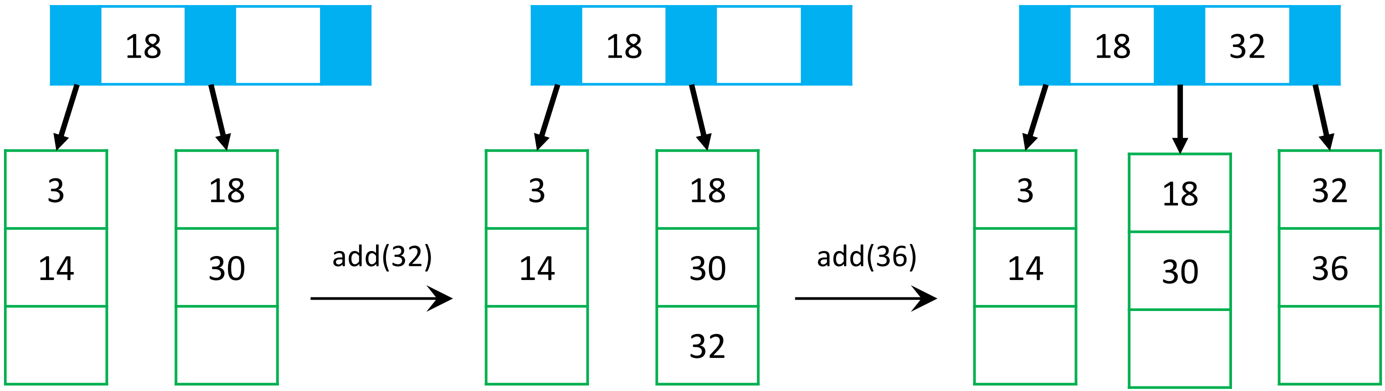


Special case: the **root** is a **leaf node**

- 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

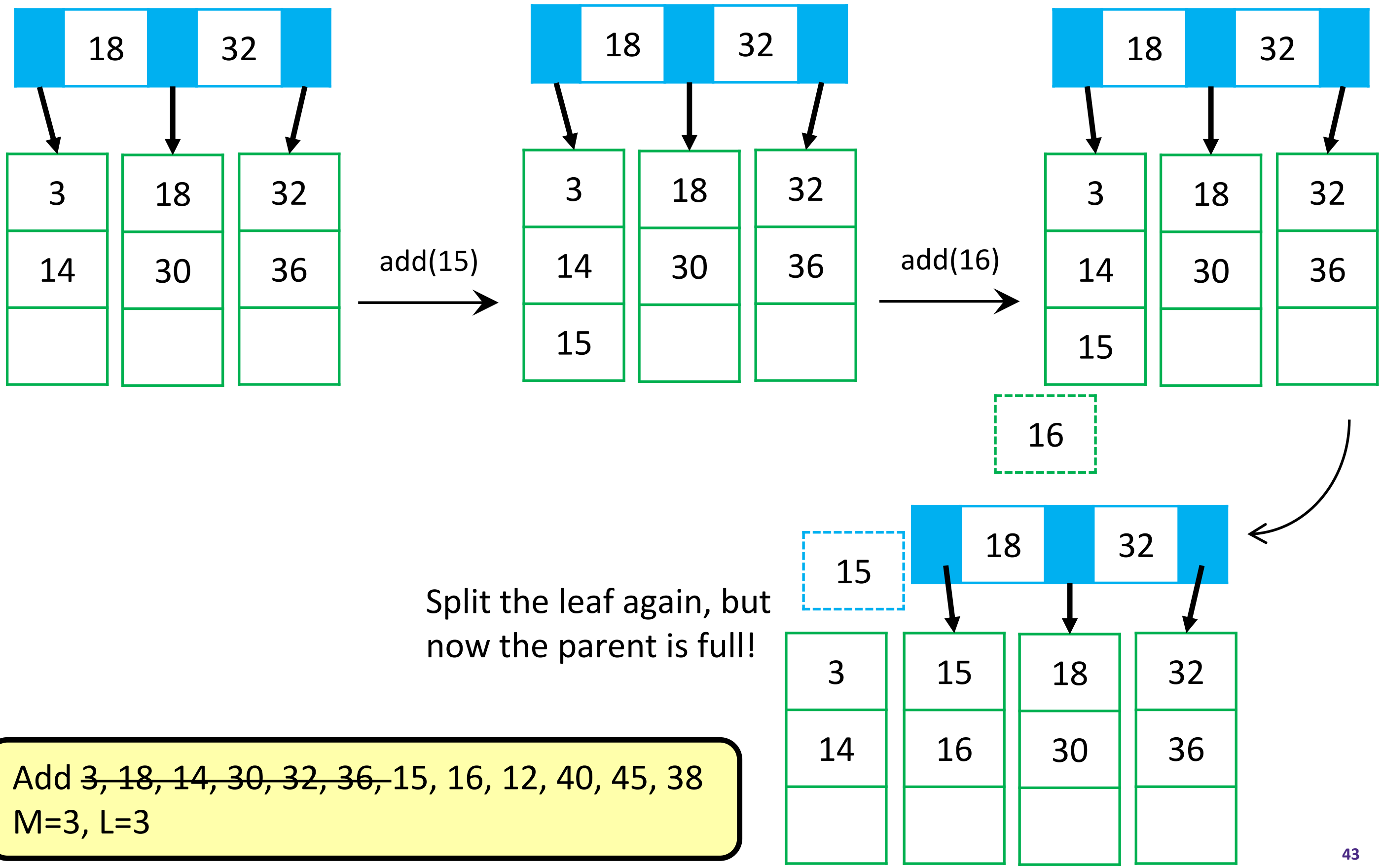
Add Example: Answer (3 of 7)



Split the leaf again

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

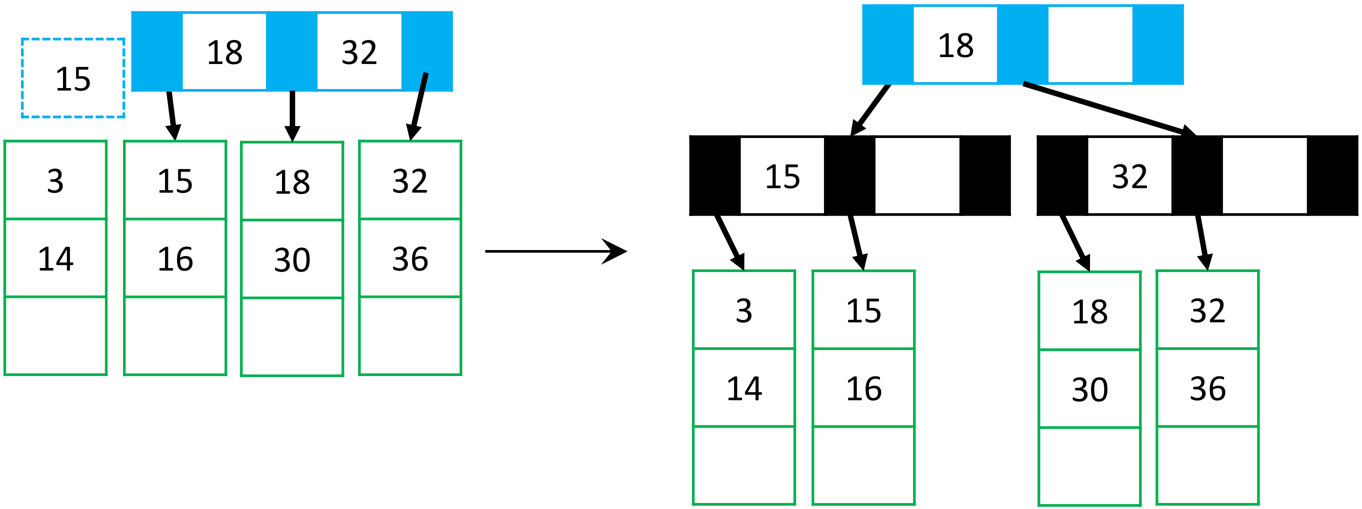
Add Example: Answer (4 of 7)



Split the leaf again, but now the parent is full!

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

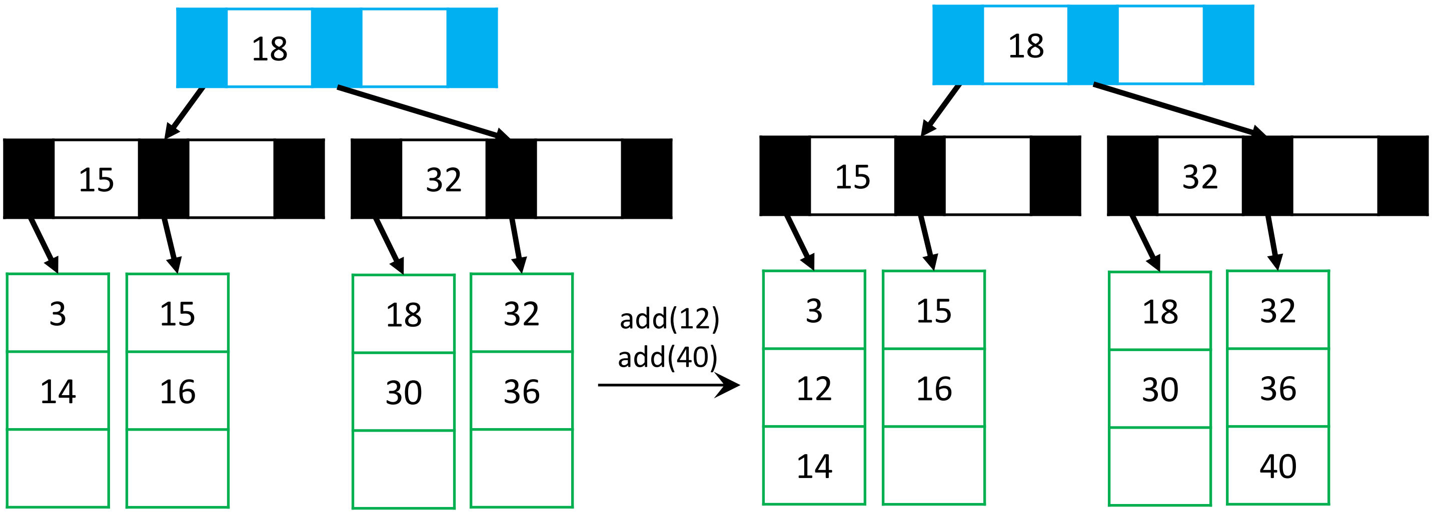
Add Example: Answer (5 of 7)



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

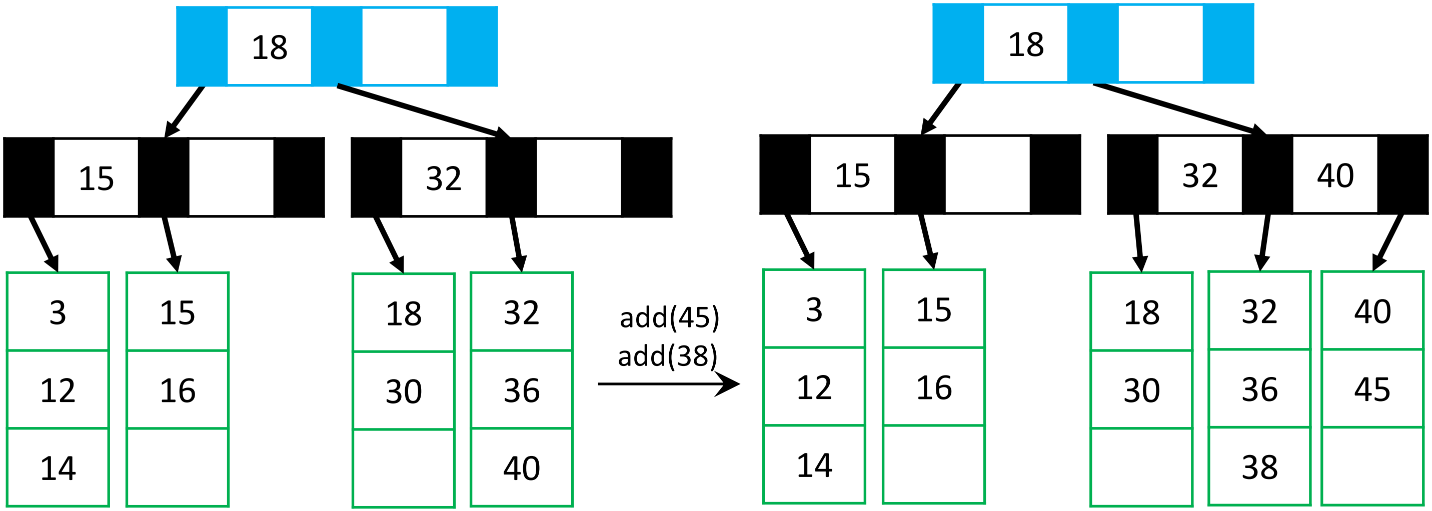
Add ~~3, 18, 14, 30, 32, 36, 15,~~ 16, 12, 40, 45, 38
M=3, L=3

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