LEC 11

CSE 373

Memory & Caching, B+ Trees

BEFORE WE START

Which of the following statements is true about an AVL Tree?

a) It remains perfectly balanced after an insert
b) The get operation has a better best-case runtime than get for a normal BST
c) Rotations always happen at the tree's root
d) At most one rotation (or double rotation) is needed to rebalance after an insert

pollev.com/uwcse373

Instructor: Aaron Johnston

TAs: Timothy Akintilo, Brian Chan, Joyce Elauria, Eric Fan, Farrell Fileas, Melissa Hovik, Leona Kazi, Keanu Vestil, Siddharth Vaidyanathan, Howard Xiao

Announcements

- EX2 (Due **TONIGHT 11:59pm**)
- P2 (Due next Wednesday)
- Mid-Quarter Survey out now
  - Let us know how the course is going!
- Exam I
  - Start forming groups if you haven't already! Consider posting on Discord's #find-a-partner channel
  - Practice exam released on Monday to help give you a picture of what to expect
  - Section next week will also be exam review
  - We highly recommend reviewing section problems, exercises, and post-lecture review questions!

Learning Objectives

After this lecture, you should be able to...

1. Contrast the CPU, RAM, the cache, and Disk in terms of their storage space and the time to access them

2. Explain why arrays tend to lead to better performance than linked lists, in terms of spatial locality

3. Describe how B+ Trees help minimize disk accesses and trace a get() operation in a B+ Tree (*Non-objective:* Be able to construct, modify, or explain every detail of a B+ Tree)

Review AVL Trees

INVARIANT

AVL Invariant

For every node, the height of its left and right subtrees may only differ by at most 1

Operation	Case	Runtime
`containsKey(key)`	best	Θ(1)
	worst	Θ(log n)
`insert(key)`	best	Θ(log n)
	worst	Θ(log n)
`delete(key)`	best	Θ(log n)
	worst	Θ(log n)

PROS

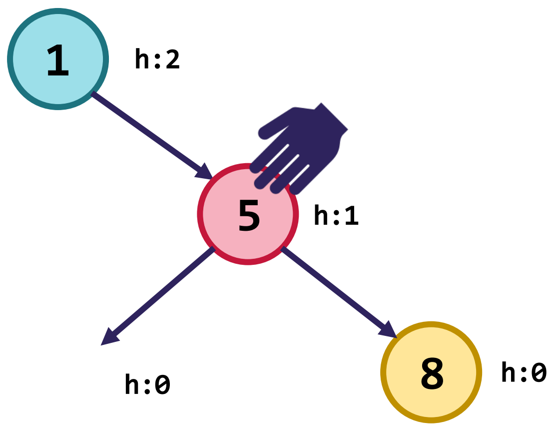
- All operations on an AVL Tree have a logarithmic worst case
  - Because these trees are always balanced!
- The act of rebalancing adds no more than a constant factor to insert and delete

➢ Asymptotically, just better than a normal BST!

CONS

- Relatively difficult to program and debug (so many moving parts during a rotation)
- Additional space for the height field
- Though asymptotically faster, rebalancing *does* take some time
  - Depends how important every little bit of performance is to you

Review Fixing AVL Invariant



Review Fixing AVL Invariant: Left Rotation

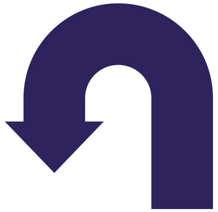
- In general, we can fix the AVL invariant by performing rotations wherever an imbalance was created
- **Left Rotation**
  - Find the node that is violating the invariant (here, 1)
  - Let it "fall" left to become a left child

- Apply a left rotation whenever the newly inserted node is located under the **right child of the right child**

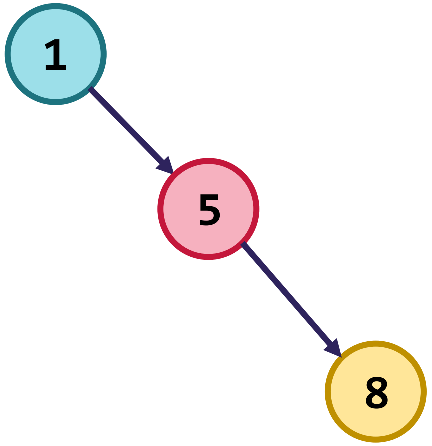
Review 4 AVL Rotation Cases

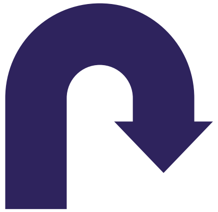
”Line” Cases

Solve with 1 rotation

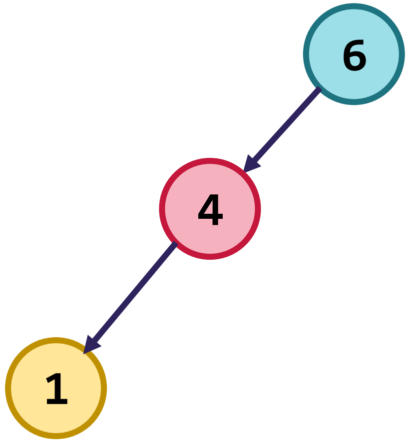


Left Rotation



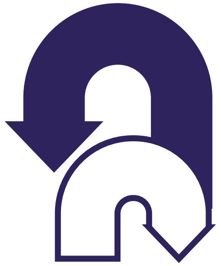


Right Rotation

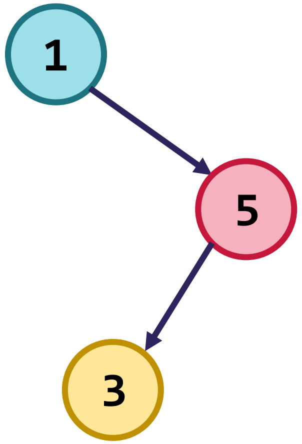


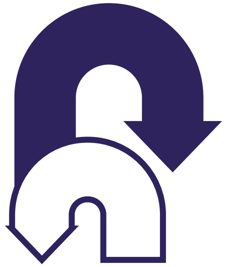
”Kink” Cases

Solve with 2 rotations

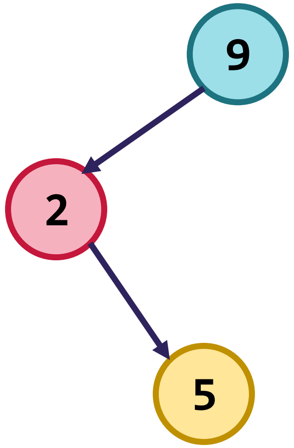


Right/Left Rotation





Left/Right Rotation

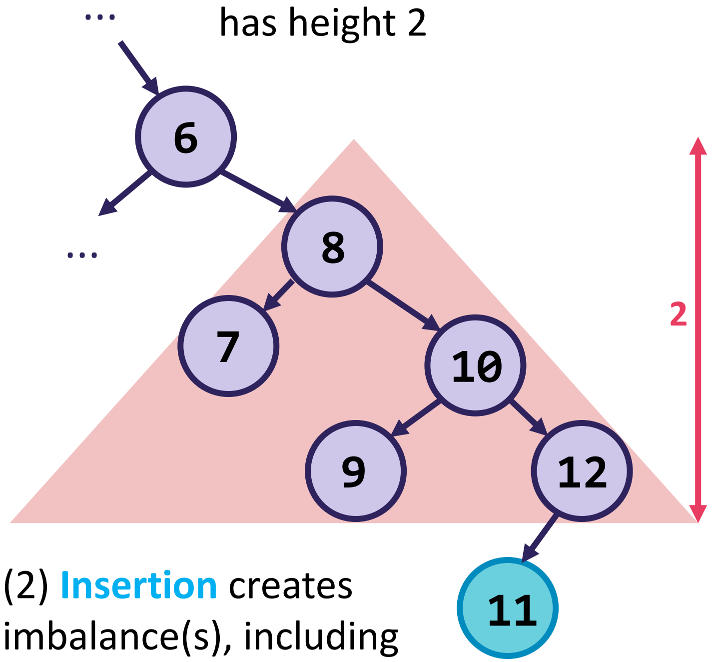


Review AVL `insert()`: Approach

- Our overall algorithm:
  1. Insert the new node as in a BST (a new leaf)
  2. For each node *on the path from the root to the new leaf*:
     - The insertion may (or may not) have changed the node's height
     - Detect height imbalance and perform a *rotation* to restore balance

- Facts that make this easier:
  - Imbalances can only occur along the path from the new leaf to the root
  - We only have to address the lowest unbalanced node
  - Applying a rotation (or double rotation), restores the height of the subtree before the insertion -- when everything was balanced!
  - Therefore, we need ***at most one rebalancing operation***

(1) Originally, whole tree balanced, and **this subtree** has height 2



(2) **Insertion** creates imbalance(s), including **the subtree** (8 is lowest unbalanced node)

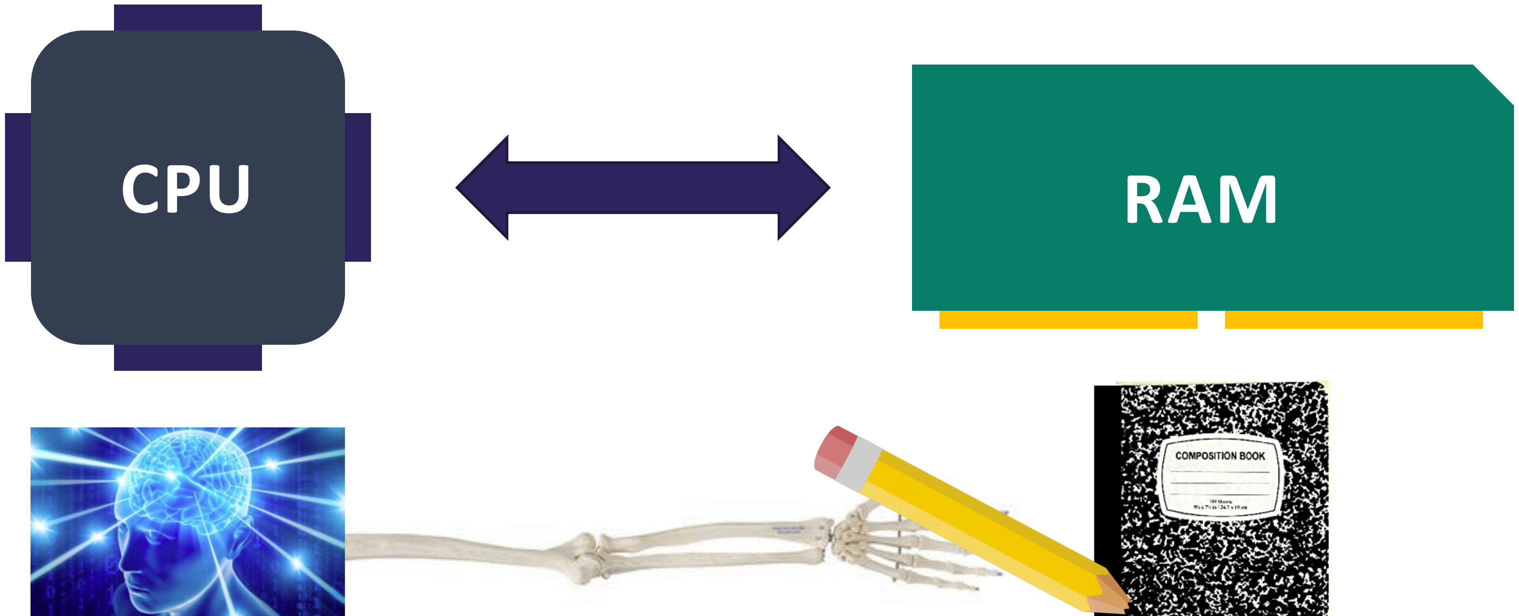
(3) Since the rotation on 8 will restore **the subtree** to height 2, whole tree balanced again!

Lecture Outline

- **Memory & Caching**
  - **How Memory Looks**
  - How Memory Is Used
- B+ Trees

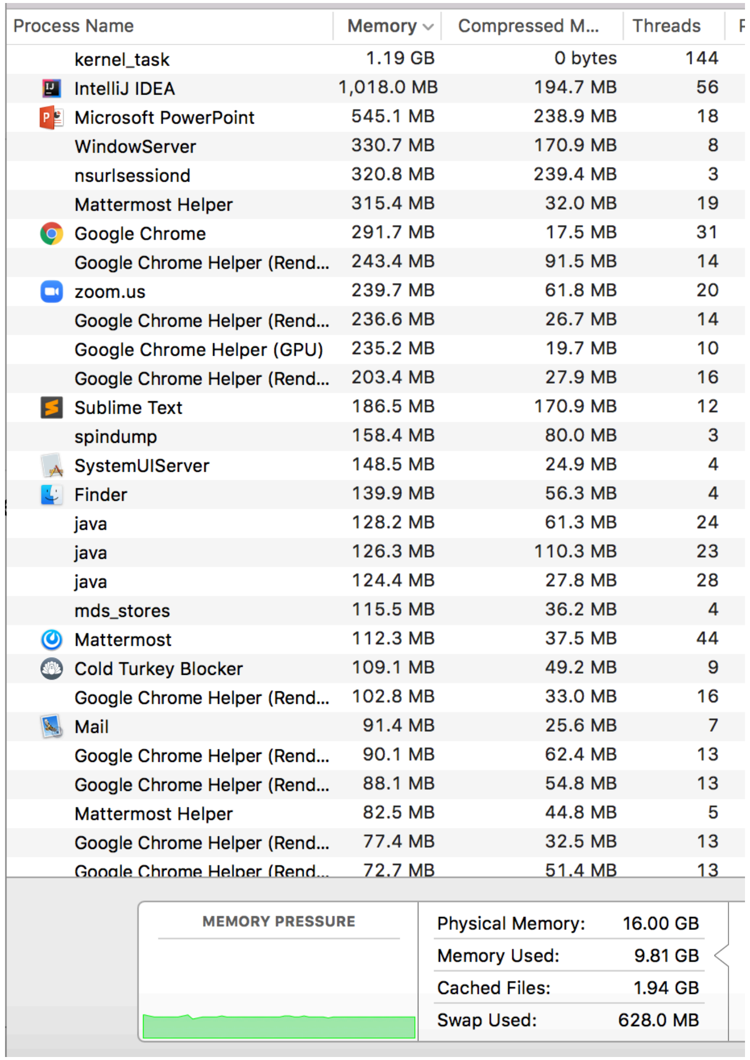
So… What *is* a Computer?

- At the simplest level, think of a computer as being two components:
  - CPU: Central Processing Unit (The "brain". When any operation is run, it's running in the CPU. Takes in inputs and evaluates an output.)
  - RAM: Random Access Memory (The "notebook". Where data is kept track of, and stored between operations. Inputs are read from here and outputs are written here.)



RAM (Random-Access Memory)

- **RAM is where the programs you run store their data.**
  - Data structures, variables, method call frames, etc. all stored here!
- Often just called "Memory" or "Main Memory"



Process Name	Memory	Compressed M...	Threads
kernel_task	1.19 GB	0 bytes	144
IntelliJ IDEA	1,018.0 MB	194.7 MB	56
Microsoft PowerPoint	545.1 MB	238.9 MB	18
WindowServer	330.7 MB	170.9 MB	8
nsurlsessiond	320.8 MB	239.4 MB	3
Mattermost Helper	315.4 MB	32.0 MB	19
Google Chrome	291.7 MB	17.5 MB	31
Google Chrome Helper (Rend...	243.4 MB	91.5 MB	14
zoom.us	239.7 MB	61.8 MB	20
Google Chrome Helper (Rend...	236.6 MB	26.7 MB	14
Google Chrome Helper (GPU)	235.2 MB	19.7 MB	10
Google Chrome Helper (Rend...	203.4 MB	27.9 MB	16
Sublime Text	186.5 MB	170.9 MB	12
spindump	158.4 MB	80.0 MB	3
SystemUIServer	148.5 MB	24.9 MB	4
Finder	139.9 MB	56.3 MB	4
java	128.2 MB	61.3 MB	24
java	126.3 MB	110.3 MB	23
java	124.4 MB	27.8 MB	28
mds_stores	115.5 MB	36.2 MB	4
Mattermost	112.3 MB	37.5 MB	44
Cold Turkey Blocker	109.1 MB	49.2 MB	9
Google Chrome Helper (Rend...	102.8 MB	33.0 MB	16
Mail	91.4 MB	25.6 MB	7
Google Chrome Helper (Rend...	90.1 MB	62.4 MB	13
Google Chrome Helper (Rend...	88.1 MB	54.8 MB	13
Mattermost Helper	82.5 MB	44.8 MB	5
Google Chrome Helper (Rend...	77.4 MB	32.5 MB	13
Google Chrome Helper (Rend...	72.7 MB	51.4 MB	13

MEMORY PRESSURE

Physical Memory:	16.00 GB
Memory Used:	9.81 GB
Cached Files:	1.94 GB
Swap Used:	628.0 MB

Think of RAM as a Giant Array!

- Fortunately, as programmers we don’t need to understand the circuitry below!
- We think about RAM through the **abstraction** of a giant array:
  - Stores data in specific locations
  - Indices to describe those locations (we call them addresses for memory)
  - We can jump to any index (“random access”)

- RAM is really a physical chip in your computer consisting of complicated circuitry

LOW-LEVEL REALITY

HIGH-LEVEL ABSTRACTION

Simple Data in RAM

a: refers to address 107
letter: refers to address 113

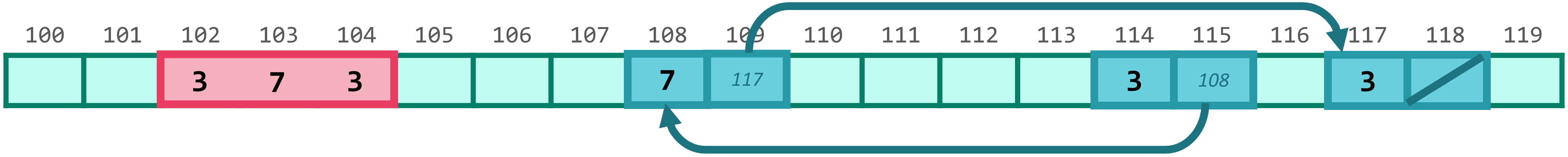
```
int a = 5;
char letter = ‘z’
```

100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119
							5						‘z’						

Data Structures in RAM

```
int[] array = new int[3];
array[0] = 3;
array[1] = 7;
array[2] = 3;
```

```
Node front = new Node(3);
front.next = new Node(7);
front.next.next = new Node(3);
```



- An array is a contiguous block of memory (a bunch of slots next to each other)
- A linked list is a series of nodes, with references to each other
  - How to reference? Simply store the address!
  - Nodes do not need to be contiguous, or even in order

Lecture Outline

- Memory & Caching
  - How Memory Looks
  - How Memory Is Used
- B+ Trees

Buying Bubble Tea

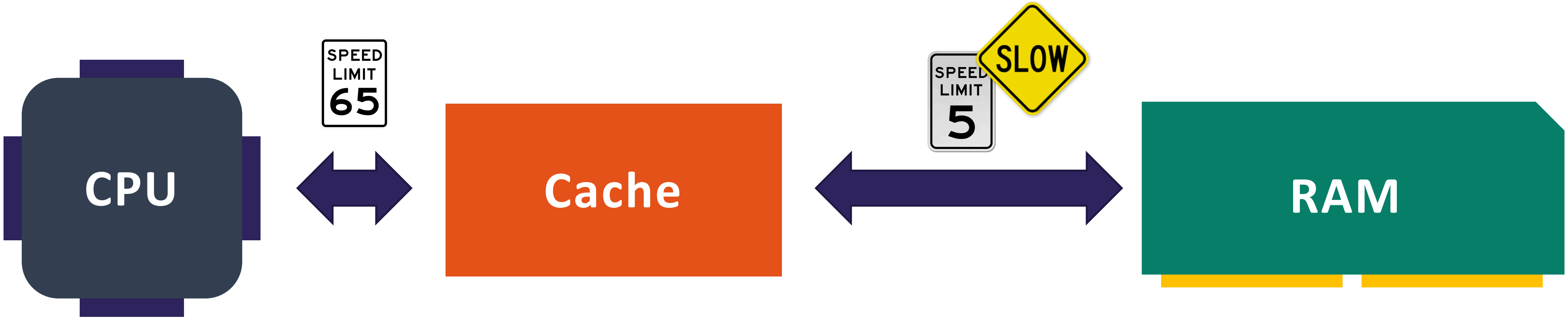
- Suppose there's some ~~treat~~ essential grocery you need every few hours
- As soon as you realize you're thirsty, you:
  - (1) Walk to the store (2) Buy a bubble tea (3) Walk back home (4) Enjoy
- But you repeat this multiple times a day! It takes so long to walk to the store, and that's a lot of time spent away from 373 lecture...

Buying Bubble Tea: Planning Ahead

- Could this be more efficient?
- Since you know it's likely you'll want another bubble tea in a few hours, what if you do what any reasonable person would: buy a bubble tea minifridge and store a handful closer to home!

Cache

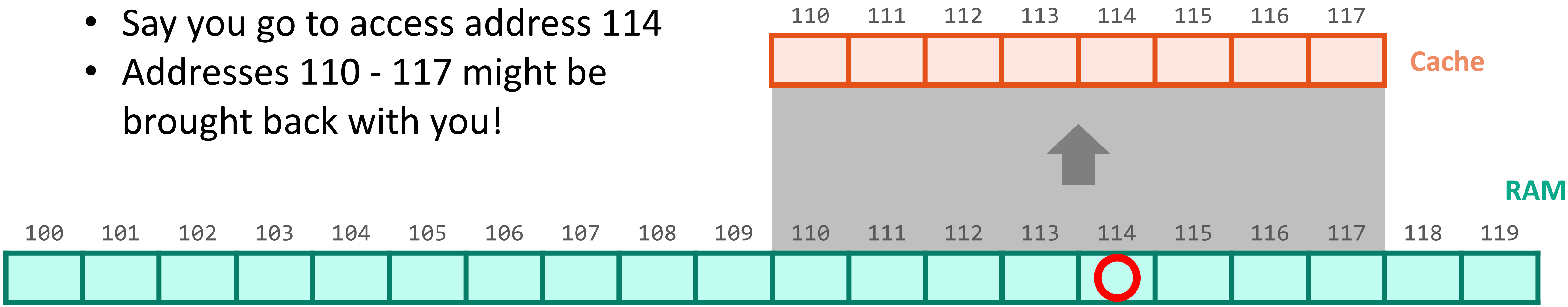
- Pronounced “cash”
- Intermediate storage between the CPU and RAM
  - RAM takes a long time to access, but is gigantic. Cache is much faster (closer to the CPU where data gets processed), but smaller.
- Store a copy of some data here
  - When we’re about to go grab an address from RAM, we check the cache first – and we *love* when the data’s there, because it’s much faster!



Bringing More Data Back

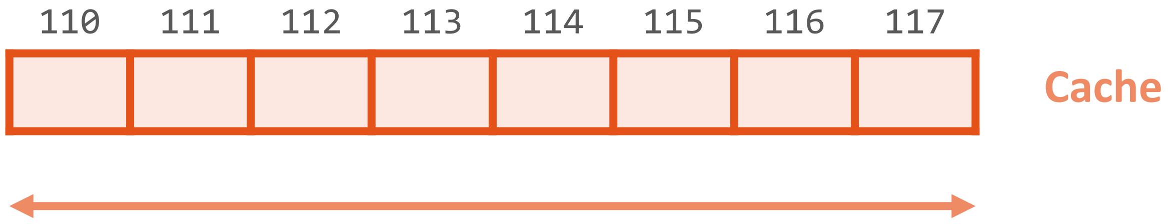
- If we need to go all the way to RAM, might as well make it count!
- Your computer *automatically* grabs a whole chunk of data around each address from RAM when you access it
  - That chunk of data is then copied to the cache
  - Your computer knows its's likely you'll want a nearby address soon
  - Bringing back multiple addresses of data costs nothing: the hardware is designed to grab many at a time

- Say you go to access address 114
- Addresses 110 - 117 might be brought back with you!



Cache Implications: Arrays

- This has a major impact on programming with arrays!
  - Suppose we're looping through everything in an array. When we access index 0, we grab a whole chunk of the array and put it in the cache – now the next (block size) accesses are much faster!
  - For a short array, we might even grab the whole thing and bring it into the cache



The chunk of memory that gets brought back is a "block"

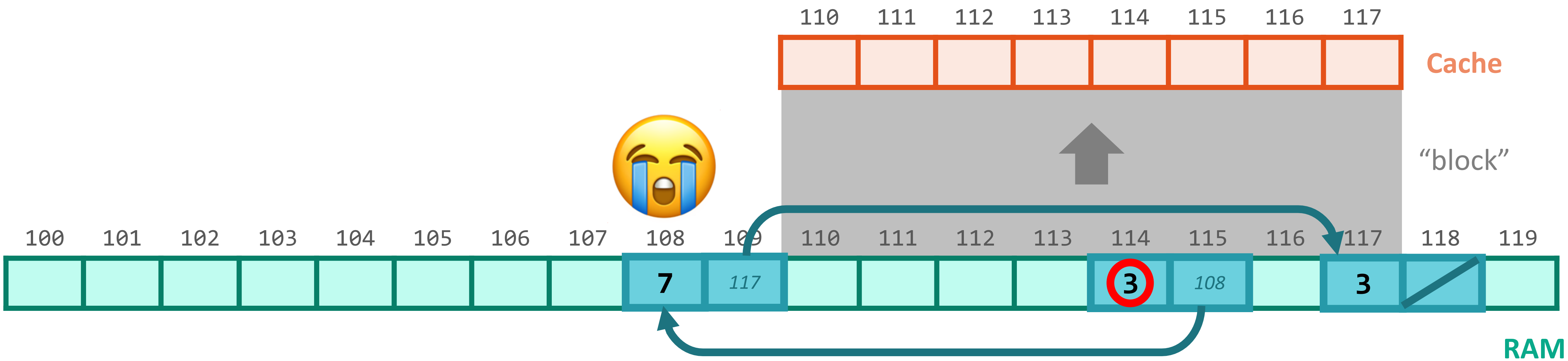
Characterizing Cache-Friendly Programs

- **Spatial locality**: tendency for programs to access locations nearby to recent locations
  - Plenty of our programs exhibit spatial locality: e.g. looping through an array
- **Temporal locality**: tendency for programs to access data that was recently accessed
  - Plenty of our programs exhibit temporal locality: e.g. adding to sum variable over and over

- Programs with spatial and temporal locality benefit the most from caching!

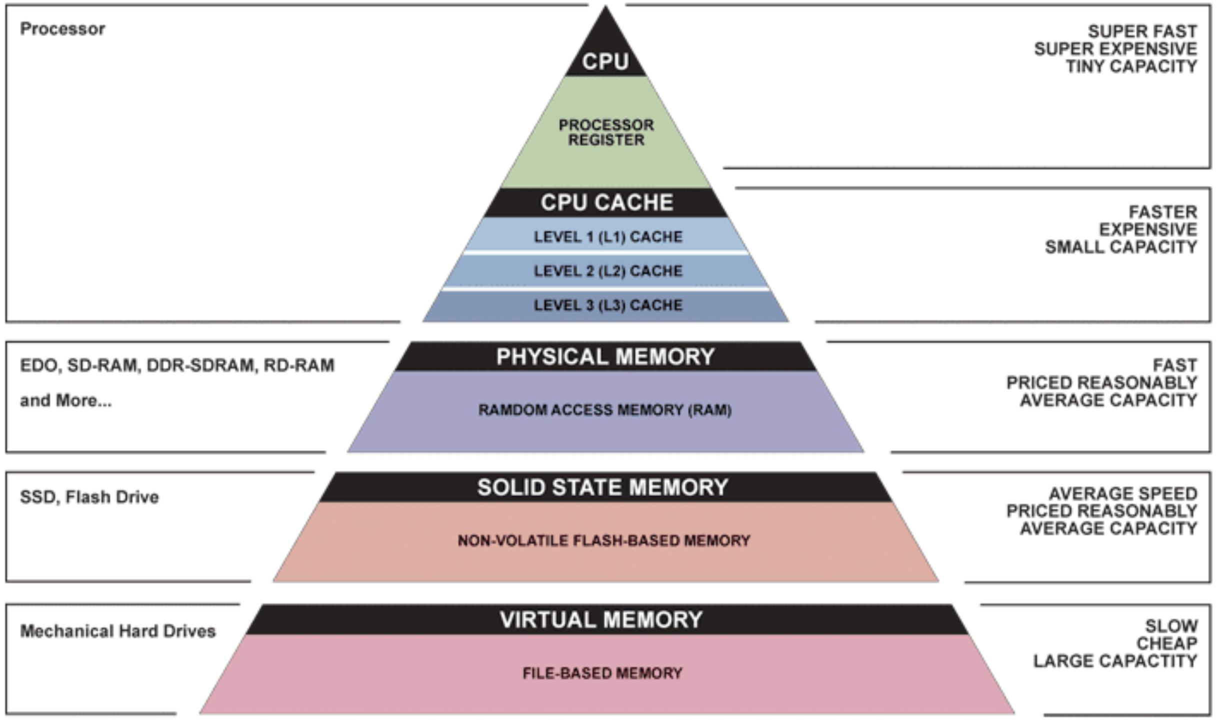
Cache Implications: Linked Lists

- Linked lists can be spread out all over the RAM array
  - Do not exhibit strong spatial locality!
- Don't get the same cache benefits – frequently the next list node is far enough away that it's not included on the same block



Memory Architecture

- Typically multiple caches (progressively smaller and faster: L1, L2, & L3)
- Beyond RAM is the disk, which is way, way, WAY slower – but *much* bigger, & disk memory persists when the computer is off (RAM gets cleared)
  - **Similar idea: chunk of data gets pulled into RAM when accessed on disk (called a "page")**



▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

Asymptotic Analysis, Meet The Real World

- Asymptotic analysis tells us iterating through an array and a linked list are the same complexity class (linear)
  - This is still true: *growth rates* are the same, and asymptotic analysis is a helpful tool to capture that
  - But arrays are frequently a *significant* constant factor faster due to cache performance! One area asymptotic analysis isn't a good tool for
- https://repl.it/repls/MistyroseLinedTransformation *(~15 sec to run)*

"Latency Numbers Everyone Should Know" from Jeff Dean, Senior Fellow at Google and UW Alum!

L1 cache reference	**0.5 ns**	
Branch mispredict	5 ns	
L2 cache reference	7 ns	
Mutex lock/unlock	100 ns	
Main memory reference	**100 ns**	
Compress 1K bytes with Zippy	10,000 ns	0.01 ms
Send 1K bytes over 1 Gbps network	10,000 ns	0.01 ms
Read 1 MB sequentially from memory	250,000 ns	0.25 ms
Round trip within same datacenter	500,000 ns	0.5 ms
Disk seek	**10,000,000 ns**	10 ms
Read 1 MB sequentially from network	10,000,000 ns	10 ms
Read 1 MB sequentially from disk	30,000,000 ns	30 ms
Send packet CA->Netherlands->CA	150,000,000 ns	150 ms

Where
1 ns = 10^{-9} seconds
1 ms = 10^{-3} seconds

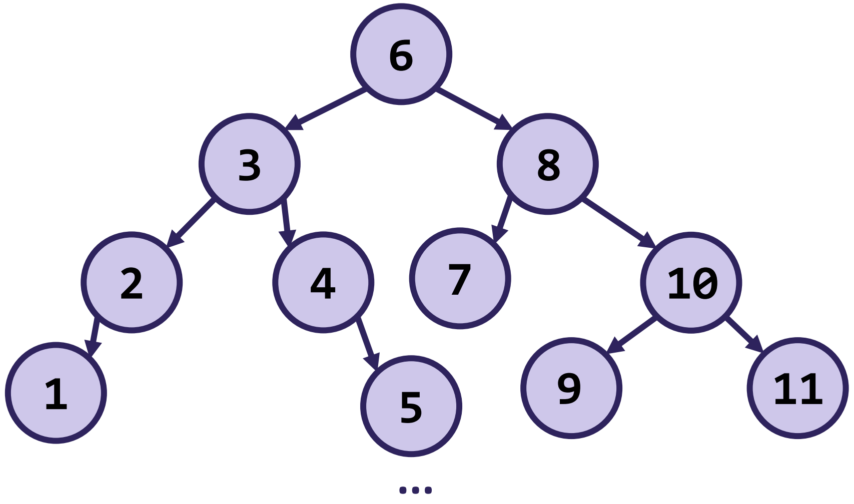
Lecture Outline

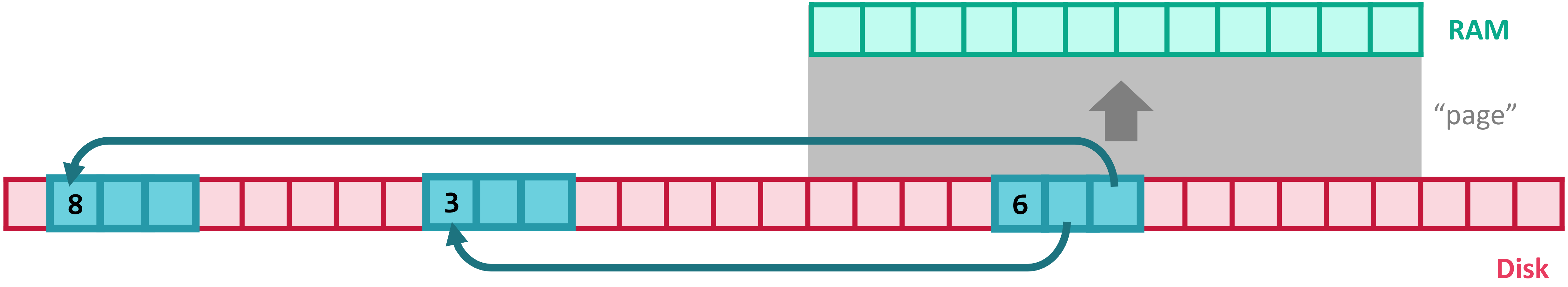
- Memory & Caching
  - How Memory Looks
  - How Memory Is Used
- **B+ Trees**

Minimizing Disk Accesses

A laptop these days might have:
8 GB of RAM
250 GB of Disk space

- Let's consider a truly massive amount of data – too much data to fit in RAM (some has to be stored on disk)
  - This is very common! For example, a database
- What will happen if we store it in a giant AVL tree? Say height 40, so $2^{40} = 1.1 * 10^{12}$ nodes
  - Similar problem as before, just with disk this time: nodes are too spread out to be captured on a single disk read ("page")





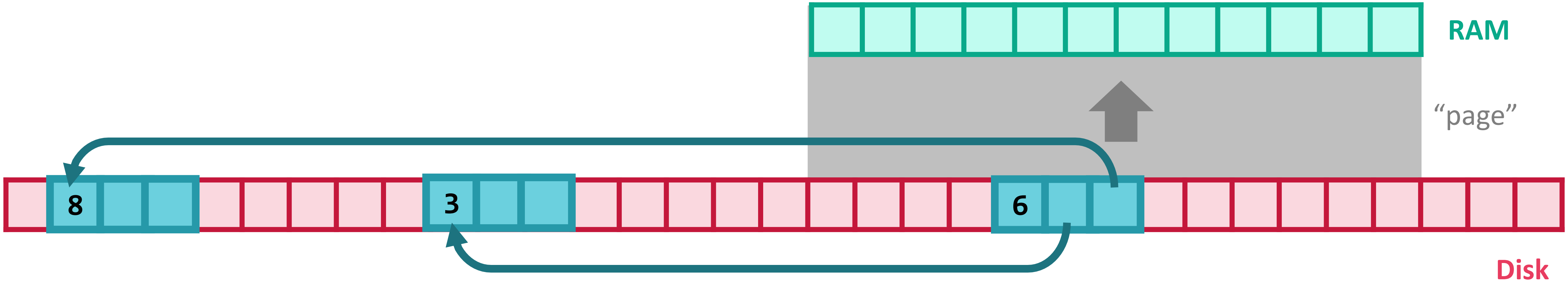
Minimizing Disk Accesses

A laptop these days might have:
8 GB of RAM
250 GB of Disk space

- Let's consider a truly massive amount of data – too much data to fit in RAM (some has to be stored on disk)
  - This is very common! For example, a database
- What will happen if we store it in a giant AVL tree? Say height 40, so $2^{40} = 1.1 * 10^{12}$ nodes
  - Similar problem as before, just with disk this time: nodes are too spread out to be captured on a single disk read ("page")

?

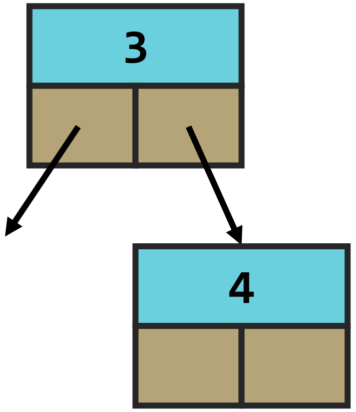
Our goal:
A data structure optimized to make **as few disk accesses as possible**! (suitable for large amounts of data)

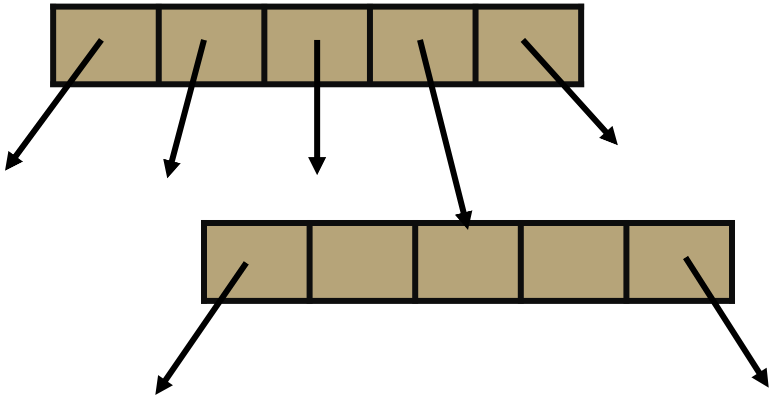


Minimizing Disk Accesses: Idea 1/3

- Idea: Node size of our BSTs/AVL Trees is small, but we move a whole page at a time in from disk
  - What if we could stuff more useful information in each node?
- First, let's generalize the number of children: while a Binary Tree has at most 2 children, an "M-ary" Tree has at most M children

Binary Search Tree





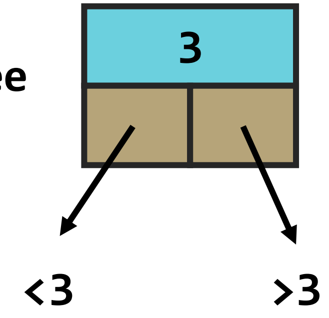
M-ary Tree

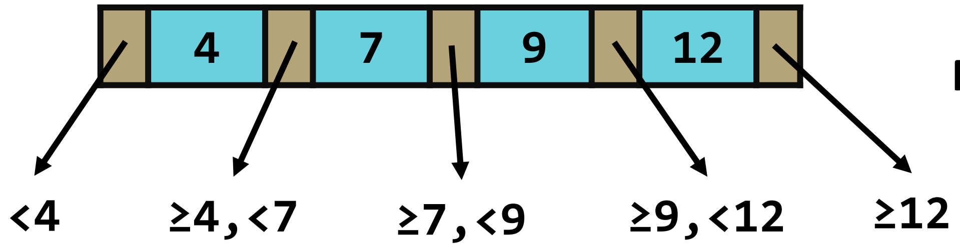
- This is incomplete: How do we keep these children organized? What happens to the key?

Minimizing Disk Accesses: Idea 2/3

- How do we keep these children organized? What happens to the key?
- In a Binary Search Tree, the key divides the contents of the **child subtrees**
  - Same principle: in our tree, we have a **sorted** array of M-1 keys, which divide the contents of child subtrees

Binary Search Tree



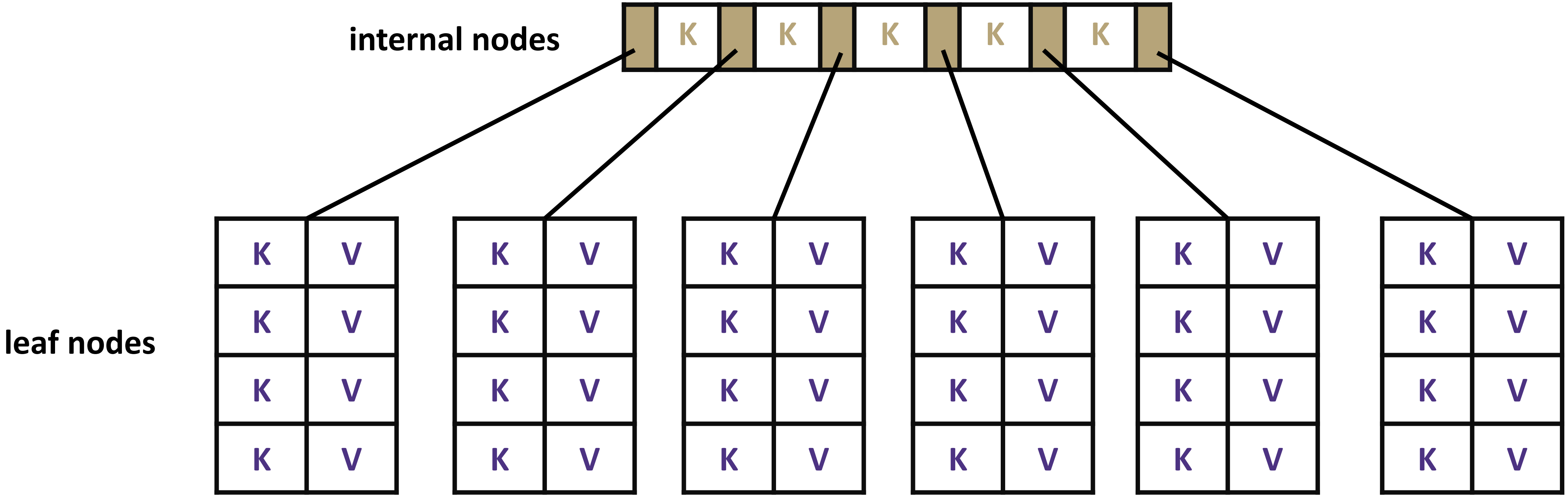


M-ary Tree

- Suppose we want to store values too (implement the Map ADT, useful for a database)? Where should we put those?

Minimizing Disk Accesses: Idea 3/3

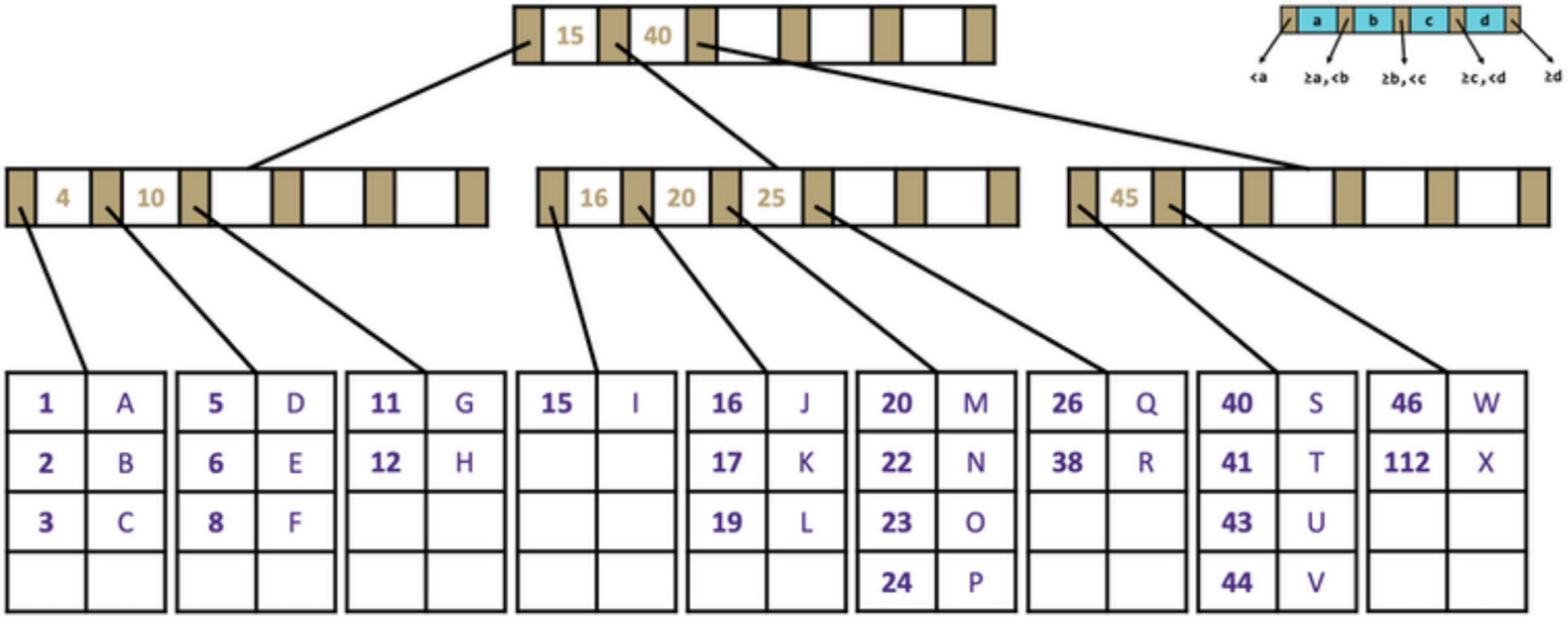
- We can pack all the key/value pairs into the leaf nodes, to really maximize stuffing in useful information
- This is a **B+ Tree**: a disk-friendly data structure™
  - Internal nodes become "fenceposts" that guide us to the leaves, leaves have all the data
  - **Both types of nodes sized to fit on a single page**!



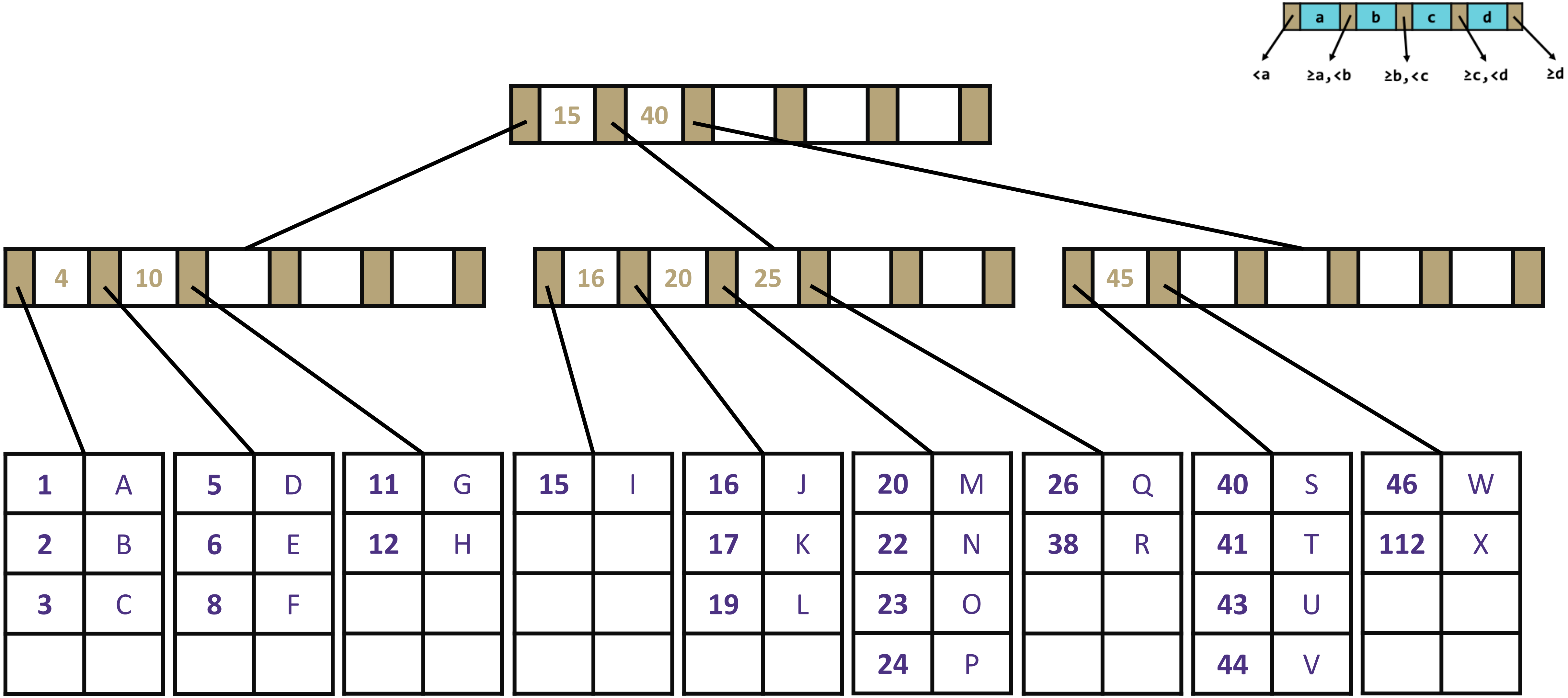
Poll Everywhere

pollev.com/uwcse373

B+ Tree Example: get(23)



B+ Tree Example: get(23)



Why Are B+ Trees so Disk-Friendly? (Summary)

1. We **minimized the height** of the tree by adding more keys/potential children at every node. Because the nodes are more spread out at a shallower place in the tree, it takes fewer nodes (disk-accesses) to traverse to a leaf.
2. All relevant information about a single node **fits in one page** (If it's an internal node: all the keys it needs to determine which branch it should go down next. If it's a leaf: the relevant K/V pairs).
3. We use **as much of the page as we can**: each node contains many keys that are all brought in at once with a single disk access, basically "for free".
4. The time needed to do a search within a node is **insignificant** compared to disk access time, so looking within a node is also "free".

What About Inserting/Removing?

- Beyond the scope of this class
- Our goal in 373: to learn enough about B+ Tree usage so you know when to consider using one in your program! You don't need to be able to implement.

- Takeaways:
  - Disk lookups are slow, so if you have large amounts of data (enough that it spills over onto the disk), consider using a B+ trees!
    - Databases use these *all* the time! Even the very core file system in your computer makes use of B+ trees
  - B+ trees minimize the # of disk accesses by stuff as much data into each node so that the height of tree is short, and every node requires just one disk access

B+ Tree Invariants

- Defined by 3 different invariants:
  1. B+ trees must have two different types of nodes: internal nodes and leaf nodes
     - An **Internal Node** contains *M* pointers to children and *M* – 1 **sorted** keys. (M must be greater than 2)
     - A **Leaf Node** contains *L* key-value pairs, sorted by key.
  2. B+ trees order invariant
     - For any given key k, all subtrees to the left may only contain keys that satisfy x < k
     - All subtrees to the right may only contain keys x that satisfy k >= x
  3. B+ trees structure invariant
     - If n <= *L*, the root is a leaf
     - If n >= *L*, root node must be an internal node containing 2 to M children
     - All nodes must be at least half-full

Diving Deeper into the Computer

- In CSE 373, we only need to know enough about the computer's workings to understand how it could impact performance
- But there's *so* much more to learn if you're interested! A really cool topic to explore
- Great place to get started: https://www.youtube.com/watch?v=fpnE6UAfbtU
- There are plenty of UW ECE courses that go into these details!