# Lecture 25: B Trees

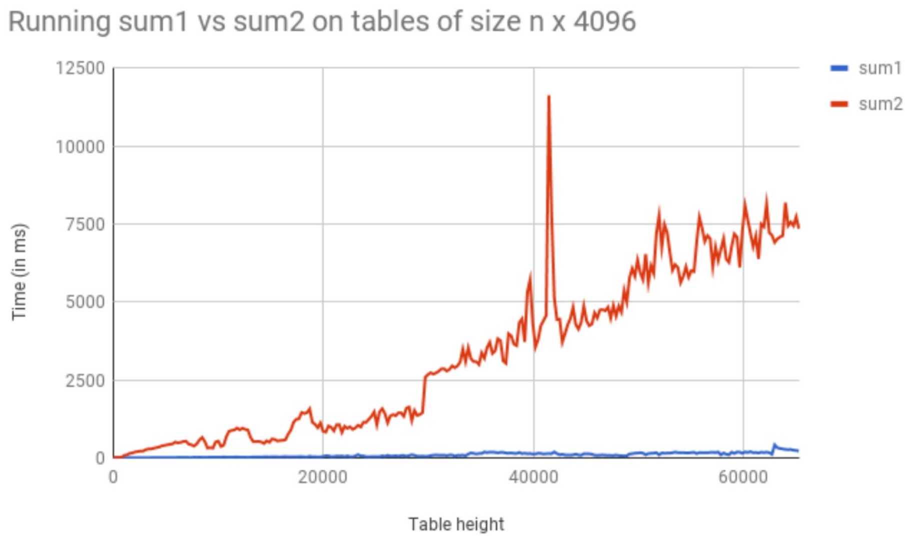
CSE 373: Data Structures and Algorithms

## Warm Up

Why might these two methods, who have identical asymptotic analysis, produce such different runtime graphs?

```
public int sum1(int n, int m, int[][] table) {
   int output = 0;
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         output += table[i][j];
      }
   }
   return output;
}
```

```
public int sum2(int n, int m, int[][] table) {
   int output = 0;
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         output += table[j][i];
      }
   }
   return output;
}
```



## Announcements

Class Policy Updates:

- Everyone gets +2 late days (thanks TAs!)
- Extending the late turn in from 3 days after due date to 5 days after due date

P4 due Wednesday June 2nd

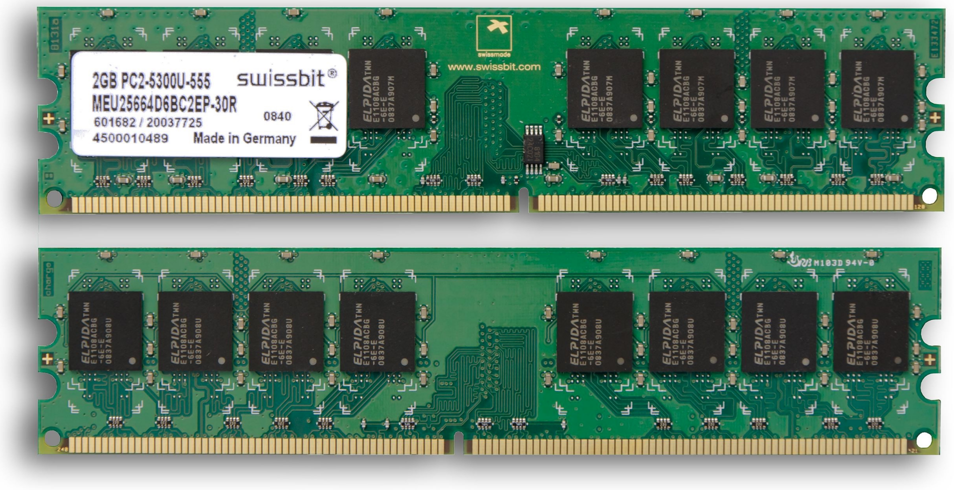
Tech Career Resources

- Section 9 Thursday 5/27 Interview Prep

## Memory & Locality!

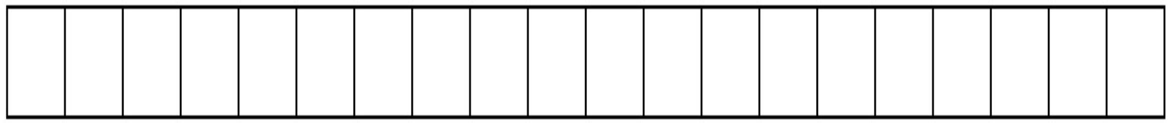
# RAM can be represented as a huge array

RAM:
- addresses, storing stuff at specific locations
- random access



=

Arrays
- indices, storing stuff at specific locations
- random access

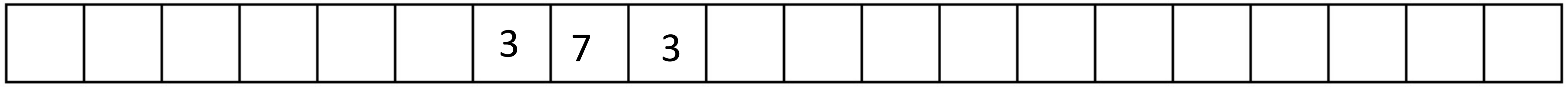


This is a main takeaway

If you're interested in deeper than this : https://www.youtube.com/watch?v=fpnE6UAfbtU or take some EE classes?

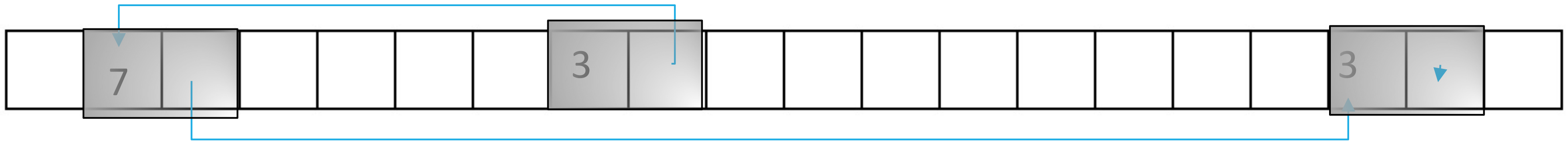
# A rough view of arrays and linked lists

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

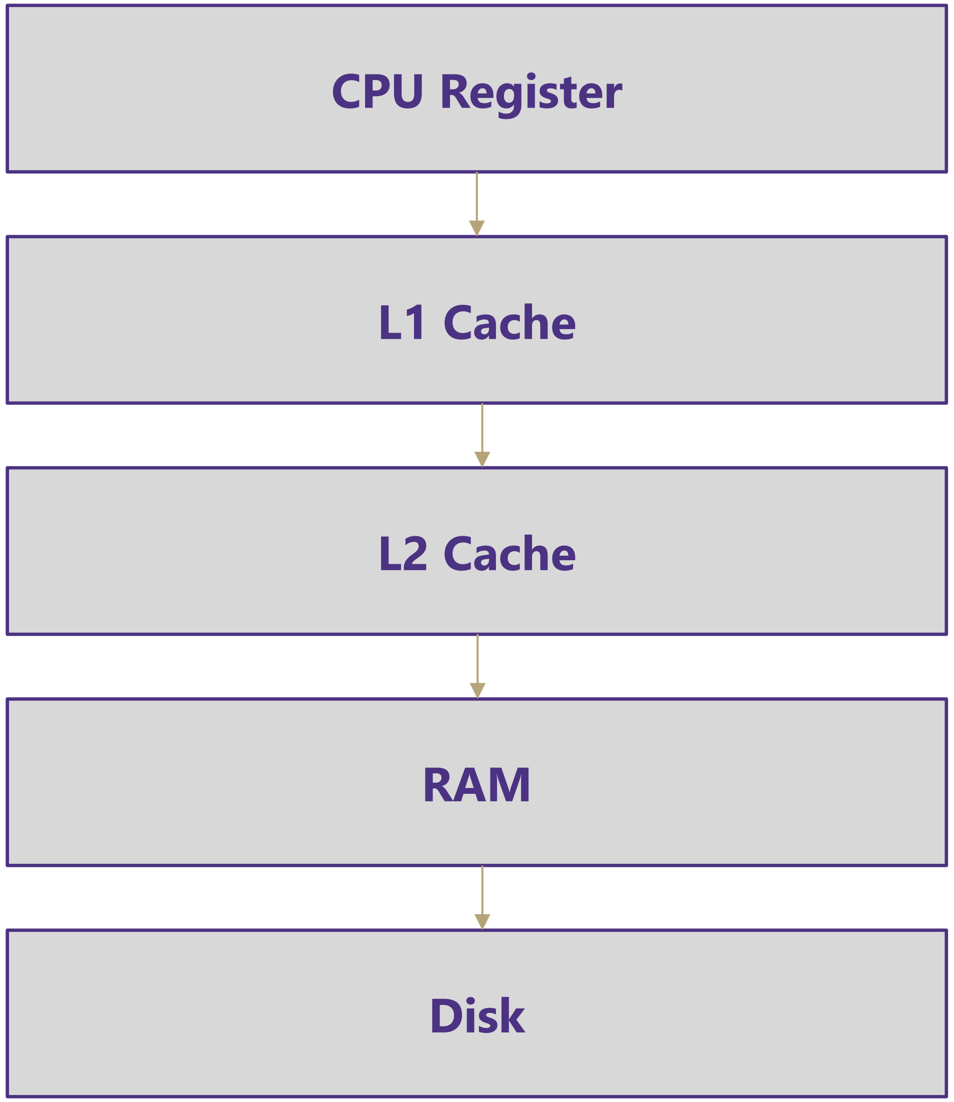


```
Node front = new Node(3);
front.next = new Node(7);
front.next.next = new Node(3);
```

(drawing singly linked list instead of doubly because drawings are hard / the two are similar)



# Memory Architecture



| What is it? | Typical Size | Time |
|---|---|---|
| The brain of the computer! | 32 bits | ≈free |
| Extra memory to make accessing it faster | 128KB | 0.5 ns |
| Extra memory to make accessing it faster | 2MB | 7 ns |
| Working memory, what your programs need | 8GB | 100 ns |
| Large, longtime storage | 1 TB | 8,000,000 ns |

# Memory Architecture

Takeaways:

- the more memory a layer can store, the slower it is (generally)
- accessing the disk is **very** slow

Computer Design Decisions

- Physics
  - Speed of light
  - Physical closeness to CPU
- Cost
  - “good enough” to achieve speed
  - Balance between speed and space

# Locality

How does the OS minimize disk accesses?

## Spatial Locality

Computers try to partition memory you are likely to use close by

- Arrays
- Fields

## Temporal Locality

Computers assume the memory you have just accessed you will likely access again in the near future

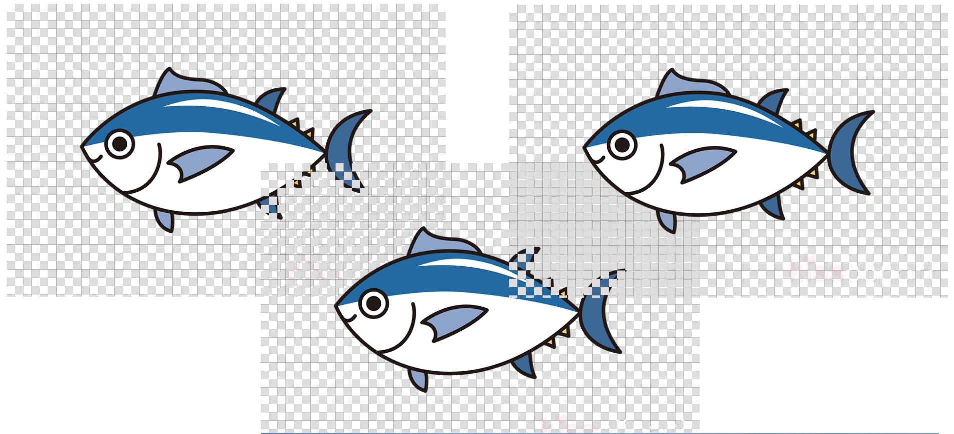
## Leveraging Spatial Locality

When looking up address in “slow layer”

- bring in more than you need based on what’s near by
- cost of bringing 1 byte vs several bytes is the same
- Data Carpool!

## How memory is used and moves around

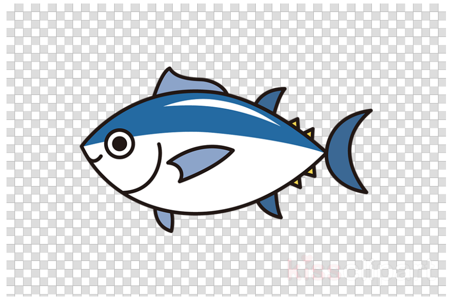
shutterstock.com • 298428176

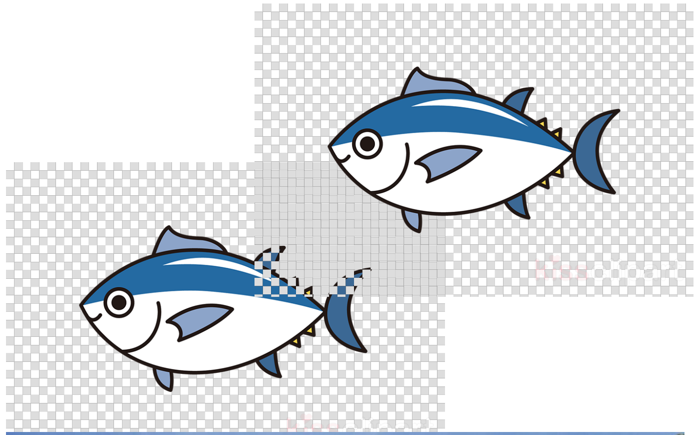


SAFEWAY

SAFEWAY

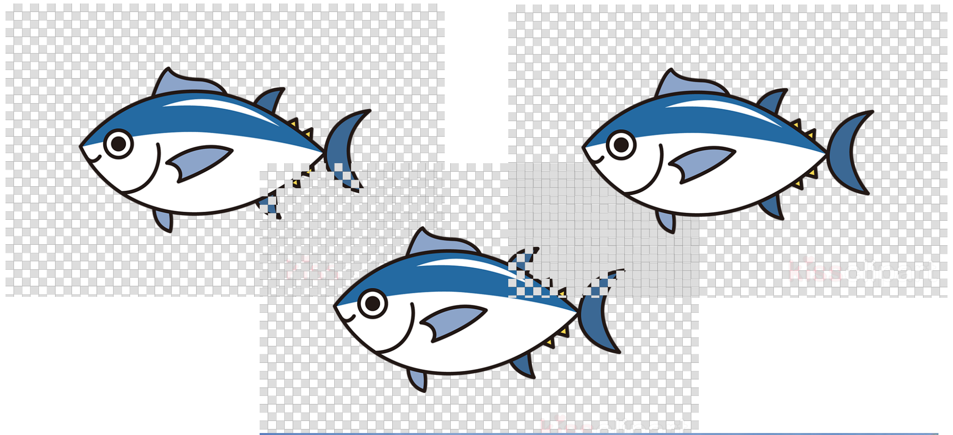
shutterstock.com • 298428176





SAFEWAY

shutterstock.com • 298428176

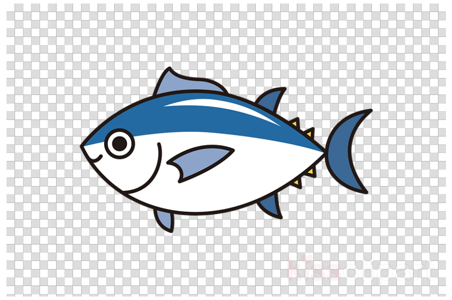
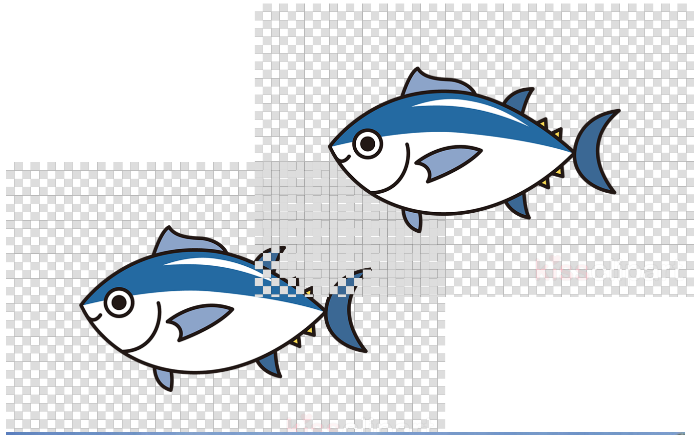


SAFEWAY

shutterstock.com • 298428176

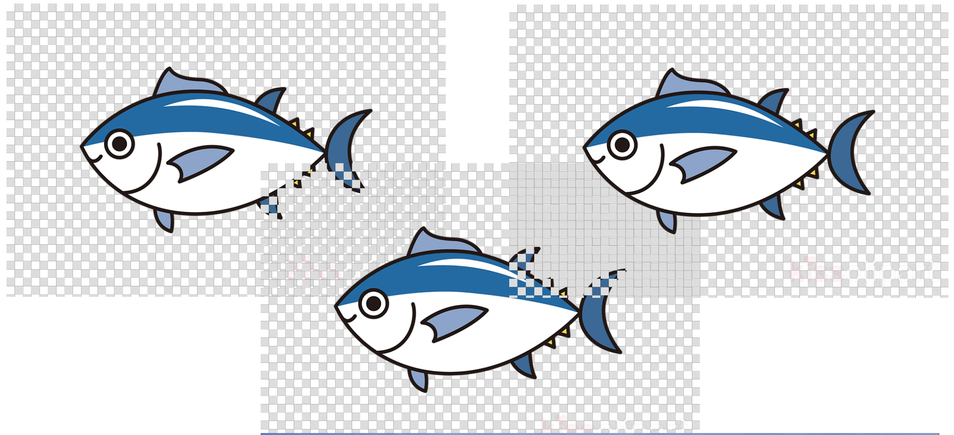
SAFEWAY

shutterstock.com • 298428176

SAFEWAY

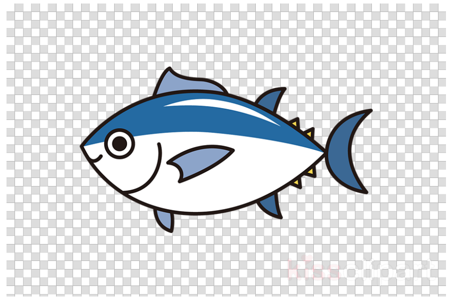
shutterstock.com • 298428176

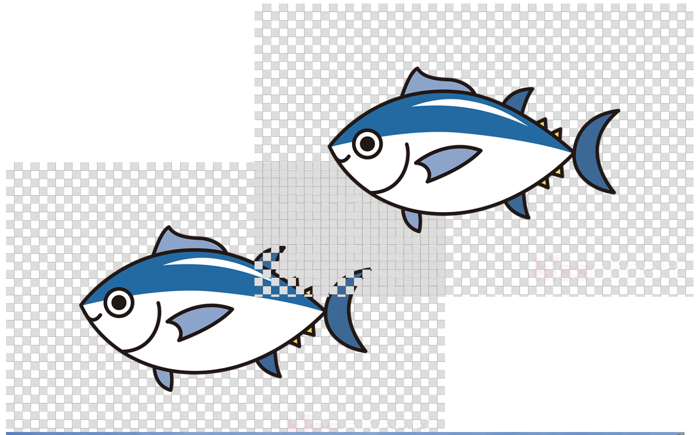


SAFEWAY

shutterstock.com • 298428176

shutterstock.com • 298428176





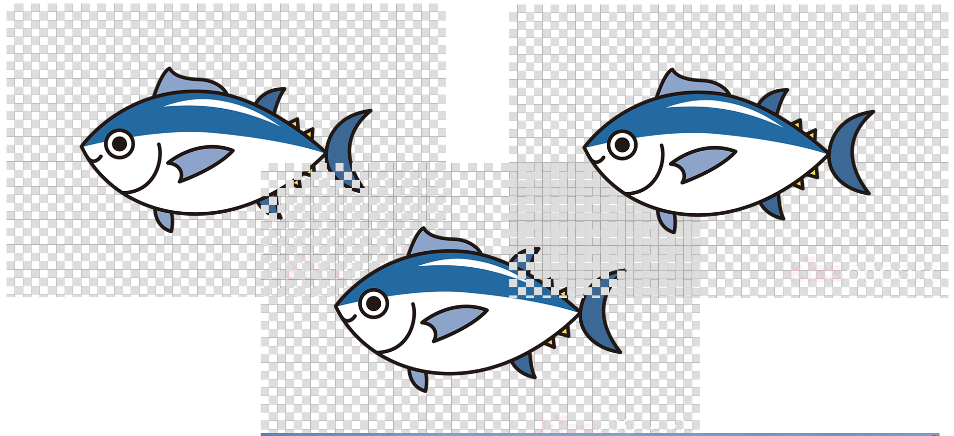
SAFEWAY

## Solution to Mercy’s traveling problem

If we know Mercy is going to keep eating tuna . . . Why not buy a bunch during a single trip and save them all somewhere closer than the store?

Let’s get Mercy a refrigerator!

shutterstock.com • 298428176

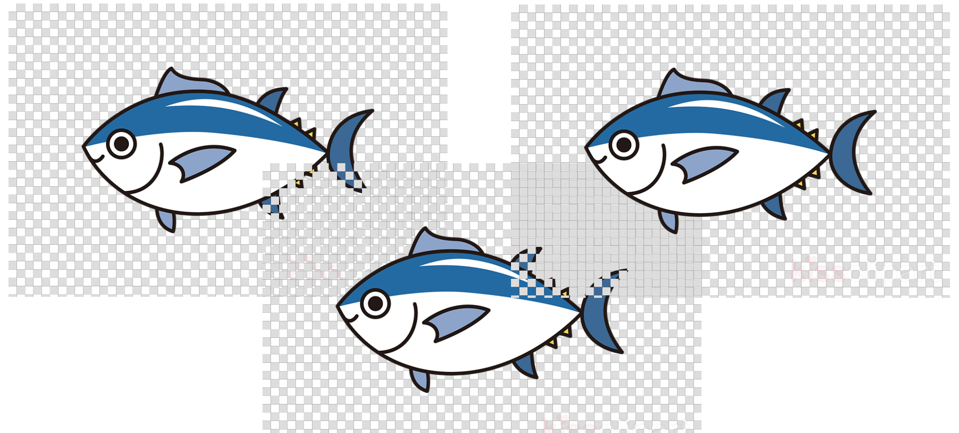


SAFEWAY

shutterstock.com • 298428176

SAFEWAY

shutterstock.com • 298428176



SAFEWAY

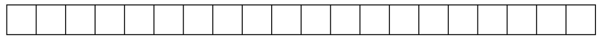
# Before

CPU

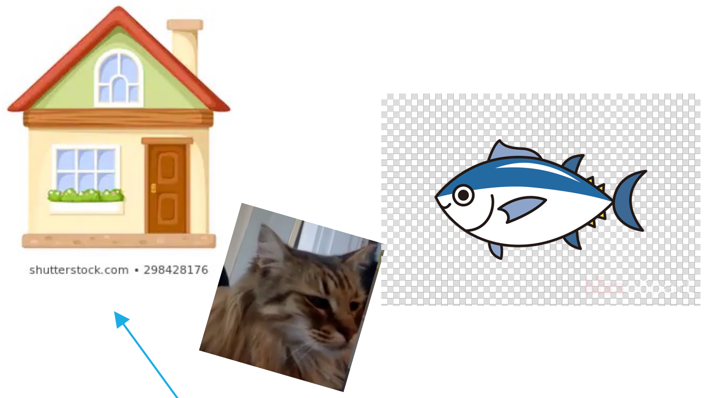
CPU – kind of like the home / brain of your computer. Pretty much all computation is done here and data needs to move here to do anything significant with it (math, if checks, normal statement execution).

Data travels between RAM and the CPU, but it's slow

RAM



# After



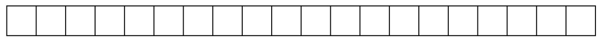
Bring a bunch of food back when you go all the way to the store

CPU

Cache!

Bring a bunch of data back when you go all the way to RAM

RAM



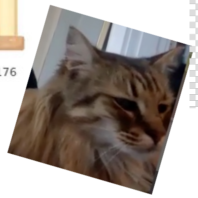
## Cache

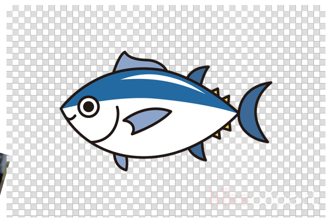
- Rough definition: a place to store some memory that's smaller and closer to the CPU compared to RAM. Because caches are closer to the CPU (where your data generally needs to go to be computed / modified / acted on) getting data from cache to CPU is a lot quicker than from RAM to CPU. This means we love when the data we want to access is conveniently in the cache.

- Generally we always store some data here in hopes that it will be used in the future and that we save ourselves the distance / time it takes to go to RAM.

- Analogy from earlier: The refrigerator (a cache) in your house to store food closer to you than the store. Walking to your fridge is much quicker than walking to the store!

# After





Bring a bunch of food back when you go all the way to the store

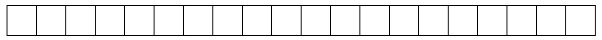
CPU

This is a big idea!

Cache!

Bring a bunch of data back when you go all the way to RAM

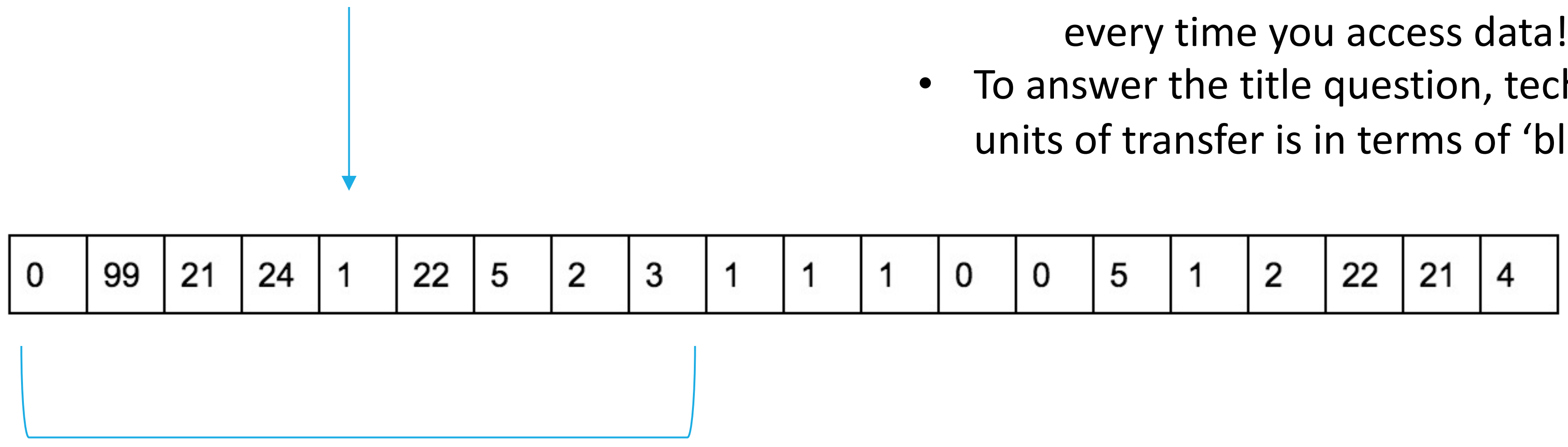
RAM



# How is a bunch of memory taken from RAM?

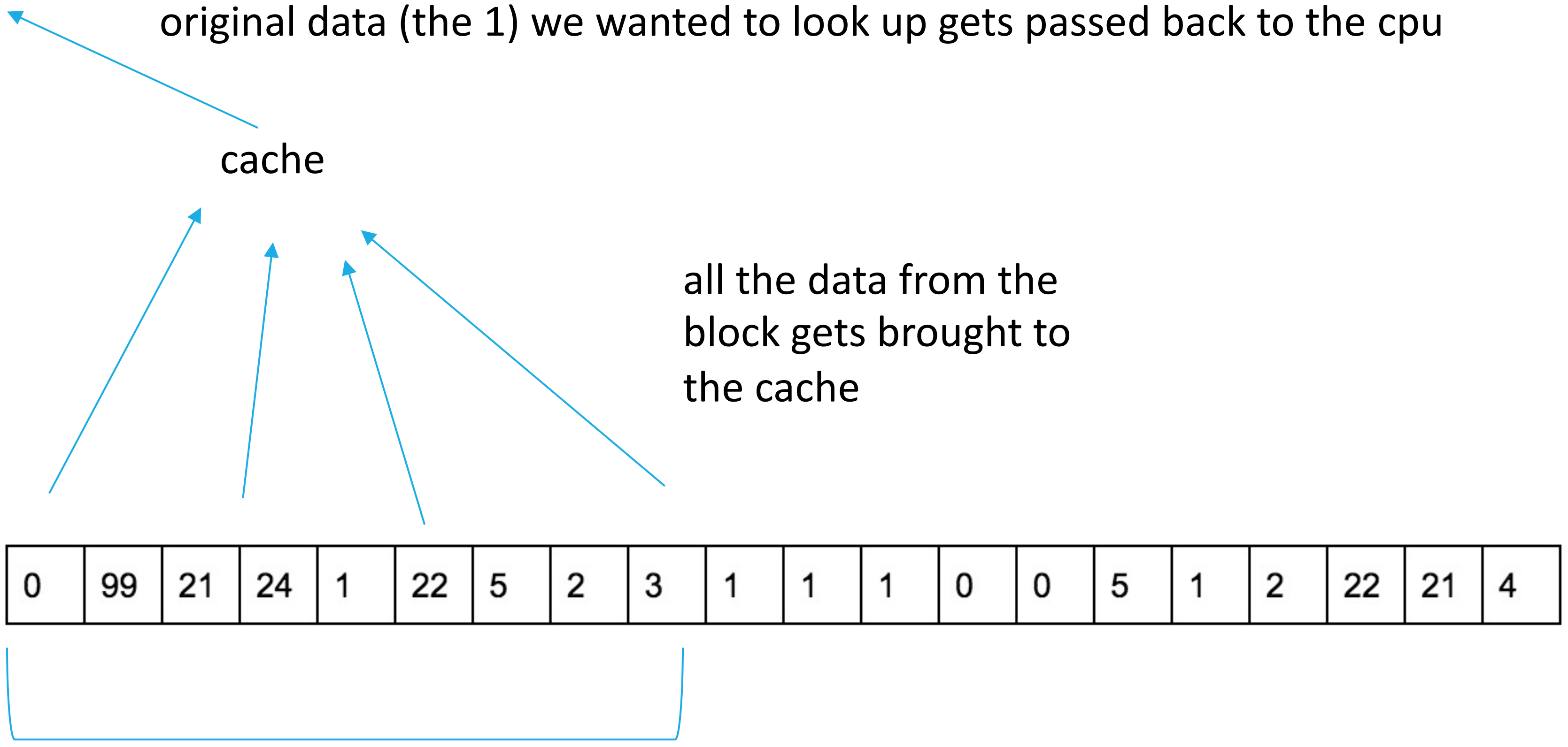
This is a big idea (continued)!

- Imagine you want to retrieve the 1 at index 4 in RAM
- Your computer is smart enough to know to grab some of the surrounding data because computer designers think that it's reasonably likely you'll want to access that data too.
    - (You don't have to do anything in your code for this to happen – it happens automatically every time you access data!)
- To answer the title question, technically the term / units of transfer is in terms of 'blocks'.



# How is a bunch of memory taken from RAM? (continued)

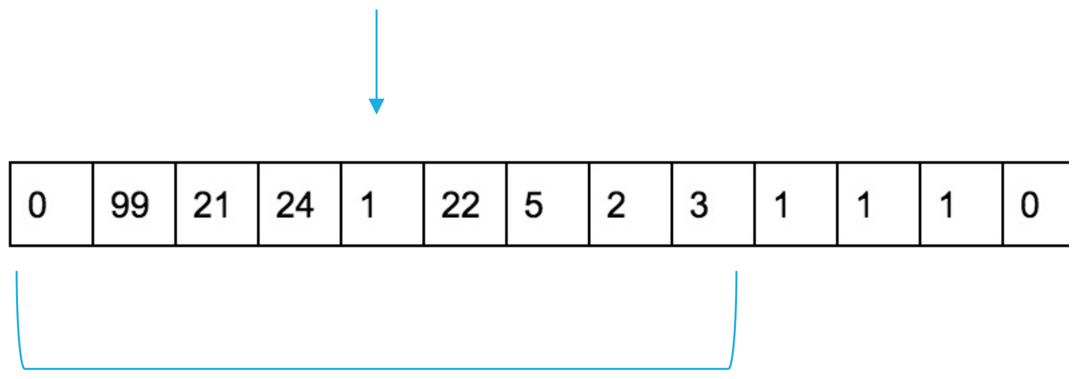
CPU


original data (the 1) we wanted to look up gets passed back to the cpu
cache
all the data from the block gets brought to the cache
0
99
21
24
1
22
5
2
3
1
1
1
0
0
5
1
2
22
21
4

# How does this pattern of memory grabbing affect our programs?

- This should have a major impact on programming with arrays. Say we access an index of an array that is stored in RAM. Because we grab a whole bunch of contiguous memory even when we just access one index in RAM, we'll probably be grabbing other nearby parts of our array and storing that in our cache for quick access later.

Imagine that the below memory is just an entire array of length 13, with some data in it.



Just by accessing one element we bring the nearby elements back with us to the cache. In this case, it's almost all of the array!

## Leveraging Temporal Locality

When looking up address in “slow layer”

Once we load something into RAM or cache, keep it around or a while

- But these layers are smaller
  - When do we “evict” memory to make room?

## Moving Memory

Amount of memory moved from **disk** to **RAM**

- Called a "**block**" or "**page**"
  - ≈4kb
  - Smallest unit of data on disk

Amount of memory moved from **RAM** to **Cache**

- called a "**cache line**"
  - ≈64 bytes

Operating System is the Memory Boss

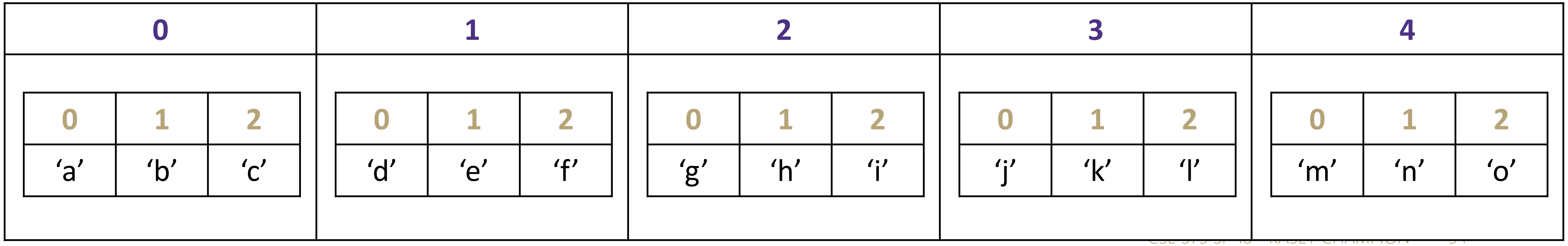
- controls page and cache line size
- decides when to move data to cache or evict

# Thought Experiment

```
public int sum1(int n, int m, int[][] table) {
   int output = 0;
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         output += table[i][j];
      }
   }
   return output;
}
```

```
public int sum2(int n, int m, int[][] table) {
   int output = 0;
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
         output += table[j][i];
      }
   }
   return output;
}
```

Why does sum1 run so much faster than sum2?
sum1 takes advantage of spatial and temporal locality



## Java and Memory

What happens when you use the “**new**” keyword in Java?

- Your program asks the **J**ava **V**irtual **M**achine for more memory from the “heap”
  - Pile of recently used memory
- If necessary the JVM asks Operating System for more memory
  - Hardware can only allocate in units of page
  - If you want 100 bytes you get 4kb
  - Each page is contiguous

What happens when you create a new array?
- Program asks JVM for one long, contiguous chunk of memory

What happens when you create a new object?
- Program asks the JVM for any random place in memory

What happens when you read an array index?
- Program asks JVM for the address, JVM hands off to OS
- OS checks the L1 caches, the L2 caches then RAM then disk to find it
- If data is found, OS loads it into caches to speed up future lookups

What happens when we open and read data from a file?
- Files are always stored on disk, must make a disk access

# Array v Linked List

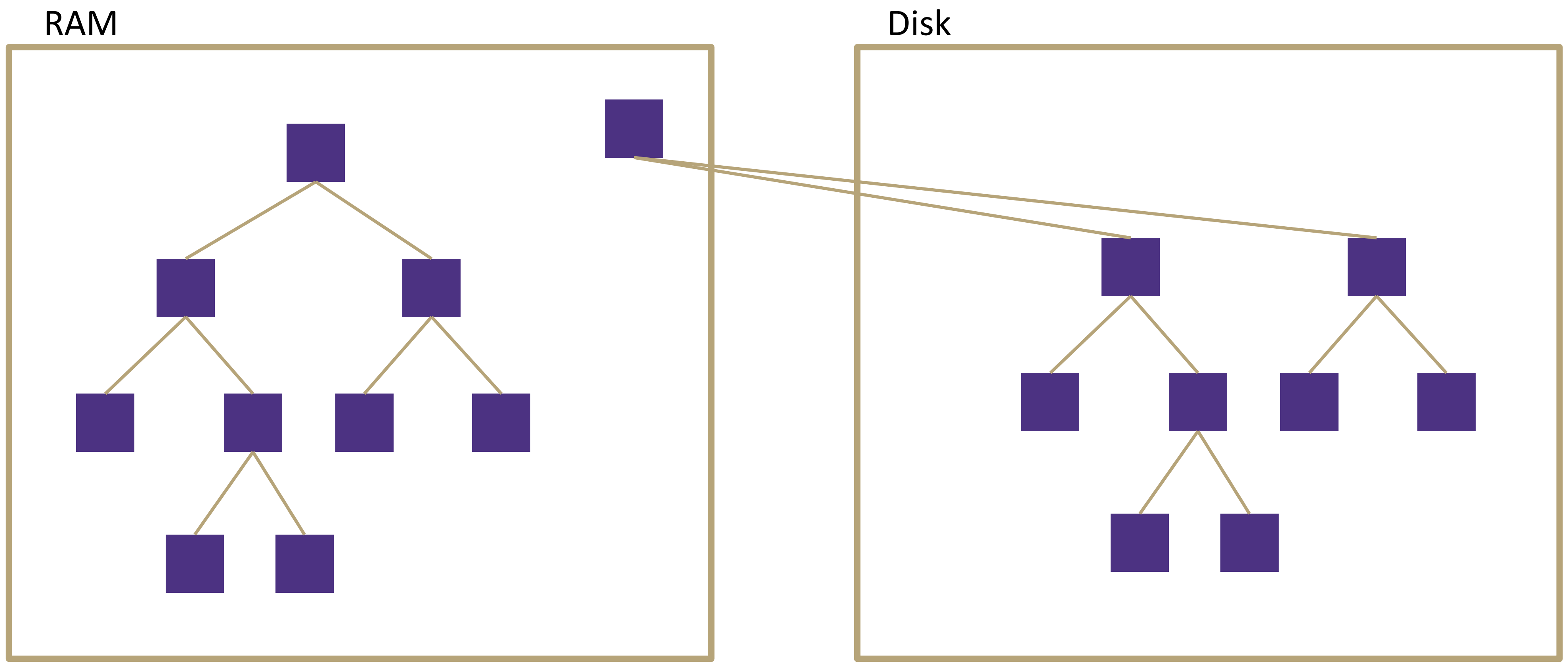
Is iterating over an ArrayList faster than iterating over a LinkedList?

Answer:

LinkedList nodes can be stored in memory, which means the don't have spatial locality. The ArrayList is more likely to be stored in contiguous regions of memory, so it should be quicker to access based on how the OS will load the data into our different memory layers.

## Thought Experiment

Suppose we have an AVL tree of height 50. What is the **best** case scenario for number of disk accesses? What is the **worst** case?



# Maximizing Disk Access Effort

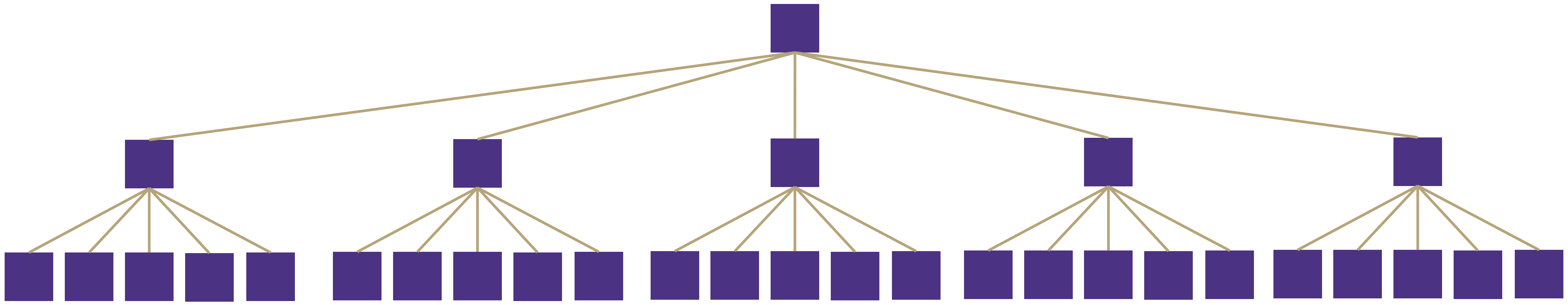
Instead of each node having 2 children, let it have M children.

- Each node contains a sorted array of children

Pick a size M so that fills an entire page of disk data

Assuming the M-ary search tree is balanced, what is its height? **$\log_m(n)$**

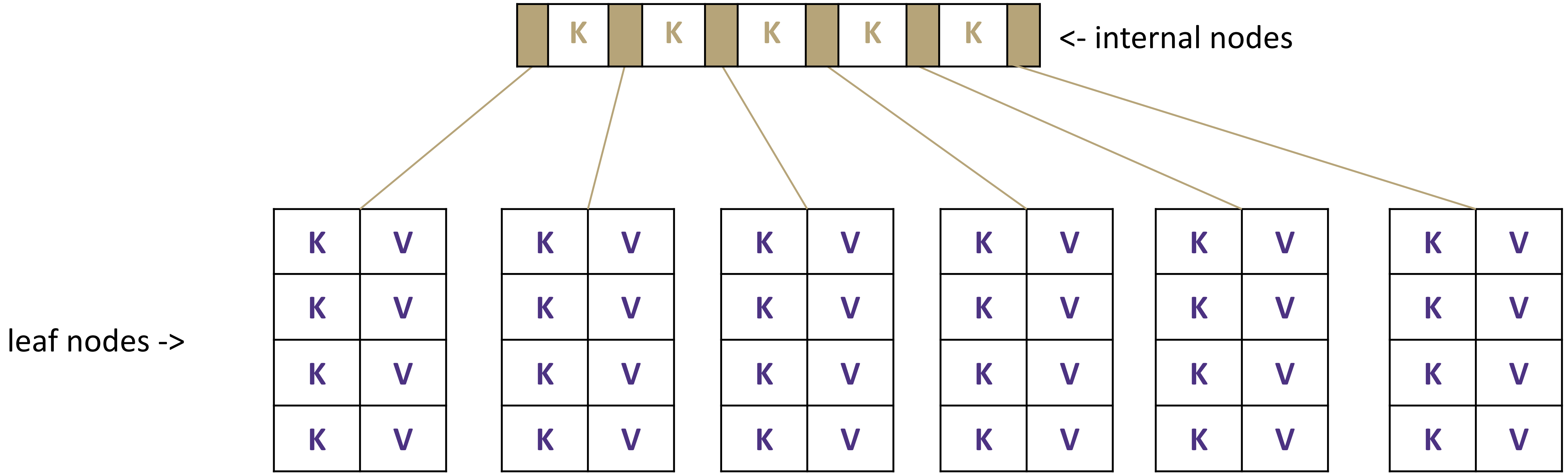
What is the worst case runtime of get() for this tree? **$\log_2(m)$ to pick a child**
**$\log_m(n) * \log_2(m)$ to find node**



# Maximizing Disk Access Effort

If each child is at a different location in disk memory – expensive!

What if we construct a tree that stores keys together in branch nodes, all the values in leaf nodes



## B Trees

Has 3 invariants that define it

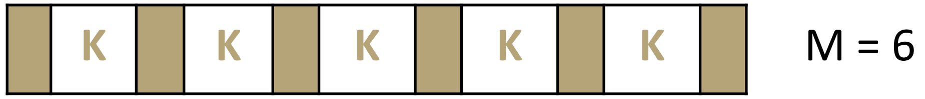
1. B-trees must have two different types of nodes: internal nodes and leaf nodes

2. B-trees must have an organized set of keys and pointers at each internal node

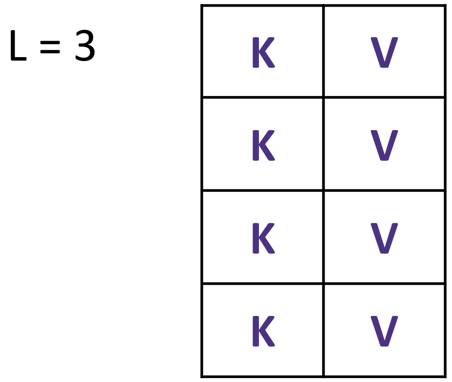
3. B-trees must start with a leaf node, then as more nodes are added they must stay at least half full

# Node Invariant

Internal nodes contain M pointers to children and M-1 **sorted** keys

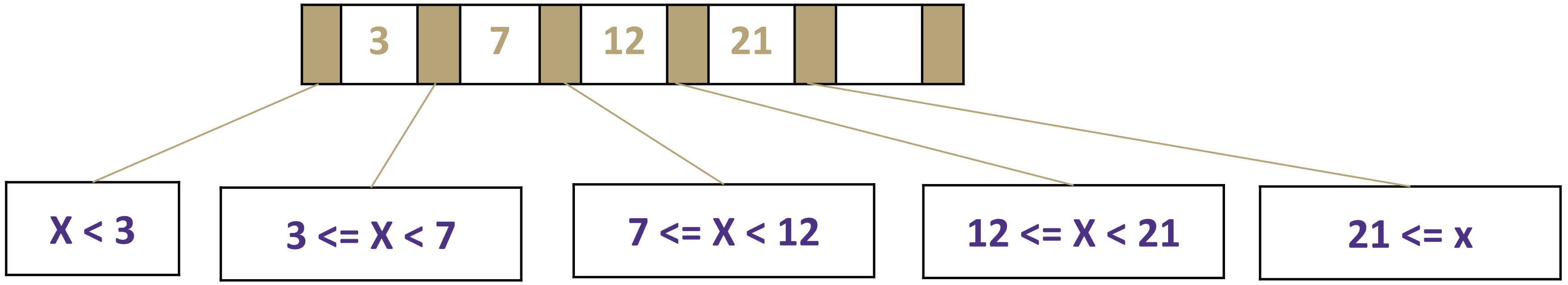


A leaf node contains L key-value pairs, sorted by key



## Order Invariant

For any given key k, all subtrees to the left may only contain keys x that satisfy x < k. All subtrees to the right may only contain keys x that satisfy k >= x



# Structure Invariant

If n <= L, the root node is a leaf

| | |
|---|---|
| K | V |
| K | V |
| K | V |
| K | V |

When n > L the root node **must** be an internal node containing 2 to M children

All other internal nodes must have M/2 to M children

All leaf nodes must have L/2 to L children

All nodes must be at least **half-full** The root is the only exception, which can have as few as 2 children

- Helps maintain balance
- Requiring more than 2 children prevents degenerate Linked List trees

# B-Trees

Has 3 invariants that define it

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
   - Leaf Node contains L key-value pairs, sorted by key.

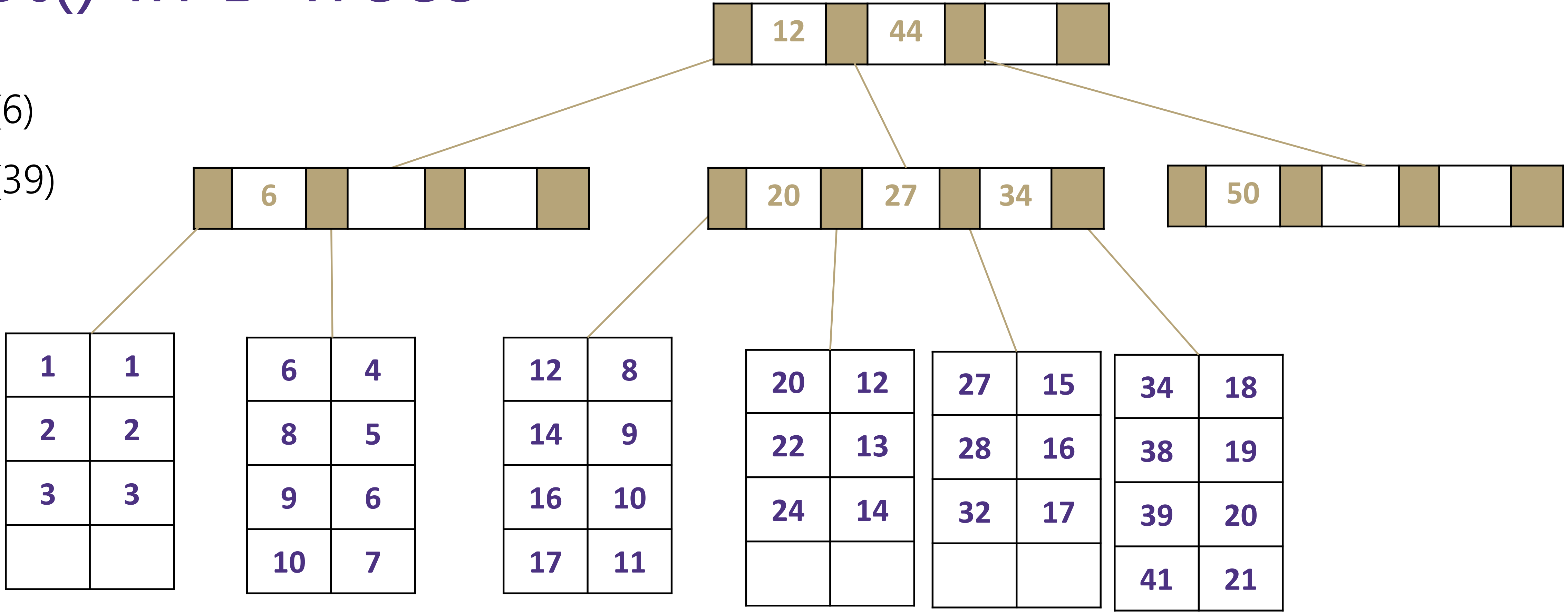
2. B-trees order invariant
   - For any given key k, all subtrees to the left may only contain keys that satisfy x < k
   - All subtrees to the right may only contain keys x that satisfy k >= x

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

# get() in B Trees

get(6)

get(39)



Worst case run time = $\log_m(n)\log_2(m)$

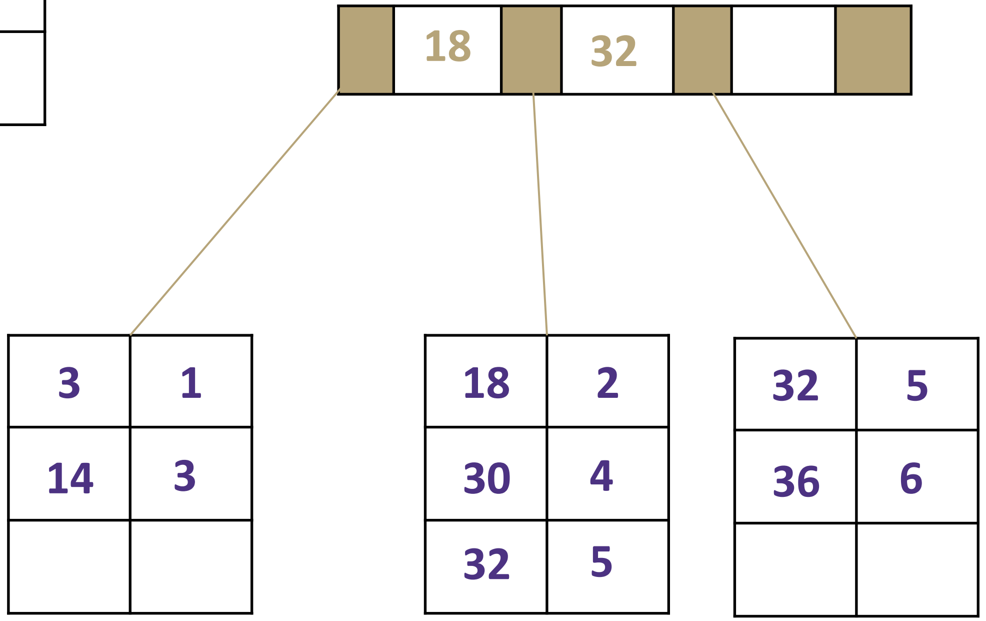
Disk accesses = $\log_m(n)$ = height of tree

# put() in B Trees

Suppose we have an empty B-tree where M = 3 and L = 3. Try inserting 3, 18, 14, 30, 32, 36

| | |
|---|---|
| 3 | 1 |
| 18 | 2 |
| 14 | 3 |

| | |
|---|---|
| 3 | 1 |
| 14 | 3 |
| 18 | 2 |

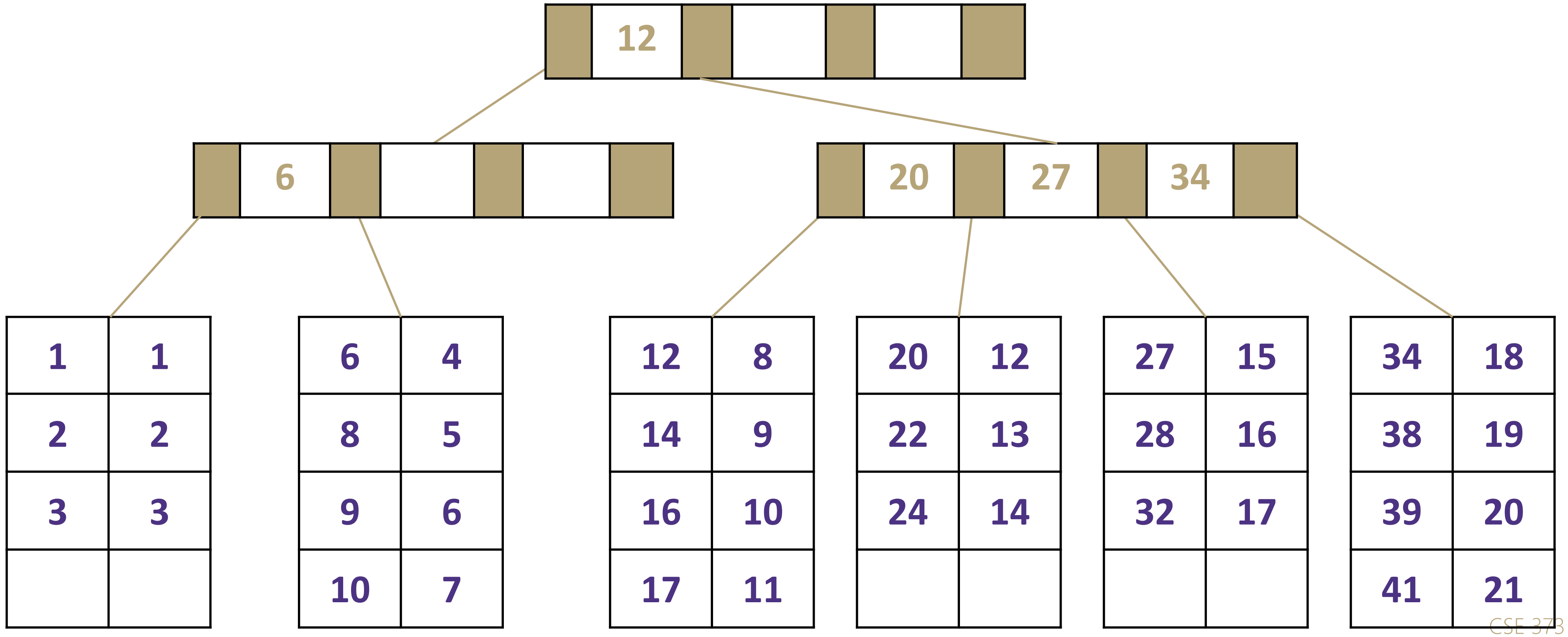


# Warm Up

What operations would occur in what order if a call of get(24) was called on this b-tree?

What is the M for this tree? What is the L?

If Binary Search is used to find which child to follow from an internal node, what is the runtime for this get operation?



# *Review:* B-Trees

Has 3 invariants that define it

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
   - 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

# Put() for B-Trees

Build a new b-tree where M = 3 and L = 3.

Insert (3,1), (18,2), (14,3), (30,4) where (k,v)

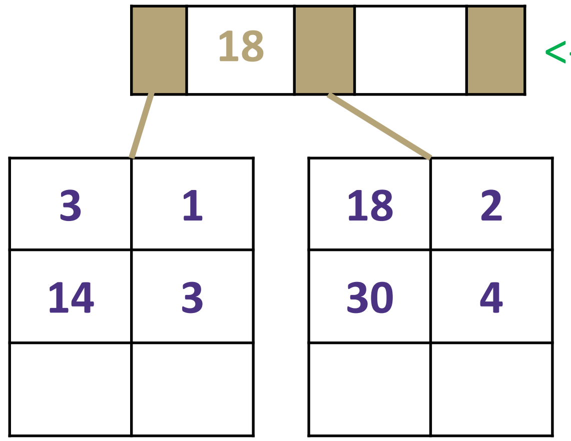
When n <= L b-tree root is a leaf node

wrong ->

| | |
|---|---|
| 3 | 1 |
| 18 | 2 |
| 14 | 3 |

No space for (30,4) ->split the node

Create two new leafs that each hold ½ the values and create a new internal node

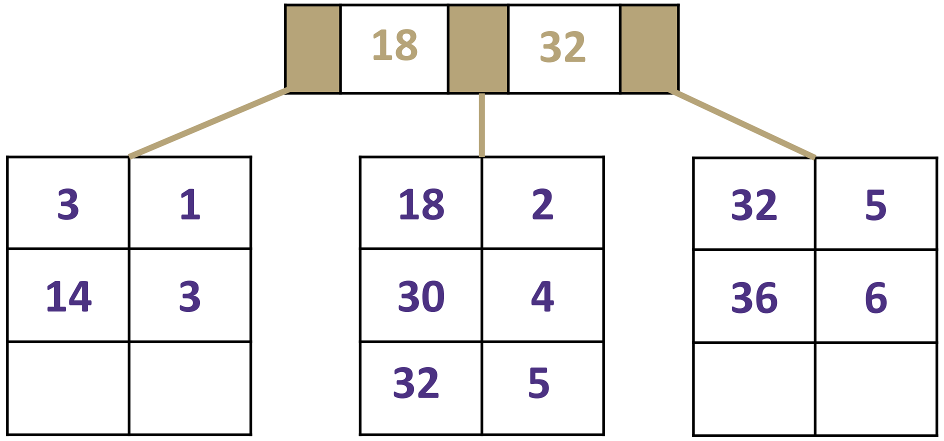


<- use smallest value in larger subset as sign post

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

## You try!

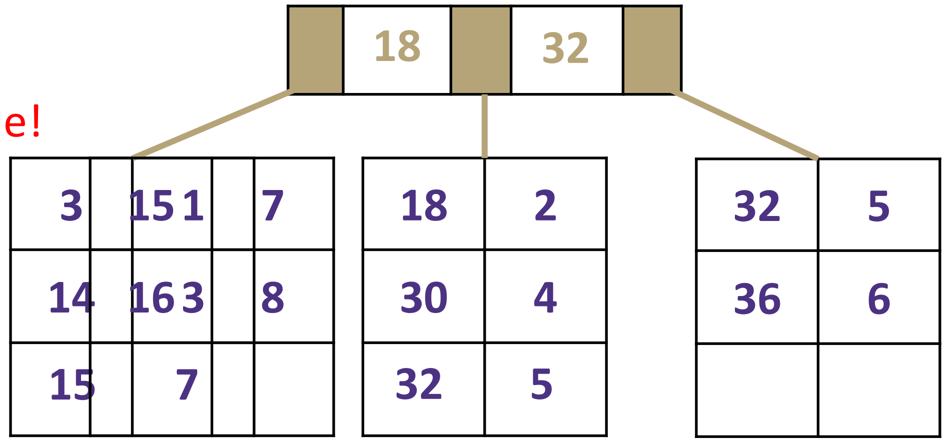
Try inserting (32, 5) and (36, 6) into the following tree



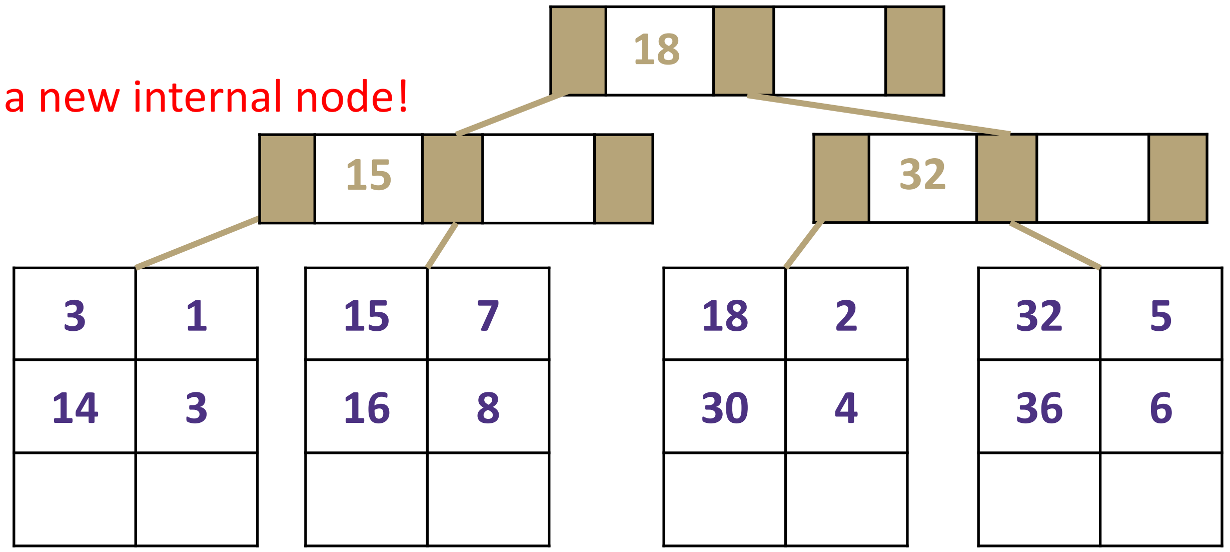
# Splitting internal nodes

Try inserting (15, 7) and (16, 8) into our existing tree

Make a new internal node!



Make a new internal node!



# B-tree Run Time

Time to find correct leaf **Height = $\log_m(n)\log_2(m)$ = tree traversal time**

Time to insert into leaf **$\Theta(L)$**

Time to split leaf **$\Theta(L)$**

Time to split leaf's parent internal node **$\Theta(M)$**

Number of internal nodes we might have to split **$\Theta(\log_m(n))$**

All up worst case runtime: **$\Theta(L + M\log_m(n))$**