Lecture 13: Computer Memory

CSE 373 Data Structures and Algorithms

Administrivia

Sorry no office hours this afternoon :/

Midterm review session Monday 6-8pm Sieg 134 (hopefully)

Written HW posted later today – individual assignment

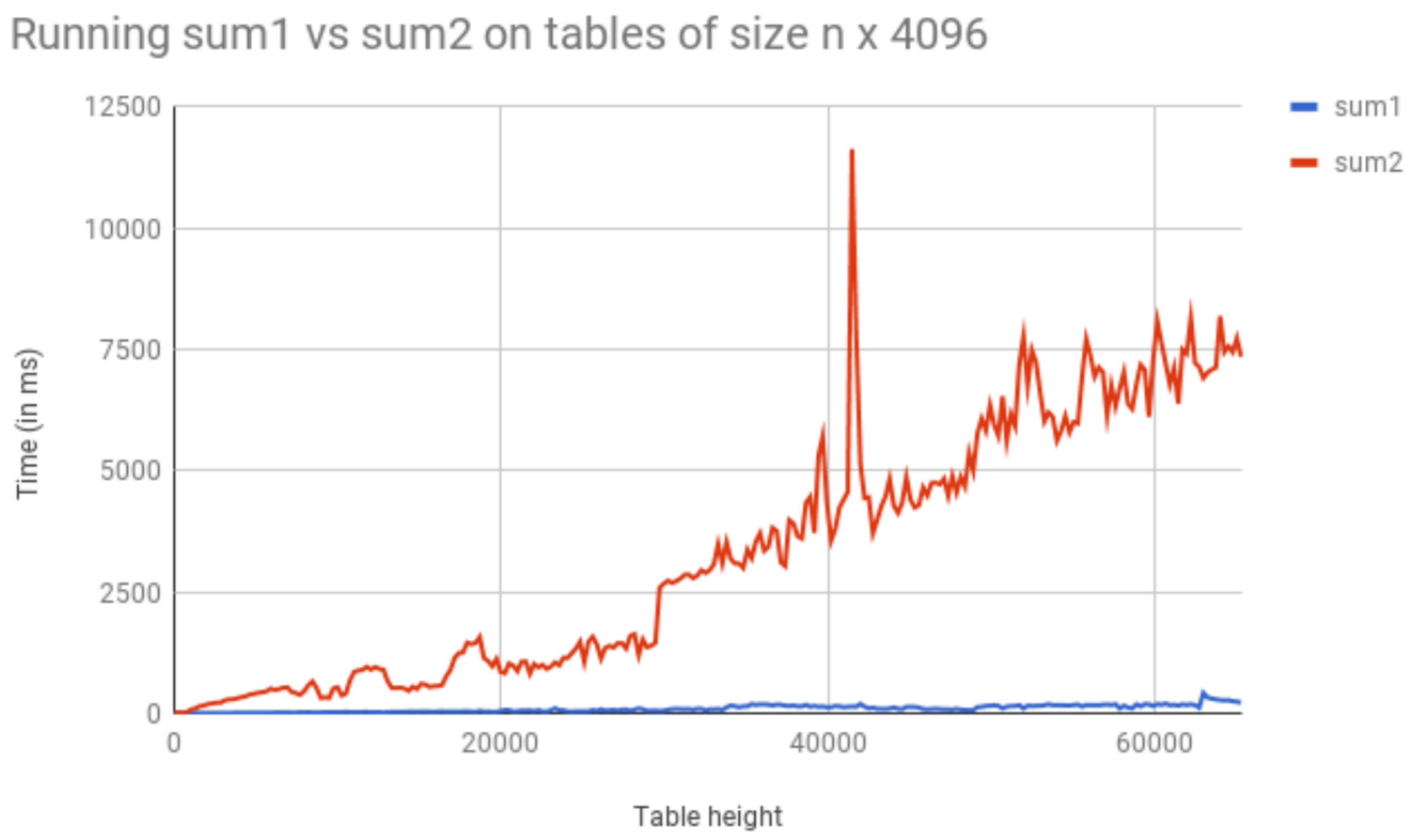
Thought experiment

```
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;
}
```

What do these two methods do?
What is the big-Θ
Θ(n*m)

Warm Up


Running sum1 vs sum2 on tables of size n x 4096
12500
10000
7500
5000
2500
0
0
20000
40000
60000
Time (in ms)
Table height
sum1
sum2

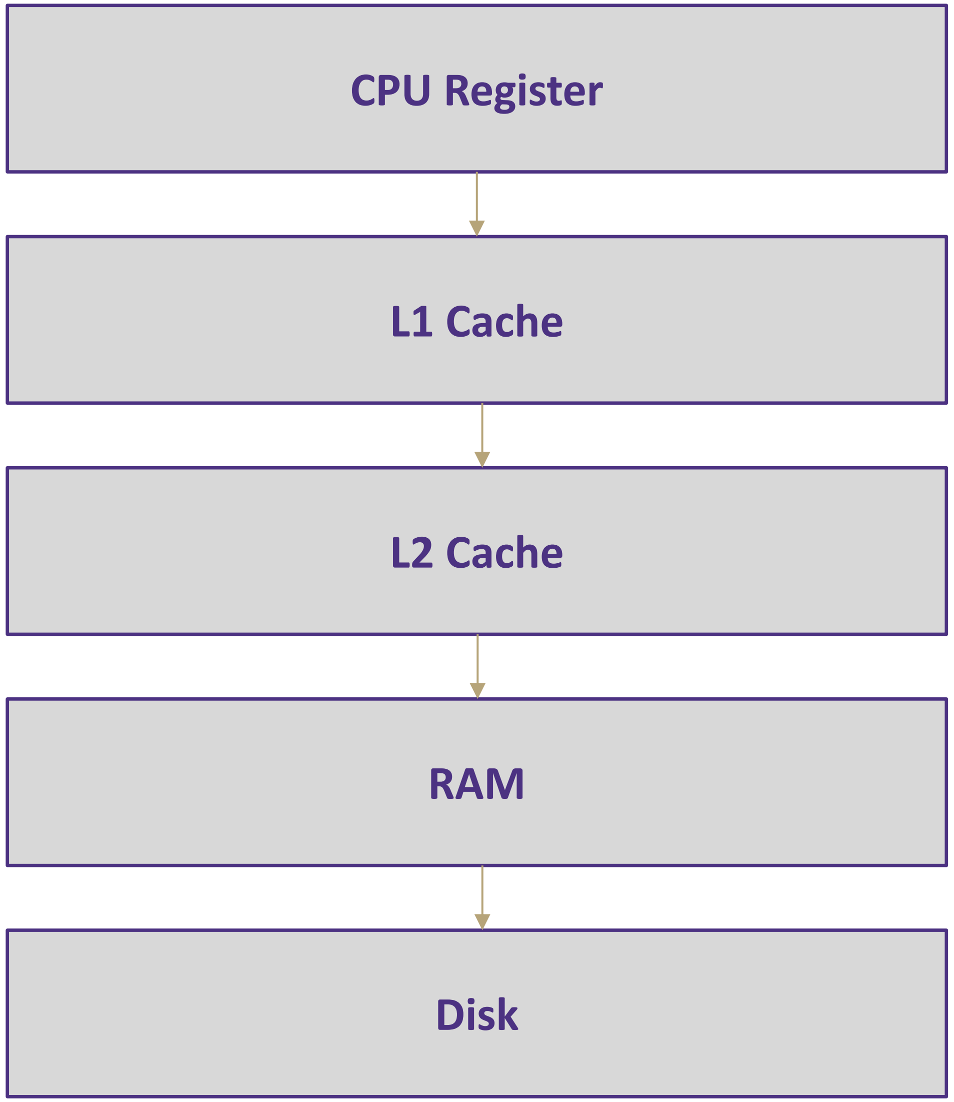
Incorrect Assumptions

Accessing memory is a quick and constant-time operation Lies!

Sometimes accessing memory is cheaper and easier than at other times

Sometimes accessing memory is very slow

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

Review: Binary, Bits and Bytes

binary

A base-2 system of representing numbers using only 1s and 0s

- vs decimal, base 10, which has 9 symbols

bit

The smallest unit of computer memory represented as a single binary value either 0 or 1

byte

The most commonly referred to unit of memory, a grouping of 8 bits

Can represent 265 different numbers (28)

1 Kilobyte = 1 thousand bytes (kb)

1 Megabyte = 1 million bytes (mb)

1 Gigabyte = 1 billion bytes (gb)

Decimal	Decimal Break Down	Binary	Binary Break Down
0	$(0 * 10^0)$	0	$(0 * 2^0)$
1	$(1 * 10^0)$	1	$(1 * 2^0)$
10	$(1 * 10^1) + (0 * 10^0)$	1010	$(1 * 2^3) + (0 * 2^2) + (1 * 2^1) + (0 * 2^0)$
12	$(1 * 10^1) + (2 * 10^0)$	1100	$(1 * 2^3) + (1 * 2^2) + (0 * 2^1) + (0 * 2^0)$
127	$(1 * 10^2) + (1 * 10^1) + (2 * 10^0)$	01111111	$(0 * 2^7) + (1 * 2^6) + (1 * 2^5) + (1 * 2^4)(1 * 2^3) + (1 * 2^2) + (1 * 2^1) + (1 * 2^0)$

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!

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

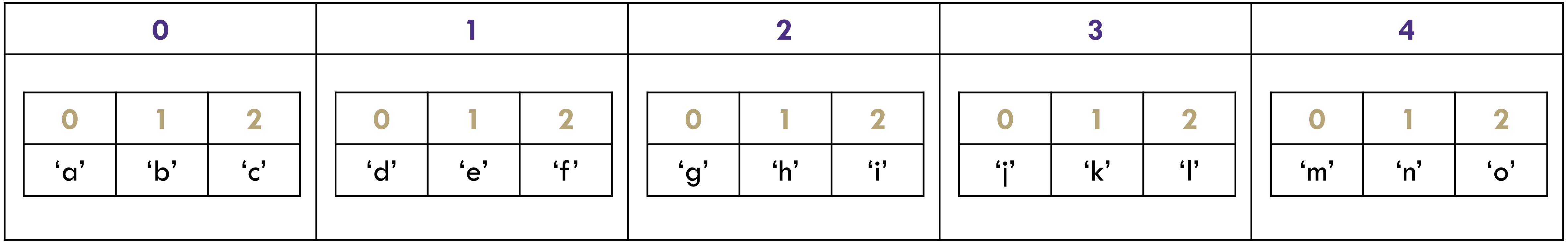
- controls page and cache line size
- decides when to move data to cache or evict

Warm Up

```
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

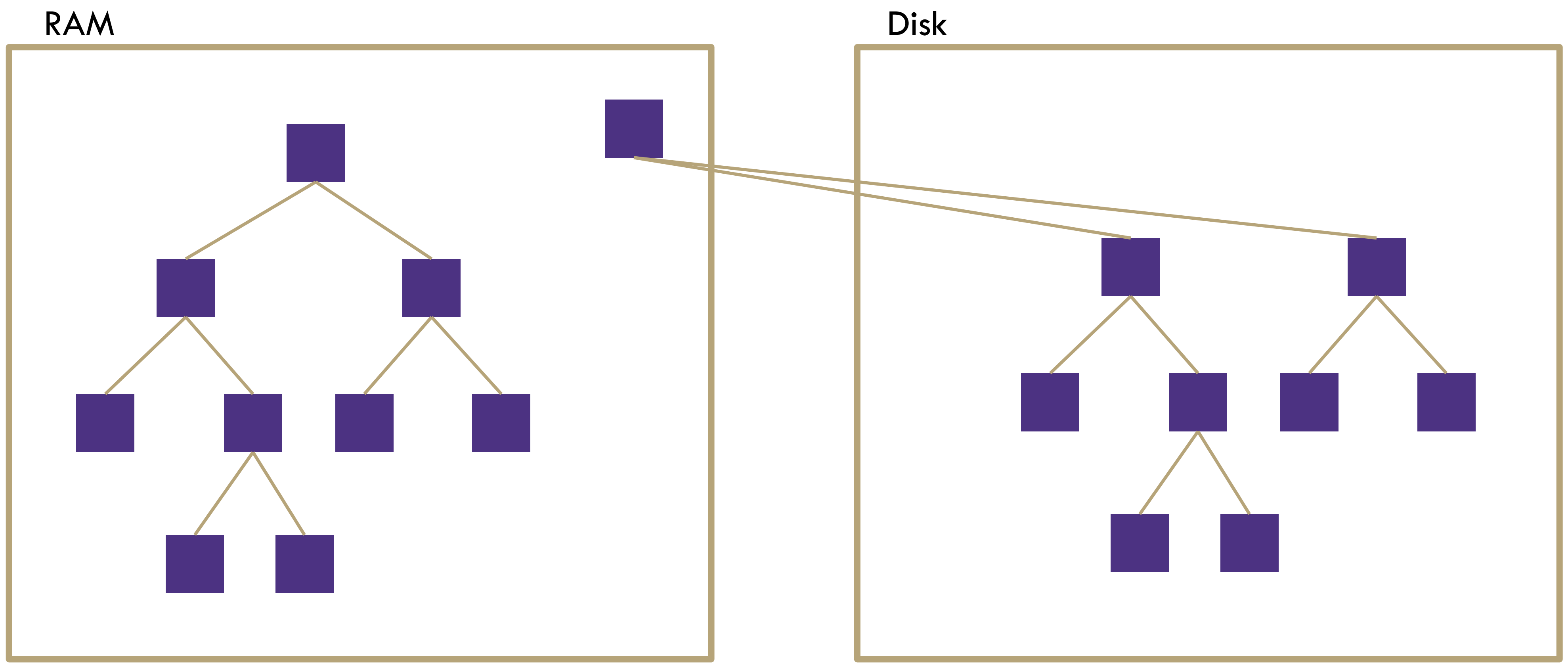
Is iterating over an ArrayList faster than iterating over a LinkedList?

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

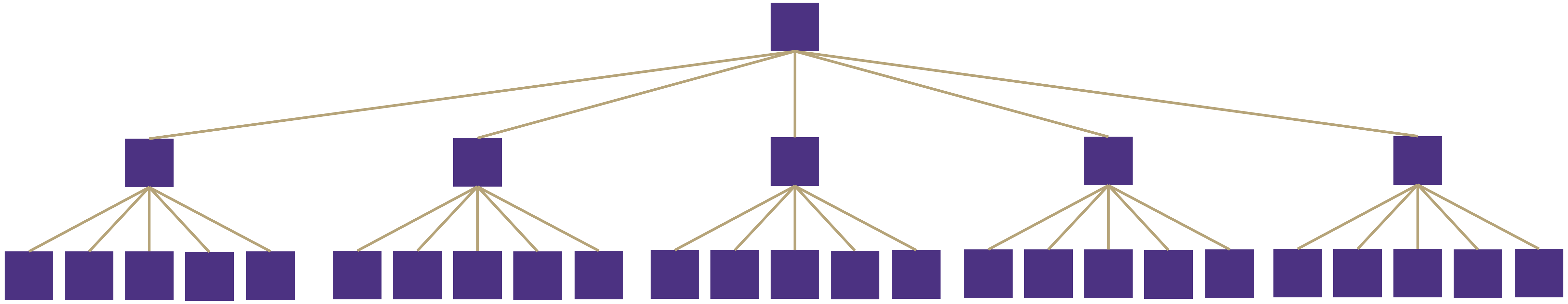
Pick a size M so that fills an entire page of disk data

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

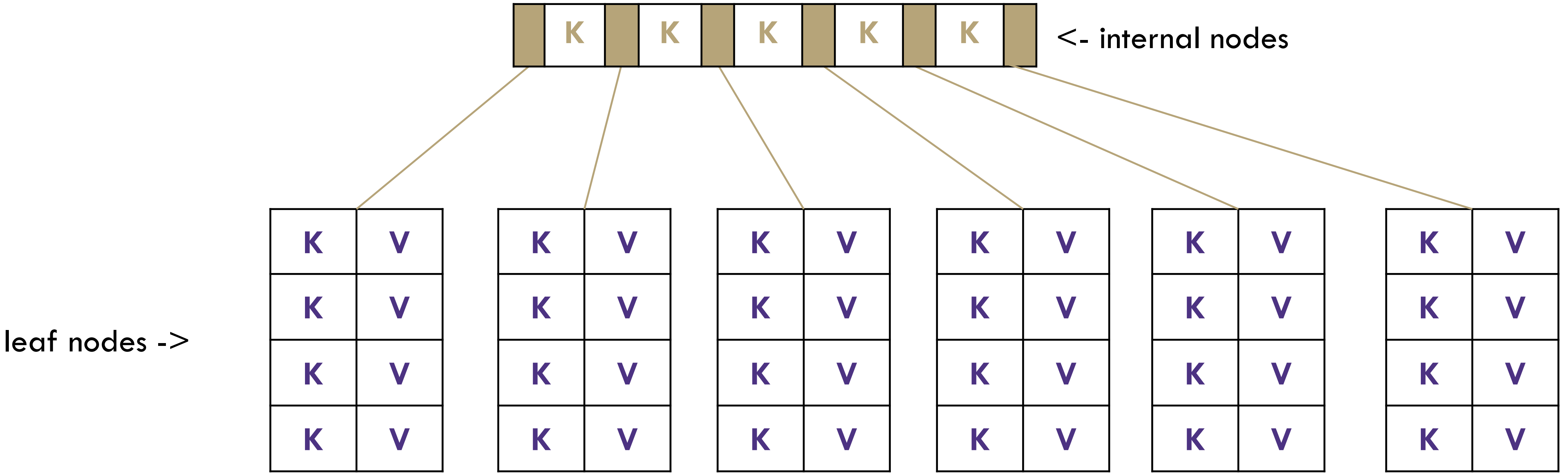
$\log_m(n) * \log_2(m)$ to find node



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



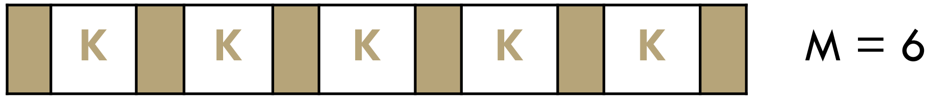
B Trees

Has 3 invariants that define it

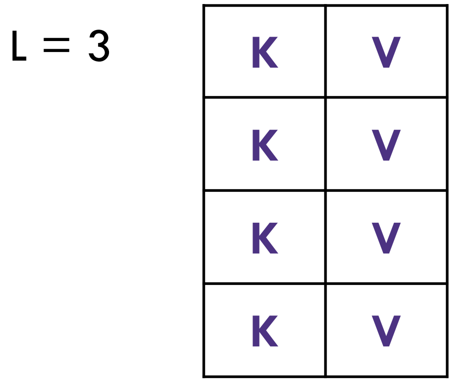
1. B-trees must have two different types of nodes: internal nodes and leaf nodes
2. B-trees must have an organized set of keys and pointers at each internal node
3. B-trees must start with a leaf node, then as more nodes are added they must stay at least half full

Node Invariant

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

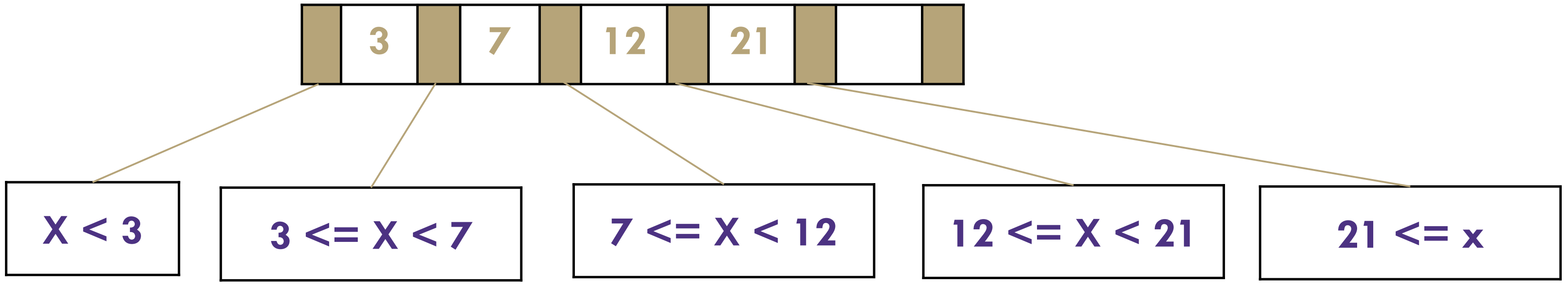


A leaf node contains L key-value pairs, sorted by key



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.

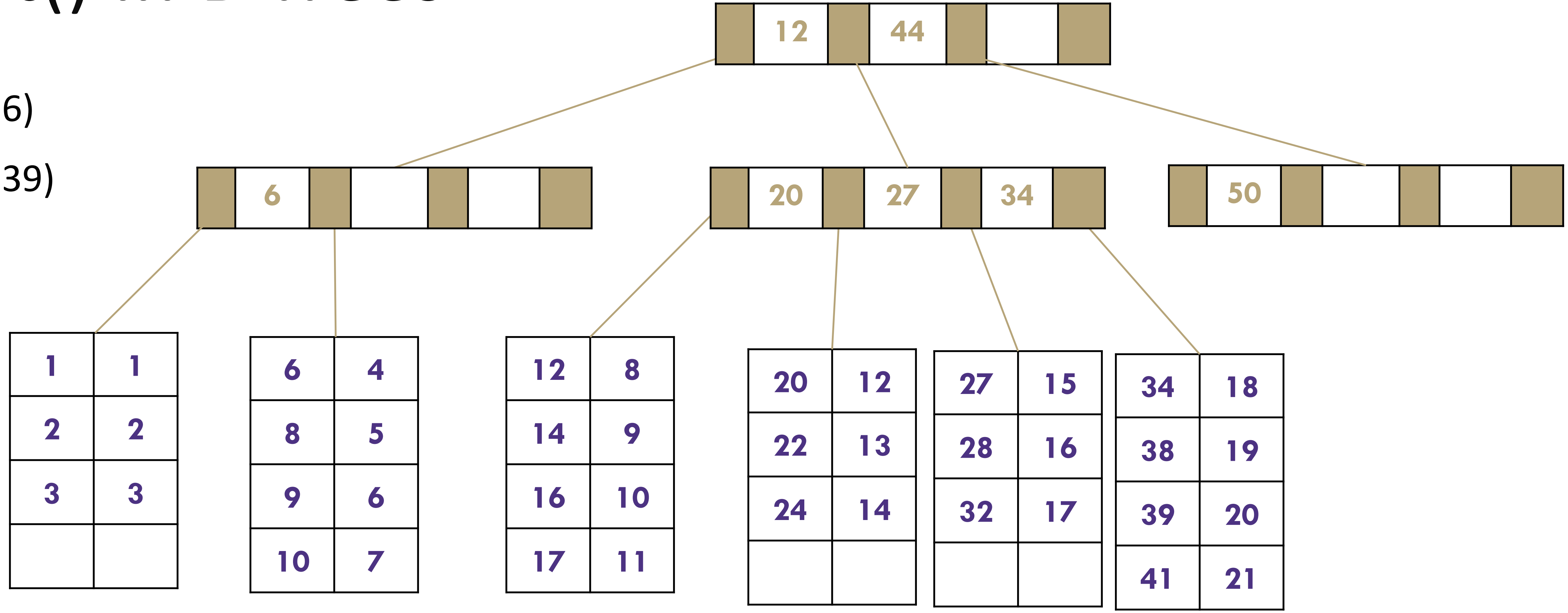
2. B-trees order invariant
   - For any given key k, all subtrees to the left may only contain keys that satisfy x < k
   - All subtrees to the right may only contain keys x that satisfy k >= x

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 = $\log_m(n)\log_2(m)$

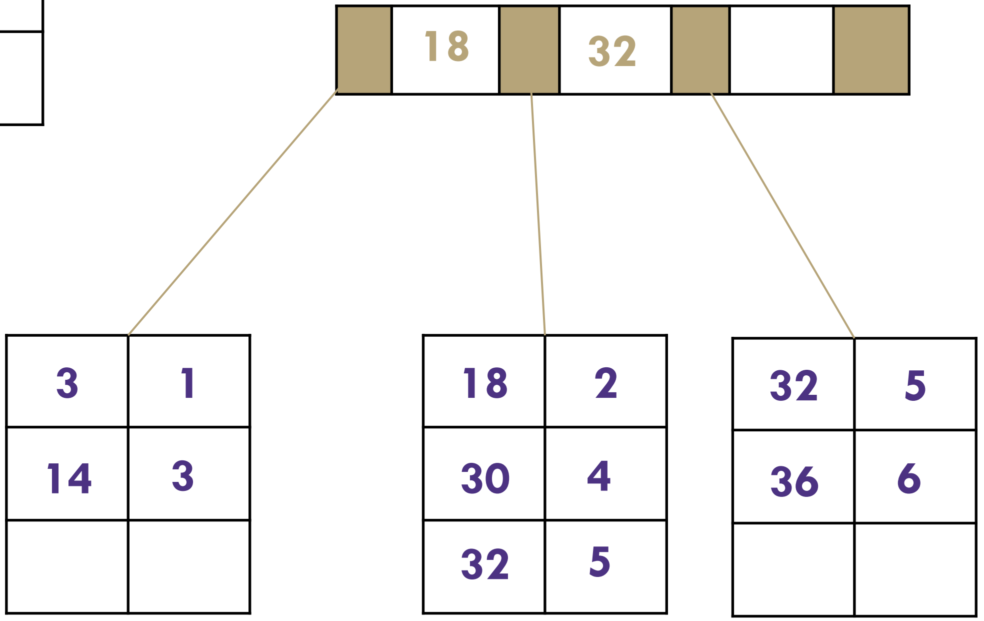
Disk accesses = $\log_m(n)$ = height of tree

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

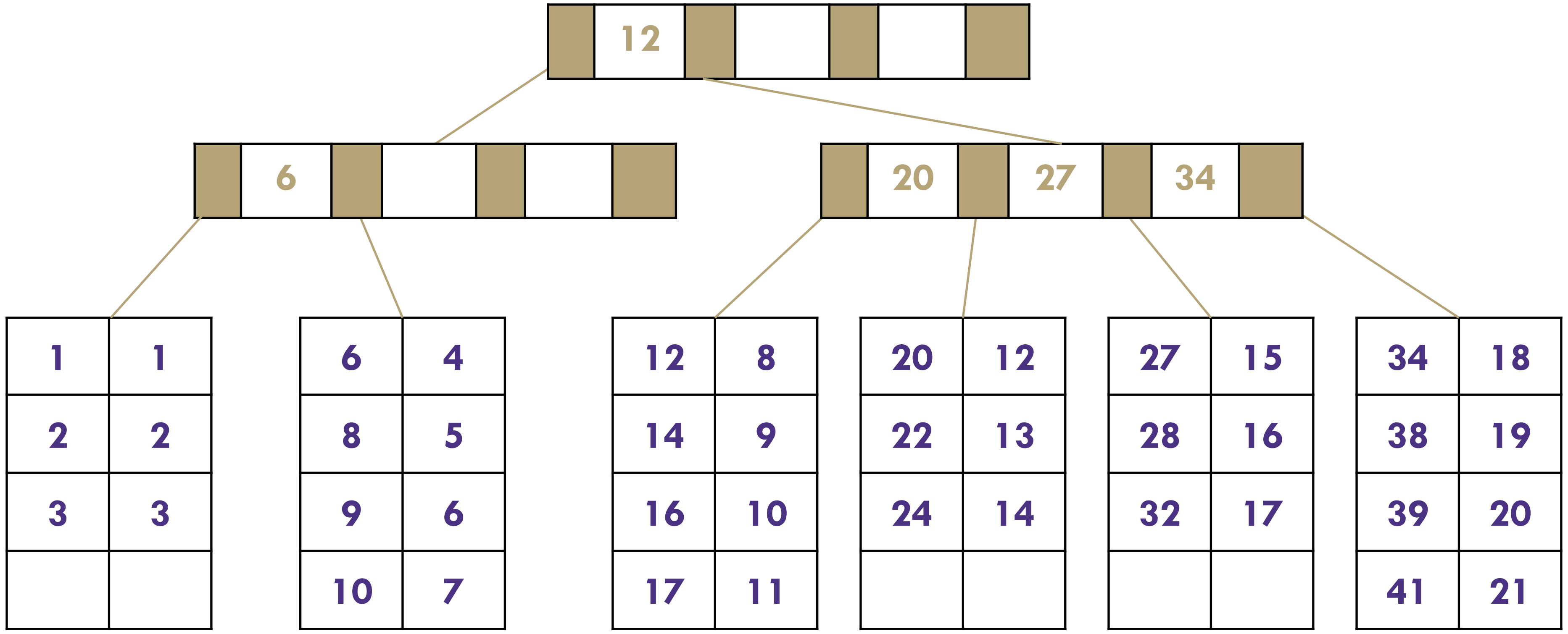


Warm Up

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)

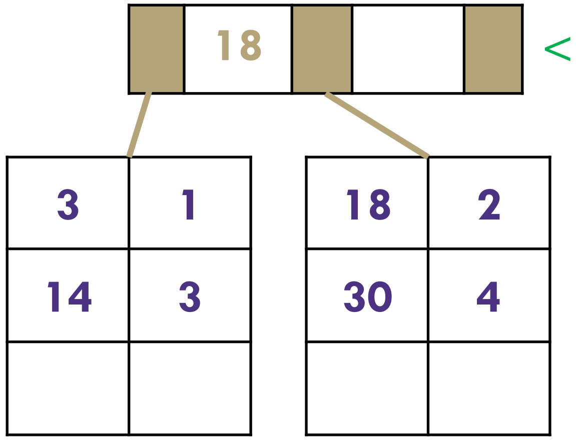
When n <= L b-tree root is a leaf node

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

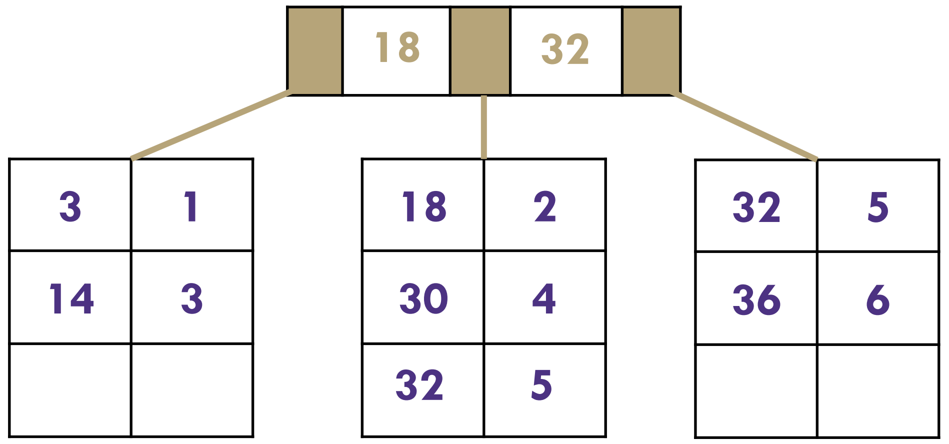
2. B-trees order invariant

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

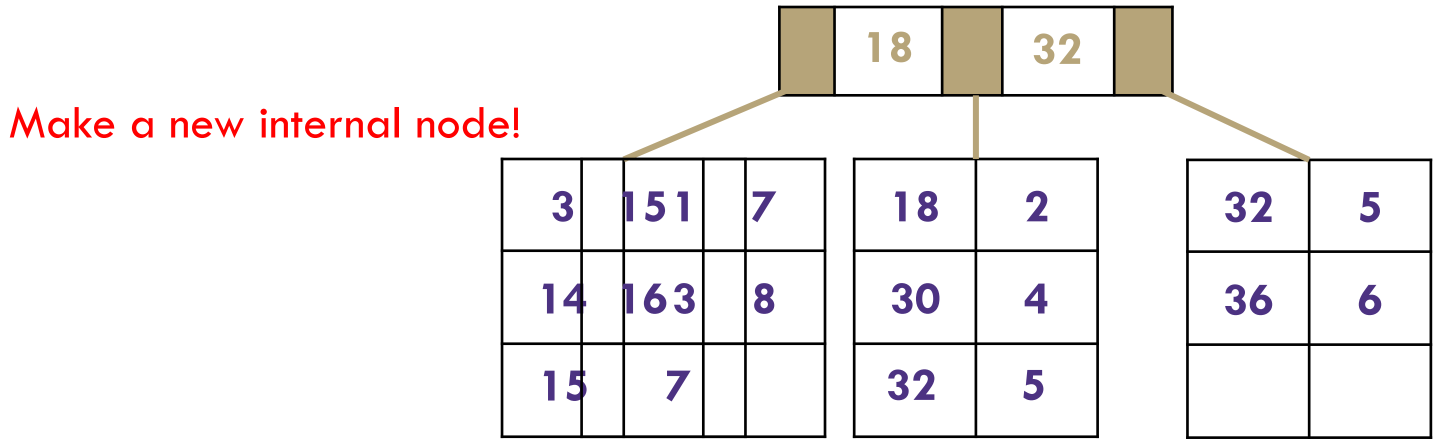
You try!

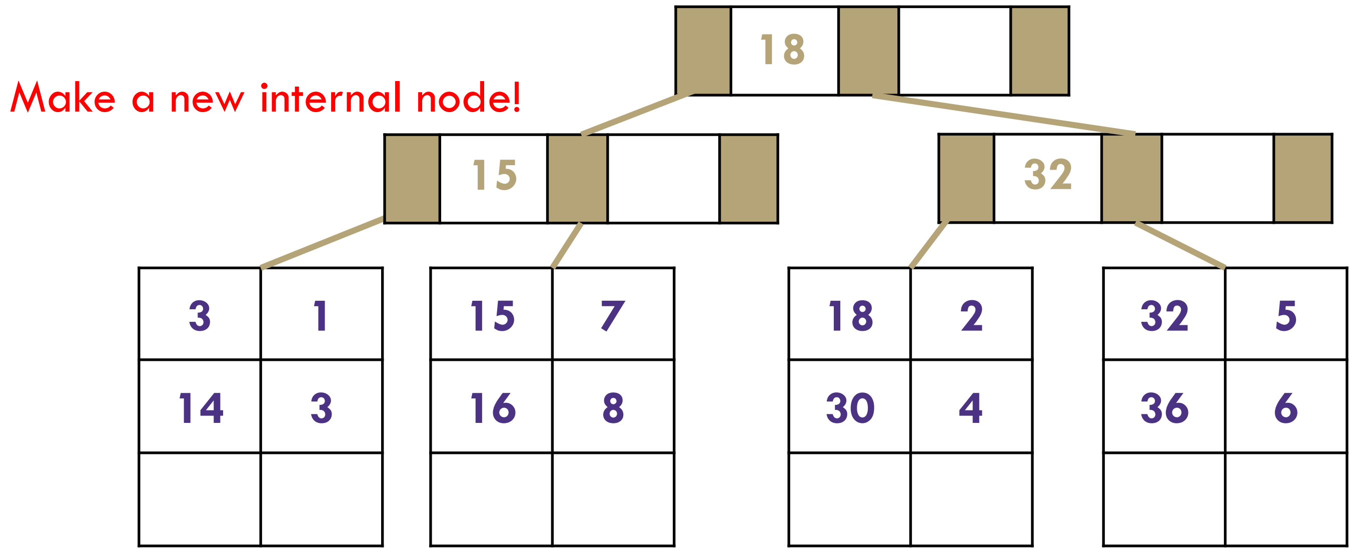
Try inserting (32, 5) and (36, 6) into the following tree



Splitting internal nodes

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





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: **$\Theta(L + M\log_m(n))$**