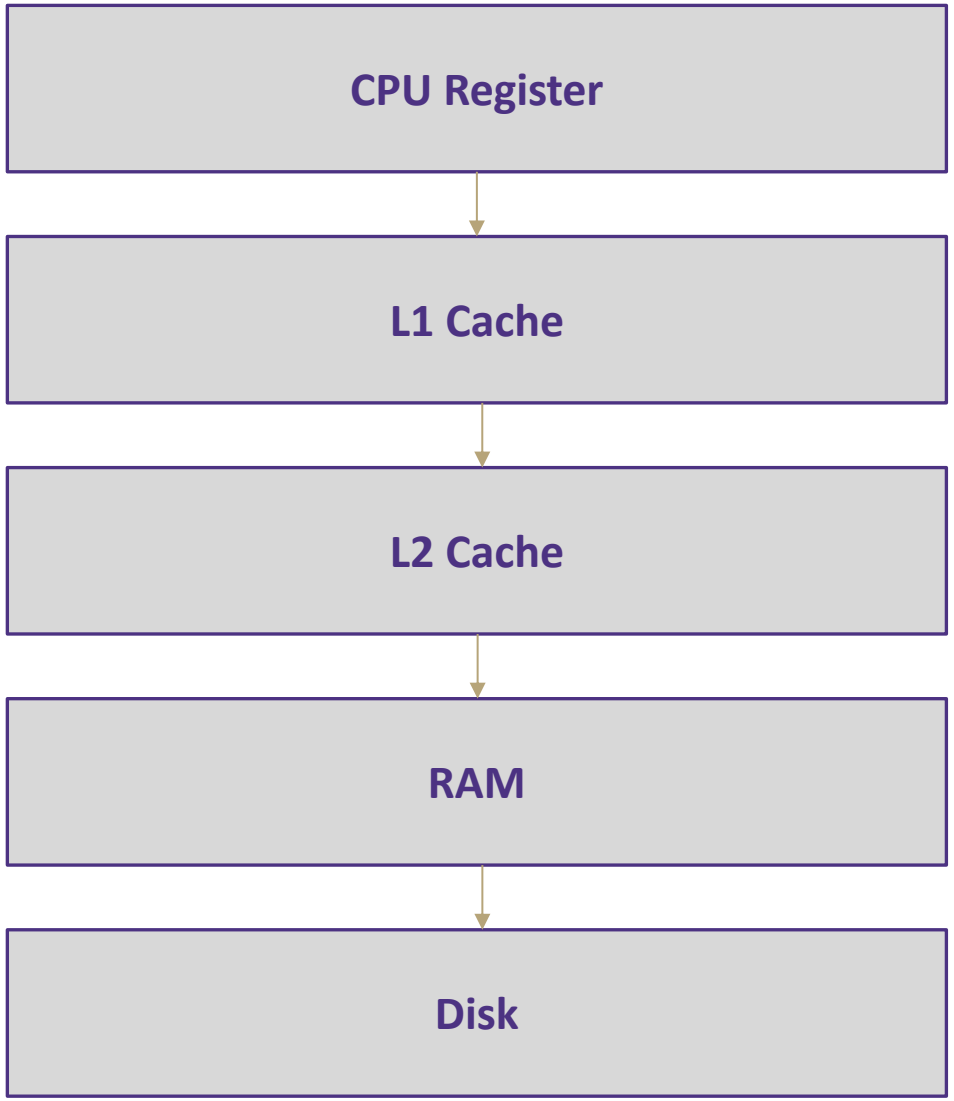
B trees

Data Structures and Algorithms

Warm Up

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?

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

Locality

How does the OS minimize disk accesses?

Spatial Locality

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

- Arrays

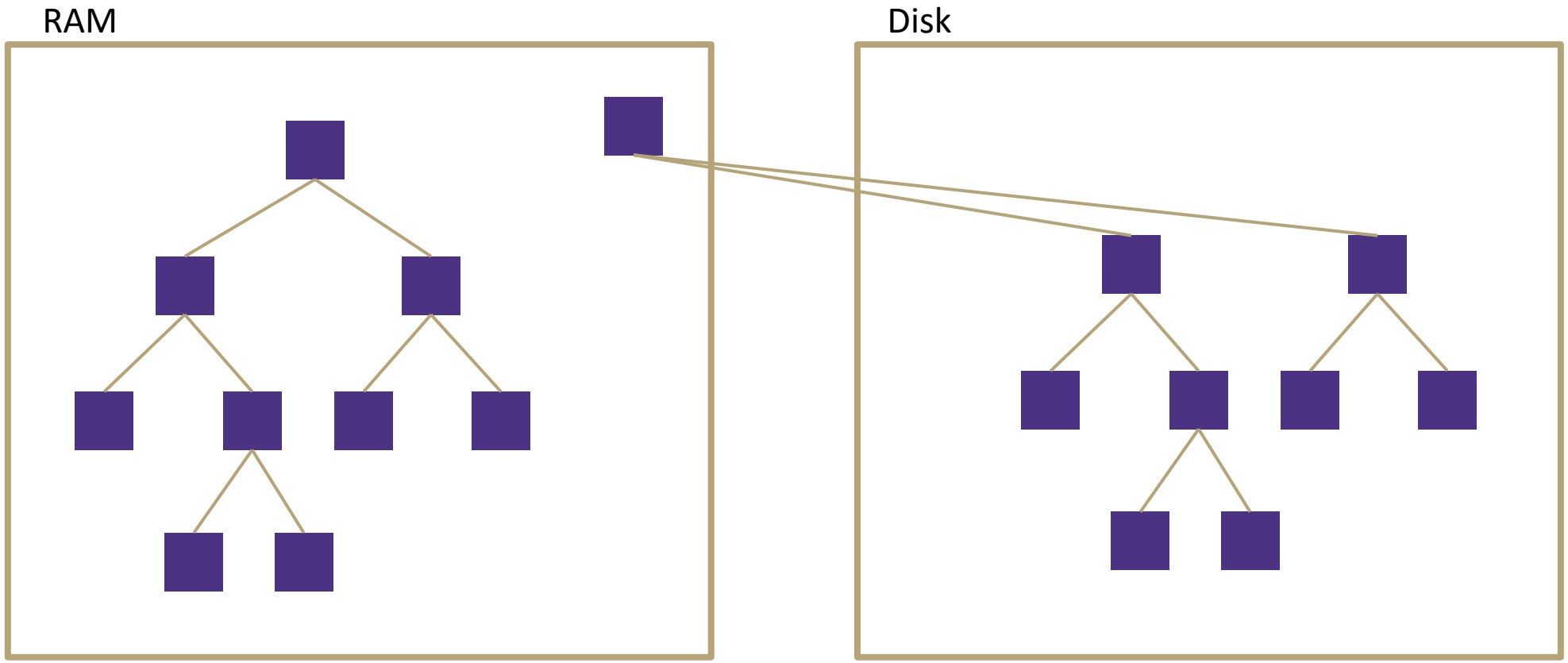
- Fields

Temporal Locality

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

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

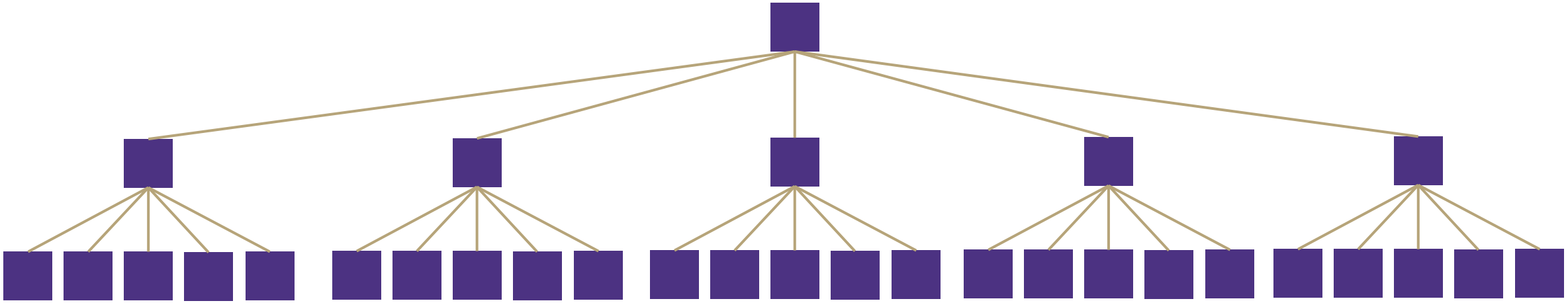
Instead of each node having 2 children, let it have M children.

- 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? **$\log_m(n)$**

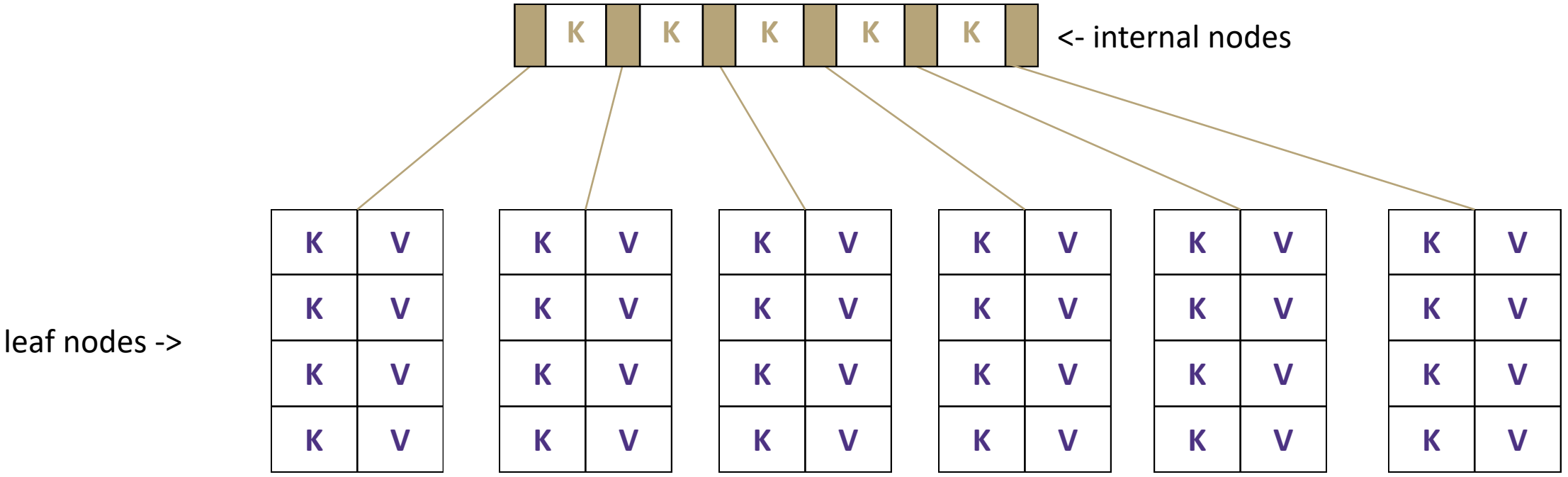
What is the worst case runtime of get() for this tree? **$\log_2(m)$ to pick a child**
$\log_m(n) * \log_2(m)$ to find node



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

M = 6

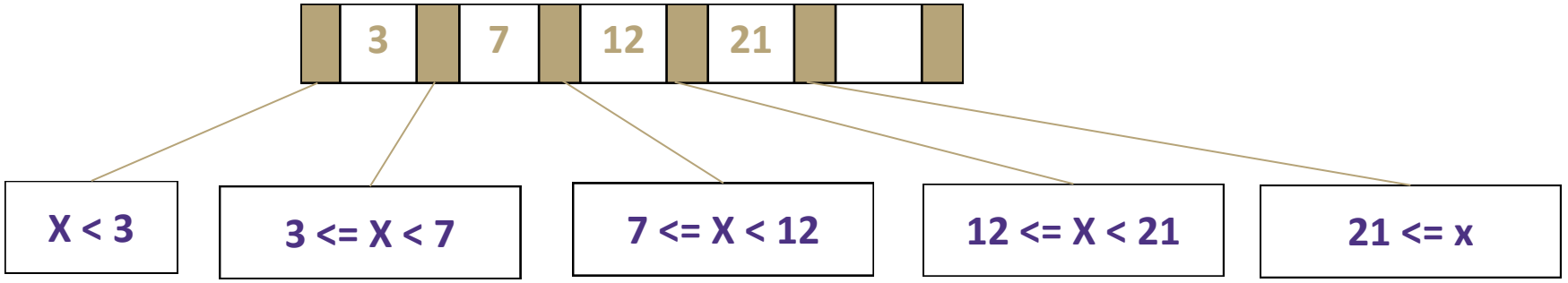
A leaf node contains L key-value pairs, sorted by key

L = 3

K	V
K	V
K	V
K	V

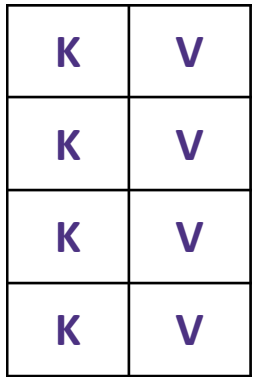
Order Invariant

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



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

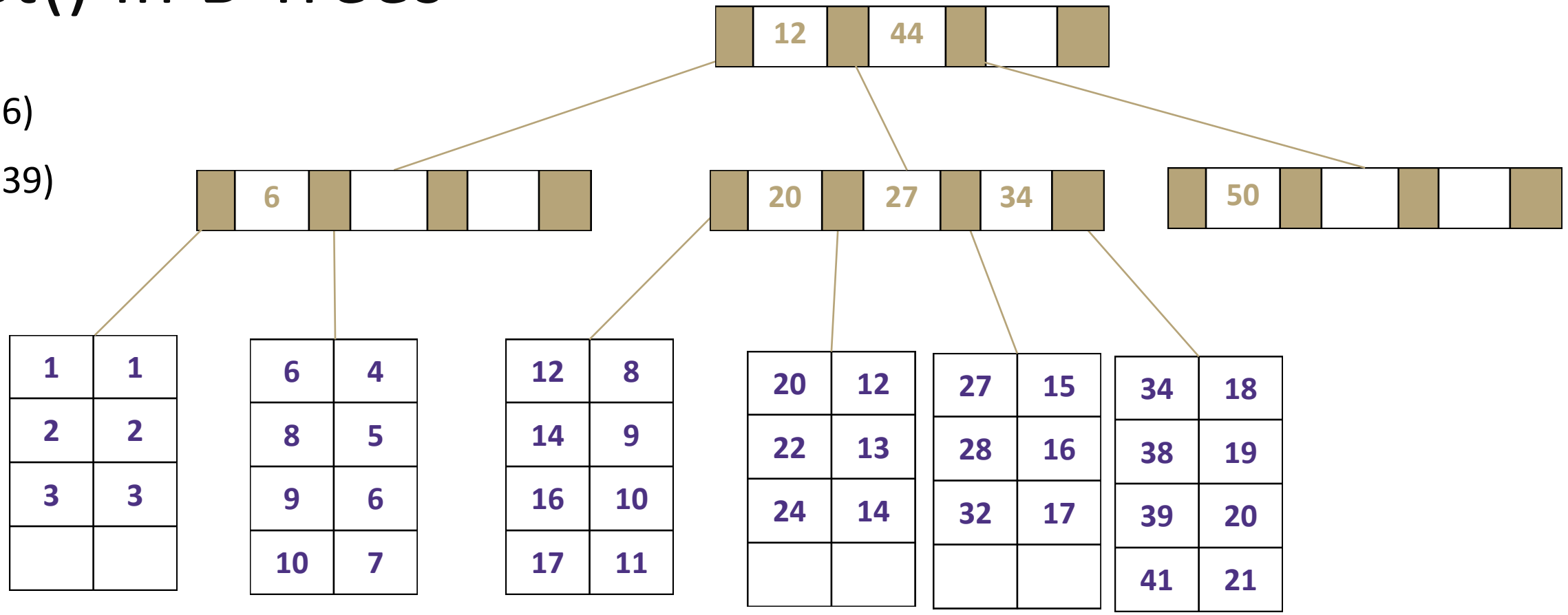
Has 3 invariants that define it

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