B+ trees

Data Structures and Algorithms

Administrivia

P6 - Seam Carving due Friday June 12th
- NO LATE ASSIGNMENTS

Exercise 6 due Monday June 1st

Midterm 2 goes out this Friday May 29th 8:30am PDT
- Due Sunday May 31st 8:30am PDT
- NO LATE ASSIGNMENTS
- No midterm topic questions during Office hours
- Piazza your questions – private post default
- Zach & Kasey will post some Midterm help zoom office hours

Start finding groups for final assessment
- Come to lecture on Friday for "group finding mixer"
- Come to final review session next week!
- Go to final section next week

Final Assessment Logistics

Released Friday June 5th 11:59pm

Due Wednesday June 10th 11:59PM

- 120 hours for ~2 hour assignment
- NO LATE ASSIGNMENTS ACCEPTED

Group assignment!

- No limit on group sizes. We recommend size 4 groups though
- Open note

Gradescope Assignment (like exercises)

No OH questions answered during assessment

- Keep an eye on Piazza

Study resources

Design decision style questions from:

- section
- exercises
- post-lecture questions
- past finals

Review session next week

- Please fill out google survey!

Final Assessment Topics

- Code modeling (asymptotic analysis and case analysis)
  - Some based on homework data structures
- Graph modeling & algorithms
  - BFS/DFS
  - MST
    - (Prims vs Kruskals)
  - Dijkstra’s
  - Topological Sort
- Design decisions, choosing an implementation and a design, explaining why that is optimal and the runtime and memory implications of your design
- ADTs (core functionality, what functionality is it optimized for, how do different implementations impact this ADT)
  - List
  - Stack
  - Queue
  - Deque
  - Map
  - Priority Queue
  - Disjoint Set
- Sorting Algorithms (basic understanding of how algorithm works,
  - Selection
  - Insertion
  - Heap (in place as a memory optimization)
  - Merge
  - Quick (in place optimization)
- Data structures
  - Array
  - Linked Nodes
  - BST
  - AVL Tree
  - Binary Heap
  - Disjoint Set Implementations
  - Homework implementations
- NOT ON
  - Tries
  - Tree method
  - Simplifying summations

Final Assessment Questions

Possible Question Types (Super set)

- Code Modeling
  - given a piece of code what is the best/worst runtimes
- Testing/Debugging
  - Given a piece of code or a design - how would you approach testing it?
  - What inputs would you select to exercise a given piece of code to expose possible bugs?
- Graph Modeling
  - Given a scenario and data set, how would you represent it as a graph?
  - How can you apply/adapt graph algorithms to find your desired outcome?
- Design Decisions
  - Given a scenario which ADT is the best fit to optimize for the functionality you need?
  - Given a scenario which data structure would you use to implement a solution, what are the implications of that solution for runtime and memory usage?
  - For a given design - how would you adapt the chosen data structures, algorithms to solve the problem? How does this design perform in regards to runtime and memory usage
  - Given a variation on an algorithm we talked about in class, compare and contrast by analyzing the differences and recognizing when this variation is better or worse.
- Reductions
  - How can you use an algorithm from class to solve a question about a given scenario by modifying the input (reducing the current problem to the one the algorithm solves)?

Roadmap

- review Friday concepts of caching, memory hierarchy
- AVL trees and disk accesses
- M-ary search trees (a generalization of binary trees except you can have m number of children)
- B+ trees
  - databases!
  - file systems!
  - when you have large amounts of data, consider using a B+ tree for its optimization for hardware.
  - high level motivation/idea (not going to go deep into the algorithms for maintaining invariants)

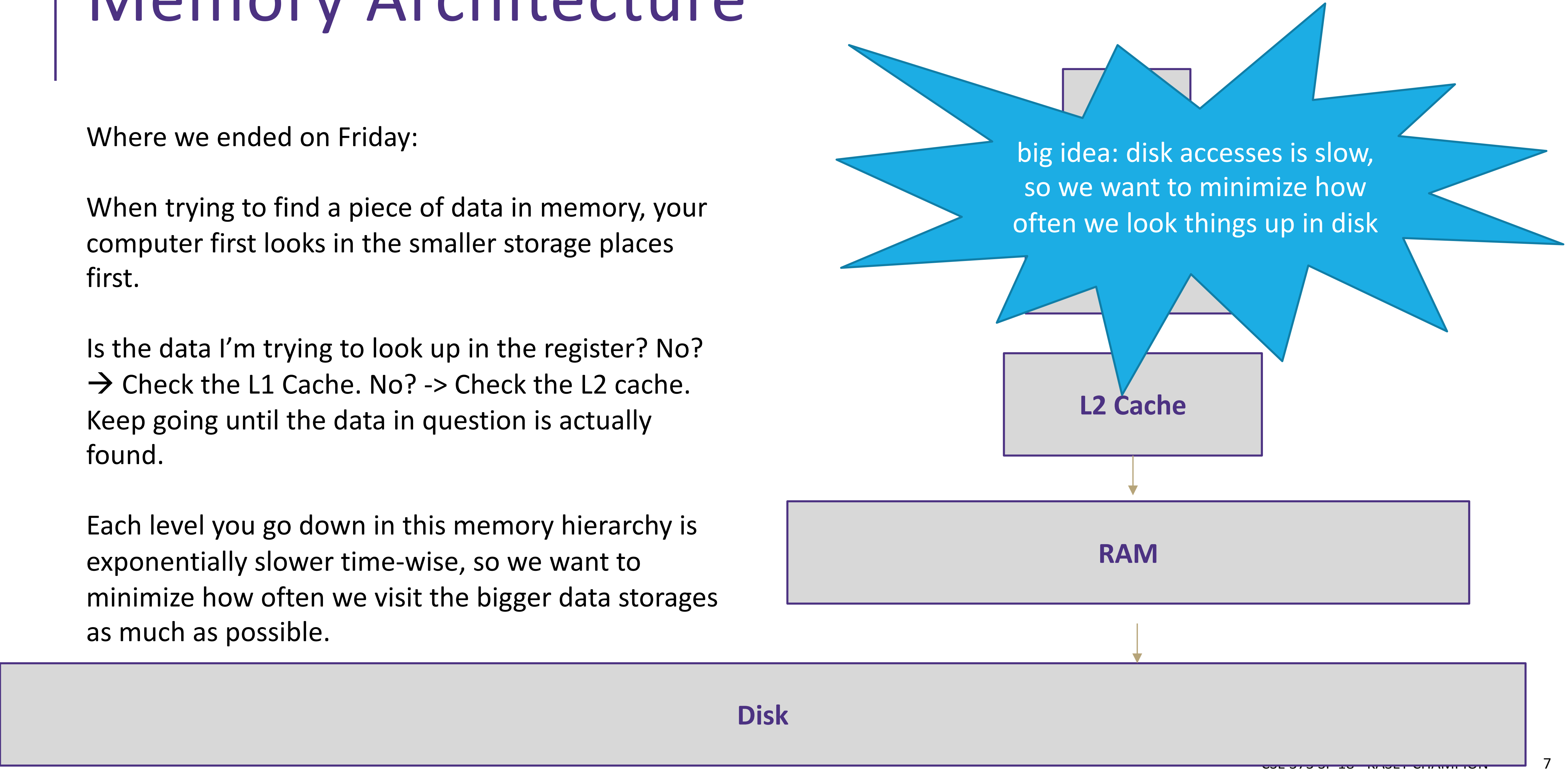
Memory Architecture

Where we ended on Friday:

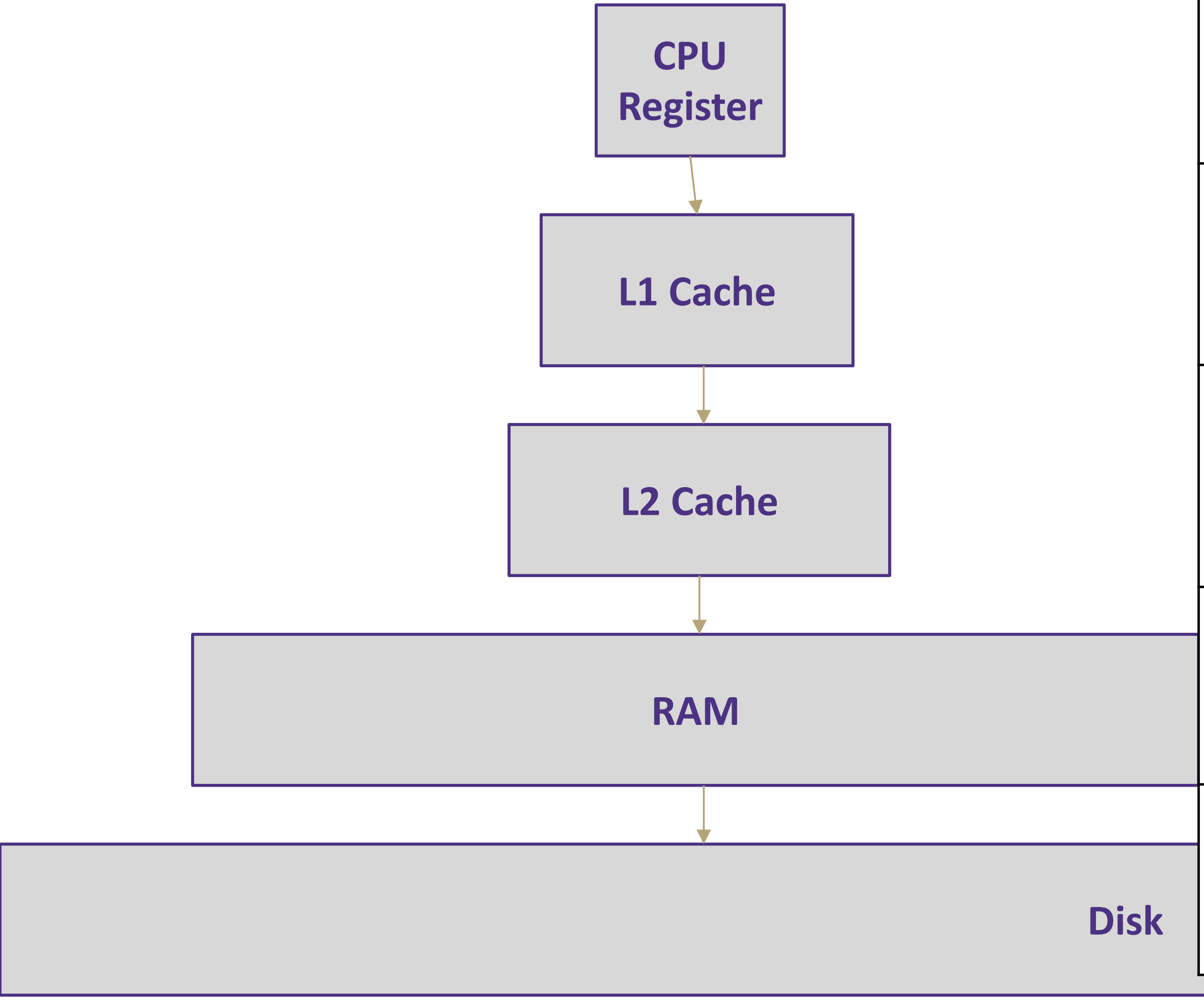
When trying to find a piece of data in memory, your computer first looks in the smaller storage places first.

Is the data I'm trying to look up in the register? No? → Check the L1 Cache. No? -> Check the L2 cache. Keep going until the data in question is actually found.

Each level you go down in this memory hierarchy is exponentially slower time-wise, so we want to minimize how often we visit the bigger data storages as much as possible.


big idea: disk accesses is slow, so we want to minimize how often we look things up in disk

Memory Architecture

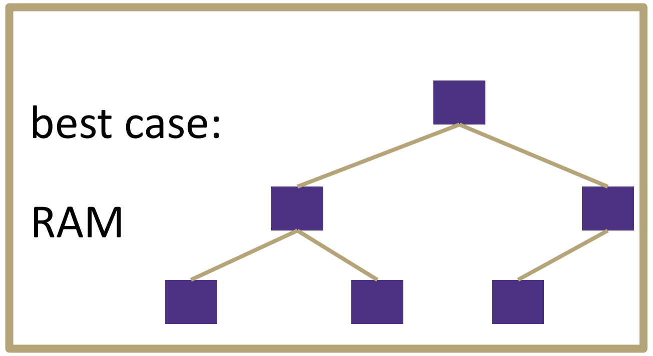


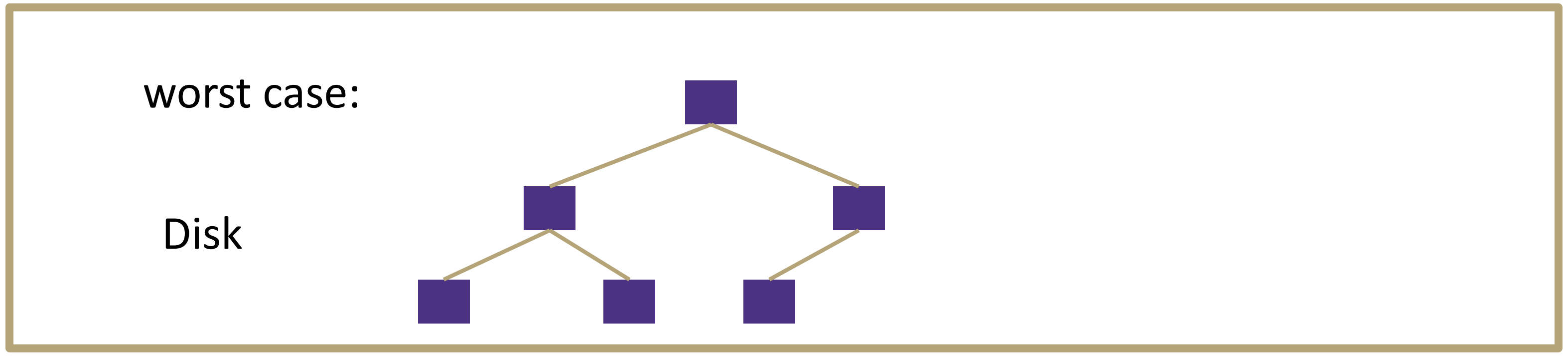
What is it?	Typical Size	Time
The brain of the computer!	32 bits	≈free
Extra memory to make accessing it faster	128KB	0.5 ns
Extra memory to make accessing it faster	2MB	7 ns
Working memory, what your programs need	8GB	100 ns
Large, longtime storage	1 TB	8,000,000 ns

Questions? Review anything?

Thought Experiment

Suppose we have an AVL tree of height 50 (looooots of data – 2^50 nodes). What is the **best** case scenario for number of disk accesses to access a leaf? What is the **worst** case when accessing a leaf?





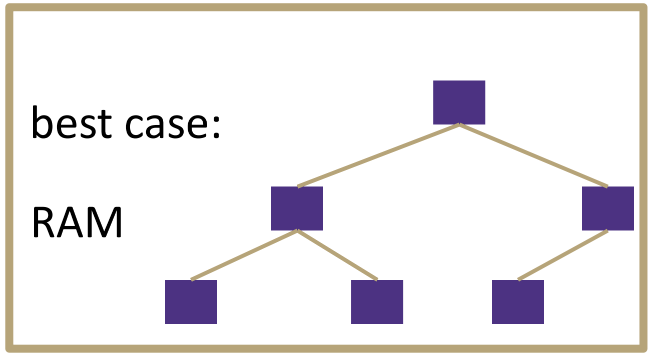
Worst-case everything is disk and log(n) disk accesses (one for each node we look at).

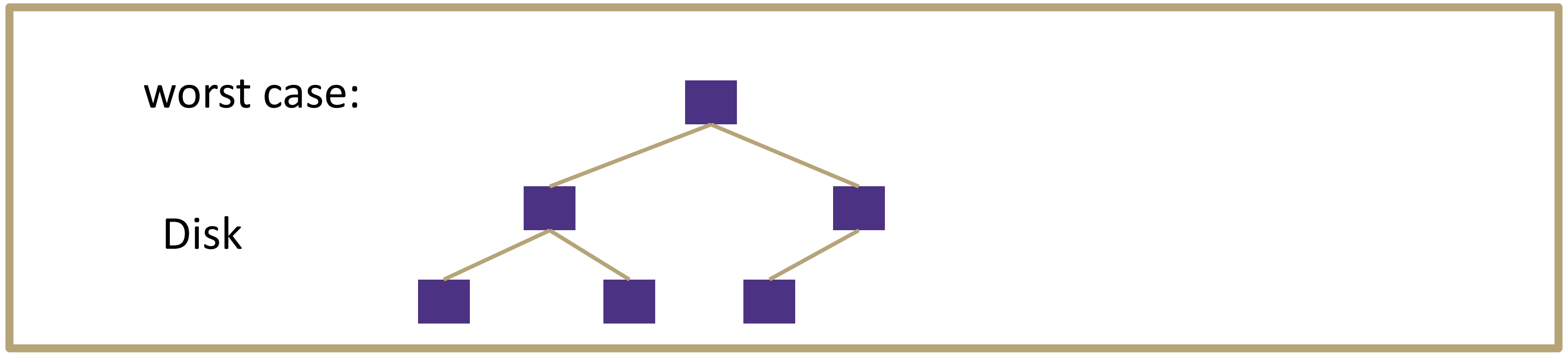
Recall that for object node-based structures (linked lists, binary trees, etc.) they are stored not contiguously in memory.

It's true that your computer will pull in a bunch of extra data from disk (called a page) just like we do for RAM going to the caches, when we access our current node, but because of how objects are stored in memory it's very unlikely we'd pull in any other nodes in the tree.

Thought Experiment

Suppose we have an AVL tree of height 50 (looooots of data – 2^50 nodes). What is the **best** case scenario for number of disk accesses to access a leaf? What is the **worst** case when accessing a leaf?





Takeaway: It is inevitable that some data will be stored on disk that we have to look up when we have so much data that it can't all fit in RAM.

We're going to try to optimize a dictionary/database data structure to make as few disk accesses as possible (ideal for large amounts of data)

Worst-case everything is disk and log(n) disk accesses (one for each node we look at).

Recall that for object node-based structures (linked lists, binary trees, etc.) they are stored not contiguously in memory.

It's true that your computer will pull in a bunch of extra data from disk (called a page) just like we do for RAM going to the caches, when we access our current node, but because of how objects are stored in memory it's very unlikely we'd pull in any other nodes in the tree.

Minimizing # of Disk Accesses

Idea: Node sizes of our AVL trees / binary trees is usually small and the amount of data (a page) we move from Disk to RAM is bigger than that. If we can stuff more useful information in each node maybe we can decrease the height of the tree and decrease the # of disk accesses required.

Instead of each node having 2 children (binary tree), let's generalize this and let it have M children.

- Each node contains a **sorted** array of children

Pick a size M big enough so that the node fills an entire page of disk data.

M-ary search trees

note: the details of M-ary search trees are vague (it's not well-defined and is just a transition into B+ trees, so don't worry an actual example. It turns out we need the ideas from B+ trees to make this feasible.)

Instead of each node having 2 children (a binary tree), let it have M children and call it a M-ary tree.

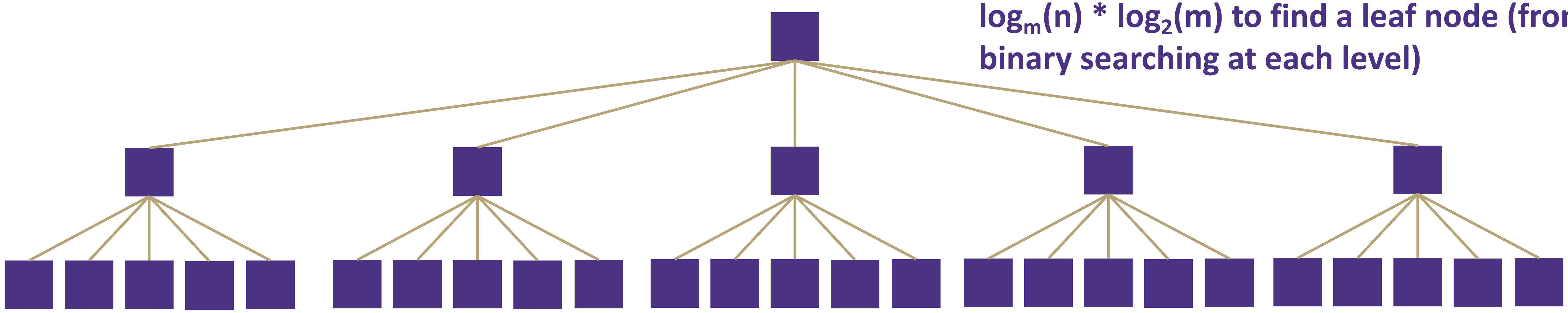
- Each node contains a **sorted** array of children

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

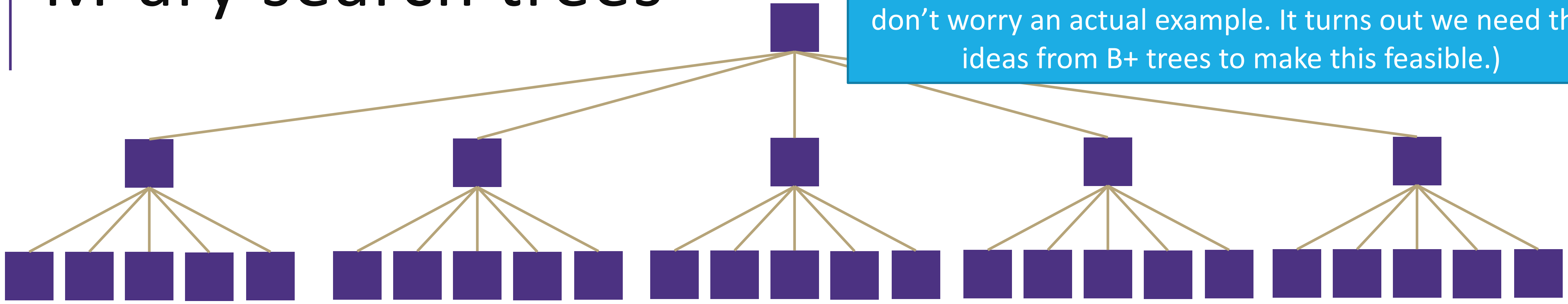
What is the worst case runtime of get() for this tree?

$\log_2(m)$ to pick a child with binary search
$\log_m(n) * \log_2(m)$ to find a leaf node (from binary searching at each level)



M-ary search trees

note: the details of M-ary search trees are vague (it's not well-defined and is just a transition into B+ trees, so don't worry an actual example. It turns out we need the ideas from B+ trees to make this feasible.)



How many disk accesses are we potentially making? Should be fewer but there's still some inefficiency:

If we focus on 1 node trying to find which child branch to go down … the keys are stored inside each node. So if we want to see what the keys of the children are we have to make another disk access! Every time we go down a level in the tree, we're potentially making a bunch of disk accesses (**$\log_2(m)$** of them).

Questions? Review anything?

Roadmap

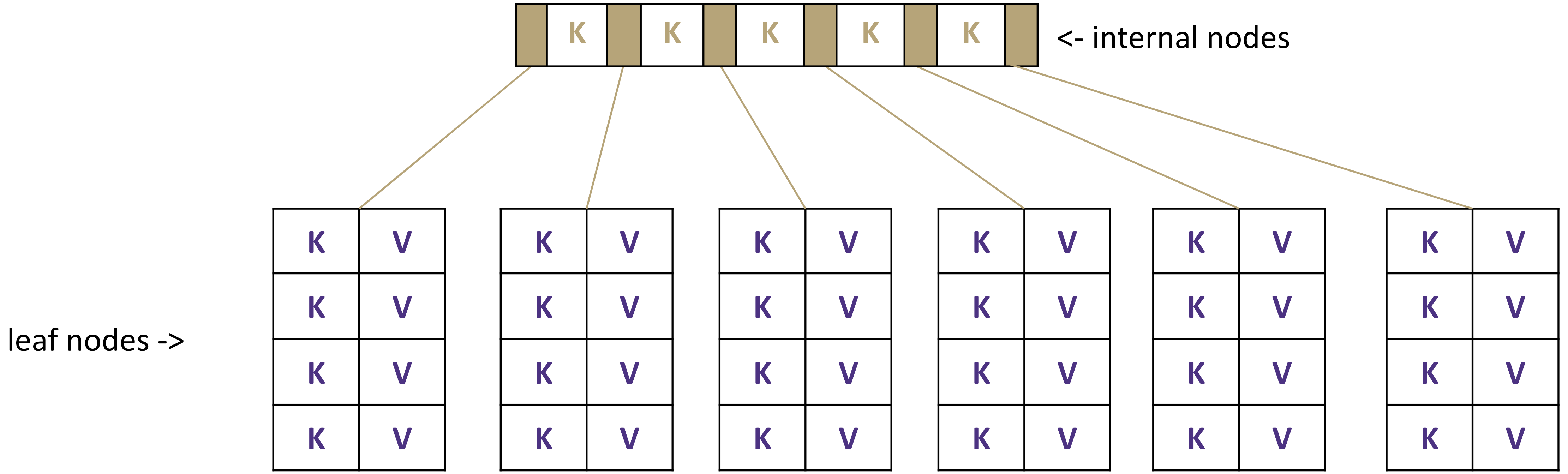
- review Friday concepts of caching, memory hierarchy
- AVL trees and disk accesses
- M-ary search trees (a generalization of binary trees except you can have m number of children)
- B+ trees
  - databases!
  - file systems!
  - when you have large amounts of data, consider using a B+ tree for its optimization for hardware.
  - high level motivation/idea (not going to go deep into the algorithms for maintaining invariants)

B+ trees

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

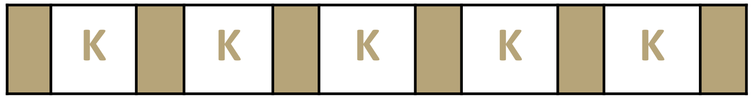
What if we construct an M-ary search tree that:

- stores keys together in branch nodes
- all the values in leaf nodes (so we can really maximize stuffing useful information in each node)



Node Invariant

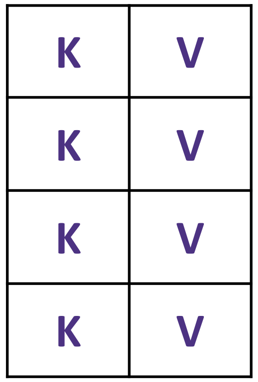
Internal nodes contain M pointers to children and M-1 **sorted** keys



M = 6

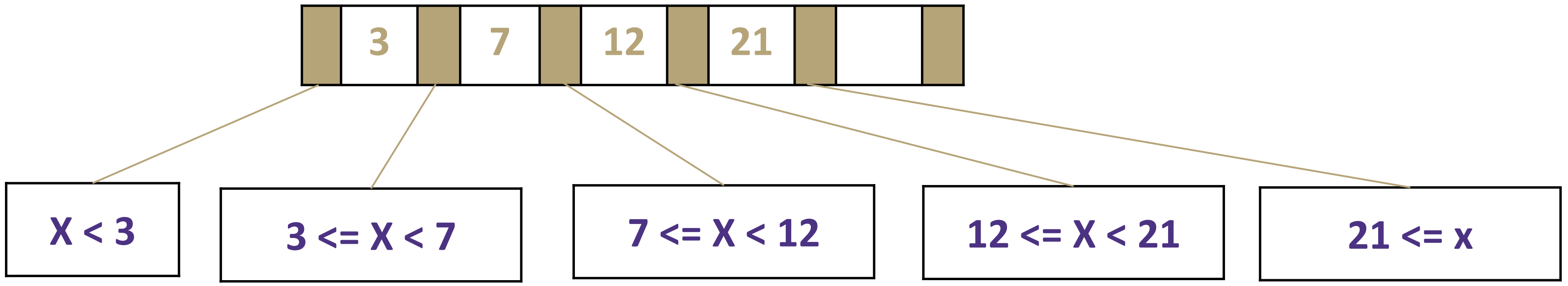
A leaf node contains L key-value pairs, sorted by key

L = 3

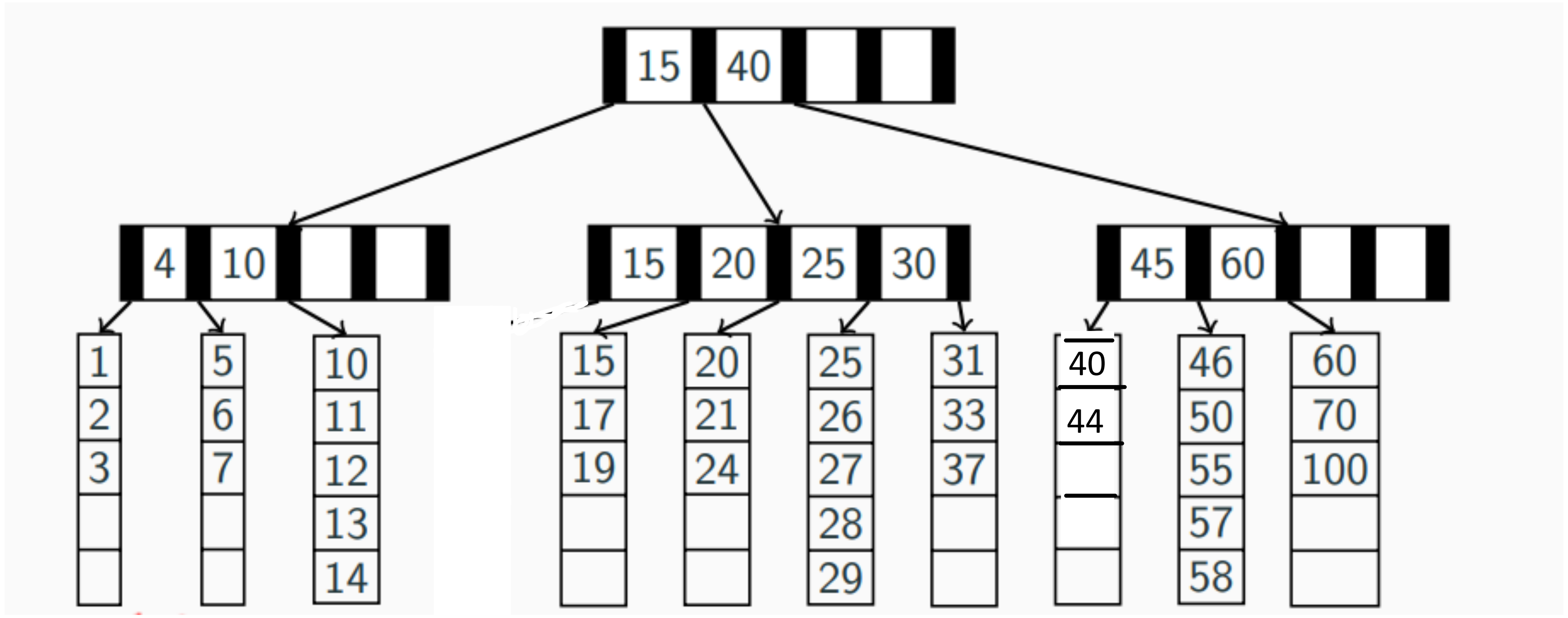


Order Invariant

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



B+ tree example: get(46)



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 or if it’s a leaf: the relevant K/V pairs).
3. We use as much of the page 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 binary search within a node is insignificant compared to disk access time.

What about the implementation details? for adding/removing/etc. ?

It’s fine – more details than we want to focus on atm. We have enough details about B+ trees to be able to use them / figure out good use cases for them, and that’s all we’re going to take away from this class.

Takeaways:

- disk lookups are slow, so if you have large amounts of data (so some or a lot of it is on disk), consider using a 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 time we visit a node it’s only one disk access
- if you ever have scenarios where you have large amounts of data (large databases, file systems, etc.) and need a choice of map: consider B+ trees!

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

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.

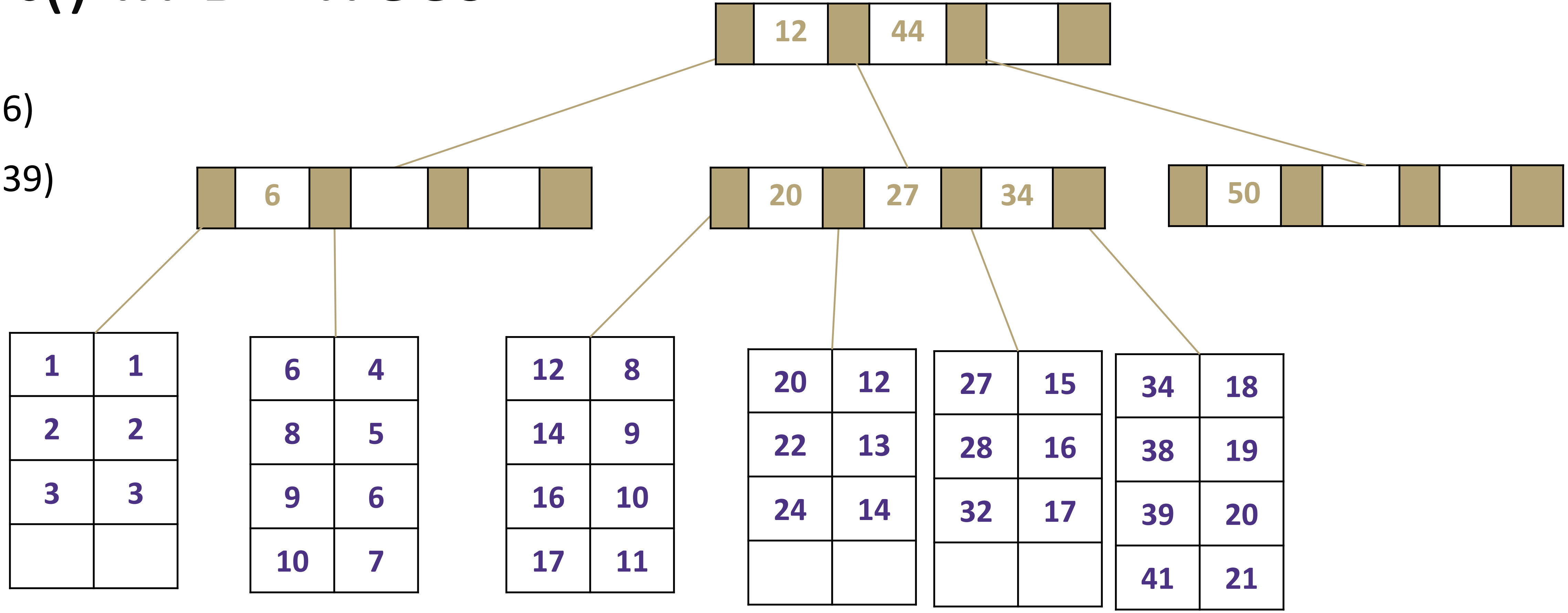
2. B+ trees order invariant
   - For any given key k, all subtrees to the left may only contain keys that satisfy x < k
   - All subtrees to the right may only contain keys x that satisfy k >= x

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