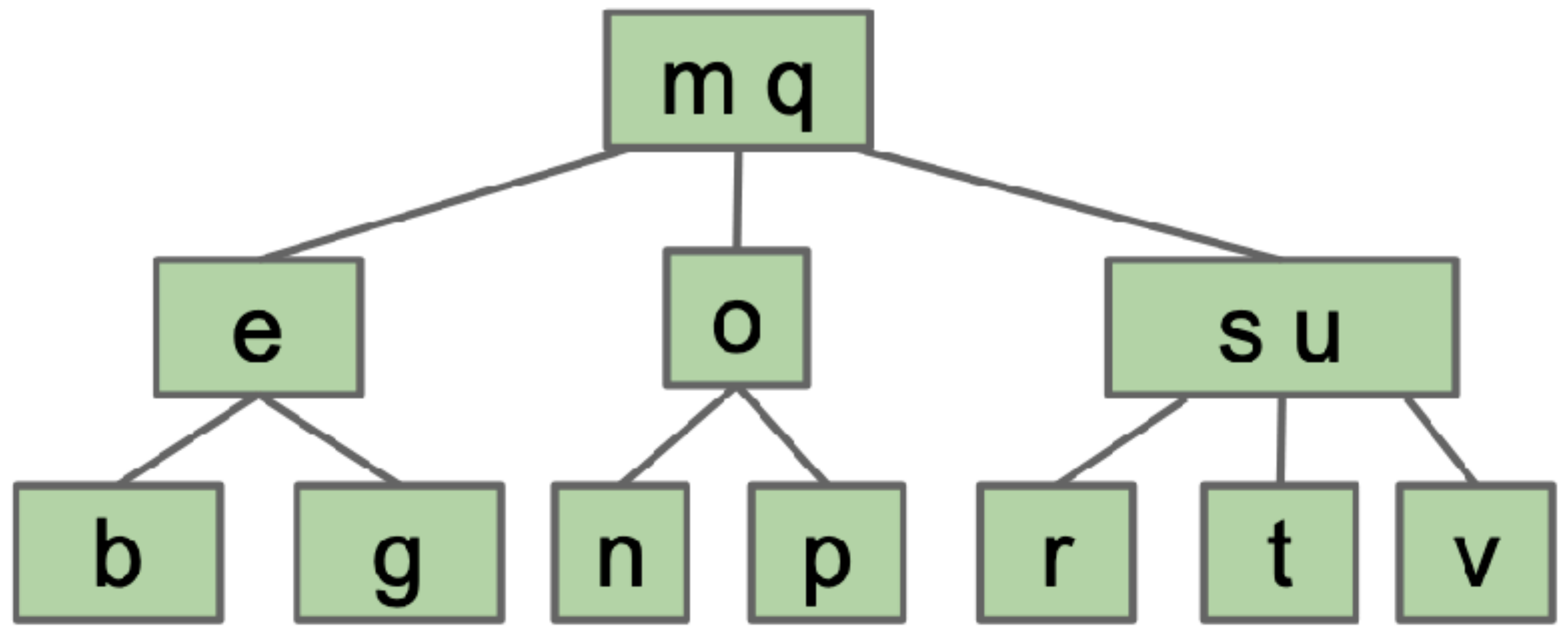
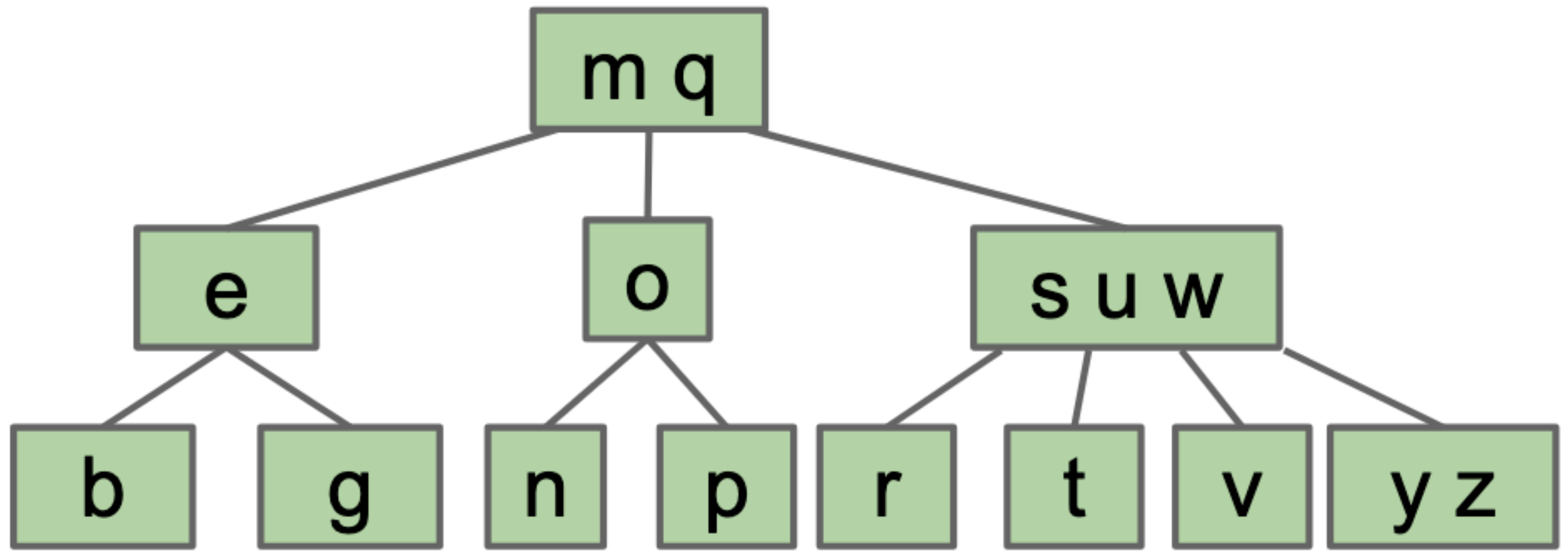
CS62 Class 18: B-Trees

Searching



2-3 tree

(max 2 items per node,
each node can have 2 or 3 children)



2-3-4 tree

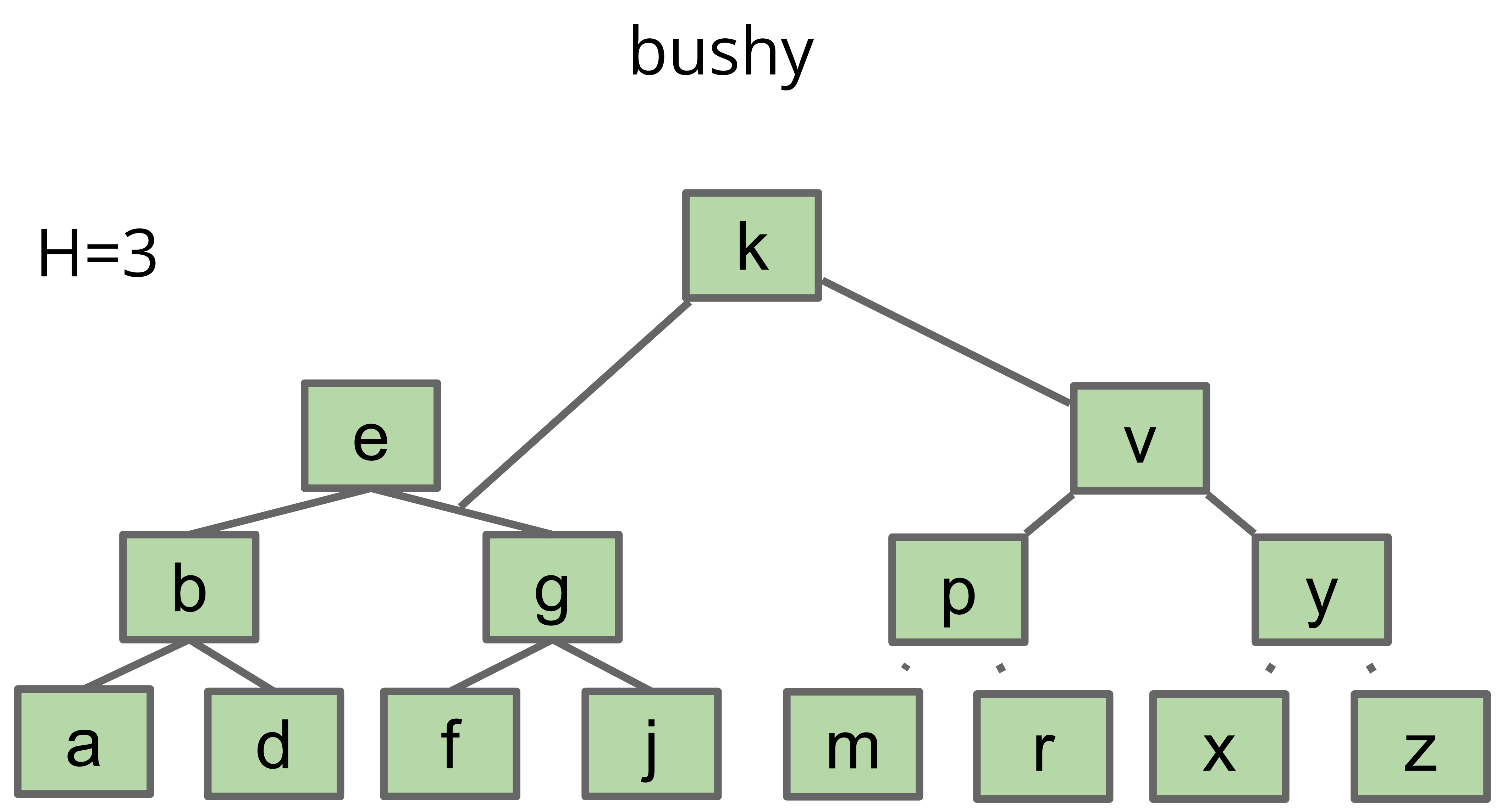
(max 3 items per node,
each node can have 2, 3, or 4 children)

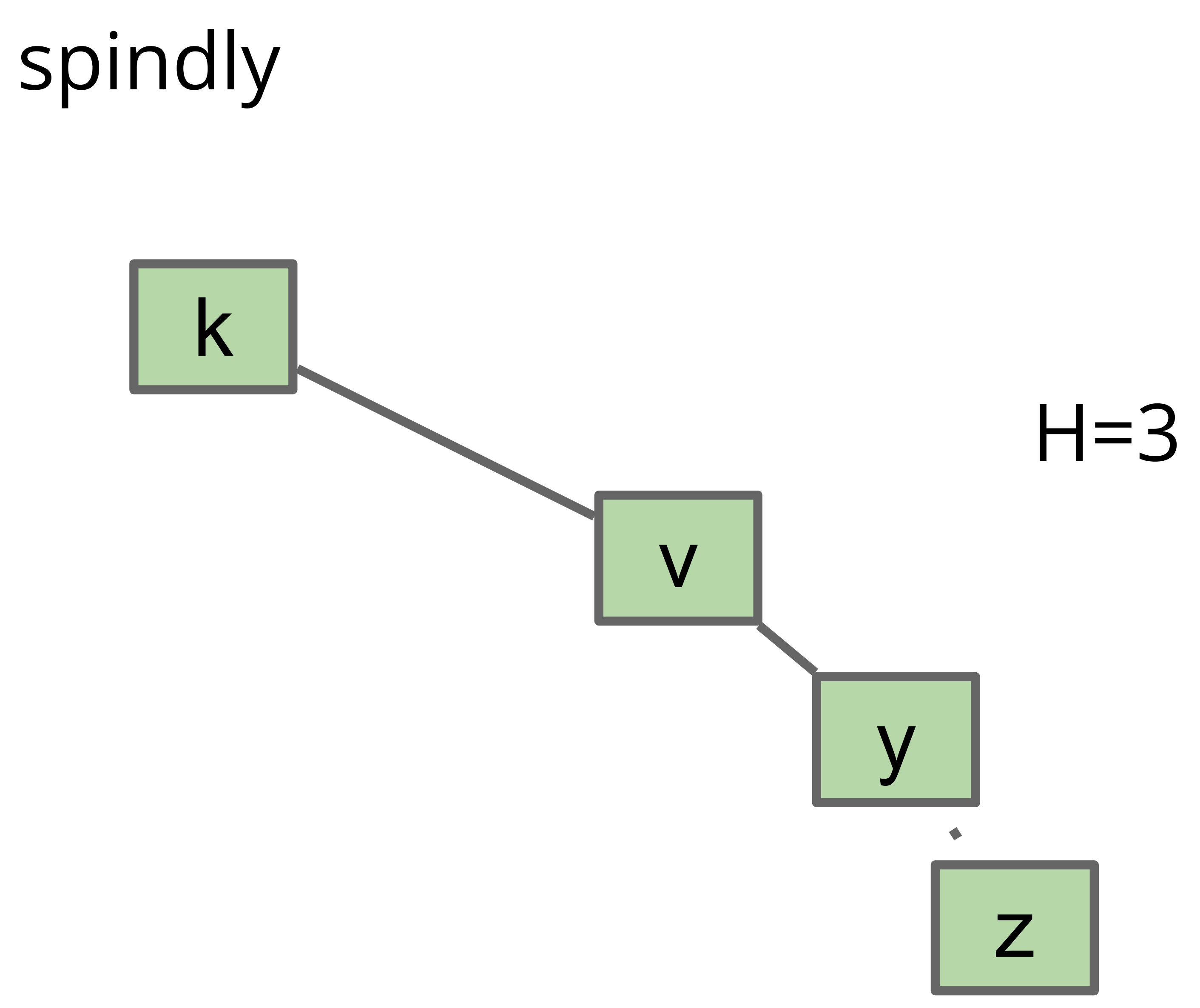
Agenda

- B-Trees
  - Motivation/derivation
    - Splitting nodes
    - Chain reaction splitting
  - Terminology
  - Construction & Invariants
  - Performance
  - (Note: we won’t cover deletion)

Last time

- Binary search trees (BSTs) can be “bushy” or “spindly” and BST operations take O(h), where h is the height of the tree. Bushy trees are of height log(n), while spindly trees are height n.
- Dramatic difference!





Last time review

Which of these statements are true?

A. Worst case BST height is $\Theta(N)$. "Is equal to linear."

B. BST height is $O(N)$. "Is less than or equal to linear."

C. BST height is $O(N^2)$. "Is less than or equal to quadratic."

Last time review

Which of these statements are true?

A. Worst case BST height is Θ(N). "Is equal to linear."

B. BST height is O(N). "Is less than or equal to linear."

C. BST height is $O(N^2)$. "Is less than or equal to quadratic."

All are **true**!

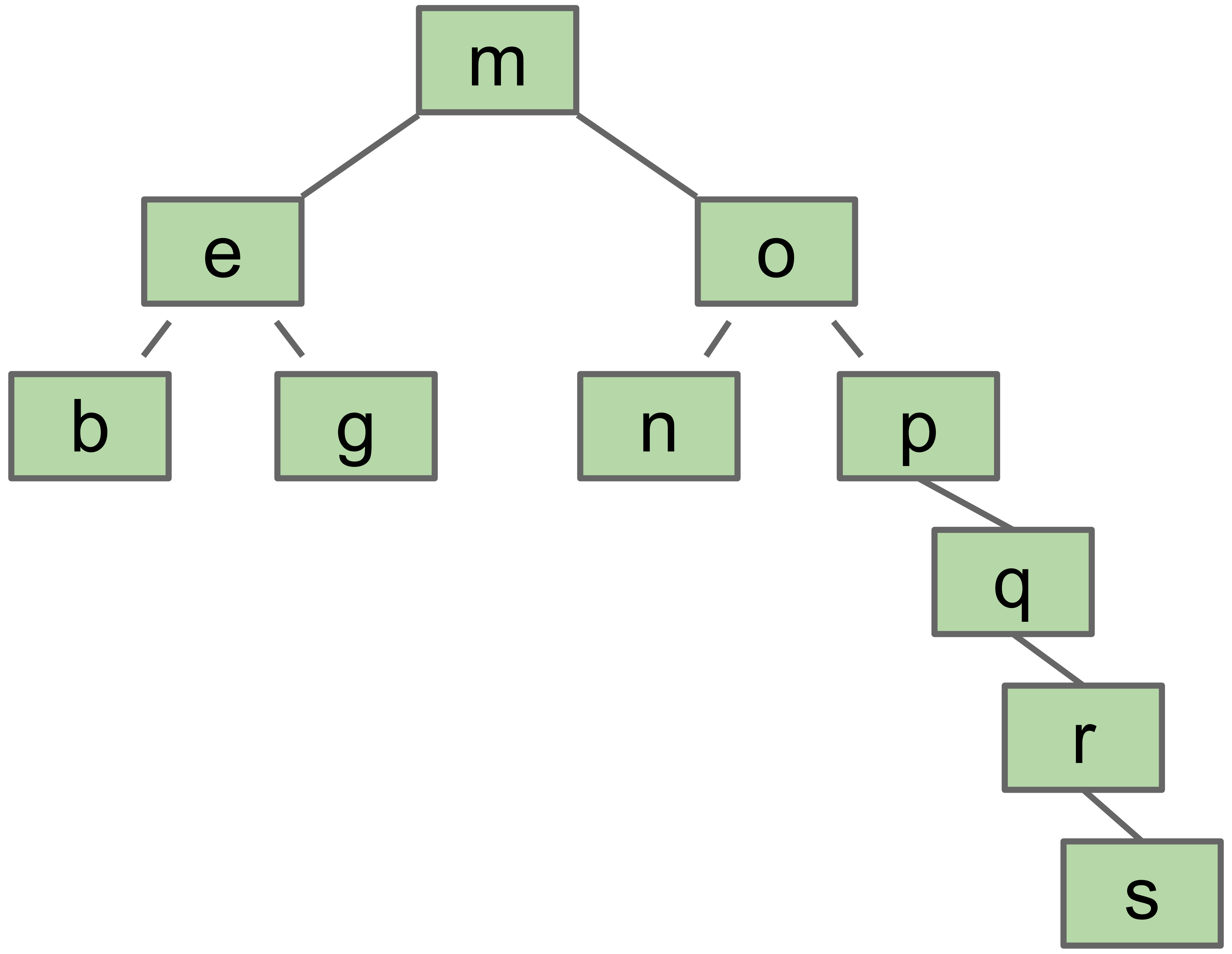
- A worst case (spindly tree) has a height that grows exactly linearly - Θ(N).
- All BSTs have a height that grows linearly or better - O(N).
- All BSTs have a height that grows quadratically or better - $O(N^2)$.
  - However, A is the *most informative* statement. $N^2 > N$, but we want the smallest upper bound, which is given to us by Θ(N). (equal, not less than or equal to)

B-trees: motivation

Good News and Bad News

Good news: BSTs have great performance if we insert items randomly. Performance is Θ(log N) per operation.

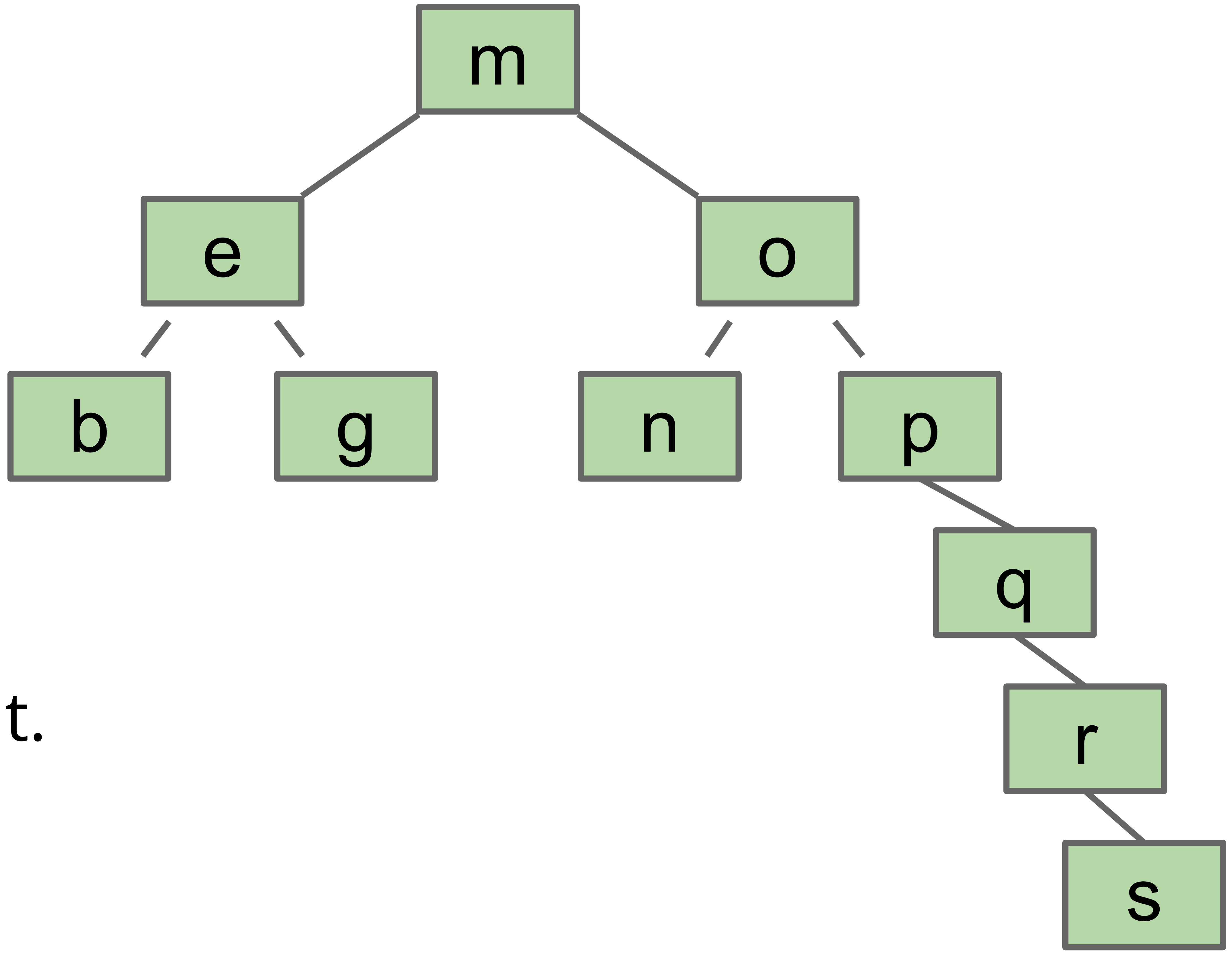
Bad News: We can't always insert our items in a random order. Why?



Good News and Bad News

Bad News: We can’t always insert our items in a random order. Why?

- Data comes in over time, don’t have all at once.
  - Example: Storing dates of events.
    - add(“01-Jan-2019, 10:31:00”)
    - add(“01-Jan-2019, 18:51:00”)
    - add(“02-Jan-2019, 00:05:00”)
    - add(“02-Jan-2019, 23:10:00”)



In this lecture, we’ll do something totally different.

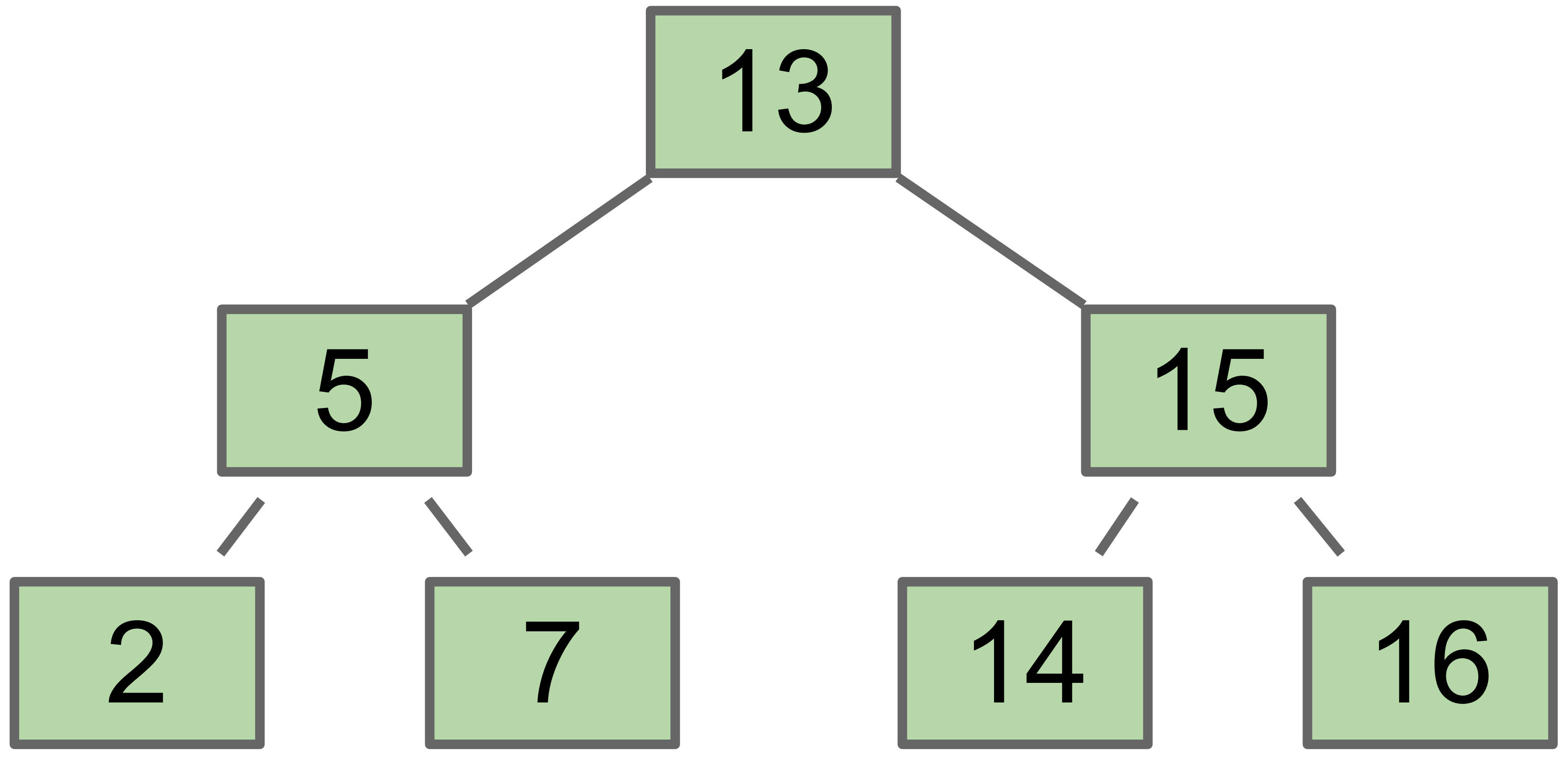
Avoiding Imbalance through Overstuffing

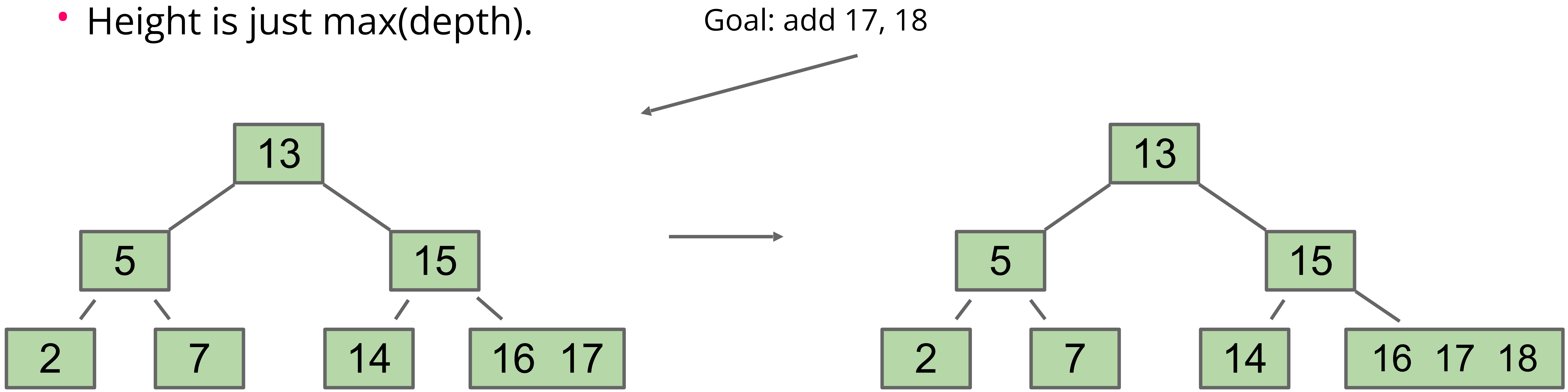
The problem is adding new leaves at the bottom, which can create spindly trees.

Why don’t we avoid new leaves allt ogether?

Why not just “overstuff” the existing leaf nodes?

- “Overstuffed tree” always has balanced height, because leaf depths never change.
- Height is just max(depth).

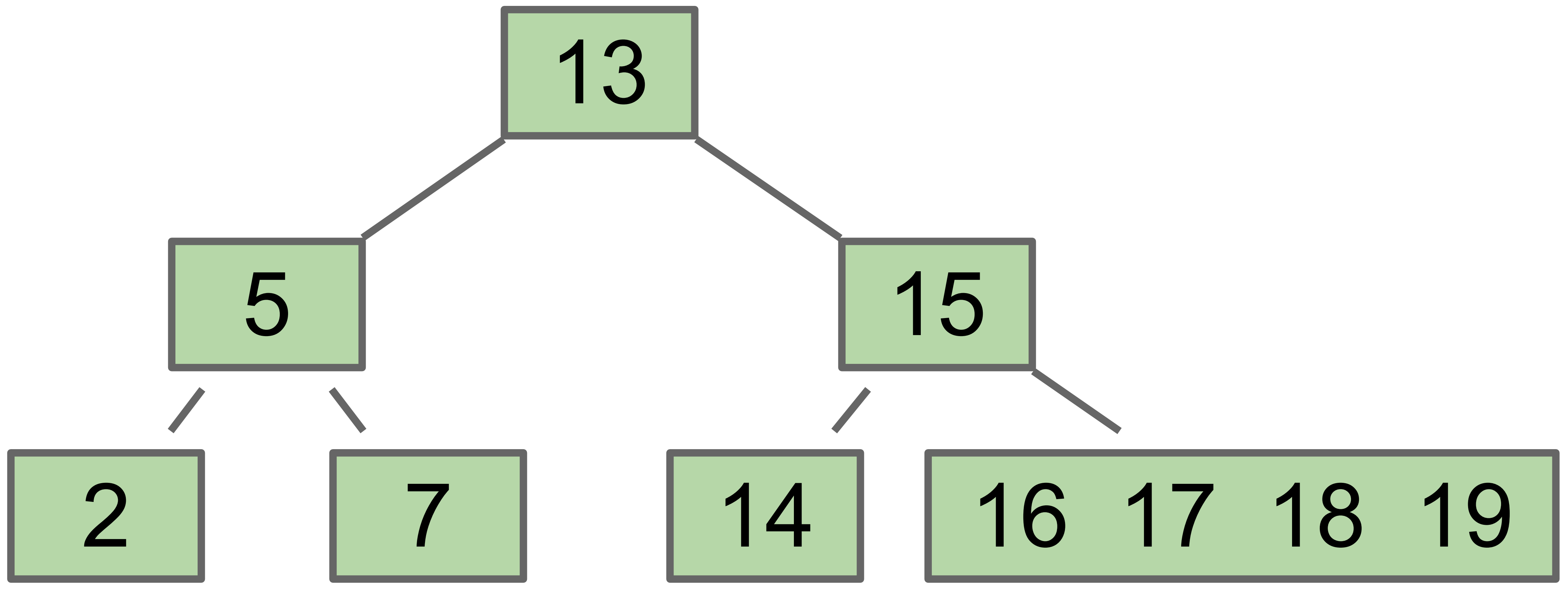




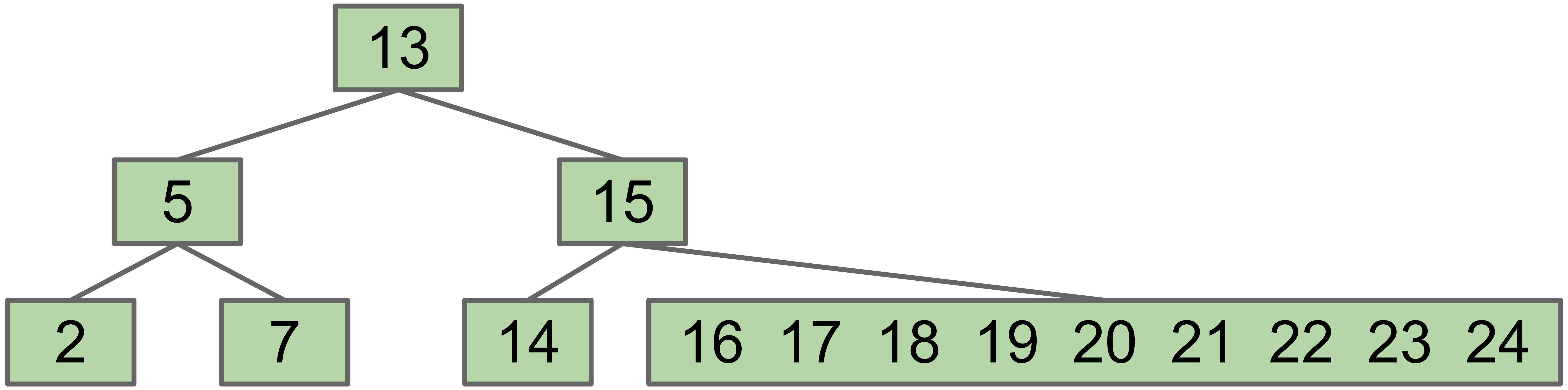
Avoiding Imbalance through Overstuffing

Overstuffed trees are a logically consistent but very weird data structure.

- contains(18):
  - Is 18 > 13? Yes, go right.
  - Is 18 > 15? Yes, go right.
  - Is 16 = 18? No.
  - Is 17 = 18? No.
  - Is 18 = 18? Yes! Found it.



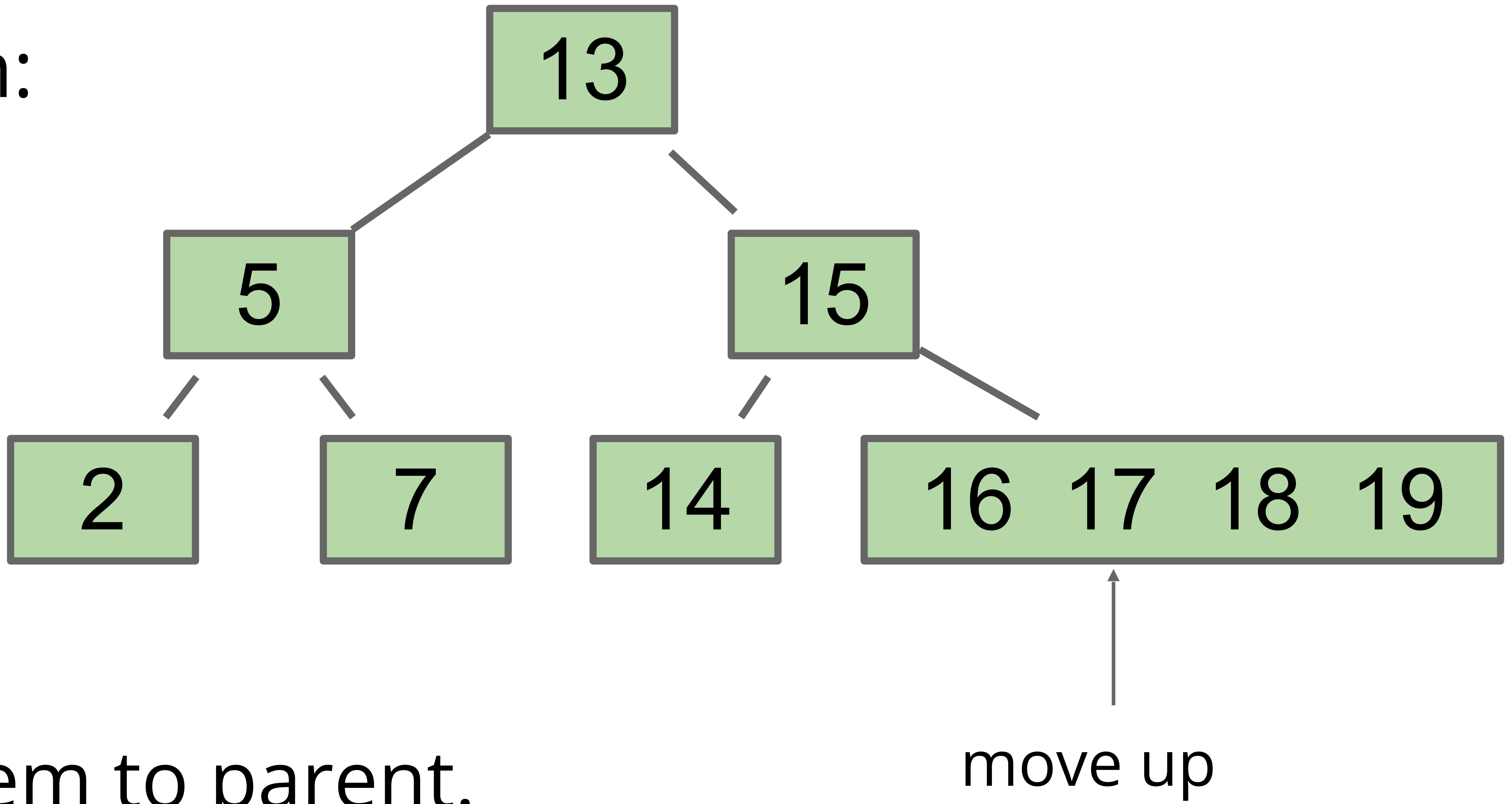
Q: What is the problem with this idea?



Revising Our Overstuffed Tree Approach: Moving Items Up

Height is balanced, but we have a new problem:

- Leaf nodes can get too juicy.



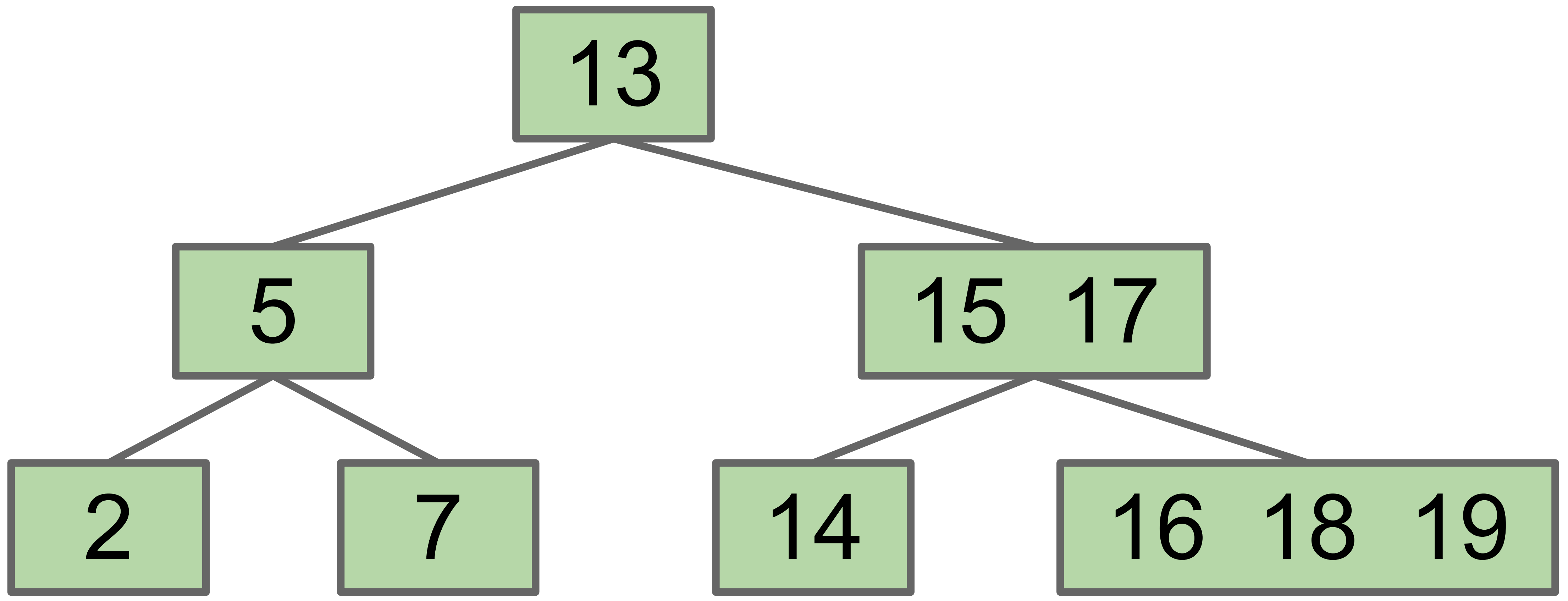
Solution?

- Set a limit L on the number of items, say L=3.
- If any node has more than L items, give an item to parent.
  - Which one? Let's say (arbitrarily) the left-middle.

Q: What's the problem now?

- 16 is to the right of 17.

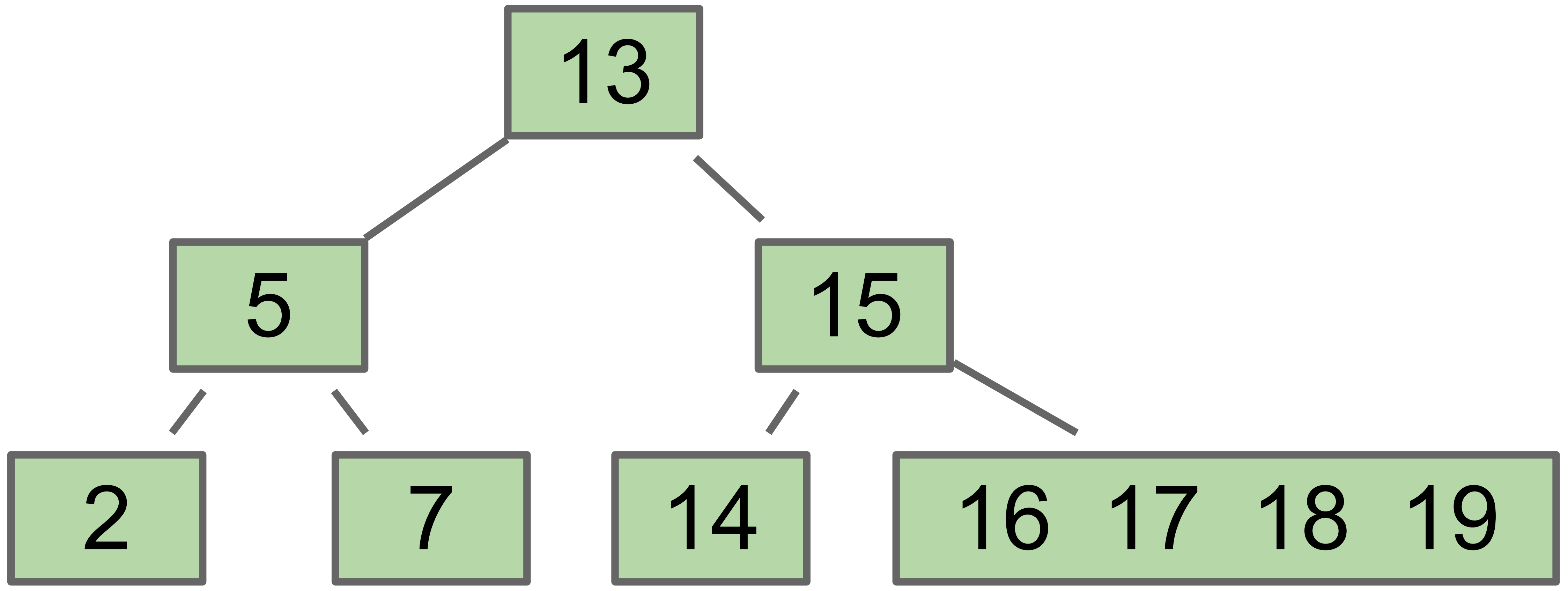
Q: How can we tweak this idea to make it work?



Revising Our Overstuffed Tree Approach: Node Splitting

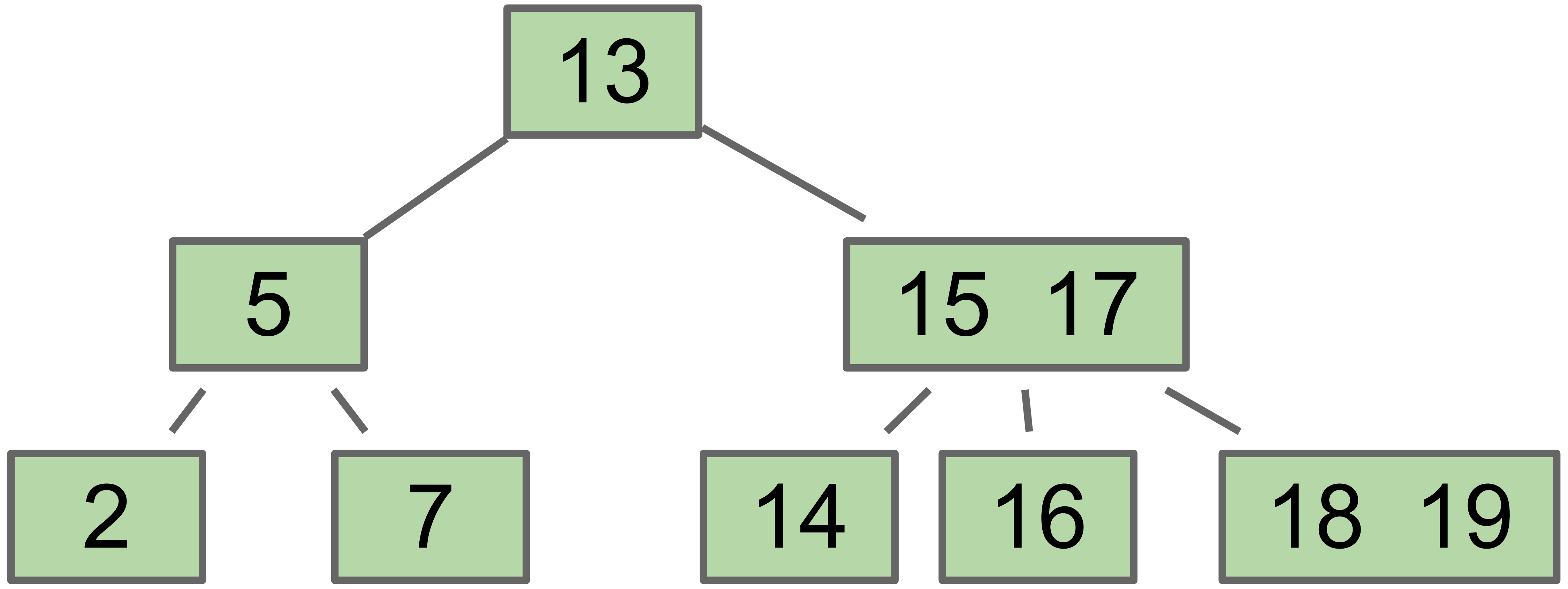
Height is balanced, but we have a new problem:

- Leaf nodes can get too juicy.



Solution?

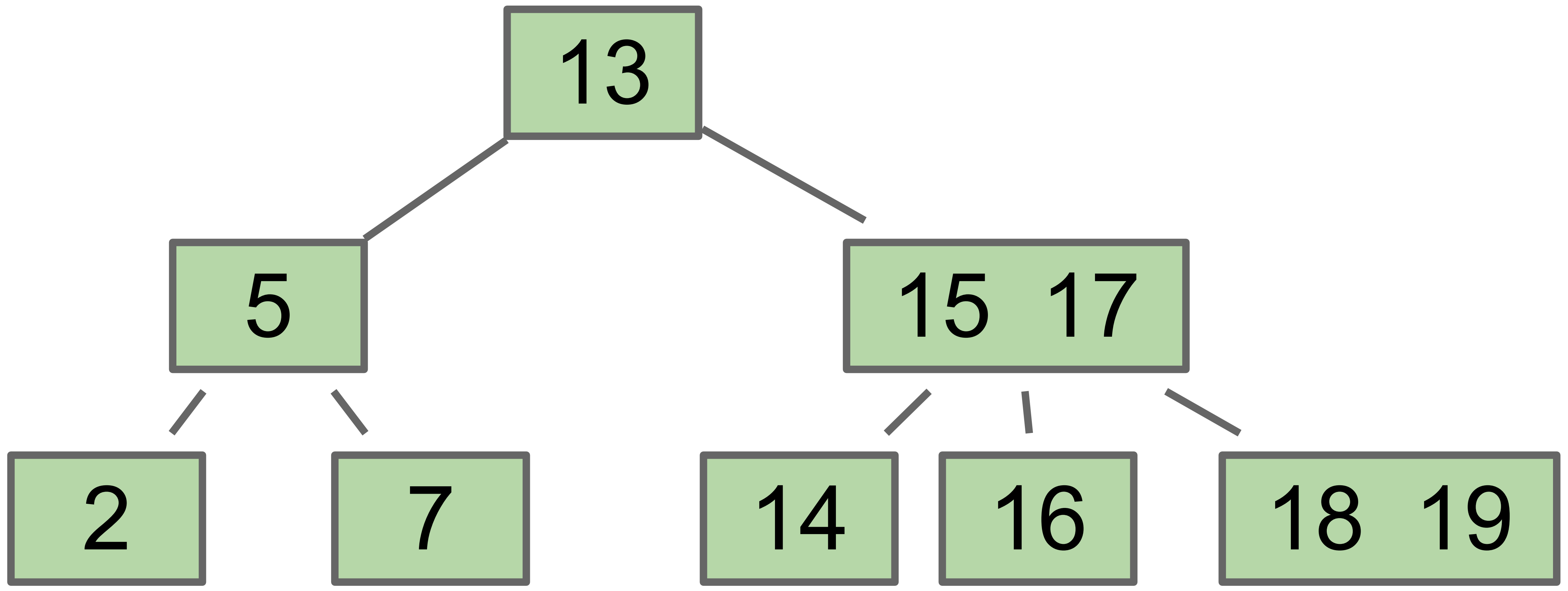
- Set a limit L on the number of items, say L=3.
- If any node has more than L items, give an item to parent.
  - **Pulling item out of full node splits it into left and right.**
  - Parent node now has **three children!** (no longer a binary tree)



Searching

Now, to search for a node in the new split tree:

- contains(18):
  - 18 > 13, so go right
  - 18 > 15, so compare vs. 17
  - 18 > 17, so go right
  - 18 = 18, we found it!



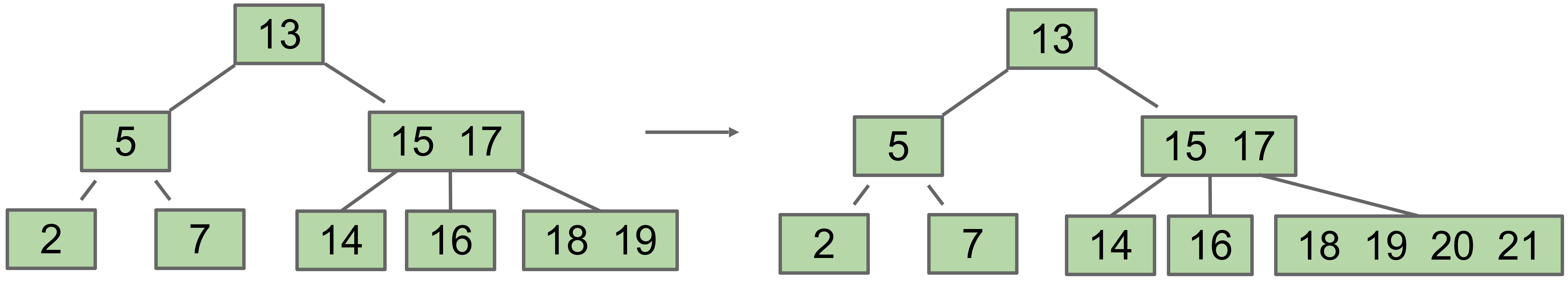
Examining a node costs us O(L) (L is the number of items in a node) compares, but that's OK since L is constant.

What if a non-leaf node gets too full? Can we split that?

- Sure, we'll do this in a few slides, but first…

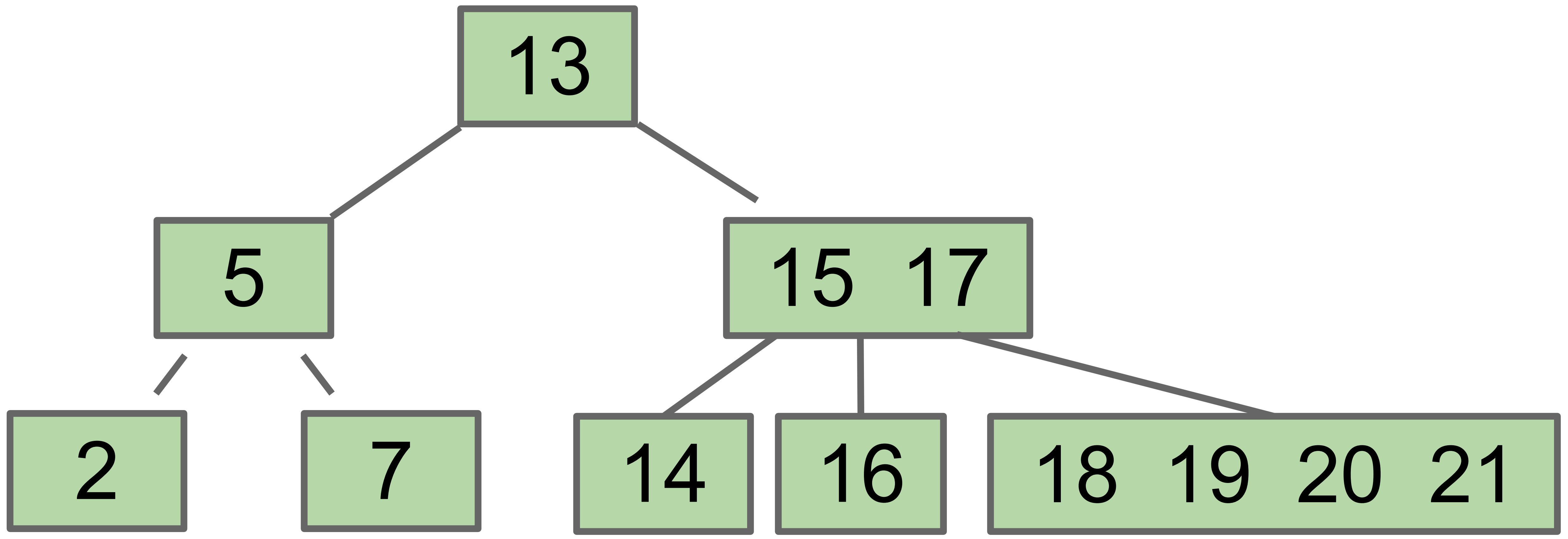
Worksheet time!

- Suppose we add 20, 21:



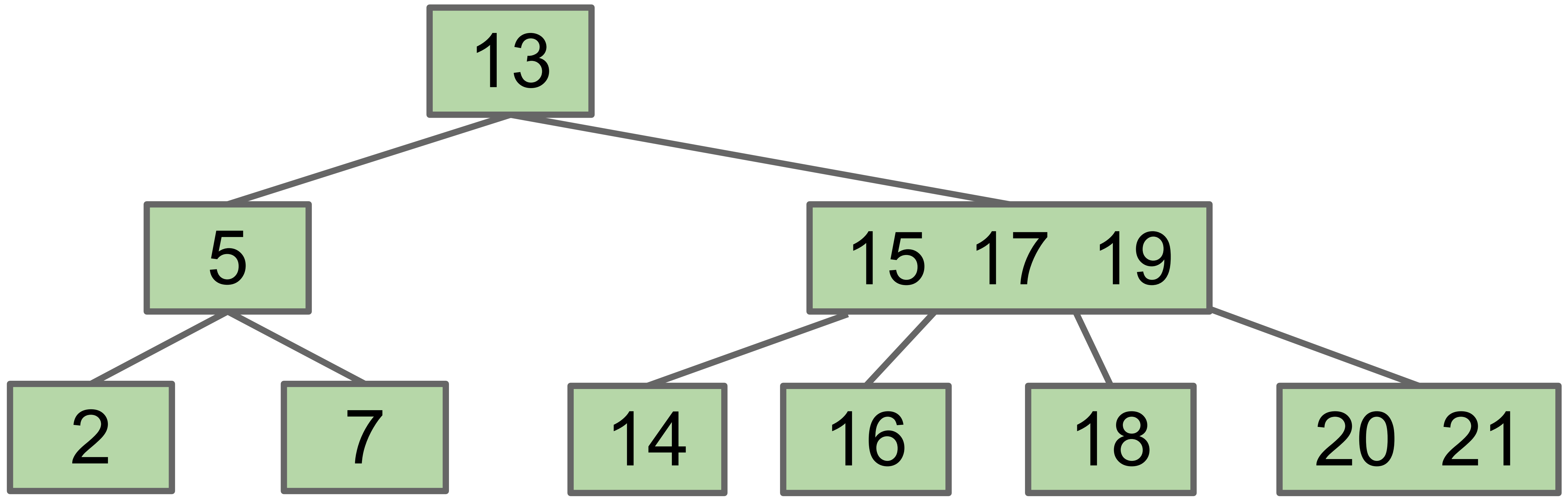
If our cap is at most L=3 items per node, draw post-split tree.

Worksheet answers



Original

move 19 up, then split [18] to the left, [20 21] to the right



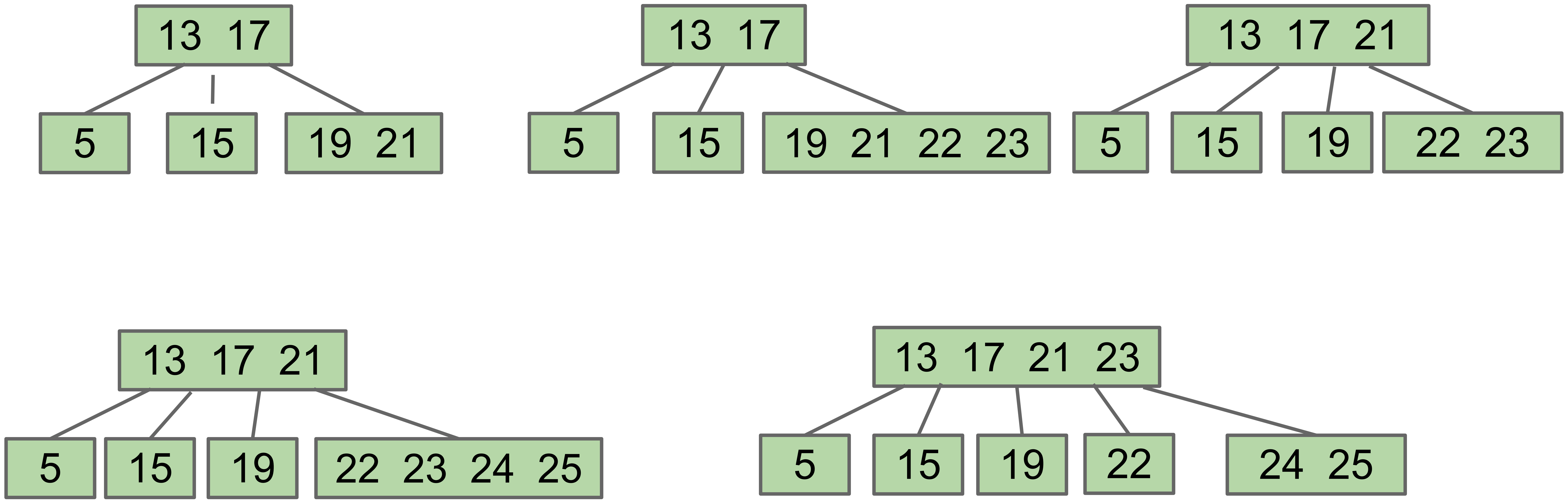
Chain reaction splitting

Insertion: Chain Reaction

Suppose we add 25, 26:

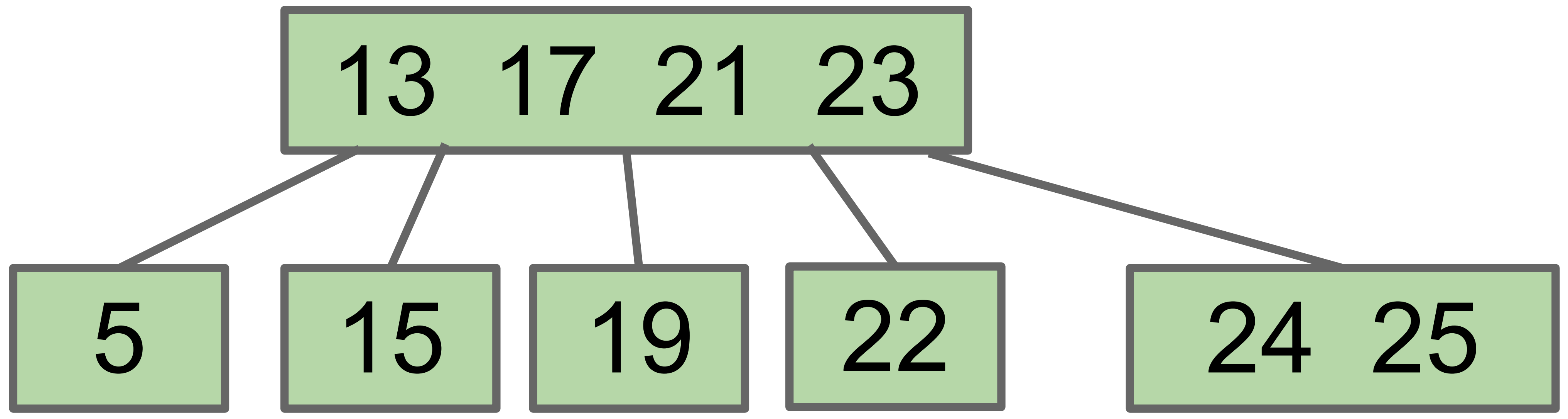
13
5
15 17 19
2
7
14
16
18
20 21
13
5
15 17 19
2
7
14
16
18
20 21 25 26
13
5
15 17 19 21
2
7
14
16
18
20
25 26
13 17
5
15
19 21
2
7
14
16
18
20
25 26

What Happens If The Root Is Too Full?



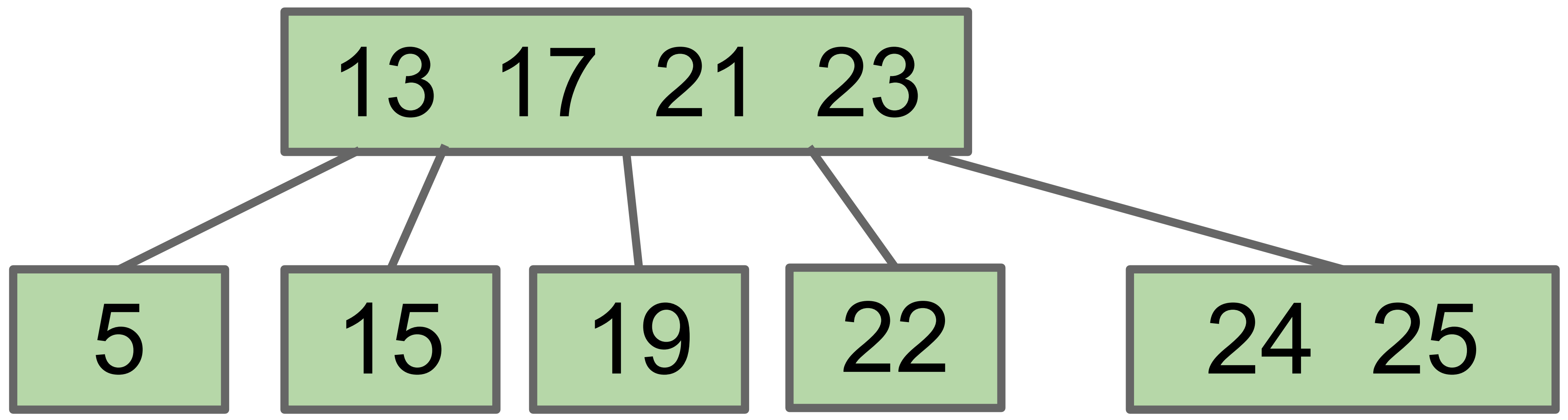
Draw the tree after the root is split.

Worksheet time!



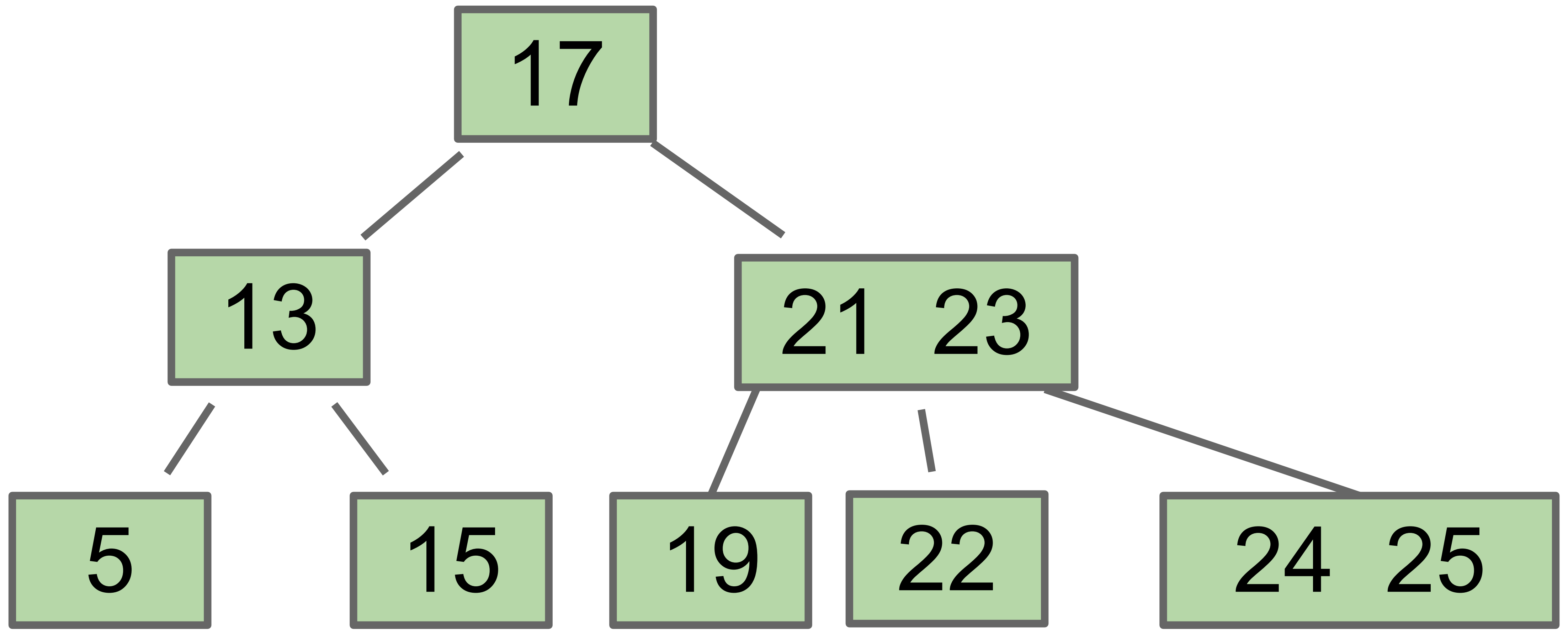
- Draw the tree after the root is split.

Worksheet answers



Original

17 becomes the new root
its left child is 13
its right child is [21, 23]
we recursively bring along their children, making sure they are in the proper left/right place

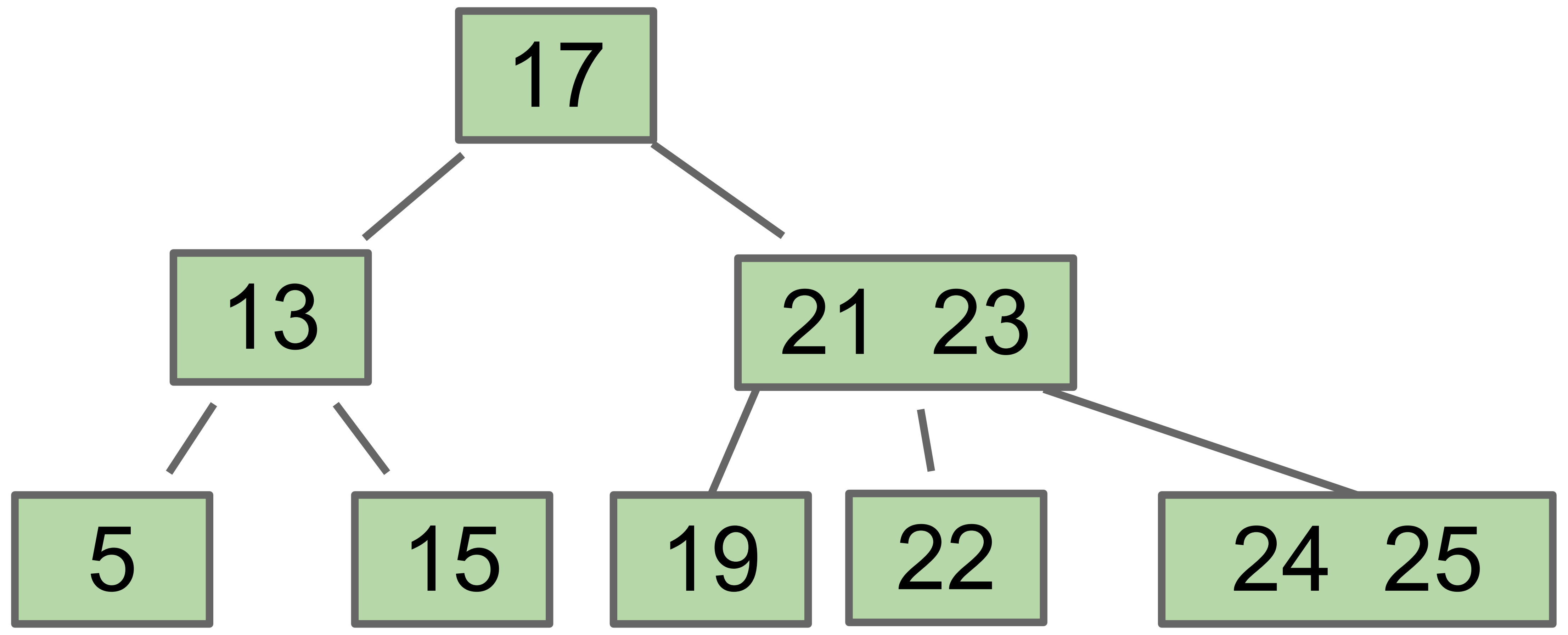


Perfect Balance

Observation: These splitting trees have perfect balance.

- If we split the root, every node gets pushed down by exactly one level.
- If we split a leaf node or internal node, the height doesn't change.

All operations have guaranteed O(log N) time (more details soon).



B-tree terminology

The Real Name for Splitting Trees is “B Trees”

B-trees

- B-trees of order L=3 (like we used today) are also called a 2-3-4 tree or a 2-4 tree.
  - “2-3-4” refers to the number of children that a node can have, e.g. a 2-3-4 tree node may have 2, 3, or 4 children.
- B-trees of order L=2 are also called a 2-3 tree.
- B for bushy? Balanced?

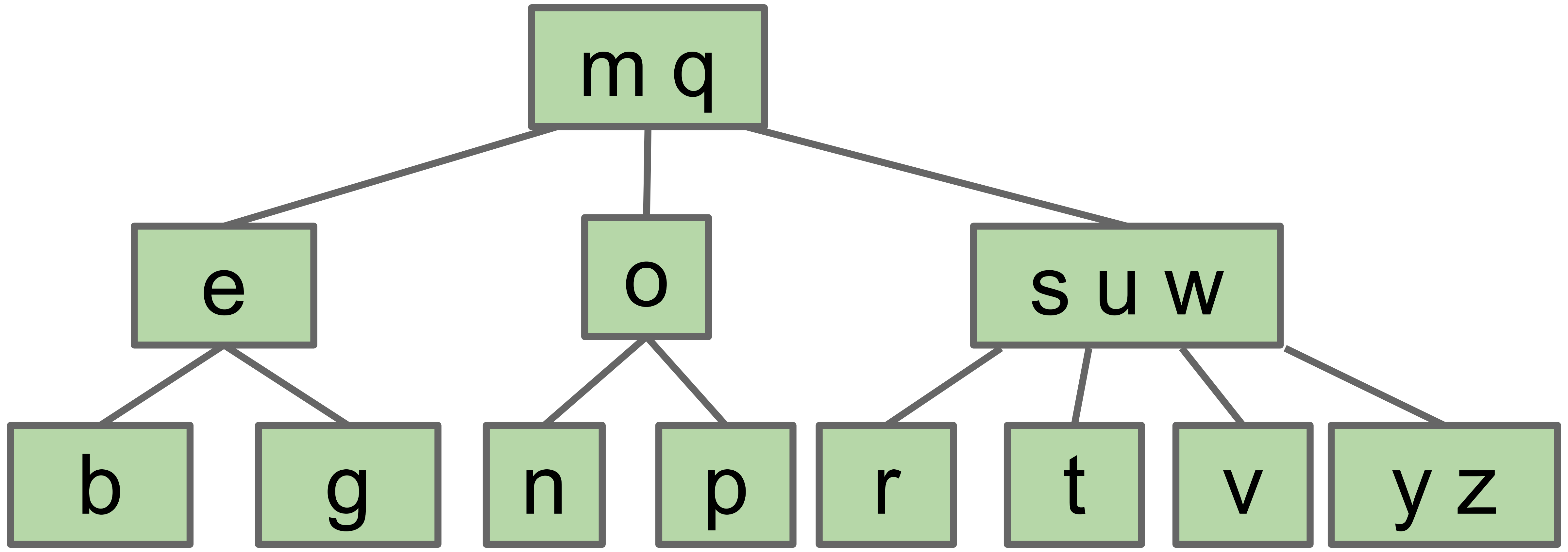
The origin of "B-tree" has never been explained by the authors. As we shall see, "balanced," "broad," or "bushy" might apply. Others suggest that the "B" stands for Boeing. Because of his contributions, however, it seems appropriate to think of B-trees as "Bayer"-trees.

- Douglas Corner (The Ubiquitous B-Tree)

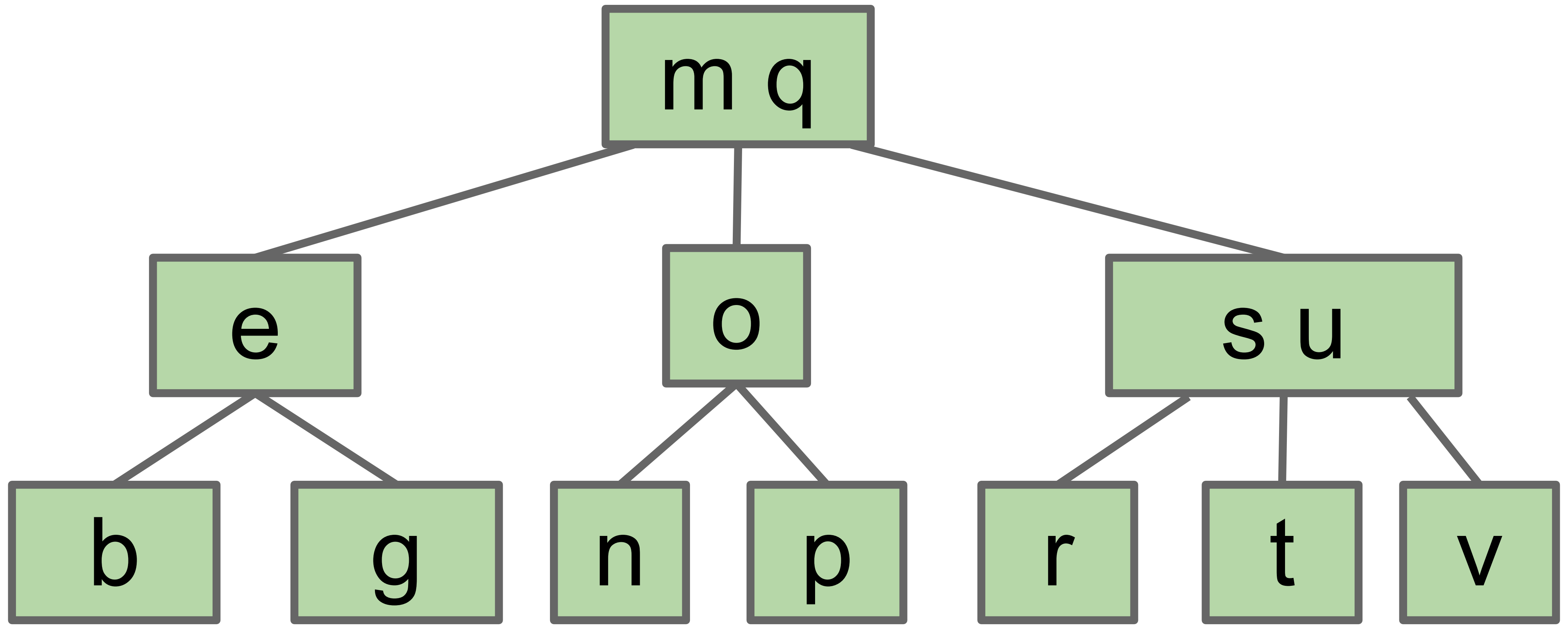
B-Trees

B-Trees are most popular in two specific contexts:

- Small L (L=2 or L=3):
  - Used as a conceptually simple balanced search tree (as today).
- L is very large (say thousands).
  - Used in practice for databases and filesystems (i.e. systems with very large records).



2-3-4 a.k.a. 2-4 Tree (L=3):
Max 3 items per node.
Max 4 non-null children per node.

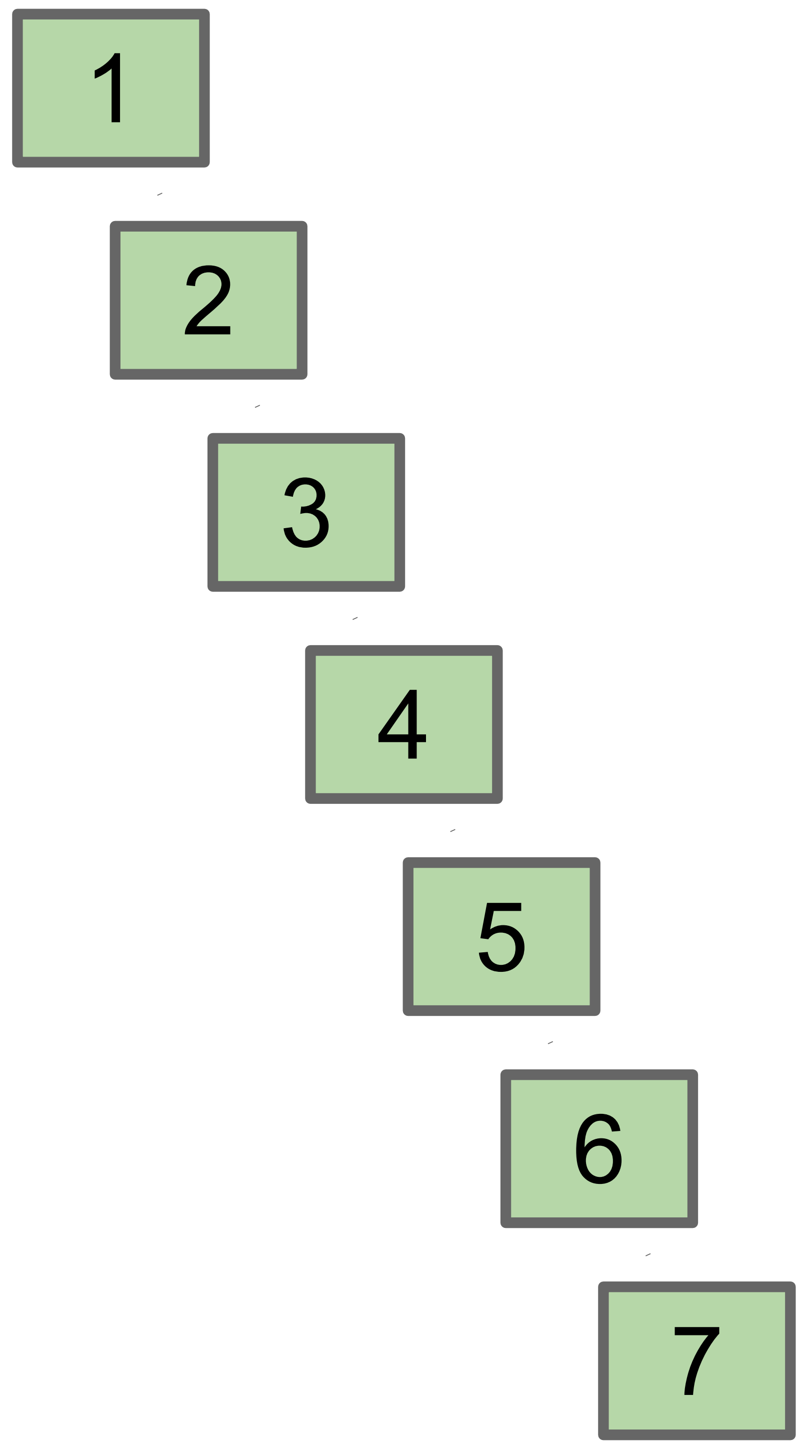


2-3 Tree (L=2):
Max 2 items per node.
Max 3 non-null children per node.

Construction

Going back to our motivation...

Add the numbers 1, 2, 3, 4, 5, 6, then 7 (in that order) into a regular BST.

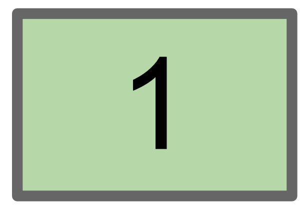


Resulting BST is spindly.

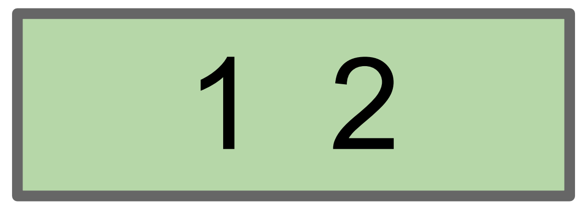
Exercise together

Add 1, 2, 3, 4, 5, 6, 7 (in that order) into a **2-3** tree. For L=2, pass the middle item up.

Adding 1.

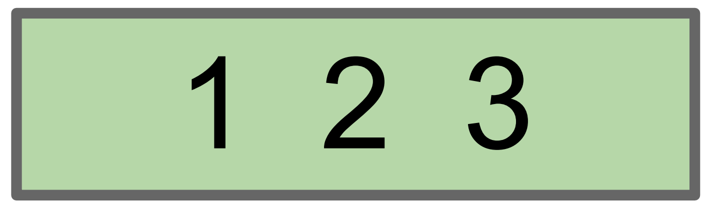


Adding 2.

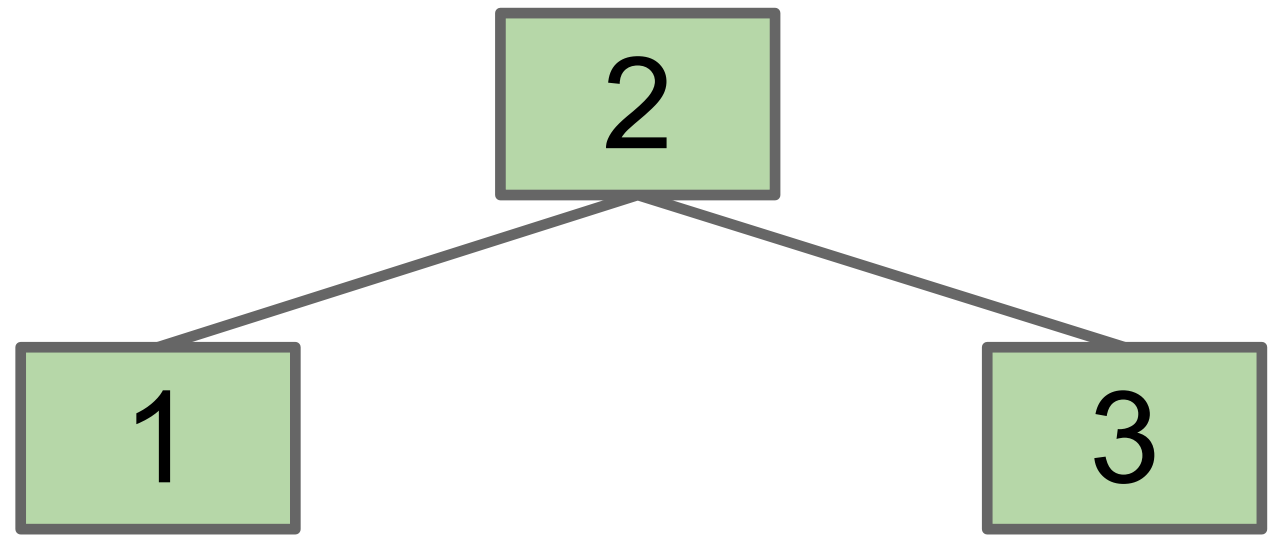


Adding 3. Node is too full, so we need to split.

Before split



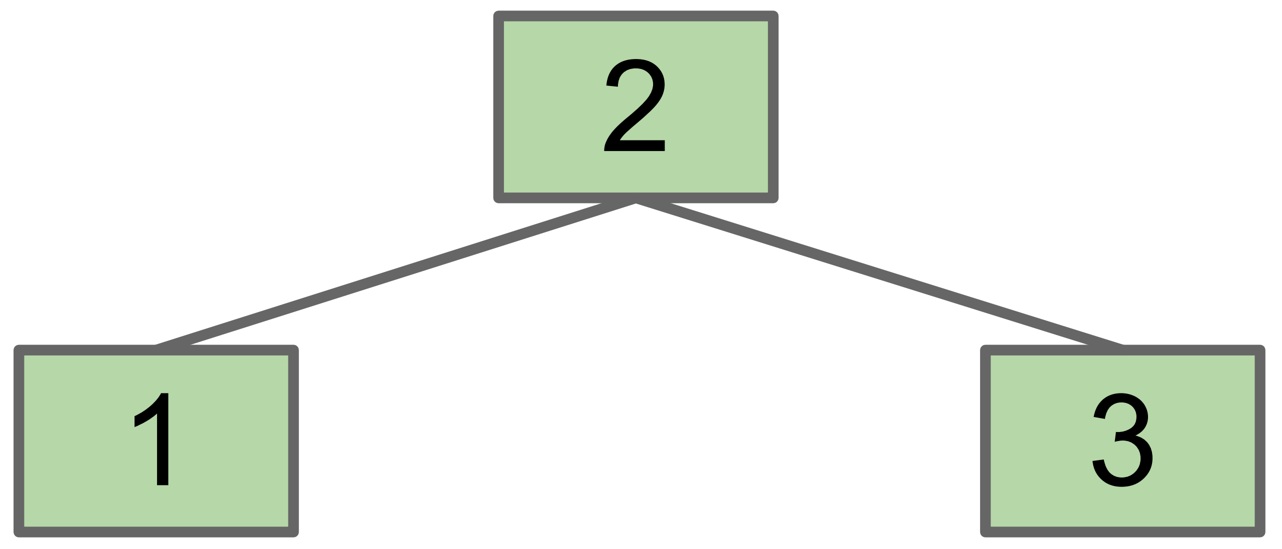
After split



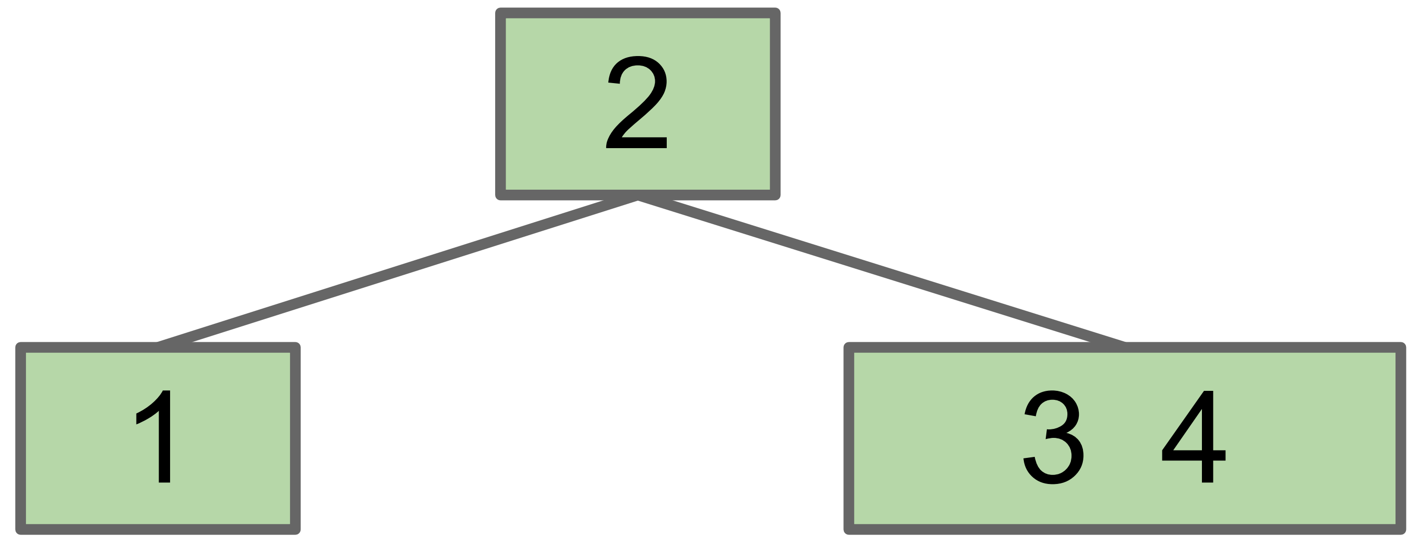
Exercise together

Add 1, 2, 3, 4, 5, 6, 7 (in that order) into a 2-3 tree. For L=2, pass the middle item up.

After adding 3.

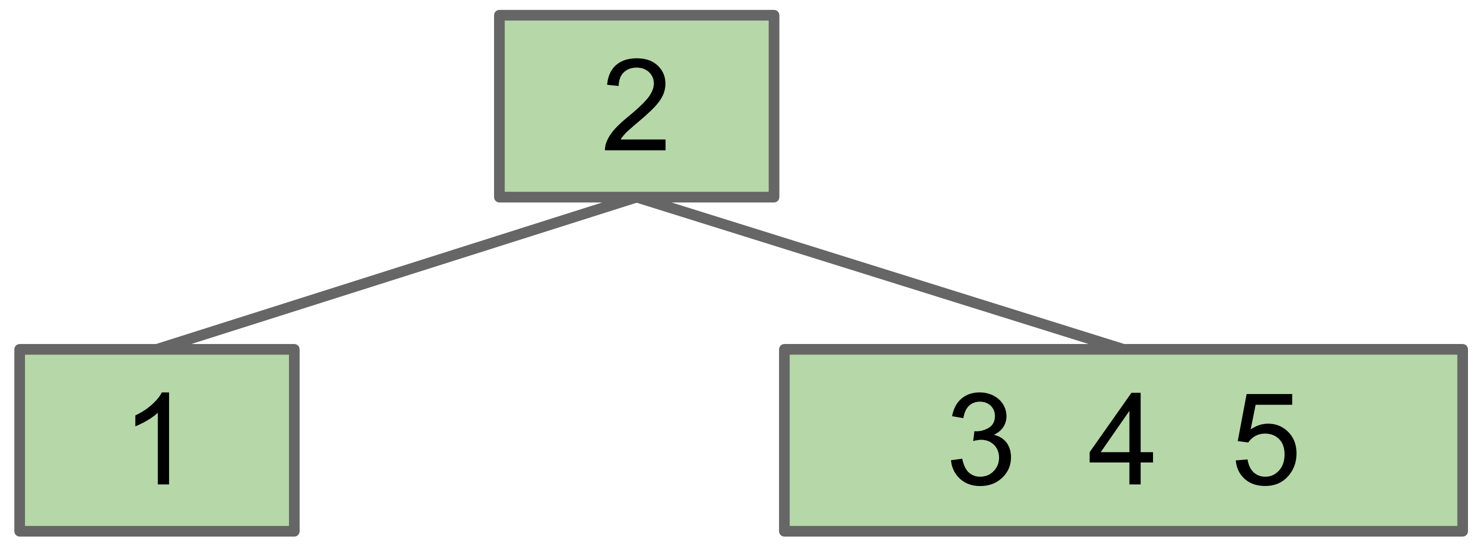


Adding 4.

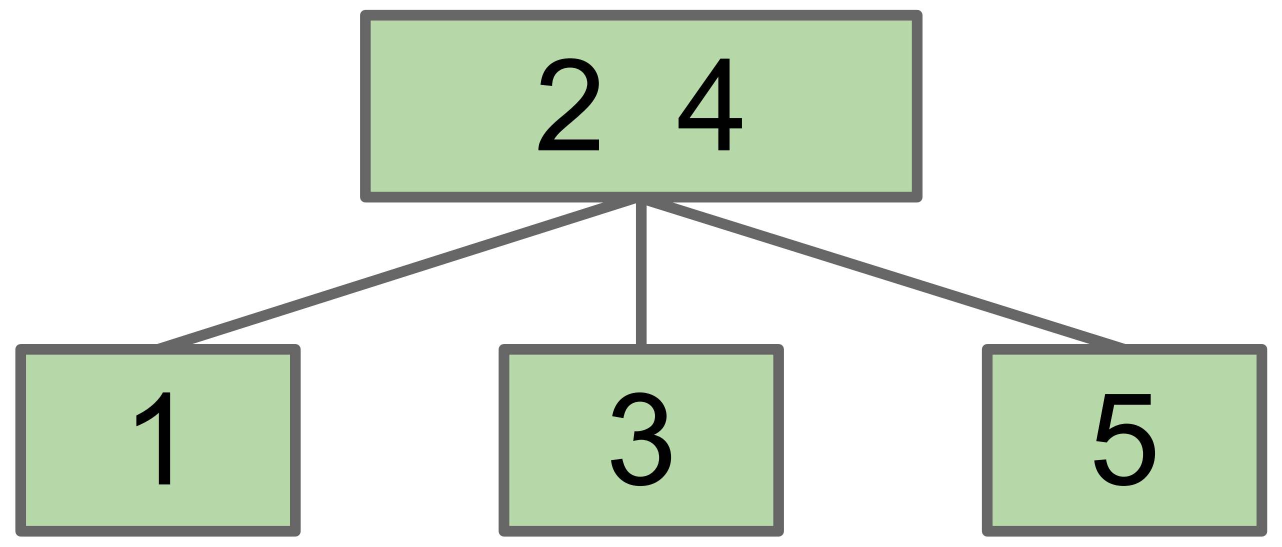


Adding 5. Node is too full, so we need to split.

Before split



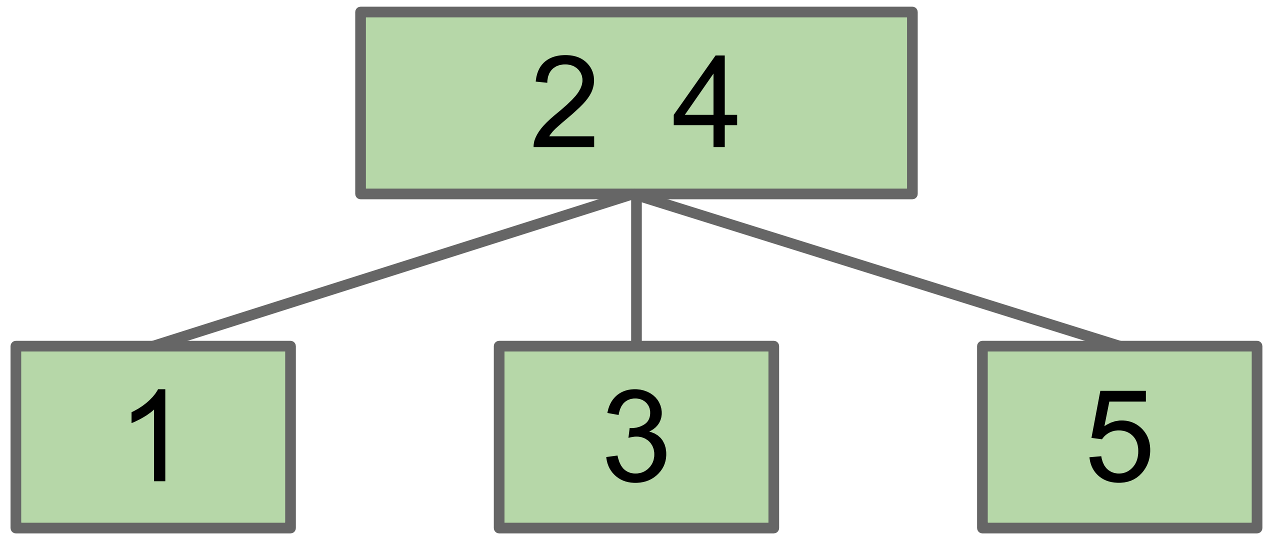
After split



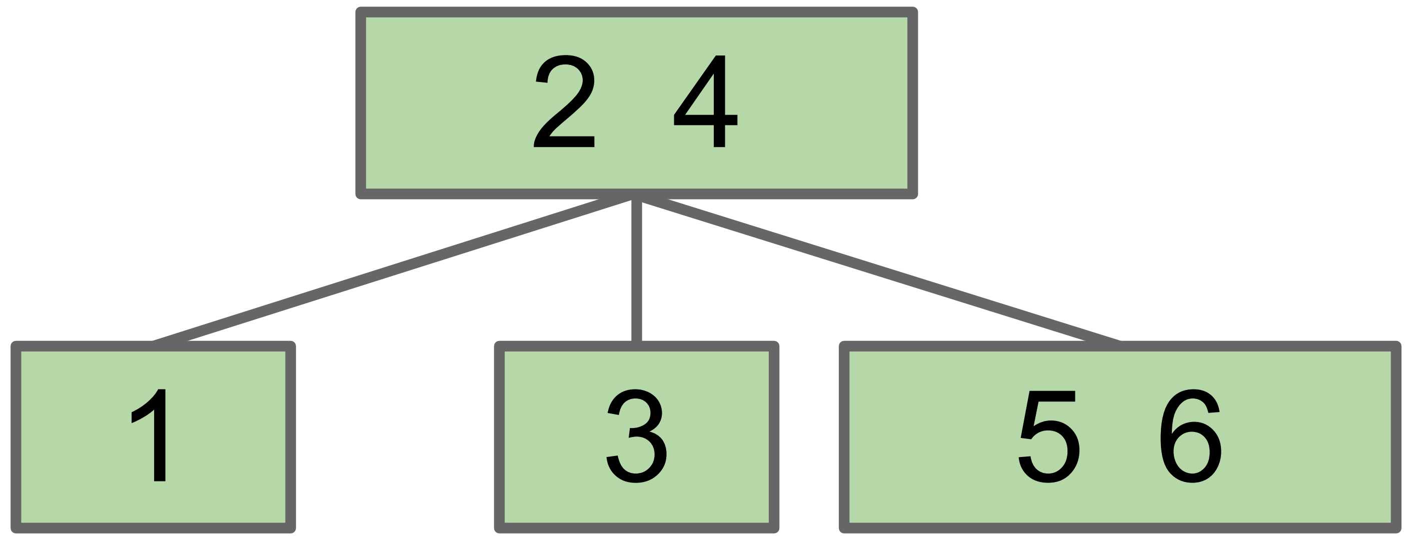
Exercise together

Add 1, 2, 3, 4, 5, 6, 7 (in that order) into a 2-3 tree. For L=2, pass the middle item up.

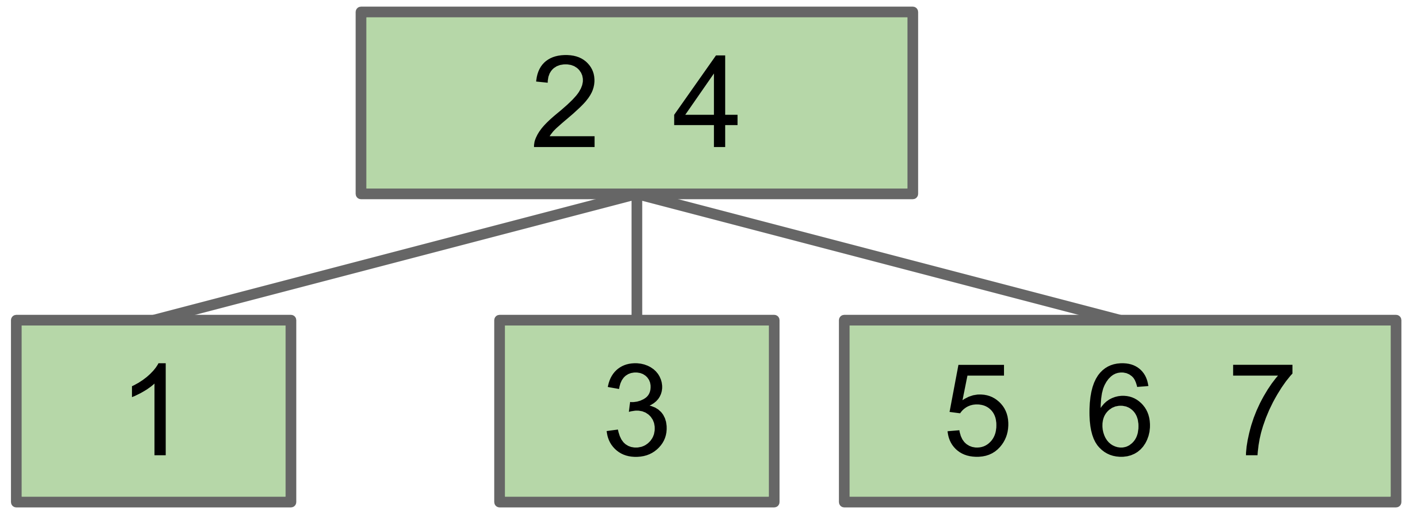
After adding 5.



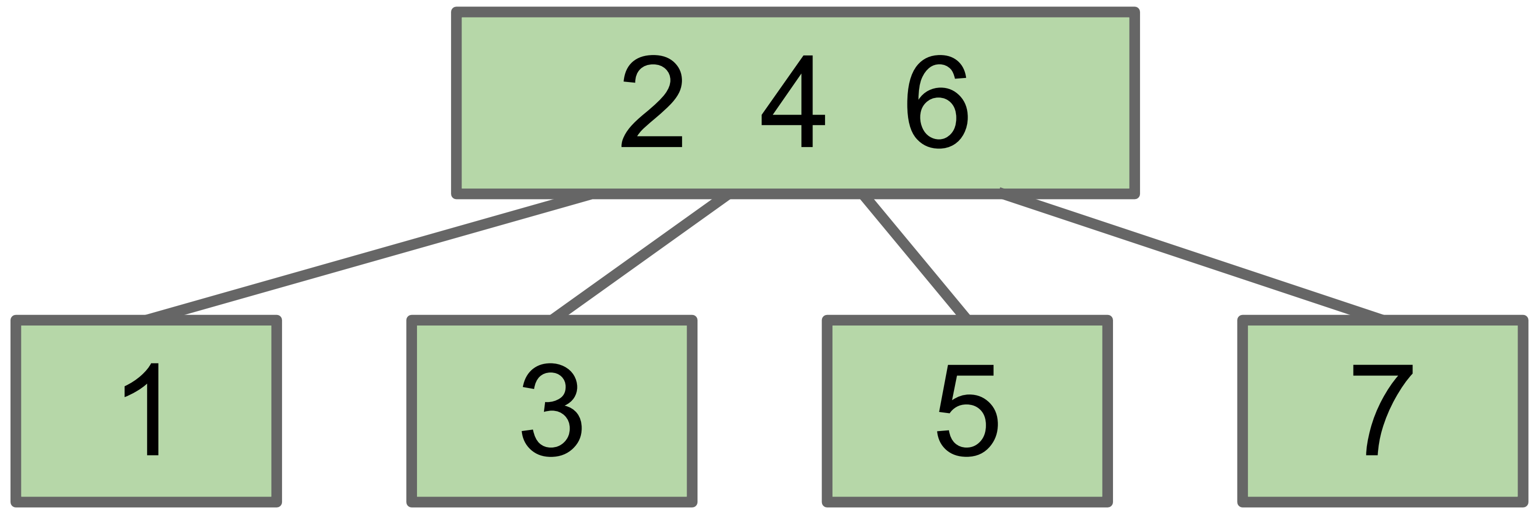
Adding 6.



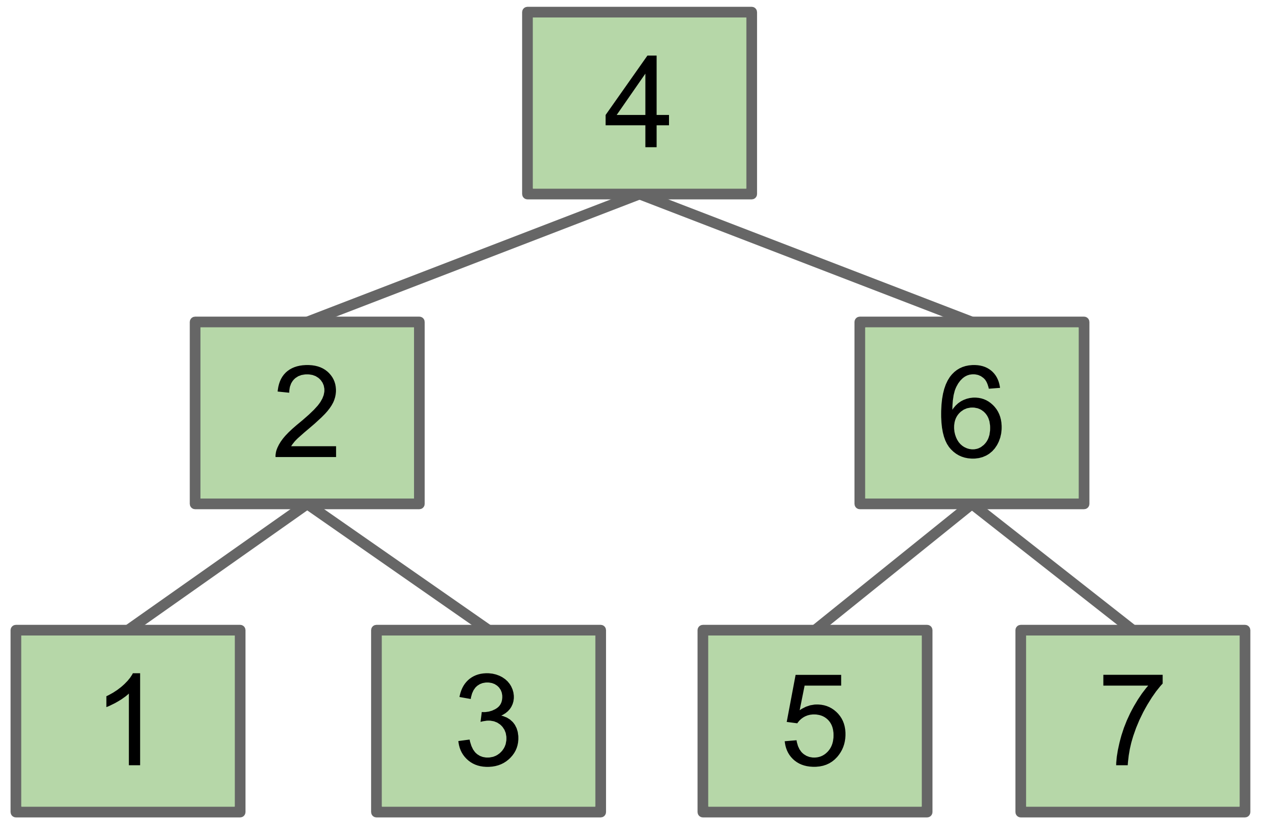
Adding 7. Node is too full, so we need to split.



Before split



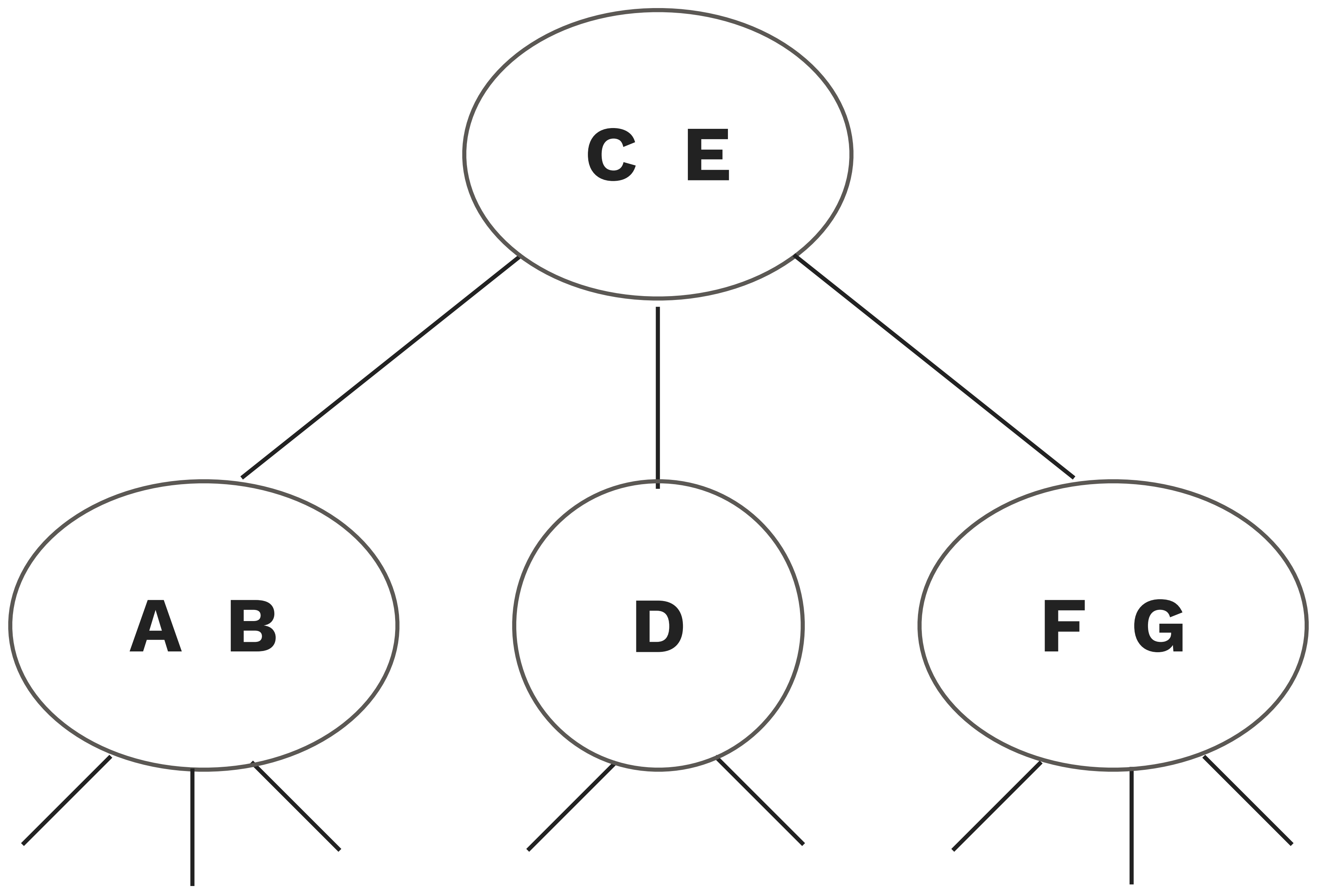
After one split



After two splits

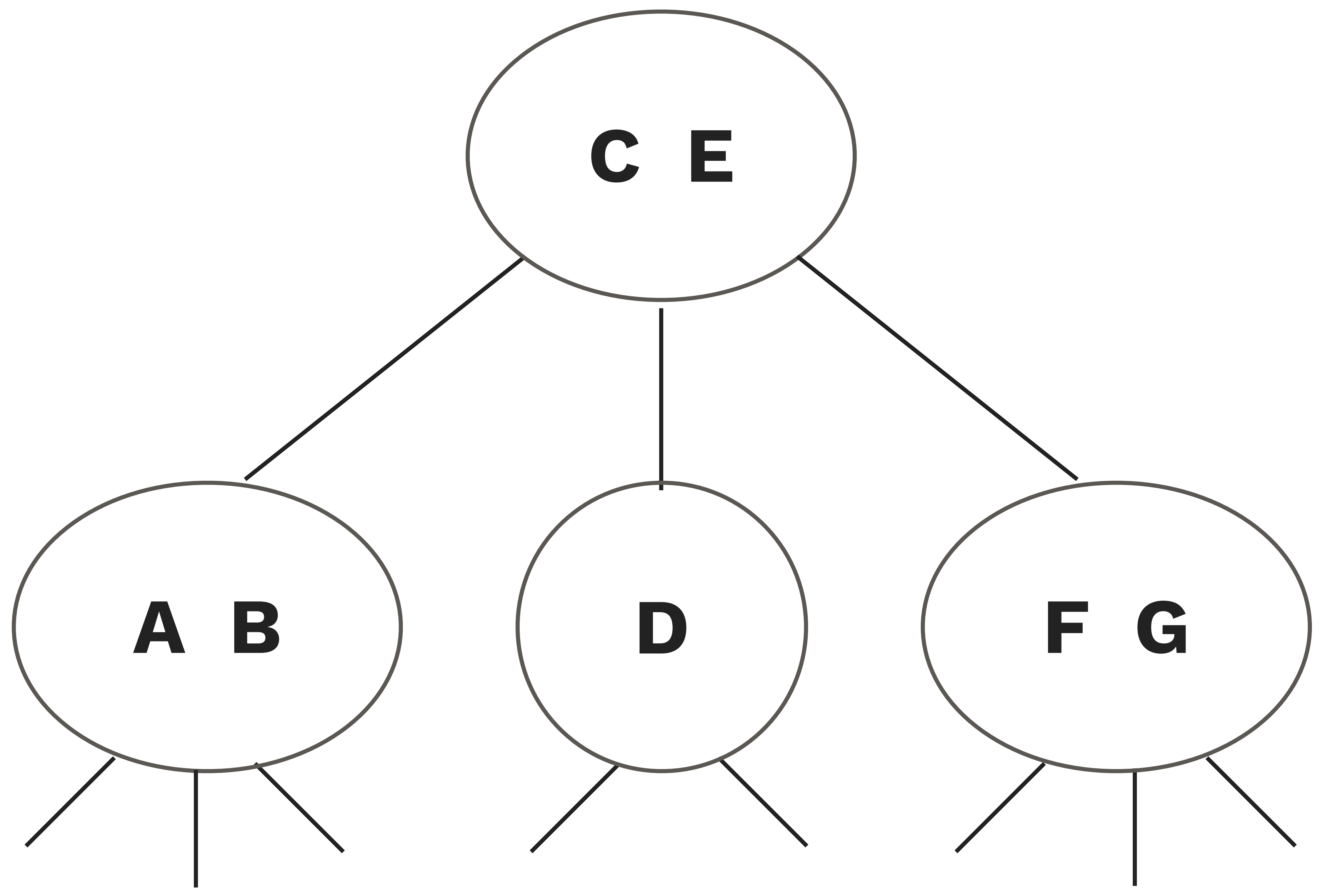
Worksheet time!

- Given the following 2-3-4 tree, insert keys H, I, J, K.



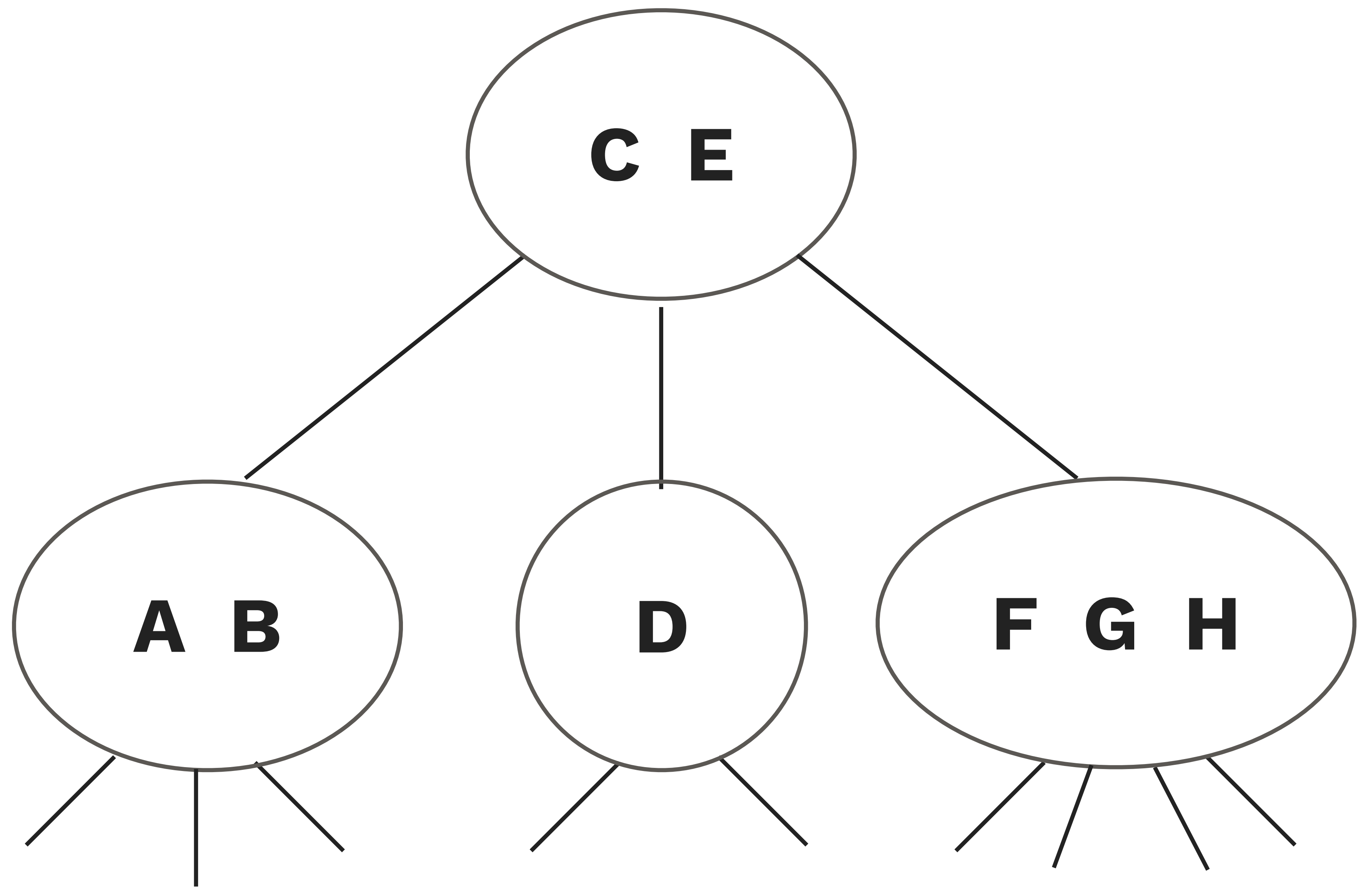
Note, here we have explicitly visualized the null child nodes

Worksheet answer – H

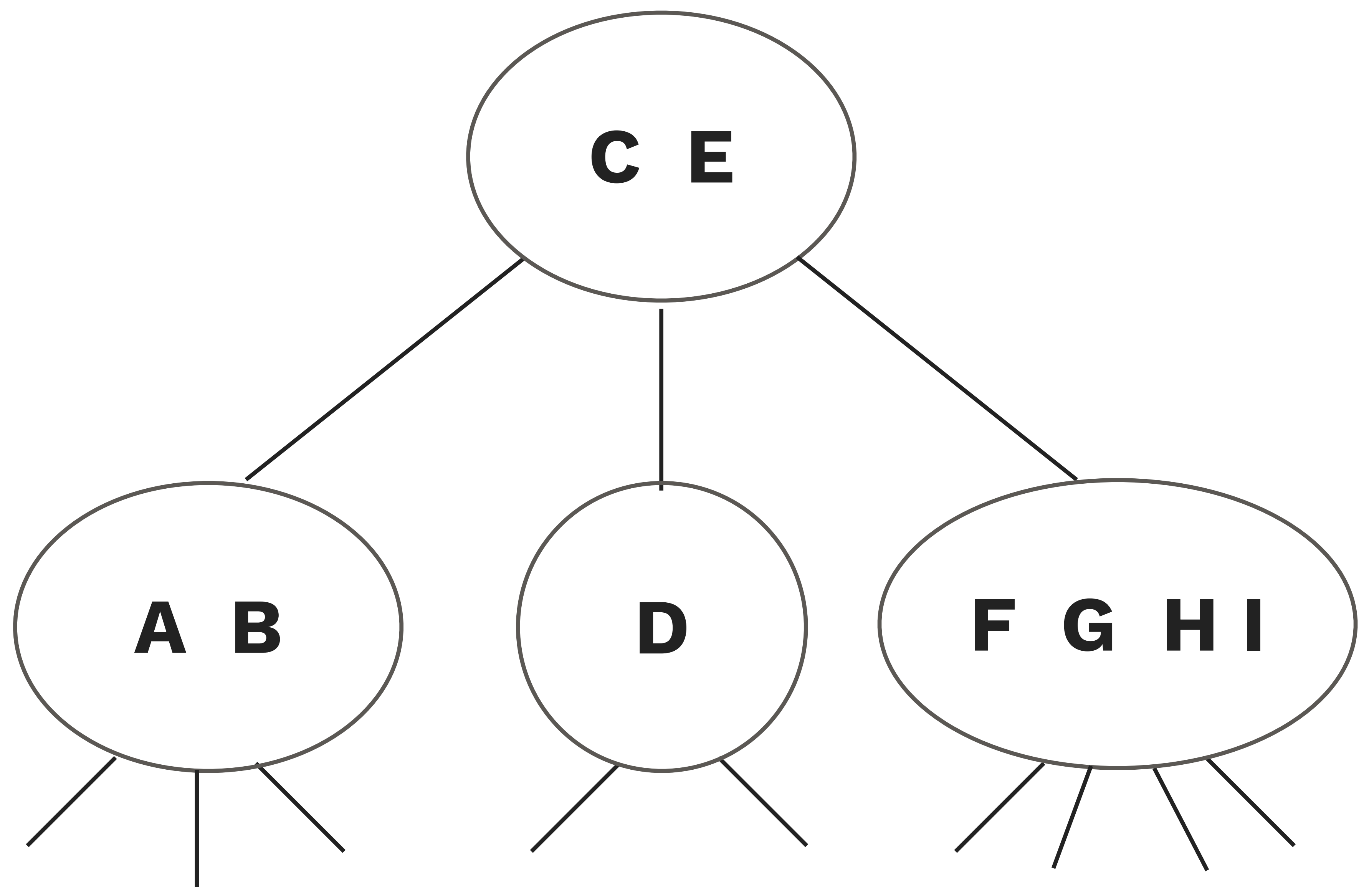


Insert H at the end.

No splitting needed.

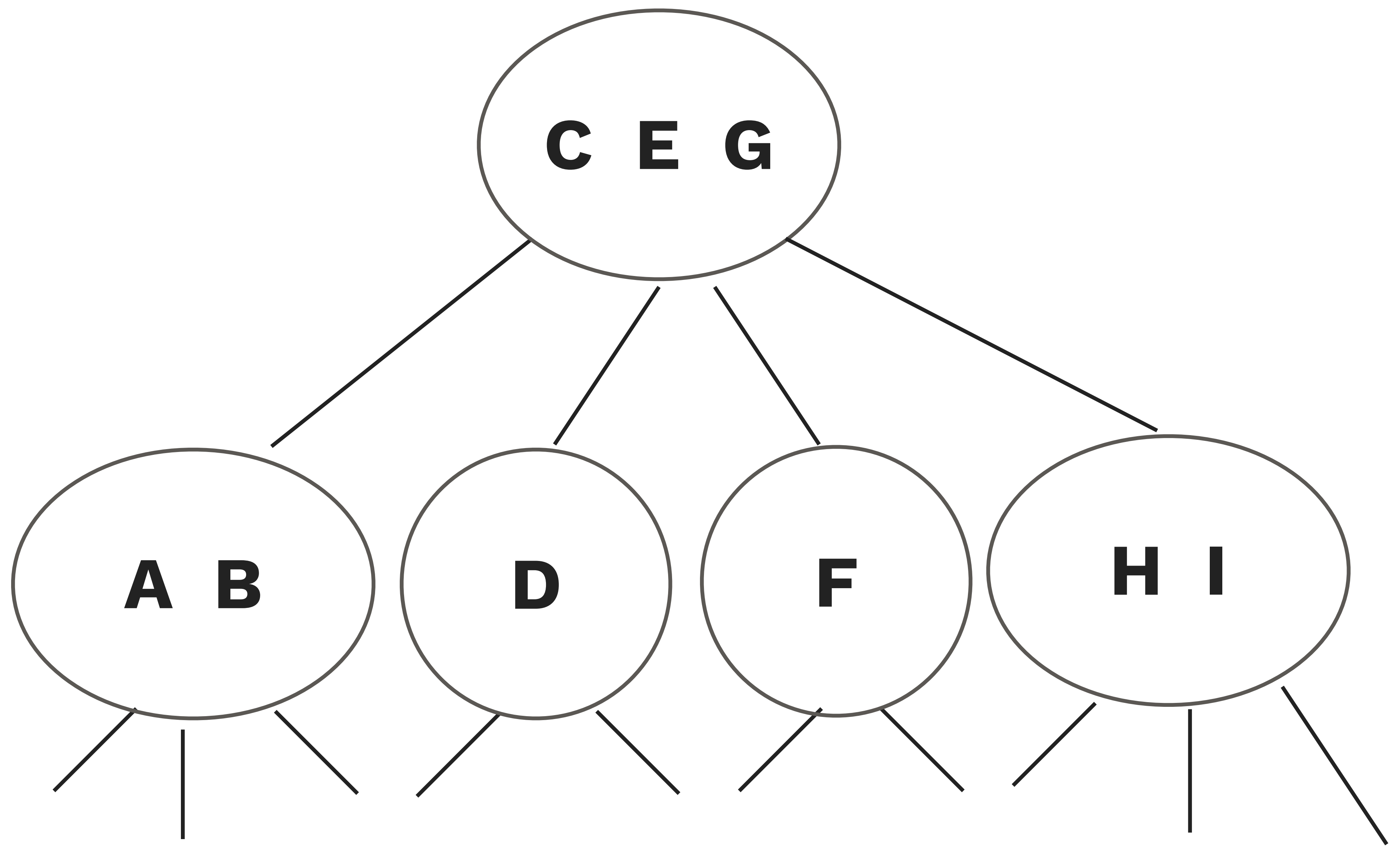


Worksheet answer - I

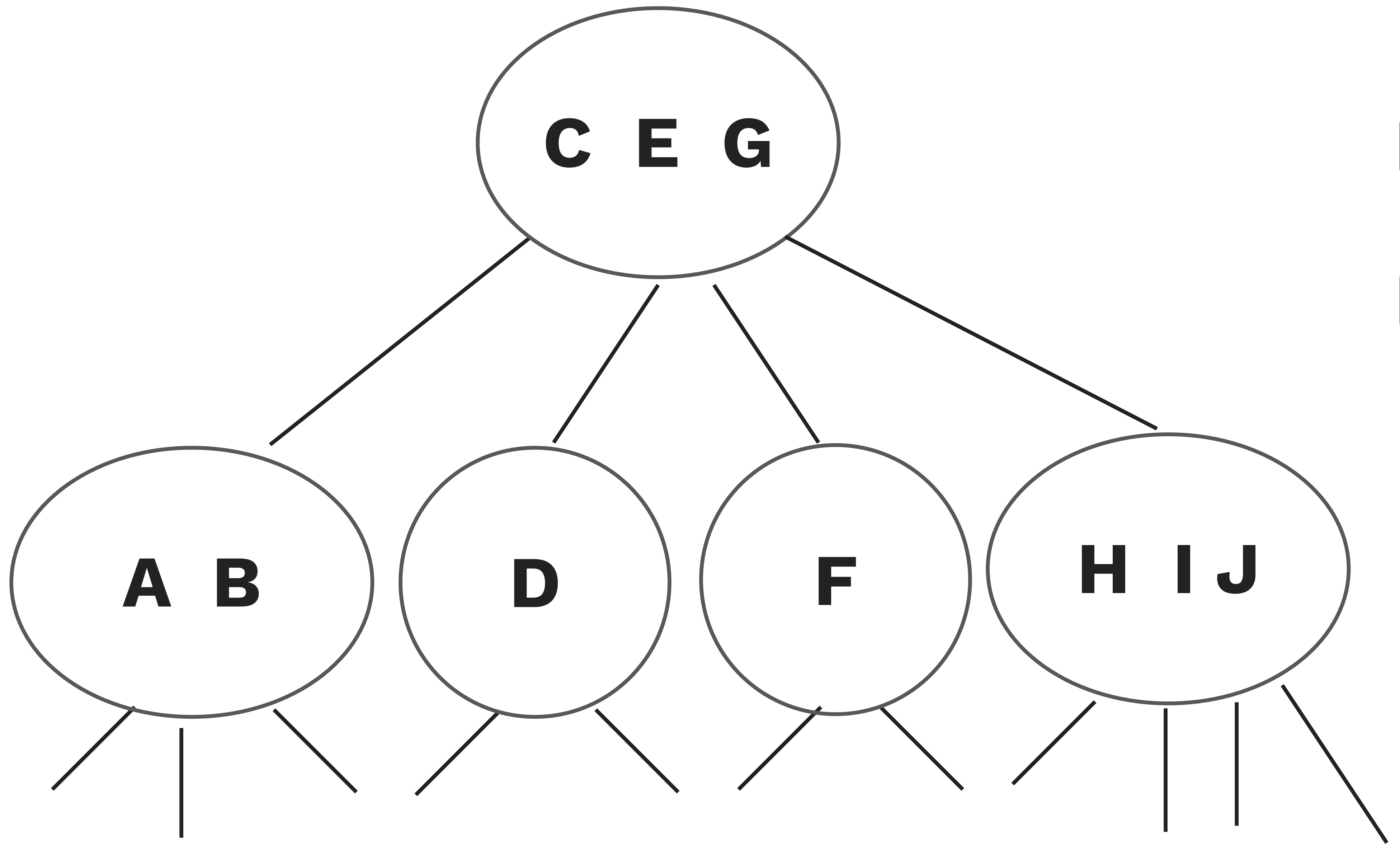


Insert I at the end.

Split and move up G.



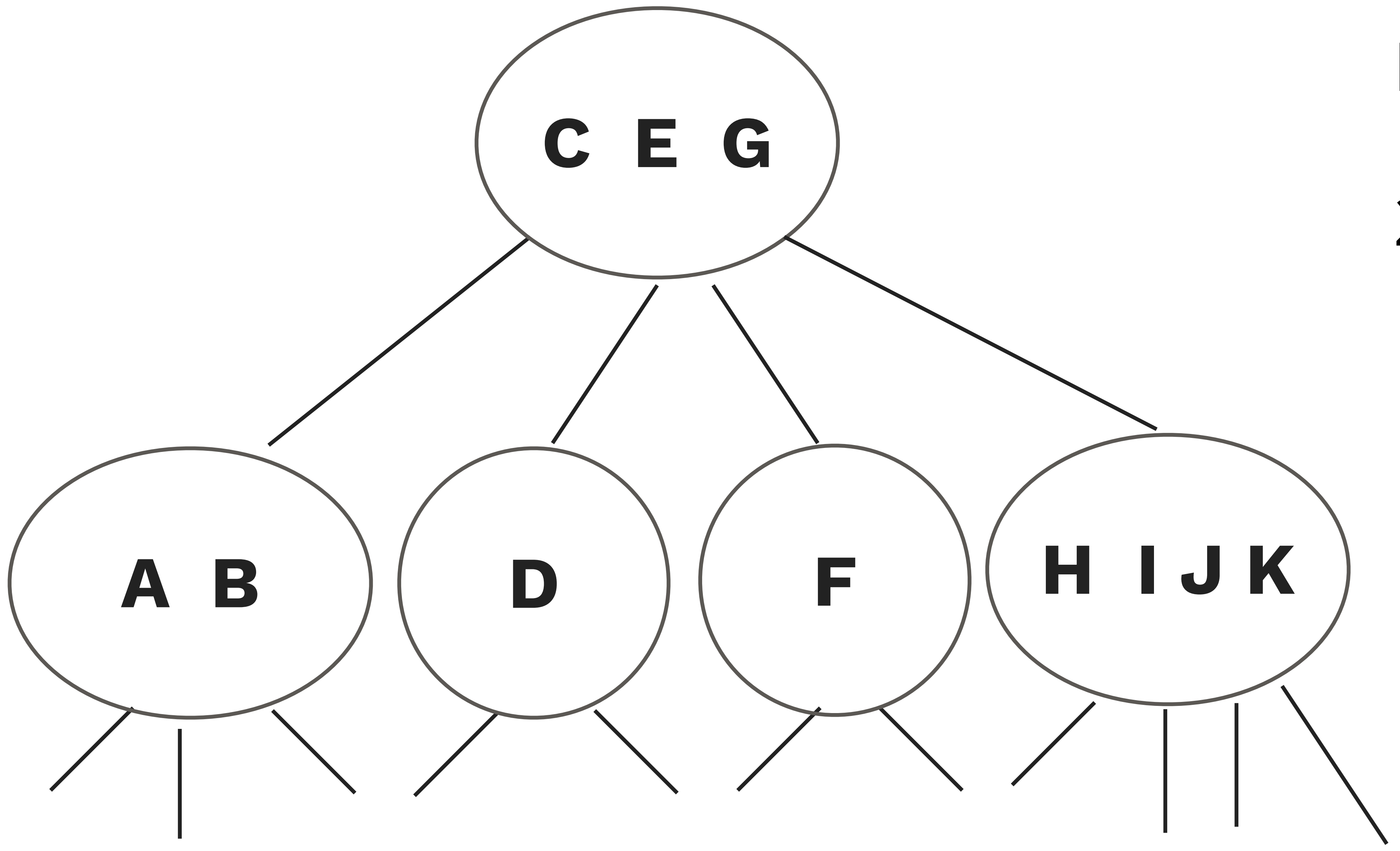
Worksheet answer – J



Insert J at the end.

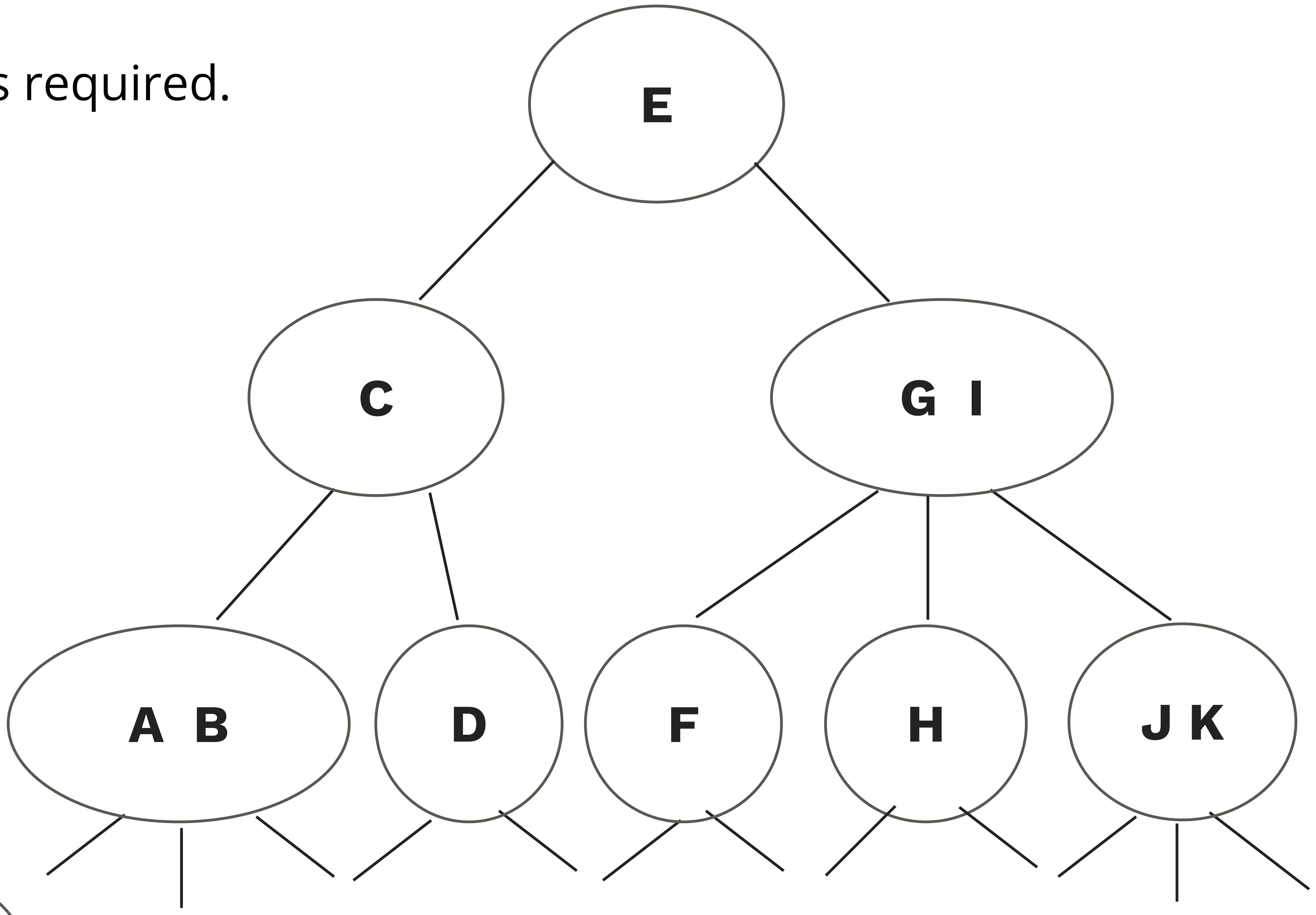
No splitting needed.

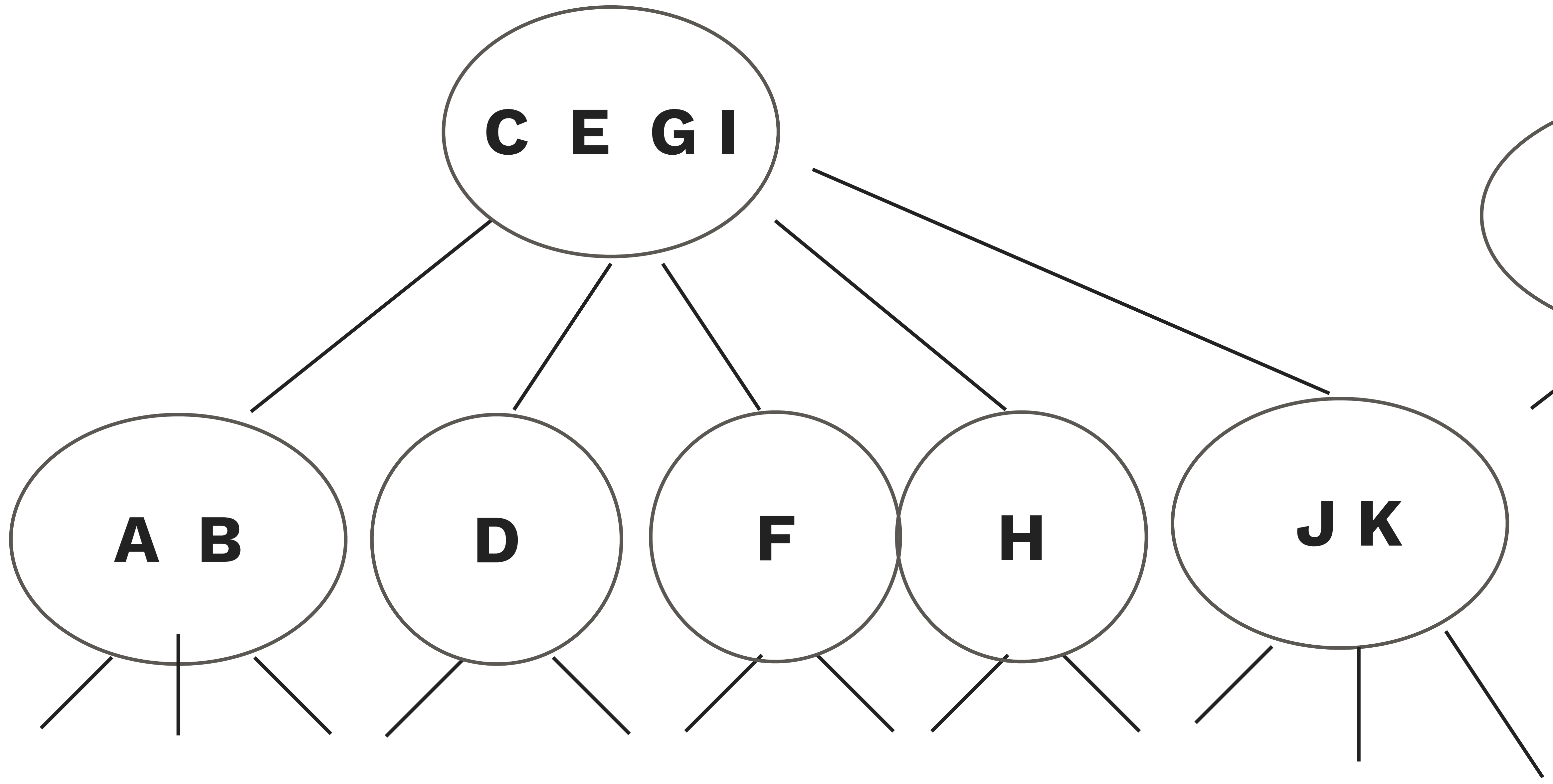
Worksheet answer – K



Insert K at the end.

2 splits required.

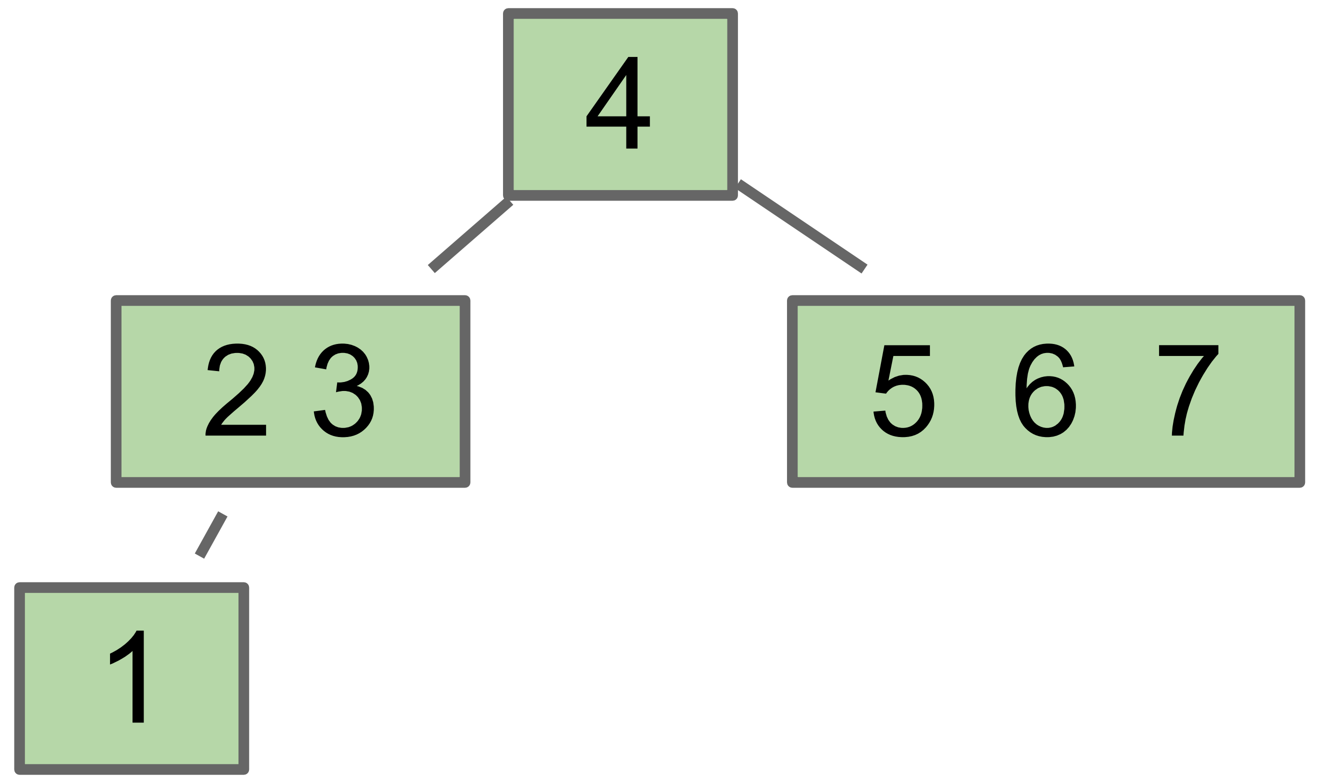




B-Tree Invariants

Because of the way B-Trees are constructed, we get two nice invariants:

- All **leaves** must be the **same distance from the root.**
- A non-leaf node with k items must have exactly **k+1 children**.
- Example: The tree given below is impossible. Why?
  - Leaves ([1] and [5 6 7]) are a different distance from the source.
  - Non-leaf node [2 3] has two items but only only one child. Should have three children.

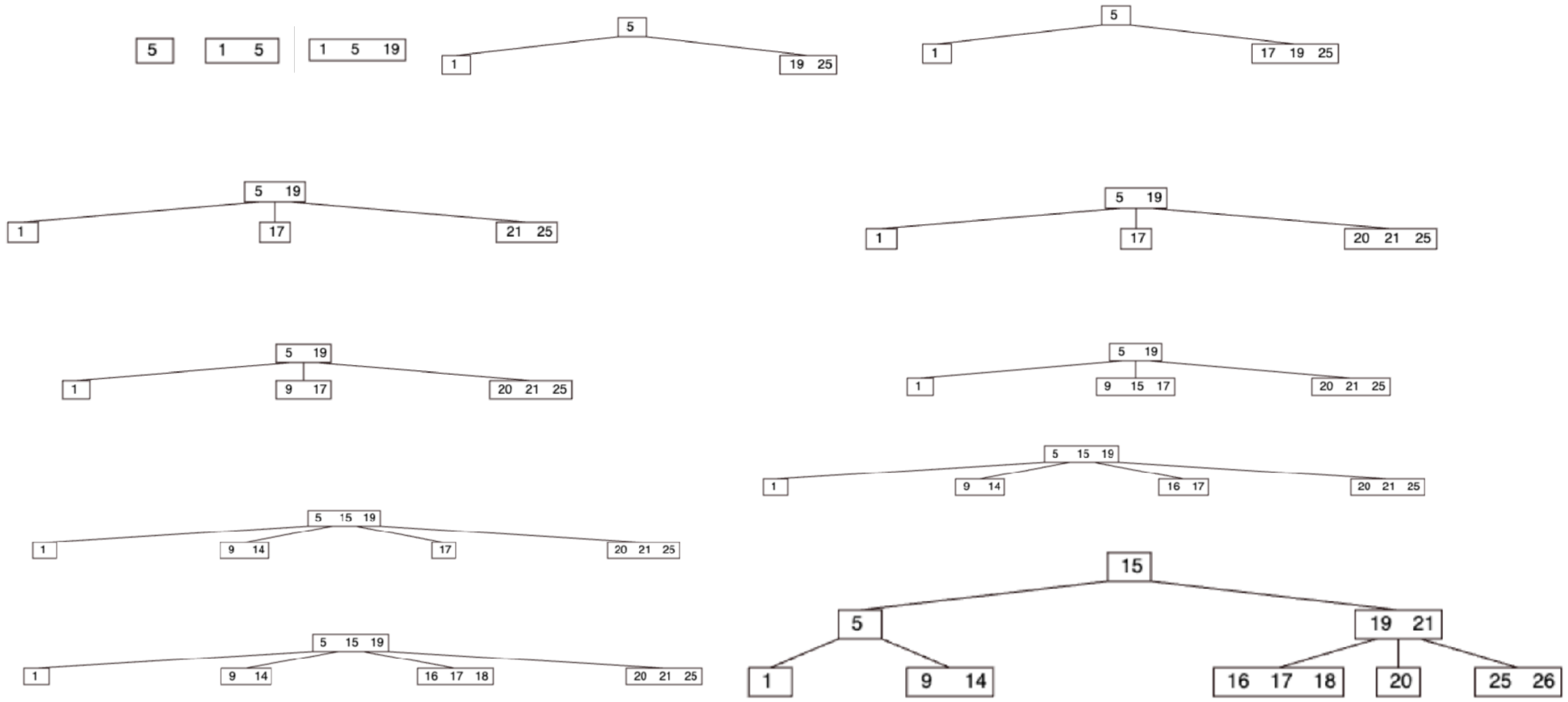


We have not proven these invariants rigorously, but try thinking them through. These invariants guarantee that our trees will be bushy.

Worksheet time!

- Draw the 2-3-4 tree that results when you insert the keys:
  5, 1, 19, 25, 17, 21, 20, 9, 15, 14, 16, 18, 26 in that order in an initially empty tree.

Worksheet answers



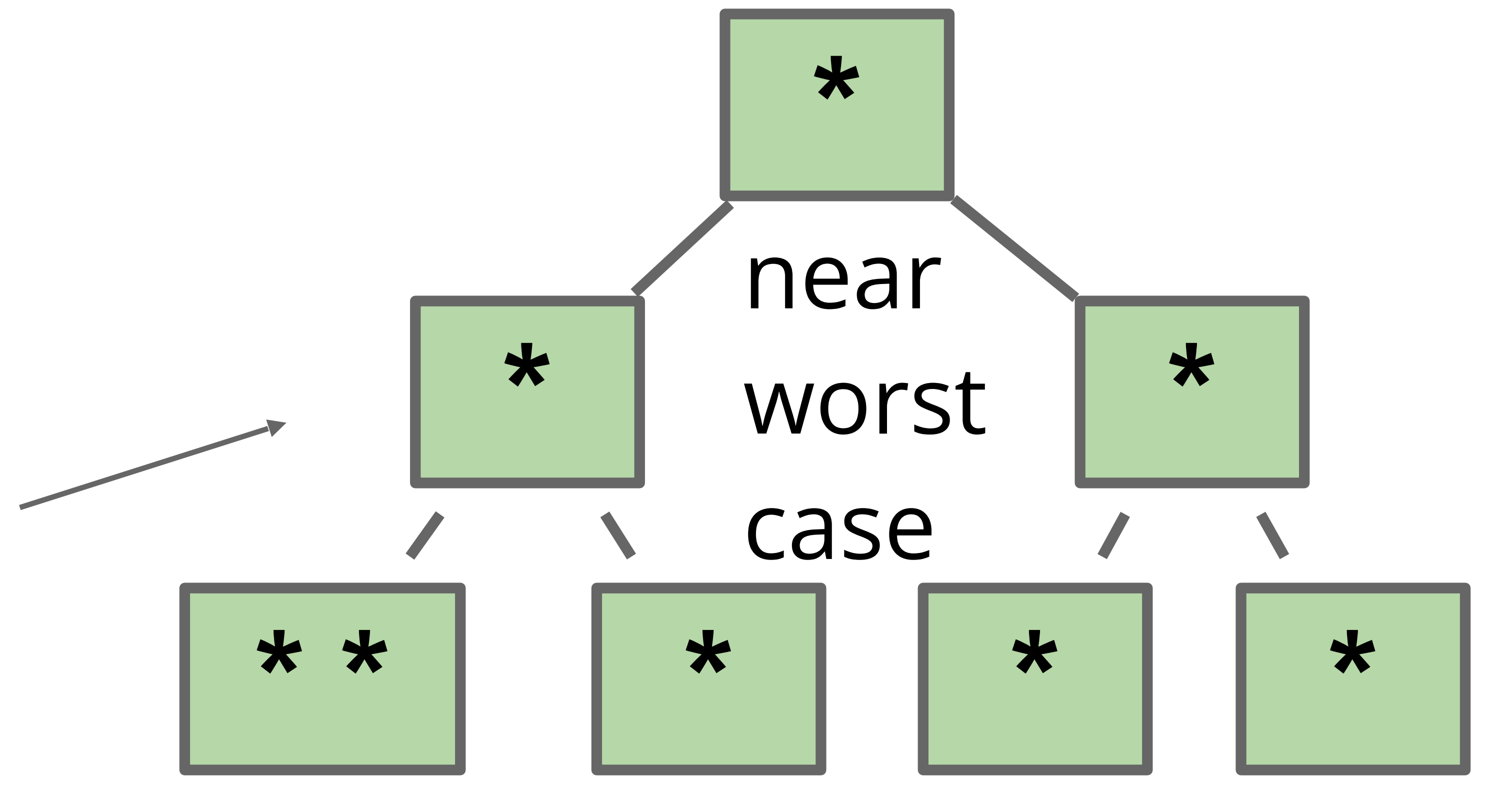
Performance

Height of a B-Tree with Limit L

L: Max number of items per node.

Height: Between ~$\log_{L+1}(N)$ and ~$\log_2(N)$

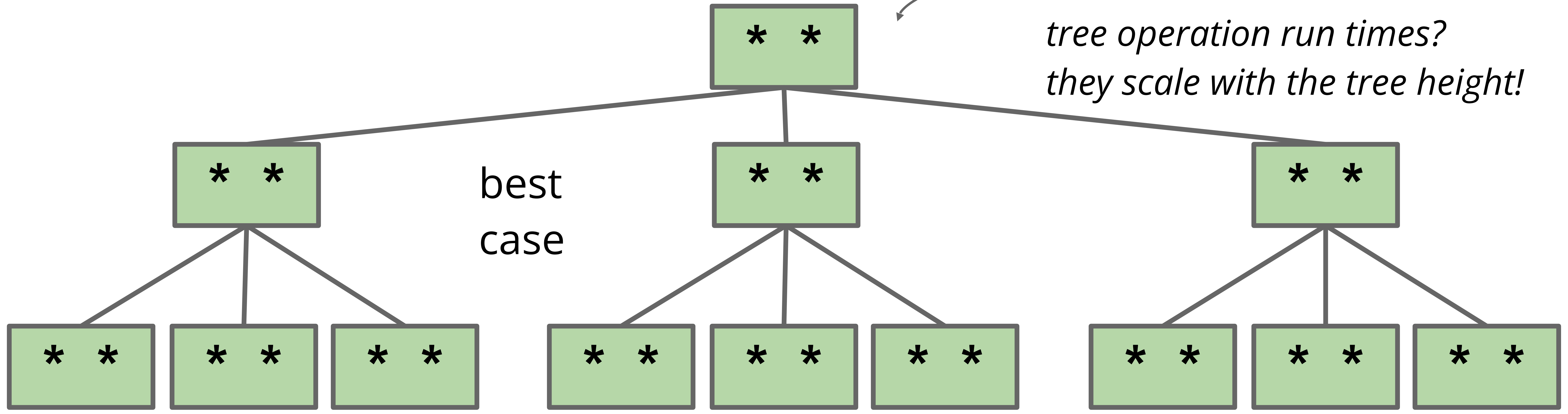
- Largest possible height is all non-leaf nodes have 1 item.
- Smallest possible height is all nodes have L items.
- Overall height is therefore $\Theta(\log N)$.



N: 8 items
L: 2 max per node
H: 2

Height grows with $\log_2(N)$

...and what do we remember about tree operation run times? they scale with the tree height!



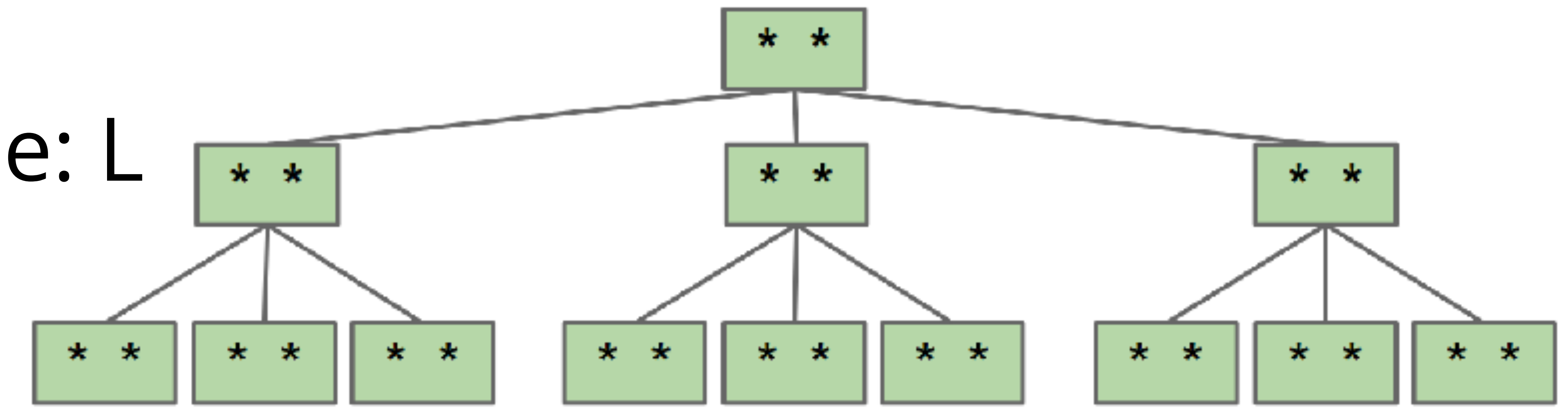
N: 26 items
L: 2 max per node
H: 2

Height grows with log3(N)

Runtime for `find`

Runtime for find/contains/search:

- Worst case number of nodes to inspect: H + 1
- Worst case number of items to inspect per node: L
- Overall runtime: O(HL)



Since H = Θ(log N), overall runtime is O(L log N).

- Since L is a constant, runtime is therefore O(log N).

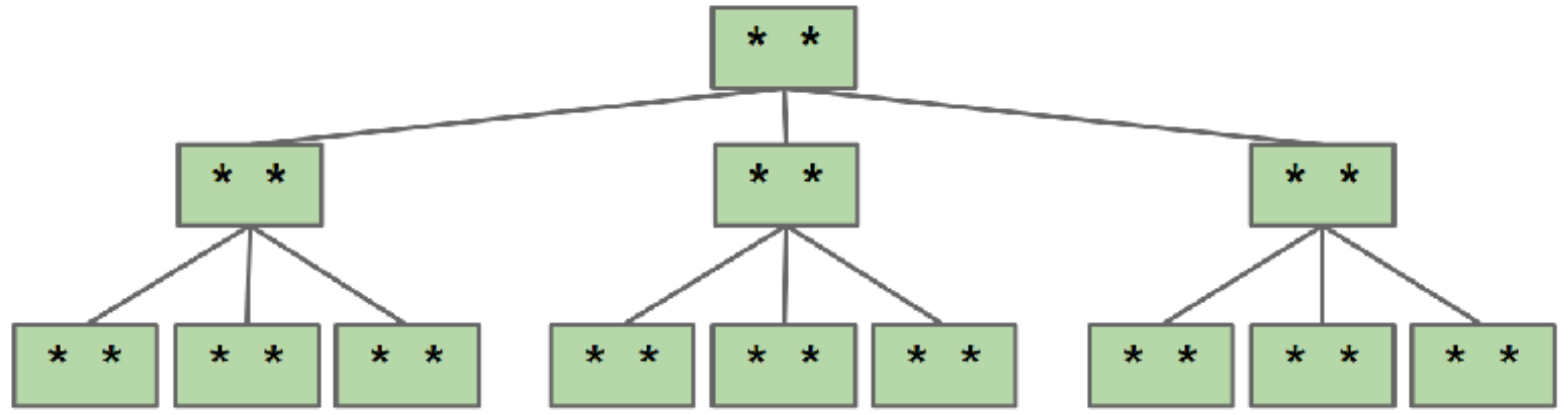
Runtime for `insert`

Q: What is the run time of a split operation?

A: Constant (move pointers around)

Runtime for `insert`:

- Worst case number of nodes to inspect: H + 1
- Worst case number of items to inspect per node: L
- Worst case number of split operations: H + 1
- Overall runtime: O(HL)



Since H = Θ(log N), overall runtime is O(L log N).

- Since L is a constant, runtime is therefore O(log N).

Bottom line: find and insert are both O(log N).

Summary for dictionary operations

- B-trees are doing better than binary search trees!

	Worst case			Average case		
	Search	Insert	Delete	Search	Insert	Delete
BST	n	n	n	$\log n$	$\log n$	$\sqrt{n}$
B-Trees	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$

Note, delete is out of scope for this class

Summary

B-Trees are a modification of the binary search tree that avoids Θ(N) worst case.

- Nodes may contain between 1 and L items.
- search works almost exactly like a normal BST.
- insertion works by adding items to existing leaf nodes.
  - If nodes are too full, they split.
- Resulting tree has perfect balance. Runtime for operations is O(log N).
- Limitations: have not discussed deletion. Have not discussed how splitting works if L > 3.
- B-trees are more complex data structures, but they can efficiently handle ANY insertion order.

Lecture 18 wrap-up

- No HW this week!
- Checkpoint 2 review slides will be up in a bit
- Checkpoint 2 is next Monday, please schedule SDRC tests

Resources

- Reading from textbook: Chapter 3.3 (Pages 424-447); https://algs4.cs.princeton.edu/33balanced/
- B-tree visualization: https://yongdanielliang.github.io/animation/web/24Tree.html and https://www.cs.usfca.edu/~galles/visualization/BTree.html
- Practice problems behind this slide
- Most of these slides come from UC Berkeley's data structures course

Problem 1

- Draw the 2-3-4 tree that results when you insert the keys 25, 12, 16, 13, 24, 8, 3, 18, 1, 5, 19, 20 in that order into an initially empty tree.

Problem 2

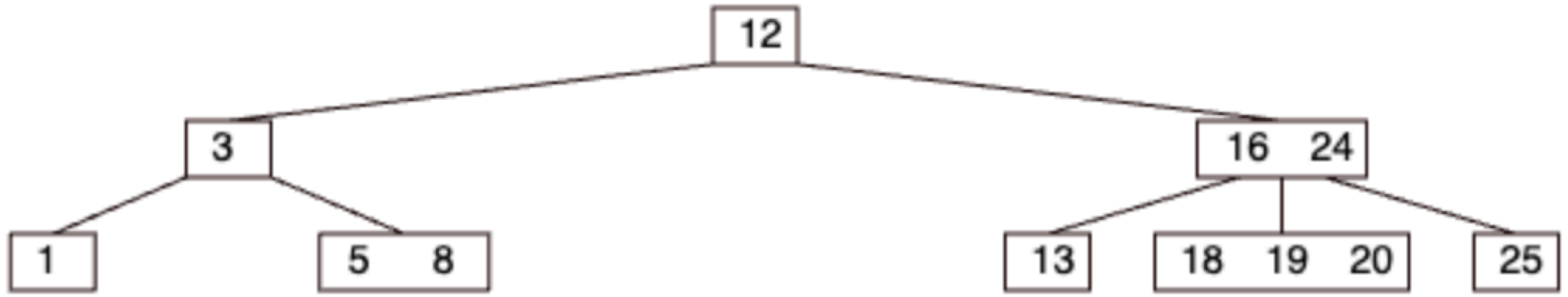
- Find an insertion order for the keys 1, 2, 3, 4, 5, 6, and 7 that leads to a 2-33 tree of height 1.

Problem 3

- Find an insertion order for the keys 19, 5, 1, 18, 3, 8, 24, 13 that leads to a 2-3-4 tree of height 1.

ANSWER 1

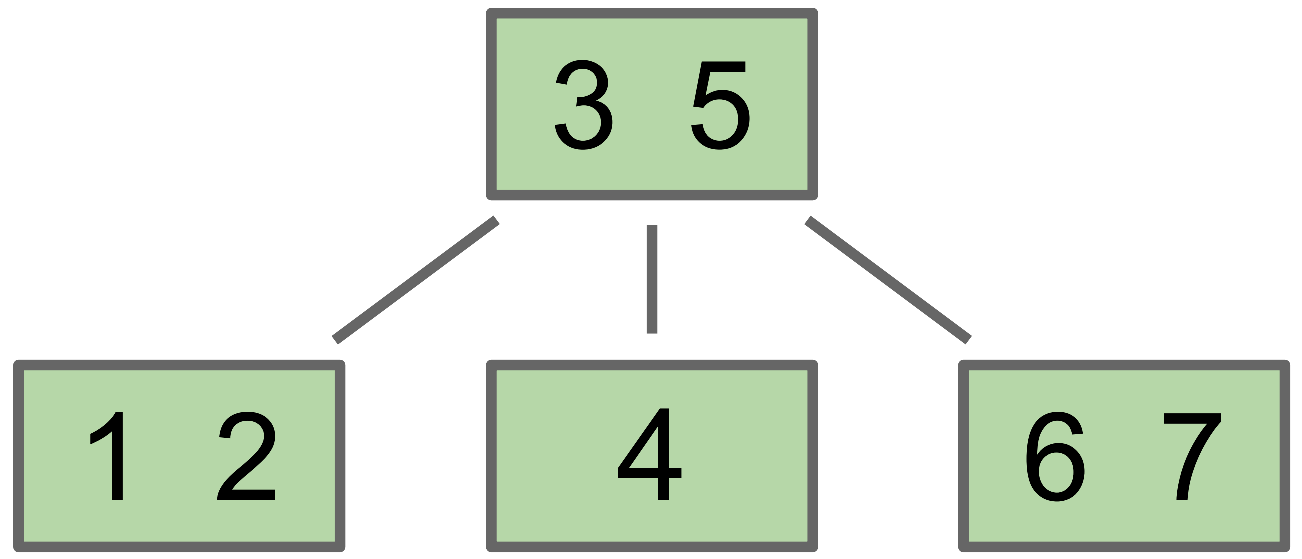
- Draw the 2-3-4 tree that results when you insert the keys 25, 12, 16, 13, 24, 8, 3, 18, 1, 5, 19, 20 in that order into an initially empty tree.



ANSWER 2

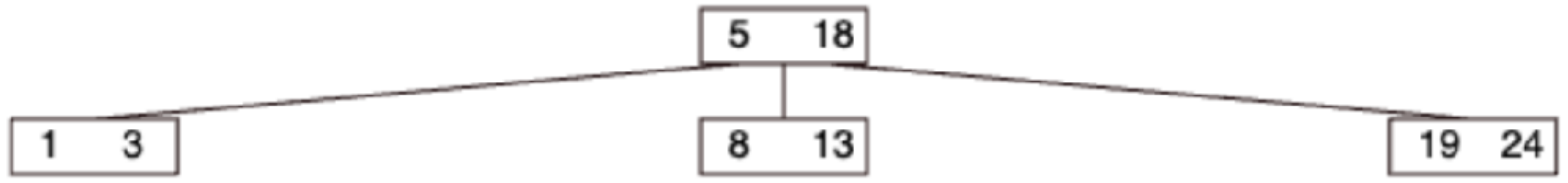
Find an order such that if you add the items 1, 2, 3, 4, 5, 6, and 7 in that order, the resulting 2-3 tree has height 1.

- One possible answer: 2, 3, 4, 5, 6, 1, 7



ANSWER 3

- Find an insertion order for the keys 19, 5, 1, 18, 3, 8, 24, 13 that leads to a 2-3-4 tree of height 1.
- Example of insertion order: 5 1 13 24 18 3 8 19


5 18
1 3
8 13
19 24