Announcements

- Midterm 2: April 16th, it will cover Topics, 4, 5 (step-counting and recurrence relations), 6, and 7.
- Midterm Review Session: 4/13, 5pm-7pm, GS 906
- Programming Project 4 is ongoing, due April 21st

Topic 8: Beyond Binary Search Trees

By: Professor Lynam

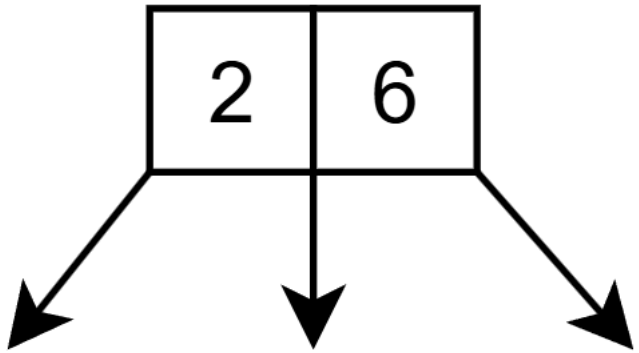
2-3 Trees

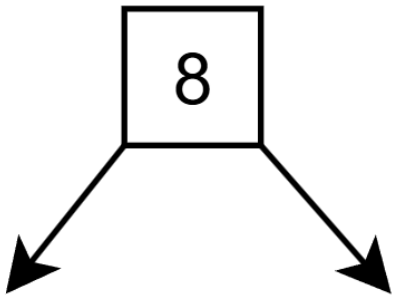
- Is there any reason to store more than one data value in a tree node?
- Yes: we could have a tree with the same data as a binary tree, but it's shorter, which means…
- Pros: Fewer levels to traverse!
- Con: More key comparisons per node

2-3 Trees

- Definition: *2-3 Tree*
- *A 2-3 tree is a balanced, ordered search tree of one or two key values (and two or three child references) per node, such that the tree's growth occurs at its root.*

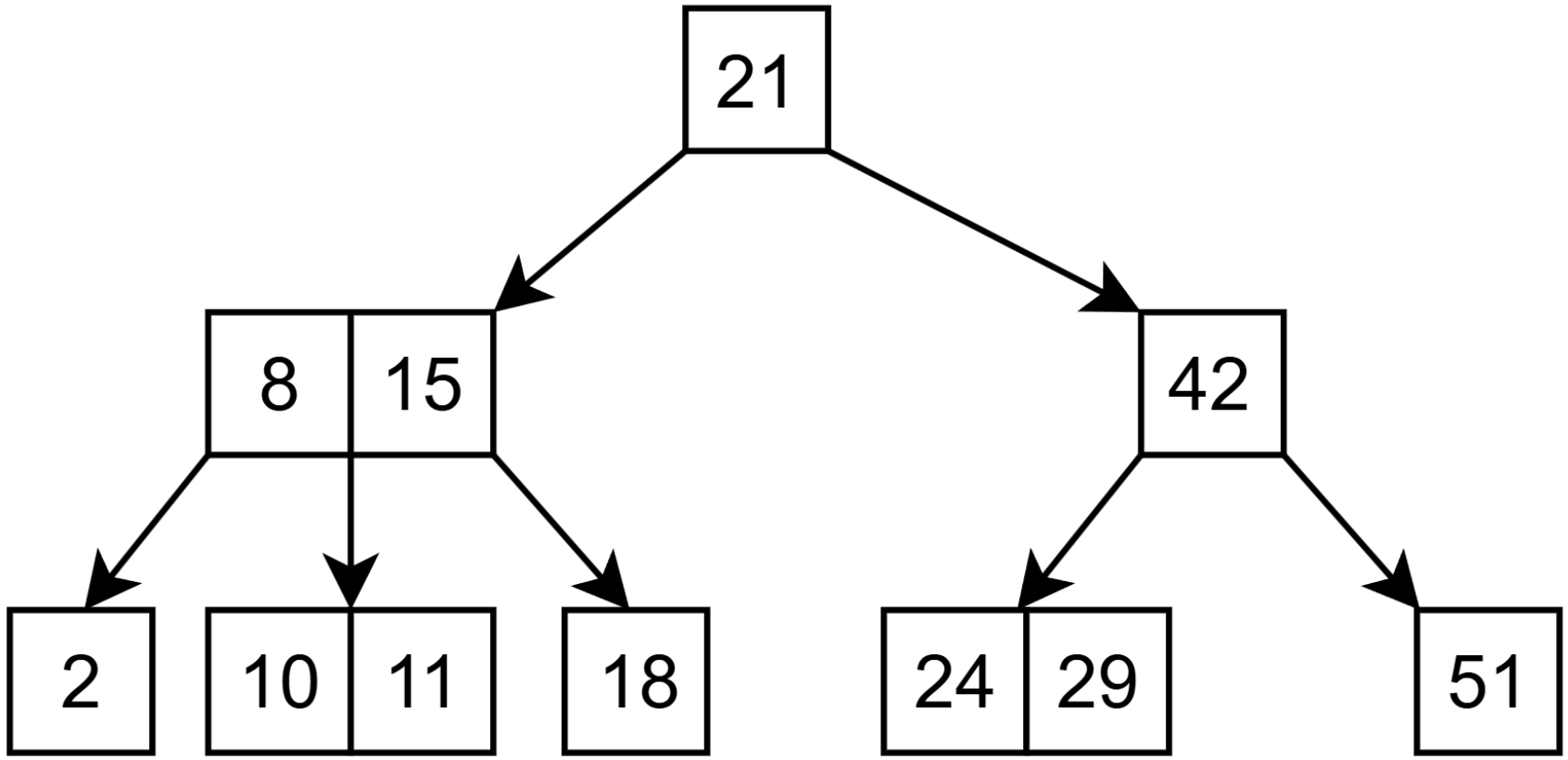
- A 2-node might look like this:
- A 3-node might look like this:





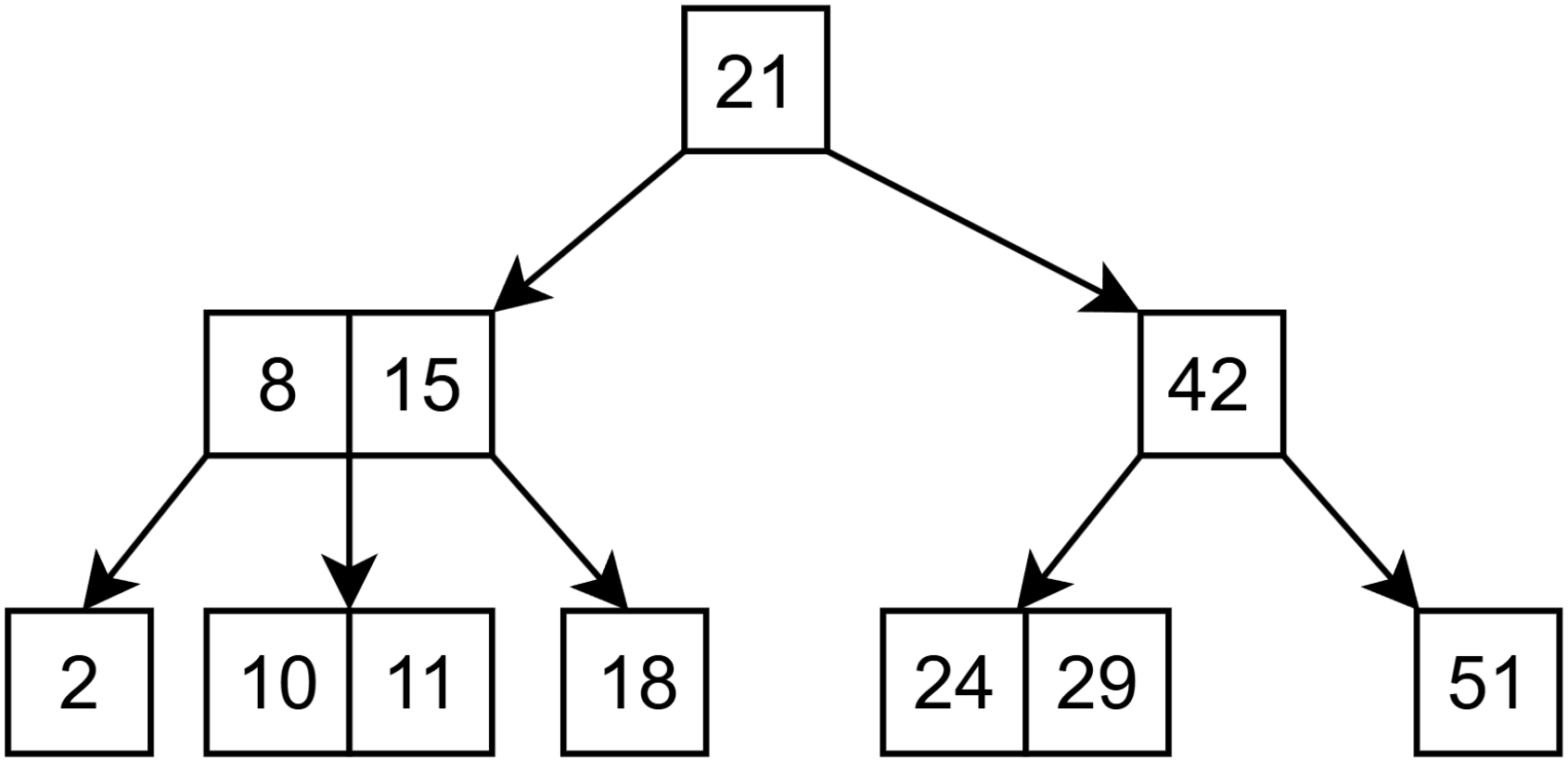
2-3 Trees

- Notes: It's balanced, with all leaves at the same level
- The 2-nodes and 3-nodes can be anywhere in the tree



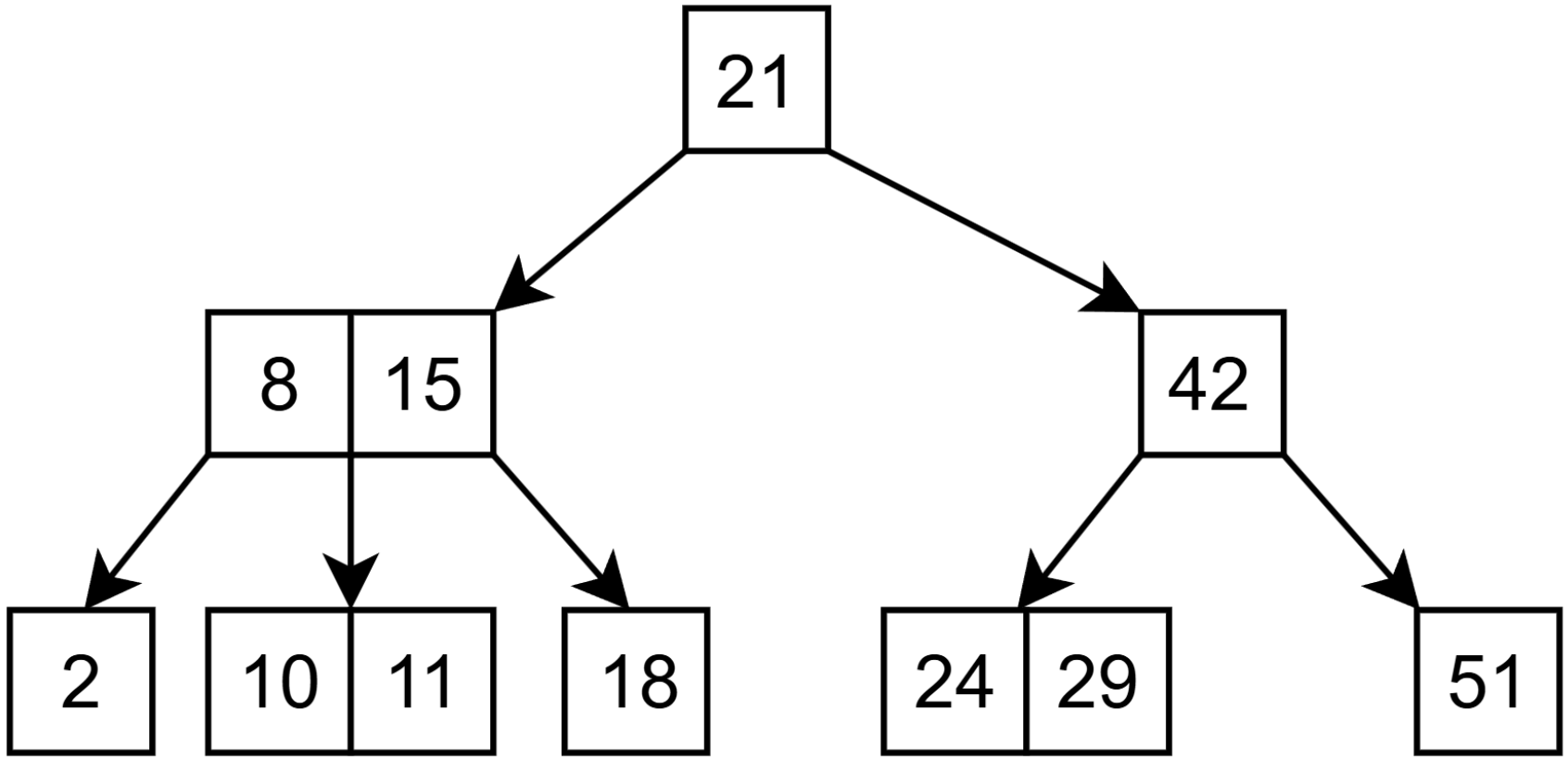
2-3 Trees

- Searching: Find 11.



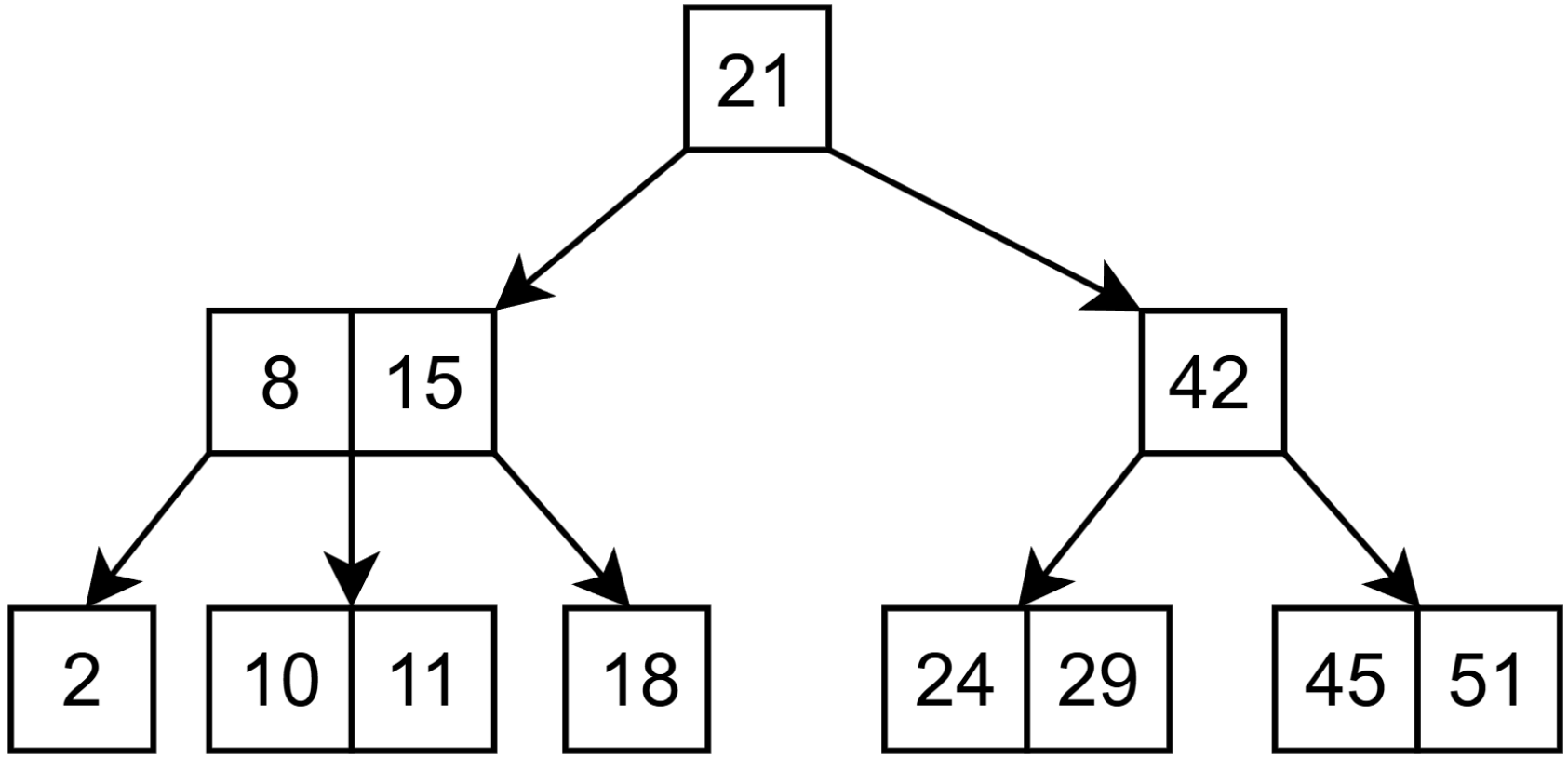
2-3 Trees

- Inserting: Insert 45.
- Start by finding the leaf where you'd expect to find the key.
- If the leaf is a 2-node, add the key (turning it into a 3-node)



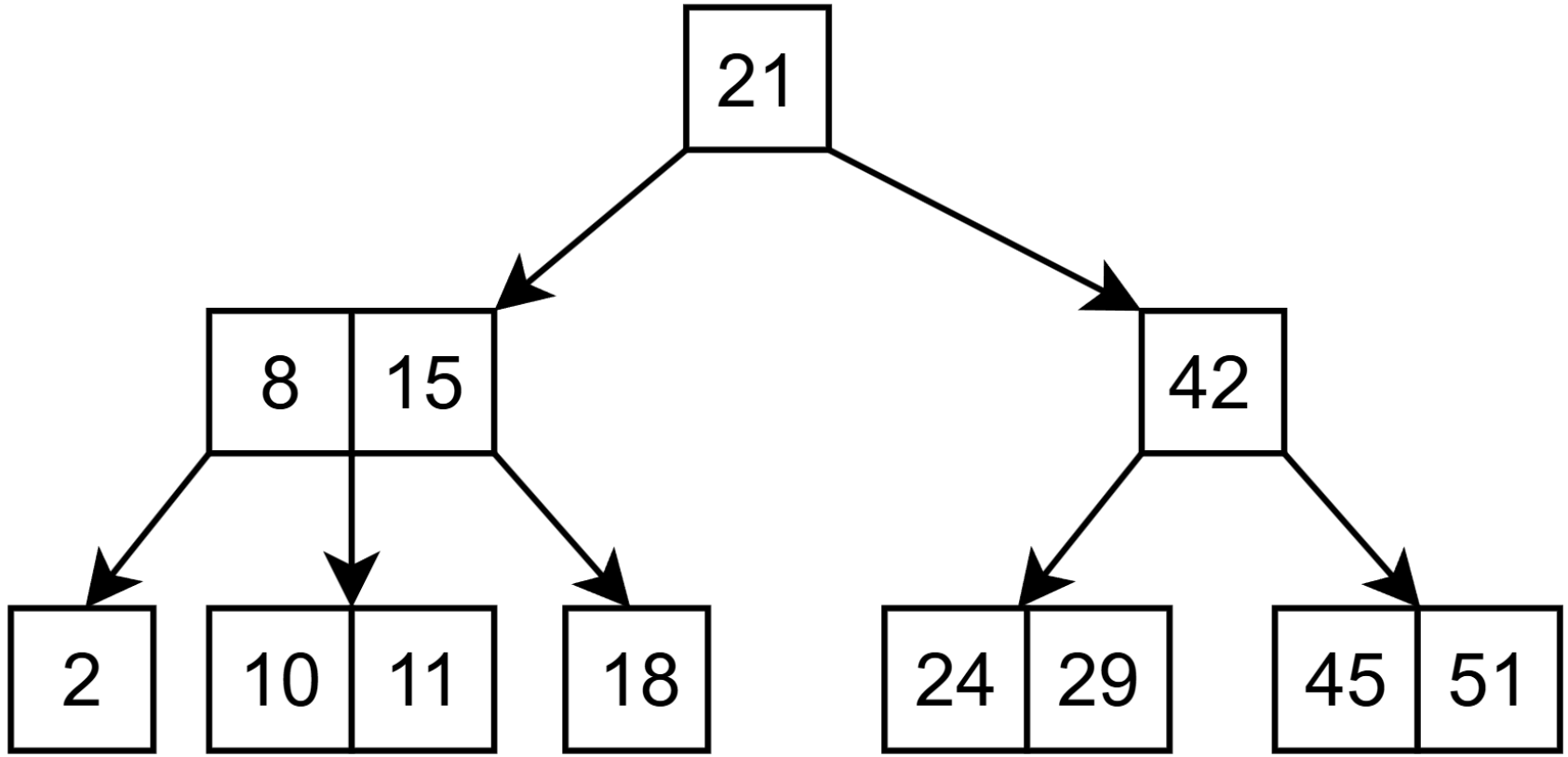
2-3 Trees

- Inserting: Insert 45.
- Start by finding the leaf where you'd expect to find the key.
- If the leaf is a 2-node, add the key (turning it into a 3-node)



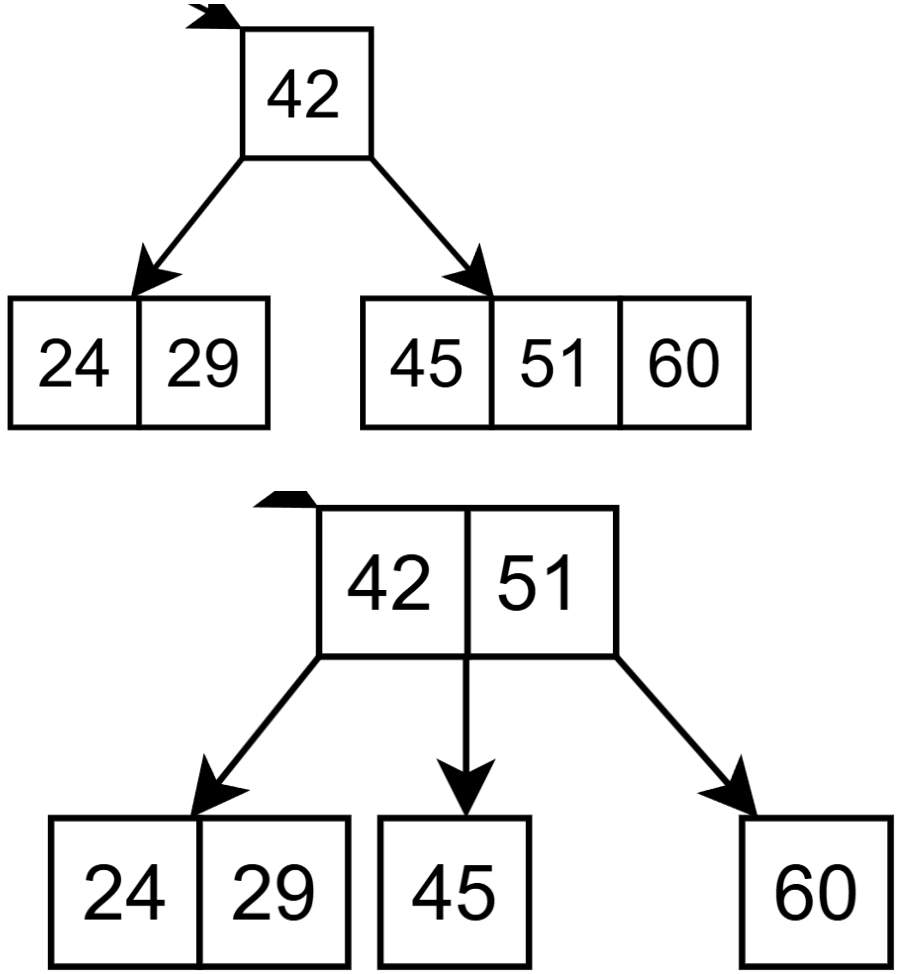
2-3 Trees

- Inserting: Insert 60.
- What if the leaf is already a 3-node?
- Find the median of the 3 values, ‘promote’ it to the parent, make 2-nodes for the other values, and attach them to the parent



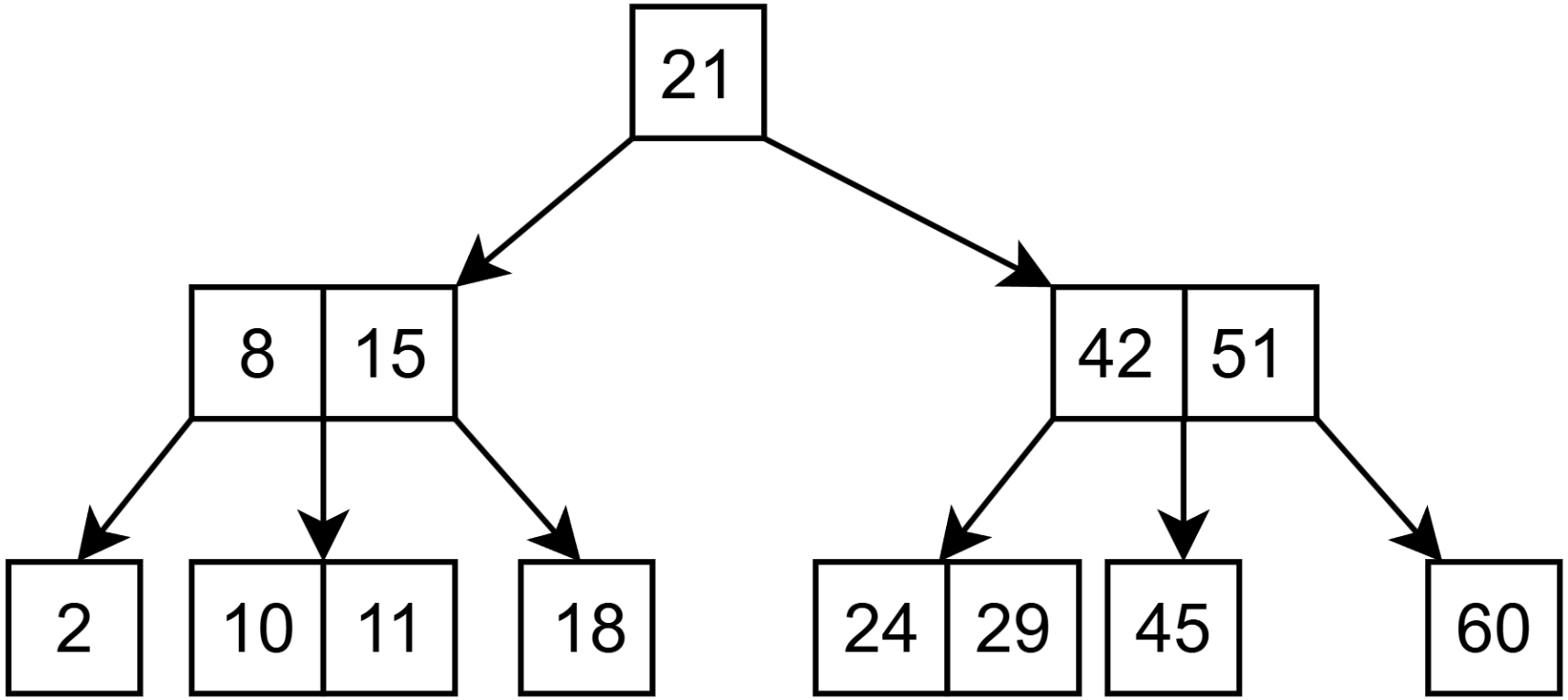
2-3 Trees

- Inserting: Insert 60.
- Find the median of the 3 values, ‘promote’ it to the parent, make 2-nodes for the other values, and attach them to the parent



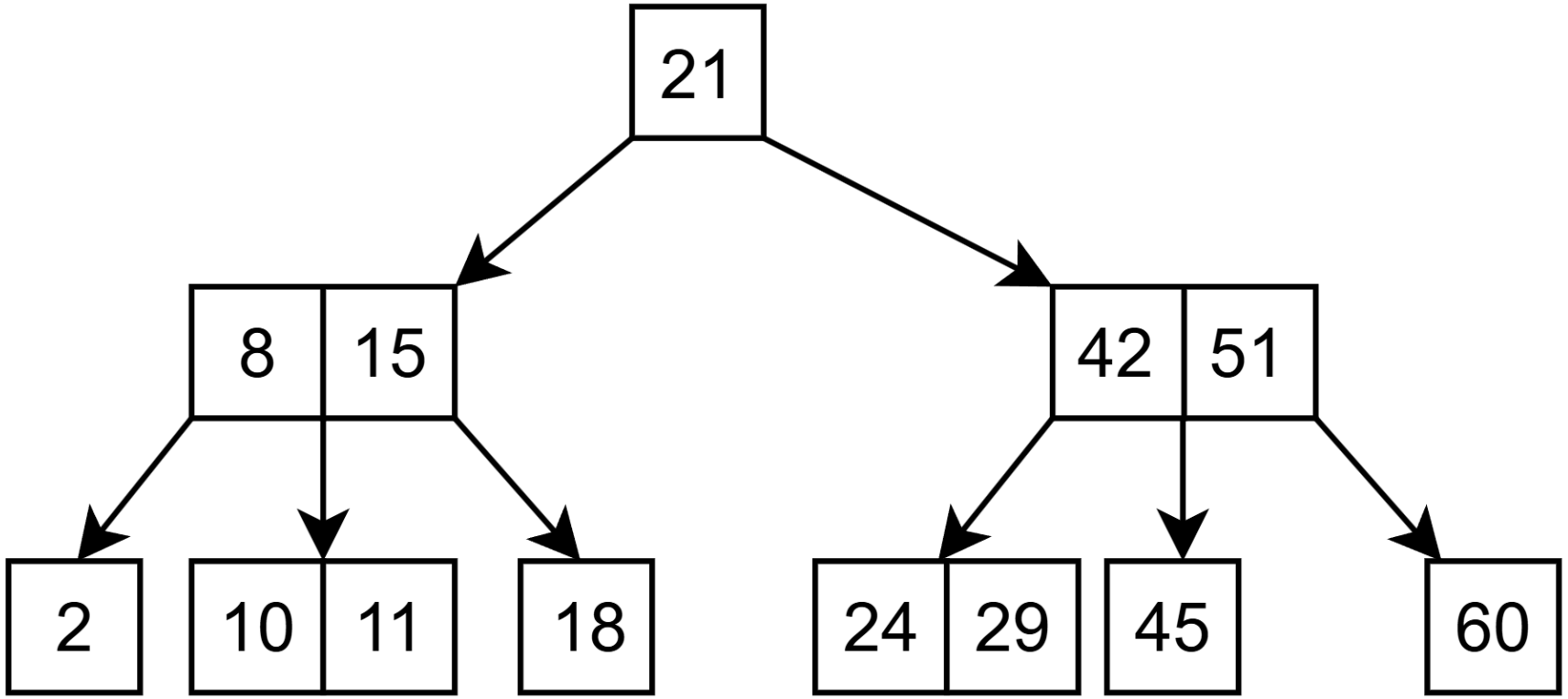
2-3 Trees

- Inserting: Insert 25.
- But what if the parent is already a 3-node?



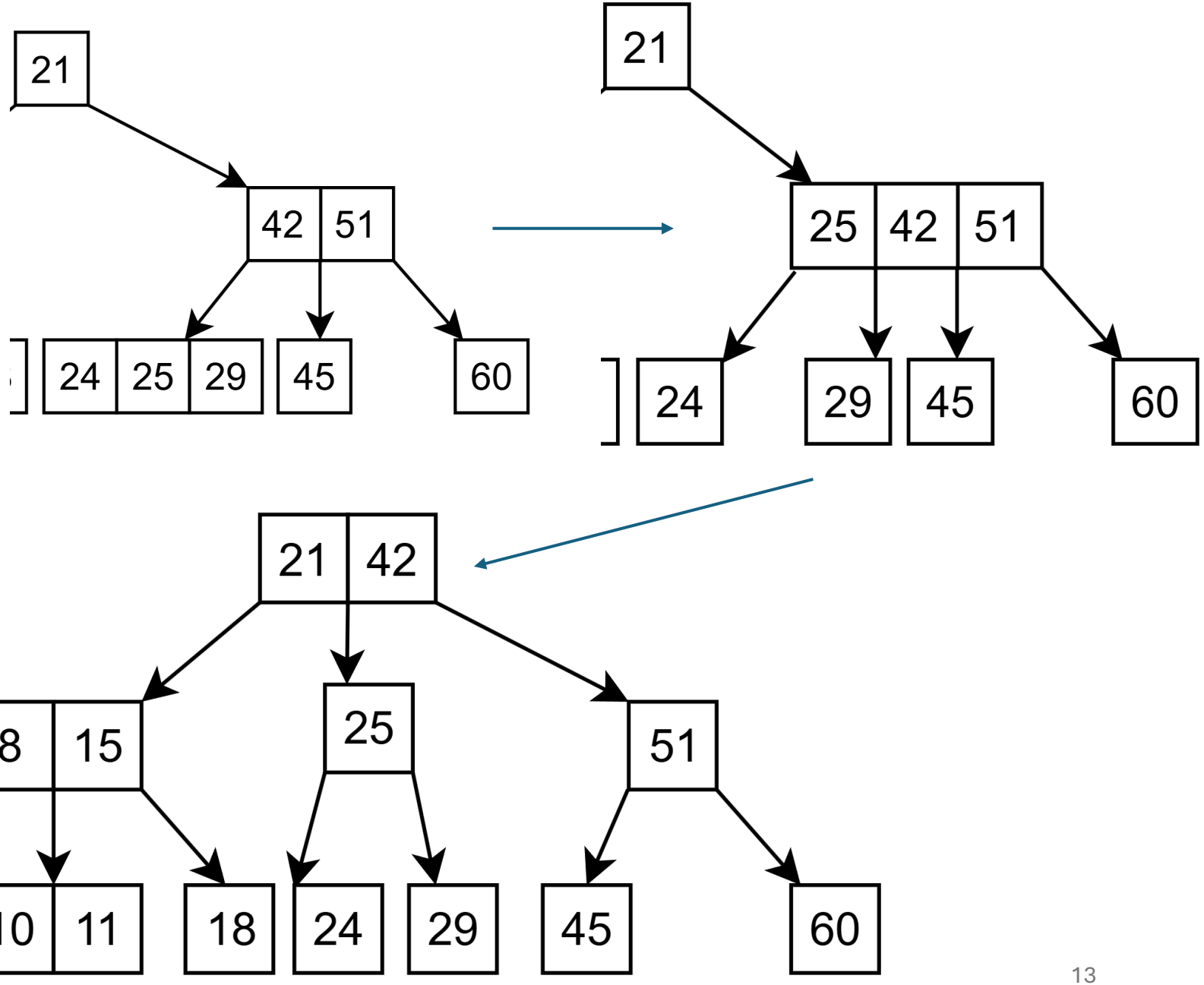
2-3 Trees

- Inserting: Insert 25.
- But what if the parent is already a 3-node?
- Keep promoting based on the medians!



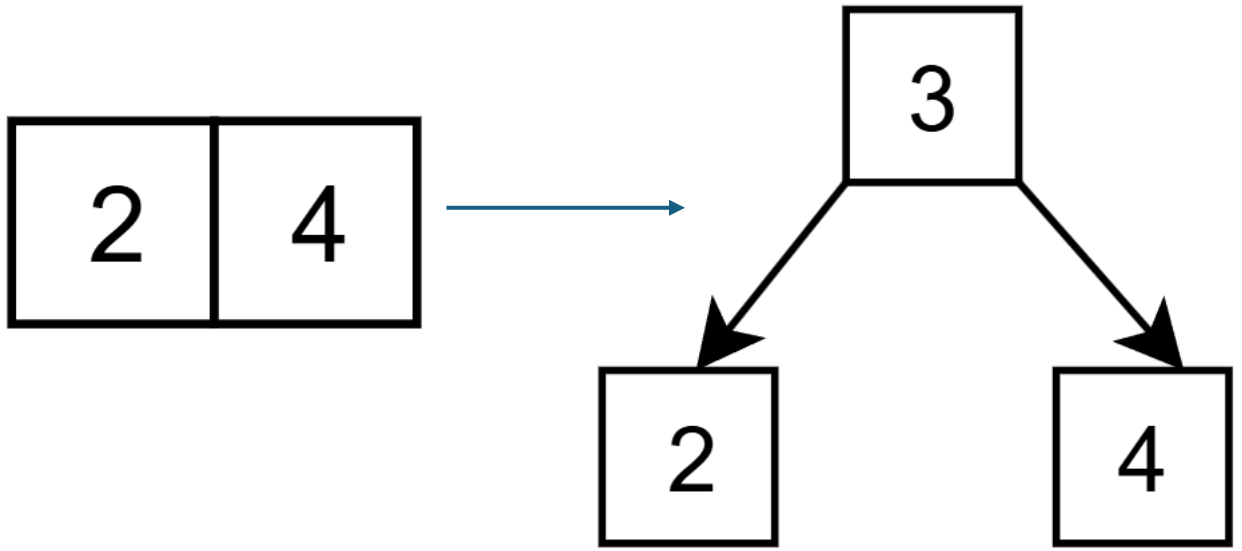
2-3 Trees

- Inserting: Insert 25.
- But what if the parent is already a 3-node?
- Keep promoting based on the medians!



2-3 Trees

- Inserting: Insert 3.
- But what if the root is a 3-node?
- Promote the median into a new root 2-node!



2-3 Trees

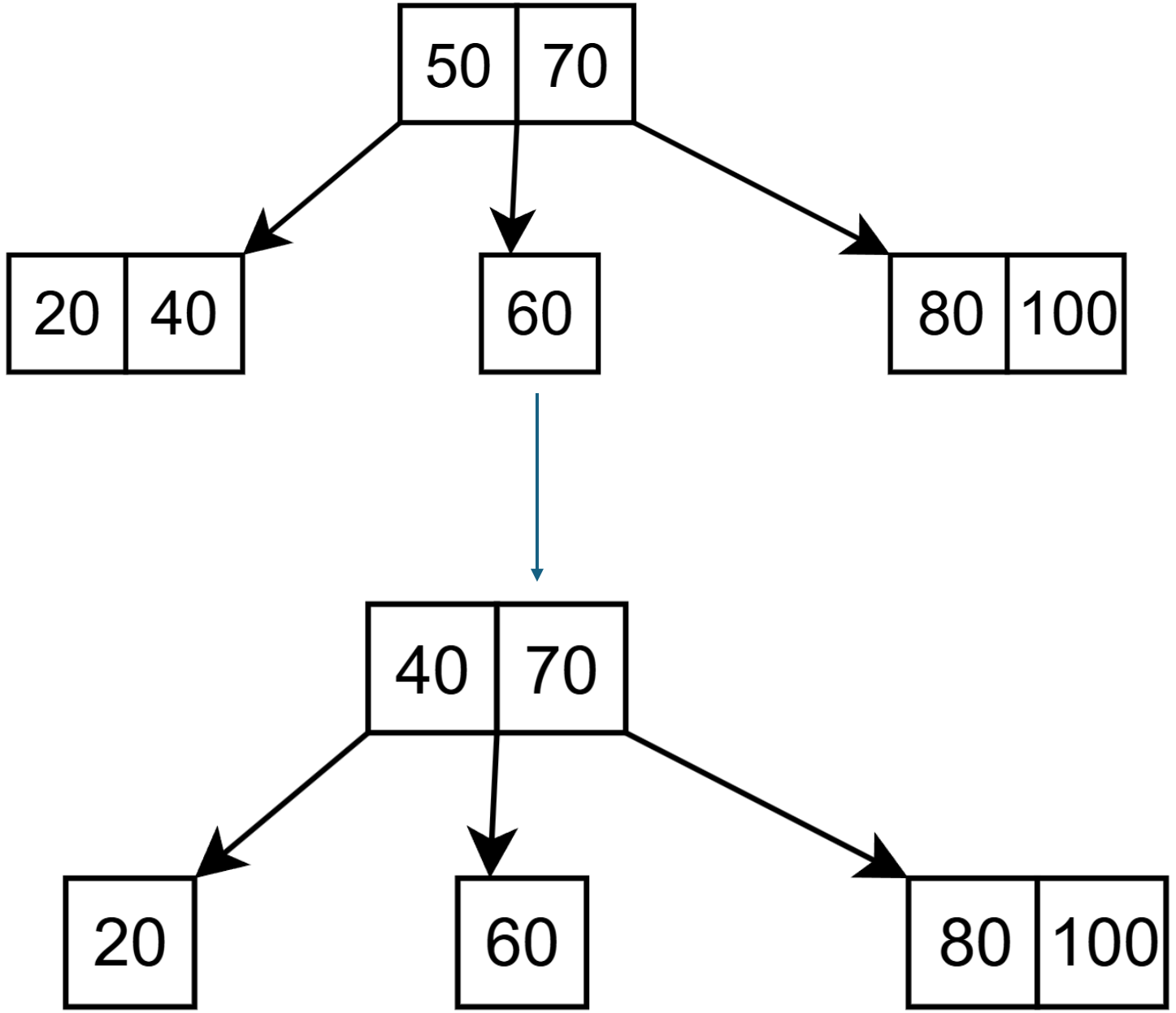
- This ‘promotion’ method allows a 2-3 to grow while maintaining its height balance
- If the root splits, the tree’s height is increased by one, but all the leaves are still on the same level
- If the deletion process (which we’re about to go over) reaches the root and leaves it empty, the tree’s height decreases by one (and the leaves are still all on the same level)

2-3 Trees

2-3 Deletion:
If the victim is an internal node:
    Replace it with its inorder predecessor or successor (repeatedly, till we reach a leaf)
    \\ Note that the deletion is now from leaf node!
If the leaf is a 3-node:
    Leave it in place, but as a 2-node
Else if the leaf has an adjacent sibling 3-node:
    Borrow a key from it (via the parent's separating value)
Else: \\ leaf was 2-node, its adjacent siblings are also 2-nodes
    Collapse the node, one sibling, and the parent's separating value into a single 3-node

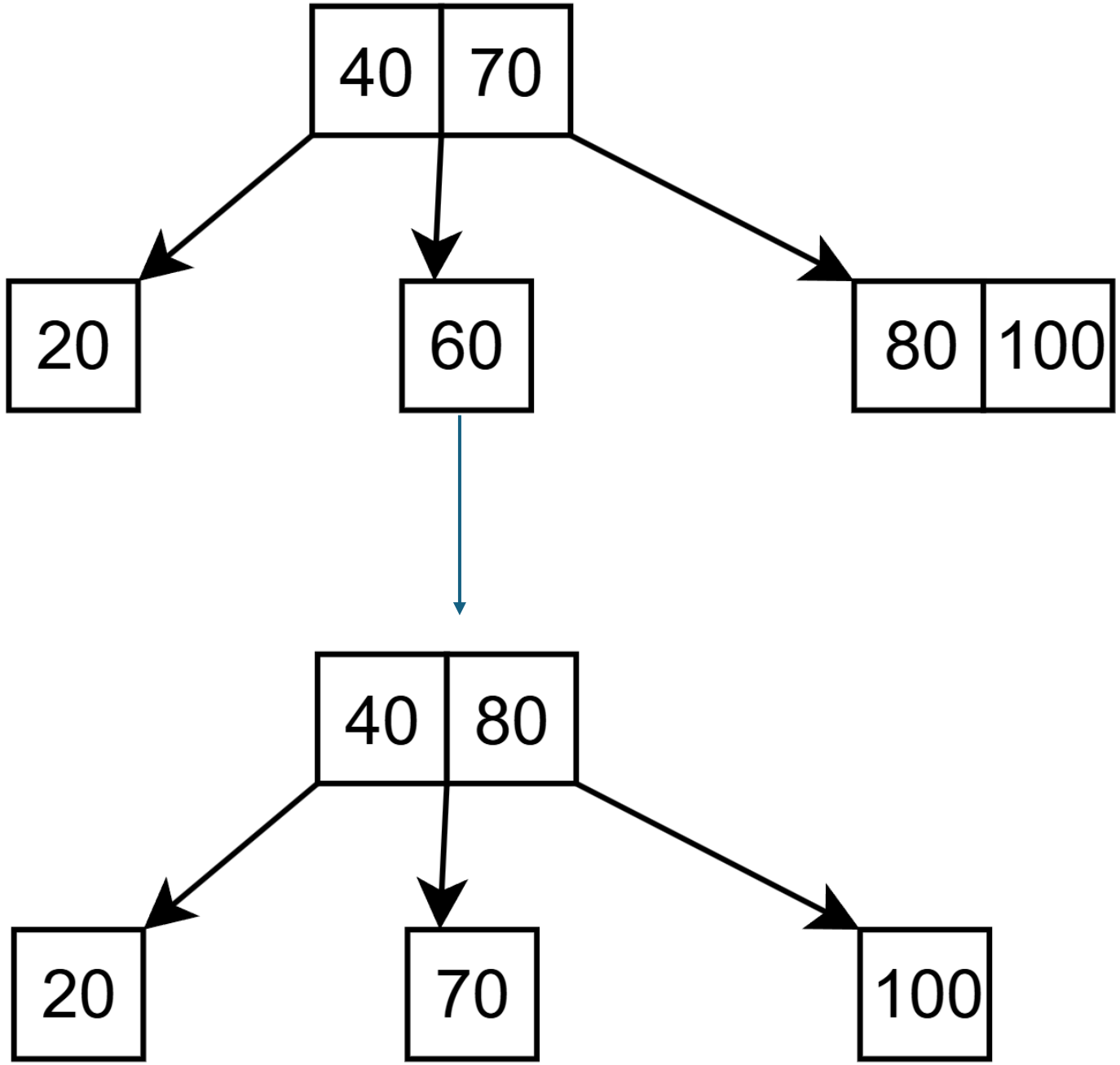
2-3 Trees

- Deletion: Delete 50.
- Just replace 50 with its inorder predecessor



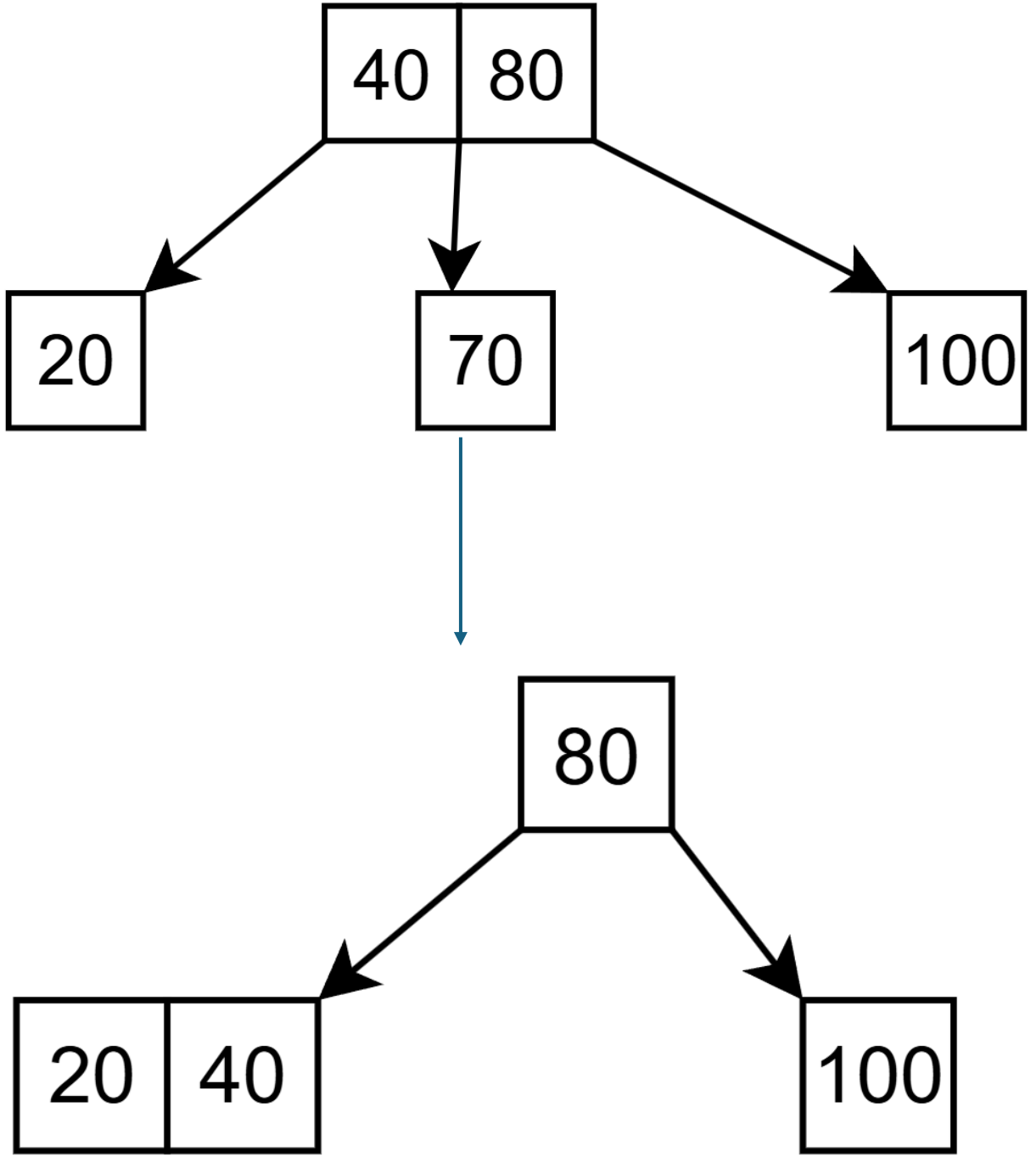
2-3 Trees

- Deletion: Delete 60.
- Else if the leaf has an adjacent sibling 3-node:
- Borrow a key from it (via the parent's separating value)



2-3 Trees

- Deletion: Delete 70.
- Else: (leaf was 2-node, its adjacent siblings are also 2-nodes)
- Collapse the node, one sibling, and the parent's separating value into a single 3-node



2-3-4 Trees

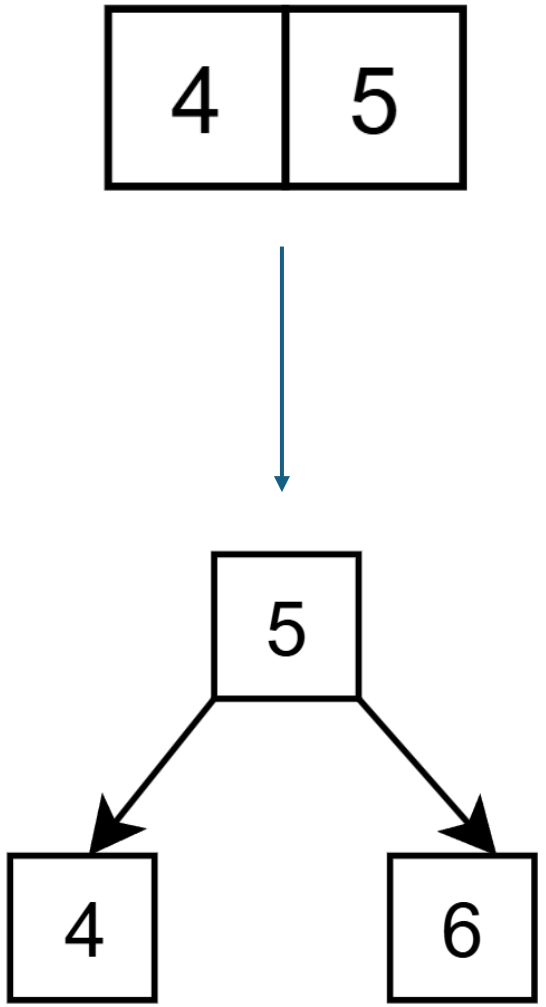
- If two values per node is good, is three better?
- Maybe: you get a shallower tree (more data per level)… But more comparisons per node
- A Red-Black tree is a binary tree implementation of a 2-3-4 tree

2-3-4-5-… Trees

- How far can we take this idea of adding data to each node?
- When we store the data on disk, the node size can be made to fit the block size of the disk
- This idea leads to a *B-Tree*, which is used for indexing in databases, and more generally in file systems

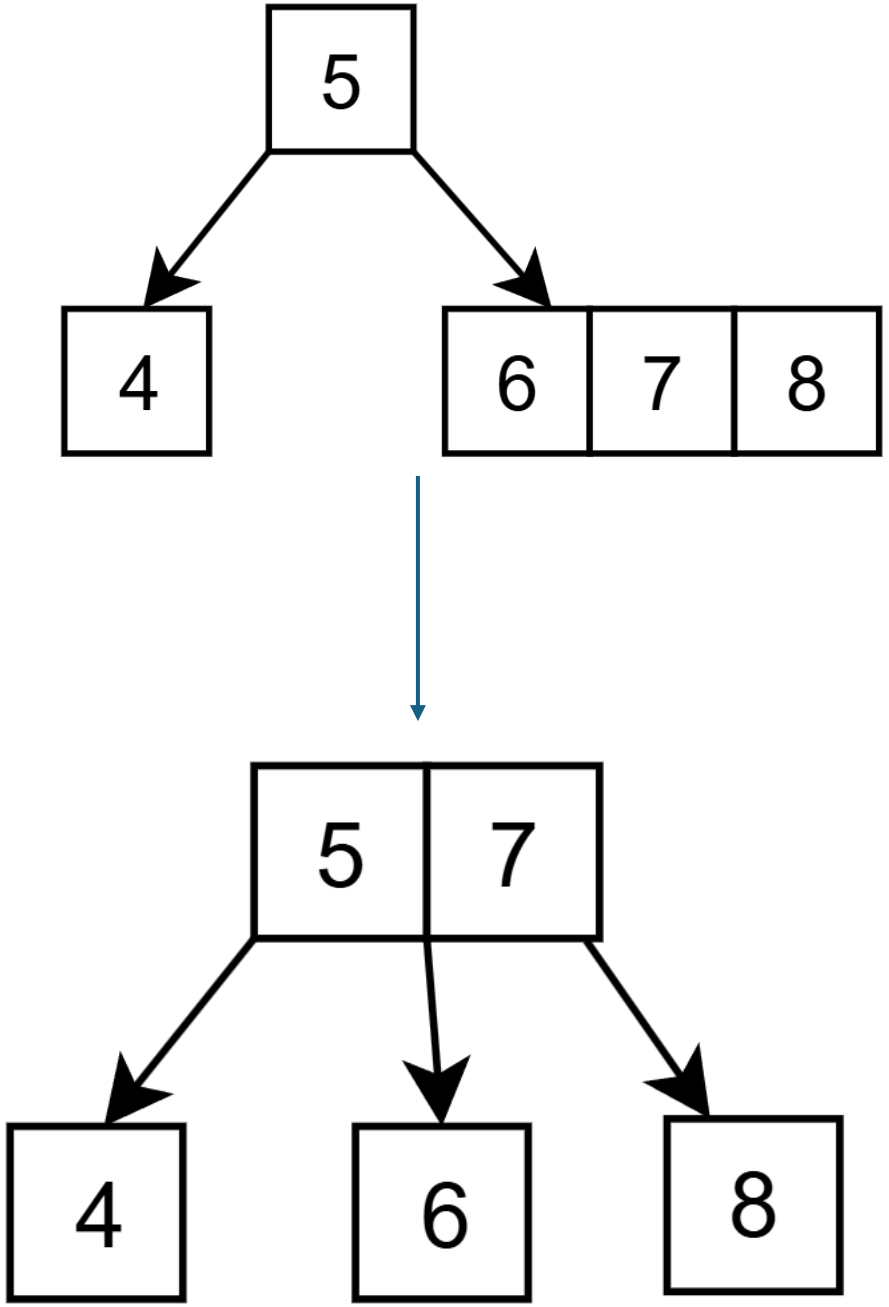
2-3 Trees

- Before we move on, let's get some practice in. Build a 2-3 Tree using the following values (in order): 4, 5, 6, 7, 8, 9, and 10



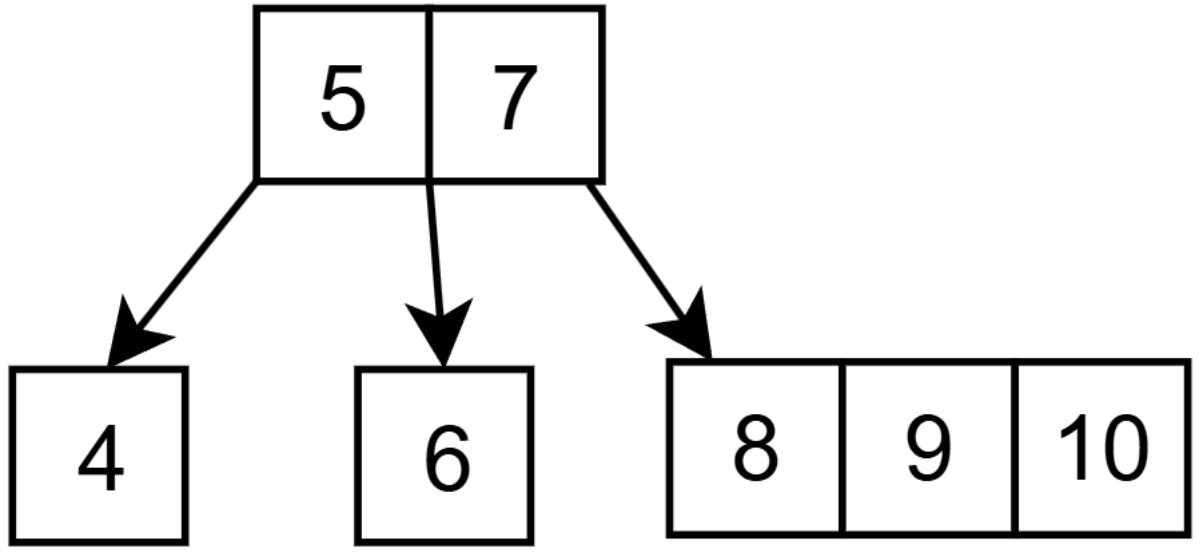
2-3 Trees

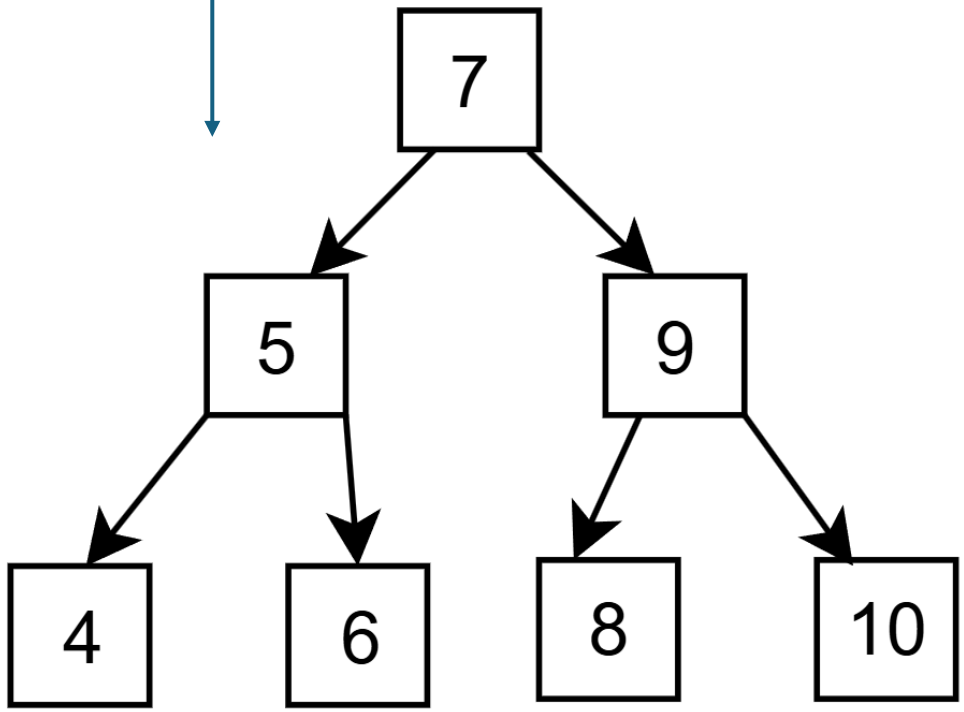
- Before we move on, let's get some practice in. Build a 2-3 Tree using the following values (in order): 4, 5, 6, 7, 8, 9, and 10



2-3 Trees

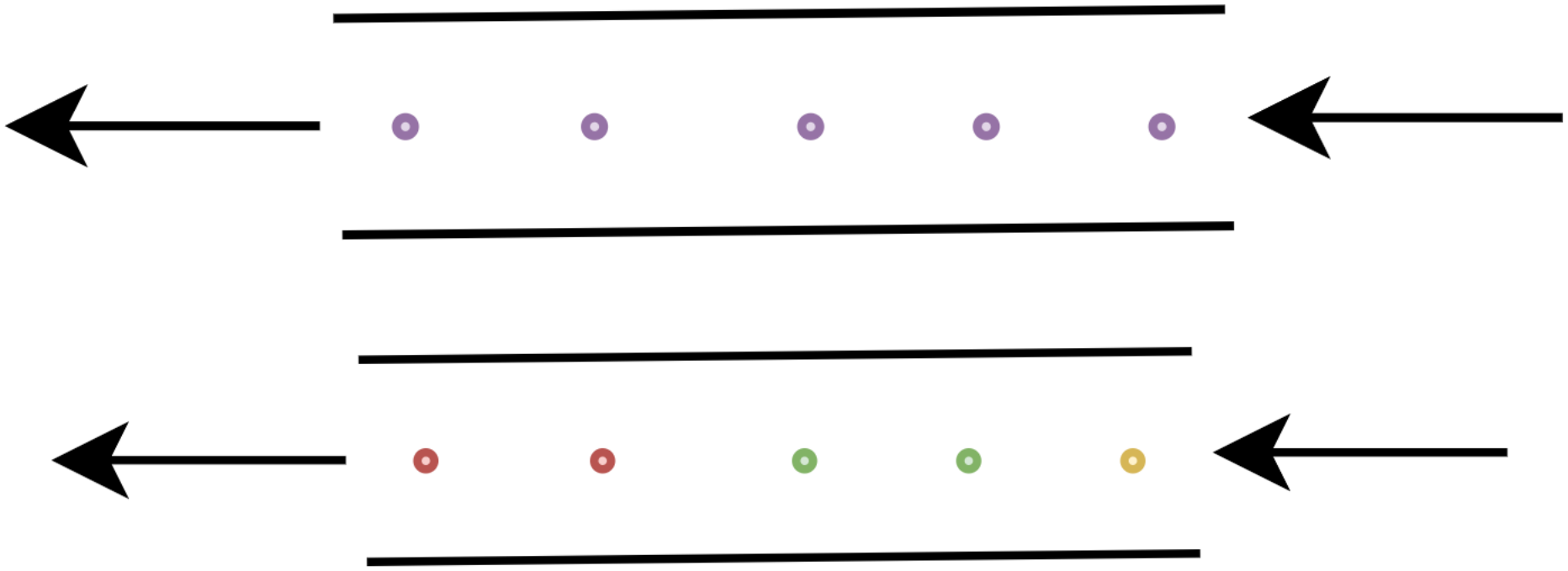
- Before we move on, let's get some practice in. Build a 2-3 Tree using the following values (in order): 4, 5, 6, 7, 8, 9, and 10





Priority Queues

- A plain queue is first-come-first-served (FCFS): elements are added (enqueued) at the rear and removed (dequeued from the front)
- In contrast, the content of a priority queue is maintained in *priority order*

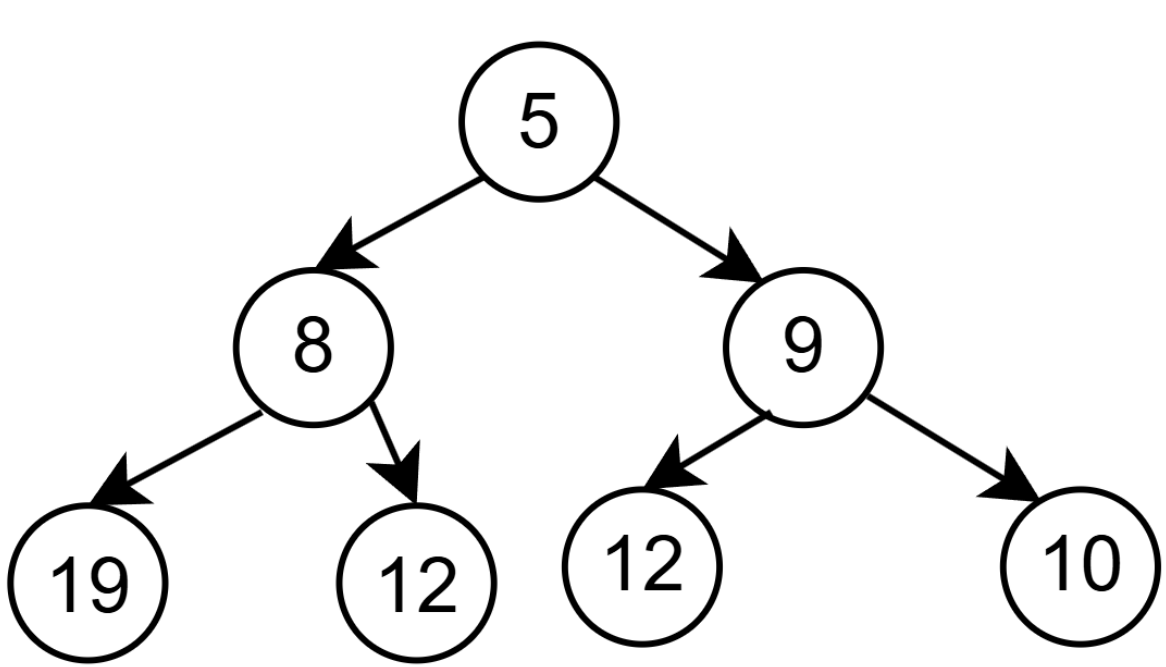


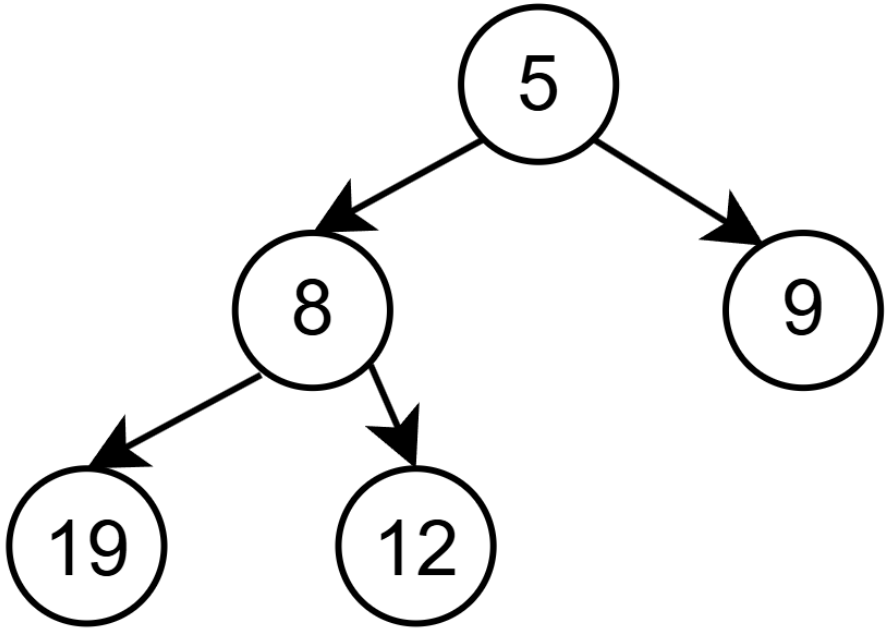
2-Heaps

- 2-Heaps are:
- An efficient representation for priority queues
- An example of a non-BST binary tree!

2-Heaps

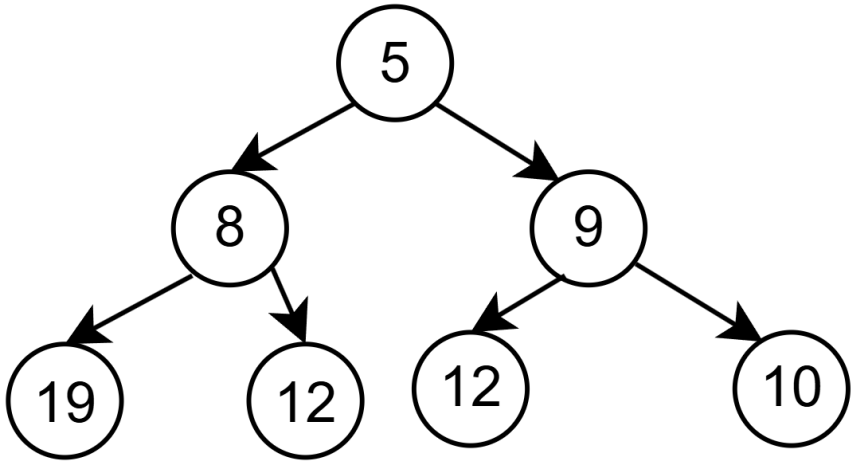
- 2-Heaps maintain a *Structure Property*: A 2-Heap is complete by this definition:
- **All levels of the tree are full of nodes, except perhaps the last, which is full on the left and empty on the right.**

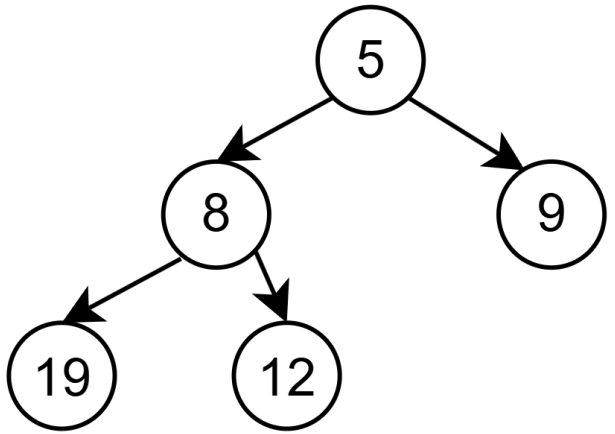




2-Heaps

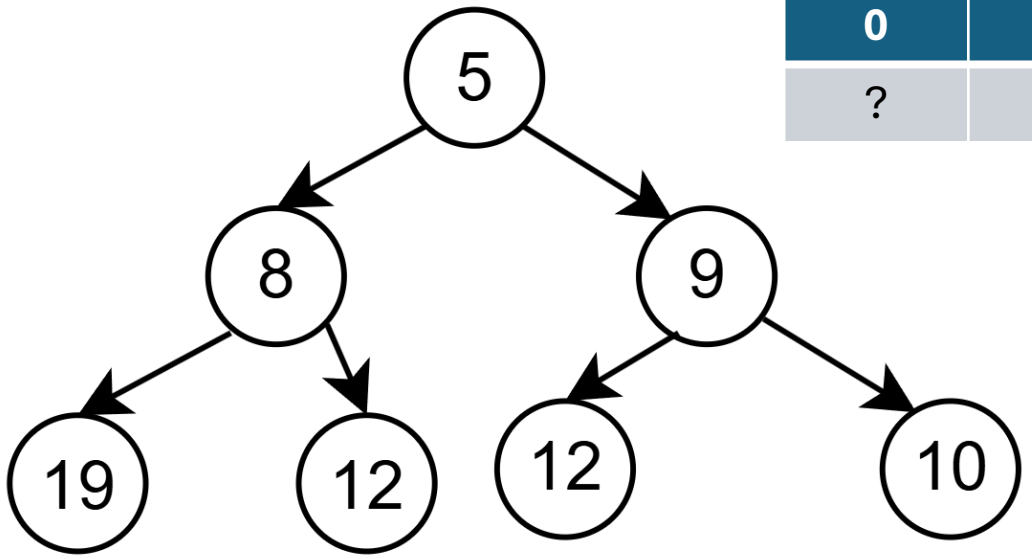
- 2-Heaps maintain an *Order Property*: Describing the legal arrangements of data within the complete heap. There are options:
- **Min Heap: For any non-root node N in the heap, the data value held in the parent N must be $\leq$ to the value held in N**
- **Max Heap: For any non-root node N in the heap, the data value held in the parent N must be $\geq$ to the value held in N**





2-Heaps

- We draw 2-Heaps as trees, but we store them linearly (in an array).
- Let's figure out how this tree would be stored:



0	1	2	3	4	5	6	7	…
?	5	8	9	19	12	12	10	…

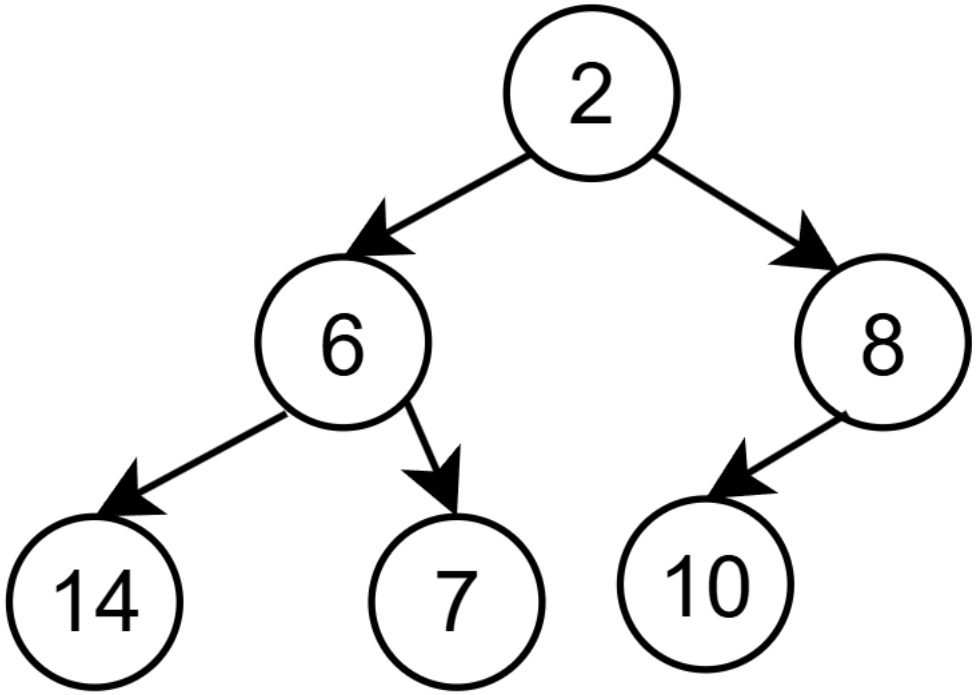
- If the parent is at index i, it's left child is at 2i, right child 2i+1
- If the child is at index i, it's parent is at index $\left\lfloor \frac{i}{2} \right\rfloor$
- What's at index 0? Could be unused, could be a 'sentinel' with +/- inf, could have the current number of elements in the heap

2-Heaps

- Let’s talk about enqueuing for a 2-Heap. To keep it a 2-Heap, we must maintain both *structure* and *order*
- To maintain *structure*, we always insert the new item at the leftmost empty location on the last level of the heap
- To maintain *order*, we “bubble-up” the new item until the parent is smaller than the child

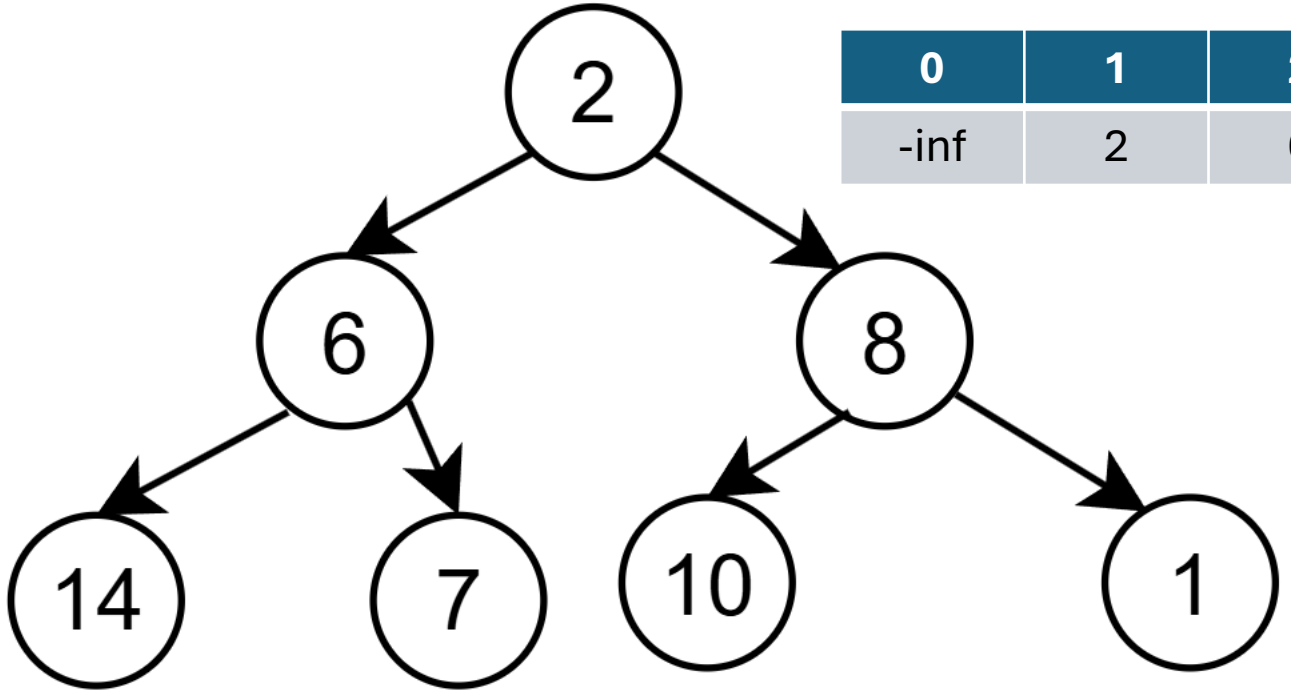
2-Heaps

- Let’s look at an example. Insert 1 into this 2-Heap. First, where to insert?



2-Heaps

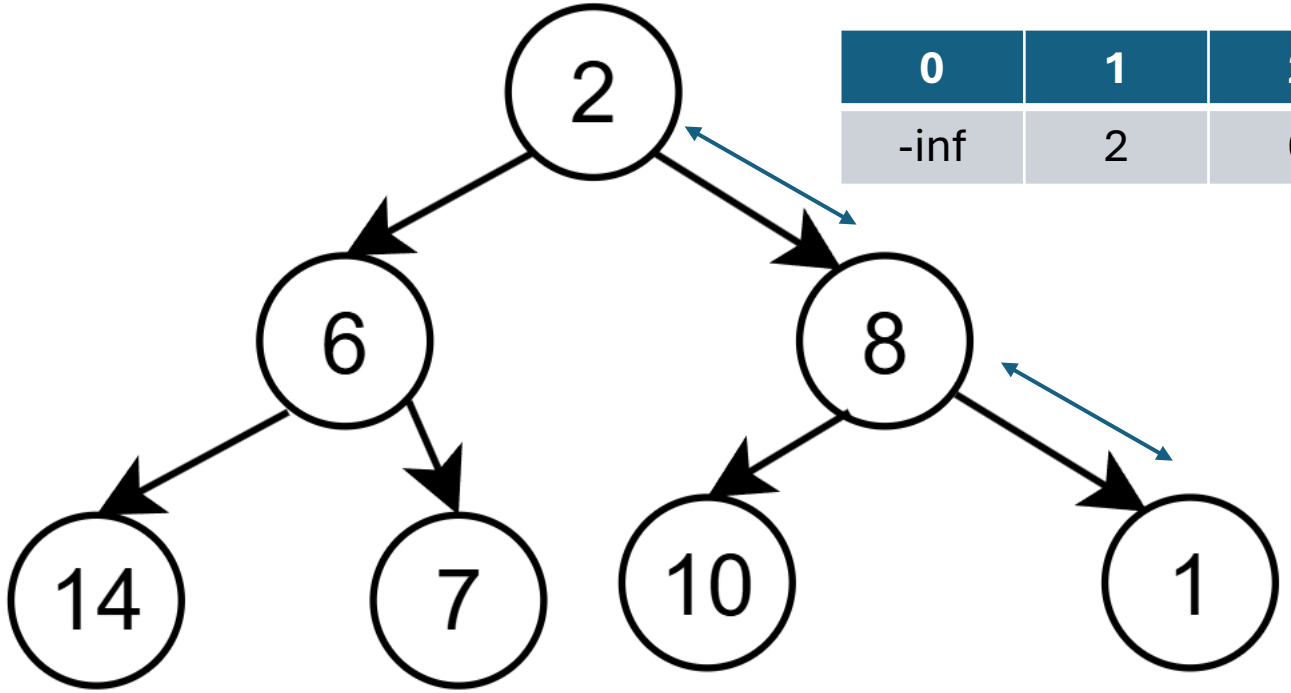
- Let's look at an example. Insert 1 into this 2-Heap. Second, what would the array holding this 2-Heap look like?



0	1	2	3	4	5	6	7	…
-inf	2	6	8	14	7	10	1	…

2-Heaps

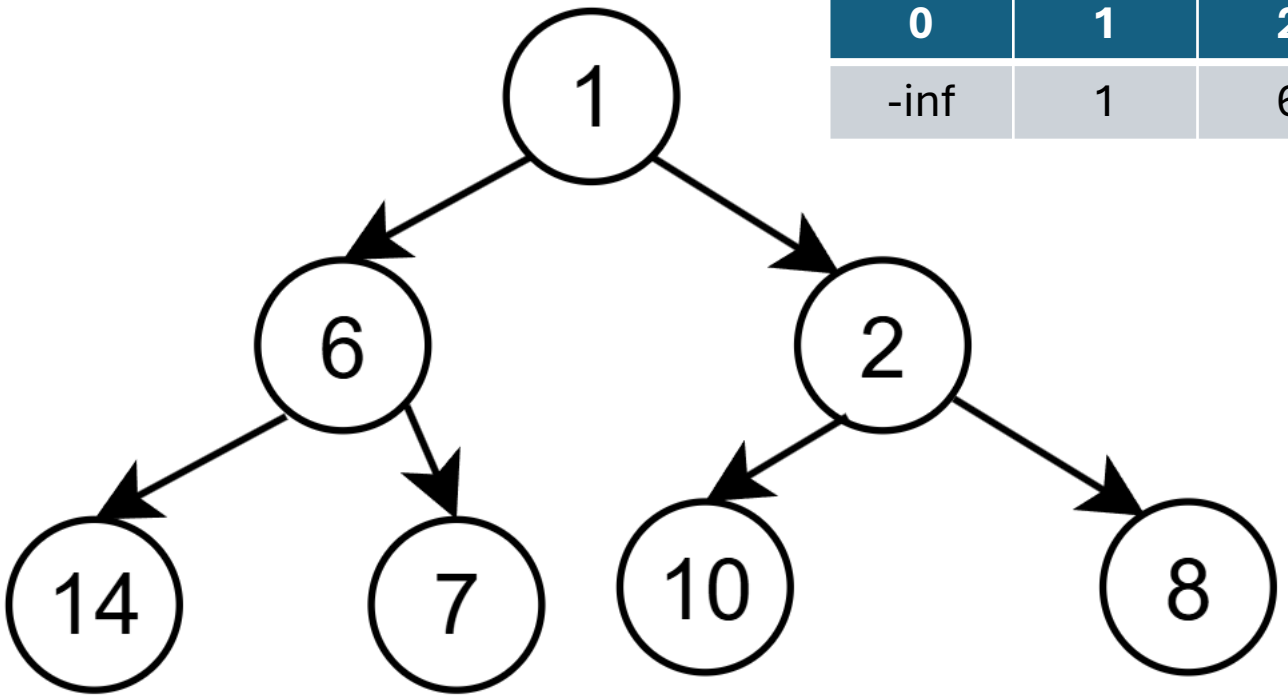
- Let's look at an example. Insert 1 into this 2-Heap. Now we have to "bubble-up" the new item. How?



0	1	2	3	4	5	6	7	...
-inf	2	6	8	14	7	10	1	...

2-Heaps

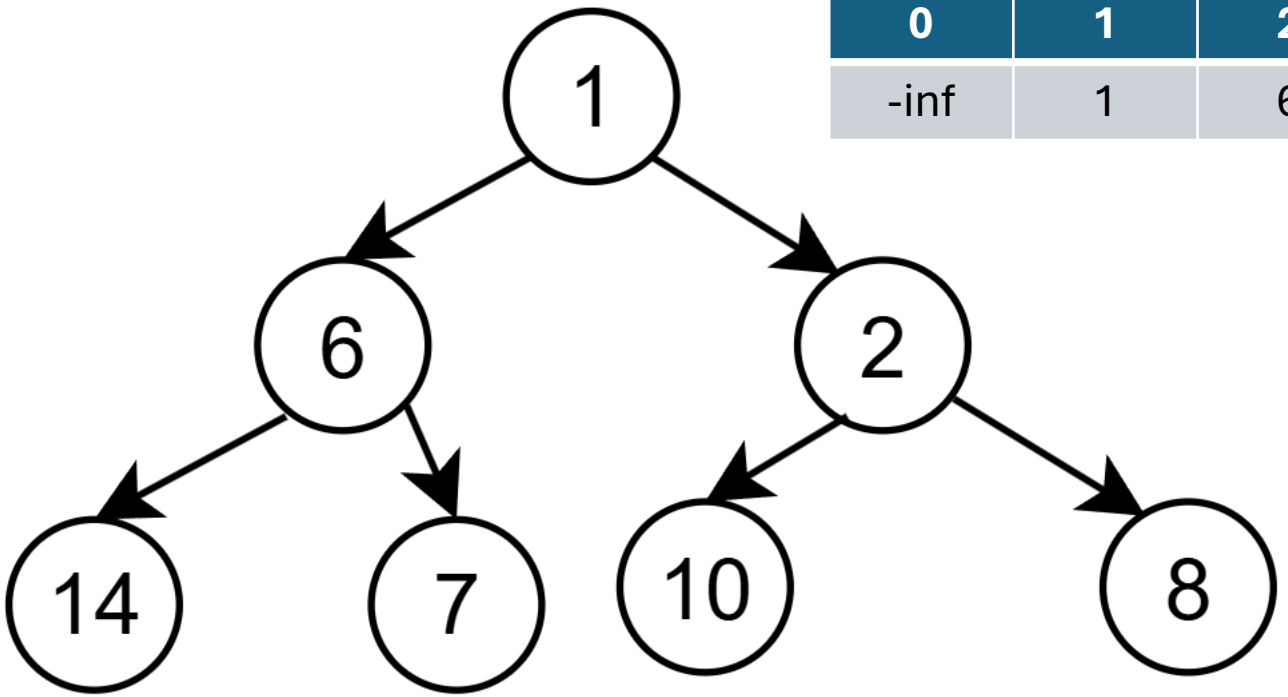
- Let's look at an example. Insert 1 into this 2-Heap. Now we're done; what would the final array for the 2-Heap look like?



0	1	2	3	4	5	6	7	…
-inf	1	6	2	14	7	10	8	…

2-Heaps

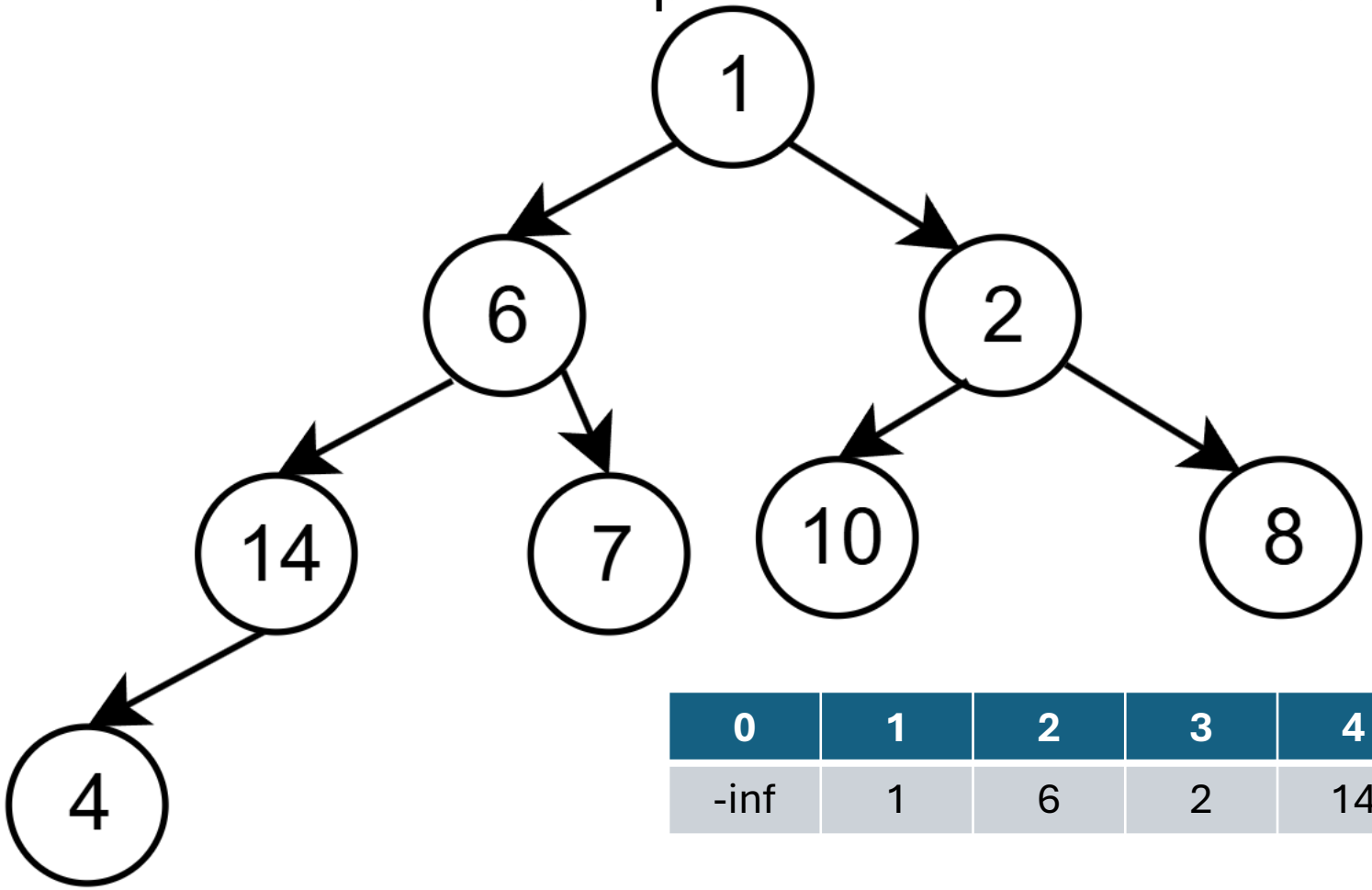
- Let's look at another example. Insert 4 into this 2-Heap. Where must we insert the new item?



0	1	2	3	4	5	6	7	...
-inf	1	6	2	14	7	10	8	...

2-Heaps

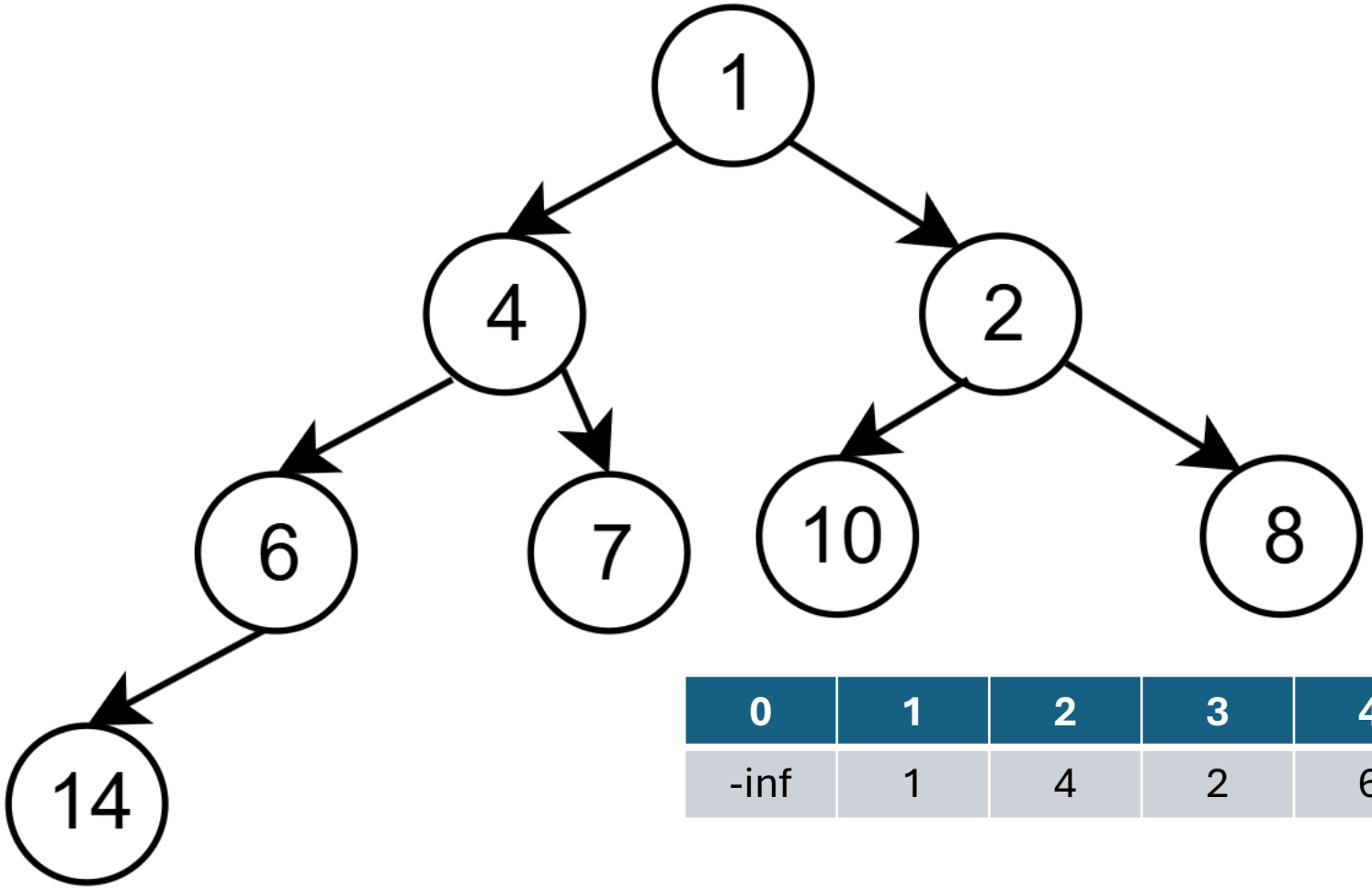
- Let's look at another example. Insert 4 into this 2-Heap. Now how far do we bubble up? And what would the new final array be?



0	1	2	3	4	5	6	7	8	...
-inf	1	6	2	14	7	10	8	4	...

2-Heaps

- Let's look at another example. Insert 4 into this 2-Heap. Now how far do we bubble up? And what would the new final array be?



0	1	2	3	4	5	6	7	8	...
-inf	1	4	2	6	7	10	8	14	...

2-Heaps

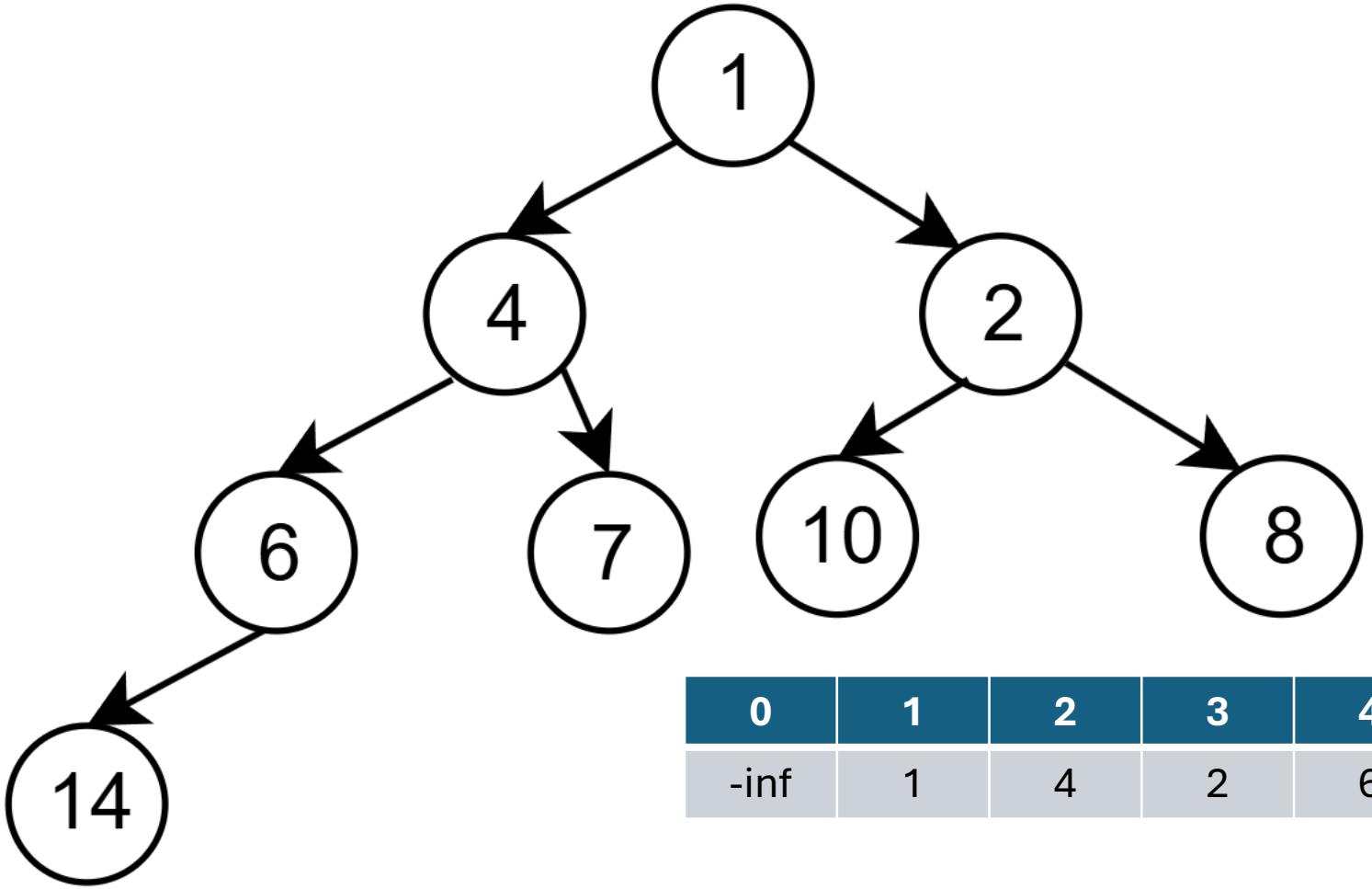
- So how efficient is this process? There are two steps, adding the item to the tree structure, and then bubbling it up (if necessary)
- O(1) to add it the structure
- $O(\log_2 n)$ to bubble up
- Overall: $O(\log_2 n)$

2-Heaps

- So we know how to enqueue to a 2-Heap… What about dequeuing?
- We want to remove the “highest priority” item, which will *always* be at the root. So first, remove the root value. But this leaves a hole in the tree!
- Step 2 is to fill the hole with last value on the last row, and “bubble *down*” to move it to an acceptable location.
- “Bubbling down” means swapping it with the smaller of the two children, repeatedly until both children are larger than the item

2-Heaps

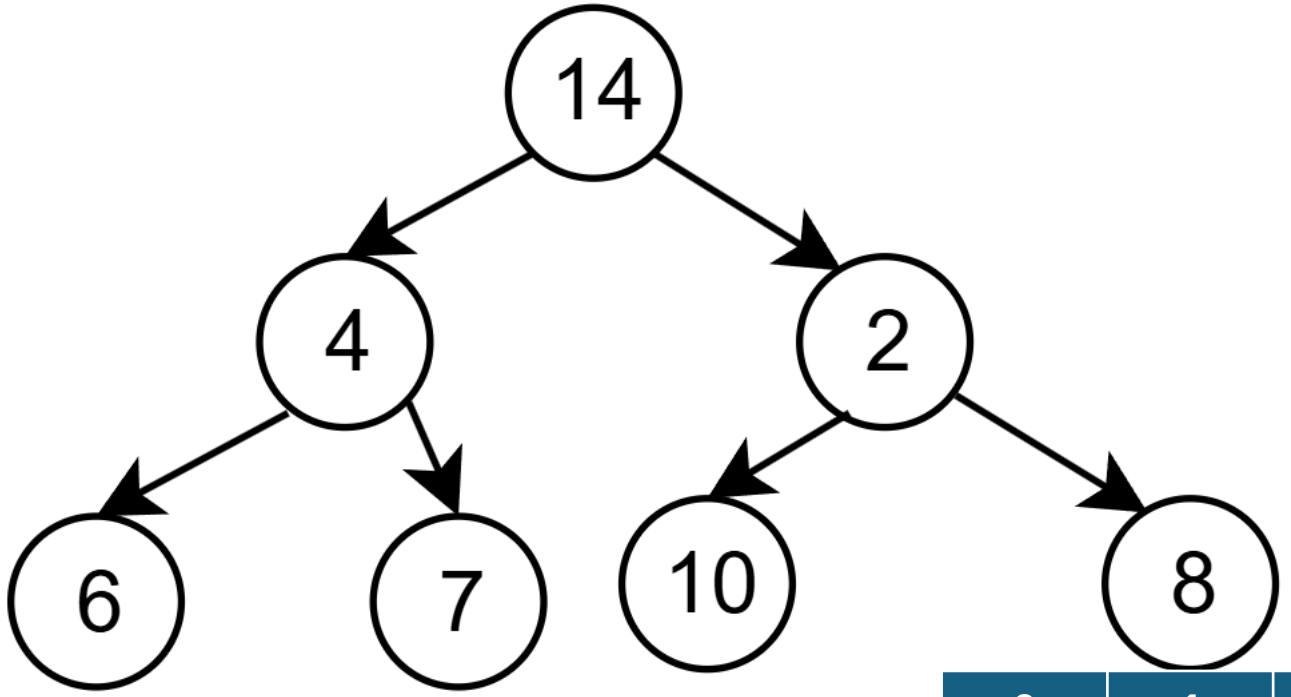
- Let's look at an example. Dequeue once from this 2-Heap. What's the first step?



0	1	2	3	4	5	6	7	8	...
-inf	1	4	2	6	7	10	8	14	...

2-Heaps

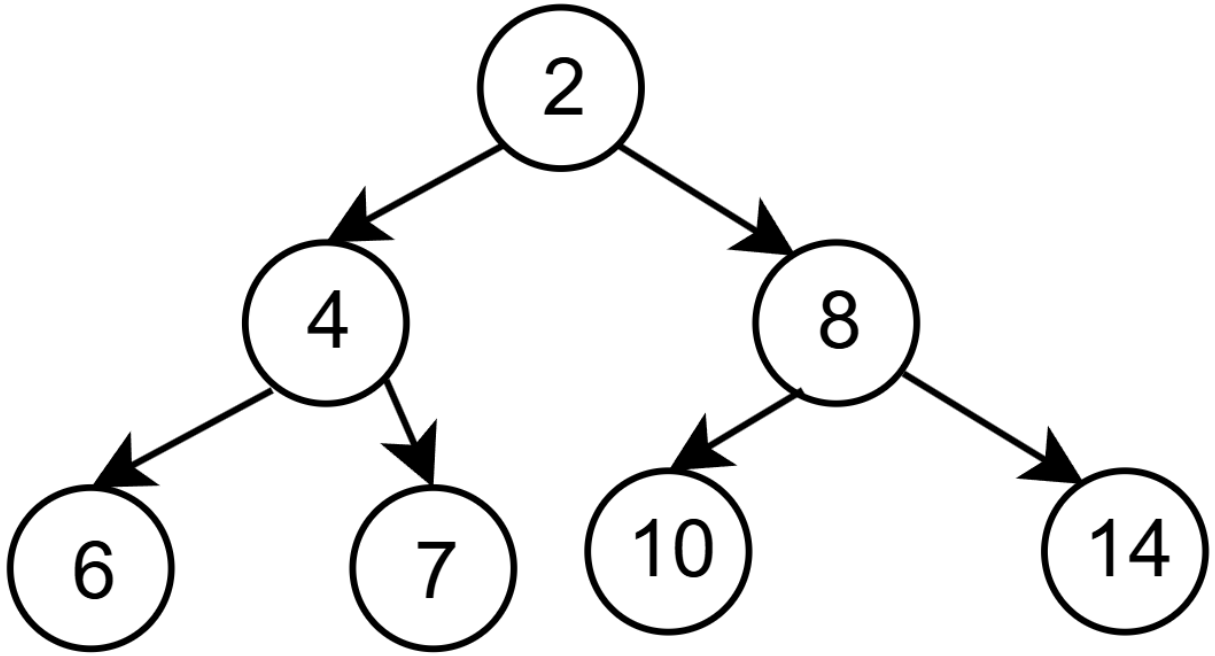
- Let's look at an example. Dequeue once from this 2-Heap. Remove 1 from the 2-Heap, replace it with the right-most item on the last level. Now "bubble down"



0	1	2	3	4	5	6	7	...
-inf	14	4	2	6	7	10	8	...

2-Heaps

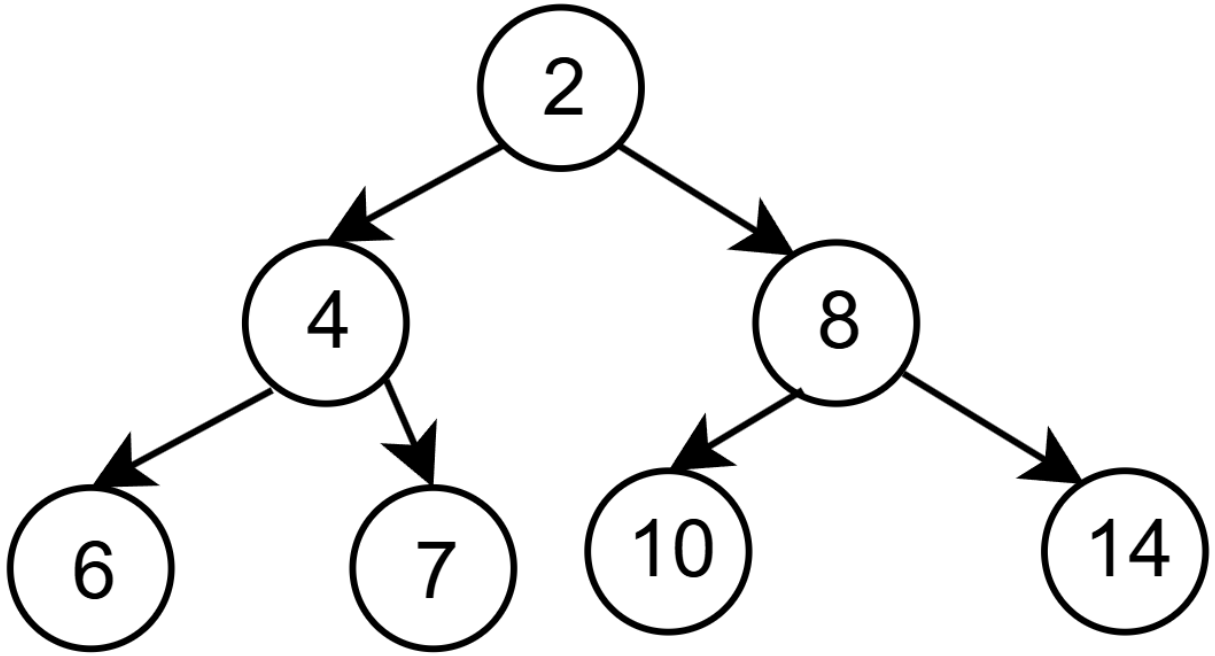
- Let's look at an example. Dequeue once from this 2-Heap. Remove 1 from the 2-Heap, replace it with the right-most item on the last level. Now "bubble down"



0	1	2	3	4	5	6	7	...
-inf	2	4	8	6	7	10	14	...

2-Heaps

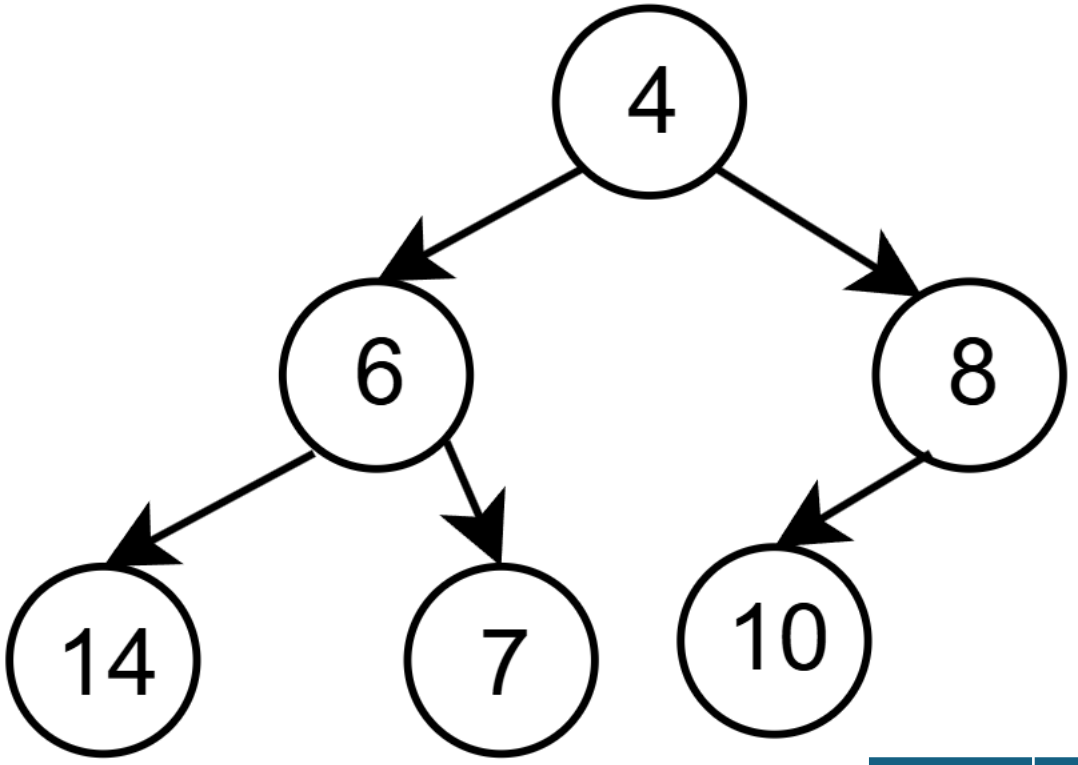
- Let’s do it again. Dequeue again from this 2-Heap.



0	1	2	3	4	5	6	7	…
-inf	2	4	8	6	7	10	14	…

2-Heaps

- Let’s do it again. Dequeue again from this 2-Heap.



0	1	2	3	4	5	6	…
-inf	4	6	8	14	7	10	…

2-Heaps

- So how efficient is this dequeuing? There are two steps, replacing the root, and then bubbling it down (if necessary)
- O(1) to replace the root
- $O(\log_2 n)$ to bubble down
- Overall: $O(\log_2 n)$

2-Heaps

- Let's examine some applications of 2-Heaps. First up, Heapsort!
- Heapsort works like this:

1. Insert the n items into a max heap
2. Dequeue the n items, placing the dequeued value in the spot opened when we replaced the root

- That's all! How about efficiency?
- We have to build the heap first, then destroy it:
- n insertions at $O(\log_2 n)$ time: $O(n * \log_2 n)$
- n deletions at $O(\log_2 n)$ time: $O(n * \log_2 n)$
- So overall (no surprise here): $O(n * \log_2 n)$

2-Heaps

- Second up, a familiar algorithm…
- Dijkstra's! Remember, Dijkstra's maintains Known and Fringe lists. The algorithm has us move the *next-smallest* Fringe distance to Known.
- 2-Heap works well for that! The dequeue operation always gives us the smallest value (for a min heap). However… there is a better option (we'll talk about it in a few slides)

2-Heaps

- Can we do a 3-heap? Where each node can have up to 3 children?
- Yes! But as always, there are pros and cons. Similarly to 2-3 Trees…
- Pros: Squatter tree
- Cons: More children to compare
- A note about 3-heaps: we can still use arrays to represent them. The math required to find children/parent indices just changes slightly

2-Heaps

- Some examples of other types of heaps:
- Fibonacci Heaps: Works extremely well in support Dijkstra's algorithm
- Leftist Heaps: essentially an unbalanced 2-Heap (has a structure with a short right side and an extended left side)
- Skew Heaps: A Leftist Heap that's self-adjusting
- Binomial Queues: Makes Skew Heap insertions more efficient
- …and many more!