TREES

PART TWO

HEAPS AND BALANCED TREES

HEAPS

Binary Trees

- A binary tree is a special tree such that each node has **at most two children**.
  - This means each node can have 0, 1, or 2 children.
- Children are referred to as the left child and right child.

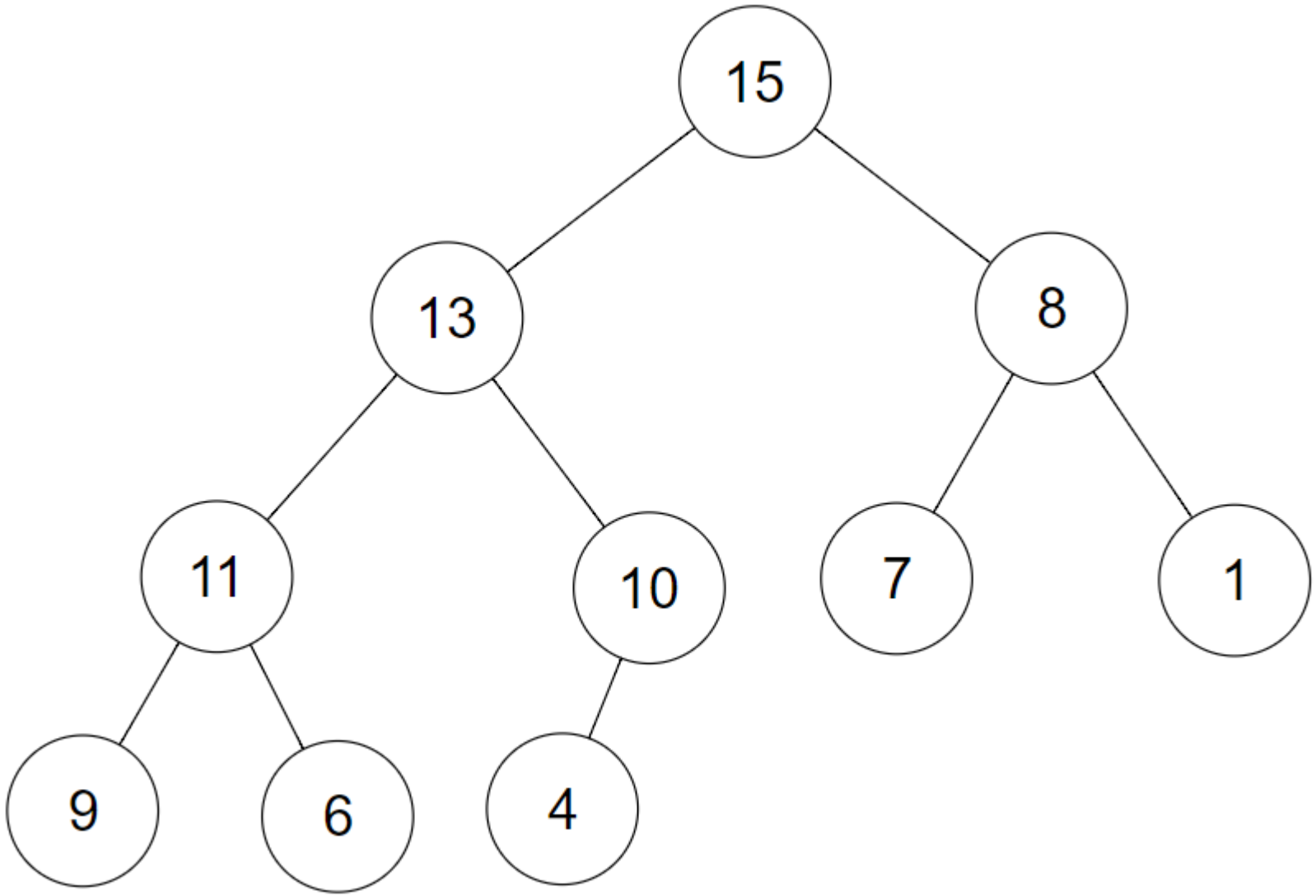
Terminology

- Full: a binary tree such that every non-leaf has **exactly** two children.
- Complete: a binary tree such that:
  - every non-leaf has exactly two children *except* possibly on the second-lowest level
  - on the lowest level, the leaf nodes are as far left as possible

Heaps

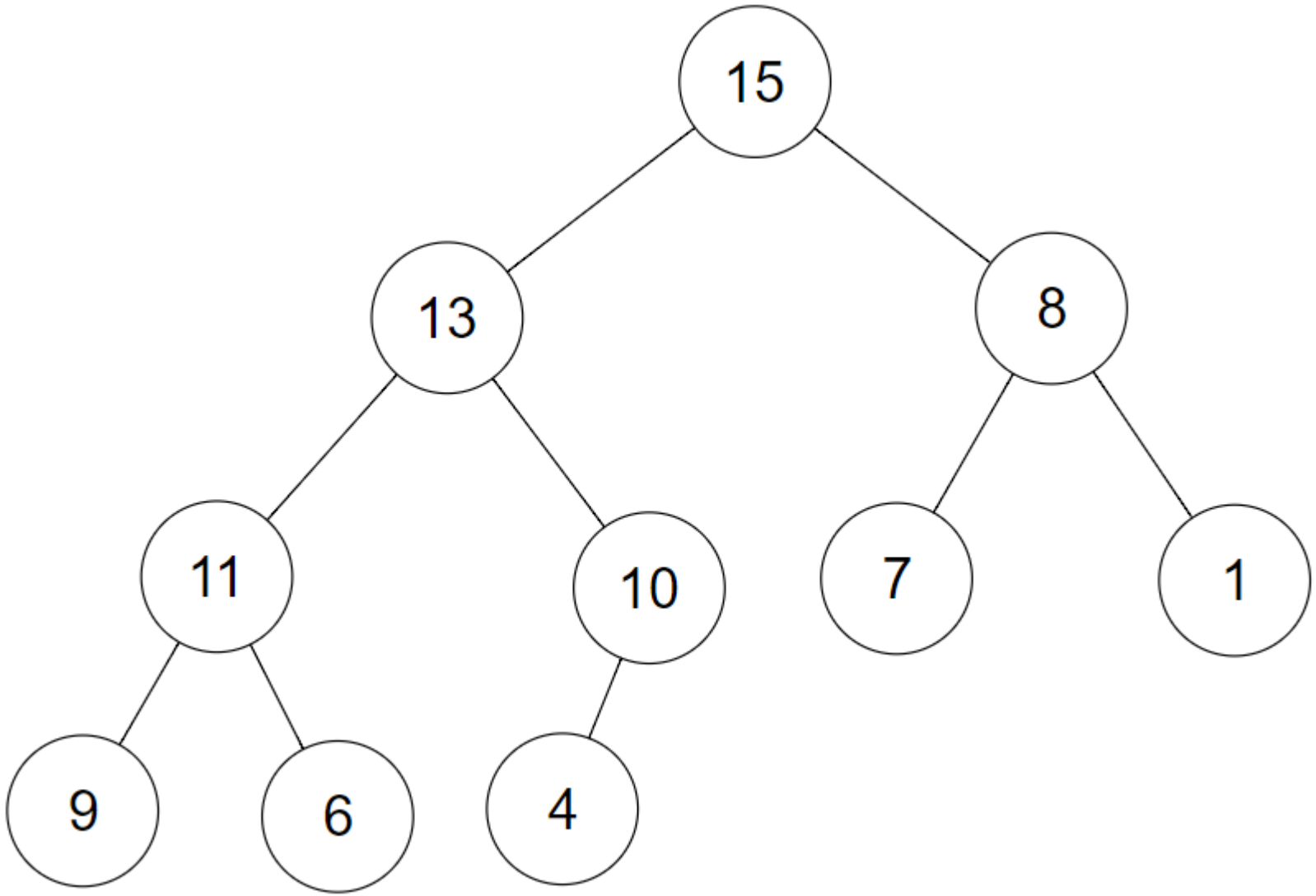
- A tree is a **heap** if:
  - it is complete **AND**
  - the nodes are organized as either a maxheap or minheap
- In a maxheap, each parent node is greater than or equal to all child nodes (in both subtrees).
- In a minheap, each parent node is less than or equal to all child nodes (in both subtrees).

Example



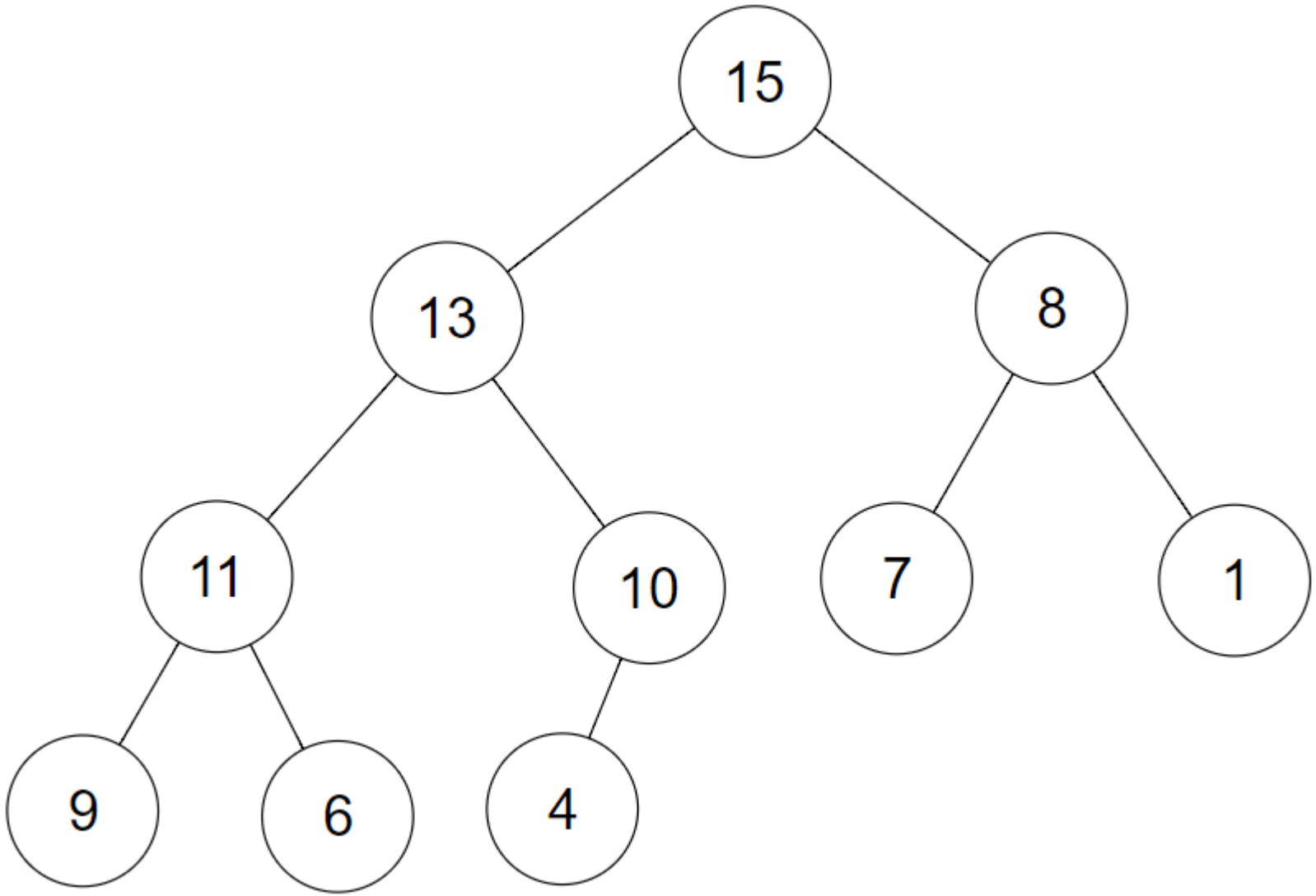
Example

Binary tree?
Complete?
Max structure?



Example

Binary tree? yes
Complete? yes
Max structure? yes



Heaps

- A tree must meet both tests to be a heap- it must be complete and have the min/max structure.

- In a heap, there is no guaranteed relationship between sibling nodes in a heap.
  - You only know that both sibling nodes are smaller/greater than their parent.
  - You don’t know if a left child or right child is smaller/larger.

- We’ll use maxheaps in our examples. But all of the material could apply to minheaps too!

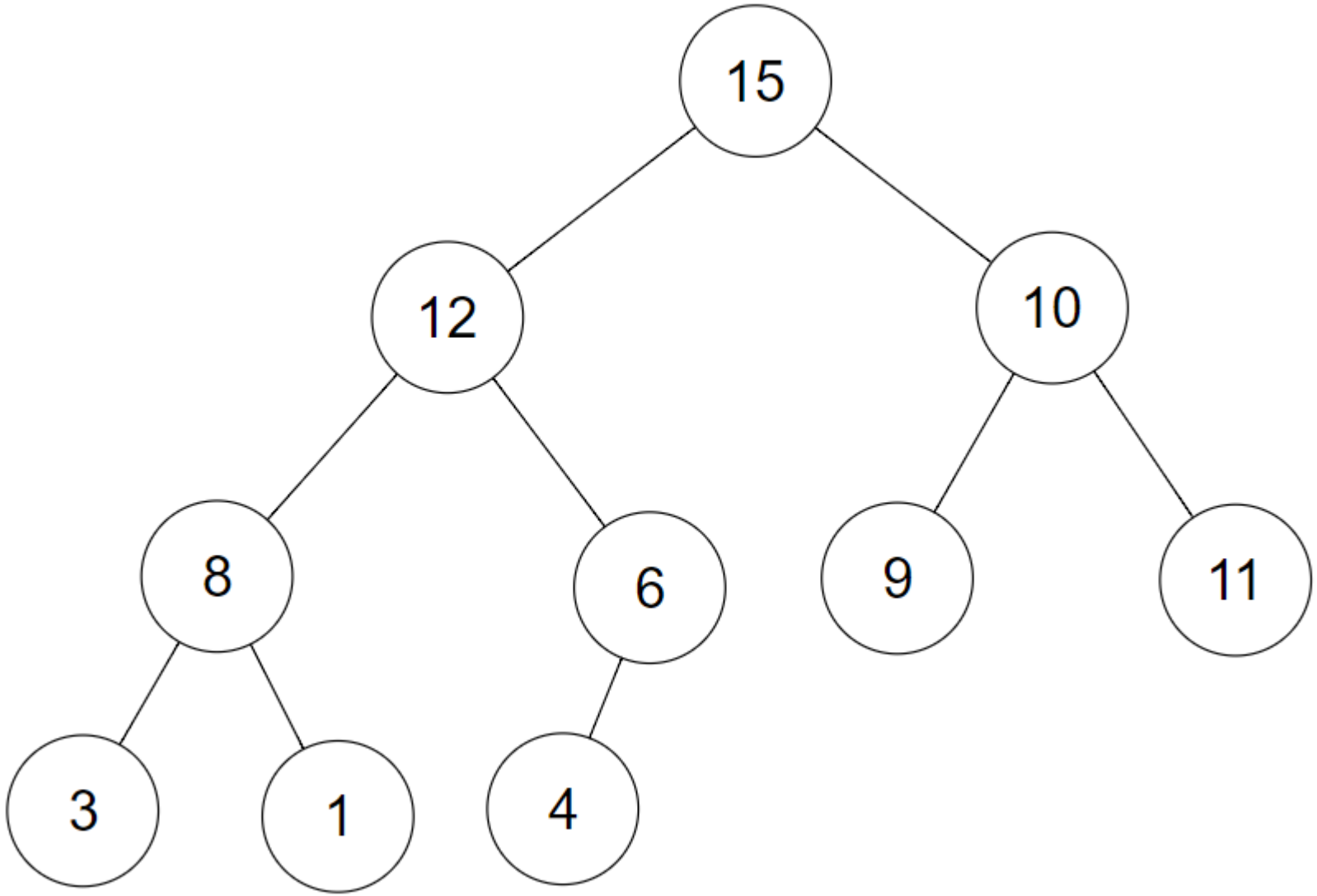
Maxheaps

- The greatest node is always at the root
- Any subtree in a maxheap is also a maxheap.

- There is no relationship *between* subtrees.

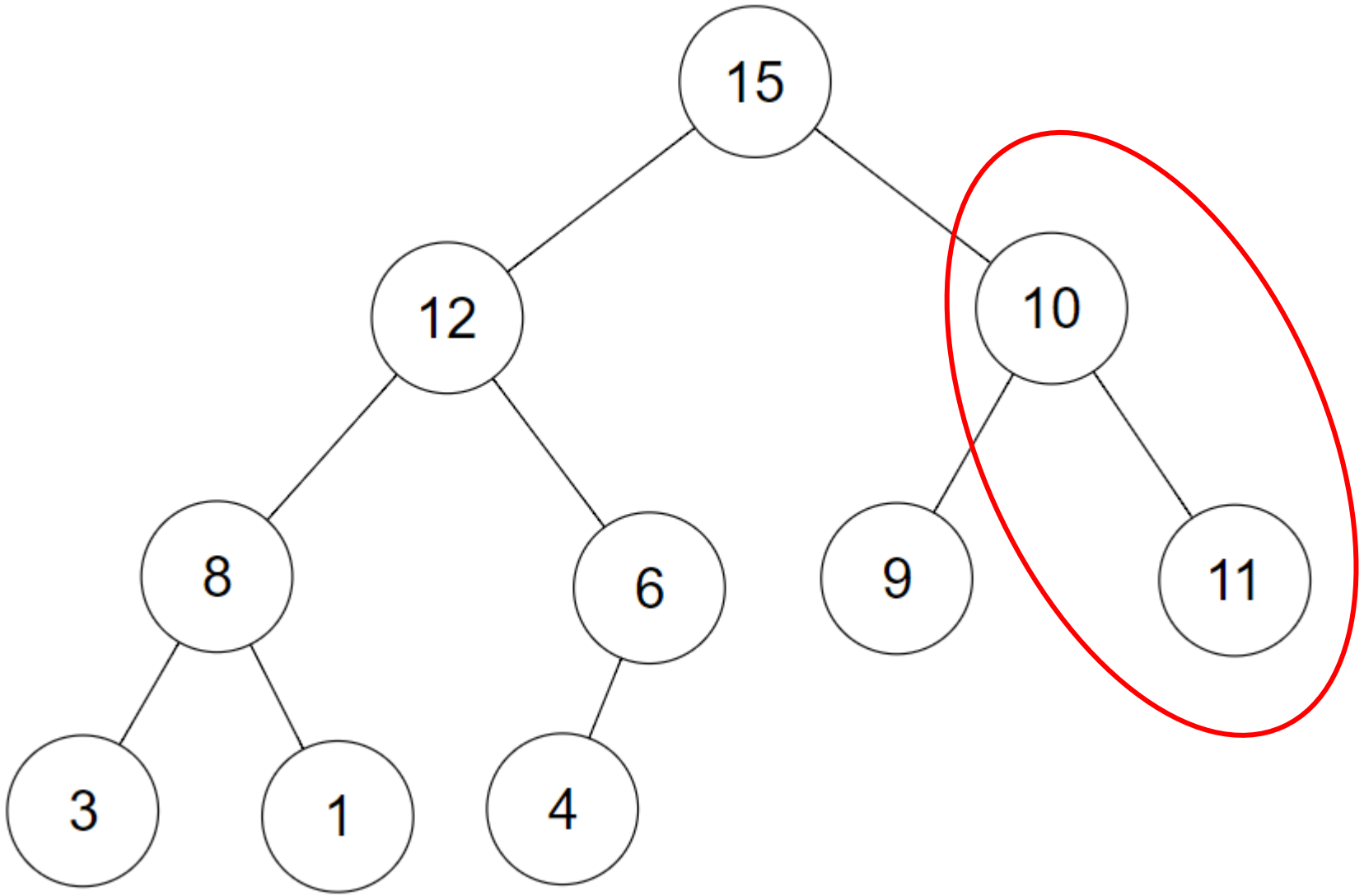
Example

Binary tree?
Complete?
Max structure?



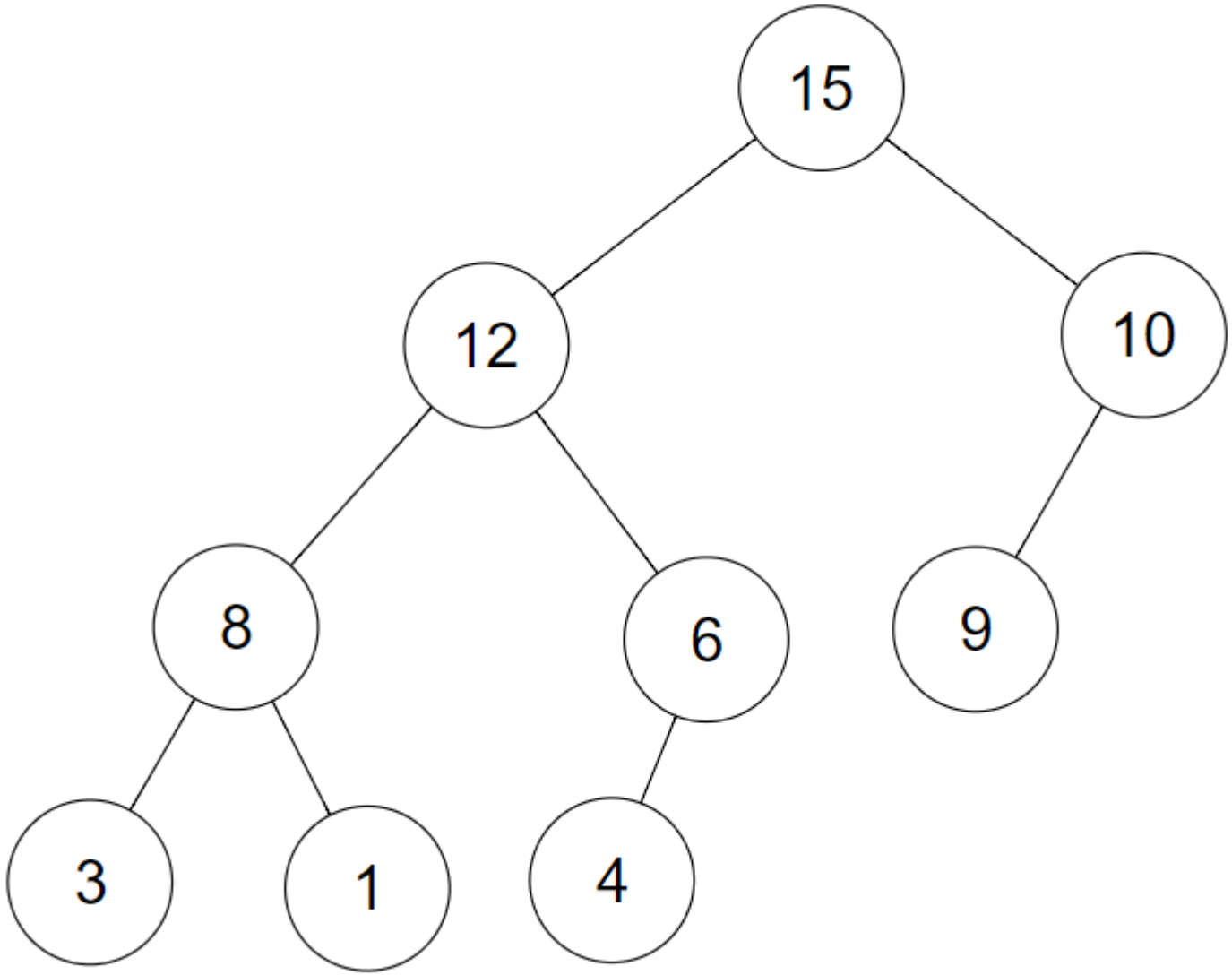
Example

Binary tree? yes
Complete? yes
Max structure? no!



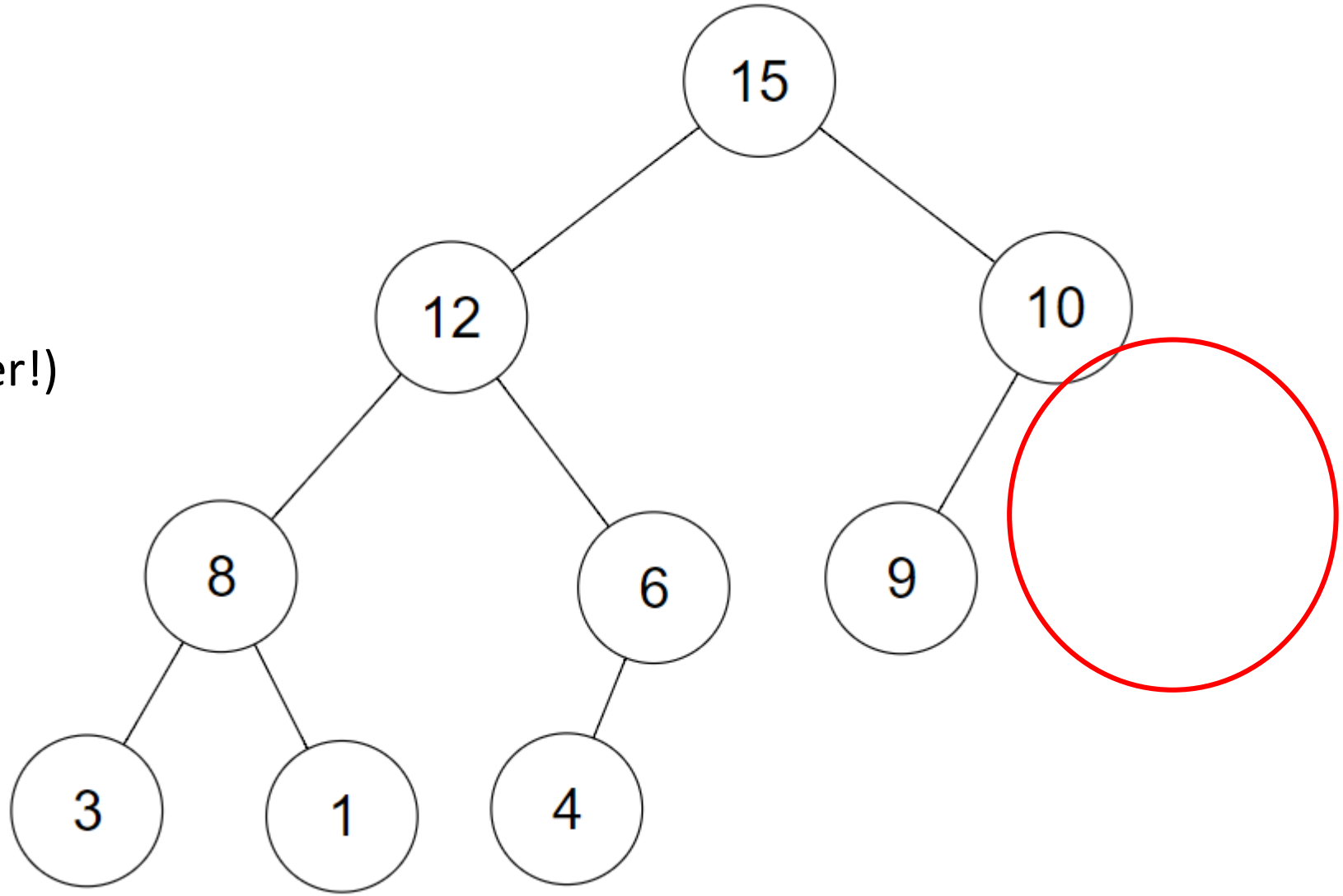
Example

Binary tree?
Complete?
Max structure?



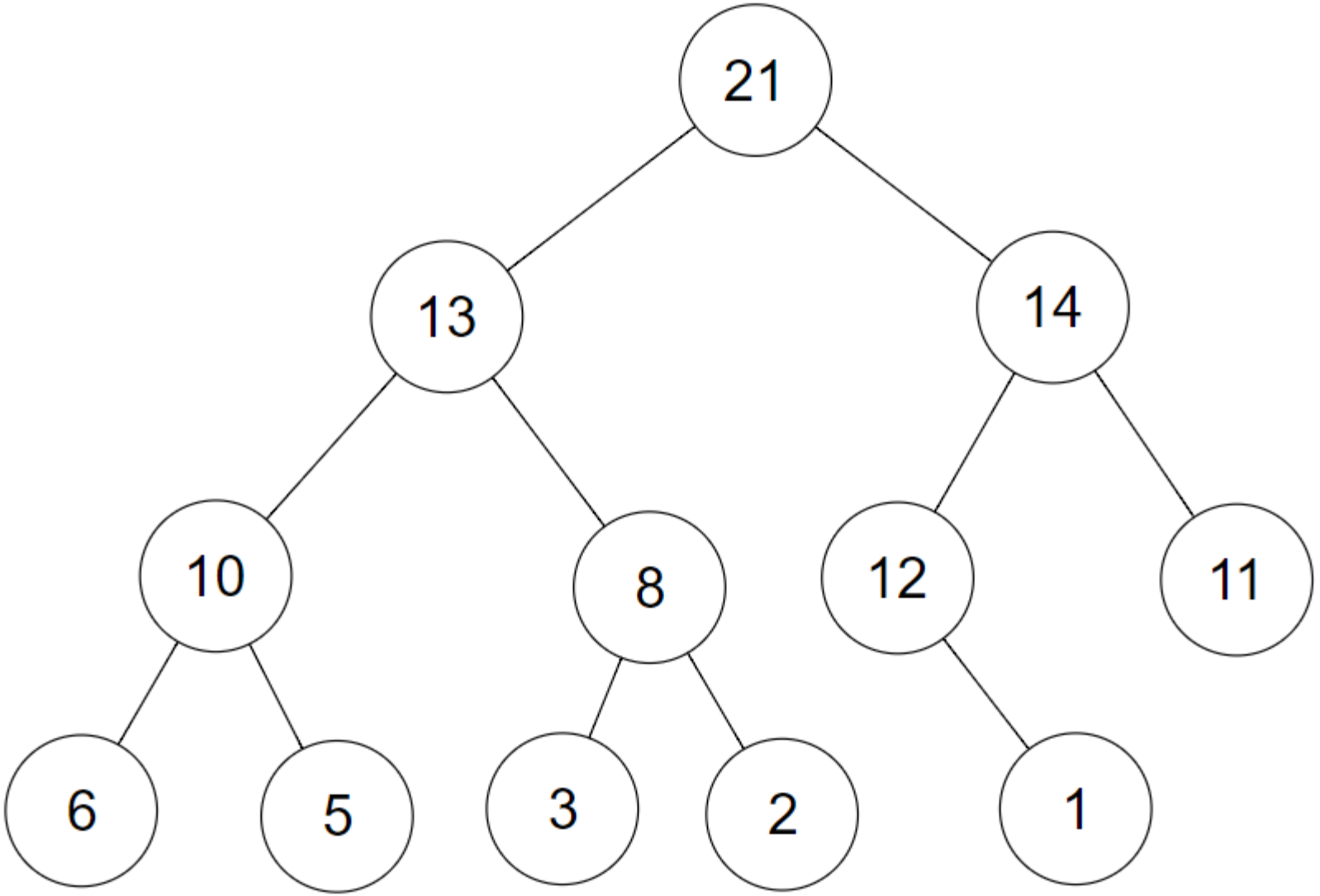
Example

Binary tree? yes
Complete? no!
Max structure? (doesn't matter!)



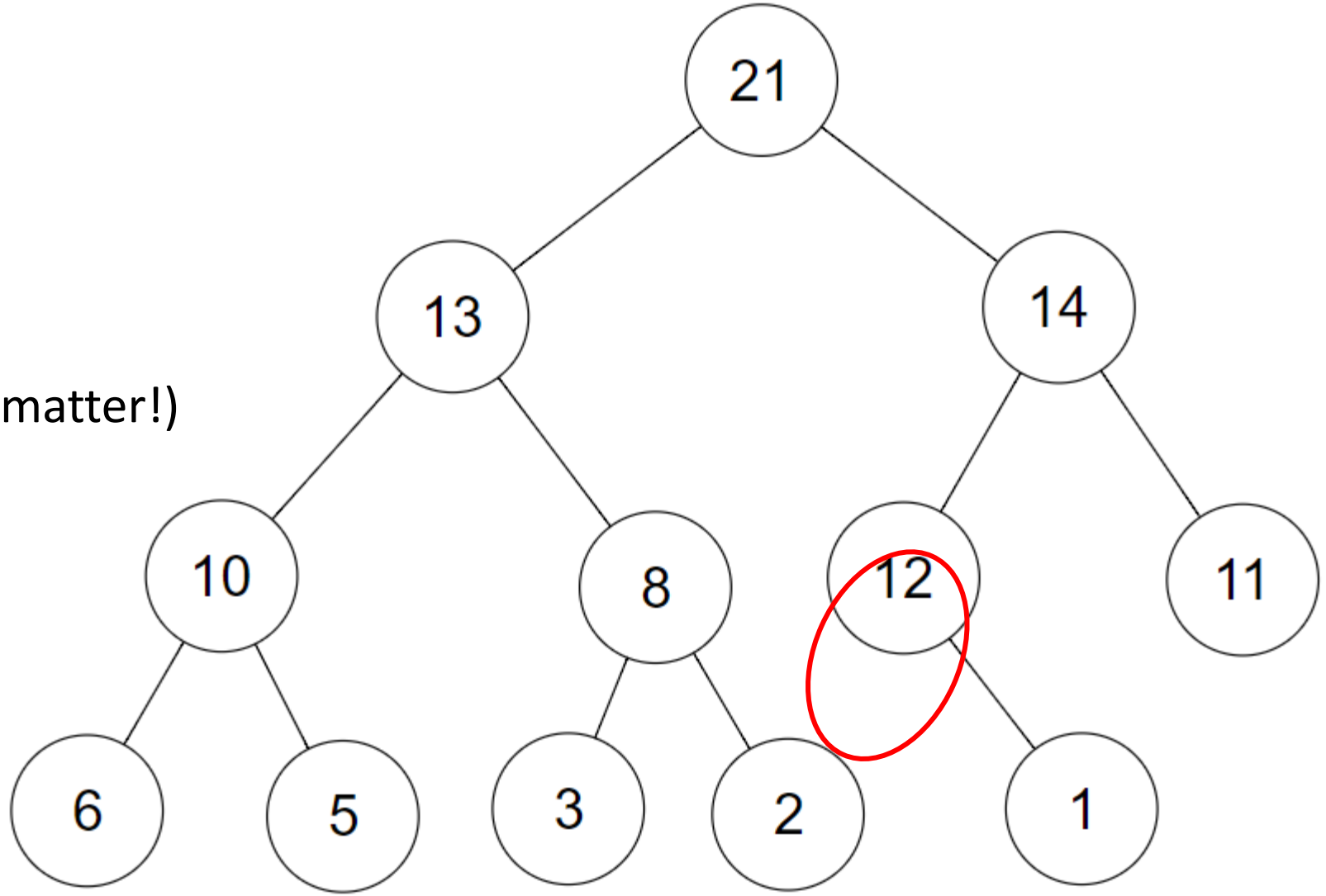
Example

Binary tree?
Complete?
Max structure?



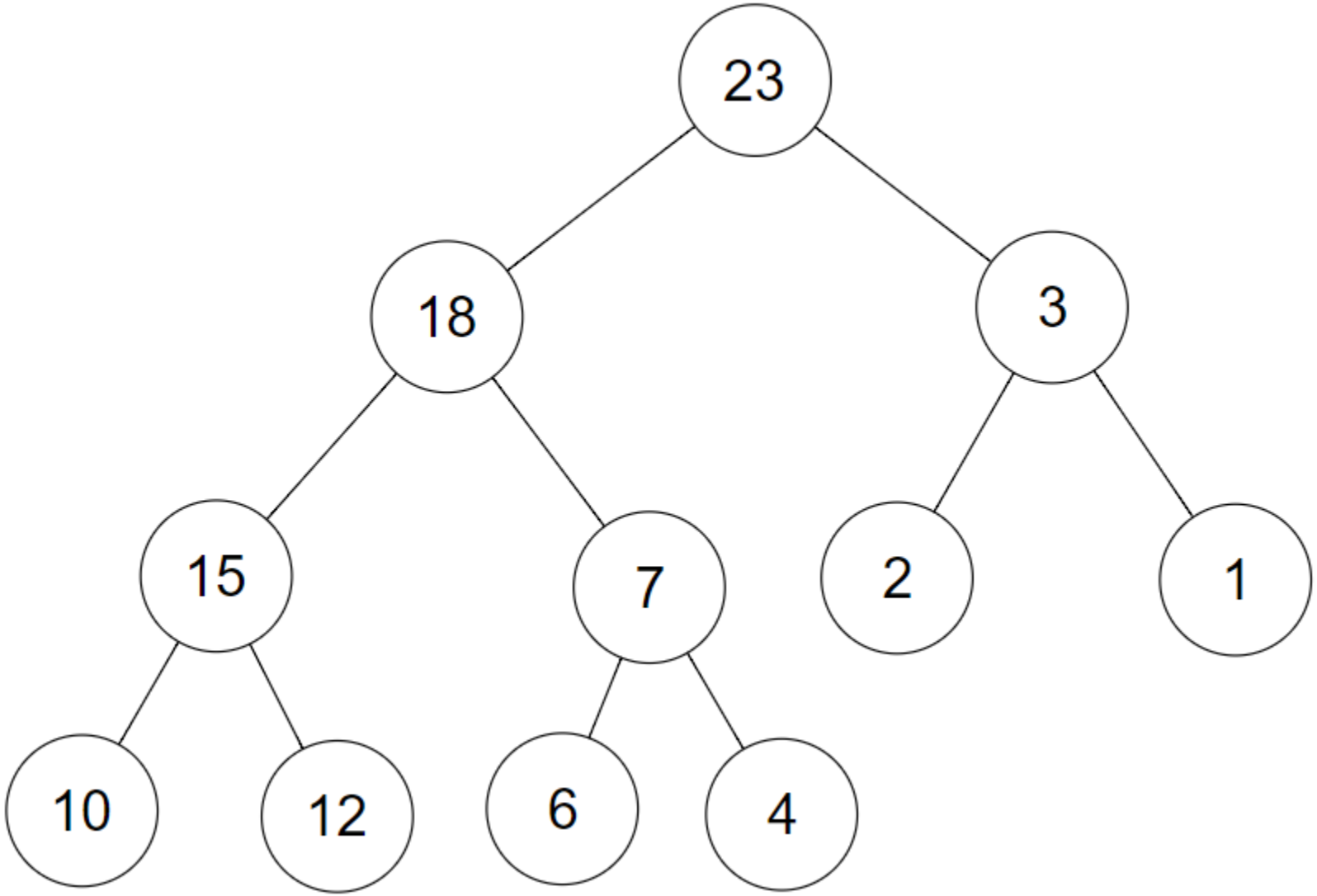
Example

Binary tree? yes
Complete? no!
Max structure? (doesn't matter!)



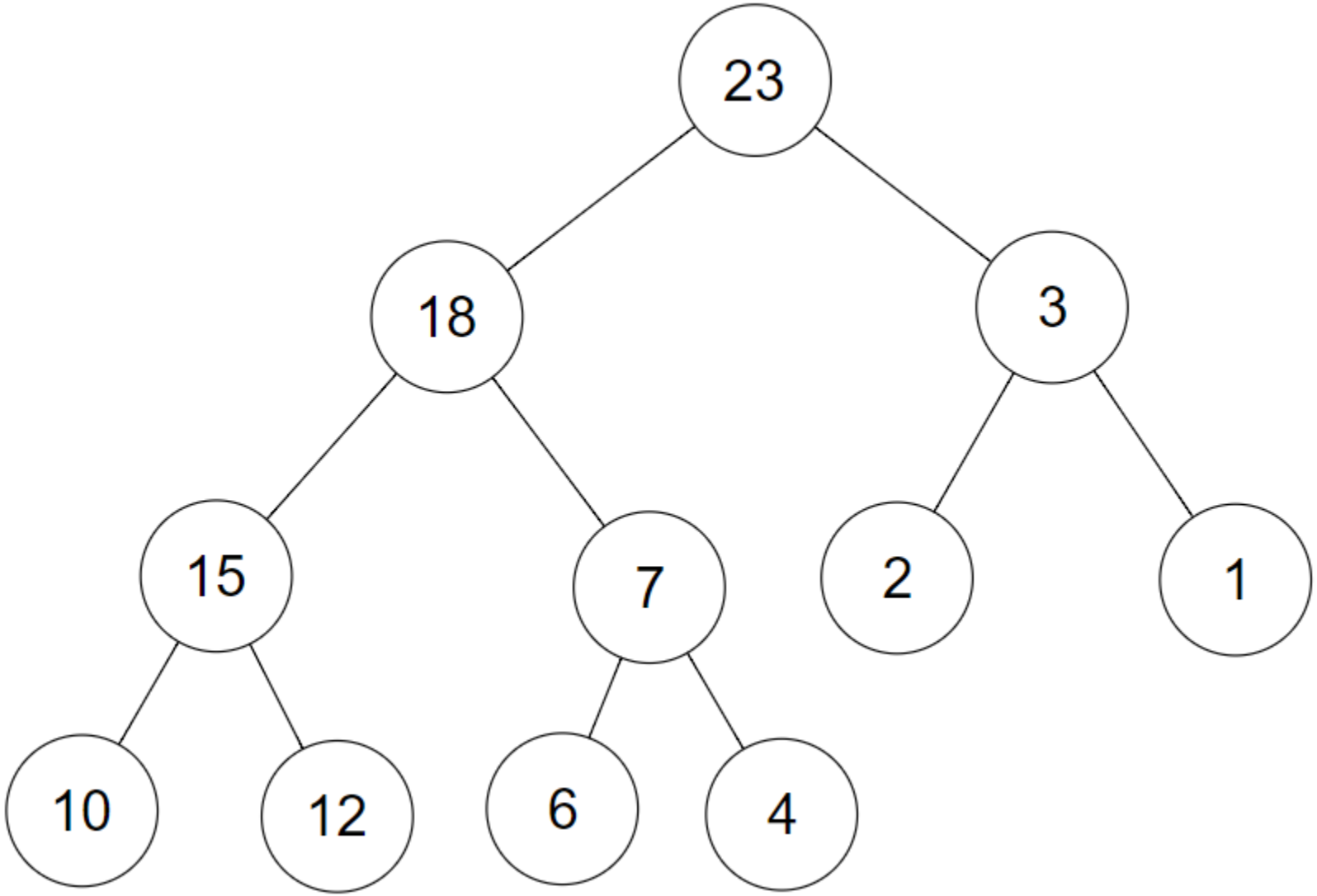
Example

Binary tree?
Complete?
Max structure?



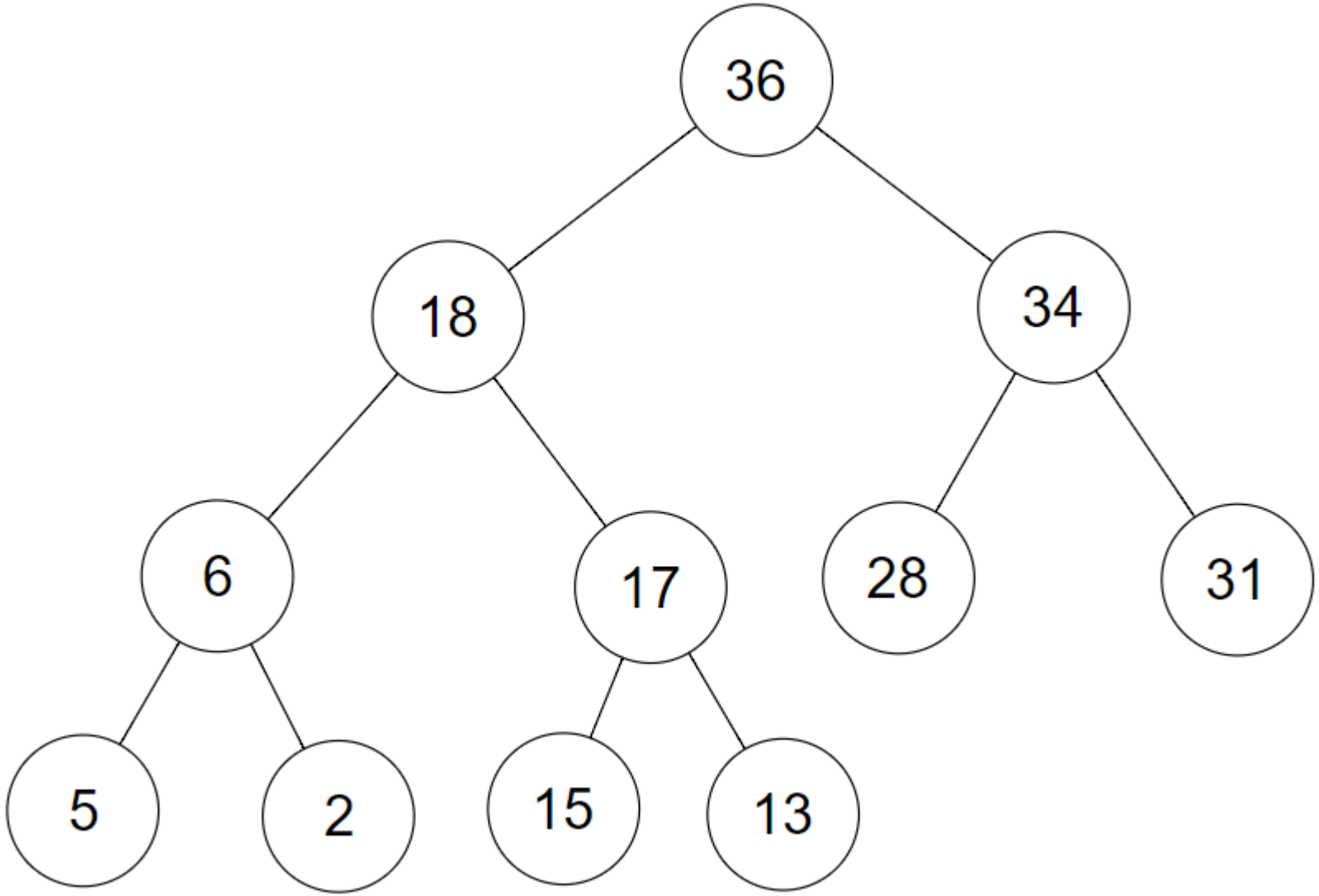
Example

Binary tree? yes
Complete? yes
Max structure? yes



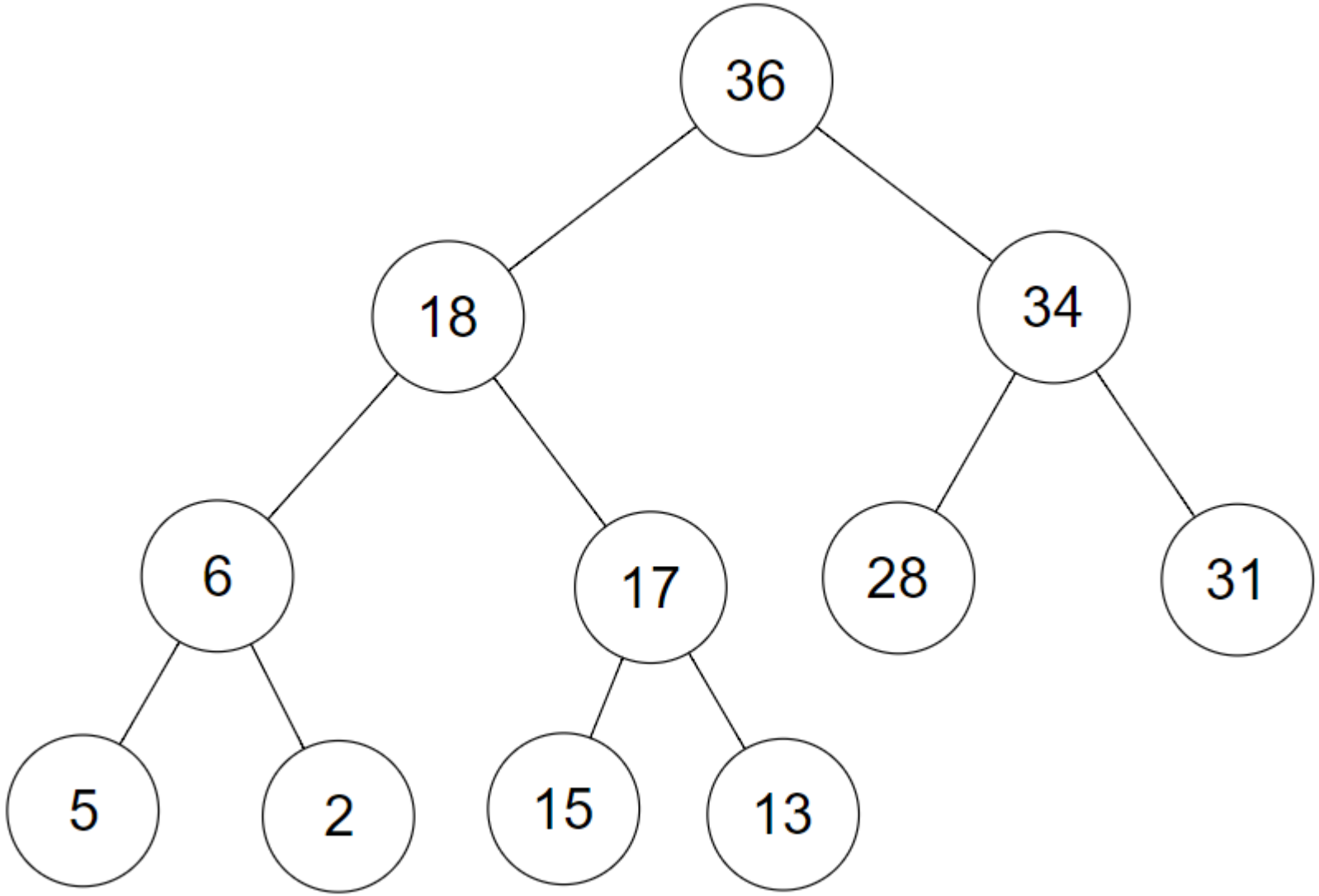
Example

Binary tree?
Complete?
Max structure?



Example

Binary tree? yes
Complete? yes
Max structure? yes

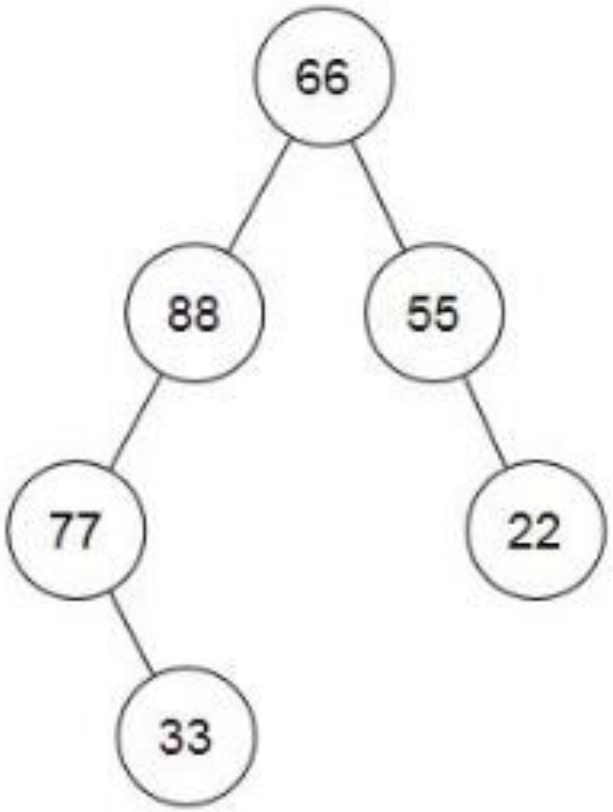


TREE IMPLEMENTATIONS

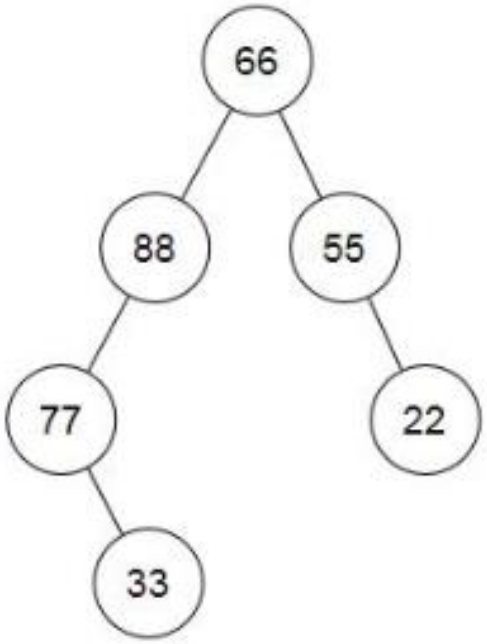
Implementing Trees (and Heaps) with Arrays

- We can implement any tree with the *heapform array*.
- Setup:
  - size of array: maximum number of nodes + 1
  - leave position 0 empty
  - put the root in position 1
- The general rule is:
  - store the left child of the node in array[i] at position array[2i]
  - store the right child of the node in array[i] in position array[2i+1]
  - The parent of any node is at array[i/2] (note that the integer division *truncates* the decimal value)

Example (Binary Tree- Not a Heap!)

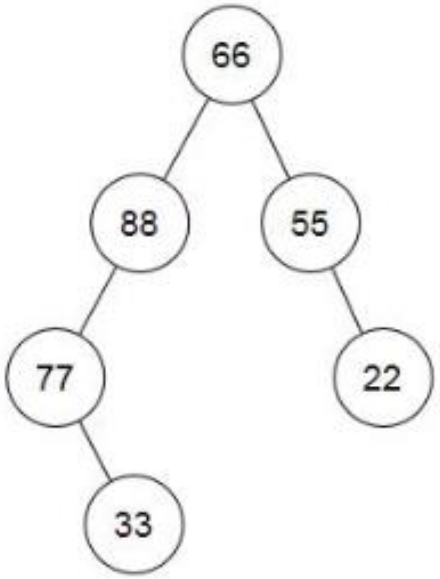


Example (Binary Tree- Not a Heap!)



0	1	2	3	4	5	6	7	8	9
-	66	88	55	77	*	*	22	*	33

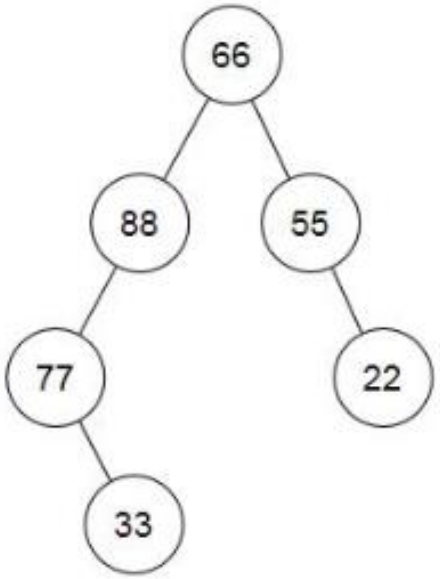
Example



0	1	2	3	4	5	6	7	8	9
-	66	88	55	77	*	*	22	*	33

- Children of 66 (index 1) are at 2*1=2 and (2*1)+1=3
- Children of 88 (index 2) are at 2*2=4 and (2*2)+1 = 5

Example



0	1	2	3	4	5	6	7	8	9
-	66	88	55	77	*	*	22	*	33

- Parent of 88 (index 2) is at 2/2 = 1
- Parent of 55 (index 3) is at 3/2 = 1
- Parent of 22 (index 7) is at 7/2 = 3

Adding to a Heap

- Two approaches: upheaping and reheaping
- Upheaping: put the value in the correct leaf (to keep the tree complete) then repeatedly swap (trade places with) the parent value until the new value is in the correct location based on the max property
- Reheaping: repeatedly transform smaller semiheaps (or subheaps) into heaps

Upheaping

1. Put the value in the correct leaf position.
2. Check if the value is less than its parent.
   a) If yes- you're done!
   b) If no, swap the new value with the parent. Return to step 2.

Upheaping

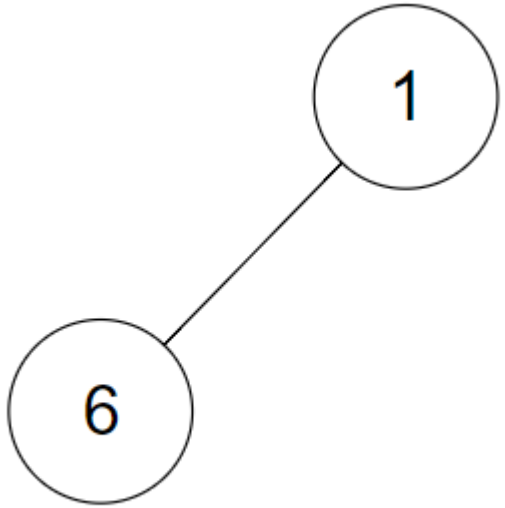
- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using upheaping.

- Insert 1.
  - This is a maxheap.
  - We’re done.

Upheaping

- Example: Insert 1, 6, 2, 7, 4 into a maxheap using upheaping.

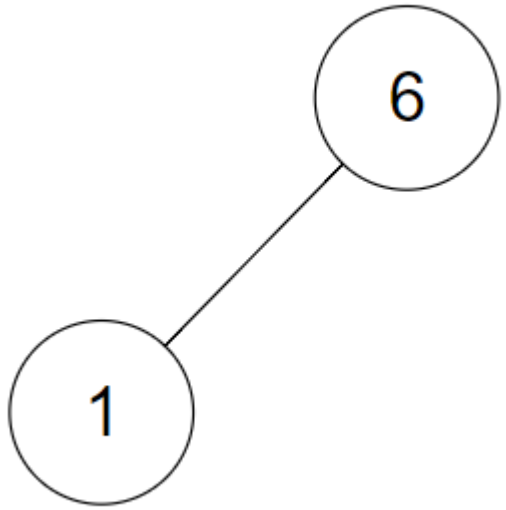
- Insert 6.
  - Make it the left child of A to keep the tree complete.
  - Is 6 < 1? No!



Upheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using upheaping.

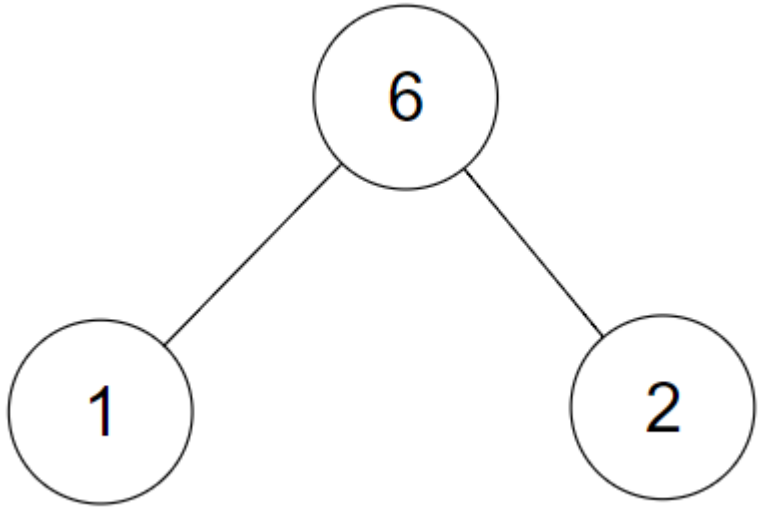
- Swap 6 with 1.
- Is 6 < parent? No parent, so we're done.



Upheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using upheaping.

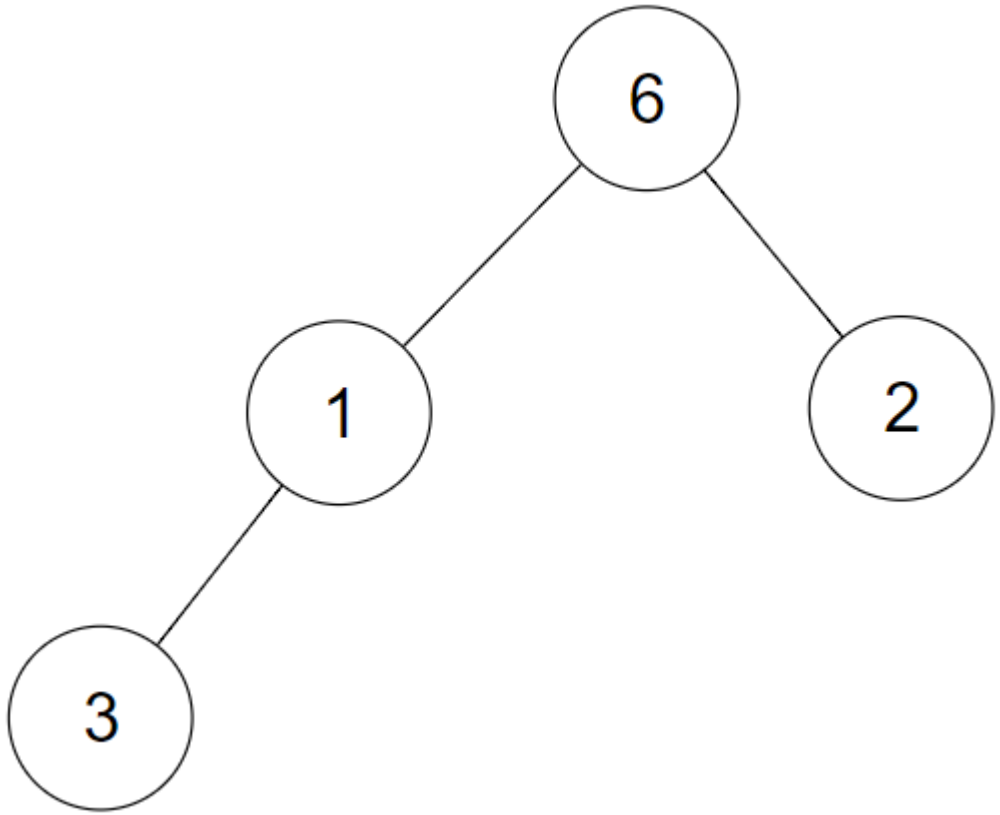
- Insert 2.
  - The tree is complete.
  - Is 2 < 6? Yes, so we're done.



Upheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using upheaping.

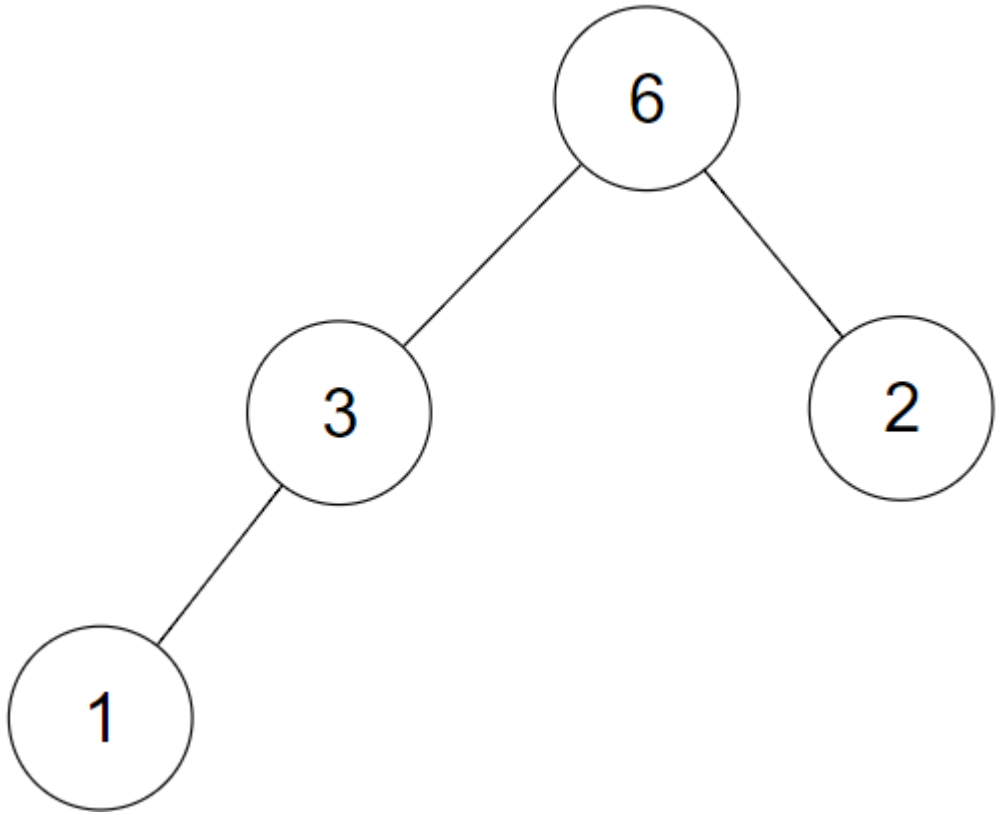
- Insert 3.
  - The tree is complete.
  - Is 3 < 1? No.



Upheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using upheaping.

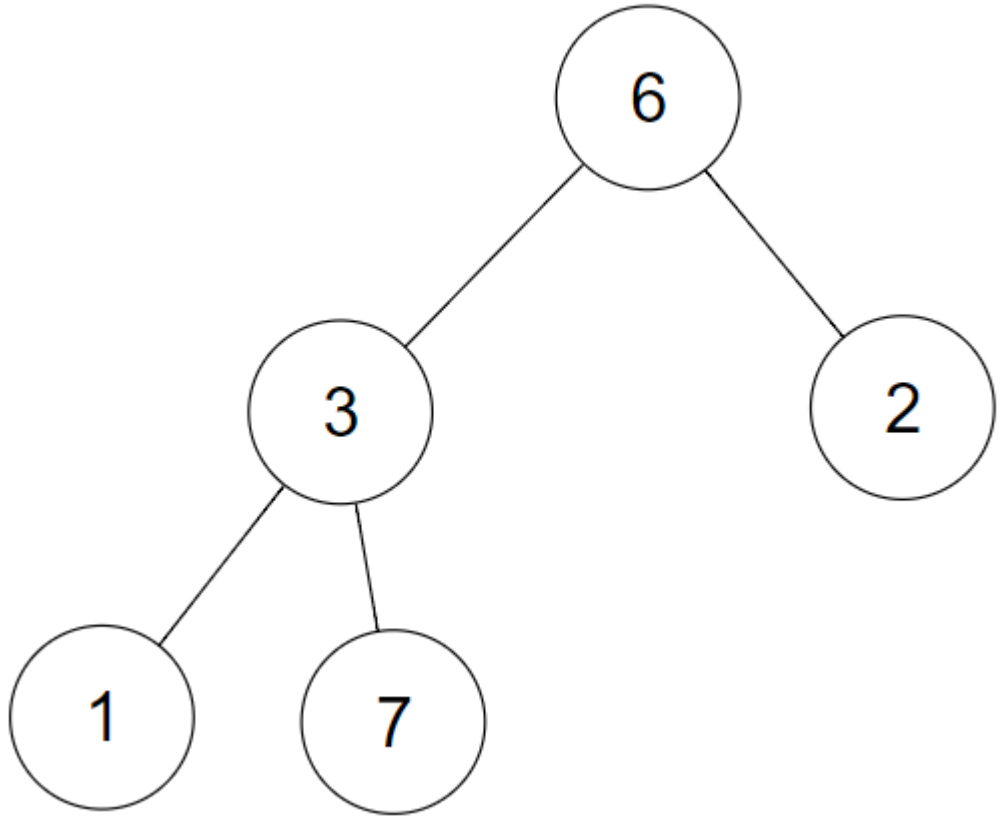
- Swap 3 and 1.
- Is 3 < 6? Yes, so we're done.



Upheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using upheaping.

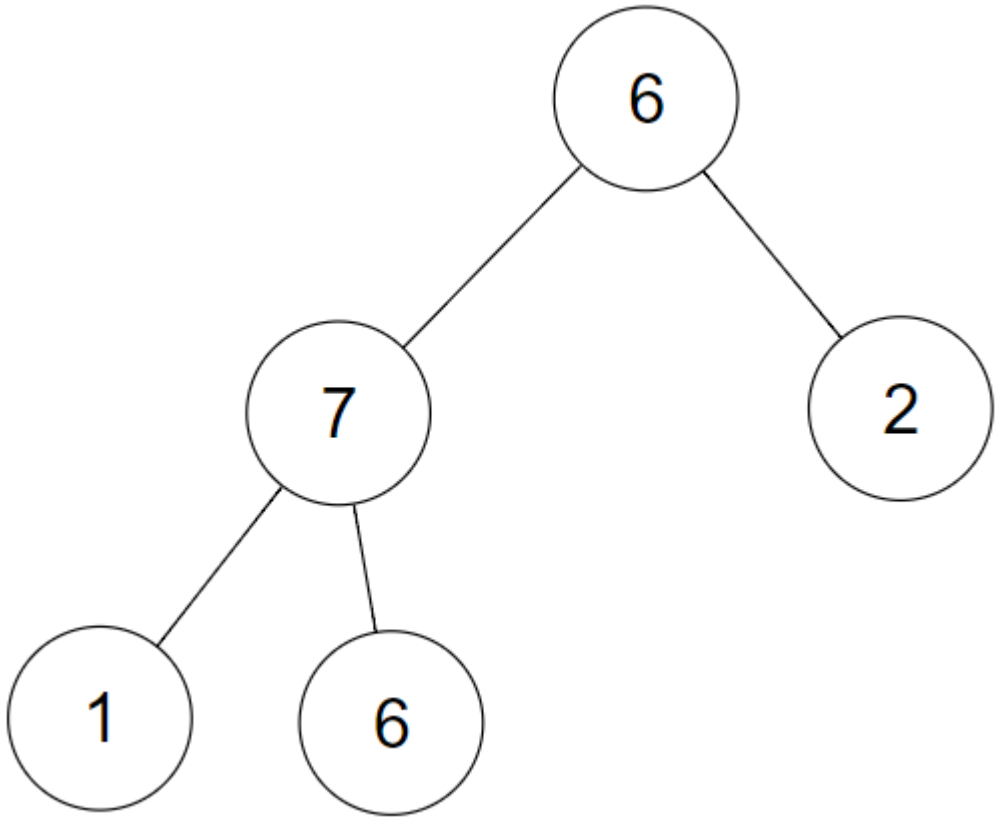
- Insert 7.
  - The tree is complete.
  - Is 7 < 3? No.



Upheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using upheaping.

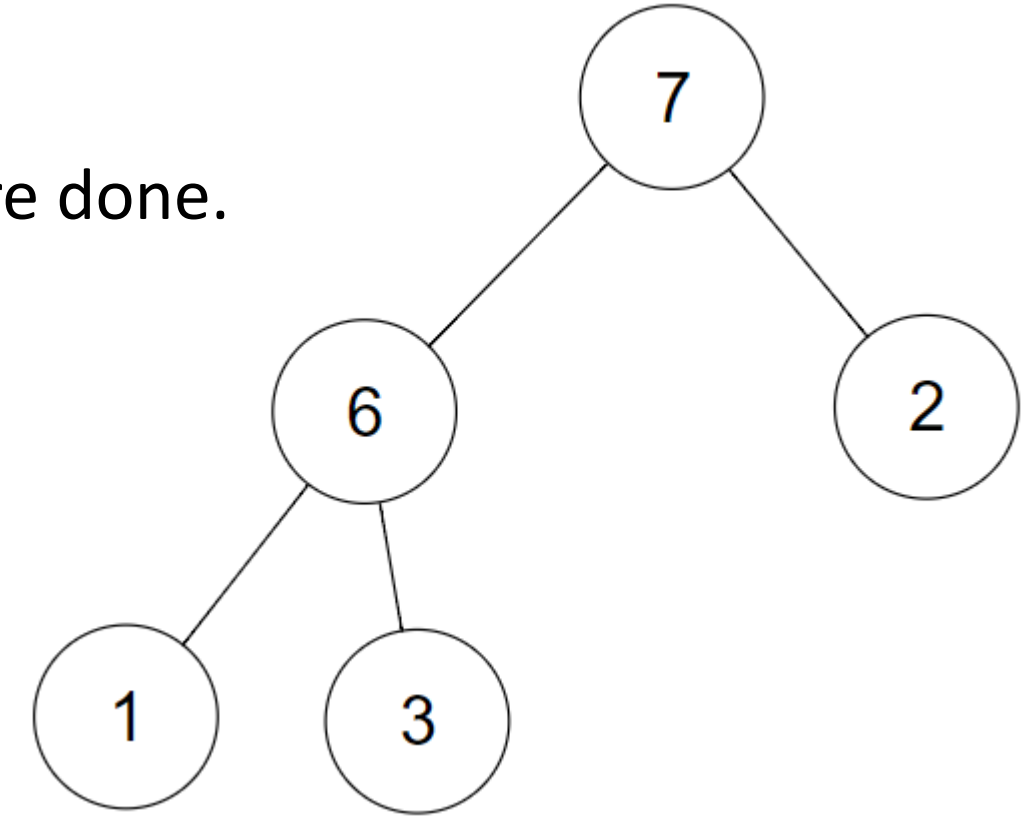
- Swap 7 and 3.
- Is 7 < 6? No.



Upheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using upheaping.

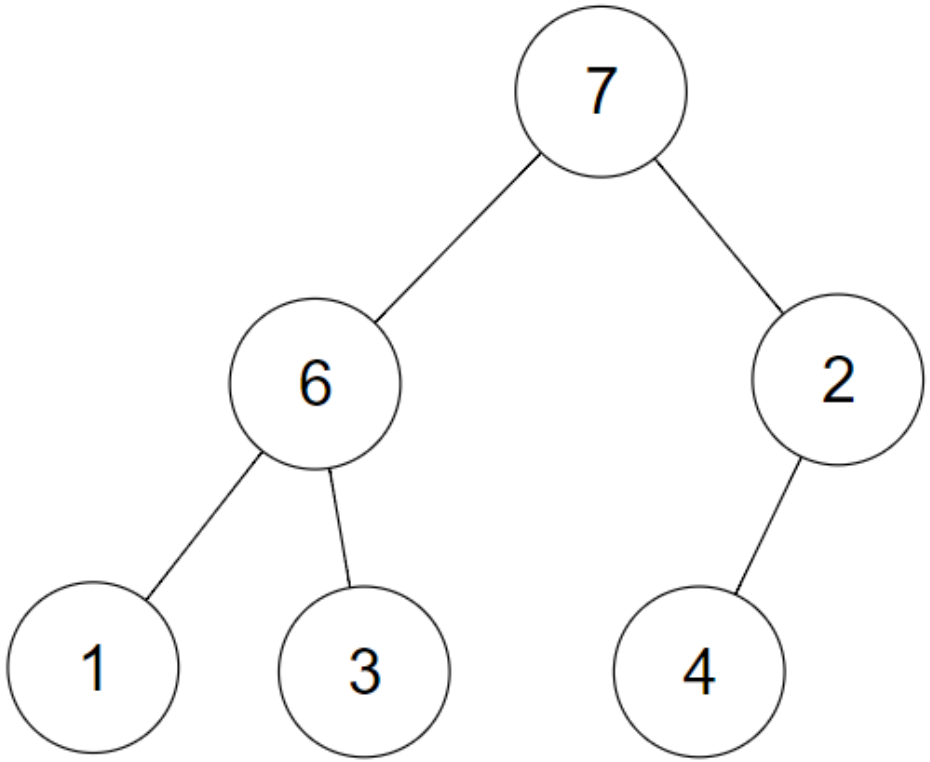
- Swap 7 and 6.
- Is 7 < parent? No parent so we're done.



Upheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using upheaping.

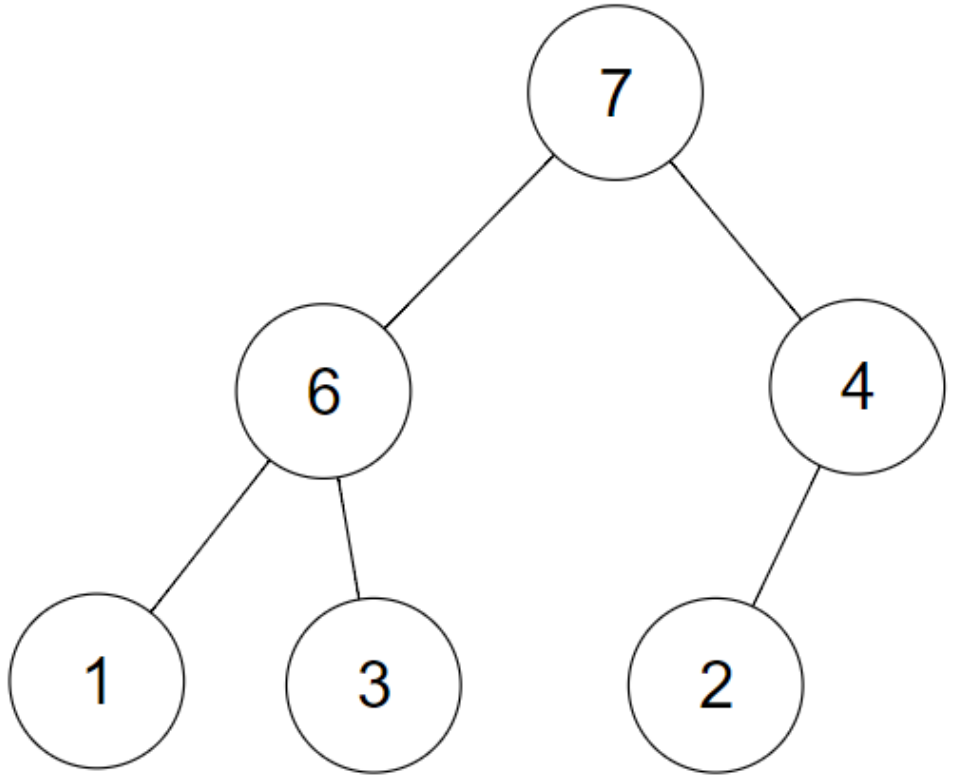
- Insert 4.
  - The tree is complete.
  - If 4 < 2? No.



Upheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using upheaping.

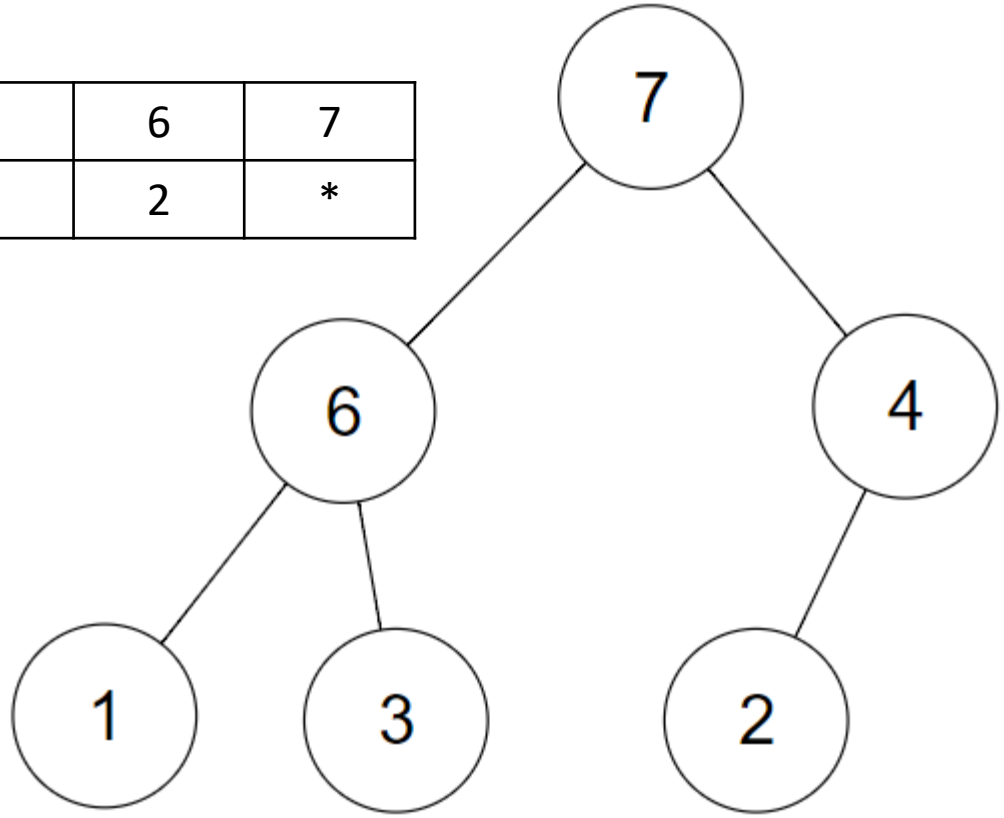
- Swap 4 and 2.
- Is 4 < 7? Yes, so we're done.



Upheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using upheaping.

0	1	2	3	4	5	6	7
-	7	6	4	1	3	2	*



Upheaping

- This upheap method is O(log n).
- So to create a heap of n nodes, you'd have to invoke this algorithm n times.
- Creating a heap using upheaping is O(n log n).

Upheaping

- With upheap, we have a valid heap after every single addition.
- This is necessary if adding a single element, but when building a complete from a set of elements, it does more work than is necessary.

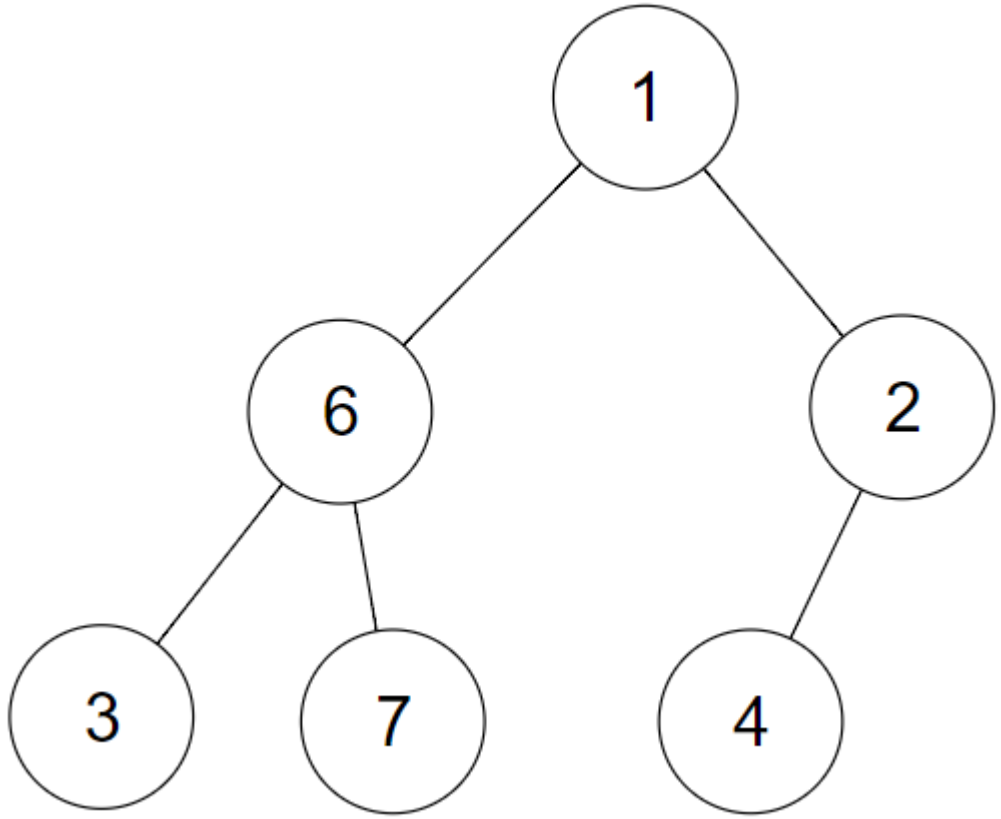
Reheaping

- A leaf is a valid heap.
  - Any single-node tree would be a valid heap!
- If we can take advantage of this, we can focus only on the interior nodes and root to check for what needs to be swapped.
- With this approach, we'll only need to invoke the reheap method n/2 times, meaning creating a heap with reheap is O(n).

Reheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using reheaping.

- Build the tree in insertion order so that it is complete.

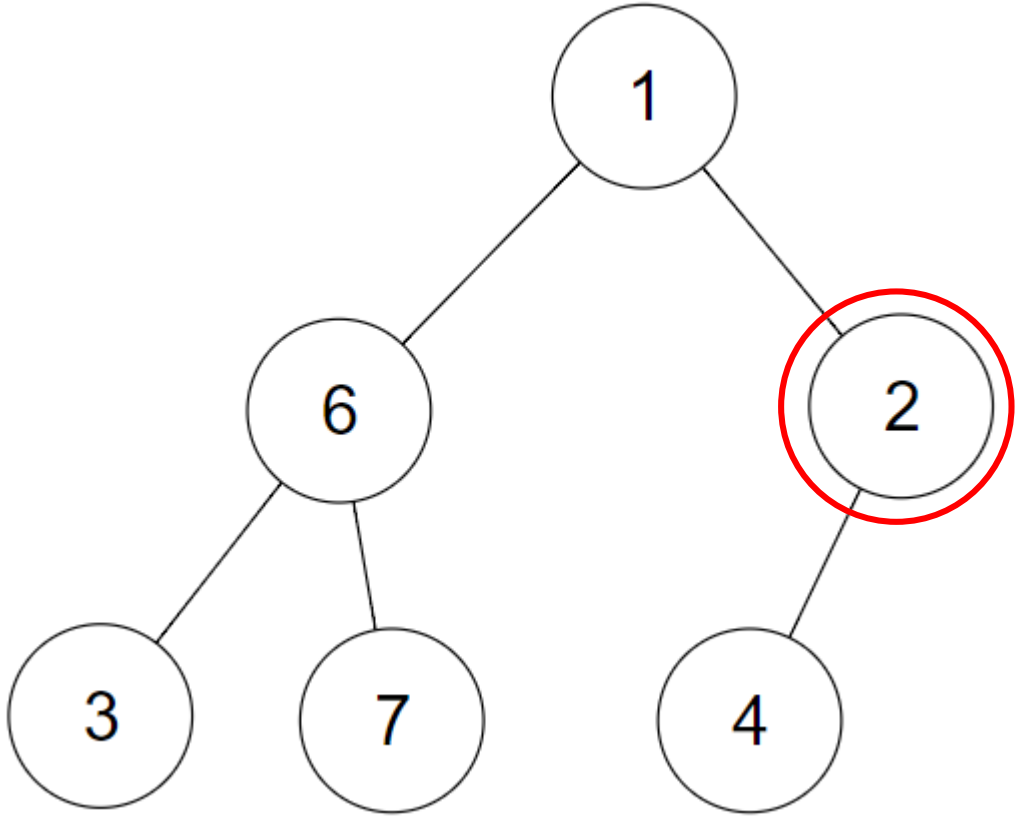


0	1	2	3	4	5	6	7
-	1	6	2	3	7	4	*

Reheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using reheaping.

- Start backwards from the lowest interior node.
  - This is the right-most node in the level above the leaves.
  - This is at location n/2: 6/2=3
  - n is the number of nodes.
  - Remember how integer division works!

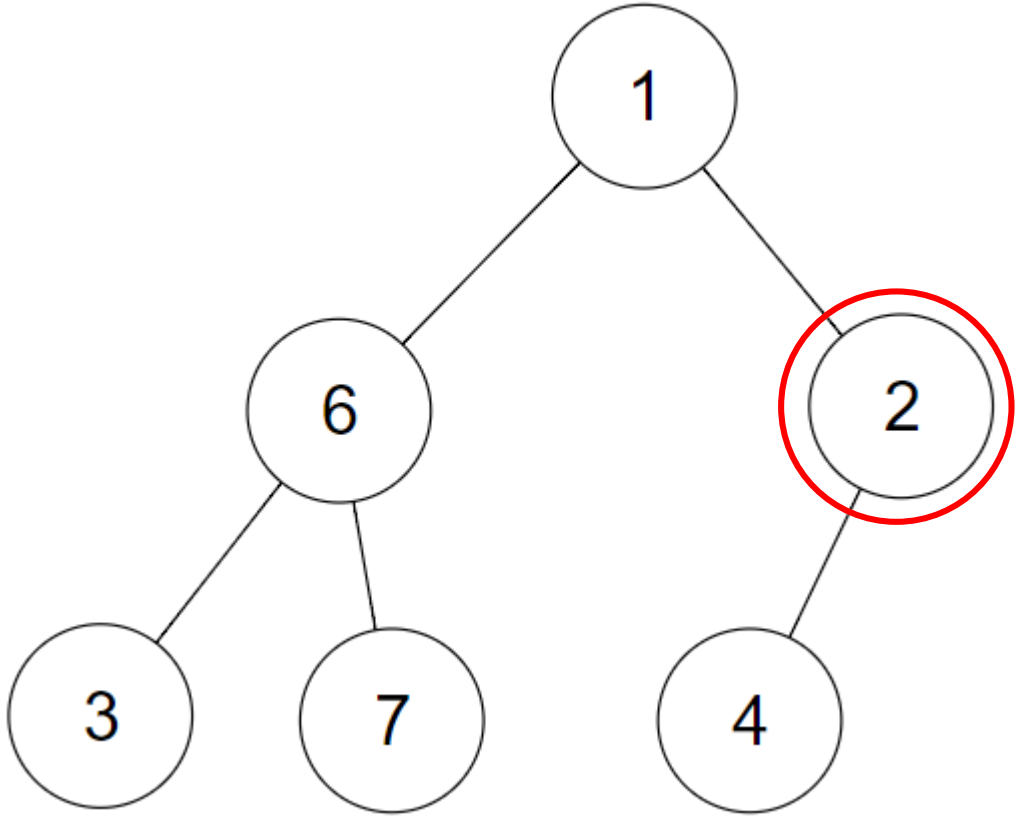


0	1	2	3	4	5	6	7
-	1	6	2	3	7	4	*

Reheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using reheaping.

- Starting at index 3, if the element there is less than its child, swap it.

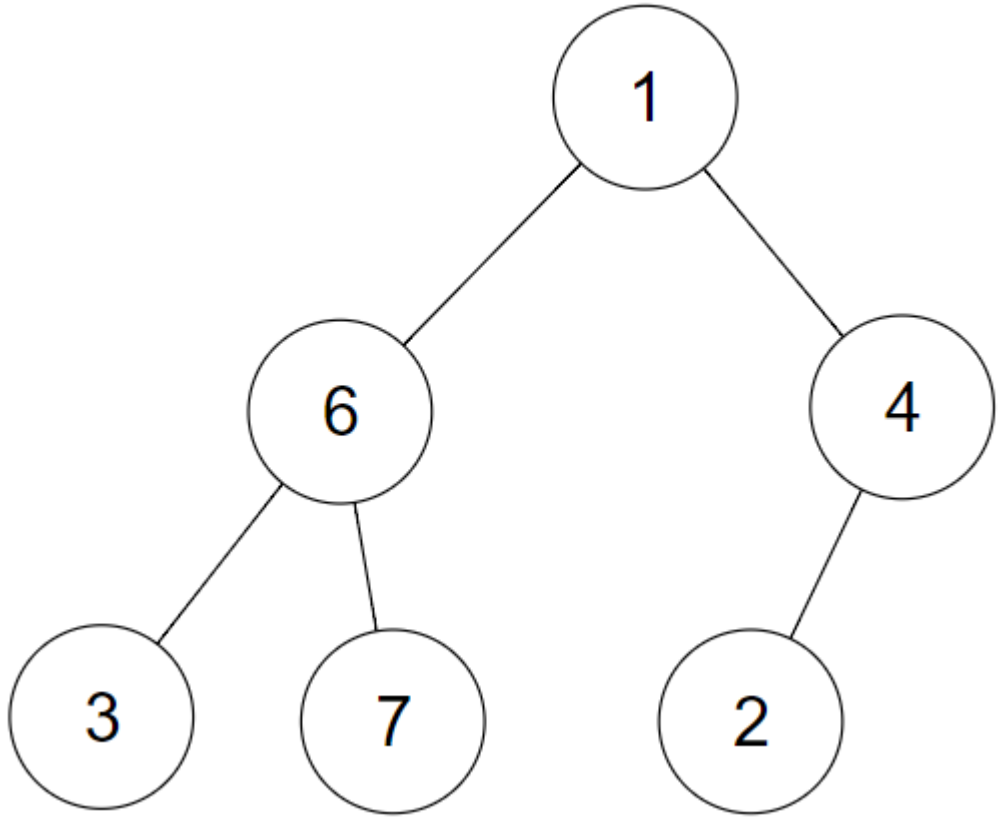


0	1	2	3	4	5	6	7
-	1	6	2	3	7	4	*

Reheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using reheaping.

- Starting at index 3, if the element there is less than its child, swap it.
- The subtree that begins at position 3 is now a valid subheap.

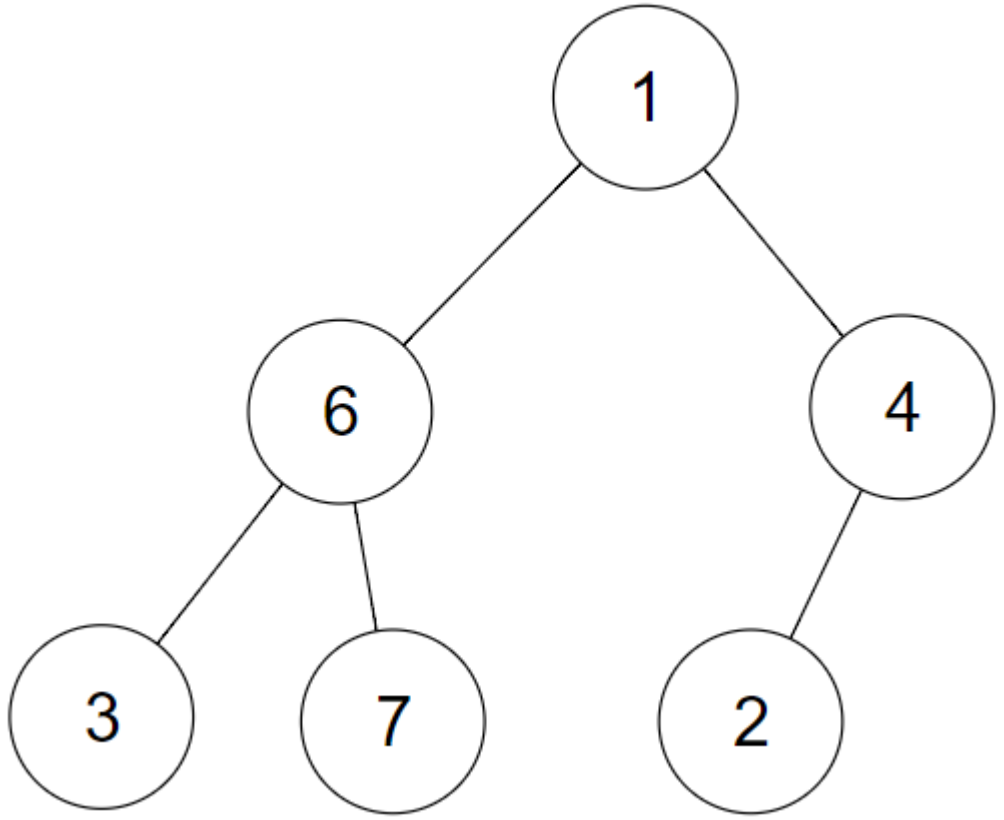


0	1	2	3	4	5	6	7
-	1	6	4	3	7	2	*

Reheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using reheaping.

- We keep looking backwards-backwards based on the *array* positions.

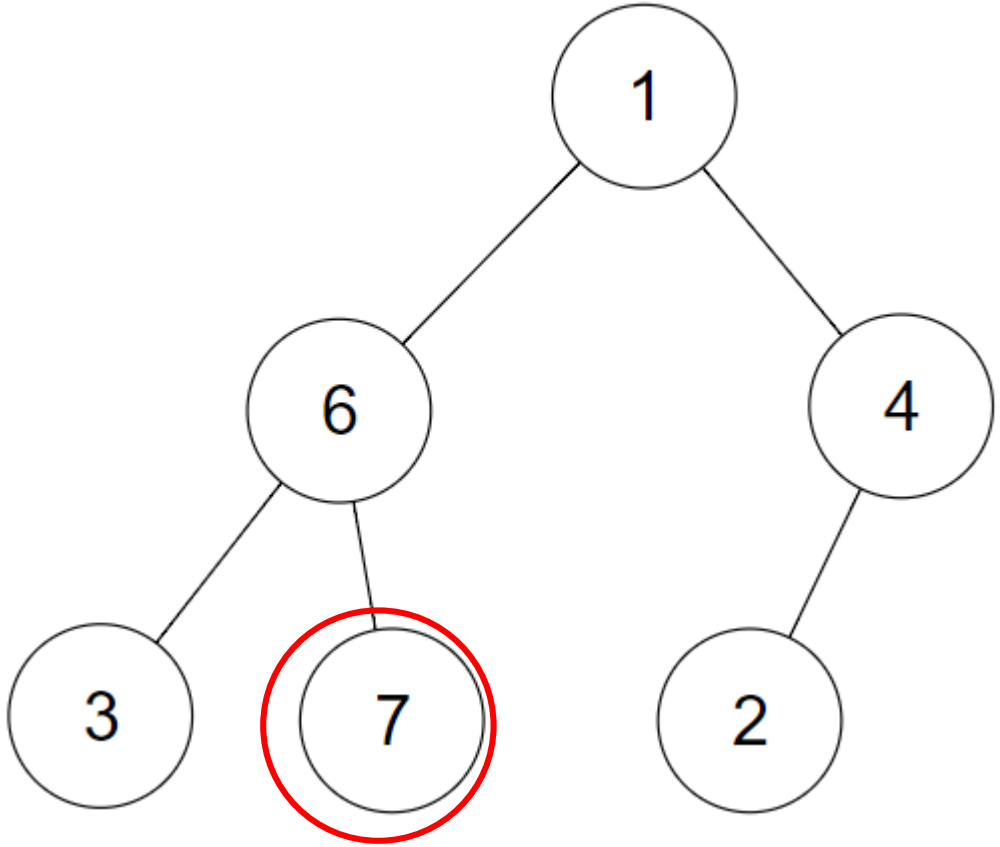


0	1	2	3	4	5	6	7
-	1	6	4	3	7	2	*

Reheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using reheaping.

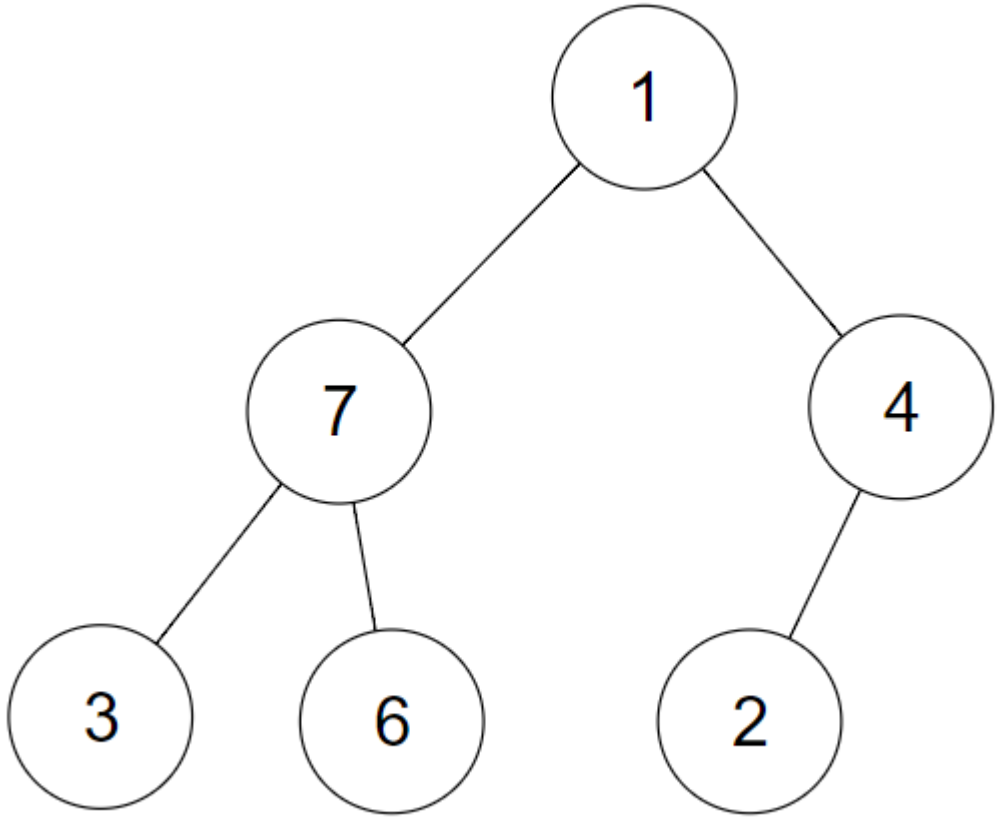
- Look at index 2: value 6.
- 6 has two children. Put the max of these three at the root of this subtree.



0	1	2	3	4	5	6	7
-	1	6	4	3	7	2	*

Reheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using reheaping.



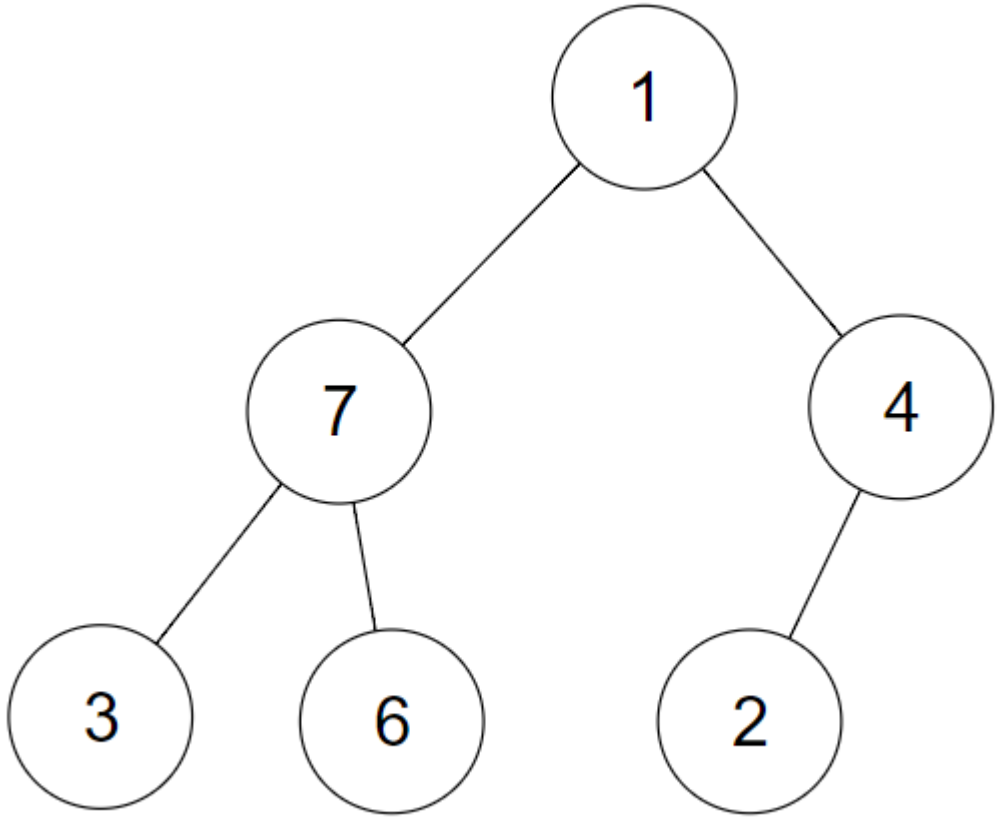
- Swap 6 and 7.
- The subtree starting at location 2 is now a valid subheap.
- Continue looking backward.

0	1	2	3	4	5	6	7
-	1	7	4	3	6	2	*

Reheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using reheaping.

- Look at index 1.
- 1 has two children. Put the max of those three notes in the root of this subtree.

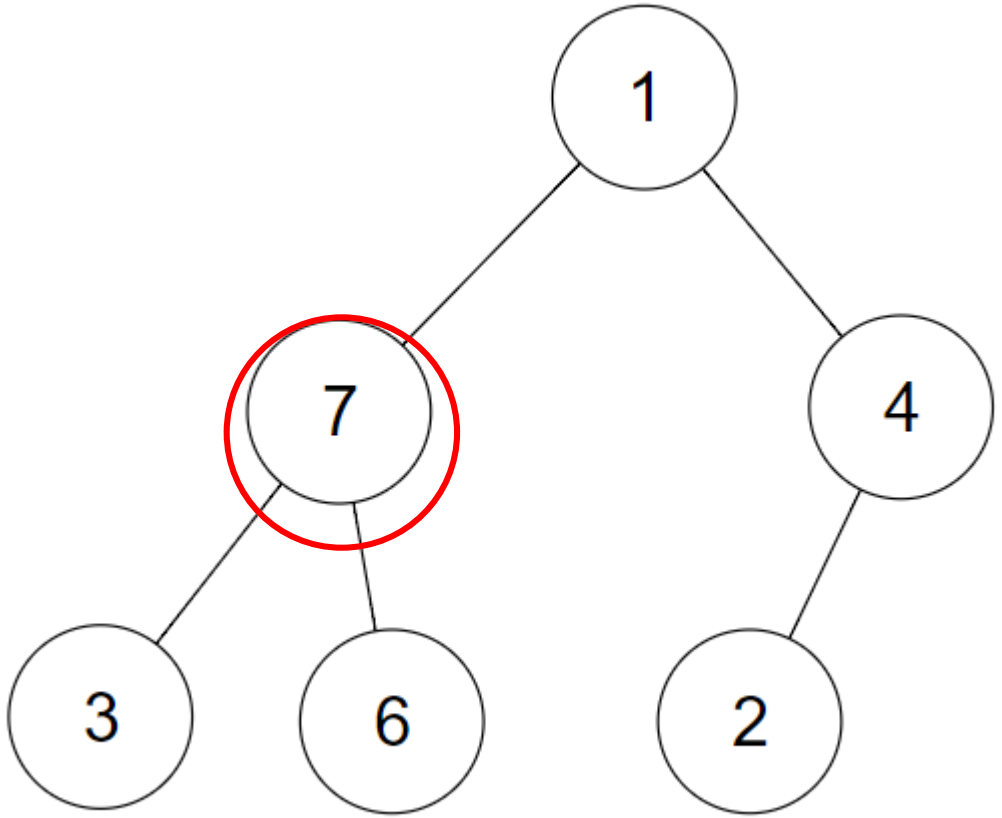


0	1	2	3	4	5	6	7
-	1	7	4	3	6	2	*

Reheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using reheaping.

- Look at index 1.
- 1 has two children. Put the max of those three notes in the root of this subtree.

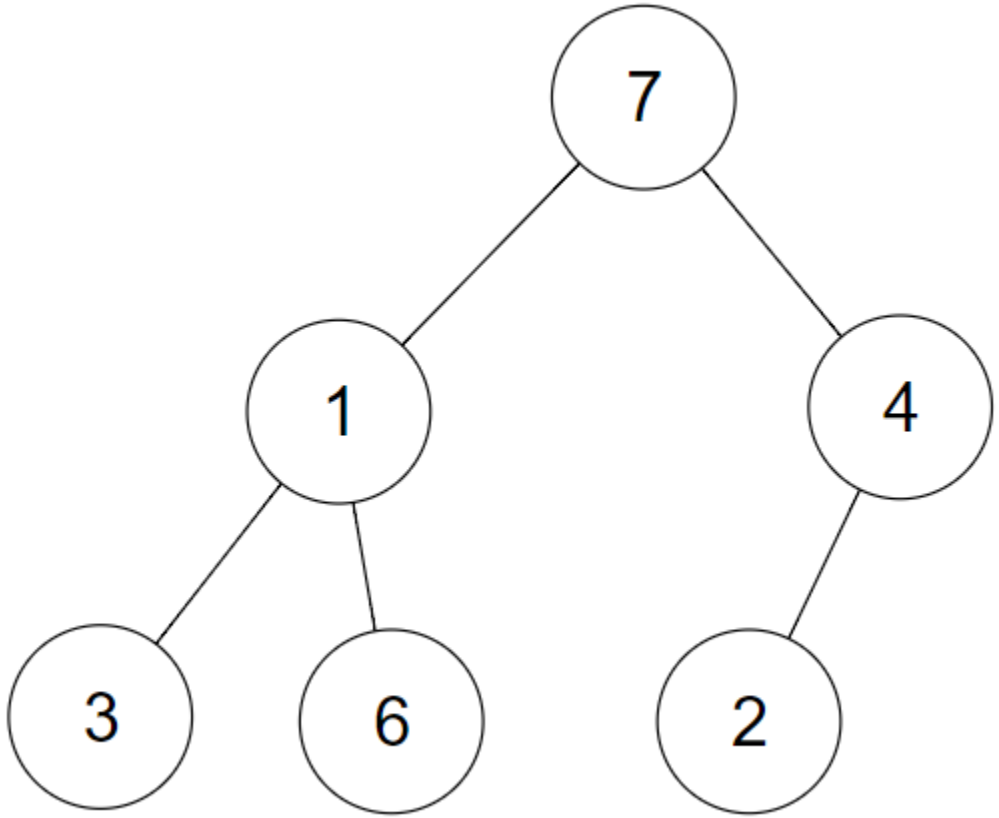


0	1	2	3	4	5	6	7
-	1	7	4	3	6	2	*

Reheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using reheaping.

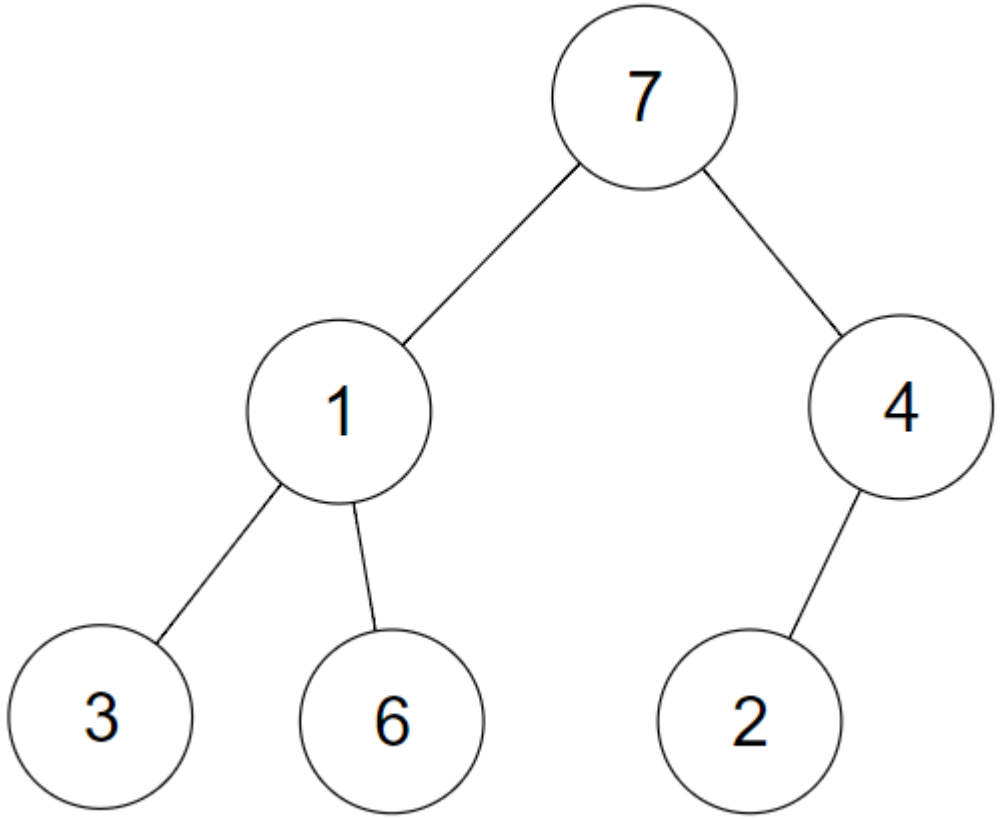
- Swap 1 and 7.



0	1	2	3	4	5	6	7
-	7	1	4	3	6	2	*

Reheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using reheaping.



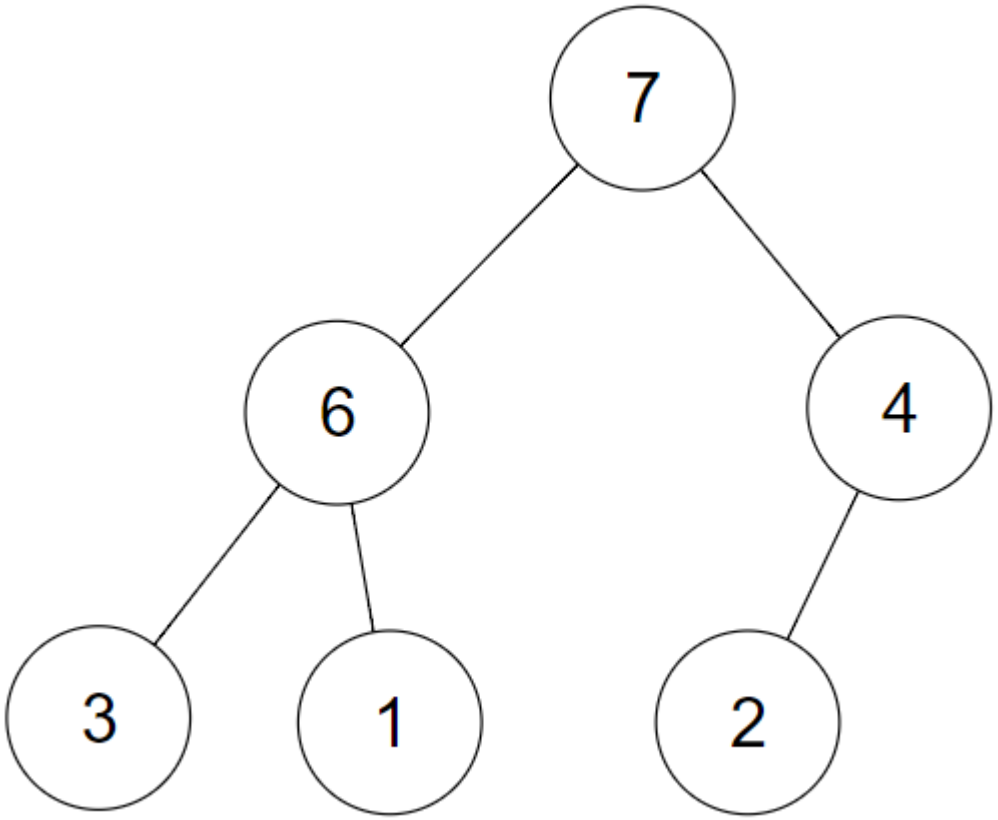
- We've made a swap higher up in the tree, so we need to propagate the change down the tree.
- Reexamine the subtree with root of 1 and put the max at the root of this subtree.

0	1	2	3	4	5	6	7
-	7	1	4	3	6	2	*

Reheaping

- Example: Insert 1, 6, 2, 3, 7, 4 into a maxheap using reheaping.

- In the worst case, this could go all the way back down to the lowest level.



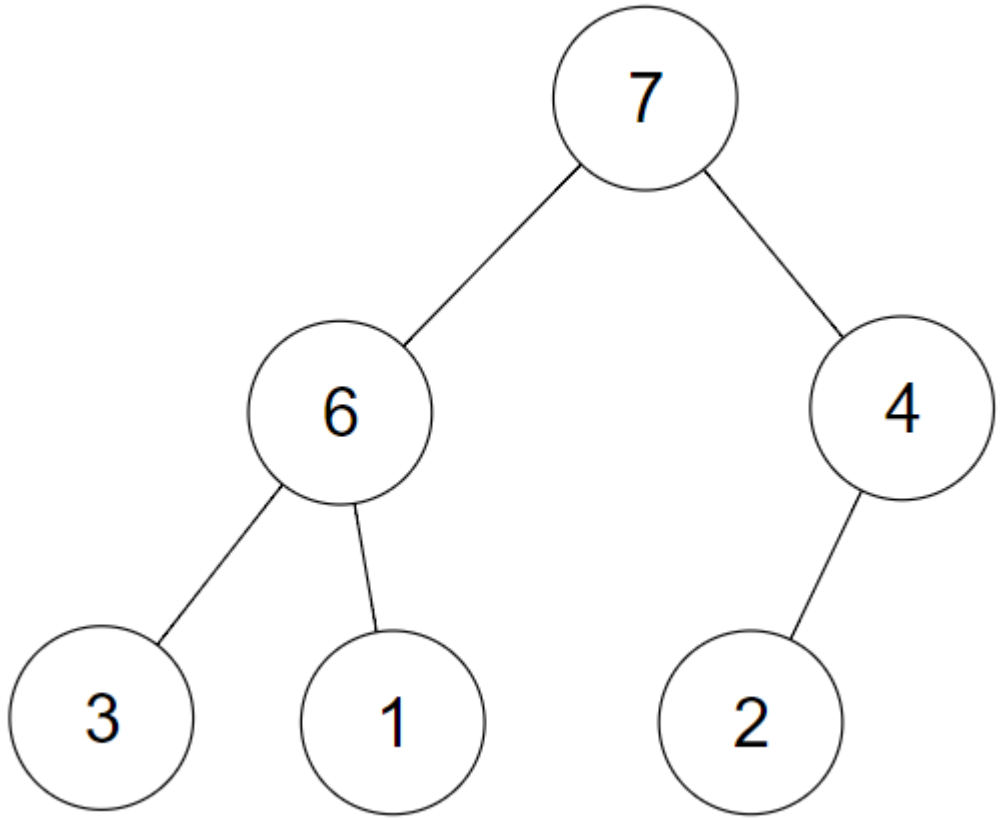
0	1	2	3	4	5	6	7
-	7	6	4	3	1	2	*

Removing from a Heap

- We typically use heaps for problems where we need the largest (or smallest) value. So when we retrieve a value, it's at the root.
- To remove the root:
  - Swap the max (root) with the right-most leaf on the lowest level.
    - This maintains the complete structure.
  - Swap the new max (the new root) down to the correct location.
    - Reheap it!
    - Find the max of it and its children and swap.
    - Repeat as needed.

Example: Removing the Root

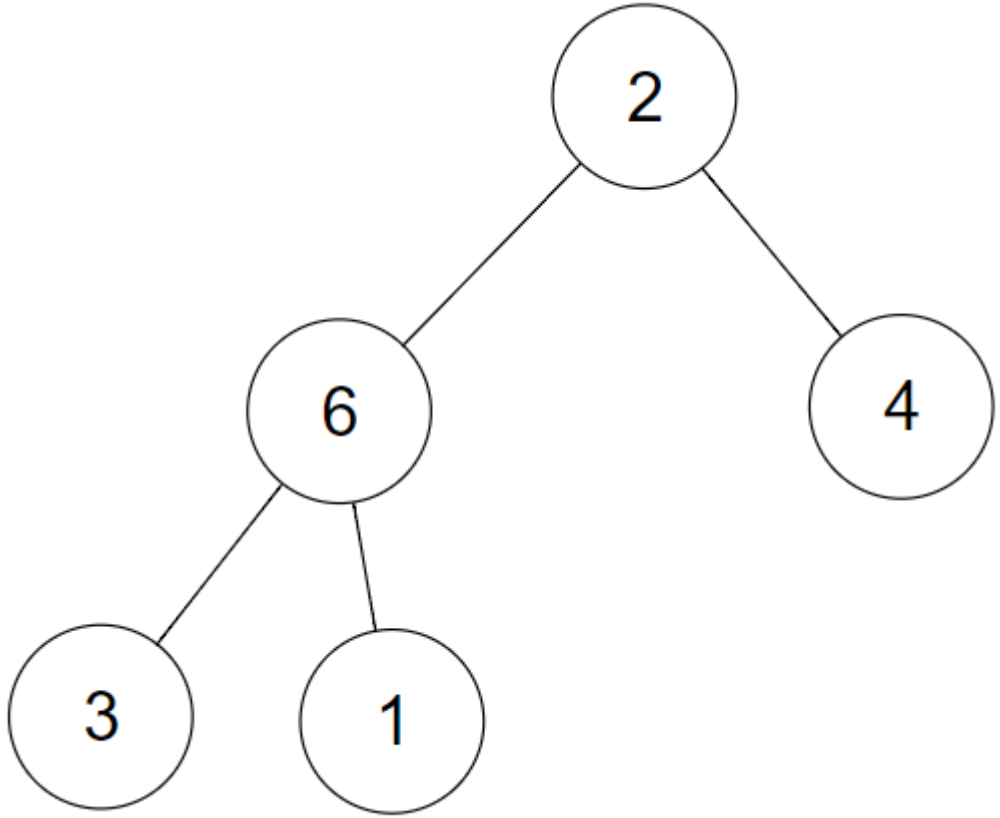
- The root is 7 (the max in the tree).



Example: Removing the Root

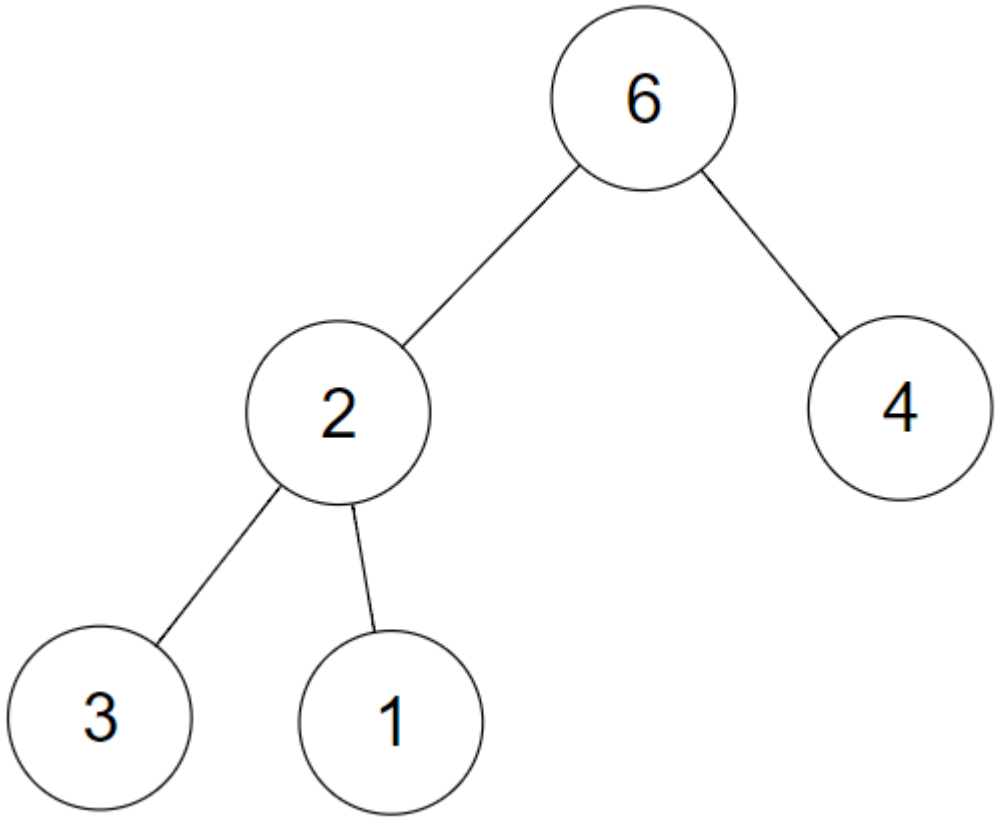
- The root is 7 (the max in the tree).

- Begin by swapping the right-most leaf on the lowest level with the root.
- The tree is still complete, which is good!
- But it no longer has the max structure.



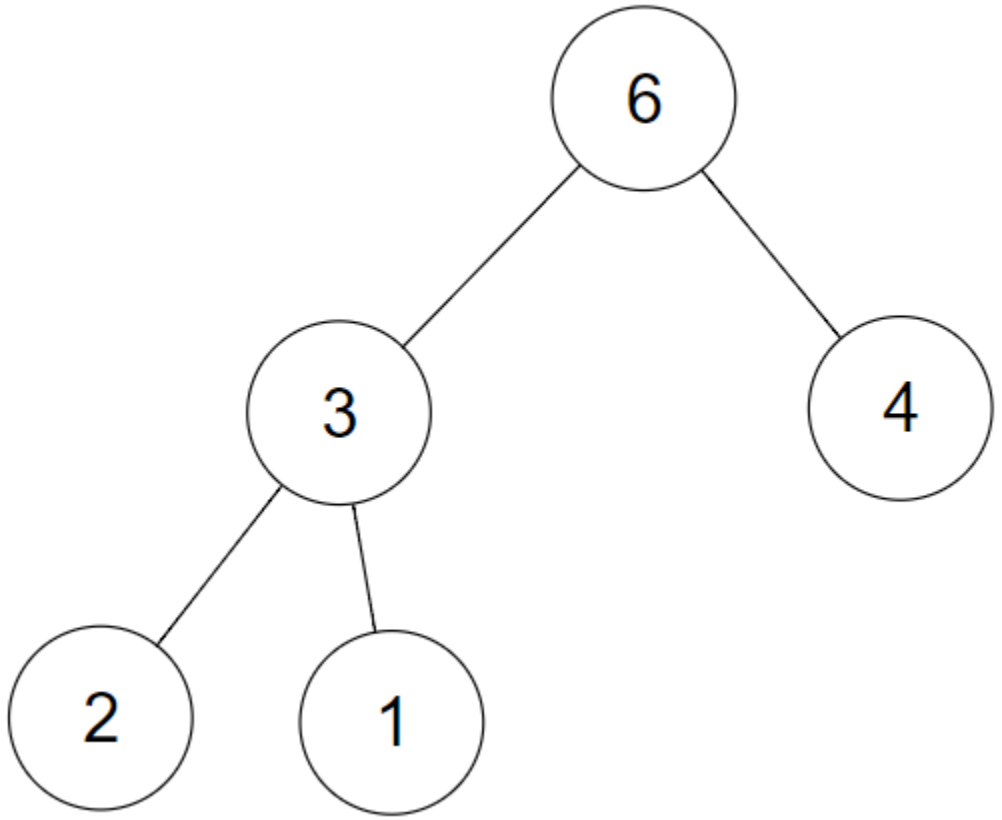
Example: Removing the Root

- Put the max of the 2-4-6 in the root.

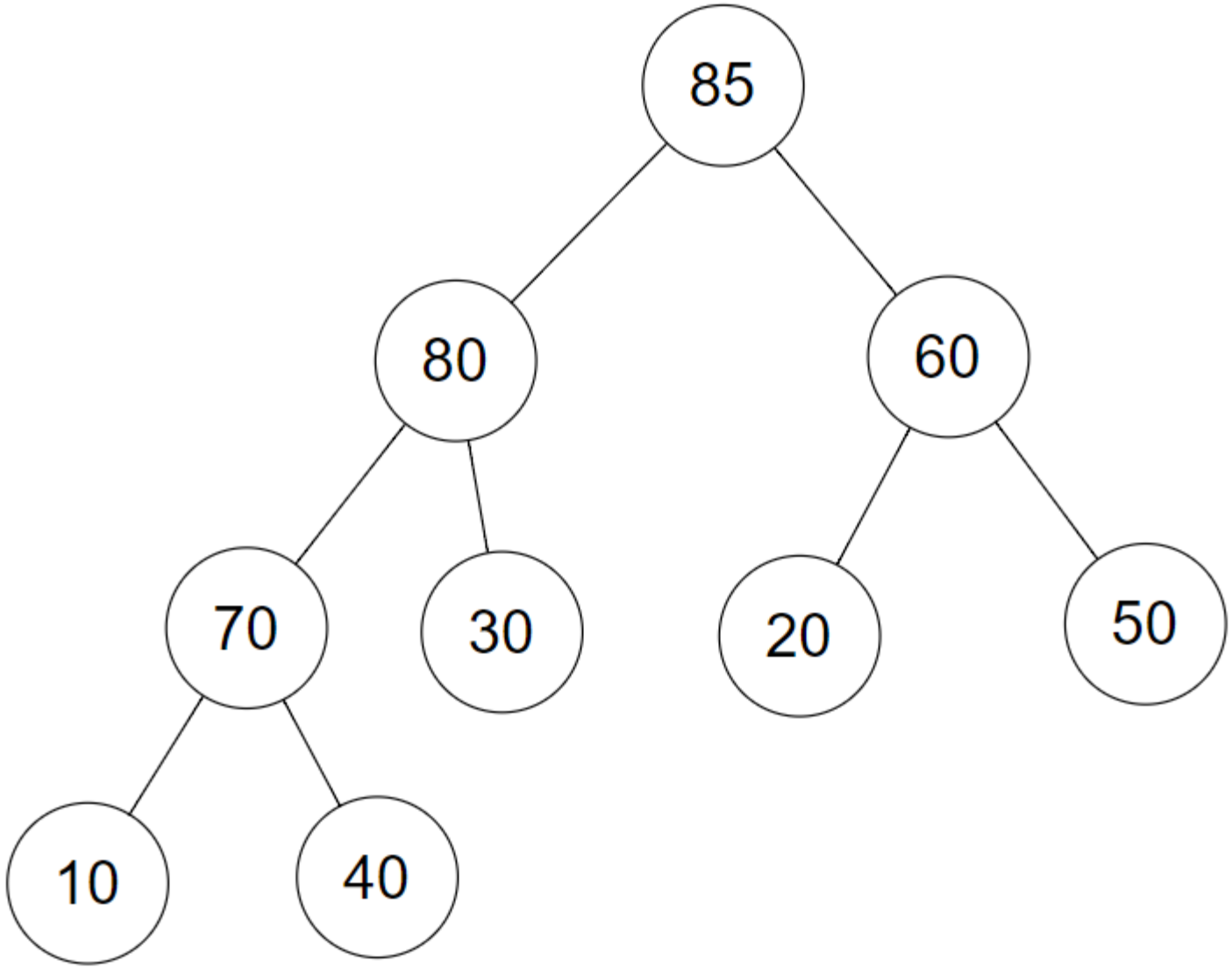


Example: Removing the Root

- Put the max of the 2-3-1 in the root of that subtree.

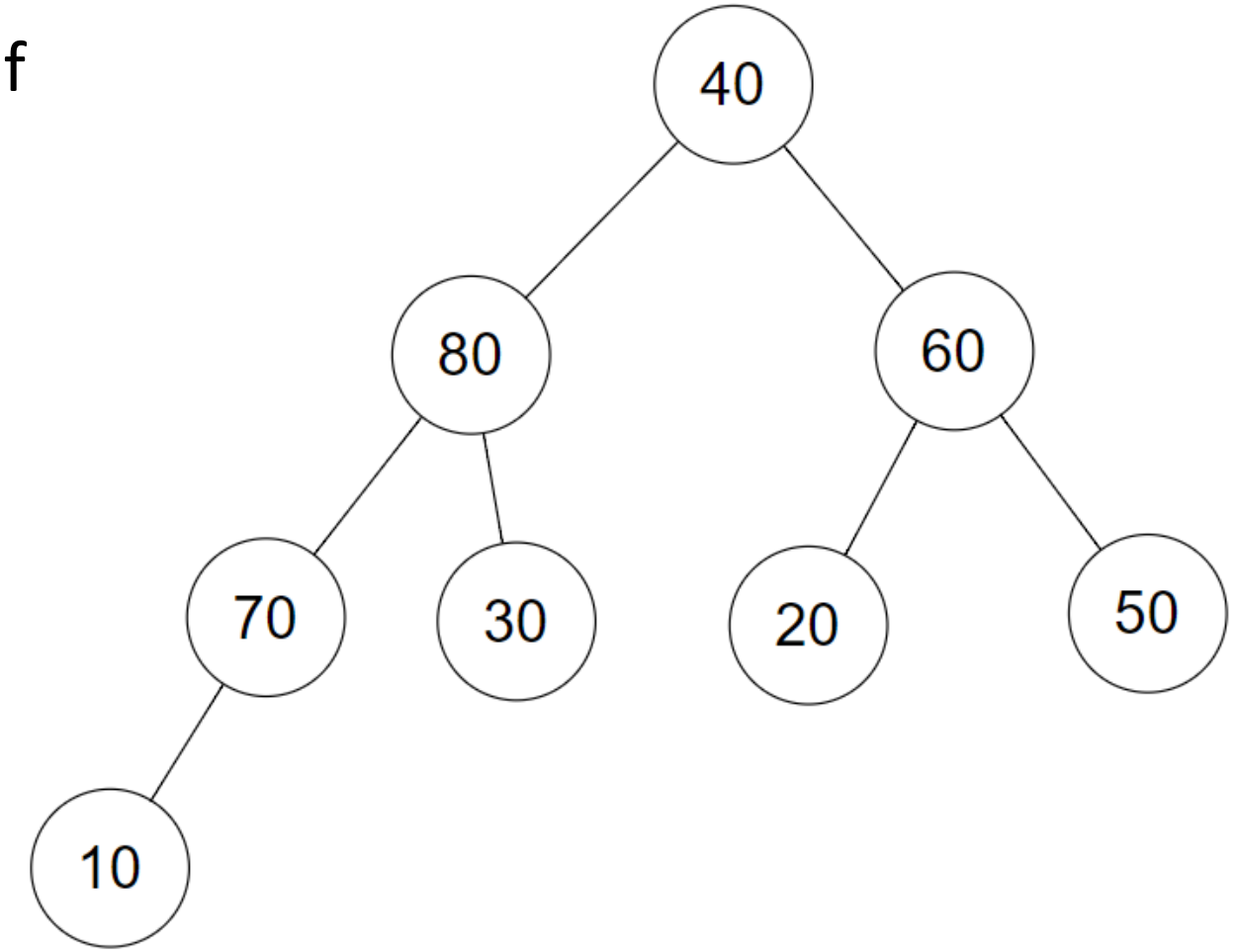


Example: Removing the Root



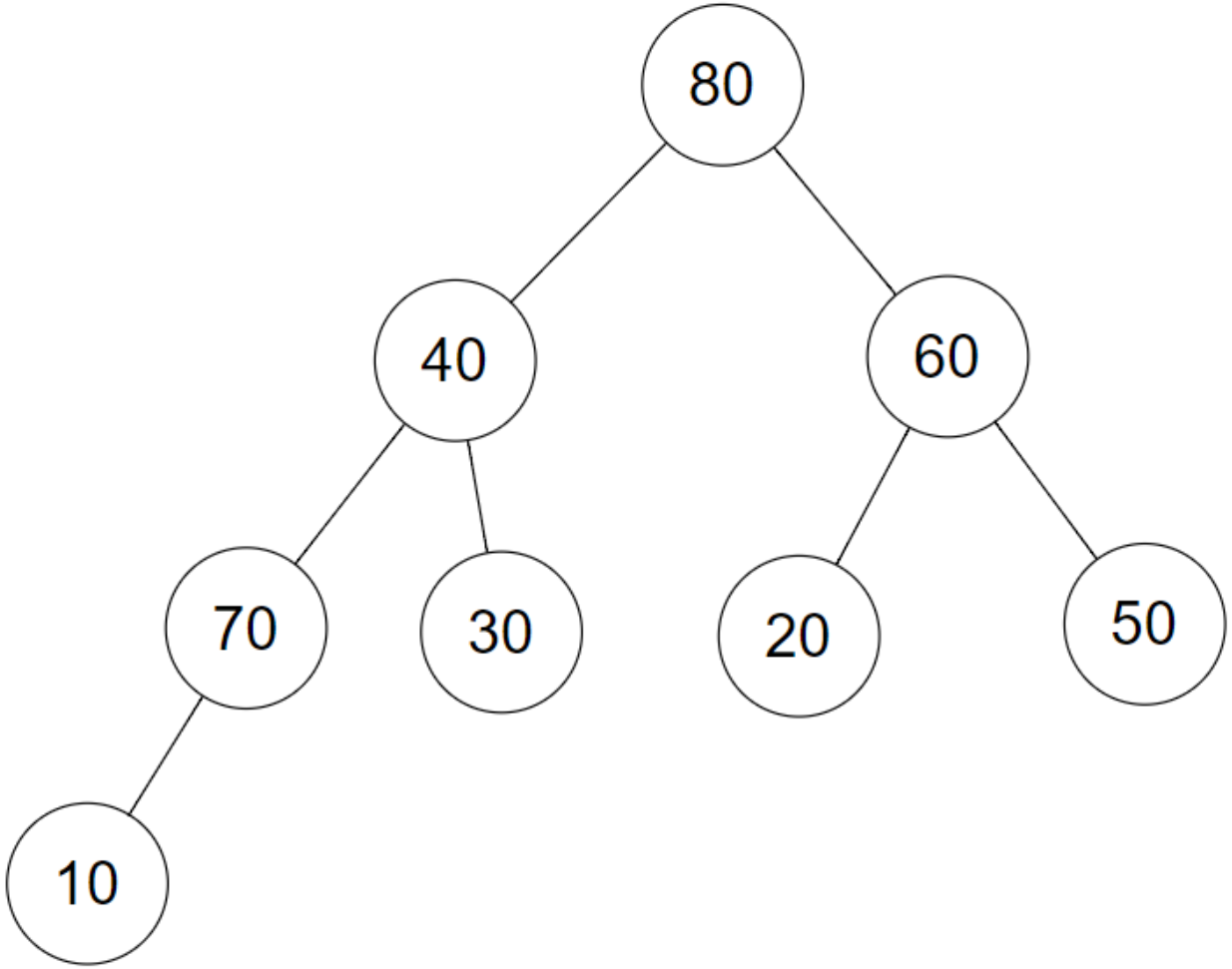
Example: Removing the Root

- Swap the right-most leaf in the lowest level with root.



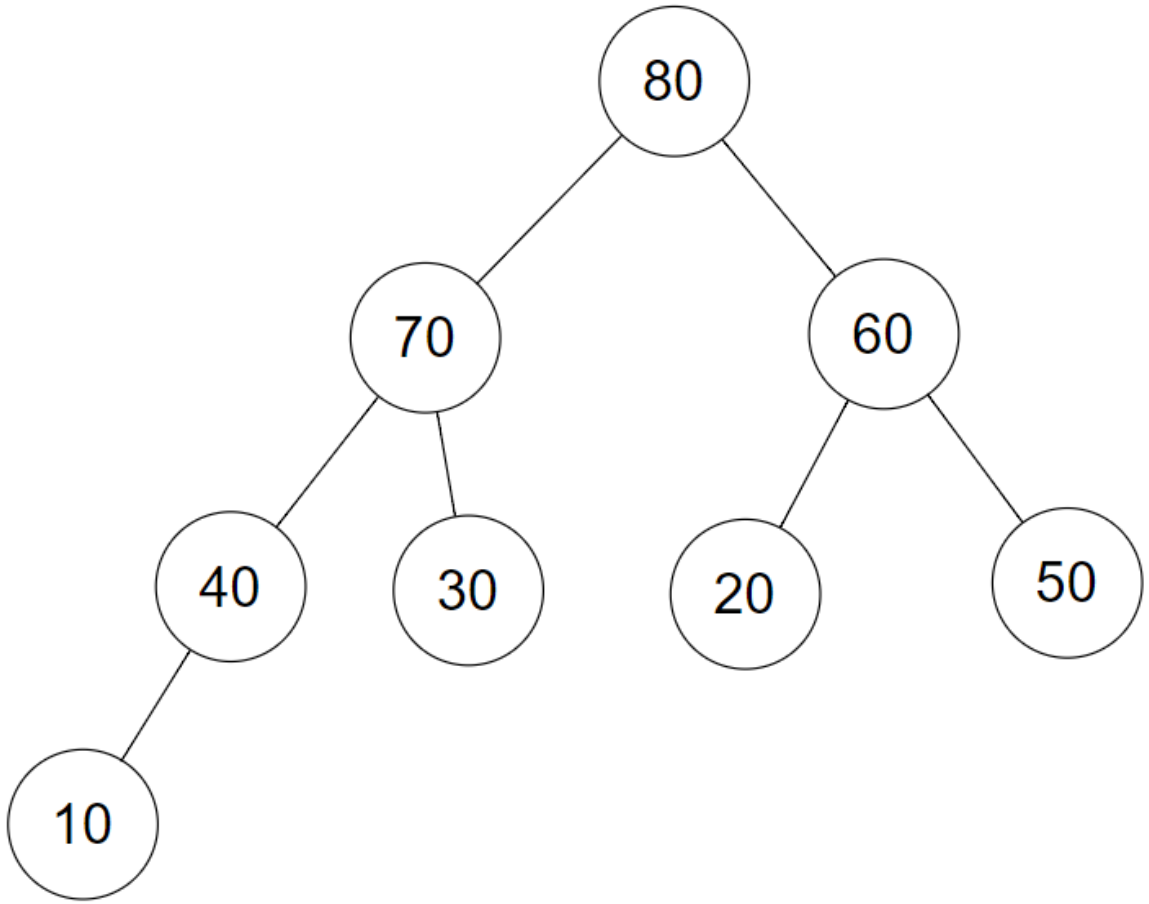
Example: Removing the Root

- Re-heap the 40-80-60



Example: Removing the Root

- Re-heap the 40-70-30



Using Heaps

- Heaps are often used to implement a priority queue.
  - Heaps keep elements in an order such that you can always access the element with the highest priority.
  - Inserting and removing elements are O(log n).
- You can use a heap to sort values- *heapsort*.
  - Creating a heap and removing values one at a time (using the delete-the-root algorithm) gives you the items in descending order.
  - O(n log n)
- Graph algorithms
- Problems such as finding the kth smallest or largest element in an array

BALANCED TREES

Binary Search Tree

- Binary Tree: a tree such that each node has at most two children (left and right)
- Binary Search Tree: an **ordered** binary tree, such that:
  - **all data** in the left **subtree** of a node is less than the node
  - **all data** in the right **subtree** of a node is greater than the node

Binary Search Tree

- The shape of a BST affects the efficiency of searching the tree.
  - Best case: O(log n)
  - Worst case: O(n)
- The best case occurs when a BST is *balanced*.
- There are two strategies to ensure balance.
  - Adjust a tree that is out of balance.
    - AVL trees are used for this.
  - Always keep a tree in balance.
    - B-Trees are used for this.

B-Trees

- B-Trees have far-ranging applications.

- Balance factor: the balance factor of any node is
  height(left subtree) – height(right subtree)
- Height-balanced: a BST is considered height-balanced if the balance factor is 0, 1, or -1
  - height-balanced is also sometimes called AVL

- A B-Tree is a general concept.
  - Let’s first look at 2-3 Trees and 2-4 Trees.

Nodes

- 2-nodes
  - Traditional nodes store a single object with links to two other nodes (left and right).
  - The nodes have 1 data element and 2 links.
- 3-nodes
  - These nodes have 2 data elements and 3 links.
- 4-nodes
  - These nodes have 3 data elements and 4 links

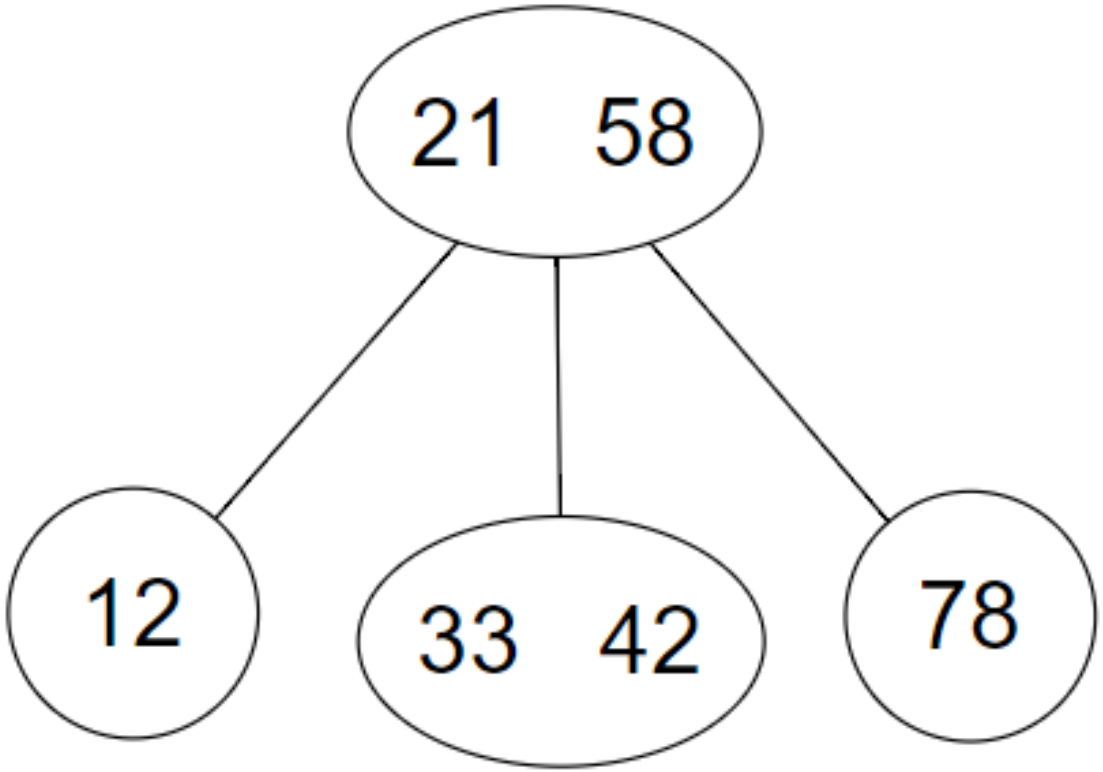
- The number refers to the number of **child nodes** (or links)- **not** the number of data elements contained in a node.

Nodes in a B-Tree

- 2-nodes in a binary search tree or a B-Tree
  - all data in the left subtree is less than the node
  - all data in the right subtree is greater than the node
- 3-nodes in a binary search tree or a B-Tree
  - all data in the left subtree is less than the smaller data element
  - all data in the right subtree is greater than the larger data element
  - all data in the middle subtree is between the smaller and larger data elements

Searching a B-Tree

- Searching a B-Tree is straightforward.
- Use the same ideas that you would use to search a normal BST.
  - If value is smaller than left, go down left subtree.
  - If value is between, go down middle subtree.
  - If value is greater than right, go down right subtree.



B-Trees: 2-3 Trees

- B-Trees maintain their balance during the process of adding the tree.
- A 2-3 tree is a search tree that uses 2-nodes and 3-nodes
- Interior nodes must each have two or three children
  - They are either 2-nodes or 3-nodes
- All leaves occur on the lowest levels
  - This is what makes them balanced!

Adding to a 2-3 Tree

- Search for the **leaf** where the new element would go and add there, just like you would for a regular BST.
  - Note that you **never add a new leaf!** You add to the existing leaf where the value belongs!
- If the leaf is a 2-node
  - Add the element to that 2-node, thus converting it to a 3-node
- If the leaf is a 3-node
  - *Pretend* to add it in this proper place, which would create a 4-node.
  - But we aren't allowed to have 4-nodes in a 2-3 tree.
  - So we need to *split* this *pretend/illegal* node.

Adding to a 2-3 Tree: Splitting a Node

- Split the left and right values into individual 2-nodes.
- Bump the middle value up to the parent node.
  - If the parent node was a 2-node, convert it to a 3-node and you're done.
  - If the parent was already a 3-node, then you have again created an illegal 4-node, so you need to repeat the split.

Building a 2-3 Tree

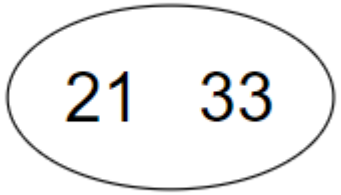
- Example: Insert 21, 33, 12, 58, 78, 42, 63, 15, 18 into a 2-3 tree.

- Add 21.

Building a 2-3 Tree

- Example: Insert 21, 33, 12, 58, 78, 42, 63, 15, 18 into a 2-3 tree.

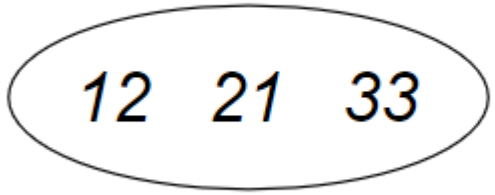
- Add 33.
  - Find the **existing** leaf.
  - This is the 21 node.
  - This is a 2-node, so we convert it to a 3-node.



Building a 2-3 Tree

- Example: Insert 21, 33, 12, 58, 78, 42, 63, 15, 18 into a 2-3 tree.

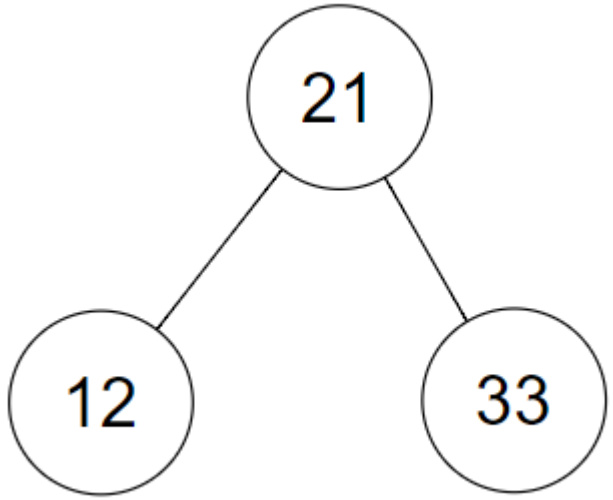
- Add 12.
  - Find the **existing** leaf.
  - This is the 21-33 node.
  - This is a 3-node, so we pretend to add, creating an illegal 4-node.



Building a 2-3 Tree

- Example: Insert 21, 33, 12, 58, 78, 42, 63, 15, 18 into a 2-3 tree.

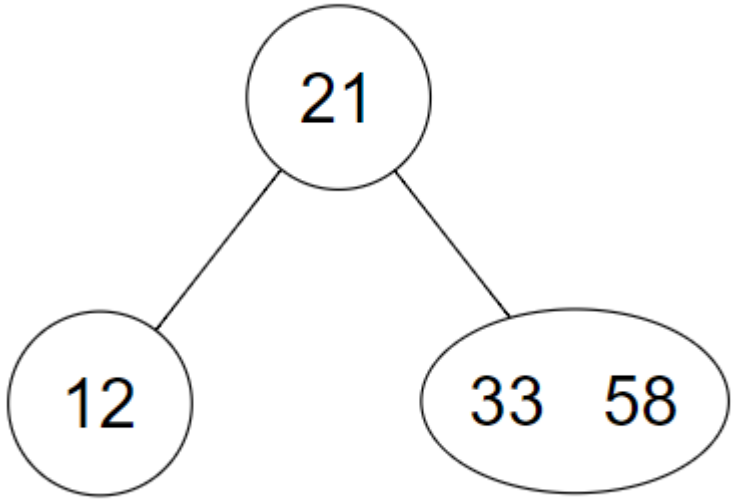
- Split!
  - Left and right values become 2-nodes.
  - Middle value becomes the parent.



Building a 2-3 Tree

- Example: Insert 21, 33, 12, 58, 78, 42, 63, 15, 18 into a 2-3 tree.

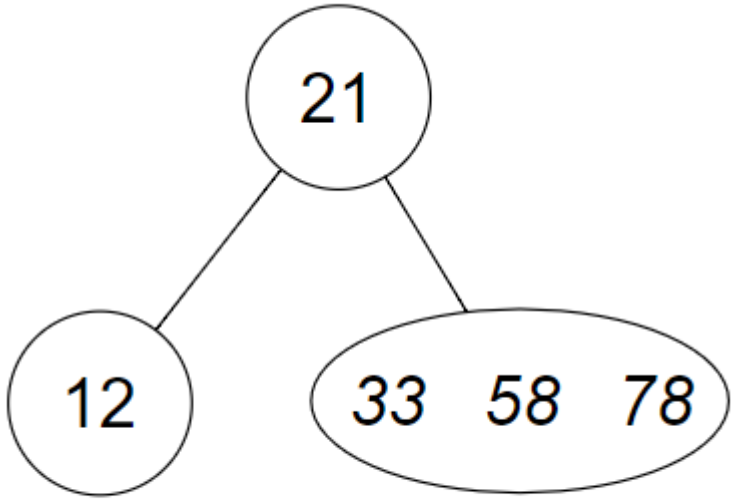
- Add 58.
  - Find the leaf where it belongs- 33.
  - This is a 2-node so insert it to make a 3-node.



Building a 2-3 Tree

- Example: Insert 21, 33, 12, 58, 78, 42, 63, 15, 18 into a 2-3 tree.

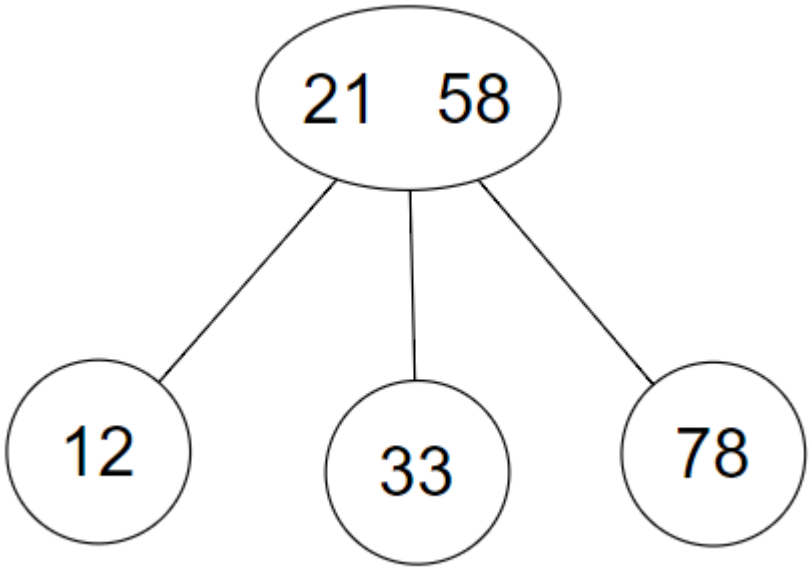
- Add 78.
  - Find the leaf where it belongs- the 33-58 node.
  - This is a 3-node so we *pretend* to create a 4-node.



Building a 2-3 Tree

- Example: Insert 21, 33, 12, 58, 78, 42, 63, 15, 18 into a 2-3 tree.

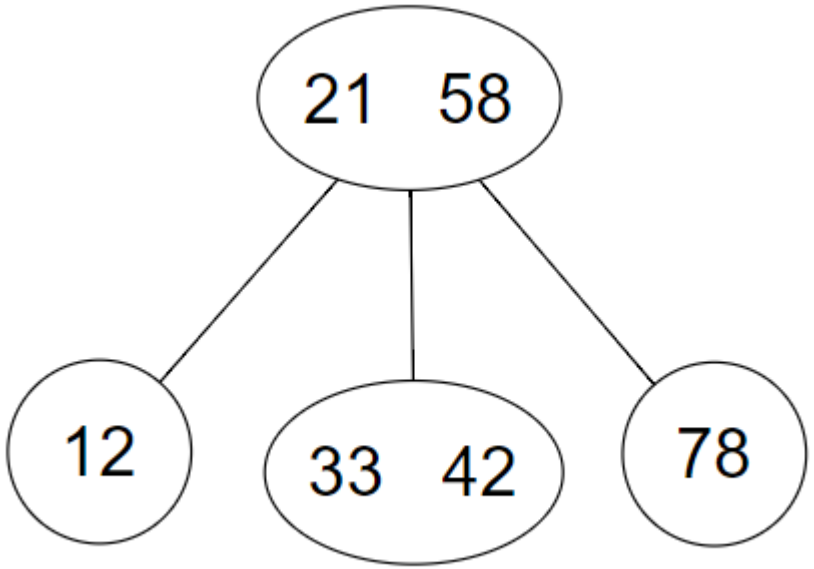
- Split!
  - Left and right values become 2-nodes.
  - Middle node gets bumped up.
  - The parent was a 2-node, so we convert it to a 3-node and are done.
- Confirm you still have a B-Tree!



Building a 2-3 Tree

- Example: Insert 21, 33, 12, 58, 78, 42, 63, 15, 18 into a 2-3 tree.

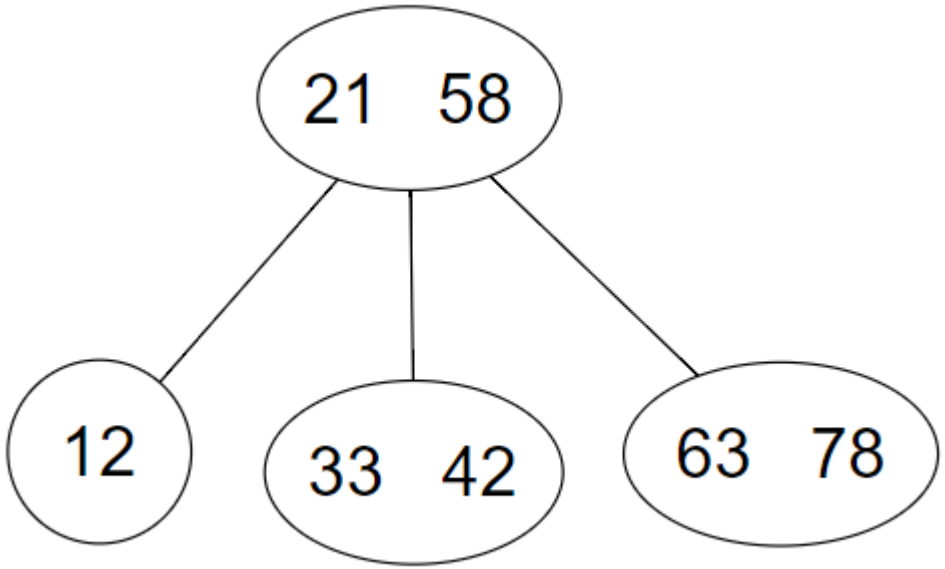
- Add 42.
  - Find the leaf- the 33 node.
  - This is a 2-node, so we add and create a 3-node.



Building a 2-3 Tree

- Example: Insert 21, 33, 12, 58, 78, 42, 63, 15, 18 into a 2-3 tree.

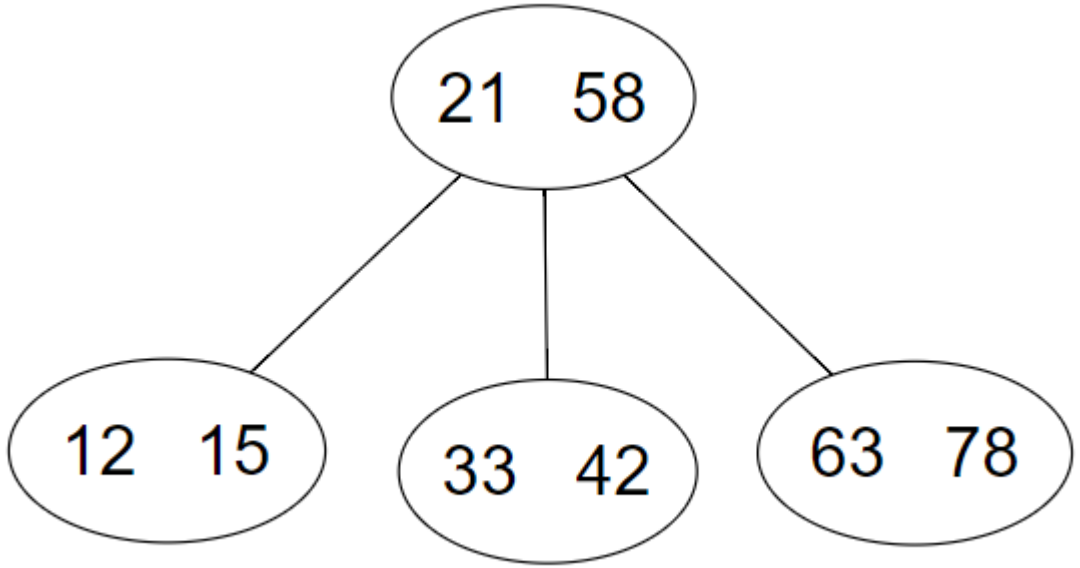
- Add 63.
  - Find the leaf- the 78.
  - This is a 2-node, so add to make a 3-node.



Building a 2-3 Tree

- Example: Insert 21, 33, 12, 58, 78, 42, 63, 15, 18 into a 2-3 tree.

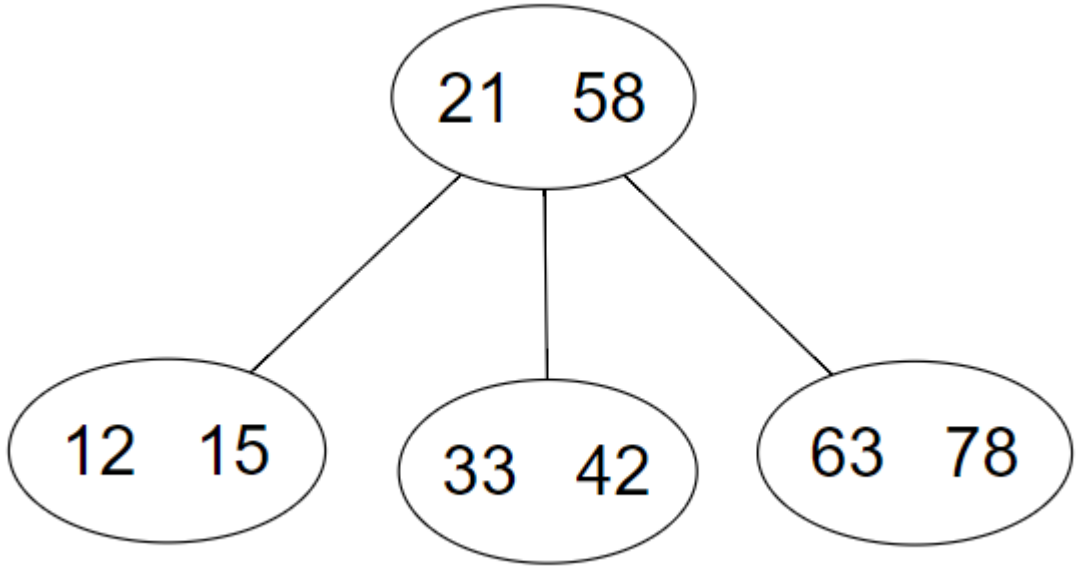
- Add 15.
  - Find the leaf- the 12 node.
  - This is a 2-node so we add and make it a 3-node.



Building a 2-3 Tree

- Example: Insert 21, 33, 12, 58, 78, 42, 63, 15, 18 into a 2-3 tree.

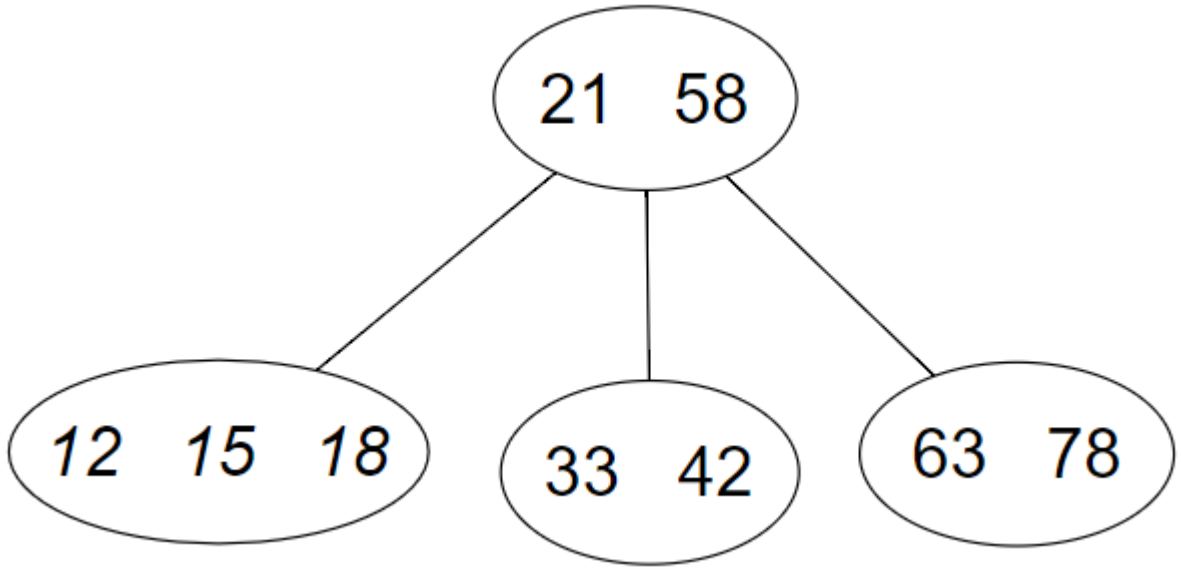
- This tree is currently *ripe for splitting.*
  - This means that no matter what value is added next, a split will occur because any value will create an illegal 4-node.



Building a 2-3 Tree

- Example: Insert 21, 33, 12, 58, 78, 42, 63, 15, 18 into a 2-3 tree.

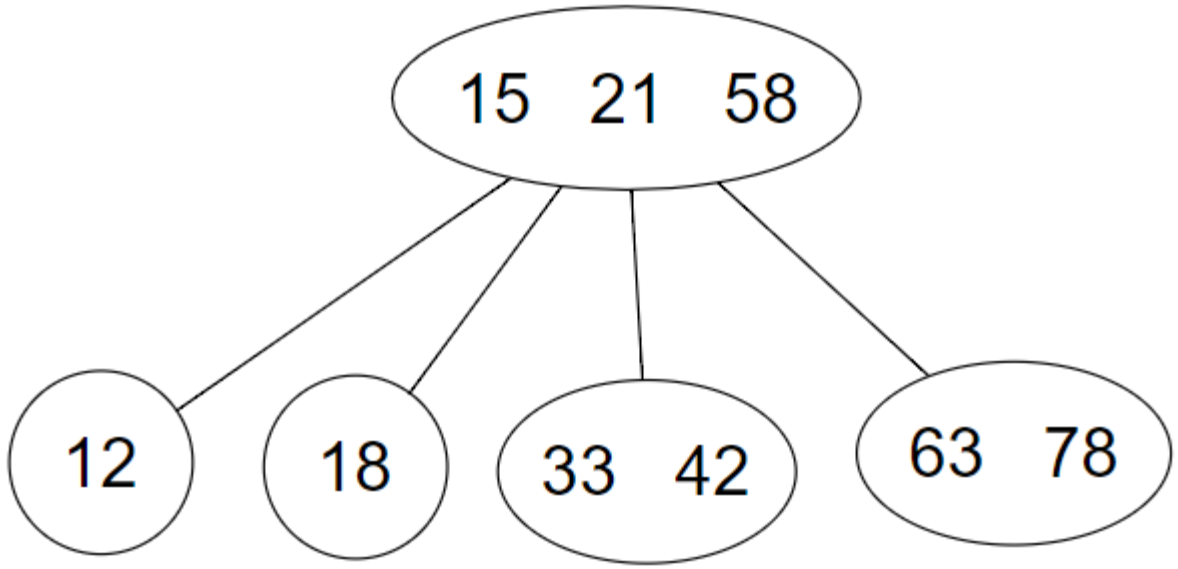
- Add 18.
  - Find the leaf- the 12-15 node.
  - Pretend to create a 4-node.



Building a 2-3 Tree

- Example: Insert 21, 33, 12, 58, 78, 42, 63, 15, 18 into a 2-3 tree.

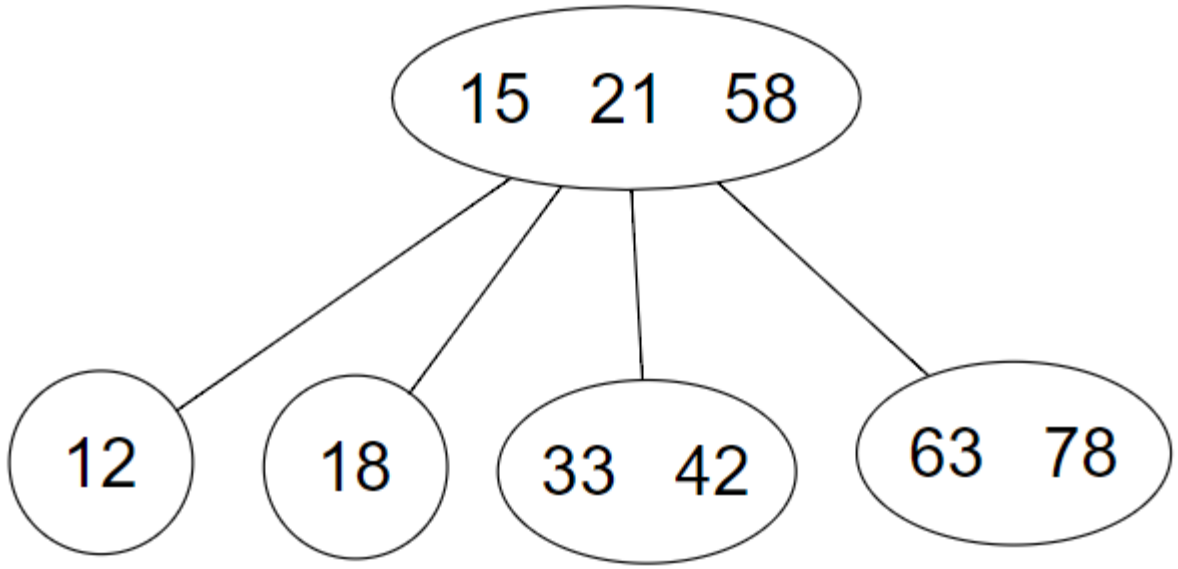
- Split!
  - Left and right become 2-nodes.
  - Parent gets bumped up.



Building a 2-3 Tree

- Example: Insert 21, 33, 12, 58, 78, 42, 63, 15, 18 into a 2-3 tree.

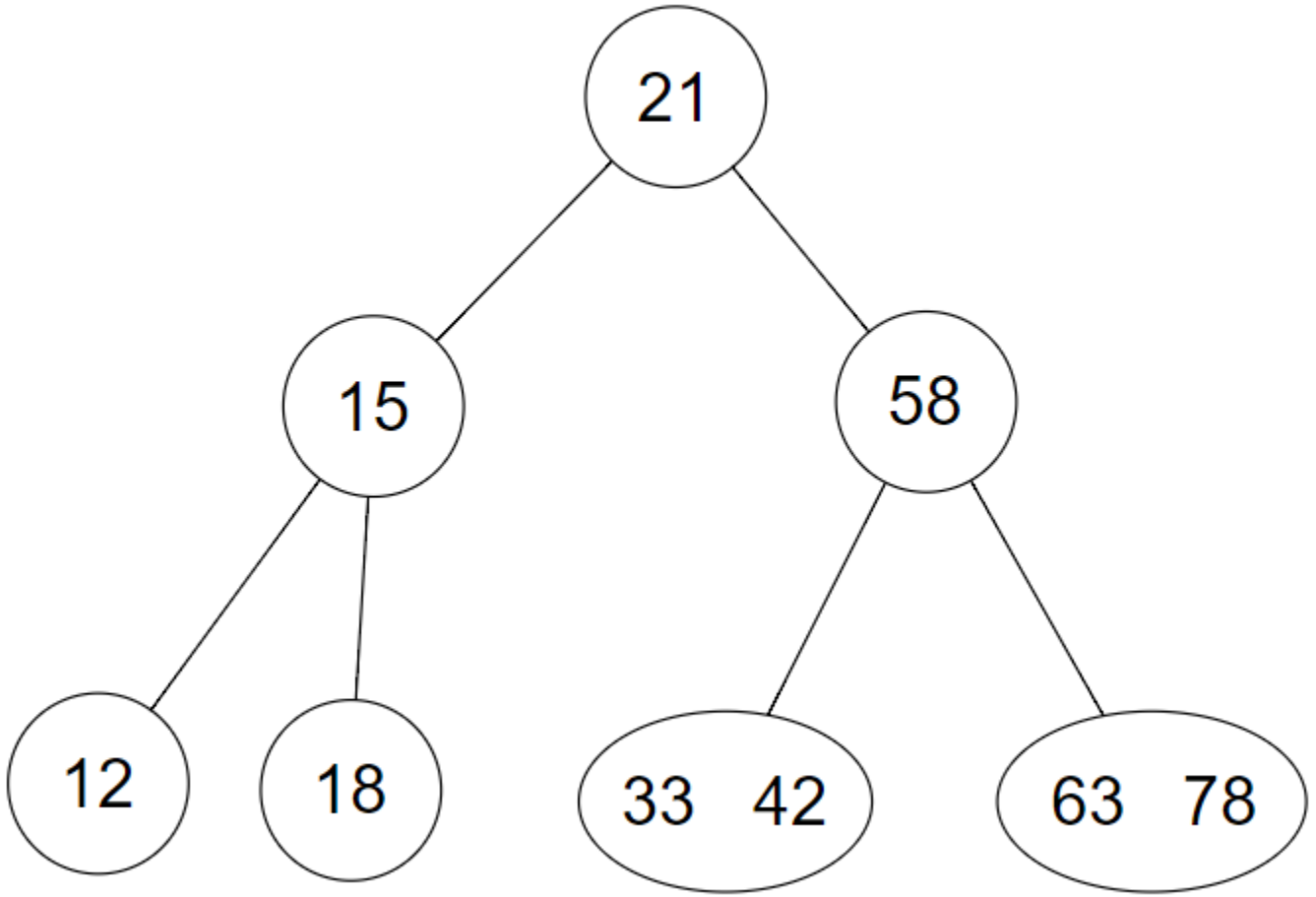
- We've created another 4-node.
- Split again!



Building a 2-3 Tree

- Example: Insert 21, 33, 12, 58, 78, 42, 63, 15, 18 into a 2-3 tree.

- Split!
  - Left and right become 2-nodes.
  - Middle gets bumped up a level.

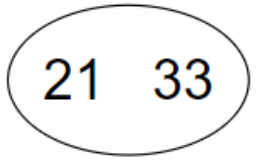


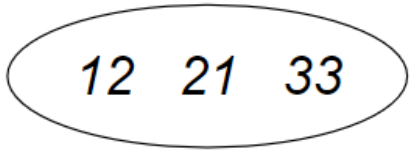
Avoid Common Errors

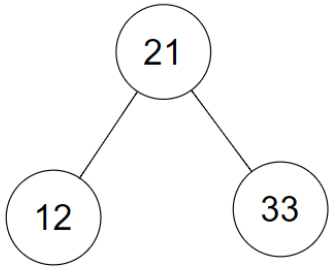
- Remember to bump up the parent when splitting!
- Don’t add elements to interior nodes.
  - When adding a new element, follow the path **all the way down to a leaf.**
  - Add at the leaf- either by adding to a 2-node to create a 3-node or by adding to a 3-node to create an illegal 4-node (and then splitting).
- Don’t add elements as new 2-node leaves.
  - 2-node leaves are only created through the splitting process.

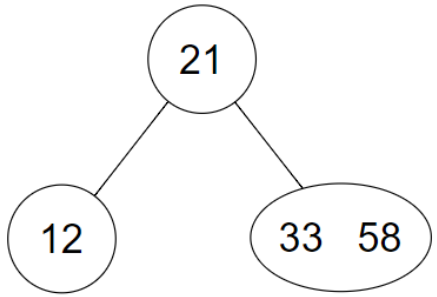
INCORRECT!!!

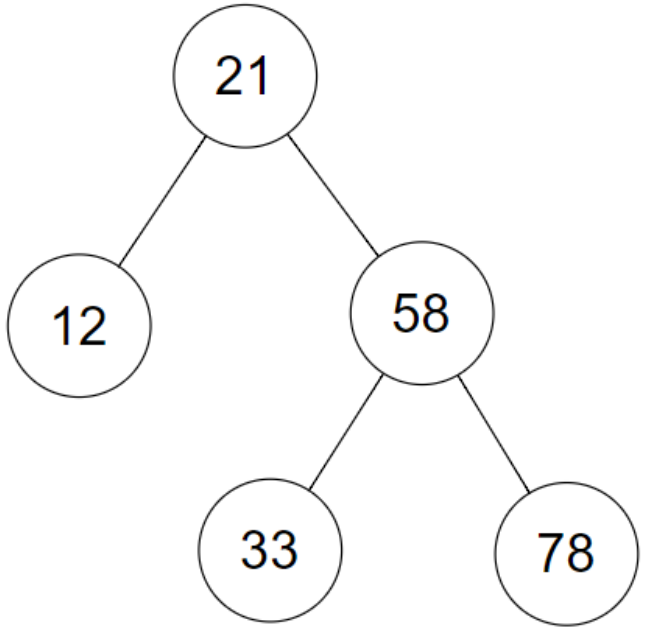
- 21, 33, 12, 58, 78, 42, 63, 15, 18





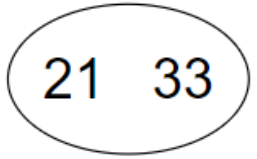


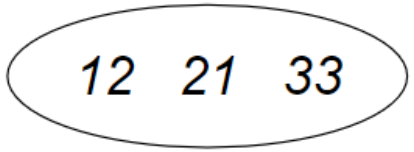


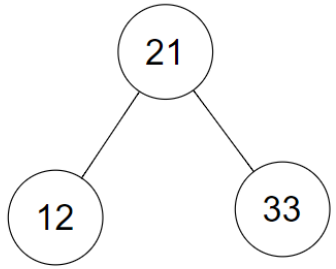


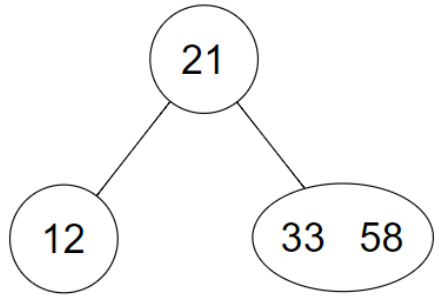
INCORRECT!!!

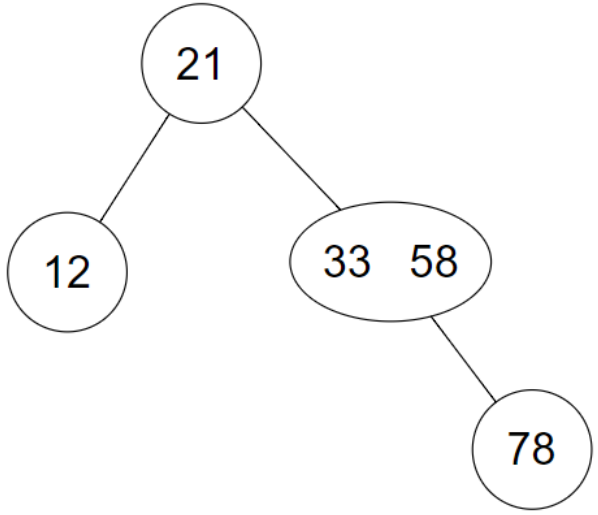
- 21, 33, 12, 58, 78, 42, 63, 15, 18





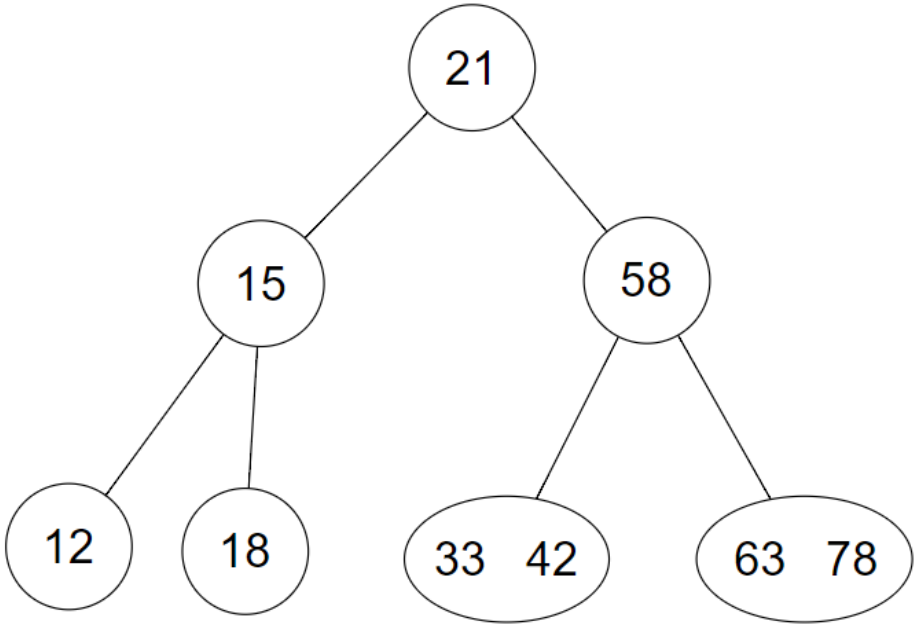


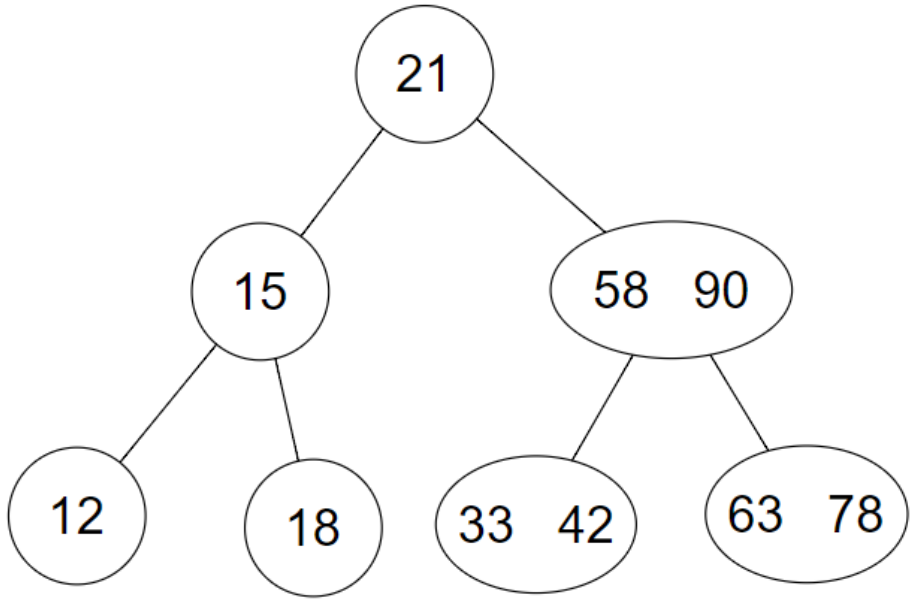




INCORRECT!!!

- 21, 33, 12, 58, 78, 42, 63, 15, 18, 90





B-Trees: 2-4 Trees

- A search tree that uses 2-nodes, 3-nodes, and 4-nodes.
- Also called a 2-3-4 tree.
- Interior nodes must each have two, three, or four children.
  - They are either 2-nodes, 3-nodes, or 4-nodes.
- All leaves occur on the lowest levels
  - This is what makes them balanced!

Adding to a 2-4 Tree

- Start the same way as for a 2-3 tree: search for the **leaf** where the new element would go and add there.
- During this search, if you find a 4-node, you split right away.
- We allow these trees to have 4-nodes **unless** we find them when searching during an addition.
  - At that time, we split them.
  - We then resume the search for where the new element belongs.
- This is a *preemptive* split to create space for the new element.

Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

- Add 70.

Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

- Add 80.
  - Find the leaf where it belongs- to the right of 70.

Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

- Add 90.
  - Find the leaf where it belongs- the 70-80 node.
  - This creates a 4-node.
  - We are allowed to have these!

Building a 2-4 Tree

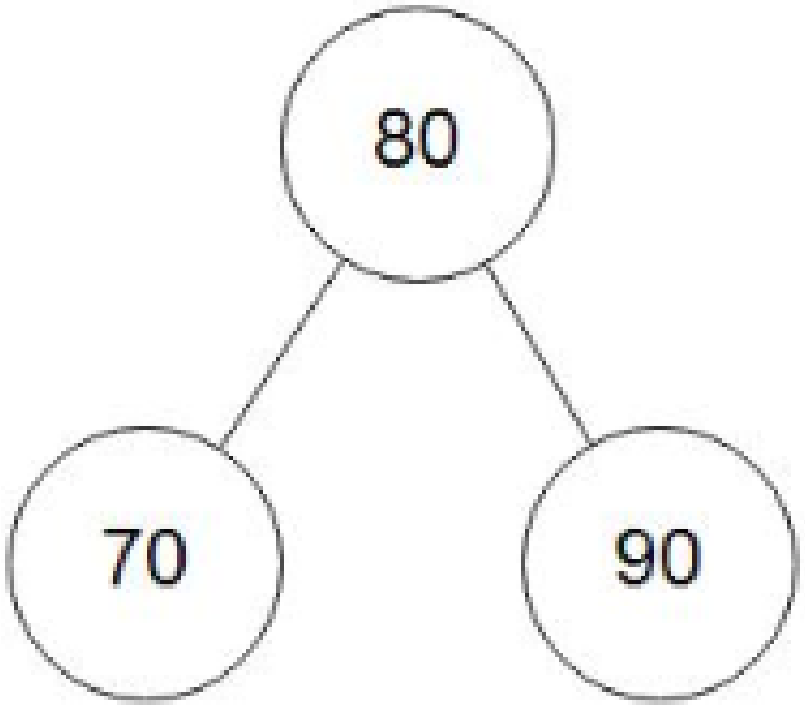
- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

- Add 20.
  - Find the leaf where it belongs…
  - As soon as we begin to search, we find a 4-node! (the root)
  - Split right away

Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

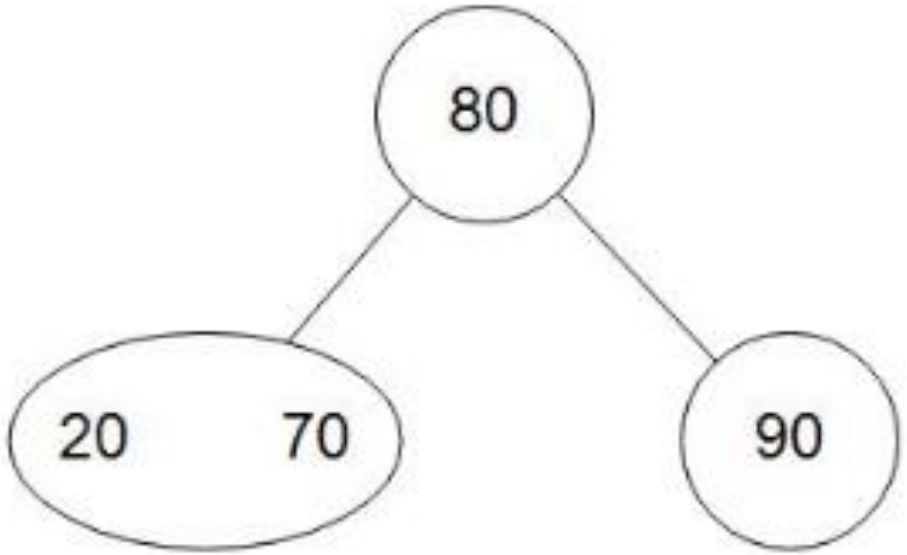
- Split!



Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

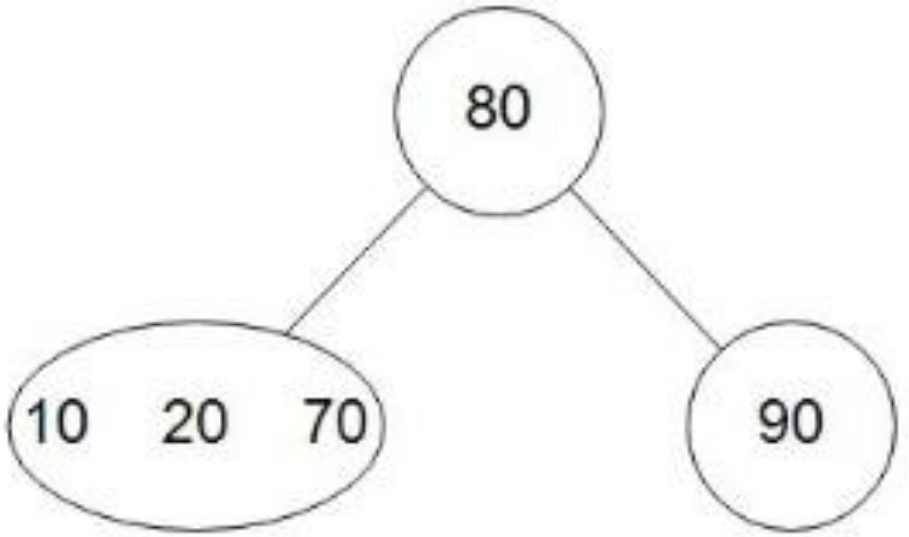
- Resume the search for the leaf where 20 belongs- it's the 70 node.



Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

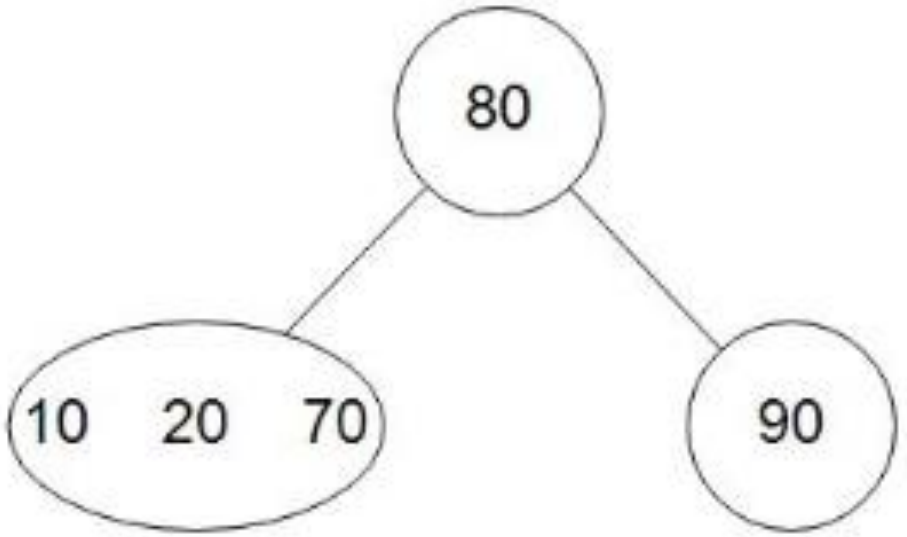
- Add 10.
  - Search for the leaf where it belongs.
  - This creates a 4-node, but that's allowed during the add.



Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

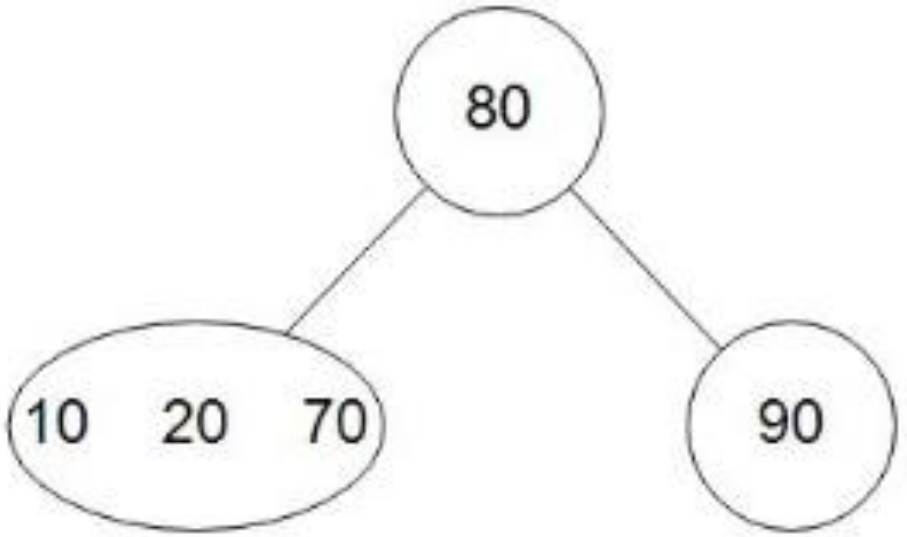
- Add 50.
  - Find the leaf where it belongs.
  - During the search we encounter the 10-20-70 node.
  - Split!



Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

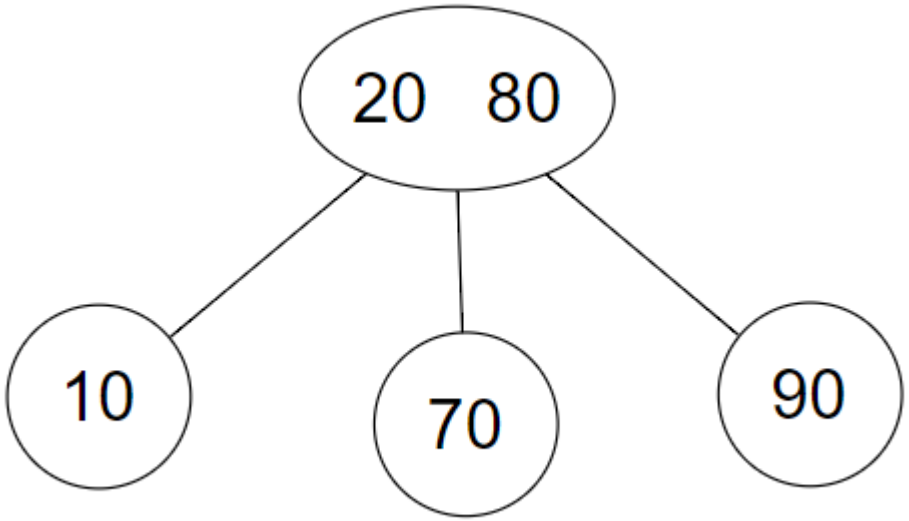
- Add 50.
  - Find the leaf where it belongs.
  - During the search we encounter the 10-20-70 node.
  - Split!



Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

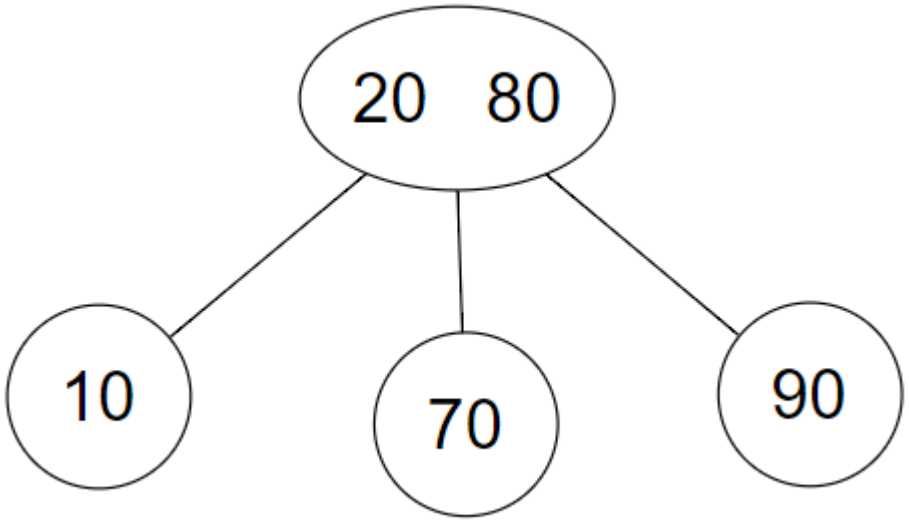
- Add 50.
  - Find the leaf where it belongs.
  - During the search we encounter the 10-20-70 node.
  - Split!



Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

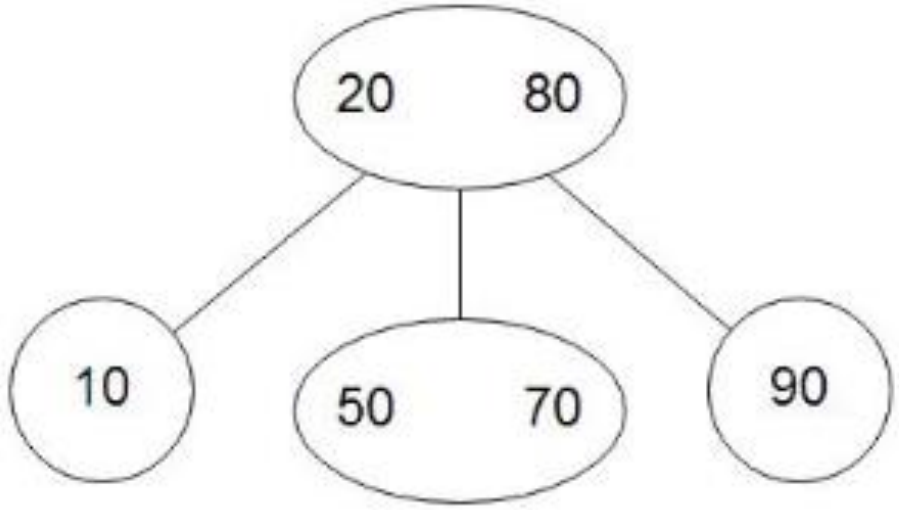
- Split!



Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

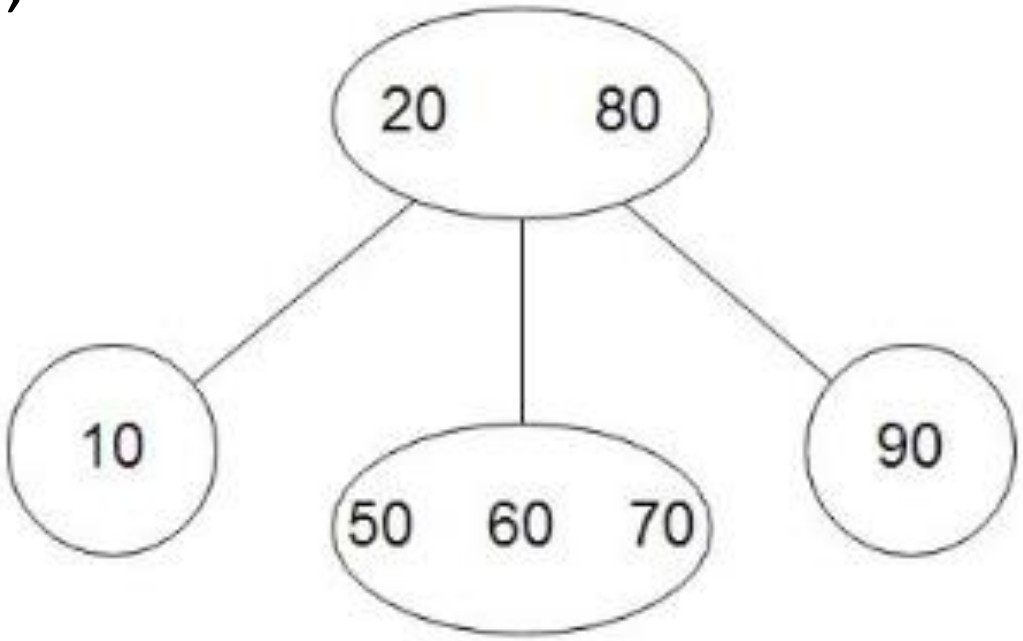
- Continue the search for the leaf for 50 and insert.



Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

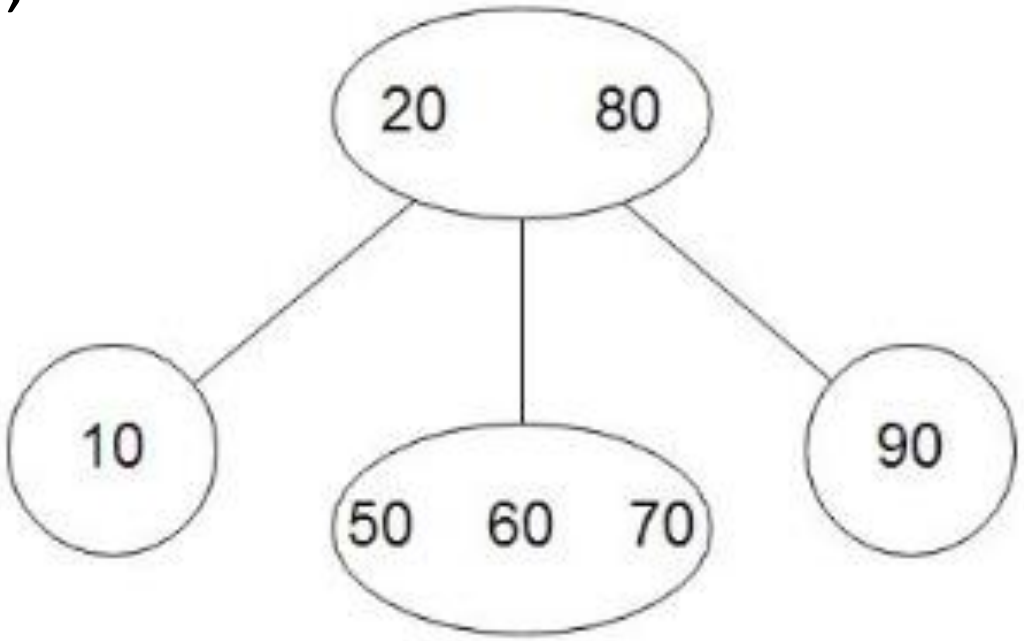
- Add 60.
  - Search for the leaf and add.
  - This creates a 4-node.



Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

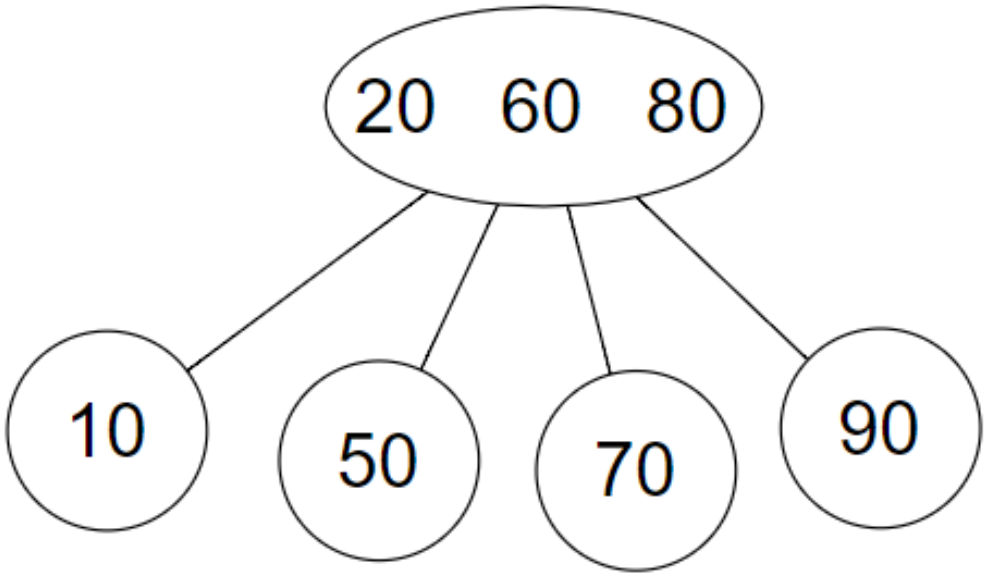
- Add 40.
  - Search for the leaf.
  - We encounter a 4-node.
  - Split!



Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

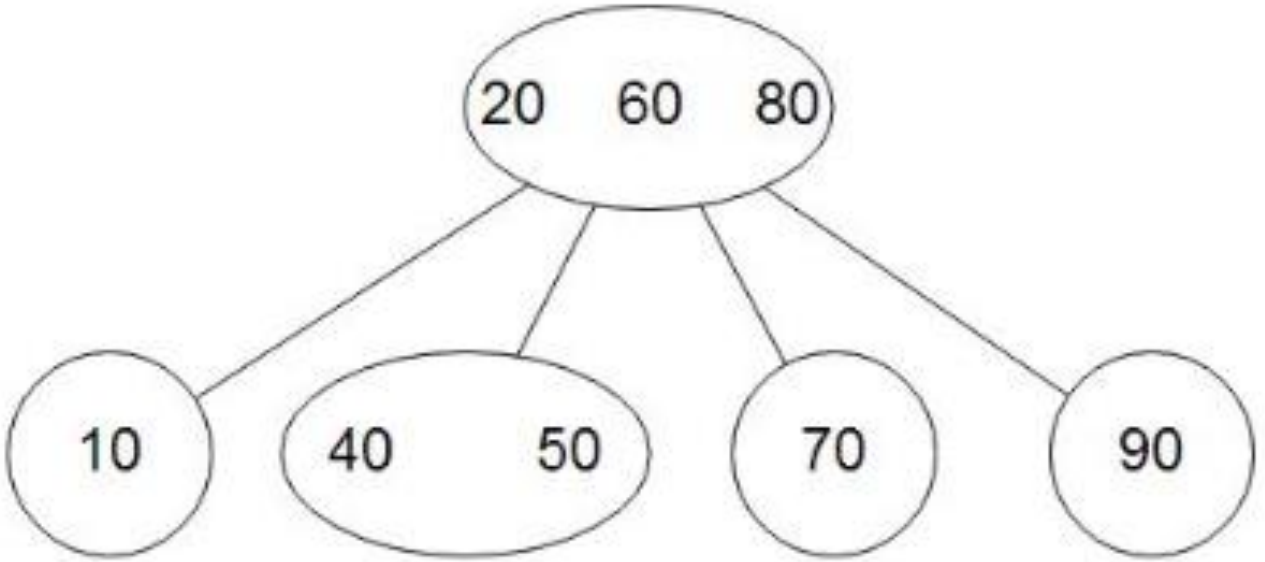
- Split!
  - Note that this split created another 4-node.
  - But that's allowed!
  - We only split them when we are *searching* for where to put the new element.



Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

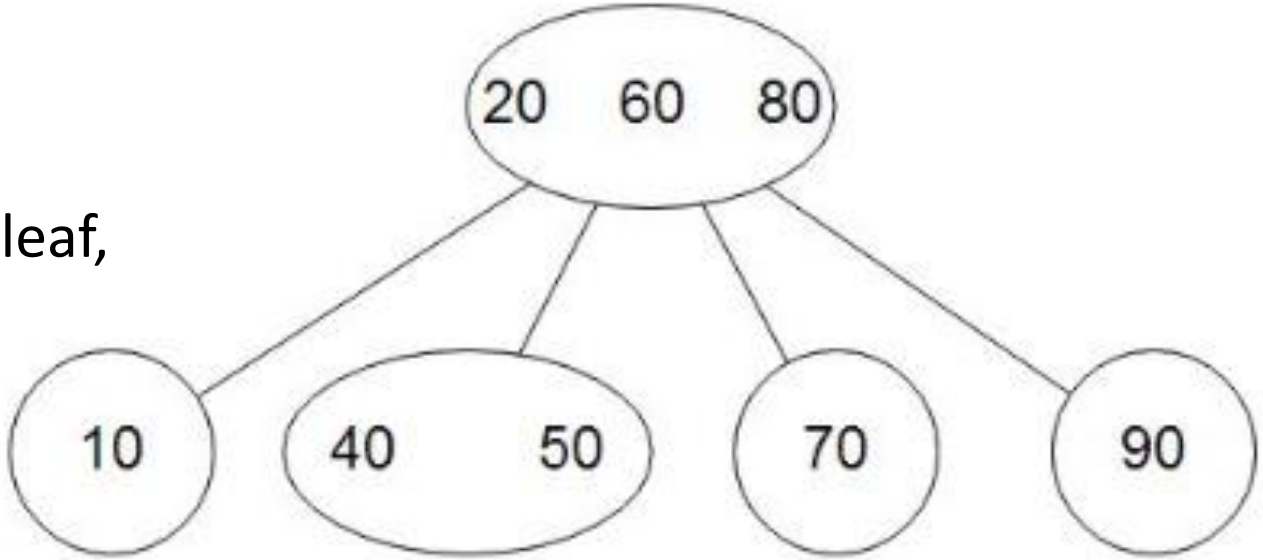
- Add 40.



Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

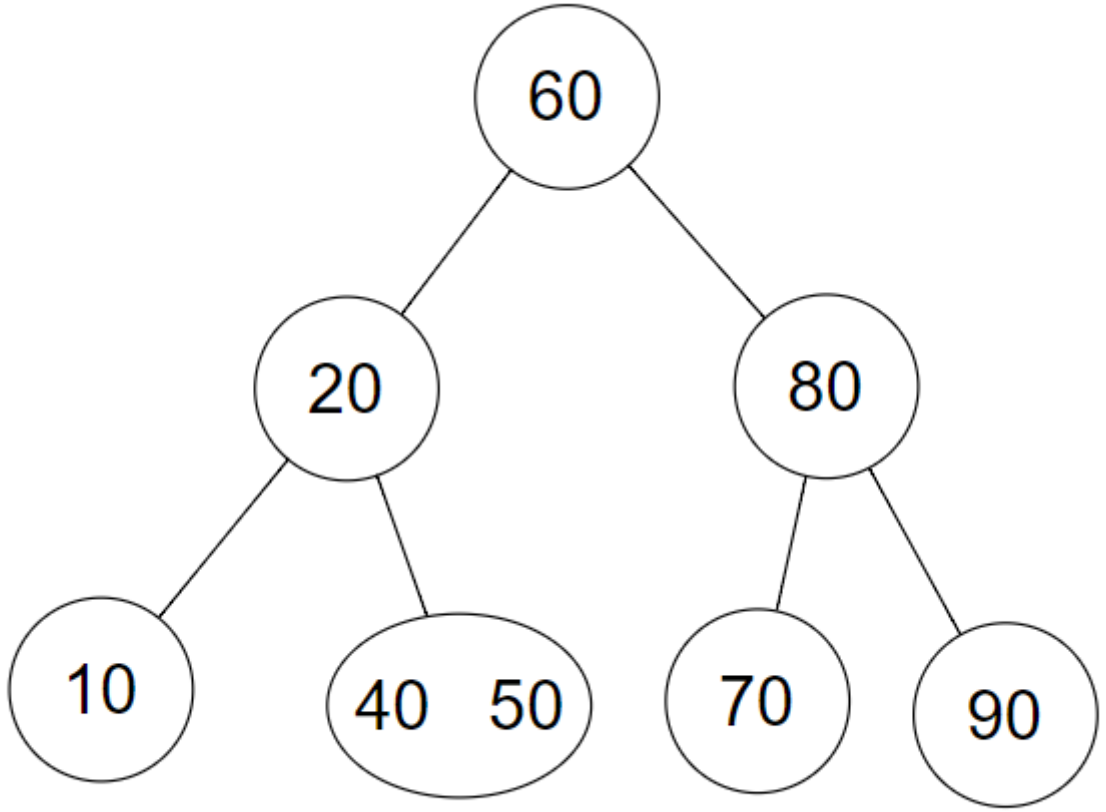
- Add 30.
  - As soon as we search for the leaf, we encounter the 4-node.
  - Split!



Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

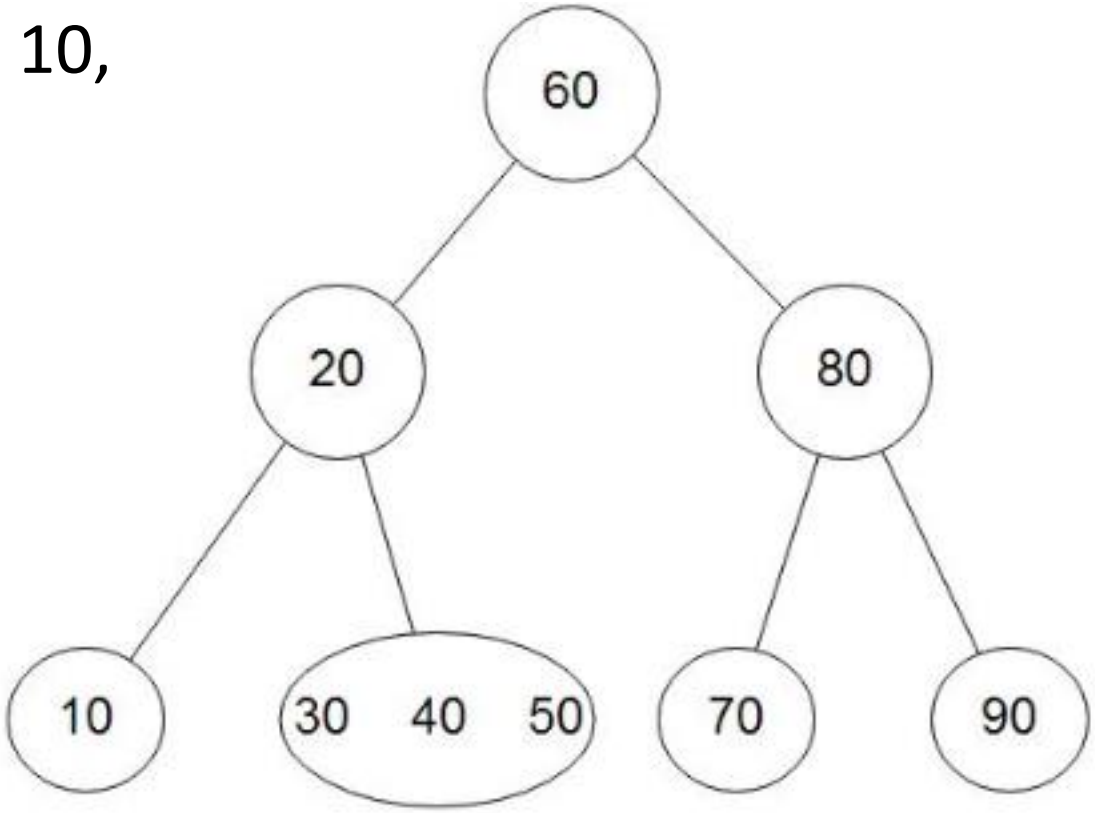
- Split!



Building a 2-4 Tree

- Example: Insert 70, 80, 90, 20, 10, 50, 60, 40, 30 into a 2-4 tree.

- Add 30.



General B-Trees

- A B-Tree of order m is a balanced search tree such that:
  - all leaves are at the lowest level
  - the root has no children or between 2 and m children
  - interior nodes have between ceiling(m/2) and m children
- For a 2-3 tree (m=3):
  - all leaves at lowest level
  - the root has 0, 2, or 3 children (it is a 2-node or 3-node)
  - all interior nodes have between 2 and 3 children (they are 2-nodes or 3-nodes)
- For a 2-4 tree (m=4):
  - all leaves are al the lowest level
  - the root has 0, 2, 3, or 4 children (it is a 2-node, 3-node, or 4-node)
  - all interior nodes have between 2, 3, and 4 children (they are 2-nodes, 3-nodes, or 4-nodes)
- B-Trees are often used for external storage (e.g., file systems and databases).