Unit 10: Trees

Anthony Estey

CSC 115: Fundamentals of Programming II

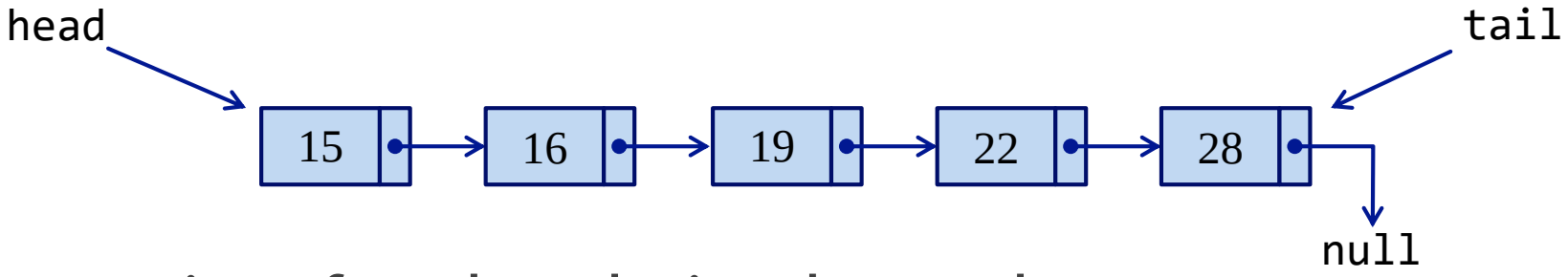
University of Victoria

Unit 10 Overview

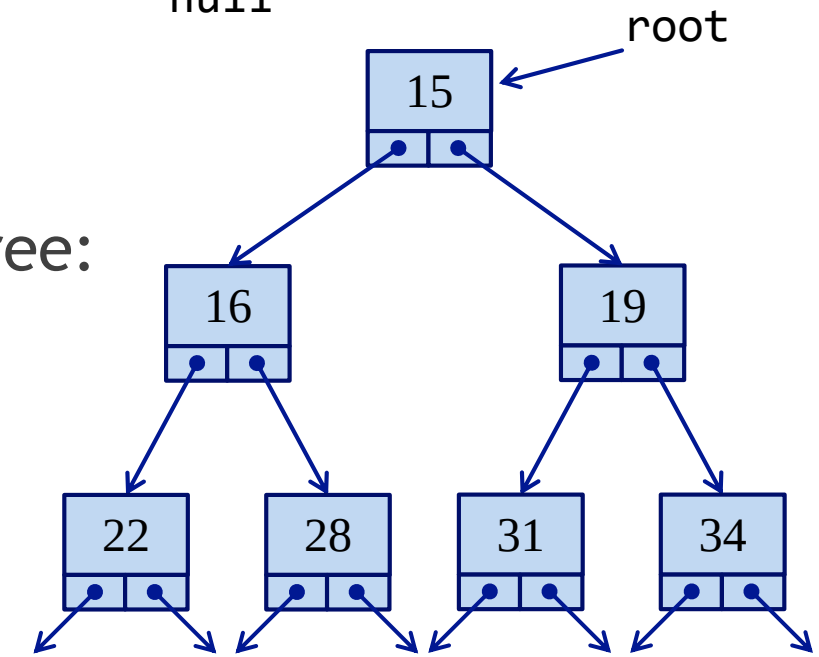
- Related Reading:
  - Textbook: Section 12.2
- Learning Objectives: (You should be able to…)
  - describe the properties associated with the Tree data structure
  - implement a tree using an array-based or reference-based representation
  - write code to calculate the height and depth of a node
  - describe the differences between the four tree traversal algorithms (in-order, pre-order, post-order and level-order)

Tree Structure

- We have worked with linked lists already:
  - nodes are chained together, starting at the **head**, and ending at the **tail**:



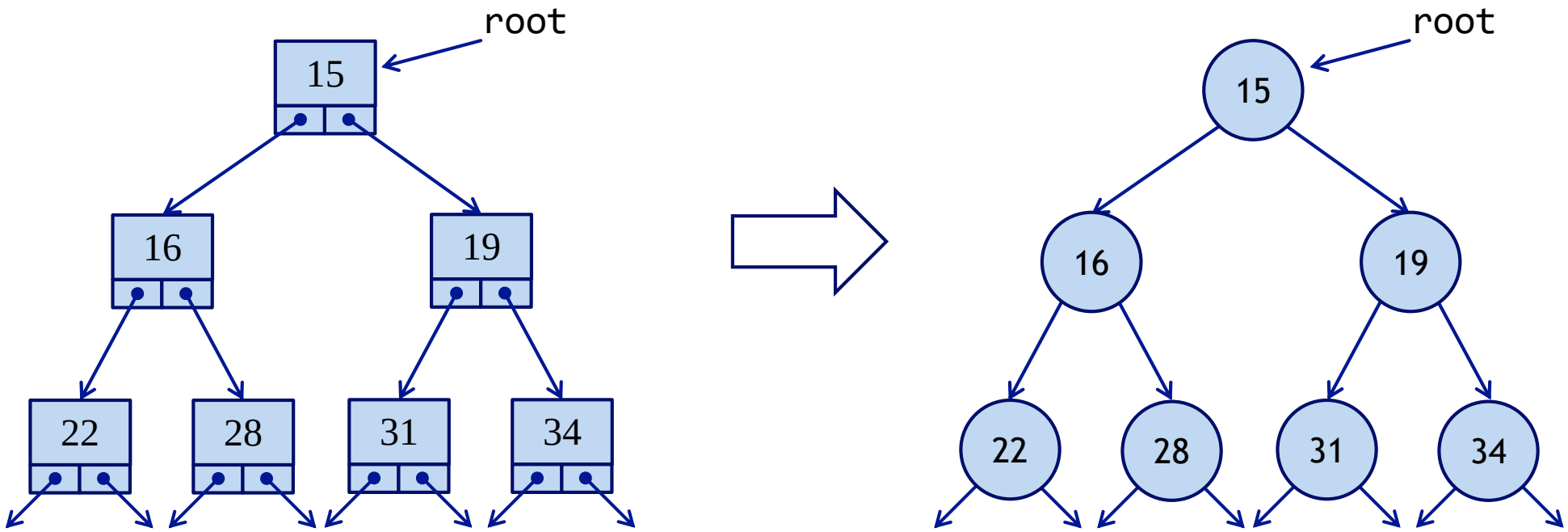
- Trees also consist of nodes chained together
  - trees start at the top (called the **root**)
  - and each node links to other nodes lower in the tree:



- Nodes have many similarities
  - data field, references to other nodes

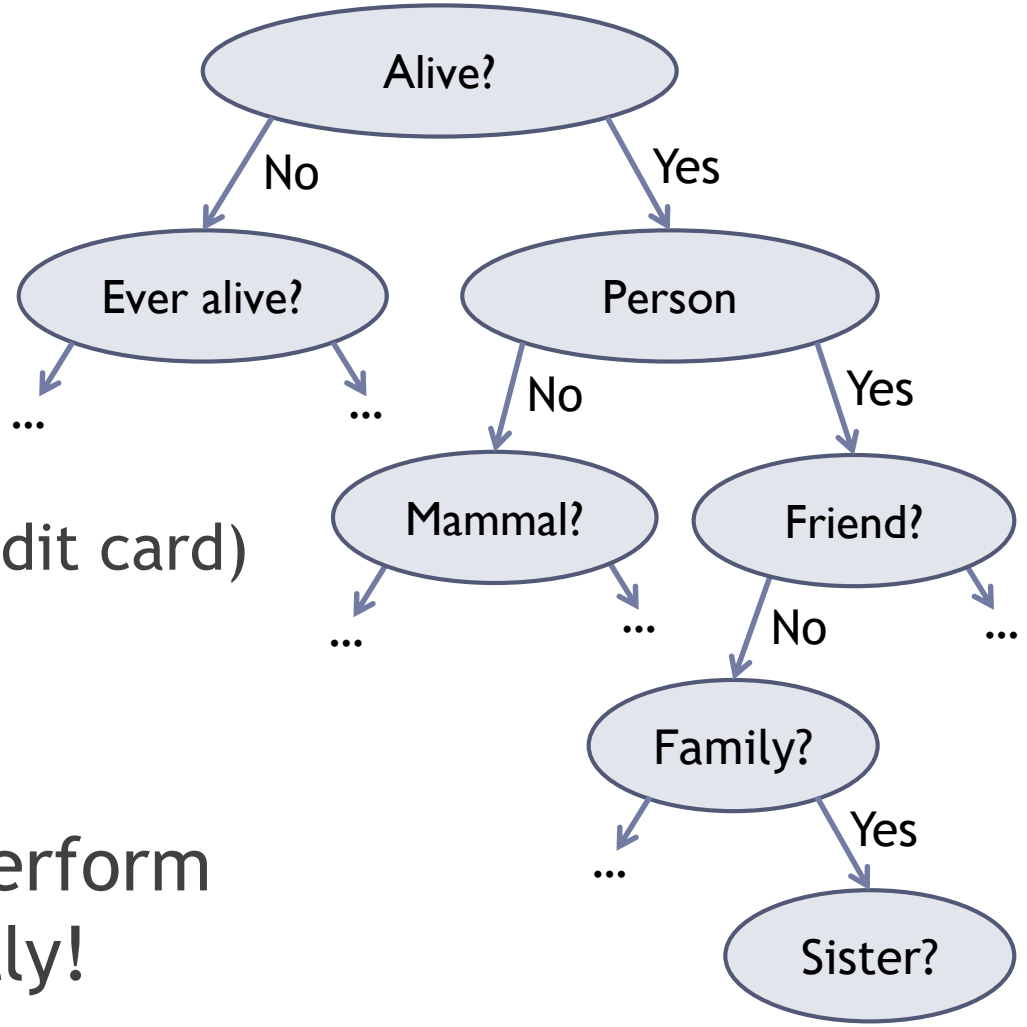
Tree Visualization

- Often trees are represented visually as vertices and edges
  - each vertex is draw as a circle, and represents the data component of a node
  - the edges represent the references to other nodes within the data structure



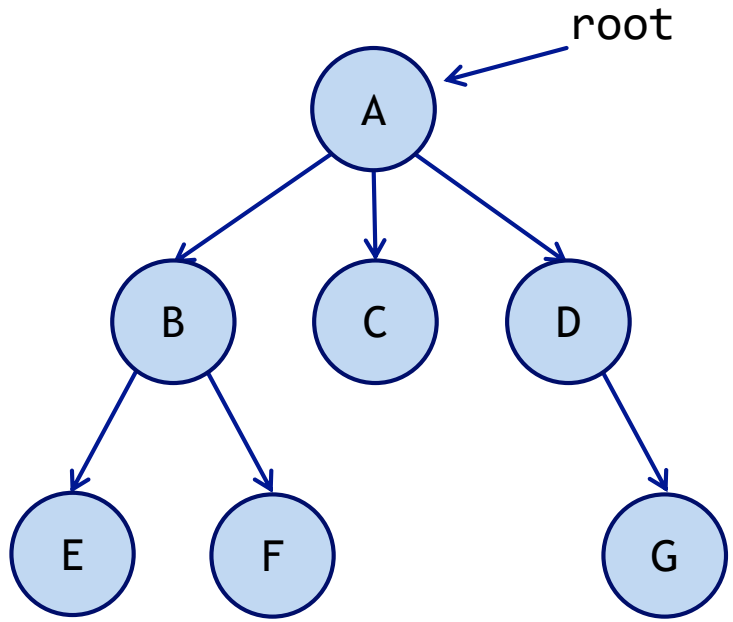
Why Trees?

- Application:
  - Why would we use a Tree instead of a list?
  - One example: Decision tree
- Decision trees are used for many things:
  - 21 questions (shown to the right)
  - Going through menu options (phone your credit card)
  - Medical diagnosis
  - Video games
- We will also see they will even help us perform operations on a list of elements efficiently!



Tree Relationships

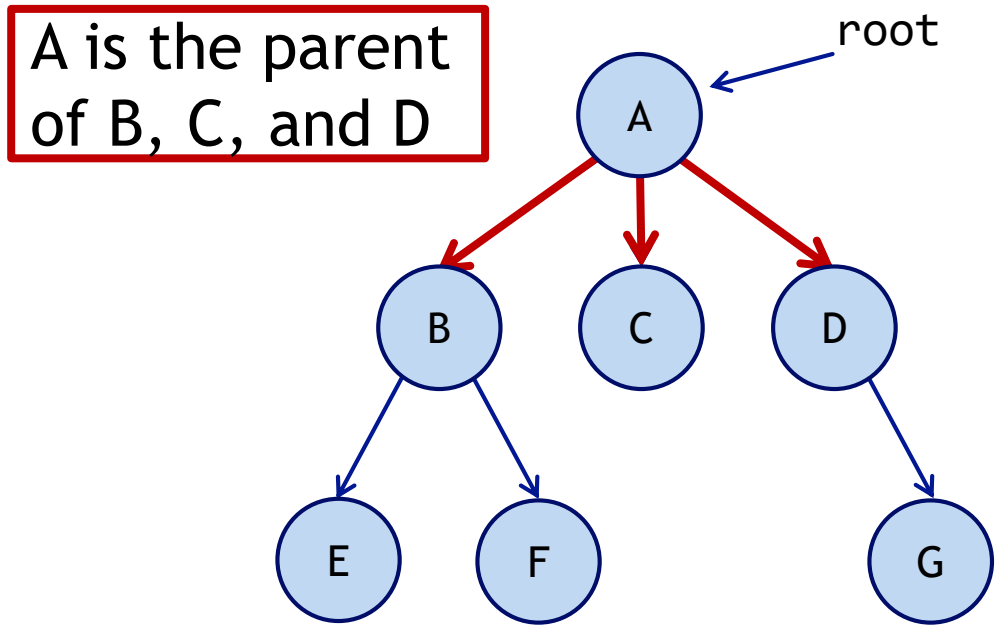
- Here is some of the general terminology associated with the relationship between nodes in a tree



Tree Relationships

- Here is some of the general terminology associated with the relationship between nodes in a tree

- Node v is said to be a **child** of u, and u the **parent** of v if:
  - there is a edge connecting u and v
  - u is above v in the tree



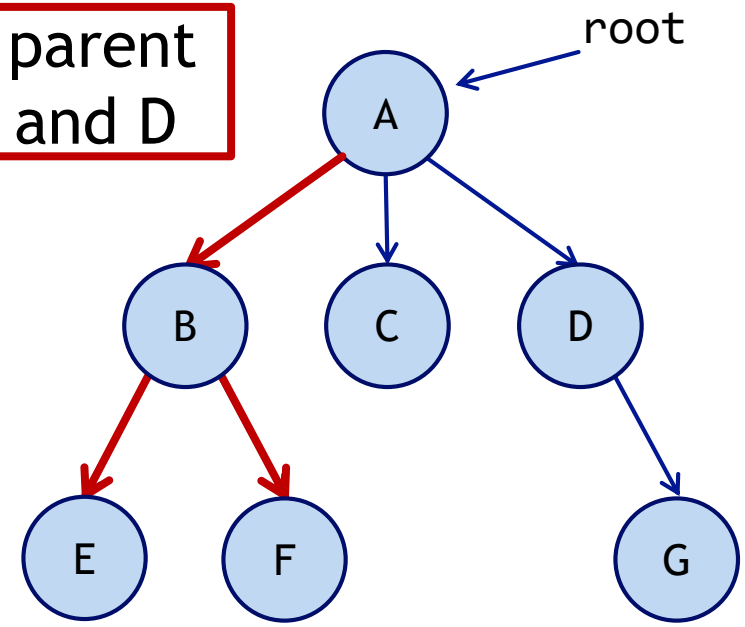
Tree Relationships

- Here is some of the general terminology associated with the relationship between nodes in a tree

- Node v is said to be a **child** of u, and u the **parent** of v if:
  - there is a edge connecting u and v
  - u is above v in the tree

- This relationship can be generalized:
  - B, E, and F are **descendants** of A

A is the parent of B, C, and D



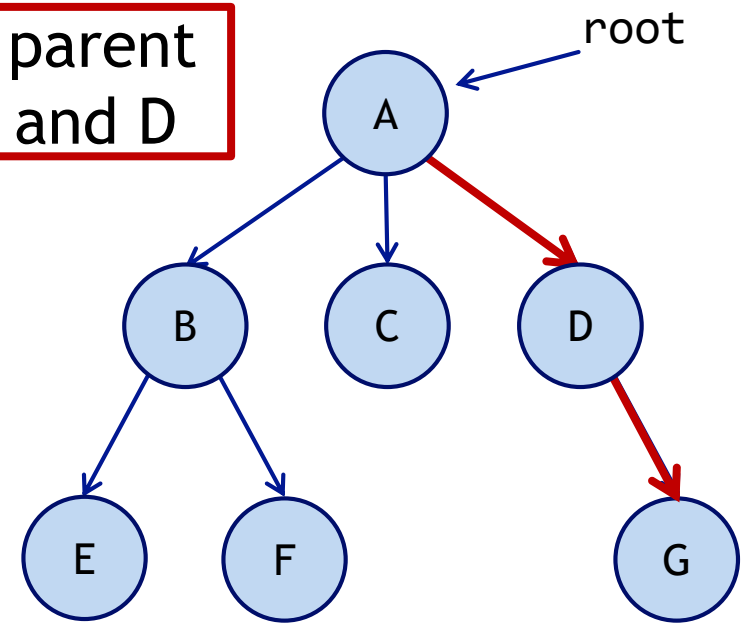
Tree Relationships

- Here is some of the general terminology associated with the relationship between nodes in a tree

- Node v is said to be a **child** of u, and u the **parent** of v if:
  - there is a edge connecting u and v
  - u is above v in the tree

- This relationship can be generalized:
  - B, E, and F are **descendants** of A
  - D and A are **ancestors** of G

A is the parent of B, C, and D



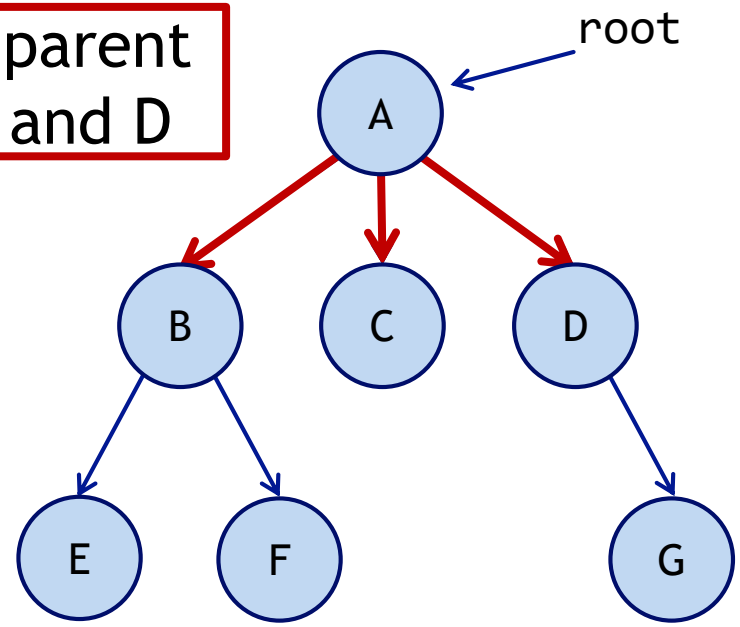
Tree Relationships

- Here is some of the general terminology associated with the relationship between nodes in a tree

- Node v is said to be a **child** of u, and u the **parent** of v if:
  - there is a edge connecting u and v
  - u is above v in the tree

- This relationship can be generalized:
  - B, E, and F are **descendants** of A
  - D and A are **ancestors** of G
  - B, C, and D are **siblings**

A is the parent of B, C, and D



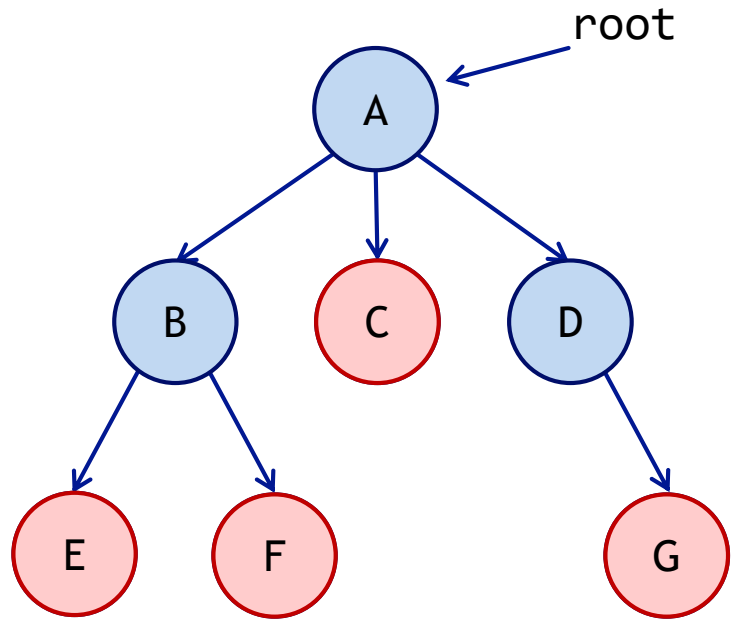
Tree Terminology Reference

- root: the single no with no parent
- leaf: a node with no children
- child: a node pointed to by me
- parent: the node that points to me
- sibling: another child of my parent
- ancestor: my parent or my parent’s parent
- descendent: my child or my child’s descendent
- subtree: a node and its descendents

More Terminology

- A **leaf** is node with no children

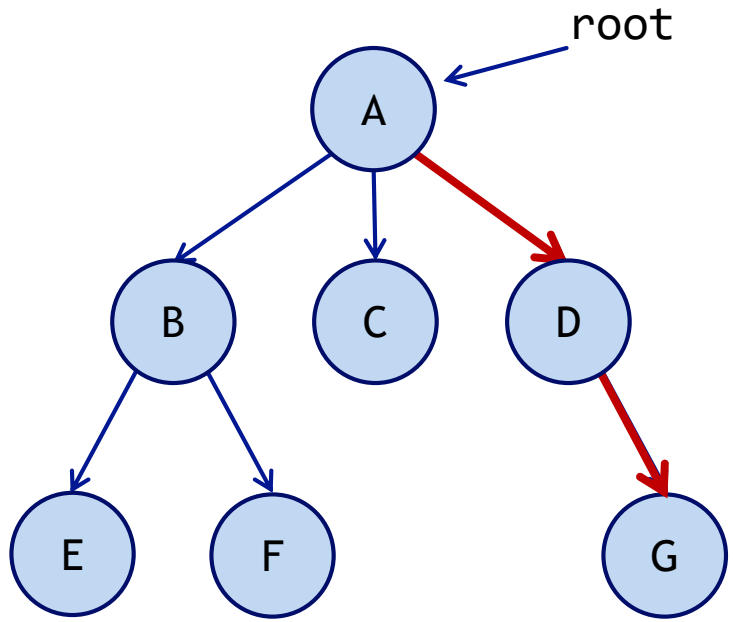
C, E, F, and G are leaf nodes



More Terminology

- A **leaf** is node with no children
- A **path** is a sequence of nodes $v_1, \ldots, v_n$ where v_i is a parent of v_{i+1}

A path from A to D to G

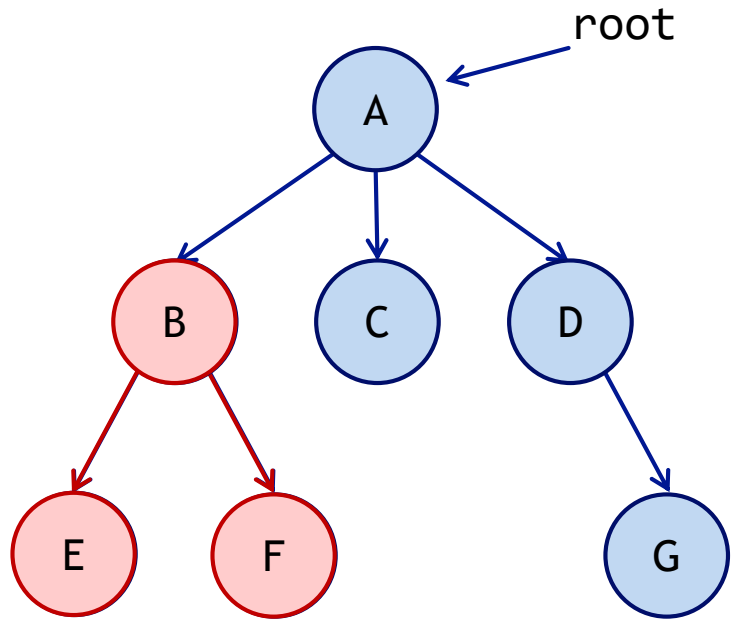


More Terminology

- A **leaf** is node with no children
- A **path** is a sequence of nodes $v_1, \ldots, v_n$ where v_i is a parent of v_{i+1}
- A **subtree** is a any node in the tree along with its descendants

There is a subtree rooted at B

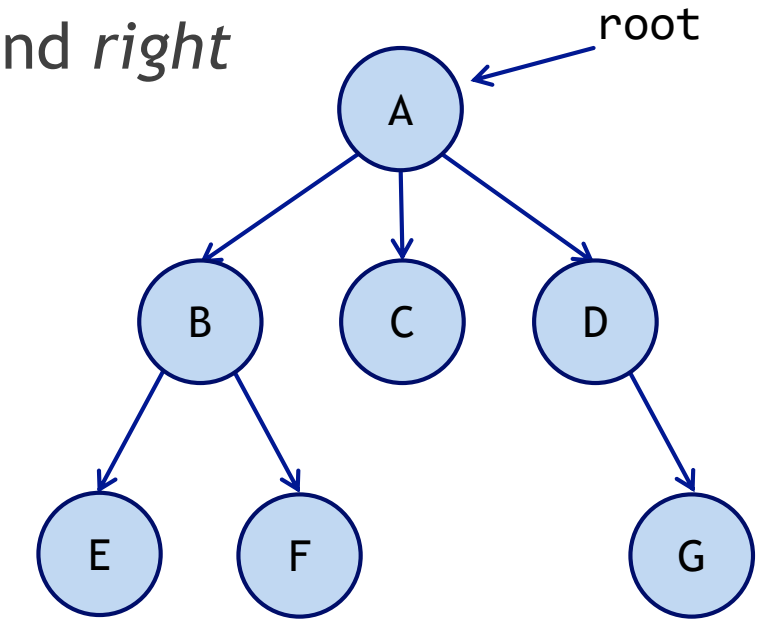
It contains nodes B, E, and F



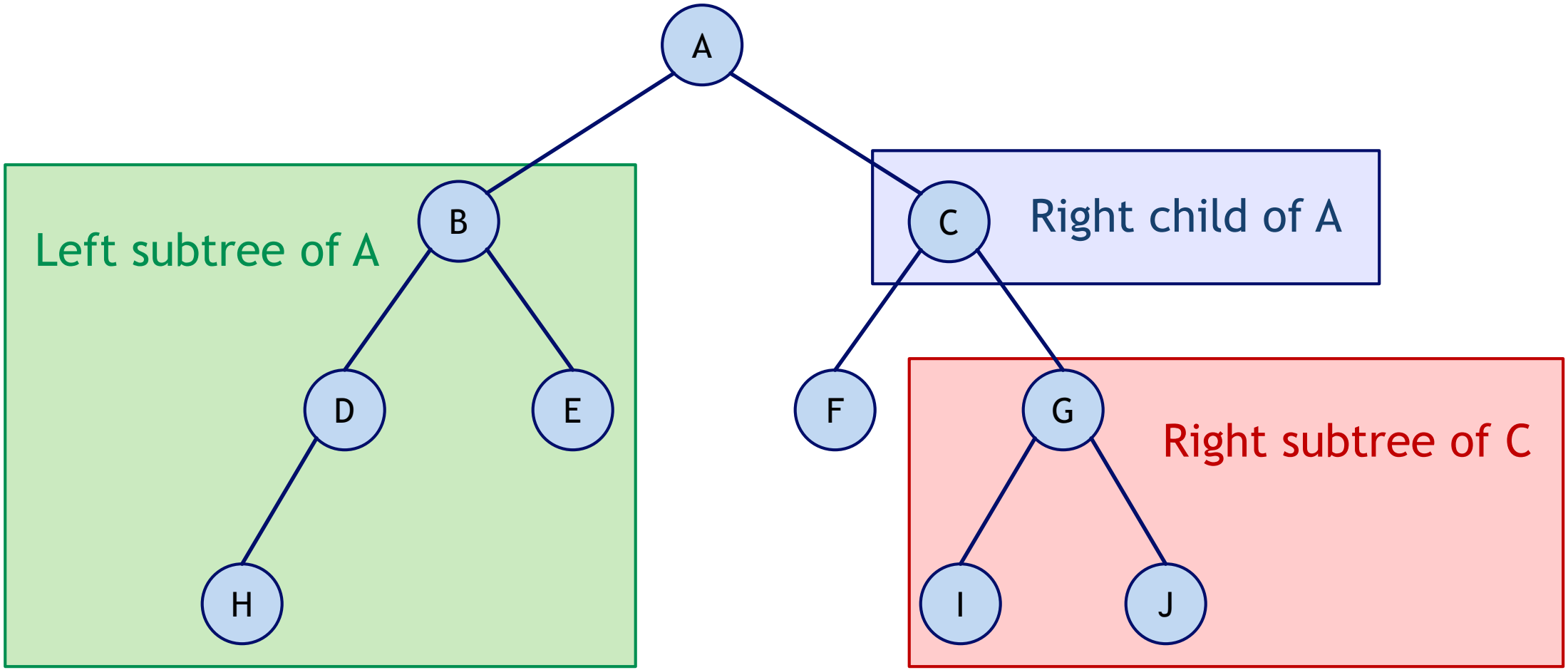
More Terminology

- A **leaf** is node with no children
- A **path** is a sequence of nodes $v_1, \ldots, v_n$ where v_i is a parent of v_{i+1}
- A **subtree** is a any node in the tree along with its descendants
- A **binary tree** is a tree with at most two children per node
  - The children are typically referred to as *left* and *right*

This is not a binary tree;
node A has 3 children



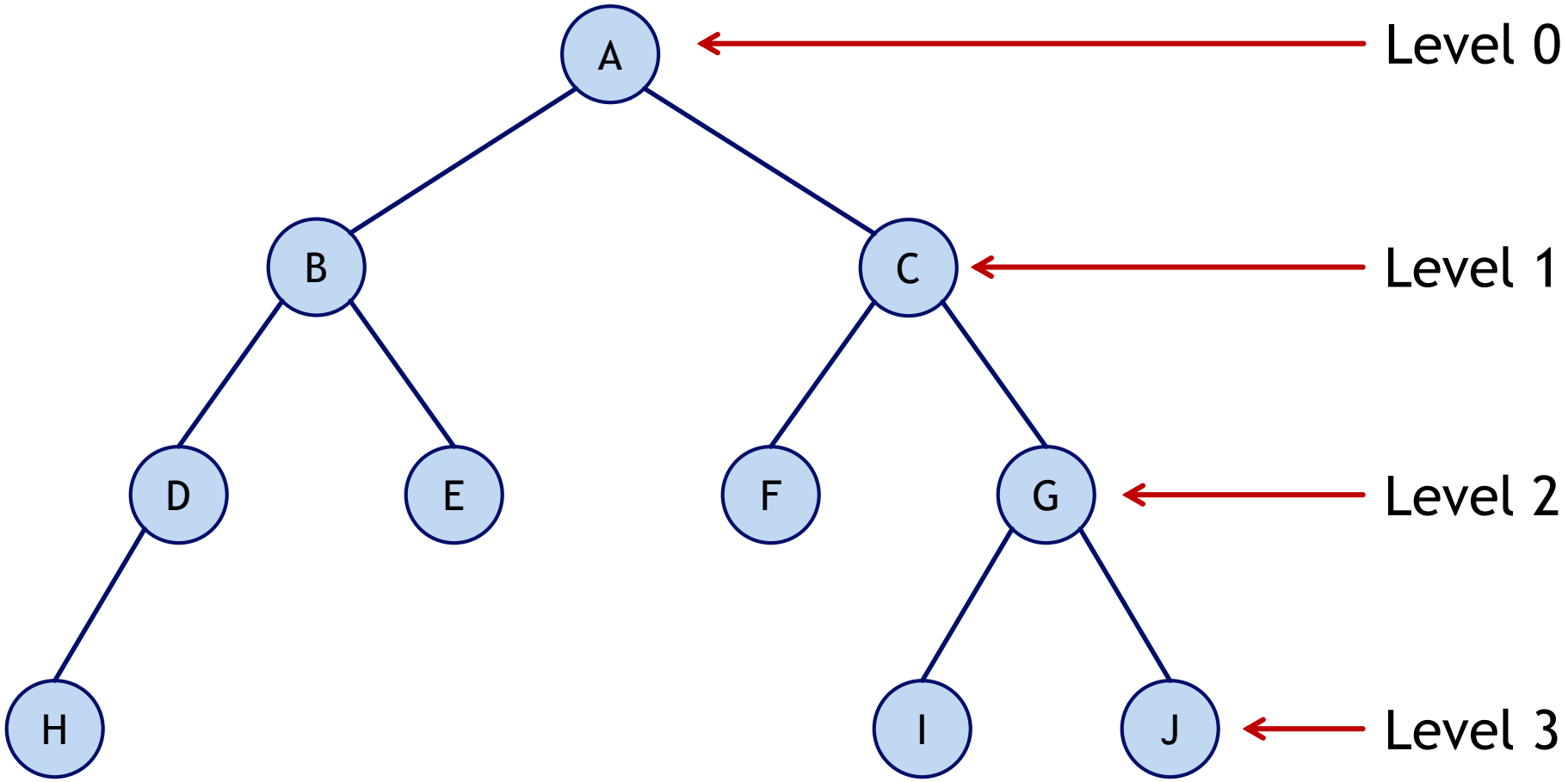
More Terminology



Measuring Height and Depth

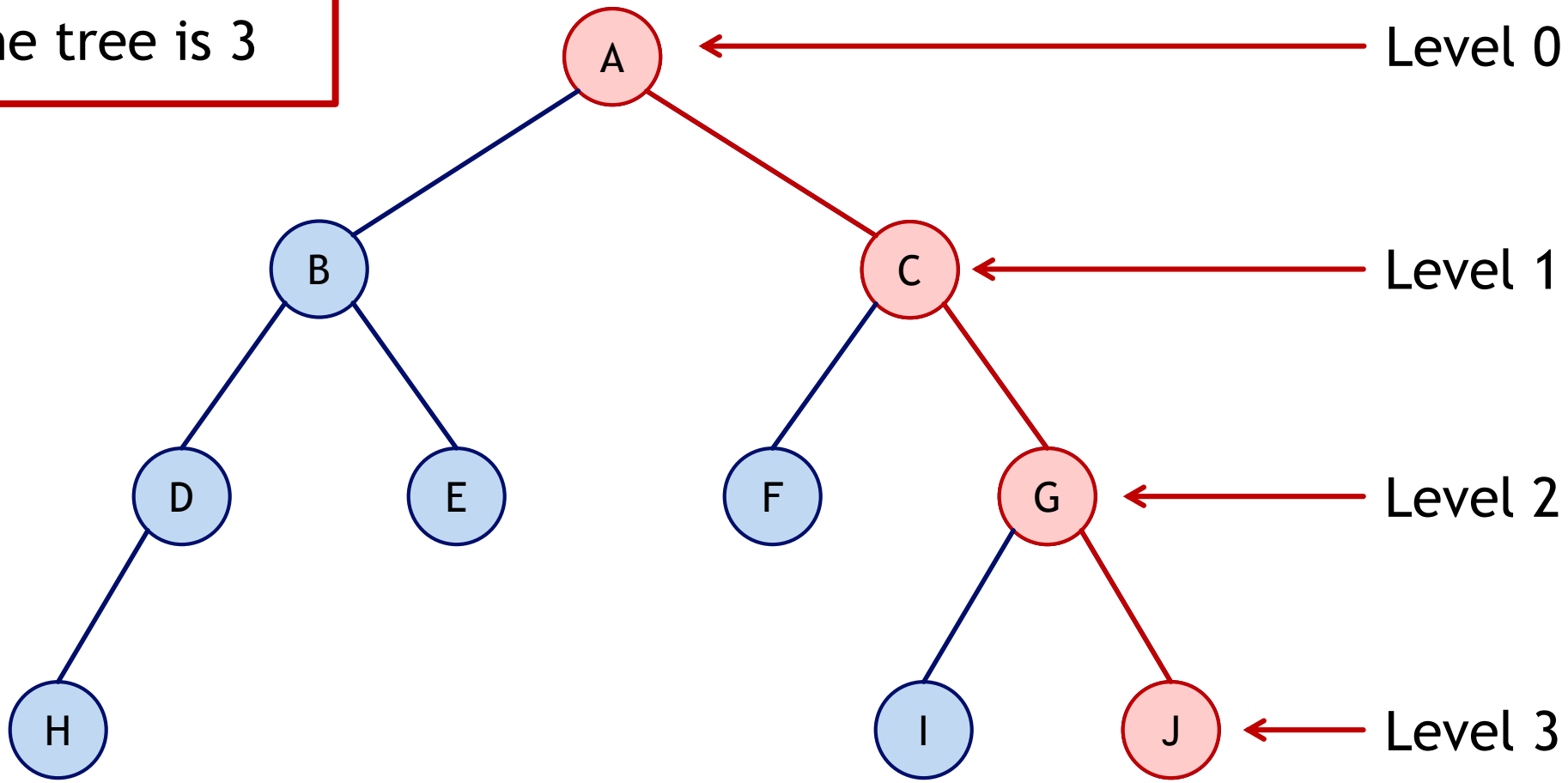
- The **height** of node v is the length of the longest path from v to a leaf
  - The height of a tree is the height of the $root$
- The **depth** of node v is the length of the path from the $root$ to v
  - This is also referred to as the **level** of the node
- This leads to a slightly different definition of the height of a tree:
  - The height of a tree is the number of different levels found in the tree

Measuring Height and Depth



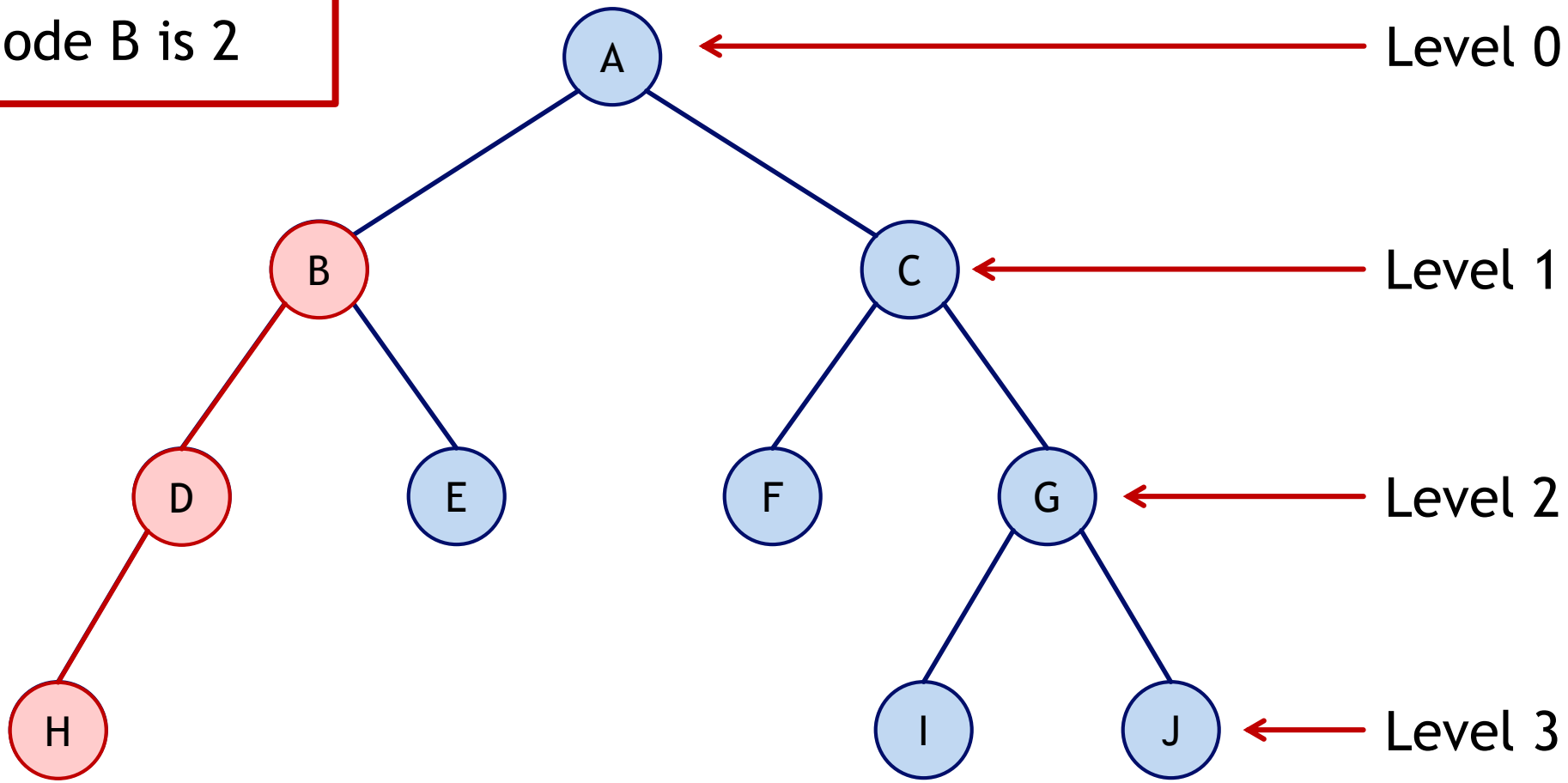
Measuring Height and Depth

Height of the tree is 3



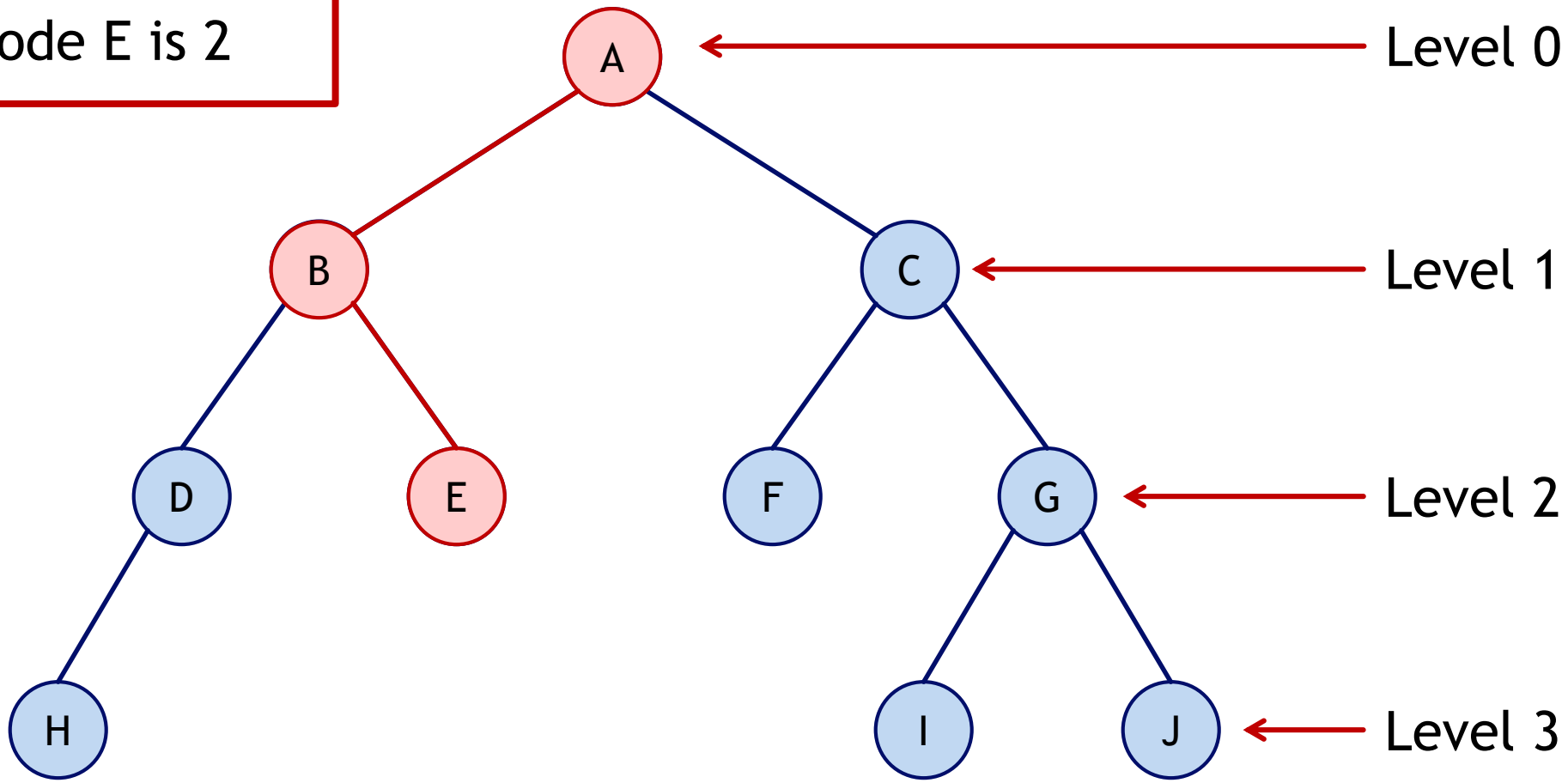
Measuring Height and Depth

Height of node B is 2



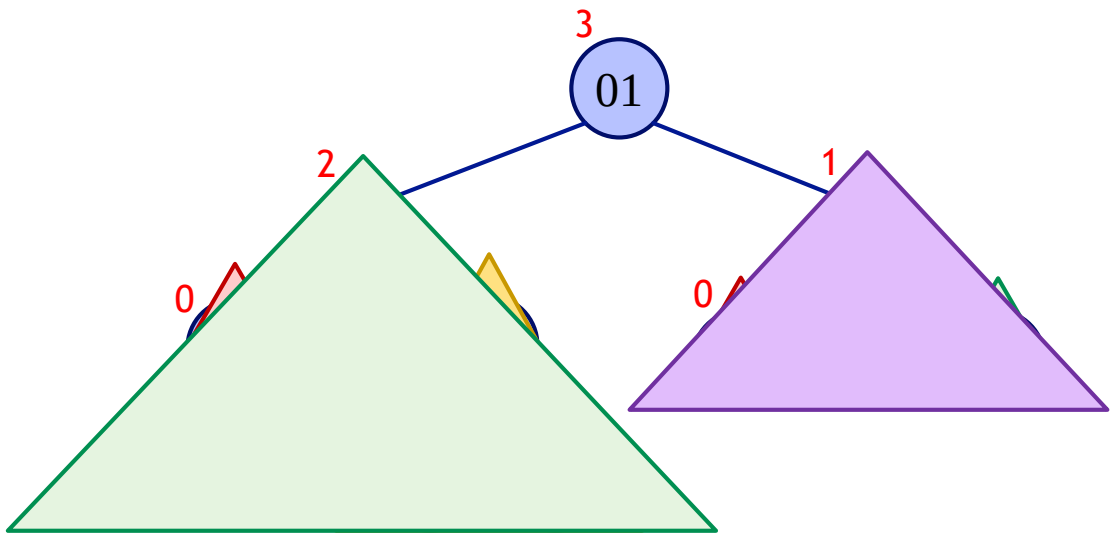
Measuring Height and Depth

Depth of node E is 2



Height

- The height of a binary tree T is defined as follows:
  - If T is a *leaf*, the height of T is 0
  - Otherwise, the height of T is $1 +$ the maximum height of T's left and right subtrees

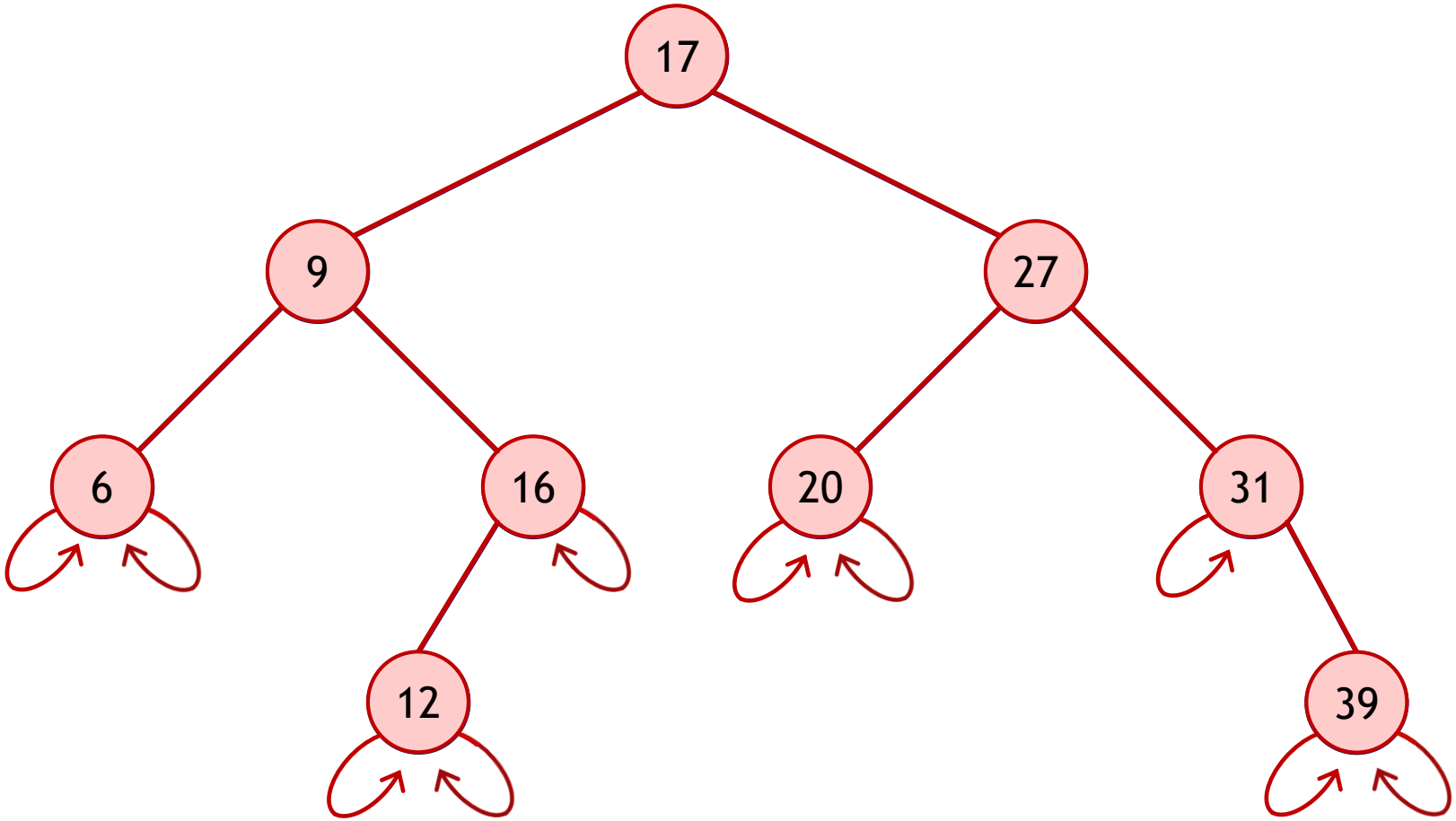


Tree Traversals

- To visit all of the elements in a list, we typically went from the front (head) to the back (tail), or from back to the front
- The structure of a tree allows us to visit all of the elements in different ways
- Later we will see it is necessary to visit the elements in different orders depending on the application
- There are three traversal algorithms that are naturally recursive:
  - pre-order
  - in-order
  - post-order

Tree Traversals

- Pre-order Traversal
  - visit node
  - visit left subtree
  - visit right subtree

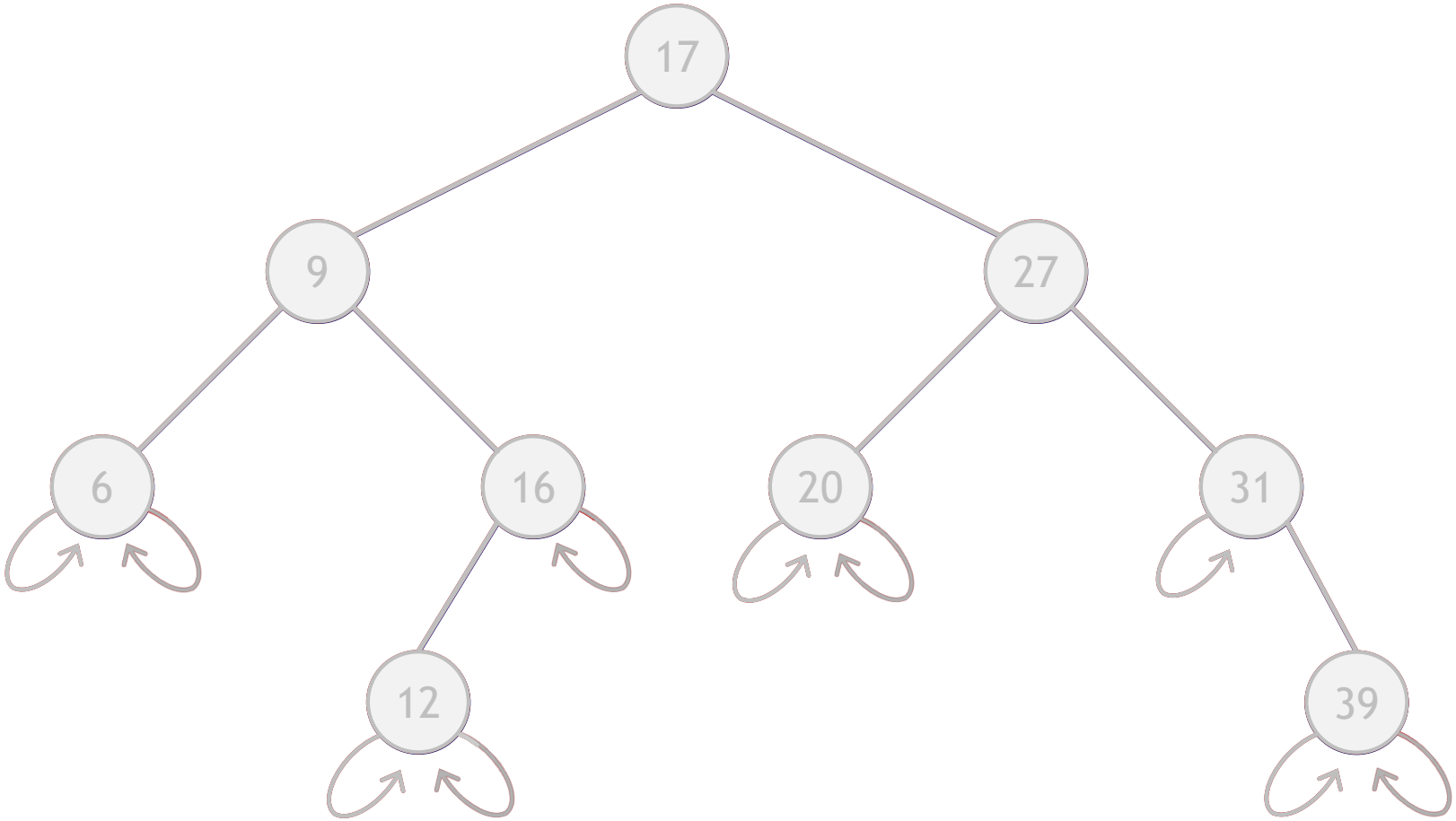


- Output: 17 9 6 16 12 27 20 31 39

Done!

Tree Traversals

- In-order Traversal
  - visit left subtree
  - visit node
  - visit right subtree

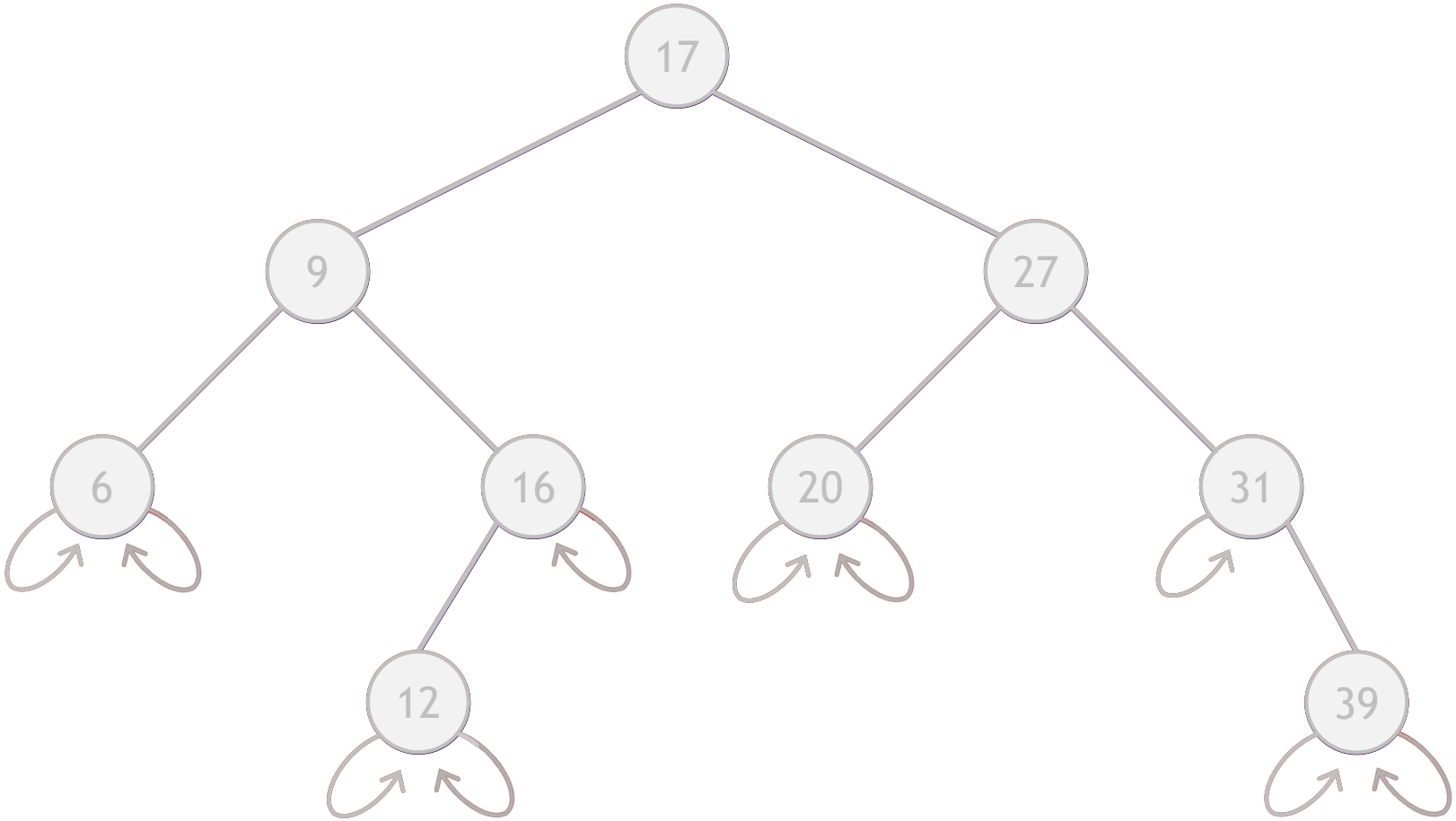


- Output: 6 9 12 16 17 20 27 31 39

Done!

Tree Traversals

- Post-order Traversal
  - visit left subtree
  - visit right subtree
  - visit node

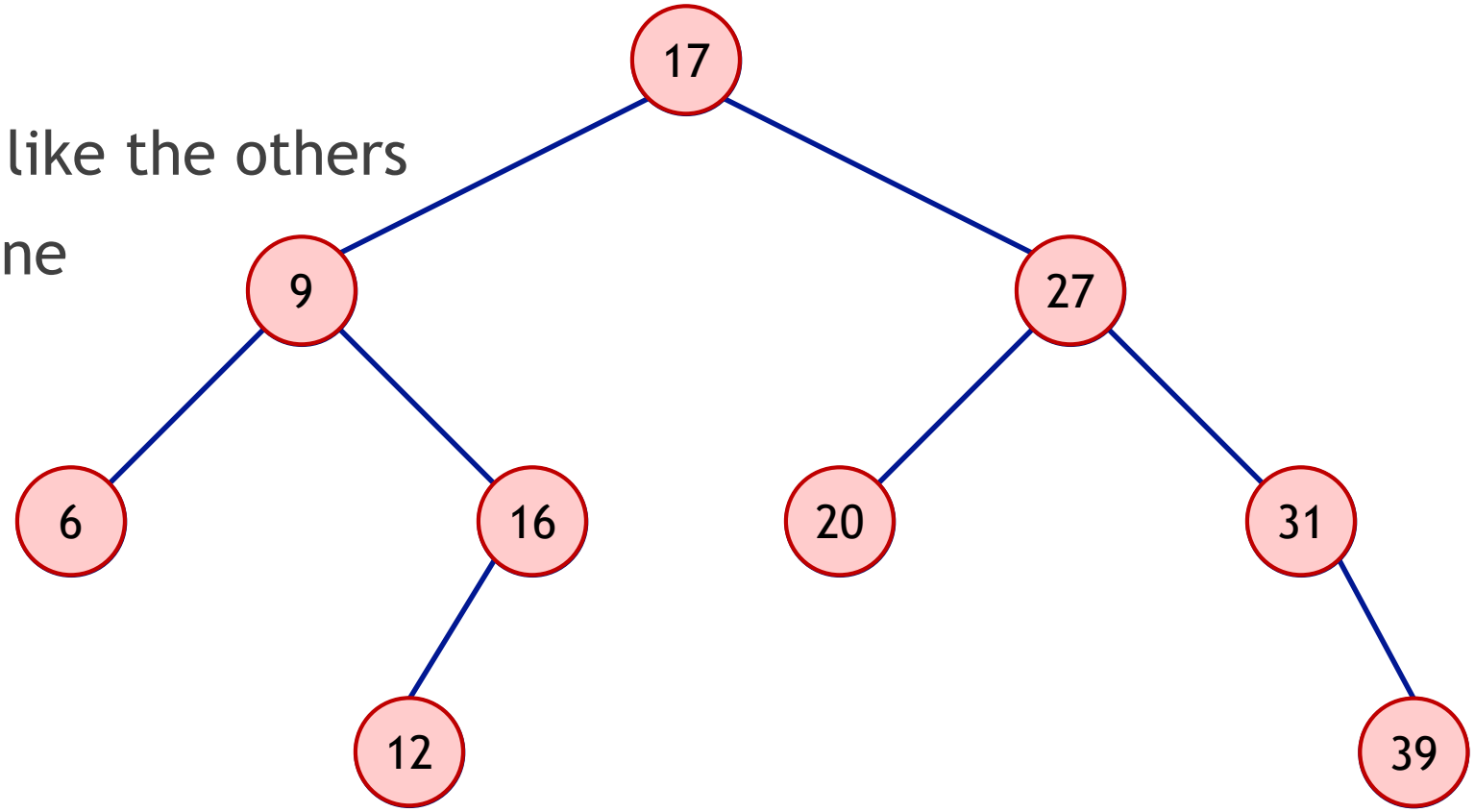


- Output: 6 12 16 9 20 39 31 27 17

Done!

Tree Traversals

- Level-order
  - Not naturally recursive, like the others
  - visit each level one by one

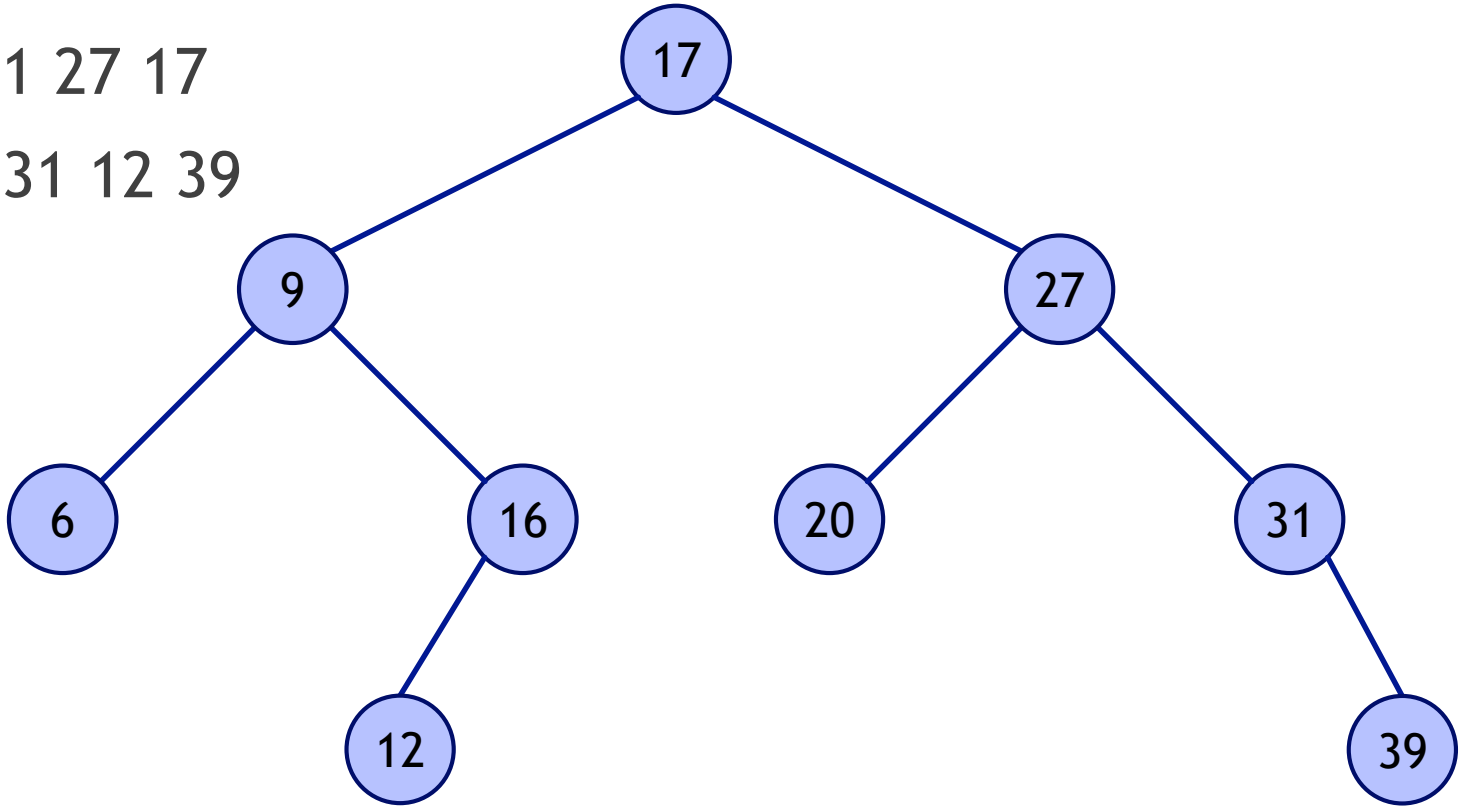


- Output: 17 9 27 6 16 20 31 12 39

Done!

Tree Traversals - Recap

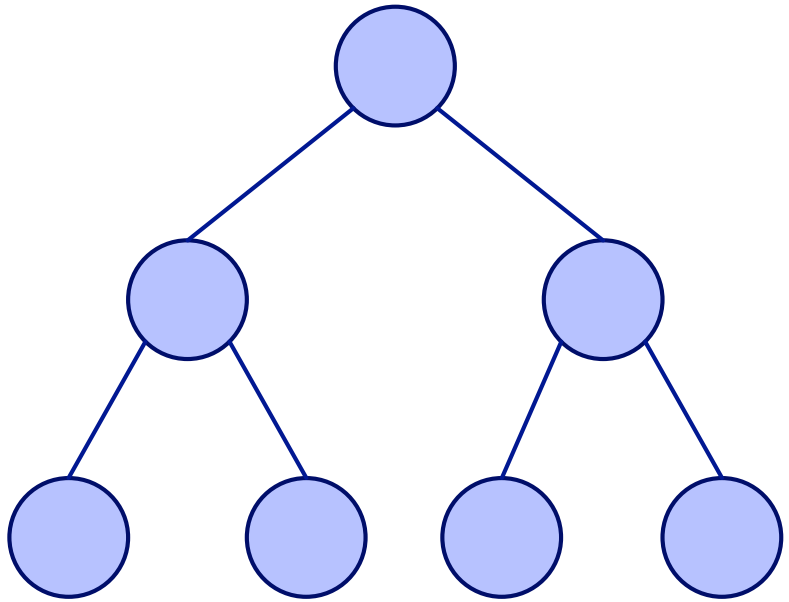
- Same tree, but the elements are visited in different orders:
  - Pre-order: 17 9 6 16 12 27 20 31 39
  - In order: 6 9 12 16 17 20 27 31 39
  - Post-order: 6 12 16 9 20 39 31 27 17
  - Level-order: 17 9 27 6 16 20 31 12 39



Binary Tree Classifications (Perfect)

- A binary tree is *perfect* if:
  - No node has only one child
  - All leaf nodes have the same depth

- A perfect binary tree of height h has
  - $2^{h+1} - 1$ total nodes, of which 2^h are leaves
  - Solving for h, we get $h \leq \log_2(n+1) - 1$
  - Height is bound by $\log_2 n$



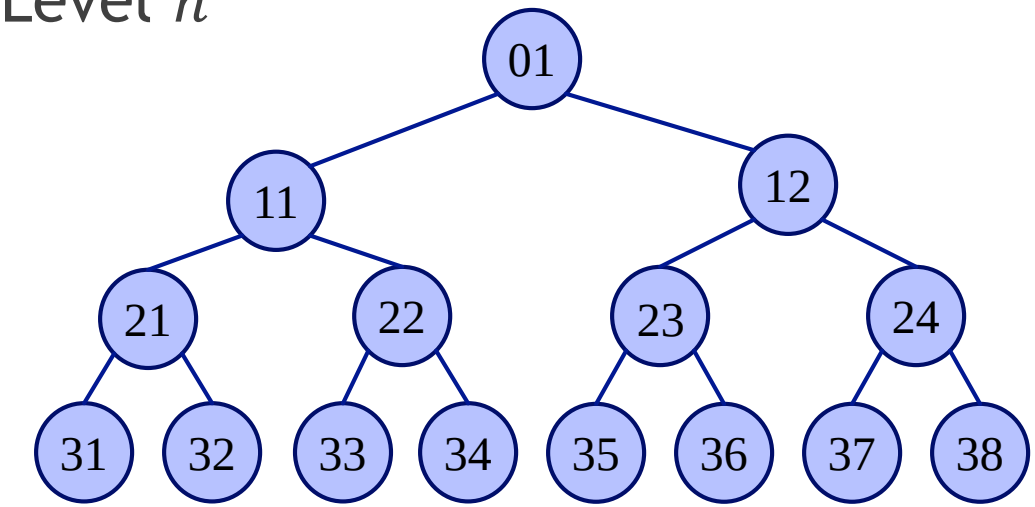
Binary Tree Classifications (Perfect)

- In a perfect binary tree, each level doubles the number of nodes
  - Level 1 has 2 nodes (2^1)
  - Level 2 has 4 nodes (2^2), or 2 times the number of nodes from Level 1
  - Level 3 has 8 nodes (2^3), or 2 times the number of nodes from Level 2
- Therefore a tree with h levels has $2^{h+1} - 1$ nodes
  - With 2^h nodes on Level h

Level 1: 2 nodes

Level 2: 4 nodes

Level 3: 8 nodes



Total nodes: 3 ($2^2 - 1$)

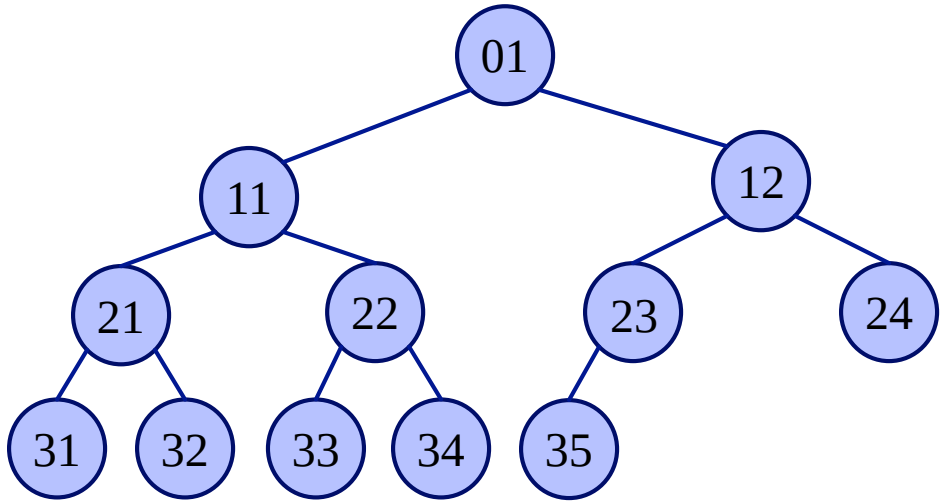
Total nodes: 7 ($2^3 - 1$)

Total nodes: 15 ($2^4 - 1$)

Binary Tree Classifications (Complete)

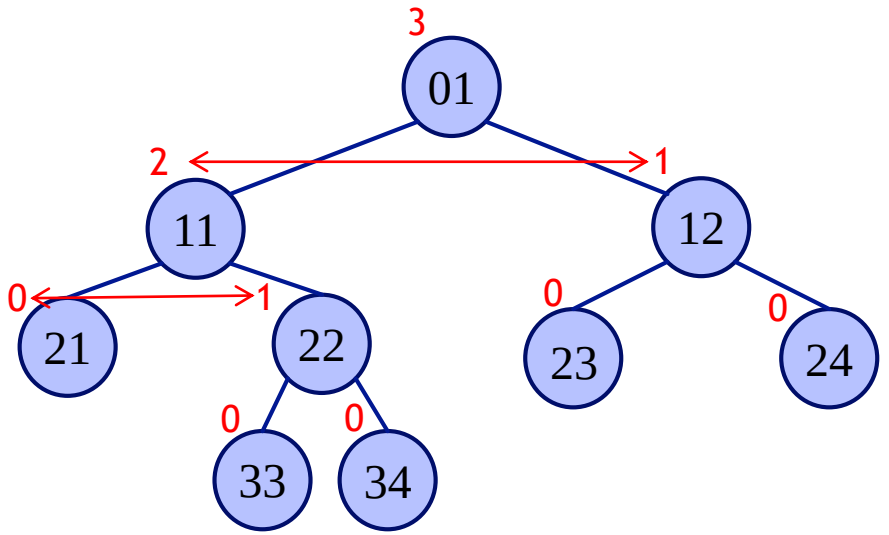
- A binary tree is *complete* if
  - The leaves are found on at most 2 levels
  - The second to bottom level is completely filled
  - The leaves on the bottom level are as far left as possible

- Missing nodes (from a perfect tree):
  - are on the bottom level
  - on the right-most side



Binary Tree Classifications (Balanced)

- A binary tree is *balanced* if
  - For any node v in the tree, the height of v's right subtree is at most one different from the height of its left subtree)



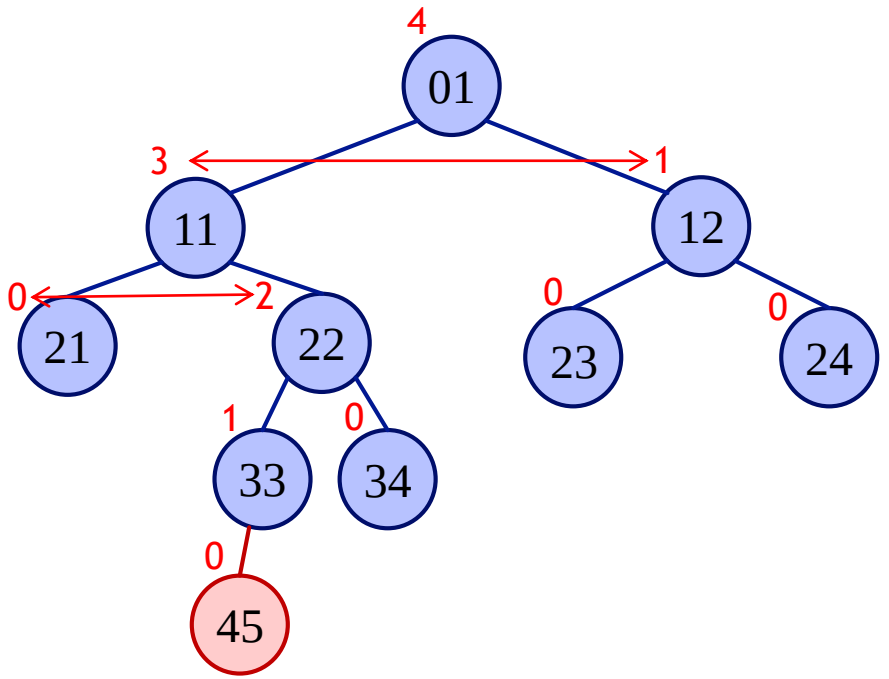
Binary Tree Classifications (Balanced)

- A binary tree is *balanced* if
  - For any node v in the tree, the height of v's right subtree is at most one different from the height of its left subtree)

- What about if we add node 45?

Node 11 is unbalanced!

So is the root node (01)

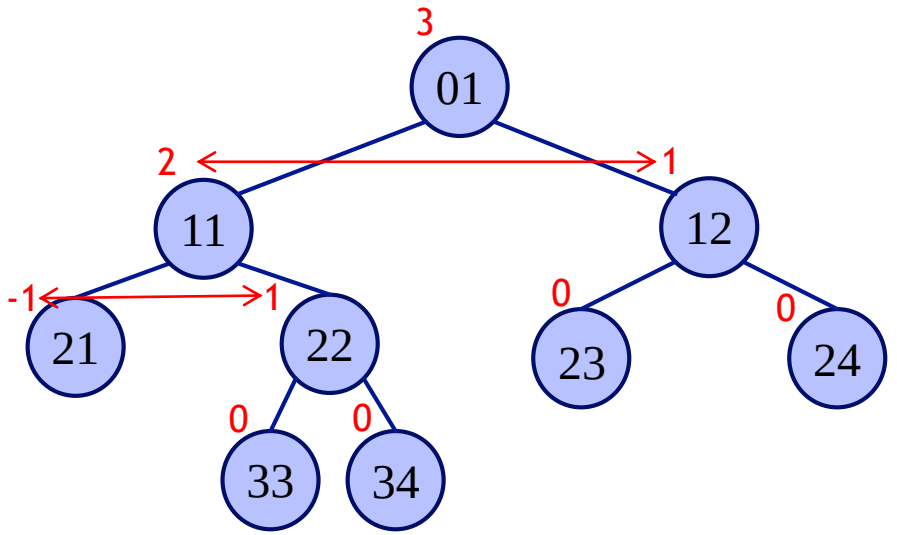


Binary Tree Classifications (Balanced)

- A binary tree is *balanced* if
  - For any node v in the tree, the height of v's right subtree is at most one different from the height of its left subtree)

- What about if we remove node 21?

Node 11 is unbalanced!



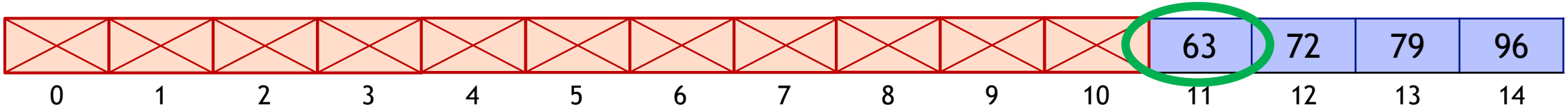
Why is this important?

- The worst-case access time for any node is bound by the tree's height
  - The height of a balanced tree is bound by $O(\log n)$

- To reach any node from the root, the longest the path will be is bound by the height
  - So a tree of height h containing 2^{h+1} nodes means that we can reach a ***lot*** of nodes very quickly

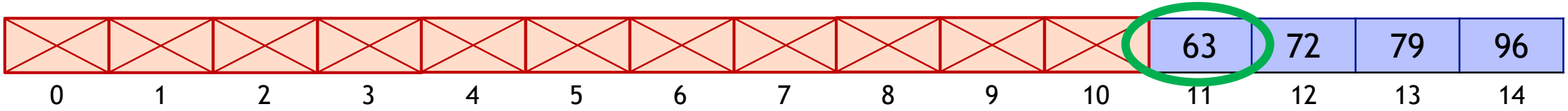
- Let's take a look

A quick comparison

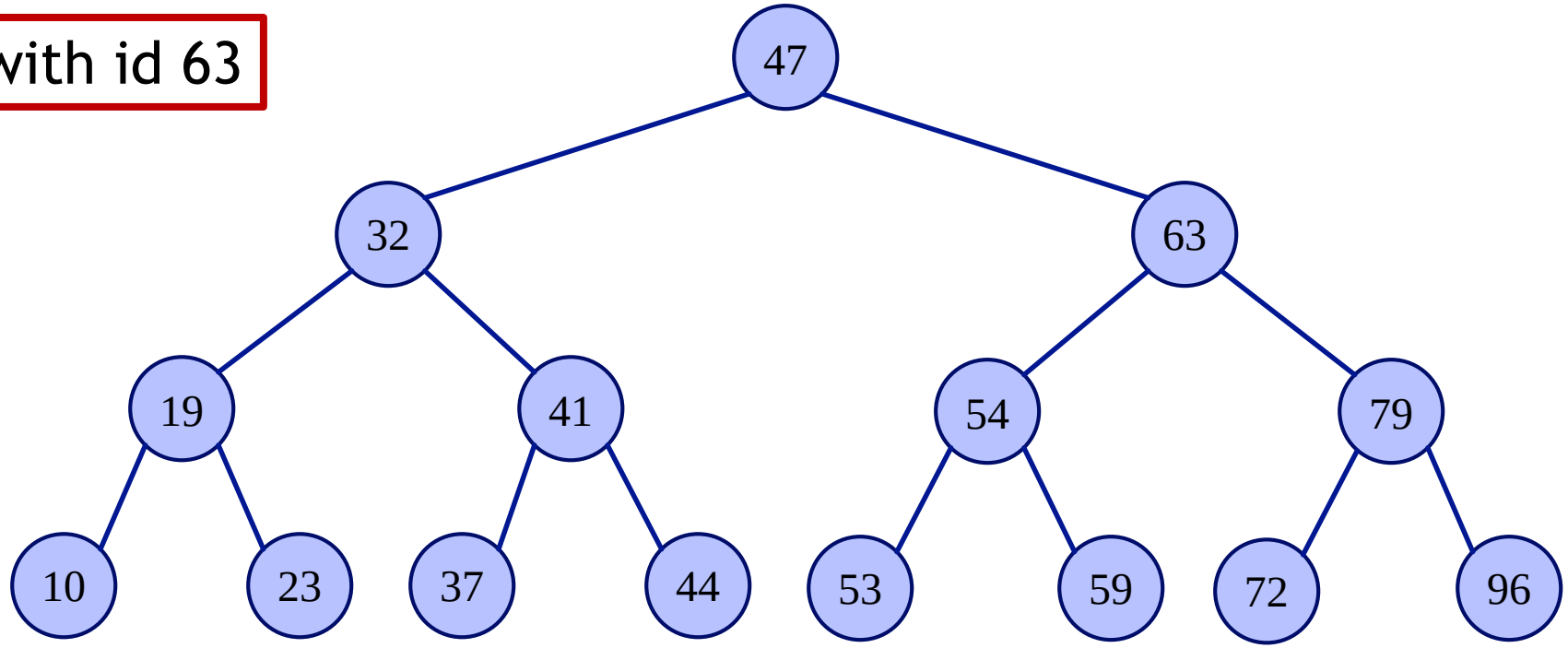


Find item with id 63

A quick comparison



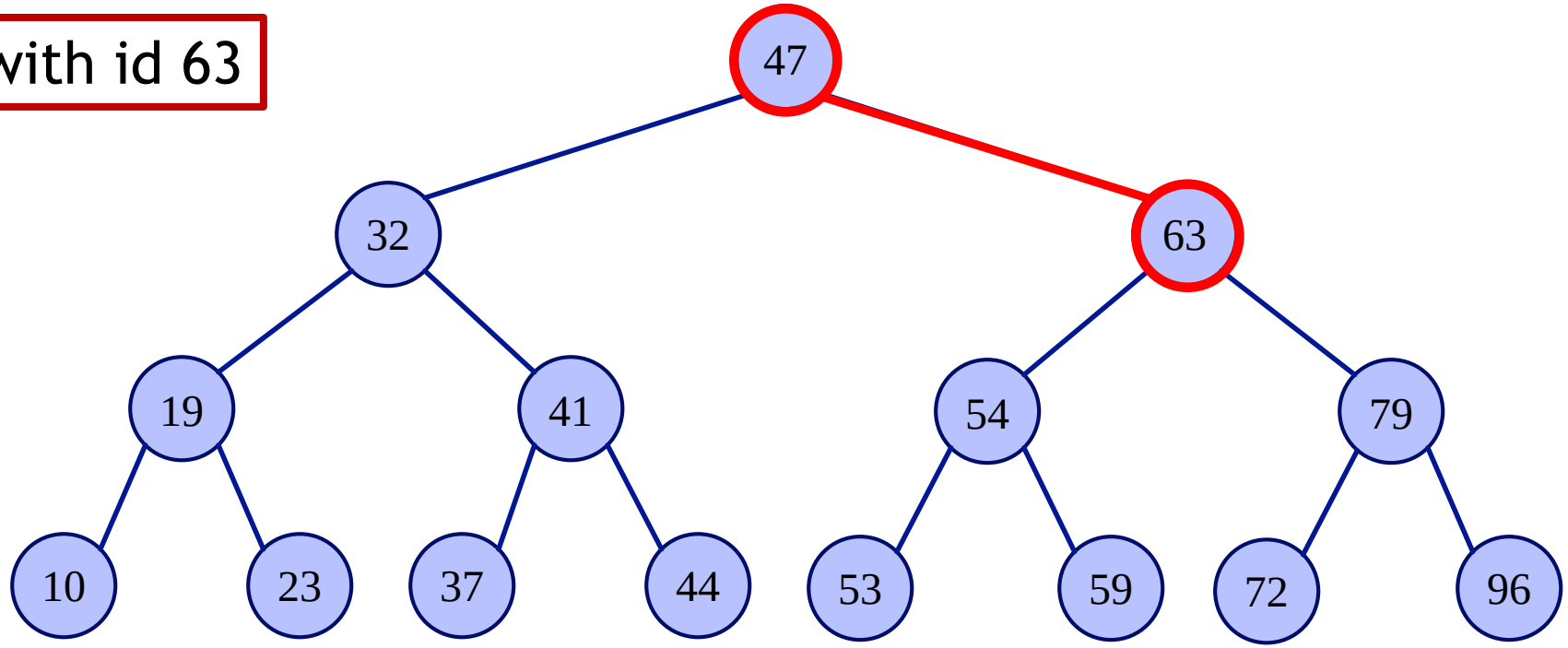
Find item with id 63



A quick comparison

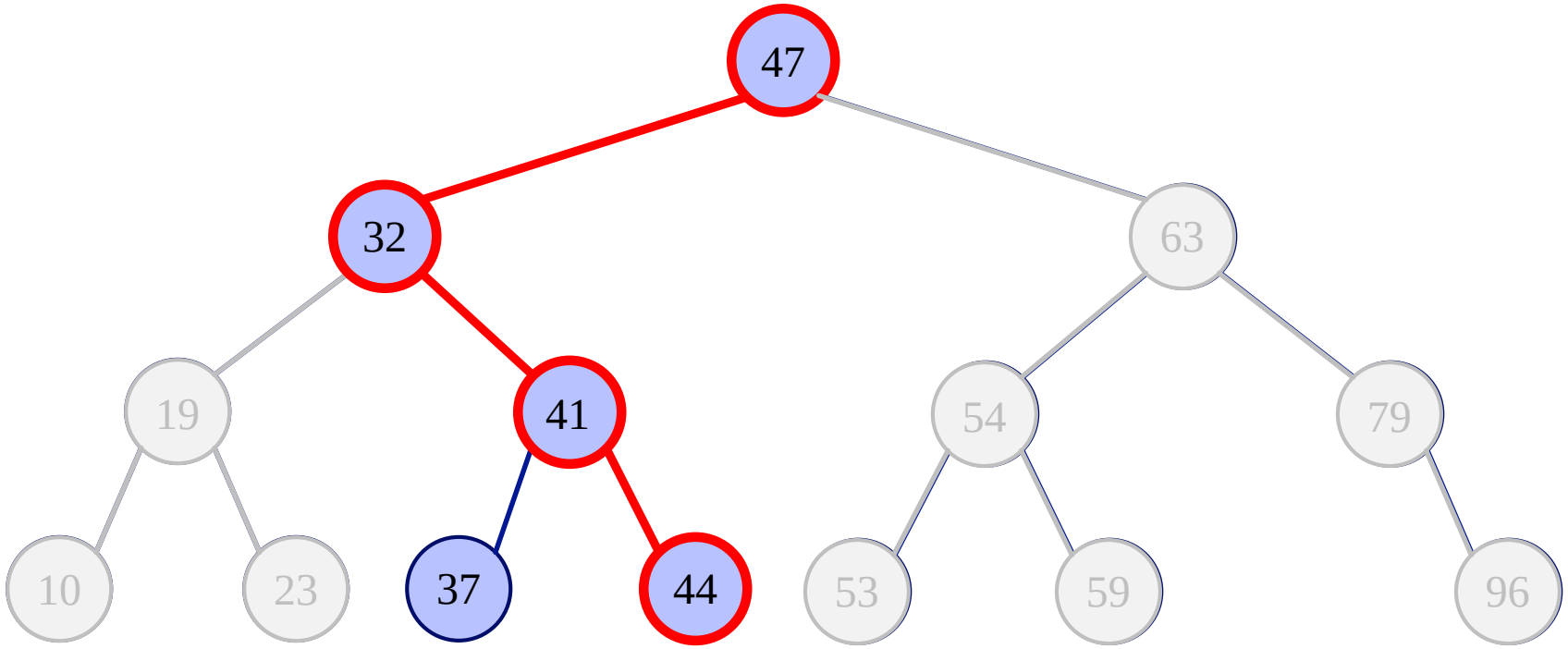
10	19	23	32	37	41	44	47	53	54	59	63	72	79	96
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Find item with id 63



A quick comparison

10	19	23	32	37	41	44	47	53	54	59	63	72	79	96
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

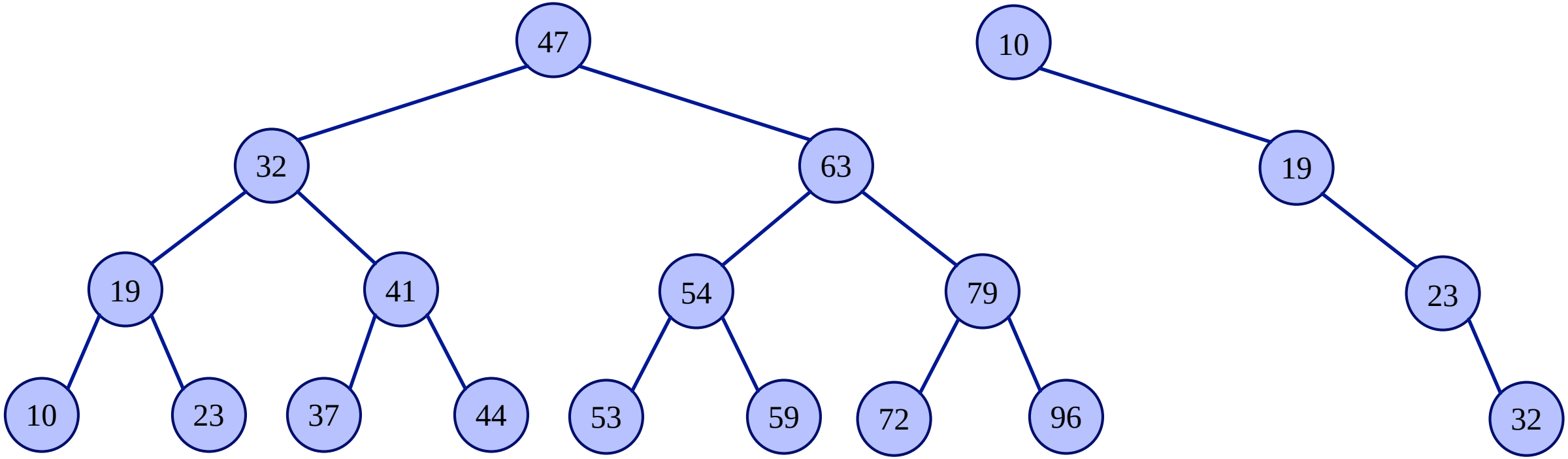


This scales!

- With 15 items, in the worst case we had to visit 4 elements
  - assume we have a balanced Binary Tree containing a larger number of items
- Here are some other sizes, and the worst case searches:
  - 1,000 items: visit at most 10 elements
  - 1,000,000 items: visit at most 20 elements
  - 1,000,000,000 items: visit at most 30 elements
- We can search through *certain* Binary Trees very quickly!

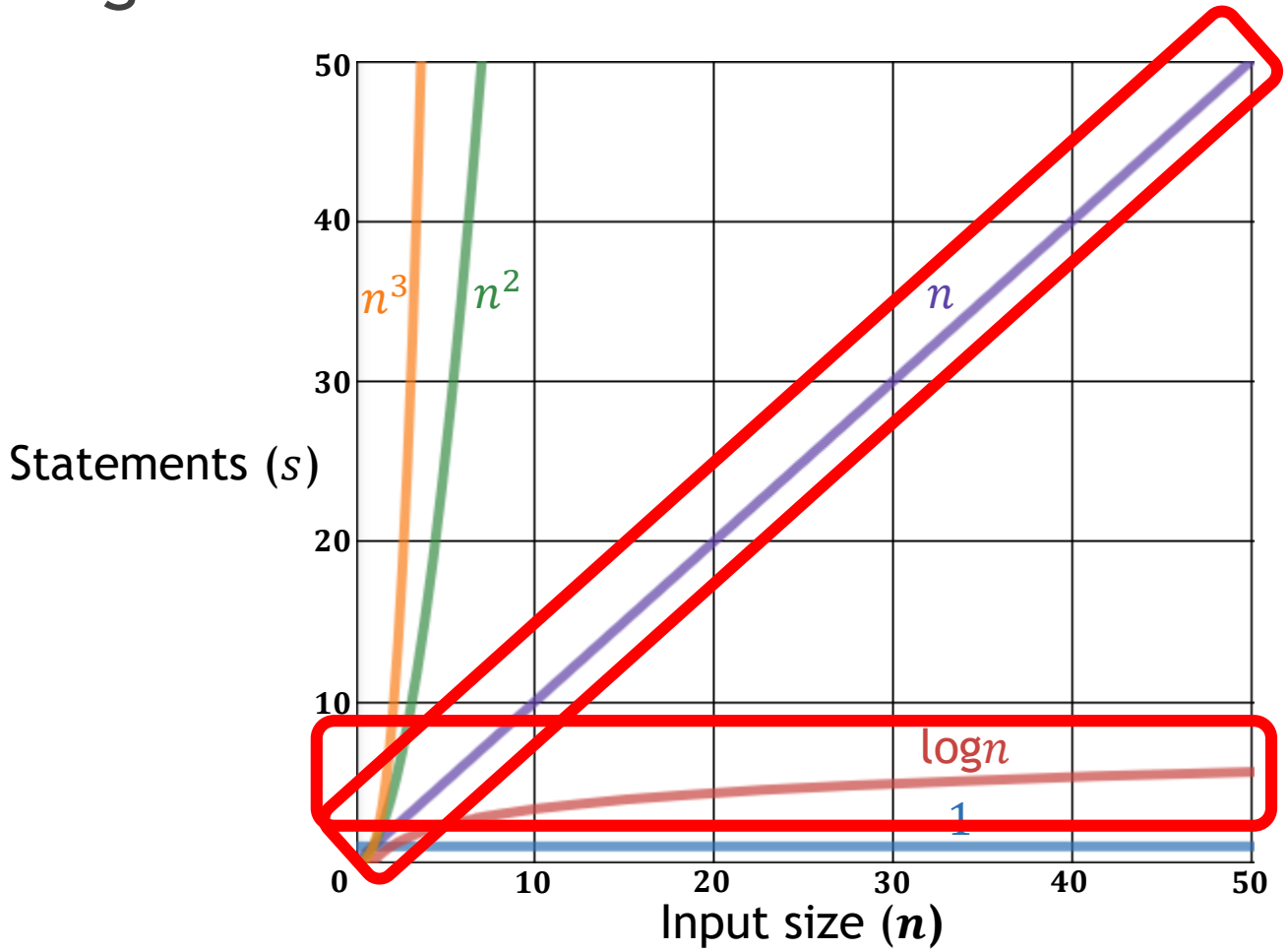
It’s not quite that simple

- I said there are certain types of trees that allow for very fast searches
  - What types of things might negatively affect the efficiency of a Binary Tree?
  - We want our Binary Trees to remain **balanced**!



Asymptotic Analysis

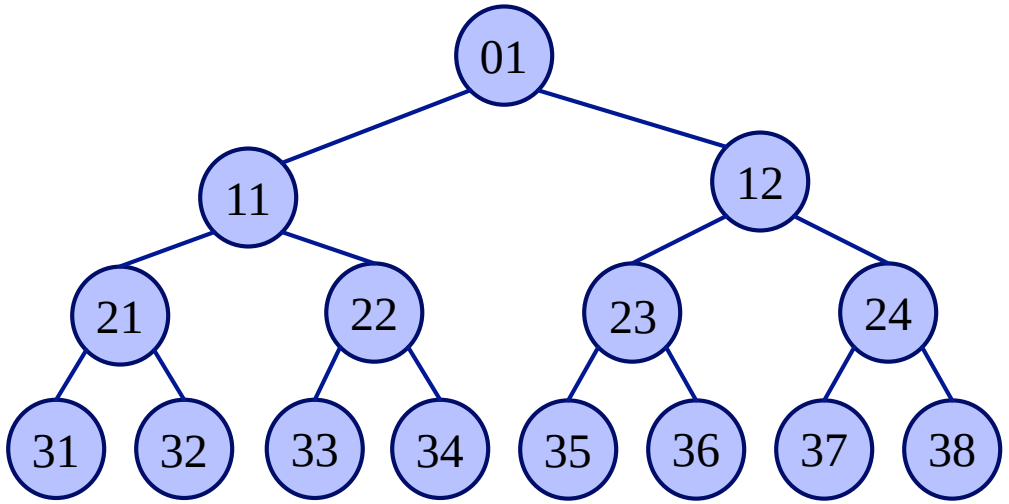
- In general, the algorithms we will see in this course will likely fall under one of the following categories:
  - Constant -- $O(1)$
  - Logarithmic -- $O(log\ n)$
  - Linear -- $O(n)$
  - Quadratic -- $O(n^2)$
  - Cubic -- $O(n^3)$



Graph images created using https://www.desmos.com/

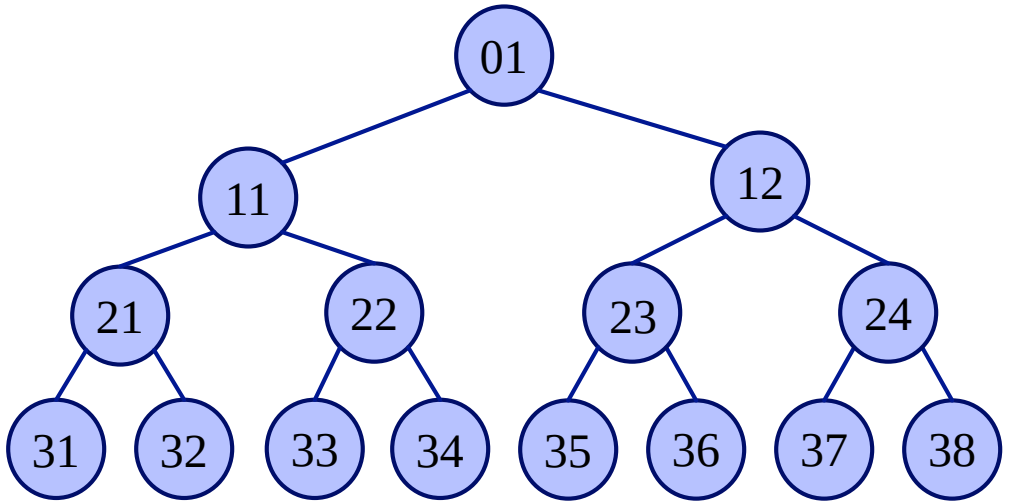
Recap: Binary Tree Classifications

- *Perfect* tree:
  - all nodes on all levels are filled (both *complete* and *balanced*)



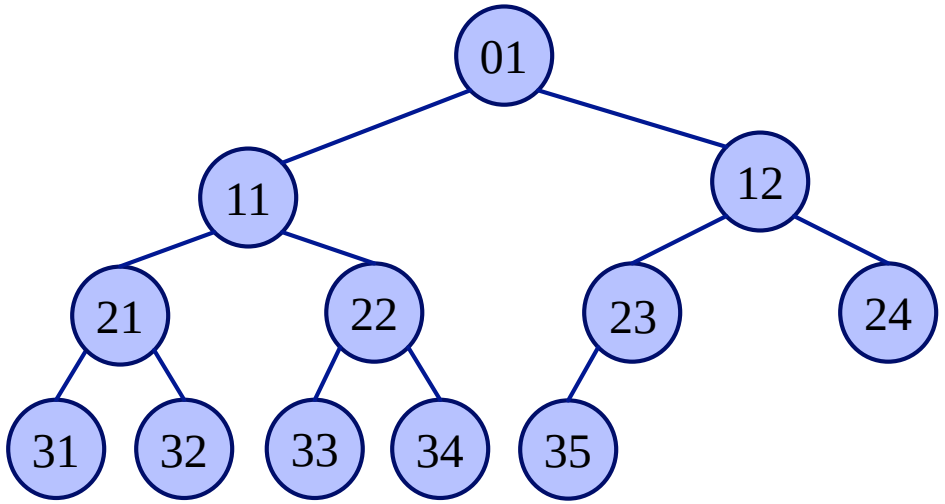
Recap: Binary Tree Classifications

- *Perfect* tree:
  - all nodes on all levels are filled (both *complete* and *balanced*)
- *Complete* tree:
  - all nodes except those on last level are filled (any missing are on the right)



Recap: Binary Tree Classifications

- *Perfect* tree:
  - all nodes on all levels are filled (both *complete* and *balanced*)
- *Complete* tree:
  - all nodes except those on last level are filled (any missing are on the right)
- Balanced tree:
  - No nodes left and right subtree heights differ by more than 1

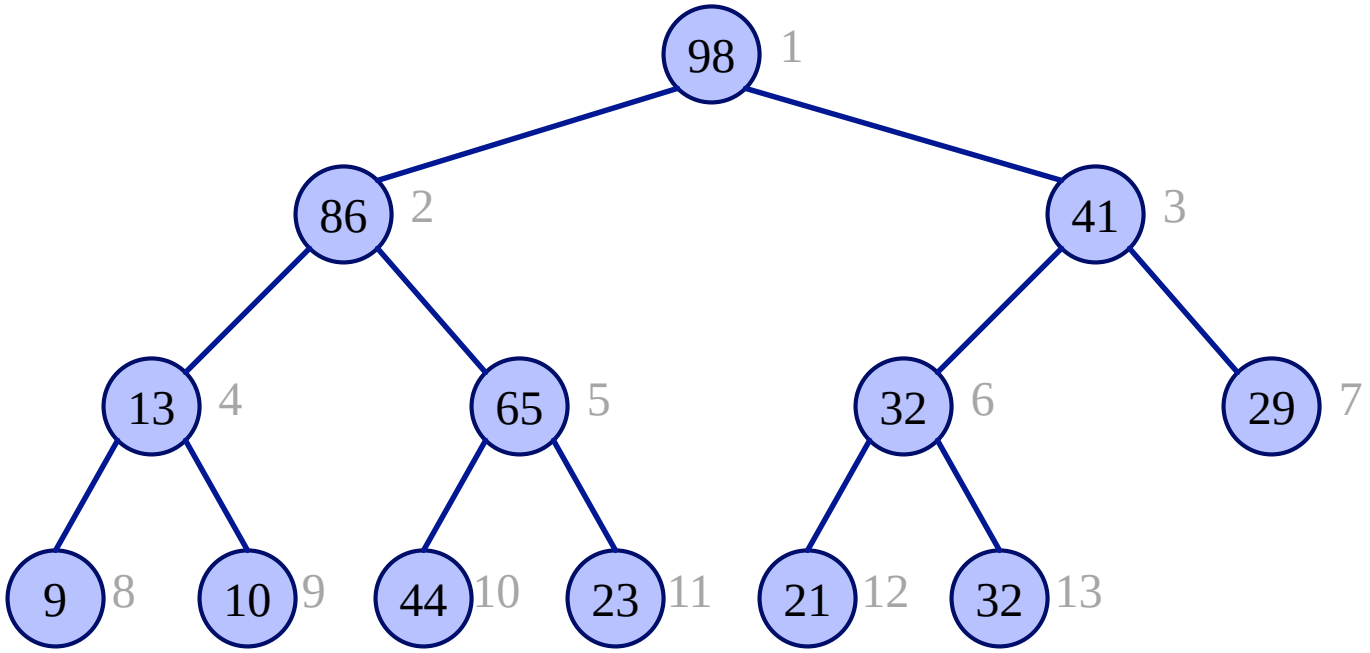


0-based or 1-based array

- When we implement a tree using an array, we can store the values beginning and index 0 or index 1
  - Starting at index 1 wastes part of the array (index 0), but makes it easier to access the indexes of a node's children or parent
  - Starting at index 0 is better for memory utilization, but is a little more complicated to access the indexes of the children or the parent of a node

Tree example using a 1-based array

Tree:

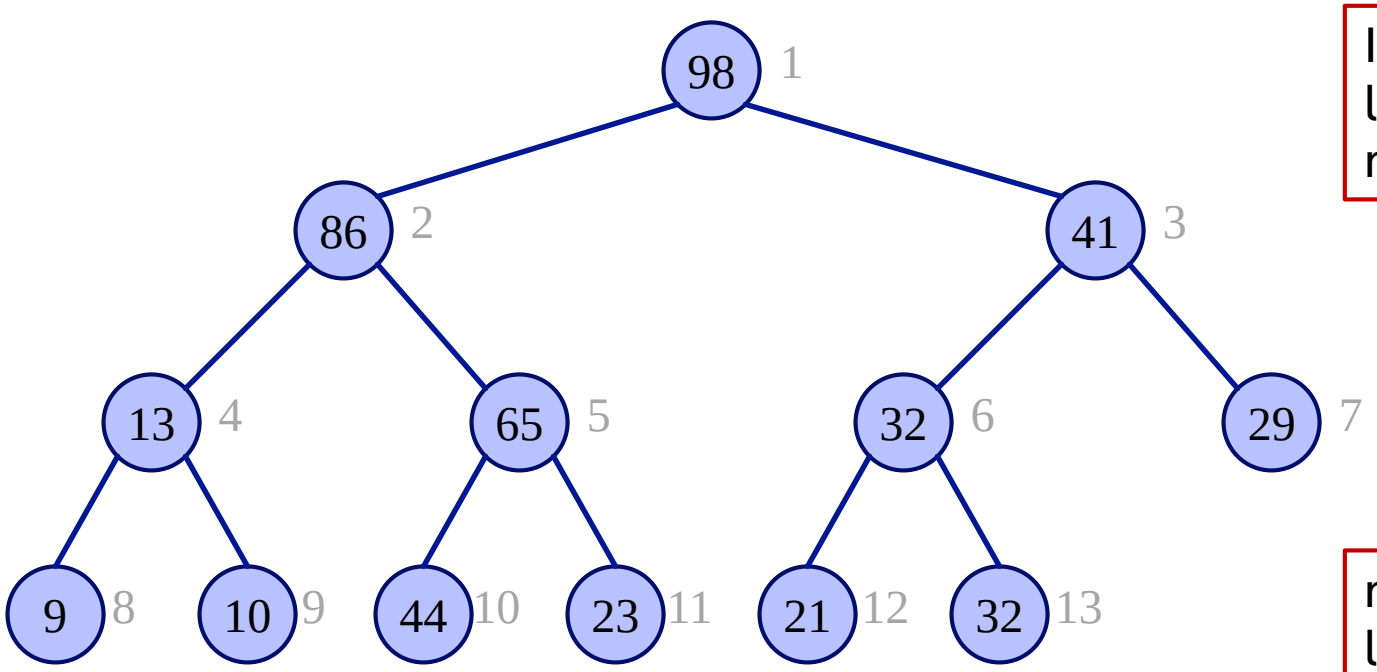


Underlying array:

value	--	98	86	41	13	65	32	29	9	10	44	23	21	32
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Tree example using a 1-based array

Tree:



Index of children?
left: $2(i)$
right: $2(i) + 1$

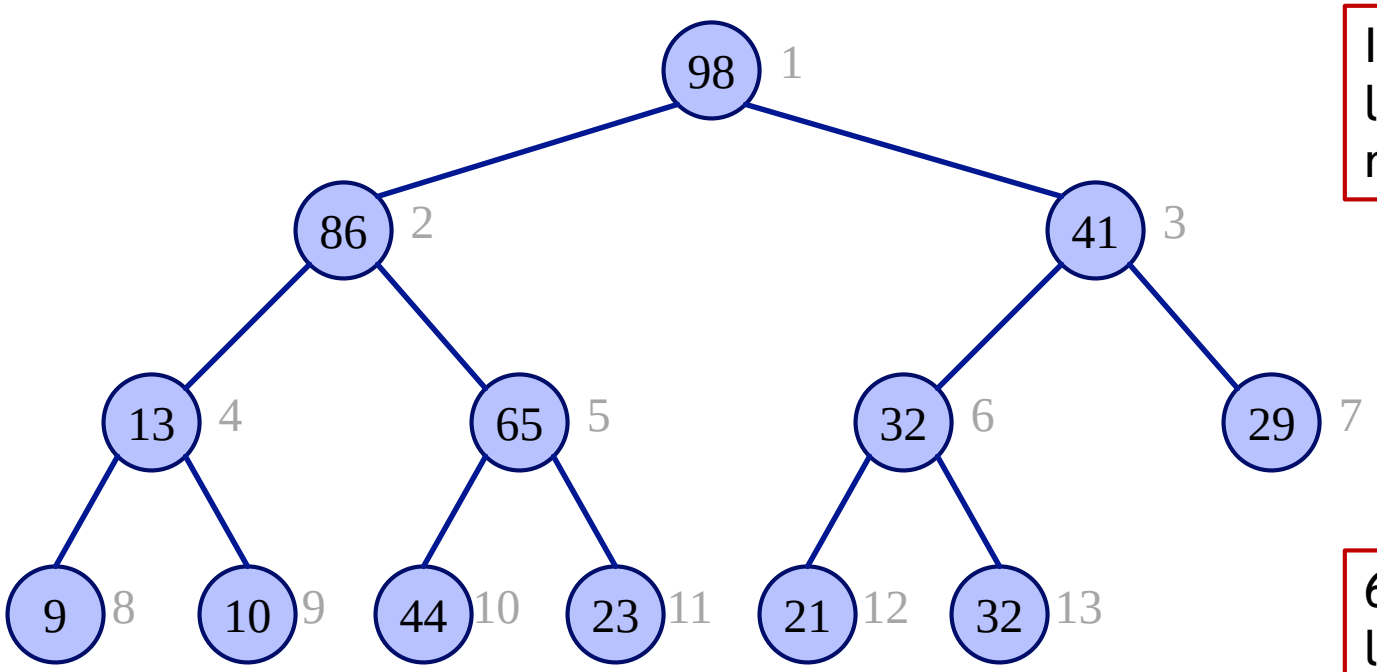
root index: 1
left: $2(1) = 2$
right: $2(1) + 1 = 3$

Underlying array:

value	--	98	86	41	13	65	32	29	9	10	44	23	21	32
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Tree example using a 1-based array

Tree:



Index of children?
left: $2(i)$
right: $2(i) + 1$

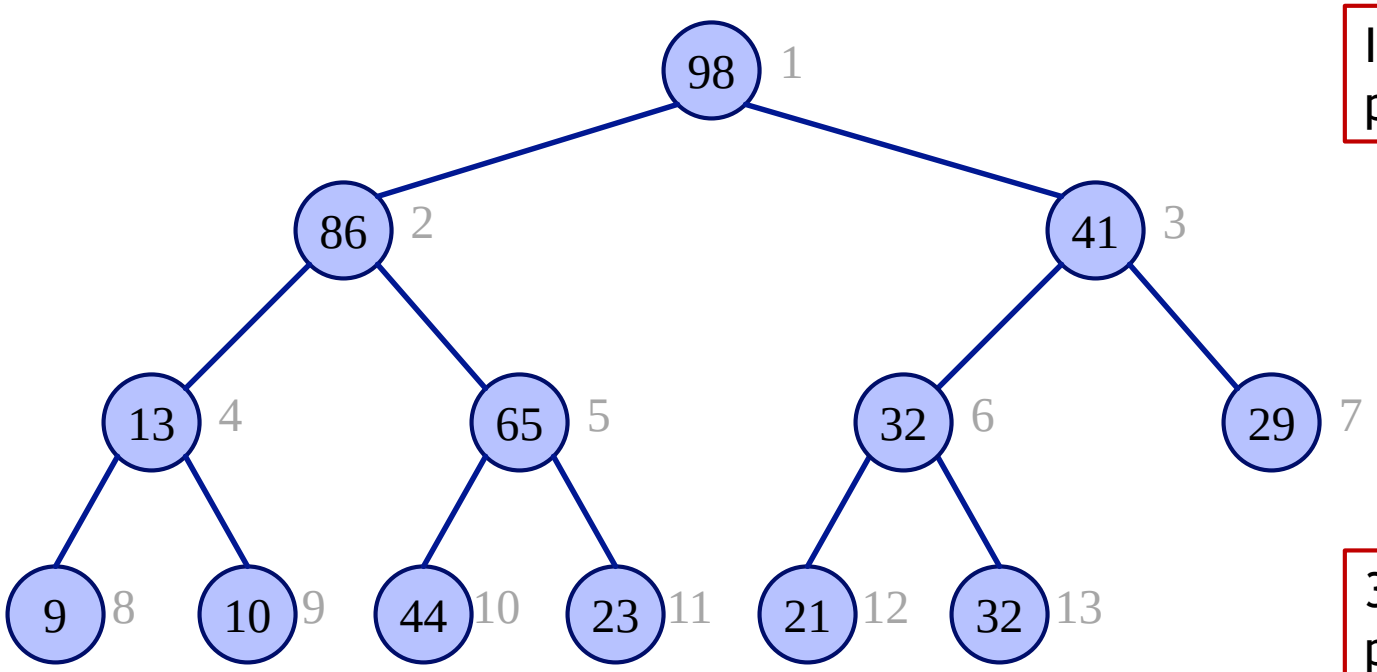
65's index: 5
left: $2(5) = 10$
right: $2(5) + 1 = 11$

Underlying array:

value	--	98	86	41	13	65	32	29	9	10	44	23	21	32
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Tree example using a 1-based array

Tree:



Index of parent?
parent: $\lfloor i/2 \rfloor$

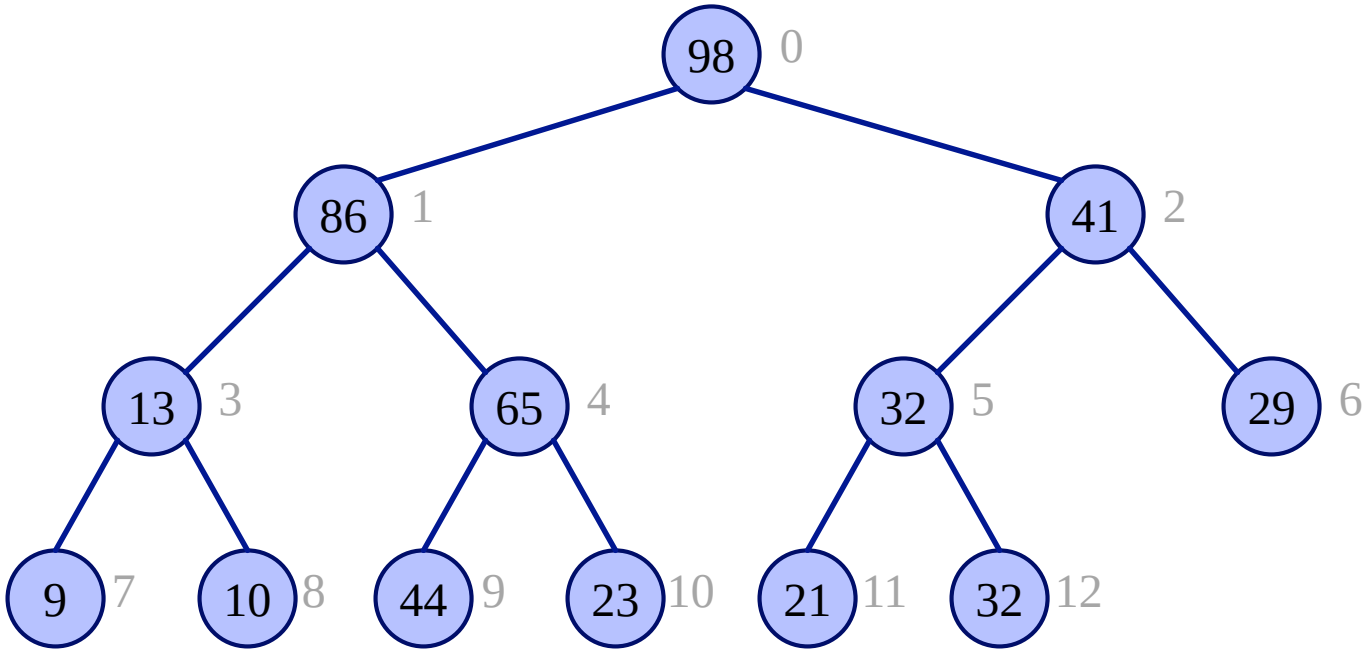
32's index: 6
parent: $6/2 = 3$

Underlying array:

value	--	98	86	41	13	65	32	29	9	10	44	23	21	32
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Tree example using a 0-based array

Tree:

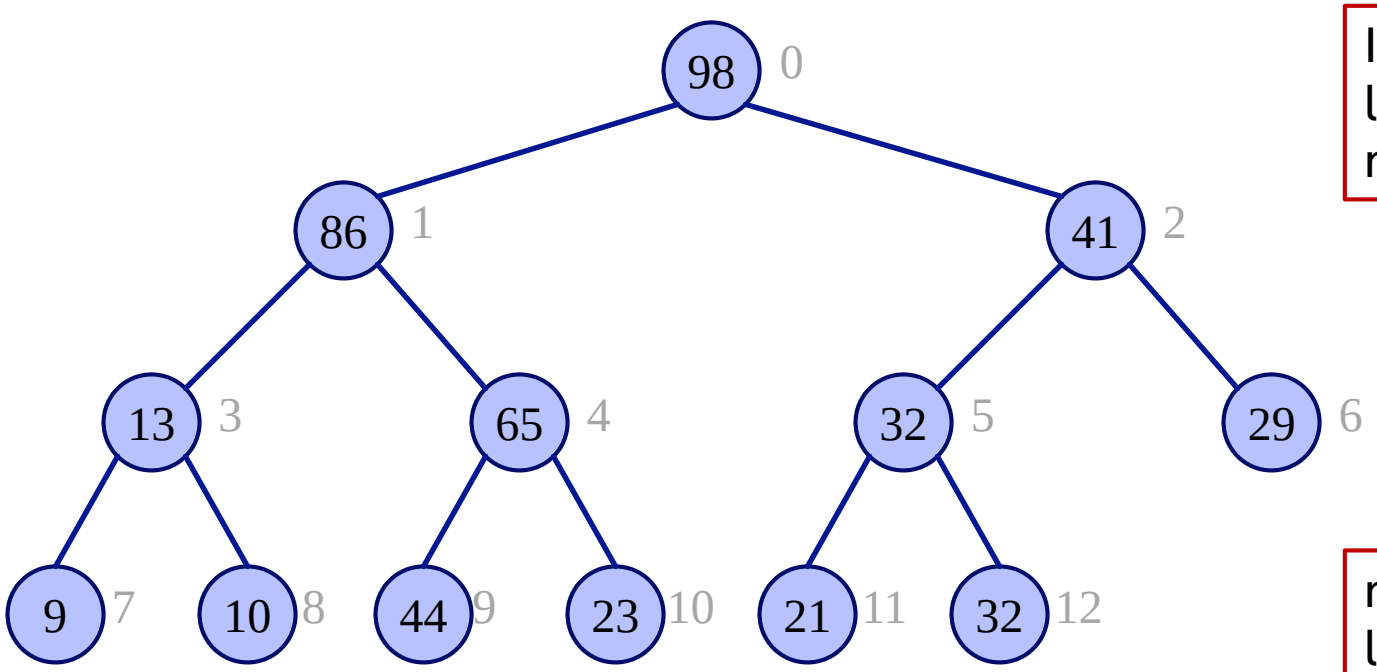


Underlying array:

value	98	86	41	13	65	32	29	9	10	44	23	21	32
index	0	1	2	3	4	5	6	7	8	9	10	11	12

Tree example using a 0-based array

Tree:



Index of children?
left: $2(i) + 1$
right: $2(i) + 2$

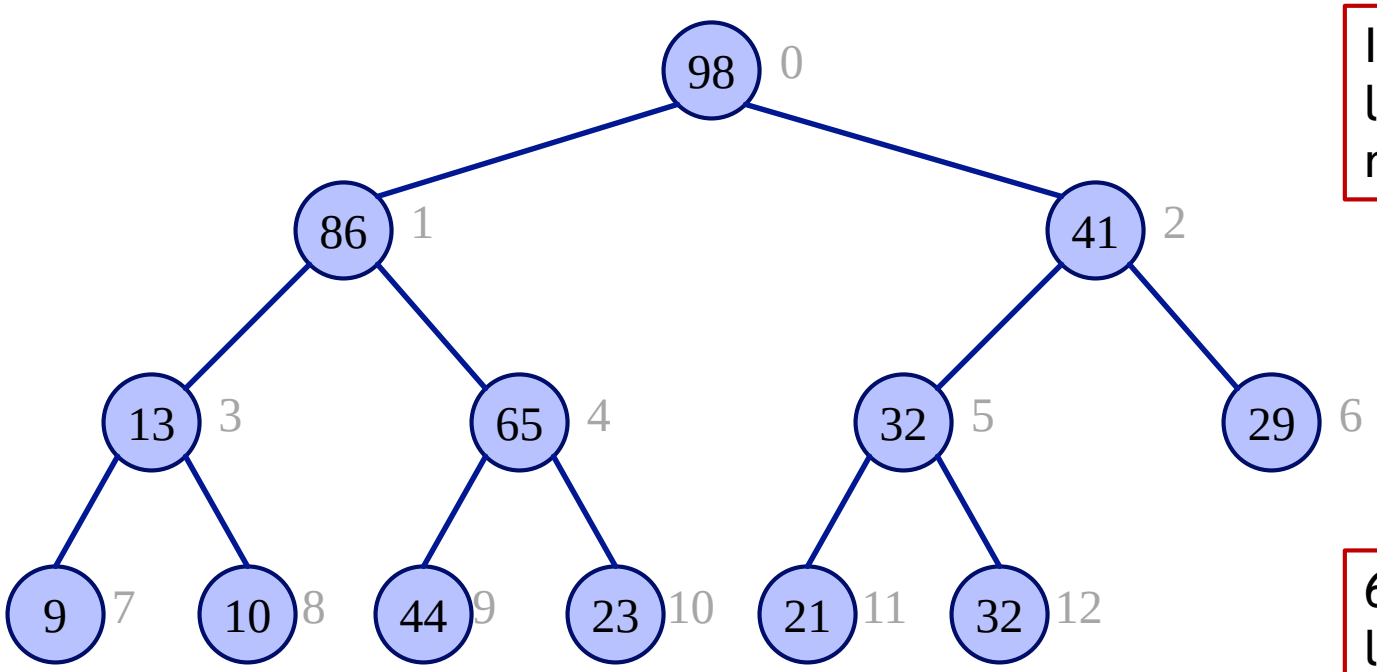
root index: 0
left: $2(0) + 1 = 1$
right: $2(0) + 2 = 2$

Underlying array:

value	98	86	41	13	65	32	29	9	10	44	23	21	32
index	0	1	2	3	4	5	6	7	8	9	10	11	12

Tree example using a 0-based array

Tree:



Index of children?
left: $2(i) + 1$
right: $2(i) + 2$

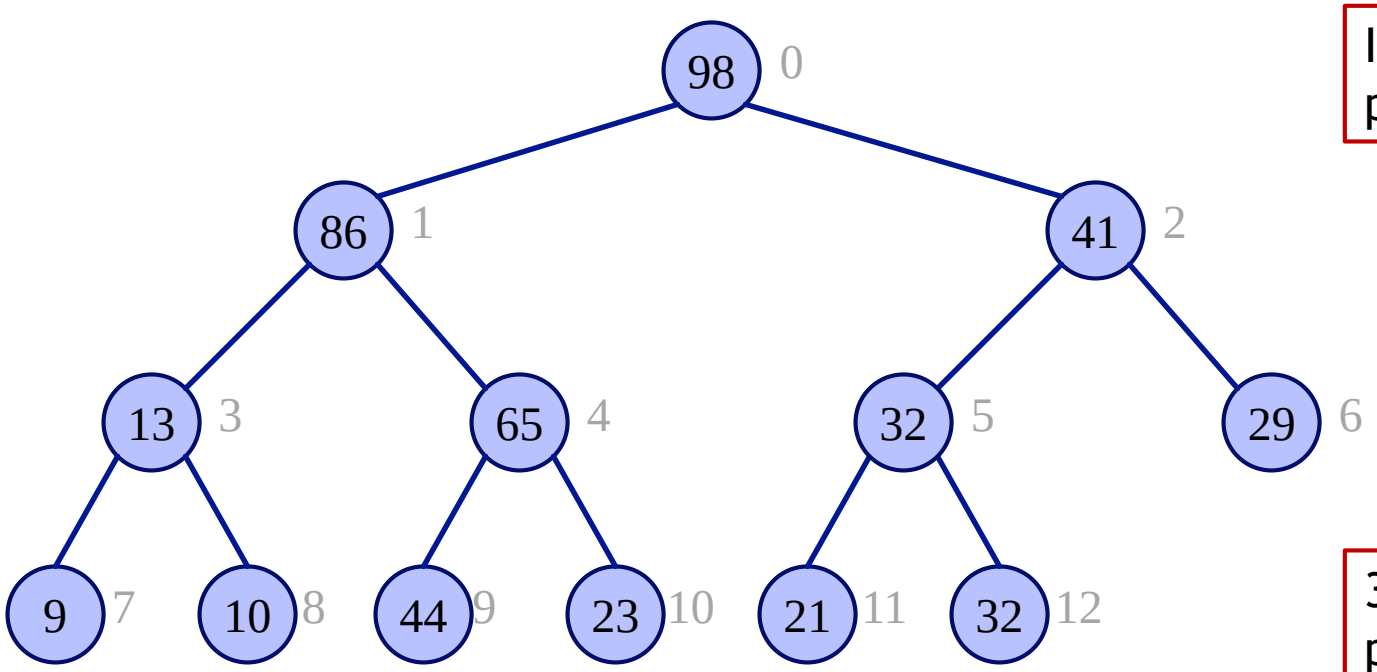
65's index: 4
left: $2(4) + 1 = 9$
right: $2(4) + 2 = 10$

Underlying array:

value	98	86	41	13	65	32	29	9	10	44	23	21	32
index	0	1	2	3	4	5	6	7	8	9	10	11	12

Tree example using a 0-based array

Tree:



Index of parent?
parent: $\lfloor (i-1)/2 \rfloor$

32's index: 5
parent: $\lfloor (5-1)/2 \rfloor = 2$

Underlying array:

value	98	86	41	13	65	32	29	9	10	44	23	21	32
index	0	1	2	3	4	5	6	7	8	9	10	11	12