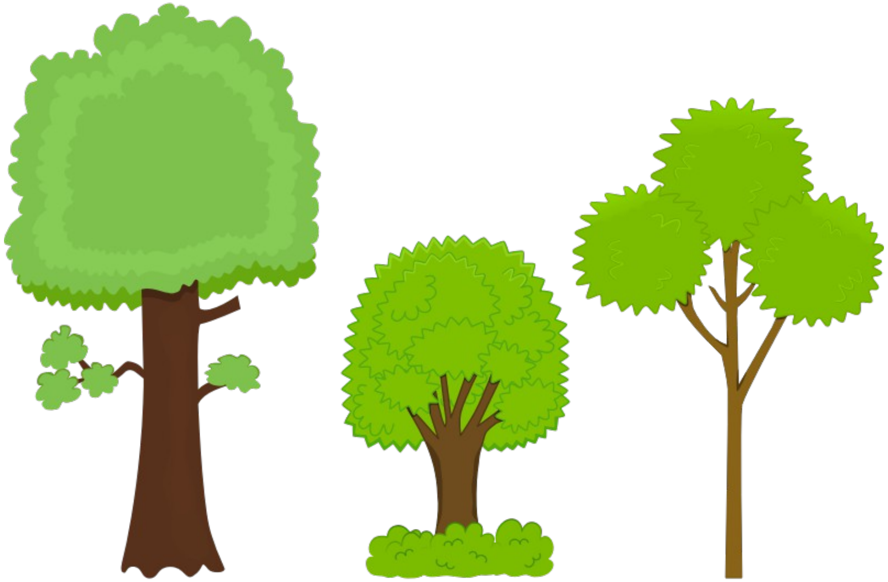
CS240
Fall 2014

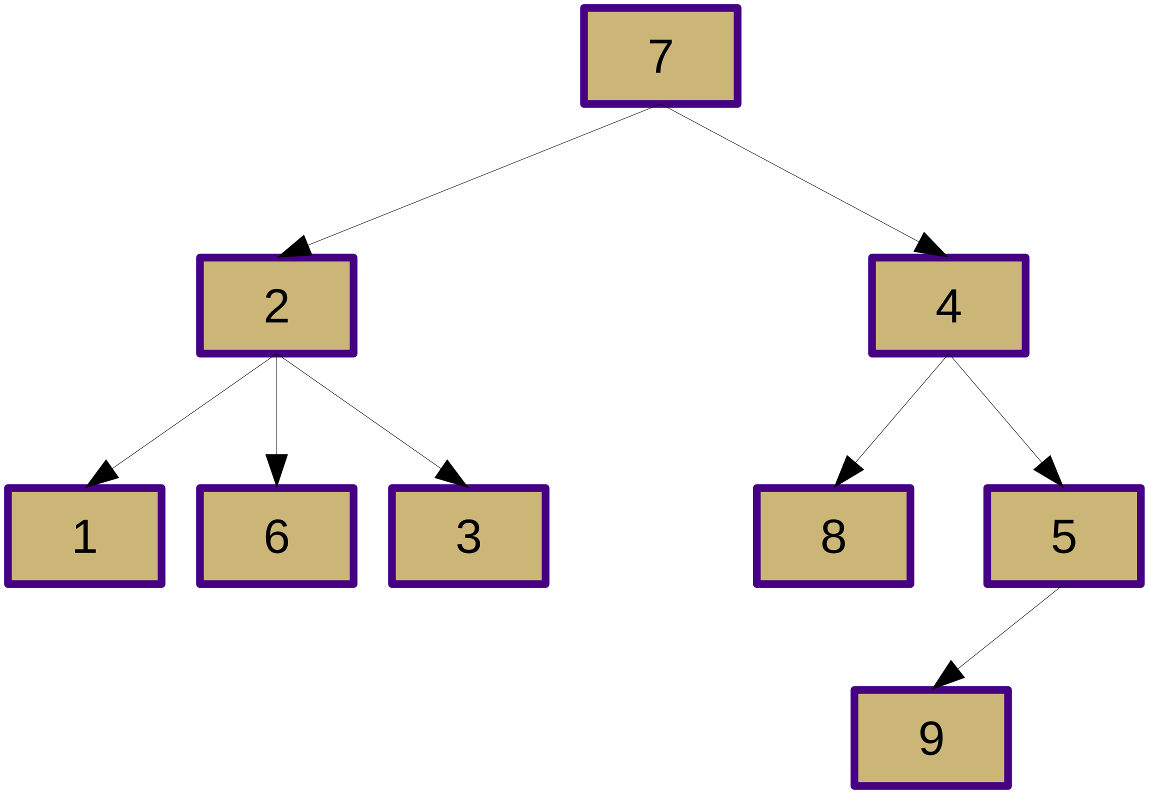
Mike Lam, Professor



Trees

Trees

- Hierarchical data structure

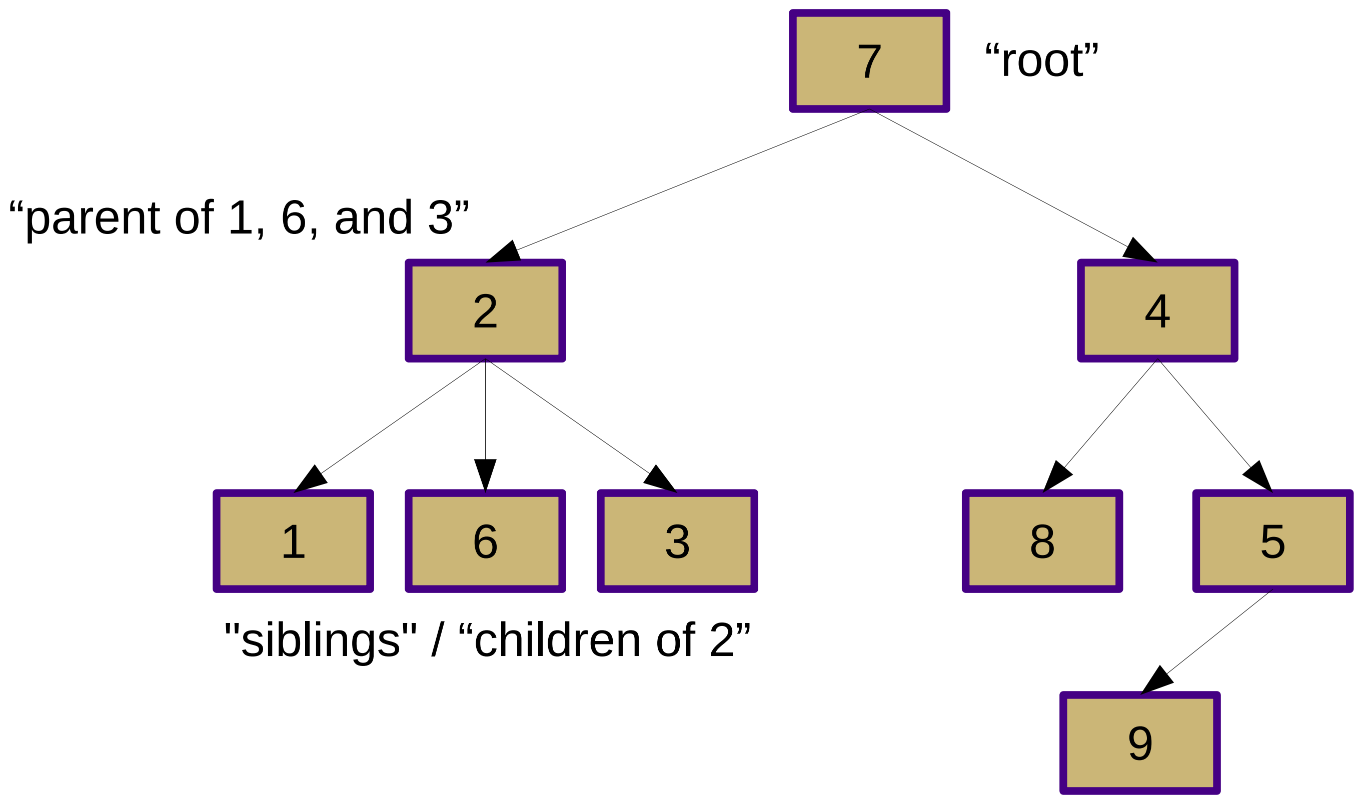


Tree Definitions

- Most generally: "an undirected graph with exactly one path between any pair of nodes"
- Textbook definition: a set of nodes with parent/child relationships
  - The "root" node has no parent
  - Each non-root node has a unique parent node
  - Parent nodes may have multiple children
- General conditions:
  - All nodes are connected
  - There are no cycles
  - Every edge is necessary to maintain connectivity
  - Contains n-1 edges for n nodes

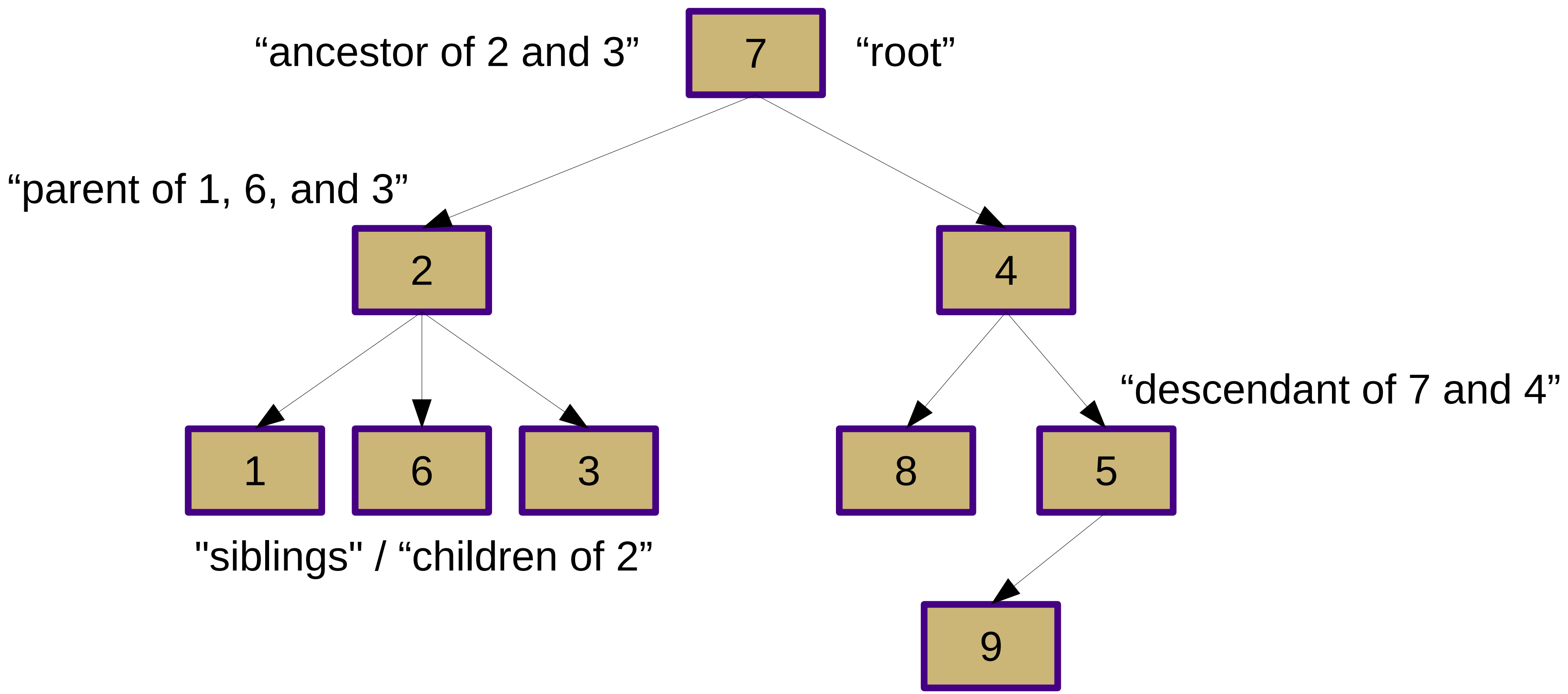
Trees

- Hierarchical data structure



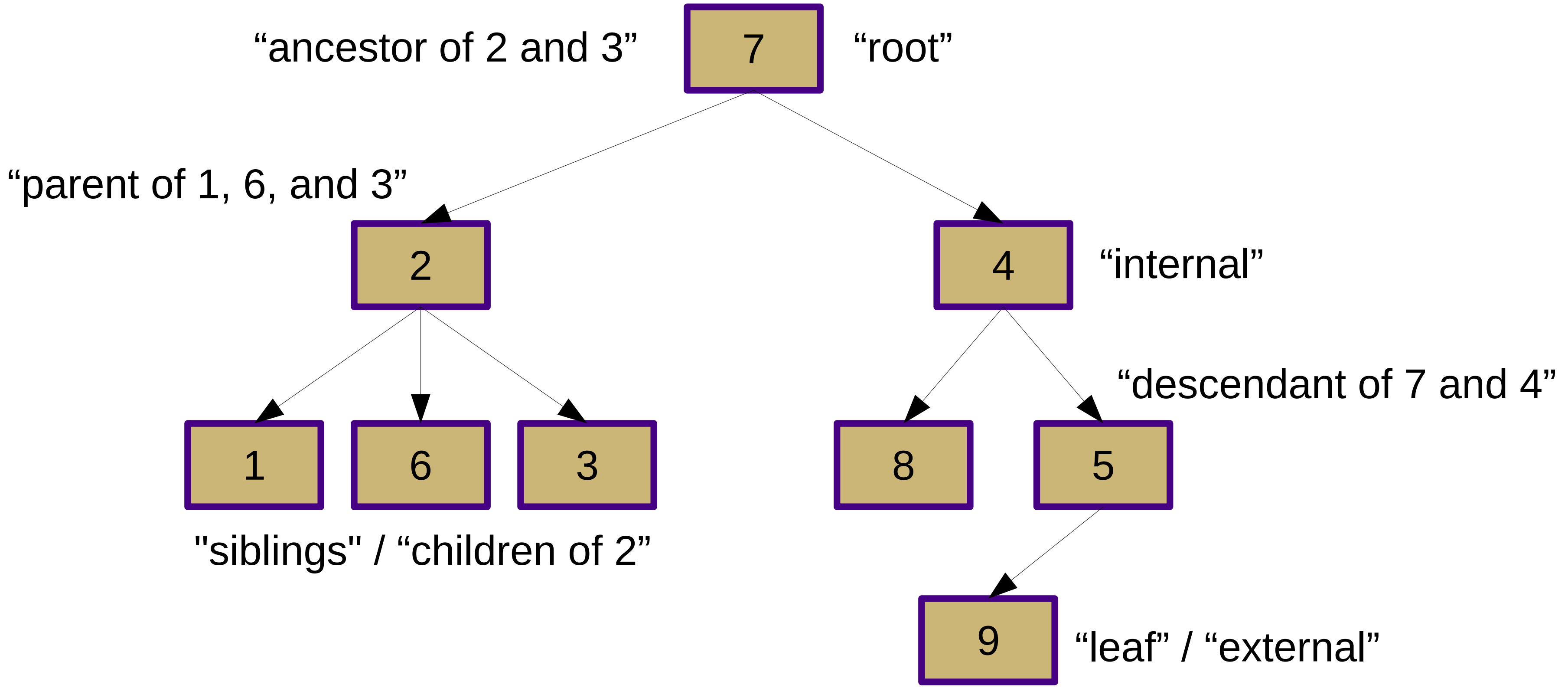
Trees

- Hierarchical data structure



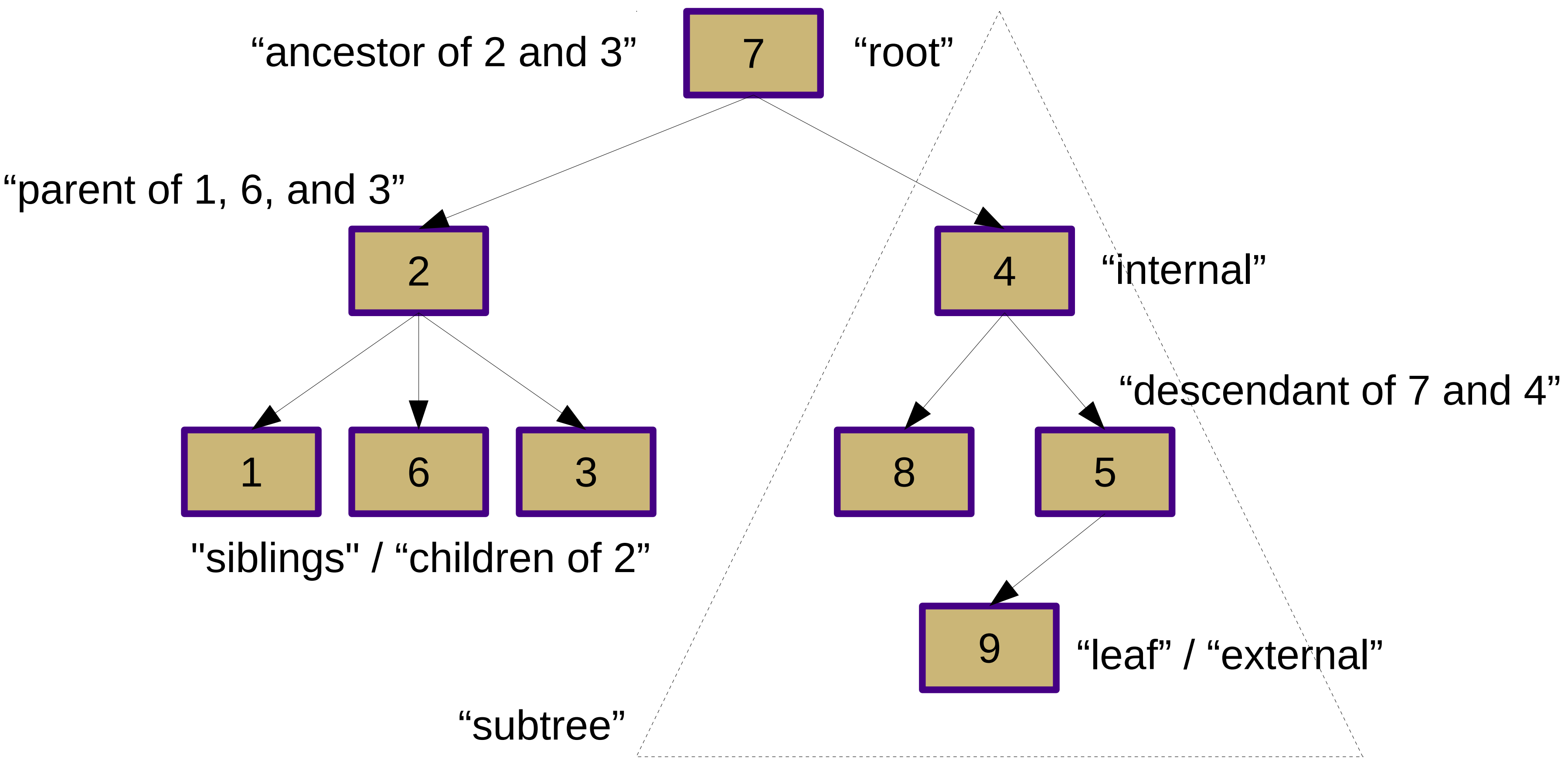
Trees

- Hierarchical data structure



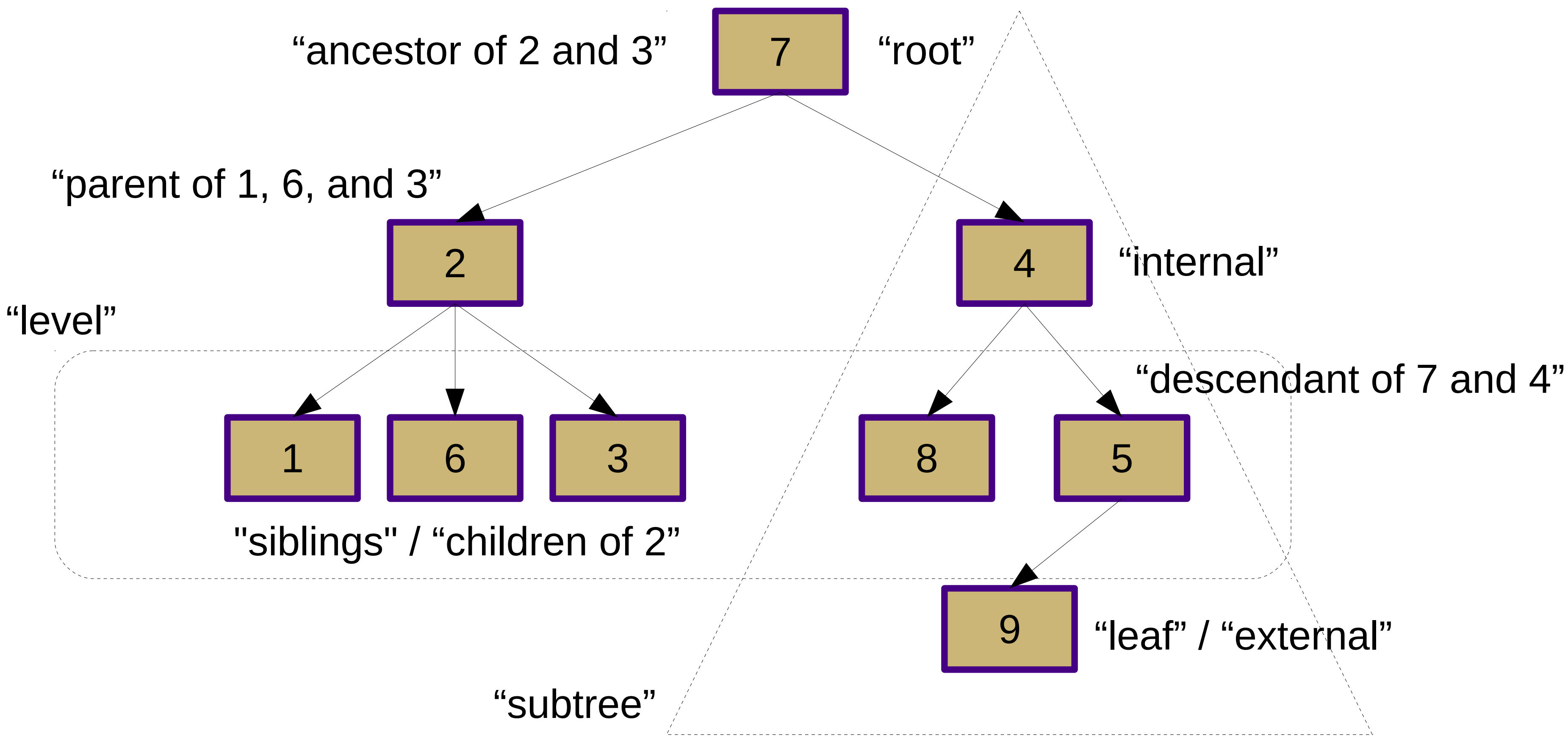
Trees

- Hierarchical data structure



Trees

- Hierarchical data structure

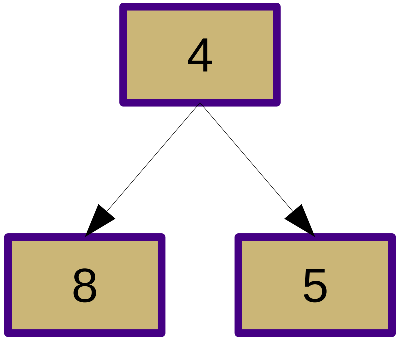


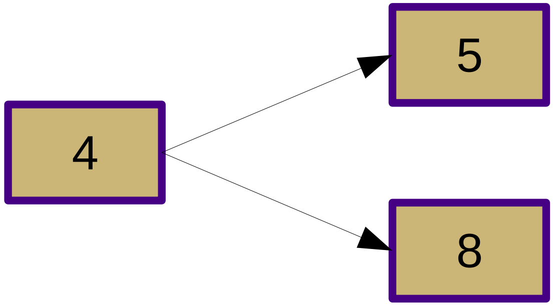
Tree Definitions

- *Parent*: directly "above" in the hierarchy
- *Child*: directly "below" in the hierarchy
- *Ancestor*: "above" in the hierarchy
- *Descendant*: "below" in the hierarchy
- *Sibling:* child of the same parent
- *External / Leaf*: no children
- *Internal*: one or more children
- *Level*: all nodes with the same depth
- *Subtree*: a child node and all of its descendants

Tree Visualization

- "Above" and "below" refer to hierarchical relationships
- Trees can be visualized in any orientation/direction
  - Top-down is the most common
  - Left-right is also occasionally useful
  - Natural trees are bottom-up





Tree Definitions

- N-ary tree: each node has at most N children
  - 2-ary trees are called "binary trees"
    - Left and right subtrees
    - "Full" or "proper" if all nodes have either zero or two children
  - 3-ary trees are called "ternary trees"
    - Left, middle, and right subtrees
- Ordered tree: meaningful linear relationship among children of each node
  - Visualized with left-to-right arrangement of siblings
  - We will exploit ordered binary trees for fast searching

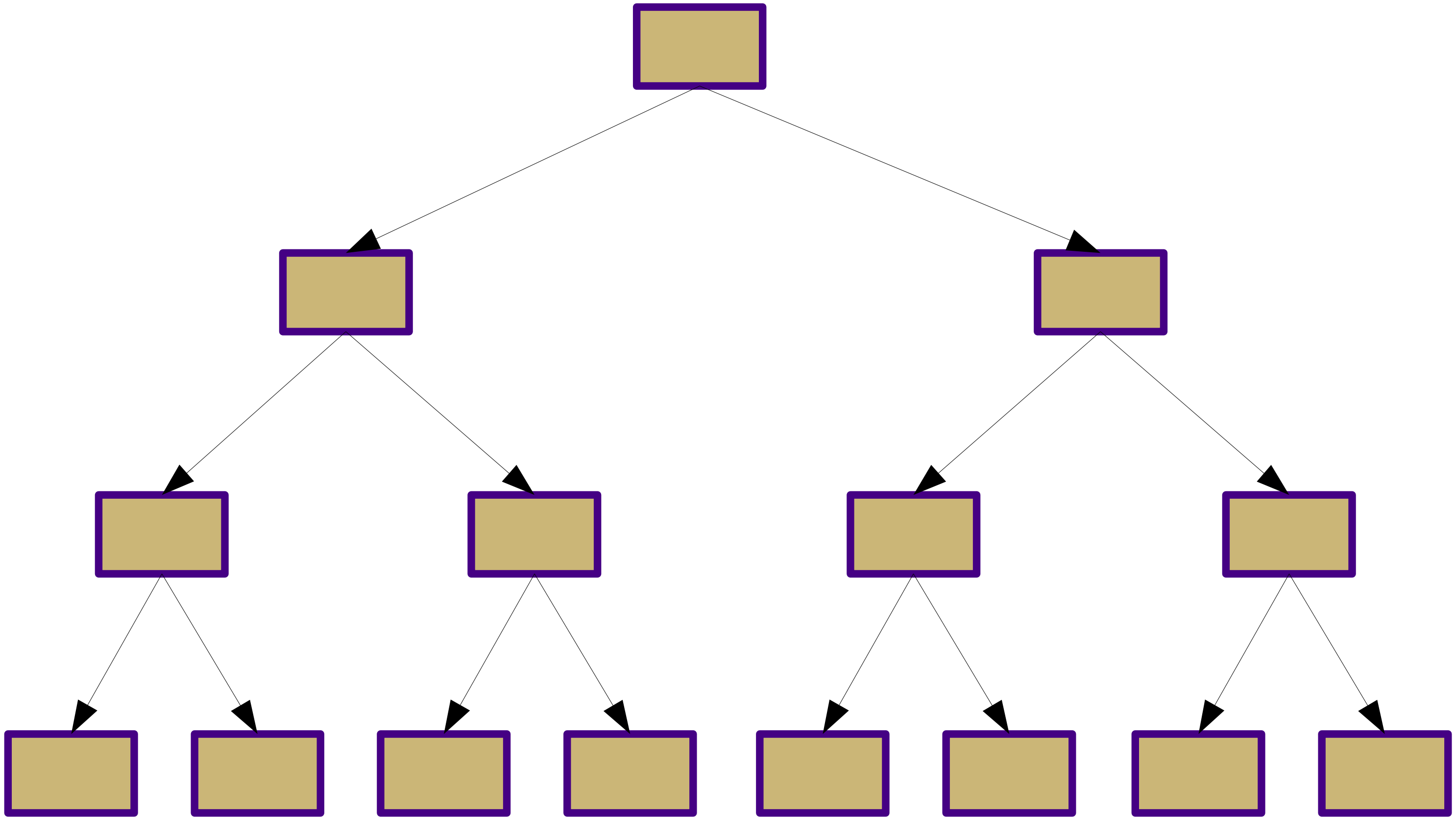
Depth vs. Height

- Node depth (textbook definition):
  - *depth(root) = 0*
  - *depth(p) = 1+depth(p.parent)*
  - Top-down definition
  - Informally: number of ancestors
- Node height (textbook definition):
  - *height(leaf) = 0*
  - *height(p) = 1 + max([height(c) for c in p.children])*
  - Bottom-up definition
  - Informally: number of edges to lowest descendant

Caveat

- Textbook definition of tree height:
  - The height of a tree with a single node is zero
  - In general, the height of a tree is the maximum leaf depth
- This is similar to our skip list definition of height
  - A skip list with a single sentinel node had a height of zero
- **Intuition**: the height of a tree is equal to the number of edges between the root and the lowest leaf

Binary Tree Height



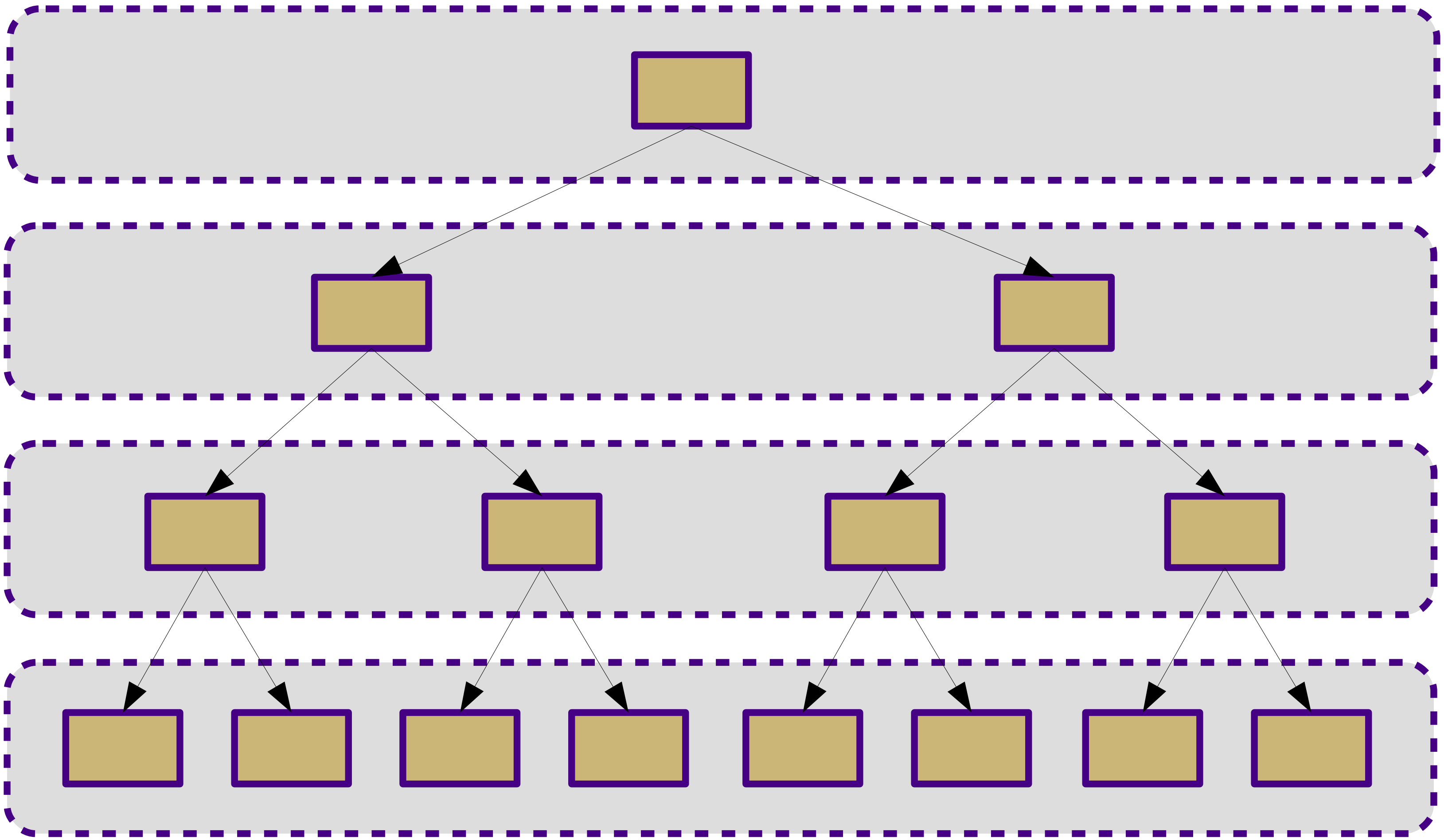
Binary Tree Height

Level 0
Nodes:

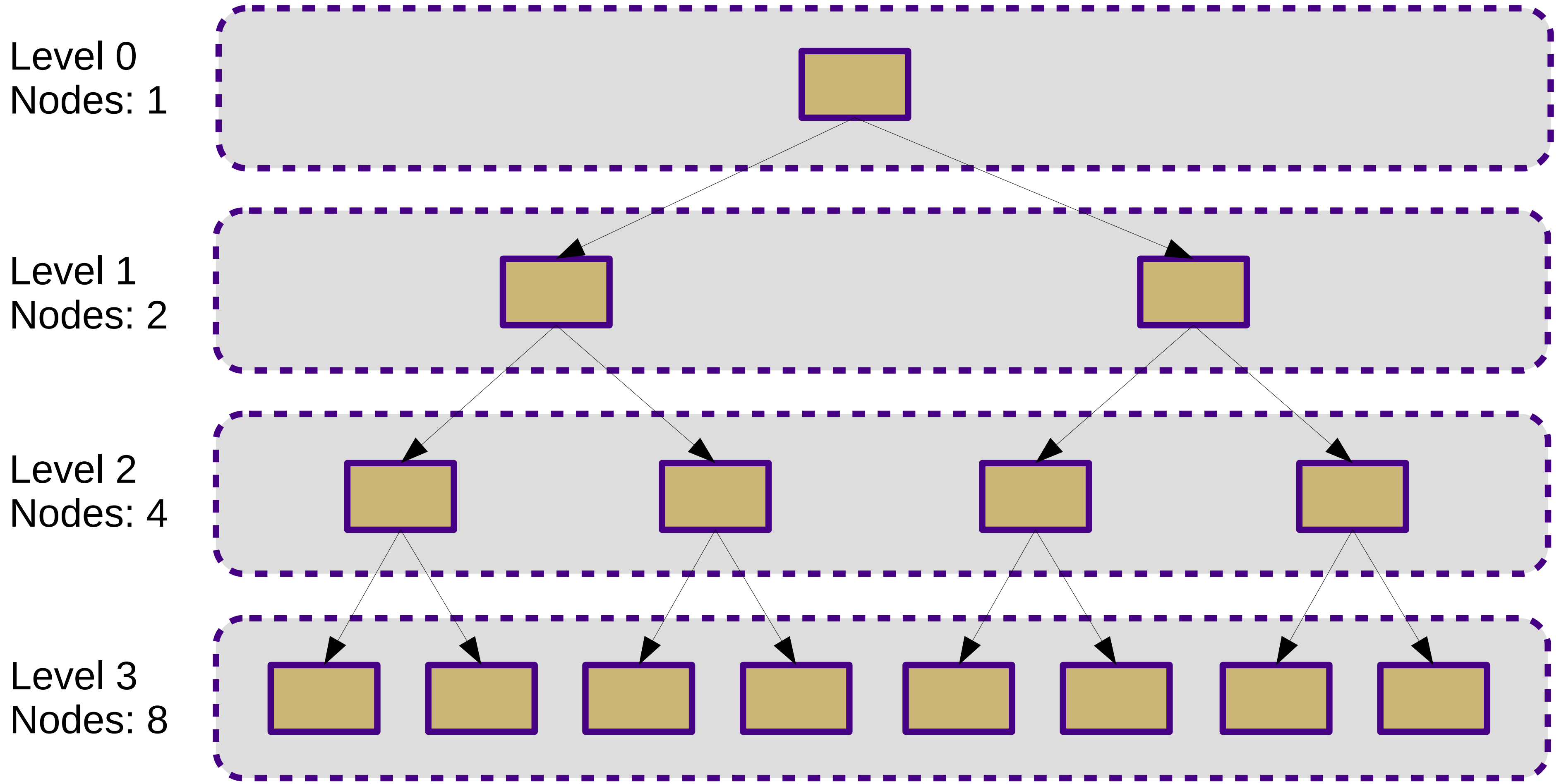
Level 1
Nodes:

Level 2
Nodes:

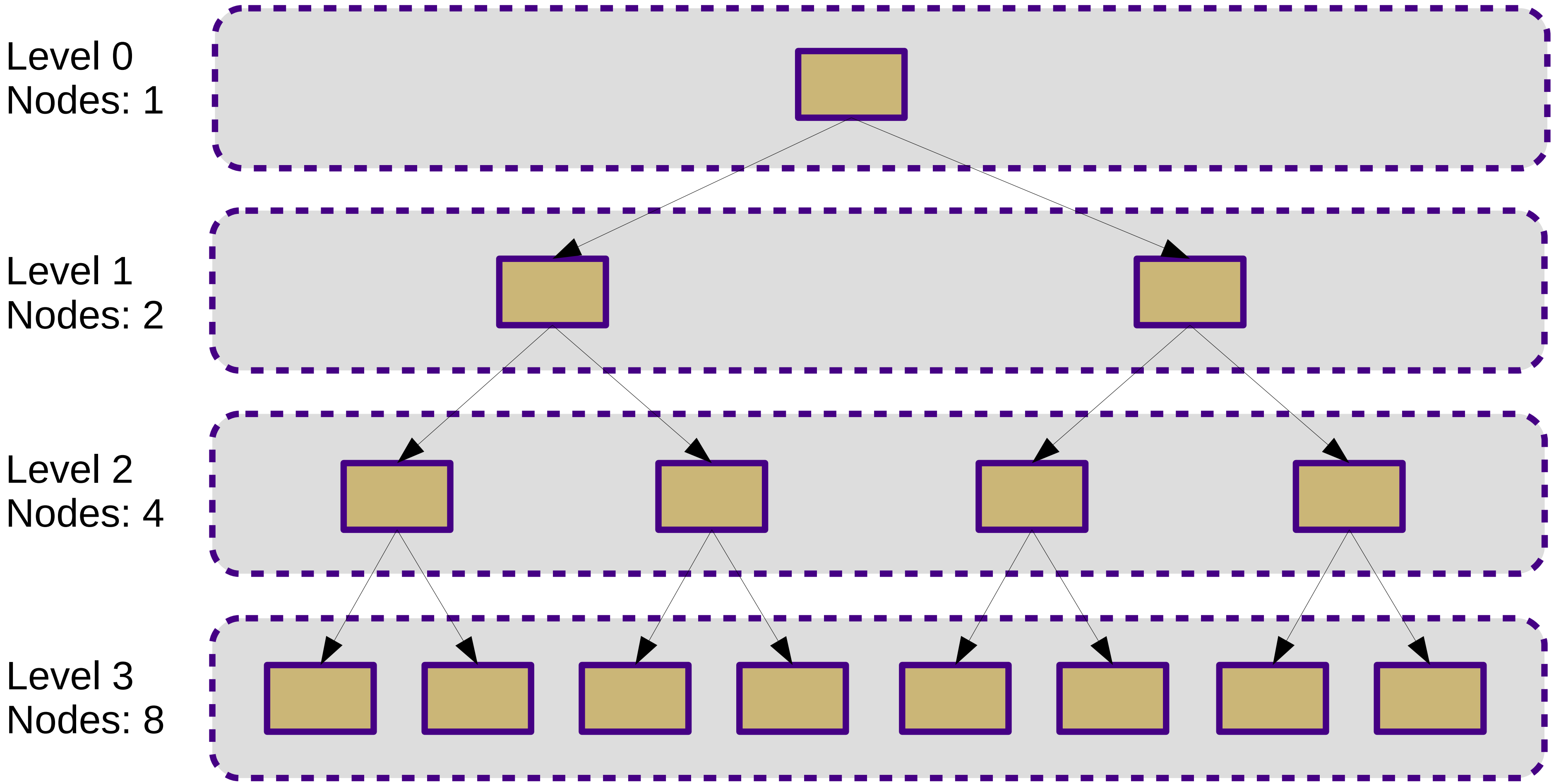
Level 3
Nodes:



Binary Tree Height



Binary Tree Height



In general, level *d* has at most 2^d nodes

Max # of nodes in a binary tree with height *h* is $2^{n+1}-1$

Binary Tree Height

- Key observation: the number of nodes grows exponentially as the height increases
  - Alternatively: the height grows logarithmically as the number of nodes increases:

$$h(t) \in O(\log n(t))$$

- This should lead to $O(\log n)$ or $O(n \log n)$ operations for tree-based structures
  - But only if we can exploit some kind of hierarchical structure in the data

Tree Implementation

- Similar to linked or skip lists
- Node object
  - Reference to data element
  - References to children
    - Alternatively: references to subtrees
    - If binary: "left" and "right"
  - Optional: reference to parent
- Tree object
  - Reference to root node
  - Could track # of nodes and/or tree height

Tree Implementation

- Space usage: $O(n)$
- Non-mutating operations:
  - is_empty: $O(1)$
  - height: $O(n)$
  - depth(p): $O(d_p+1)$
- Mutating operations:
  - insert (given location): $O(1)$
  - delete (given location): $O(1)$

Tree Implementation

- Textbook uses a "Position" wrapper for tree nodes
  - This is a generalization of the "iterator" concept
  - Also sometimes called "cursors"
- Textbook includes several layers of implementation
  - Tree
  - BinaryTree
  - LinkedBinaryTree
- Both of these are good ideas
  - But they are overly complicated for the concepts we wish to explore in this class
  - We will mostly use our own (simpler) implementations

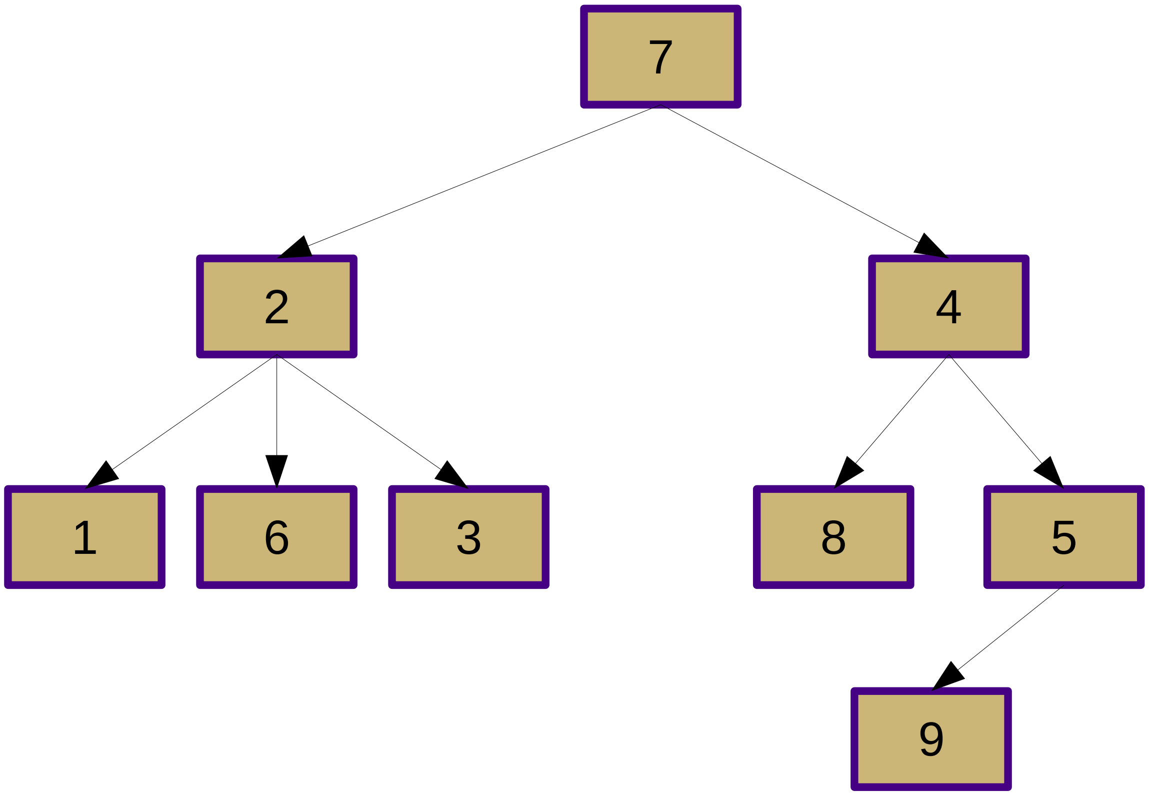
Tree Implementation

```
class BinaryTree:
    """ Represents a simple binary tree. """

    class _Node:
        """ Internal node representation. """

        def __init__(self, value, left=None, right=None):
            """ Create a node with a given value and
                optional subtrees.
            """
            self.element = value
            self.left = left
            self.right = right

    def __init__(self, root):
        """ Create a tree with the given root node. """
        self._root = root
```

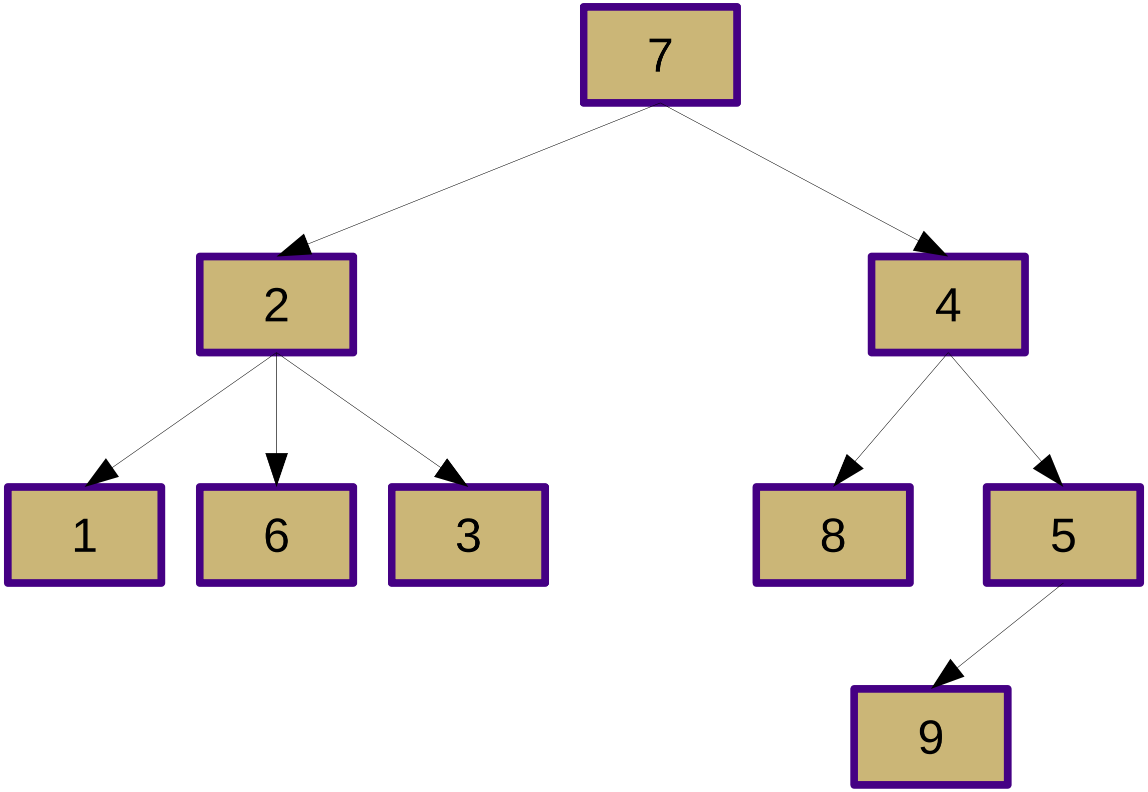
Tree Traversal

- Textbook uses Positions
  - We will just write traversal routines
- Preorder
  - Process parent first, then children
- Postorder
  - Process children first, then parent
- Inorder (binary trees only)
  - Process left child, then parent, then right child
- Breadth-first
  - Process each level of the tree in order

Tree Traversal



Tree Traversal



Preorder: 7, 2, 1, 6, 3, 4, 8, 5, 9
Postorder: 1, 6, 3, 2, 8, 9, 5, 4, 7
Breadth-first: 7, 2, 4, 1, 6, 3, 8, 5, 9

Recursive Traversal

- Recursive traversals
  - Preorder, postorder, and inorder
  - Process current node and children
  - The only difference is ordering
- Non-recursive traversal
  - Breadth-first
  - Use a queue to keep track of unprocessed nodes