Binary Trees

Outline

- Tree Concepts
  - Trees
  - Binary Trees
  - Implementation of a Binary Tree
- Tree Traversals – Depth First
  - Preorder
  - Inorder
  - Postorder
- Breadth First Tree Traversal
- Binary Search Trees

Trees

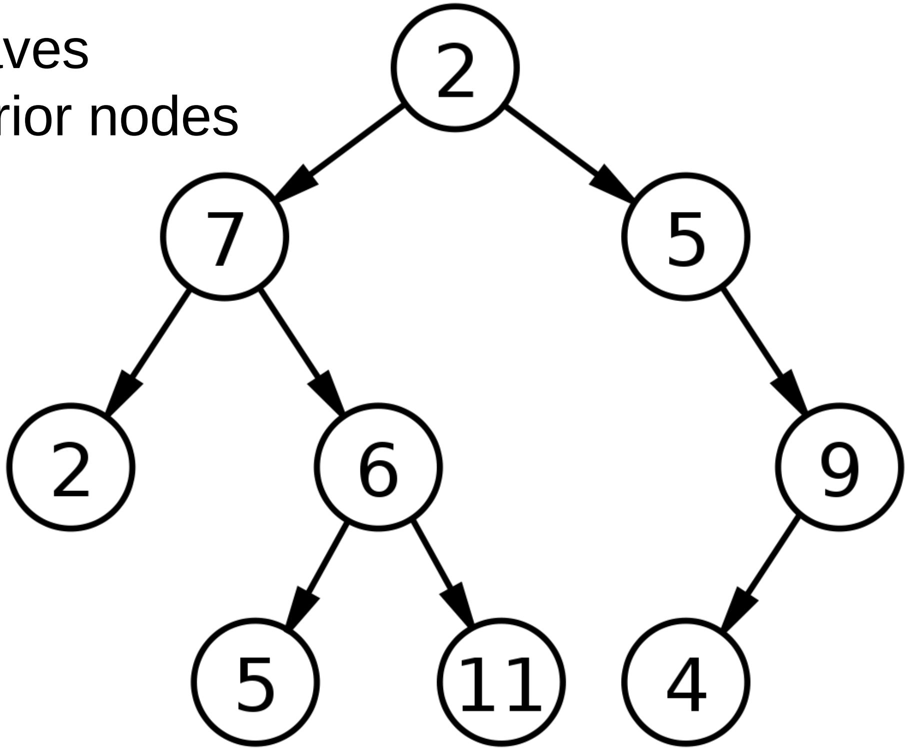
- Another Abstract Data Type
- Heavily used for indexing data points
- Data structure made of nodes and pointers
- Much like a linked list
  - The difference between the two is how they are organized.
- A linked list represents a linear structure
  - A predecessor/successor relationship between the nodes of the list
- A **tree** represents a **hierarchical relationship** between the nodes (ancestral relationship)
  - A node in a tree can have several successors, which we refer to as children
  - A nodes predecessor would be its parent

Trees

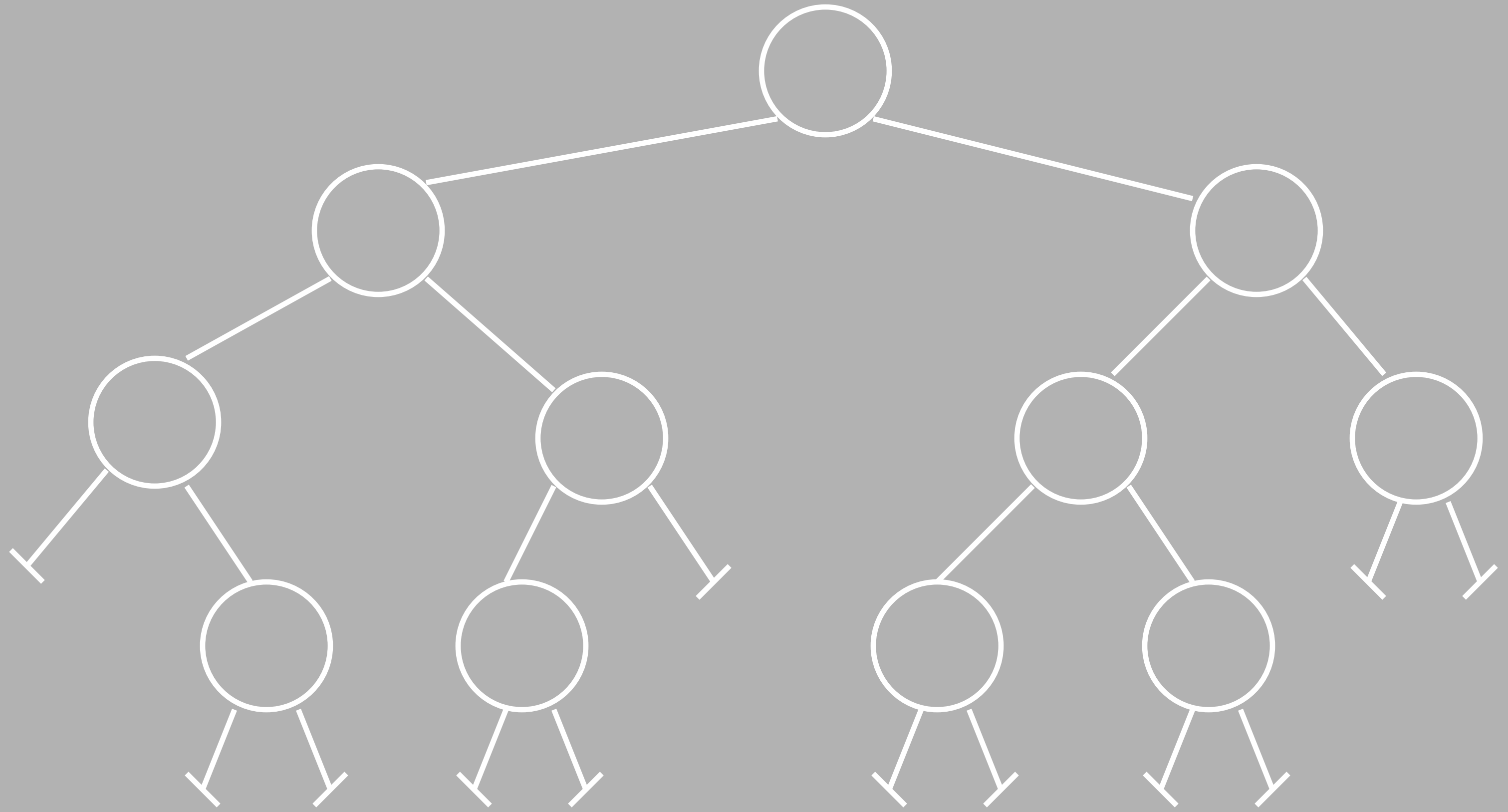
- General Tree Information:
  - Top node in a tree is called the **root**
    - the root node has no parent above it
  - Every node in the tree can have “children” nodes
    - Each child node can, in turn, be a parent to its children and so on
  - Nodes having no children are called **leaves**
  - Any node that is not a root or a leaf is an **interior node**
  - The **height** of a tree is defined to be the length of the longest path from the root to a leaf in that tree.
    - A tree with only one node (the root) has a height of zero.

Tree Stuff

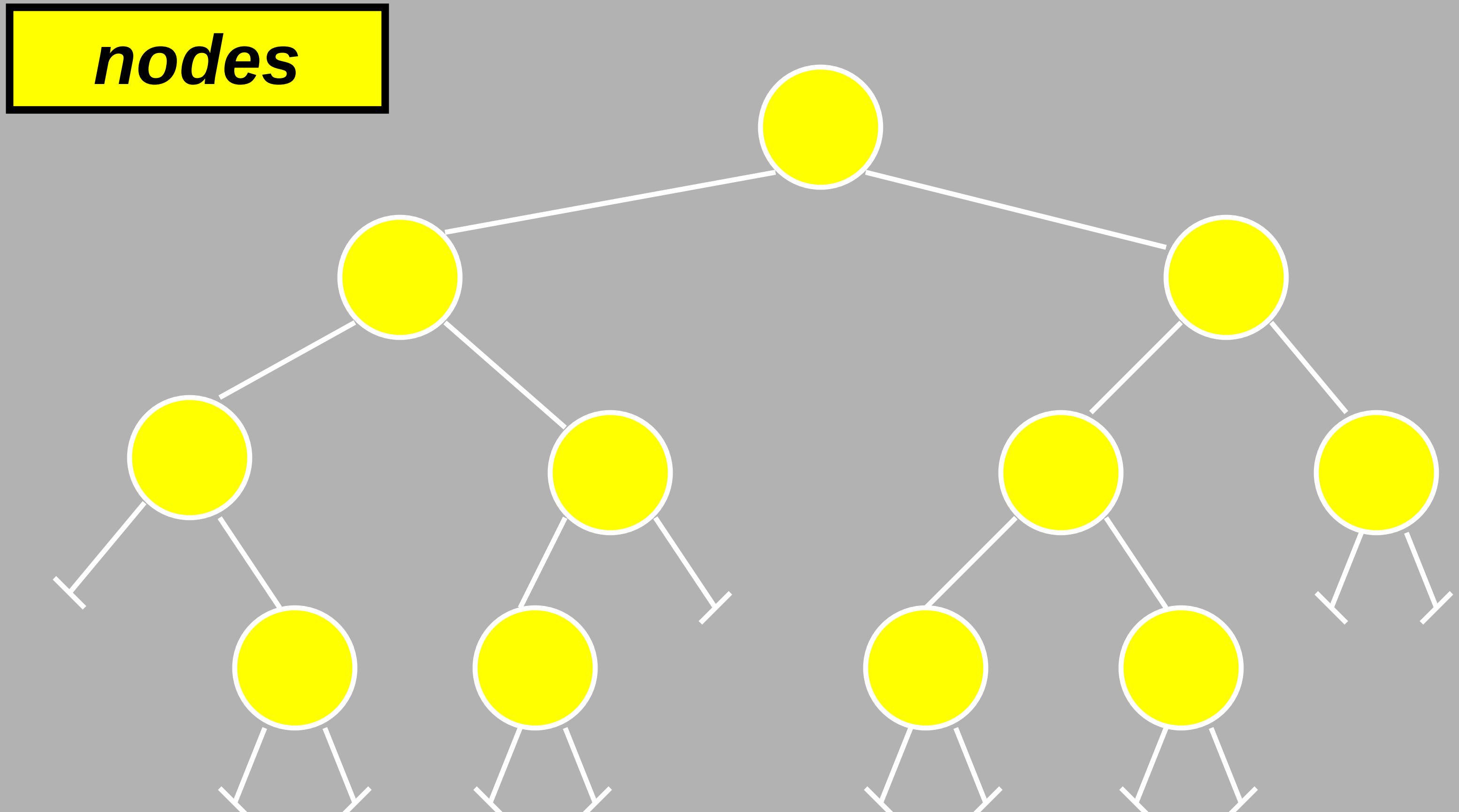
- Trees:
  - Here's an example picture of a tree:
    - 2 is the root
    - 2, 5, 11, and 4 are leaves
    - 7, 5, 6, and 9 are interior nodes



Parts of a Tree

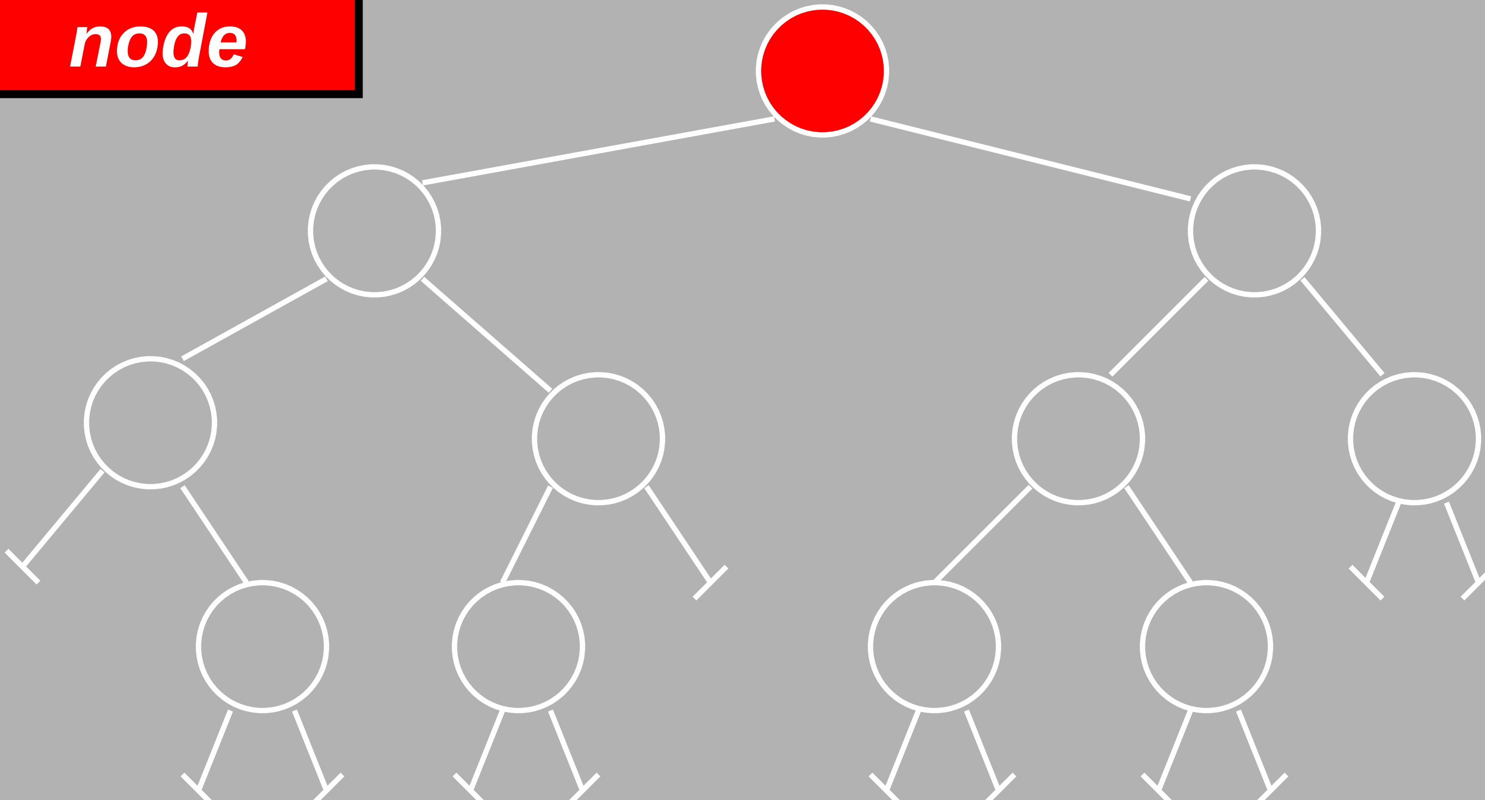


Parts of a Tree

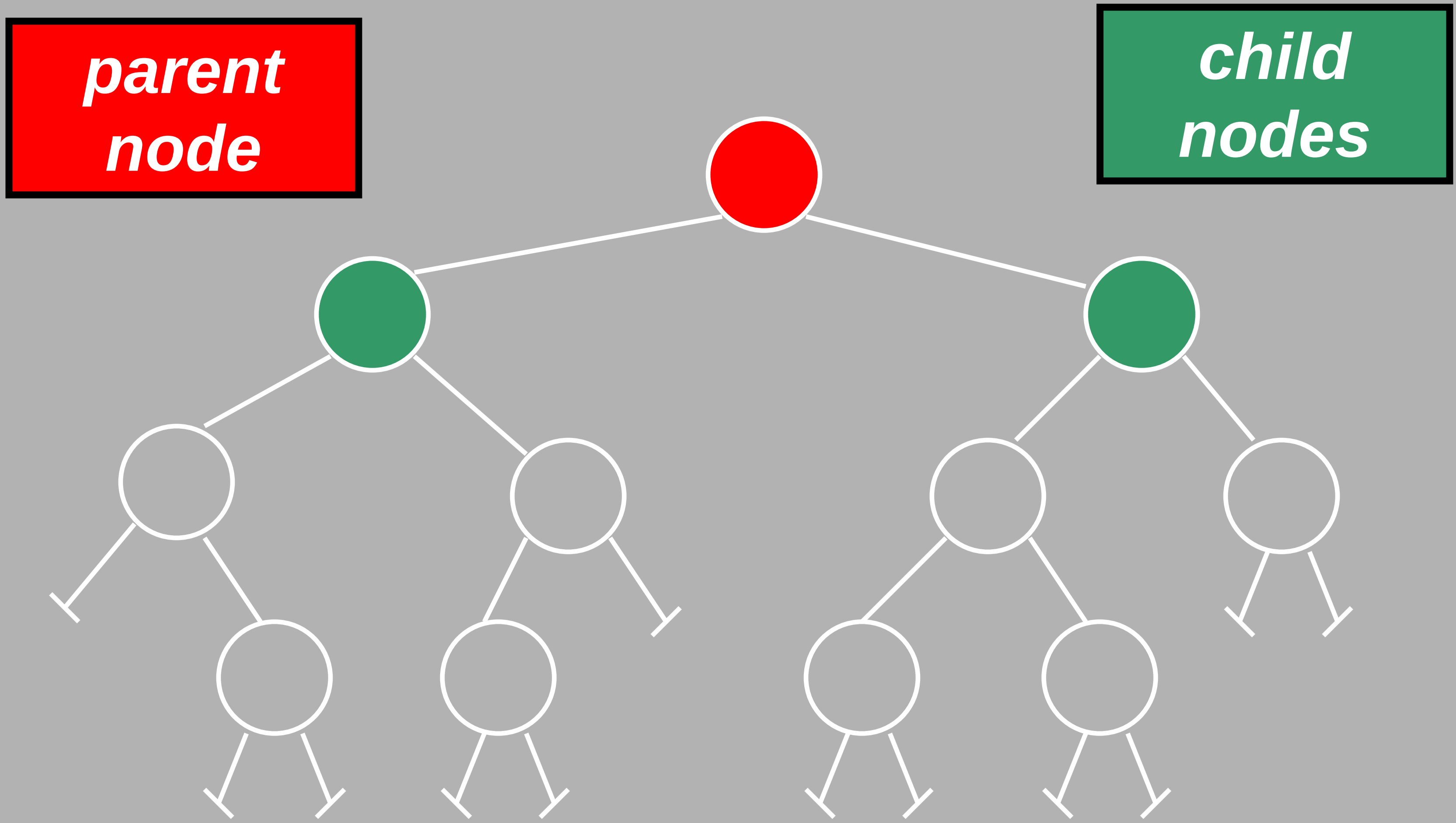


Parts of a Tree

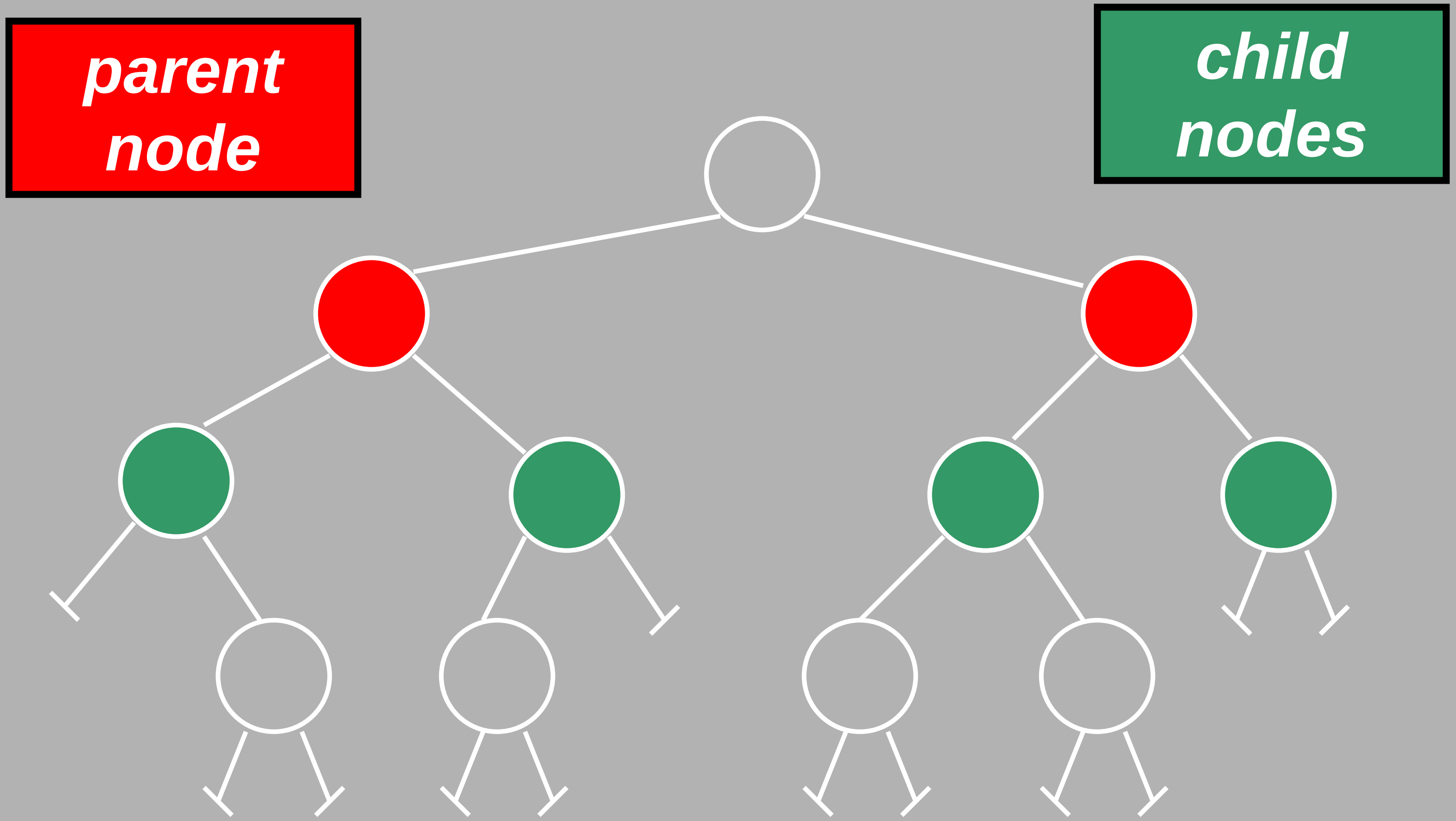
parent node



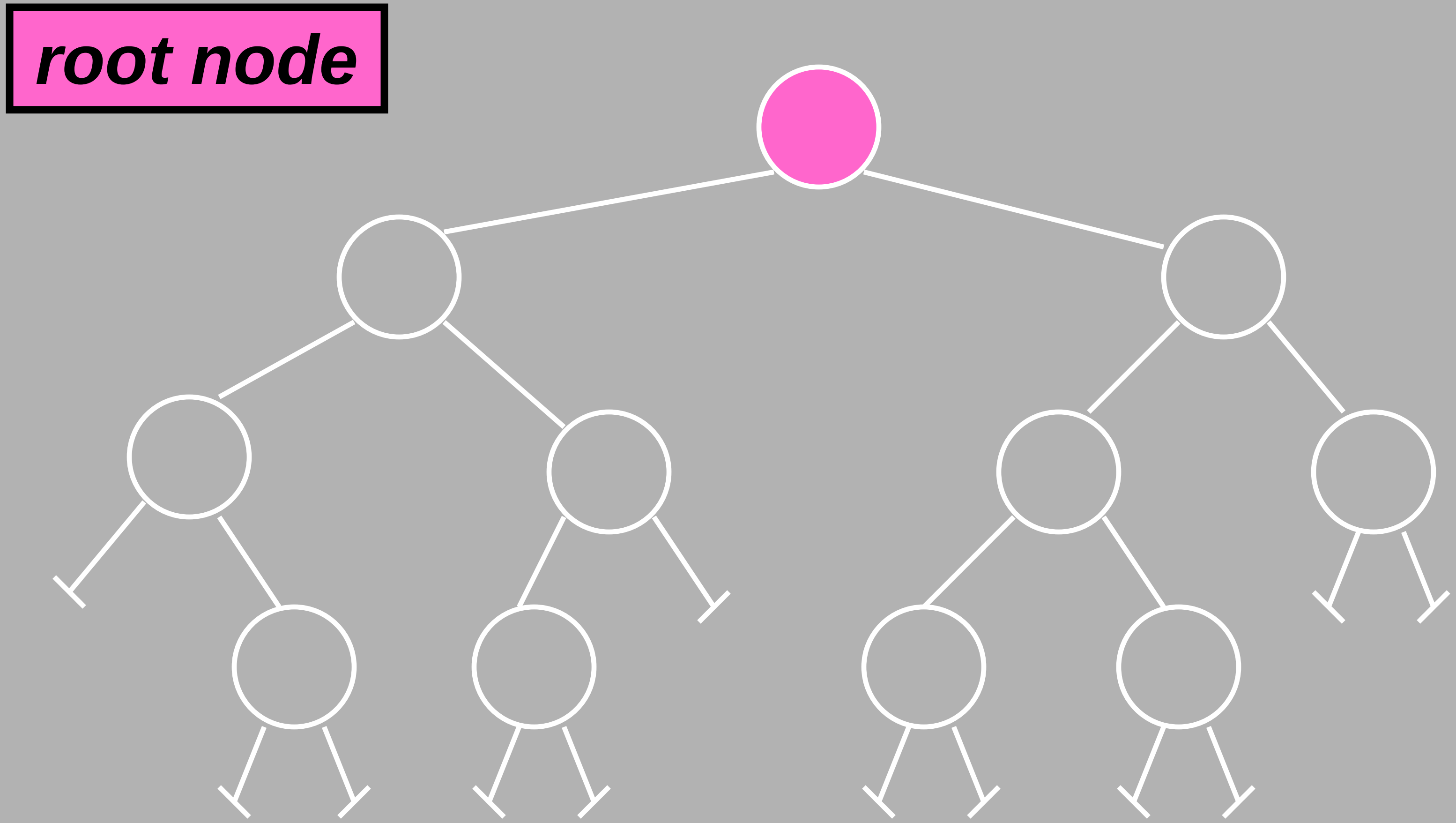
Parts of a Tree



Parts of a Tree

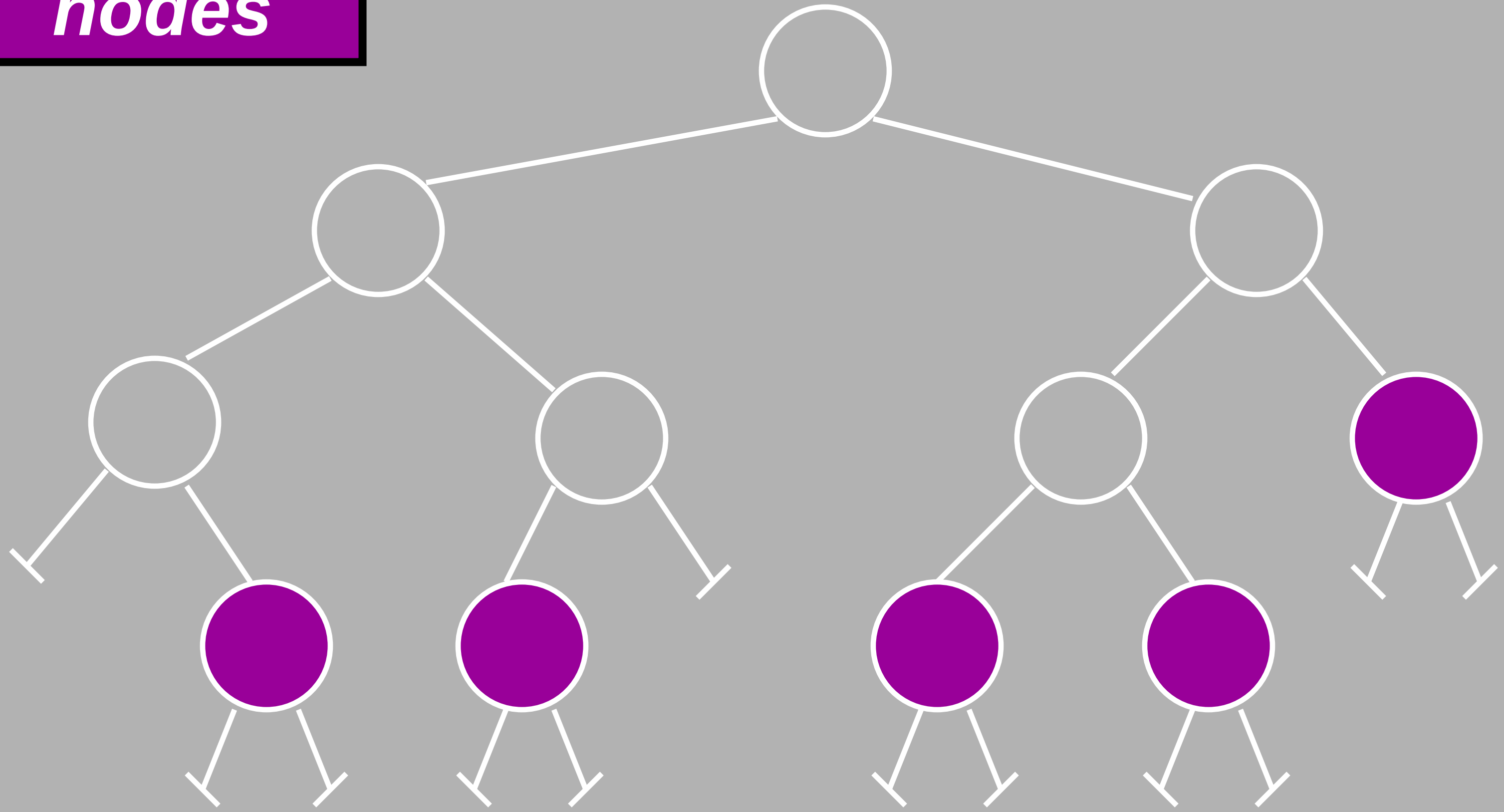


Parts of a Tree

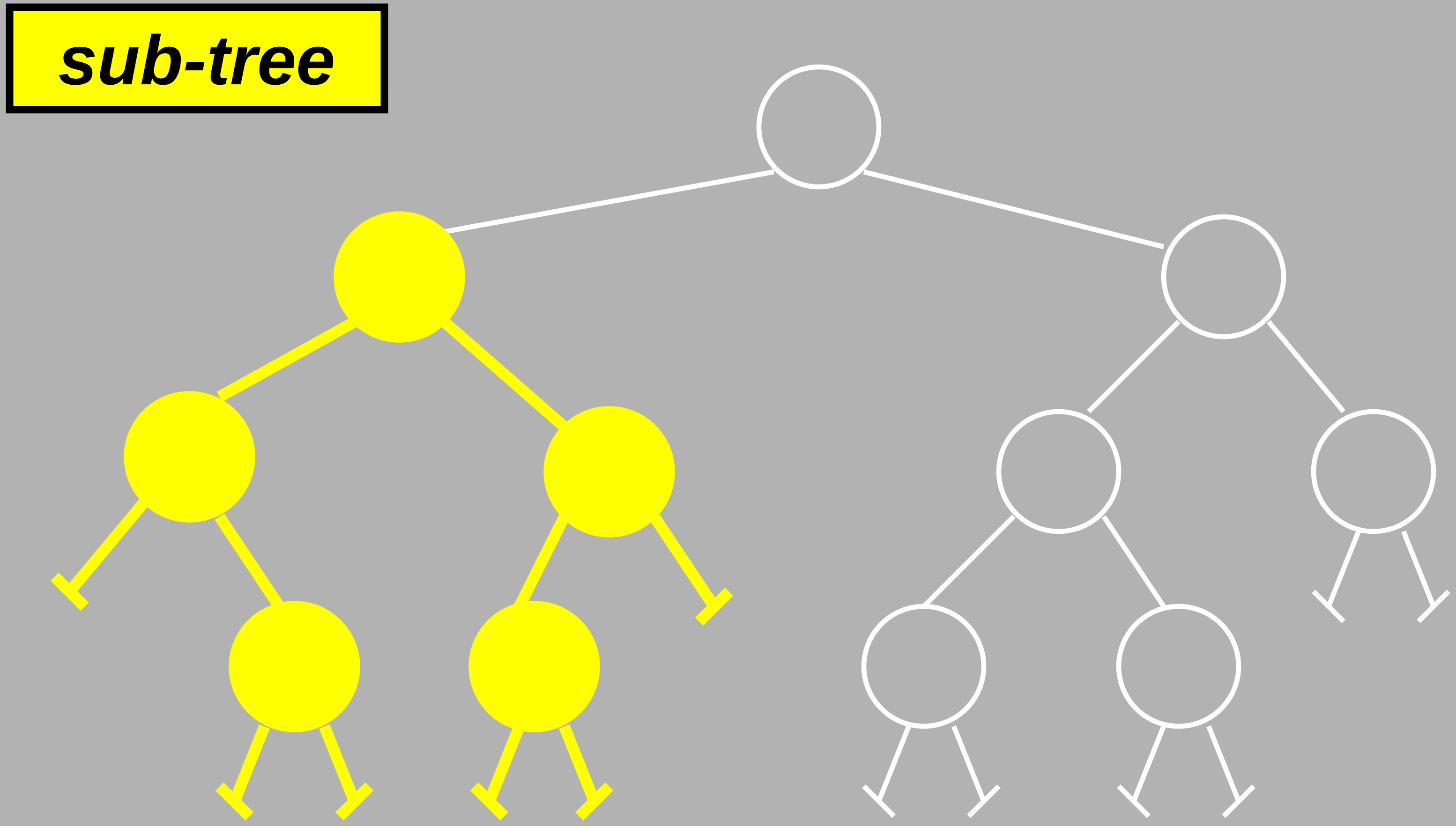


Parts of a Tree

leaf nodes

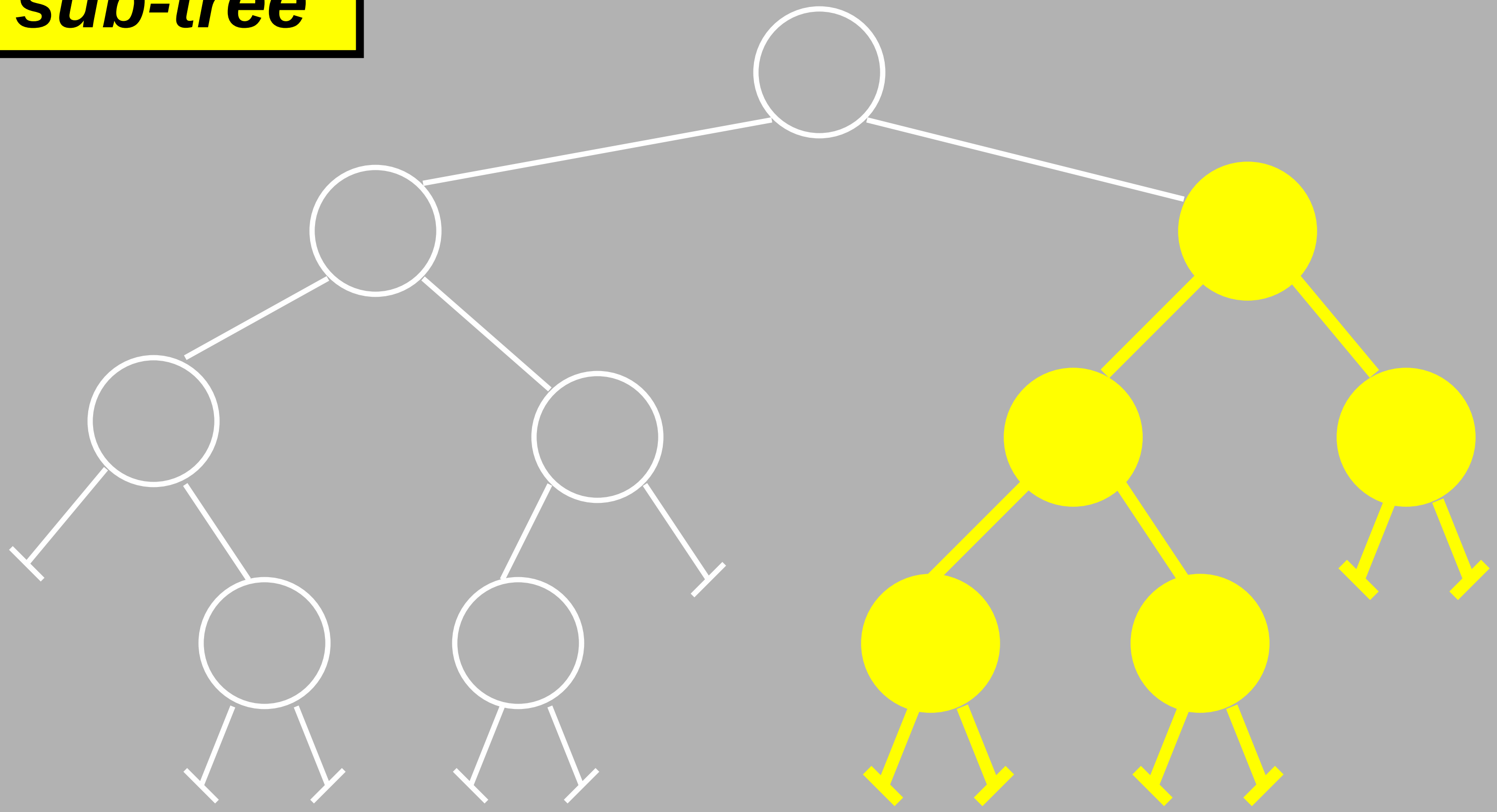


Parts of a Tree



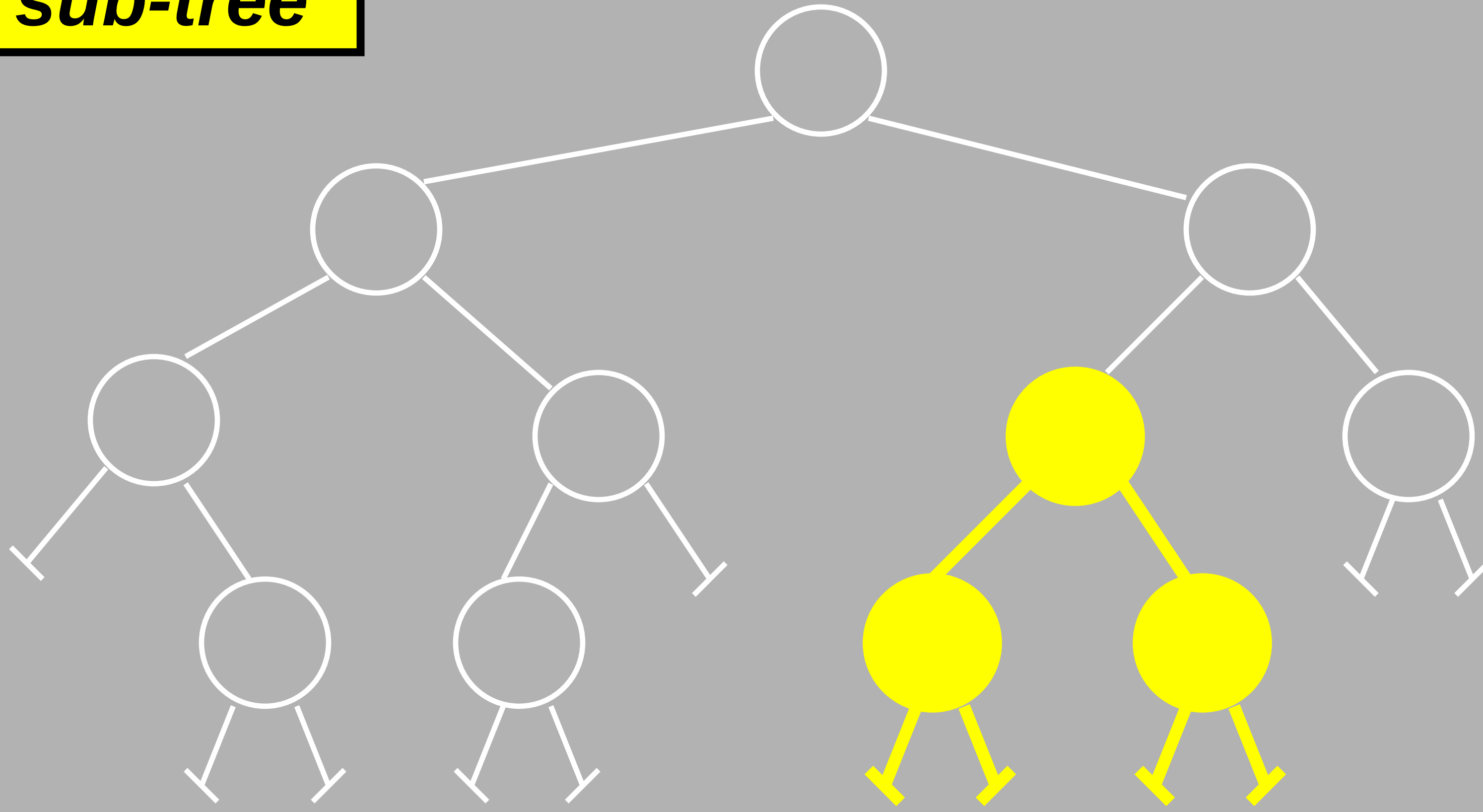
Parts of a Tree

sub-tree



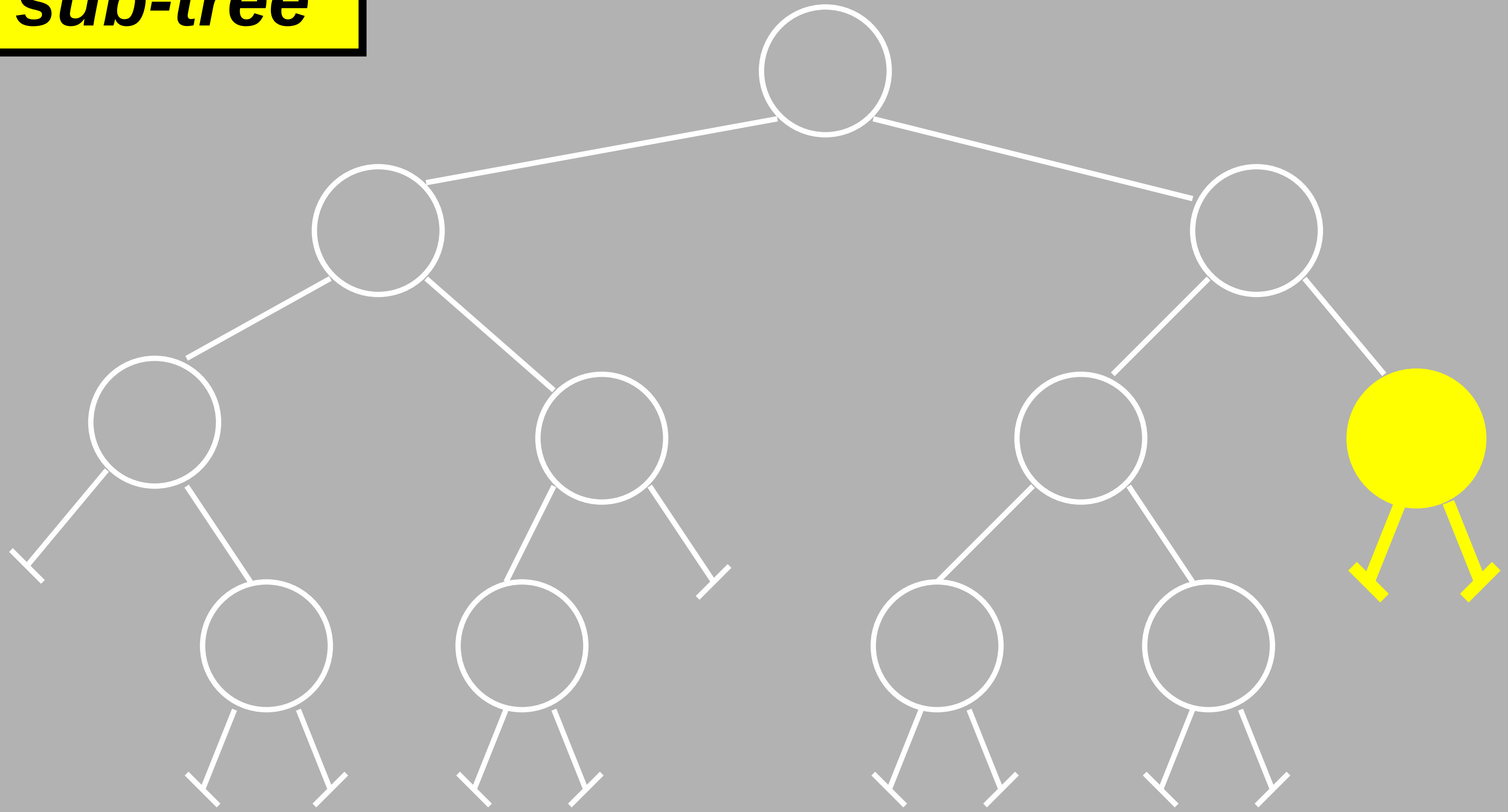
Parts of a Tree

sub-tree



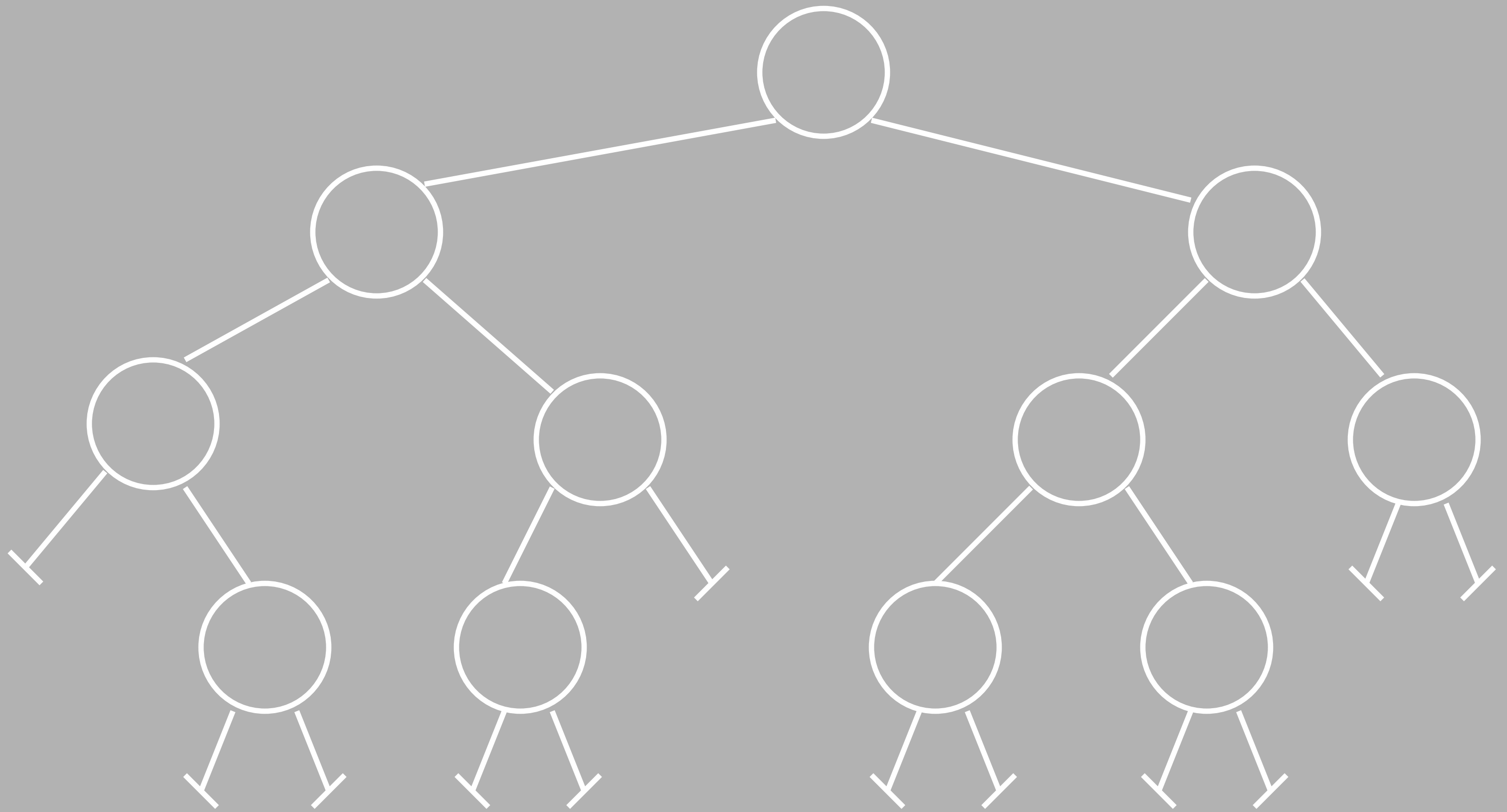
Parts of a Tree

sub-tree



Binary Tree

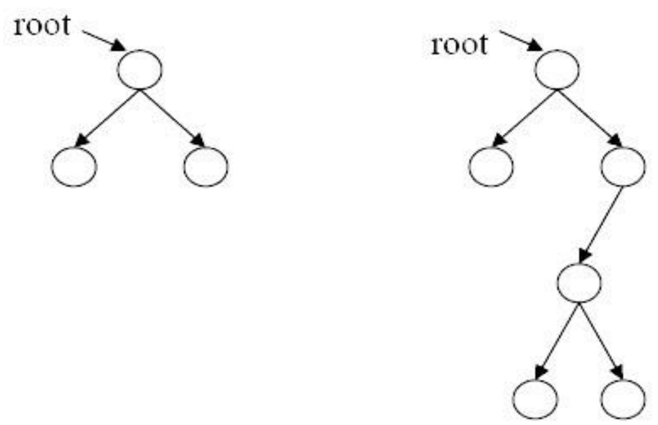
- Each node can have **at most 2** children



Binary Trees

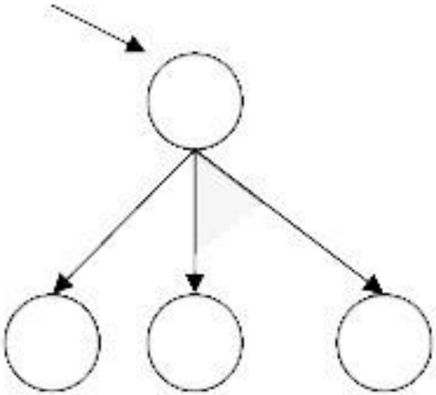
- A tree in which each node can have a maximum of two children
  - Each node can have no child, one child, or two children
  - And a child can only have one parent
  - Pointers help us to identify if it is a right child or a left one

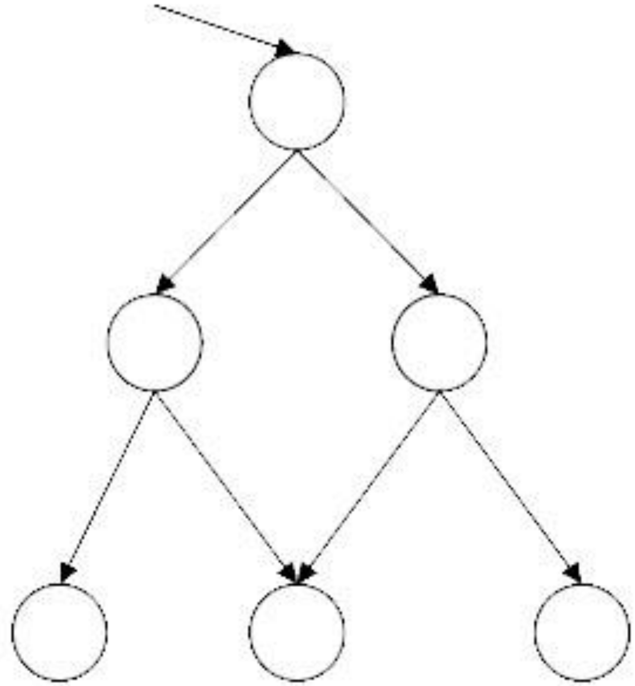
Examples of two Binary Trees:



Trees

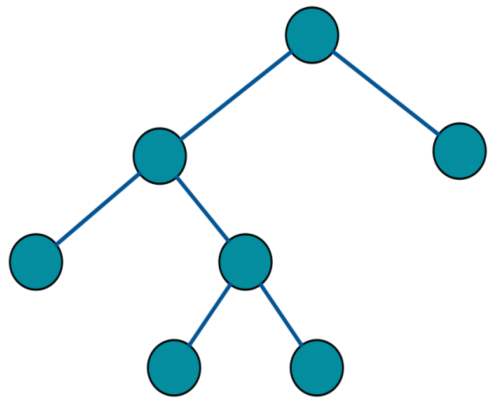
- A tree with maximum n child of a node is called n-ary tree.
- Examples of trees that are NOT Binary Trees:

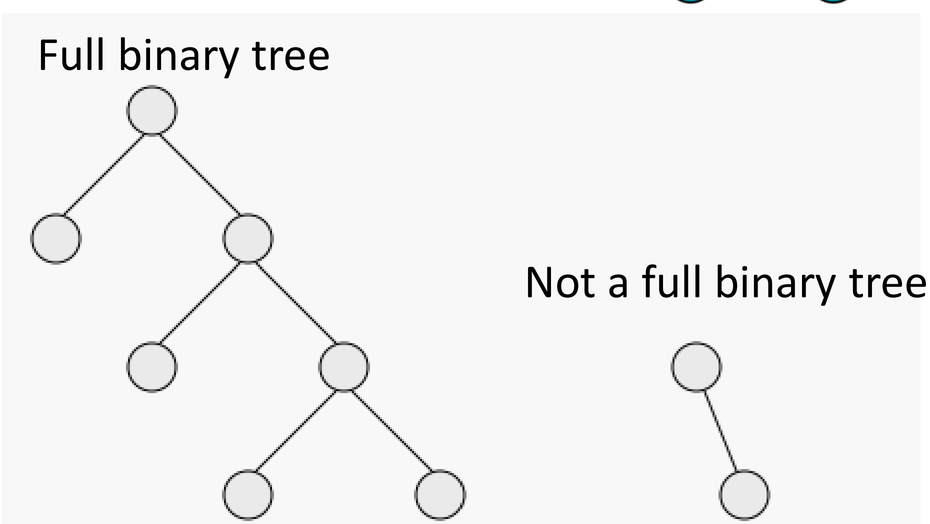


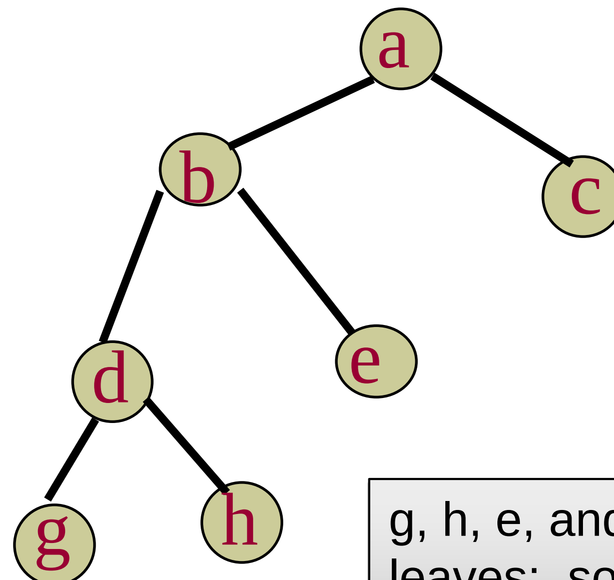


More Binary Tree

- A **full** binary tree:
  - Every node, other than the leaves, has two children
  - A binary tree T is full if each node is either a leaf or possesses exactly two child nodes.



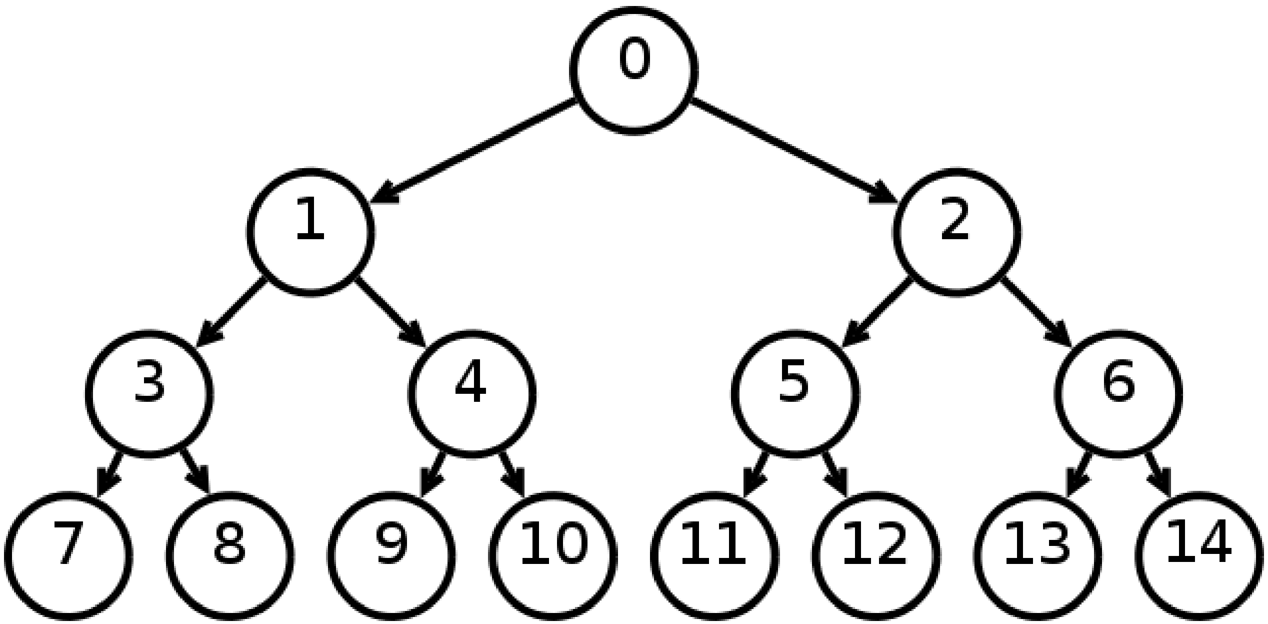




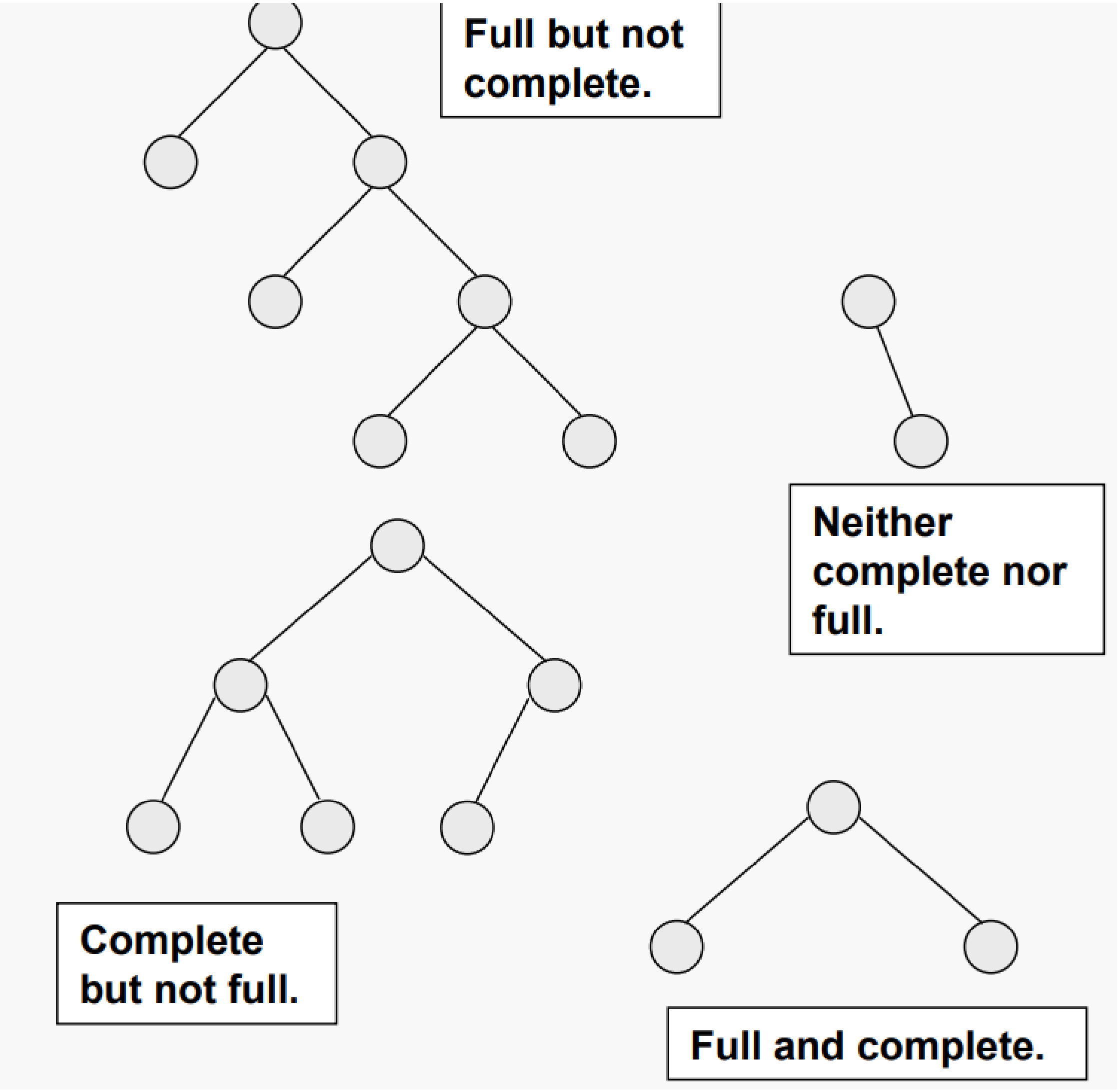
g, h, e, and c are leaves: so they have no children.

More Binary Tree

- A **complete** binary tree:
  - Every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



Complete Vs Full Binary Tree



Again Height of a Binary Tree

- The **height of a binary tree** is the largest number of edges in a path from the root node to a leaf node.

- Essentially, it is the **height** of the root node.

- Note that if a **tree** has only one node, then that node is at the same time the root node and the only leaf node, so the **height** of the **tree** is 0.

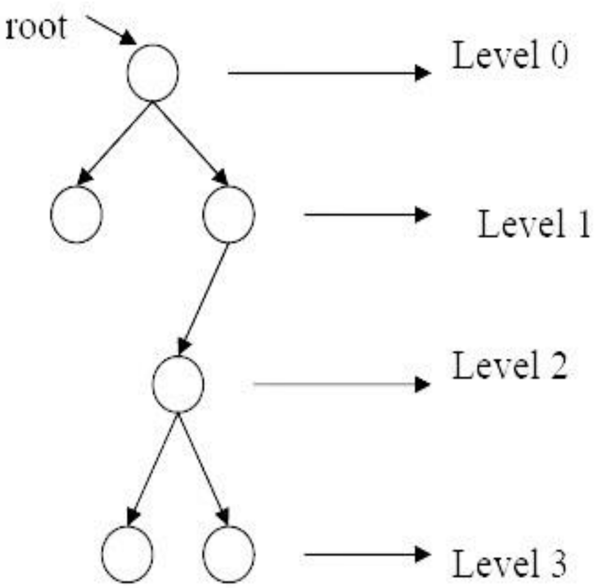
More Binary Tree

- **The root of the tree is at level 0**
- The level of any other node in the tree is one more than the level of its parent
- Total max# of nodes (n) in a

complete binary tree:

  - $n = 2^{h+1} – 1$ (maximum)
  - So, if height = 3, then $n = 2^{3+1} – 1 = 15$
  - See the example in two slides ago on complete binary tree

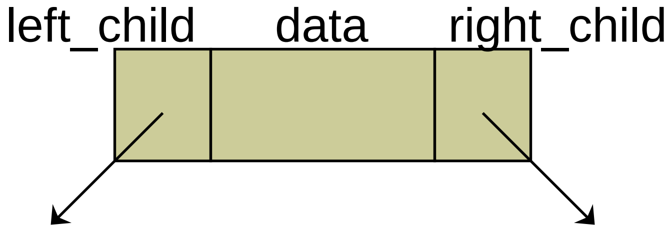
- Height (h) of the tree:
  - $h = \log((n + 1)/2)$
  - If we have 15 nodes
  - $h = \log(16/2) = \log(8) = 3$



Implementation of a Binary Tree:

- A binary tree has a natural implementation using linked storage
- Each node of a binary tree has both left and right subtrees that can be reached with pointers:

```
struct tree_node {
         int data;
         struct tree_node *left_child;
         struct tree_node *right_child;
}
```



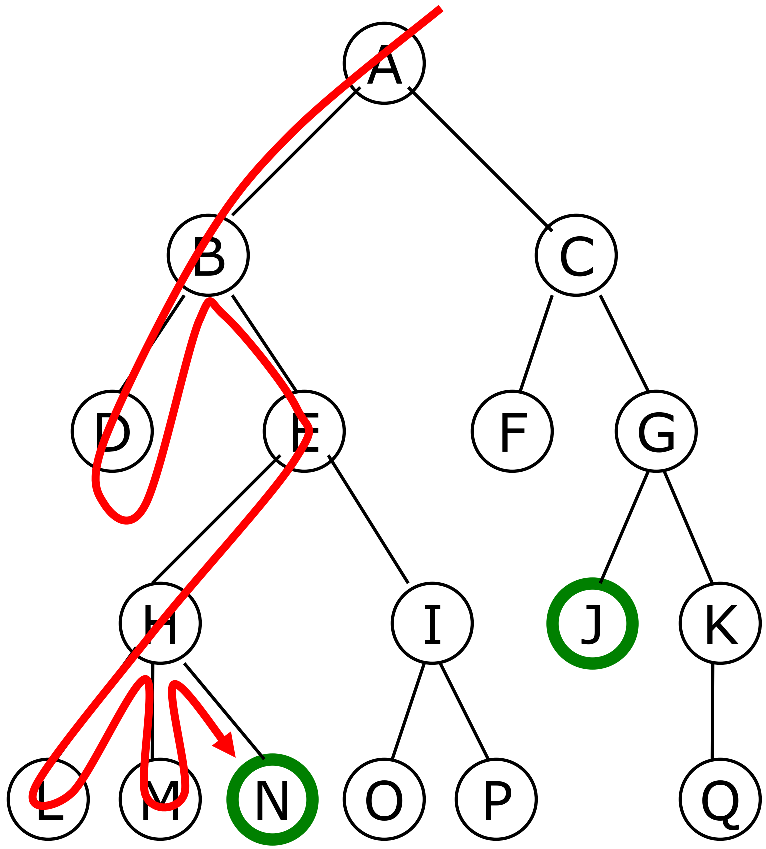
Tree Traversals – Depth First

- **Traversal of Binary Trees**:
  - We need a way of walk through through a tree for searching, inserting, etc.
  - In Linked lists are traversed from the head to the last node sequentially
    - Can’t we just “do that” for binary trees.?.
      - NO! There is no such natural linear ordering for nodes of a tree.
  - Turns out, there are THREE ways/orderings of traversing a binary tree:
    - Preorder,
    - Inorder, and
    - Postorder

Tree Traversals – Depth First

But before we get into the nitty gritty of those three, let’s describe..

Tree Traversals – Depth First



- A depth-first search (DFS) explores a path all the way to a leaf before backtracking and exploring another path
- For example, after searching A, then B, then D, the search backtracks and tries another path from B
- Node are explored in the order A B D E H L M N I O P C F G J K Q
- N will be found before J

Tree Traversals – Depth First

- There are 3 ways/orderings of traversing a binary tree (all 3 are depth first search methods):
  - **Preorder**, **Inorder**, and **Postorder**
  - These names are chosen according to the step at which the root node is visited:
    - With **preorder** traversal, the root is visited before its left and right subtrees.
    - With **inorder** traversal, the root is visited between the subtrees.
    - With **postorder** traversal, the root is visited after both subtrees.

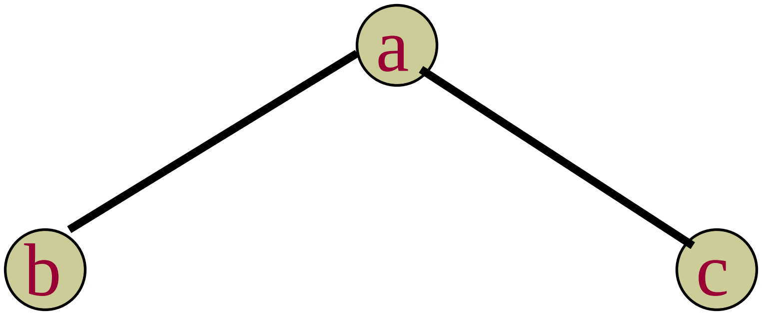
Tree Traversals - Preorder

- the root is visited before its left and right subtrees.
- For the following example, the "**visiting**" of a node is **represented by printing** that node
  - Code for Preorder Traversal:

```
void preorder (struct tree_node *p) {
        if (p != NULL) {
                printf("%d ", p->data);
                preorder(p->left_child);
                preorder(p->right_child);
        }
}
```

Tree Traversals - Preorder

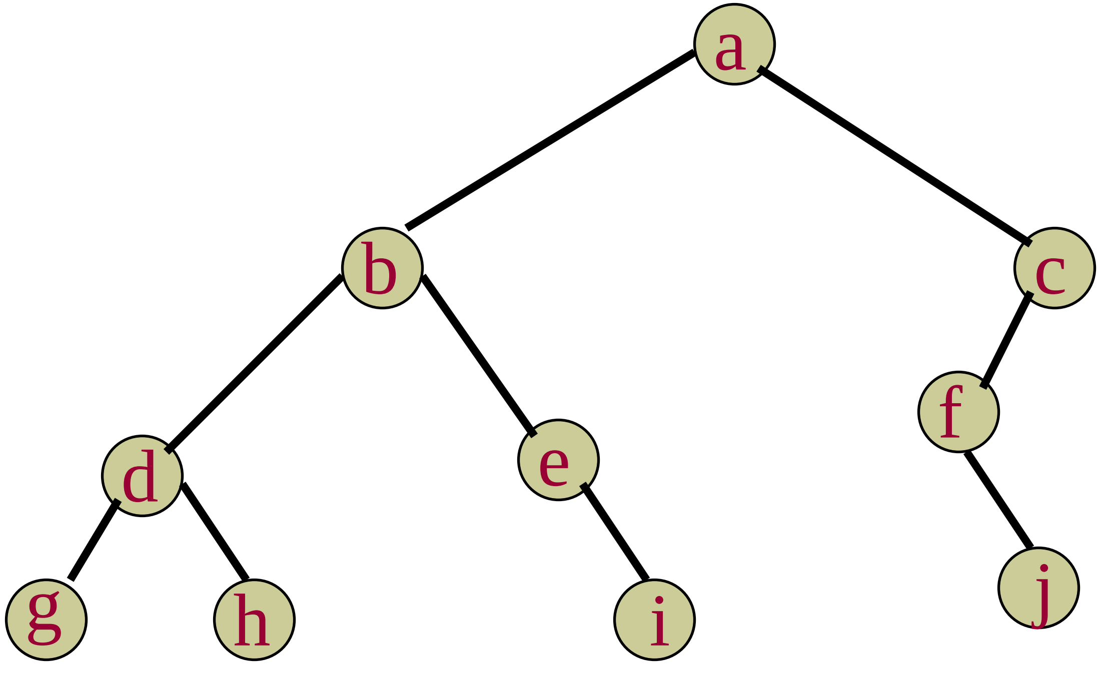
- Preorder Traversal – Example 1
  - the root is visited before its left and right subtrees



a b c

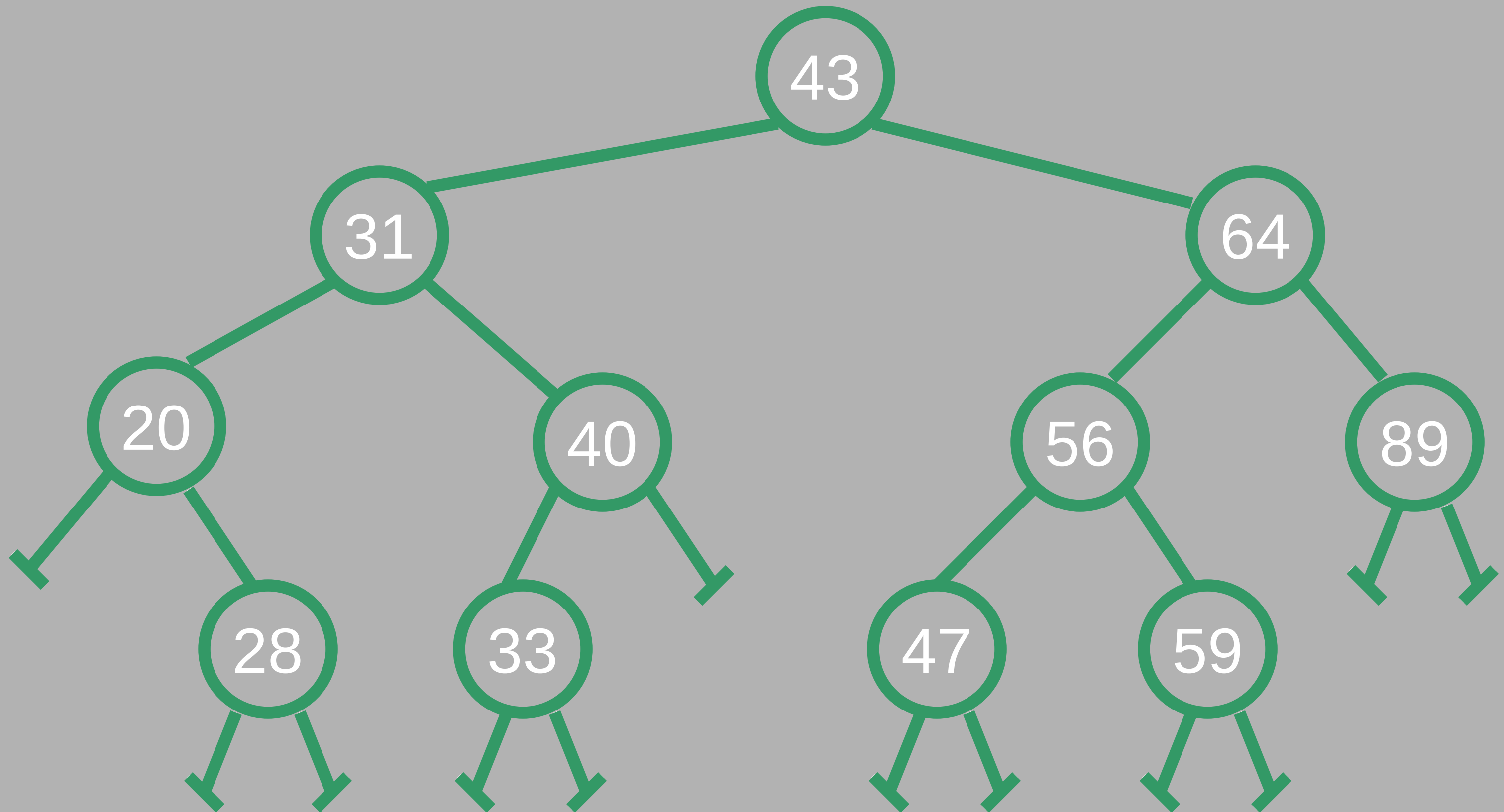
Tree Traversals - Preorder

- Preorder Traversal – Example 2



Order of Visiting Nodes: a b d g h e i c f j

Example: Preorder



43	31	20	28	40	33	64	56	47	59	89

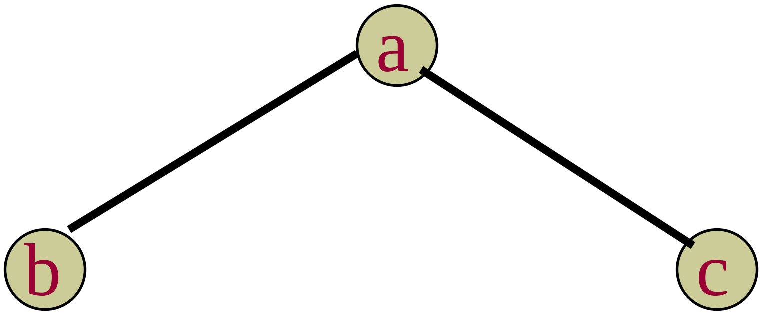
Tree Traversals - Inorder

- the root is visited between the left and right subtrees
  - For the following example, the “**visiting**” of a node is **represented by printing** that node
- Code for Inorder Traversal:

```
void inorder (struct tree_node *p) {
        if (p != NULL) {
                inorder(p->left_child);
                printf(“%d ”, p->data);
                inorder(p->right_child);
        }
}
```

Tree Traversals - Inorder

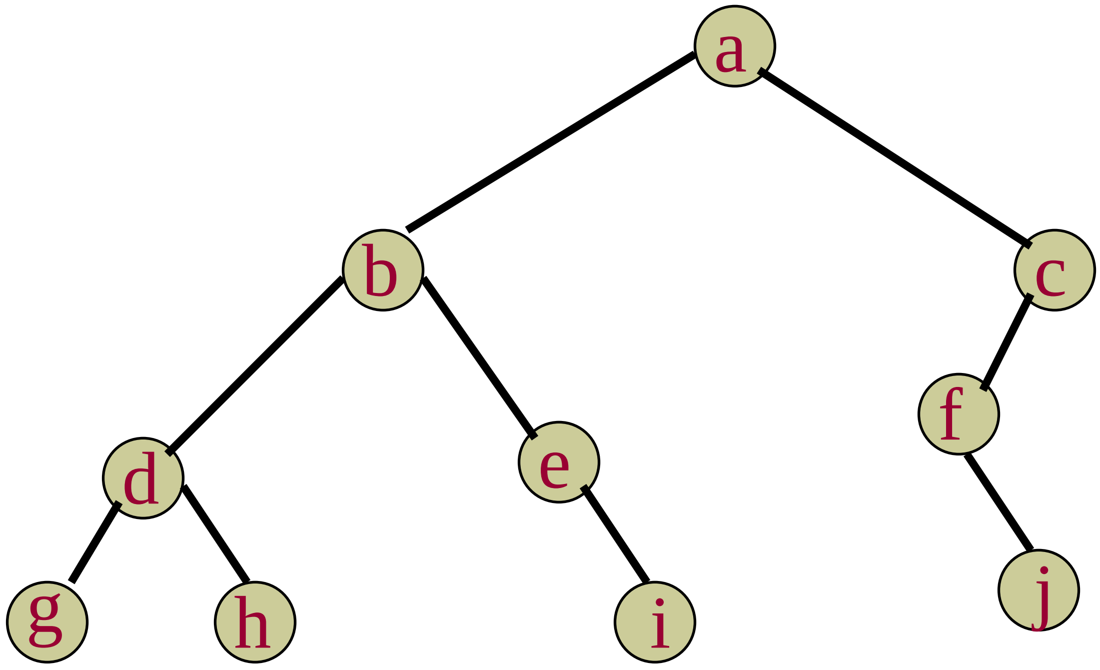
- Inorder Traversal – Example 1
  - the root is visited between the subtrees



b a c

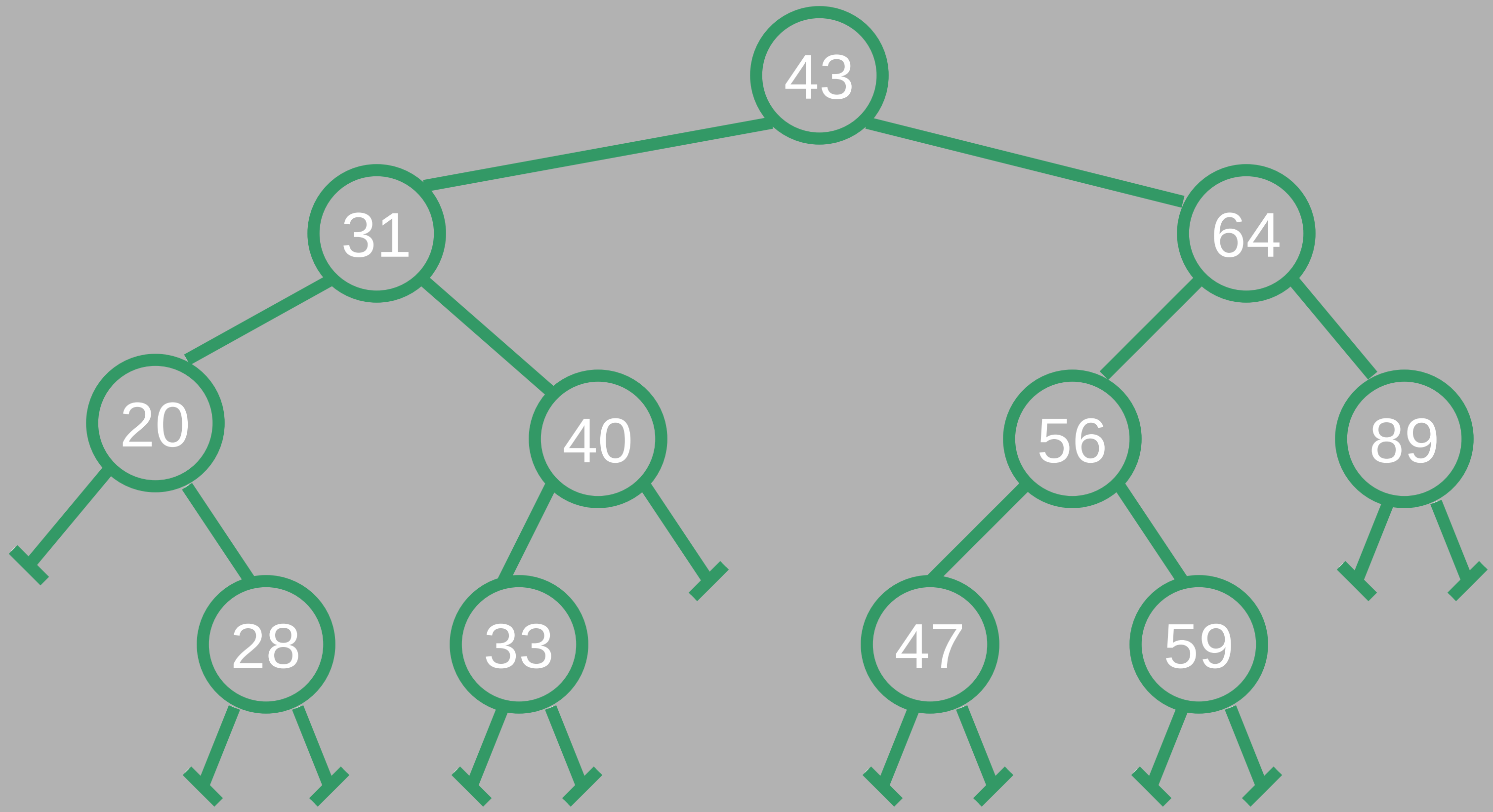
Tree Traversals - Inorder

- Inorder Traversal – Example 2



Order of Visiting Nodes: g d h b e i a f j c

Example 3: Inorder



20	28	31	33	40	43	47	56	59	64	89

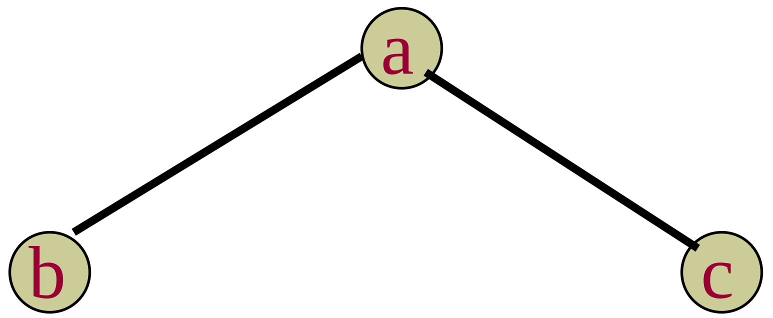
Tree Traversals – Postorder

- Postorder Traversal
  - the root is visited after both the left and right subtrees
    - For the following example, the “**visiting**” of a node is **represented by printing** that node
  - Code for Postorder Traversal:

```
void postorder (struct tree_node *p) {
        if (p != NULL) {
                postorder(p->left_child);
                postorder(p->right_child);
                printf(“%d ”, p->data);
        }
}
```

Tree Traversals – Postorder

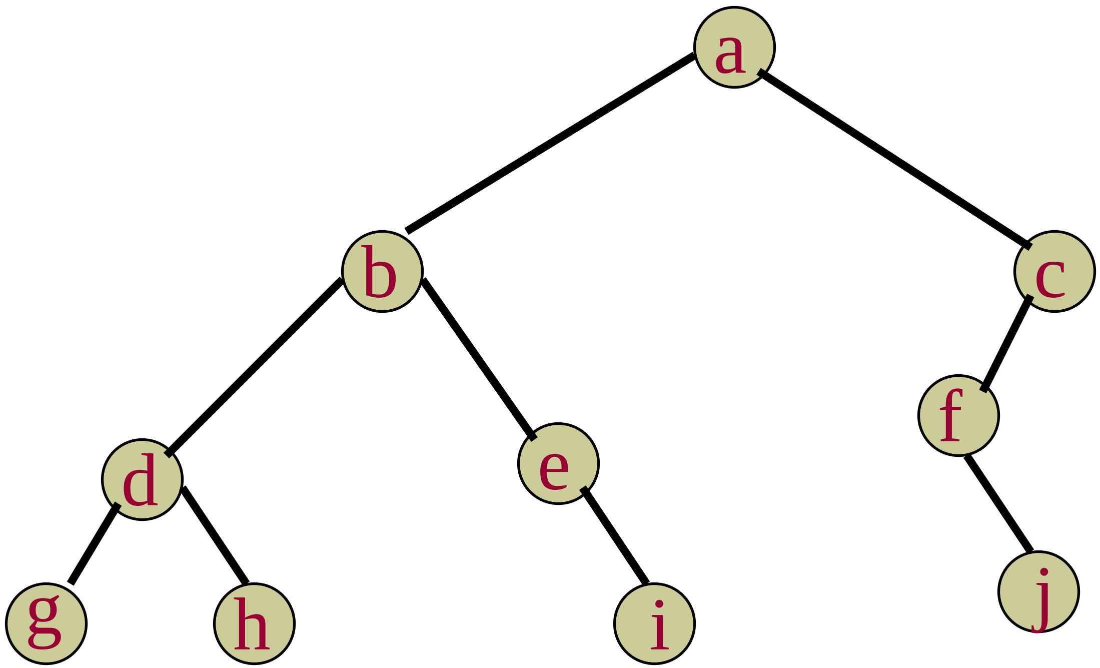
- Postorder Traversal – Example 1
  - the root is visited after both subtrees



b c a

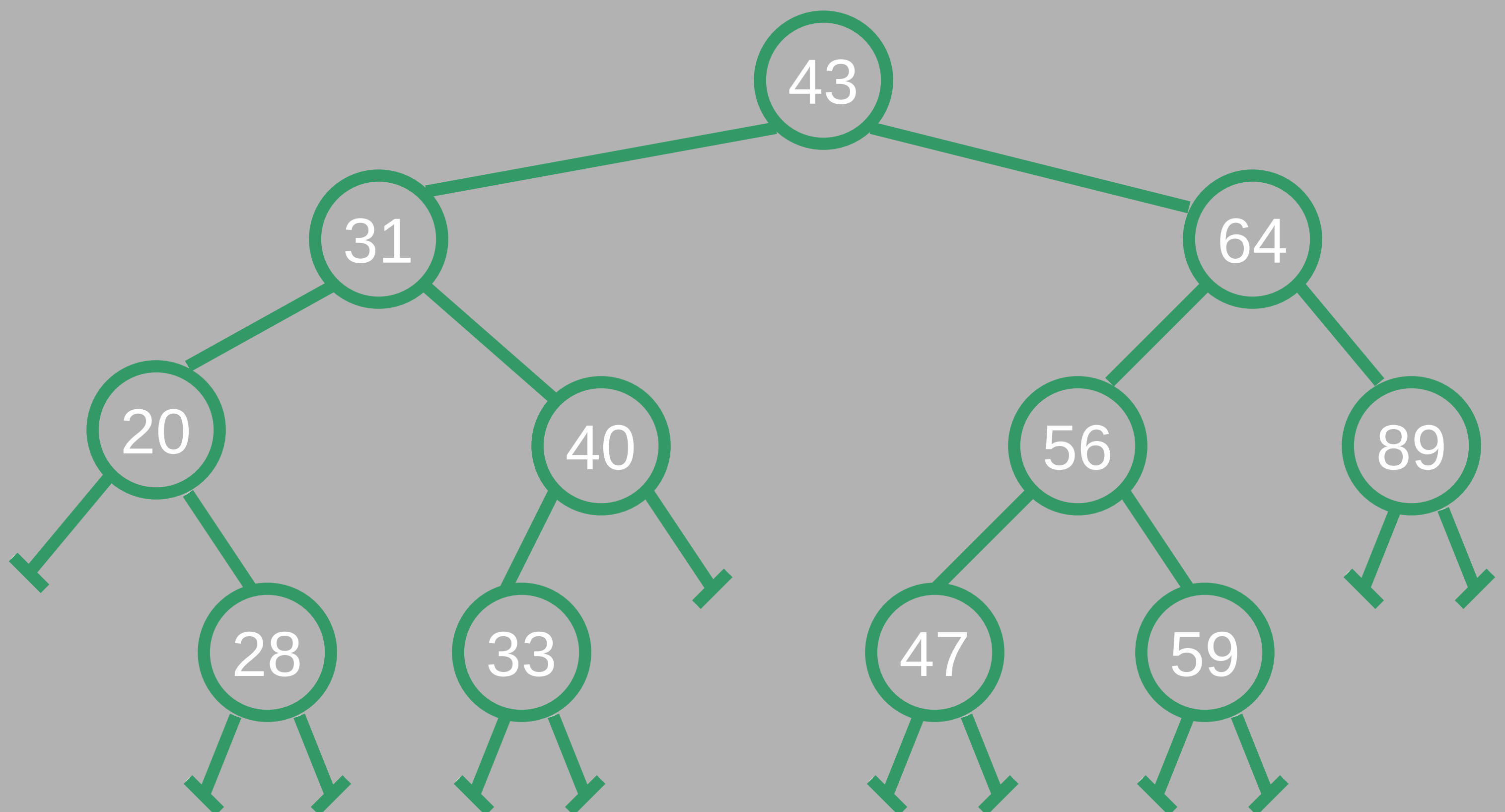
Tree Traversals – Postorder

- Postorder Traversal – Example 2



Order of Visiting Nodes: g h d i e b j f c a

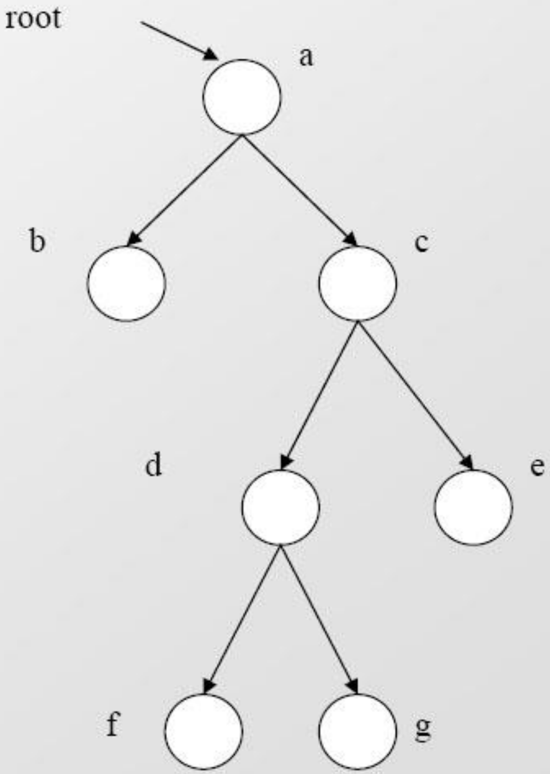
Example: Postorder



28	20	33	40	31	47	59	56	89	64	43

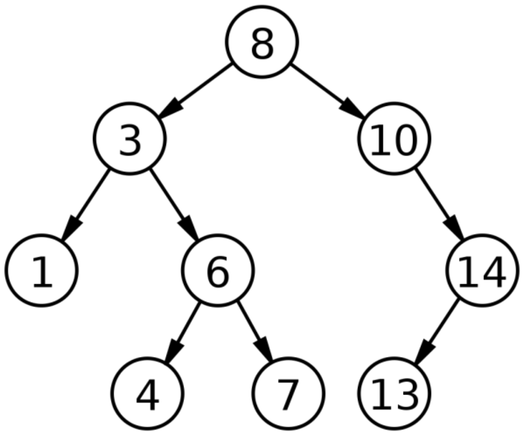
Tree Traversals

- Final Traversal Example
  - Preorder: a b c d f g e
  - Inorder: b a f d g c e
  - Postorder: b f g d e c a

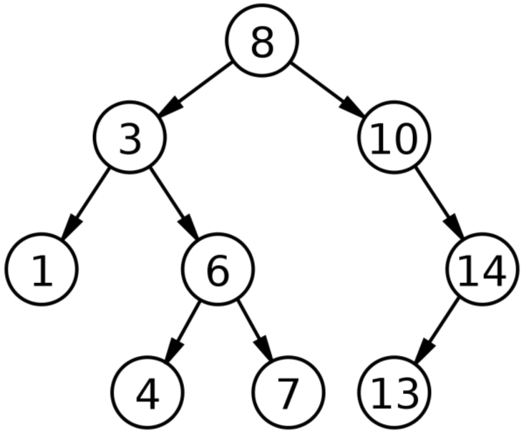


Binary Search Tree

- Binary Search Trees
  - We’ve seen how to traverse binary trees
  - But it is not quite clear how this data structure helps us
    - **What is the purpose of binary trees?**
  - What if we added a restriction…
  - Consider the following
  - binary tree:
  - What pattern can you see?



Binary Search Tree



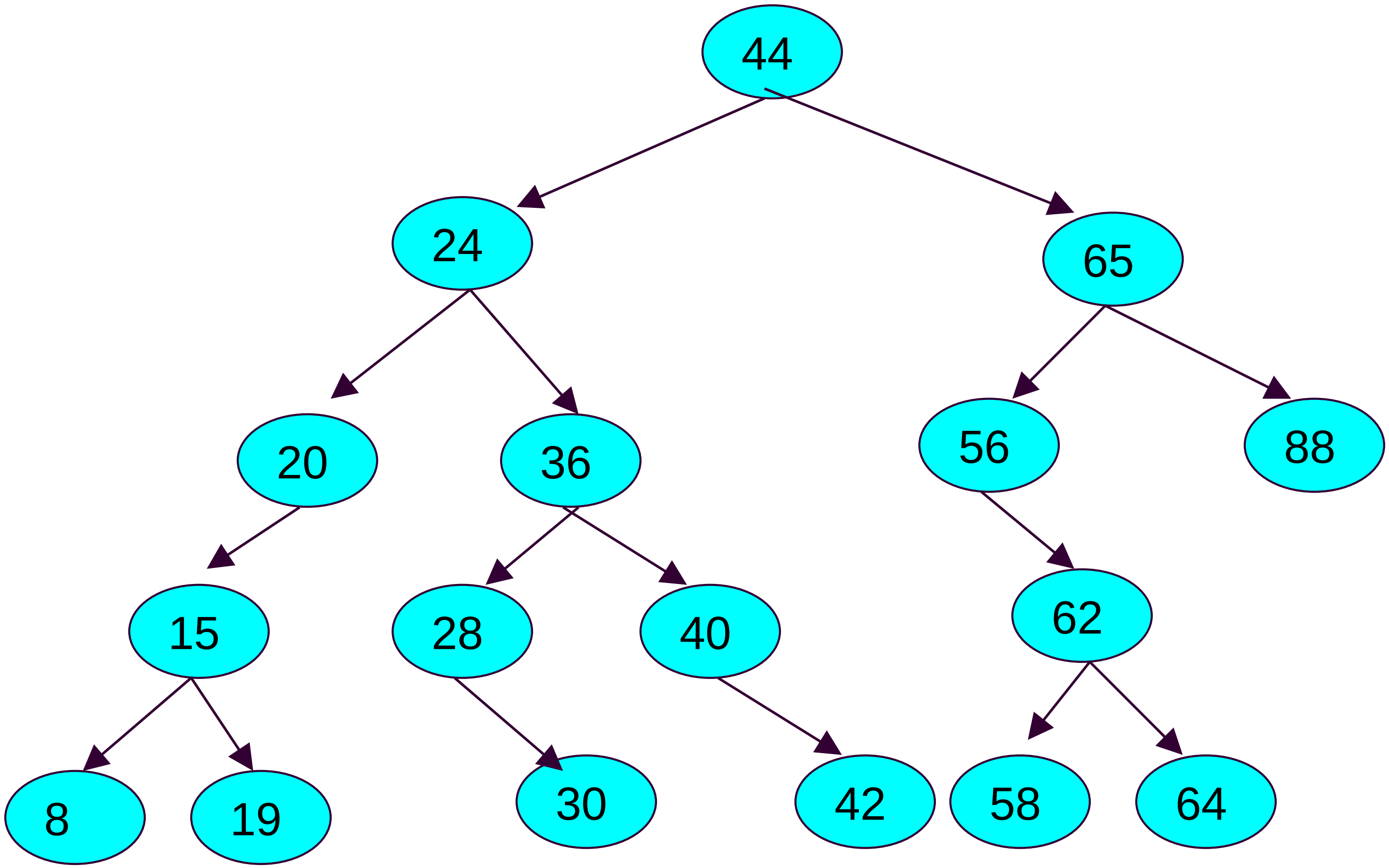
- Binary Search Trees
  - For each node N, all the values stored in the left subtree of N are LESS than the value stored in N.
  - Also, all the values stored in the right subtree of N are GREATER than the value stored in N.
  - Why might this property be a desirable one?
      - **Searching for a node is super fast!**
  - Normally, if we search through n nodes, it takes O(n) time
  - But notice what is going on here:
    - This **ordering property** of the tree **tells us where to search**
    - We choose to look to the left **OR** look to the right of a node
    - We are **HALVING** the search space

…**O(log n)** time

Binary Search Tree

- ALL of the data values in the left subtree of each node are **smaller** than the data value in the node itself (root of said subtree)
  - Stated another way, the value of the node itself is larger than the value of every node in its left subtree.
- ALL of the data values in the right subtree of each node are **larger** than the data value in the node itself (root of the subtree)
  - Stated another way, the value of the node itself is smaller than the value of every node in its right subtree.
- Both the left and right subtrees, of any given node, are themselves binary search trees.

Binary Search Tree



A Binary Search Tree

Binary Search Tree

- A binary search tree, commonly referred to as a **BST**, is **extremely useful for efficient searching**

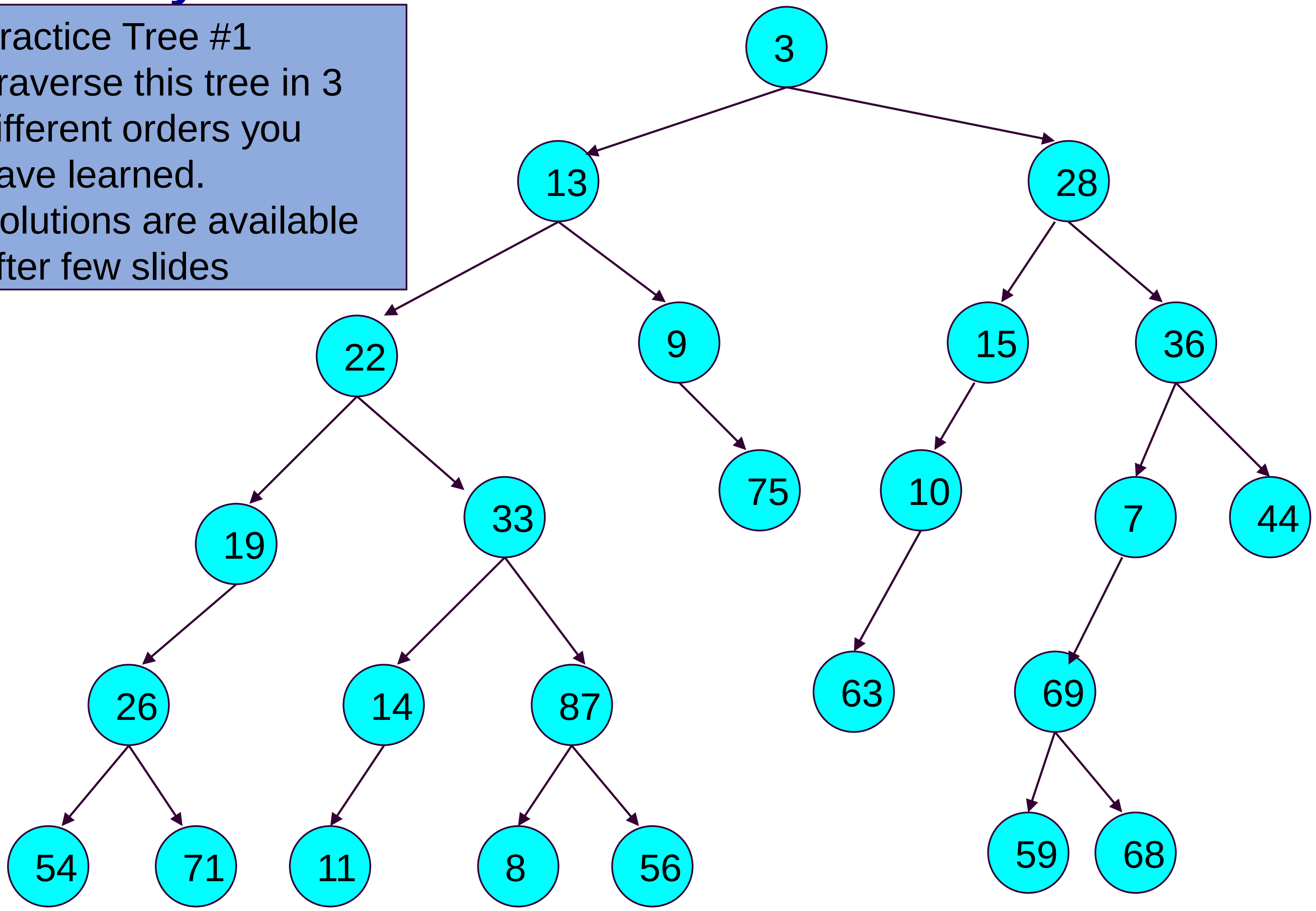
- Basically, a BST amounts to embedding the binary search into the data structure itself.
- Notice how the root of every subtree in the BST on the previous page is the root of a BST.

- **So, how about inserting something in a BST?**
- This ordering of nodes in the tree means that insertions into a BST are not placed arbitrarily
  - Rather, there is a specific way to insert
  - We will learn it

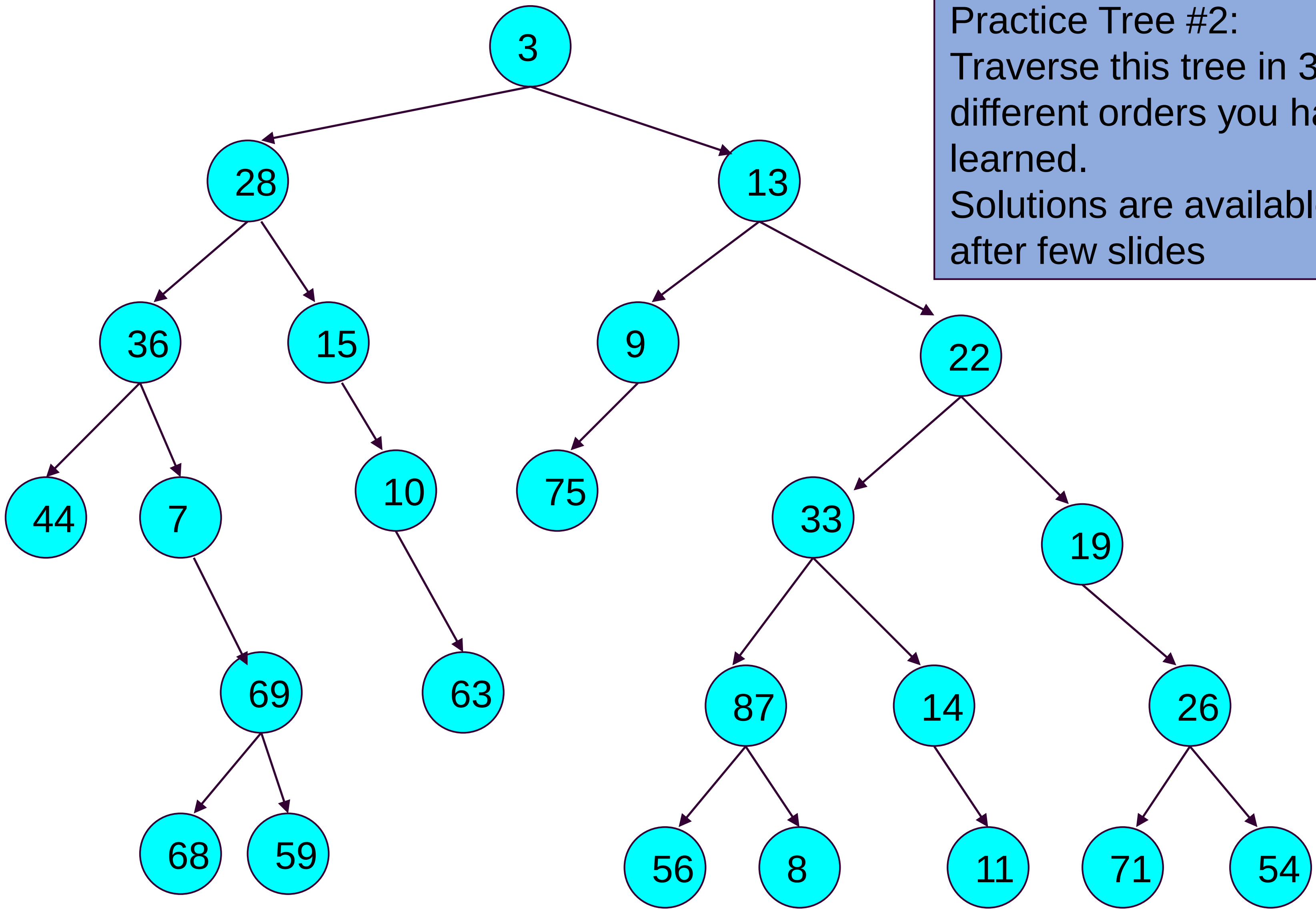
Binary Trees Practice

Binary Tree Traversals – Practice Problems

Practice Tree #1
Traverse this tree in 3 different orders you have learned.
Solutions are available after few slides



Binary Tree Traversals – Practice Problems



Practice Tree #2:
Traverse this tree in 3 different orders you have learned.
Solutions are available after few slides

Practice Problem Solutions – Tree #1

- Preorder Traversal:

  3, 13, 22, 19, 26, 54, 71, 33, 14, 11, 87, 8, 56, 9, 75, 28, 15, 10, 63, 36, 7, 69, 59, 68, 44

- Inorder Traversal:

  54, 26, 71, 19, 22, 11, 14, 33, 8, 87, 56, 13, 9, 75, 3, 63, 10, 15, 28, 59, 69, 68, 7, 36, 44

- Postorder Traversal:

  54, 71, 26, 19, 11, 14, 8, 56, 87, 33, 22, 75, 9, 13, 63, 10, 15, 59, 68, 69, 7, 44, 36, 28, 3

Practice Problem Solutions – Tree #2

- Preorder Traversal:

  3, 28, 36, 44, 7, 69, 68, 59, 15, 10, 63, 13, 9, 75, 22, 33, 87, 56, 8, 14, 11, 19, 26, 71, 54

- Inorder Traversal:

  44, 36, 7, 68, 69, 59, 28, 15, 10, 63, 3, 75, 9, 13, 56, 87, 8, 33, 14, 11, 22, 19, 71, 26, 54

- Postorder Traversal:

  44, 68, 59, 69, 7, 36, 63, 10, 15, 28, 75, 9, 56, 8, 87, 11, 14, 33, 71, 54, 26, 19, 22, 13, 3

References and Acknowledgement

1.) Many slides and text where taken from:
https://www.cs.ucf.edu/courses/cop3502/spr2012/notes/COP3502_22_BinaryTrees1.pdf

2. More Reading Resource:
http://www.cs.ucf.edu/~dmarino/ucf/transparency/cop3502/lec/BinaryTrees-1.doc