TREES

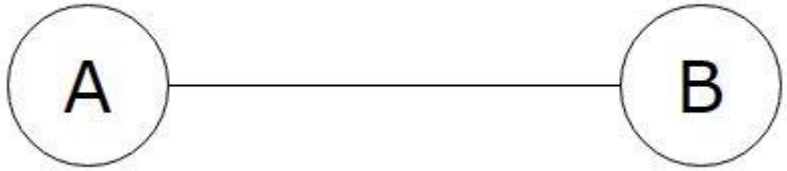
The Tree ADT

- A tree is an abstract data structure.

Terminology

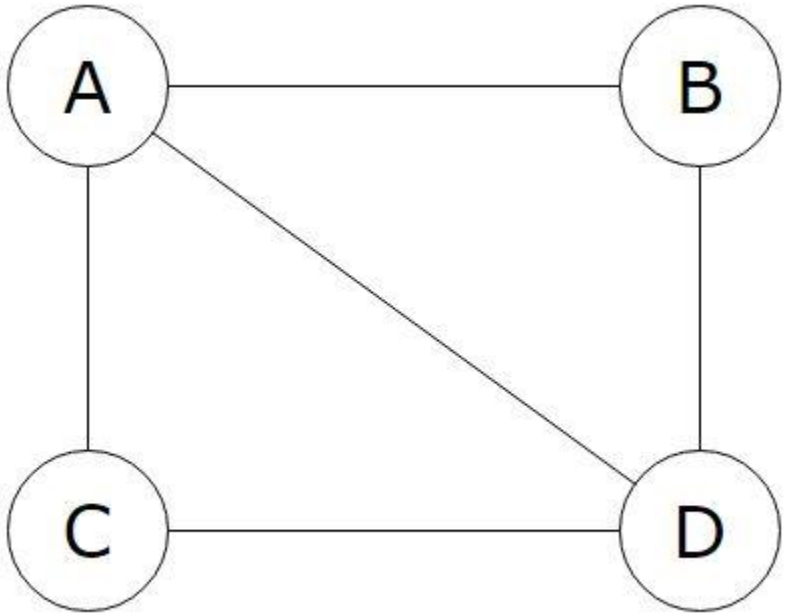
- Node: an element

- Edge: a connection between two nodes



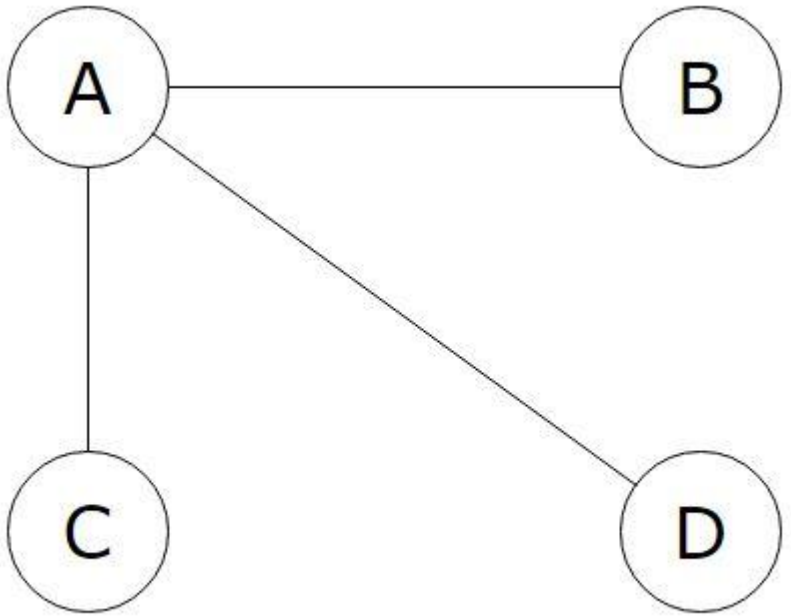
Terminology

- Graph: a set of nodes (vertices) and edges
- Cycle: a path with the same starting and ending node



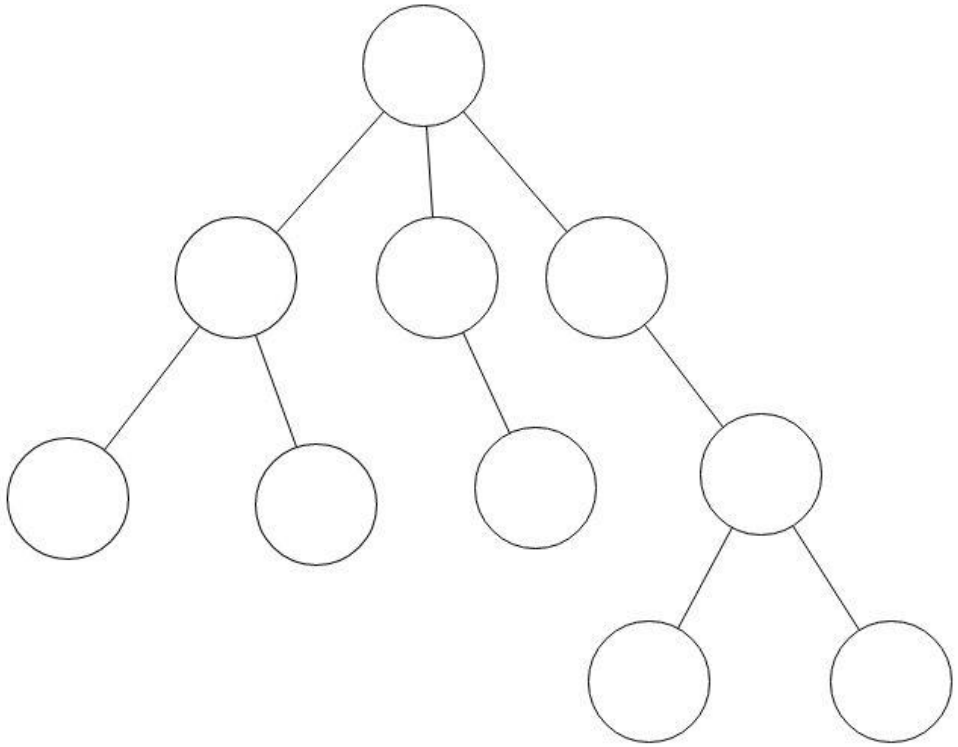
Terminology

- Tree: a graph with no cycles (*acyclic*)

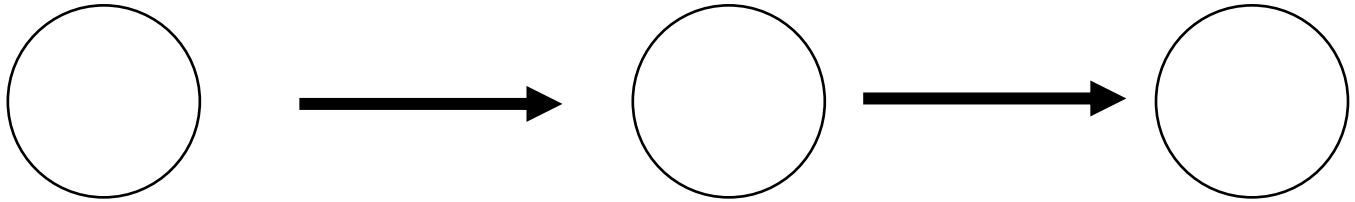


Terminology

- Tree: a graph with no cycles (*acyclic*)

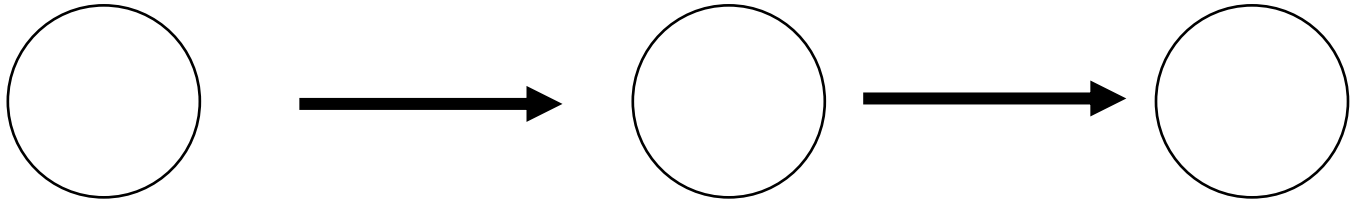


Examples of Nodes/Vertices and Edges



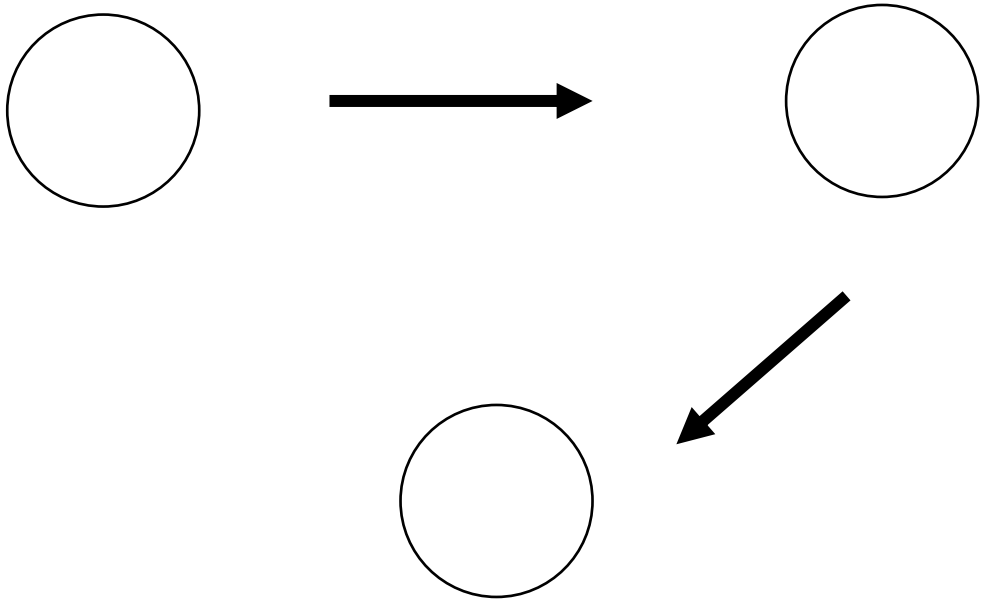
- Graph?
- Tree?
- Linked List?

Examples of Nodes/Vertices and Edges



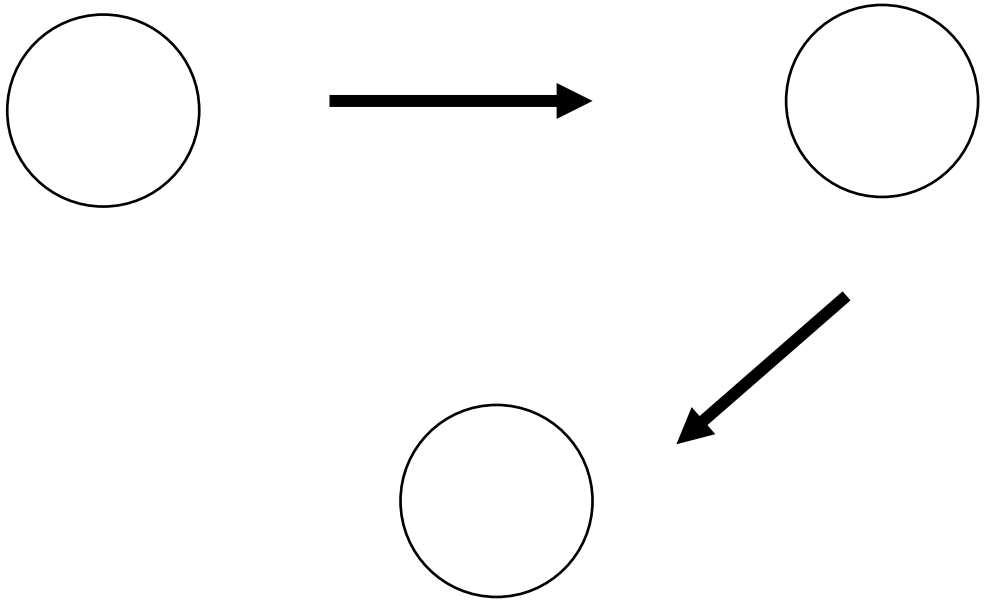
- Graph? yes! set of nodes and edges
- Tree? yes! a graph with no cycles

Examples of Nodes/Vertices and Edges



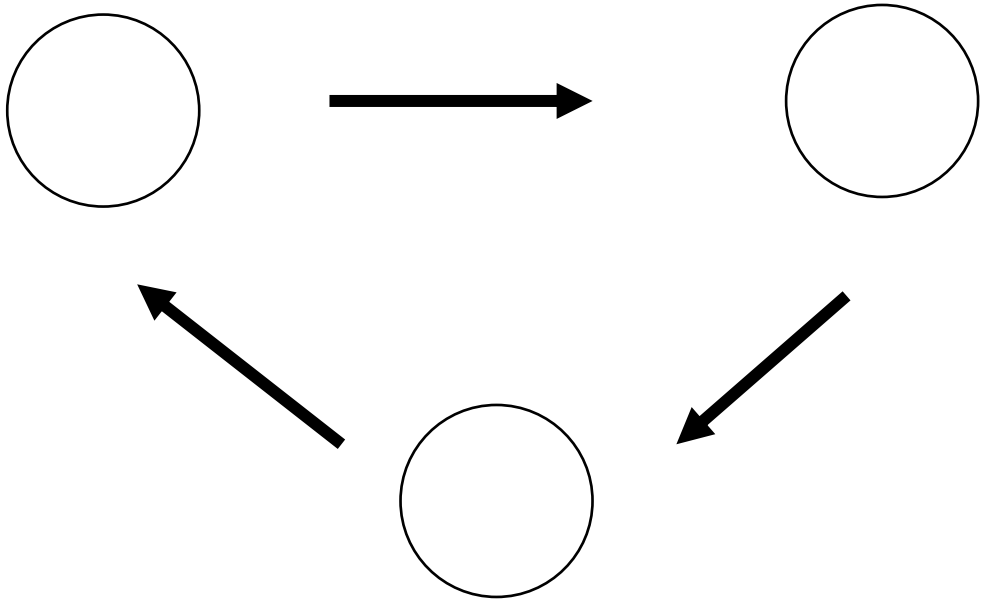
- Graph?
- Tree?

Examples of Nodes/Vertices and Edges



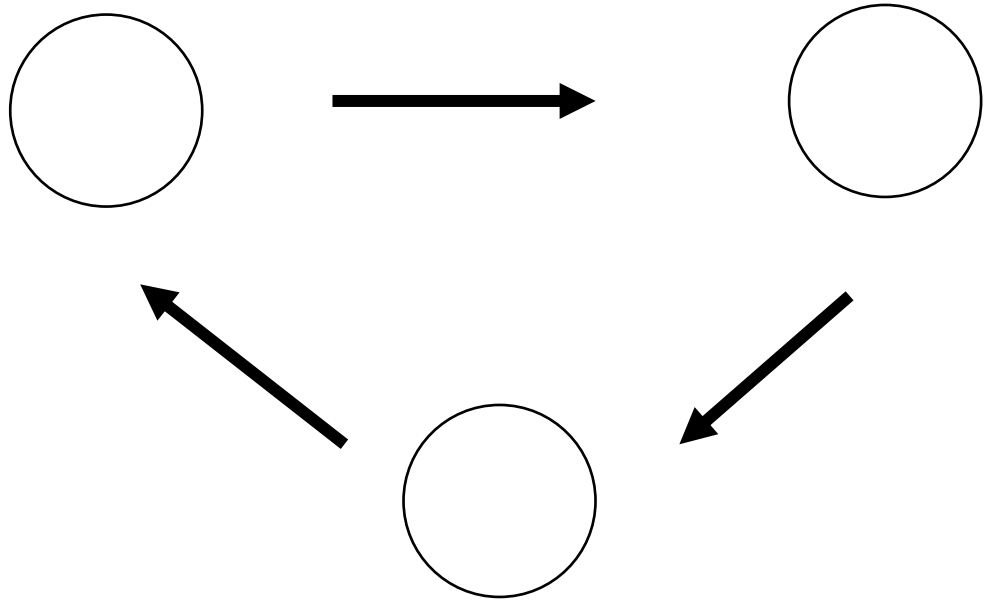
- Graph? yes
- Tree? yes

Examples of Nodes/Vertices and Edges



- Graph?
- Tree?

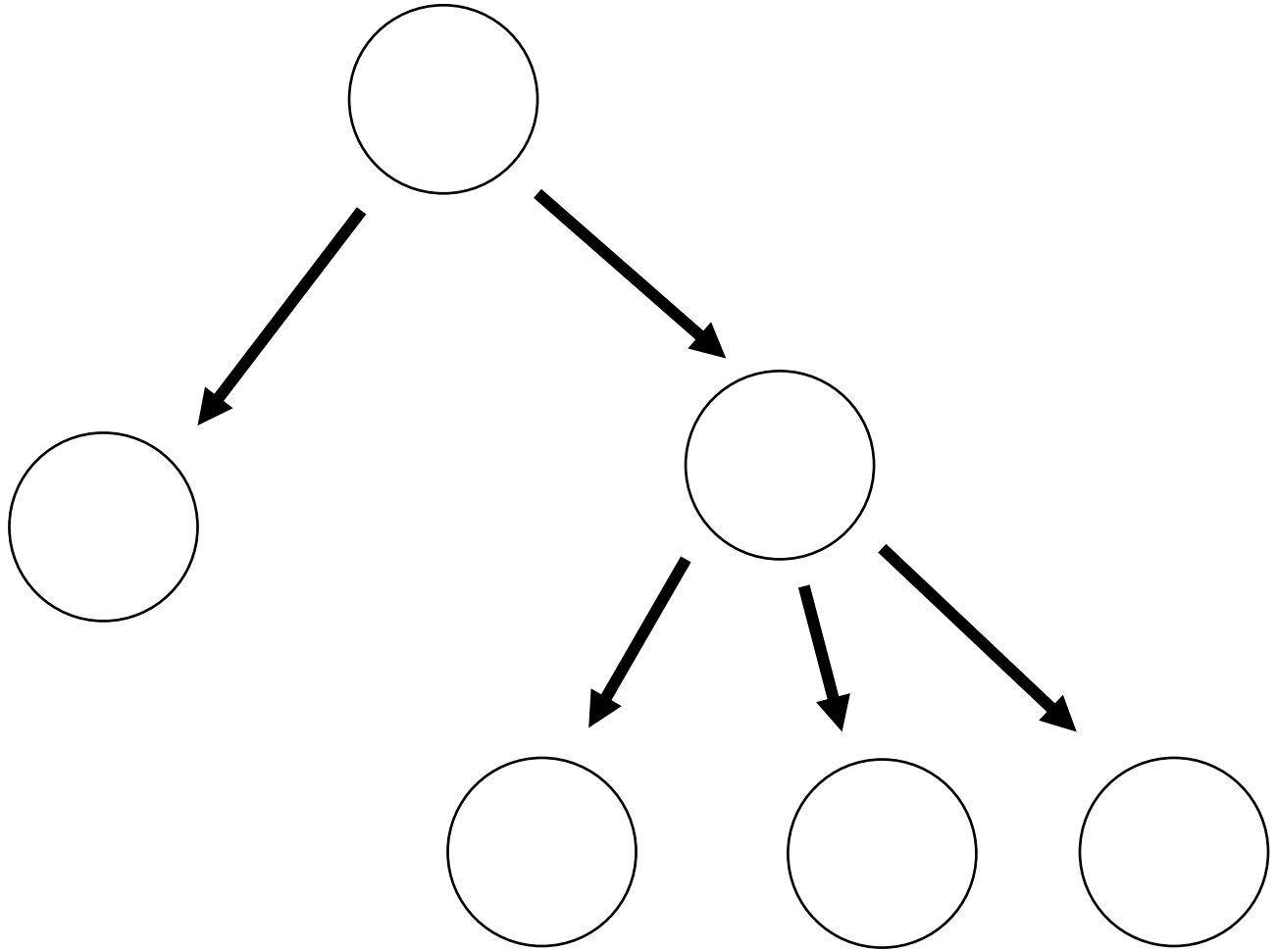
Examples of Nodes/Vertices and Edges



- Graph? yes
- Tree? no

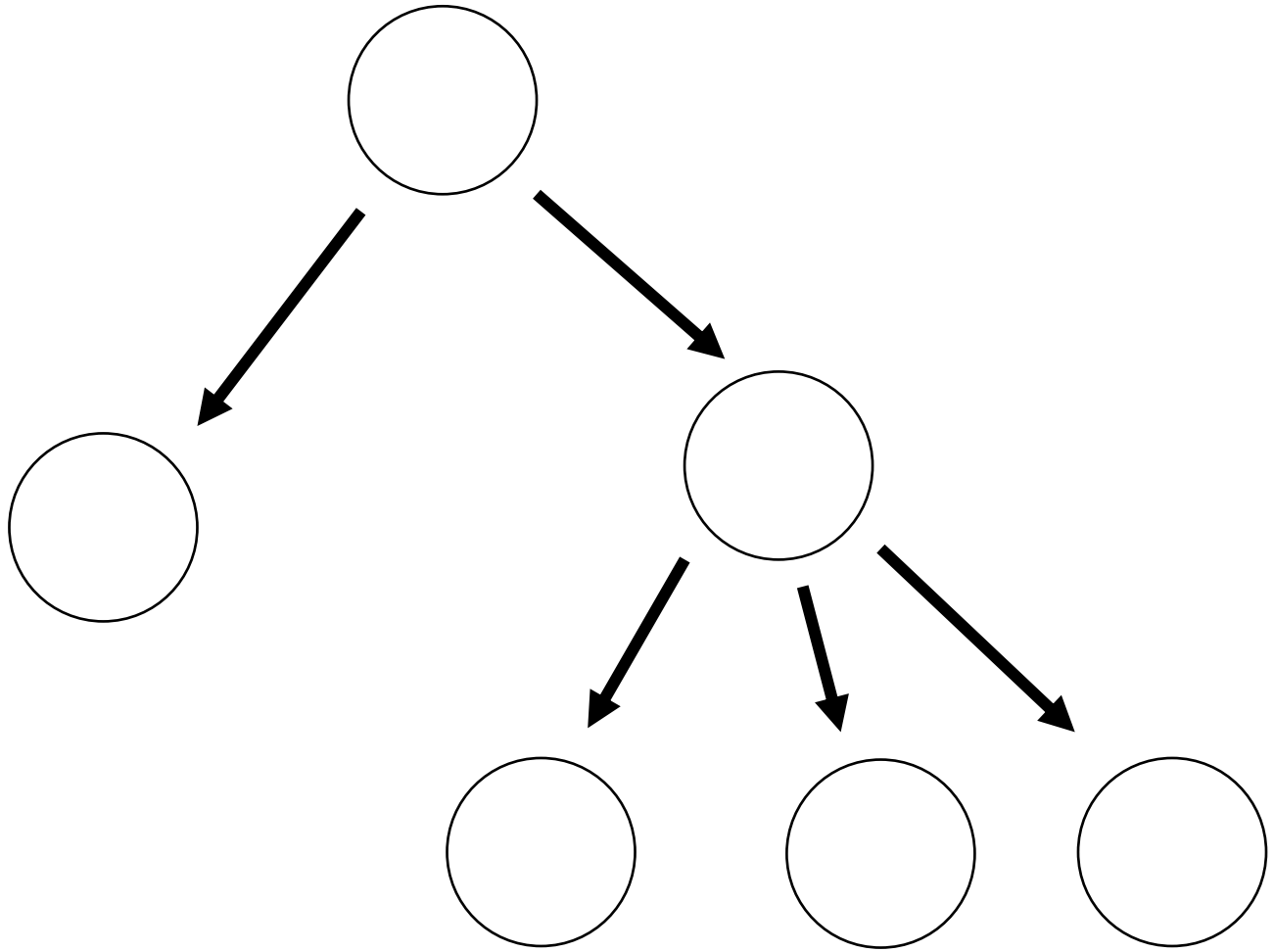
Examples of Nodes/Vertices and Edges

- Graph?
- Tree?



Examples of Nodes/Vertices and Edges

- Graph? yes
- Tree? yes



Terminology

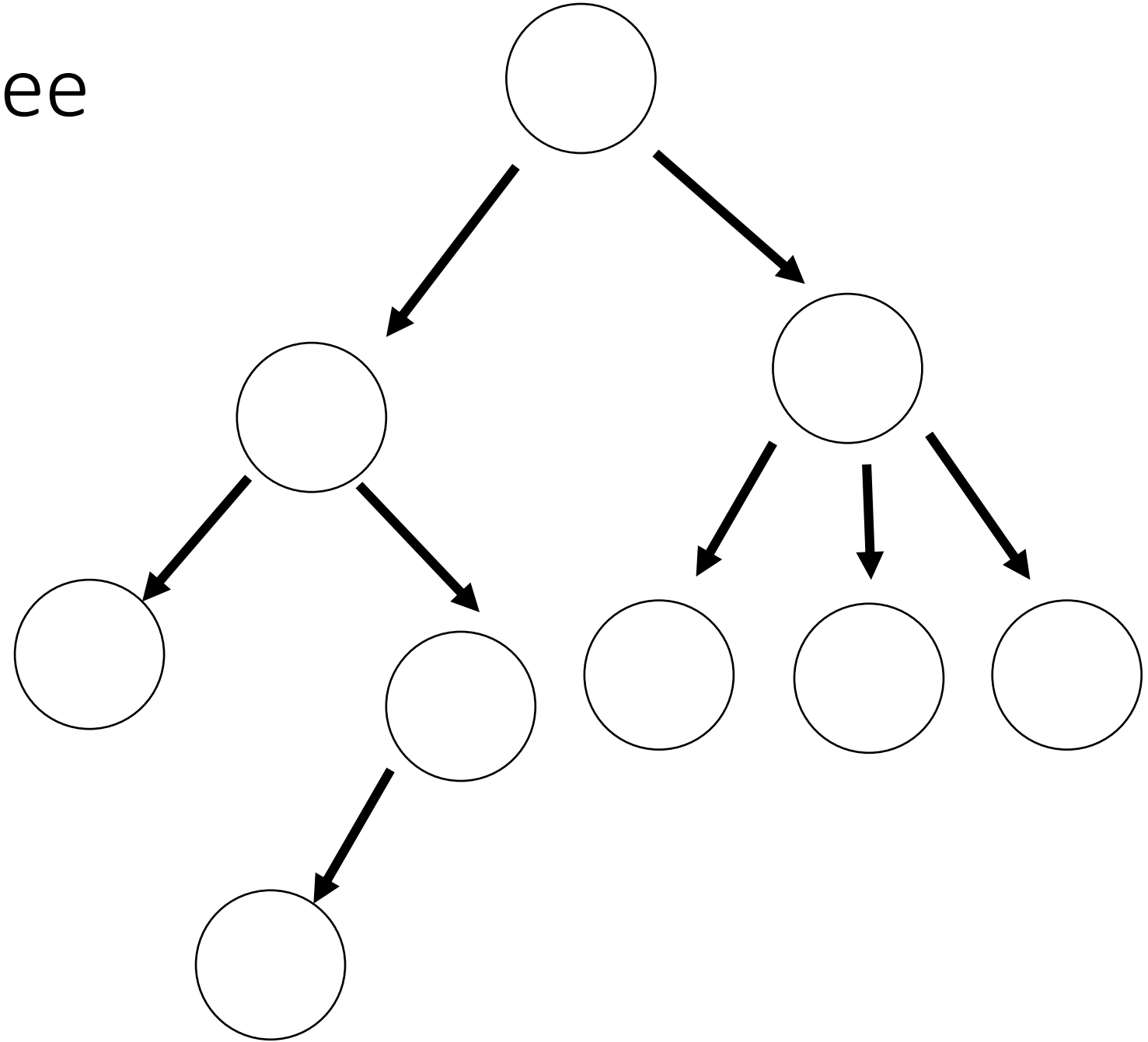
- Tree: a graph with no cycles

- Parent: a node above the current node in a hierarchy
- Child: a node below the current node in a hierarchy

- Leaf: a node with no children
- Interior node: a node with a parent and a child
- Root: the top node in a hierarchy (a node with no parent)

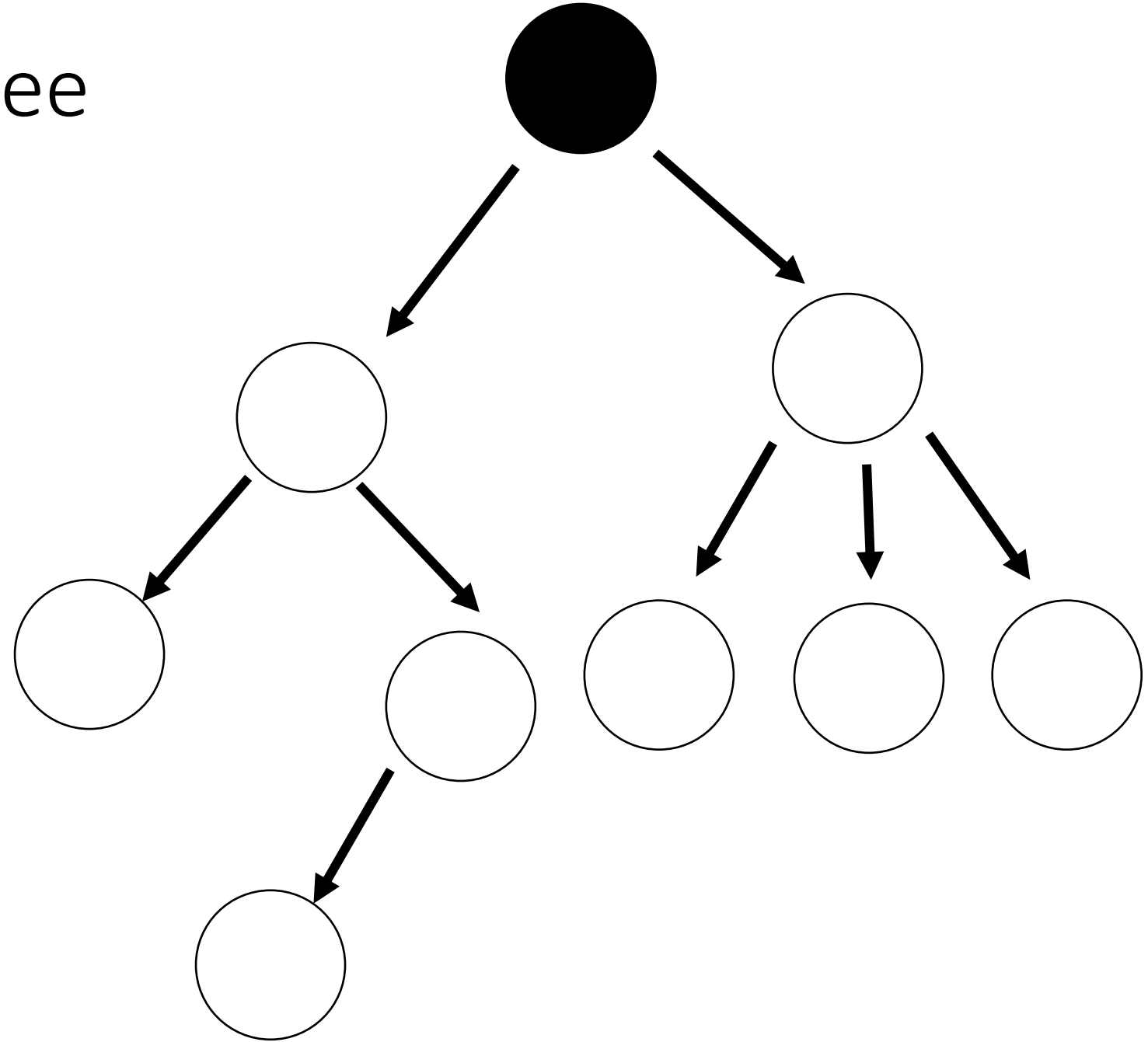
- Sibling: a node with the same parent as the current node

Example Tree



Example Tree

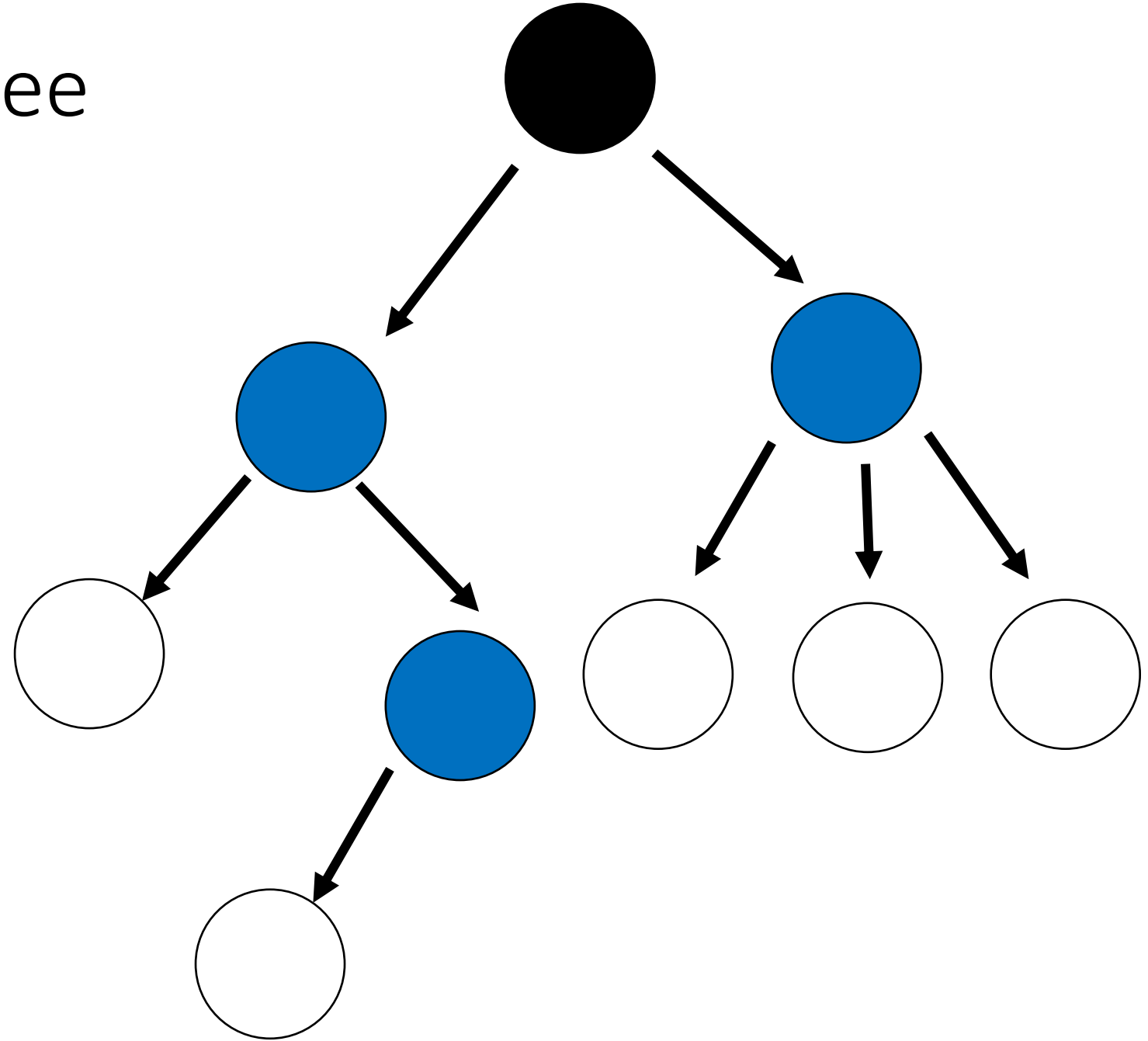
Root Node



Example Tree

Root Node

Interior Node

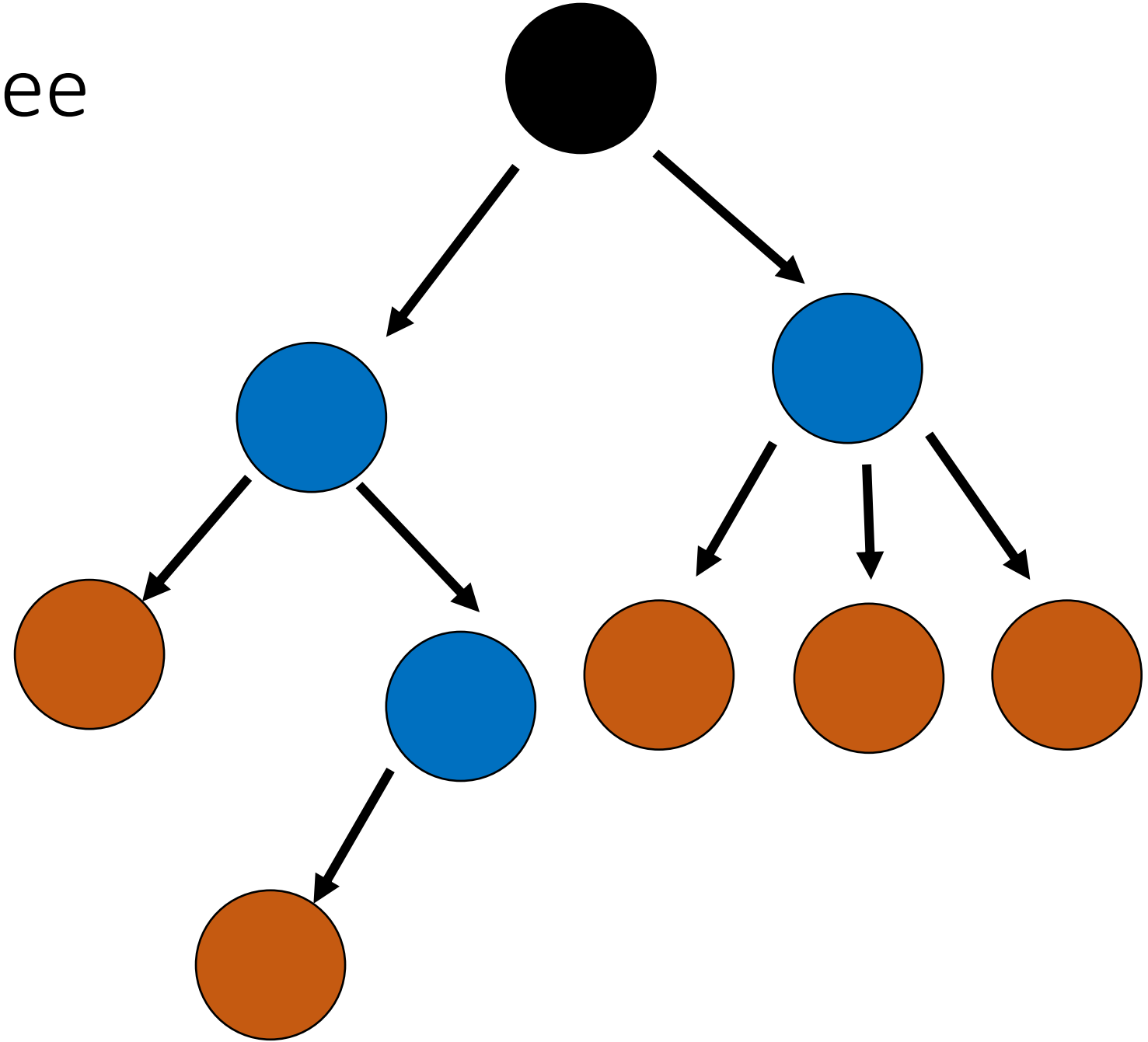


Example Tree

Root Node

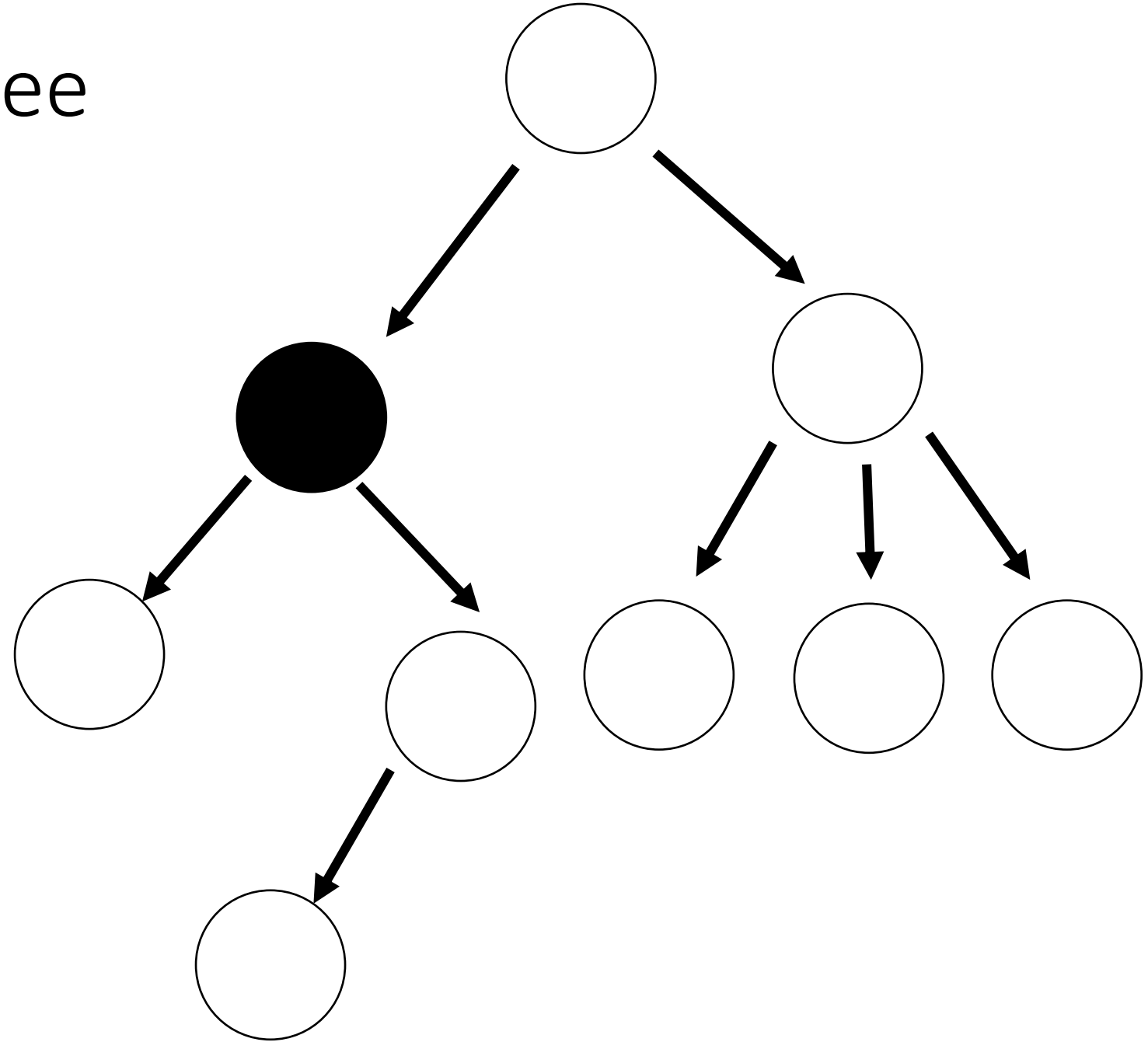
Interior Node

Leaf Node



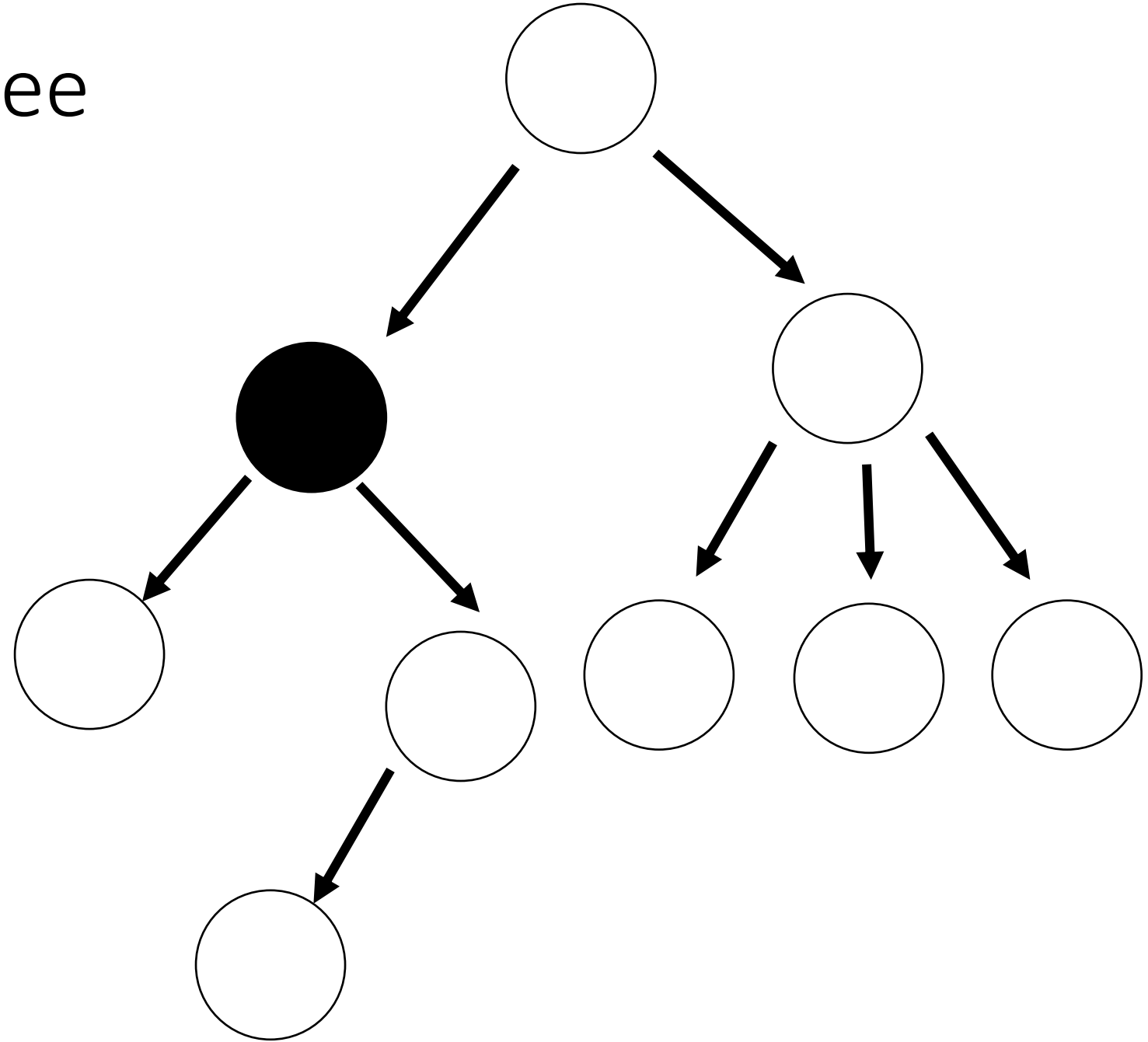
Example Tree

Parent?



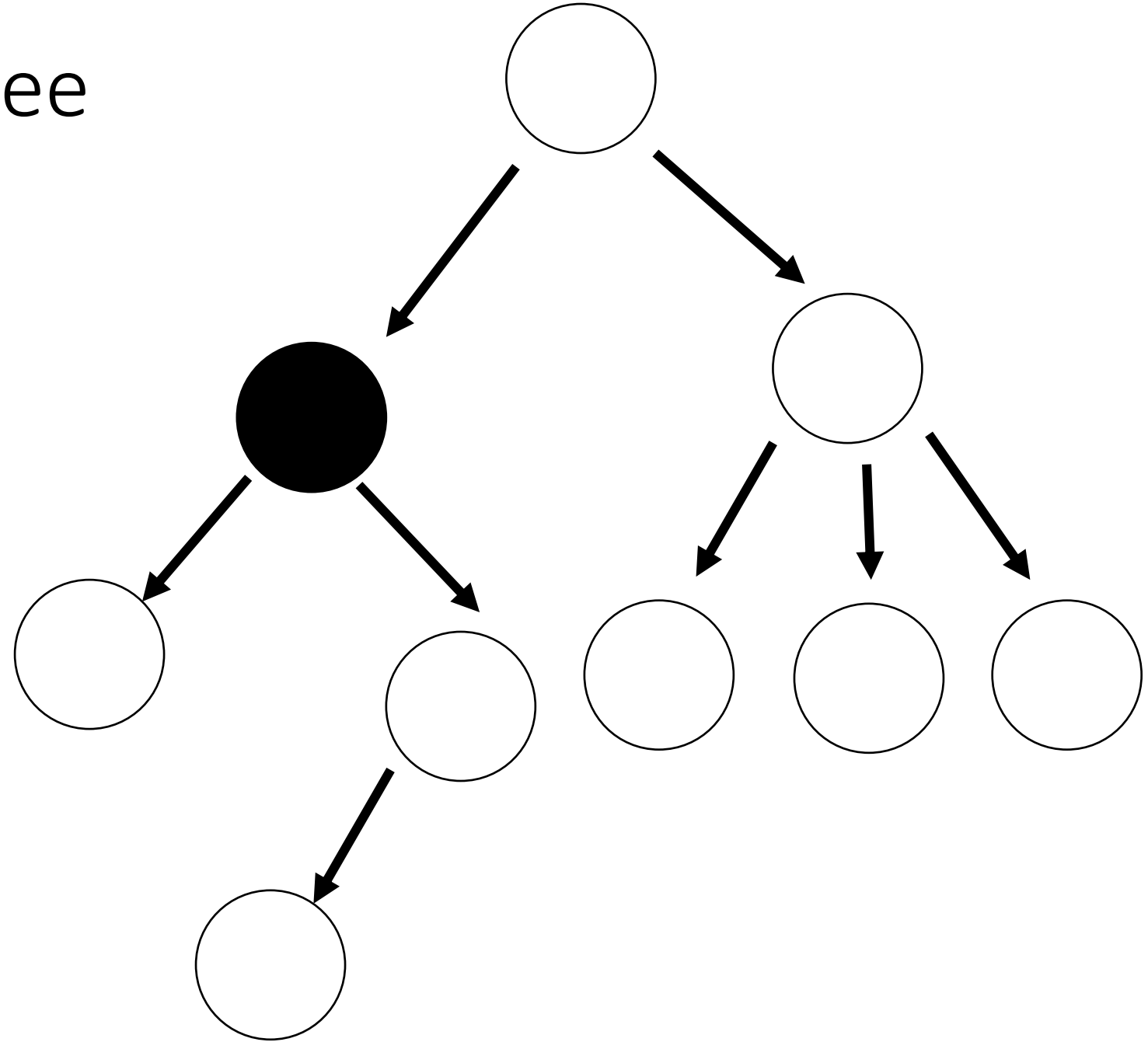
Example Tree

Parent? yes
Child? yes



Example Tree

Parent? yes
Child? yes
Sibling? yes



Terminology

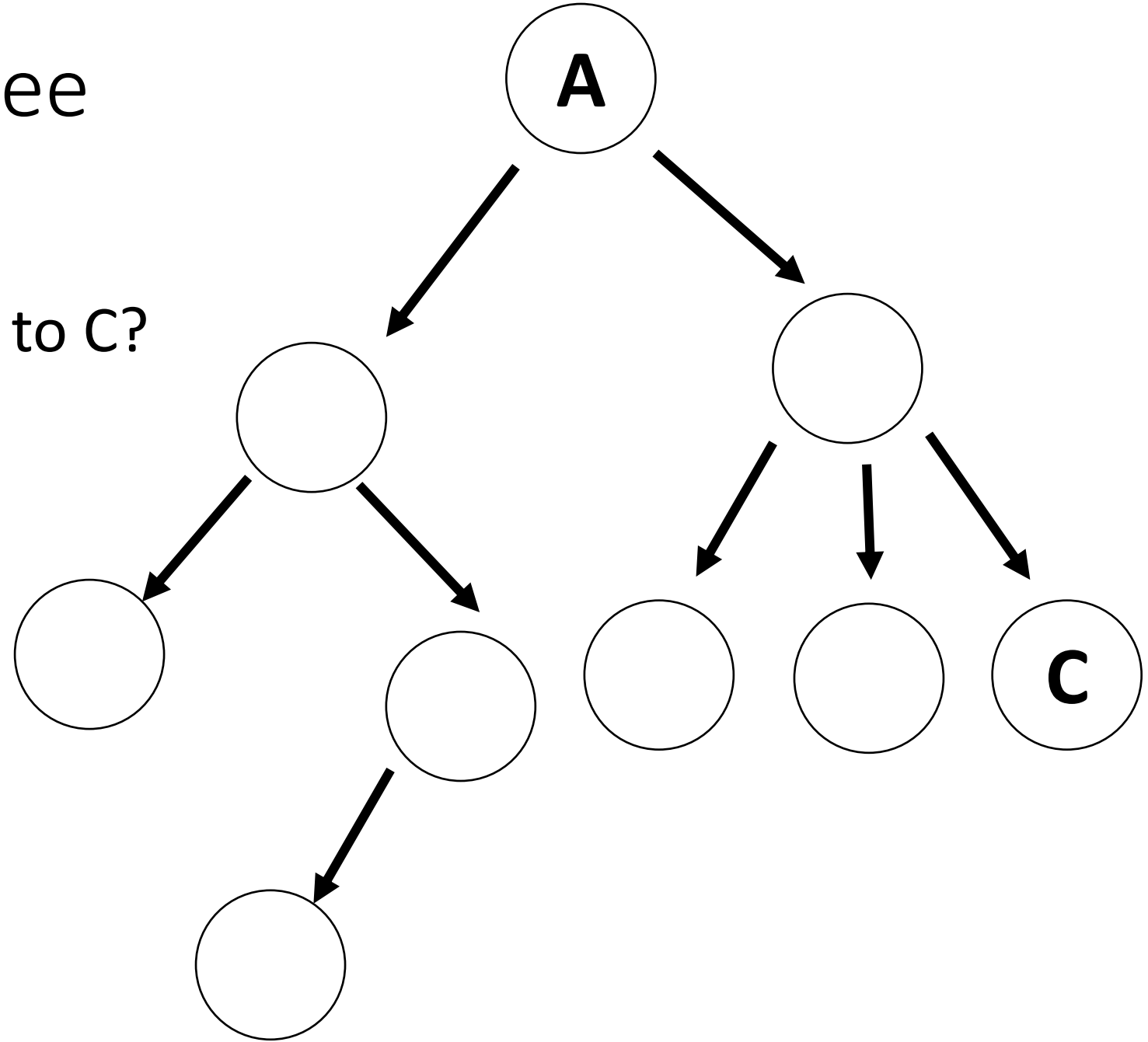
- Tree: a graph with no cycles

- Path length: the number of edges from the root to the current node
- Height: the length of the path from the root to the deepest node
  - A one-node tree (just a root node) has height 0

- An alternate definition assumes a one-node tree has a height of 1.

Example Tree

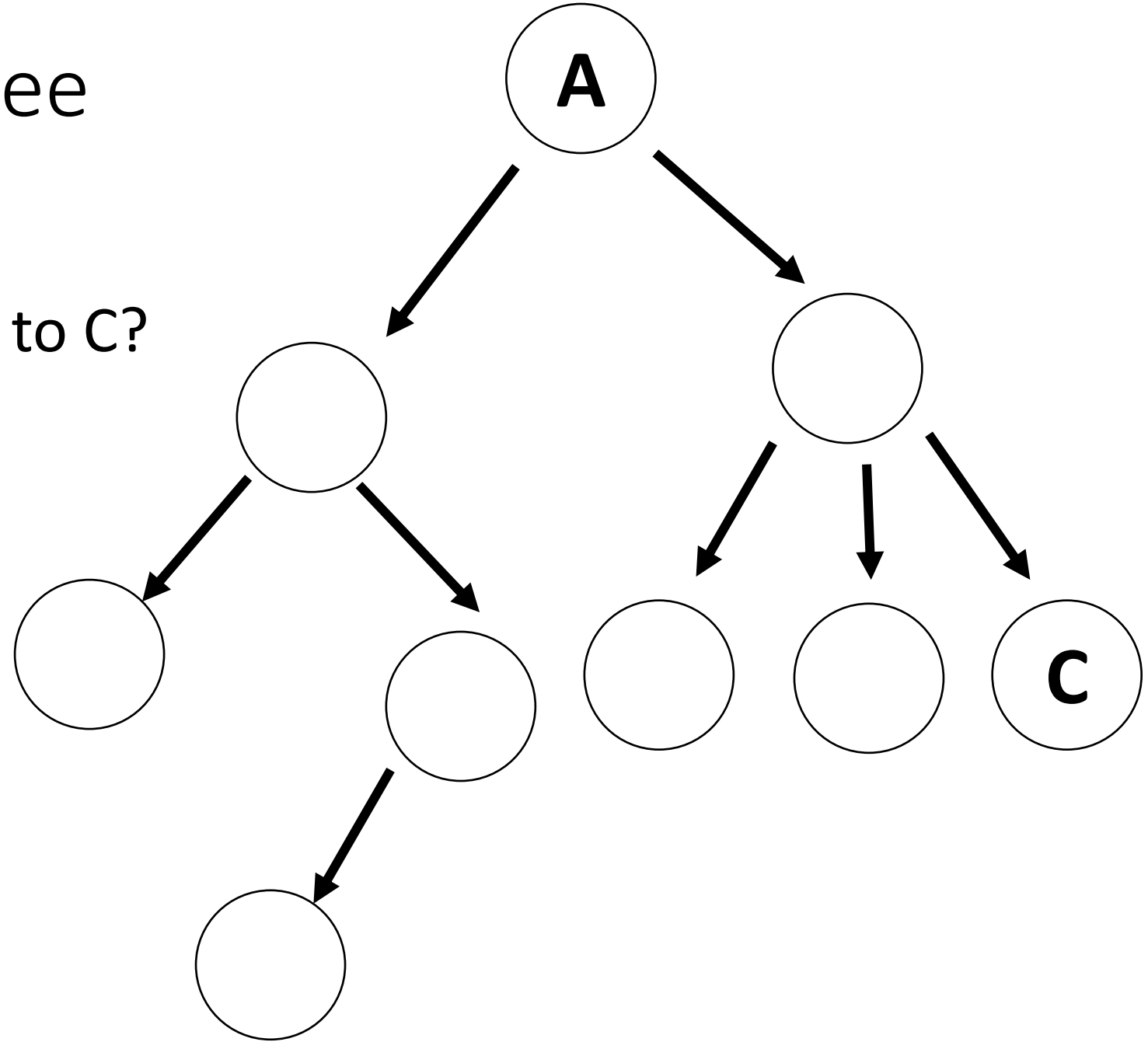
Path length from A to C?



Example Tree

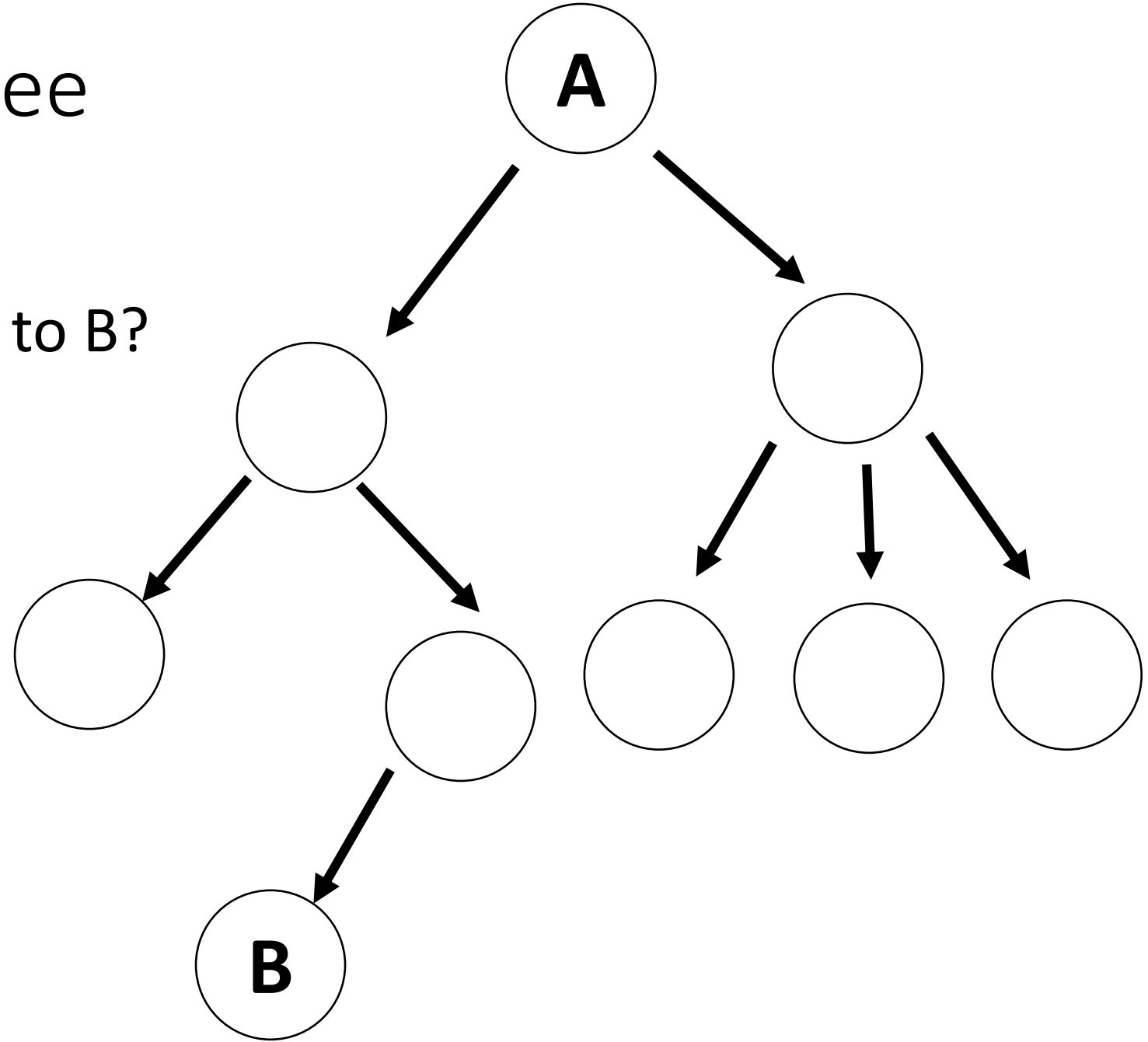
Path length from A to C?

2



Example Tree

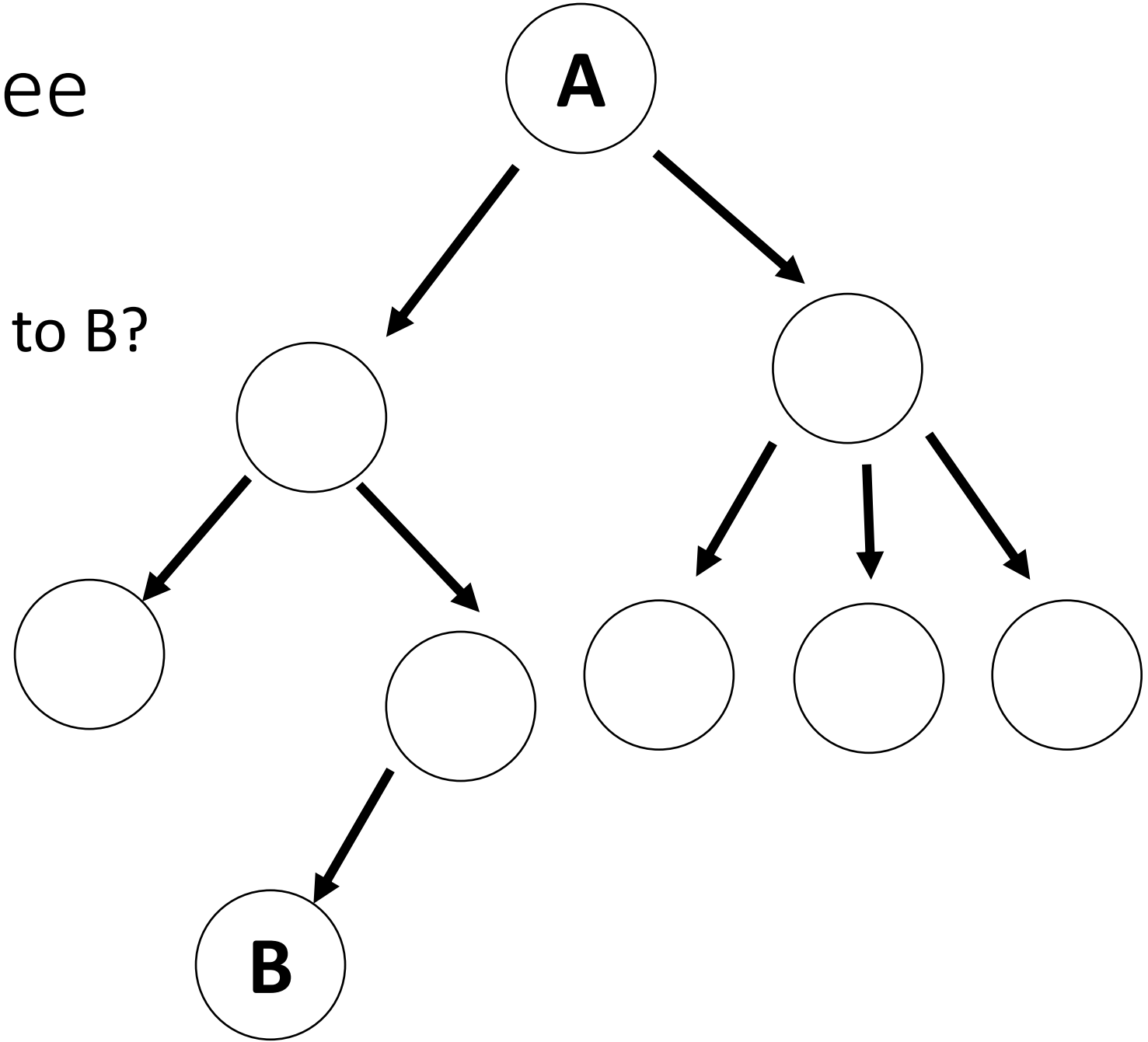
Path length from A to B?



Example Tree

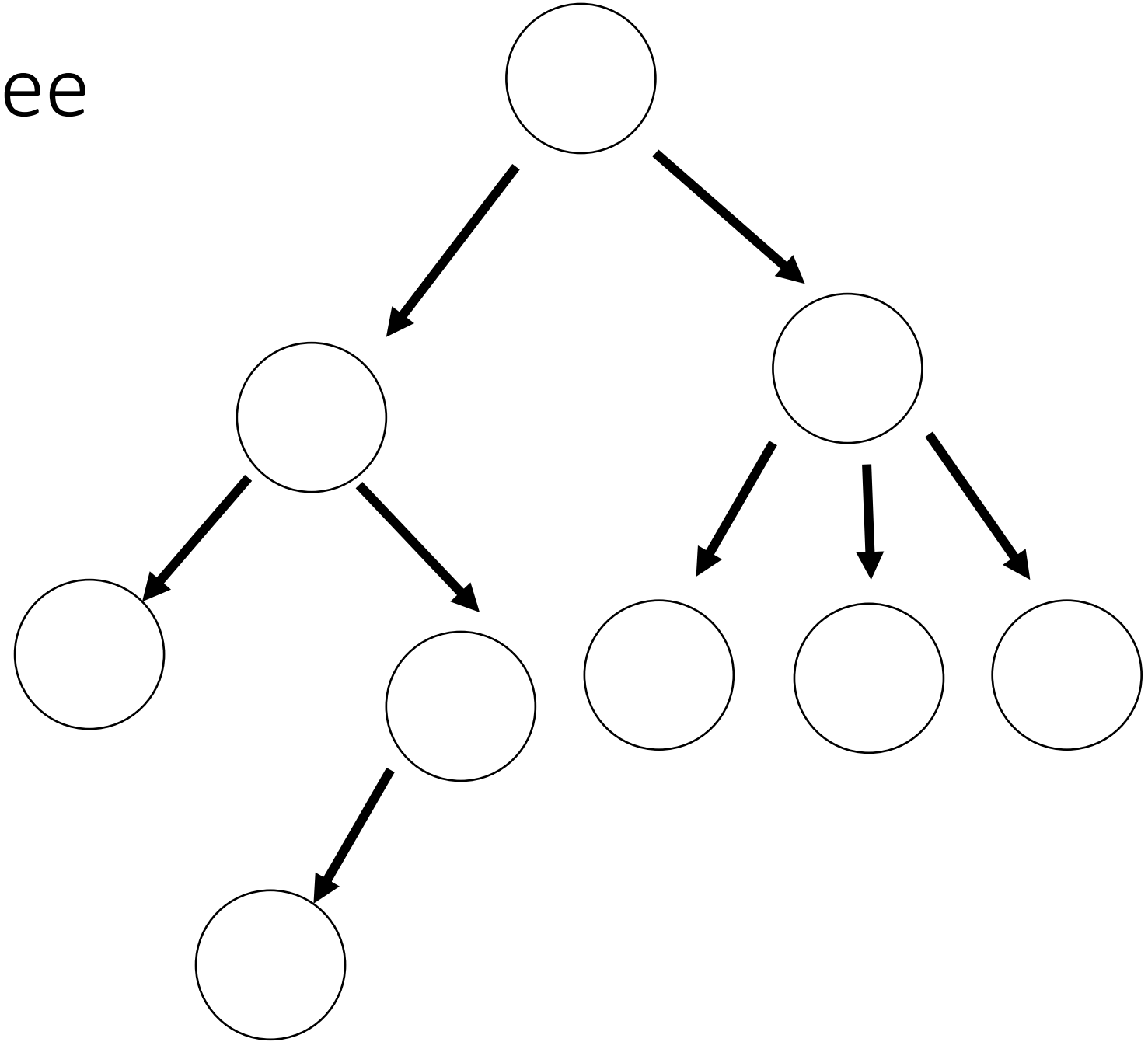
Path length from A to B?

3



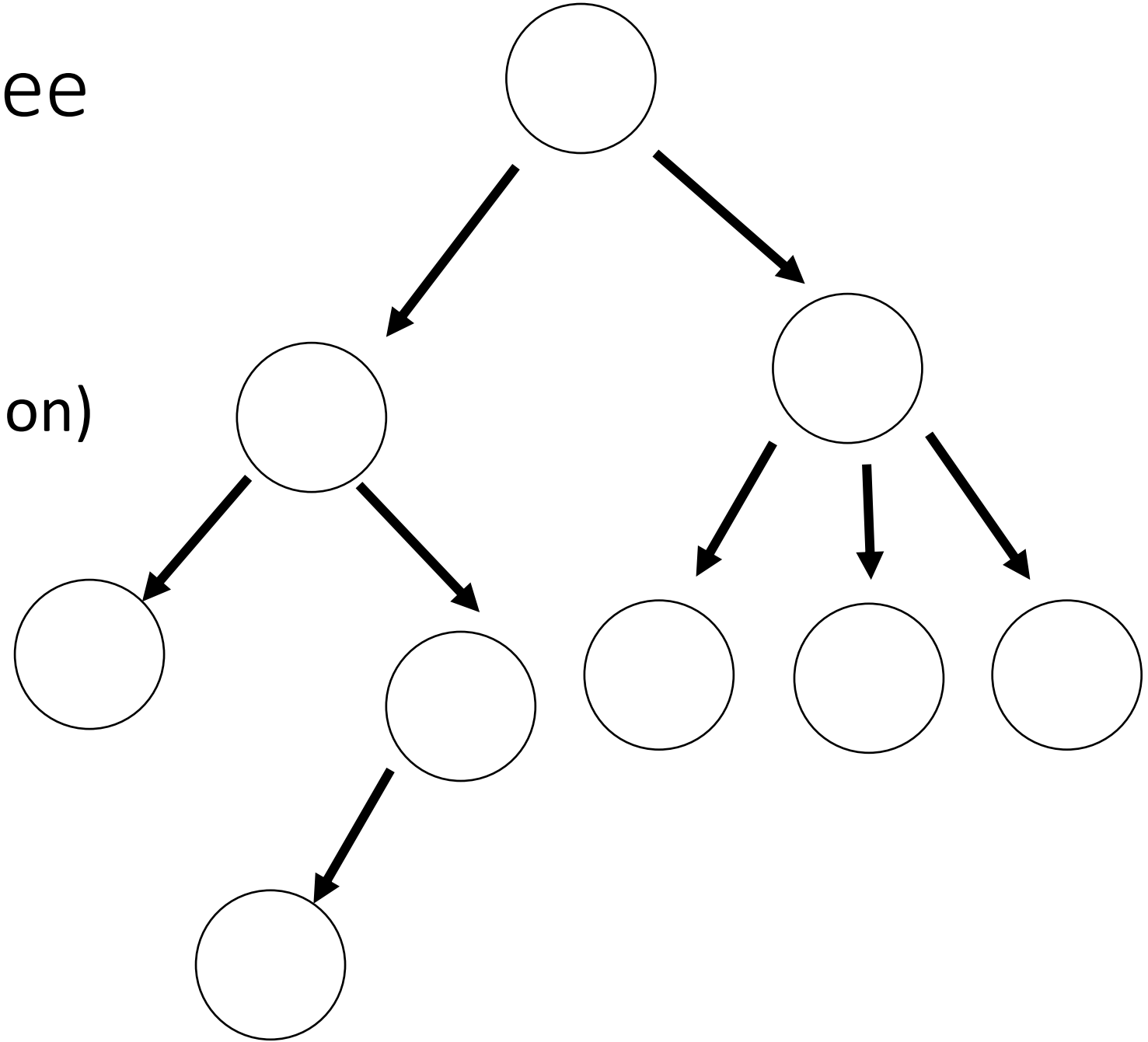
Example Tree

Height?

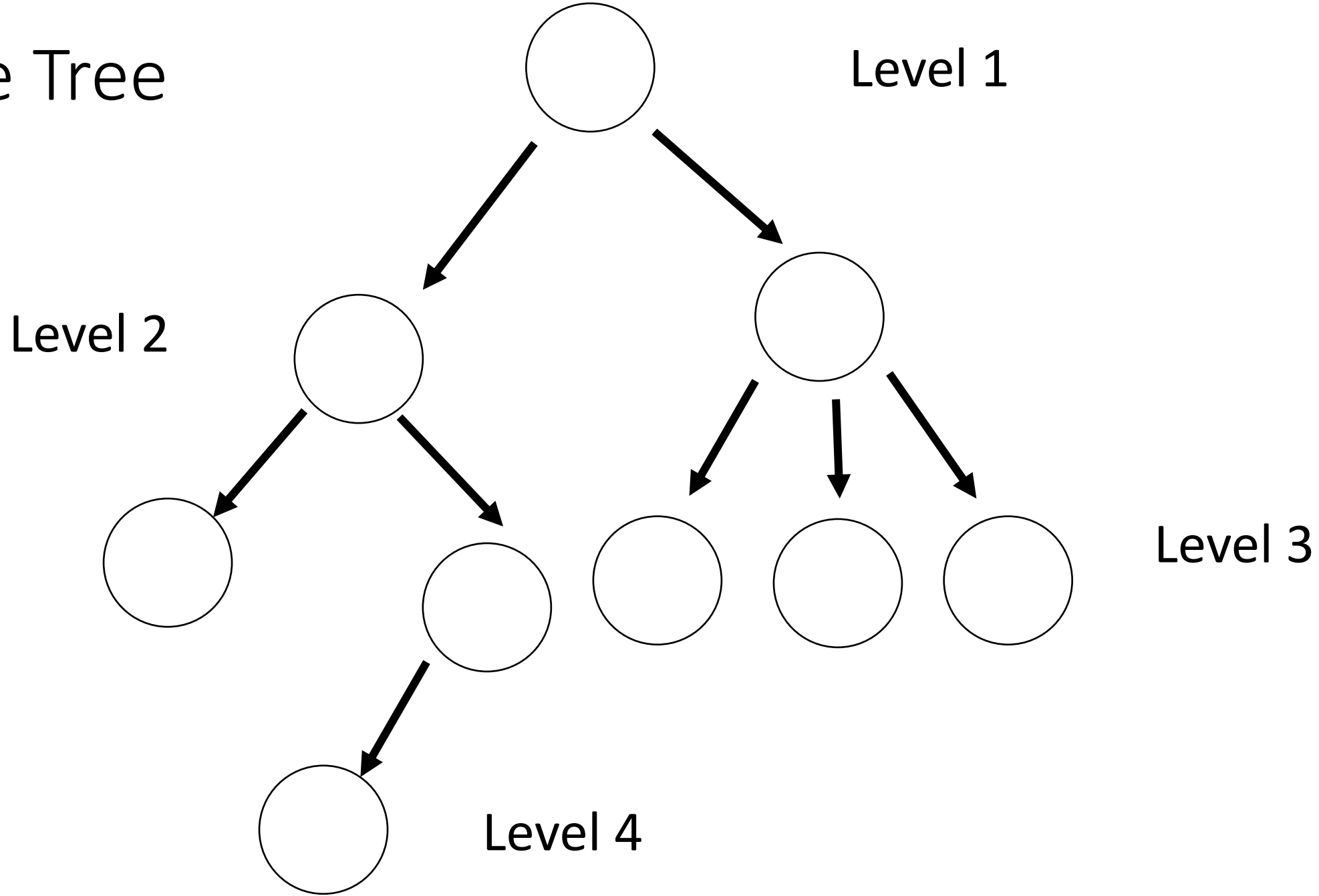


Example Tree

Height?
3 (with first definition)



Example Tree



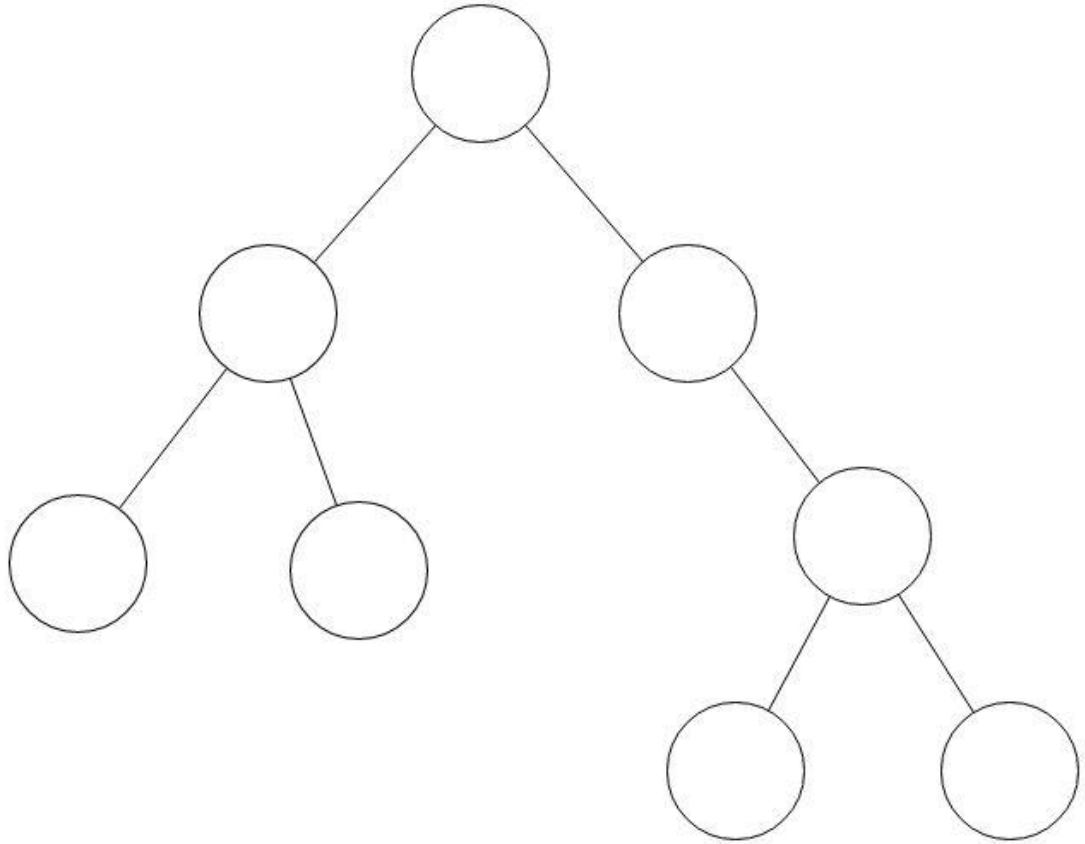
BINARY TREES

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.

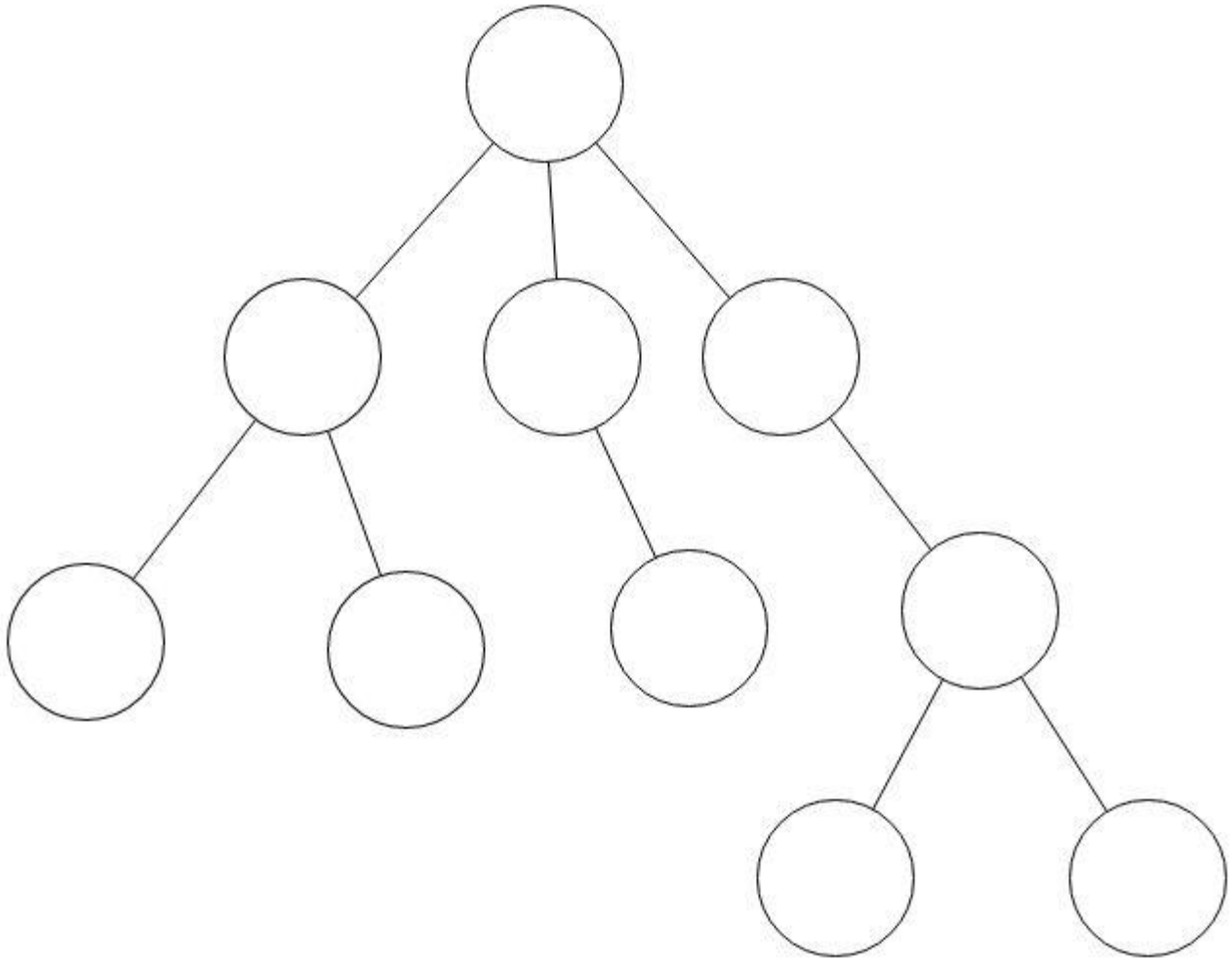
Binary Trees

- A binary tree is a special tree such that each node has **at most two children**.



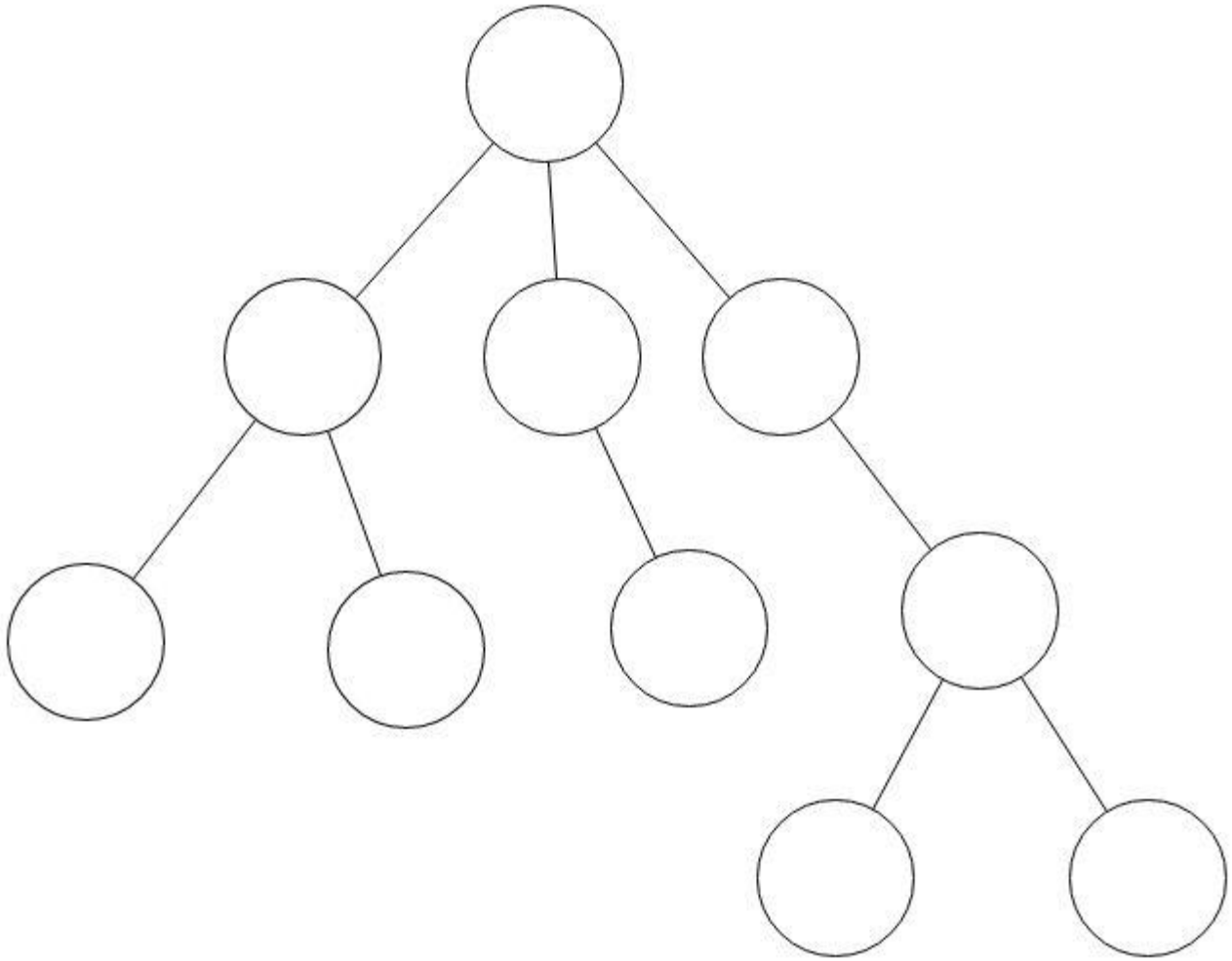
Examples of Nodes/Vertices and Edges

- Graph?
- Tree?
- Binary Tree?



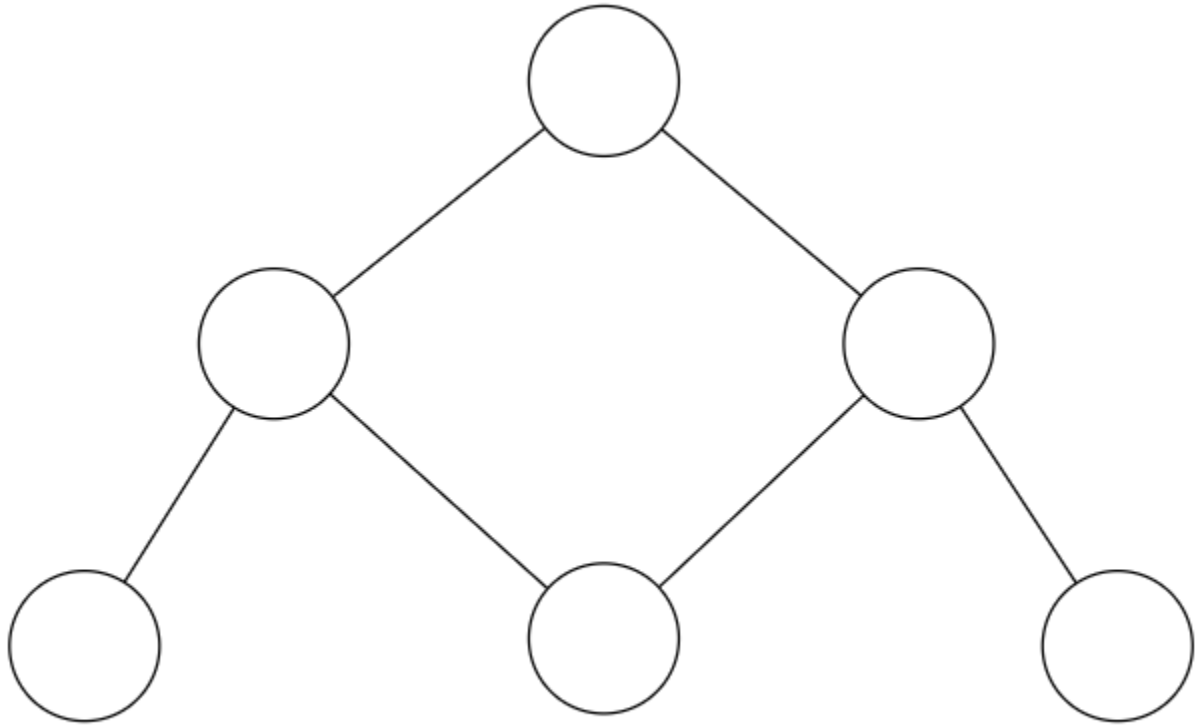
Examples of Nodes/Vertices and Edges

- Graph? yes
- Tree? yes
- Binary Tree? no



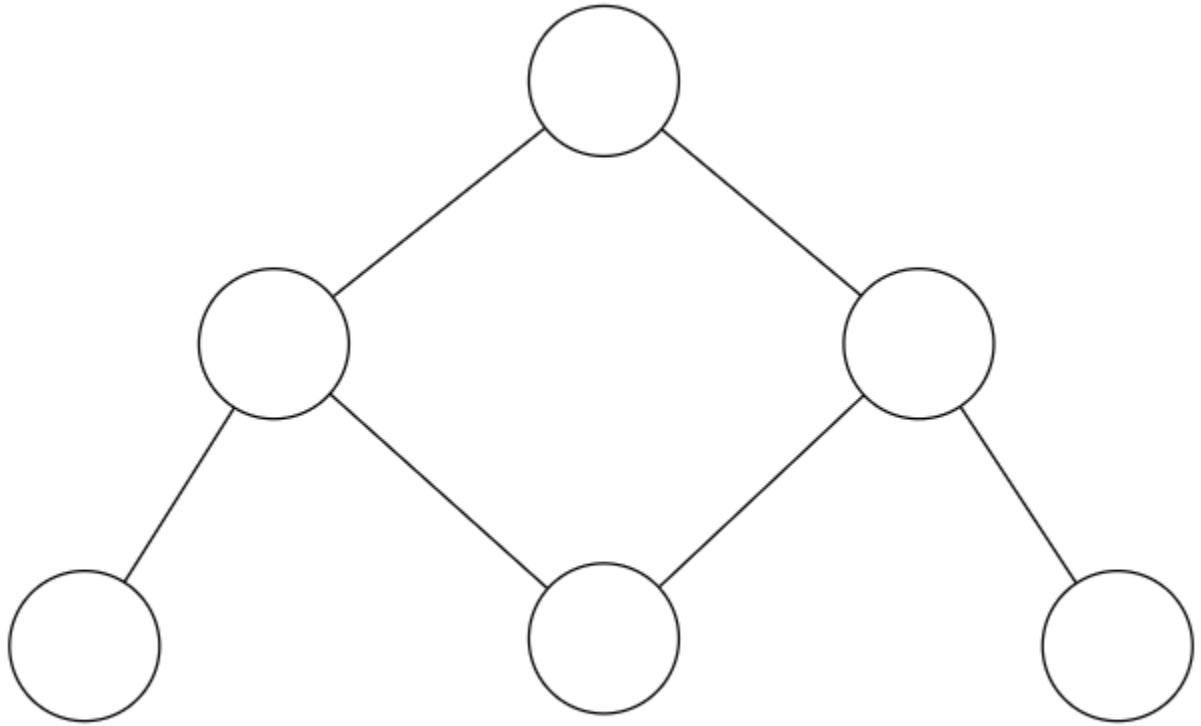
Examples of Nodes/Vertices and Edges

- Graph?
- Tree?
- Binary Tree?



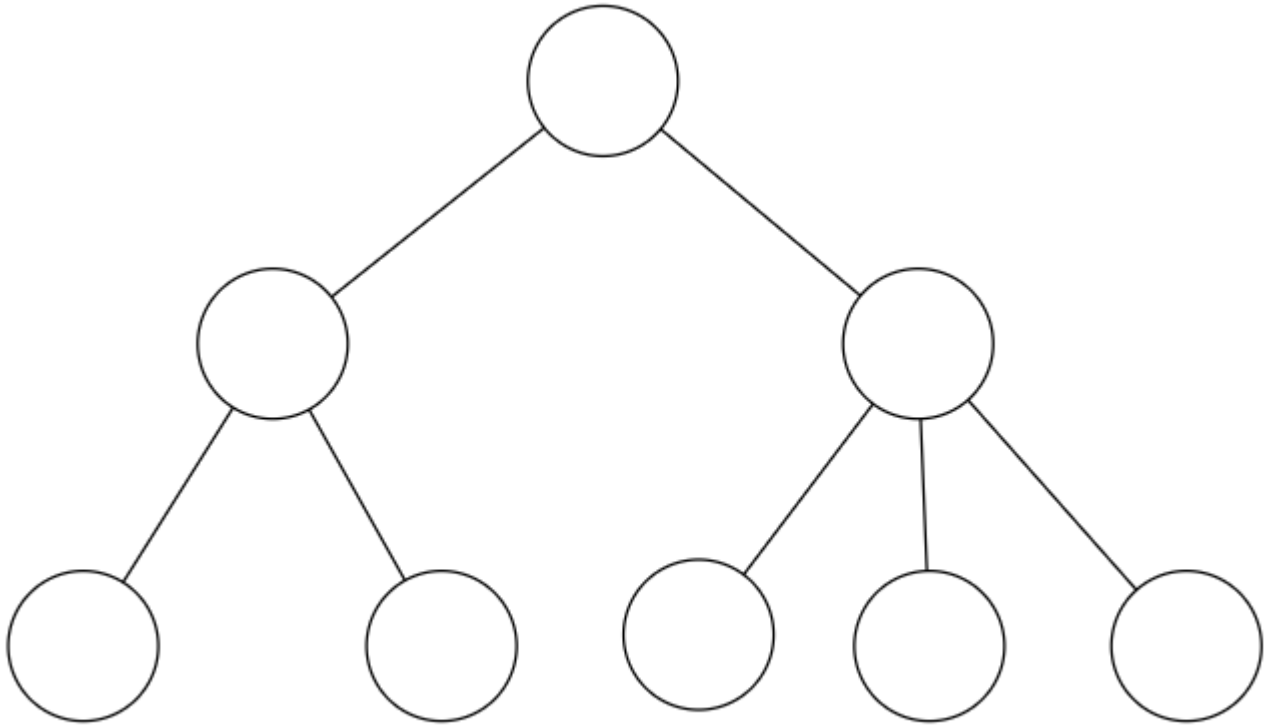
Examples of Nodes/Vertices and Edges

- Graph? yes
- Tree? no
- Binary Tree? no



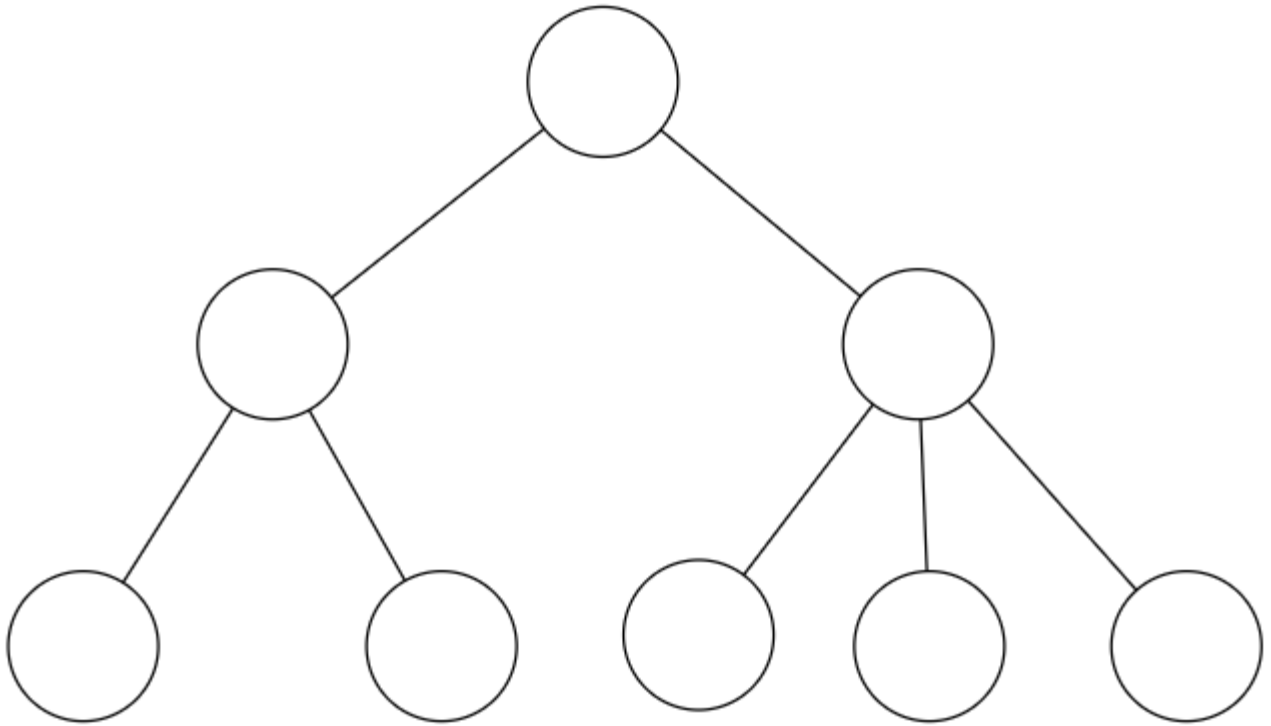
Examples of Nodes/Vertices and Edges

- Graph?
- Tree?
- Binary Tree?



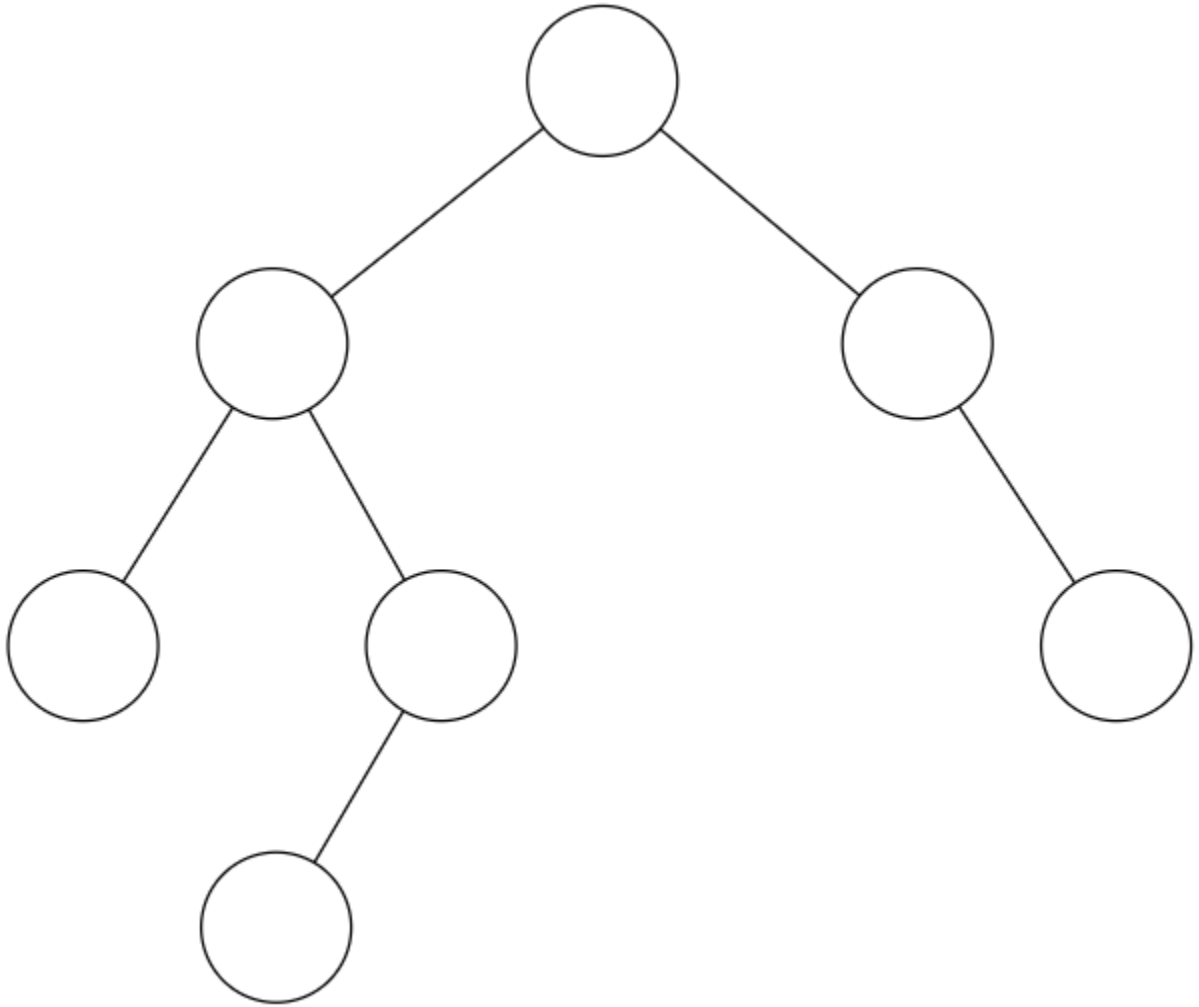
Examples of Nodes/Vertices and Edges

- Graph? yes
- Tree? yes
- Binary Tree? no



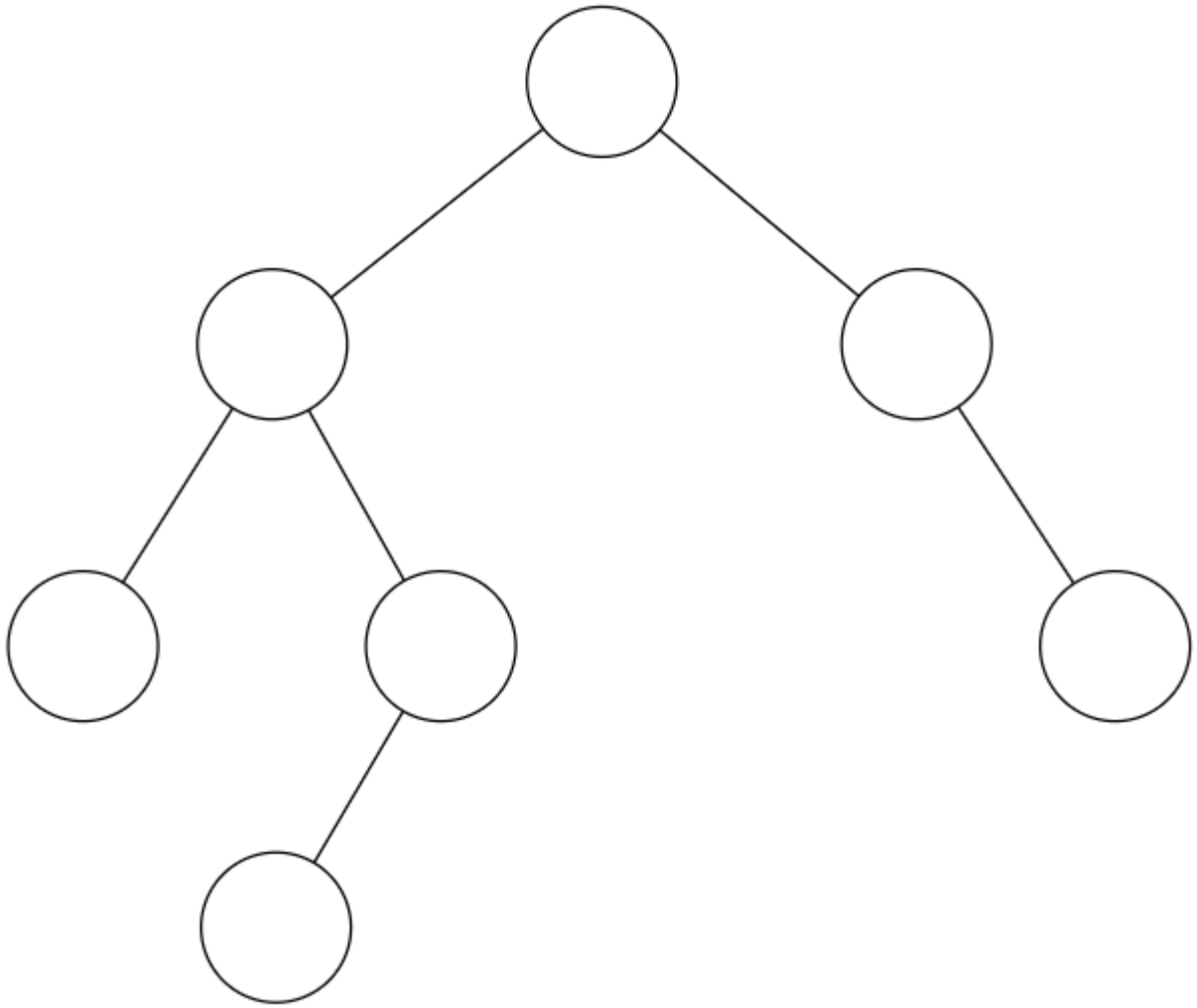
Examples of Nodes/Vertices and Edges

- Graph?
- Tree?
- Binary Tree?



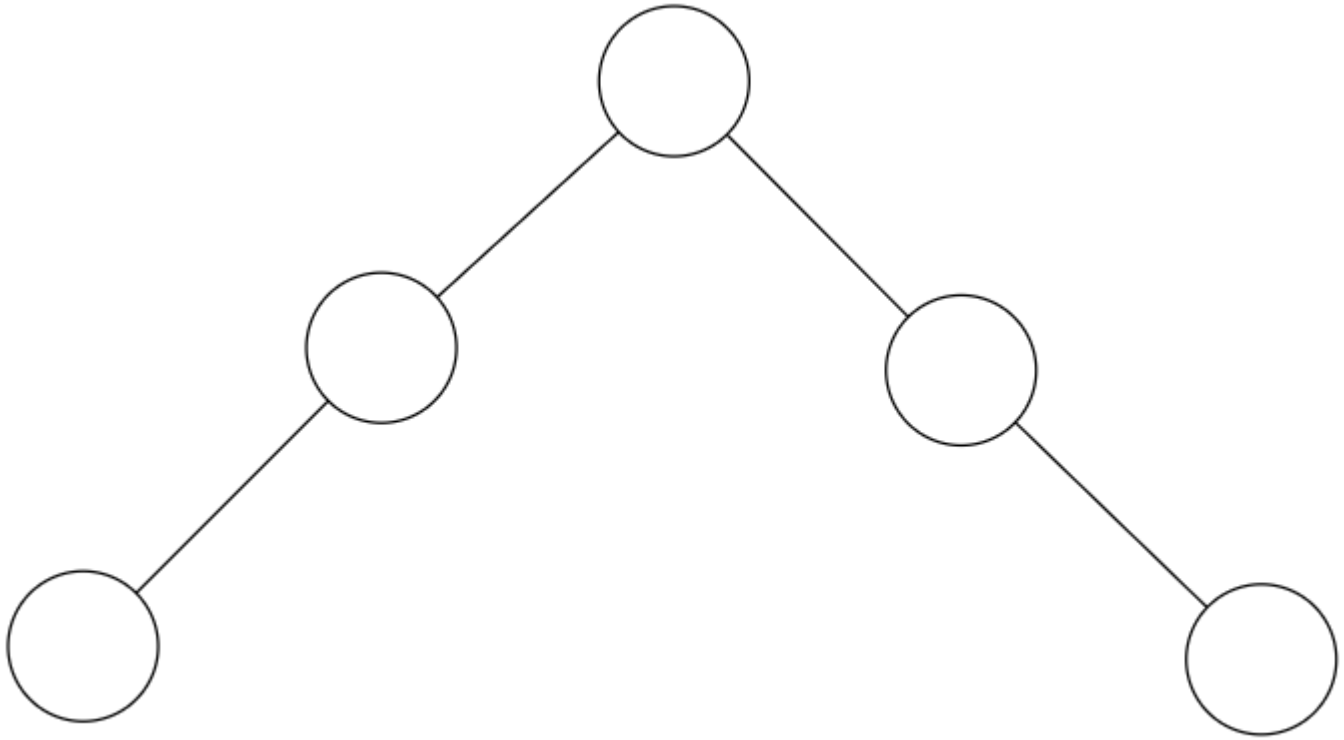
Examples of Nodes/Vertices and Edges

- Graph? yes
- Tree? yes
- Binary Tree? yes



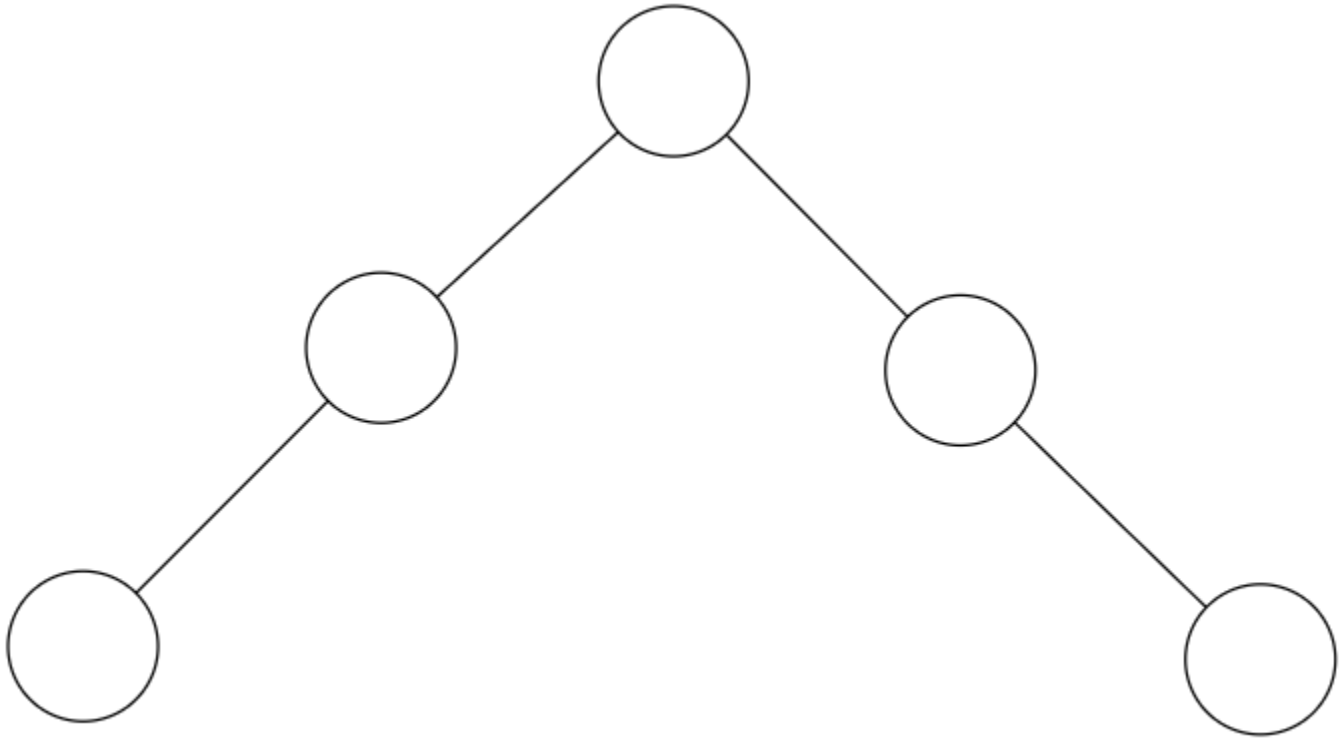
Examples of Nodes/Vertices and Edges

- Graph?
- Tree?
- Binary Tree?



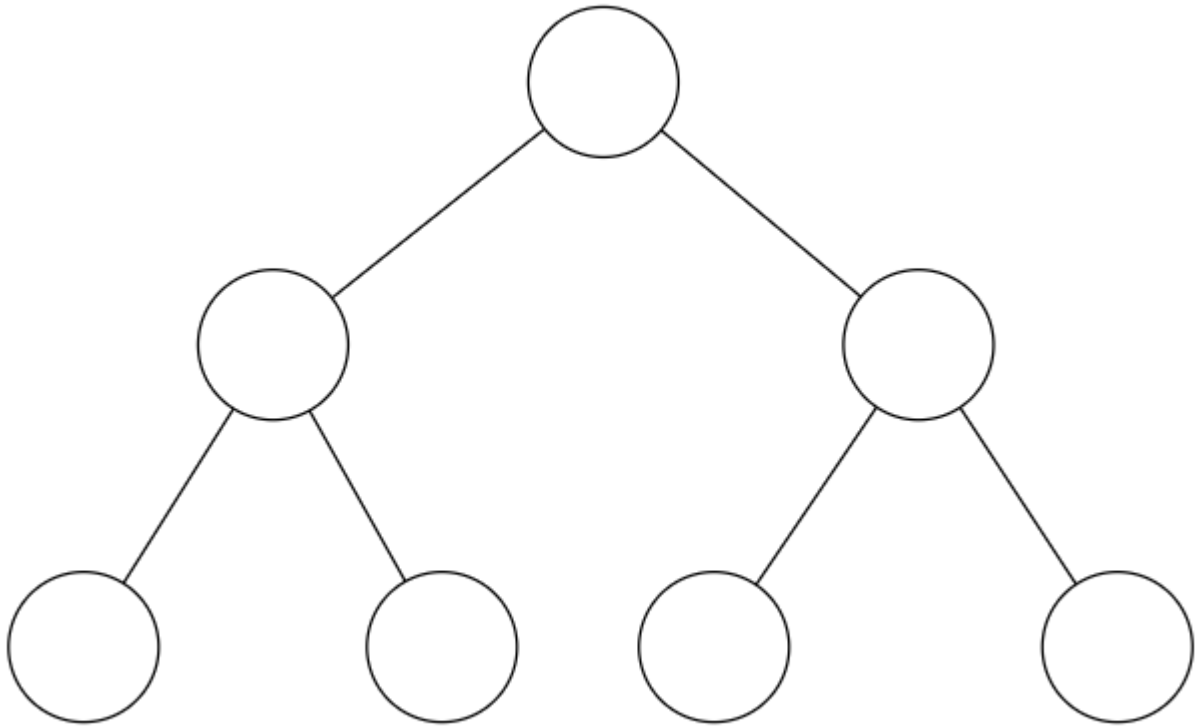
Examples of Nodes/Vertices and Edges

- Graph? yes
- Tree? yes
- Binary Tree? yes



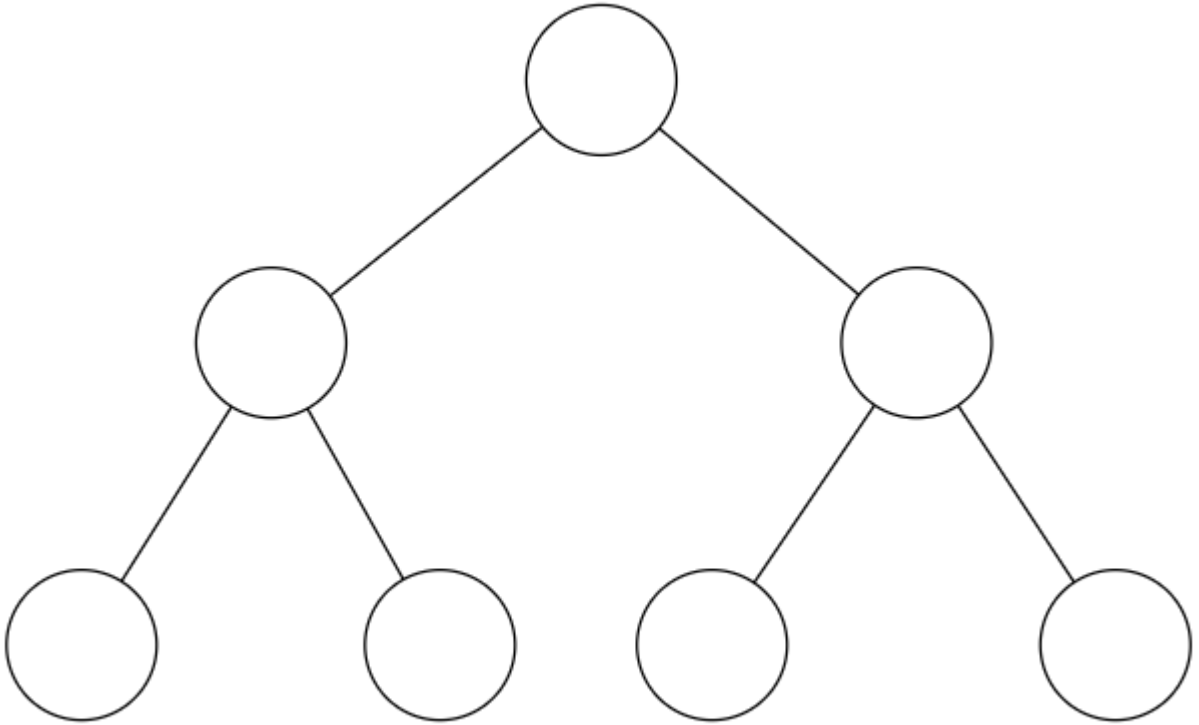
Examples of Nodes/Vertices and Edges

- Graph?
- Tree?
- Binary Tree?



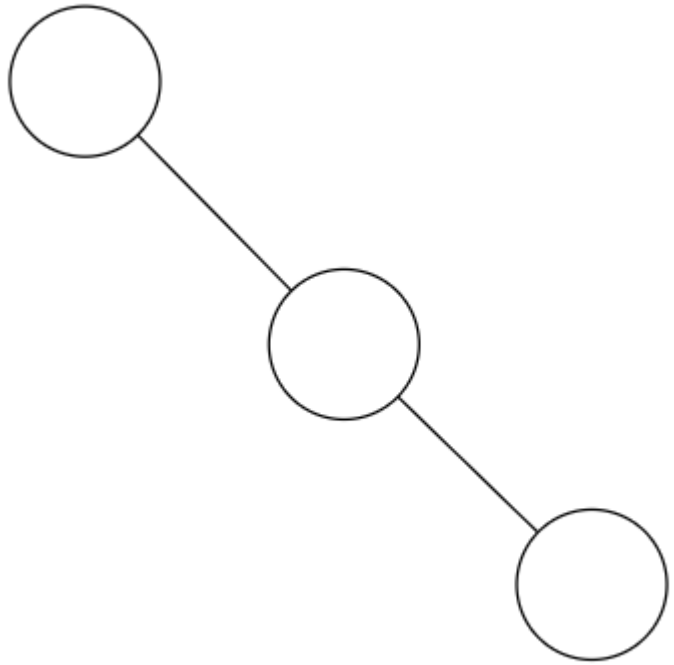
Examples of Nodes/Vertices and Edges

- Graph? yes
- Tree? yes
- Binary Tree? yes



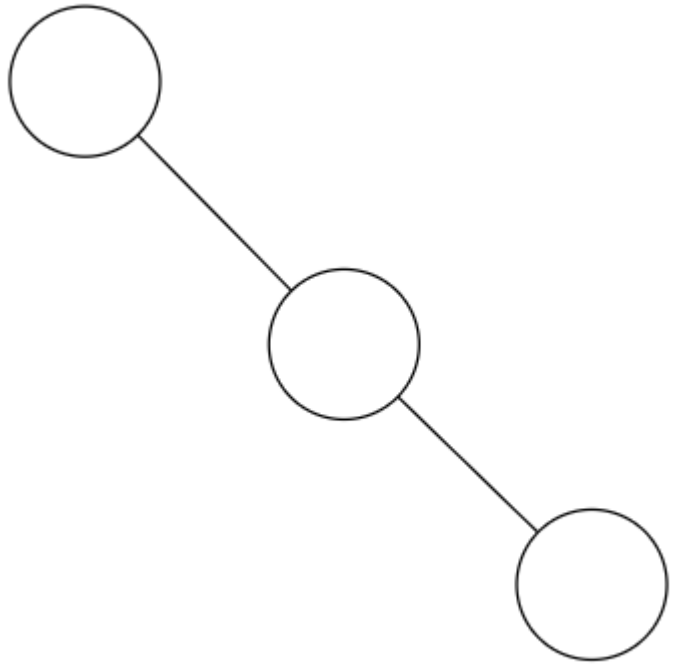
Examples of Nodes/Vertices and Edges

- Graph?
- Tree?
- Binary Tree?



Examples of Nodes/Vertices and Edges

- Graph? yes
- Tree? yes
- Binary Tree? yes



Terminology

- A binary tree is a tree such that each node has at most two children.

Terminology

- Full: a binary tree such that every non-leaf has **exactly** two children **and** every leaf is on the same level.

Alternate Definition:

- Full: a binary tree such that every non-leaf has **exactly** two children.
  - In other words, each node has either 0 or 2 children.
  - There is no requirement that leaves be on the same level.

- The first definition is more stringent.
  - If a tree is full by the 1st, it's full by the 2nd.
  - If a tree is full by the 2nd, it might not be full by the 1st.

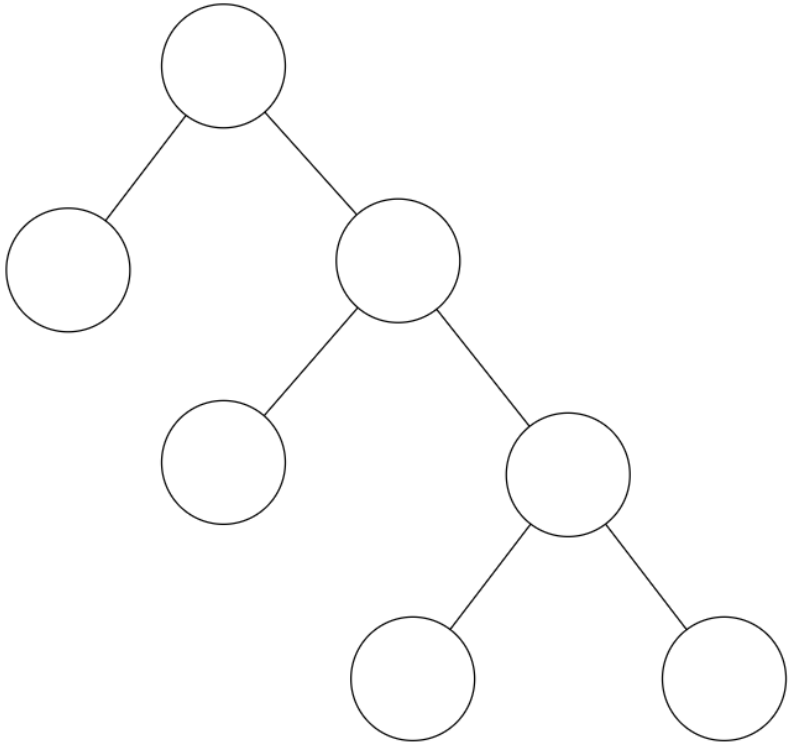
Terminology

- Full: a binary tree such that every non-leaf has **exactly** two children **and** every leaf is on the same level.

using this definition: no, not full

- Alternate Definition of Full: a binary tree such that every non-leaf has **exactly** two children.
  - In other words, each node has either 0 or 2 children.

using this definition: yes, full



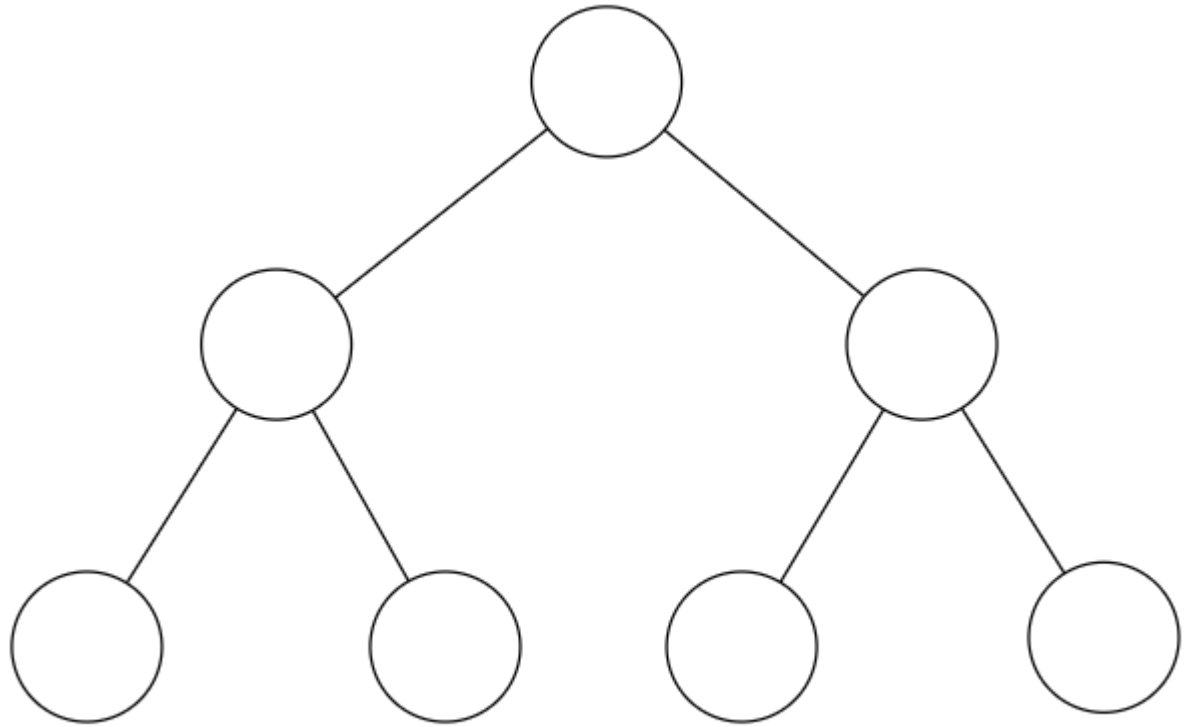
Terminology

- 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

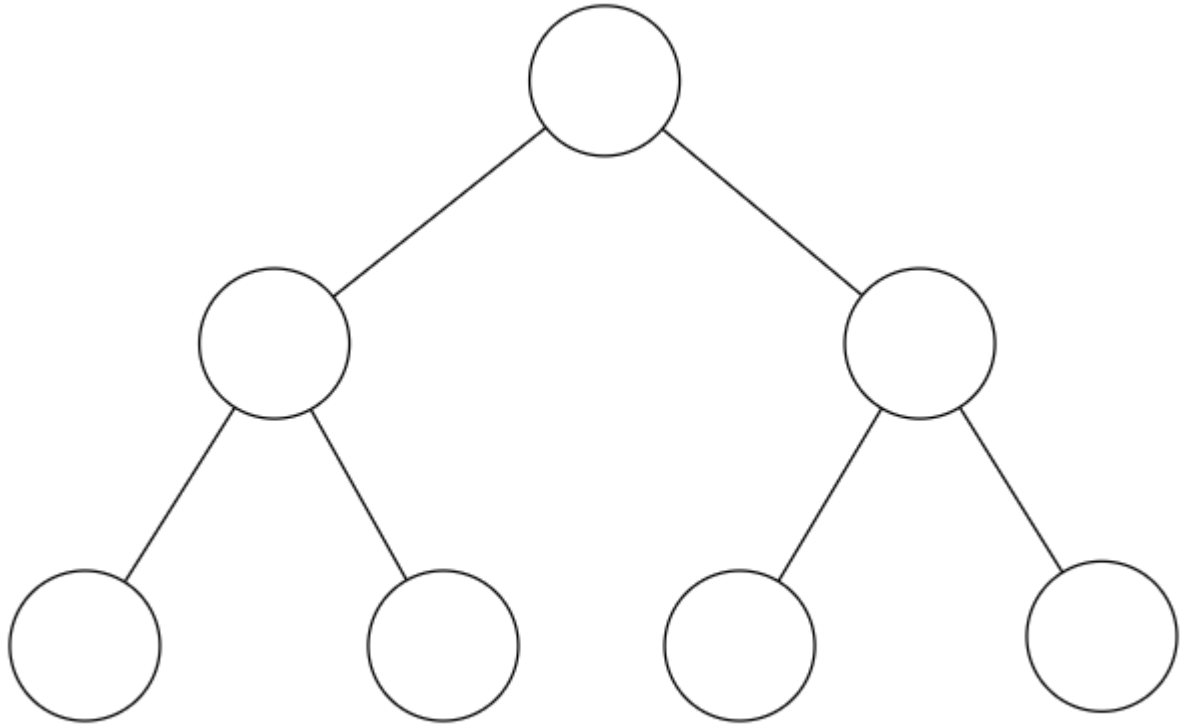
Terminology

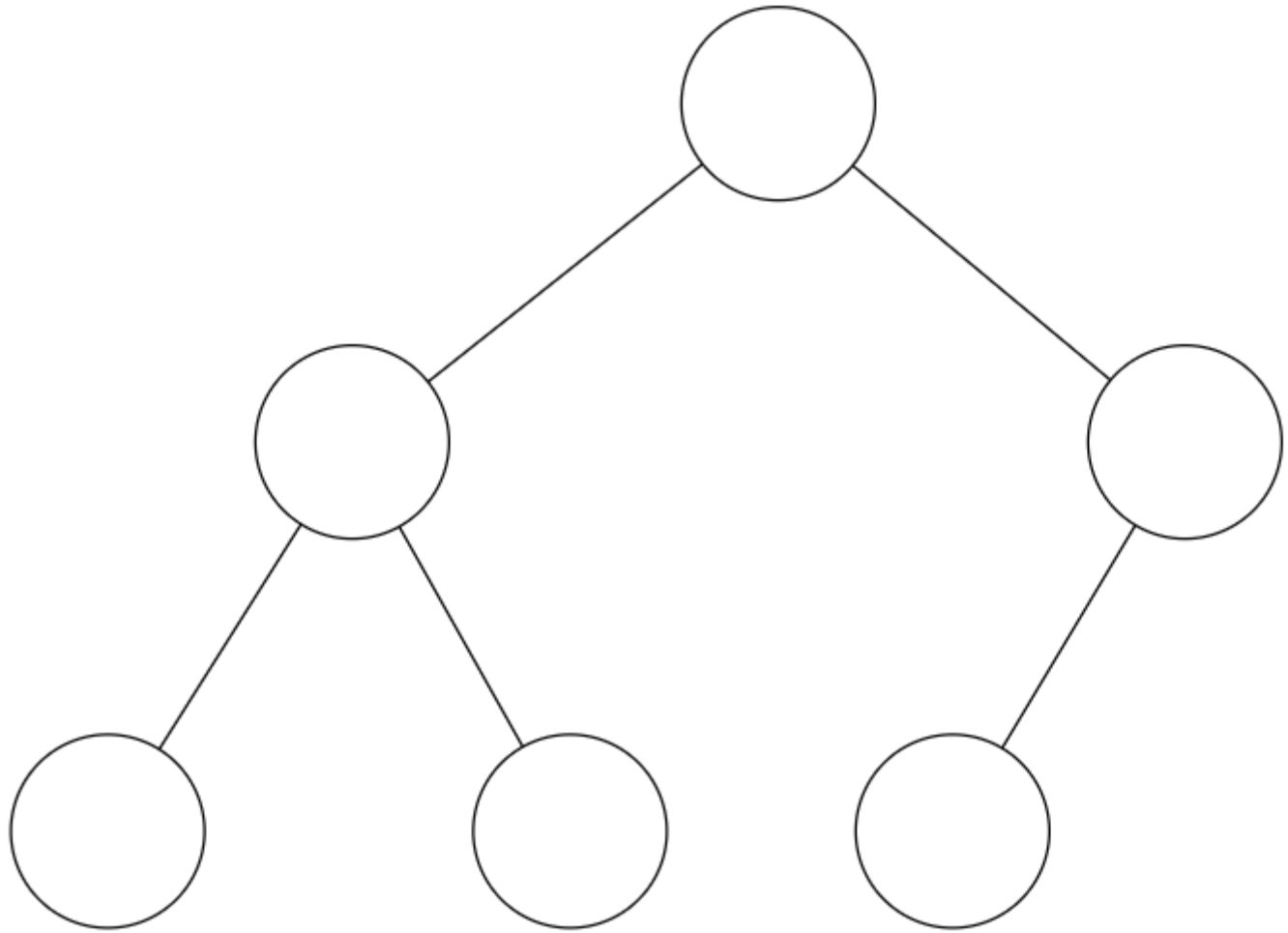
- A complete tree might or might not be full.

- A full tree by 1st definition is always complete.
- A full tree by 2nd definition might or might not be complete.

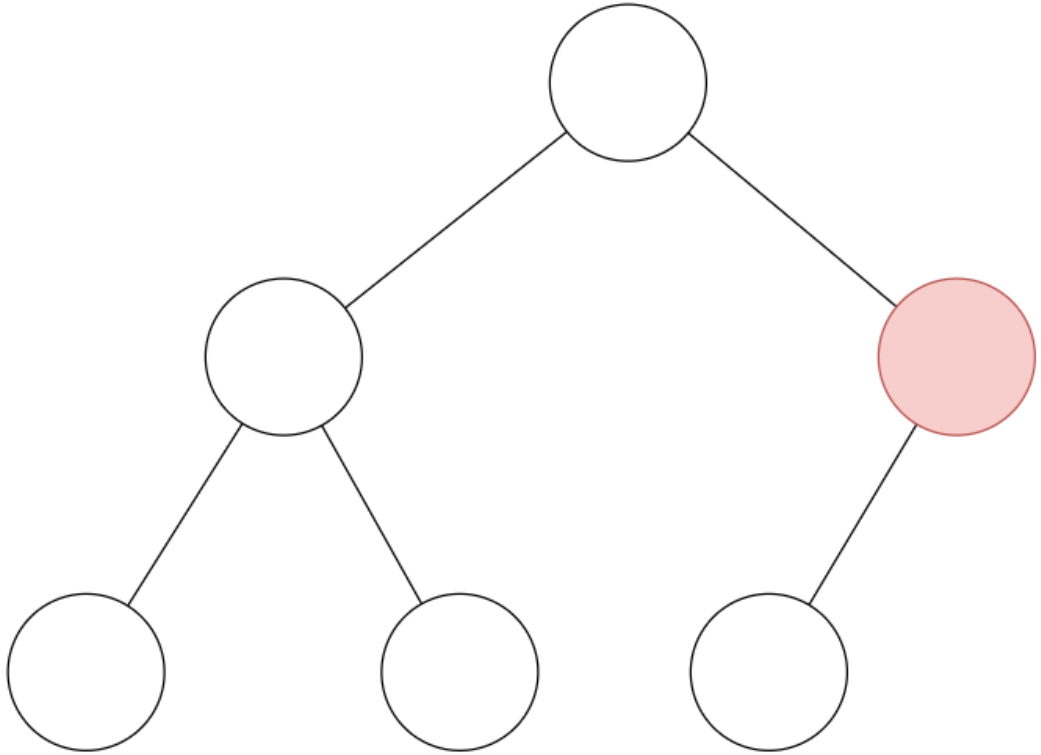


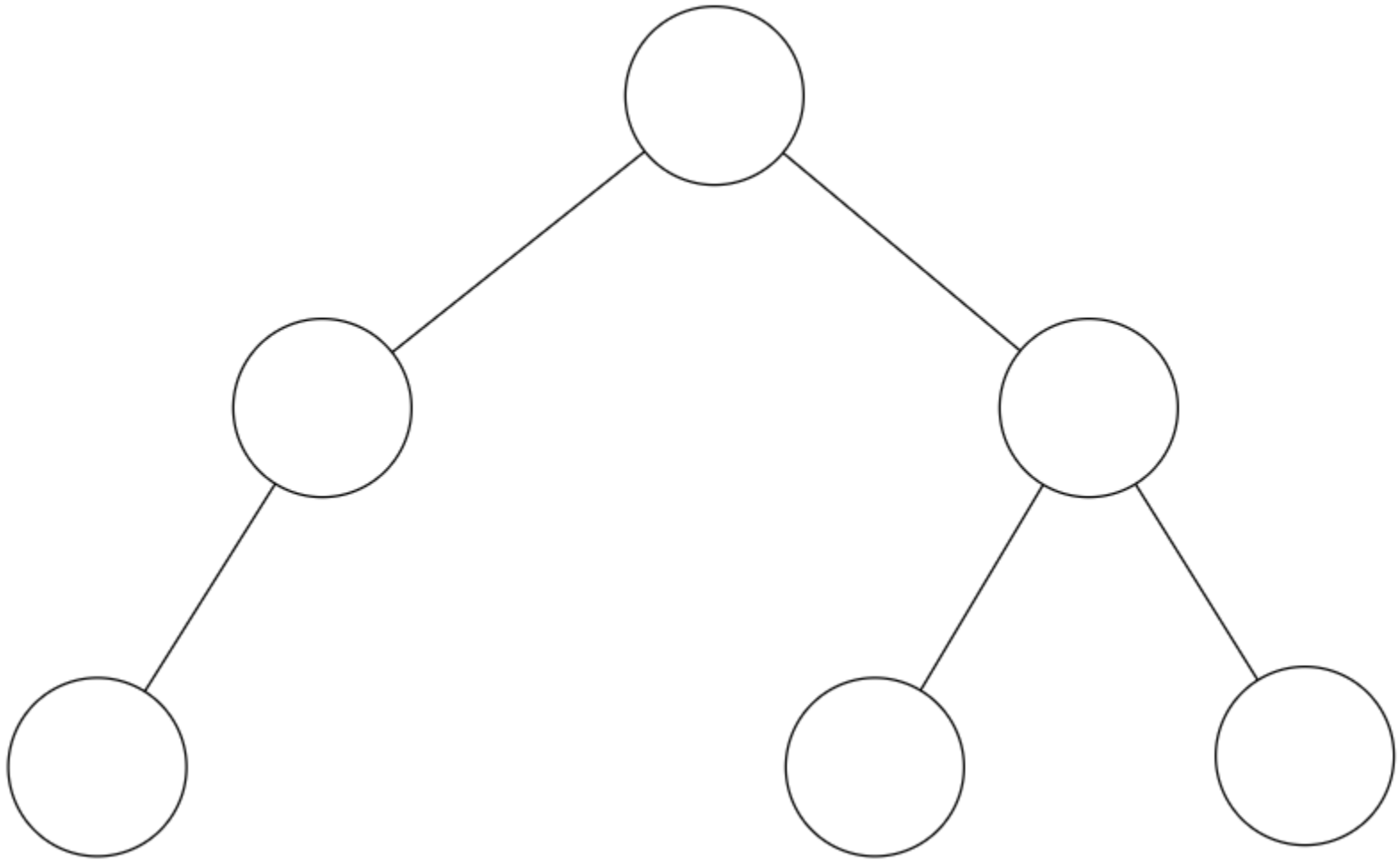
- Full by 1st? Yes
  - Each non-leaf has two children.
  - Each leaf is on the same level.
- Full by 2nd? Yes
  - Each non-leaf has two children.
- Complete? Yes
  - Each non-leaf has two children except possibly on second lowest level.
  - On the lowest level, the leaf nodes are as far left as possible.



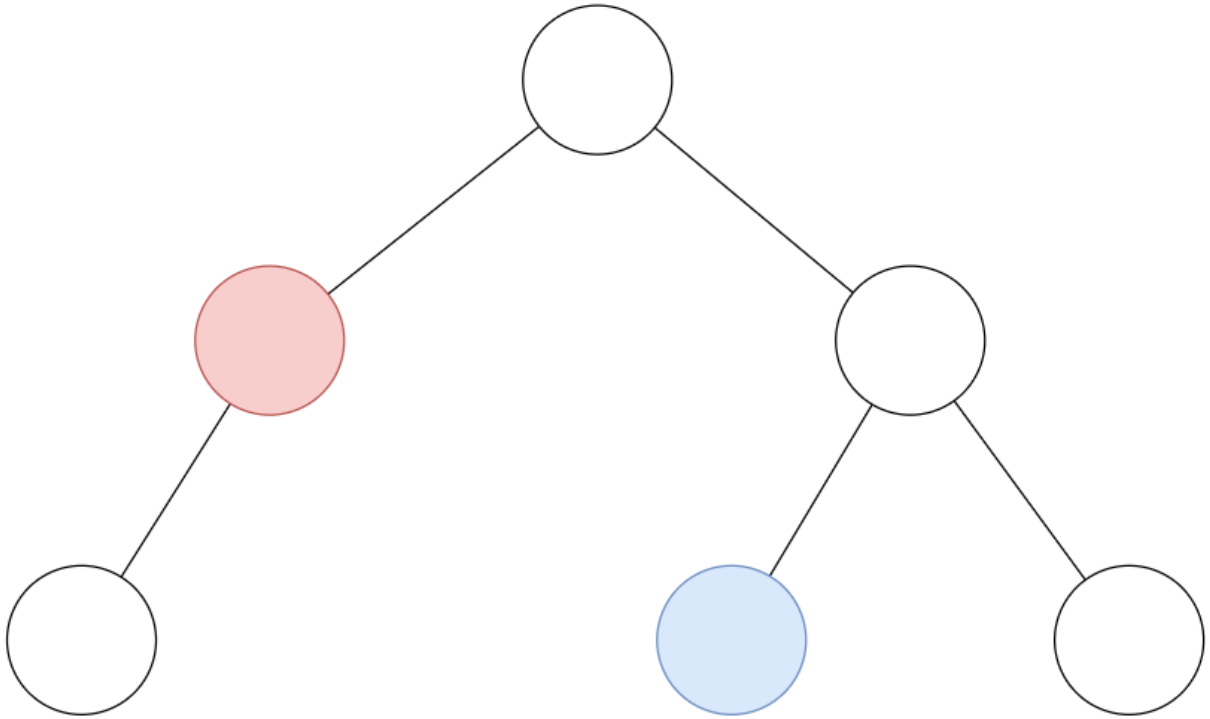


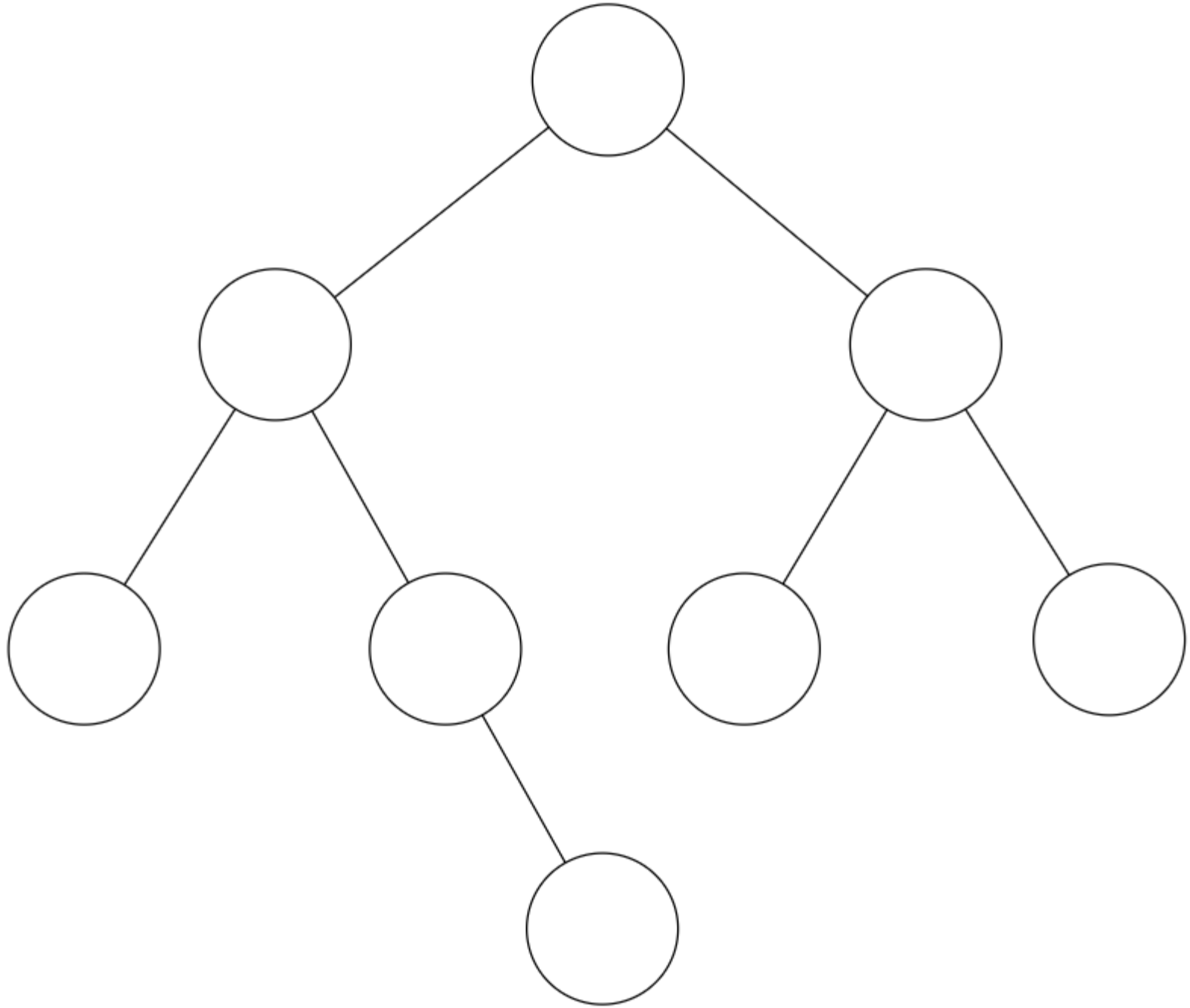
- Full by 1st? No
  - ~~Each non-leaf has two children.~~
  - Each leaf is on the same level.
- Full by 2nd? No
  - ~~Each non-leaf has two children.~~
- Complete? Yes
  - Each non-leaf has two children except possibly on second lowest level.
  - On the lowest level, the leaf nodes are as far left as possible.



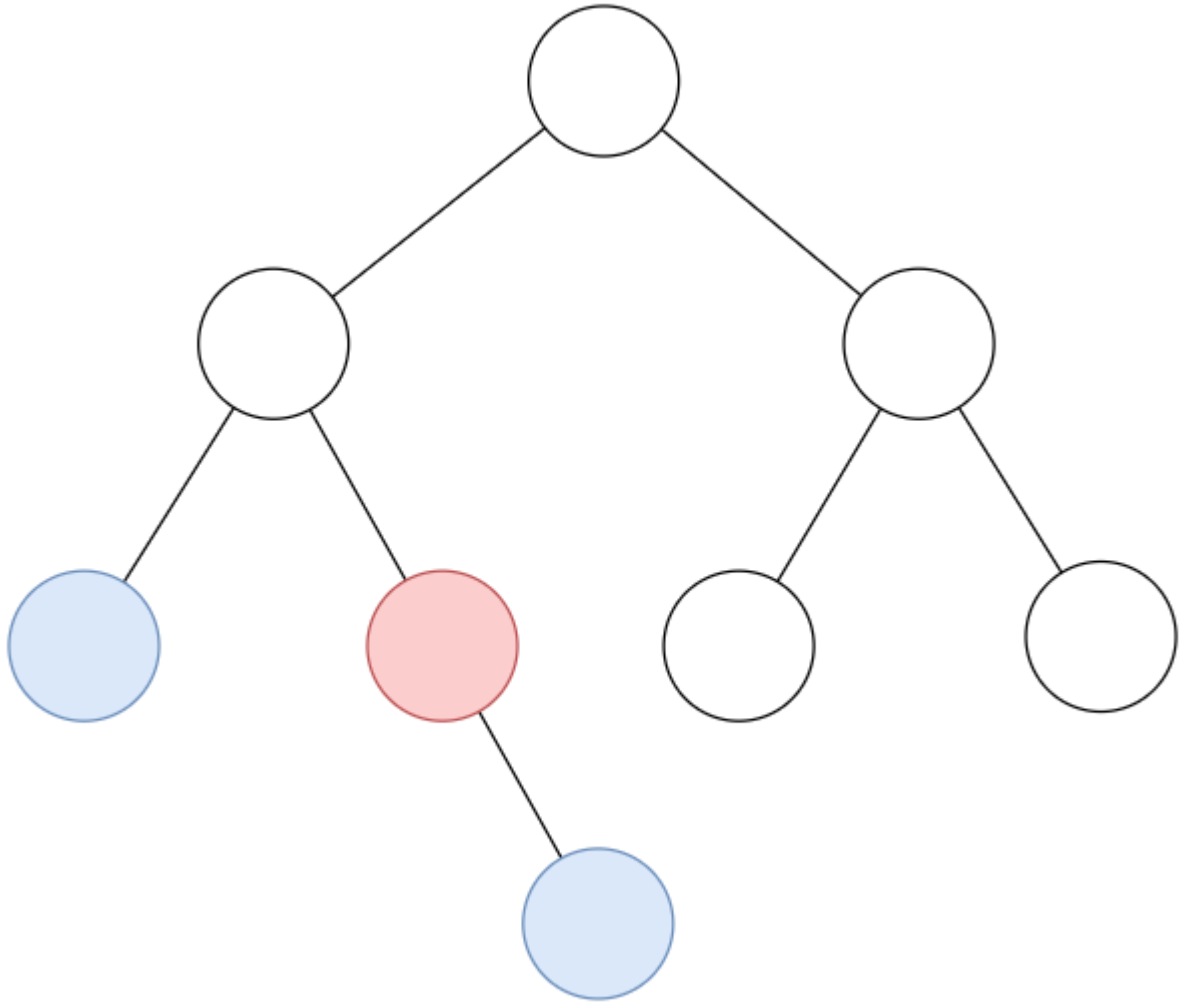


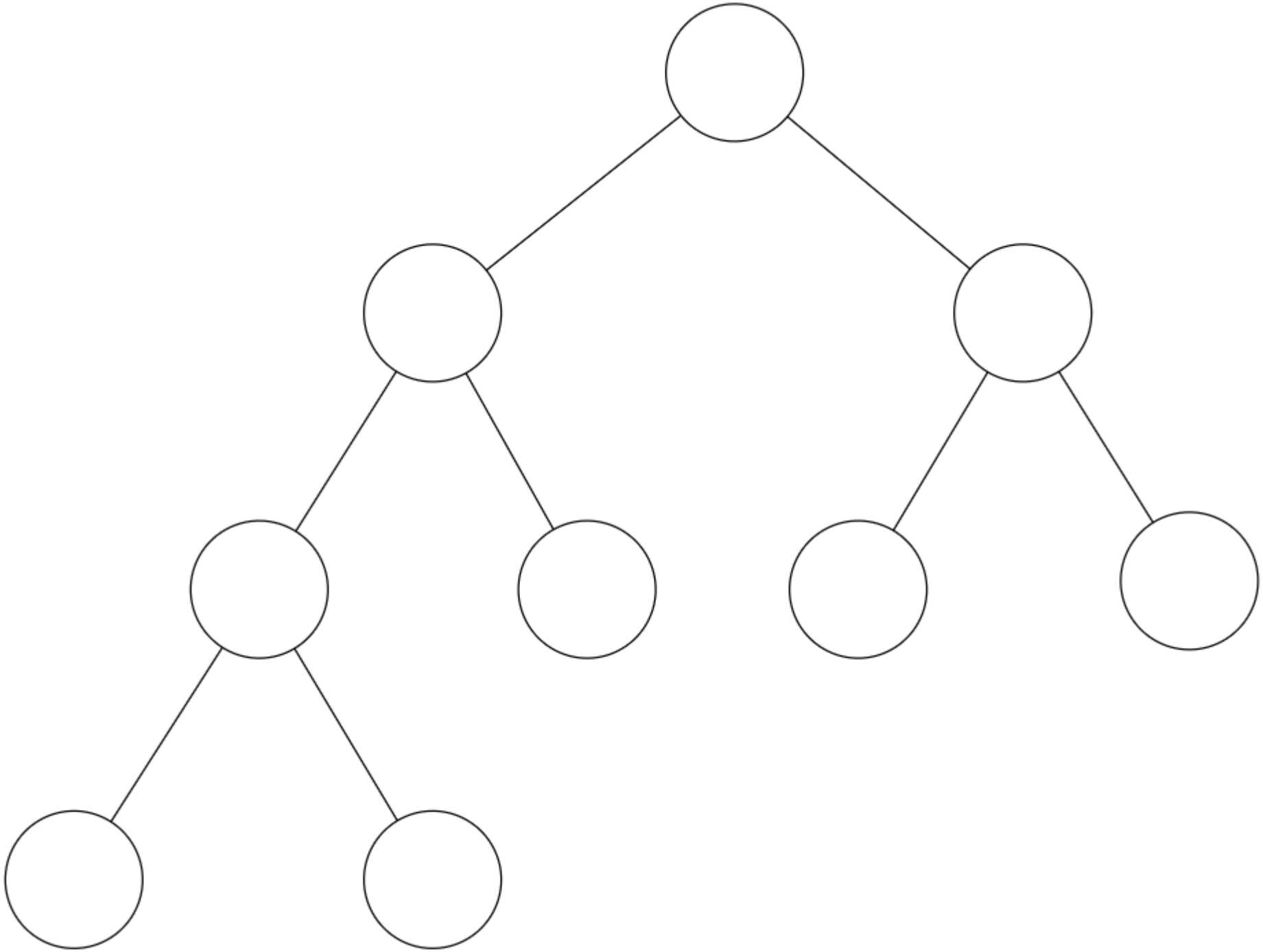
- Full by 1st? No
  - ~~Each non-leaf has two children.~~
  - Each leaf is on the same level.
- Full by 2nd? No
  - ~~Each non-leaf has two children.~~
- Complete? No
  - Each non-leaf has two children except possibly on second lowest level.
  - ~~On the lowest level, the leaf nodes are as far left as possible.~~



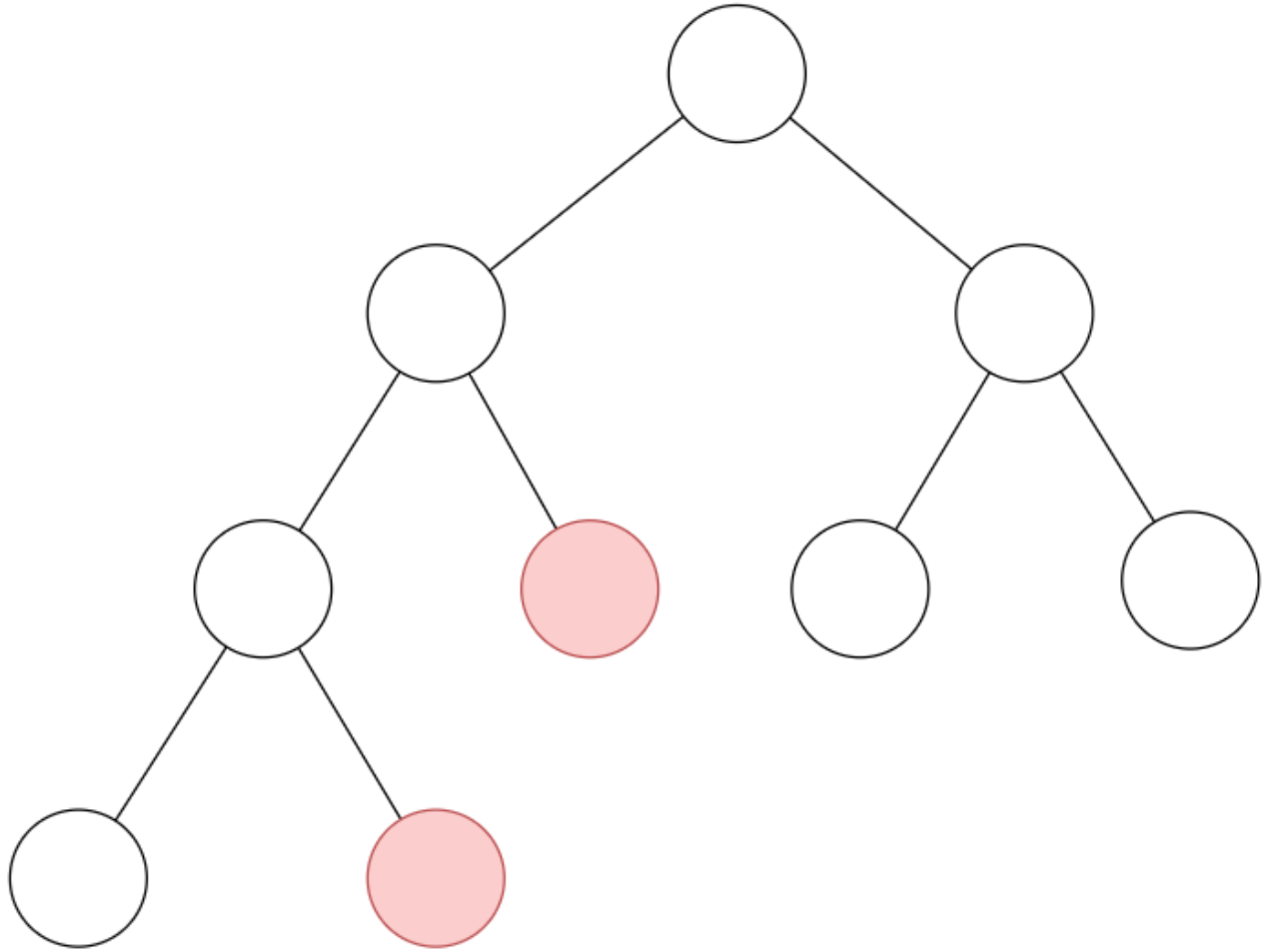


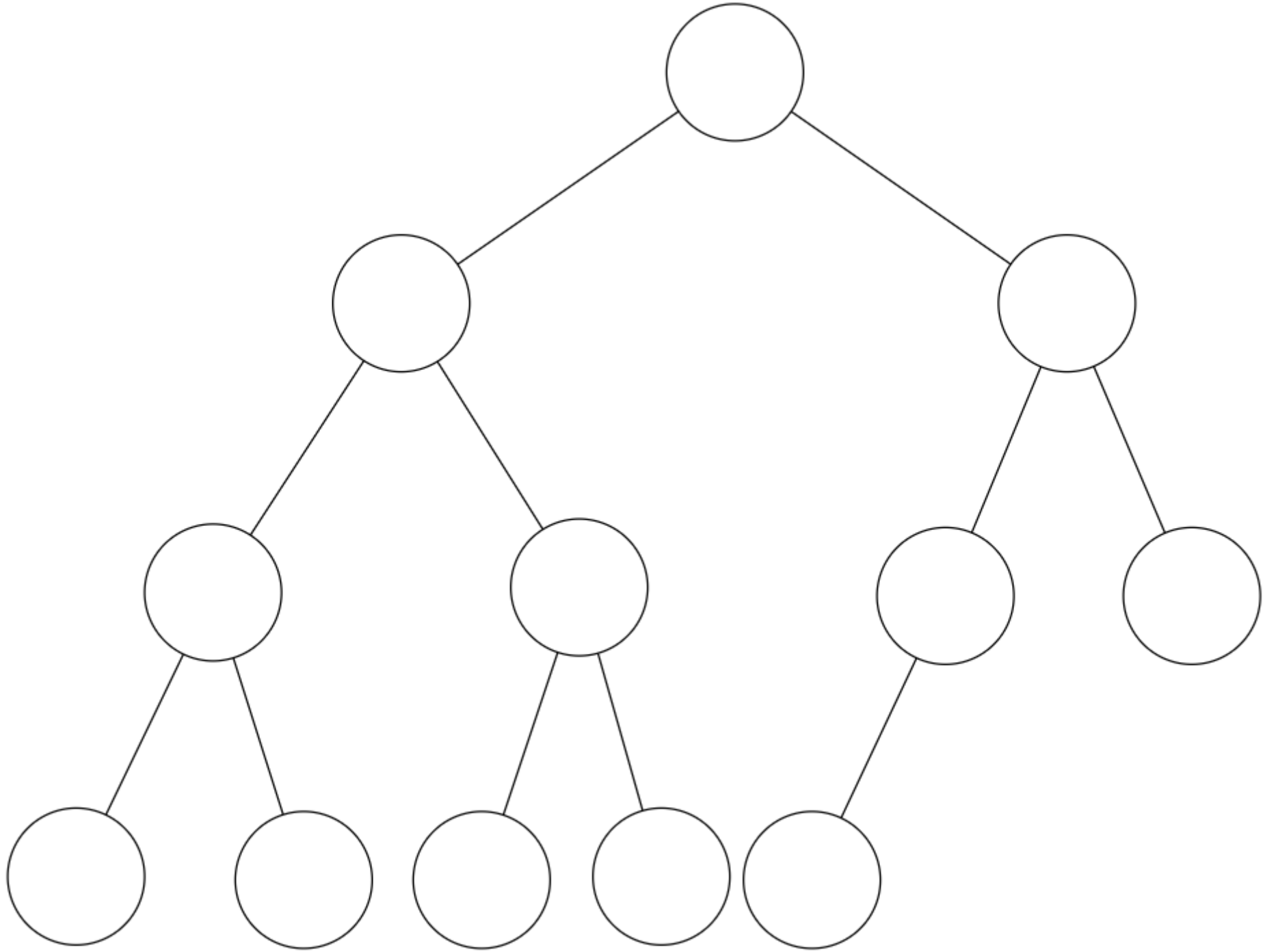
- Full by 1st? No
  - ~~Each non-leaf has two children.~~
  - ~~Each leaf is on the same level.~~
- Full by 2nd? No
  - ~~Each non-leaf has two children.~~
- Complete? No
  - Each non-leaf has two children except possibly on second lowest level.
  - ~~On the lowest level, the leaf nodes are as far left as possible.~~



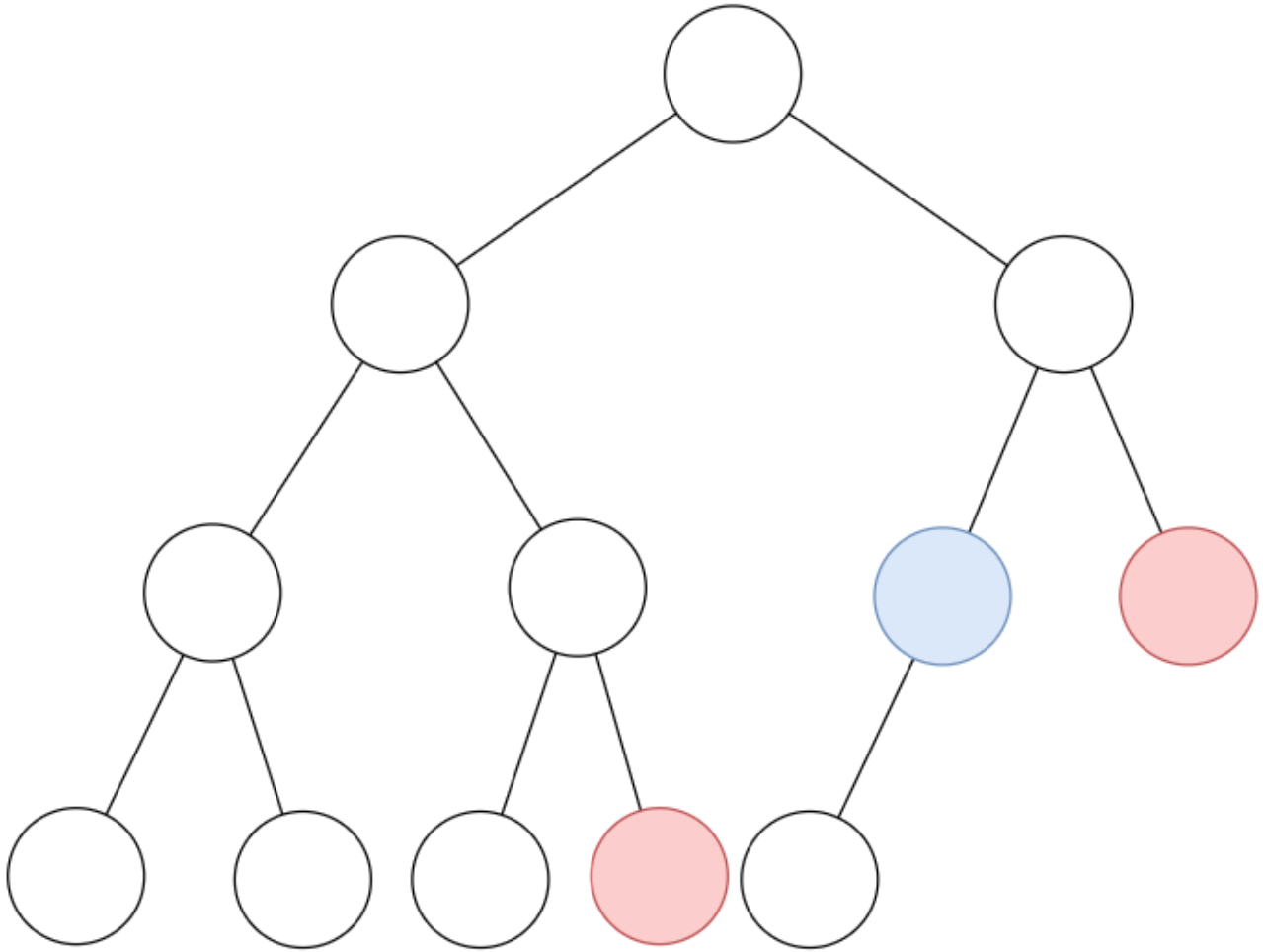


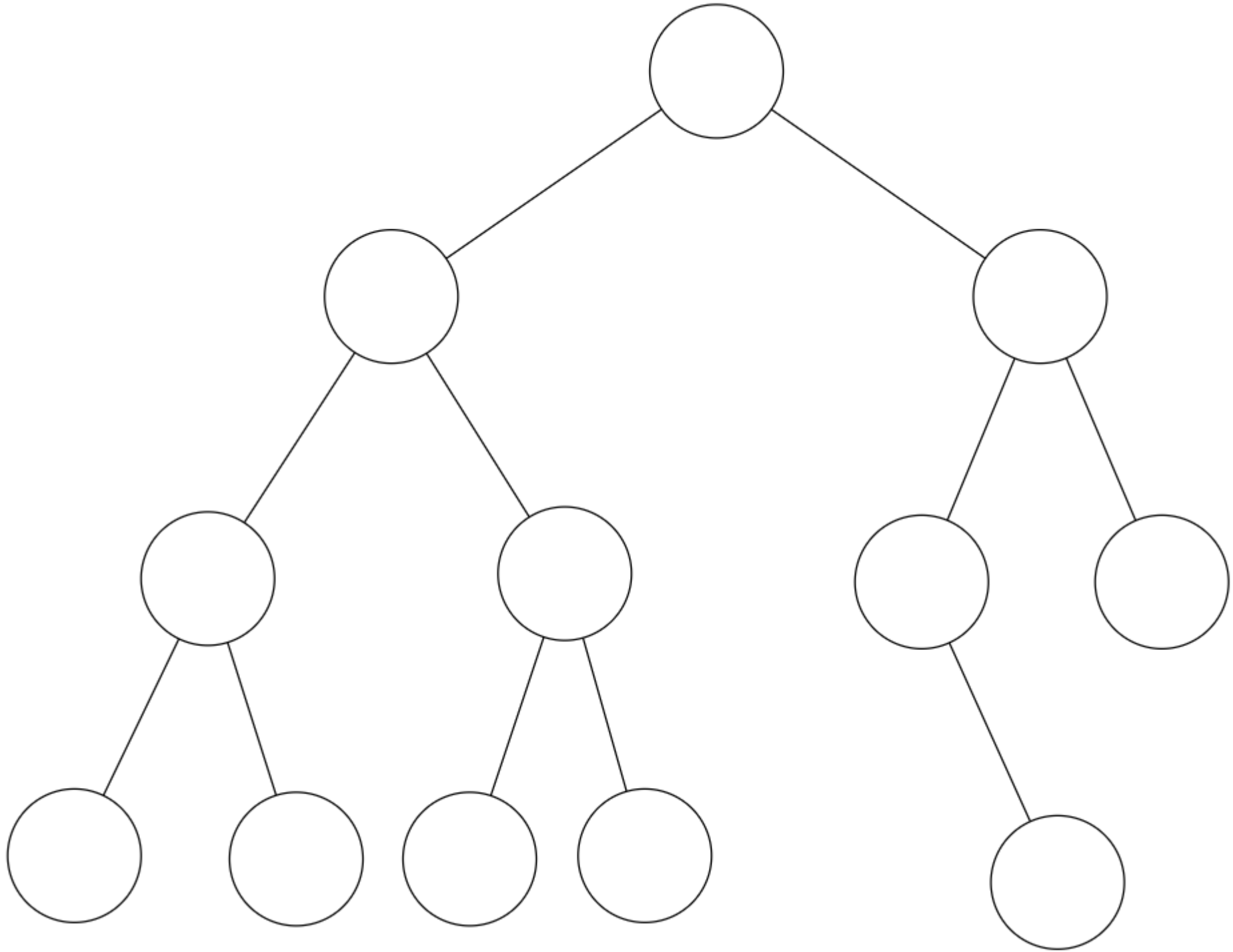
- Full by 1st? No
  - Each non-leaf has two children.
  - ~~Each leaf is on the same level.~~
- Full by 2nd? Yes
  - Each non-leaf has two children.
- Complete? Yes
  - Each non-leaf has two children except possibly on second lowest level.
  - On the lowest level, the leaf nodes are as far left as possible.



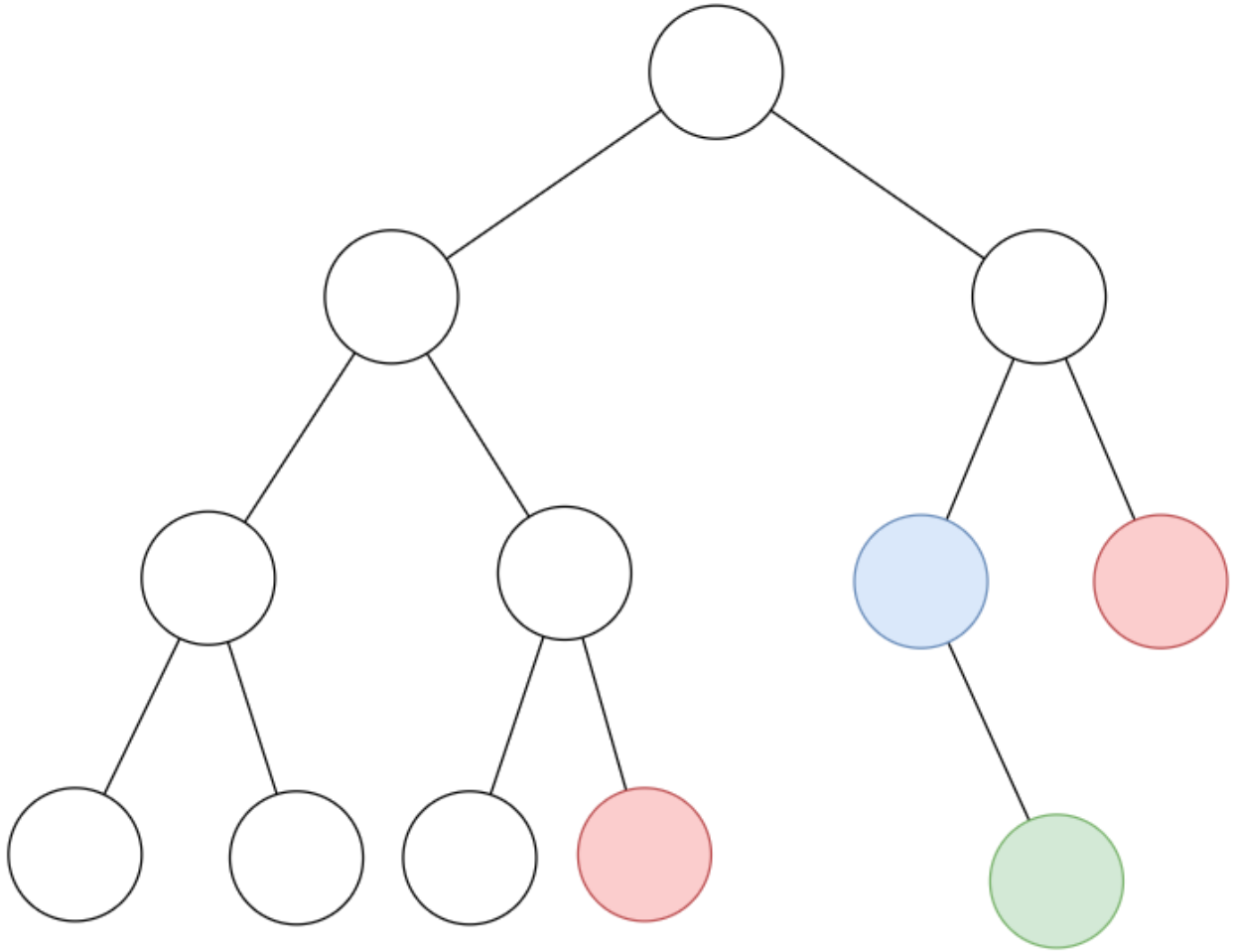


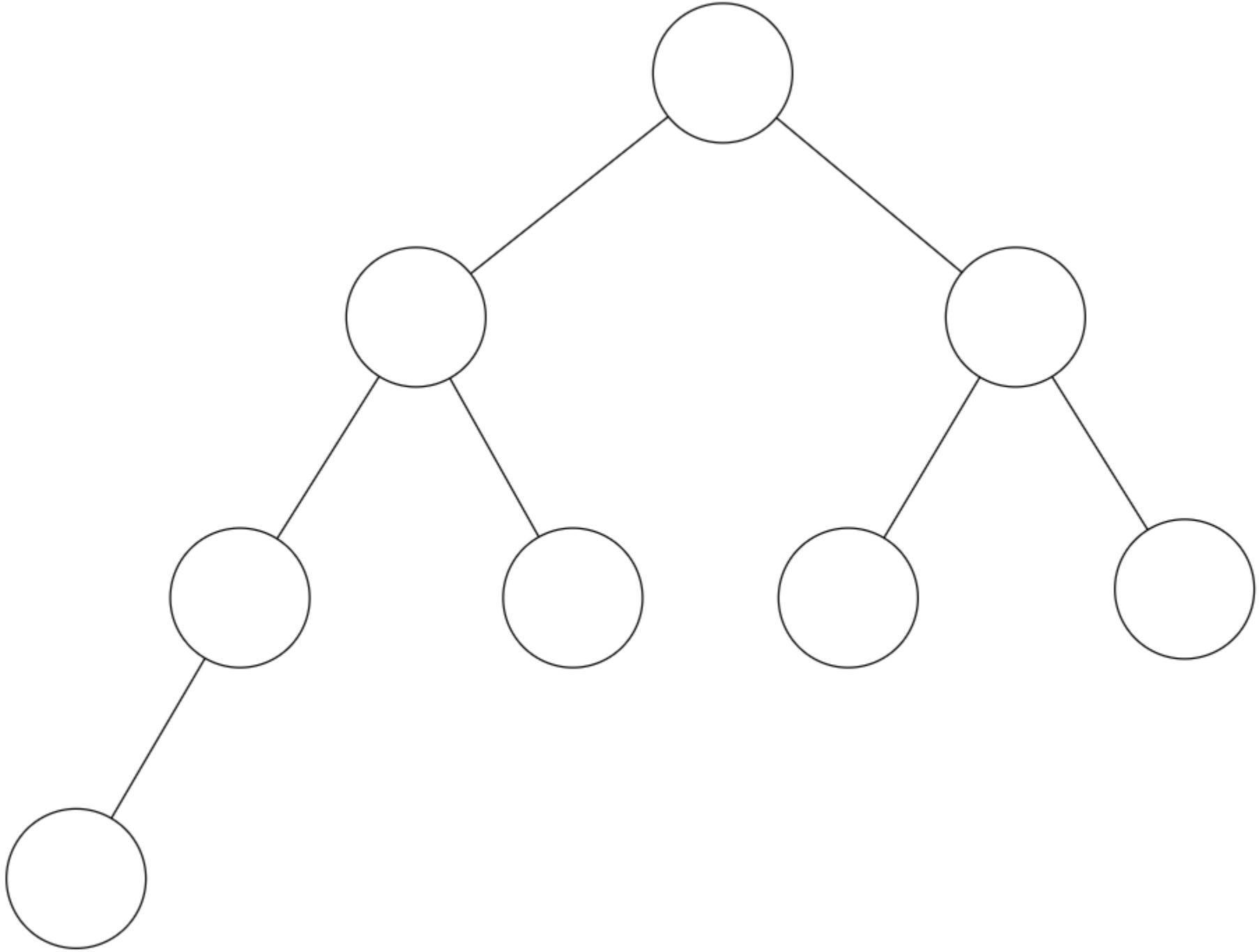
- Full by 1st? No
  - ~~Each non-leaf has two children.~~
  - ~~Each leaf is on the same level.~~
- Full by 2nd? No
  - ~~Each non-leaf has two children.~~
- Complete? Yes
  - Each non-leaf has two children except possibly on second lowest level.
  - On the lowest level, the leaf nodes are as far left as possible.



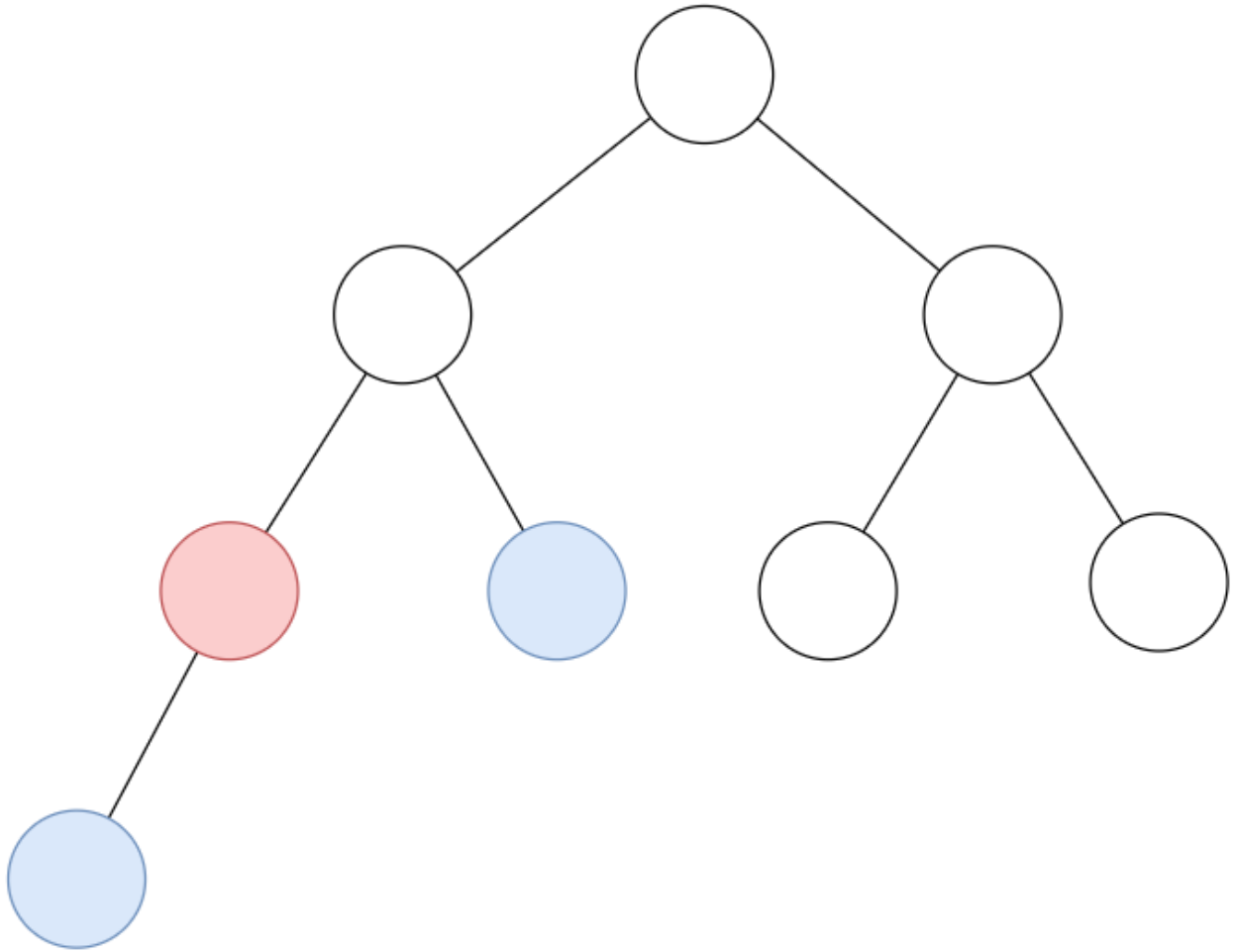


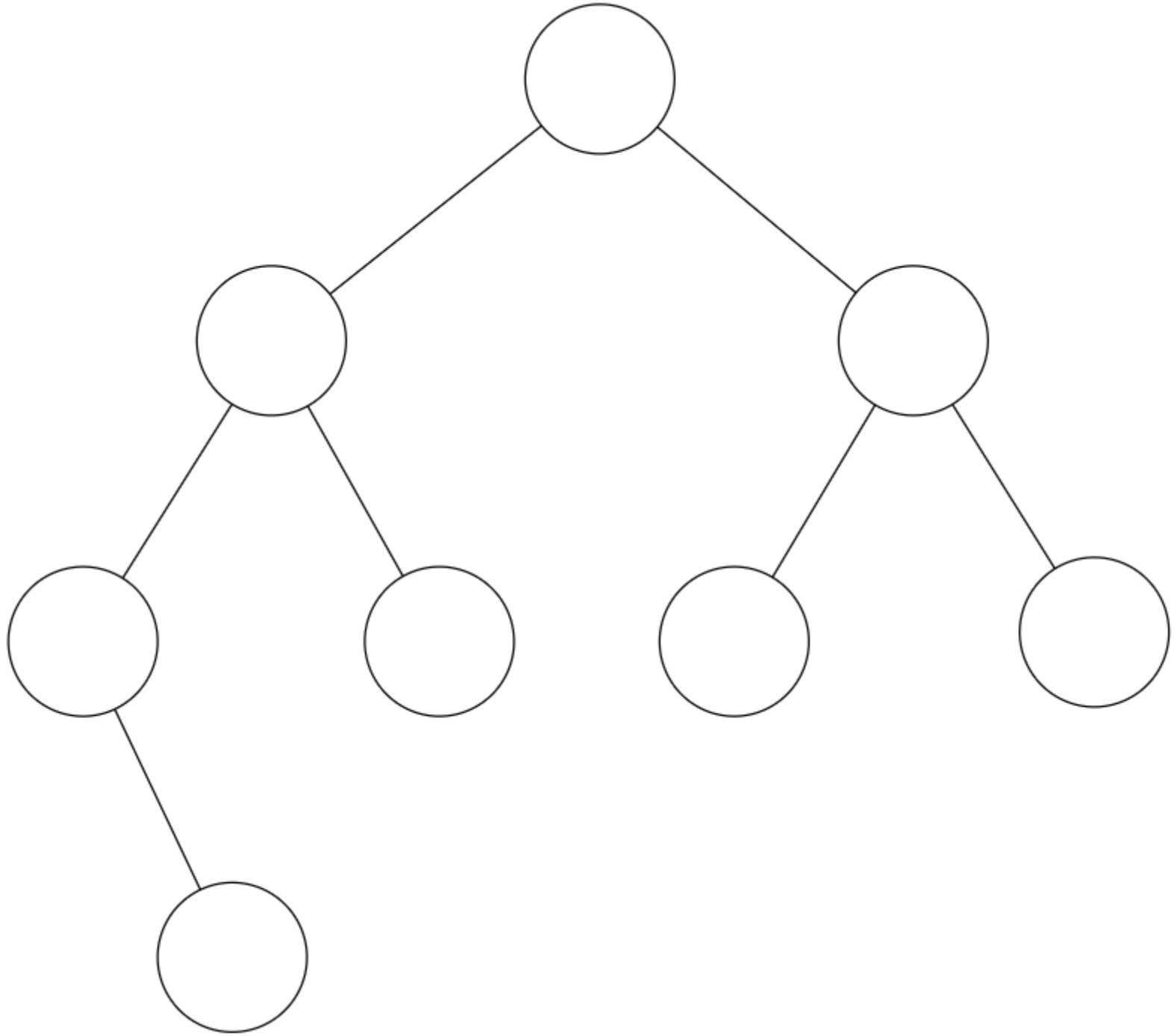
- Full by 1st? No
  - ~~Each non-leaf has two children.~~
  - ~~Each leaf is on the same level.~~
- Full by 2nd? No
  - ~~Each non-leaf has two children.~~
- Complete? No
  - Each non-leaf has two children except possibly on second lowest level.
  - ~~On the lowest level, the leaf nodes are as far left as possible.~~



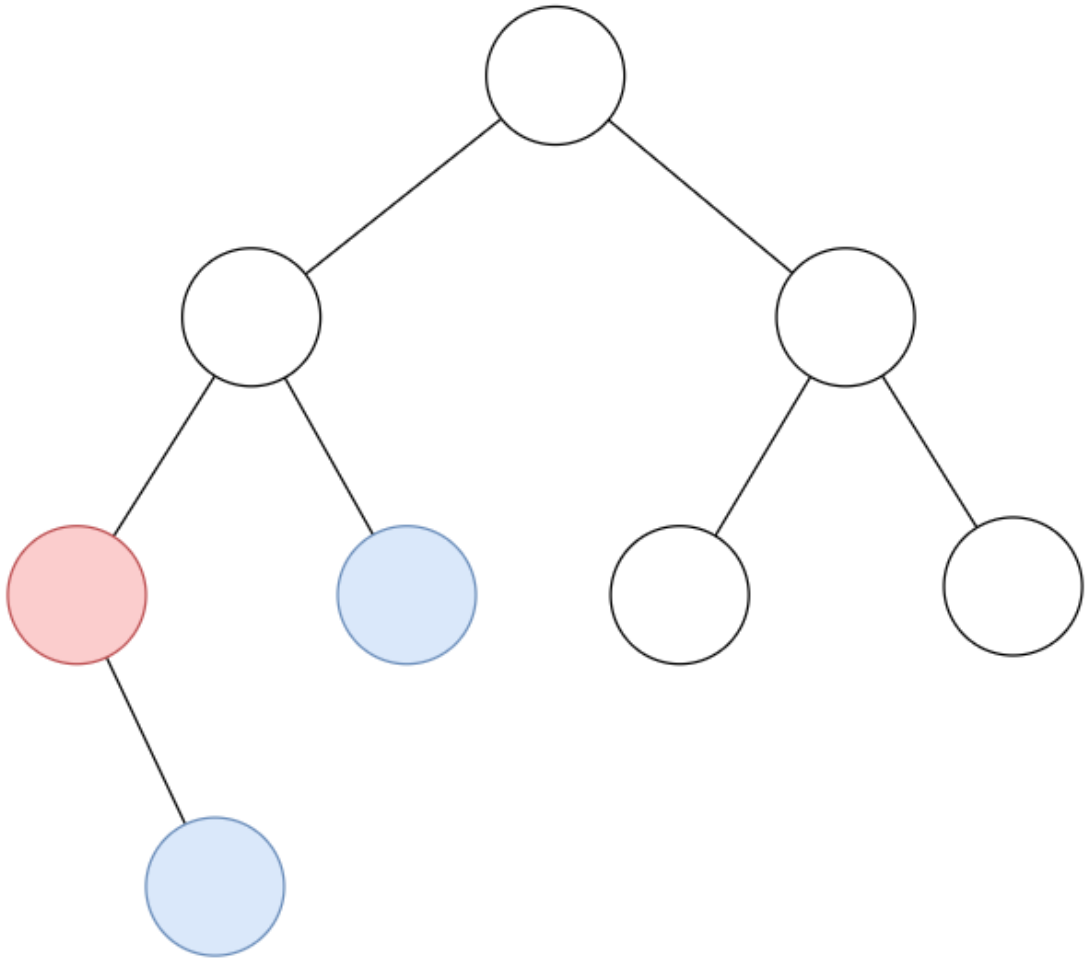


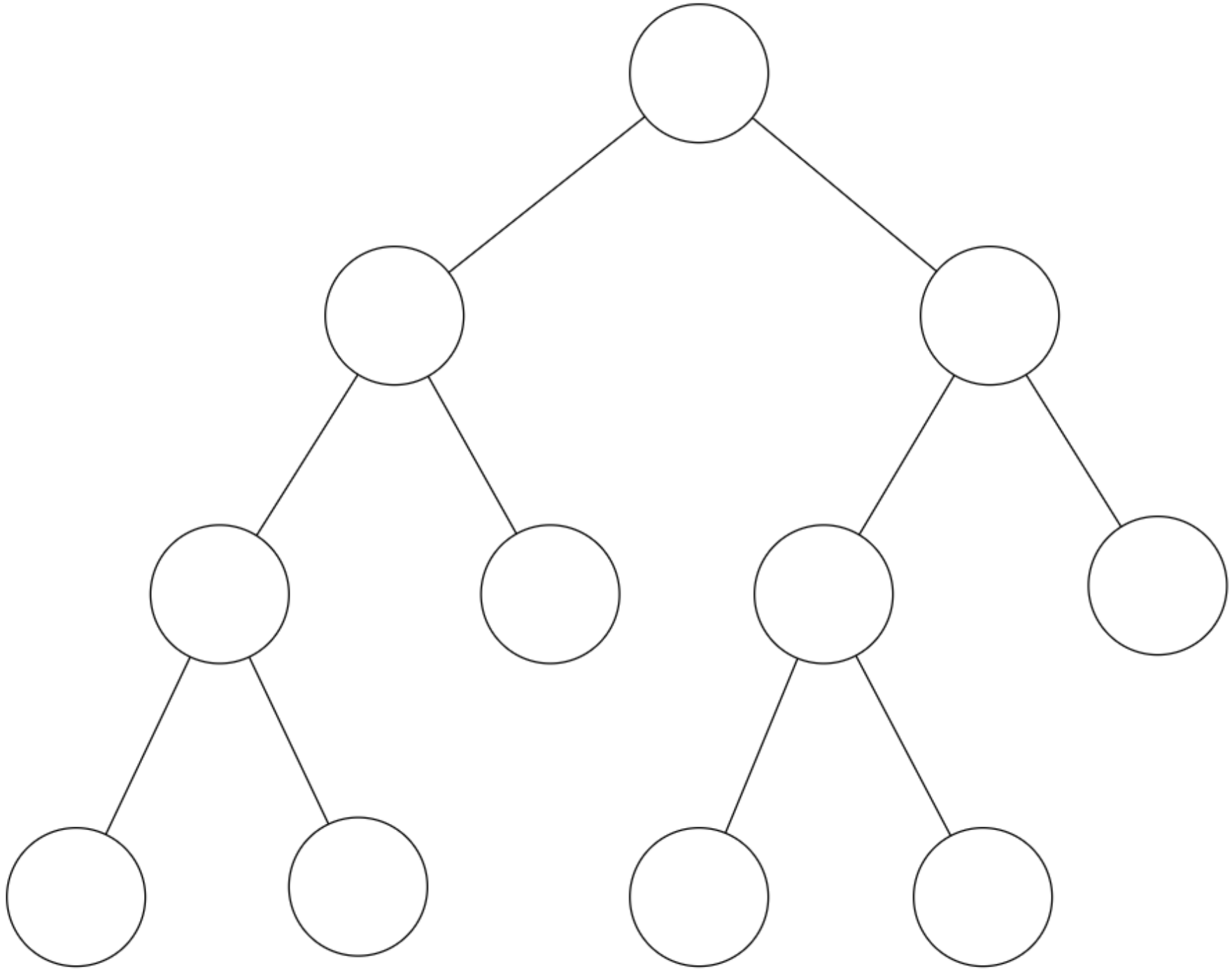
- Full by 1st? No
  - ~~Each non-leaf has two children.~~
  - ~~Each leaf is on the same level.~~
- Full by 2nd? No
  - ~~Each non-leaf has two children.~~
- Complete? Yes
  - Each non-leaf has two children except possibly on second lowest level.
  - On the lowest level, the leaf nodes are as far left as possible.



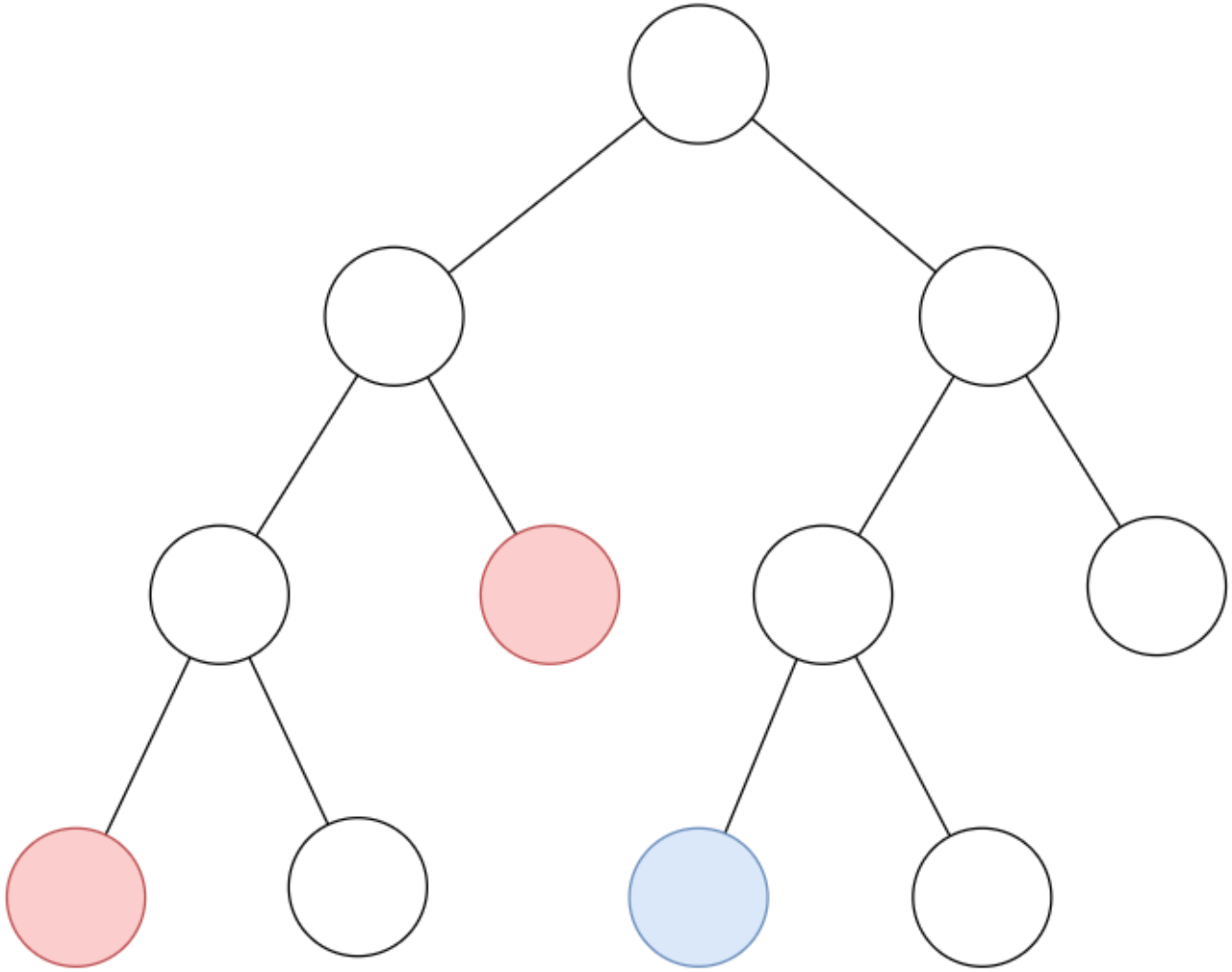


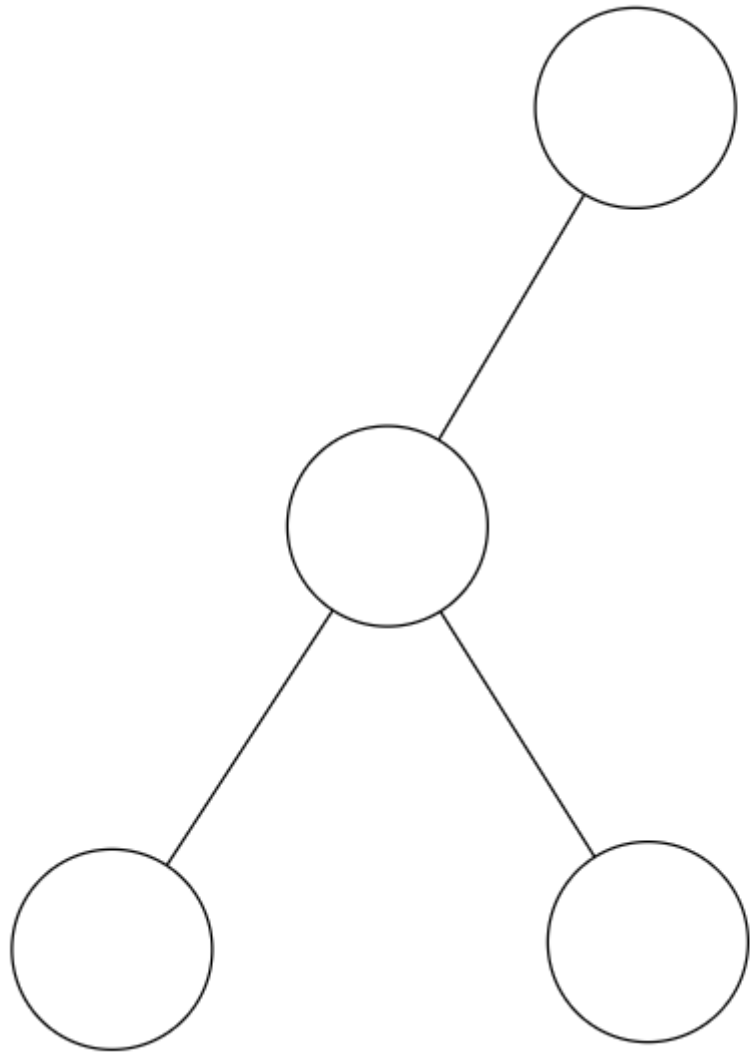
- Full by 1st? No
  - ~~Each non-leaf has two children.~~
  - ~~Each leaf is on the same level.~~
- Full by 2nd? No
  - ~~Each non-leaf has two children.~~
- Complete? No
  - Each non-leaf has two children except possibly on second lowest level.
  - ~~On the lowest level, the leaf nodes are as far left as possible.~~



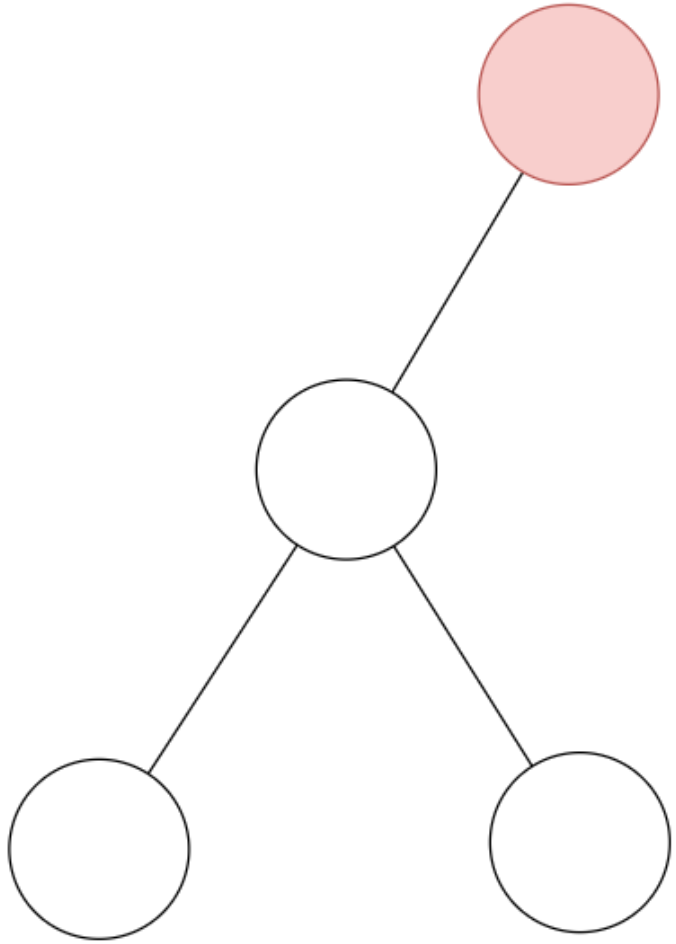


- Full by 1st? No
  - Each non-leaf has two children.
  - ~~Each leaf is on the same level.~~
- Full by 2nd? Yes
  - Each non-leaf has two children.
- Complete? No
  - Each non-leaf has two children except possibly on second lowest level.
  - ~~On the lowest level, the leaf nodes are as far left as possible.~~





- Full by 1st? No
  - ~~Each non-leaf has two children.~~
  - Each leaf is on the same level.
- Full by 2nd? No
  - ~~Each non-leaf has two children.~~
- Complete? No
  - ~~Each non-leaf has two children except possibly on second lowest level.~~
  - On the lowest level, the leaf nodes are as far left as possible.



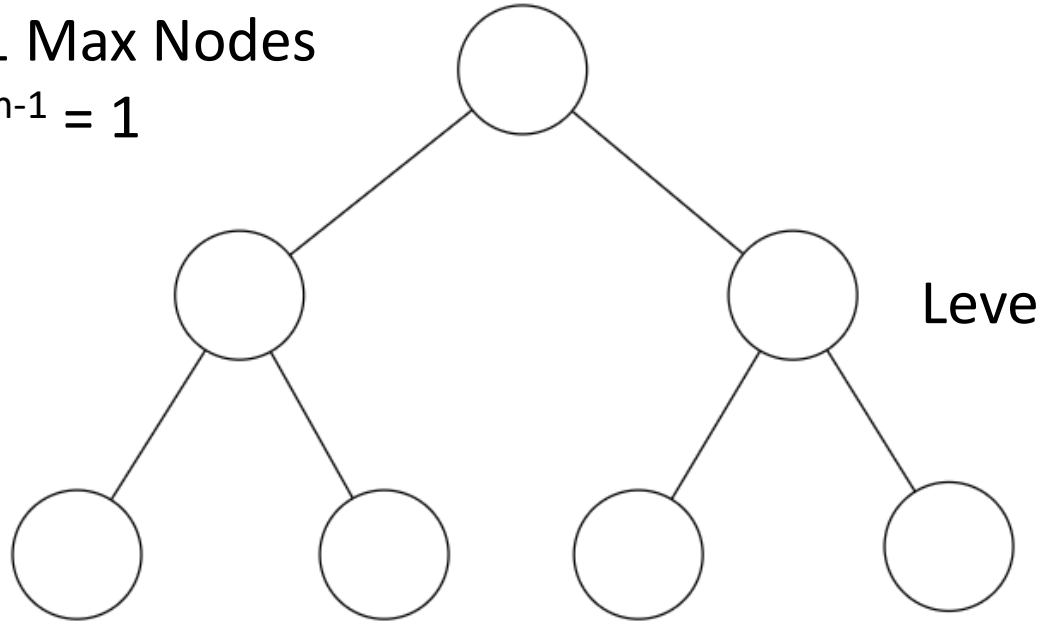
Properties of Binary Trees

- There are at most 2^{h-1} nodes at any level h.
  - Assume the root is level 1.

- There are at most $2^{h}-1$ total nodes in a tree with height h.
  - This assumes the 1-based height definition, where height is the number of levels (not the 0-based).
  - A full tree (using the first definition) will have this many nodes!

Binary Tree Example

Level 1 Max Nodes
$2^{h-1} = 1$



Level 2 Max Nodes
$2^{h-1} = 2$

Level 3 Max Nodes
$2^{h-1} = 4$

Max Nodes Total
$2^{h}-1 = 7$

TREE TRAVERSALS

Binary Tree Traversal

- We can access data stored in a tree by *traversing* the tree
- Traverse means to travel
  - We travel over all the elements in a tree
- This is similar to *iterating* over all the elements of a list

- Traversing a tree can be implemented with recursion or with stacks.

- Three options for when to process the data stored in the current node (compared to when you process child nodes or *subtrees*)

```
public void traversal (Node current) {
  if (current != null) {
    preOrderProcess(current.data);  // process before looking at subtrees

    traversal(current.left);

    inOrderProcess(current.data);    // process in between looking at left and right subtree

    traversal(current.right);

    postOrderProcess(current.data);  // process after looking at subtrees
  }
}
```

PreOrder Traversal

- Processing before looking at subtrees is a *preorder* traversal.
- Each node is processed **before** its children and subtrees.
- You would use preorder whenever the action requires data from higher in the tree.
- Example: if you're printing out a directory structure, you need to know the higher level folders before you can print out the current folder/file

PostOrder Traversal

- Processing after looking at both subtrees is *postorder* traversal.
- Each node is processed **after** its children and subtrees.
- You would use postorder whenever the action requires data from lower in the tree.
- Example: if you were processing and then deleting nodes that met some criteria, you'd want to be sure to delete any nodes lower on the tree before deleting the higher nodes.

InOrder Traversal

- Processing after looking at the left subtree but before looking at the right is an *inorder* traversal.
- For a special kind of binary tree, a binary **search** tree, the inorder traversal gives you a sorted list of the nodes.

Visitation Sequence

- If you were to invoke all three processing steps, each node would be processed three times.
- This is called the *visitation sequence*.

Traversing General Trees

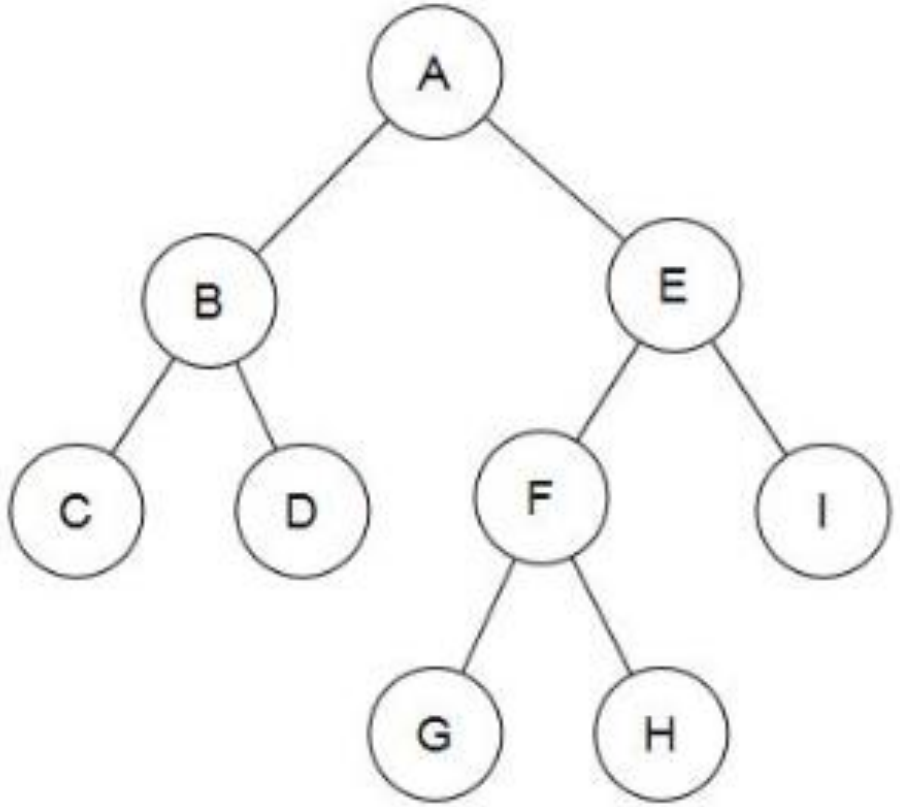
- Preorder and postorder traversals can be used for all trees, not just binary trees.
- Inorder traversals only make sense for binary trees since a general (non-binary) tree might not have a *middle* node.

Finding the Traversal Order

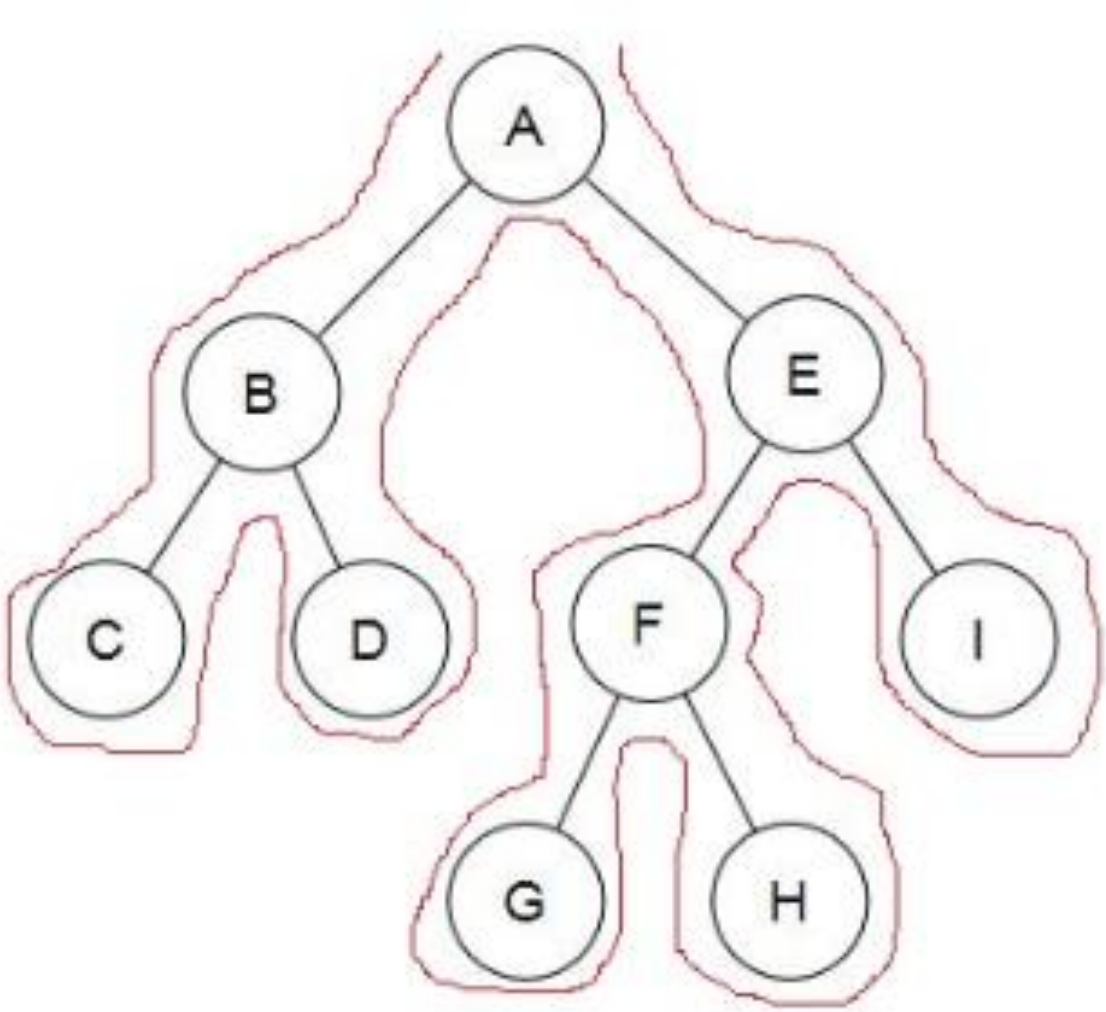
- One way to find the traversal is by drawing a trace-line.
- Start at the root node and draw a line **counterclockwise** around all nodes.
- Follow the trace line and process the node when the line touches it on the left, bottom, or right side, depending on the kind of traversal:
  - preorder processes a node when it is first reached- when the line is on the left of the node
  - inorder processes a node when it is reached the second time- when the line is on the bottom of the node
  - postorder processes a node when it is reached the third (and final) time- when the line is on the right of the node

Example

Graph?
Tree?
Binary Tree?
Full?
Complete?

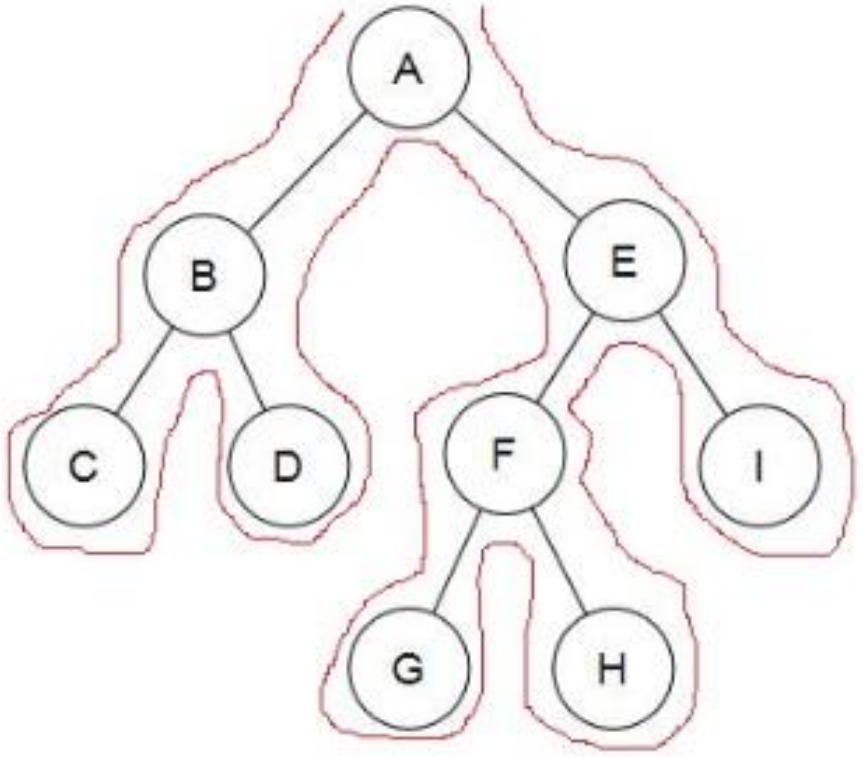


Example



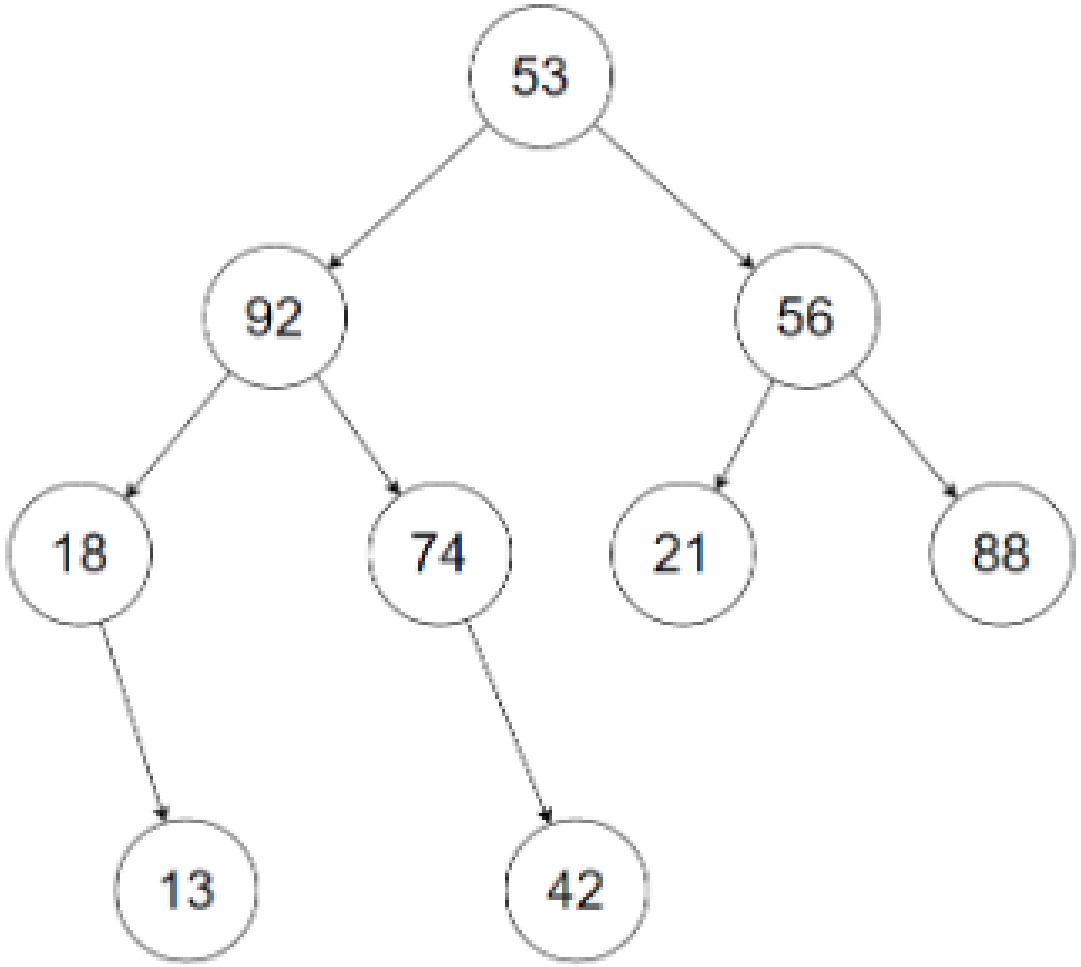
Example

- preorder: A B C D E F G H I
- inorder: C B D A G F H E I
- postorder: C D B G H F I E A



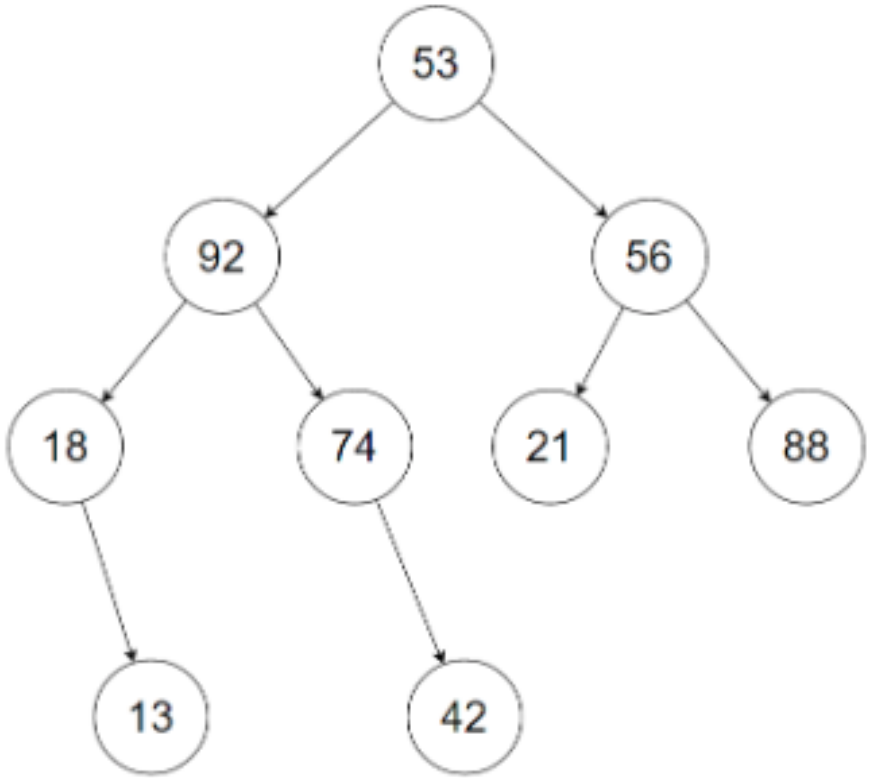
- visitation sequence: A B C C C B D D D B A E F G G G F H H H F E I I I E A

Example



Example

- Preorder: 53 92 18 13 74 42 56 21 88
- Postorder: 13 18 42 74 92 21 88 56 53
- Inorder: 18 13 92 74 42 53 21 56 88
  - Note that 18 comes *before* 13
  - Note that 74 comes *before* 42



Recursive Method for Traversal

- Recursion is fairly straightforward for binary trees.
- Example:

```
public void preOrderTreeMethod(Node current) {
  if(current != null) {
    do something with the current node
    preOrderTreeMethod(current.left);
    preOrderTreeMethod(current.right);
  }
}
```

- Perform task on one node.
- Then recursively perform the task on both child nodes.

Using Stacks for Traversal

```
public void nonRecursiveTreeMethod(Node startingNode) {
    create a stack of nodes
    push the startingNode onto the stack
    while(the stack is not empty) {
        pop off a node
        do something with the current node
        if current node has a left child, push the left child onto stack
        if current node has a right child, push the right child onto stack
    }
}
```

- Perform task on one node.
- Use the stack to keep track of nodes that still need processing.

Depth-First and Breadth-First Traversals

- The three traversals we've seen are *depth-first* traversals.
  - This means they go deep down one path before coming back up to continue the traversal.
  - Depth-first traversals are typically implemented with stacks (or recursion).
- *Breadth-first* traversals traverse a tree level-by-level.
  - They are typically implemented with queues.
  - This kind of traversal is not as common for trees, but is used in some AI-related and gaming applications.
  - Breadth-first traversal is more common for graphs.
  - Example: a "spidering" search of the web: get all pages one-link from the current page; then get all pages two-links from the current page; etc.

TREE IMPLEMENTATIONS

Implementing Trees

- We can use linked nodes or an array!

Implementing Binary Trees with Nodes

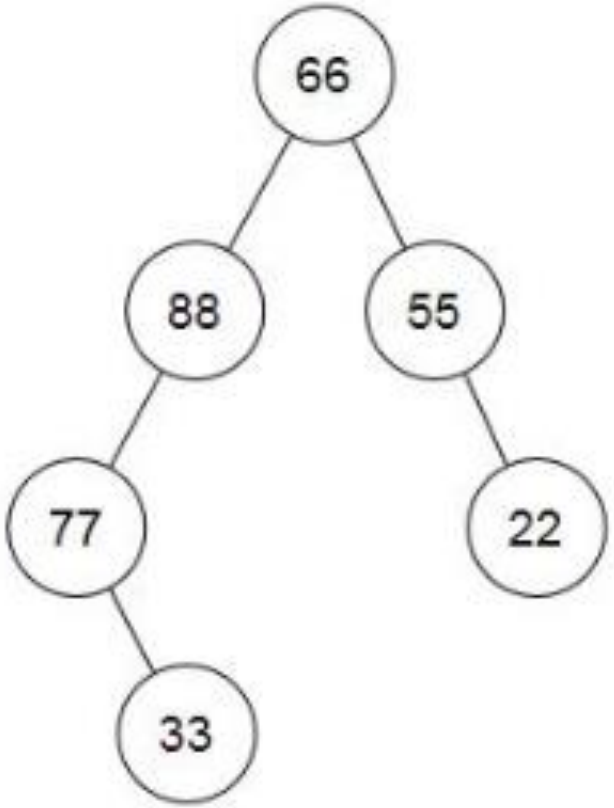
- This is pretty straightforward!
- We know that each node can have at most two child nodes.
  - We know the structure of the tree.
- Create a node that has **two** instance data variables (left and right) instead of just one (next).

- A leaf node will have left==null and right==null.

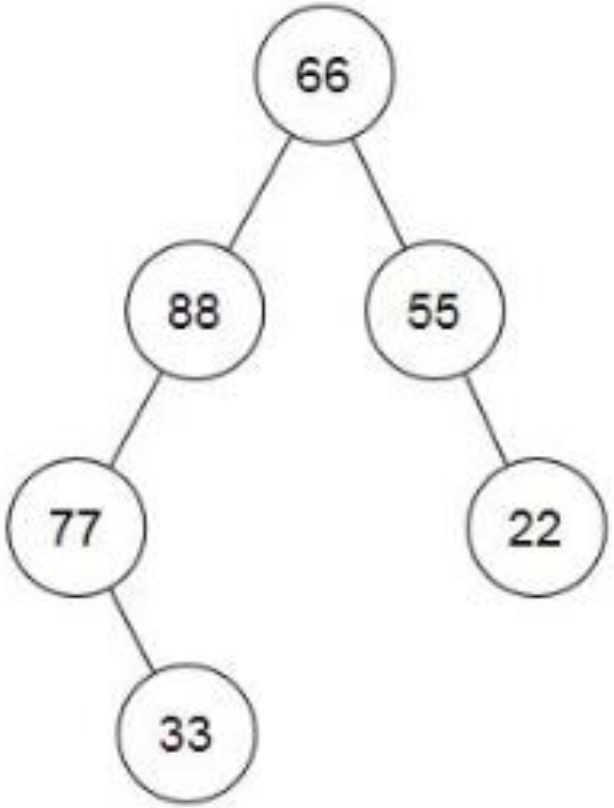
Implementing Binary Trees with Arrays

- When we use an array, we call it 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]

Example

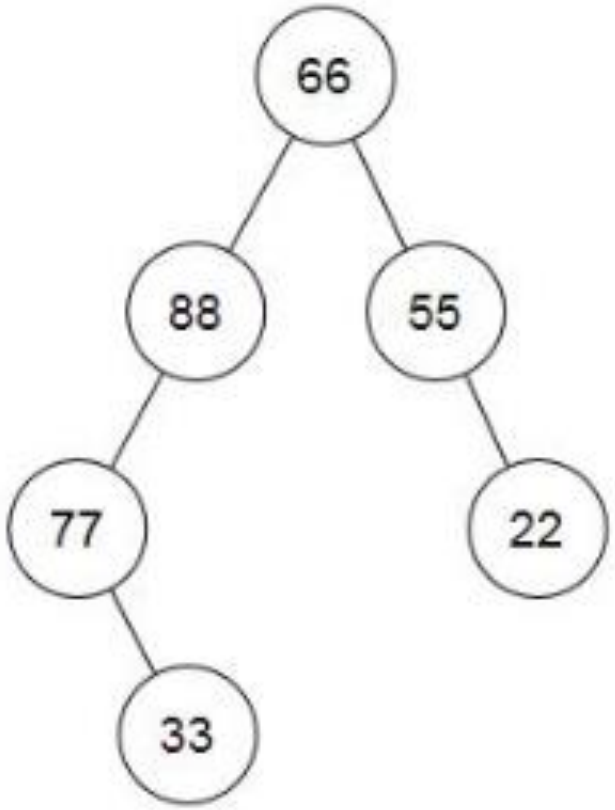


Example



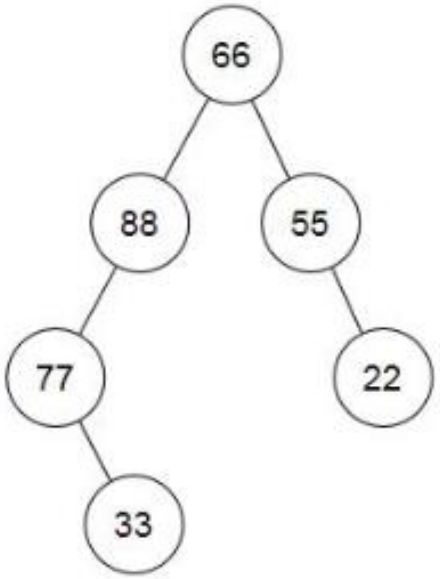
- How big should the array be?
- Max nodes = 2^h-1 = 15 so the array length will be 16

Example



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

Implementing General Trees

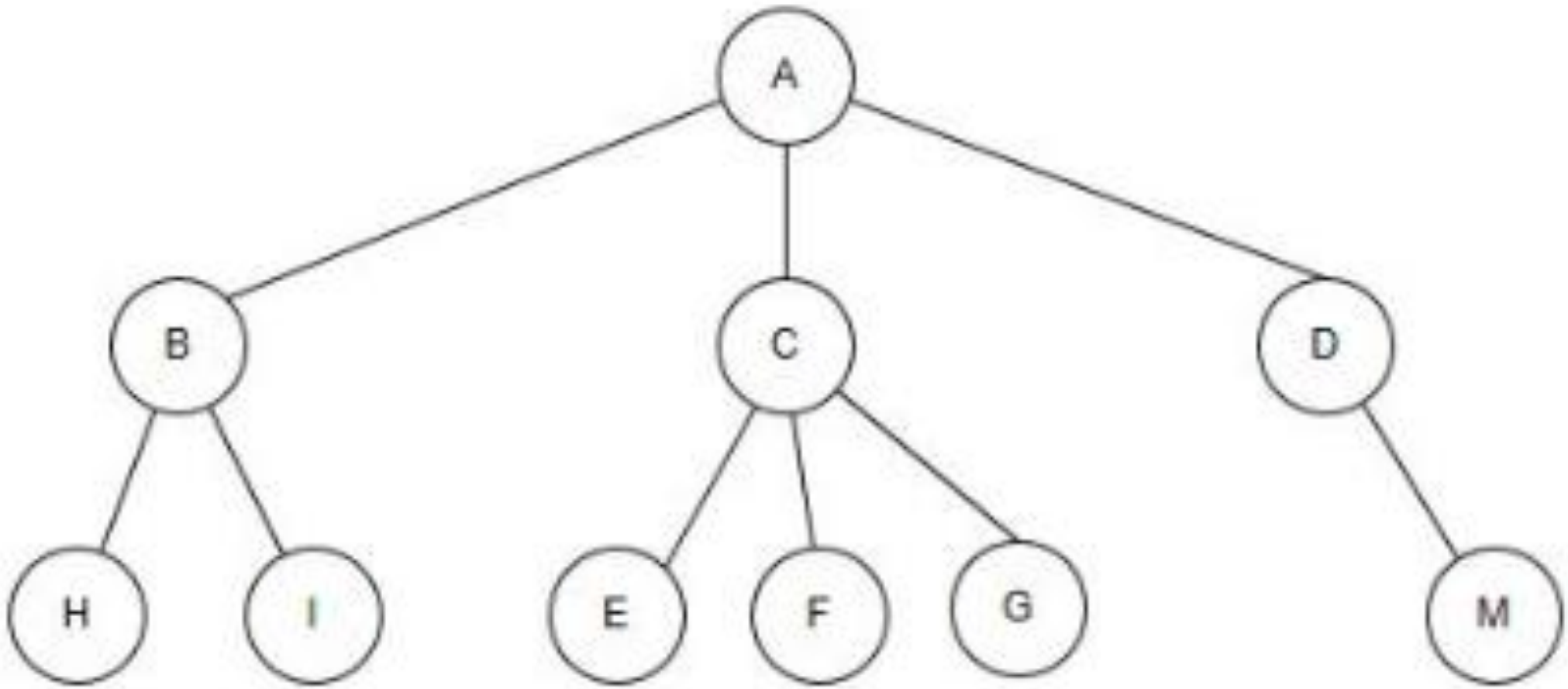
- This is more challenging than implementing binary trees.
- In a general tree, nodes can have any number of children.
  - We don’t know the general structure!
- Difficult to represent this kind of tree with linked nodes.

- One approach is to keep a *list* of child nodes.
- Another approach is to convert the general tree to a binary tree. We can then implement it as a binary tree.

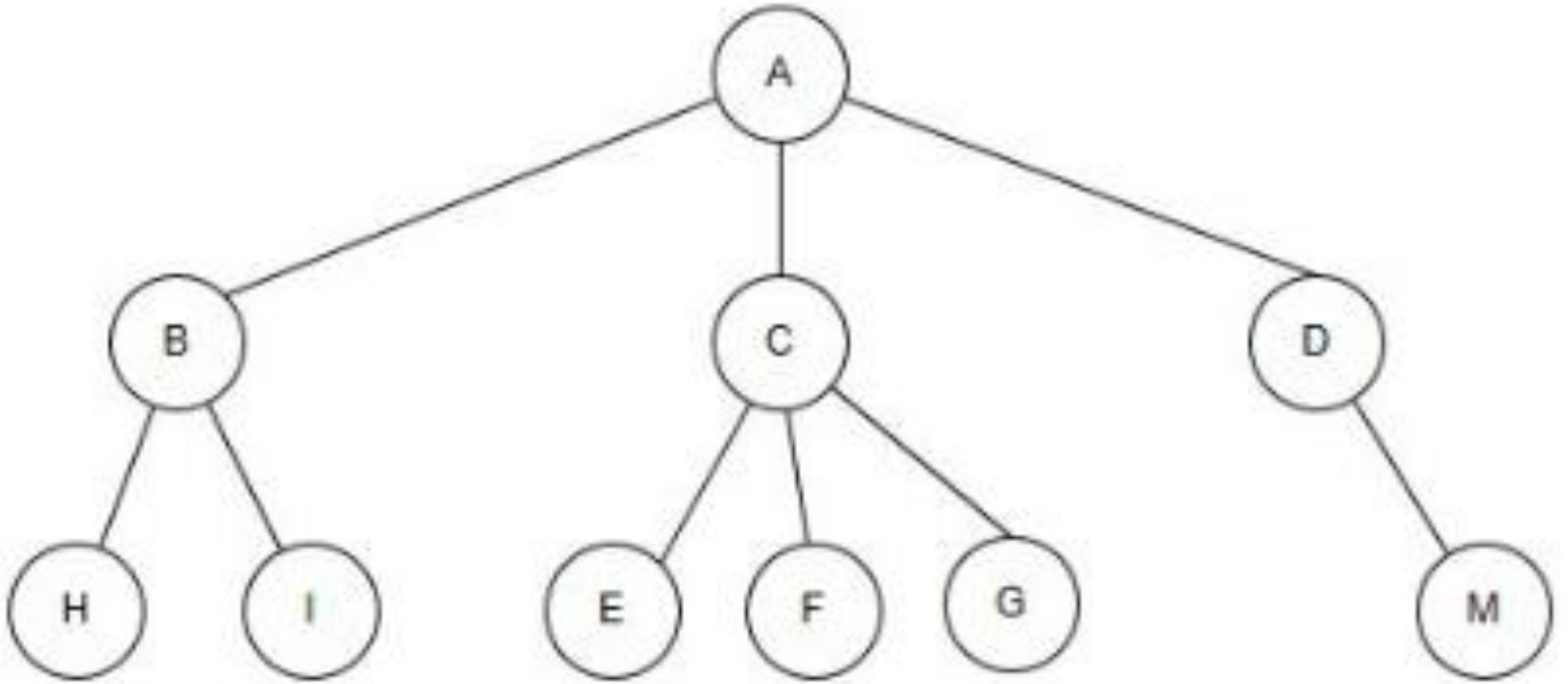
Converting General Tree to Binary Tree

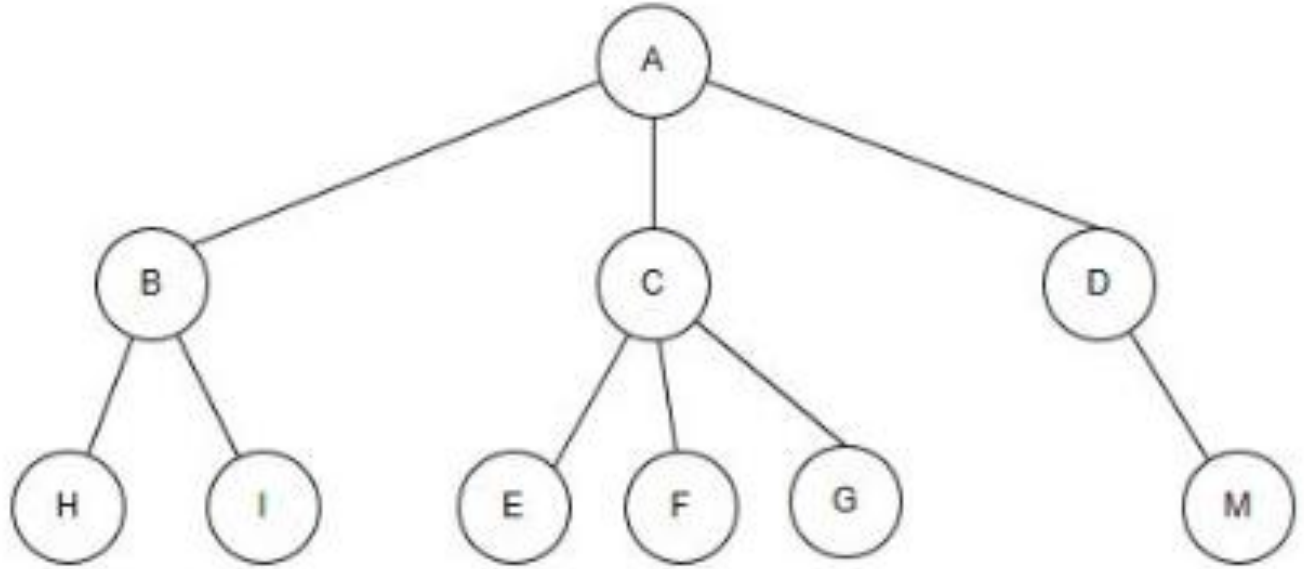
- To convert any tree into a binary tree, use the following rules:
  - make the 1^{st} child of a node its left child
  - make the i^{th} child of a node the right child of the $(i-1)^{st}$ child of the node
- Another way to say this is:
  - make the left child of a node its current left child
  - make the right child of a node its current right sibling

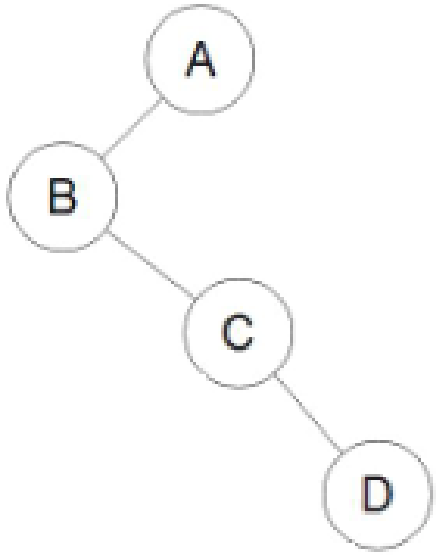
- When using this approach:
  - The preorder traversal of the two trees will be the same.
  - The postorder traversal of the general tree will be the same as the inorder traversal of the new binary tree.


A
B
C
D
H
I
E
F
G
M

B is 1st child
C is 2nd child
D is 3rd child



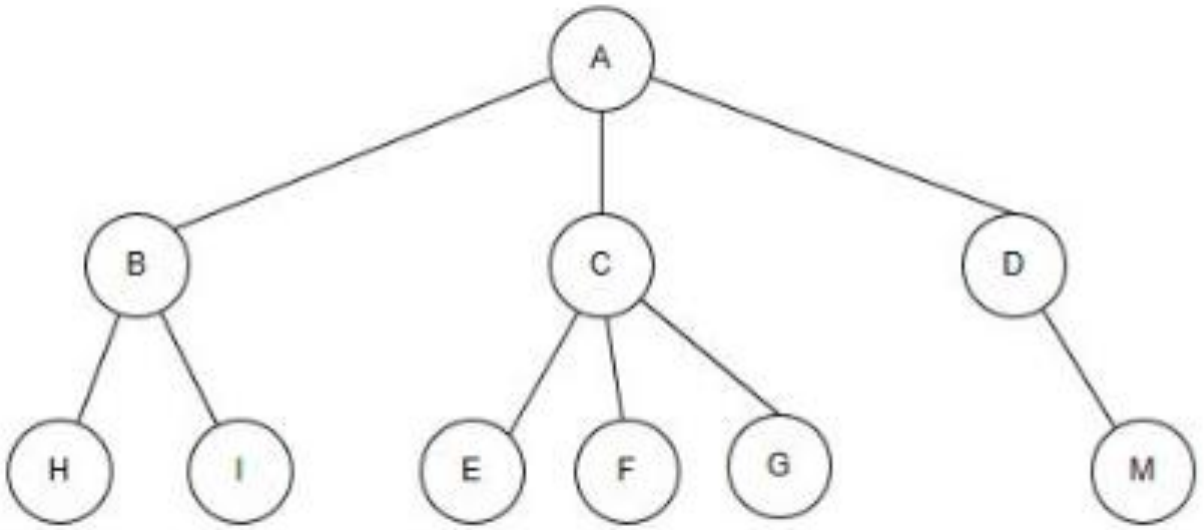


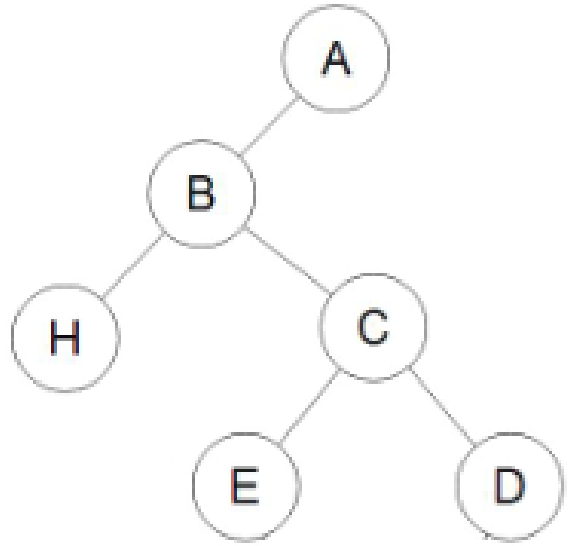


B becomes the left child

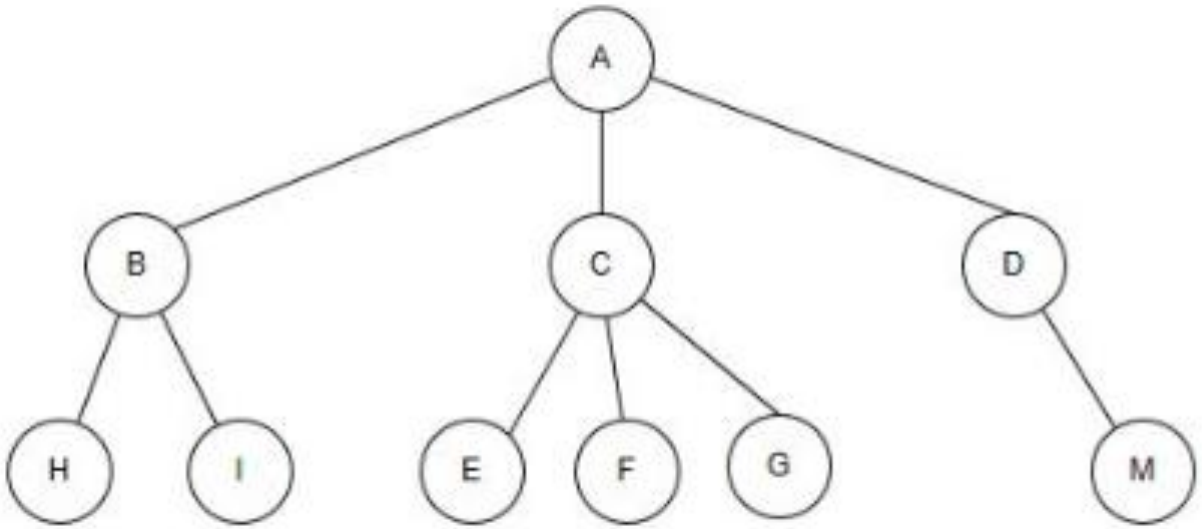
C (2nd) becomes right child of 2-1=1st (B)

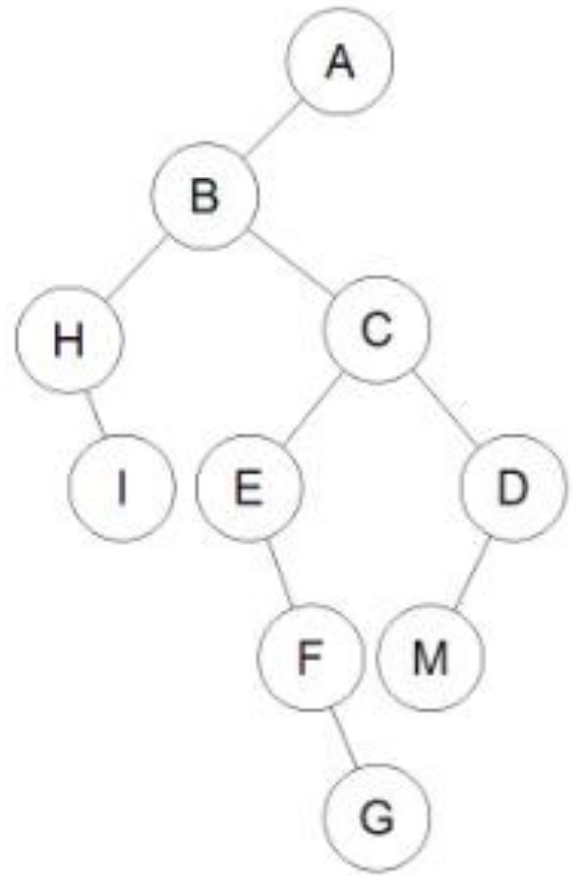
D (3rd) becomes right child of 3-1=2nd (C)

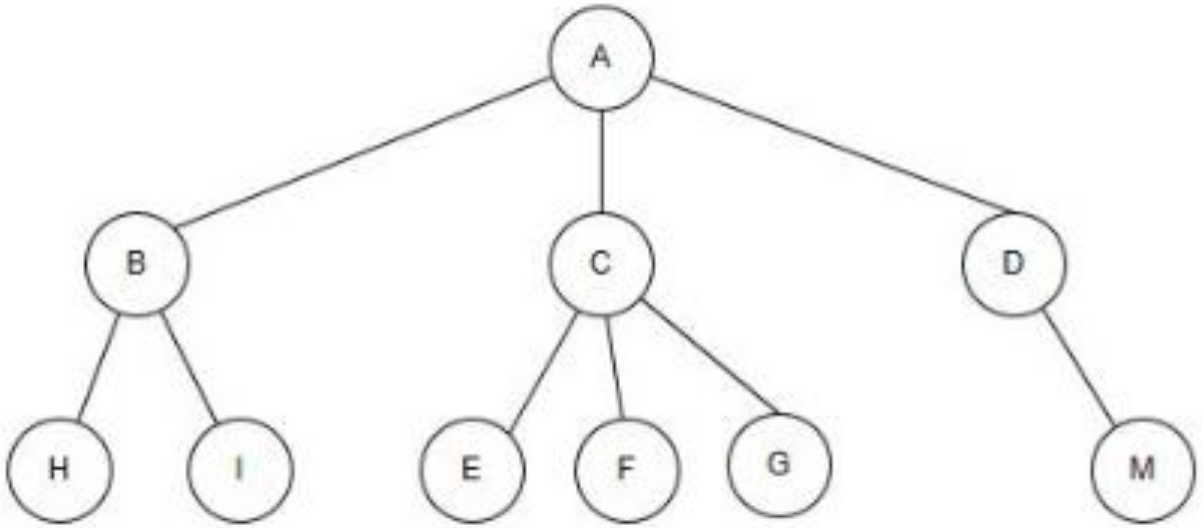




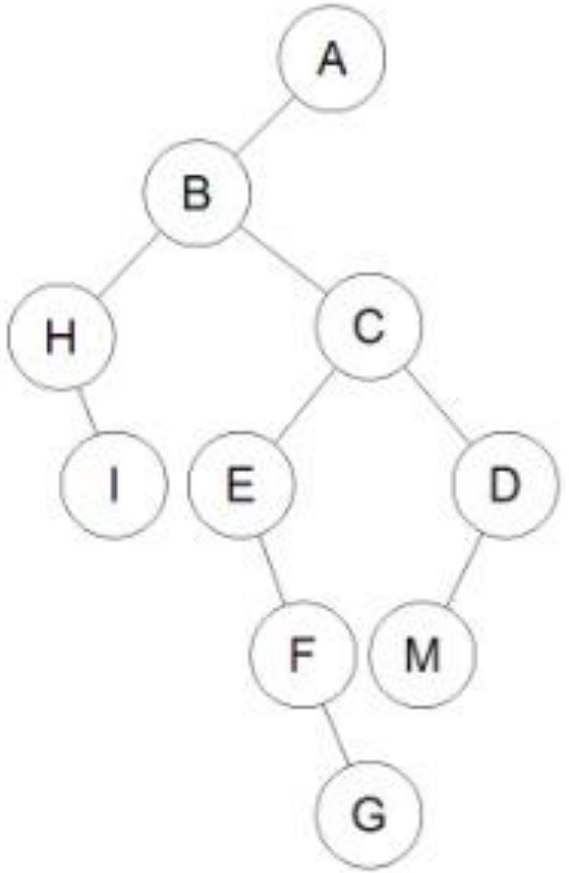
A's new left child is its current left child: B
A's new right child is its current right sibling: none
B's new left child is its current left child: H
B's new right child is its current right sibling: C
C's new left child is its current left child: E
C's new right child is its current right sibling: D


A
B
C
D
H
I
E
F
G
M


A
B
H
C
I
E
D
F
M
G



- Preorder: A B H I C E F G D M
- Postorder: H I B E F G C M D A



- Preorder: A B H I C E F G D M
- Inorder: H I B E F G C M D A

BINARY SEARCH TREES

Binary Search Tree

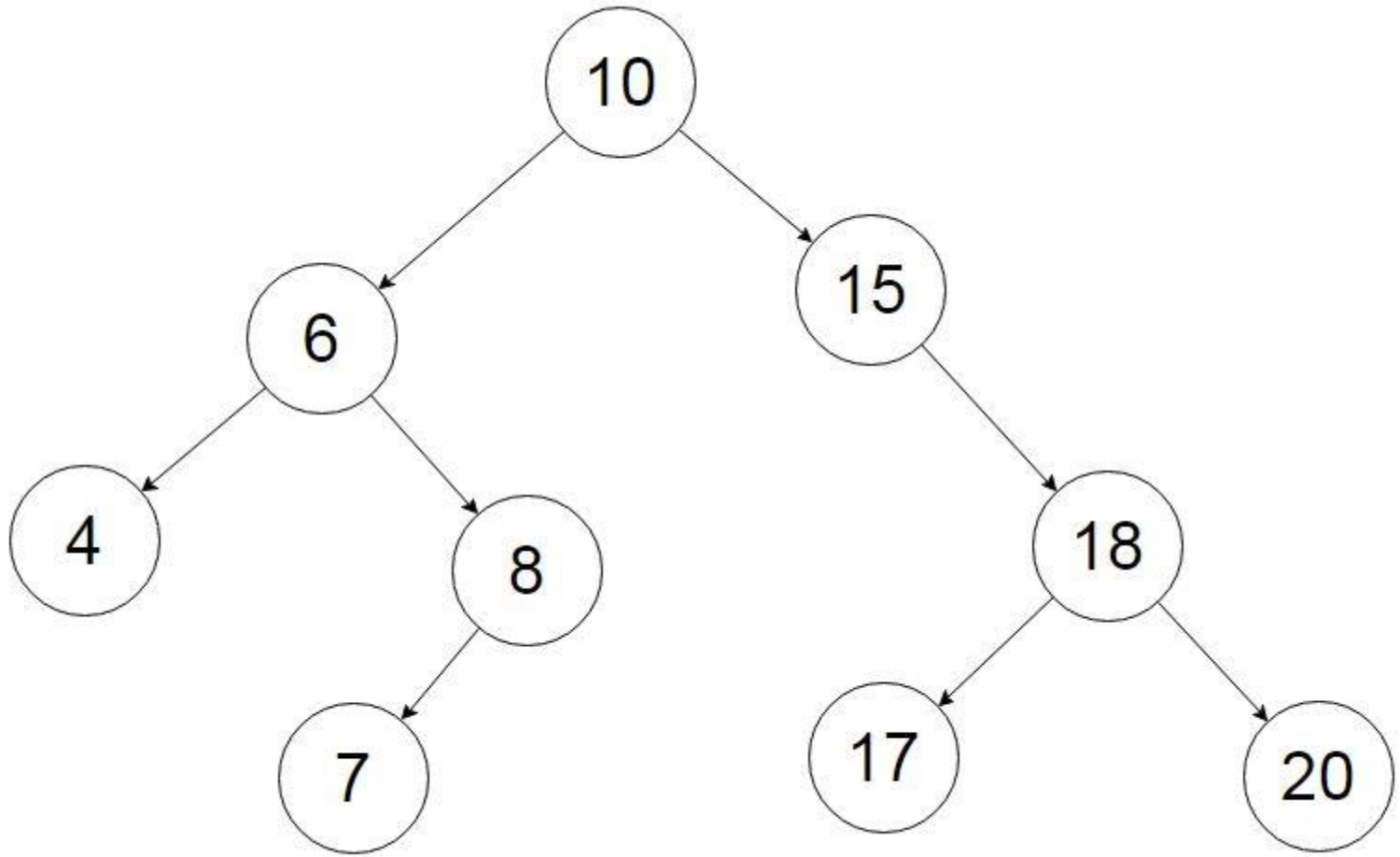
- Binary Tree: a tree such that each node has at most two children (left and right)
- Binary Search Tree: an **ordered** binary tree, such that:
  - **all data** in the left **subtree** of a node is less than the node
  - **all data** in the right **subtree** of a node is greater than the node

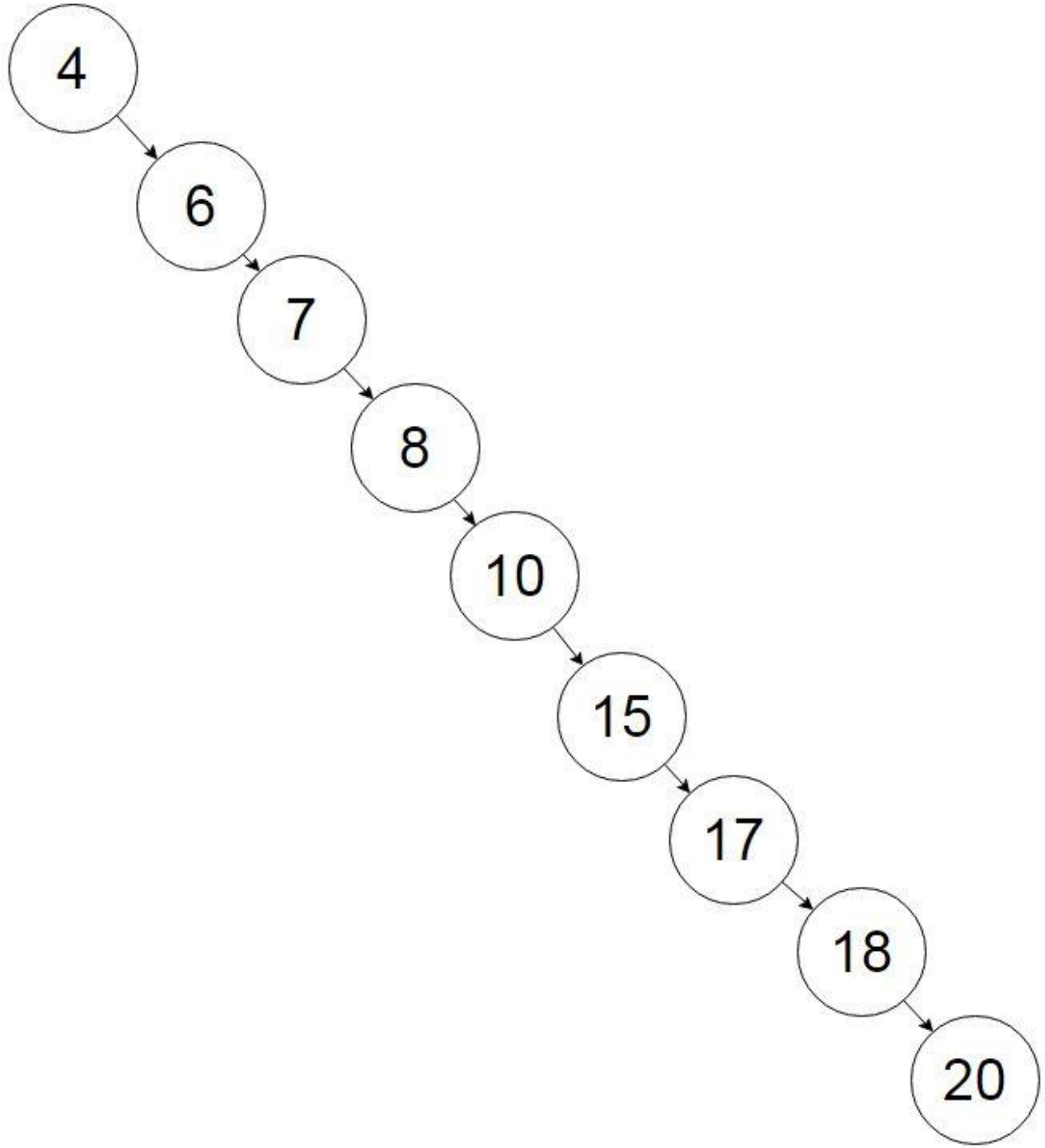
- This rule applies to **all** nodes- not just the root!
- This rule describes **all data in the subtree**- not just the children!

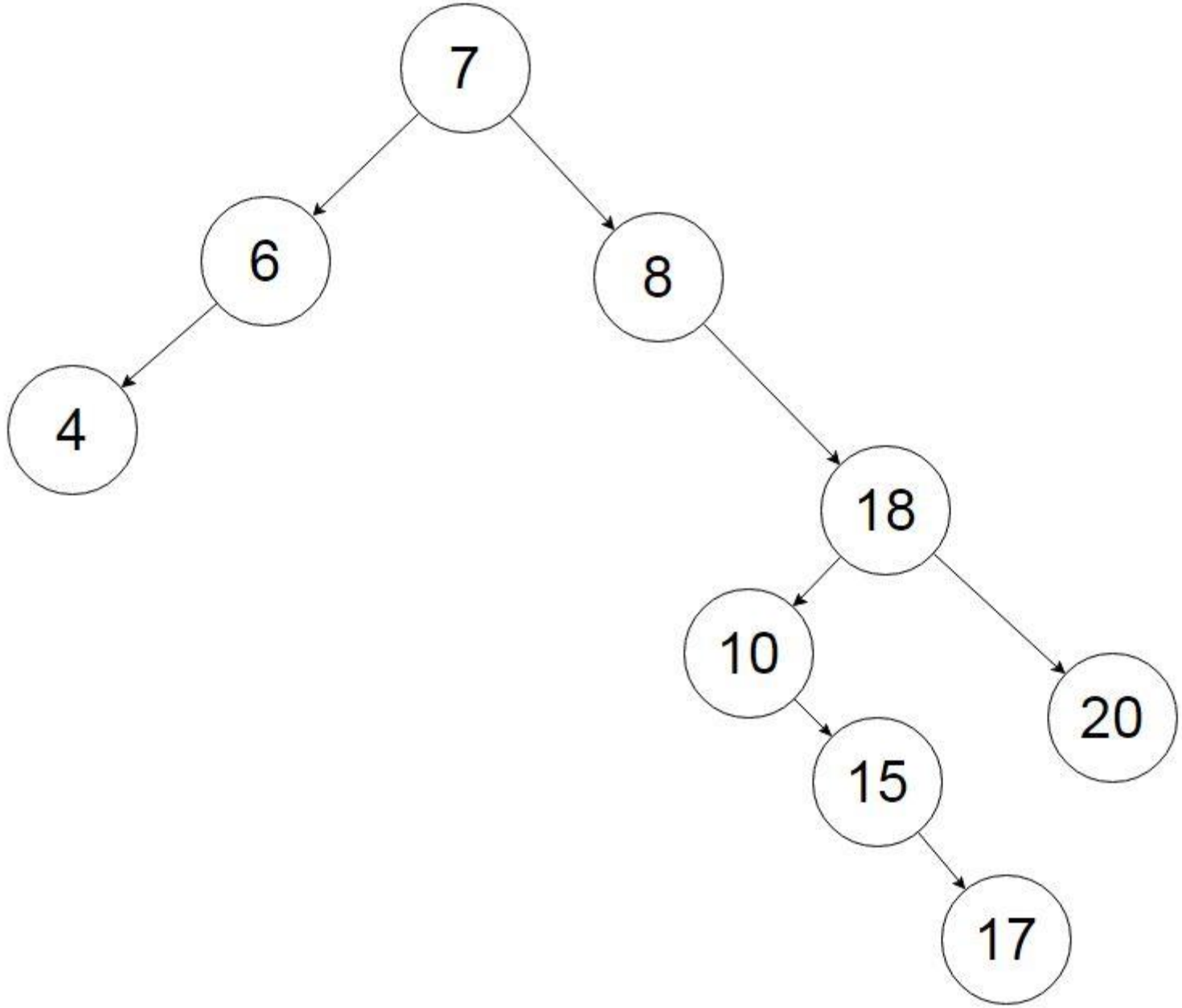
Binary Search Trees (BSTs)

- In many BSTs, there are no duplicates allowed.

- An inorder traversal of a BST gives you the data in ascending sorted order.
- We now have another way to sort data: create a BST and then perform an inorder traversal!

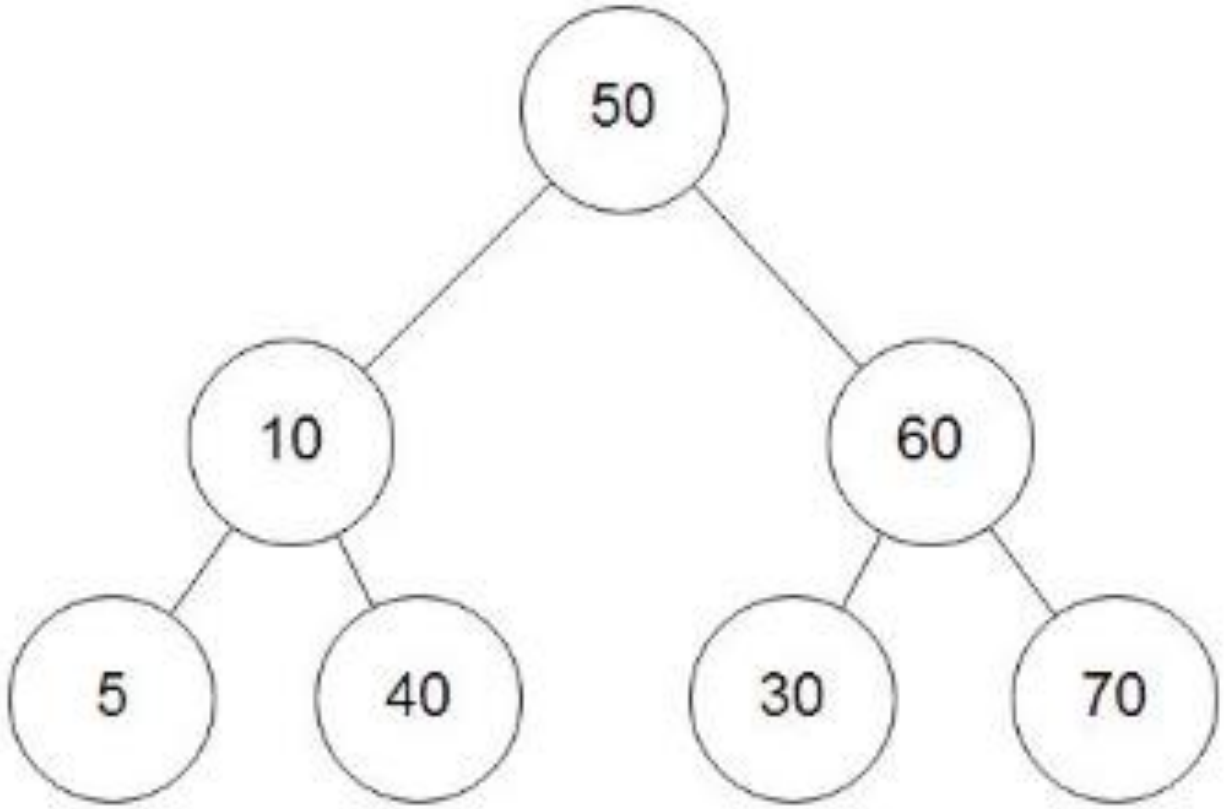

10
6
15
4
8
18
7
17
20


4
6
7
8
10
15
17
18
20

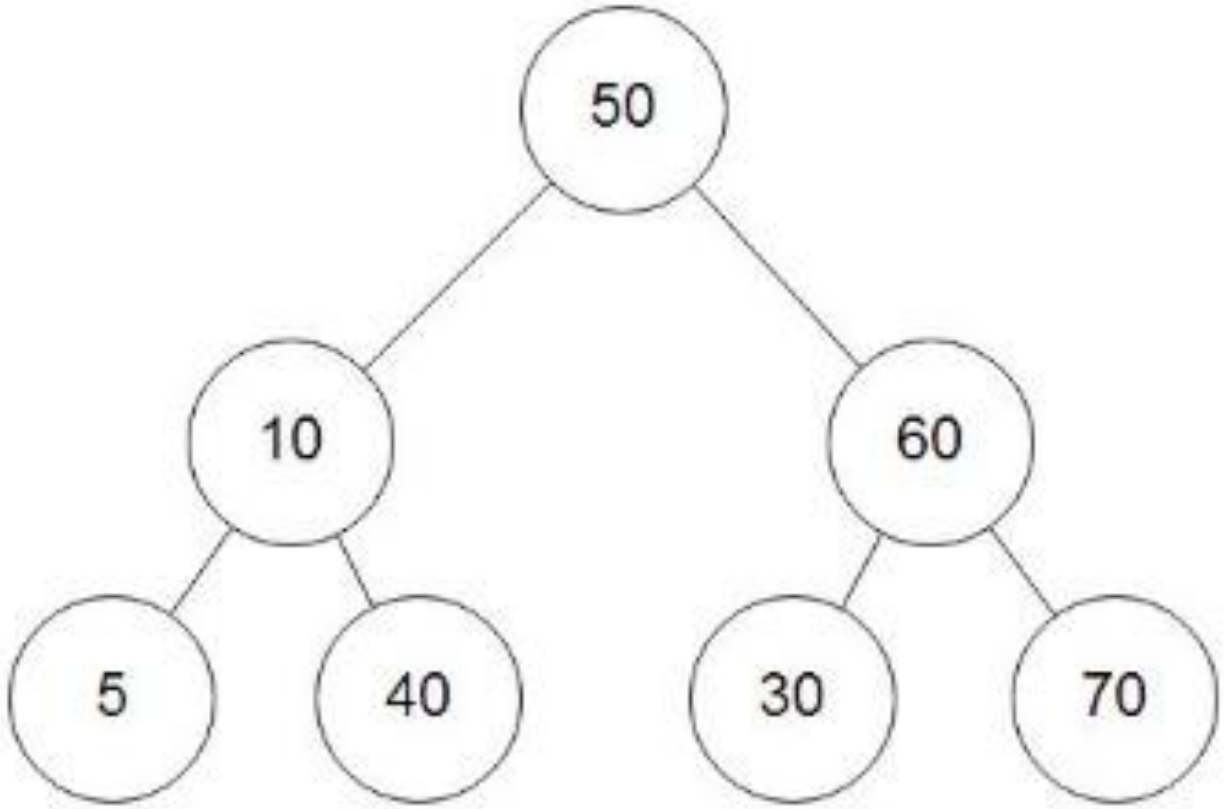

7
6
8
4
18
10
20
15
17

Example

Tree?
Binary Tree?
BST?



Example



Tree? yes

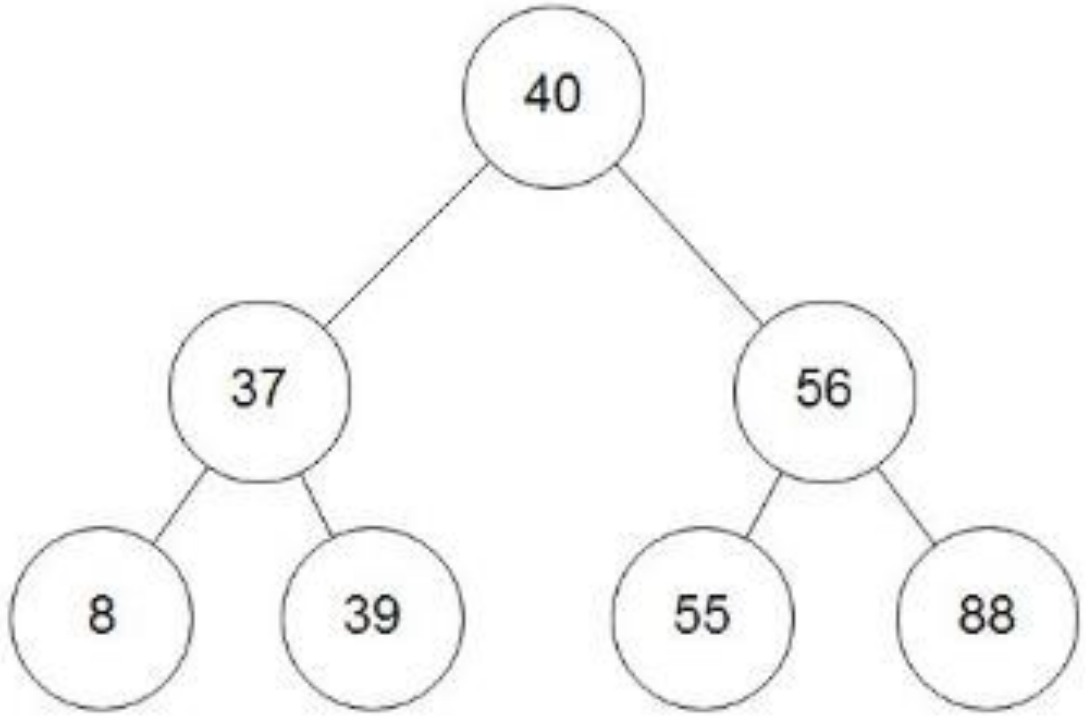
Binary Tree? yes

BST? no!

the 30 node is in the right subtree of 50

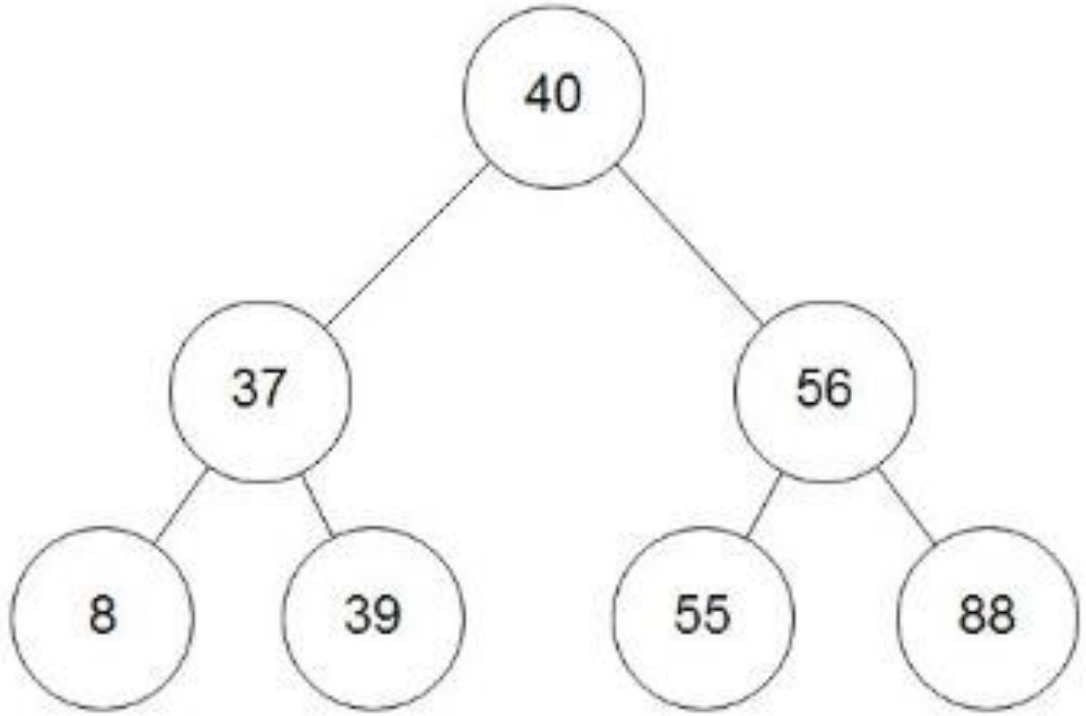
Example

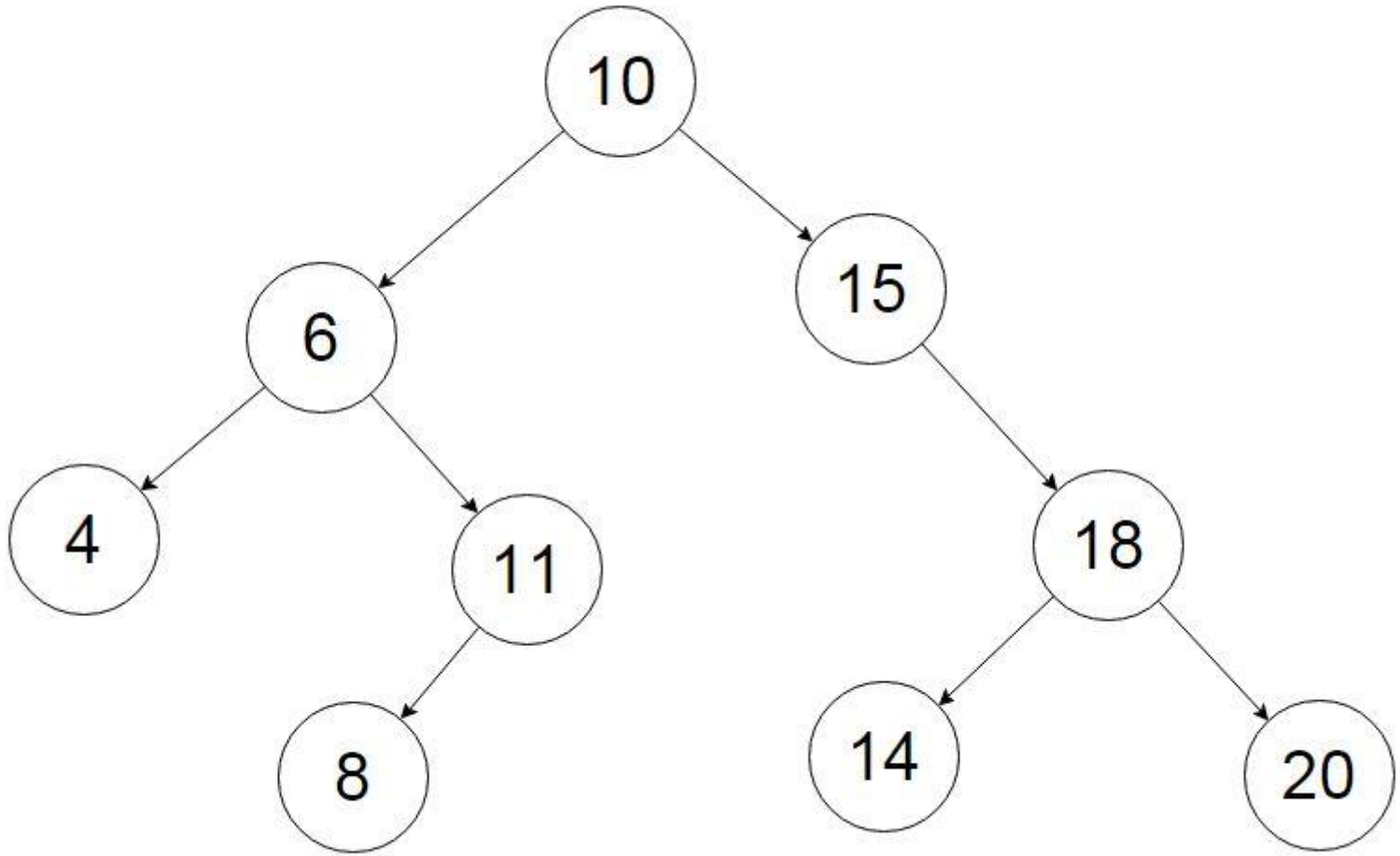
Tree?
Binary Tree?
BST?

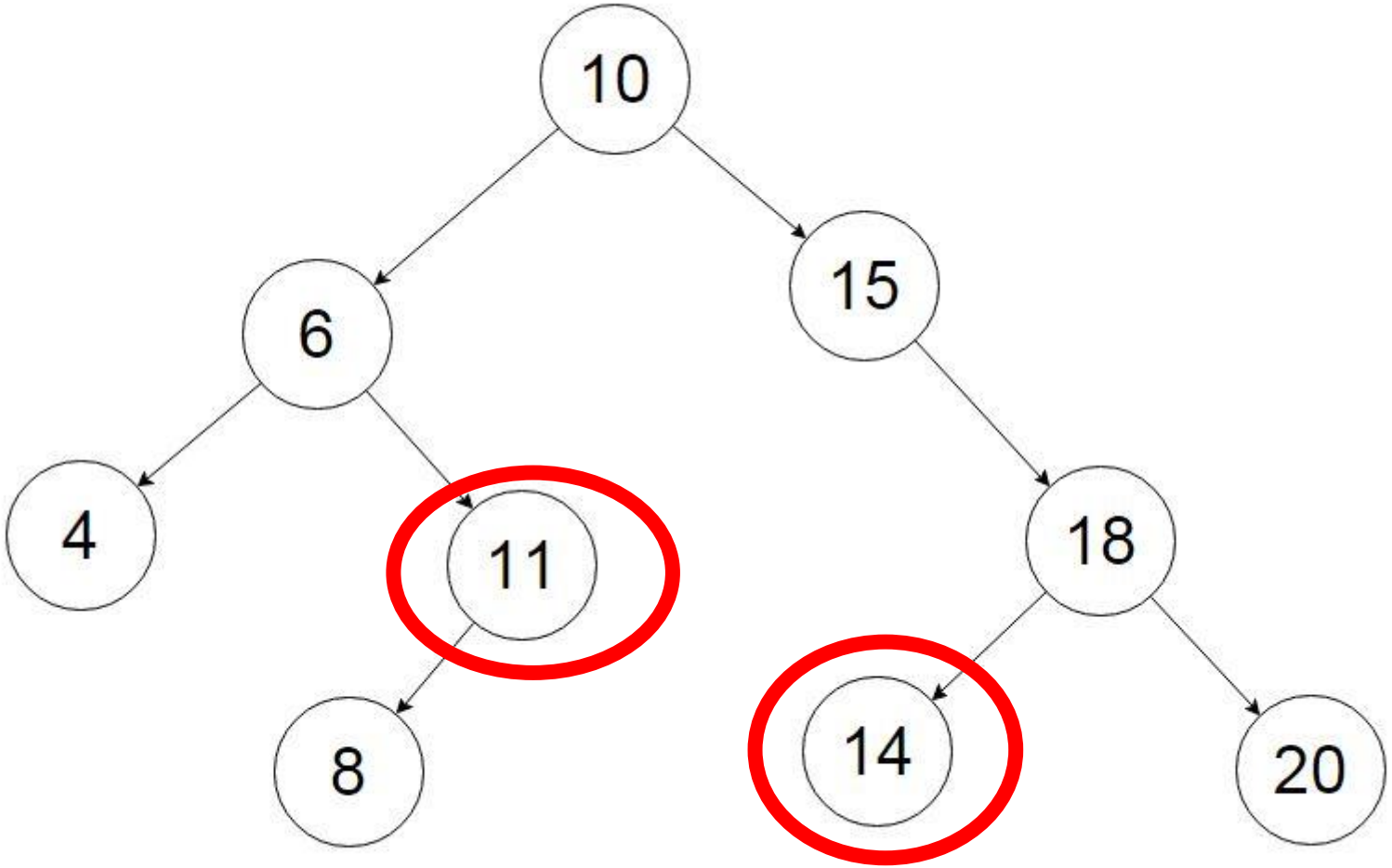


Example

Tree?	yes
Binary Tree?	yes
BST?	yes




10
6
15
4
11
18
8
14
20


10
6
15
4
11
18
8
14
20

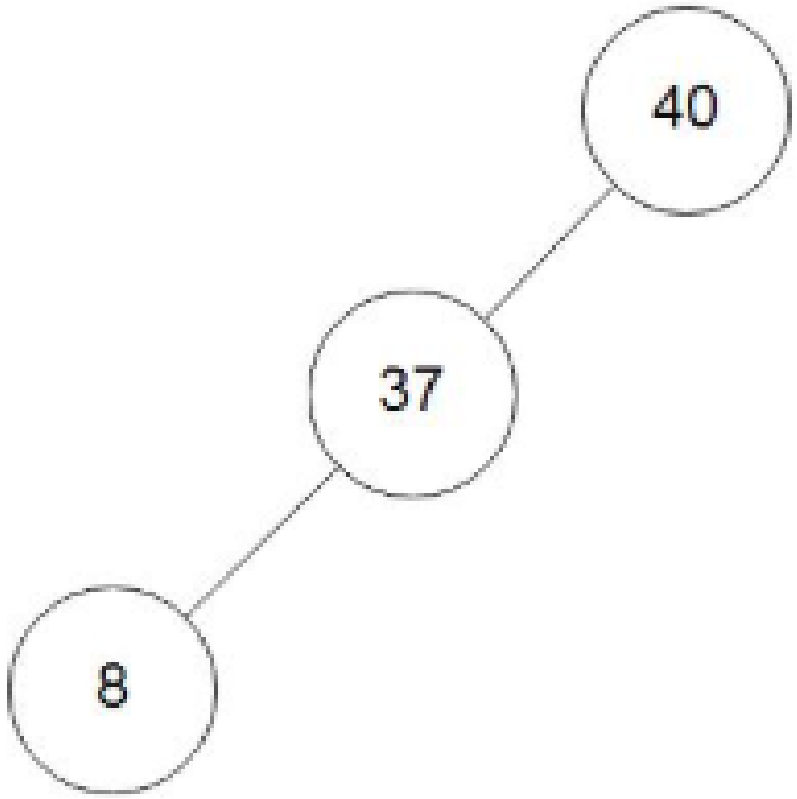
Searching a BST

- An advantage of storing data in a BST is that search can be fast.
- We never need to backtrack up a tree to find a value- we always know which branch to follow.

- But not all BST searches will be efficient!

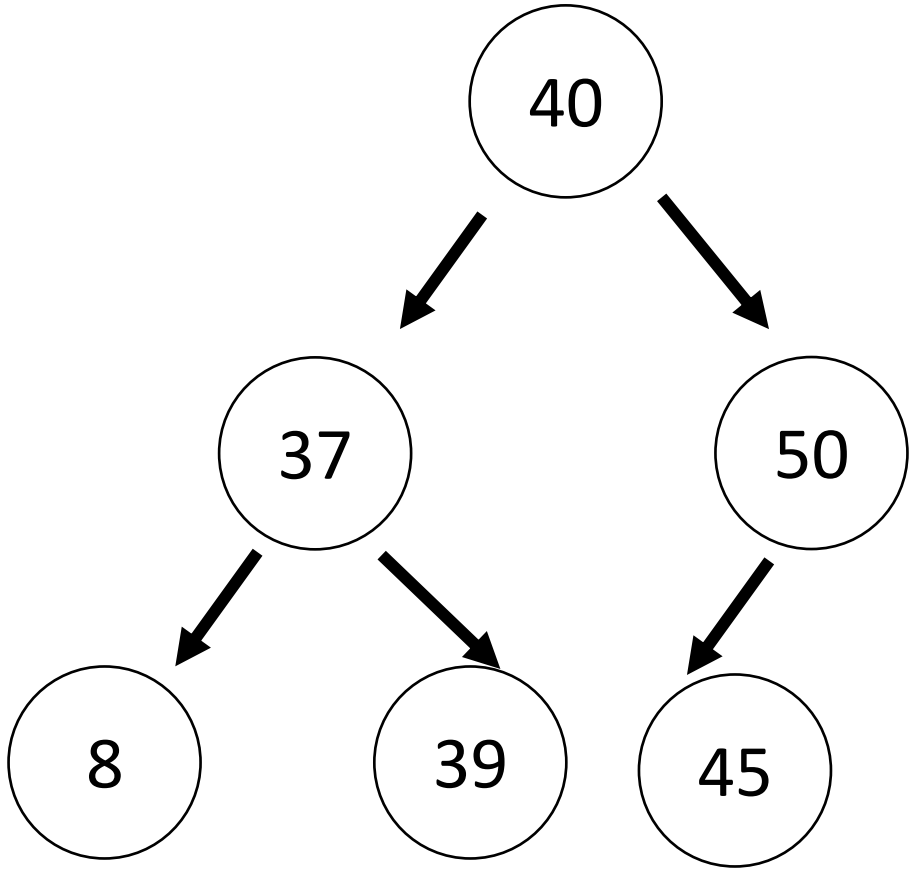
Searching a BST

- The worst case for searching a BST is O(n).



Searching a BST

- The worst case for searching a **complete** BST is O(log n)!



Balance

- *Balance* is important.
  - It allows for more efficient searches! (And insertions and deletions!)
- You can either keep a tree always in balance or you can build a tree and then balance it.
- More to come on balance…

BST Search Algorithm

```
public T bstSearch(Node<T> root, T searchValue) {
    Node<T> currentNode = root;
    while(currentNode != null) {
        if(searchValue.compareTo(currentNode.data) < 0 ) ) {
            currentNode = currentNode.left;
        } else if(searchValue.compareTo(currentNode.data) > 0)) {
            currentNode = currentNode.right;
        } else {
            return currentNode.data;
        }
    }
    return null;  // the search value is not in the tree
}
```

Inserting into a BST

- BSTs are a “place for everything and everything in its place” kind of structure.
- There is **one unique** BST for any given **sequence** of input data.
  - Meaning any particular *insertion order* of data.
  - *Unique* means one-of-a-kind. There is only one possible correct BST.

- Note: there is **not** one unique BST for any *set* of data- only any *sequence* of data.

- To insert into a BST, you perform a search and then insert the node where you expected to find it!

Example

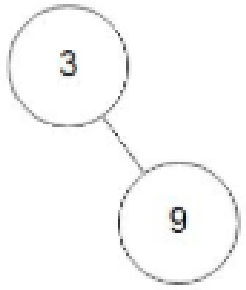
- Data input sequence: 3 9 17 2 4 1

Example

- Data input sequence: 3 9 17 2 4 1

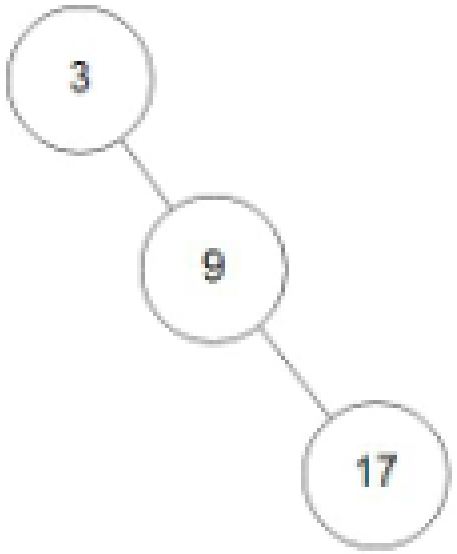
Example

- Data input sequence: 3 9 17 2 4 1



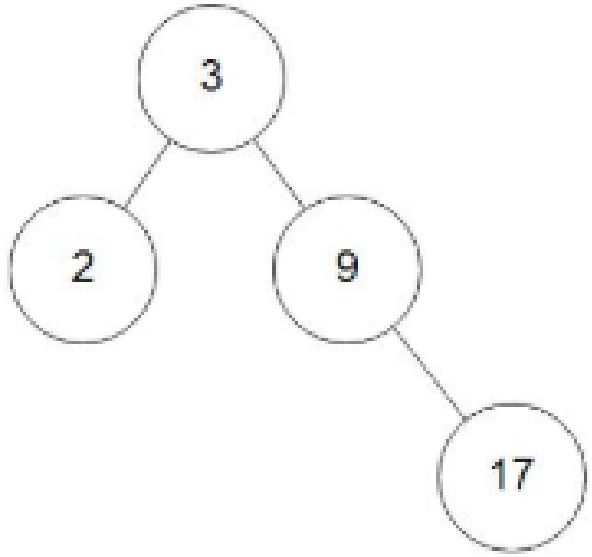
Example

- Data input sequence: 3 9 17 2 4 1



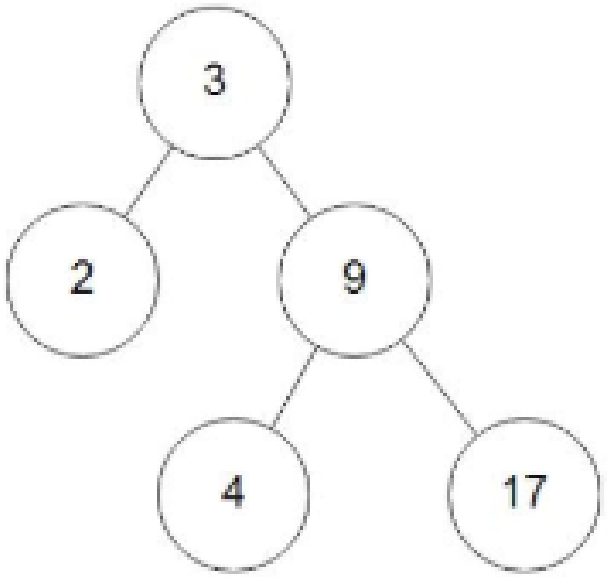
Example

- Data input sequence: 3 9 17 2 4 1



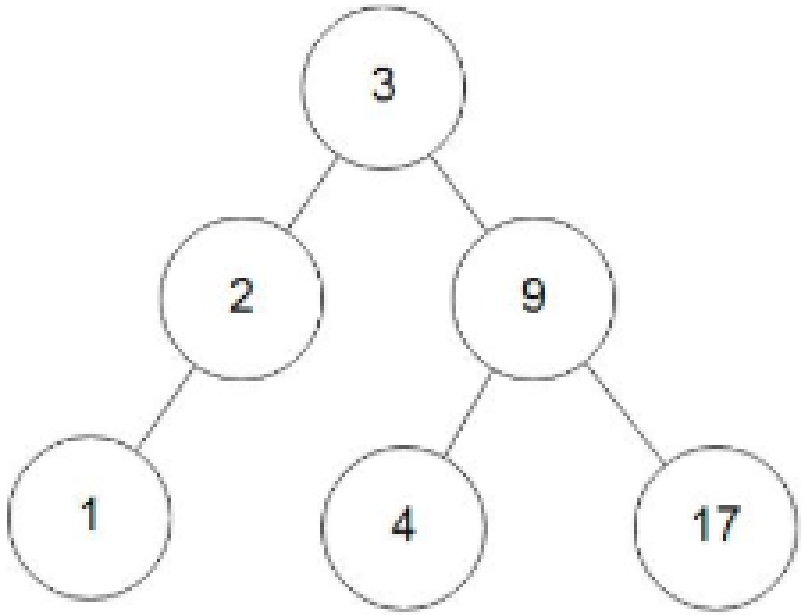
Example

- Data input sequence: 3 9 17 2 4 1



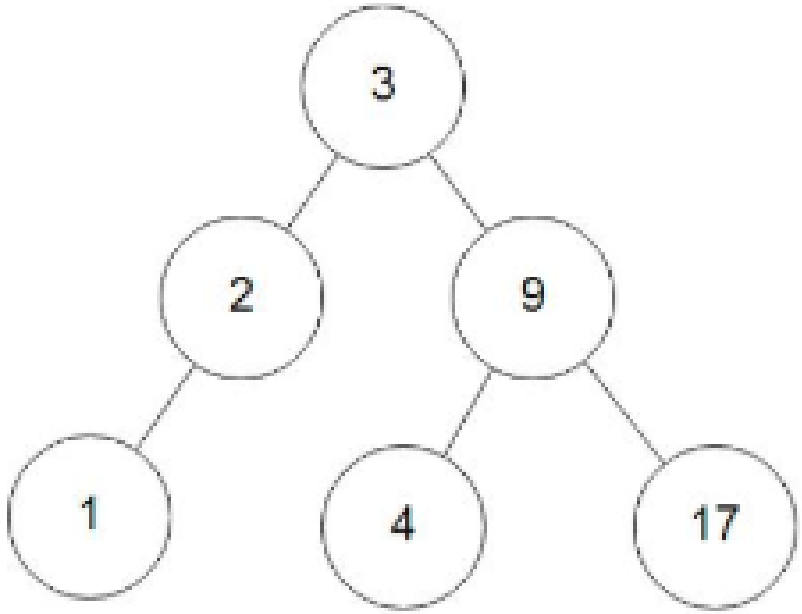
Example

- Data input sequence: 3 9 17 2 4 1



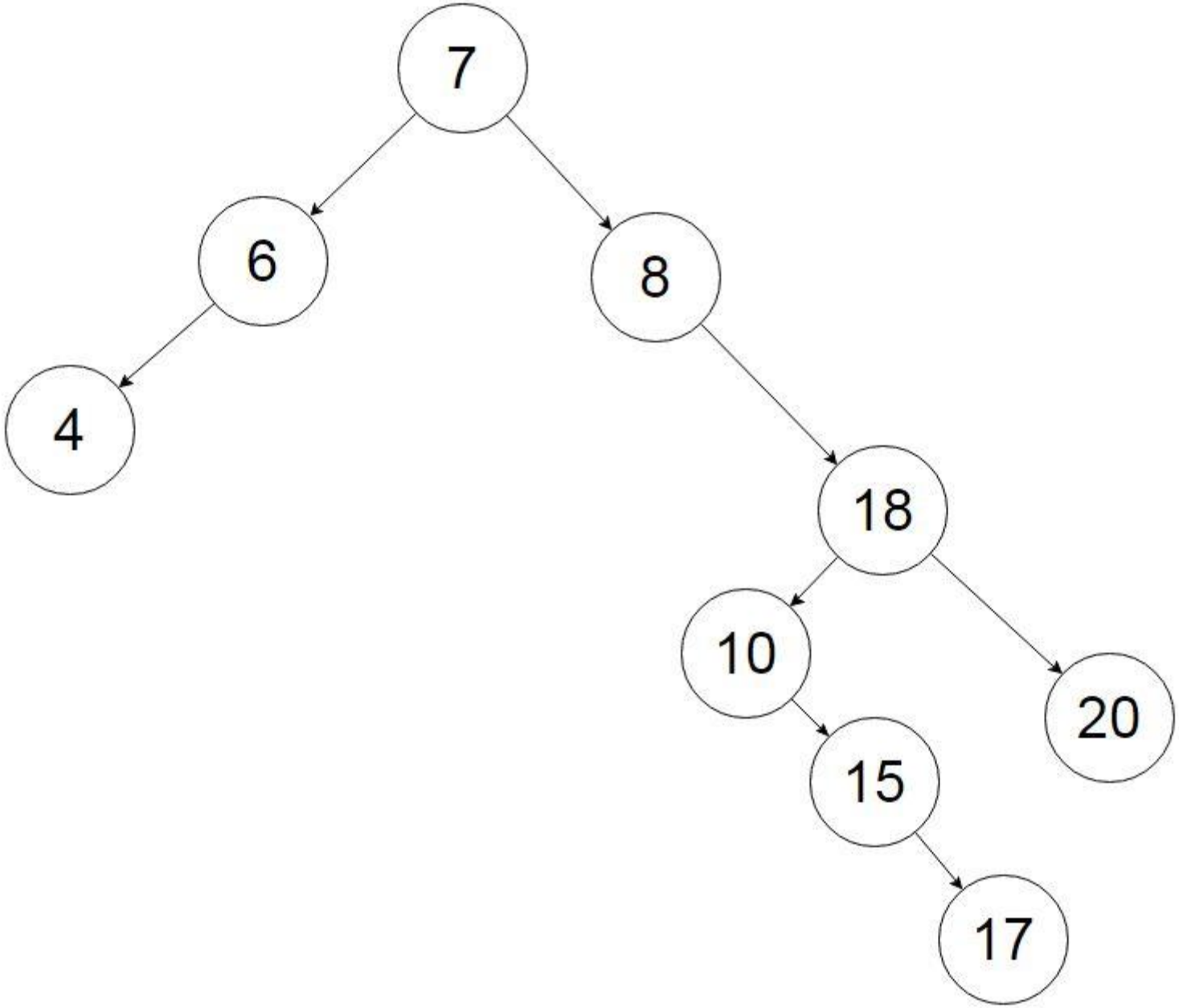
Example

- This answer is unique!

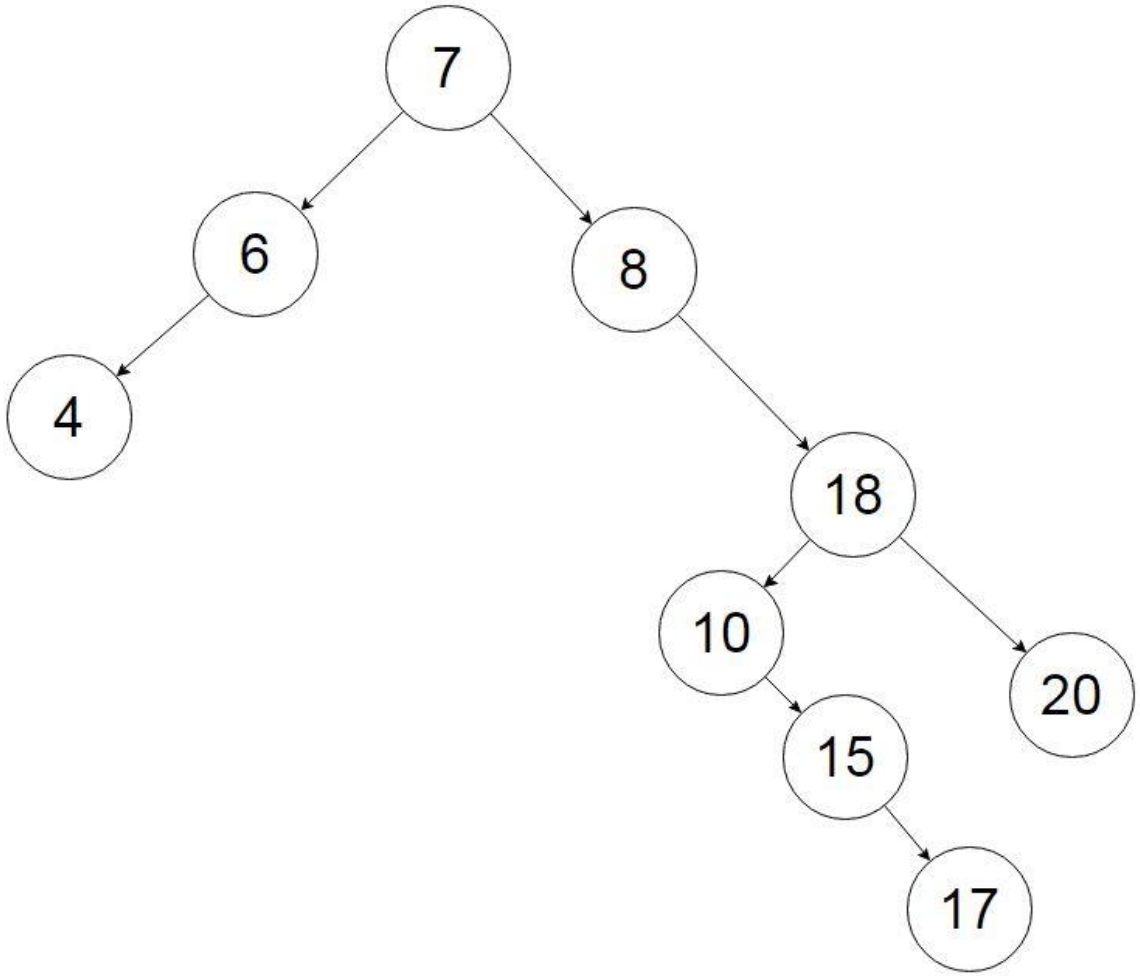


- What would the BST tree look like for this same data but in a different sequence? 17 9 3 2 1 4
  - (answer not shown on slides)

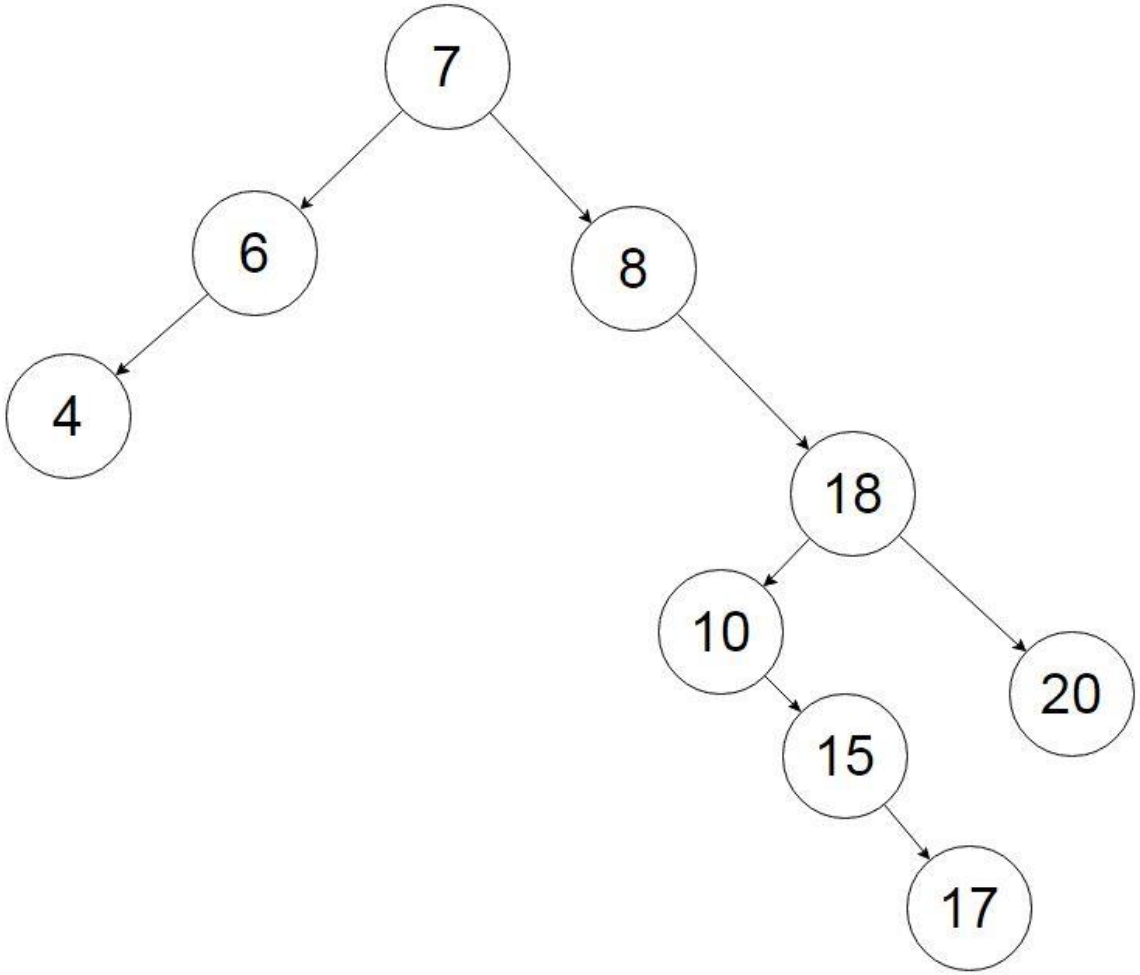
A BST



Example: Add 12



Example: Add 12, 13, 5



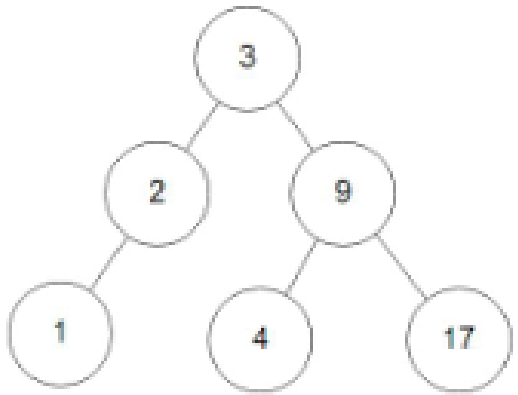
Removing from a BST

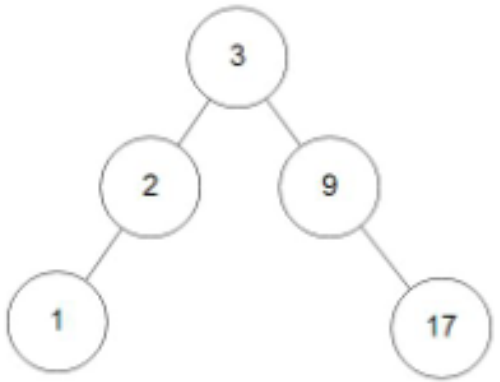
- We need to handle deleting a leaf (node with 0 children), a node with one child, and a node with two children.

Removing a Leaf

- Set the parent's new child to null
- Confirm you still have a valid BST.

- Example: delete 4

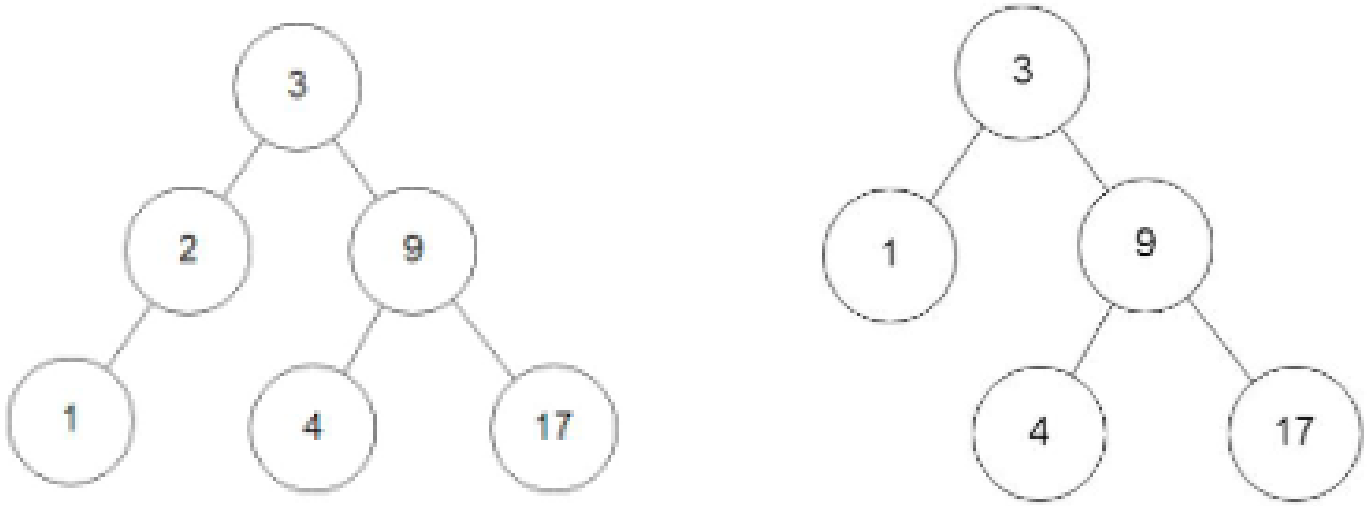




Removing a Node with One Child

- Set the parent's child to be the deleted node's child.
- Confirm you still have a valid BST.

- Example: delete 2

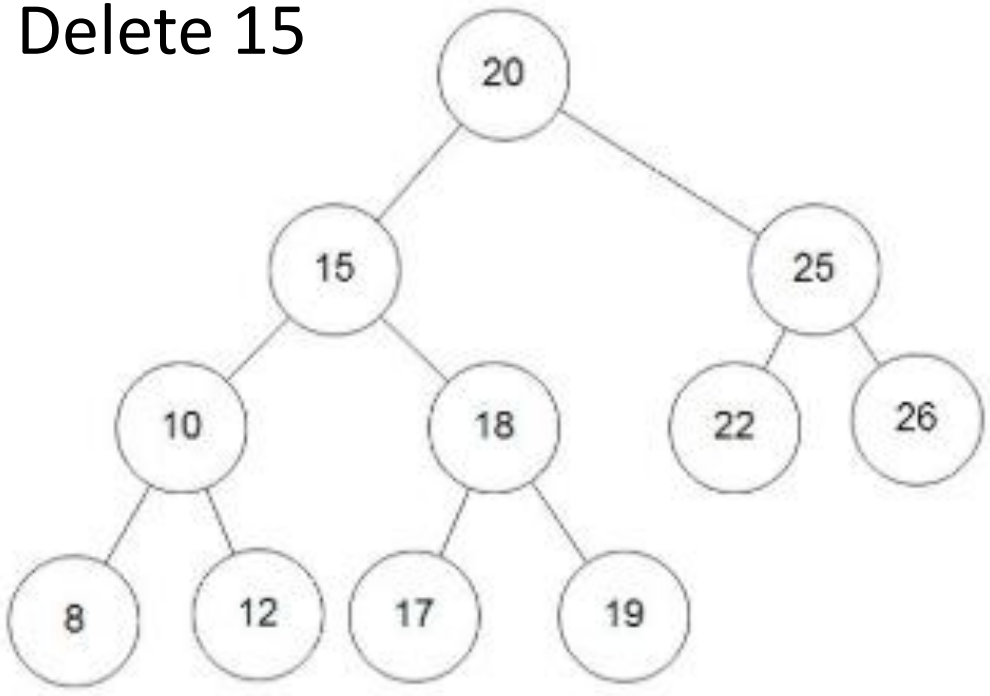


Removing a Node with Two Children

- Option A: Set the parent's new child to be the *inorder successor*
  - a *successor* is the node that comes after
  - set the parent's new child to be the left-most node in the deleted node's right subtree
- Confirm you still have a valid BST.

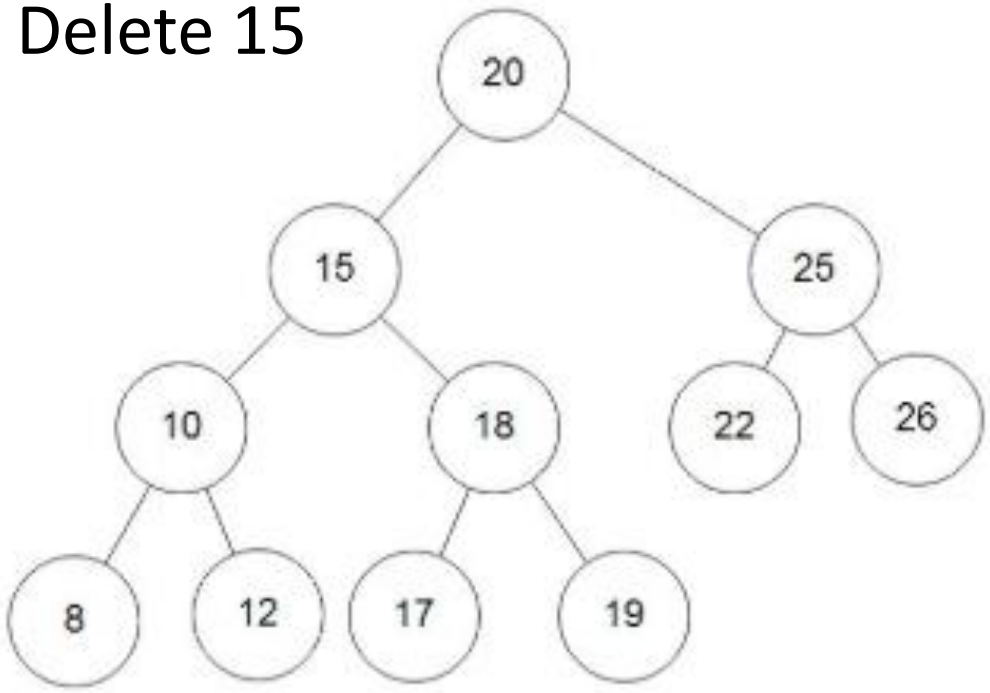
Removing a Node with Two Children

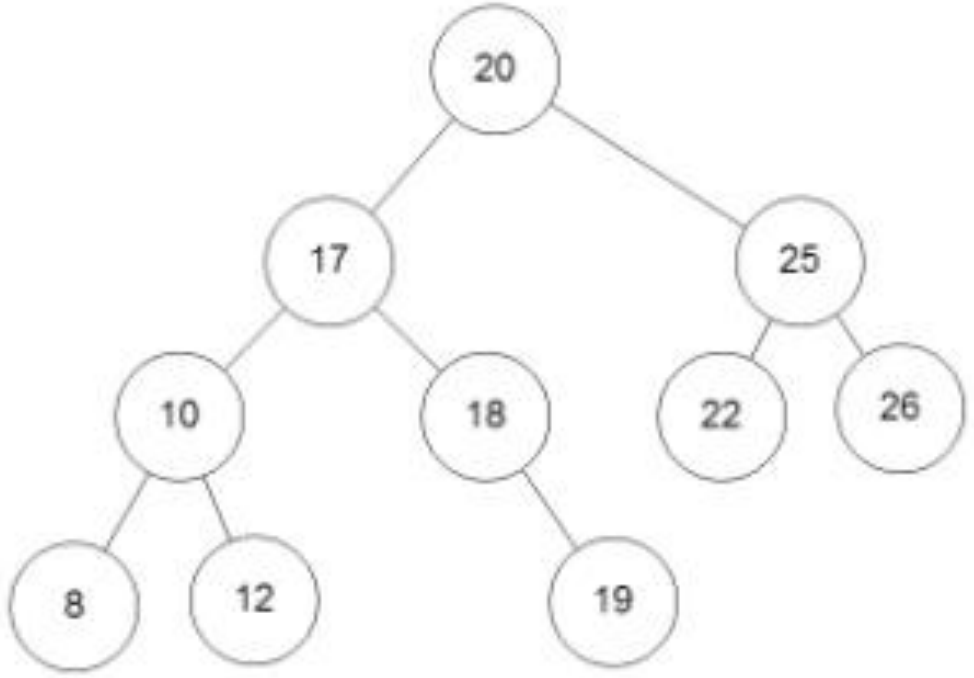
- Option A: Set the parent's new child to be the *inorder successor*
  - set the parent's new child to be the left-most node in the deleted node's right subtree
- Delete 15



Removing a Node with Two Children

- Option A: Set the parent's new child to be the *inorder successor*
  - set the parent's new child to be the left-most node in the deleted node's right subtree
- Delete 15



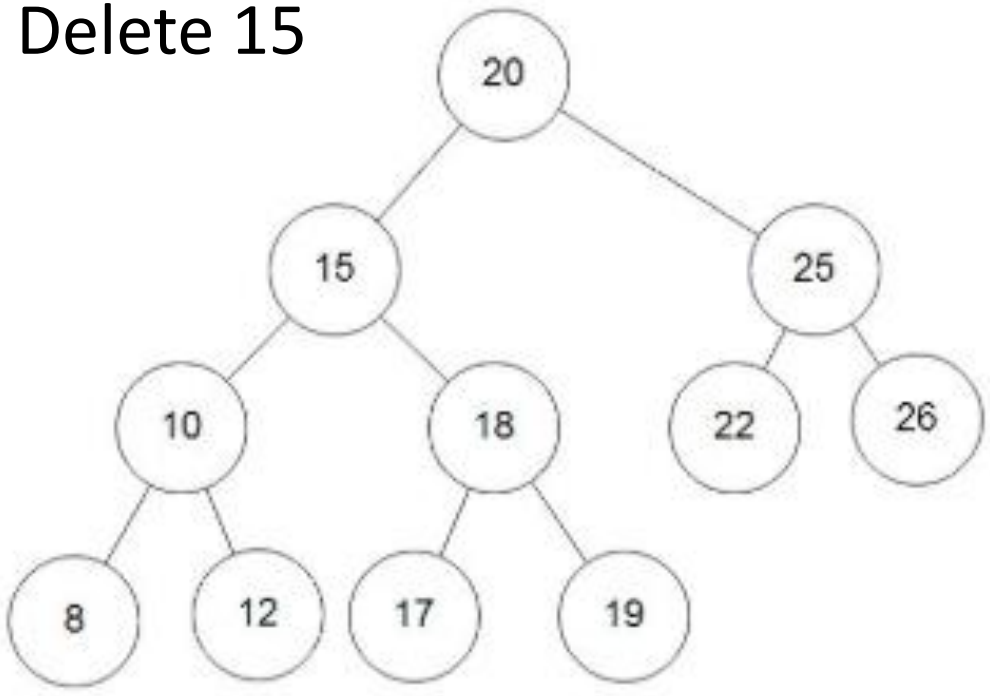


Removing a Node with Two Children

- Option B: Set the parent's new child to be the *inorder predecessor*
  - a *predecessor* is the node that comes before
  - set the parent's new child to be the right-most node in the deleted node's left subtree
- Confirm you still have a valid BST.

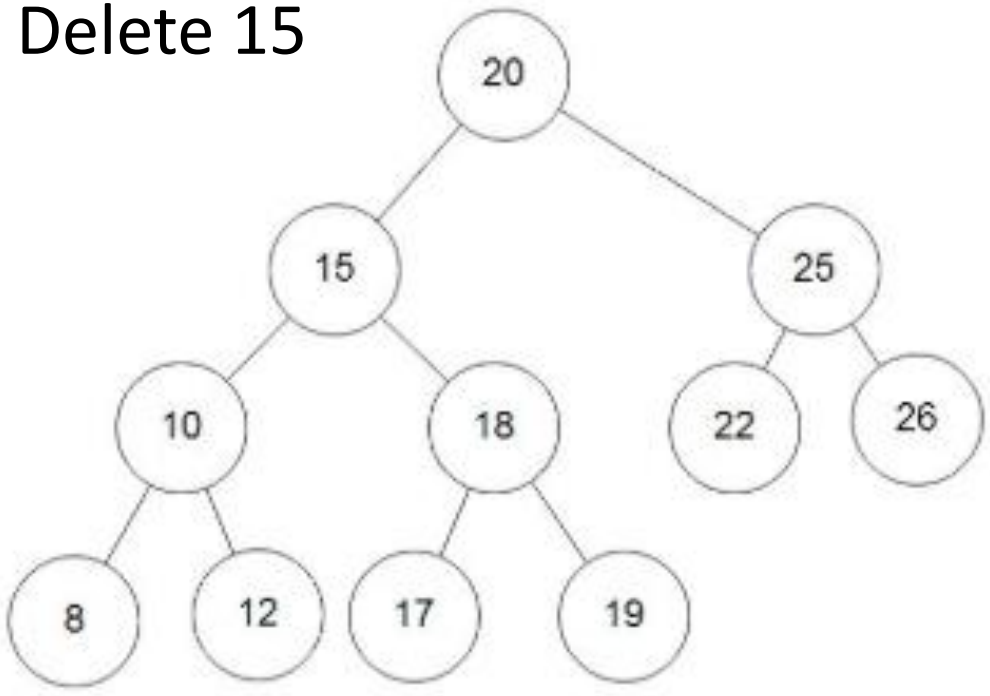
Removing a Node with Two Children

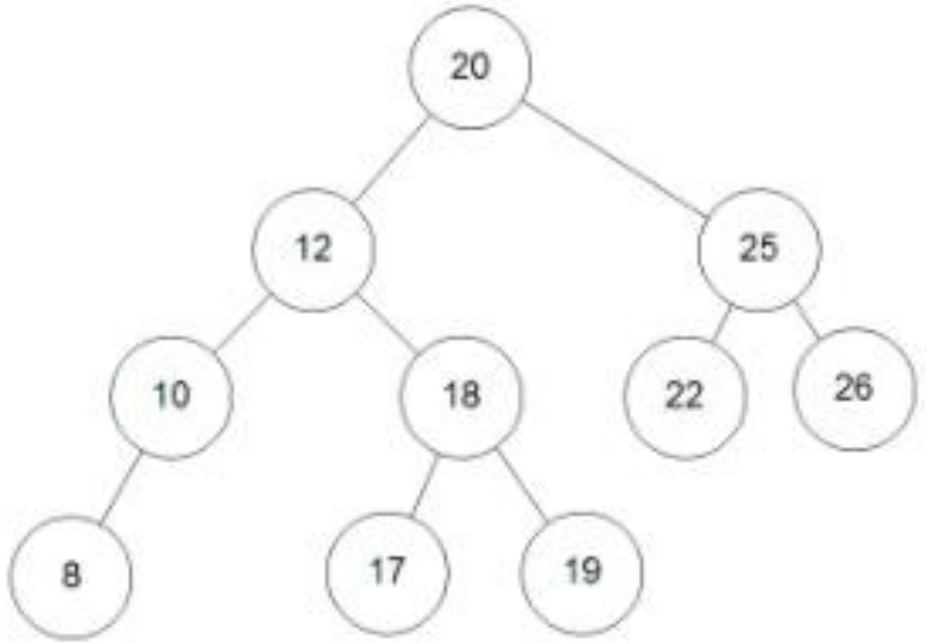
- Option B: Set the parent's new child to be the *inorder predecessor*
  - set the parent's new child to be the right-most node in the deleted node's left subtree
- Delete 15



Removing a Node with Two Children

- Option B: Set the parent's new child to be the *inorder predecessor*
  - set the parent's new child to be the right-most node in the deleted node's left subtree
- Delete 15

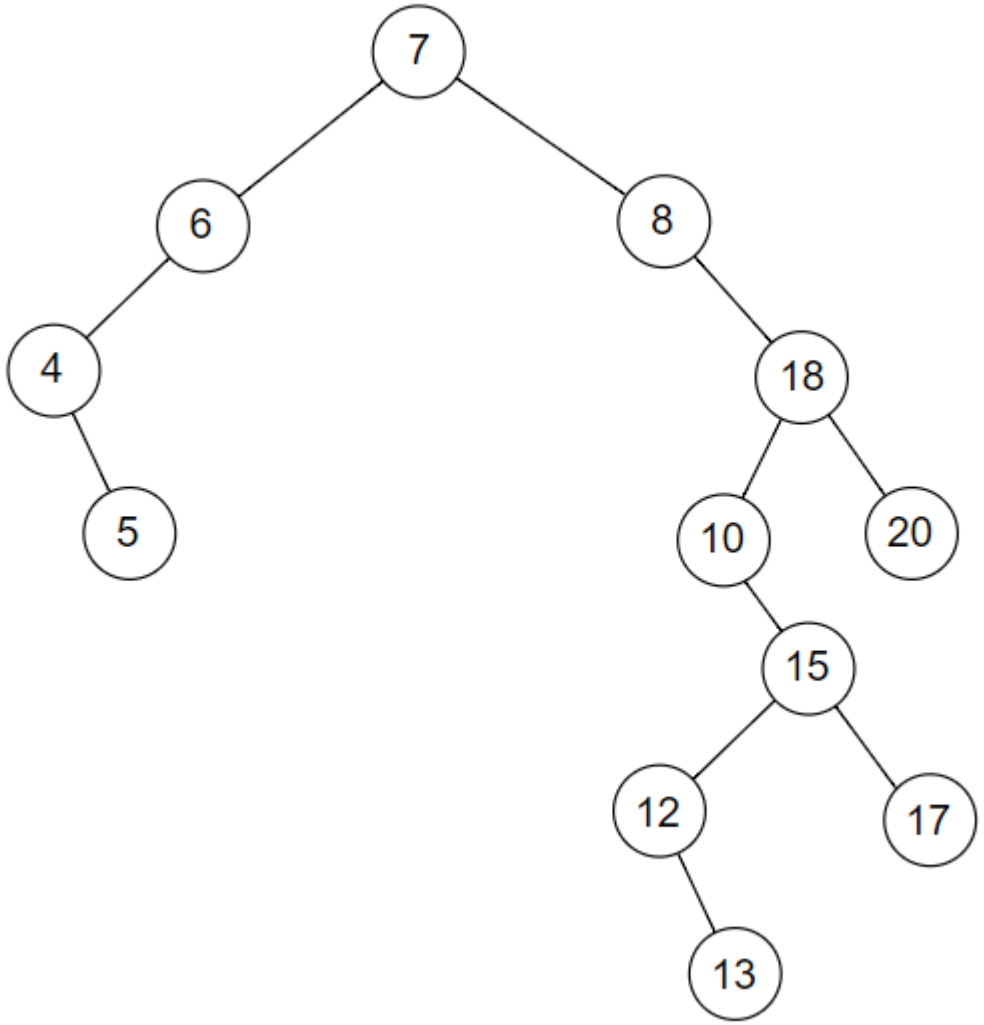




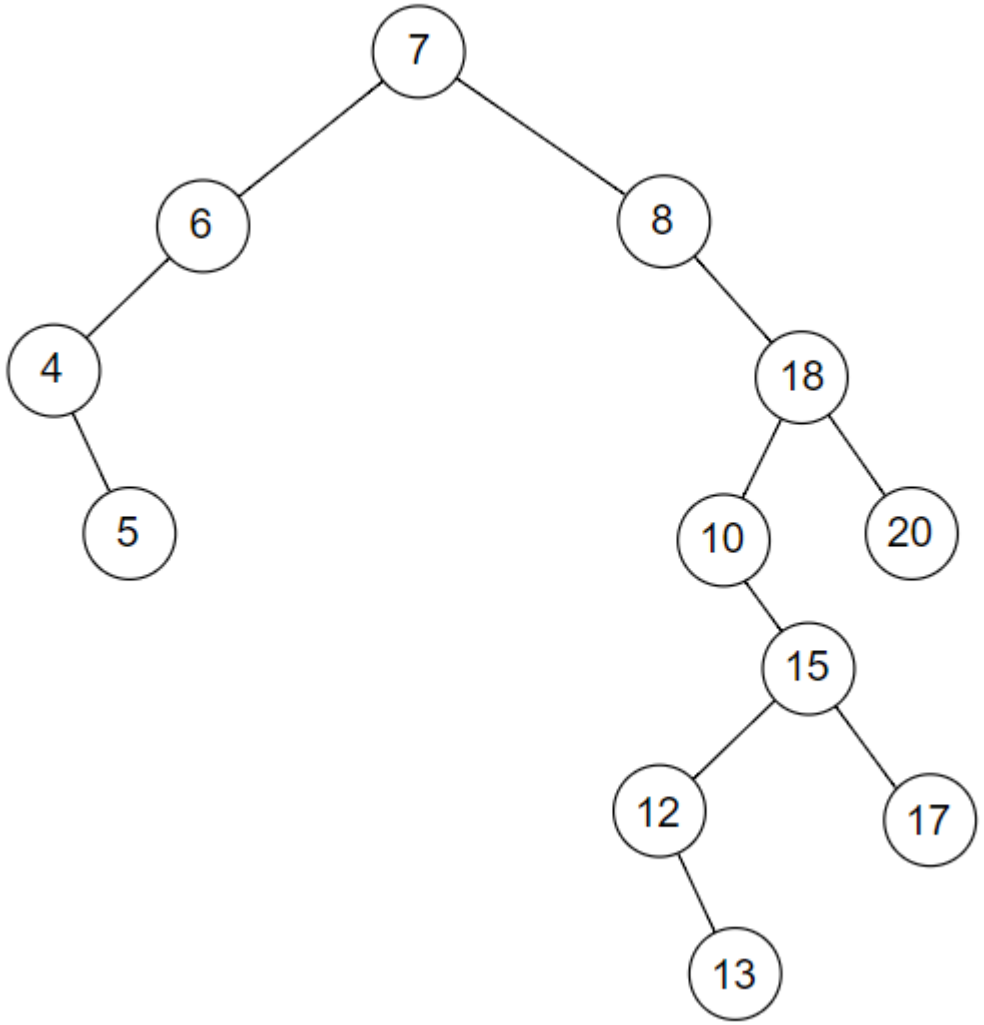
Removing a Node from a BST

- Use these algorithms!
  - You might find a different node to swap in that works in a particular case, but it isn’t guaranteed to work in all cases!
- Always double-check that you still have a valid BST.

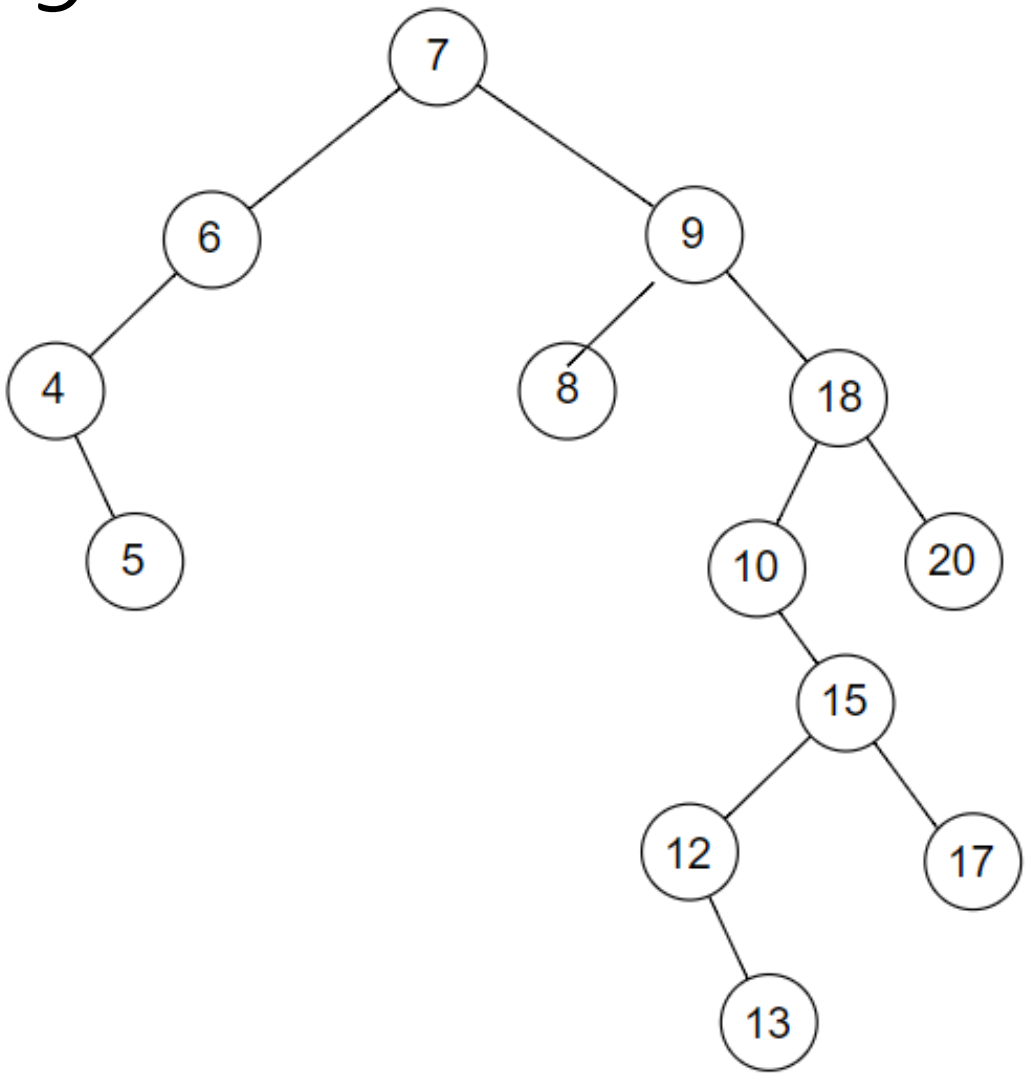
Example: Remove 18



Example: Remove 18



Example: Delete 9



Example: Delete 9