Maps & BST

CS143: lecture 9

Byron Zhong, July 3

Lists

Recap

- Lists: [🍏, 🍎, 🍐, 🍊, 🍋]
- It is an ordered collection of elements:
  - Ordered: 1st, 2nd, 3rd, ...
  - Elements can be homogeneous or heterogeneous.
- Elements are referred to by their *index*
- What if we want to use something other than a number?

Maps

- What if we want to build a *mapping* between one element to another element?
- { 🍏 : ⚽, 🍎 : 🏀, 🍐 : 🏈, 🍊 : ⚾, 🍋 : 🥎 }
- Maps!
    - aka dictionaries, associative array...
- A map is a data structure that stores key-value pairs
    - Each key appears at most once

Maps

Operations

- `insert(k, v)`
- `remove(k)`
- `lookup(k)`
- `size`
- `traverse (to visit all)`

Maps

Can we use lists?

- Yes!
- Each element of the list can be a pair (key, value)
- `insert(k, v)`:
    - `append((k, v))`
- `lookup(k)`:
    - Go through the entire list and compare each `k`
- `remove(k)`:
    - `lookup(k)` and `remove`

Maps

Complexity

	lookup		insert		remove	
	average	worst	average	worst	average	worst
ArrayList	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Linked List	O(n)	O(n)	O(1)	O(1)	O(1)	O(1)

Maps

Can we do better with lists?

- What if we can sort the keys?
- Lookup is faster
  - We can do binary search

Binary Search

Find 19

1	4	6	7	9	12	17	19	25	30	35

Binary Search

Find 19

1	4	6	7	9	12	17	19	25	30	35

Binary Search

Find 19

1	4	6	7	9	12	17	19	25	30	35

Maps

Can we do better with lists?

- What if we can sort the keys?
- Lookup is faster
  - We can do binary search
  - To search a sorted list with n elements, we only need $O(\log_2 n)$
- However
  - ArrayList is bad at insert
  - Linked list is bad at random access

Maps

Complexity

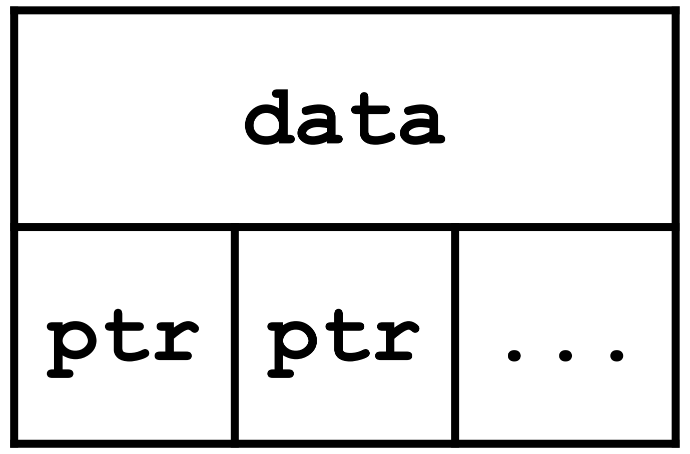
	lookup		insert		remove	
	average	worst	average	worst	average	worst
ArrayList	O(n)		O(1)	O(n)	O(1)	
Linked List	O(n)		O(1)		O(1)	
ArrayList (sorted)	O(log n)		O(n)		O(n)	
Linked List (sorted)	O(n)		O(1)		O(1)	

Maps

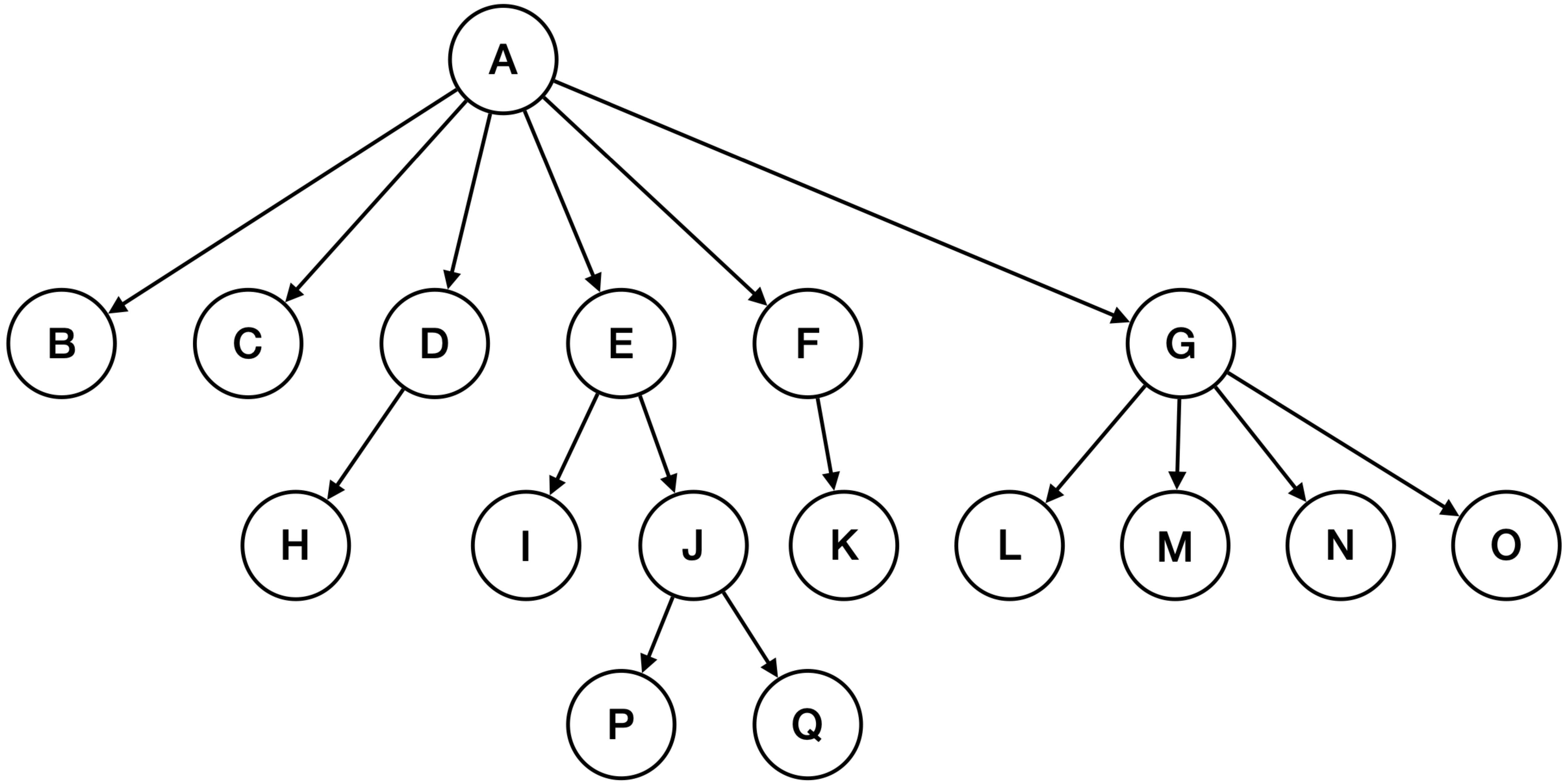
Can we have the benefits of both?

- Yes!
- New data structure: Binary Search Tree!

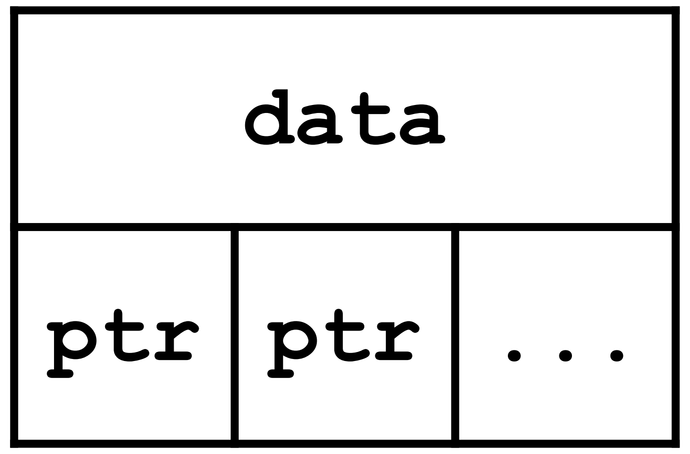
Trees



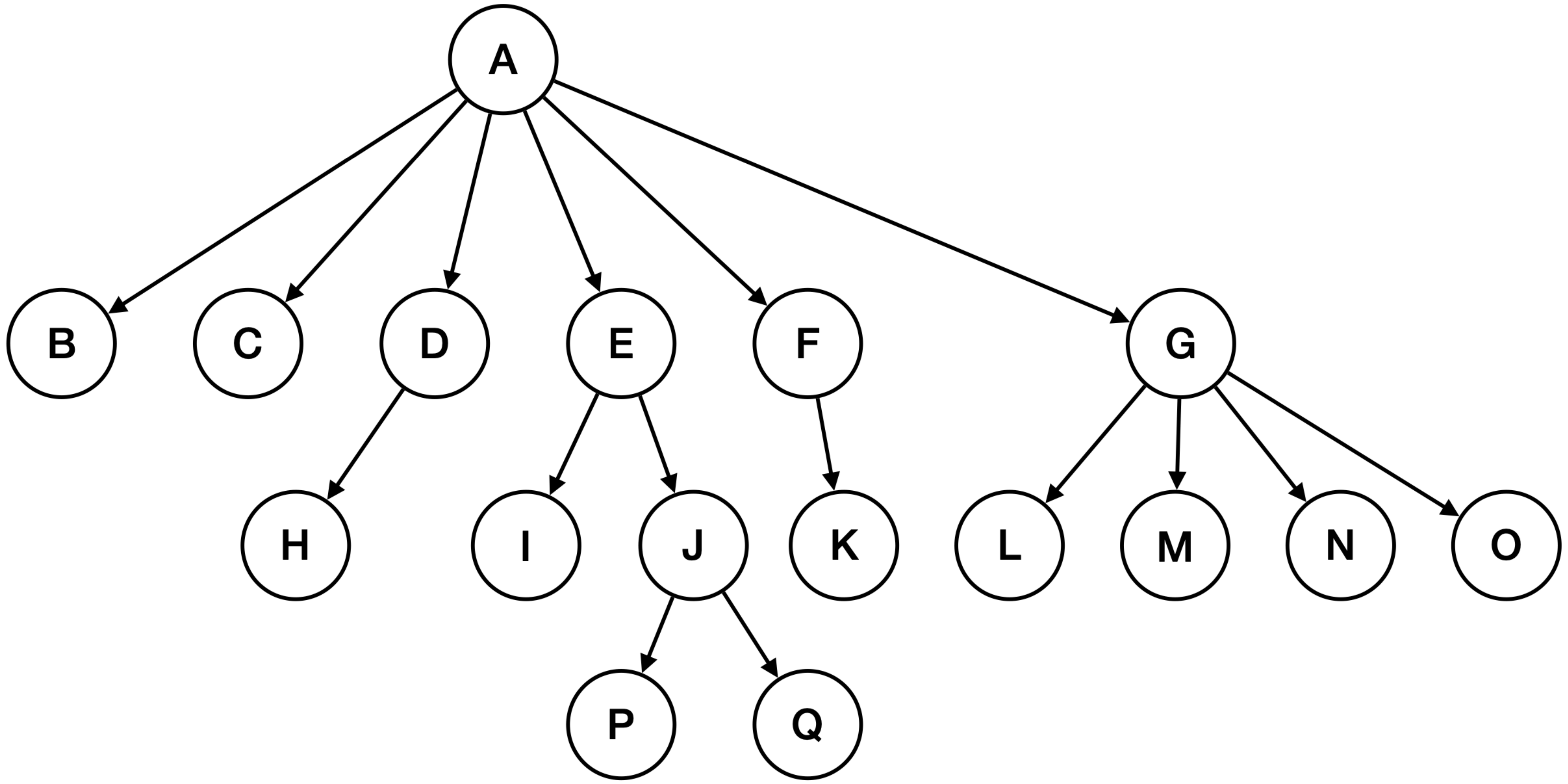
- Like a linked list, but have 1 *or more* `next` pointers.



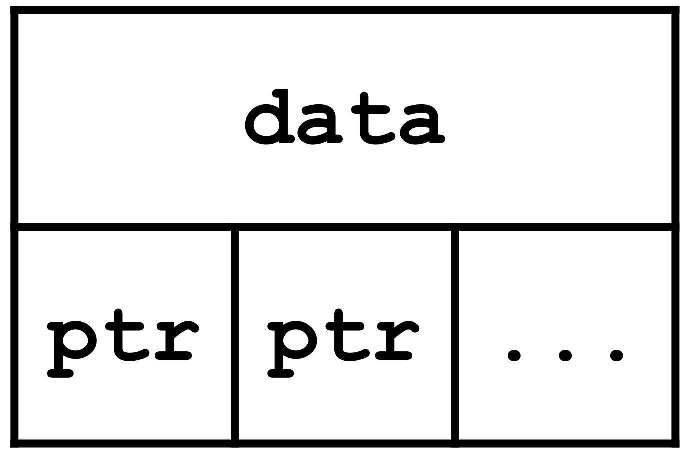
Trees



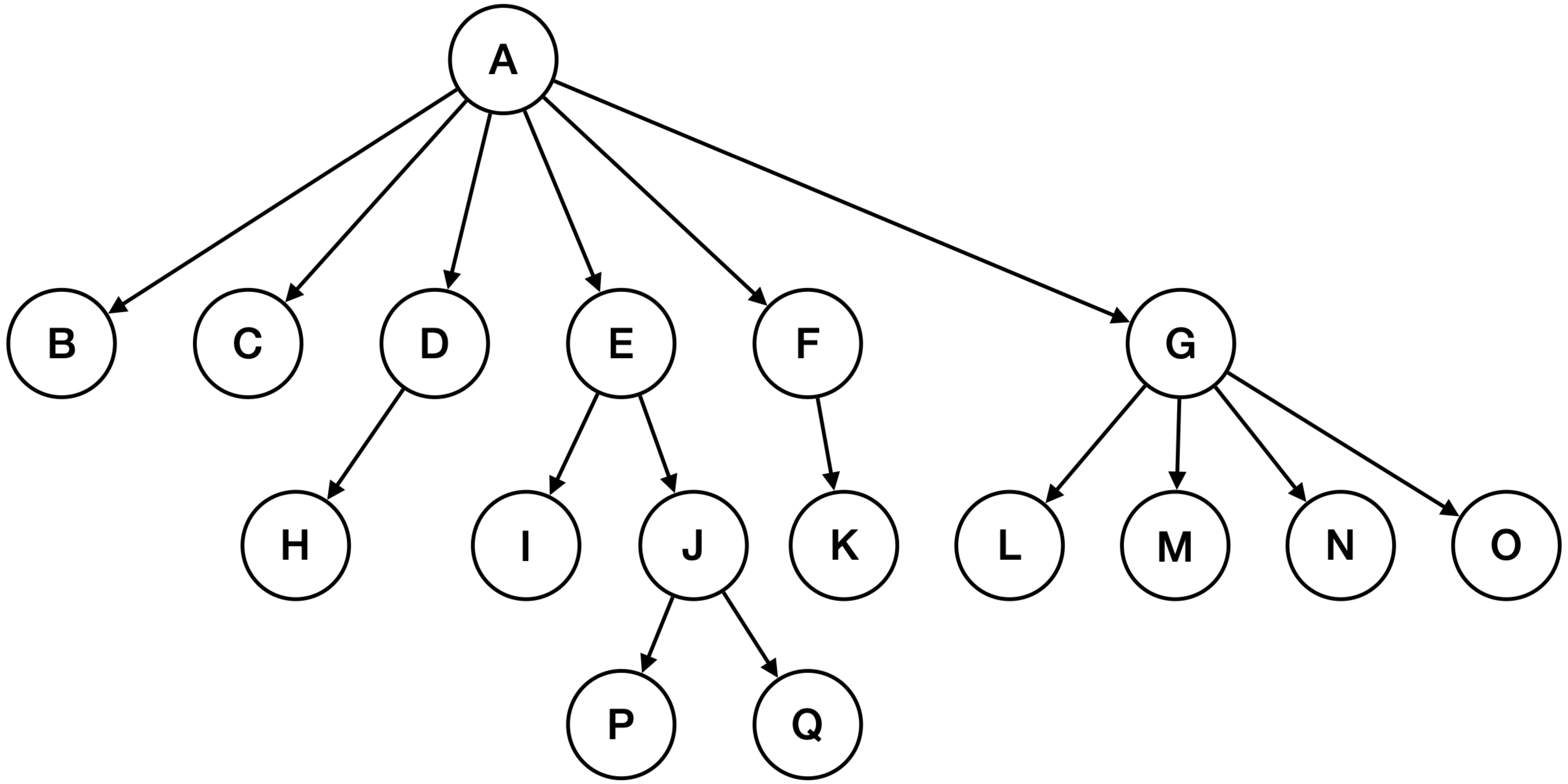
- A *tree* can be empty (NULL) or a node
  - where a node contains some data plus 1 *or more* pointers pointing to *trees*.



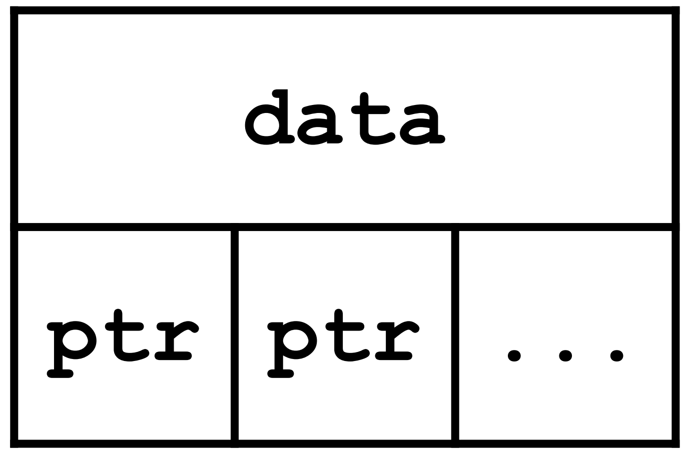
Trees



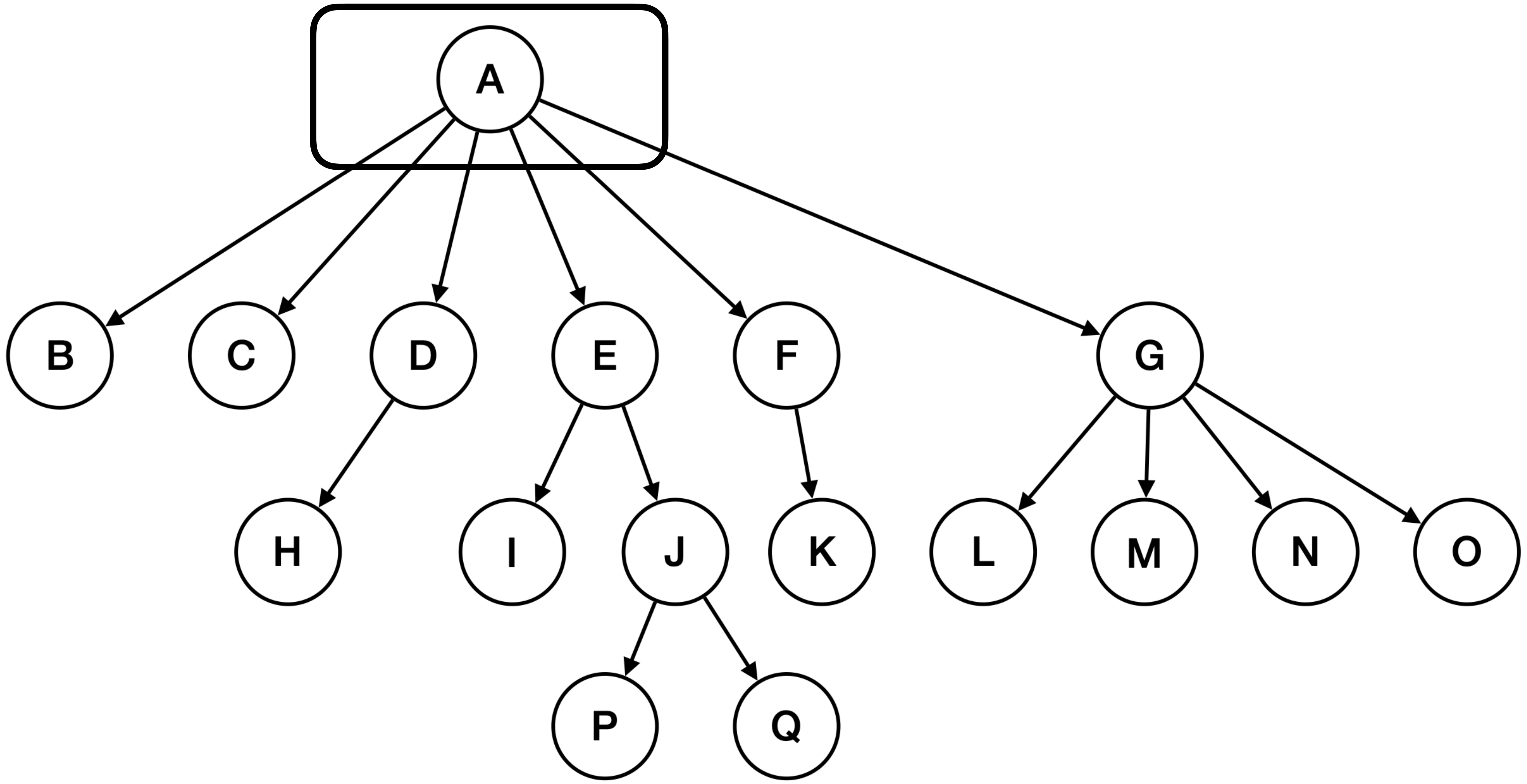
- A non-empty tree has a root



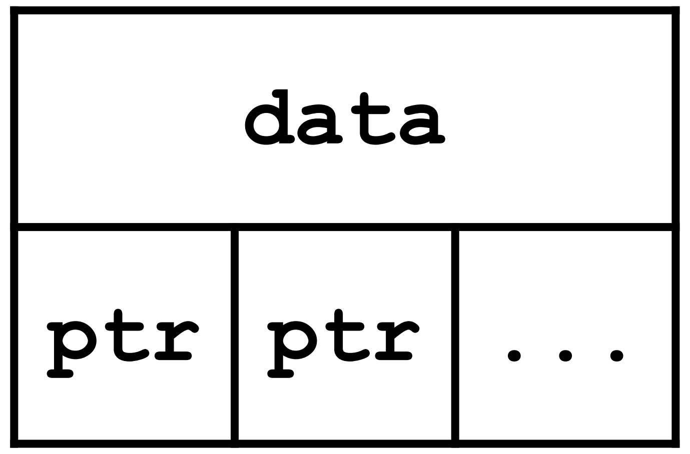
Trees



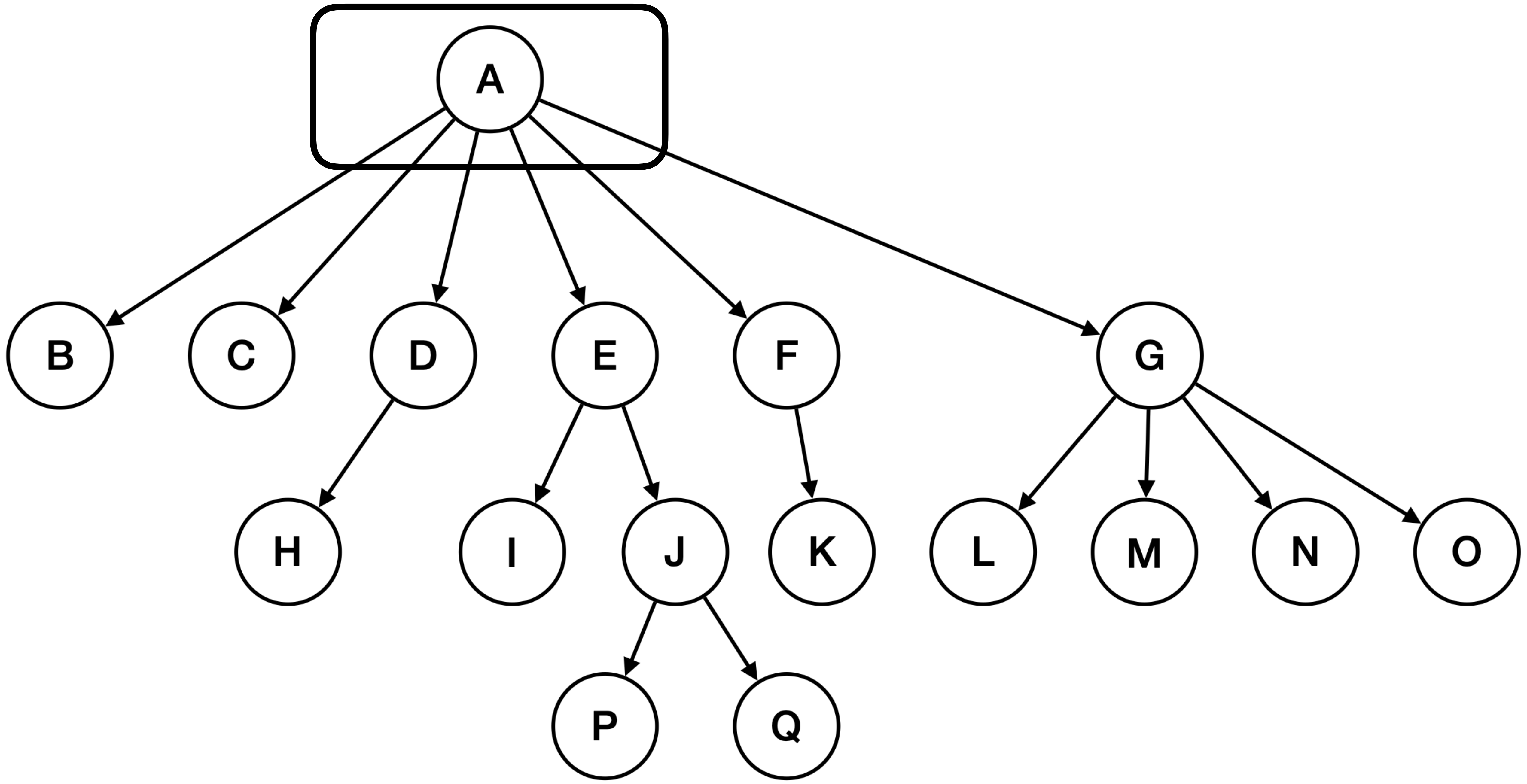
- A non-empty tree has a root



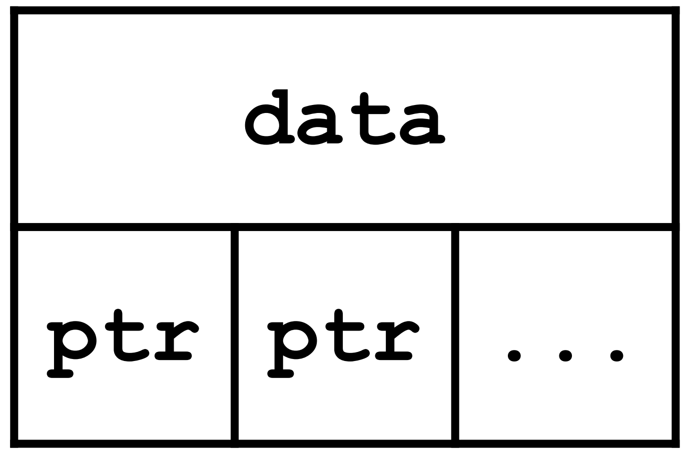
Trees



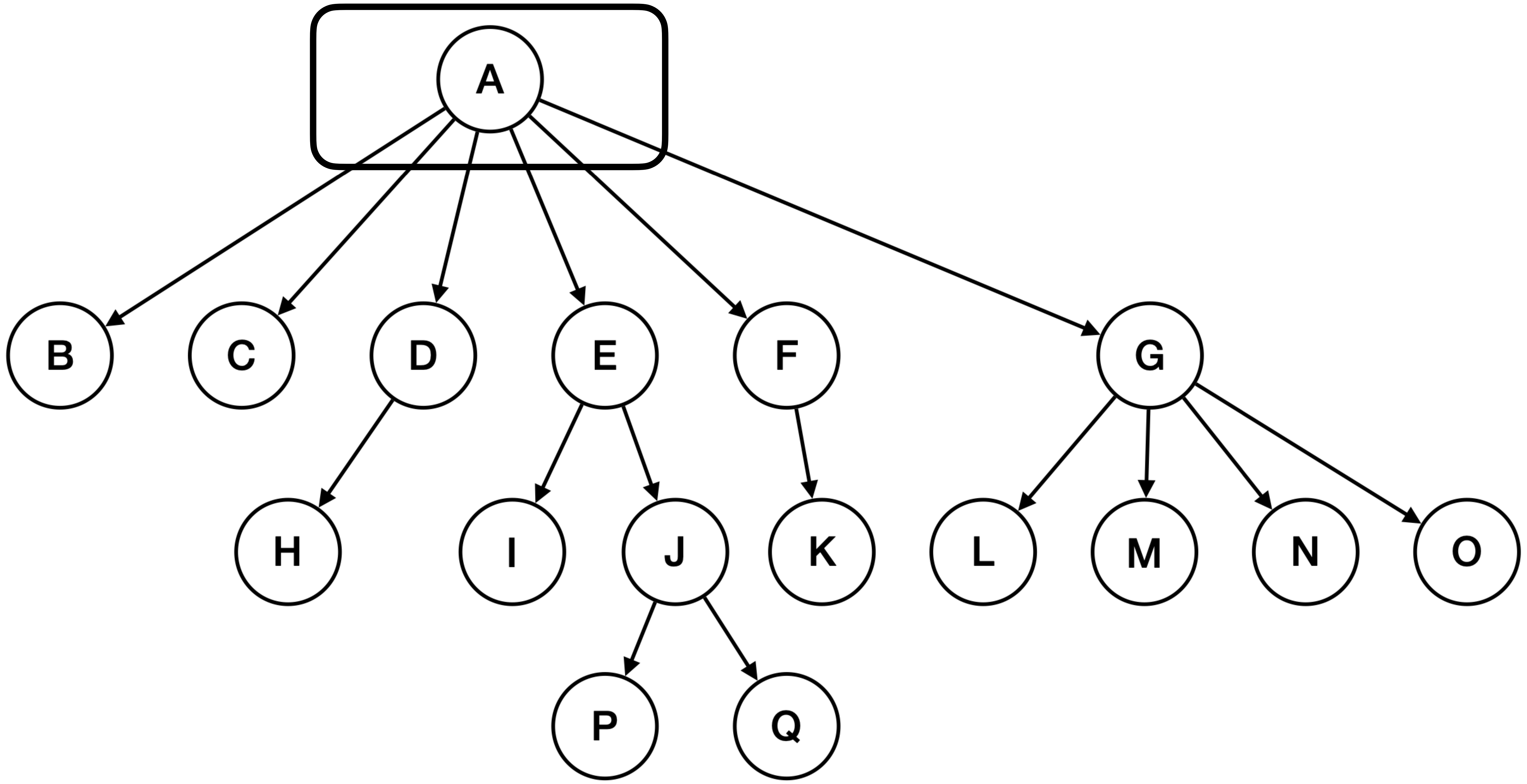
- A *parent* node points to multiple *child* nodes.



Trees



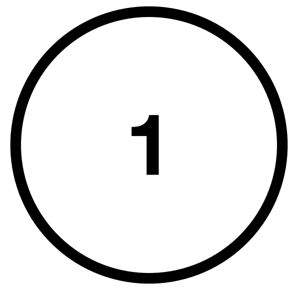
- A *parent* node points to multiple *child* nodes.
- Every node has exactly one parent, except the root which has no parents.



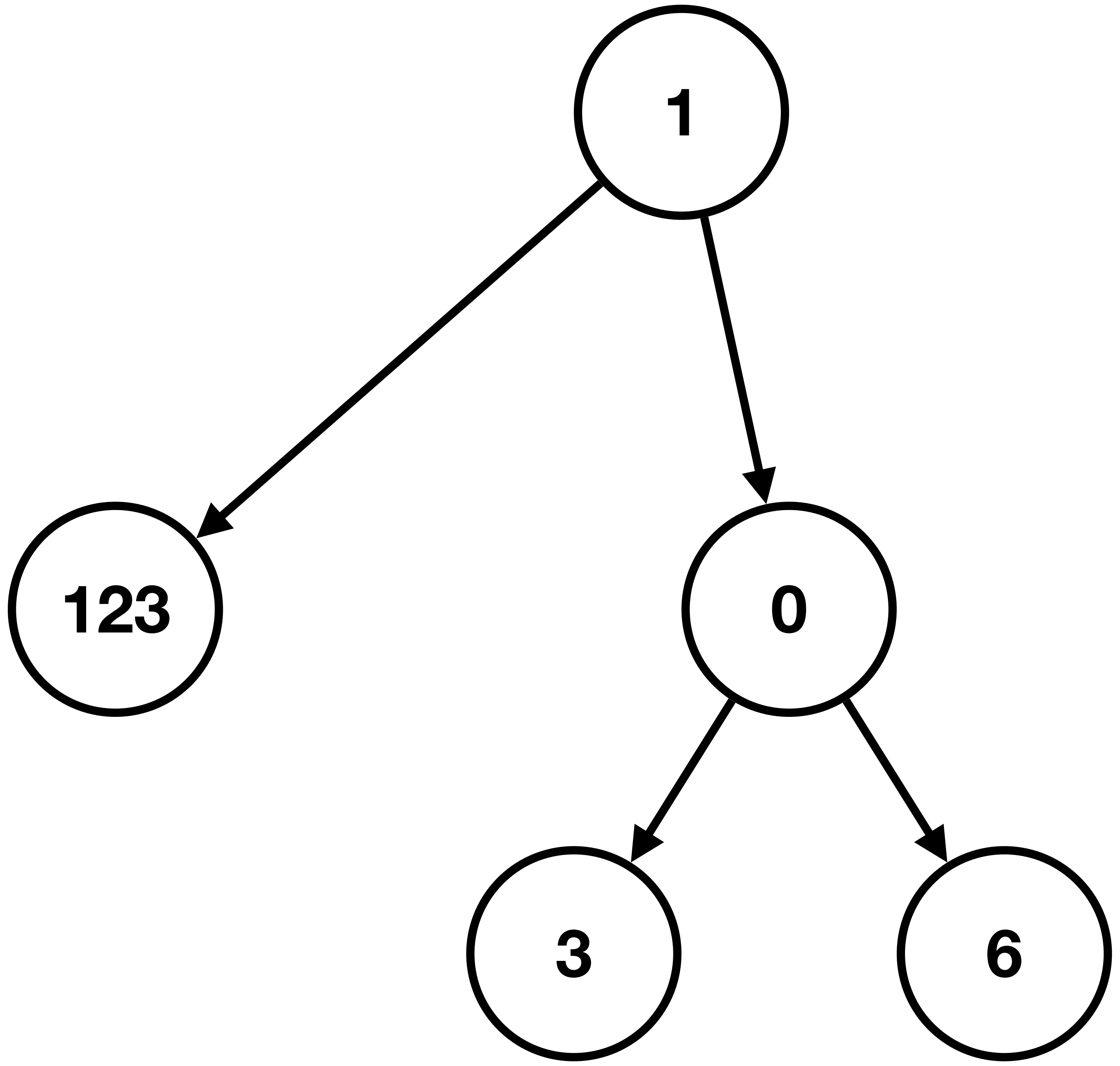
Trees

- A *tree* can be either
  - empty, *or*
  - a node contains some data plus 1 *or more* pointers pointing to *trees (subtrees)*.
- A *parent* node points to multiple *child* nodes.
- Every node has exactly one parent, except the root which has no parents.

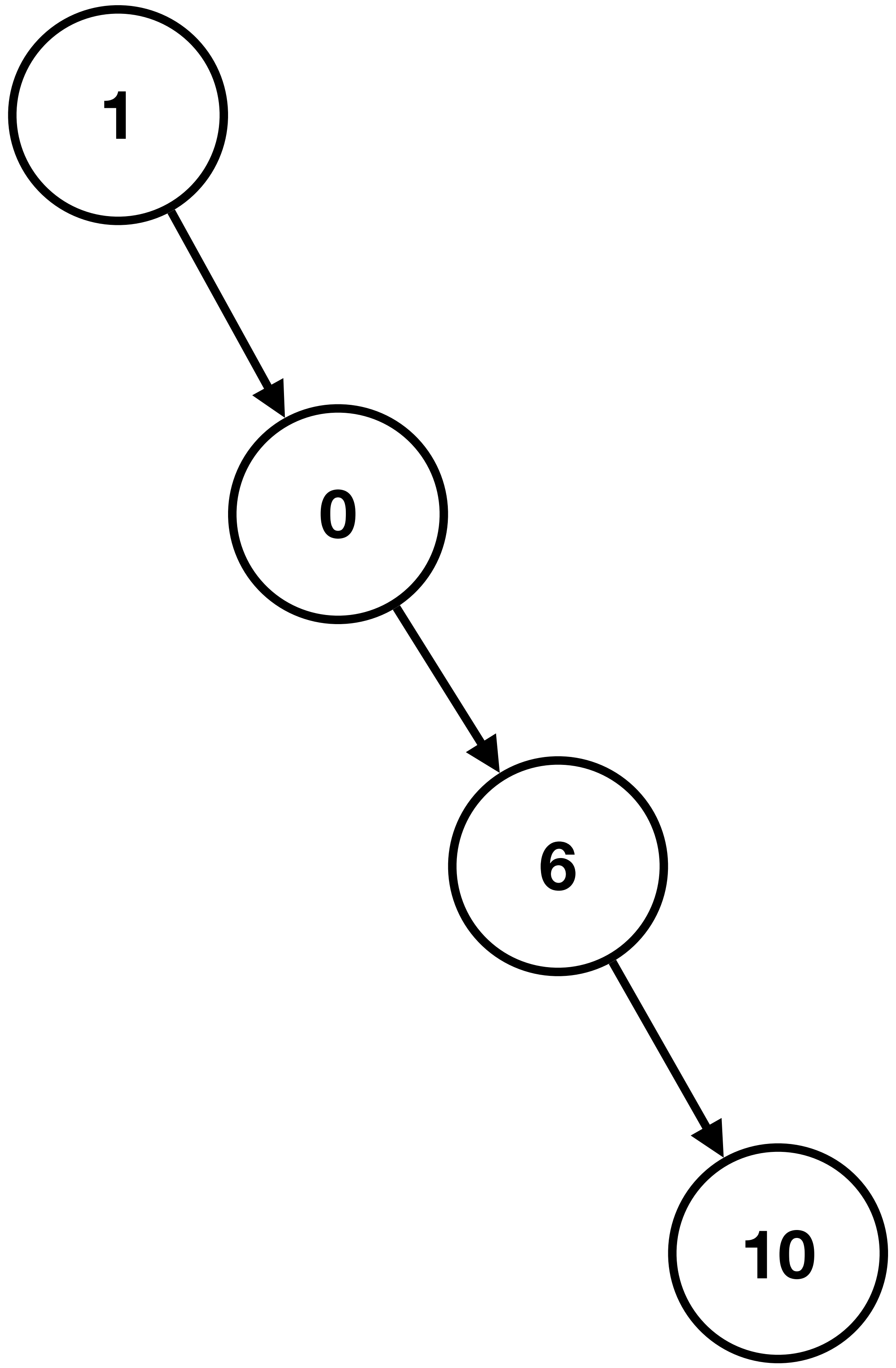
Is this a tree?



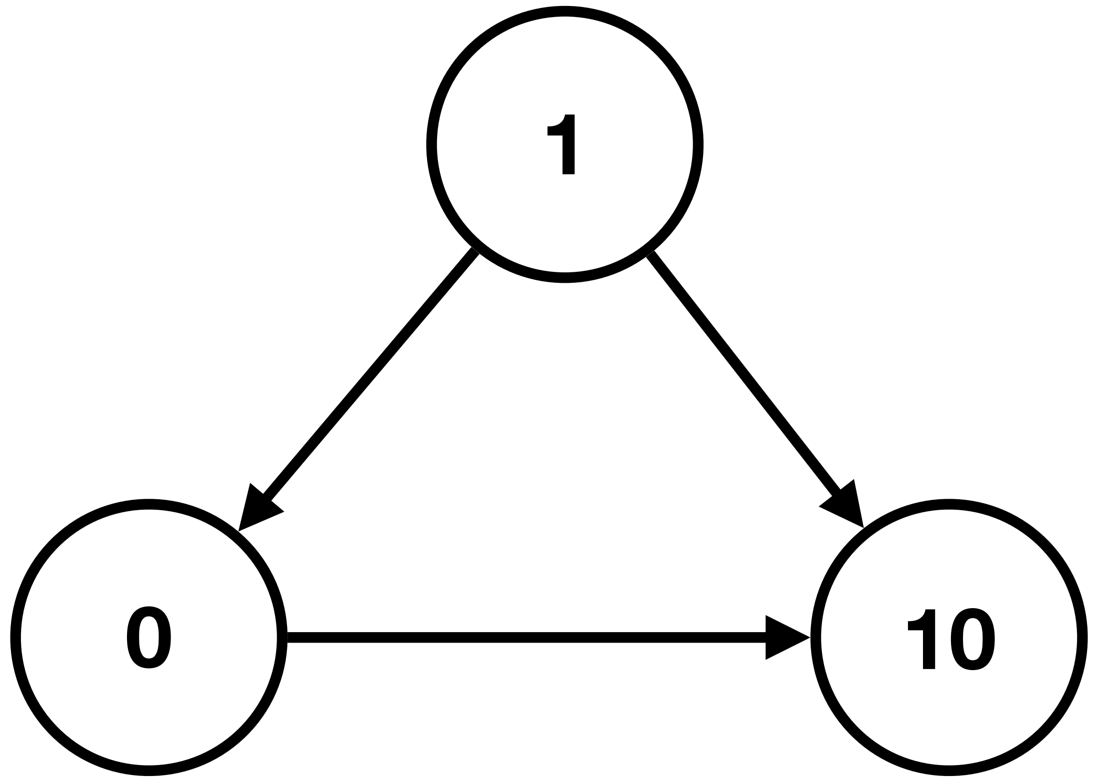
Is this a tree?



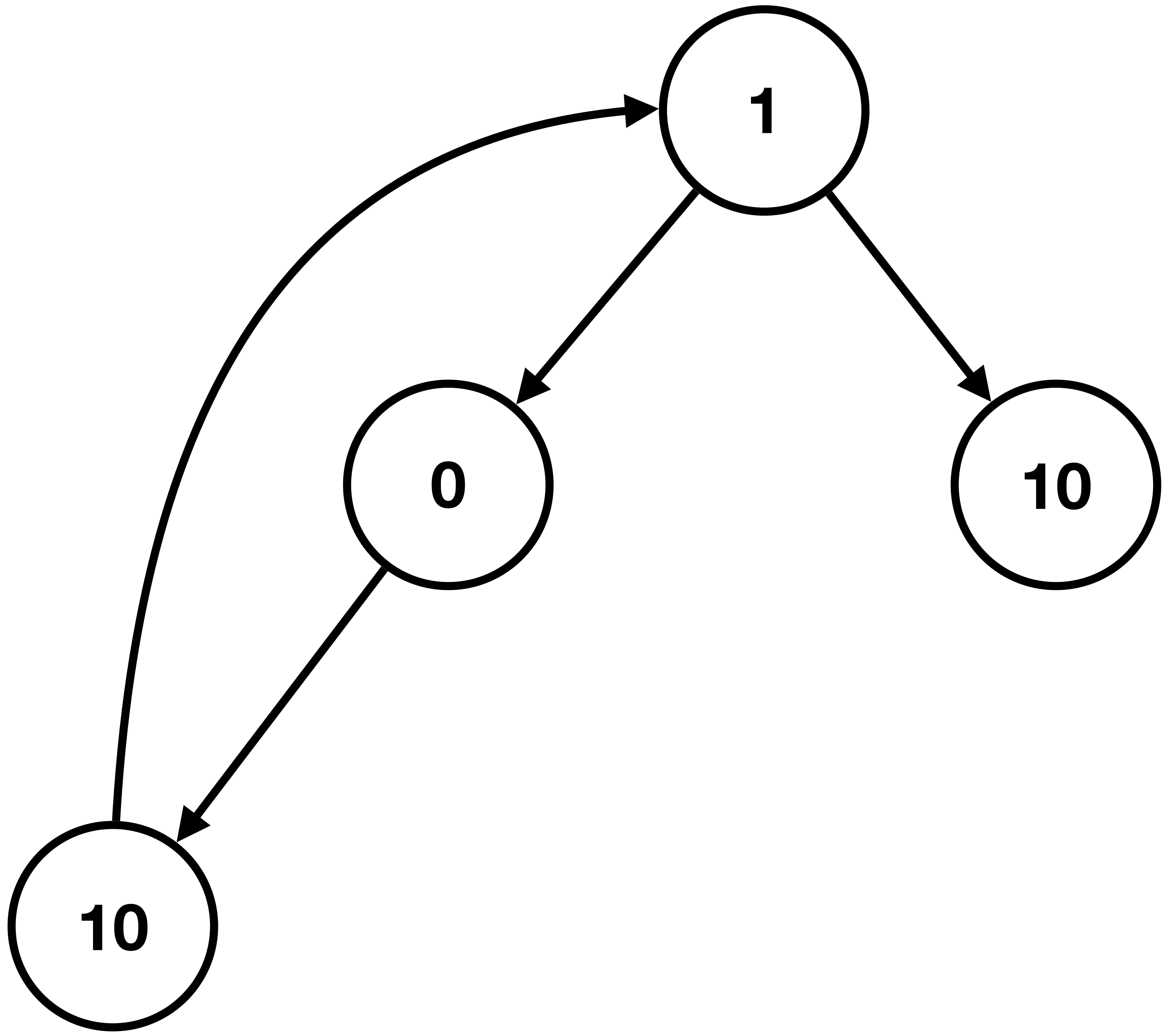
Is this a tree?



Is this a tree?



Is this a tree?



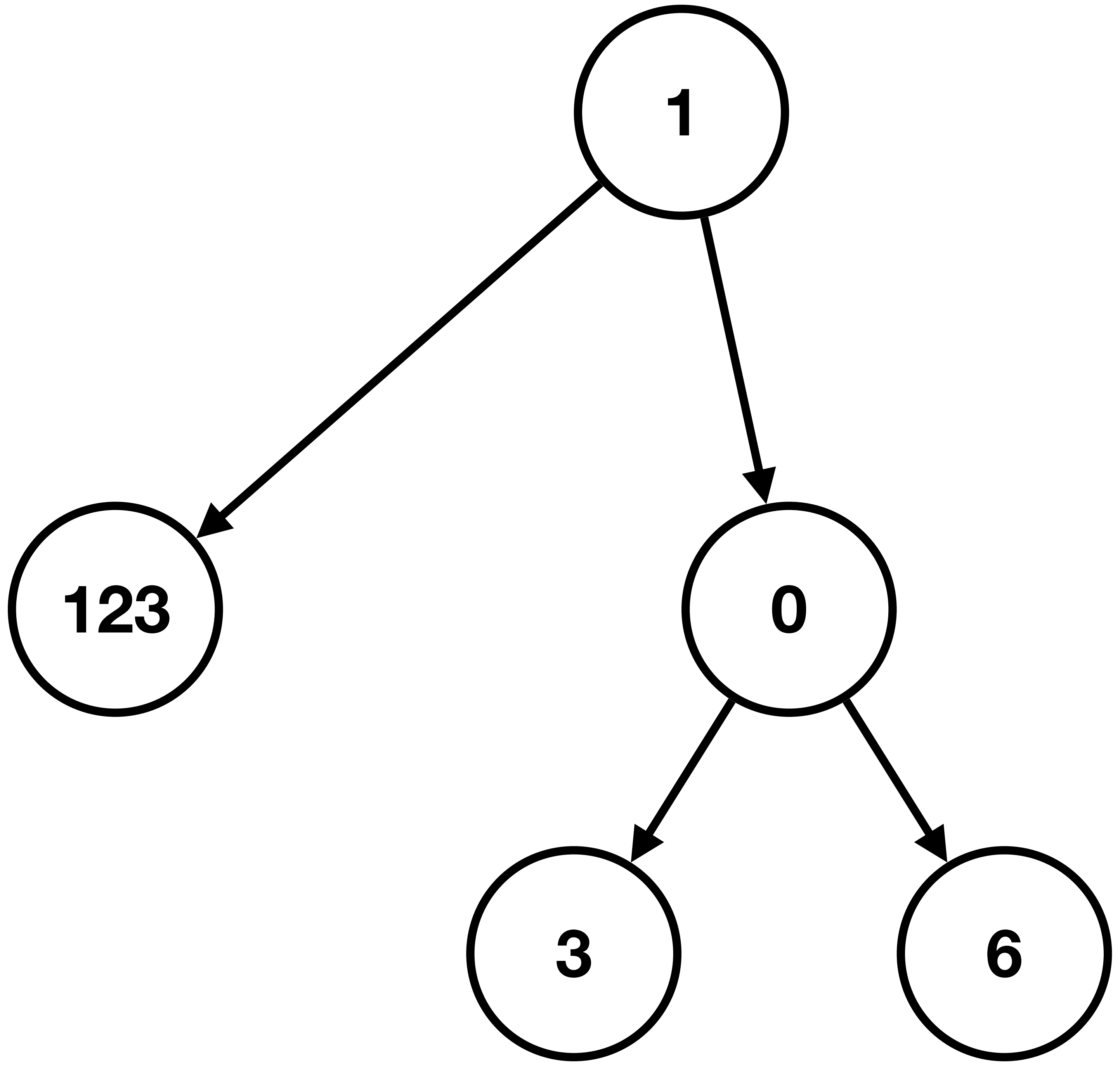
Is this a tree?

Is this a tree?

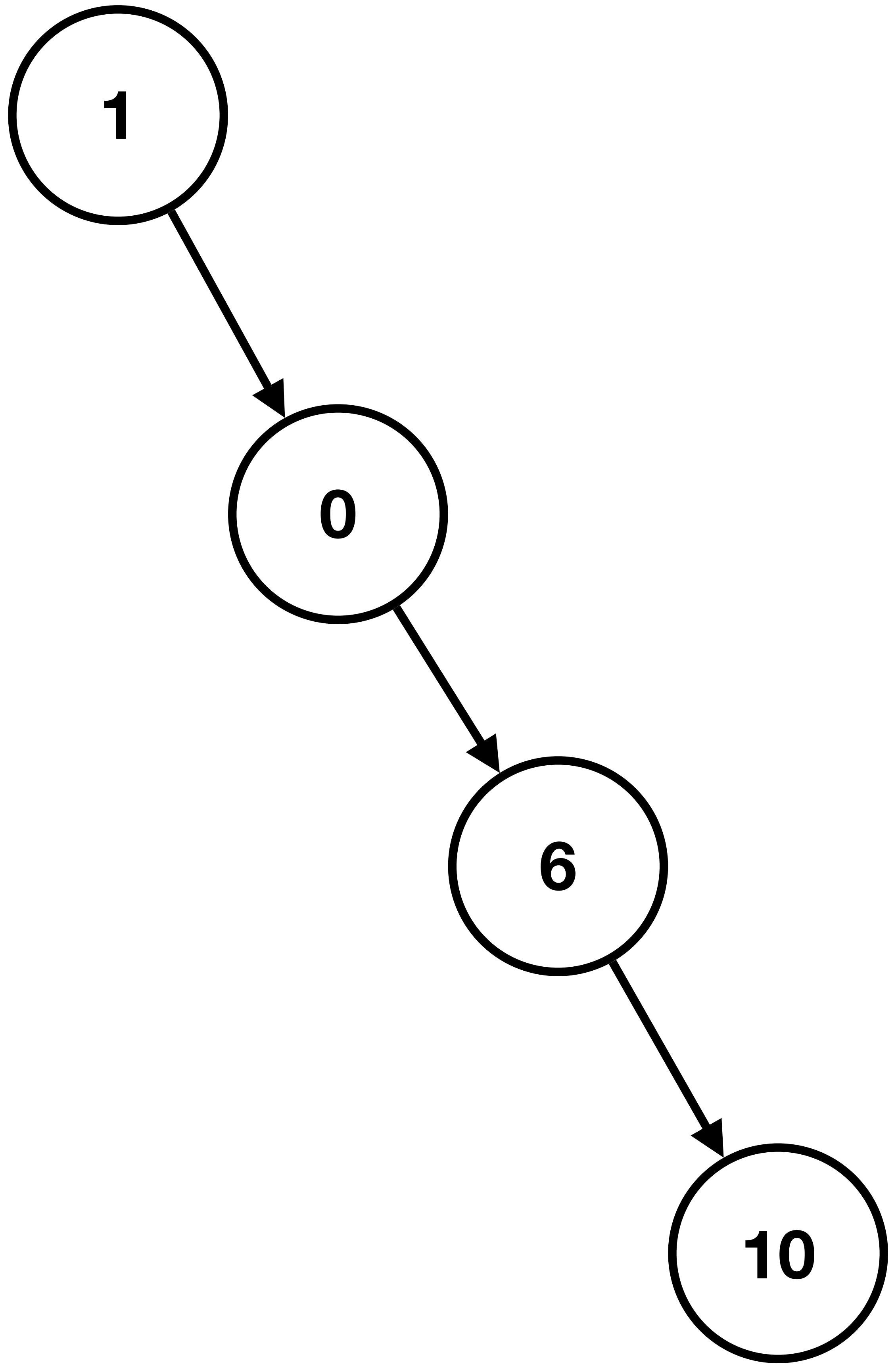
Binary Tree

- A *tree* can be either
  - empty, *or*
  - a node contains some data plus 2 pointers pointing to *trees (subtrees)*.
- A *parent* node points to multiple *child* nodes.
- Every node has exactly one parent, except the root which has no parents.

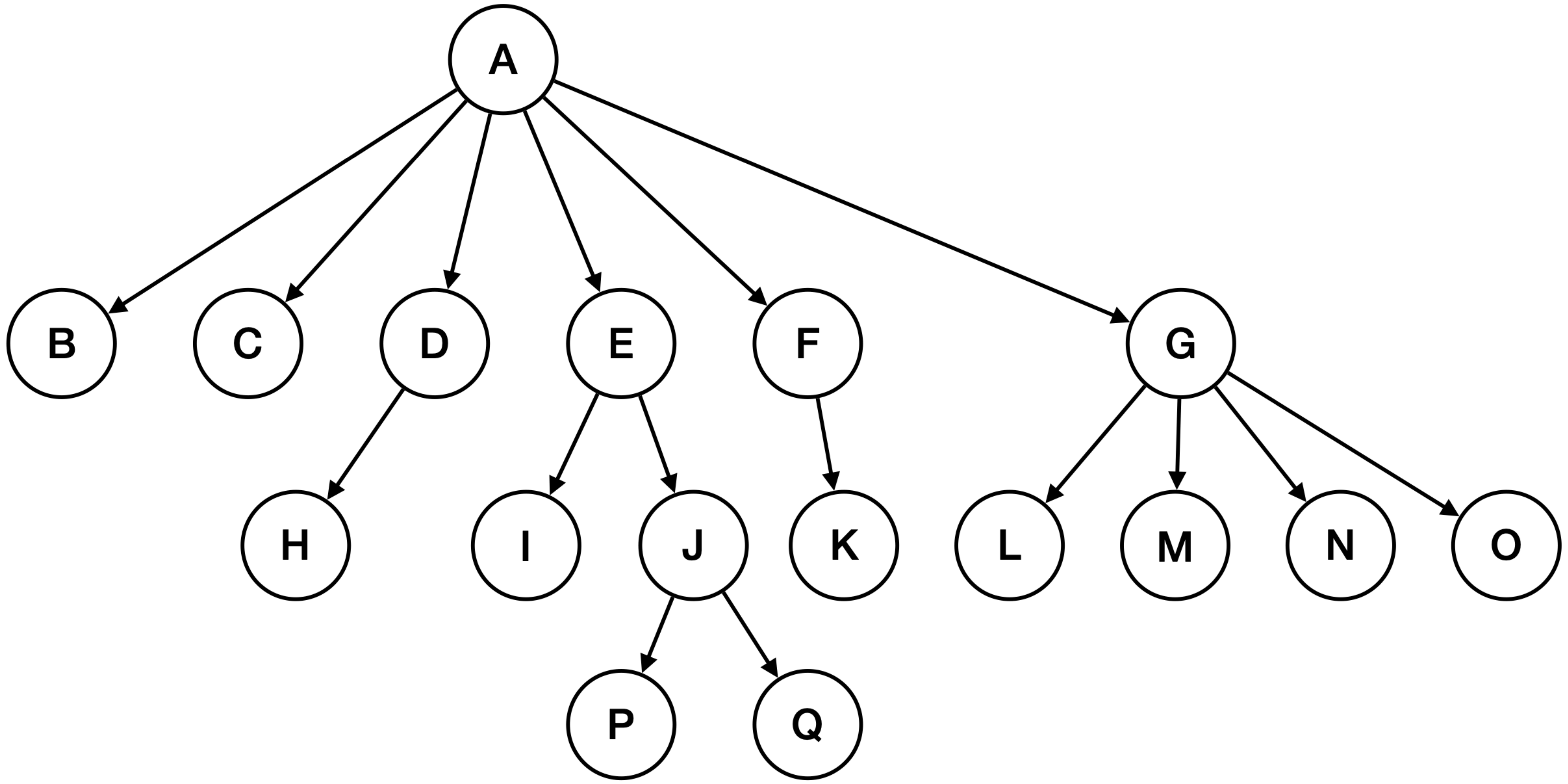
Is this a binary tree?



Is this a binary tree?



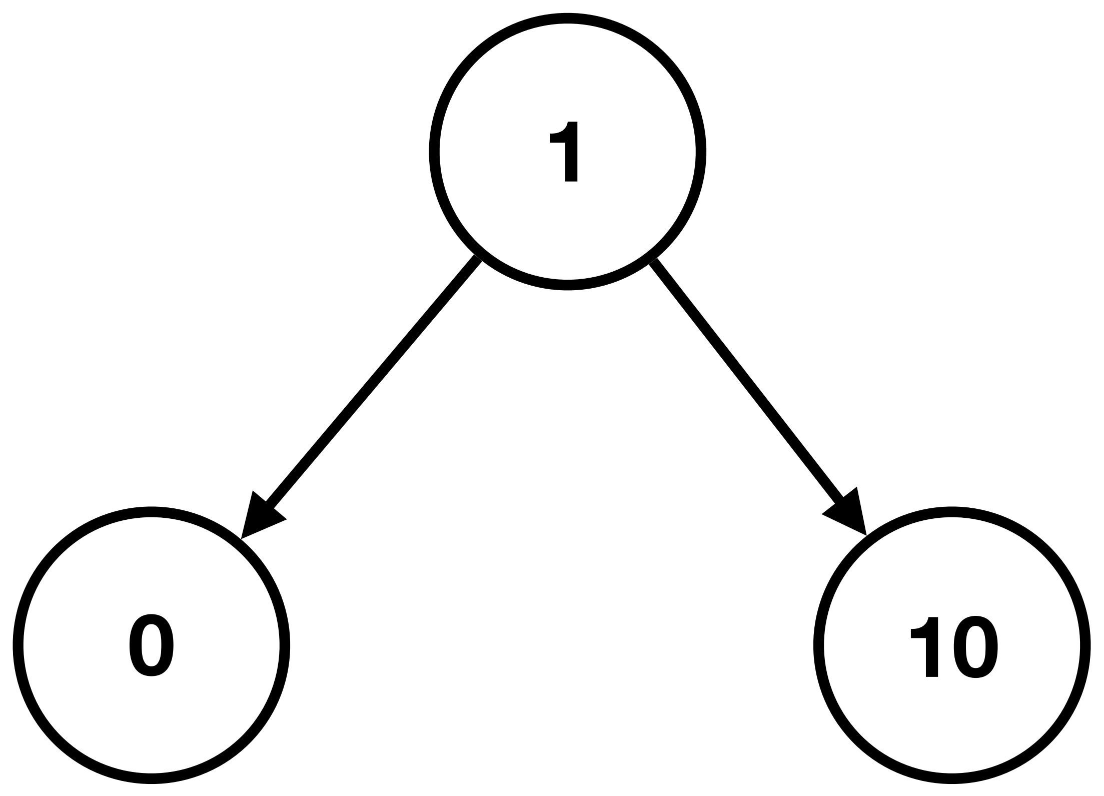
Is this a binary tree?



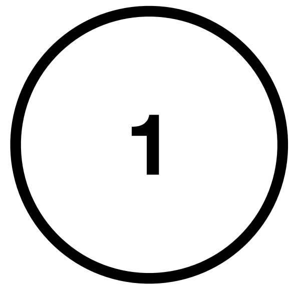
Binary Search Tree

- A binary search tree is a binary tree where
- For a given node *n* with key *k*,
  - All nodes with keys less than *k* are in *n*'s left subtree.
  - All nodes with keys greater than *k* are in *n*'s right subtree.

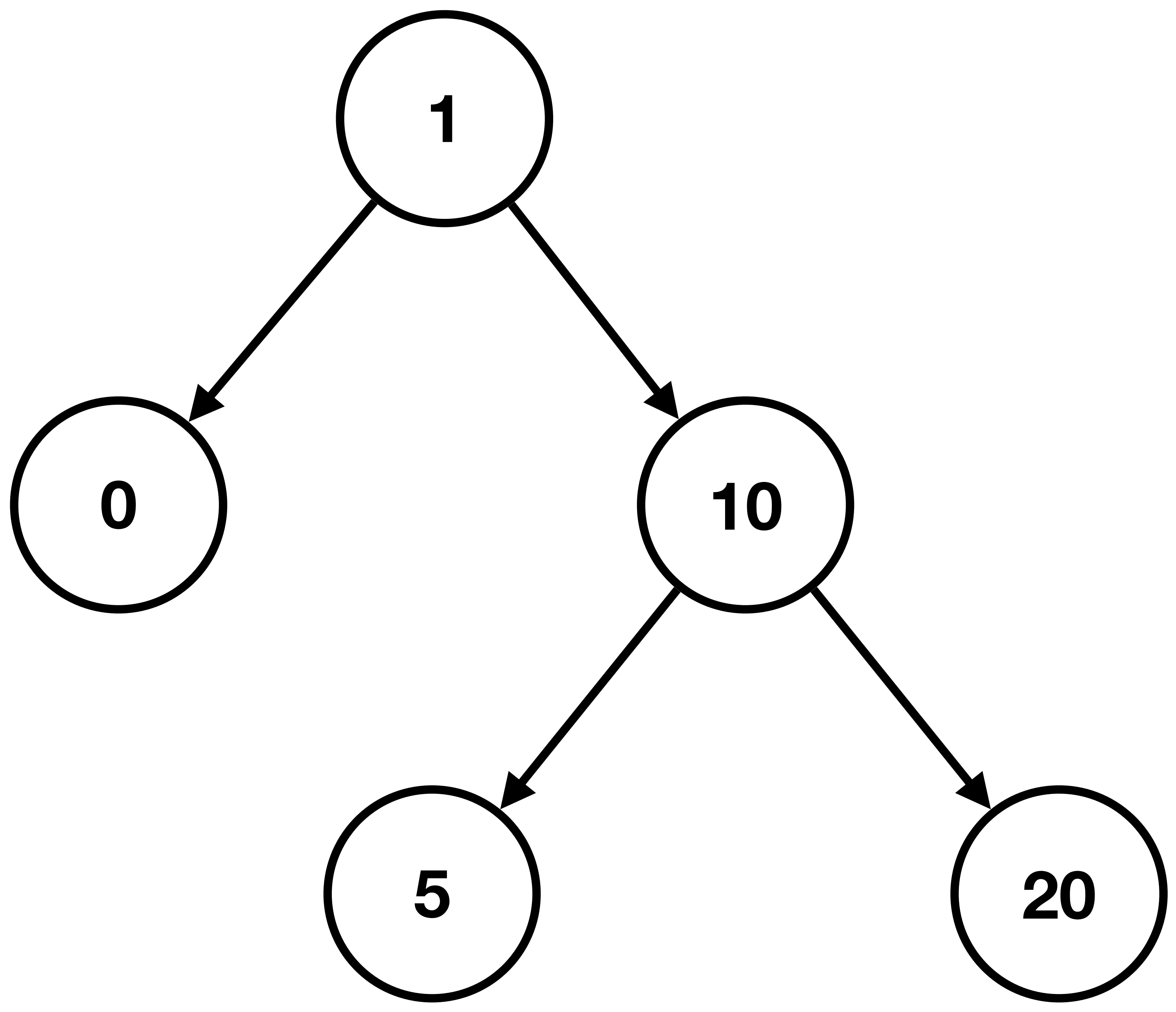
Is this a BST?



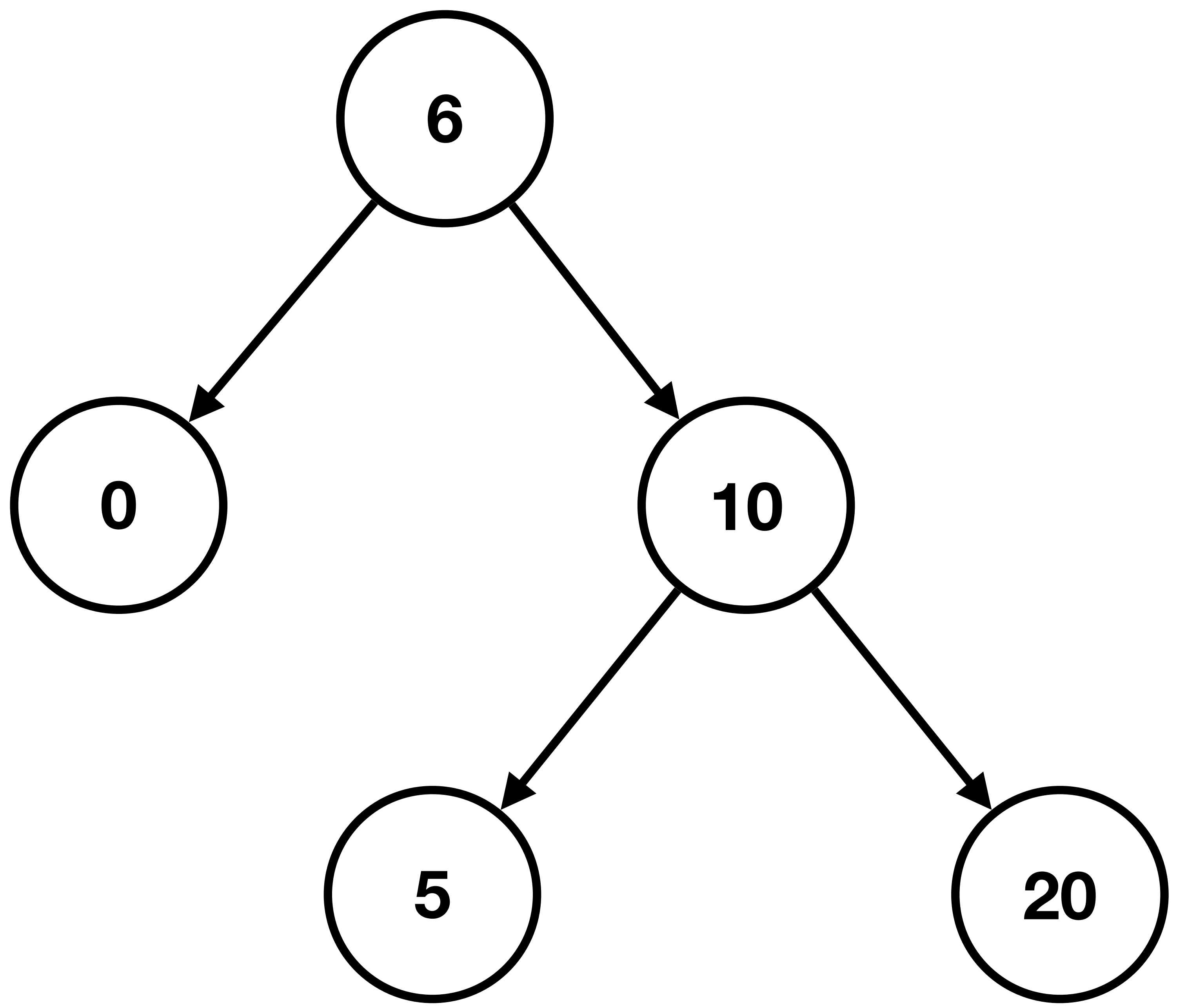
Is this a BST?



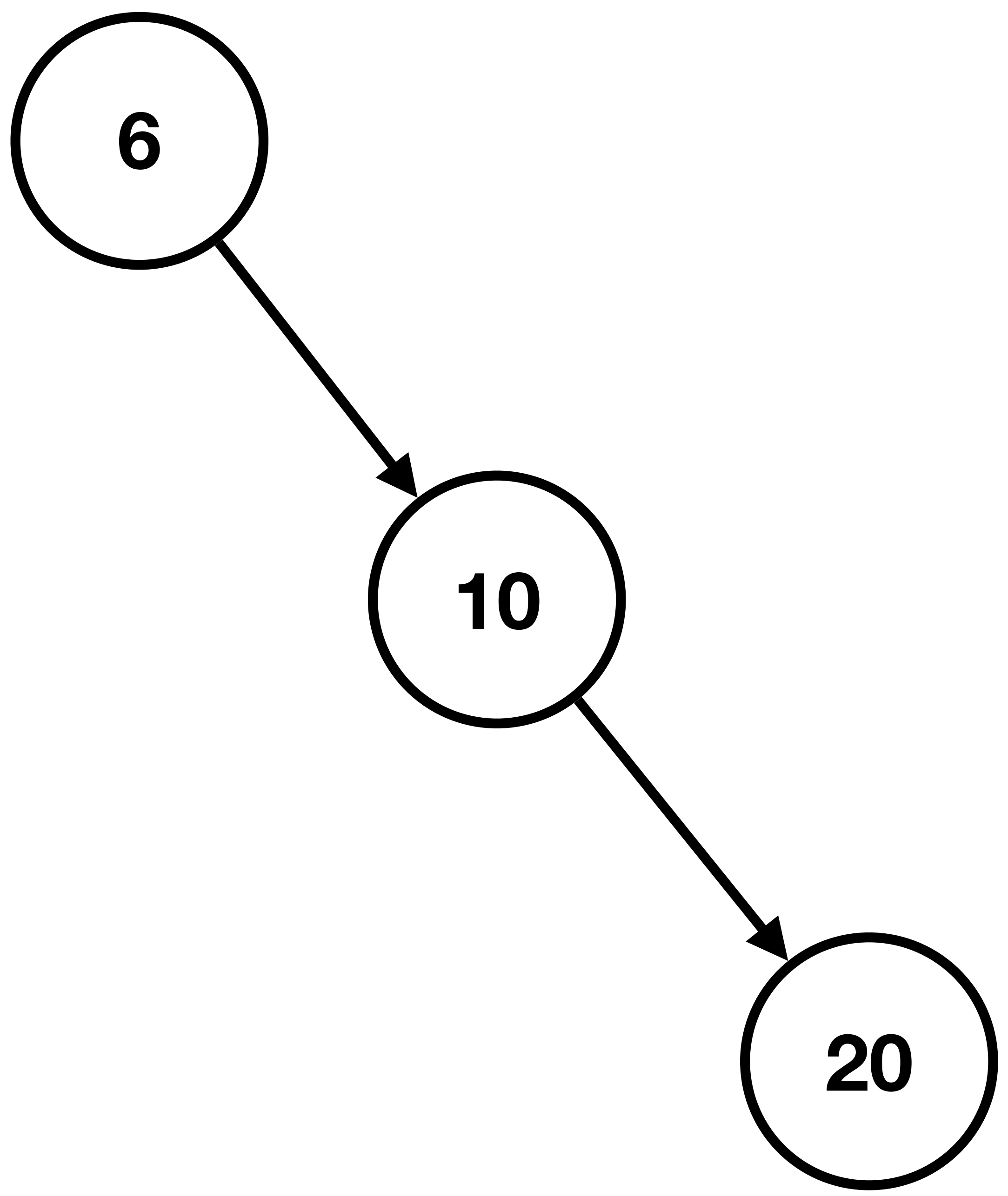
Is this a BST?



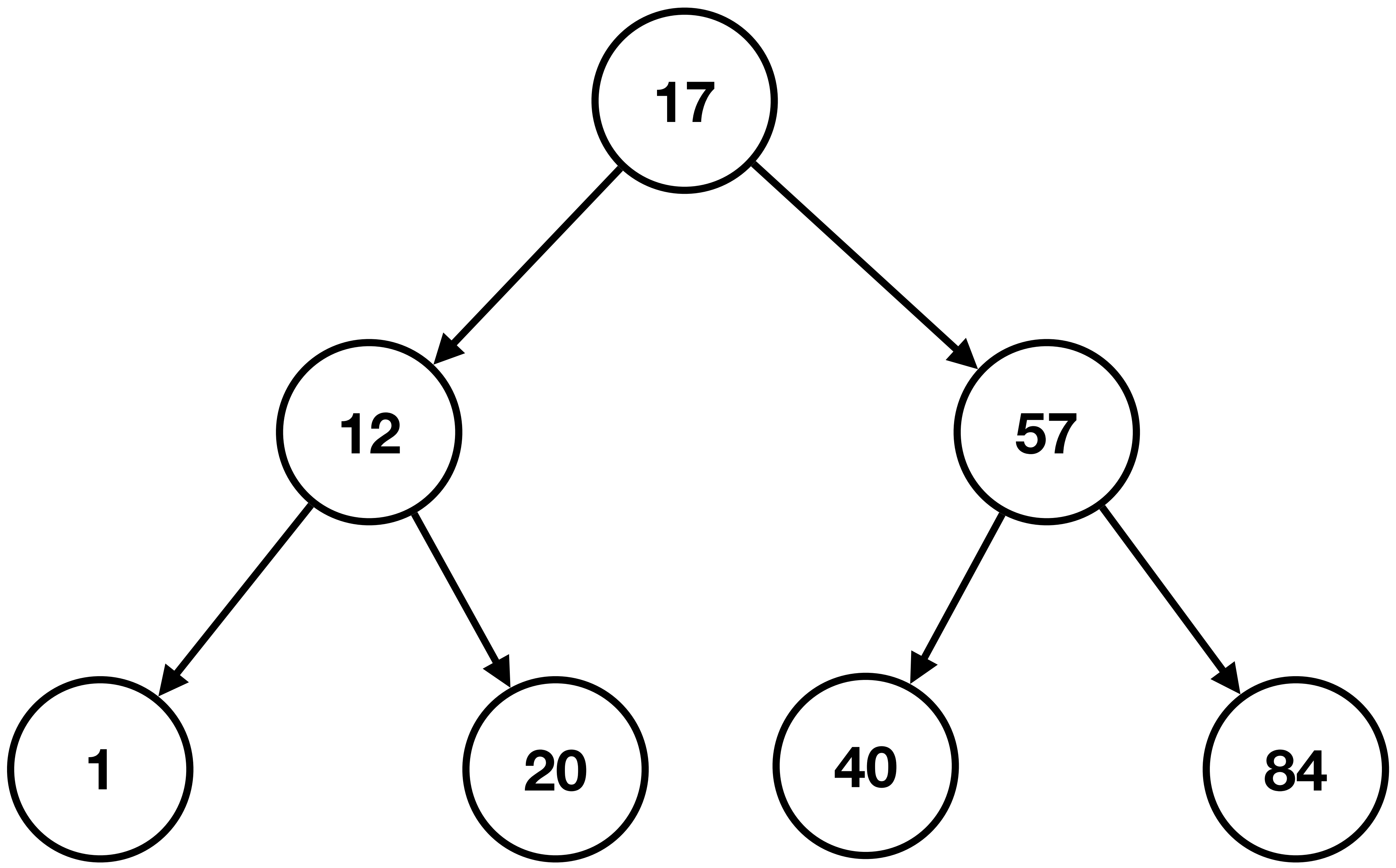
Is this a BST?



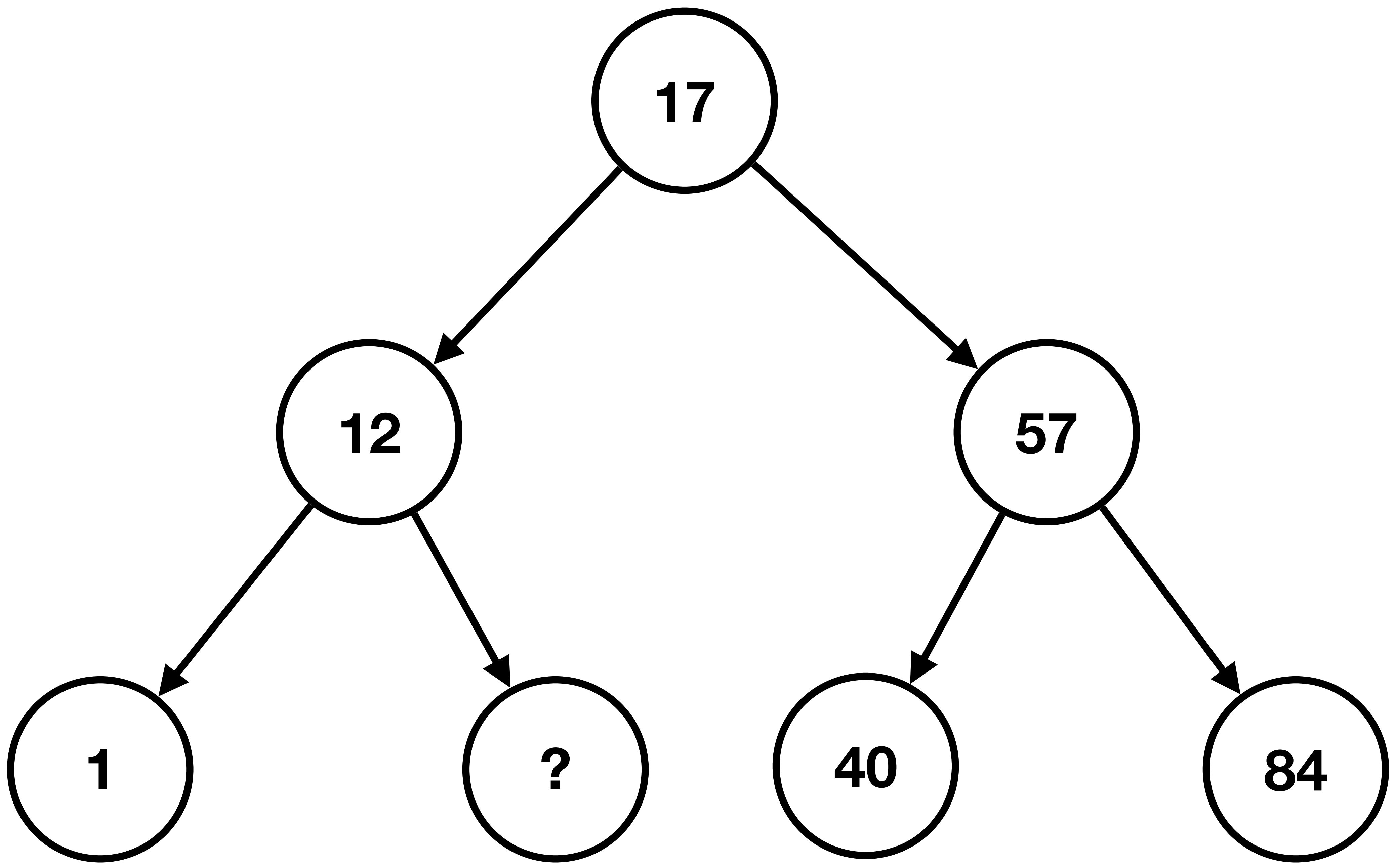
Is this a BST?



Is this a BST?



Is this a BST?



Binary Search Tree

- A binary search tree is a binary tree where
- For a given node *n* with key *k*,
    - All nodes with keys less than *k* are in *n*'s left subtree.
    - All nodes with keys greater than *k* are in *n*'s right subtree.