Maps & BST

CS143: lecture 13

Konstantinos Ameranis, July 21

Lists

Recap

Lists

Recap

- Lists: [🍏, 🍎, 🍐, 🍊, 🍋]

Lists

Recap

- Lists: [🍏, 🍎, 🍐, 🍊, 🍋]
- It is an ordered collection of elements:

Lists

Recap

- Lists: [🍏, 🍎, 🍐, 🍊, 🍋]
- It is an ordered collection of elements:
  - Ordered: 1st, 2nd, 3rd, ...

Lists

Recap

- Lists: [🍏, 🍎, 🍐, 🍊, 🍋]
- It is an ordered collection of elements:
  - Ordered: 1st, 2nd, 3rd, ...
  - Elements can be homogeneous or heterogeneous.

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*

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

Maps

- What if we want to build a *mapping* between one element to another element?

Maps

- What if we want to build a *mapping* between one element to another element?
- { 🍏 : ⚽, 🍎 : 🏀, 🍐 : 🏈, 🍊 : ⚾, 🍋 : 🥎 }

Maps

- What if we want to build a *mapping* between one element to another element?
- { 🍏 : ⚽, 🍎 : 🏀, 🍐 : 🏈, 🍊 : ⚾, 🍋 : 🥎 }
- Maps!

Maps

- What if we want to build a *mapping* between one element to another element?
- { 🍏 : ⚽, 🍎 : 🏀, 🍐 : 🏈, 🍊 : ⚾, 🍋 : 🥎 }
- Maps!
    - aka dictionaries, associative array...

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

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

Maps

Operations

- `insert(k, v)`

Maps

Operations

- `insert(k, v)`
- `remove(k)`

Maps

Operations

- `insert(k, v)`
- `remove(k)`
- `lookup(k)`

Maps

Operations

- `insert(k, v)`
- `remove(k)`
- `lookup(k)`
- `size`

Maps

Operations

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

Maps

Can we use lists?

Maps

Can we use lists?

- Yes!

Maps

Can we use lists?

- Yes!
- Each element of the list can be a pair (key, value)

Maps

Can we use lists?

- Yes!
- Each element of the list can be a pair (key, value)
- `insert(k, v):`

Maps

Can we use lists?

- Yes!
- Each element of the list can be a pair (key, value)
- `insert(k, v)`:
    - `append((k, v))`

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):`

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`

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)`:

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

Maps

Complexity

	lookup		insert		remove	
	average	worst	average	worst	average	worst
ArrayList						

Maps

Complexity

	lookup		insert		remove	
	average	worst	average	worst	average	worst
ArrayList	O(n)					

Maps

Complexity

	lookup		insert		remove	
	average	worst	average	worst	average	worst
ArrayList	O(n)	O(n)				

Maps

Complexity

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

Maps

Complexity

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

Maps

Complexity

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

Maps

Complexity

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

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						

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)					

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)				

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)			

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)		

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(n)	

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(n)	O(n)

Maps

Can we do better with lists?

Maps

Can we do better with lists?

- What if we can sort the keys?

Maps

Can we do better with lists?

- What if we can sort the keys?
- Lookup is faster

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

Maps

Can we do better with lists?

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

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 $O(\log_2 n)$

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

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

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(n)	
Linked List	O(n)		O(1)		O(n)	
ArrayList (sorted)						

Maps

Complexity

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

Maps

Complexity

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

Maps

Complexity

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

Maps

Complexity

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

Maps

Complexity

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

Maps

Complexity

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

Maps

Complexity

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

Maps

Can we have the benefits of both?

Maps

Can we have the benefits of both?

- Yes!

Maps

Can we have the benefits of both?

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

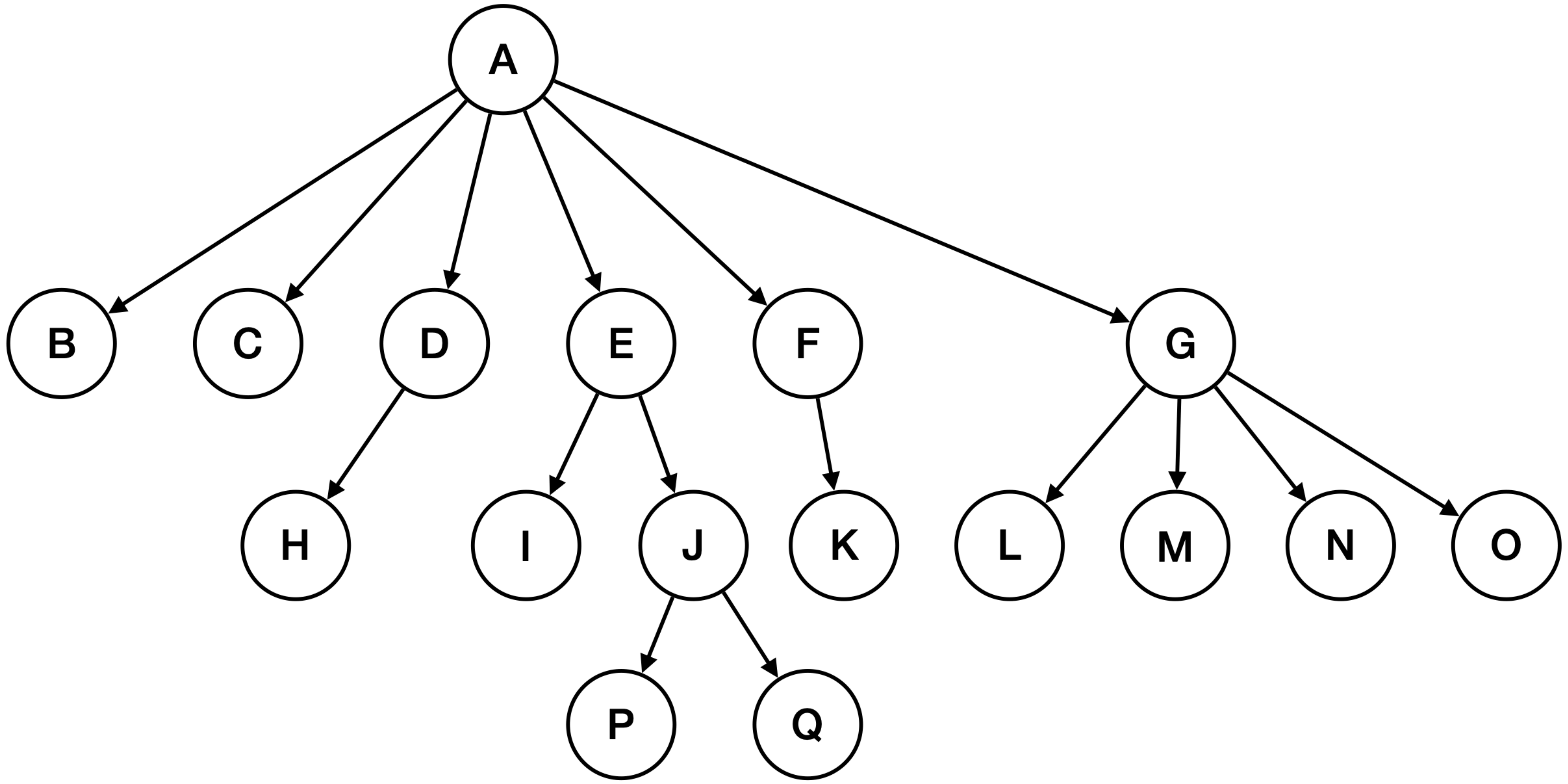
Trees

Trees

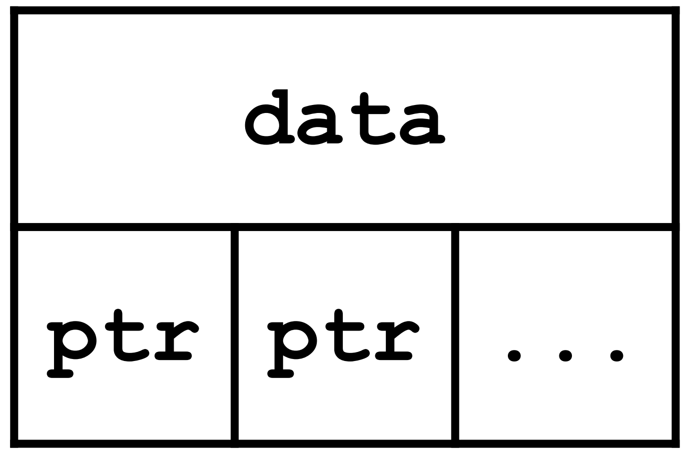
- Like a linked list, but have 1 *or more* `next` pointers.

Trees

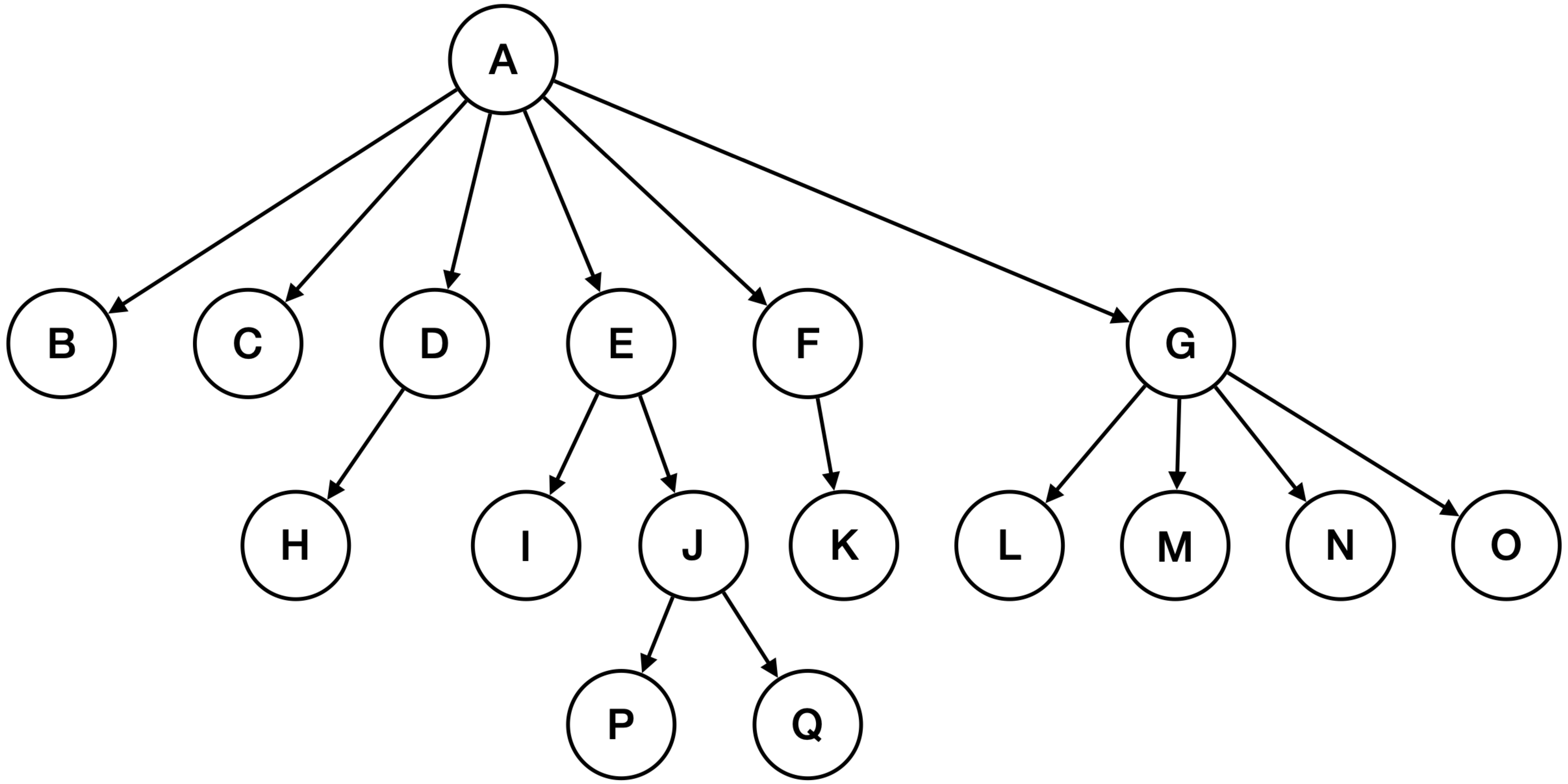
- Like a linked list, but have 1 *or more* `next` pointers.



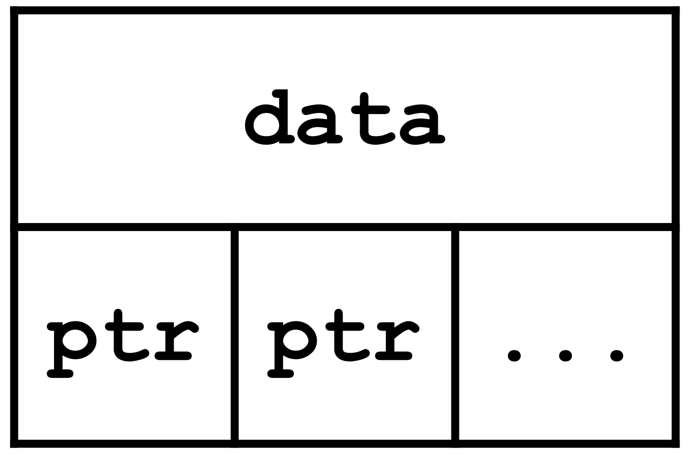
Trees



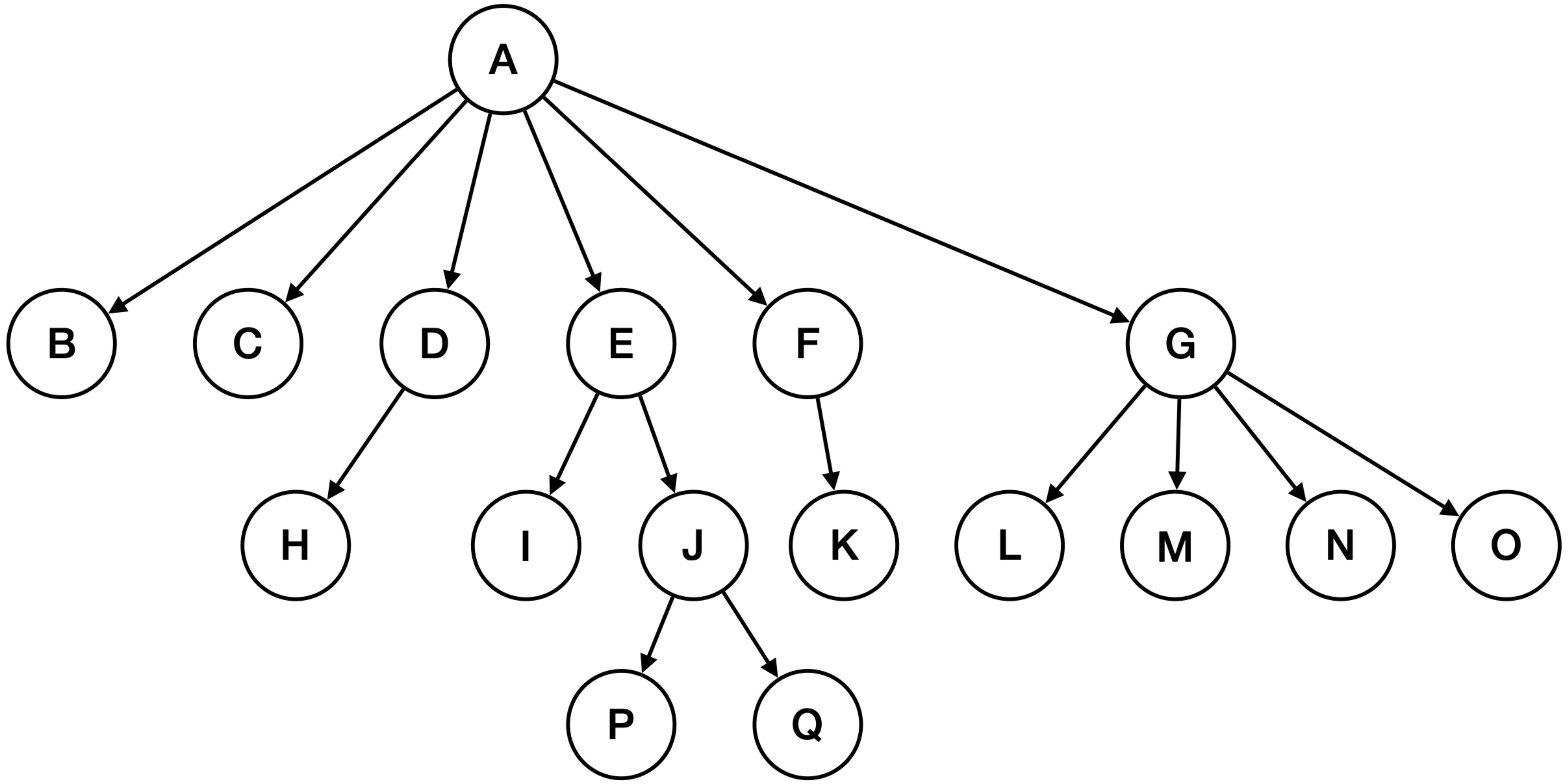
- Like a linked list, but have 1 *or more* `next` pointers.



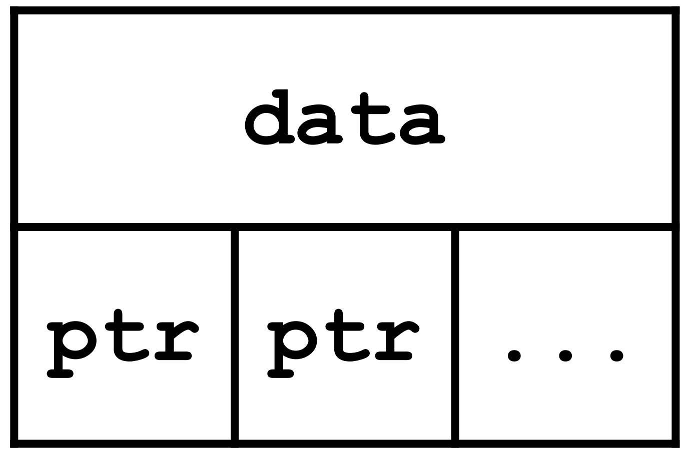
Trees



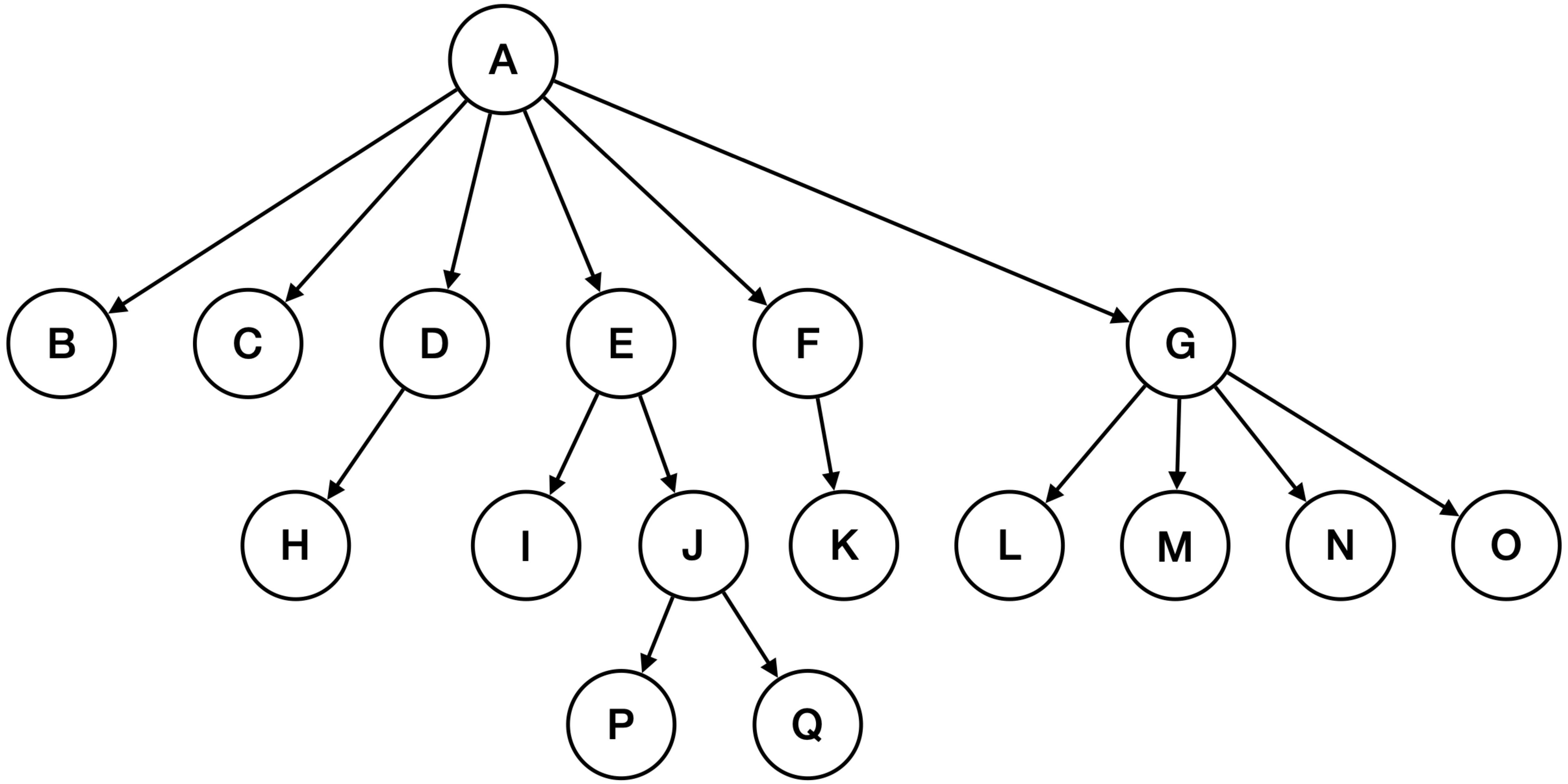
- A *tree* can be empty (NULL) or a node
  - where a node contains some data plus 1 *or more* pointers pointing to *trees*.



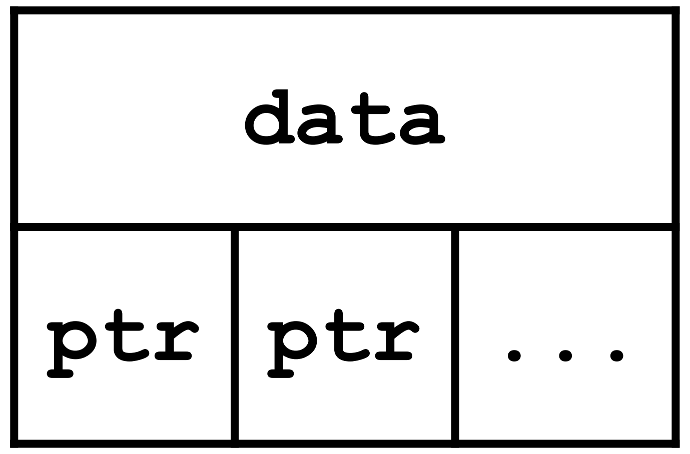
Trees



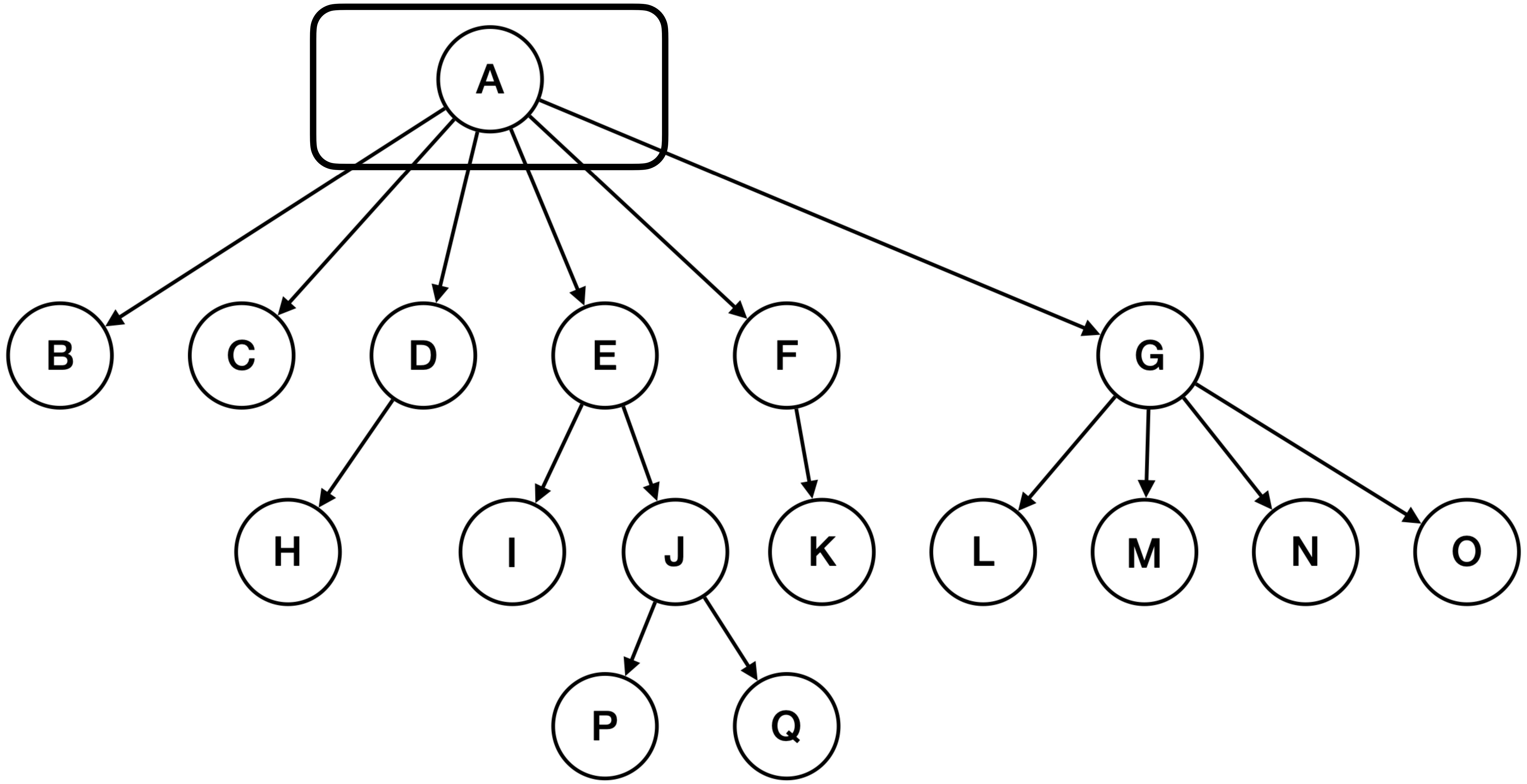
- A non-empty tree has a root



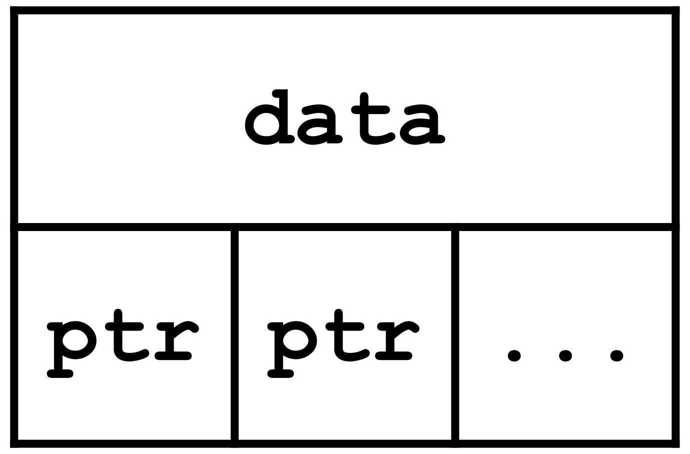
Trees



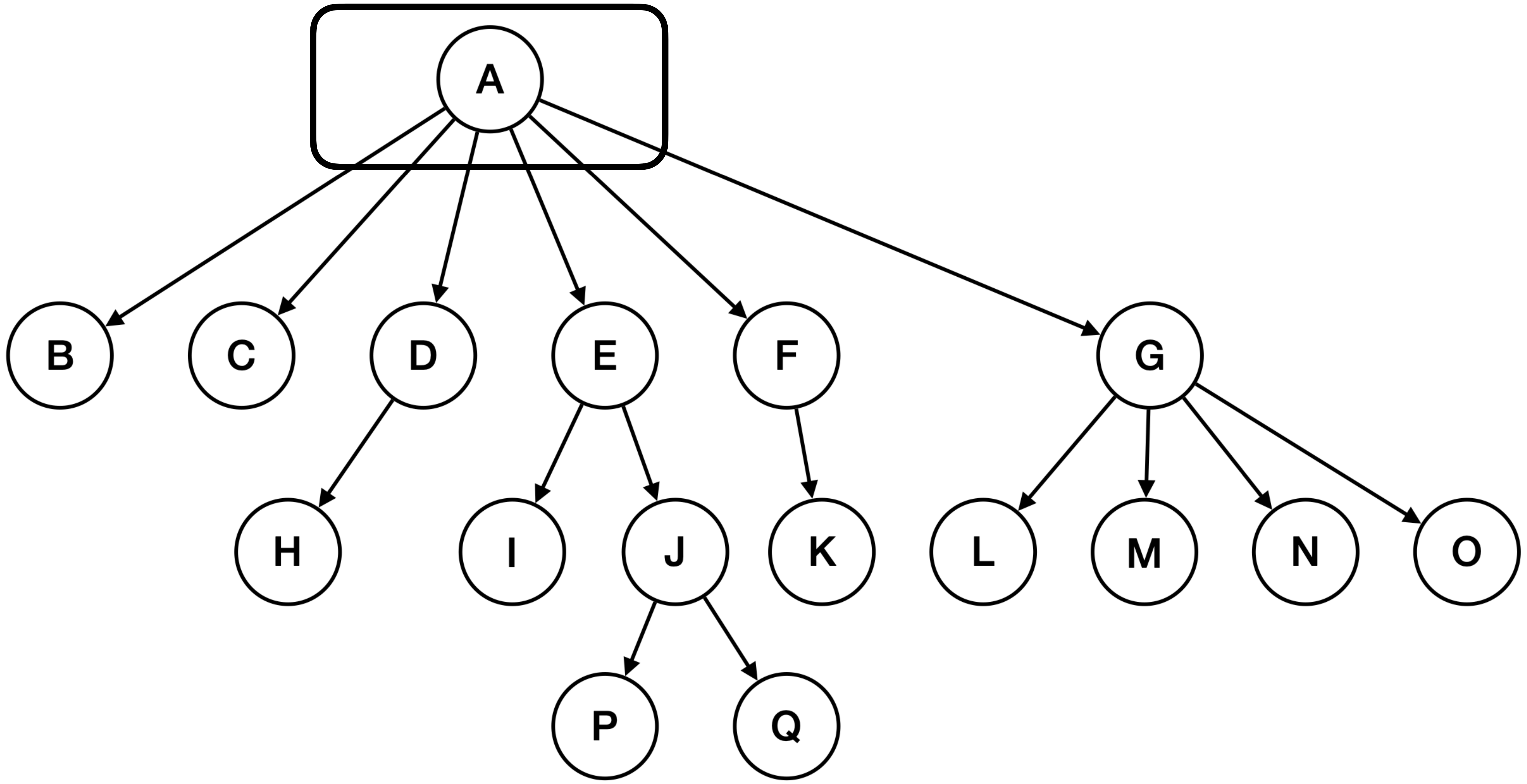
- A non-empty tree has a root



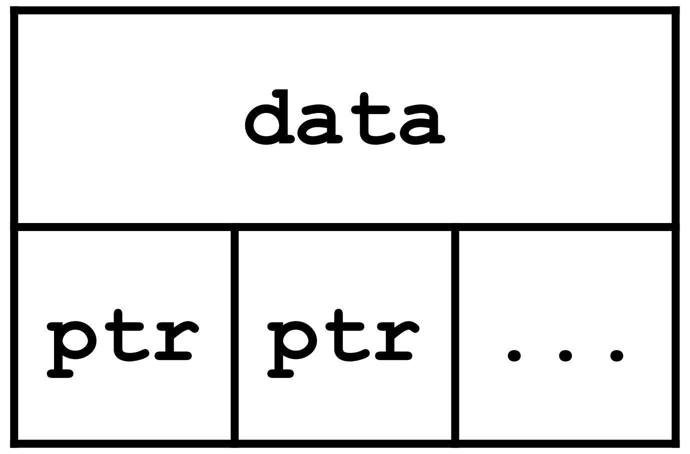
Trees



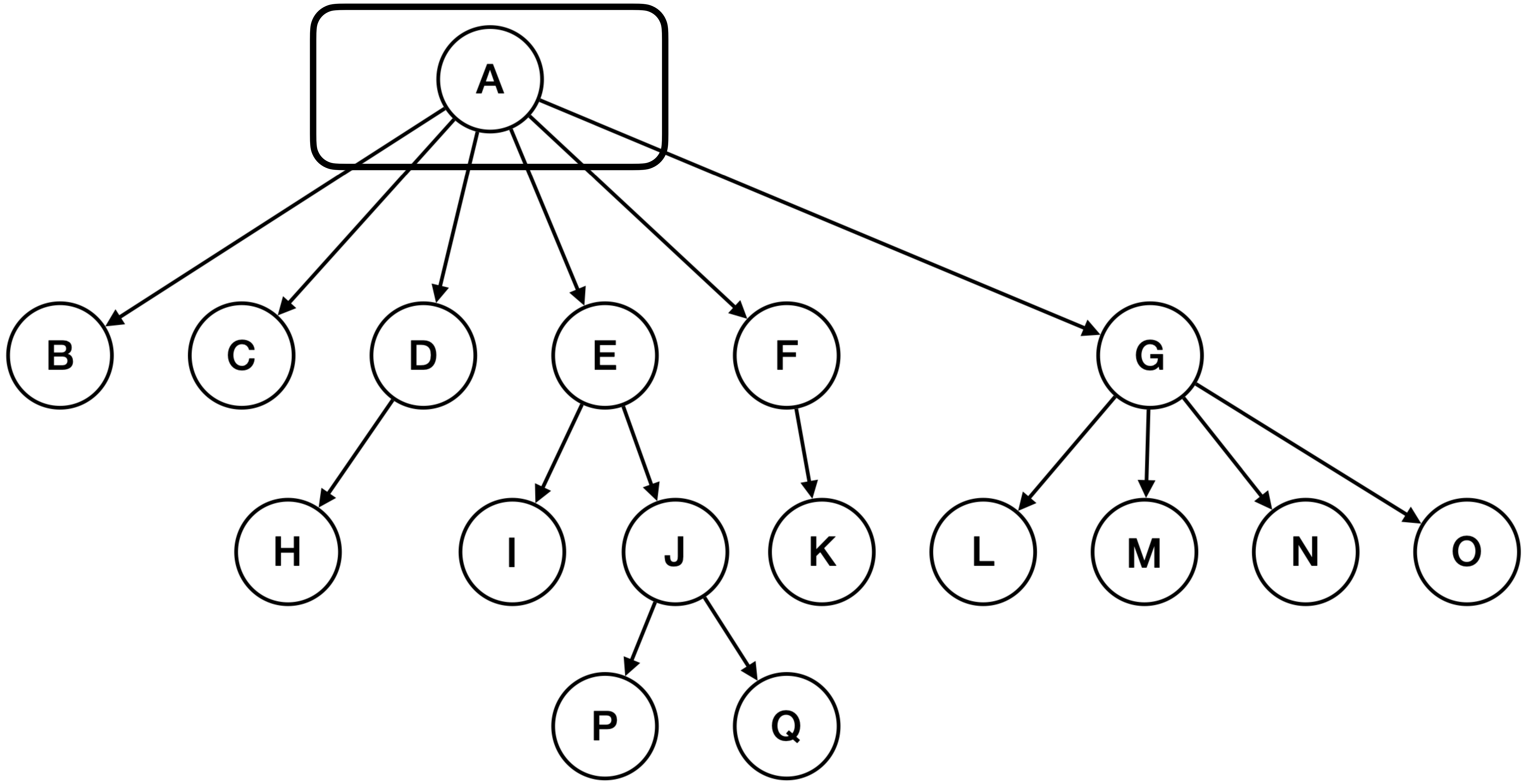
- A *parent* node points to multiple *child* nodes.



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.

Trees

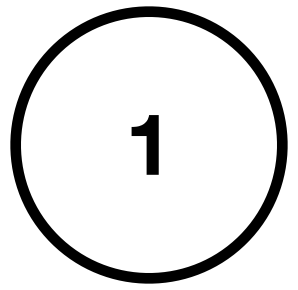
- A *tree* can be either
  - empty, *or*
  - a node contains some data plus 1 *or more* pointers pointing to *trees (subtrees)*.
- A *parent* node points to multiple *child* nodes.
- Every node has exactly one parent, except the root which has no parents.

Trees

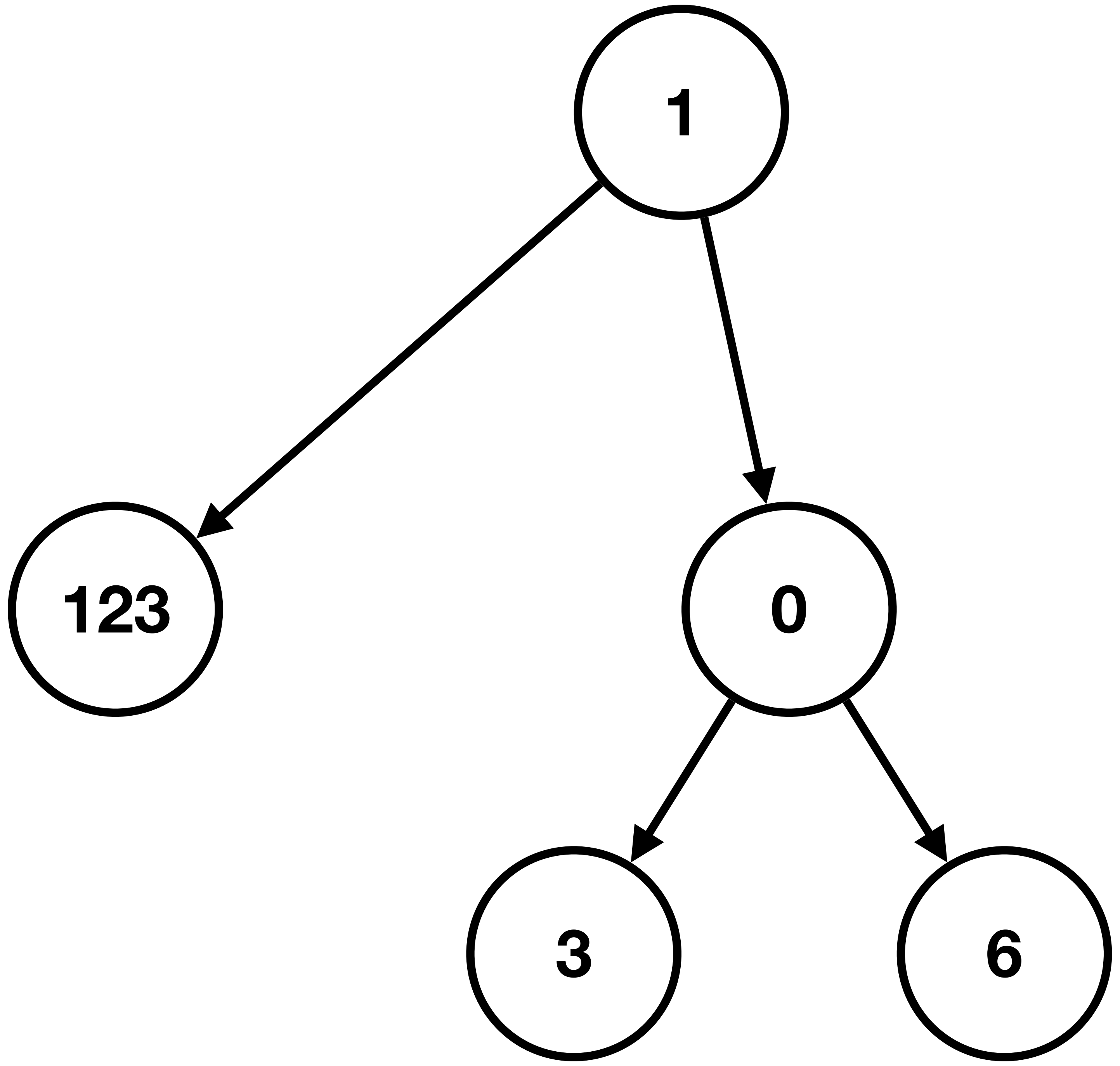
```
struct tree_node {
        void *elem;              /* the elem in this node        */
        struct tree_node *left;  /* pointer to the left subtree  */
        struct tree_node *right; /* pointer to the right subtree */
};
```

```
struct llist {
        void *elem;
        struct llist *next;
};
```

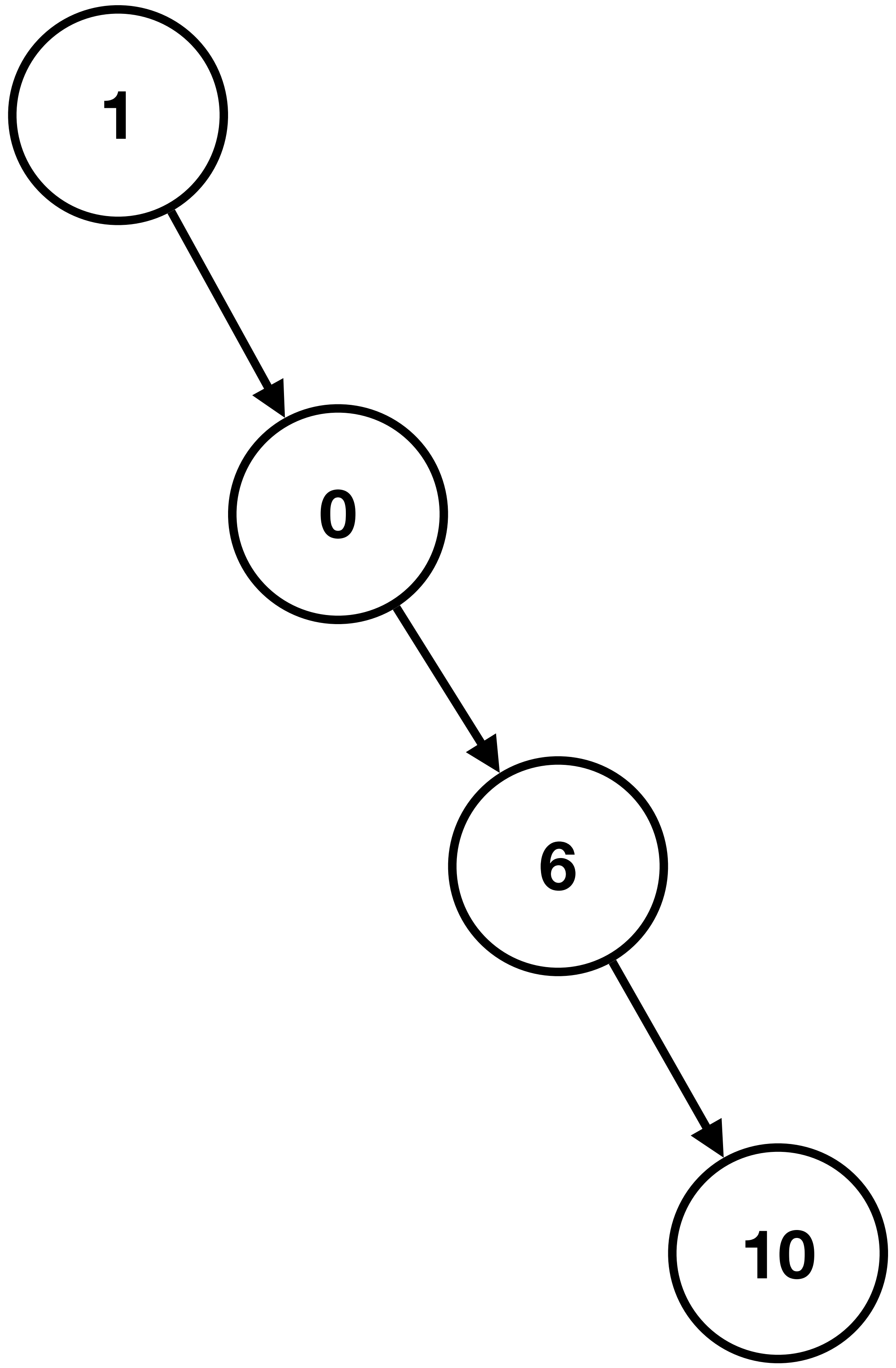
Is this a tree?



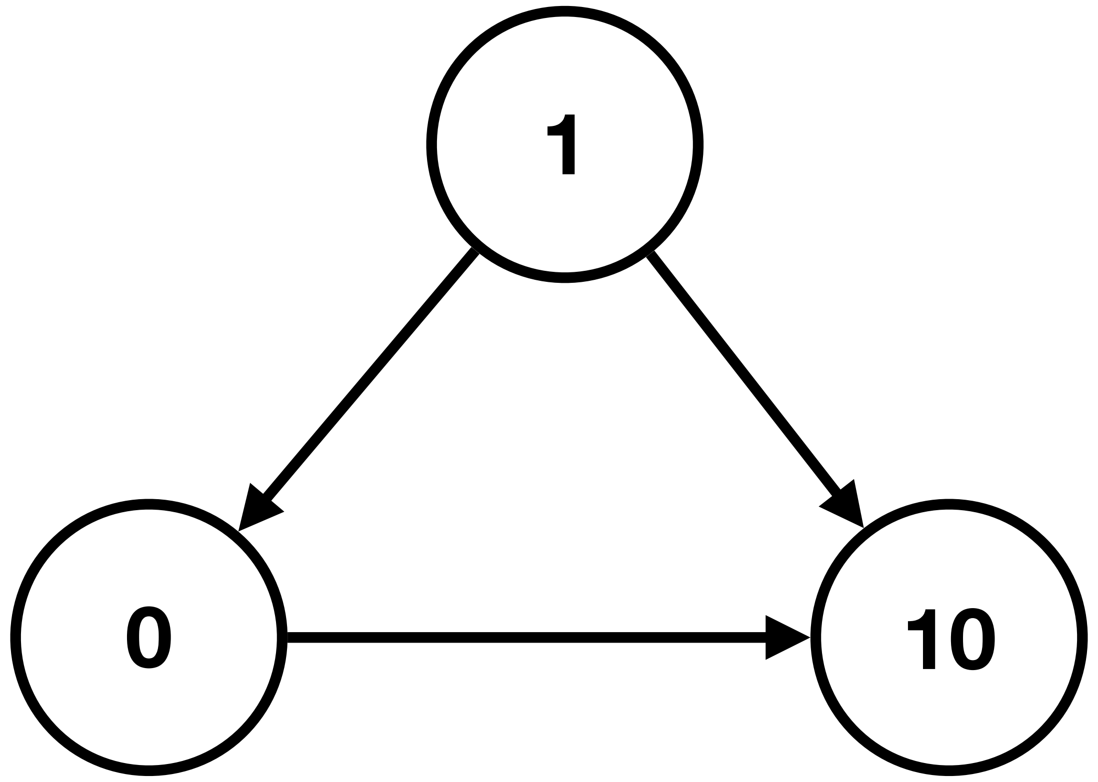
Is this a tree?



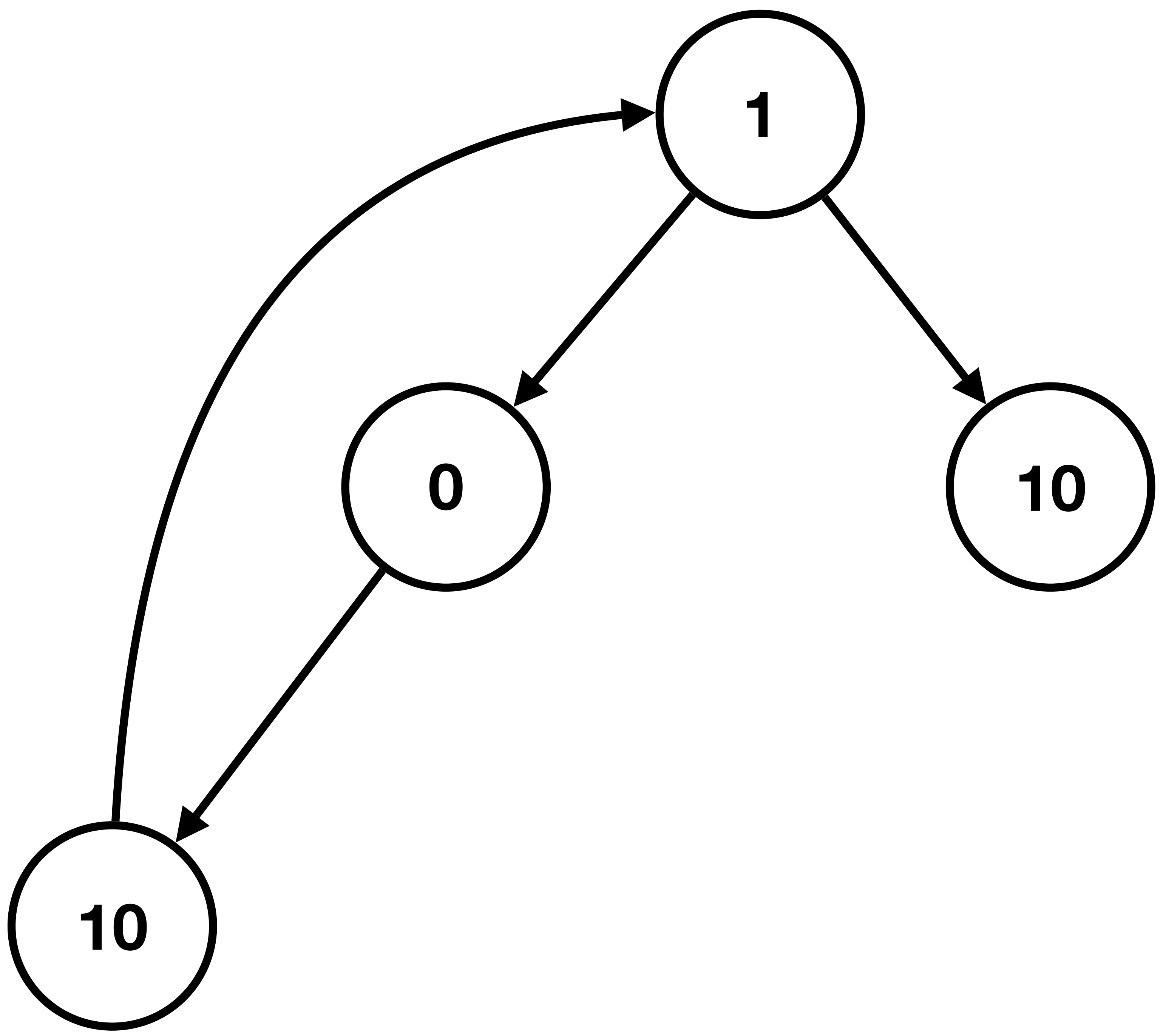
Is this a tree?



Is this a tree?



Is this a tree?

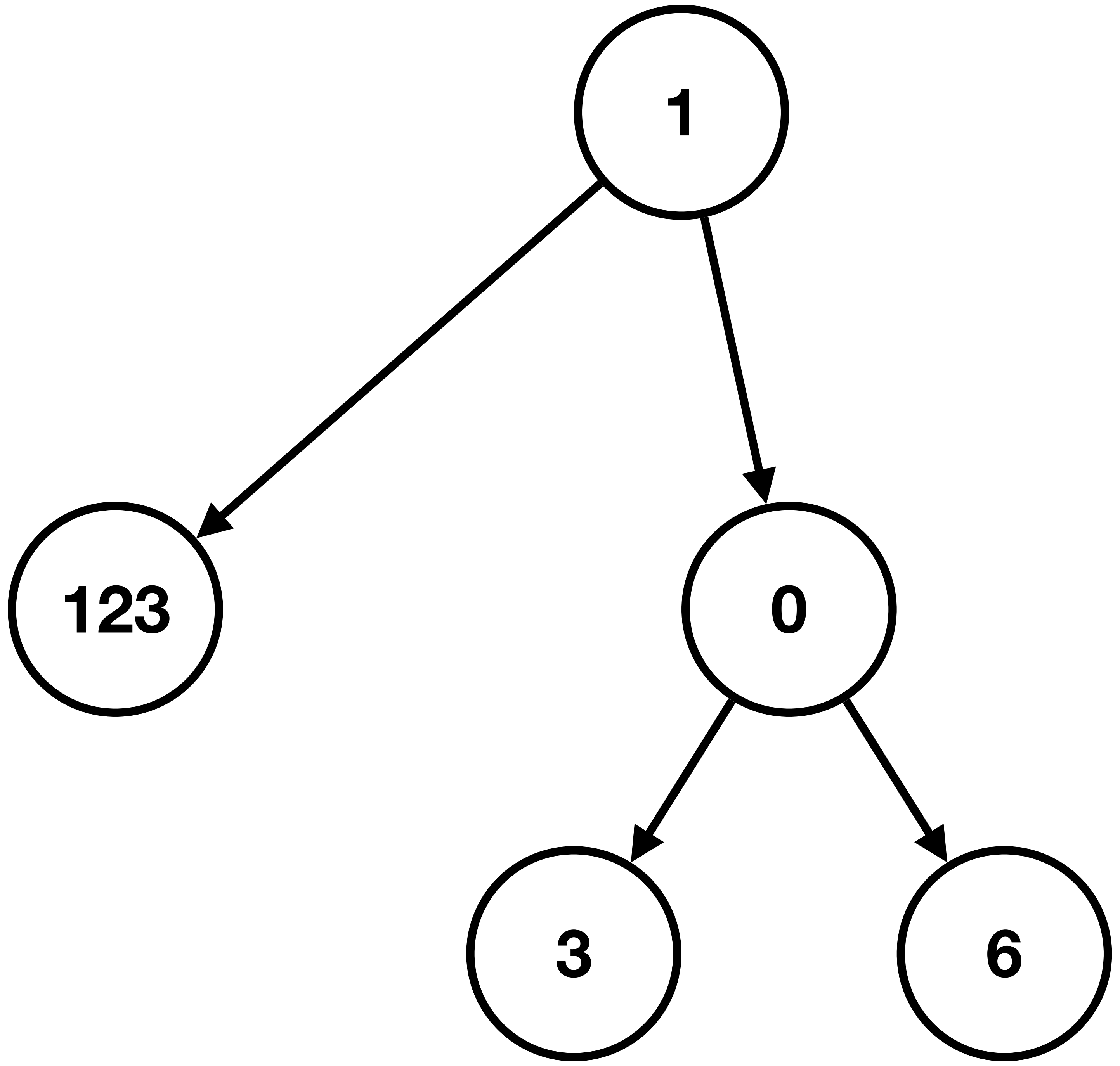


Is this a tree?

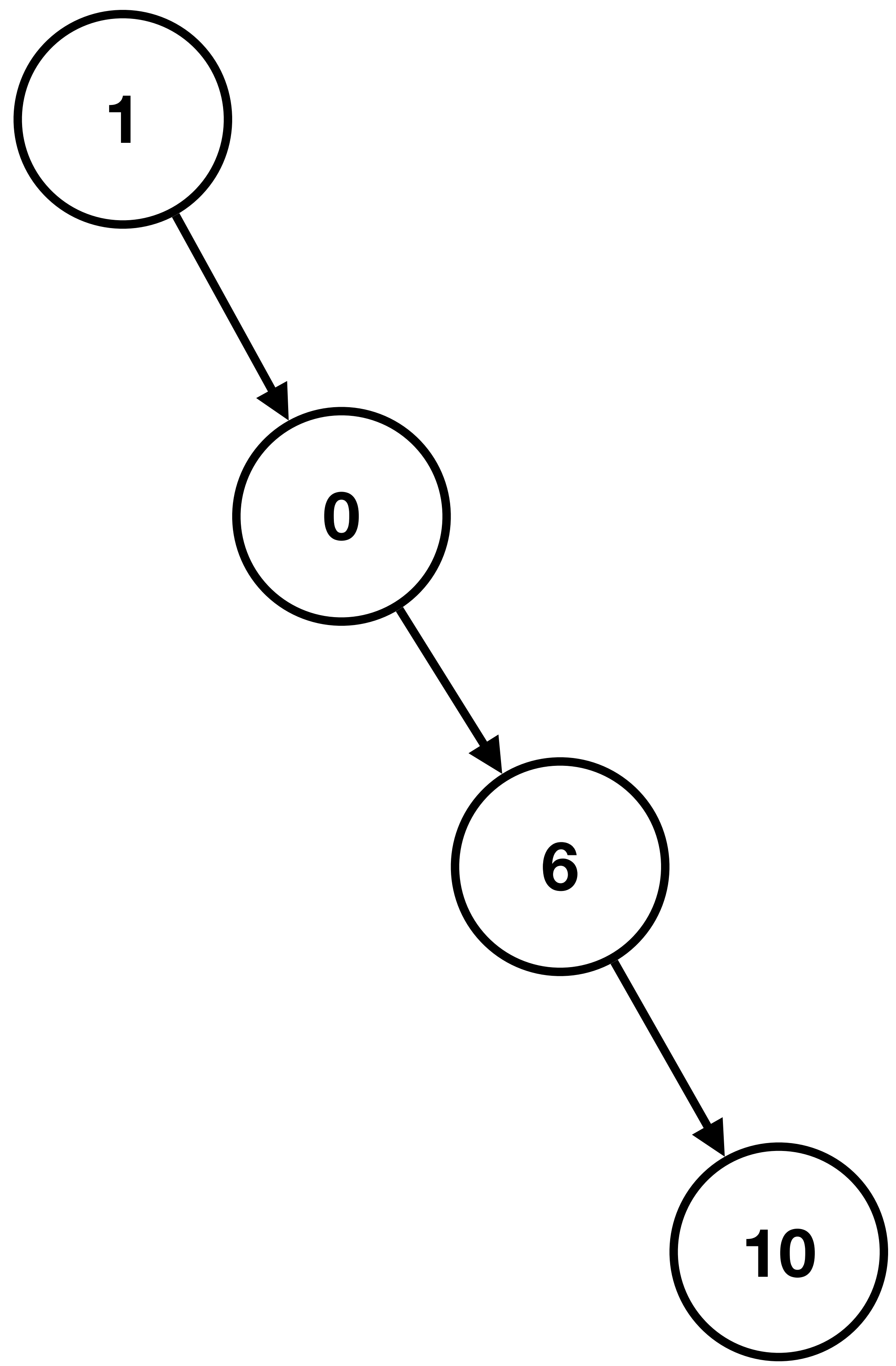
Binary Tree

- 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.

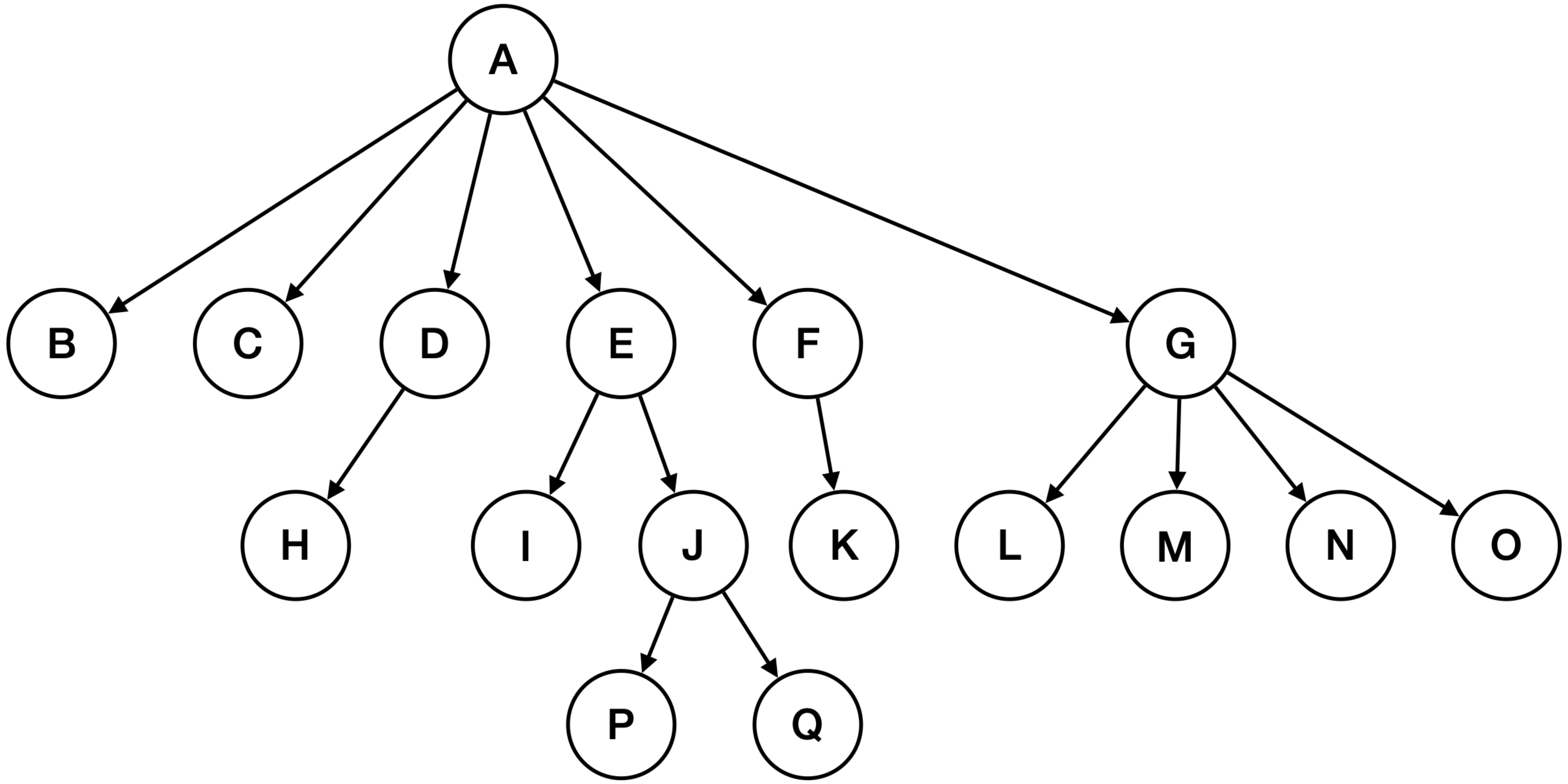
Is this a binary tree?



Is this a binary tree?



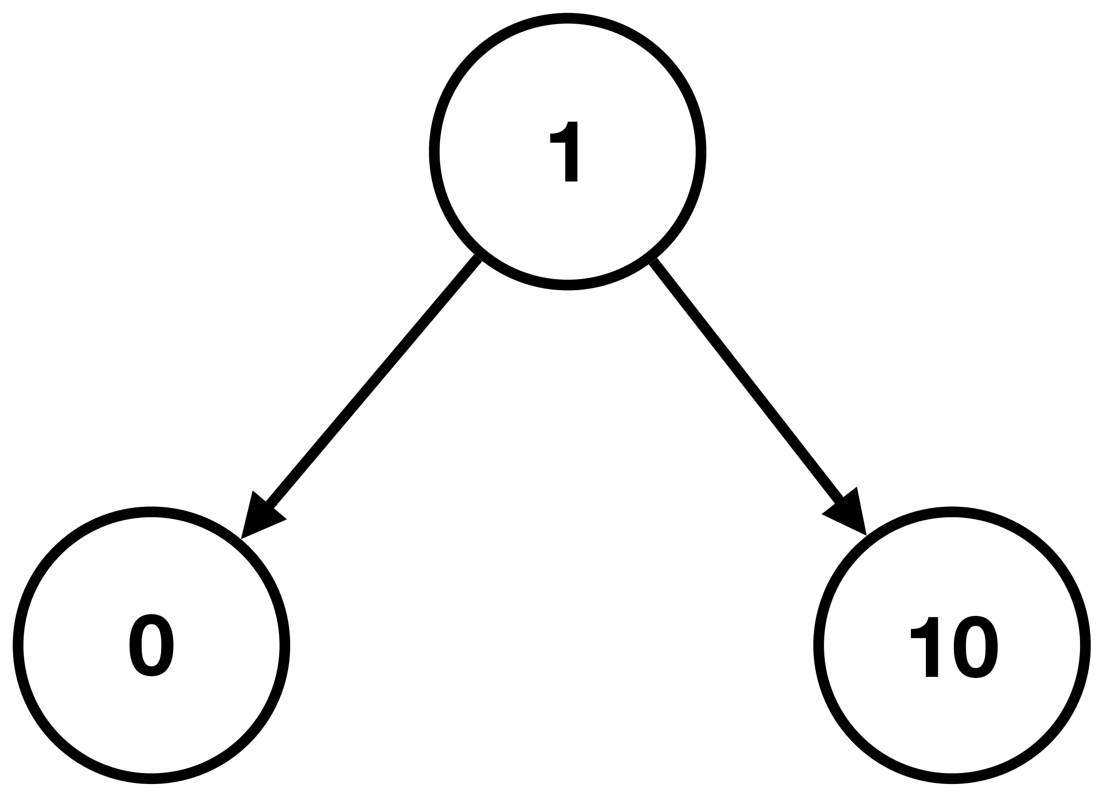
Is this a binary tree?



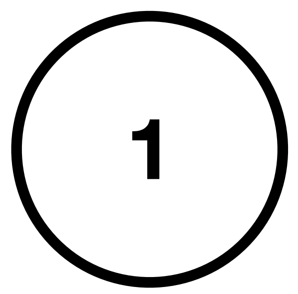
Binary Search Tree

- 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.

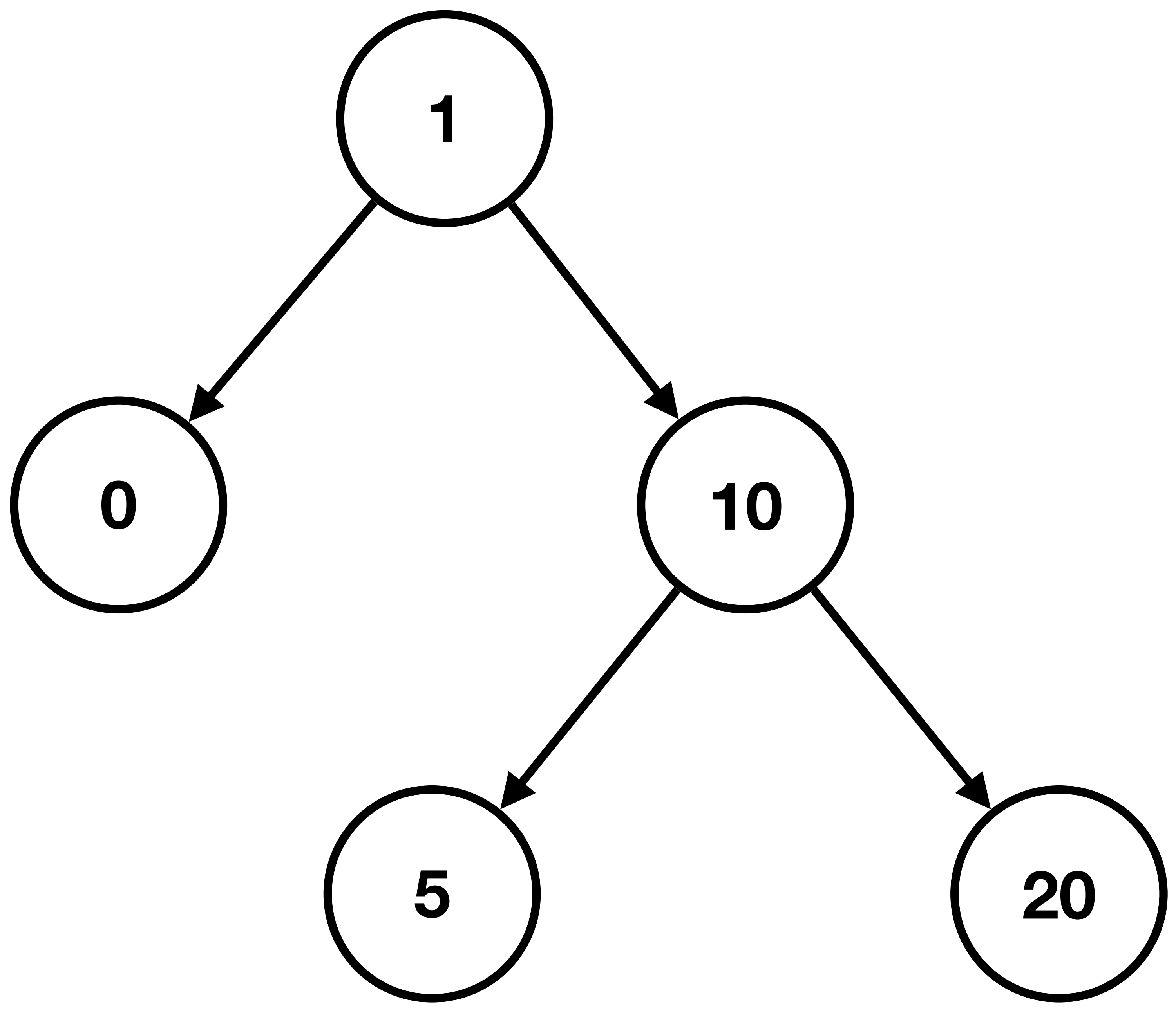
Is this a BST?



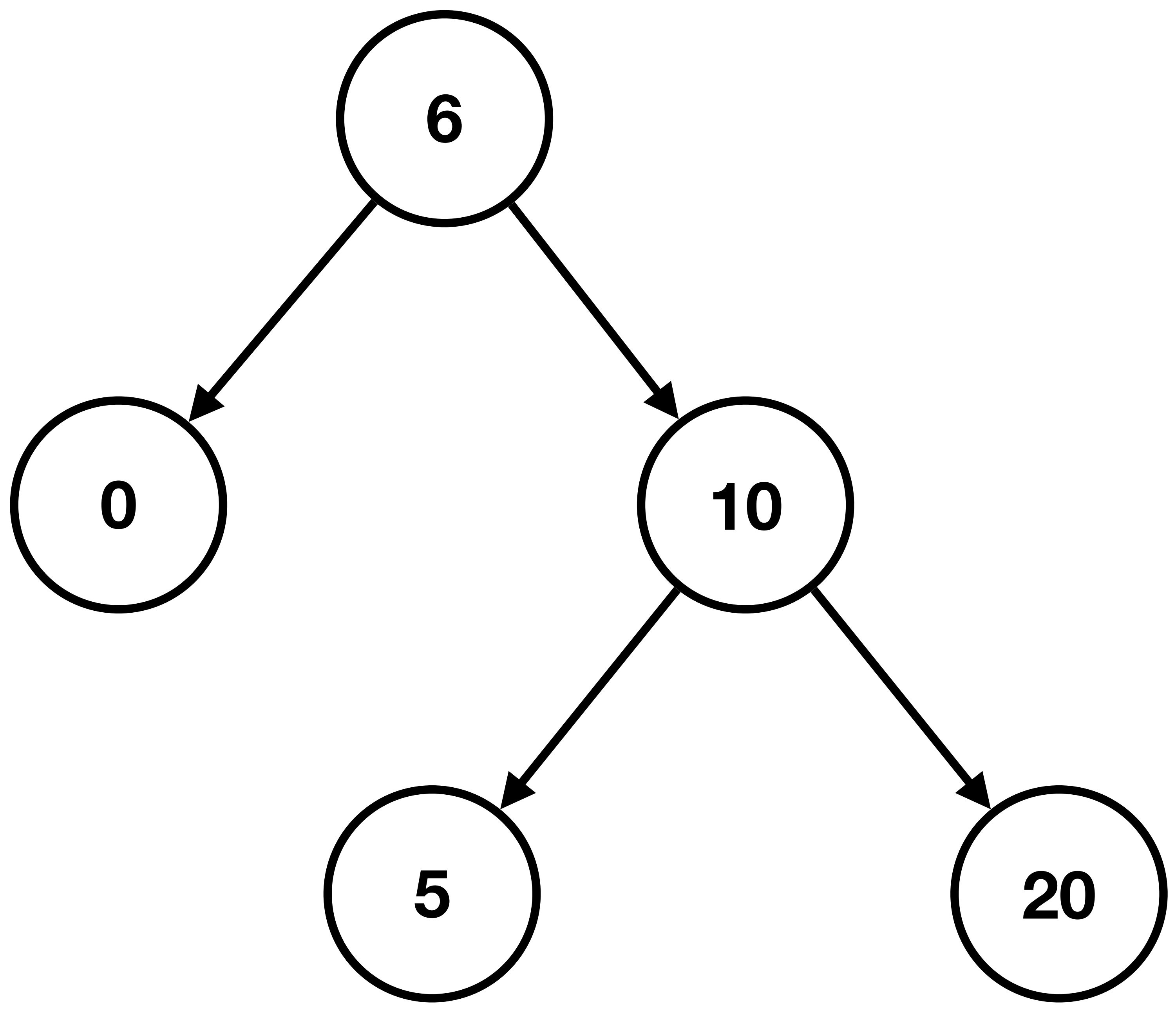
Is this a BST?



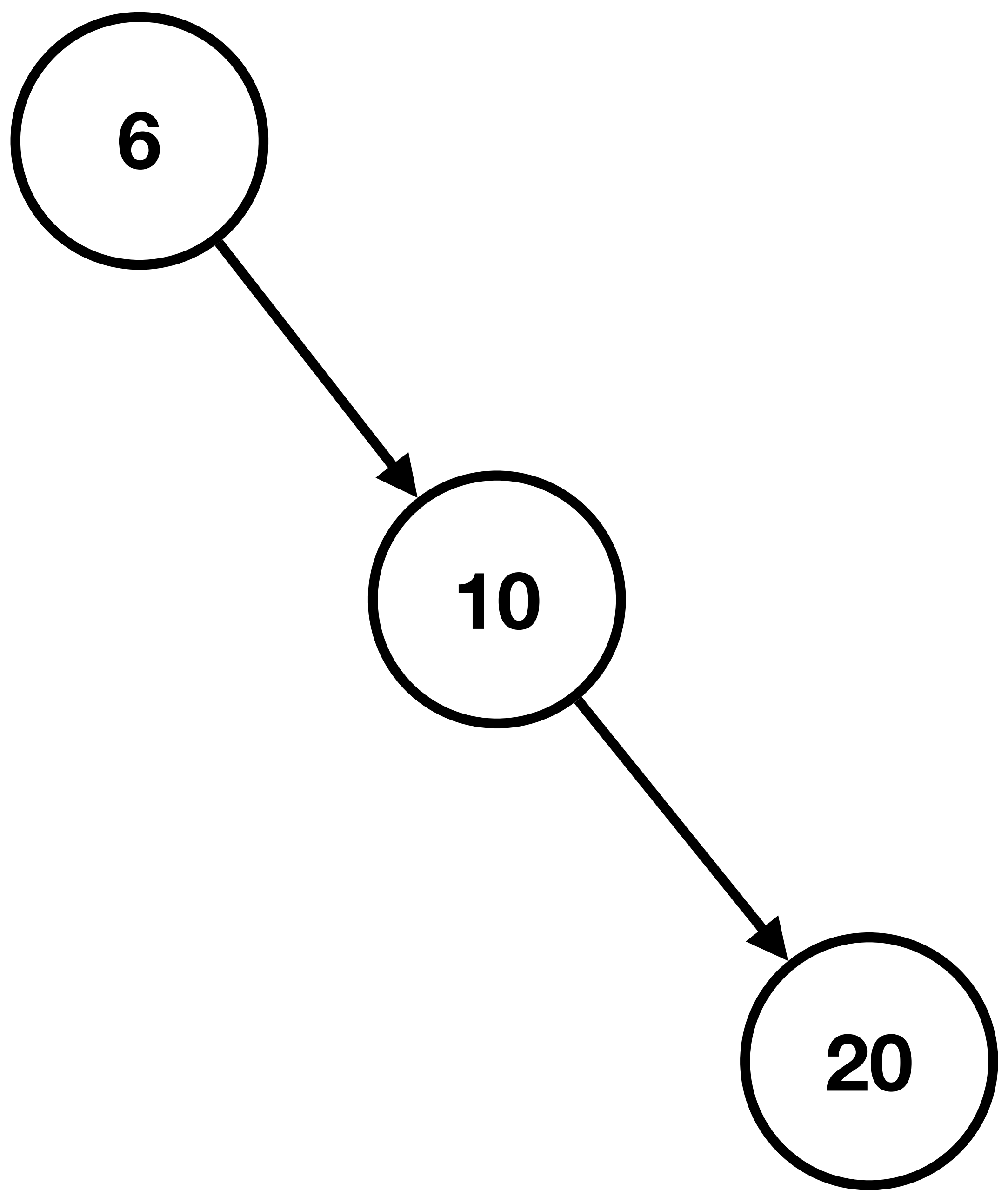
Is this a BST?



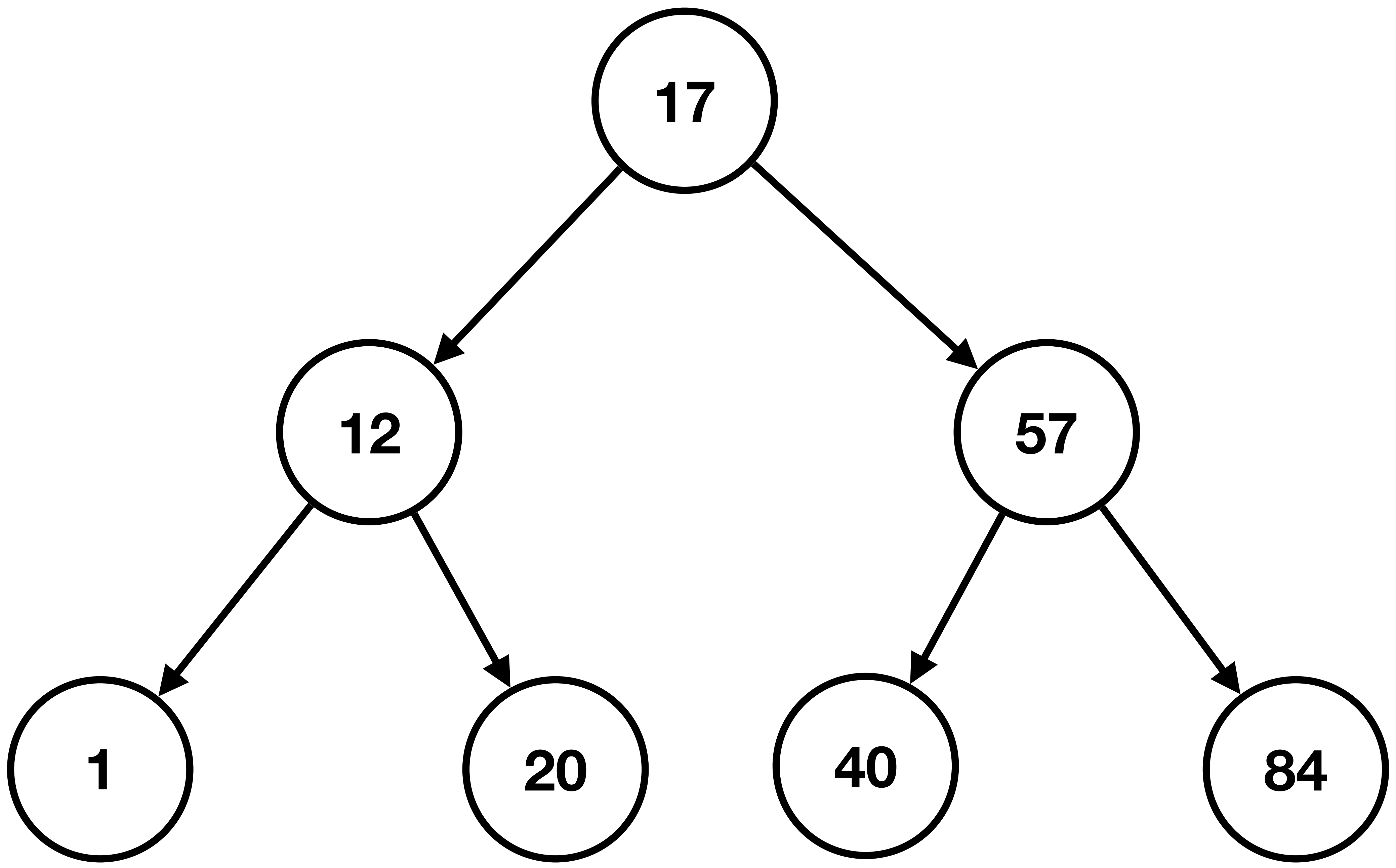
Is this a BST?



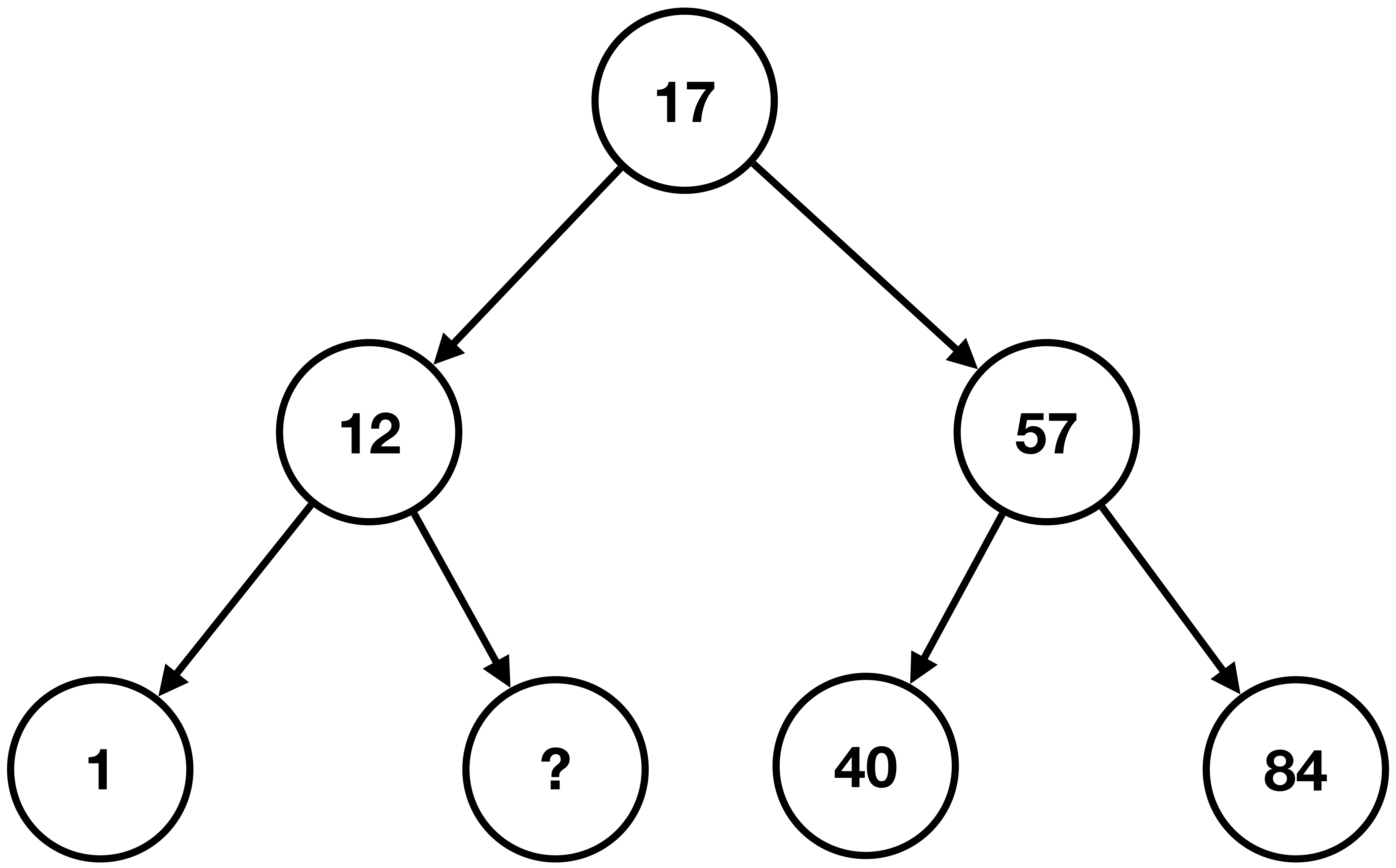
Is this a BST?



Is this a BST?



Is this a BST?

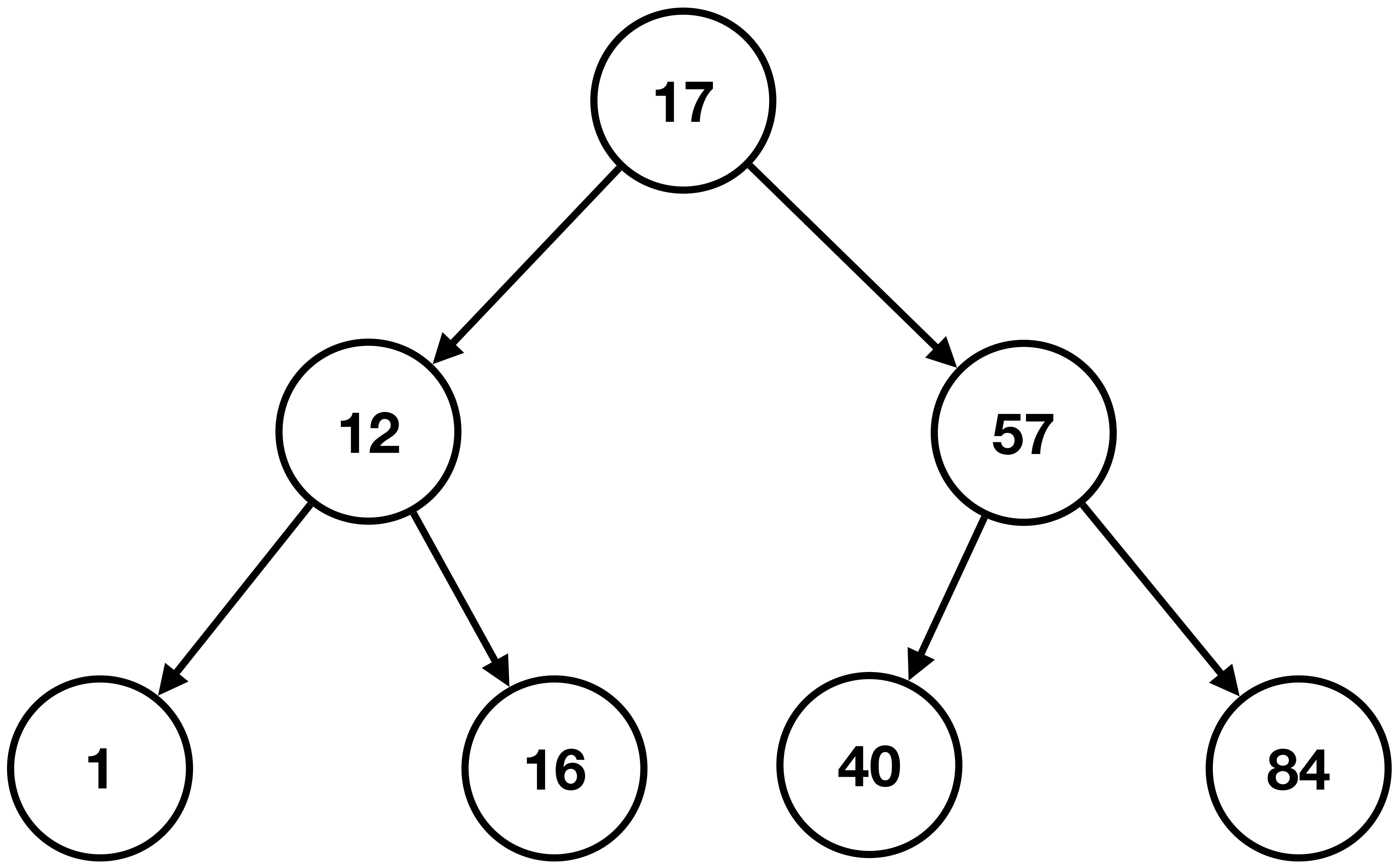


Binary Search Tree

- 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.

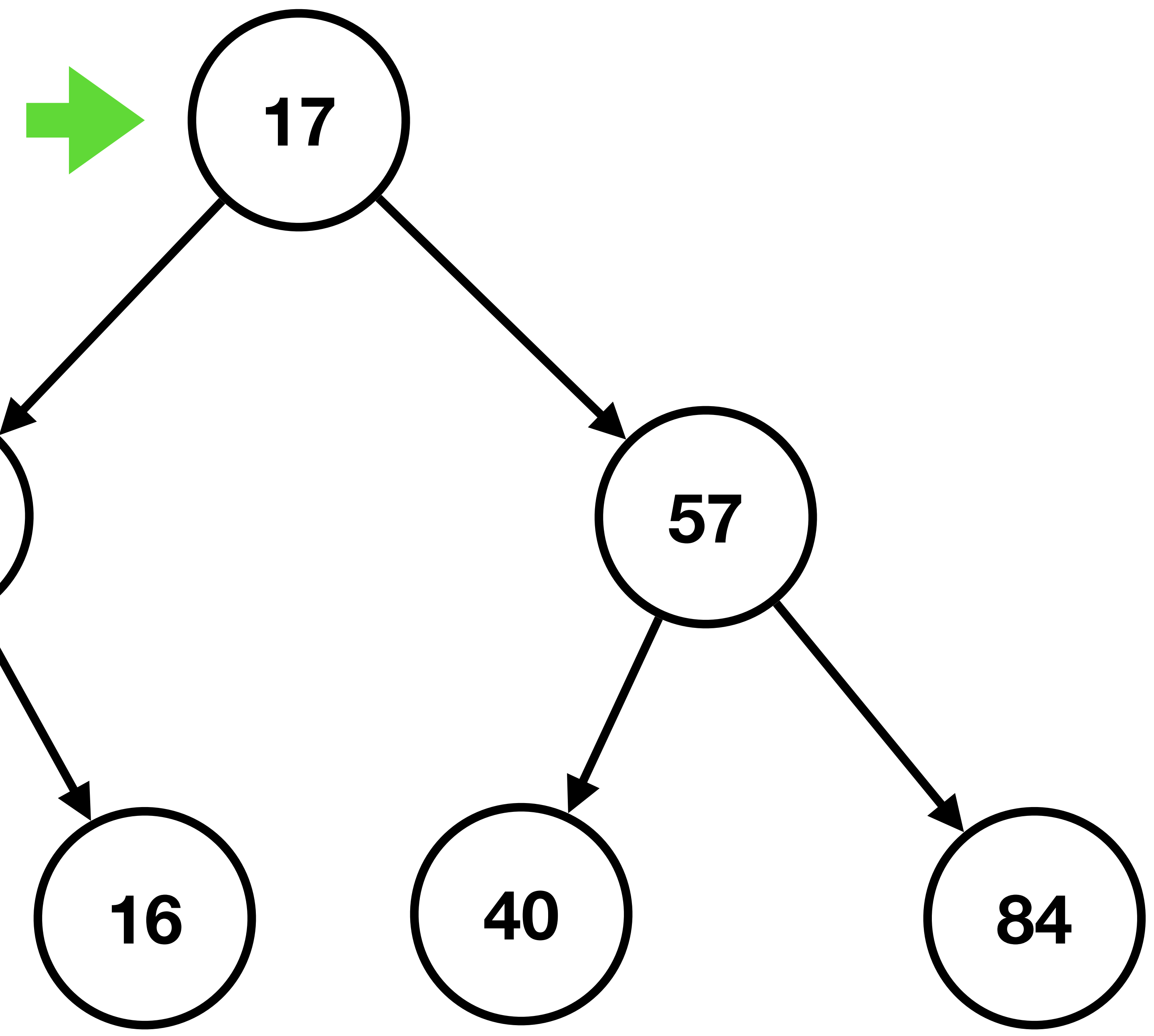
BST

Look up 16



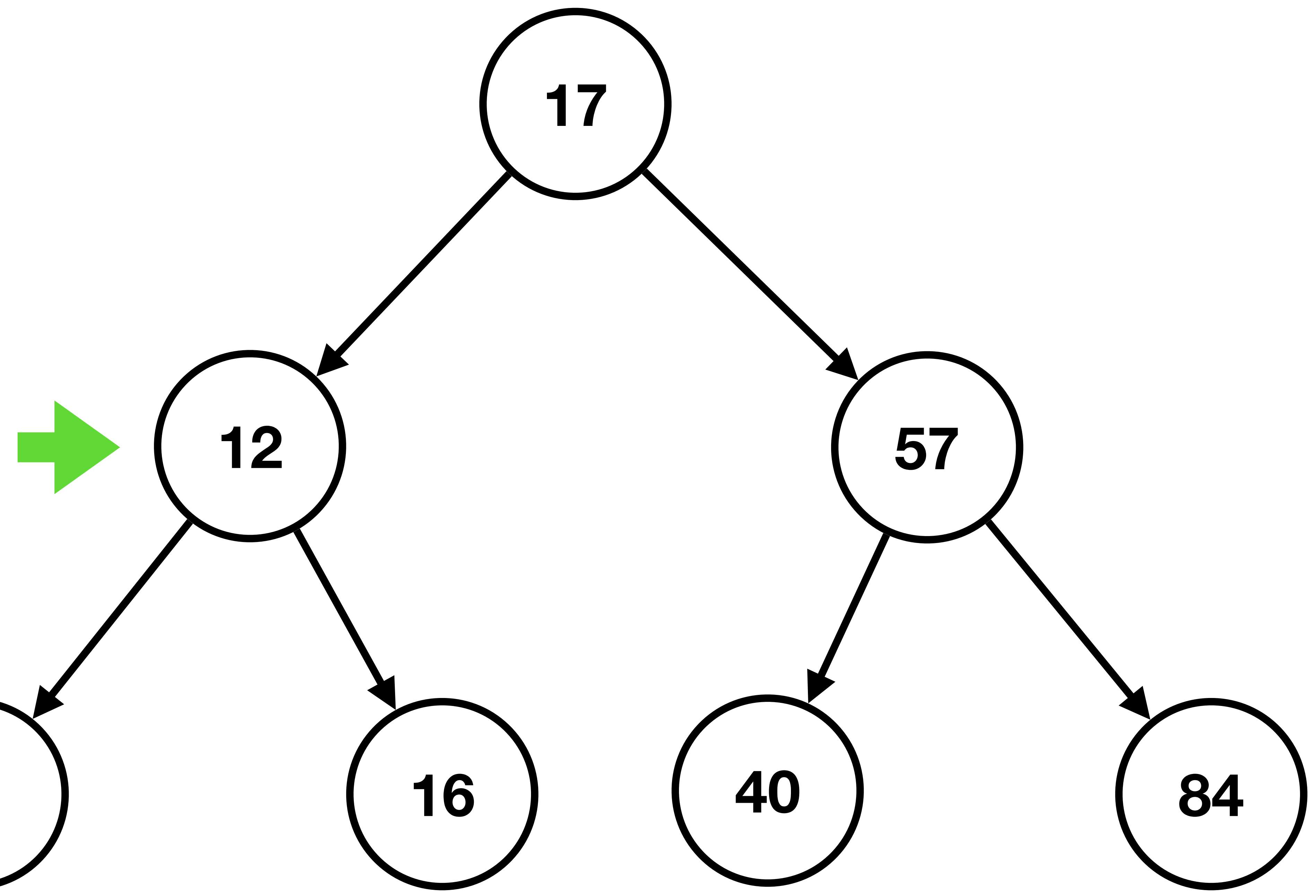
BST

Look up 16



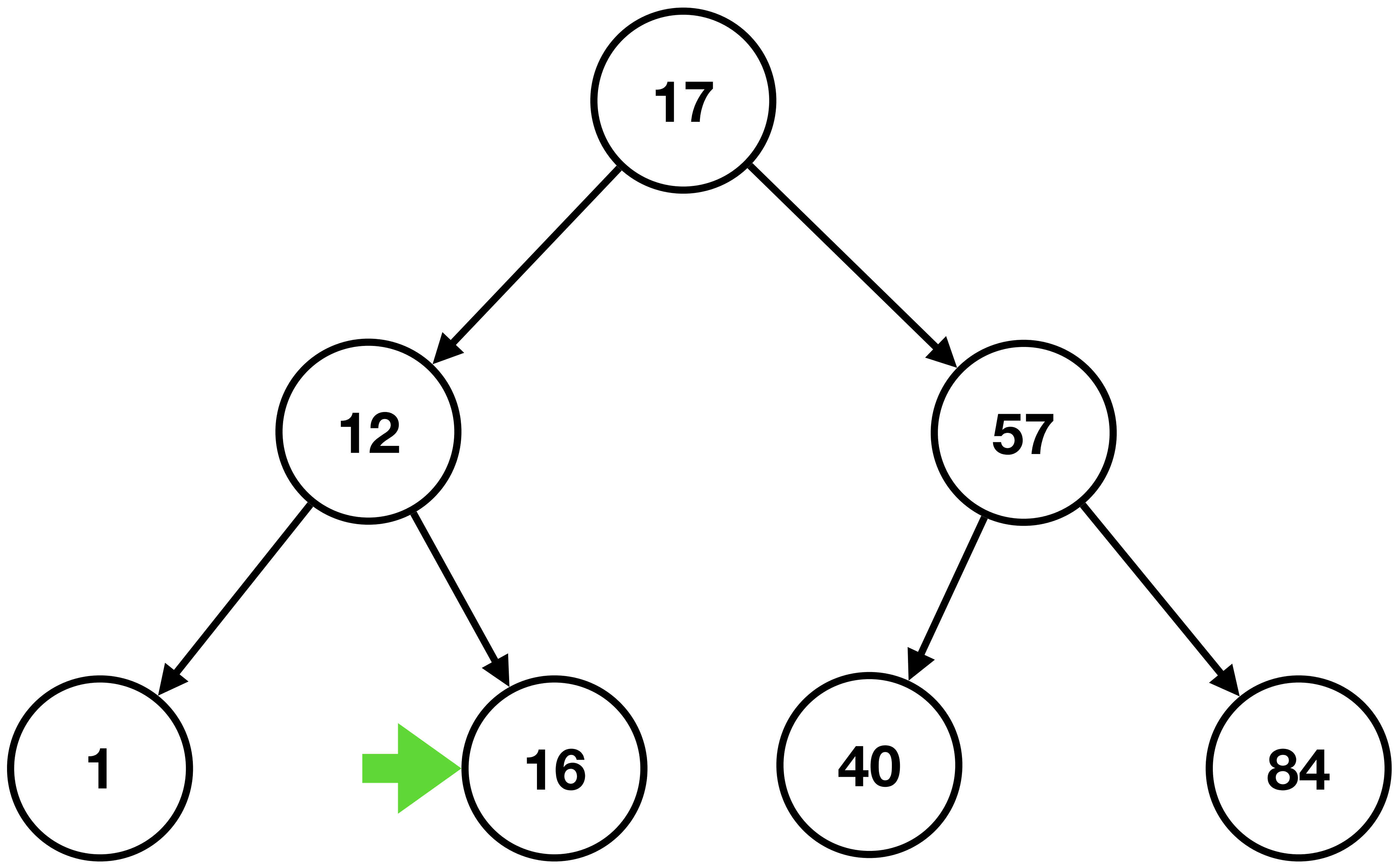
BST

Look up 16



BST

Look up 16



BST

Look up

BST

Look up

- For a given node *n* with key *k*,

BST

Look up

- For a given node *n* with key *k*,
  - If *k* is what we want, return the data.

BST

Look up

- For a given node *n* with key *k*,
  - If *k* is what we want, return the data.
  - If what we want < *k*, explore left

BST

Look up

- For a given node *n* with key *k*,
  - If *k* is what we want, return the data.
  - If what we want $< k$, explore left
  - If what we want $> k$, explore right

BST
Look up

- For a given node *n* with key *k*,
  - If *k* is what we want, return the data.
  - If what we want $< k$, explore left
  - If what we want $> k$, explore right
- Complexity?

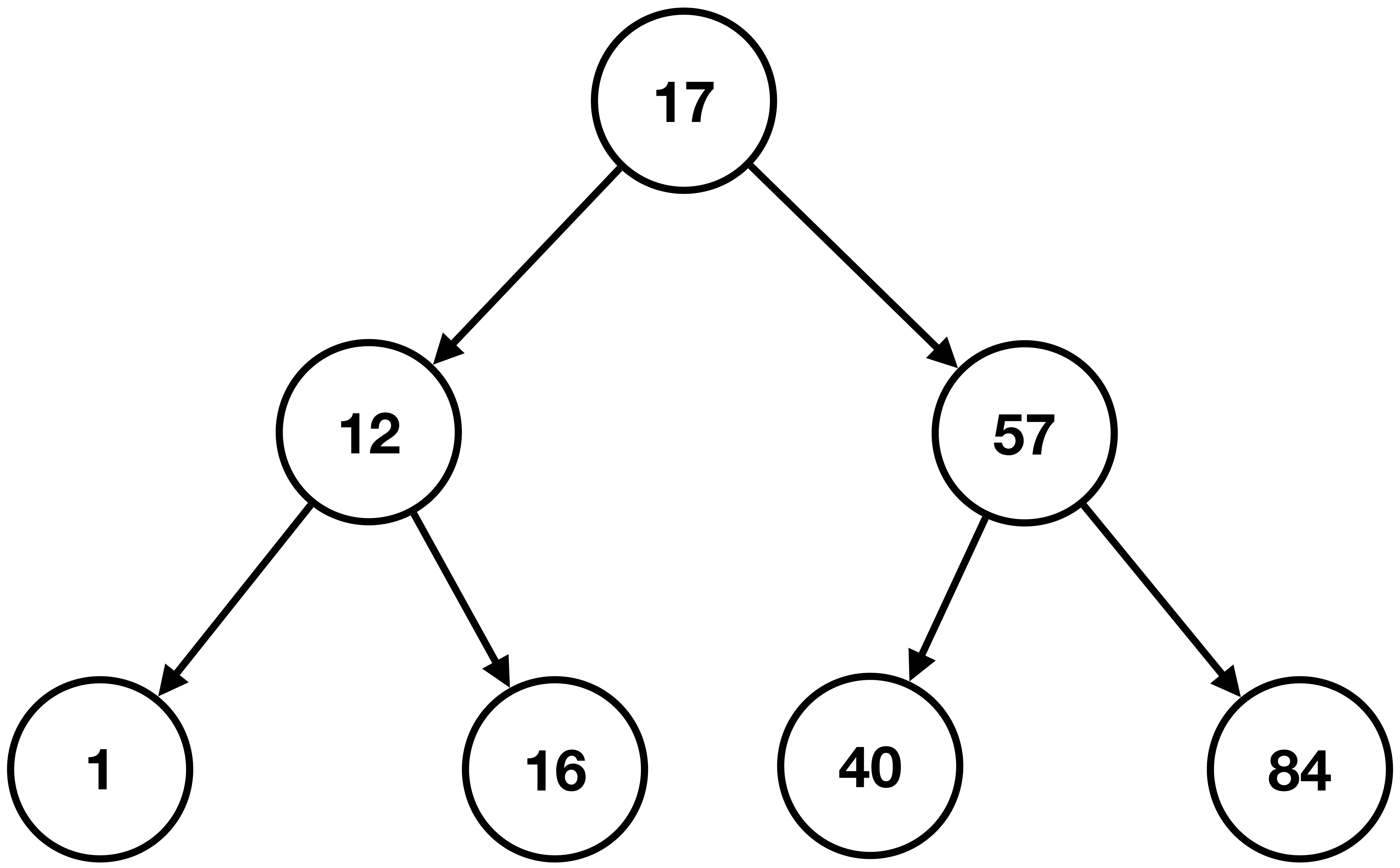
BST

Look up

- For a given node *n* with key *k*,
  - If *k* is what we want, return the data.
  - If what we want $<$ *k*, explore left
  - If what we want $>$ *k*, explore right
- Complexity?
  - $O(\text{height})$

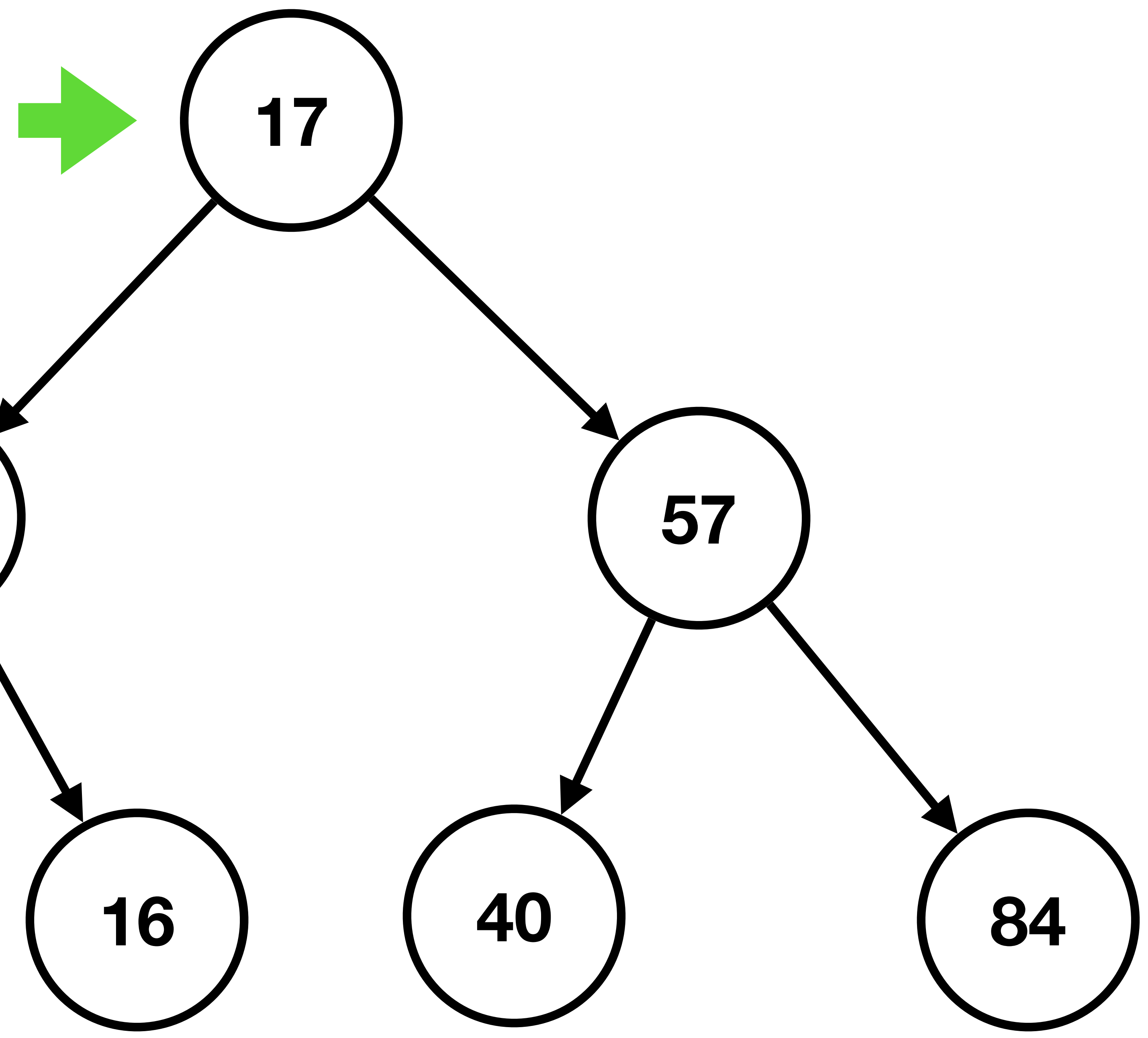
BST

Insert 18



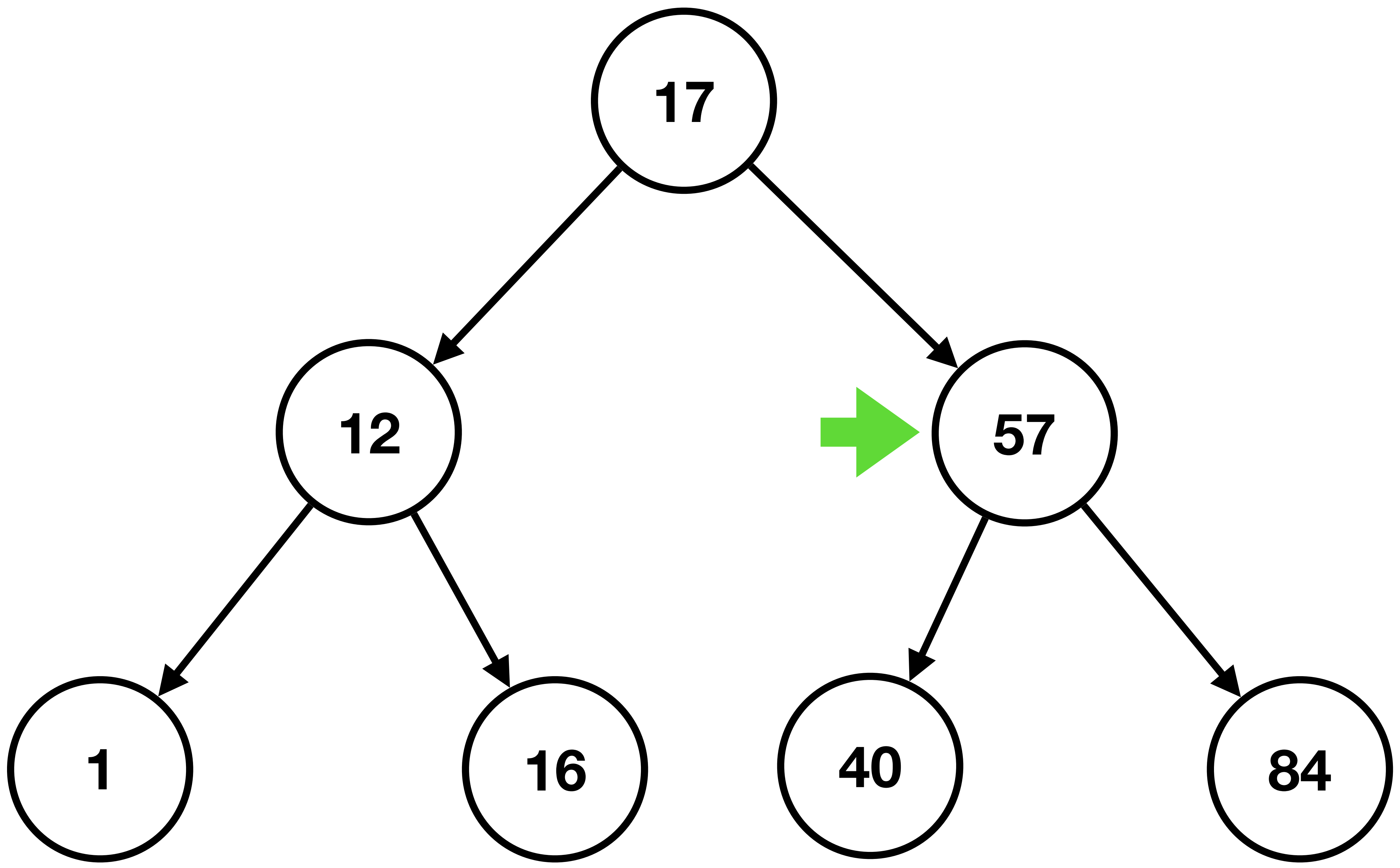
BST

Insert 18



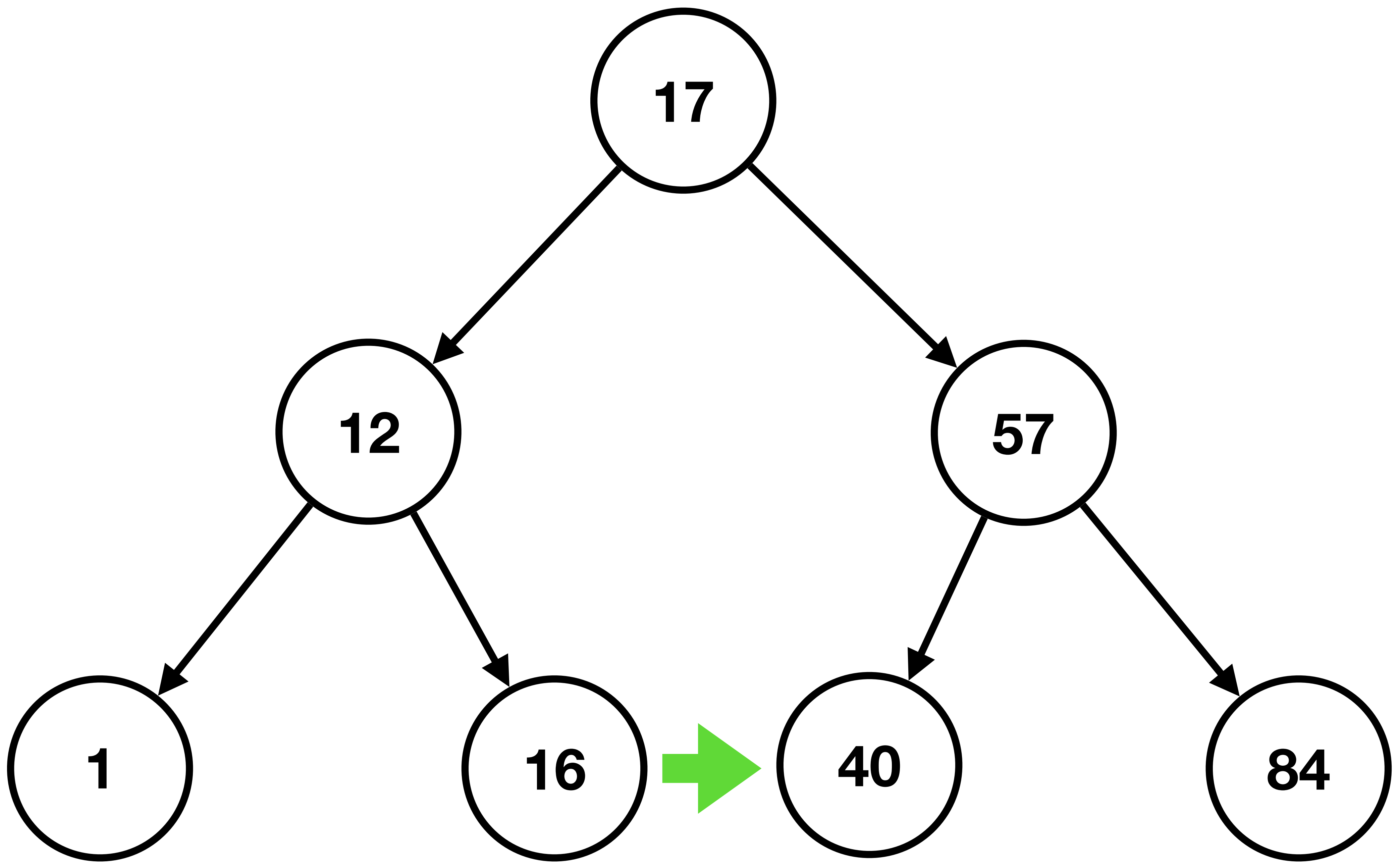
BST

Insert 18



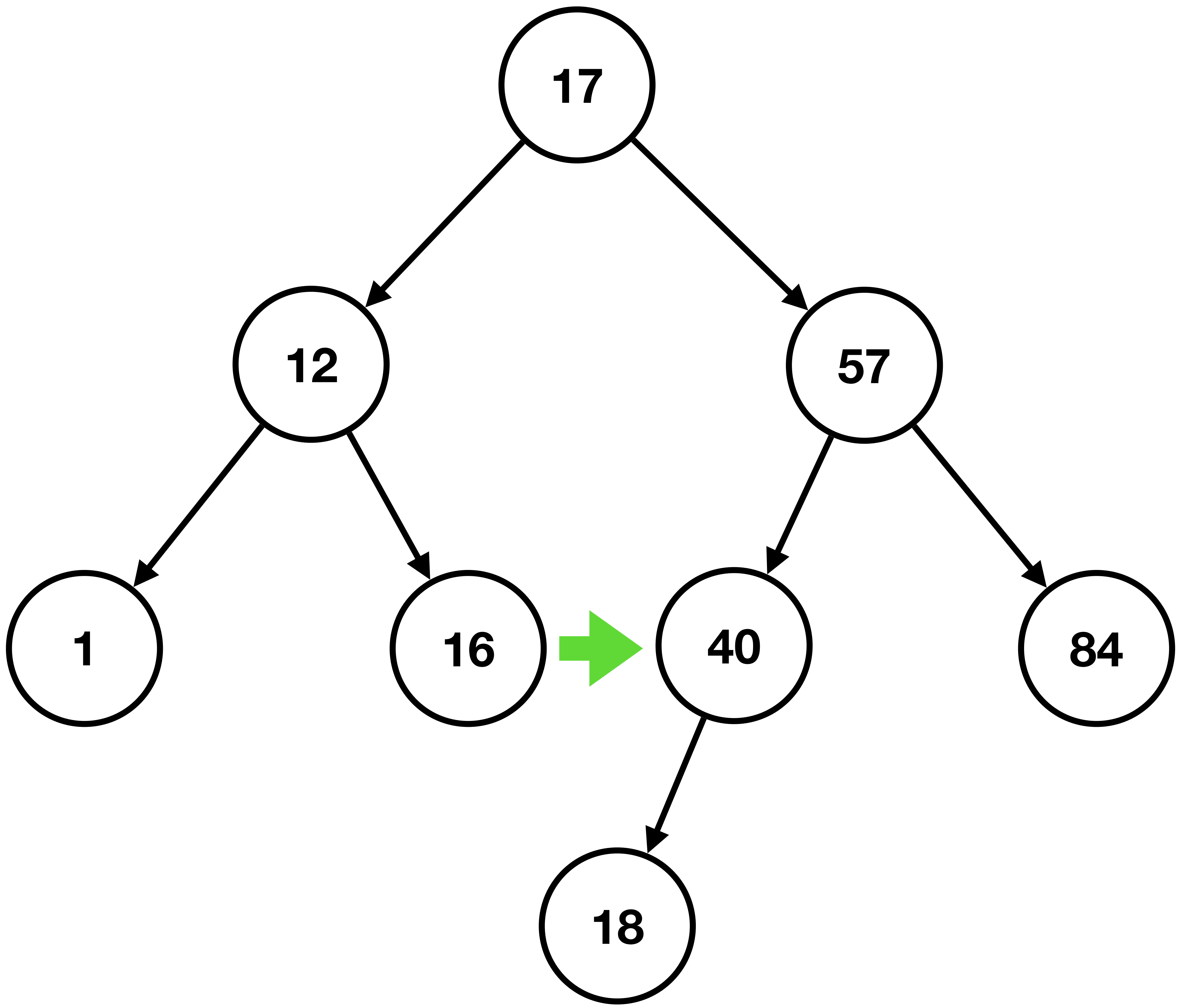
BST

Insert 18



BST

Insert 18



BST

Insert

BST
Insert

- Tree is empty: Make new node, set it as root

BST

Insert

- Tree is empty: Make new node, set it as root
- If item < key, insert left

BST

Insert

- Tree is empty: Make new node, set it as root
- If item < key, insert left
- If item > key, insert right

BST

Insert

- Tree is empty: Make new node, set it as root
- If item < key, insert left
- If item > key, insert right
- if Item == key, replace the node

BST
Insert

- Tree is empty: Make new node, set it as root
- If item < key, insert left
- If item > key, insert right
- if Item == key, replace the node
- Complexity?

BST

Insert

- Tree is empty: Make new node, set it as root
- If item $<$ key, insert left
- If item $>$ key, insert right
- if Item == key, replace the node
- Complexity?
    1. Find correct spot in tree to insert $O(\text{height})$

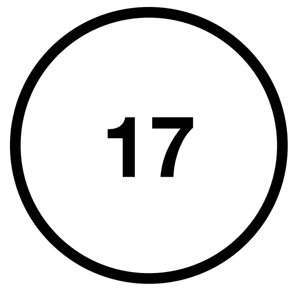
BST
Insert

- Tree is empty: Make new node, set it as root
- If item < key, insert left
- If item > key, insert right
- if Item == key, replace the node
- Complexity?
    1. Find correct spot in tree to insert $O(\text{height})$
    2. Create a new node and return pointer $O(1)$

BST

Height

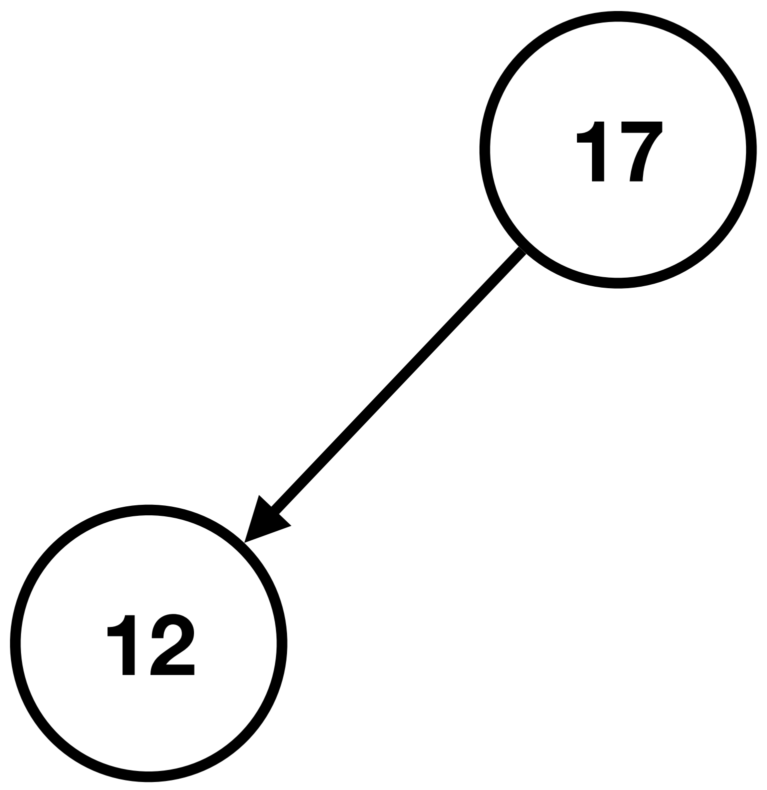
- Insert: 17, 12, 57, 1, 16, 40, 84



BST

Height

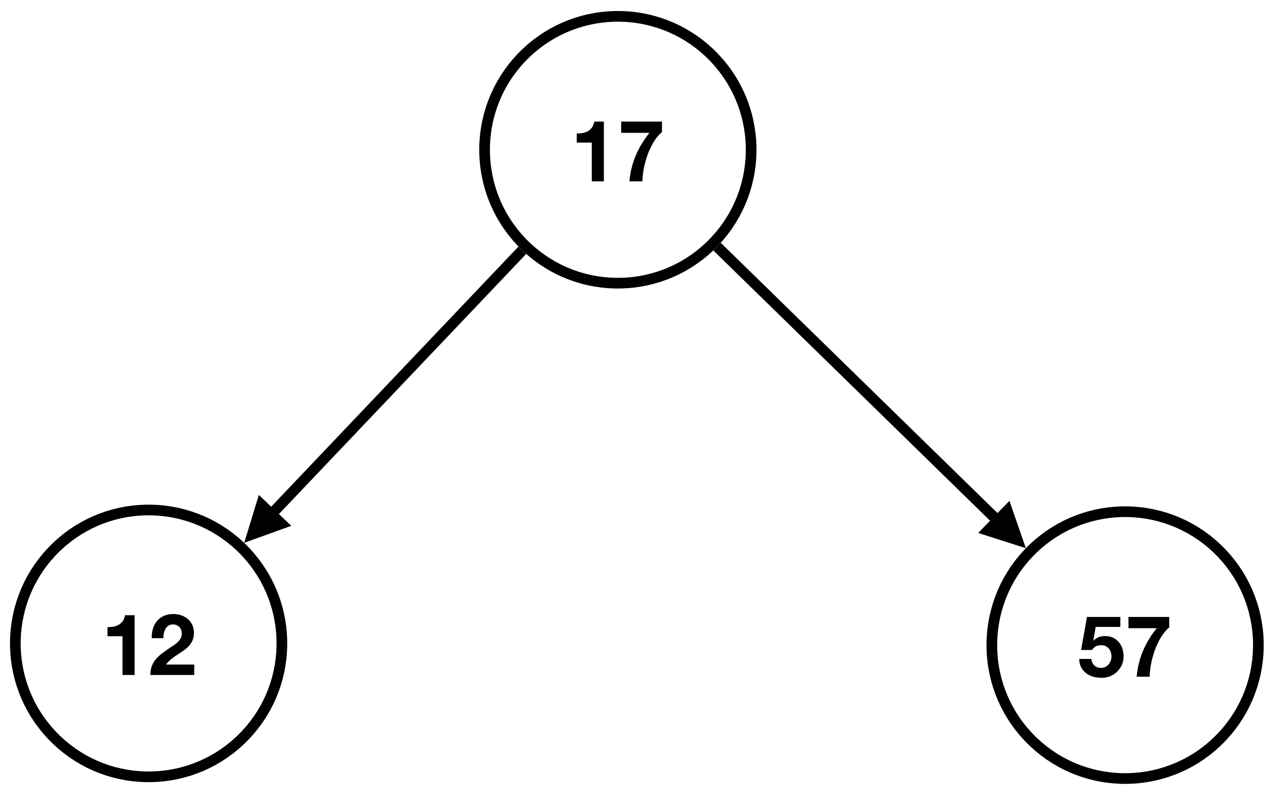
- Insert: 17, 12, 57, 1, 16, 40, 84



BST

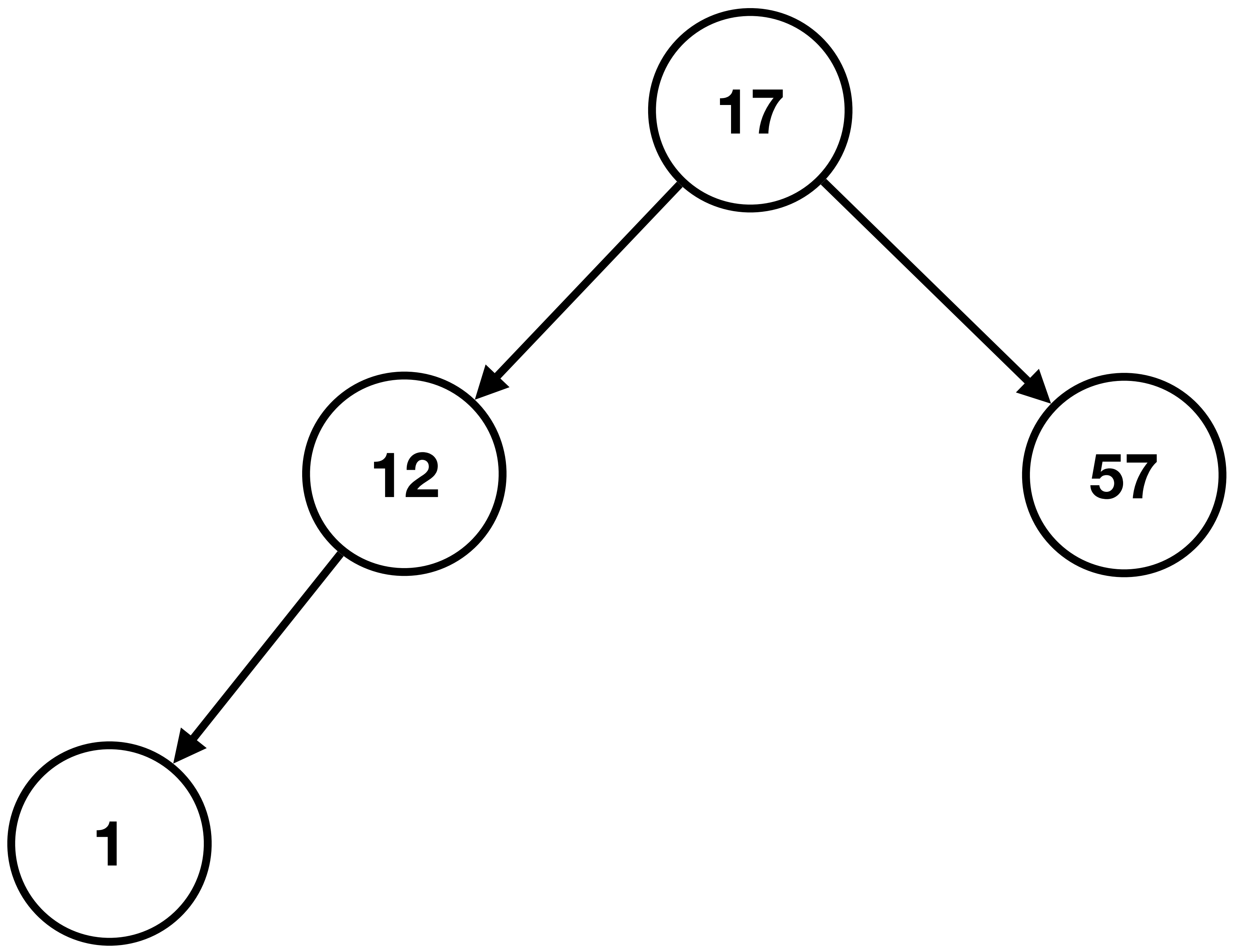
Height

- Insert: 17, 12, 57, 1, 16, 40, 84



BST
Height

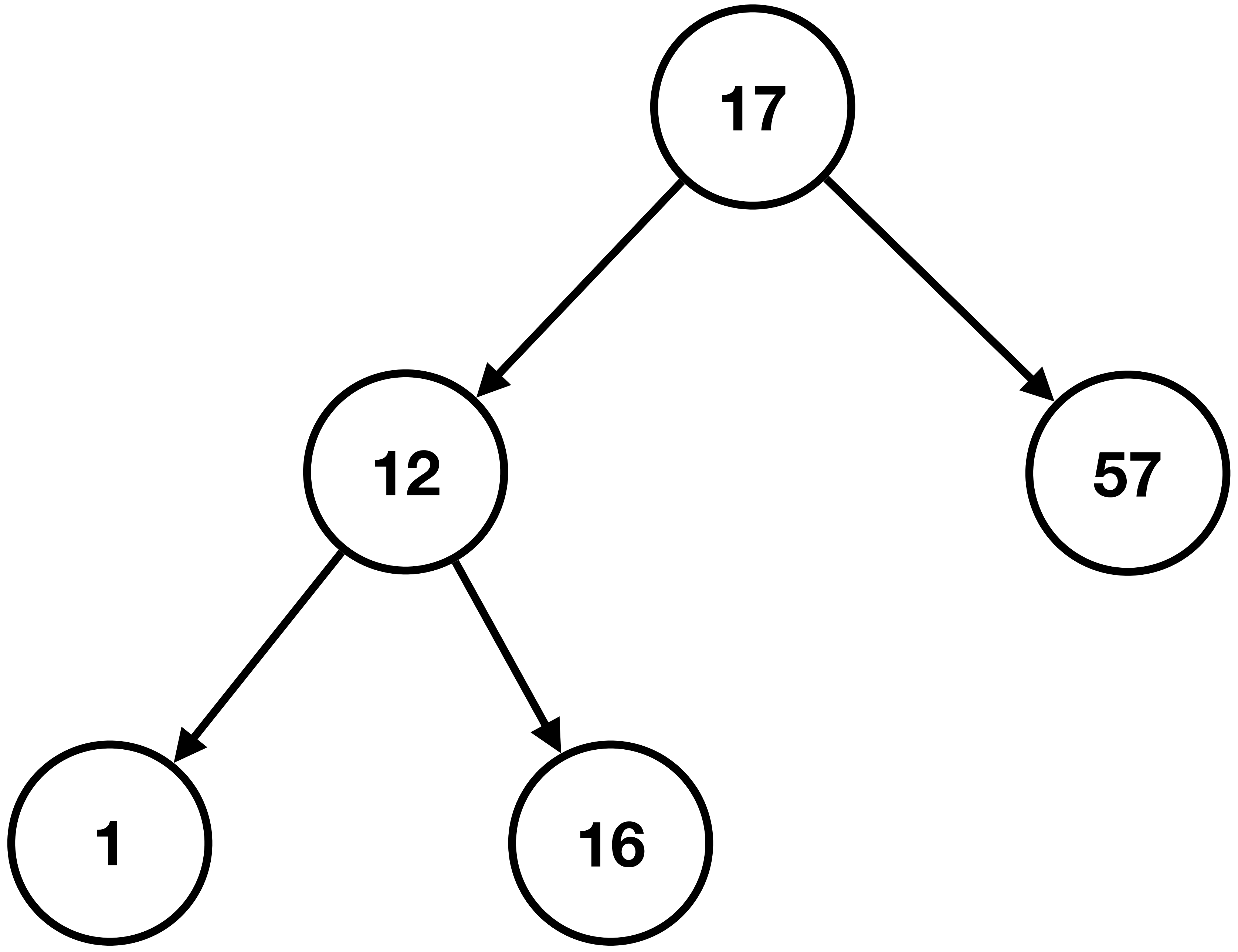
- Insert: 17, 12, 57, 1, 16, 40, 84



BST

Height

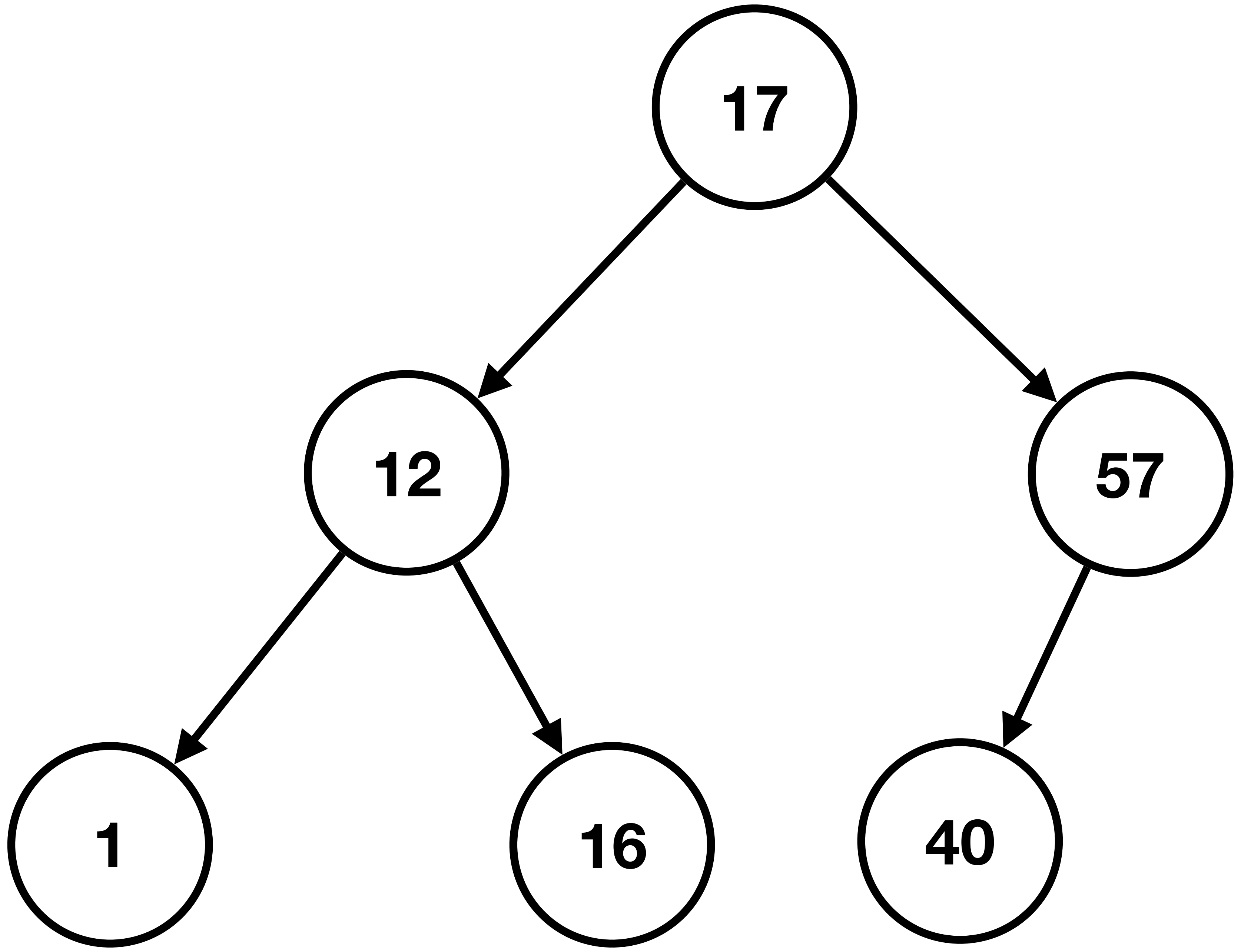
- Insert: 17, 12, 57, 1, 16, 40, 84



BST

Height

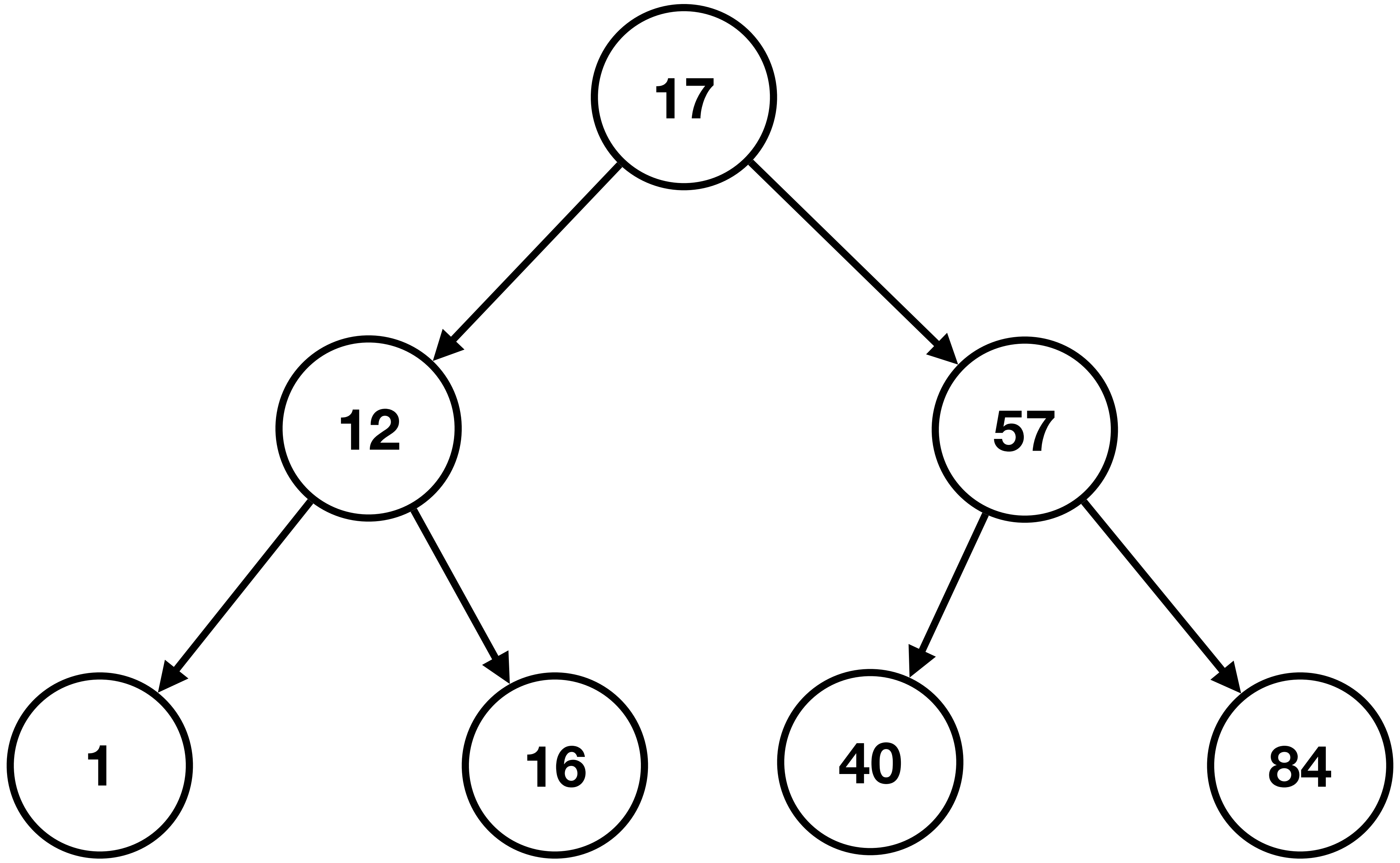
- Insert: 17, 12, 57, 1, 16, 40, 84



BST

Height

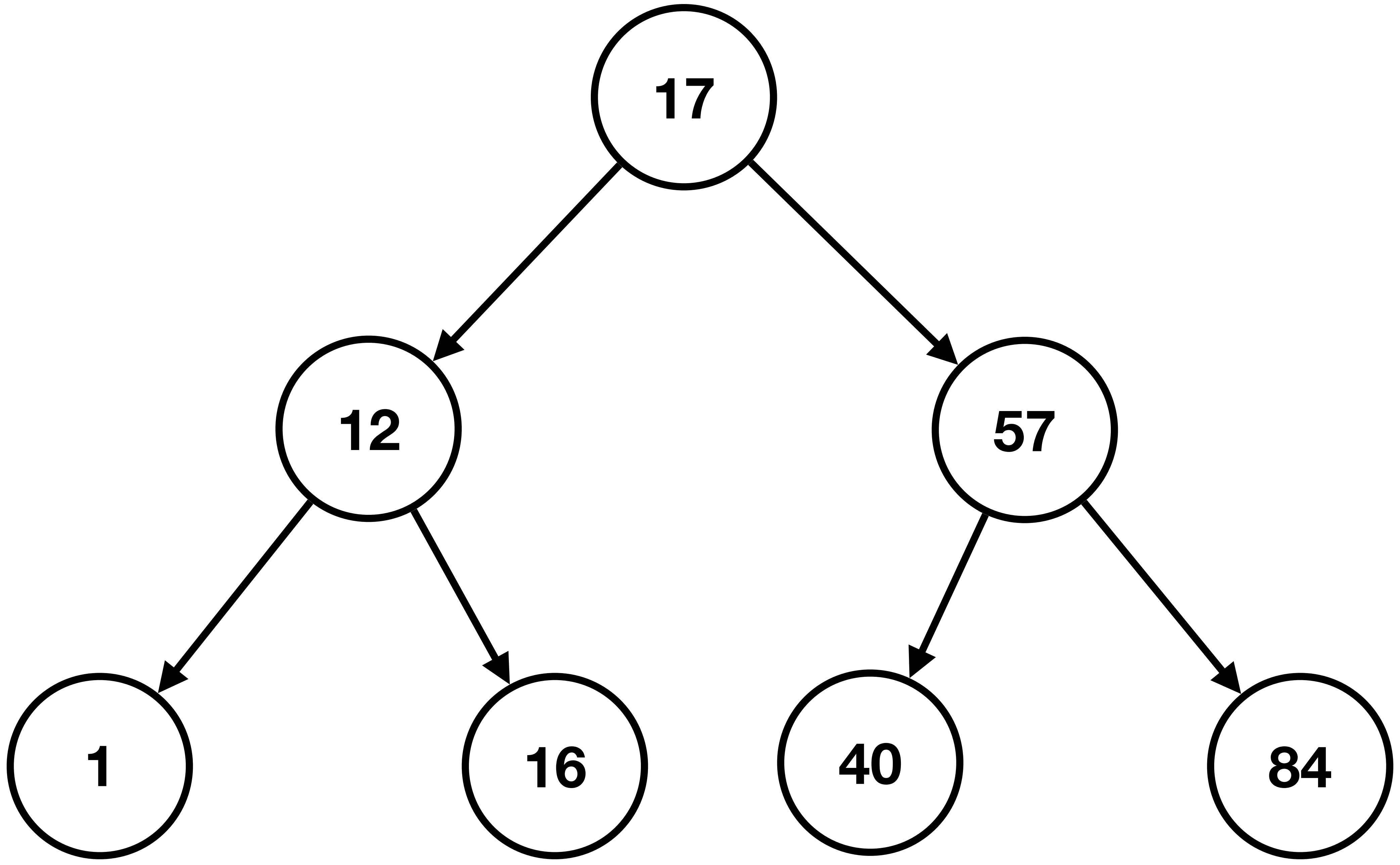
- Insert: 17, 12, 57, 1, 16, 40, 84



BST

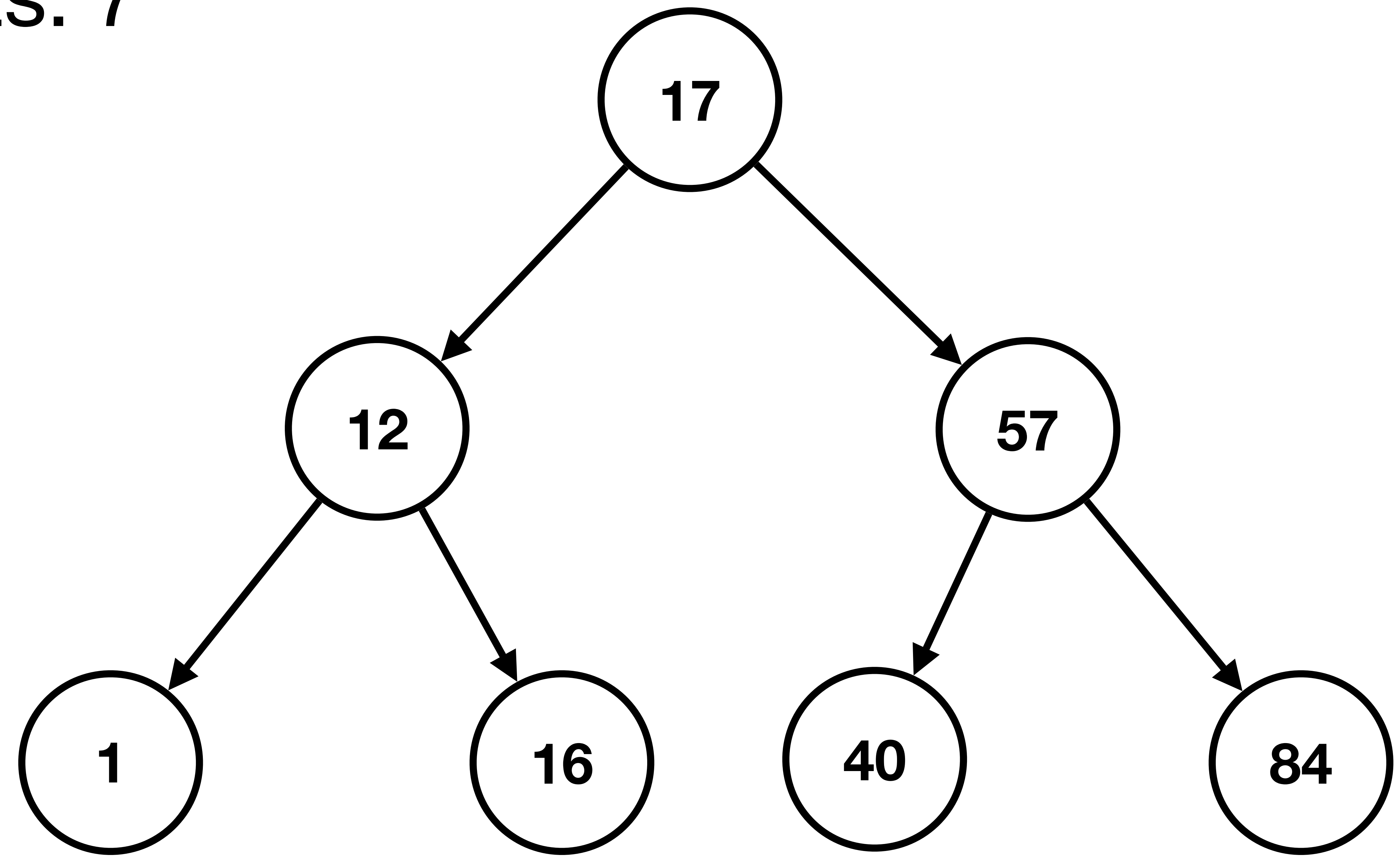
Height

- Insert: 17, 12, 57, 1, 16, 40, 84



BST
Height

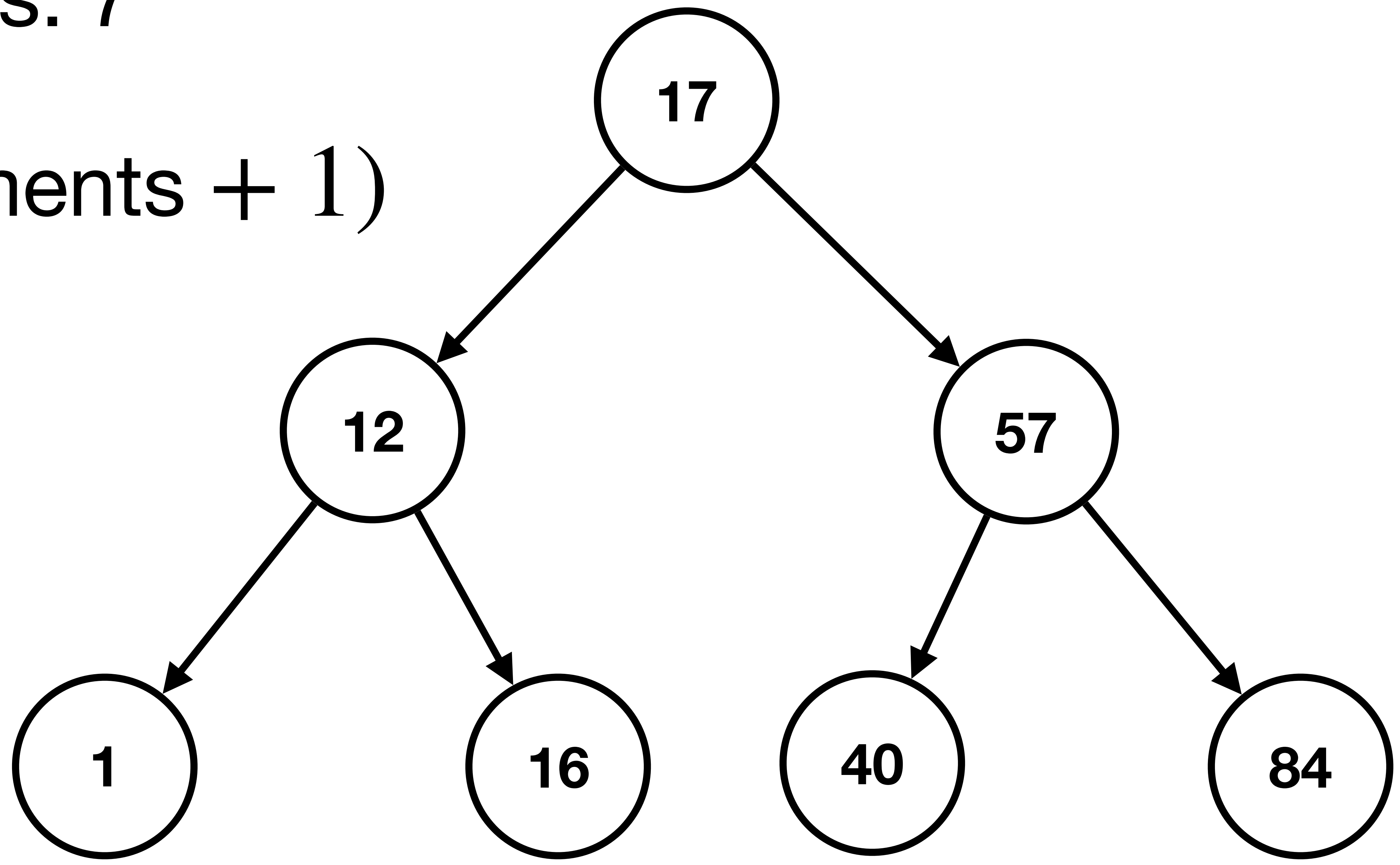
- Insert: 17, 12, 57, 1, 16, 40, 84
- Height: 3, #elements: 7



BST

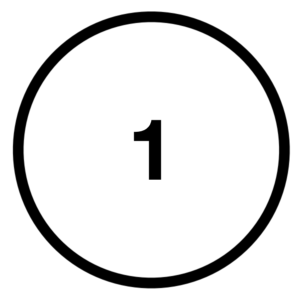
Height

- Insert: 17, 12, 57, 1, 16, 40, 84
- Height: 3, #elements: 7
- height = $\log_2(\#\text{elements} + 1)$



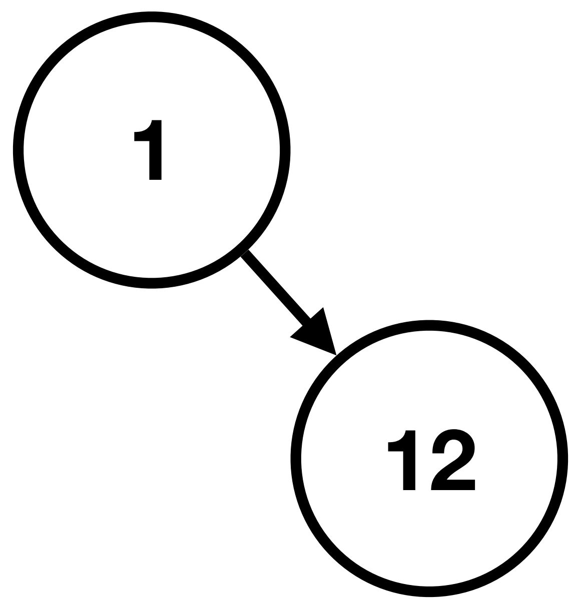
BST
Height

- Insert: 1, 12, 16, 17, 40, 57, 84



BST

Height

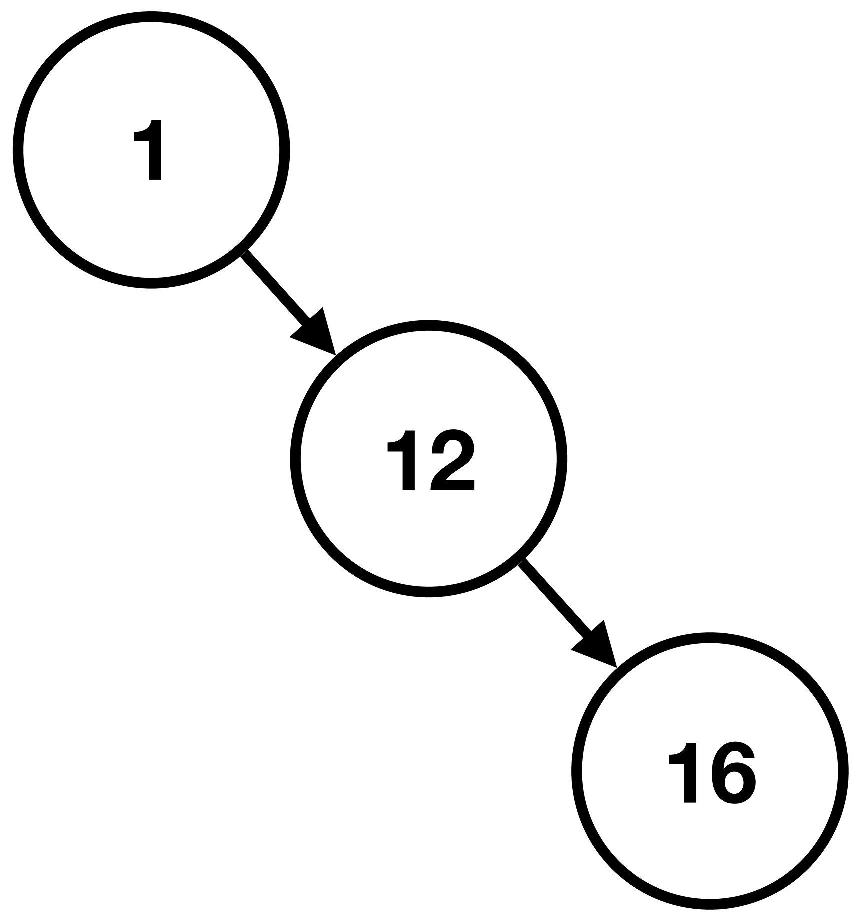


- Insert: 1, 12, 16, 17, 40, 57, 84

BST

Height

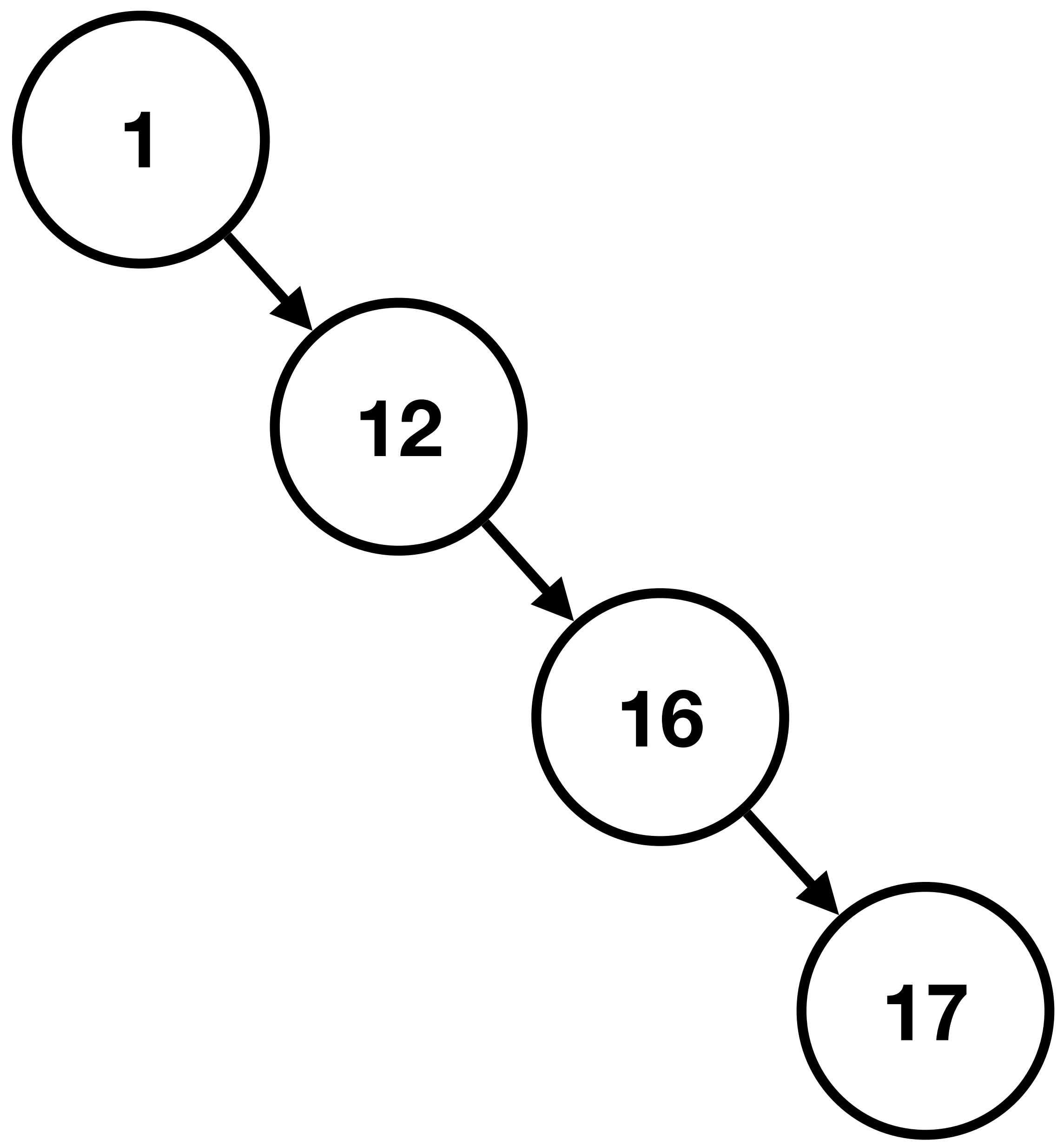
- Insert: 1, 12, 16, 17, 40, 57, 84



BST

Height

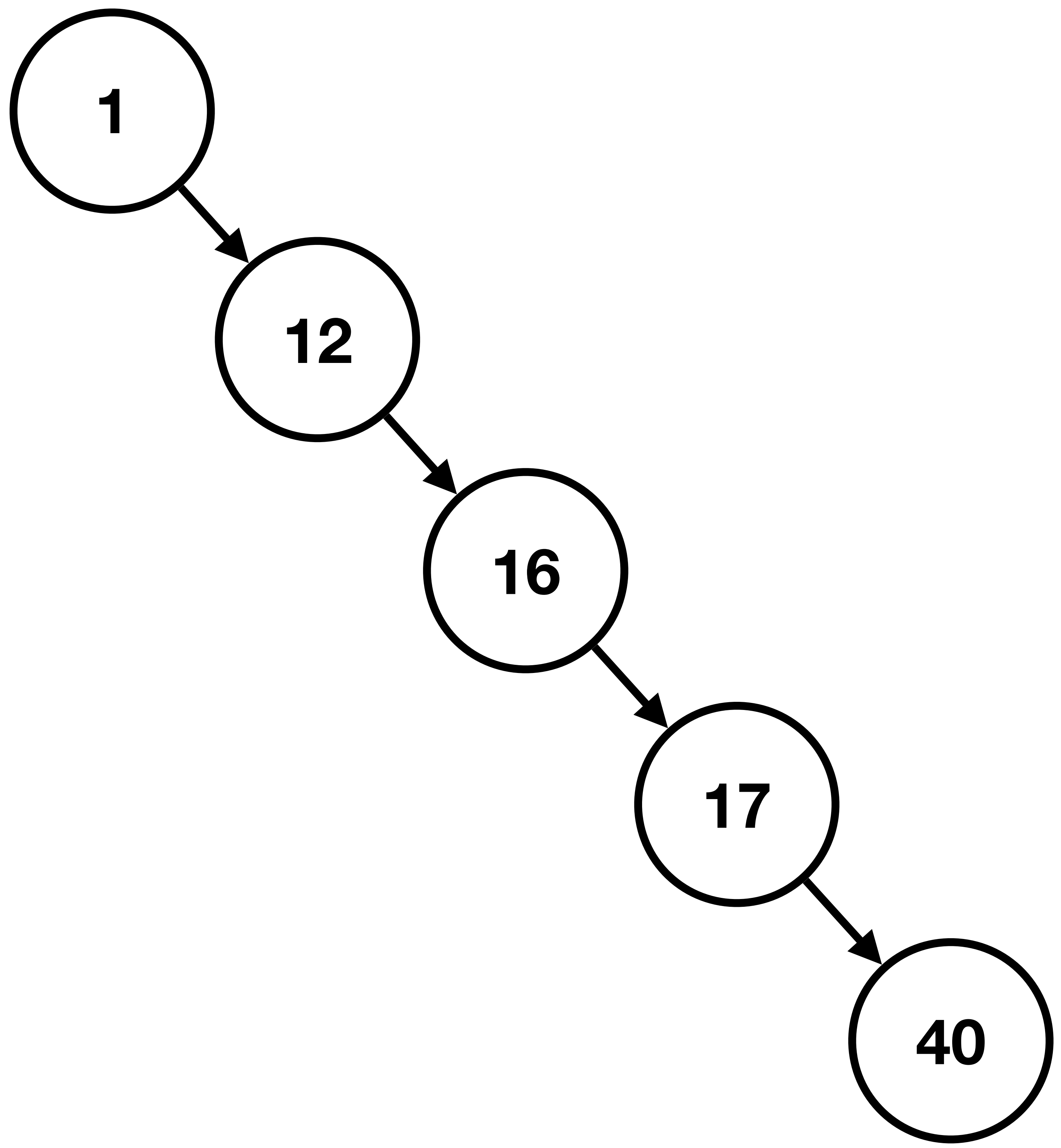
- Insert: 1, 12, 16, 17, 40, 57, 84



BST

Height

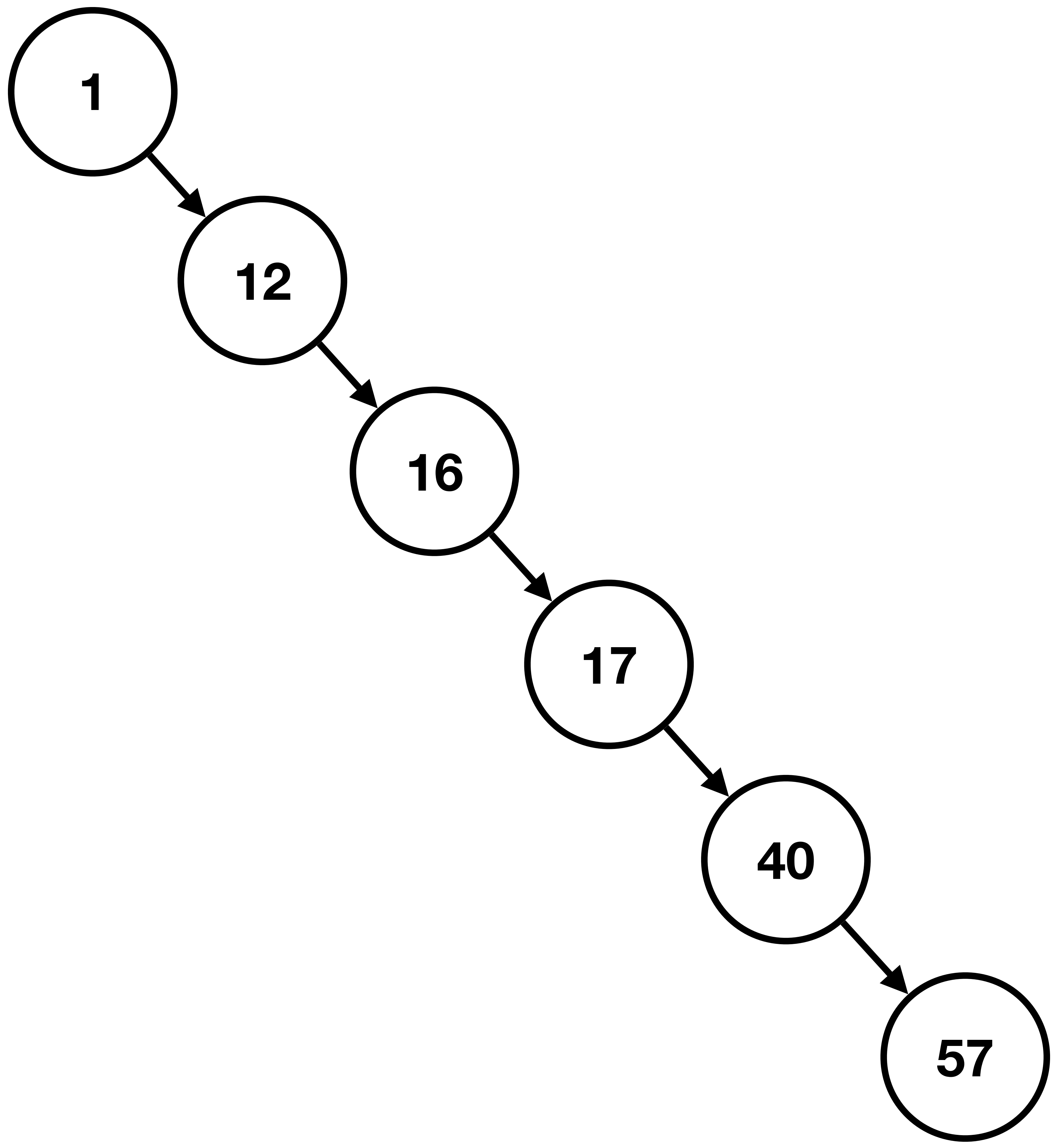
- Insert: 1, 12, 16, 17, 40, 57, 84



BST

Height

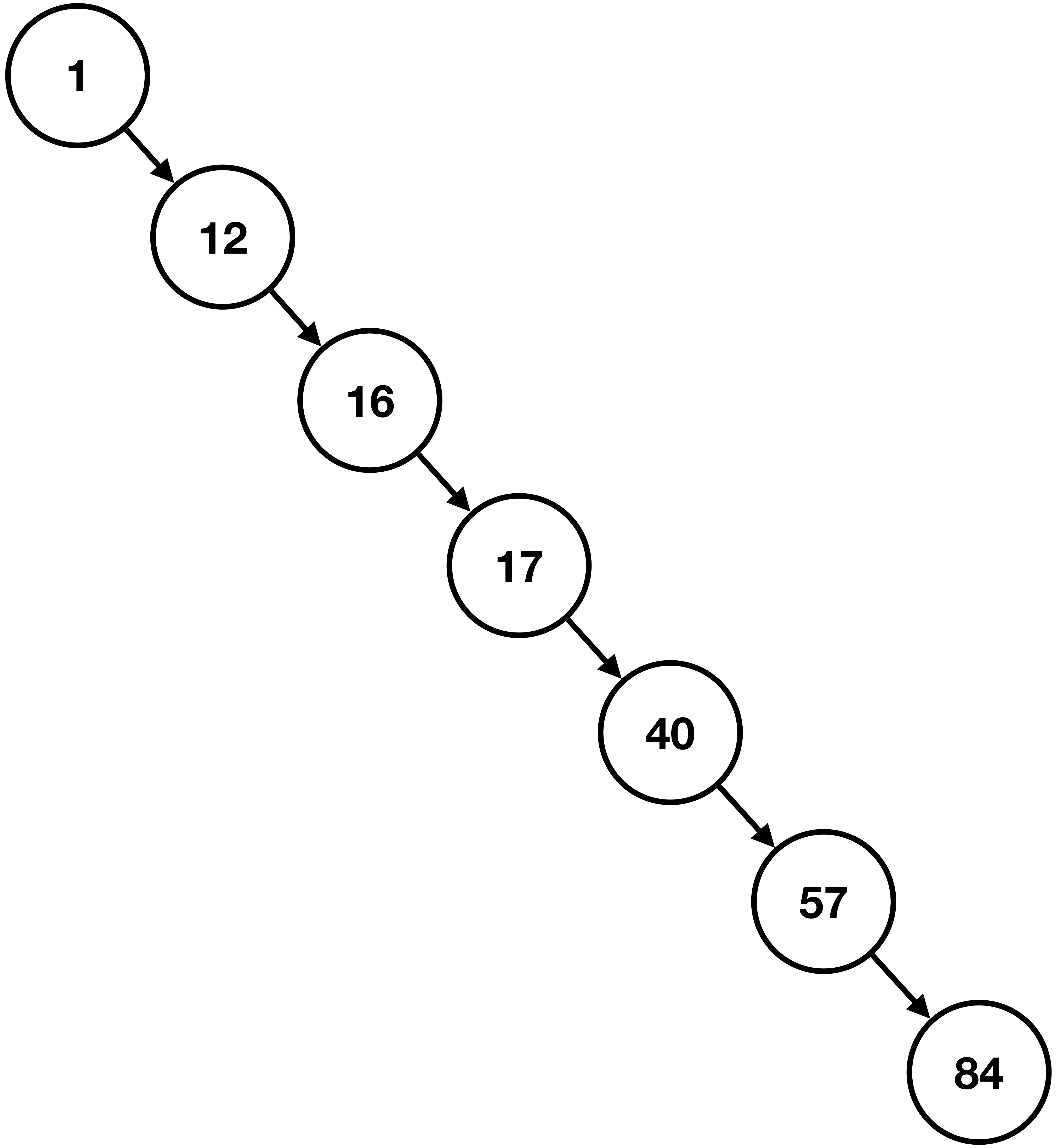
- Insert: 1, 12, 16, 17, 40, 57, 84



BST

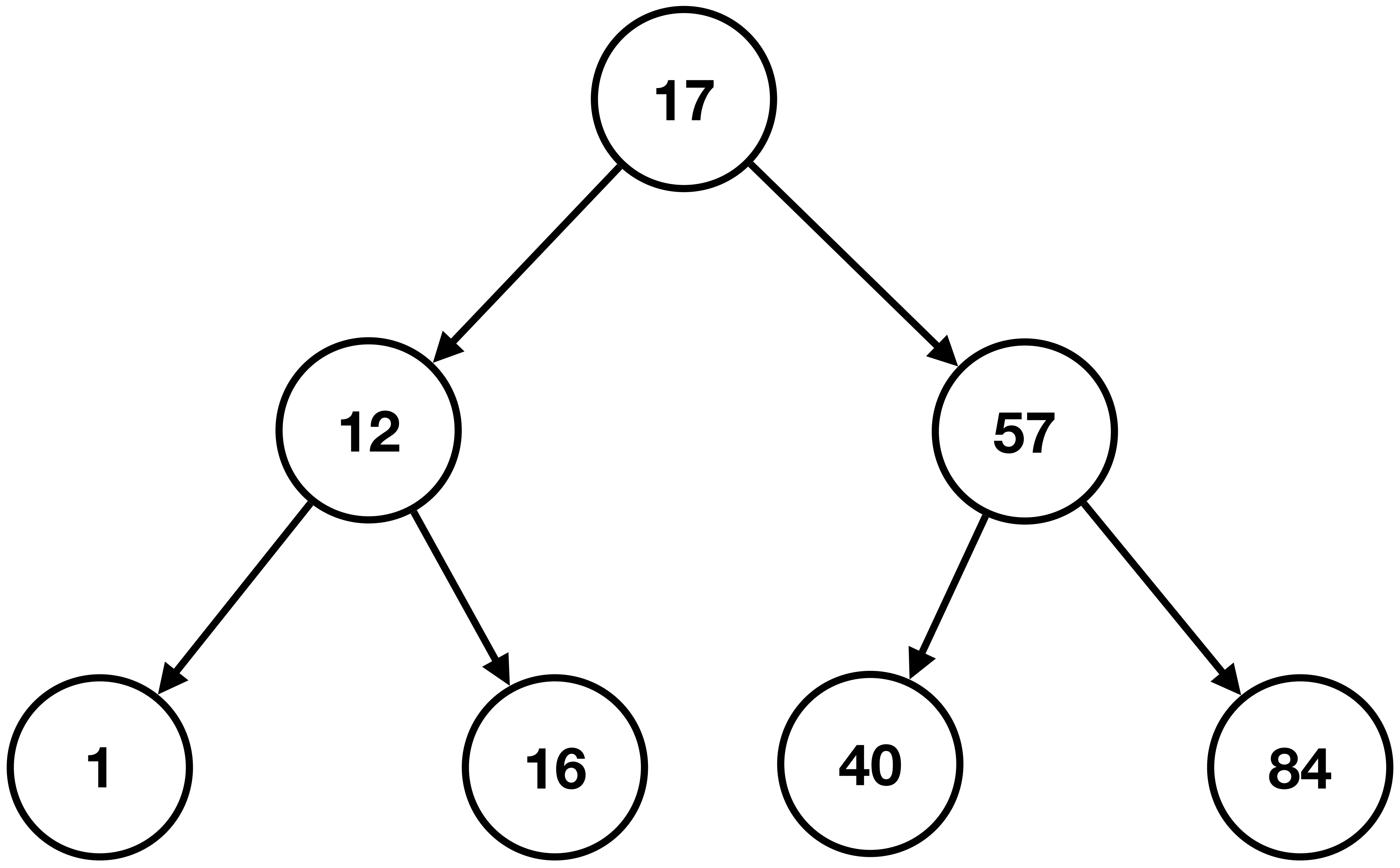
Height

- Insert: 1, 12, 16, 17, 40, 57, 84

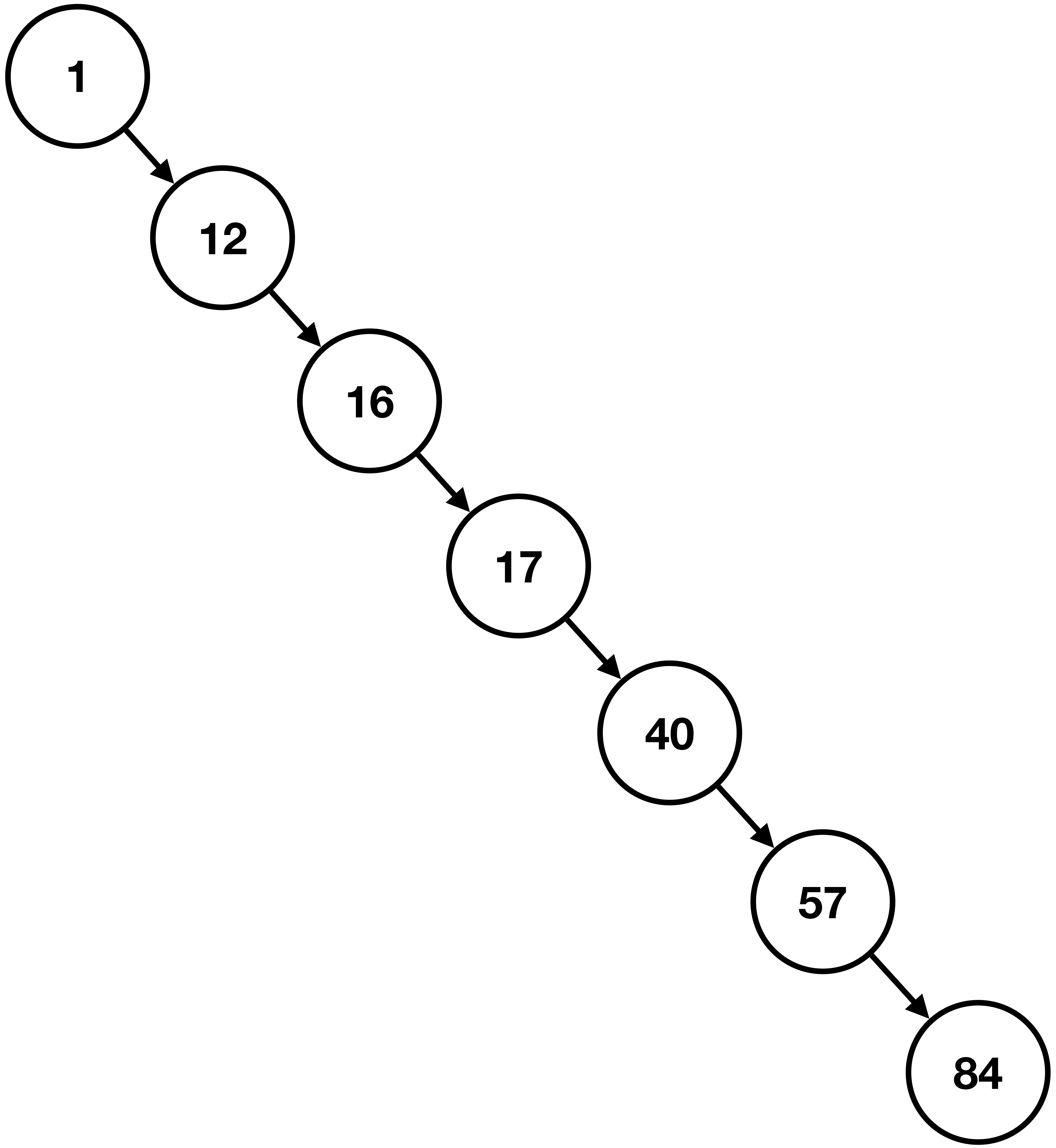


BST

Height



Balanced



Unbalanced

BST

Complexity

BST

Complexity

- lookup, insert:

BST

Complexity

- lookup, insert:
  - $O(\log n)$ for a well-balanced BST

BST

Complexity

- lookup, insert:
  - $O(\log n)$ for a well-balanced BST
  - $O(n)$ in general :(

BST

Complexity

- lookup, insert:
  - $O(\log n)$ for a well-balanced BST
  - $O(n)$ in general :(
- There are self-balancing BSTs

BST

Complexity

- lookup, insert:
  - $O(\log n)$ for a well-balanced BST
  - $O(n)$ in general :(
- There are self-balancing BSTs
  - Red-black trees, AVL trees, ...

BST

Remove

BST

Remove

- First, find node to remove

BST

Remove

- First, find node to remove
  - same in lookup and insert

BST

Remove

- First, find node to remove
  - same in lookup and insert
- Easy case: the node is a leaf

BST

Remove

- First, find node to remove
  - same in lookup and insert
- Easy case: the node is a leaf
  - Delete it

BST

Remove

- First, find node to remove
  - same in lookup and insert
- Easy case: the node is a leaf
  - Delete it
  - Don't forget to update the parent's pointer

BST

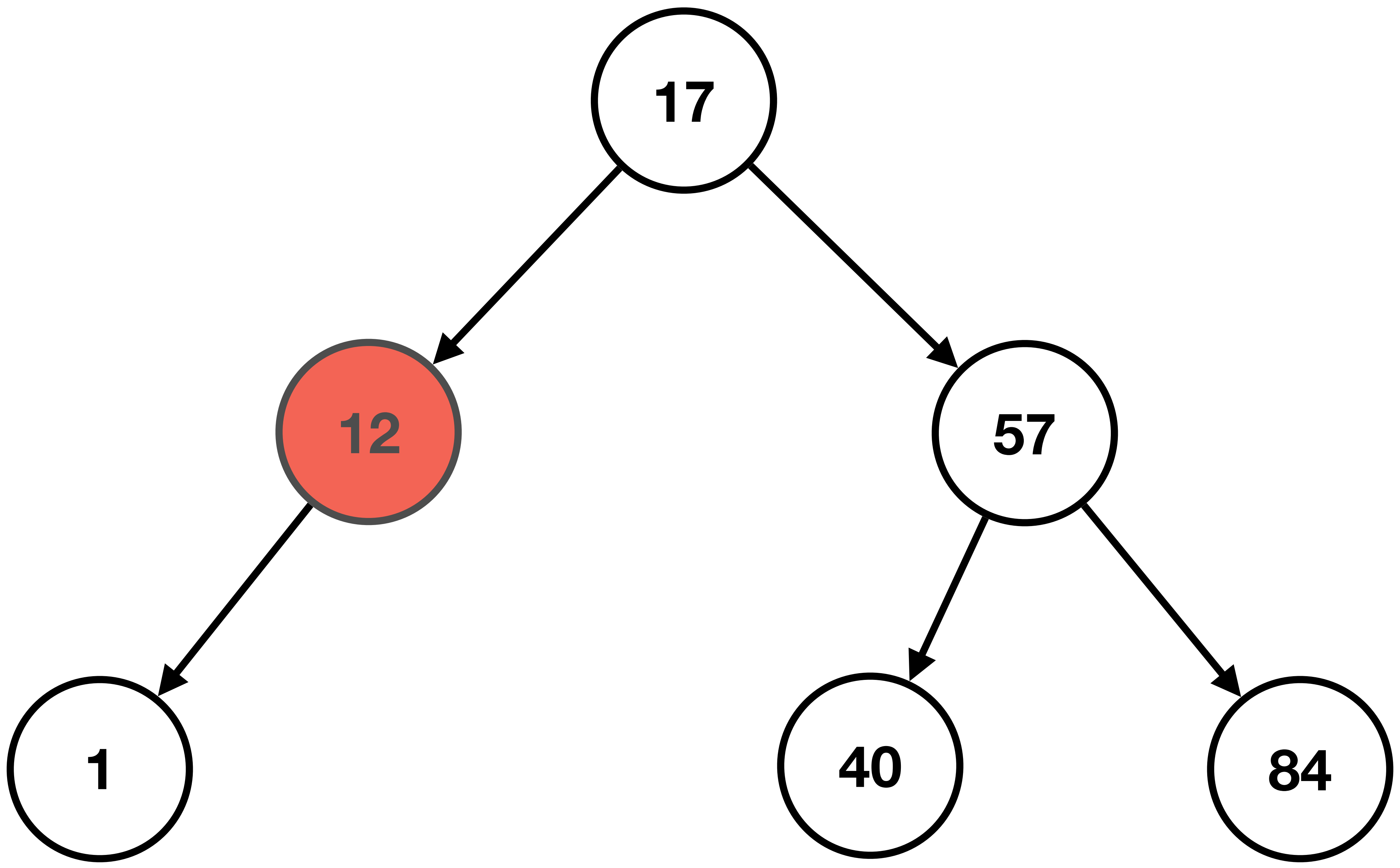
Remove

- Harder case: node to be removed has one child

BST

Remove

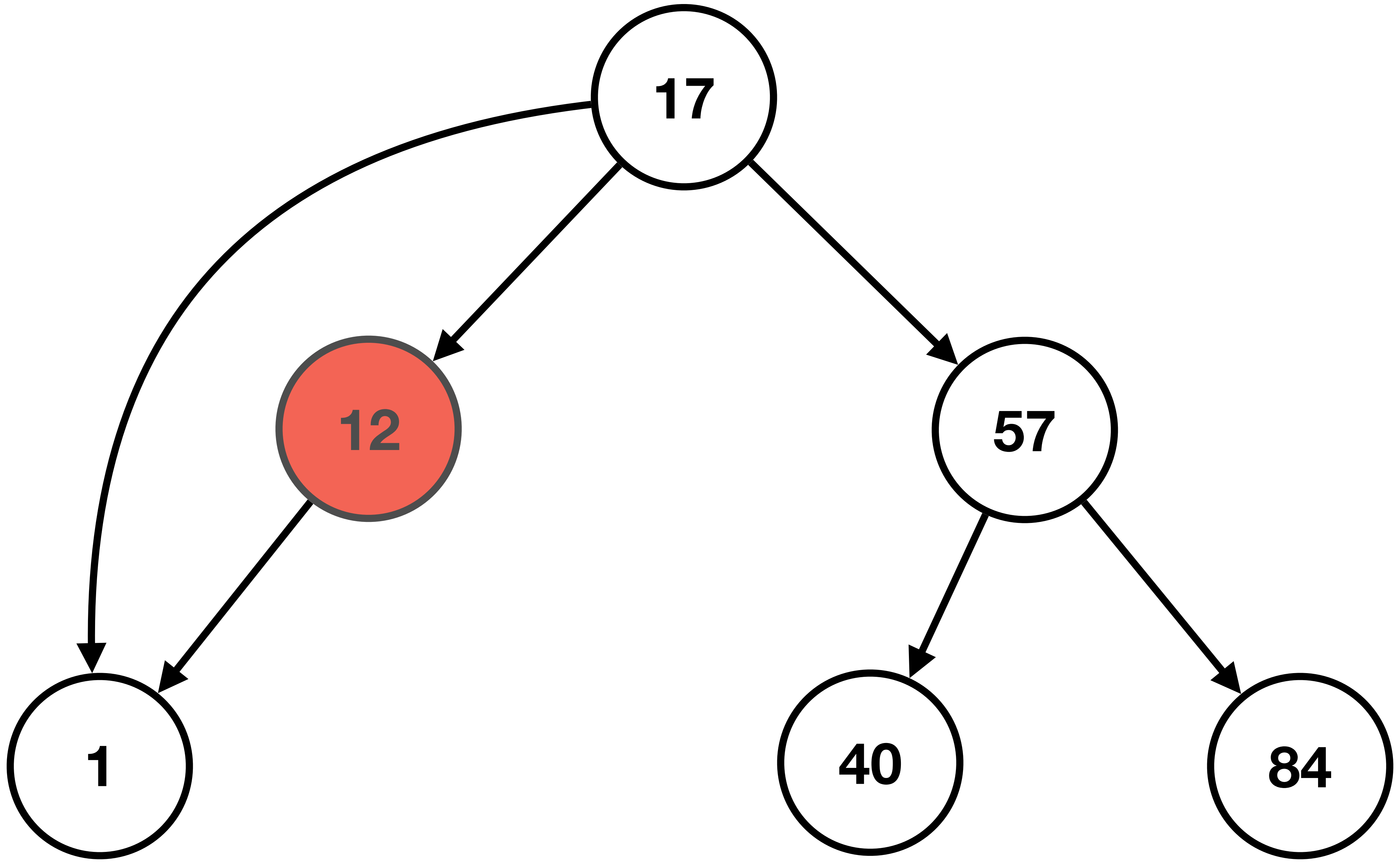
- Harder case: node to be removed has one child



BST

Remove

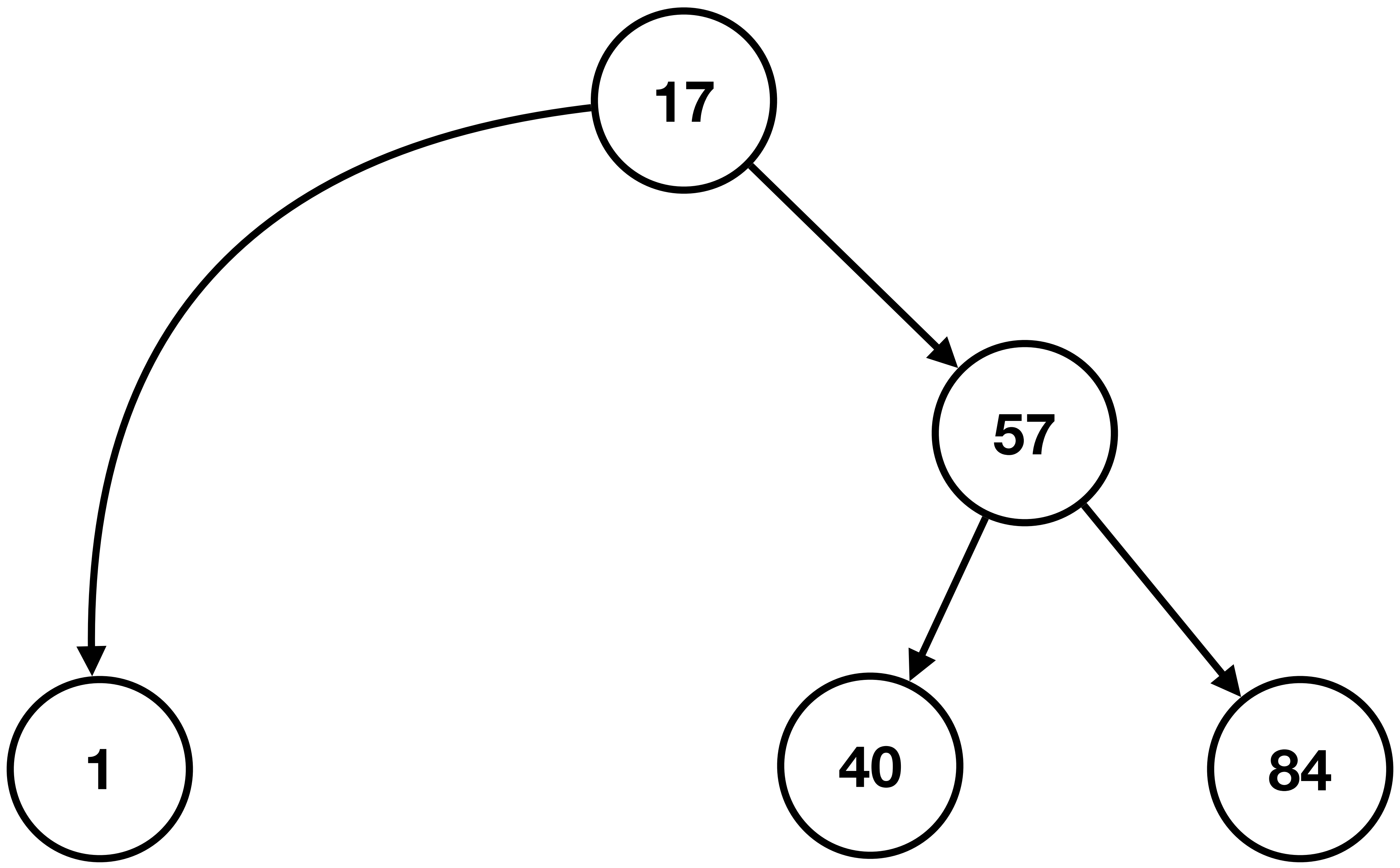
- Harder case: node to be removed has one child
  - Bypass this node



BST

Remove

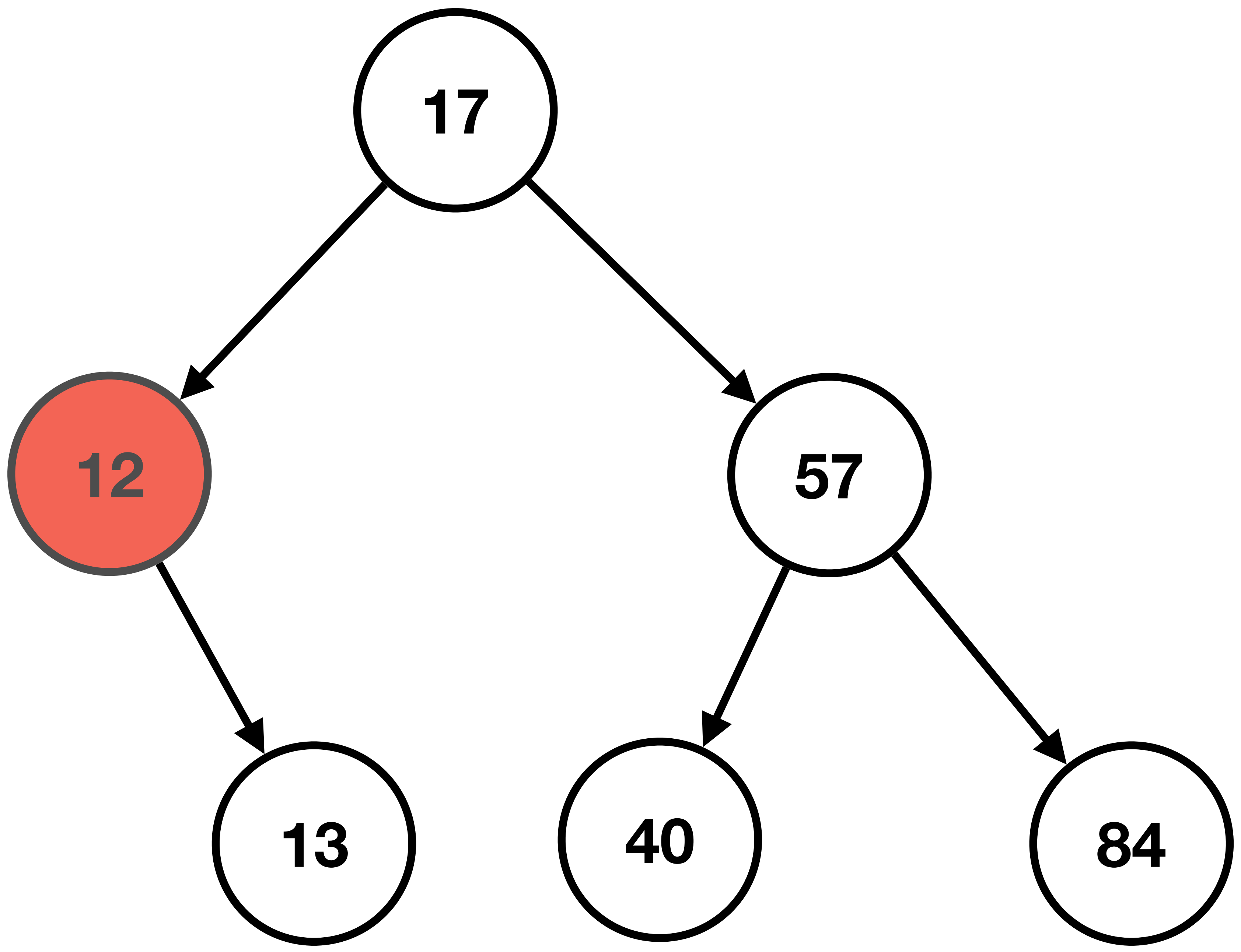
- Harder case: node to be removed has one child
  - Bypass this node



BST

Remove

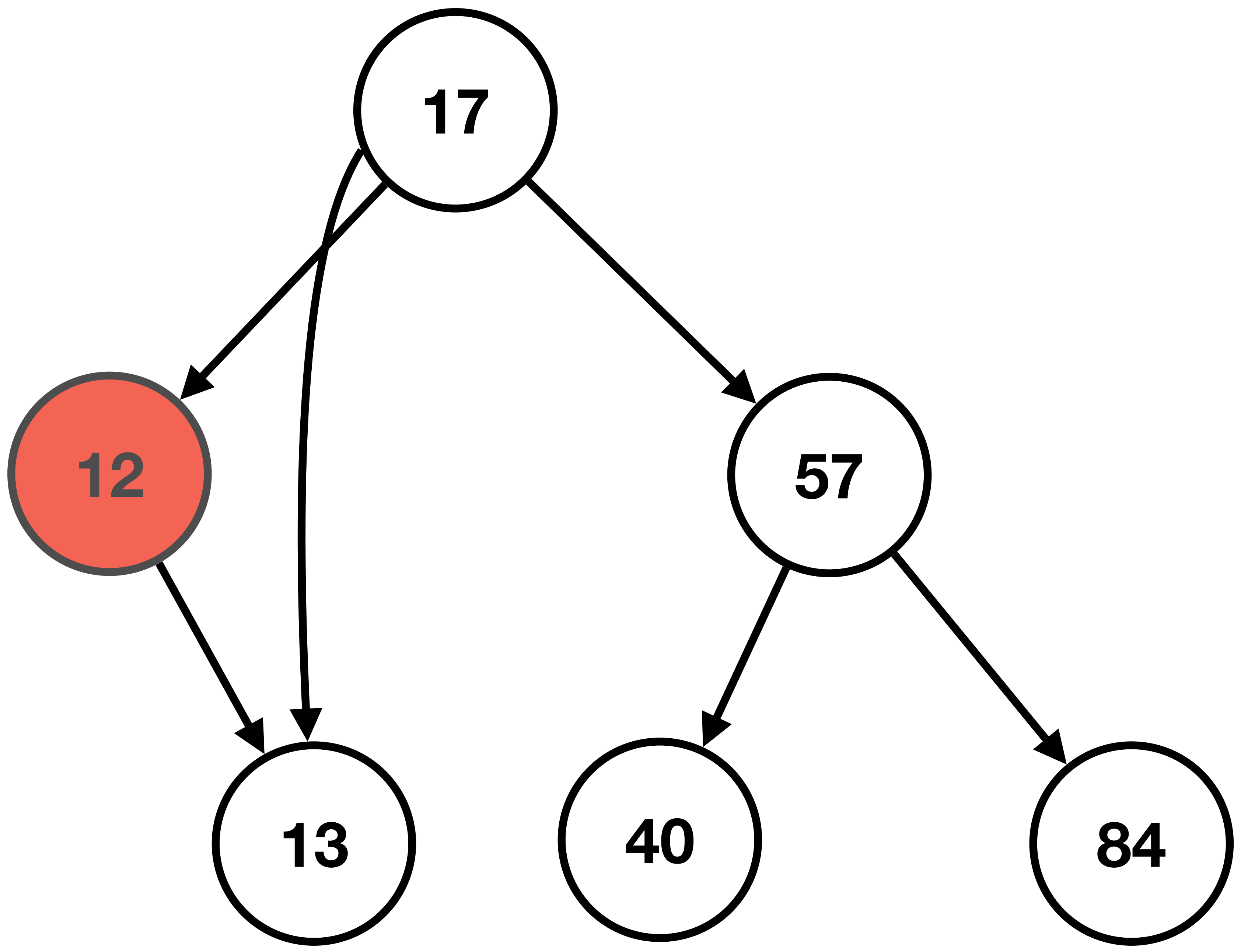
- Harder case: node to be removed has one child
  - Bypass this node



BST

Remove

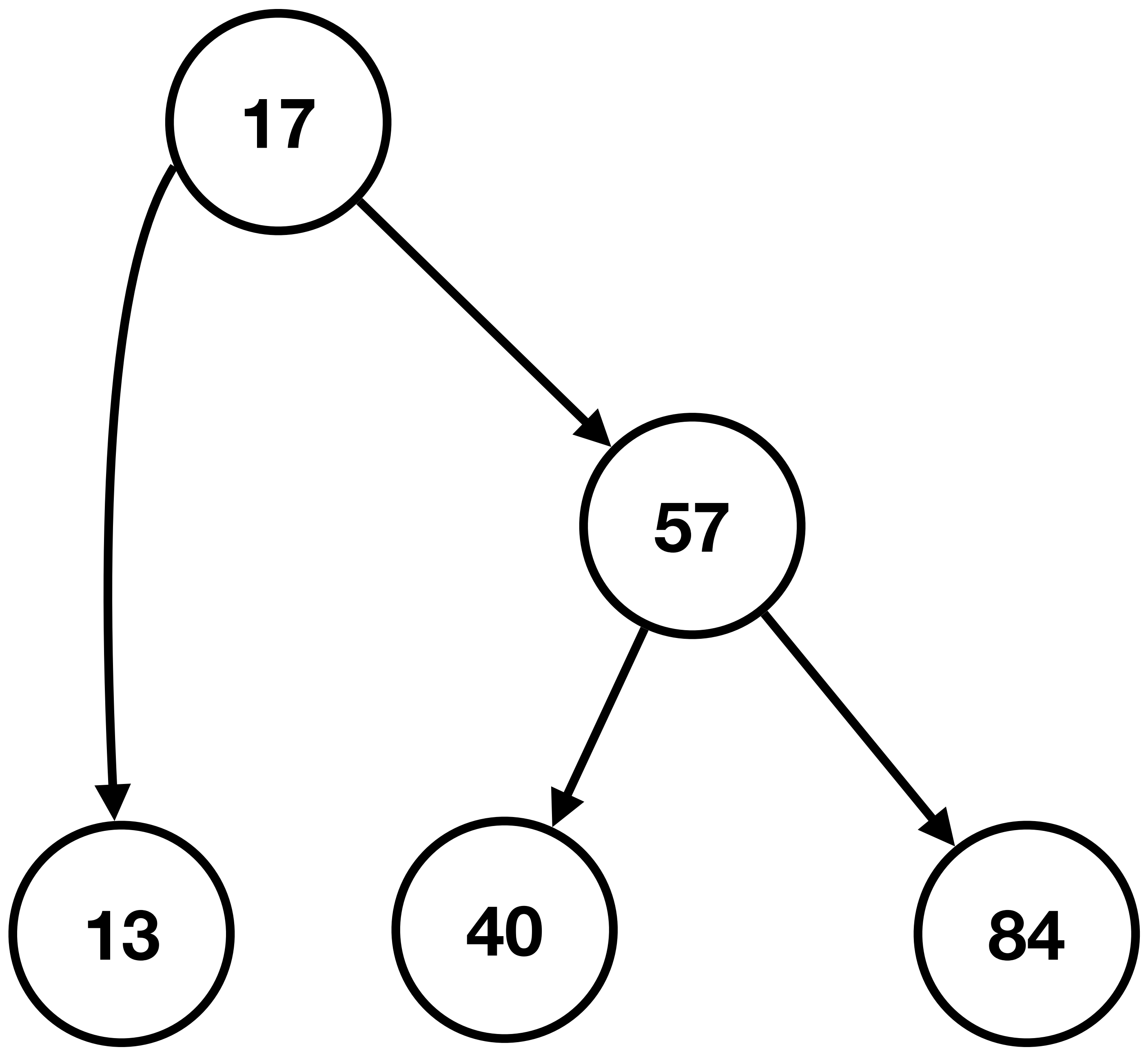
- Harder case: node to be removed has one child
  - Bypass this node



BST

Remove

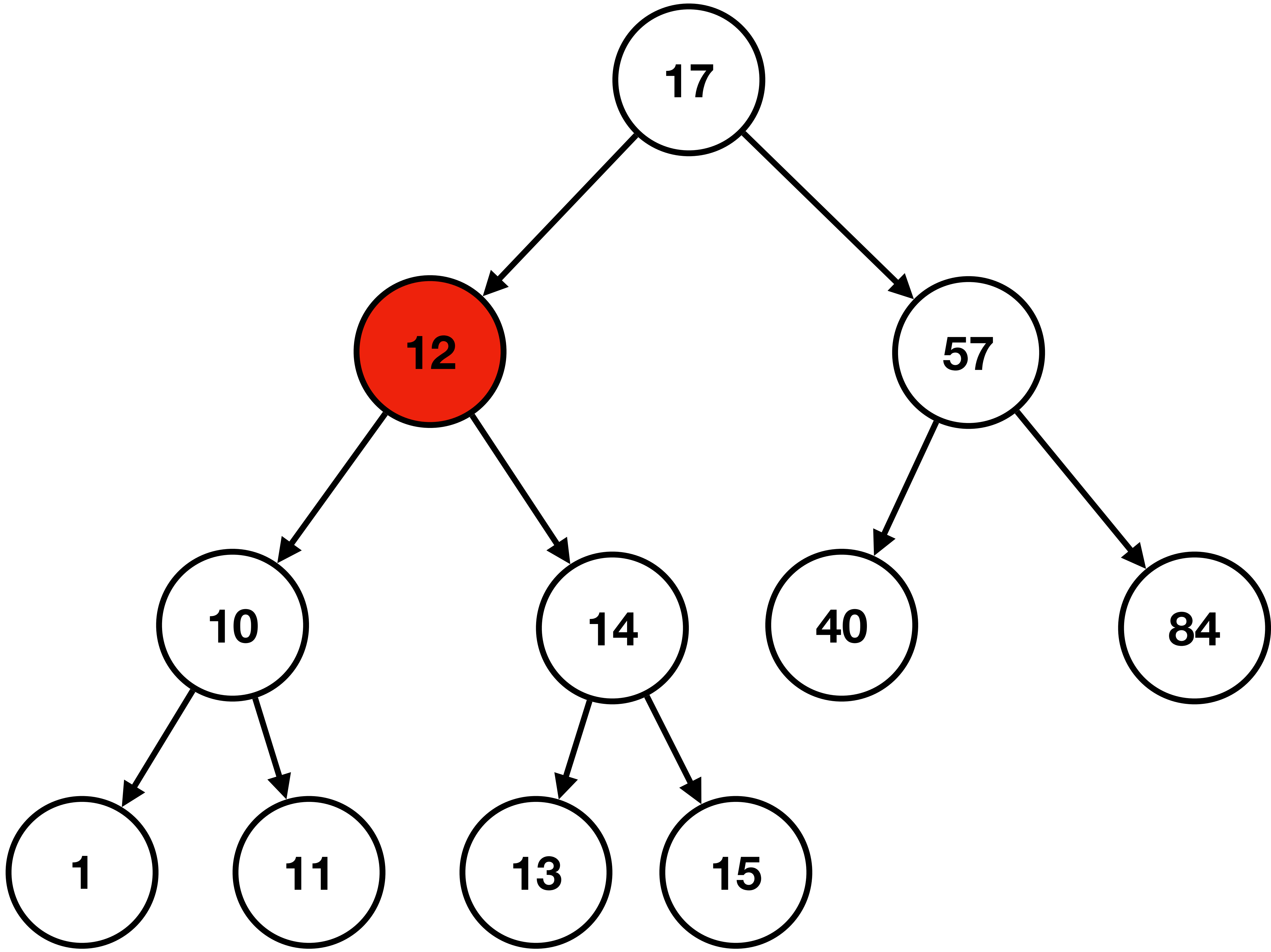
- Harder case: node to be removed has one child
  - Bypass this node



BST

Remove

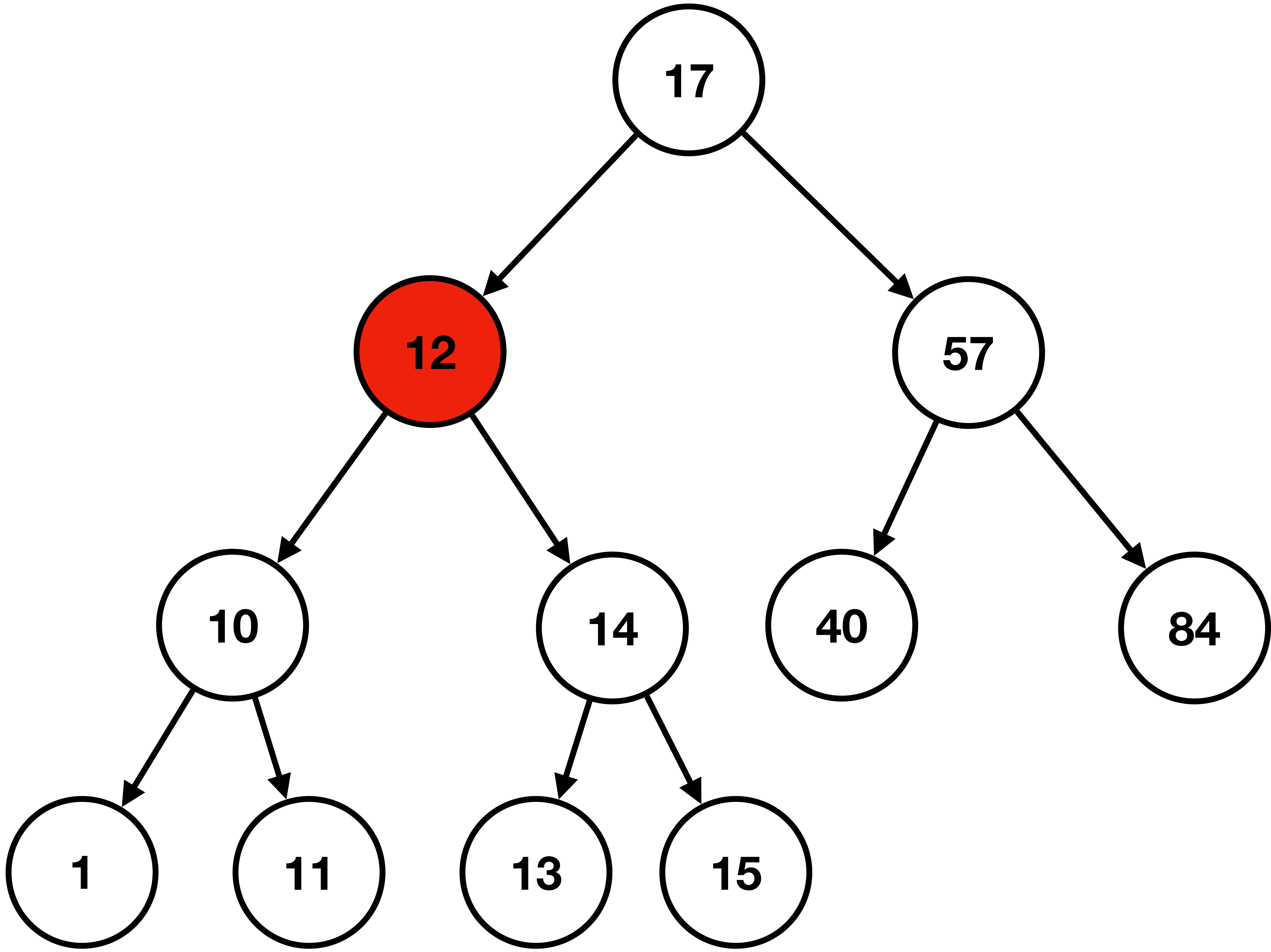
- Hardest case: node to be removed has two children



BST

Remove

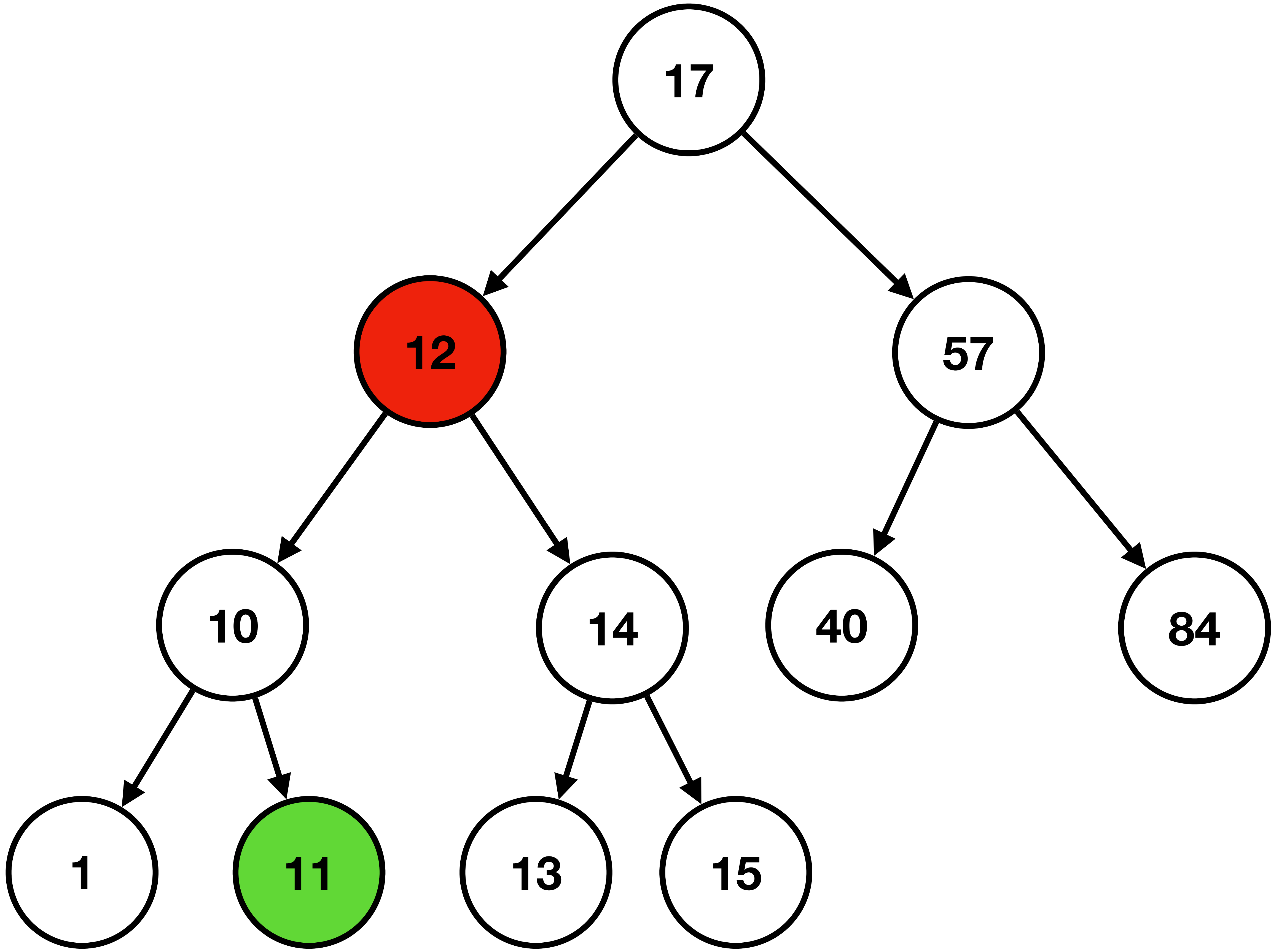
- Hardest case: node to be removed has two children
    - Replace 12 with a value that's:
        - Larger than everything in left subtree
        - Smaller than .. in right ..



BST

Remove

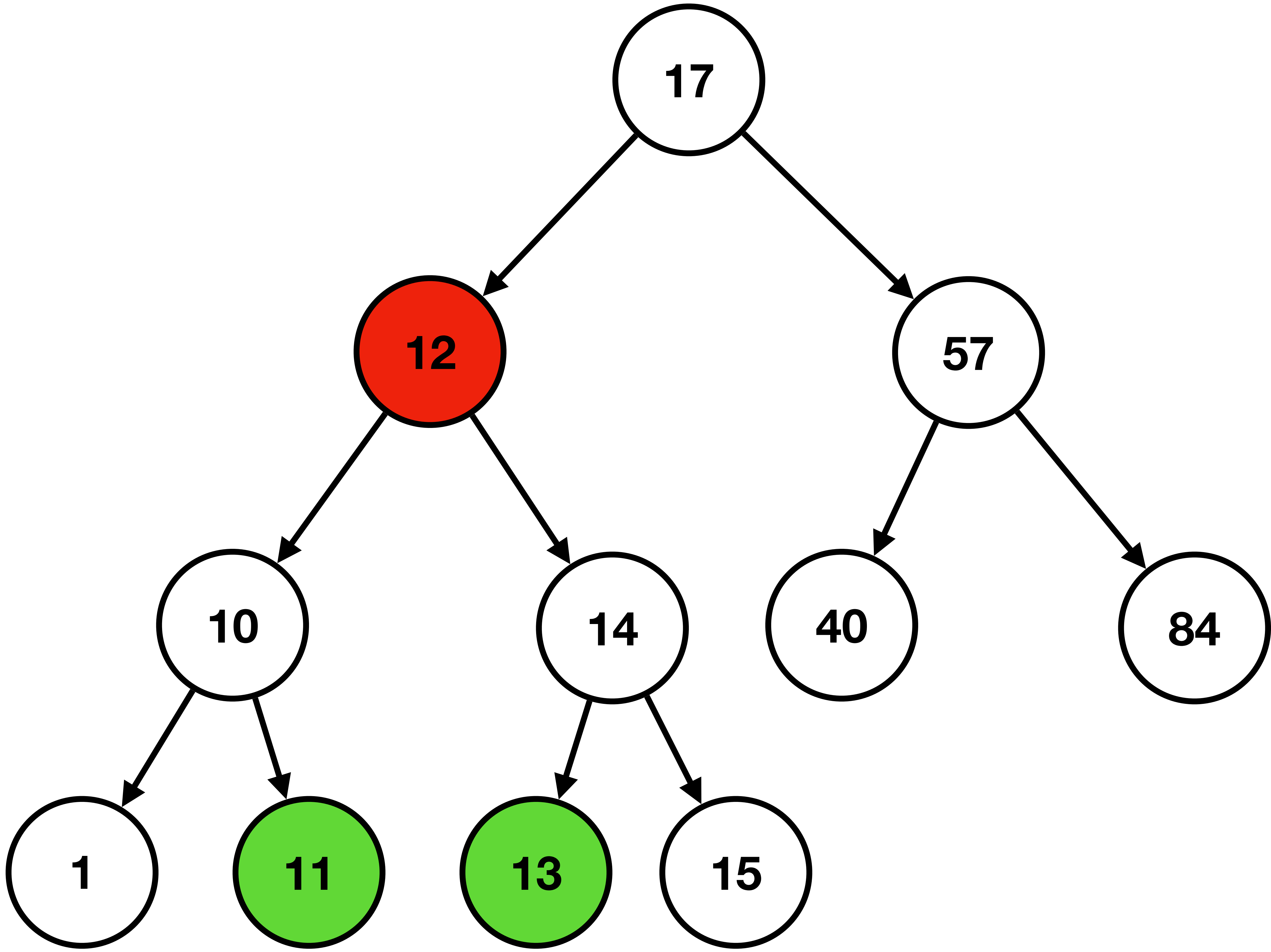
- Hardest case: node to be removed has two children
  - Replace 12 with a value that's:
    - Larger than everything in left subtree
    - Smaller than .. in right ..



BST

Remove

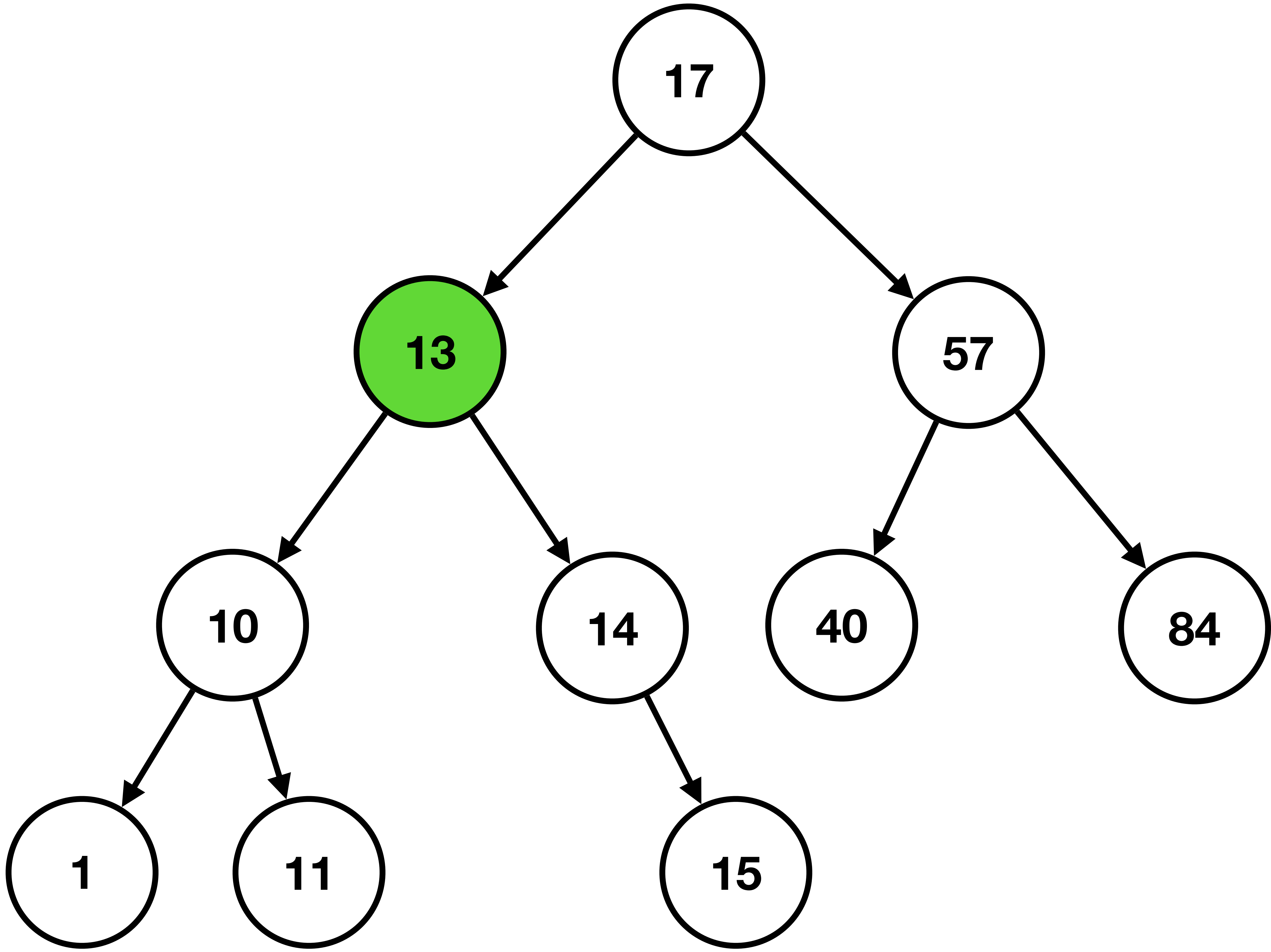
- Hardest case: node to be removed has two child
  - Replace 12 with a value that's:
    - Larger than everything in left subtree
    - Smaller than .. in right ..



BST

Remove

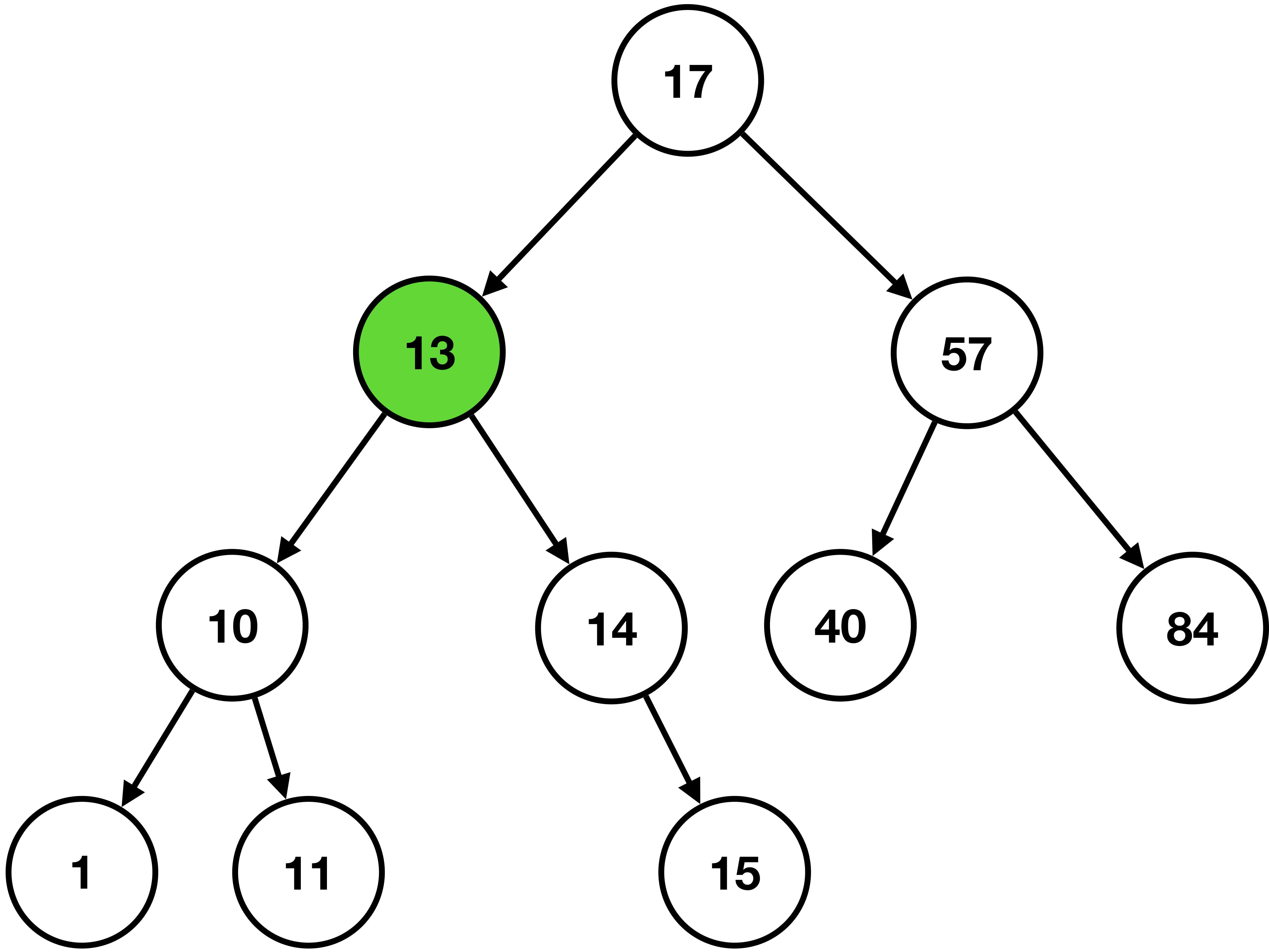
- Hardest case: node to be removed has two child
  - Replace 12 with a value that's:
    - Larger than everything in left subtree
    - Smaller than .. in right ..



BST

Remove

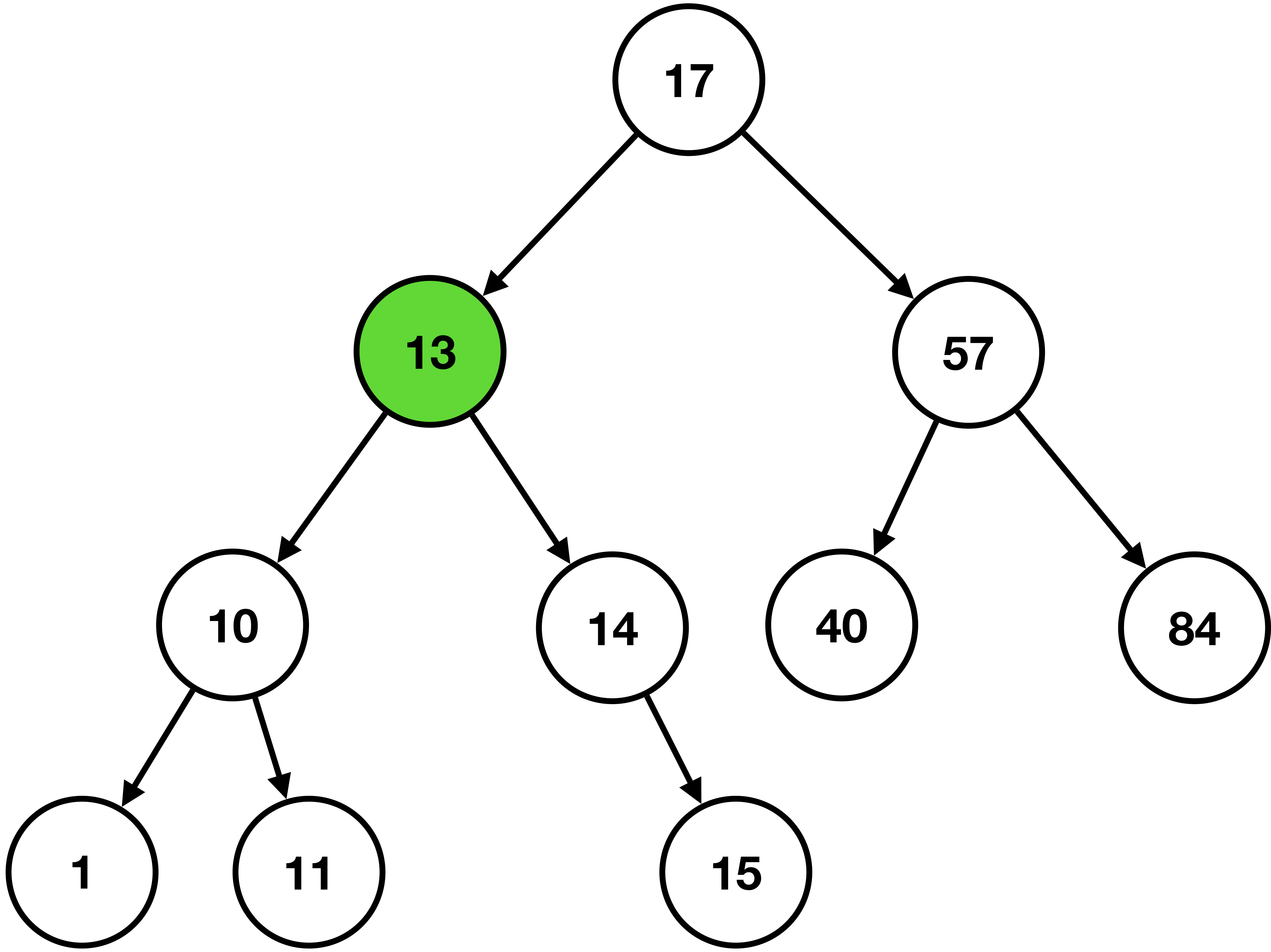
- Hardest case: node to be removed has two children



BST

Remove

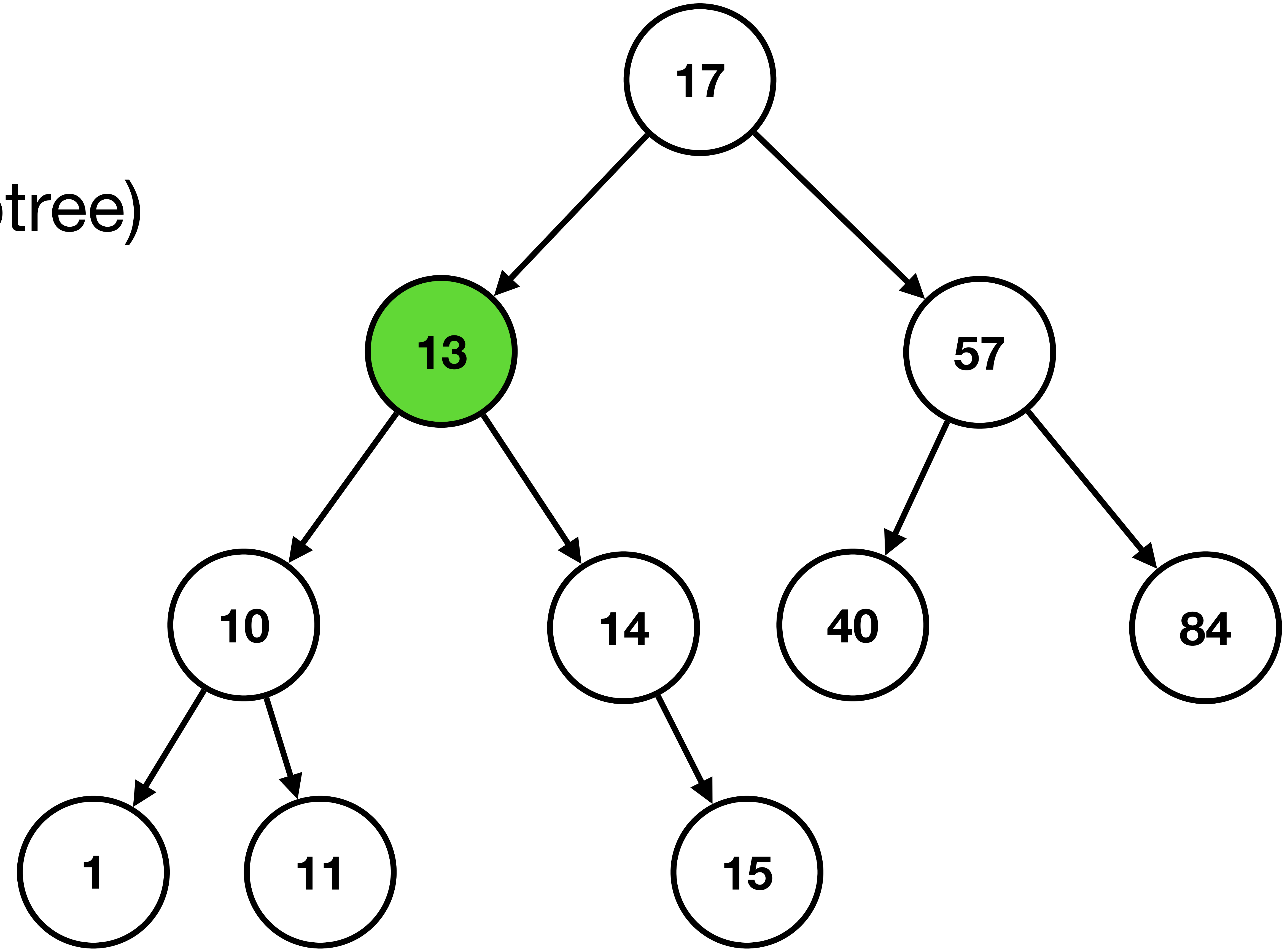
- Hardest case: node to be removed has two children
  - Find min(right substree), replace



BST

Remove

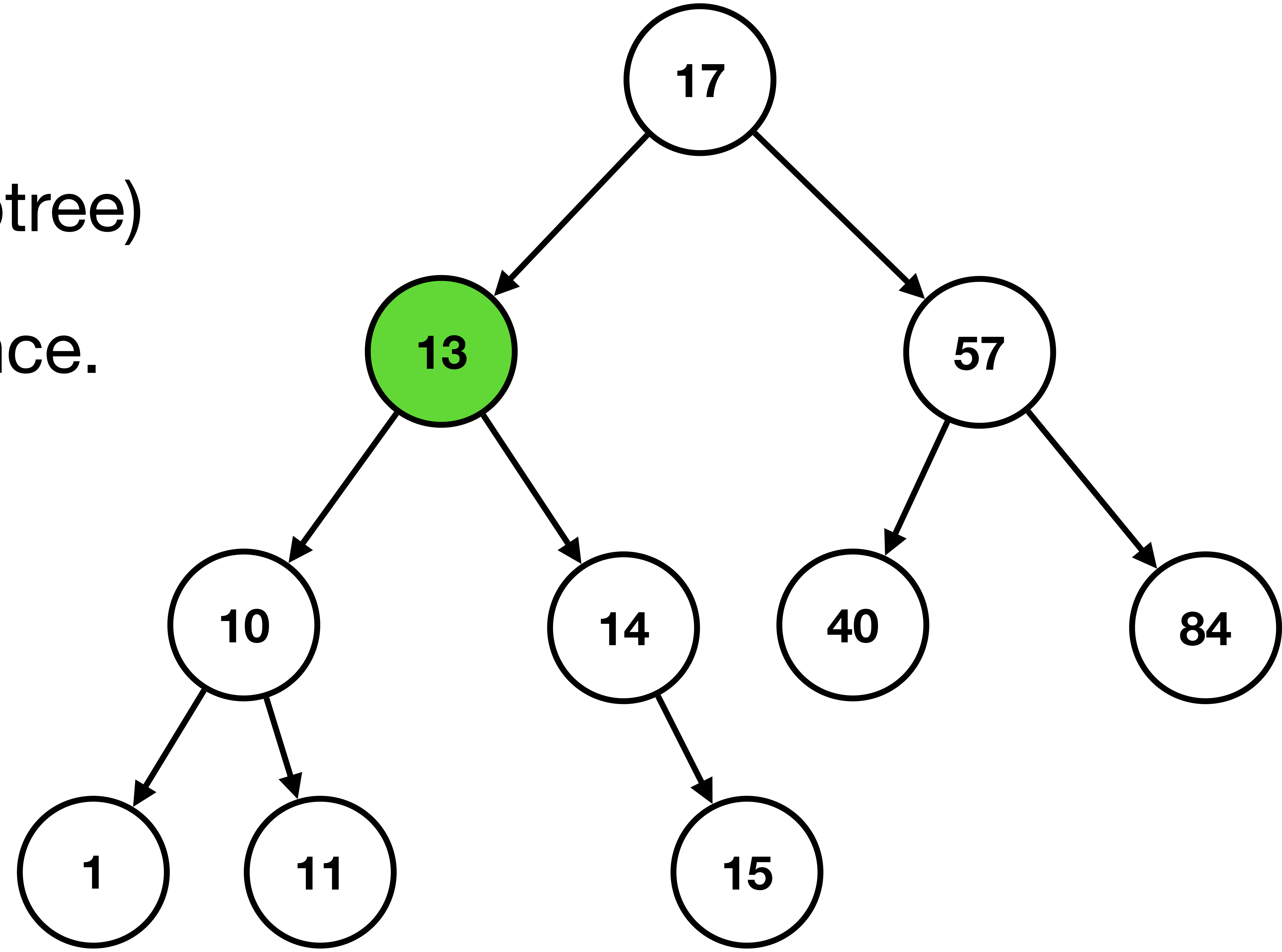
- Hardest case: node to be removed has two children
  - Find min(right substree), replace
  - Call remove recursively on min(right subtree)



BST

Remove

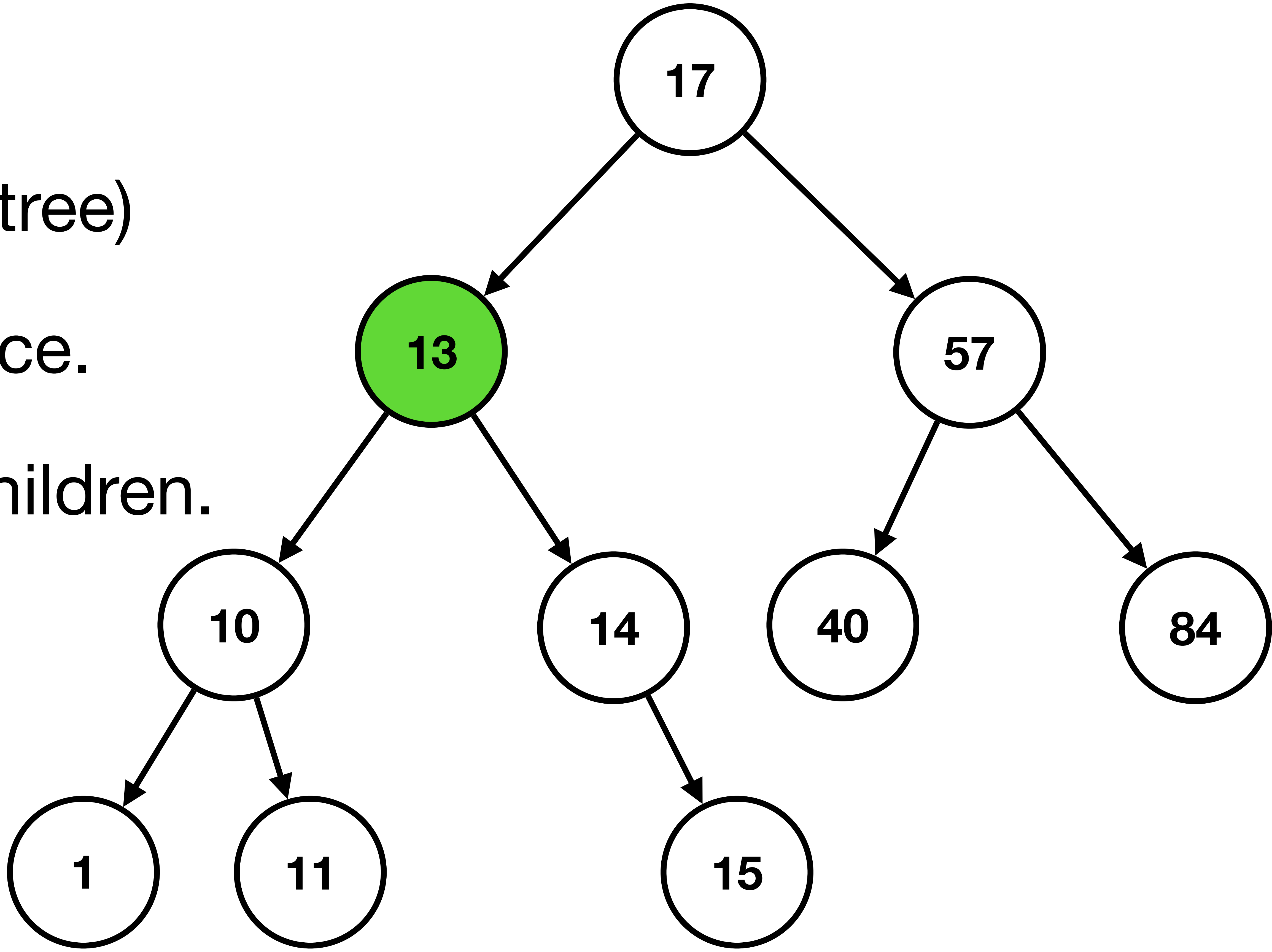
- Hardest case: node to be removed has two children
  - Find min(right substree), replace
  - Call remove recursively on min(right subtree)
    - This recursive call will only happen once.



BST

Remove

- Hardest case: node to be removed has two children
  - Find min(right substree), replace
  - Call remove recursively on min(right subtree)
    - This recursive call will only happen once.
    - min(right subtree) cannot have both children.



BST

Remove

BST
Remove

1. Find node to remove

BST

Remove

1. Find node to remove

- Easy case: node is leaf -- delete

BST

Remove

1. Find node to remove

- Easy case: node is leaf -- delete
- Harder case: node to remove has one child -- bypass

BST

Remove

1. Find node to remove

- Easy case: node is leaf -- delete
- Harder case: node to remove has one child -- bypass
- Hardest case: node to remove has both

BST

Remove

1. Find node to remove

- Easy case: node is leaf -- delete
- Harder case: node to remove has one child -- bypass
- Hardest case: node to remove has both
  - Find min(right subtree) -- replace

BST

Remove

1. Find node to remove

- Easy case: node is leaf -- delete
- Harder case: node to remove has one child -- bypass
- Hardest case: node to remove has both
  - Find min(right subtree) -- replace
  - Remove min(right subtree)

BST

Remove Complexity

1. Find node to remove

- Easy case: node is leaf -- delete
- Harder case: node to remove has one child -- bypass
- Hardest case: node to remove has both
  - Find min(right subtree) -- replace
  - Remove min(right subtree)

BST

Remove Complexity

1. Find node to remove **<-- O(height)**

- Easy case: node is leaf -- delete
- Harder case: node to remove has one child -- bypass
- Hardest case: node to remove has both
  - Find min(right subtree) -- replace
  - Remove min(right subtree)

BST

Remove Complexity

1. Find node to remove <-- O(height)

- Easy case: node is leaf -- delete <-- O(1)
- Harder case: node to remove has one child -- bypass
- Hardest case: node to remove has both
  - Find min(right subtree) -- replace
  - Remove min(right subtree)

BST

Remove Complexity

1. Find node to remove **<-- O(height)**

- Easy case: node is leaf -- delete **<-- O(1)**
- Harder case: node to remove has one child -- bypass **<-- O(1)**
- Hardest case: node to remove has both
  - Find min(right subtree) -- replace
  - Remove min(right subtree)

BST

Remove Complexity

1. Find node to remove <-- **O(height)**

- Easy case: node is leaf -- delete <-- **O(1)**
- Harder case: node to remove has one child -- bypass <-- **O(1)**
- Hardest case: node to remove has both
  - Find min(right subtree) -- replace <-- **O(height)**
  - Remove min(right subtree)

BST

Remove Complexity

1. Find node to remove **<-- O(height)**

- Easy case: node is leaf -- delete **<-- O(1)**
- Harder case: node to remove has one child -- bypass **<-- O(1)**
- Hardest case: node to remove has both
  - Find min(right subtree) -- replace **<-- O(height)**
  - Remove min(right subtree) **<-- O(height)**

BST

Remove

BST

Remove

Overall complexity:

BST
Remove

Overall complexity:

$$O(\text{height}) + O(1) + O(1) + O(\text{height}) = O(\text{height})$$

BST

Remove

Overall complexity:

$$O(\text{height}) + O(1) + O(1) + O(\text{height}) = O(\text{height})$$

Same as insert and lookup.

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

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)	
BST						

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)	
BST	O(log n)					

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)	
BST	O(log n)	O(n)				

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)	
BST	O(log n)	O(n)	O(log n)			

Maps

Complexity

<table>
<tr><th rowspan="2"></th><th colspan="2">lookup</th><th colspan="2">insert</th><th colspan="2">remove</th></tr>
<tr><th>average</th><th>worst</th><th>average</th><th>worst</th><th>average</th><th>worst</th></tr>
<tr><td>ArrayList</td><td colspan="2">O(n)</td><td>O(1)</td><td>O(n)</td><td colspan="2">O(1)</td></tr>
<tr><td>Linked List</td><td colspan="2">O(n)</td><td colspan="2">O(1)</td><td colspan="2">O(1)</td></tr>
<tr><td>ArrayList (sorted)</td><td colspan="2">O(log n)</td><td colspan="2">O(n)</td><td colspan="2">O(n)</td></tr>
<tr><td>Linked List (sorted)</td><td colspan="2">O(n)</td><td colspan="2">O(1)</td><td colspan="2">O(1)</td></tr>
<tr><td>BST</td><td>O(log n)</td><td>O(n)</td><td>O(log n)</td><td>O(n)</td><td></td><td></td></tr>
</table>

Maps

Complexity

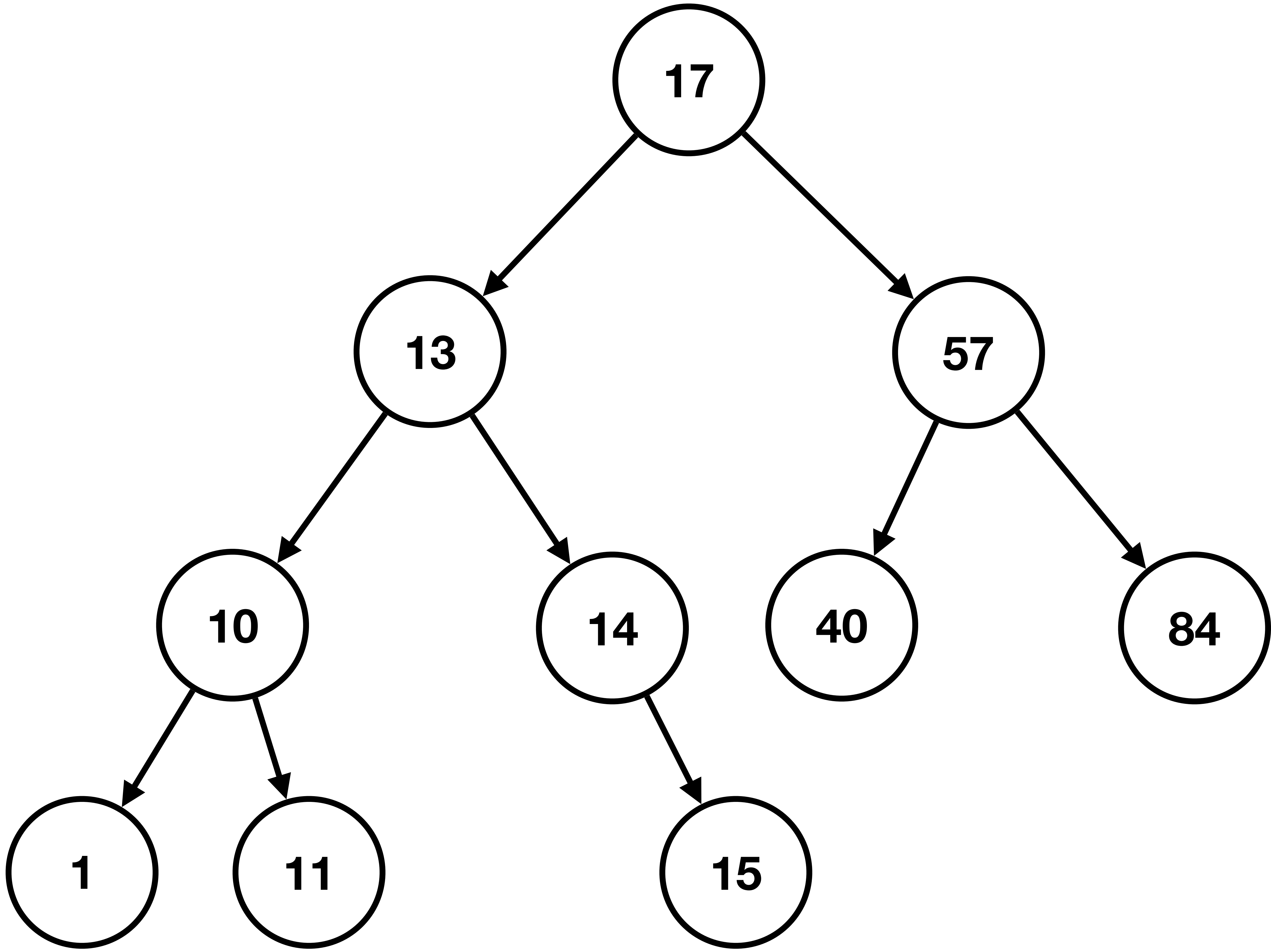
	lookup		insert		remove	
	average	worst	average	worst	average	worst
ArrayList	O(n)		O(1)	O(n)	O(1)	
Linked List	O(n)		O(1)		O(1)	
ArrayList (sorted)	O(log n)		O(n)		O(n)	
Linked List (sorted)	O(n)		O(1)		O(1)	
BST	O(log n)	O(n)	O(log n)	O(n)	O(log n)	

Maps

Complexity

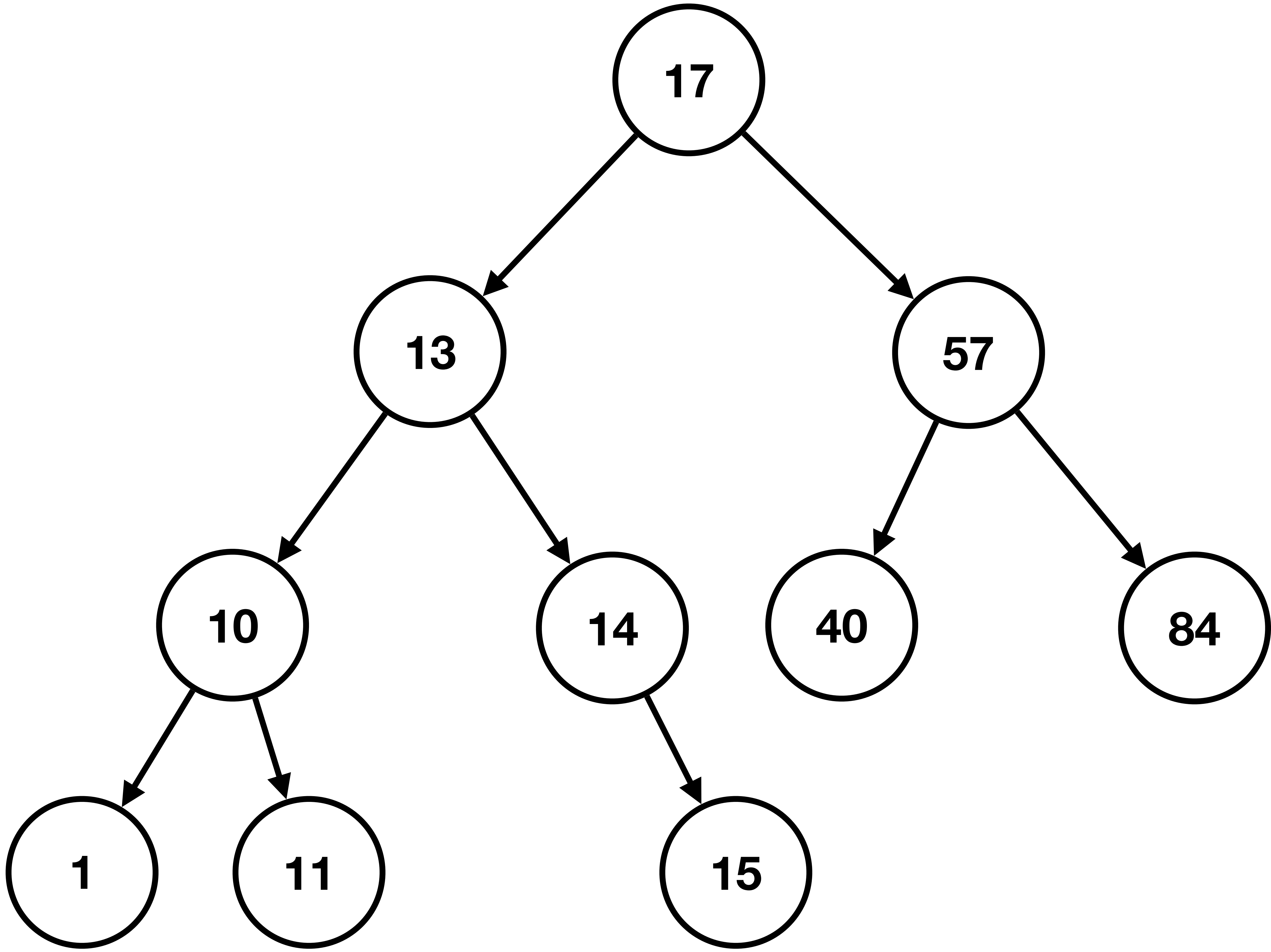
	lookup		insert		remove	
	`average`	`worst`	`average`	`worst`	`average`	`worst`
ArrayList	O(n)		O(1)	O(n)	O(1)	
Linked List	O(n)		O(1)		O(1)	
ArrayList (sorted)	O(log n)		O(n)		O(n)	
Linked List (sorted)	O(n)		O(1)		O(1)	
BST	O(log n)	O(n)	O(log n)	O(n)	O(log n)	O(n)

Back to sorting



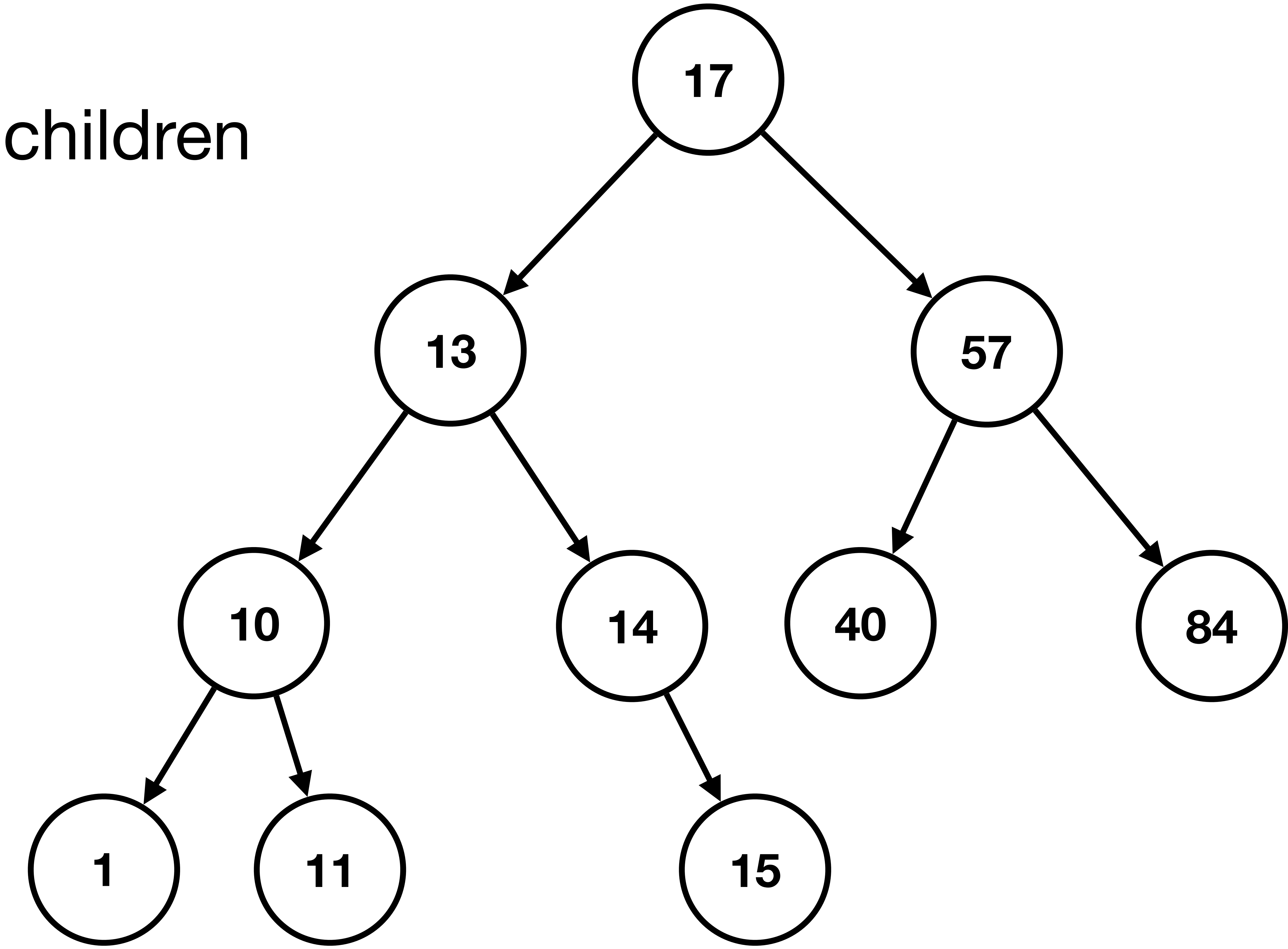
Back to sorting

- If we have a BST, how can we visit all nodes in sorted order?



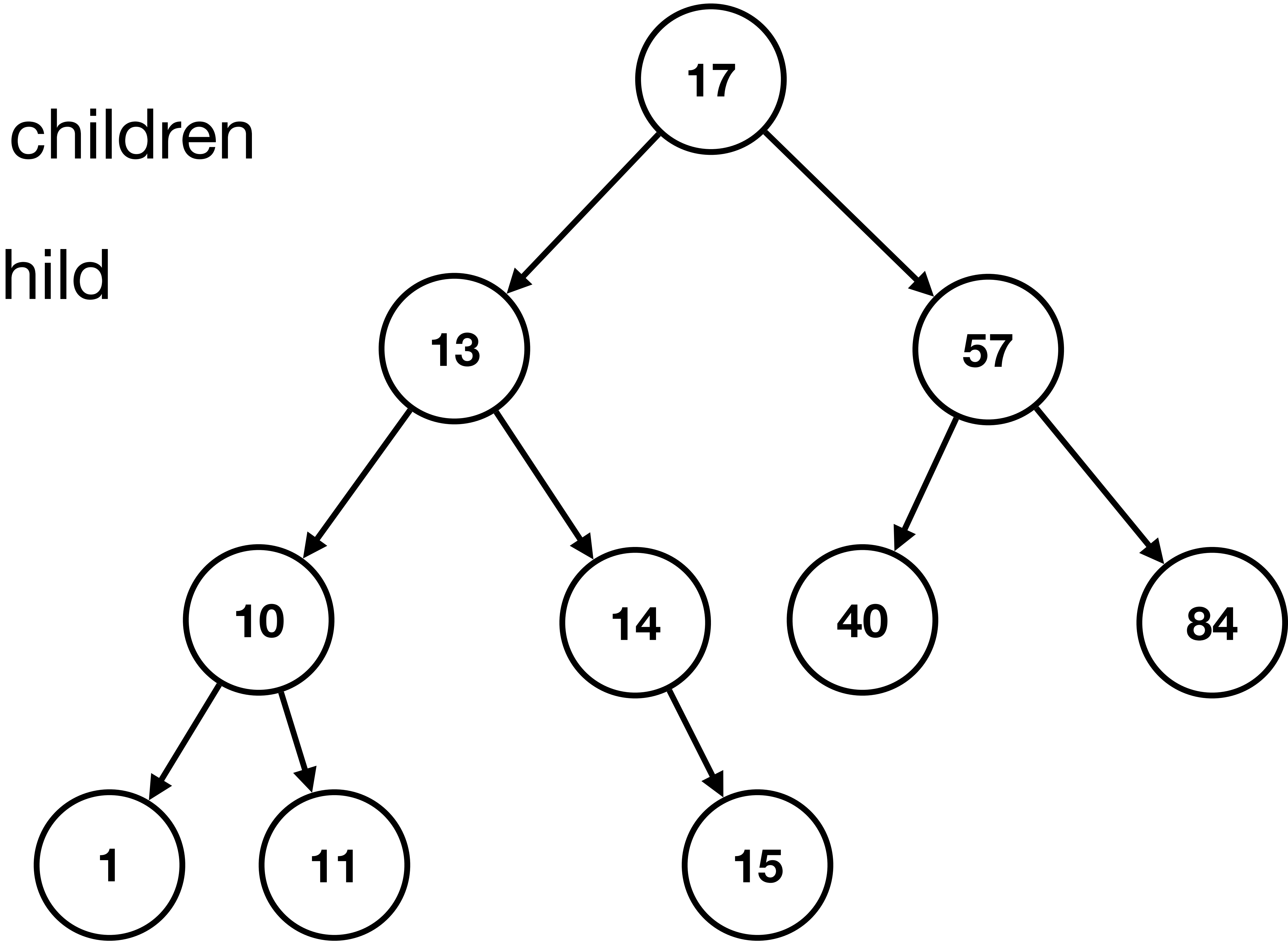
Back to sorting

- If we have a BST, how can we visit all nodes in sorted order?
  - pre-order traversal: curr first, then both children



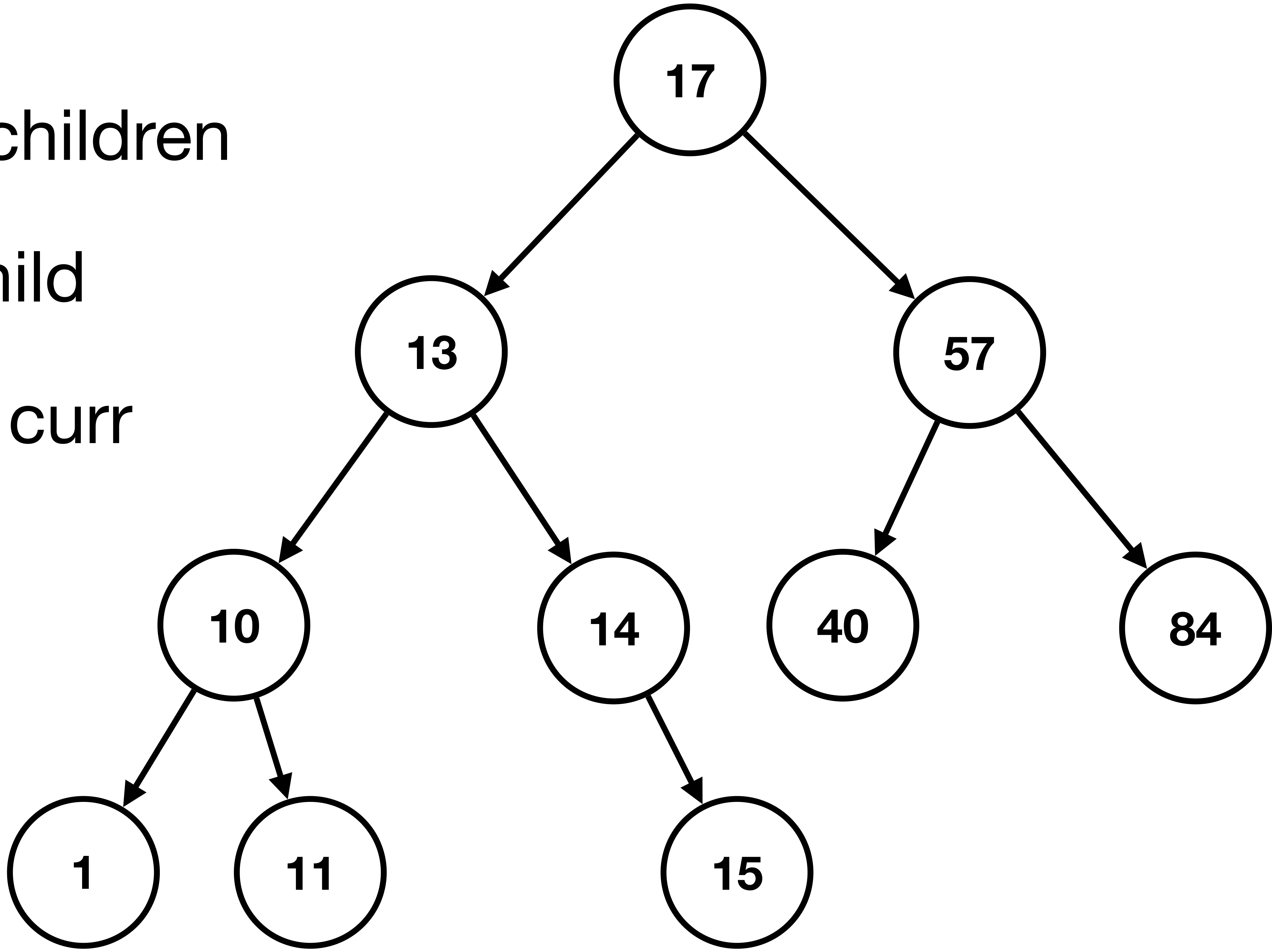
Back to sorting

- If we have a BST, how can we visit all nodes in sorted order?
  - pre-order traversal: curr first, then both children
  - in-order traversal: left child, curr, right child



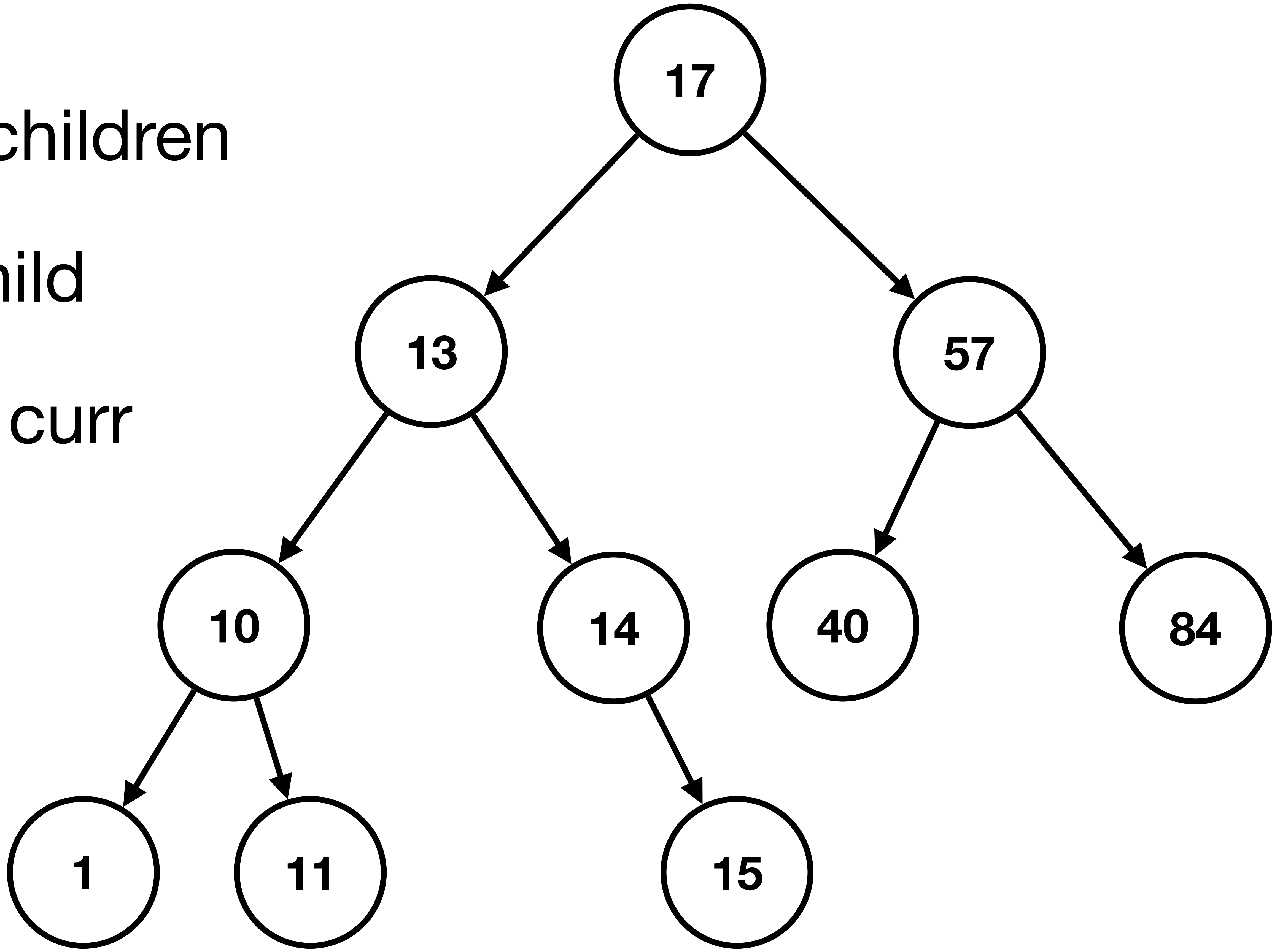
Back to sorting

- If we have a BST, how can we visit all nodes in sorted order?
  - pre-order traversal: curr first, then both children
  - in-order traversal: left child, curr, right child
  - post-order traversal: both children, then curr



Back to sorting

- If we have a BST, how can we visit all nodes in sorted order?
    - pre-order traversal: curr first, then both children
    - in-order traversal: left child, curr, right child
    - post-order traversal: both children, then curr



Sorting

In-order Traversal

```
void walk(struct tree_node *tree,
          void (*visit)(void *key, void *value, void *data),
          void *data);
```

Sorting

In-order Traversal

```
void walk(struct tree_node *tree,
          void (*visit)(void *key, void *value, void *data),
          void *data)
{
```

Sorting

In-order Traversal

```
void walk(struct tree_node *tree,
          void (*visit)(void *key, void *value, void *data),
          void *data)
{
        if (tree == NULL) {
```

Sorting

In-order Traversal

```
void walk(struct tree_node *tree,
          void (*visit)(void *key, void *value, void *data),
          void *data)
{
        if (tree == NULL) {
                return;
```

Sorting

In-order Traversal

```
void walk(struct tree_node *tree,
          void (*visit)(void *key, void *value, void *data),
          void *data)
{
        if (tree == NULL) {
                return;
        }
```

Sorting

In-order Traversal

```
void walk(struct tree_node *tree,
          void (*visit)(void *key, void *value, void *data),
          void *data)
{
        if (tree == NULL) {
                return;
        }
        walk(tree->left, visit, data);
```

Sorting

In-order Traversal

```
void walk(struct tree_node *tree,
          void (*visit)(void *key, void *value, void *data),
          void *data)
{
        if (tree == NULL) {
                return;
        }
        walk(tree->left, visit, data);
        visit(tree->key, tree->value, data);
```

Sorting

In-order Traversal

```
void walk(struct tree_node *tree,
          void (*visit)(void *key, void *value, void *data),
          void *data)
{
        if (tree == NULL) {
                return;
        }
        walk(tree->left, visit, data);
        visit(tree->key, tree->value, data);
        walk(tree->right, visit, data);
```

Sorting

In-order Traversal

```
void walk(struct tree_node *tree,
          void (*visit)(void *key, void *value, void *data),
          void *data)
{
        if (tree == NULL) {
                return;
        }
        walk(tree->left, visit, data);
        visit(tree->key, tree->value, data);
        walk(tree->right, visit, data);
}
```

Sorting

In-order Traversal

```
void walk(struct tree_node *tree,
          void (*visit)(void *key, void *value, void *data),
          void *data)
{
        if (tree == NULL) {
                return;
        }
        walk(tree->left, visit, data);
        visit(tree->key, tree->value, data);
        walk(tree->right, visit, data);
}
```

Sorting

Pre-order Traversal

```
void walk(struct tree_node *tree,
          void (*visit)(void *key, void *value, void *data),
          void *data)
{
        if (tree == NULL) {
                return;
        }
        visit(tree->key, tree->value, data);
        walk(tree->left, visit, data);
        walk(tree->right, visit, data);
}
```

Sorting

Post-order Traversal

```
void walk(struct tree_node *tree,
          void (*visit)(void *key, void *value, void *data),
          void *data)
{
        if (tree == NULL) {
                return;
        }
        walk(tree->left, visit, data);
        walk(tree->right, visit, data);
        visit(tree->key, tree->value, data);
}
```