Sorting II

CS143: lecture 9

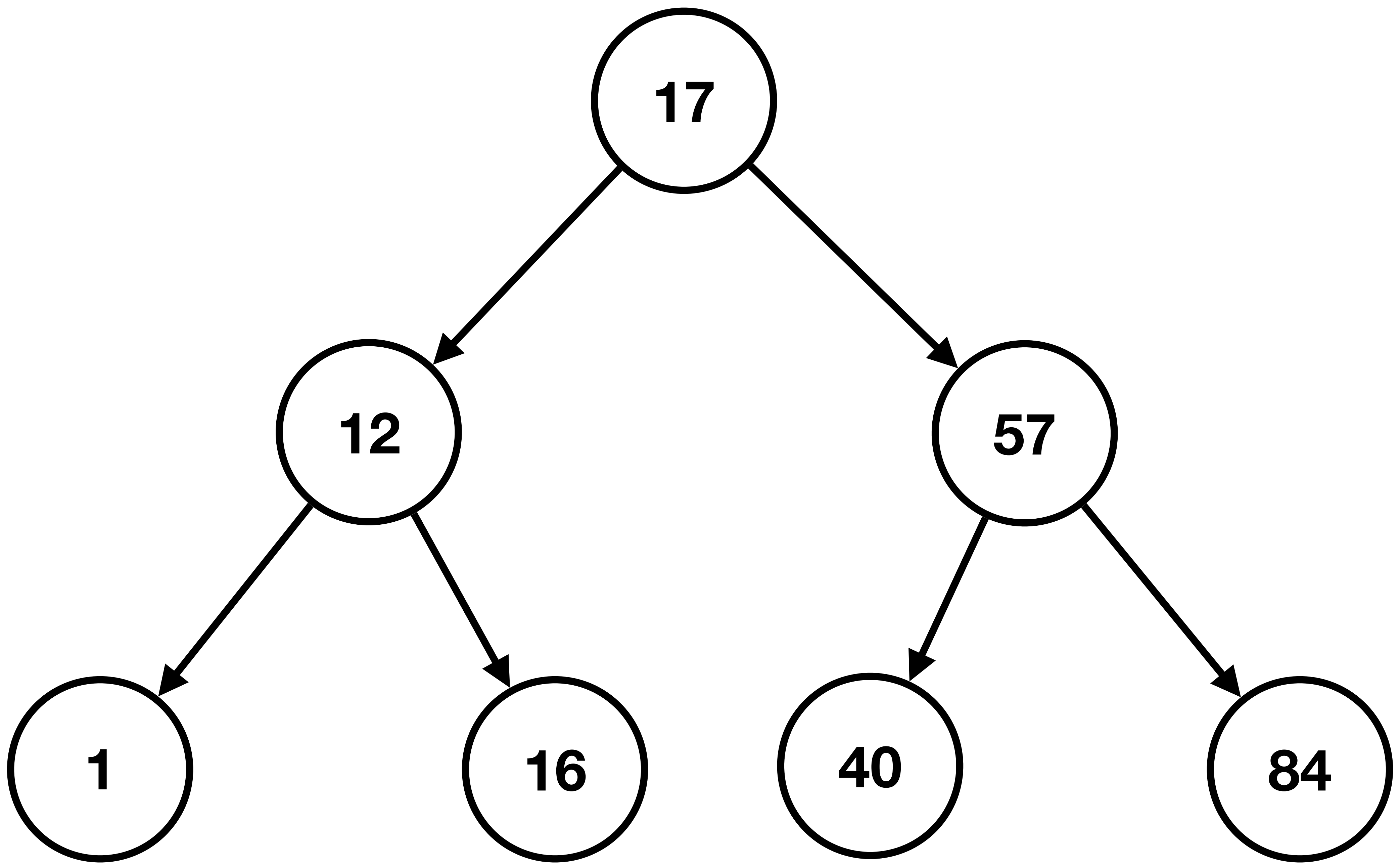
Byron Zhong, July 3

In-class Quiz

- Monday, July 10: 6:00-7:20pm
- Cheat sheet: 1 letter-size, double-sided, *hand-written* note
- Topics:
  - C Basics: Syntax, Pointers, Functions, Arrays, Types, ...
  - Heap and Stack: Pass-by-reference, frames, malloc and free, strings (hw1)
  - Lists: Array Lists, Linked List, sorting
  - BST

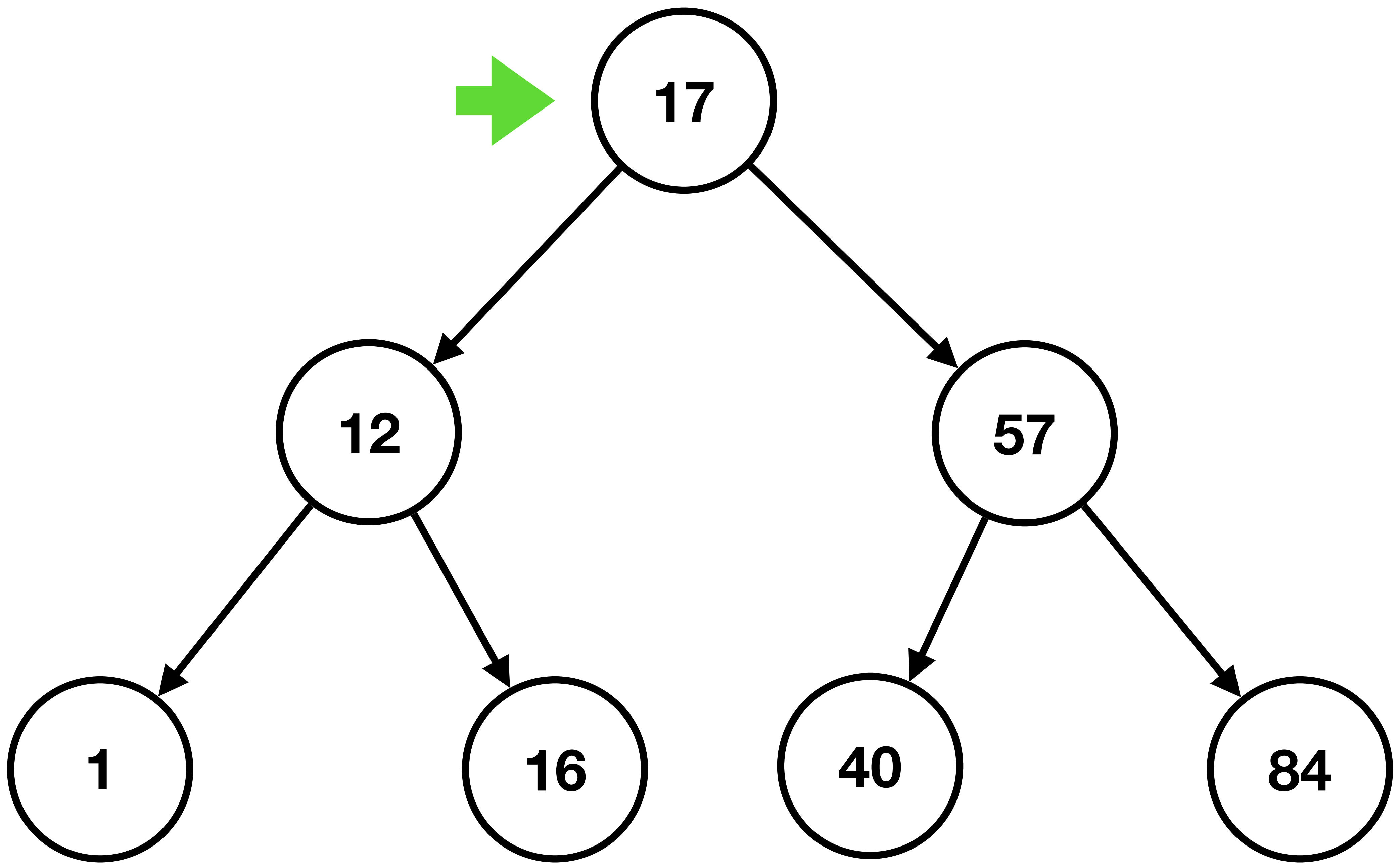
BST Review

Look up 16



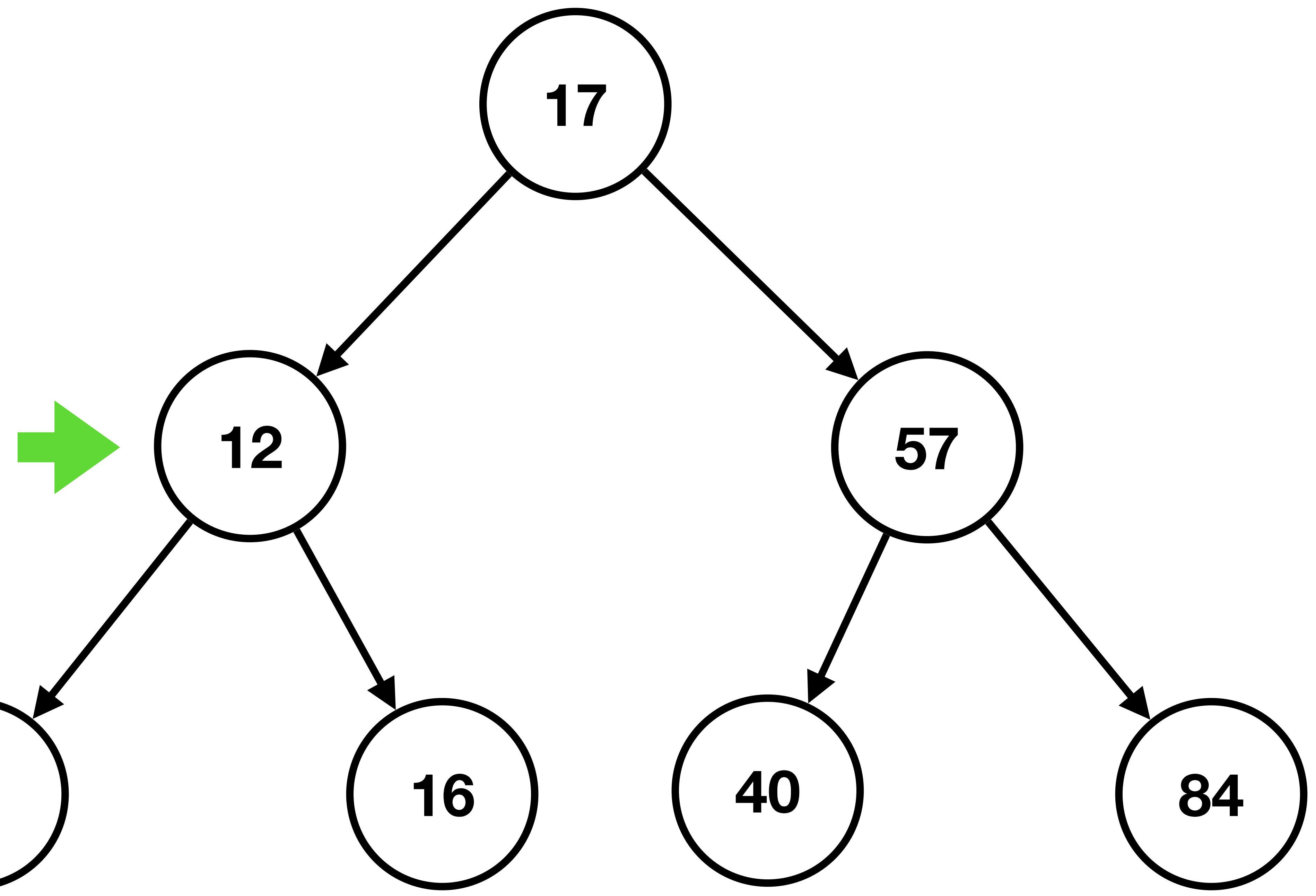
BST Review

Look up 16



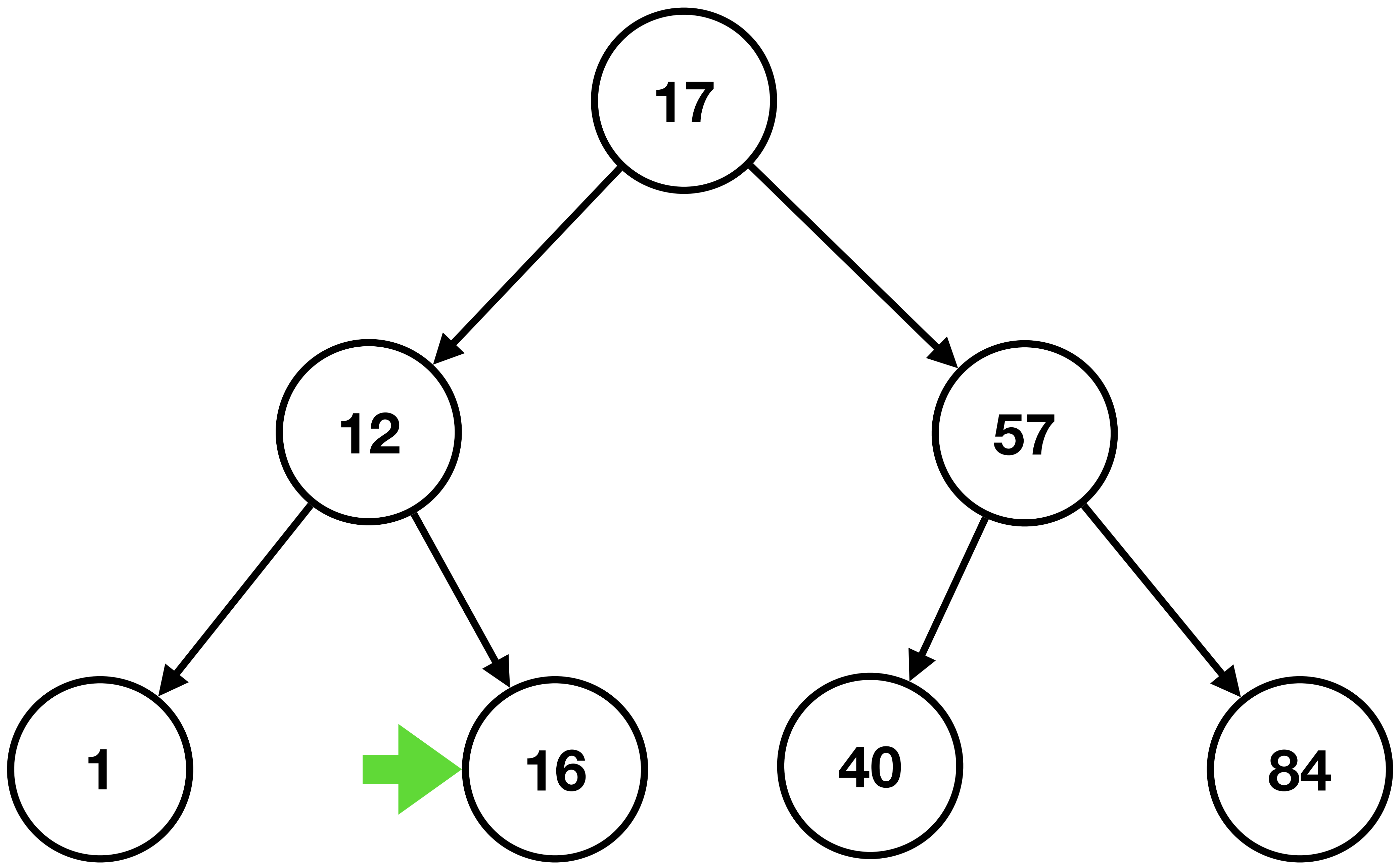
BST Review

Look up 16



BST Review

Look up 16



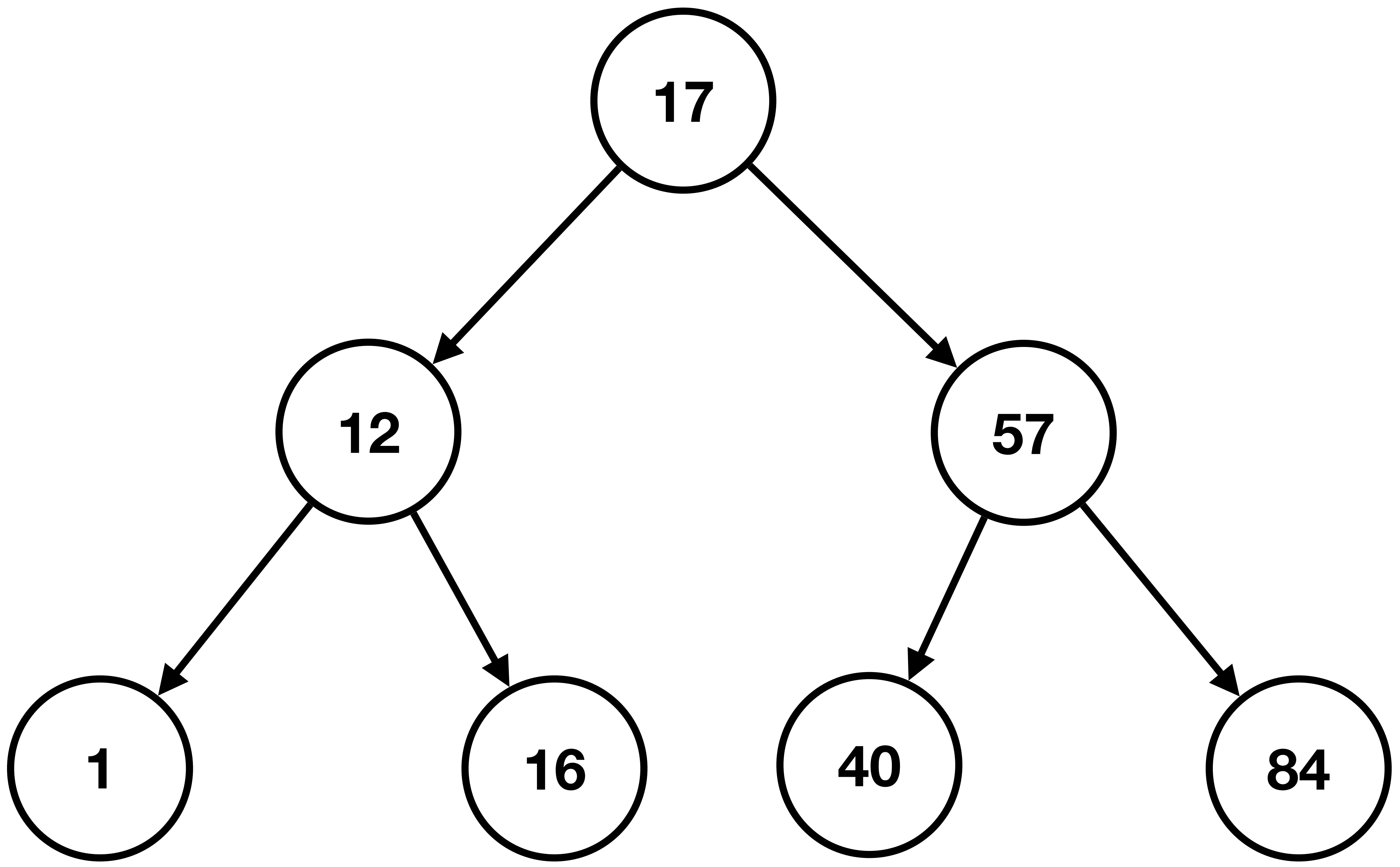
BST Review

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?
  - $O(\text{height})$

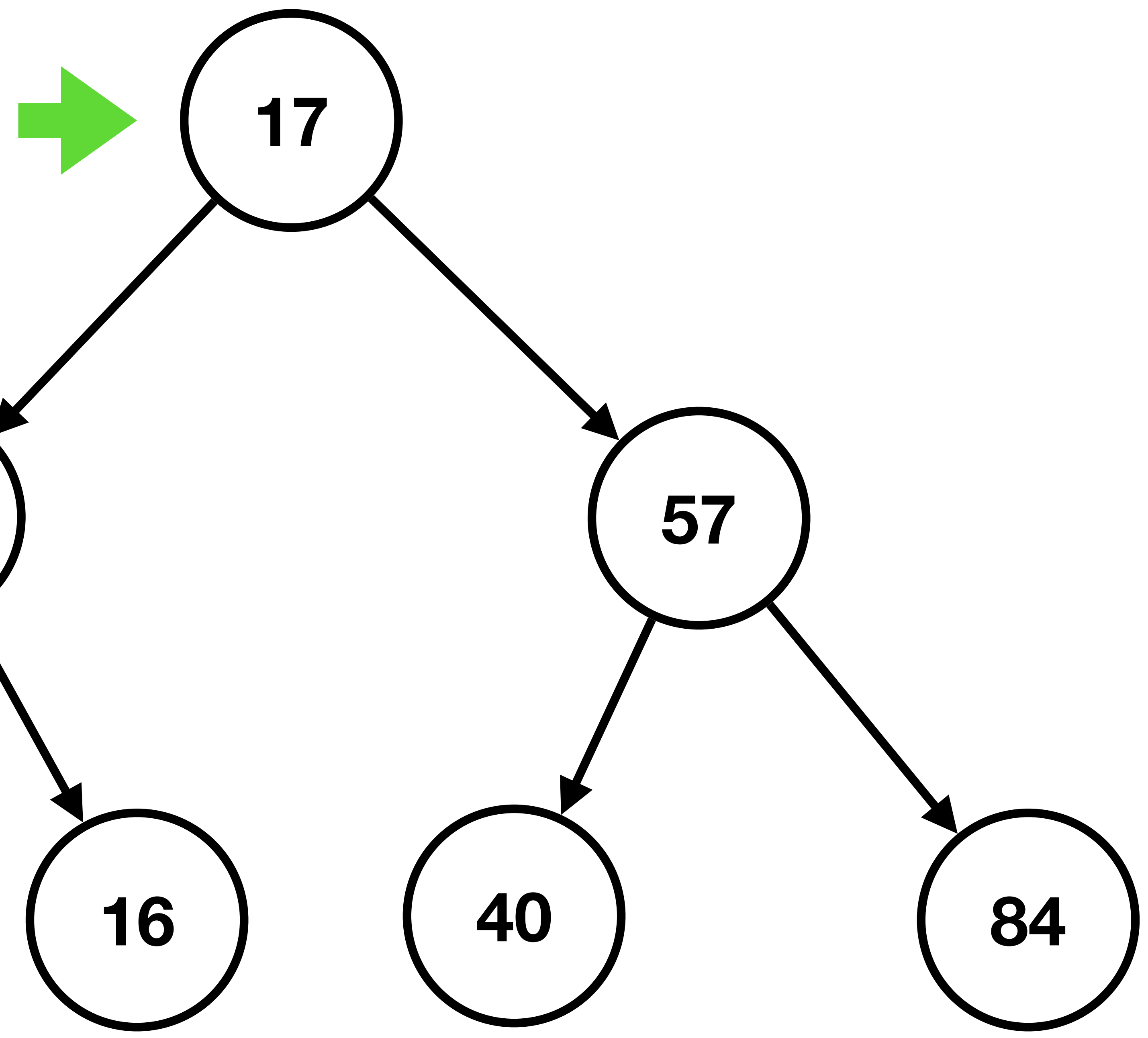
BST Review

Insert 18



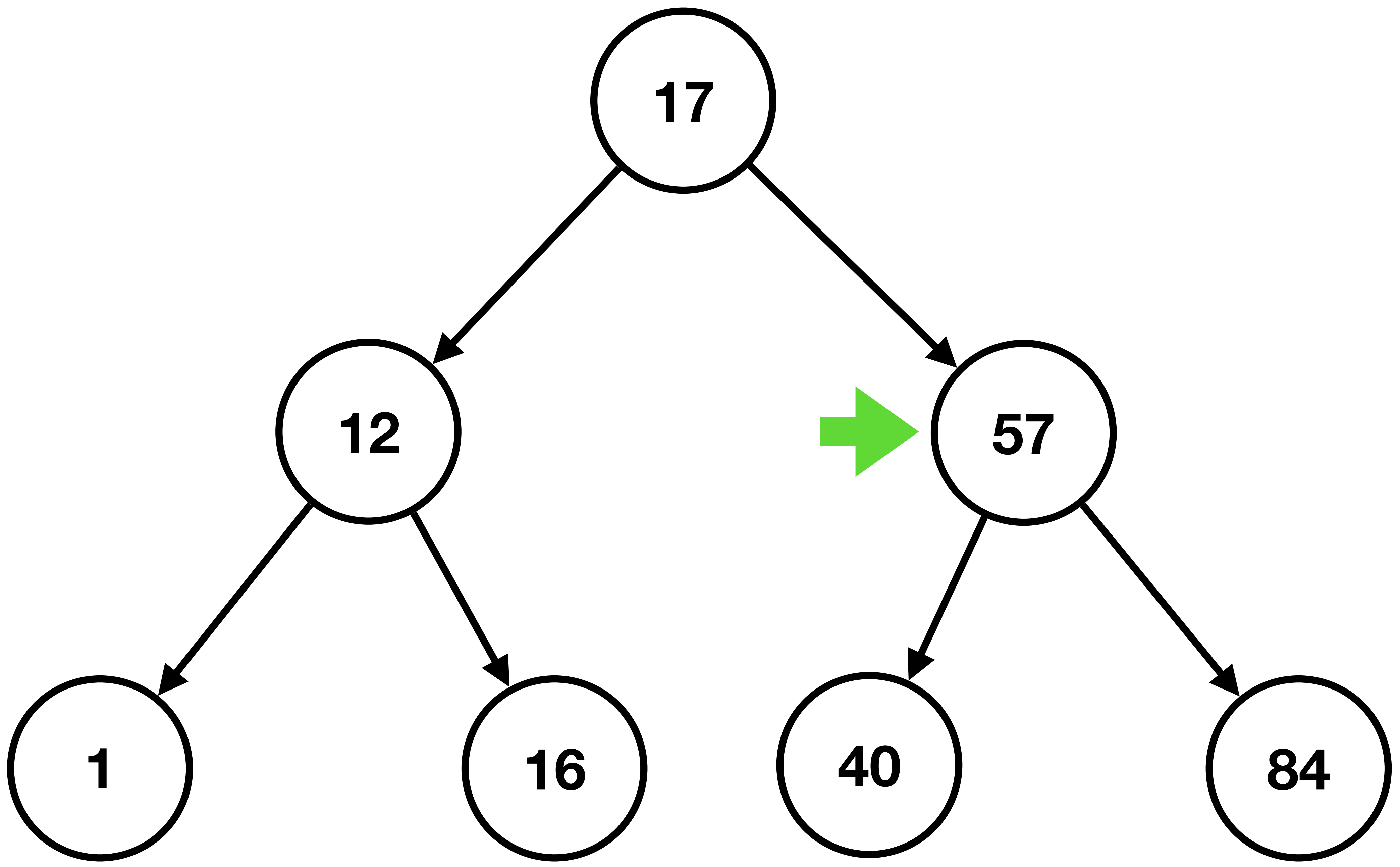
BST Review

Insert 18



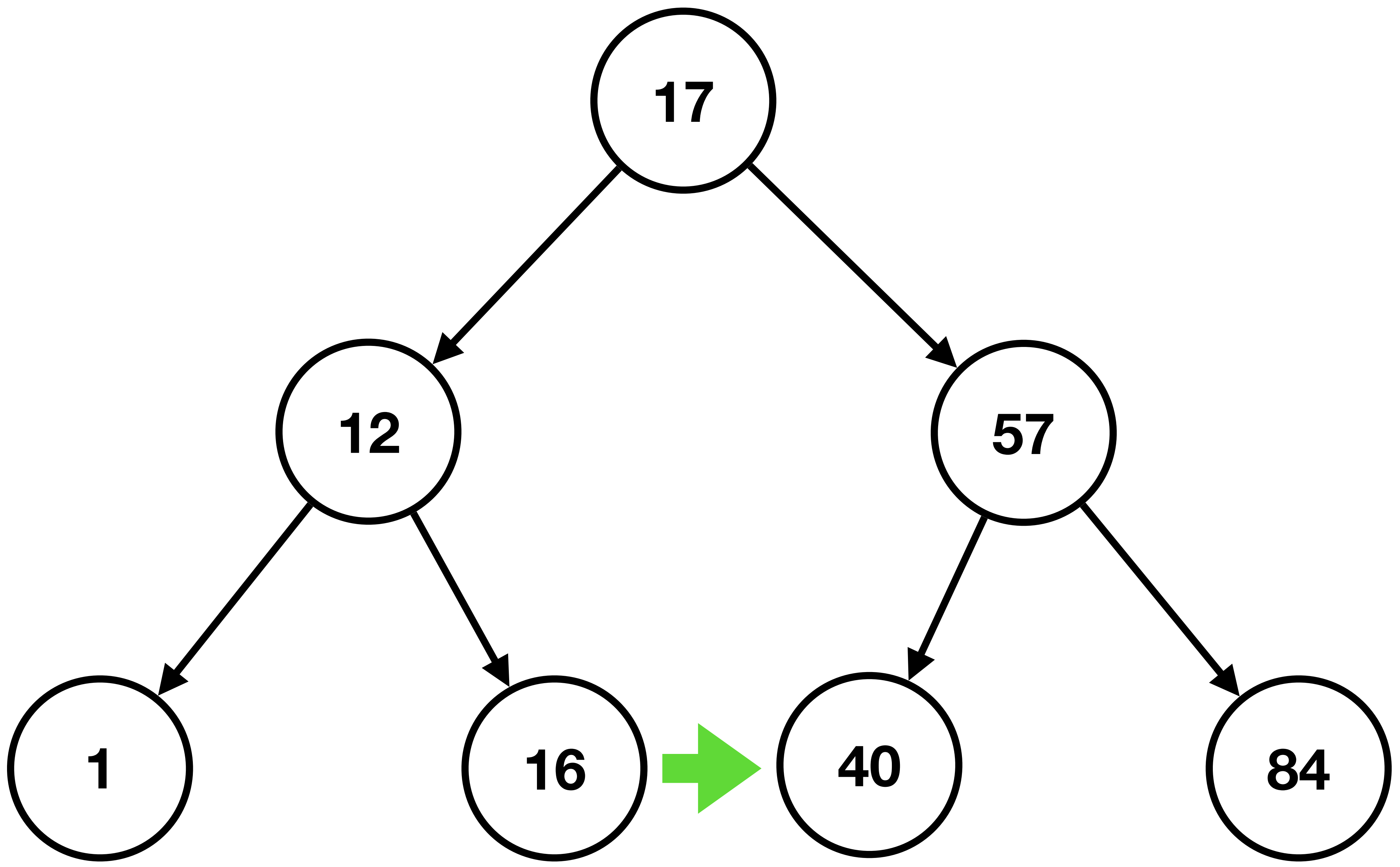
BST Review

Insert 18



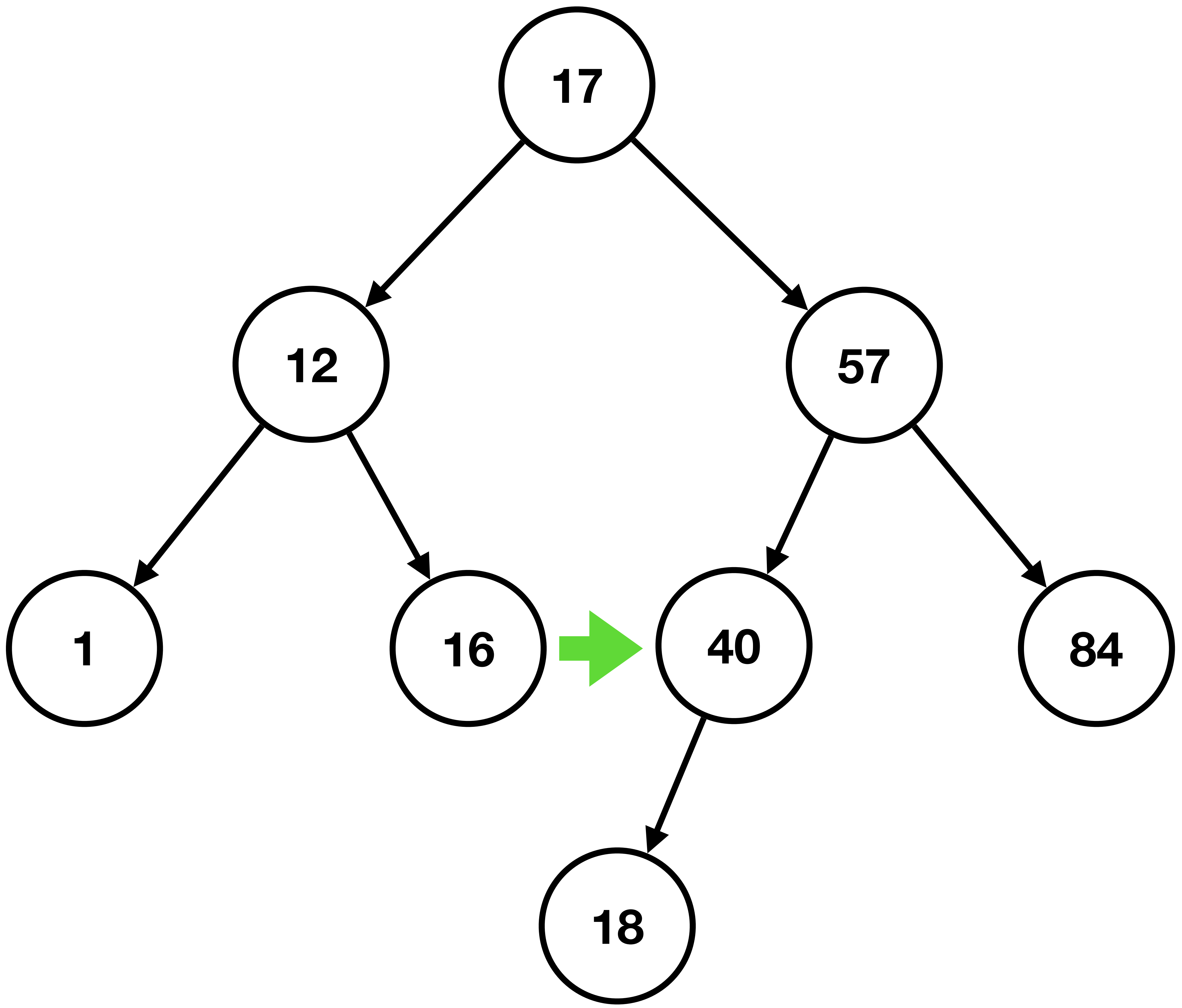
BST Review

Insert 18



BST Review

Insert 18



BST Review

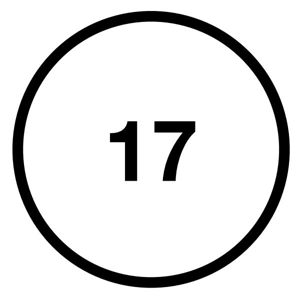
Insert

- 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 $O(1)$

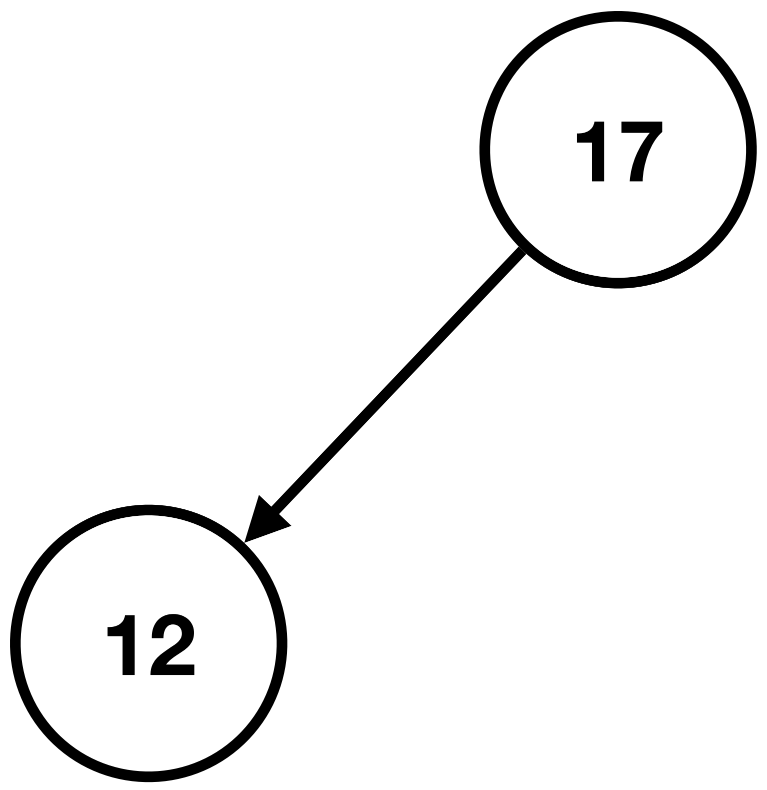
BST
Height

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



BST
Height

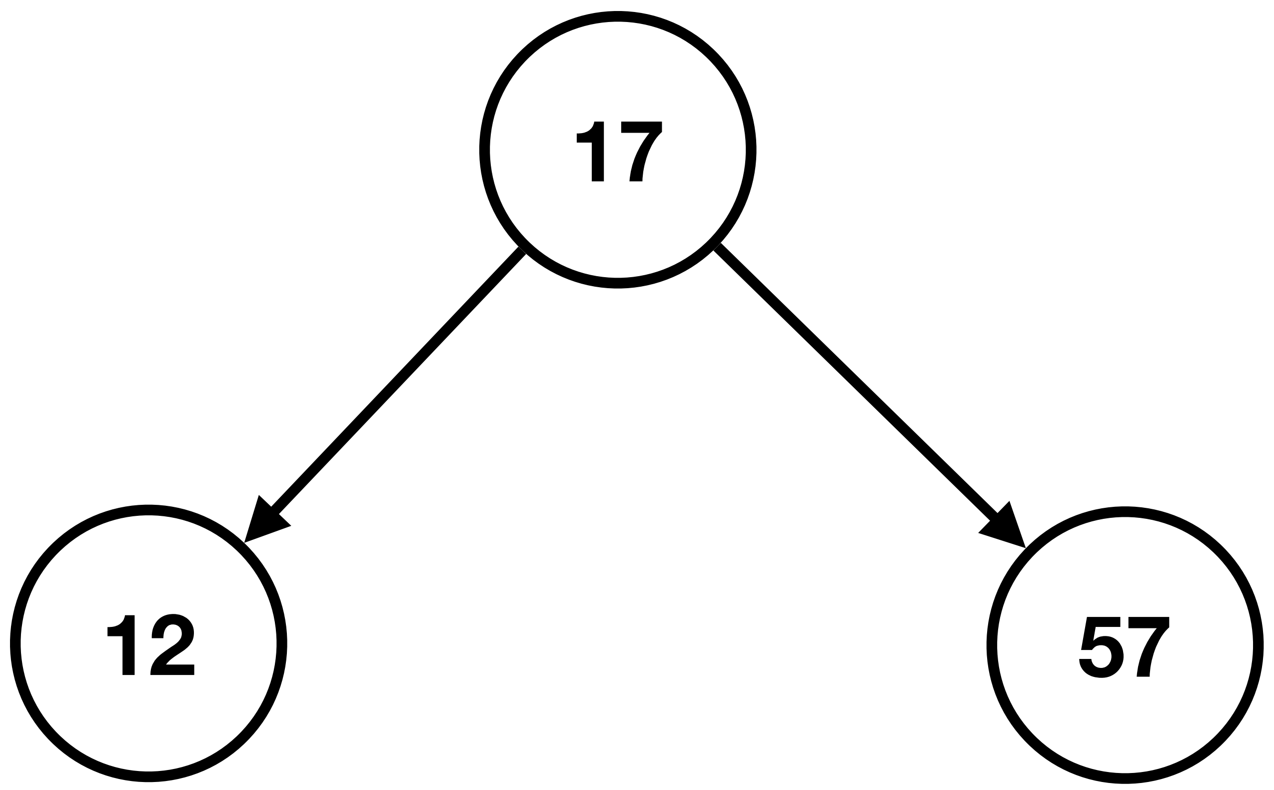
- Insert: 17, 12, 57, 1, 16, 40, 84



BST

Height

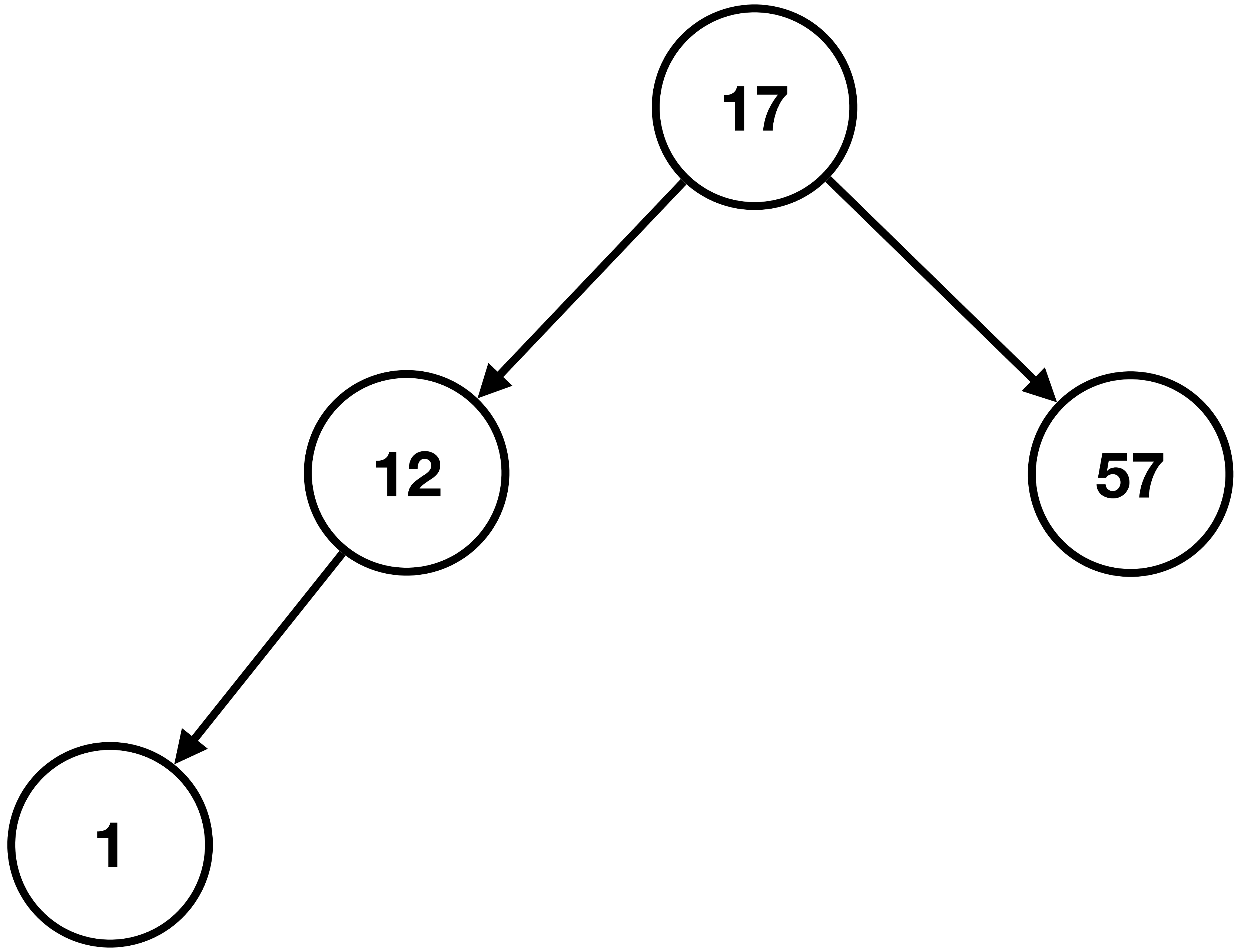
- Insert: 17, 12, 57, 1, 16, 40, 84



BST

Height

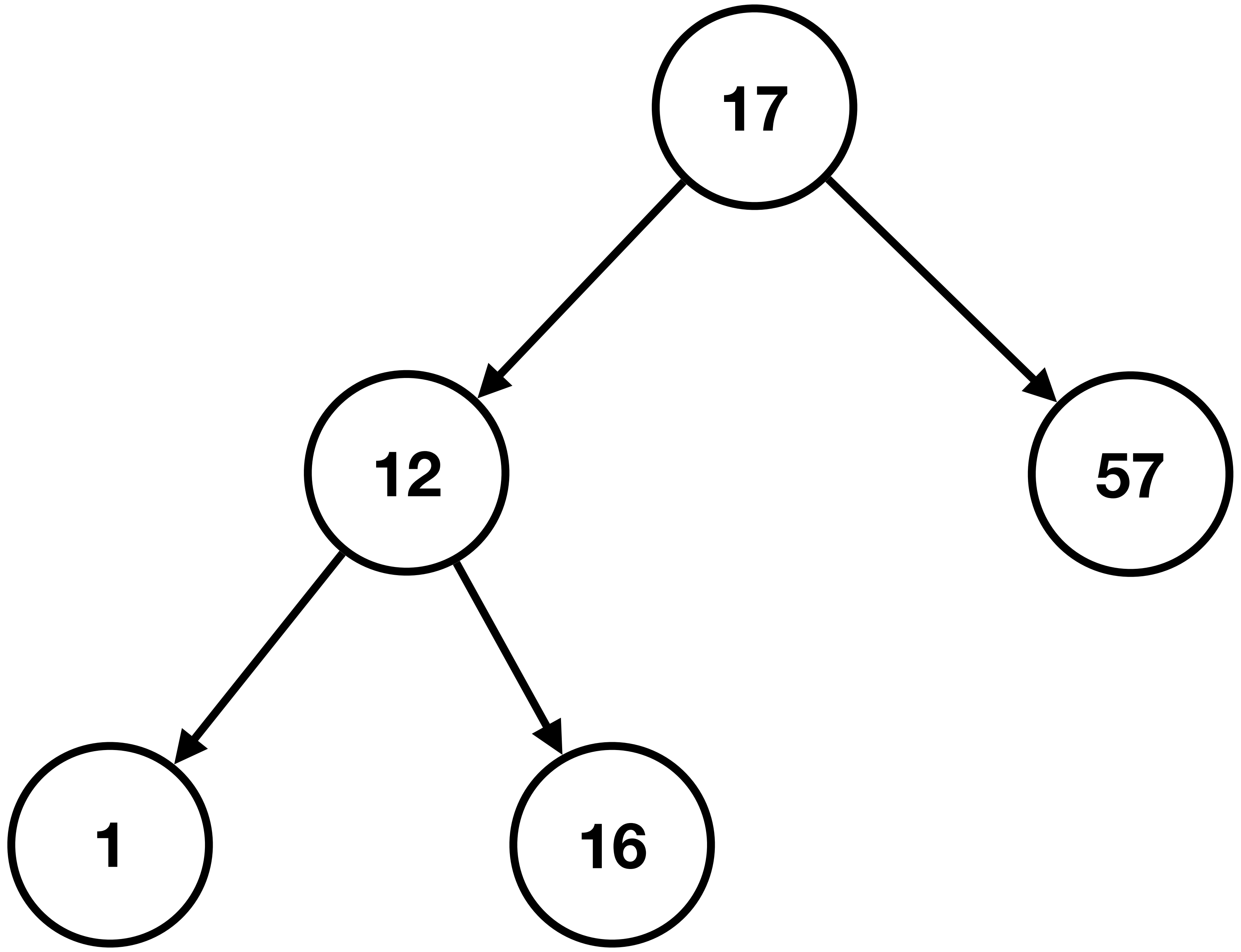
- Insert: 17, 12, 57, 1, 16, 40, 84



BST

Height

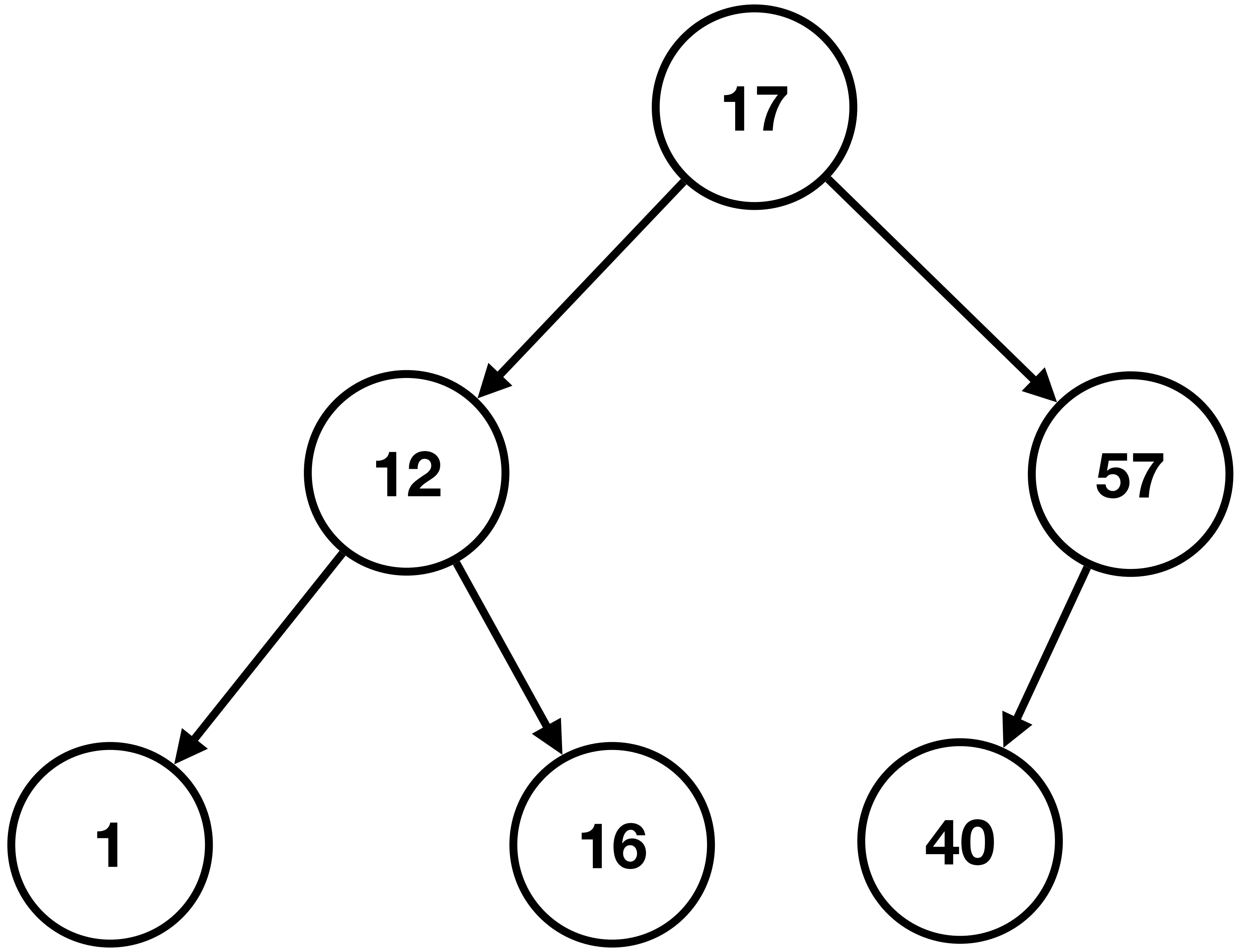
- Insert: 17, 12, 57, 1, 16, 40, 84



BST

Height

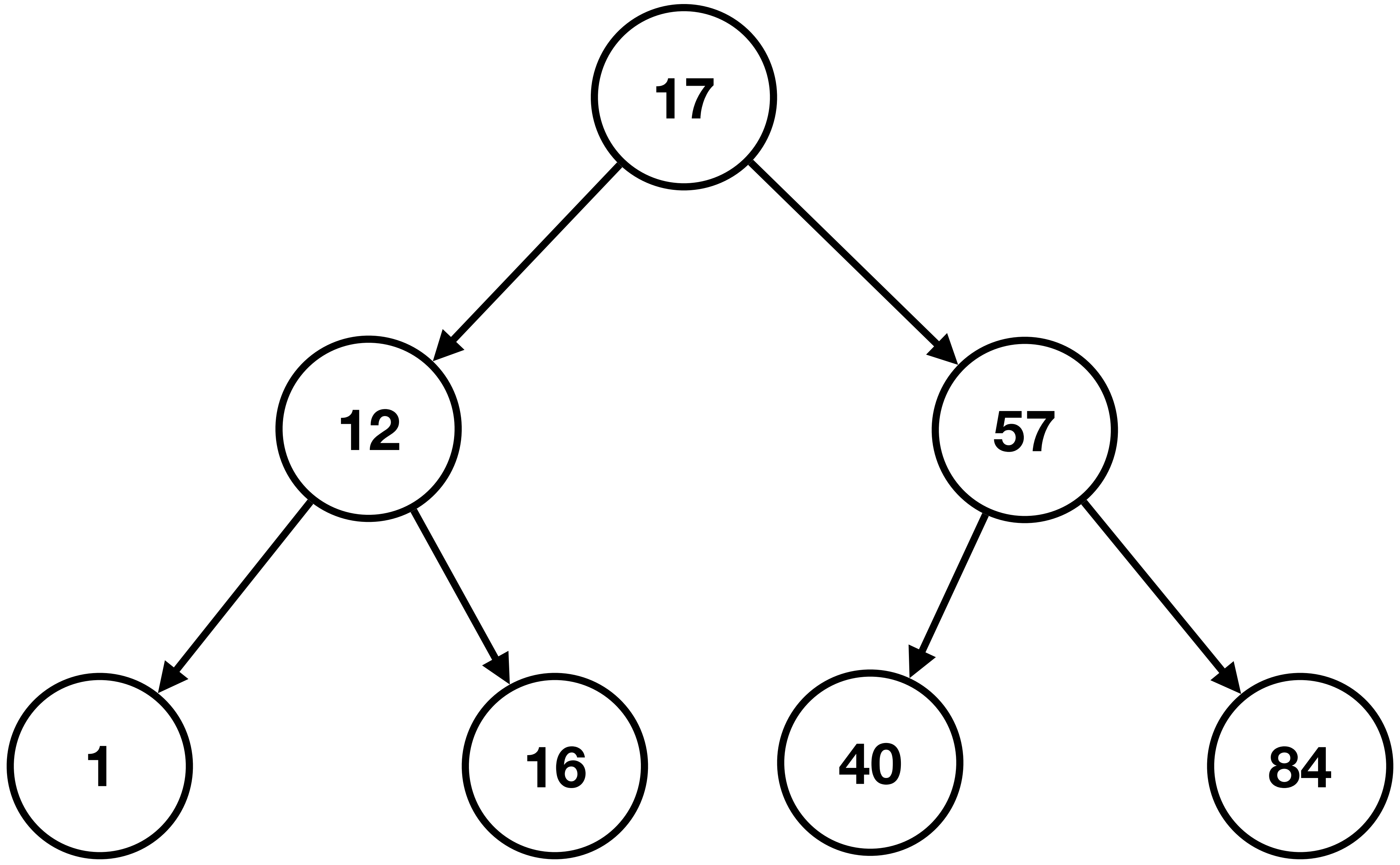
- Insert: 17, 12, 57, 1, 16, 40, 84



BST

Height

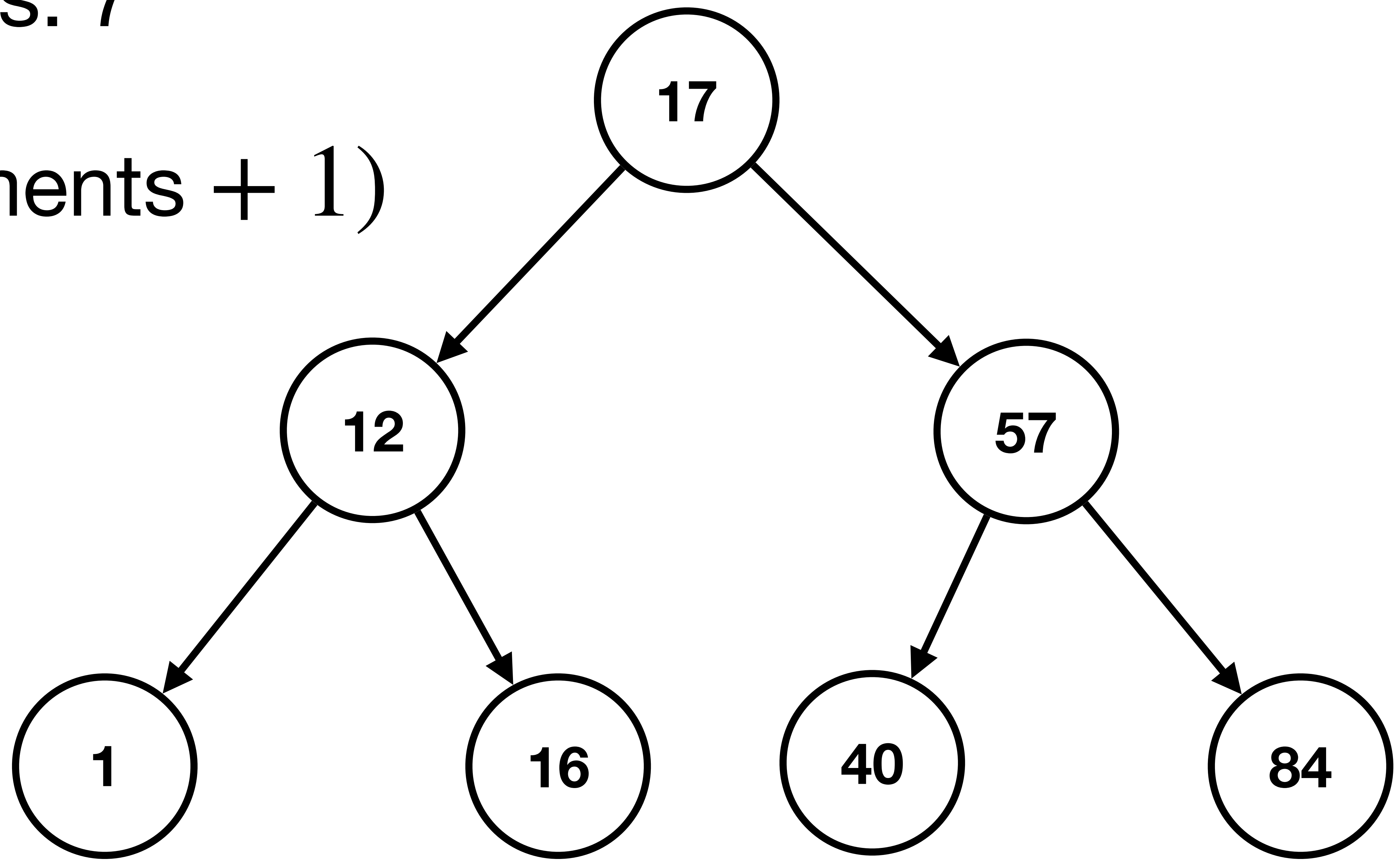
- Insert: 17, 12, 57, 1, 16, 40, 84



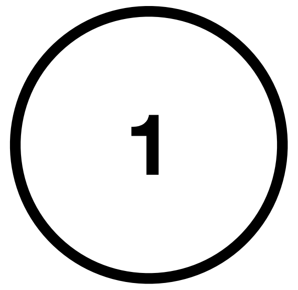
BST

Height

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

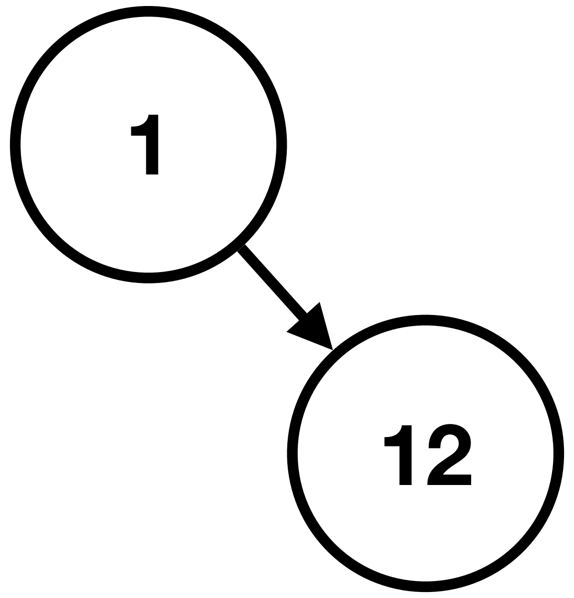


BST
Height



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

BST
Height

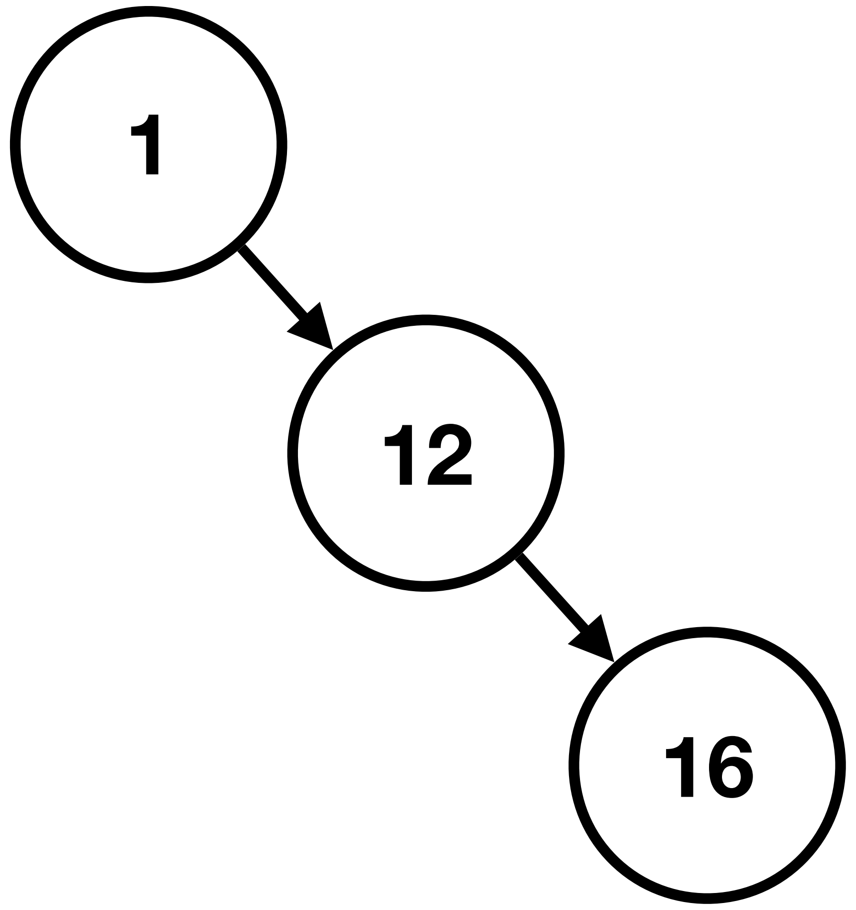


- Insert: 1, 12, 16, 17, 40, 57, 84

BST

Height

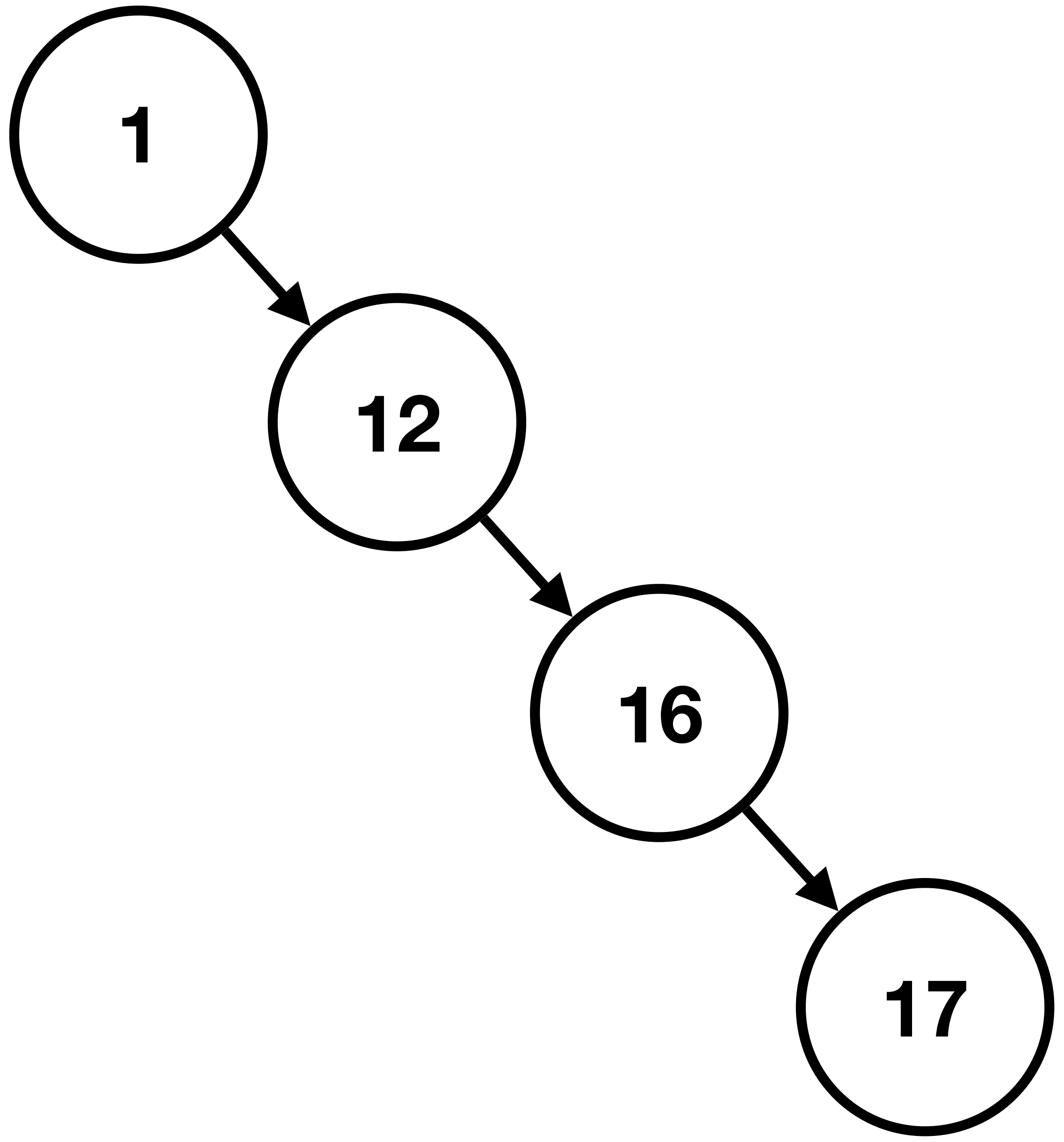
- Insert: 1, 12, 16, 17, 40, 57, 84



BST

Height

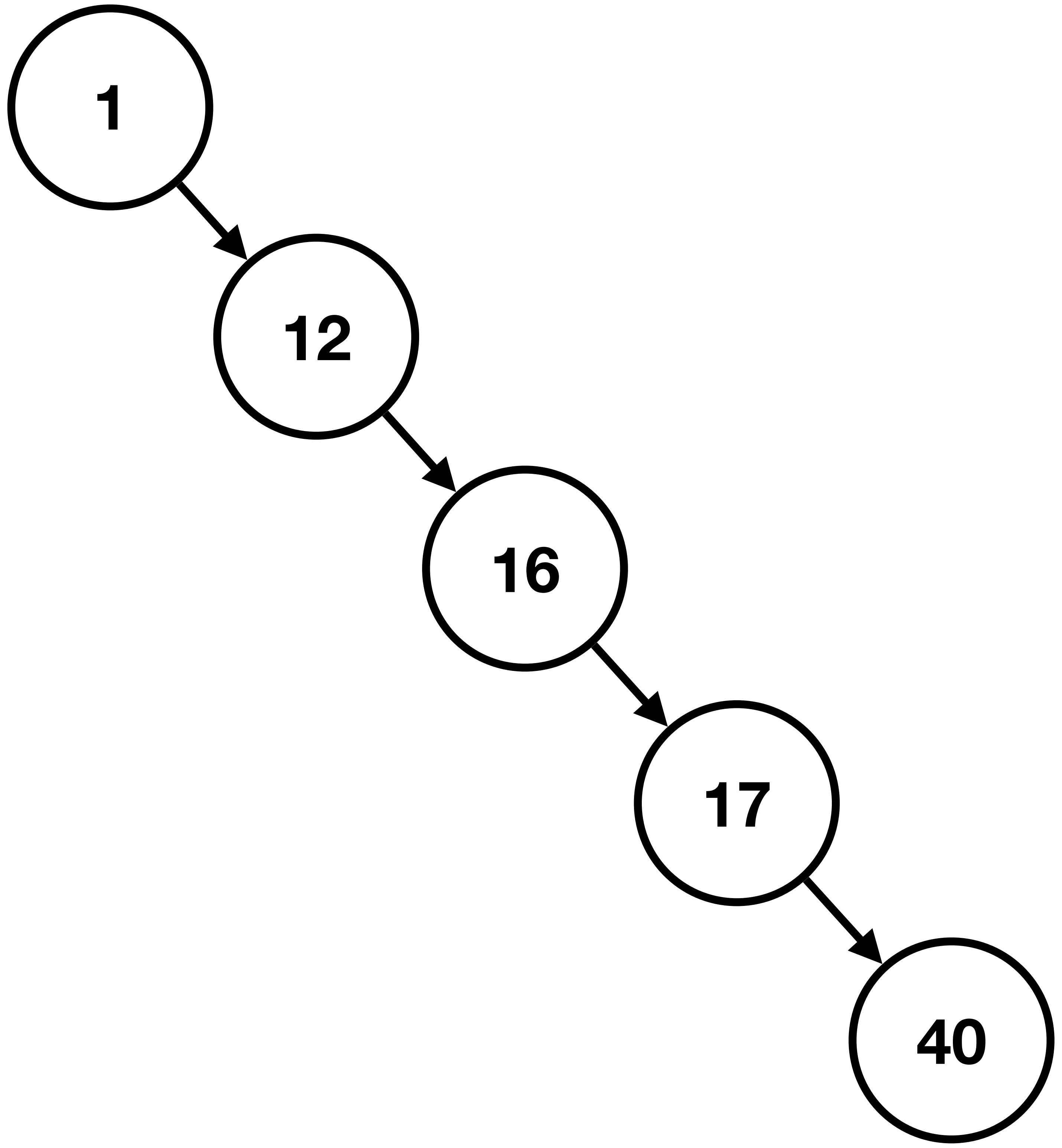
- Insert: 1, 12, 16, 17, 40, 57, 84



BST

Height

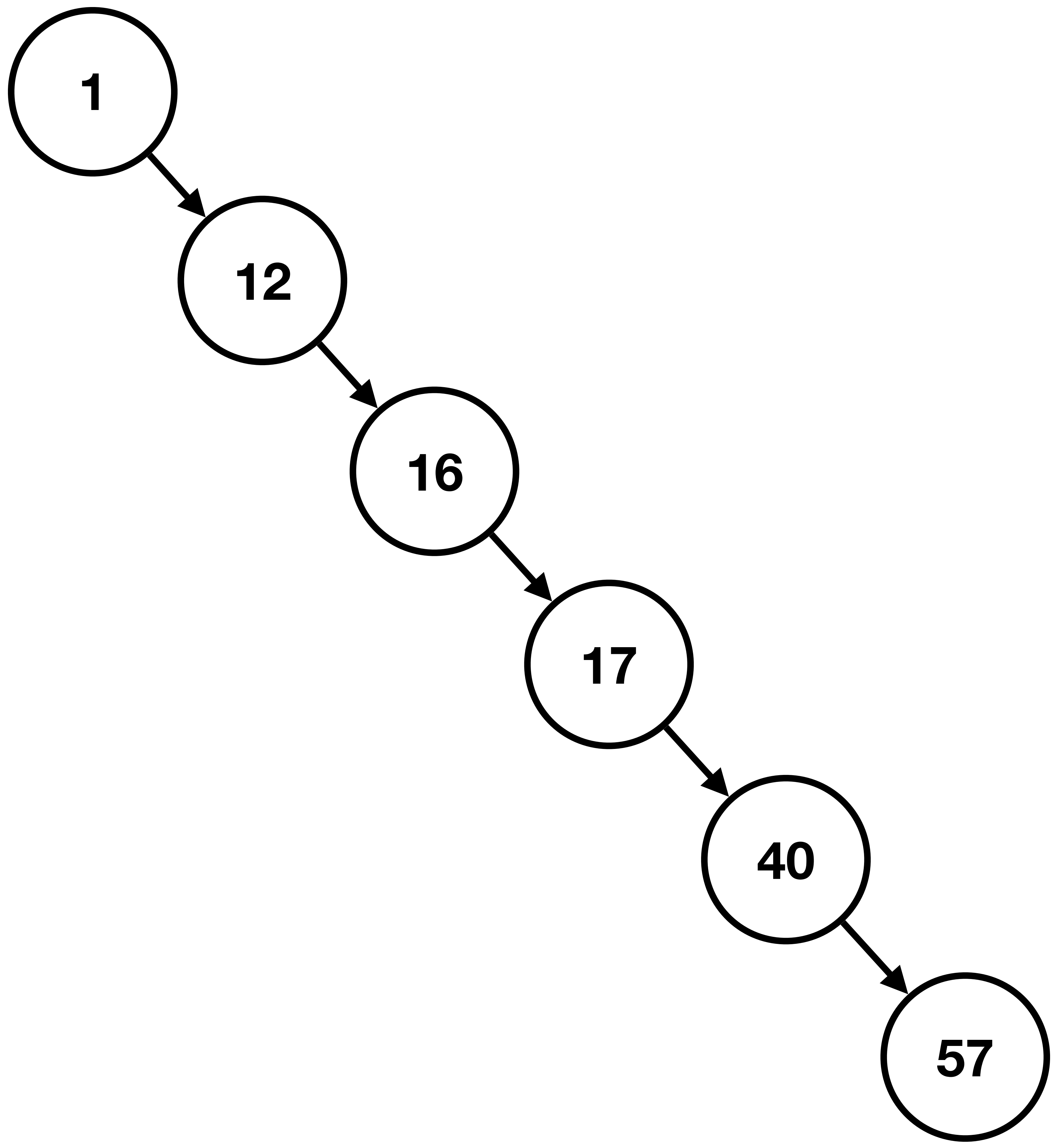
- Insert: 1, 12, 16, 17, 40, 57, 84



BST

Height

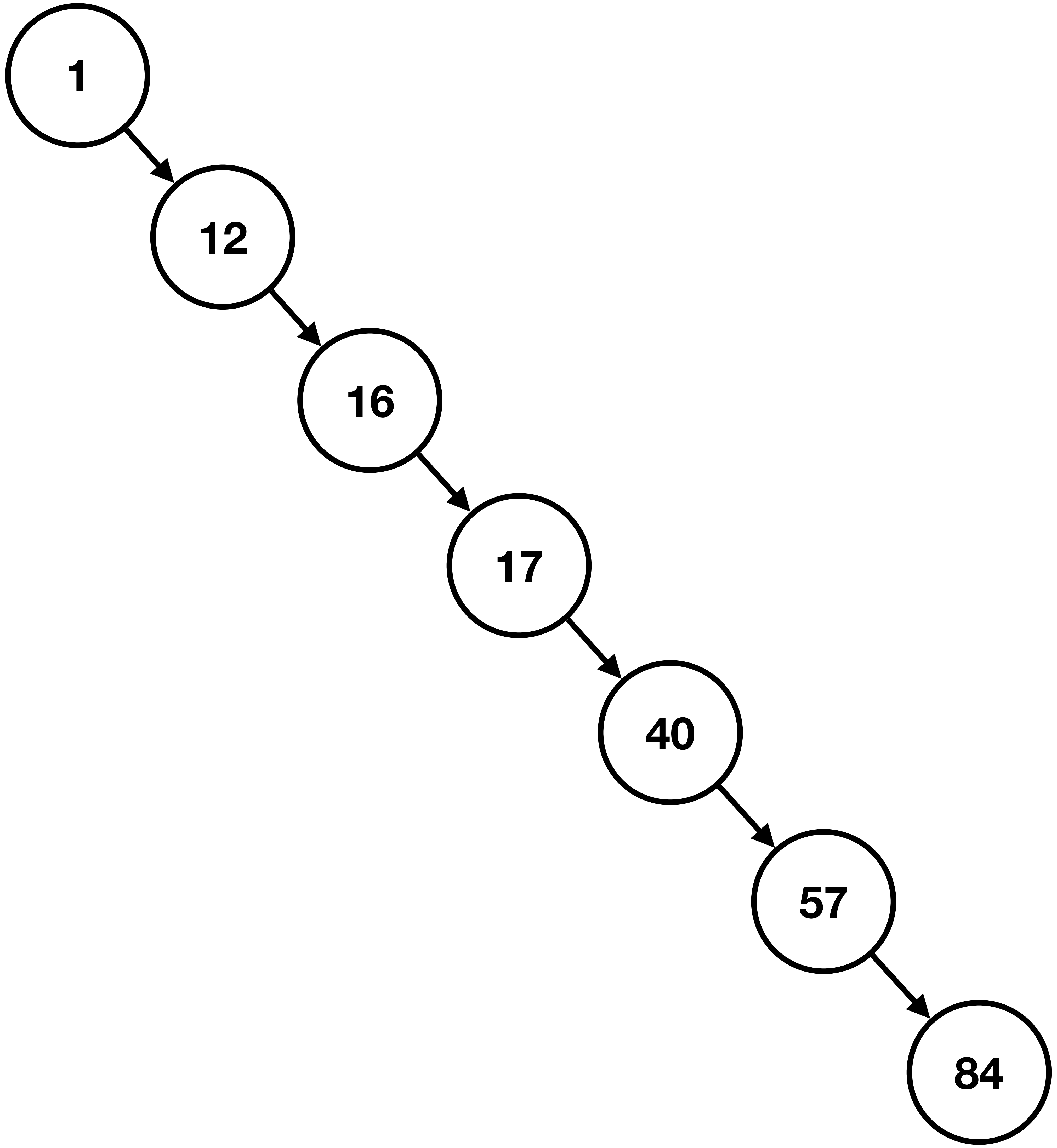
- Insert: 1, 12, 16, 17, 40, 57, 84



BST

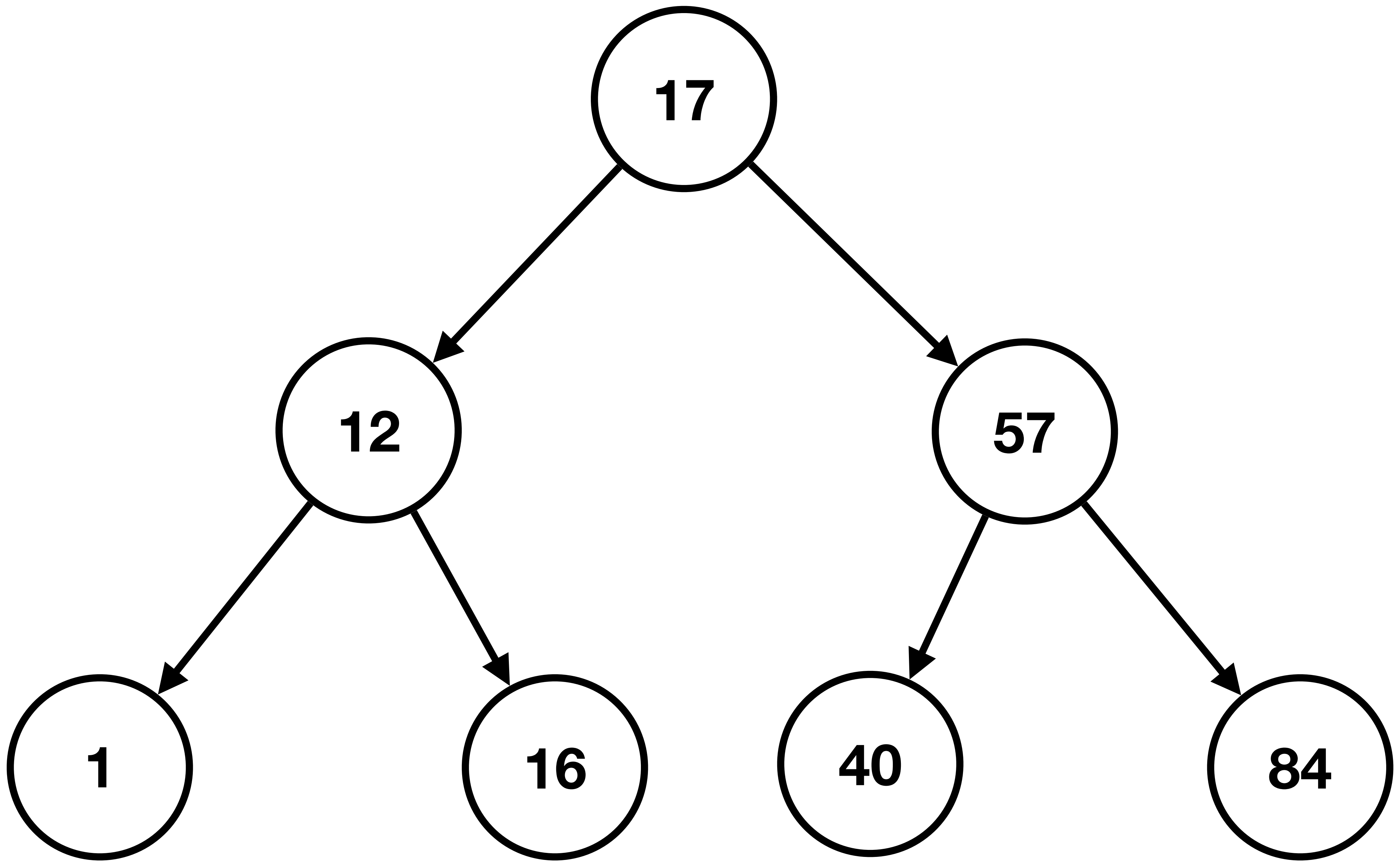
Height

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

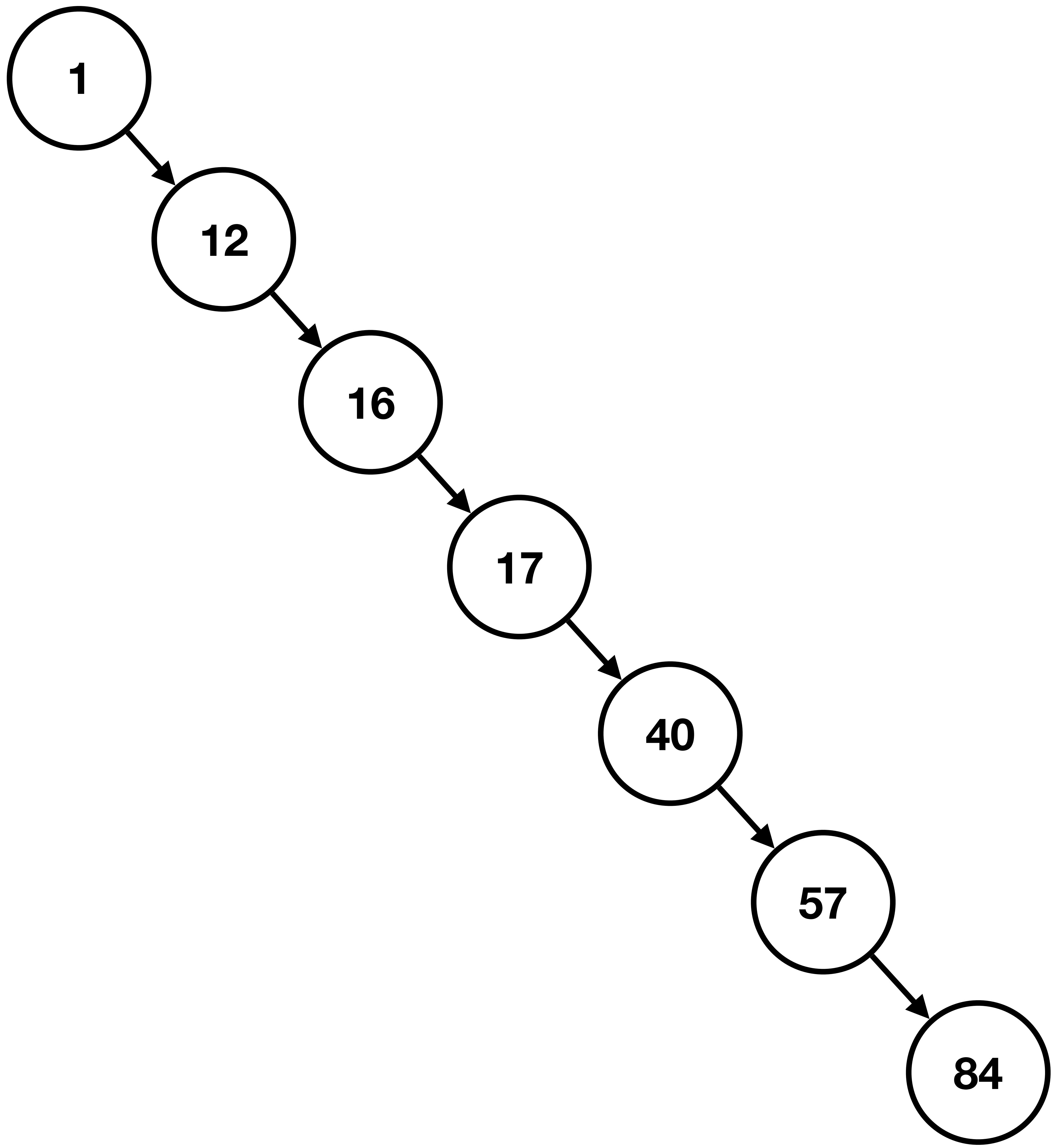


BST

Height



Balanced



Unbalanced

BST

Complexity

- 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

- 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

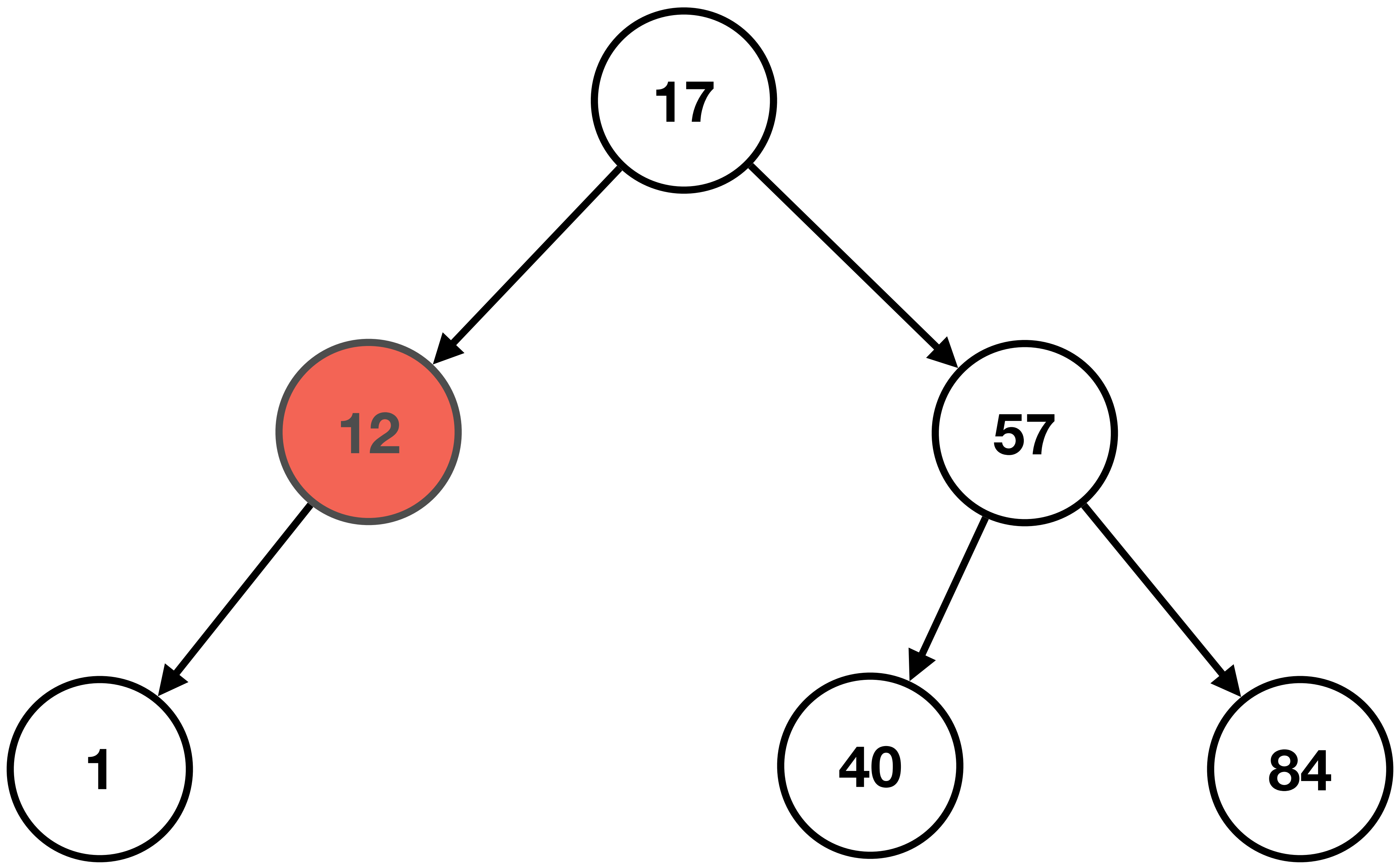
Remove

- Harder case: node to be removed has one child

BST

Remove

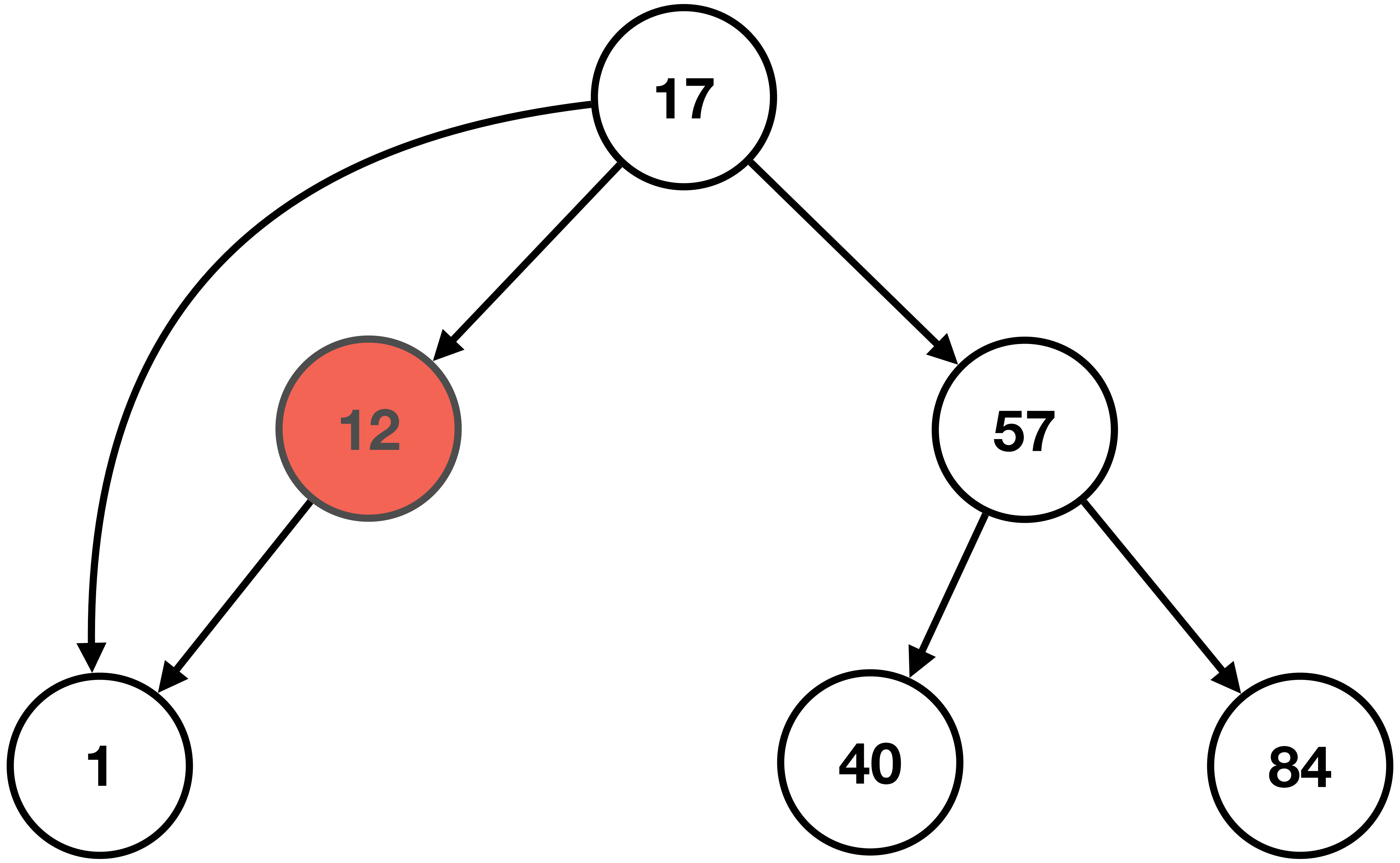
- Harder case: node to be removed has one child



BST

Remove

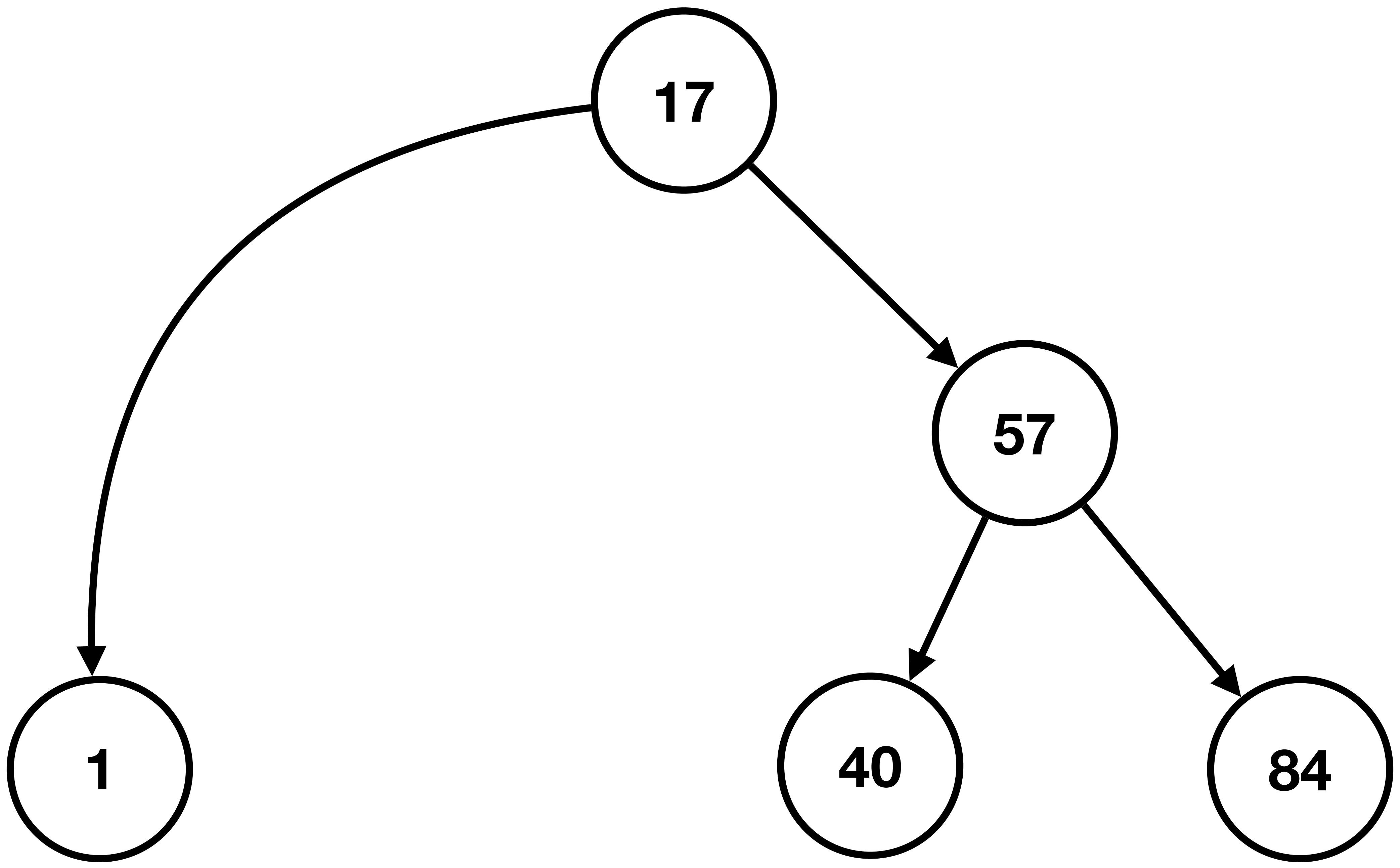
- Harder case: node to be removed has one child
  - Bypass this node



BST

Remove

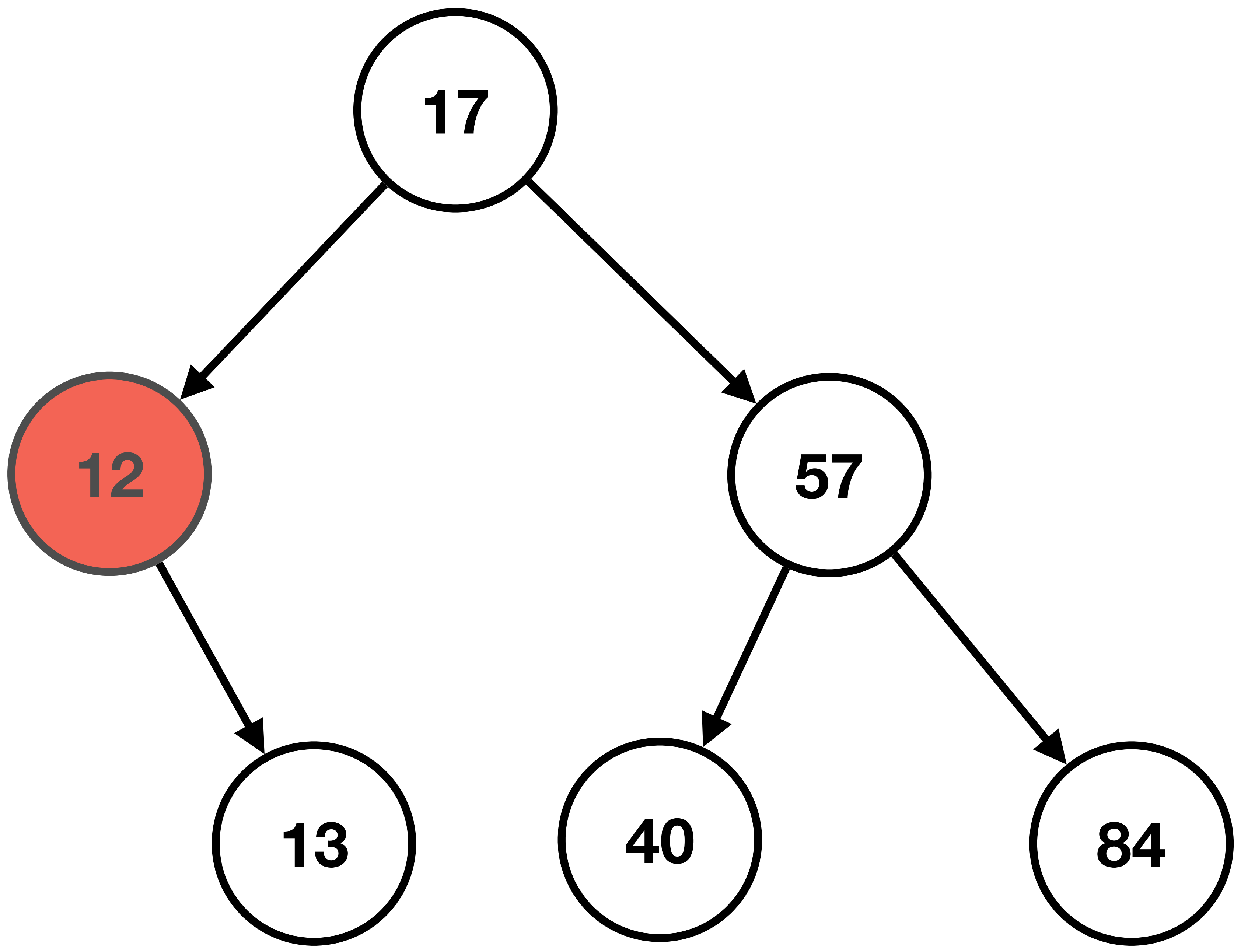
- Harder case: node to be removed has one child
  - Bypass this node



BST

Remove

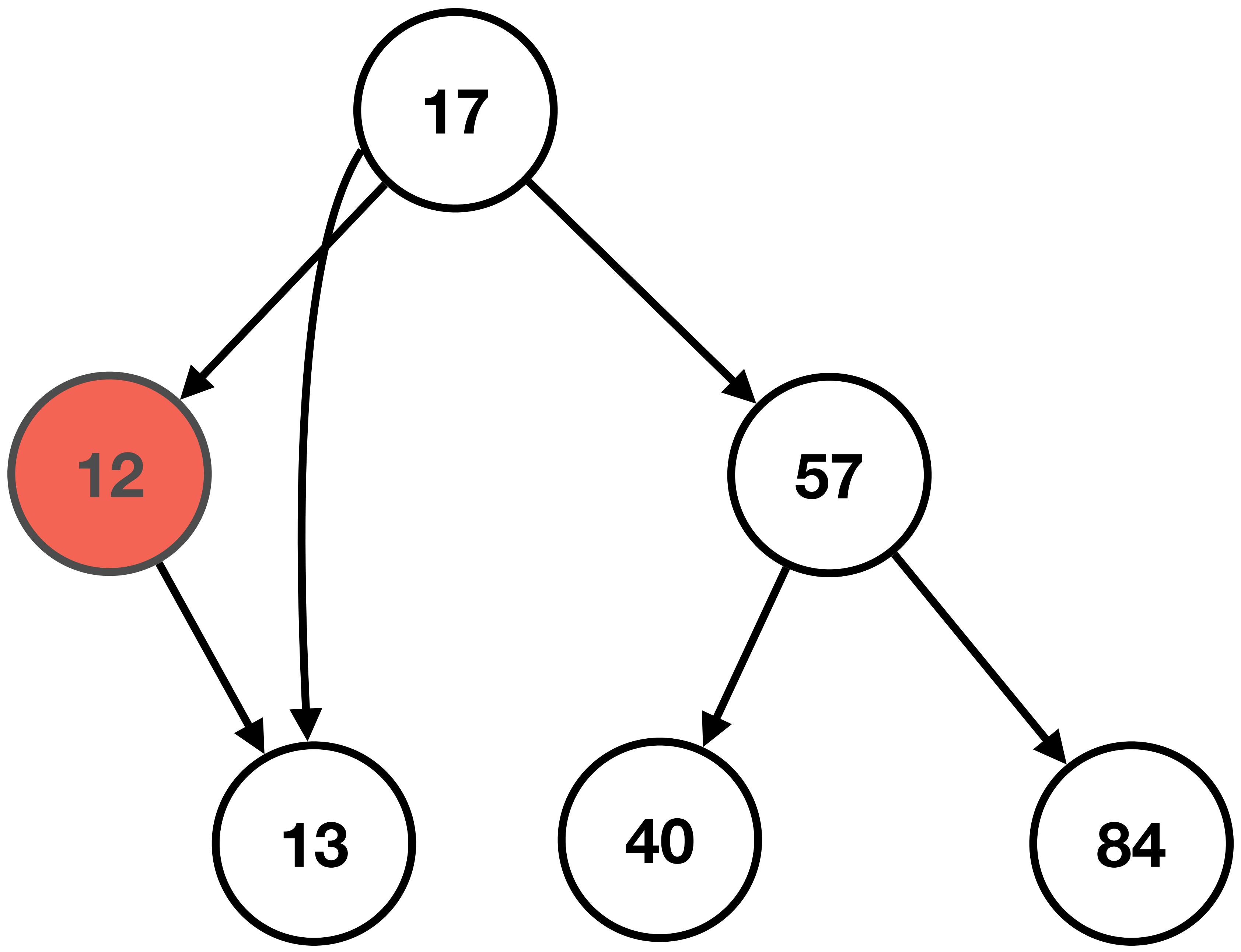
- Harder case: node to be removed has one child
  - Bypass this node



BST

Remove

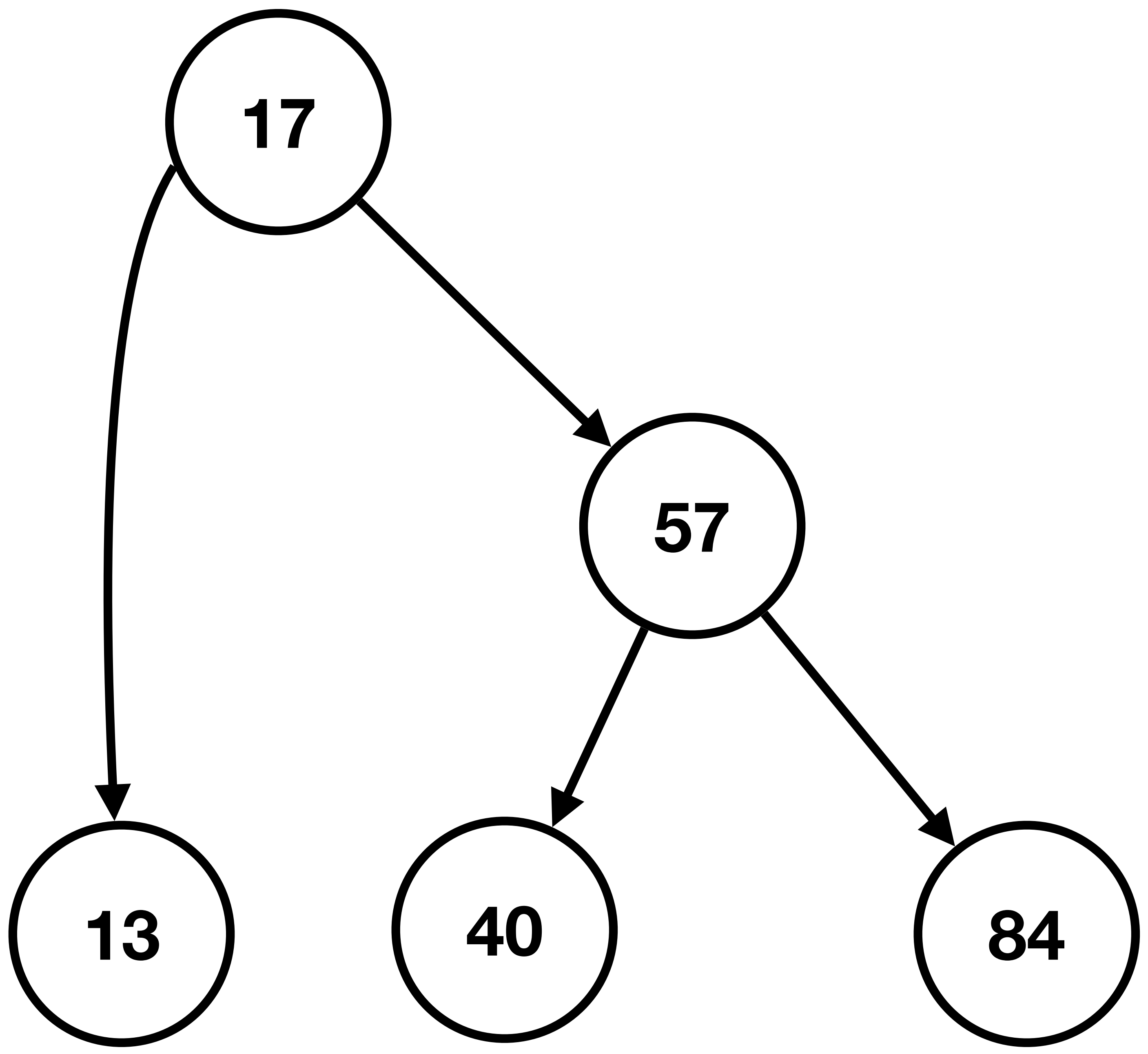
- Harder case: node to be removed has one child
  - Bypass this node



BST

Remove

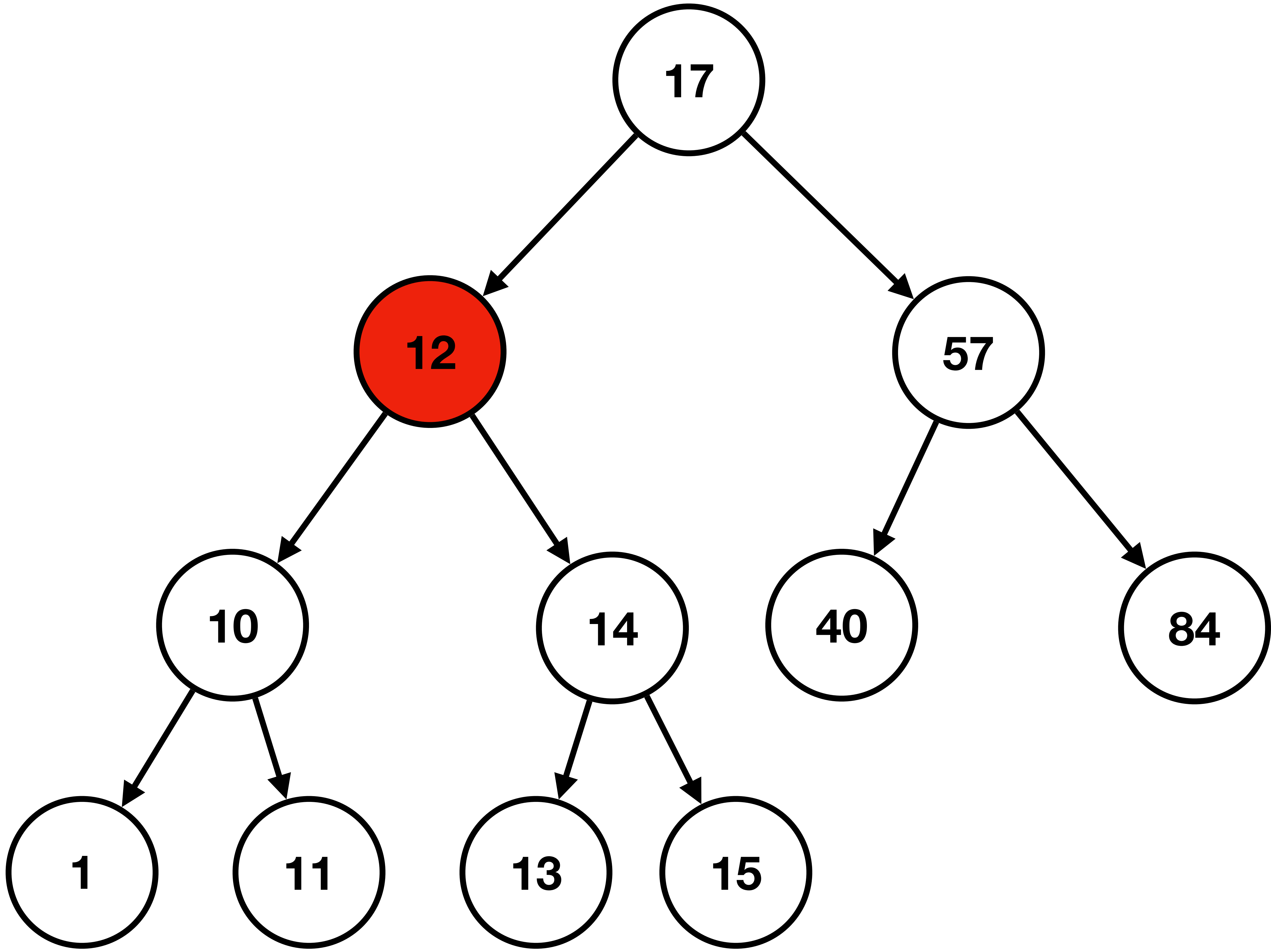
- Harder case: node to be removed has one child
  - Bypass this node



BST

Remove

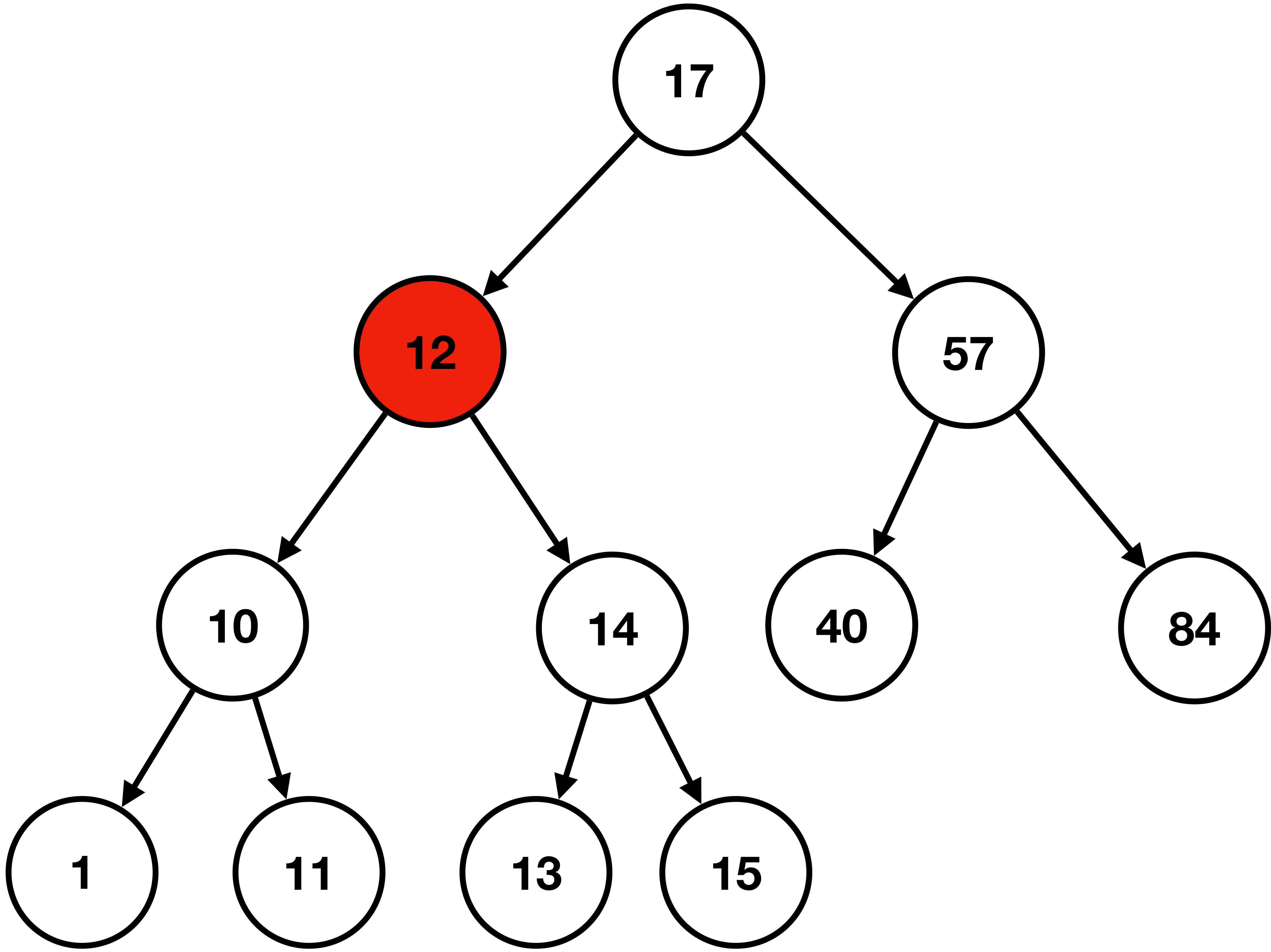
- Hardest case: node to be removed has two children



BST

Remove

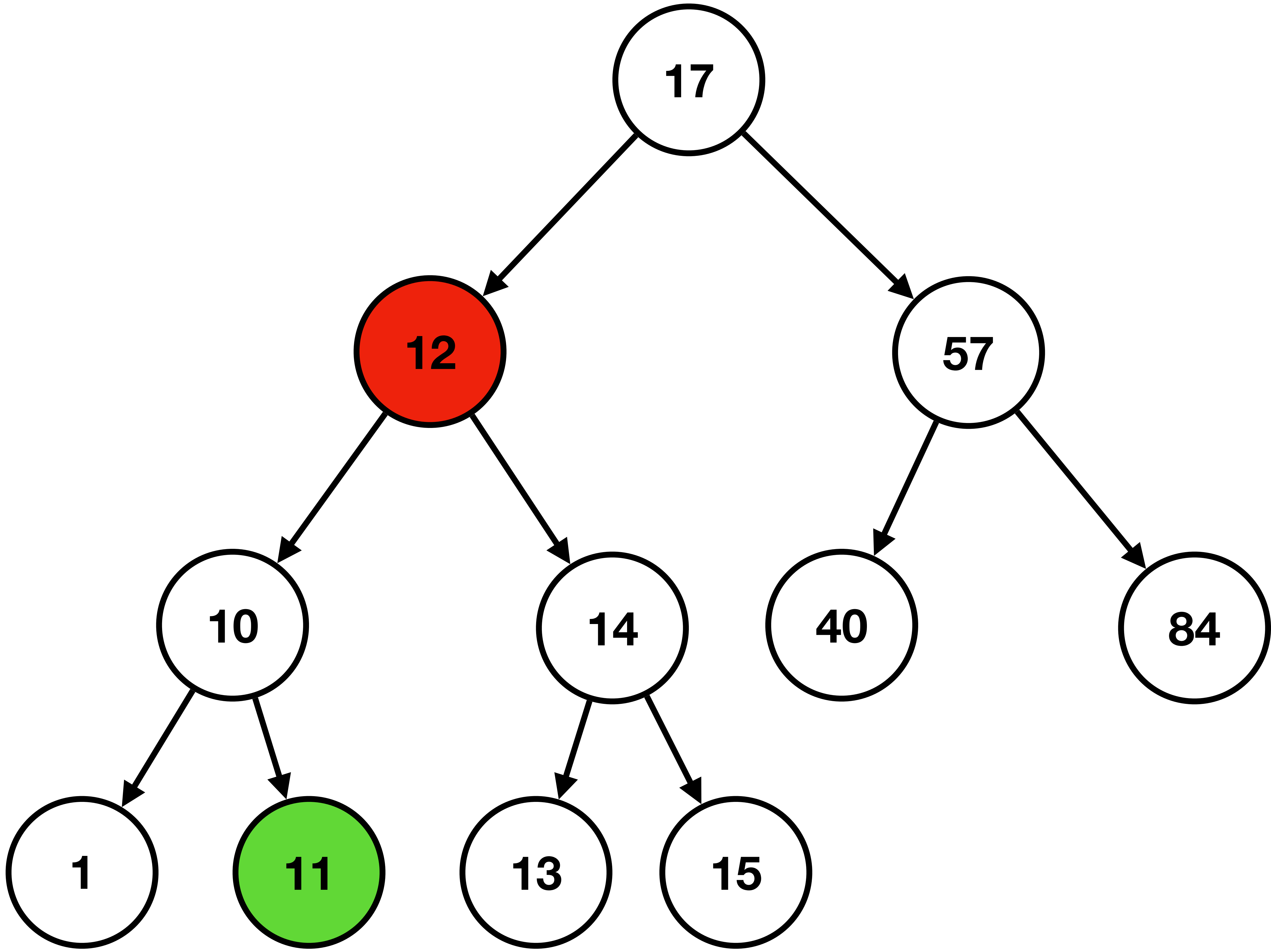
- Hardest case: node to be removed has two children
  - Replace 12 with a value that's:
    - Larger than everything in left subtree
    - Smaller than .. in right ..



BST

Remove

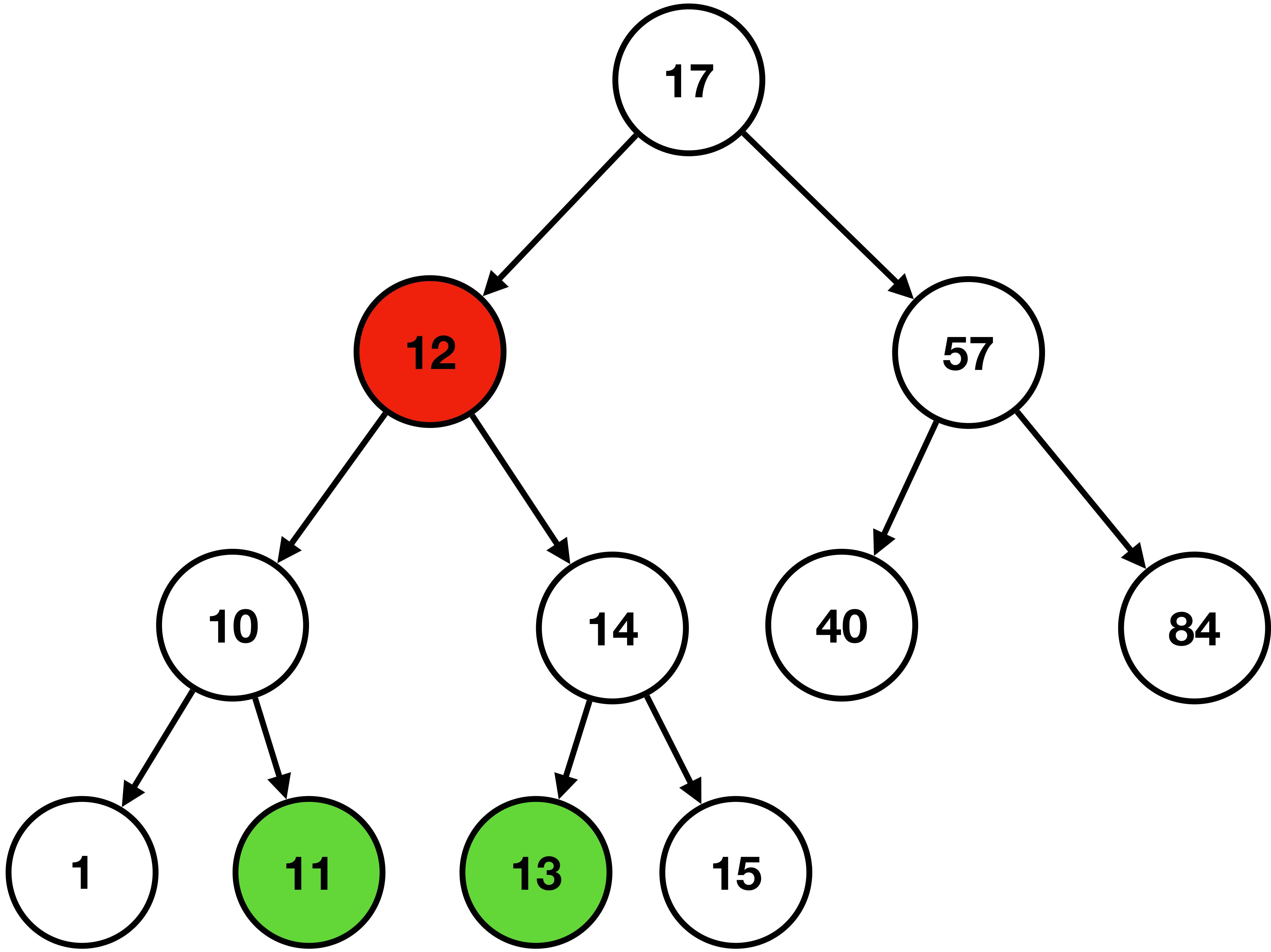
- Hardest case: node to be removed has two children
  - Replace 12 with a value that's:
    - Larger than everything in left subtree
    - Smaller than .. in right ..



BST

Remove

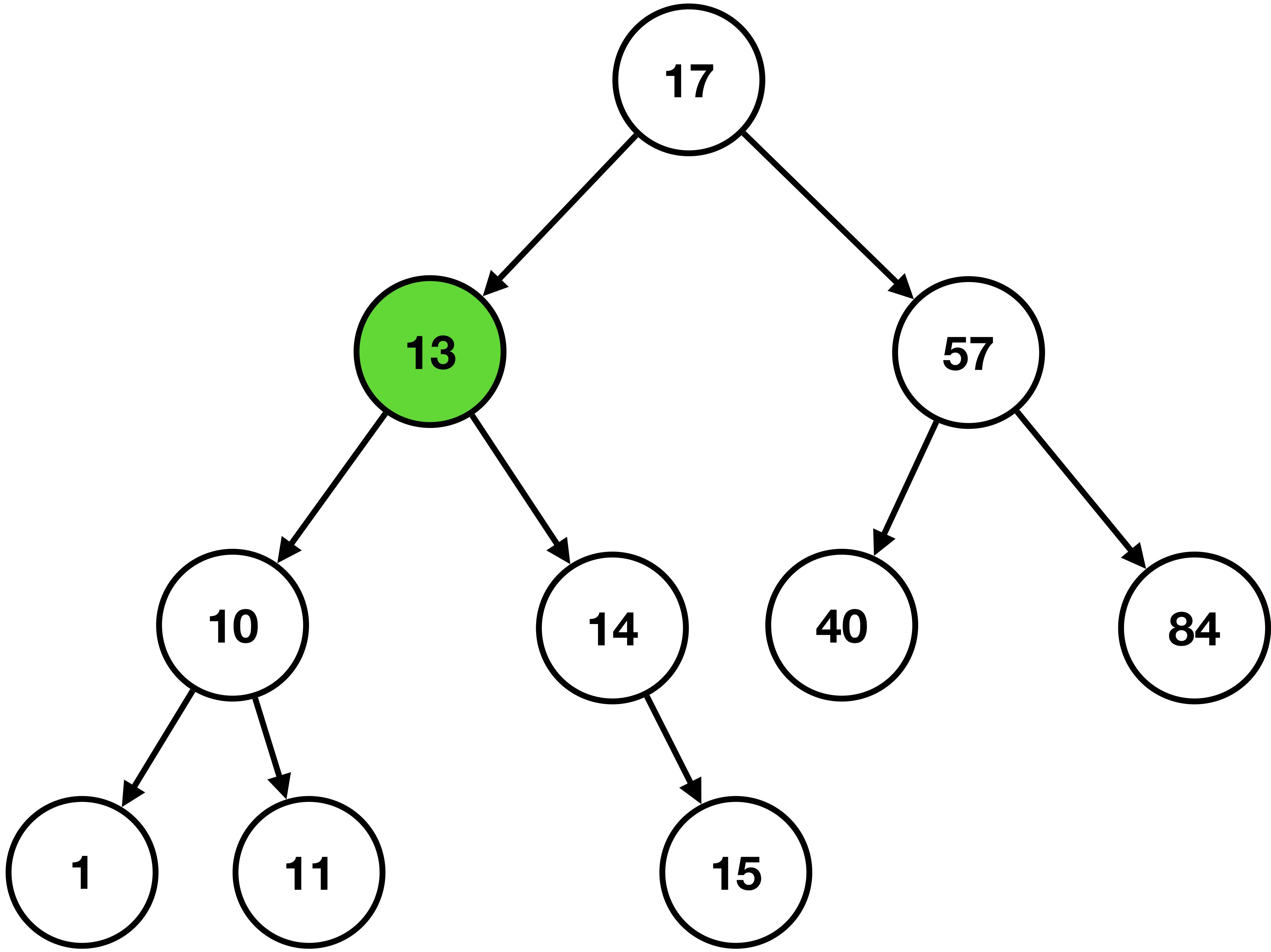
- Hardest case: node to be removed has two child
    - Replace 12 with a value that's:
        - Larger than everything in left subtree
        - Smaller than .. in right ..



BST

Remove

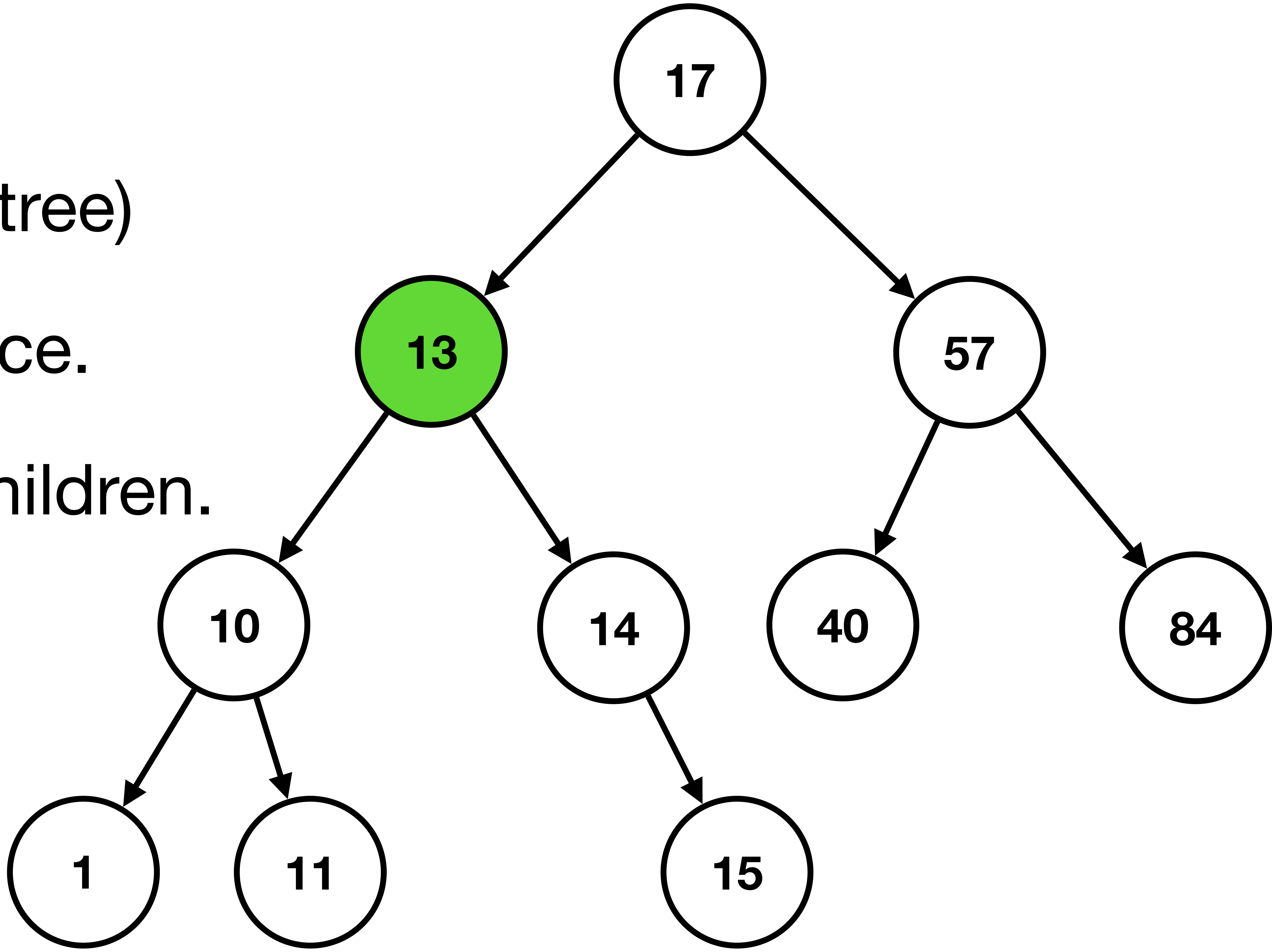
- Hardest case: node to be removed has two child
  - Replace 12 with a value that's:
    - Larger than everything in left subtree
    - Smaller than .. in right ..



BST

Remove

- Hardest case: node to be removed has two children
  - 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

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 **<-- 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

Overall complexity:

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

Same as insert and lookup.

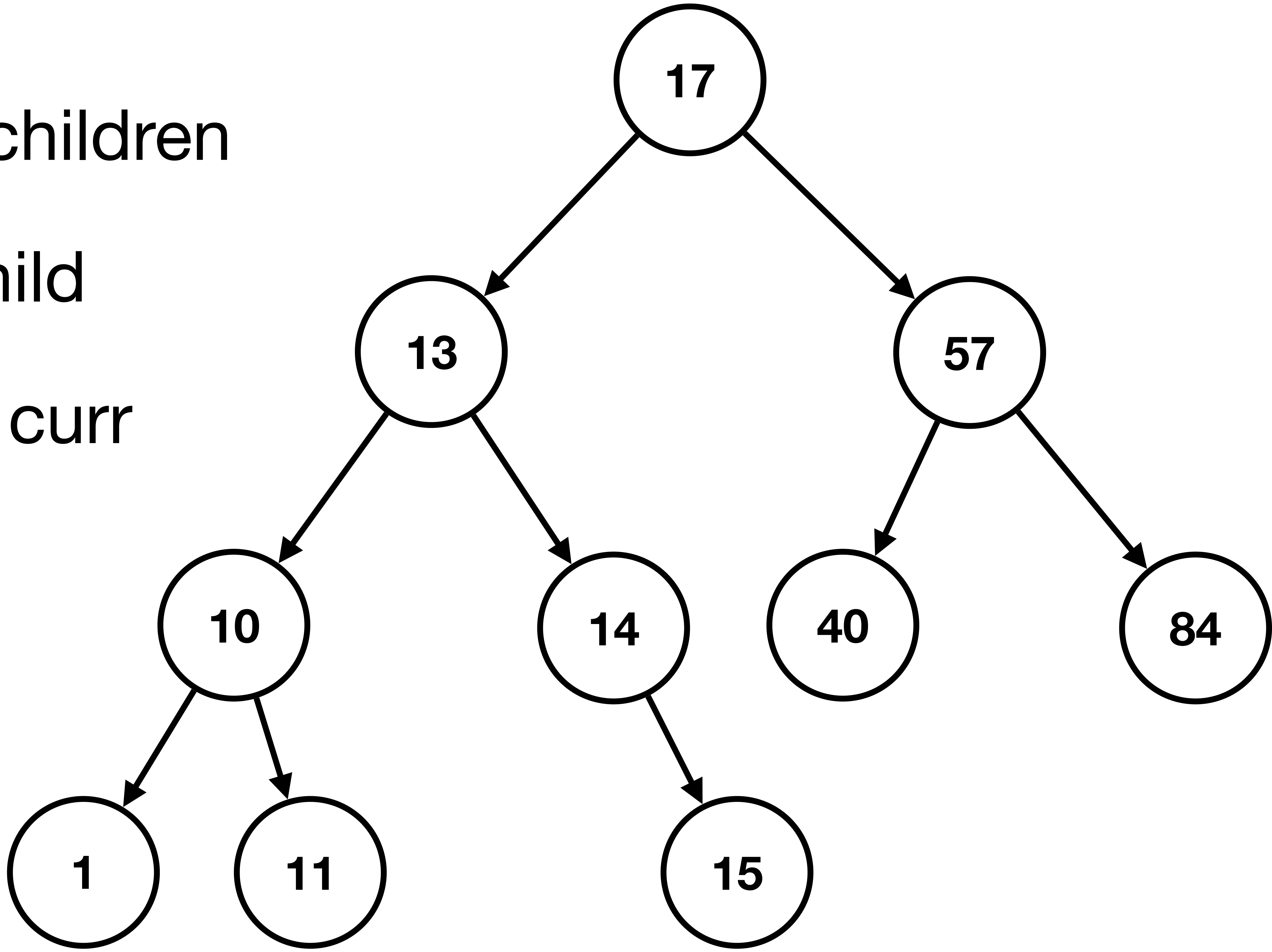
Maps

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

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)
{
        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);
}
```

Back to sorting

Tree Sort

Tree sort:

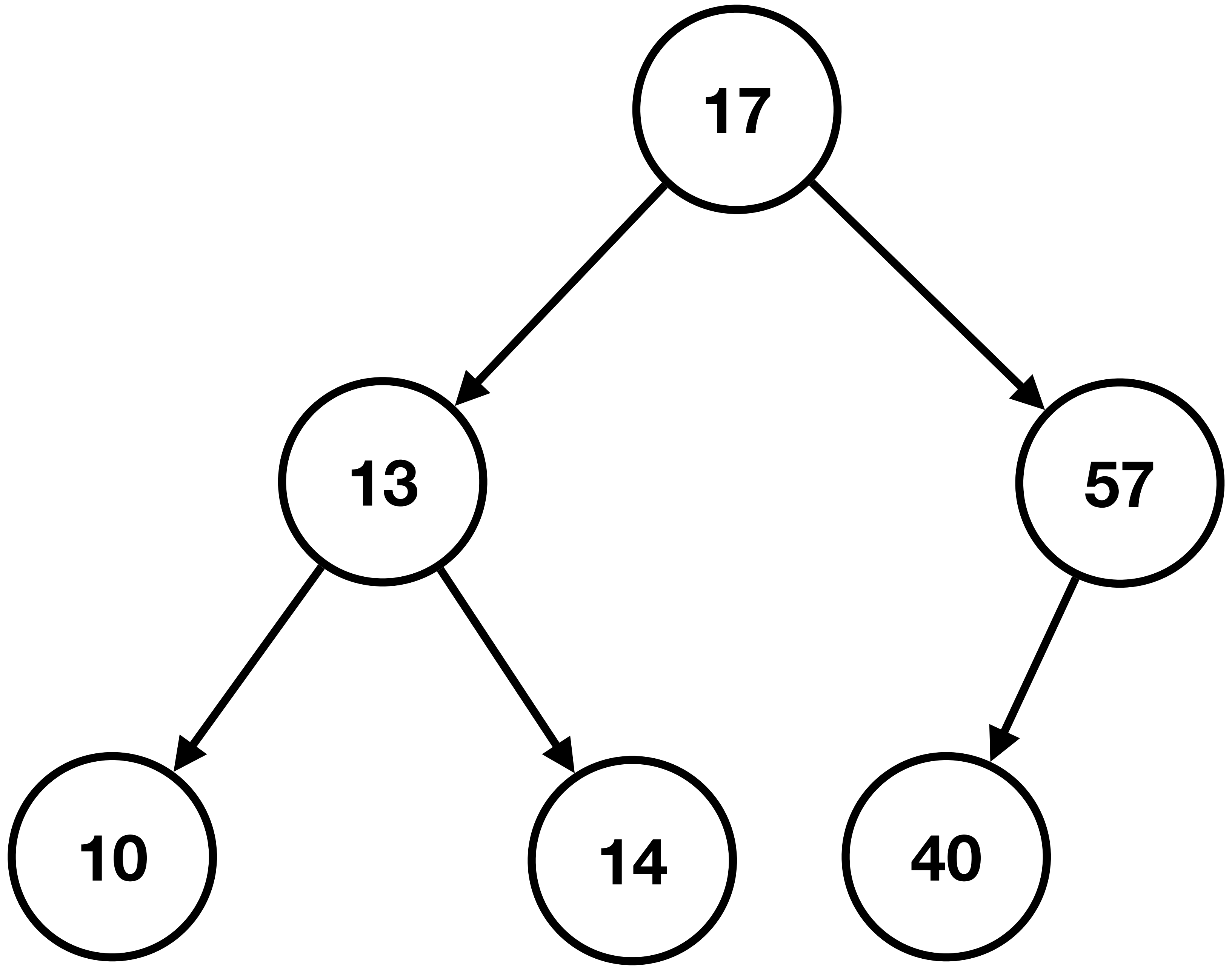
- Insert each elements into a (self-balancing) BST -- $n \cdot O(\log n)$
- In-order walk over the tree -- $O(n)$
- Overall: $O(n \log n)$!
- But we need extra memory
- Also balancing is costly

Sorting
Heap

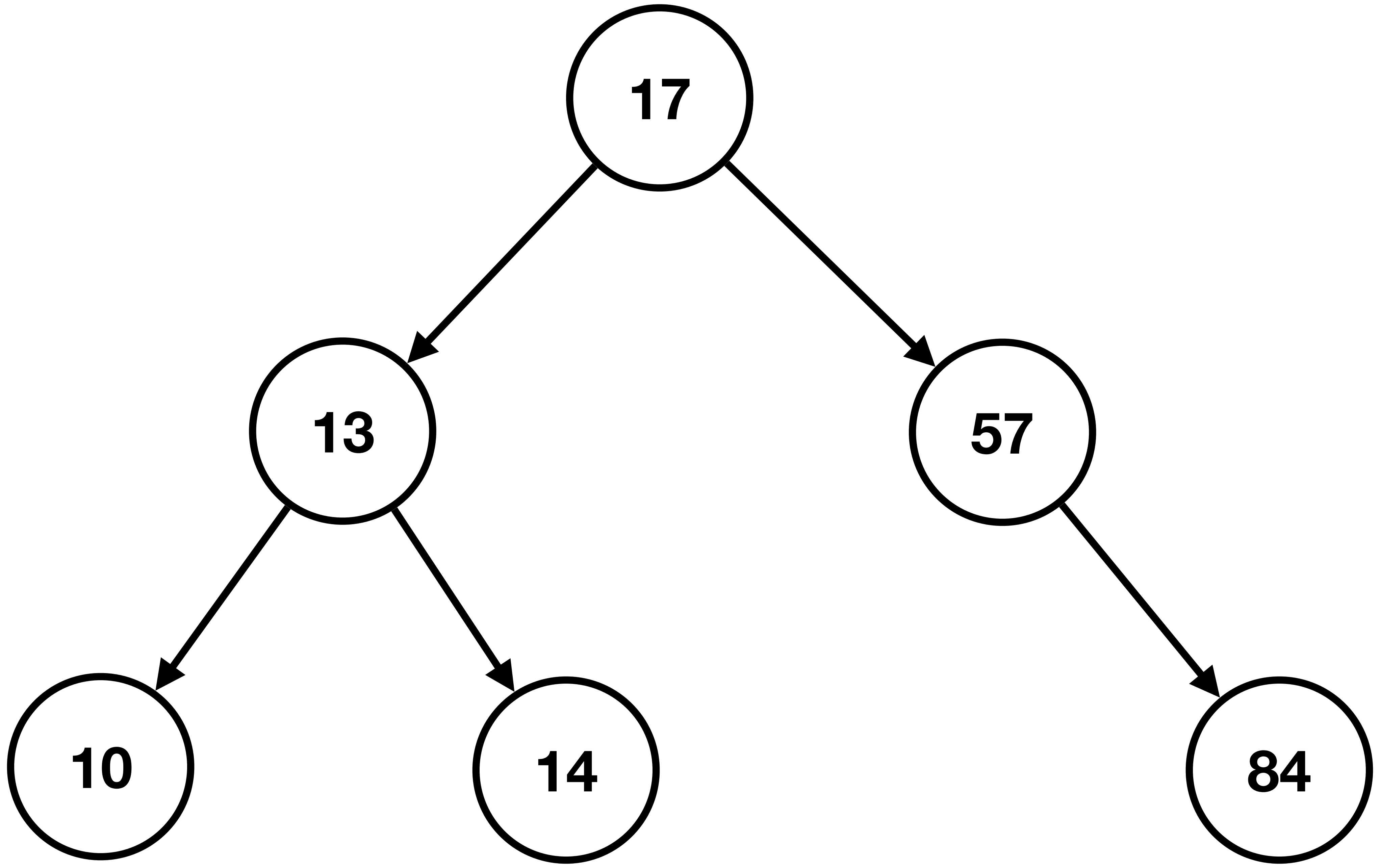
- BST requirements:
  - all nodes on the left < root < all nodes on the right
- A new arrangement:
  - parent is less than any children.
  - order between children does not matter.
  - extra requirement: the tree is *complete*
    - each level except the lowest is full, lowest level fills from the left

Sorting

Heap



Complete

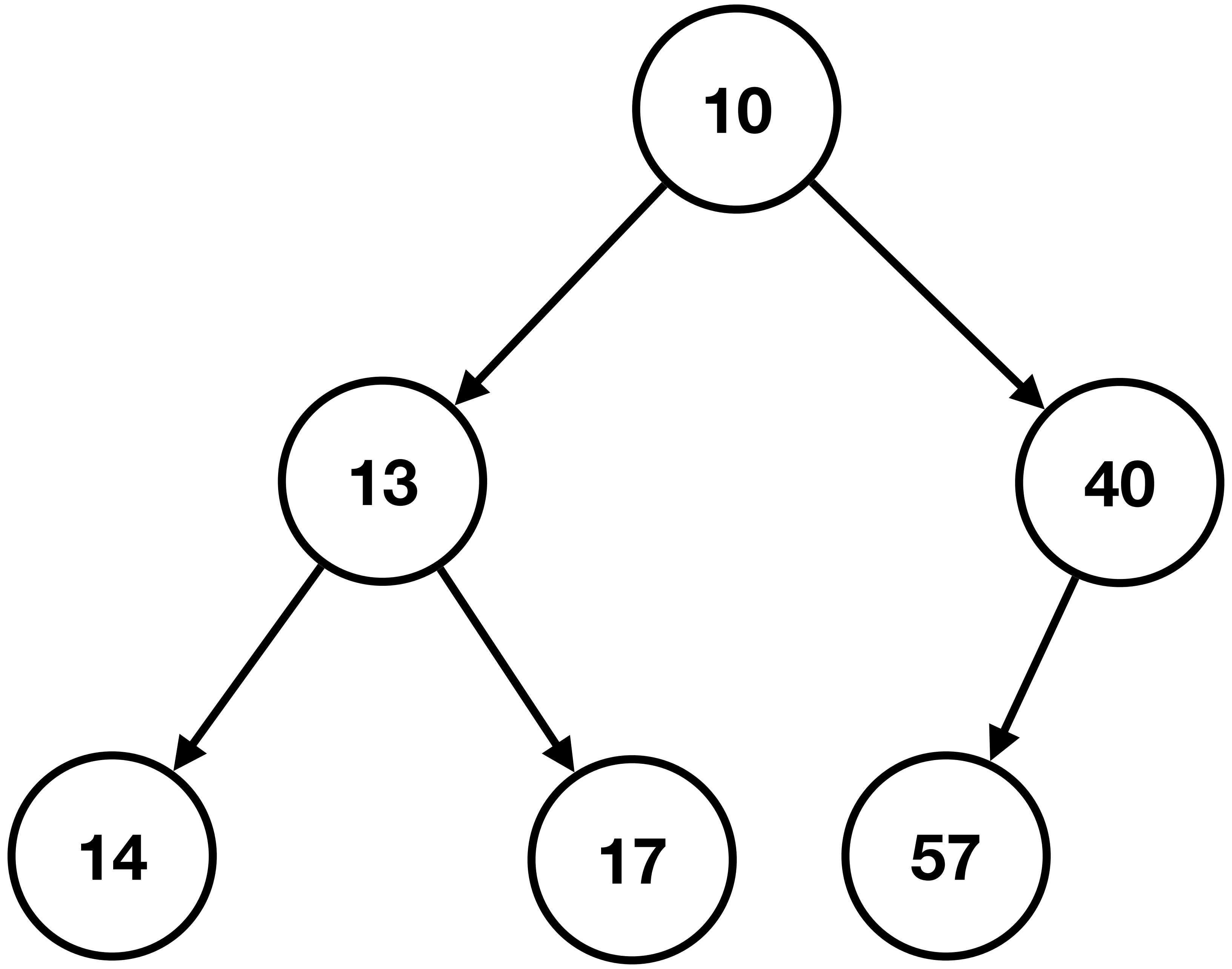


Incomplete

Sorting

What is the best way to store this?

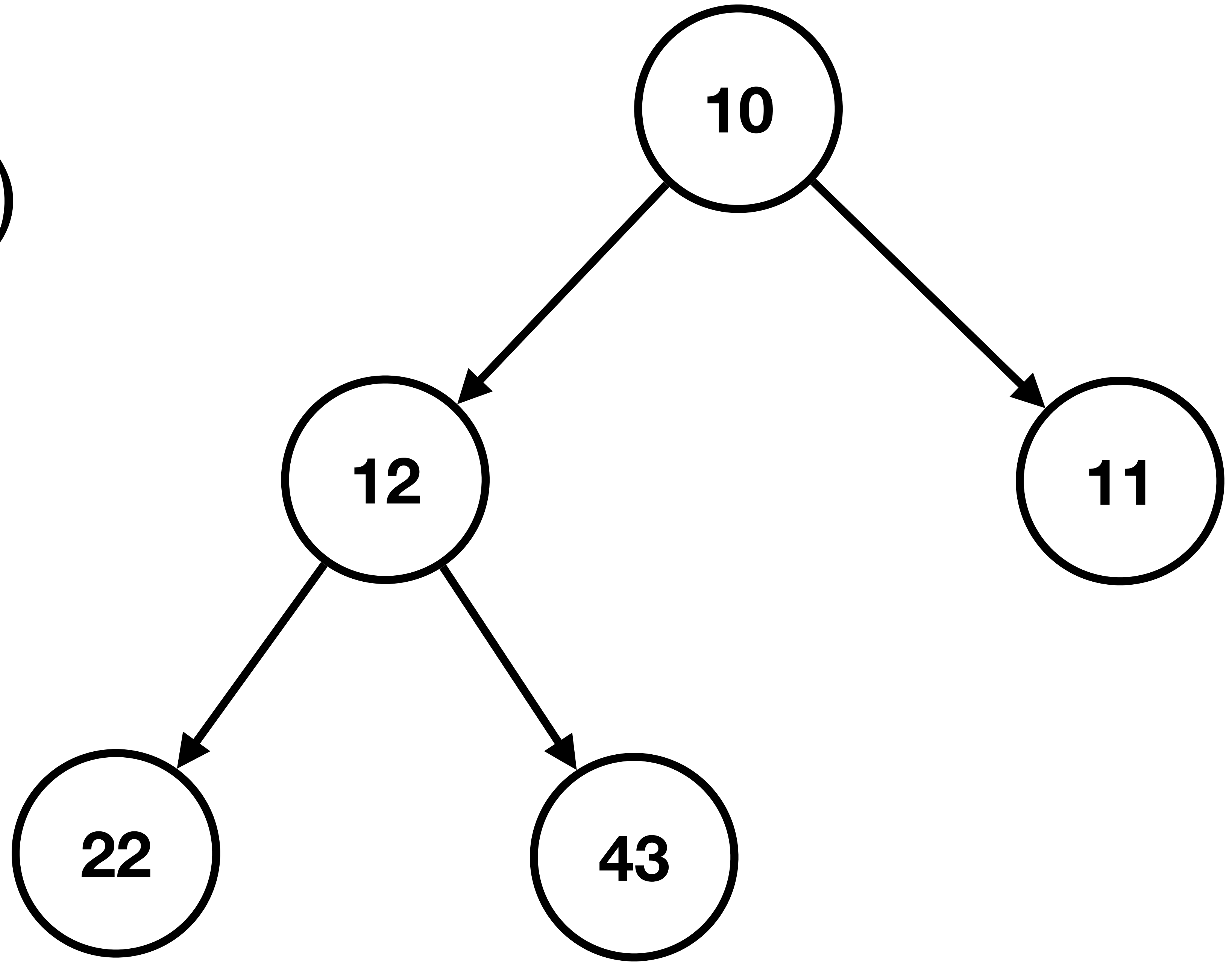
- Could use nodes and pointers...
- Or, we can use a data structure that provides constant-time access to elements:
- array



Sorting

What is the best way to store this?

- Start array at 1 instead of 0 (make math easier)



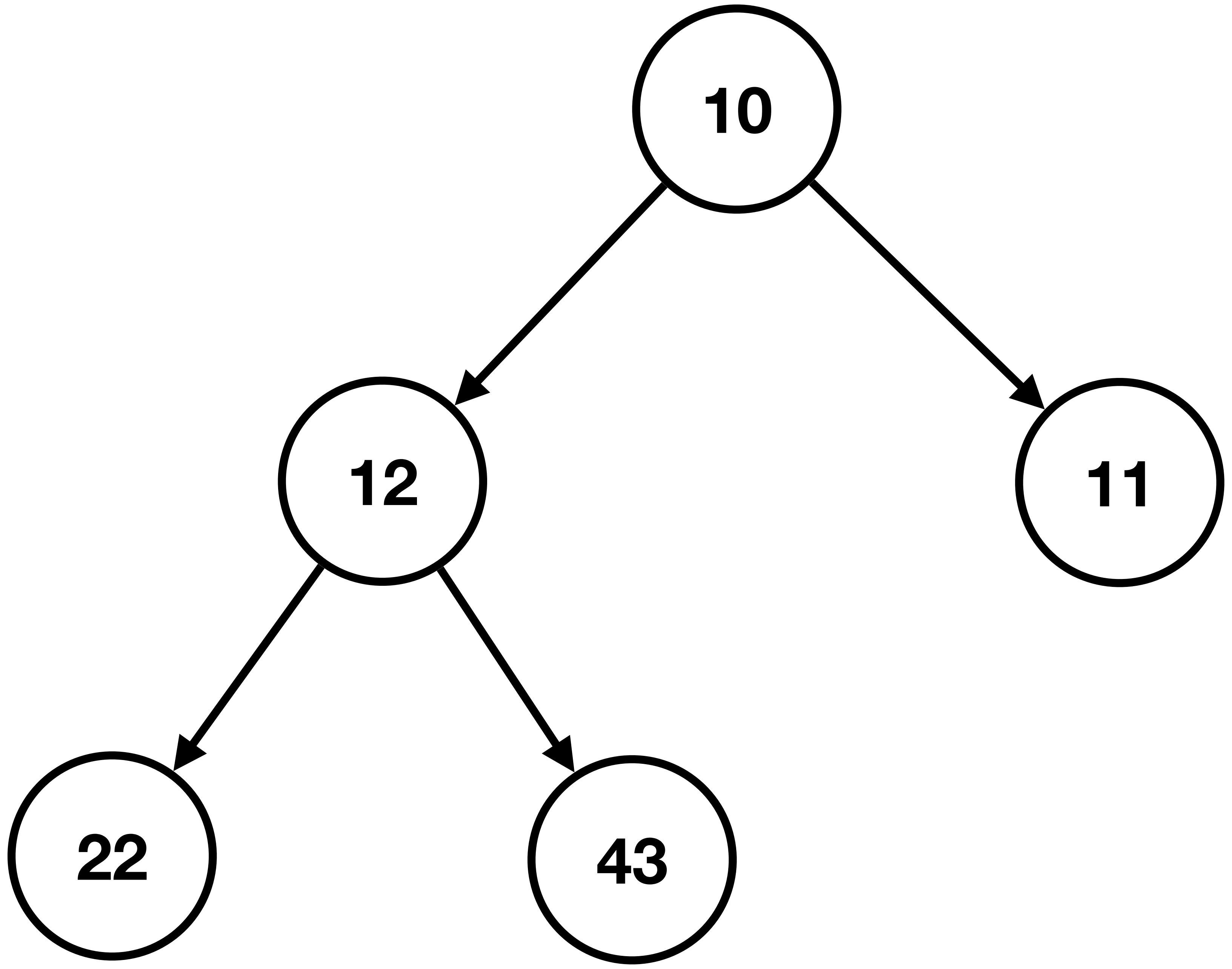
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	10	12	11	22	43					

Sorting

What is the best way to store this?

- Start array at 1 instead of 0 (make math easier)
- For an element at position i:
    - left child: $2i$
    - right child: $2i + 1$
    - parent: $\lfloor i/2 \rfloor$

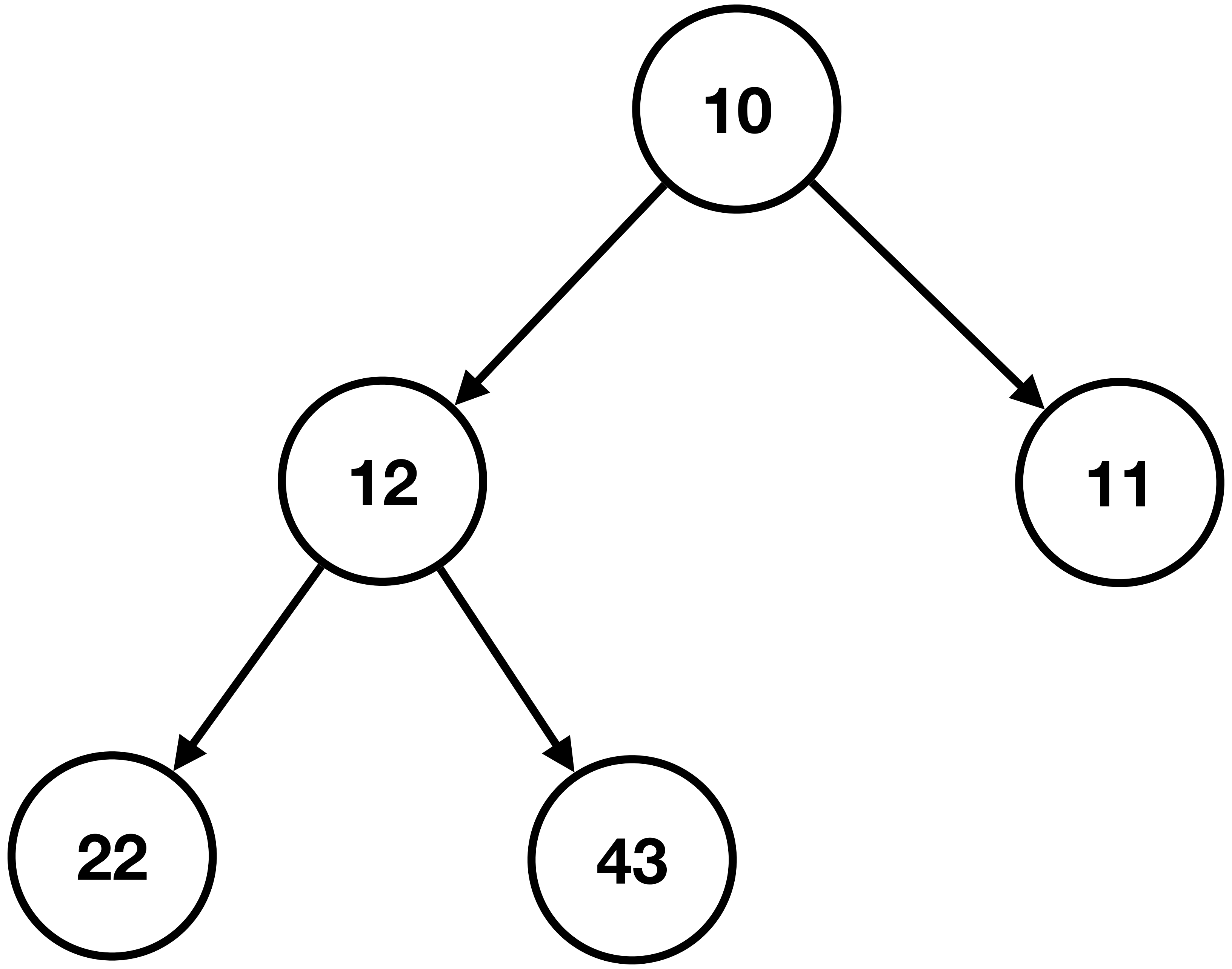
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	10	12	11	22	43					



Sorting

Heap

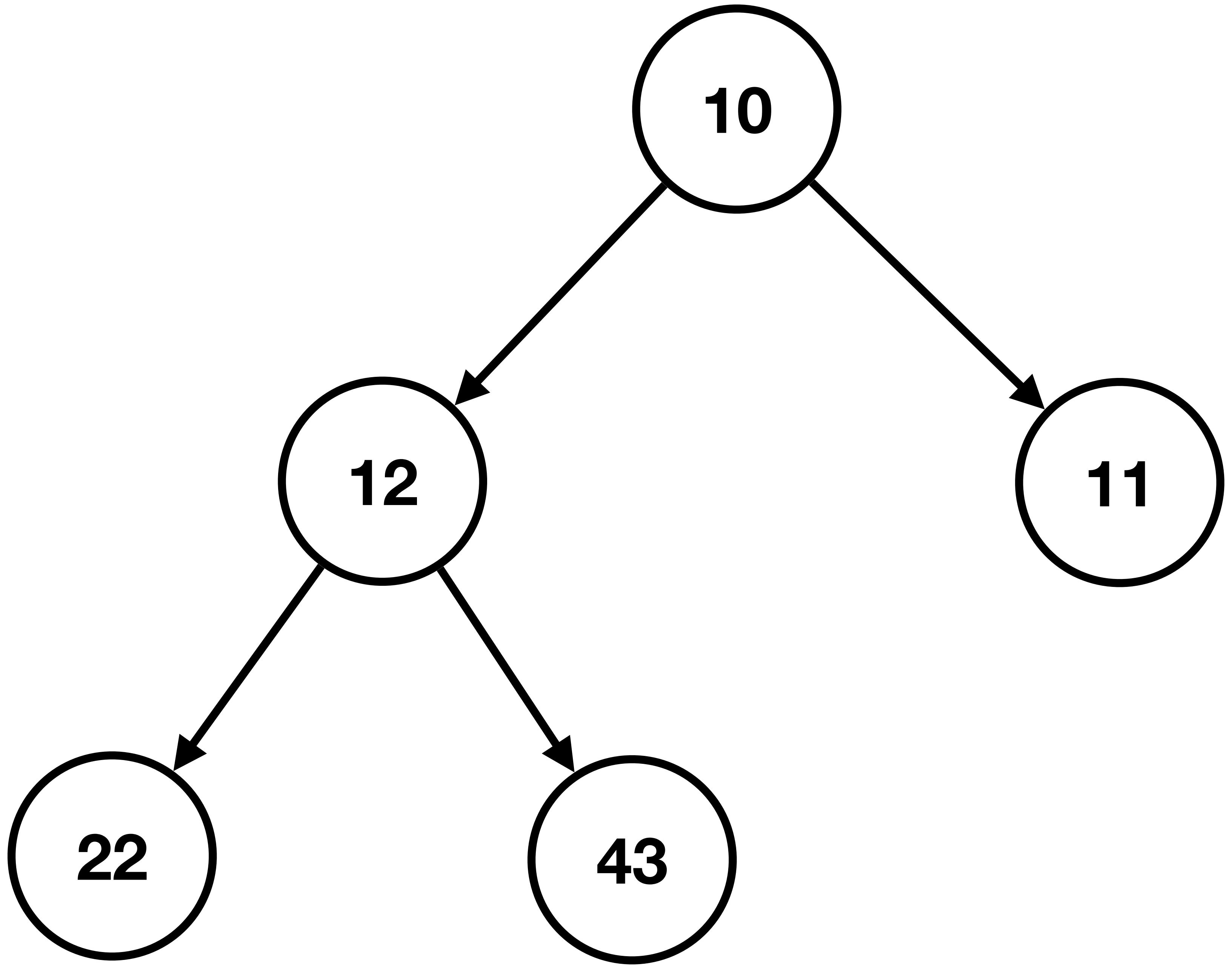
- This is called a heap:
  - Note: this is *data structure* heap, and has nothing to do with *memory* heap.
- Comes in two flavors:
  - Min Heap (smaller on top)
  - Max Heap (larger on top)



Sorting

Heap

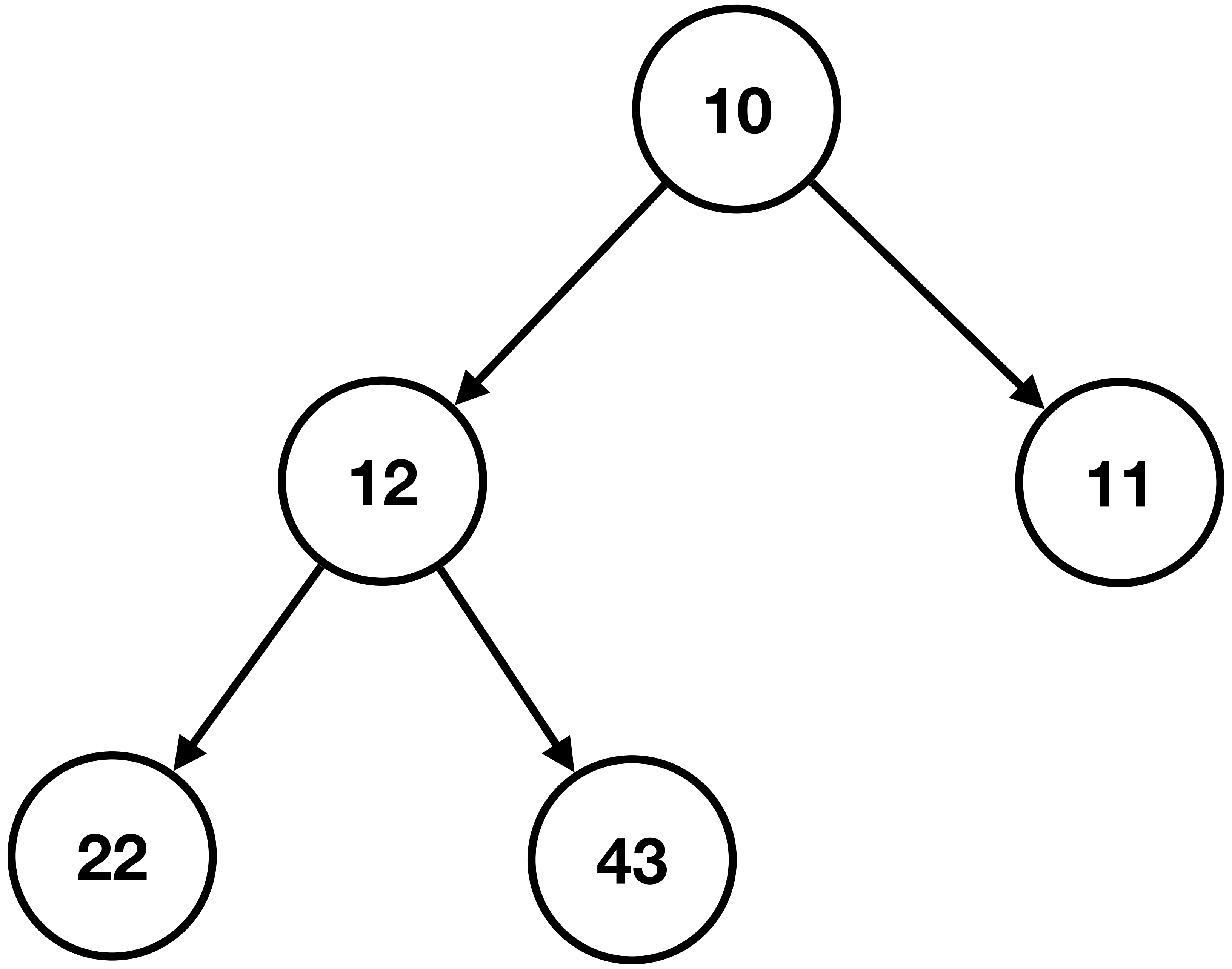
- Heap Operations:
  - `get_min`
  - `insert`
  - `remove_min`



Sorting

`get_min`

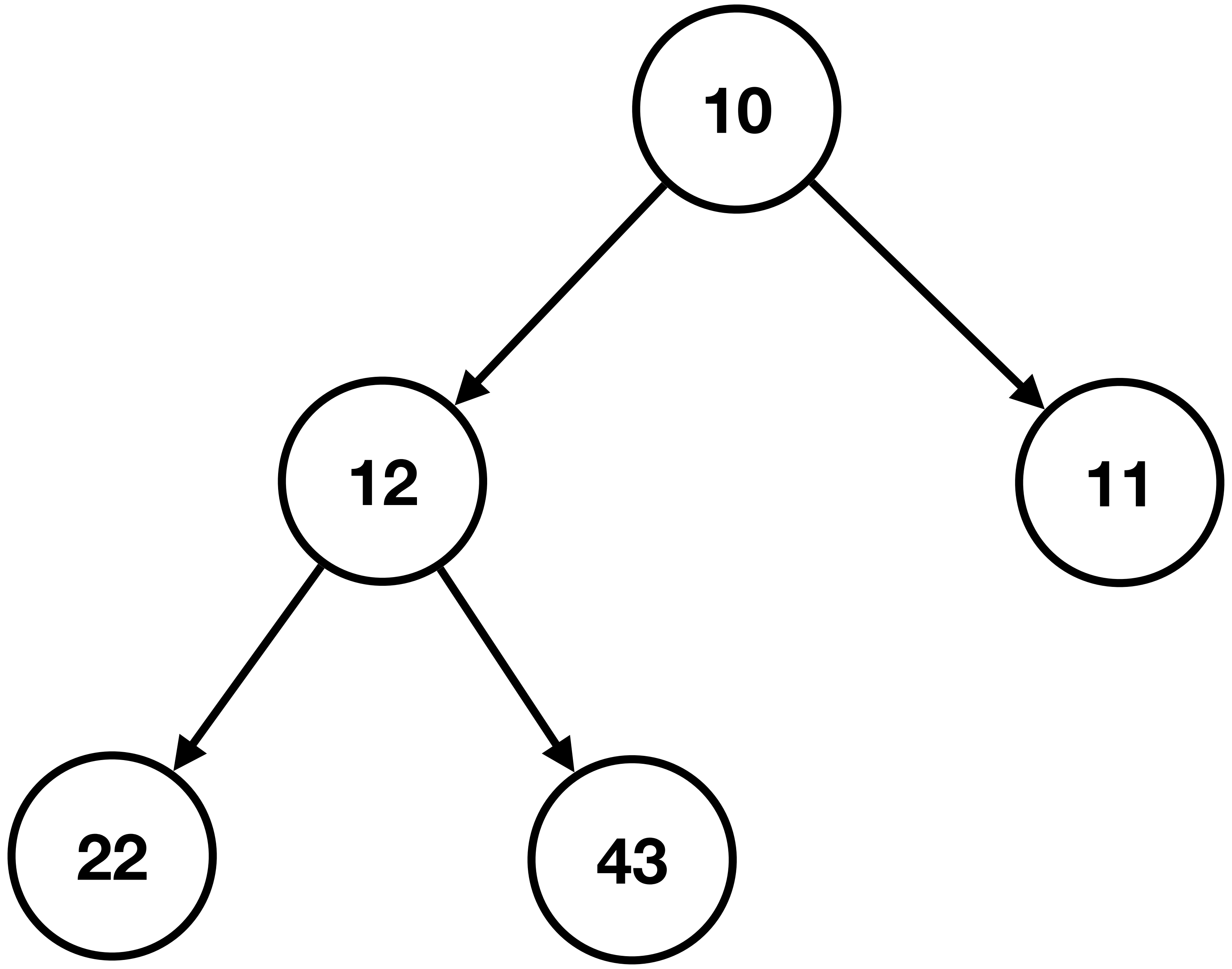
- return `h[1]`
- `O(1)`



Sorting

`insert(k)`

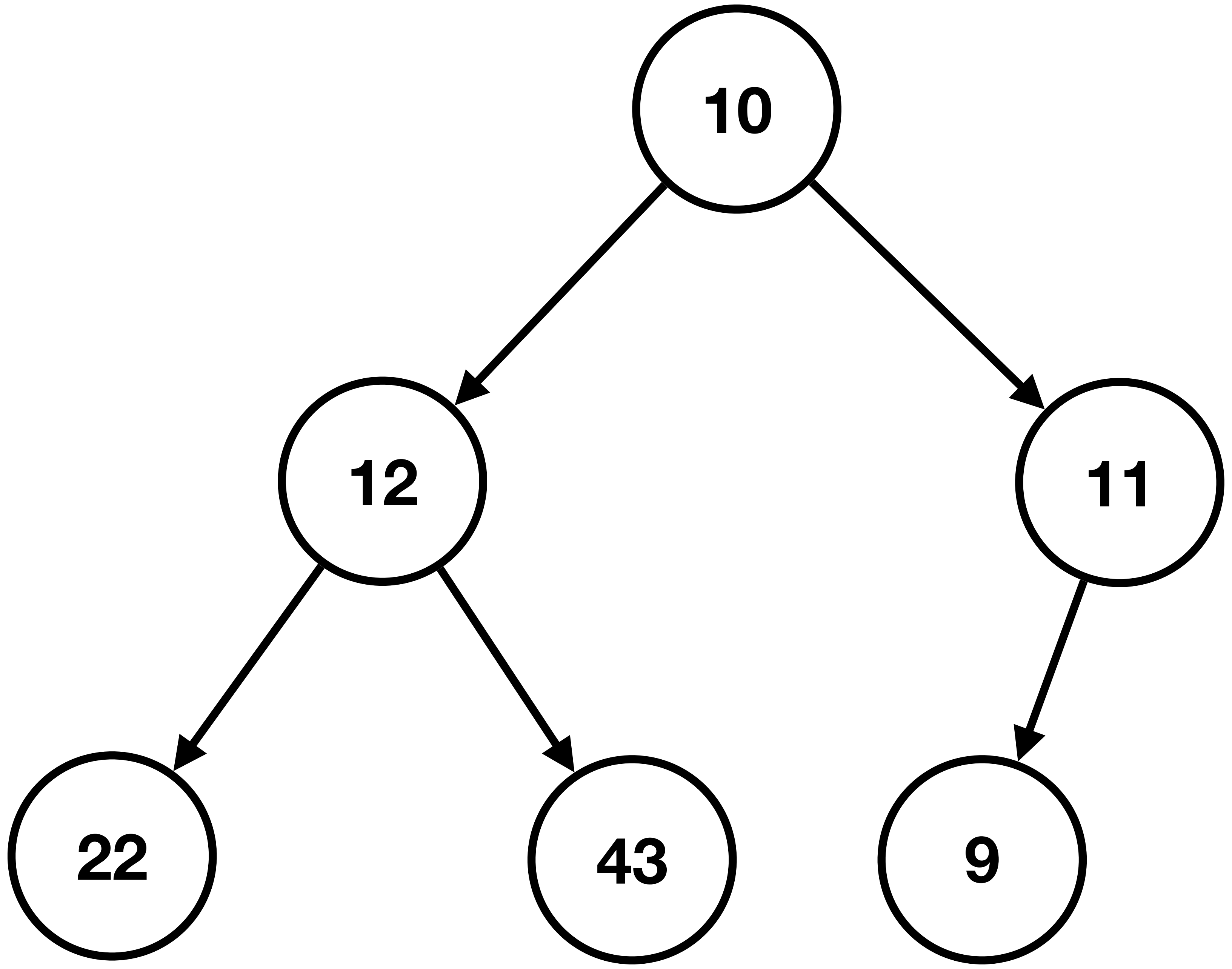
- How might we go about inserting 9?



Sorting

`insert(k)`

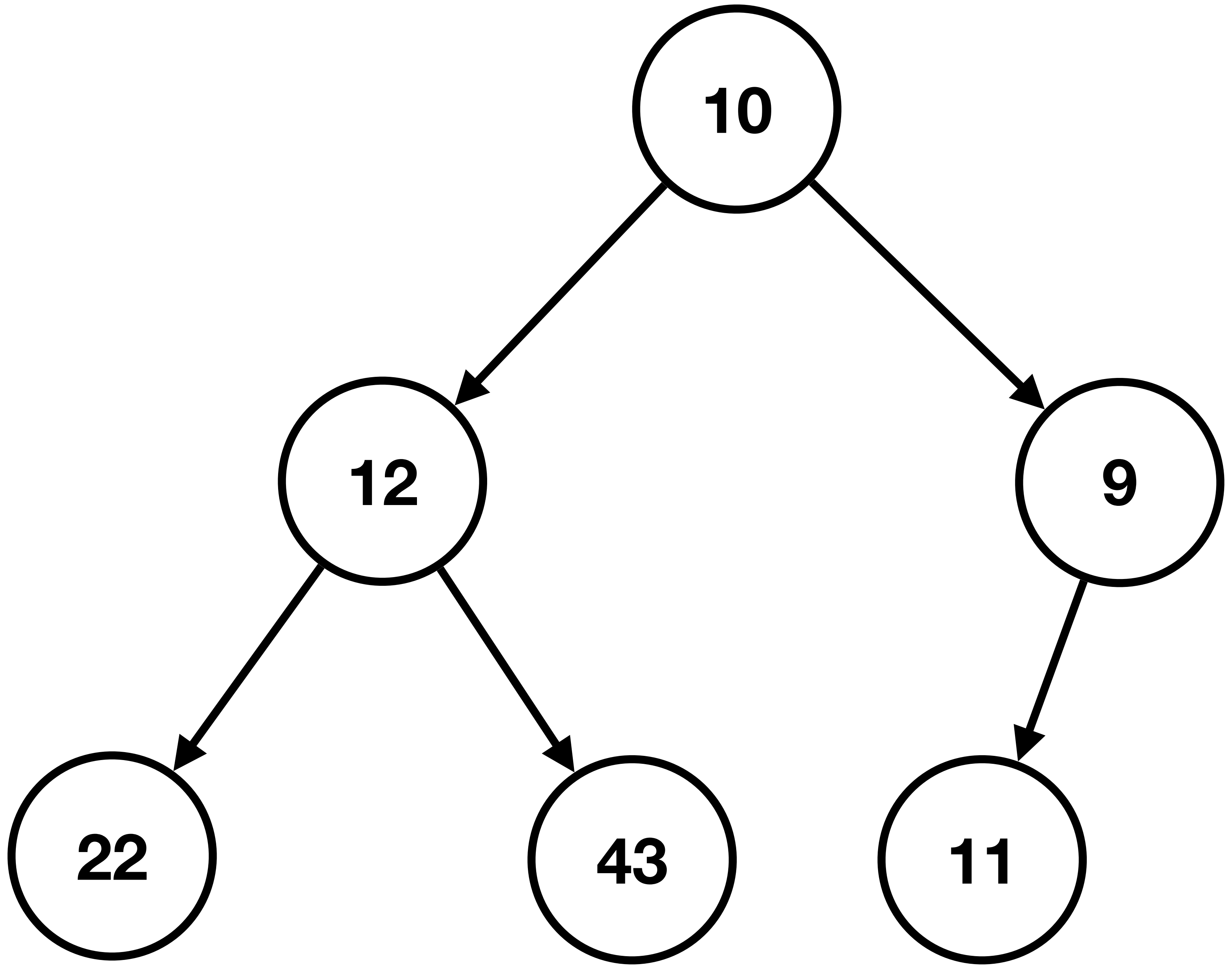
- How might we go about inserting 9?
  - Insert at h[size + 1]
  - bubble up until you get to root
    - Compare with its parent, if in correct order, stop
    - If not, swap and continue going up



Sorting

`insert(k)`

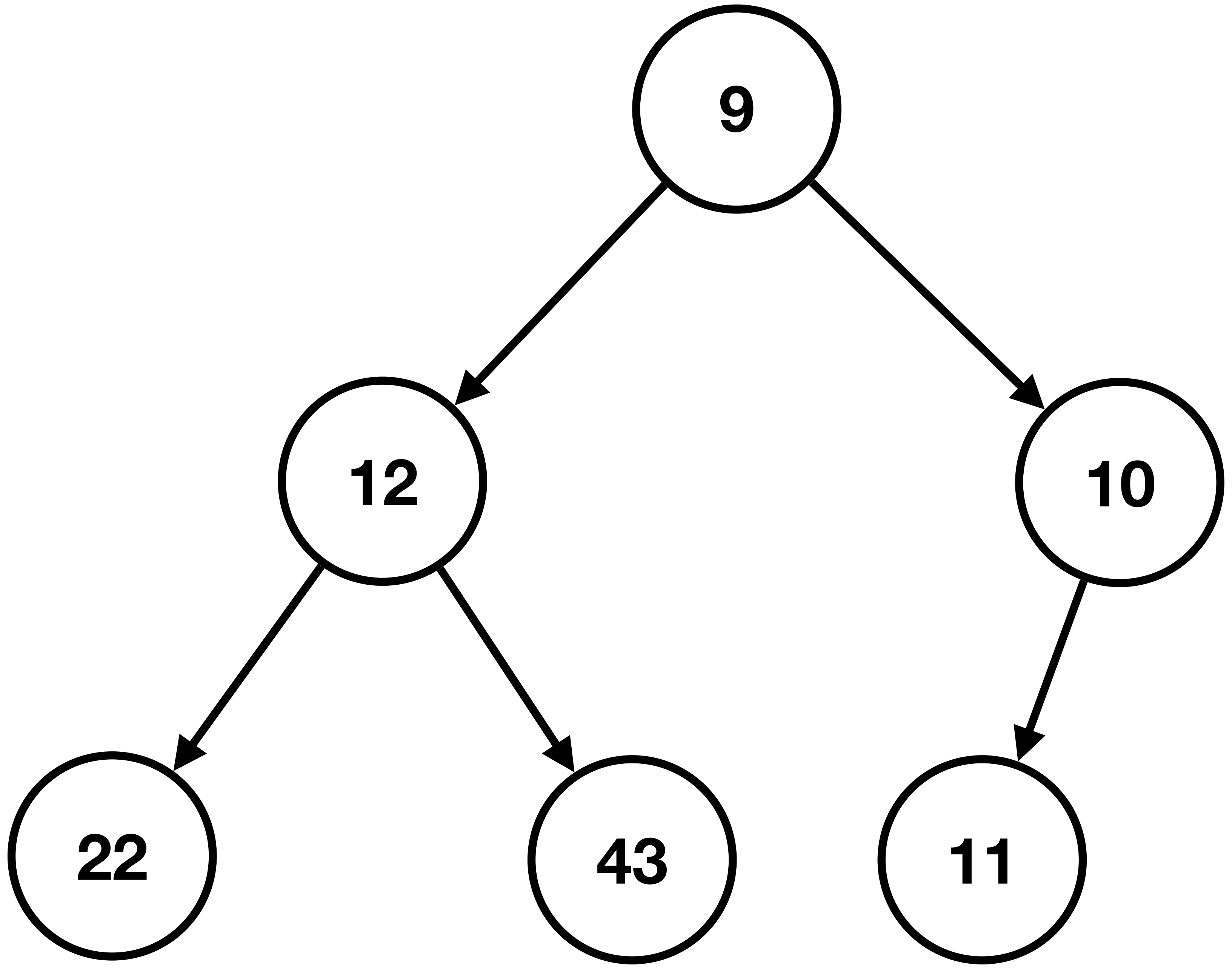
- How might we go about inserting 9?
  - Insert at h[size + 1]
  - bubble up until you get to root
    - Compare with its parent, if in correct order, stop
    - If not, swap and continue going up



Sorting

`insert(k)`

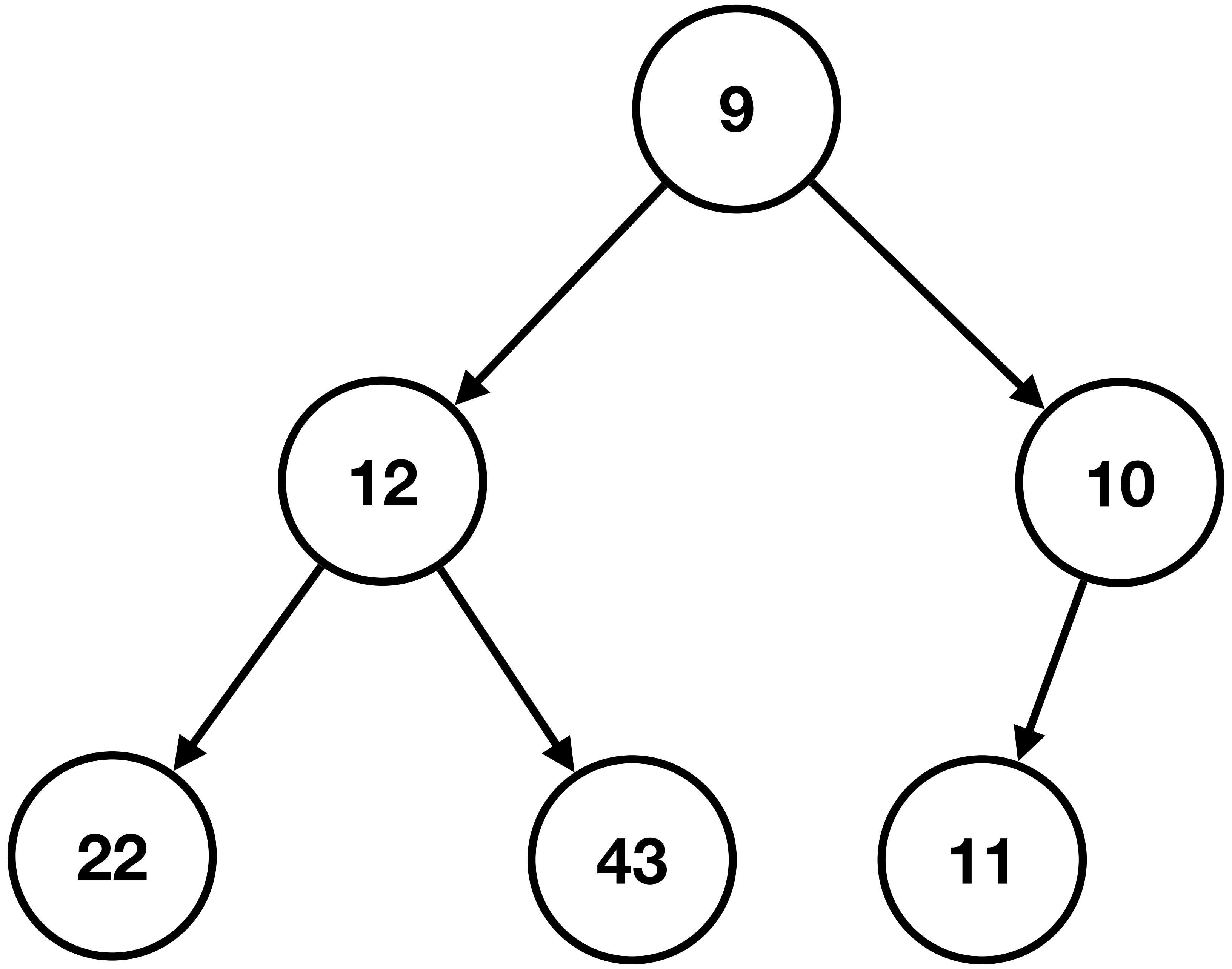
- How might we go about inserting 9?
  - Insert at h[size + 1]
  - bubble up until you get to root
    - Compare with its parent, if in correct order, stop
    - If not, swap and continue going up



Sorting

`insert(k)`

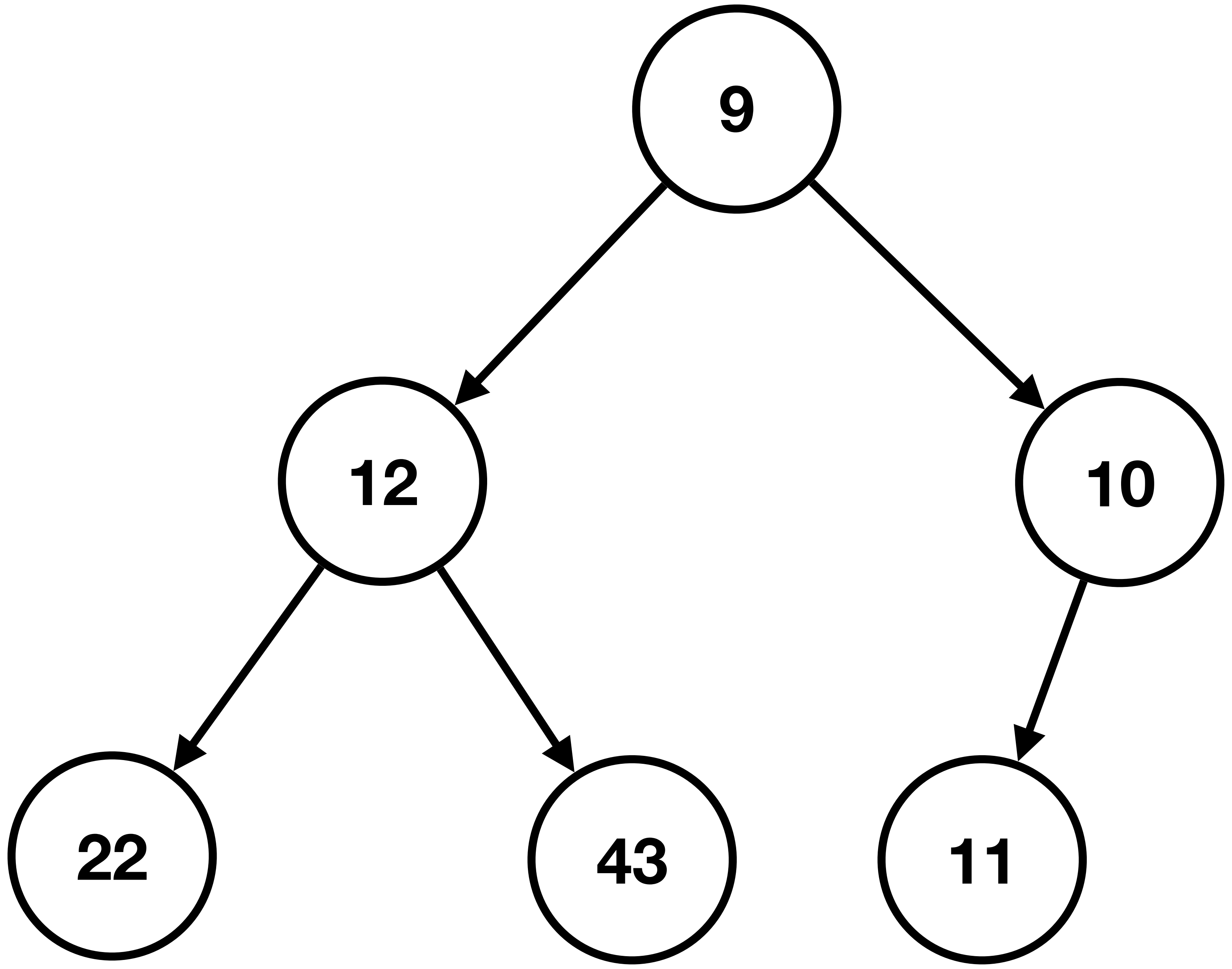
- How might we go about inserting 9?
  - Insert at h[size + 1]
  - bubble up until you get to root
    - Compare with its parent, if in correct order, stop
    - If not, swap and continue going up
- Complexity: O(height) = O(log n)



Sorting

`remove_min()`

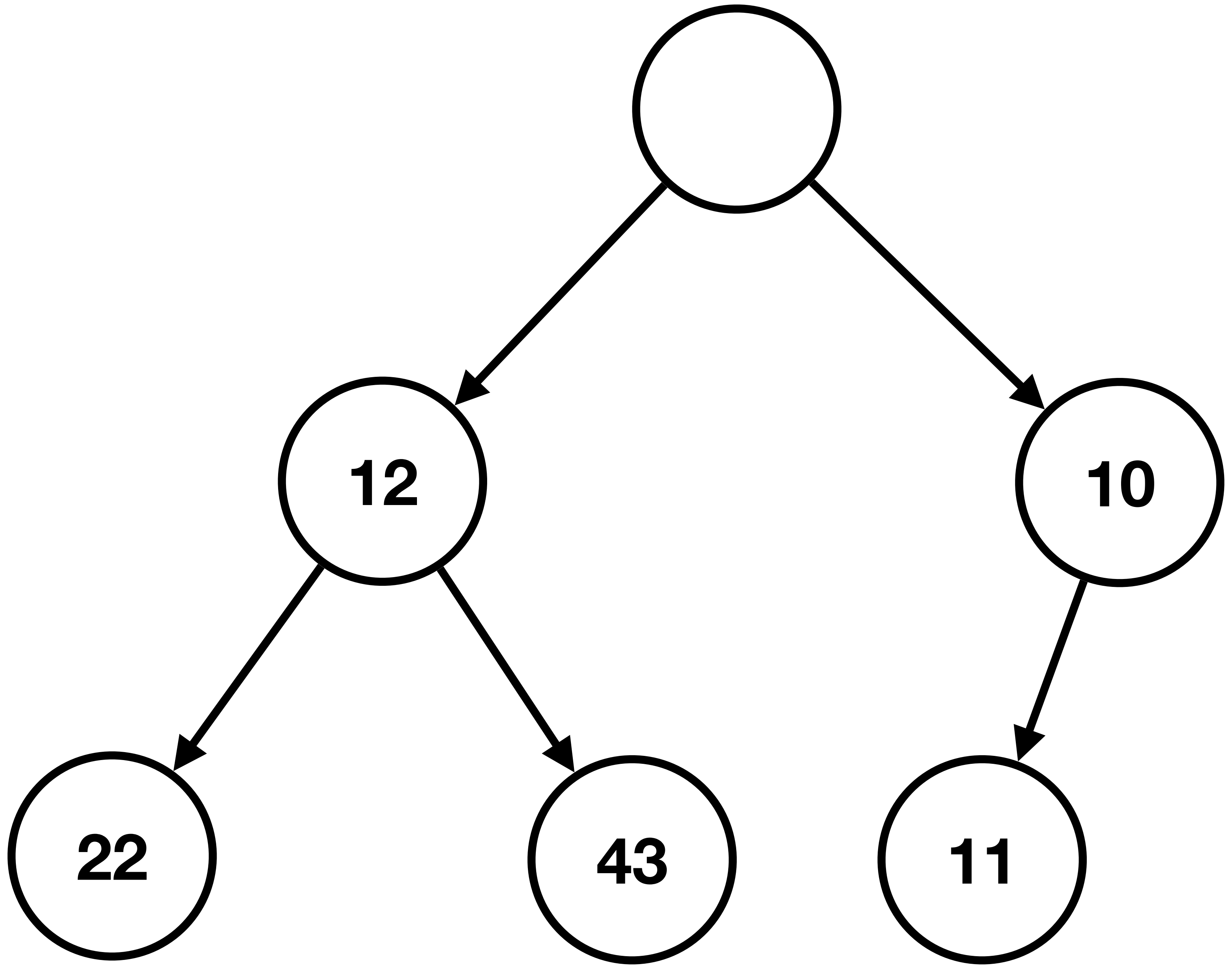
- How about removal?



Sorting

`remove_min()`

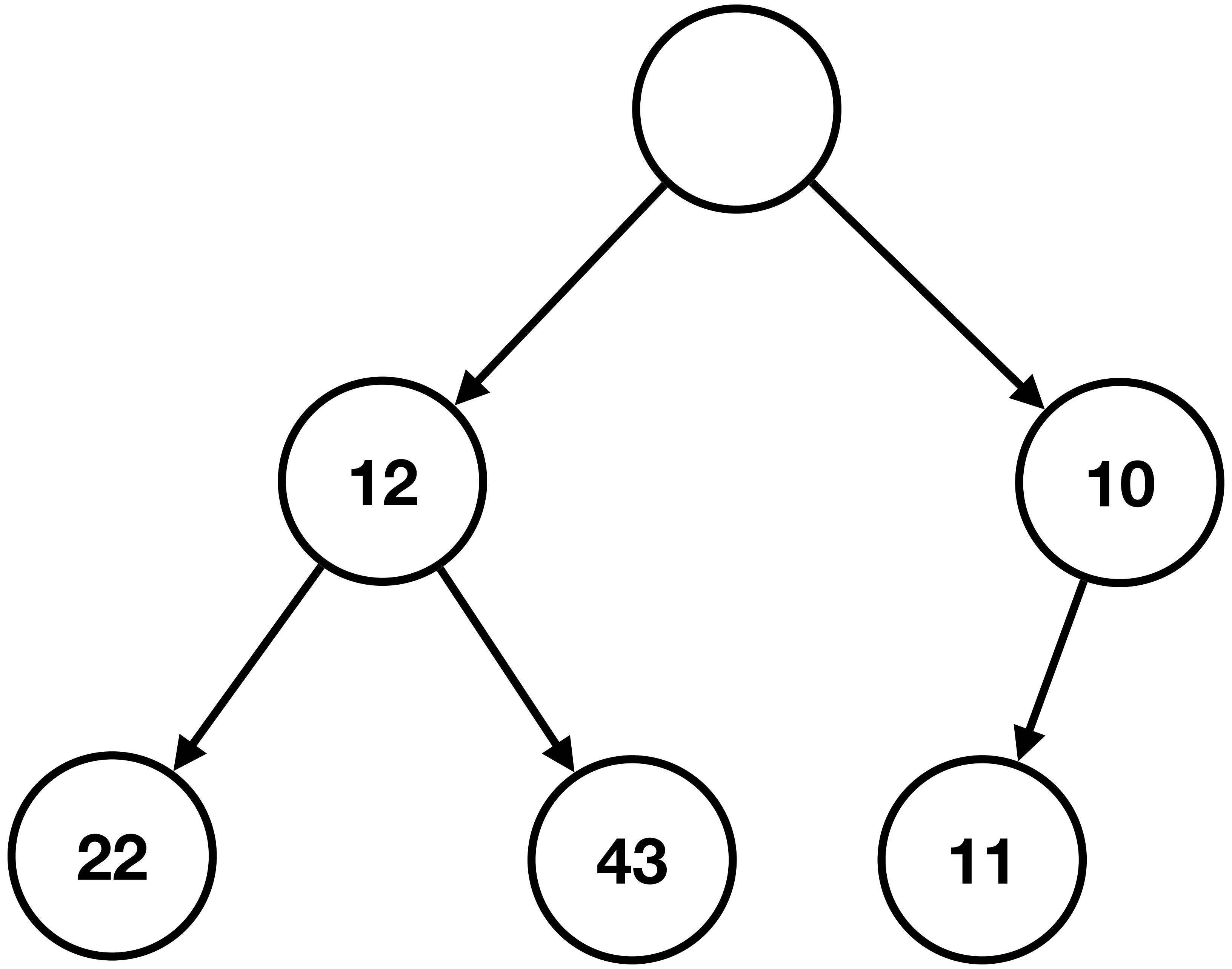
- How about removal?
  - Remove root



Sorting

`remove_min()`

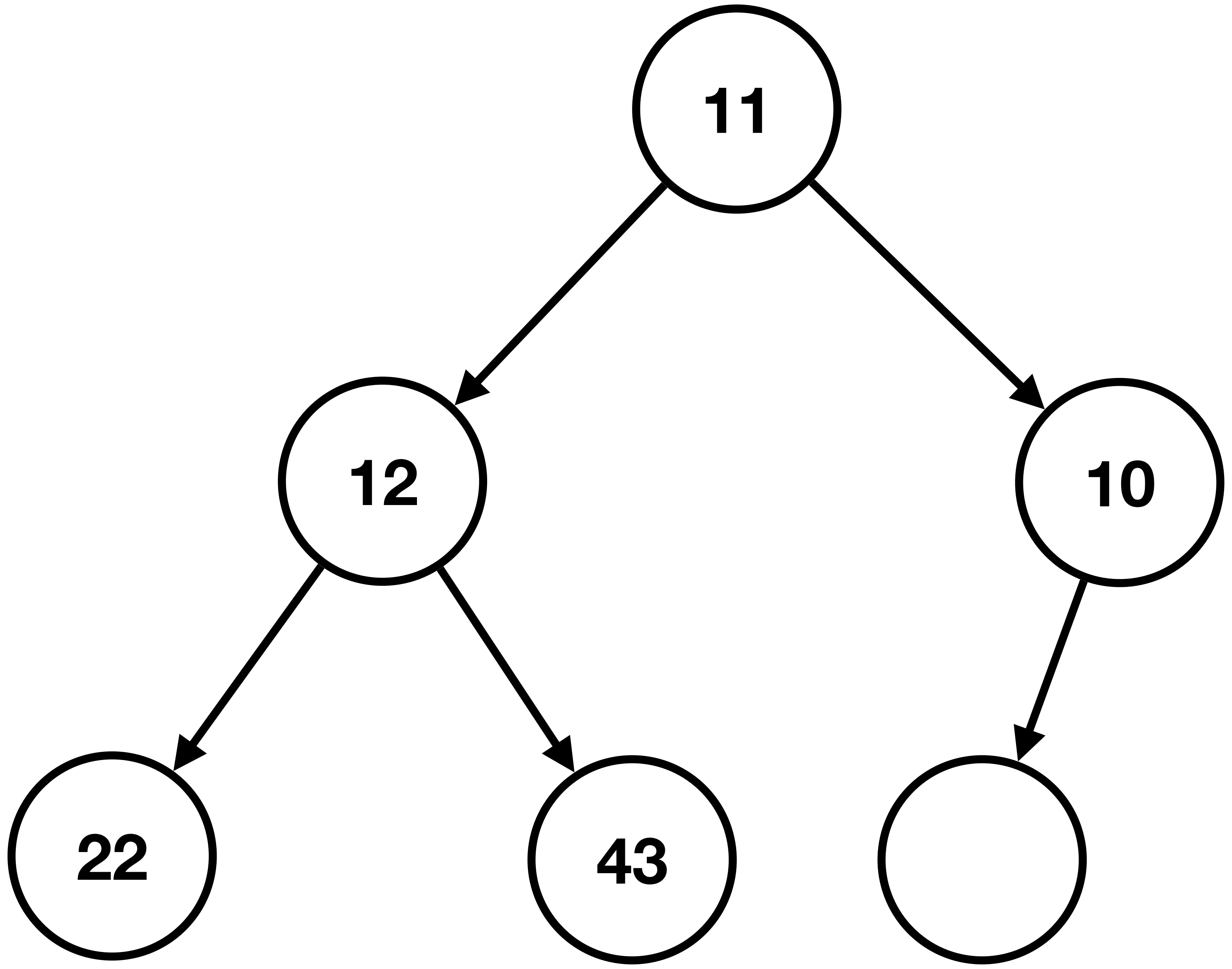
- How about removal?
  - Remove root
  - Maintain shape: replace the root with the last element



Sorting

`remove_min()`

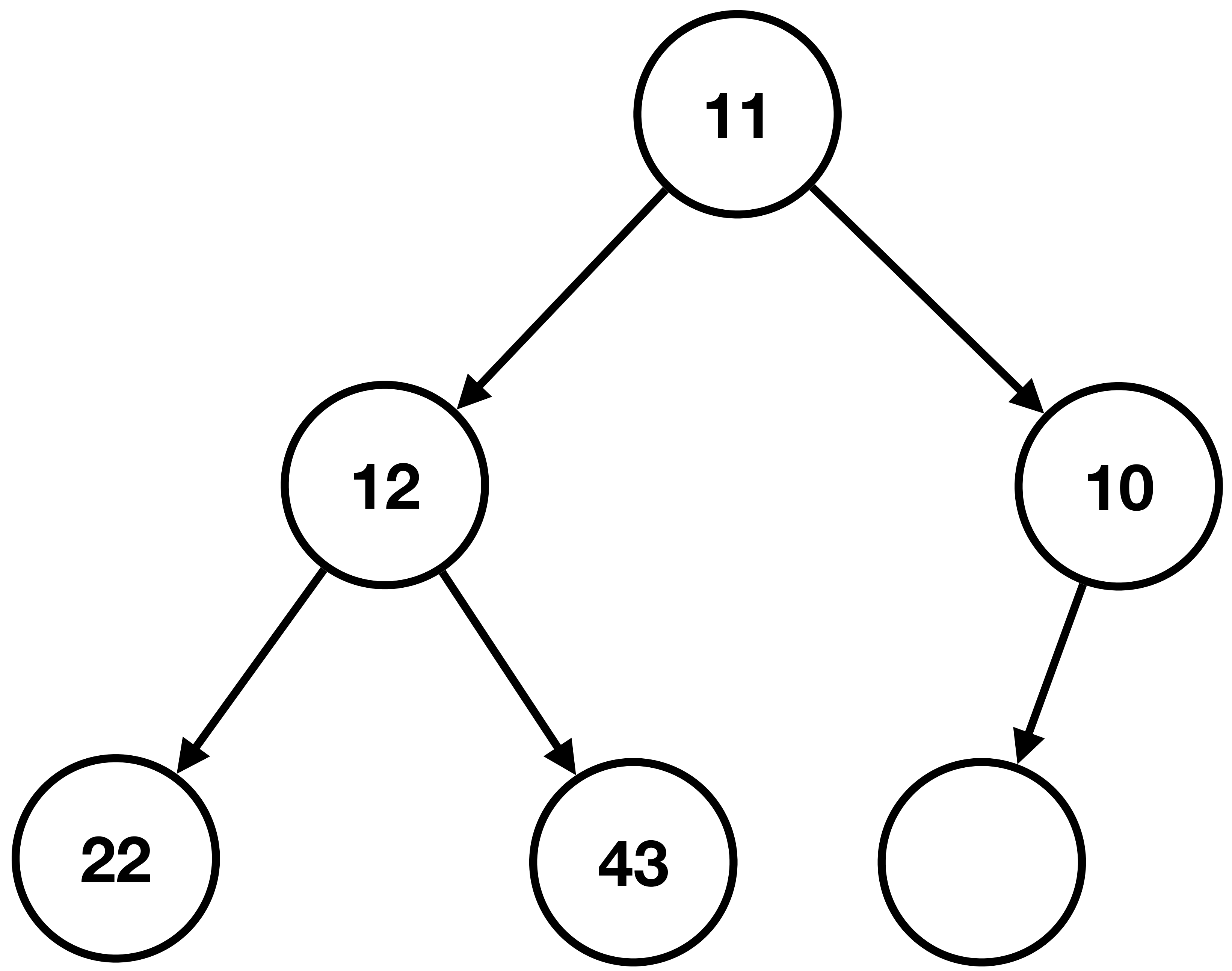
- How about removal?
  - Remove root
  - Maintain shape: replace the root with the last element



Sorting

`remove_min()`

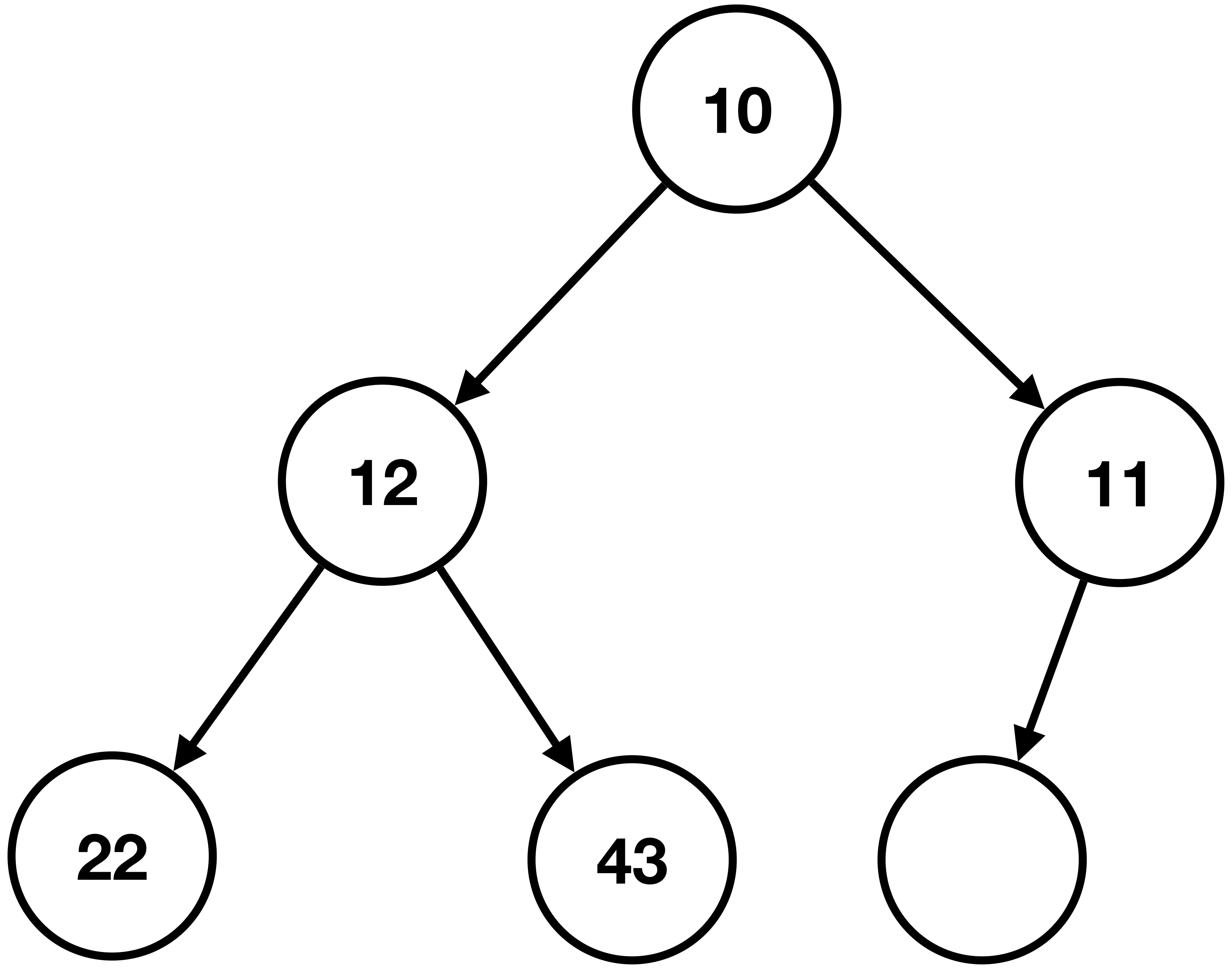
- How about removal?
  - Remove root
  - Maintain shape: replace the root with the last element
  - Bubble-down:
    - Compare with its children
    - Swap with the smallest child, and continue bubbling down



Sorting

`remove_min()`

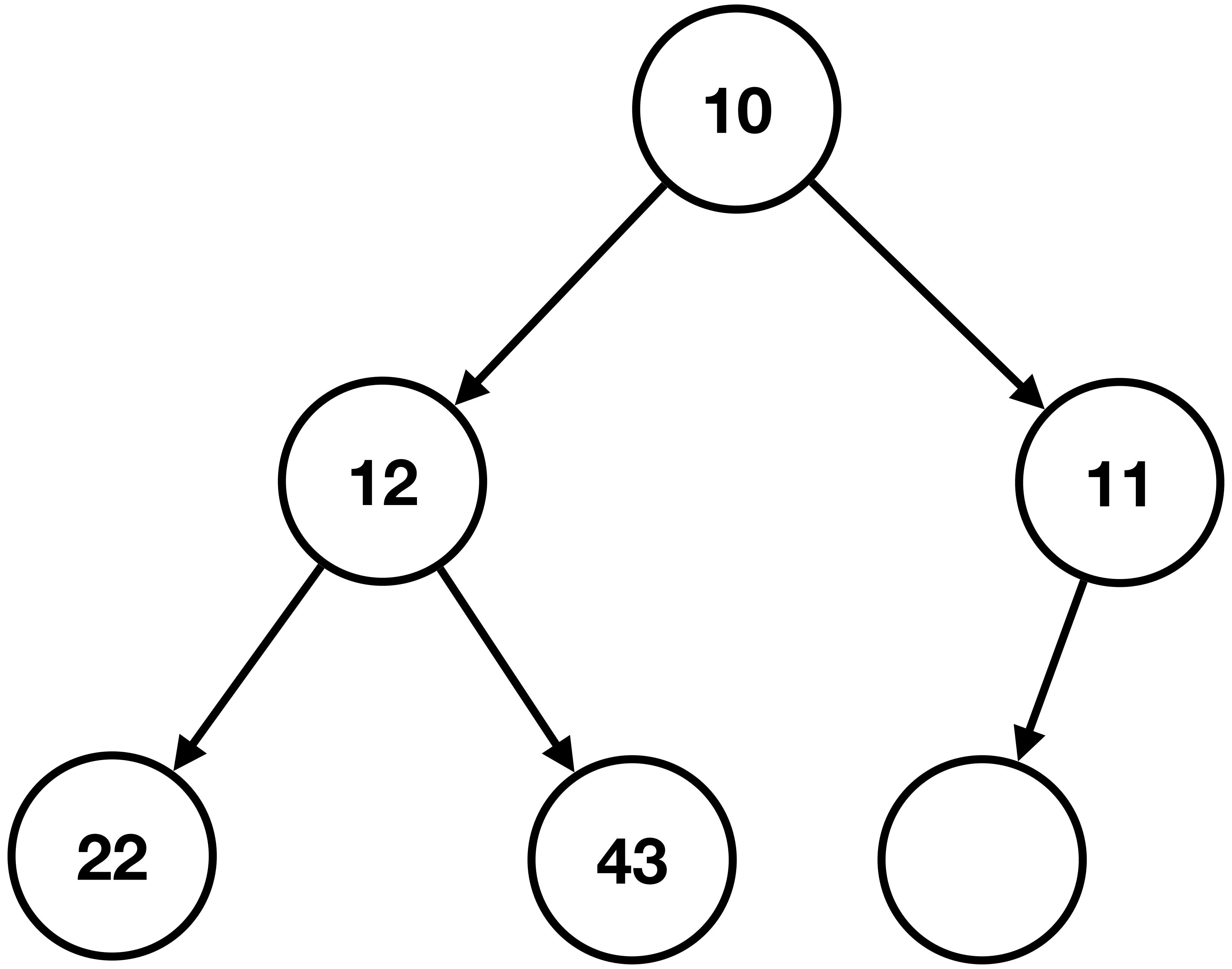
- How about removal?
  - Remove root
  - Maintain shape: replace the root with the last element
  - Bubble-down:
    - Compare with its children
    - Swap with the smallest child, and continue bubbling down



Sorting

`remove_min()`

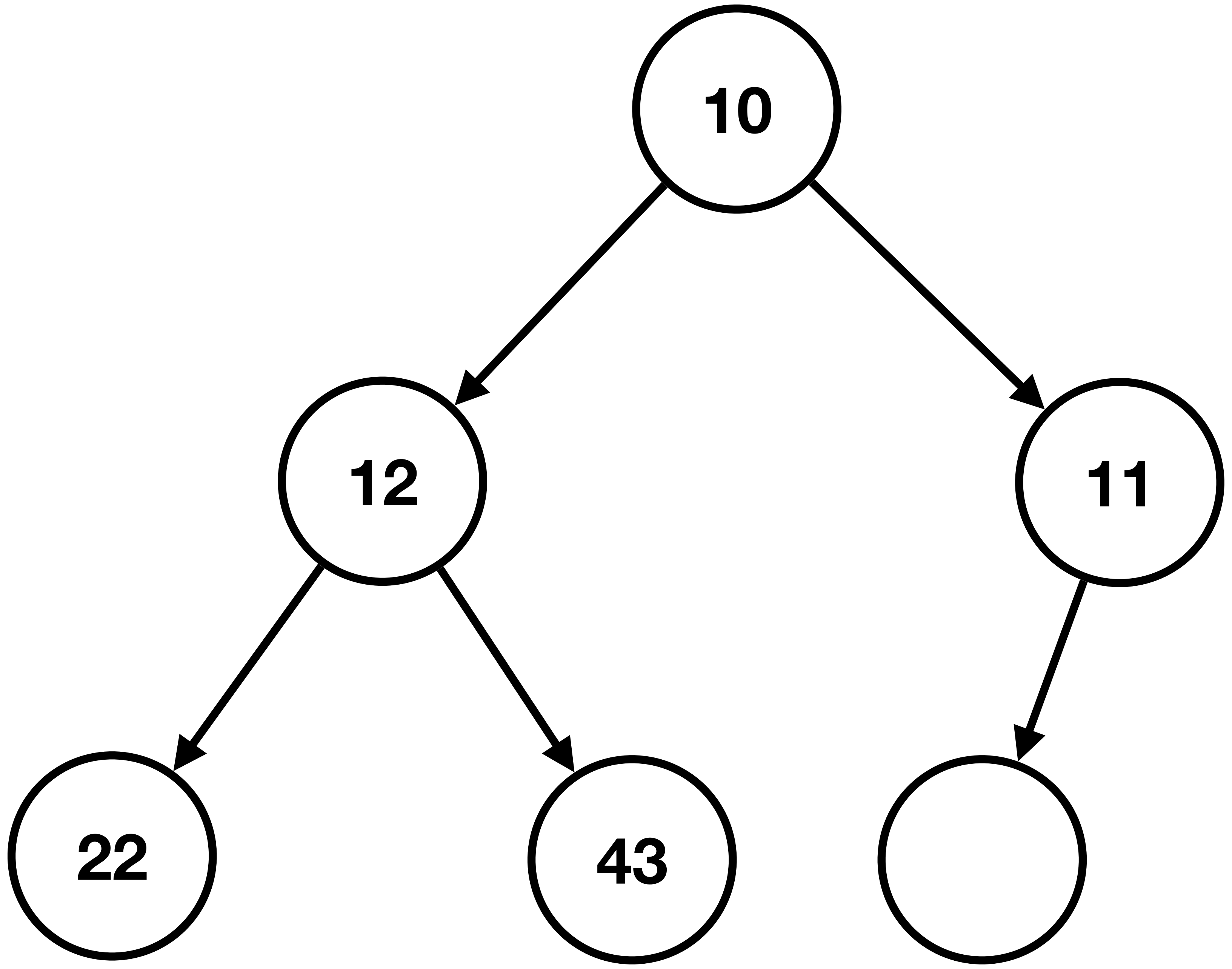
- How about removal?
  - Remove root
  - Maintain shape: replace the root with the last element
  - Bubble-down:
    - Compare with its children
    - Swap with the smallest child, and continue bubbling down
- Complexity: O(height) = O(log n)



Sorting

Heap Sort

- Heap Operations:
  - Insert: O(log n)
  - get_min: O(1)
  - remove: O(log n)
- Heap Sort:
  - Construct the heap: n * O(log n)
  - Remove all elements: n * O(log n)



Quiz Review

Variables

In one slide

- A variable is a named location in memory.
  - A variable has a type (thereby size) and a location in memory.
- A variable needs to be *declared* before use. Syntax: `type name;`
  - The first assignment to a variable is called *initialization*.
  - A variable contains junk between declaration and initialization.
- An *array* is a *contiguous* block of elements of the *same type*.
  Syntax: `type name[number];`
  - Fixed size
  - Access/modify by index, syntax: `name[index]`. This index is not checked.
- A *string* is a NUL-terminated array of characters.

Pointers

Checkpoint I

- `type *name;` declares a variable of type "pointer to `type`"
- `*name` "dereferences" name -- following the address contained in `name` for reading or writing
- `&name` gets the address of name -- if name has type "`type`", `&name` has type "`type *`"
- Pointers can be used for passing arguments by references.
- Pointers enable sharing the same piece of data between functions.

Pointers

Review

type : int
value: 25

&x

type : int *
value: 100

type : int *
value: 100

&x_p

type : int **
value: 108

type : int
value: 25

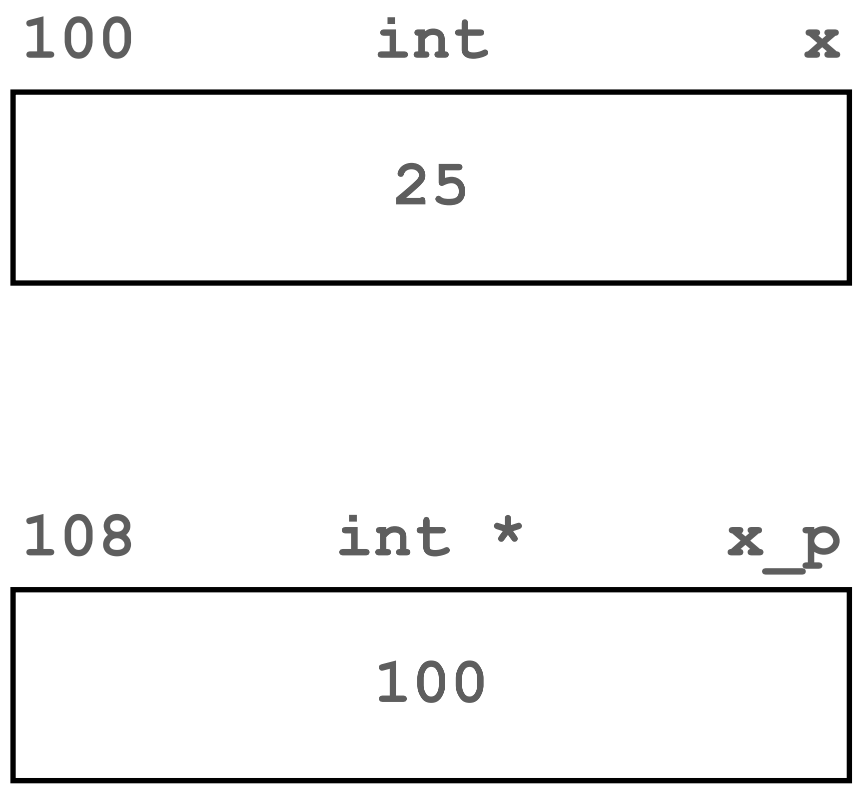
***x**

error

type : int *
value: 100

***x_p**

type : int
value: 25



Pointers

Example: Multiple return values

```python
def divide(x, y):
    q = 0
    while y <= x:
        x -= y
        q += 1
    return q, x

q, r = divide(7, 3)
print(q, r) # 2, 1
```

```c
void divide(int x, int y, int *q_p, int *r_p)
{
        int q = 0;
        while (y <= x) {
            x -= y;
            q += 1;
        }
        *q_p = q;
        *r_p = x;
}

int main(void)
{
        int q, r;
        divide(7, 3, &q, &r);

        printf("%d %d\n", q, r); // 2, 1

        return 0;
}
```

The Heap

Stack vs Heap

Stack

- Acquire memory:
  - declare variables
  - size: compiler calculates *before* running (static)
- Release memory:
  - do nothing
  - You can't forget to release

Heap

- Acquire memory:
  - `ptr = malloc(n)`
  - size: you provide *during* running (dynamic)
- Release memory:
  - `free(ptr)`
  - You can forget to release; *memory leak*

- Accessing released memory is bad; *memory error*

Pointers

Example: Array

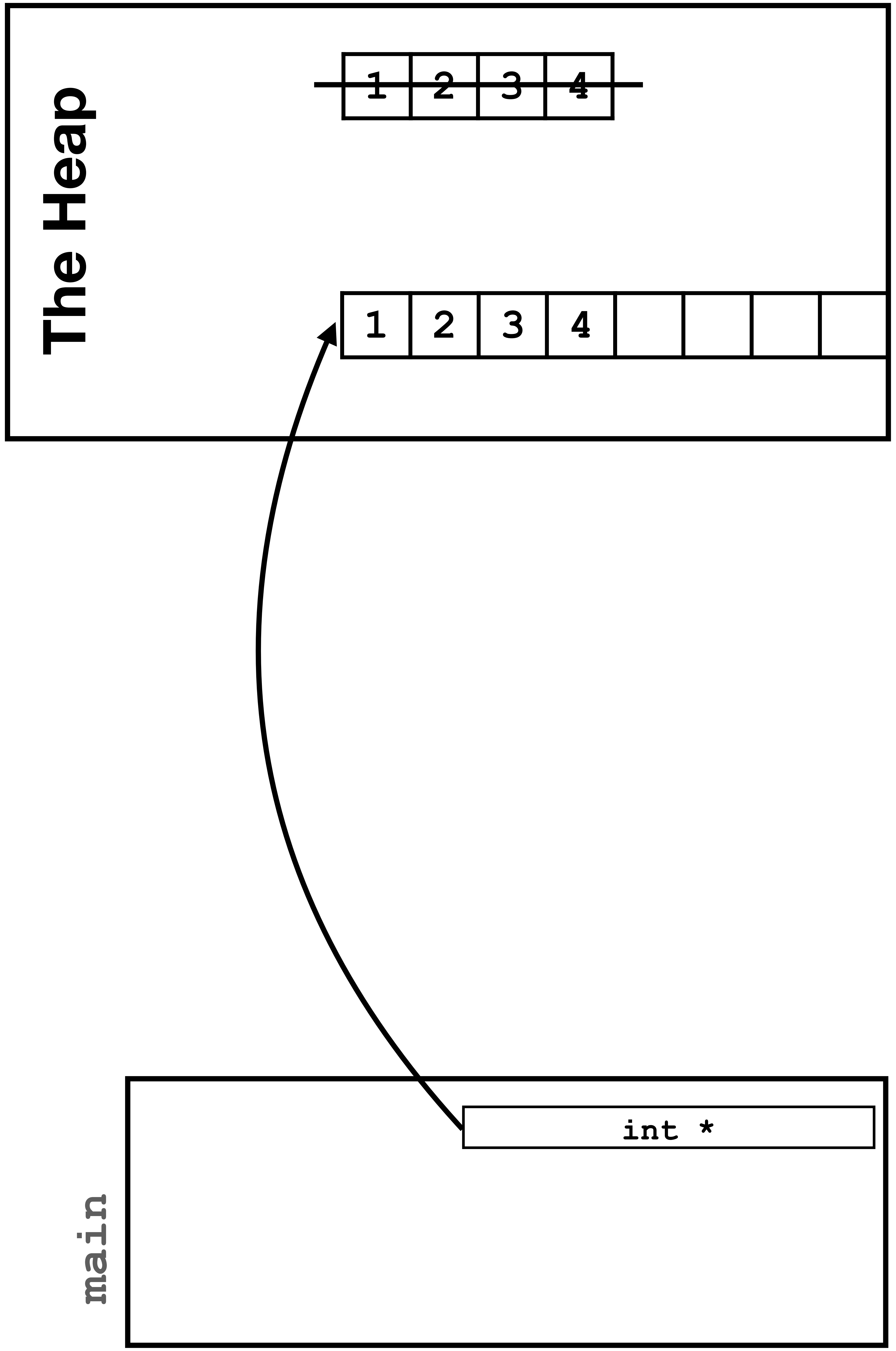
```c
int sum(int *arr, int n)
{
        int sum = 0;
        for (int i = 0; i < n; i++) {
                sum += arr[i];
        }
        return sum;
}

int main(void)
{
        int numbers[7] = { 0, 1, 2, 3, 4, 5, 6 };

        printf("%d\n", sum(numbers, 7));

        return 0;
}
```

- Even when `number` is a massive array, no copying is needed
- `&numbers[0] == numbers`
- Pitfall: `==` does pointer comparison between arrays, does not compare elements
    - use `for` loop

Array

Growing an array

- Pointers serve as an indirection.
  - We aren't changing the size of the array; we are changing which array the pointers point to.
  - By changing the address of the pointer, it seems to the user that we have changed the size of the array.
- We create and delete memory however we want thanks to the heap.



Linked Lists

Sorting

Selection, Insertion, Bubble

```
For i = 1 to n - 1:
    min = index of smallest in A[i + 1 : n]
    swap A[min] and A[i]

For i = 2 to n:
    j = i - 1
    while j > 0 and A[j] > A[i]:
        A[j + 1] = A[j]
        j = j - 1
    A[j] = A[i]

while True:
    swapped = False
    for i = 0 to n - 1:
        if A[i] > A[i + 1]:
            swap A[i], A[i + 1]
            swapped = True
    if not swapped:
        break
```