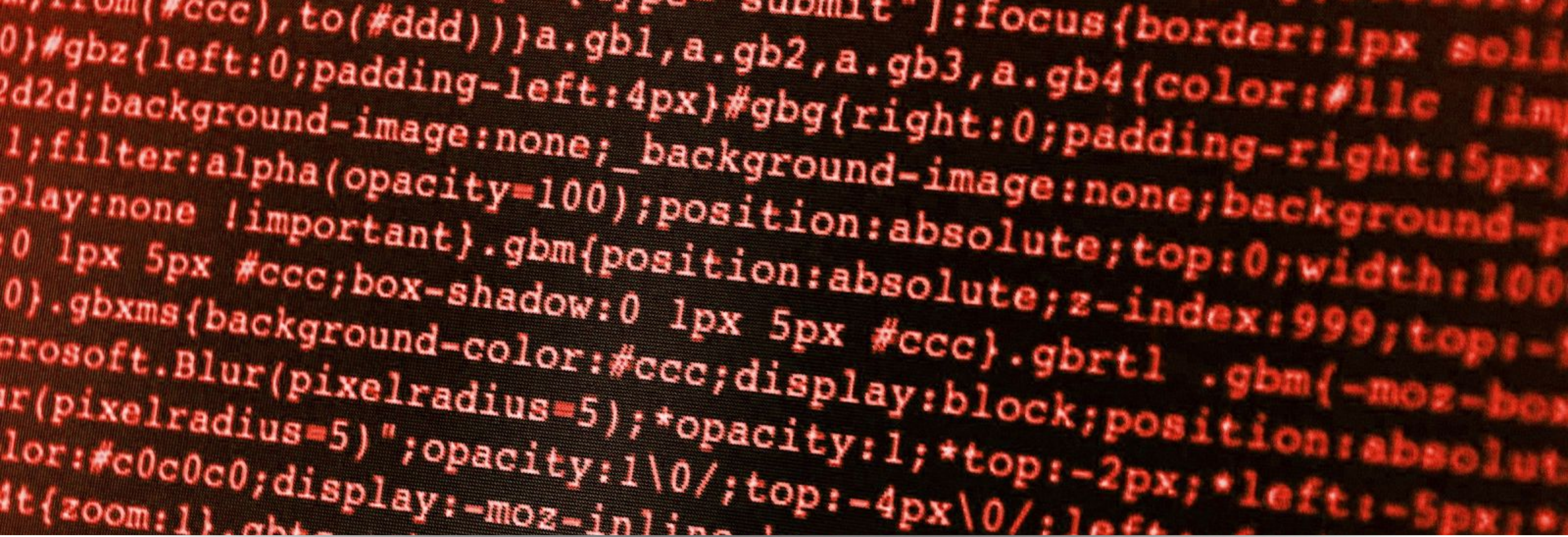


04

Week of February 10, 2020

Priority Queues, Heaps, Union-Find

Announcements

- Lab 3 due 2/14 and Lab 4 due 2/21
- Project 1 final grading is in process
  - Please review and understand the project grading policy - the score from final grading is what goes in the gradebook, and that is not necessarily the highest score you see in the autograder
- Project 2 is **due 2/20**
- Midterm 2/26, 6:30pm to 8:30pm
  - If you need an alternate time, or SSD accommodations, instructions are in a pinned Piazza announcement - you must complete the alternate exam request form as soon as possible! (**Deadline: Wednesday, February 12, 2020**)

  - If you wish to receive the SSD accommodations you are entitled to, you must do BOTH of 1) filling out the alternate exam request form, AND 2) providing your SSD form to the staff - SSD forms can go to eecs281admin@umich.edu electronically (if you don't receive a reply within a day, post on Piazza), or to any staff member on paper.

Announcements

- My lab slides available on https://preetiramaraj.github.io/teaching.html

Agenda

- Priority Queue ADT
- Heaps and Heapsort
- Sets and Union Find
- Handwritten Problem

Priority Queues

Priority Queues

- Priority queues are abstract container types that support two operations:
    1. insert a new item
    2. remove an item with the **largest key**
- There are a number of different possible implementations: binary heap, pairing heap, etc.
- You will work with several of these in Project 2!

STL Priority Queues

- Implemented with a binary heap
- Operations:

These complexities are specific to STL's `std::priority_queue`. They are not inherent to the priority queue itself, which is abstract

Name	Function	Complexity
`push`	Inserts an item into the priority queue	O(log n)
`pop`	Removes (without returning) the highest priority item	O(log n)
`top`	Returns (without removing) the highest priority item	O(1)
`empty`	Returns true if the priority queue is empty	O(1)
`size`	Returns the number of items in the priority queue	O(1)

Priority Queues

```
template <
     class T,
     class Container = vector<T>,
     class Compare = less<typename Container::value_type>
   > class priority_queue
```

- If you want to override the comparison functor, you must also set the container type used (you can still use default vector<TYPE>):

```
std::priority_queue<int, std::vector<int>, std::greater<int>> min;
```

Priority Queues

- Practice question: find the k^{th} largest element in an unsorted vector.

Priority Queues

- Practice question: find the k^{th} largest element in an unsorted vector.
- Approach 1:
  - add all elements to a max heap
  - pop k times
  - O(n + k log n) time; O(n) space
  - pitfall: range based constructor should be used while creating the heap, else it is O(n log n) time!

Priority Queues

- Practice question: find the k^{th} largest element in an unsorted vector.
- Approach 1:
  - add all elements to a max heap
  - pop k times
  - O(n + k log n) time; O(n) space
  - pitfall: range based constructor should be used while creating the heap, else it is O(n log n) time!
- Approach 2:
  - add first k elements to a min heap
  - push and pop each of the remaining n - k elements. This way, PQ always of size k!
  - residue at the end is the 'k' largest elements of the vector
  - top of the PQ is the k'th largest element
  - O(n log k) time, O(k) space

Heaps and Heapsort

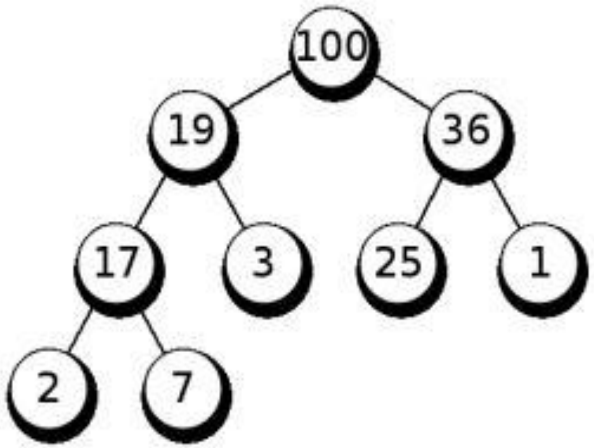
Binary Heap

- A max binary heap is a binary tree with the following properties:
  - Each node has an equal or higher priority to the priority of both of its children (based on the comparator)
  - It is complete: all levels of the heap are full, except possibly the last
    - the last level is filled from left to right

- Binary heaps can be used to create priority queues!
  - You will do this in project 2
  - They are used to implement `std::priority_queue`

Binary Heap Implementation

- Often implemented in code using arrays:



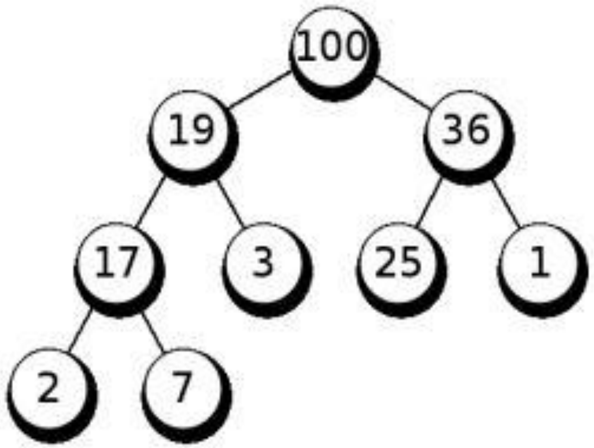
Array-based

[100, 19, 36, 17, 3, 25, 1, 2, 7]

Binary Heap Implementation

- Often implemented in code using arrays:

Tree-based



Array-based

[100, 19, 36, 17, 3, 25, 1, 2, 7]

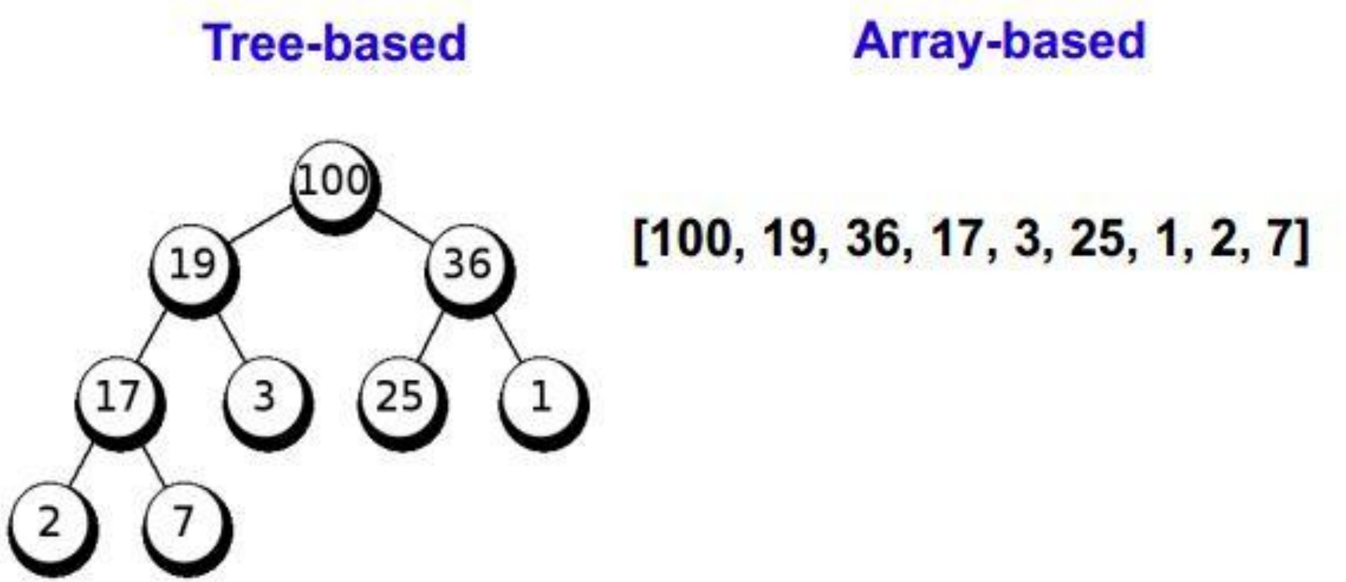
Given a node at position *i* in the array…

What is the index of *i's* parent?

What are the indices of *i*'s two children?

Binary Heap Implementation

- Often implemented in code using arrays:



Given a node at position *i* in the array...

What is the index of *i*'s parent?
(i - 1)/2

What are the indices of *i*'s two children?
left child: 2i + 1
right child: 2i + 2

Binary Heap

- Practice question: which of the following represents a min-heap? A max-heap? Select all that apply.

A. [1, 8, 4, 9, 12, 11, 7]
B. [3, 4, 5, 7, 12, 11, 8, 6, 13]
C. [13, 10, 6, 8, 7, 4, 2, 5, 1, -1, 0]
D. [-1, -9, -3, -10, -11, -5, -7, -12, -13]

Binary Heap

- Practice question: which of the following represents a min-heap? A max-heap? Select all that apply.

A. [1, 8, 4, 9, 12, 11, 7] **MIN-HEAP**
B. [3, 4, 5, 7, 12, 11, 8, 6, 13]
C. [13, 10, 6, 8, 7, 4, 2, 5, 1, -1, 0] **MAX-HEAP**
D. [-1, -9, -3, -10, -11, -5, -7, -12, -13] **MAX-HEAP**

Maintaining the Heap Property

- What if the priority of an item **increases**?
  - We need to fix from the bottom up: **fixUp()**
- How do we fix up?
  - Swap the altered node with its parent, moving up until either
    1. we reach the root
    2. we reach a parent with a larger or equal key

- Question: what is the complexity of fixUp()?

Maintaining the Heap Property

- What if the priority of an item **increases**?
  - We need to fix from the bottom up: **fixUp()**
- How do we fix up?
  - Swap the altered node with its parent, moving up until either
    1. we reach the root
    2. we reach a parent with a larger or equal key

- Question: what is the complexity of fixUp()?

  O(number of levels in the heap) = O(log n)

Maintaining the Heap Property

- What if the priority of an item **decreases**?
  - We need to fix from the bottom up: **fixDown()**
- How do we fix down?
  - Swap the altered node with the greater of its children, moving down until:
    1. we reach the bottom of the heap
    2. both children have a smaller of equal key

- Question: what is the complexity of fixDown()?

Maintaining the Heap Property

- What if the priority of an item **decreases**?
  - We need to fix from the bottom up: **fixDown()**
- How do we fix down?
  - Swap the altered node with the greater of its children, moving down until:
    1. we reach the bottom of the heap
    2. both children have a smaller of equal key

- Question: what is the complexity of fixDown()?

  O(number of levels in the heap) = O(log n)

Heap Insertion

- How do you insert an element into the heap?
  1. insert the new item into the bottom of the heap
  2. call **fixUp()** on the newly inserted item

- Calling **fixUp()** will move the item to its correct position within the heap

Heap Removal

- How do you remove an element from the heap?
    1. remove the root item by replacing it with the last element in heap
    2. delete the last item in the heap
    3. call **fixDown()** on the element that is now in the root position

- Calling **fixDown()** will move the item to its correct position within heap

Priority Queues

- Practice question: consider an empty MAXIMUM priority queue. If we insert the elements 14, 4, 5, 23, 9, 11, 2 into the heap (in this order), and then we remove the most extreme element twice, what are possible array representations of the heap?

  A. [11, 5, 9, 4, 2]
  B. [11, 9, 5, 2, 4]
  C. [11, 5, 9, 2, 4]
  D. [11, 9, 5, 4, 2]

Priority Queues

- Practice question: consider an empty MAXIMUM priority queue. If we insert the elements 14, 4, 5, 23, 9, 11, 2 into the heap (in this order), and then we remove the most extreme element twice, what are possible array representations of the heap?

A. [11, 5, 9, 4, 2]
B. [11, 9, 5, 2, 4]
C. [11, 5, 9, 2, 4]
D. [11, 9, 5, 4, 2]

Make Heap / Heapify: Idea #1

- Build a heap from scratch:
  1. start with an empty heap
  2. insert items one by one into the heap

- What is the complexity of this method?

Make Heap / Heapify: Idea #1

- Build a heap from scratch:
    1. start with an empty heap
    2. insert items one by one into the heap

- What is the complexity of this method?

O(n log n)

Complexity of insert is O(log n) and we do this n times!

And also possible O(n) for memory to create a new vector to hold the elements!

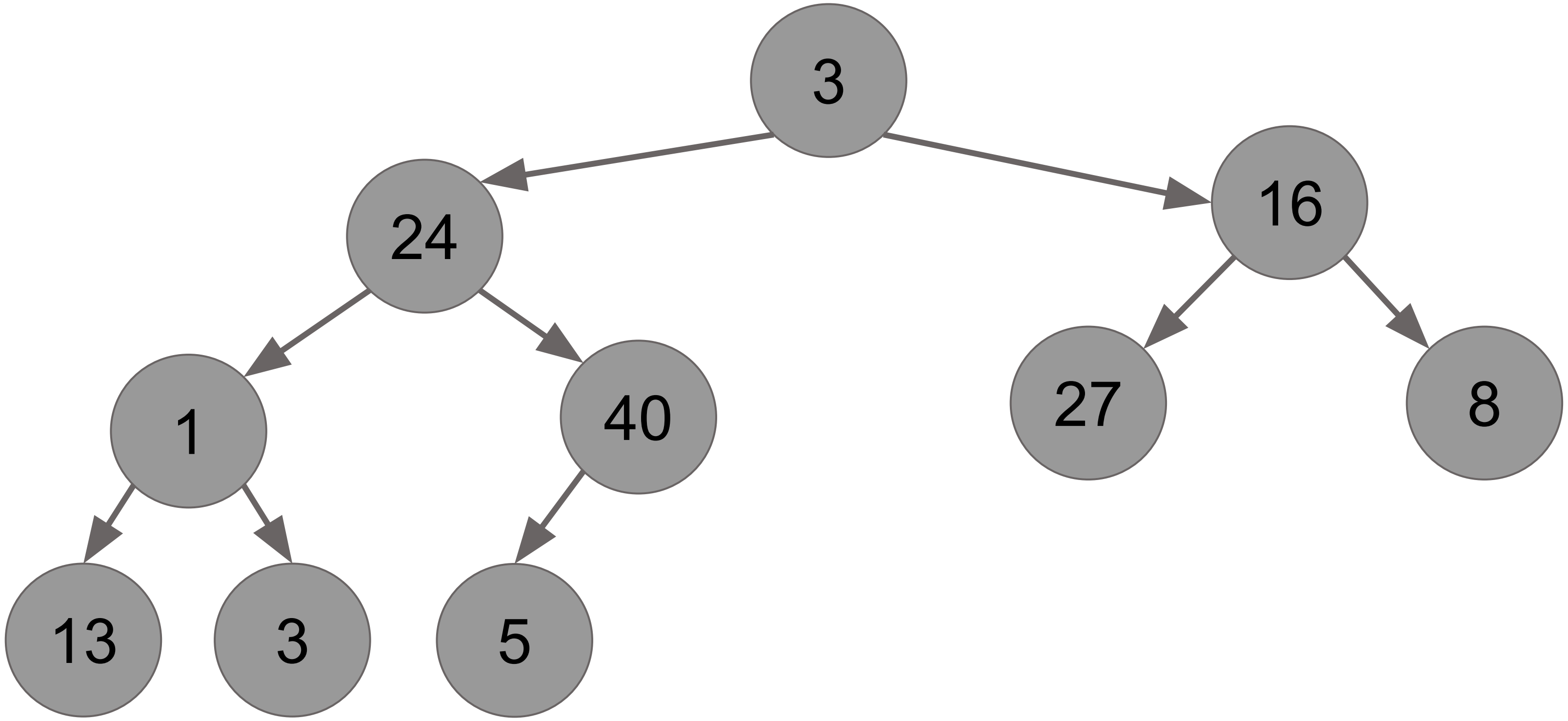
Make Heap / Heapify: Idea #2

- Step 1: Initialize heap's underlying array to the unsorted array.
- Step 2: There are two methods we can choose to enforce heap invariants:
  - Method #1: repeatedly call **fixUp()** starting from the top of the array and moving down. This is equivalent to repeatedly inserting items into a heap.
  - Method #2: repeatedly call **fixDown()** starting from the bottom of the heap and moving up. This is equivalent to making many small heaps and gradually merging them by adding roots and finding the correct positions for them.
- Which approach is better?

Make Heap / Heapify: Idea #2

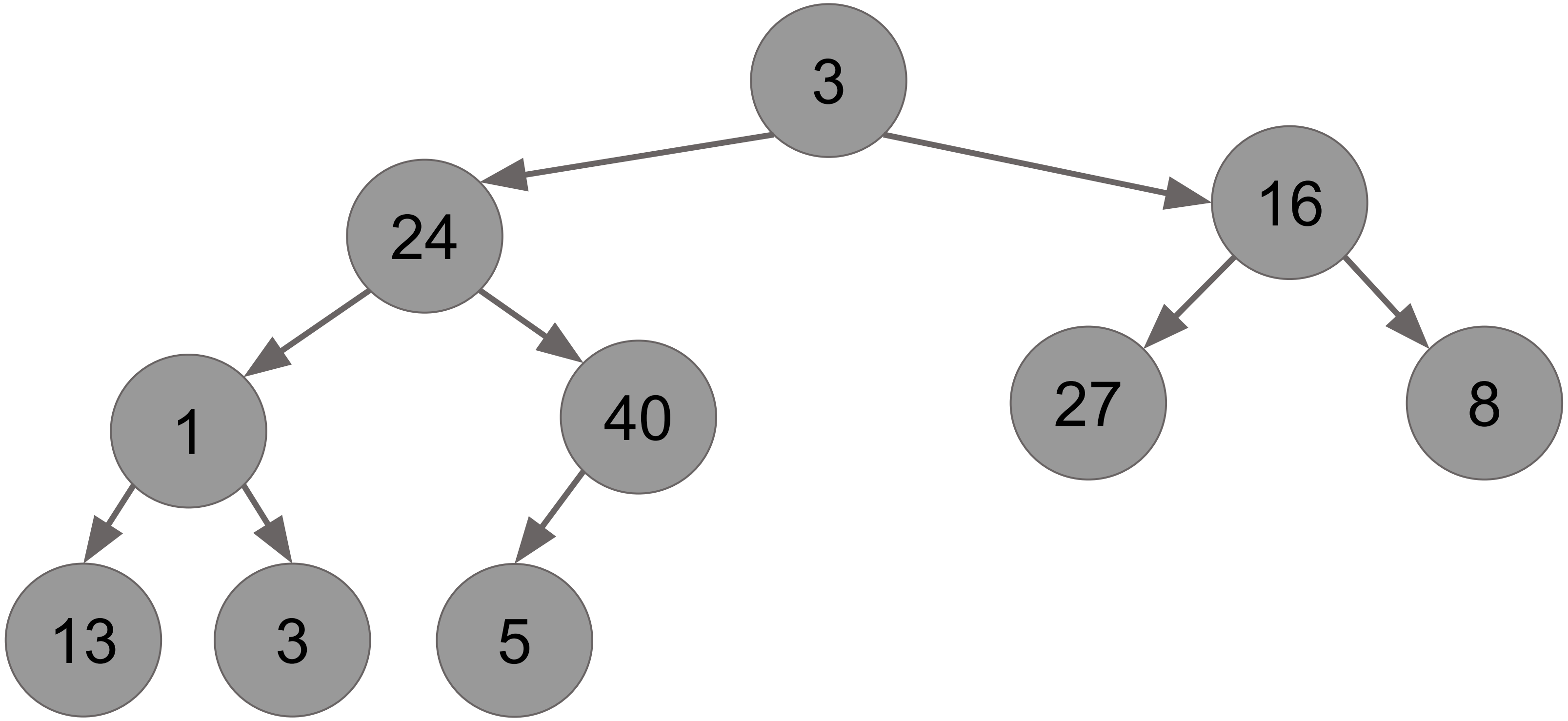
- Step 1: Initialize heap's underlying array to the unsorted array.
- Step 2: There are two methods we can choose to enforce heap invariants:
  - Method #1: repeatedly call **fixUp()** starting from the top of the array and moving down. This is equivalent to repeatedly inserting items into a heap.
  - Method #2: repeatedly call **fixDown()** starting from the bottom of the heap and moving up. This is equivalent to making many small heaps and gradually merging them by adding roots and finding the correct positions for them.
- Which approach is better? **Method #2: calling fixDown() starting from the bottom. Let's look at why...**

Using (Top-down) Fix Up: Less Efficient



Key idea:
Complexity of fixing the position of one item is O(the depth of the item) and we need to fix every item to ensure that it is in the right place.

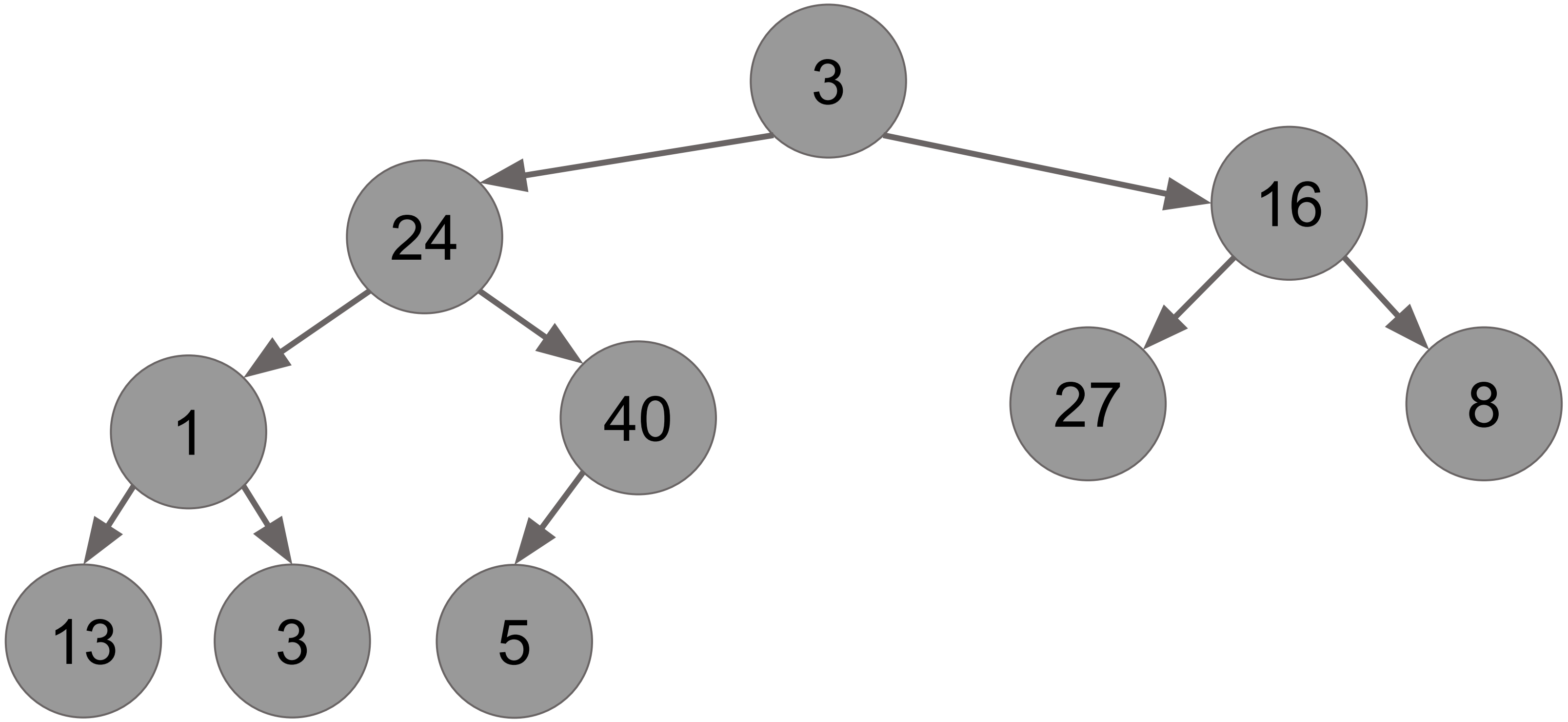
Using (Top-down) Fix Up: Less Efficient



Depth (level from root)	Max #Nodes
0	1
1	2
2	4
...	
Log(n)	n/2

Key idea:
Complexity of fixing the position of one item is O(the depth of the item) and we need to fix every item to ensure that it is in the right place.

Using (Top-down) Fix Up: Less Efficient

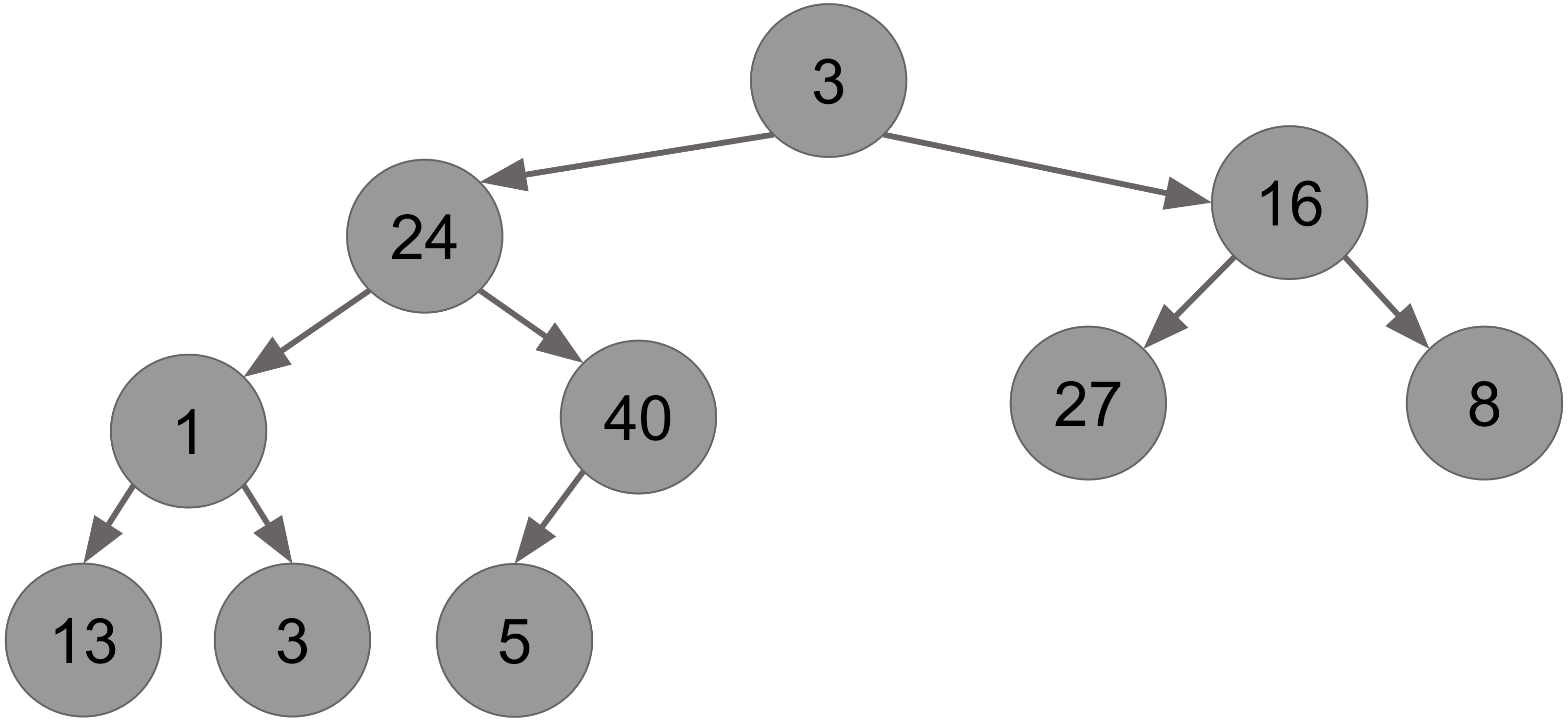


Depth (level from root)	Max #Nodes
0	1
1	2
2	4
...	
Log(n)	n/2

Key idea:
Complexity of fixing the position of one item is O(the depth of the item) and we need to fix every item to ensure that it is in the right place.

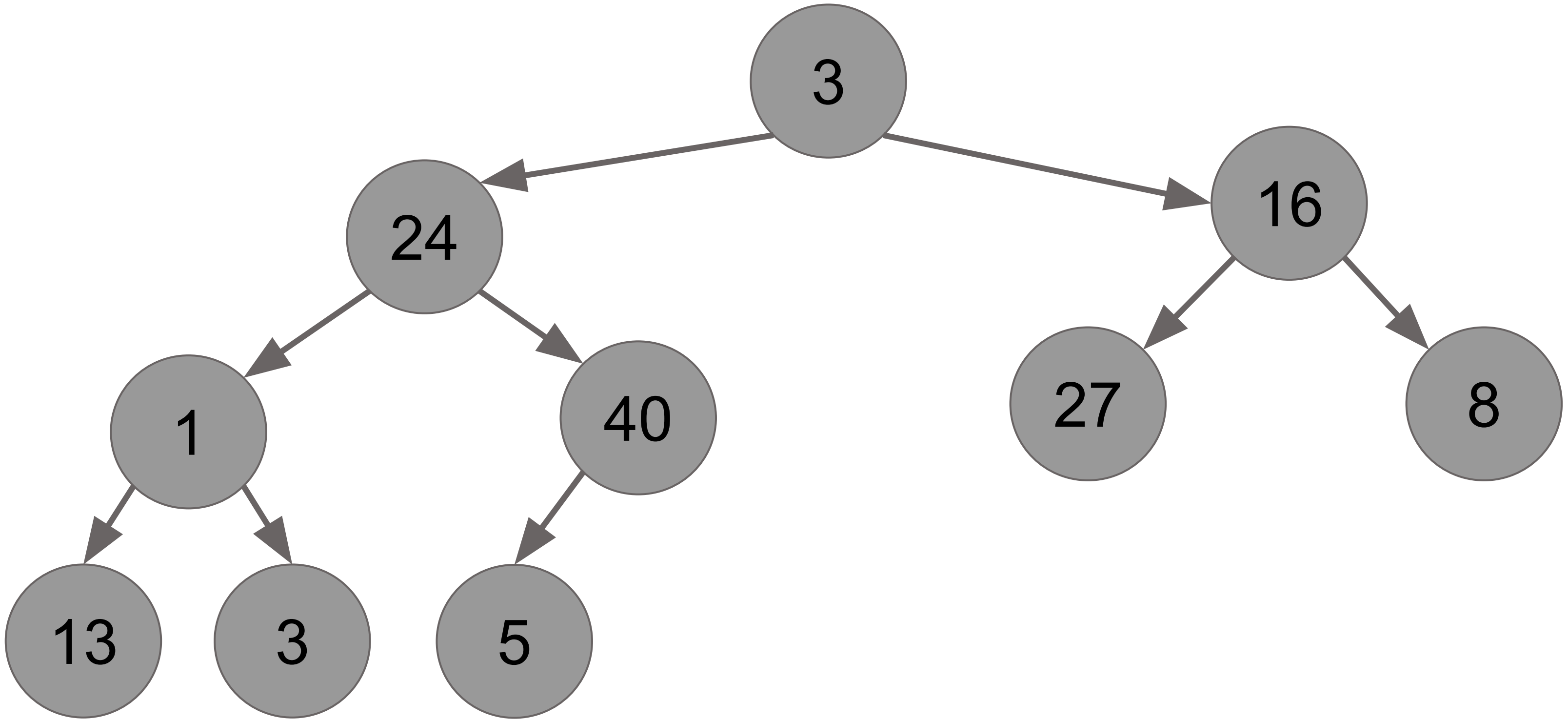
Total complexity =
0 * 1 + 1 * 2 + 2 *4 + ... log(n) * (n /2) =
O(n log n)

Using (Bottom-up) Fix Down: More Efficient



Key idea:
Complexity of fixing the position of one item is O(the depth of the item) and we need to fix every item to ensure that it is in the right place.

Using (Bottom-up) Fix Down: More Efficient

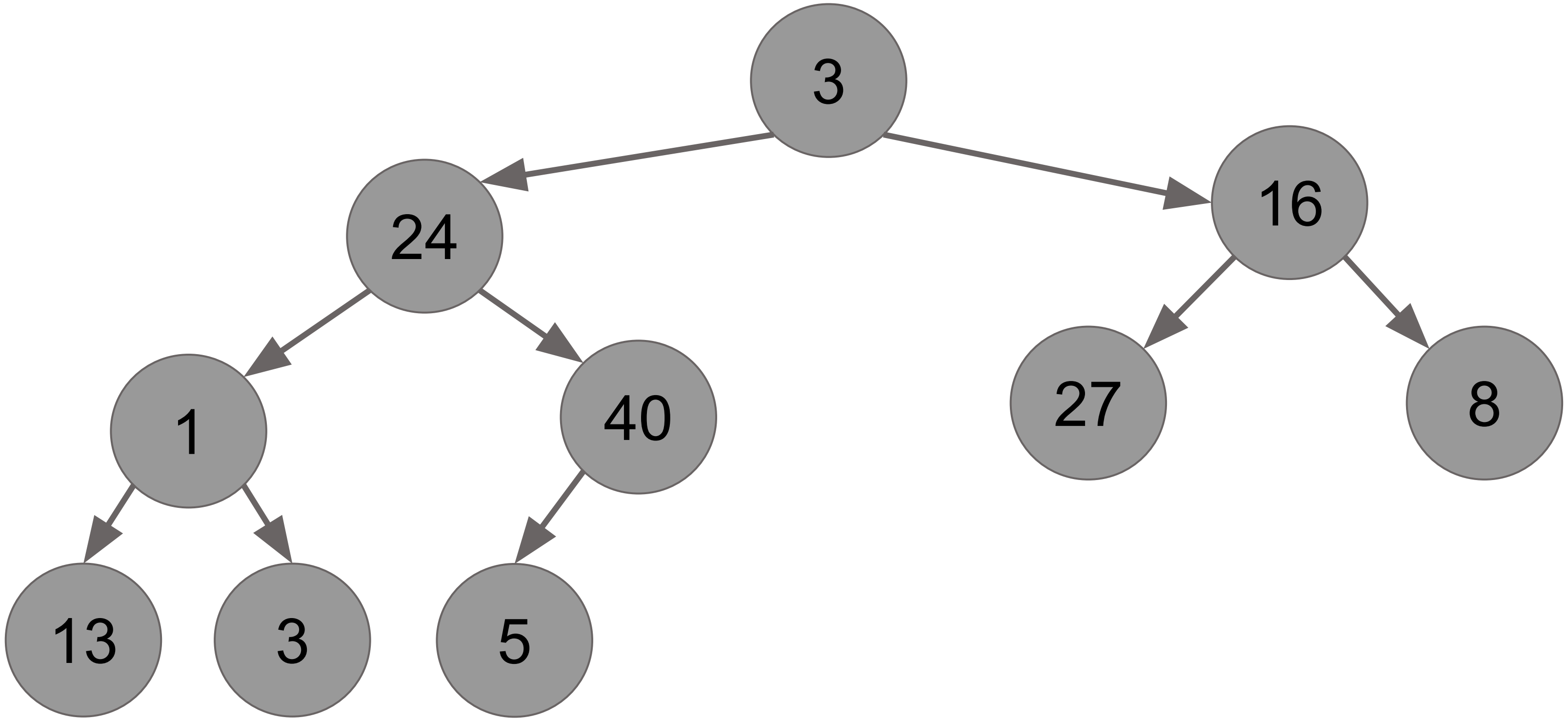


Height (level from leaf)	Max #Nodes
Log(n)	1
...	
2	n/8
1	n/4
0	n/2

Key idea:
Complexity of fixing the position of one item is O(the depth of the item) and we need to fix every item to ensure that it is in the right place.

Using (Bottom-up) Fix Down: More Efficient

Height (level from leaf)	Max #Nodes
Log(n)	1
...	
2	n/8
1	n/4
0	n/2



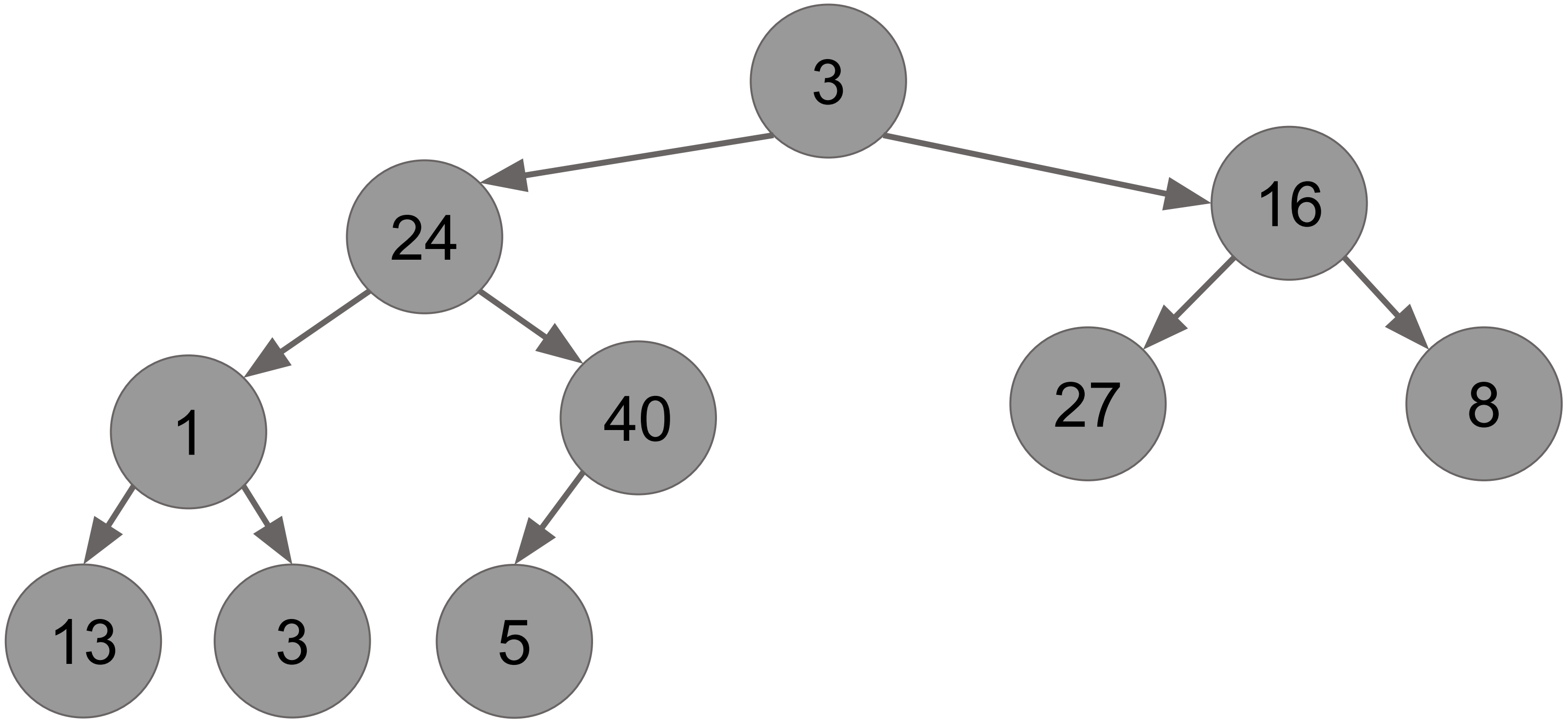
$$1 \cdot \frac{n}{2} + 2 \cdot \frac{n}{4} + 3 \cdot \frac{n}{8} + 4 \cdot \frac{n}{16} + \cdots + \log_2(n) \cdot \frac{n}{2^{\log_2(n)}}$$

$$\leq n\left[\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots\right]$$

$$= n \cdot 2$$

Key idea:
Complexity of fixing the position of one item is O(the depth of the item) and we need to fix every item to ensure that it is in the right place.

Using (Bottom-up) Fix Down: More Efficient



Height (level from leaf)	Max #Nodes
Log(n)	1
...	
2	n/8
1	n/4
0	n/2

$$1 \cdot \frac{n}{2} + 2 \cdot \frac{n}{4} + 3 \cdot \frac{n}{8} + 4 \cdot \frac{n}{16} + \cdots + \log_2(n) \cdot \frac{n}{2^{\log_2(n)}}$$

$$\leq n \left[\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots\right]$$

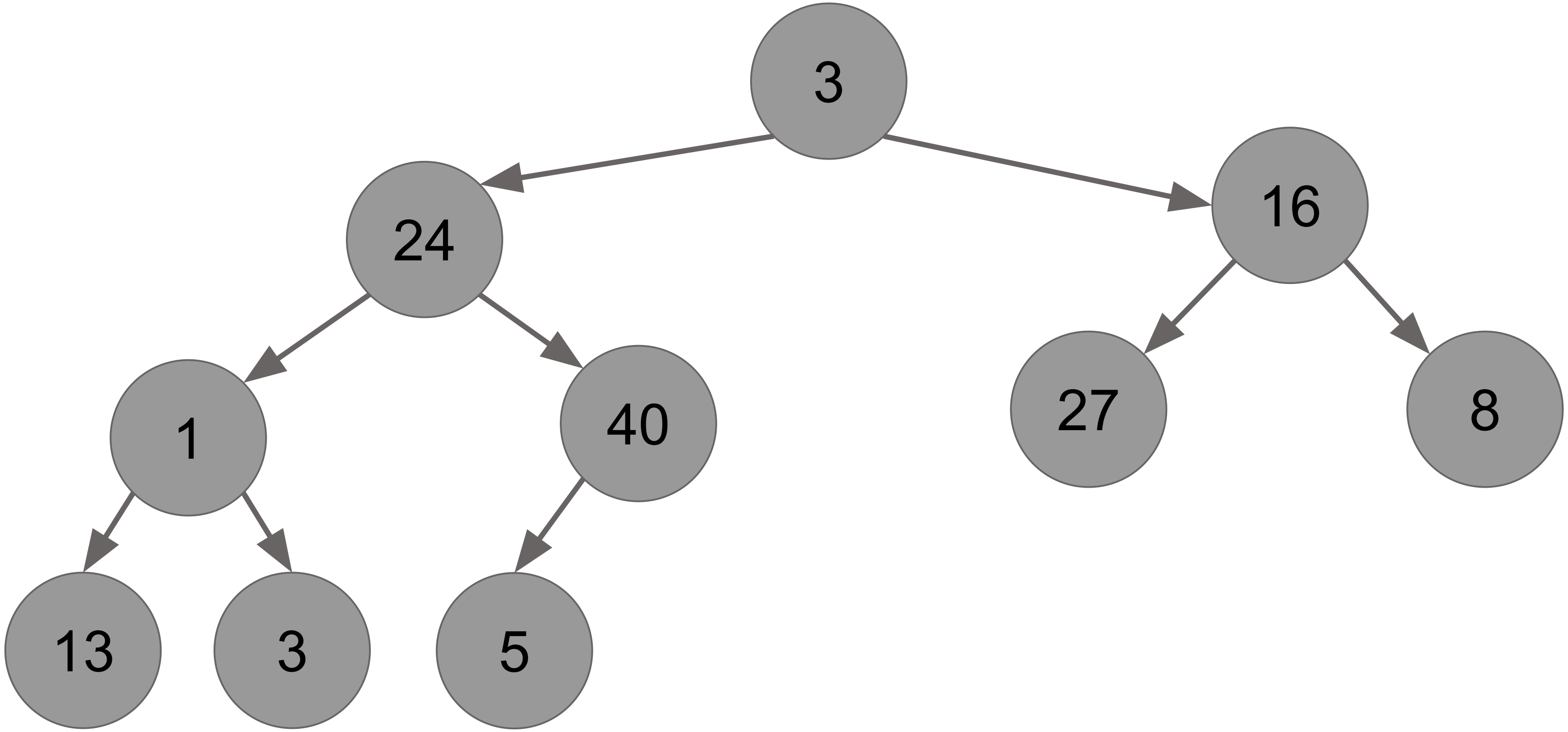
$$= n \cdot 2$$

Key idea:
Complexity of fixing the position of one item is O(the depth of the item) and we need to fix every item to ensure that it is in the right place.

This method is O(n) instead of O(n log n)!

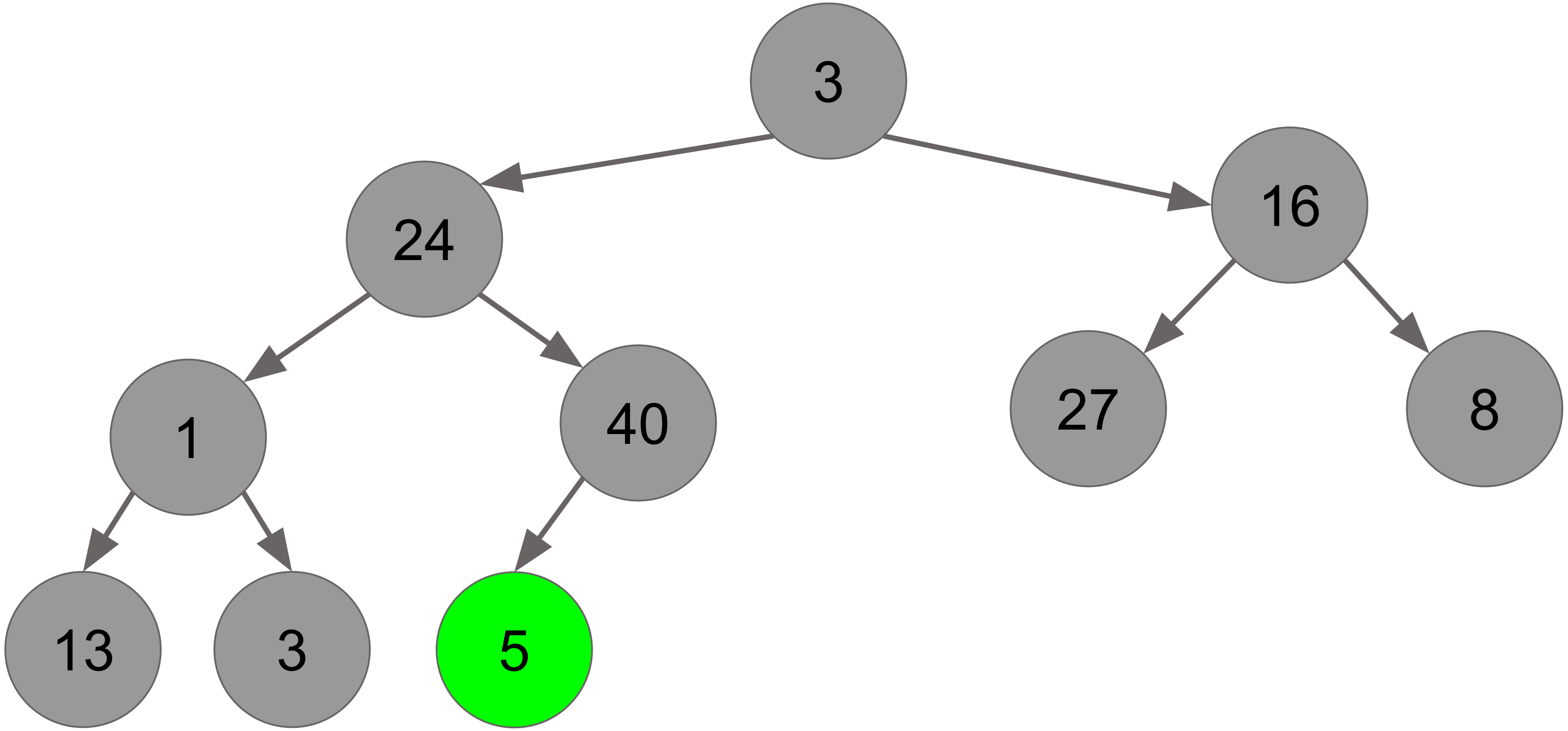
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



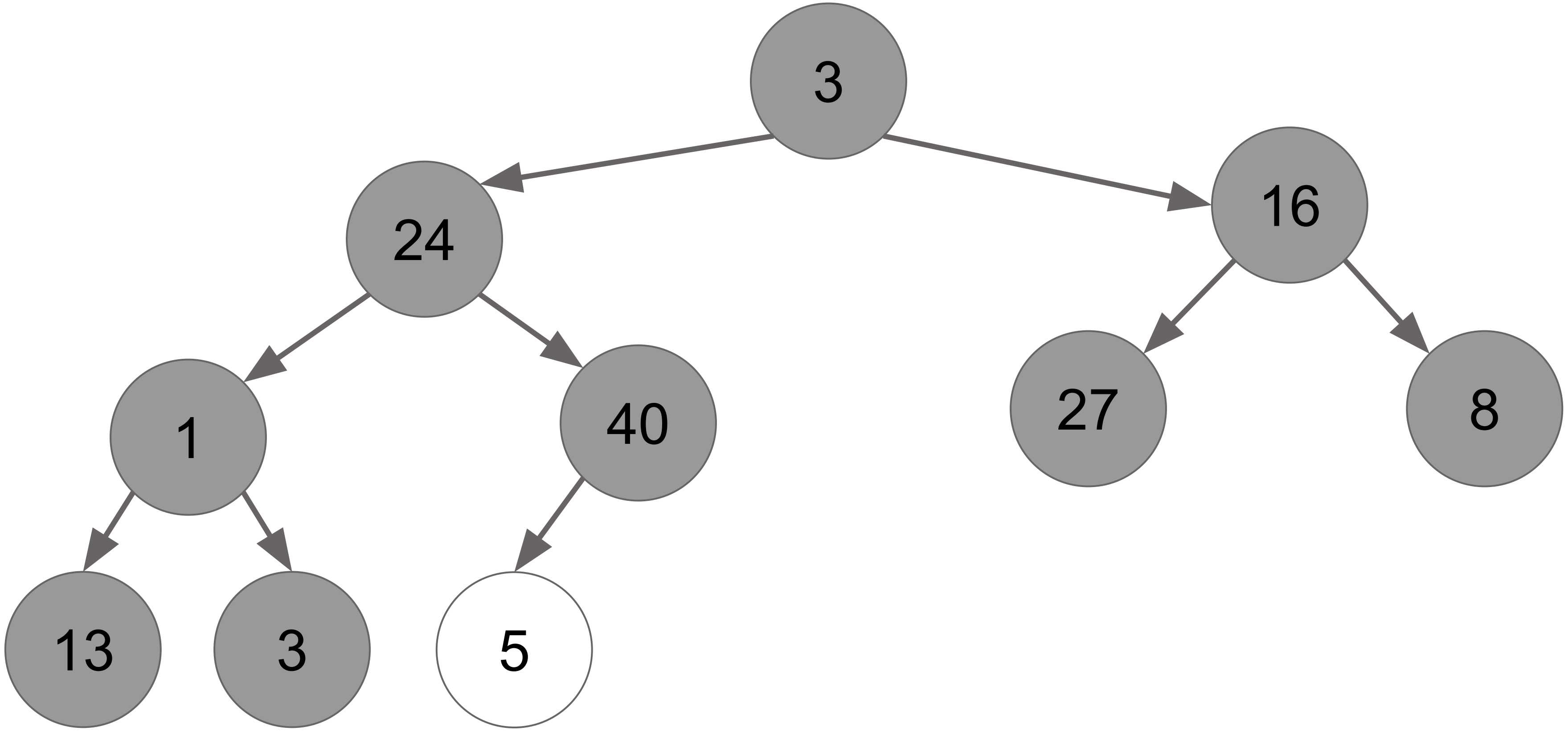
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



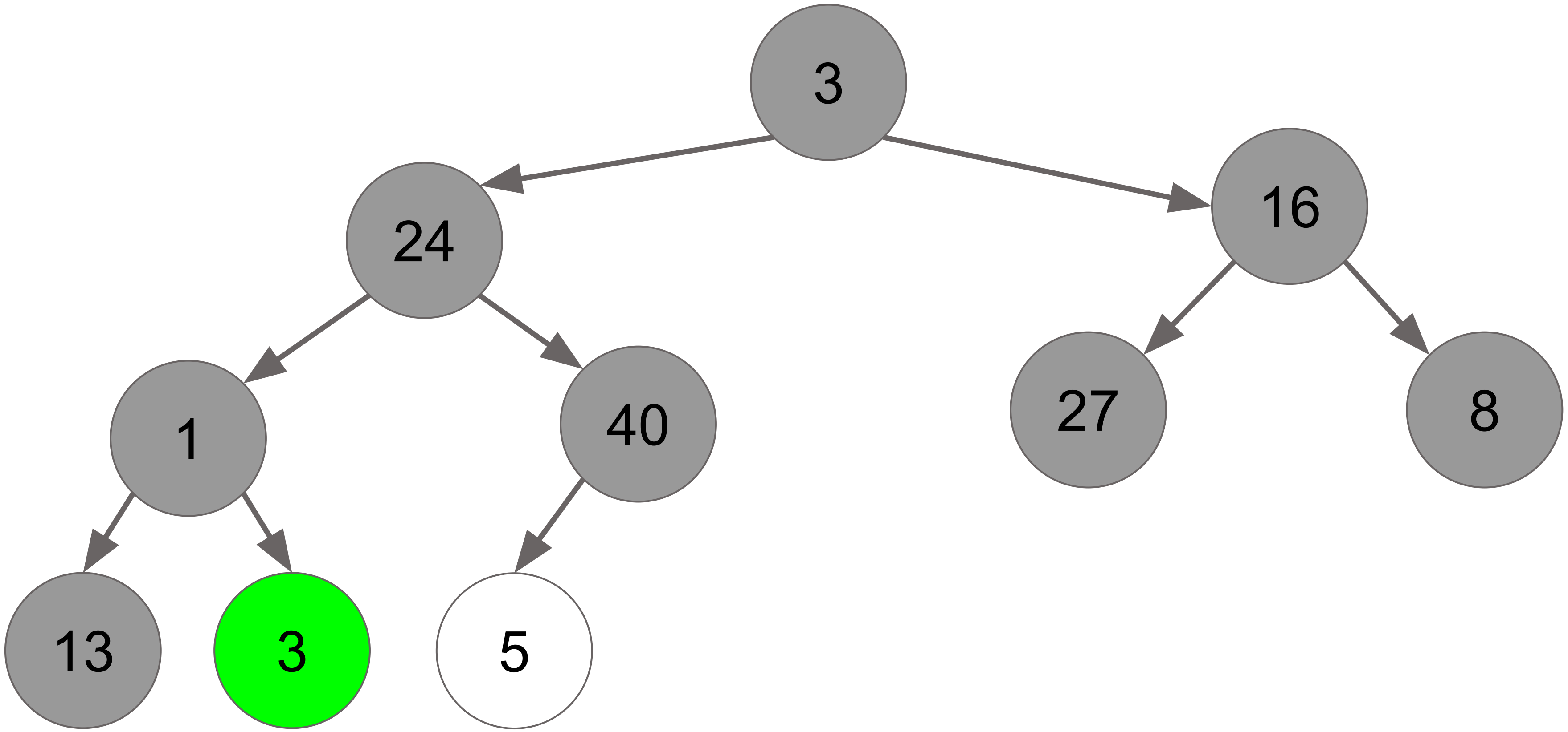
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



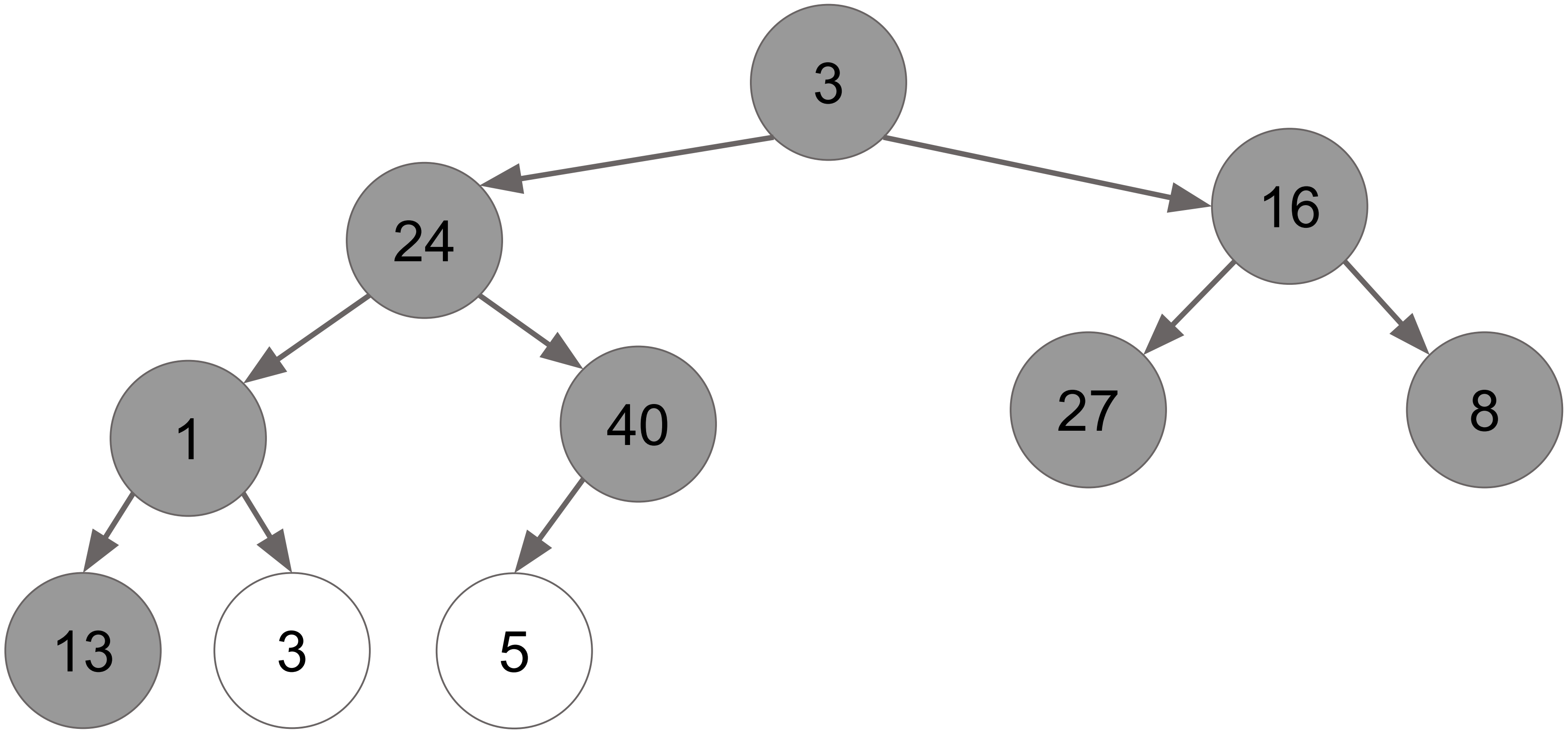
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



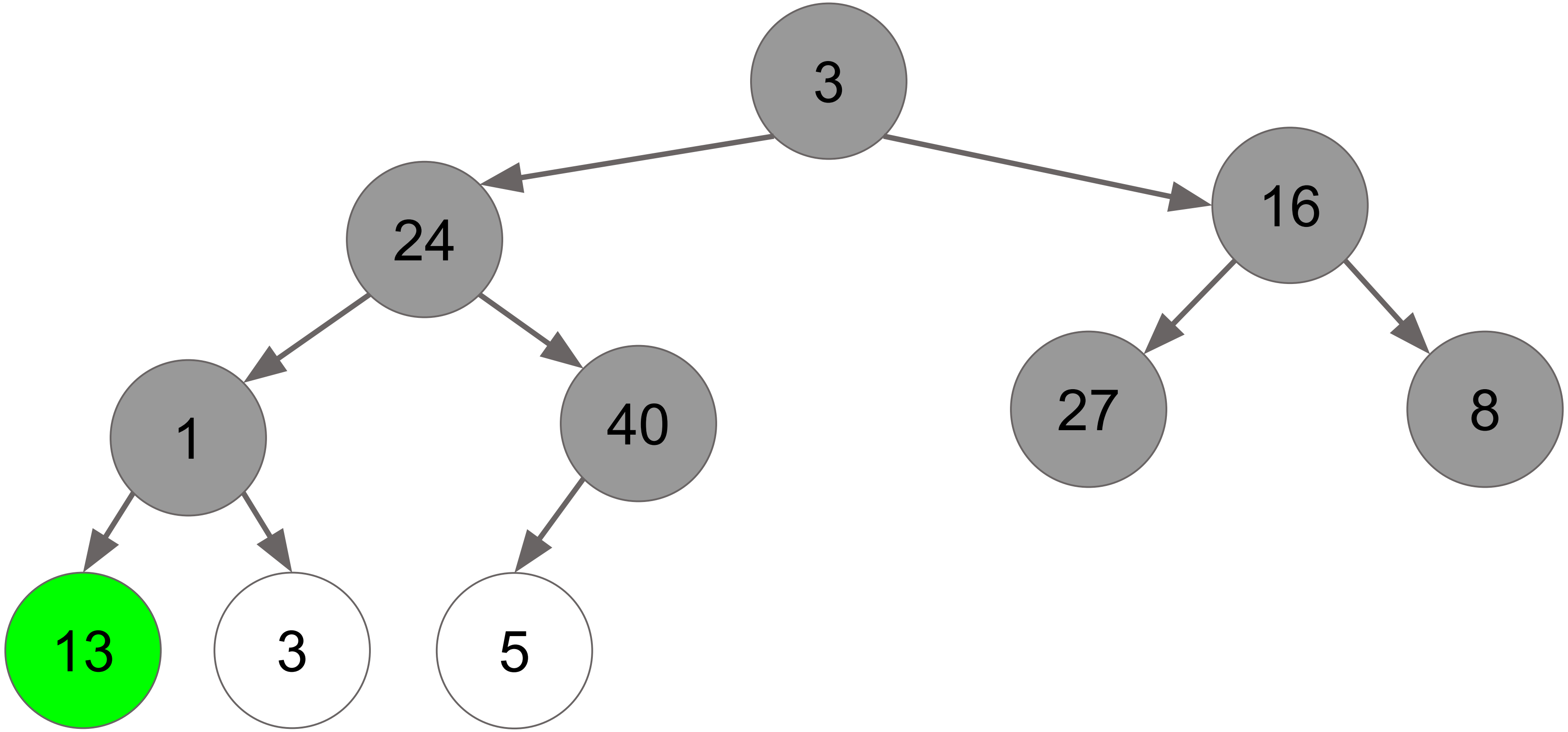
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



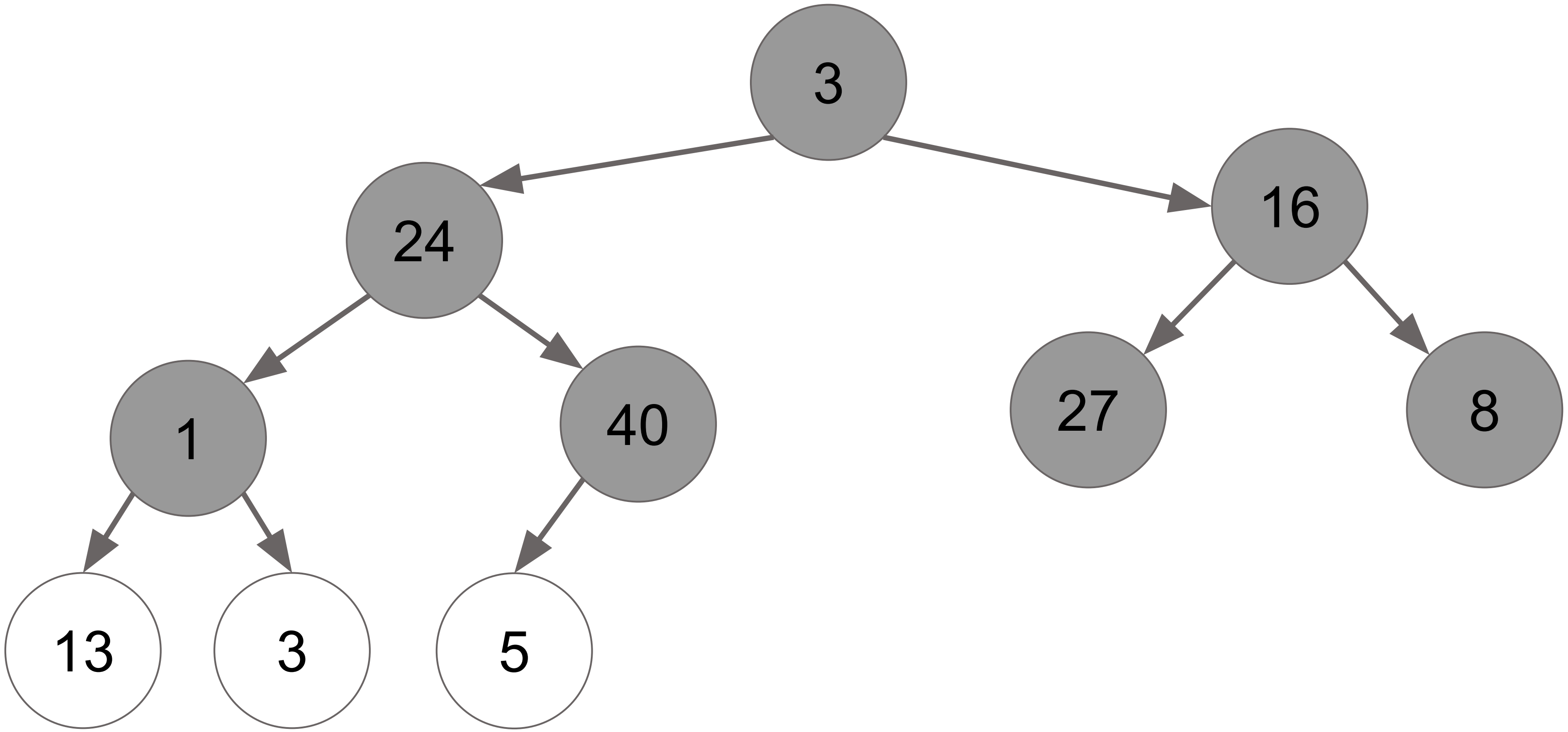
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



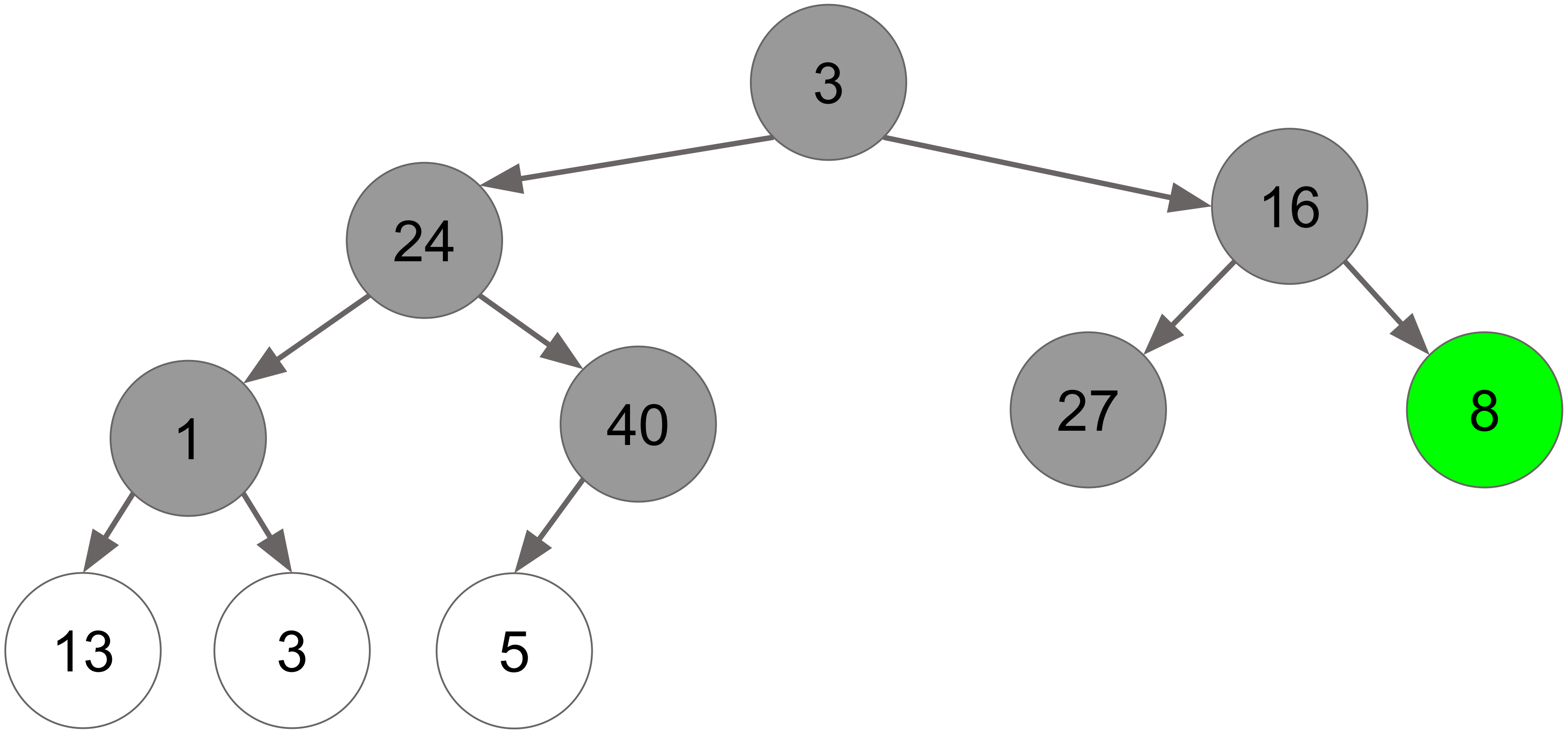
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



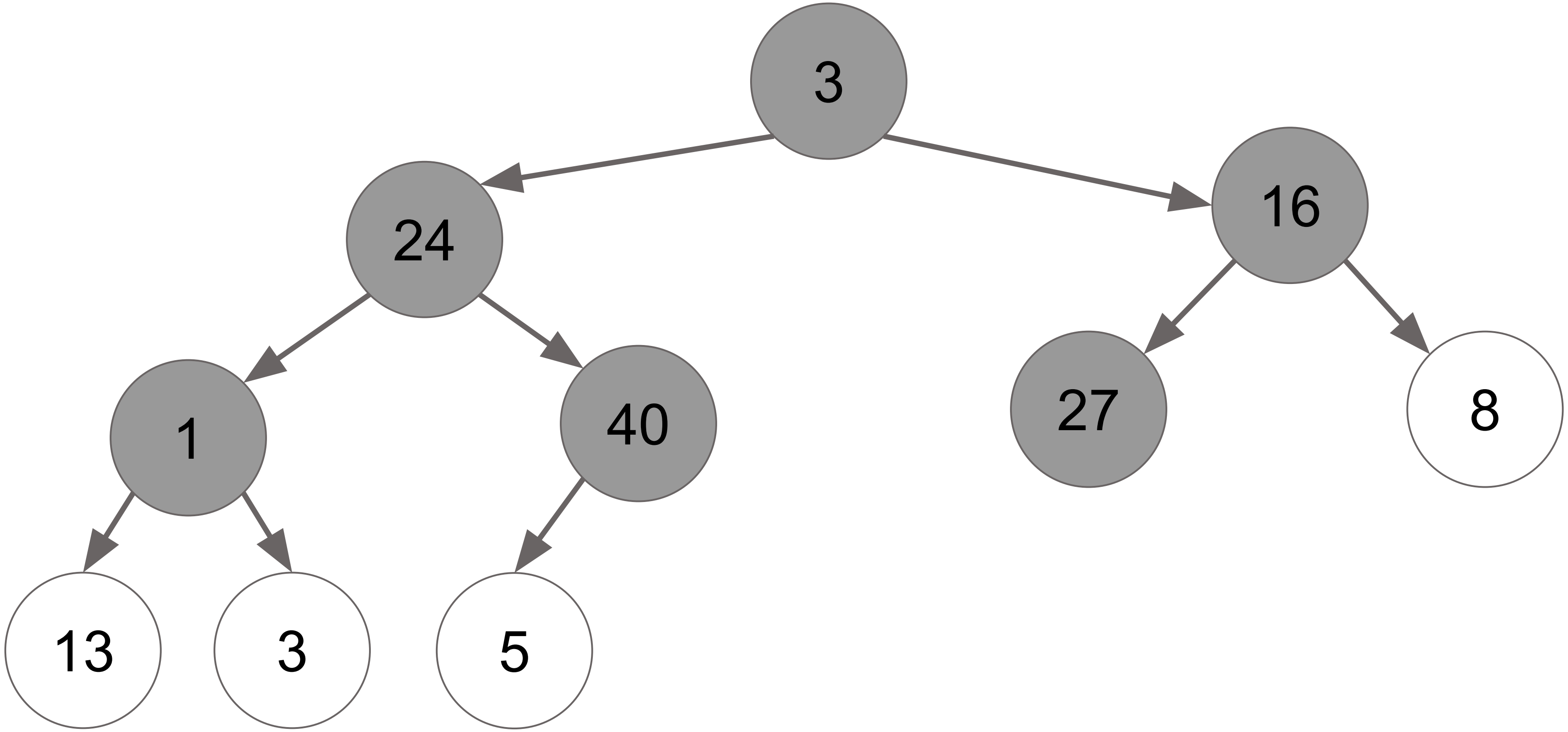
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



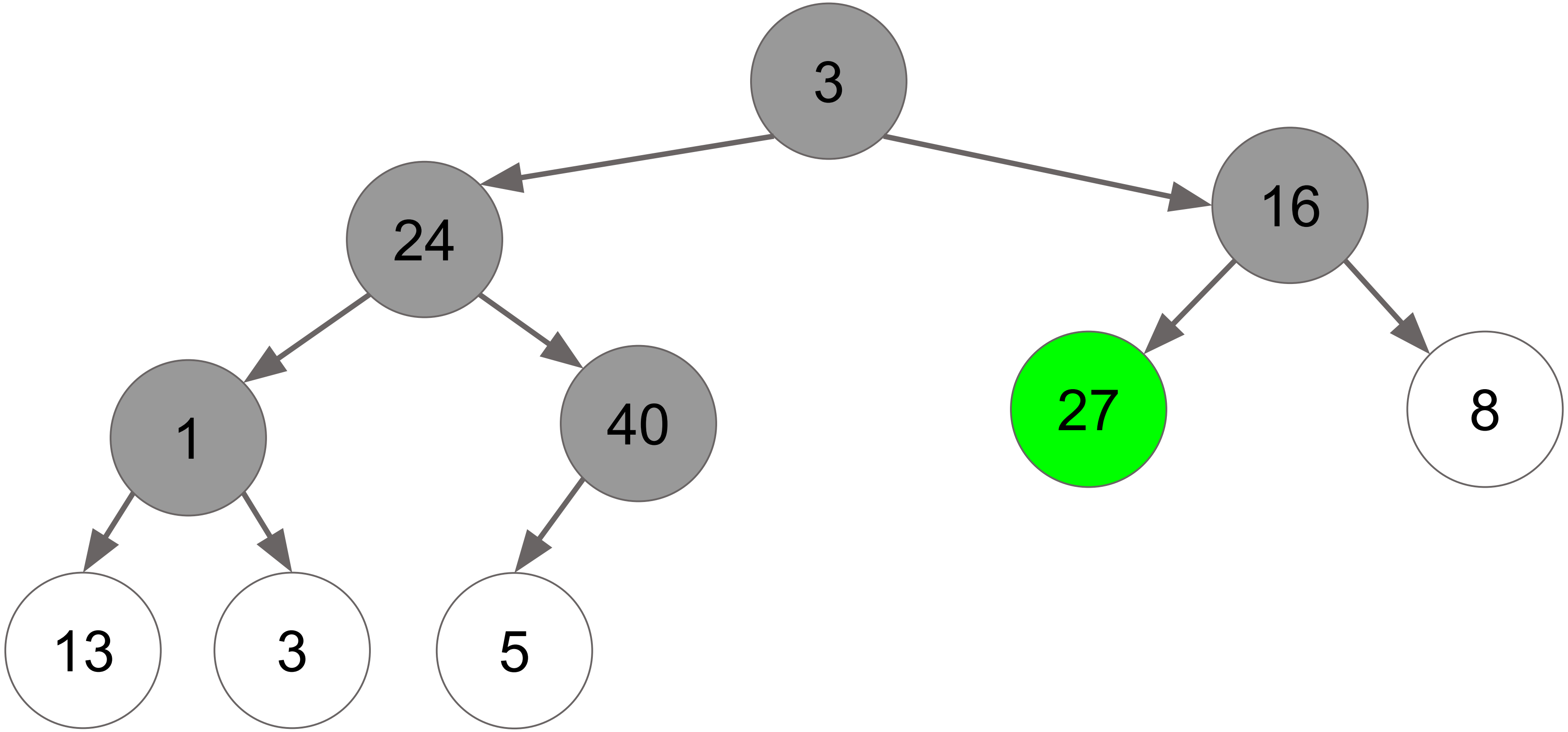
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



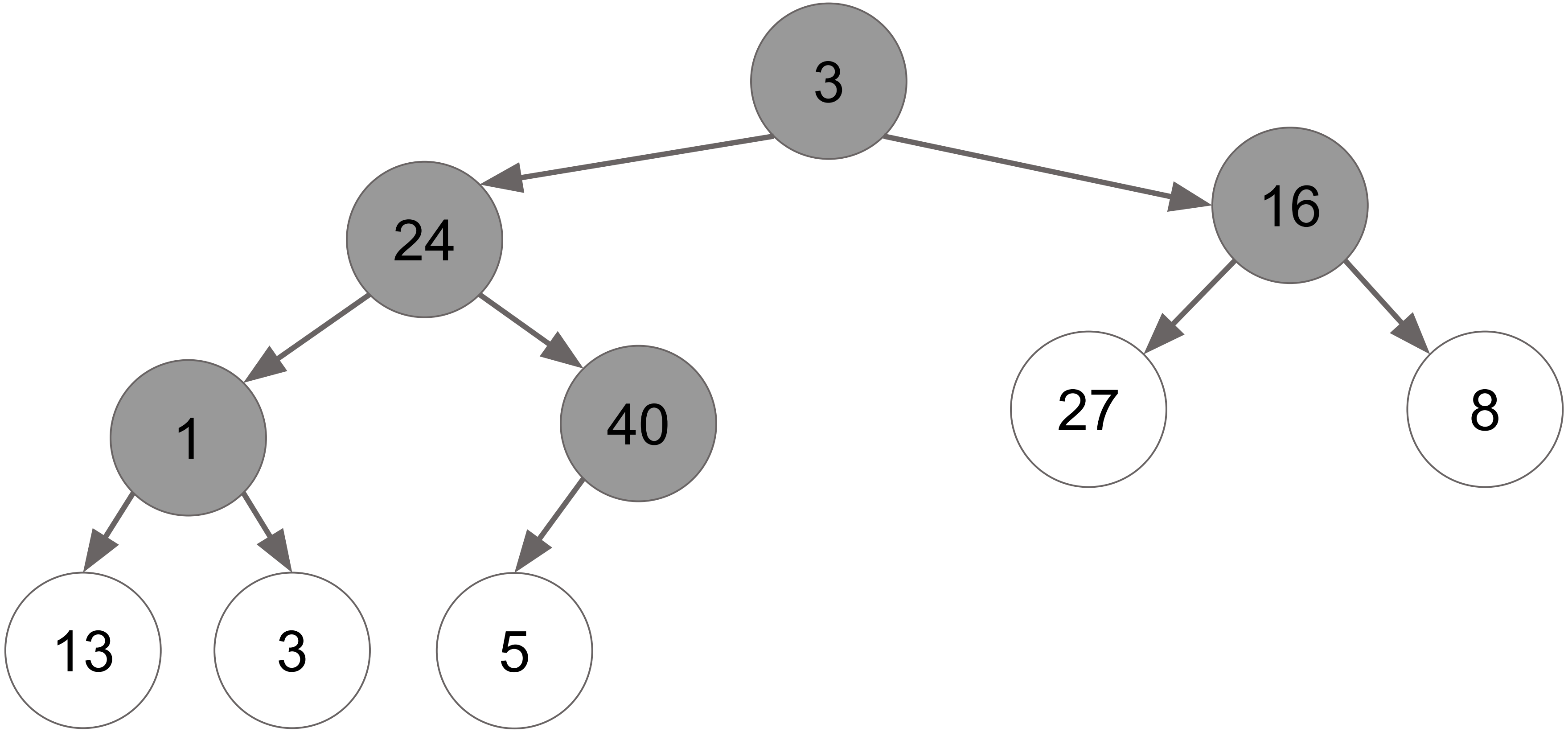
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



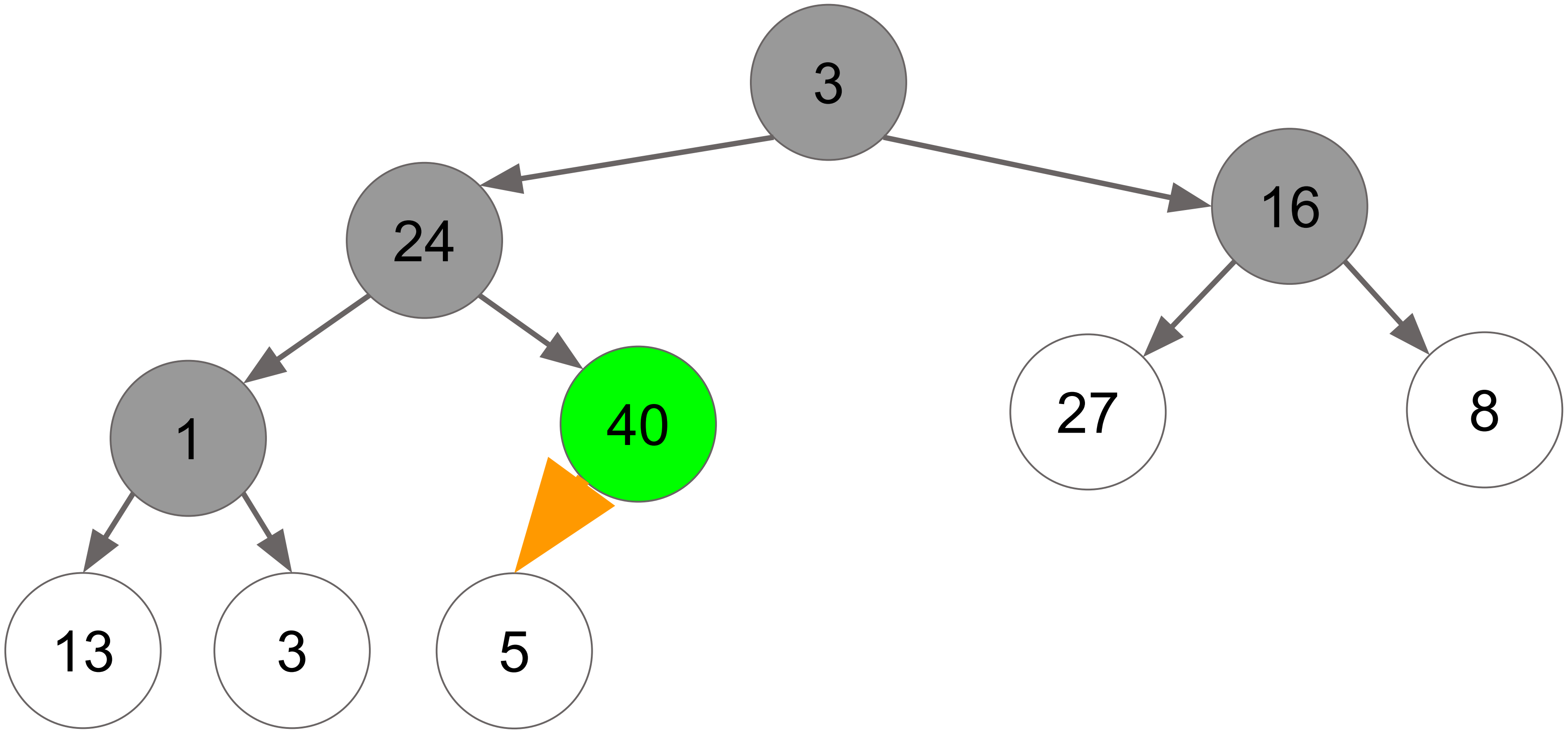
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



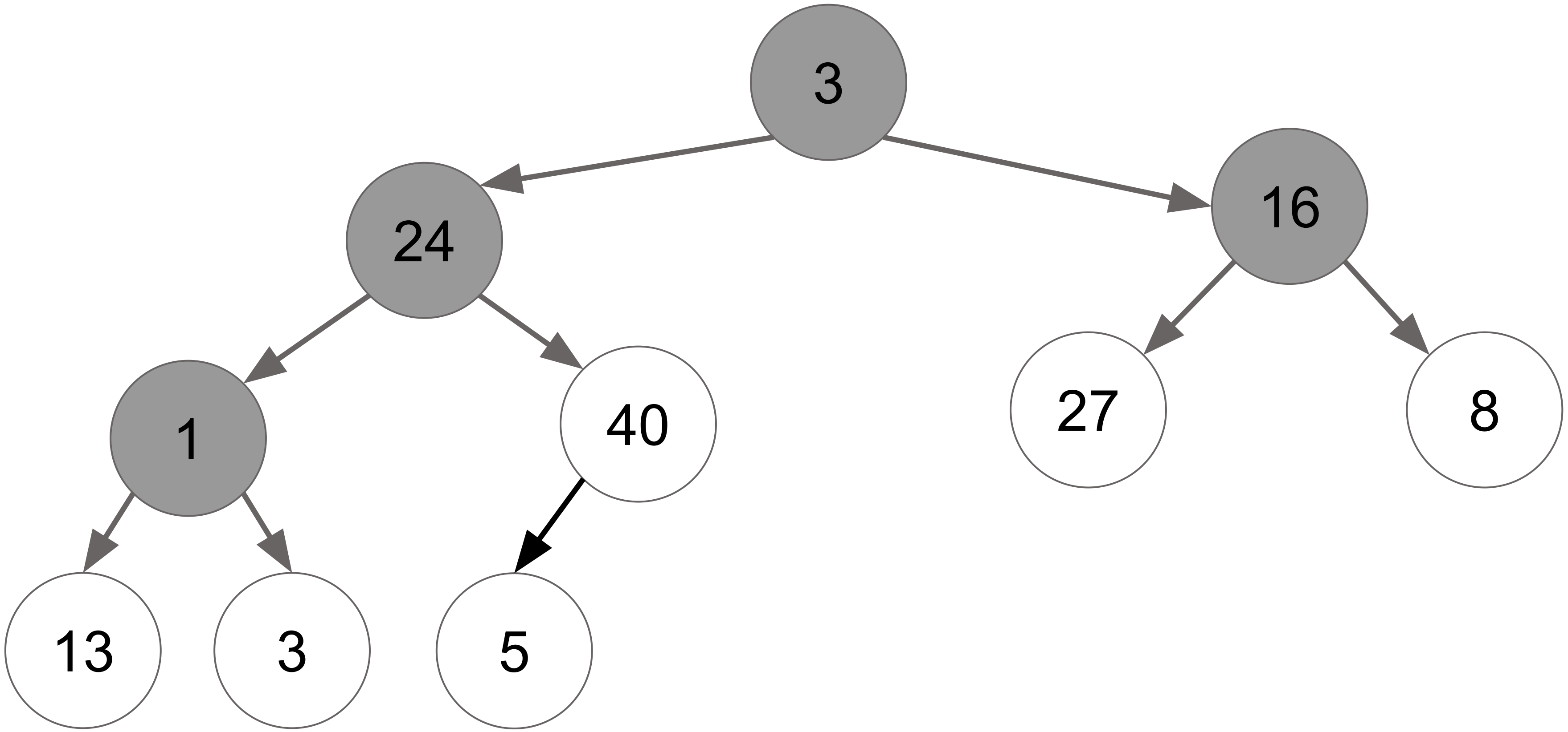
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



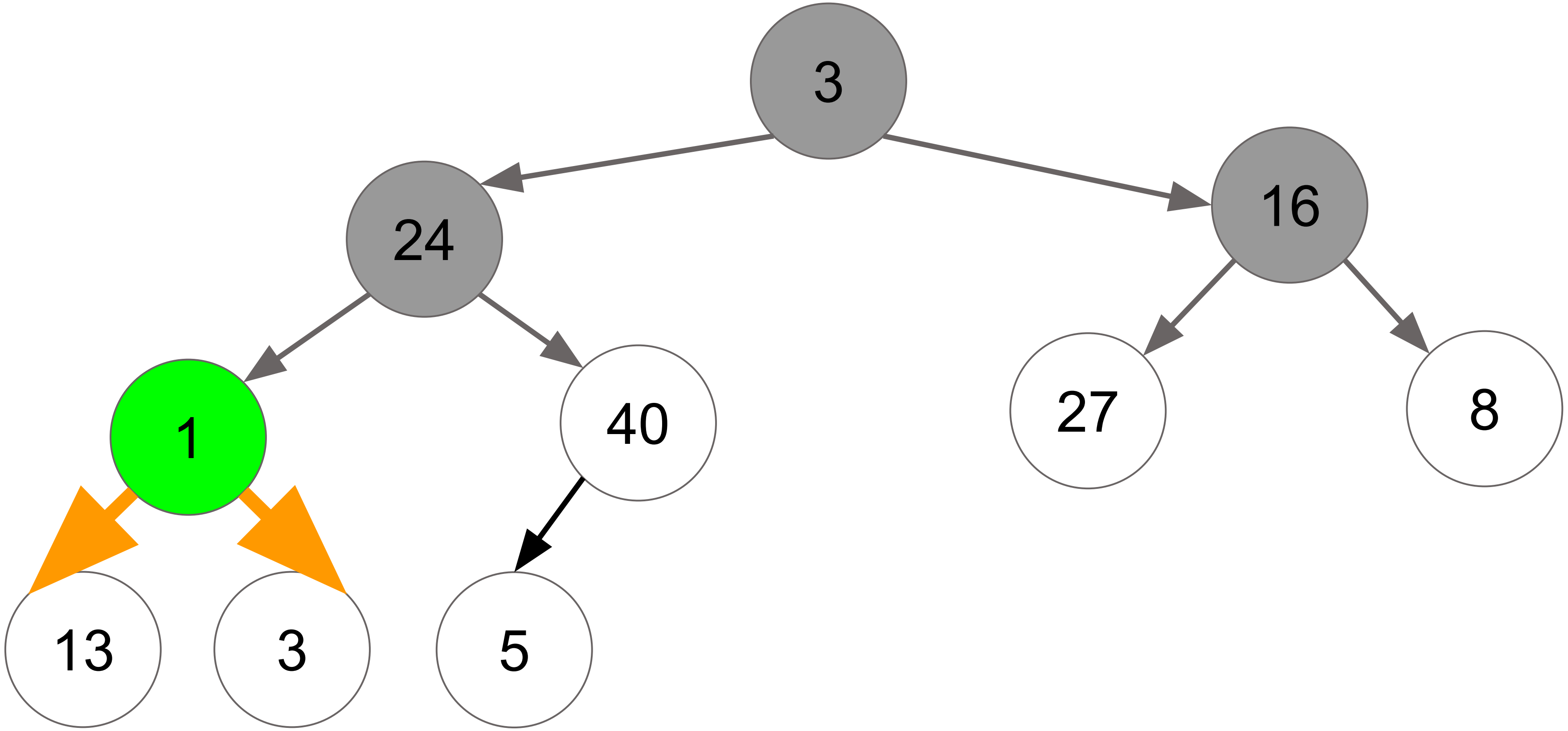
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



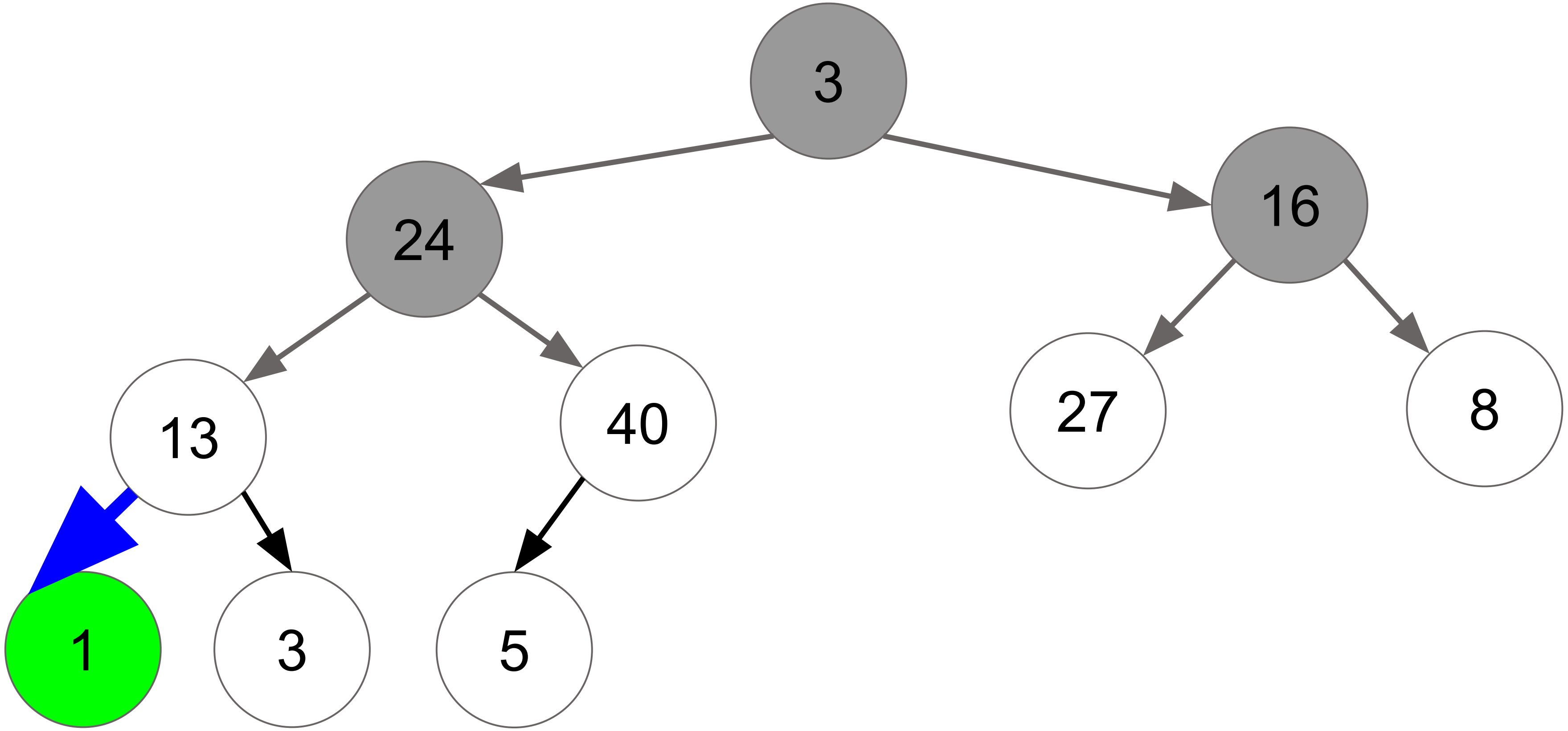
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



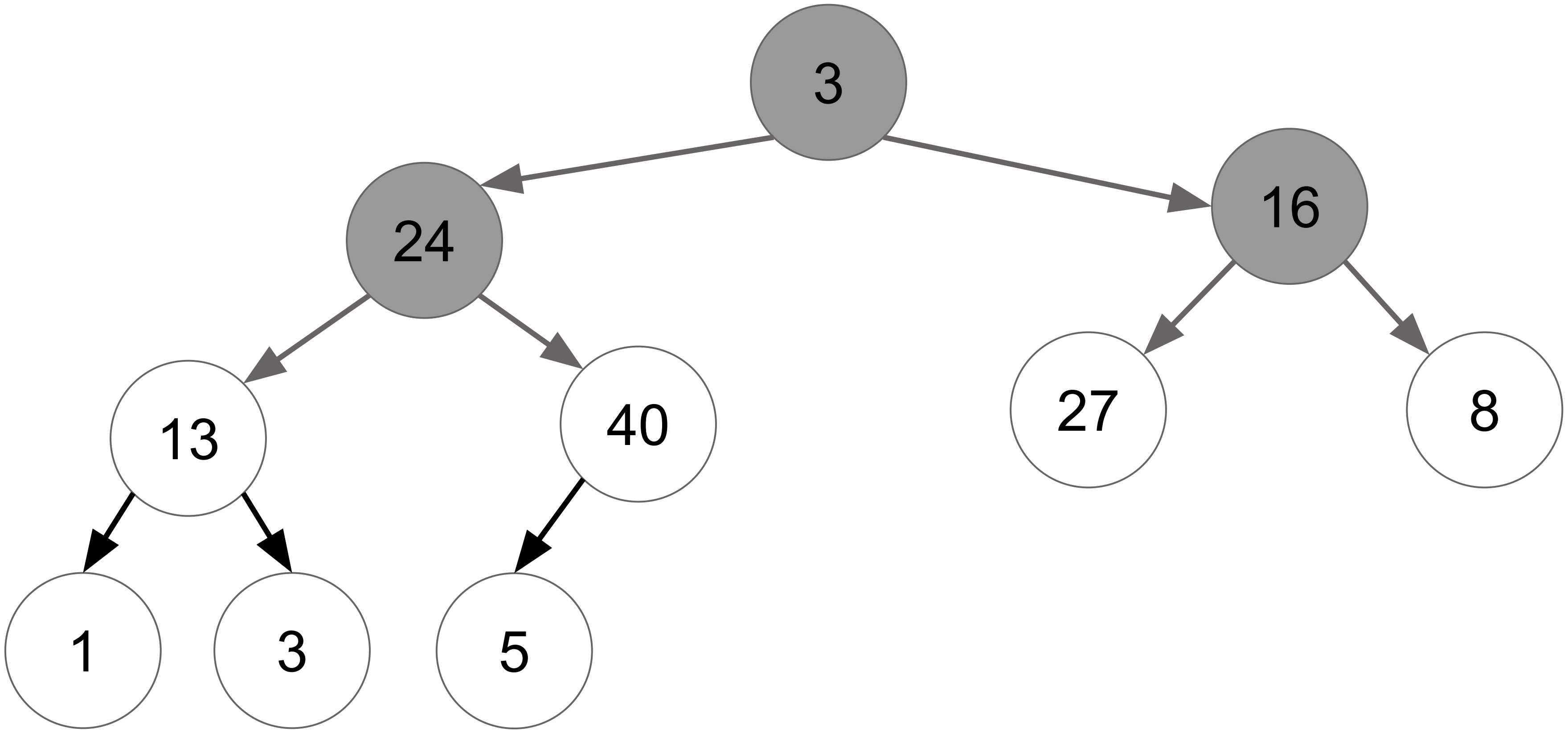
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



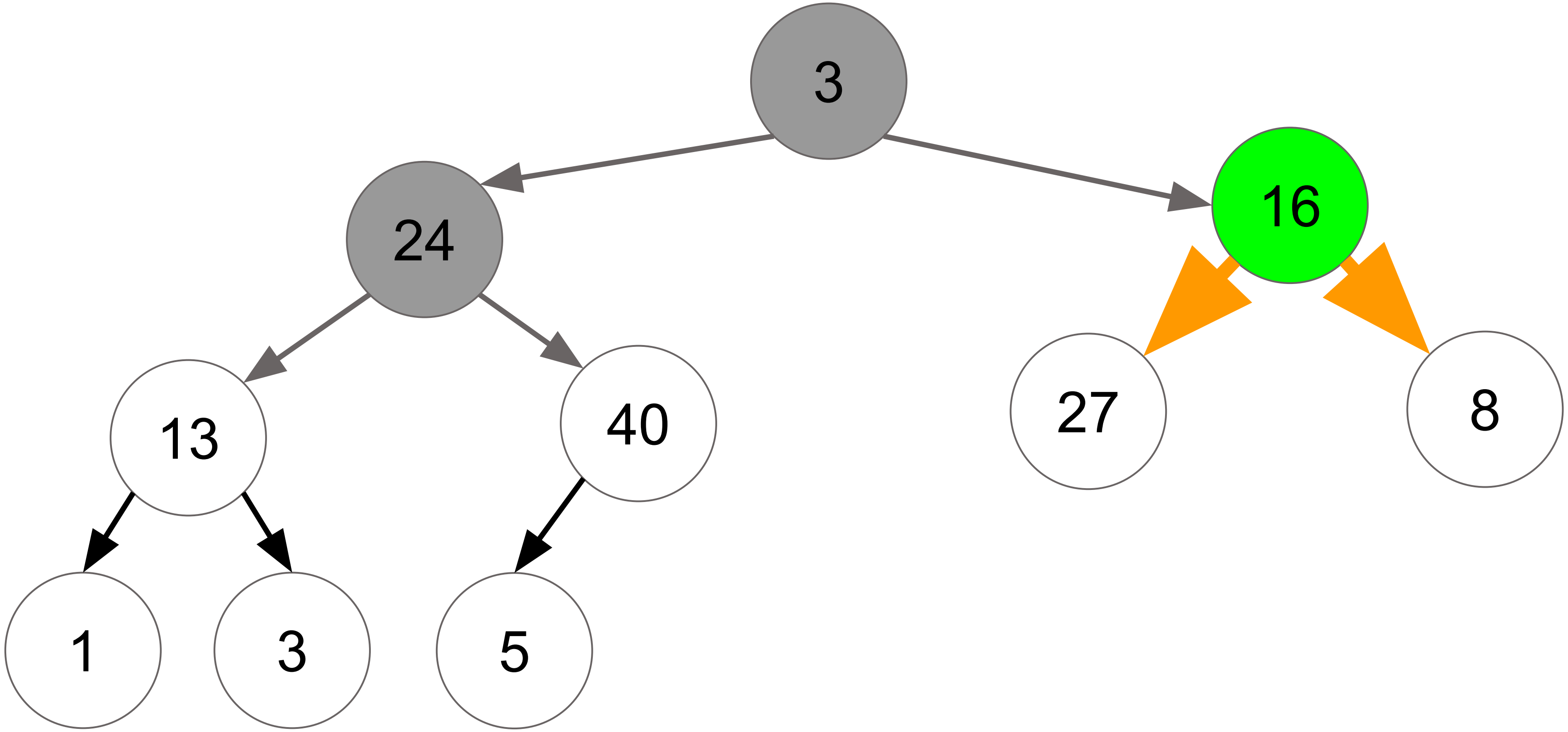
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



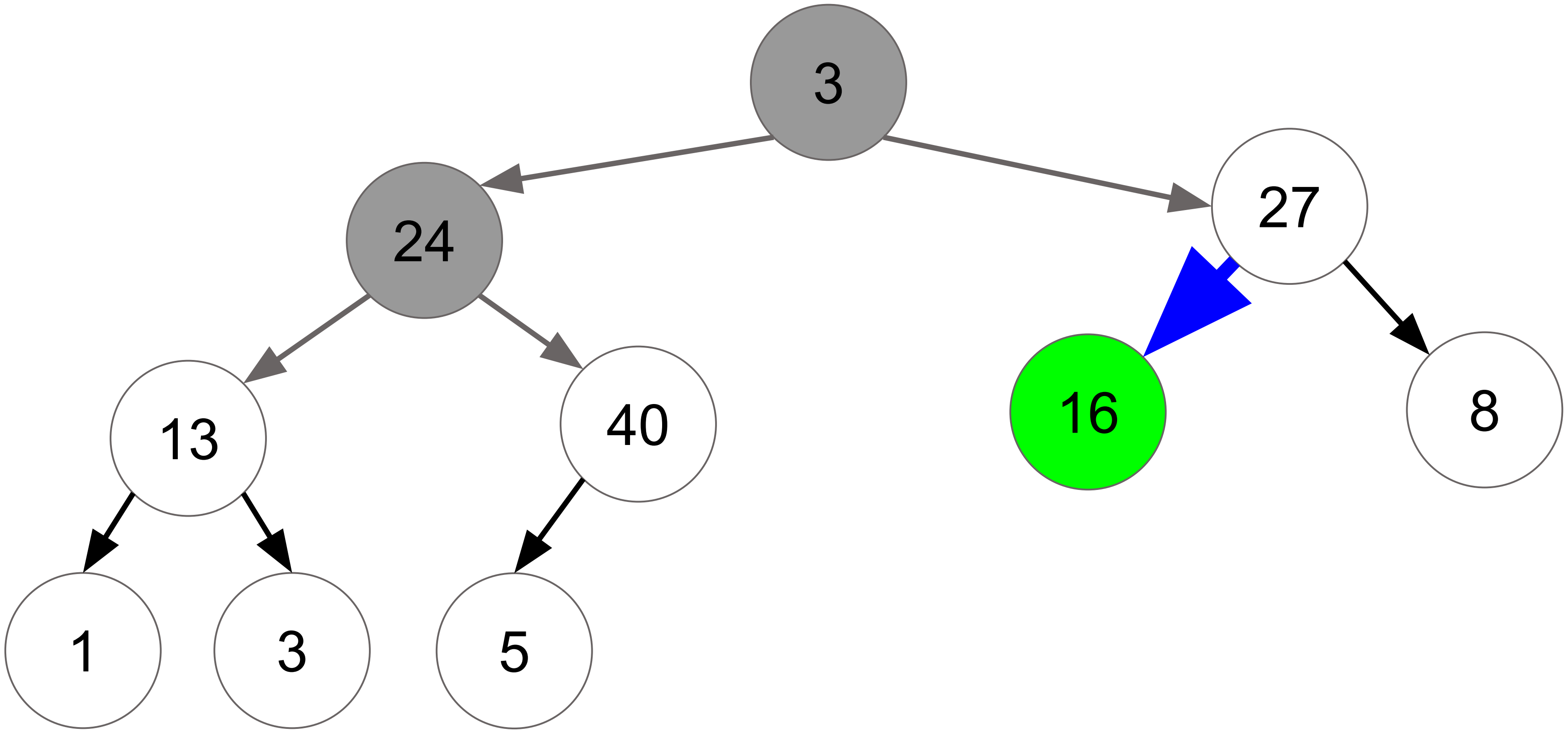
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



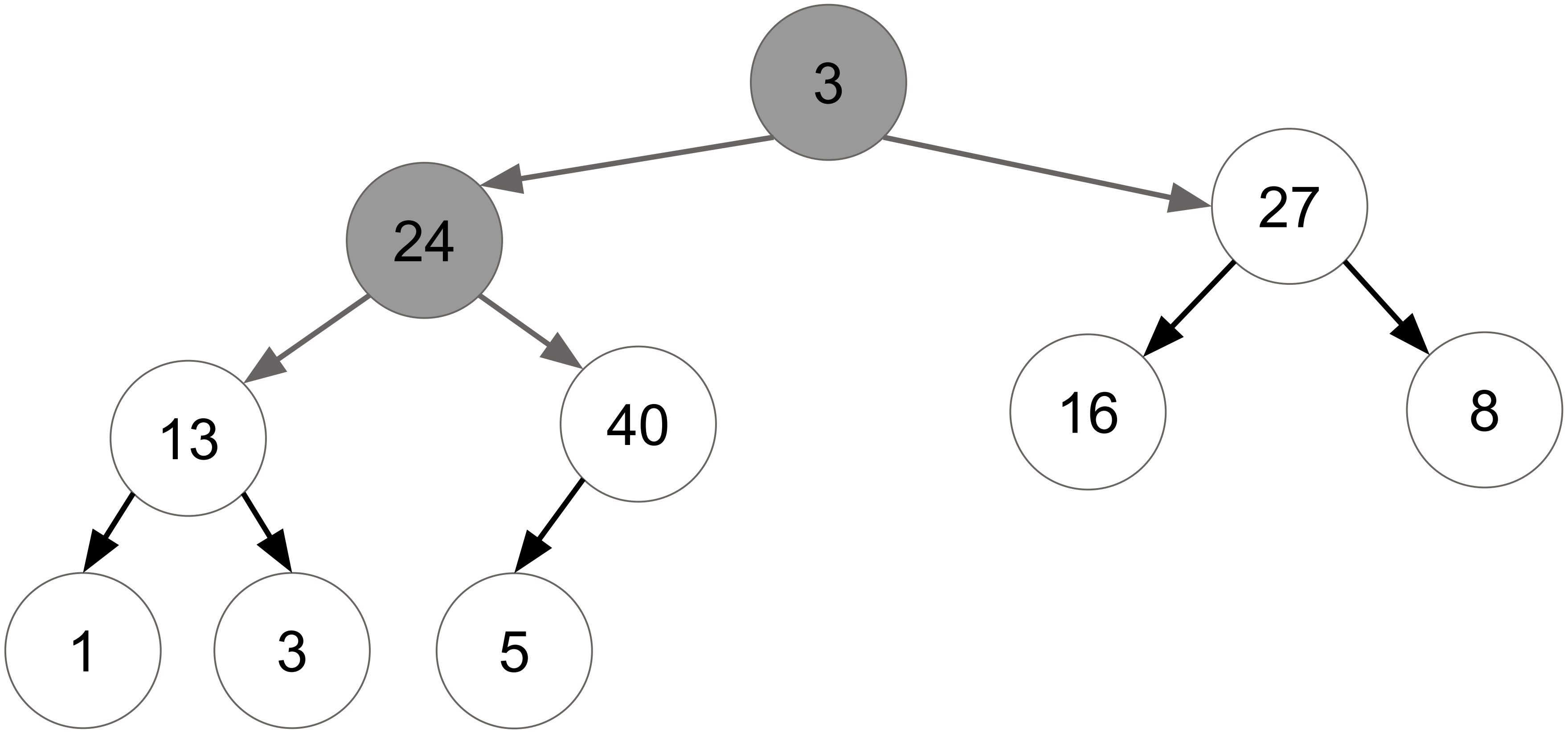
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



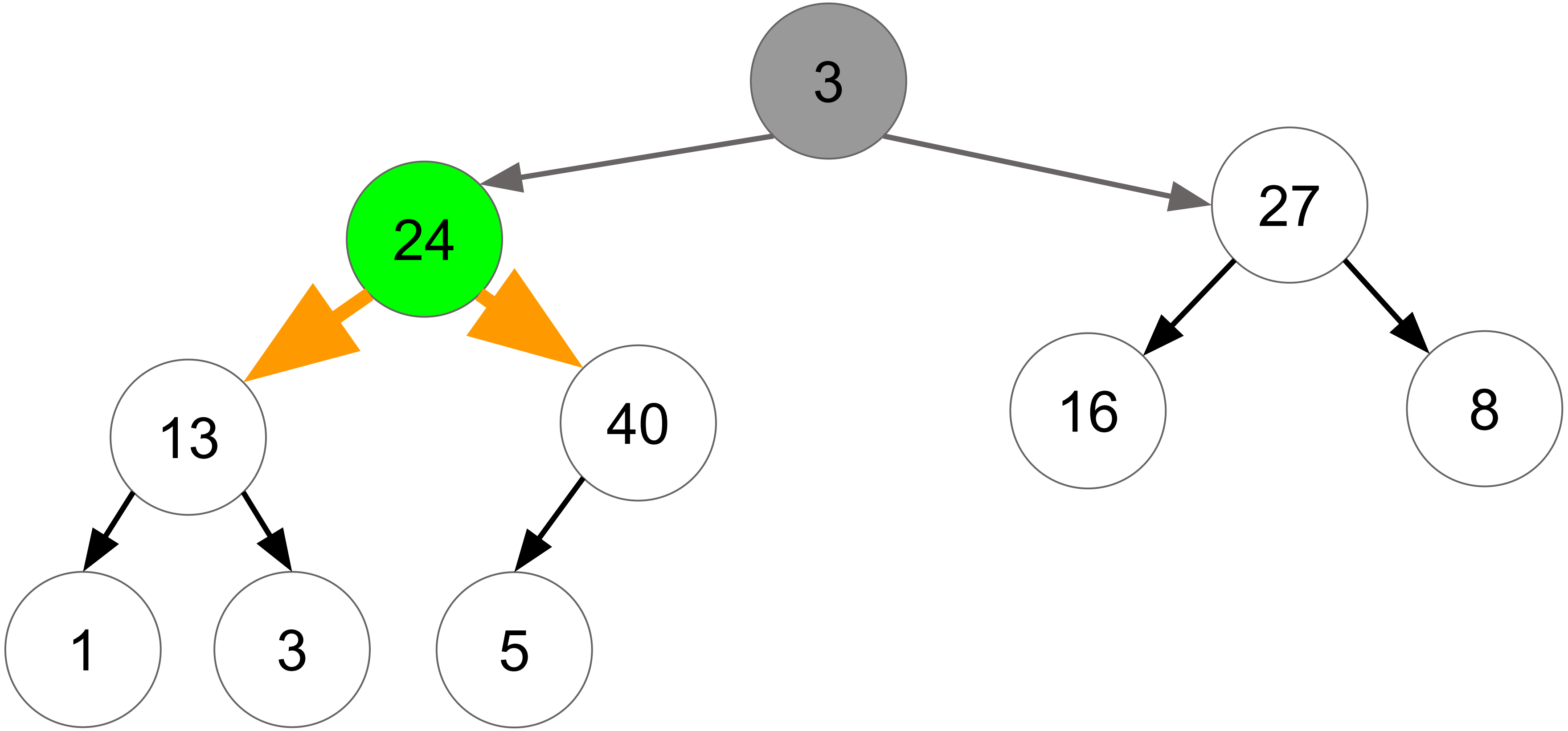
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



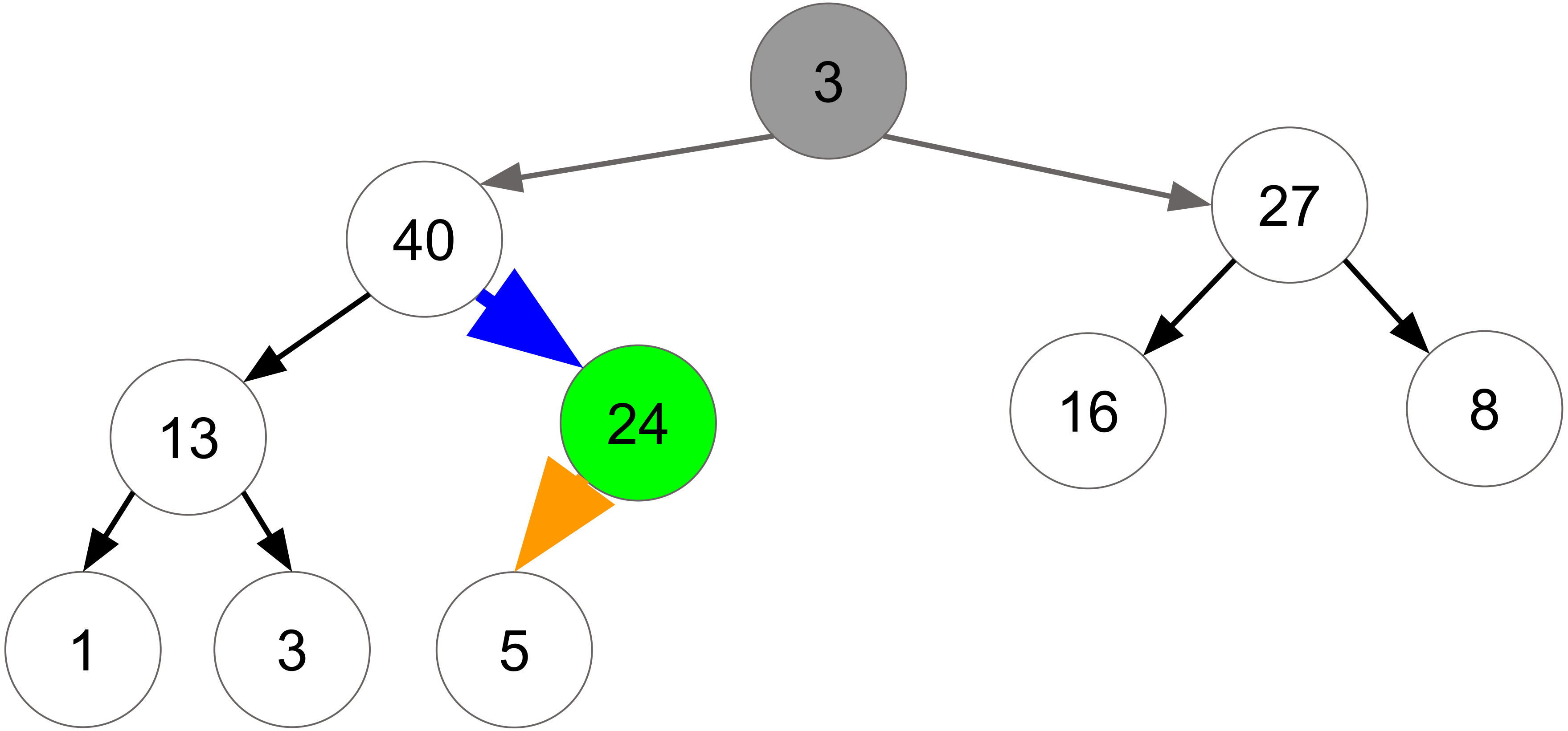
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



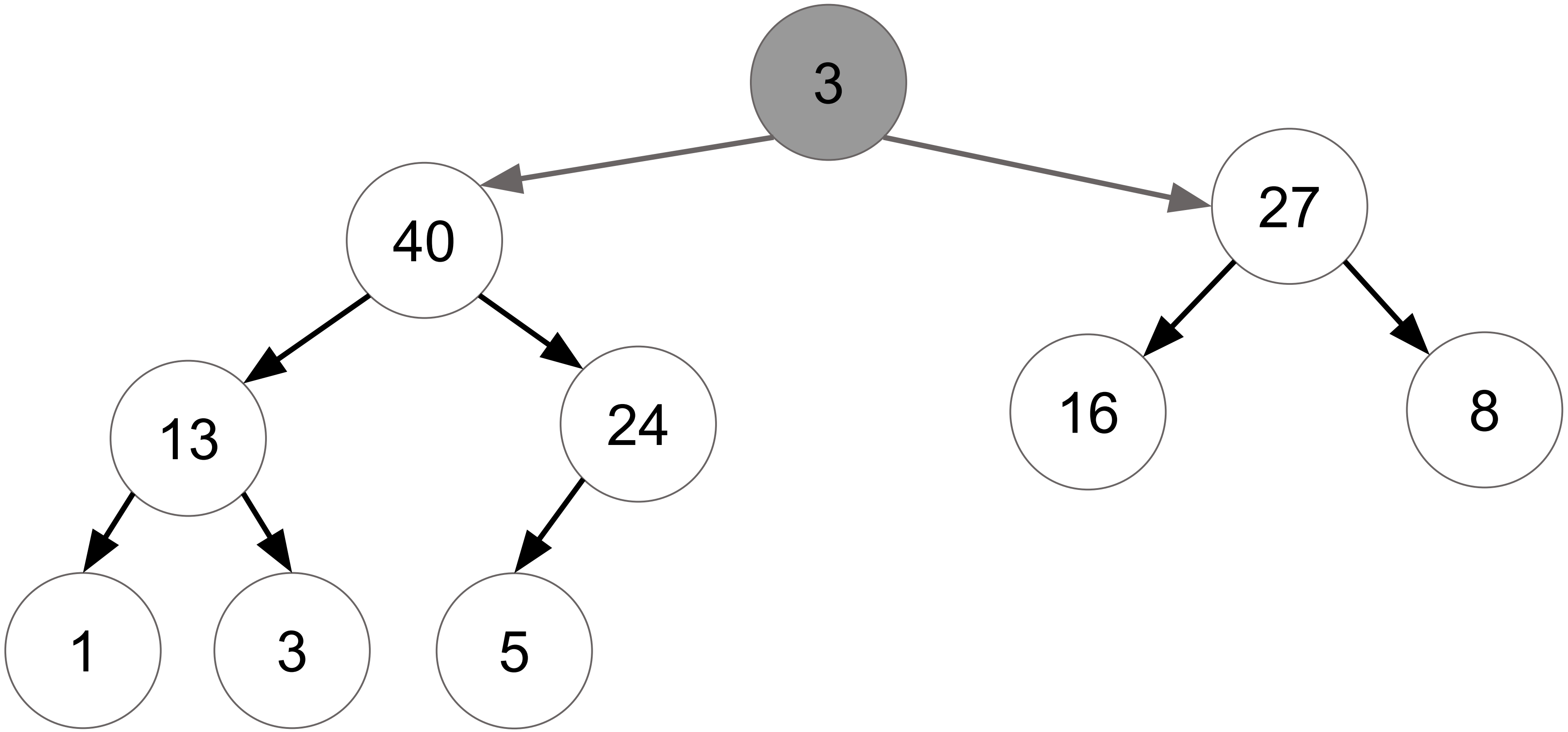
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



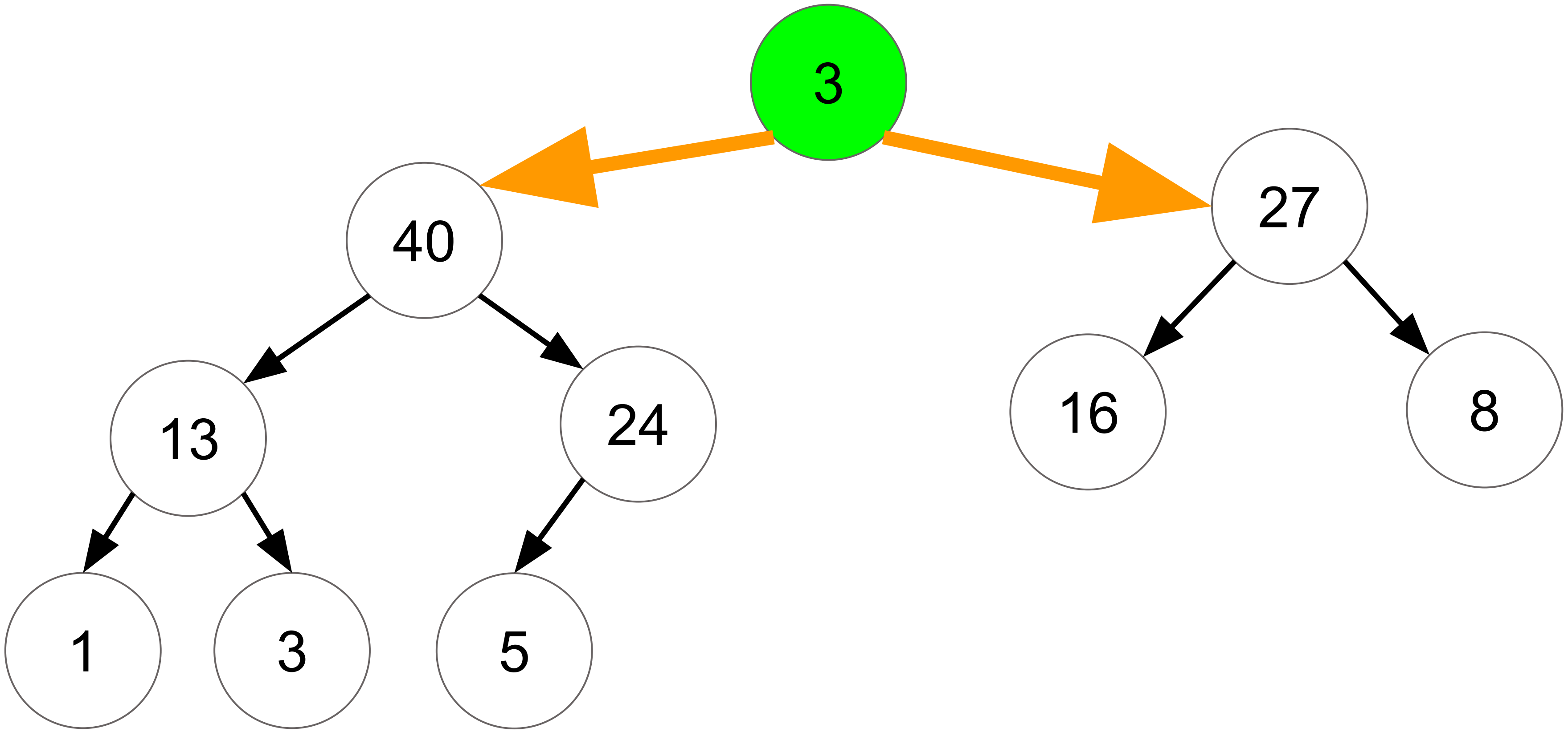
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



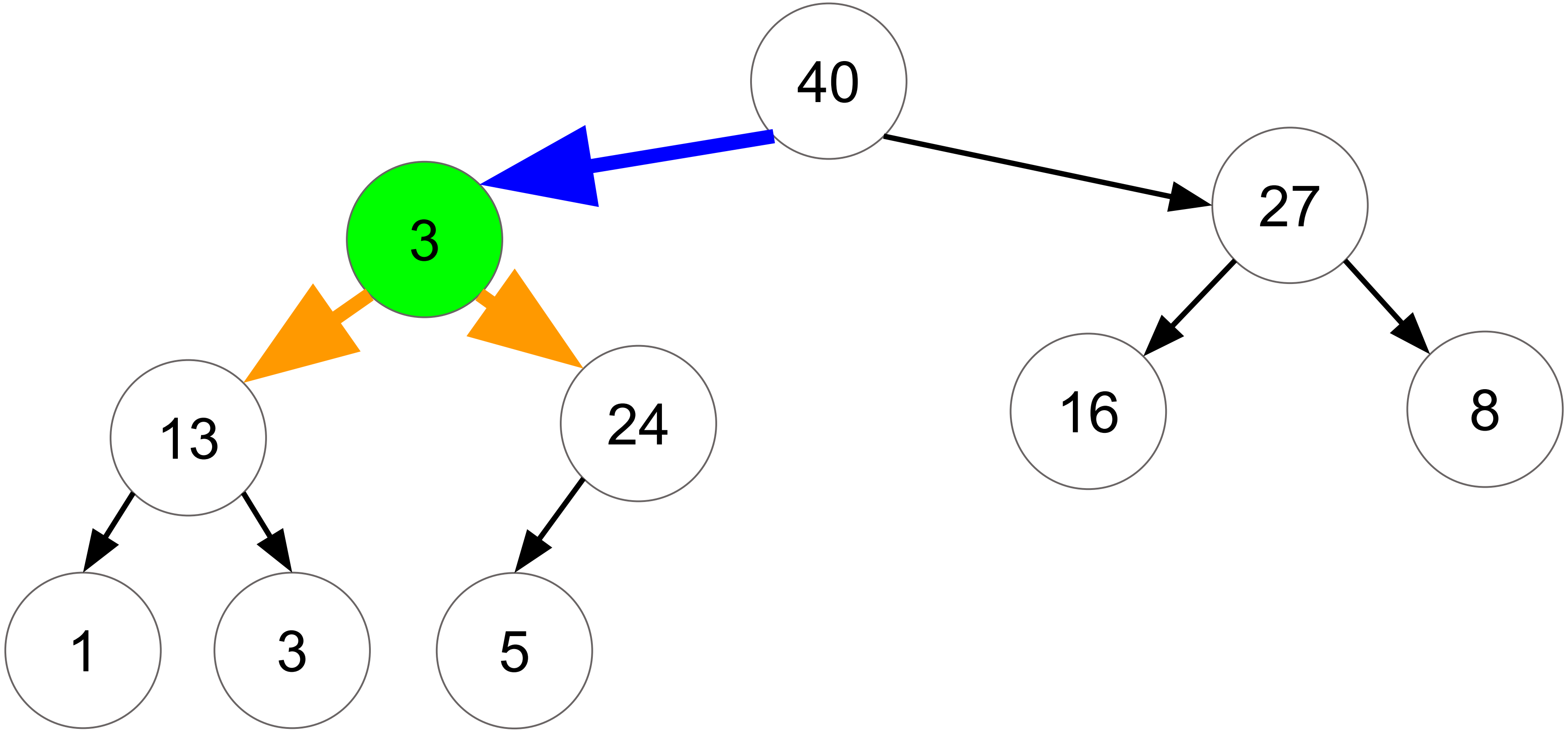
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



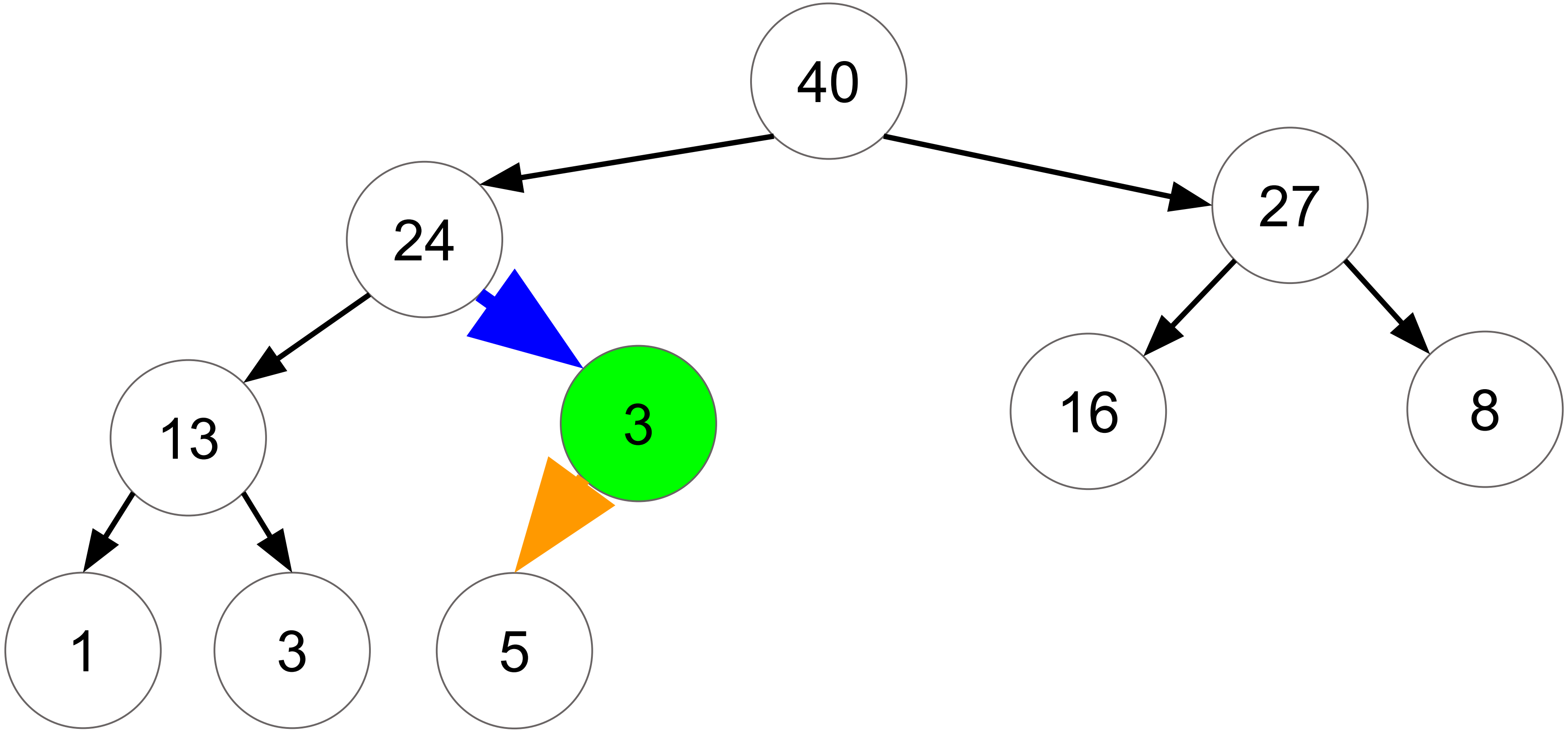
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



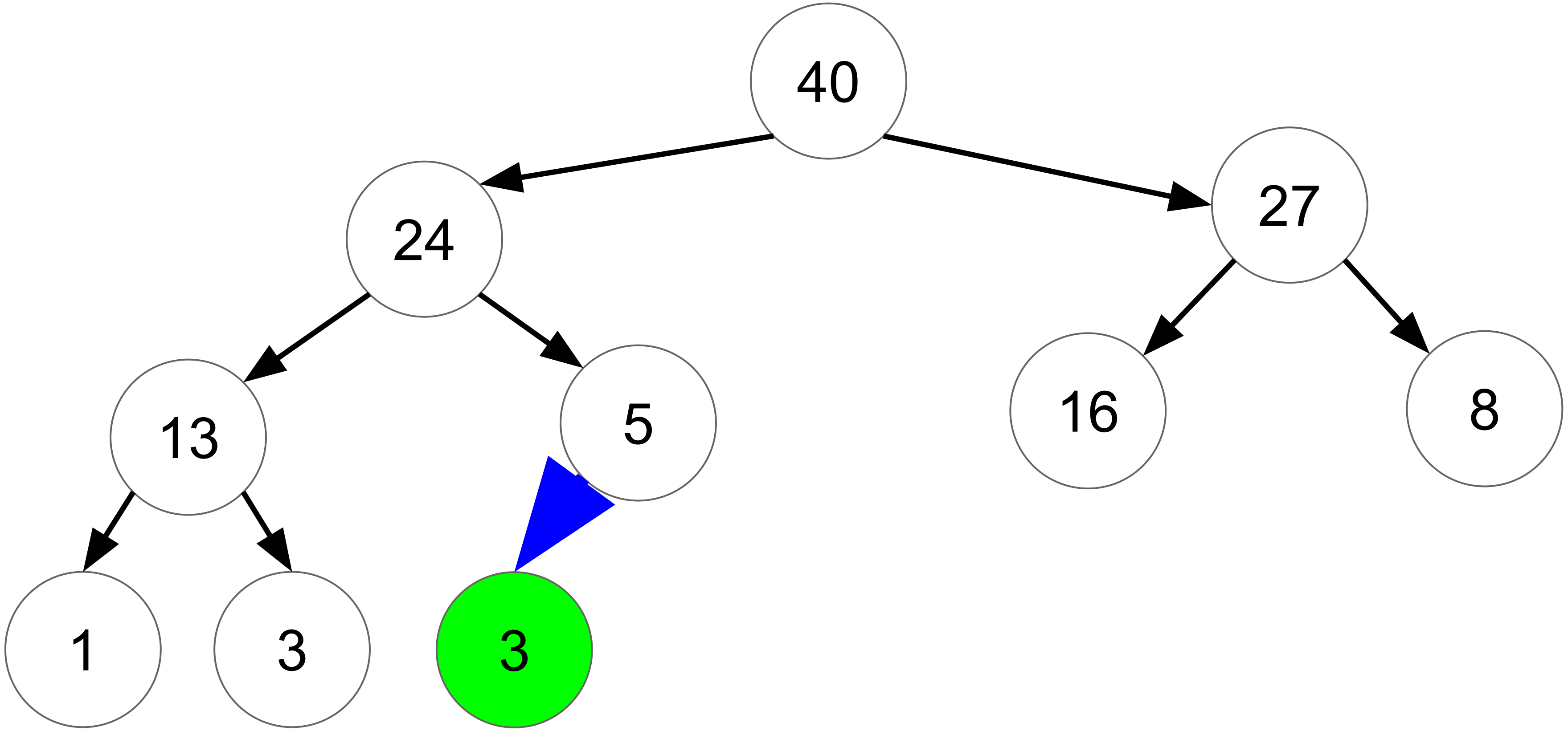
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



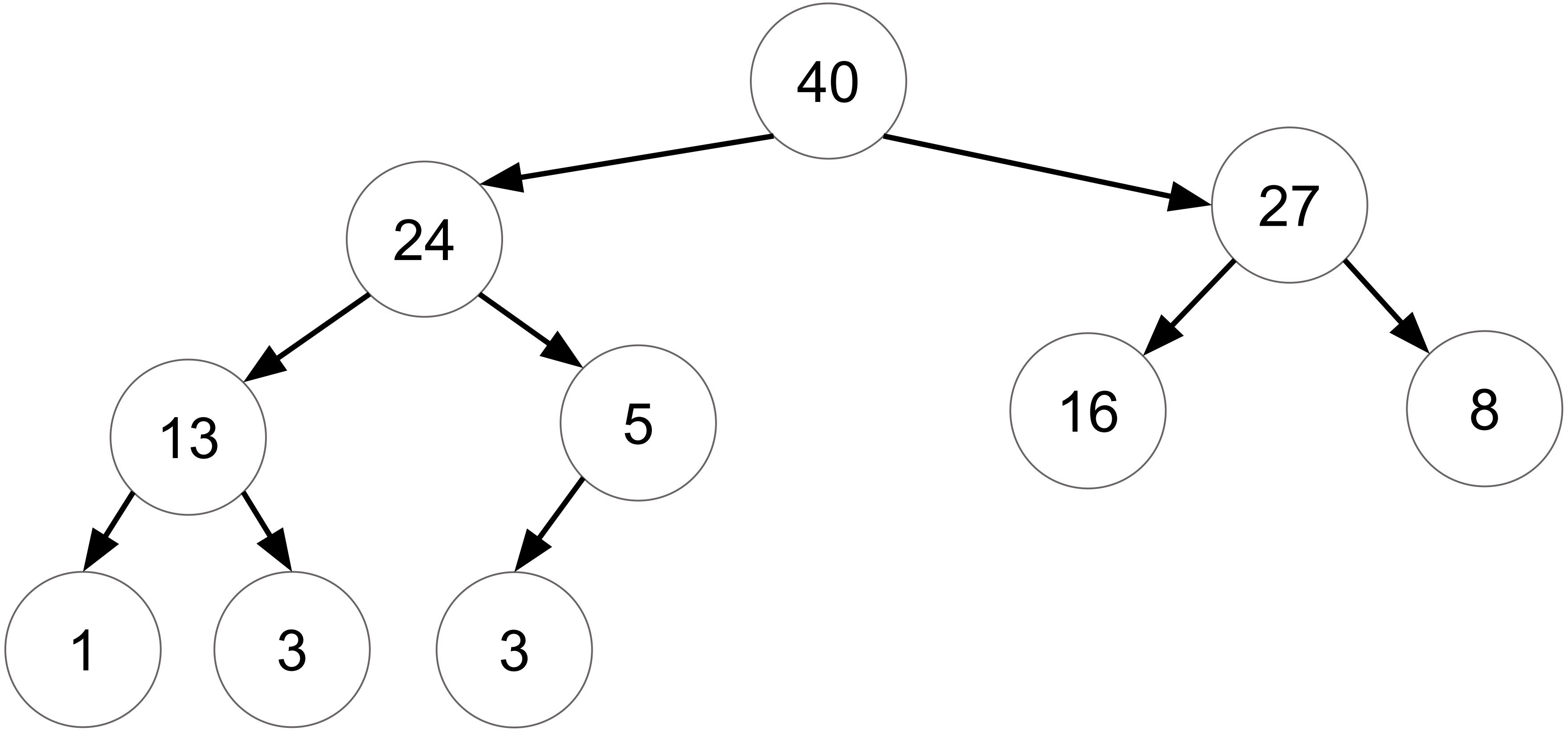
Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



Heapify Example: Fix Down From Bottom

- Fix this heap by continuously calling fix down, starting from the bottom:



Make Heap: Summary

- Calling fixDown() starting from the bottom has complexity O(n)
- Calling fixUp() starting from the top has complexity O(n log n)
- Key idea:
  - The bottom level of the heap has the <u>greatest number of items</u>
  - Calling fixDown() on these items would require no work, since we already know they are in the correct position
  - Calling fixUp() on these same items would require O(log n) work for each item, since these items are log n levels from the top of the tree
  - This means that the fixDown() method is more efficient!
  - We are effectively building many small heaps and merging them by adding new nodes, which costs O(log n) - but mostly on very small heaps, limiting the work

Heapsort

Heapsort

- Uses a binary heap to sort items:
  - Build a heap using make heap / heapify
  - identify largest element, move it to the end, find next largest element, move it to the end, repeat...

Heapsort

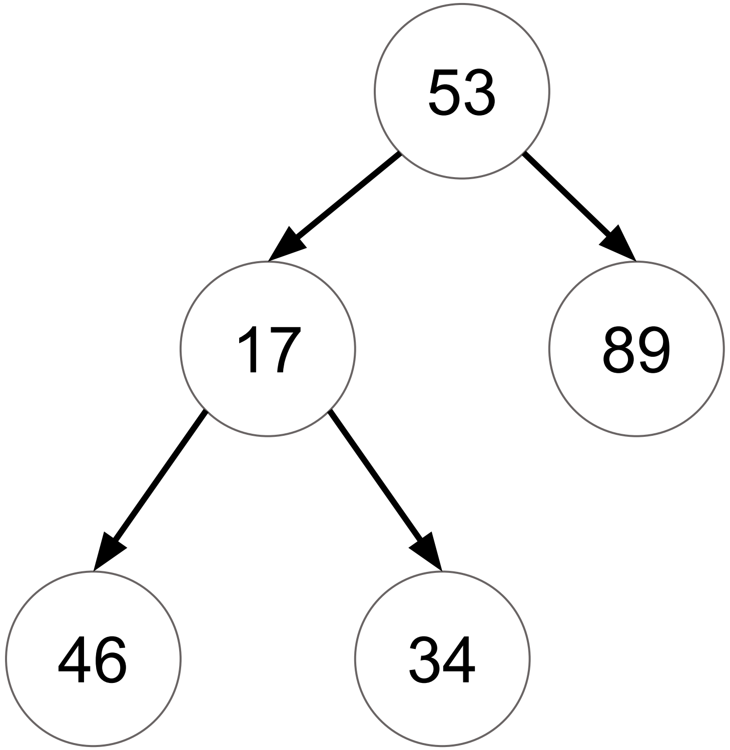
- Uses a binary heap to sort items:
  - Build a heap using make heap / heapify
  - identify largest element, move it to the end, find next largest element, move it to the end, repeat...

- The complexity of sorting n items is O(n log n):
  - O(n) for the heapify process
  - O(log n) for each removal (since you call fixDown each time you remove an item): since this is done n times, the total complexity of this process is O(n log n)
  - **O(n + n log n) = O(n log n)** - the O(n log n) complexity of removal+fixDown dominates the O(n) complexity of heapify

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]



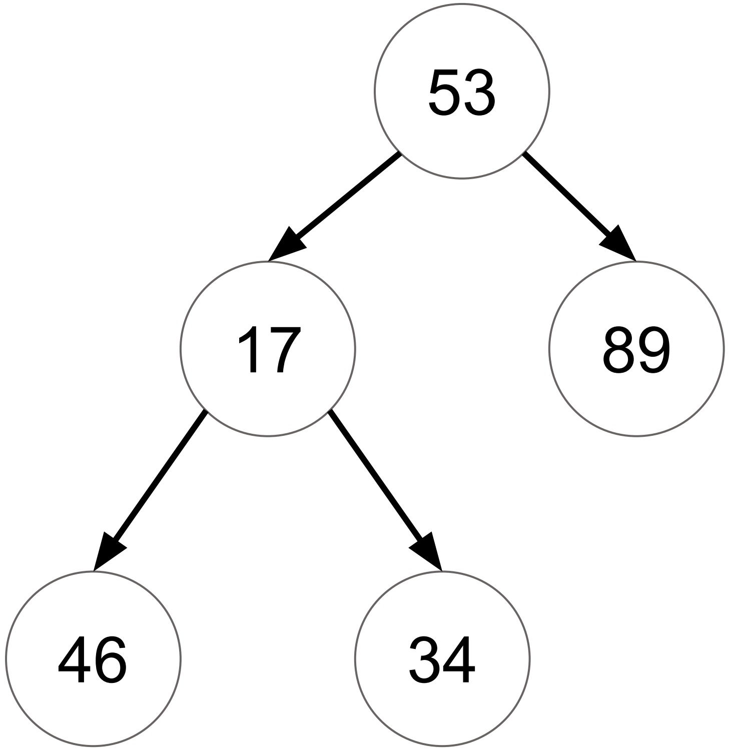
Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

We want to find the largest element so that we can move it to the end...

First step: heapify this into a max heap!



Array Representation

53	17	89	46	34

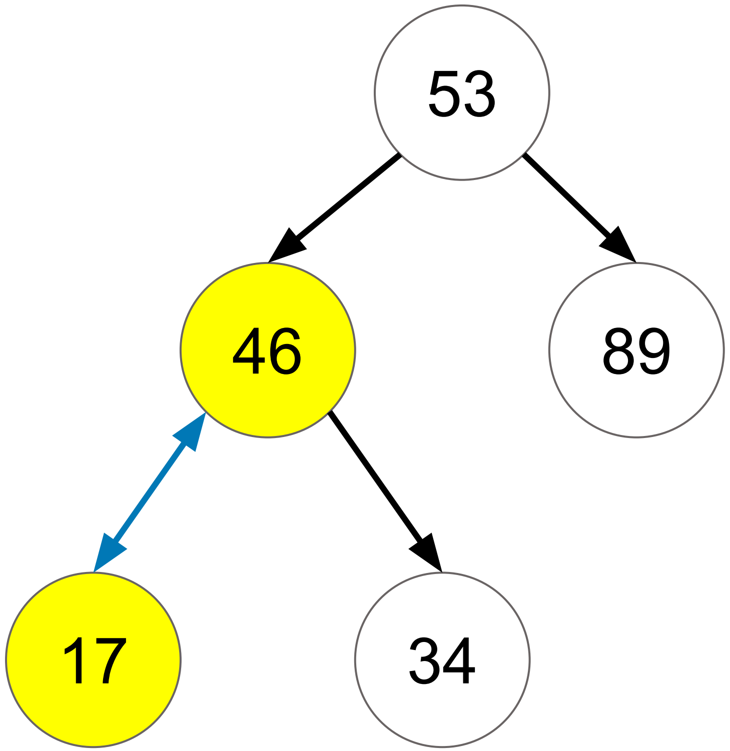
Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

We want to find the largest element so that we can move it to the end...

First step: heapify this into a max heap!



Array Representation

53	46	89	17	34

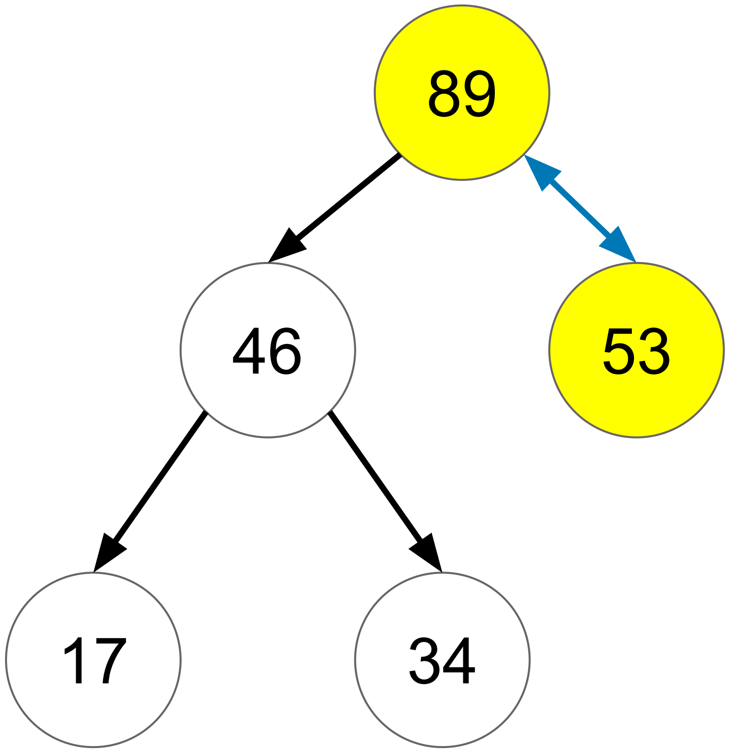
Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

We want to find the largest element so that we can move it to the end...

First step: heapify this into a max heap!



Array Representation

89	46	53	17	34

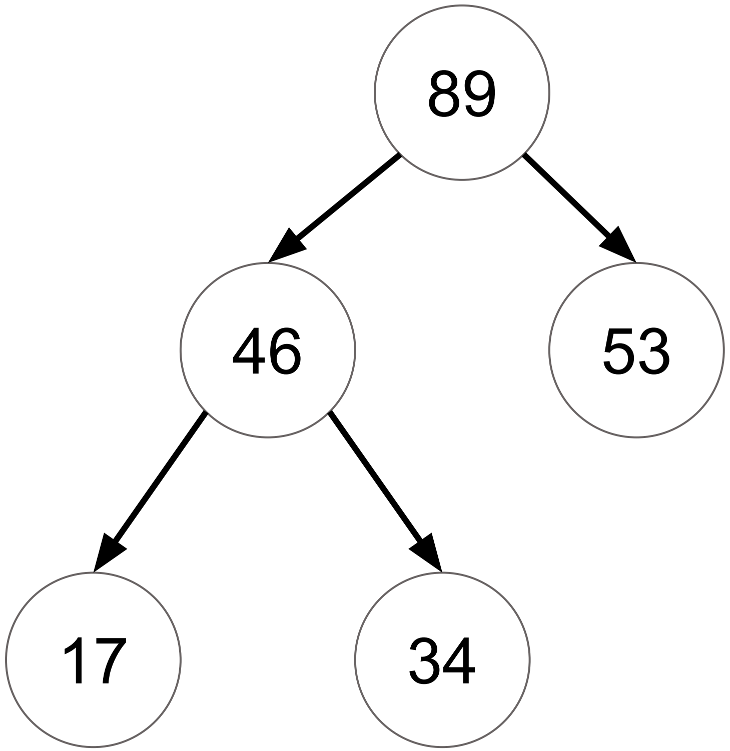
Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

89 is the largest element (the top of the heap), so we move it to the end.

This is done by swapping 89 with the last element, 34.



Array Representation

89	46	53	17	34

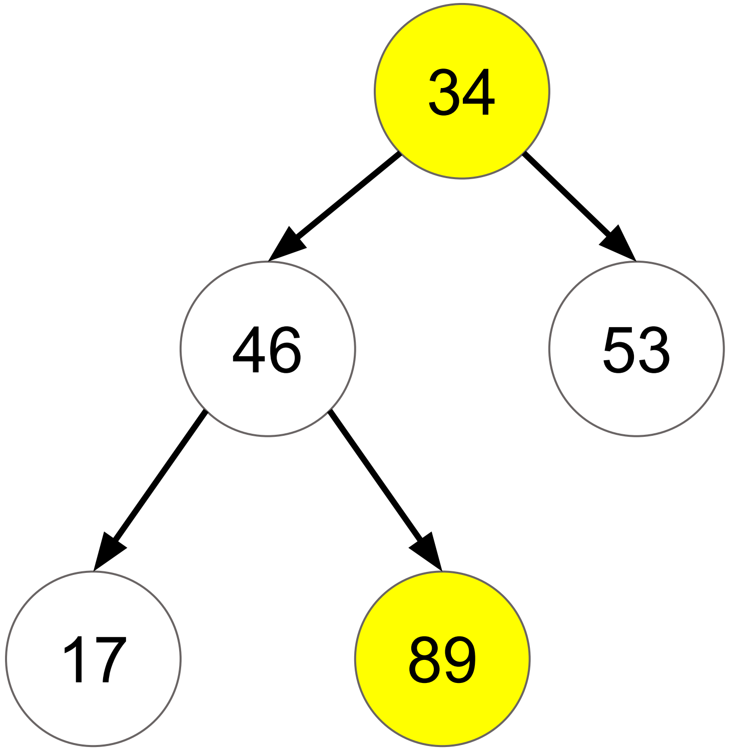
Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

89 is the largest element (the top of the heap), so we move it to the end.

This is done by swapping 89 with the last element, 34.



Array Representation

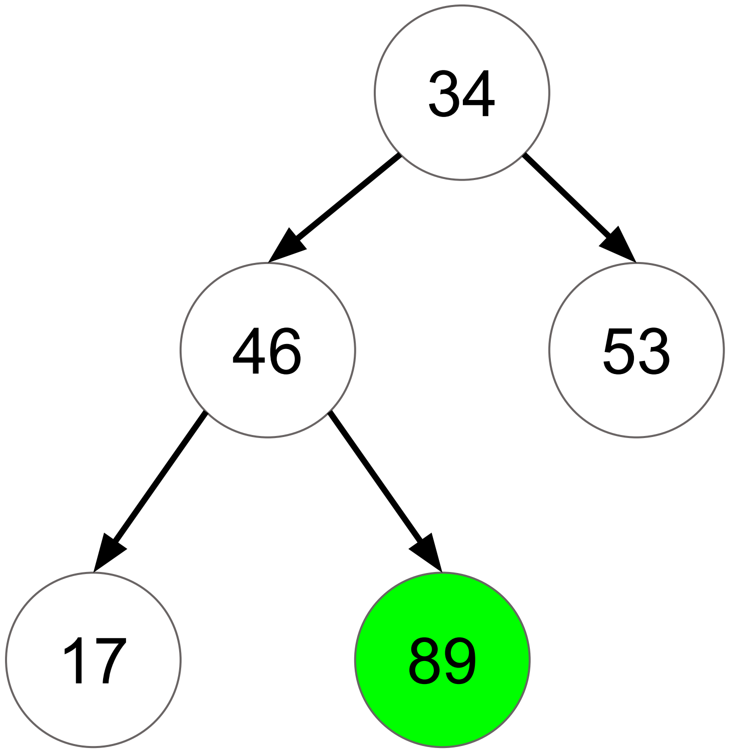
34	46	53	17	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Now we want to find the second largest element to place before 89. Since 89 is the largest, we can find the second largest by building a max-heap for all the *other* elements!



Note that 89 is in the correct position here... we don't want to move it, so we'll just ignore the fact that it exists while we mess with the other elements.

Array Representation

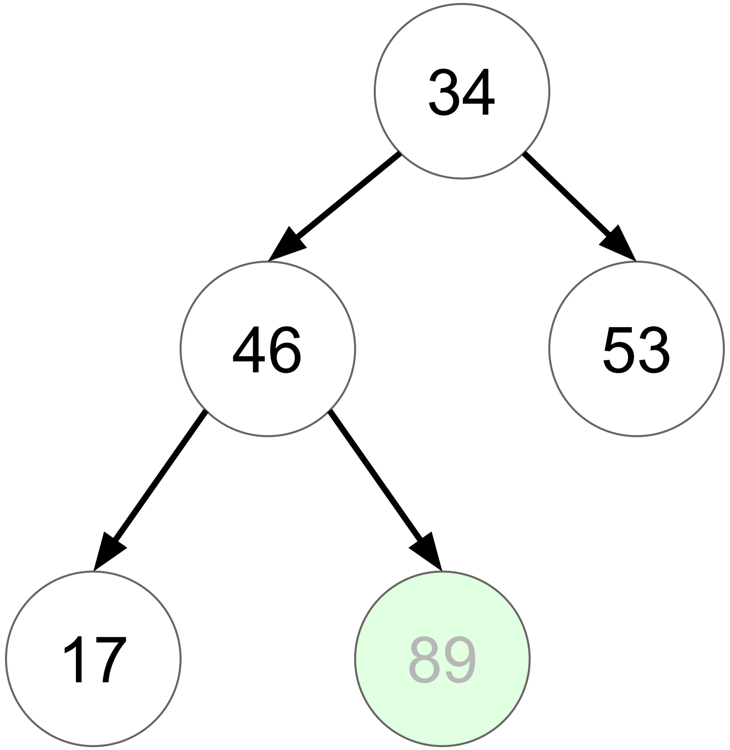
34	46	53	17	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Fix the position of 34 so that the heap is a valid max-heap.



Array Representation

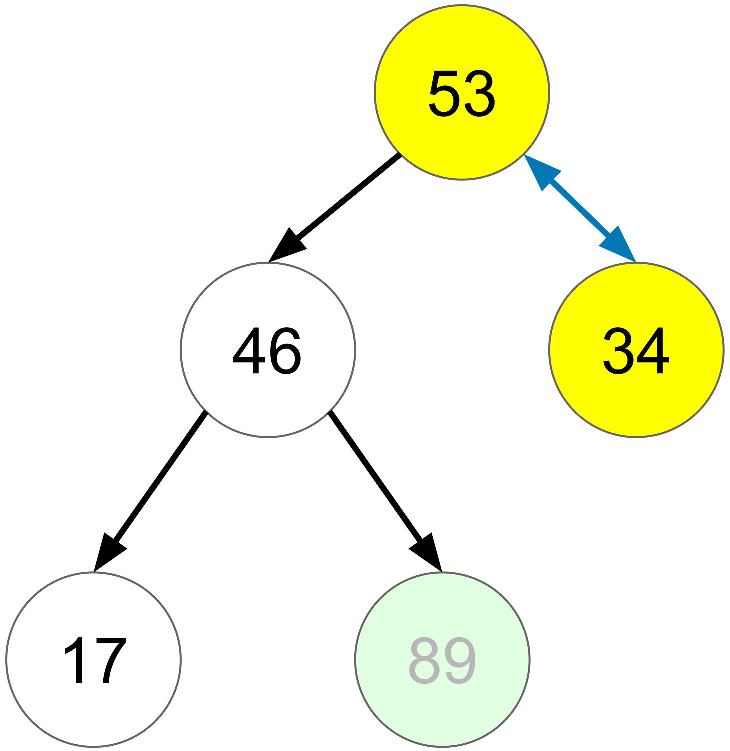
34	46	53	17	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Fix the position of 34 so that the heap is a valid max-heap.



Array Representation

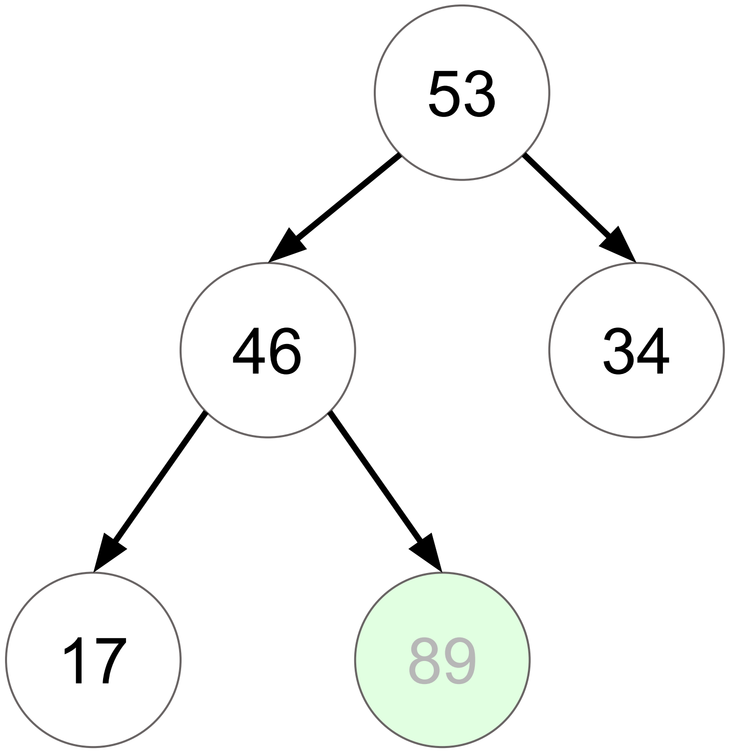
53	46	34	17	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

The heap is now valid, so 53 is the next largest element. Let’s move it to the end.



Array Representation

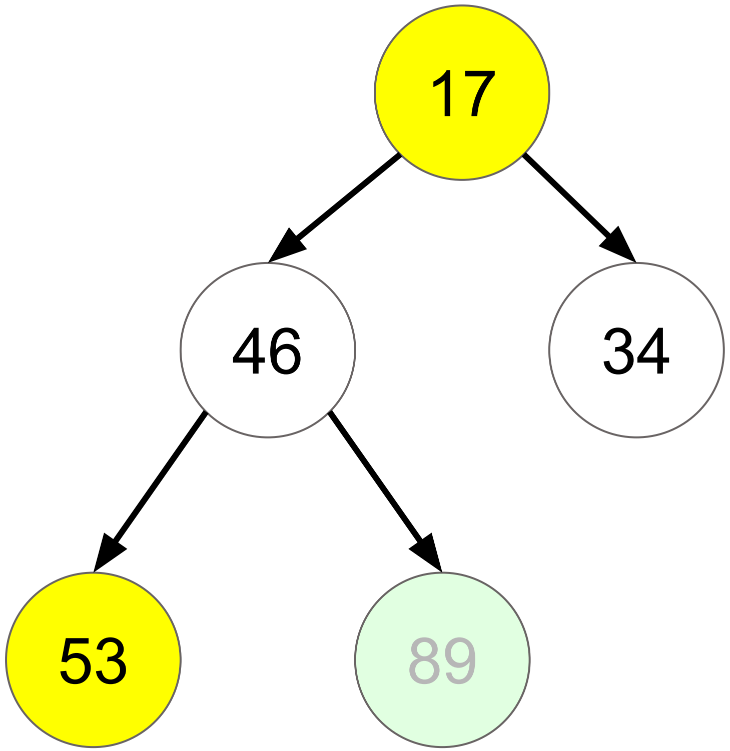
53	46	34	17	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

The heap is now valid, so 53 is the next largest element. Let's move it to the end.



Array Representation

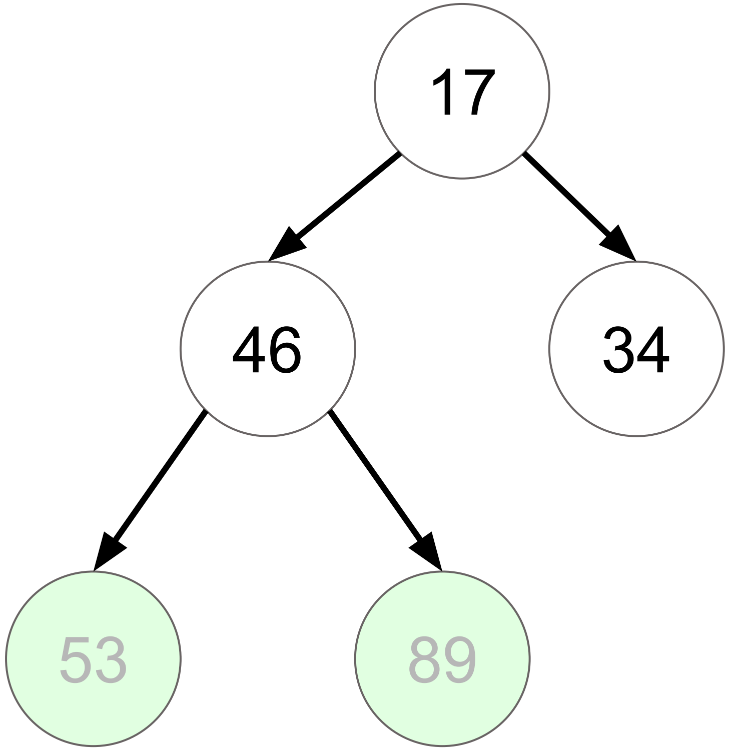
17	46	34	53	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Repeat until the array is completely sorted!



Array Representation

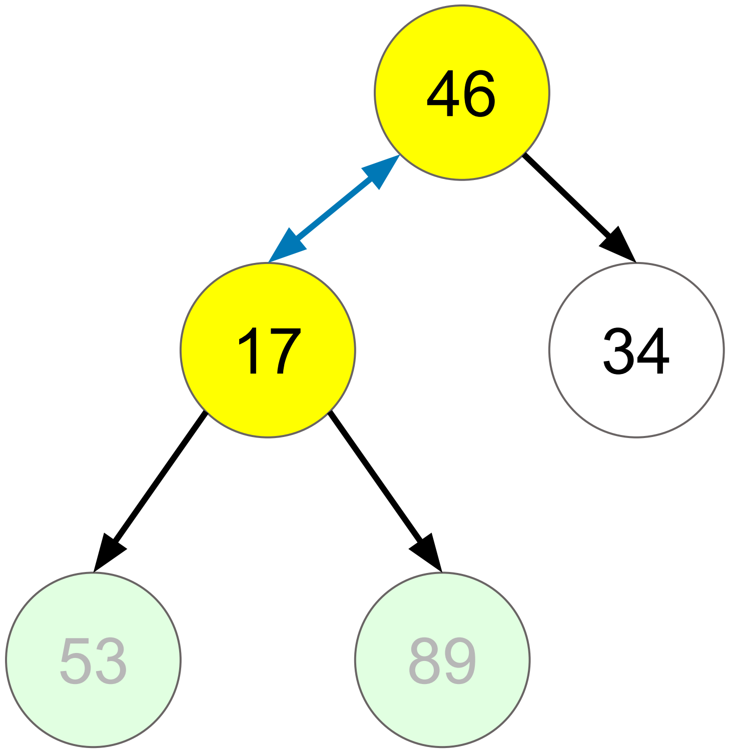
17	46	34	53	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Repeat until the array is completely sorted!



Array Representation

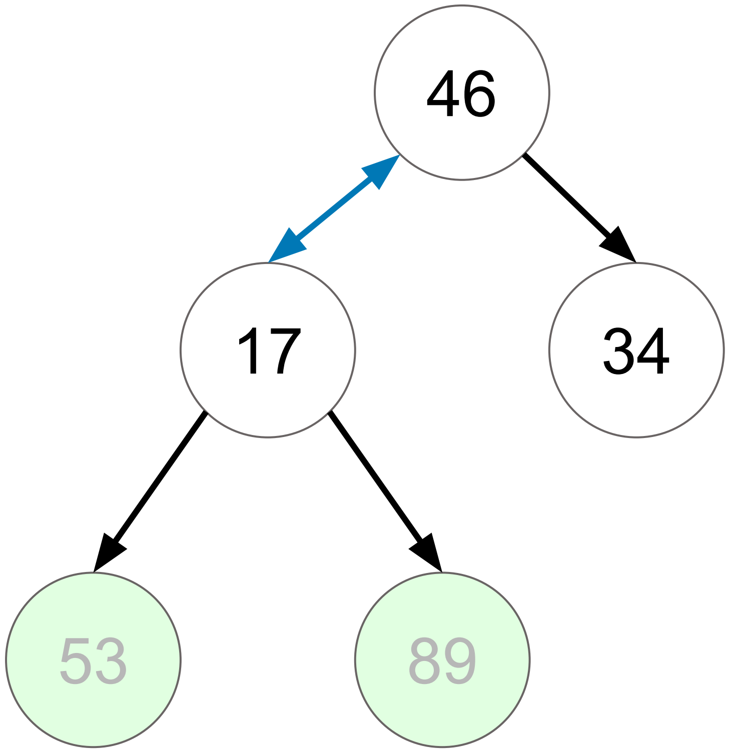
46	17	34	53	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Repeat until the array is completely sorted! (46 is the largest element here)



Array Representation

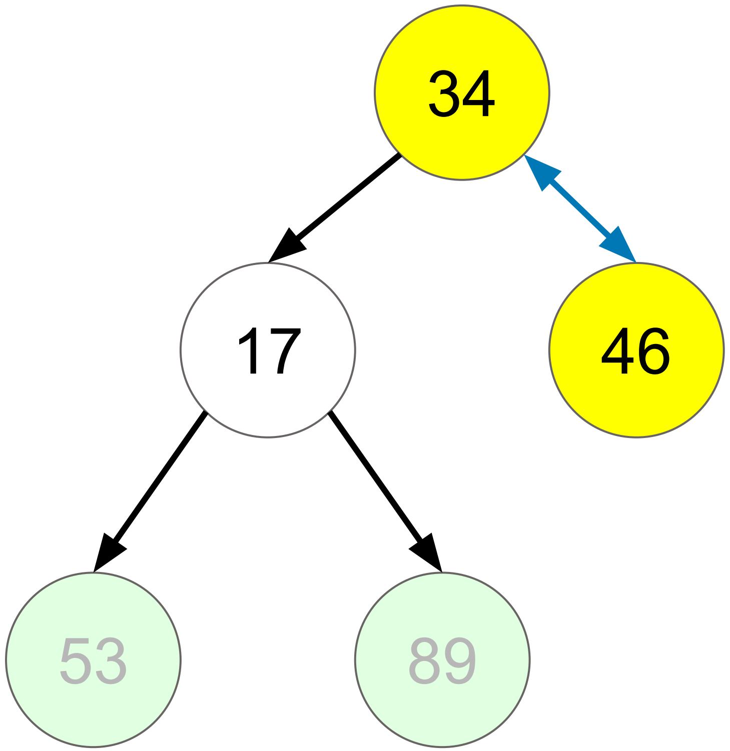
46	17	34	53	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Repeat until the array is completely sorted!



Array Representation

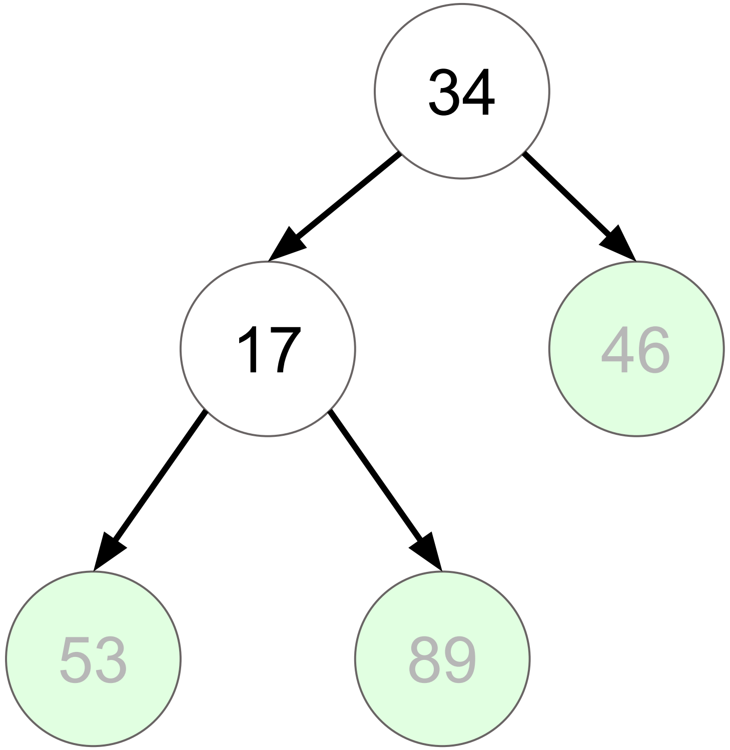
34	17	46	53	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Repeat until the array is completely sorted!



Array Representation

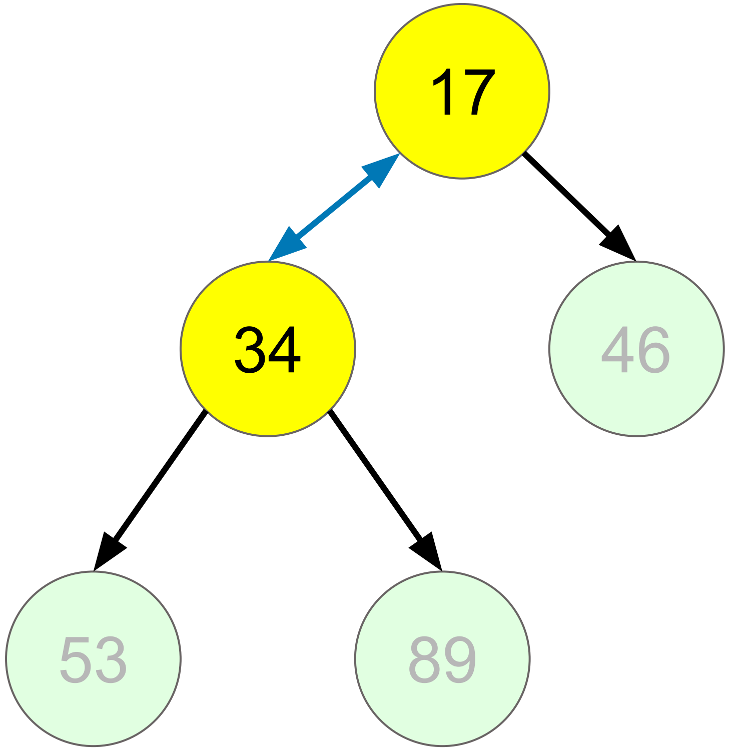
34	17	46	53	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Repeat until the array is completely sorted!



Array Representation

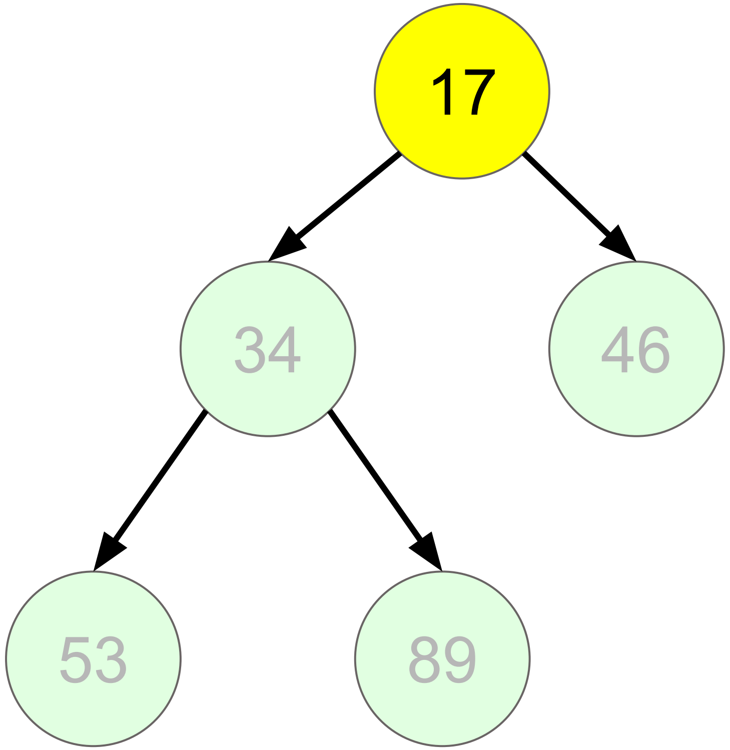
17	34	46	53	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Repeat until the array is completely sorted!



Array Representation

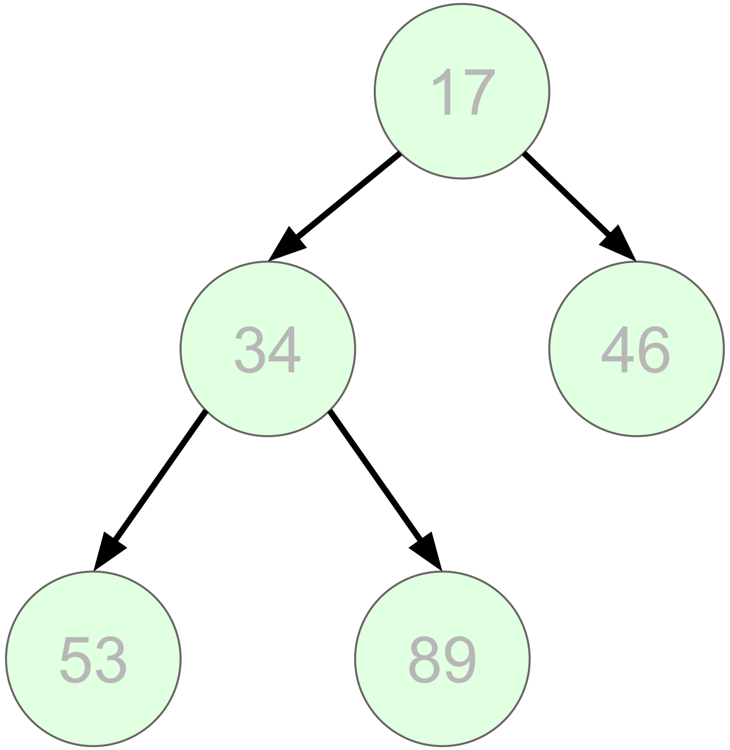
17	34	46	53	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Repeat until the array is completely sorted!



Array Representation

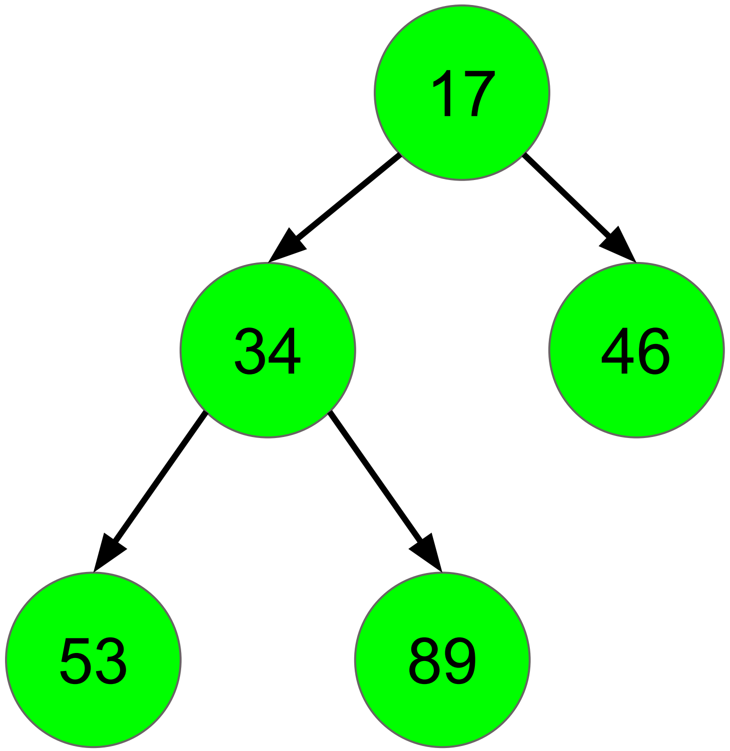
17	34	46	53	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Repeat until the array is completely sorted!



Array Representation

17	34	46	53	89

Sets and Union Find

Sets

- A collection of objects
  - With a set, you can ONLY check one thing:
    - is an object contained in a set?
- All of the following are the "same set":
  - {1, 2, 3}
  - {1, 1, 1, 1, 2, 3}
  - {3, 2, 1}
  - {3, 2, 2, 1, 2, 3, 2, 3, 1, 3, 2, 1}
- The empty set is sometimes useful
  - $\varnothing$: the empty set containing nothing

Set Operations

- Union ($A \cup B$)
  - Set of all objects that are members of A or B.
- Intersection ($A \cap B$)
  - Set of all objects that are members of A and B.
- Set Difference ($A - B$)
  - Set of all objects that are members of A but are not members of B.

Set Operations

- Let A = {1, 2, 3} and B = {1, 3, 5}
  - What is $A \cup B$?
  - What is $A \cap B$?
  - What is $A - B$?

Set Operations

- Let A = {1, 2, 3} and B = {1, 3, 5}
  - What is $A \cup B$?

    {1, 2, 3, 5}

  - What is $A \cap B$?

    {1, 3}

  - What is A – B?

    {2}

Set Operations

- No need to insert or erase elements often?
  - Sorted vector! Binary search to check membership

- Need to insert or erase elements often?
  - We will learn later (data structures that can perform operations in nearly O(1) time)

Set Operations

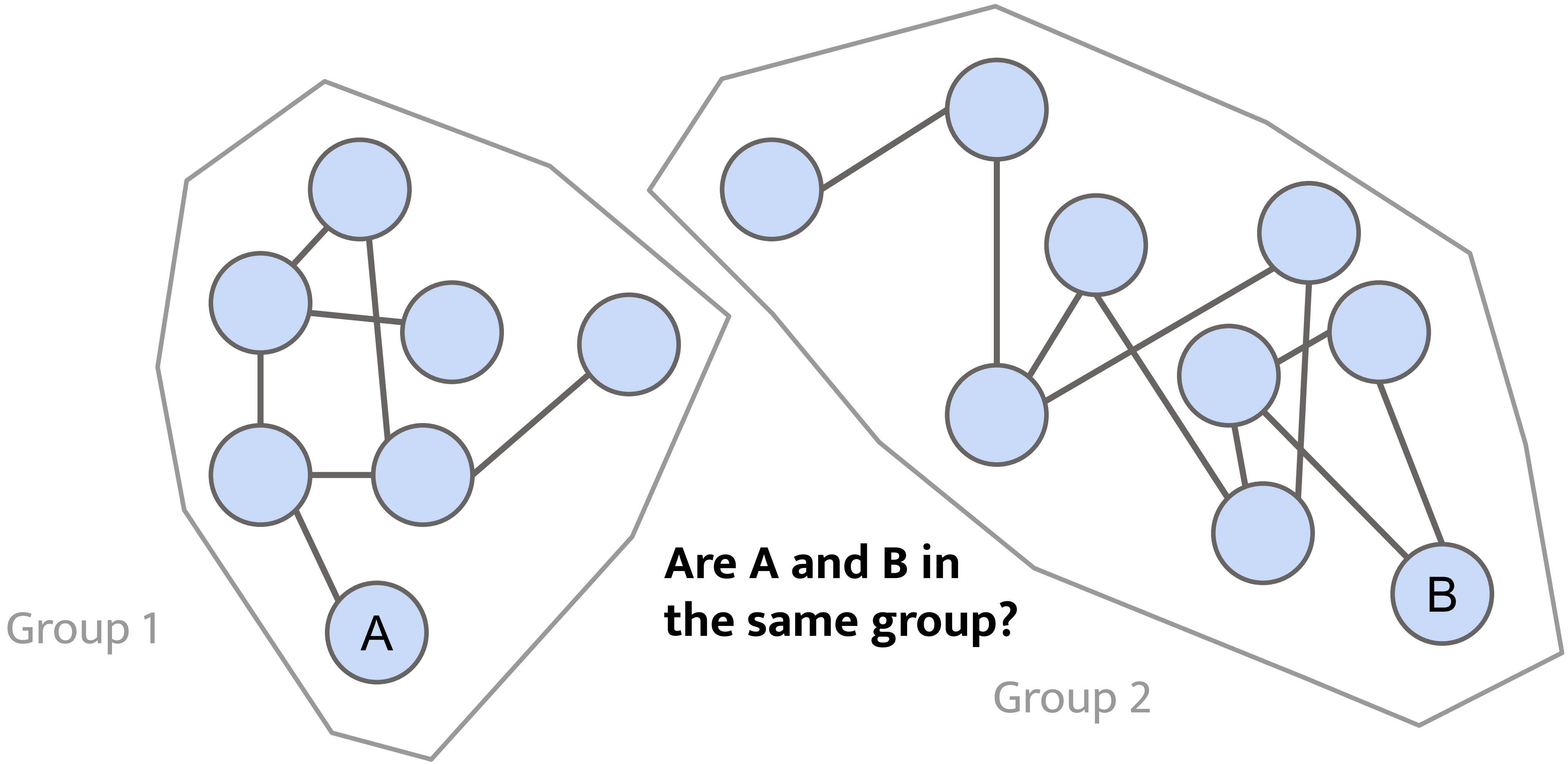
- Practice: How would you implement the following operations on two sorted input vectors and one output vector?
  - Union (A ∪ B)
  - Intersection (A ∩ B)
  - Set Difference (A – B)

Set Operations

- Union (A ∪ B)
  - Iterate over each vector. Push the lower element to the output vector, and increment its iterator. If the elements are the same, only push one and increment both, to avoid duplication.
- Intersection (A ∩ B)
  - Iterate over each vector. If the elements are the same, push one to the output vector and increment both. Otherwise, increment the iterator of the lower element (in case the next element matches the higher element).
- Set Difference (A – B)
  - Iterate over each vector. If the elements are the same, increment both. If A's element compares lower, push it to the output vector. Increment the iterator of the lower element (in case the next element matches the higher element).

Set Operations

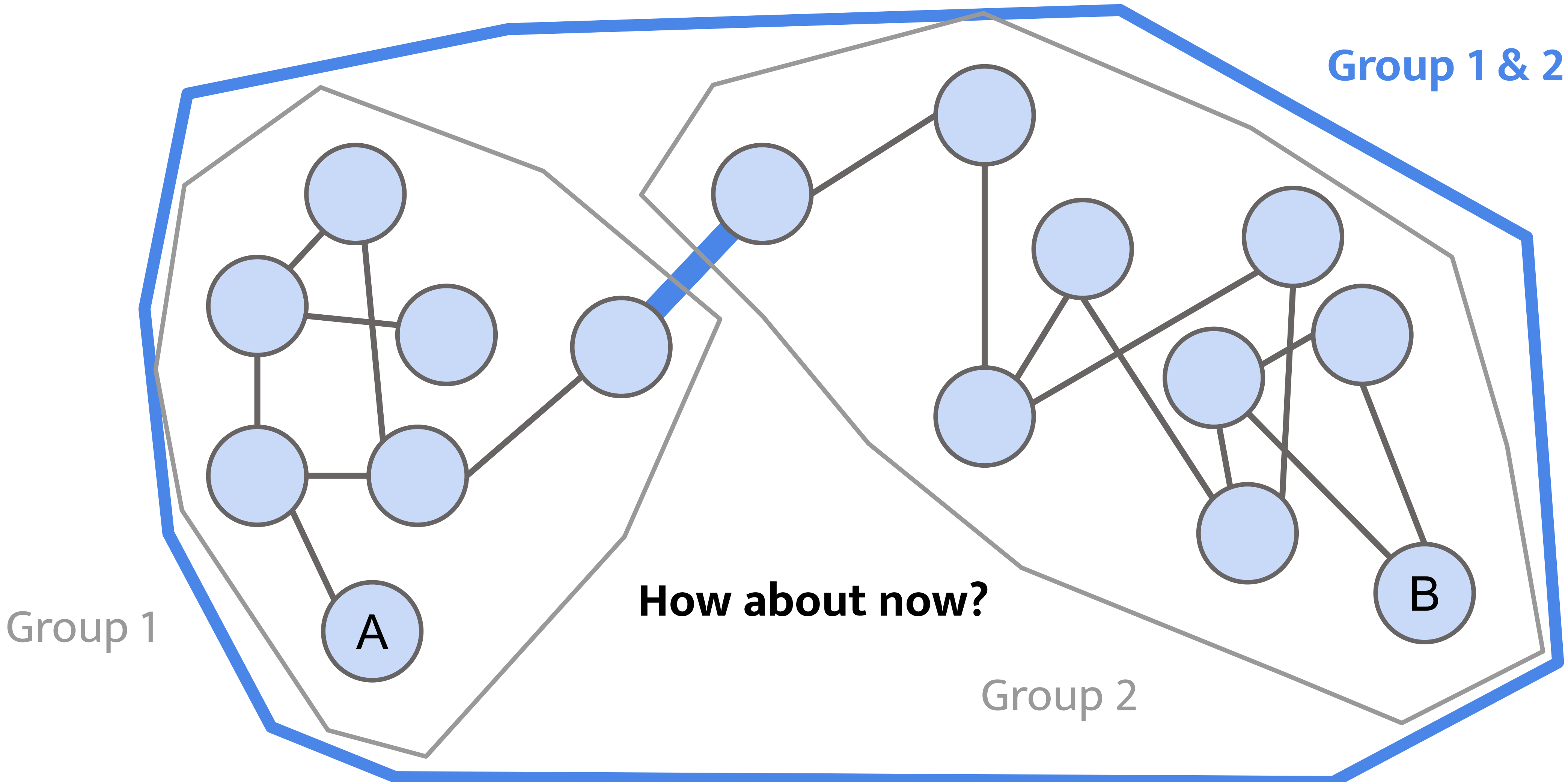
- How do we know if two things are in the same group (set)?



Are A and B in the same group?

Set Operations

- How do we know if two things are in the same group (set)?



Union Find

- Union Find (or Disjoint Set) is a data structure for managing "disjoint sets" - a way to tell whether two items are in the same group.
- Each item has a "representative" that helps identify the group they are in

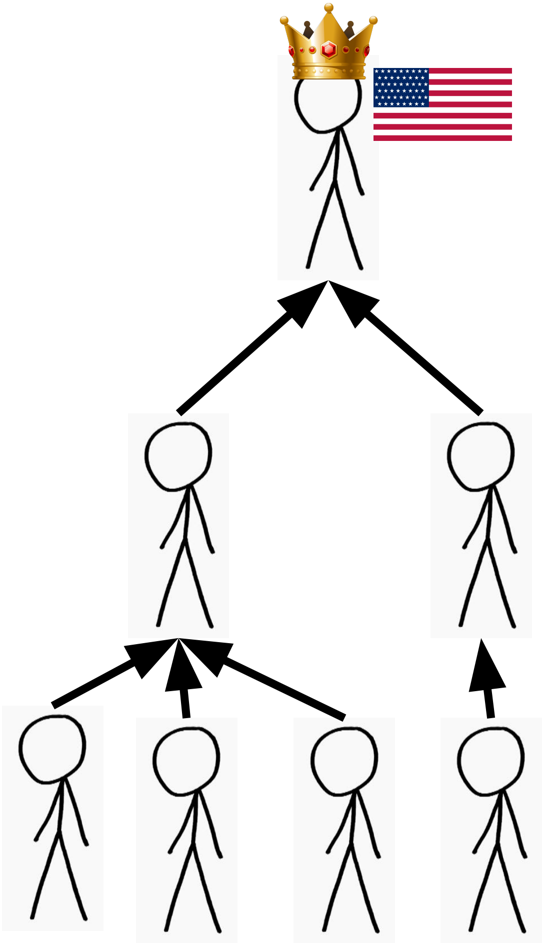
- **union(x, y)** joins x and y so that they become part of the same group
  - if x and y are in different groups, the groups will be combined into a larger group
- **find(x)** returns the "representative" of the group that x belongs to

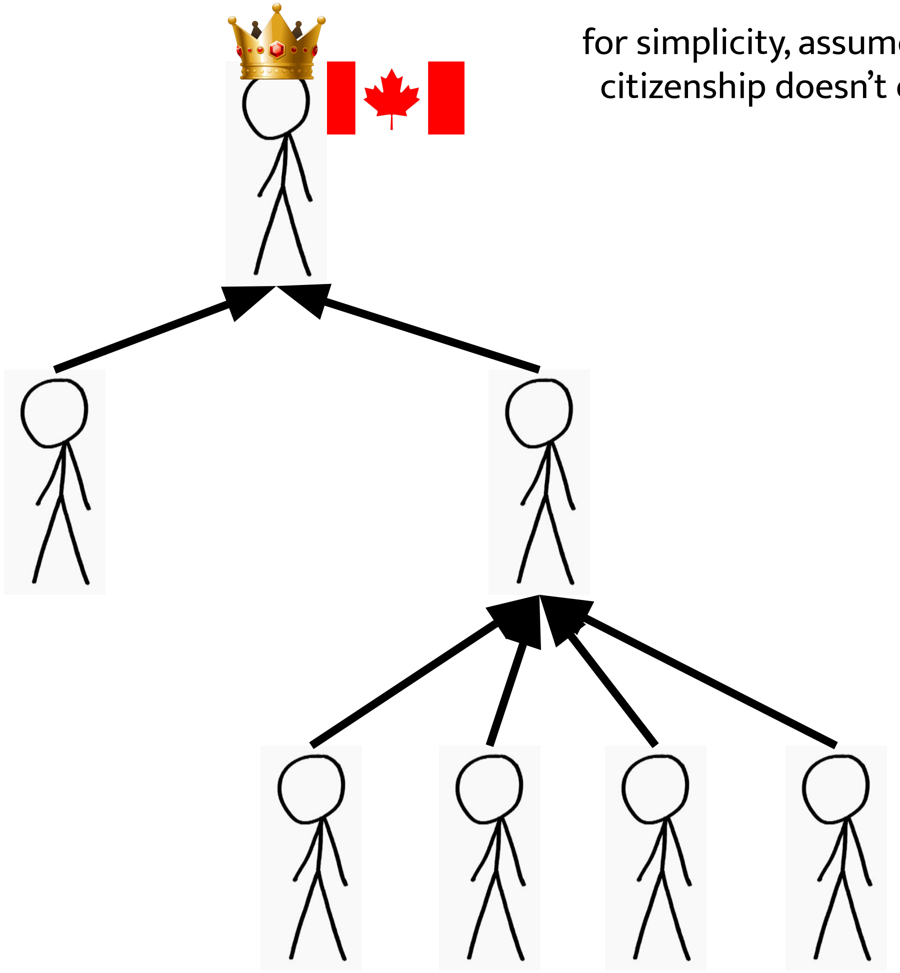
Union(x, y) Complexity Analysis

- Union-find has many applications:
  - counting the number of connected components in a graph
  - seeing if connecting two nodes in a graph will form a cycle (Kruskal's algorithm)

Union Find

- How can we tell if two people are citizens of the same country?

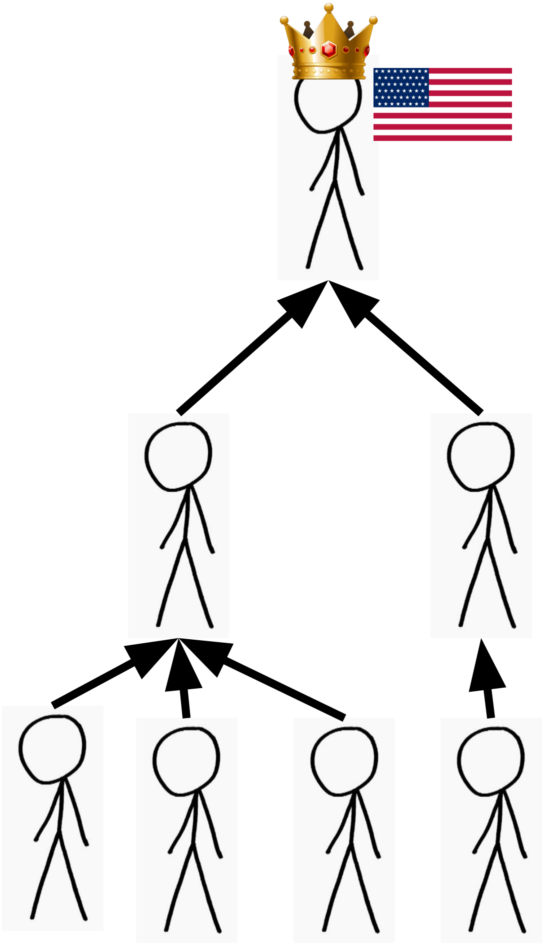




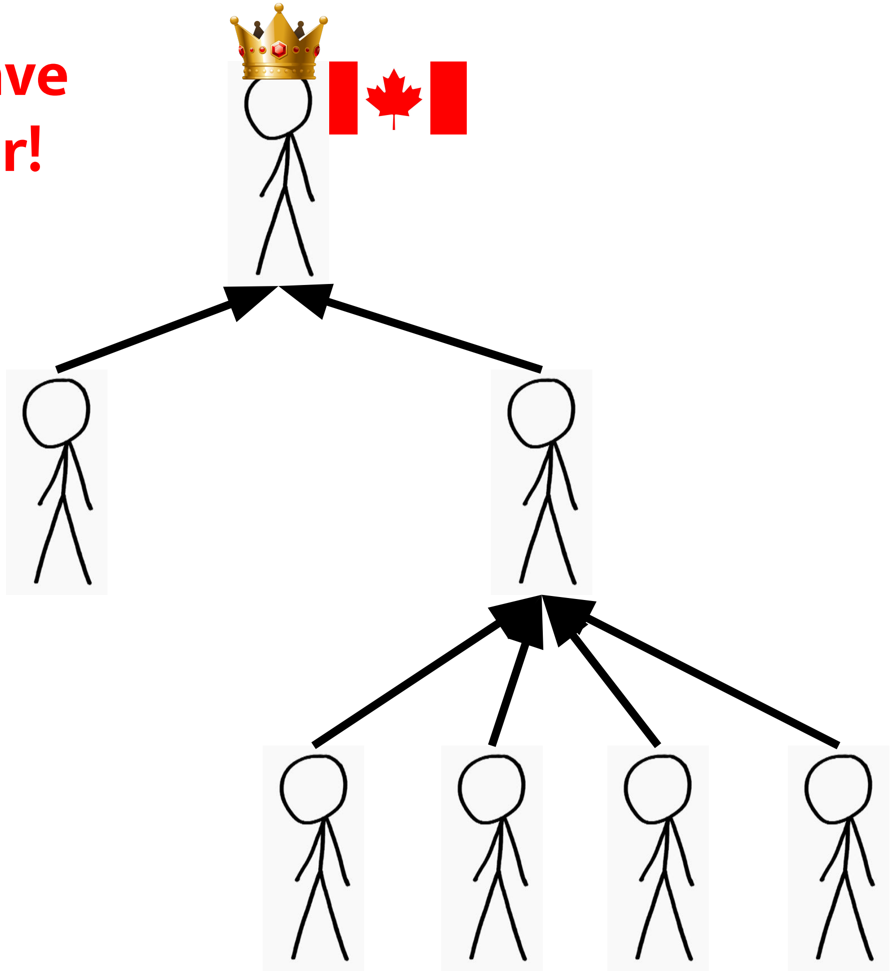
for simplicity, assume that dual citizenship doesn't exist here

Union Find

- How can we tell if two people are citizens of the same country?

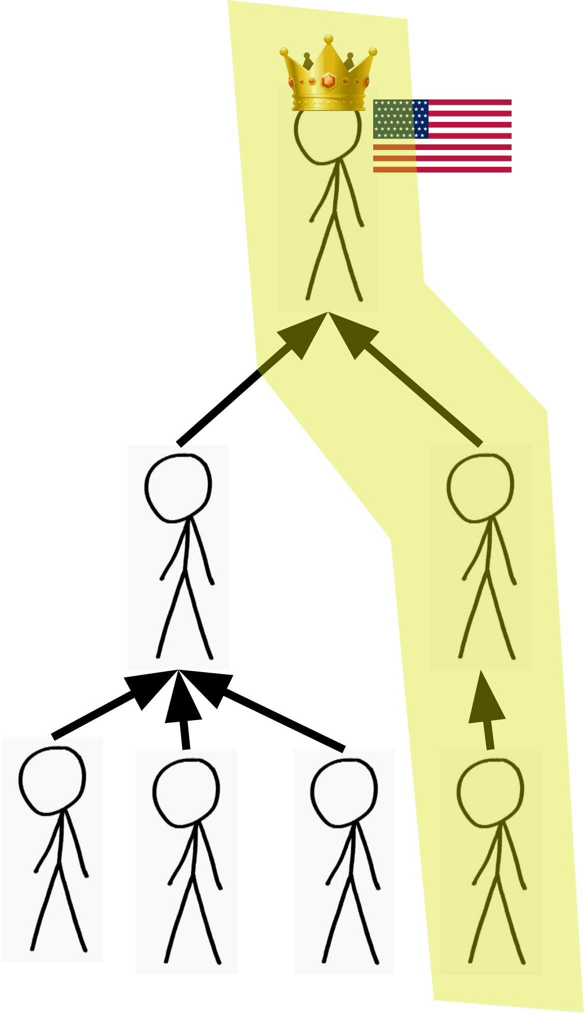


Check if they have the same leader!



Union Find

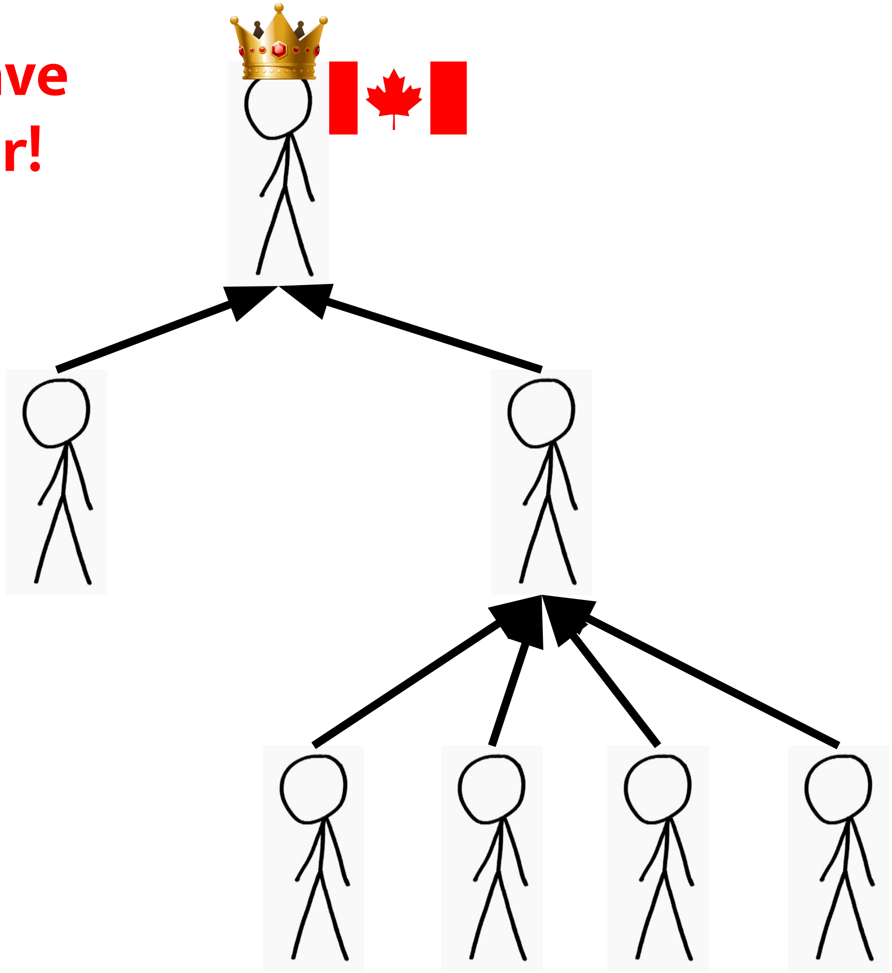
- How can we tell if two people are citizens of the same country?



Check if they have the same leader!

How do we find someone's leader?

Walk up the tree until we find someone whose leader is themself!

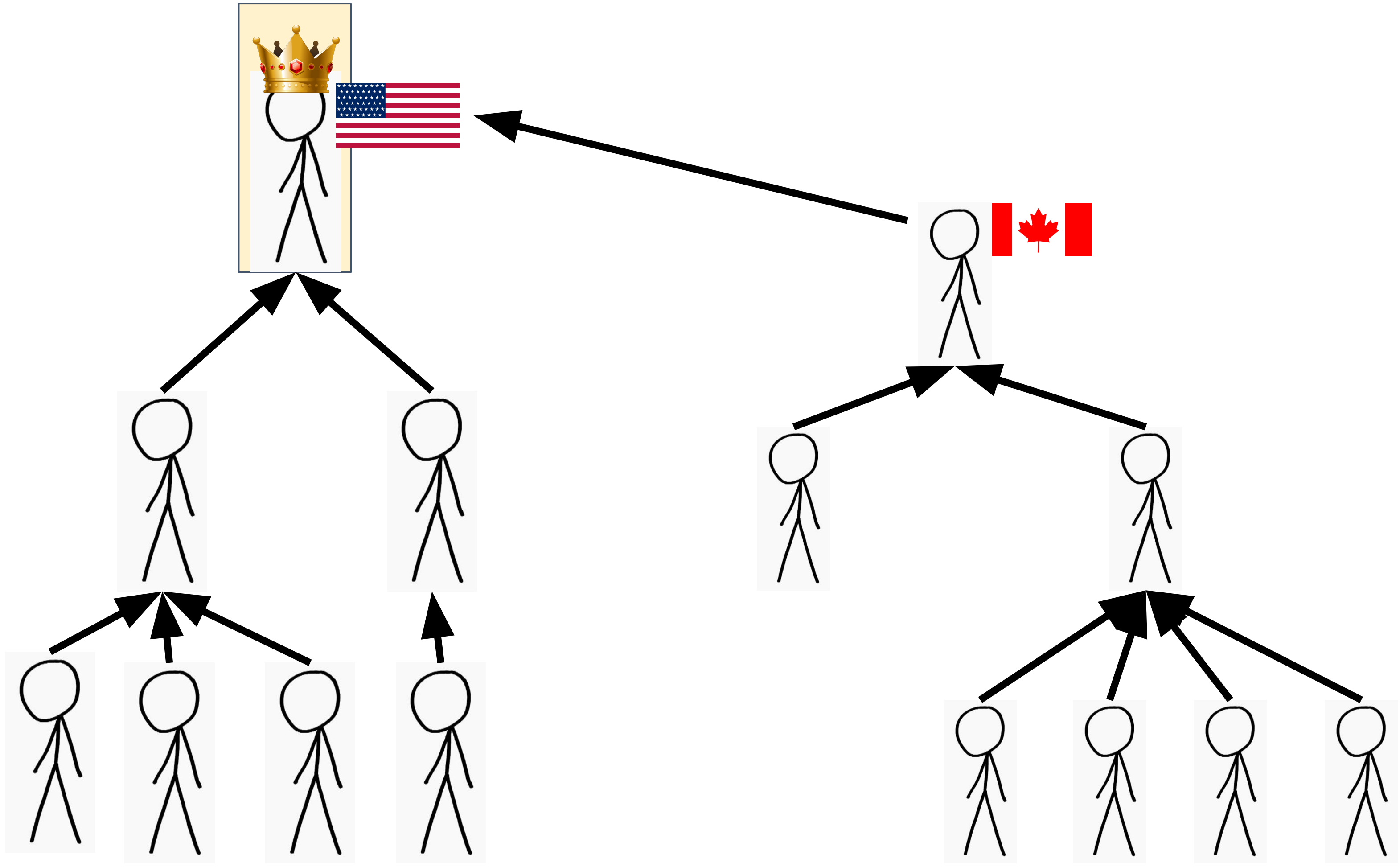


Union Find

- Suppose we have two groups. How do we combine these two groups?
  - After combining, all elements must have the same leader!
  - We want to modify as few nodes as possible (in order to be fast).
- Hypothetical example: suppose the U.S. and Canada merge and all Canadians gain American citizenship.
  - How do we modify the diagram on the previous slide to reflect this?

Union Find

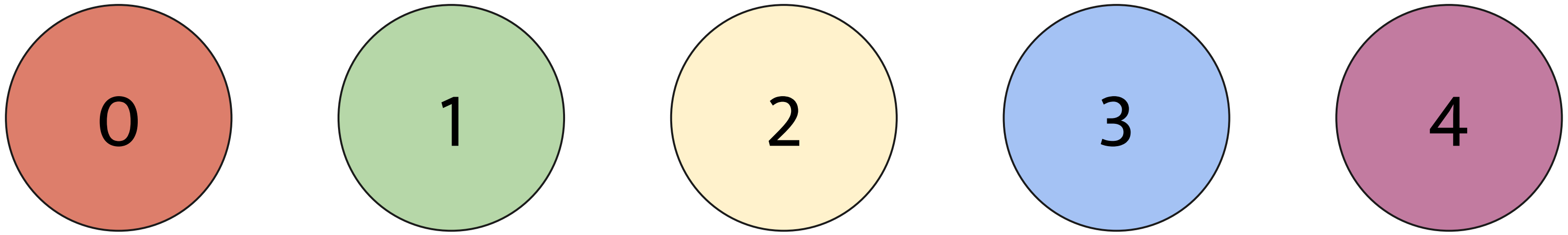
- Solution: the leader of one group is now led by the other group's leader!



Union Find: Array Implementation

- Example: union-find using an array - the value at each index of the array is the "representative" of that vertex.

Index	0	1	2	3	4
Value	0	1	2	3	4

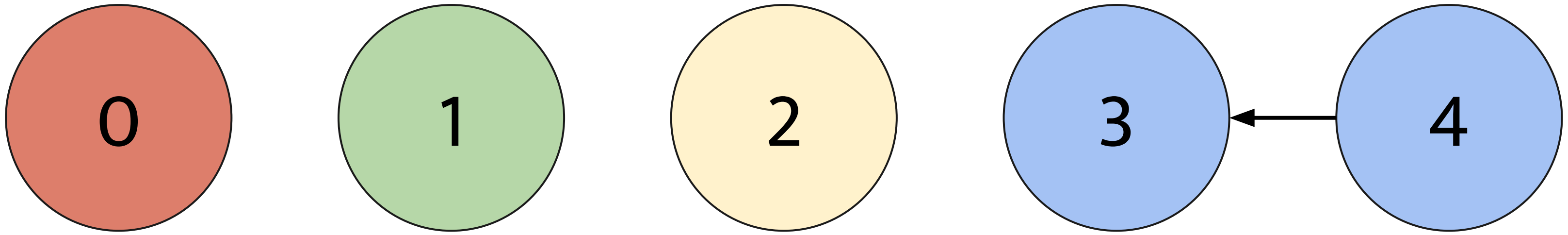


Here, we have five separate groups. Because every representative is different, nothing is connected.

Union Find: Array Implementation

- Example: union-find using an array - the value at each index of the array is the "representative" of that vertex.

Index	0	1	2	3	4
Value	0	1	2	3	3

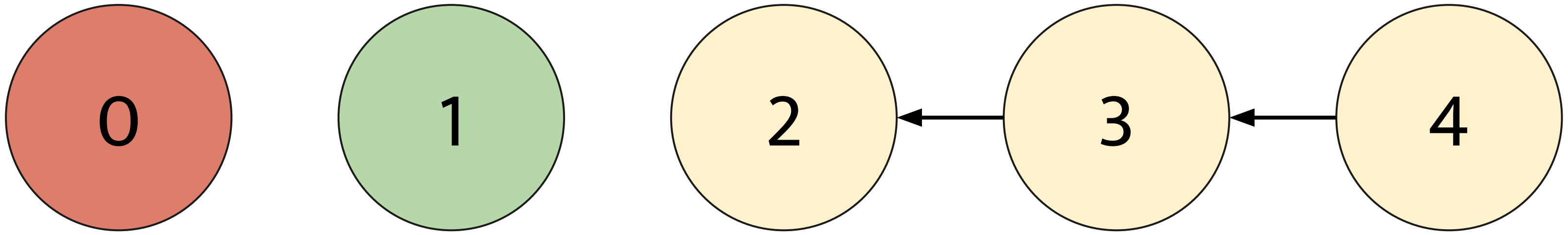


Union(3, 4) - now 4's representative becomes 3.

Union Find: Array Implementation

- Example: union-find using an array - the value at each index of the array is the "representative" of that vertex.

Index	0	1	2	3	4
Value	0	1	2	2	3

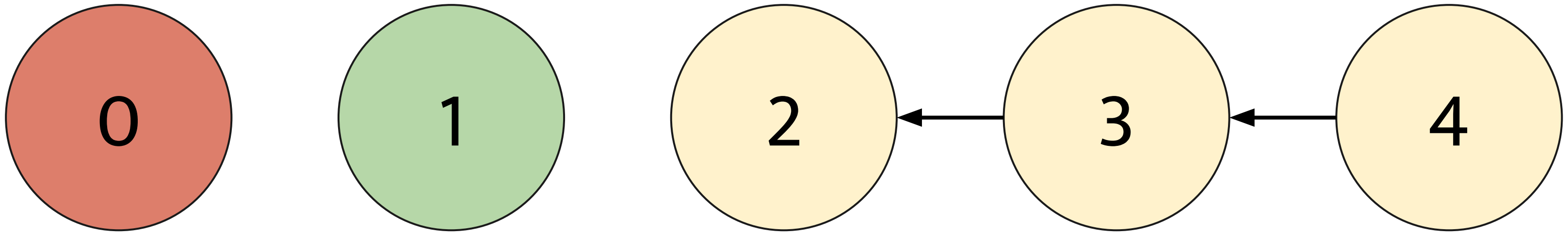


Union(2, 3) - 3's representative is now 2. 2 is now the ultimate rep for the group (notice how 2's rep is itself, but 3 and 4's are not themselves).

Union Find: Array Implementation

- Example: union-find using an array - the value at each index of the array is the “representative” of that vertex.

Index	0	1	2	3	4
Value	0	1	2	2	3



Find(4) without path compression - 4 looks at its rep: 3. 3 looks at its rep: 2. 2 looks at its rep: **itself** (the ultimate rep). Thus, we return 2.

Find(x) Without Path Compression

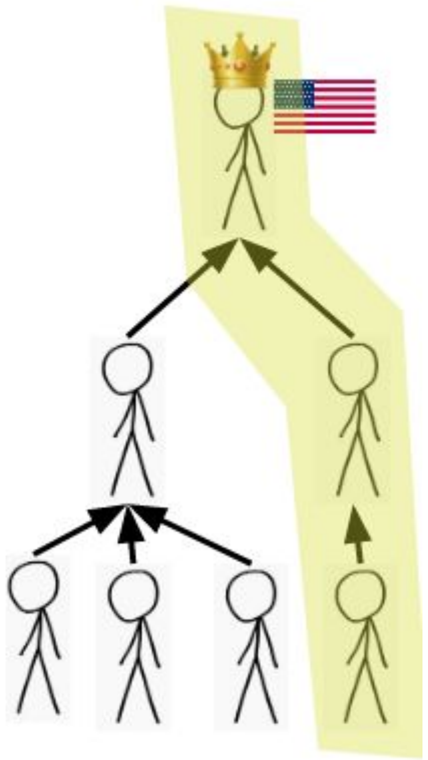
- Assuming that `reps` is the name of our vector, the following function implements `find(x)` **without** path compression.

```
size_t find(size_t x) {
    while (reps[x] != x) {
        x = reps[x];
    }
    return x;
}
```

Union Find: Path Compression

- Problem: suppose we call find on an element multiple times.
  - We have to move up the tree multiple times!
  - Fix: every time we call find on an element, we **move the element closer to its representative**

This process is known as **path compression**!



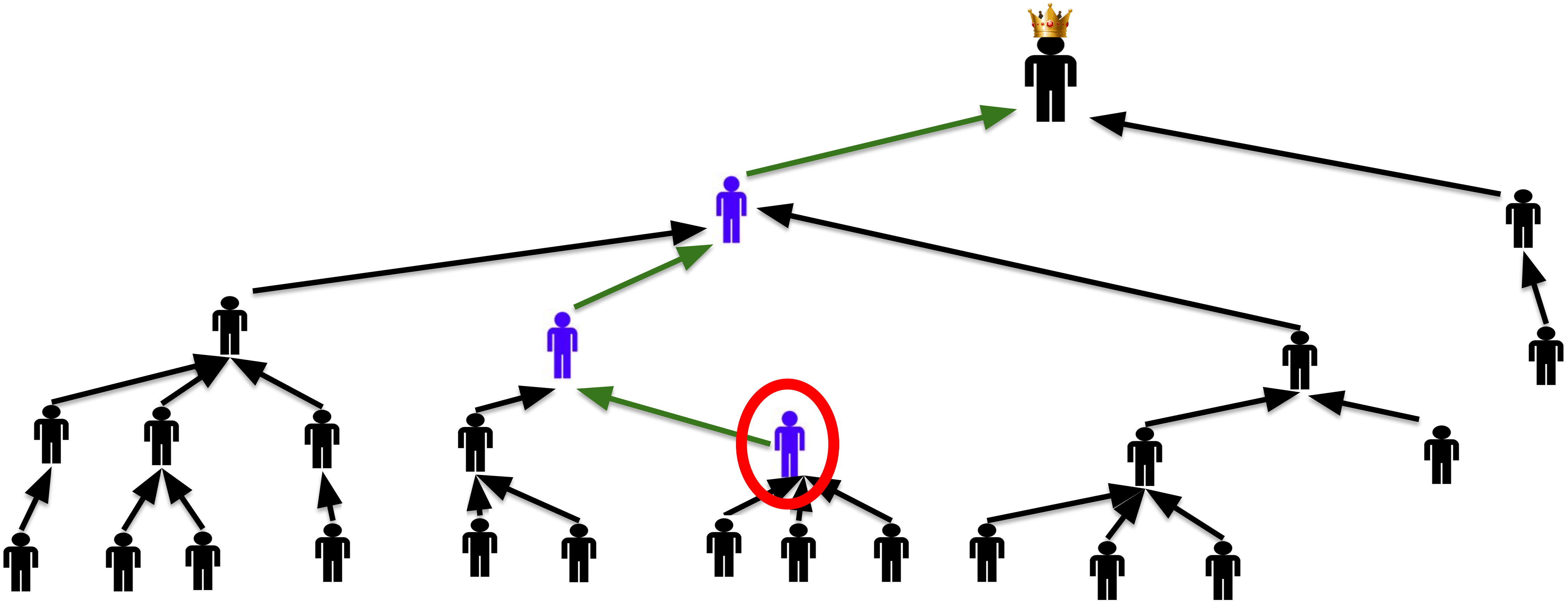
Union Find: Path Compression

- Every time we call find on an element, we **move the element closer to its representative** so we can reduce the work we have to do if find is called again on that element (e.g. we have to move up fewer levels).

```
find(x):
   if reps[x] == x: return x
   reps[x] = find(reps[x])
   return reps[x]
```

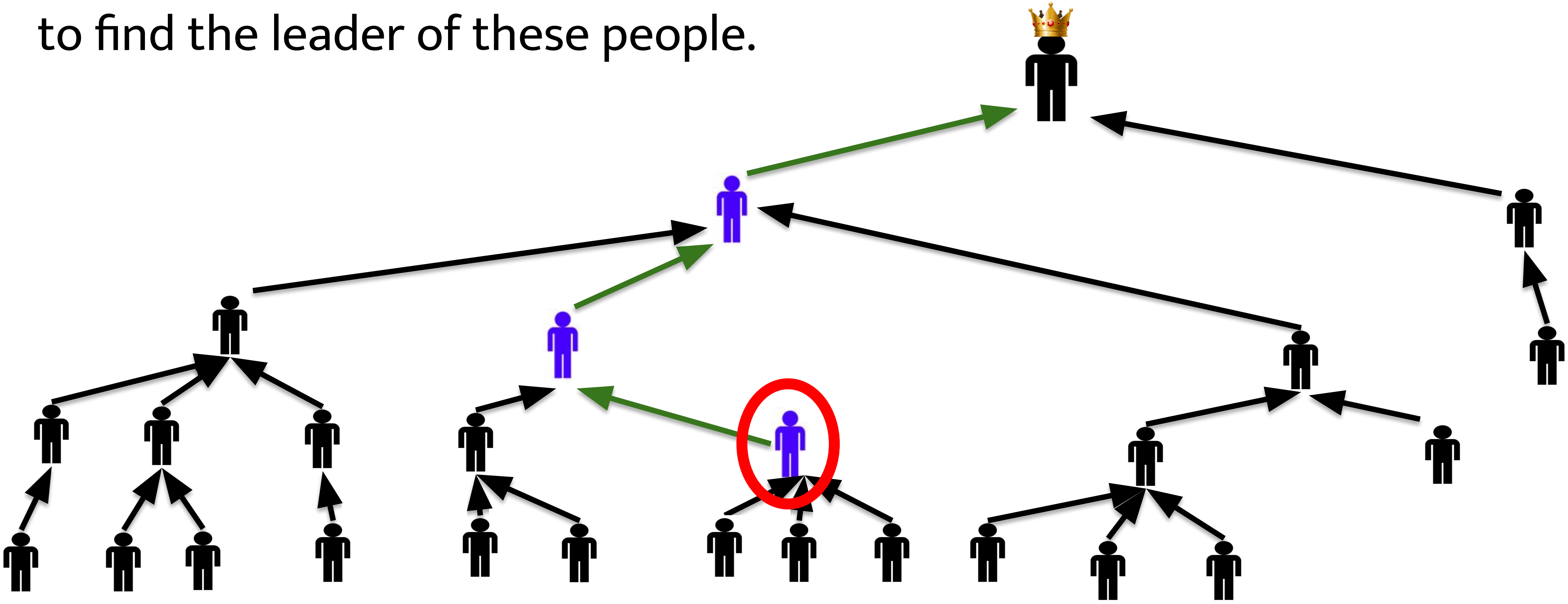
Union Find: Path Compression

- Every time we call find on the circled person, we must move up the tree to find its ultimate representative.



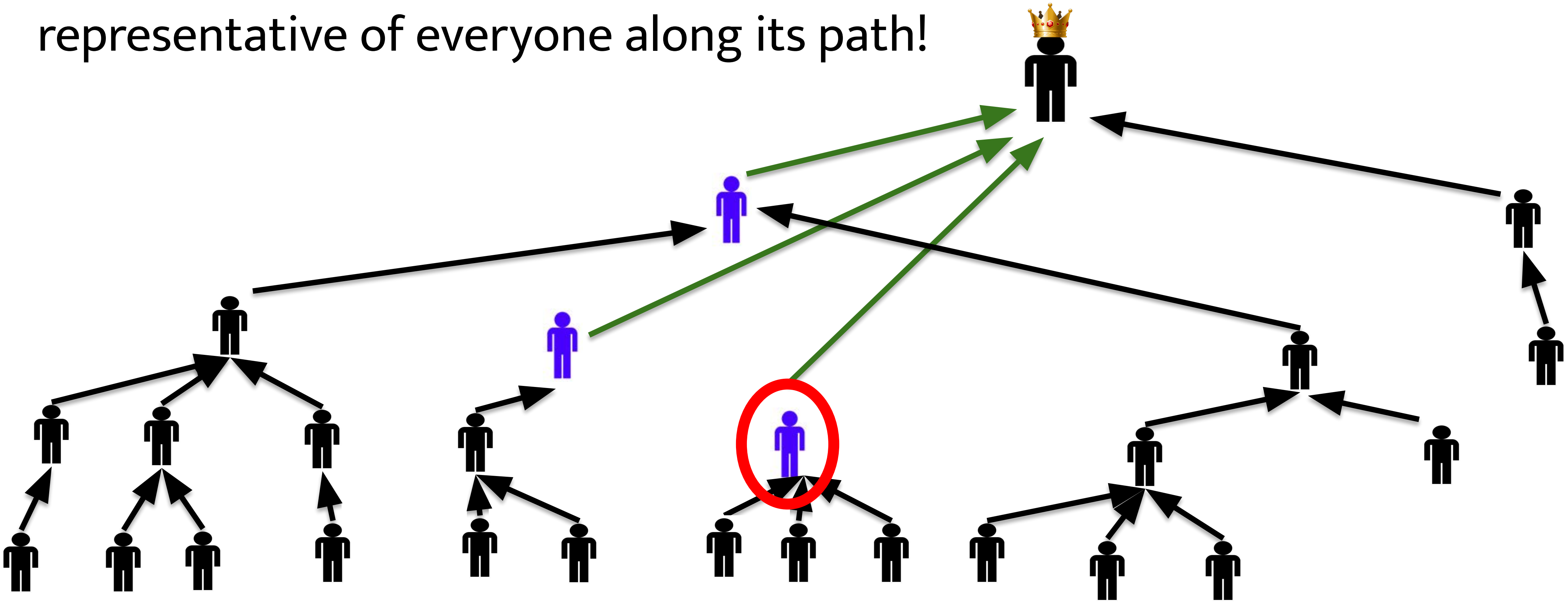
Union Find: Path Compression

- However, after calling find once, we know that all the purple people have the same leader! We don't want to repeat this work over again if we want to find the leader of these people.



Union Find: Path Compression

- To reduce the number of intermediaries we have to visit the next time find is called on the circled person, we can make the leader the ultimate representative of everyone along its path!



Find(x) in the Worst Case

- What is the worst-case time complexity of `find(x)`, given a union-find container with *n* elements?

Find(x) in the Worst Case

- What is the worst-case time complexity of `find(x)`, given a union-find container with *n* elements?

Index	0	1	2	3	4
Value	0	0	1	2	3

O(n) - imagine `find(4)` on the above array of reps:

$4 \longrightarrow 3$

$3 \longrightarrow 2$

$2 \longrightarrow 1$

$1 \longrightarrow 0$

$0 \longrightarrow 0$ (finally, we return 0)

Find(x) in the Worst Case

- What is the worst-case time complexity of `find(x)`, given a union-find container with *n* elements?

Index	0	1	2	3	4
Value	0	0	1	2	3

O(n) - imagine `find(4)` on the above array of reps:

Problem: every time we call `find` on 4, we have to do an O(n) process.

How to use path compression here?

Find(x) in the Worst Case

- What is the worst-case time complexity of `find(x)`, given a union-find container with *n* elements?

Index	0	1	2	3	4
Value	0	0	1	2	3

find(4); // O(n), but updates all parts of path to point to the final rep, 0.

Index	0	1	2	3	4
Value	0	0	0	0	0

find(4); // future `find` calls on 4 will be O(1) since 4 directly points to 0.
Notice that `find(2)` and `find(3)` would also be faster now! Doing a little extra work in `find` to reassign reps saves a lot of time in future calls.

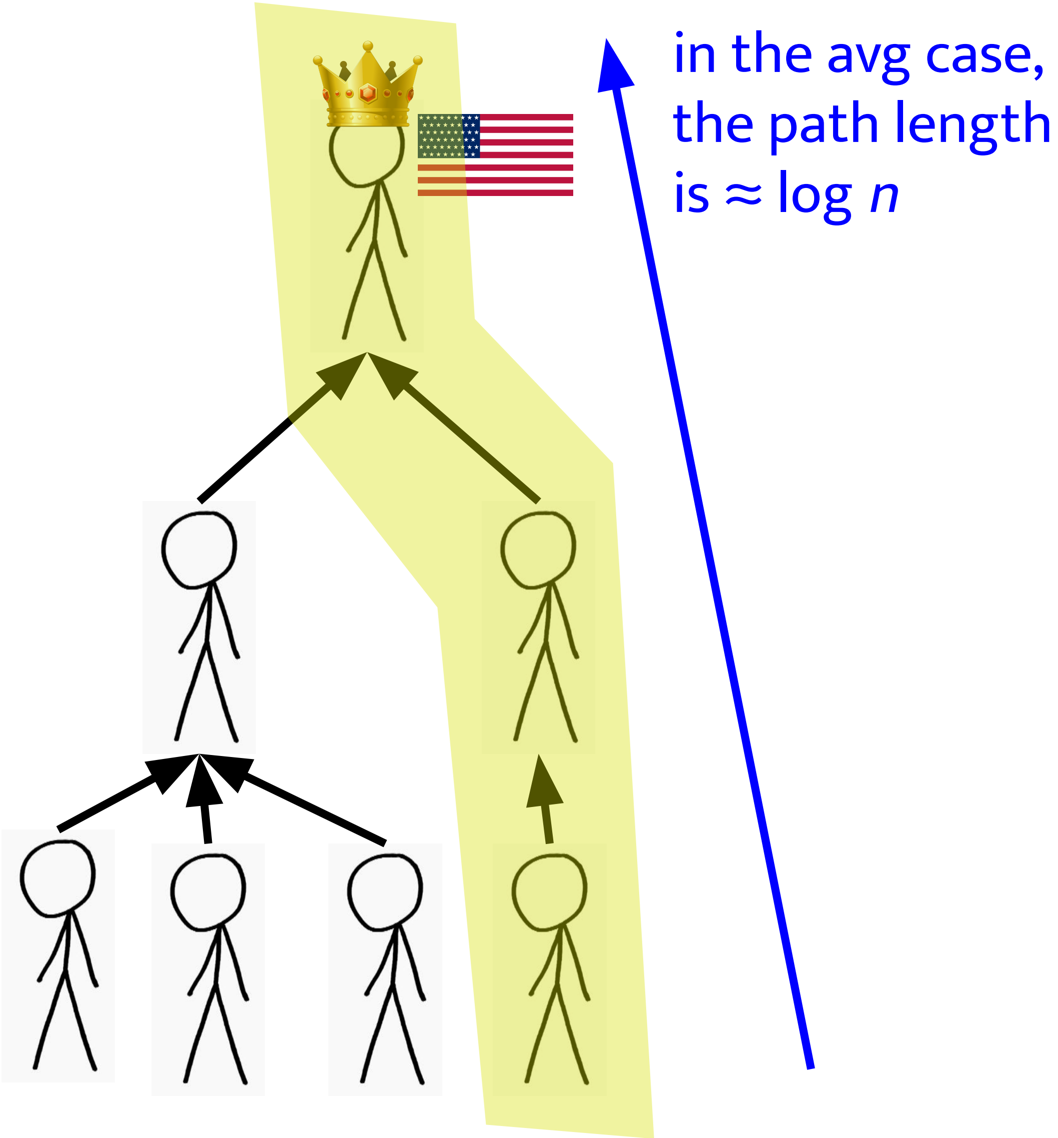
Find(x) With Path Compression

- The following function implements `find(x)` **with** path compression.

```
size_t find(size_t x) {
  size_t pathStart = x;
  // Pass 1 - find the ultimate rep
  while (reps[x] != x) {
    x = reps[x];
  }
  // x is now the ultimate rep
  // Pass 2 - path compression
  while (reps[pathStart] != x) {
    size_t tmp = reps[pathStart];
    reps[pathStart] = x; // Update path to ultimate rep
    pathStart = tmp;
  }
  return x;
}
```

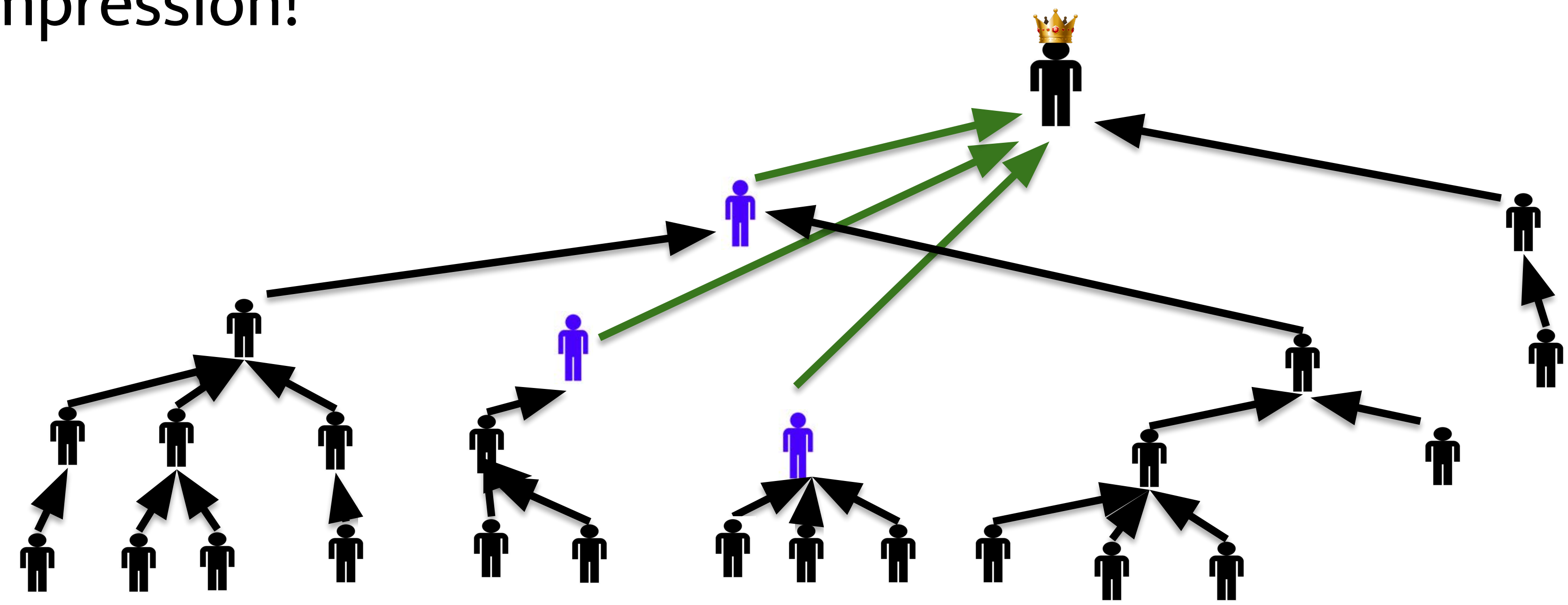
Find(x) in the Average Case

- Without path compression, average time complexity of find is O(log *n*).
  - The average path length for this tree structure from the starting element to the ultimate representative is logarithmic, O(log *n*).



Find(x) Complexity Analysis

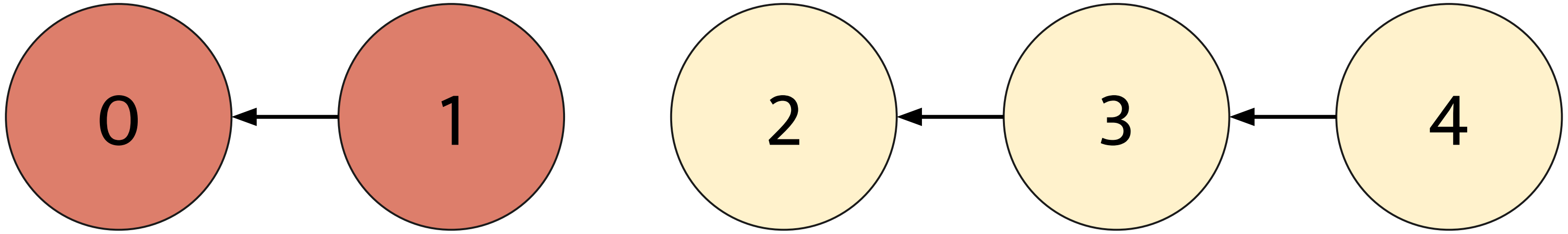
- With path compression, complexity of find becomes amortized **O(α(*n*)), or essentially O(1)** - this is the inverse Ackermann function, which grows very slowly... you don't need to worry about the details.
- This is better both in the average and worst case, so we might as well use path compression!



Union Find: Array Implementation

- Let’s look at how the “union” process takes place using our array-based union-find container.

Index	0	1	2	3	4
Value	0	0	2	2	3

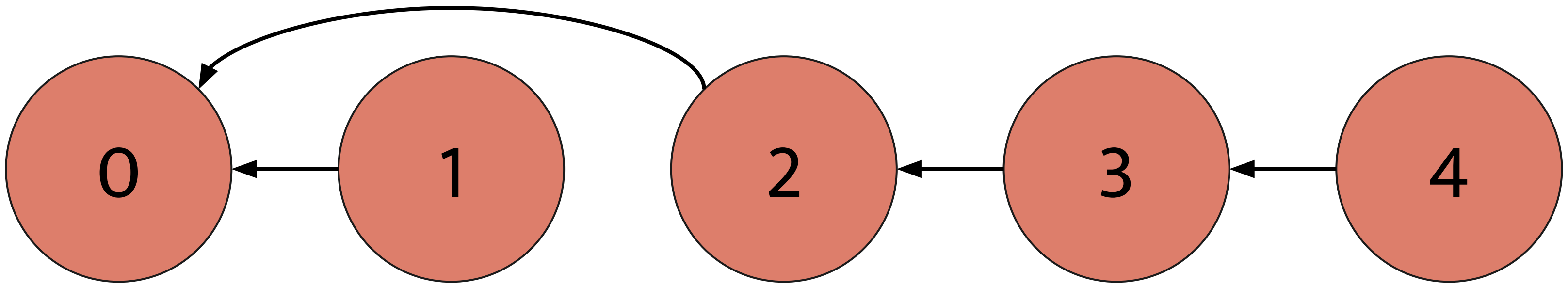


How do we handle **union(1, 4)** in this situation?

Union Find: Array Implementation

- Let’s look at how the “union” process takes place using our array-based union-find container. Now find(x) on any member will return the correct ultimate rep!

Index	0	1	2	3	4
Value	0	0	0	2	3



How do we handle **union(1, 4)** in this situation? We find the ultimate rep of both, and have one rep point to the other rep! **reps[find(4)] = find(1);**

Union(x, y)

- Assuming that `reps` is the name of our vector, the following function implements `union(x, y)`.

```
void set_union(int x, int y) {
    reps[find(y)] = find(x);
}
```

NOTE: you cannot name your function "union" because the word "union" is a reserved word in C++.

Union(x, y) Complexity Analysis

- Depends on the complexity of find because you call find twice for every time you call union! All other work is constant.
- When using **path compression**, union becomes amortized $\mathbf{O(\alpha(\mathit{n})) \approx O(1)}$.

Thus, you should use path compression when implementing union-find!

Union(x, y) Complexity Analysis

- Depends on the complexity of find because you call find twice for every time you call union! All other work is constant.
- When using **path compression**, union becomes amortized $\mathbf{O(\alpha(\mathit{n})) \approx O(1)}$.

Thus, you should use path compression when implementing union-find!

- Note that your decision to use path compression (or not use it) does **NOT** change the functionality of **union(x, y)** and **find(x)**.
- **find(x)** still returns x's ultimate representative - it will just take longer without path compression!

Union Find: Putting It All Together

```
class UnionFind {
private:
  vector<size_t> reps;
public:
  UnionFind(size_t size) {
     reps.reserve(size);
     for (unsigned i = 0; i < size; ++i) {
        // at the beginning, every
        // node represents itself!
        reps.push_back(i);
     }
  }
  size_t find(x);
  void set_union(x, y);
};
```

Handwritten Problem

- Given *n* ropes, find the **minimum possible** cost of connecting ropes, where the cost of connecting two ropes is the sum of their lengths.
  - **Example:** consider four ropes of lengths 10, 5, 8, and 11. The minimum cost to join all four ropes is **68**:
    1. join ropes of length 5 and 8 to get a rope of length 13 (net cost = 13)
    2. join ropes of length 10 and 11 to get a rope of length 21 (net cost= 13 + 21)
    3. join ropes of length 13 and 21 to get a rope of length 34 (net cost = 13 + 21 + 34)
  - Thus, the minimum cost is 13 + 21 + 34 = 68.

```
// calculate minimum cost required to join n ropes
int joinRopes(vector<int>& ropeLengths);
```

Handwritten Problem Review

Lab 4 Written Problem: Connecting Ropes

- Find the minimum cost of connecting ropes:
  - for example, if we had seven ropes of length 1 and one rope of length 100…
    - we would keep on connecting the ropes of length 1 until it becomes one rope
      - this combined rope would have a length of 7
    - we then combine this rope of length 7 with the rope of length 100
  - notice that we connect the smallest ropes first… what is a good data structure for this?

```
               .-----$107----.
               |             |
        .-----$7------.      |
        |             |      |
     .-$4---.      .-$3-.    |
     |      |      |    |    |
   .$2-.  .$2-.  .$2-.  |    |
   |   |  |   |  |   |  |    |
   1   1  1   1  1   1  1  100

Sum: 1+1+1+1+1+1+1+100      = 107
Total Cost: 107+7+4+3+2+2+2 = 127
```

Lab 4 Written Problem: Connecting Ropes

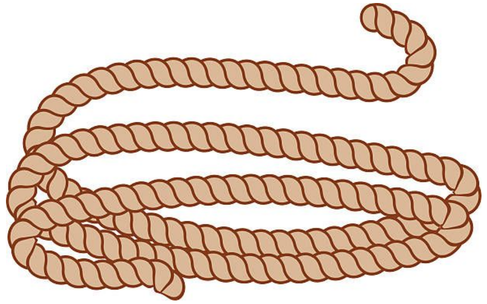
- Find the minimum cost of connecting ropes:
  - minimize the sum of lengths, so connect the shortest ropes ⟶ priority queue!

```
// calculate minimum cost required to join n ropes
int joinRopes(vector<int>& ropeLengths) {
   priority_queue<int, vector<int>, greater<int>> pq(ropes.begin(), ropes.end());
   int cost = 0;
   while (pq.size() > 1) {
      int rope1 = pq.top();
      pq.pop();
      int rope2 = pq.top();
      pq.pop();
      pq.push(rope1 + rope2);
      cost += (rope1 + rope2);
   }
   return cost;
}
```

we want a min PQ, so we use greater<int> here

if you want to efficiently convert data in a container into a heap (PQ), consider using a range constructor

connect the two shortest ropes you have until you only have one rope left, incrementing cost by the combined lengths of the ropes along the way



Lab 4 Written Problem: Connecting Ropes

- Find the minimum cost of connecting ropes:
  - minimize the sum of lengths, so connect the shortest ropes ⟶ priority queue!

```cpp
// calculate minimum cost required to join n ropes
int joinRopes(vector<int>& ropeLengths) {
   priority_queue<int, vector<int>, greater<int>> pq(ropes.begin(), ropes.end());
   int cost = 0;
   while (pq.size() > 1) {
       int rope1 = pq.top();
       pq.pop();
       int rope2 = pq.top();
       pq.pop();
       pq.push(rope1 + rope2);
       cost += (rope1 + rope2);
   }
   return cost;
}
```

Why not just use a sorted array???

[redacted]

Lab 4 Written Problem: Connecting Ropes

- Find the minimum cost of connecting ropes:
  - minimize the sum of lengths, so connect the shortest ropes ⟶ priority queue!

```
// calculate minimum cost required to join n ropes
int joinRopes(vector<int>& ropeLengths) {
   priority_queue<int, vector<int>, greater<int>> pq(ropes.begin(), ropes.end());
   int cost = 0;
   while (pq.size() > 1) {
       int rope1 = pq.top();
       pq.pop();
       int rope2 = pq.top();
       pq.pop();
       pq.push(rope1 + rope2);
       cost += (rope1 + rope2);
   }
   return cost;
}
```

Why not just use a sorted array???

Insertion is O(n) in an array, compared to O(log n) in a priority queue!

Heapify is O(n) while sorting is O(n log n).