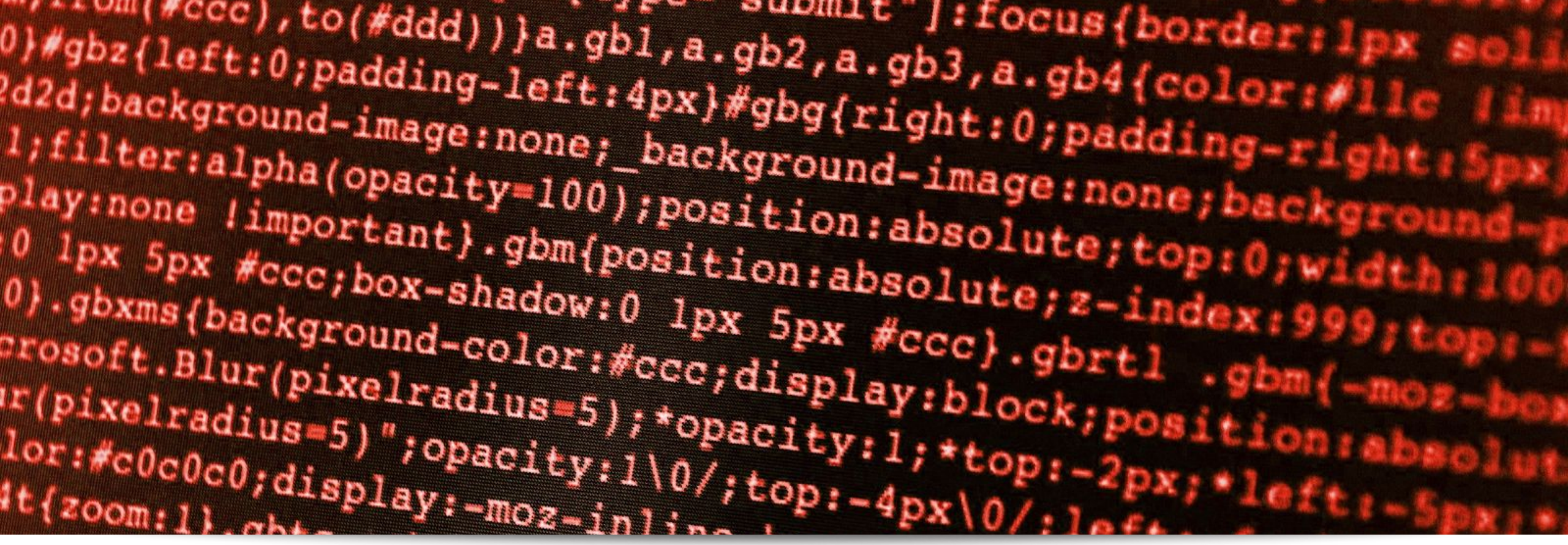


04

September 24-30, 2024

Priority Queues, Heaps, Union-Find

Announcements

- Lab 3 AG + Quiz due 9/30, Lab 4 Handwritten due 9/30, and Lab 4 AG + Quiz due 10/7
- Project 1 final grading soon
  - 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 currently see in the autograder
- Project 2 is out, **due October 10th at 11:59pm**
- Midterm **October 17th** at 7:00-9:00 PM
  - If you need an alternate time, or SSD accommodations, instructions will be in a pinned Ed Announcement - you must complete the alternate exam request form as soon as possible!
  - Direct any administrative questions to eecs281admin@umich.edu

Agenda

- Lab 3 Handwritten Review
- Priority Queue ADT
- Heaps and Heapsort
- Sets and Union Find
- Handwritten Problem

Handwritten Problem Review

Lab 3 Written Problem: Anagrams

- Check if two strings are anagrams, assuming that they only contain lowercase letters and spaces.
- One possible solution: since you know that there are only 26 possible characters that the strings can have, just keep track of the character counts in a vector of size 26!
- **Note**: do NOT assume anything if you are asked a similar question during an interview: what if the strings can contain any character?
  - **str1**: “(/ .o.)\ ⌒ └(o` _´o)┘ ⌒ /(.o. \)”
  - **str2**: “\(o ⌒ o)_ /`⌒ └/..\┘ _´(.□.)(□)”
- We’ll cover this case in a later lab!

this example is a bit extreme, but you can’t predict what characters the strings may have

Lab 3 Written Problem: Anagrams

```
bool isAnagram(string s1, string s2) {
    vector<int> charsCount(26, 0);

    for (char letter: s1)
        if (letter != ' ')
            ++charsCount[letter - 'a'];

    for (char letter: s2) {
        if (letter != ' ') {
            if (--charsCount[letter - 'a'] < 0)
                return false;
        }
    }

    for (int count: charsCount) {
        if (count > 0) {
            return false;
        }
    }

    return true;
}
```

add to a character's count if you encounter it in the first string

go through the second string and check if the occurrences match

if you ever run out of letters or have some left over, the two strings are not anagrams!

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 **highest priority**
- 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
```

- Template arguments:
  - the type of object stored in the priority queue
  - the underlying container used (the default is usually fine)
  - a function object (functor) type used to determine the priority between two objects - defaults to less<T>, which creates a MAX HEAP
- 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

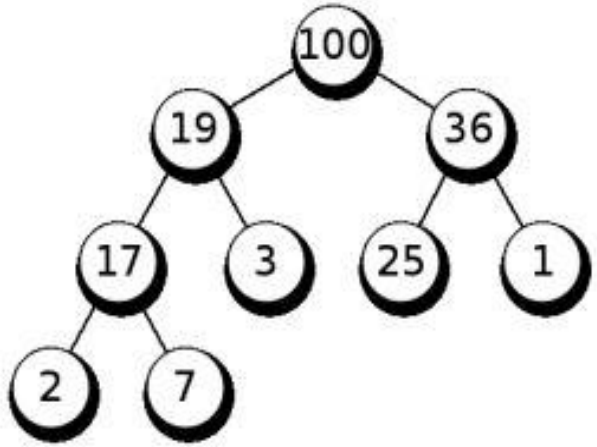
- A max binary heap is a binary tree with the following properties:
  - Each node has an equal or higher priority to the priority of both of its children (based on the comparator)
  - It is complete: all levels of the heap are full, except possibly the last
    - the last level is filled from left to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

Tree-based



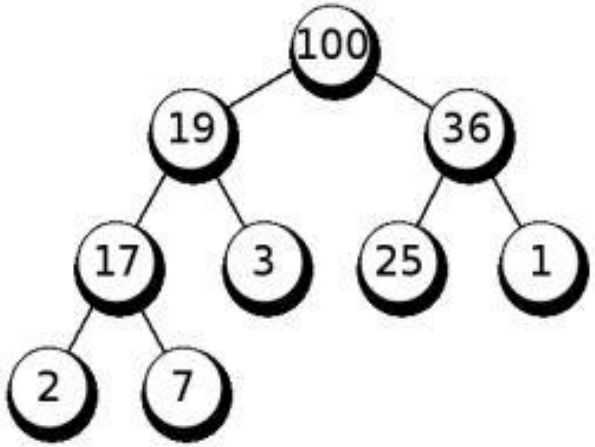
Array-based

[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...

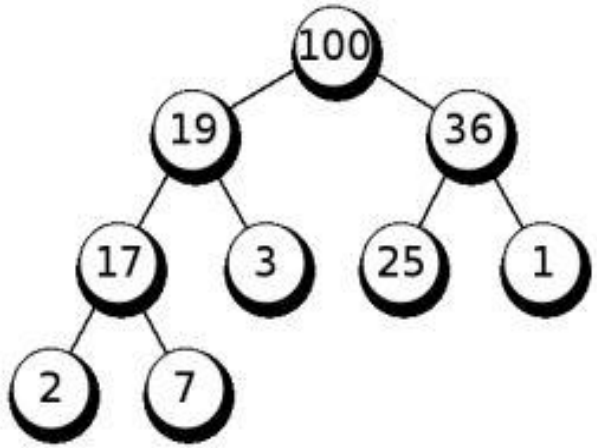
What is the index of *i's* parent?

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

Binary Heap Implementation

- Often implemented in code using arrays:

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?
(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 top down: **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 or 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 top down: **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 or 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!

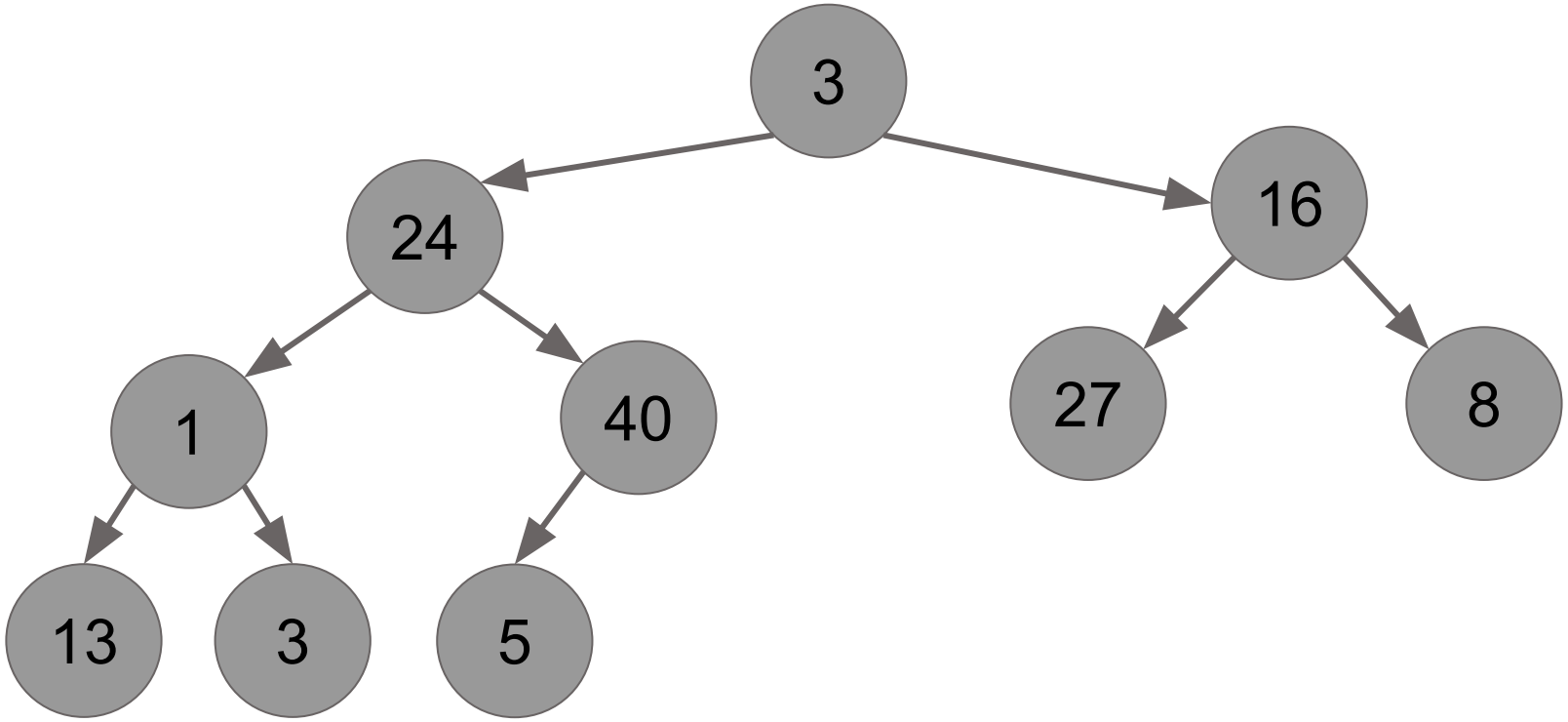
Make Heap / Heapify: Idea #2

- 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

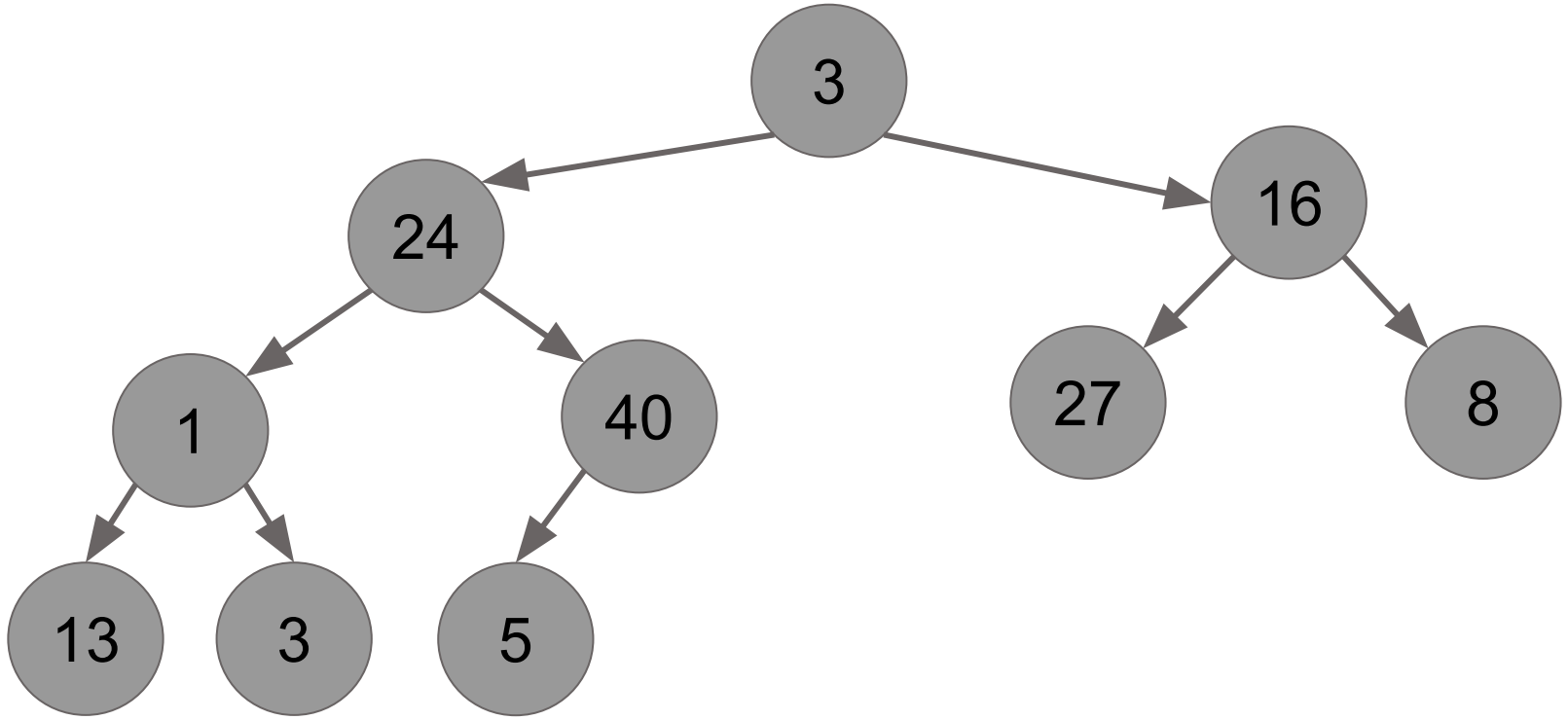
- Step 1: Initialize heap's underlying array to the unsorted array.
- Step 2: There are two methods we can choose to enforce heap invariants:
  - Method #1: repeatedly call **fixUp()** starting from the top of the array and moving down. This is equivalent to repeatedly inserting items into a heap.
  - 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 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.

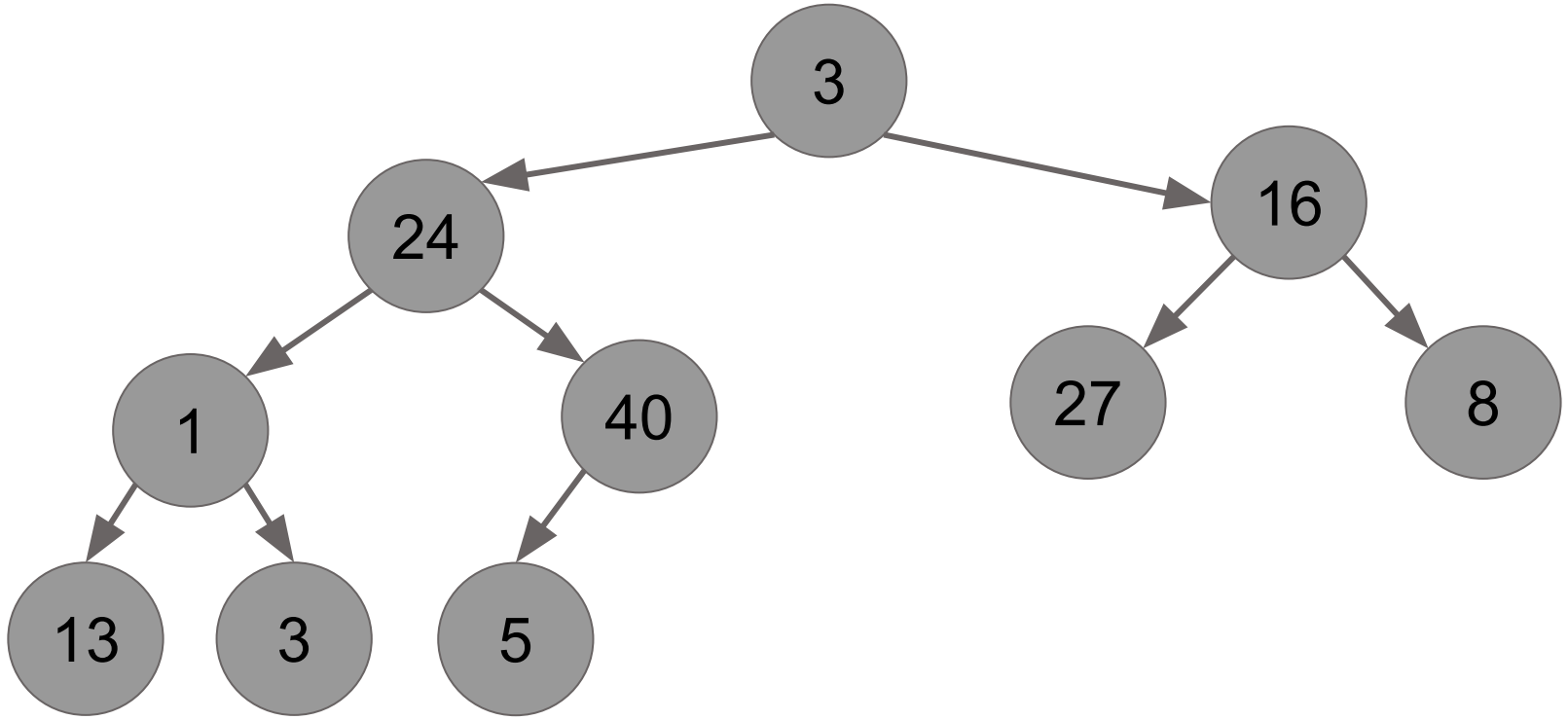
Using Fix Up: Less Efficient



Depth	Max #Nodes
1	1
2	2
3	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 Fix Up: Less Efficient

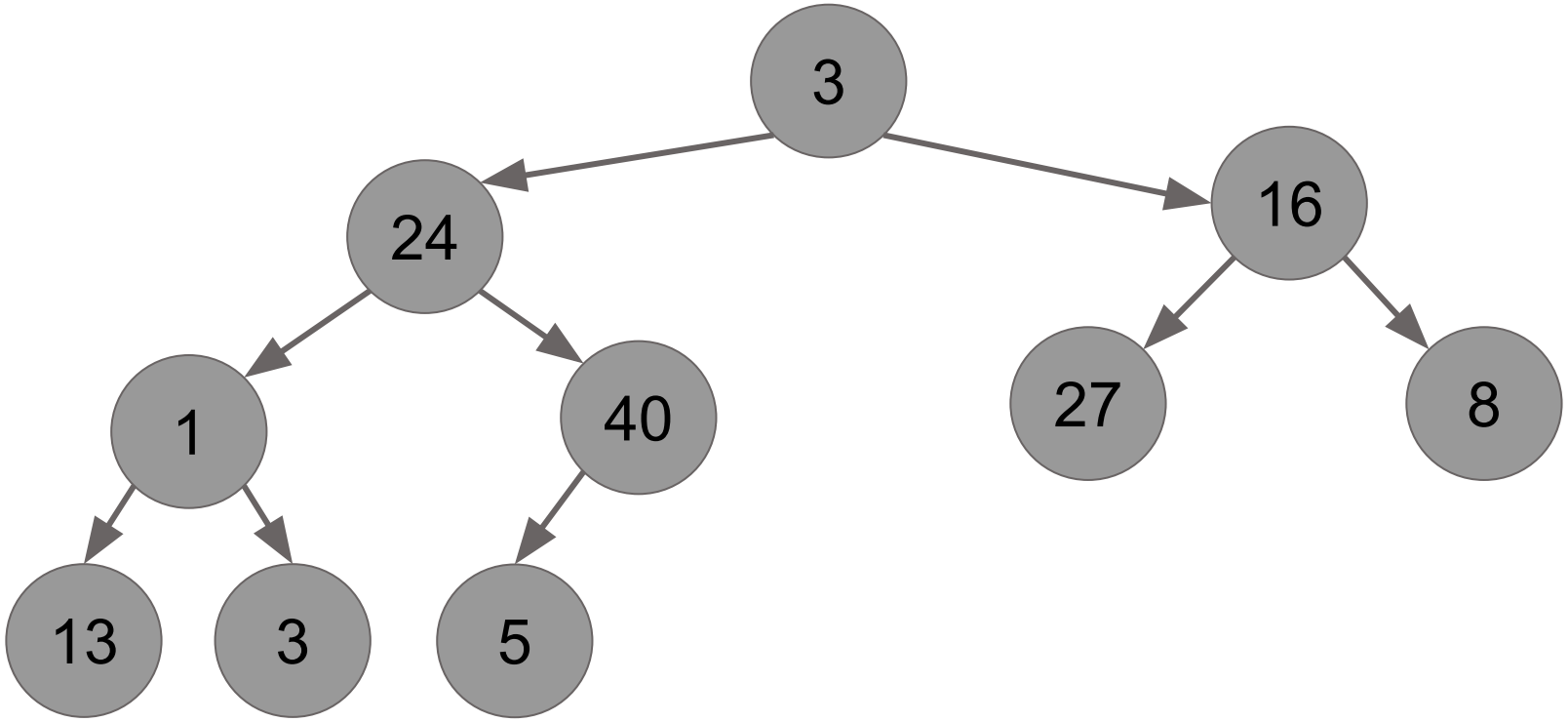


Depth	Max #Nodes
1	1
2	2
3	4
…	
Log(n)	n/2

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.

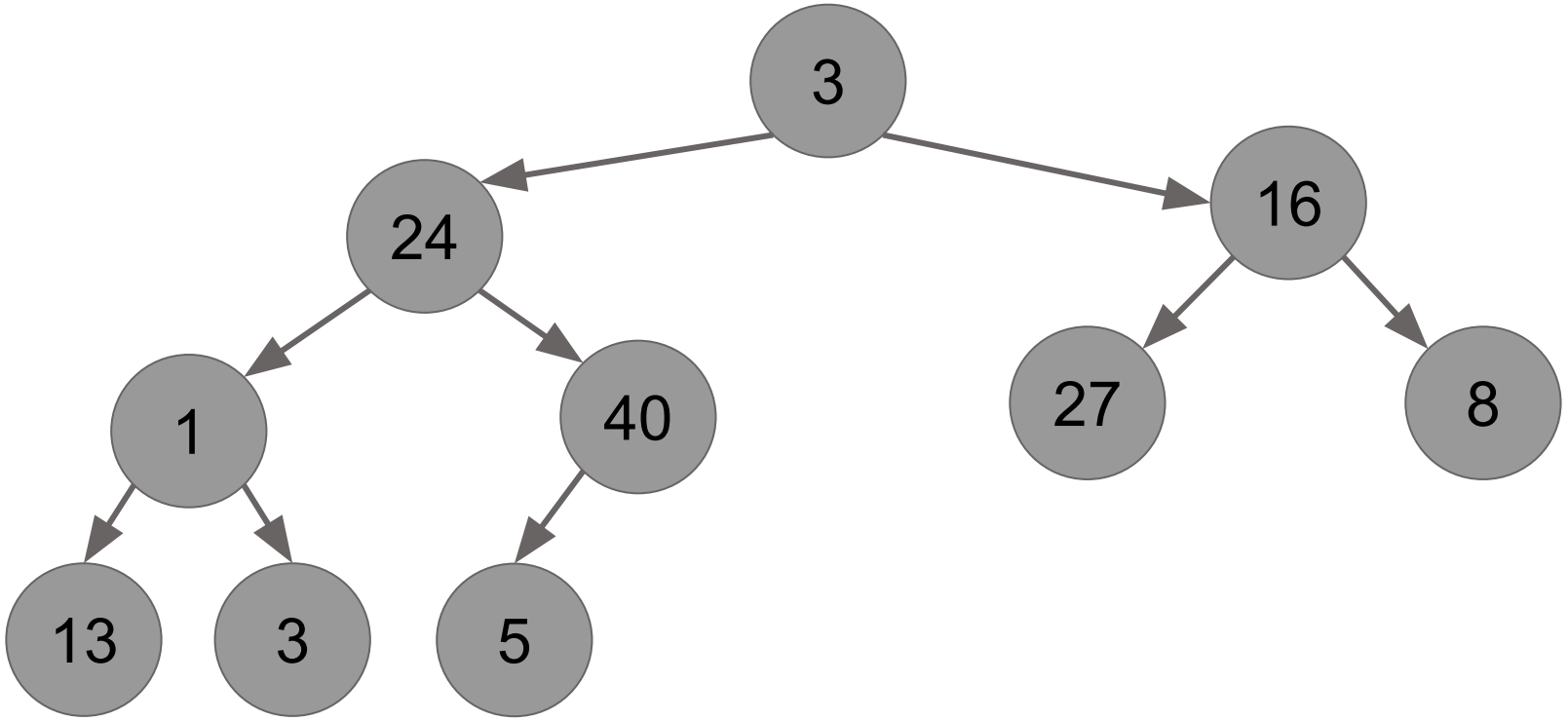
Total complexity =
1 * 1 + 2 * 2 + 3 *4 + … log(n) * (n /2) =
O(n log n)

Using Fix Down: More Efficient



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

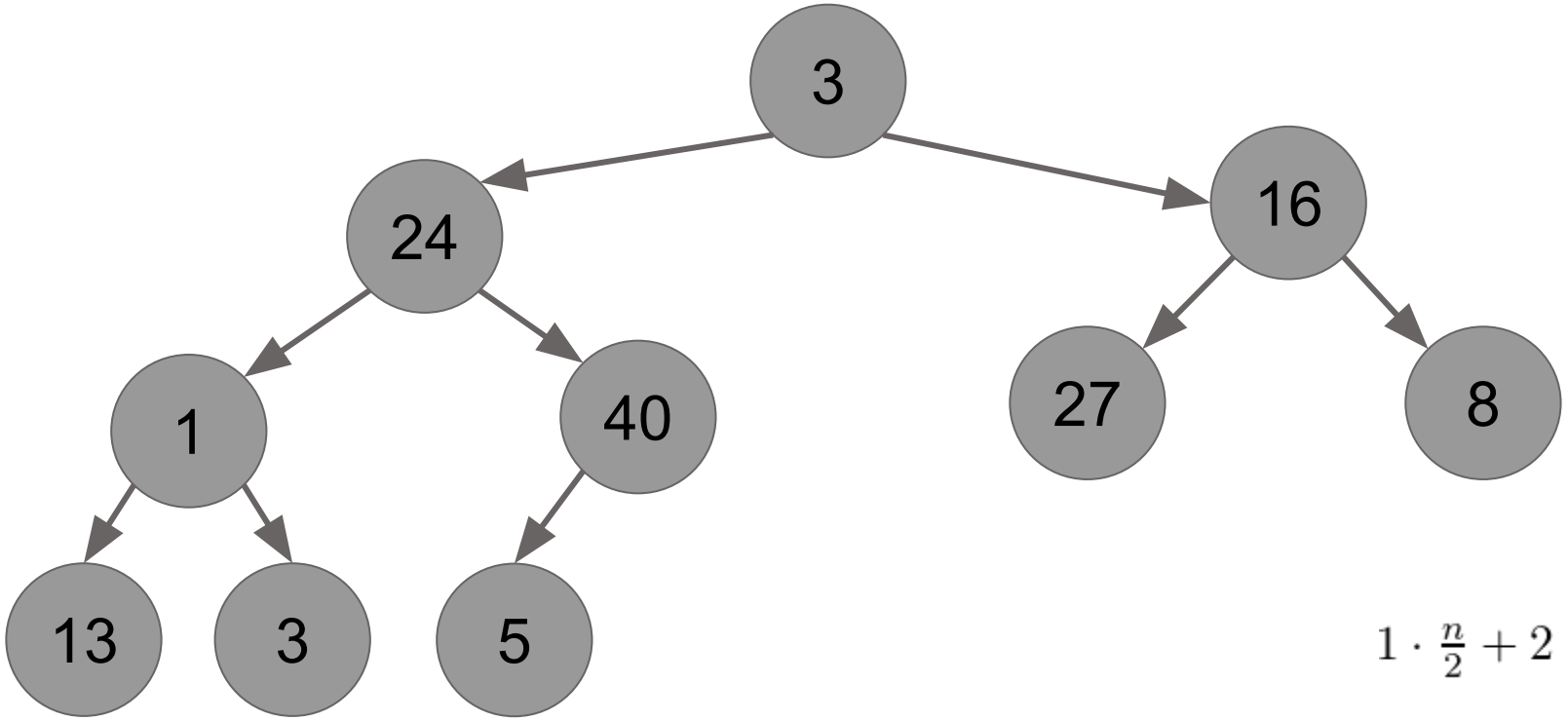
Using Fix Down: More Efficient



Height	Max #Nodes
Log(n)	1
…	
3	n/8
2	n/4
1	n/2

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

Using Fix Down: More Efficient



Height	Max #Nodes
Log(n)	1
…	
3	n/8
2	n/4
1	n/2

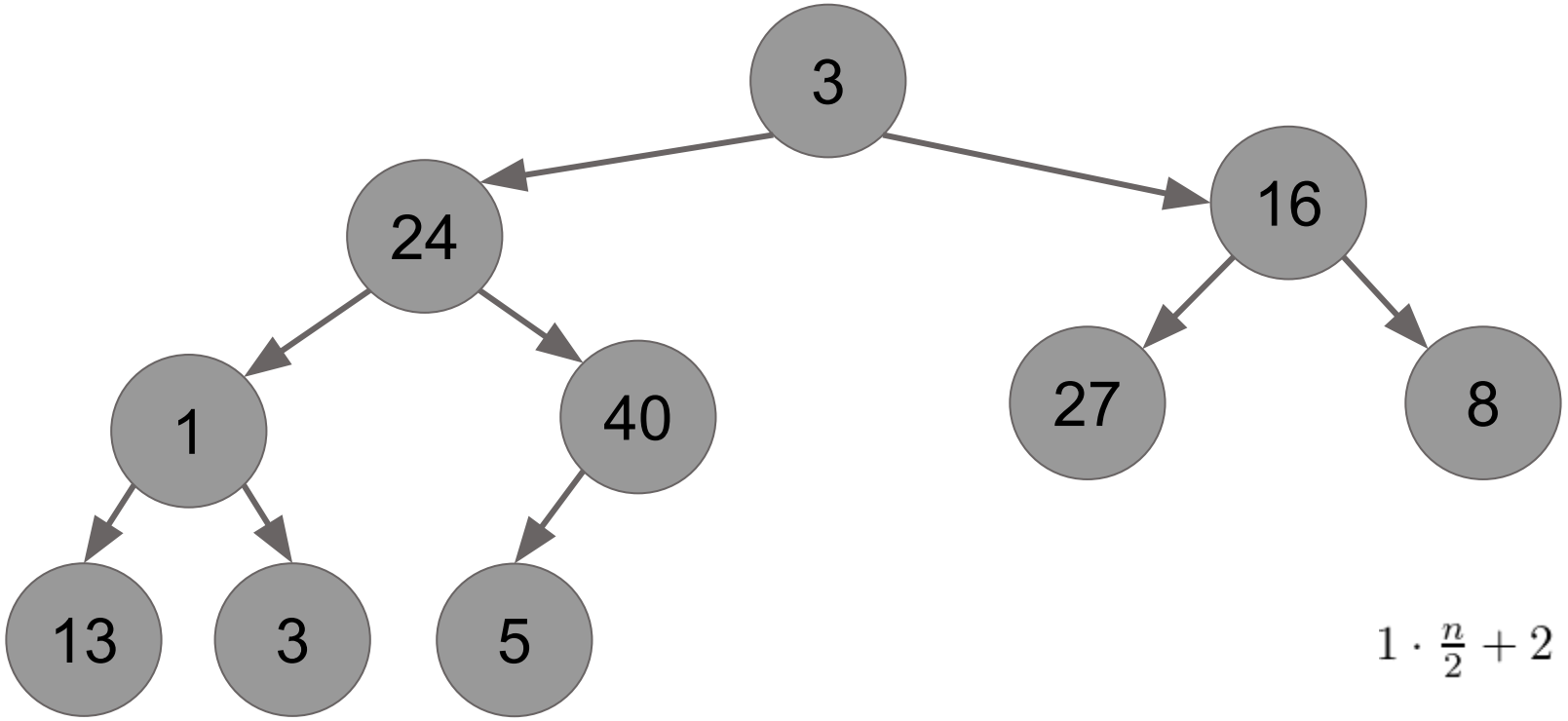
$$1 \cdot \frac{n}{2} + 2 \cdot \frac{n}{4} + 3 \cdot \frac{n}{8} + 4 \cdot \frac{n}{16} + \cdots + \log_2(n) \cdot \frac{n}{2^{\log_2(n)}}$$

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

$$= n \cdot 2$$

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

Using Fix Down: More Efficient



Height	Max #Nodes
Log(n)	1
…	
3	n/8
2	n/4
1	n/2

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

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

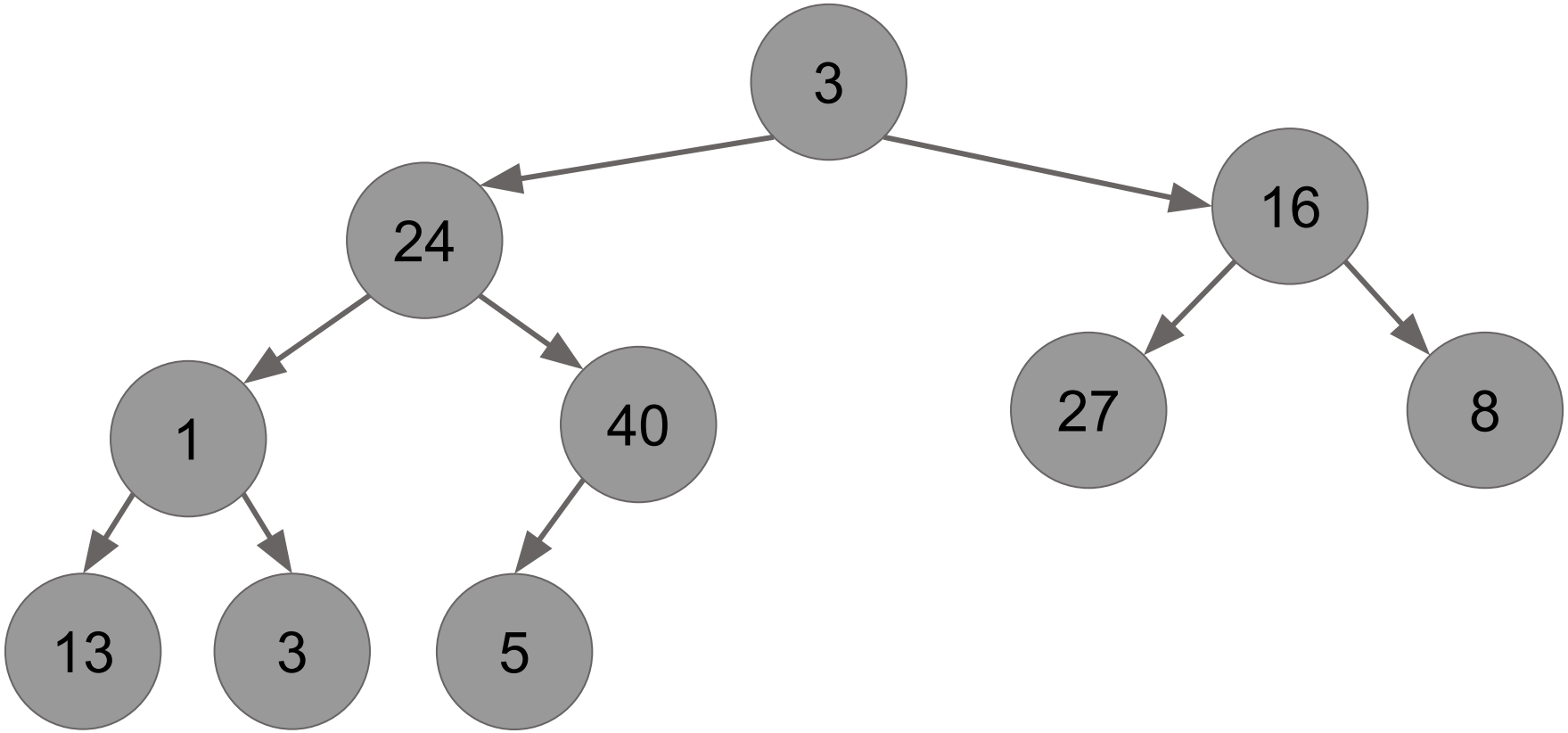
$$= n \cdot 2$$

Key idea:
Complexity of fixing the position of one item is O(the height 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)!

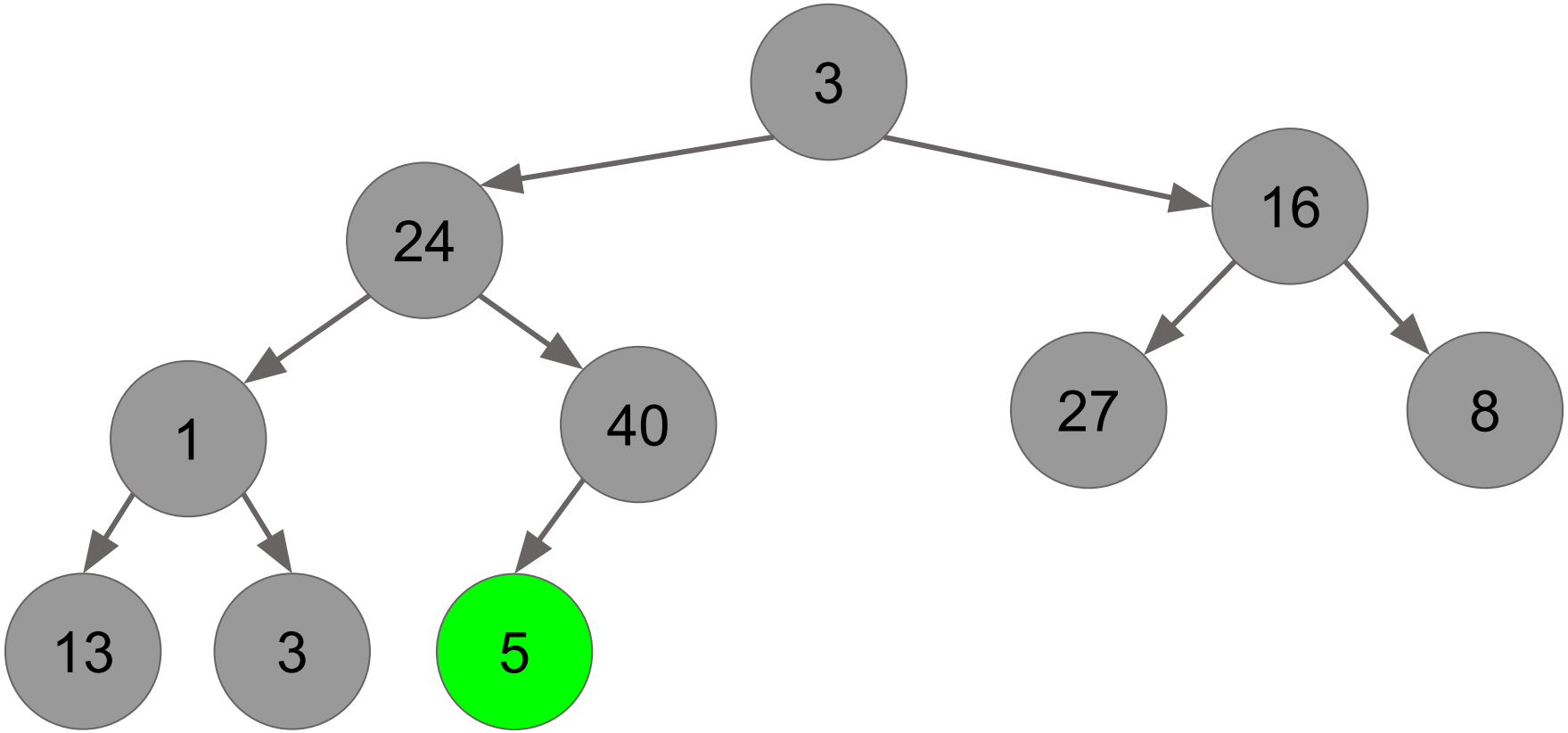
Heapify Example: Fix Down From Bottom

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



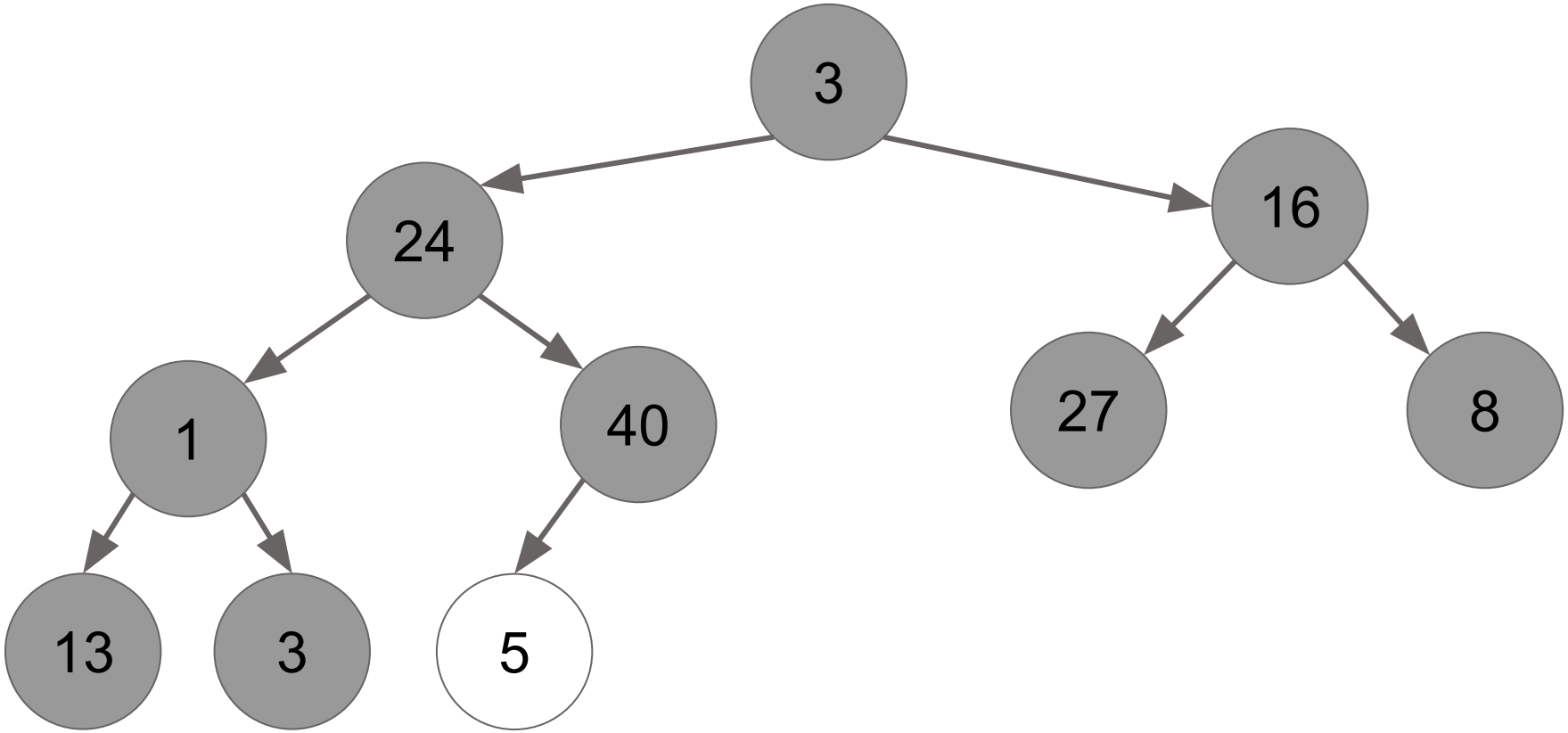
Heapify Example: Fix Down From Bottom

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



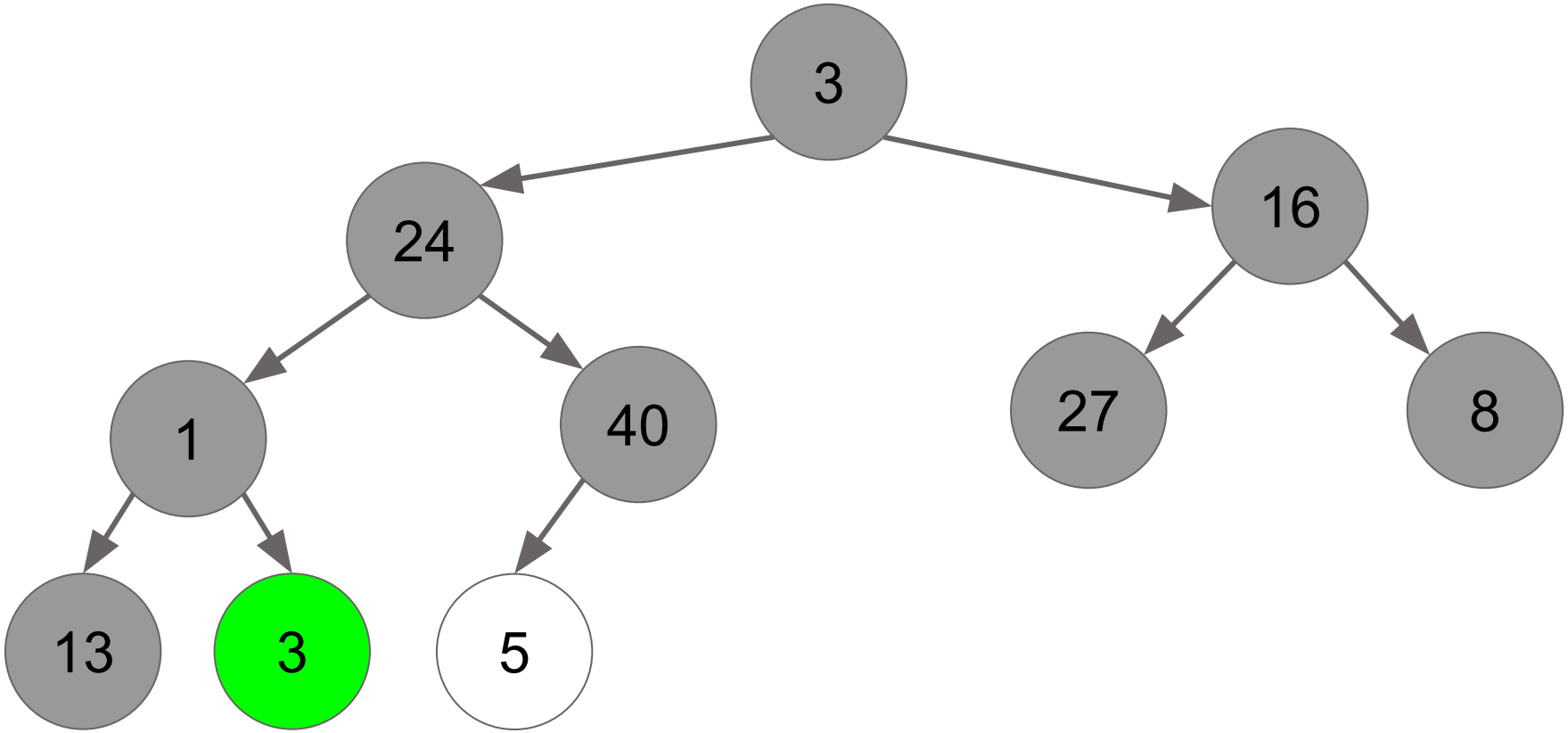
Heapify Example: Fix Down From Bottom

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



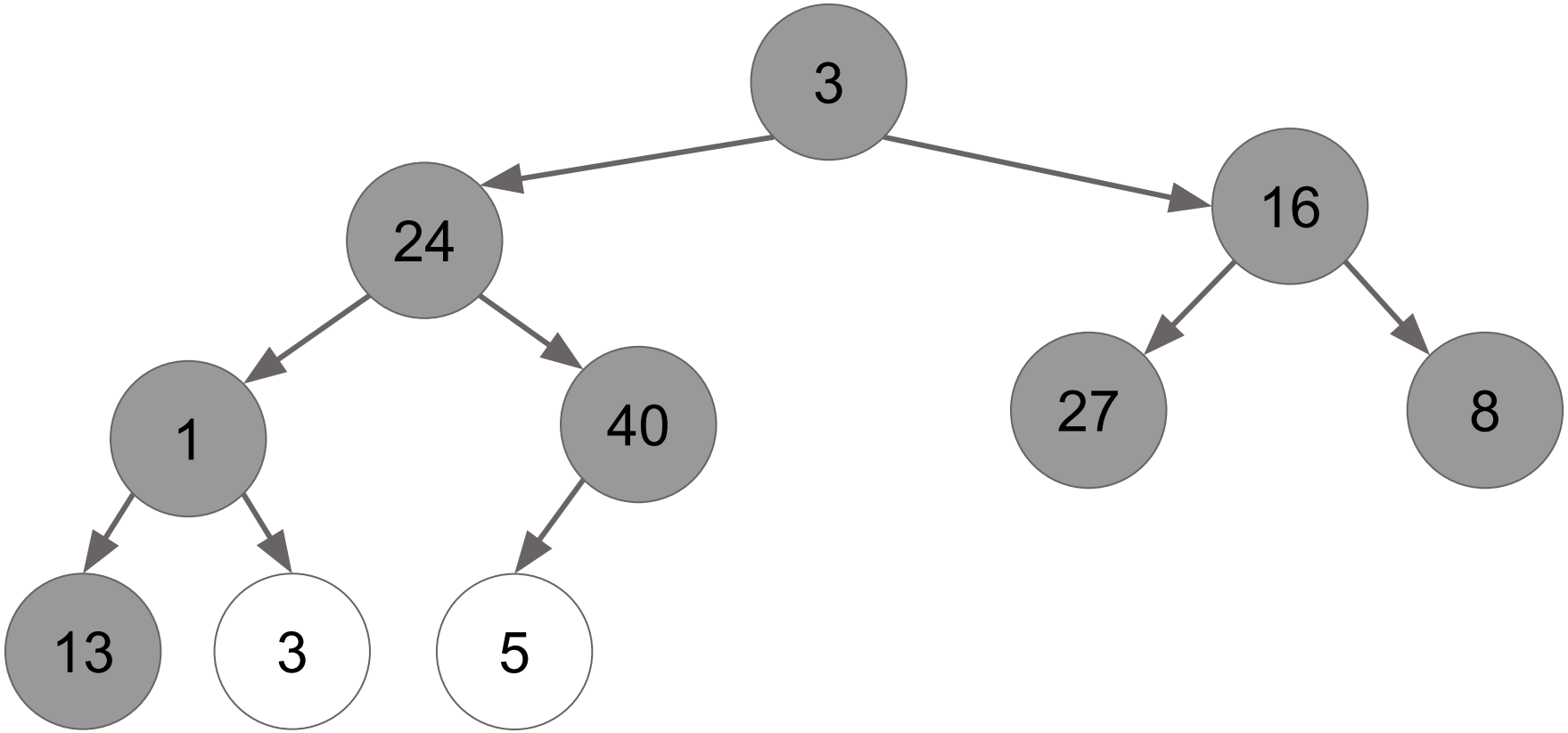
Heapify Example: Fix Down From Bottom

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



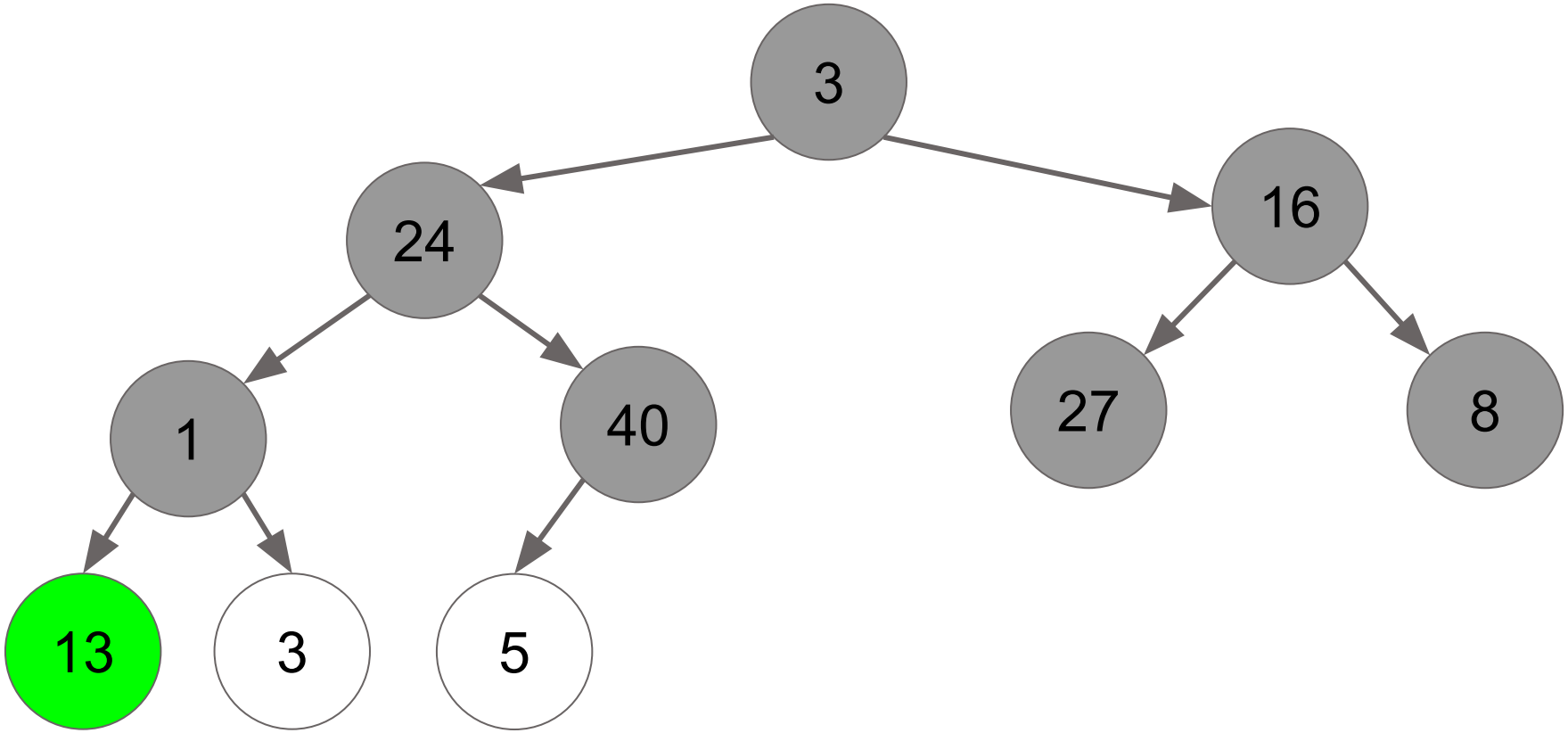
Heapify Example: Fix Down From Bottom

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



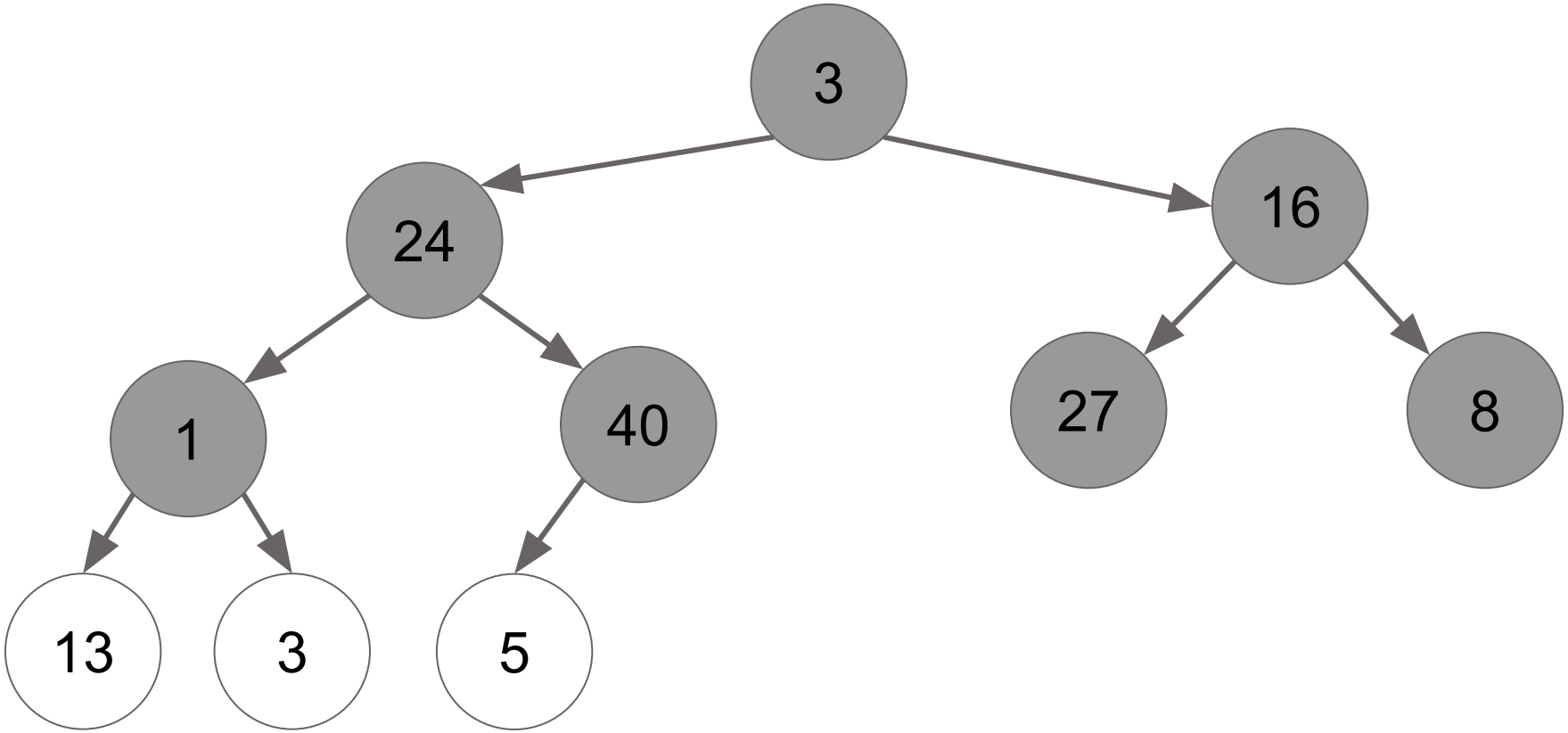
Heapify Example: Fix Down From Bottom

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



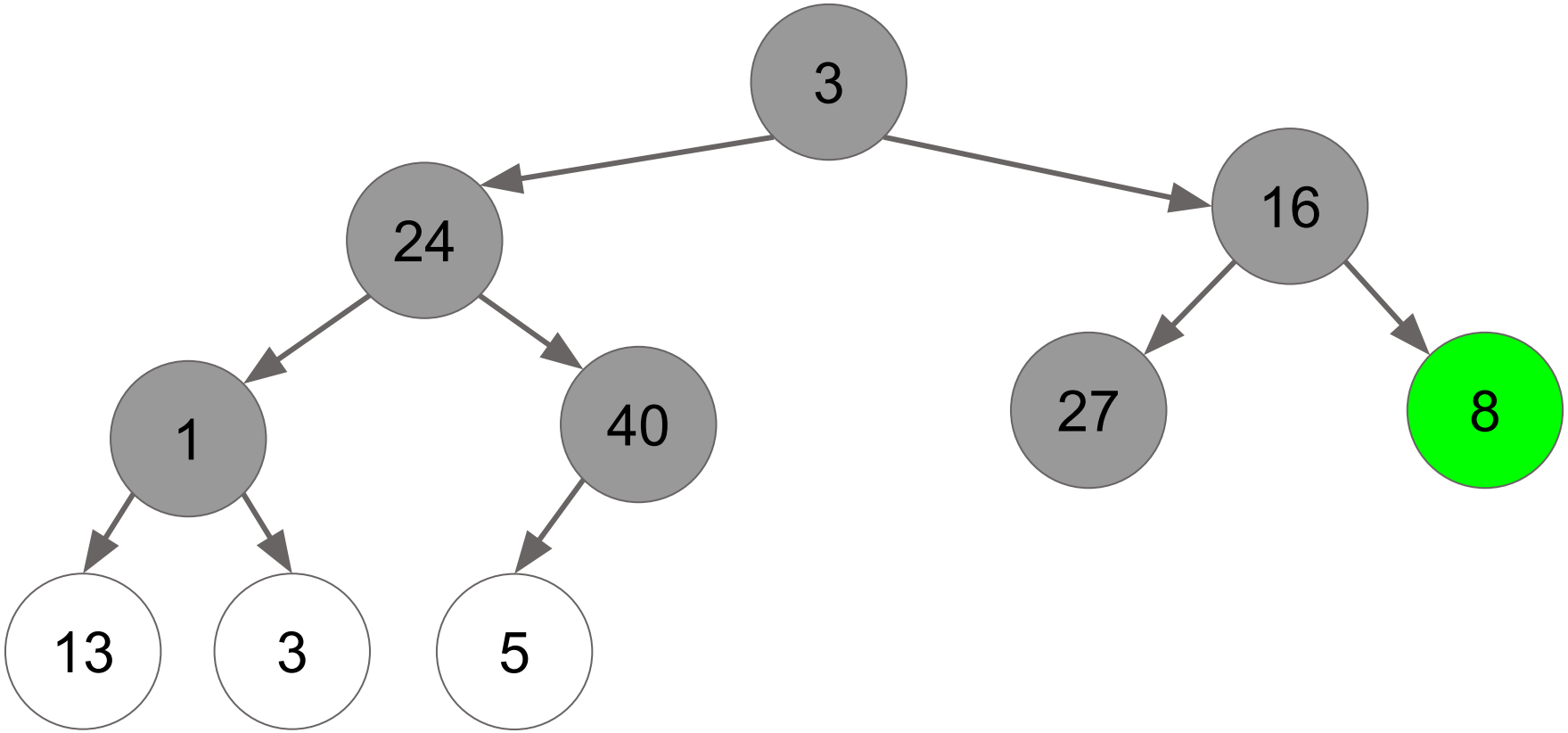
Heapify Example: Fix Down From Bottom

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



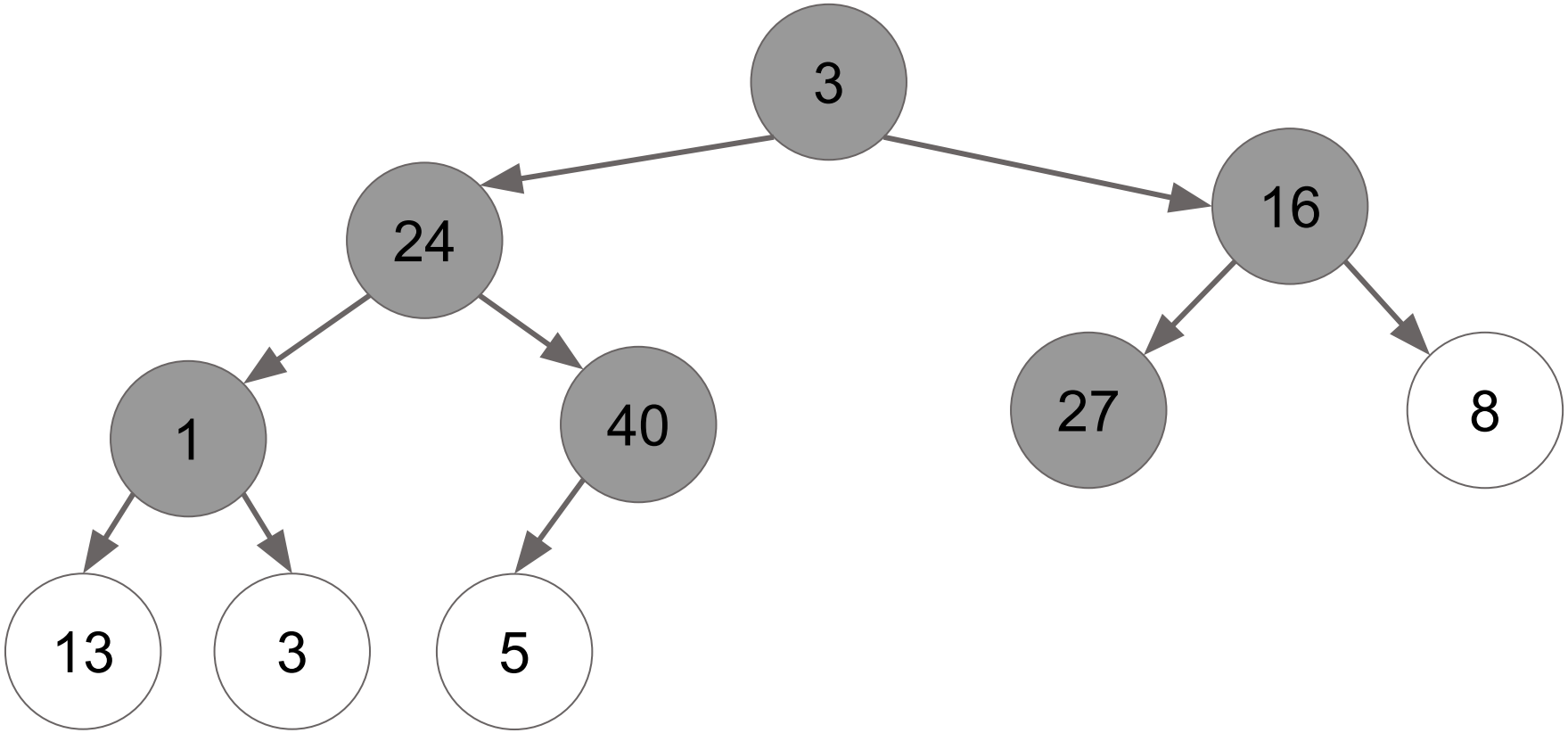
Heapify Example: Fix Down From Bottom

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



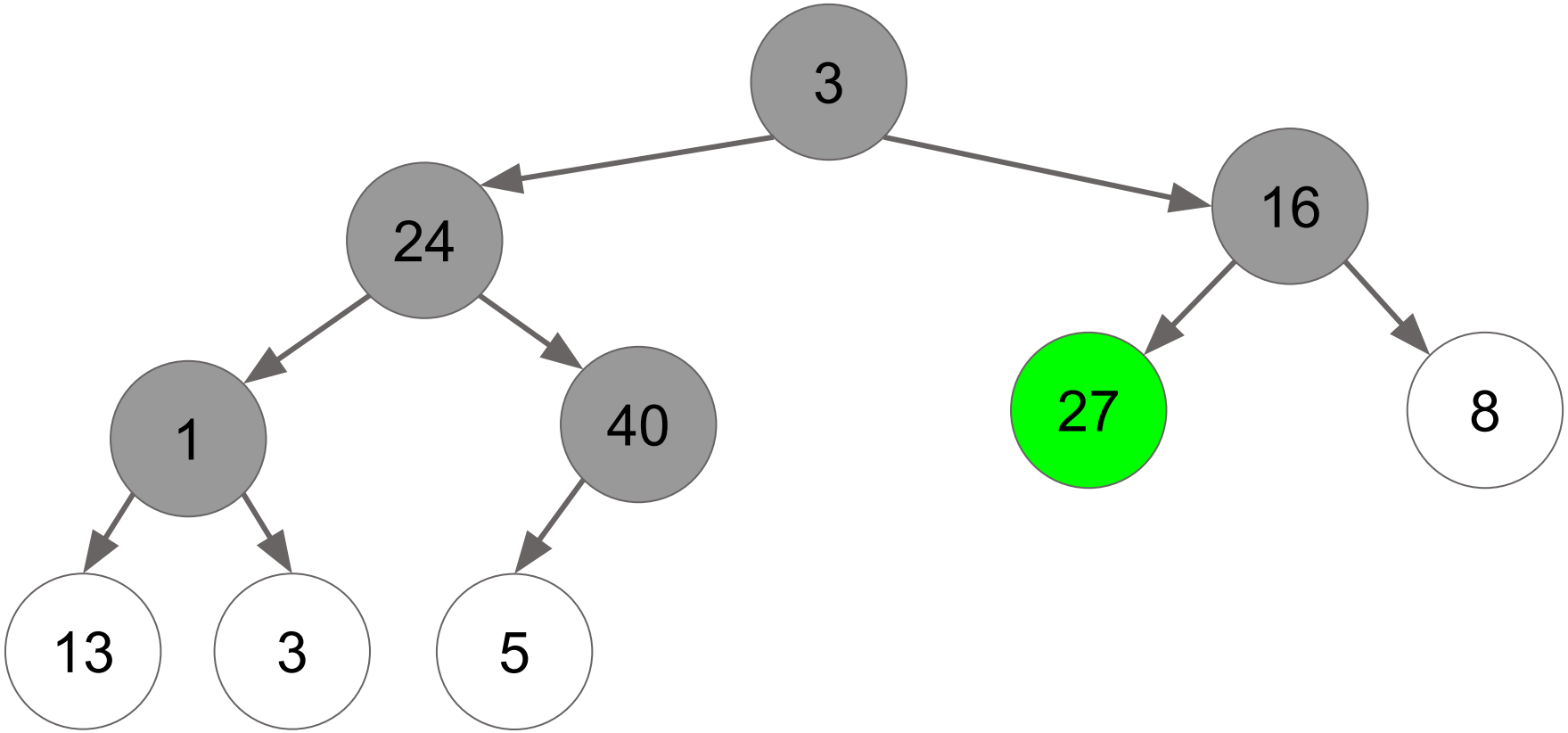
Heapify Example: Fix Down From Bottom

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



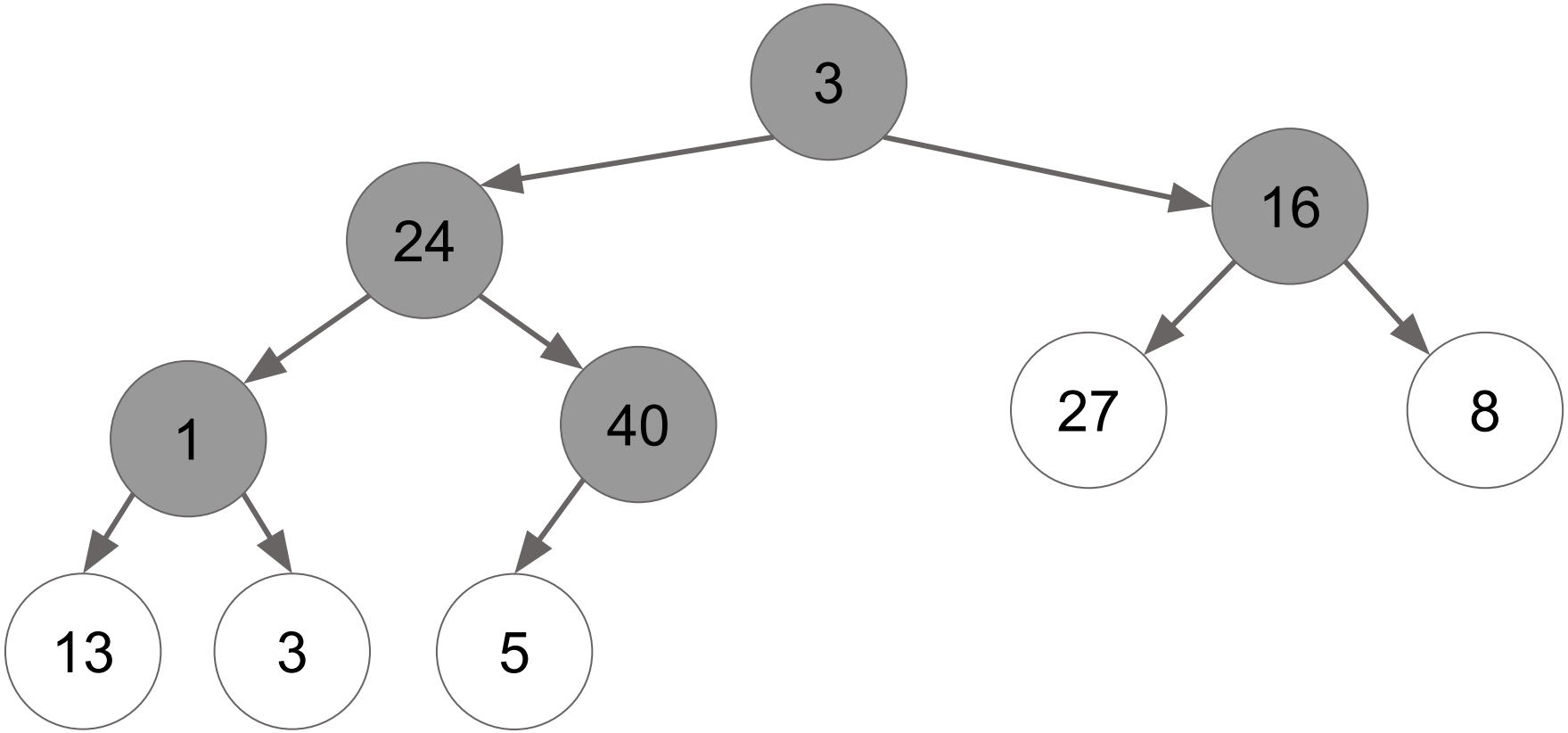
Heapify Example: Fix Down From Bottom

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



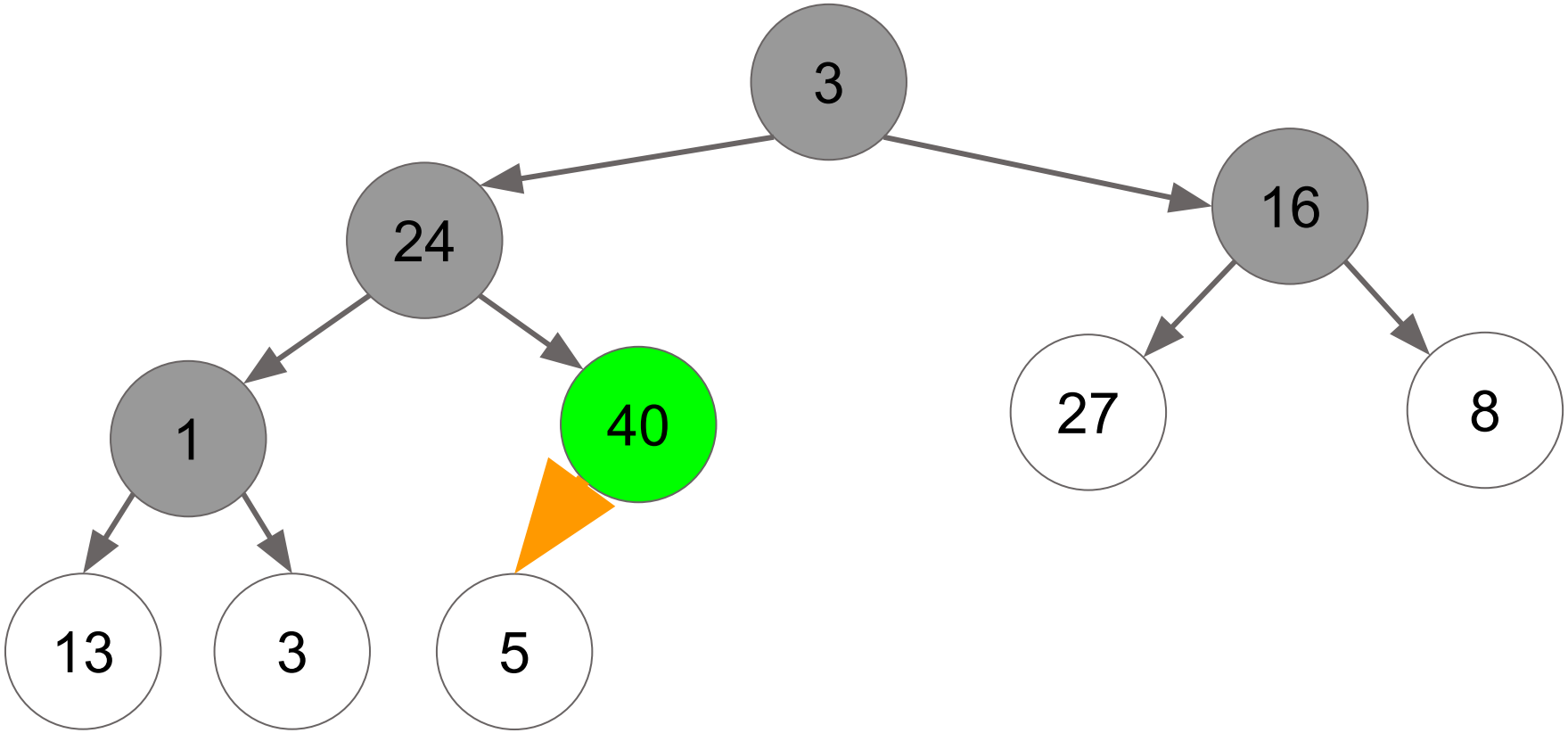
Heapify Example: Fix Down From Bottom

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



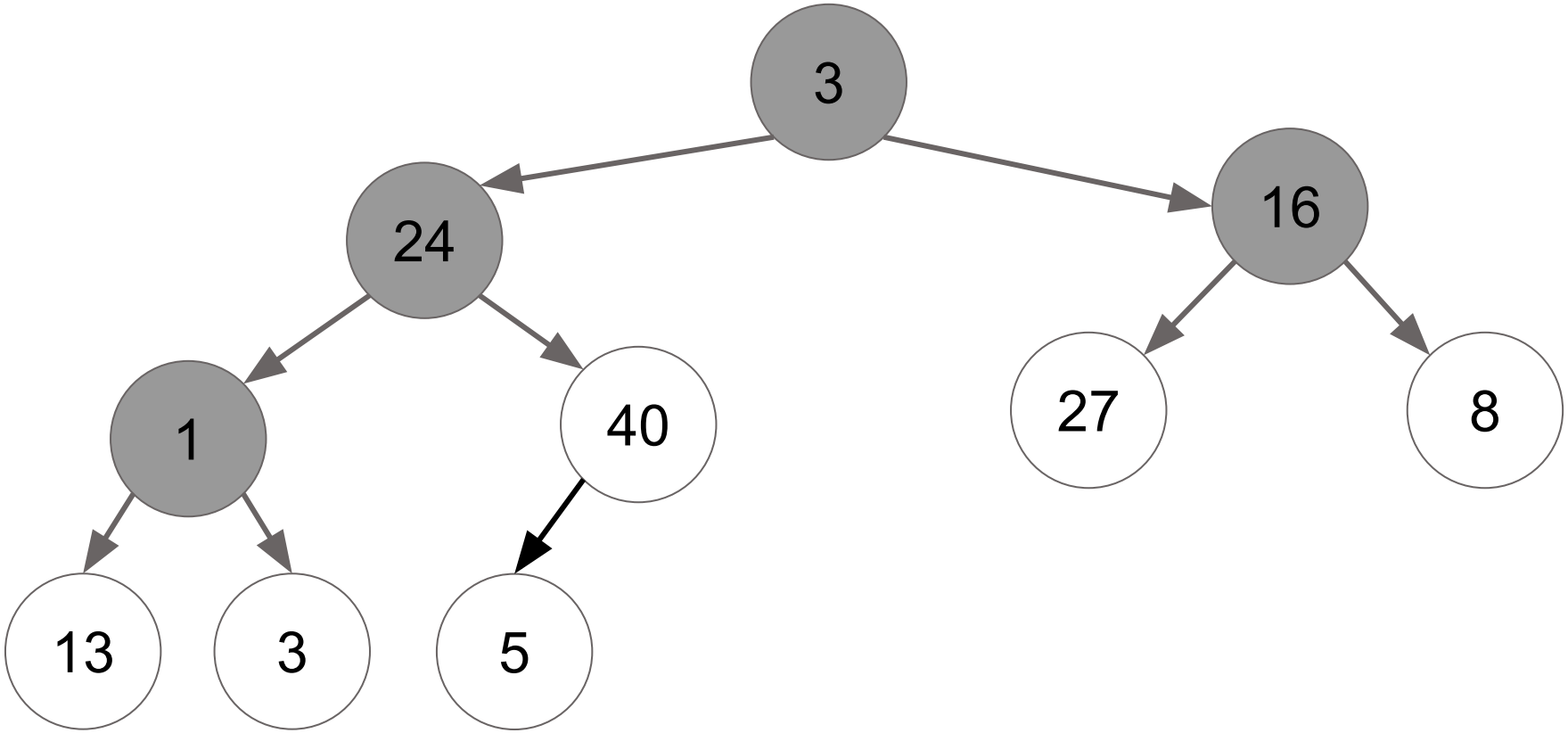
Heapify Example: Fix Down From Bottom

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



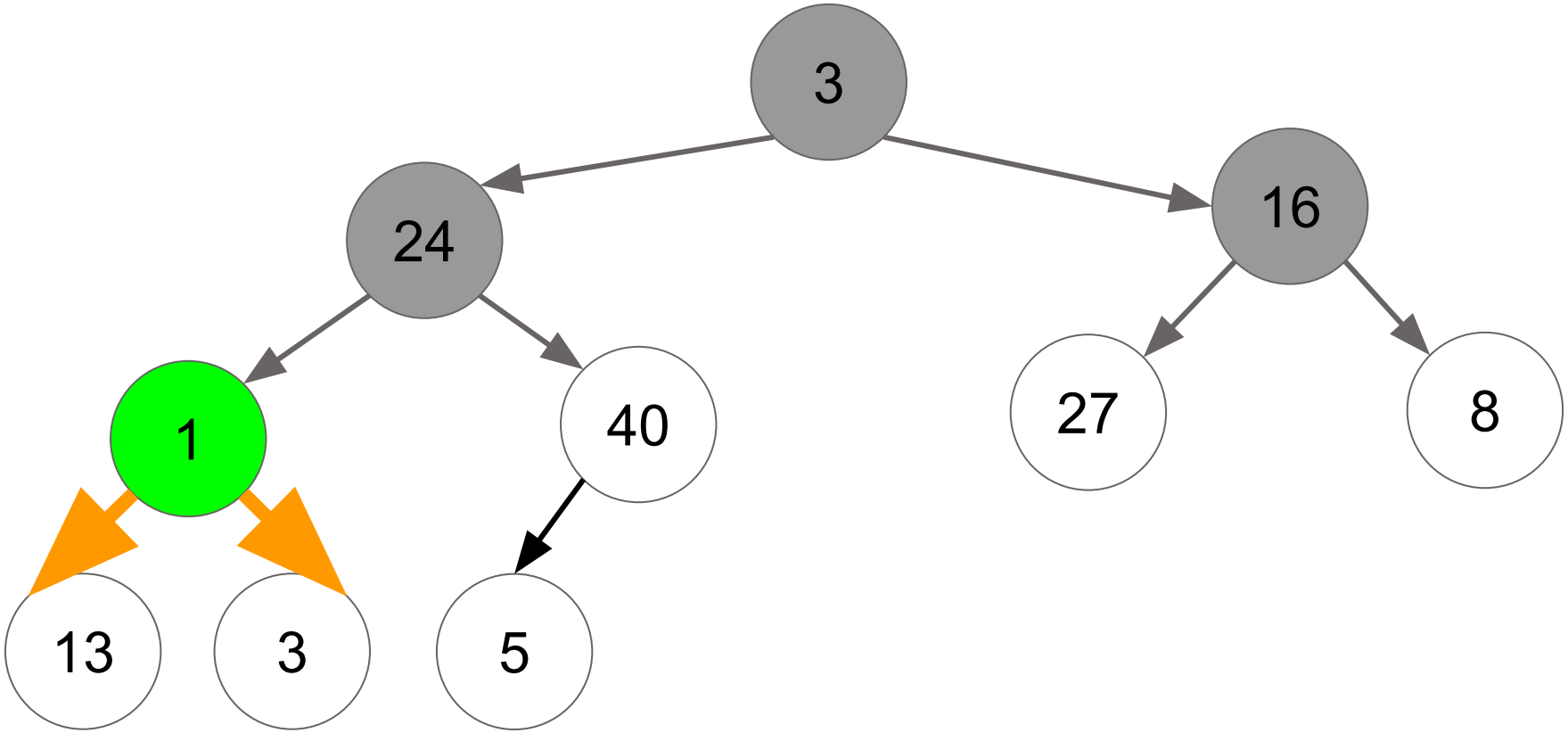
Heapify Example: Fix Down From Bottom

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



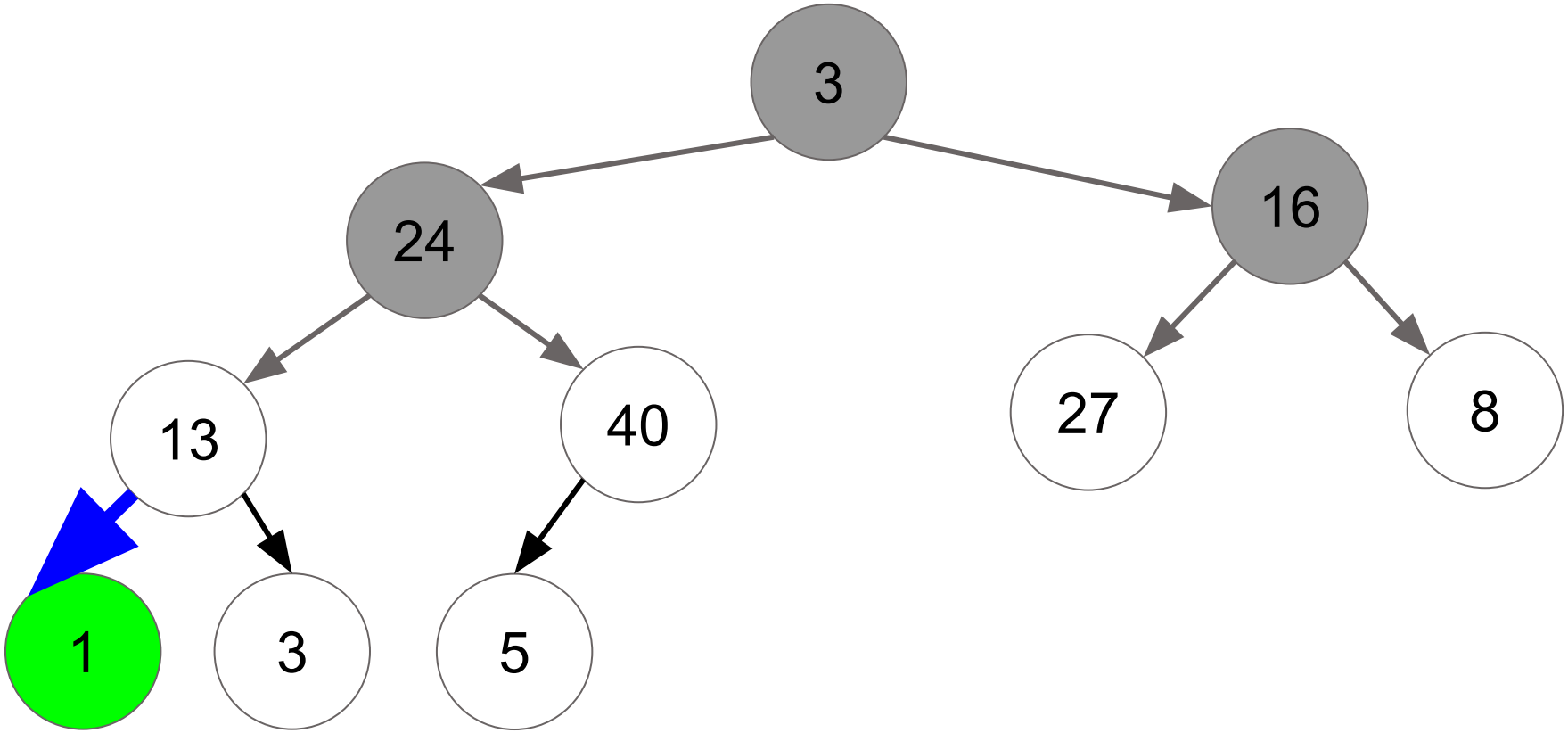
Heapify Example: Fix Down From Bottom

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



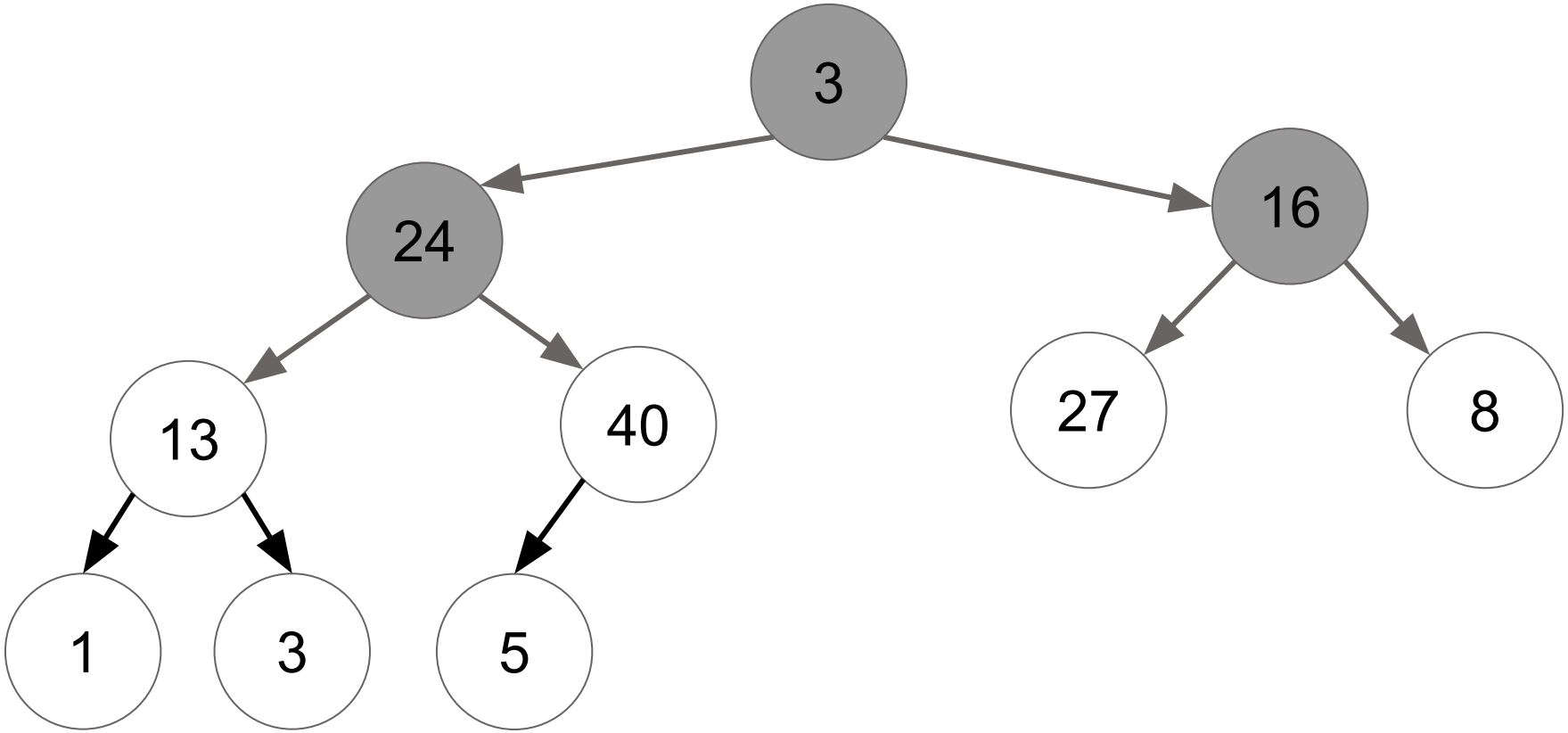
Heapify Example: Fix Down From Bottom

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



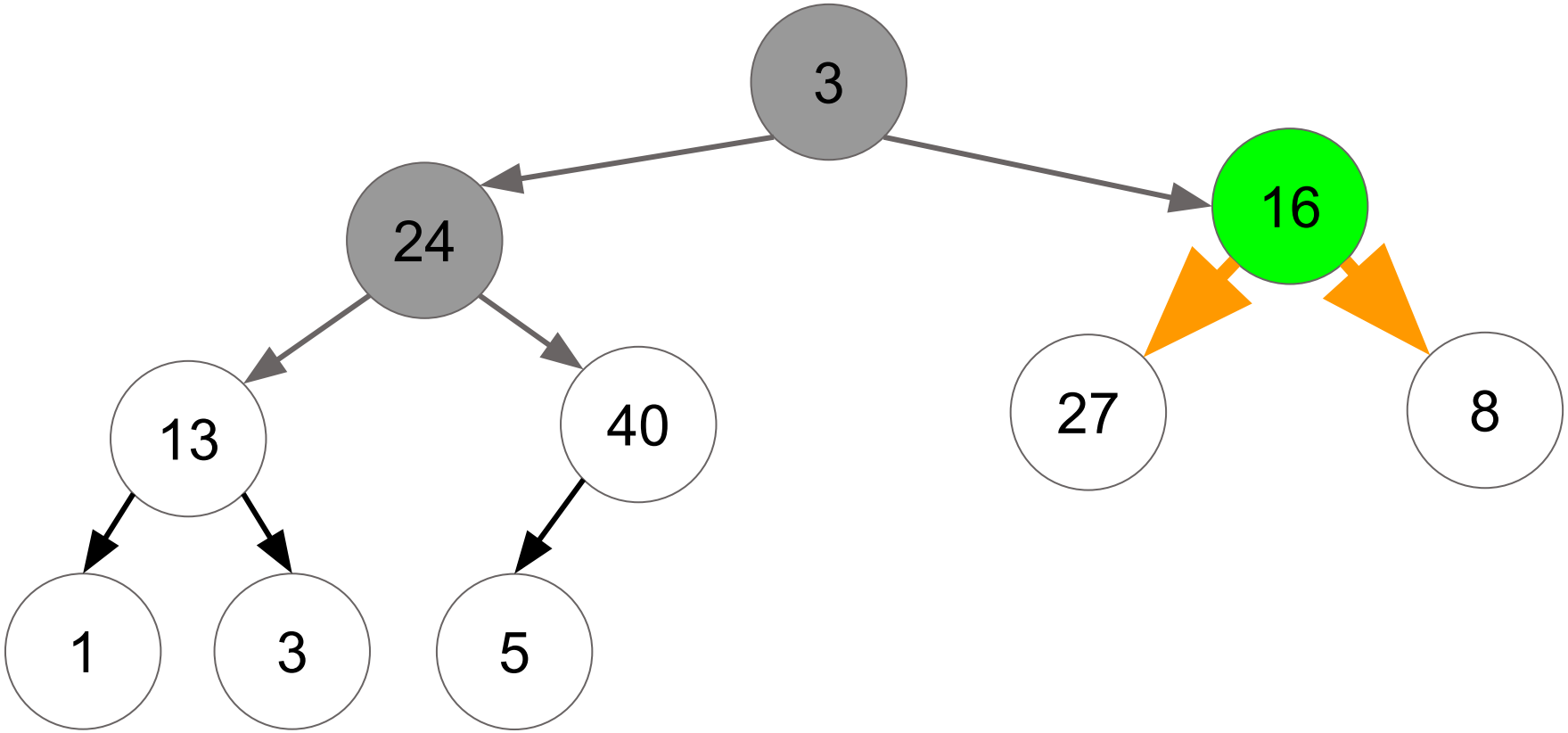
Heapify Example: Fix Down From Bottom

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



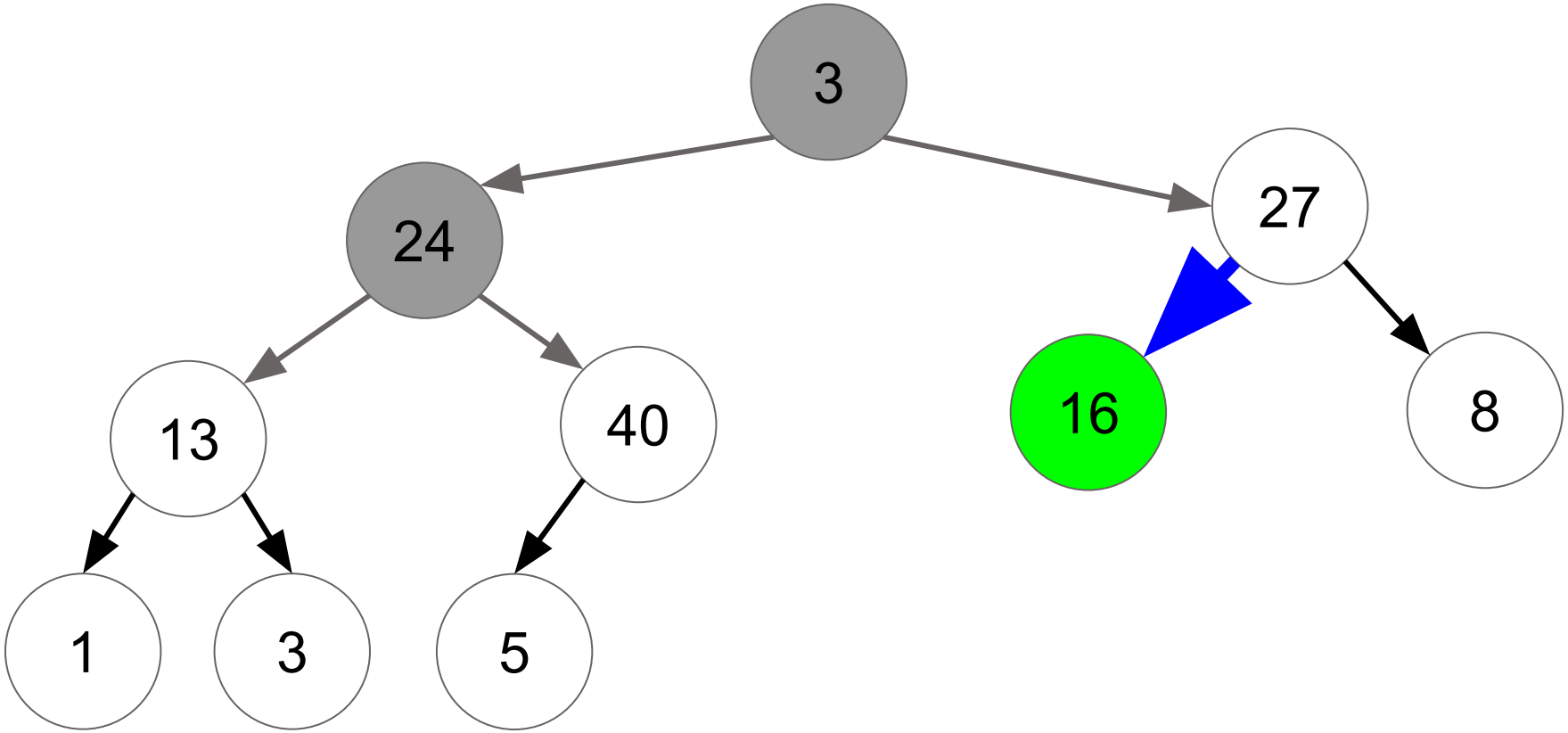
Heapify Example: Fix Down From Bottom

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



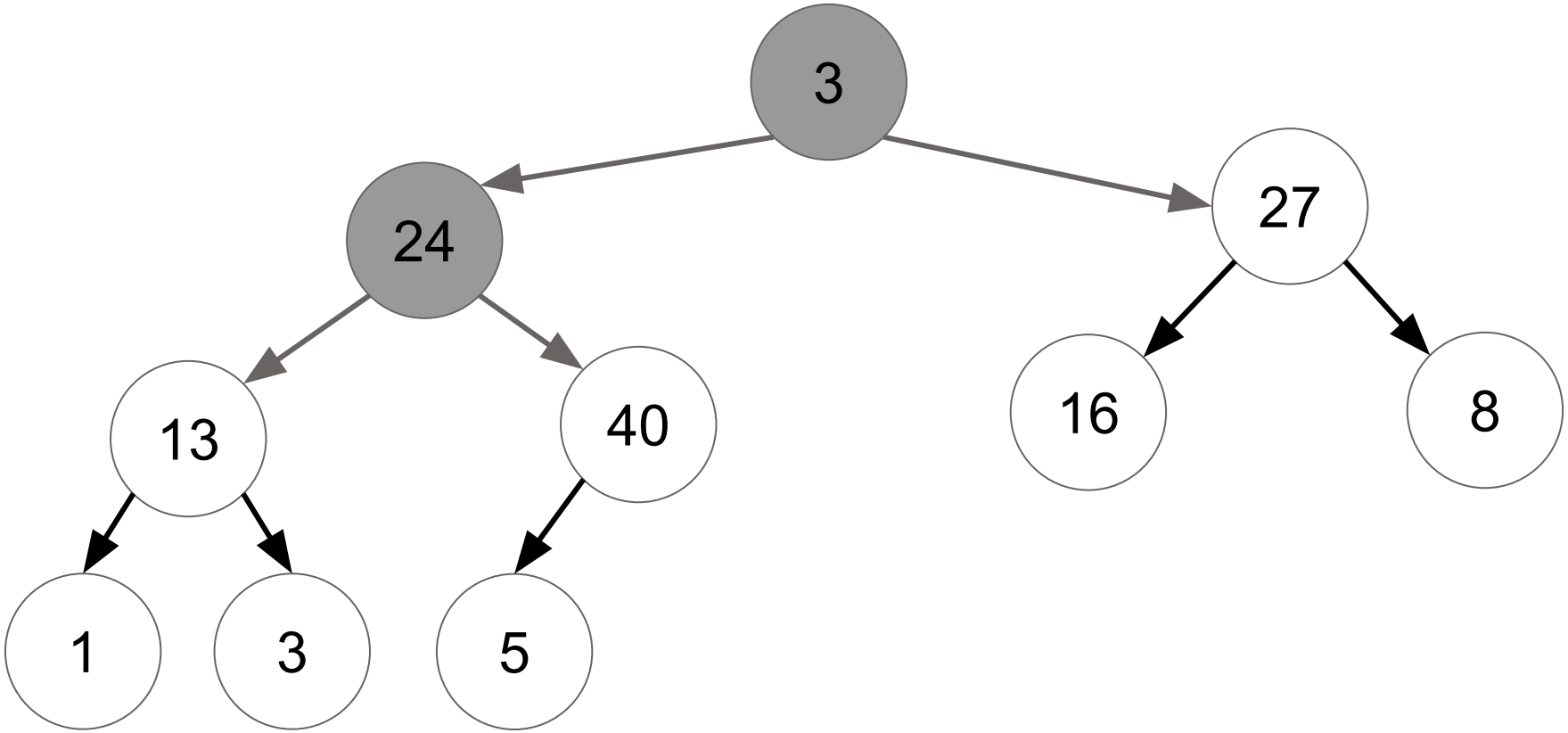
Heapify Example: Fix Down From Bottom

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



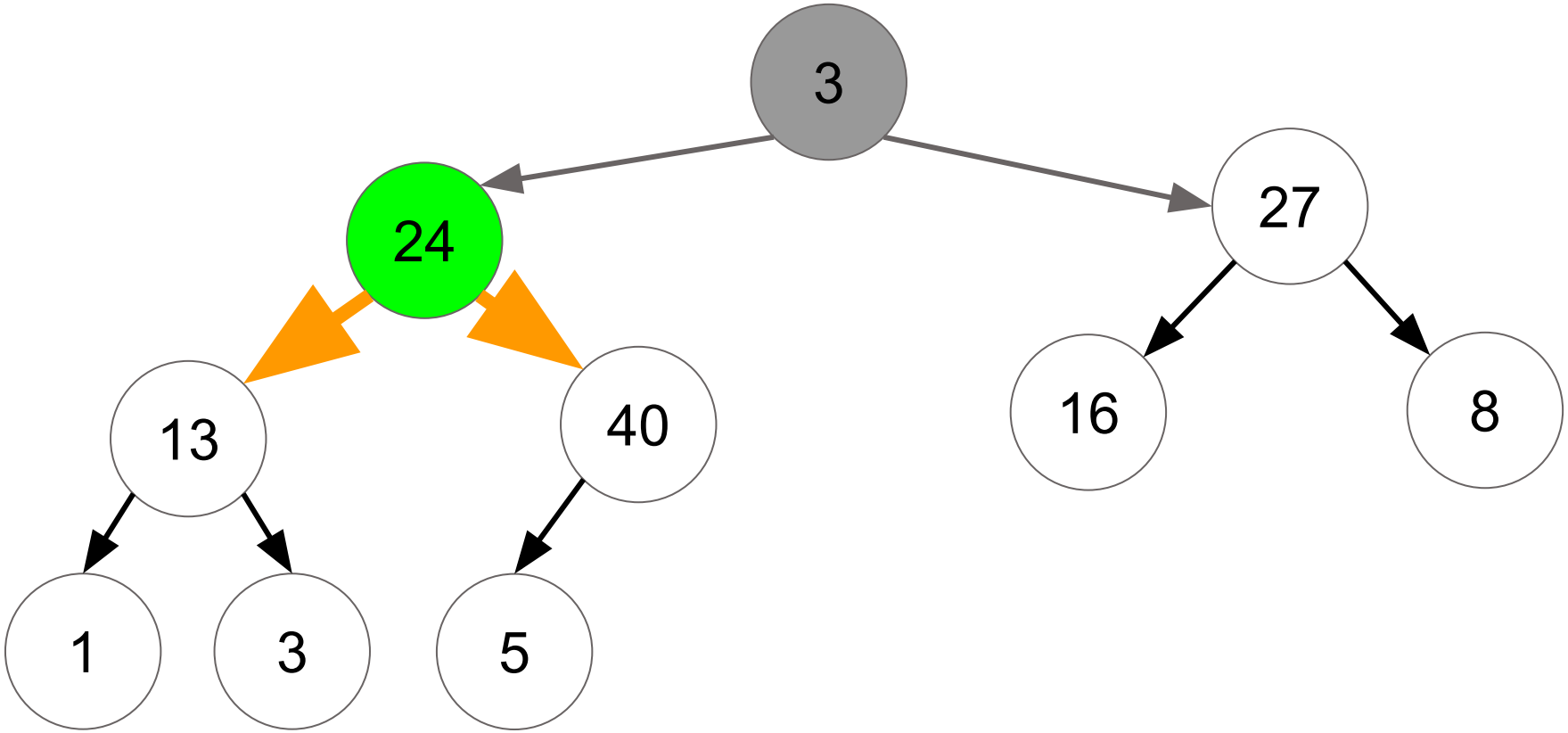
Heapify Example: Fix Down From Bottom

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



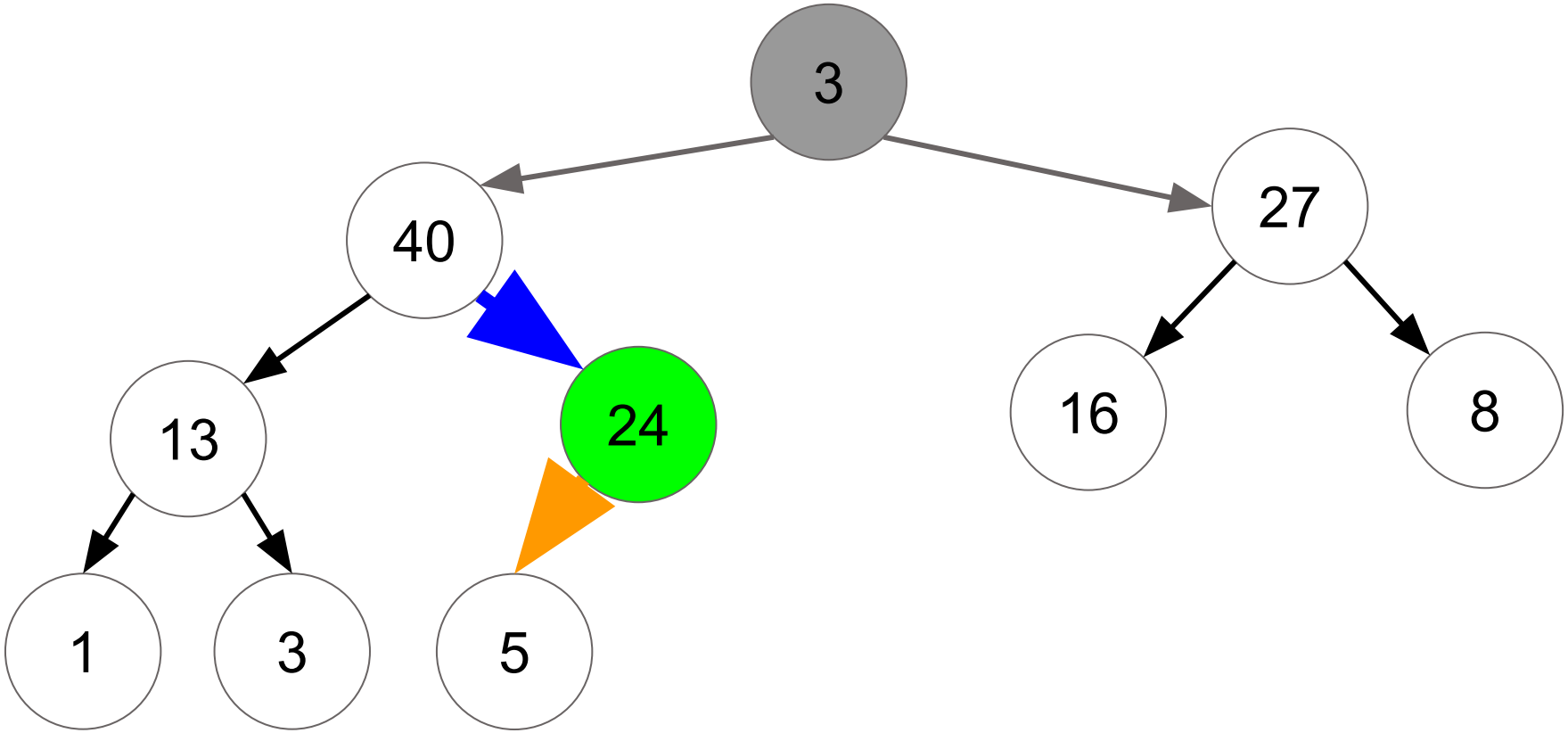
Heapify Example: Fix Down From Bottom

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



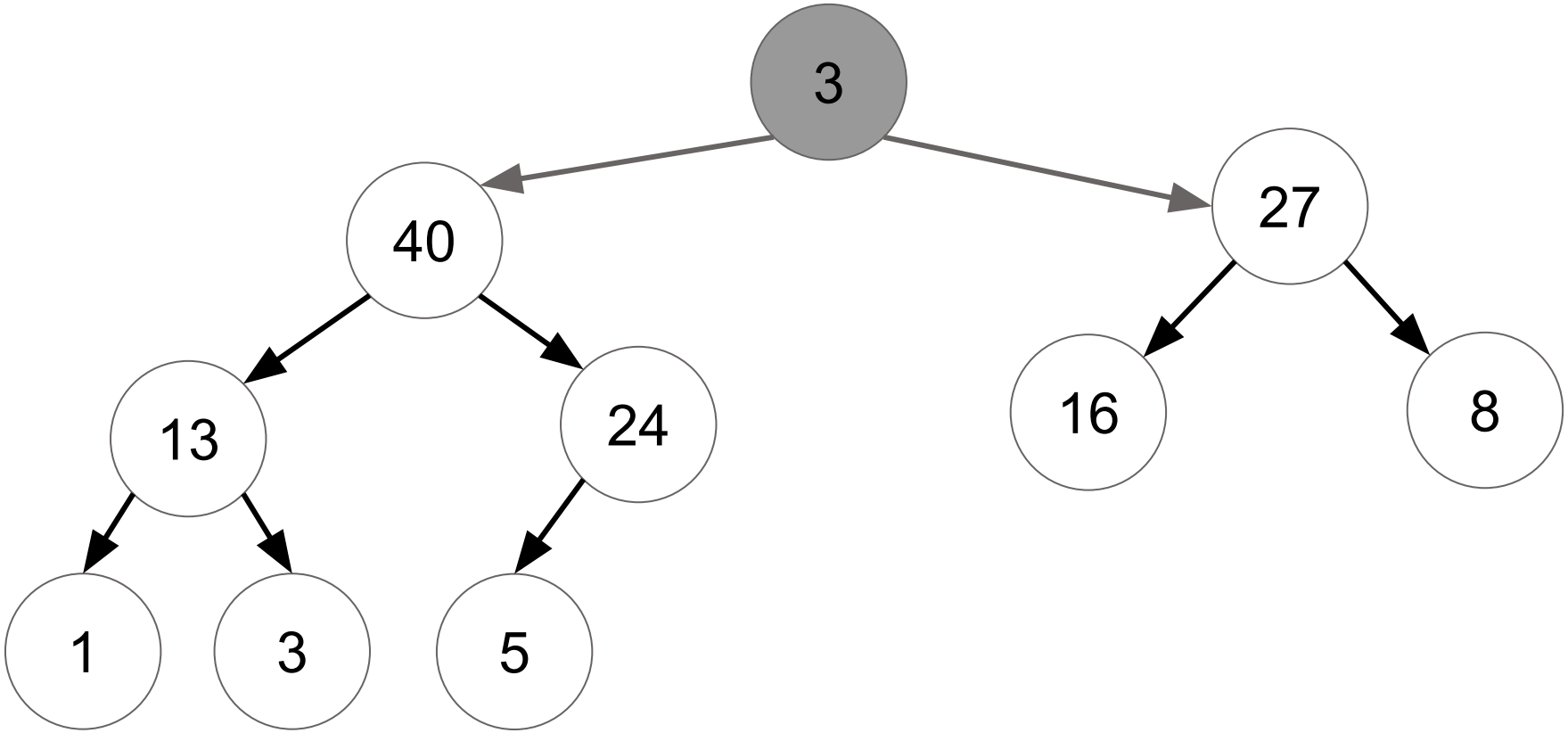
Heapify Example: Fix Down From Bottom

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



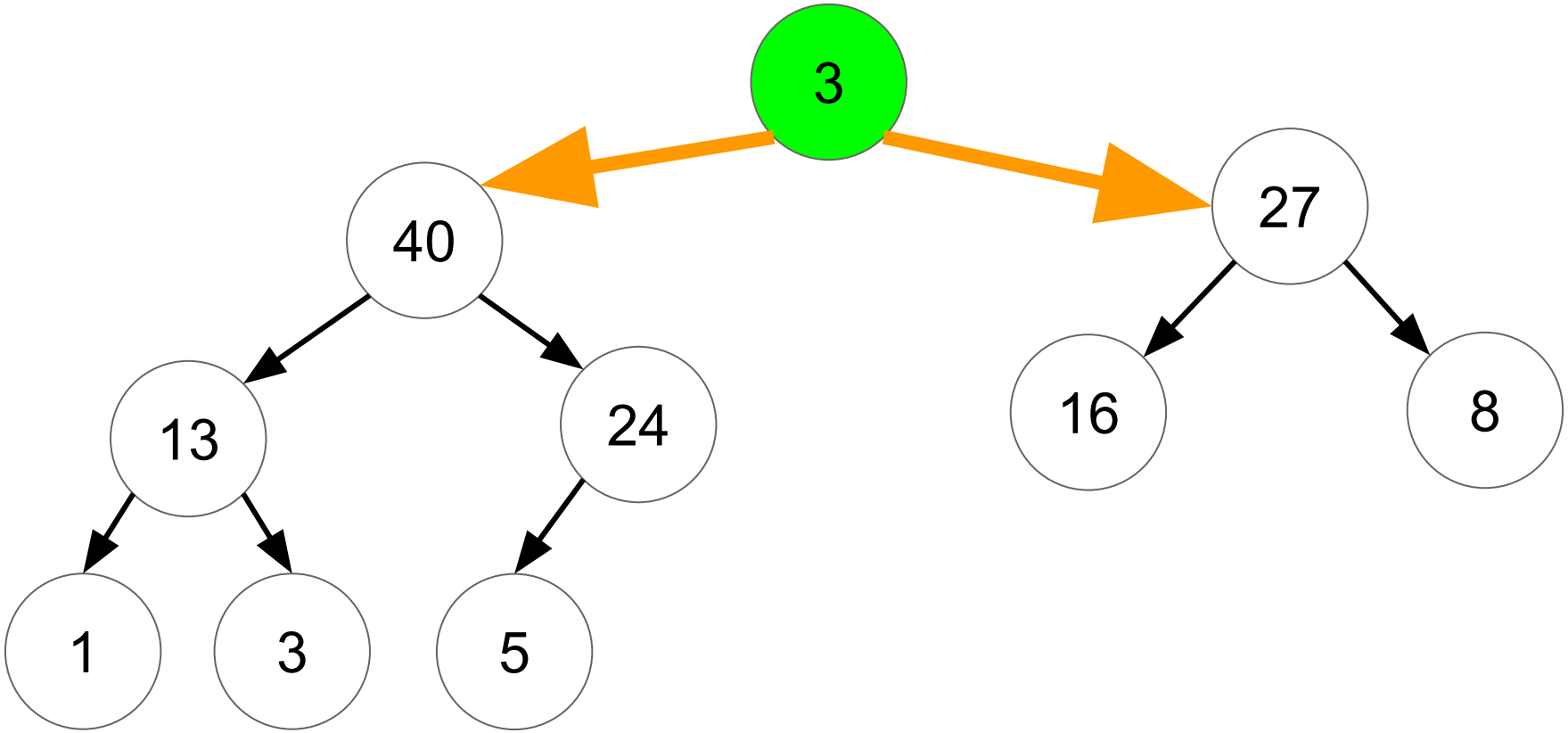
Heapify Example: Fix Down From Bottom

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



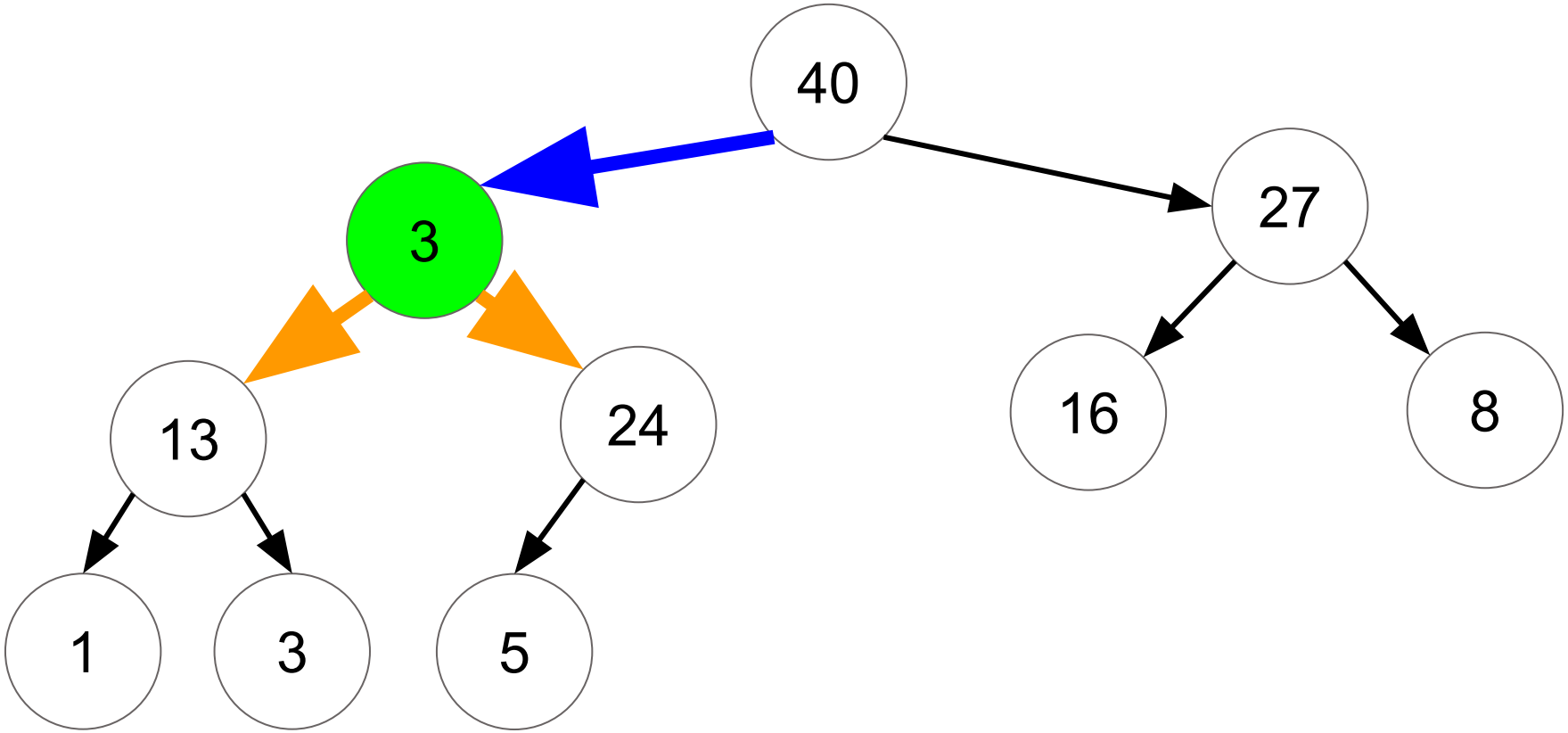
Heapify Example: Fix Down From Bottom

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



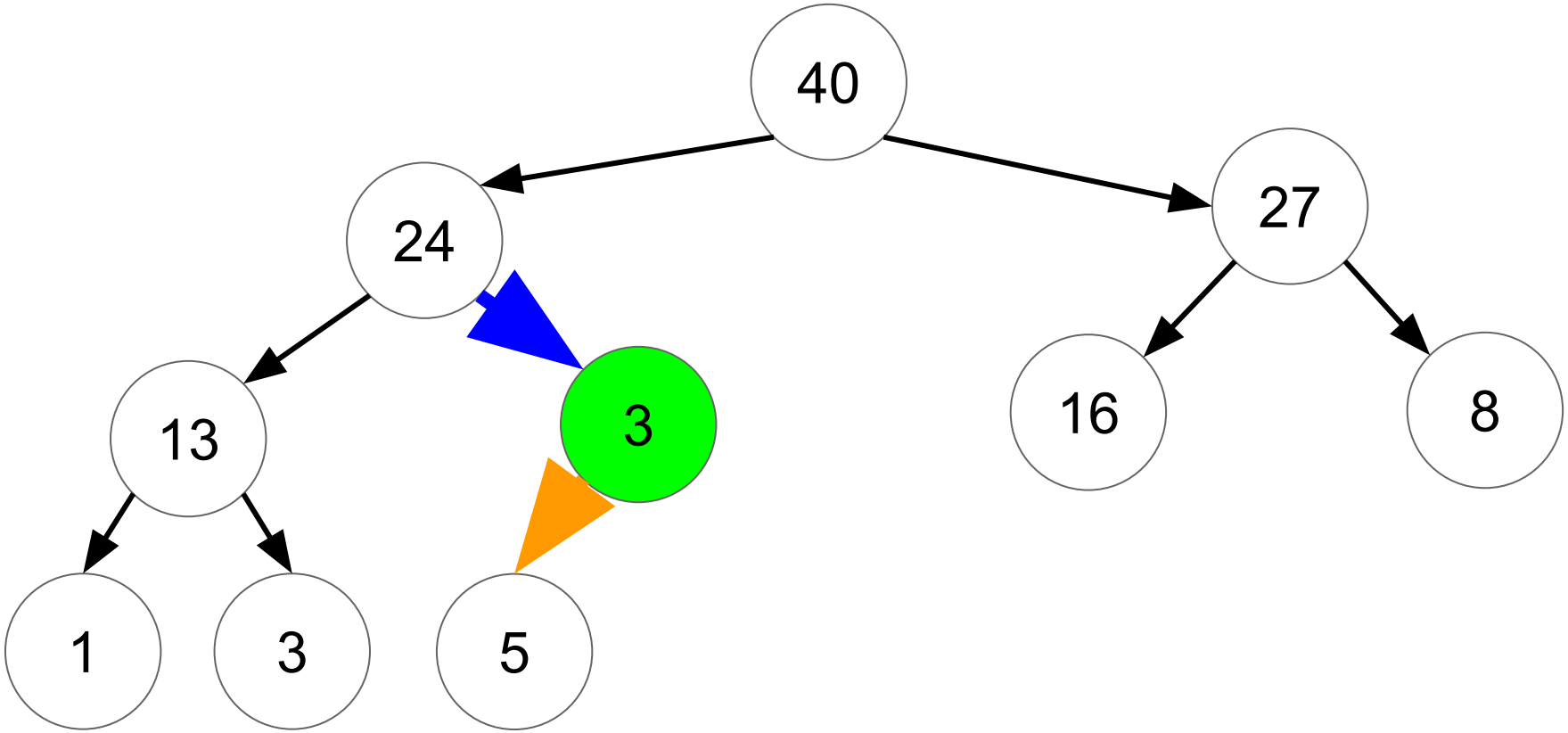
Heapify Example: Fix Down From Bottom

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



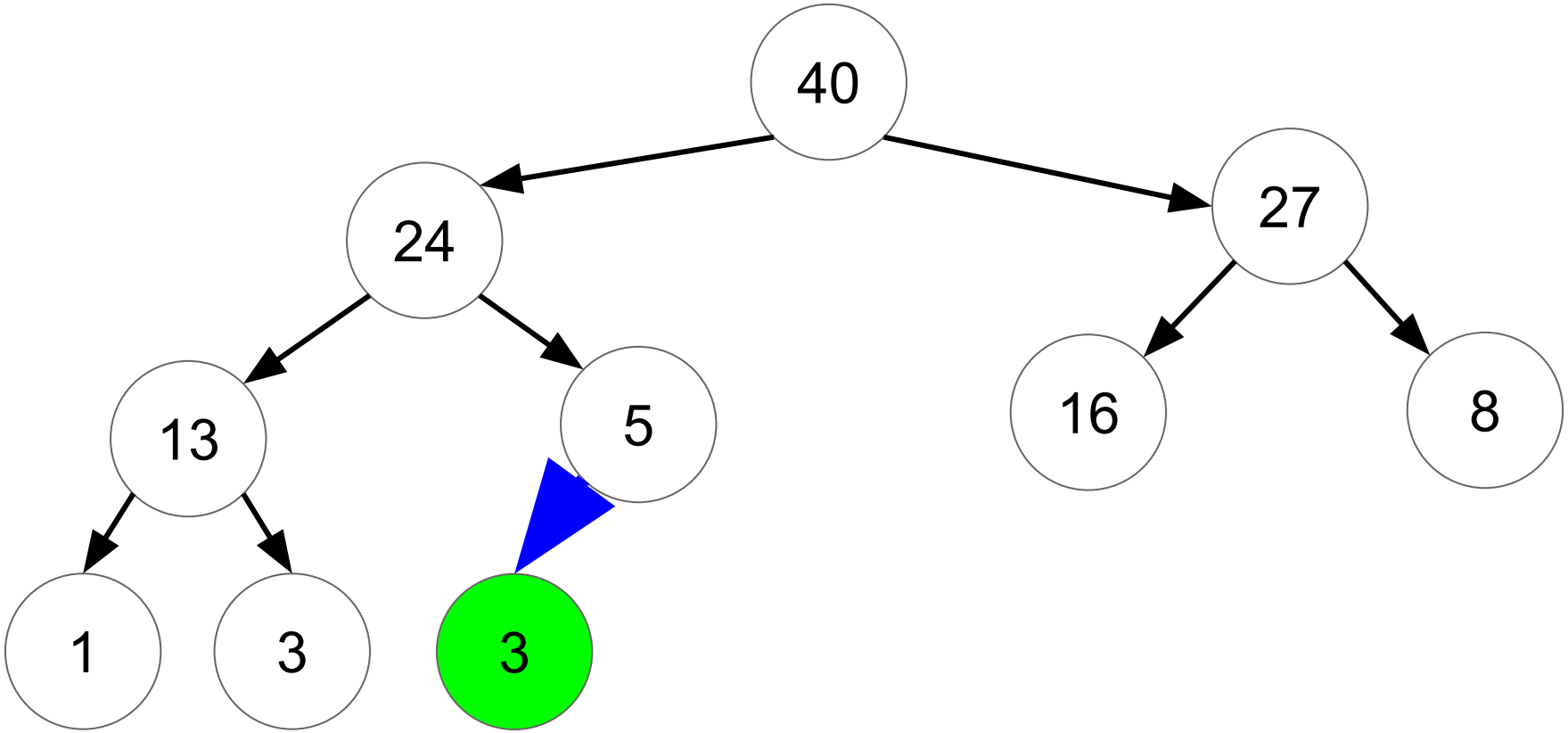
Heapify Example: Fix Down From Bottom

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



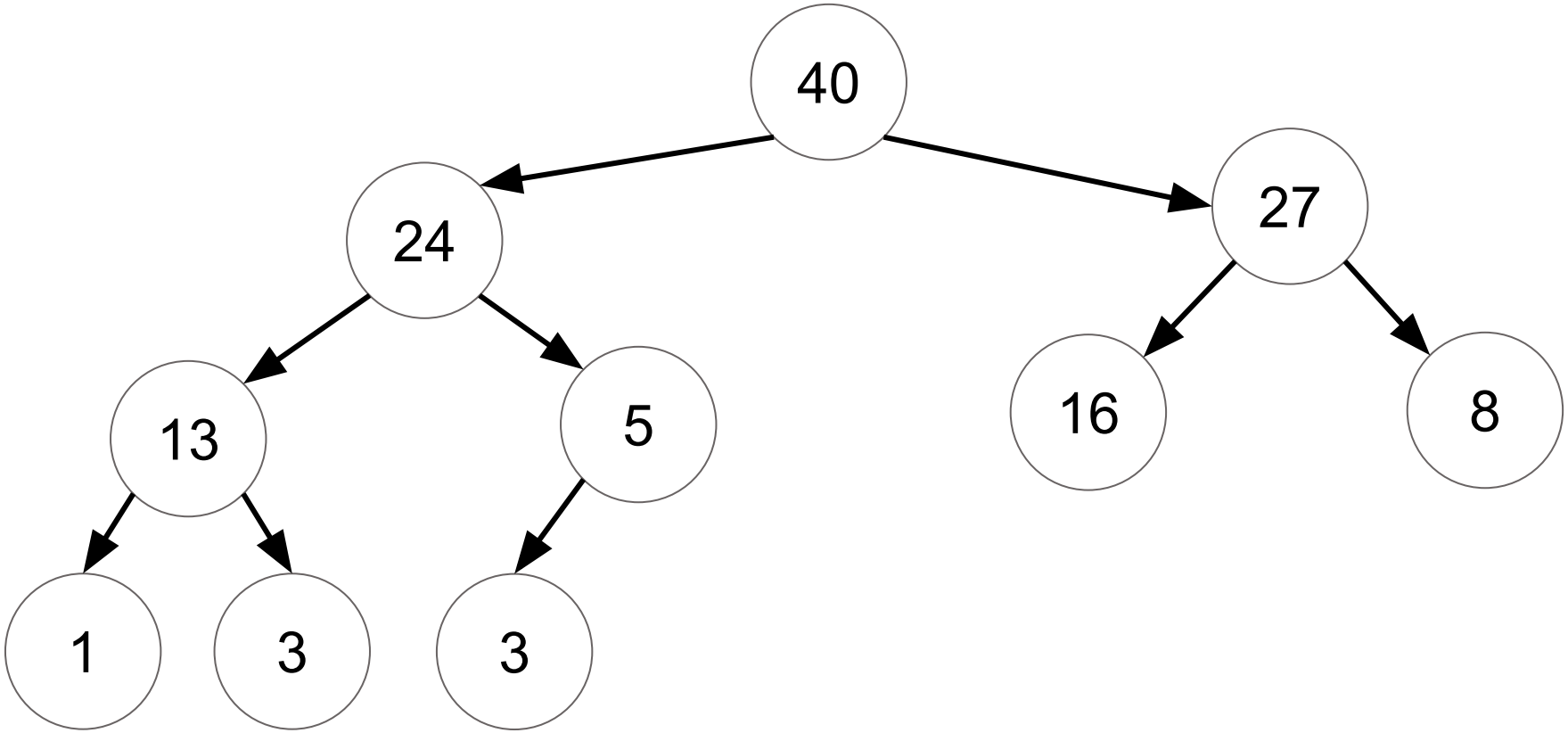
Heapify Example: Fix Down From Bottom

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



Heapify Example: Fix Down From Bottom

- Fix this heap by repeatedly 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 greatest number of items
  - 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

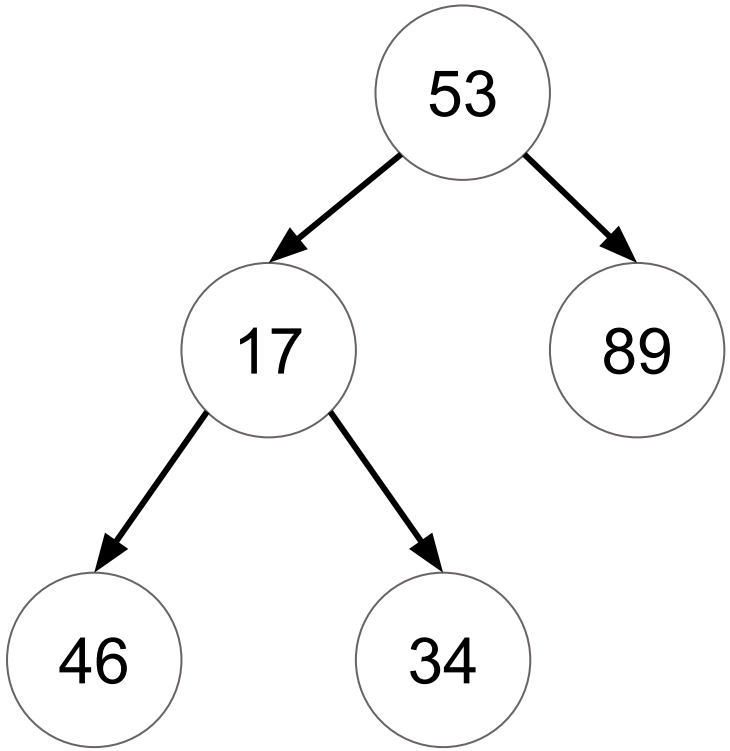
- Uses a binary heap to sort items:
  - identify largest element, move it to the end, find next largest element, move it to the end, repeat...
  - Build a heap using make heap / heapify
  - Remove items from the heap one at a time to get them in sorted order

- 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]



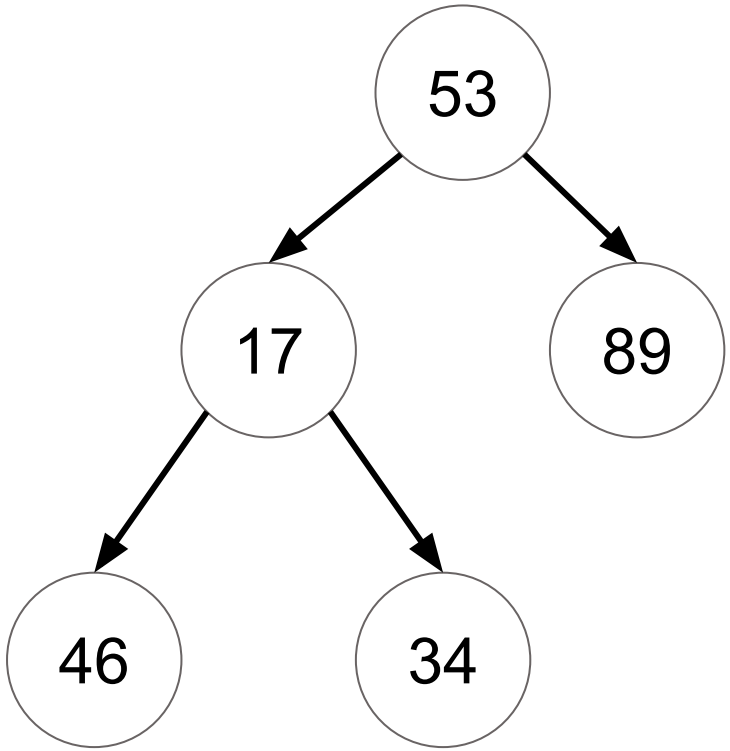
Heapsort

- 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

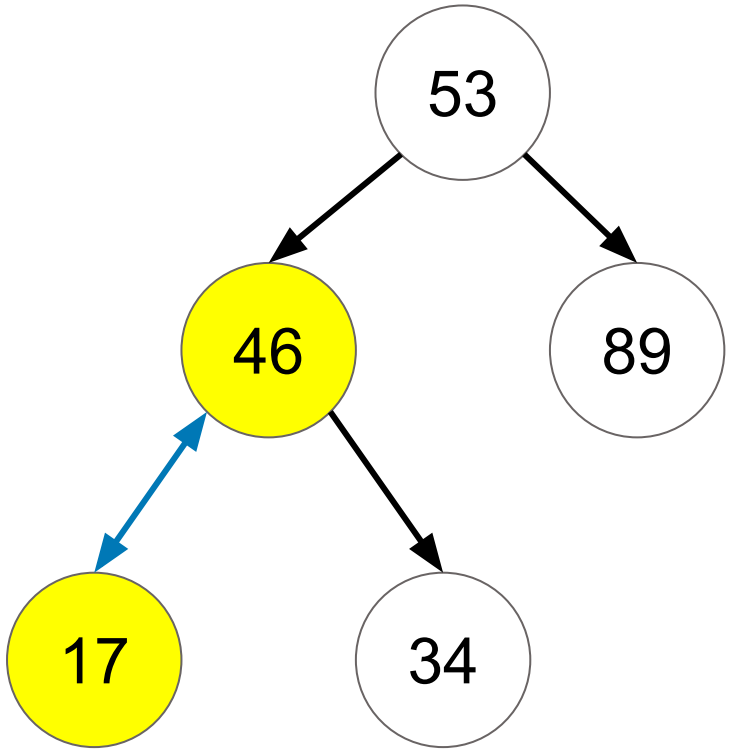
Heapsort

- 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

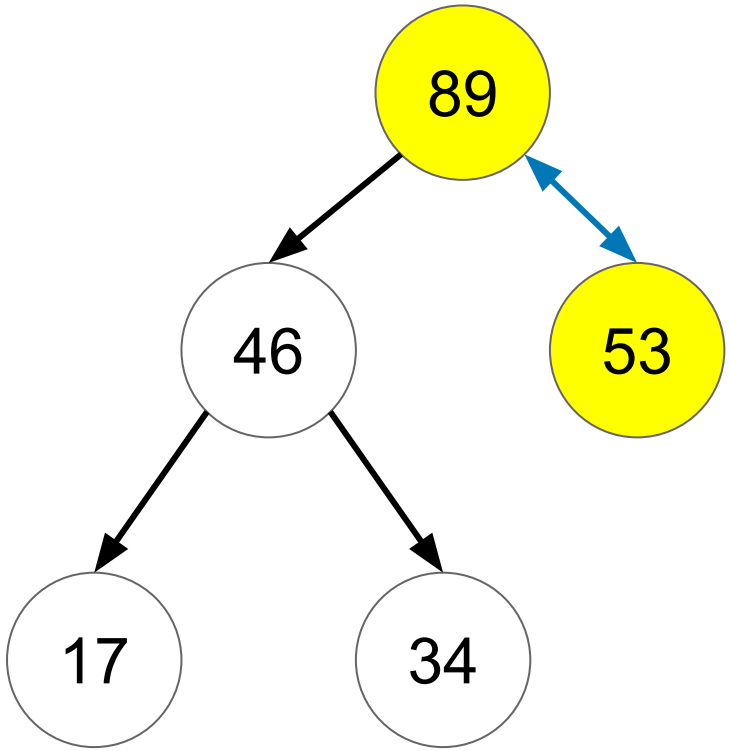
Heapsort

- 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

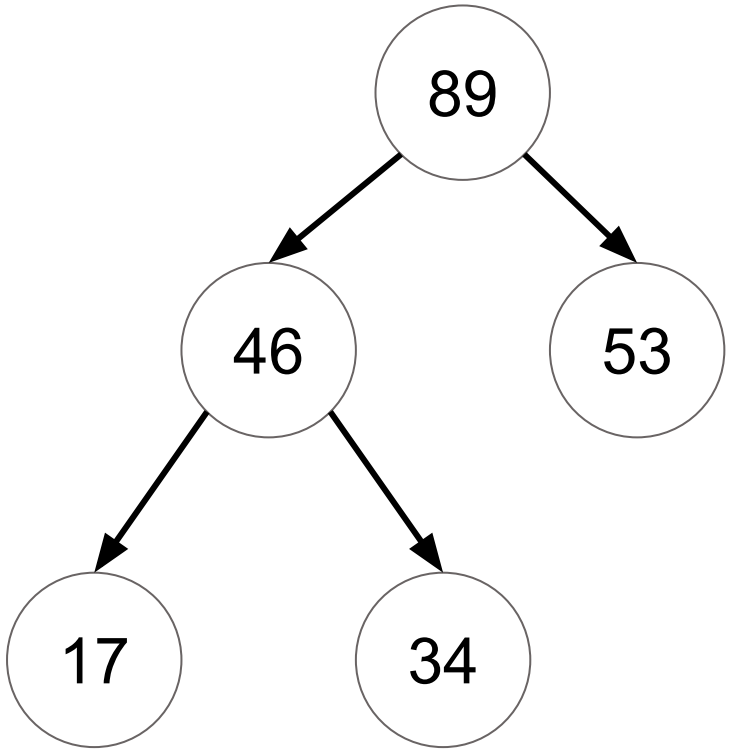
Heapsort

- 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

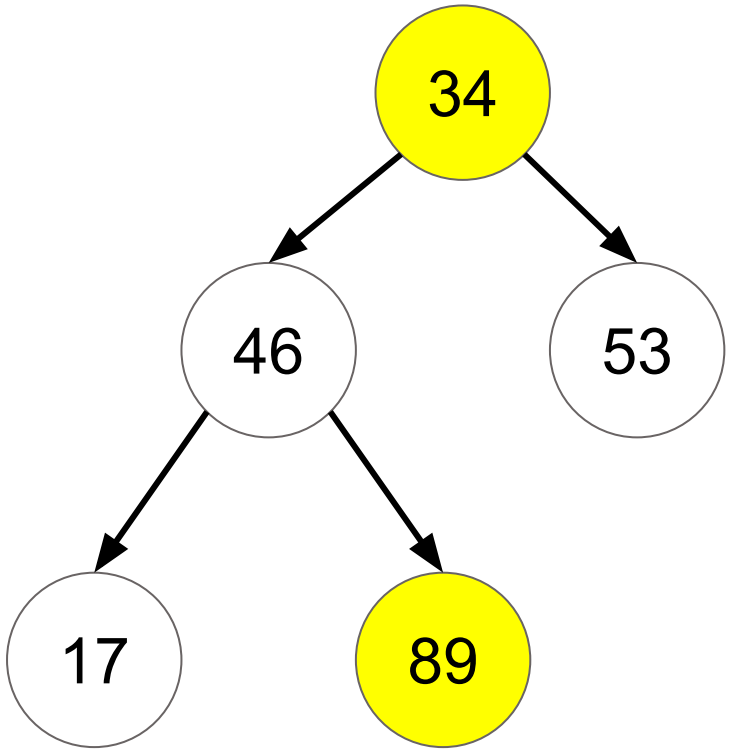
Heapsort

- 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

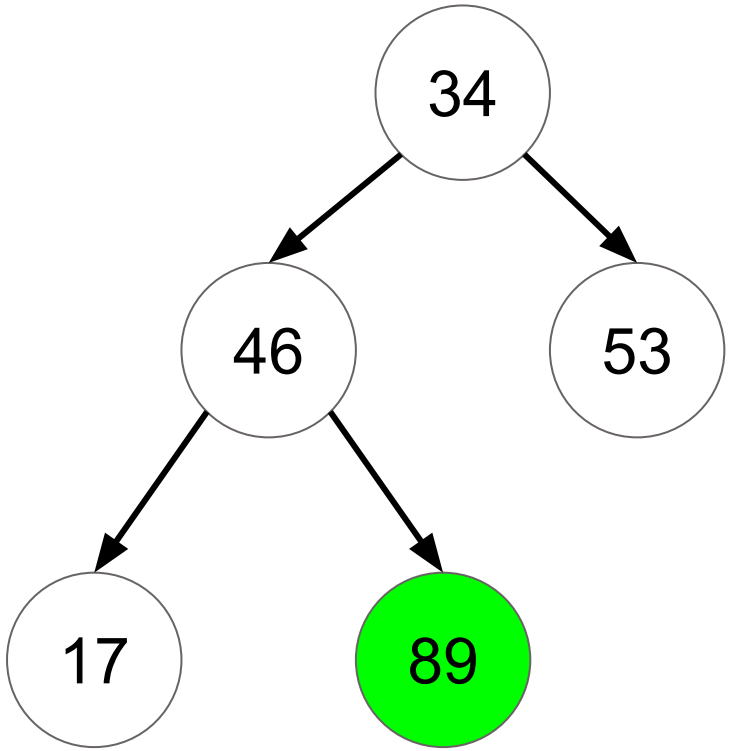
34	46	53	17	89

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

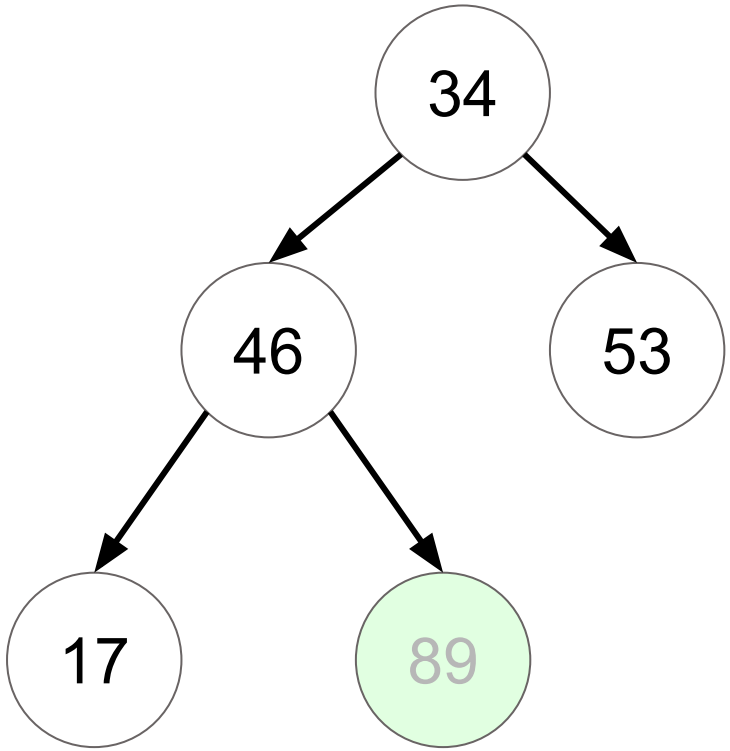
34	46	53	17	89

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

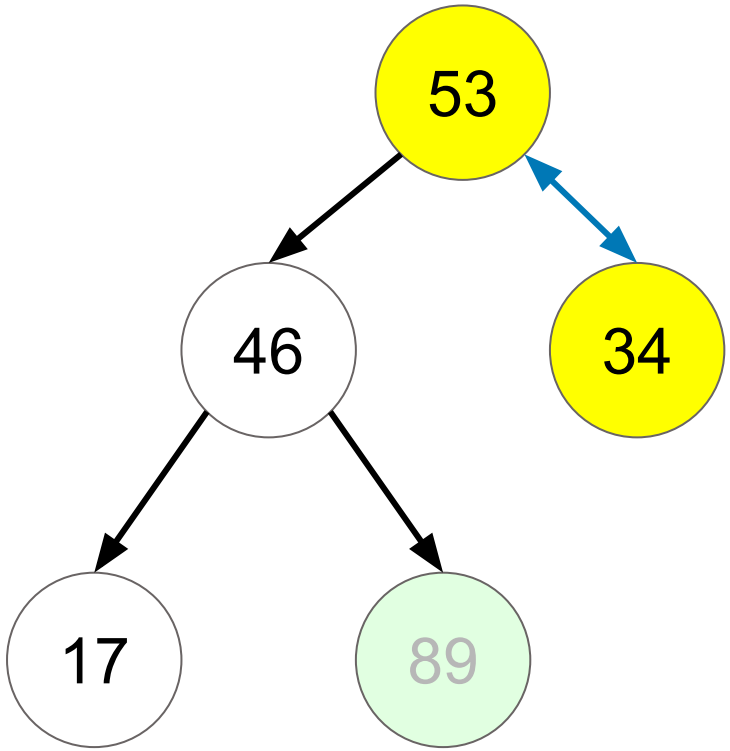
34	46	53	17	89

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

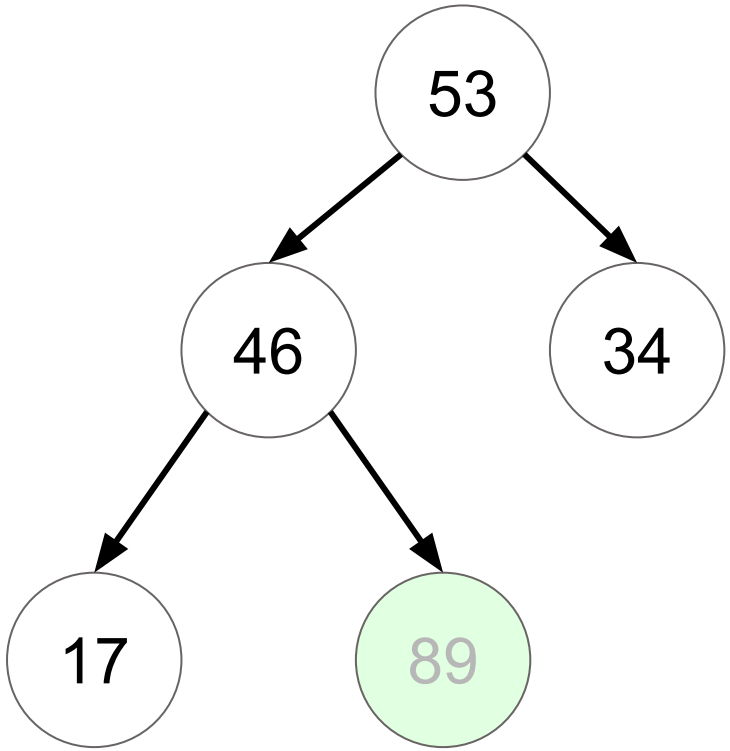
53	46	34	17	89

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

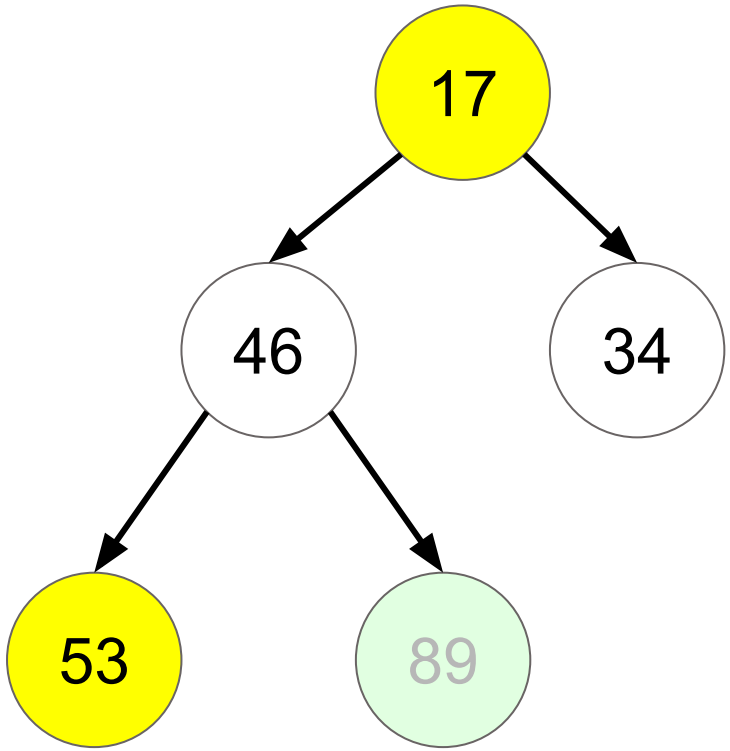
53	46	34	17	89

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

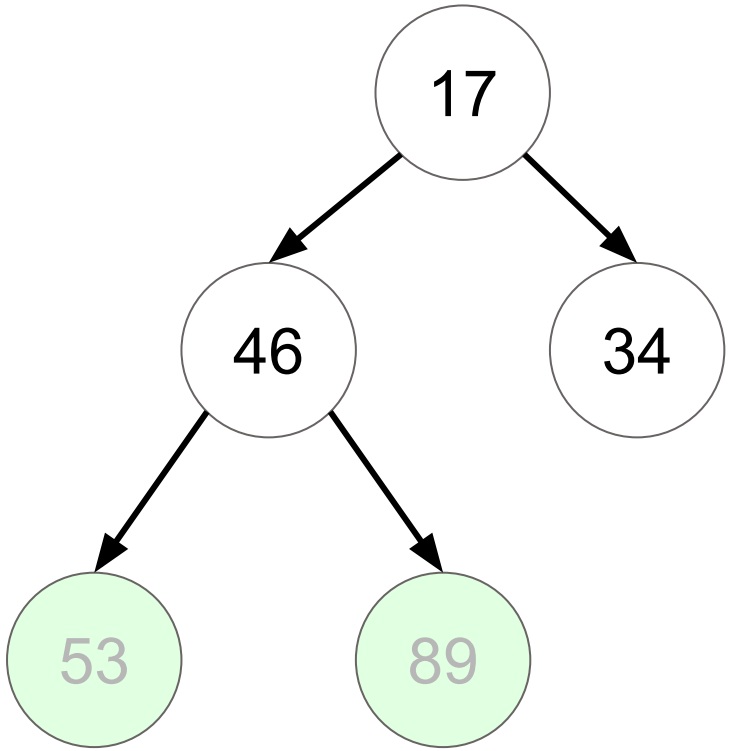
17	46	34	53	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Repeat until the array is completely sorted!



Array Representation

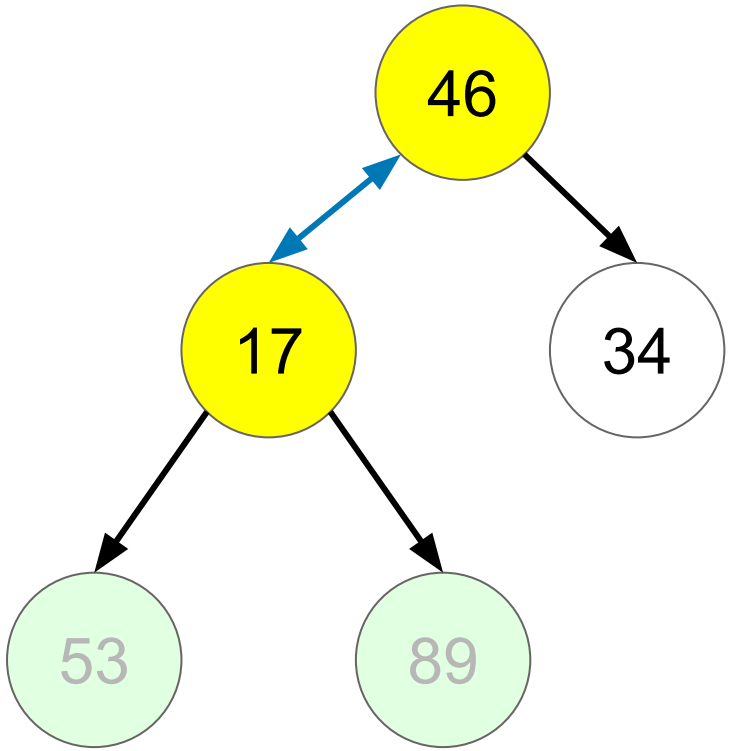
17	46	34	53	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Repeat until the array is completely sorted!



Array Representation

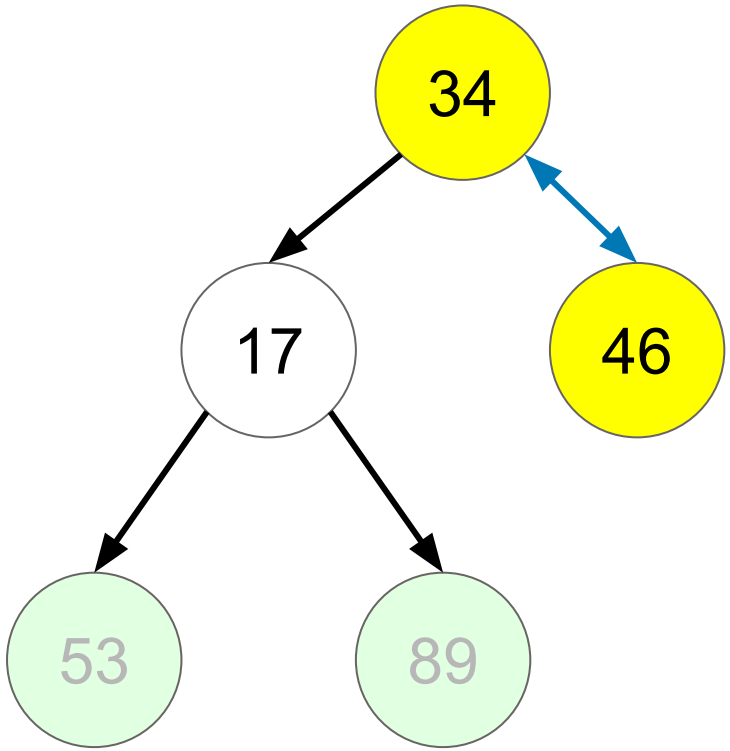
46	17	34	53	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Repeat until the array is completely sorted!



Array Representation

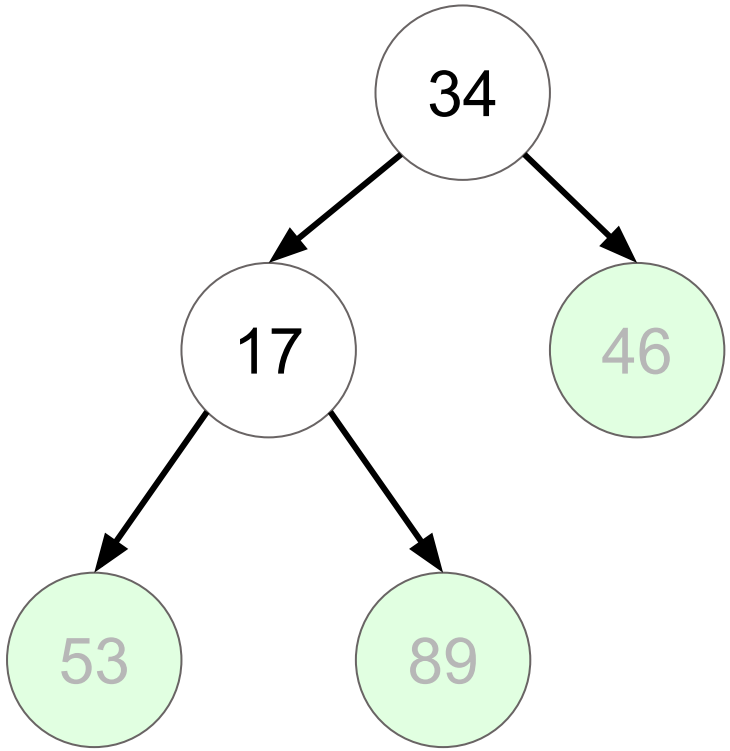
34	17	46	53	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Repeat until the array is completely sorted!



Array Representation

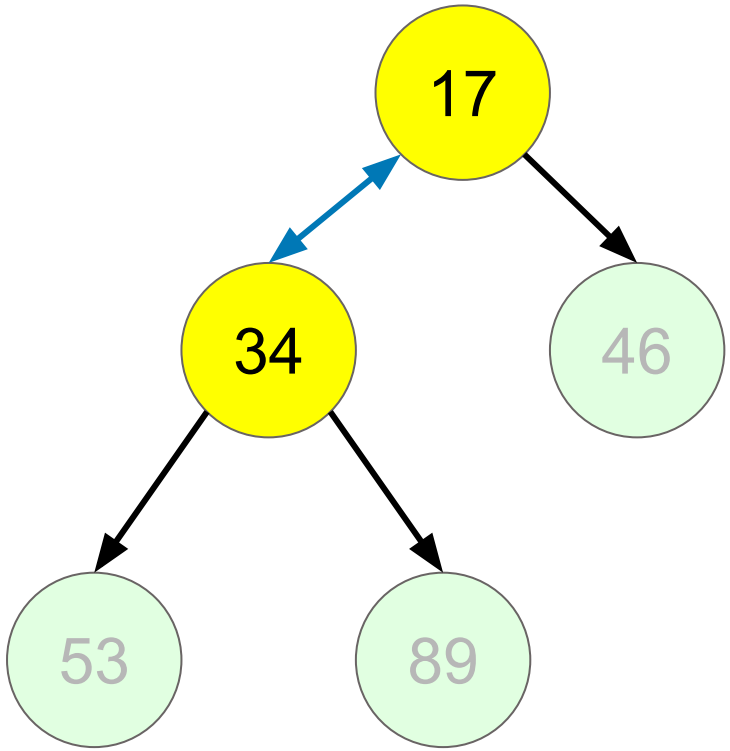
34	17	46	53	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Repeat until the array is completely sorted!



Array Representation

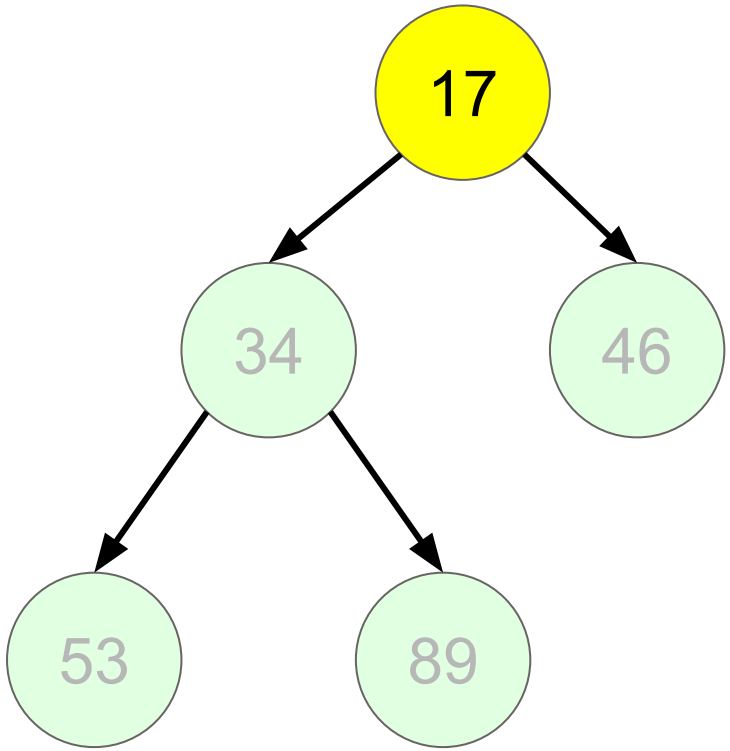
17	34	46	53	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Repeat until the array is completely sorted!



Array Representation

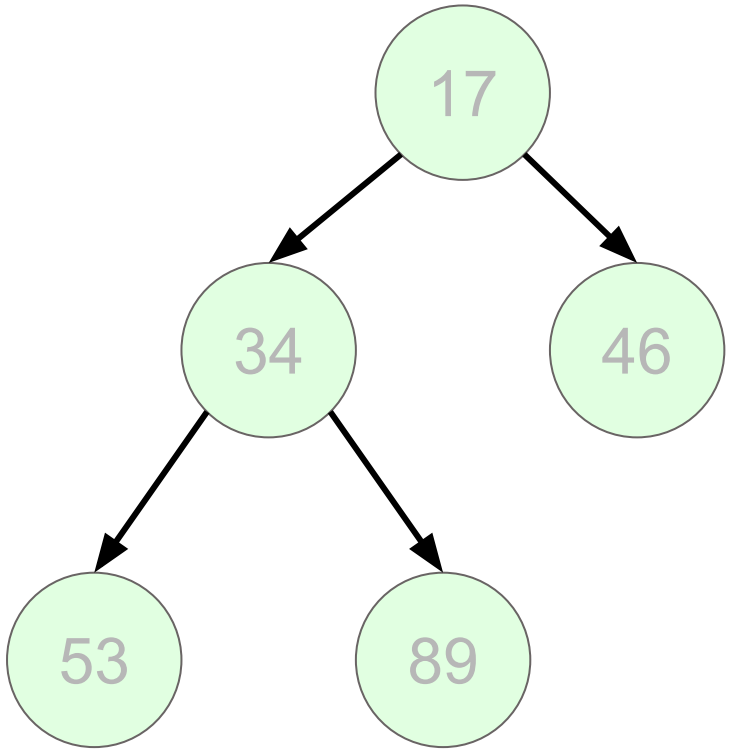
17	34	46	53	89

Heapsort

- Example: heapsort the following array

[53, 17, 89, 46, 34]

Repeat until the array is completely sorted!



Array Representation

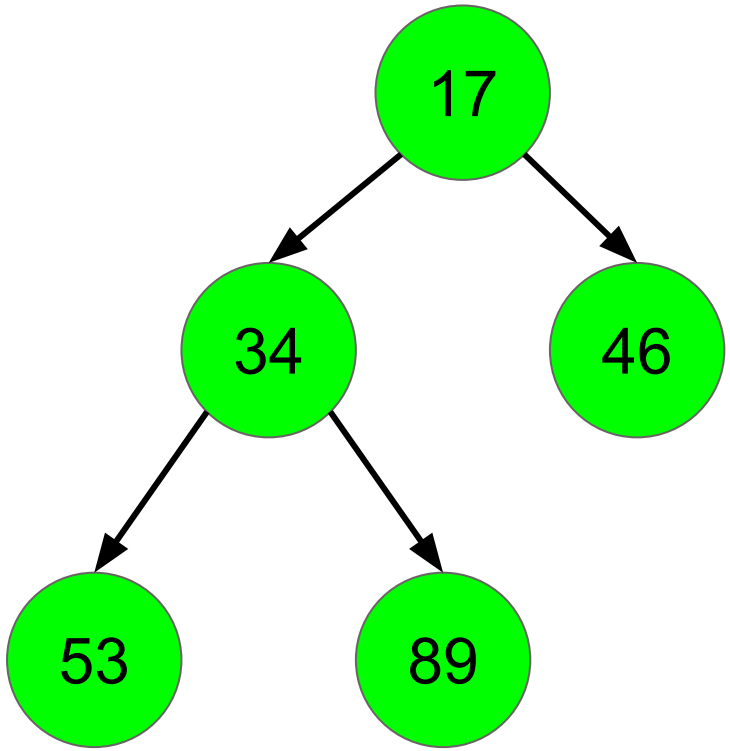
17	34	46	53	89

Heapsort

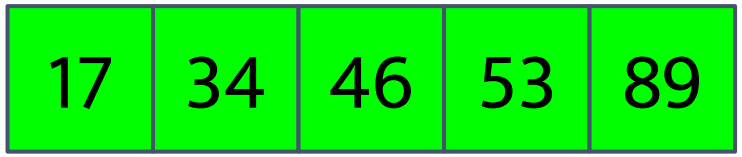
- Example: heapsort the following array

[53, 17, 89, 46, 34]

Repeat until the array is completely sorted!



Array Representation



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
  - ∅: 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 ∪ B?

    {1, 2, 3, 5}

  - What is A ∩ B?

    {1, 3}

  - What is A – B?

    {2}

Set Operations

- A sorted vector is a good implementation for a set that doesn't need to insert or erase elements often. Use binary search to check membership!
- For a set that needs to insert or erase elements often, we'll see effective implementations later in the class. We'll see another data structure soon that can perform some useful set operations in nearly O(1) time.
- **Practice**: How would you implement the following operations on two sorted input vectors and one output vector?
- Union ($A \cup B$)
- Intersection ($A \cap B$)
- 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

- Other terminology:
  - Symmetric Difference ($A \ominus B$)
    - set of all objects that are in either A or B, but not both
  - Cartesian Product ($A \times B$)
    - set whose members are all possible ordered pairs (a, b), where a is a member of A and b is a member of B
  - Power Set (P(A))
    - set whose members are all possible subsets of A
- Practice:
  - what is the cardinality (number of elements in the set) of $A \times B$?
  - what is the cardinality of P(A)?

Set Operations

- Other terminology:
  - Symmetric Difference ($A \ominus B$)
    - set of all objects that are in either A or B, but not both
  - Cartesian Product ($A \times B$)
    - set whose members are all possible ordered pairs (a, b), where a is a member of A and b is a member of B
  - Power Set (P(A))
    - set whose members are all possible subsets of A
- Practice:
  - what is the cardinality (number of elements in the set) of $A \times B$? $|A| \times |B|$
  - what is the cardinality of P(A)? $2^{|A|}$

Set Operations

- Let A = {1, 2, 3} and B = {1, 3, 5}
  - What is $A \ominus B$?
  - What is $A \times B$?
  - What is P(A)?

Set Operations

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

    {2, 5}

  - What is A ✕ B?

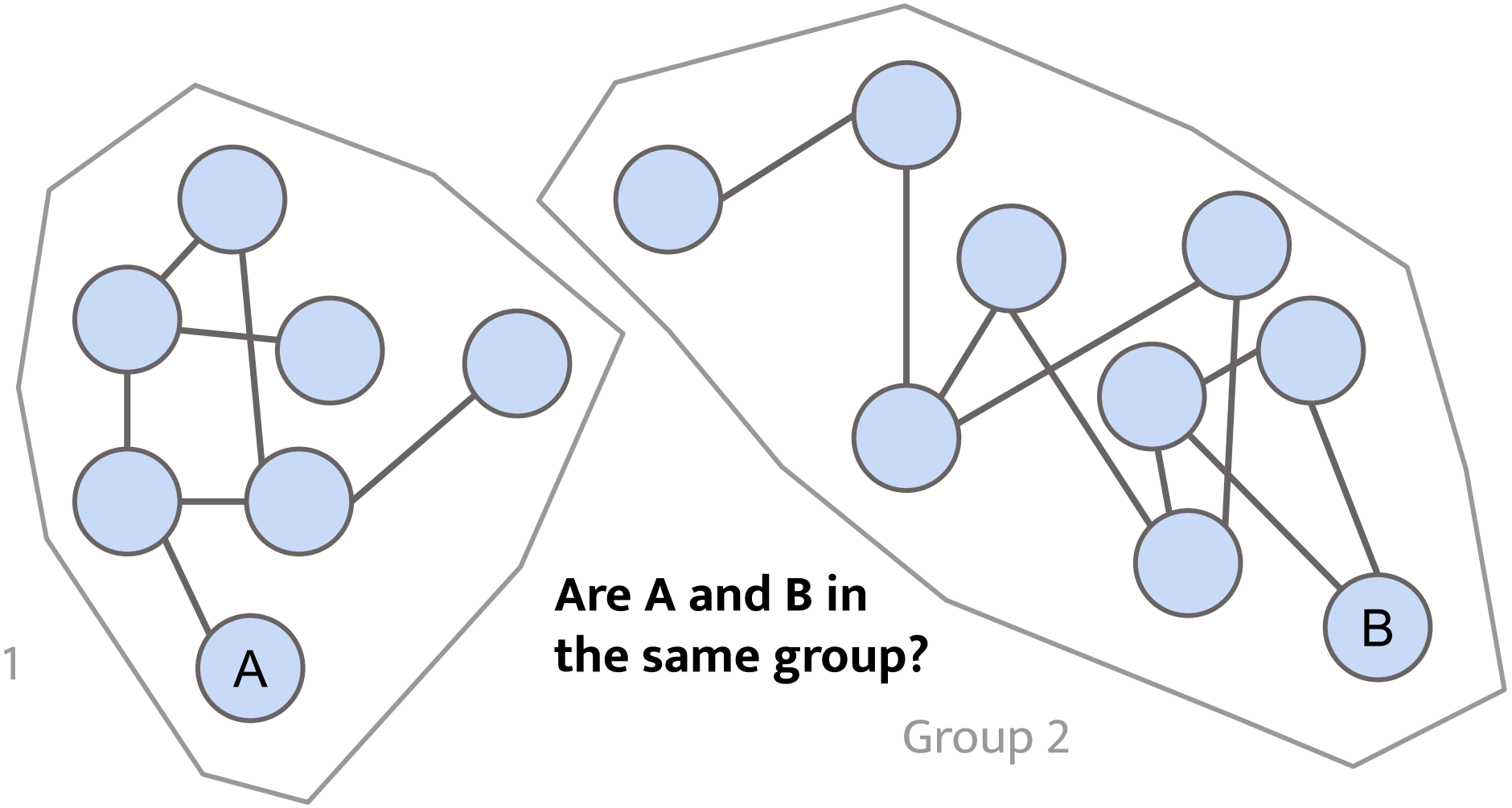
    {{1, 1}, {1, 3}, {1, 5}, {2, 1}, {2, 3}, {2, 5}, {3, 1}, {3, 3}, {3, 5}}

  - What is P(A)?

    {∅, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}}

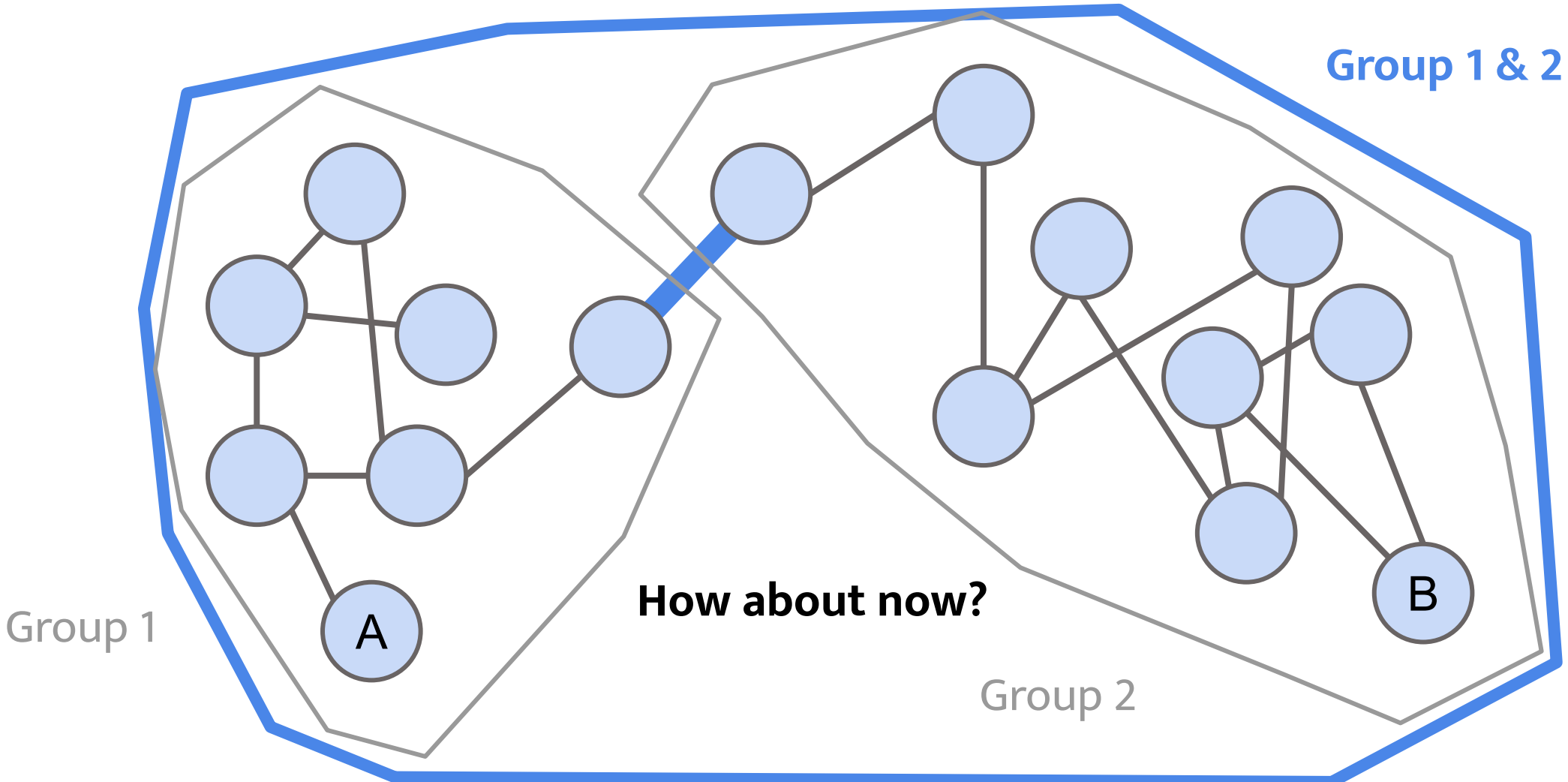
Set Operations

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



Set Operations

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

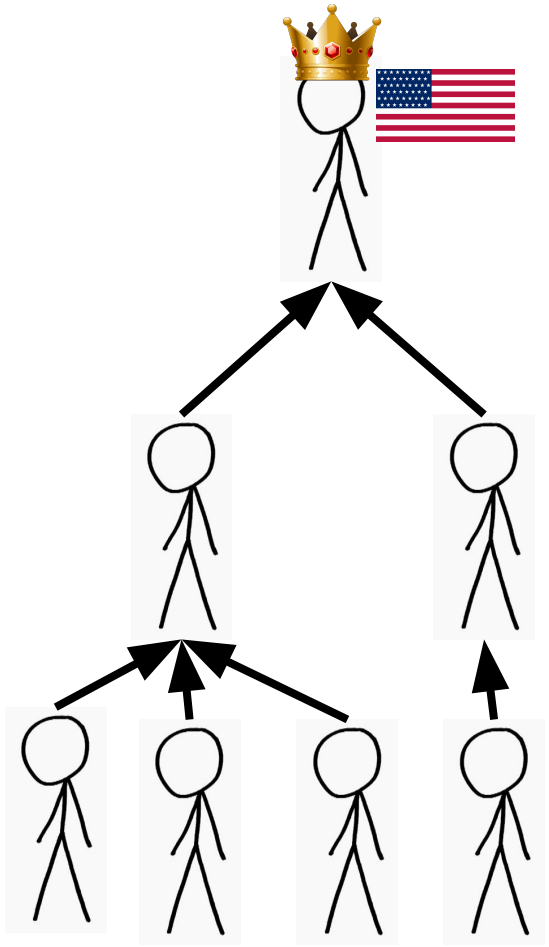


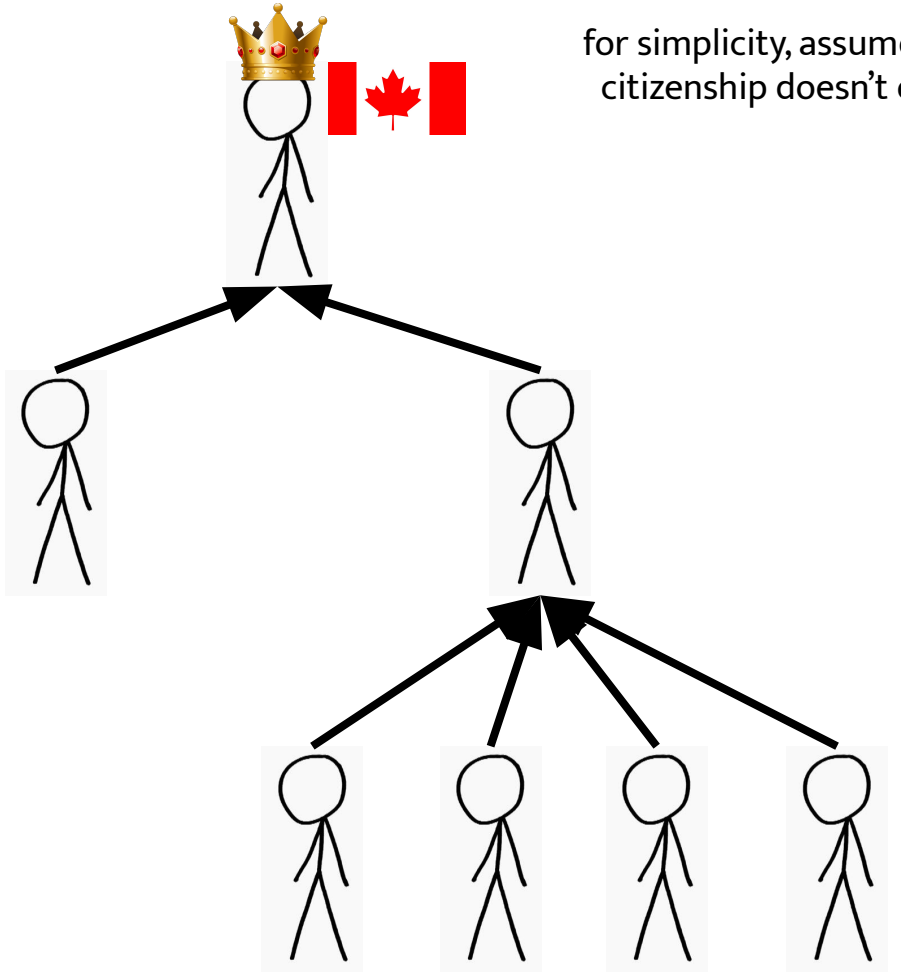
Union Find

- 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.
- Traditional sets won't work for this task - we want to quickly check whether **known** elements are in the same set, not check whether **any** element is part of a single set.
- Implemented using an array of integers and two functions - **union** and **find**
- 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
- Possible implementation:
  - Hold representative for each item x - but then we have to scan through the entire vector to update representatives upon a union, which we want to avoid.
  - Can we encode a union while only updating one representative?

Union Find

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

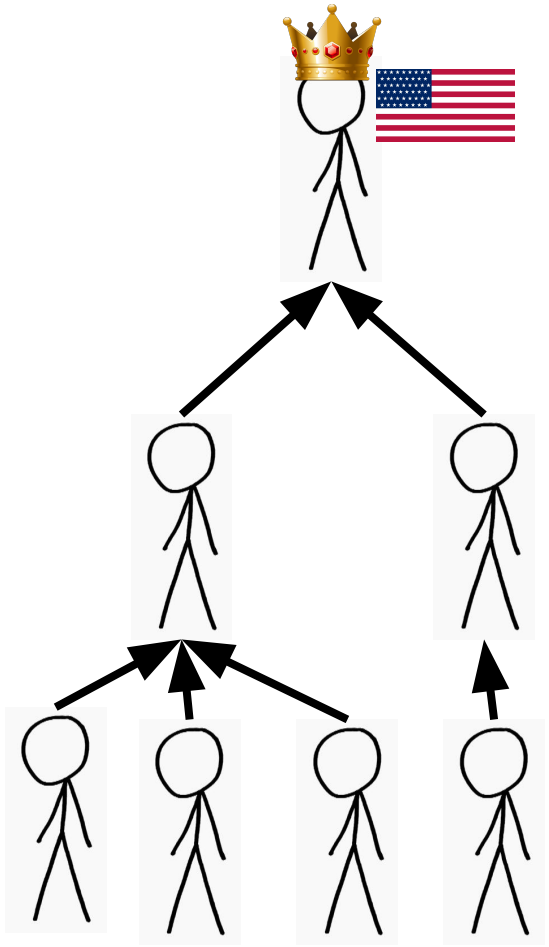




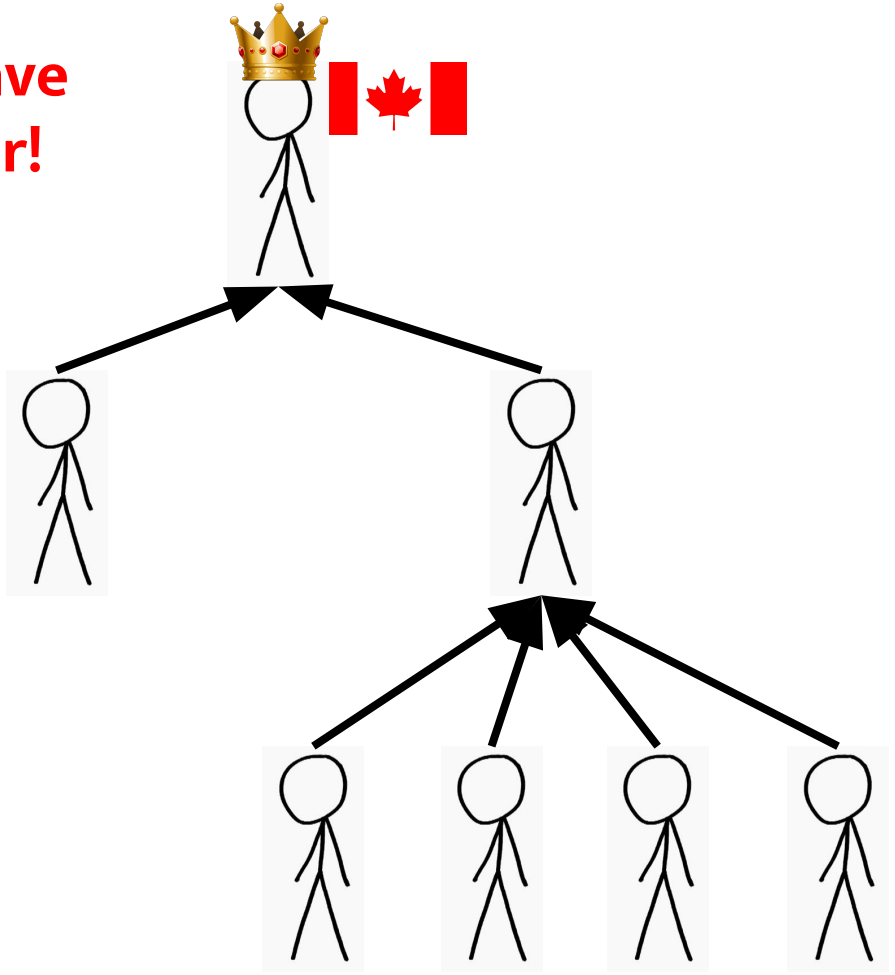
for simplicity, assume that dual citizenship doesn't exist here

Union Find

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

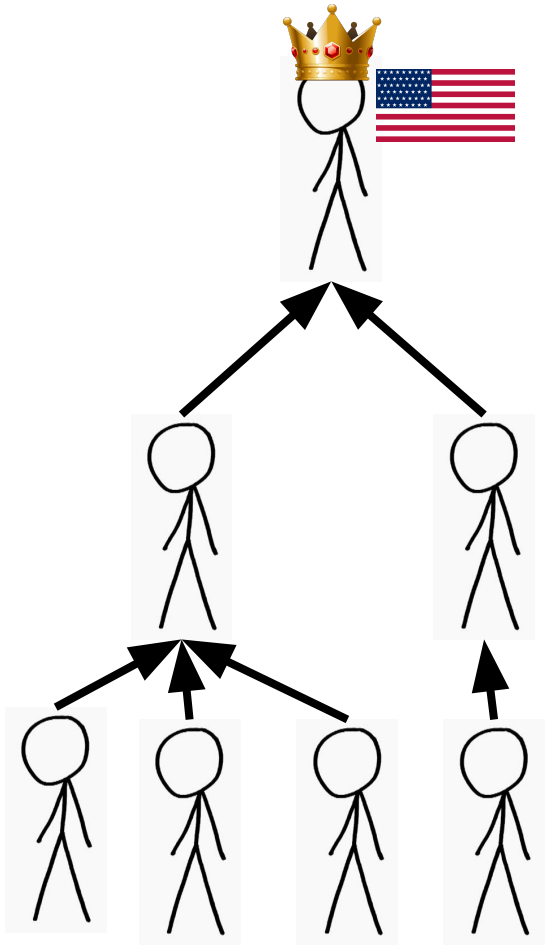


Check if they have the same leader!



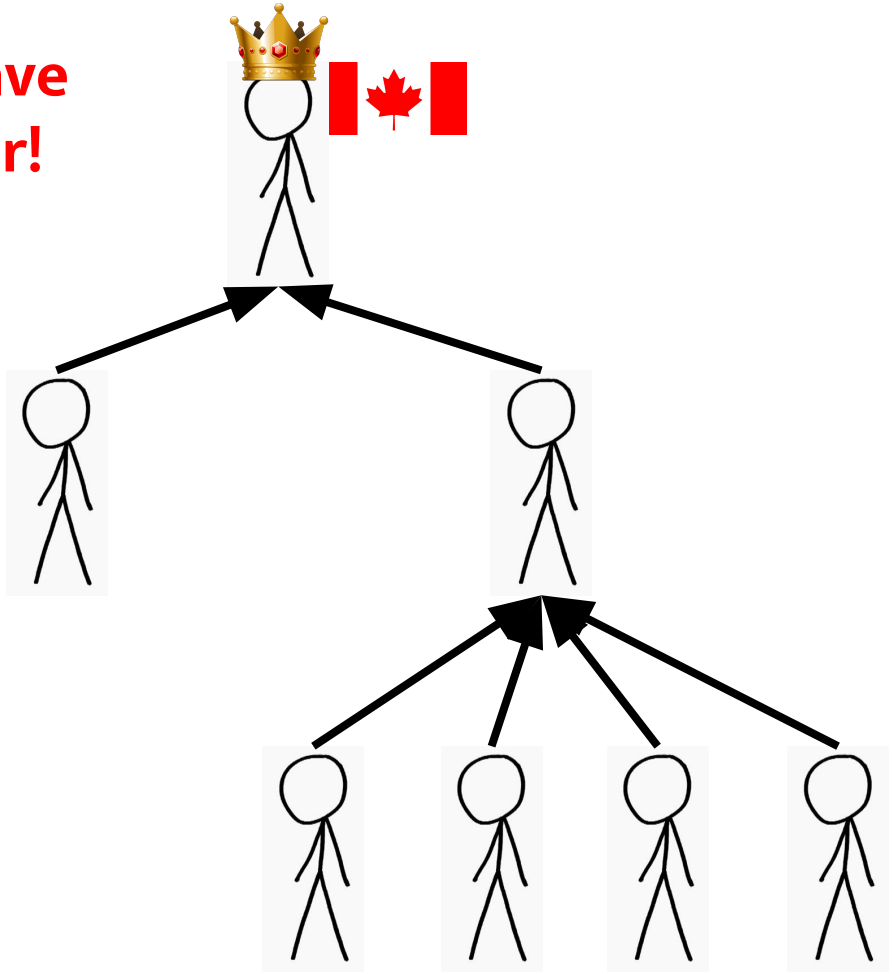
Union Find

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



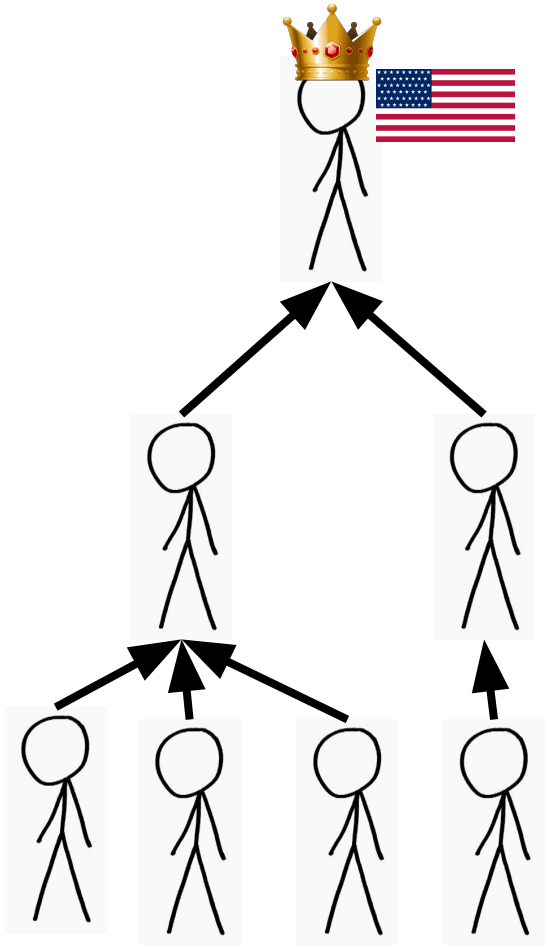
Check if they have the same leader!

How do we find someone's leader?



Union Find

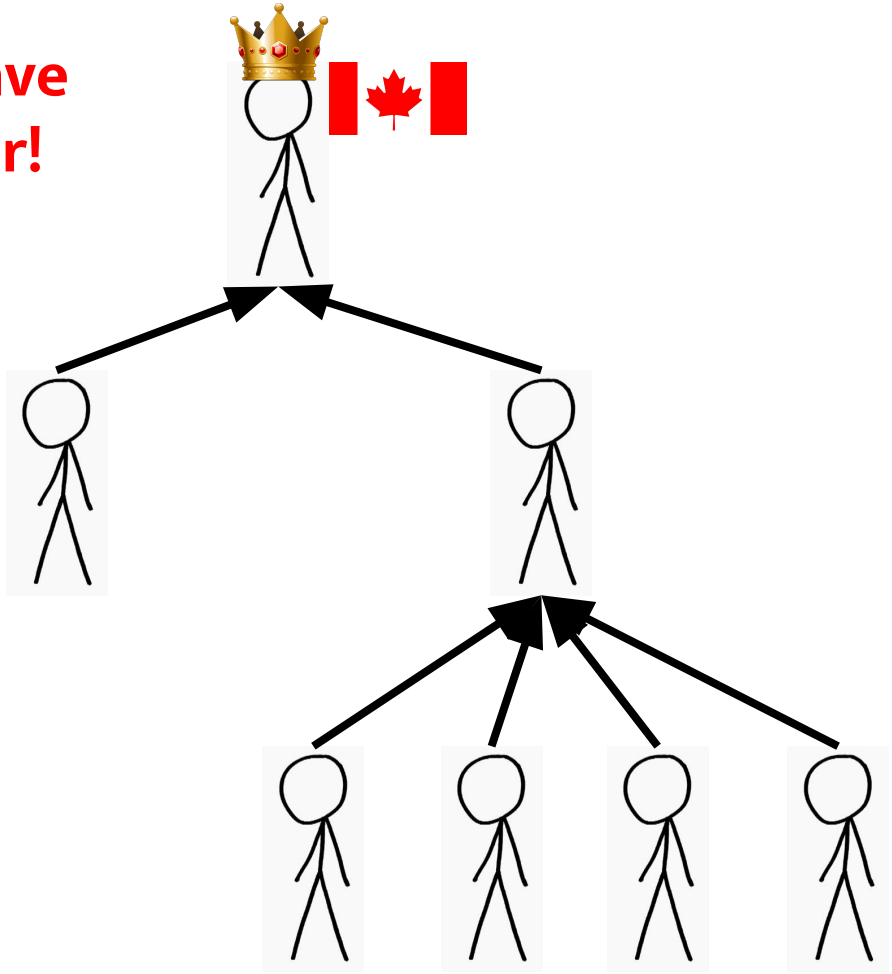
- How can we tell if two people are citizens of the same country?



Check if they have the same leader!

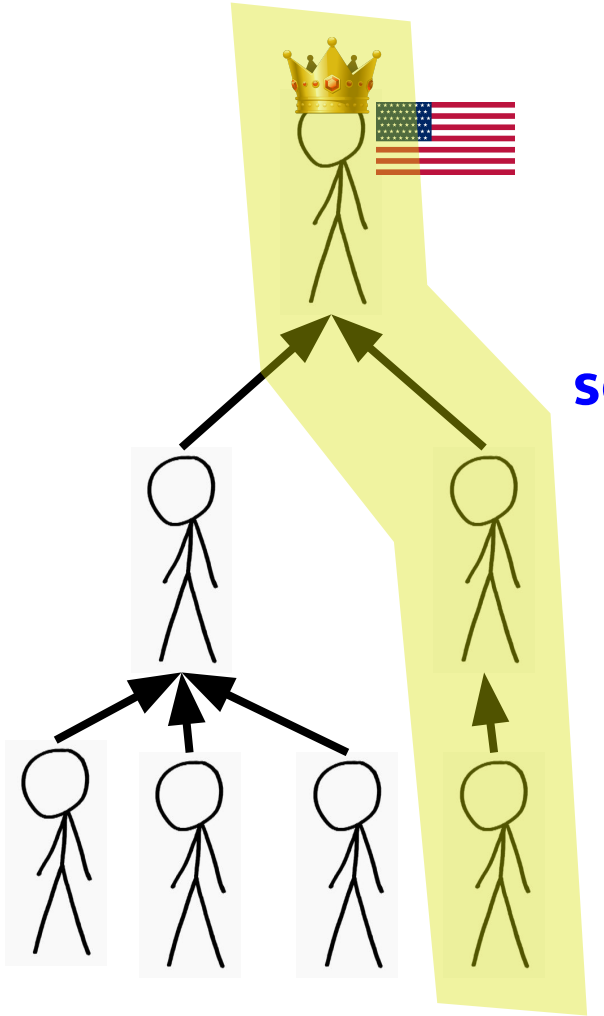
How do we find someone's leader?

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



Union Find

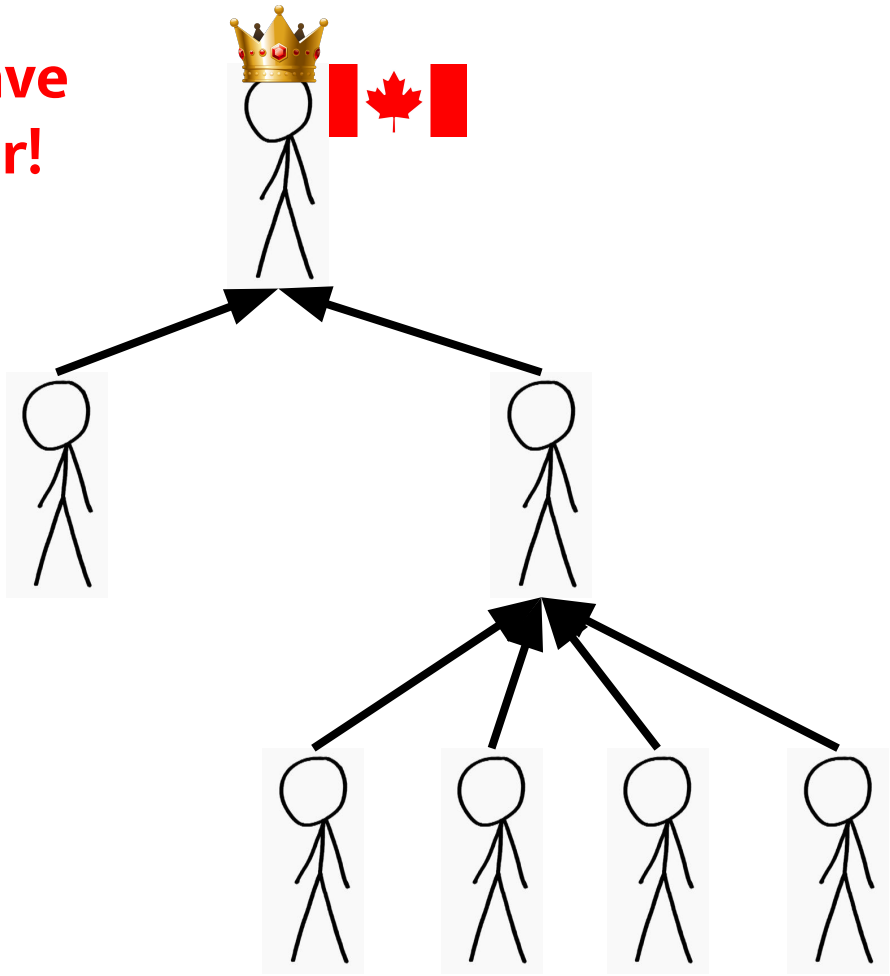
- How can we tell if two people are citizens of the same country?



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!

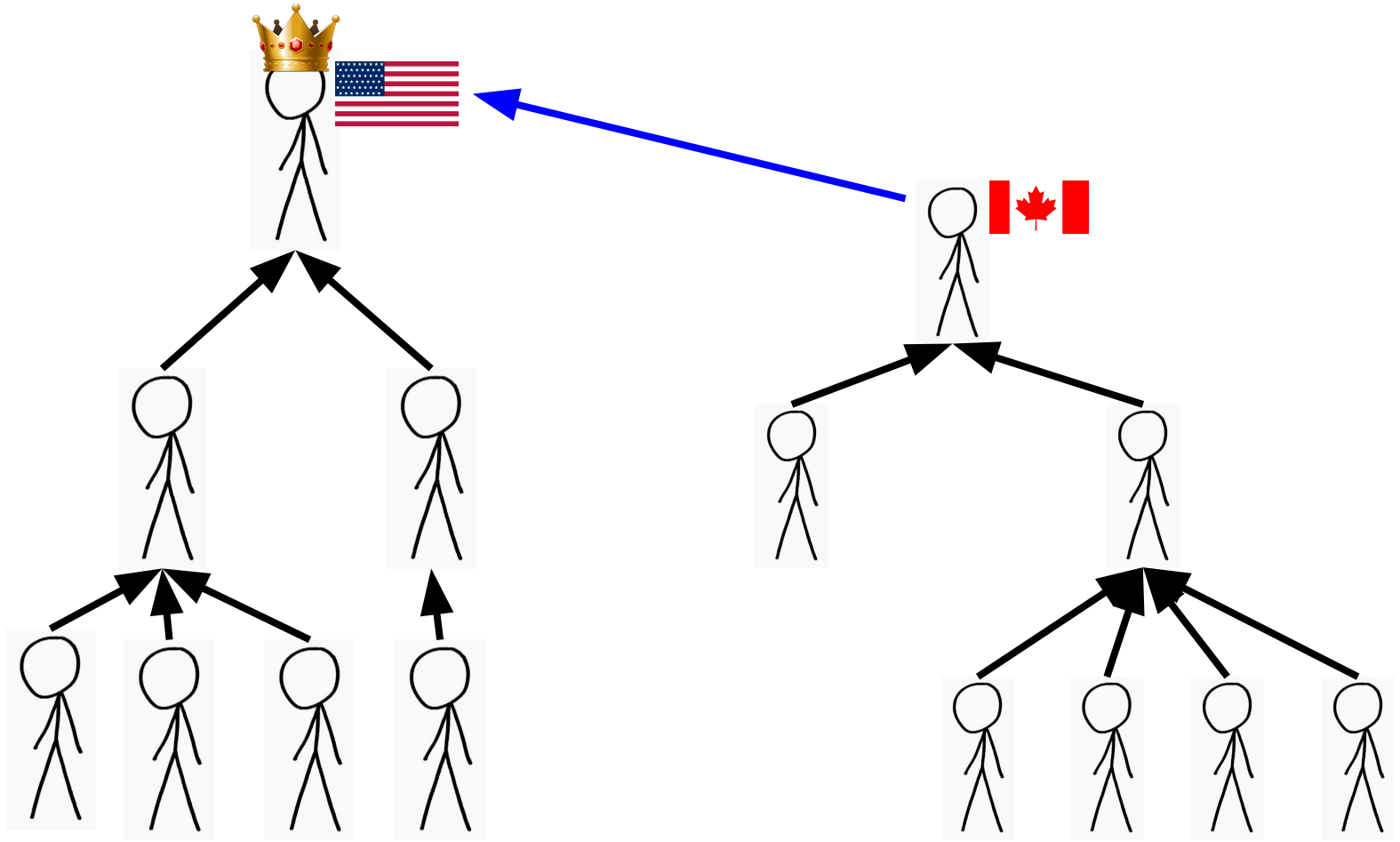


Union Find

- 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: Path Compression

- Problem: suppose we call find on an element multiple times.
  - It takes work to move up the tree to find this element's representative, especially if we have to move up multiple levels!
  - But as long as the representative is the same, all of the elements along our path to the leader will still have the same representative - we don't have to do all this work over again if find is called on an element multiple times.
  - Fix: 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).
- This process is known as **path compression**!

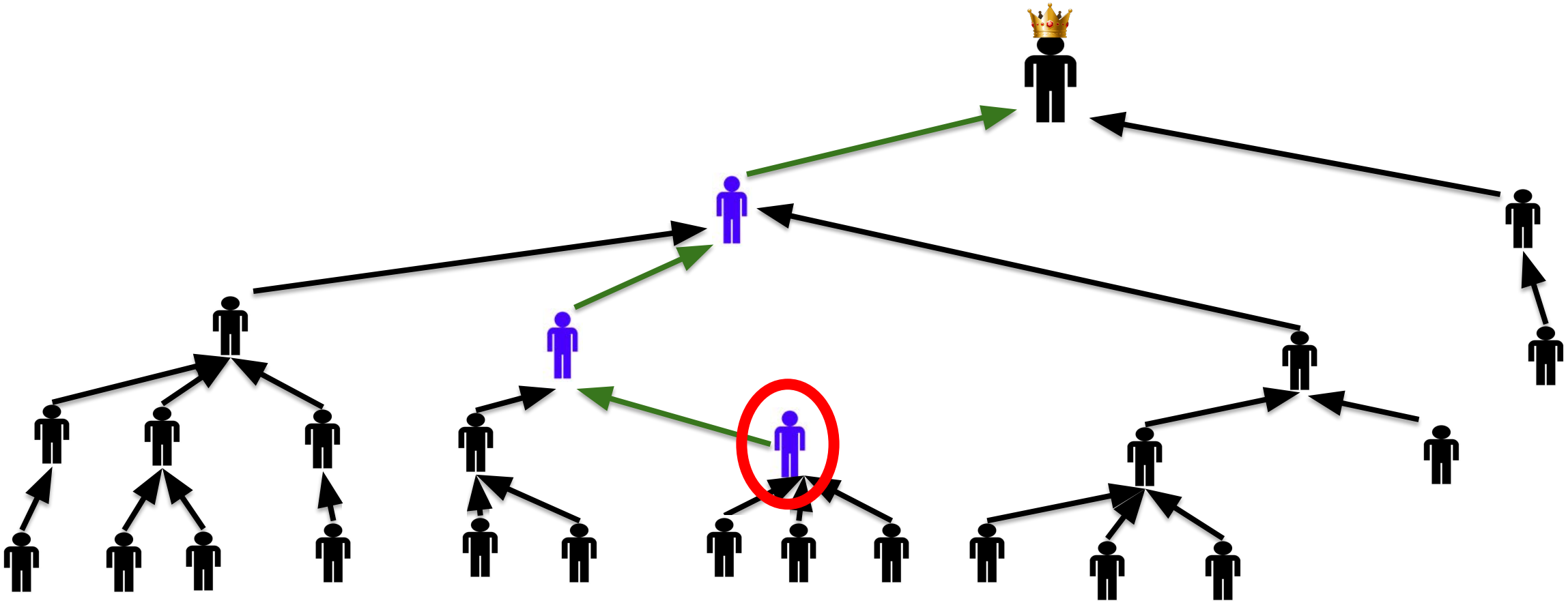
Union Find: Path Compression

- 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).
- This process is known as **path compression**!

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

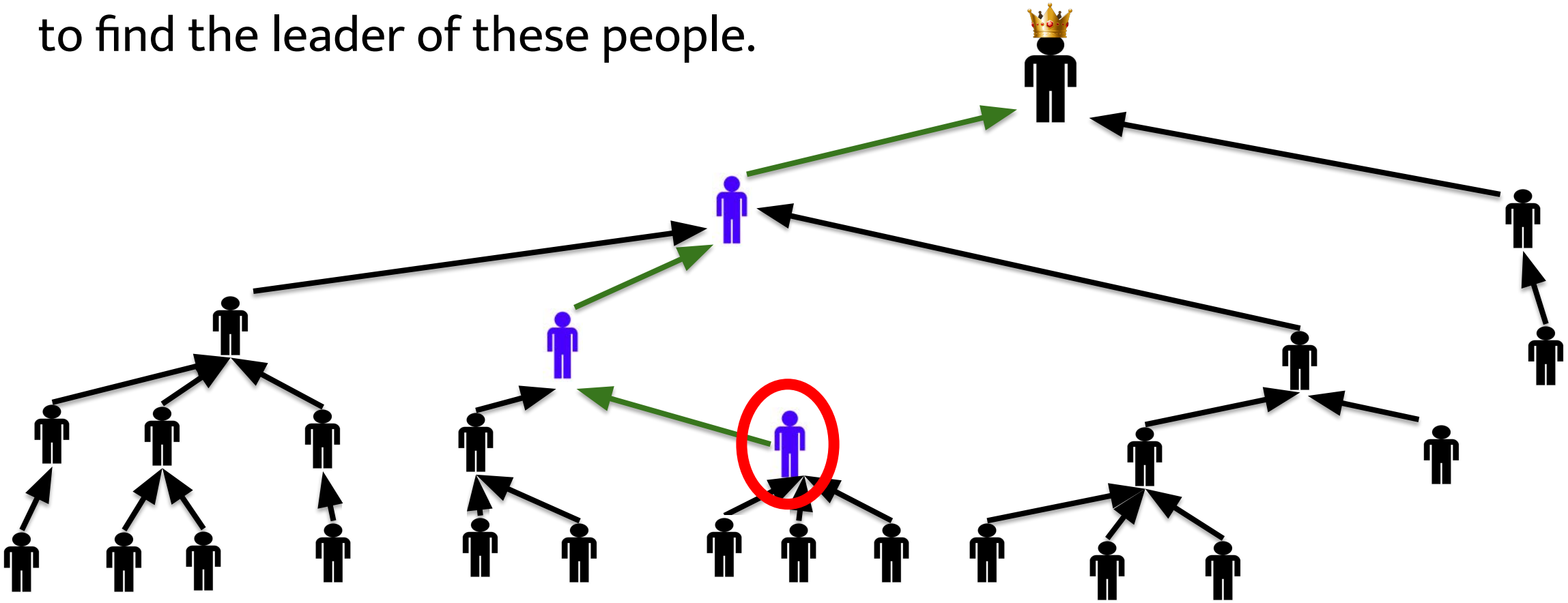
Union Find: Path Compression

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



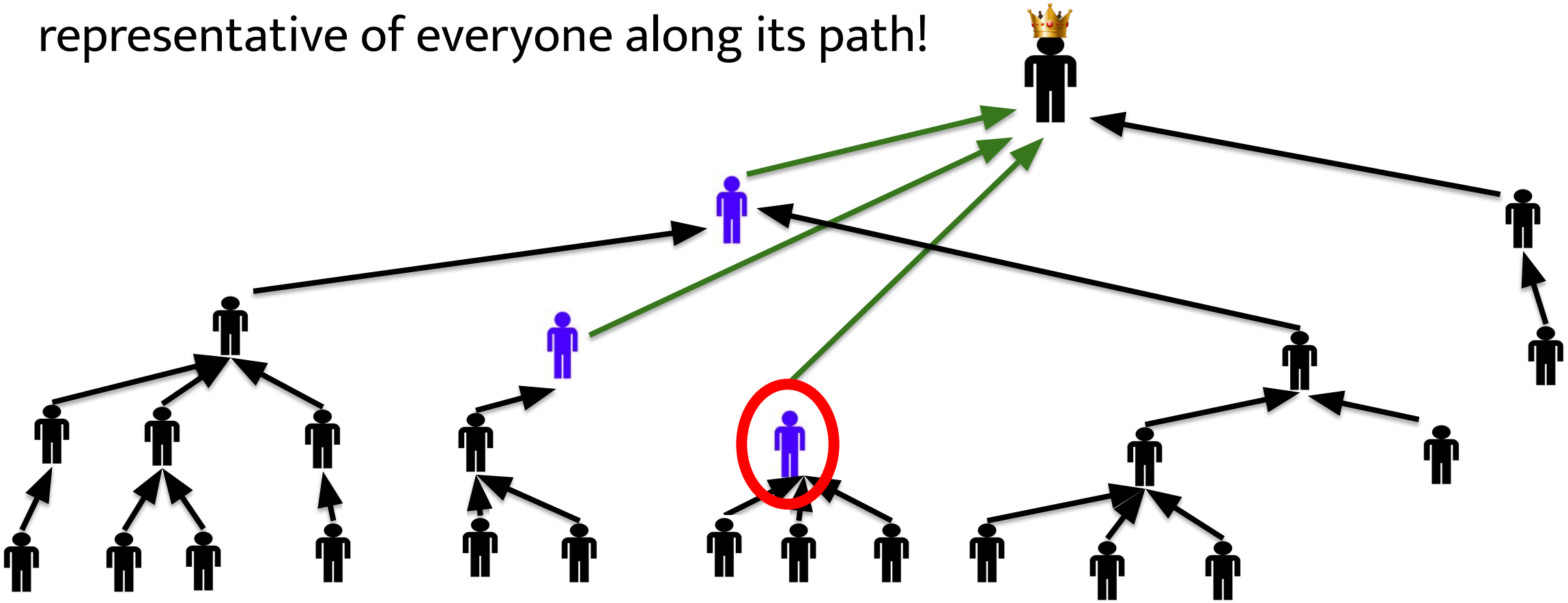
Union Find: Path Compression

- 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

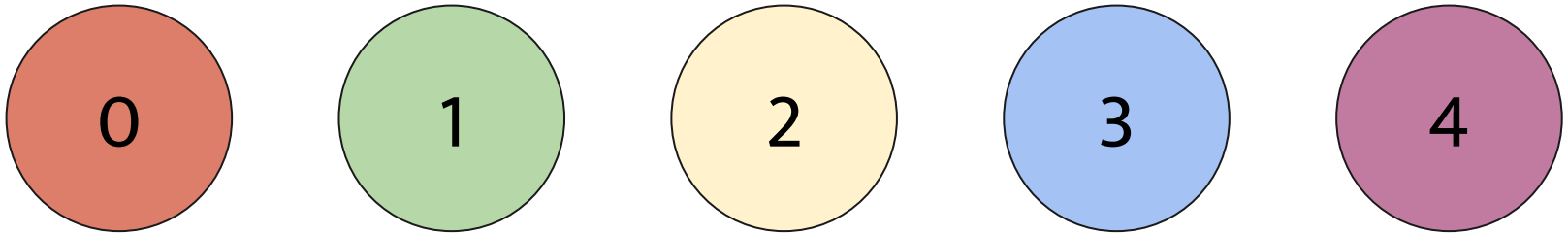
- To reduce the number of intermediaries we have to visit the next time find is called on the circled person, we can make the leader the ultimate representative of everyone along its path!



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

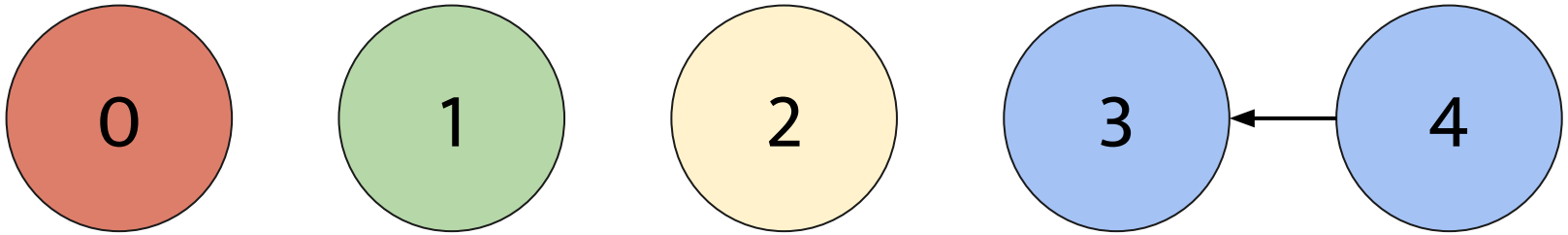


Here, we have five separate groups. Because every representative is different, nothing is connected.

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

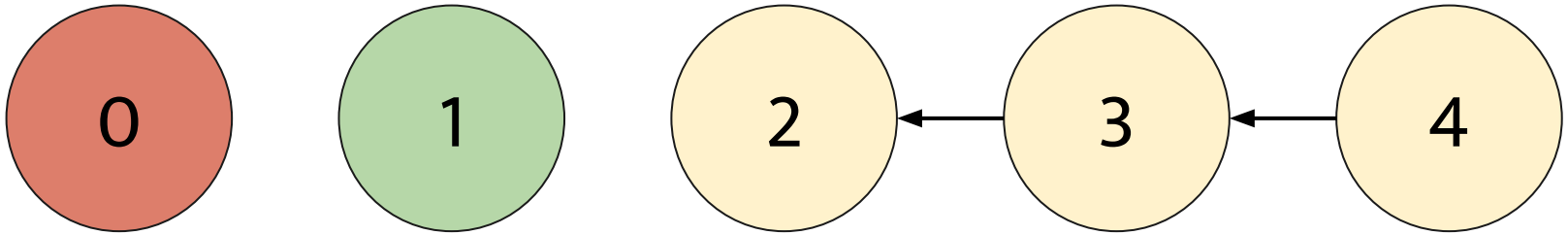


Union(3, 4) - now 4's representative becomes 3.

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

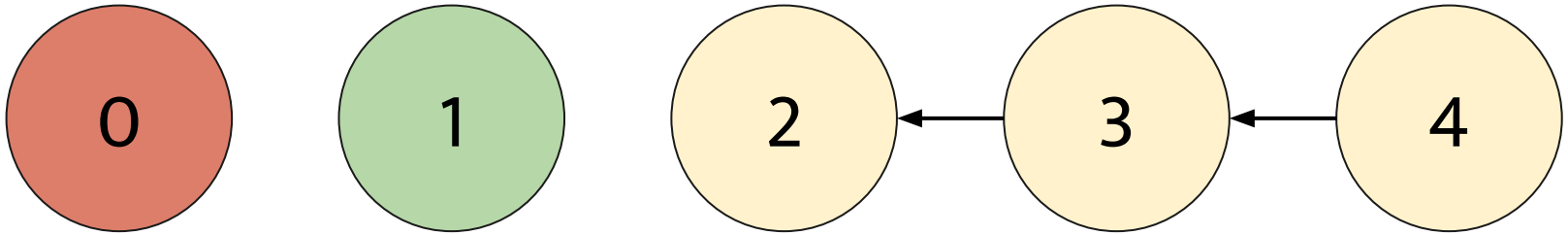


Union(2, 3) - 3's representative is now 2. 2 is now the ultimate rep for the group (notice how 2's rep is itself, but 3 and 4's are not themselves).

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;
}
```

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 do we fix this? We go back through and update the whole path to point to the ultimate representative (path compression)!

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.

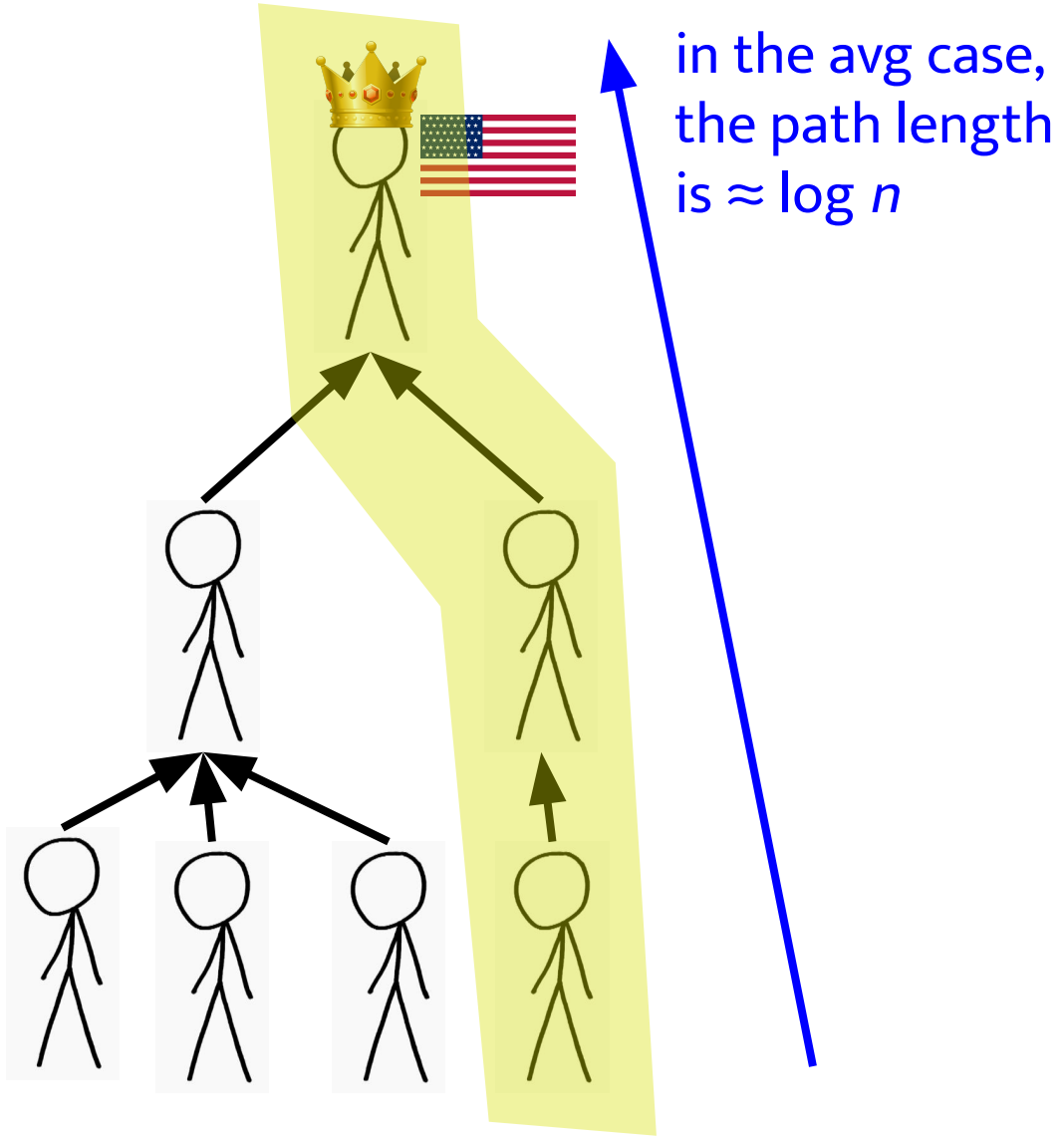
Find(x) With Path Compression

- 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;
}
```

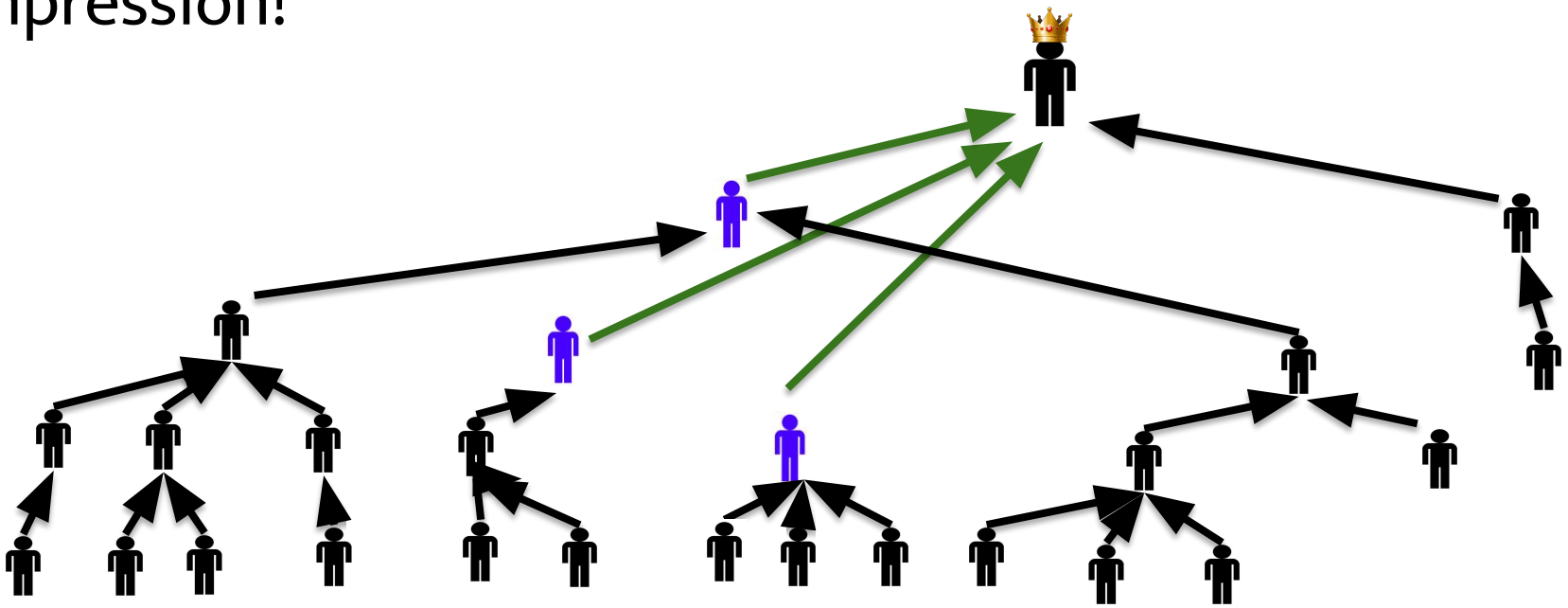
Find(x) in the Average Case

- Without path compression, average time complexity of find is O(log *n*).
  - There's only one ultimate representative: the head of a tree. With a random union pattern, we should expect a branching structure instead of a stick structure. Since the most elements are low in a tree, you are more likely to call find on an element farther from the root. 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

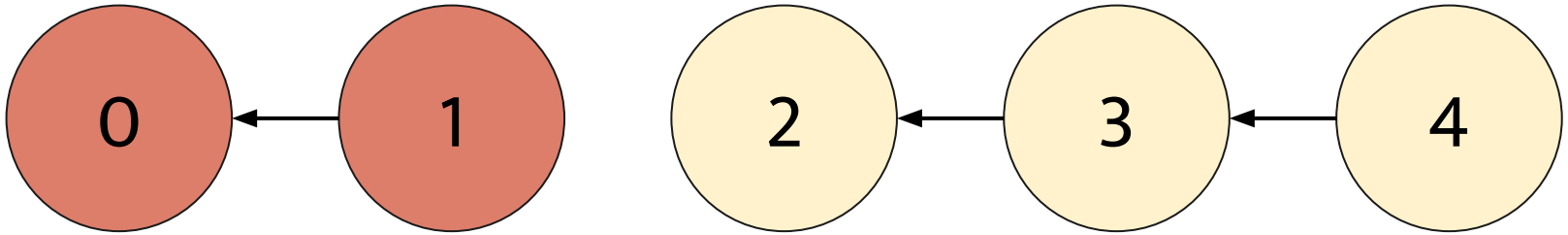
- With path compression, complexity of find becomes amortized **O(α(*n*))**, **or essentially O(1)** - this is the inverse Ackermann function, which grows very slowly... you don't need to worry about the details.
- This is better both in the average and worst case, so we might as well use path compression!



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

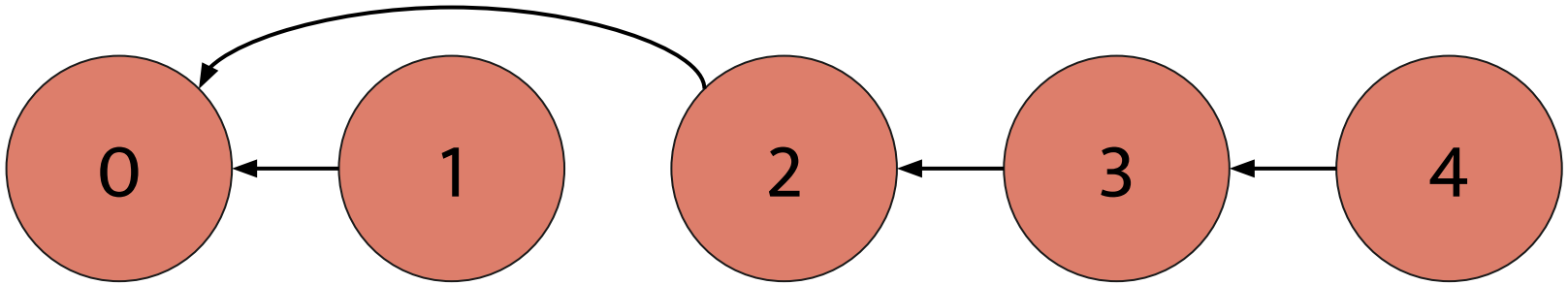


How do we handle **union(0, 2)** in this situation?

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(0, 2)** in this situation? We find the ultimate rep of both, and have one rep point to the other rep! **reps[find(2)] = find(0);**

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 **O(α(*n*)) ≈ 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 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: 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);
};
```

Functors

Functors

- Functors (i.e. function objects) are objects that can be called as if they were ordinary functions.
  - often used with STL containers and algorithms
  - made possible by overloading `operator()`
  - comparison functors are often referred to as comparators

Functors: An Example

```
class Person {
    int age;
public:
    int get_age() const {return age;}
};

class PersonComparator {
public:
    bool operator()(const Person& p1, const Person& p2) const {
        return p1.get_age() < p2.get_age();
    }
};
```

This comparator takes in two Person objects and checks if Person 1 is younger than Person 2!

Functors: An Example

```
class Person {
    int age;
public:
    int get_age() const {return age;}
};

class PersonComparator {
public:
    bool operator()(const Person& p1, const Person& p2) const {
        return p1.get_age() < p2.get_age();
    }
};
```

```
Person person1;
Person person2;

PersonComparator my_functor;

if (my_functor(person1, person2))
   cout << "Person1 is youngest";
else
   cout << "Person2 is youngest";
```

This comparator takes in two Person objects and checks if Person 1 is younger than Person 2!

Custom Object Priority Queues

```
class PersonComparator {
public:
    bool operator()(const Person& p1, const Person& p2) const {
        return p1.get_age() < p2.get_age();
    }
};
```

With this comparator, you can now build a priority queue of Person objects:

```
std::priority_queue<Person, std::vector<Person>, PersonComparator> myPplPQ;
```

Custom Object Priority Queues

- In fact, as long as `operator<` is defined for an object of type T, we can build a priority queue for objects of type T.

```
struct Person {
   int age;
   Person(int age_in) : age(age_in) {}
   bool operator<(const Person &p) const { return age < p.age; }
};
```

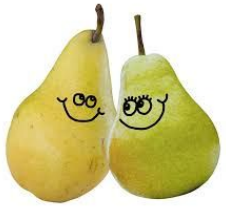
- Since `operator<` is defined for the `Person` object above, the following code would run without any issues:

```
vector<Person> personvec = {Person{25}, Person{33}, Person{19}};
priority_queue<Person> myPQ (personvec.begin(), personvec.end());
cout << myPQ.top().age; // prints 33, the oldest person in the PQ
```

Pairs and Tuples

Pairs

- A **pair** is a data type that groups together two values (which can be of different types) and treats them as a single unit. They are defined in the utility library, so you should **`#include <utility>`** if you want to use them.
- The two values of a pair can be accessed using its members, `first` and `second`. This is similar to how a `struct` with two members would behave!
  - However, pairs can also be copied, swapped, assigned, and compared!
  - Thus, if you want to bind two small data types together, a pair would allow you to do so without forcing you to redefine simple operations (such as comparisons).



Constructing a Pair

- The `make_pair()` function can be used to construct a pair:

```
#include <utility>
#include <string>

// Method #1
std::pair<std::string, int> pair1 = std::make_pair("eecs", 281);

// Method #2
std::pair<std::string, int> pair2{"eecs", 281};

// Method #3
std::pair<std::string, int> pair3 = {"eecs", 281};
```

Accessing Elements in a Pair

- To access elements of a pair, use `.first` for the first value and `.second` for the second value.

```
#include <utility>
#include <string>

// Make pair
std::pair<std::string, int> myPair = std::make_pair("eecs", 281);

// Modify the elements of the pair
myPair.first = "math";
myPair.second = 217;
```

Comparing Pairs

- Two pairs are equal if and only if they have identical `first` and `second` values. In the case of less than and greater than comparisons, pairs have default comparators that first compare the `first` elements and, in the case of a tie, then compare the `second` elements.
  - If `pair1 = {"parrot", 2}` and `pair2 = {"carrot", 3}`

    `pair1 < pair2` would return **`false`** (since 'p' > 'c')
  - If `pair1 = {"parrot", 2}` and `pair2 = {"parrot", 3}`

    `pair1 < pair2` would return **`true`** (since "parrot" == "parrot" and 2 < 3)

Tuples

- Like pairs, **tuples** group together elements - but the number of elements grouped as a single object can be greater than two!
- To use tuples and tuple operations in your program, you must include the tuple library using `#include <tuple>`.

Constructing a Tuple

- The `make_tuple()` function can be used to construct a tuple:

```
#include <tuple>
#include <string>

std::tuple<std::string, int, std::string, double> myTuple
   = std::make_tuple("eecs", 281, "paoletti", 3.14);
```

- Tuple comparison is similar to its pair counterpart: for comparison, the first component gets compared first, the second component gets compared second (if the first components are the same), the third component gets compared third (if the first two are ties)... and so on.

Accessing Elements in a Tuple

- To access elements in a tuple, use the `get` operation:

```
get<i>(myTuple);
```

where i represents the position of a component in a tuple, using zero indexing - i.e. `get<i>(myTuple)` returns the i^{th} element of `myTuple`.

```
std::tuple<std::string, int, std::string, double> myTuple
   = std::make_tuple("eecs", 281, "paoletti", 3.14);

   get<0>(myTuple) returns "eecs"
   get<1>(myTuple) returns 281
   get<2>(myTuple) returns "paoletti"
   get<3>(myTuple) returns 3.14
```

NOTE: you must know the value of the index i at **compile** time. Passing an index into i during runtime would cause an error!

Tuple Concatenation

- The `tuple_cat()` function allows you to concatenate two tuples together. For example:

```
tuple<string, int, double> paoletti = make_tuple("EECS", 281, 3.14);
tuple<double, string, int> darden = make_tuple(1.618, "EECS", 370);
```

- If you ran `tuple_cat()` on these two tuples, you would end up with a tuple that has six components:

```
tuple<string, int, double, double, string, int> combined = tuple_cat(paoletti, darden);
```

```
get<0>(combined) returns "EECS"
get<1>(combined) returns 281
get<2>(combined) returns 3.14
get<3>(combined) returns 1.618
get<4>(combined) returns "EECS"
get<5>(combined) returns 370
```

Handwritten Problem

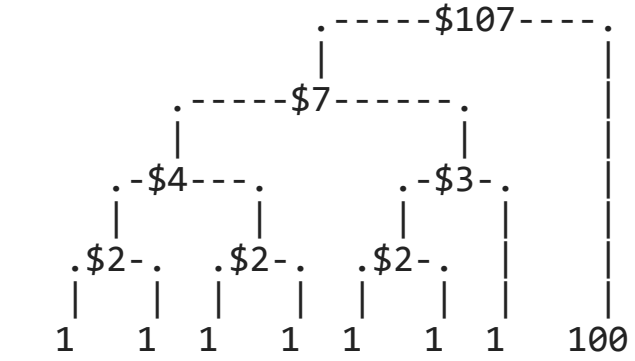
Handwritten Problem

- 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 join_ropes(vector<int>& rope_lengths);
```

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?




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