EECS 281 Midterm Review

Administrative

- Midterm is **Thursday, October 20th** from **7:00 - 9:00 pm**
- Room assignments will be distributed on Piazza @6
- You will be allowed a one-sided single sheet (8.5 by 11) of paper to bring into the exam.
- No labs the week of the midterm

- **These slides are available in Files->Exam - Practice ->Exam 1->F22 Midterm Review Slides and a recording will be posted in the same spot after the review session.**

General Exam Review

- Q25 Iterators
- Lecture / Lab Slides
- Canvas Quizzes
- Project Algorithms
- Class Notes by Andrew Zhou (ajzhou.gitlab.io/eecs281/notes/)

Content Review

Complexity/Recurrence

Asymptotic Complexity

Big-O

- an **asymptotic upper bound**, generally a worst-case complexity measure
- $f \in O(g)$ means f grows **at most** as quickly as g

Big-Omega

- an **asymptotic lower bound**, generally a best-case complexity measure
- $f \in \Omega(g)$ means f grows **at least** as quickly as g

Big-Theta

- an **asymptotic tight bound**, if a function is both $O(f)$ and $\Omega(f)$, then the function is $\Theta(f)$
- $f \in \Theta(g)$ means f and g grow at the same rate

Complexity Analysis

- In general,

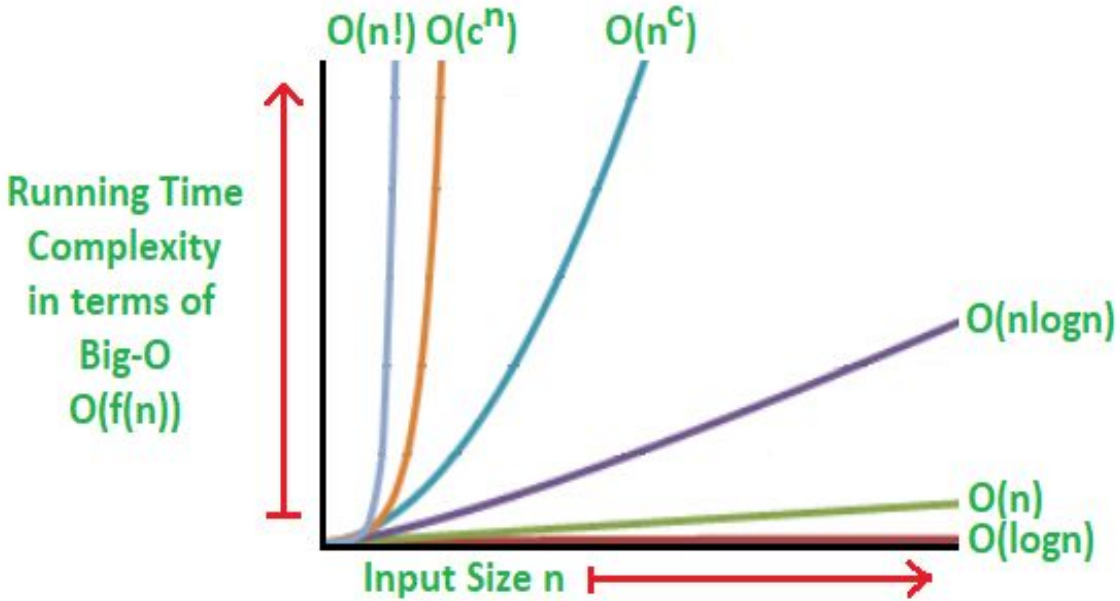
Slowest-growing → Fastest-growing

Constant	Linear	Polynomial	Exponential	Factorial
1	3n	n^3	2^n	n!

f(n) = O(g(n)) when f(n) is slower-or-similar-growing than g(n)

f(n) = Ω(g(n)) when f(n) is faster-or-similar-growing than g(n)

Complexity Analysis



O(n!),O(c^n),O(n^c) - Worst

O(nlogn) - Bad

O(n) - Fair

O(logn) - Good

O(1) - Best

Source: https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/

Average Case vs Amortized Complexity

Average-case complexity refers to the average cost of the function over all possible inputs

- Recall the concept of expected value from 203, think of average case as the kind of case that you would most likely get
- Ex: `std::sort` is average-case O(n log n) time

Amortized complexity refers to the average cost of the function over multiple operations

- To compute, assume the function is called many times in sequence, and calculate the average complexity of those calls
- Ex: `std::vector::push_back` is amortized O(1) time

Average Case vs Amortized Complexity

- You work at an ice cream store
- Every day, ***n*** customers arrive and each orders ***a few ice creams***

Average Case vs Amortized Complexity

- At the end of every day, a food critic arrives and orders ***the total number of ice creams ordered over the entire day (before the food critic), cubed***

Average Case vs Amortized Complexity

- **Operation:** selling ice cream to a customer
- **Resource of interest:** # ice creams sold

- **What is the average case complexity of selling ice cream?**

- **What is the amortized complexity of selling ice cream *over an entire day?***

Average Case vs Amortized Complexity

- **Operation:** selling ice cream to a customer
- **Resource of interest:** # ice creams sold

- **What is the average case complexity of selling ice cream?**
  O(1)
  Avg case = avg customer
- **What is the amortized complexity of selling ice cream *over an entire day?***

Average Case vs Amortized Complexity

- **What is the amortized complexity of selling ice cream *<u>over an entire day?</u>***

  $O(n^2)$

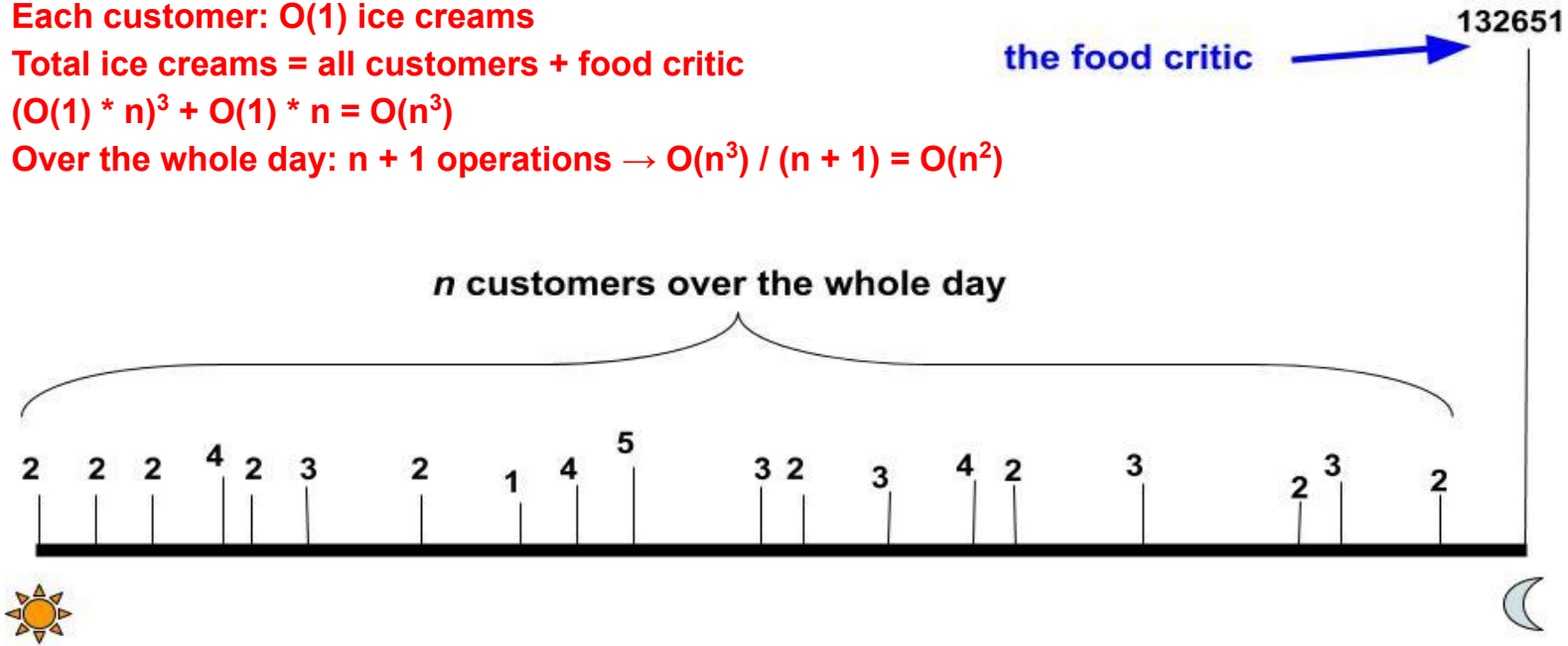
  Amortized = avg over whole day

  Each customer: O(1) ice creams

  Total ice creams = all customers + food critic

  $(O(1) * n)^3 + O(1) * n = O(n^3)$

  Over the whole day: n + 1 operations → $O(n^3) / (n + 1) = O(n^2)$



Tips

1. Lower-order terms can be ignored: $n^2 + n + 1 = O(n^2)$
2. Highest-order coefficient should be ignored: $3n^2 + 7n + 42 = O(n^2)$
3. Usually, the complexity of a loop can be inferred from its bounds
    a. There *are* exceptions, and to be sure, you need to read the entire loop

Sample question

The NRMP, which is often known as “The Match”, is an algorithm used to place training healthcare professionals in residency and fellowship programs: http://www.nrmp.org/matching-algorithm/.

When there are N applicants, M programs, and k voided tentative matches, what is the complexity that best describes the algorithm? For simplicity, ignore the fact that the actual algorithm considers couples who wish to match together. Provide an answer in terms of those variables.

Master’s Theorem / Recurrence

Recurrence Relations

Definition

A *recurrence equation* describes the overall running time on a problem of size *n* in terms of the running time on smaller inputs. [CLRS]

Ex. Fibonacci Sequence: f(n) = f(n-2) + f(n-1)

Recurrence Relations

Equation for Runtime of a Recursive Program

There are 3 steps to making a Recurrence Relation

1. Identify the complexity of each recursive call
   - Ex. O(1), O(n), …
2. Identify how much the input shrinks each call
   - Ex. T(n-1), T(n/2), …
3. Identify how many recursive calls are made *per iteration*
   - Ex. 1, 2 ,…

$T(n)$ = <3>*<2> + <1>

Recurrence Relations

Factorial Example

```
1 int factorial(int n) {
2   if (n == 0)
3     return 1;
4   return n * factorial(n - 1);
5 } // factorial()
```

T(*n*) = <3>*<2> + O(1)

Recurrence Relations

Factorial Example

```
1 int factorial(int n) {
2   if (n == 0)
3     return 1;
4   return n * factorial(n - 1);
5 } // factorial()
```

T(*n*) = <3>*T(*n* – 1) + O(1)

Recurrence Relations

Factorial Example

```
1 int factorial(int n) {
2   if (n == 0)
3     return 1;
4   return n * factorial(n - 1);
5 } // factorial()
```

$T(n) = 1*T(n - 1) + O(1)$

The Master Theorem

Let $T(n)$ be a monotonically increasing function that satisfies:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$T(1) = 1$$

Where $a \geq 1$, $b \geq 1$ If $f(n) \in \Theta(n^c)$, then:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^c \\ \Theta(n^c \log_2 n) & \text{if } a = b^c \\ \Theta(n^c) & \text{if } a < b^c \end{cases}$$

The Master Theorem

T(n) = 2*T(n / 2) + O(n)

- a = 2
- b = 2
- c = 1

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^c \\ \Theta(n^c \log_2 n) & \text{if } a = b^c \\ \Theta(n^c) & \text{if } a < b^c \end{cases}$$

The Master Theorem

T(n) = 2*T(n / 2) + O(n)

a = 2

b = 2

c = 1

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^c \\ \Theta(n^c \log_2 n) & \text{if } a = b^c \\ \Theta(n^c) & \text{if } a < b^c \end{cases}$$

T(n) = O($n$logn)

Recurrence	Example	Big-O Solution
$T(n) = T(n / 2) + c$	Binary Search	$O(\log n)$
$T(n) = T(n - 1) + c$	Sequential Search	$O(n)$
$T(n) = 2T(n / 2) + c$	Tree Traversal	$O(n)$
$T(n) = T(n - 1) + c_1 * n + c_2$	Selection/etc. Sorts	$O(n^2)$
$T(n) = 2T(n / 2) + c_1 * n + c_2$	Merge/Quick Sorts	$O(n \log n)$

The Master Theorem

You **cannot** use the Master Theorem if:

- $T(n)$ is not monotonically increasing: $T(n) = \sin(n)$
  - $T(n)$ both increases and decreases
- $f(n)$ is not a polynomial
- b cannot be expressed as a constant
  - $T(n) = \text{sqrt}(n)$

Example of when **not** to use:

$T(n) = T(n - 1) + n$

Because $T(n) \neq aT(n / b) + f(n)$

Solving Recurrence Relations with Substitution

The Substitution Method

1. Write out T(n), T(n - 1), T(n - 2)
2. Substitute T(n - 1) and T(n - 2) into T(n)
3. Look for a pattern
4. Use a summation formula

Arrays & Linked Lists

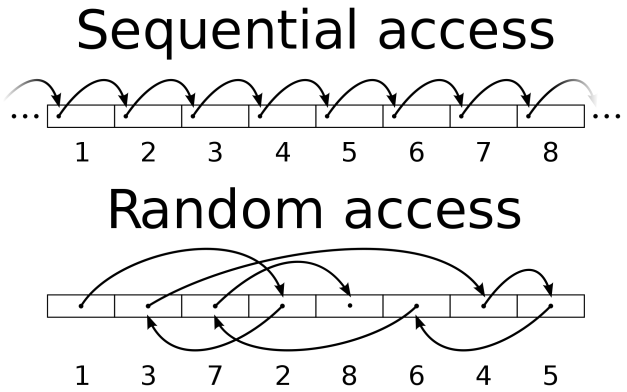
Container Access Order

Sequential Access

- Must start at the ends of the data structure and linearly move to the *n*th item
- Ex: linked-lists and binary search trees

Random Access

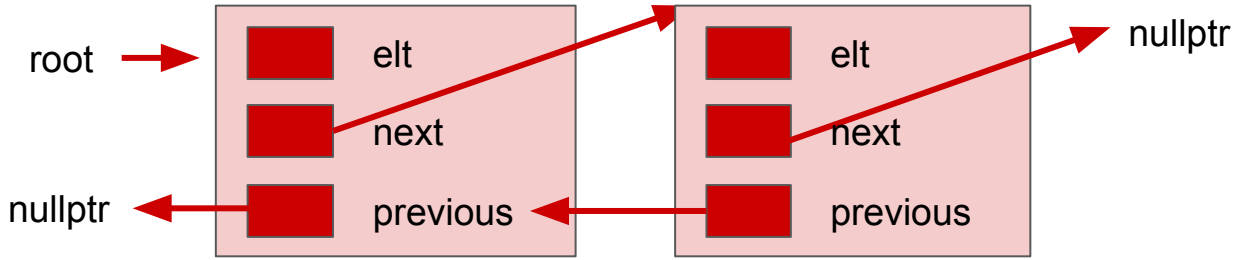
- Can directly access a point in O(1) time
- Ex: arrays, `std::vector`



Array-based Containers: Vectors

- **Capacity**:
  - Number of elements the underlying array is *capable* of holding
  - vector<T>.reserve(capacity);
- **Size**:
  - Number of elements added to the underlying array
  - vector<T>.resize(size);
  - `push_back()`
    - Achieves amortized O(1)
    - Reallocation O(n)

Linked-Lists



Sequential access only

Requires memory overhead for the `Node` data structure

- Singly linked – each node contains a pointer to the next node
- Doubly linked – each node contains a pointer to the next node *and* the previous node

Invariants

- The last node has a next pointer of `nullptr`
- If the list is empty, the root Node pointer is a `nullptr`

Choosing between array and linked list

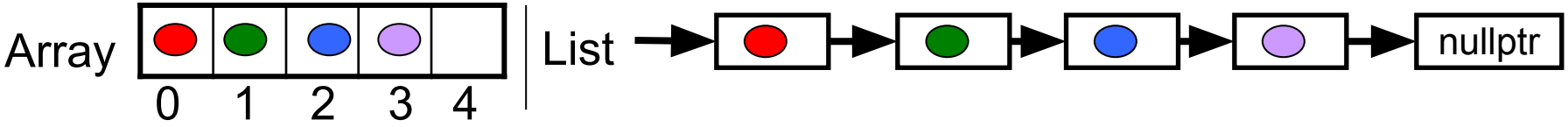
In general, the `std::vector` is the most efficient. Why?

- Cache-friendliness
- Amortized O(1) push
- If your data types are small enough, copying/moving is unlikely to be a significant portion of time

When to use a linked list:

- Code is dominated by splitting and merging of very large lists (and you have verified this)
- Lots of inserting or deleting from the middle of the list
- You have measured and profiled

Arrays vs. Linked Lists



	Arrays	**Linked Lists**
Access	Random in *O*(1) time Sequential in *O*(1) time	Random in *O*(*n*) time Sequential in *O*(1) time
Insert **Append**	Inserts in *O*(*n*) time Appends in *O*(1) time (vector)	Inserts in *O*(*n*) time (O(1) if given location) Appends in *O*(*n*) time (O(1) if tail pointer)
Bookkeeping	`ptr` to beginning `currentSize` (or `ptr` to end of space used) `maxSize` (or `ptr` to end of allocated space – needed if dynamic sizing occurs)	`head` pointer to first node `size` (optional) `tail` ptr to last node (optional) In each node, `ptr` to next node, and possibly previous in doubly linked list. Wasteful for small data items
Memory	Wastes memory if oversized Wastes time if repeatedly undersized	Allocates memory as needed Requires memory for pointers

Complexity of Deleting a Node?

- Assume you're given a pointer to the node you want to delete
- Can we delete in O(1) time?
- Doubly-linked: trivial, just stitch up the pointer
- Singly-linked?

Complexity of Deleting a Node?

- Assume you're given a pointer to the node you want to delete
- Can we delete in O(1) time?
- Doubly-linked: trivial, just stitch up the pointer
- Singly-linked?
  - Overwrite data in node to delete with next node data
  - Delete next node
  - Assumes data can be able to be copied or moved

Container Types

What is a Container?

A **container** is a class, a data structure or an abstract data type (ADT) whose instances are collections of other objects. In other words, they store objects in an organized way that follows specific access rules

Types of Containers

Type	Distinctive interfaces (not all methods listed)
Container	Supports `add()` and `remove()` operations
Searchable Container	Adds `find()` operations
Sequential Container	Allows iteration over elements in some order
Ordered Container	Sequential container which maintains current order. Can arbitrarily insert new elements anywhere. Example: Books on a shelf
Sorted Container	Sequential container with pre-defined order. Can NOT arbitrarily insert elements. Example: Students sorted by ID

Interfaces for Sorted and Ordered Containers

Interface	Sorted	Ordered
`addElement(val)`	Override searchable container version	Inherited from searchable container
`remove(val)`	Inherited	Inherited
`find(val)`	Inherited	Inherited
`operator[]()`	Yes	Yes
`withdraw(iterator)`	Yes	Yes
`insertAfter()`	No	Yes
`insertBefore()`	No	Yes

Common Container Complexities

Ordered Containers

Data Structure	operator[]	insertAfter()	delete()
Array	O(1)	O(n)	O(n)
Linked-list	O(n)	O(1)	O(1)

Sorted Containers

Data Structure	find()	operator[]	delete()
Array	O(log n)	O(1)	O(n)
Linked-list	O(n)	O(n)	O(1)

Ordered and Sorted Containers

Sorted Container	Ordered Container
Iterable/Searchable container* that maintains a strict value-based comparison ordering that determines the order in which elements are iterated over. If two sorted containers contain the exact same elements, they are identical.	General container where each element deterministically maintains its position specified by the user regardless of what elements are added after it or came before it. For every collection of N elements, there may be up to N! different ordered containers that hold them.

*If a container is not iterable, it is hard to argue whether it is truly sorted

Clarifying Questions

- Is a priority queue iterable/searchable?

- Is a priority queue an ordered container?

Clarifying Questions

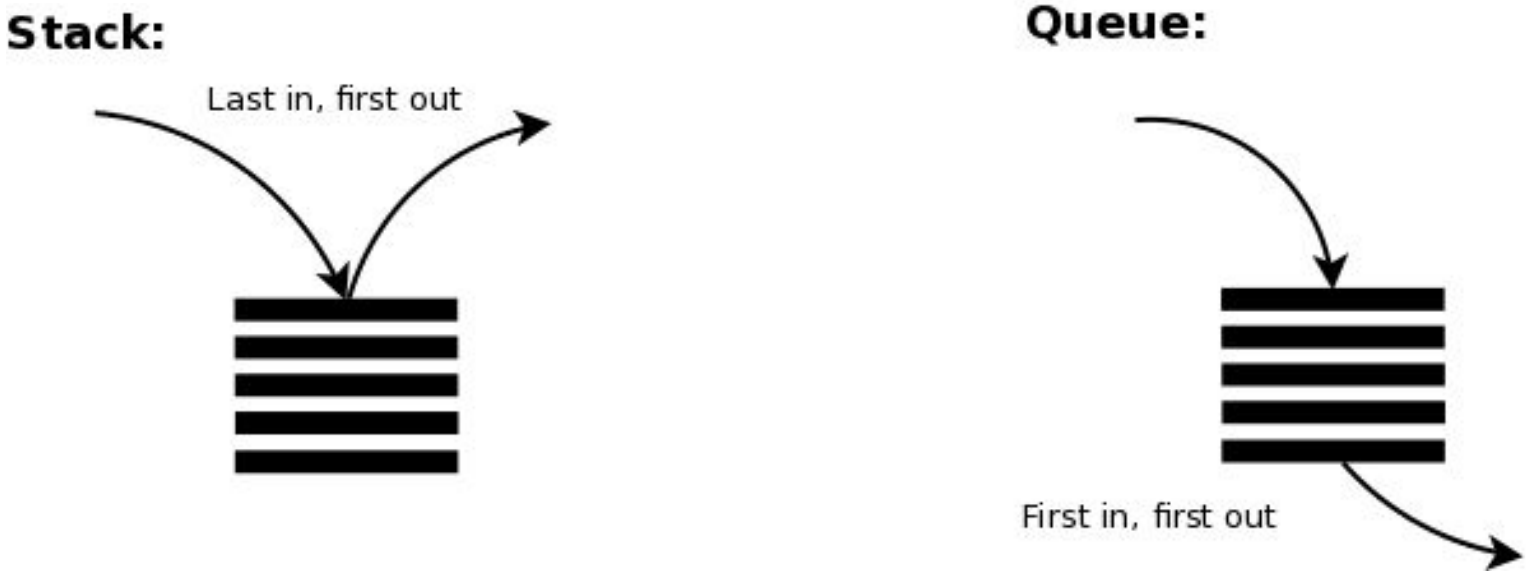
- Is a priority queue iterable/searchable?

No, elements must be removed to gain this behavior.

- Is a priority queue an ordered container?

No/Only in terms of priority

Stacks/Queues



Stacks/Queues in STL

	Stack	Queue
Default	deque	deque
Optional	list, vector	list

Stack

Last-in, First-out container

- `push`
- `pop`
- **`top`**
- `size`
- `empty`

Operation	Stack Behavior
push(value)	append value on top of stack
pop()	remove top value from stack
top()/front()	return top value of stack
size()	return # of elements in stack
empty()	return whether size() is 0

Queue

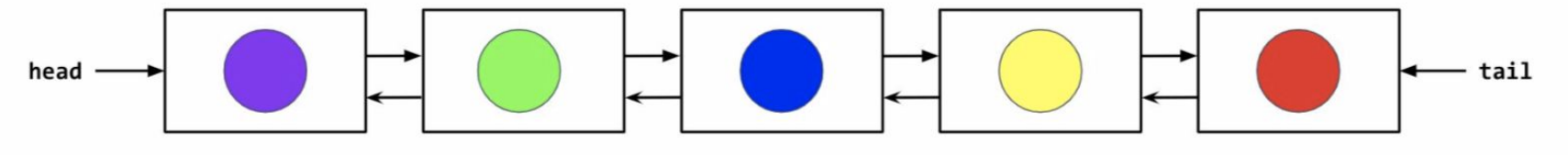
First-in, First-out container

- `push`
- `pop`
- **`front`**
- `size`
- `empty`

Operation	Queue Behavior
push(value)	append value at back of queue
pop()	remove value at front of queue
top()/front()	return value at front of queue
size()	return # of elements in queue
empty()	return whether size() is 0

Deques

- A deque can be used to support efficient front and back access!
  - a stack and queue all in one
  - but can also traverse using iterators and supports `operator[]` access
- Supports O(1) `.push_front()`, `.push_back()`, `.pop_front()`, `.pop_back()`, `.front()`, `.back()`, and `operator[]`.
- A simple implementation: a doubly-linked list:



- Potential problems with this implementation?

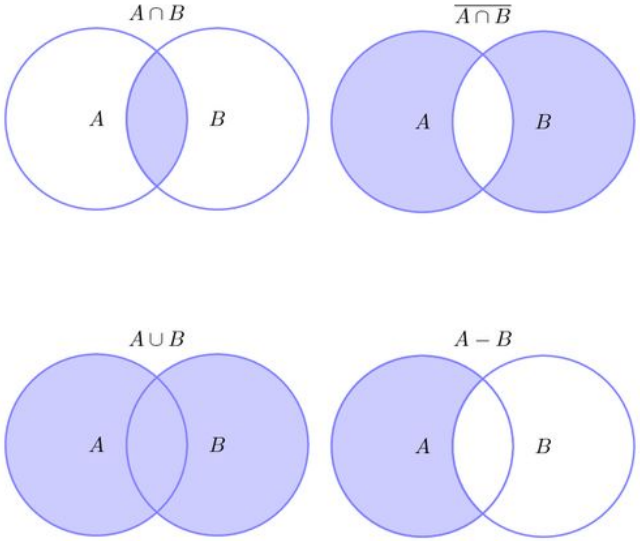
Sets and Union-Find

Let’s Set Things Up...

Set - A searchable collection of unique elements. Elements can be compared for equality.

Typical operations between two sets A and B

- Union(A, B) = elements of A or B
- Intersection(A, B) = elements of A and B
- Difference(A, B) = elements of A and not B
- Symmetric Diff(A, B) = elements of A xor B



Performing and Representing Set Operations

Sets can be represented by any searchable or iterable container

For sorted data, all set operations can be done in linear time

For unsortable data, set operations will generally take $O(n^2)$ time (with exceptions, we'll get to hashing very soon)

Disjoint Sets

- Sets are **disjoint** if they do not share any elements
- Many applications require representing and operating on disjoint sets. For example, graph connectivity

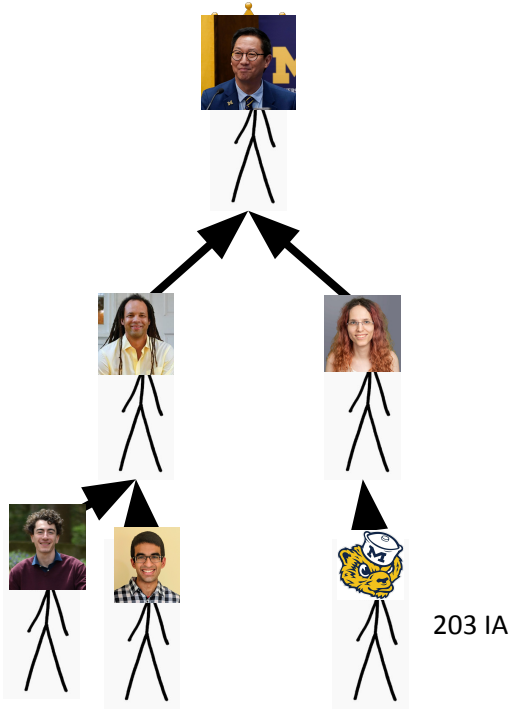
Will I be able to travel from one location to another?

Union Find

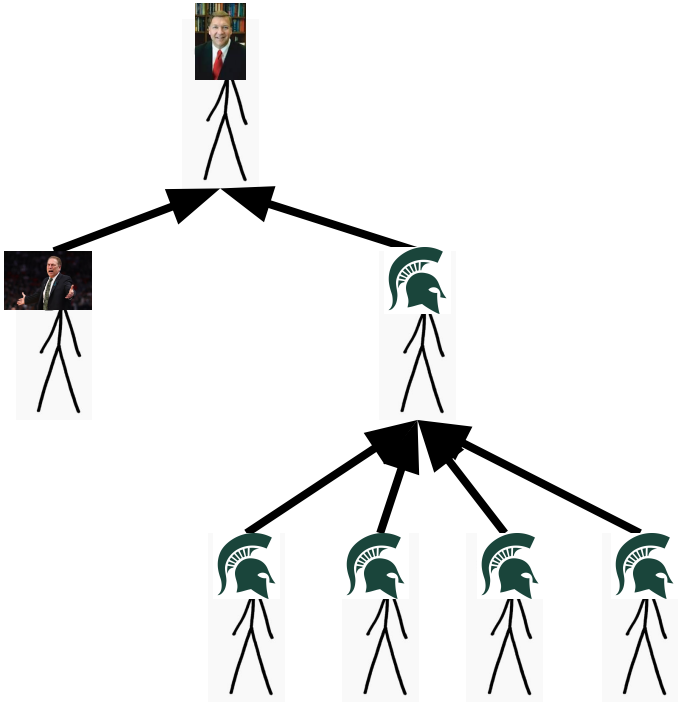
- Union Find (or Disjoint Set) is a data structure for managing “disjoint sets” - a way to tell whether two items are in the same group.
- Traditional sets won't work for this task - we want to quickly check whether **known** elements are in the same set, not check whether **any** element is part of 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 part of the same University?

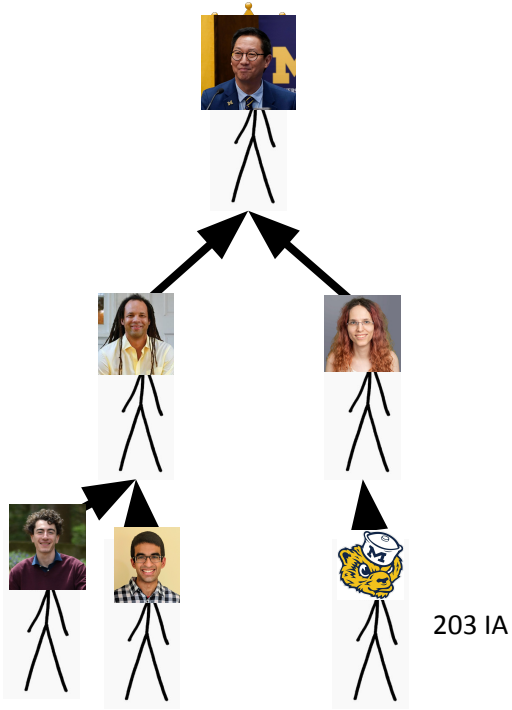


203 IA

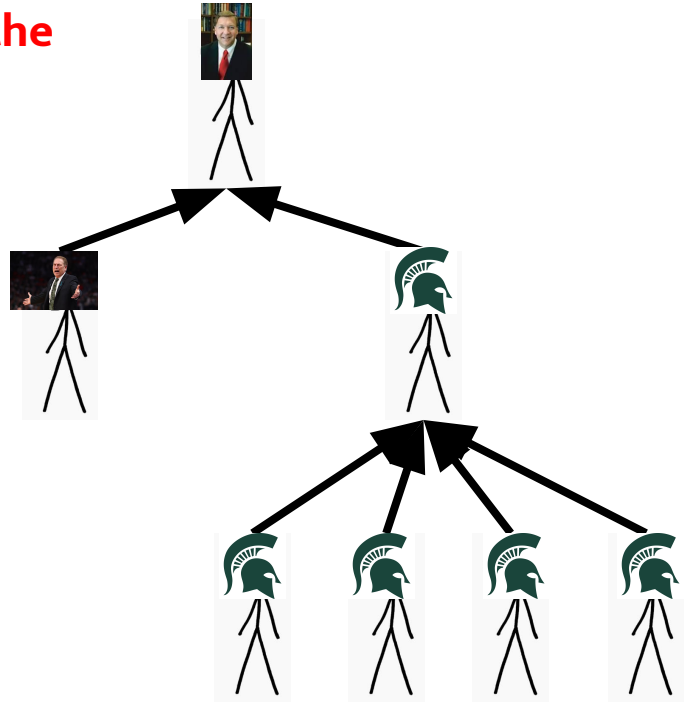


Union Find

- How can we tell if two people are part of the same University?

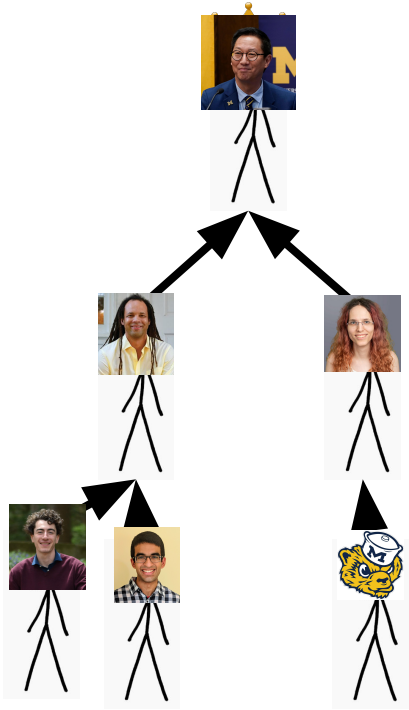


Check who is the boss!



Union Find

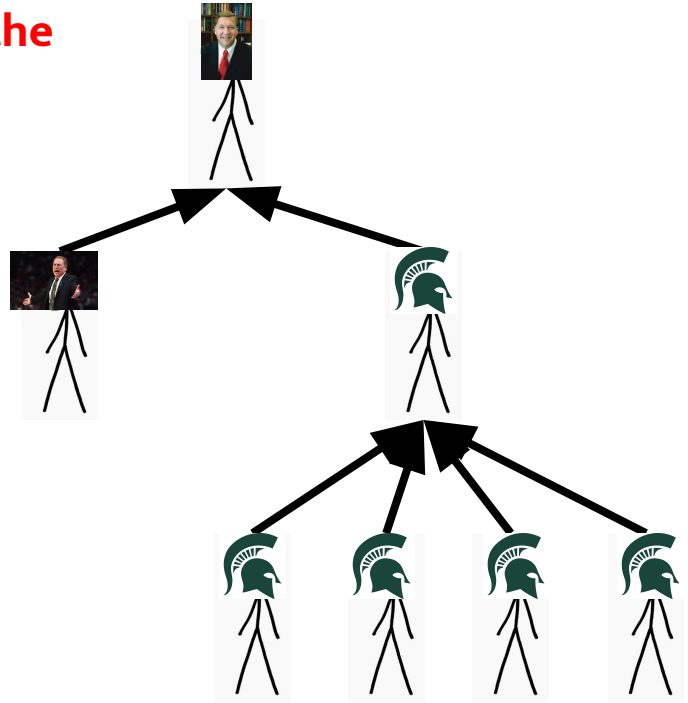
- How can we tell if two people are part of the same University?



Check who is the boss!

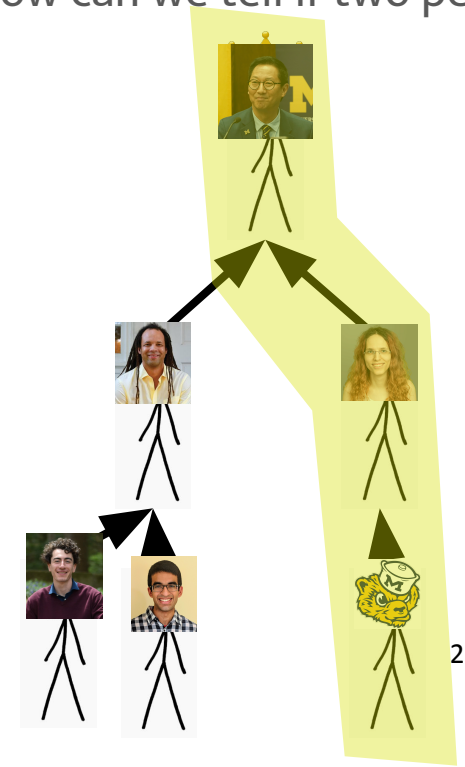
How do we find someone's boss?

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



Union Find

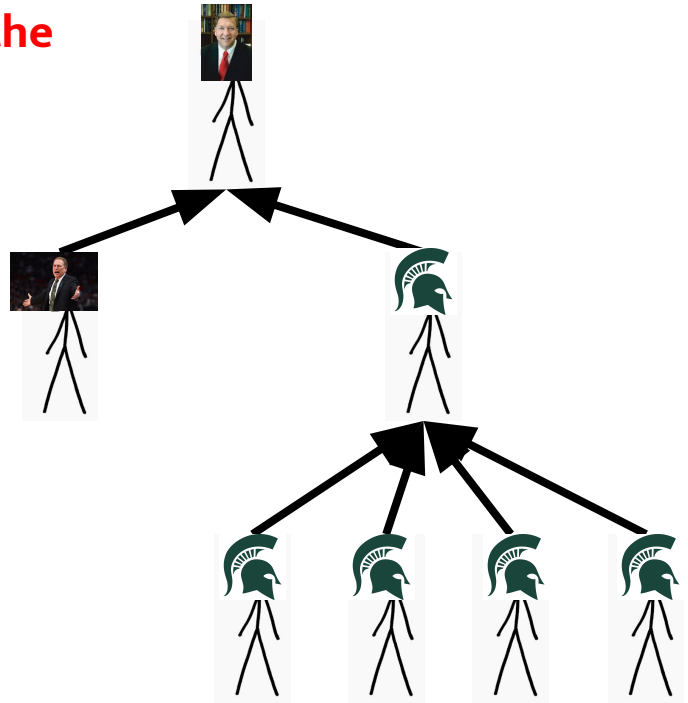
- How can we tell if two people are part of the same University?



Check who is the boss!

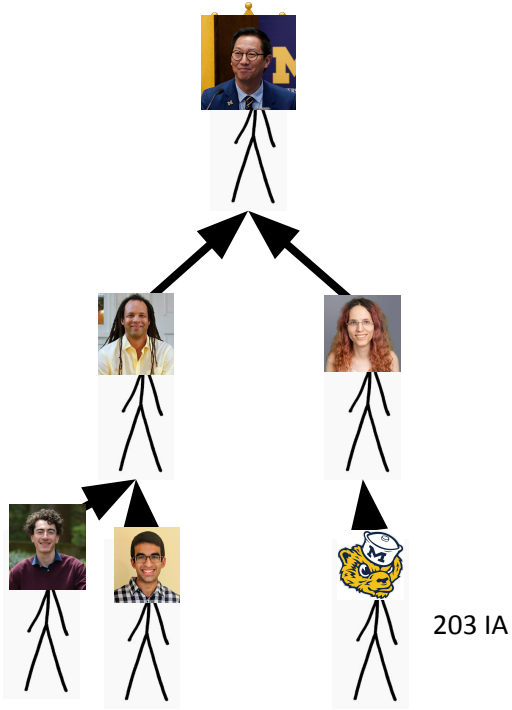
How do we find someone's boss?

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

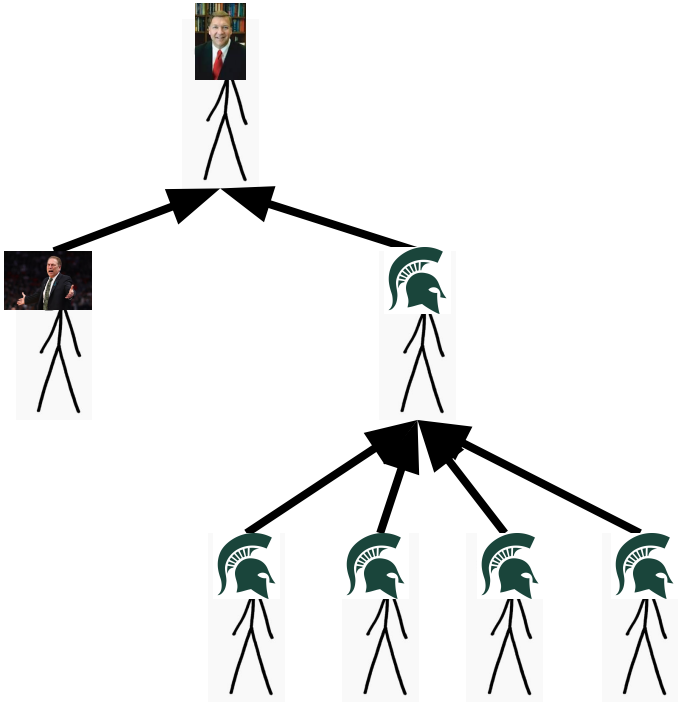


Union Find

- How can we tell if two people are part of the same University?

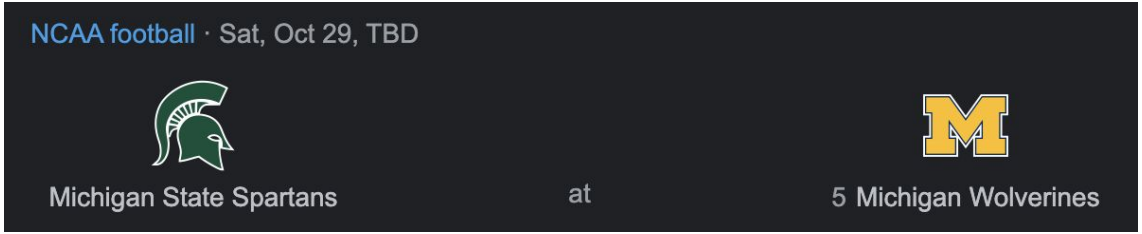


203 IA



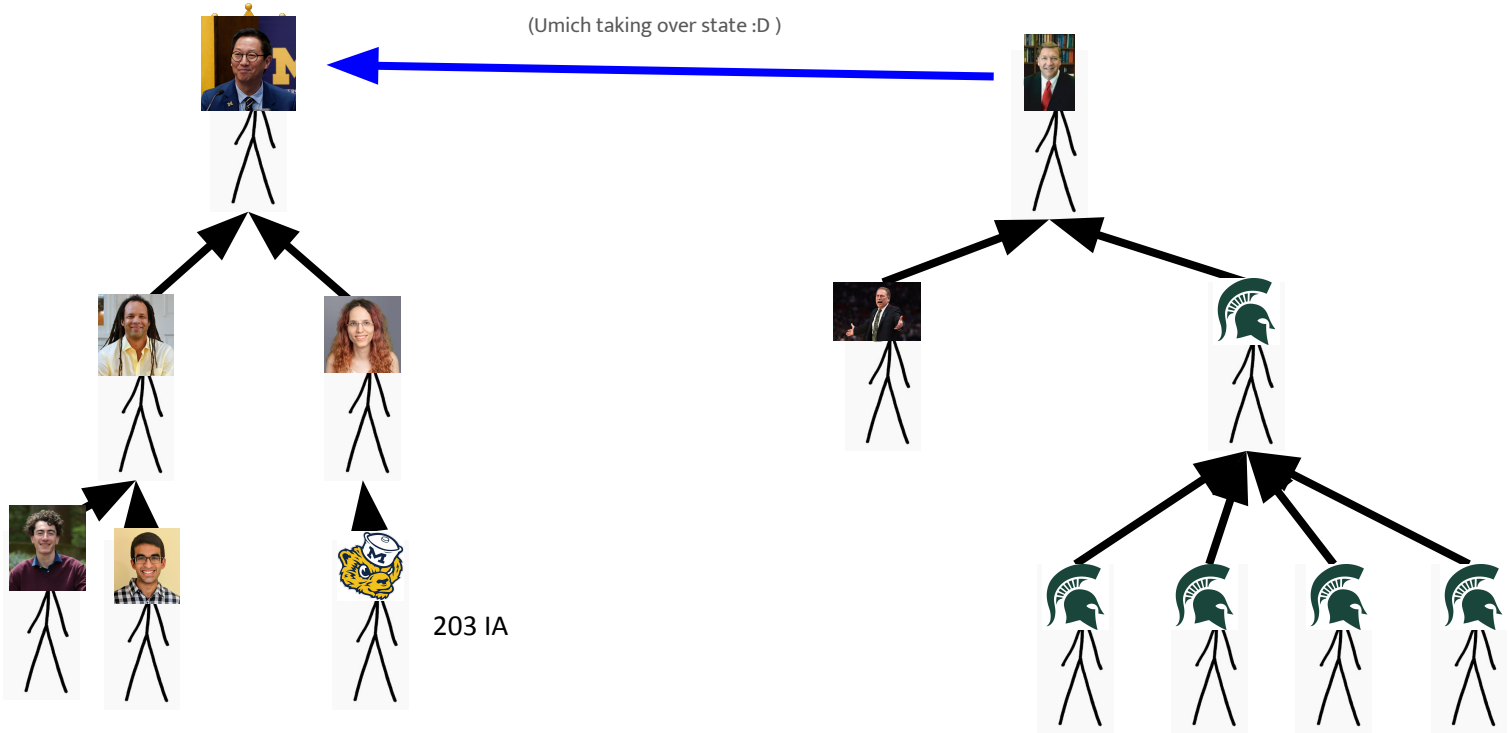
Union Set

- Suppose we have two groups. How do we combine these two groups?
  - After combining, all elements must have the same boss!
  - We want to modify as few nodes as possible (in order to be fast).
- Hypothetical example: suppose the University of Michigan and Michigan State University combined.
  - How do we modify the diagram on the previous slide to reflect this?



Union Find

- Solution: the boss 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**!

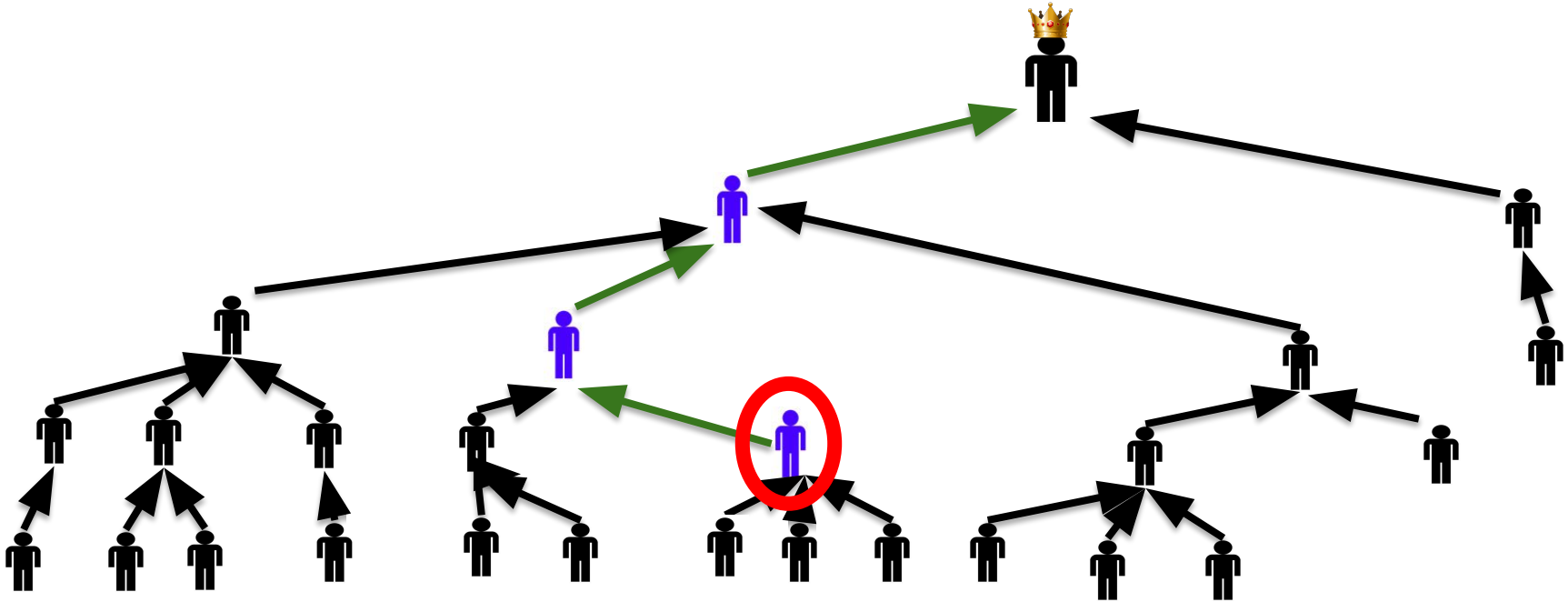
Union Find: Path Compression

- 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]
```

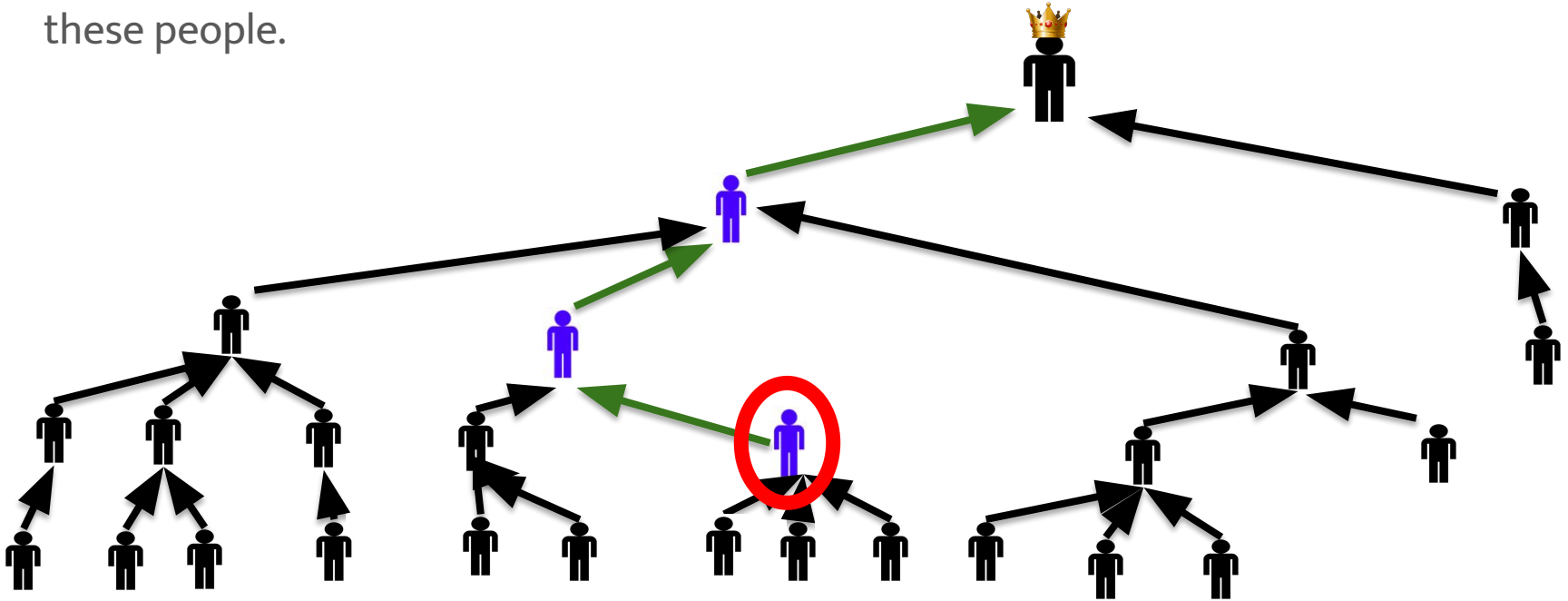
Union Find: Path Compression

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



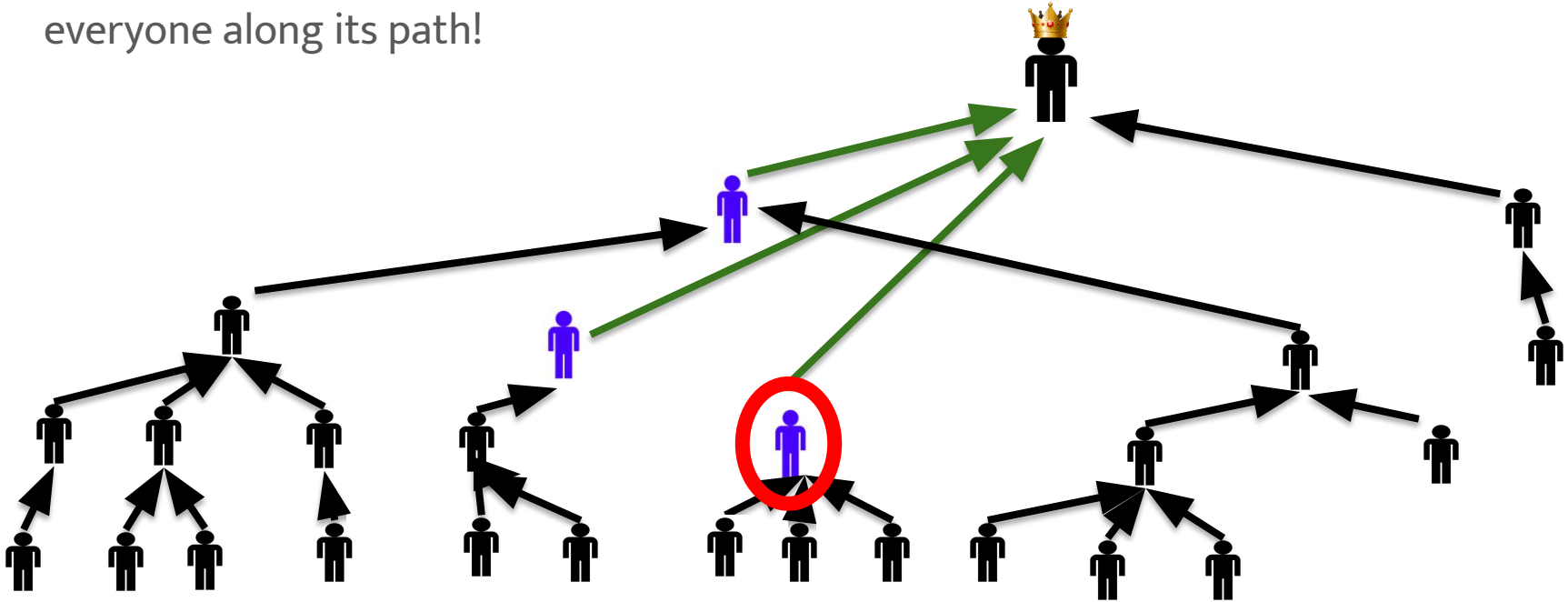
Union Find: Path Compression

- 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!



STL & Iterators

Standard Template Library Containers

Container	Purpose
`std::vector`	Dynamically resizing array
`std::list`	Doubly-linked list
`std::forward_list`	Singly-linked list
`std::deque`	Double-ended queue
`std::queue`	Queue (adapter over `std::deque`)
`std::stack`	Stack (adapter over `std::deque`)
`std::priority_queue`	Binary heap (wrapper over `std::vector`)
`And more!`	Soon™

STL Container Use Cases

- `std::vector` – very compact in memory, very cache-local, amortized constant `push/pop_back`, random access
  - Random `insert/erase` requires linear copying!
  - Elements are stored contiguously in memory.
- `std::deque` – fairly compact in memory, fairly cache-local, amortized constant `push_back/front` & `pop_back/front`, random access
  - Good choice if you need to `push/pop` at the front and back
- `std::list` – very spread out in memory, two pointers per element of memory overhead, constant `insert/erase`, no random access
  - Elements are never moved/copied, good for large objects.
  - Random `insert/erase` (given an iterator) is constant time.

Standard Template Library Algorithms

Function	Effect
`std::sort`	Sorts iterator range using a comparator (default `std::less`)
`std::upper_bound`	Returns the first element greater than a value in an iterator range
`std::lower_bound`	Returns the first element not less than a value in an iterator range
`std::nth_element`	Partially sorts a range until the nth element is in the right position
`std::make_heap`	Turns a range into a binary heap (`heapify`)
`std::pop_heap`	Performs a `pop` on a range that is a binary heap
`std::push_heap`	Performs a `push` on a range that is a binary heap
`std::remove/find/copy/(…)_if`	Performs the specified action on elements where the given functor returns `true`

And many more……

Standard Template Library Algorithms

Function	Iterator Requirement	Complexity
`std::sort`	`RandomIterator`	`O(n log n)` in `distance(first, last)`
`std::upper_bound`	`ForwardIterator`	`O(log n)` in `distance(first, last)`
`std::lower_bound`	`ForwardIterator`	`O(log n)` in `distance(first, last)`
`std::nth_element`	`RandomIterator`	`O(n)` in `distance(first, last)`
`std::make_heap`	`RandomIterator`	`O(n)` in `distance(first, last)`
`std::pop_heap`	`RandomIterator`	`O(log n)` in `distance(first, last)`
`std::push_heap`	`RandomIterator`	`O(log n)` in `distance(first, last)`
`std::remove/find/copy/(…)_if`	`ForwardIterator`	`O(n)` in `distance(first, last)`

Iterators

- An abstraction of a “pointer” that works with arrays, `std::vector`, `std::list`, `std::string`, and other STL data structures
- Dereference with `*it`, increment with `++it`
- Provide a common interface for using STL algorithms:

```
std::vector<double> v;

std::list<int> l;

std::sort(v.begin(), v.end()); // operates over a range

std::remove(l.begin(), l.end(), 1); // ditto
```

Iterators in Containers

Supported by `vector`, `string`, `list`, `deque` (and other iterable containers):

- `.begin()` → this is INCLUSIVE
- `.end()` → this is EXCLUSIVE (one past the last element)
- `.cbegin()`, `.cend()` → `const` iterators so you can't modify their value
- `.rbegin()`, `.rend()` → reverse iterators (`it++` goes backward)
  - `.rbegin()` → INCLUSIVE (the last element)
  - `.rend()` → EXCLUSIVE (one before the first element)

iterator type names are long, use **`auto`** instead

Iterators

- Different kinds of iterators provide different capabilities
  - E.g. can't "jump *n* spots forward" in a list iterator
- These are *not* types, just requirements on real iterator types
- `InputIterator` can read values while moving forward
- `OutputIterator` can write values while moving forward
- `ForwardIterator` can read or write values while moving forward
- `BidirectionalIterator` can read or write values while moving either direction
- `RandomIterator` can do pointer-like arithmetic and comparisons

What operation is supported by pointers but never iterators?

Iterators

- Different kinds of iterators provide different capabilities
  - E.g. can't "jump *n* spots forward" in a list iterator
- These are *not* types, just requirements on real iterator types
- `InputIterator` can read values while moving forward
- `OutputIterator` can write values while moving forward
- `ForwardIterator` can read or write values while moving forward
- `BidirectionalIterator` can read or write values while moving either direction
- `RandomIterator` can do pointer-like arithmetic and comparisons

What operation is supported by pointers but never iterators? `delete`

Sorting Algorithms

Sorting Algorithms

- Elementary Sorts
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
- Advanced Sorts
  - Mergesort
  - Quicksort
  - Heapsort
  - Counting/Bucket Sort

Bubble Sort

- Algorithm:
  - Step through list to be sorted, comparing each pair of adjacent items
  - Swap if they are in the wrong order.
  - Each time pass through a smaller section of the list

- “Bubbles” largest element to the top

Bubble Sort

- Pseudocode

```
For i in range [0 -> n) {
    For j in range (n -> i] {
        if (arr[j] > arr[j - 1]) {
            swap(arr[j], arr[j - 1]);
        }
    }
}
```

- The “lightest” element “bubbles” up to the top while the “heaviest” element sinks to the bottom

Bubble Sort

- Bubble sort in action:

Compare adjacent elements

Bubble Sort

- Bubble sort in action:

Swap if in the wrong order

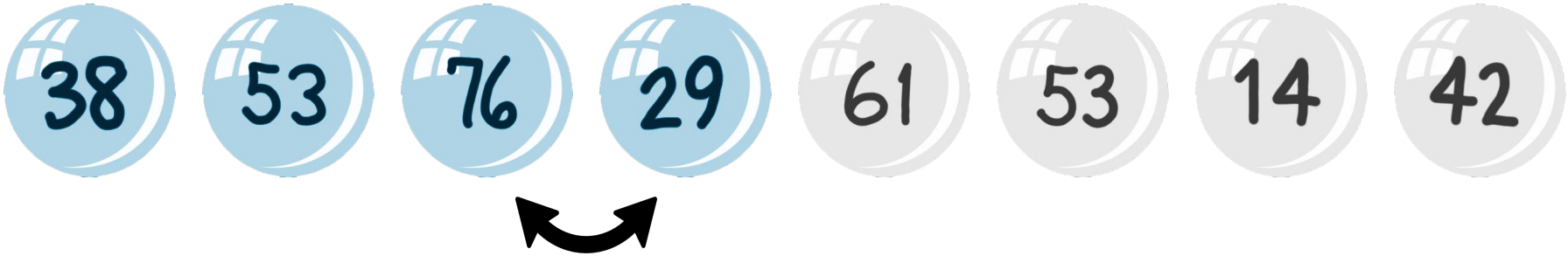
Bubble Sort

- Bubble sort in action:

Repeat with the rest of the list

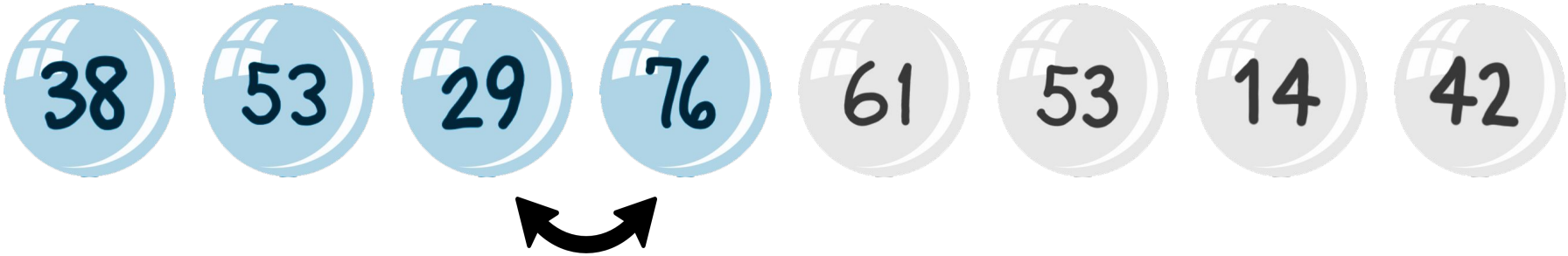
Bubble Sort

- Bubble sort in action:



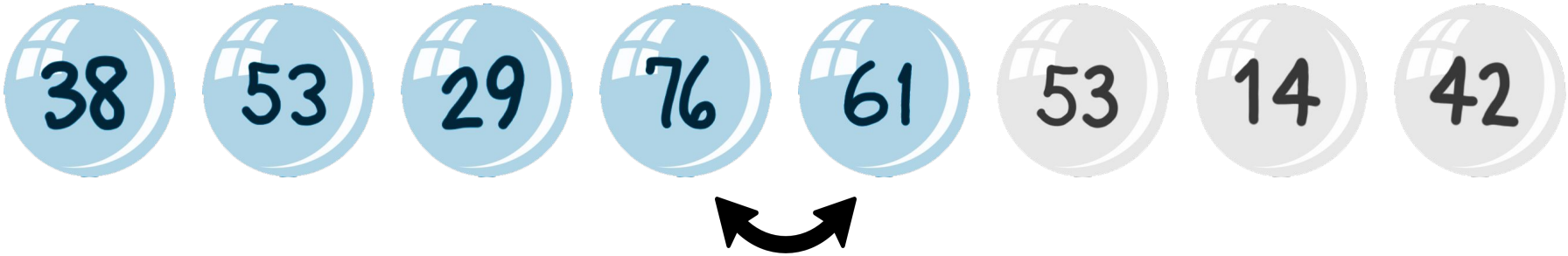
Bubble Sort

- Bubble sort in action:



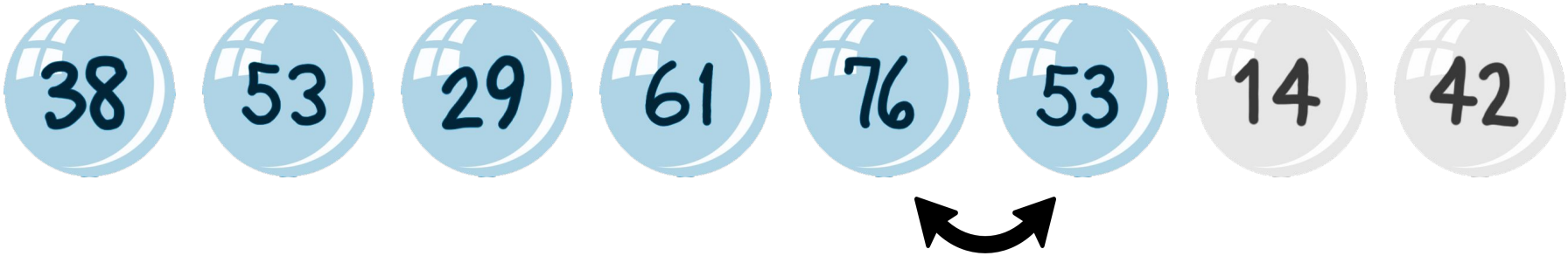
Bubble Sort

- Bubble sort in action:



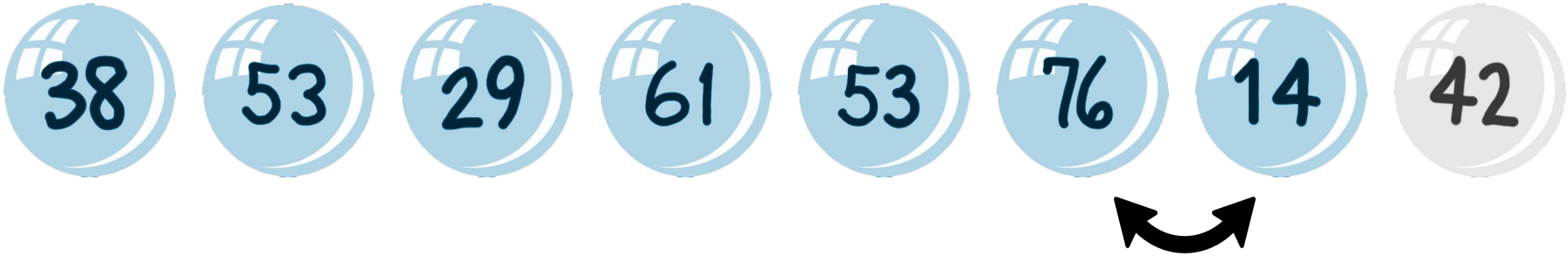
Bubble Sort

- Bubble sort in action:



Bubble Sort

- Bubble sort in action:

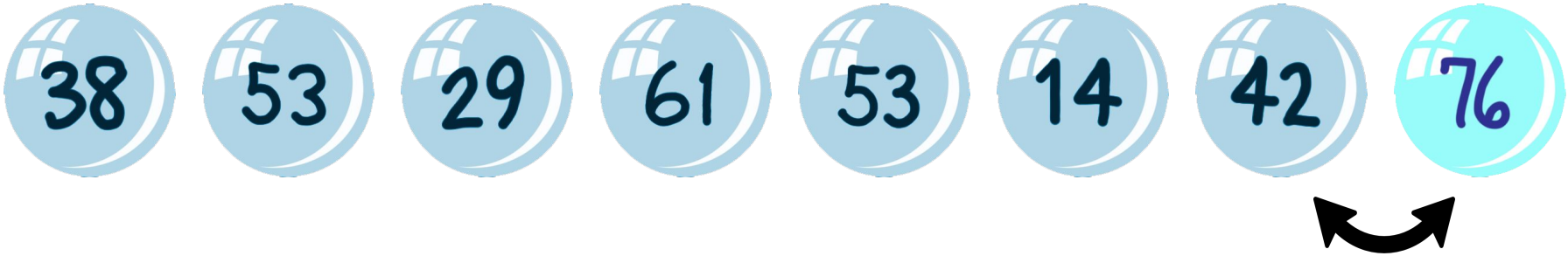


Bubble Sort

- Bubble sort in action:

Bubble Sort

- Bubble sort in action:



in this version of bubble sort, larger elements "bubble" to the top!

Bubble Sort

- Bubble sort in action:

Repeat this process with
the unsorted sublist
(excluding 76)

Bubble Sort (non-adaptive)

- Best case - $O(n^2)$
- Worst case - $O(n^2)$
- Average case - $O(n^2)$
- Memory consumption - O(1) extra
- Stable - yes

Bubble Sort (adaptive)

- Add a boolean flag - break out when it’s already sorted
  - Better for nearly sorted lists
- Pseudocode

```
For i in range [0 -> n) {
    Has_swapped := false
    For j in range (n -> i] {
        if (arr[j] > arr[j - 1]) {
            swap(arr[j], arr[j - 1]);
            Has_swapped := true
        }
    }
    If not has_swapped { break; }
}
```

Bubble Sort (adaptive)

- Best case - O(n)
- Worst case - $O(n^2)$
- Average case - $O(n^2)$
- Memory consumption - O(1) extra
- Stable - yes

Selection Sort

- Algorithm:
  - Divide input list into sublist of sorted items and sublist of unsorted items.
  - Find the smallest element in the unsorted sublist and exchange it with the leftmost unsorted element
  - Move the sublist boundaries one element to the right.

- Find the smallest element, swap with first position. Find the second smallest element, swap with the second position, etc.
- Good when auxiliary memory is limited - minimal copying!

Selection Sort

- Pseudocode

```
For i in range [0, n) {
    Smallest_element := arr[i];
    For j in range [i, n) {
        If arr[j] < arr[smallest_element]
            Smallest_element := j;
    }
    swap(arr[i], arr[smallest_element])
}
```

Selection Sort

- Selection sort in action:

Start with entire list as sublist of unsorted

Selection Sort

- Selection sort in action:

the gray shading indicates the minimum value seen so far for each traversal

Selection Sort

- Selection sort in action:

Find the minimum value in the unsorted sublist

Selection Sort

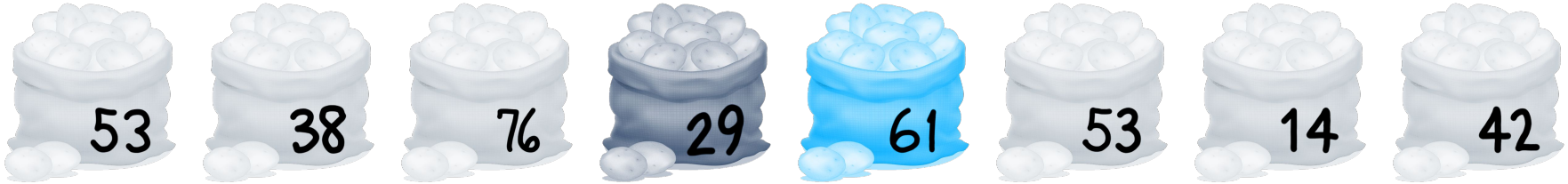
- Selection sort in action:

Selection Sort

- Selection sort in action:

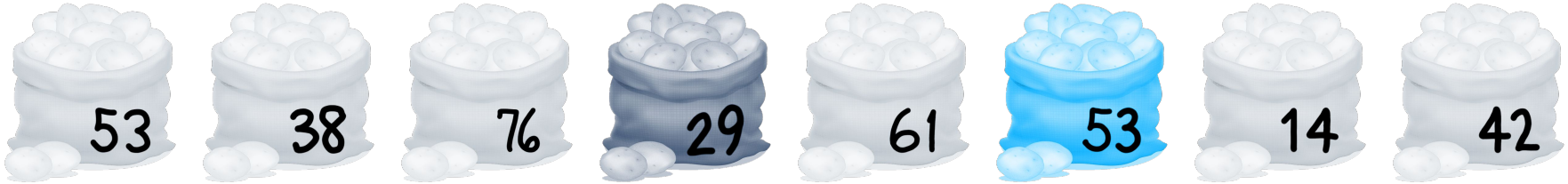
Selection Sort

- Selection sort in action:



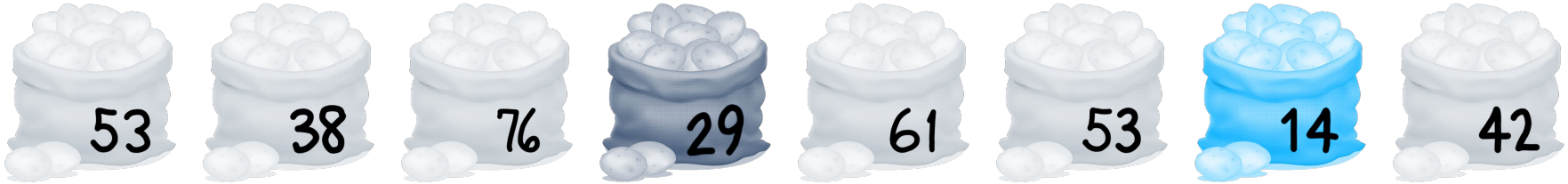
Selection Sort

- Selection sort in action:



Selection Sort

- Selection sort in action:

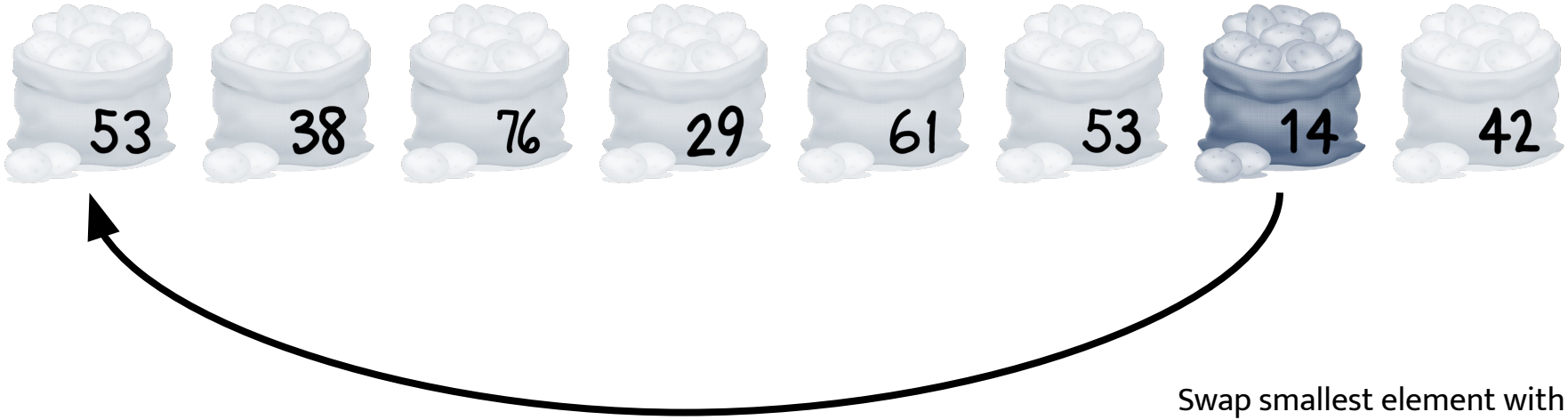


Selection Sort

- Selection sort in action:

Selection Sort

- Selection sort in action:



Swap smallest element with the leftmost element in the unsorted sublist

Selection Sort

- Selection sort in action:

14 is now locked into place in the sorted sublist

Selection Sort

- Selection sort in action:

Repeat with the unsorted sublist

Selection Sort

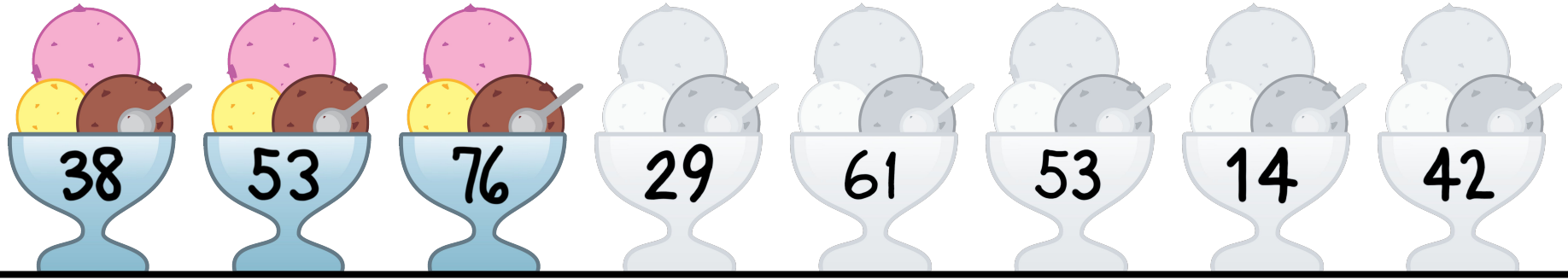
- Best case - $O(n^2)$
- Average case - $O(n^2)$
- Worst case - $O(n^2)$
- Memory Consumption - O(1) extra
- Stable - No

Insertion Sort (non-adaptive)

- Algorithm:
  - Divide input list into sublist of sorted items and sublist of unsorted items.
  - Take whatever item is next in the unsorted sublist and swap it with adjacent elements in the sorted sublist until the item is in its proper place in the sorted sublist.

Insertion Sort (non-adaptive)

- Insertion sort in action:

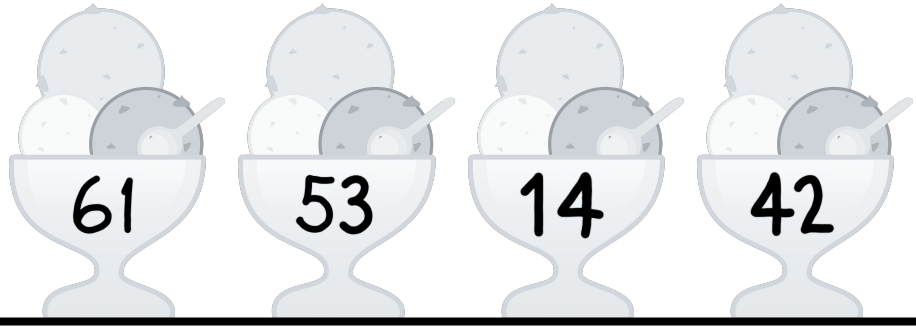


Sorted Sublist

Unsorted Sublist

Insertion Sort (non-adaptive)

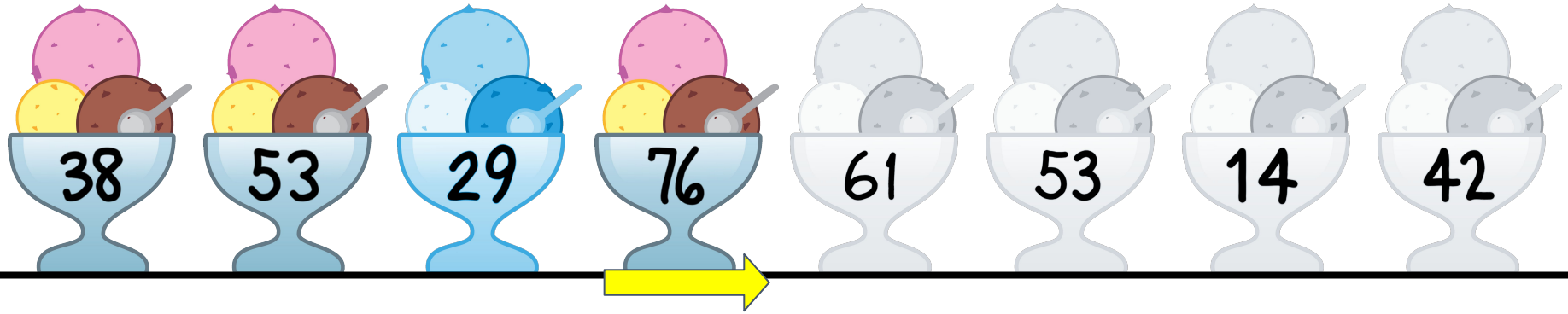
- Insertion sort in action:



Take the next element in the unsorted sublist

Insertion Sort (non-adaptive)

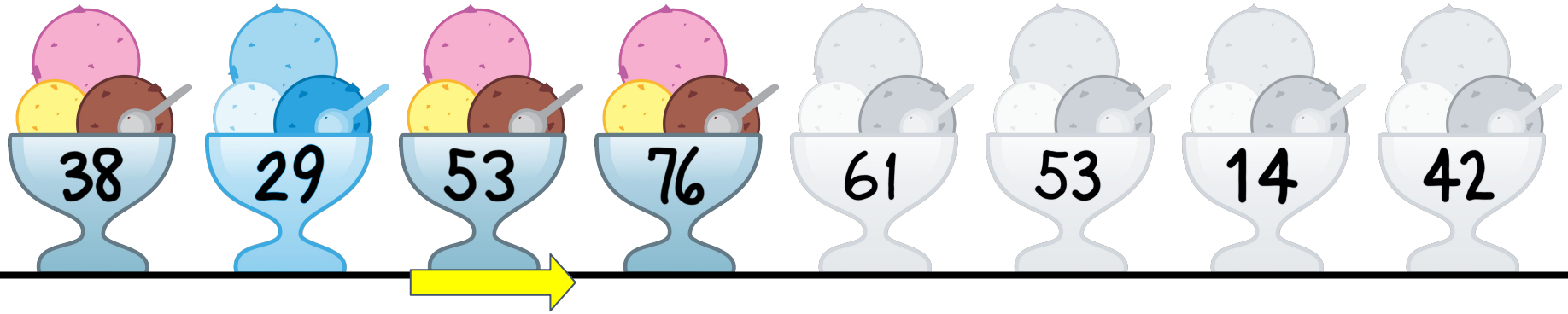
- Insertion sort in action:



Swap with adjacent element until in the correct position

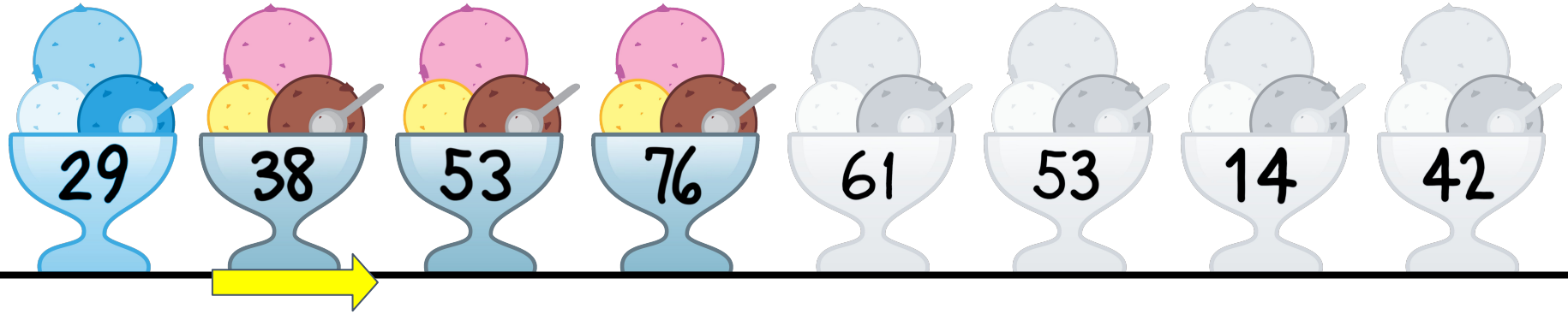
Insertion Sort (non-adaptive)

- Insertion sort in action:



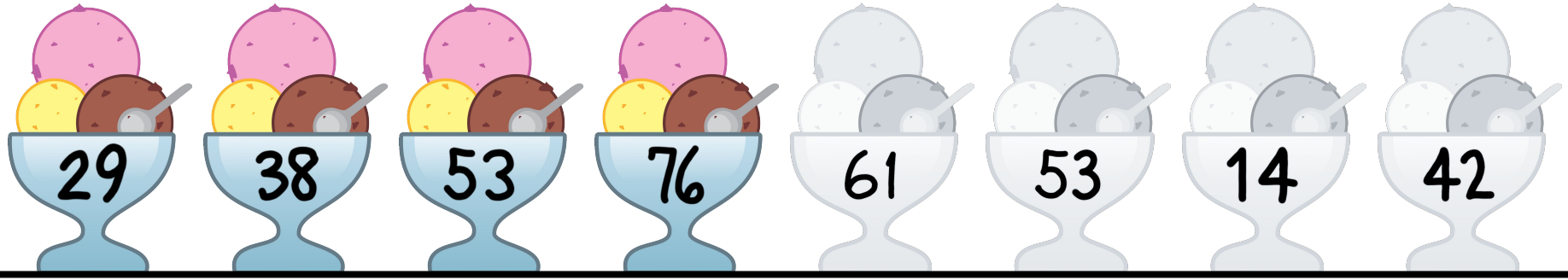
Insertion Sort (non-adaptive)

- Insertion sort in action:



Insertion Sort (non-adaptive)

- Insertion sort in action:



Insertion Sort (non-adaptive)

- Best case - $O(n^2)$
- Average case - $O(n^2)$
- Worst case - $O(n^2)$
- Memory Consumption - O(1) extra
- Stable - yes

Insertion Sort (non adaptive)

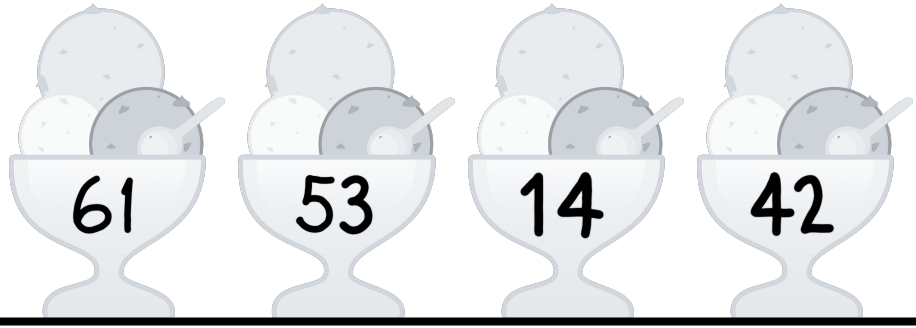
- Pseudocode

```
For i in range [0, n) {
    For j in range [i -> 0) {
        If (arr[j] < arr[j-1]) {
            swap(arr[j], arr[j-1])
        }
    }
}
```

- How can we make this adaptive?

Insertion Sort (adaptive)

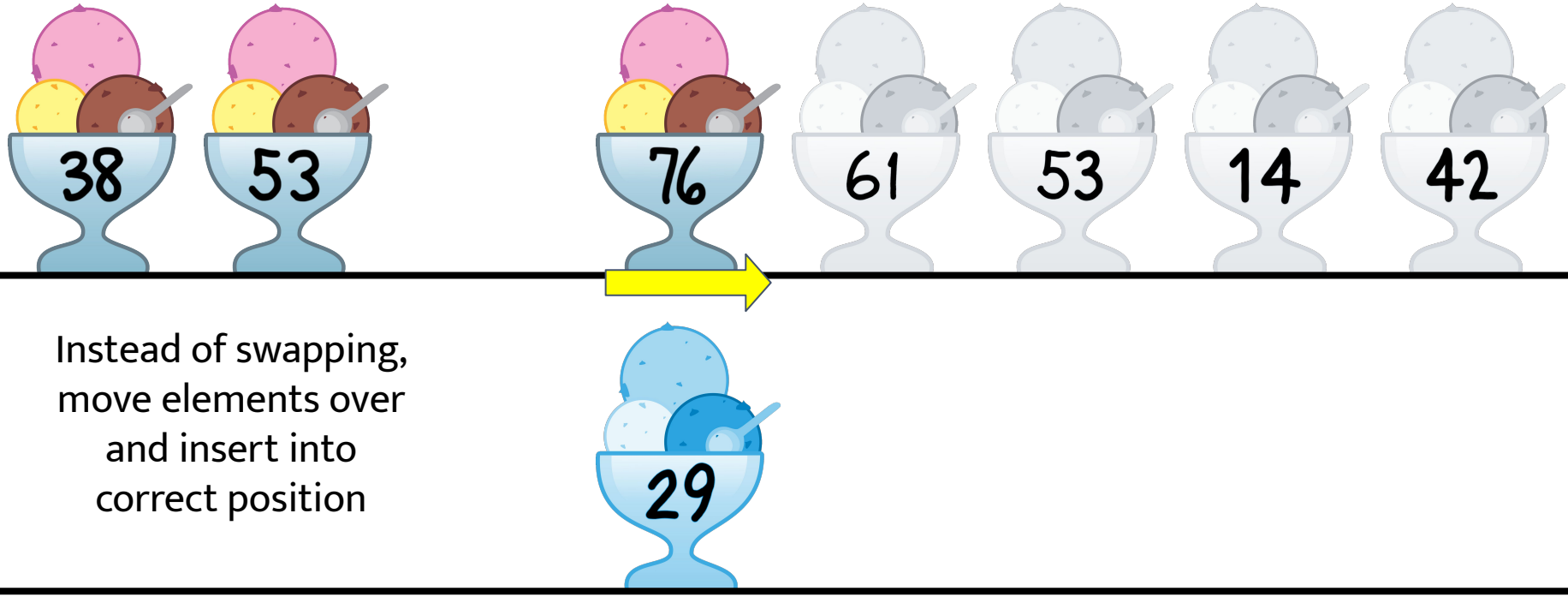
- Insertion sort in action:



Instead of swapping, move elements over and insert into correct position

Insertion Sort (adaptive)

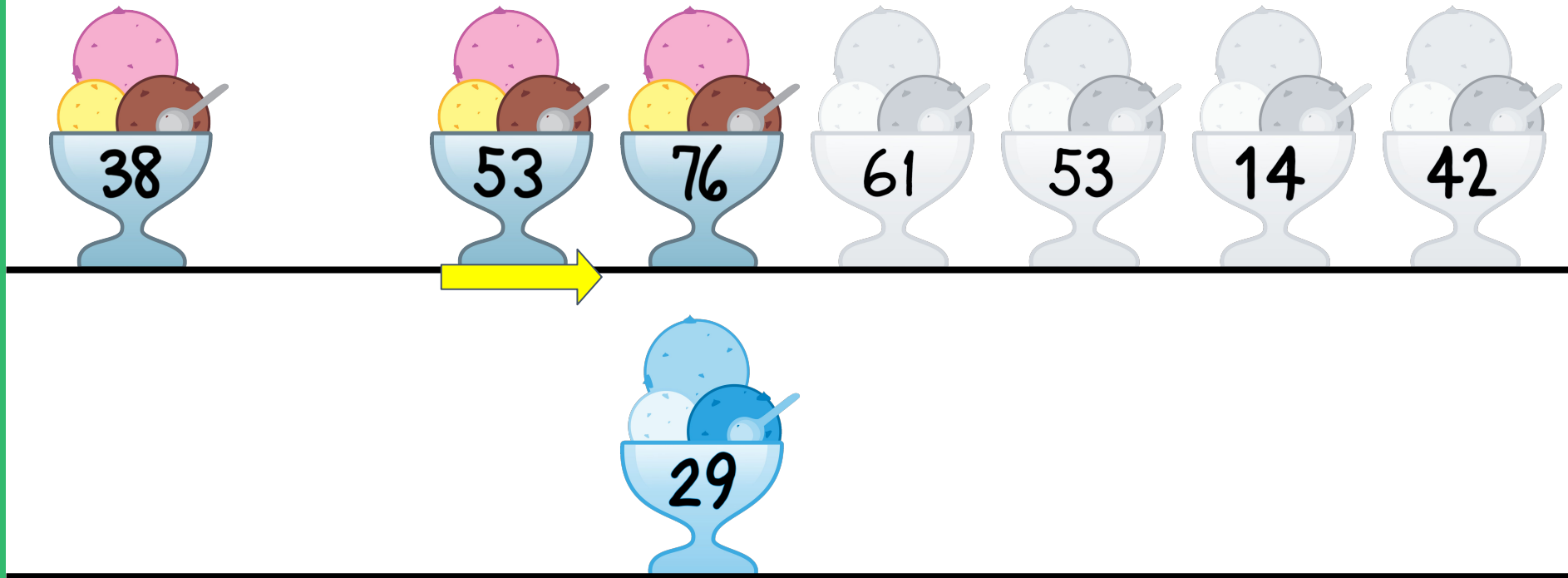
- Insertion sort in action:



Instead of swapping, move elements over and insert into correct position

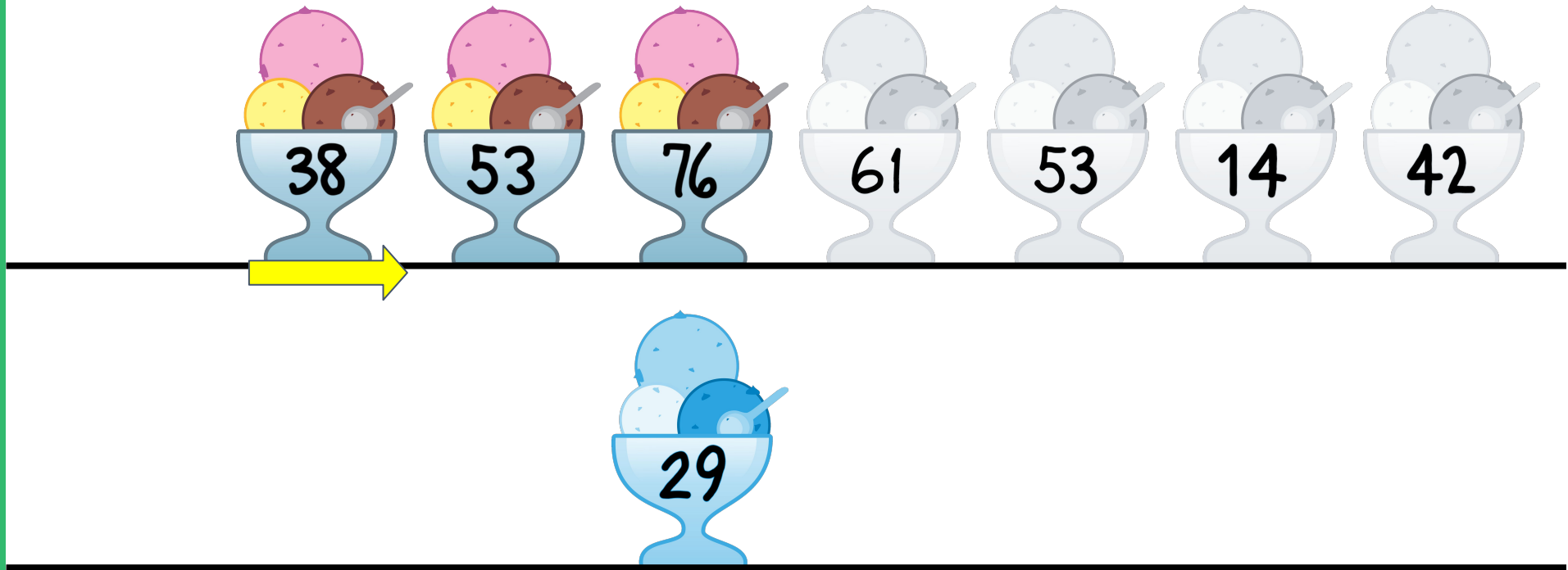
Insertion Sort (adaptive)

- Insertion sort in action:



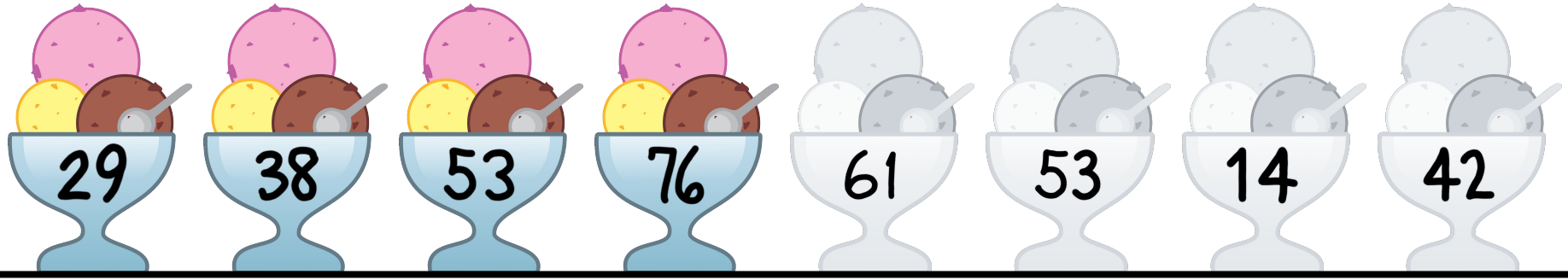
Insertion Sort (adaptive)

- Insertion sort in action:



Insertion Sort (adaptive)

- Insertion sort in action:



Insertion Sort (adaptive)

- Best case - O(n)
- Average case - $O(n^2)$
- Worst case - $O(n^2)$
- Memory Consumption - O(1) extra
- Stable - yes

- One of the best sorting algorithms on small input sizes
- Good for nearly sorted

Stability

- Given a list `[{3}, {3}, {3}, {1}, {2}]`
- Imagine as `[{3, 1}, {3, 2}, {3, 3}, {1, 4}, {2, 5}]`

- Sort the above list so that with respect to the set of pairs with the first element as a 3, the list of second elements is sorted.

- For example
  - `[{1, 4}, {2, 5}, {3, 1}, {3, 2}, {3, 3}]` is stable
  - `[{1, 4}, {2, 5}, {3, 3}, {3, 2}, {3, 1}]` is unstable
  - `[{1, 4}, {2, 5}, {3, 3}, {3, 1}, {3, 2}]` is unstable

- Which algorithm(s) could we use?

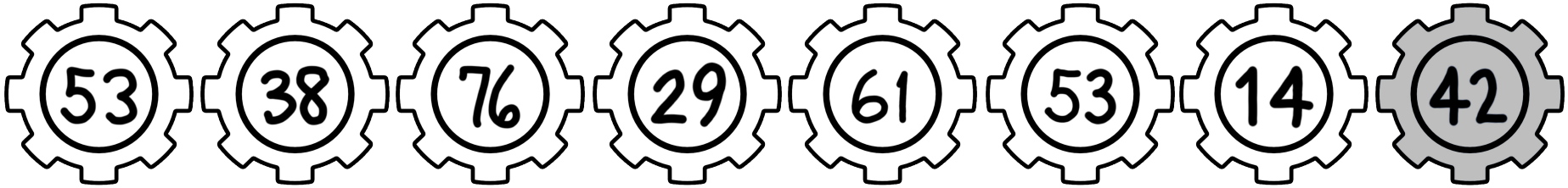
Quicksort

- Algorithm:
  - Pick a pivot from the list.
  - Elements less than pivot come before pivot
  - Elements greater than or equal to pivot come after it
  - After partitioning, the pivot is in the final position.
  - Recursively sort the sub-list of elements with smaller values and the sub-list of elements with greater values.

- Partition the array by putting an element in the spot where it would go if the array were sorted

Quicksort

- Quicksort in action (with last element as pivot):



L

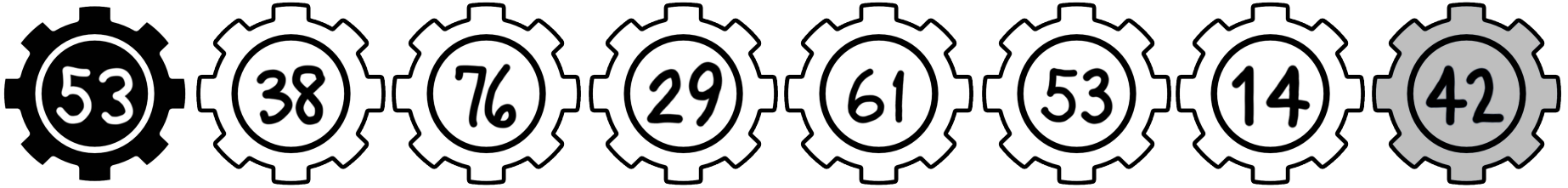
Increment L until you find an element greater than pivot

R

Decrement R until you find an element less than pivot

Quicksort

- Quicksort in action (with last element as pivot):



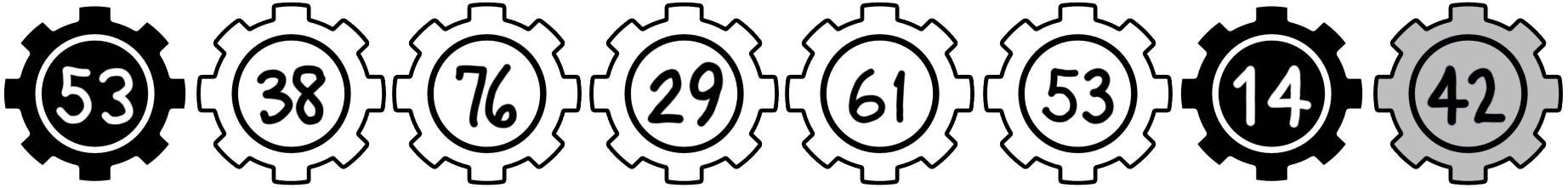
L

R

> 42

Quicksort

- Quicksort in action (with last element as pivot):



L

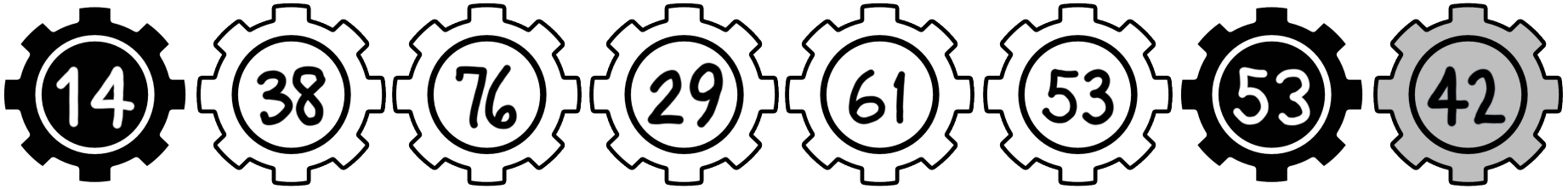
> 42

R

< 42

Quicksort

- Quicksort in action (with last element as pivot):



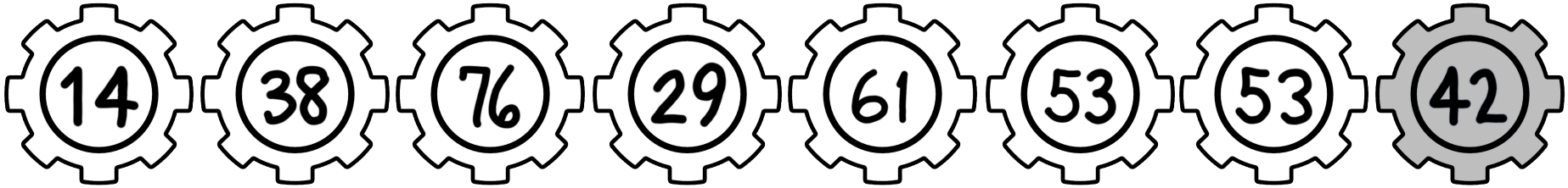
L

R

Swap L and R then repeat

Quicksort

- Quicksort in action (with last element as pivot):



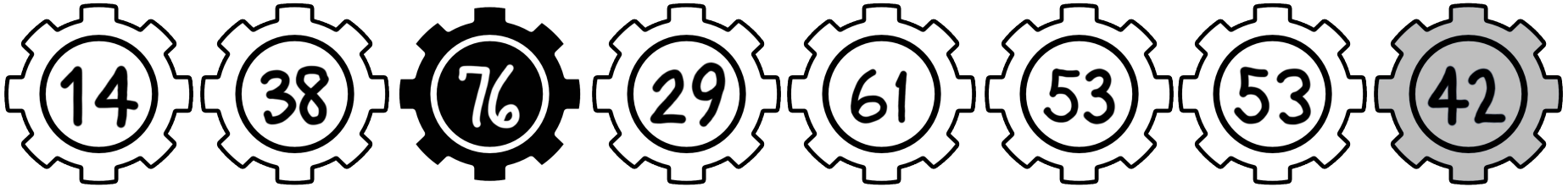
L

R

< 42

Quicksort

- Quicksort in action (with last element as pivot):



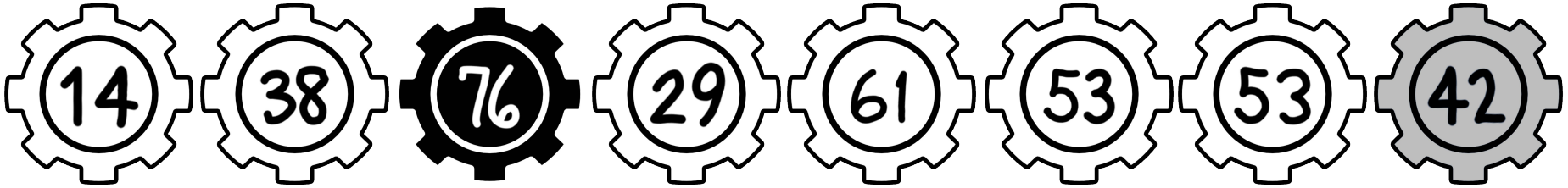
L

R

> 42

Quicksort

- Quicksort in action (with last element as pivot):



L

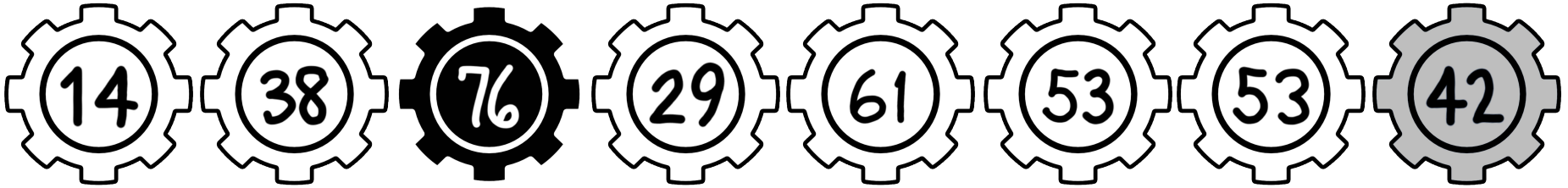
> 42

R

> 42

Quicksort

- Quicksort in action (with last element as pivot):



L

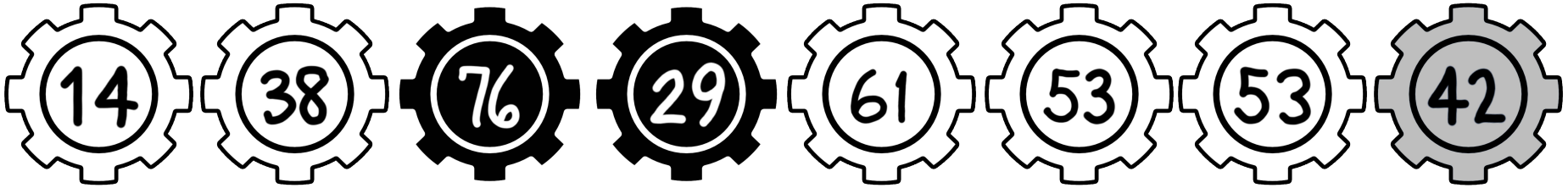
> 42

R

> 42

Quicksort

- Quicksort in action (with last element as pivot):

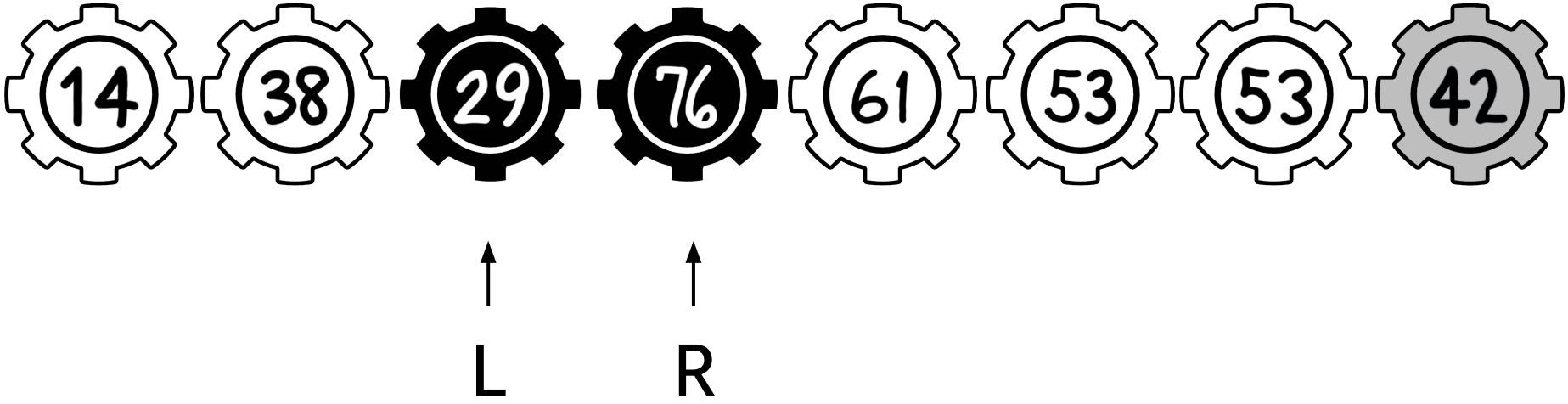


L R

> 42 < 42

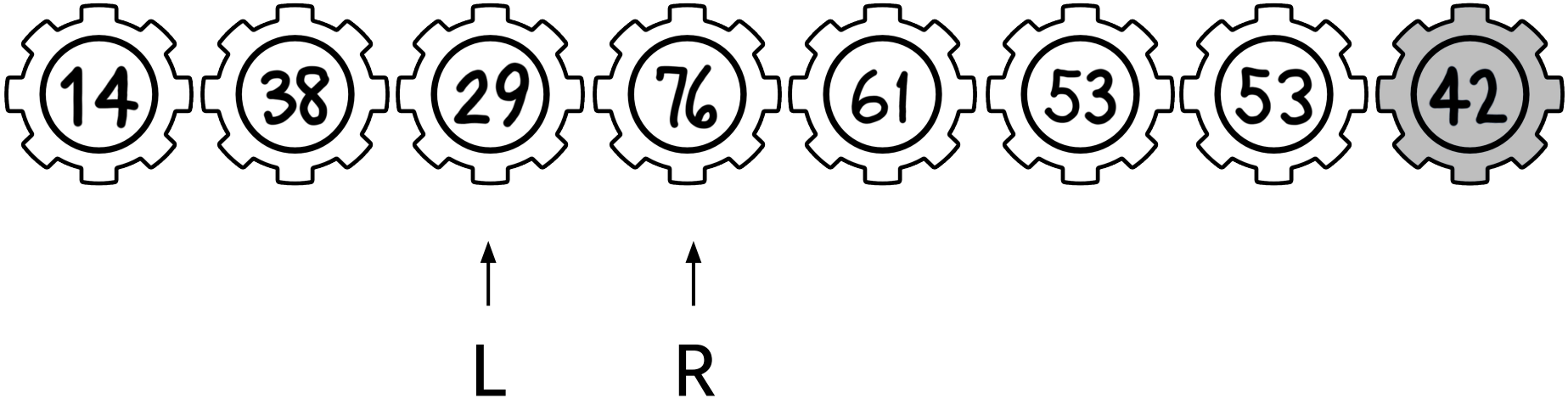
Quicksort

- Quicksort in action (with last element as pivot):



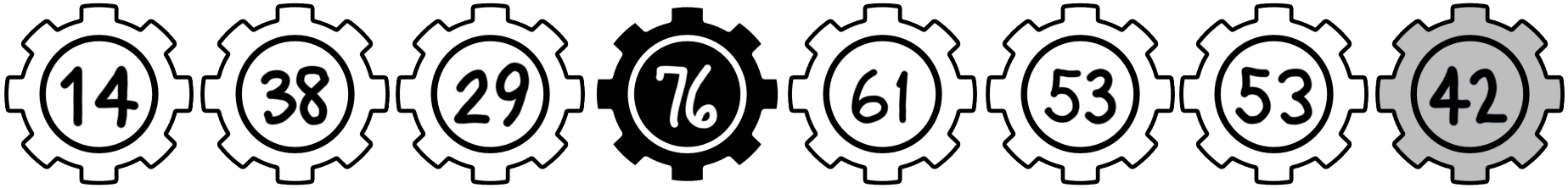
Quicksort

- Quicksort in action (with last element as pivot):



Quicksort

- Quicksort in action (with last element as pivot):



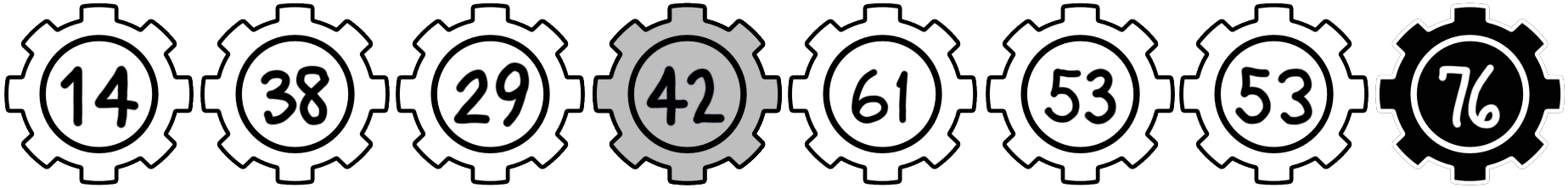
R

L

left and right are equal...
swap with the pivot!

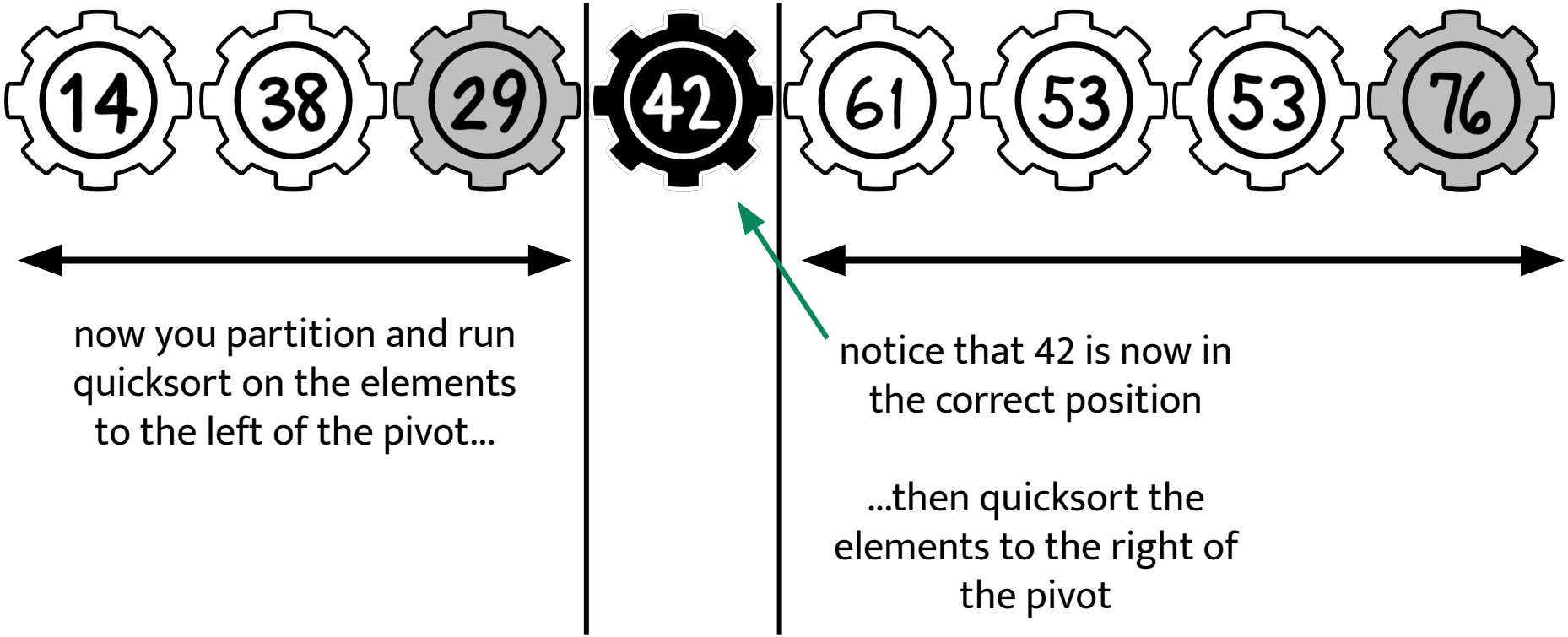
Quicksort

- Quicksort in action (with last element as pivot):

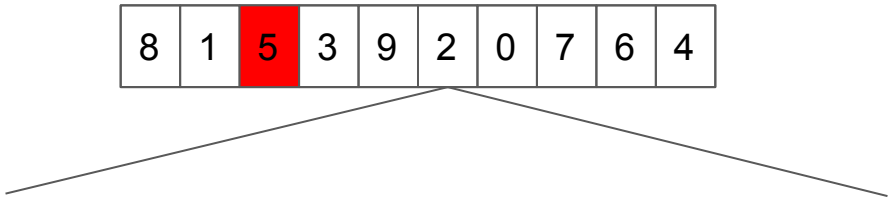


Quicksort

- Quicksort in action (with last element as pivot):

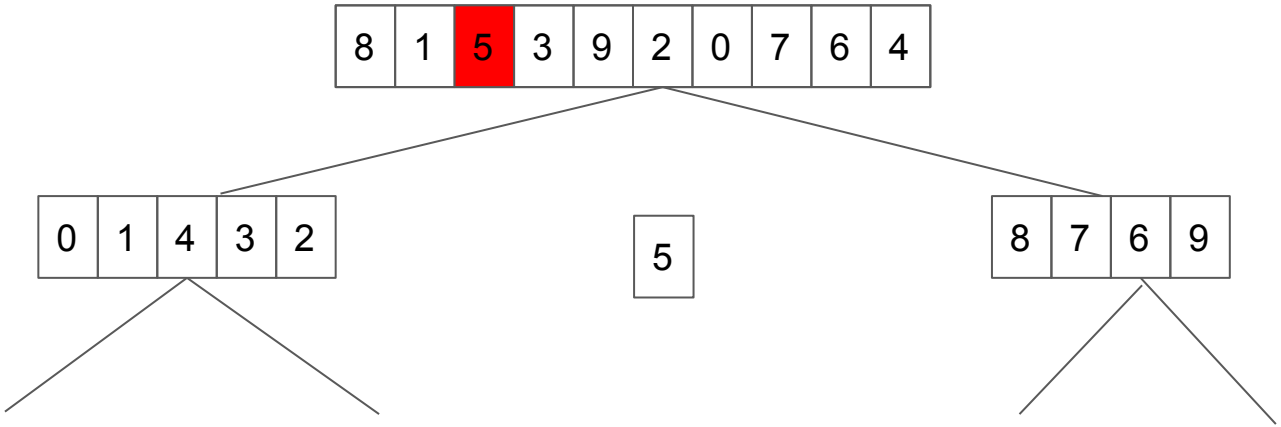


Quicksort Complexity

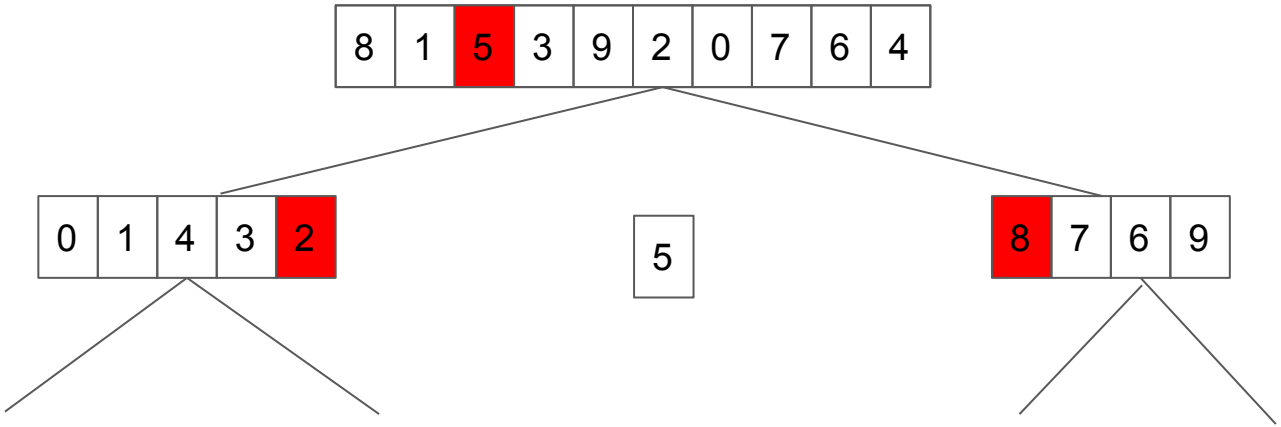


What happens when we select a pivot that always divides the list in half?

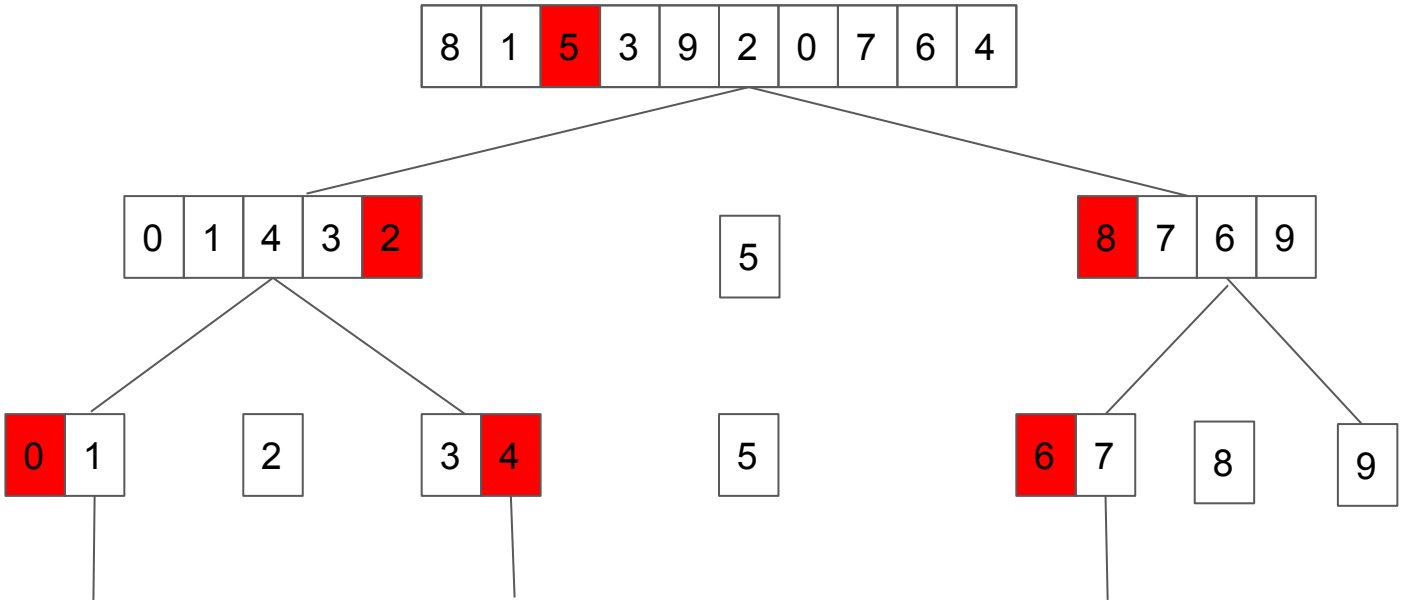
Quicksort Complexity



Quicksort Complexity

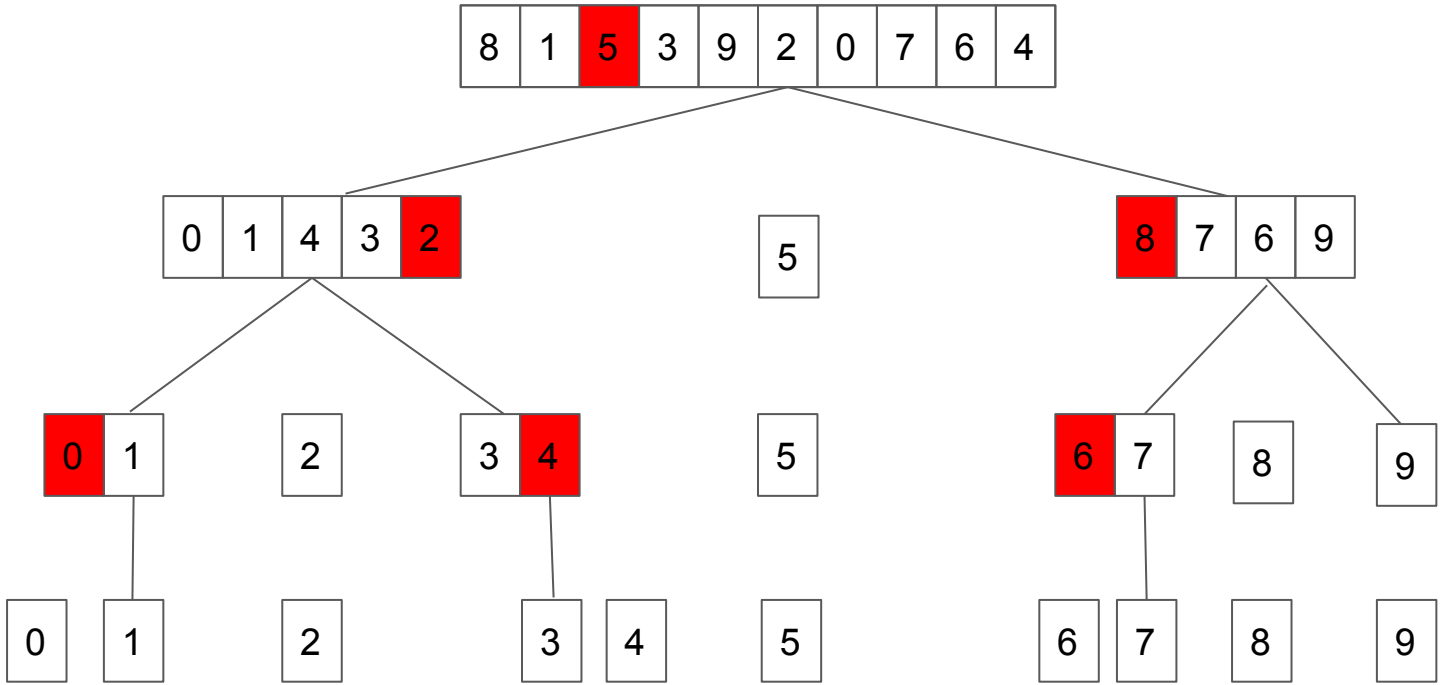


Quicksort Complexity

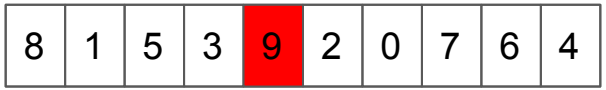


Quicksort Complexity

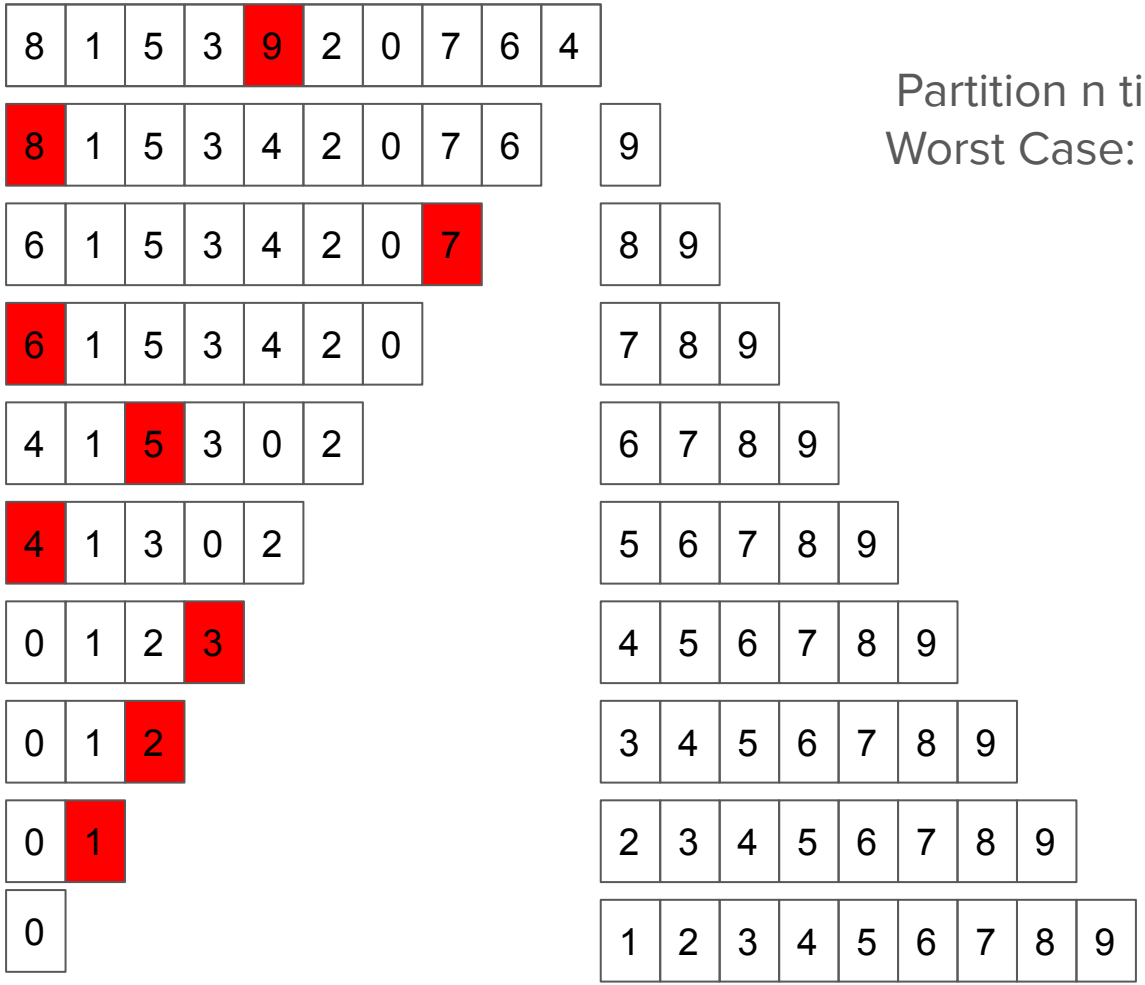
Partition log n times!
Best Case: O(n log n)



Quicksort Complexity



What happens when we select the largest (or smallest) element as the pivot?



Partition n times!
Worst Case: $O(n^2)$

Quicksort

- Best case - O(n log n)
- Worst case - $O(n^2)$
- Average case - O(n log n)
- Memory consumption - O(log n) extra
- Stable - no

Quicksort

- What if you can find the median in O(n) time?
- How will the complexity of Quick sort change?

Quicksort

- What if you can find the median in O(n) time?
- How will the complexity of Quick sort change?
- Complexity: O(n) + O(n log n) = **O(n log n)**

Quicksort

- What if you can find the median in O(n) time?
- How will the complexity of Quick sort change?
- Complexity: **O(n log n)**

- What if in a parallel universe, you find a function that gives you the median in O(1) time
- How will the complexity of Quick sort change?

Quicksort

- What if you can find the median in O(n) time?
- How will the complexity of Quick sort change?
- Complexity: **O(n log n)**

- What if in a parallel universe, you find a function that gives you the median in O(1) time
- How will the complexity of Quick sort change?
- Complexity: O(1) + O(n log n) = **O(n log n)**

Priority Queues/Heaps

Priority Queues

- An abstract container type that supports two operations:
  1. Insert a new item
  2. Remove the item with the **largest key**
- There are a number of different possible implementations: binary heap, pairing heap, etc.

STL Priority Queue

- 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 Queue - Using One

```
struct Request {
    int time_of_arrival;
    bool urgent;
    std::string owner;
};

struct Compy {
    bool operator()(const Request& x, const Request& y) const {
        if (y.urgent && !x.urgent) {
            return true; // y has higher priority
        }
        if (x.urgent && !y.urgent) {
            return false; // x has higher priority
        }
        // if y arrived before x (its time of arrival is smaller)
        // then y has higher priority
        return x.time_of_arrival > y.time_of_arrival;
    }
};
```

```
std::priority_queue<
     Request,
     std::vector<Request>,
     Compy
> my_pq;

void add_request(const Request
&r) {
     my_pq.push(r);
}
Request next_request() {
     assert(!my_pq.empty());
     Request r = my_pq.top();
     my_pq.pop();
     return r;
}
```

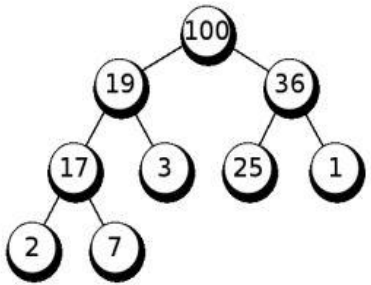
Binary Heap—Properties

- Binary tree with the following attributes:
  - Each node has an equal or higher priority to the value 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
- A ***max-heap*** (std::less) is a tree with the property that the value at every node is at least as large (>=) as its children's values
- ***Min-heap*** (std::greater): every node is at least as small (<=) as its children's values

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

Formulas are slightly different for 1-indexing!

Maintaining the Heap Property

- What if the priority of an item **increases**?
- We need to fix from the bottom up: **fixUp()**
- Swap the node with its parent, moving up until:
    1. We reach the root
    2. We reach a parent with a larger or equal key
- What is the complexity of **fixUp()**?
    - O(number of levels in the heap) = O(logn)

Heap Insertion

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

1. Insert the new item into the bottom of the heap
2. Call **fixUp()** on the newly inserted item

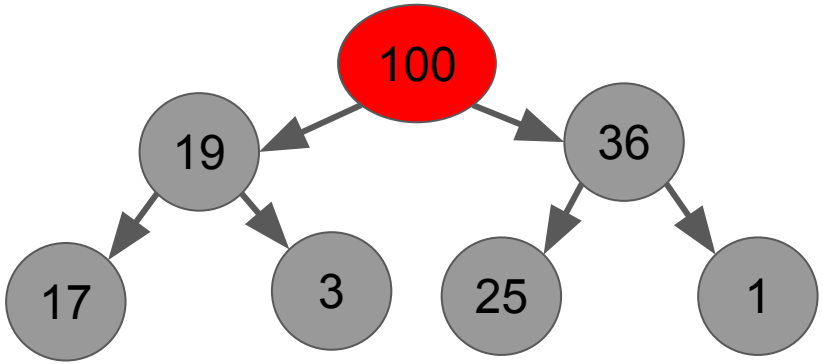
Maintaining the Heap Property

- What if the priority of an item **decreases**?
- We need to fix from the top down: **fixDown()**
- Swap the 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
- What is the complexity of **fixDown()**?
    - O(number of levels in the heap) = O(logn)

Heap Removal

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

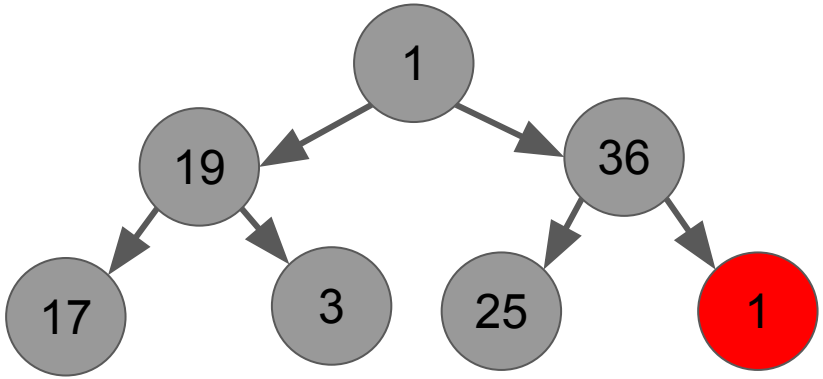
1. Remove the root item by replacing it with the last element in the heap
2. Delete the last element in the heap
3. Call **fixDown()** on the element that is now in the root position



Heap Removal

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

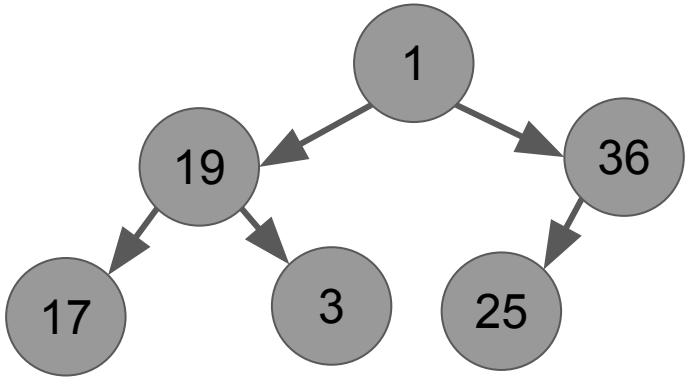
1. Remove the root item by replacing it with the last element in the heap
2. Delete the last element in the heap
3. Call **fixDown()** on the element that is now in the root position



Heap Removal

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

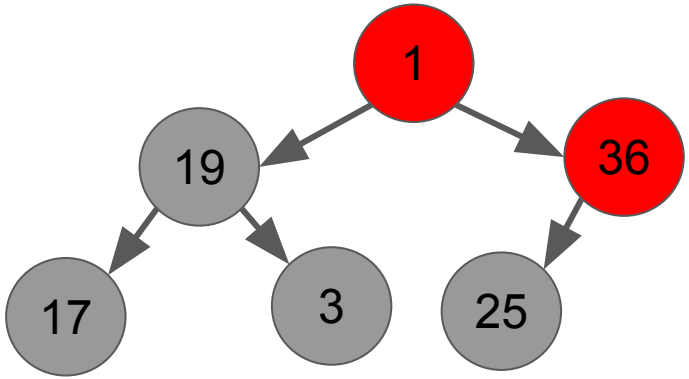
1. Remove the root item by replacing it with the last element in the heap
2. Delete the last element in the heap
3. Call **fixDown()** on the element that is now in the root position



Heap Removal

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

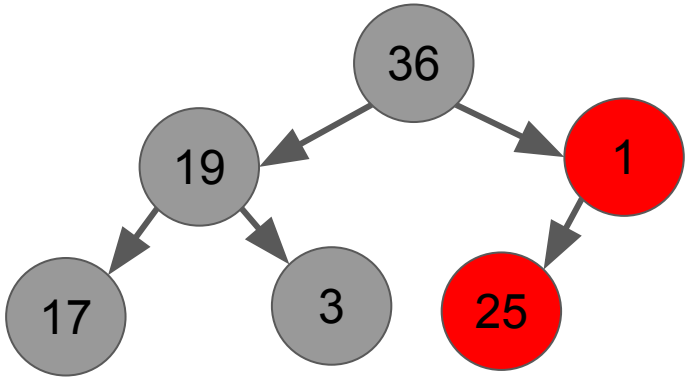
1. Remove the root item by replacing it with the last element in the heap
2. Delete the last element in the heap
3. Call **fixDown()** on the element that is now in the root position



Heap Removal

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

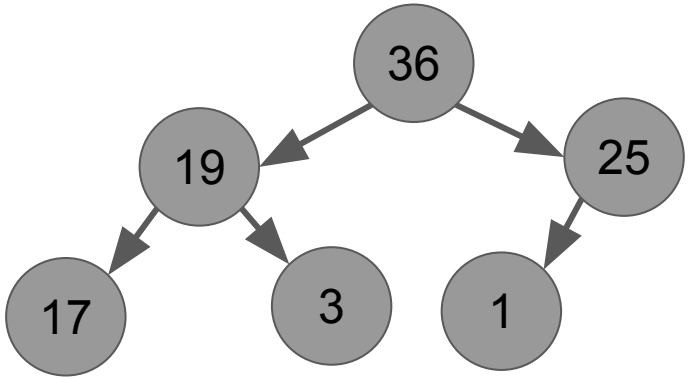
1. Remove the root item by replacing it with the last element in the heap
2. Delete the last element in the heap
3. Call **fixDown()** on the element that is now in the root position



Heap Removal

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

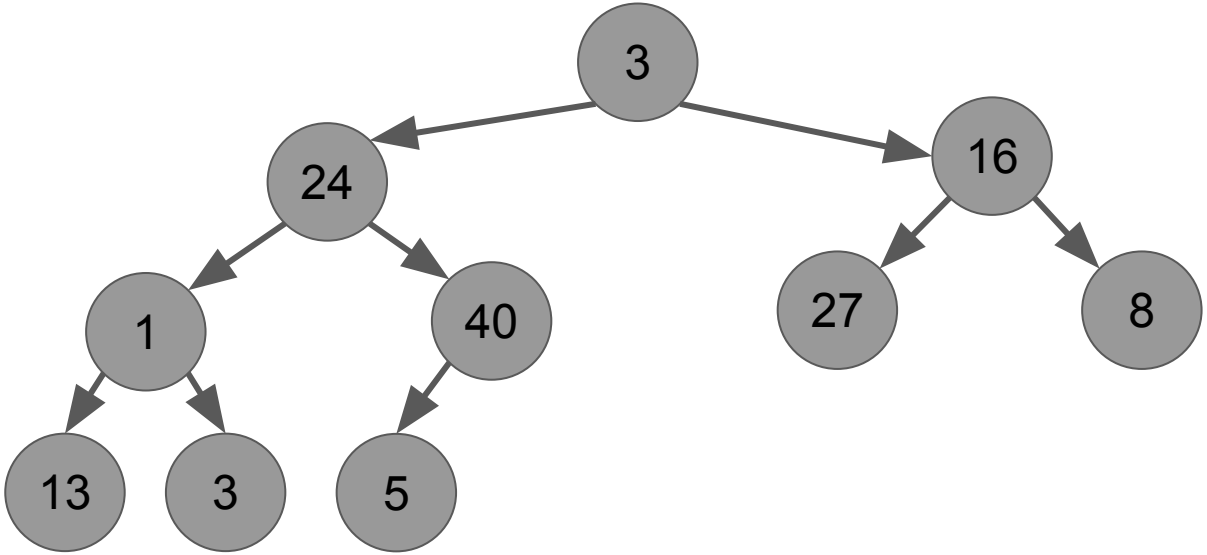
1. Remove the root item by replacing it with the last element in the heap
2. Delete the last element in the heap
3. Call **fixDown()** on the element that is now in the root position



Heapify

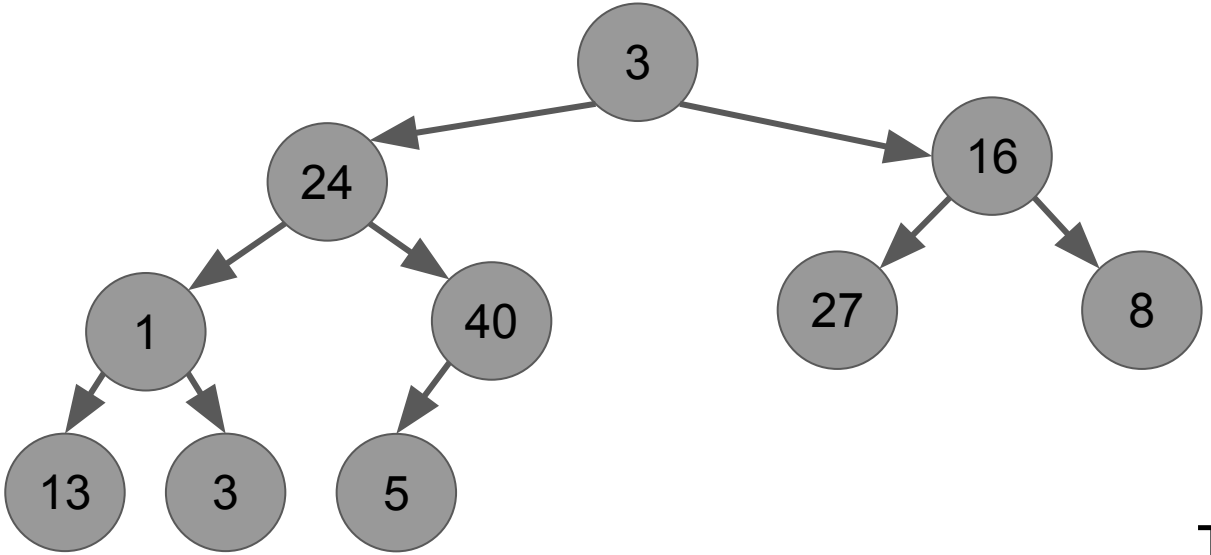
- Called on an unsorted vector
- Different from inserting one item at a time!
- Calls fixDown() from bottom-up
- Saves time on the last n/2 nodes
- Done when you use the STL’s range-based constructor

Using fixDown: 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 fixUp: Less Efficient

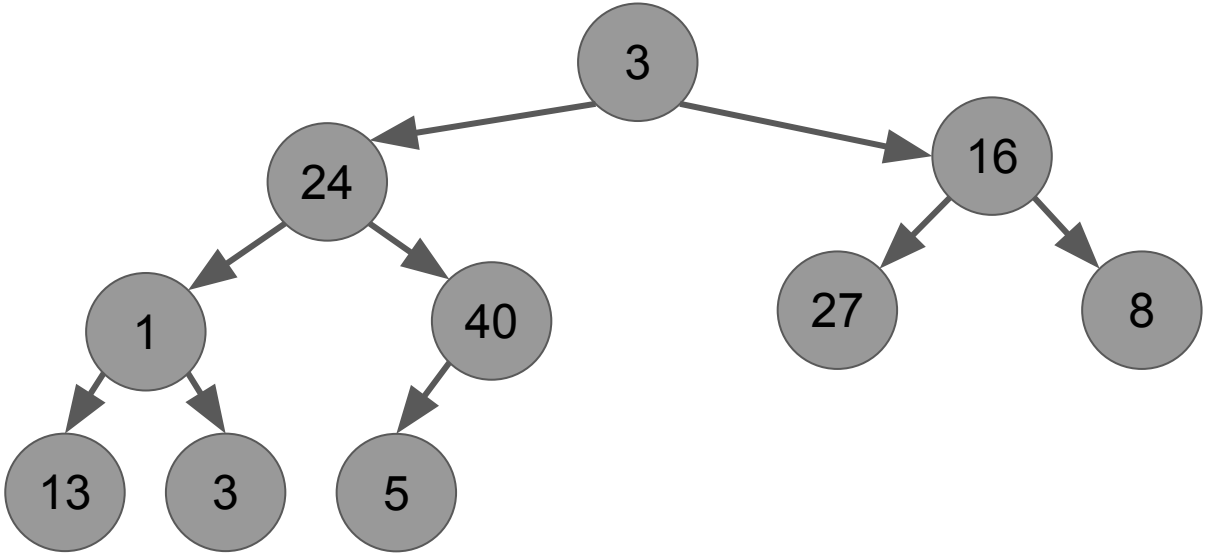


Depth	Max #Nodes
0	1
1	2
2	4
…	
Log(n)	n/2

Top-down fix up!

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 fixUp: Less Efficient

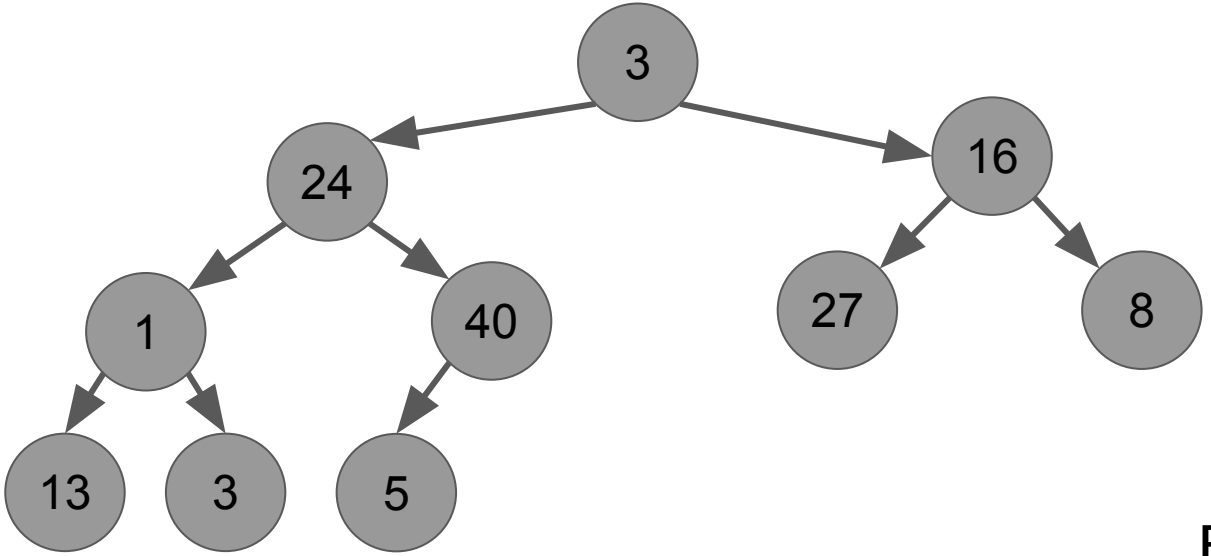


Depth	Max #Nodes
0	1
1	2
2	4
…	
Log(n)	n/2

Key idea:
Complexity of fixing the position of one item is O(the height 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 fixDown: More Efficient

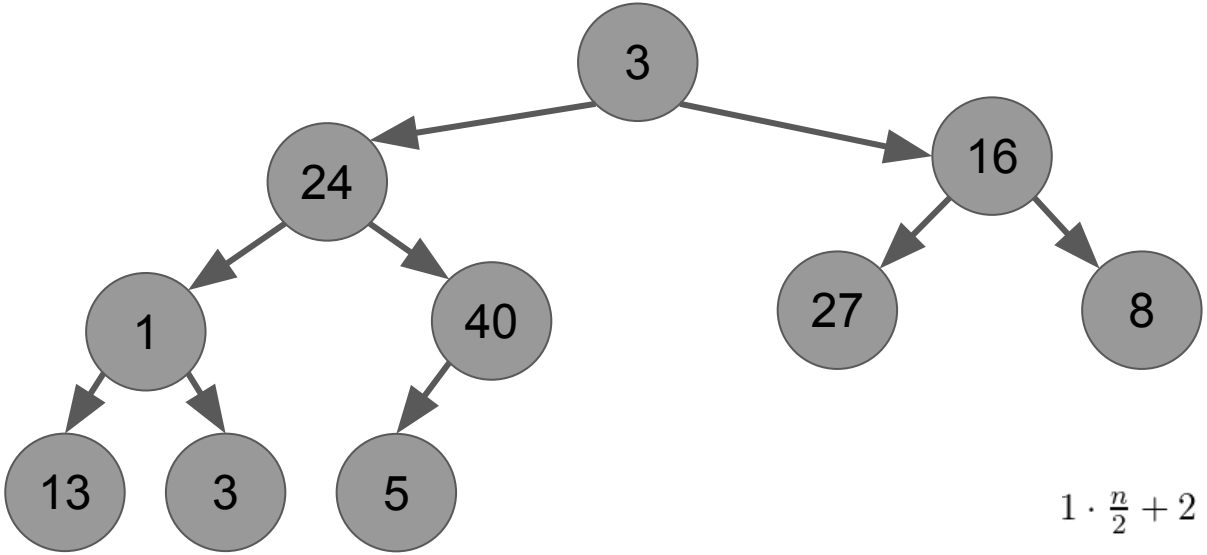


Height	Max #Nodes
Log(n)	1
…	
2	n/8
1	n/4
0	n/2

Bottom-up fix down!

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 fixDown: More Efficient



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

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

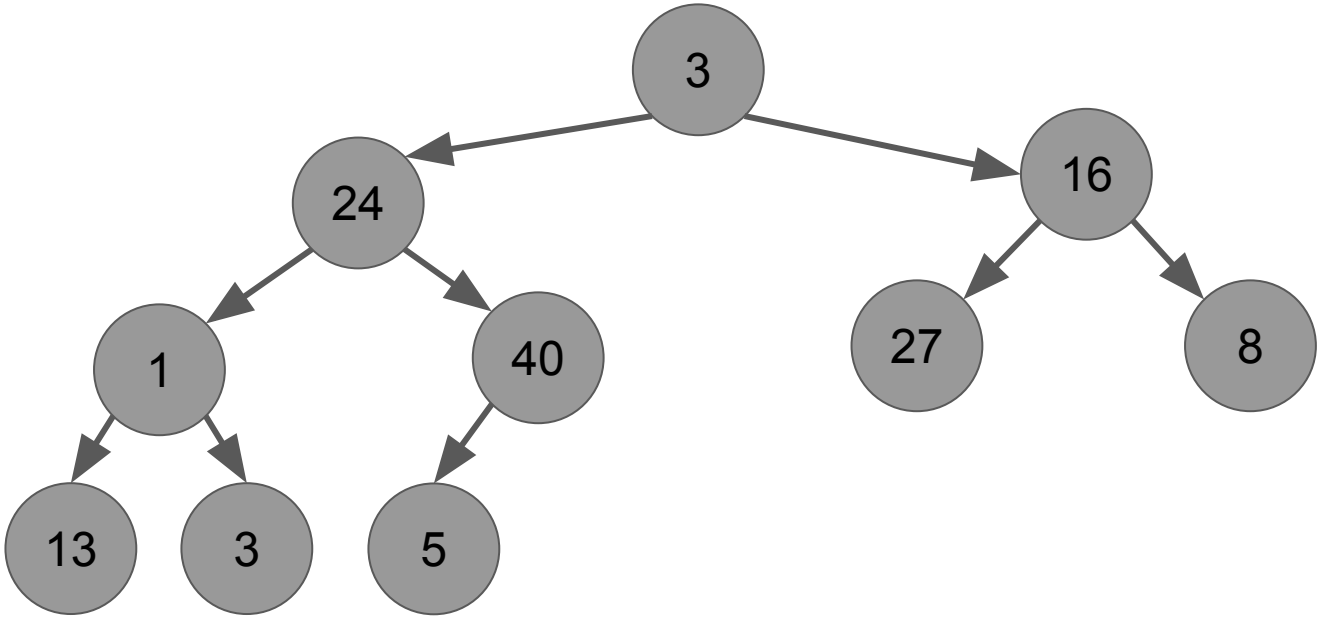
$$\leq n \left[\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots \right]$$

$$= n \cdot 2$$

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

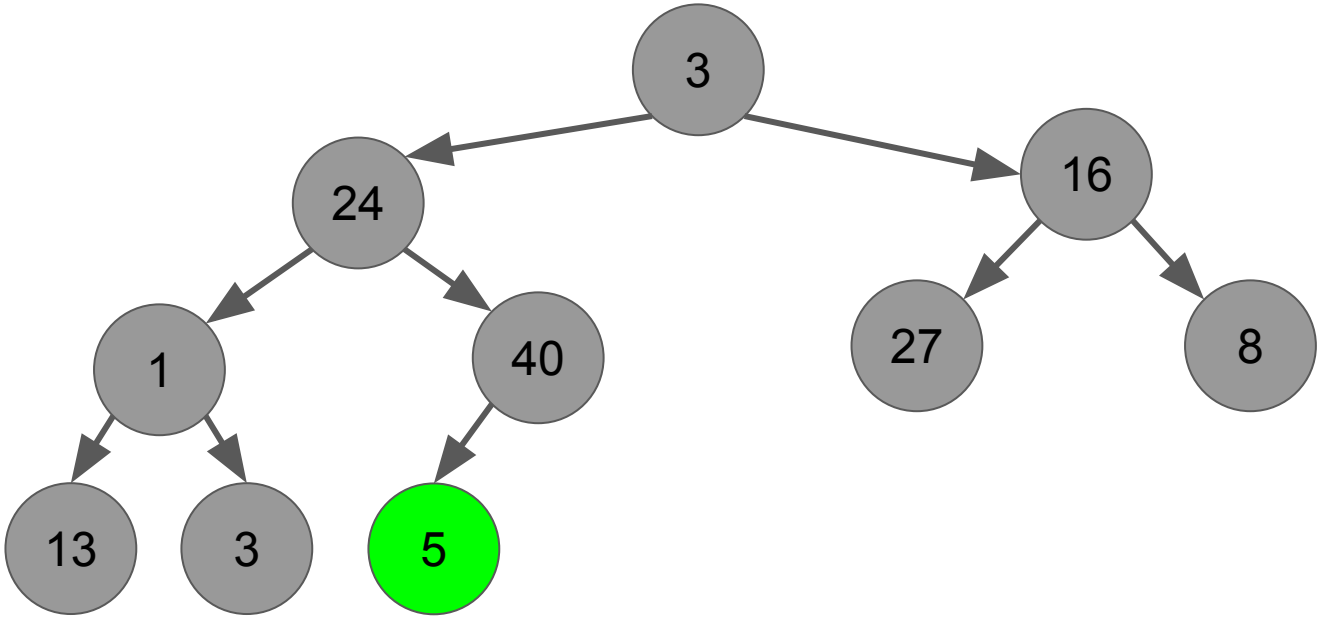
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



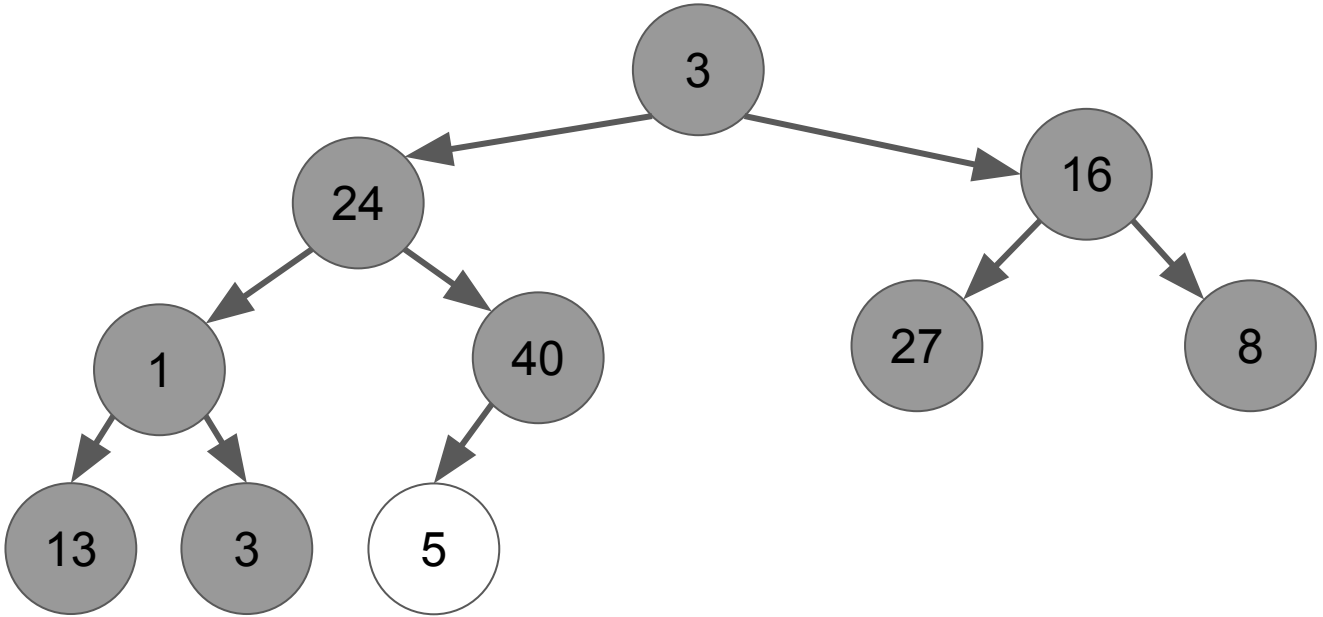
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



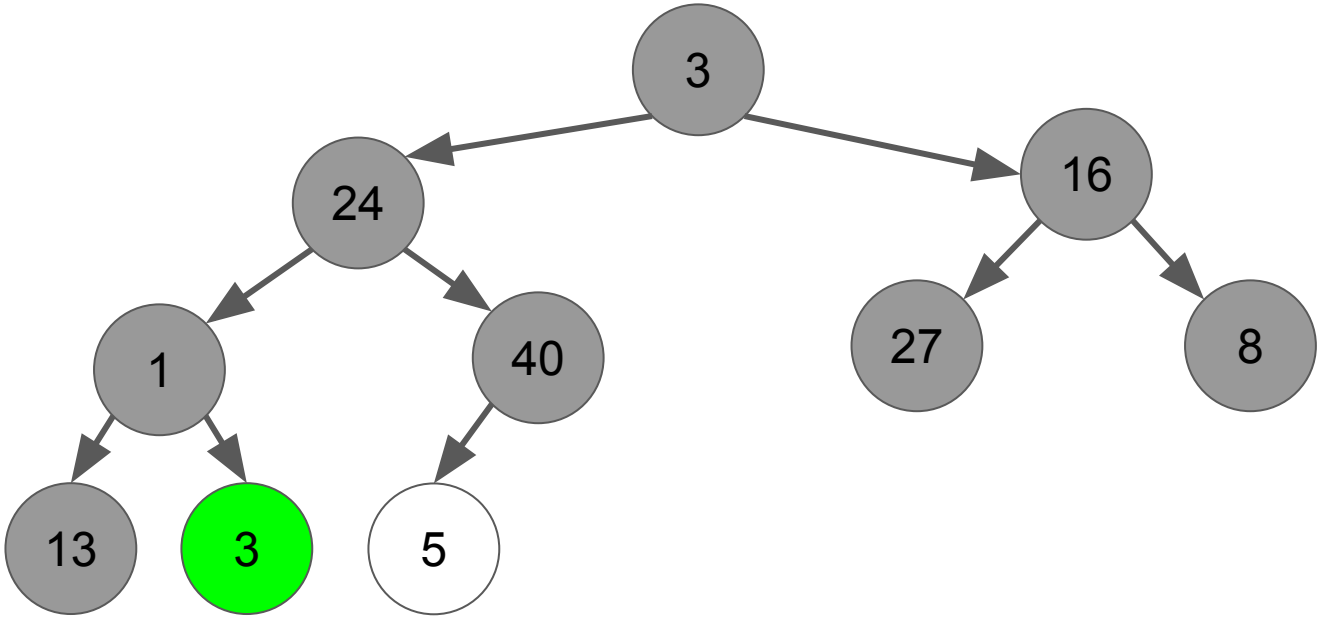
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



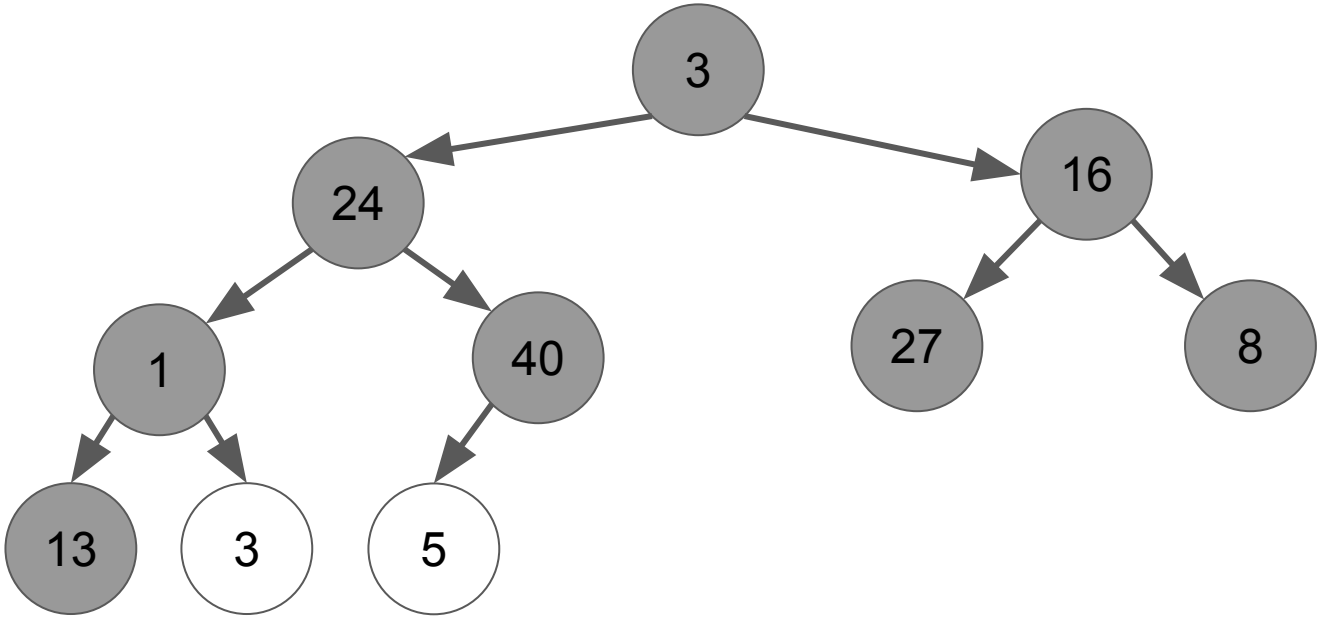
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



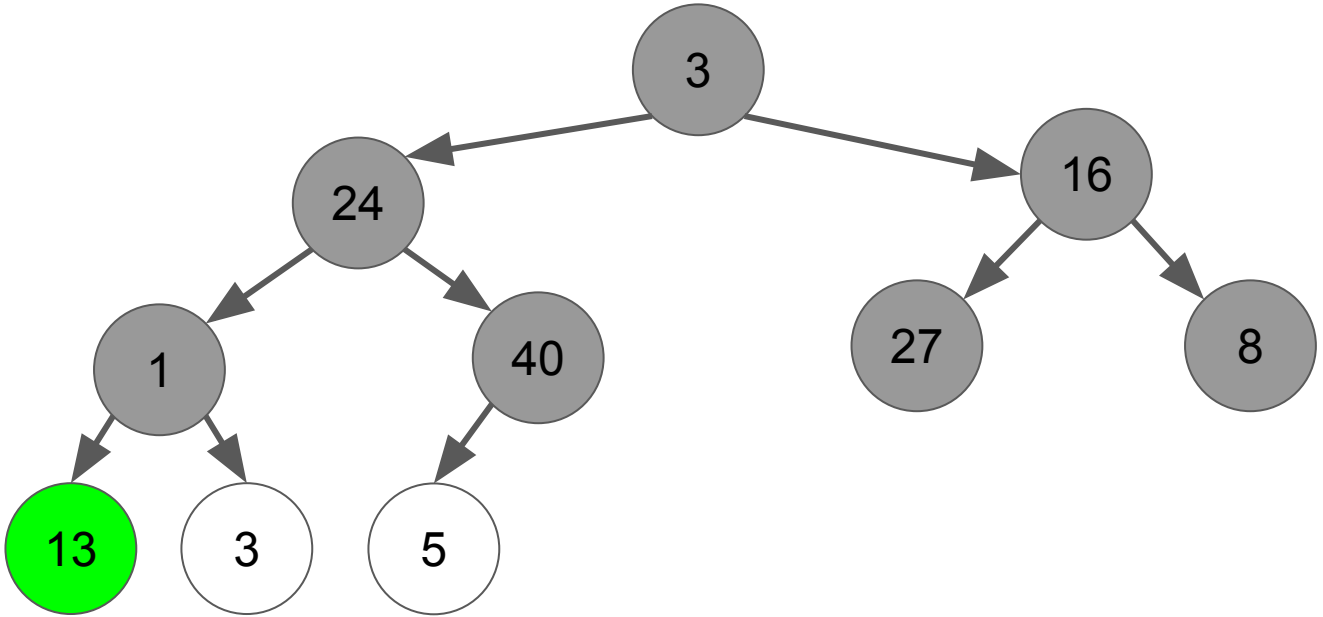
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



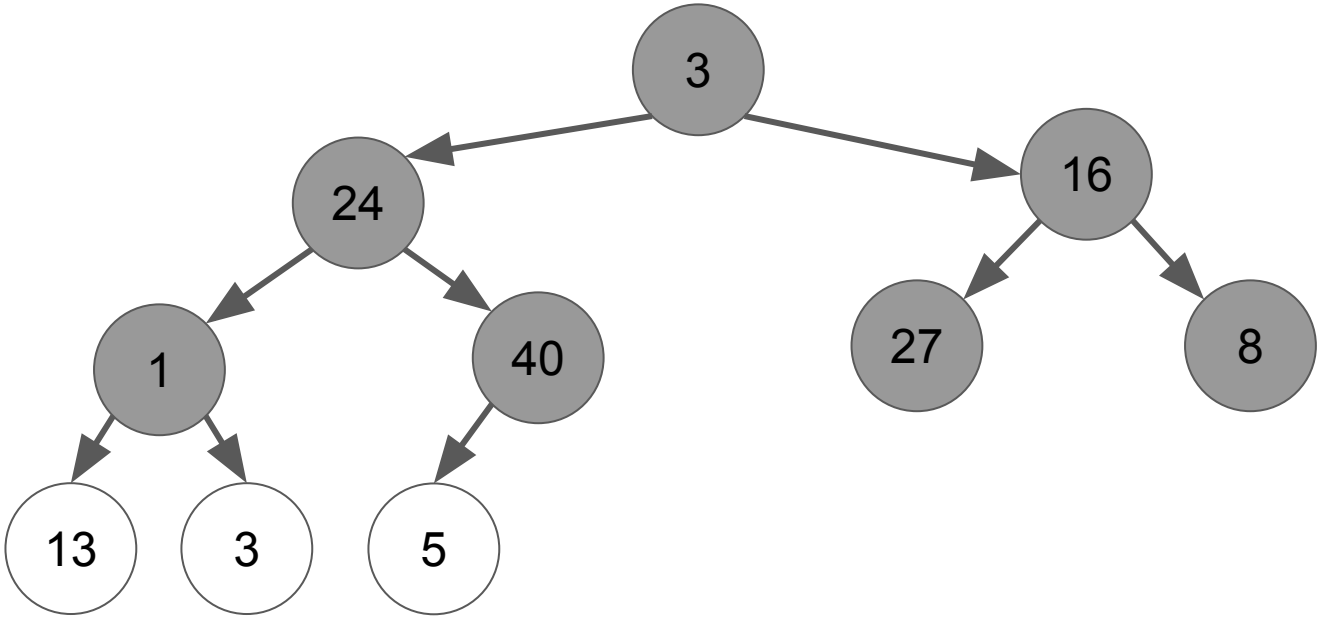
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



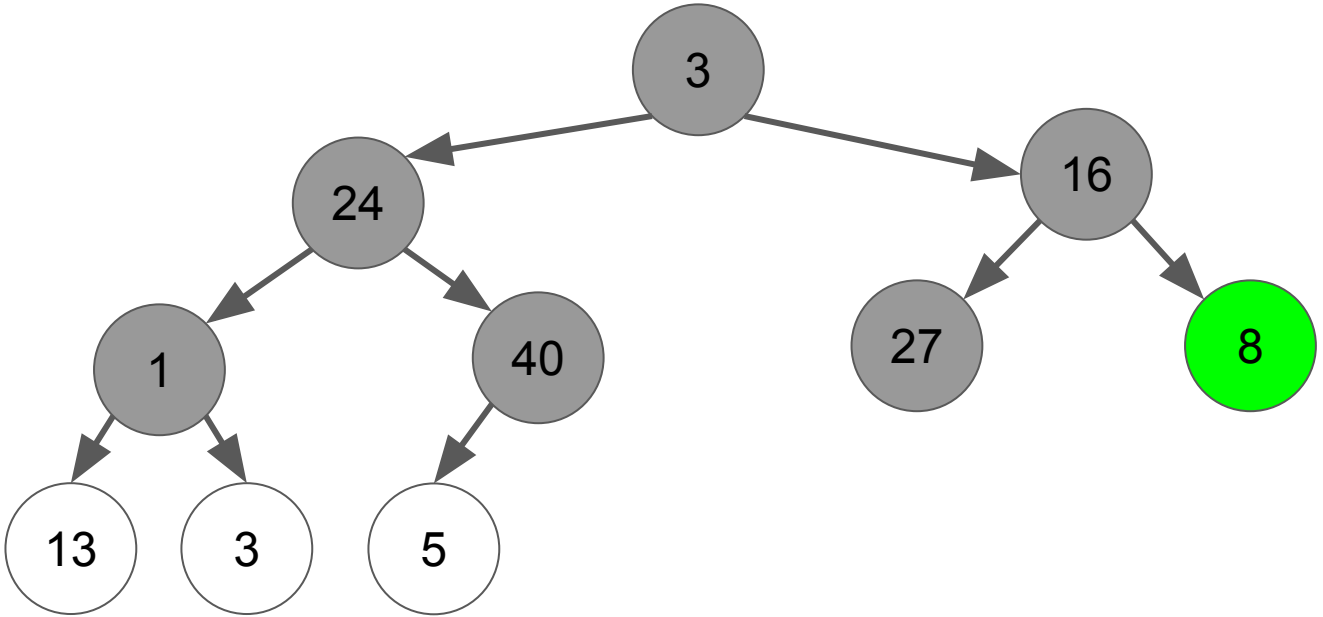
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



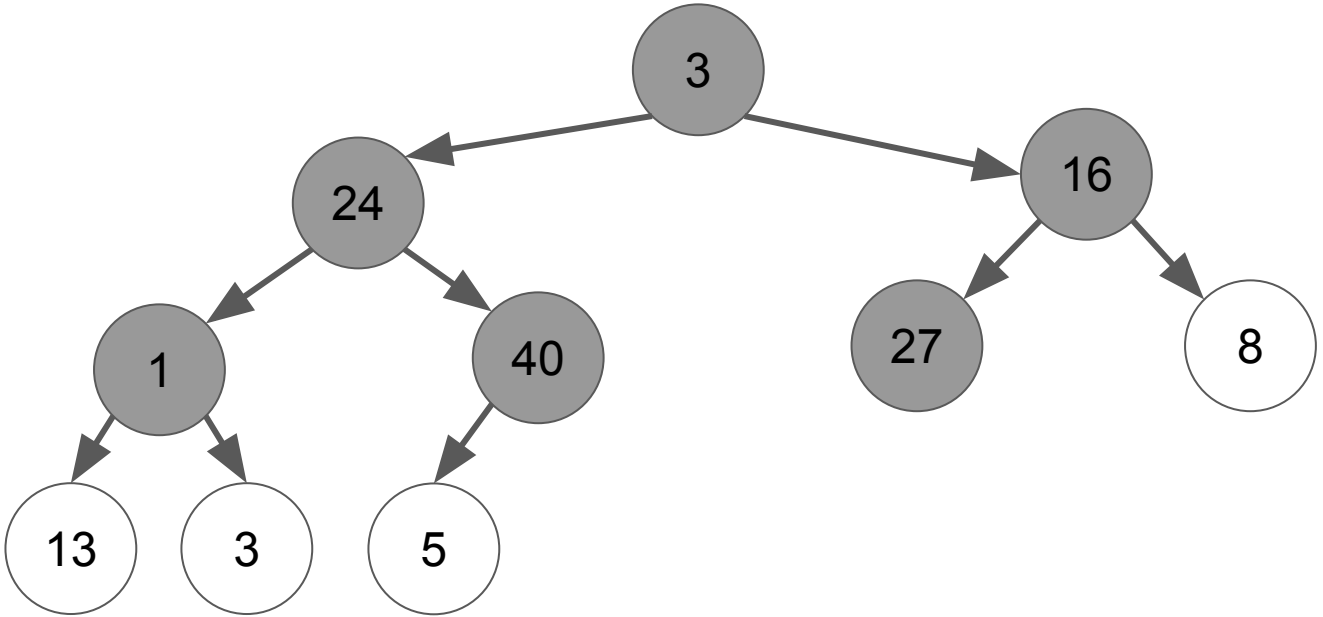
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



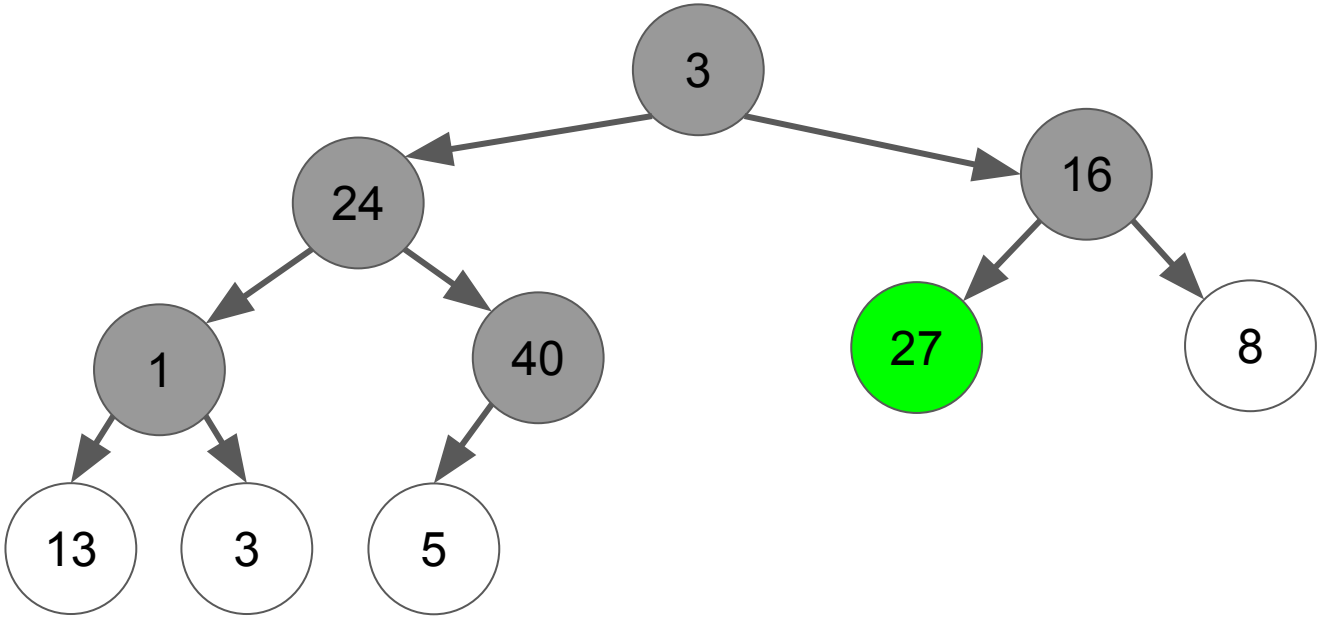
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



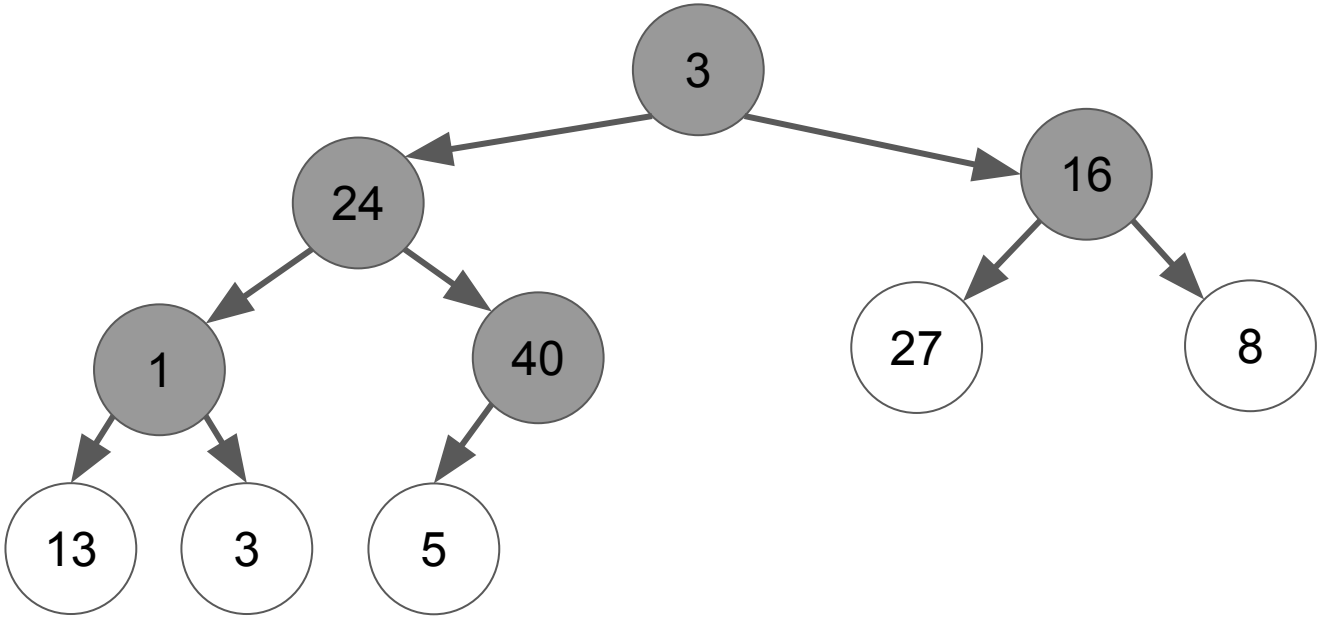
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



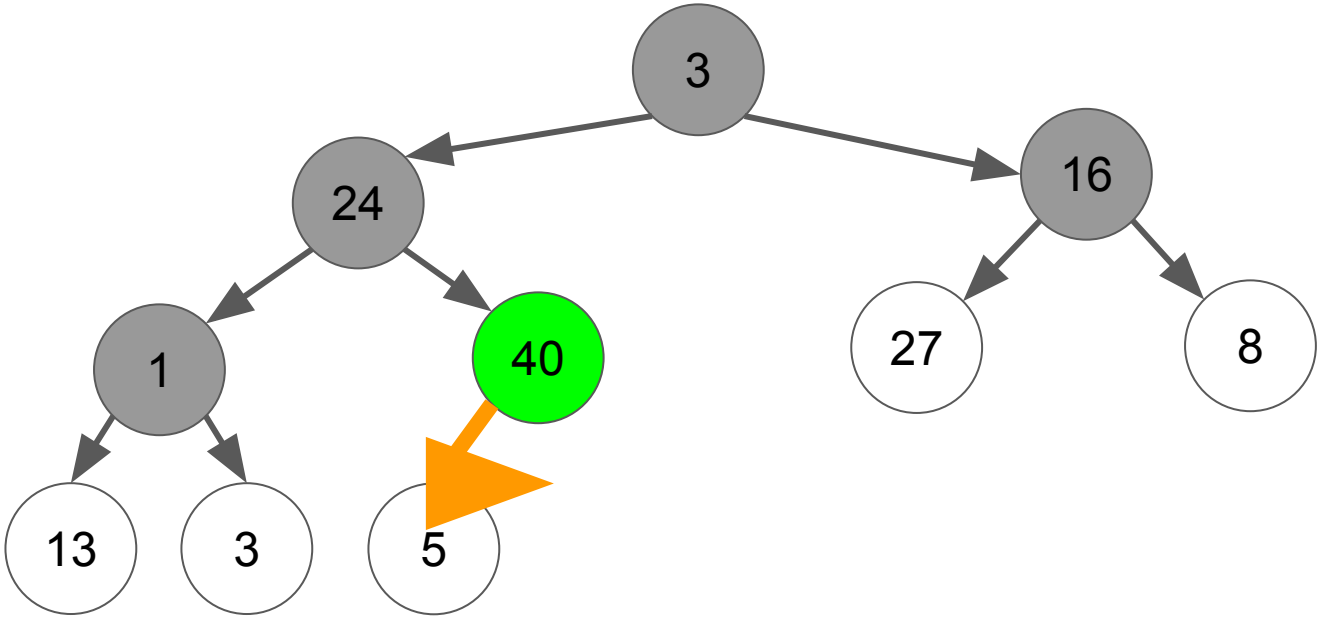
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



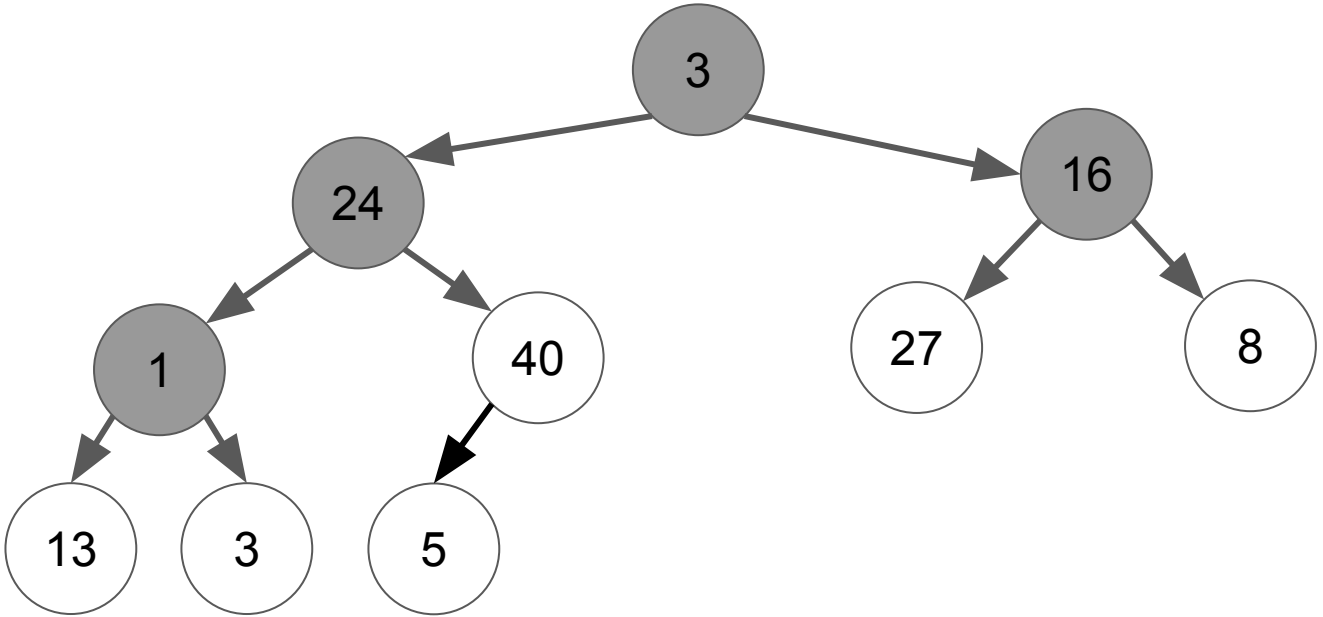
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



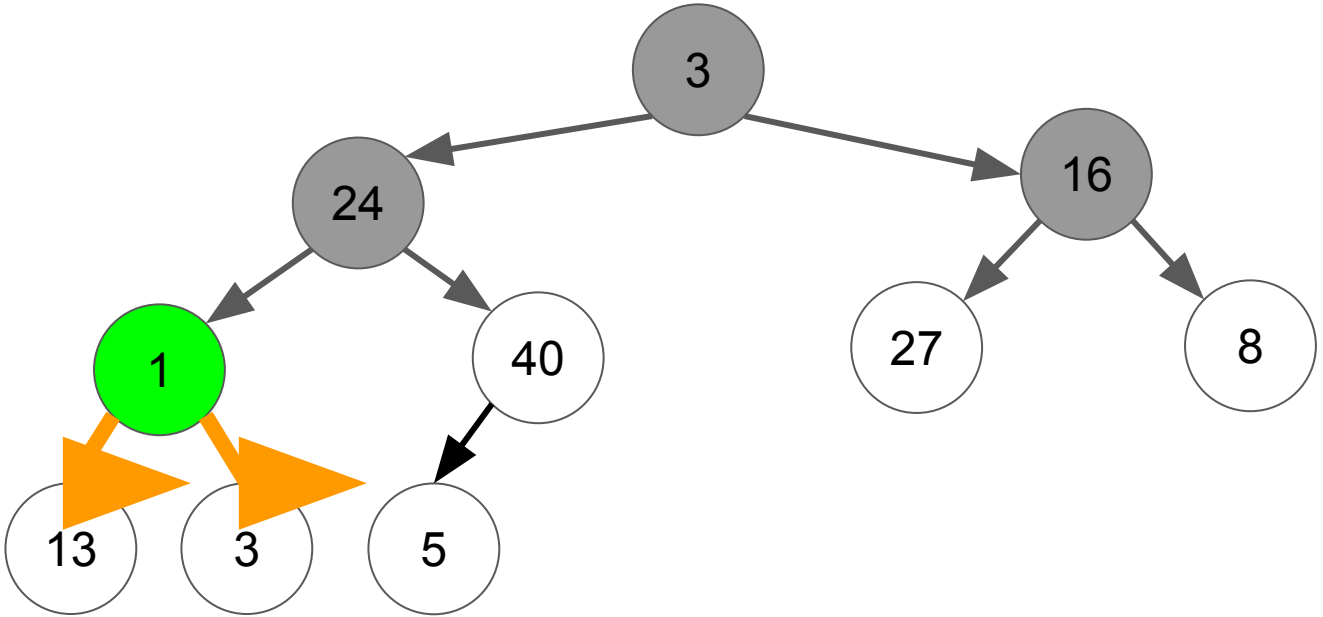
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



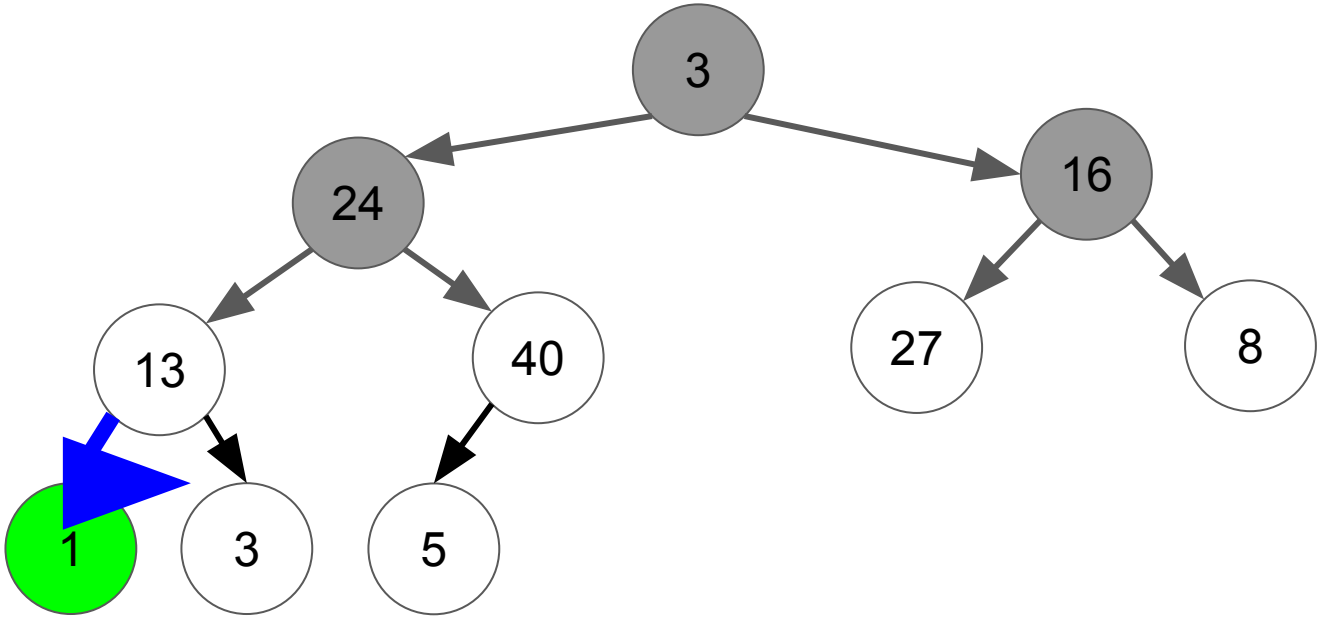
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



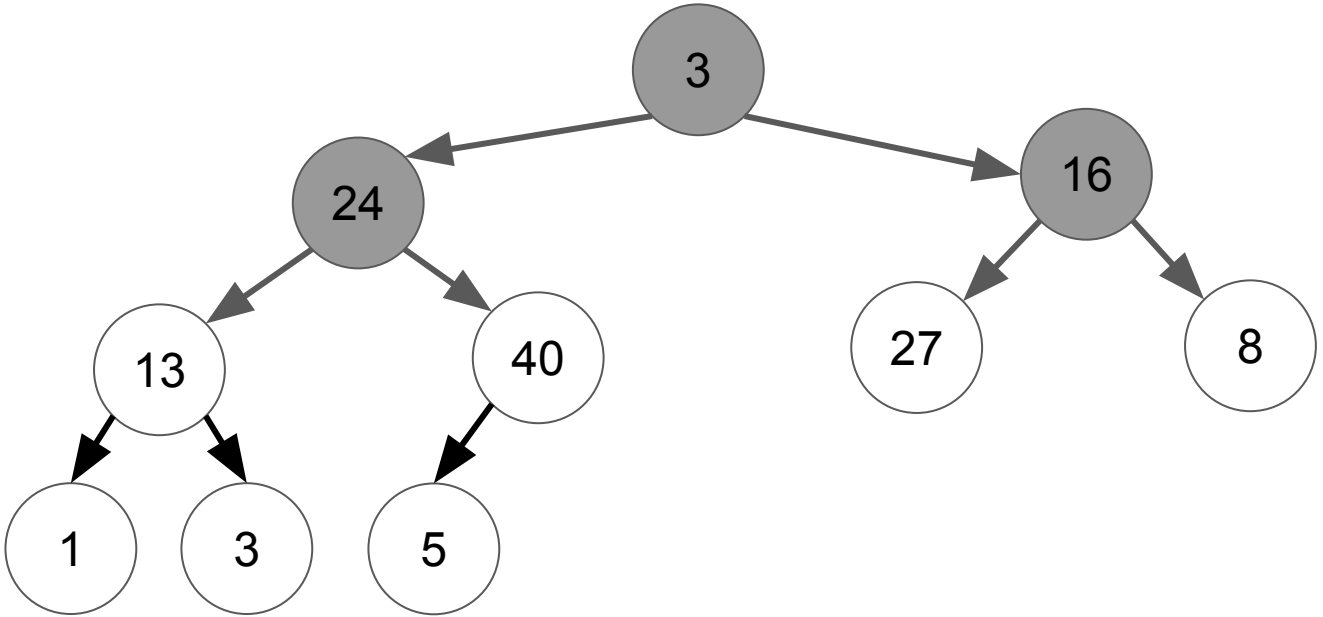
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



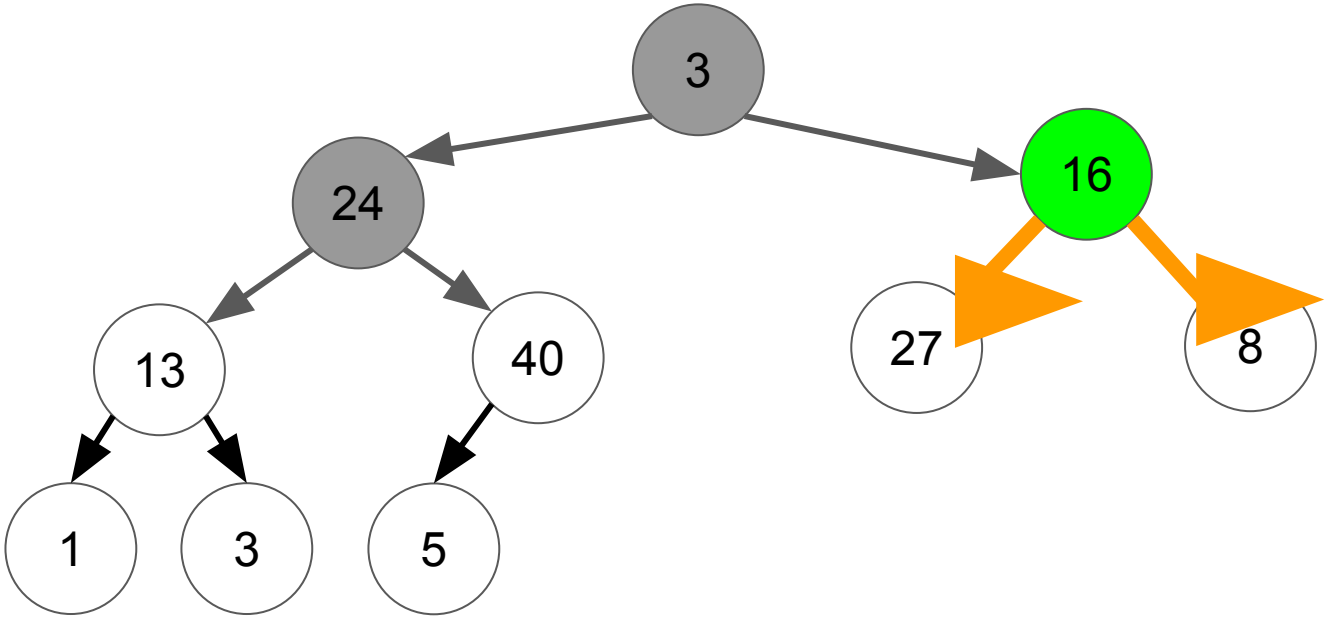
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



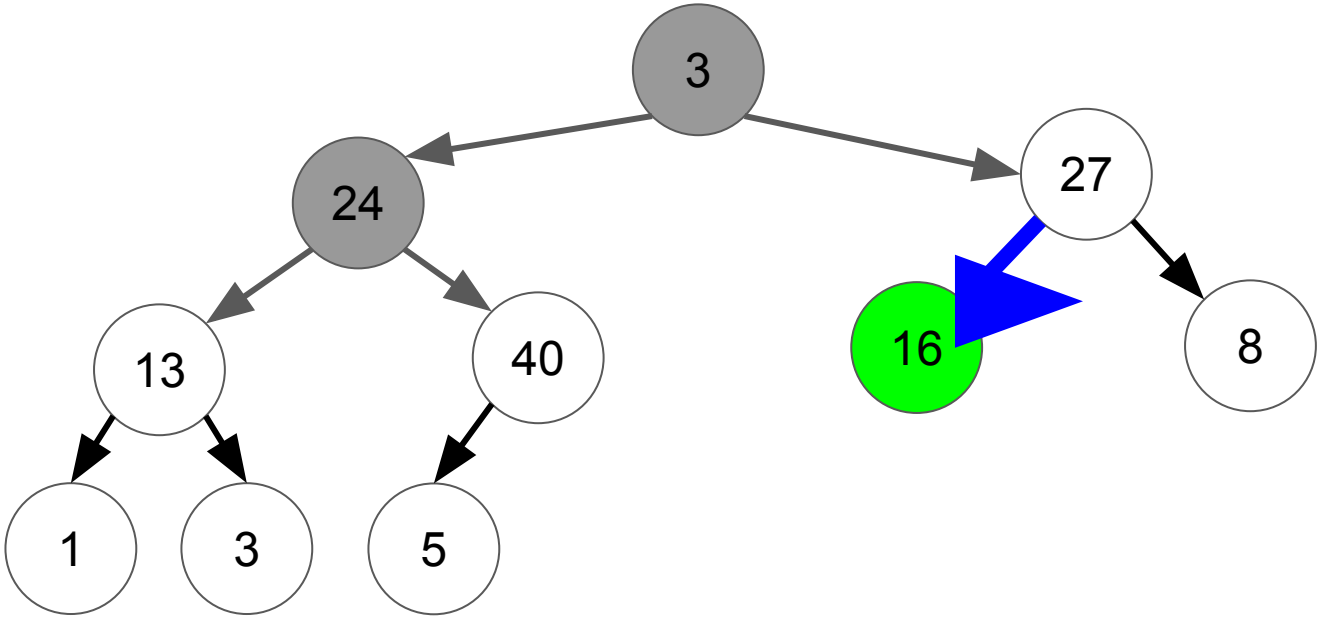
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



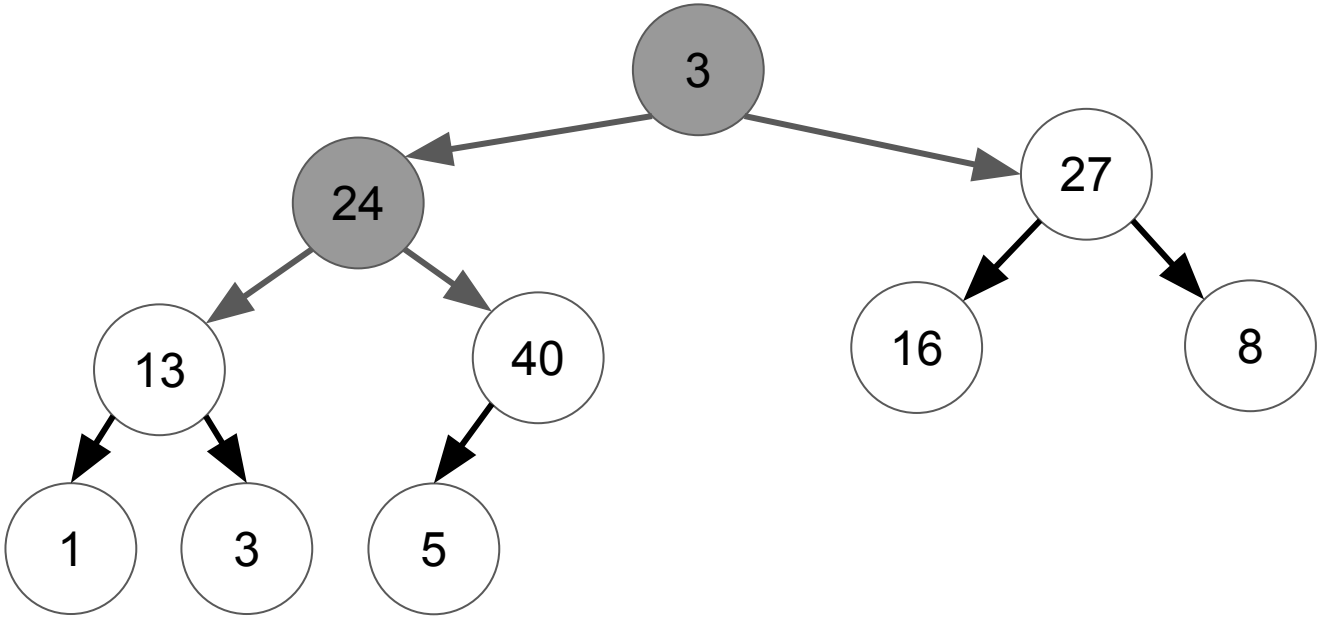
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



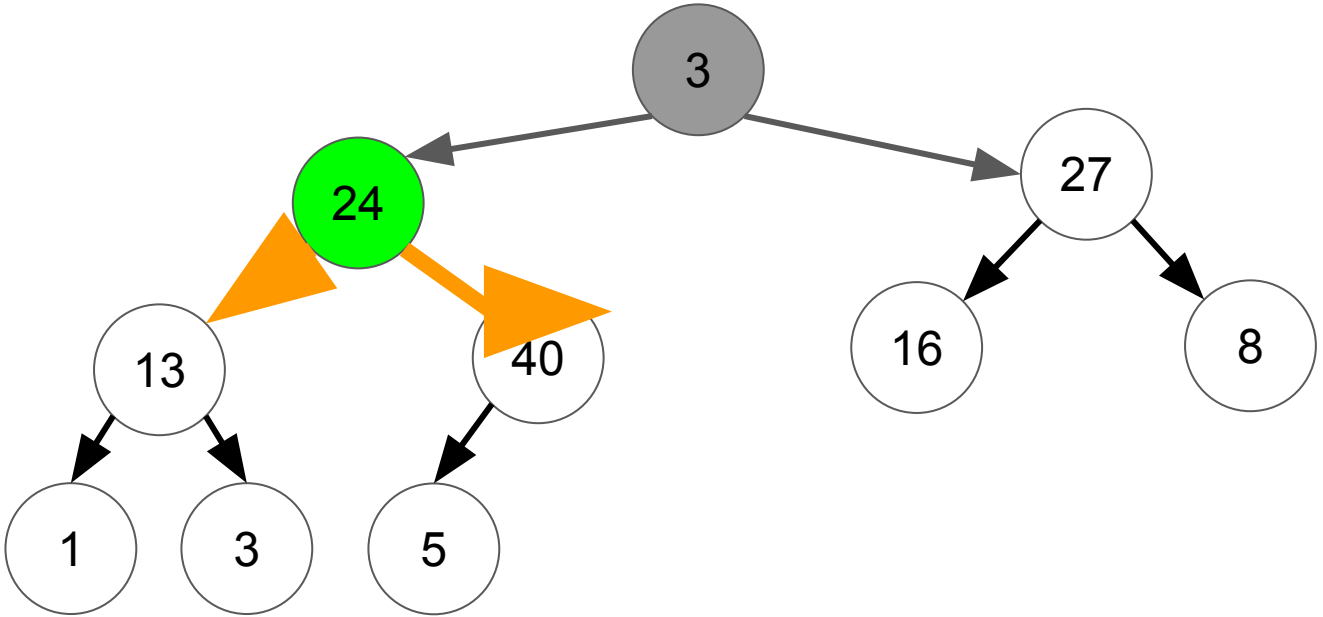
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



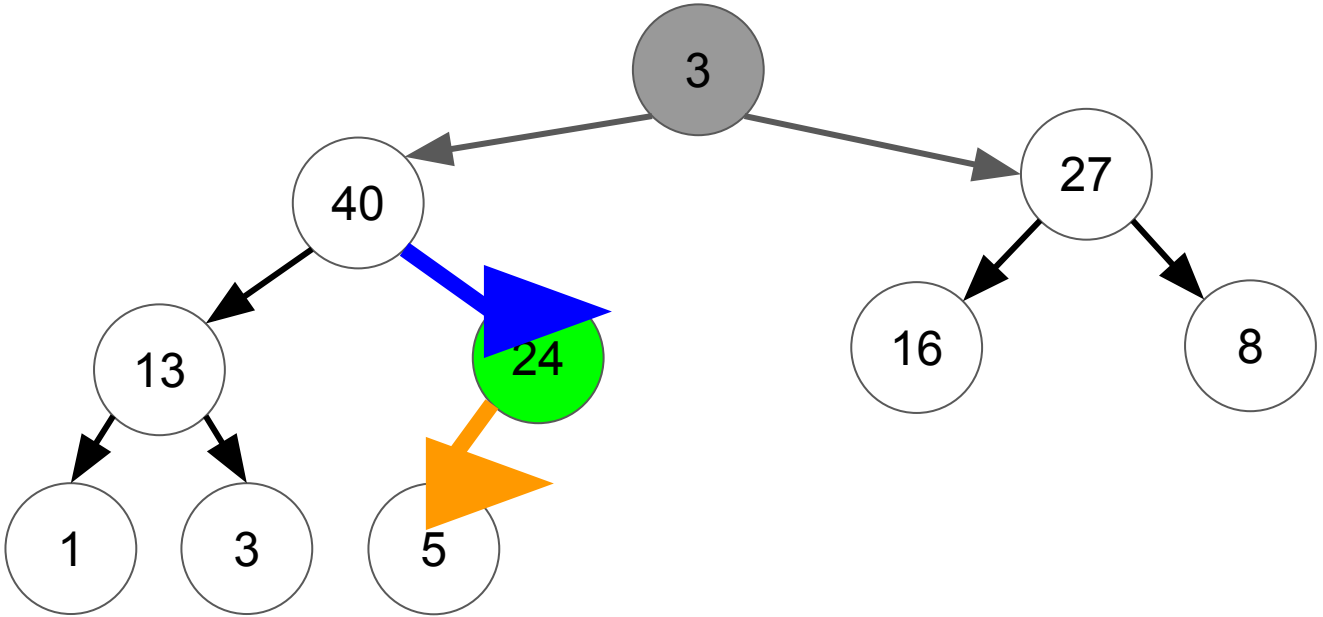
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



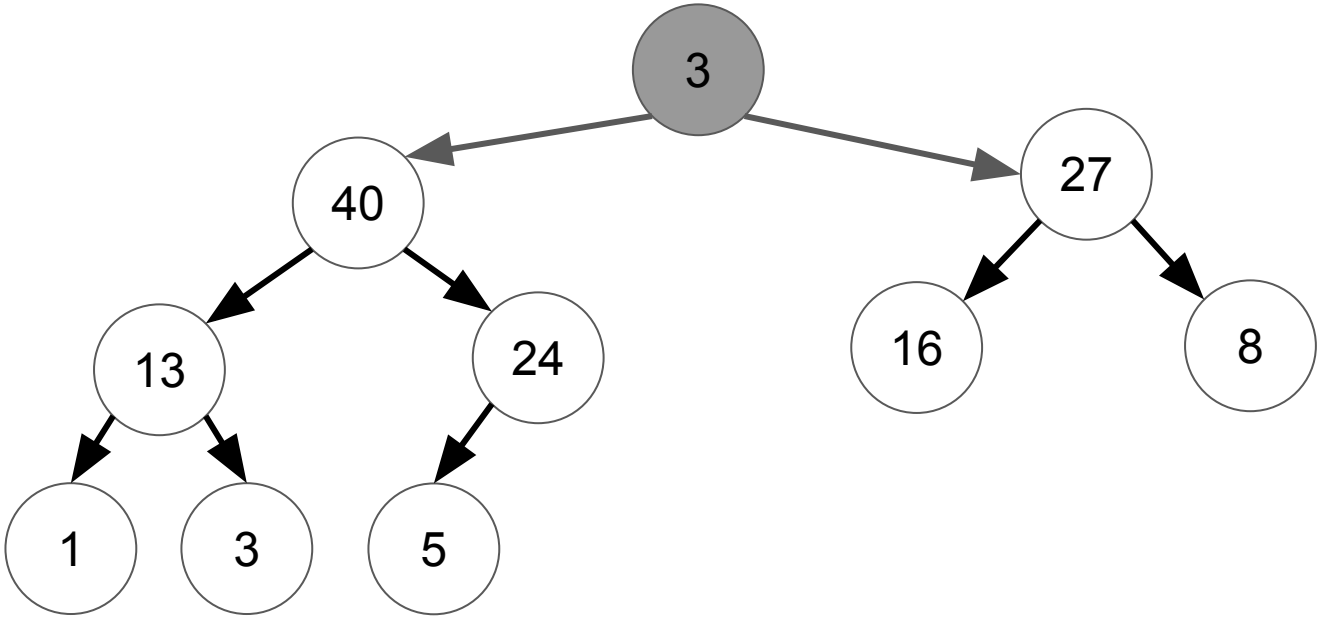
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



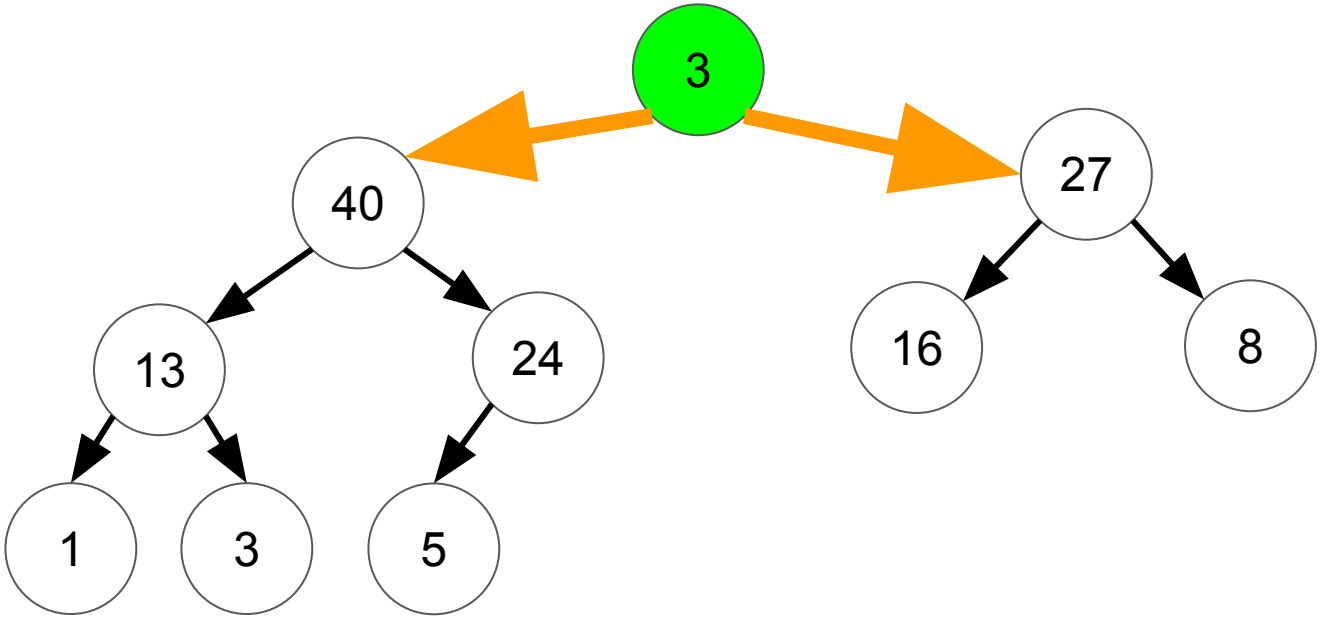
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



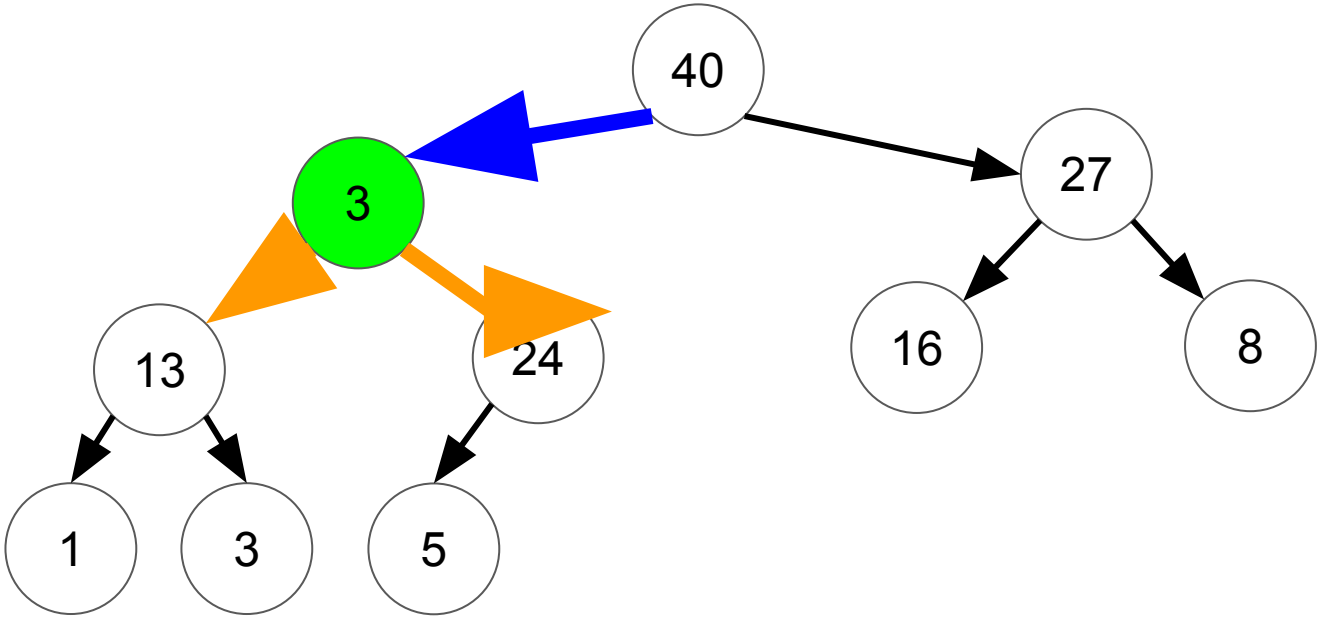
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



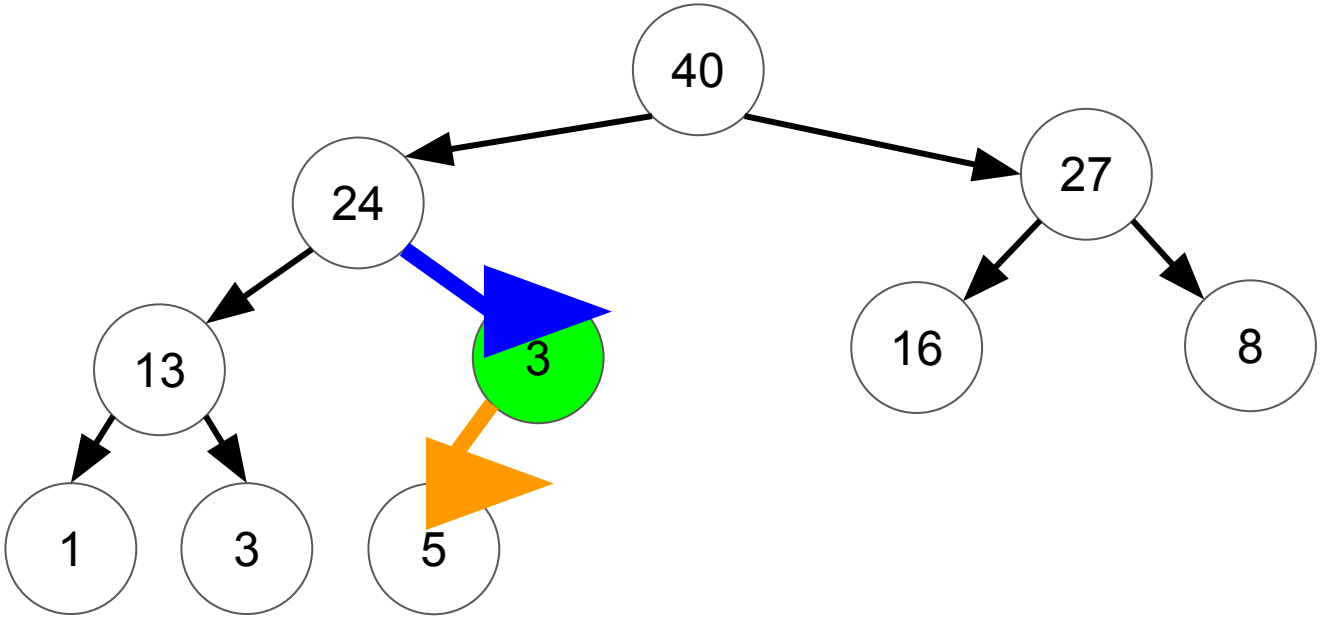
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



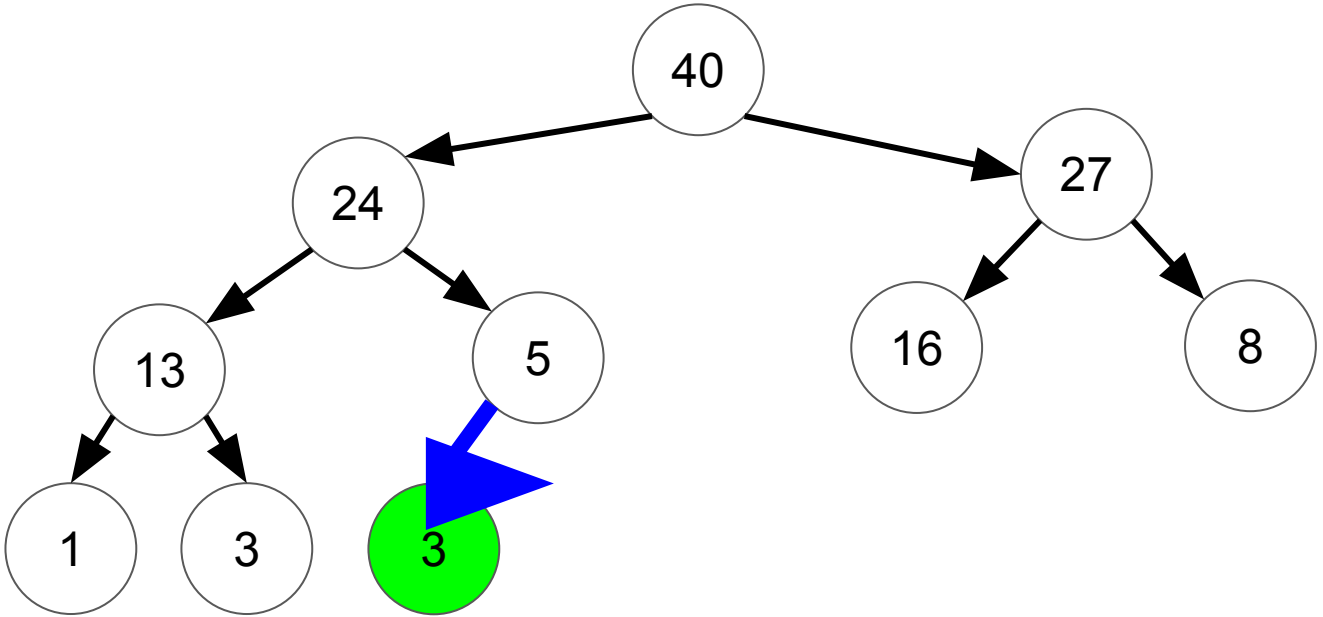
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



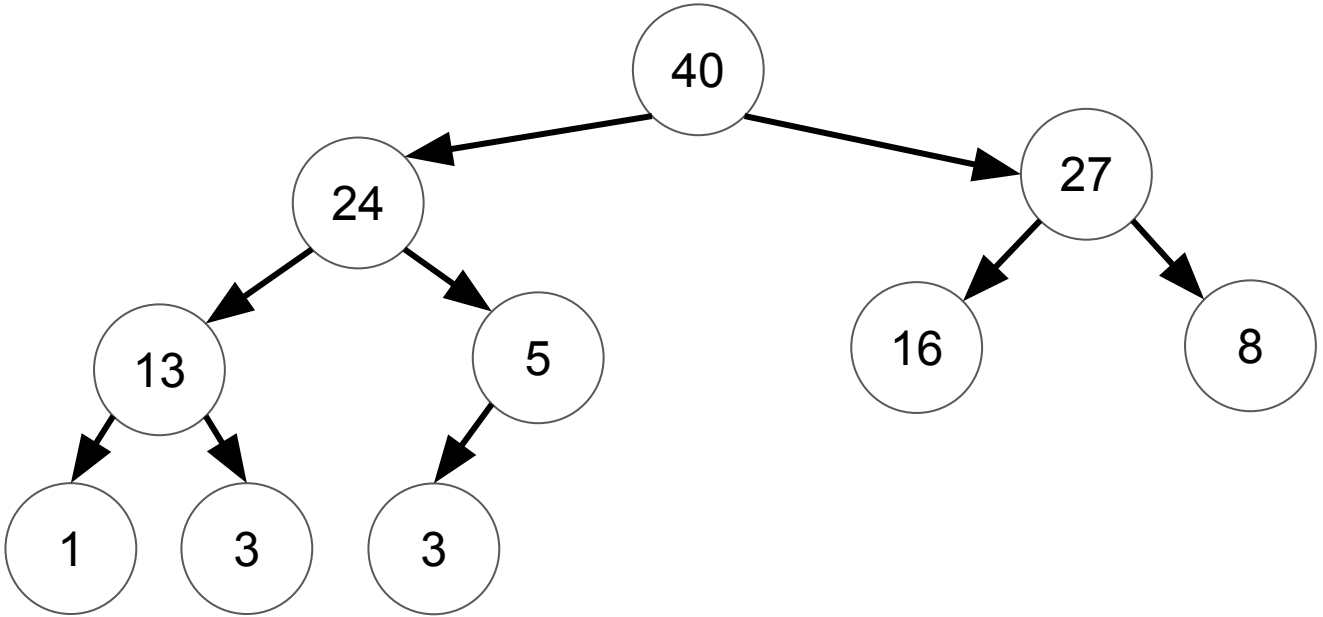
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



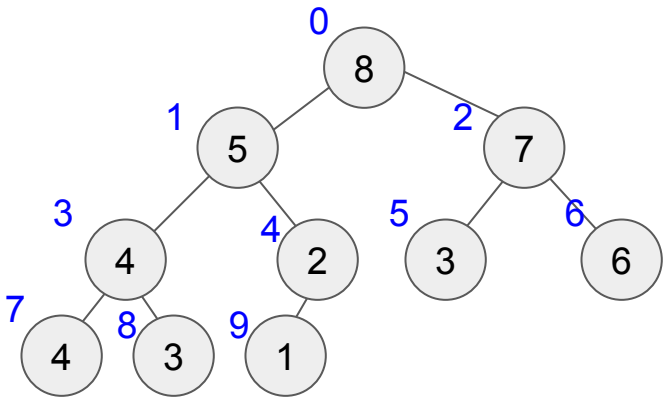
Heapify Example - Fix Down From Bottom

[3, 24, 16, 1, 40, 27, 8, 13, 3, 5]



Priority Queue - Implementations - Binary

A binary heap can be stored conveniently in an array:



(assuming that the root is at **0**) - **the root is always the top() element**

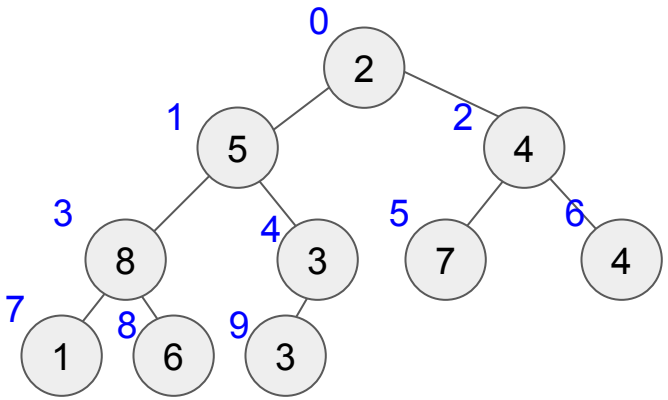
The parent of node stored at location **i** is given as **(i-1)/2**

Conversely, the children of i are located at **2*i+1** and **2*i+2**.

0	1	2	3	4	5	6	7	8	9
8	5	7	4	2	3	6	4	3	1

Priority Queue - Implementations - Binary

Step 1 - Make Heap

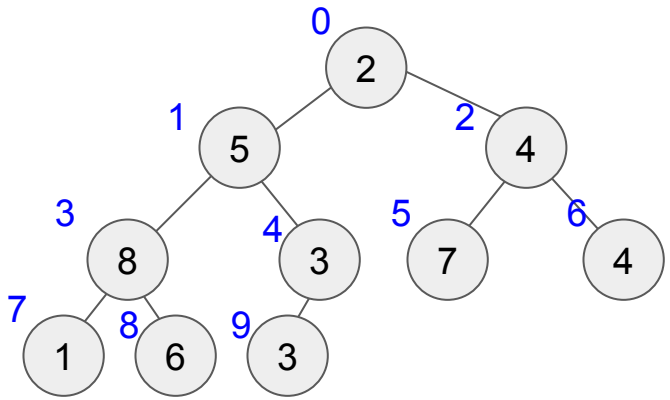


First the array is all messed up - we need to make a heap out of the array.

0	1	2	3	4	5	6	7	8	9
2	5	4	8	3	7	4	1	6	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap



Two approaches which work

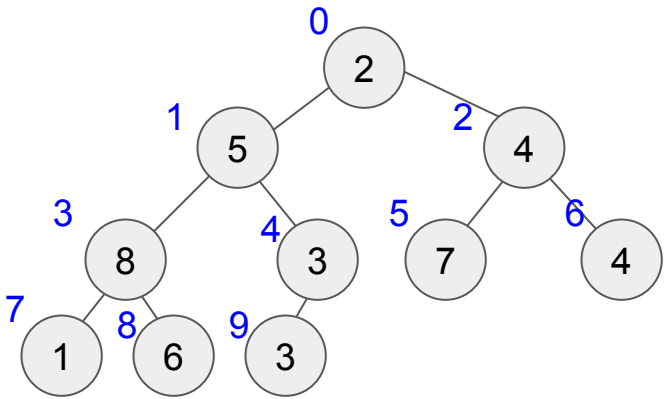
1. Fix up - Top to bottom level wise
2. Fix down - Bottom to top, level wise

Which one to choose?

0	1	2	3	4	5	6	7	8	9
2	5	4	8	3	7	4	1	6	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap



Two approaches which work

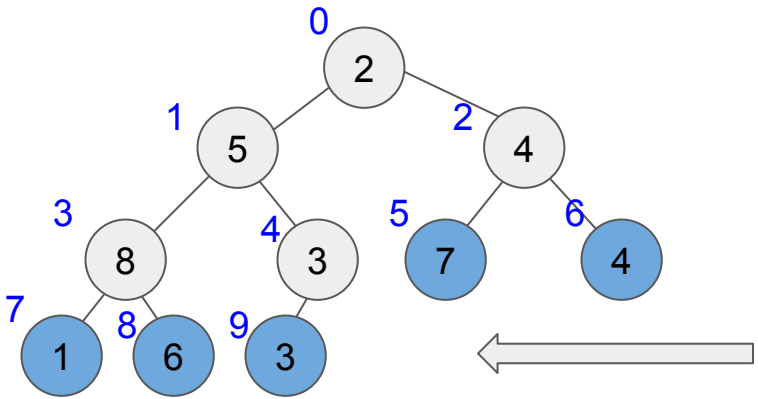
1. Fix up - Top to bottom level wise O(nlogn)
2. Fix down - Bottom to top, level wise O(n)

Which one to choose?

0	1	2	3	4	5	6	7	8	9
2	5	4	8	3	7	4	1	6	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap



Two approaches which work

1. Fix up - Top to bottom level wise O(nlogn)
2. Fix down - Bottom to top, level wise O(n)

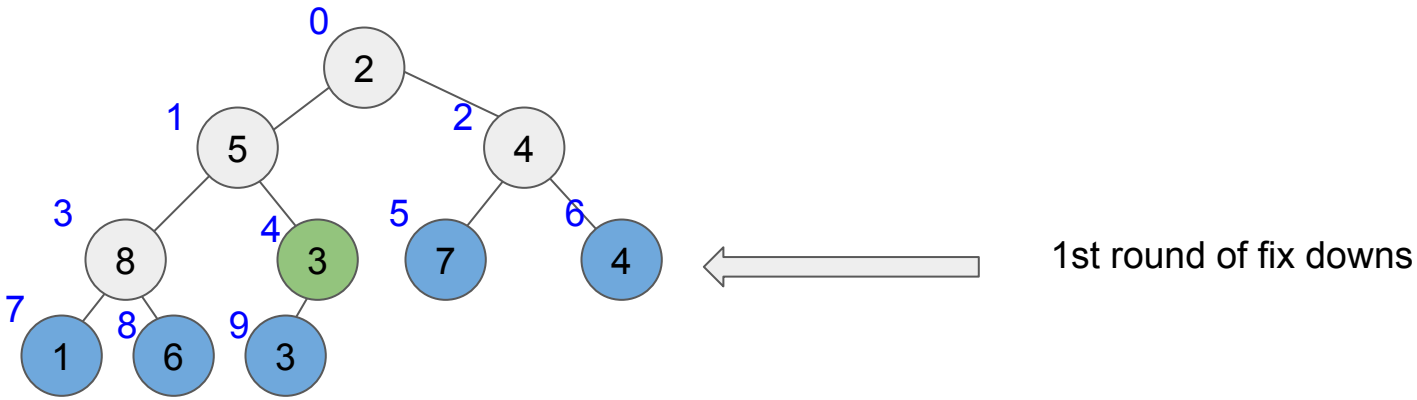
Which one to choose?

Nothing to fix down here, hence we start one level before the bottom most

0	1	2	3	4	5	6	7	8	9
2	5	4	8	3	7	4	1	6	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

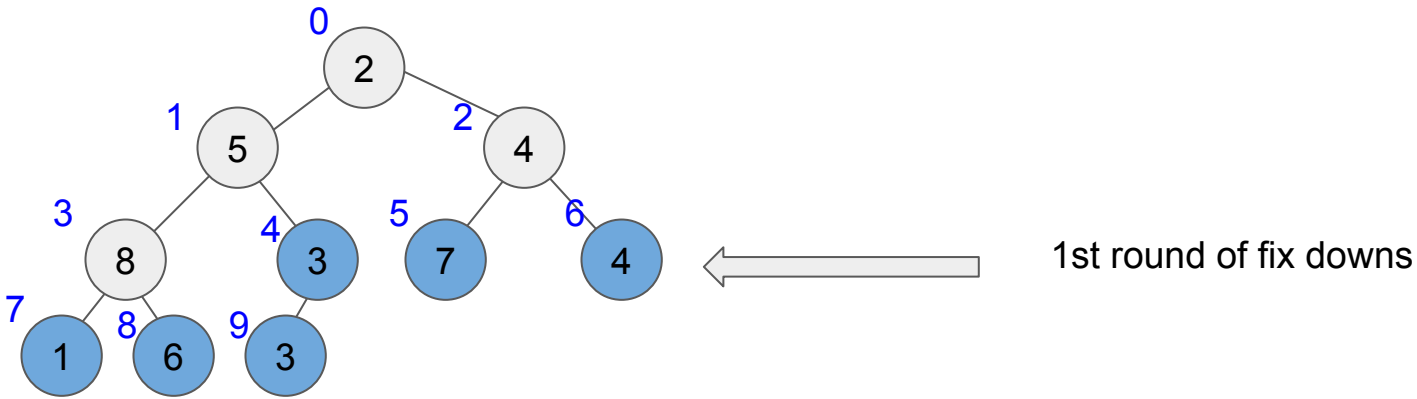


1st round of fix downs

0	1	2	3	4	5	6	7	8	9
2	5	4	8	3	7	4	1	6	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

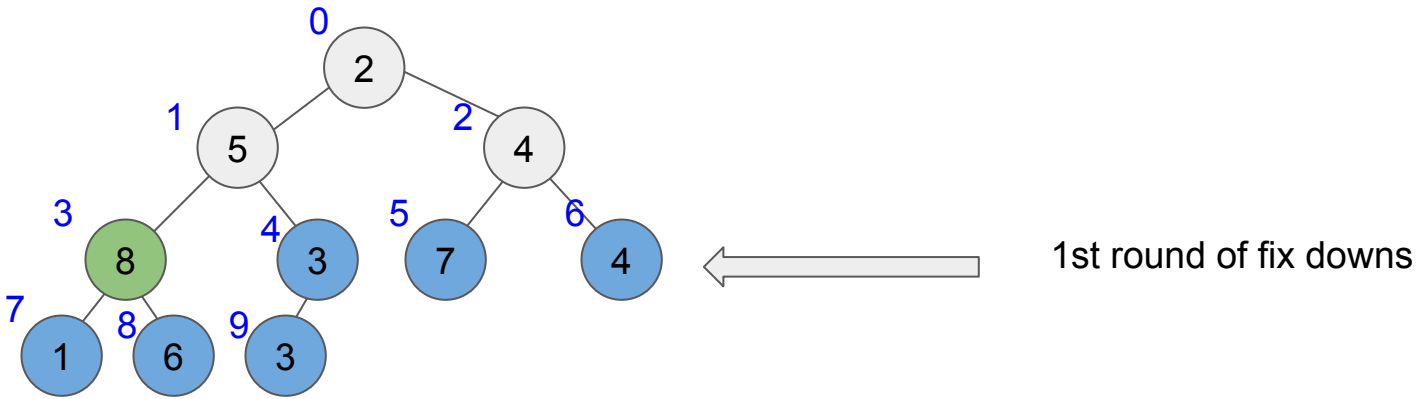


1st round of fix downs

0	1	2	3	4	5	6	7	8	9
2	5	4	8	3	7	4	1	6	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

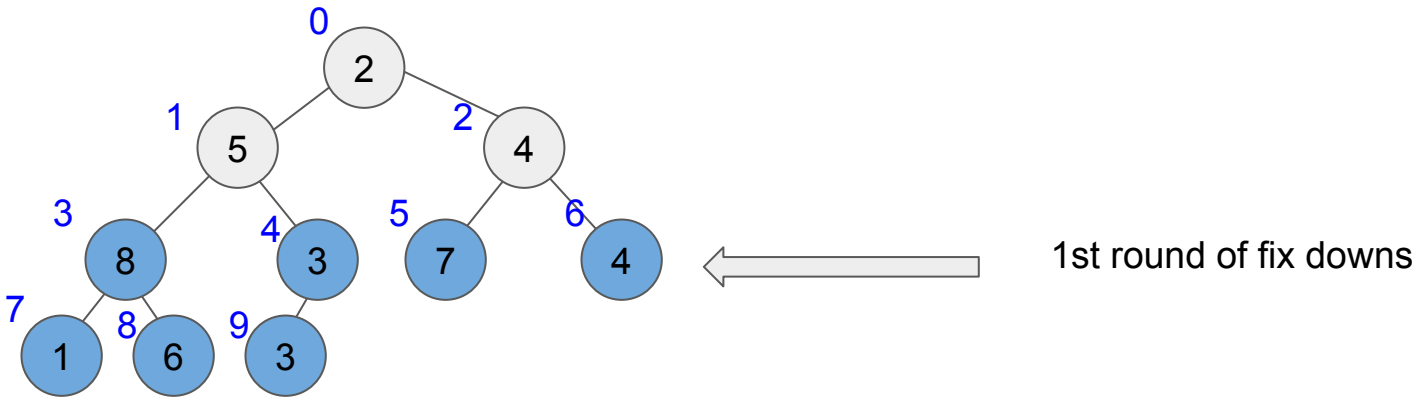


1st round of fix downs

0	1	2	3	4	5	6	7	8	9
2	5	4	8	3	7	4	1	6	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

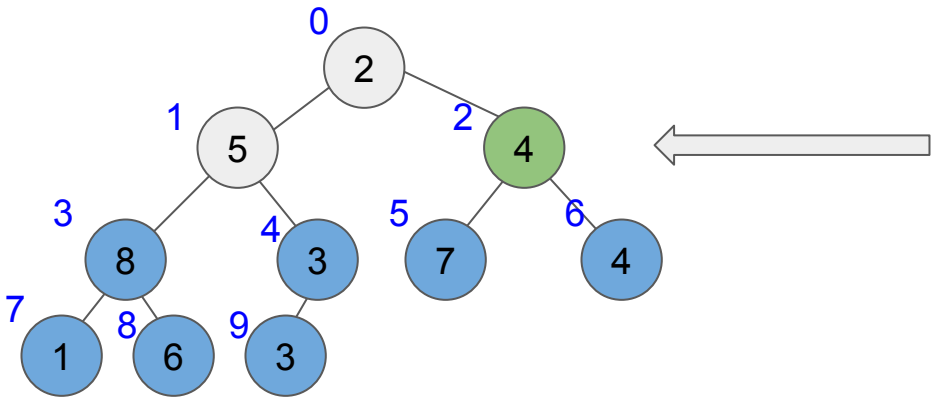


1st round of fix downs

0	1	2	3	4	5	6	7	8	9
2	5	4	8	3	7	4	1	6	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

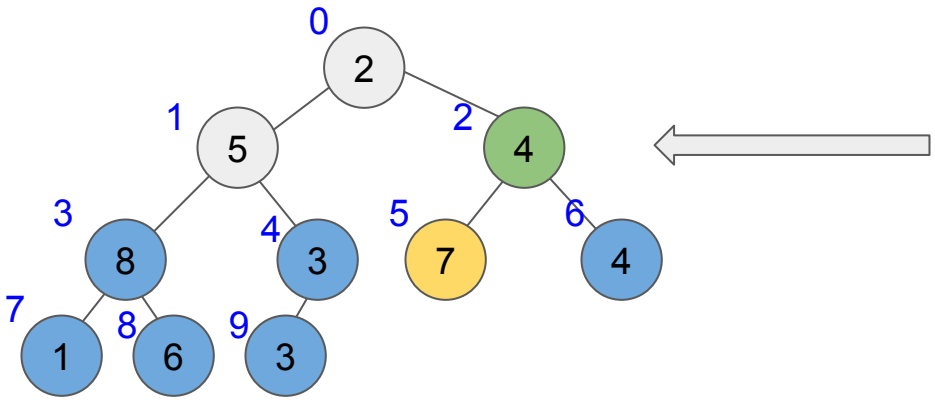


2nd round of fix downs

0	1	2	3	4	5	6	7	8	9
2	5	4	8	3	7	4	1	6	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

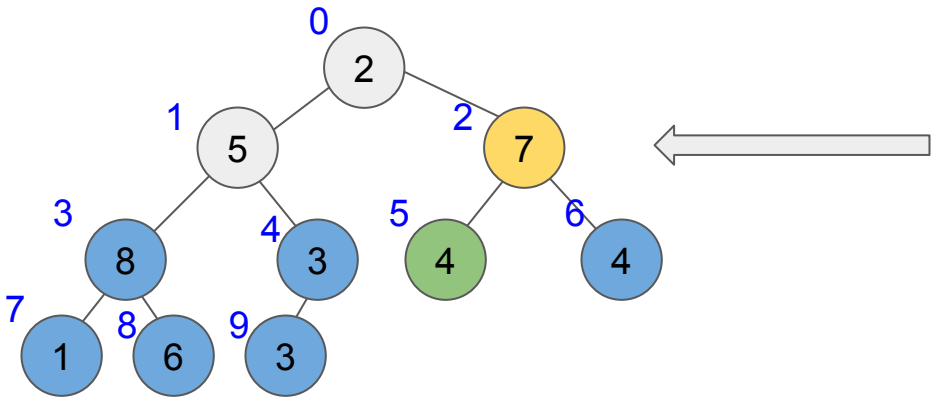


2nd round of fix downs

0	1	2	3	4	5	6	7	8	9
2	5	4	8	3	7	4	1	6	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

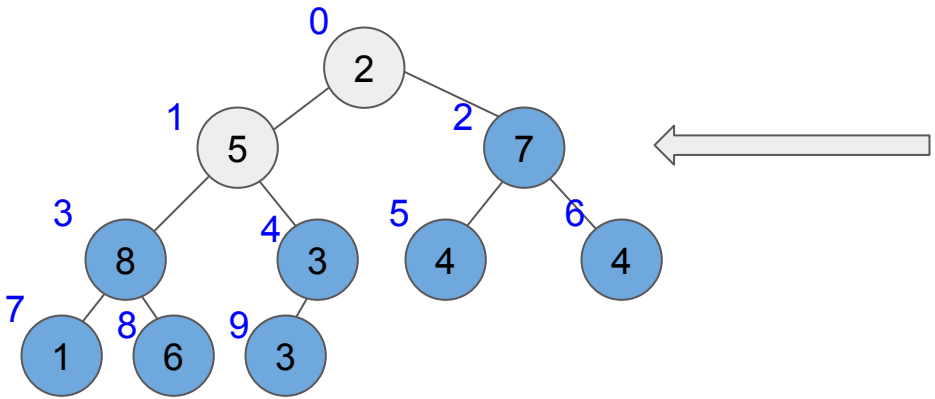


2nd round of fix downs

0	1	2	3	4	5	6	7	8	9
2	5	7	8	3	4	4	1	6	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

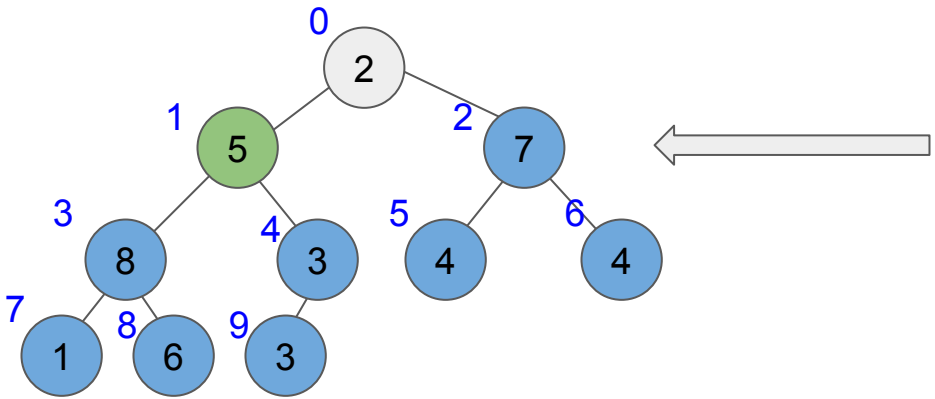


2nd round of fix downs

0	1	2	3	4	5	6	7	8	9
2	5	7	8	3	4	4	1	6	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

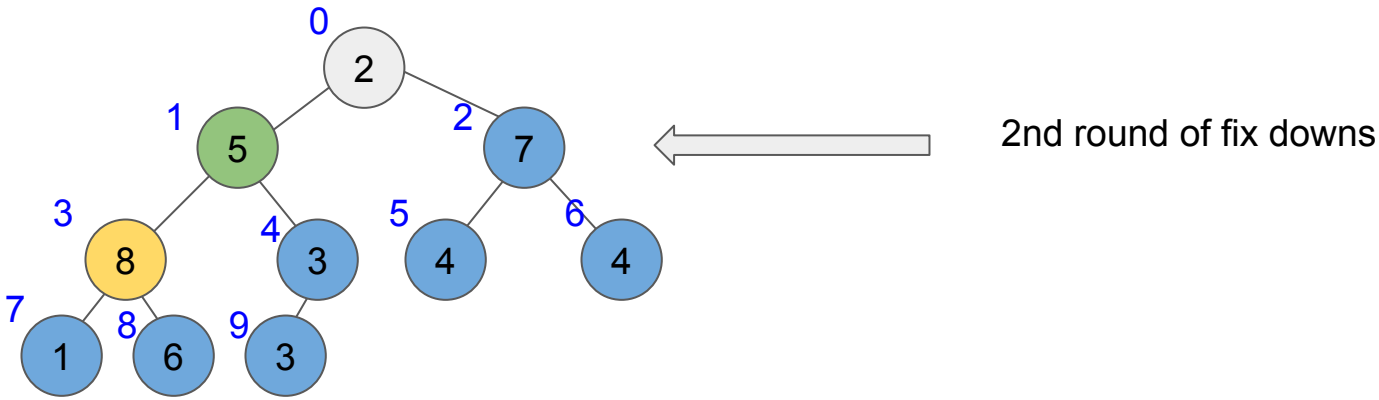


2nd round of fix downs

0	1	2	3	4	5	6	7	8	9
2	5	7	8	3	4	4	1	6	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

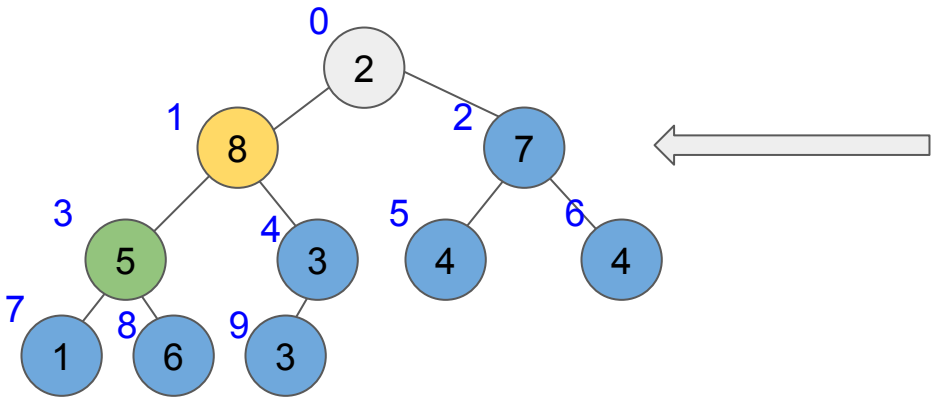


2nd round of fix downs

0	1	2	3	4	5	6	7	8	9
2	5	7	8	3	4	4	1	6	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

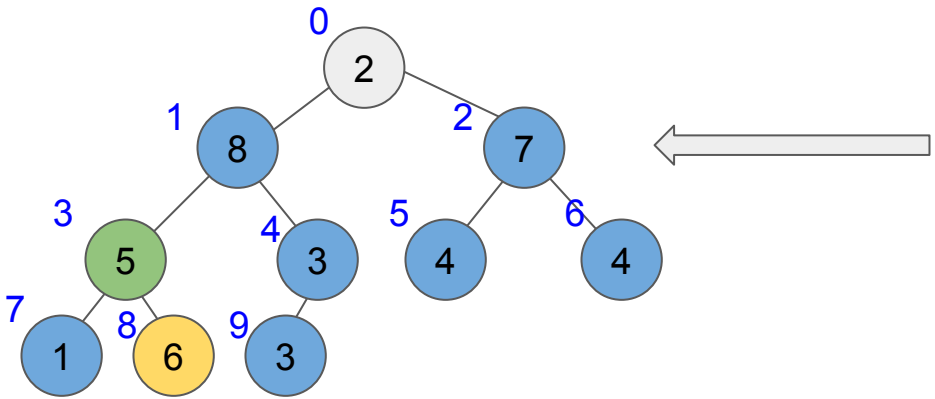


2nd round of fix downs

0	1	2	3	4	5	6	7	8	9
2	8	7	5	3	4	4	1	6	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

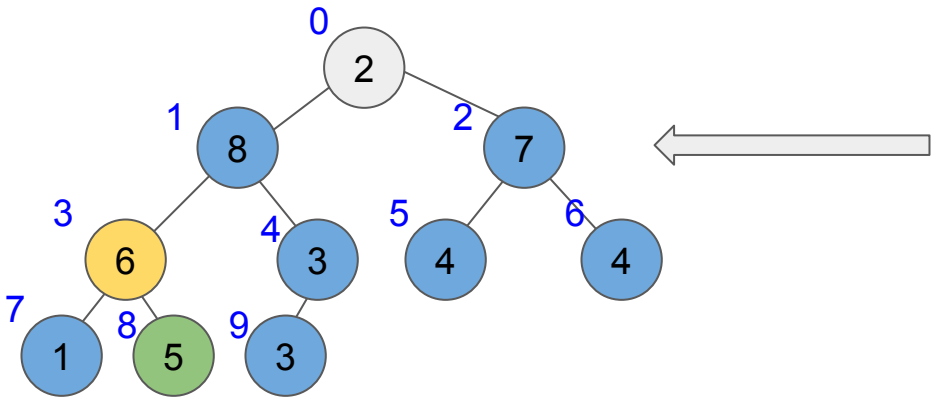


2nd round of fix downs

0	1	2	3	4	5	6	7	8	9
2	8	7	5	3	4	4	1	6	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

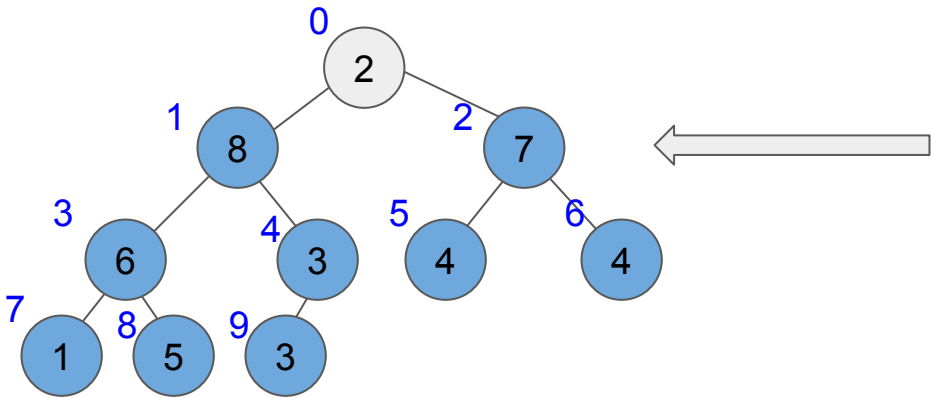


2nd round of fix downs

0	1	2	3	4	5	6	7	8	9
2	8	7	6	3	4	4	1	5	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

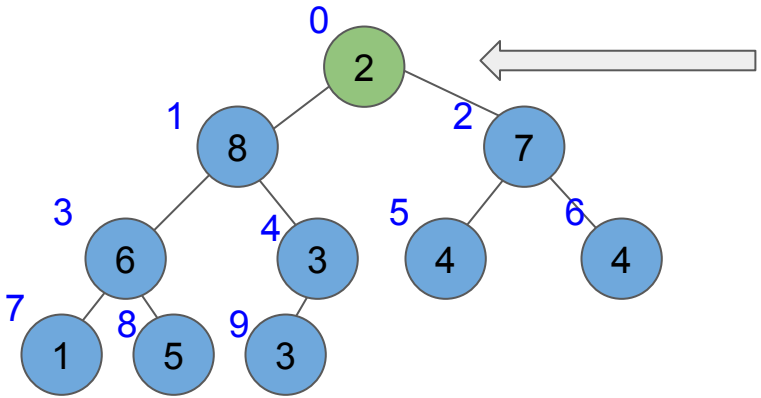


2nd round of fix downs

0	1	2	3	4	5	6	7	8	9
2	8	7	6	3	4	4	1	5	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

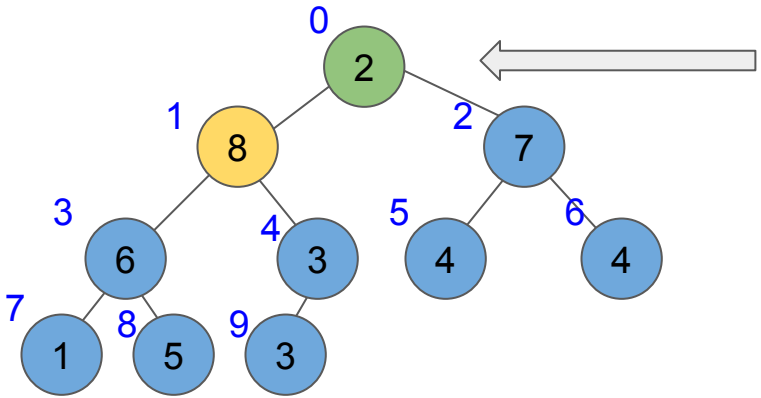


3rd round of fix downs

0	1	2	3	4	5	6	7	8	9
2	8	7	6	3	4	4	1	5	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

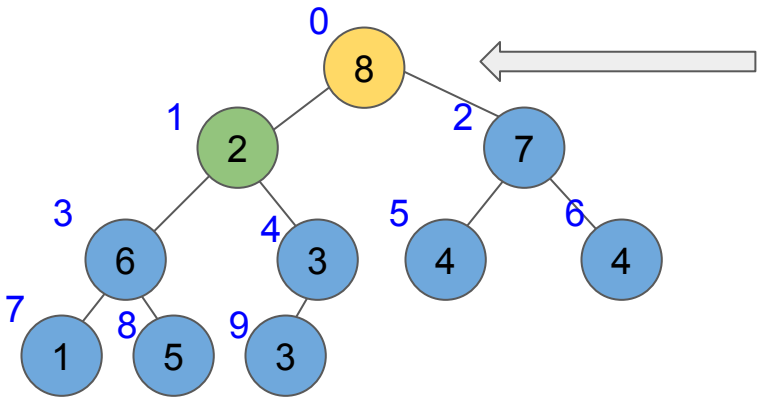


3rd round of fix downs

0	1	2	3	4	5	6	7	8	9
2	8	7	6	3	4	4	1	5	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

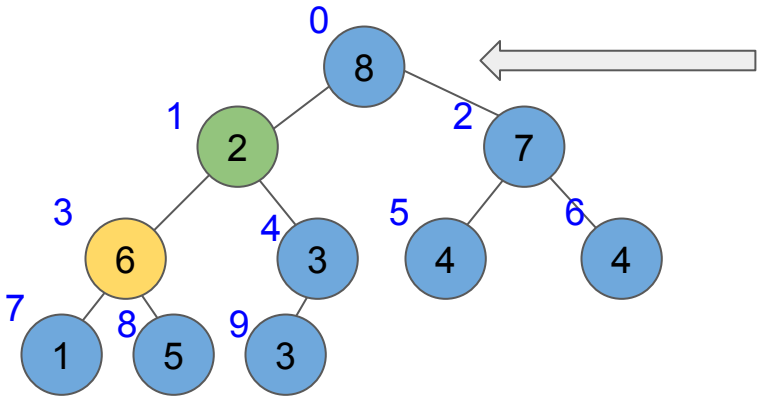


3rd round of fix downs

0	1	2	3	4	5	6	7	8	9
8	2	7	6	3	4	4	1	5	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

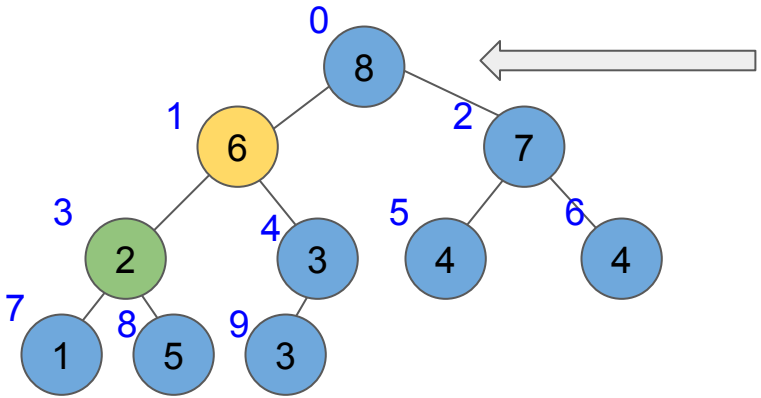


3rd round of fix downs

0	1	2	3	4	5	6	7	8	9
8	2	7	6	3	4	4	1	5	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

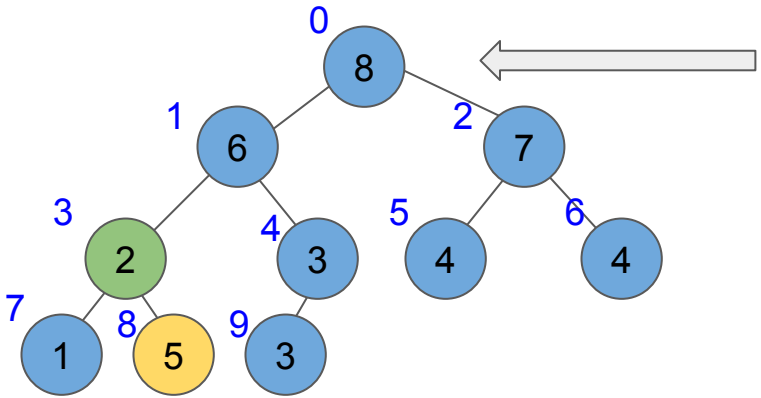


3rd round of fix downs

0	1	2	3	4	5	6	7	8	9
8	6	7	2	3	4	4	1	5	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

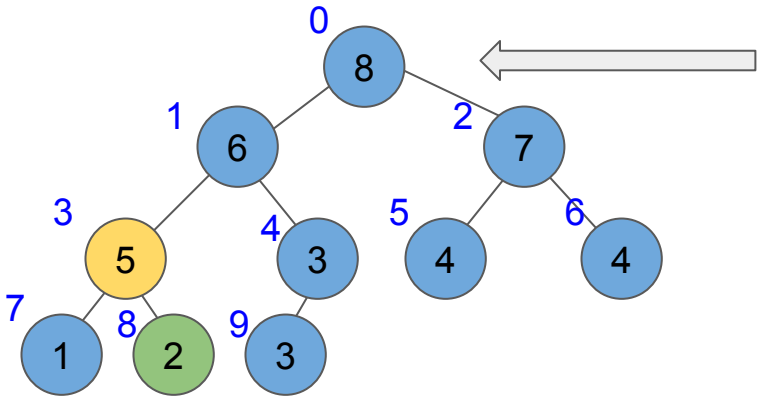


3rd round of fix downs

0	1	2	3	4	5	6	7	8	9
8	6	7	2	3	4	4	1	5	3

Priority Queue - Implementations - Binary

Step 1 - Make Heap

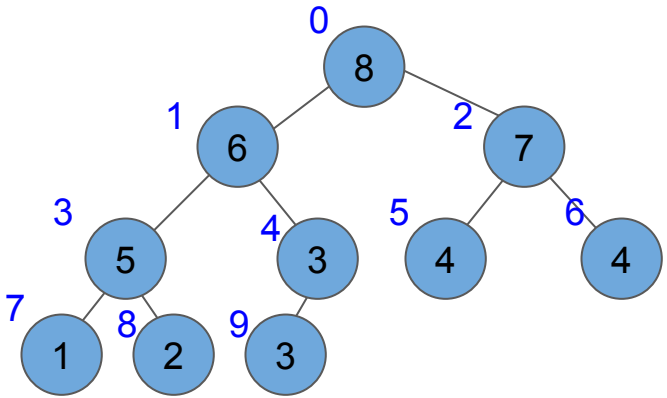


3rd round of fix downs

0	1	2	3	4	5	6	7	8	9
8	6	7	5	3	4	4	1	2	3

Priority Queue - Implementations - Binary

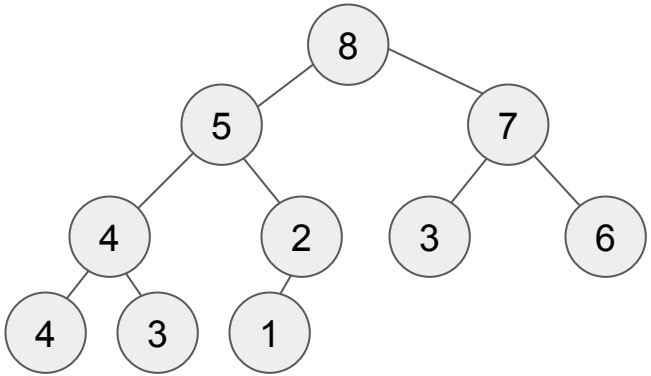
Resultant Heap & Array



0	1	2	3	4	5	6	7	8	9
8	6	7	5	3	4	4	1	2	3

Priority Queue - Implementations - Binary - Push

To insert an element into the PQ, we start by pushing it to the end of the vector.



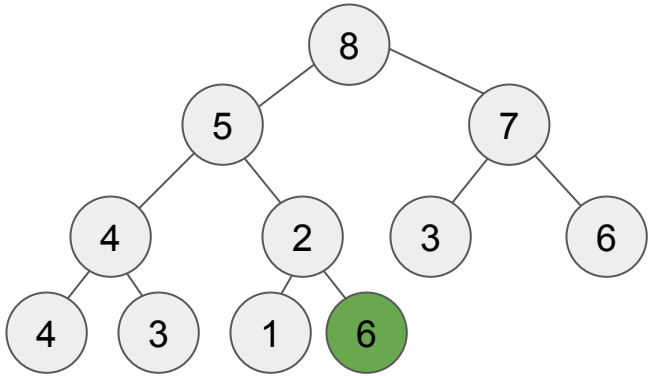
Now, we *fix up* from that location

- Is it larger than its parent?
  - If so, swap and repeat
  - Otherwise, you're done.

8	5	7	4	2	3	6	4	3	1

Priority Queue - Implementations - Binary - Push

To insert an element into the PQ, we start by pushing it to the end of the vector.



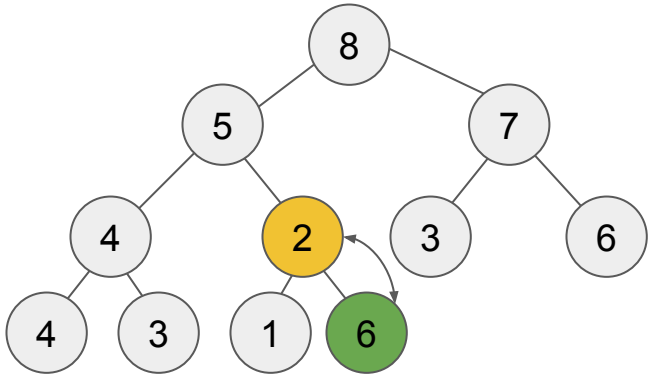
Now, we *fix up* from that location

- Is it larger than its parent?
  - If so, swap and repeat
  - Otherwise, you're done.

8	5	7	4	2	3	6	4	3	1	6

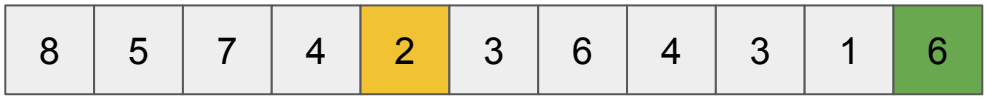
Priority Queue - Implementations - Binary - Push

To insert an element into the PQ, we start by pushing it to the end of the vector.



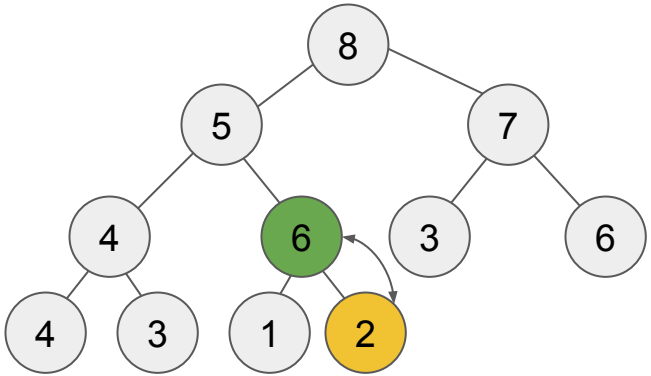
Now, we *fix up* from that location

- **Is it larger than its parent?**
  - If so, swap and repeat
  - Otherwise, you're done.



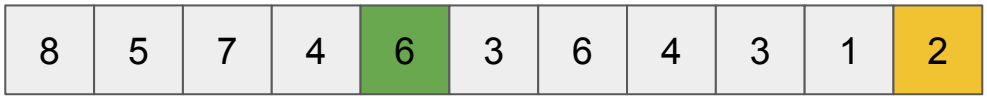
Priority Queue - Implementations - Binary - Push

To insert an element into the PQ, we start by pushing it to the end of the vector.



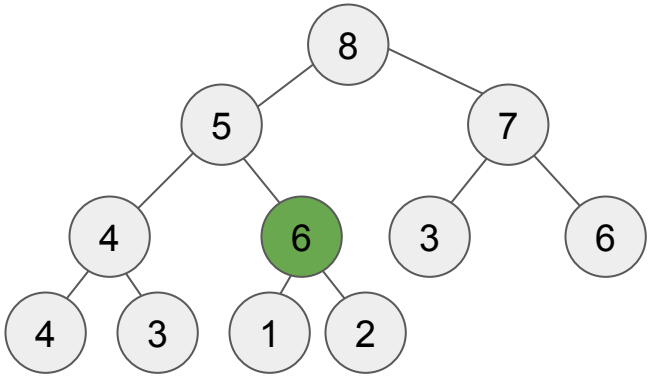
Now, we *fix up* from that location

- Is it larger than its parent?
  - **If so, swap and repeat**
  - Otherwise, you're done.



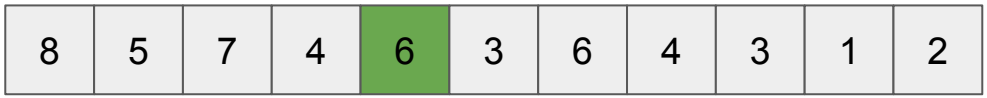
Priority Queue - Implementations - Binary - Push

To insert an element into the PQ, we start by pushing it to the end of the vector.



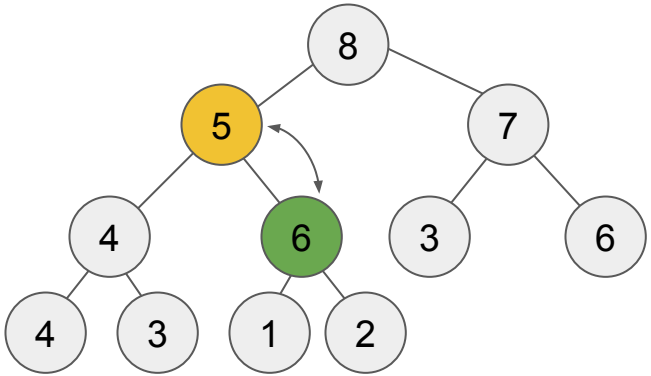
Now, we *fix up* from that location

- Is it larger than its parent?
  - If so, swap and **repeat**
  - Otherwise, you're done.



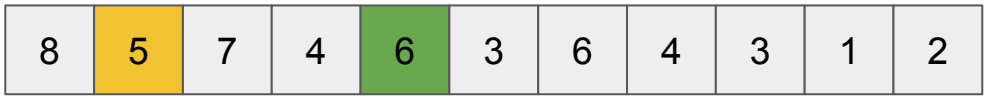
Priority Queue - Implementations - Binary - Push

To insert an element into the PQ, we start by pushing it to the end of the vector.



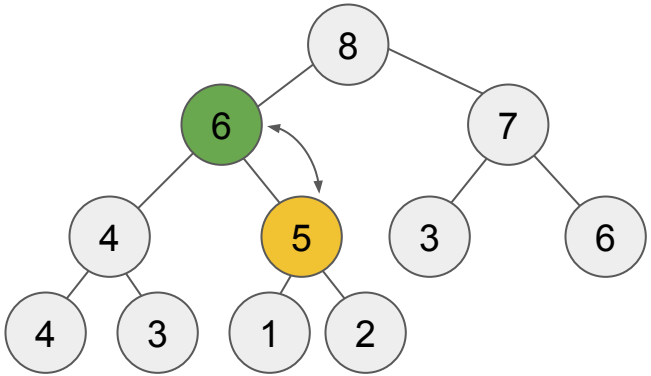
Now, we *fix up* from that location

- **Is it larger than its parent?**
  - If so, swap and repeat
  - Otherwise, you're done.



Priority Queue - Implementations - Binary - Push

To insert an element into the PQ, we start by pushing it to the end of the vector.



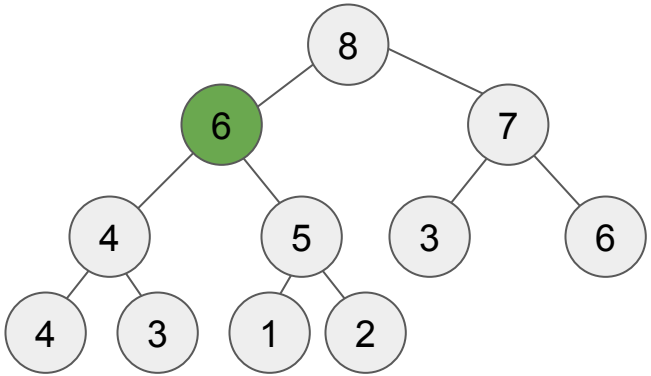
Now, we *fix up* from that location

- Is it larger than its parent?
  - **If so, swap and repeat**
  - Otherwise, you're done.

8	6	7	4	5	3	6	4	3	1	2

Priority Queue - Implementations - Binary - Push

To insert an element into the PQ, we start by pushing it to the end of the vector.



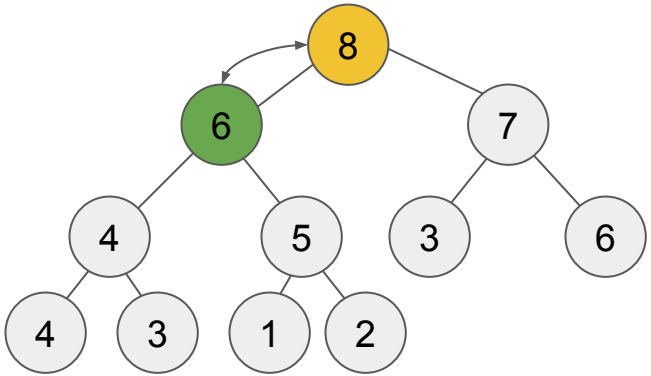
Now, we *fix up* from that location

- Is it larger than its parent?
  - If so, swap and **repeat**
  - Otherwise, you're done.

8	6	7	4	5	3	6	4	3	1	2

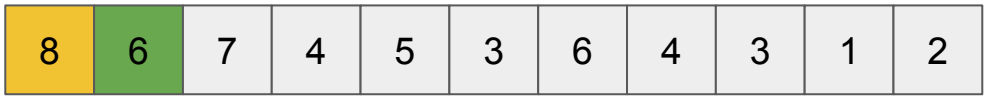
Priority Queue - Implementations - Binary - Push

To insert an element into the PQ, we start by pushing it to the end of the vector.



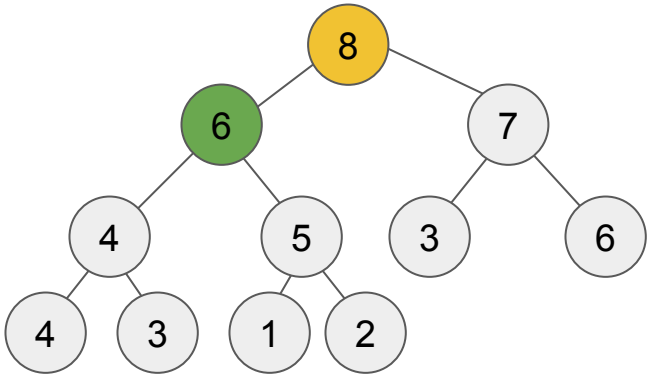
Now, we *fix up* from that location

- **Is it larger than its parent?**
  - If so, swap and repeat
  - Otherwise, you're done.



Priority Queue - Implementations - Binary - Push

To insert an element into the PQ, we start by pushing it to the end of the vector.



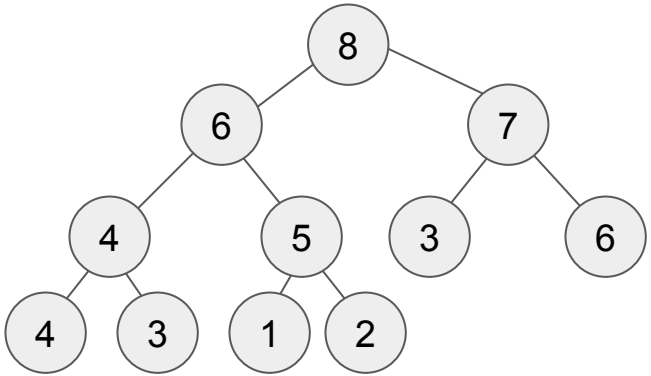
Now, we *fix up* from that location

- Is it larger than its parent?
  - If so, swap and repeat
  - **Otherwise, you're done.**

8	6	7	4	5	3	6	4	3	1	2

Priority Queue - Implementations - Binary - Push

To insert an element into the PQ, we start by pushing it to the end of the vector.



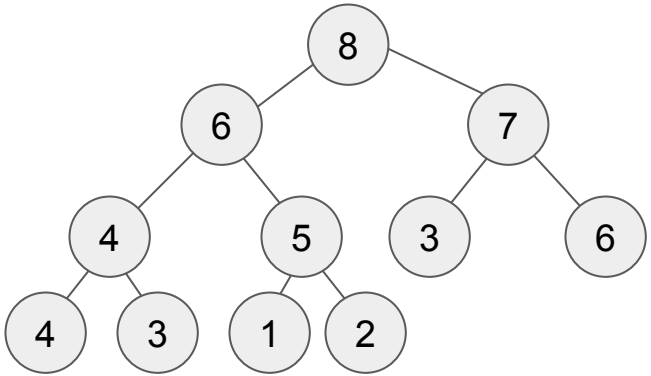
Now, we *fix up* from that location

- Is it larger than its parent?
  - If so, swap and repeat
  - Otherwise, you're done.
- **Done**

8	6	7	4	5	3	6	4	3	1	2

Priority Queue - Implementations - Binary - Pop

Popping is only slightly more complicated. We want to remove the *root* which is harder, since we can't just remove the 0 index without sliding everything down and being very slow to fix.



So, we start by swapping the root with the last element.

Then, we pop back to remove the old root.

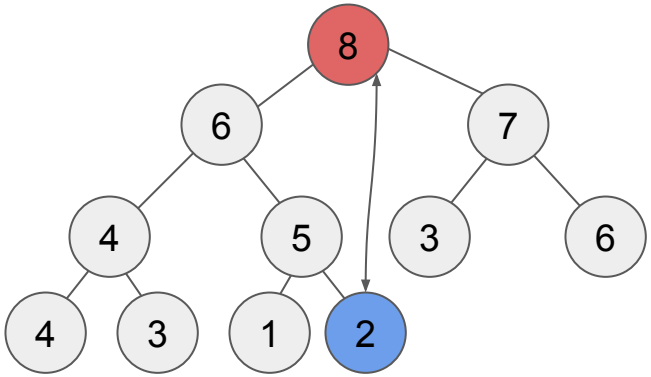
Now we *fix down* starting at the new root.

- Is it smaller than either child?
  - If so, swap with the larger, repeat on that new position.
  - Otherwise, you're done.

8	6	7	4	5	3	6	4	3	1	2

Priority Queue - Implementations - Binary - Pop

Popping is only slightly more complicated. We want to remove the *root* which is harder, since we can't just remove the 0 index without sliding everything down and being very slow to fix.



So, we start by swapping the root with the last element.

Then, we pop back to remove the old root.

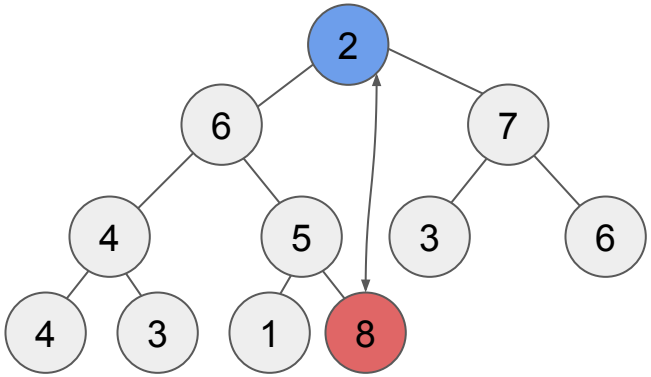
Now we *fix down* starting at the new root.

- Is it smaller than either child?
  - If so, swap with the larger, repeat on that new position.
  - Otherwise, you're done.

8	6	7	4	5	3	6	4	3	1	2

Priority Queue - Implementations - Binary - Pop

Popping is only slightly more complicated. We want to remove the *root* which is harder, since we can't just remove the 0 index without sliding everything down and being very slow to fix.

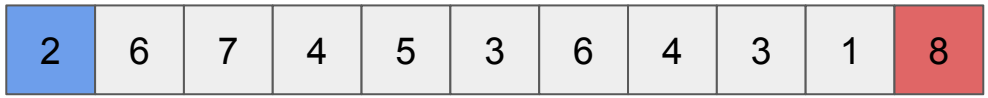


So, we start by swapping the root with the last element.

Then, we pop back to remove the old root.

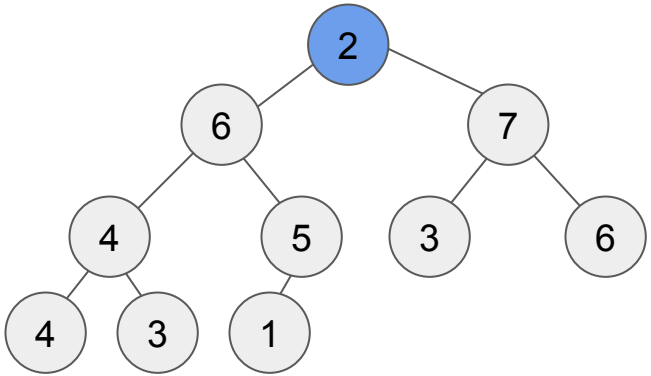
Now we *fix down* starting at the new root.

- Is it smaller than either child?
  - If so, swap with the larger, repeat on that new position.
  - Otherwise, you're done.



Priority Queue - Implementations - Binary - Pop

Popping is only slightly more complicated. We want to remove the *root* which is harder, since we can't just remove the 0 index without sliding everything down and being very slow to fix.



So, we start by swapping the root with the last element.

Then, we pop back to remove the old root.

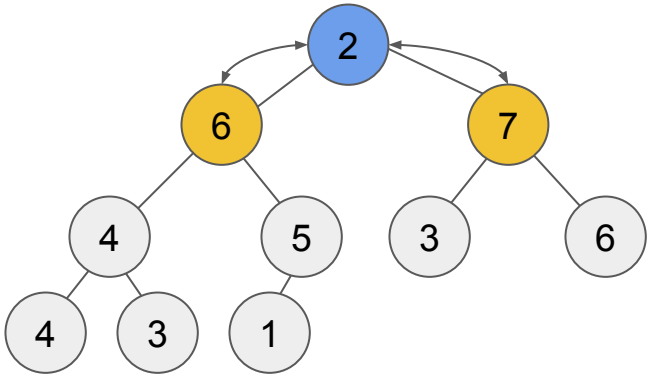
Now we *fix down* starting at the new root.

- Is it smaller than either child?
  - If so, swap with the larger, repeat on that new position.
  - Otherwise, you're done.

2	6	7	4	5	3	6	4	3	1

Priority Queue - Implementations - Binary - Pop

Popping is only slightly more complicated. We want to remove the *root* which is harder, since we can't just remove the 0 index without sliding everything down and being very slow to fix.

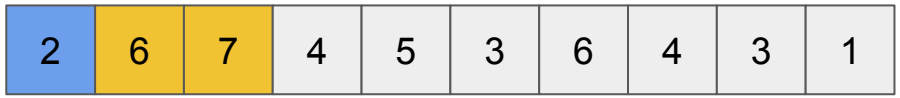


So, we start by swapping the root with the last element.

Then, we pop back to remove the old root.

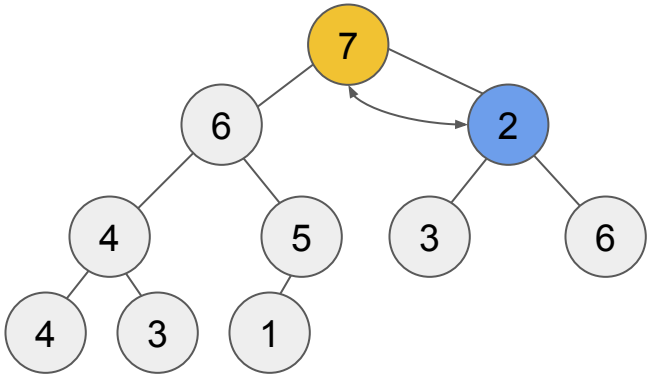
Now we *fix down* starting at the new root.

- **Is it smaller than either child?**
  - If so, swap with the larger, repeat on that new position.
  - Otherwise, you're done.



Priority Queue - Implementations - Binary - Pop

Popping is only slightly more complicated. We want to remove the *root* which is harder, since we can't just remove the 0 index without sliding everything down and being very slow to fix.

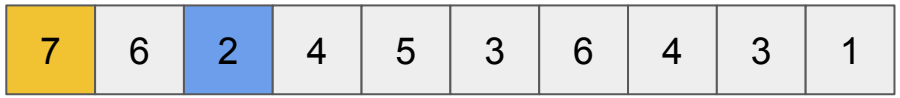


So, we start by swapping the root with the last element.

Then, we pop back to remove the old root.

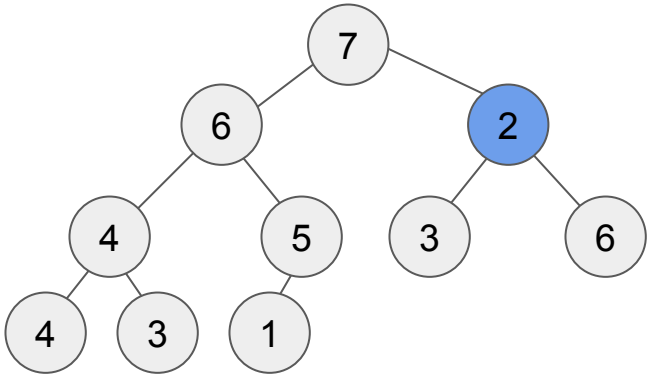
Now we *fix down* starting at the new root.

- Is it smaller than either child?
  - **If so, swap with the larger, repeat on that new position**
  - Otherwise, you're done.



Priority Queue - Implementations - Binary - Pop

Popping is only slightly more complicated. We want to remove the *root* which is harder, since we can't just remove the 0 index without sliding everything down and being very slow to fix.

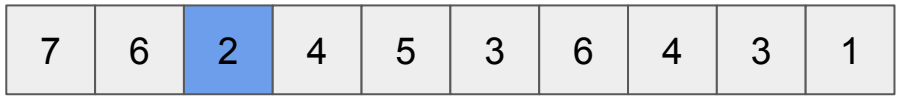


So, we start by swapping the root with the last element.

Then, we pop back to remove the old root.

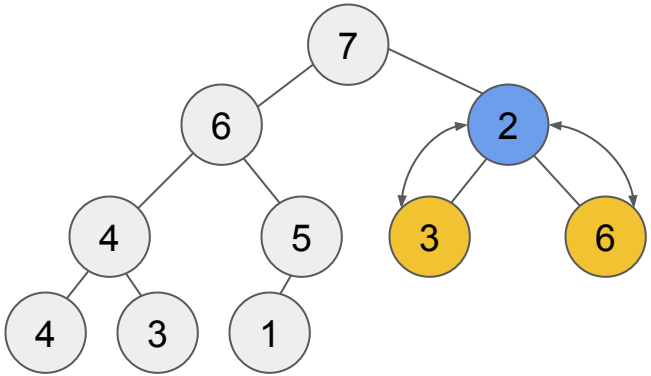
Now we *fix down* starting at the new root.

- Is it smaller than either child?
  - If so, swap with the larger, **repeat on that new position.**
  - Otherwise, you're done.



Priority Queue - Implementations - Binary - Pop

Popping is only slightly more complicated. We want to remove the *root* which is harder, since we can't just remove the 0 index without sliding everything down and being very slow to fix.

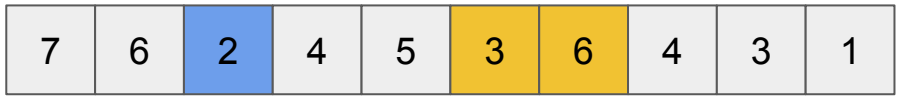


So, we start by swapping the root with the last element.

Then, we pop back to remove the old root.

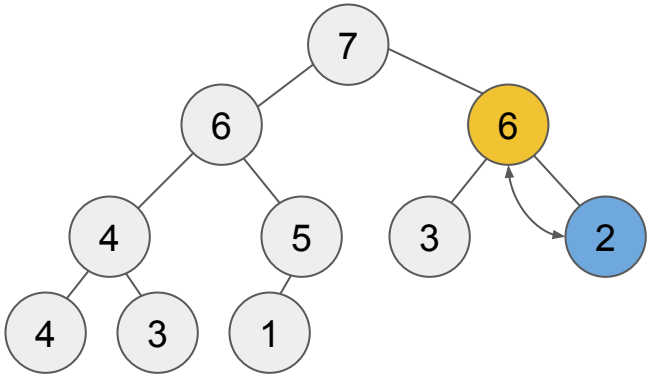
Now we *fix down* starting at the new root.

- **Is it smaller than either child?**
  - If so, swap with the larger, repeat on that new position.
  - Otherwise, you're done.



Priority Queue - Implementations - Binary - Pop

Popping is only slightly more complicated. We want to remove the *root* which is harder, since we can't just remove the 0 index without sliding everything down and being very slow to fix.

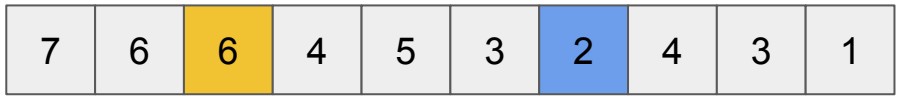


So, we start by swapping the root with the last element.

Then, we pop back to remove the old root.

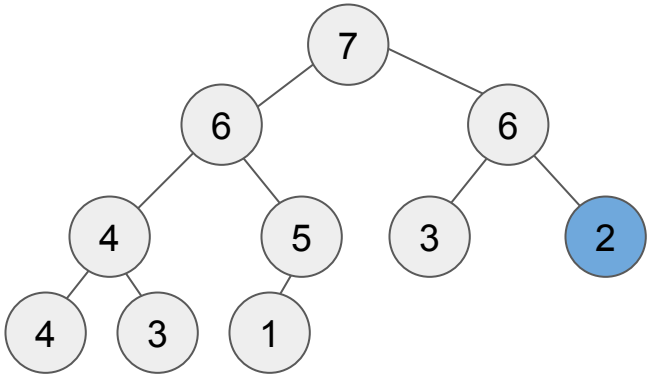
Now we *fix down* starting at the new root.

- Is it smaller than either child?
  - **If so, swap with the larger, repeat on that new position**
  - Otherwise, you're done.



Priority Queue - Implementations - Binary - Pop

Popping is only slightly more complicated. We want to remove the *root* which is harder, since we can't just remove the 0 index without sliding everything down and being very slow to fix.

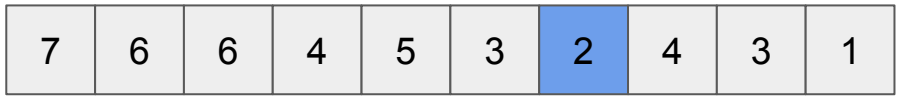


So, we start by swapping the root with the last element.

Then, we pop back to remove the old root.

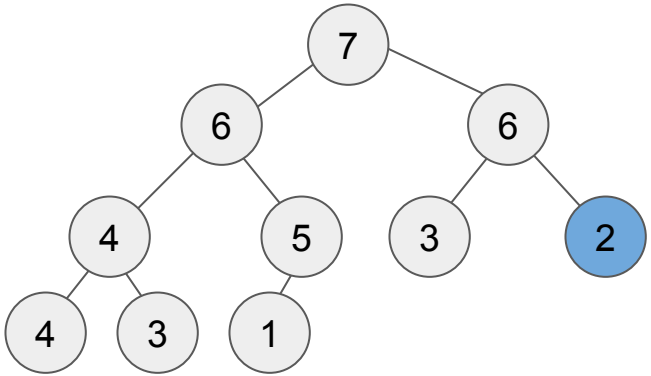
Now we *fix down* starting at the new root.

- Is it smaller than either child?
  - If so, swap with the larger, **repeat on that new position.**
  - Otherwise, you're done.



Priority Queue - Implementations - Binary - Pop

Popping is only slightly more complicated. We want to remove the *root* which is harder, since we can't just remove the 0 index without sliding everything down and being very slow to fix.



So, we start by swapping the root with the last element.

Then, we pop back to remove the old root.

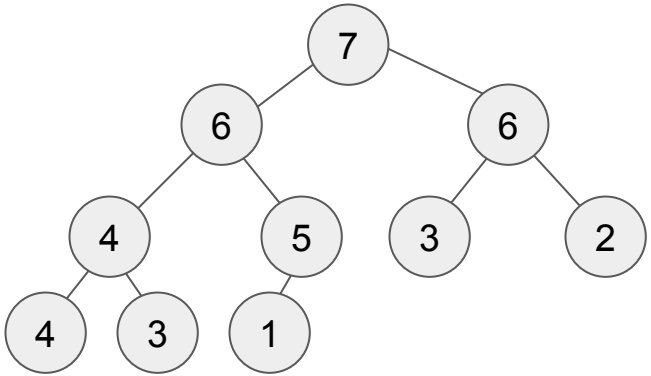
Now we *fix down* starting at the new root.

- **Is it smaller than either child? *It has no children***
  - If so, swap with the larger, repeat on that new position.
  - Otherwise, you're done.

7	6	6	4	5	3	2	4	3	1

Priority Queue - Implementations - Binary - Pop

Popping is only slightly more complicated. We want to remove the *root* which is harder, since we can't just remove the 0 index without sliding everything down and being very slow to fix.



So, we start by swapping the root with the last element.

Then, we pop back to remove the old root.

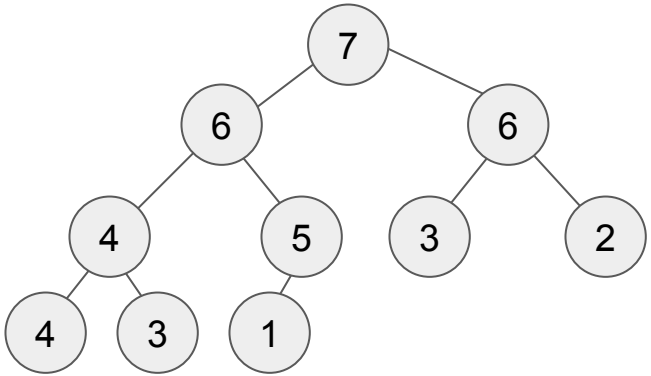
Now we *fix down* starting at the new root.

- Is it smaller than either child? *It has no children*
  - If so, swap with the larger, repeat on that new position.
  - **Otherwise, you're done.**

7	6	6	4	5	3	2	4	3	1

Priority Queue - Implementations - Binary - Pop

Popping is only slightly more complicated. We want to remove the *root* which is harder, since we can't just remove the 0 index without sliding everything down and being very slow to fix.



So, we start by swapping the root with the last element.

Then, we pop back to remove the old root.

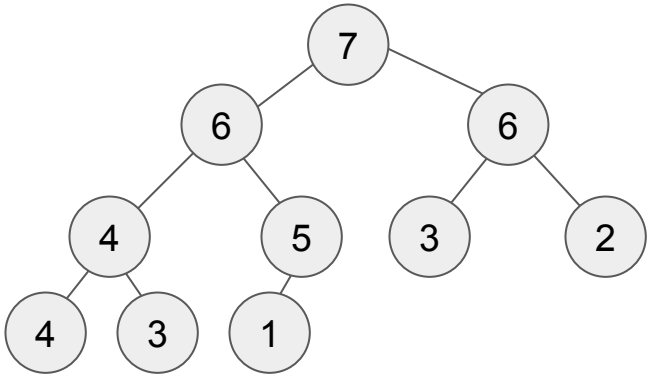
Now we *fix down* starting at the new root.

- Is it smaller than either child? *It has no children*
  - If so, swap with the larger, repeat on that new position.
  - Otherwise, you're done.
- **Done**

7	6	6	4	5	3	2	4	3	1

Priority Queue - Implementations - Binary - Time

How fast do these operations run?



top() -> returns data[0] in **O(1) time**

push() ->
- push_back + fix_up;
- fix_up runs in time proportional to tree height,
- Tree height is $\log_2(n)$
- **O(log n)**

pop() ->
- Swap, pop_back, fix_down;
- Fix_down runs in time proportional to tree height,
- **O(log n)**

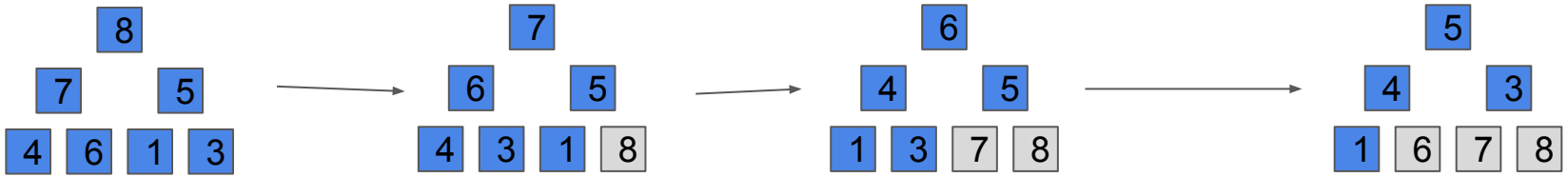
7	6	6	4	5	3	2	4	3	1

Heapsort

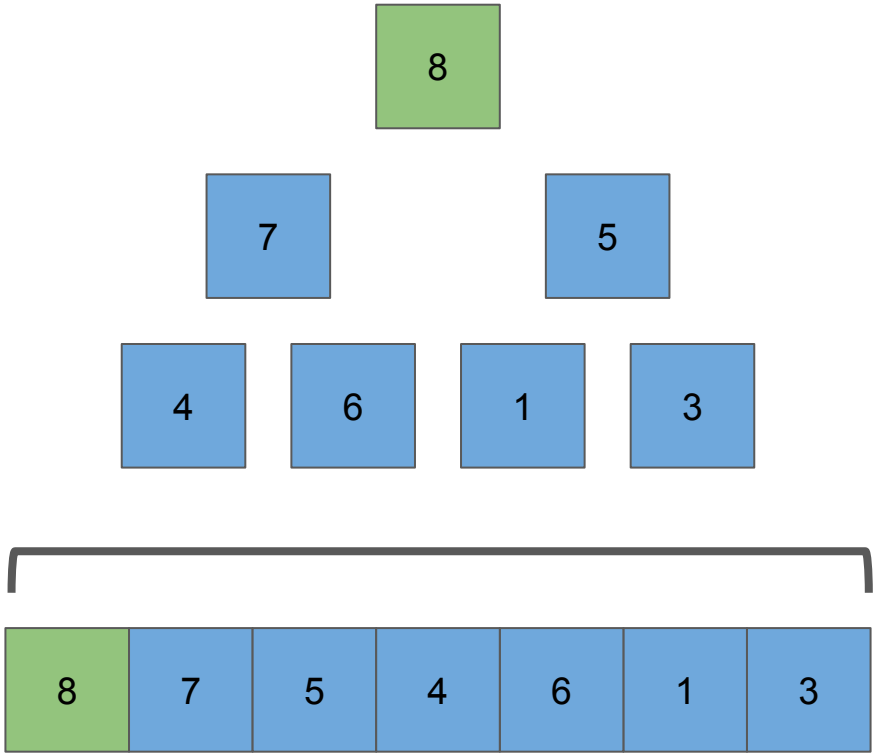
- Heapify + one more step
  - Heapify is just “building” the heap—takes O(n) time
  - To sort, you then need to pop each item off of the heap
- Not stable
- Worst-case O(nlog(n))
- In place

Heapsort

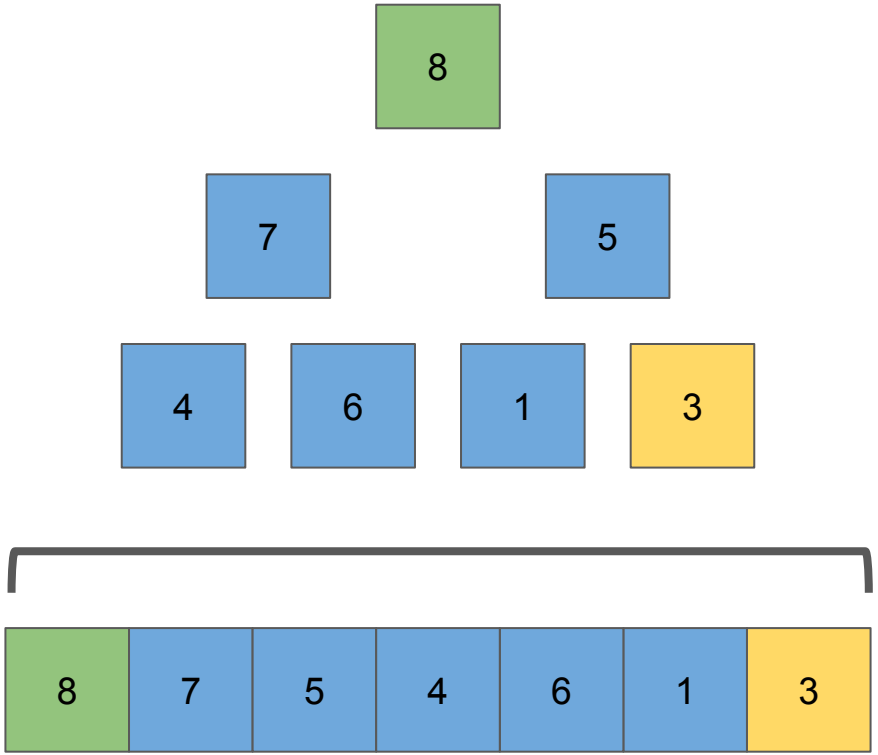
- Heapify the array, constructing a **max** heap
- Take the top element, swap it with the last element in the heap.
  - After the swap, the last element is no longer “in” the heap (but it’s still in the array!)
- Call `fix_down` on the new top element, so the new max is placed there
- Repeat until the heap is empty
- Construct a min heap with the array
- Best, worst, and average-case: O(n log n)
- Memory: O(1) additional - it’s in-place



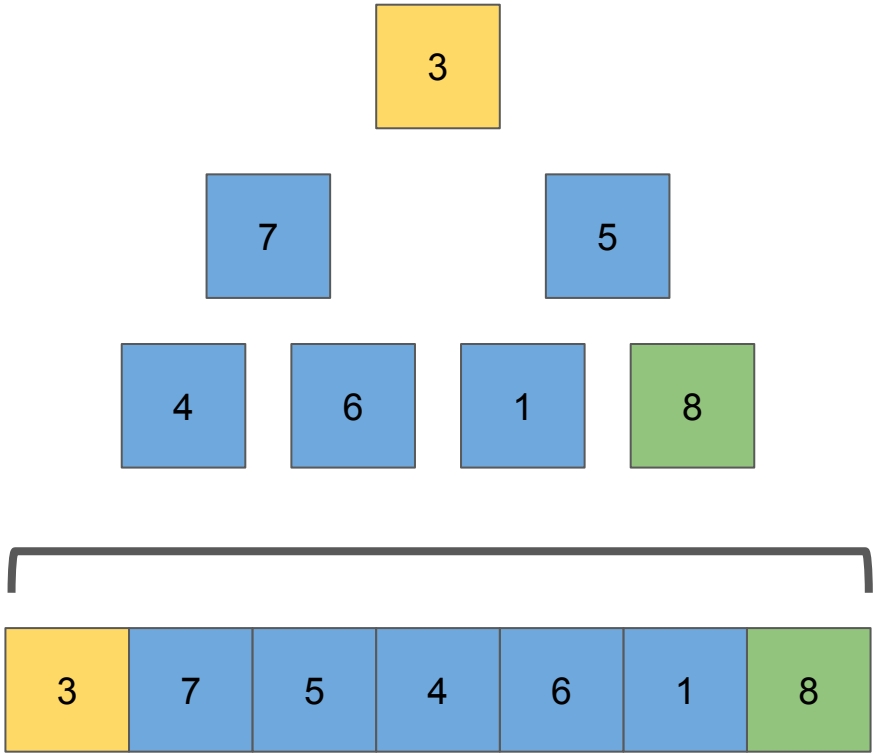
Heapsort



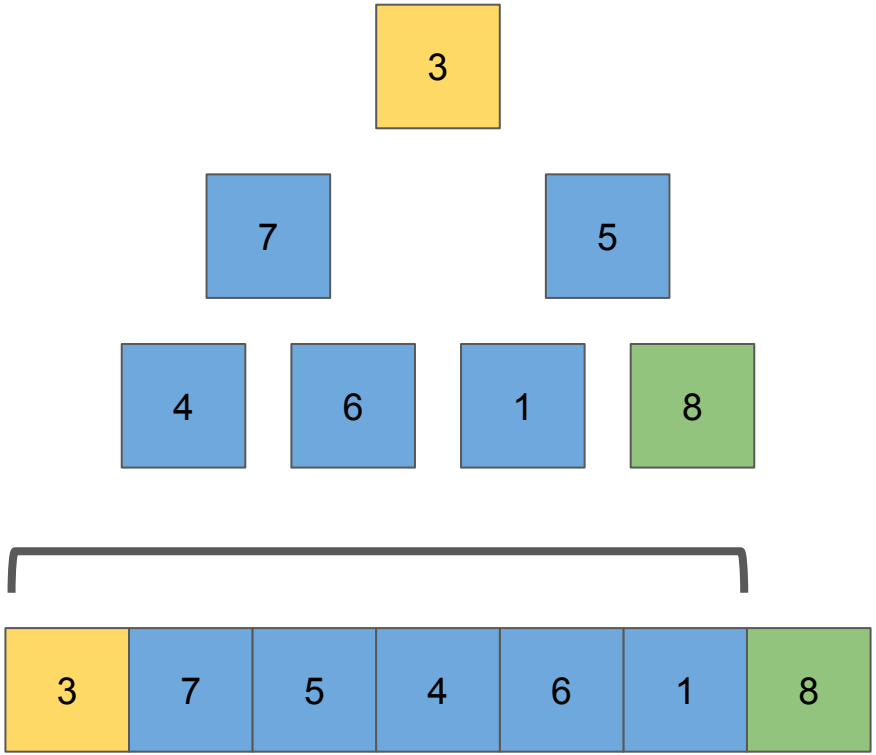
Heapsort



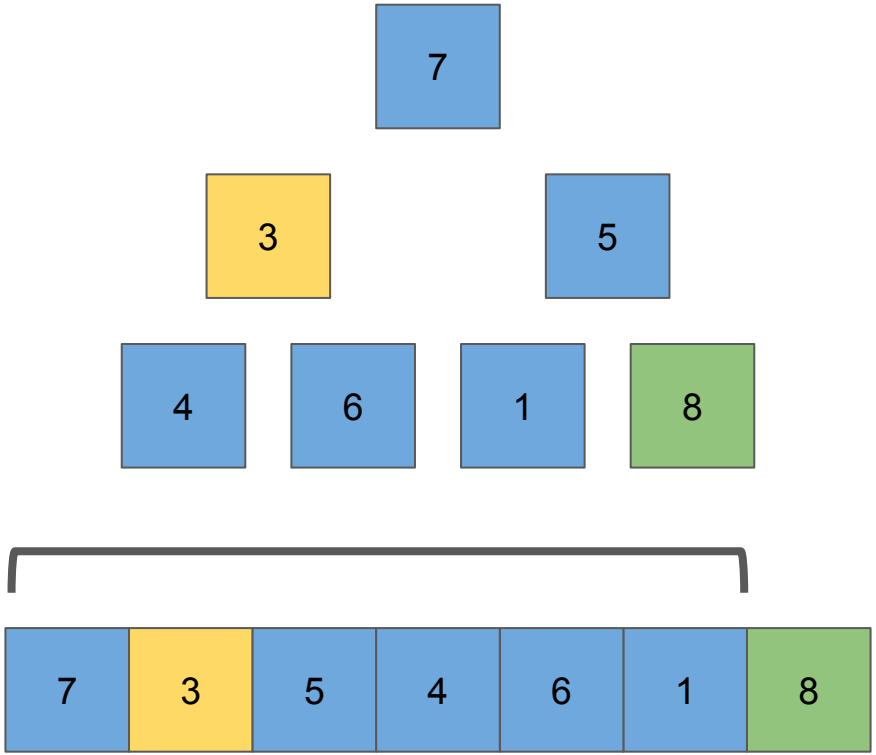
Heapsort



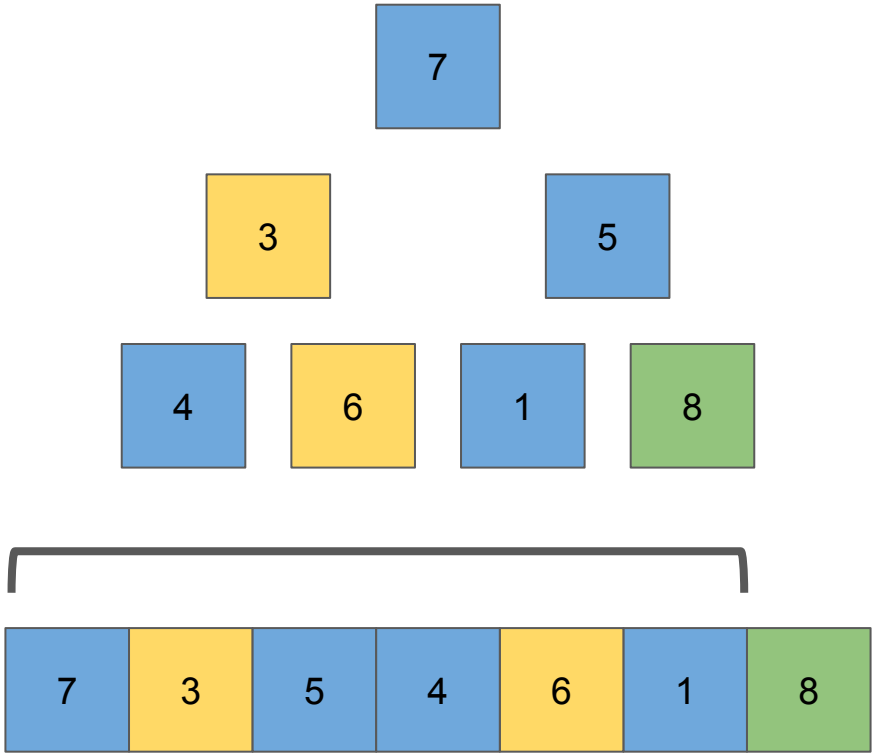
Heapsort



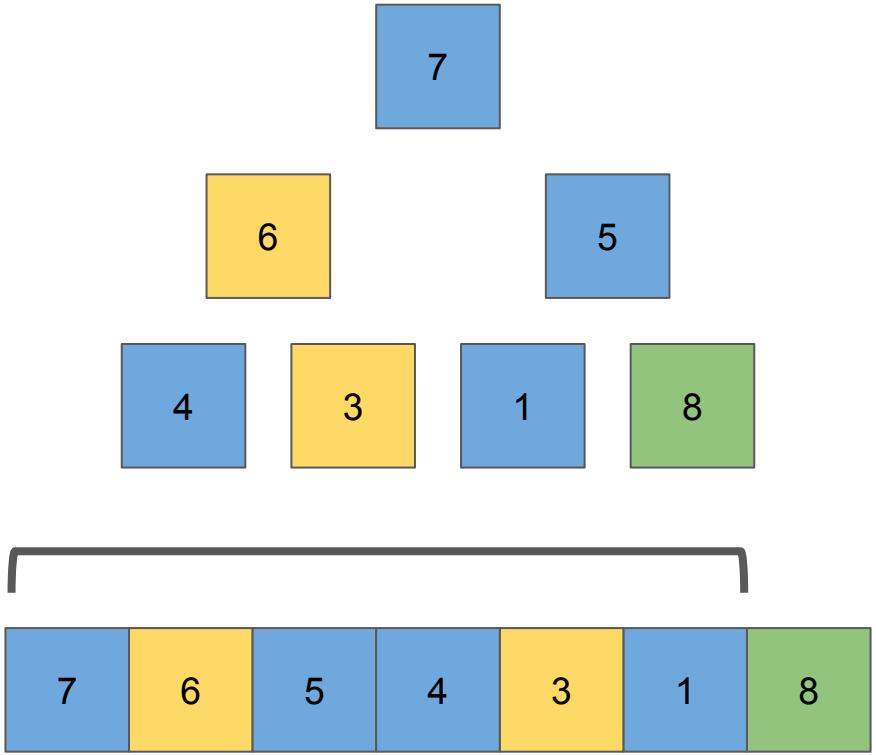
Heapsort



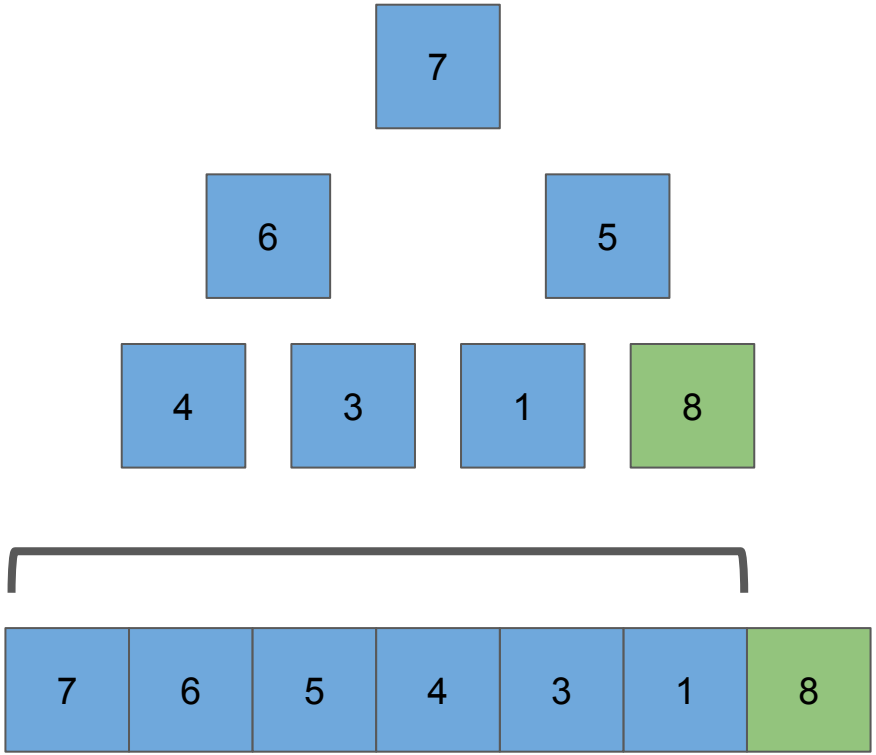
Heapsort



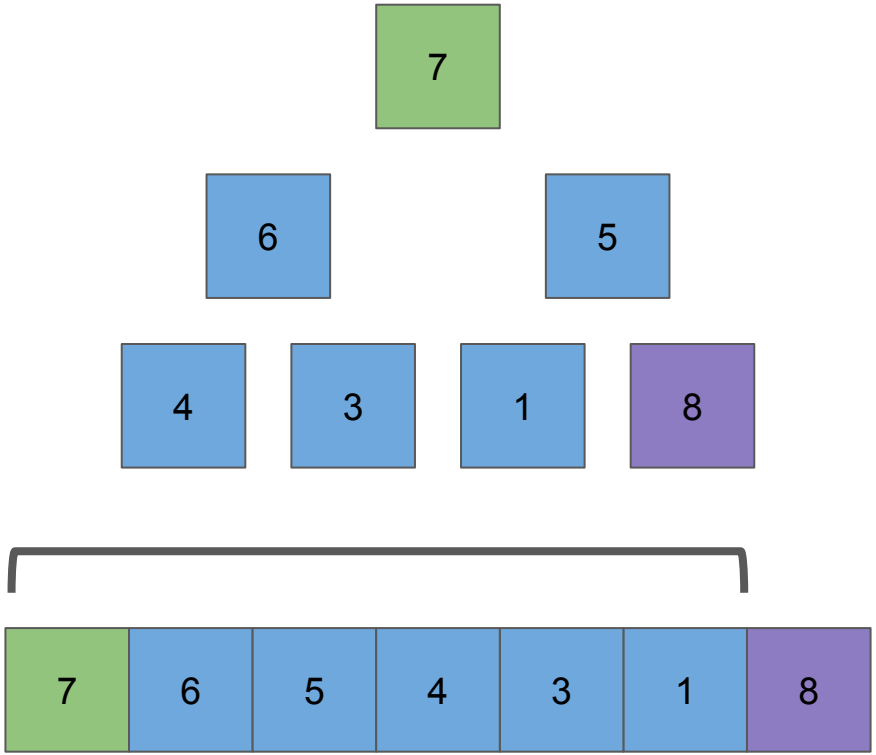
Heapsort



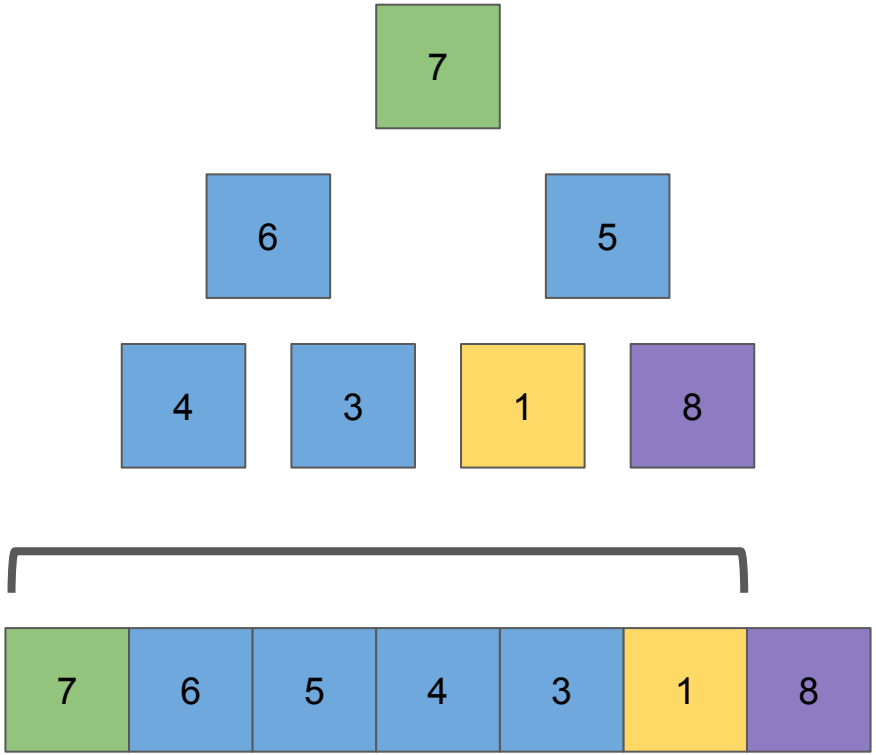
Heapsort



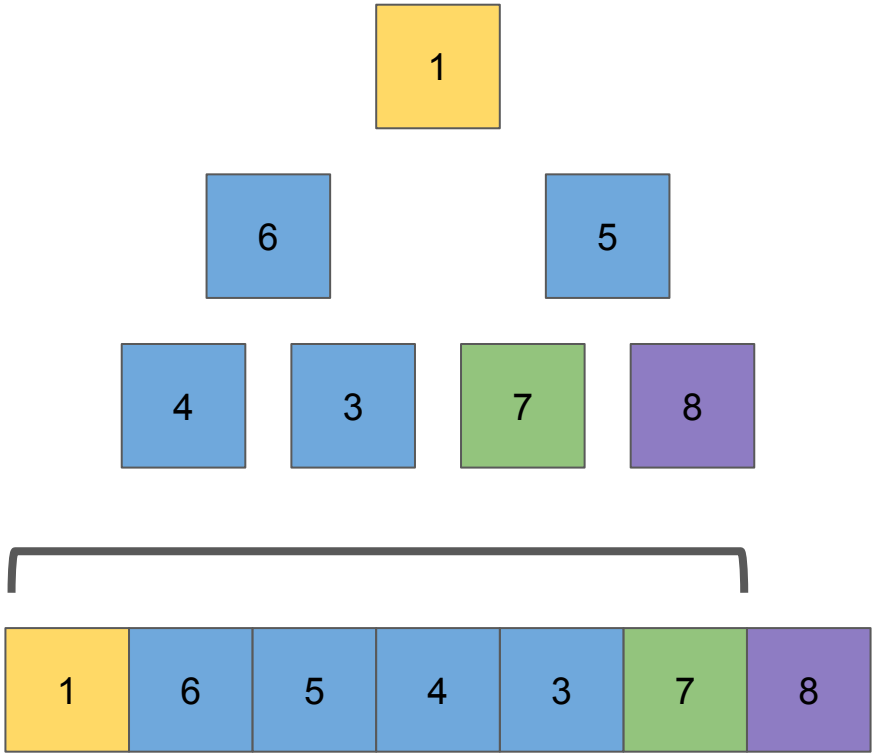
Heapsort



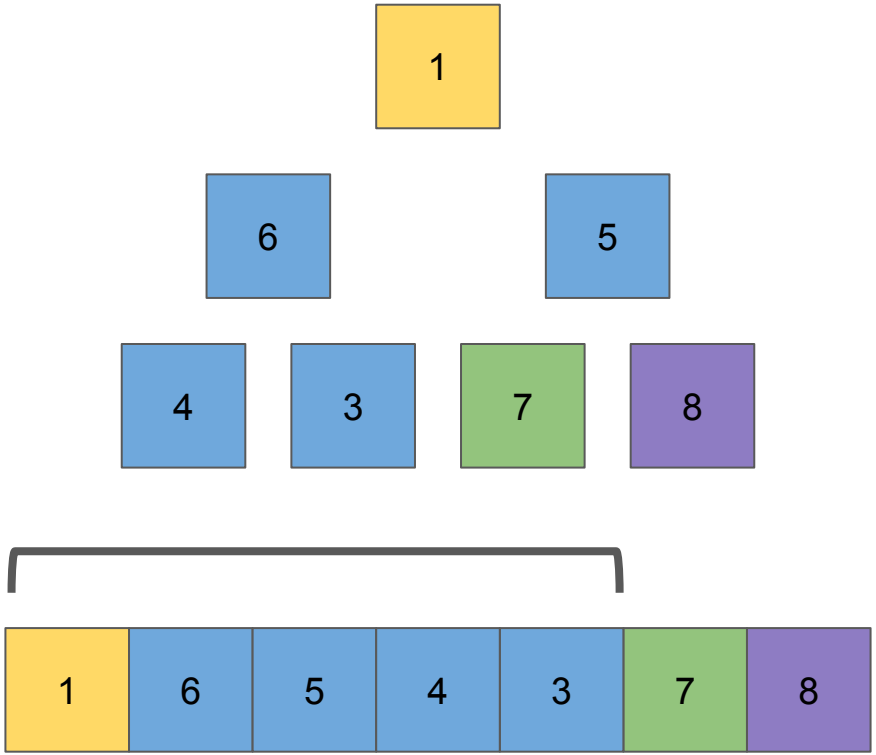
Heapsort



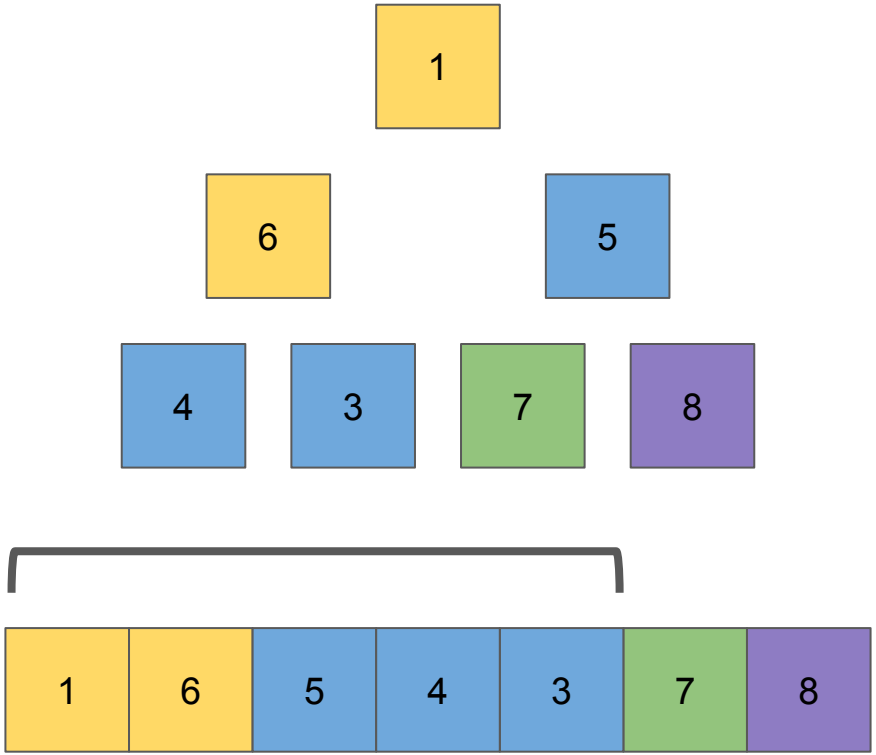
Heapsort



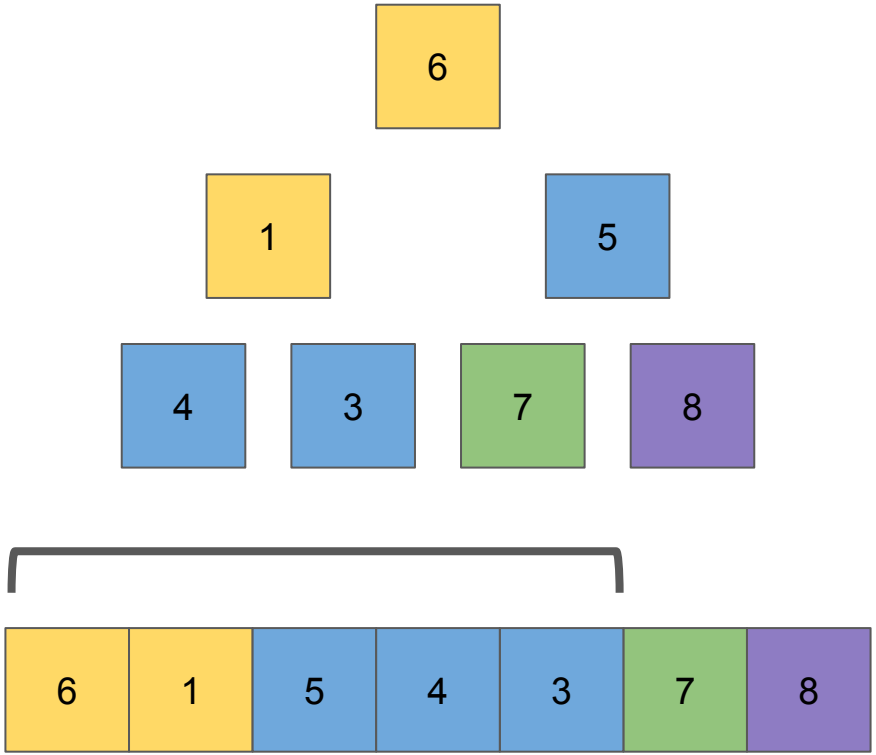
Heapsort



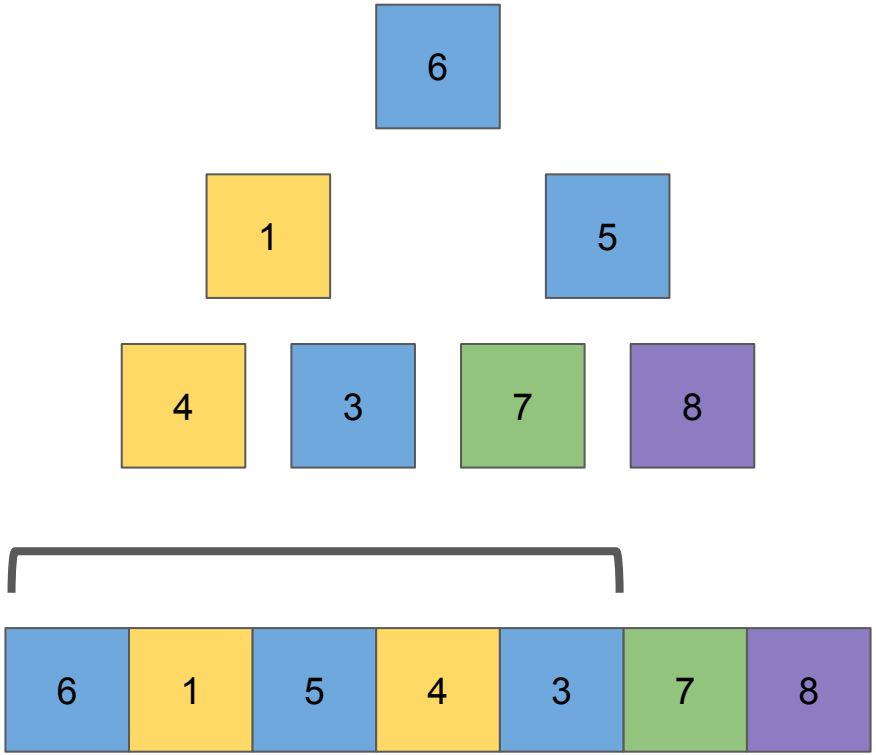
Heapsort



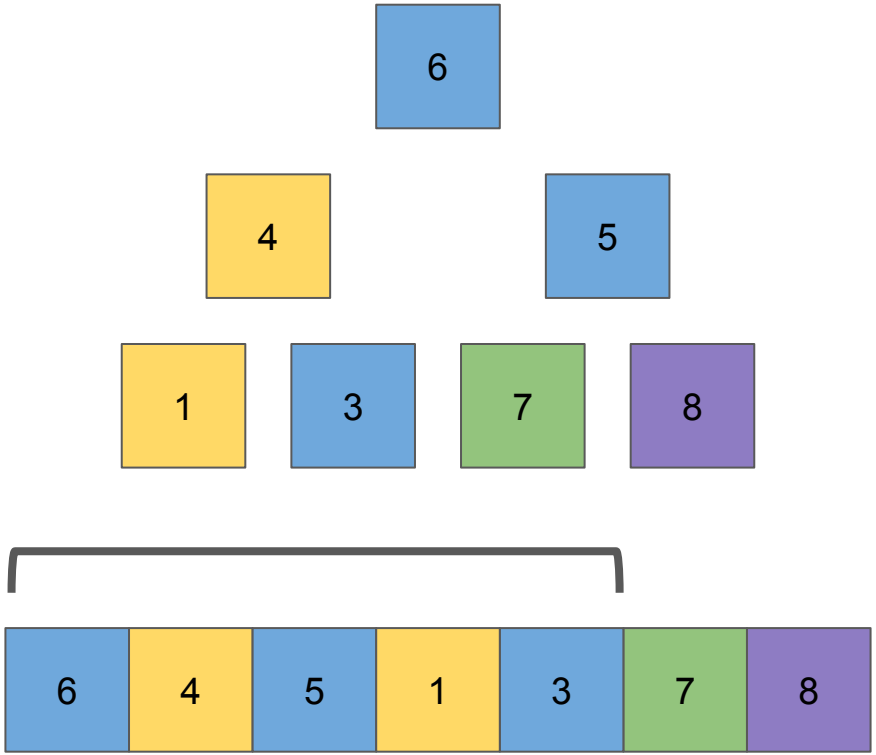
Heapsort



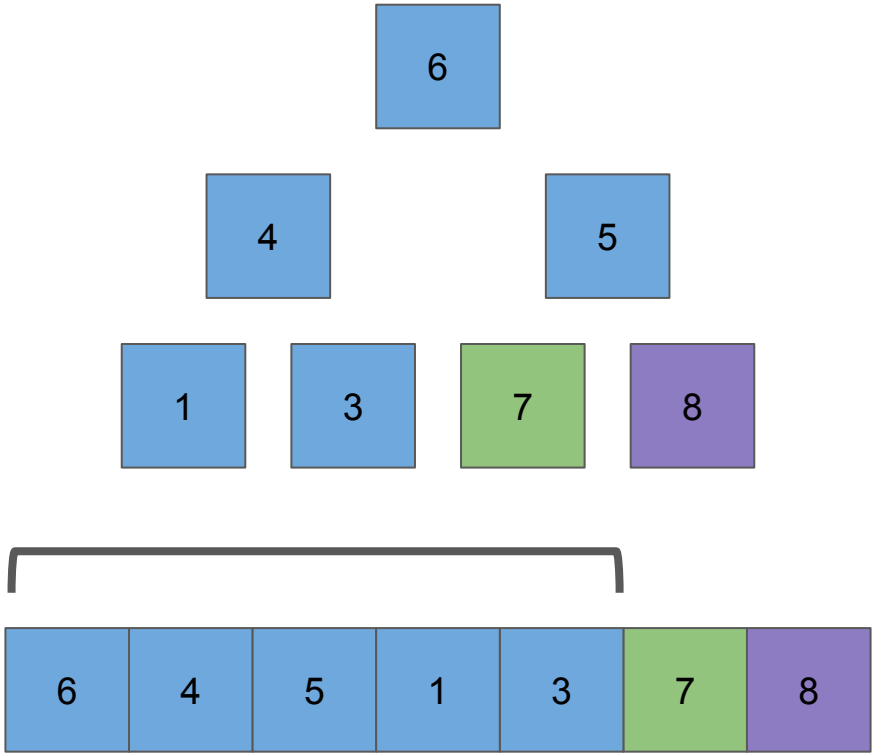
Heapsort



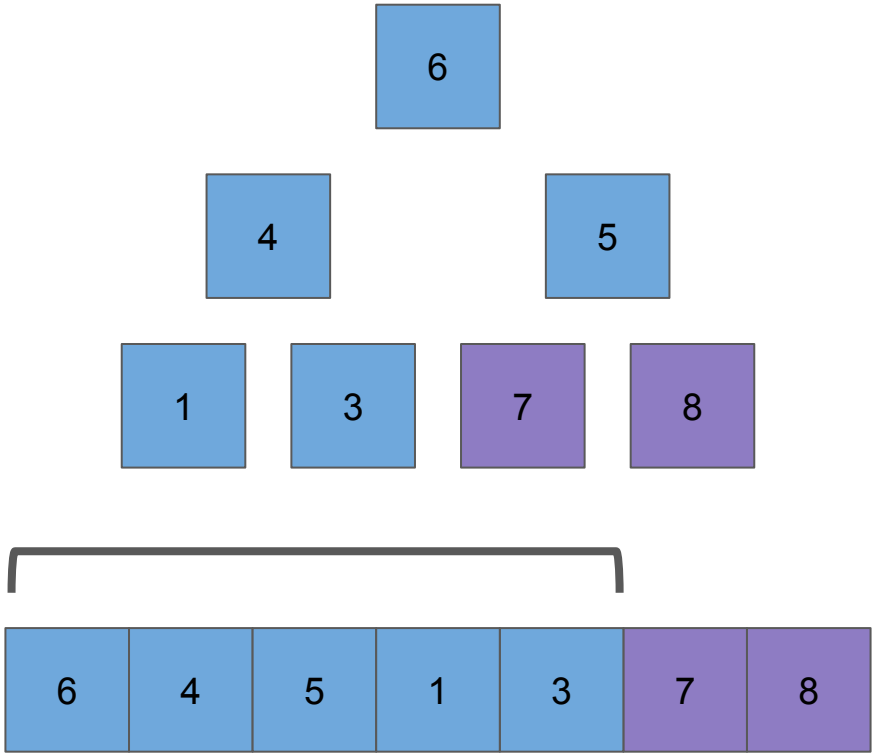
Heapsort



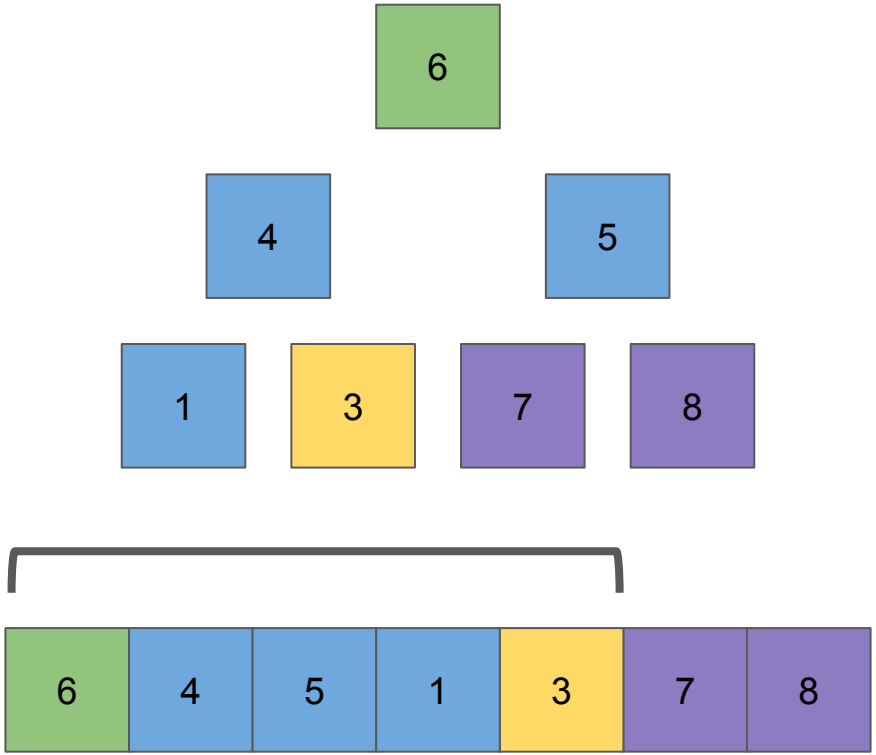
Heapsort



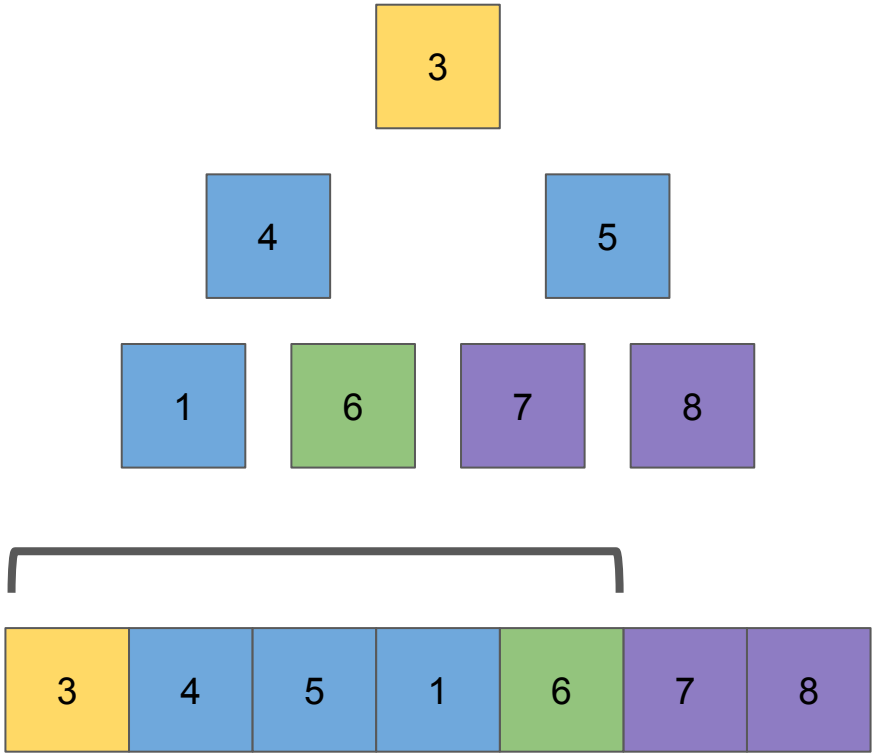
Heapsort



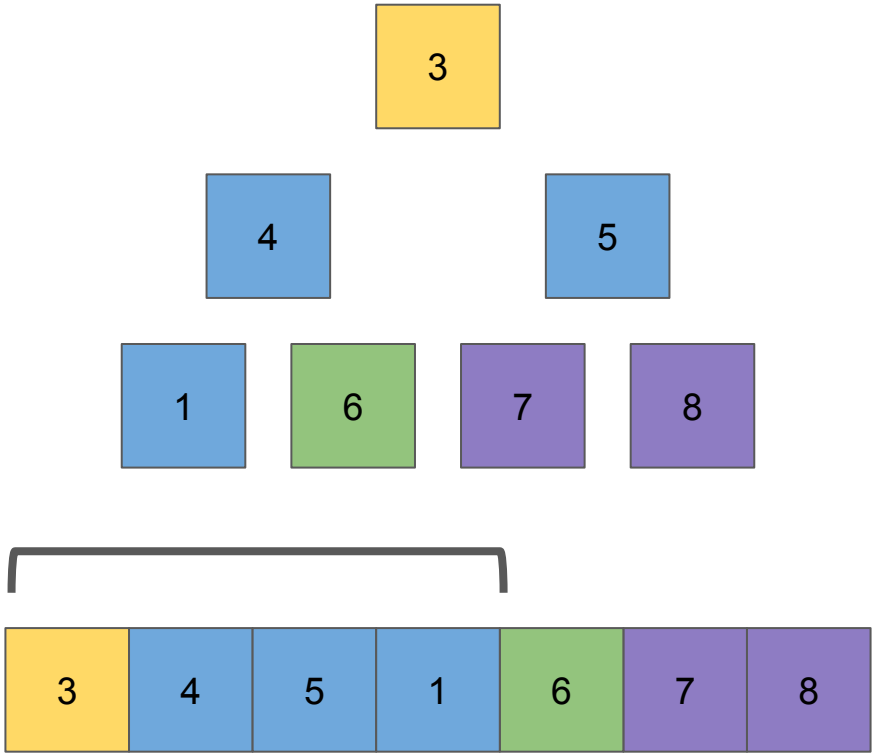
Heapsort



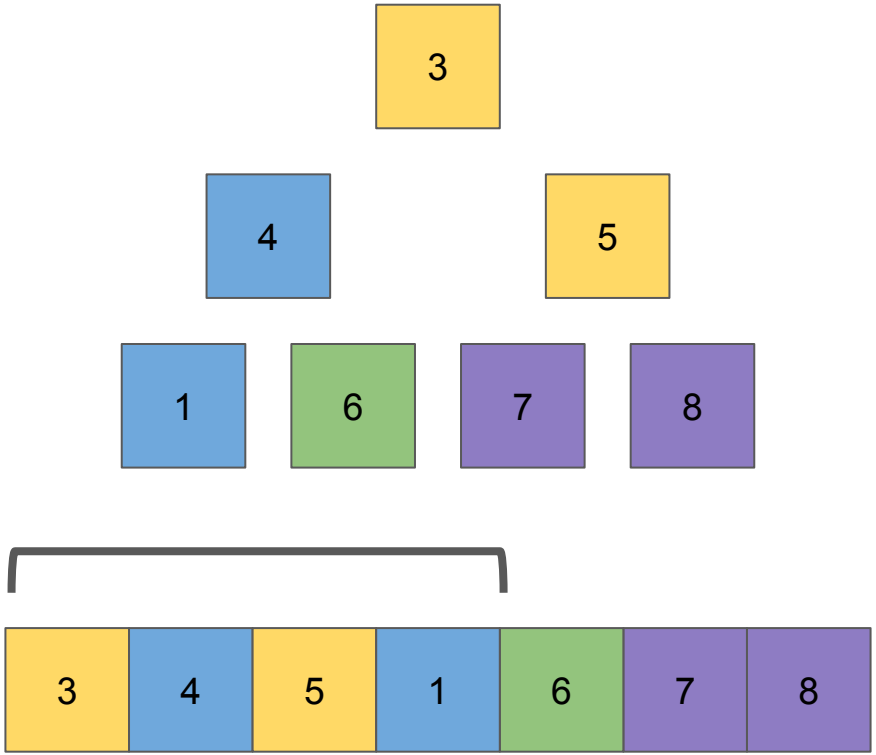
Heapsort



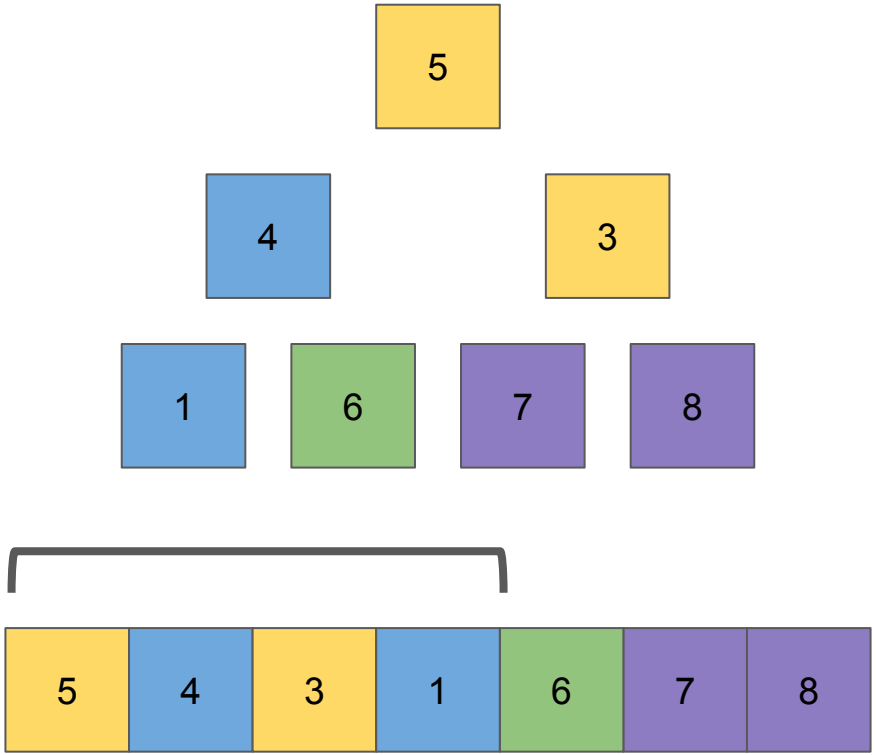
Heapsort



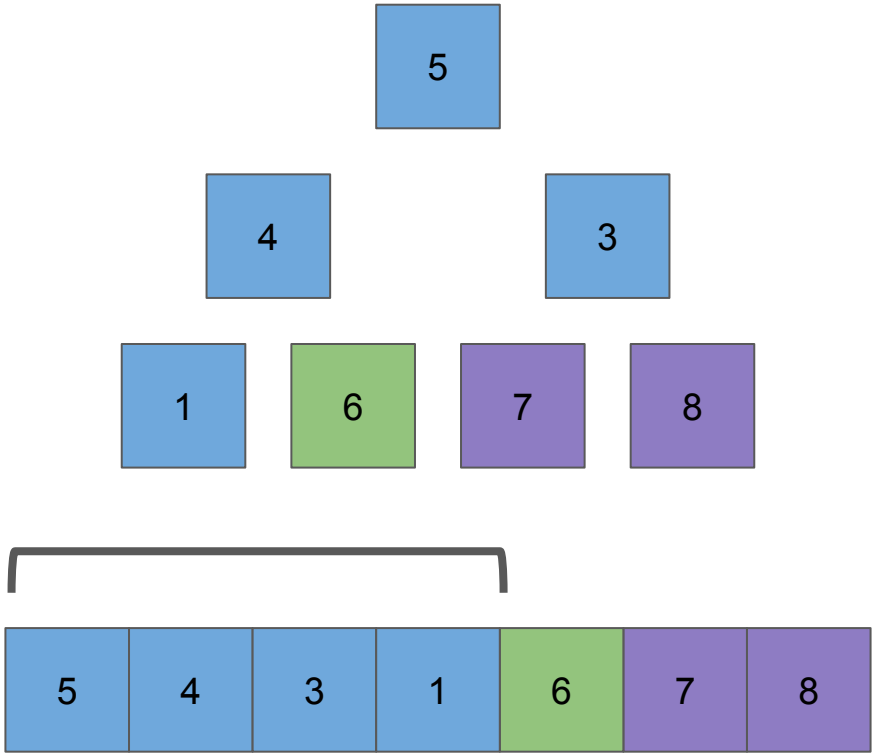
Heapsort



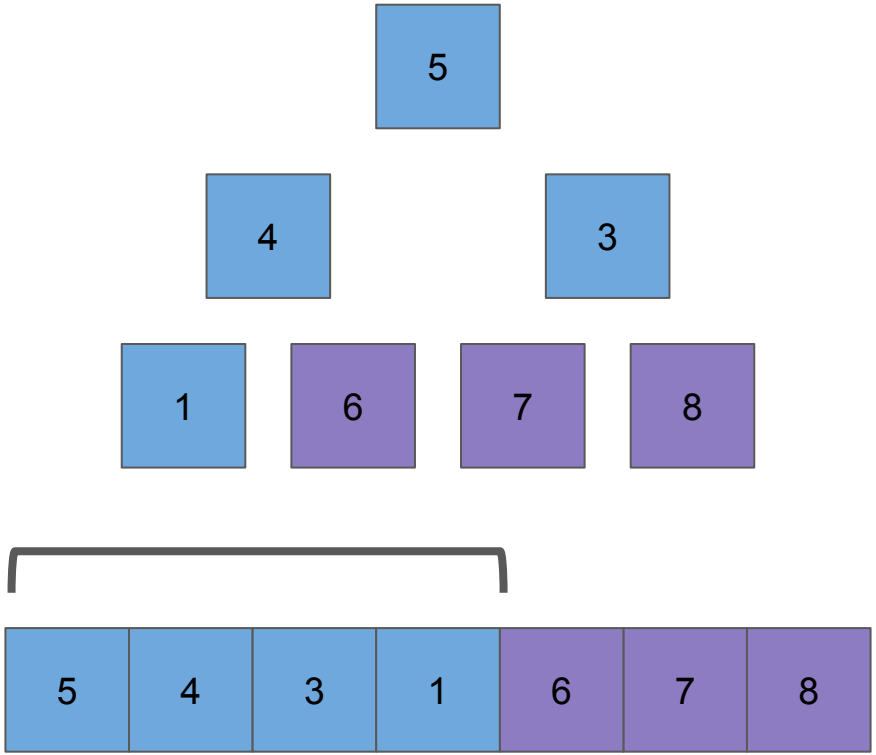
Heapsort



Heapsort



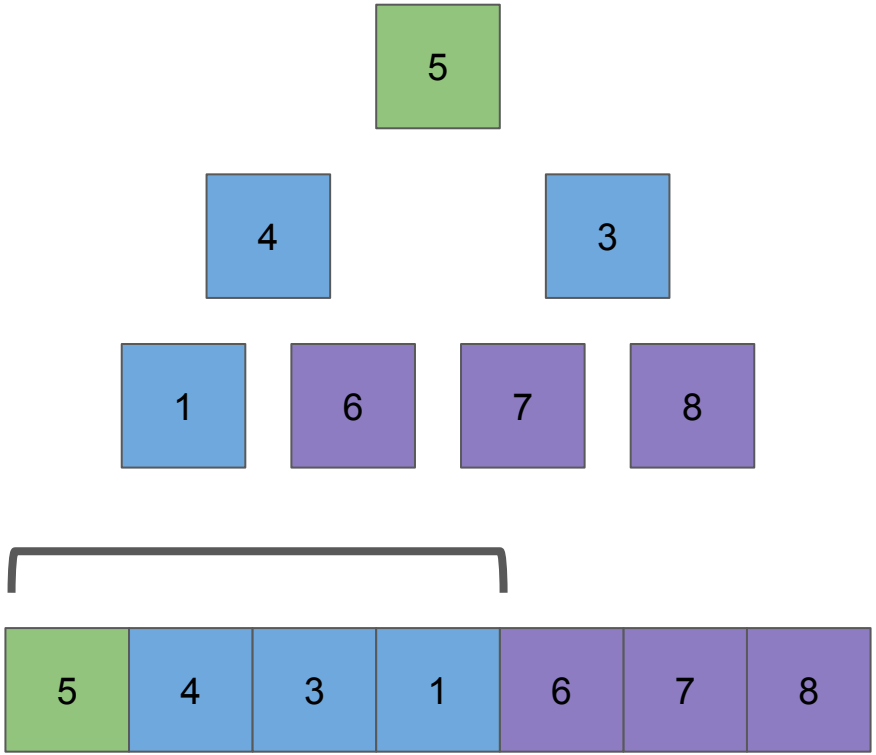
Heapsort



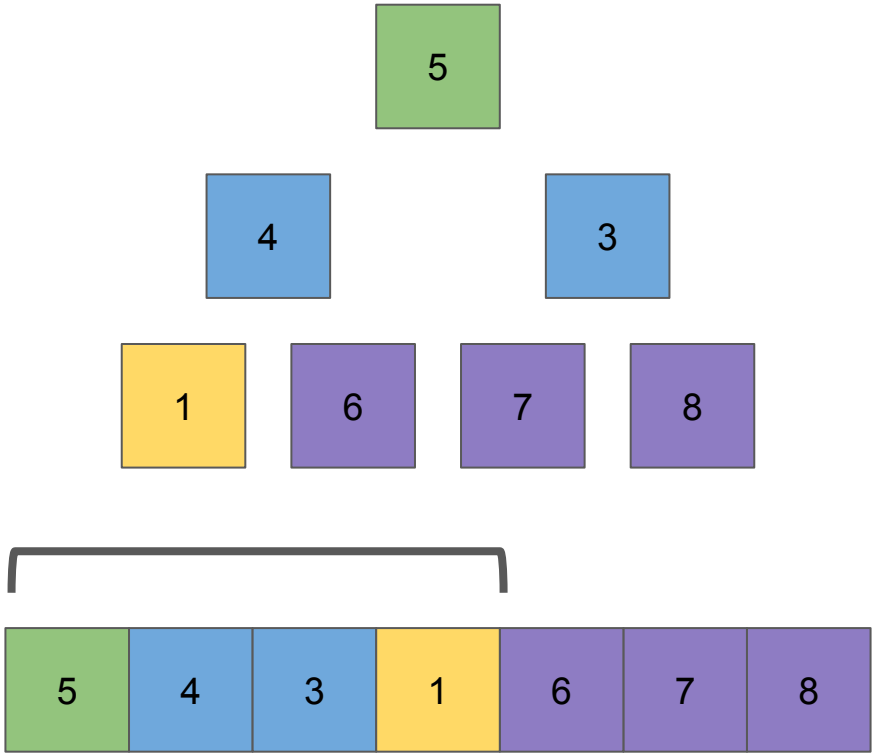
Preparing for the exam:

- Know how to represent heap as a vector
- Know how heapify and heapsort work
  - Complexity of each operation
- Practice problems inserting into and removing from a heap

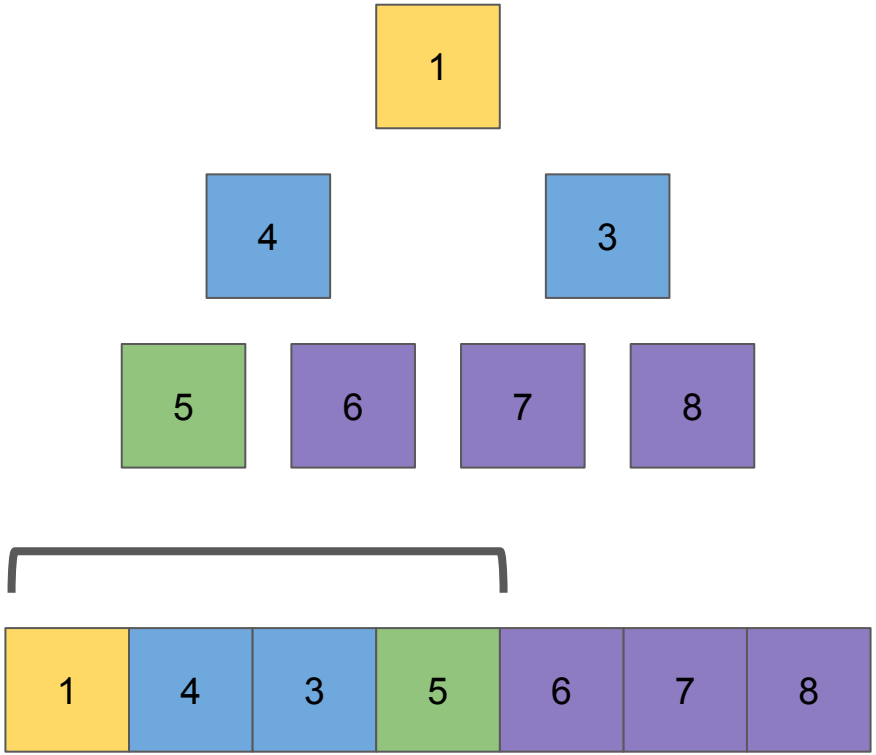
Heapsort



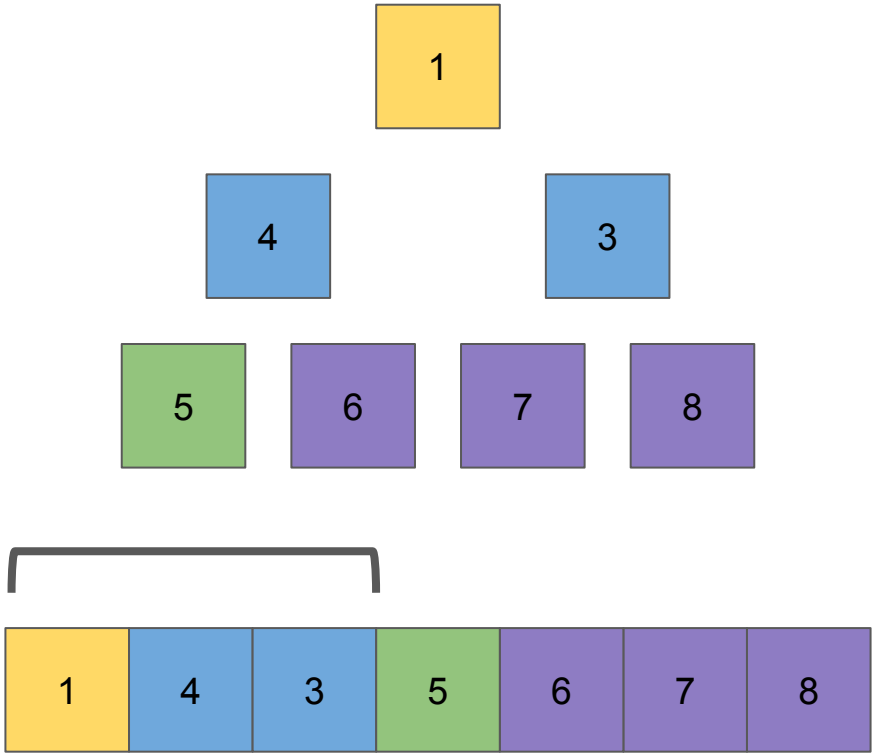
Heapsort



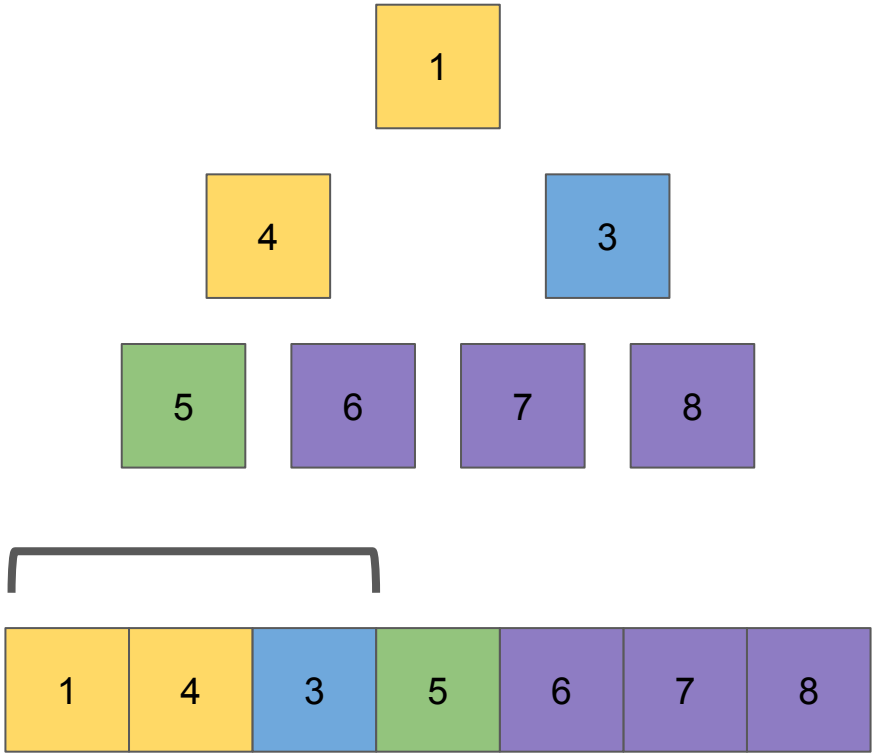
Heapsort



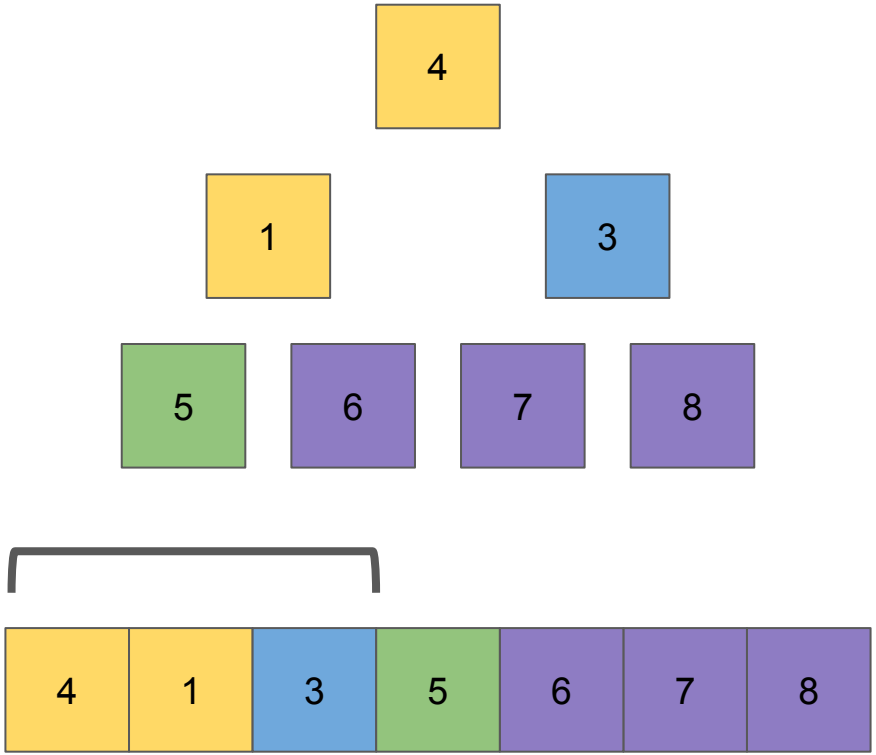
Heapsort



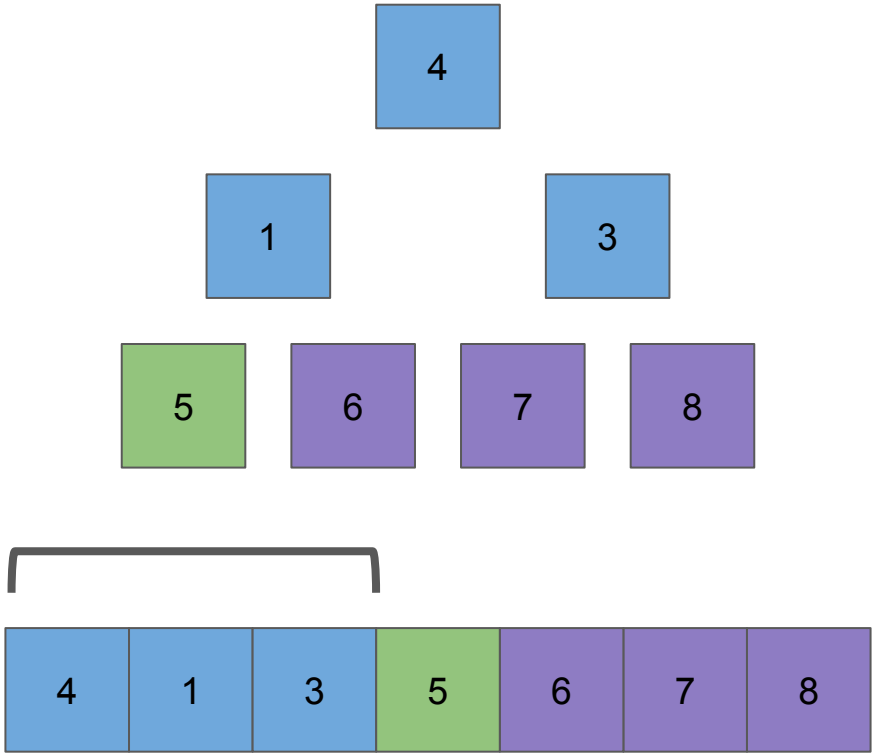
Heapsort



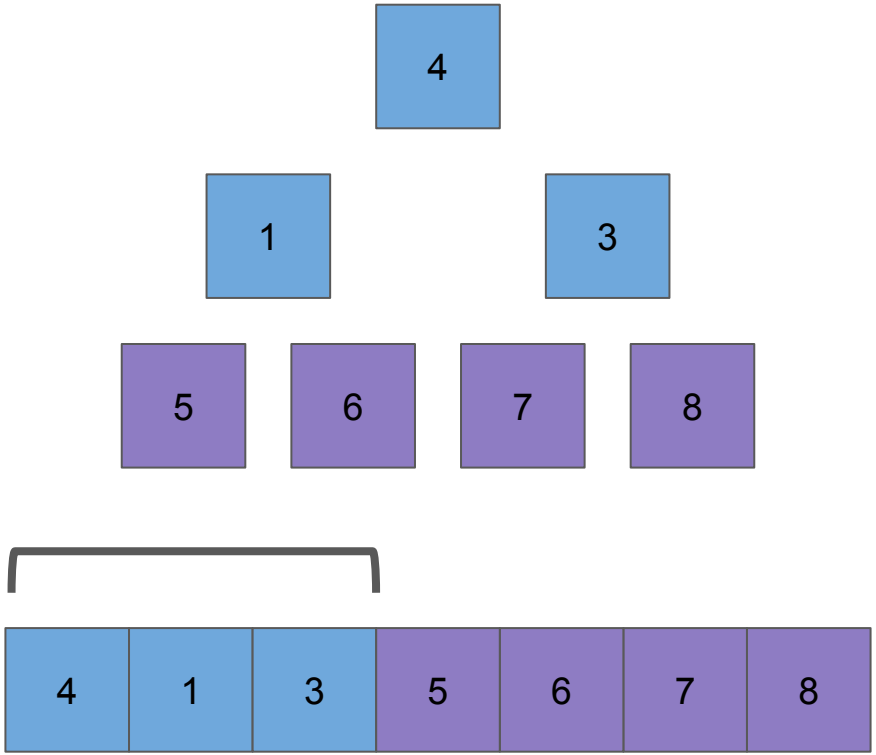
Heapsort



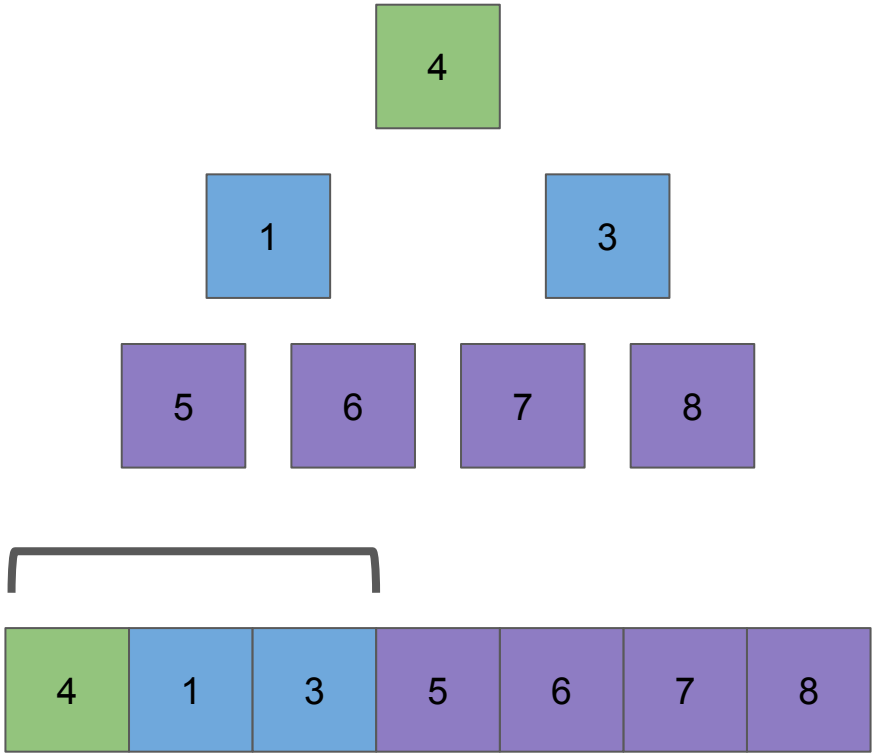
Heapsort



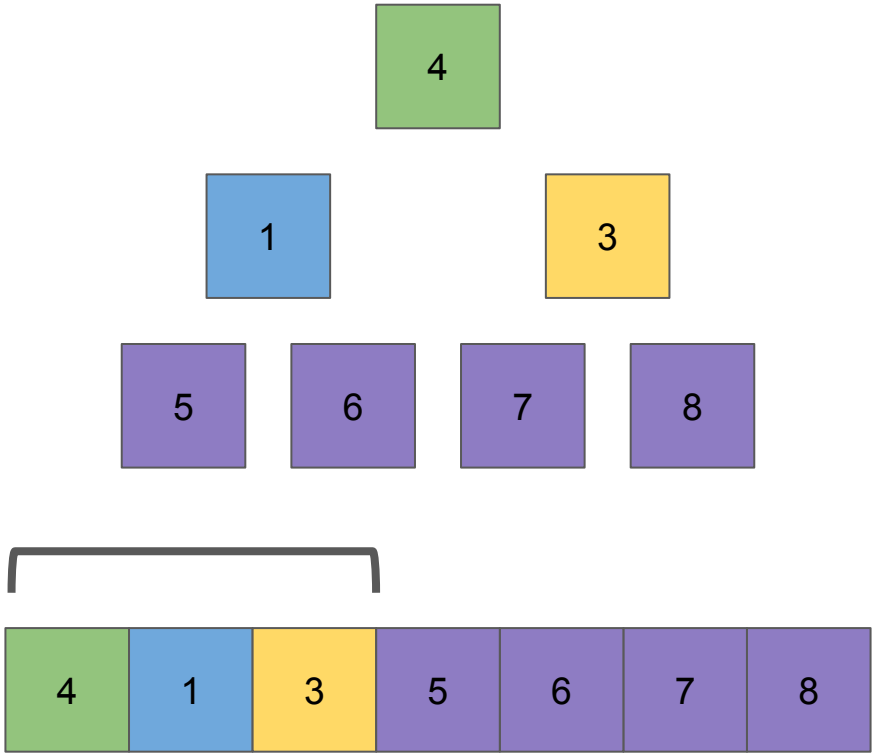
Heapsort



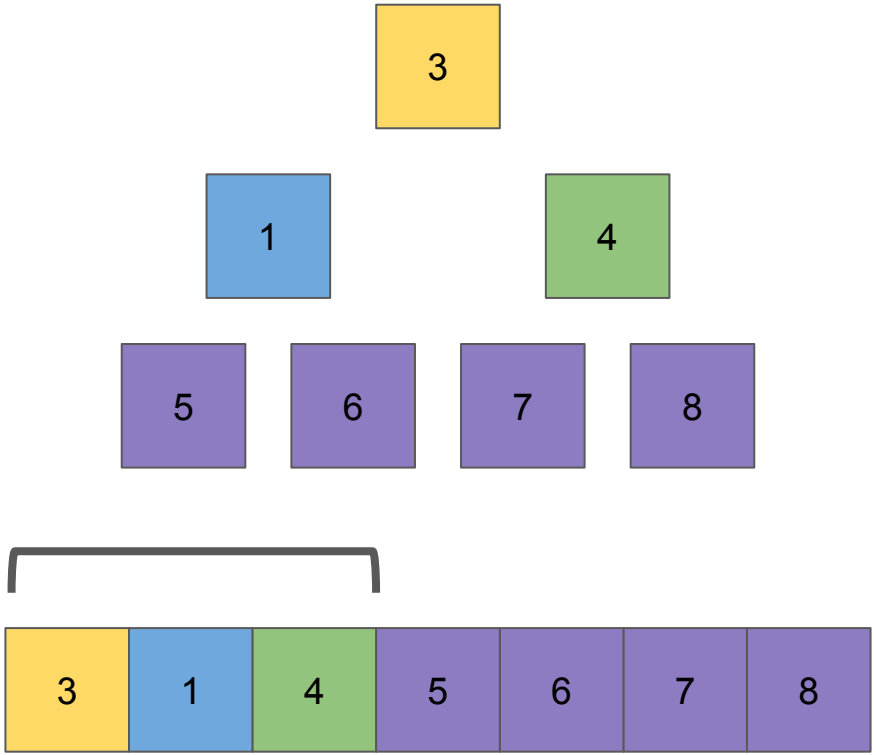
Heapsort



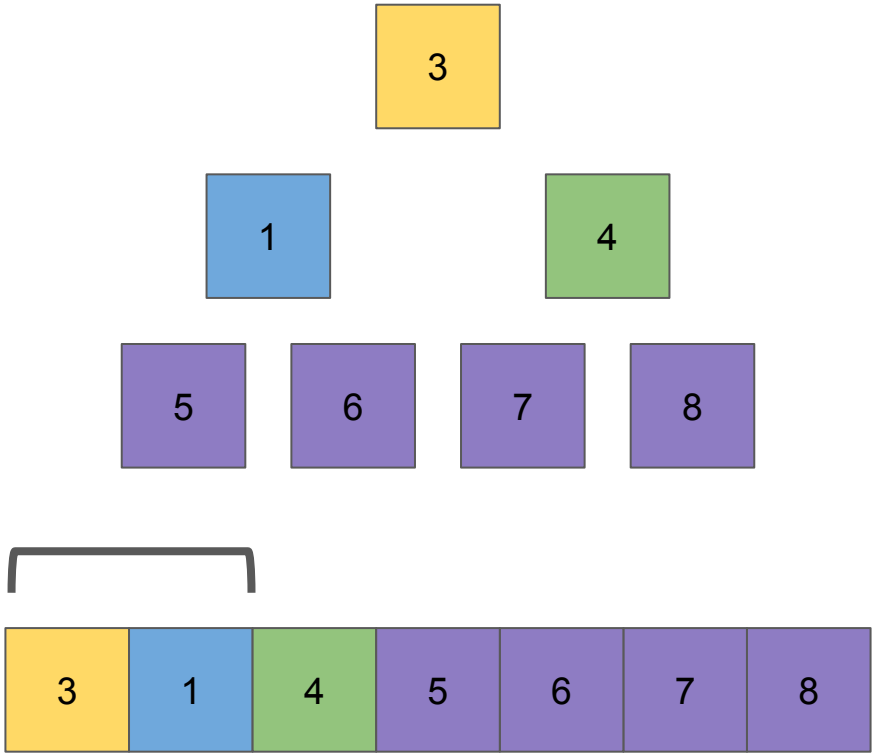
Heapsort



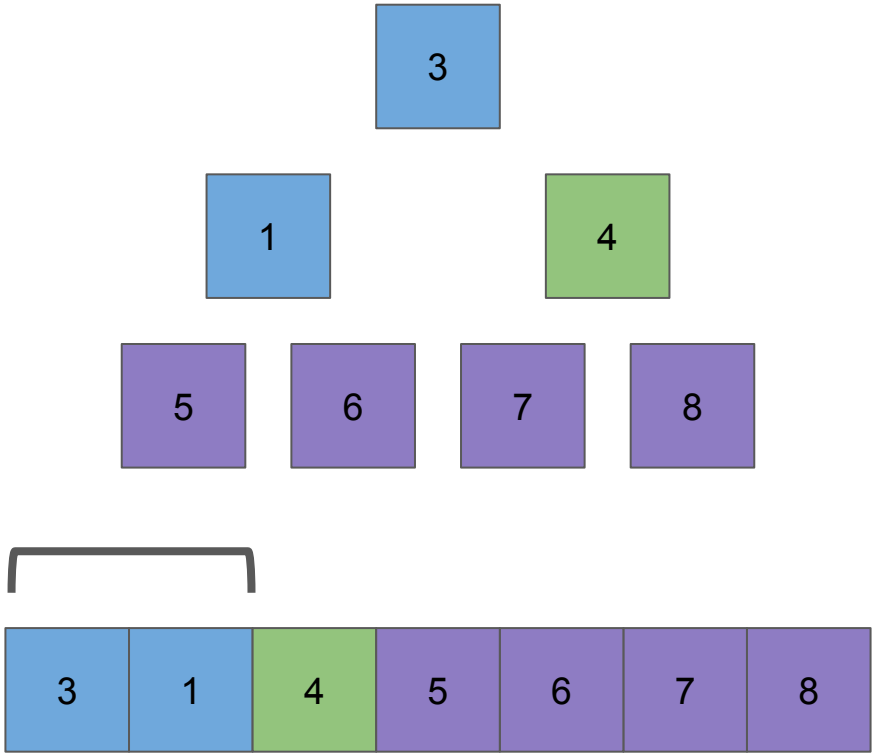
Heapsort



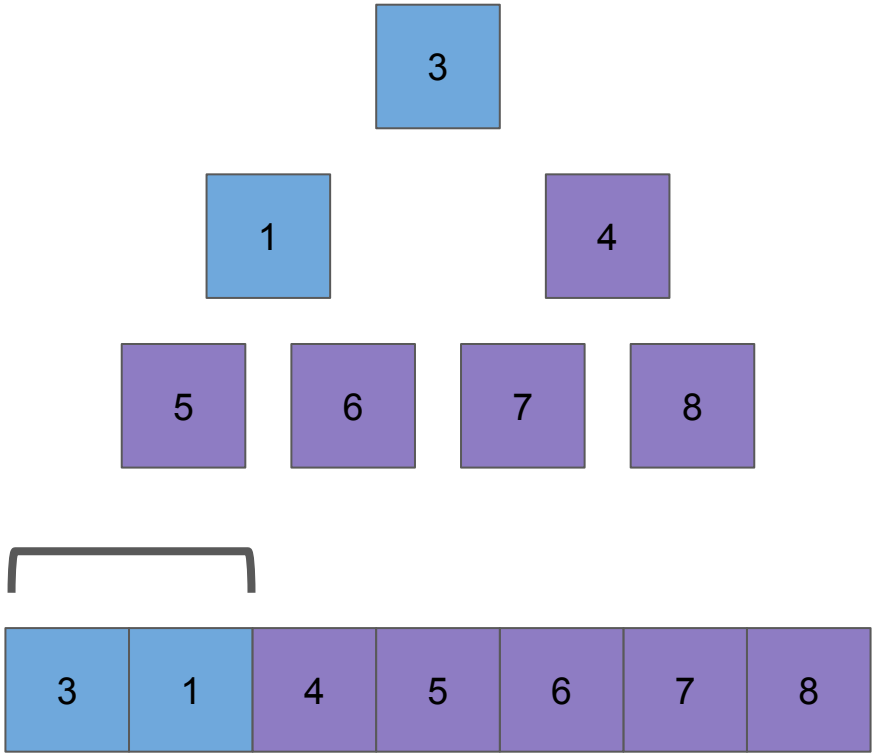
Heapsort



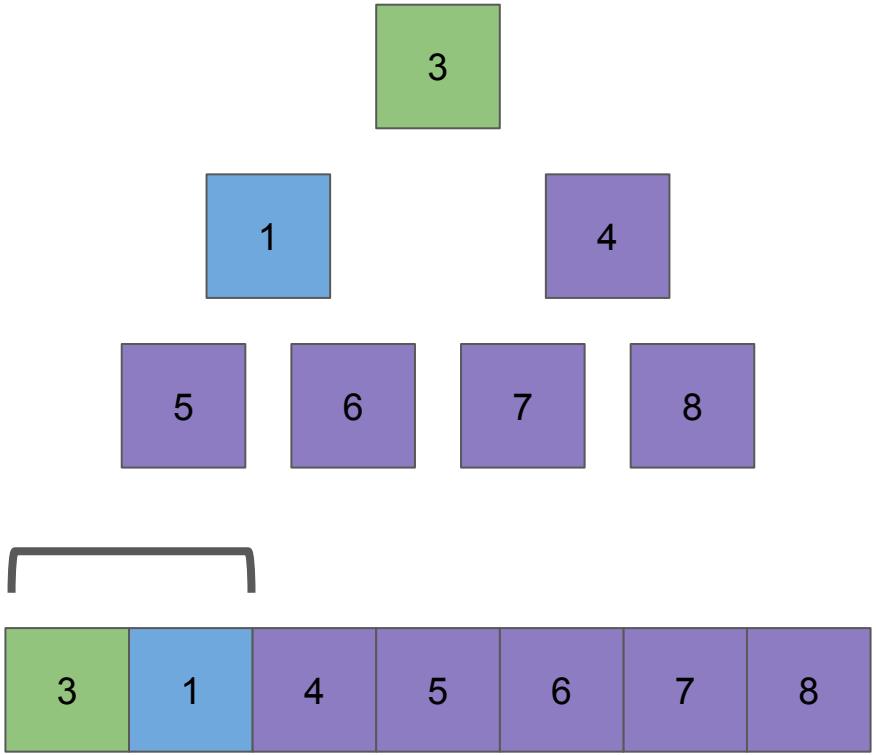
Heapsort



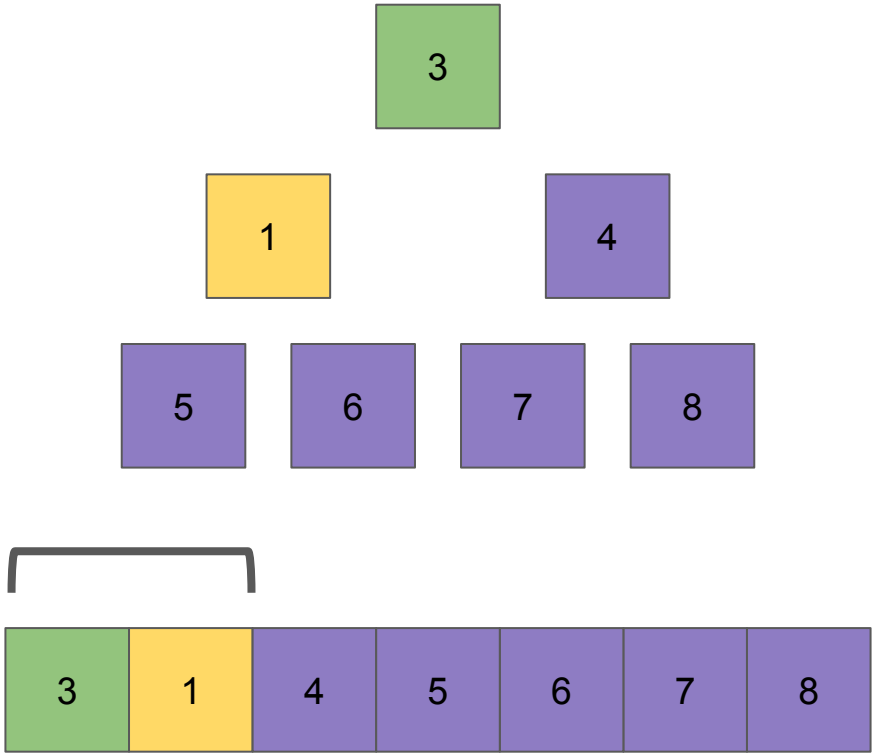
Heapsort



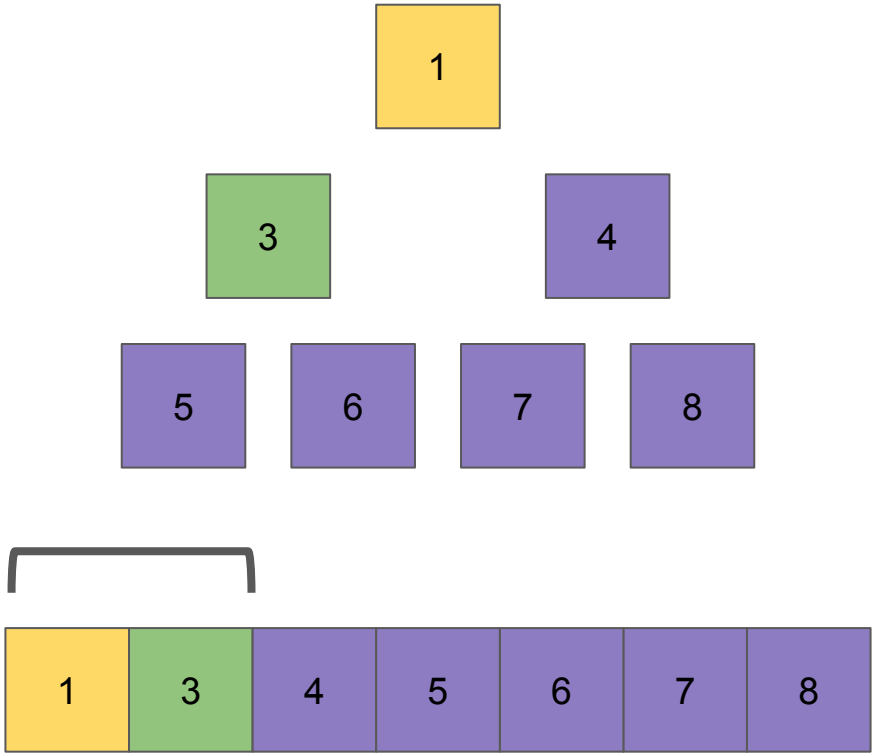
Heapsort



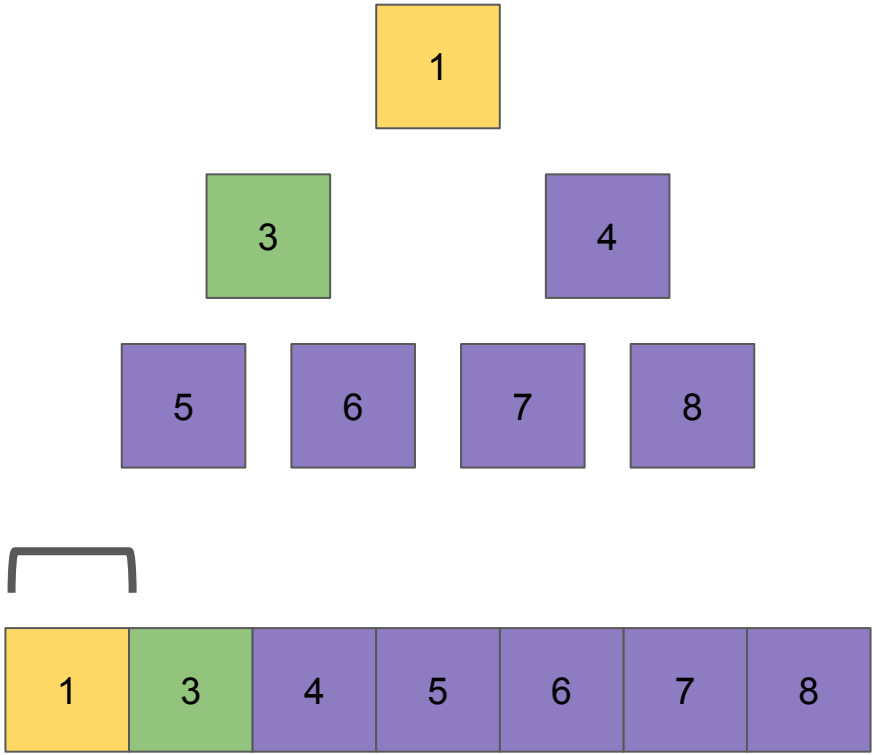
Heapsort



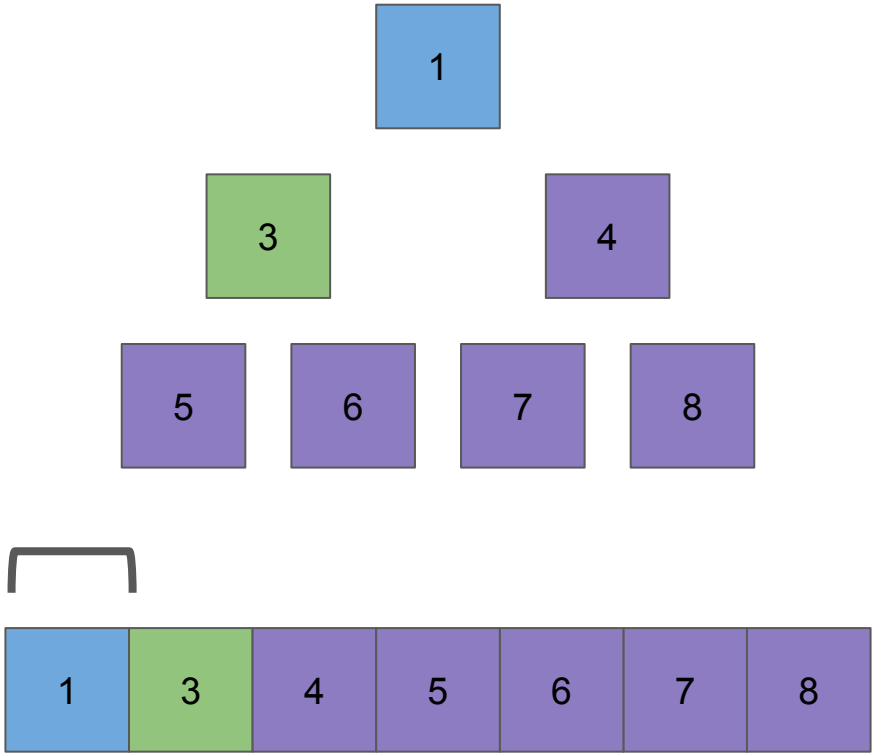
Heapsort



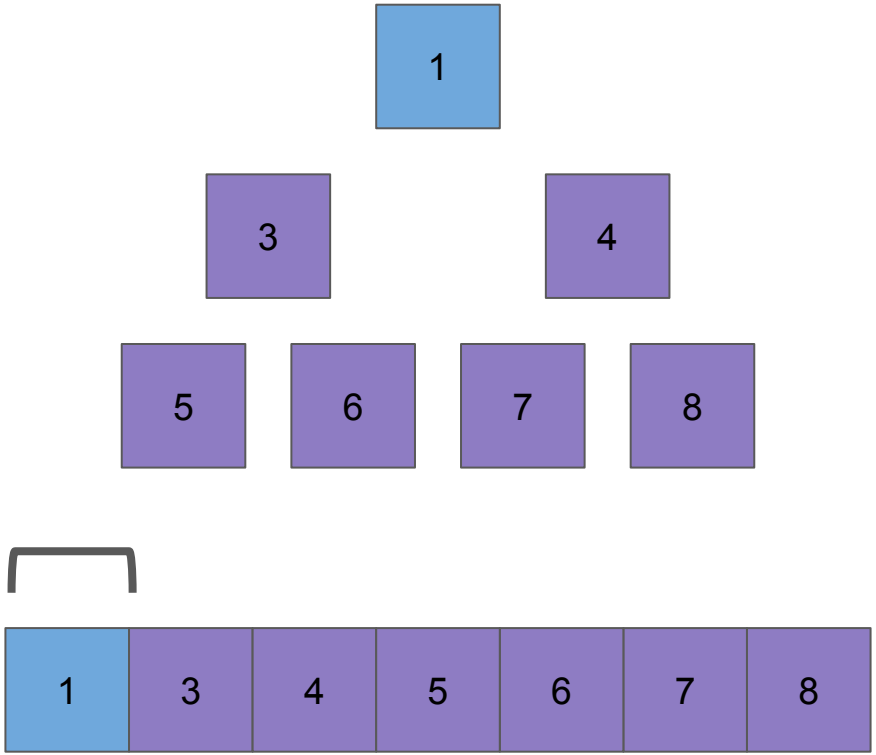
Heapsort



Heapsort



Heapsort



Heapsort