EECS 281 Midterm Review

Fall 2023

Administrative

- Midterm is **Thursday, October 19th** from **7:00 - 9:00 PM**
- Room assignments have been distributed on Piazza!
- No labs this week! Labs resume on 10/23 :)

- **These slides and a link to the recording will be available in Files → Exam - Practice → Exam 1 → F23 Midterm Review Slides, as well as on Piazza (see original announcement which will be updated)**

What Should I Bring?

- You will be allowed a **two-sided** single sheet (8.5” x 11”) of paper to bring into the exam
  - You may include any information that you think will be helpful to you on the exam
  - The note sheet can be handwritten or printed, as long as you put it together yourself
- Also bring *more than one pencil*, a pen (optional), an eraser (optional but highly recommended), and **your MCard**
- Calculators are not allowed (you won’t need one anyway)

Room Assignments (aagankin-loucks)

Room	First Uniqname In This Room	Last Uniqname In This Room
BBB1670	aagankin	alexroz
BBB1690	alexyen	amoghmm
CHRYS133	amulyag	artyom
CHRYS220	aruder	cuprum
COOLG906	cycai	dskalit
CSRB2246	duharry	emmalin
DOW1005	endald	gabehutt
DOW1010	gabigabi	hariskh
DOW1014	harmas	ilanagot
DOW1017	ilanbrei	jerrywa
DOW1018	jerusham	jkzh
DOW2150	jmabr	jtdiaz
DOW2166	jubayrha	kathyzz
DOW3150	kaving	kshinde
EECS1003	ksrikar	lhendrix
EECS1005	lhmnb	loucks

- Room assignments are by UNIQNAME, not first or last name or UMID# or anything else!
- Each room has a set number of copies based on how many people are supposed to be there, so please go to your assigned room!
  - Don’t just go to a room because you prefer the location!
  - If there are too many people in a room we will prioritize those who are actually assigned there and start kicking others out!

Room Assignments (lpinjic-zyuhan)

EECS1008	lpinjic	lukemw
EECS1012	luoz	manng
EECS1200	manvithm	mileslow
EECS1303	mingyliu	netog
EECS1311	netraj	orejevic
EWRE 185	orozcoen	prakharg
EECS3427	pranavjo	raki
EECS3433	rakshasr	rfost
FXB1008	rishabg	sachink
FXB1012	sagbho	shalimma
FXB1032	shangjun	shivac
GFL107	shivgov	sravanid
GGBL1571	srbagri	voronovy
GGBL2147	vpasqua	xuboz
GGBL2153	xuduo	yuyzhang
LBME1130	ywj	zjjing
DOW1206	zniu	zyuhan

- If anything changes, trust Dr. Paoletti, Piazza, and official announcements over what you see here
- COOLG906 is room G906, on the Ground floor of COOLey. Not room 906 in the “COOLG” building…
- Make sure you know where your room is!
  - CSRB, FXB, GFL, and LBME are kind of far from the Pierpont bus stop!
  - EWRE185 and DOW1206 can be hard to find!

Room Assignments (again)

Room	First Uniqname In This Room	Last Uniqname In This Room
BBB1670	aagankin	alexroz
BBB1690	alexyen	amoghmm
CHRYS133	amulyag	artyom
CHRYS220	aruder	cuprum
COOLG906	cycai	dskalit
CSRB2246	duharry	emmalin
DOW1005	endald	gabehutt
DOW1010	gabigabi	hariskh
DOW1014	harmas	ilanagot
DOW1017	ilanbrei	jerrywa
DOW1018	jerusham	jkzh
DOW2150	jmabr	jtdiaz
DOW2166	jubayrha	kathyzz
DOW3150	kaving	kshinde
EECS1003	ksrikar	lhendrix
EECS1005	lhmnb	loucks
EECS1008	lpinjic	lukemw
EECS1012	luoz	manng
EECS1200	manvithm	mileslow
EECS1303	mingyliu	netog
EECS1311	netraj	orejevic
EWRE 185	orozcoen	prakharg
EECS3427	pranavjo	raki
EECS3433	rakshasr	rfost
FXB1008	rishabg	sachink
FXB1012	sagbho	shalimma
FXB1032	shangjun	shivac
GFL107	shivgov	sravanid
GGBL1571	srbagri	voronovy
GGBL2147	vpasqua	xuboz
GGBL2153	xuduo	yuyzhang
LBME1130	ywj	zjjing
DOW1206	zniu	zyuhan

Exam Format

- The exam is designed to be completed in two hours
- You will have two hours to complete the exam
  - This starts from the time everyone in your room is allowed to flip open their exam booklets
  - Do not enter your exam room or open your exam booklet until instructed to do so
- 100 points total
  - 60%: 24 multiple choice questions worth 2.5 points each
  - 40%: 2 free response (coding) questions worth 20 points each
- Question 25 is about using iterators in a way similar to existing STL functions
- Question 26 could be about anything we have learned so far

Exam Logistics/Grading FAQs

Q: Will X be on the exam?

Q: How is line count enforced?

Q: What will the FRQ rubric be like?

Q: When will grades be released?

Q: Is the exam curved?

Q: *your question here*

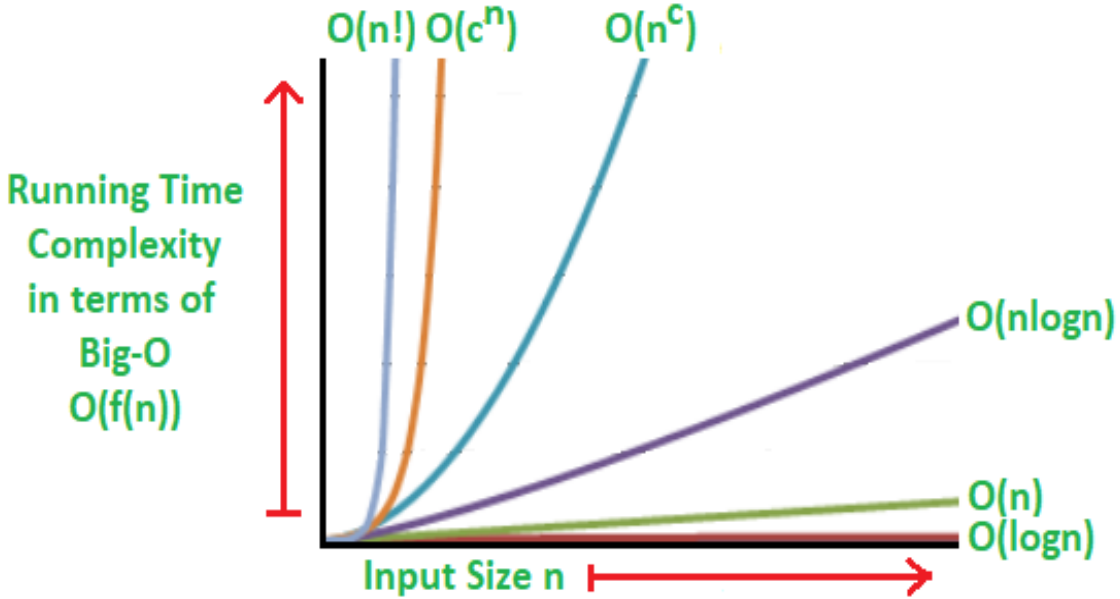
General Review Strategies

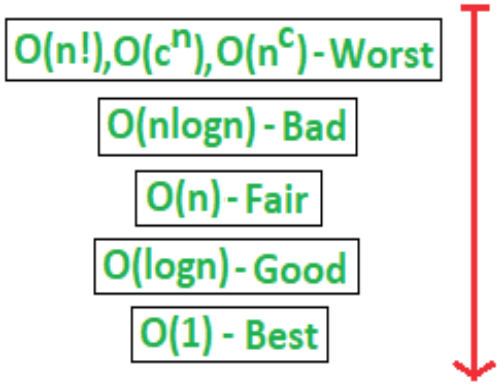
- Lecture recordings
- Lecture / lab slides
- Practice exam questions
- Lab quizzes (PDFs)
- Project algorithms
- Writing your note sheet
- Class notes by Andrew Zhou
  - https://ajzhou.gitlab.io/eecs281/notes/combined/
  - Must know: chapters 4-15
    (except 10.5 Pairing Heaps, 14.7 Mergesort, and 14.9-14.11)
  - Be comfortable with concepts in chapters 1-3

Midterm Content

- (Asymptotic) Complexity and Runtime Analysis, Math Foundations
- Recursion and the Master Theorem
- Container Data Structures and Array-Based Containers
- The Standard Template Library (STL)
- Stack, Queue, and Priority Queue ADTs
- Ordered Arrays and Related Algorithms
- Set Operations (Union/Intersection) and Union-Find
- Elementary Sorts and Library-Implemented Sorts
- Quicksort, but **NOT MERGESORT**
- Heaps and Heapsort

Useful Graphics





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

Useful Graphics

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

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

$$T(1) = 1$$

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

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

Useful Graphics

	Input	Output	Forward	Bidirectional	Random
Supports dereference ($\star$) and read	✓		✓	✓	✓
Supports dereference ($\star$) and write		✓	✓	✓	✓
Supports forward movement (++)	✓	✓	✓	✓	✓
Supports backward movement (−−)				✓	✓
Supports multiple passes			✓	✓	✓
Supports == and !=	✓		✓	✓	✓
Supports pointer arithmetic (+, −, etc.)					✓
Supports pointer comparison (<, >, etc.)					✓

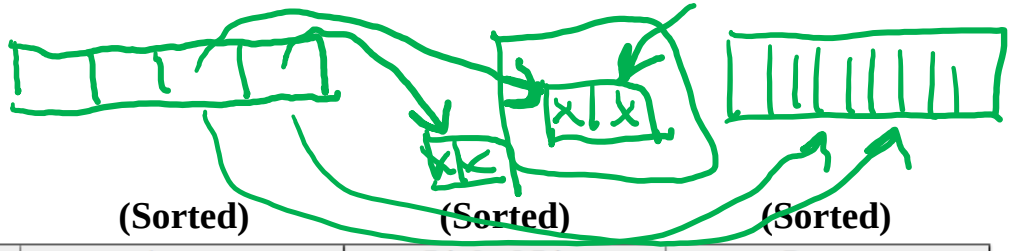
Useful Graphics

Container Type	Iterator Category
std::vector<>	Random Access
std::deque<>	Random Access
std::string<>	Random Access
std::list<>	Bidirectional
std::set<>	Bidirectional
std::multiset<>	Bidirectional
std::map<>	Bidirectional
std::multimap<>	Bidirectional
std::unordered_set<>	Forward
std::unordered_multiset<>	Forward
std::unordered_map<>	Forward
std::unordered_multimap<>	Forward
std::forward_list<>	Forward

Useful Graphics

Operation	Array	Linked List	Deque
`add_value(val)` *(inserts value anywhere in container)*	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`remove(val)` *(remove value from container, but you must find it first)*	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
`remove(iterator)` *(remove value from container, but you are given its iterator)*	$\Theta(n)$	$\Theta(n)$ if singly-linked $\Theta(1)$ if doubly-linked	$\Theta(n)$
`find(value)` *(find a value in the container)*	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
`iterator::operator*()` *(dereference an iterator)*	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`operator[](index)` *(access element with a position at index)*	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
`insert_after(iterator, val)` *(insert an element after a provided iterator)*	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
`insert_before(iterator, val)` *(insert an element before a provided iterator)*	$\Theta(n)$	$\Theta(n)$ if singly-linked $\Theta(1)$ if doubly-linked	$\Theta(n)$

Useful Graphics



Operation	Array	Linked List	Deque
`add_value(val)` *(inserts value anywhere in container)*	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
`remove(val)` *(remove value from container, but you must find it first)*	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
`remove(iterator)` *(remove value from container, but you are given its iterator)*	$\Theta(n)$	$\Theta(n)$ if singly-linked $\Theta(1)$ if doubly-linked	$\Theta(n)$
`find(value)` *(find a value in the container)*	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(\log(n))$
`iterator::operator*()` *(dereference an iterator)*	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`operator[](index)` *(access element with a position at index)*	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
`insert_after(iterator, val)` *(insert an element after a provided iterator)*	N/A	N/A	N/A
`insert_before(iterator, val)` *(insert an element before a provided iterator)*	N/A	N/A	N/A

Useful Graphics

Sort	Best	Average	Worst	Memory	Stable?	Adaptive?
Bubble	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	O(1)	Yes	Yes
Selection	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	O(1)	No	No (not without extra memory)
Insertion	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	O(1)	Yes	Yes
Heap	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	O(1)	No	No
Merge	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	O(n)	Yes (if merge is stable)	No
Quick	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	O(log n)	No (unless partition is stable)	No

Content Review

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

Average Case vs. Amortized Complexity

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

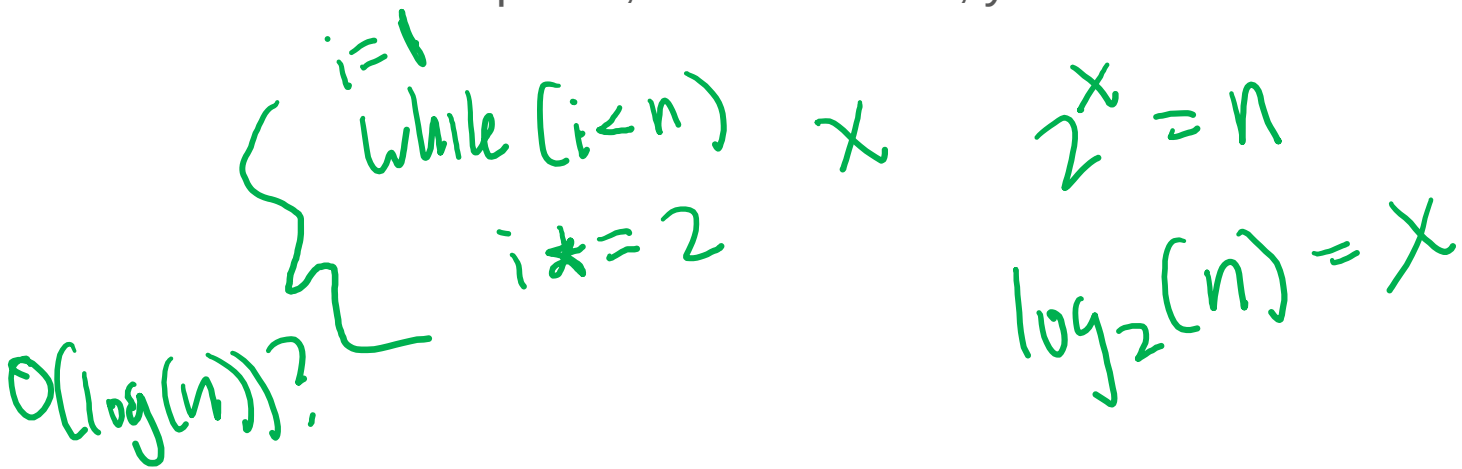
- Recall the concept of expected value from 203, think of average case as the kind of case that you would most likely get
- Ex: `std::sort` is average-case O(n log n) time

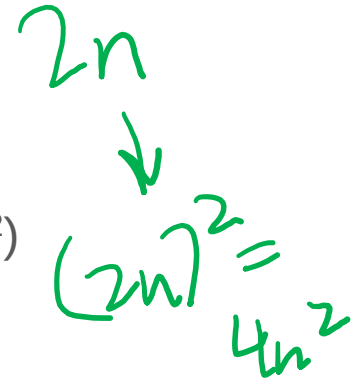
Amortized complexity refers to the average cost of the function over multiple operations

- 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

Complexity Analysis 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





Data Structures/Containers You Should Know

- Array (fixed size and dynamically resizing containers)
  - `std::string` works like `vector<char>` in many cases (plus more!)
- Linked list
- Stack
- Queue
- Double-ended queue (deque) **NOT dequeue, that’s a verb
- Priority queue (interface, stl wrapper)
- Binary heap (common PQ implementation, tree and array representations)
- Sorted vector as a set
- Disjoint set/union find

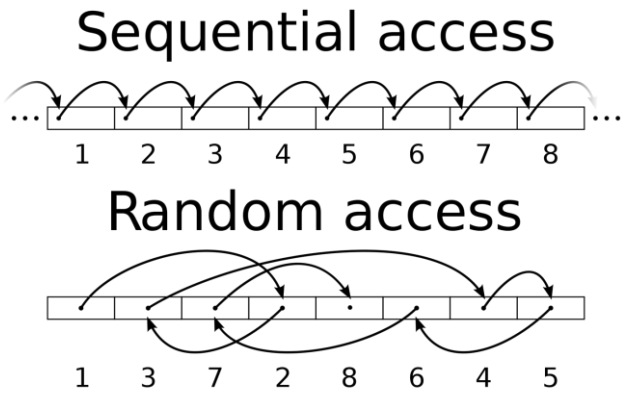
Container Access Order

Sequential Access

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

Random Access

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



Standard Template Library Containers

Container	Purpose
`std::vector`	Dynamically resizing array
`std::list`	Doubly-linked list
`std::forward_list`	Singly-linked list (more memory efficient)
`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 Base Container Use Cases

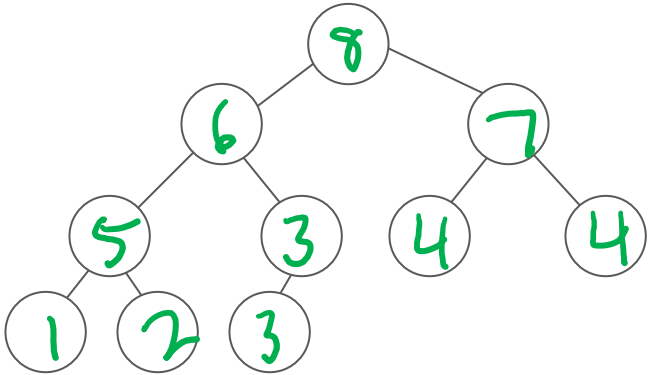
- `std::vector` – very compact in memory, very cache-local, amortized constant `push/pop_back`, random access
  - Random `insert/erase` requires linear copying!
  - Elements are stored contiguously in memory.
- `std::deque` – fairly compact in memory, fairly cache-local, amortized constant `push_back/front` & `pop_back/front`, random access
  - Good choice if you need to `push/pop` at the front and back
  - Also will never invalidate pointers through normal use
- `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.

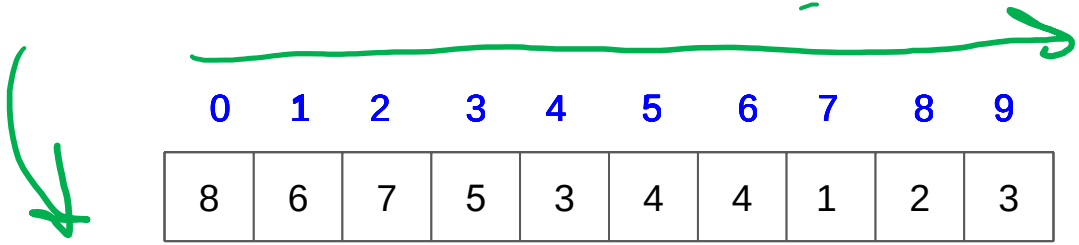
STL Adaptor Container Use Cases

- `std::stack` – constant `push()` and `pop()` using back end of `std::deque` or `std::vector`, only access to `top()`
  - “Depth-first” searching, accessing most recent items
- `std::queue` – constant `push()` and `pop()` using front and back ends of `std::deque`, only access to `front()`
  - “Breadth-first” searching, any sort of line (first come → first served)
- `std::priority_queue` – log(n) `push()` and `pop()` using binary heap stored in `std::vector` or `std::deque`, constant access to `top()`
  - Takes a comparator (default `std::less`) to determine priority
  - Can be an efficient way to do things if comparisons are required

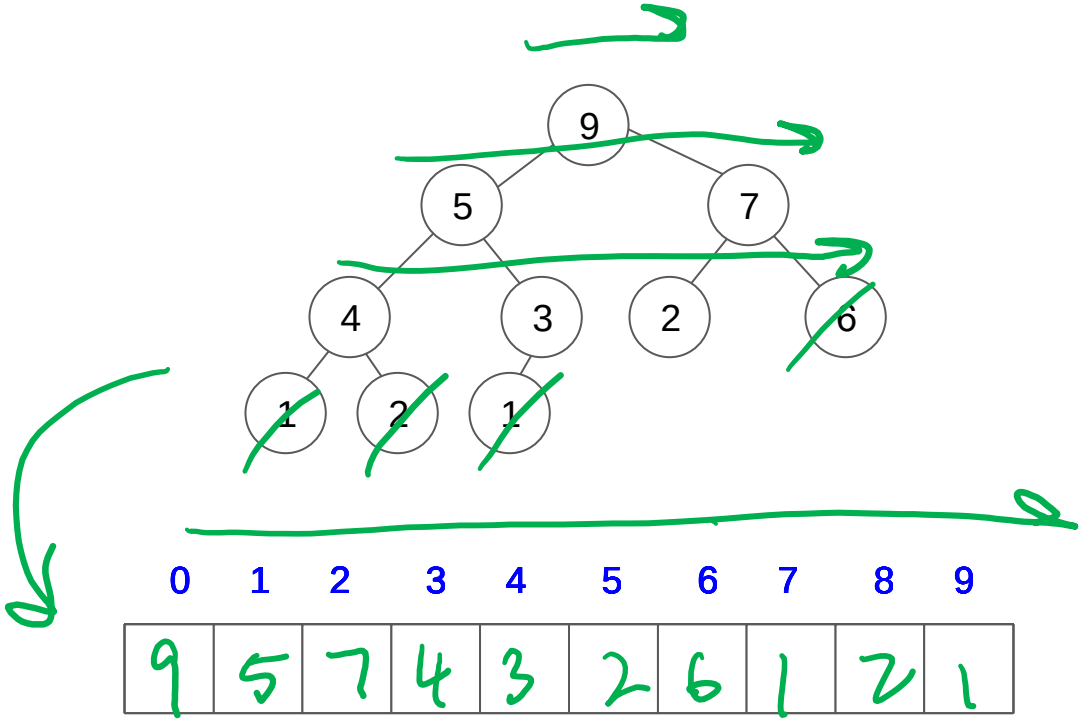
```
std::priority_queue<int> myPQ1(vec.begin(), vec.end());
std::priority_queue<double, std::deque<double>, std::greater<double>>
    myPQ2;
```

Representing Binary Heaps





Representing Binary Heaps



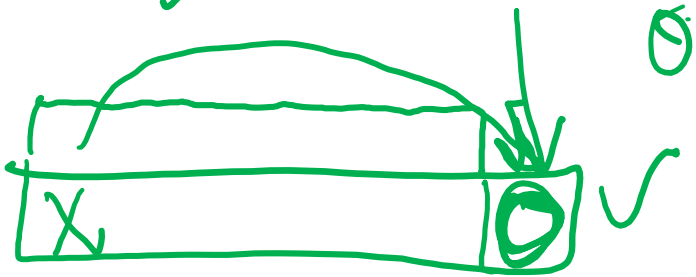
Heapsort

least → greatest

MAX heap

- Algorithm:
  - Heapify the range using the same comparator — O(n)
  - Swap the top of the heap with the end
  - Largest element is now in correct position, so ignore it → O(1)
  - Fix down the new element from the top of the heap — O(log(n))
  - Repeat swapping and fixing until range is sorted

n times



O(n + n·log n) → O(n log n)

Quicksort

- Algorithm:
  - Pick a pivot from the list
    (you can assume the last element unless specified otherwise)
  - Elements less than pivot swapped closer to front
  - Elements greater than or equal to pivot swapped closer to back
  - After partitioning, the pivot is swapped to the final position
  - Recursively sort the sub-list of elements with smaller values and the sub-list of elements with greater values.

Practice FRQs

Practice Midterm Question 25: `std::unique_copy`

Implement STL's `unique_copy()` function according to its official interface and description.

```
template<class ForwardIterator, class OutputIterator>
OutputIterator unique_copy(ForwardIterator first, ForwardIterator last,
                                                  OutputIterator result);
```

Quoting from https://en.cppreference.com/w/cpp/algorithm/unique_copy:

> "Copies the elements from the range `[first, last)`, to another range beginning at `result` in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied."

Complexity: Linear.
For elements in the ranges, exactly `last - first` applications of `operator==()` and at most `last - first` assignments.

Practice Midterm Question 25: `std::unique_copy`

For example, if you executed this code:

```
int data[6] = {1, 3, 3, 1, 1, 0}, output[6];
unique_copy(data, data + 6, output);
```

You would have this in array output:

1	3	1	0		

Implementation: Use the back of this page as a working area, then rewrite **neatly** on the front.
Limit: 15 lines of code (points deducted if longer).
You may **NOT** use other STL algorithms/functions.

std::unique_copy(...);

Practice Midterm Question 25: `std::unique_copy`

```
for(auto it = first; it != last; ++it) {

    *output = *it;

}
```

Practice Midterm Question 25: `std::unique_copy`

```
auto prev = first;   <- // empty check
  ++ first;
*result = *prev;

// loop
    : if(!(*first == *prev))
```

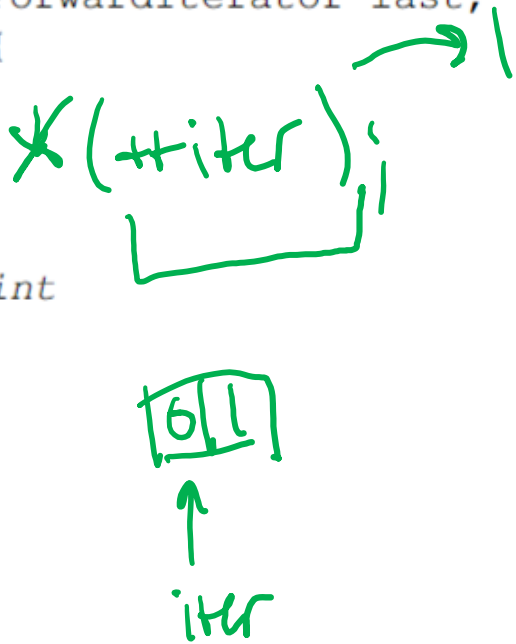
Practice Midterm Question 25: `std::unique_copy`

```
template <class ForwardIterator, class OutputIterator>
OutputIterator unique_copy(ForwardIterator first, ForwardIterator last,
                           OutputIterator result) {

    if (first == last)
        return result;

    *result++ = *first; // establish a starting point
    ForwardIterator prev = first++;

    while (first != last) {
        if (!(*first == *prev)) // *first != *prev
            *result++ = *first;
        prev = first++;
    } // while
    return result;
}
```



Practice Midterm Question 26: Finding Max Elements

Suppose you are given an array of `int` of size `n`, and a number `k`.
Write a function that will return a `std::vector<int>` containing the *k* largest Elements. The output does not need to be sorted. You can assume that $k < n$.

Requirements: Your solution **must be faster than** *O(n log n)* time.
You may use up to *O(n)* auxiliary space.

Implementation: Use the back of this page as a working area, then rewrite **neatly** on the front. Limit: 15 lines of code (points deducted if longer).
You may use any C, C++, or STL function/algorithm that you wish.

Practice Midterm Question 26: Finding Max Elements

Practice Midterm Question 26: Finding Max Elements

Practice Midterm Question 26: Finding Max Elements

Solution 1:

```
vector<int> findKMax(int arr[], size_t n, size_t k) {
    // O(n + k log n) solution: Create a max-PQ of everything,
    // which is O(n).  Then pop off the k largest items:
    // k operations, each of which is O(log n)
    priority_queue<int> myPQ(arr, arr + n);
    vector<int> output;
    output.reserve(k);
    for (size_t i = 0; i < k; ++i) {
        output.push_back(myPQ.top());
        myPQ.pop();
    } // for
    return output;
} // findKMax()
```

Practice Midterm Question 26: Finding Max Elements

Solution 2:

```
// O(k + (n-k) log k) solution: create a min-PQ of the first k items, in O(k).
// Then n-k times, push one item and pop off the SMALLEST item, leaving
// the k largest remaining: (n-k) * O(log k), since the PQ is of size k.
vector<int> findKMax2(int arr[], size_t n, size_t k) {
    // If it was k smallest, would be simpler declaration
    priority_queue<int, vector<int>, std::greater<int>> myPQ(arr, arr + k);
    vector<int> output;
    output.reserve(k);
    for (size_t i = k; i < n; ++i) {
        myPQ.push(arr[i]);
        myPQ.pop();
    } // for
    while (!myPQ.empty()) {
        output.push_back(myPQ.top());
        myPQ.pop();
    } // while
    return output;
} // findKMax2()
```

Additional Practice Question 18: `std::minmax_element`

(AKA STL Practice Question 3)

Implement STL's `minmax_element()` function according to its official interface and description.

```
template <class ForwardIterator, class Compare>
pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first,
                                                      ForwardIterator last, Compare
                                                  comp);
```

The `minmax_element()` function returns a pair with an iterator pointing to the element with the smallest value in the range `[first, last)` as the `first` element, and an iterator pointing to the largest value in the range as the `second` element. The comparisons are performed using the `comp` comparator. If more than one equivalent element has the smallest value, the first iterator points to the *first* of such elements. If more than one equivalent element has the largest value, the second iterator points to the *last* of such elements.

Additional Practice Question 18: `std::minmax_element`

For example, given the vector `vec = [3, 7, 6, 9, 5, 8, 2, 4]`, running the function with `vec.begin()` as the first argument, `vec.end()` as the second argument, and `operator<` as the comparator, a pair would be returned with an iterator to 2 as the first element and an iterator to 9 as the second element.

You may **NOT** use any STL algorithms/functions. The program must run in linear time, and you are limited to 15 lines of code.

Additional Practice Question 18: `std::minmax_element`

Additional Practice Question 18: `std::minmax_element`

Additional Practice Question 18: `std::minmax_element`

Solution:

```
template <class ForwardIterator, class Compare>
pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first,
                                                      ForwardIterator last, Compare comp) {
  ForwardIterator min = first, max = first;
  for (; first != last; ++first) {
    if (comp(*first, *min)) {
      min = first;
    }
    if (!comp(*first, *max)) {
      max = first;
    }
  }
  return {min, max};
}
```

Winter 2020 Question 27: Sorting Student IDs

You work at the university registrar, and you have a vector of *n* student ID numbers
that are currently in use at the university. To make assigning new student IDs easier, you always keep your vector of IDs in sorted order. However, a mischievous
EECS 281 student hacked into the system last night and shuffled the order of student IDs in your vector!

Fortunately, the hacker was nice enough to tell you that every ID number in the shuffled vector is **at most** *d* positions away from its correct sorted position, where *d* is an integer in the range [1, *n*). Your goal is to implement a function that can restore the sorted vector of student IDs, given the value of *d*.

Winter 2020 Question 27: Sorting Student IDs

Complexity: *O(n log(d))* time and *O(d)* auxiliary space.

Implementation: You may use anything from the STL. Limit: 20 lines of code (points deducted if longer).

Example: Given d = 2 and the following altered vector:

```
ids = [2, 1, 3, 5, 7, 4, 6]
```

the `restore_sorted_IDs()` function should restore ids so that its elements are in sorted order:

```
ids = [1, 2, 3, 4, 5, 6, 7]
```

Continued from next page

Winter 2020 Question 27: Sorting Student IDs

```
int new_index = ids.size() - d - 1;
while (!pq.empty()) {
    ids[new_index++] = pq.top();
    pq.pop();
}
}
```

O(1)

O(d+1)

log d

O((d+1)(log d))

Continued on previous page

Winter 2020 Question 27: Sorting Student IDs

```
void restore_sorted_IDs(vector<int> &ids, size_t d) {
    priority_queue<int, vector<int>, greater<int>> pq(ids.begin(),
                                                      ids.begin()+d+1);
    for(int i = 0; i < ids.size()-d-1; i++) {
        ids[i] = pq.top();          // O(1)
        pq.pop();                   // O(log(d))
        pq.push(ids[i+d+1]);        // O(log(d))
    }
}
```

O(d)

O(n-d-1)

O((n-d-1)(log d))

Winter 2020 Question 27: Sorting Student IDs

Solution 1:

```
void restore_sorted_IDs(vector<int> &ids, size_t d) {
  priority_queue<int, vector<int>, greater<int>>
      pq(ids.begin(), ids.begin() + d + 1);
  int curr_idx = 0;
  for (size_t i = d + 1; i < ids.size(); ++i) {
    ids[curr_idx++] = pq.top();
    pq.pop();
    pq.push(ids[i]);
  }
  while (!pq.empty()) {
    ids[curr_idx++] = pq.top();
    pq.pop();
  }
}
```

Winter 2020 Question 27: Sorting Student IDs

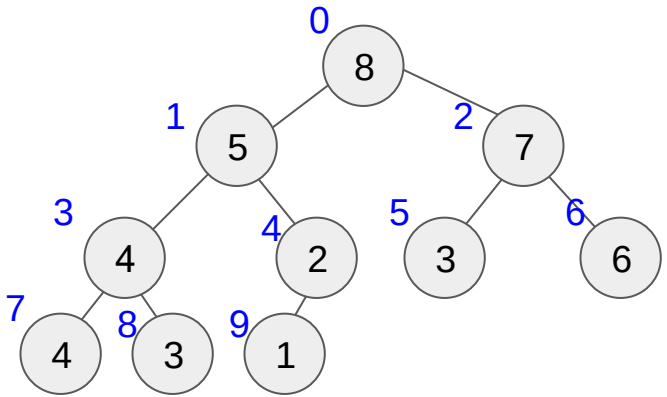
Solution 2:

```
void restore_sorted_IDs(vector<int> &ids, size_t d) {
  auto front = ids.begin();
  auto back = ids.begin() + 2 * d;
  if (d * 2 < ids.size()) {
    for (int i = 0; i < (ids.size() / d) - 1; ++i) {
      sort(front, back);
      front += d;
      back += d;
    }
  }
  sort(front, ids.end());
}
```

Additional Walkthroughs

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**

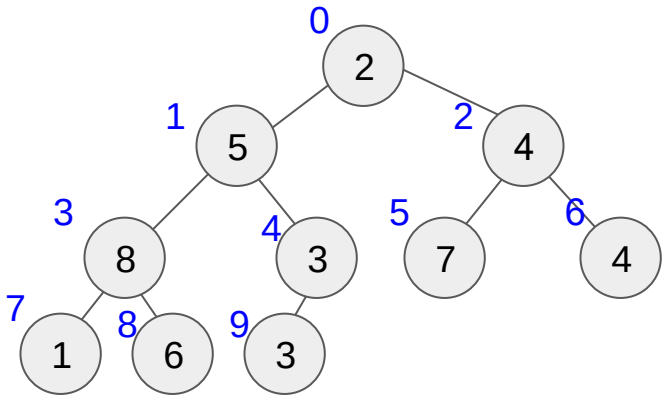
The parent of node stored at location **i** is given as **(i-1)/2**

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

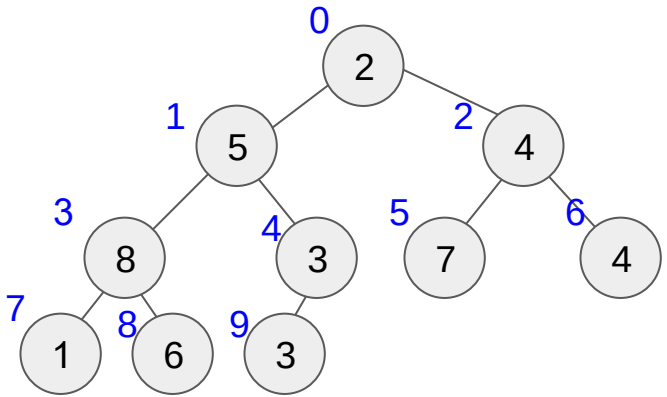


First the array is all messed up - we need to make a heap out of the array.

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

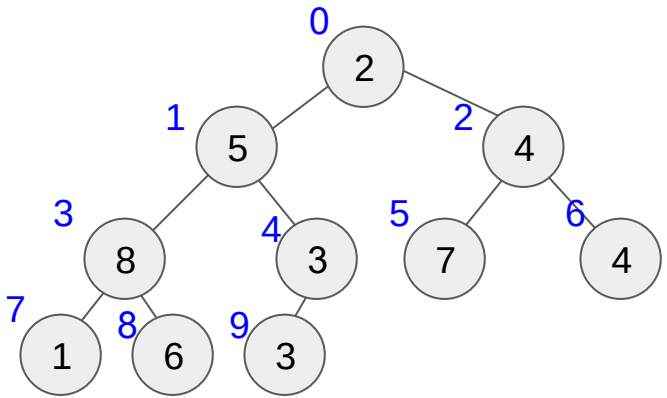
1. Fix up - Top to bottom level wise
2. Fix down - Bottom to top, level wise

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

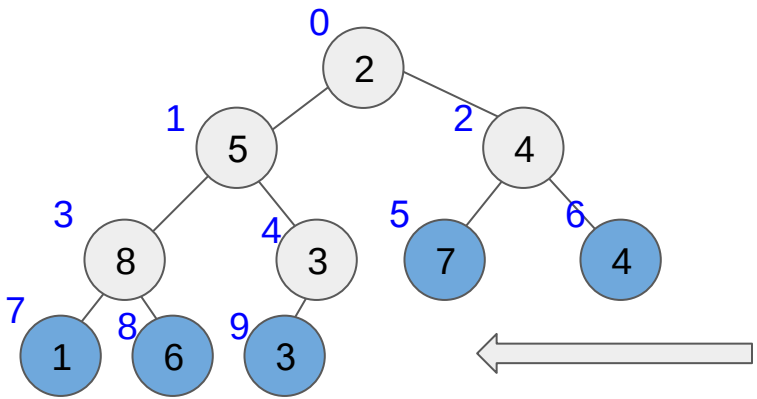
1. Fix up - Top to bottom level wise O(nlogn)
2. Fix down - Bottom to top, level wise O(n)

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)

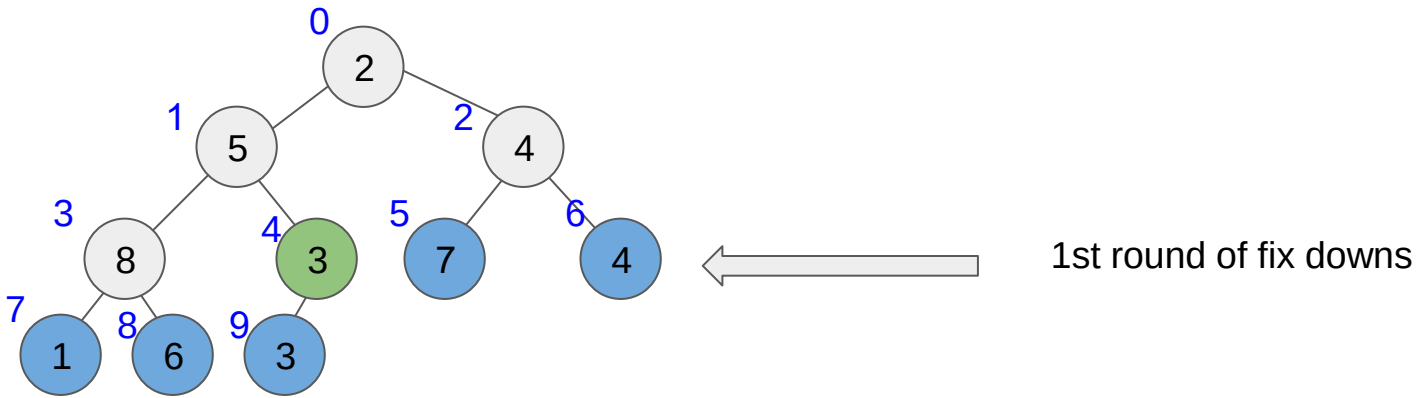
Which one to choose?

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

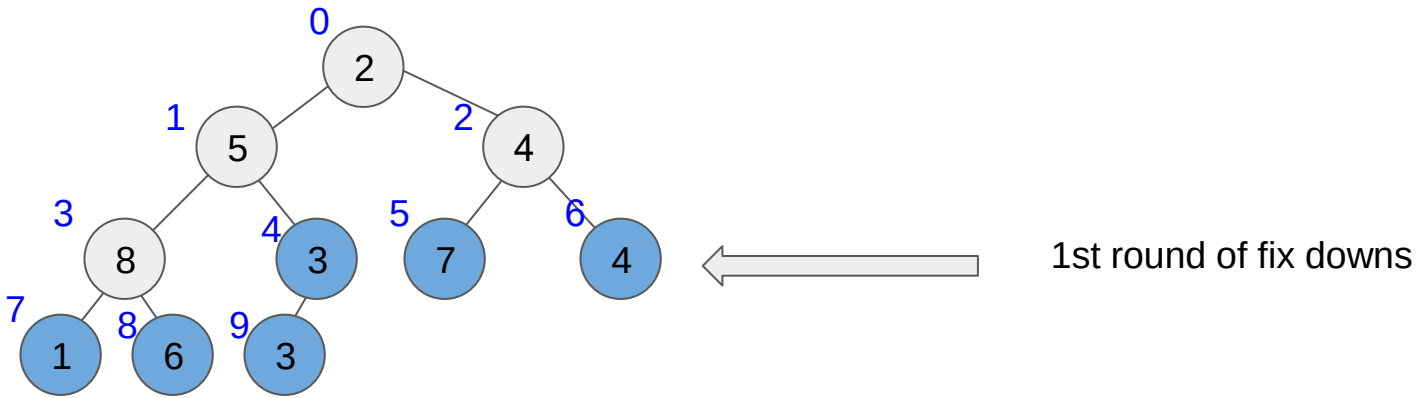


1st round of fix downs

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

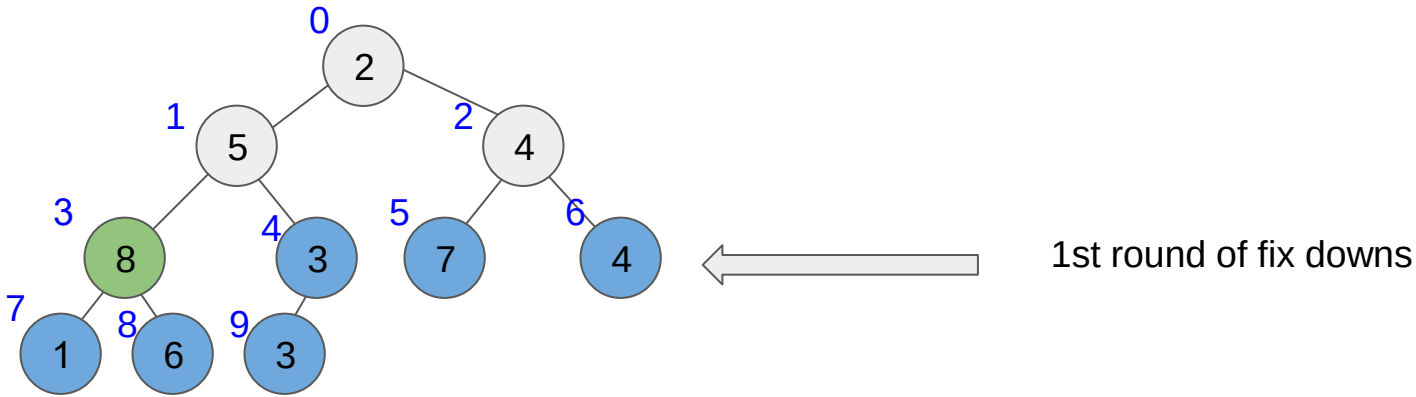


1st round of fix downs

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

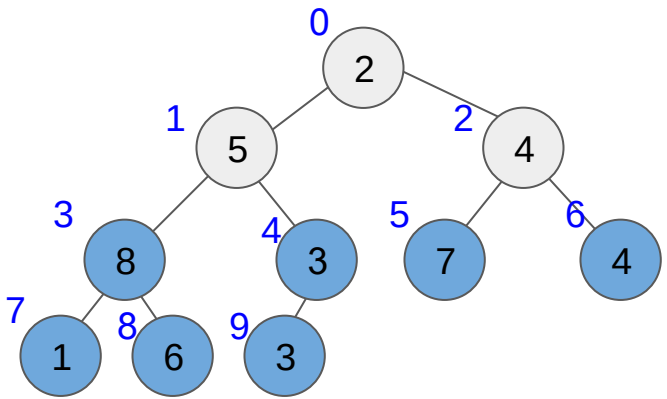


1st round of fix downs

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

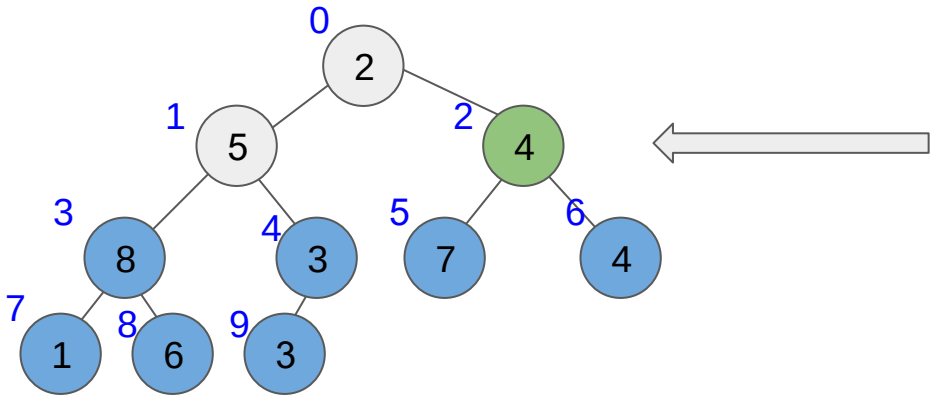


1st round of fix downs

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

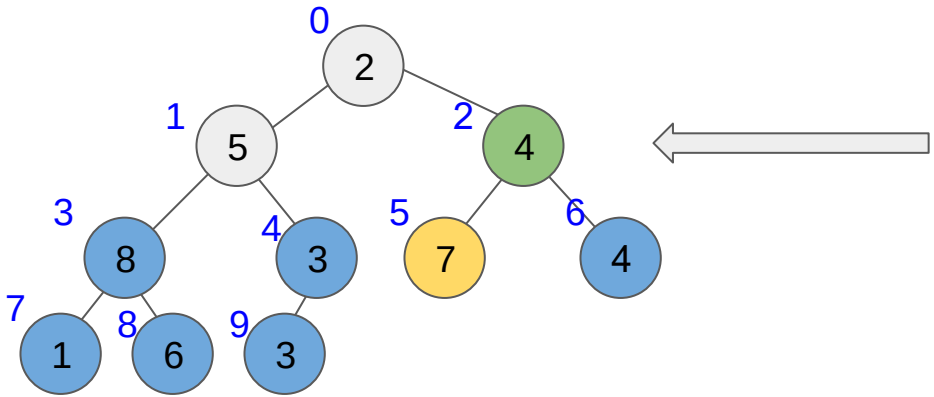


2nd round of fix downs

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

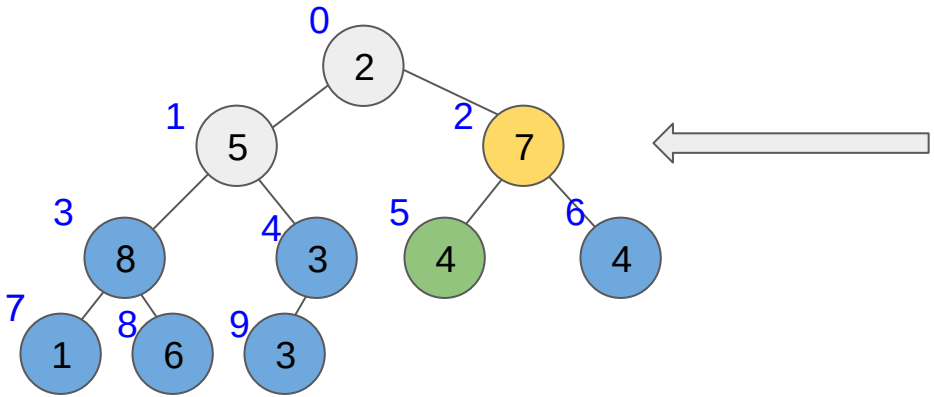


2nd round of fix downs

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

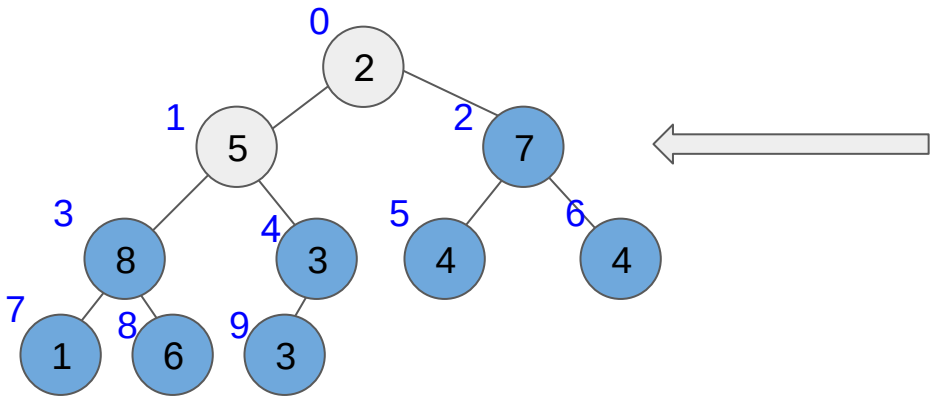


2nd round of fix downs

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

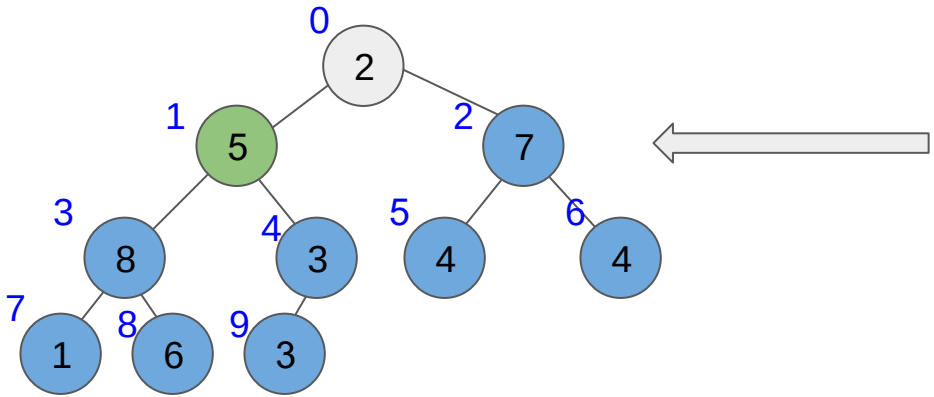


2nd round of fix downs

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

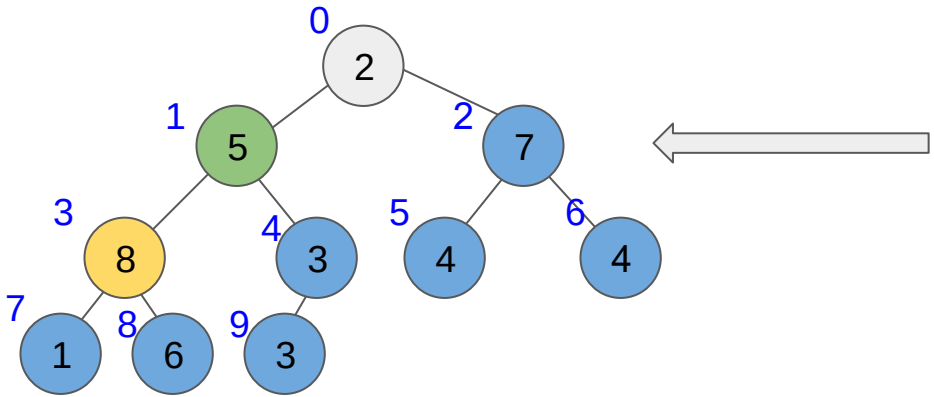


2nd round of fix downs

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

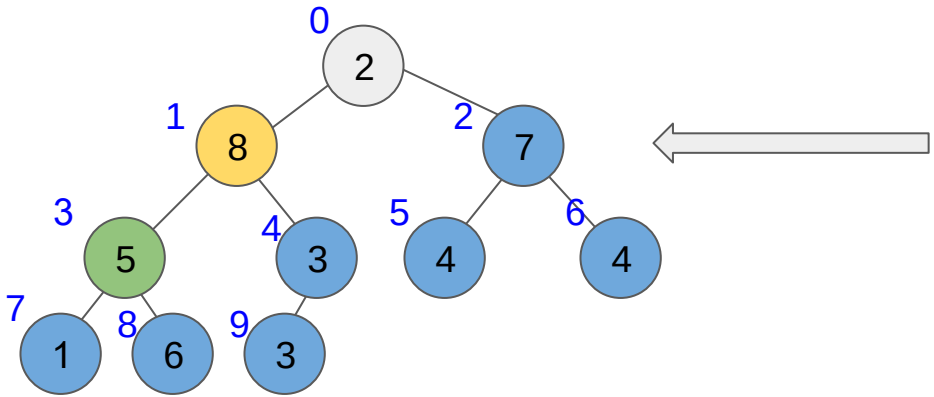


2nd round of fix downs

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

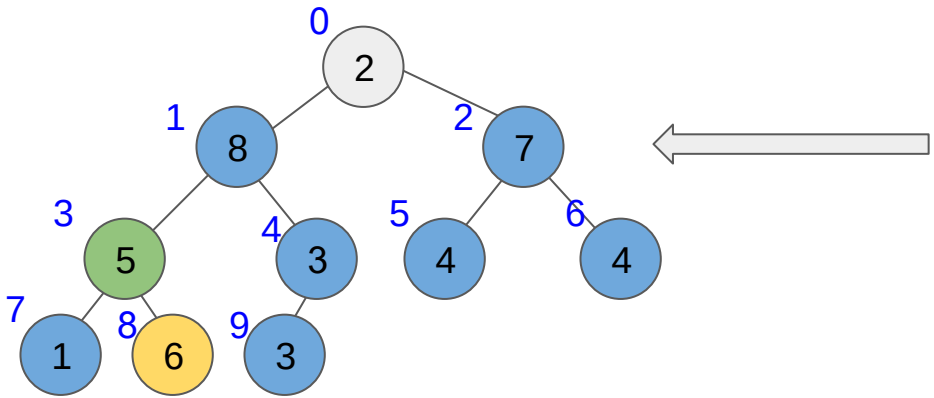


2nd round of fix downs

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

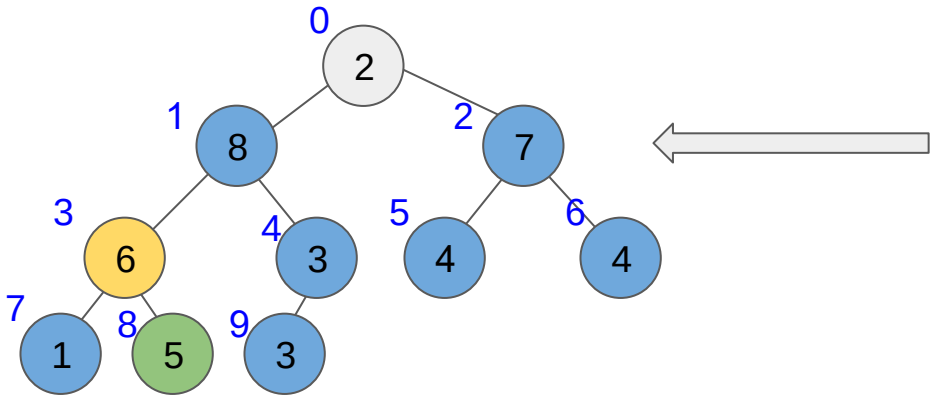


2nd round of fix downs

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

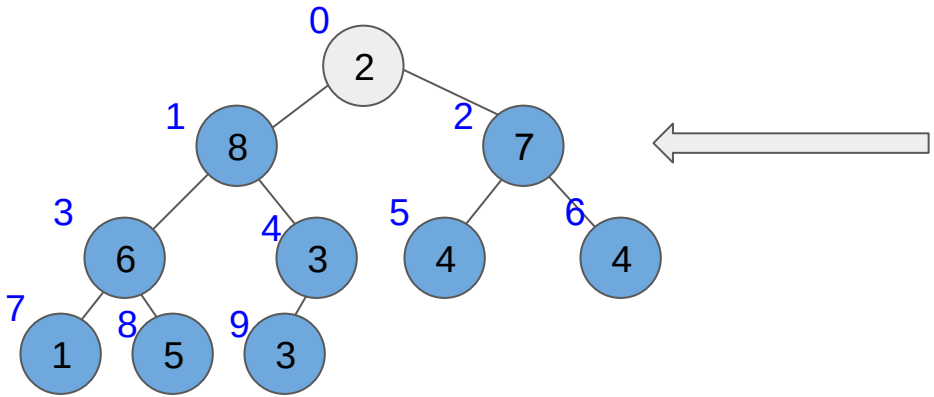


2nd round of fix downs

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

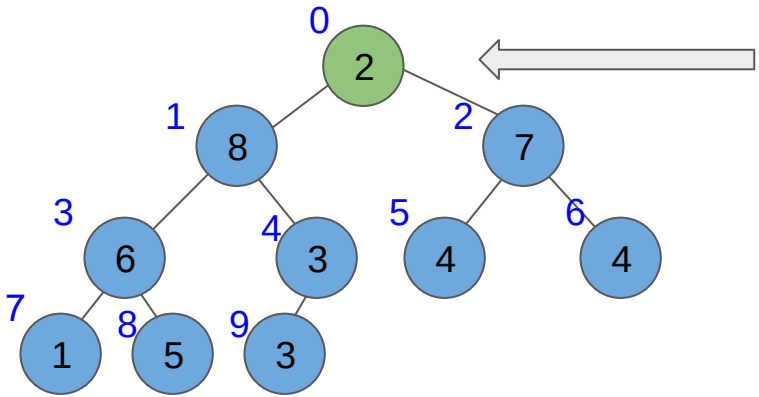


2nd round of fix downs

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

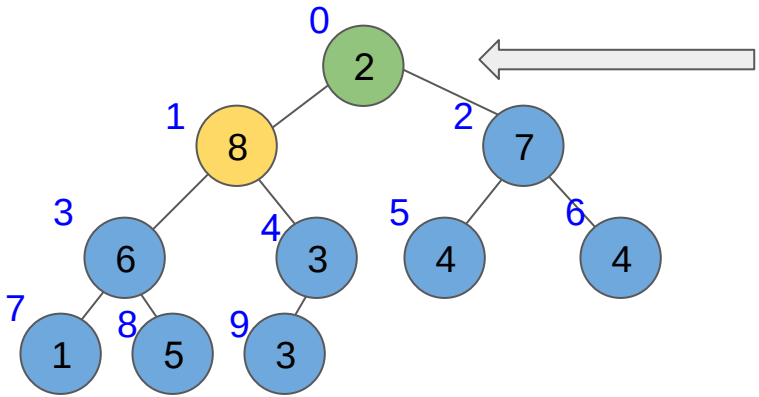


3rd round of fix downs

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

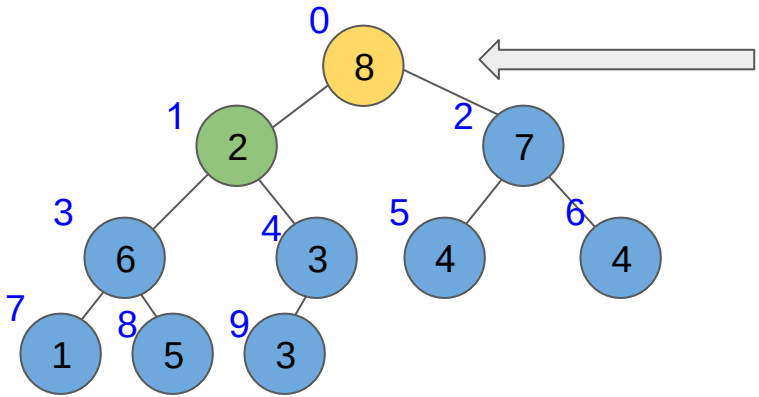


3rd round of fix downs

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

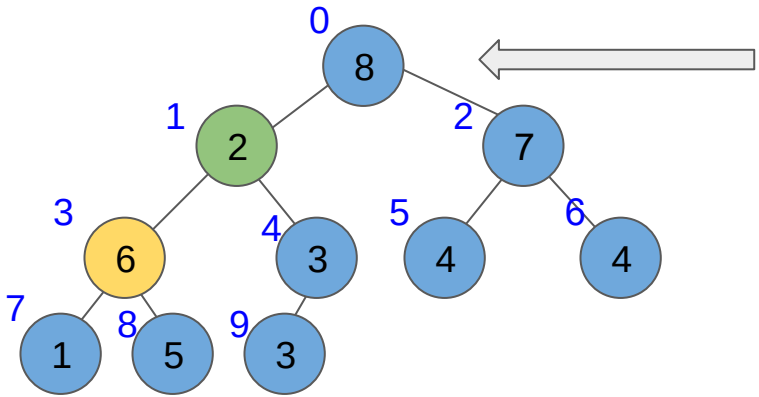


3rd round of fix downs

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

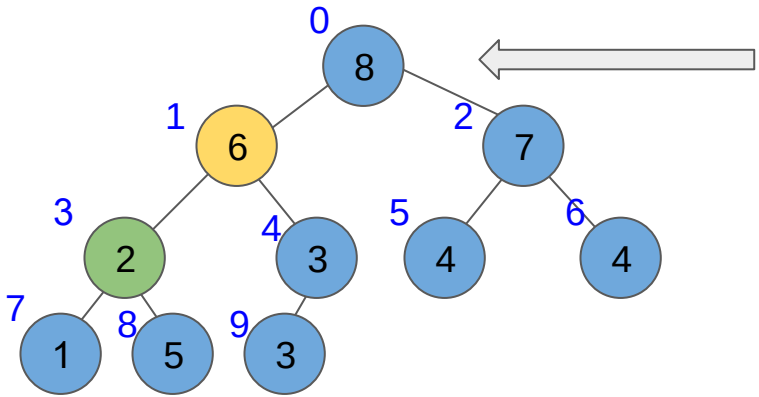


3rd round of fix downs

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

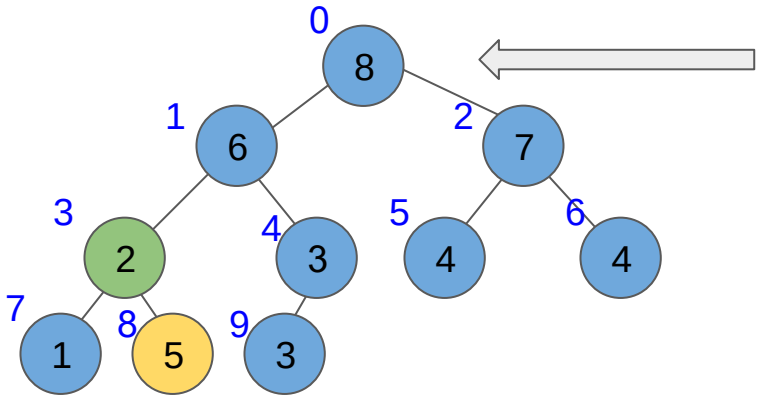


3rd round of fix downs

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

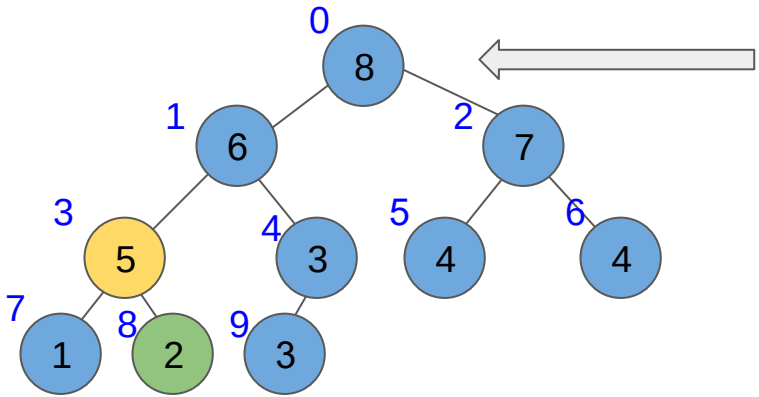


3rd round of fix downs

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

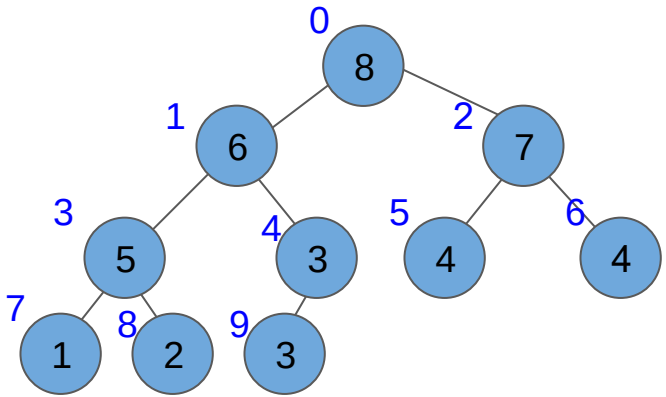


3rd round of fix downs

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

Priority Queue - Implementations - Binary

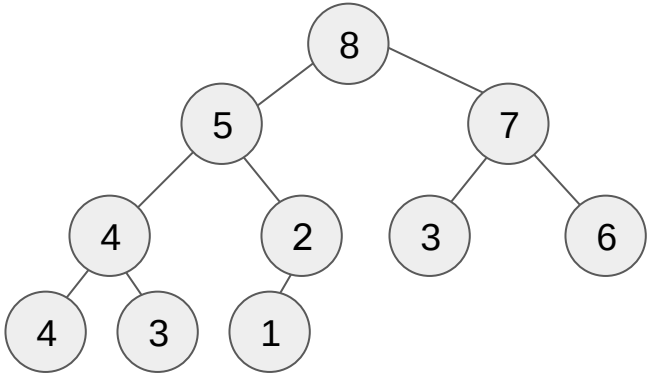
Resultant Heap & Array



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.



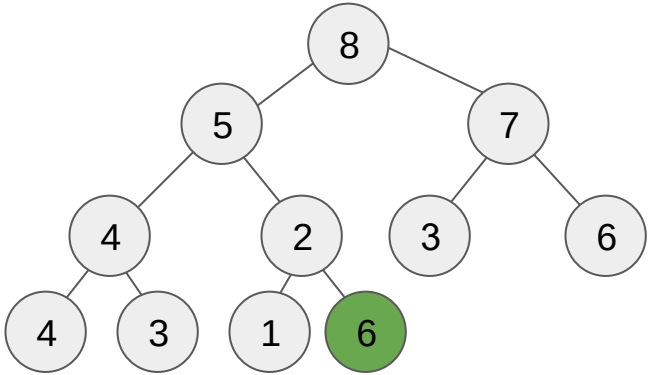
Now, we *fix up* from that location

- 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.



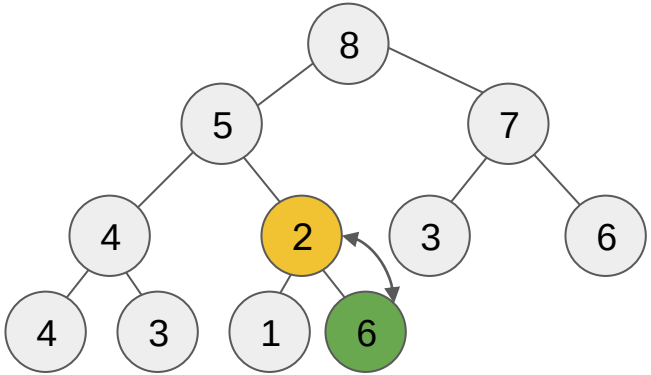
Now, we *fix up* from that location

- 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.



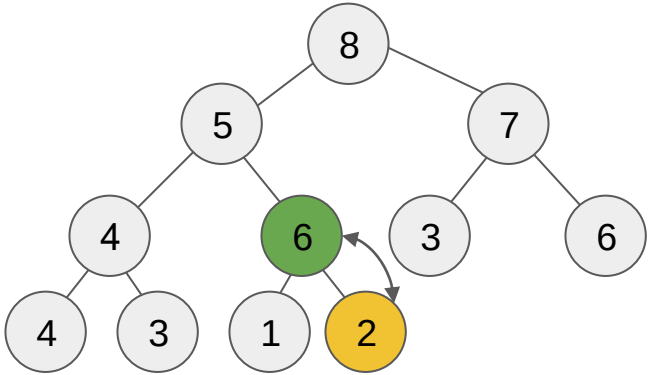
Now, we *fix up* from that location

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

8	5	7	4	2	3	6	4	3	1	6

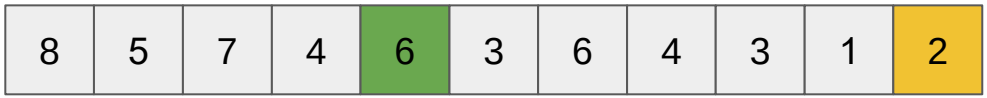
Priority Queue - Implementations - Binary - Push

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



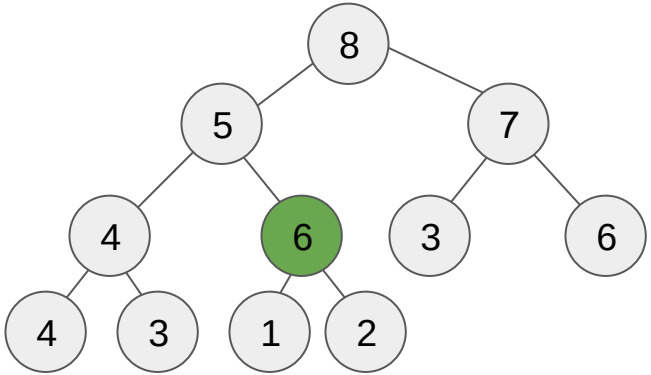
Now, we *fix up* from that location

- 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.



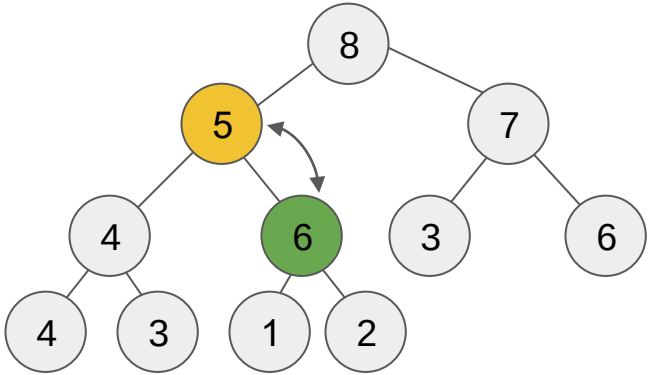
Now, we *fix up* from that location

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

8	5	7	4	6	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.



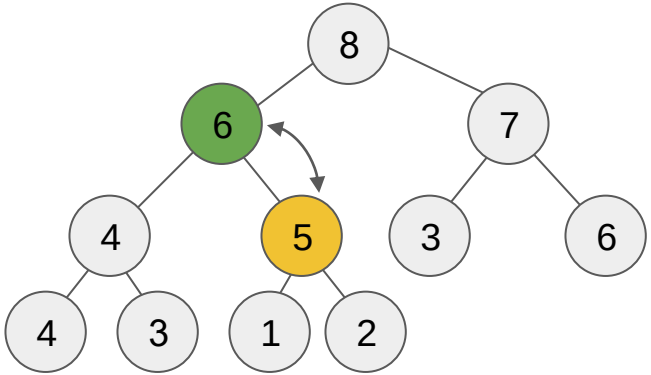
Now, we *fix up* from that location

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

8	5	7	4	6	3	6	4	3	1	2

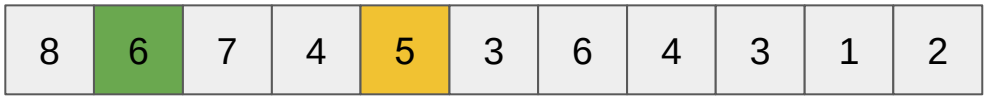
Priority Queue - Implementations - Binary - Push

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



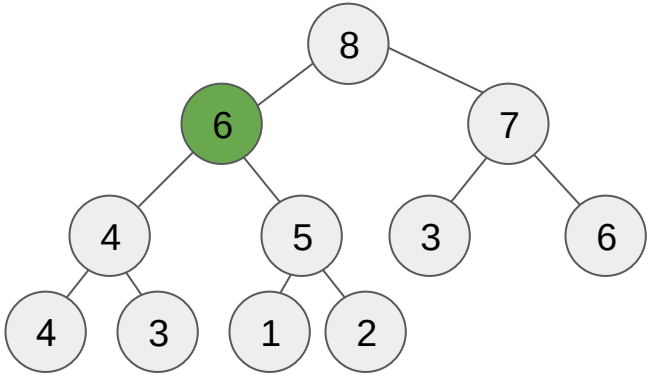
Now, we *fix up* from that location

- 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.



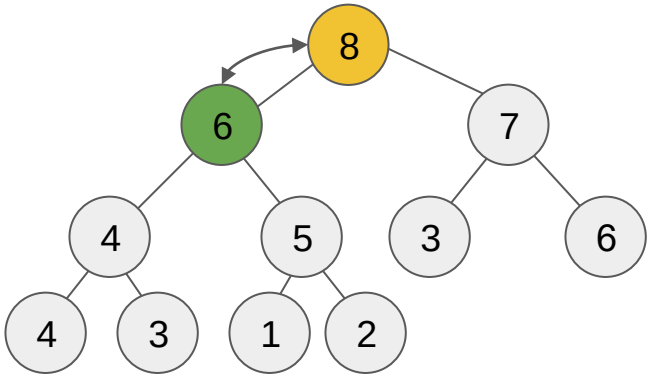
Now, we *fix up* from that location

- 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

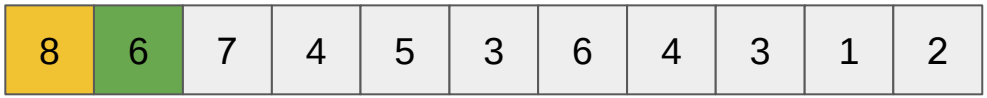
Priority Queue - Implementations - Binary - Push

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



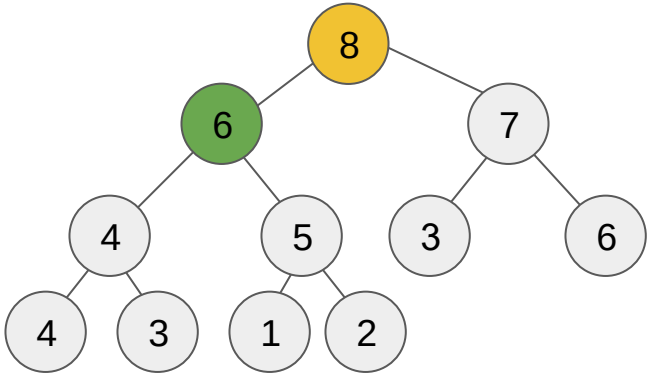
Now, we *fix up* from that location

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



Priority Queue - Implementations - Binary - Push

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



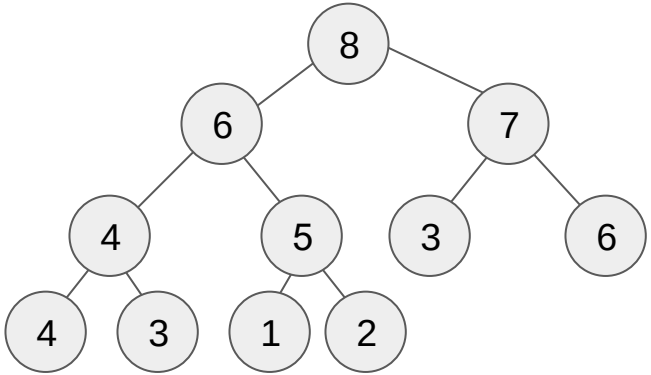
Now, we *fix up* from that location

- 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.



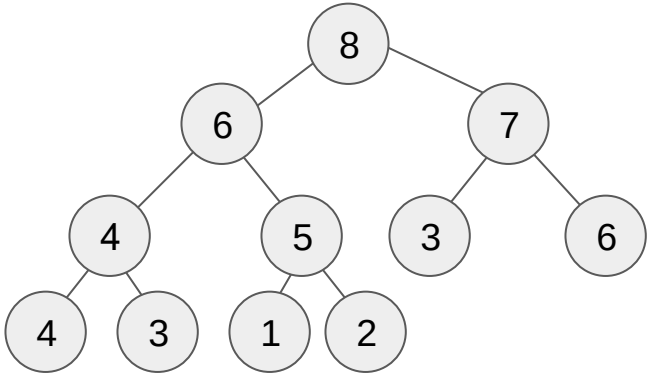
Now, we *fix up* from that location

- 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.

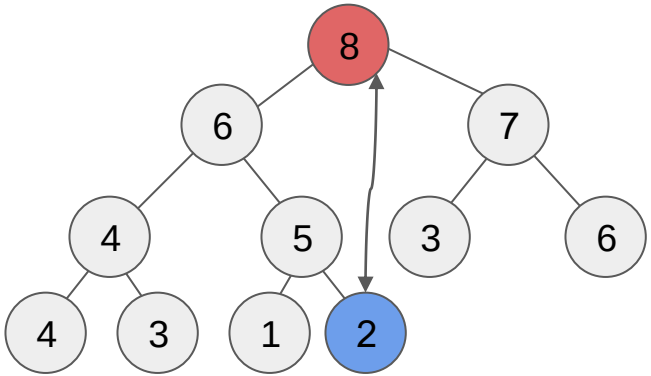
Now we *fix down* starting at the new root.

- 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.

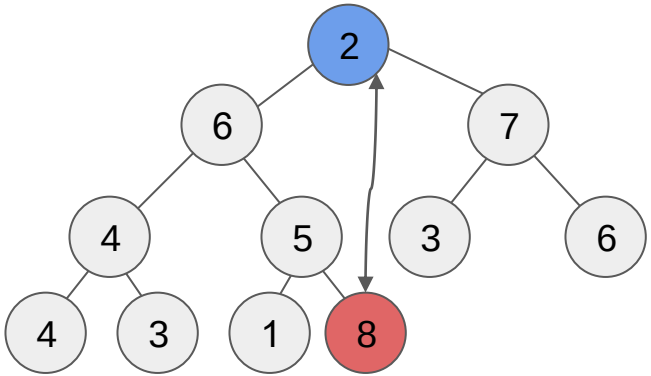
Now we *fix down* starting at the new root.

- 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.

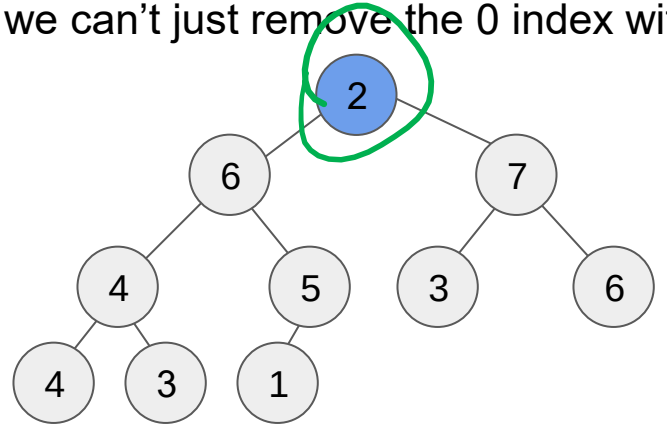
Now we *fix down* starting at the new root.

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

2	6	7	4	5	3	6	4	3	1	8

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.

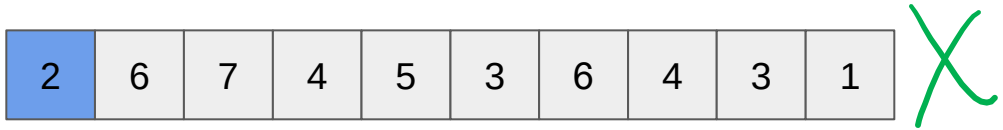


So, we start by swapping the root with the last element.

Then, we pop back to remove the old root.

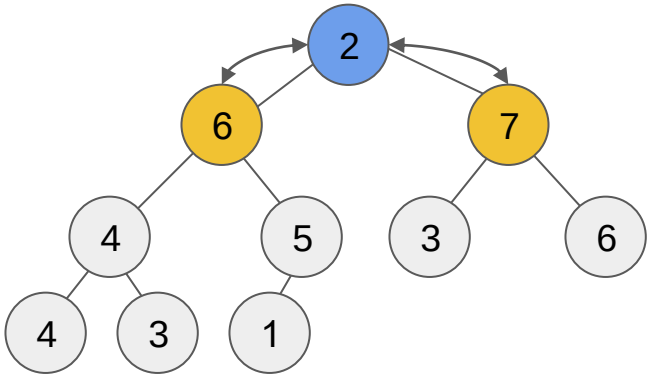
Now we *fix down* starting at the new root.

- 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.

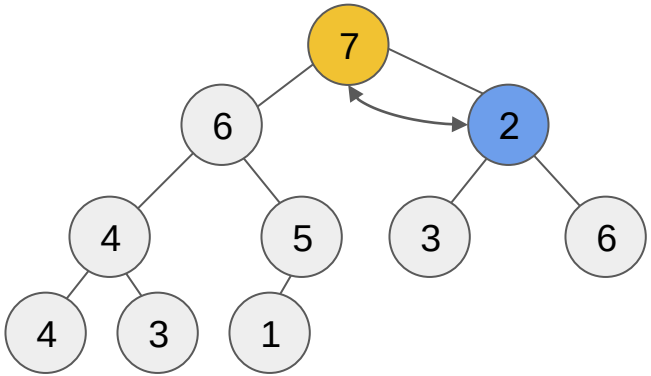
Now we *fix down* starting at the new root.

- **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.

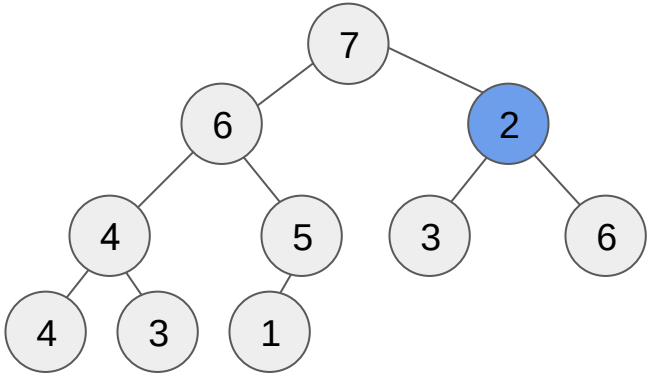
Now we *fix down* starting at the new root.

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

7	6	2	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.

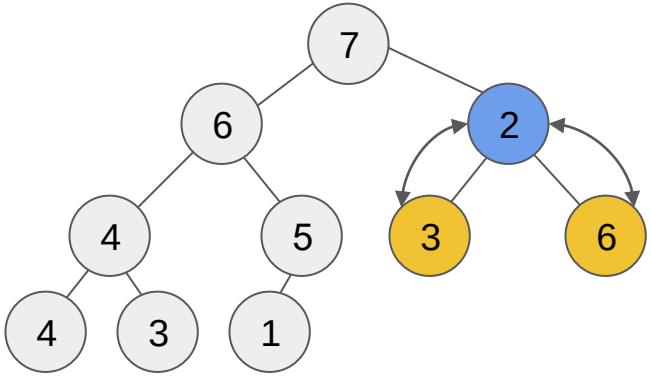
Now we *fix down* starting at the new root.

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

7	6	2	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.

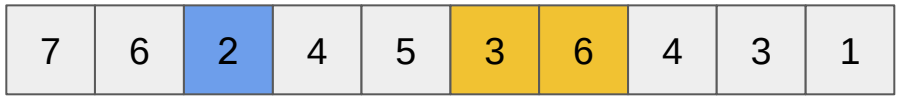


So, we start by swapping the root with the last element.

Then, we pop back to remove the old root.

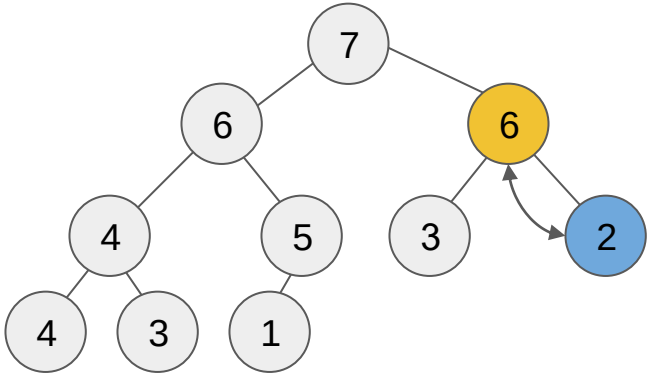
Now we *fix down* starting at the new root.

- **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.

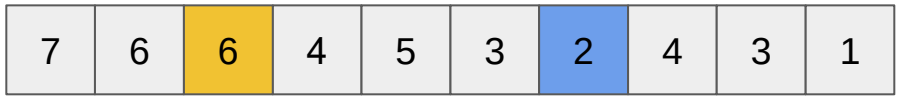


So, we start by swapping the root with the last element.

Then, we pop back to remove the old root.

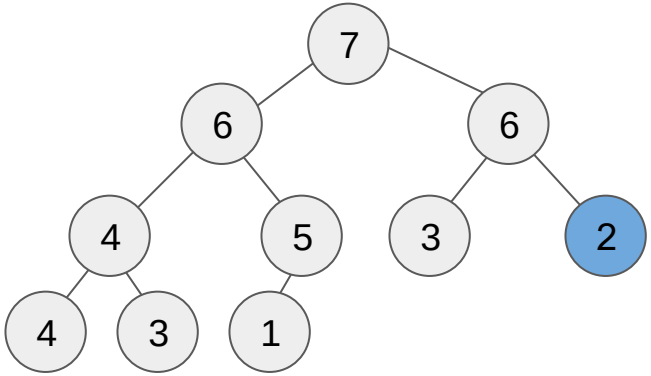
Now we *fix down* starting at the new root.

- Is it smaller than either child?
  - **If so, swap with the larger, repeat on that new positio**
  - 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.

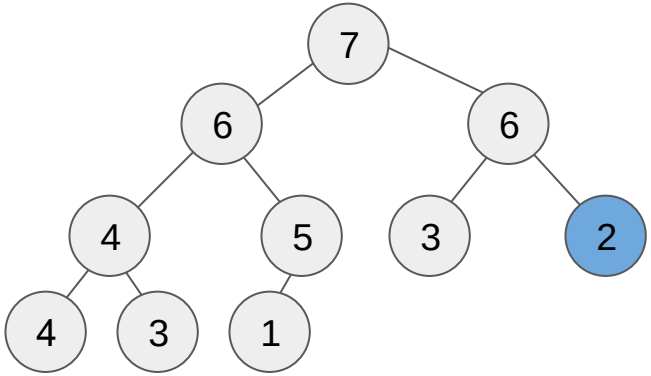
Now we *fix down* starting at the new root.

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

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.

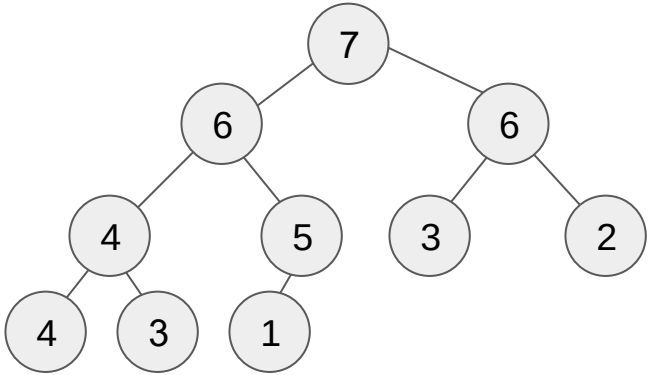
Now we *fix down* starting at the new root.

- **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.

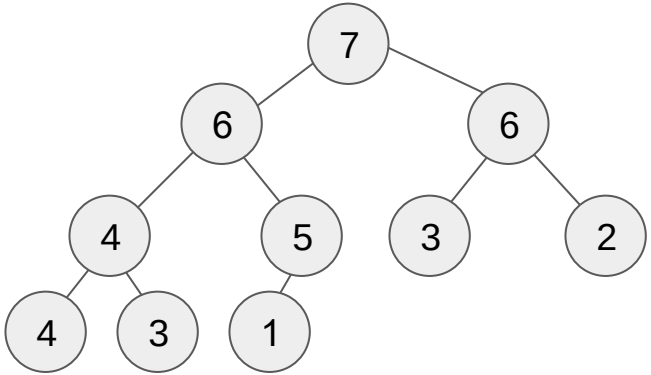
Now we *fix down* starting at the new root.

- 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.**

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.

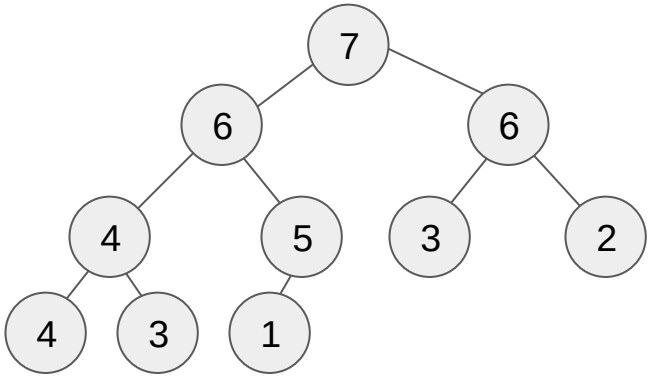
Now we *fix down* starting at the new root.

- 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?



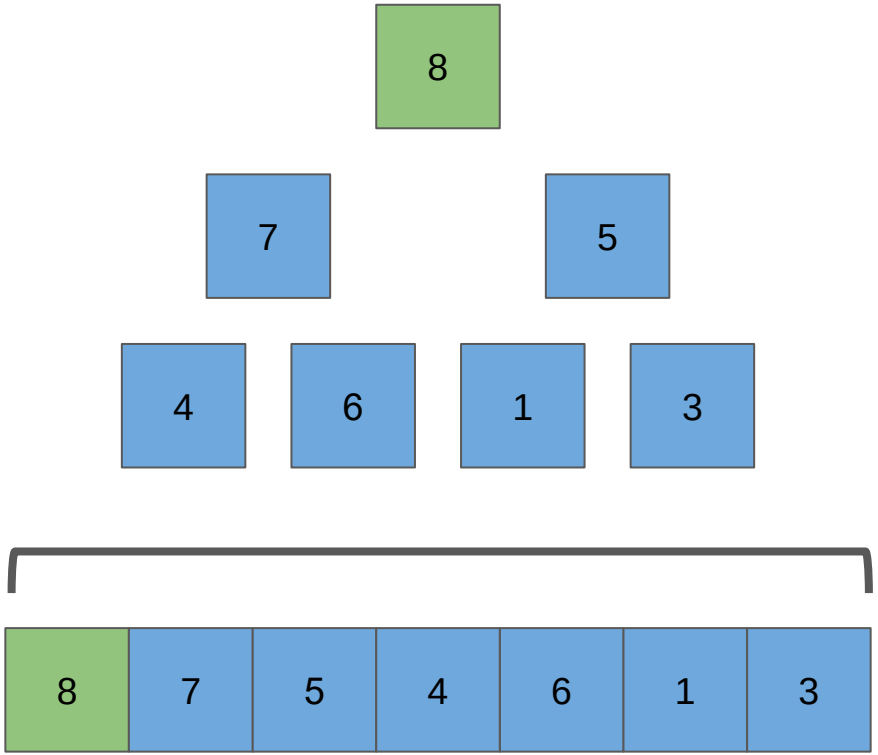
top() -> returns data[0] in **O(1) time**

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

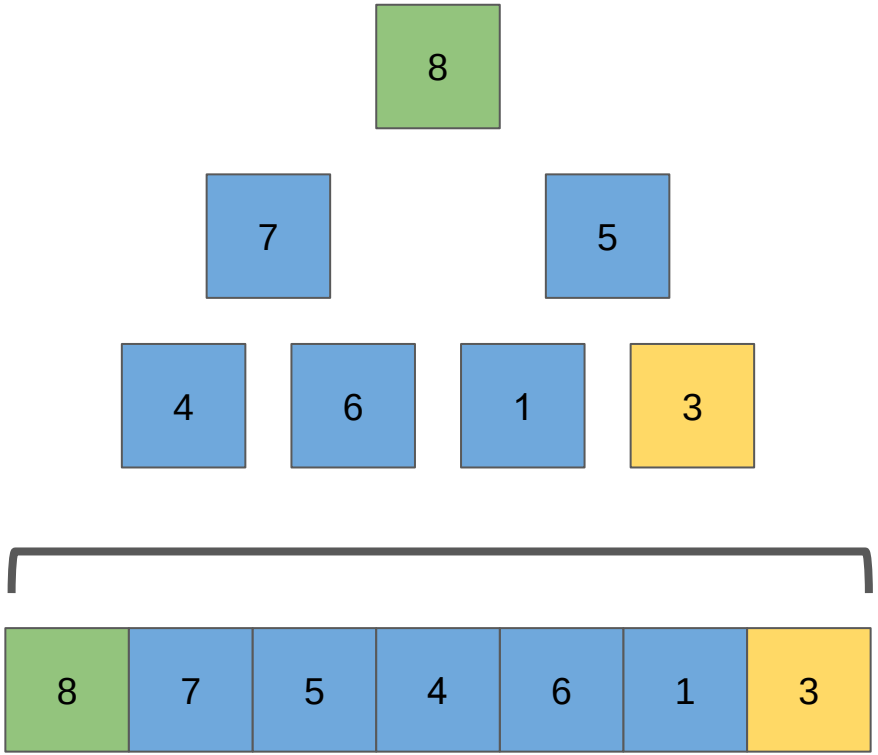
pop() ->
Swap, pop_back, fix_down;
Fix_down runs in time proportional to tree height,
O(log n)

7	6	6	4	5	3	2	4	3	1

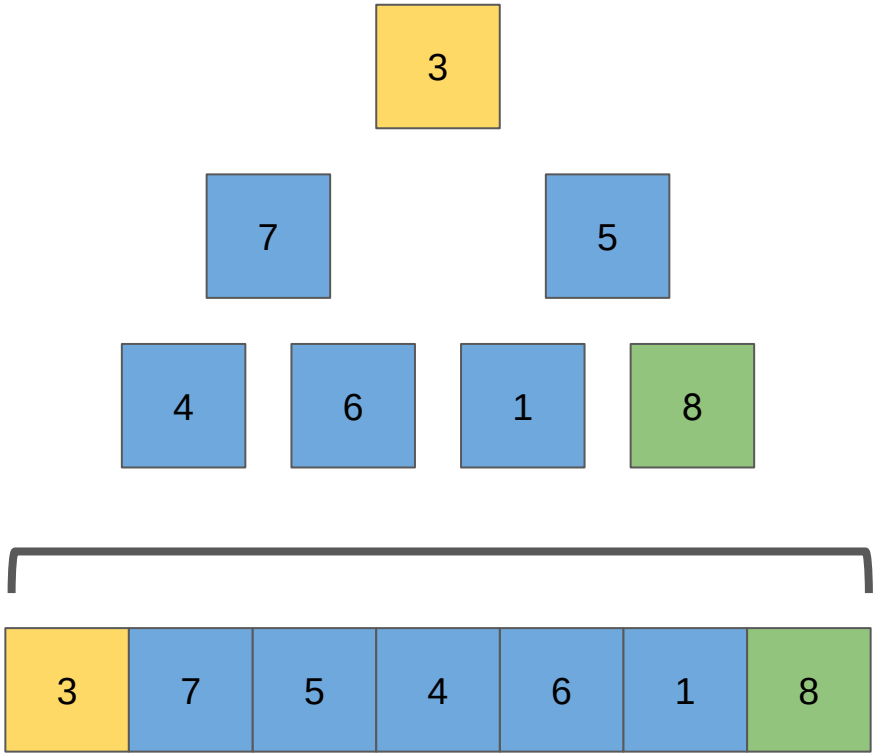
Heapsort



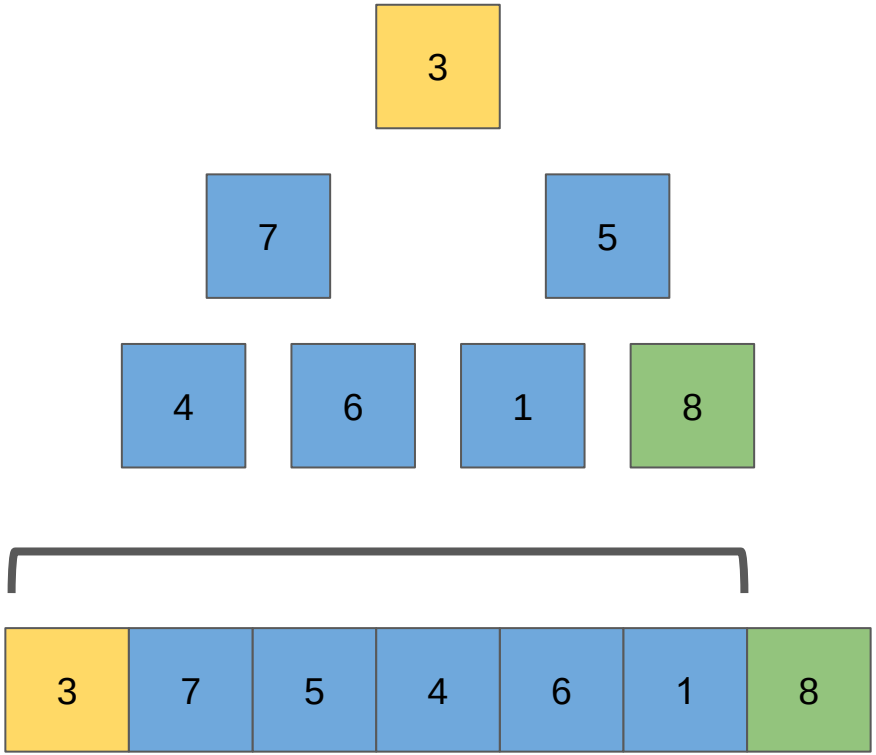
Heapsort



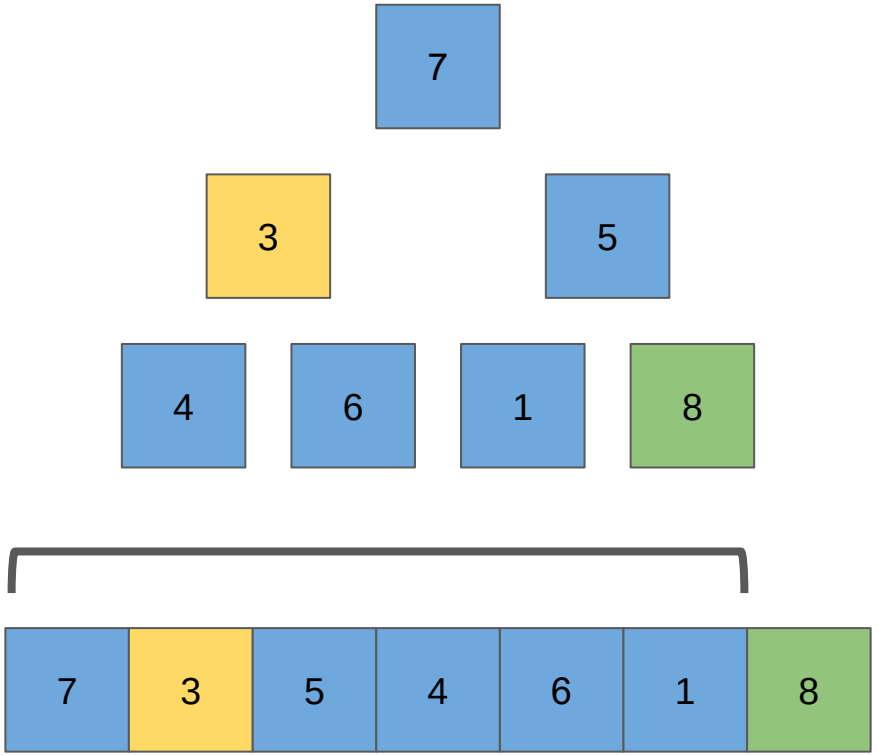
Heapsort



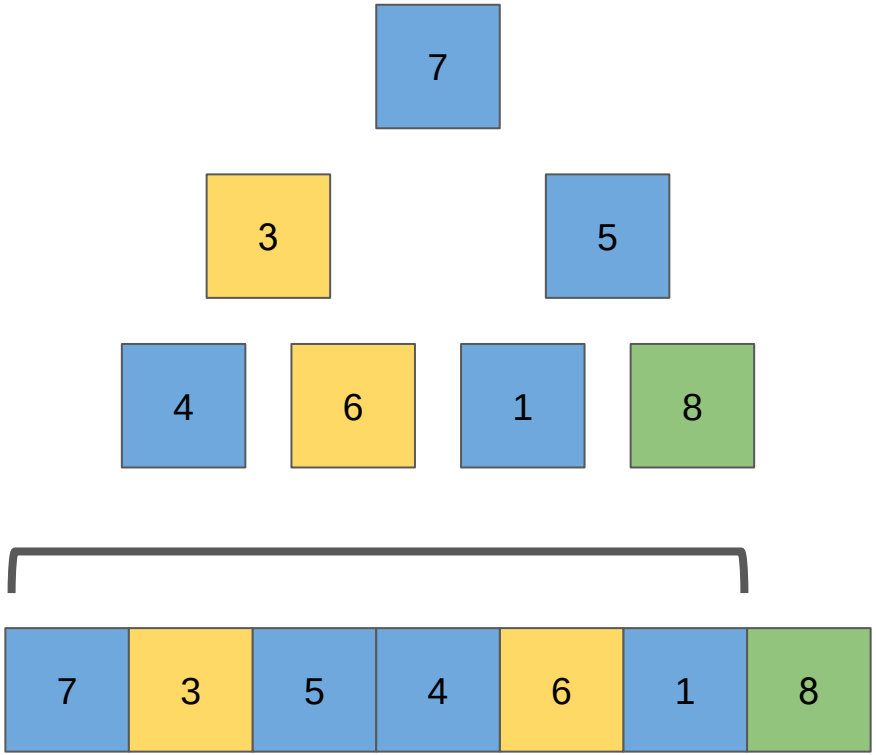
Heapsort



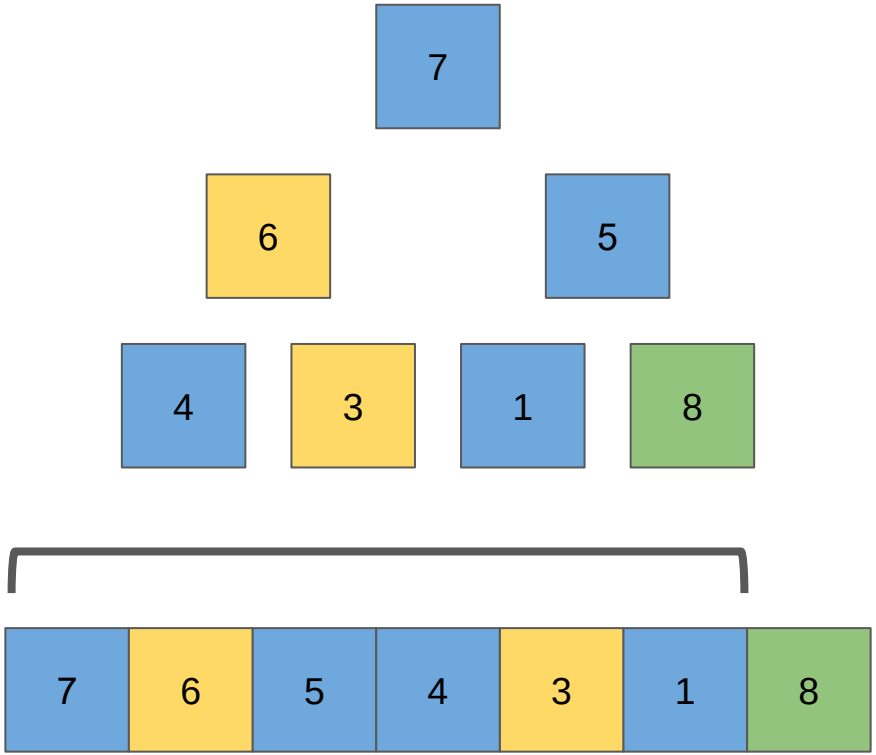
Heapsort



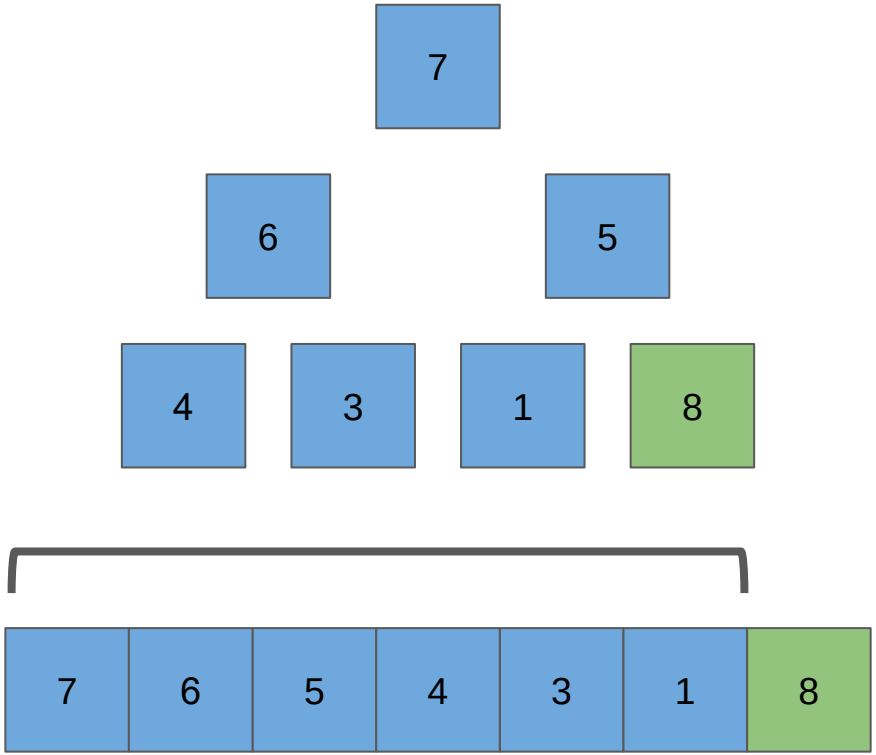
Heapsort



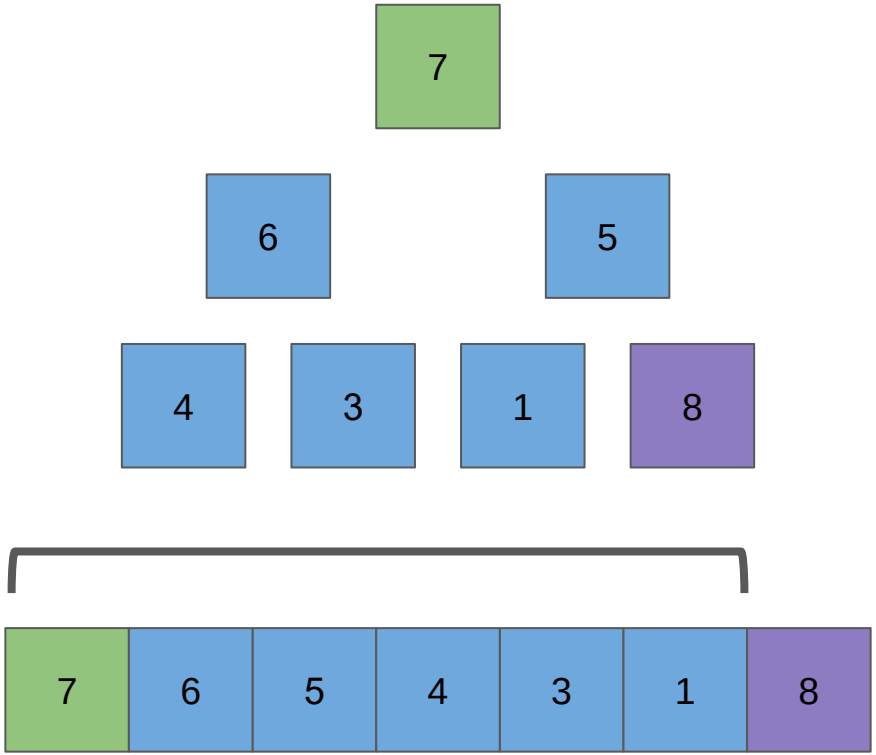
Heapsort



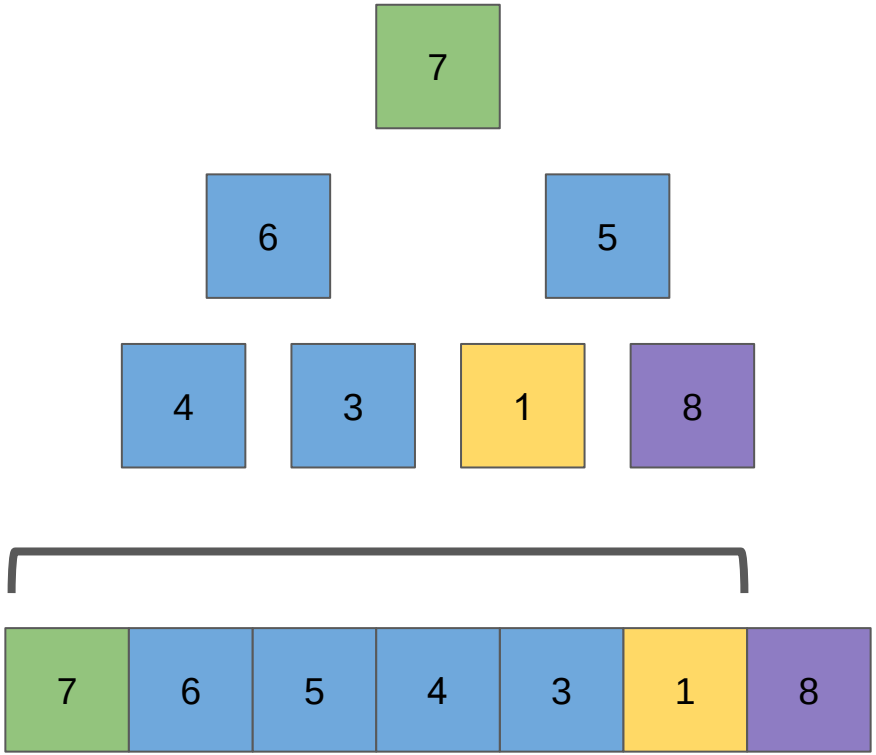
Heapsort



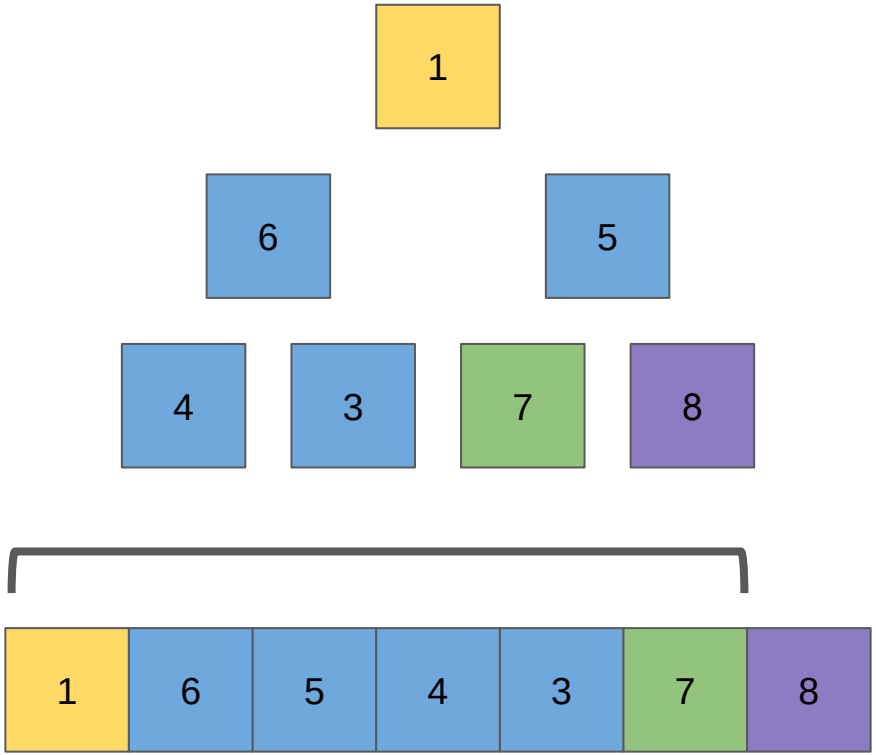
Heapsort



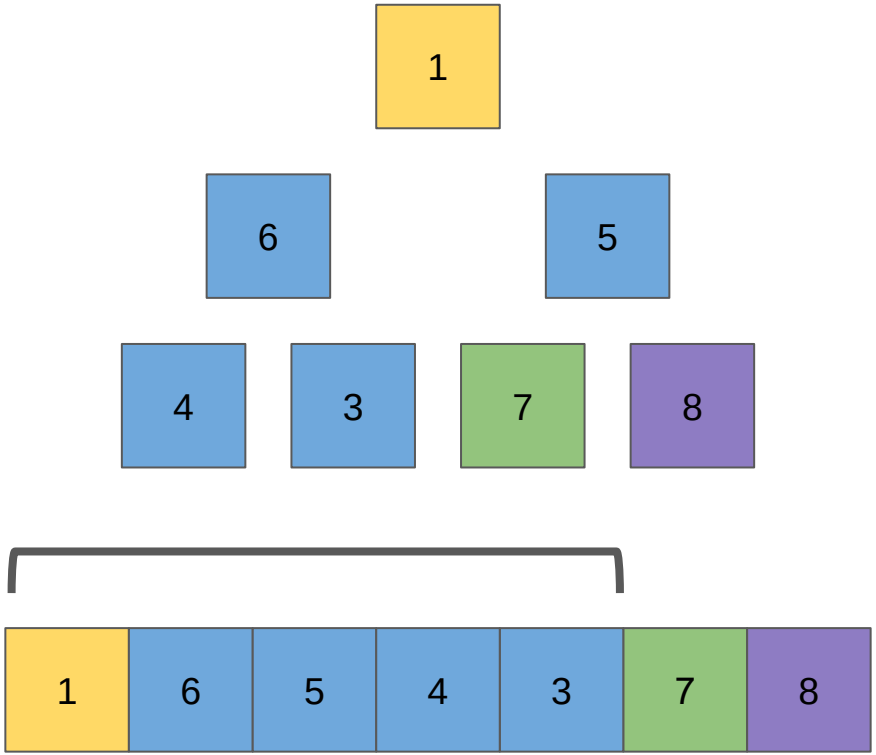
Heapsort



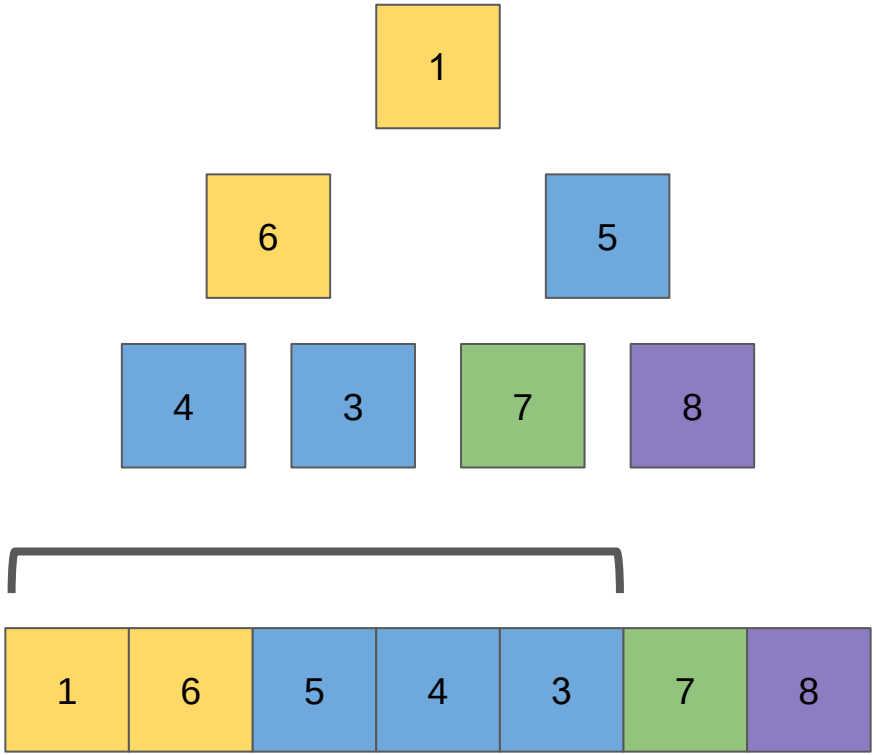
Heapsort



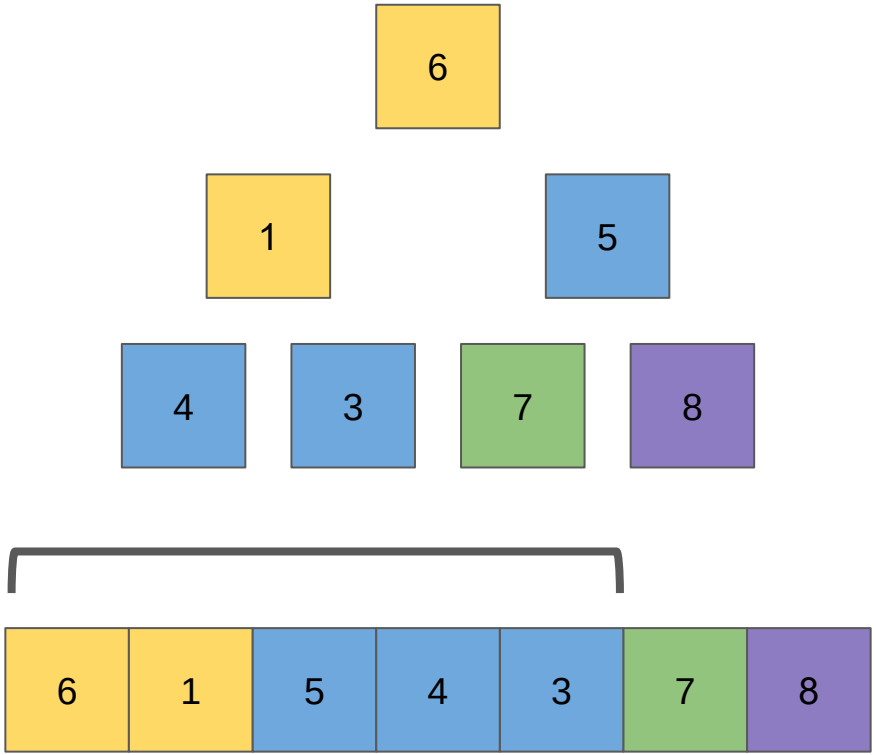
Heapsort



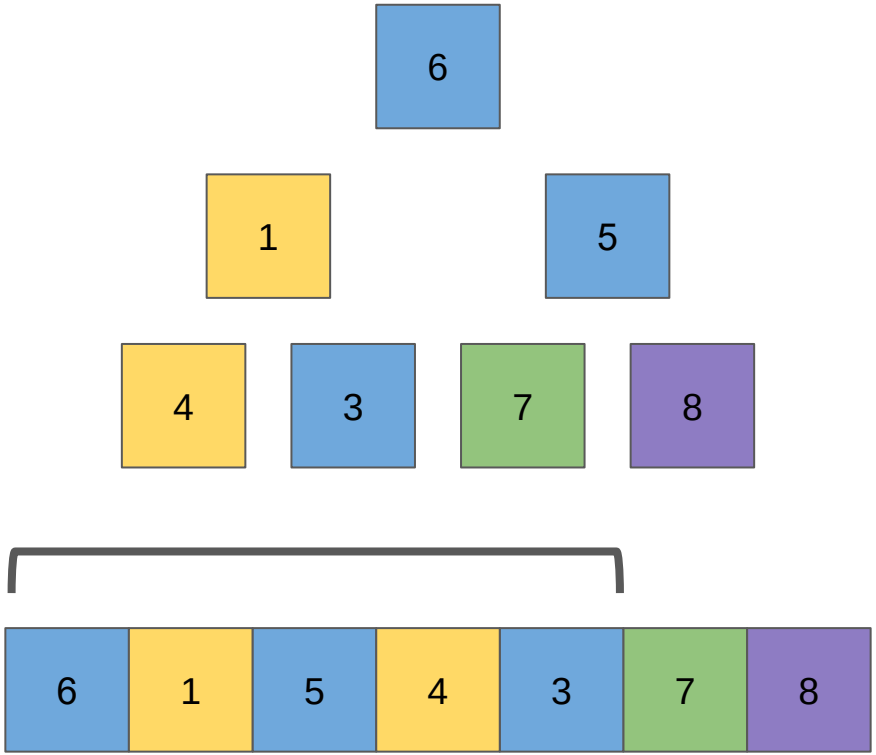
Heapsort



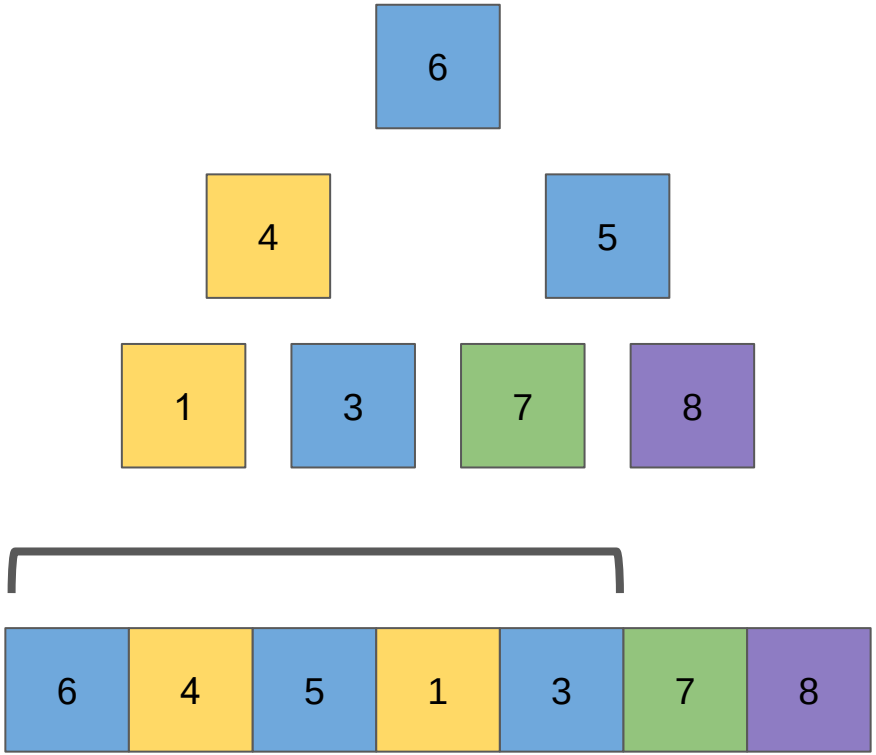
Heapsort



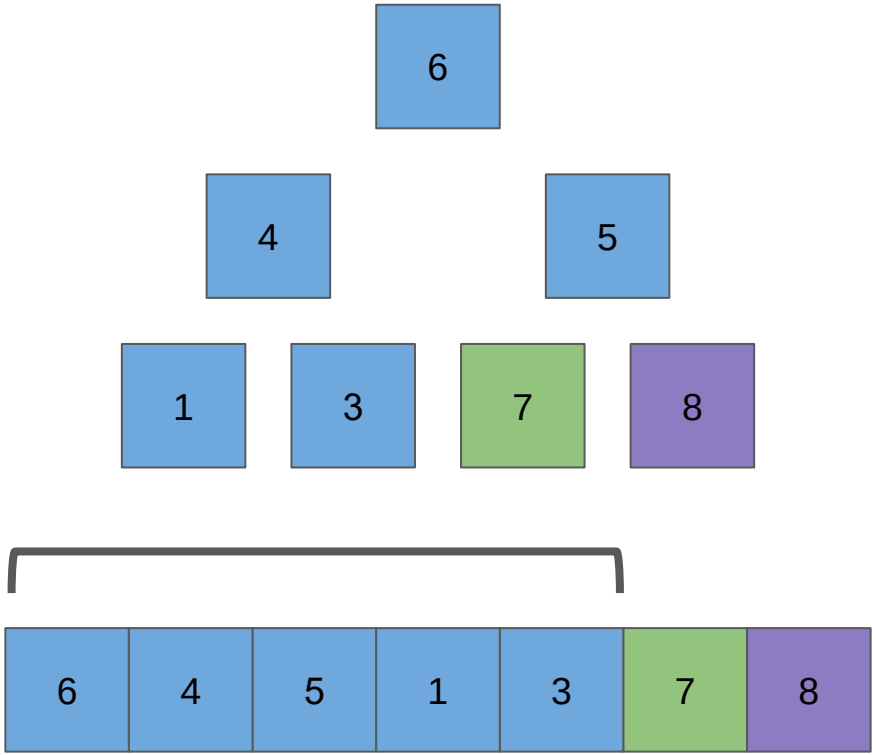
Heapsort



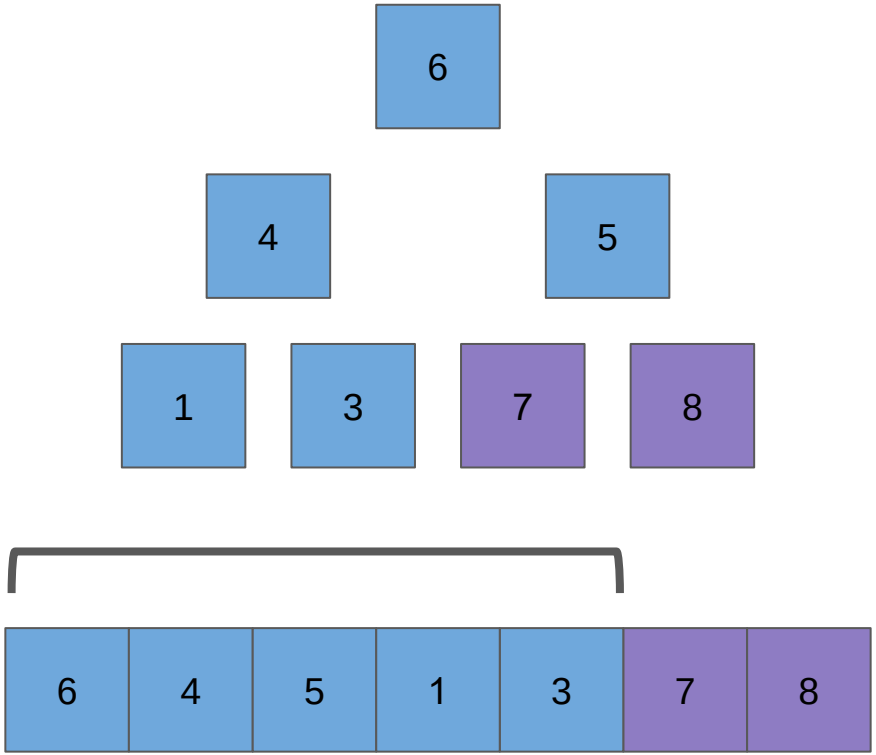
Heapsort



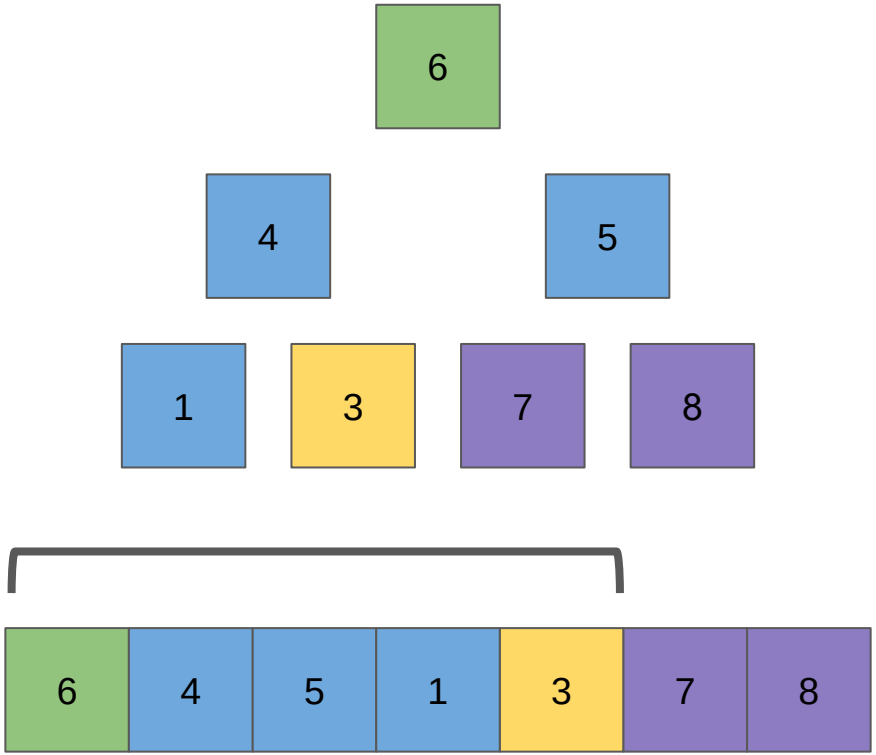
Heapsort



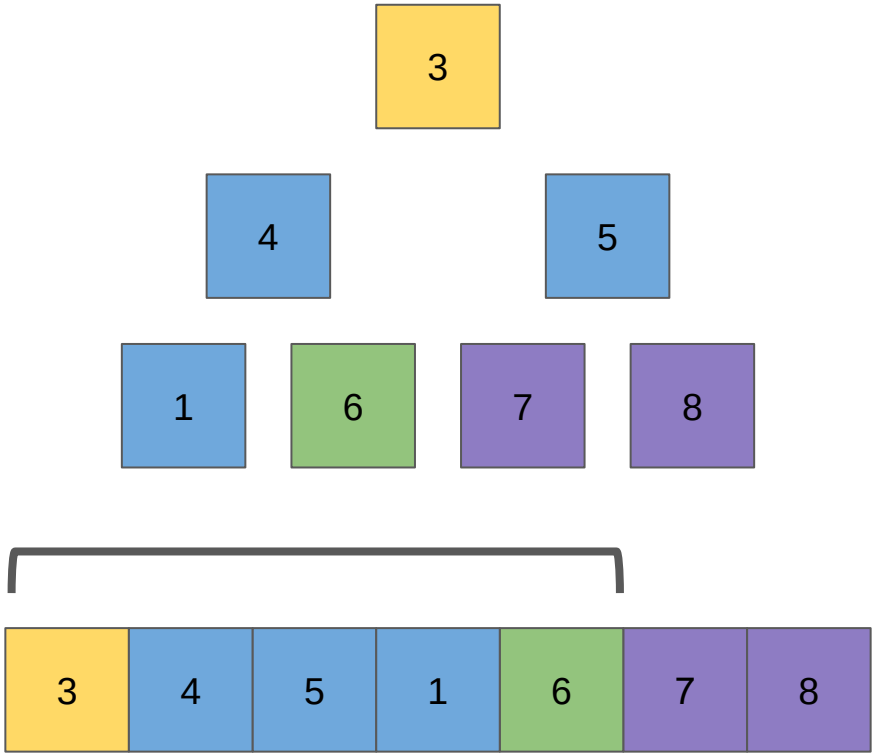
Heapsort



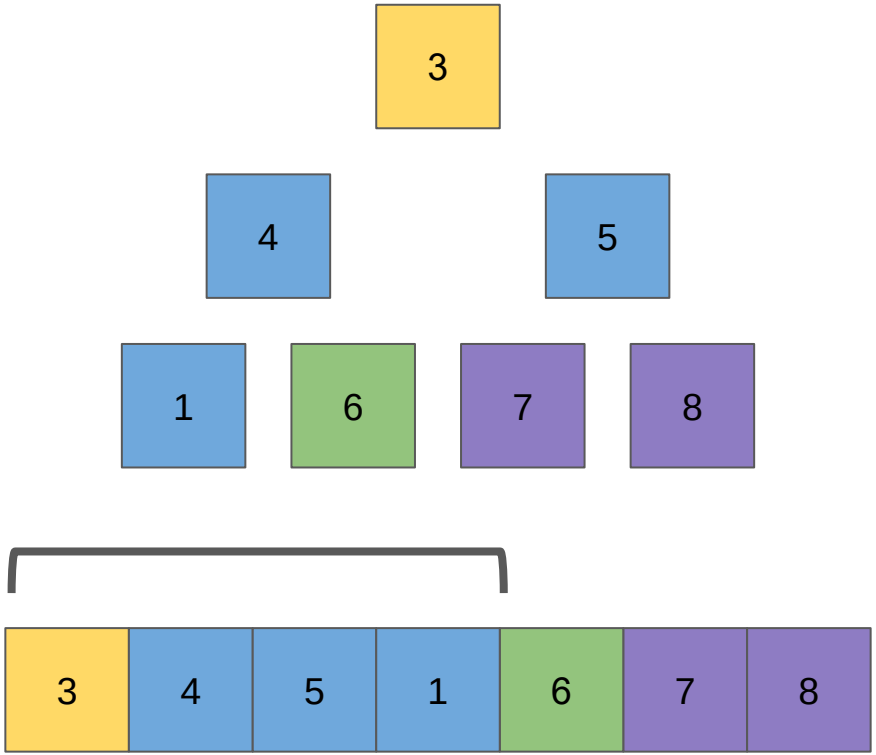
Heapsort



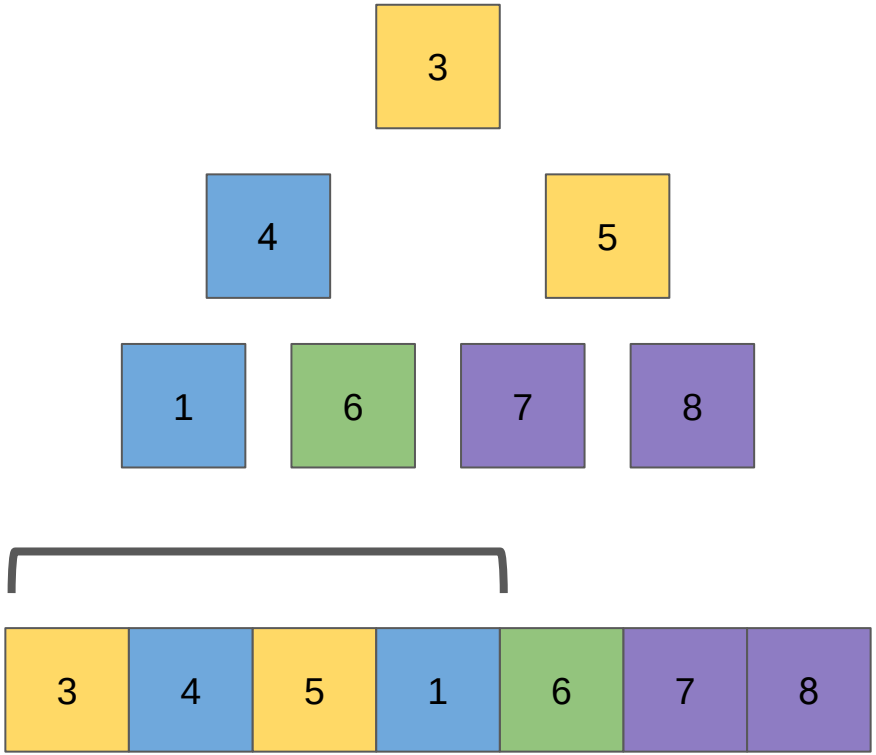
Heapsort



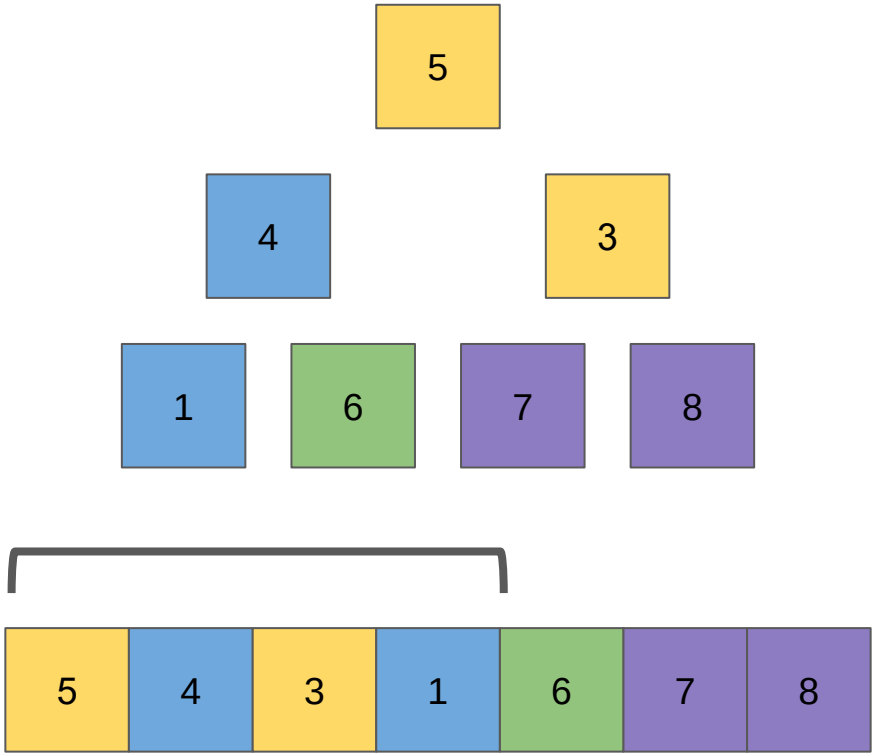
Heapsort



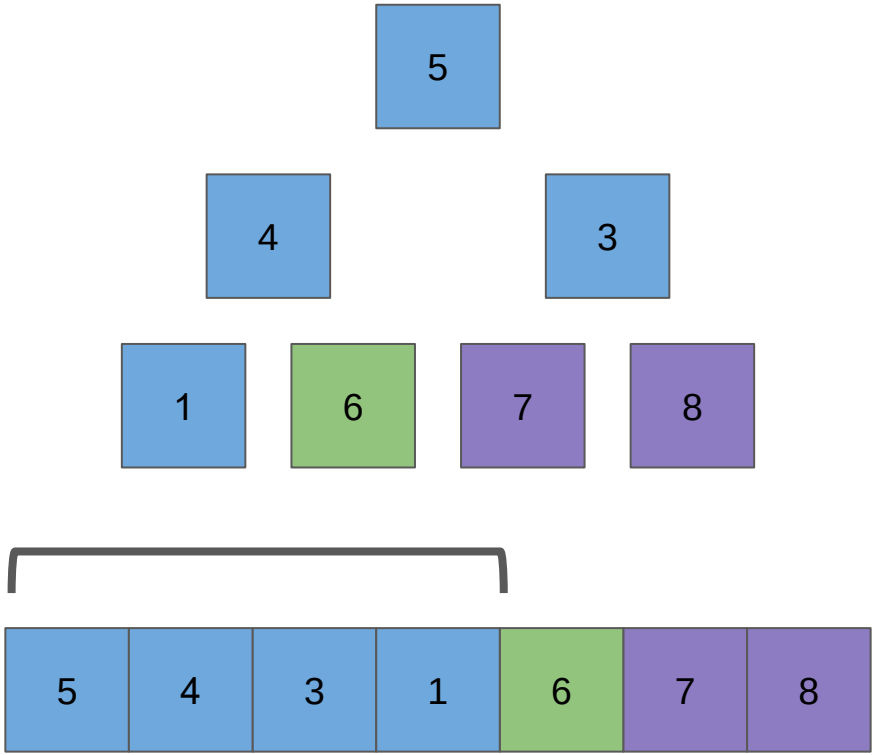
Heapsort



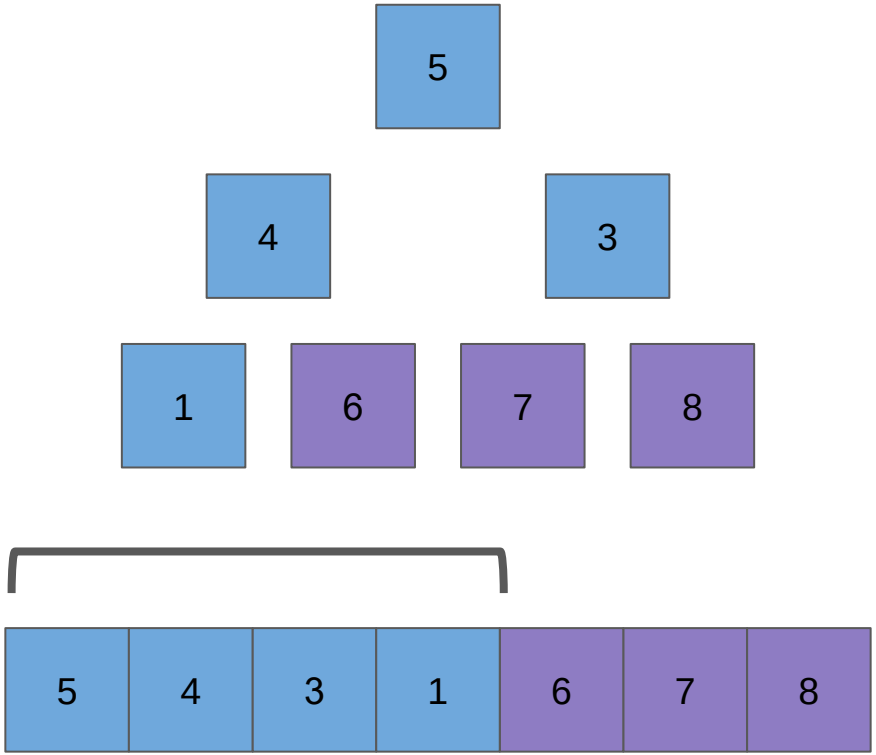
Heapsort



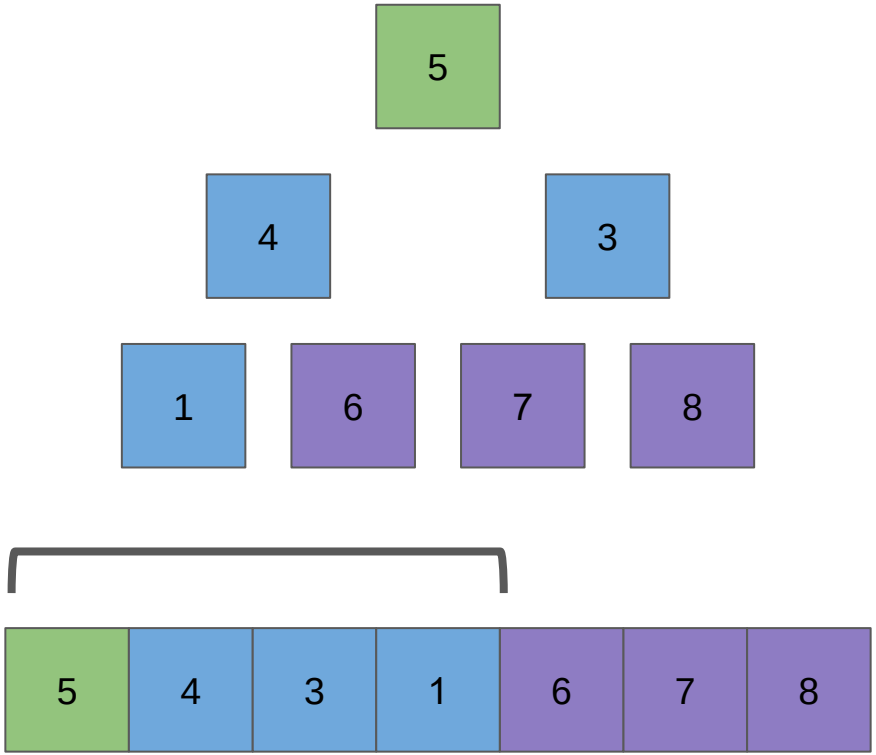
Heapsort



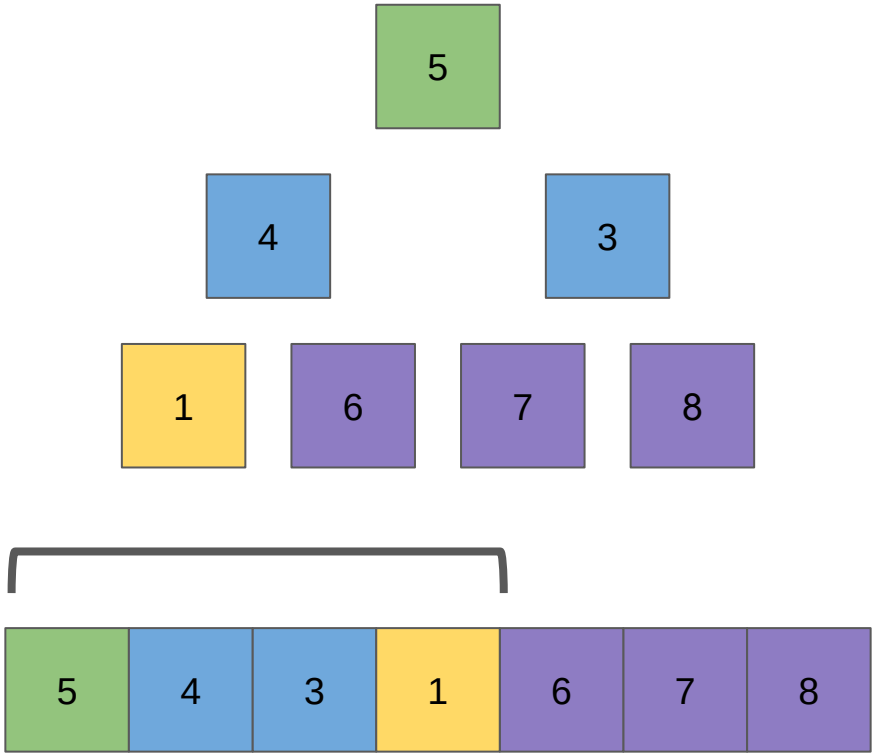
Heapsort



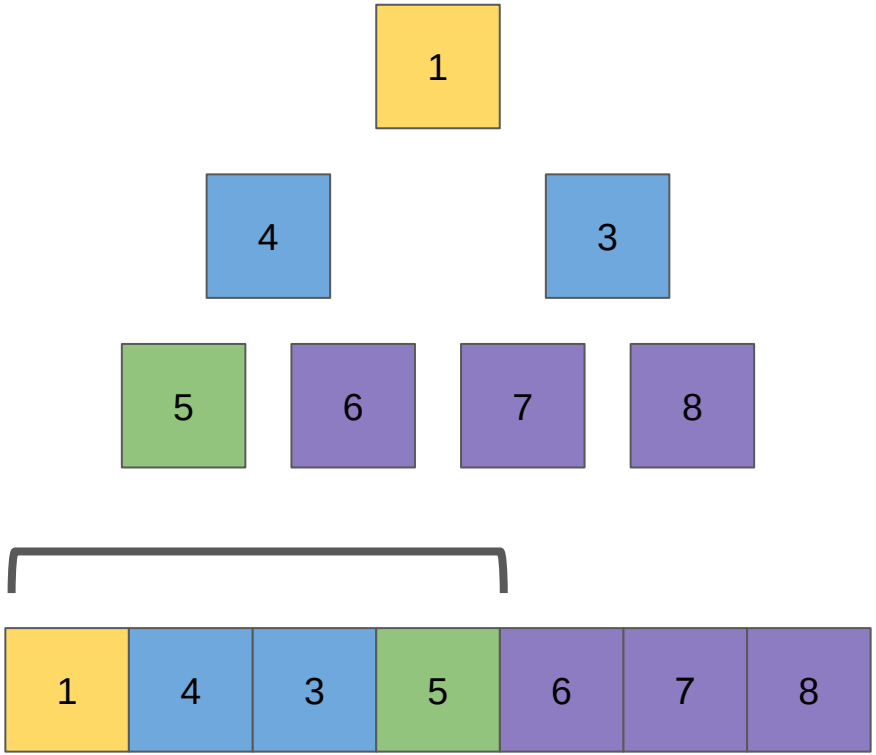
Heapsort



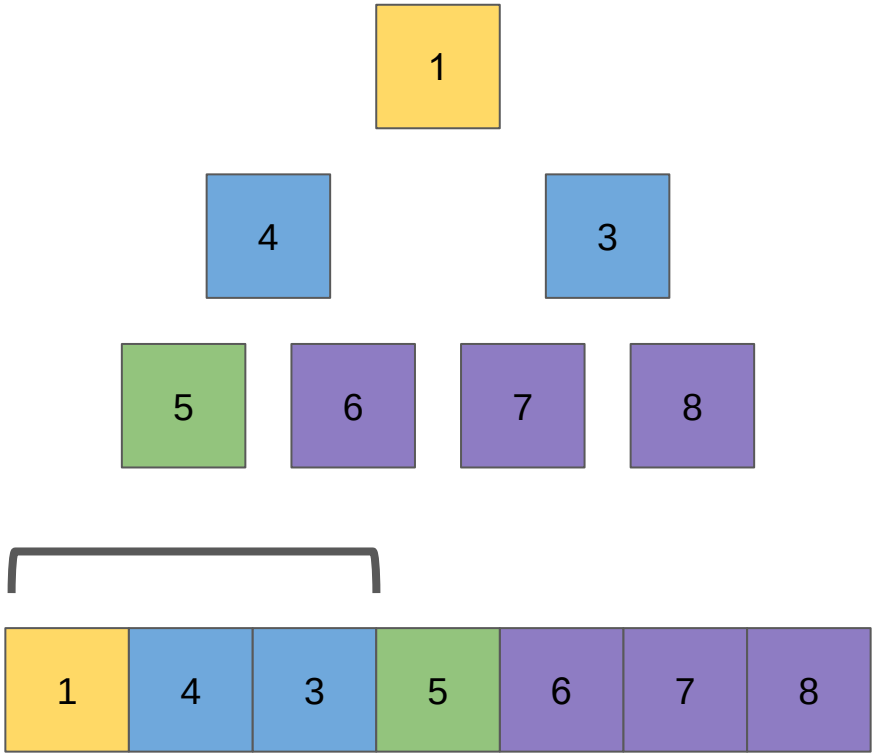
Heapsort



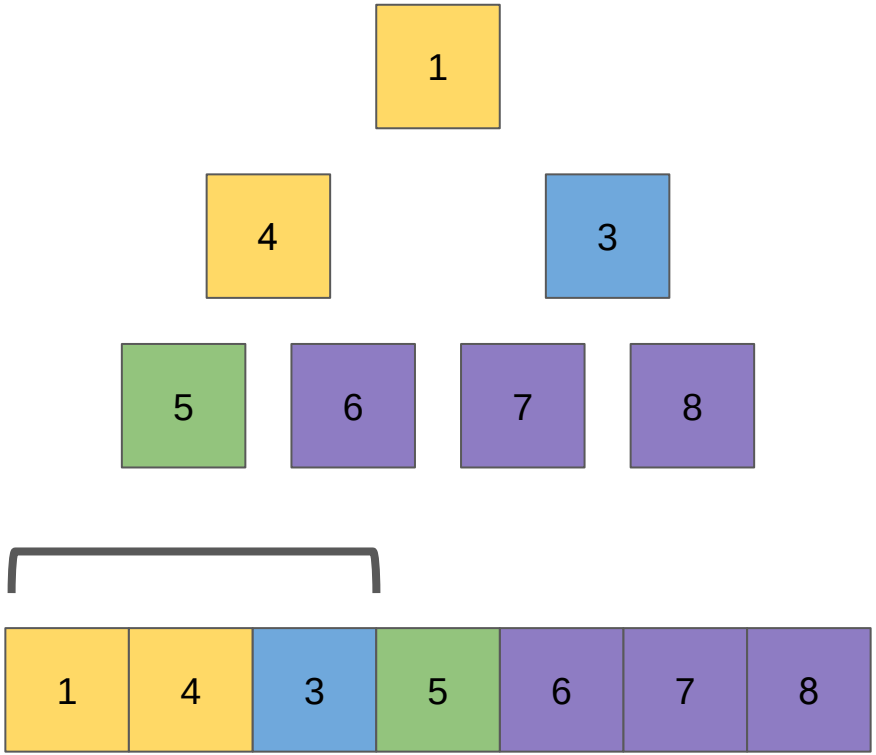
Heapsort



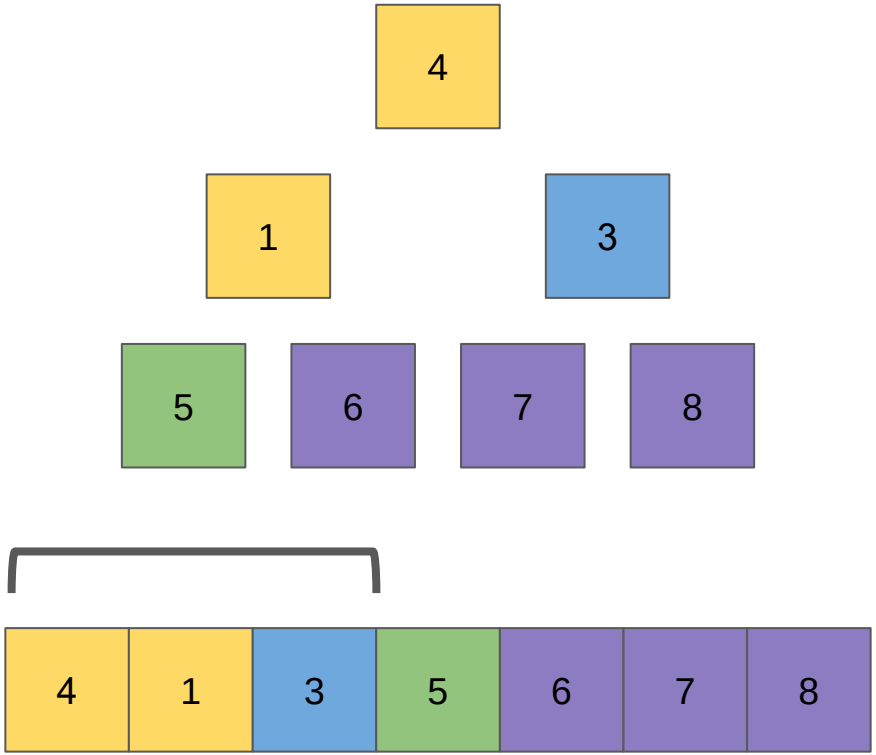
Heapsort



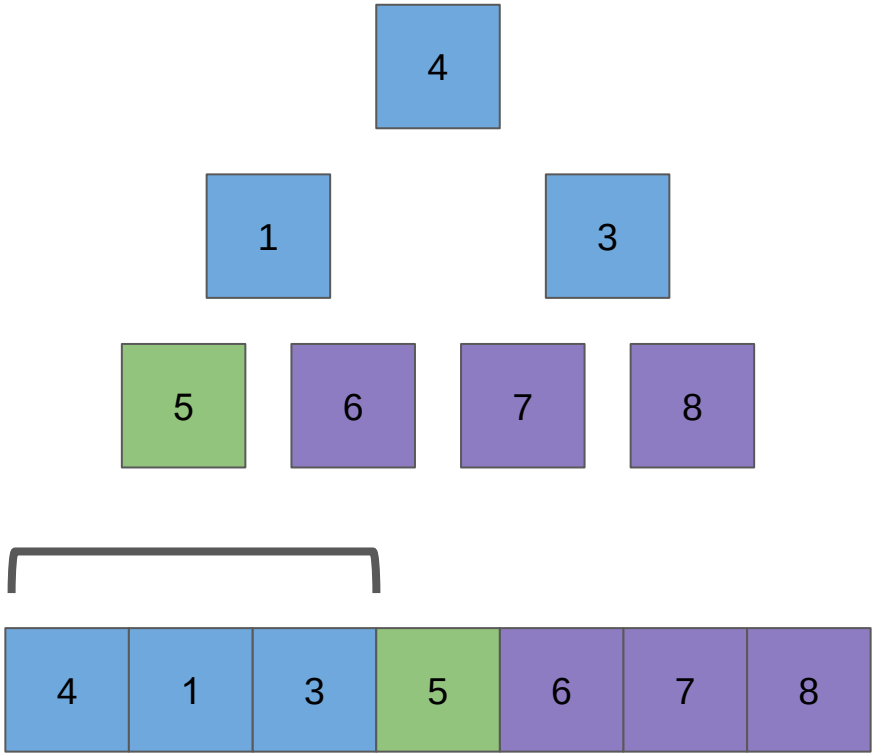
Heapsort



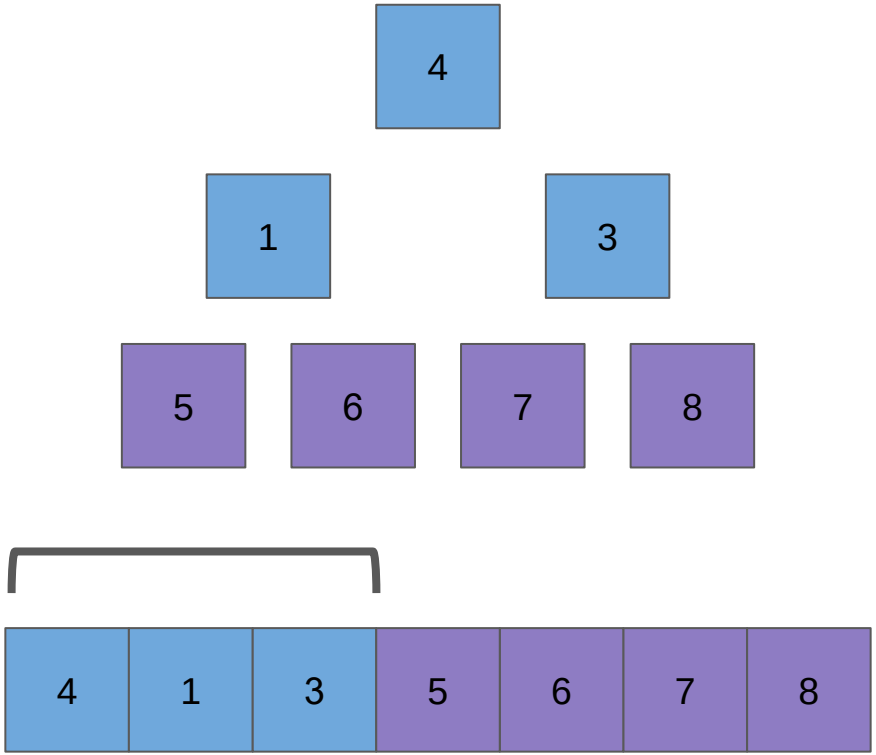
Heapsort



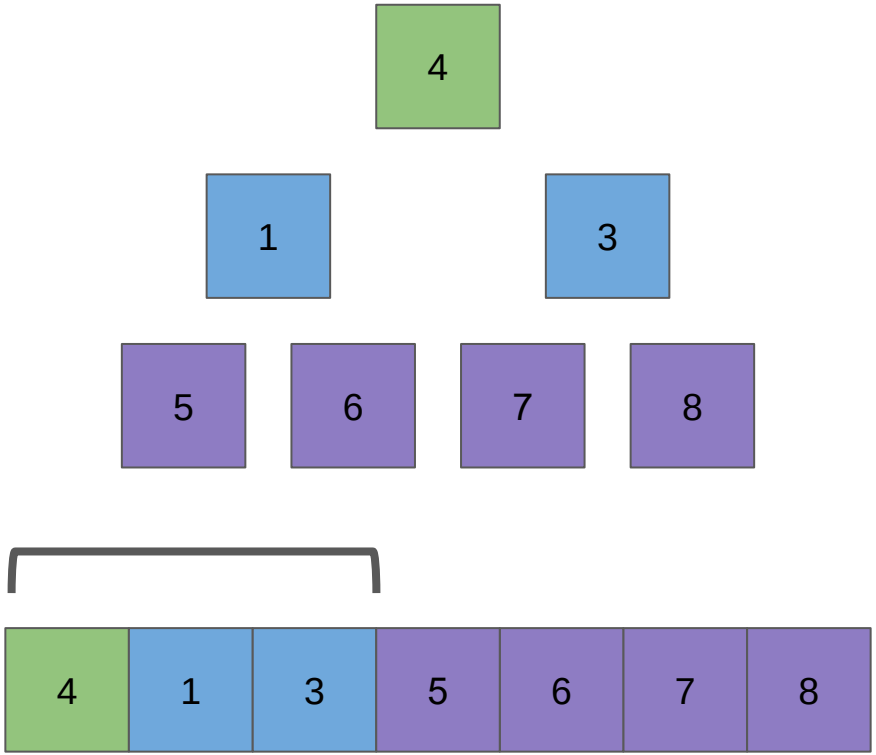
Heapsort



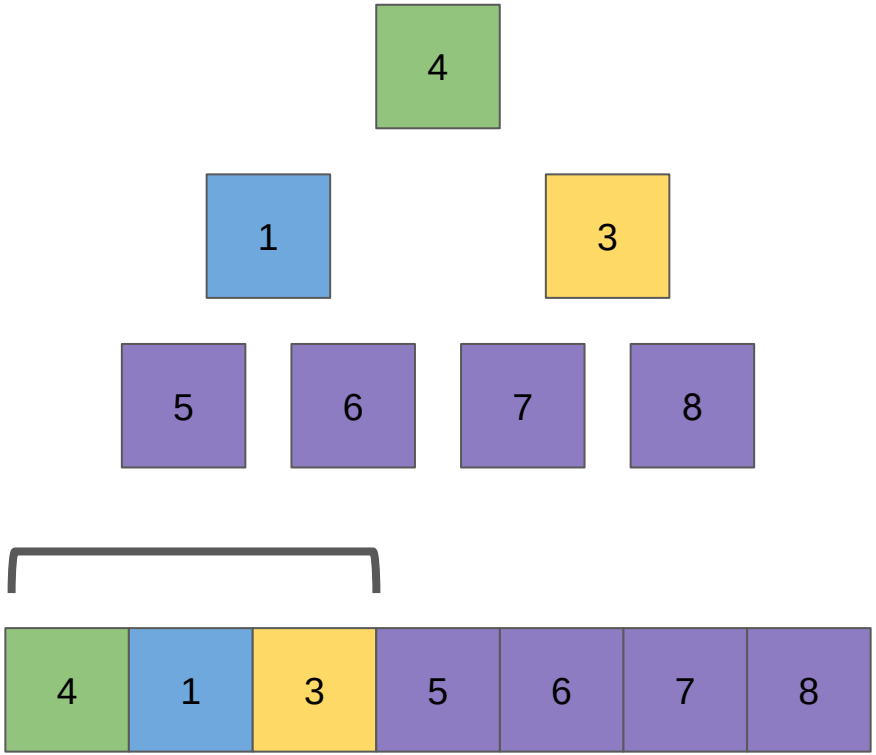
Heapsort



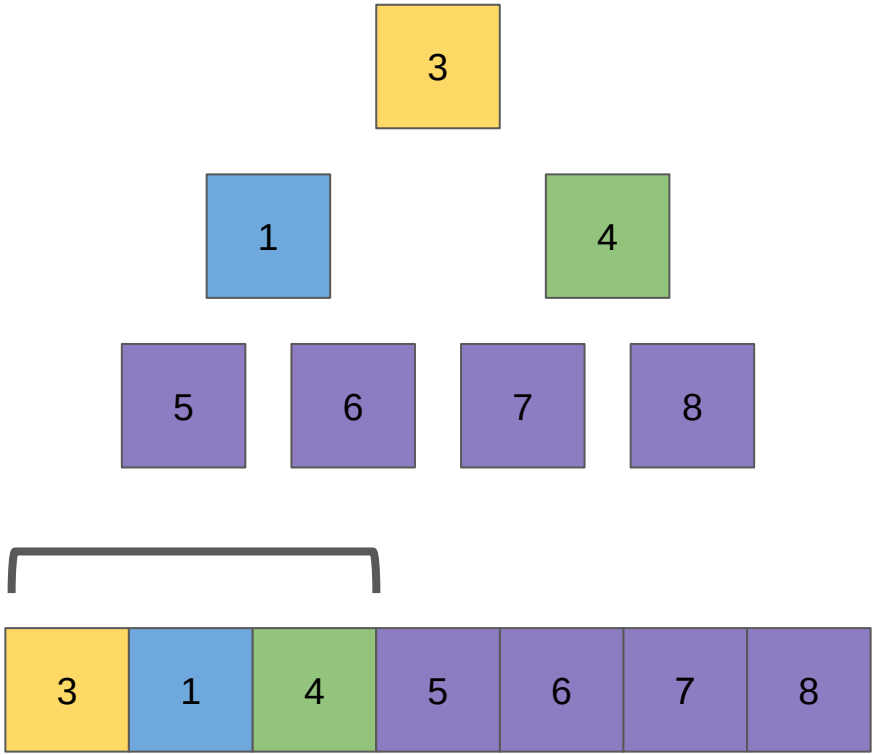
Heapsort



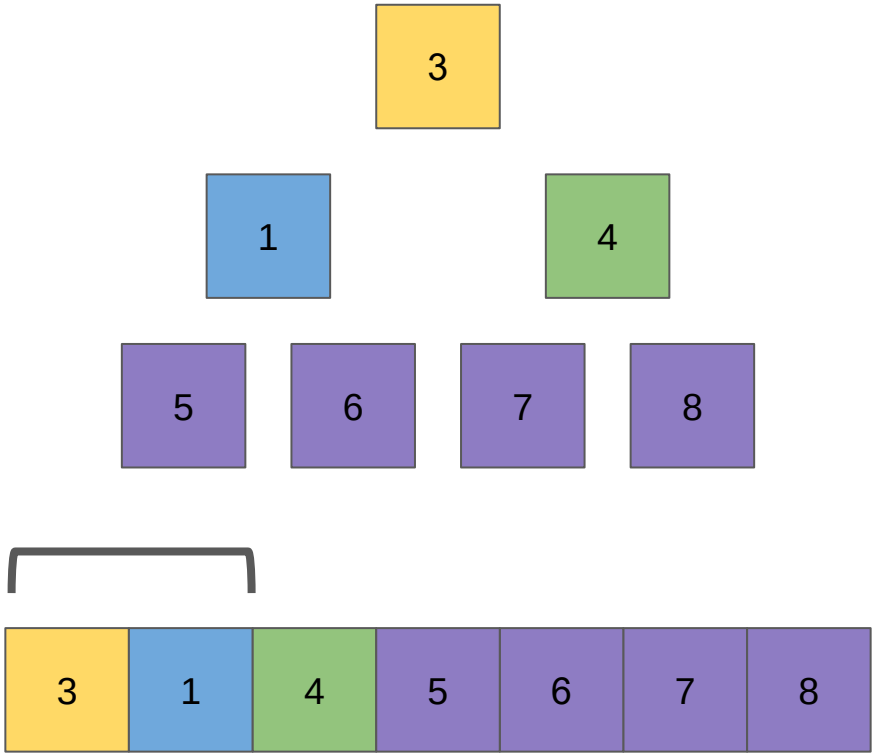
Heapsort



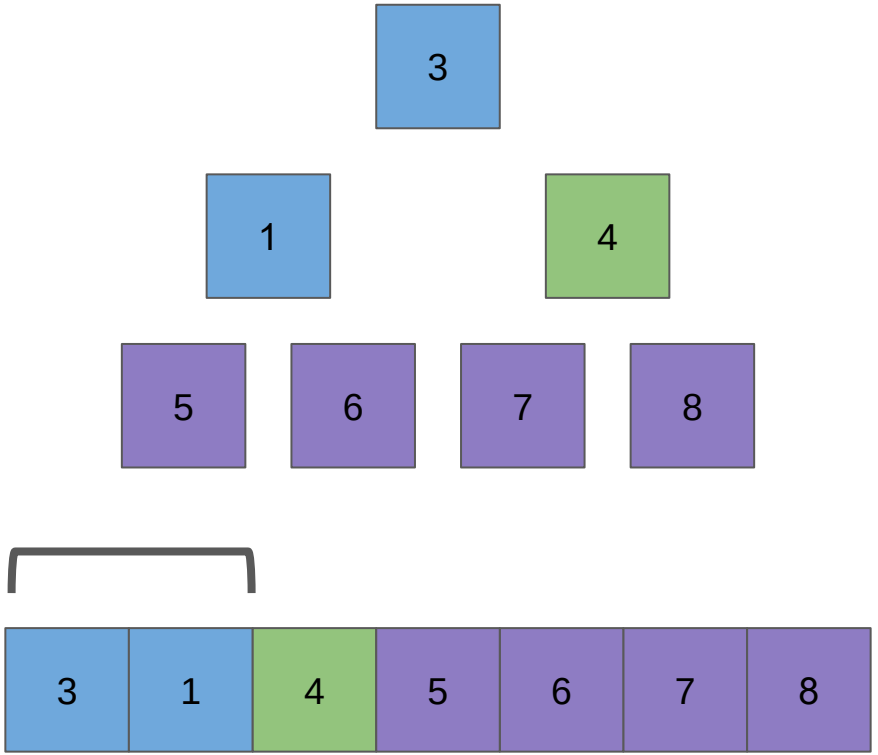
Heapsort



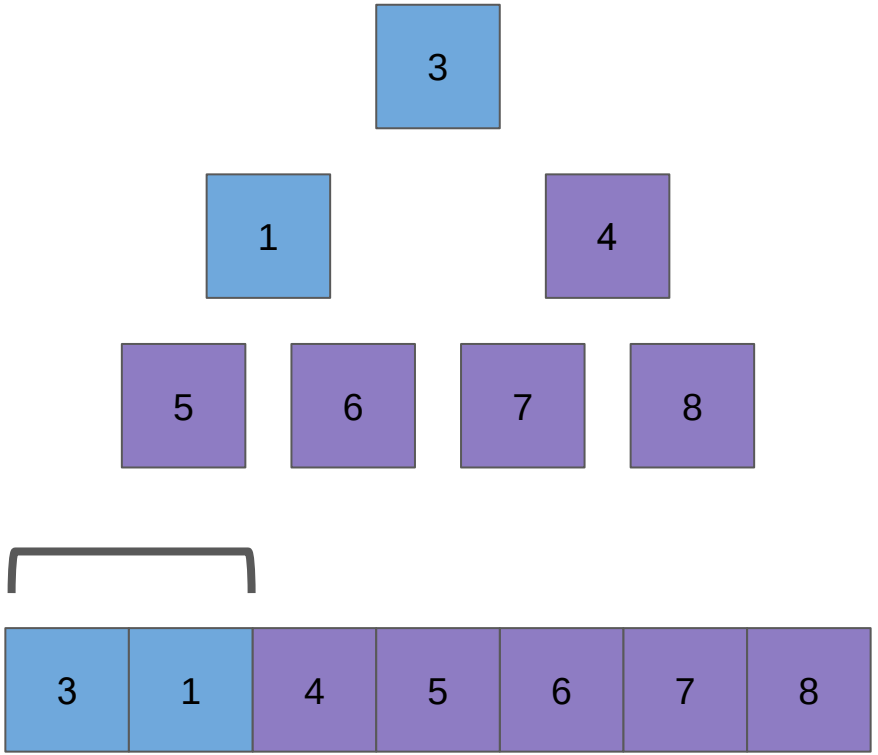
Heapsort



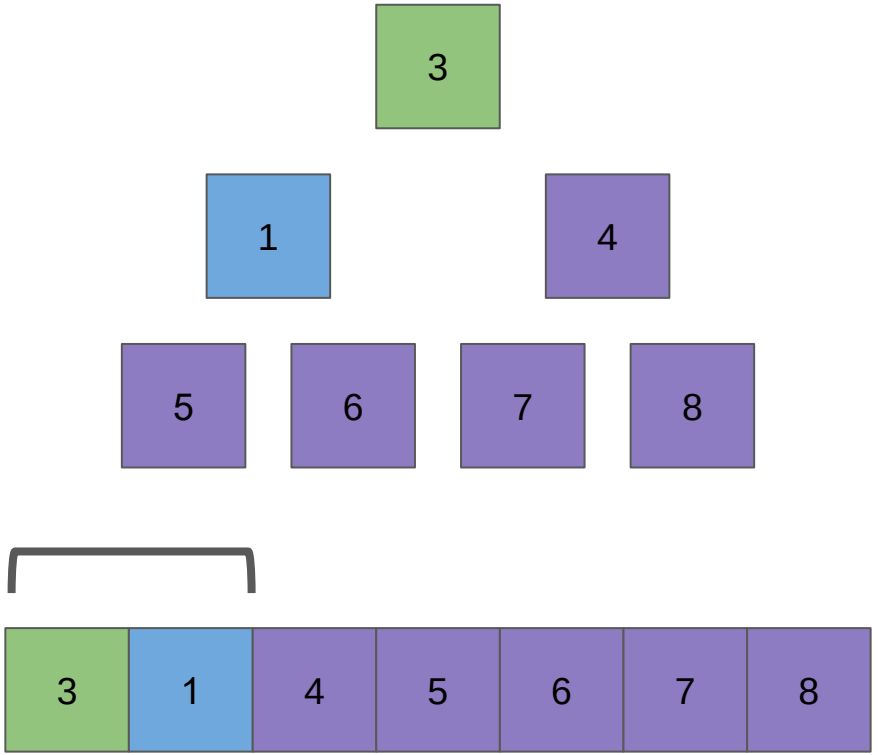
Heapsort



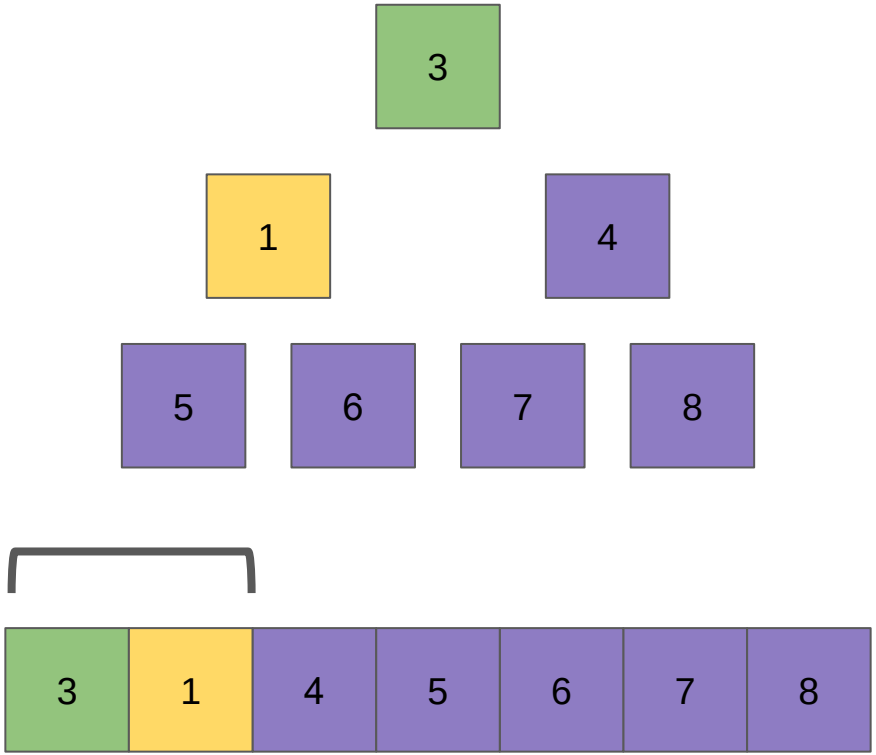
Heapsort



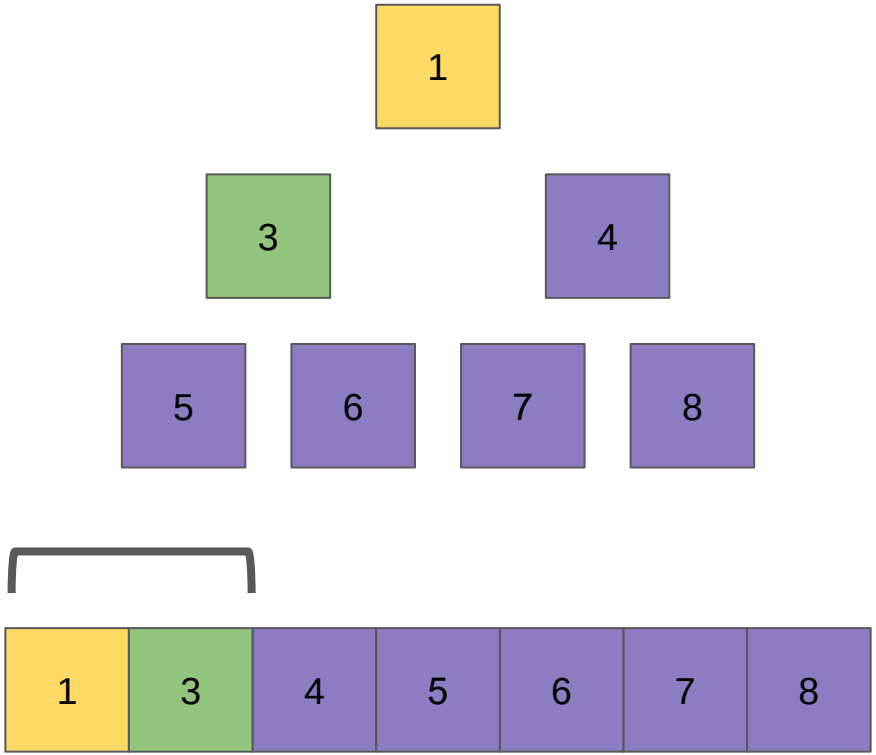
Heapsort



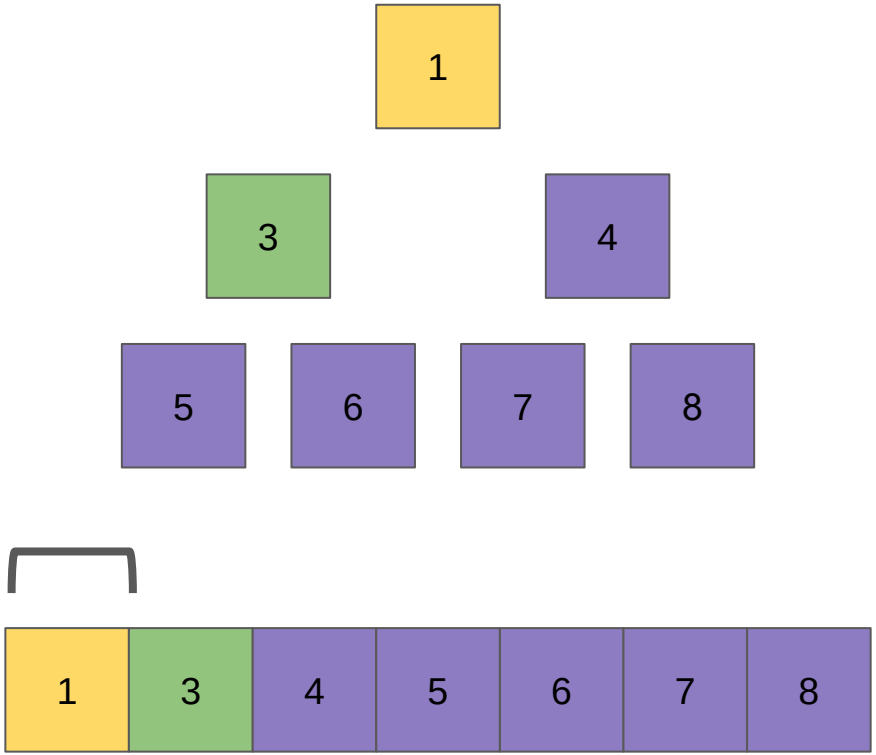
Heapsort



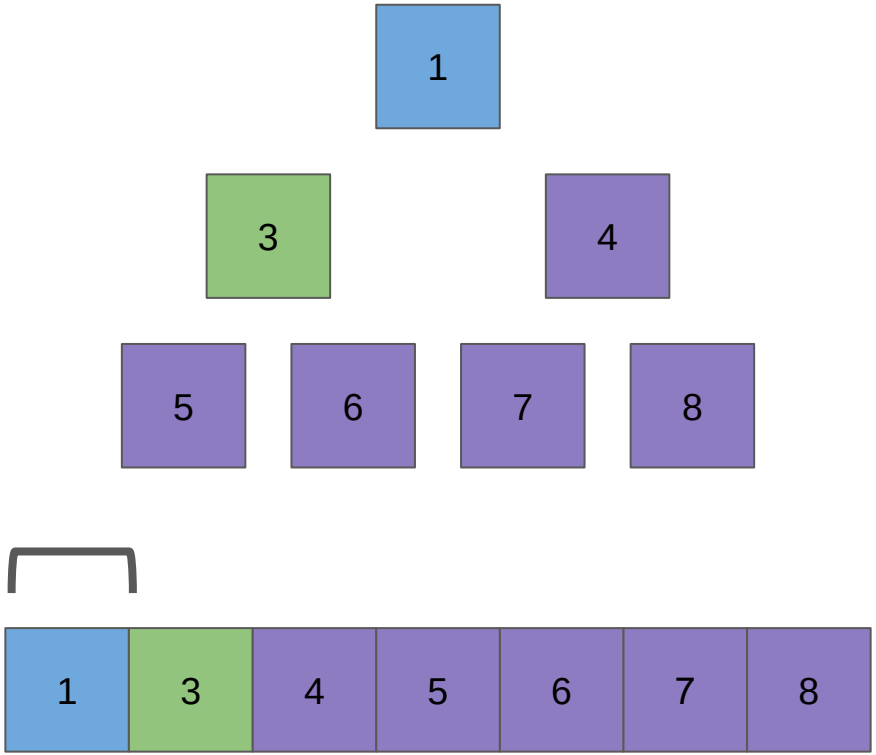
Heapsort



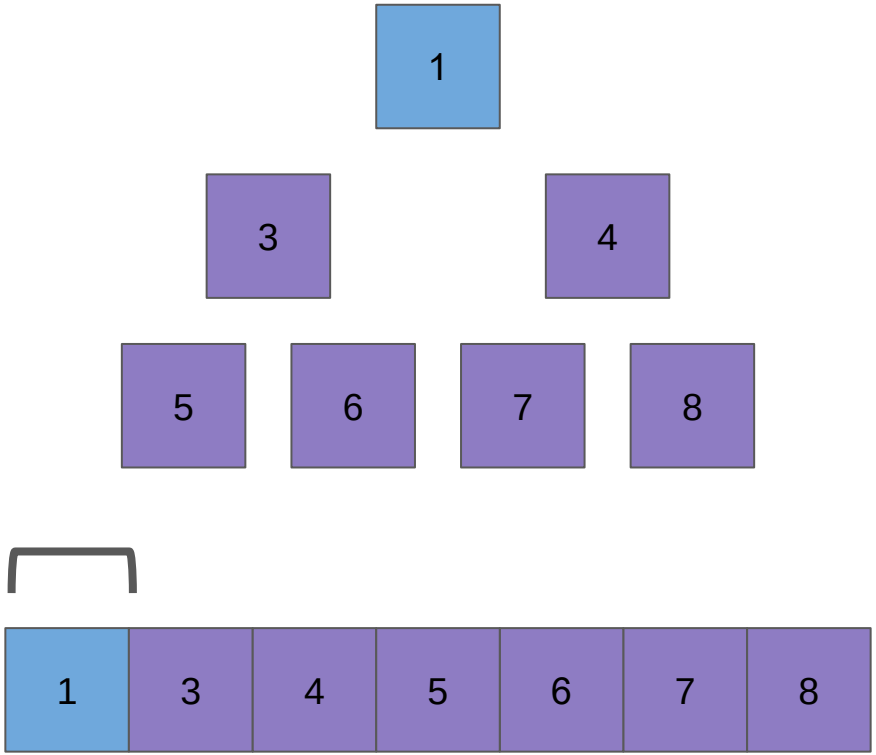
Heapsort



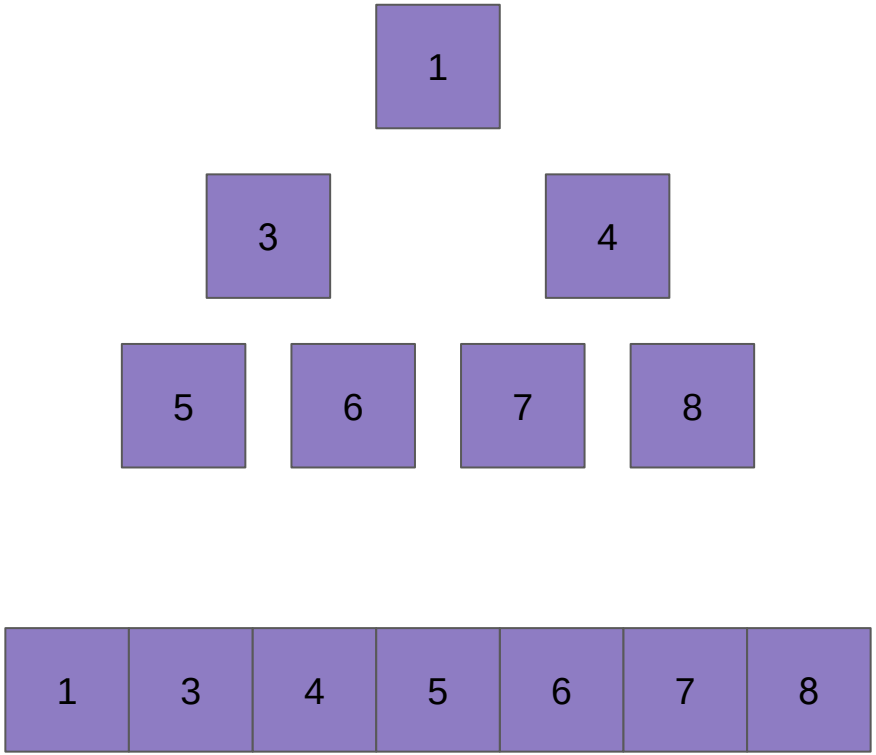
Heapsort



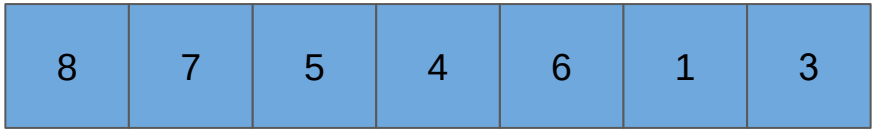
Heapsort



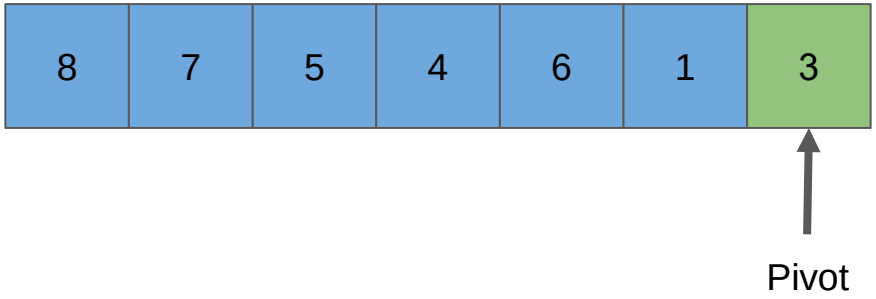
Heapsort



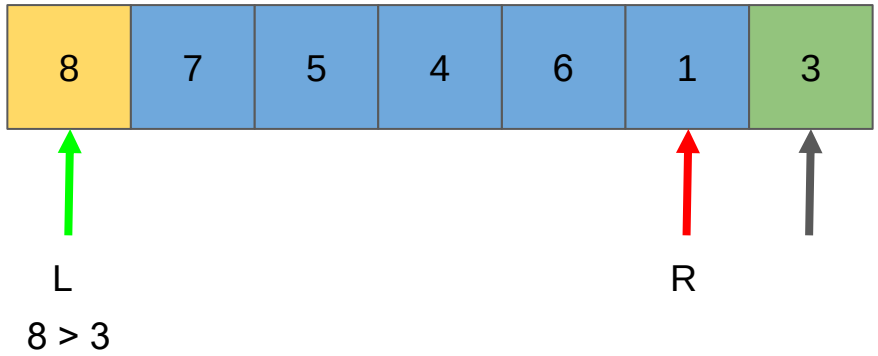
Quicksort



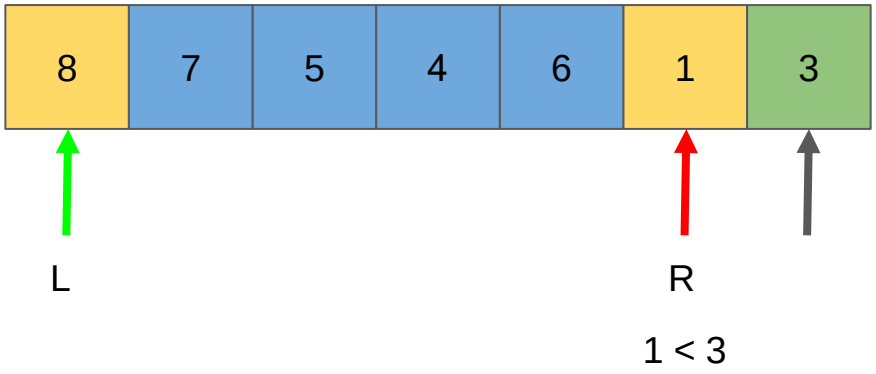
Quicksort



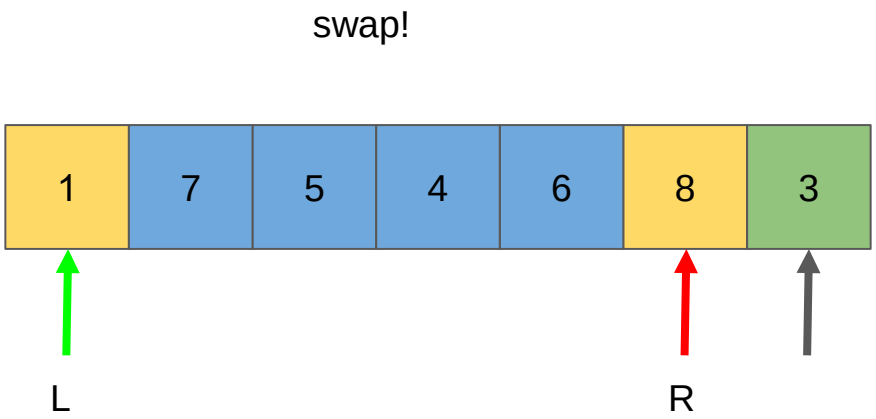
Quicksort



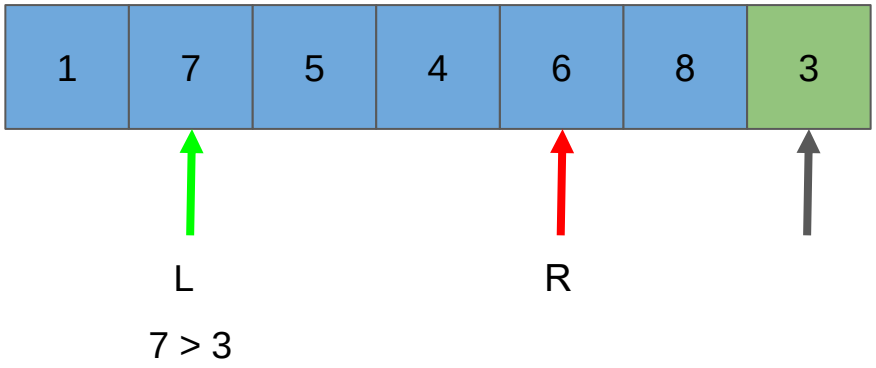
Quicksort



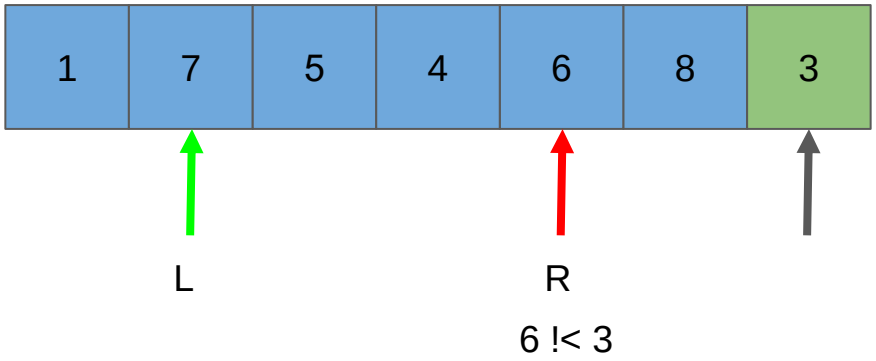
Quicksort



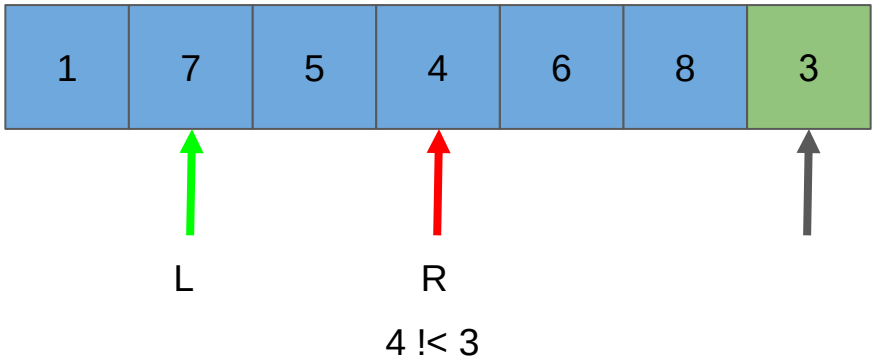
Quicksort



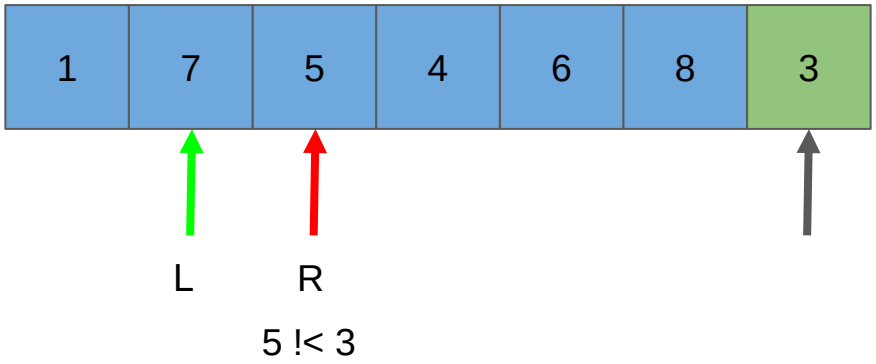
Quicksort



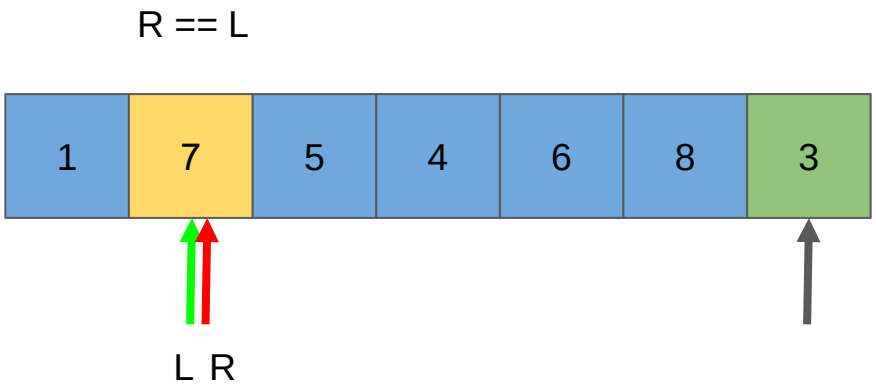
Quicksort



Quicksort

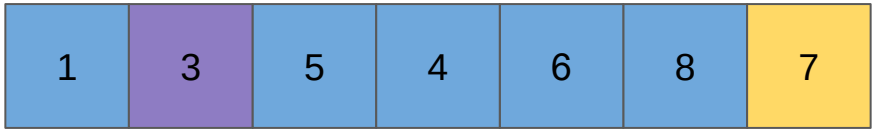


Quicksort



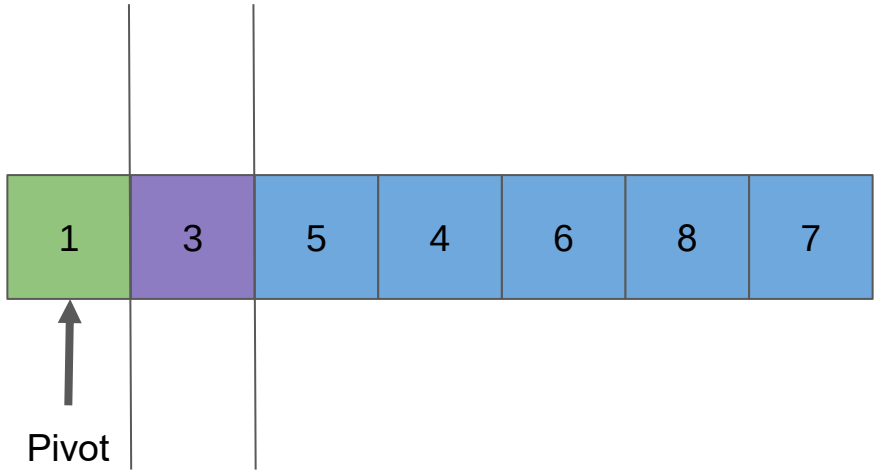
Quicksort

swap!



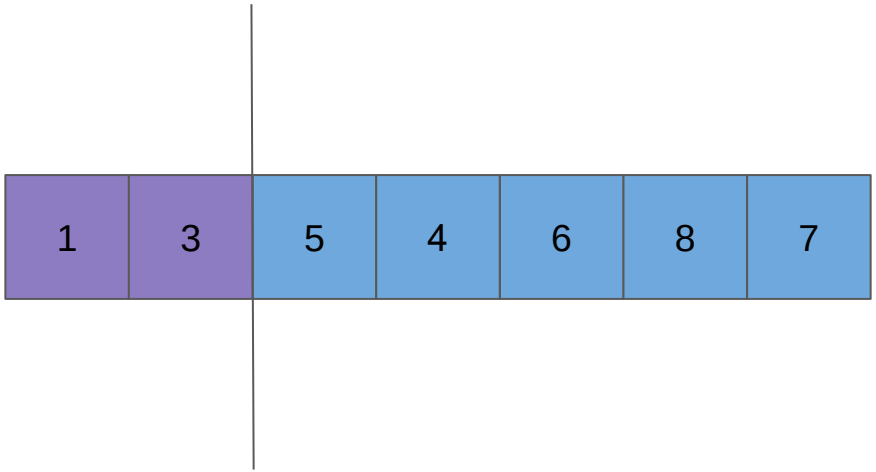
Quicksort

Now quicksort the left!

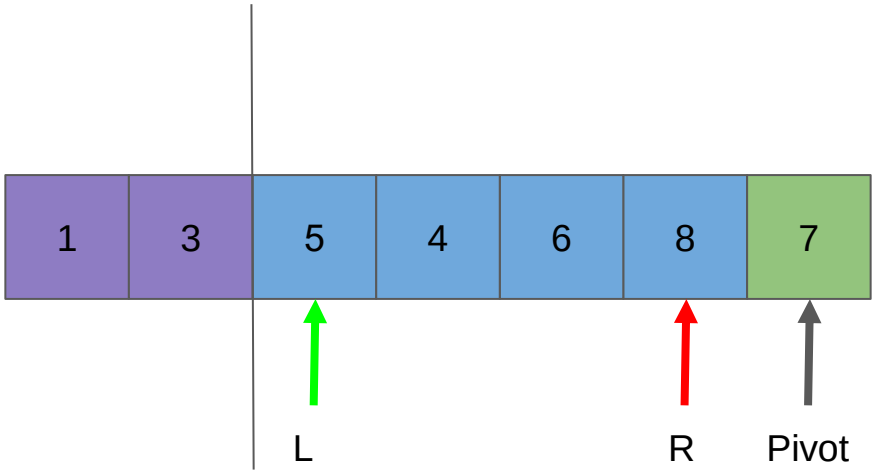


Quicksort

Length of sublist == 1
so it’s already sorted

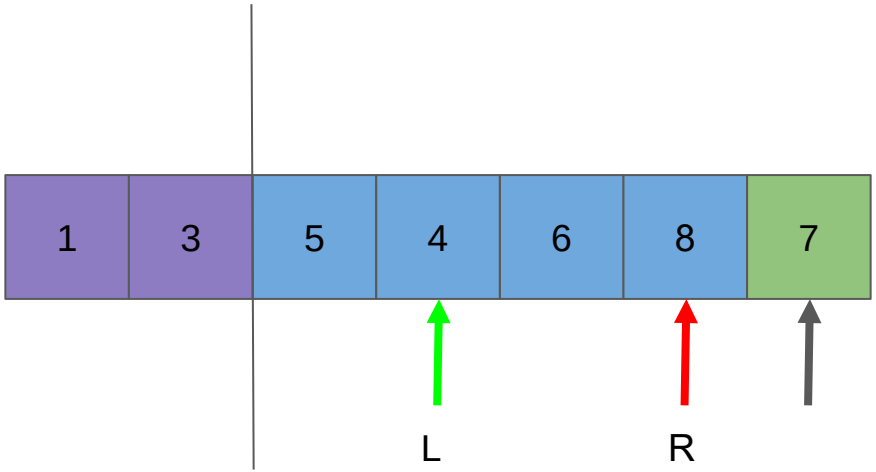


Quicksort



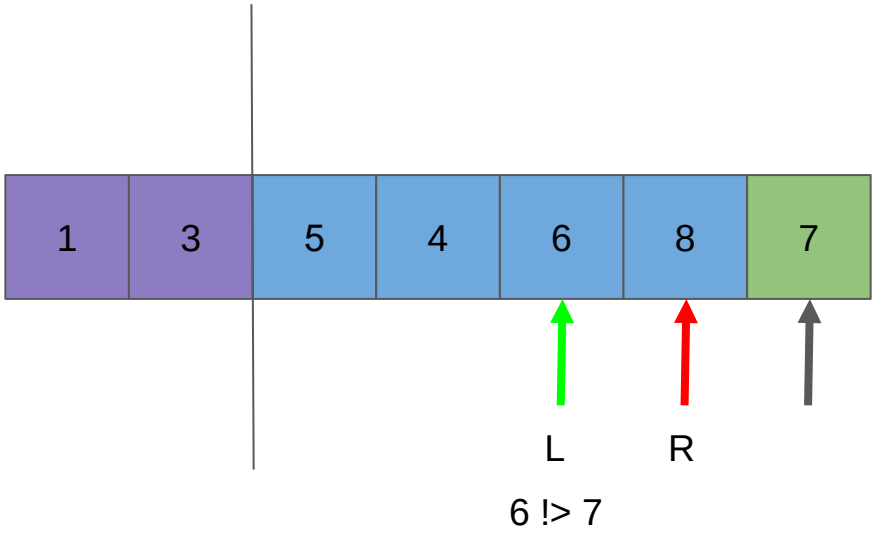
Now we sort the right side

Quicksort

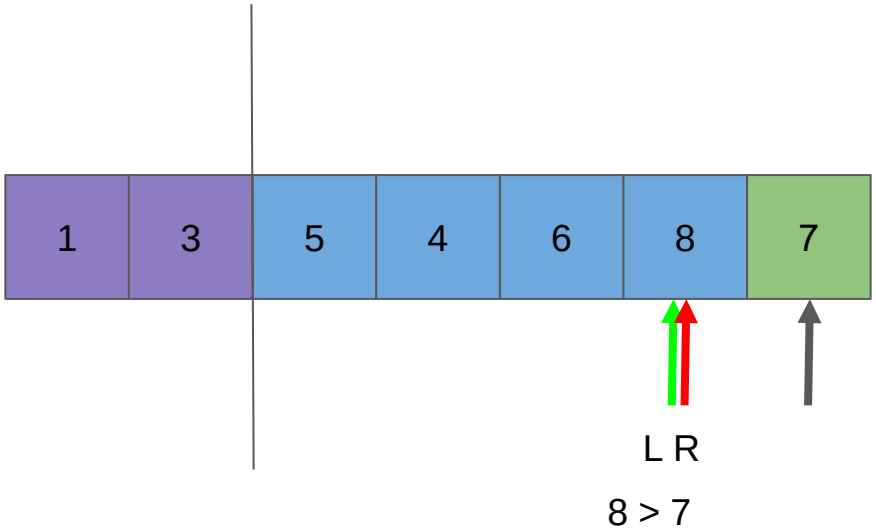


4 !> 7

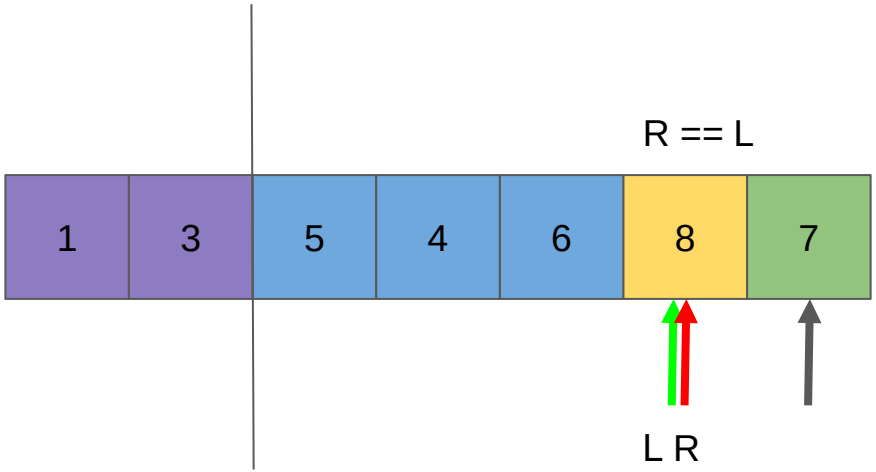
Quicksort



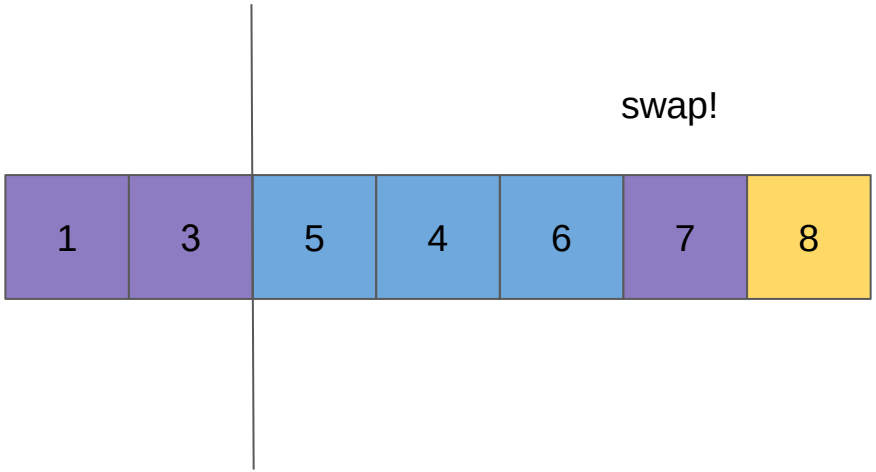
Quicksort



Quicksort

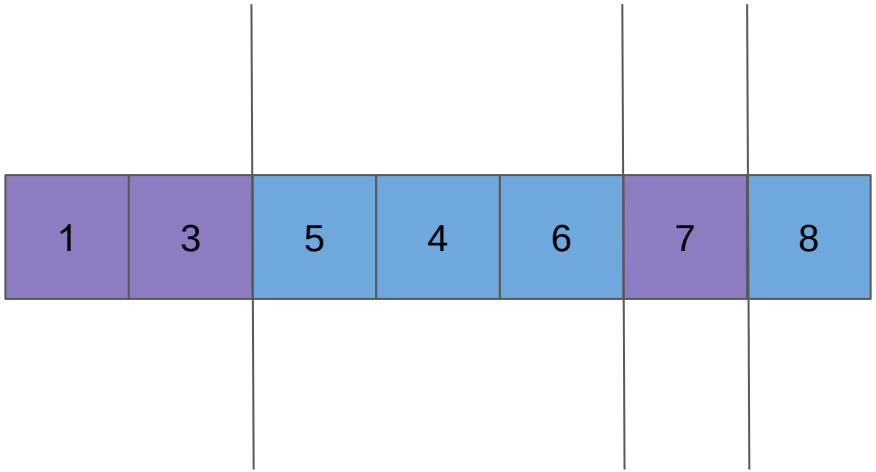


Quicksort

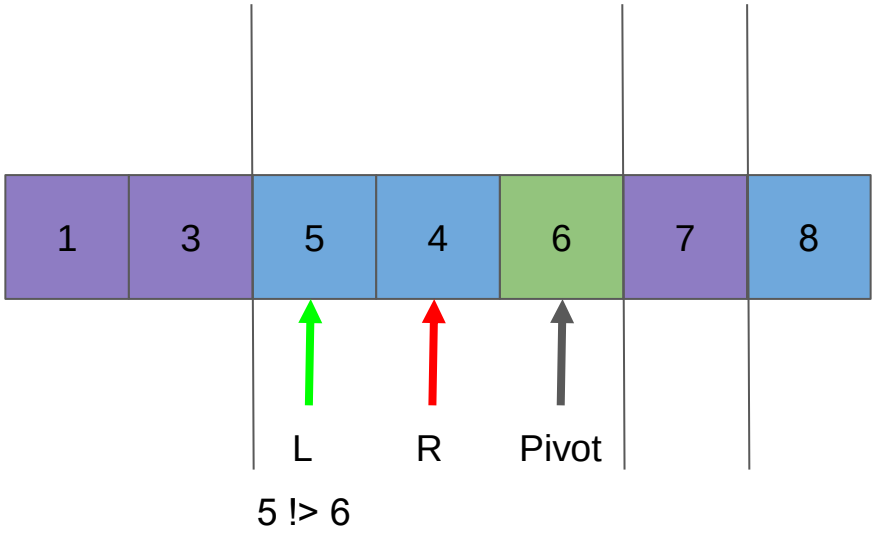


Quicksort

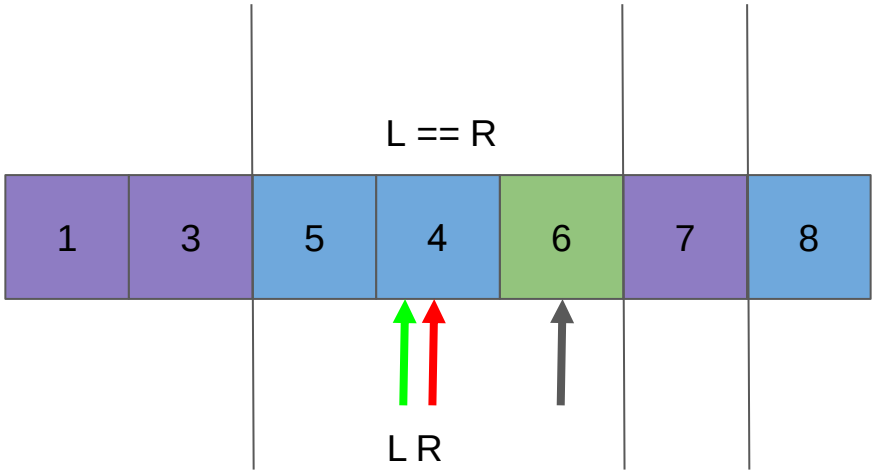
Now we sort the left sublist (of the original right sublist)



Quicksort

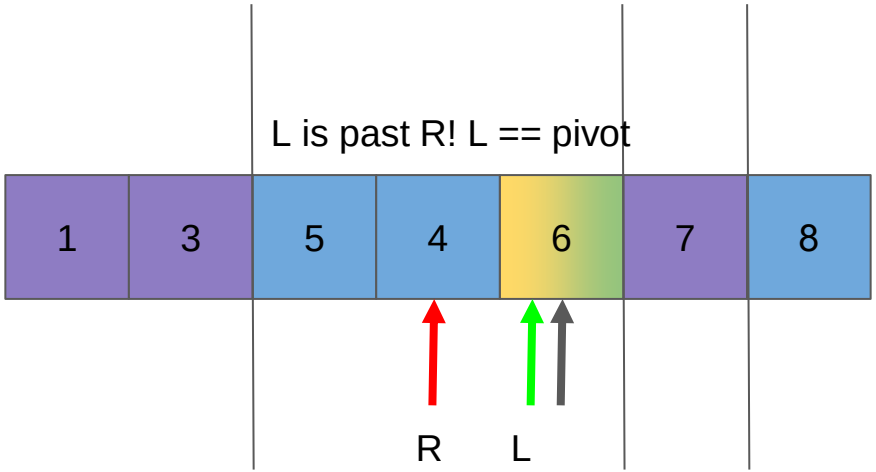


Quicksort

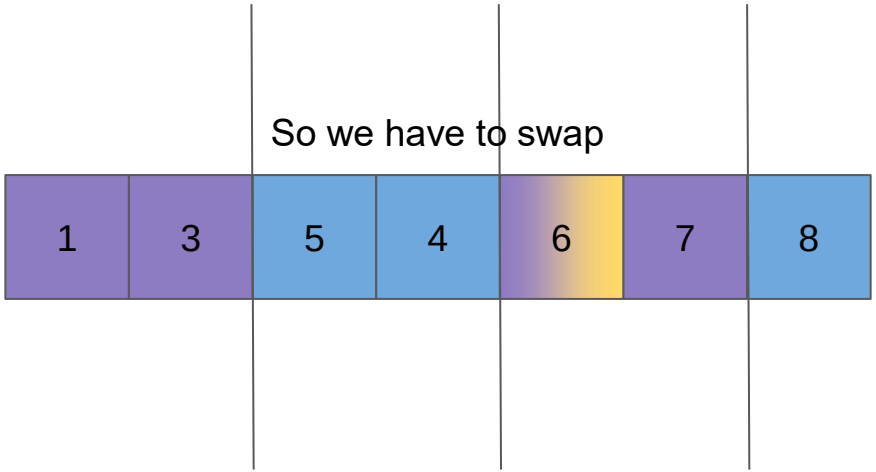


…but 4 !> 6
And R has
never moved

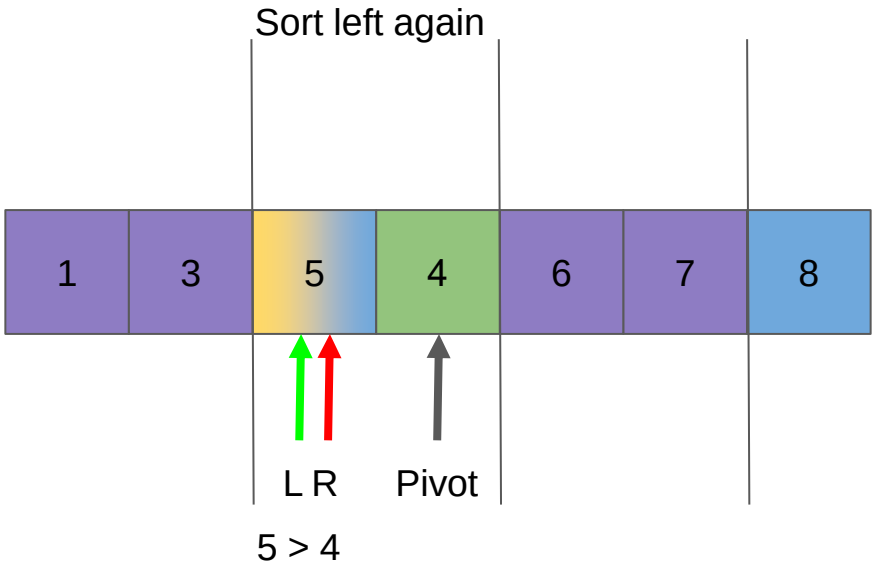
Quicksort



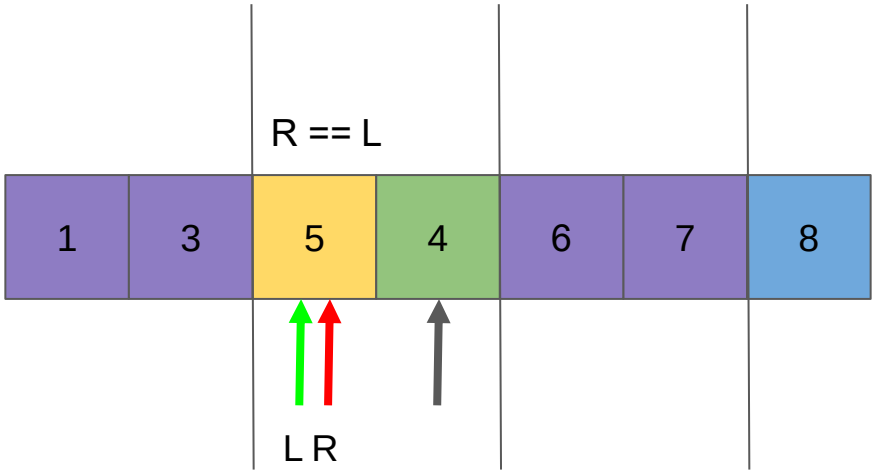
Quicksort



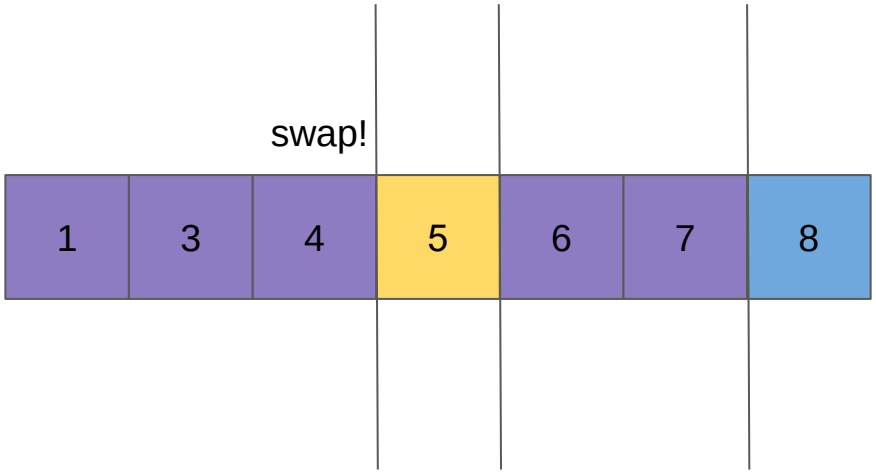
Quicksort



Quicksort

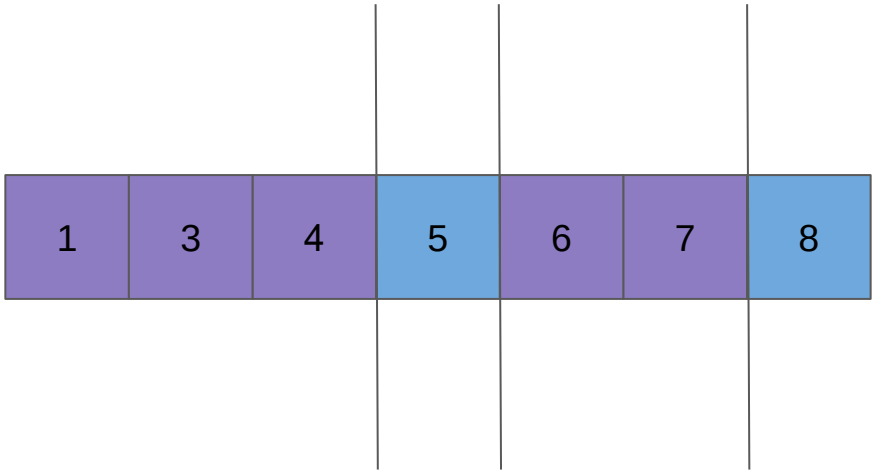


Quicksort

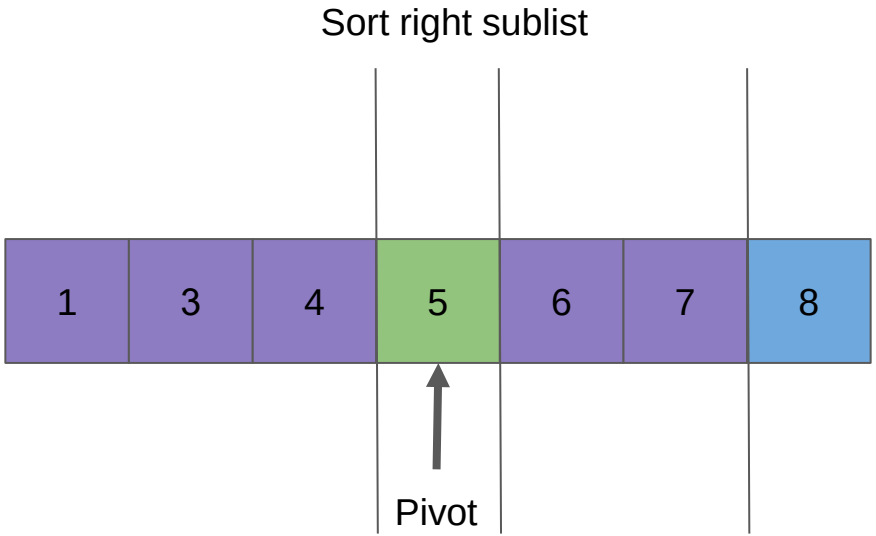


Quicksort

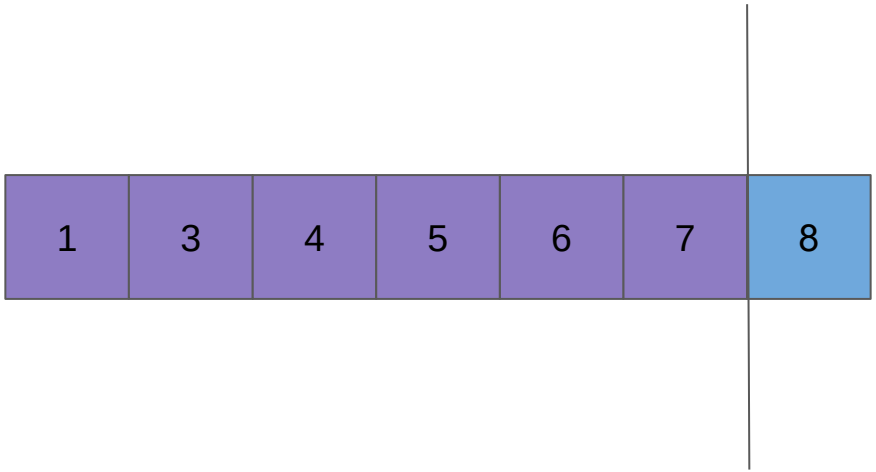
No sublist left of 4, so we go right



Quicksort

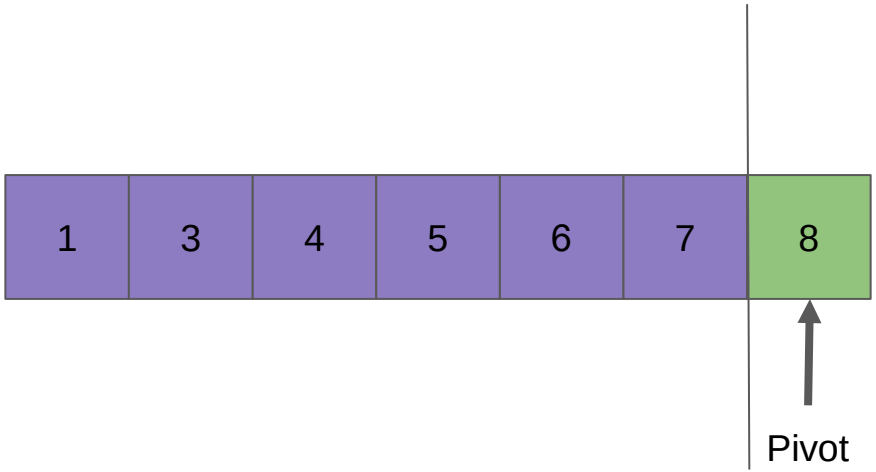


Quicksort



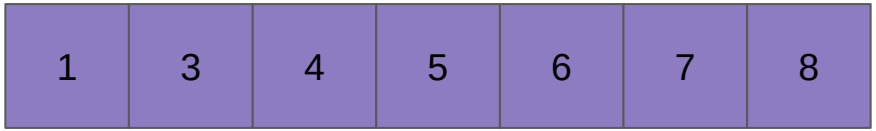
Length of sublist == 1
so it’s already sorted

Quicksort



Finally, sort the right sublist
(of the original right sublist
after the first partition)

Quicksort



Length of sublist == 1
so it’s already sorted