CS11313 - **Spring** 2022

Design & Analysis of Algorithms

Heapsort

Ibrahim Albluwi

Selection Sort?

```
SELECTION-SORT(array)

for i=n-1 ⟶ 1:
    max = FIND-MAX(array, i, 0)
    swap(array[i], array[max])
```

scan this portion of the array *linearly*.

Selection Sort?

```
SELECTION-SORT(array)

for i=n-1 ⟶ 1:
    max = FIND-MAX(array, i, 0)
    swap(array[i], array[max])
```

scan this portion of the array *linearly*.

```
HEAP-SORT(array)

prepare(array)

for i=n-1 ⟶ 1:
    max = FIND-MAX(array, i, 0)
    insert array[max] into array[i])
```

rearrange the elements so that finding the max is easy!

find the maximum element quickly!

Selection Sort?

```
SELECTION-SORT(array)

for i=n-1 ⟶ 1:
    max = FIND-MAX(array, i, 0)
    swap(array[i], array[max])
```

scan this portion of the array *linearly*.

```
HEAP-SORT(array)

prepare(array)

for i=n-1 ⟶ 1:
    max = FIND-MAX(array, i, 0)
    insert array[max] into array[i])
```

rearrange the elements so that finding the max is easy!

find the maximum element quickly!

Roadmap.

1. Review Max-Priority Queues and Heaps.
2. Learn about Heapsort.

Max-Priority Queue (Abstract Data Type)

Abstract Data Type (ADT): A specification of the possible operations on a set of values (independent of the implementation).

Examples.

ADT	Goal	operations
Stack	Remove the item most-recently added	PUSH, POP
Queue	Remove the item least-recently added	ENQUEUE, DEQUEUE

Max-Priority Queue (Abstract Data Type)

Abstract Data Type (ADT): A specification of the possible operations on a set of values (independent of the implementation).

Examples.

ADT	Goal	operations	possible implementations
Stack	Remove the item most-recently added	PUSH, POP	Singly-Linked List Doubly-Linked List Array-List
Queue	Remove the item least-recently added	ENQUEUE, DEQUEUE	

Max-Priority Queue (Abstract Data Type)

Abstract Data Type (ADT): A specification of the possible operations on a set of values (independent of the implementation).

Examples.

ADT	Goal	operations	possible implementations
Stack	Remove the item most-recently added	PUSH, POP	Singly-Linked List Doubly-Linked List Array-List
Queue	Remove the item least-recently added	ENQUEUE, DEQUEUE	
Set	Search in a group of unique items	INSERT, DELETE, CONTAINS	Binary Search Tree Hash Table Linked-List, Array-List

Max-Priority Queue (Abstract Data Type)

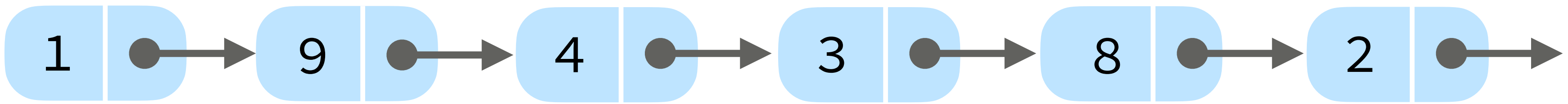
Abstract Data Type (ADT): A specification of the possible operations on a set of values (independent of the implementation).

Examples.

ADT	Goal	operations	possible implementations
Stack	Remove the item most-recently added	PUSH, POP	Singly-Linked List Doubly-Linked List Array-List
Queue	Remove the item least-recently added	ENQUEUE, DEQUEUE	
Set	Search in a group of unique items	INSERT, DELETE, CONTAINS	Binary Search Tree Hash Table Linked-List, Array-List
Max-PQ	**Remove the largest item**	INSERT, MAX, DEL-MAX	?

Max-Priority Queue (Abstract Data Type)

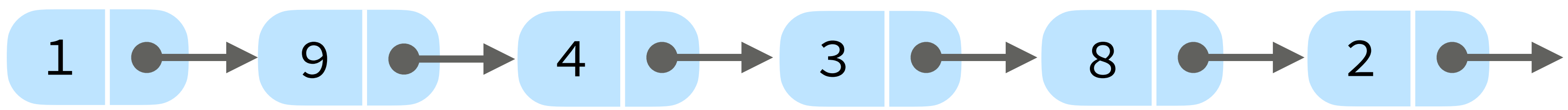
Unordered List:



Max-Priority Queue (Abstract Data Type)

Unordered List:

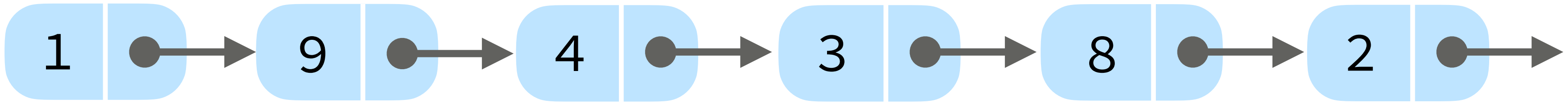
- `insert`: $\Theta(1)$ (insert to the end of the list; order does not matter)
- `max`: $\Theta(n)$ (linearly search for the max)
- `delMax`: $\Theta(n)$ (linearly search for the max and delete it)



Max-Priority Queue (Abstract Data Type)

Unordered List:

- `insert`: $\Theta(1)$ (insert to the end of the list; order does not matter)
- `max`: $\Theta(n)$ (linearly search for the max)
- `delMax`: $\Theta(n)$ (linearly search for the max and delete it)

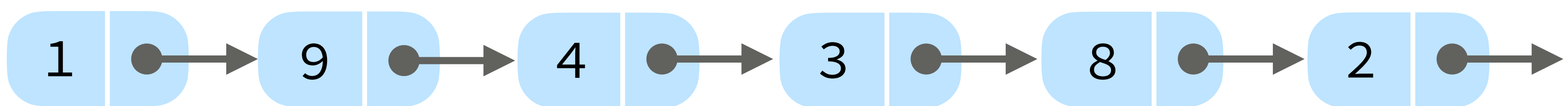


Ordered Array:

Max-Priority Queue (Abstract Data Type)

Unordered List:

- `insert`: $\Theta(1)$ (insert to the end of the list; order does not matter)
- `max`: $\Theta(n)$ (linearly search for the max)
- `delMax`: $\Theta(n)$ (linearly search for the max and delete it)



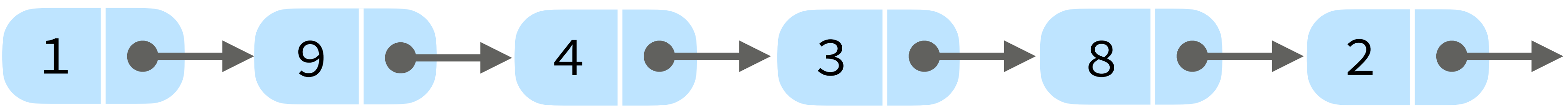
Ordered Array:

- `insert`: $O(n)$ (items need to be shifted based on where the)
- `max`: $\Theta(1)$ (max is always the last item in the array)
- `delMax`: $\Theta(1)$ (just decrement the size)

Max-Priority Queue (Abstract Data Type)

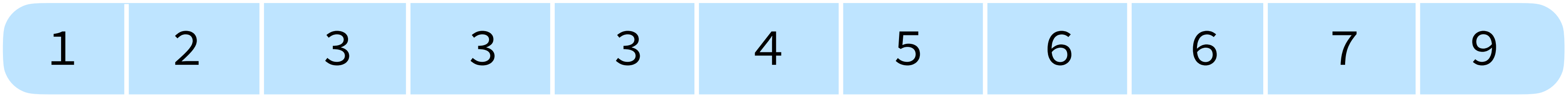
Unordered List:

- `insert`: $\Theta(1)$ (insert to the end of the list; order does not matter)
- `max`: $\Theta(n)$ (linearly search for the max)
- `delMax`: $\Theta(n)$ (linearly search for the max and delete it)



Ordered Array:

- `insert`: $O(n)$ (items need to be shifted based on where the)
- `max`: $\Theta(1)$ (max is always the last item in the array)
- `delMax`: $\Theta(1)$ (just decrement the size)



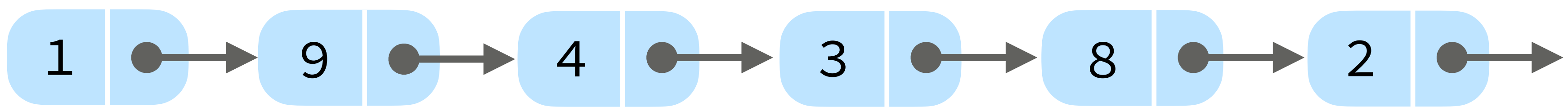
Binary Heap:

- `insert`: $O(\log n)$ (how?)
- `max`: $\Theta(1)$ (how?)
- `delMax`: $O(\log n)$ (how?)

Max-Priority Queue (Abstract Data Type)

Unordered List:

- `insert`: $\Theta(1)$ (insert to the end of the list; order does not matter)
- `max`: $\Theta(n)$ (linearly search for the max)
- `delMax`: $\Theta(n)$ (linearly search for the max and delete it)



Ordered Array:

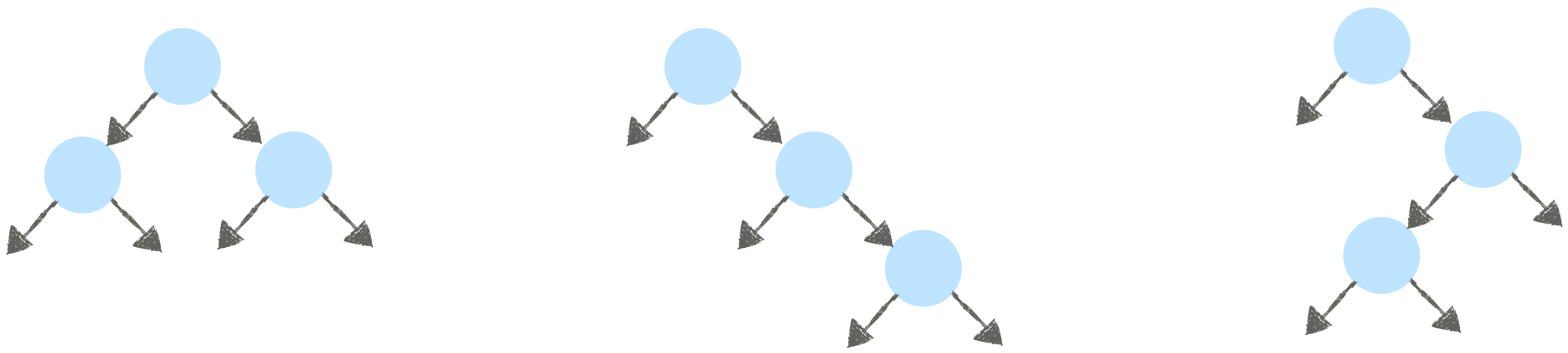
- `insert`: $O(n)$ (items need to be shifted based on where the)
- `max`: $\Theta(1)$ (max is always the last item in the array)
- `delMax`: $\Theta(1)$ (just decrement the size)

Binary Heap:

- `insert`: $O(\log n)$ (how?)
- `max`: $\Theta(1)$ (how?)
- `delMax`: $O(\log n)$ (how?)

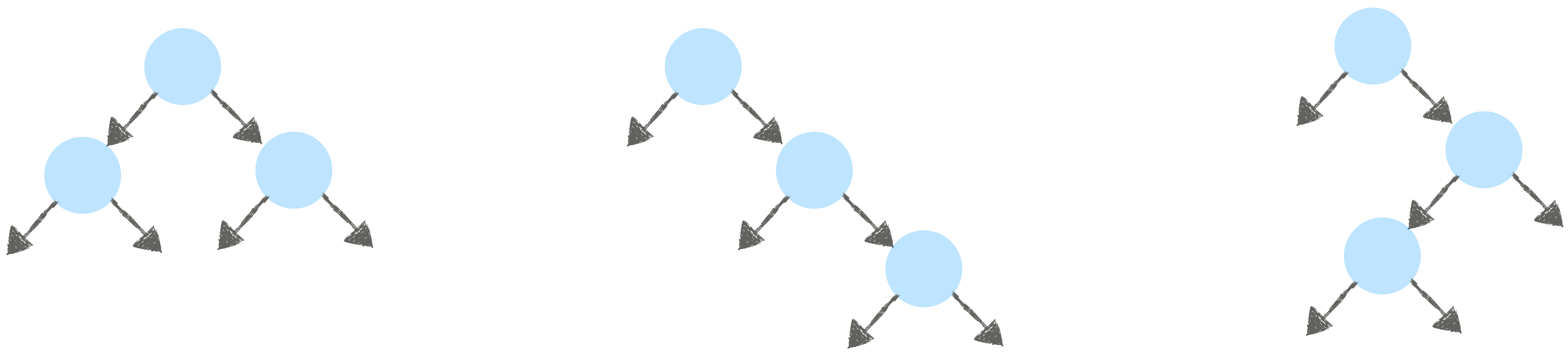
Binary Trees

Binary Tree: Empty or a node with links to left and right binary trees.



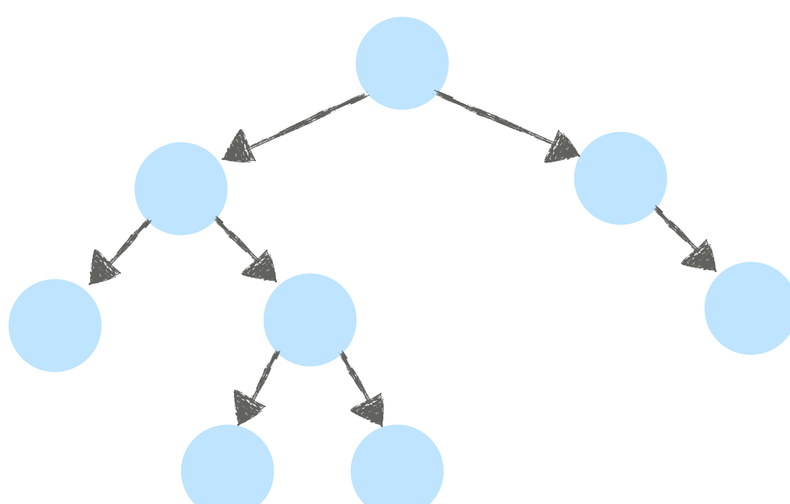
Binary Trees

Binary Tree: Empty or a node with links to left and right binary trees.

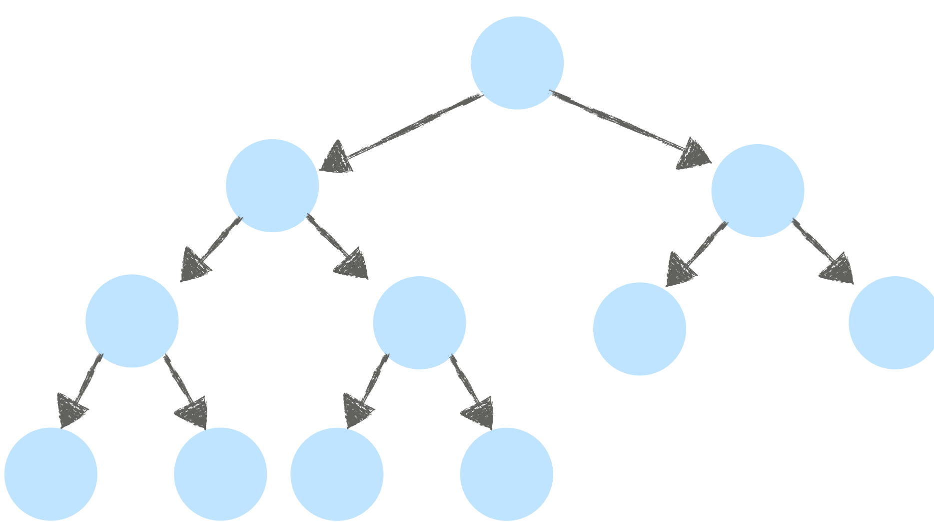


Complete Binary Tree:

- All levels are full (except possibly the last level).
- Last level is filled left-to-right.



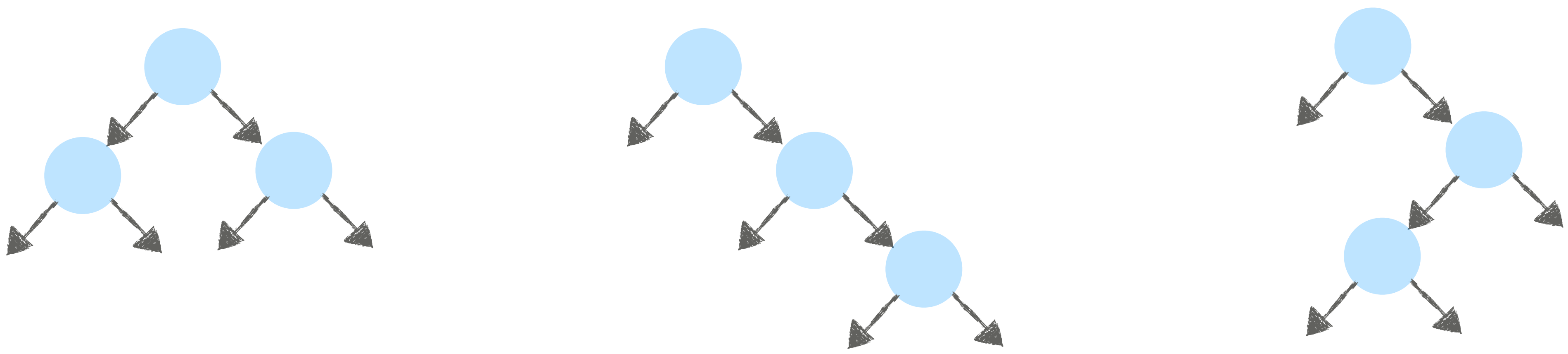
not complete



complete

Binary Trees

Binary Tree: Empty or a node with links to left and right binary trees.

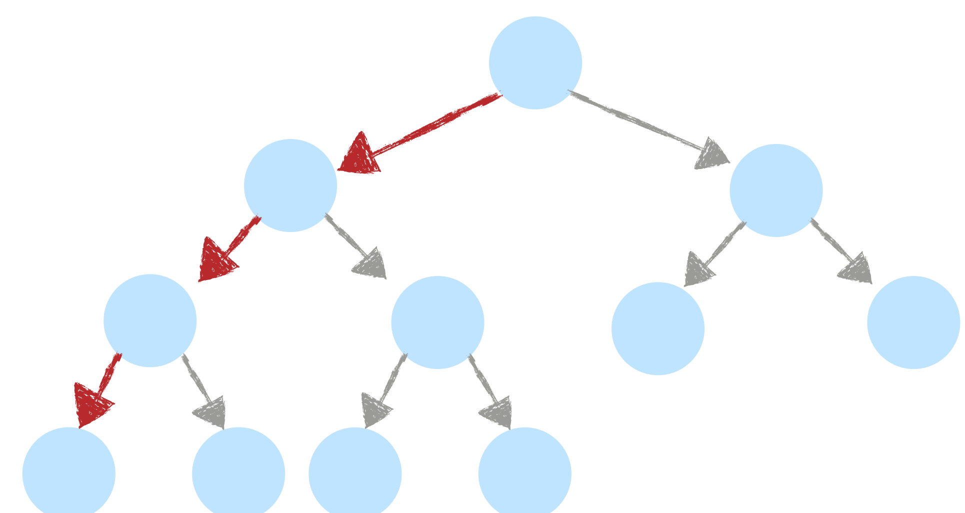


Complete Binary Tree:

- All levels are full (except possibly the last level).
- Last level is filled left-to-right.

Properties:

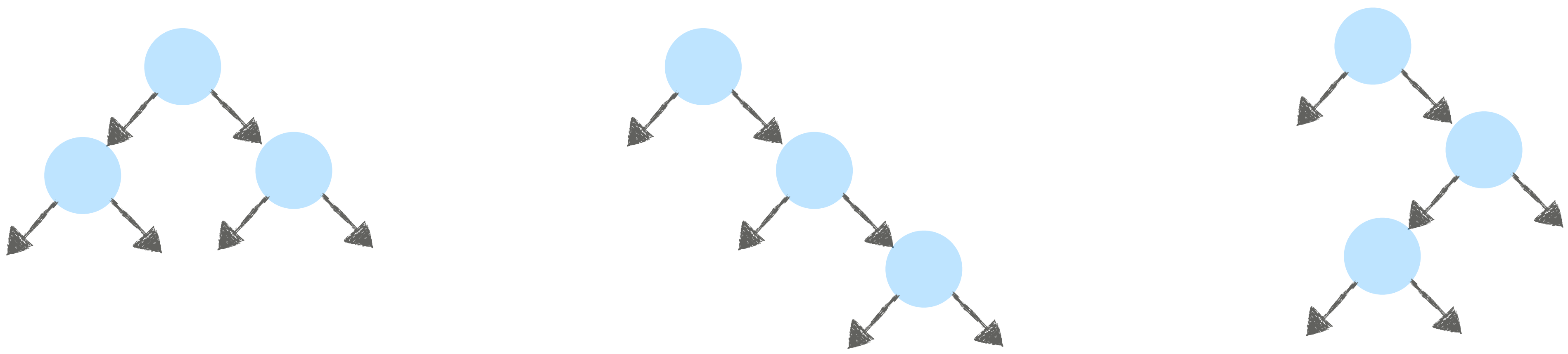
- Height if there are n nodes: $h = \lfloor \log_2 n \rfloor$



$$h = \lfloor \log_2 11 \rfloor = \lfloor 3.459 \rfloor = 3$$

Binary Trees

Binary Tree: Empty or a node with links to left and right binary trees.

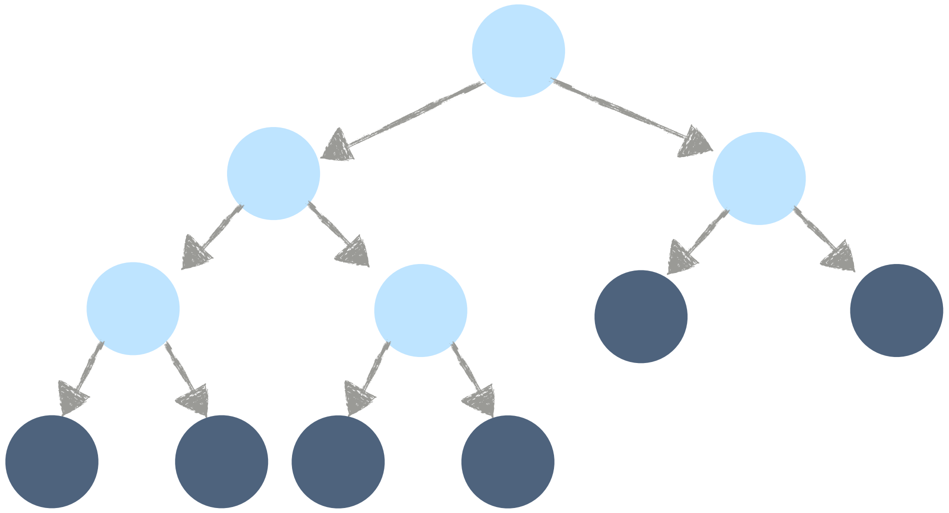


Complete Binary Tree:

- All levels are full (except possibly the last level).
- Last level is filled left-to-right.

Properties:

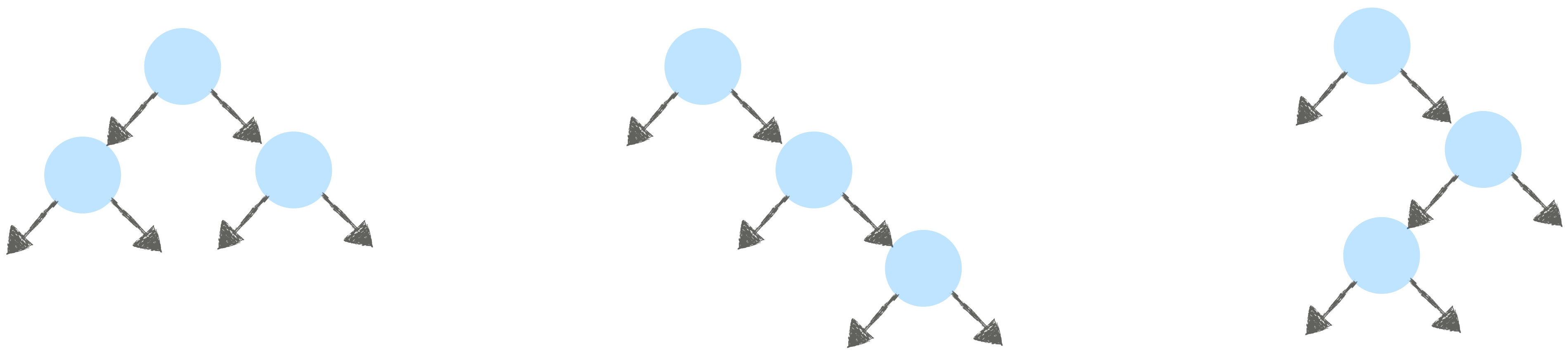
- Height if there are n nodes: $h = \lfloor \log_2 n \rfloor$
- There are $\left\lfloor \frac{n+1}{2} \right\rfloor$ leaves.



$$\left\lfloor \frac{11+1}{2} \right\rfloor = 6$$

Binary Trees

Binary Tree: Empty or a node with links to left and right binary trees.

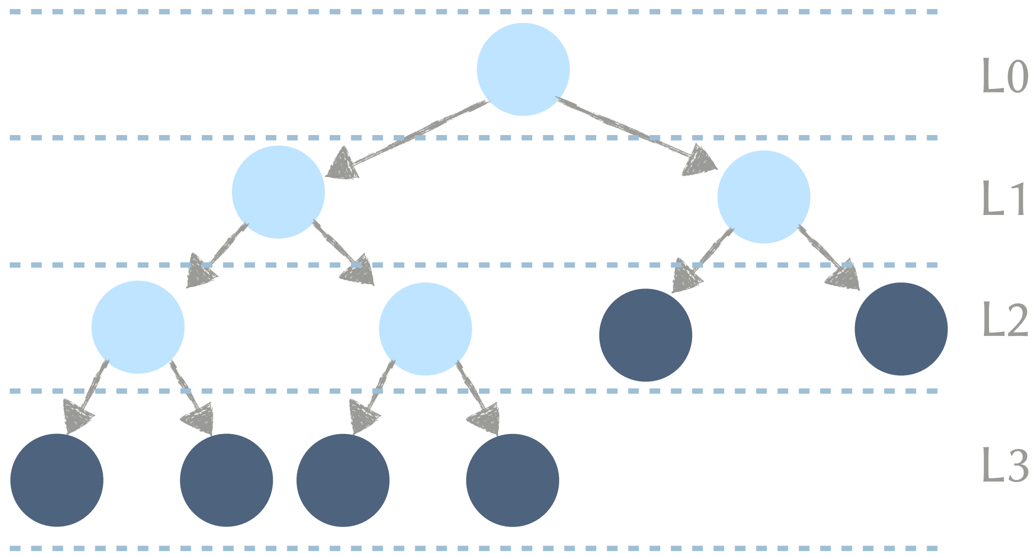


Complete Binary Tree:

- All levels are full (except possibly the last level).
- Last level is filled left-to-right.

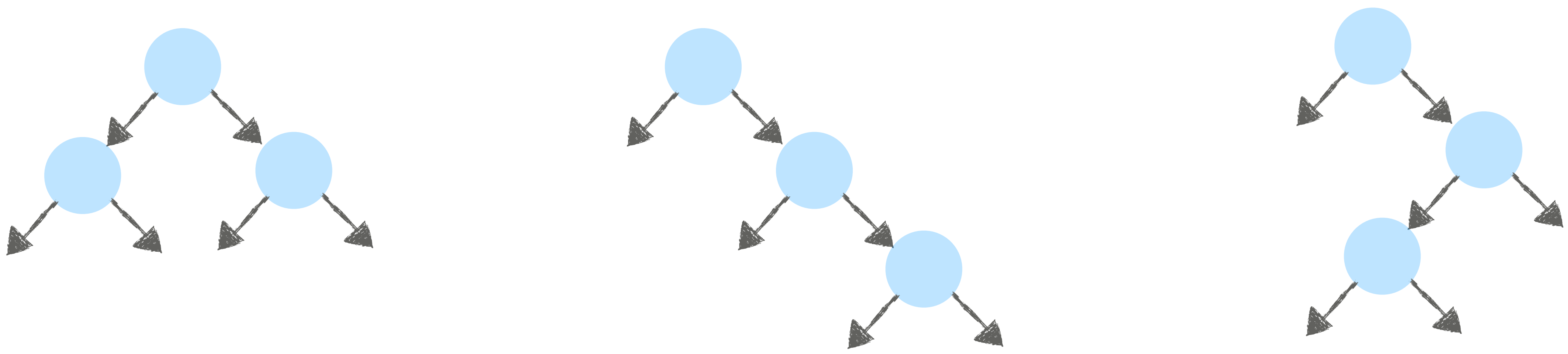
Properties:

- Height if there are n nodes: $h = \lfloor \log_2 n \rfloor$
- There are $\left\lfloor \frac{n+1}{2} \right\rfloor$ leaves.
- All leaves are at level h or $h - 1$.



Binary Trees

Binary Tree: Empty or a node with links to left and right binary trees.

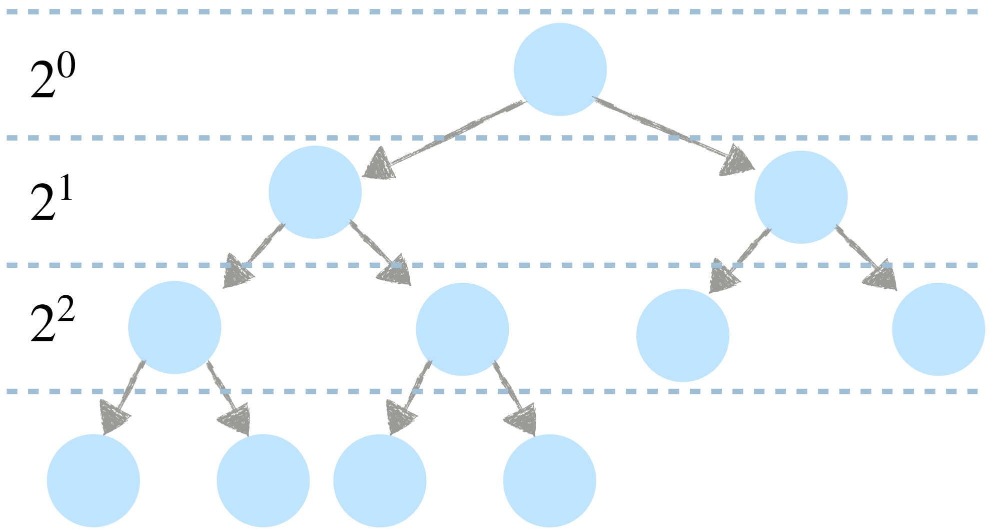


Complete Binary Tree:

- All levels are full (except possibly the last level).
- Last level is filled left-to-right.

Properties:

- Height if there are n nodes: $h = \lfloor \log_2 n \rfloor$
- There are $\left\lceil \frac{n+1}{2} \right\rceil$ leaves.
- All leaves are at level h or $h - 1$.
- Number of nodes at *internal* level $i = 2^i$



Binary Heaps (Tree Representation)

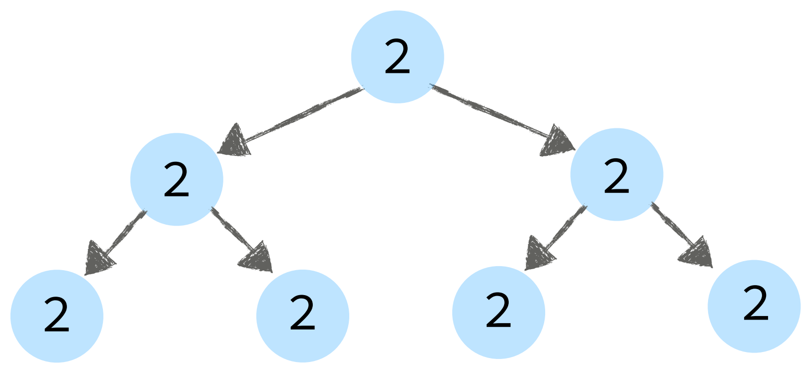
Binary Heap: (max-ordered)

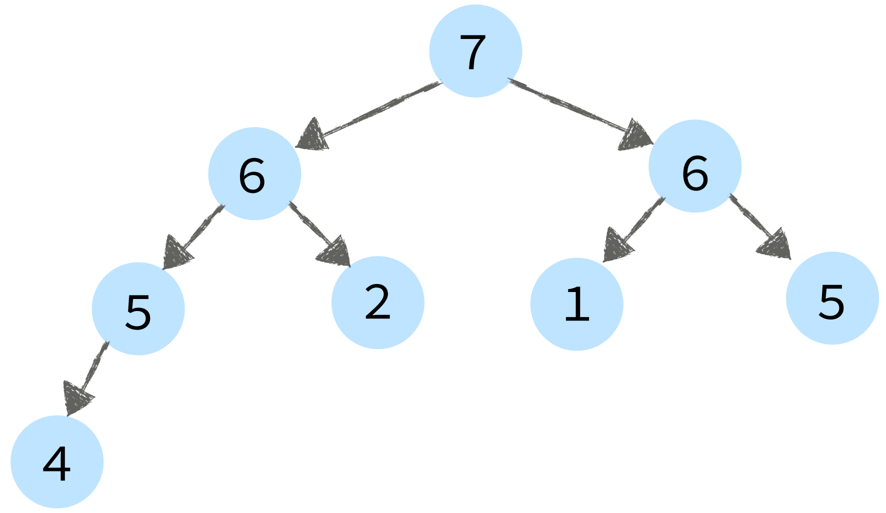
- Structure: Must be a complete binary tree.
- Order: Every node is not less than its children.

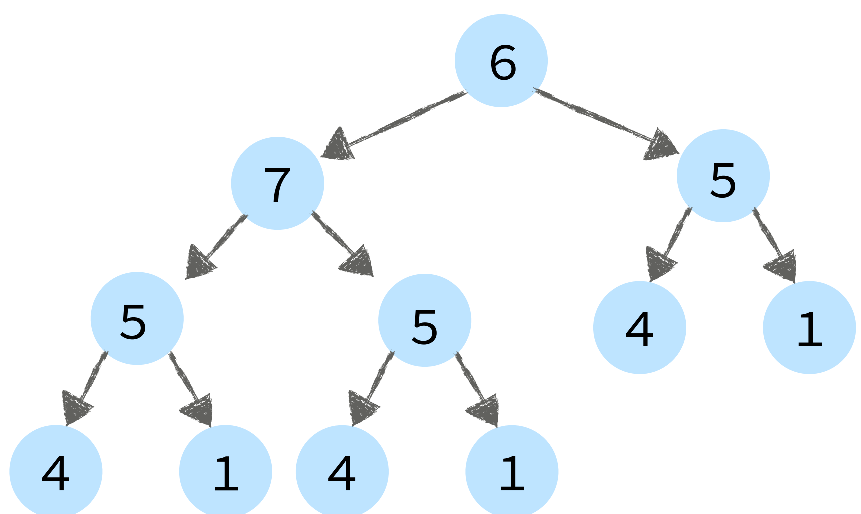
Binary Heaps (Tree Representation)

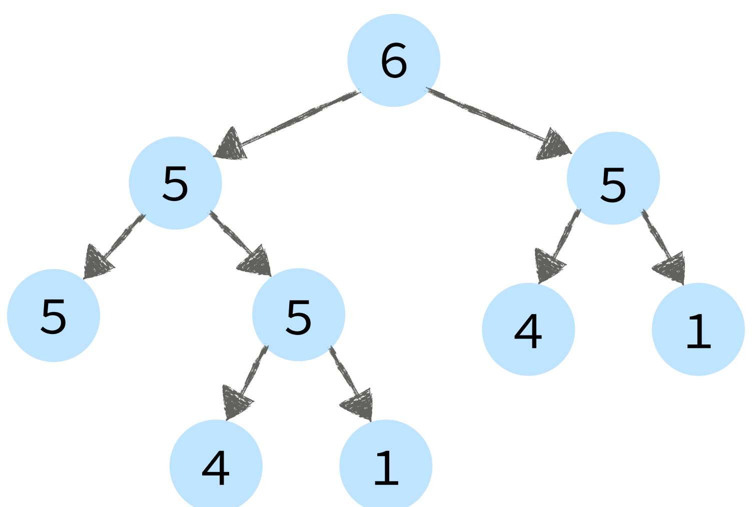
Binary Heap: (max-ordered)

- Structure: Must be a complete binary tree.
- Order: Every node is not less than its children.







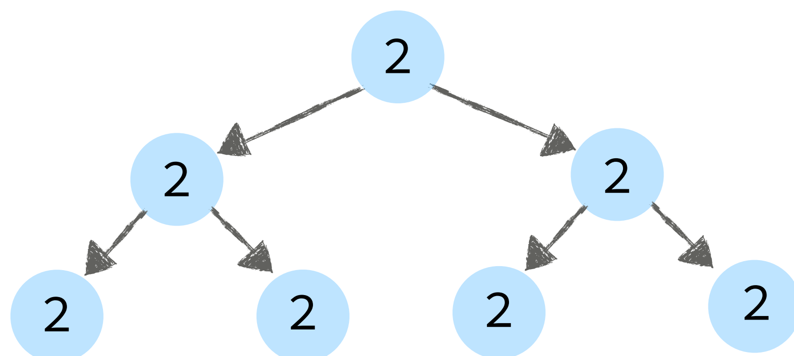


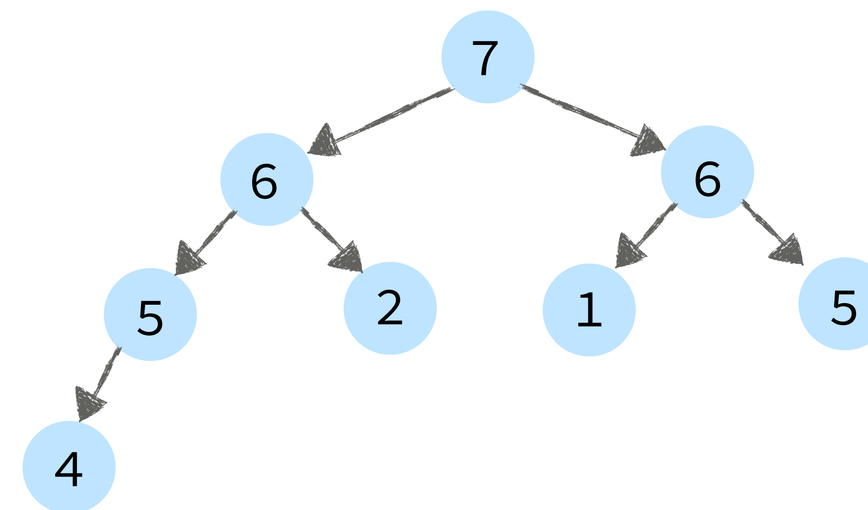
Binary Heaps (Tree Representation)

Binary Heap: (max-ordered)

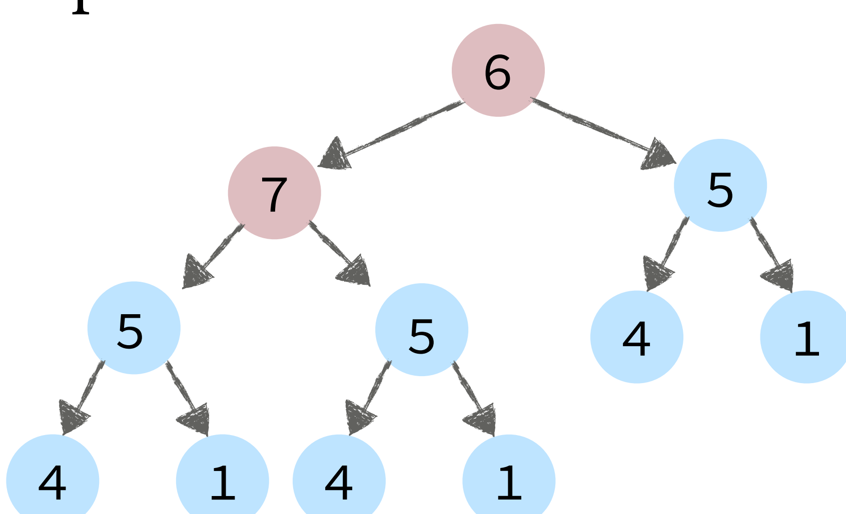
- Structure: Must be a complete binary tree.
- Order: Every node is not less than its children.

Examples:

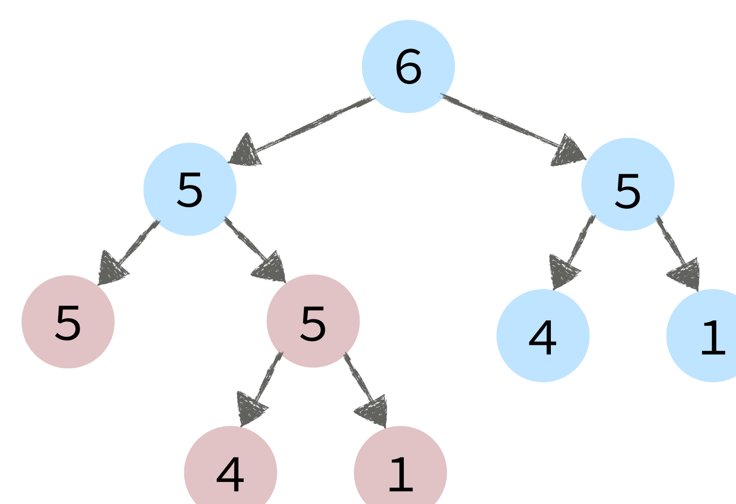




Non-Examples:



order property violated



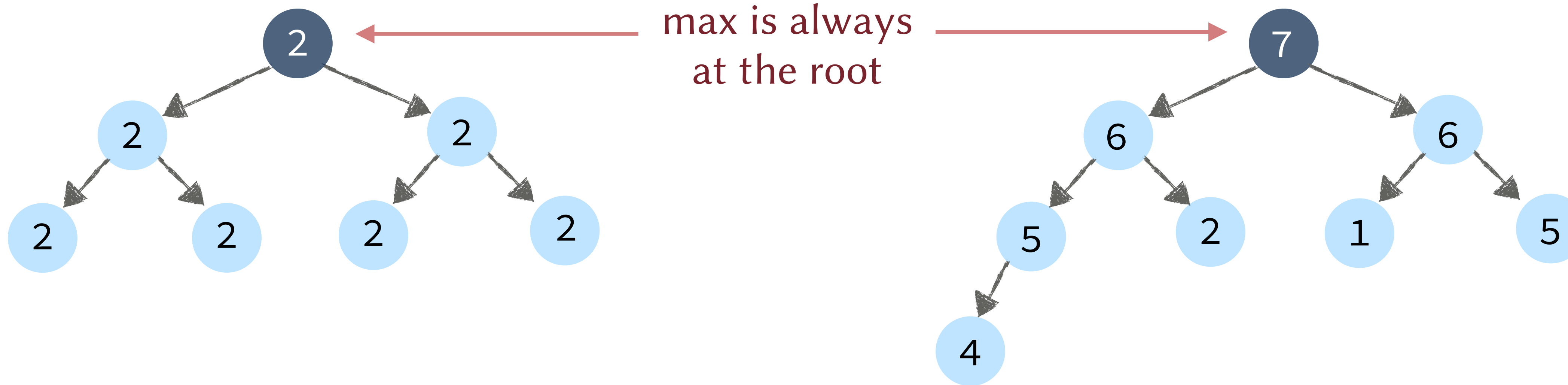
structure property violated

Binary Heaps (Tree Representation)

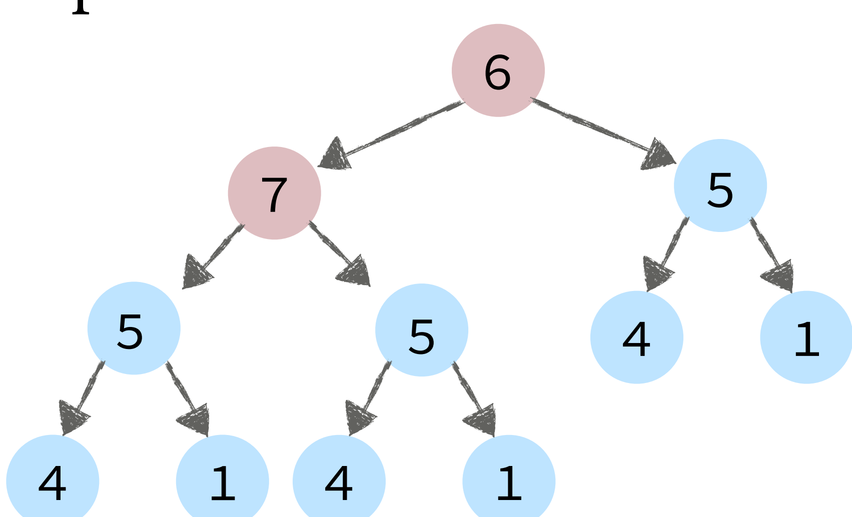
Binary Heap: (max-ordered)

- Structure: Must be a complete binary tree.
- Order: Every node is not less than its children.

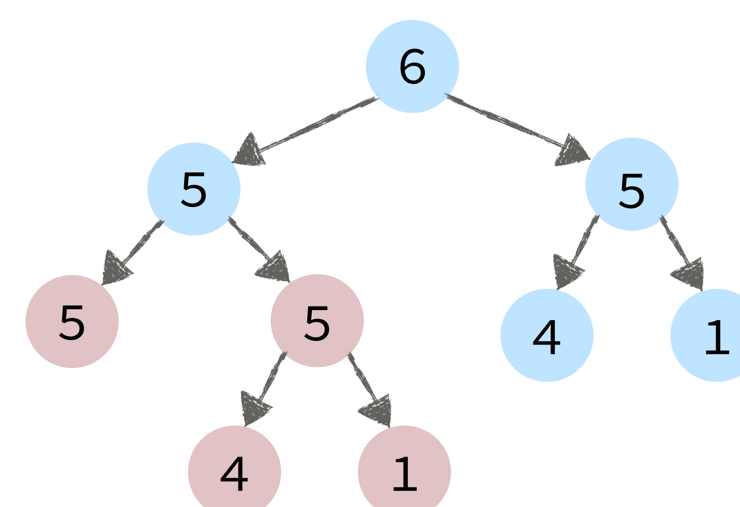
Examples:



Non-Examples:



order property violated



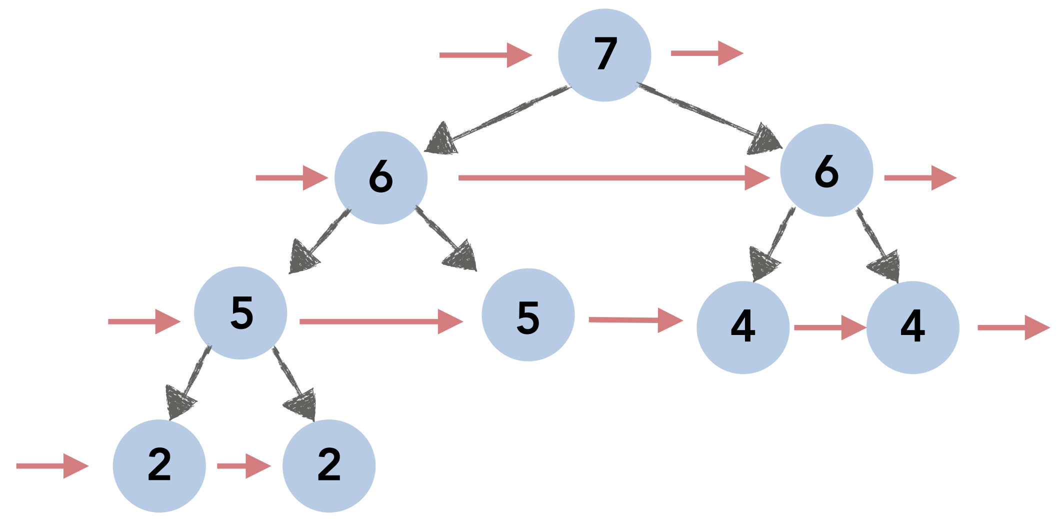
structure property violated

Binary Heaps (Array Representation)

Binary Heap: (max-ordered)

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

array has the tree
nodes in level-order

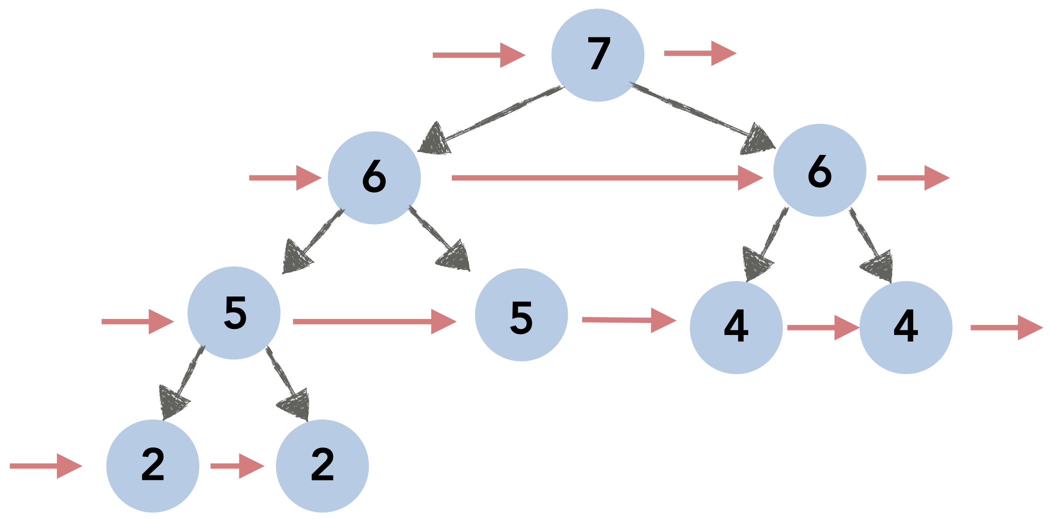


Binary Heaps (Array Representation)

Binary Heap: (max-ordered)

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

array has the tree nodes in level-order



Three simple functions.

LEFT(i)

```
return 2*i + 1
```

RIGHT(i)

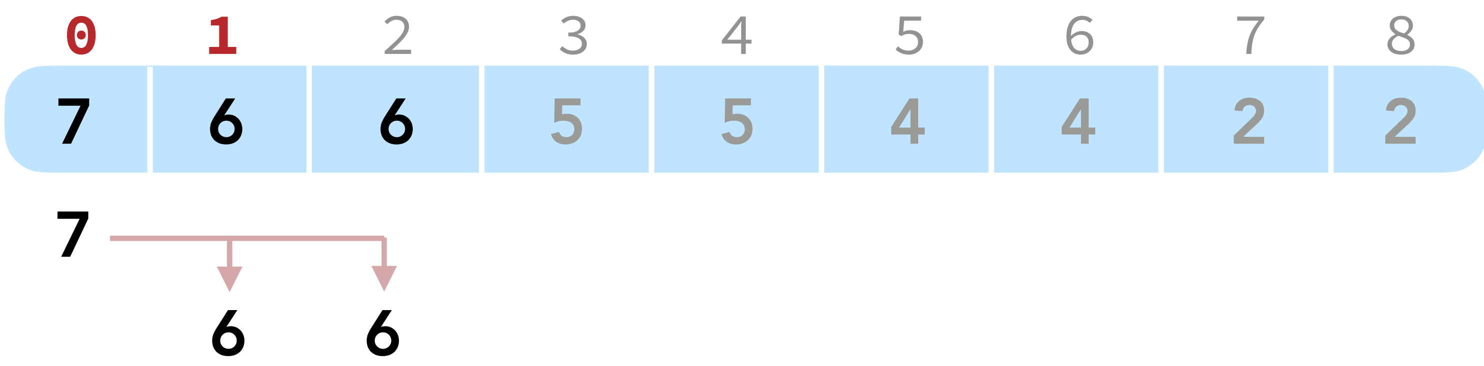
```
return 2*i + 2
```

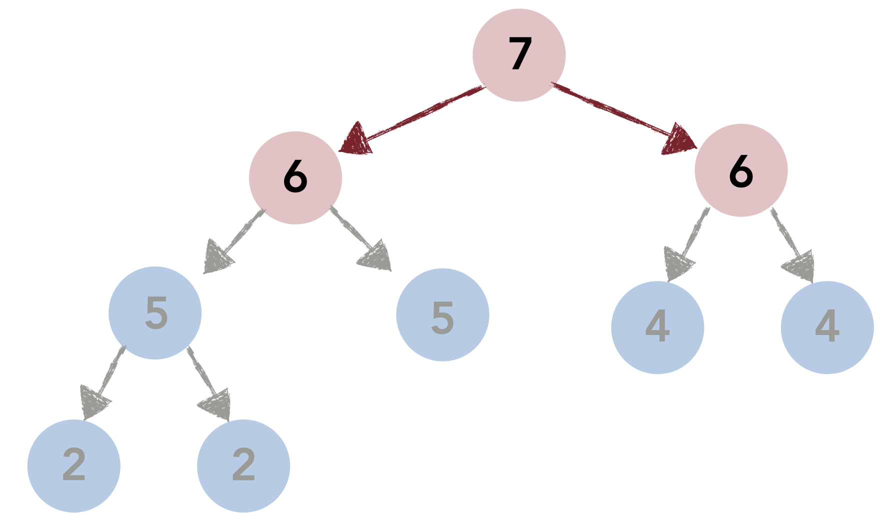
PARENT(i)

```
return (i-1)/2
```

Binary Heaps (Array Representation)

Binary Heap: (max-ordered)





Three simple functions.

LEFT(i)

```
return 2*i + 1
```

left child is at index
2*0 + 1 = 1

RIGHT(i)

```
return 2*i + 2
```

Right child is at index
2*0 + 2 = 2

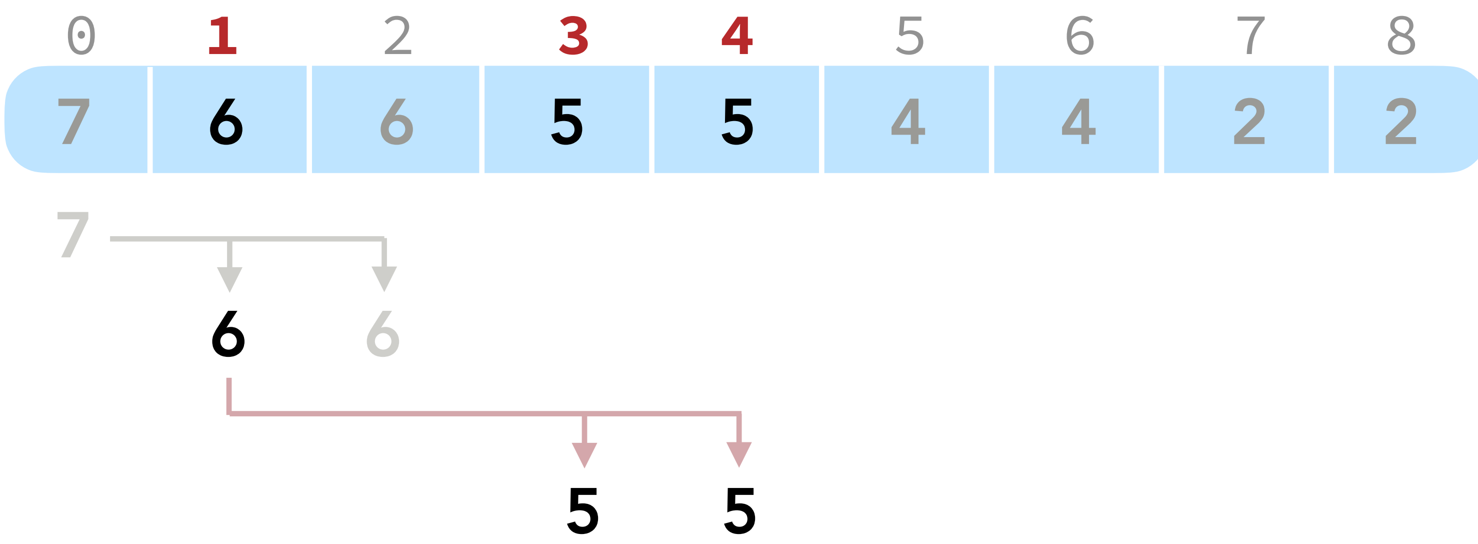
PARENT(i)

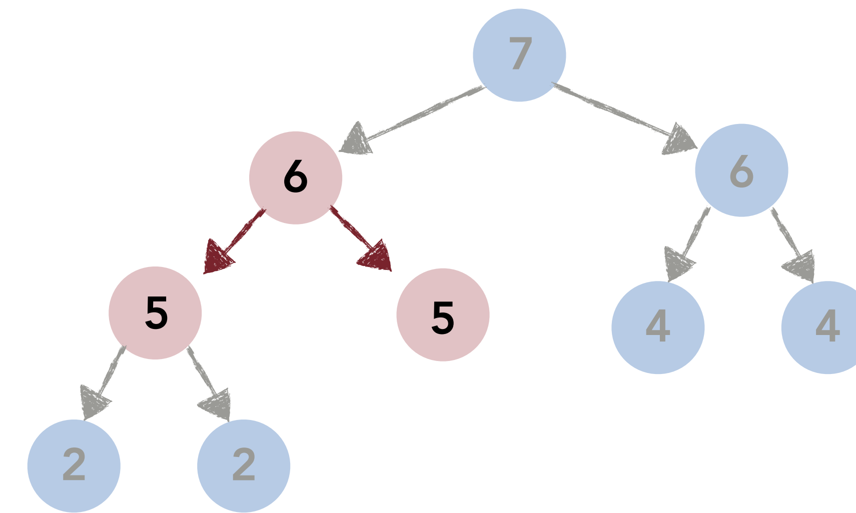
```
return (i-1)/2
```

Parent of the node at 0
is negative (no parent)

Binary Heaps (Array Representation)

Binary Heap: (max-ordered)





Three simple functions.

LEFT(i)

```
return 2*i + 1
```

left child is at index
2*1 + 1 = 3

RIGHT(i)

```
return 2*i + 2
```

Right child is at index
2*1 + 2 = 4

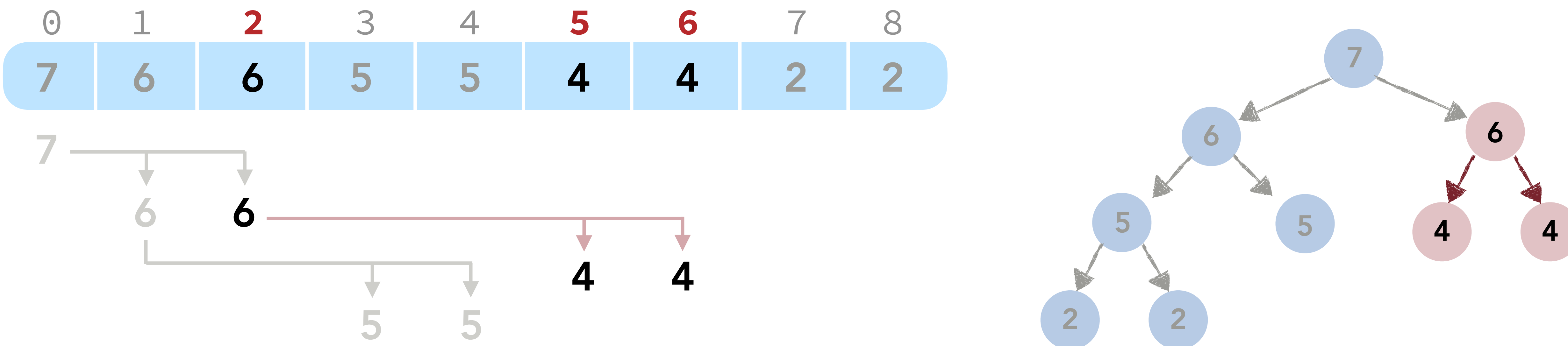
PARENT(i)

```
return (i-1)/2
```

Parent is at index
(1-1)/2 = 0

Binary Heaps (Array Representation)

Binary Heap: (max-ordered)



Three simple functions.

LEFT(i)

```
return 2*i + 1
```

left child is at index
2*2 + 1 = 5

RIGHT(i)

```
return 2*i + 2
```

Right child is at index
2*2 + 2 = 6

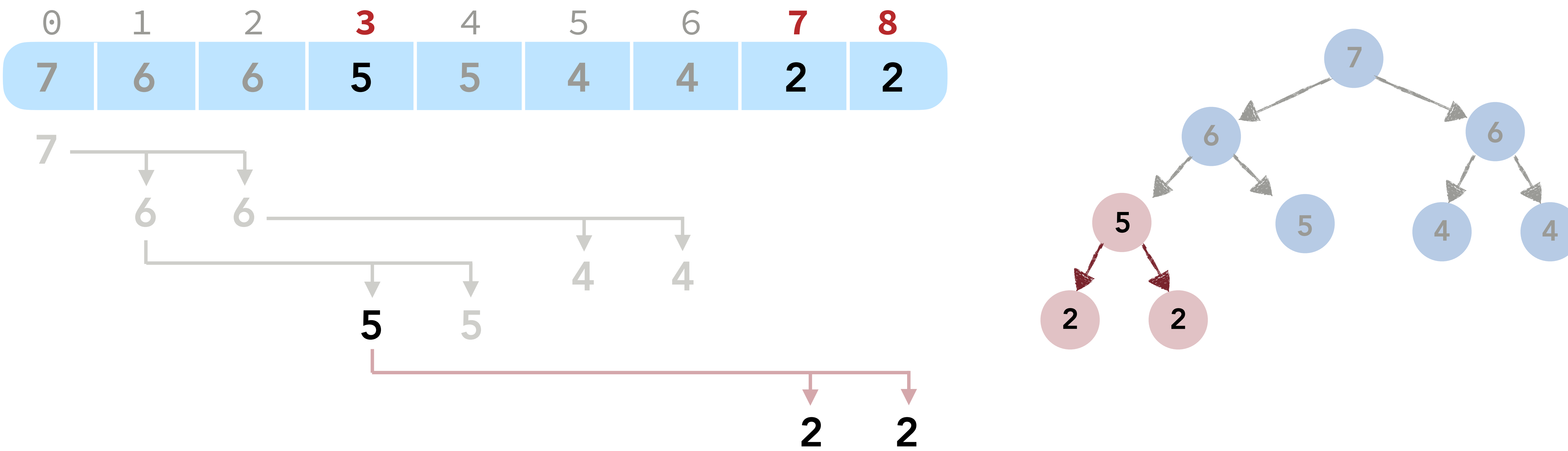
PARENT(i)

```
return (i-1)/2
```

Parent is at index
(2-1)/2 = 0

Binary Heaps (Array Representation)

Binary Heap: (max-ordered)



Three simple functions.

LEFT(i)

```
return 2*i + 1
```

left child is at index
2*3 + 1 = 7

RIGHT(i)

```
return 2*i + 2
```

Right child is at index
2*3 + 2 = 8

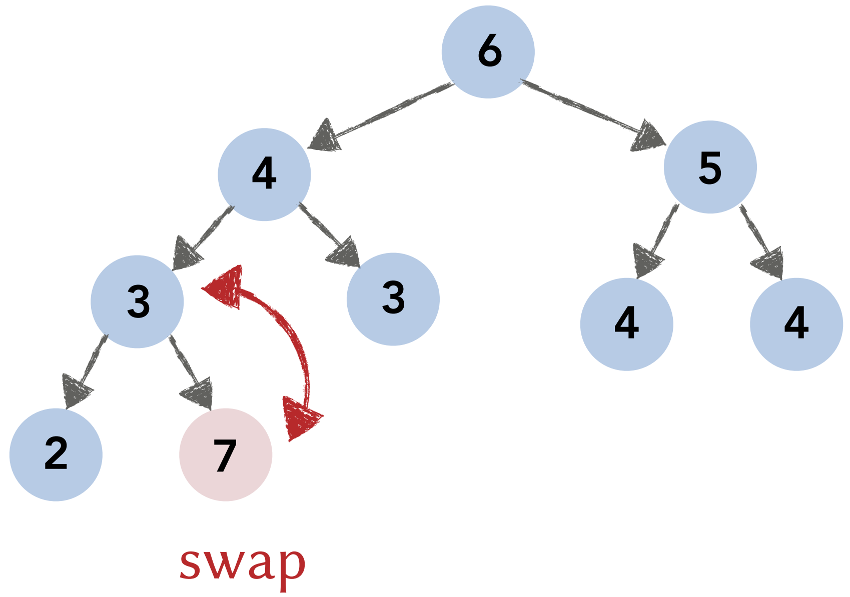
PARENT(i)

```
return (i-1)/2
```

Parent is at index
(3-1)/2 = 1

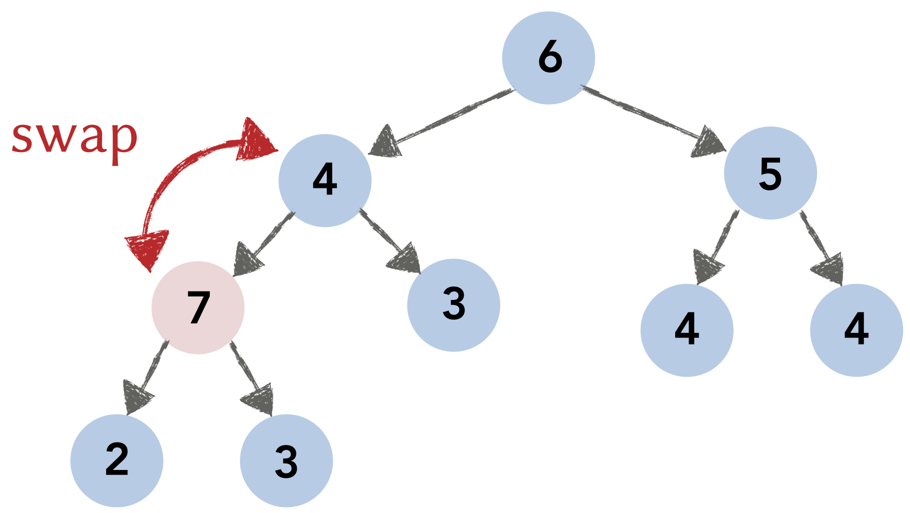
Fixing a *Locally* Broken Heap

1. If an item becomes **larger** than its parent, push it **up** the tree to maintain the heap order property.



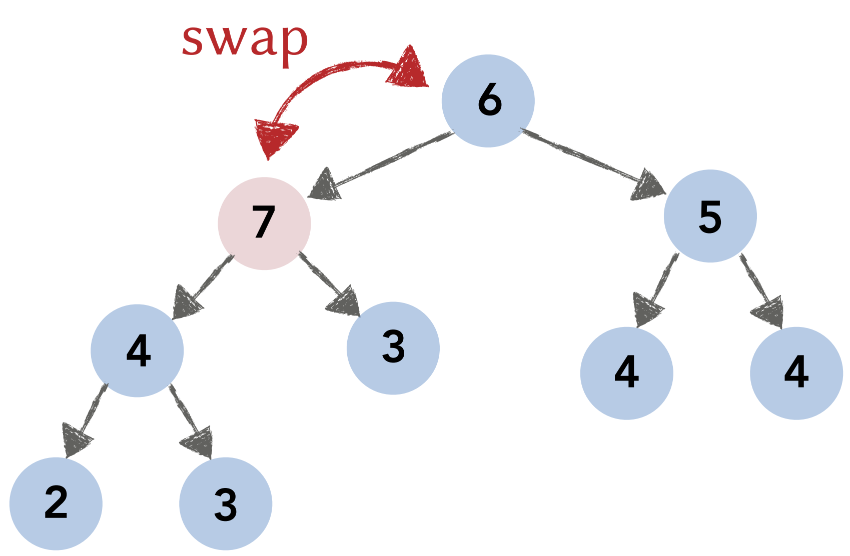
Fixing a *Locally* Broken Heap

1. If an item becomes **larger** than its parent, push it **up** the tree to maintain the heap order property.



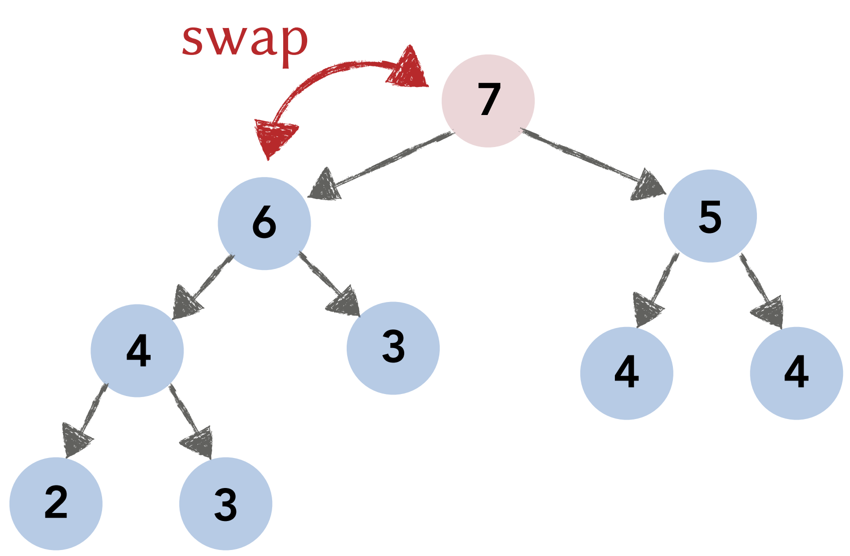
Fixing a *Locally* Broken Heap

1. If an item becomes **larger** than its parent, push it **up** the tree to maintain the heap order property.



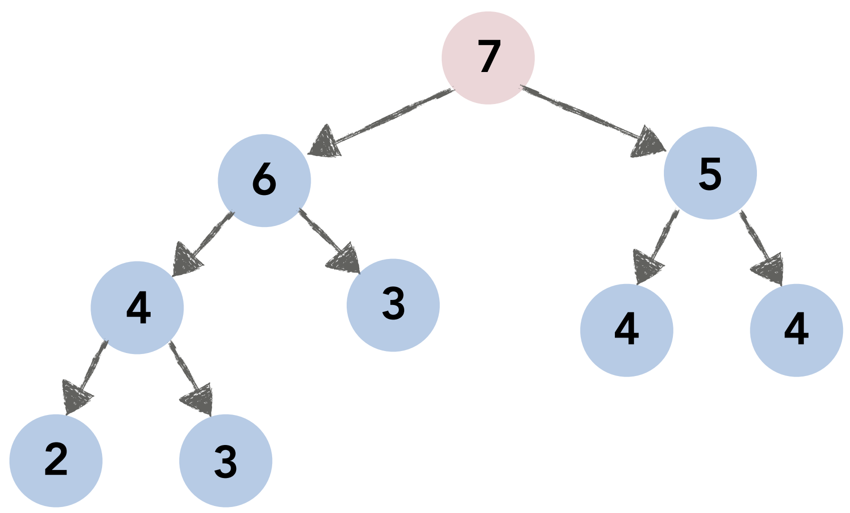
Fixing a *Locally* Broken Heap

1. If an item becomes **larger** than its parent, push it **up** the tree to maintain the heap order property.



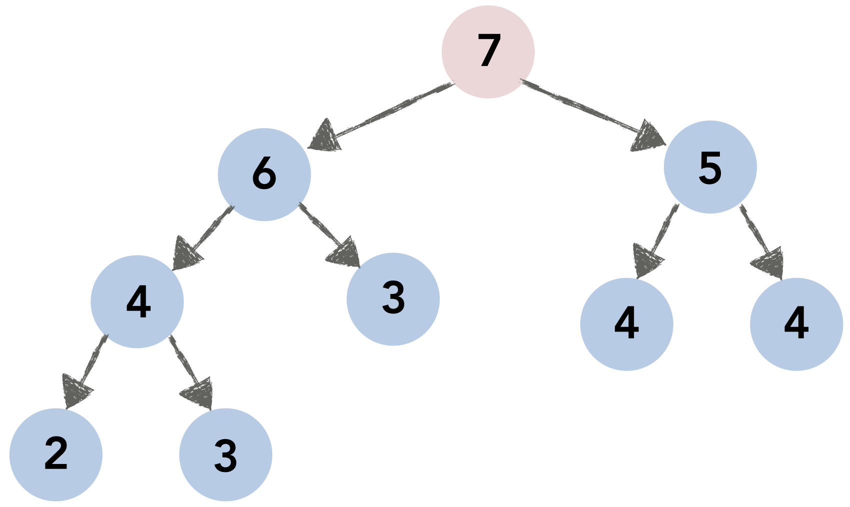
Fixing a *Locally* Broken Heap

1. If an item becomes **larger** than its parent, push it **up** the tree to maintain the heap order property.



Fixing a *Locally* Broken Heap

1. If an item becomes **larger** than its parent, push it **up** the tree to maintain the heap order property.

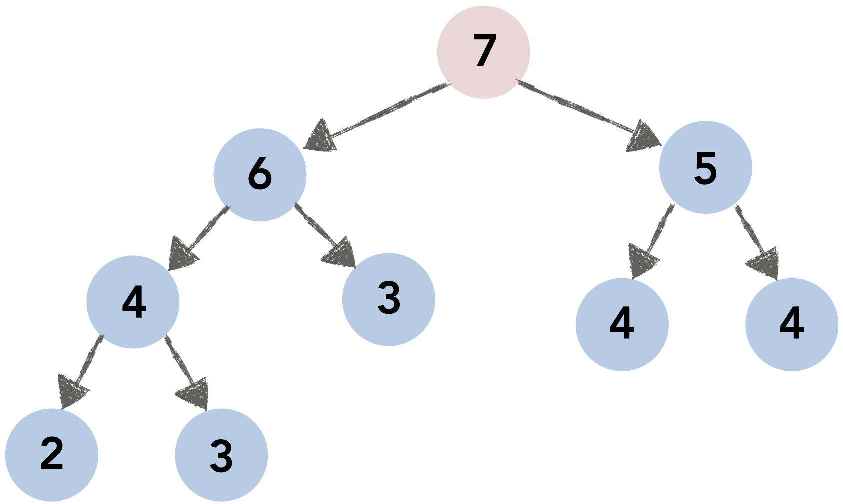


SWIM(a[], i, size)

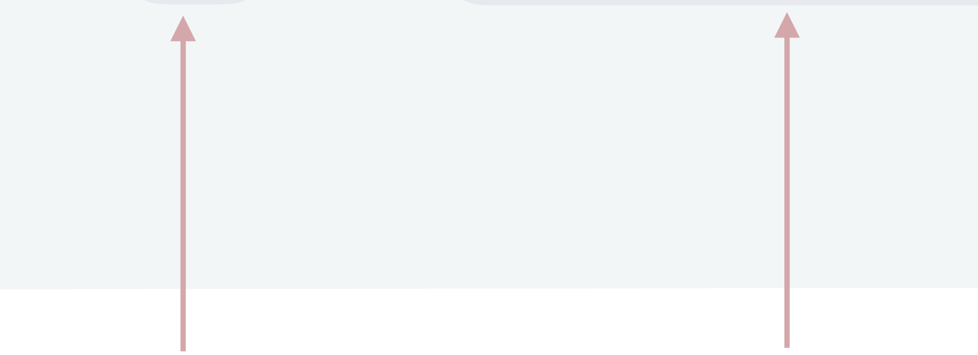
also called `SiftUp()`
(not `shiftup`) on wikipedia

Fixing a *Locally* Broken Heap

1. If an item becomes **larger** than its parent, push it **up** the tree to maintain the heap order property.

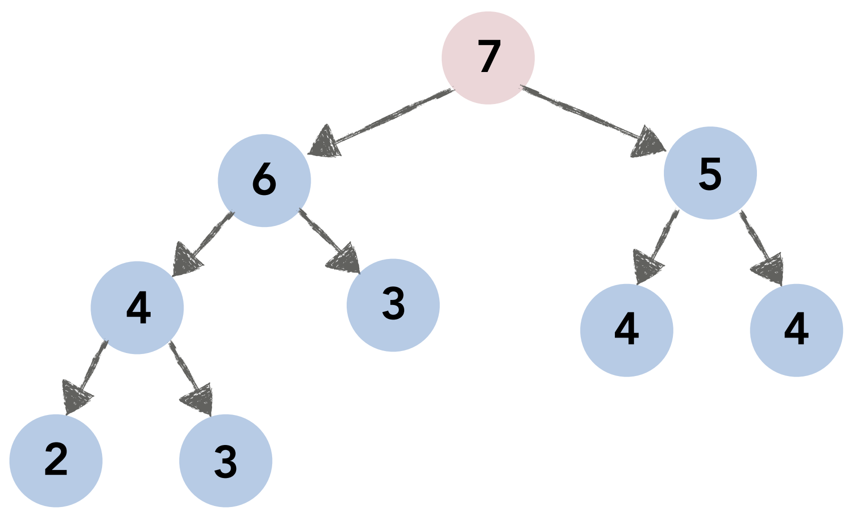



```
SWIM(a[], i, size)

while (i>0 and a[i] > a[PARENT(i)]):
```



Fixing a *Locally* Broken Heap

1. If an item becomes **larger** than its parent, push it **up** the tree to maintain the heap order property.

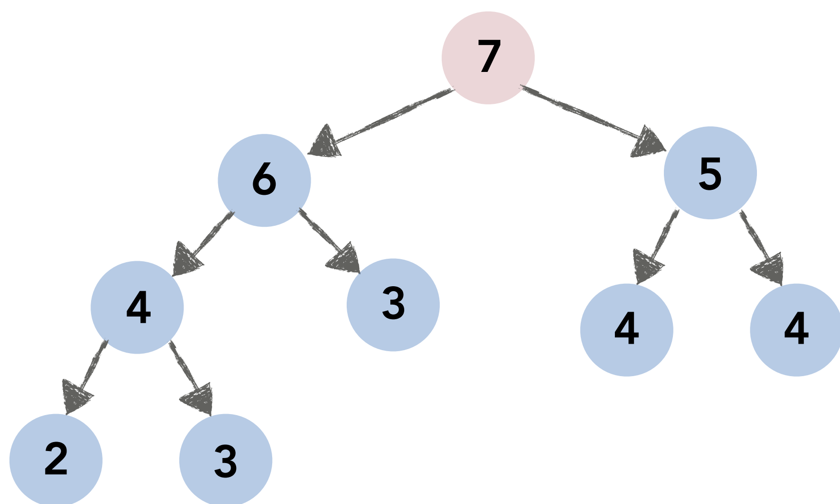



```
SWIM(a[], i, size)

while (i>0 and a[i] > a[PARENT(i)]):
    swap(a[i], a[PARENT(i)])
    i = PARENT(i)
```

swap values with the parent and move to the parent for the next iteration

Fixing a *Locally* Broken Heap

1. If an item becomes **larger** than its parent, push it **up** the tree to maintain the heap order property.

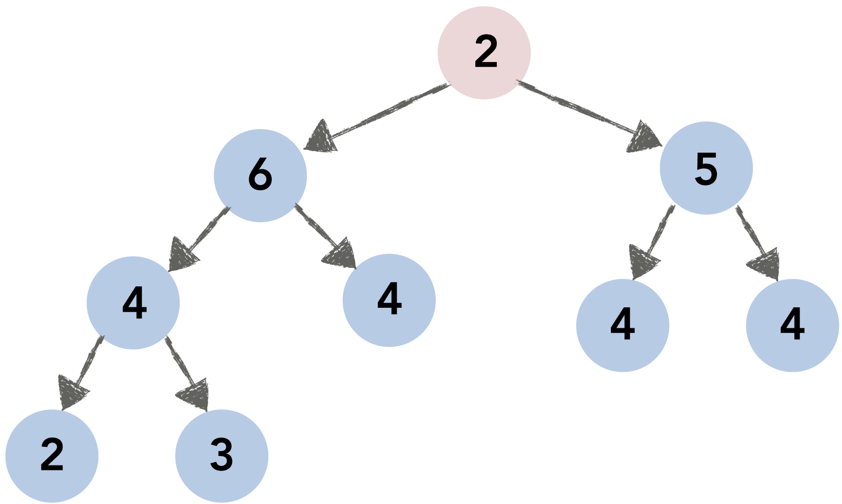



```
SWIM(a[], i, size)

while (i>0 and a[i] > a[PARENT(i)]):
    swap(a[i], a[PARENT(i)])
    i = PARENT(i)
```

Running Time. $O(\log n)$
1 swap and 1 compare per iteration. The number of iterations is bounded by the tree height.

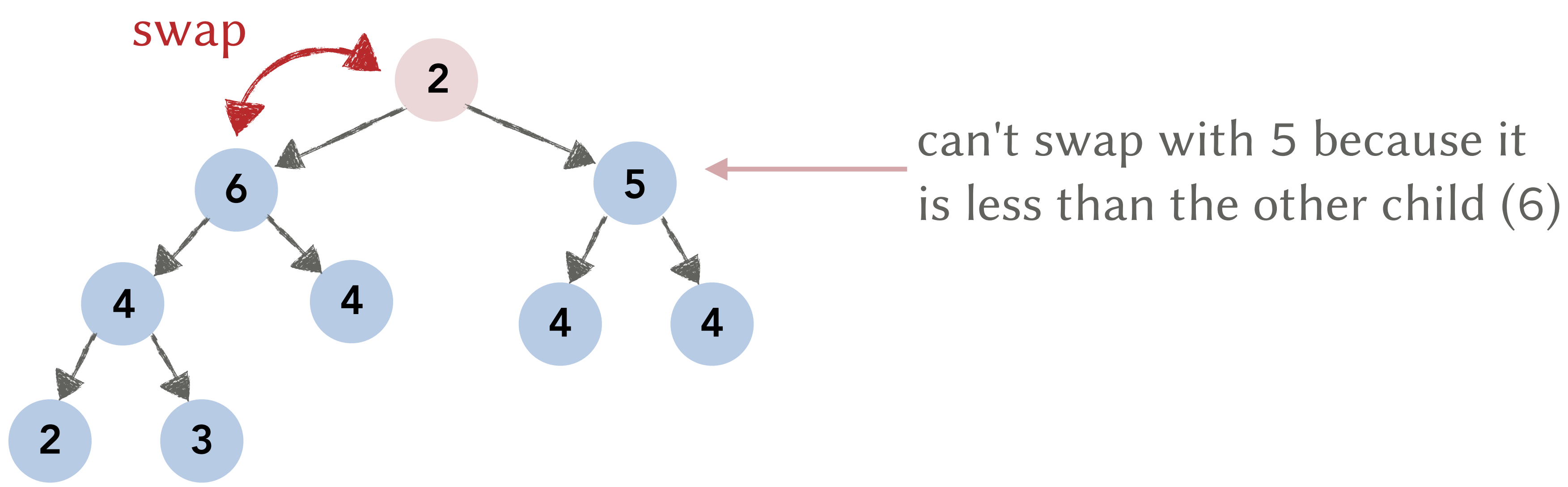
Fixing a *Locally* Broken Heap

2. If an item becomes **less** than one of its children, push it **down** the tree to maintain the heap order property.



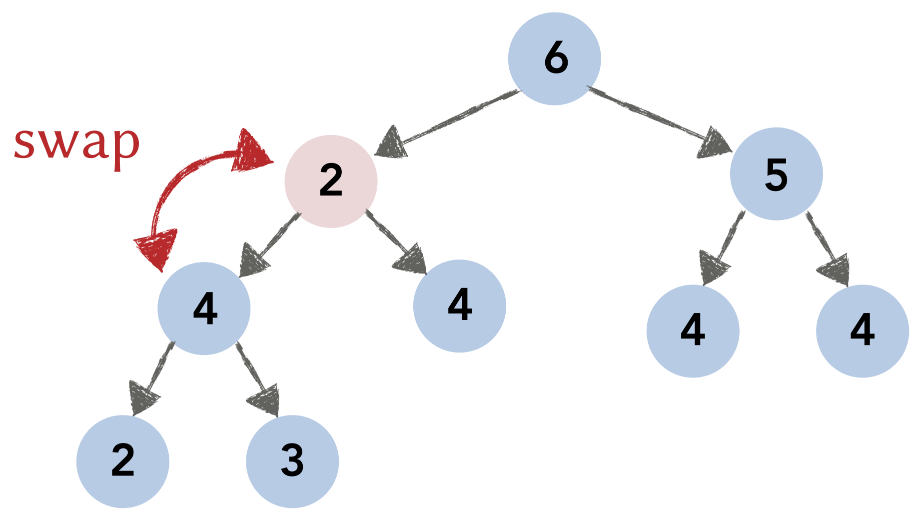
Fixing a *Locally* Broken Heap

2. If an item becomes **less** than one of its children, push it **down** the tree to maintain the heap order property.



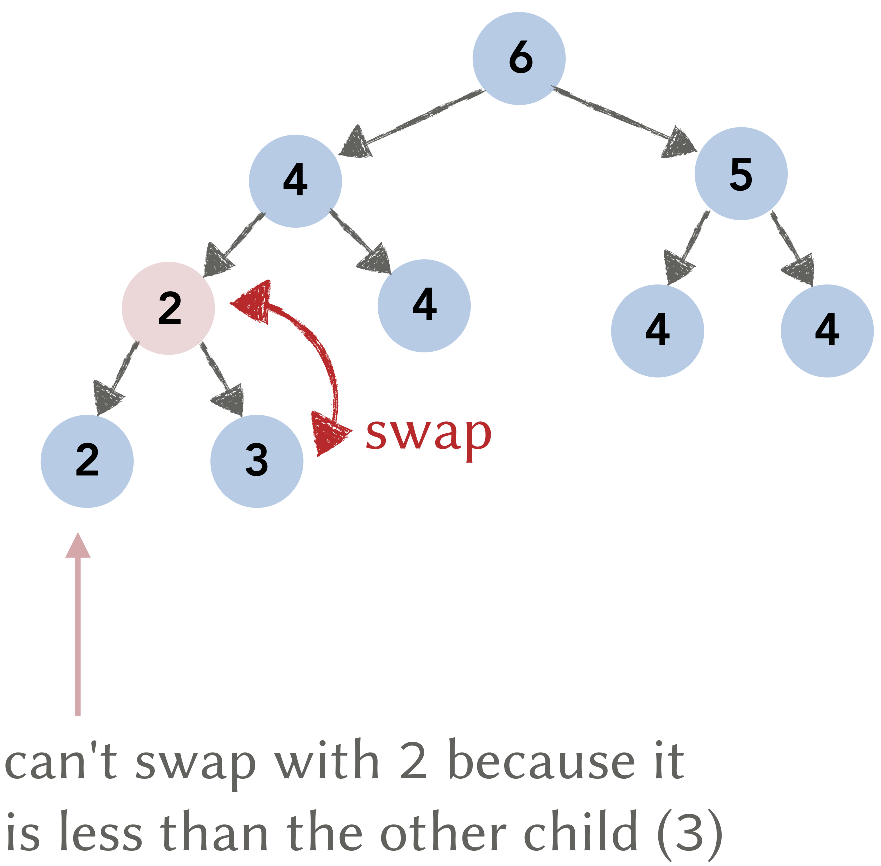
Fixing a *Locally* Broken Heap

2. If an item becomes **less** than one of its children, push it **down** the tree to maintain the heap order property.



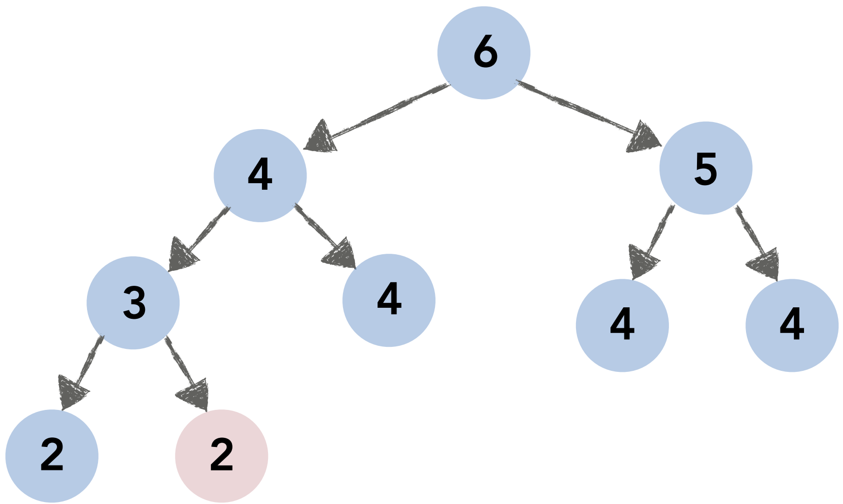
Fixing a *Locally* Broken Heap

2. If an item becomes **less** than one of its children, push it **down** the tree to maintain the heap order property.



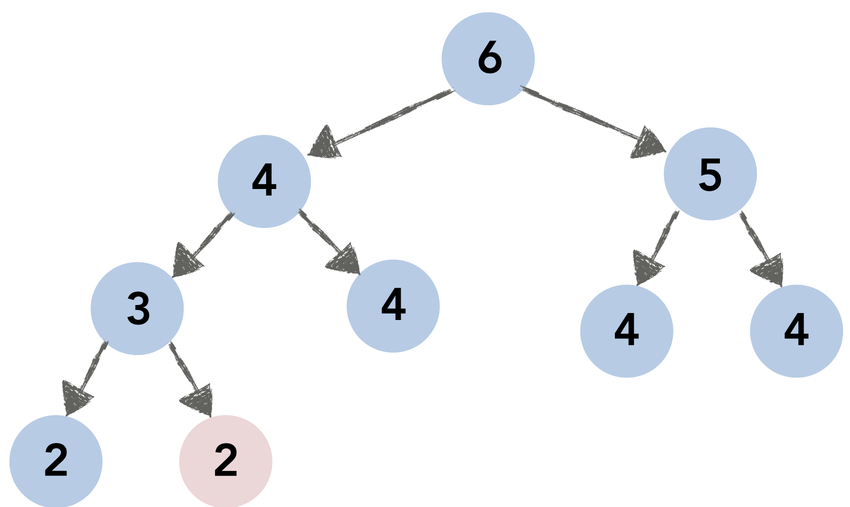
Fixing a *Locally* Broken Heap

2. If an item becomes **less** than one of its children, push it **down** the tree to maintain the heap order property.



Fixing a *Locally* Broken Heap

2. If an item becomes **less** than one of its children, push it **down** the tree to maintain the heap order property.



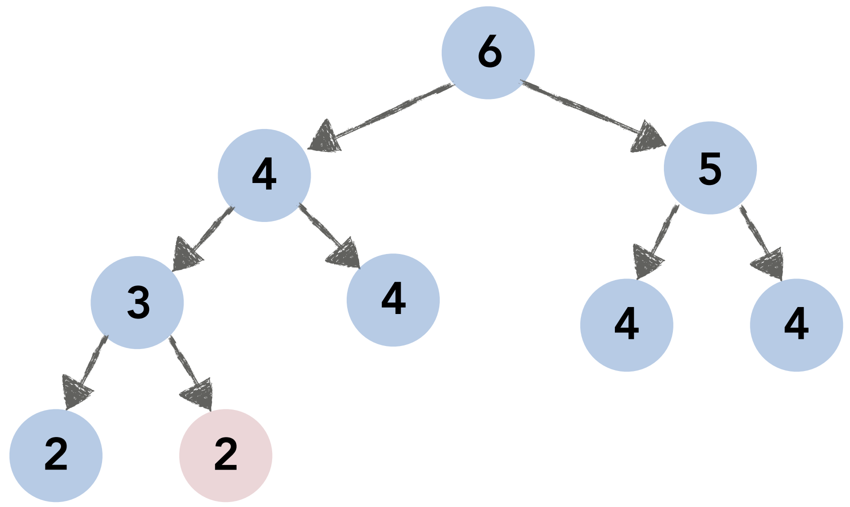

```
SINK(a[], i, size)
```

also called:

- `SIFTDOWN` on wikipedia
- `MAX-HEAPIFY` in our text-book
- `FIX-HEAP` in the slides of the other sections!

Fixing a *Locally* Broken Heap

2. If an item becomes **less** than one of its children, push it **down** the tree to maintain the heap order property.

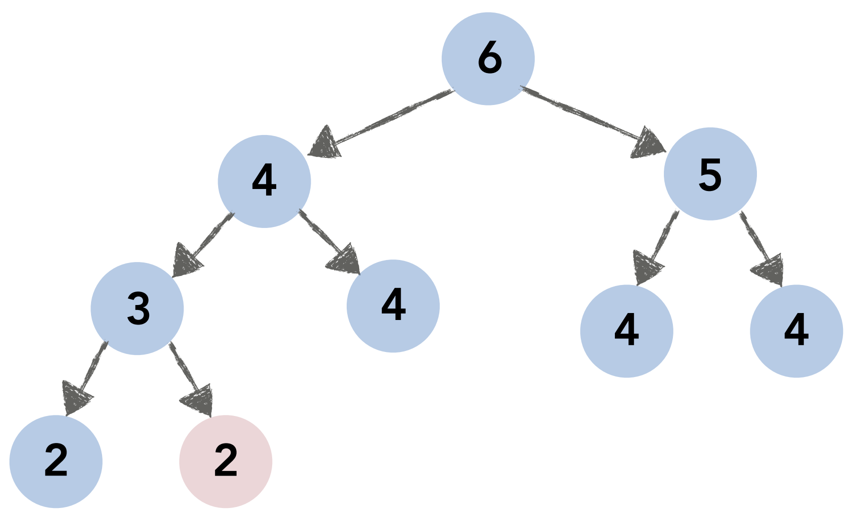



```
SINK(a[], i, size)
while (LEFT(i) < size):
```

while there is a left child

Fixing a *Locally* Broken Heap

2. If an item becomes **less** than one of its children, push it **down** the tree to maintain the heap order property.

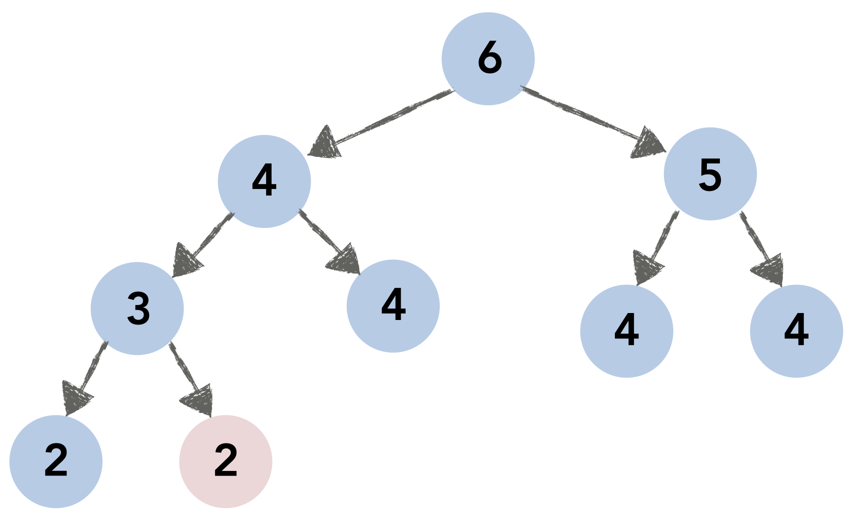



```
SINK(a[], i, size)

while (LEFT(i) < size):
   k = LEFT(i)

   if (RIGHT(i) < size):
      if (a[k] < a[RIGHT(i)]): k = RIGHT(i)
```

pick between the `left` and `right` child depending on which one is the largest.

Fixing a *Locally* Broken Heap

2. If an item becomes **less** than one of its children, push it **down** the tree to maintain the heap order property.

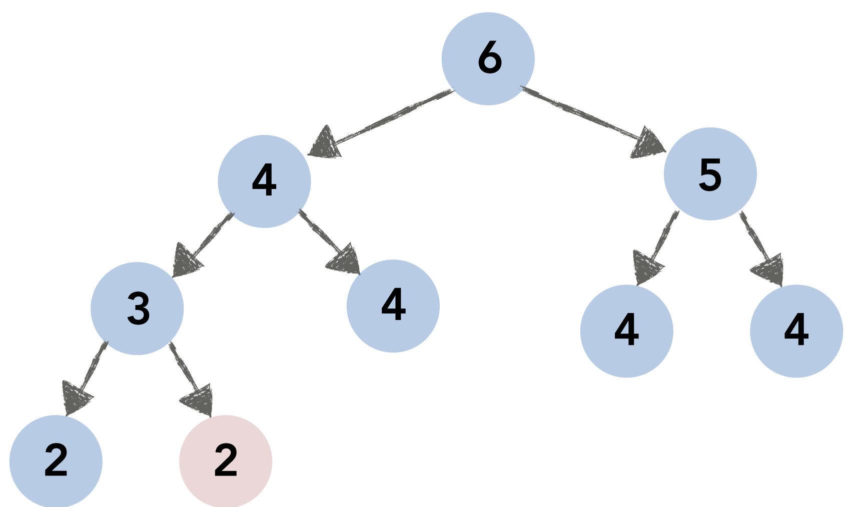



```
SINK(a[], i, size)

while (LEFT(i) < size):
   k = LEFT(i)

   if (RIGHT(i) < size):
      if (a[k] < a[RIGHT(i)]): k = RIGHT(i)
   if (a[i] < a[k]):
      swap(a[i], a[k])
      i = k
   else: break
```

swap with and move to the larger child or stop if no swap is necessary

Fixing a *Locally* Broken Heap

2. If an item becomes **less** than one of its children, push it **down** the tree to maintain the heap order property.




```
SINK(a[], i, size)

while (LEFT(i) < size):
   k = LEFT(i)

   if (RIGHT(i) < size):
      if (a[k] < a[RIGHT(i)]): k = RIGHT(i)

   if (a[i] < a[k]):
      swap(a[i], a[k])
      i = k
   else: break
```

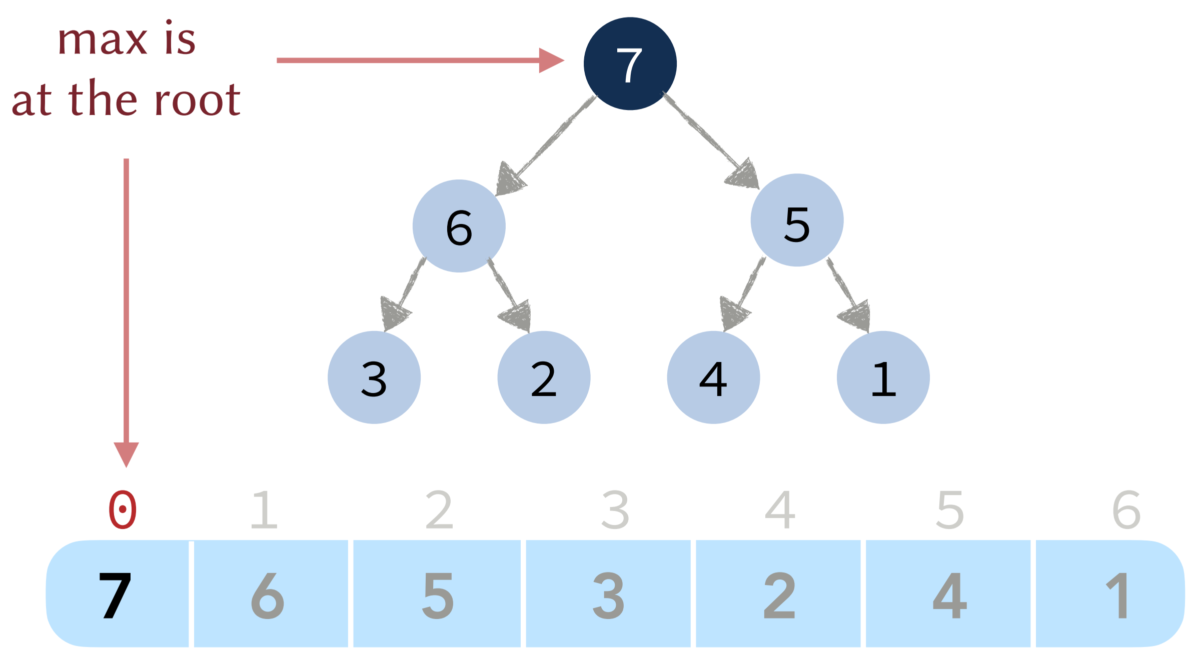
Running Time. $O(\log n)$

At most 1 swap and 2 comparisons per iteration

The number of iterations is bounded by the tree height.

Max-PQ Operations

Max: Always at index 0.
$\Theta(1)$



Max-PQ Operations

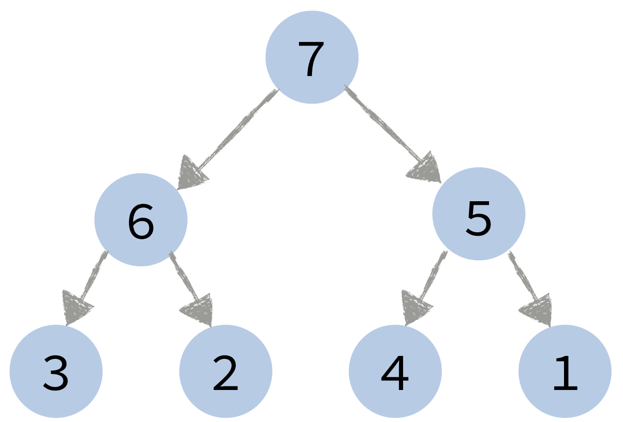
Max: Always at index 0.
$\Theta(1)$

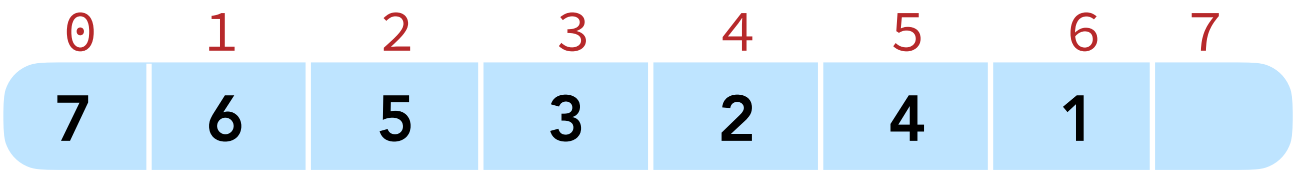
Insert: Insert at the end of the array and then **swim.**

INSERT(a[], k, size)

```
a[size] = k
size = size + 1
SWIM(a, size-1, size)
```

$O(\log n)$: Swim at most to the root.





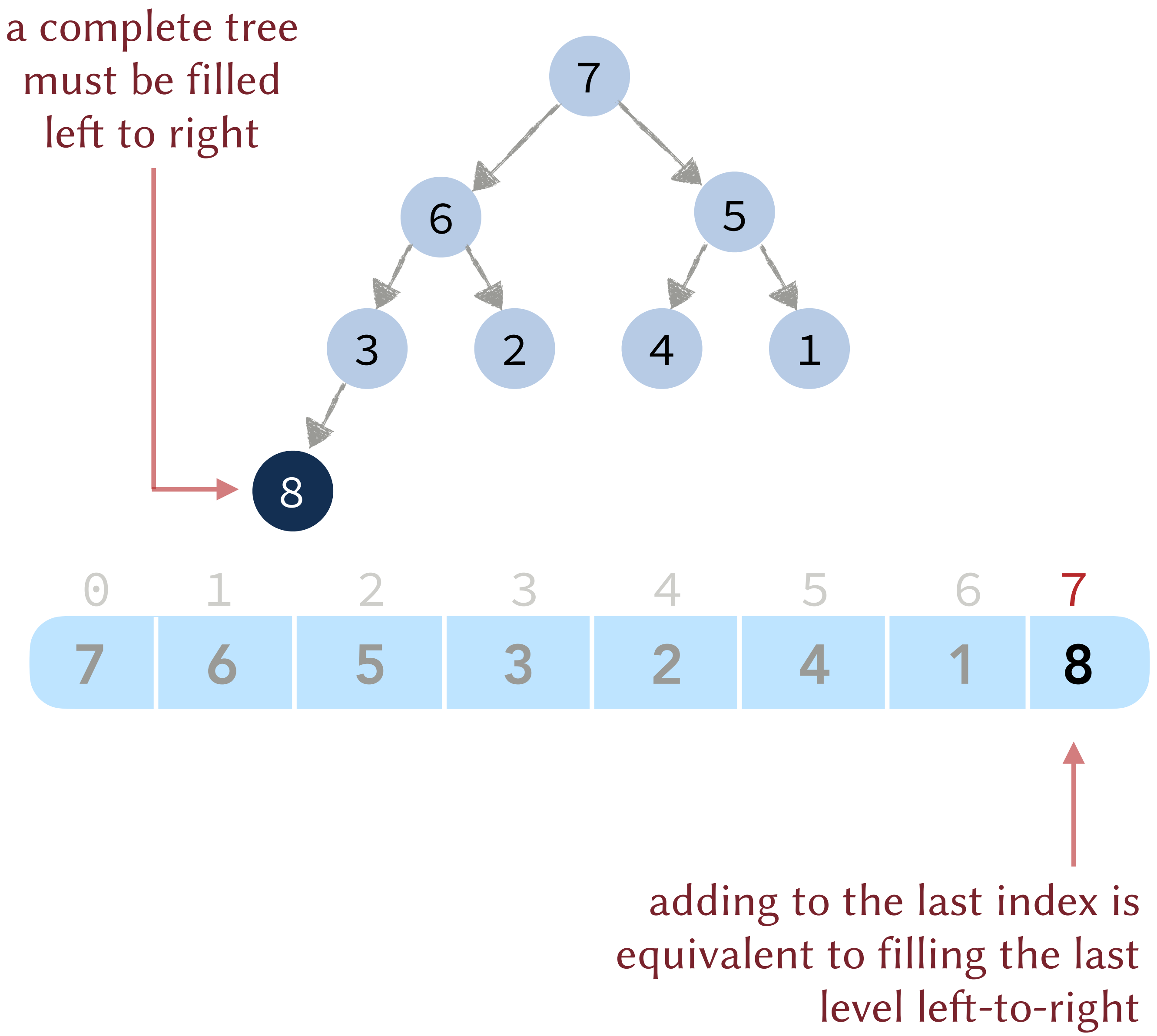
Max-PQ Operations

Max: Always at index 0.
$\Theta(1)$

Insert: Insert at the end of the array and then **swim.**

```
INSERT(a[], k, size)

  a[size] = k
  size = size + 1
  SWIM(a, size-1, size)
```

$O(\log n)$: Swim at most to the root.



Max-PQ Operations

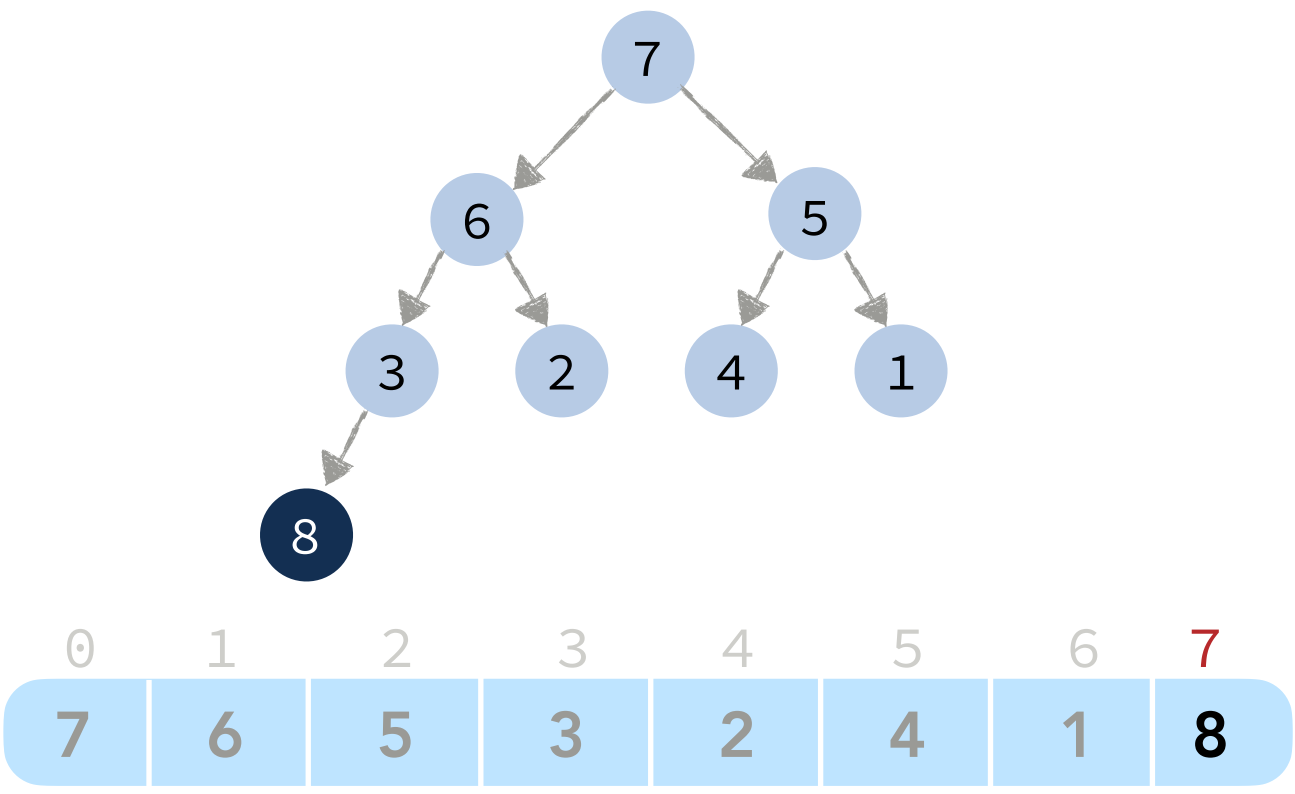
Max: Always at index 0.
$\Theta(1)$

Insert: Insert at the end of the array and then **swim.**

```
INSERT(a[], k, size)
  a[size] = k
  size = size + 1
  SWIM(a, size-1, size)
```

$O(\log n)$: Swim at most to the root.

fix the heap after inserting the new element



Max-PQ Operations

Max: Always at index 0.

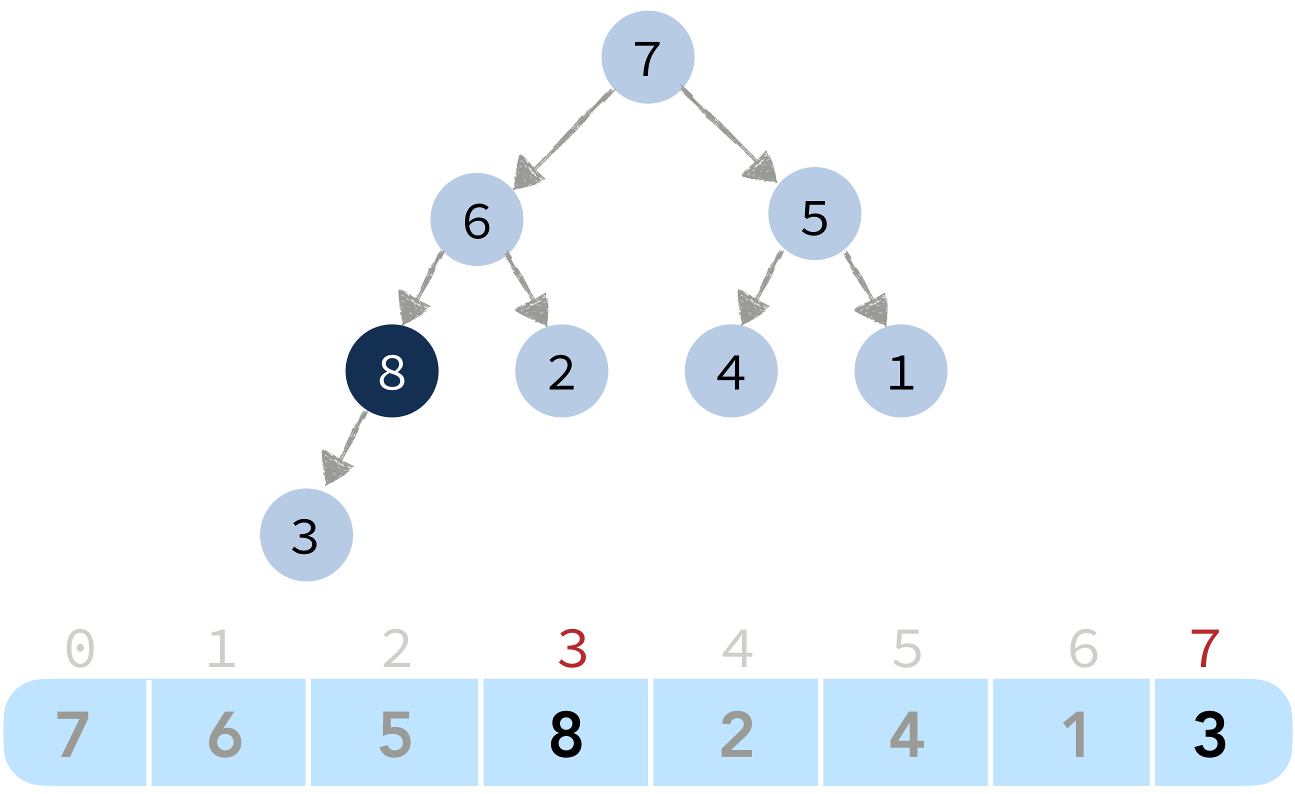
$\Theta(1)$

Insert: Insert at the end of the array and then **swim.**

```
INSERT(a[], k, size)

  a[size] = k
  size = size + 1
  SWIM(a, size-1, size)
```

$O(\log n)$: Swim at most to the root.

fix the heap after inserting the new element



Max-PQ Operations

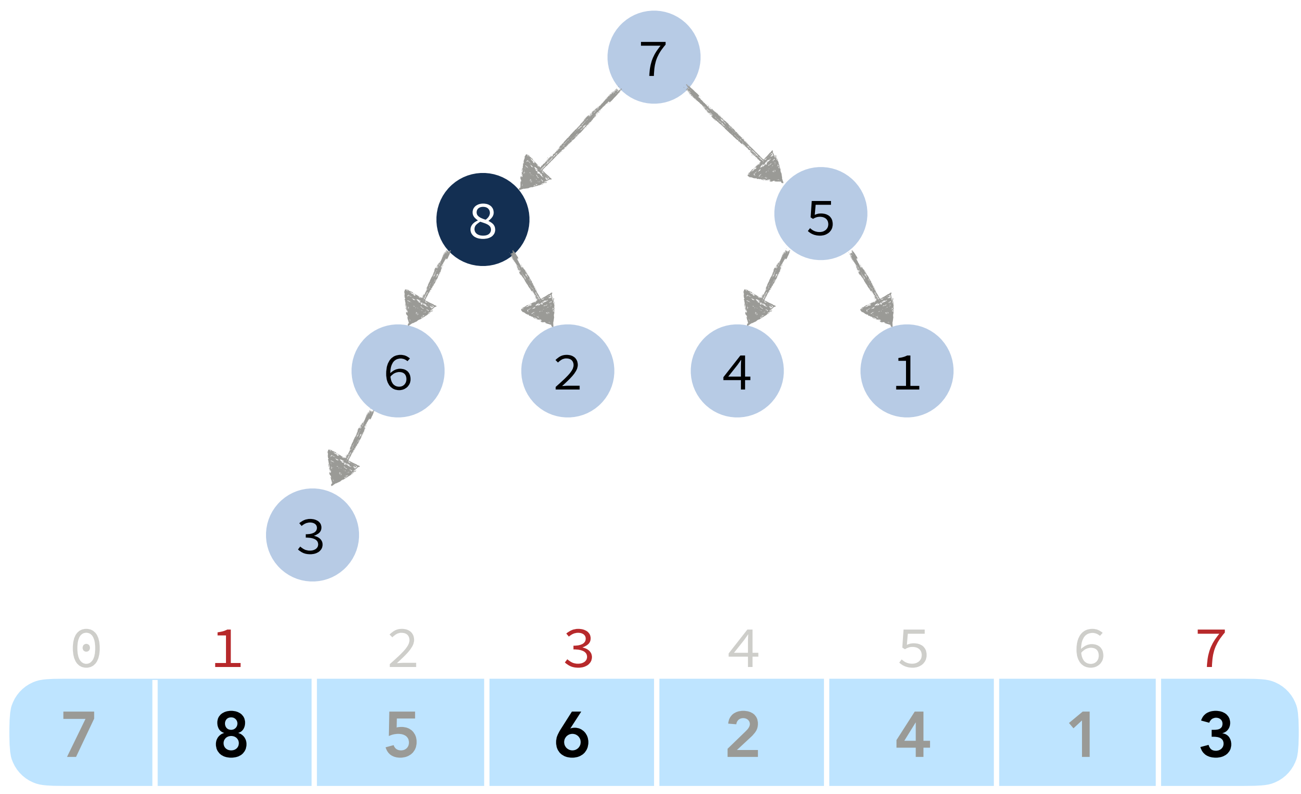
Max: Always at index 0.
$\Theta(1)$

Insert: Insert at the end of the array and then **swim.**

```
INSERT(a[], k, size)

  a[size] = k
  size = size + 1
  SWIM(a, size-1, size)
```

$O(\log n)$: Swim at most to the root.

fix the heap after inserting the new element



Max-PQ Operations

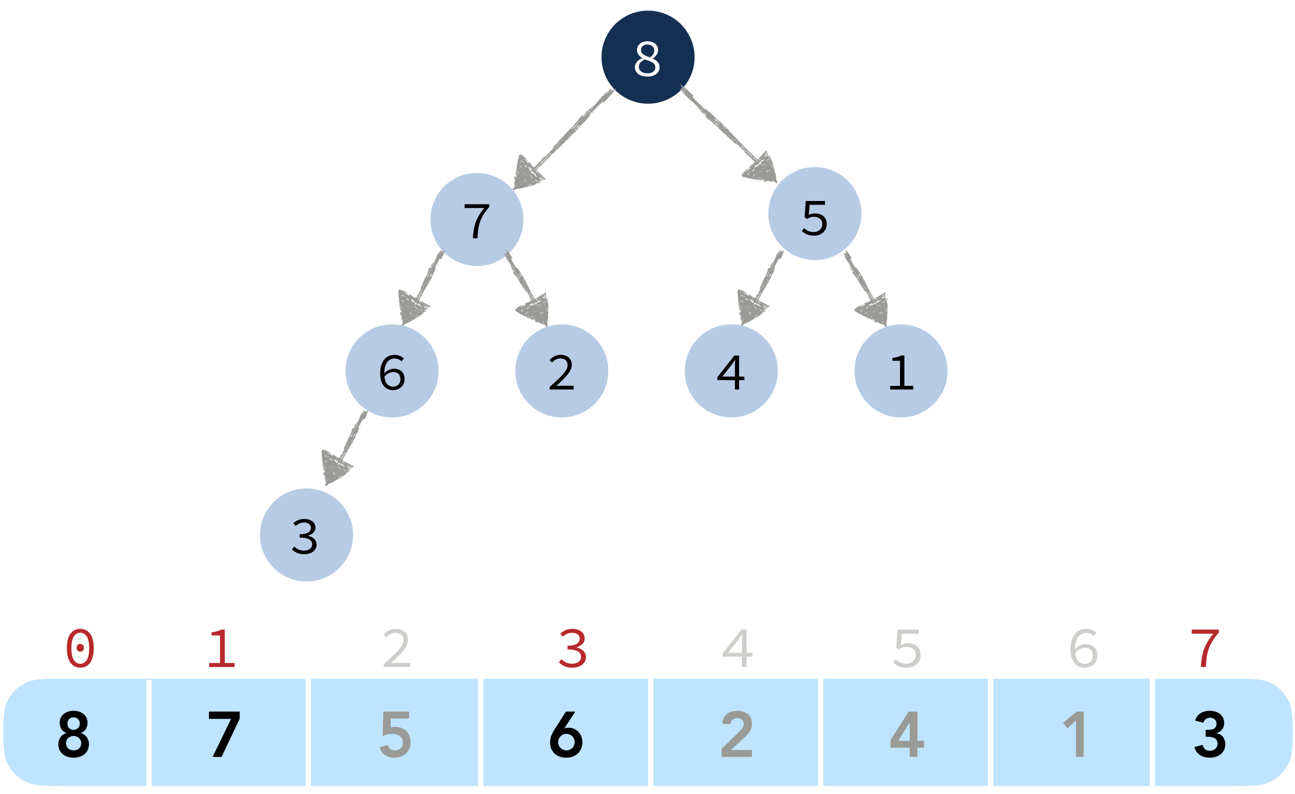
Max: Always at index 0.
$\Theta(1)$

Insert: Insert at the end of the array and then **swim.**

```
INSERT(a[], k, size)
  a[size] = k
  size = size + 1
  SWIM(a, size-1, size)
```

$O(\log n)$: Swim at most to the root.

fix the heap after inserting the new element



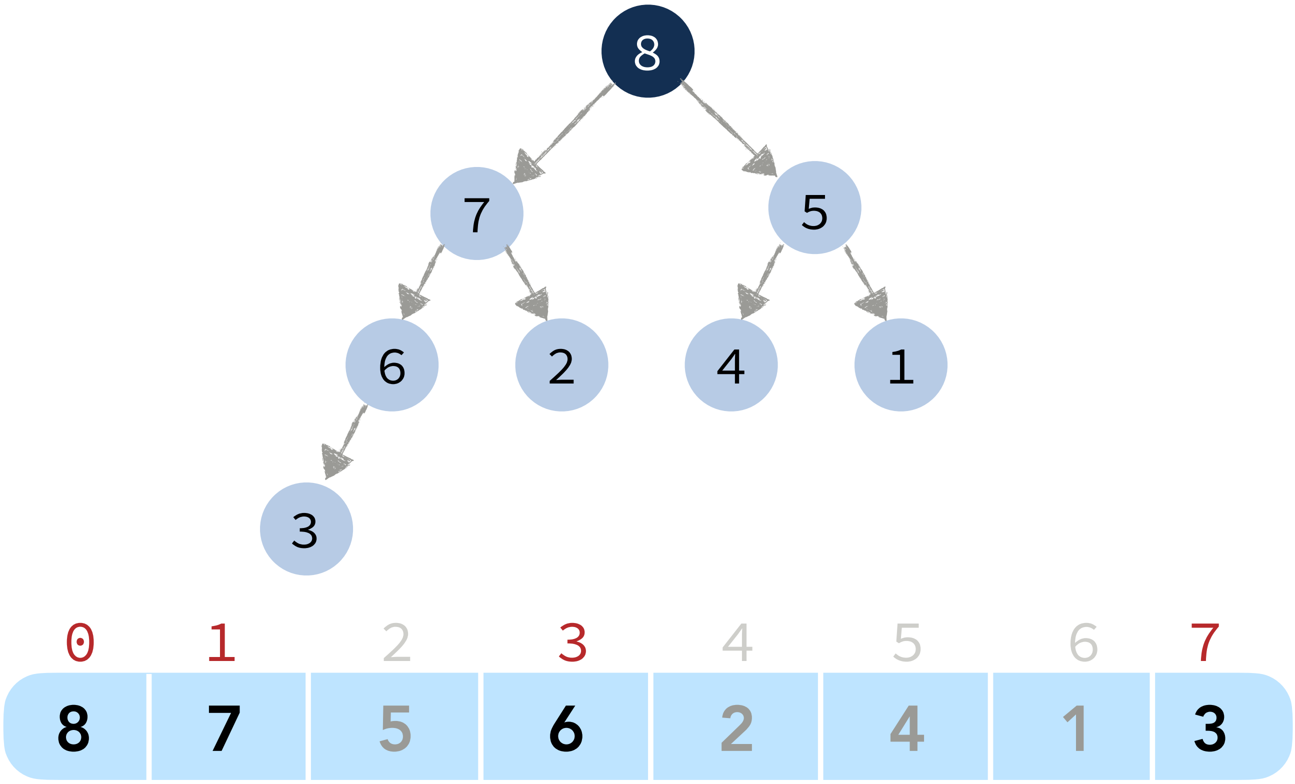
Max-PQ Operations

Max: Always at index 0.
$\Theta(1)$

Insert: Insert at the end of the array and then **swim.**

```
INSERT(a[], k, size)
  a[size] = k
  size = size + 1
  SWIM(a, size-1, size)
```

$O(\log n)$: Swim at most to the root.



Max-PQ Operations

Max: Always at index 0.
$\Theta(1)$

Insert: Insert at the end of the array and then **swim.**

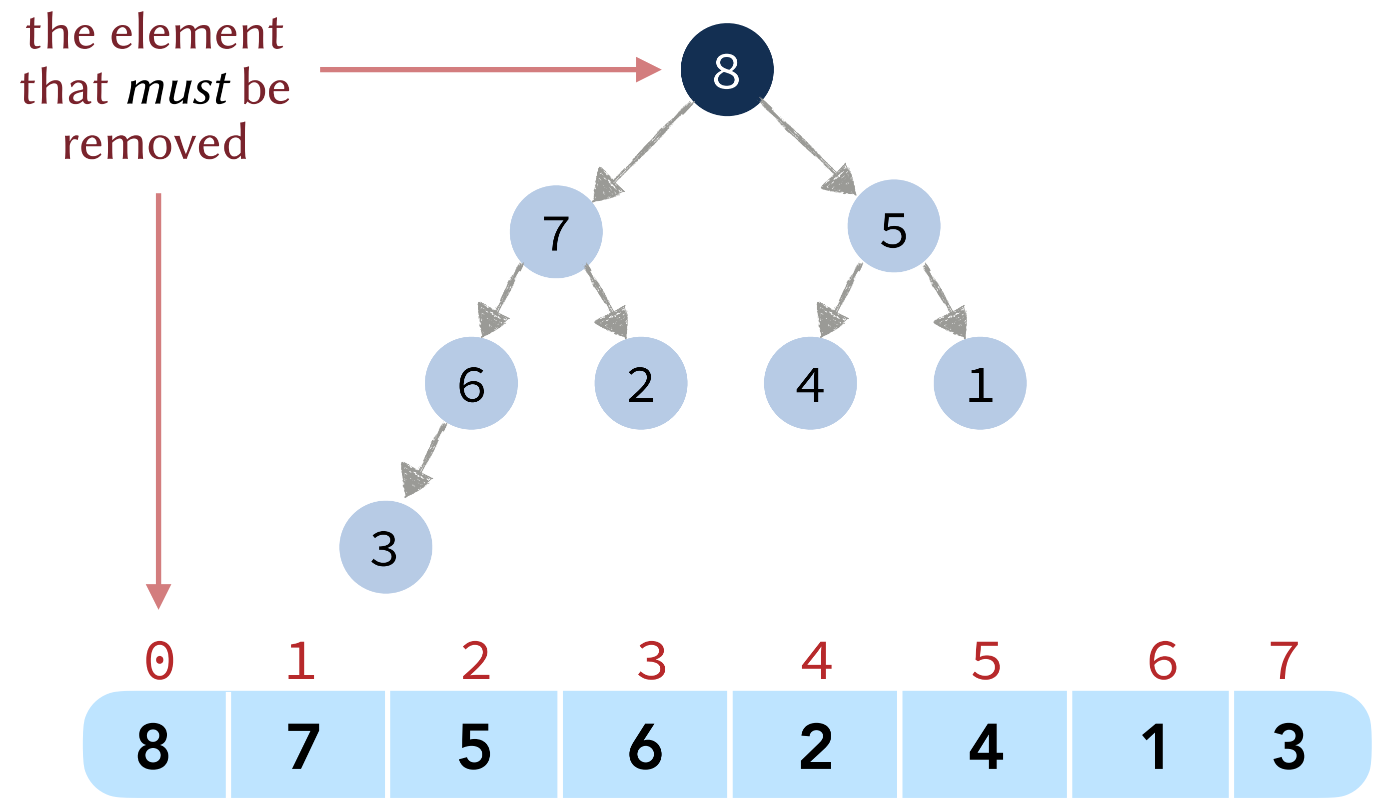
```
INSERT(a[], k, size)

  a[size] = k
  size = size + 1
  SWIM(a, size-1, size)
```

$O(\log n)$: Swim at most to the root.

Del-Max: Swap the last element with the element at index `0` and then **sink.**

```
DEL-MAX(a[], size)

  swap(a[size-1], a[0])
  size = size - 1
  SINK(a, 0, size)
```

$O(\log n)$: Sink at most to the last level.



Max-PQ Operations

Max: Always at index 0.
$\Theta(1)$

Insert: Insert at the end of the array and then **swim.**

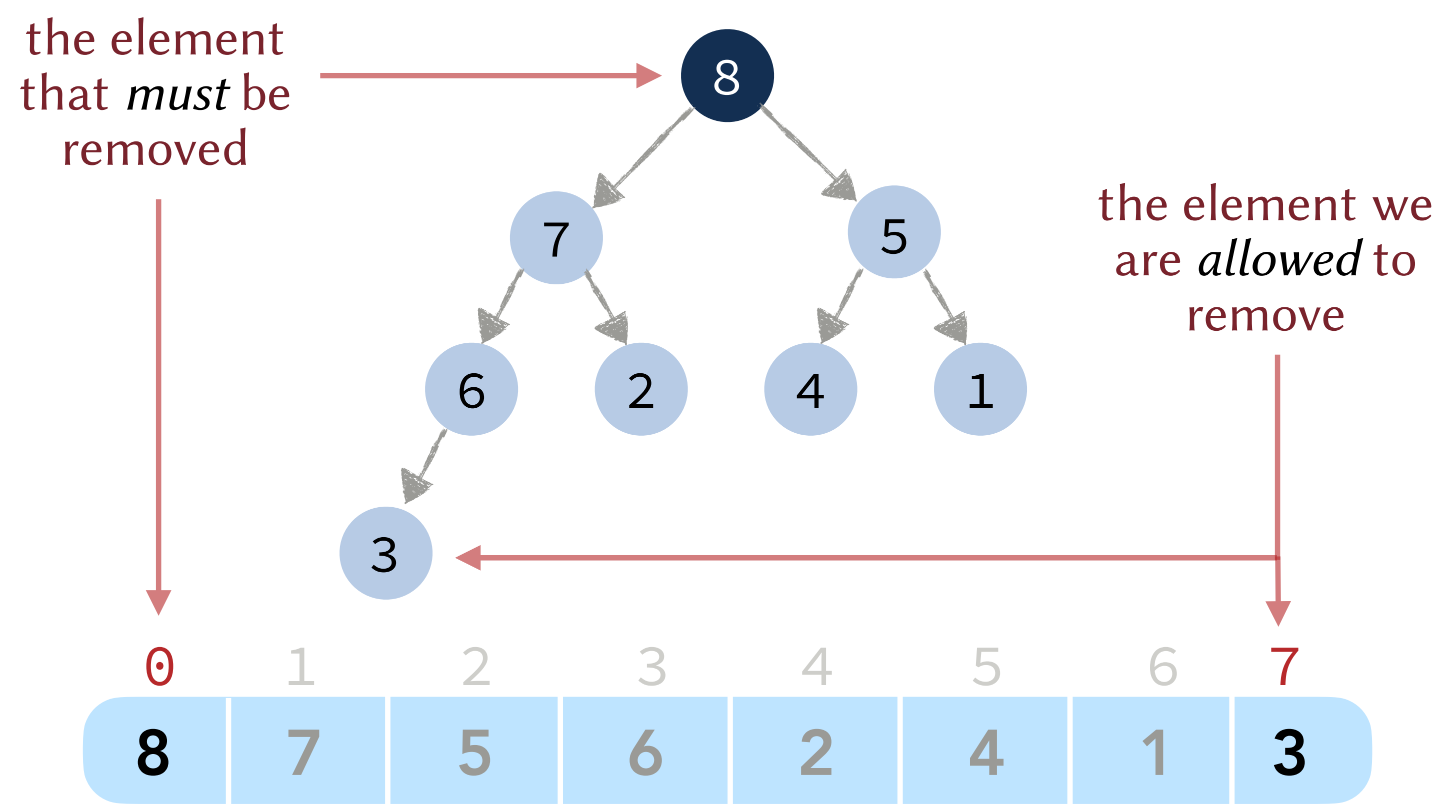
```
INSERT(a[], k, size)

  a[size] = k
  size = size + 1
  SWIM(a, size-1, size)
```

$O(\log n)$: Swim at most to the root.

Del-Max: Swap the last element with the element at index 0 and then **sink.**

```
DEL-MAX(a[], size)

  swap(a[size-1], a[0])
  size = size - 1
  SINK(a, 0, size)
```

$O(\log n)$: Sink at most to the last level.



Max-PQ Operations

`Max`: Always at index 0.
$\Theta(1)$

`Insert`: Insert at the end of the array and then **swim.**

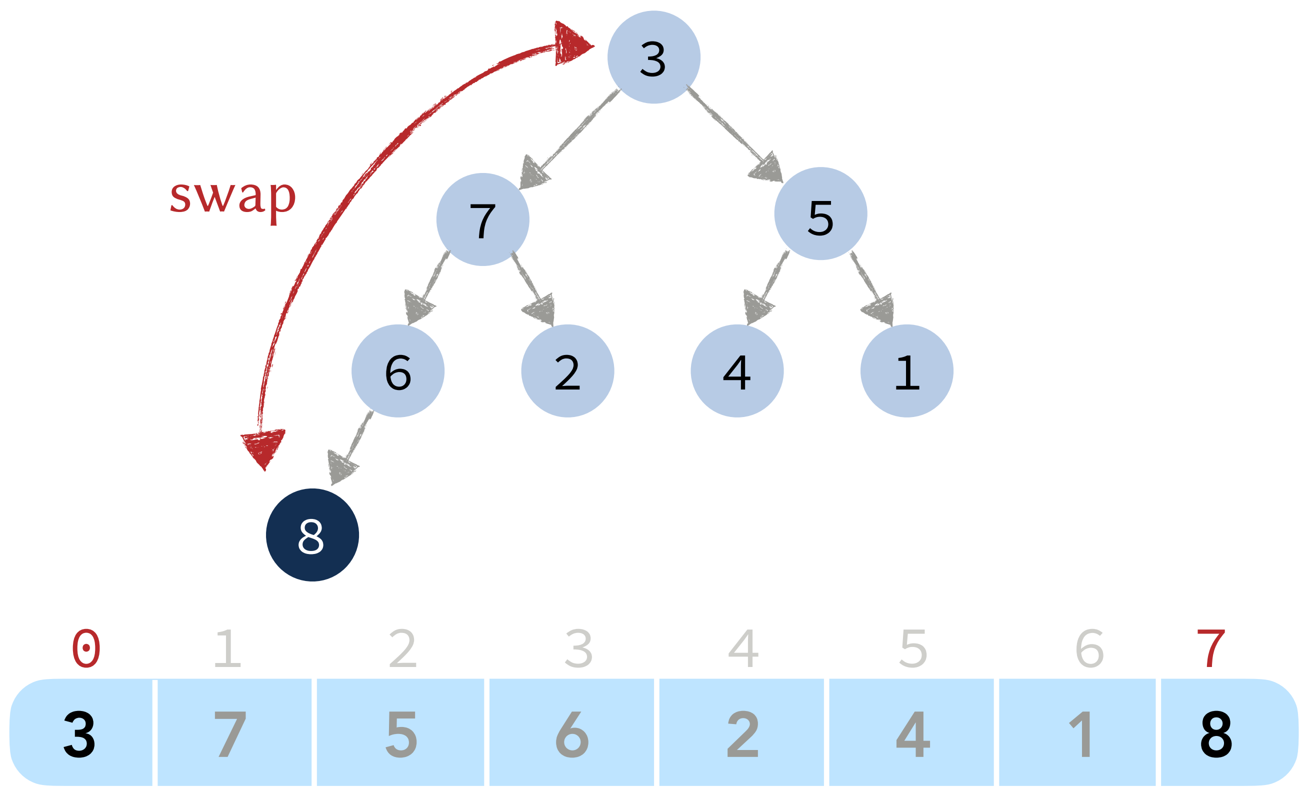
```
INSERT(a[], k, size)

  a[size] = k
  size = size + 1
  SWIM(a, size-1, size)
```

$O(\log n)$: Swim at most to the root.

`Del-Max`: Swap the last element with the element at index `0` and then **sink.**

```
DEL-MAX(a[], size)

  swap(a[size-1], a[0])
  size = size - 1
  SINK(a, 0, size)
```

$O(\log n)$: Sink at most to the last level.



Max-PQ Operations

Max: Always at index 0.
$\Theta(1)$

Insert: Insert at the end of the array and then **swim.**

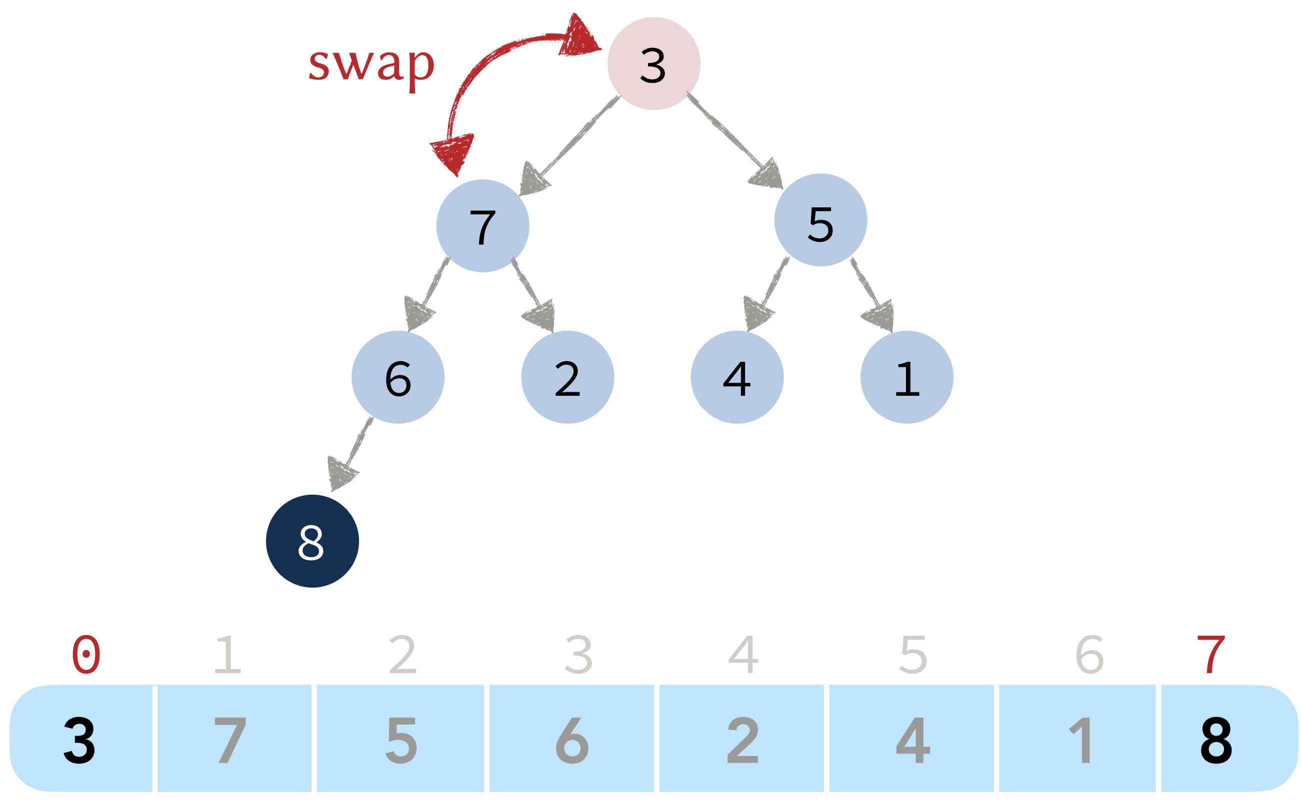
```
INSERT(a[], k, size)

  a[size] = k
  size = size + 1
  SWIM(a, size-1, size)
```

$O(\log n)$: Swim at most to the root.

Del-Max: Swap the last element with the element at index `0` and then **sink.**

```
DEL-MAX(a[], size)

  swap(a[size-1], a[0])
  size = size - 1
  SINK(a, 0, size)
```

$O(\log n)$: Sink at most to the last level.



Max-PQ Operations

Max: Always at index 0.
$\Theta(1)$

Insert: Insert at the end of the array and then **swim.**

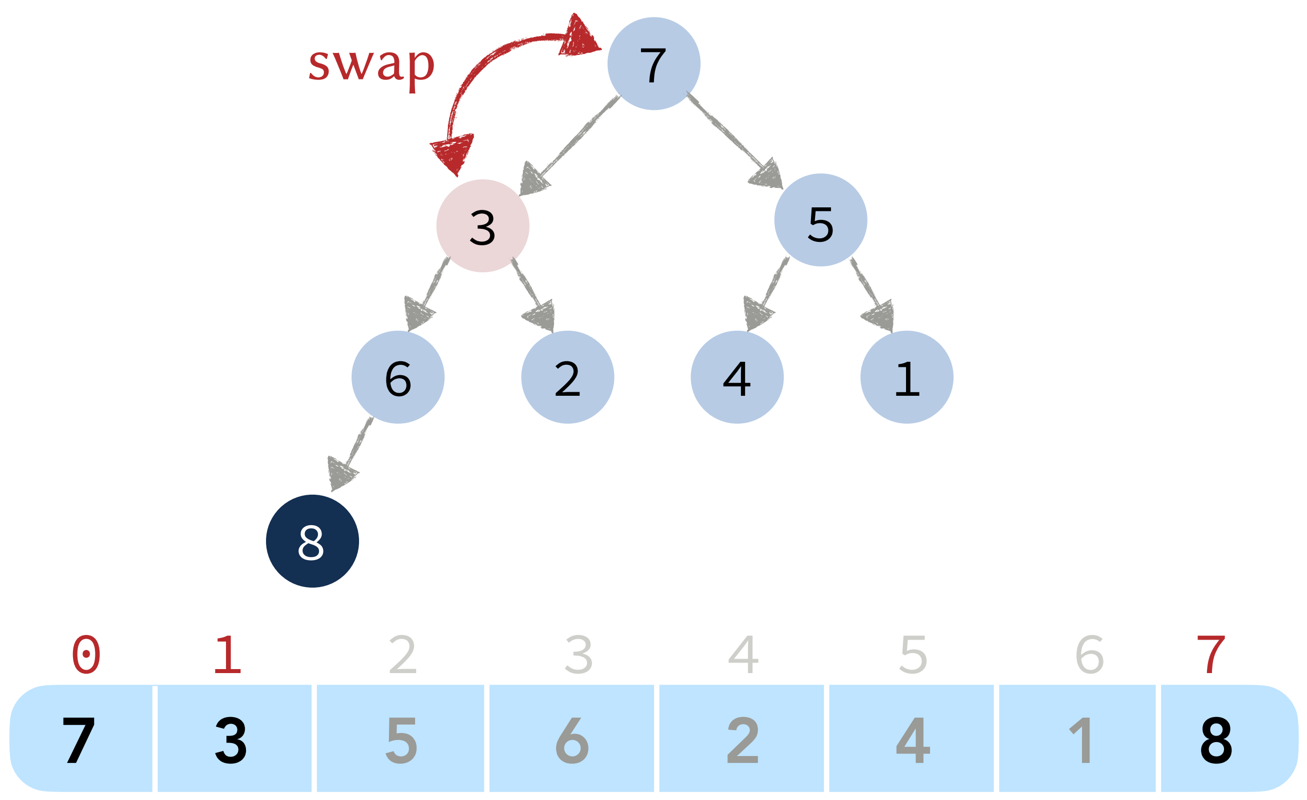
```
INSERT(a[], k, size)

  a[size] = k
  size = size + 1
  SWIM(a, size-1, size)
```

$O(\log n)$: Swim at most to the root.

Del-Max: Swap the last element with the element at index 0 and then **sink.**

```
DEL-MAX(a[], size)

  swap(a[size-1], a[0])
  size = size - 1
  SINK(a, 0, size)
```

$O(\log n)$: Sink at most to the last level.



Max-PQ Operations

`Max:` Always at index 0.
$\Theta(1)$

`Insert:` Insert at the end of the array and then **swim.**

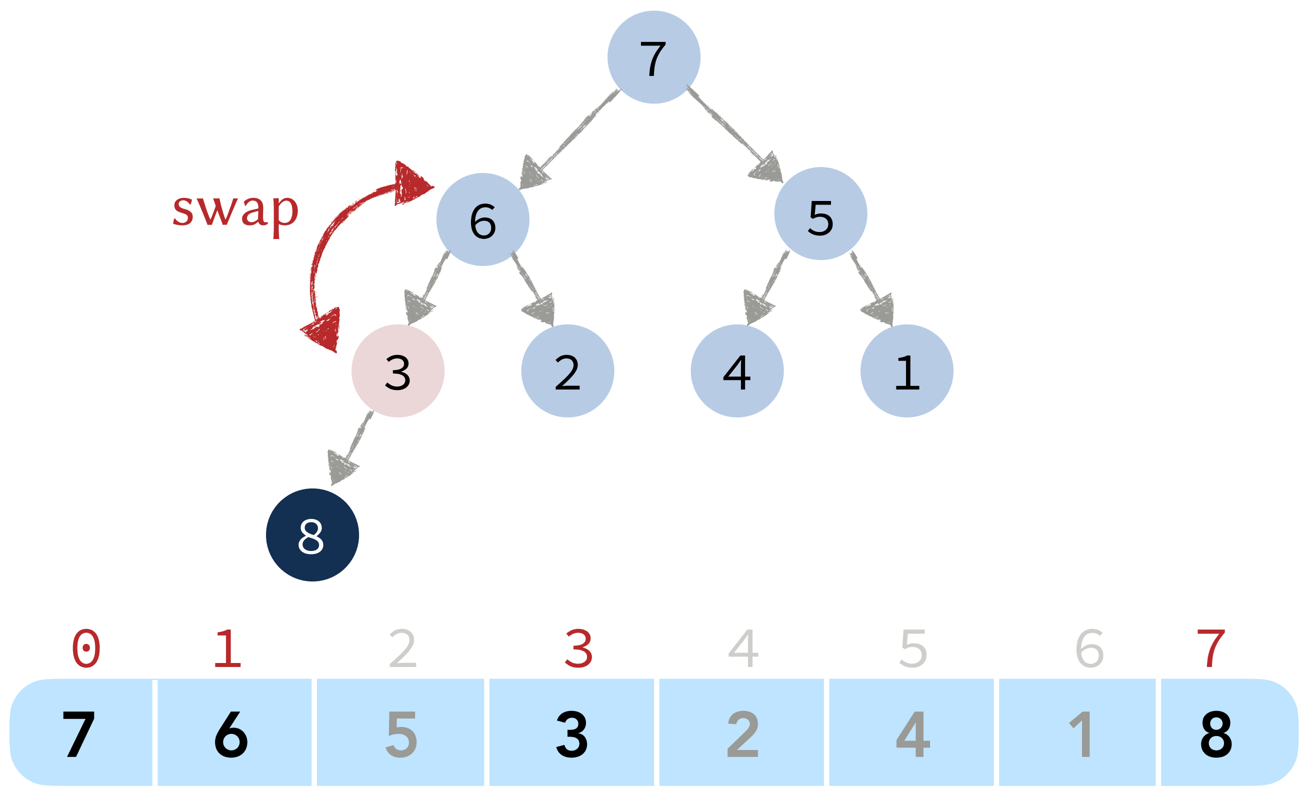
```
INSERT(a[], k, size)

  a[size] = k
  size = size + 1
  SWIM(a, size-1, size)
```

$O(\log n)$: Swim at most to the root.

`Del-Max`: Swap the last element with the element at index `0` and then **sink.**

```
DEL-MAX(a[], size)

  swap(a[size-1], a[0])
  size = size - 1
  SINK(a, 0, size)
```

$O(\log n)$: Sink at most to the last level.



Max-PQ Operations

Max: Always at index 0.
$\Theta(1)$

Insert: Insert at the end of the array and then **swim.**

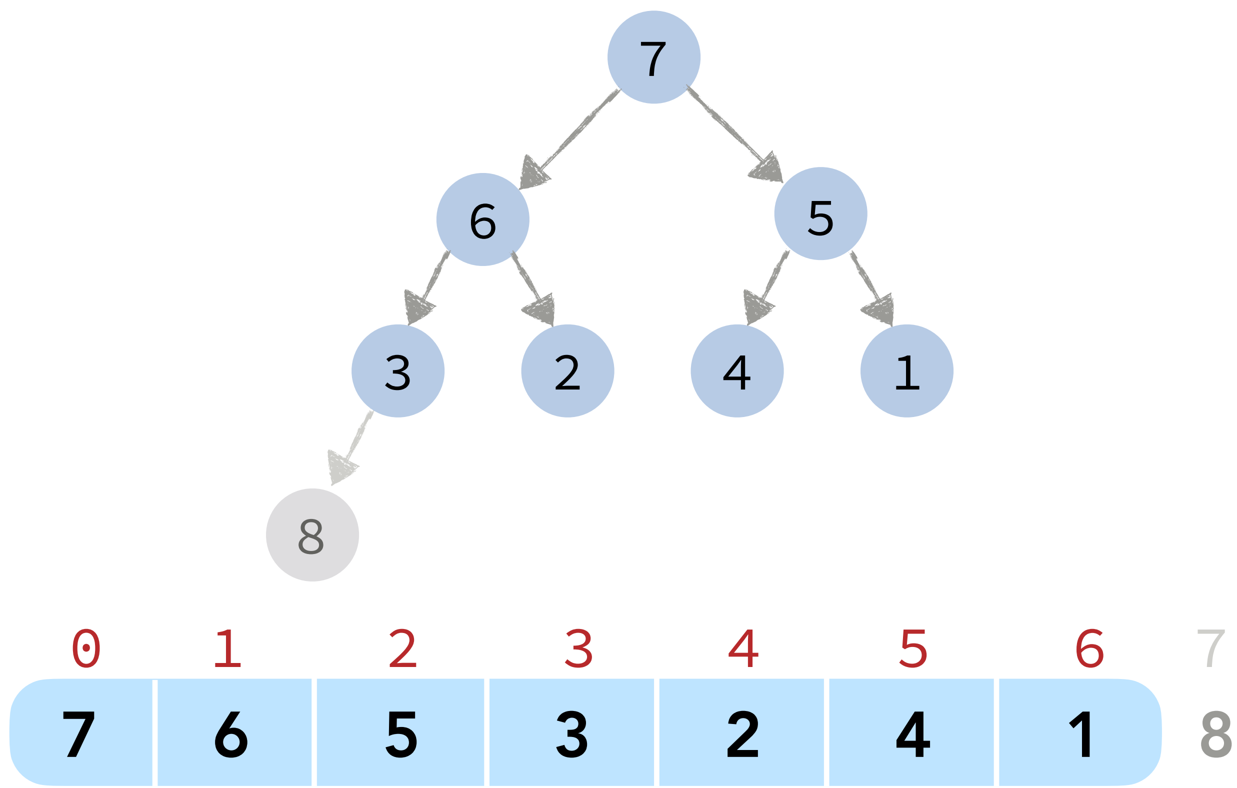
```
INSERT(a[], k, size)

  a[size] = k
  size = size + 1
  SWIM(a, size-1, size)
```

$O(\log n)$: Swim at most to the root.

Del-Max: Swap the last element with the element at index `0` and then **sink.**

```
DEL-MAX(a[], size)

  swap(a[size-1], a[0])
  size = size - 1
  SINK(a, 0, size)
```

$O(\log n)$: Sink at most to the last level.



Quiz # 1

Implement a max-PQ that supports the following operation:

`del-Random`: Removes a random element from the priority queue.

Quiz # 1

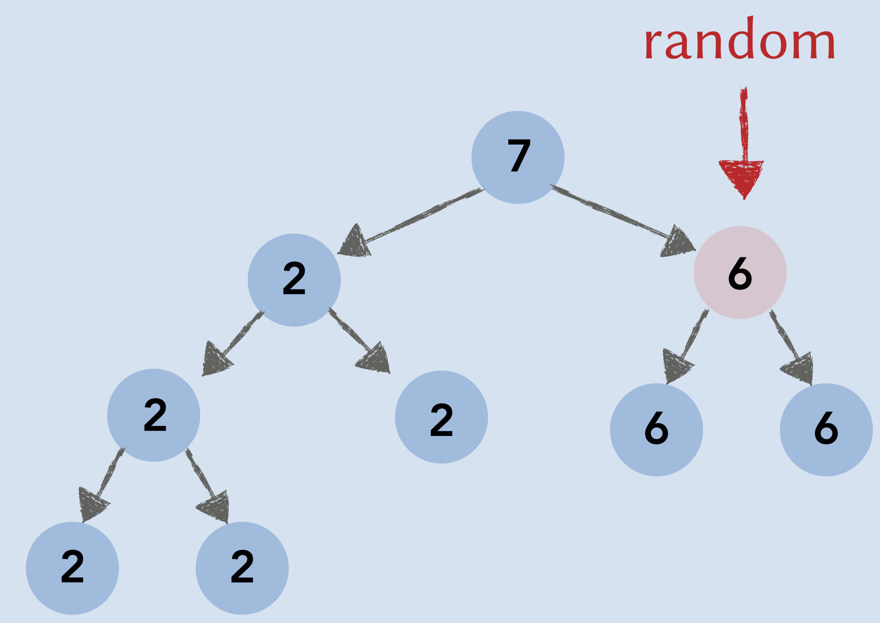
Implement a max-PQ that supports the following operation:

`del-Random`: Removes a random element from the priority queue.

Answer.

```
DEL-RANDOM(a[], size)

 k = random index in [0, size-1]
 swap(a[k], a[size-1])
 size = size-1
 SINK(a, k, size)
 SWIM(a, k, size)
```

Example 1.



Quiz # 1

Implement a max-PQ that supports the following operation:

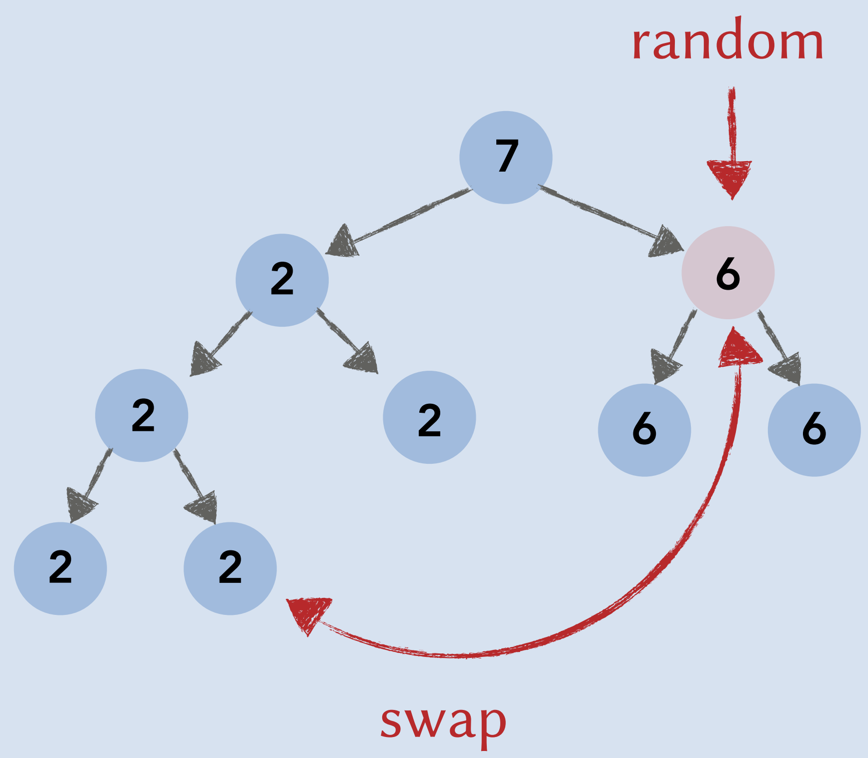
del-Random: Removes a random element from the priority queue.

Answer.

DEL-RANDOM(a[], size)

```
k = random index in [0, size-1]
swap(a[k], a[size-1])
size = size-1
SINK(a, k, size)
SWIM(a, k, size)
```

Example 1.



Quiz # 1

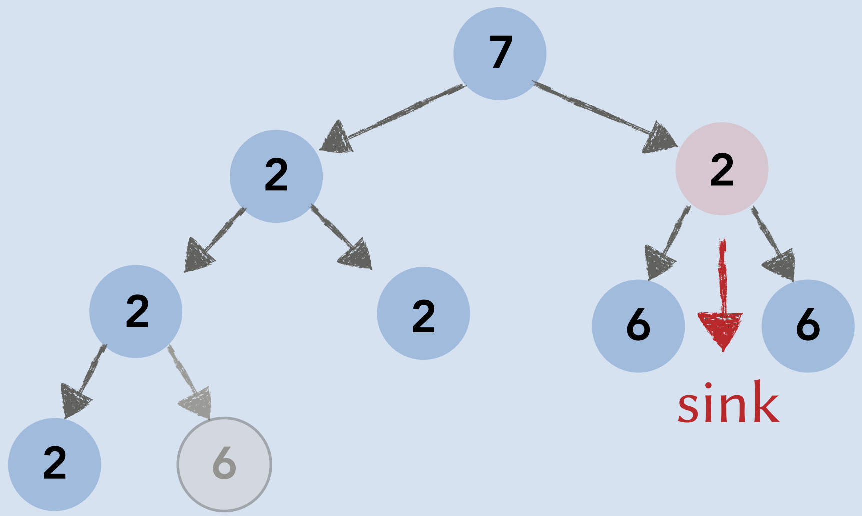
Implement a max-PQ that supports the following operation:

del-Random: Removes a random element from the priority queue.

Answer.

```
DEL-RANDOM(a[], size)

k = random index in [0, size-1]
swap(a[k], a[size-1])
size = size-1
SINK(a, k, size)
SWIM(a, k, size)
```

Example 1.



Quiz # 1

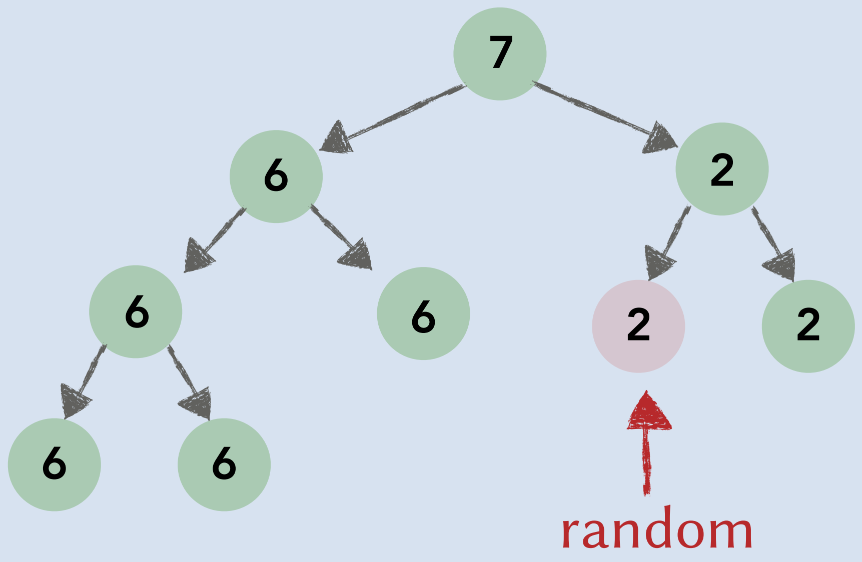
Implement a max-PQ that supports the following operation:

del-Random: Removes a random element from the priority queue.

Answer.

```
DEL-RANDOM(a[], size)

k = random index in [0, size-1]
swap(a[k], a[size-1])
size = size-1
SINK(a, k, size)
SWIM(a, k, size)
```

Example 2.



Quiz # 1

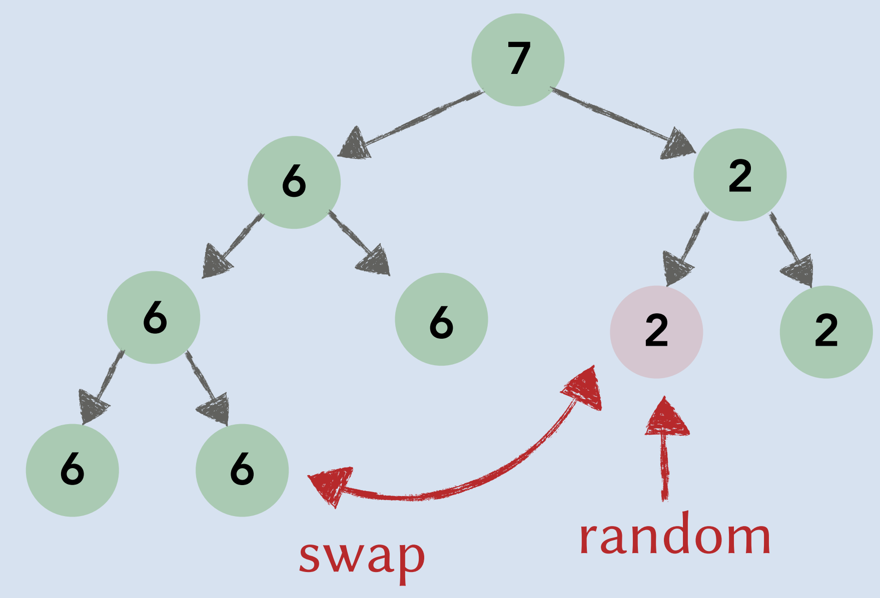
Implement a max-PQ that supports the following operation:

del-Random: Removes a random element from the priority queue.

Answer.

```
DEL-RANDOM(a[], size)

k = random index in [0, size-1]
swap(a[k], a[size-1])
size = size-1
SINK(a, k, size)
SWIM(a, k, size)
```

Example 2.



Quiz # 1

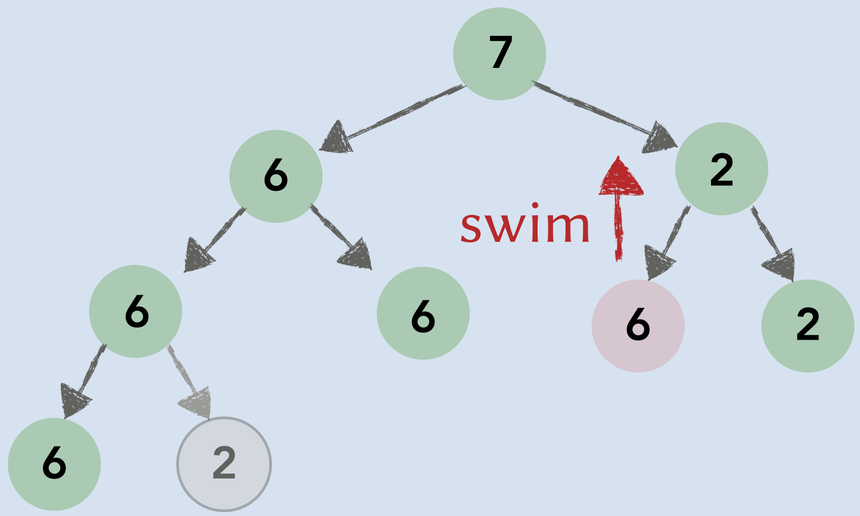
Implement a max-PQ that supports the following operation:

`del-Random`: Removes a random element from the priority queue.

Answer.

```
DEL-RANDOM(a[], size)

k = random index in [0, size-1]
swap(a[k], a[size-1])
size = size-1
SINK(a, k, size)
SWIM(a, k, size)
```

Example 2.



Heapsort

Heapsort: Naive Implementation

HEAP-SORT(a[], size)

```
heap ← An empty max heap
for i = 0 → n-1:
    heap.INSERT(a[i])

for i = n-1 → 0:
    a[i] = heap.MAX()
    heap.DELETE-MAX()
```

Heapsort: Naive Implementation

```
HEAP-SORT(a[], size)

heap ← An empty max heap
for i = 0 → n-1:
    heap.INSERT(a[i])

for i = n-1 → 0:
    a[i] = heap.MAX()
    heap.DELETE-MAX()
```

1 insert all the array elements into a max-heap

Heapsort: Naive Implementation

```
HEAP-SORT(a[], size)

heap ← An empty max heap
for i = 0 → n-1:
    heap.INSERT(a[i])

for i = n-1 → 0:
    a[i] = heap.MAX()
    heap.DELETE-MAX()
```

1. insert all the array elements into a max-heap
2. copy all the elements back from the heap to the array (in order)

Heapsort: Naive Implementation

```
HEAP-SORT(a[], size)

heap ← An empty max heap
for i = 0 → n-1:
    heap.INSERT(a[i])

for i = n-1 → 0:
    a[i] = heap.MAX()
    heap.DELETE-MAX()
```

1. insert all the array elements into a max-heap
2. copy all the elements back from the heap to the array (in order)

Running Time. (number of compares in the worst case)

- Step 1. $\log_2(1) + \log_2(2) + \log_2(3) + \ \ldots \ + \log_2(n-1) \leq \log_2(n!)$

Heapsort: Naive Implementation

```
HEAP-SORT(a[], size)

heap ← An empty max heap
for i = 0 → n-1:
    heap.INSERT(a[i])

for i = n-1 → 0:
    a[i] = heap.MAX()
    heap.DELETE-MAX()
```

1. insert all the array elements into a max-heap
2. copy all the elements back from the heap to the array (in order)

Running Time. (number of compares in the worst case)

- Step 1. $\log_2(1) + \log_2(2) + \log_2(3) + \ \dots\ + \log_2(n-1) \leq \log_2(n!)$

insert the second element into a heap of size 1

insert the last element into a heap of size n -1

Heapsort: Naive Implementation

```
HEAP-SORT(a[], size)

heap ← An empty max heap
for i = 0 → n-1:
    heap.INSERT(a[i])

for i = n-1 → 0:
    a[i] = heap.MAX()
    heap.DELETE-MAX()
```

1. insert all the array elements into a max-heap
2. copy all the elements back from the heap to the array (in order)

Running Time. (number of compares in the worst case)

- Step 1. $\log_2(1) + \log_2(2) + \log_2(3) + \ \ldots \ + \log_2(n-1) \le \log_2(n!) = O(n \log n)$
- Step 2. $2 \times (\log_2(n-1) + \log_2(n-2) + \log_2(n-3) + \ \ldots \ + \log_2(1))$
  $\le 2 \times \log_2(n!)$

Heapsort: Naive Implementation

```
HEAP-SORT(a[], size)

heap ← An empty max heap
for i = 0 → n-1:
    heap.INSERT(a[i])

for i = n-1 → 0:
    a[i] = heap.MAX()
    heap.DELETE-MAX()
```

1. insert all the array elements into a max-heap
2. copy all the elements back from the heap to the array (in order)

Running Time. (number of compares in the worst case)

- Step 1. $\log_2(1) + \log_2(2) + \log_2(3) + \ \ldots\ + \log_2(n-1) \le \log_2(n!) = O(n \log n)$
- Step 2. $2 \times (\log_2(n-1) + \log_2(n-2) + \log_2(n-3) + \ \ldots\ + \log_2(1))$
  $\le 2 \times \log_2(n!) = O(n \log n)$

check the analysis of the **SINK** operation!

Heapsort: Naive Implementation

```
HEAP-SORT(a[], size)

heap ← An empty max heap
for i = 0 → n-1:
    heap.INSERT(a[i])

for i = n-1 → 0:
    a[i] = heap.MAX()
    heap.DELETE-MAX()
```

1. insert all the array elements into a max-heap
2. copy all the elements back from the heap to the array (in order)

Running Time. (number of compares in the worst case)

- Step 1. $\log_2(1) + \log_2(2) + \log_2(3) + \ \ldots\ + \log_2(n-1) \leq \log_2(n!) = O(n \log n)$
- Step 2. $2 \times (\log_2(n-1) + \log_2(n-2) + \log_2(n-3) + \ \ldots\ + \log_2(1))$
  $\leq 2 \times \log_2(n!) = O(n \log n)$
- Total. $O(n \log n)$

Heapsort: Naive Implementation

```
HEAP-SORT(a[], size)

heap ← An empty max heap
for i = 0 → n-1:
    heap.INSERT(a[i])

for i = n-1 → 0:
    a[i] = heap.MAX()
    heap.DELETE-MAX()
```

1 insert all the array elements into a max-heap

2 copy all the elements back from the heap to the array (in order)

Running Time. (number of compares in the worst case)

- Step 1. $\log_2(1) + \log_2(2) + \log_2(3) + \ \dots \ + \log_2(n-1) \leq \log_2(n!) = O(n \log n)$
- Step 2. $2 \times (\log_2(n-1) + \log_2(n-2) + \log_2(n-3) + \ \dots \ + \log_2(1))$
  $\leq 2 \times \log_2(n!) = O(n \log n)$
- Total. $O(n \log n)$

🤔 **Can we do better?**

Not asymptotically, but we can still improve the actual running time!

Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

  CONSTRUCT-HEAP(a, size)

  while (size > 1):
      swap(a[0], a[size-1])
      size = size-1
      SINK(a, 0, size)
```

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

random array

size-1

Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1 construct a max-heap **in-place** (**convert** the array to become a heap)

How? stay tuned!

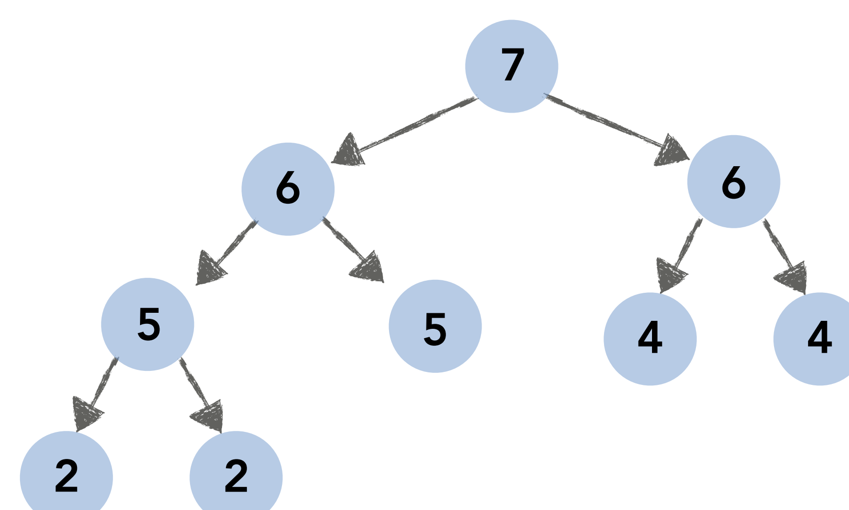
0	1	2	3	4	5	6	7	8
6	7	6	2	4	4	5	2	5

random array — size-1

CONSTRUCT-HEAP()

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

max-heap — size-1



Heapsort: A Better Implementation

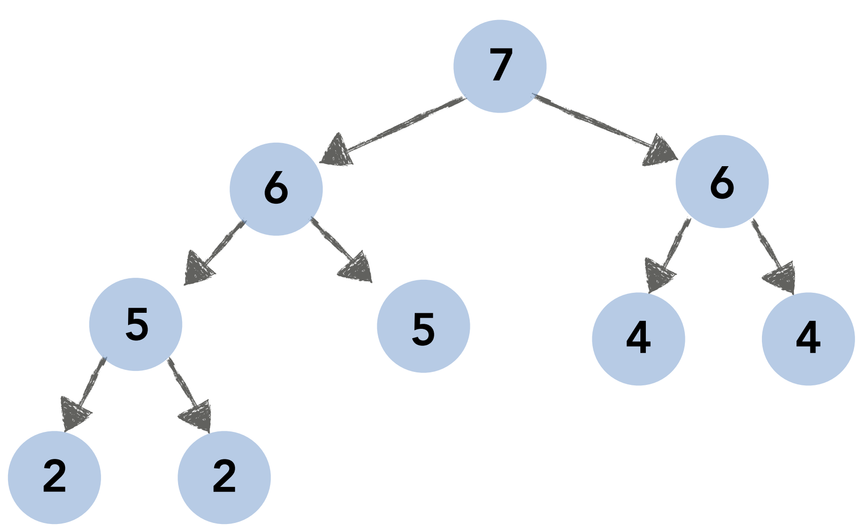
```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1. construct a max-heap in-place (change the array to become a heap)
2. repeatedly place the next maximum in its right position

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

max-heap

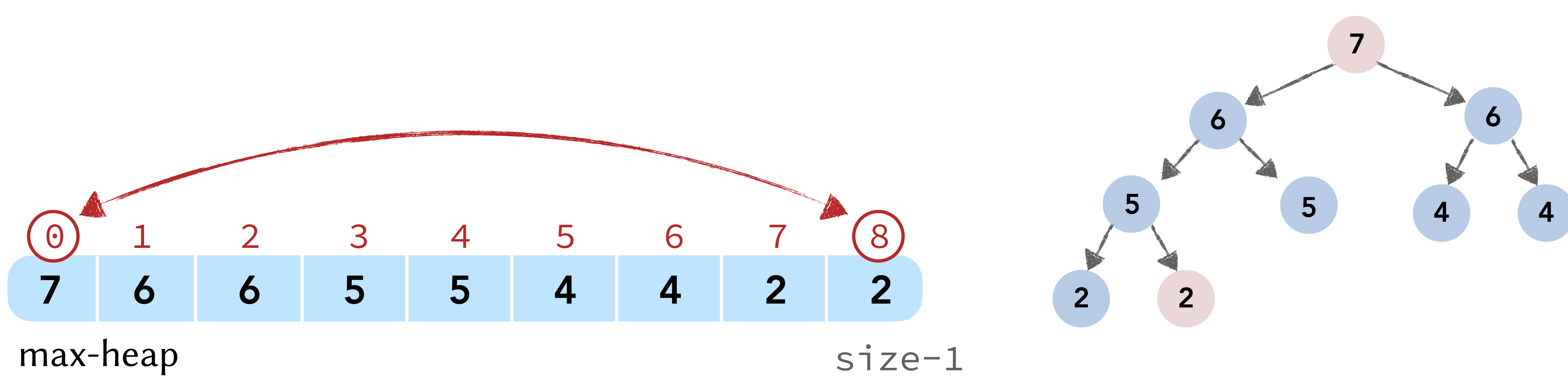
size-1



Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

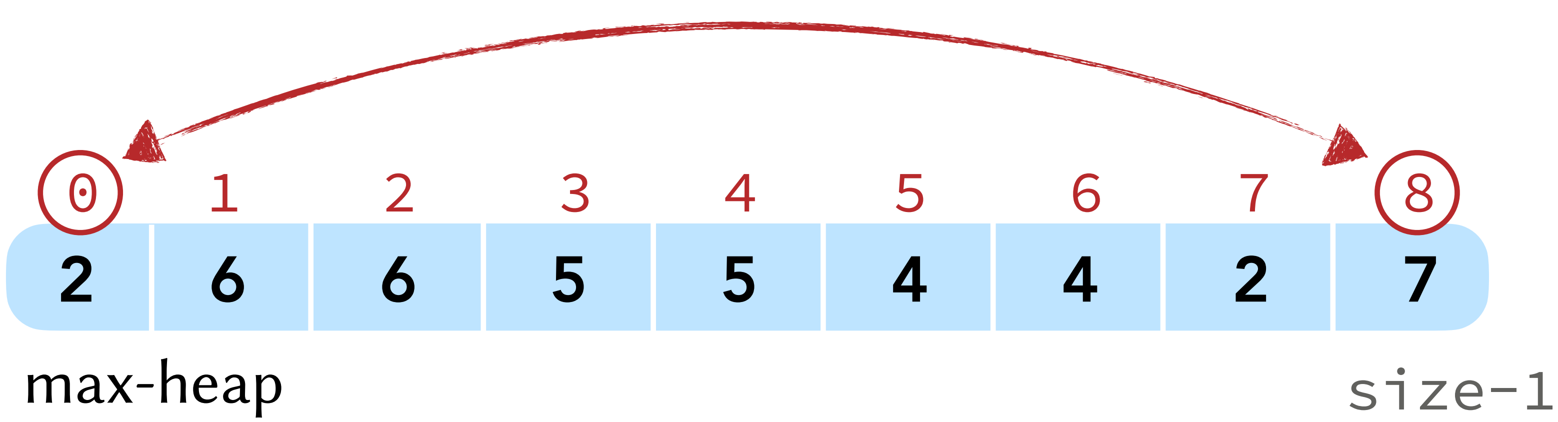
1. construct a max-heap in-place (change the array to become a heap)
2. repeatedly place the next maximum in its right position

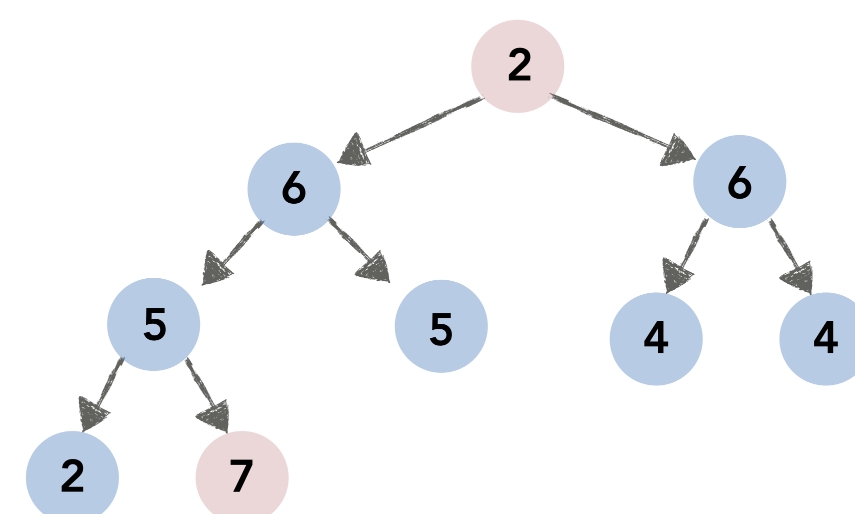


Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1. construct a max-heap in-place (change the array to become a heap)
2. repeatedly place the next maximum in its right position





Heapsort: A Better Implementation

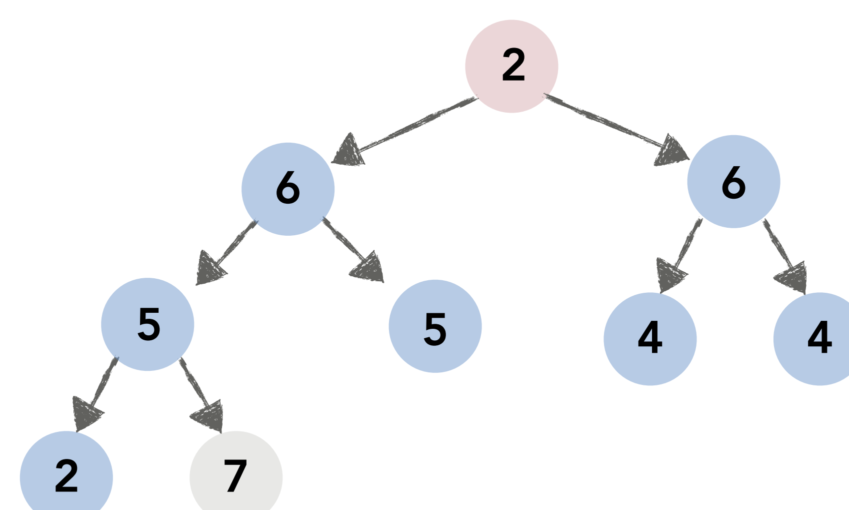
```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1. construct a max-heap in-place (change the array to become a heap)
2. repeatedly place the next maximum in its right position

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

max-heap

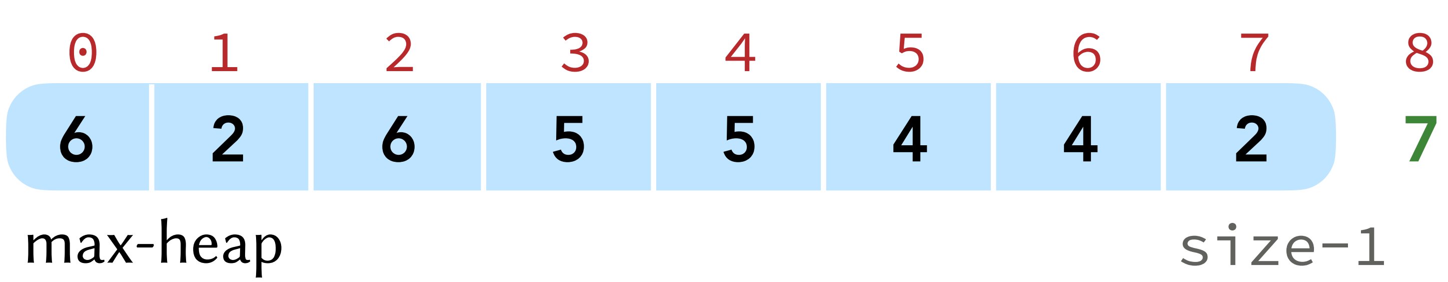
size-1

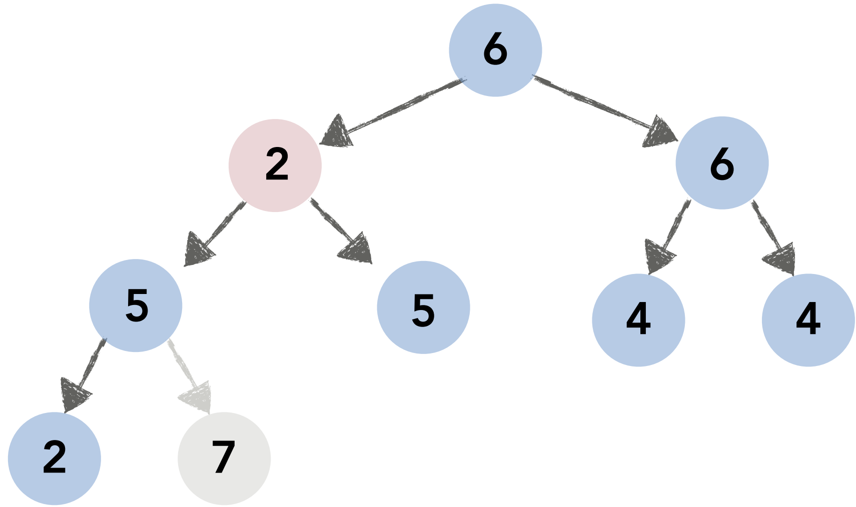


Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1. construct a max-heap in-place (change the array to become a heap)
2. repeatedly place the next maximum in its right position





Heapsort: A Better Implementation

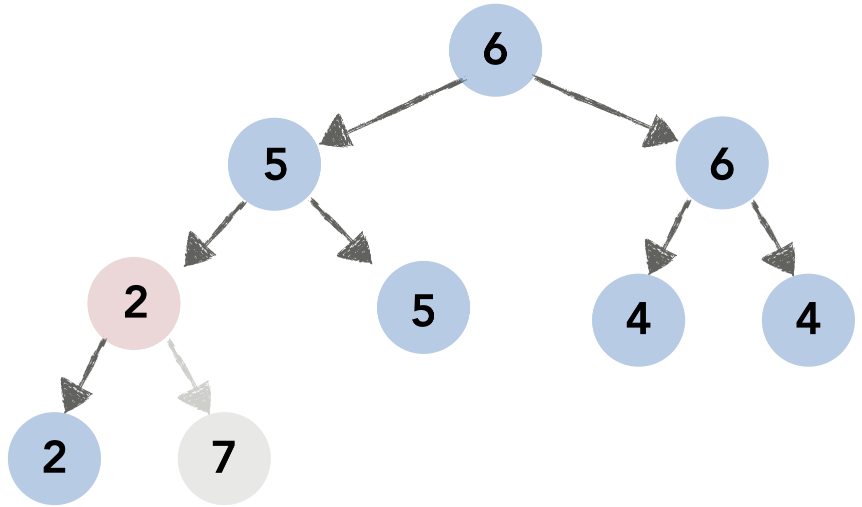
```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1. construct a max-heap in-place (change the array to become a heap)
2. repeatedly place the next maximum in its right position

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

max-heap

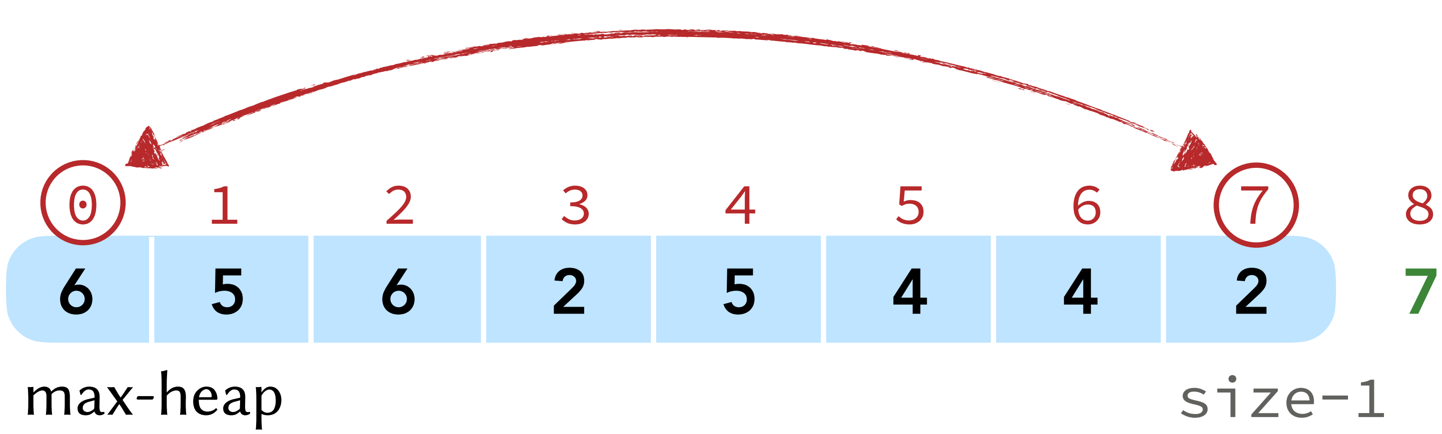
size-1

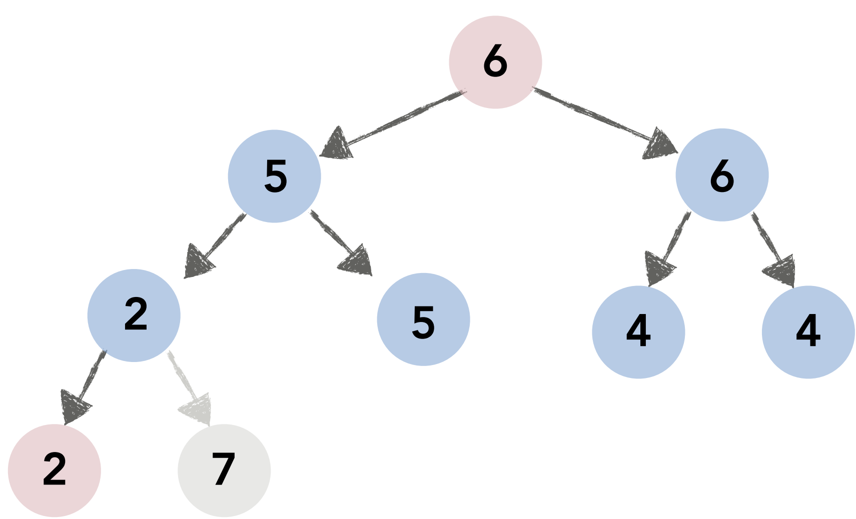


Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1. construct a max-heap in-place (change the array to become a heap)
2. repeatedly place the next maximum in its right position



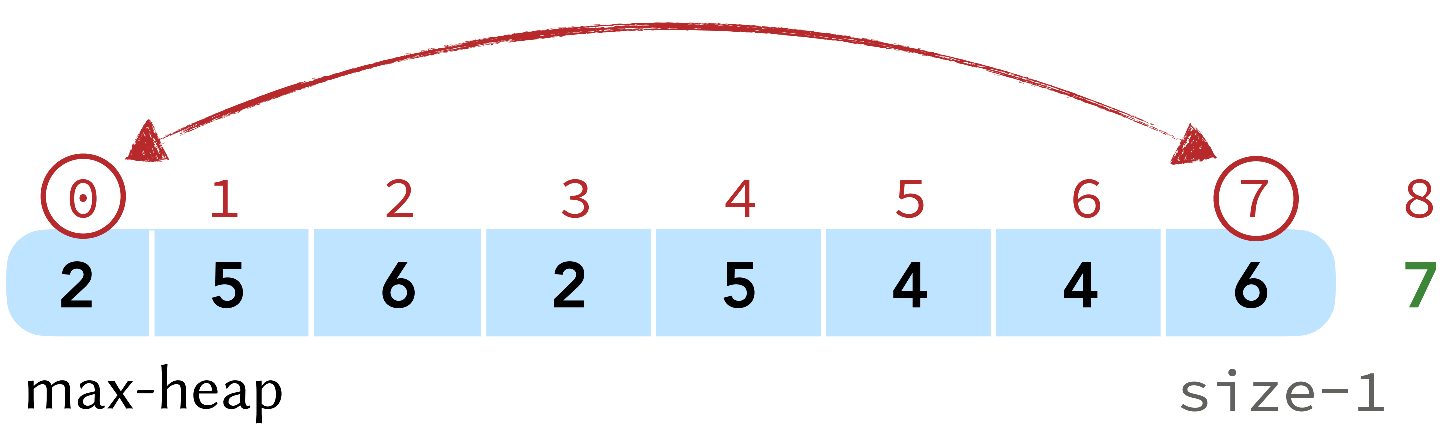


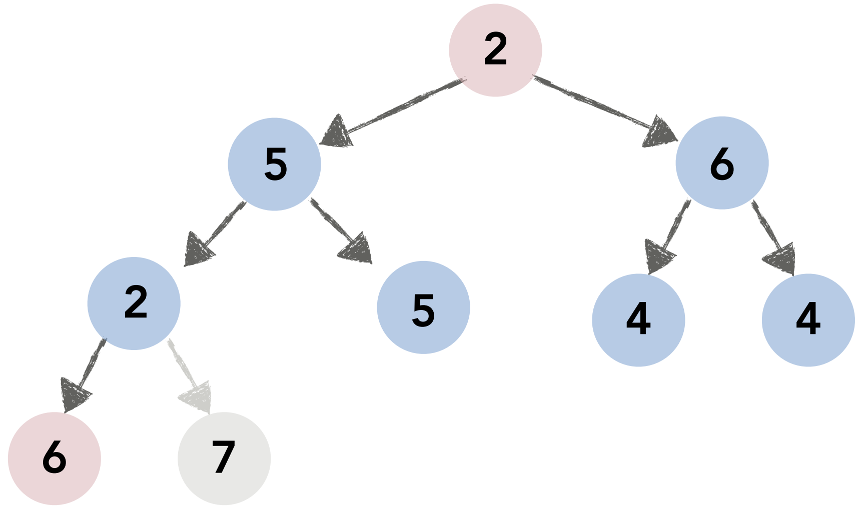
Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

  CONSTRUCT-HEAP(a, size)

  while (size > 1):
      swap(a[0], a[size-1])
      size = size-1
      SINK(a, 0, size)
```

1 construct a max-heap in-place (change the array to become a heap)

2 repeatedly place the next maximum in its right position





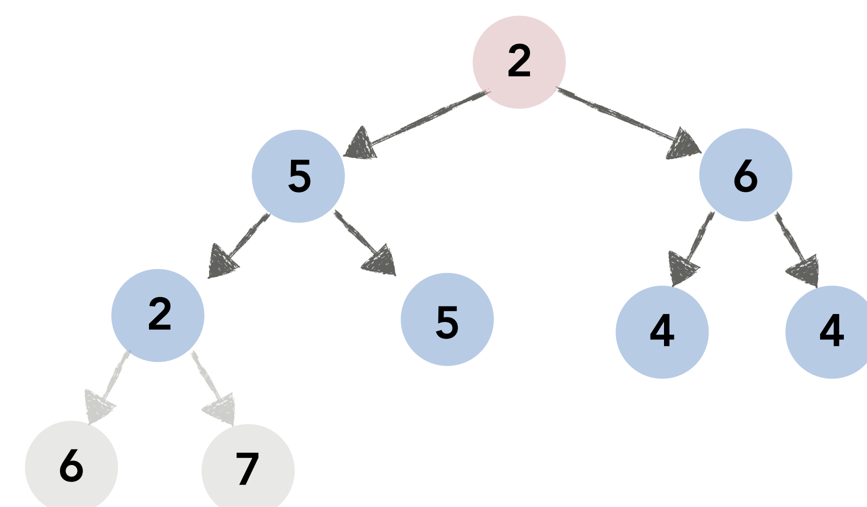
Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1. construct a max-heap in-place (change the array to become a heap)
2. repeatedly place the next maximum in its right position

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

max-heap   size-1



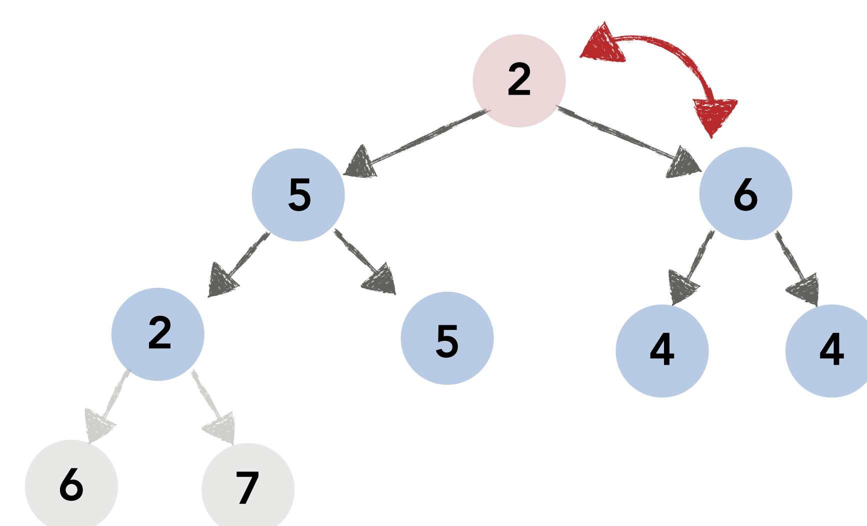
Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1. construct a max-heap in-place (change the array to become a heap)
2. repeatedly place the next maximum in its right position

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

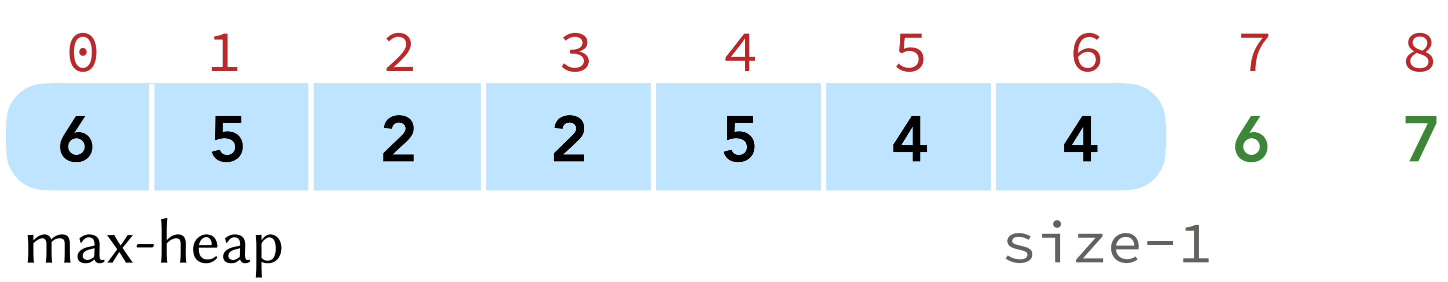
max-heap size-1

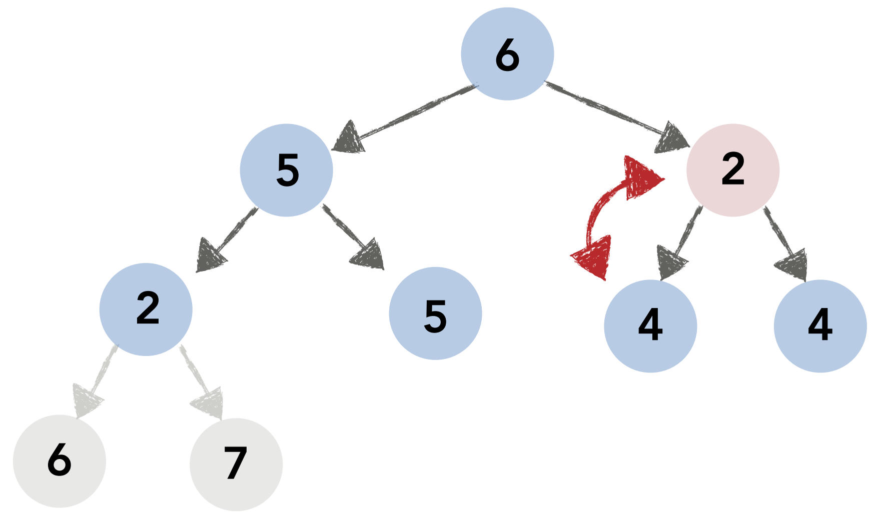


Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1. construct a max-heap in-place (change the array to become a heap)
2. repeatedly place the next maximum in its right position

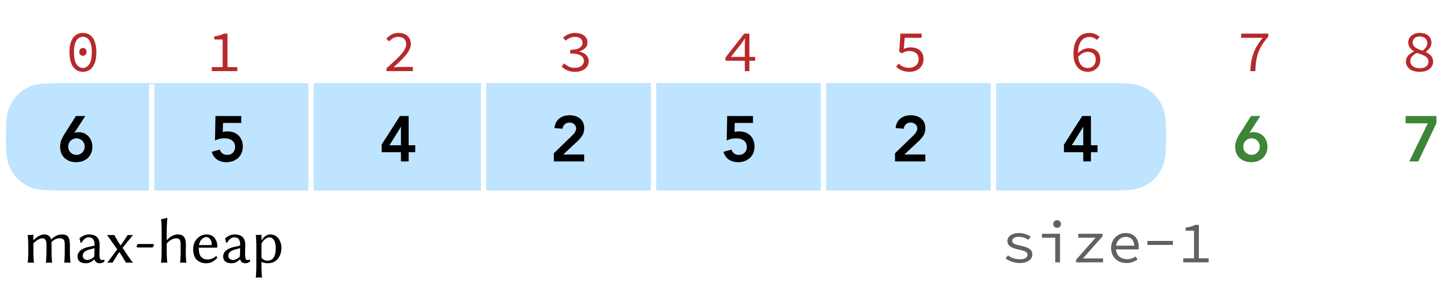


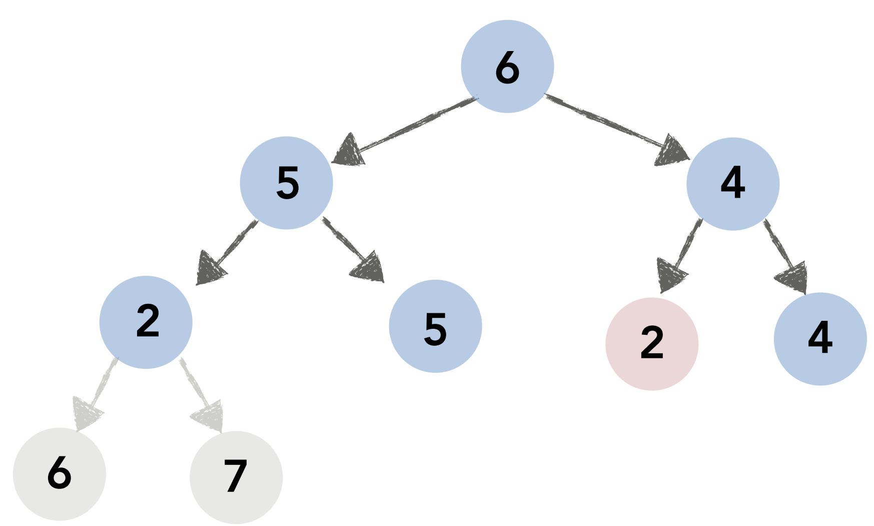


Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1. construct a max-heap in-place (change the array to become a heap)
2. repeatedly place the next maximum in its right position





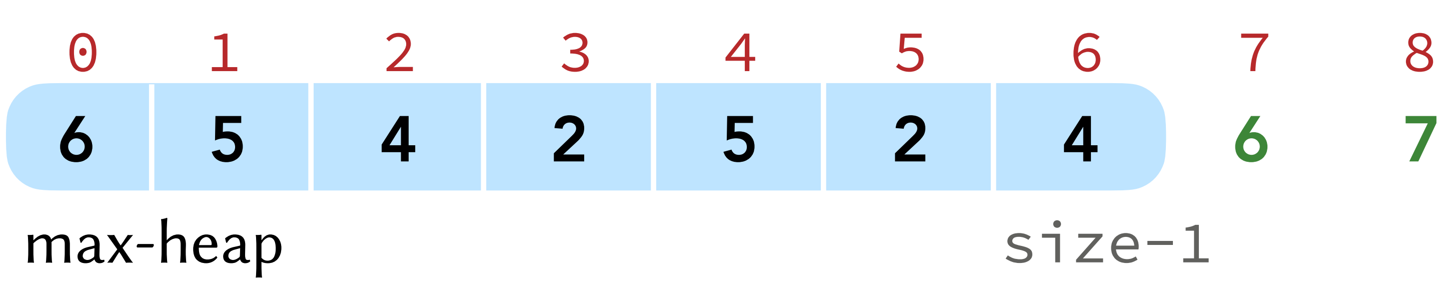
Heapsort: A Better Implementation

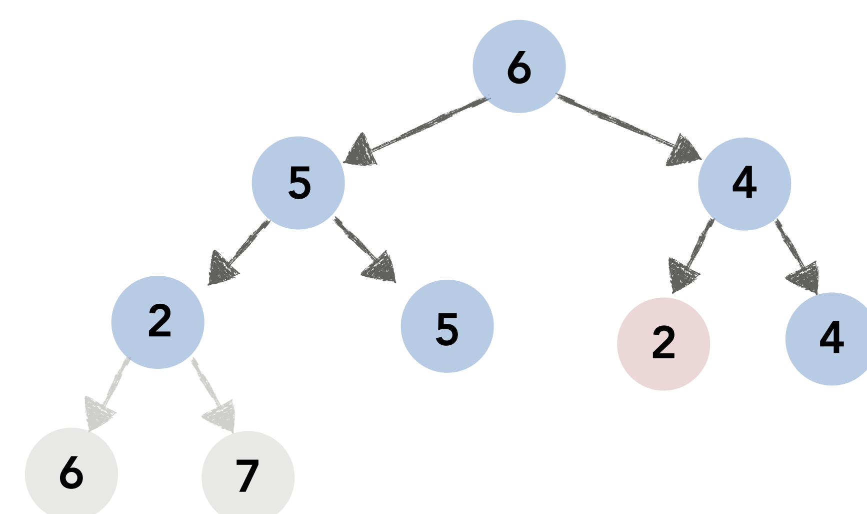
```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1. construct a max-heap in-place (change the array to become a heap)

2. repeatedly place the next maximum in its right position

repeat until all the elements are in their correct positions



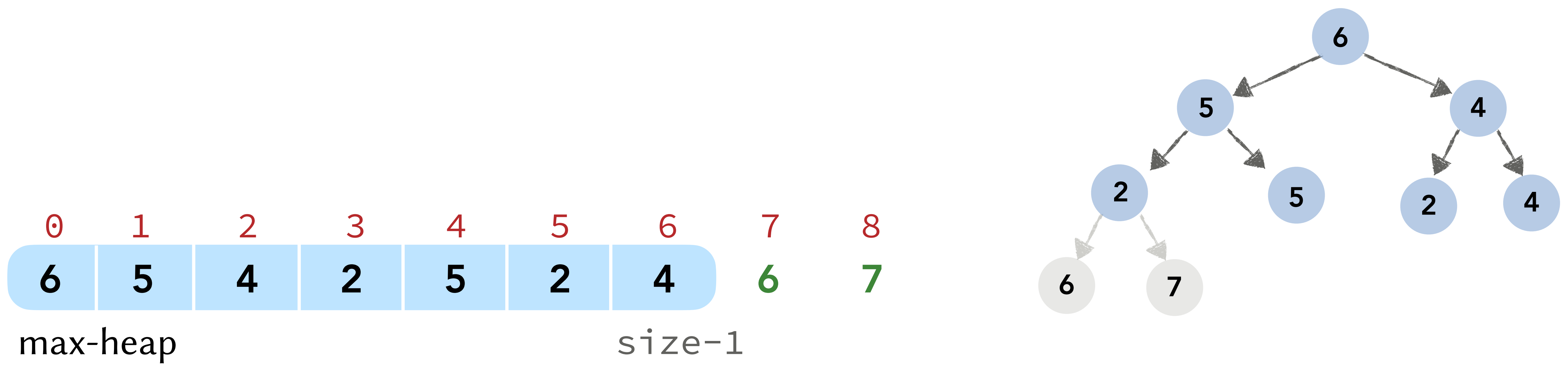


Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1. construct a max-heap in-place (change the array to become a heap)
2. repeatedly place the next maximum in its right position

repeat until all the elements are in their correct positions

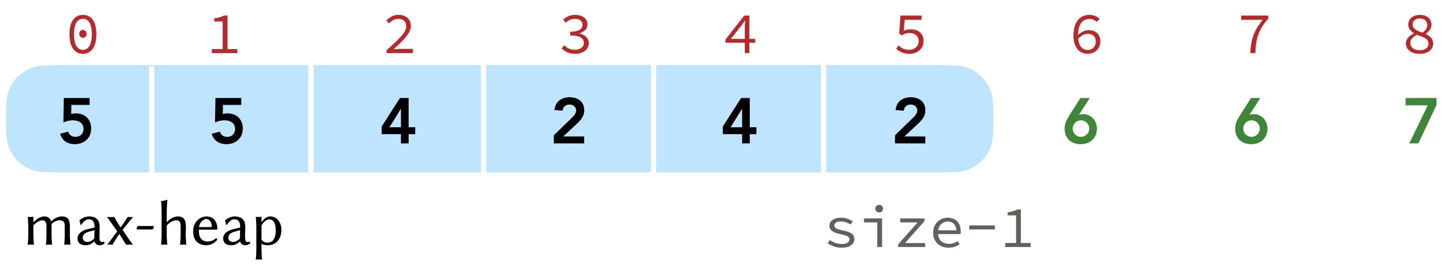


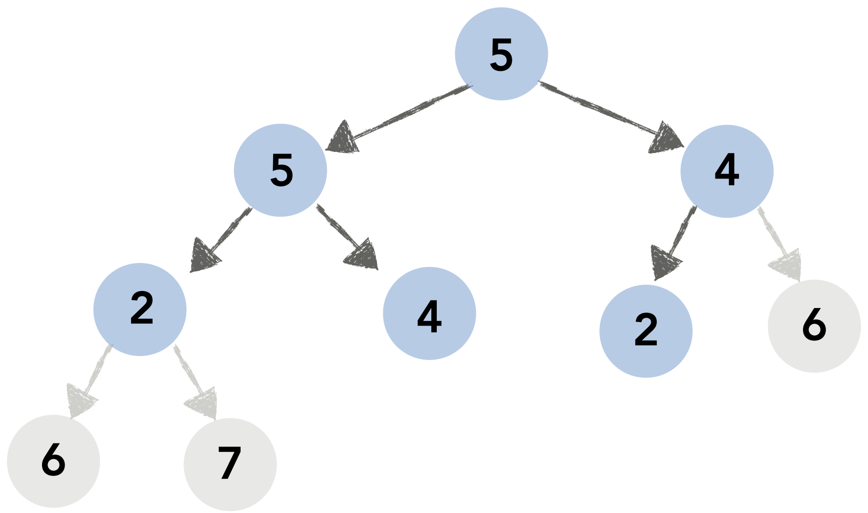
Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1. construct a max-heap in-place (change the array to become a heap)
2. repeatedly place the next maximum in its right position

repeat until all the elements are in their correct positions





Heapsort: A Better Implementation

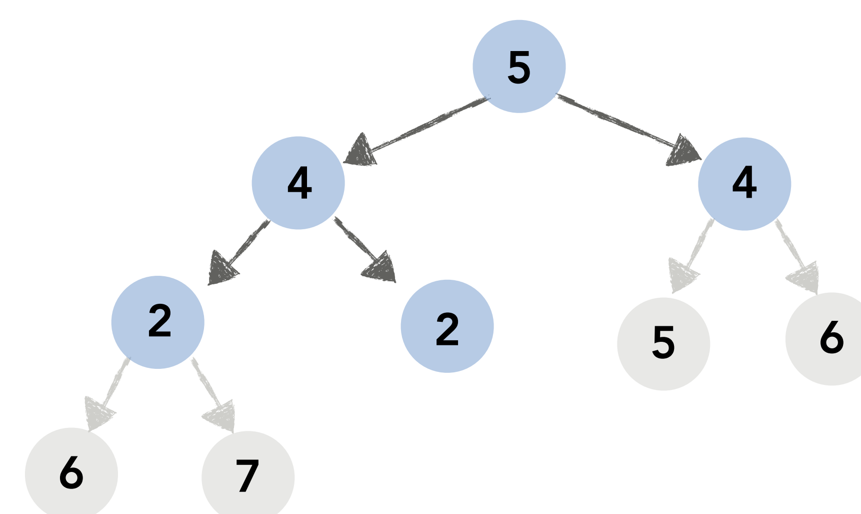
```
HEAP-SORT(a[], size)
    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1. construct a max-heap in-place (change the array to become a heap)
2. repeatedly place the next maximum in its right position

repeat until all the elements are in their correct positions

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

max-heap   size-1



Heapsort: A Better Implementation

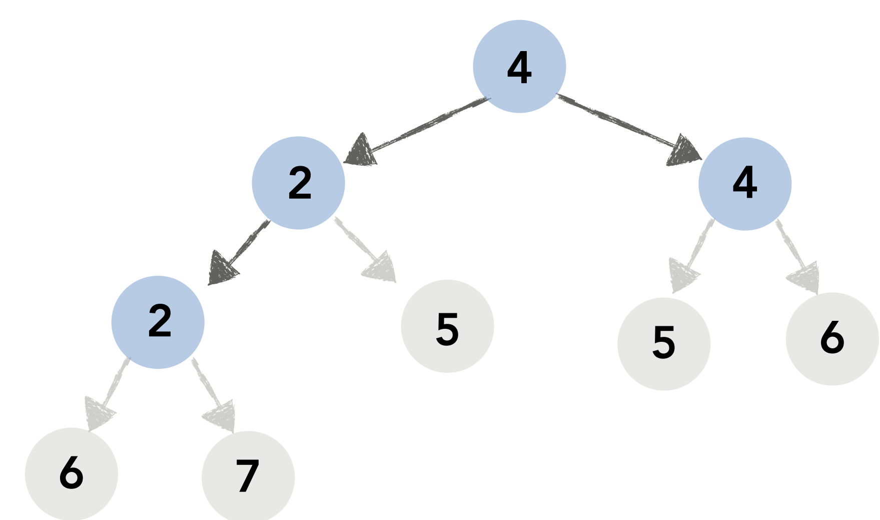
```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1. construct a max-heap in-place (change the array to become a heap)
2. repeatedly place the next maximum in its right position

repeat until all the elements are in their correct positions

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

max-heap   size-1



Heapsort: A Better Implementation

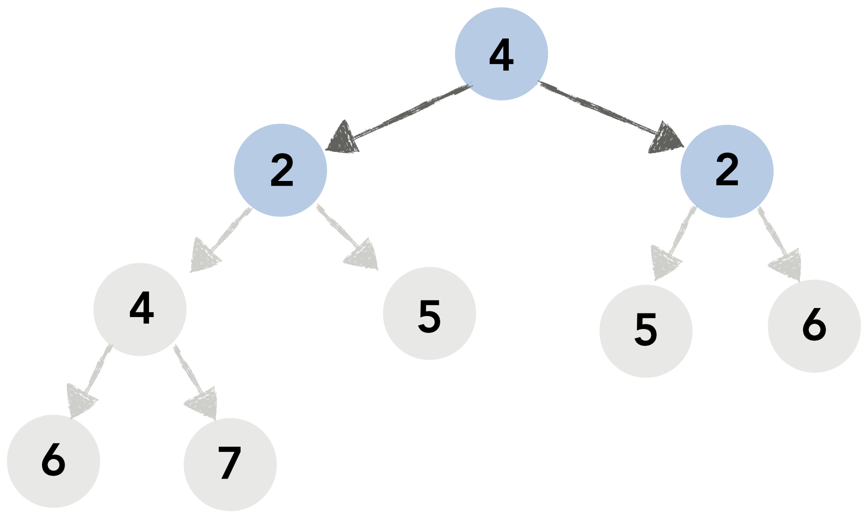
```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1. construct a max-heap in-place (change the array to become a heap)
2. repeatedly place the next maximum in its right position

repeat until all the elements are in their correct positions

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

max-heap size-1



Heapsort: A Better Implementation

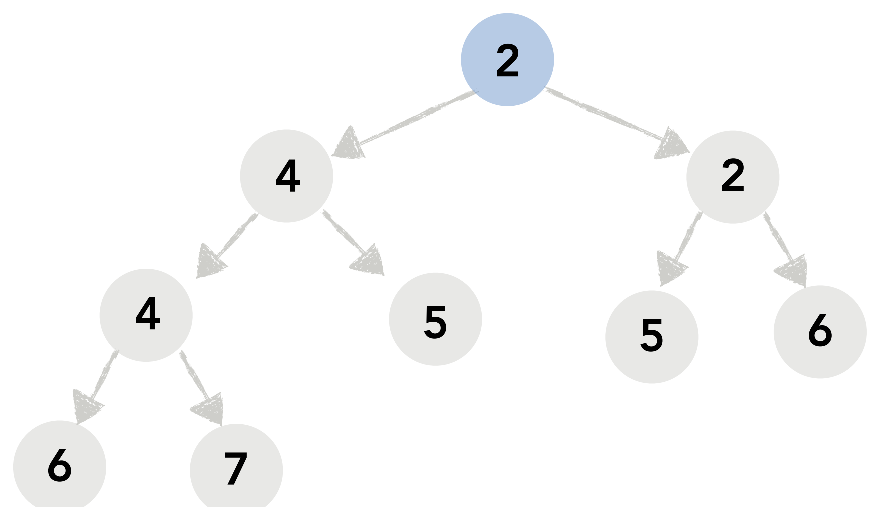
```
HEAP-SORT(a[], size)

    CONSTRUCT-HEAP(a, size)

    while (size > 1):
        swap(a[0], a[size-1])
        size = size-1
        SINK(a, 0, size)
```

1. construct a max-heap in-place (change the array to become a heap)
2. repeatedly place the next maximum in its right position

repeat until all the elements are in their correct positions

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

size-1



Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

CONSTRUCT-HEAP(a, size)

while (size > 1):
    swap(a[0], a[size-1])
    size = size-1
    SINK(a, 0, size)
```

```
CONSTRUCT-HEAP(a, size)

for i = size/2 - 1 → 0:
   SINK(a, i, size)
```

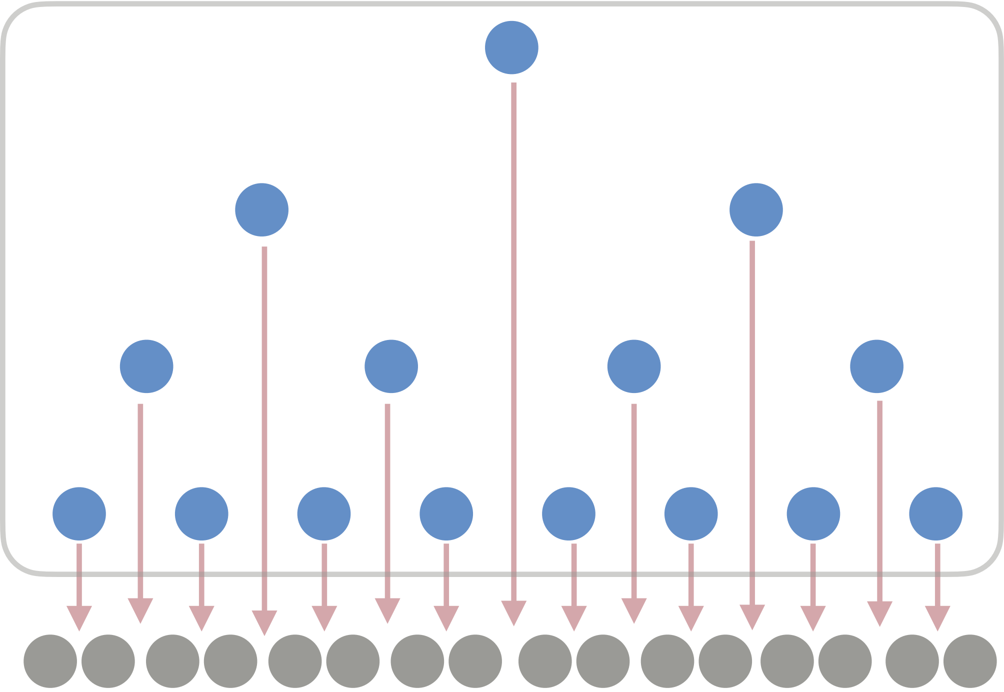
Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

  CONSTRUCT-HEAP(a, size)

  while (size > 1):
     swap(a[0], a[size-1])
     size = size-1
     SINK(a, 0, size)
```

```
CONSTRUCT-HEAP(a, size)

for i = size/2 - 1 → 0:
   SINK(a, i, size)
```

sink all the non-leaf nodes

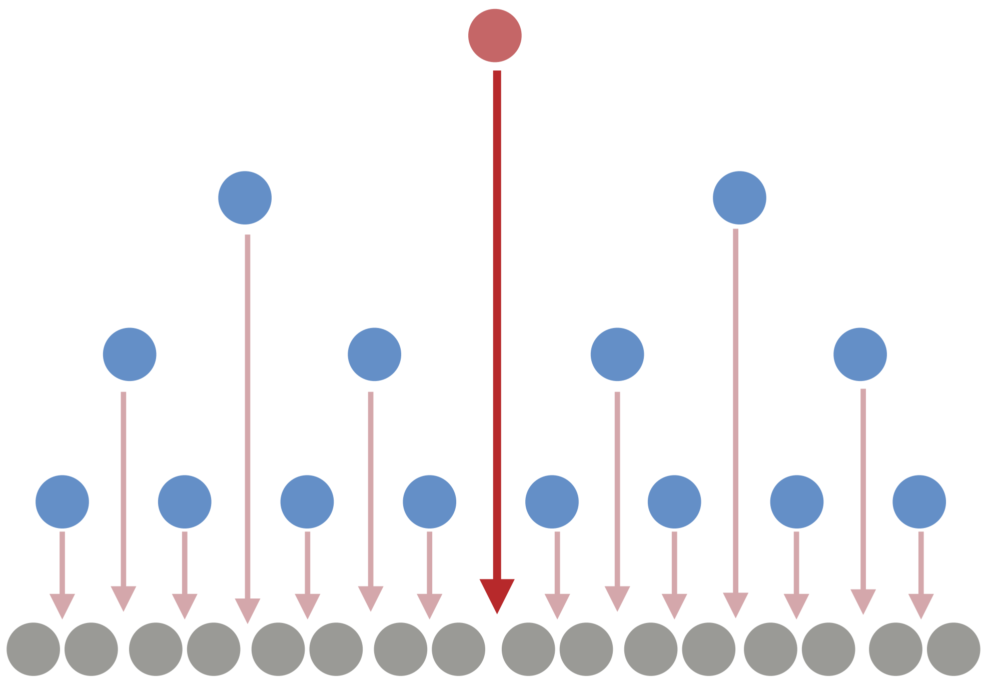
at indices 0 to size/2 - 1



Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

 CONSTRUCT-HEAP(a, size)

 while (size > 1):
    swap(a[0], a[size-1])
    size = size-1
    SINK(a, 0, size)
```

```
CONSTRUCT-HEAP(a, size)

for i = size/2 - 1 → 0:
   SINK(a, i, size)
```



Heap Construction Analysis:

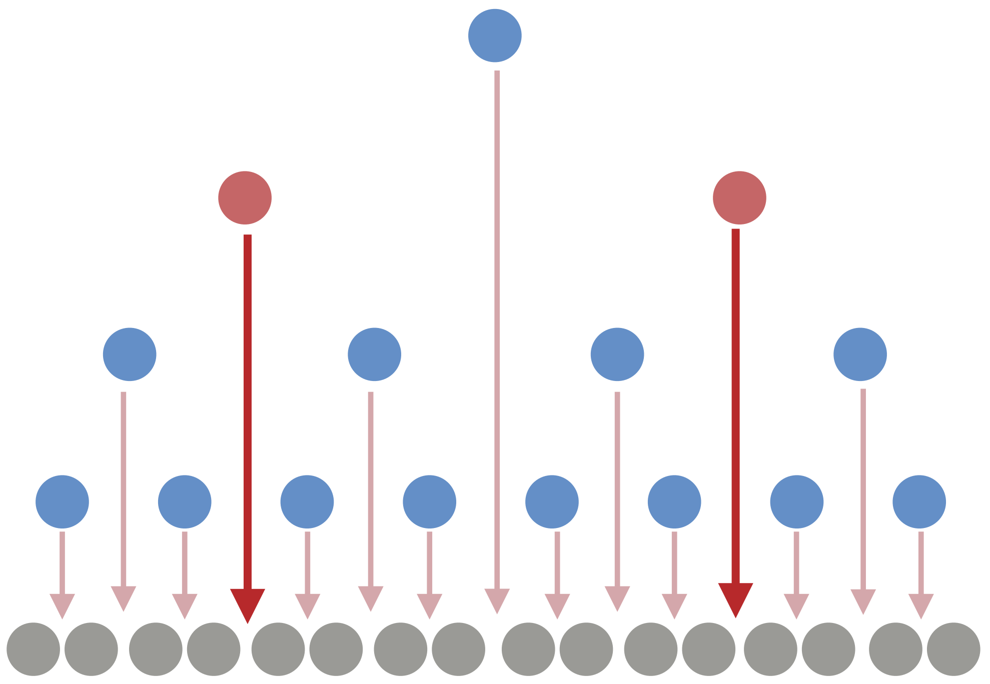
- Maximum number of swaps: $1 \bullet h$ ← number of swaps = tree height

  ↑ number of nodes

Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

 CONSTRUCT-HEAP(a, size)

 while (size > 1):
     swap(a[0], a[size-1])
     size = size-1
     SINK(a, 0, size)
```

```
CONSTRUCT-HEAP(a, size)

for i = size/2 - 1 → 0:
    SINK(a, i, size)
```



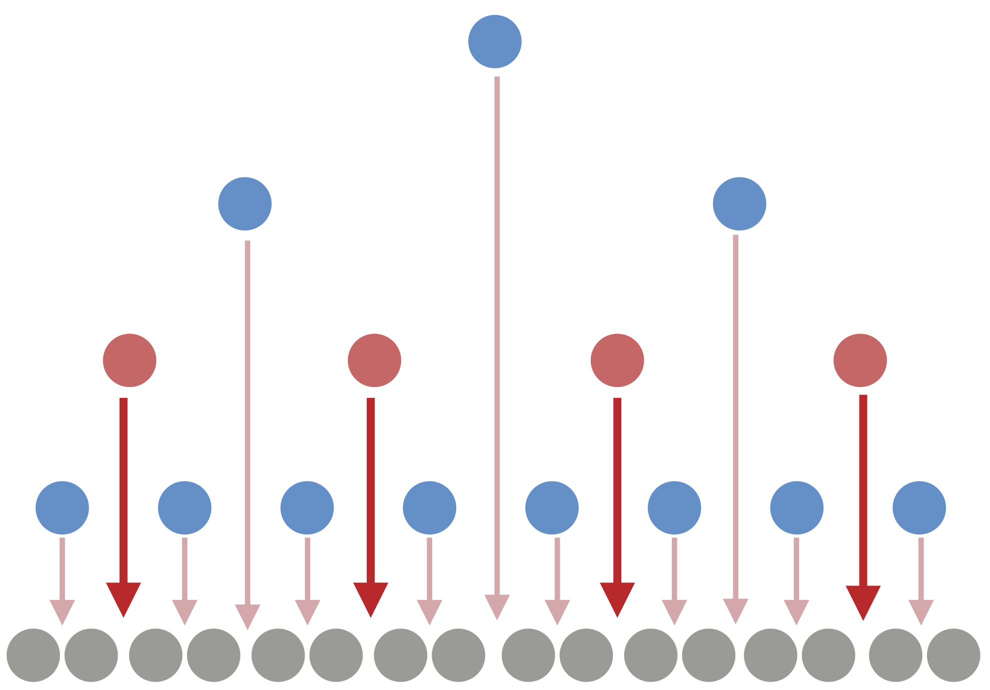
Heap Construction Analysis:

- Maximum number of swaps: $(1 \bullet h) + 2(h - 1)$

Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

 CONSTRUCT-HEAP(a, size)

 while (size > 1):
     swap(a[0], a[size-1])
     size = size-1
     SINK(a, 0, size)
```

```
CONSTRUCT-HEAP(a, size)

for i = size/2 - 1 → 0:
    SINK(a, i, size)
```



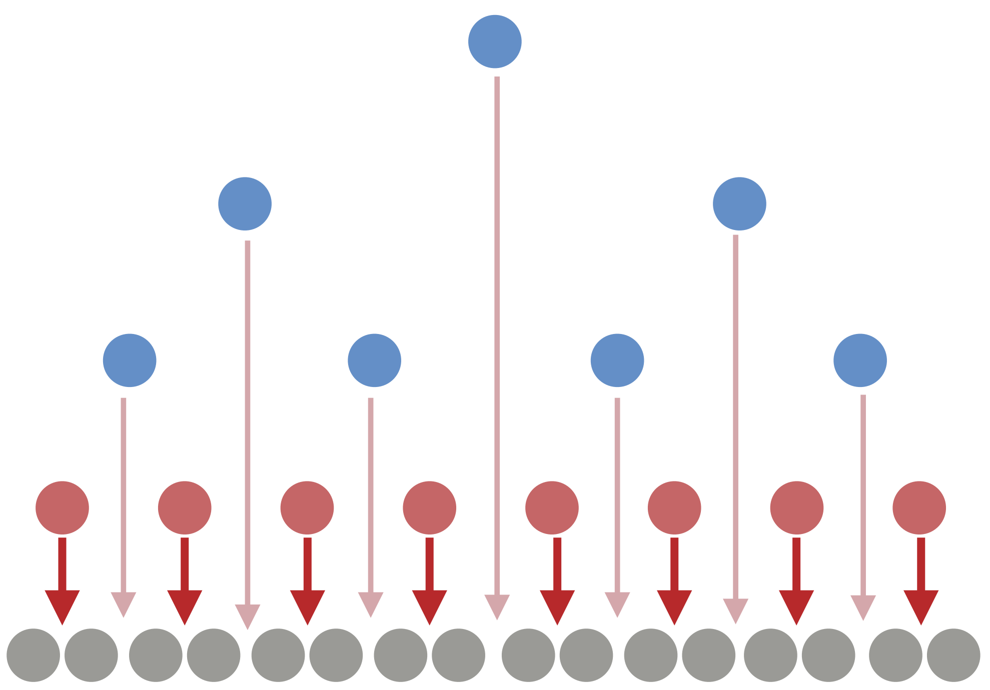
Heap Construction Analysis:

- Maximum number of swaps: $(1 \bullet h) + 2(h-1) + 4(h-2)$

Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

 CONSTRUCT-HEAP(a, size)

 while (size > 1):
    swap(a[0], a[size-1])
    size = size-1
    SINK(a, 0, size)
```

```
CONSTRUCT-HEAP(a, size)

for i = size/2 - 1 → 0:
   SINK(a, i, size)
```



Heap Construction Analysis:

- Maximum number of swaps: $(1 \bullet h) + 2(h-1) + 4(h-2) + \ \ldots \ + (\frac{n}{4} \bullet 1)$

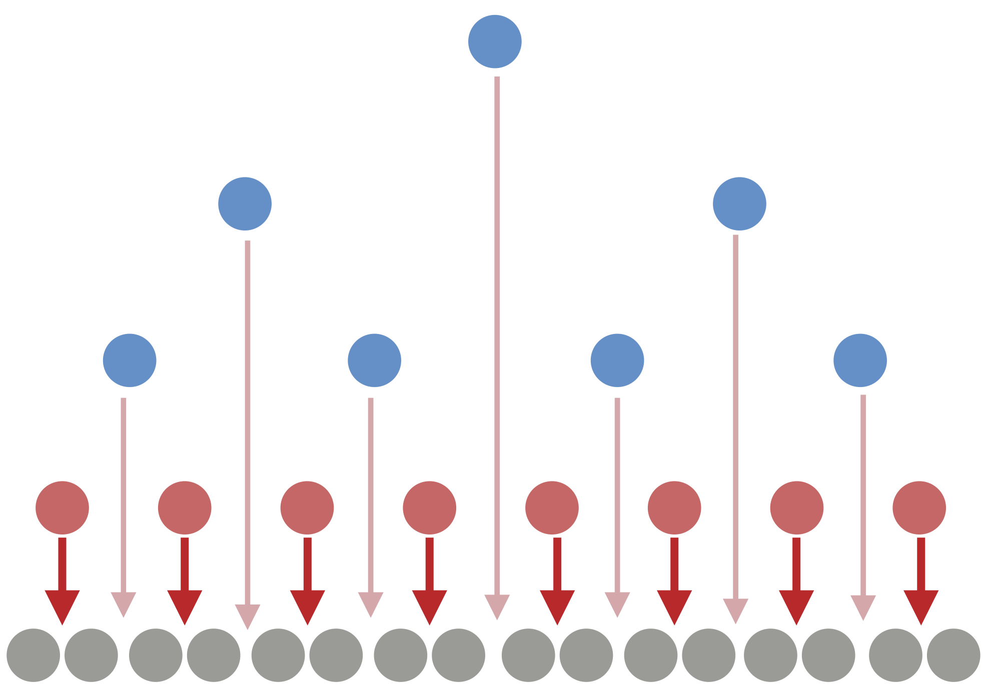
Heapsort: A Better Implementation

HEAP-SORT(a[], size)

```
CONSTRUCT-HEAP(a, size)

while (size > 1):
    swap(a[0], a[size-1])
    size = size-1
    SINK(a, 0, size)
```

CONSTRUCT-HEAP(a, size)

```
for i = size/2 - 1 → 0:
    SINK(a, i, size)
```



Heap Construction Analysis:

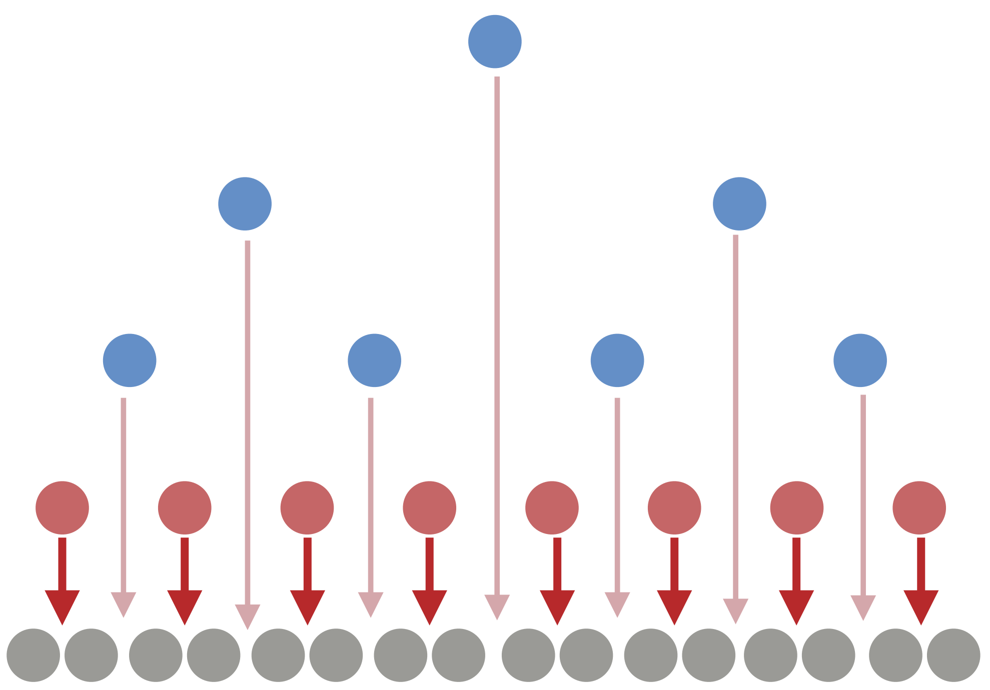
- Maximum number of swaps: $(1 \bullet h) + 2(h-1) + 4(h-2) + \ldots + \frac{n}{4}(1) = O(n)$

tricky sum (math skipped)

Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

 CONSTRUCT-HEAP(a, size)

 while (size > 1):
    swap(a[0], a[size-1])
    size = size-1
    SINK(a, 0, size)
```

```
CONSTRUCT-HEAP(a, size)

for i = size/2 - 1 → 0:
   SINK(a, i, size)
```



Heap Construction Analysis:

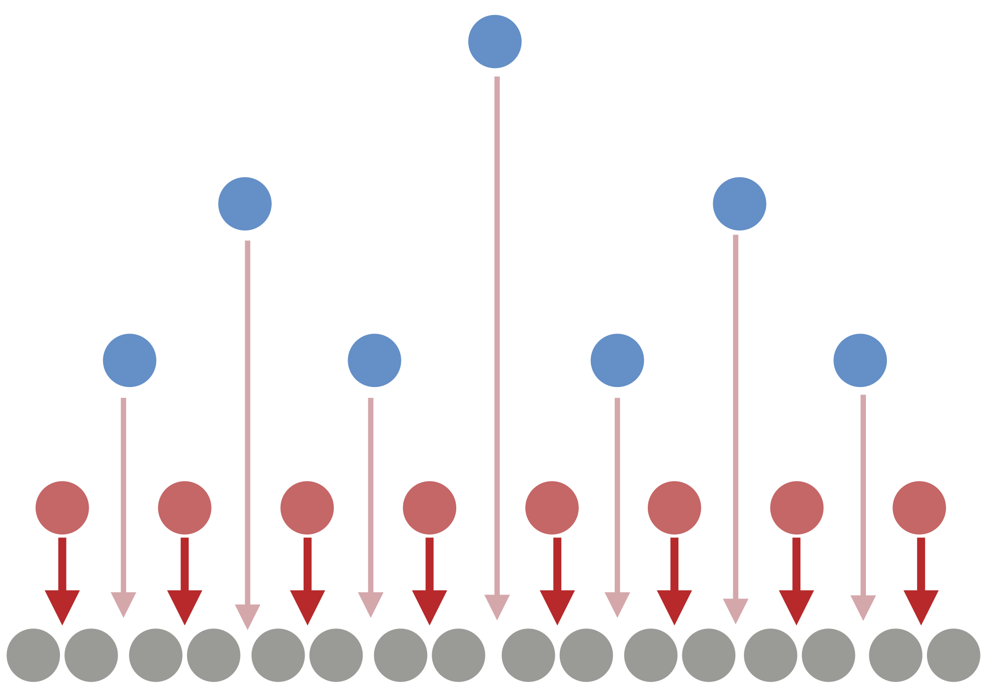
- Maximum number of swaps: $(1 \bullet h) + 2(h-1) + 4(h-2) + \ \dots \ + \frac{n}{4}(1) \quad = O(n)$

number of swaps is linear!

Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

 CONSTRUCT-HEAP(a, size)

 while (size > 1):
    swap(a[0], a[size-1])
    size = size-1
    SINK(a, 0, size)
```

```
CONSTRUCT-HEAP(a, size)

for i = size/2 - 1 → 0:
   SINK(a, i, size)
```



Heap Construction Analysis:

- Maximum number of swaps: $(1 \bullet h) + 2(h-1) + 4(h-2) + \ \ldots \ + \frac{n}{4}(1) \quad = O(n)$
- Maximum number of compares: $2 \times$ number of swaps

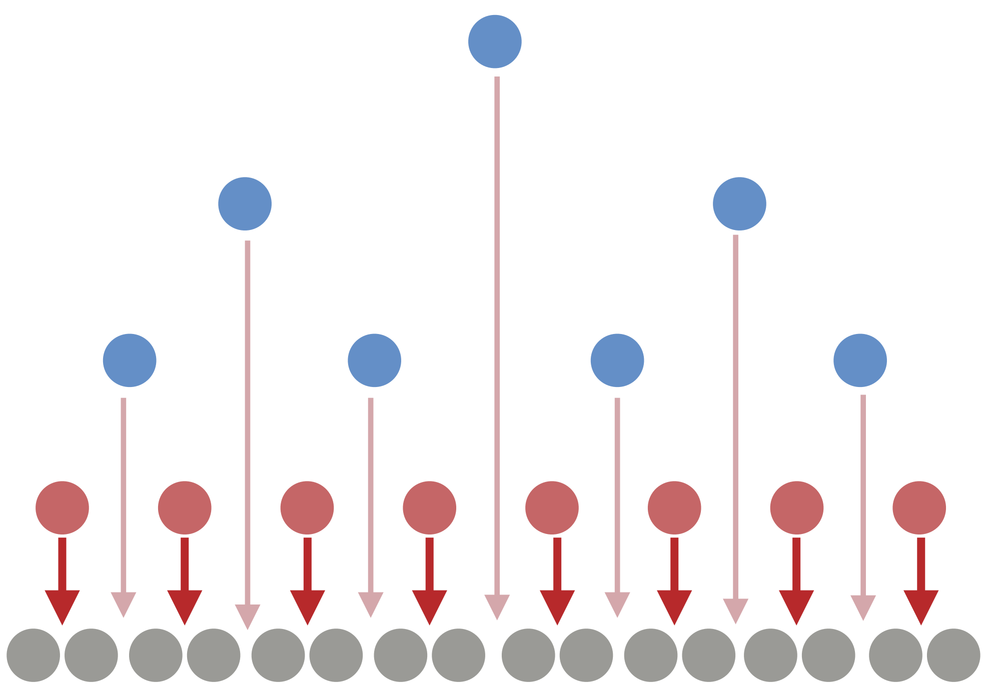
check the analysis
of the **SINK** operation!

Heapsort: A Better Implementation

```
HEAP-SORT(a[], size)

  CONSTRUCT-HEAP(a, size)

  while (size > 1):
     swap(a[0], a[size-1])
     size = size-1
     SINK(a, 0, size)
```

```
CONSTRUCT-HEAP(a, size)

for i = size/2 - 1 → 0:
   SINK(a, i, size)
```

Think. Why does this heap construction code run in $O(n)$ while inserting all the elements into a heap takes $O(n \log n)$ time?



Heap Construction Analysis:

- Maximum number of swaps: $(1 \bullet h) + 2(h-1) + 4(h-2) + \ \ldots \ + \frac{n}{4}(1) \quad = O(n)$
- Maximum number of compares: $2 \times$ number of swaps

check the analysis of the **SINK** operation!

Optional

$$\sum_{i=0}^{n} i \times 2^i = (n-1)2^{n+1} + 2$$

$$(1 \bullet h) + 2(h-1) + 4(h-2) + \ \ldots \ + \ \tfrac{n}{4}(1)$$

$$= 2^0(1 \bullet h) + 2^1(h-1) + 2^2(h-2) + \ \ldots \ + \ 2^{h-1}$$

$$= \sum_{i=0}^{h-1} 2^i(h-i) \ = \ (\sum_{i=0}^{h-1} 2^i h) - (\sum_{i=0}^{h-1} i2^i) \ = \ h(2^h - 1) - (\sum_{i=0}^{h-1} i2^i)$$

$$= h(2^h - 1) - ((h-2)2^h + 2)$$

$$= h(2^h - 1) - (h2^h - 2^{h+1} + 2)$$

$$= h2^h - h - h2^h + 2^{h+1} - 2$$

$$= 2^{h+1} - 2 \quad \longleftarrow \quad h \sim \log_2 n$$

$$= O(n)$$

Heapsort Analysis

Worst Case: $\Theta(n)$ to **construct** the heap and $\Theta(n \log n)$ to **heapsort**.

Average Case: $\Theta(n \log n)$

Heapsort Analysis

Worst Case: $\Theta(n)$ to **construct** the heap and $\Theta(n \log n)$ to **heapsort**.

Average Case: $\Theta(n \log n)$

Best Case: $\Theta(n)$ if all the elements are the same.

Heapsort Analysis

Worst Case: $\Theta(n)$ to **construct** the heap and $\Theta(n \log n)$ to **heapsort**.

Average Case: $\Theta(n \log n)$

Best Case: $\Theta(n)$ if all the elements are the same.

🤔 **Why?** Trace on a piece of paper to see why!

Heapsort Analysis

Worst Case: $\Theta(n)$ to **construct** the heap and $\Theta(n \log n)$ to **heapsort**.

Average Case: $\Theta(n \log n)$

Best Case: $\Theta(n)$ if all the elements are the same.

Running Time:

- Number of compares: At most $\sim 2n \log_2 n$.

↑

$\sim n \log_2 n$ for merge sort and
$\sim 1.39n \log_2 n$ for quicksort (on random data)

Heapsort Analysis

Worst Case: $\Theta(n)$ to **construct** the heap and $\Theta(n \log n)$ to **heapsort**.

Average Case: $\Theta(n \log n)$

Best Case: $\Theta(n)$ if all the elements are the same.

Running Time:

- Number of compares: At most $\sim 2n \log_2 n$.
- Actual running time: Slower than merge sort and quicksort because of the higher number of comparisons and the the poor use of cache.

optimizations are possible

Heapsort Analysis

Worst Case: $\Theta(n)$ to **construct** the heap and $\Theta(n \log n)$ to **heapsort**.

Average Case: $\Theta(n \log n)$

Best Case: $\Theta(n)$ if all the elements are the same.

Running Time:

- Number of compares: At most $\sim 2n \log_2 n$.
- Actual running time: Slower than merge sort and quicksort because of the higher number of comparisons and the the poor use of cache.

Memory. Heapsort is an *in-place* sorting algorithm.

Heapsort Analysis

Worst Case: $\Theta(n)$ to **construct** the heap and $\Theta(n \log n)$ to **heapsort**.

Average Case: $\Theta(n \log n)$

Best Case: $\Theta(n)$ if all the elements are the same.

Running Time:

- Number of compares: At most $\sim 2n \log_2 n$.
- Actual running time: Slower than merge sort and quicksort because of the higher number of comparisons and the the poor use of cache.

Memory. Heapsort is an *in-place* sorting algorithm.

Bottom line.

- $\Theta(n \log n)$ in the worst case and also sorts in-place at the same time.
  (Merge Sort is not in-place and Quicksort has a theoretical worst case of $\Theta(n^2)$)
- Practically, not frequently used because it is slower than merge sort and quicksort.

Introsort

From Wikipedia, the free encyclopedia

Introsort or **introspective sort** is a hybrid sorting algorithm that provides both fast average performance and (asymptotically) optimal worst-case performance. It begins with quicksort, it switches to heapsort when the recursion depth exceeds a level based on (the logarithm of) the number of elements being sorted and it switches to insertion sort when the number of elements is below some threshold. This combines the good parts of the three algorithms, with practical performance comparable to quicksort on typical data sets and worst-case $O(n \log n)$ runtime due to the heap sort. Since the three algorithms it uses are comparison sorts, it is also a comparison sort.

Introsort

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n \log n)$
Average performance	$O(n \log n)$

Used for the C++ STL `sort()` function

Sorting algorithms: summary

	inplace?	stable?	best	average	worst	remarks
selection	✔		$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	n exchanges
insertion	✔	✔	n	$\frac{1}{4} n^2$	$\frac{1}{2} n^2$	use for small n or partially ordered
merge		✔	$\frac{1}{2} n \log_2 n$	$n \log_2 n$	$n \log_2 n$	$\Theta(n \log n)$ guarantee; stable
timsort		✔	n	$n \log_2 n$	$n \log_2 n$	improves mergesort when pre-existing order
quick	✔		$n \log_2 n$	$2 n \ln n$	$\frac{1}{2} n^2$	$\Theta(n \log n)$ probabilistic guarantee; fastest in practice
3-way quick	✔		n	$2 n \ln n$	$\frac{1}{2} n^2$	improves quicksort when duplicate keys
heap	✔		$3 n$	$2 n \log_2 n$	$2 n \log_2 n$	$\Theta(n \log n)$ guarantee; in-place
?	✔	✔	n	$n \log_2 n$	$n \log_2 n$	holy sorting grail

number of compares to sort an array of n elements

By Kevin Wayne