CS11212 - Spring 2022

Data Structures & Introduction to Algorithms

Data Structures

Priority Queues

Ibrahim Albluwi

ADTs We Know So Far

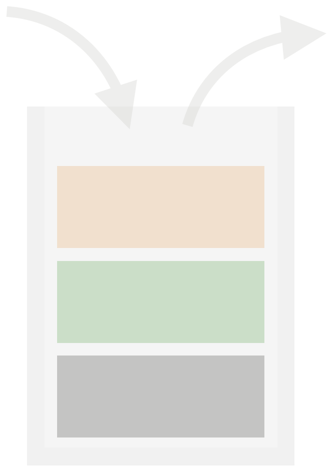
List

operations

```
add_to_head(val)
add_to_tail(val)
remove_head()
remove_tail()
remove(val)
contains(val)
```

Linked List
Array

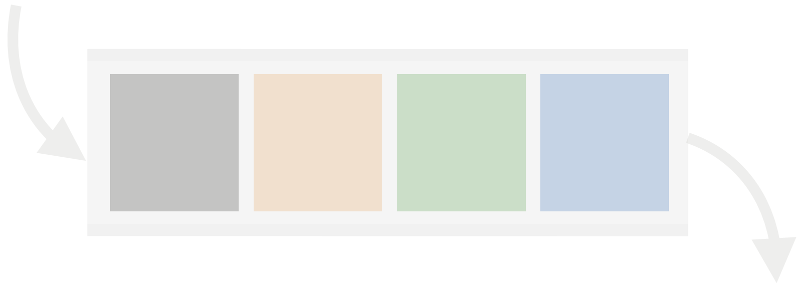
common
data structures



Stack

```
push(val)
pop()
top()
```

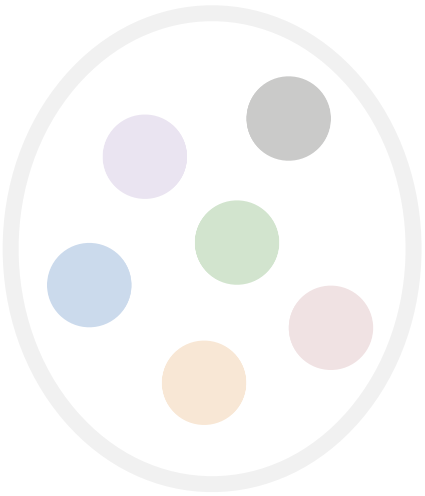
Linked List
Array



Queue

```
enqueue(val)
dequeue()
first()
last()
```

Linked List
Array

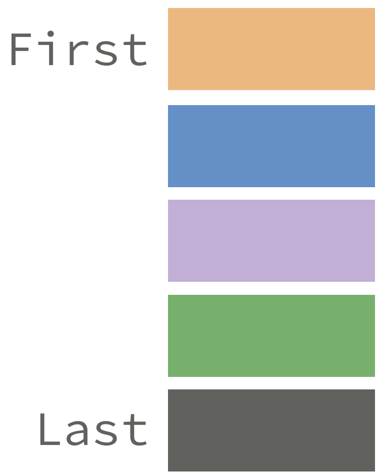


Set

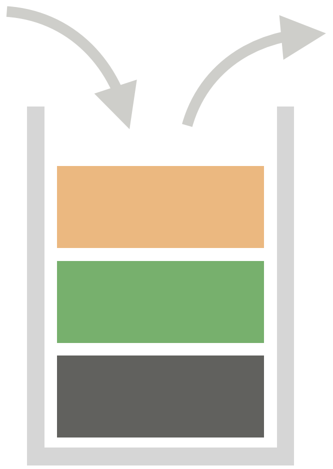
```
insert(val)
remove(val)
contains(val)
```

Linked List
Array
BST
Hash Table

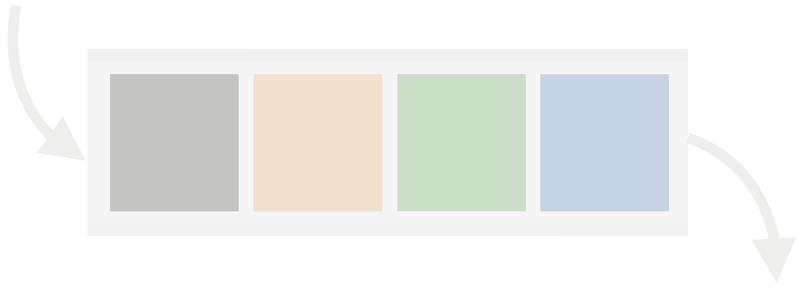
ADTs We Know So Far



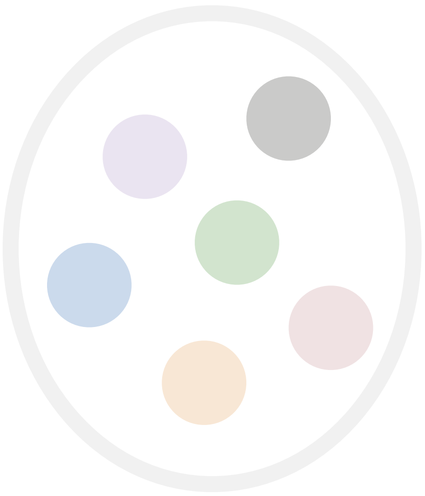
List



Stack



Queue



Set

operations

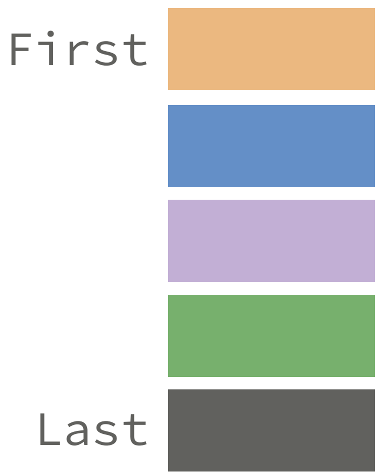
List	Stack	Queue	Set

`add_to_head(val)`
`add_to_tail(val)`
`remove_head()`
`remove_tail()`
`remove(val)`
`contains(val)`
	`push(val)`
	`pop()`
	`top()`
		`enqueue(val)`
		`dequeue()`
		`first()`
		`last()`
			`insert(val)`
			`remove(val)`
			`contains(val)`

Linked List Array	Linked List Array	Linked List Array	Linked List Array BST Hash Table

common
data structures

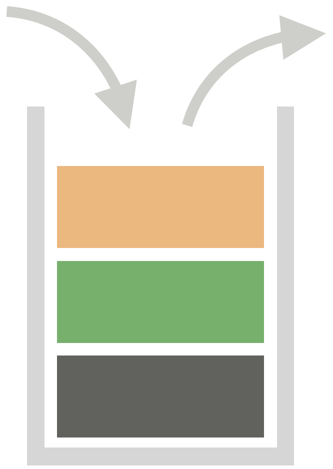
ADTs We Know So Far



List

```
add_to_head(val)
add_to_tail(val)
remove_head()
remove_tail()
remove(val)
contains(val)
```

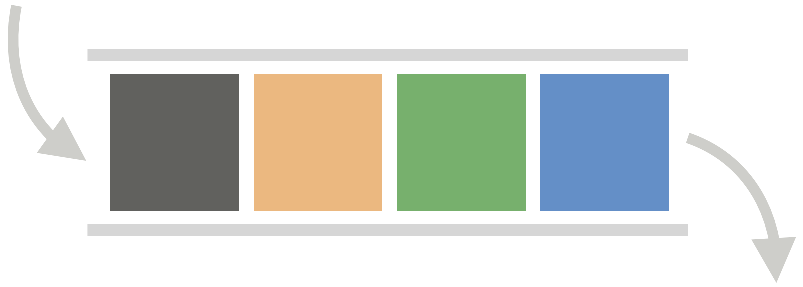
Linked List
Array



Stack

```
push(val)
pop()
top()
```

Linked List
Array



Queue

operations

```
enqueue(val)
dequeue()
first()
last()
```

Linked List
Array

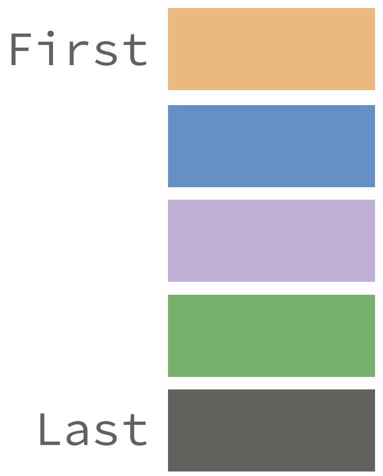
common
data structures

Set

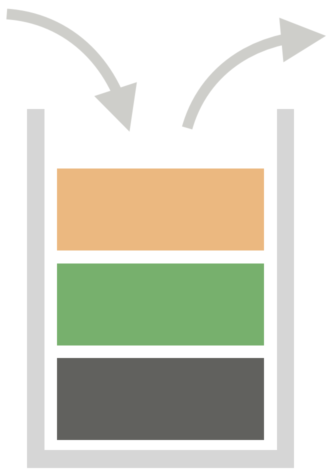
```
insert(val)
remove(val)
contains(val)
```

Linked List
Array
BST
Hash Table

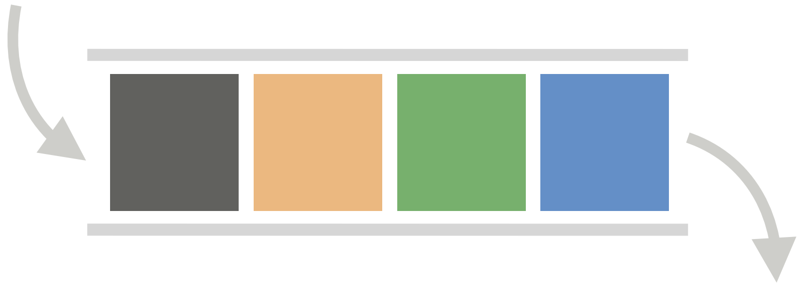
ADTs We Know So Far



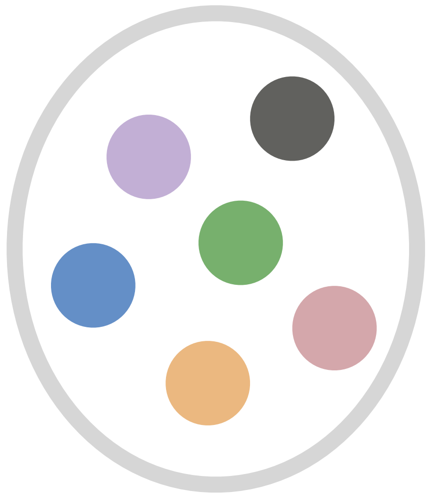
List



Stack



Queue



Set

```
add_to_head(val)
add_to_tail(val)
remove_head()
remove_tail()
remove(val)
contains(val)
```

Linked List
Array

```
push(val)
pop()
top()
```

Linked List
Array

```
enqueue(val)
dequeue()
first()
last()
```

Linked List
Array

operations

```
insert(val)
remove(val)
contains(val)
```

Linked List
Array
BST
Hash Table

common data structures

Priority Queue (a new ADT)

Max-Priority Queue

```
   T get_max() const
   T remove_max()
void insert(const T& val)
```

Min-Priority Queue

```
   T get_min() const
   T remove_min()
void insert(const T& val)
```

Priority Queue (a new ADT)

```
Max-Priority Queue

   T get_max() const
   T remove_max()
void insert(const T& val)
```

```
Min-Priority Queue

   T get_min() const
   T remove_min()
void insert(const T& val)
```

Applications.

In an emergency room, patients are assigned priority based on their condition.

Priority Queue (a new ADT)

Max-Priority Queue

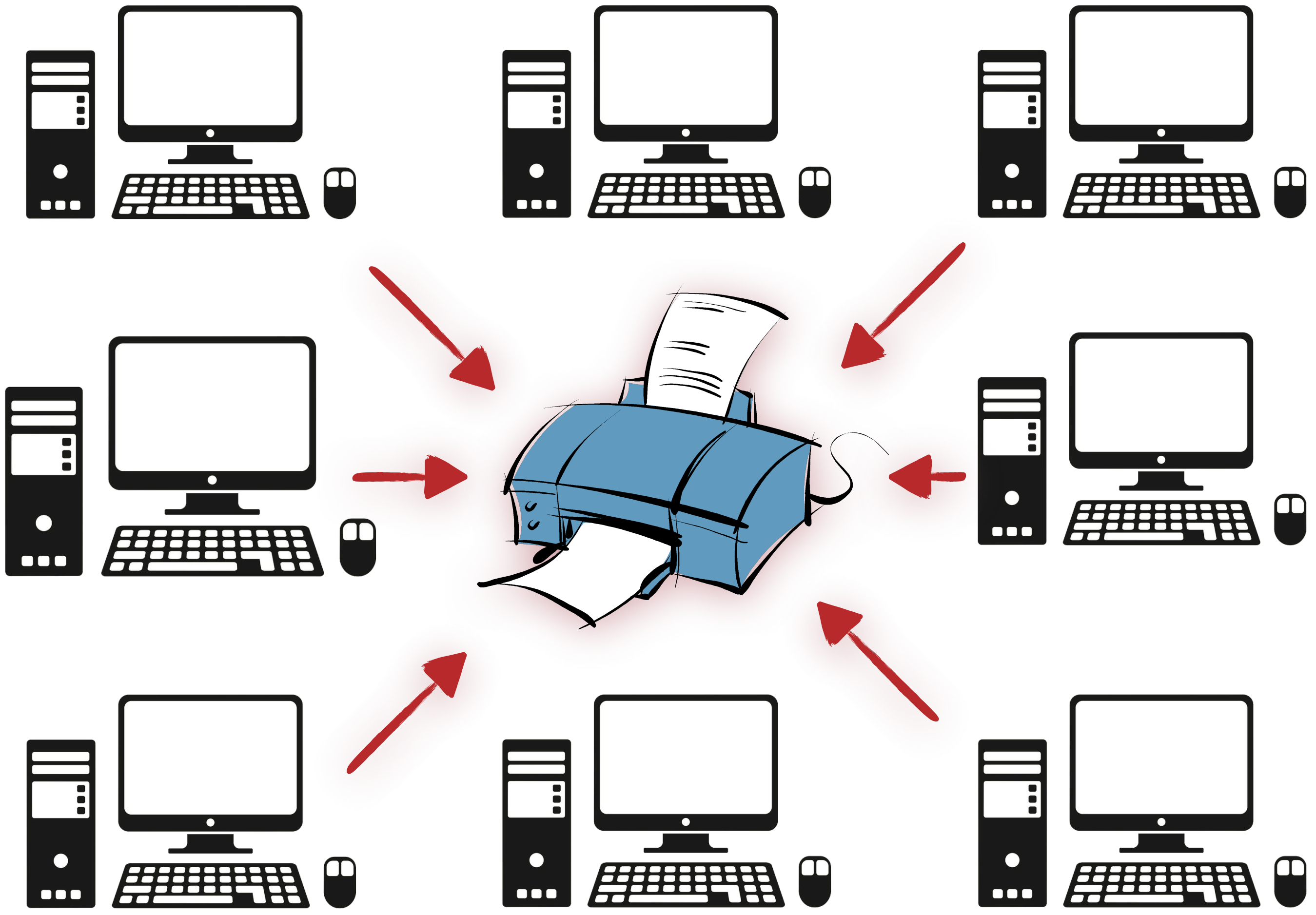
```
    T get_max() const
    T remove_max()
void insert(const T& val)
```

Min-Priority Queue

```
    T get_min() const
    T remove_min()
void insert(const T& val)
```

Applications.

In an printer queue, can be configured to give higher priority to print jobs from certain people (e.g. based on role or department)



Priority Queue (a new ADT)

Max-Priority Queue

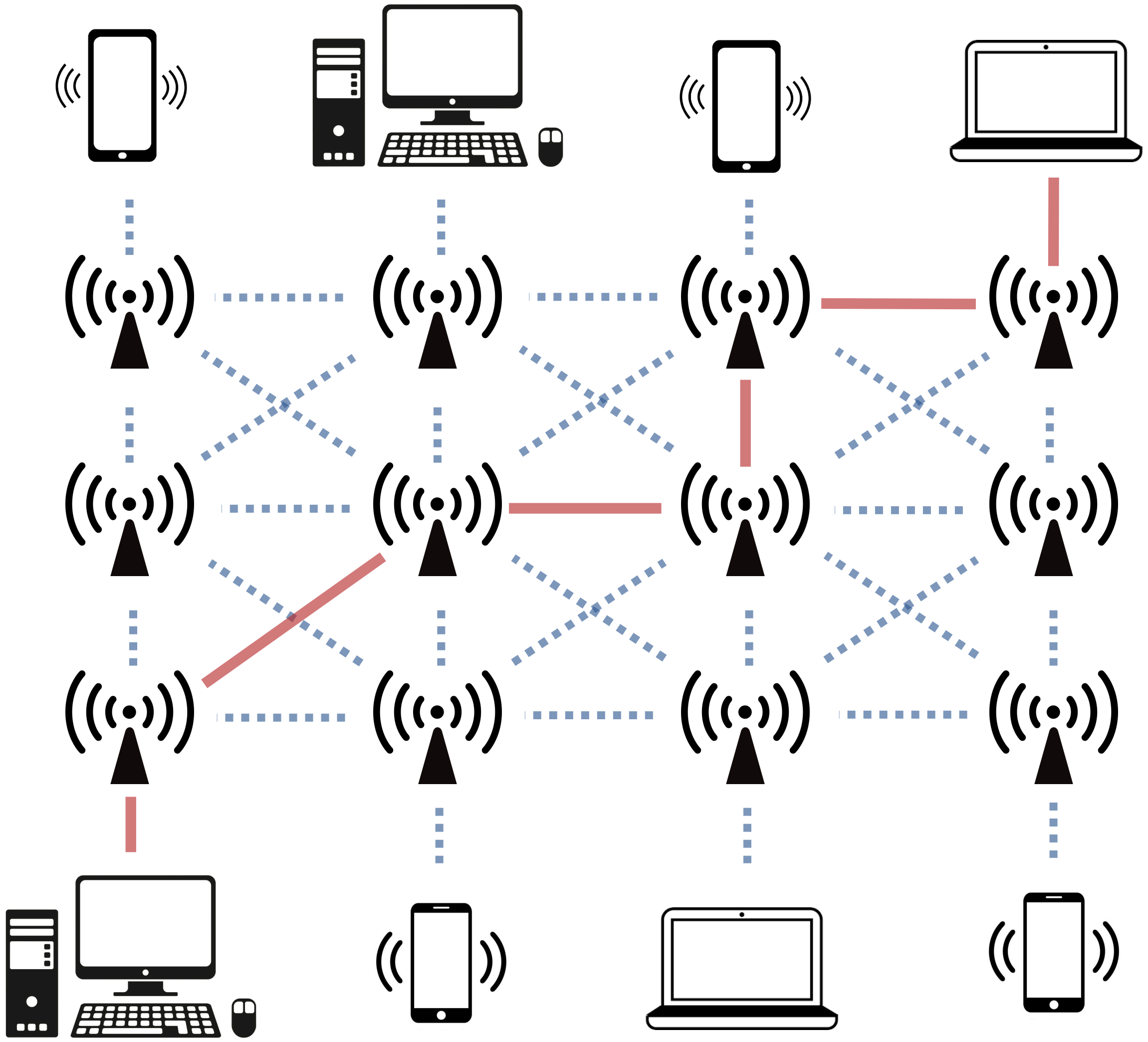
```
   T get_max() const
   T remove_max()
void insert(const T& val)
```

Min-Priority Queue

```
   T get_min() const
   T remove_min()
void insert(const T& val)
```

Applications.

A network router, can give higher priority to packets based on their type or sender (e.g. live streaming data is very important to be routed quickly)



Priority Queue (a new ADT)

```
Max-Priority Queue

   T get_max() const
   T remove_max()
void insert(const T& val)
```

```
Min-Priority Queue

   T get_min() const
   T remove_min()
void insert(const T& val)
```

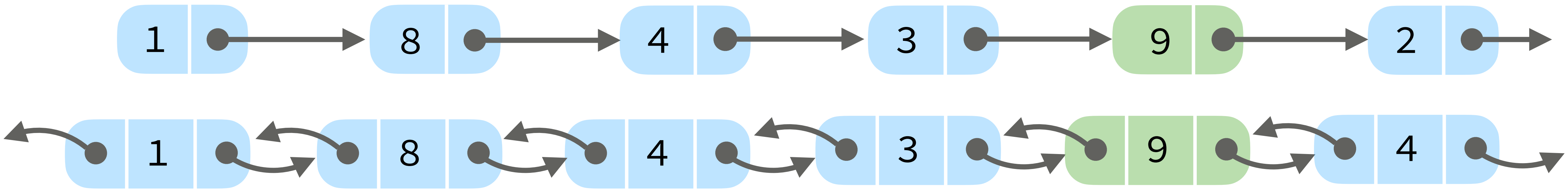
Applications.

Used in many algorithms to process data elements in ascending or descending order or to keep track of the largest (or smallest) *k* elements seen so far.

- A*
- Dijkstra's Shortest Paths Algorithm
- Prim's Minimum Spanning Trees Algorithm
- Huffman Coding
- Streaming Median
- Interrupt Handling
- etc.

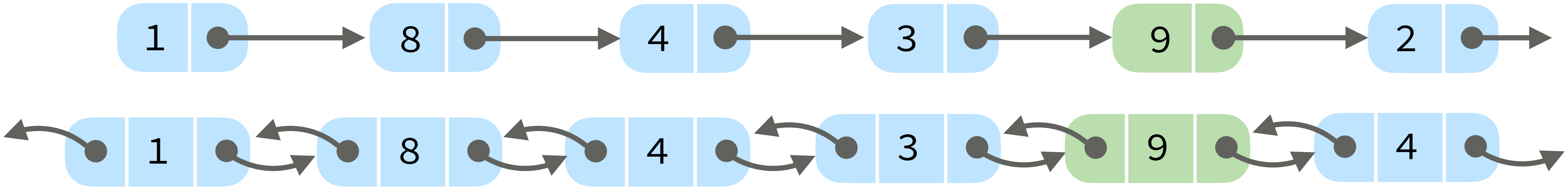
Max-Priority Queue: Possible Implementations

	`insert(val)`	`remove_max()`	`get_max()`
Unordered DLL			
Unordered SLL			



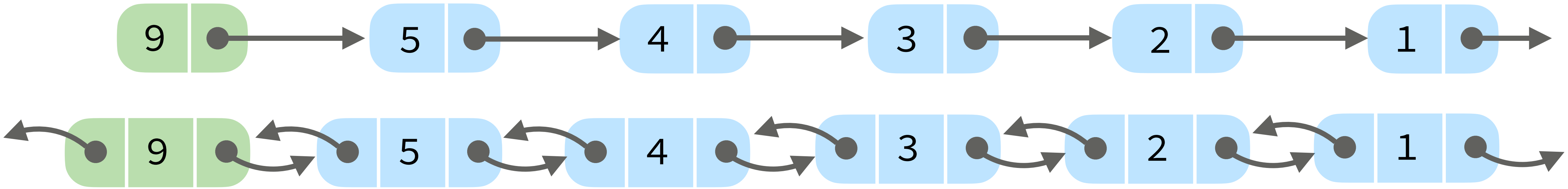
Max-Priority Queue: Possible Implementations

	insert(val)	remove_max()	get_max()
Unordered DLL	O(1)	O(n)	O(n)
Unordered SLL	O(1)	O(n)	O(n)



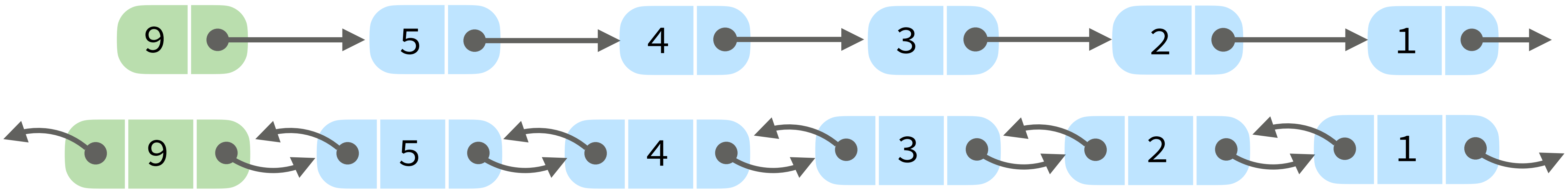
Max-Priority Queue: Possible Implementations

	`insert(val)`	`remove_max()`	`get_max()`
Unordered DLL	`O(1)`	`O(n)`	`O(n)`
Unordered SLL	`O(1)`	`O(n)`	`O(n)`
Ordered DLL			
Ordered SLL			



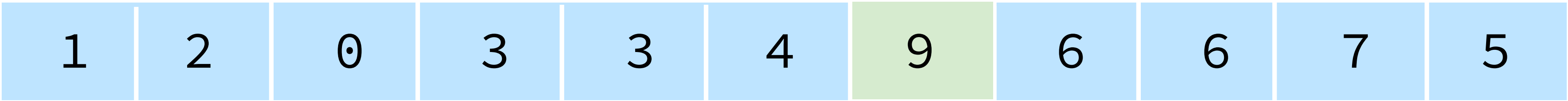
Max-Priority Queue: Possible Implementations

	`insert(val)`	`remove_max()`	`get_max()`
Unordered DLL	`O(1)`	`O(n)`	`O(n)`
Unordered SLL	`O(1)`	`O(n)`	`O(n)`
Ordered DLL	`O(n)`	`O(1)`	`O(1)`
Ordered SLL	`O(n)`	`O(1)`	`O(1)`



Max-Priority Queue: Possible Implementations

	`insert(val)`	`remove_max()`	`get_max()`
Unordered DLL	`O(1)`	`O(n)`	`O(n)`
Unordered SLL	`O(1)`	`O(n)`	`O(n)`
Ordered DLL	`O(n)`	`O(1)`	`O(1)`
Ordered SLL	`O(n)`	`O(1)`	`O(1)`
Unordered Array			



1	2	0	3	3	4	9	6	6	7	5

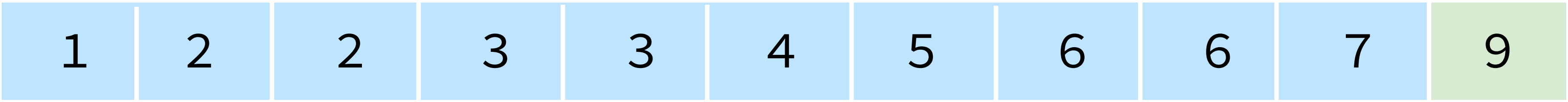
Max-Priority Queue: Possible Implementations

	insert(val)	remove_max()	get_max()
Unordered DLL	O(1)	O(n)	O(n)
Unordered SLL	O(1)	O(n)	O(n)
Ordered DLL	O(n)	O(1)	O(1)
Ordered SLL	O(n)	O(1)	O(1)
Unordered Array	O(1)	O(n)	O(n)

1	2	0	3	3	4	9	6	6	7	5

Max-Priority Queue: Possible Implementations

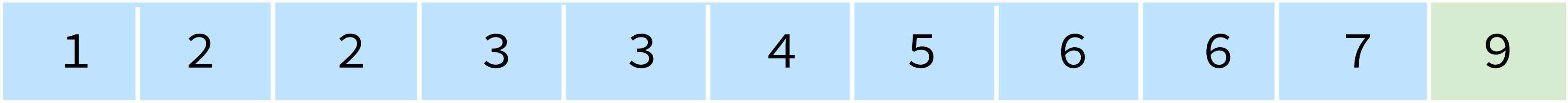
	`insert(val)`	`remove_max()`	`get_max()`
Unordered DLL	`O(1)`	`O(n)`	`O(n)`
Unordered SLL	`O(1)`	`O(n)`	`O(n)`
Ordered DLL	`O(n)`	`O(1)`	`O(1)`
Ordered SLL	`O(n)`	`O(1)`	`O(1)`
Unordered Array	`O(1)`	`O(n)`	`O(n)`
Ordered Array			



1	2	2	3	3	4	5	6	6	7	9

Max-Priority Queue: Possible Implementations

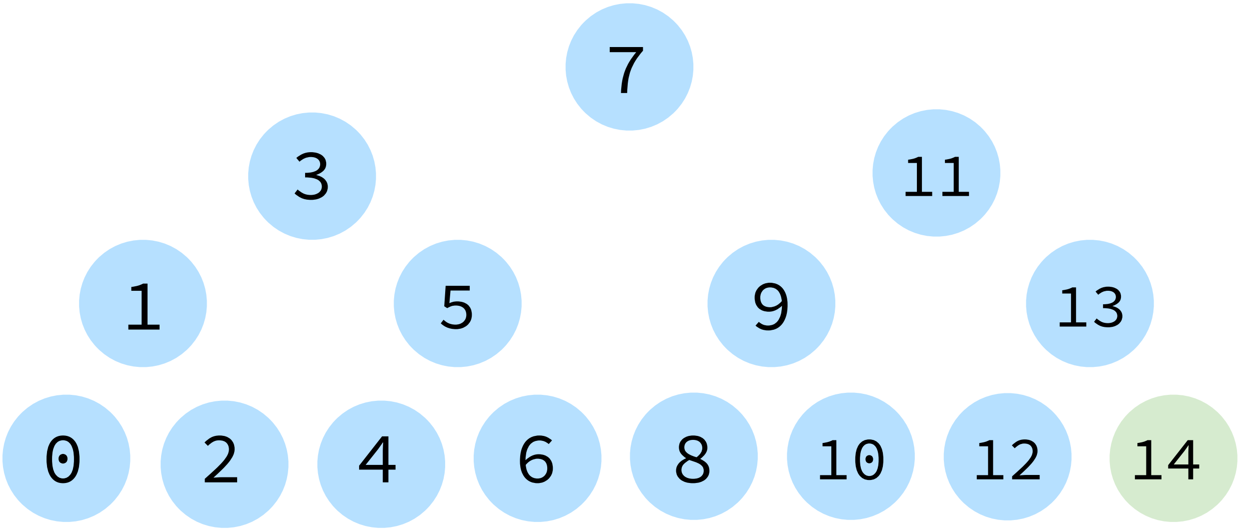
	`insert(val)`	`remove_max()`	`get_max()`
Unordered DLL	`O(1)`	`O(n)`	`O(n)`
Unordered SLL	`O(1)`	`O(n)`	`O(n)`
Ordered DLL	`O(n)`	`O(1)`	`O(1)`
Ordered SLL	`O(n)`	`O(1)`	`O(1)`
Unordered Array	`O(1)`	`O(n)`	`O(n)`
Ordered Array	`O(n)`	`O(1)`	`O(1)`



1	2	2	3	3	4	5	6	6	7	9

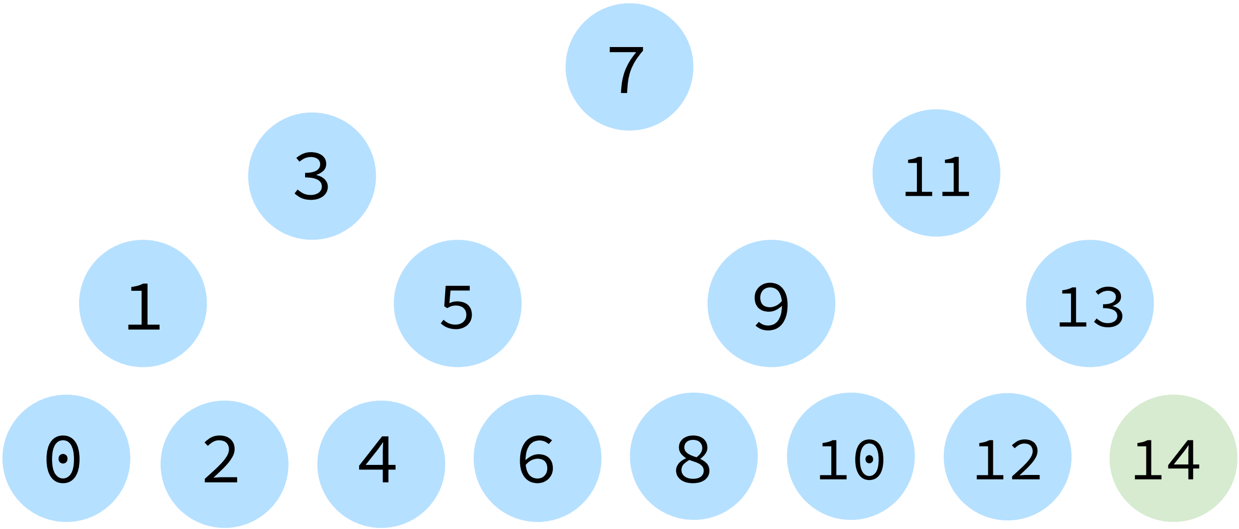
Max-Priority Queue: Possible Implementations

	`insert(val)`	`remove_max()`	`get_max()`
Unordered DLL	`O(1)`	`O(n)`	`O(n)`
Unordered SLL	`O(1)`	`O(n)`	`O(n)`
Ordered DLL	`O(n)`	`O(1)`	`O(1)`
Ordered SLL	`O(n)`	`O(1)`	`O(1)`
Unordered Array	`O(1)`	`O(n)`	`O(n)`
Ordered Array	`O(n)`	`O(1)`	`O(1)`
Balanced BST			



Max-Priority Queue: Possible Implementations

	insert(val)	remove_max()	get_max()
Unordered DLL Unordered SLL	O(1) O(1)	O(n) O(n)	O(n) O(n)
Ordered DLL Ordered SLL	O(n) O(n)	O(1) O(1)	O(1) O(1)
Unordered Array	O(1)	O(n)	O(n)
Ordered Array	O(n)	O(1)	O(1)
Balanced BST	O(log n)	O(log n)	O(log n)



Max-Priority Queue: Possible Implementations

	insert(val)	remove_max()	get_max()
Unordered DLL	O(1)	O(n)	O(n)
Unordered SLL	O(1)	O(n)	O(n)
Ordered DLL	O(n)	O(1)	O(1)
Ordered SLL	O(n)	O(1)	O(1)
Unordered Array	O(1)	O(n)	O(n)
Ordered Array	O(n)	O(1)	O(1)
Balanced BST	O(log n)	O(log n)	O(log n)

🤔 **Can we do better?**

Max-Priority Queue: Possible Implementations

	`insert(val)`	`remove_max()`	`get_max()`
Unordered DLL	`O(1)`	`O(n)`	`O(n)`
Unordered SLL	`O(1)`	`O(n)`	`O(n)`
Ordered DLL	`O(n)`	`O(1)`	`O(1)`
Ordered SLL	`O(n)`	`O(1)`	`O(1)`
Unordered Array	`O(1)`	`O(n)`	`O(n)`
Ordered Array	`O(n)`	`O(1)`	`O(1)`
Balanced BST	`O(log n)`	`O(log n)`	`O(log n)`
Binary Heap	`O(log n)`	`O(log n)`	**`O(1)`**

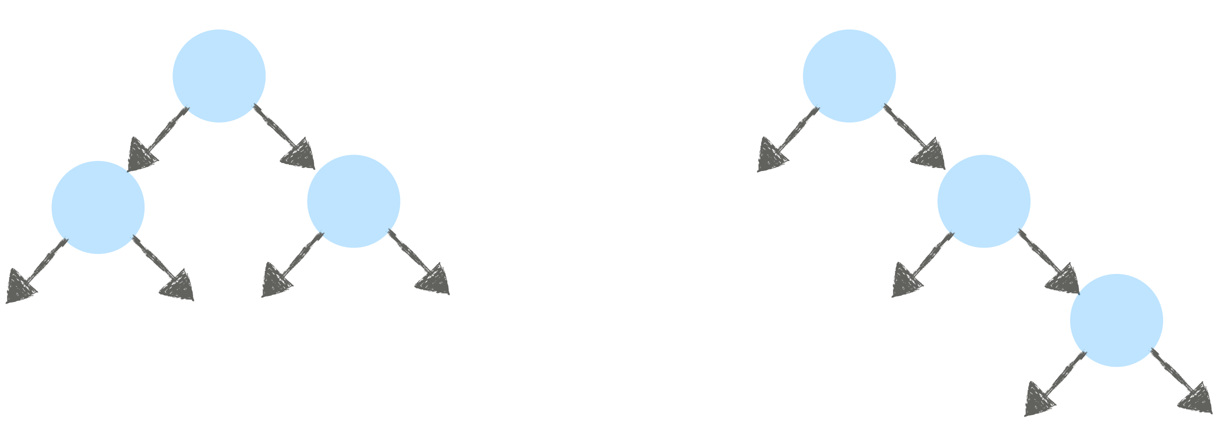
🤔 **Can we do better?**

🎉 Yes! *… slightly better*

Using a very simple data structure!

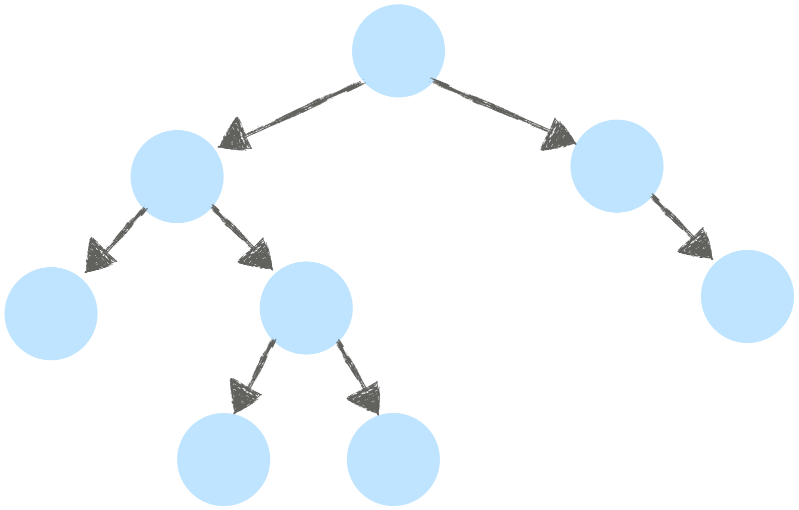
Binary Heaps are Complete Binary Trees

Binary Tree: Every node has *at most* two children.

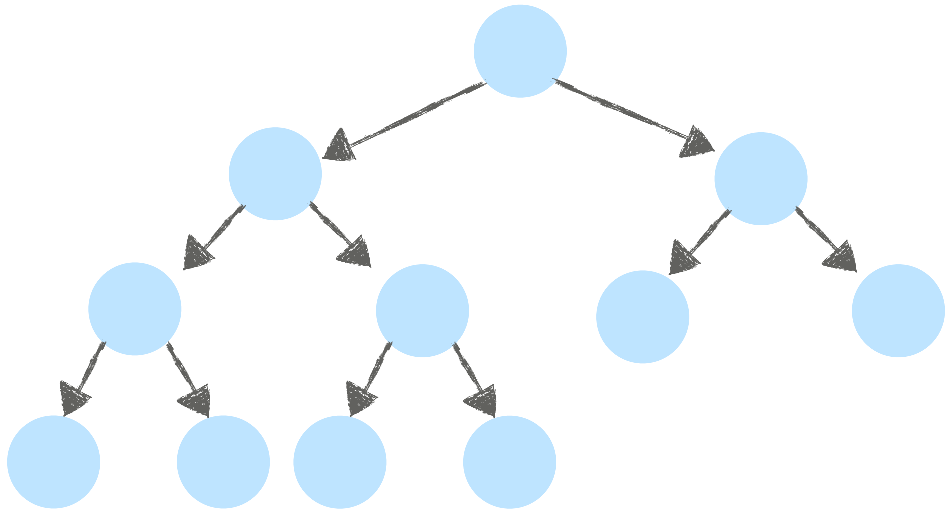


Complete Binary Tree:

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



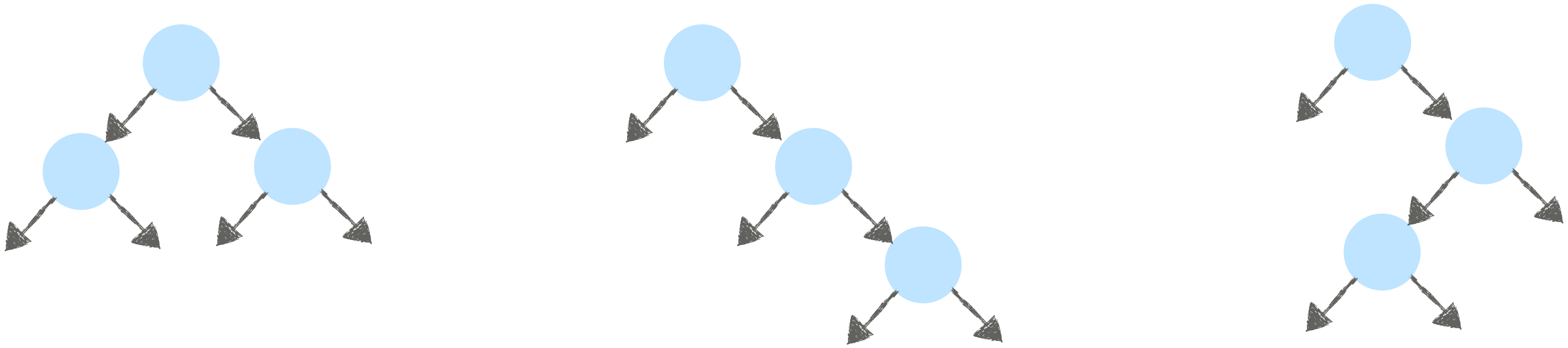
not complete



complete

Binary Heaps are Complete Binary Trees

Binary Tree: Every node has *at most* two children.

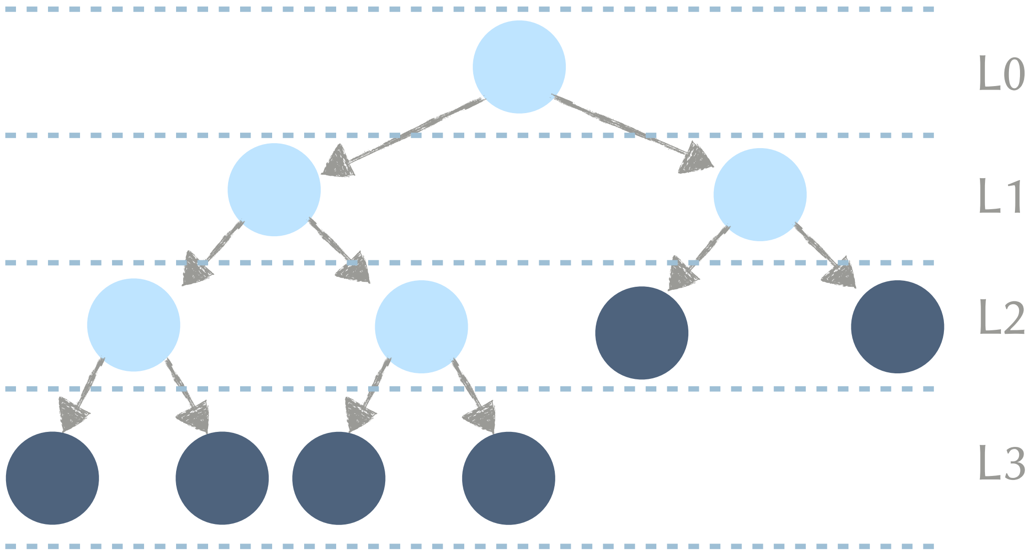


Complete Binary Tree:

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

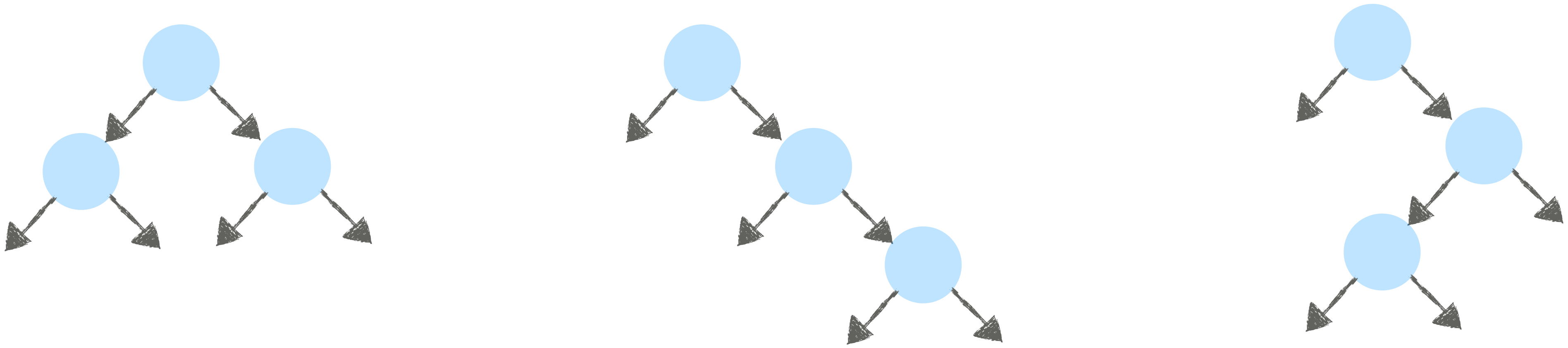
Properties:

- All leaves are at level h or $h - 1$.
  (h = tree height)



Binary Heaps are Complete Binary Trees

Binary Tree: Every node has *at most* two children.

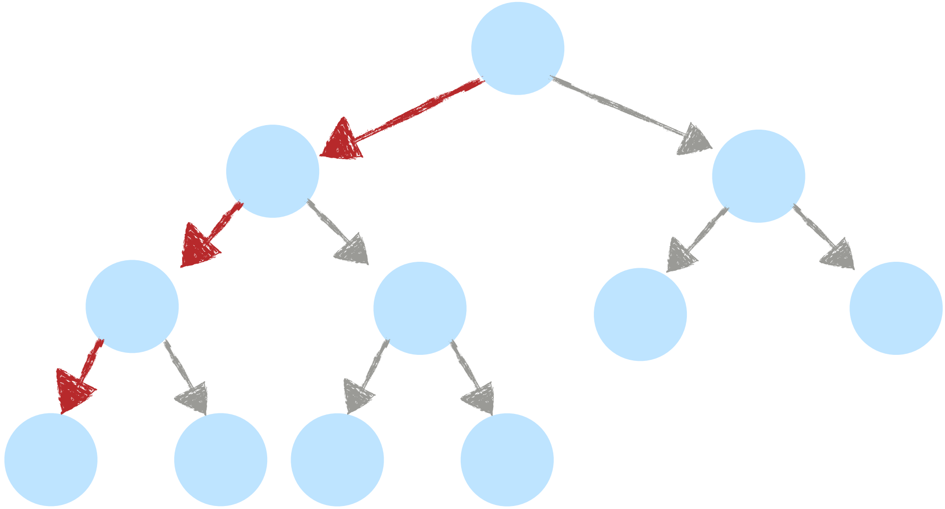


Complete Binary Tree:

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

Properties:

- All leaves are at level h or $h - 1$.
- Height if there are n nodes: $h = \lfloor \log_2 n \rfloor$



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

Binary Heaps (Tree Representation)

Binary Heap: (max-ordered)

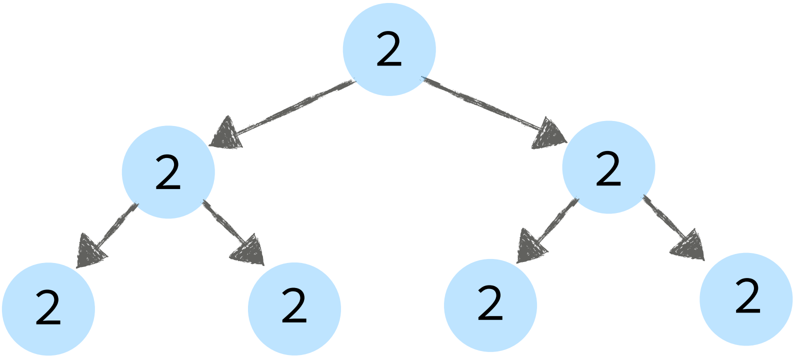
- Structure: Must be a complete binary tree.
- Order: Every node is not less than its children.

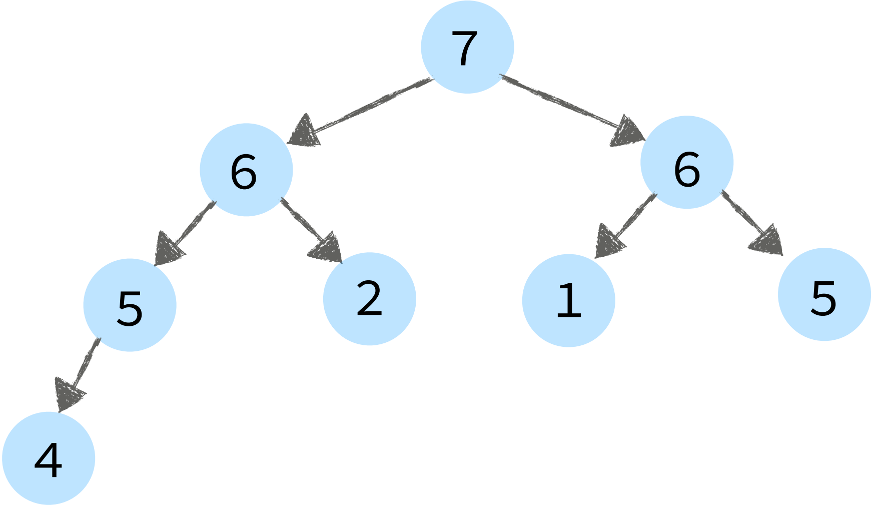
Binary Heaps (Tree Representation)

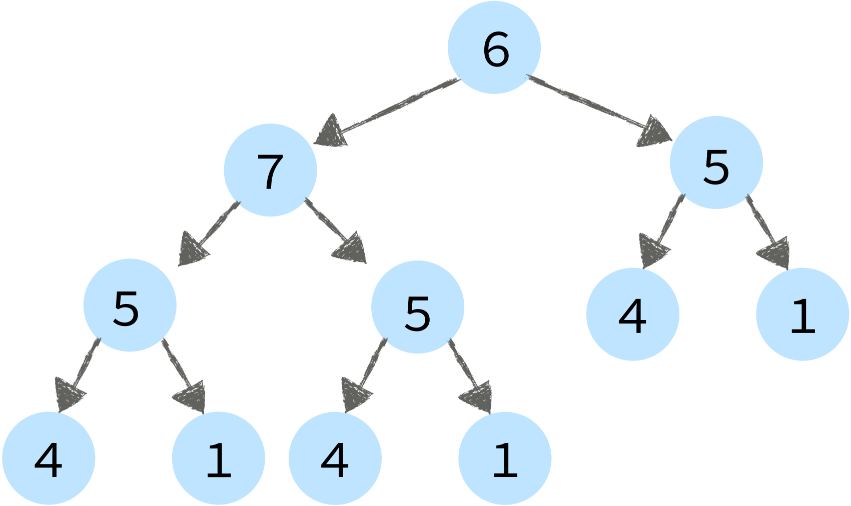
Binary Heap: (max-ordered)

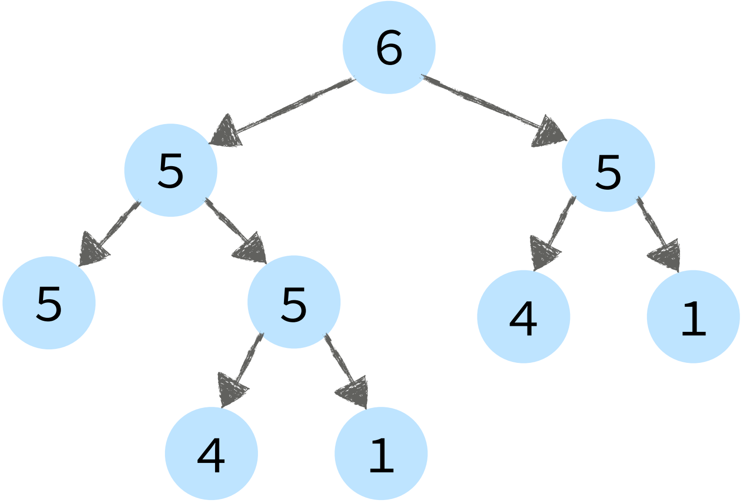
- Structure: Must be a complete binary tree.
- Order: Every node is not less than its children.

? Which of the following is a binary heap?









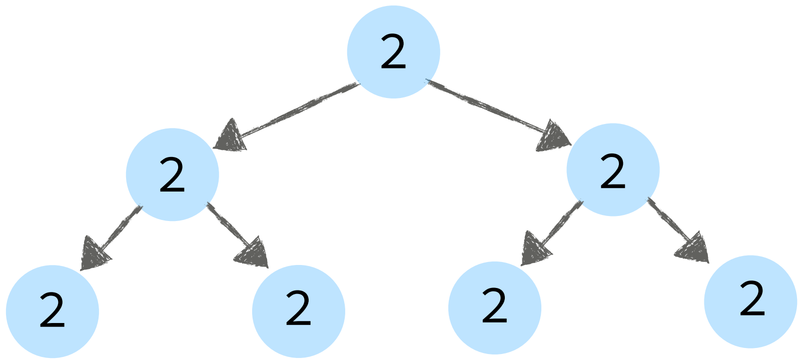
Binary Heaps (Tree Representation)

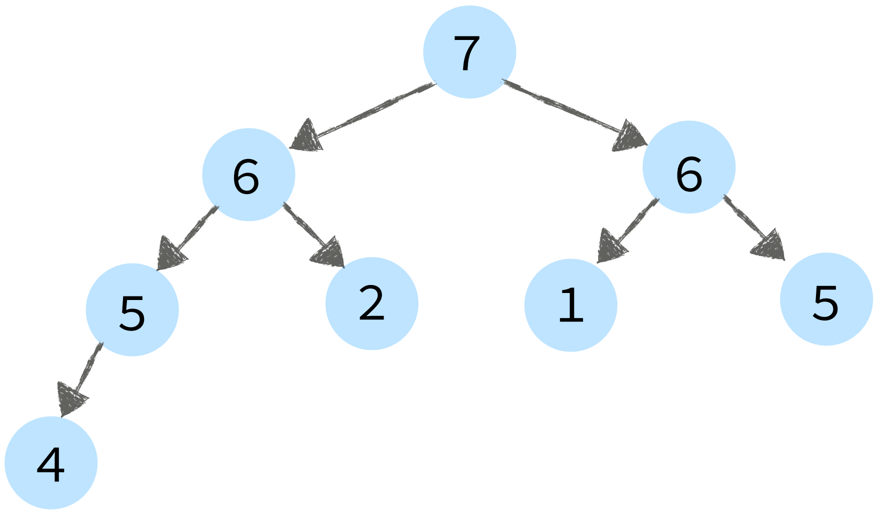
Binary Heap: (max-ordered)

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

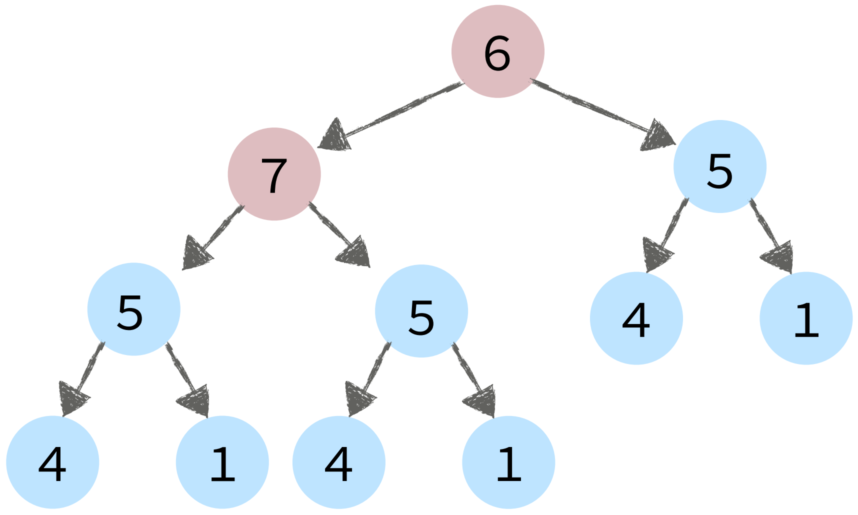
? Which of the following is a binary heap?

Examples:

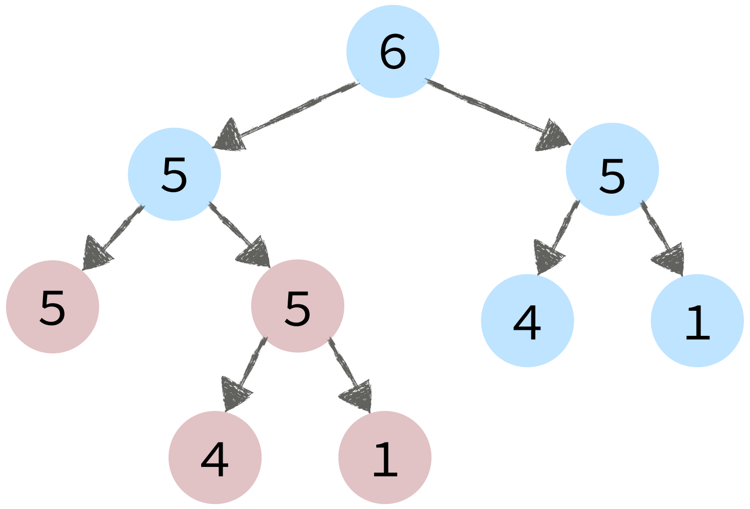




Non-Examples:



order property violated



structure property violated

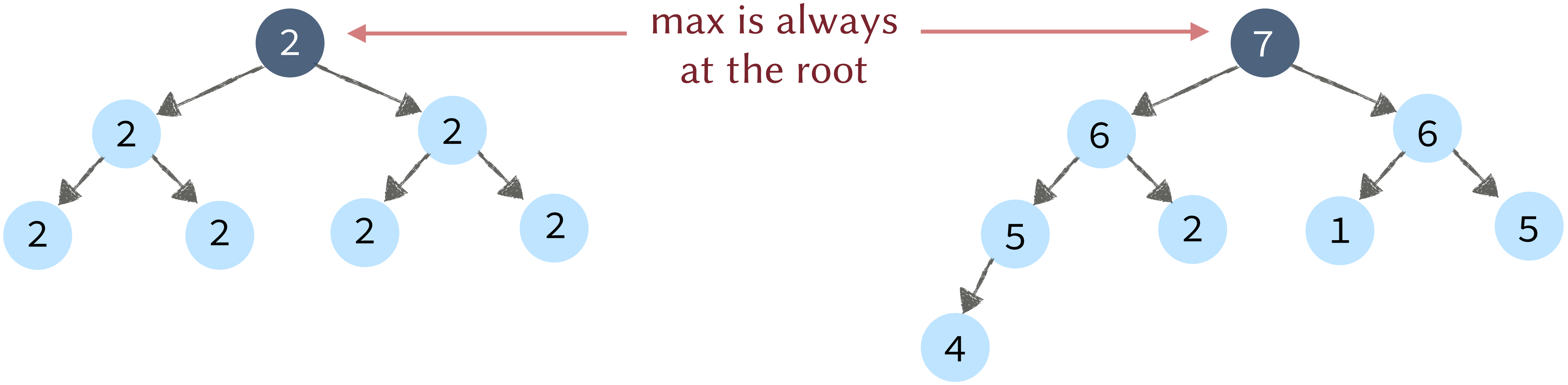
Binary Heaps (Tree Representation)

Binary Heap: (max-ordered)

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

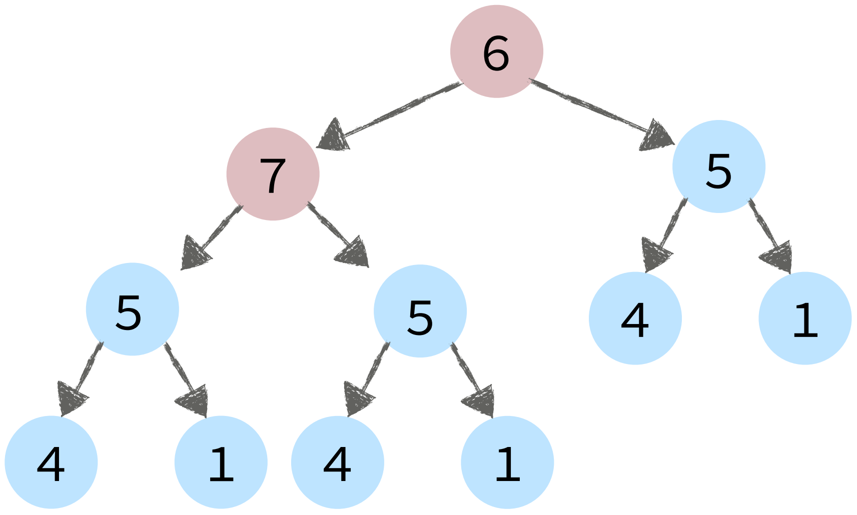
? Which of the following is a binary heap?

Examples:

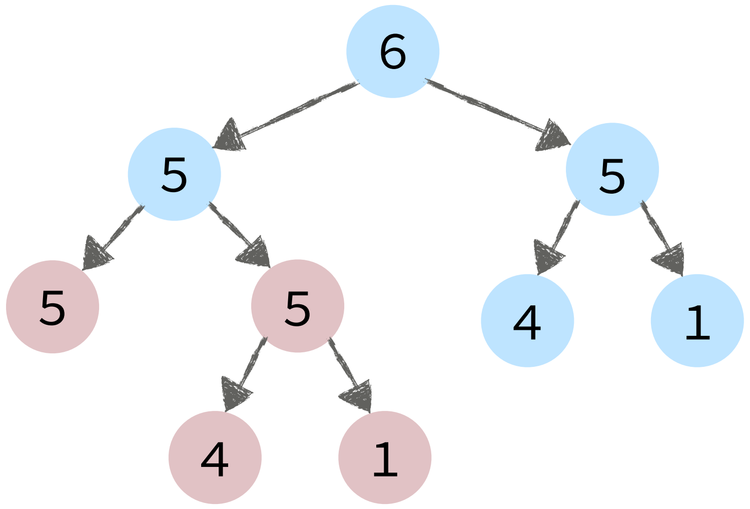


max is always at the root

Non-Examples:



order property violated



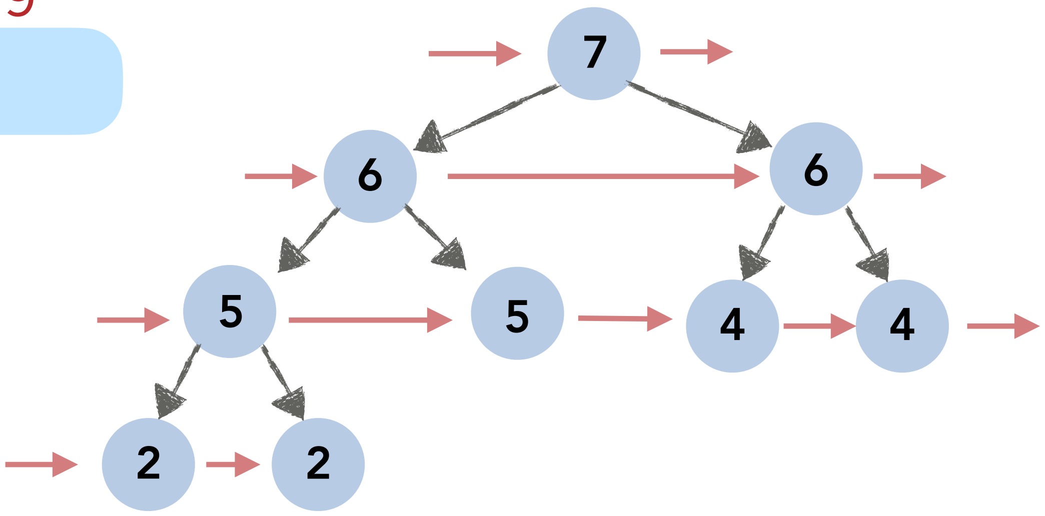
structure property violated

Binary Heaps (Array Representation)

Binary Heap: (max-ordered)

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

array has the tree nodes in level-order



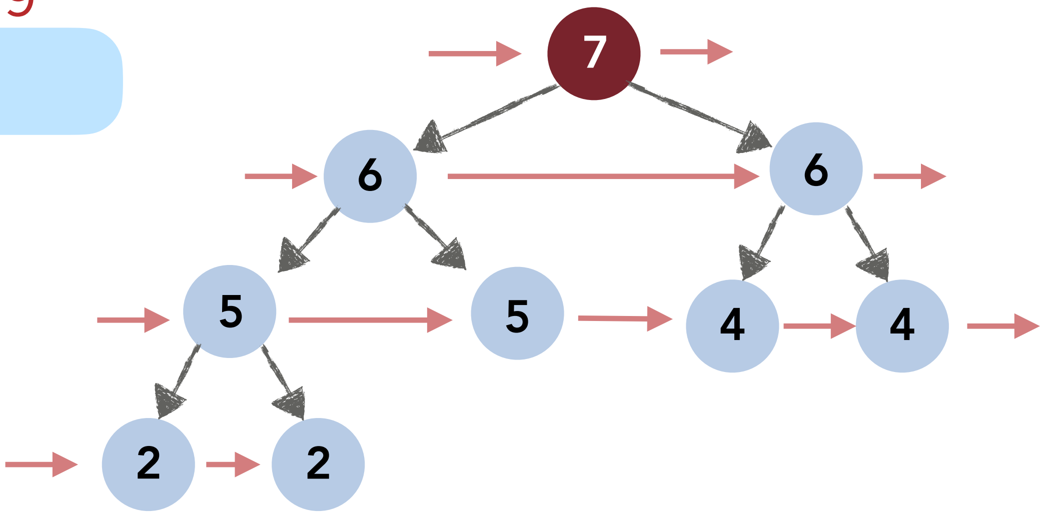
Binary Heaps (Array Representation)

Binary Heap: (max-ordered)

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

The root is always at index 0

array has the tree nodes in level-order

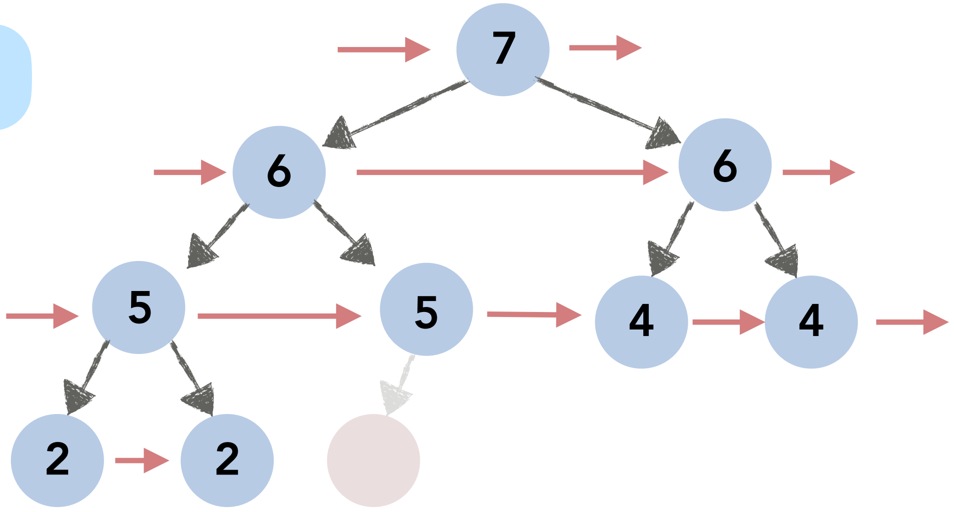


Binary Heaps (Array Representation)

Binary Heap: (max-ordered)

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

array has the tree nodes in level-order



The first empty node in the last level in the tree is the first empty cell in the array

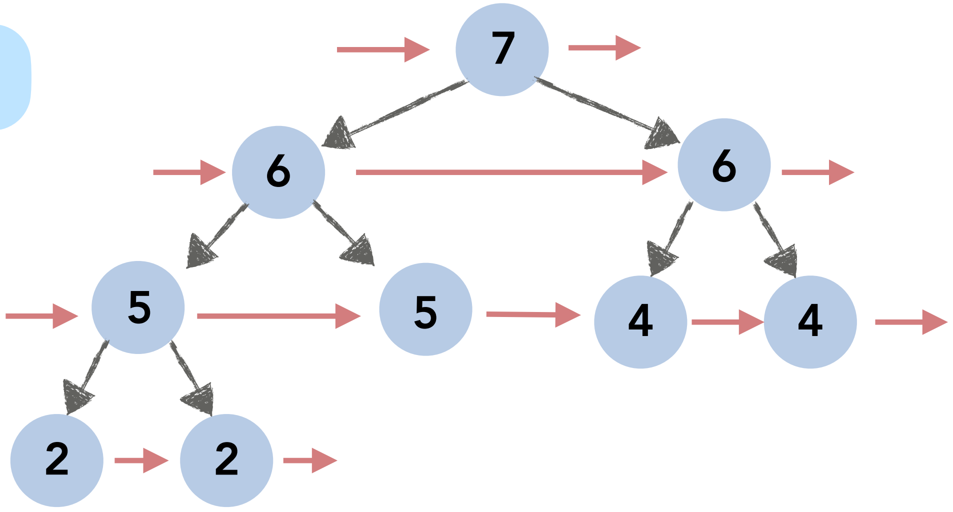
Binary Heaps (Array Representation)

Binary Heap: (max-ordered)

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

? Given the index *i* of an element, what are the indices of the left and right children of that element?

array has the tree nodes in level-order

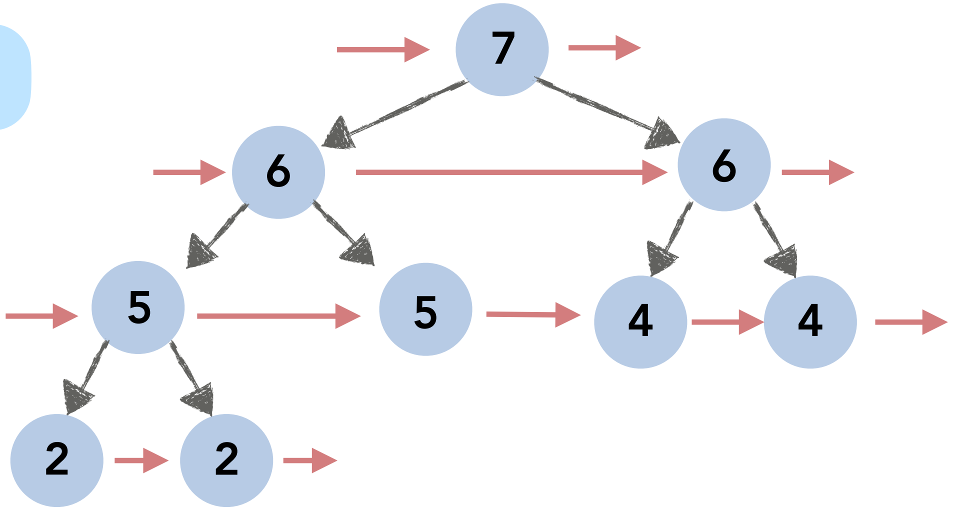


Binary Heaps (Array Representation)

Binary Heap: (max-ordered)

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

array has the tree nodes in level-order



? Given the index *i* of an element, what are the indices of the left and right children of that element?

Three simple functions.

```
int LEFT(int i)
  return 2*i + 1;
```

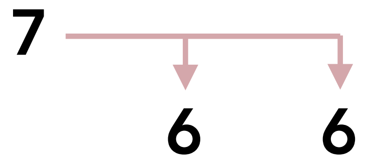
```
int RIGHT(int i)
  return 2*i + 2;
```

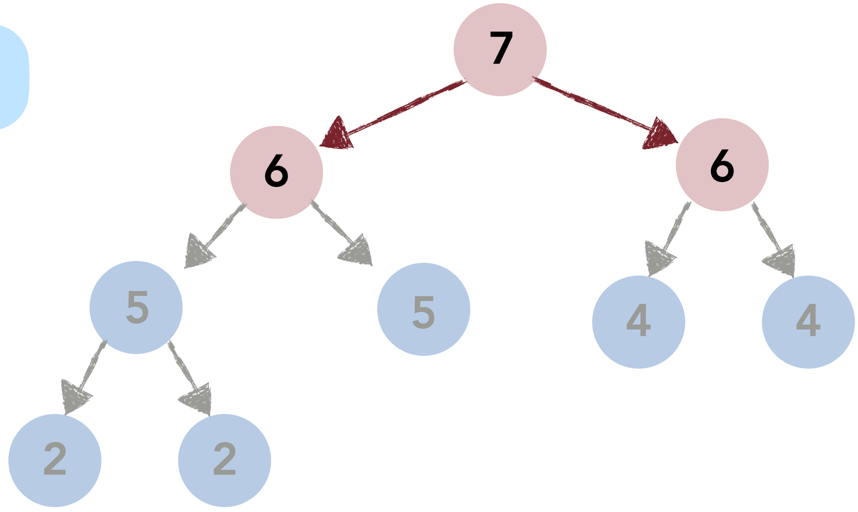
```
int PARENT(int i)
  return (i-1) / 2;
```

Binary Heaps (Array Representation)

Binary Heap: (max-ordered)

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





Three simple functions.

```
int LEFT(int i)

return 2*i + 1;
```

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

```
int RIGHT(int i)

return 2*i + 2;
```

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

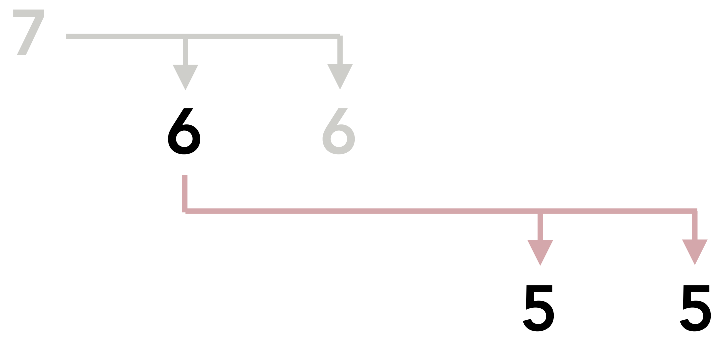
```
int PARENT(int i)

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

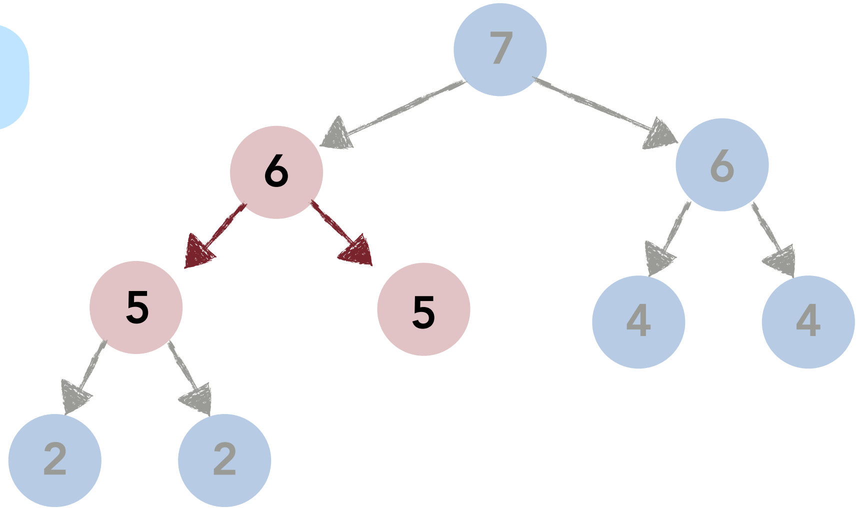
Parent of the node at `0` is `0`
I.e. It has no parent

Binary Heaps (Array Representation)

Binary Heap: (max-ordered)

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





Three simple functions.

```
int LEFT(int i)

return 2*i + 1;
```

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

```
int RIGHT(int i)

return 2*i + 2;
```

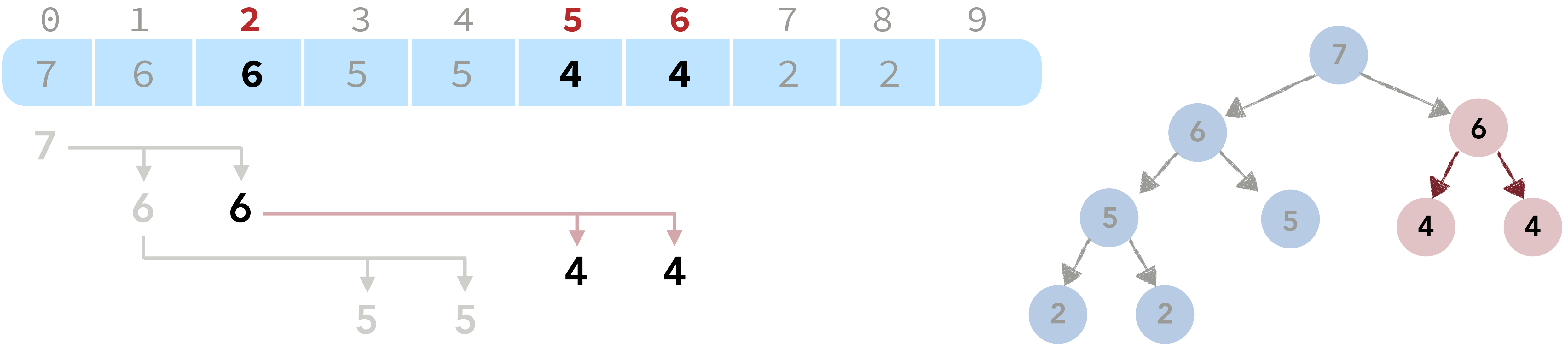
Right child is at index
2*1 + 2 = 4

```
int PARENT(int i)

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

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

Binary Heaps (Array Representation)

Binary Heap: (max-ordered)



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

Three simple functions.

```
int LEFT(int i)

return 2*i + 1;
```

left child is at index

2*2 + 1 = 5

```
int RIGHT(int i)

return 2*i + 2;
```

Right child is at index

2*2 + 2 = 6

```
int PARENT(int i)

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

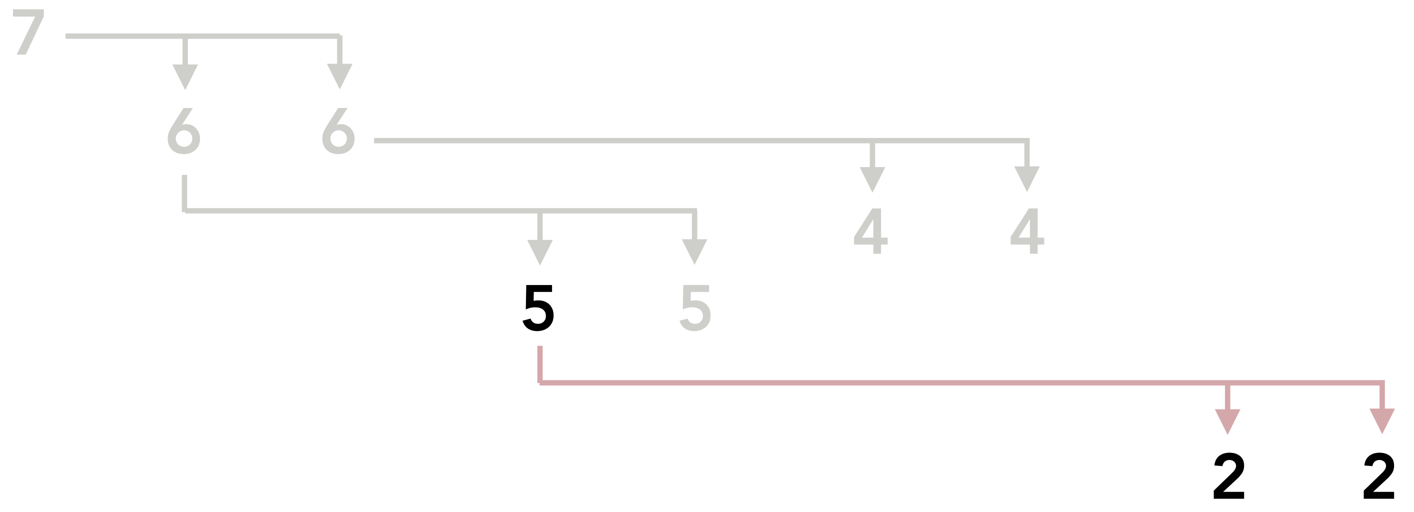
Parent is at index

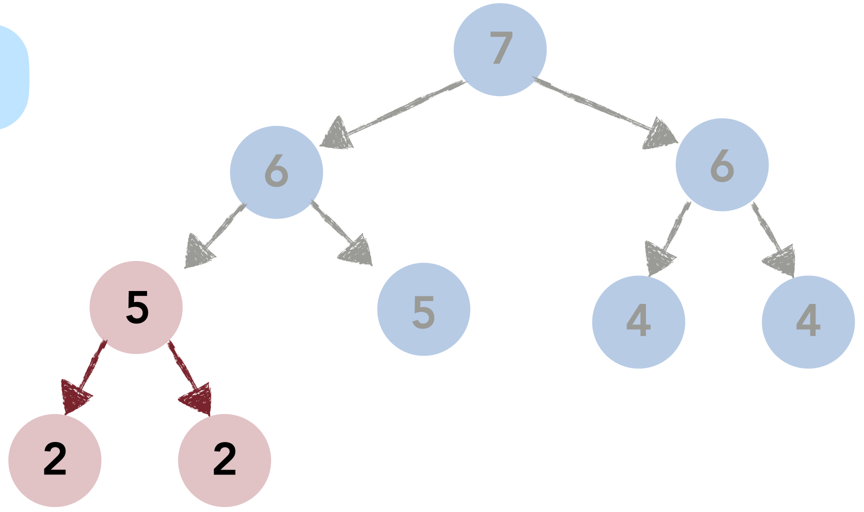
(2-1)/2 = 0

Binary Heaps (Array Representation)

Binary Heap: (max-ordered)

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





Three simple functions.

```
int LEFT(int i)

return 2*i + 1;
```

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

```
int RIGHT(int i)

return 2*i + 2;
```

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

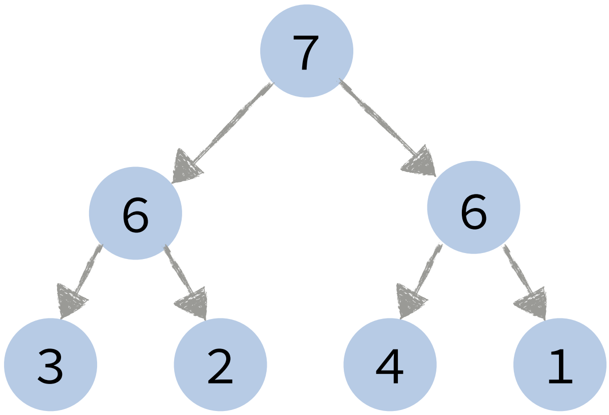
```
int PARENT(int i)

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

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

Binary Heaps: Insertion

Basic Plan.

1. Insert respecting the *structure* property.
2. Maintain the *order* property.



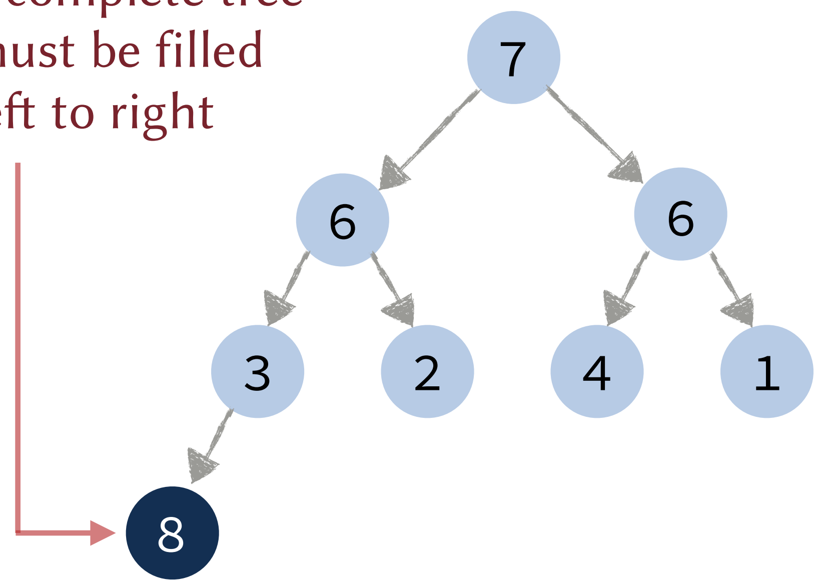
0	1	2	3	4	5	6	7	8
7	6	6	3	2	4	1		

Binary Heaps: Insertion

Basic Plan.

1. Insert respecting the *structure* property.
2. Maintain the *order* property.

Example. Insert **8**



a complete tree must be filled left to right

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

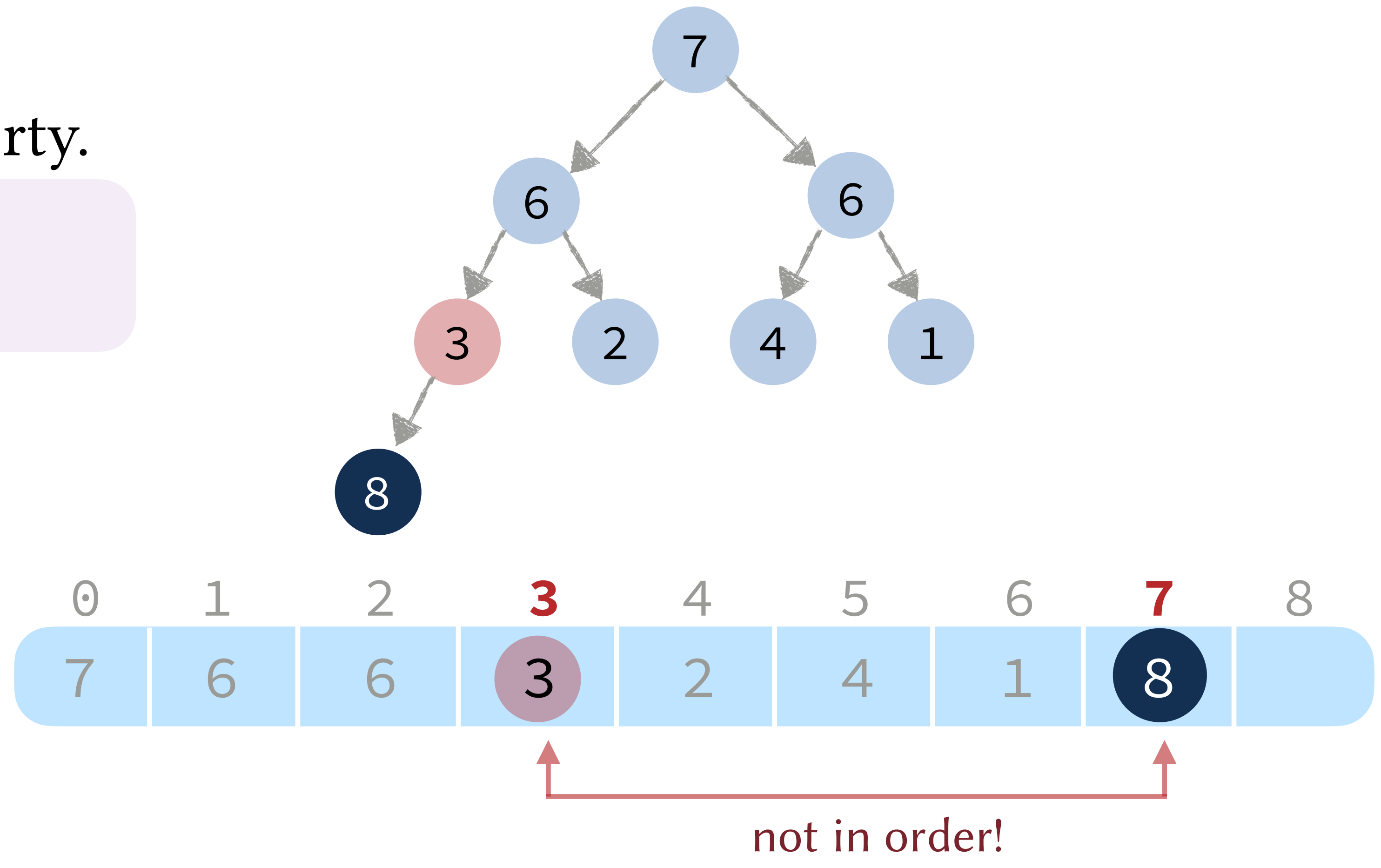
adding to the last index is equivalent to filling the last level left-to-right

Binary Heaps: Insertion

Basic Plan.

1. Insert respecting the *structure* property.
2. Maintain the *order* property.
   swap up until the heap is fixed

Example. Insert **8**



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

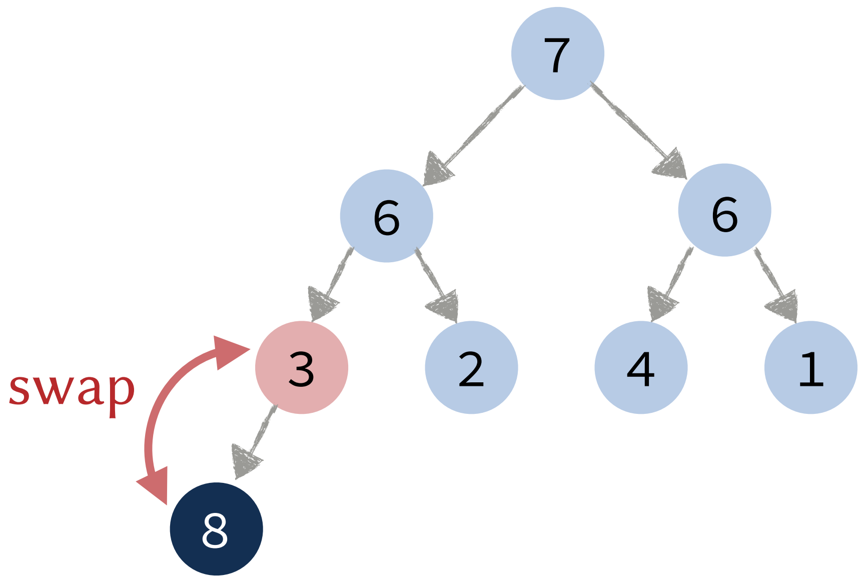
not in order!

Binary Heaps: Insertion

Basic Plan.

1. Insert respecting the *structure* property.
2. Maintain the *order* property.
   swap up until the heap is fixed

Example. Insert **8**



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

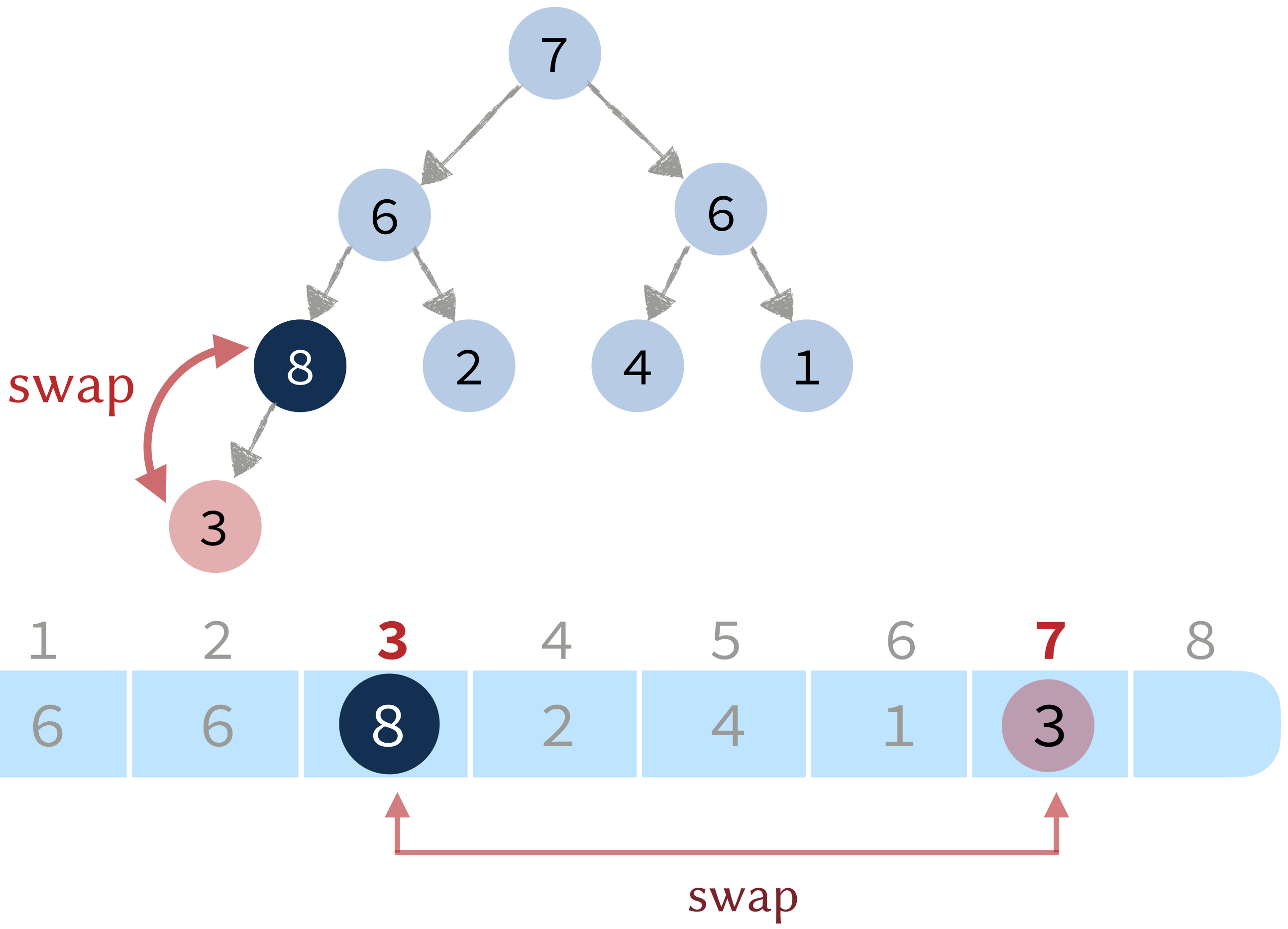
swap

Binary Heaps: Insertion

Basic Plan.

1. Insert respecting the *structure* property.
2. Maintain the *order* property.
   swap up until the heap is fixed

Example. Insert **8**



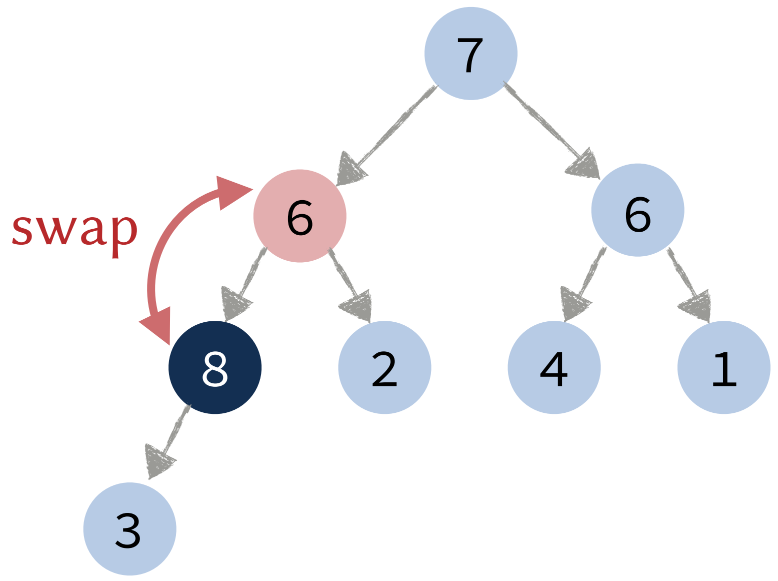
0	1	2	3	4	5	6	7	8
7	6	6	8	2	4	1	3	

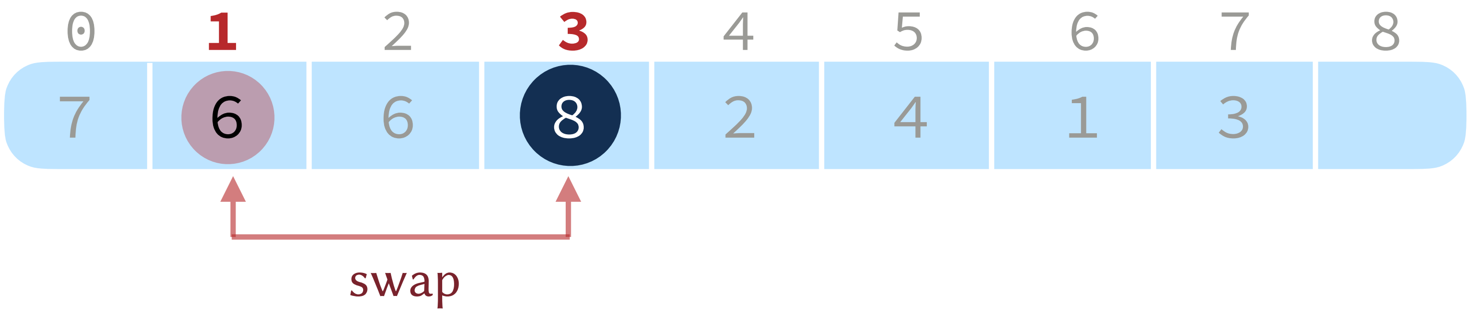
Binary Heaps: Insertion

Basic Plan.

1. Insert respecting the *structure* property.
2. Maintain the *order* property.
   swap up until the heap is fixed

Example. Insert **8**



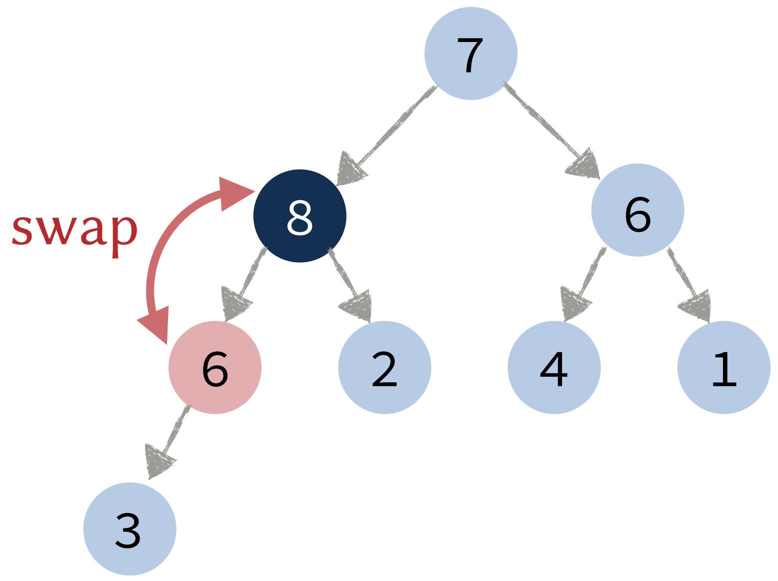


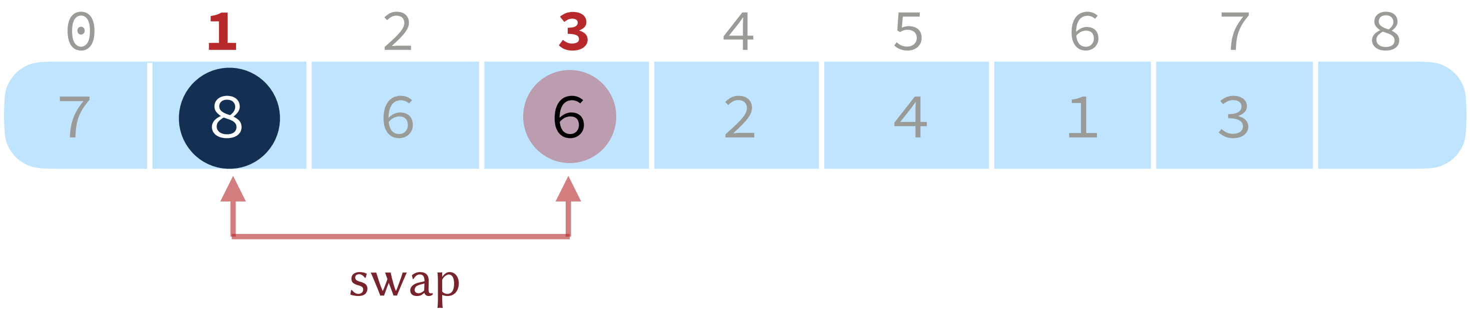
Binary Heaps: Insertion

Basic Plan.

1. Insert respecting the *structure* property.
2. Maintain the *order* property.
   swap up until the heap is fixed

Example. Insert **8**



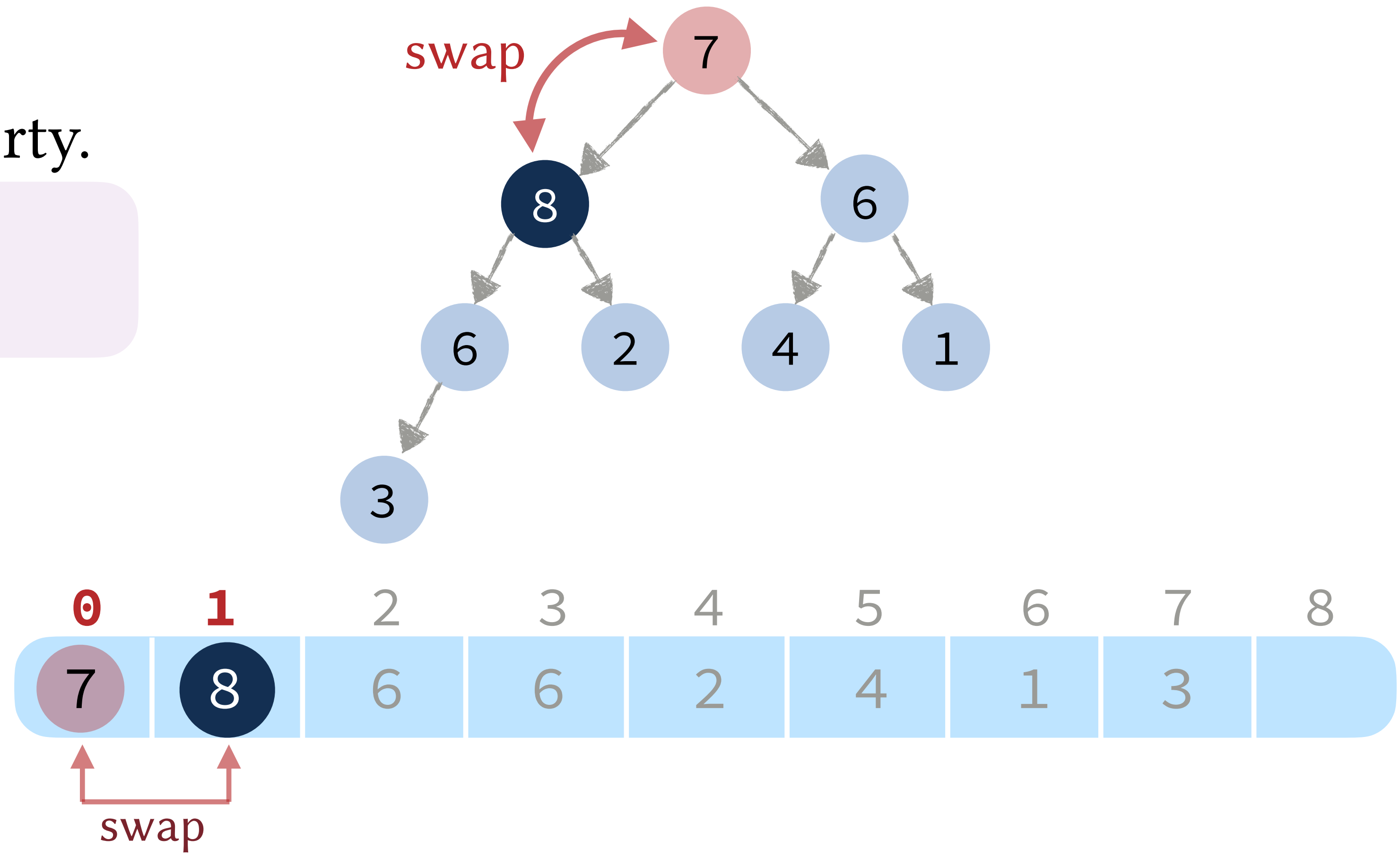


Binary Heaps: Insertion

Basic Plan.

1. Insert respecting the *structure* property.
2. Maintain the *order* property.
   swap up until the heap is fixed

Example. Insert **8**



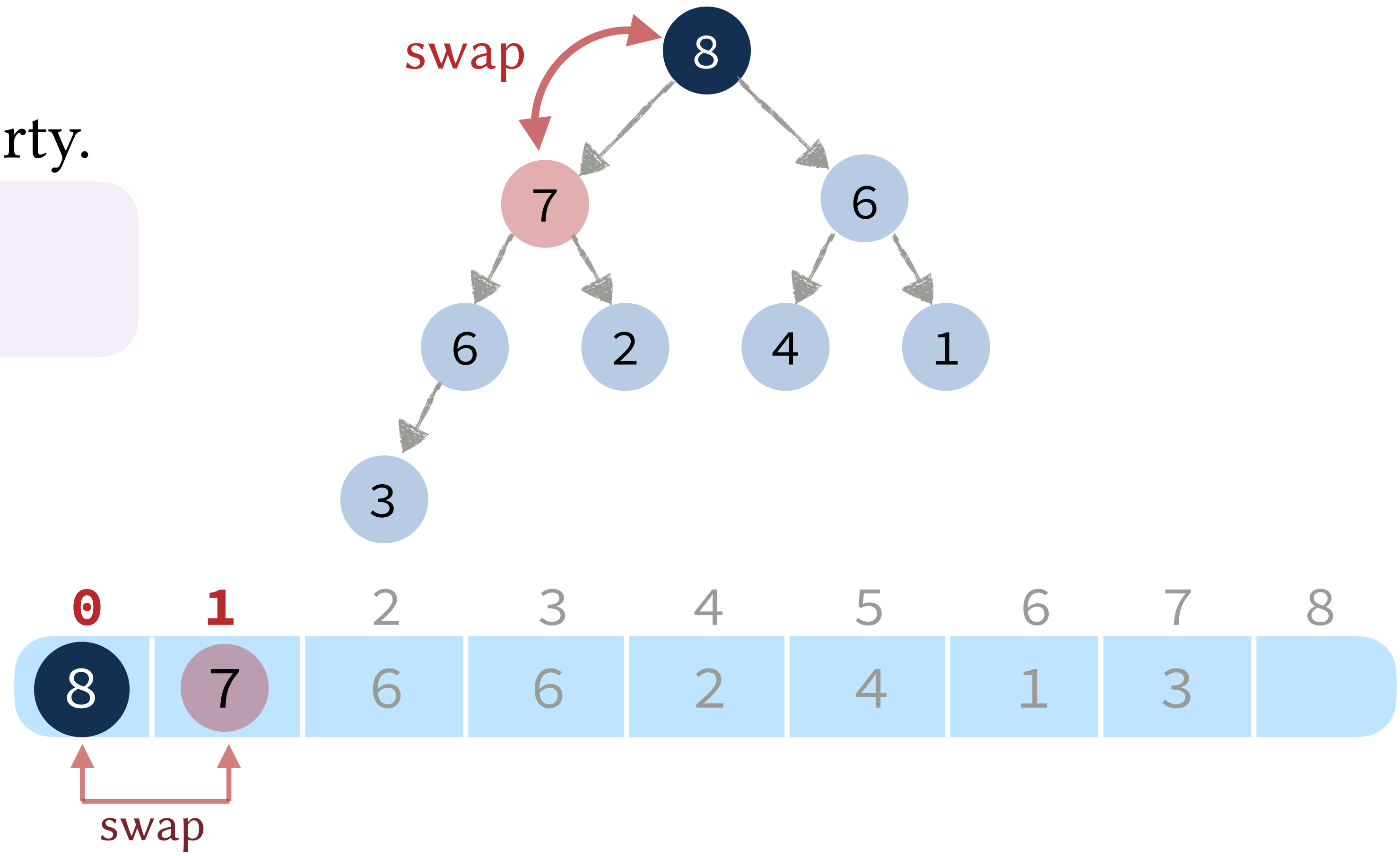
0	1	2	3	4	5	6	7	8
7	8	6	6	2	4	1	3	

Binary Heaps: Insertion

Basic Plan.

1. Insert respecting the *structure* property.
2. Maintain the *order* property.
   swap up until the heap is fixed

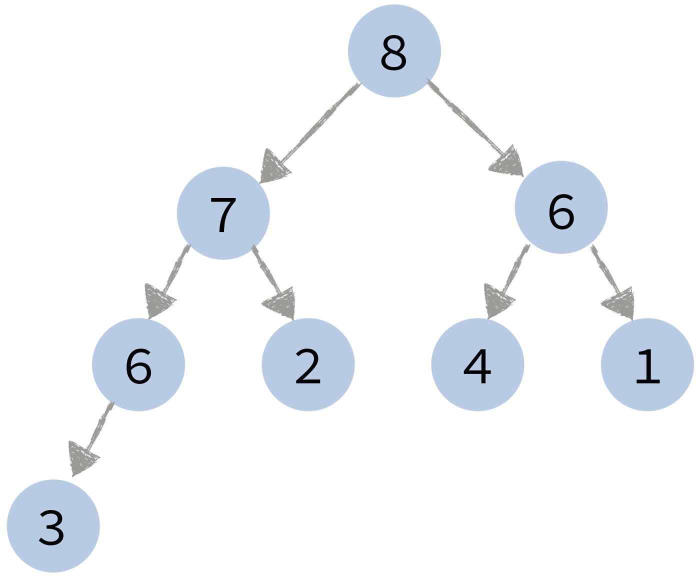
Example. Insert **8**



Binary Heaps: Insertion

Basic Plan.

1. Insert respecting the *structure* property.
2. Maintain the *order* property.



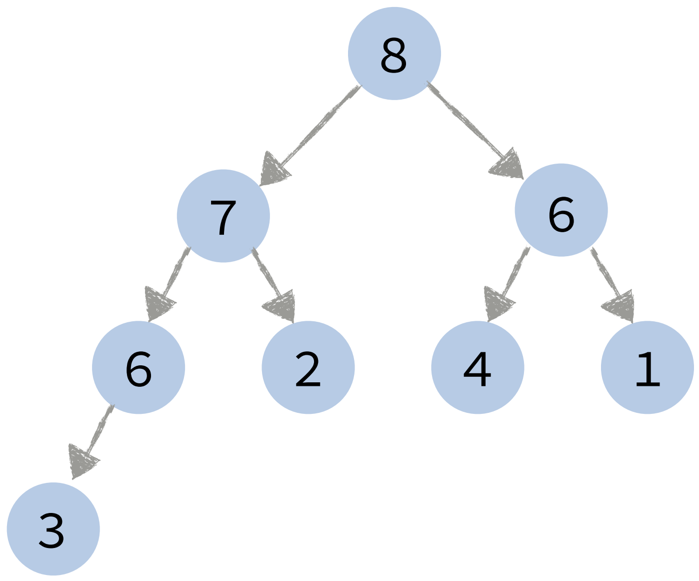
0	1	2	3	4	5	6	7	8
8	7	6	6	2	4	1	3	

Binary Heaps: Insertion

Basic Plan.

1. Insert respecting the *structure* property.
2. Maintain the *order* property.

Example. Insert **5**



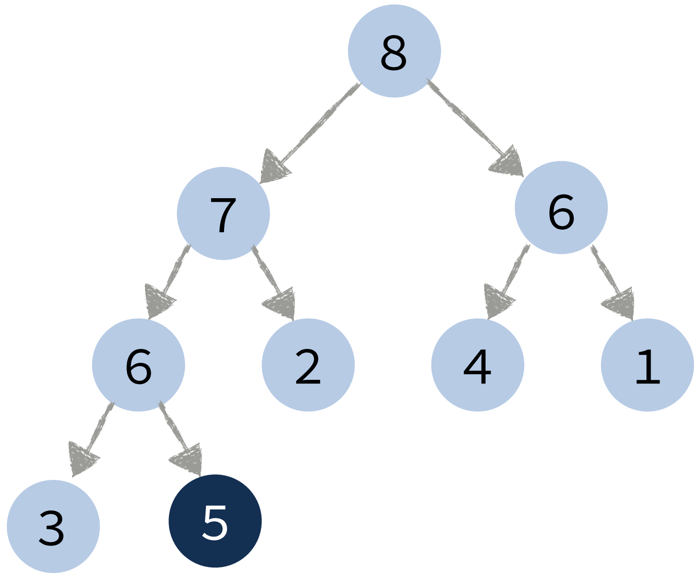
0	1	2	3	4	5	6	7	8
8	7	6	6	2	4	1	3	

Binary Heaps: Insertion

Basic Plan.

1. Insert respecting the *structure* property.
2. Maintain the *order* property.

Example. Insert **5**



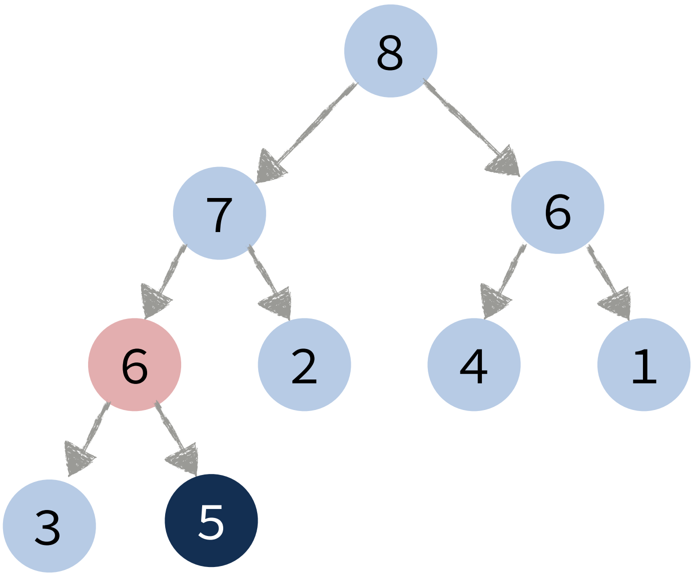
0	1	2	3	4	5	6	7	**8**
8	7	6	6	2	4	1	3	5

Binary Heaps: Insertion

Basic Plan.

1. Insert respecting the *structure* property.
2. Maintain the *order* property.

Example. Insert **5**



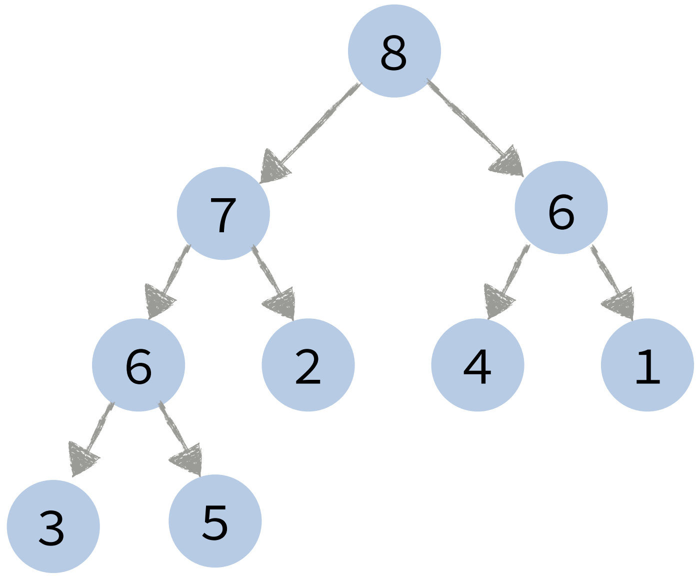
0	1	2	**3**	4	5	6	7	**8**
8	7	6	6	2	4	1	3	5

Already in order!
No swaps needed!

Binary Heaps: Insertion

Basic Plan.

1. Insert respecting the *structure* property.
2. Maintain the *order* property.



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

```
void insert(int a[], int& size, int val) {
    a[size++] = val;
    int i = size-1;
    while (i > 0  &&  a[i] > a[PARENT(i)]) {
        swap(a[i], a[PARENT(i)]);
        i = PARENT(i);
    }
}
```

optional

Binary Heaps: Insertion

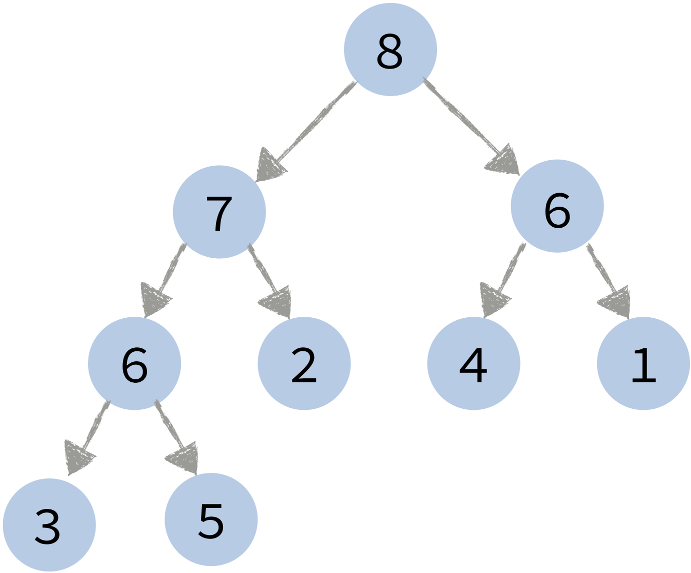
Basic Plan.

1. Insert respecting the *structure* property.
2. Maintain the *order* property.

Running Time.

Best Case: 0 swaps and 1 data compare.

Worst Case: $\lfloor \log_2 n \rfloor$ swaps and
$\lfloor \log_2 n \rfloor$ data compares.

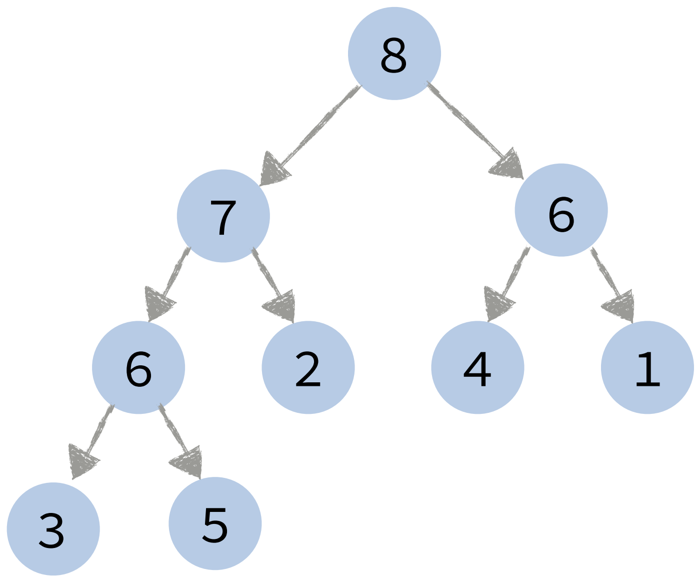


0	1	2	3	4	5	6	7	8
8	7	6	6	2	4	1	3	5

```
void insert(int a[], int& size, int val) {
    a[size++] = val;
    int i = size-1;
    while (i > 0  &&  a[i] > a[PARENT(i)]) {
        swap(a[i], a[PARENT(i)]);
        i = PARENT(i);
    }
}
```

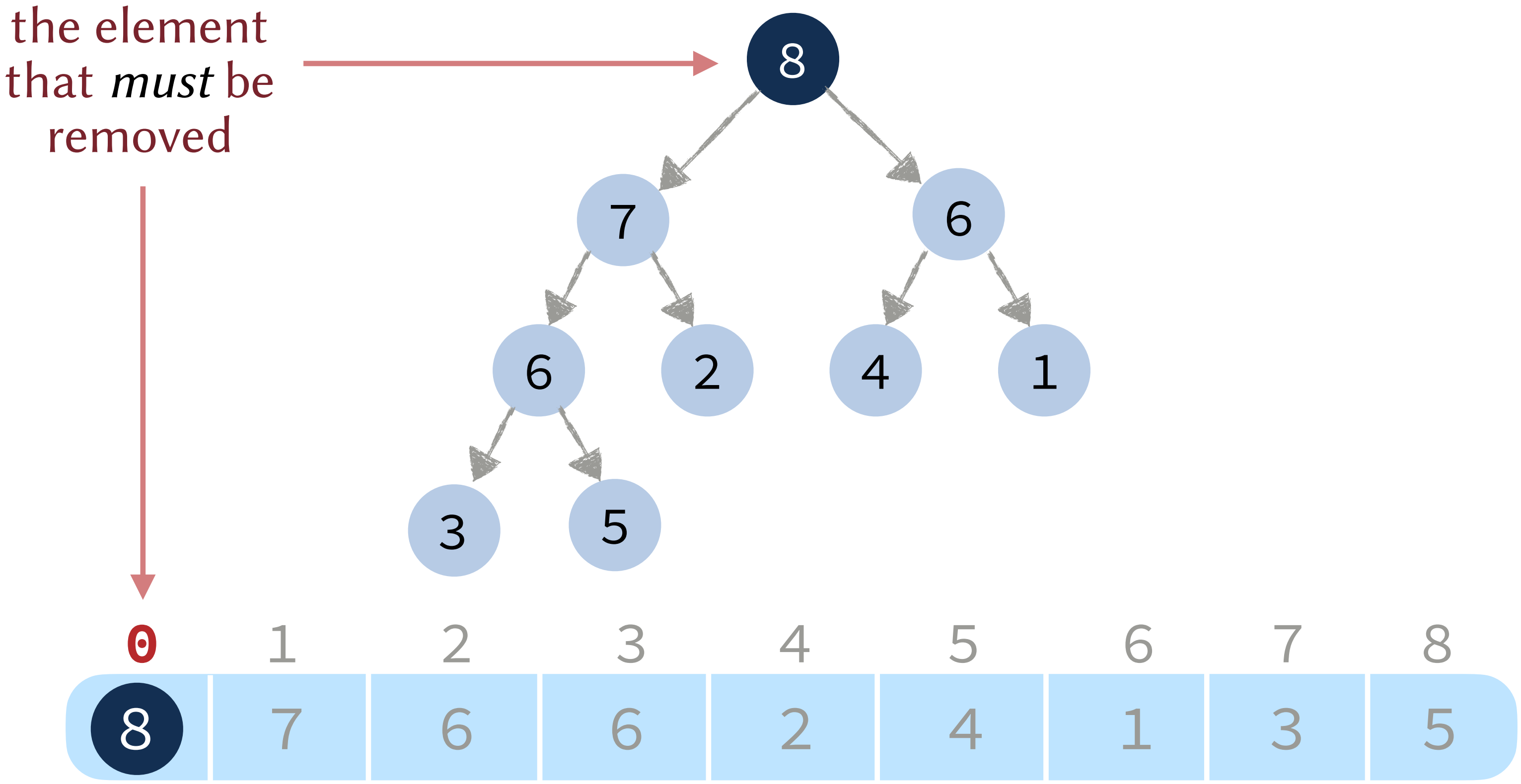
optional

Binary Heaps: Deletion



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

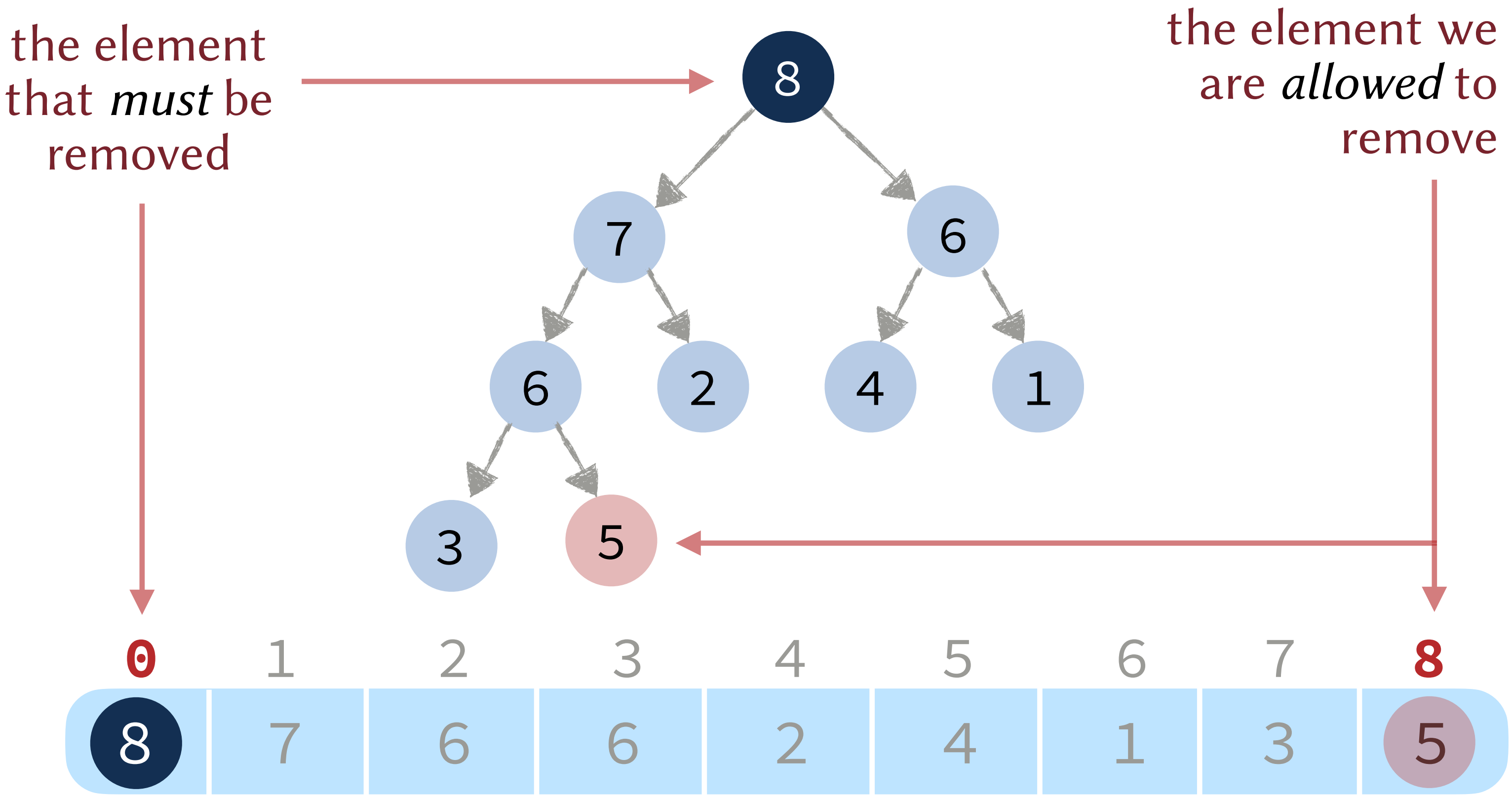
Binary Heaps: Deletion



the element that *must* be removed

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

Binary Heaps: Deletion



the element that *must* be removed

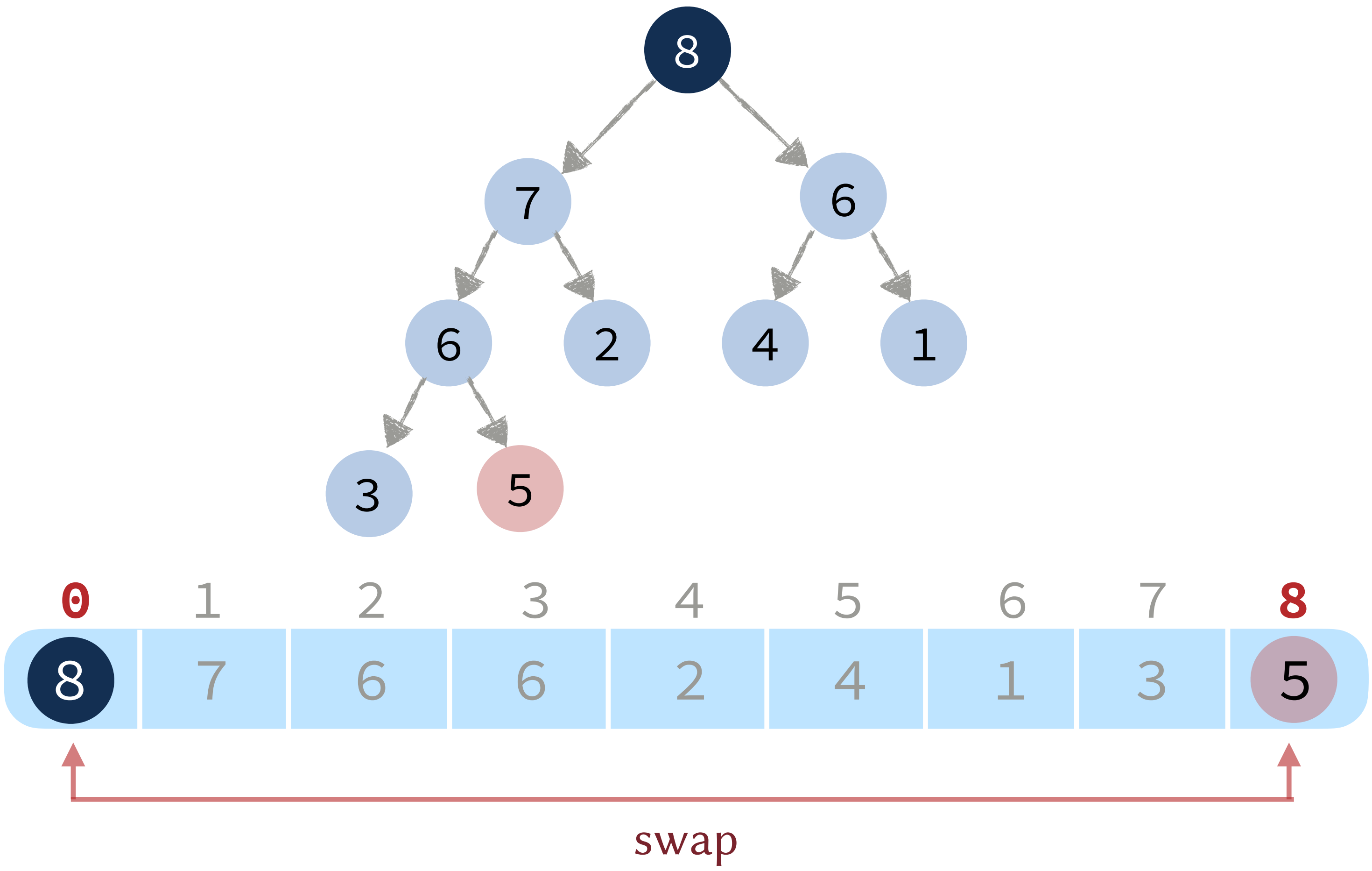
the element we are *allowed* to remove

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

Binary Heaps: Deletion

Basic Plan.

1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.

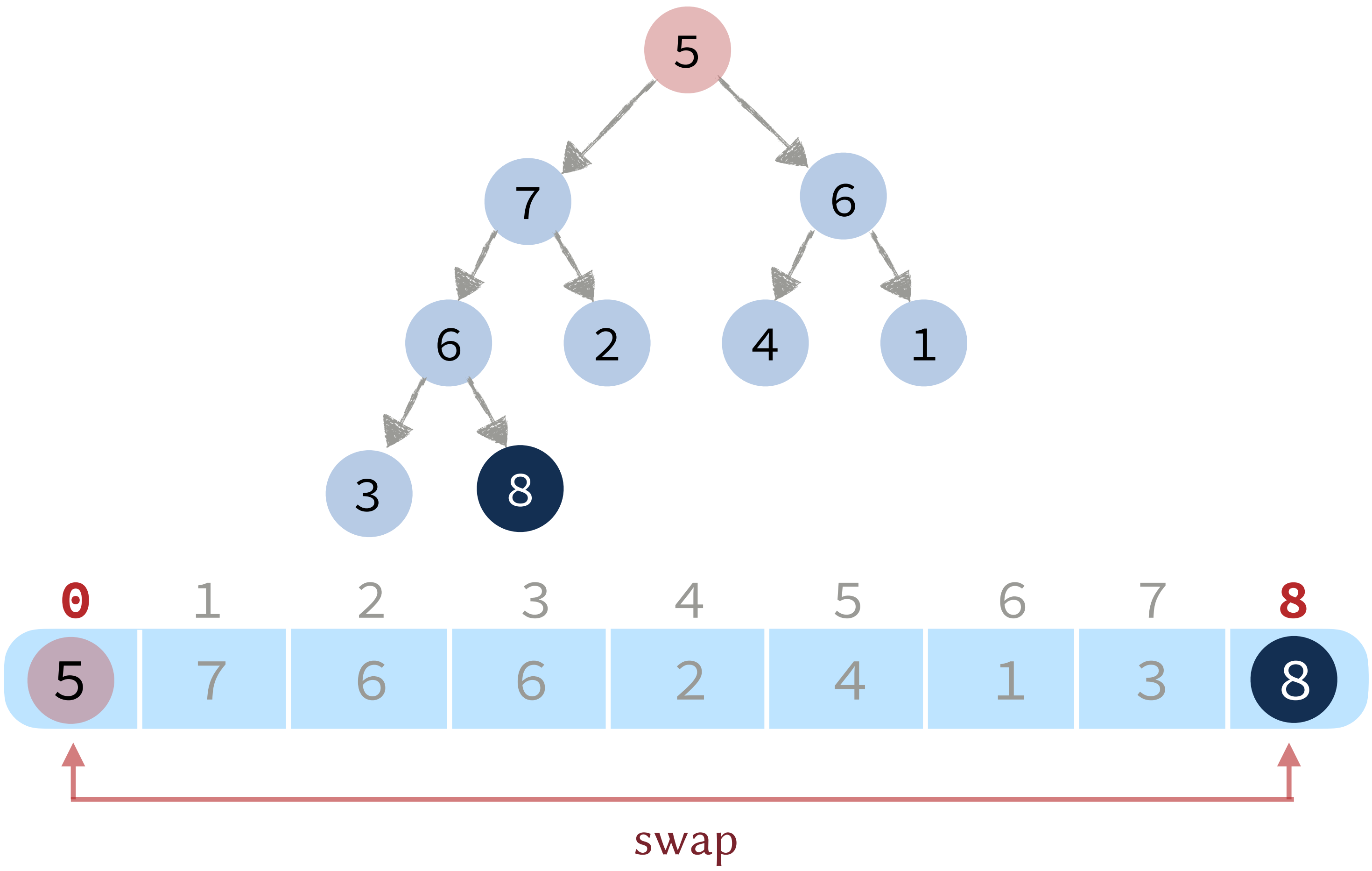


0	1	2	3	4	5	6	7	8
8	7	6	6	2	4	1	3	5

Binary Heaps: Deletion

Basic Plan.

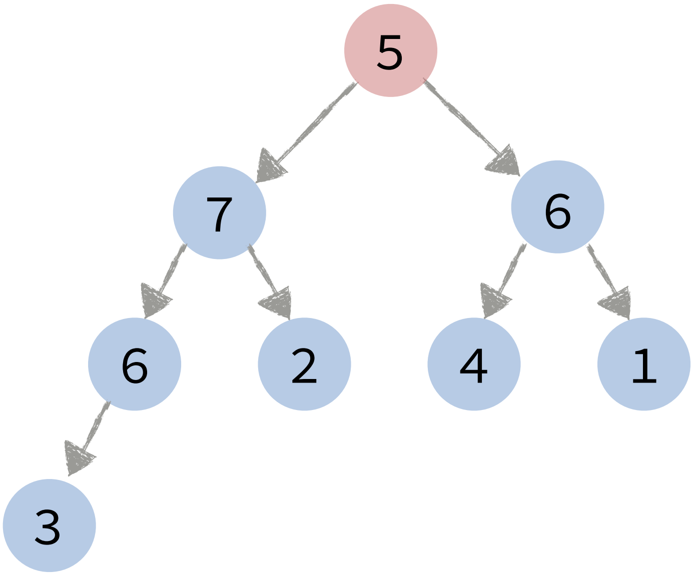
1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.



Binary Heaps: Deletion

Basic Plan.

1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.

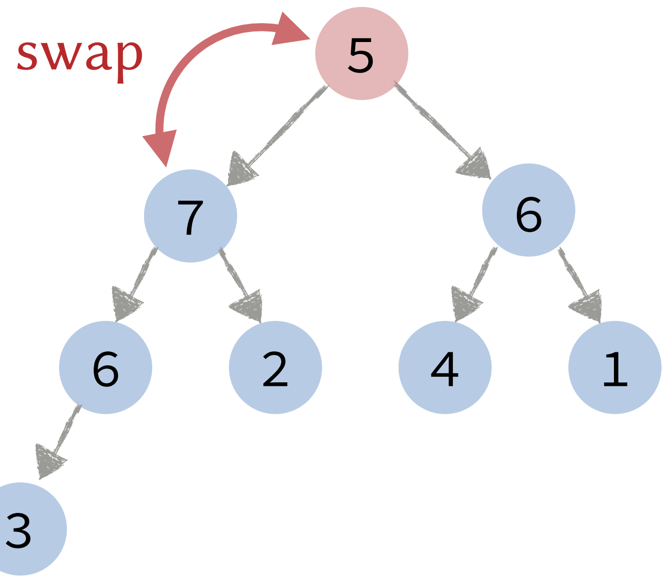


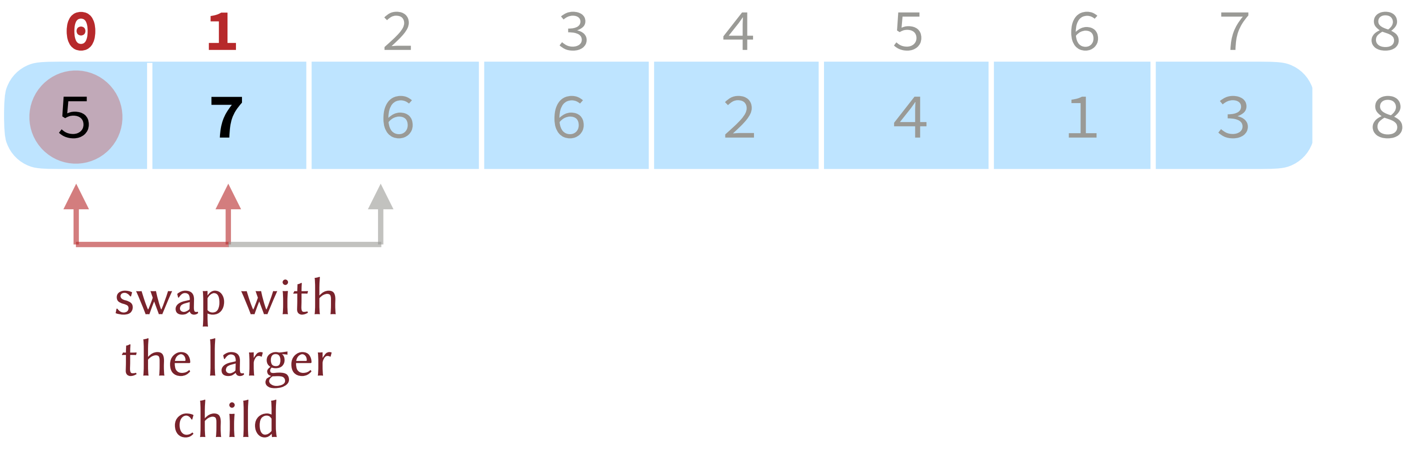
0	1	2	3	4	5	6	7	8
5	7	6	6	2	4	1	3	8

Binary Heaps: Deletion

Basic Plan.

1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.





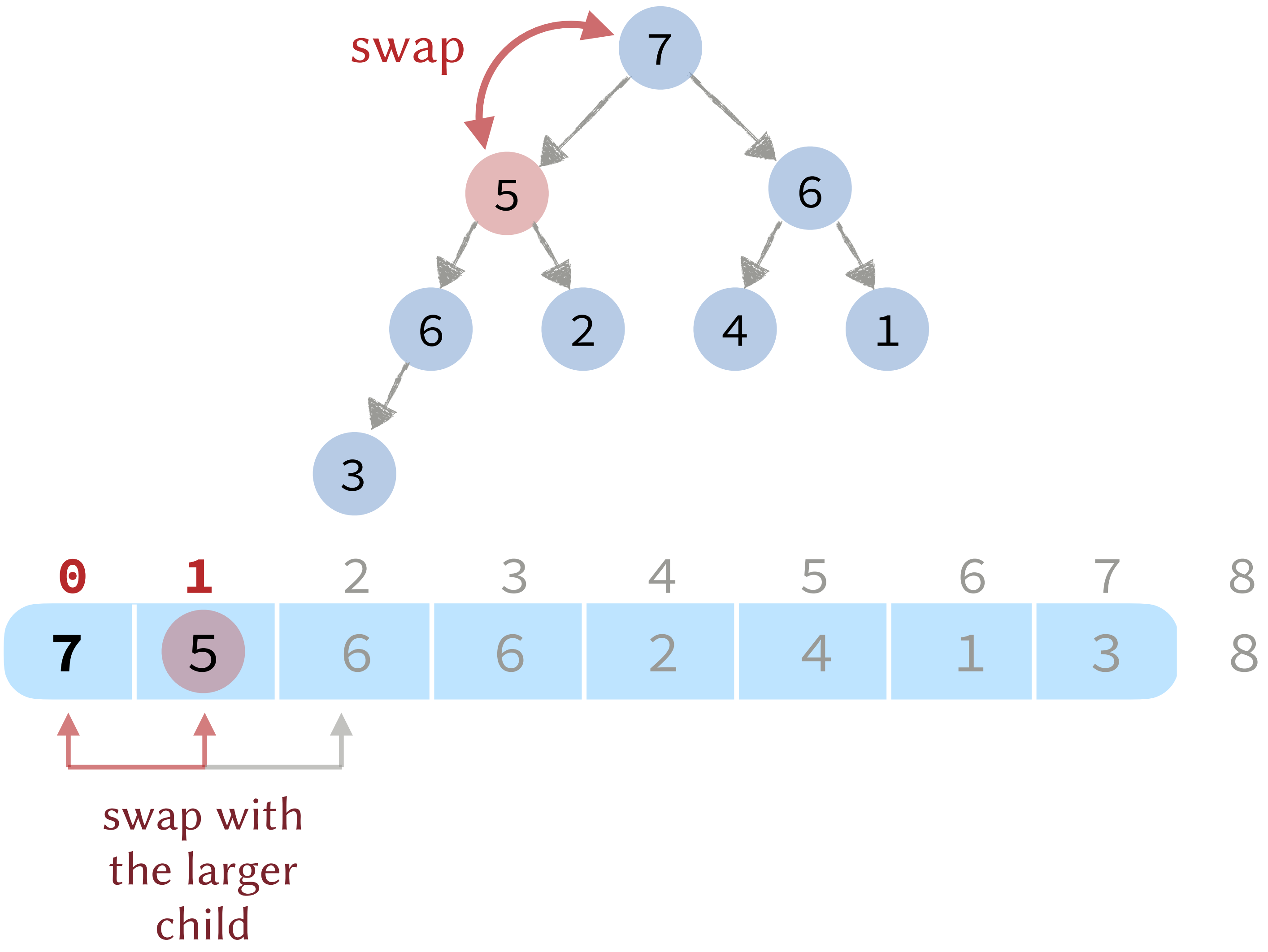
0	1	2	3	4	5	6	7	8
5	7	6	6	2	4	1	3	8

swap with the larger child

Binary Heaps: Deletion

Basic Plan.

1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.



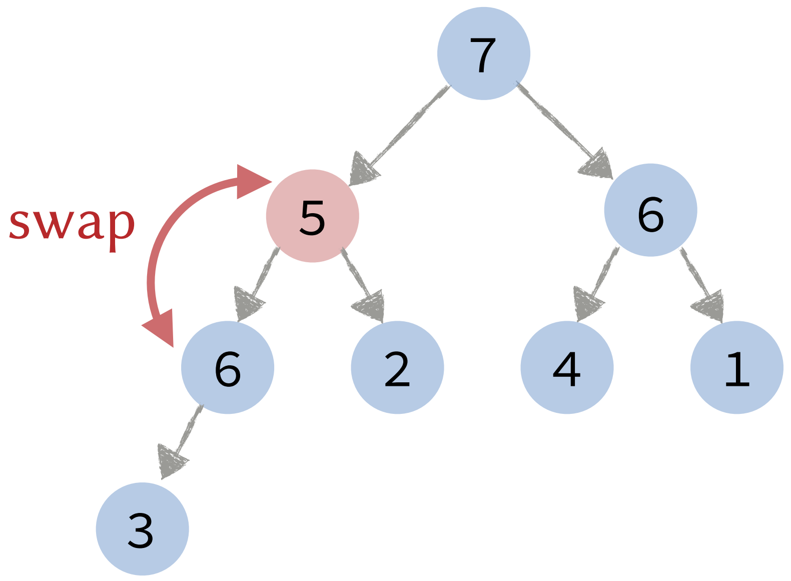
0	1	2	3	4	5	6	7	8
7	5	6	6	2	4	1	3	8

swap with the larger child

Binary Heaps: Deletion

Basic Plan.

1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.



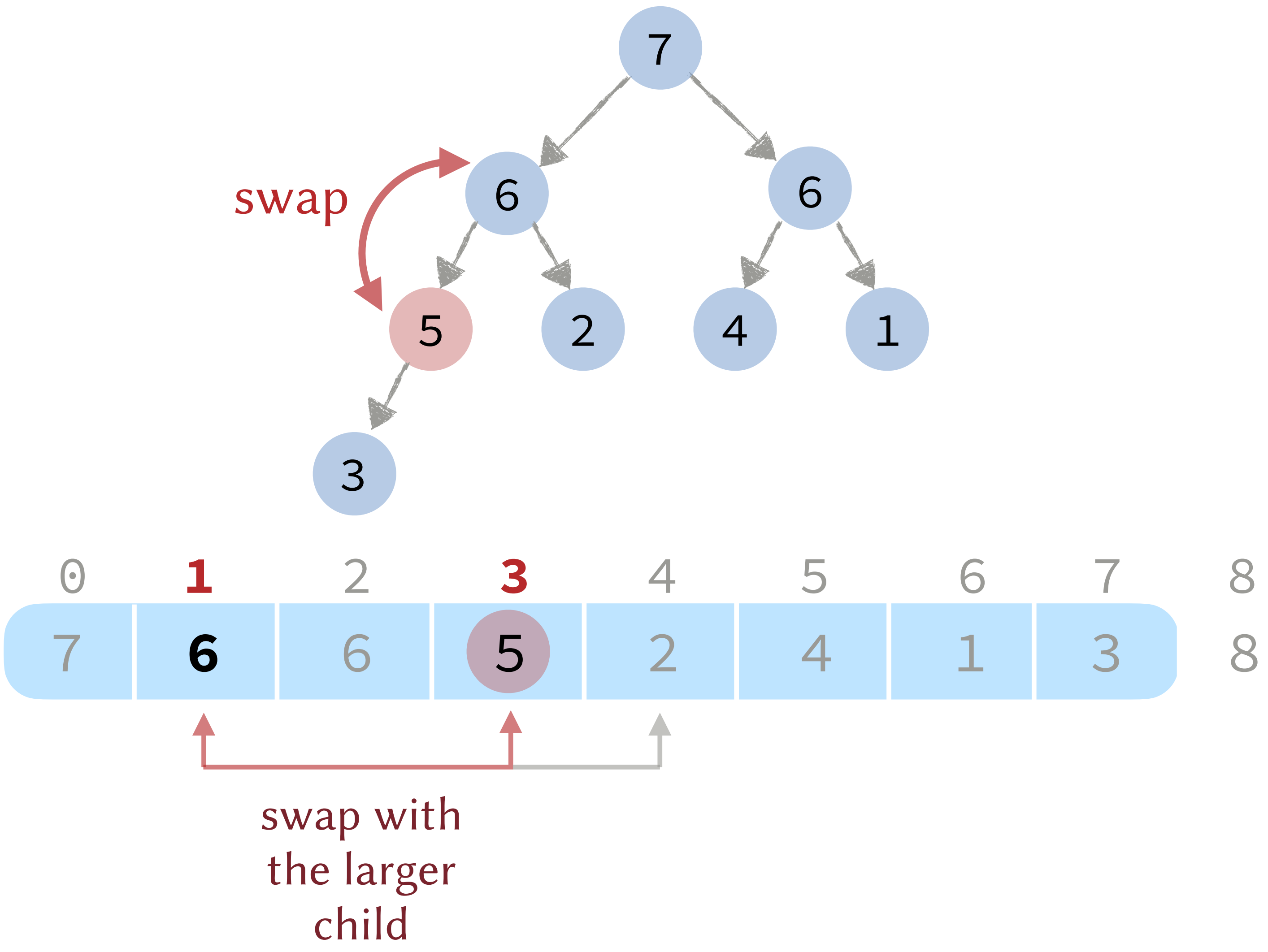
0	**1**	2	**3**	4	5	6	7	8
7	5	6	**6**	2	4	1	3	8

swap with
the larger
child

Binary Heaps: Deletion

Basic Plan.

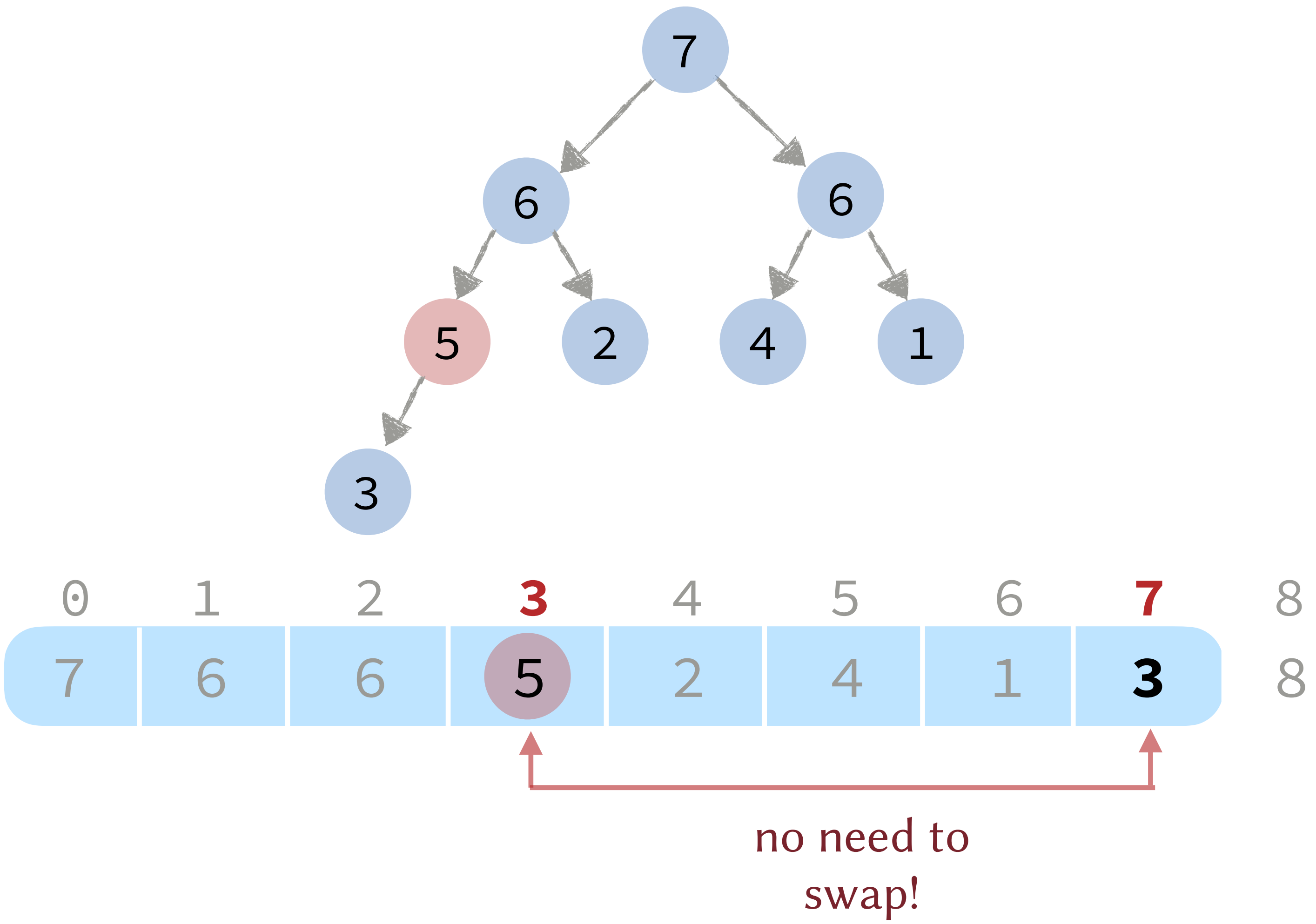
1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.



Binary Heaps: Deletion

Basic Plan.

1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.



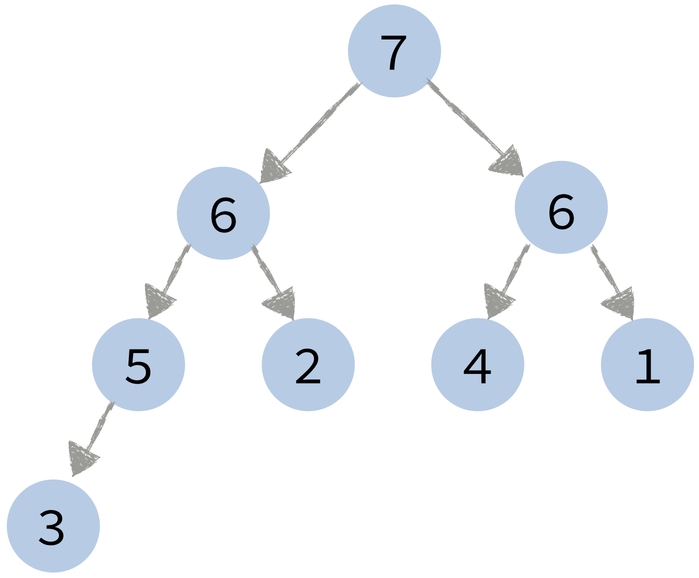
0	1	2	3	4	5	6	7	8
7	6	6	5	2	4	1	3	8

no need to swap!

Binary Heaps: Deletion

Basic Plan.

1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.

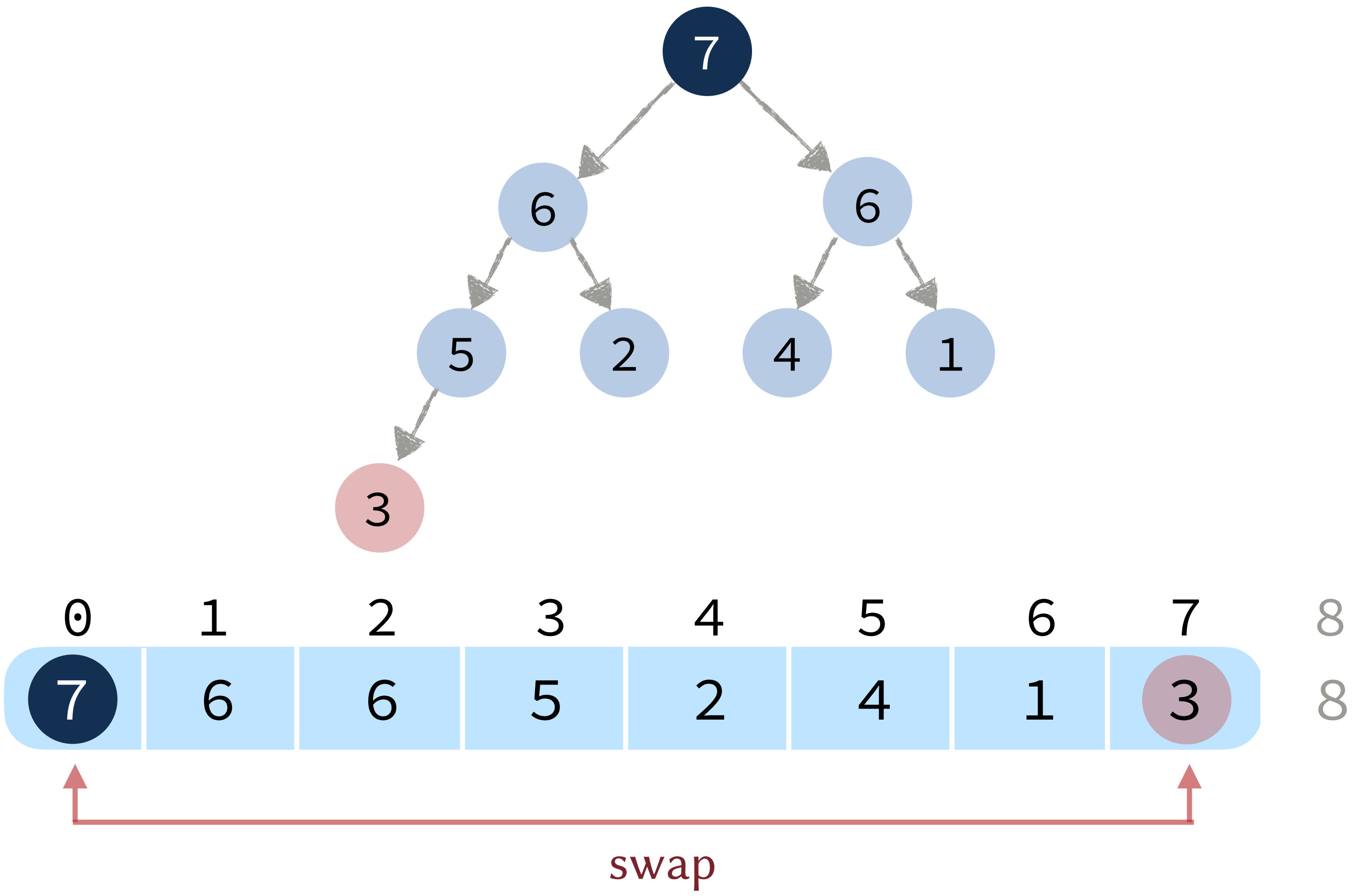


0	1	2	3	4	5	6	7	8
7	6	6	5	2	4	1	3	8

Binary Heaps: Deletion

Basic Plan.

1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.

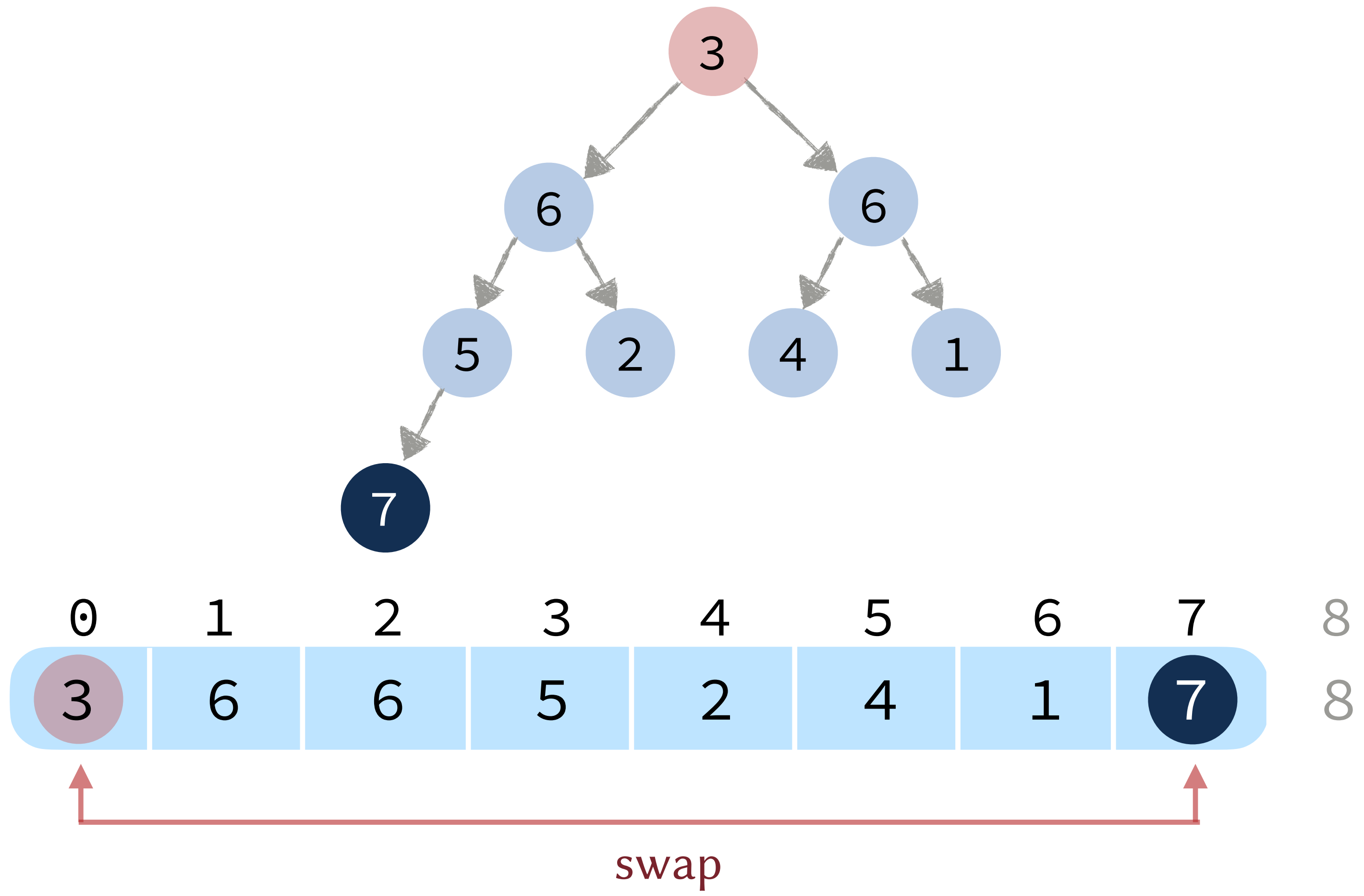


0	1	2	3	4	5	6	7	8
7	6	6	5	2	4	1	3	8

Binary Heaps: Deletion

Basic Plan.

1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.

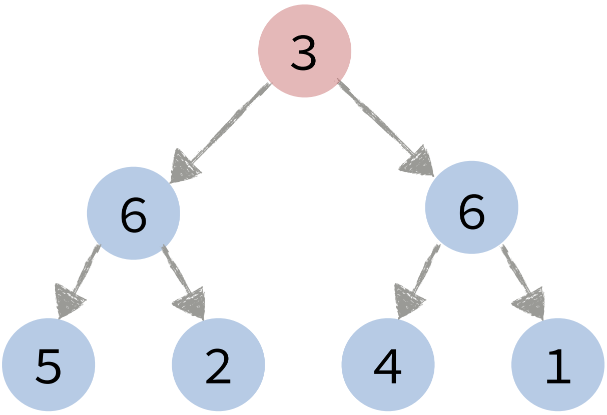


0	1	2	3	4	5	6	7	8
3	6	6	5	2	4	1	7	8

Binary Heaps: Deletion

Basic Plan.

1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.

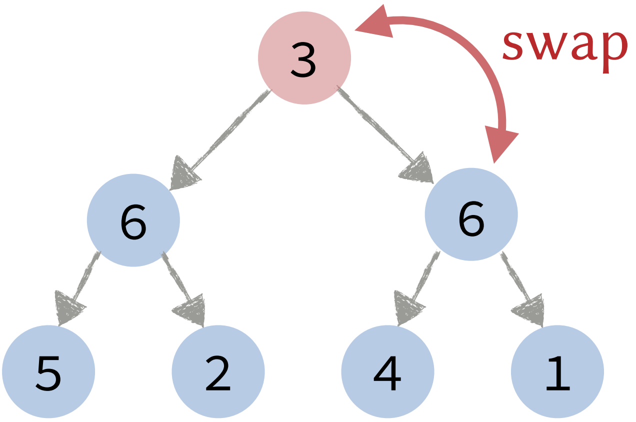


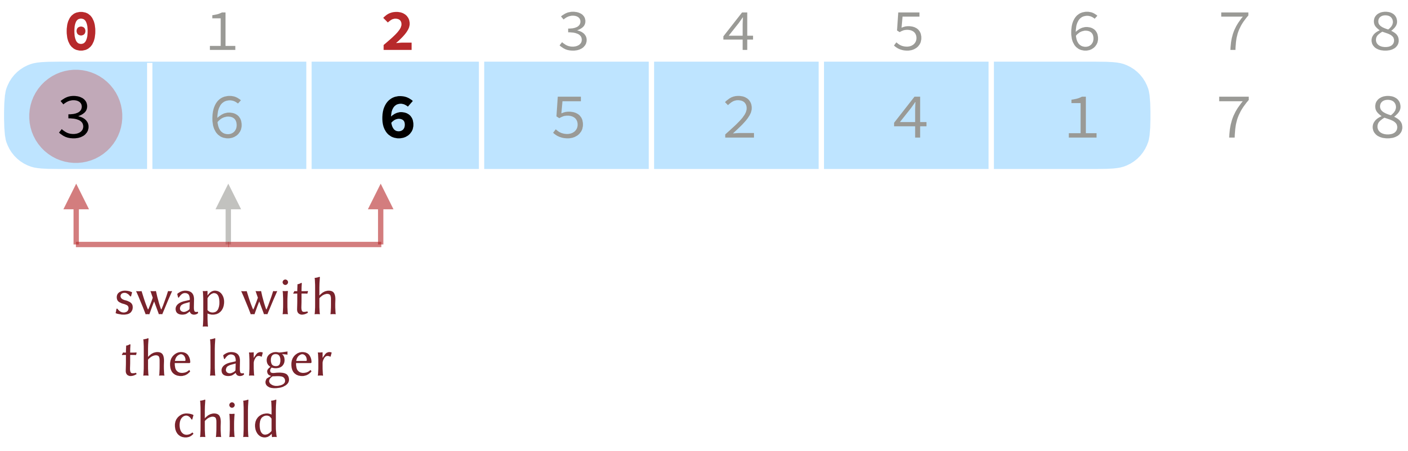
0	1	2	3	4	5	6	7	8
3	6	6	5	2	4	1	7	8

Binary Heaps: Deletion

Basic Plan.

1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.



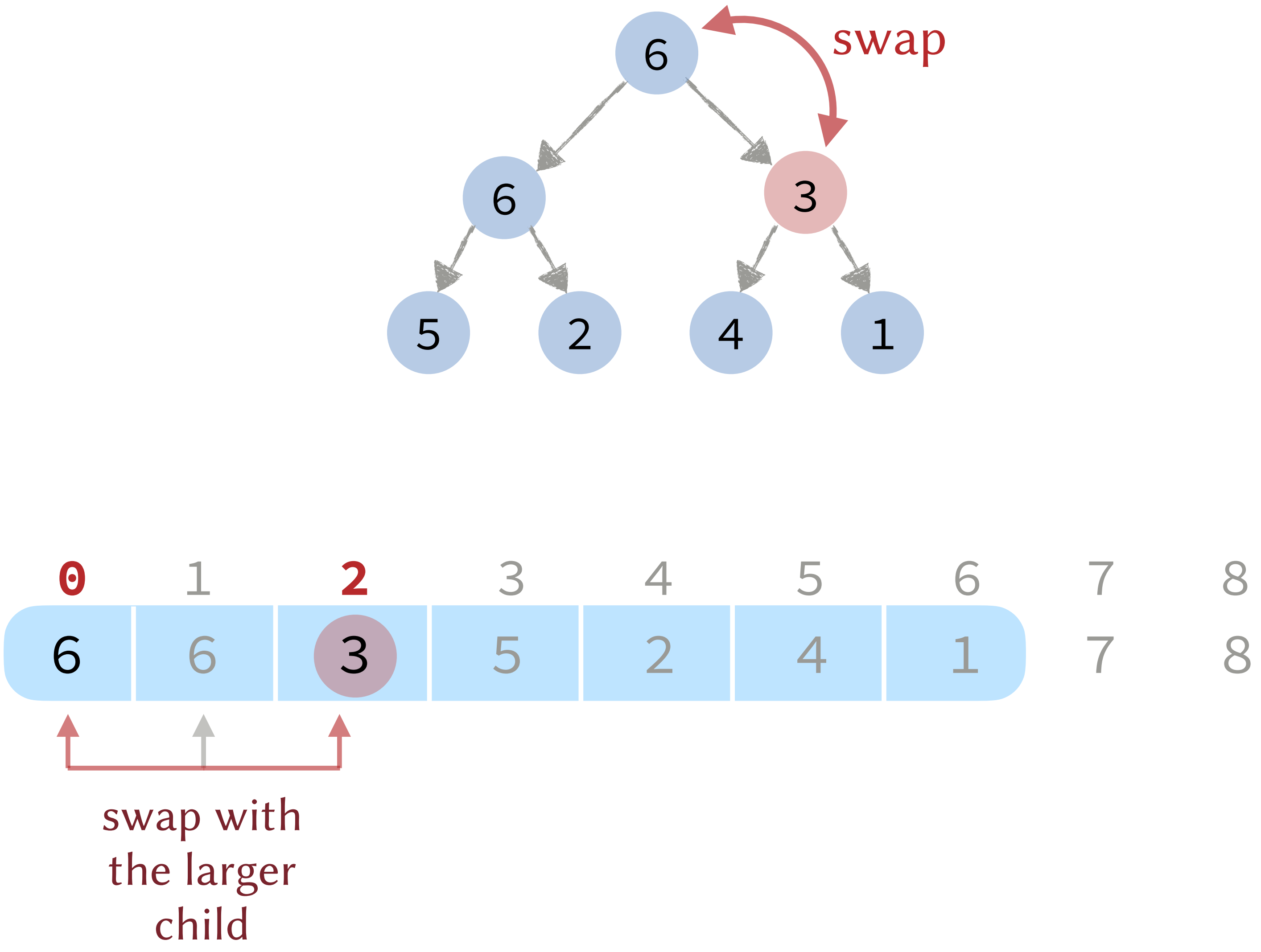


0	1	2	3	4	5	6	7	8
3	6	6	5	2	4	1	7	8

Binary Heaps: Deletion

Basic Plan.

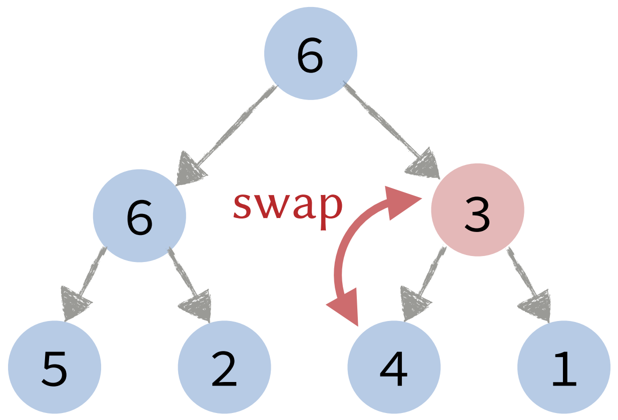
1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.

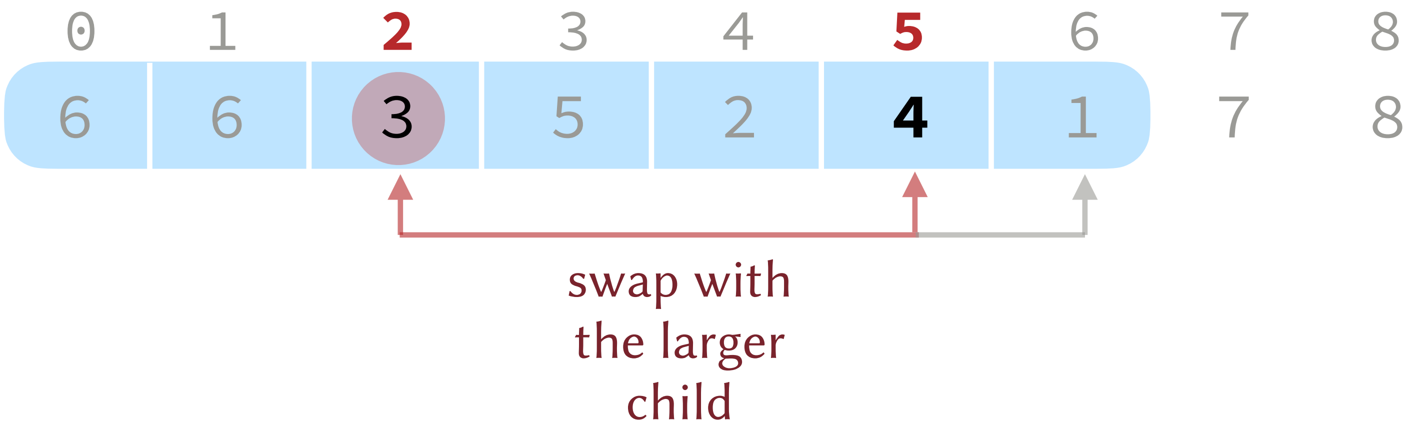


Binary Heaps: Deletion

Basic Plan.

1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.





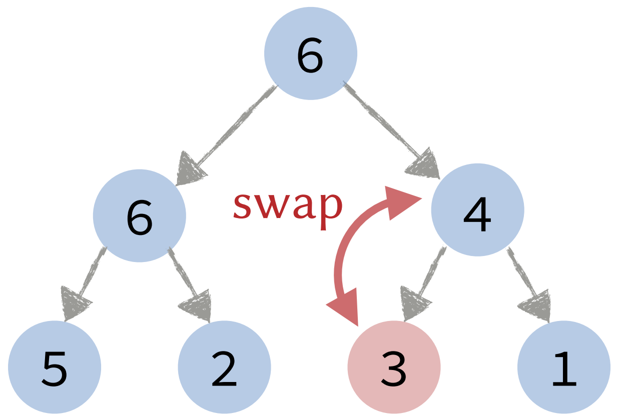
0	1	2	3	4	5	6	7	8
6	6	3	5	2	4	1	7	8

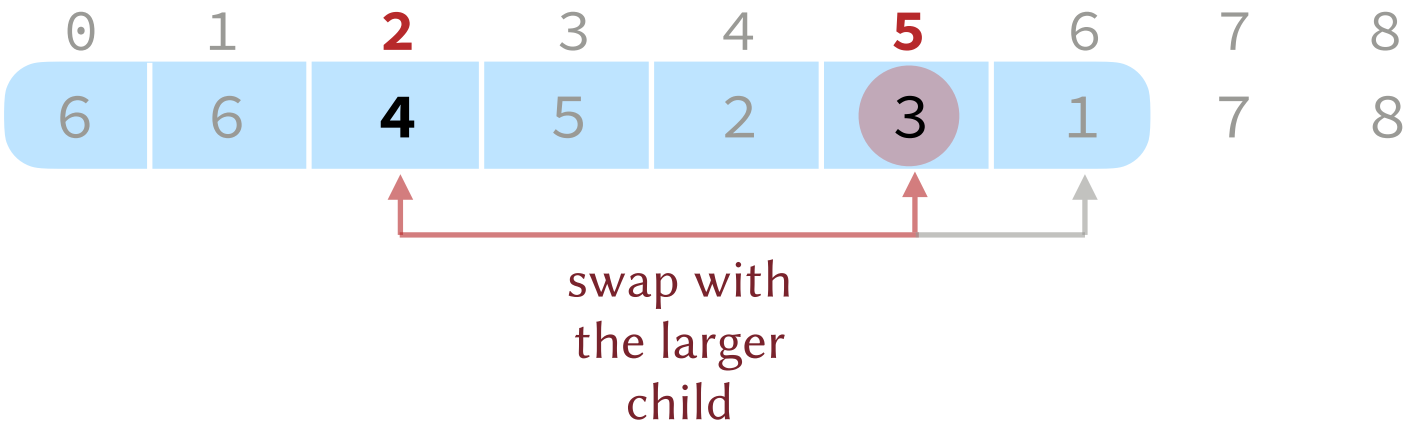
swap with the larger child

Binary Heaps: Deletion

Basic Plan.

1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.

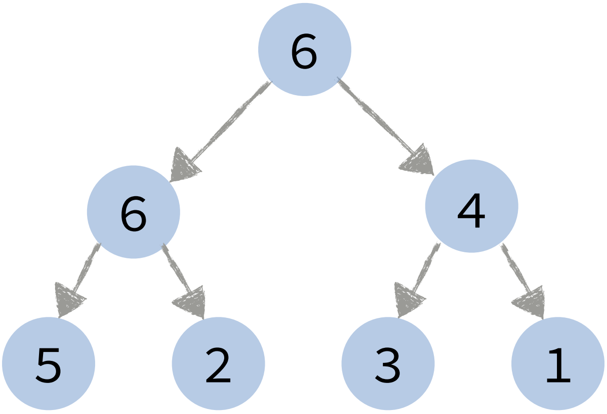




Binary Heaps: Deletion

Basic Plan.

1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.



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

Running Time.

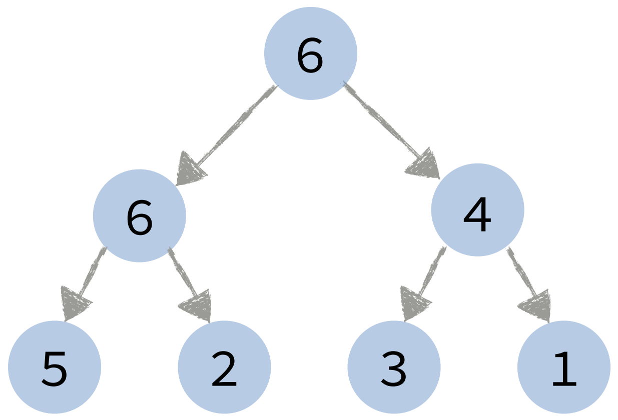
Best Case: 1 swap and 2 data compares.

Worst Case: $1 + \lfloor \log_2 n \rfloor$ swaps and $2\lfloor \log_2 n \rfloor$ data compares.

Binary Heaps: Deletion

Basic Plan.

1. Swap the first and last elements in the heap.
2. Delete the last element.
3. Fix the heap.
   swap down until the heap is fixed.



```
int remove_max(int a[],
               int& size) {

  swap(a[size-1], a[0]);
  size--;
  sink(a, 0, size);

  return a[size];
}
```

```
void sink(int a[], int i, int size) {

    while (LEFT(i) < size) {
        int 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;
    }
}
```

optional

A *Better* Selection Sort ??

```
void selection_sort(T a[], int n)
```

A *Better* Selection Sort ??

```
void use_heap_to_sort(T a[], int n)

MaxHeap<T> heap;

for (int i = 0; i < n; i++)
    heap.insert(a[i]);

for (int i = n-1; i >= 0; i--) {
    a[i] = heap.get_max();
    heap.remove_max();
}
```

A *Better* Selection Sort ??

```
void use_heap_to_sort(T a[], int n)

MaxHeap<T> heap;

for (int i = 0; i < n; i++)
    heap.insert(a[i]);

for (int i = n-1; i >= 0; i--) {
    a[i] = heap.get_max();
    heap.remove_max();
}
```

1 insert all the array elements
into a max-heap

A *Better* Selection Sort ??

```
void use_heap_to_sort(T a[], int n)

MaxHeap<T> heap;

for (int i = 0; i < n; i++)
    heap.insert(a[i]);

for (int i = n-1; i >= 0; i--) {
    a[i] = heap.get_max();
    heap.remove_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)

A *Better* **Selection Sort** ??

```
void use_heap_to_sort(T a[], int n)

MaxHeap<T> heap;

for (int i = 0; i < n; i++)
   heap.insert(a[i]);

for (int i = n-1; i >= 0; i--) {
   a[i] = heap.get_max();
   heap.remove_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. $\log_2(1) + \log_2(2) + \log_2(3) + \ \ldots \ + \log_2(n-1) \le \log_2(n!)$

insert the second element into the a heap of size 1

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

A *Better* **Selection Sort** ??

```
void use_heap_to_sort(T a[], int n)

MaxHeap<T> heap;

for (int i = 0; i < n; i++)
    heap.insert(a[i]);

for (int i = n-1; i >= 0; i--) {
    a[i] = heap.get_max();
    heap.remove_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. $2 \times (\log_2(n-1) + \log_2(n-2) + \log_2(n-3) + \ \ldots \ + \log_2(1))$

  $\le 2 \times \log_2(n!)$

A *Better* Selection Sort ??

```
void use_heap_to_sort(T a[], int n)

MaxHeap<T> heap;

for (int i = 0; i < n; i++)
   heap.insert(a[i]);

for (int i = n-1; i >= 0; i--) {
   a[i] = heap.get_max();
   heap.remove_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. $2 \times (\log_2(n-1) + \log_2(n-2) + \log_2(n-3) + \ \ldots \ + \log_2(1))$

  $\leq 2 \times \log_2(n!) = O(n \log n)$

swapping down the heap requires
2 compares to identify the larger child!

A *Better* Selection Sort ??

```
void use_heap_to_sort(T a[], int n)

MaxHeap<T> heap;

for (int i = 0; i < n; i++)
   heap.insert(a[i]);

for (int i = n-1; i >= 0; i--) {
   a[i] = heap.get_max();
   heap.remove_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. $2 \times (\log_2(n-1) + \log_2(n-2) + \log_2(n-3) + \ \ldots \ + \log_2(1))$
  $\le 2 \times \log_2(n!) = O(n \log n)$
- Total. $O(n \log n)$

https://stock.adobe.com/images/Computer-icon-flat/107615093?as_campaign=TinEye&as_content=tineye_match&epi1=107615093&tduid=e2fbc5f655eb4ac700d0f7eba0abd3f2&as_channel=affiliate&as_campclass=redirect&as_source=arvato

https://usercontent1.hubstatic.com/12803386_f520.jpg

https://www.vhv.rs/dpng/d/481-4810877_printing-press-clip-art-printing-clip-art-hd.png

https://www.shutterstock.com/image-vector/linear-flat-clinic-interior-furniture-doctor-484985722?irclickid=0nN0yYztWxyIWrPQHcWEzT3UUkGQVLTcPTqHxo0&irgwc=1&utm_medium=Affiliate&utm_campaign=TinEye&utm_source=77643&utm_term=&c3ch=Affiliate&c3nid=IR-77643