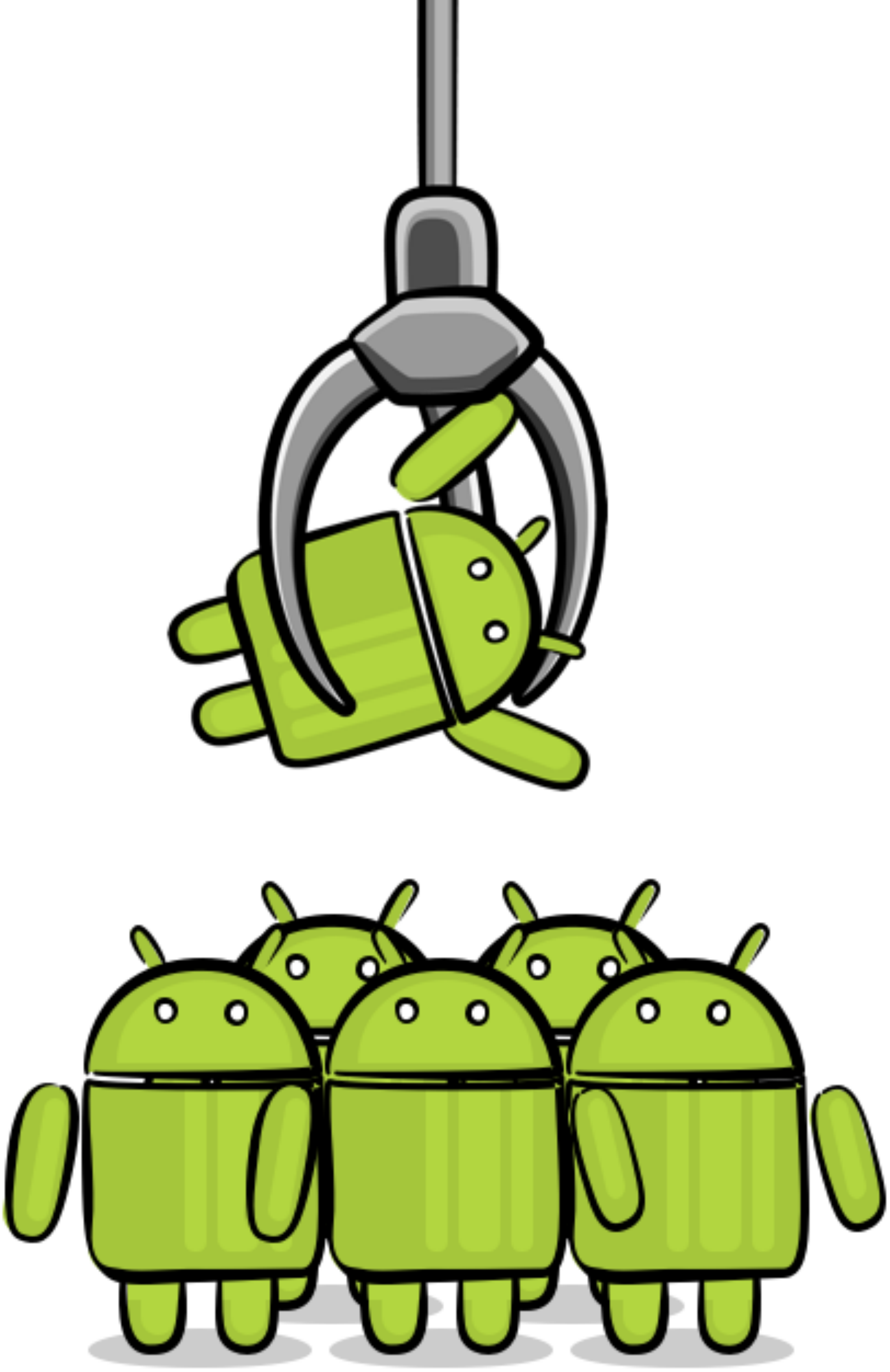
12 The Heap Data Structure Written by Irina Galata

Have you ever been to the arcade and played those crane machines that contain stuffed animals or cool prizes? These machines make it extremely difficult to win. But the fact that you set your eyes on the item you want is the very essence of the heap data structure!

Have you seen the movie *Toy Story* with the claw and the little green squeaky aliens? Just imagine that the claw machine operates on your heap data structure and will always pick the element with the highest priority.



In this chapter, you'll focus on creating a heap, and you'll see how convenient it is to fetch the minimum and maximum element of a collection.

What is a heap?

A heap is a complete binary tree data structure also known as a **binary heap** that you can construct using an array.

> **Note:** Don't confuse these heaps with memory heaps. The term heap is sometimes confusingly used in computer science to refer to a pool

> of memory. Memory heaps are a different concept and are not what you're studying here.

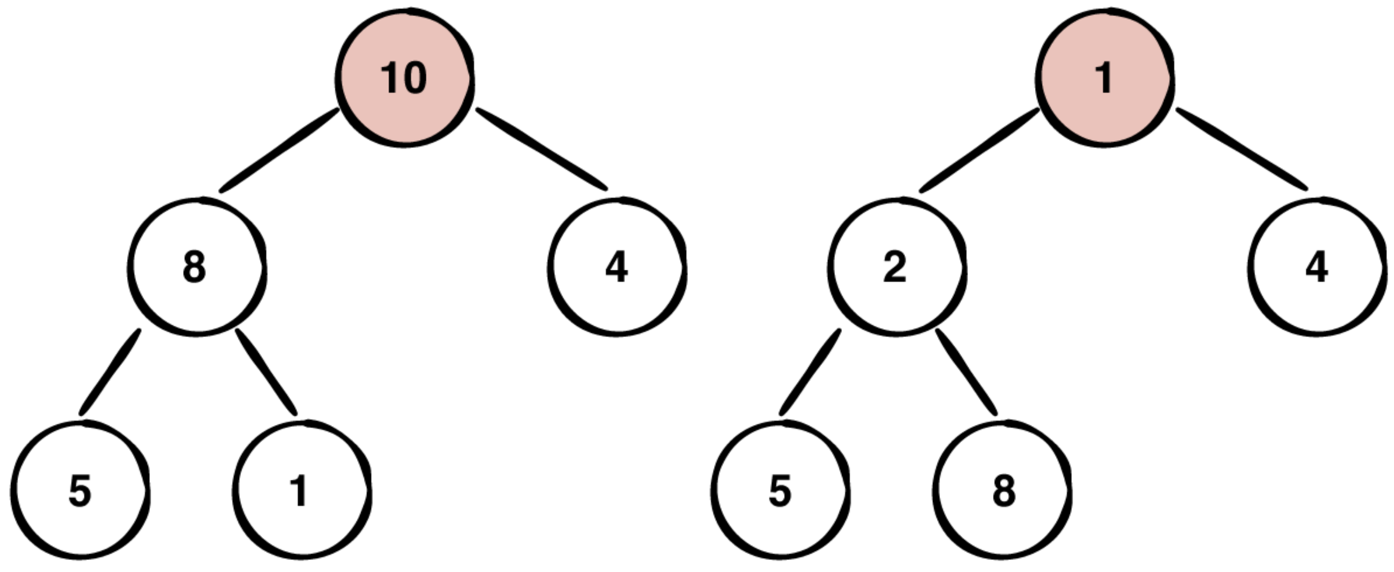
Heaps come in two flavors:

1. **Max**heap, in which elements with a **higher** value have a higher priority.
2. **Min**heap, in which elements with a **lower** value have a higher priority.

> **Note**: It's important to say that the concept of heap is valid for every type of object that can be compared to others of the same type. In this chapter you'll see mostly **Int**s but the same concepts are true for all **Comparable** types or, as you'll see later, if a **Comparator** is provided .

A heap has an important characteristic that must always be satisfied. This is known as the **heap invariant** or **heap property**.

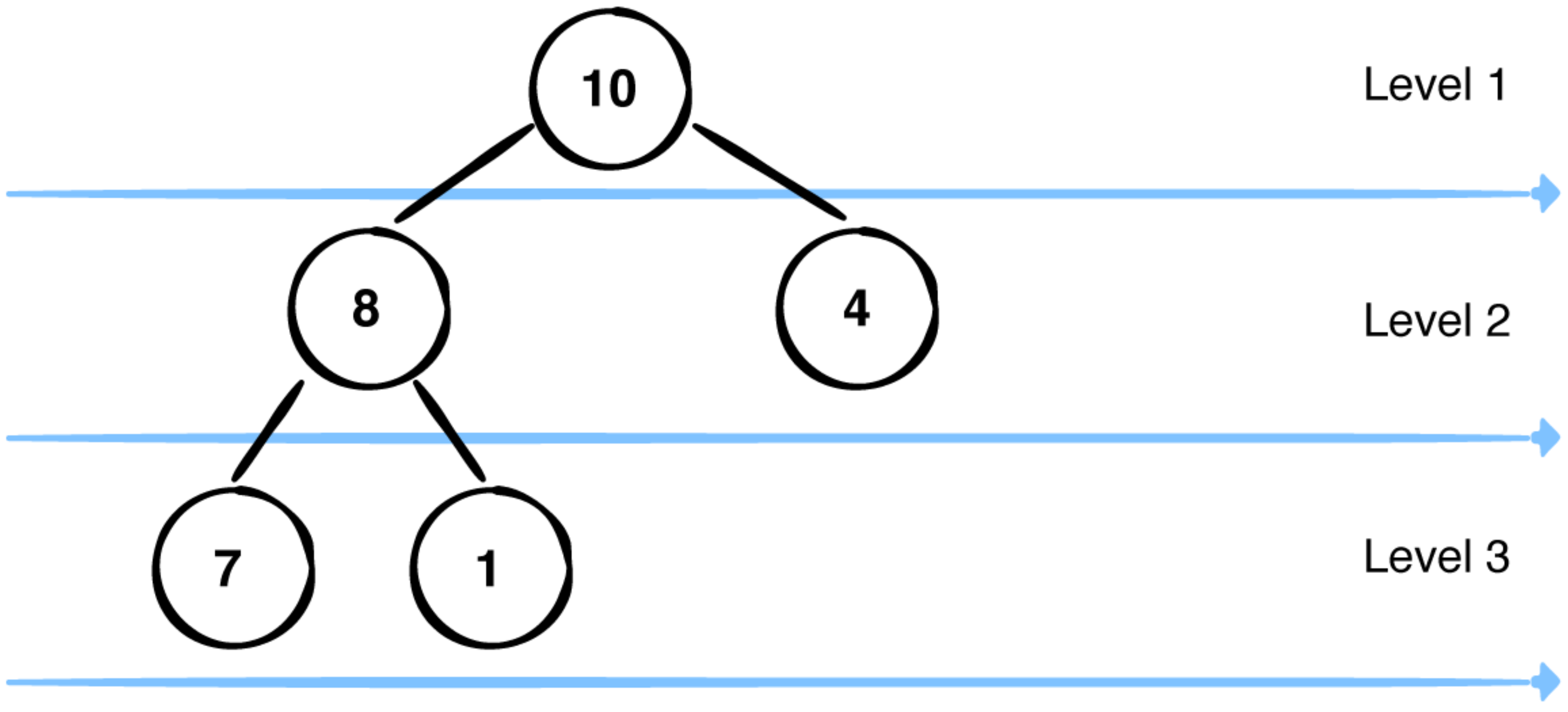
The heap property



Max Heap Min Heap

In a **maxheap**, parent nodes must always contain a value that is *greater than or equal to* the value in its children. The root node will always contain the highest value.

In a **minheap**, parent nodes must always contain a value that is *less than or equal to* the value in its children. The root node will always contain the lowest value.



Another important property of a heap is that it's a **complete** binary tree. This means that every level must be filled, except for the last level. It's like a video game wherein you can't go to the next level until you have completed the current one.

Heap applications

Some useful applications of a heap include:

- Calculating the minimum or maximum element of a collection.
- Heap sort.
- Implementing a priority queue.
- Supporting graph algorithms, like Prim's or Dijkstra's, with a priority queue.

> **Note**: You'll learn about each of these concepts in later chapters.

Common heap operations

Open the empty starter project for this chapter. Start by defining the following basic `Collection` type:

```
interface Collection<T: Any> {
  val count: Int

  val isEmpty: Boolean
    get() = count == 0

  fun insert(element: T)

  fun remove(): T?

  fun remove(index: Int): T?
}
```

Here you have a generic `Collection` interface with the basic property `count` which returns the number of elements and the boolean property `isEmpty` which just tests if the `count` is 0. It also contains the classical operations of inserting and deletion.
Given that you can define the `Heap` interface like this.

```
interface Heap<T: Any> : Collection<T> {

  fun peek(): T?
}
```

The `peek` operation is a generalization of methods returning the min or the max depending on the implementation. Because of this you can usually find the same operation with name `extract-min` or `extract-max`.

Sorting and comparing

The heap properties imply there must be a way to **compare** each element and so a way to test if an element `A` is greater, smaller or equals than the element `B`. In Kotlin, as well as in Java, this can be achieved in 2 different ways:

- `T` implements the `Comparable<T>` interface
- You can provide a `Comparator<T>` implementation

Implementing the `Comparable<T>` interface, a type `T` can only provide a single way of comparing instances of itself with others of the same type. If you use a `Comparator<T>` you can choose different way of sorting simply using different `Comparator` implementations. In both cases you need to abstract the way you compare different instances. Because of this you can define the abstract class which contains the definition of the `compare` method you'll implement in different ways depending on the 2 different approaches. This method returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

```
abstract class AbstractHeap<T: Any>() : Heap<T> {

  abstract fun compare(a: T, b: T): Int
}
```

In case of `Comparable` types you can define a `Heap` implementation like the following where the `T` type implements `Comparable<T>` and `compare` method invokes the related `compareTo` method.

```
class ComparableHeapImpl<T : Comparable<T>>() : AbstractHeap<T>() {

  override fun compare(a: T, b: T): Int = a.compareTo(b)
}
```

In case you want to use a `Comparator<T>` you can implement a `Heap` like this where the `compare` method delegates to the `Comparator<T>` you pass as parameter.

```
class ComparatorHeapImpl<T: Any>(
    private val comparator: Comparator<T>
) : AbstractHeap<T>() {

  override fun compare(a: T, b: T): Int =
    comparator.compare(a, b)
}
```

In the previous code, you'll see some errors because of the missing implementation of the `peek` operation along with the operations from the `Collection` interface, but you'll fix everything very soon. Anyway, depending on the `Comparator<T>` or the `Comparable<T>` implementation you can create different minheaps and maxheaps. Before this you need some theory.

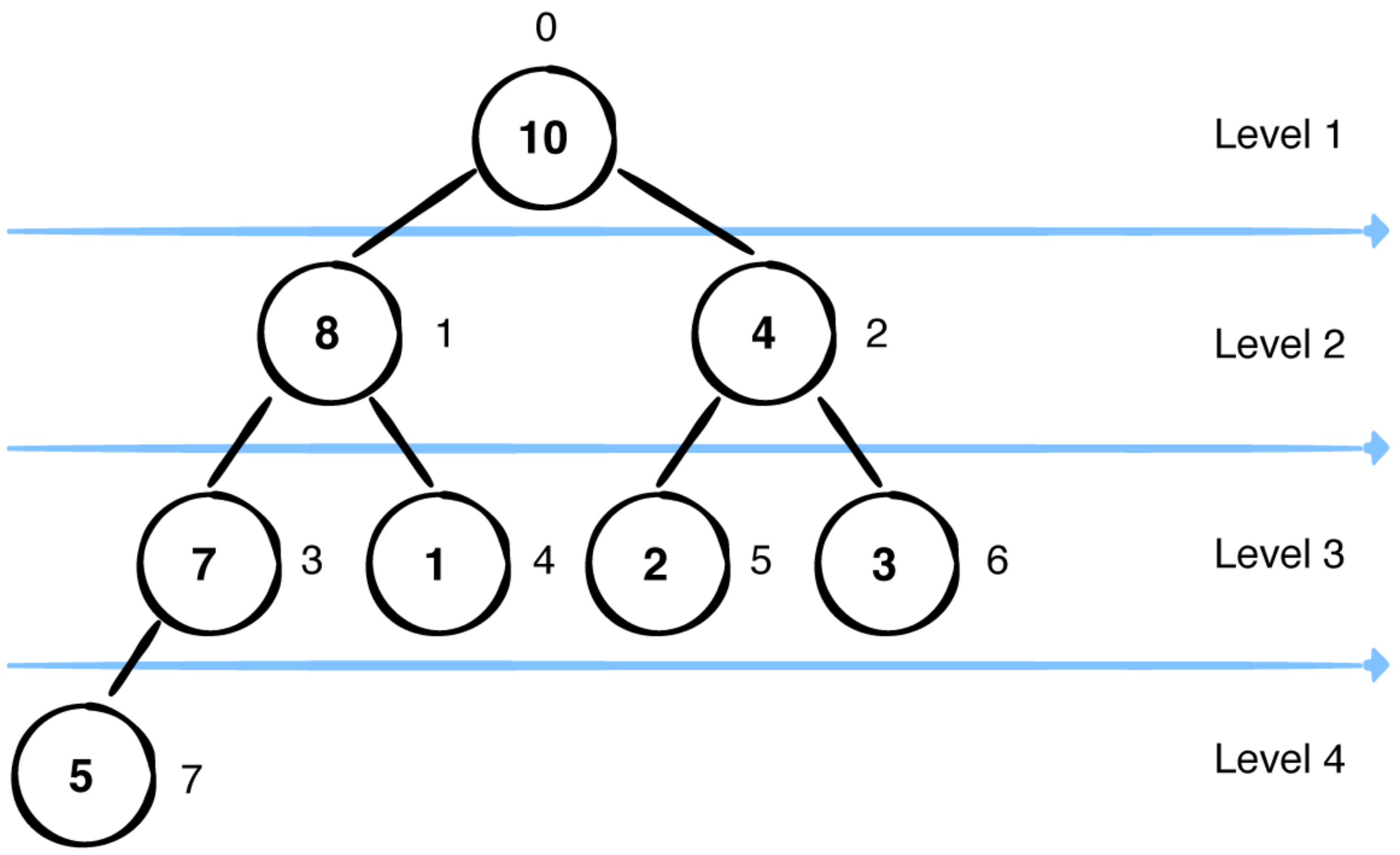
How do you represent a heap?

Trees hold nodes that store references to their children. In the case of a binary tree, these are references to a left and a right child.

Heaps are indeed binary trees, but you can represent them with a simple array. This seems like an unusual way to build a tree, but one of the benefits of this heap implementation is **efficient time** and **space complexity**, as the elements in the heap are all stored together in memory.

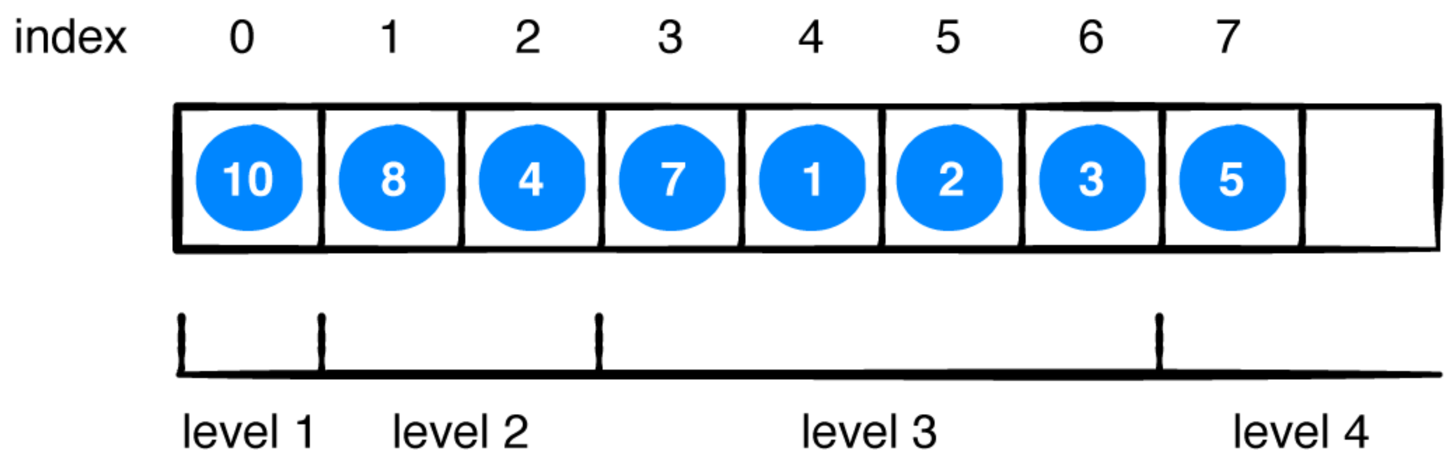
You'll see later on that **swapping** elements plays a big part in heap operations. This is also easier to do with an array than with a binary tree data structure.

It's time to look at how you can represent heaps using an array. Take the following binary heap:



To represent the heap above as an array, you would simply iterate through each element level-by-level from left to right.

Your traversal would look something like this:

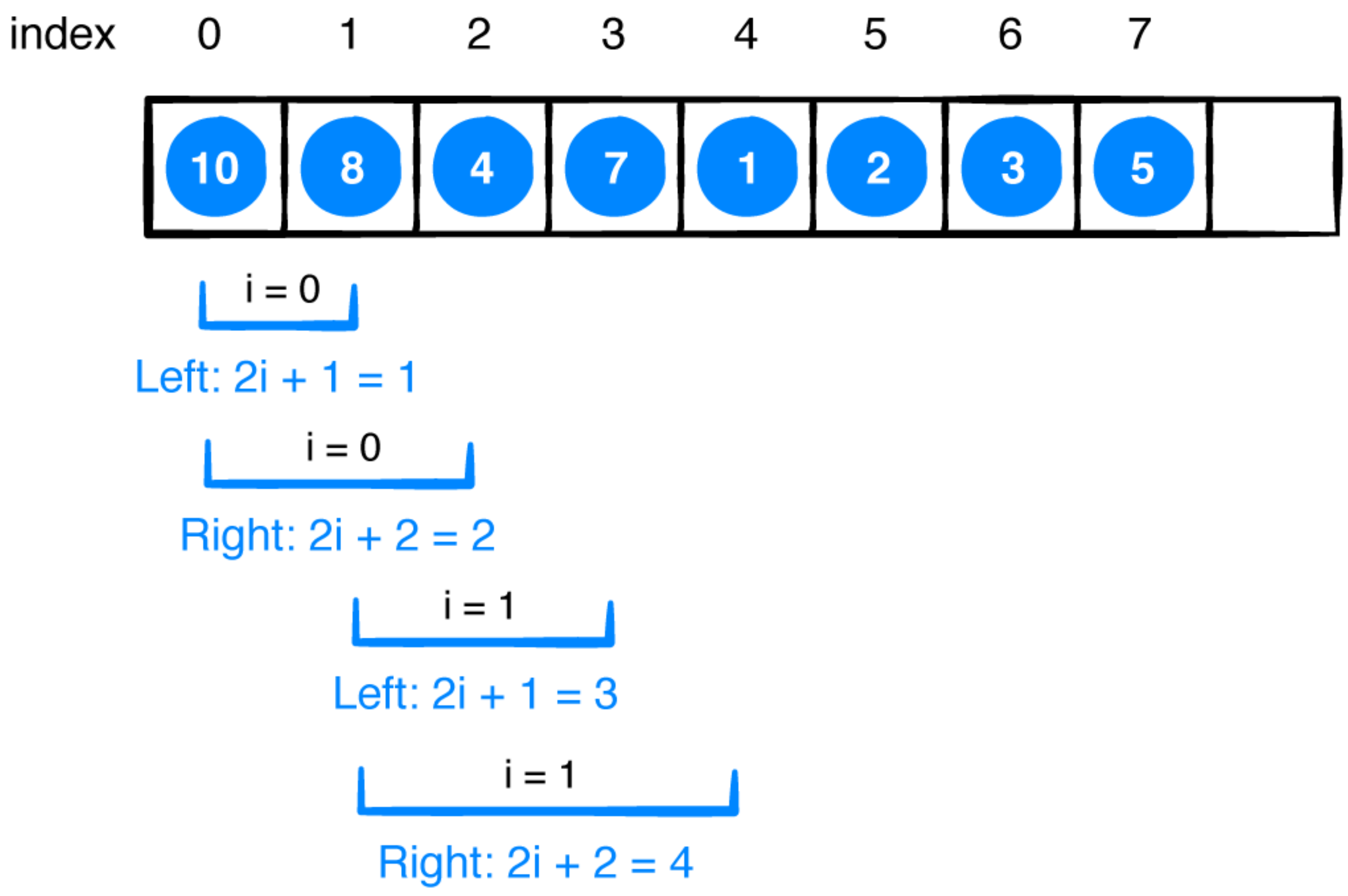


As you go up a level, you'll have twice as many nodes than in the level before.

It's now easy to access any node in the heap. You can compare this to how you'd access elements in an array: Instead of traversing down the left or right branch, you can simply access the node in your array using simple formulas.

Given a node at a zero-based index `i`:

- You can find the **left child** of this node at index `2i + 1`.
- You can find the **right child** of this node at index `2i + 2`.



You might want to obtain the parent of a node. You can solve for `i` in this case. Given a child node at index `i`, you can find this child's parent node at index `(i - 1) / 2`. Just remember this is an operation between `Int`s which returns an `Int`;in other languages you can call it the `floor` operation.

> **Note**: Traversing down an actual binary tree to get the left and right child of a node is an *O*(*log n*) operation. In a random-access data structure, such as an array, that same operation is just *O*(1).

Next, use your new knowledge to add some properties and convenience methods to the `AbstractHeap` class:

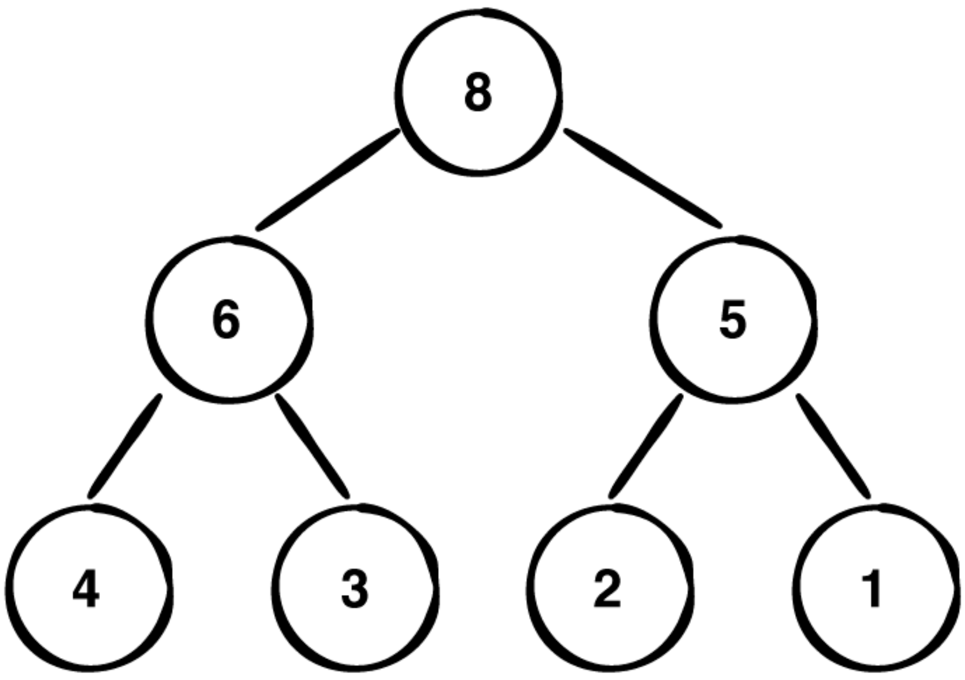
```
var elements: ArrayList<T> = ArrayList<T>()
```

```
override val count: Int
    get() = elements.size

override fun peek(): T? = elements.firstOrNull()

private fun leftChildIndex(index: Int) = (2 * index) + 1

private fun rightChildIndex(index: Int) = (2 * index) + 2

private fun parentIndex(index: Int) = (index - 1) / 2
```

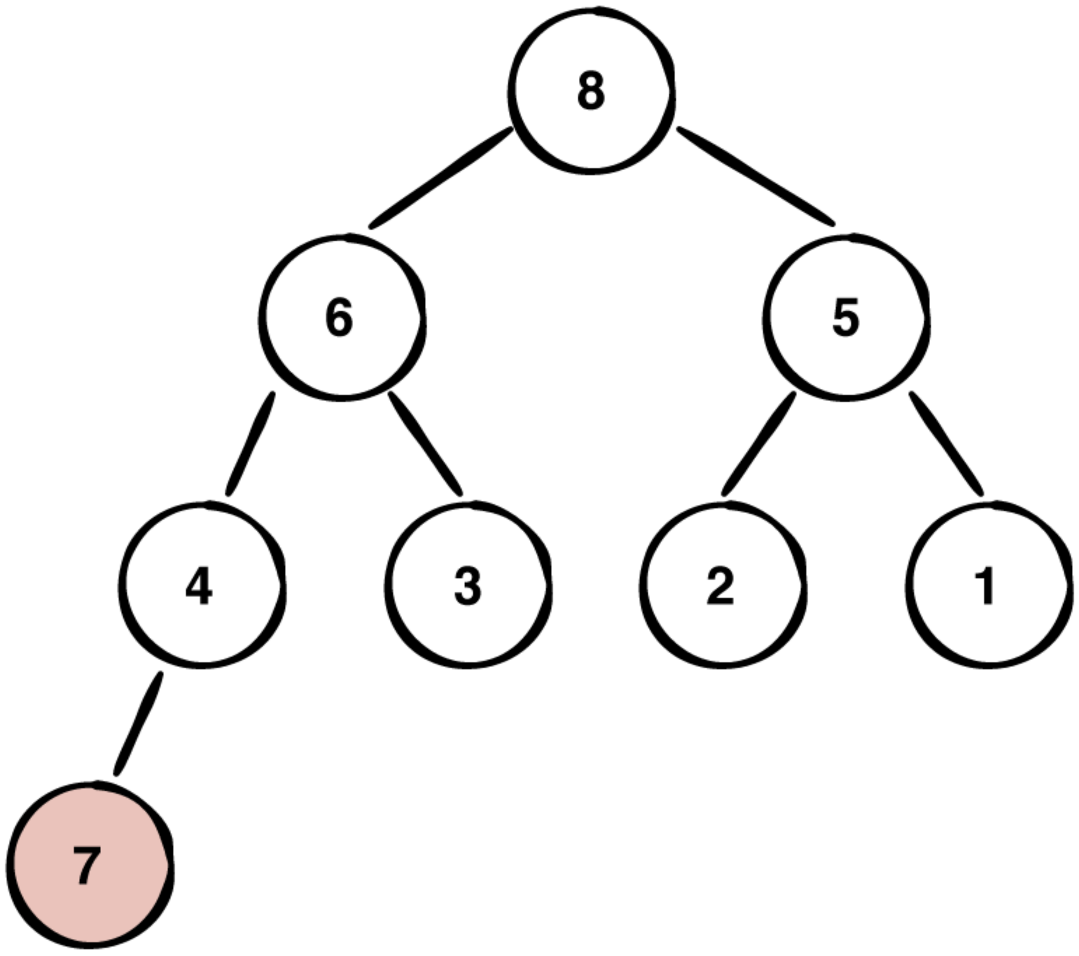
Now that you have a better understanding of how you can represent a heap using an array, you'll look at some important operations of a heap.

Inserting into a heap

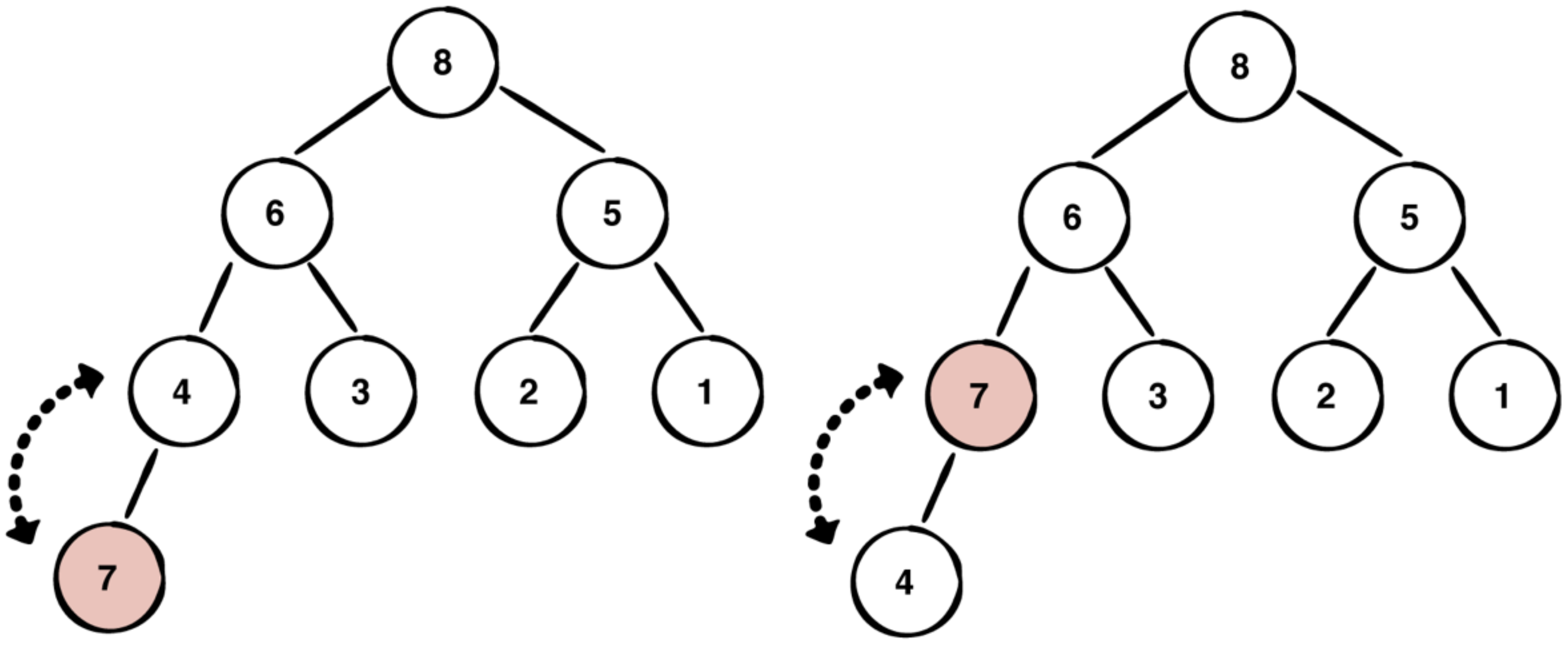
Suppose you insert a value of **7** to the heap below:

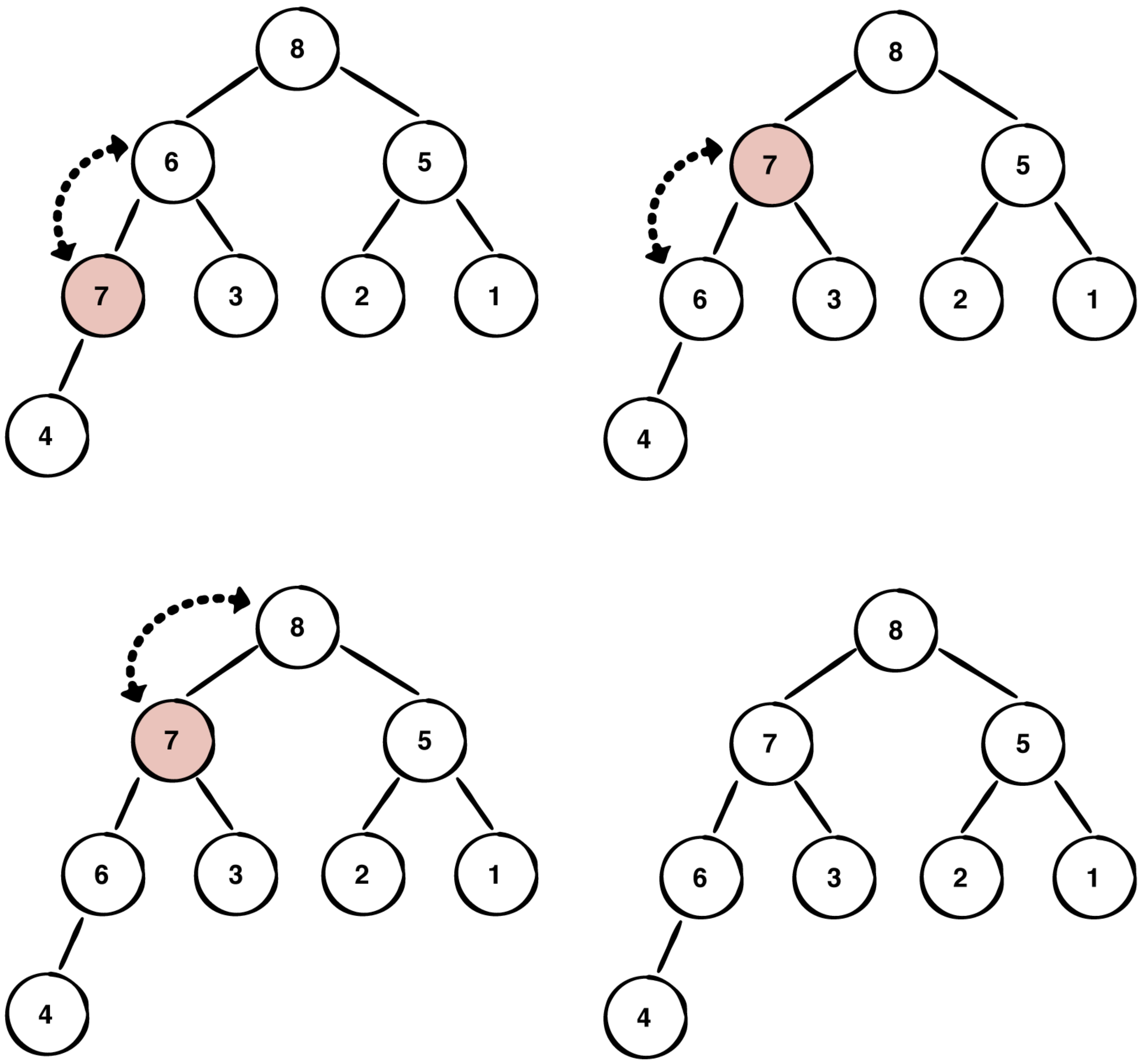


First, you add the value to the end of the heap:



Next, you must check the max heap's property. In order to do this you have to **sift up** since the node that you just inserted might have a higher priority than its parents. It does so by comparing the current node with its parent and swapping them if needed.





Your heap has now satisfied the max heap property.

Implementation of insert

Add the following code to `AbstractHeap`:

```
override fun insert(element: T) {
  elements.add(element) // 1
  siftUp(count - 1) // 2
}

private fun siftUp(index: Int) {
  var child = index
  var parent = parentIndex(child)
  while (child > 0 && compare(elements[child], elements[parent]) > 0) {
    Collections.swap(elements, child, parent)
    child = parent
    parent = parentIndex(child)
  }
}
```

As you can see, the implementation is rather straightforward:

1. `insert` appends the element to the array and then performs a sift up.
2. `siftUp` swaps the current node with its parent, as long as that node has a higher priority than its parent.

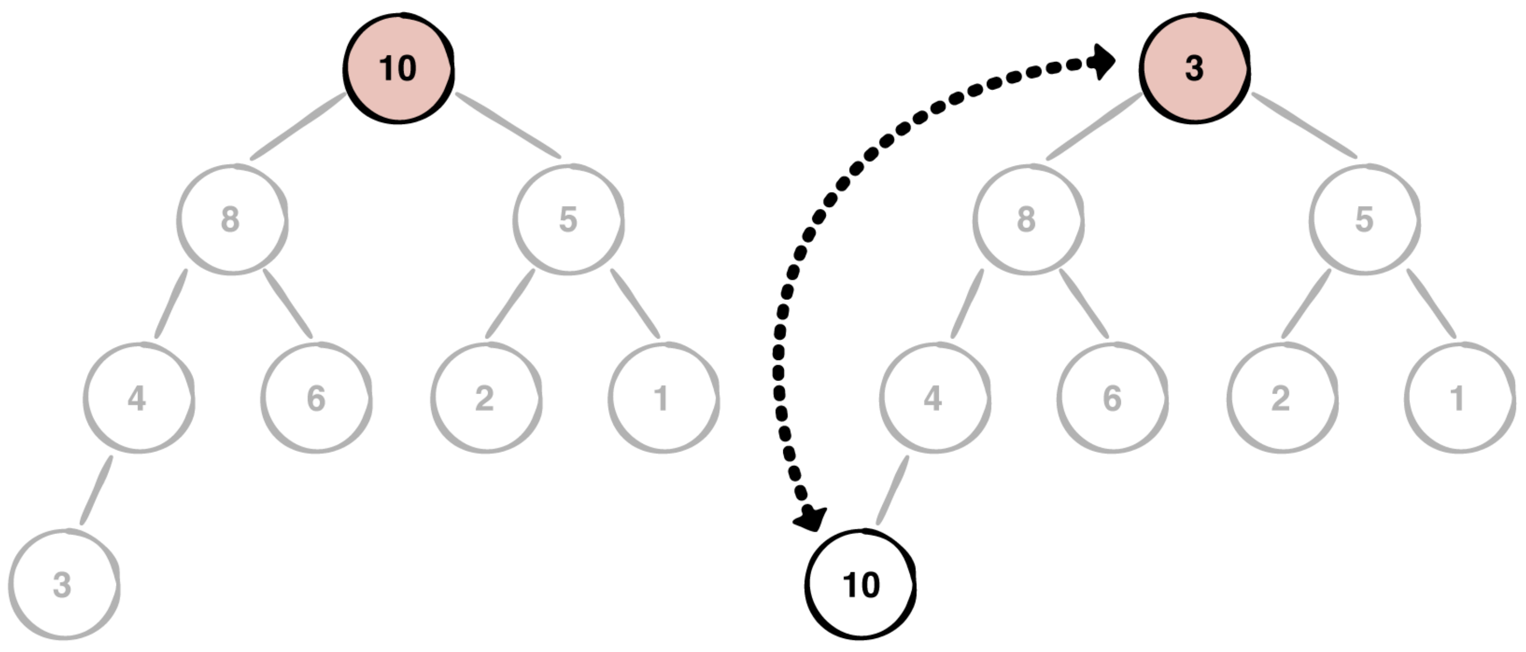
> **Complexity**: The overall complexity of `insert()` is *O*(*log n*). Appending an element in an array takes only *O*(1), while sifting up elements in a heap takes *O*(*log n*).

That's all there is to inserting an element in a heap but how can you remove an element?

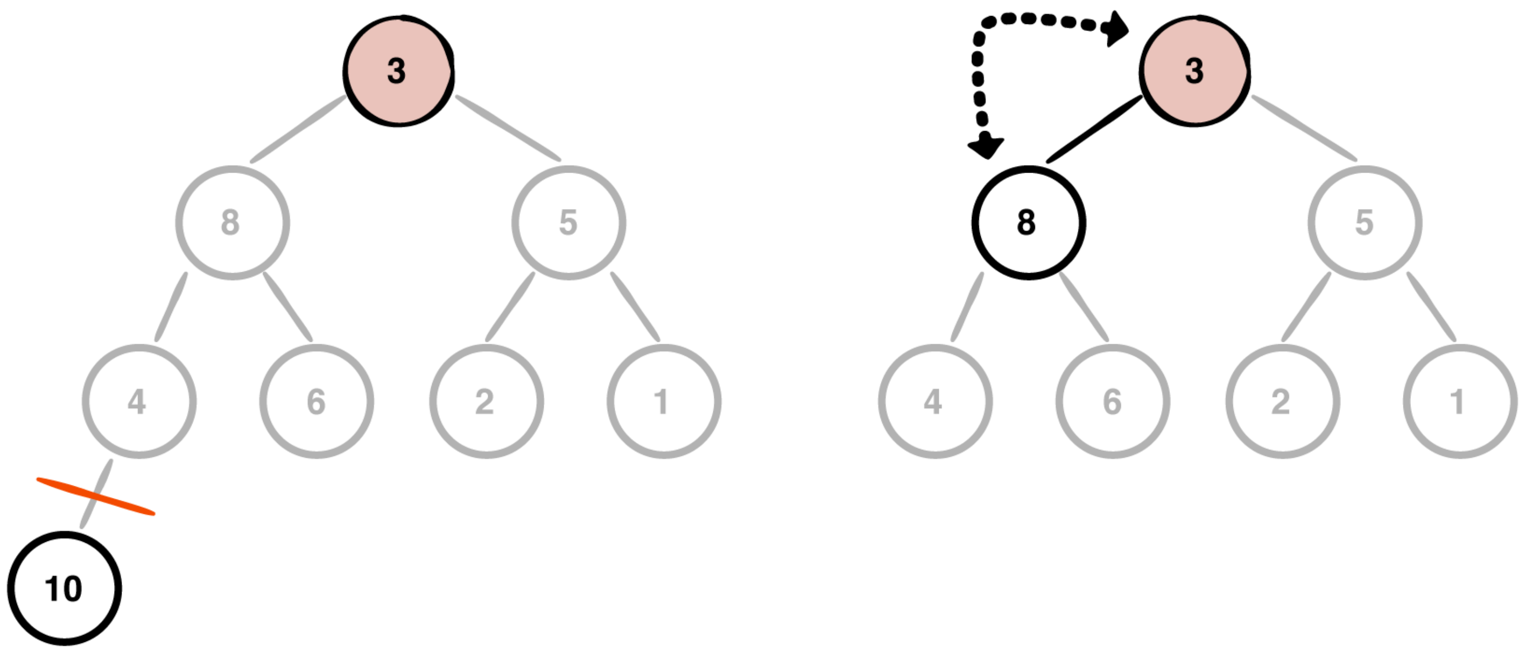
Removing from a heap

A basic remove operation removes the root node from the heap.

Take the following max heap:



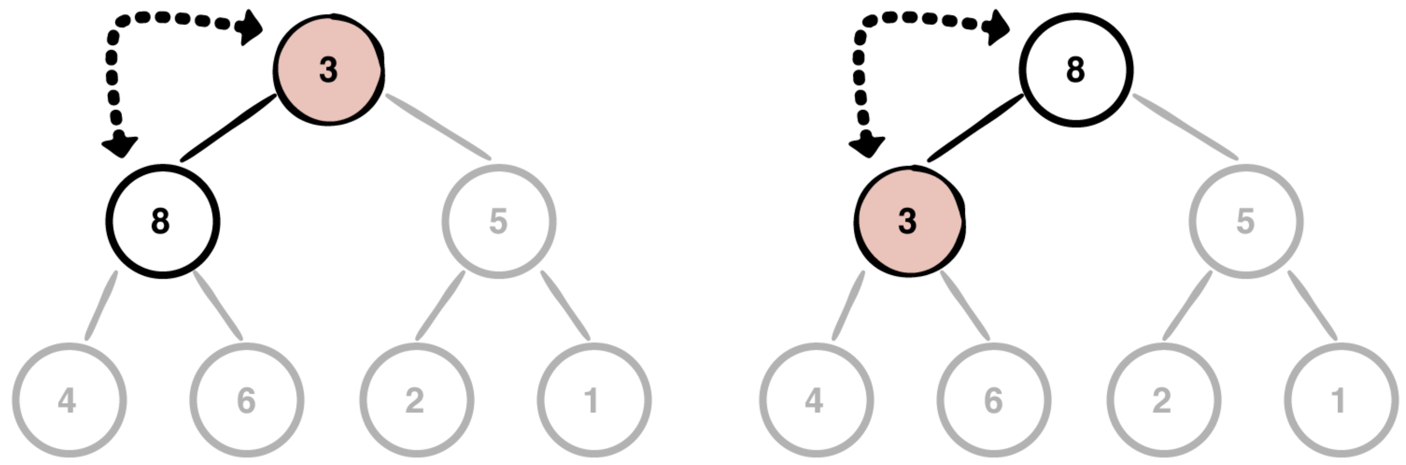
A remove operation will remove the maximum value at the root node. To do so, you must first swap the **root** node with the **last** element in the heap.



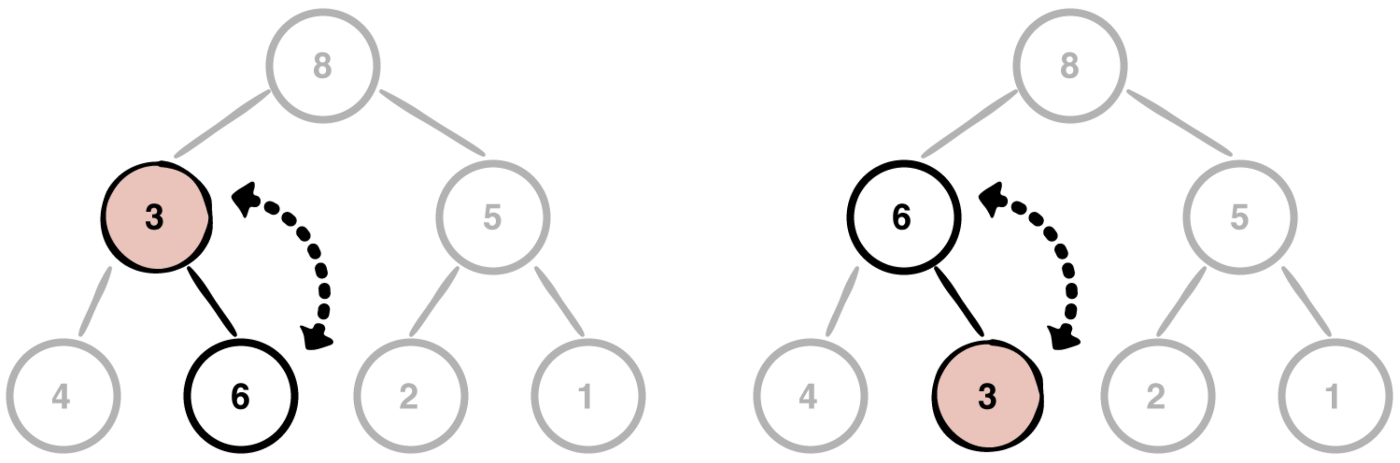
Once you've swapped the two elements, you can remove the last element and store its value so you can return it later.

Now, you must check the max heap's integrity. But first, ask yourself, *"Is it still a max heap?"*

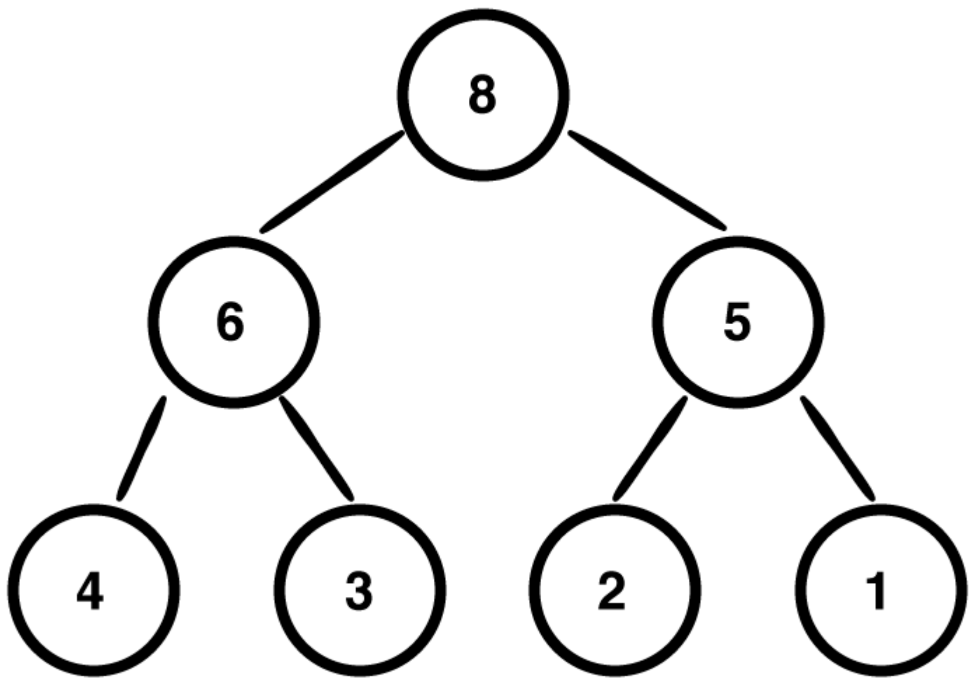
Remember: The rule for a max heap is that the value of every parent node must be larger than or equal to the values of its children. Since the heap no longer follows this rule, you must perform a **sift down**.



To perform a sift down, start from the current value **3** and check its left and right child. If one of the children has a value that is *greater* than the current value, you swap it with the parent. If both children have greater values, you swap the parent with the greater child value.



You have to continue to sift down until the node's value is not larger than the values of its children.



Once you reach the end, you're done, and the max heap's property has been restored.

Implementation of remove

Add the following code to `AbstractHeap`:

```
override fun remove(): T? {
  if (isEmpty) return null // 1

  Collections.swap(elements, 0, count - 1) // 2
  val item = elements.removeAt(count - 1) // 3
  siftDown(0) // 4
  return item
}
```

Here’s how this method works:

1. Check to see if the heap is empty. If it is, return `null`.
2. Swap the root with the last element in the heap.
3. Remove the last element (the maximum or minimum value) and return it.
4. The heap may not be a maxheap or minheap anymore, so you must perform a sift down to make sure it conforms to the rules.

To see how to sift down nodes, add the following method after `remove()`:

```
private fun siftDown(index: Int) {
  var parent = index // 1
  while (true) { // 2
    val left = leftChildIndex(parent) //3
    val right = rightChildIndex(parent)
    var candidate = parent // 4
    if (left < count &&
      compare(elements[left], elements[candidate]) > 0) {
      candidate = left //5
    }
    if (right < count &&
      compare(elements[right], elements[candidate]) > 0) {
      candidate = right // 6
    }
    if (candidate == parent) {
      return // 7
    }
    Collections.swap(elements, parent, candidate) // 8
    parent = candidate
  }
}
```

`siftDown()` accepts, as parameter, the index of the element to consider as the parent node to swap with one of the children until it finds the right position. Here's how the method works:

1. Store the `parent` index.
2. Continue sifting until you `return`.
3. Get the parent's left and right child index.
4. `candidate` is used to keep track of which index to swap with the parent.
5. If there's a left child, and it has a higher priority than its parent, make it the candidate.
6. If there's a right child, and it has an even greater priority, it will become the candidate instead.
7. If `candidate` is still `parent`, you have reached the end, and no more sifting is required.
8. Swap `candidate` with `parent` and set it as the new parent to continue sifting.

> **Complexity**: The overall complexity of `remove()` is *O*(*log n*). Swapping elements in an array takes only *O*(1), while sifting down elements in a heap takes *O*(*log n*) time.

Now that you know how to remove from the top of the heap, it's time to learn how to **add** to a heap.

Removing from an arbitrary index

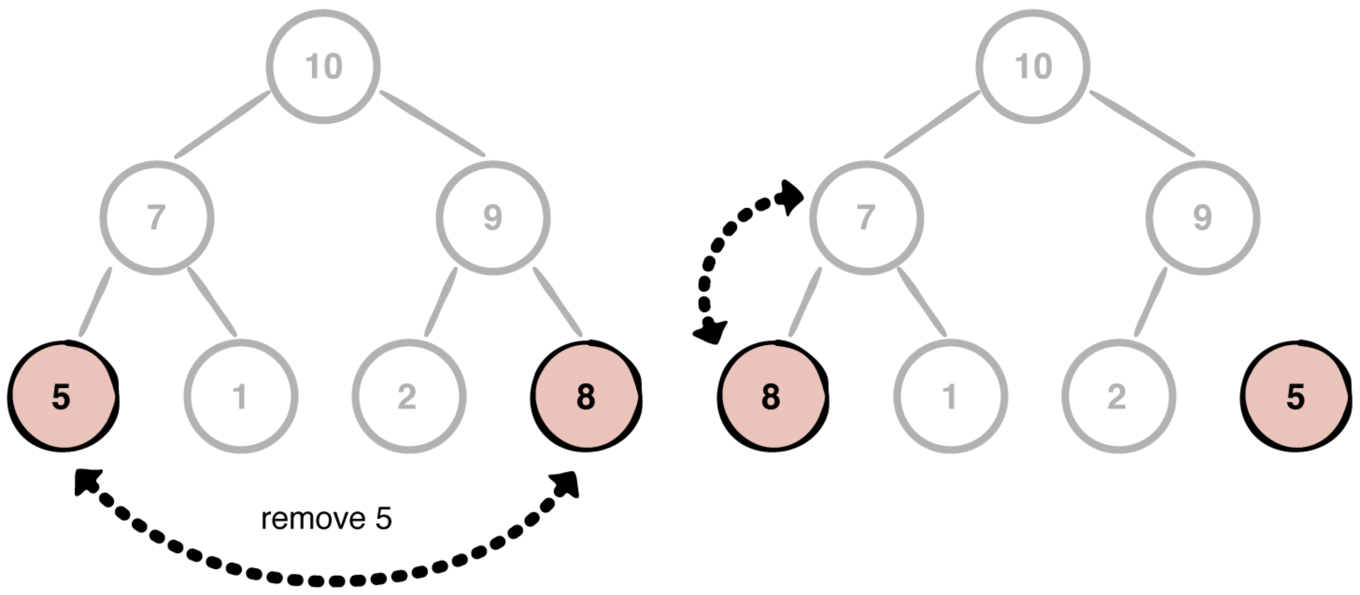
Add the following method to `AbstractHeap` removing all the previous errors:

```
override fun remove(index: Int): T? {
  if (index >= count) return null // 1

  return if (index == count - 1) {
    elements.removeAt(count - 1) // 2
  } else {
    Collections.swap(elements, index, count - 1) // 3
    val item = elements.removeAt(count - 1) // 4
    siftDown(index) // 5
    siftUp(index)
    item
  }
}
```

To remove any element from the heap, you need an index. Let's go over how this works:

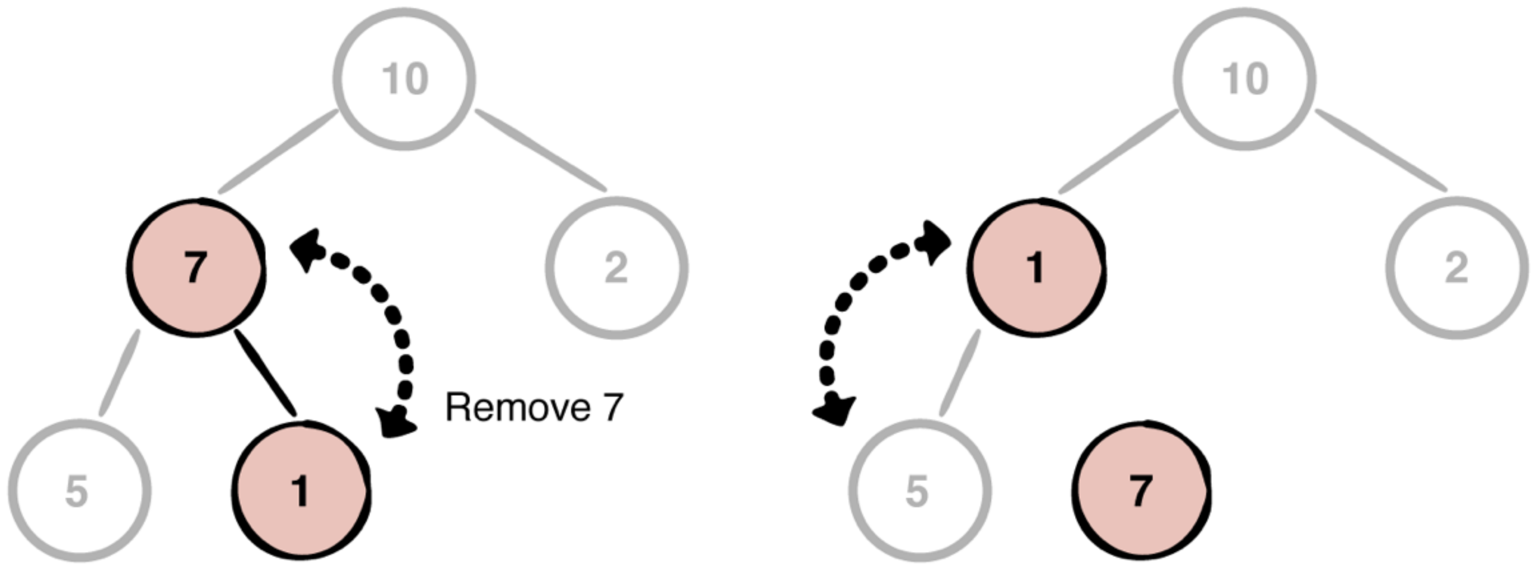
1. Check to see if the index is within the bounds of the array. If not, return `null`.
2. If you're removing the last element in the heap, you don't need to do anything special. Simply remove and return the element.
3. If you're not removing the last element, first swap the element with the last element.
4. Then, return and remove the last element.
5. Finally, perform both a sift down and a sift up to adjust the heap.

So, why do you have to perform a sift down *and* a sift up?



Shifting up case

Assume that you're trying to remove **5**. You swap 5 with the last element, which is **8**. You now need to perform a sift up to satisfy the max heap property.



Shifting down case

Now, assume that you're trying to remove **7**. You swap 7 with the last element, which is **1**. You now need to perform a sift down to satisfy the max heap property.

> **Complexity**: Removing an arbitrary element from a heap is an *O*(*log n*) operation.

But how do you find the index of the element you want to delete?

Searching for an element in a heap

To find the index of the element that you want to delete, you must perform a search on the heap. Unfortunately, heaps are not designed for fast searches.

With a binary search tree, you can perform a search in *O*(*log n*) time, but since heaps are built using an array, and the node ordering in an array is different, you can't even perform a binary search.

> **Complexity**: To search for an element in a heap is, in the worst-case, an *O*(*n*) operation, since you may have to check every element in the array:

```
private fun index(element: T, i: Int): Int? {
  if (i >= count) {
    return null // 1
  }
  if (compare(element, elements[i]) > 0) {
    return null // 2
  }
  if (element == elements[i]) {
    return i // 3
  }

  val leftChildIndex = index(element, leftChildIndex(i))
  if (leftChildIndex != null) return leftChildIndex // 4
  val rightChildIndex = index(element, rightChildIndex(i))
  if (rightChildIndex != null) return rightChildIndex // 5
  return null // 6
}
```

Here's how this implementation works:

1. If the index is greater than or equal to the number of elements in the array, the search failed. Return `null`.
2. Check to see if the element that you're looking for has higher priority than the current element at index `i`. If it does, the element you're looking for cannot possibly be lower in the heap.
3. If the element is equal to the element at index `i`, return `i`.
4. Recursively search for the element starting from the left child of `i`.
5. Recursively search for the element starting from the right child of `i`.
6. If both searches failed, the search failed. Return `null`.

> **Note**: Although searching takes *O*(*n*) time, you have made an effort to optimize searching by taking advantage of the heap's property and checking the priority of the element when searching.

Heapify an array

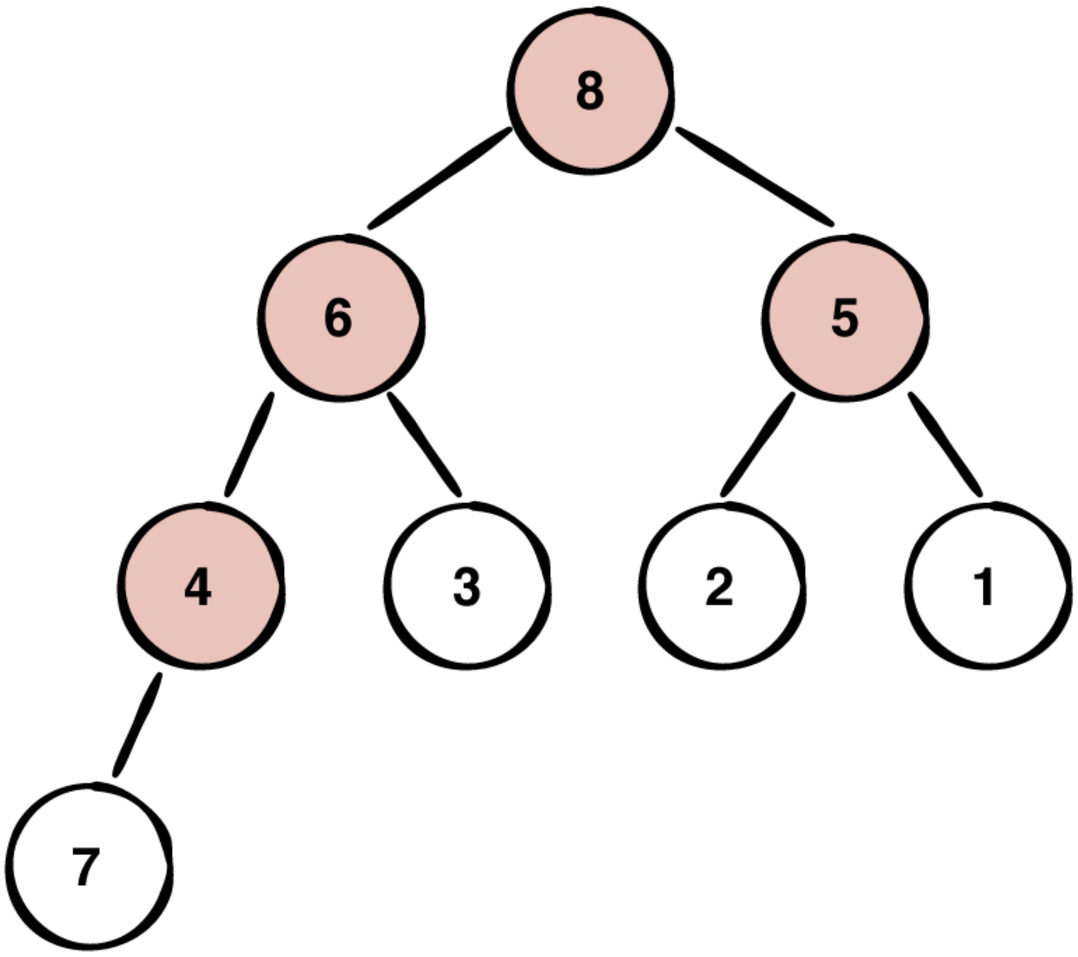
In the previous implementations of the `Heap` data structure, you have used an `ArrayList`. Other implementation could use an `Array` setting a max dimension for it. Making an existing array following the heap properties is an operation usually called **heapify**.

In order to implement this function add this code to `AbstractHeap`.

```
protected fun heapify(values: ArrayList<T>) {
  elements = values
  if (!elements.isEmpty()) {
    (count / 2 downTo 0).forEach {
      siftDown(it)
    }
  }
}
```

If a non-empty array is provided, you use this as the elements for the heap. To satisfy the heap's property, you loop through the array backward, starting from the first non-leaf node, and sift down all parent nodes.

You loop through only half of the elements because there's no point in sifting down **leaf** nodes, only parent nodes.



Number of parents = total number of elements / 2
4 = 8 / 2

With this method you can add this code to `ComparableHeapImpl`:

```
companion object {
  fun <T : Comparable<T>> create(
      elements: ArrayList<T>
  ): Heap<T> {
    val heap = ComparableHeapImpl<T>()
    heap.heapify(elements)
    return heap
  }
}
```

...and this code to `ComparatorHeapImpl`

```
companion object {
  fun <T: Any> create(
      elements: ArrayList<T>,
      comparator: Comparator<T>
  ): Heap<T> {
    val heap = ComparatorHeapImpl(comparator)
    heap.heapify(elements)
    return heap
  }
}
```

This code allows you to define a **static factory method** and create a `Heap` implementation starting from a given array and test your implementations.

Testing

You now have all the necessary tools to create and test a `Heap`. You can start using this code in order to create a `max-heap` of `Comparable` objects represented by `Int` values.

```
fun main() {
  val array = arrayListOf(1, 12, 3, 4, 1, 6, 8, 7) // 1
  val priorityQueue = ComparableHeapImpl.create(array) // 2
  while (!priorityQueue.isEmpty) { // 3
    println(priorityQueue.remove())
  }
}
```

In the previous code you:

1. create an `ArrayList` of `Int`s
2. using the array in order to create a `ComparableHeapImpl`
3. remove and print the max value until the `Heap` is empty

Notice that the elements are removed largest to smallest, and the following numbers are printed to the console:

```
12
8
7
6
4
3
1
1
```

If you want to create a `min-heap` you can replace the previous code in `main()` with the following:

```
fun main() {
  val array = arrayListOf(1, 12, 3, 4, 1, 6, 8, 7) // 1
  val inverseComparator = Comparator<Int> { o1, o2 ->  // 2
    override fun compare(o1: Int, o2: Int): Int = o2.compareTo(o1)
  }
  val minHeap = ComparatorHeapImpl.create(array, inverseComparator) // 3
  while (!minHeap.isEmpty) { // 4
    println(minHeap.remove())
  }
}
```

In this case you:

1. create an `ArrayList` of `Int`s
2. create an implementation of the `Comparator<Int>` which implements the inverse order for `Int`
3. using the array and the comparator in order to create a `ComparatorHeapImpl`
4. remove and print the value with highest priority (whose value this time is the lowest) until the `Heap` is empty

Running this code you'll get the the following output:

```
1
1
3
4
6
7
8
12
```

Challenges

Challenge 1: Find the nth smallest value

Write a function to find the `nth` smallest integer in an unsorted array. For example:

```
val integers = arrayListOf(3, 10, 18, 5, 21, 100)
```

If `n = 3`, the result should be `10`.

Solution 1

There are many ways to solve for the `nth` smallest integer in an unsorted array. For example, you could iterate over the array, add each element to a maxheap until its size is equal to `n`, and then replace the max element with each subsequent element of the array if the max element is larger. This way `peek()` will always return `nth` smallest element of the array.

Here's how you would obtain the `nth` smallest element using a maxheap:

```
fun getNthSmallestElement(n: Int, elements: ArrayList<Int>): Int? {
  if (n <= 0 || elements.isEmpty()) return null

  val heap = ComparableHeapImpl.create(arrayListOf<Int>())

  elements.forEach { // 1
    val maxElement = heap.peek()
    if (heap.count < n) {
      heap.insert(it) // 2
    } else if (maxElement != null && maxElement > it) {
      heap.remove() // 3
      heap.insert(it) // 4
    }
  }

  return heap.peek() // 5
}
```

Here's how it works:

1. Iterate over the array.
2. If the size of the maxheap is smaller than `n` insert the current element into the heap.
3. Otherwise, if the max element of the heap is larger than the current element, remove it from the heap.
4. Add the current element into the heap.

5. Once you iterated over the array, use `peek()` to return the max element of the heap, which would be `nth` smallest of the provided array.

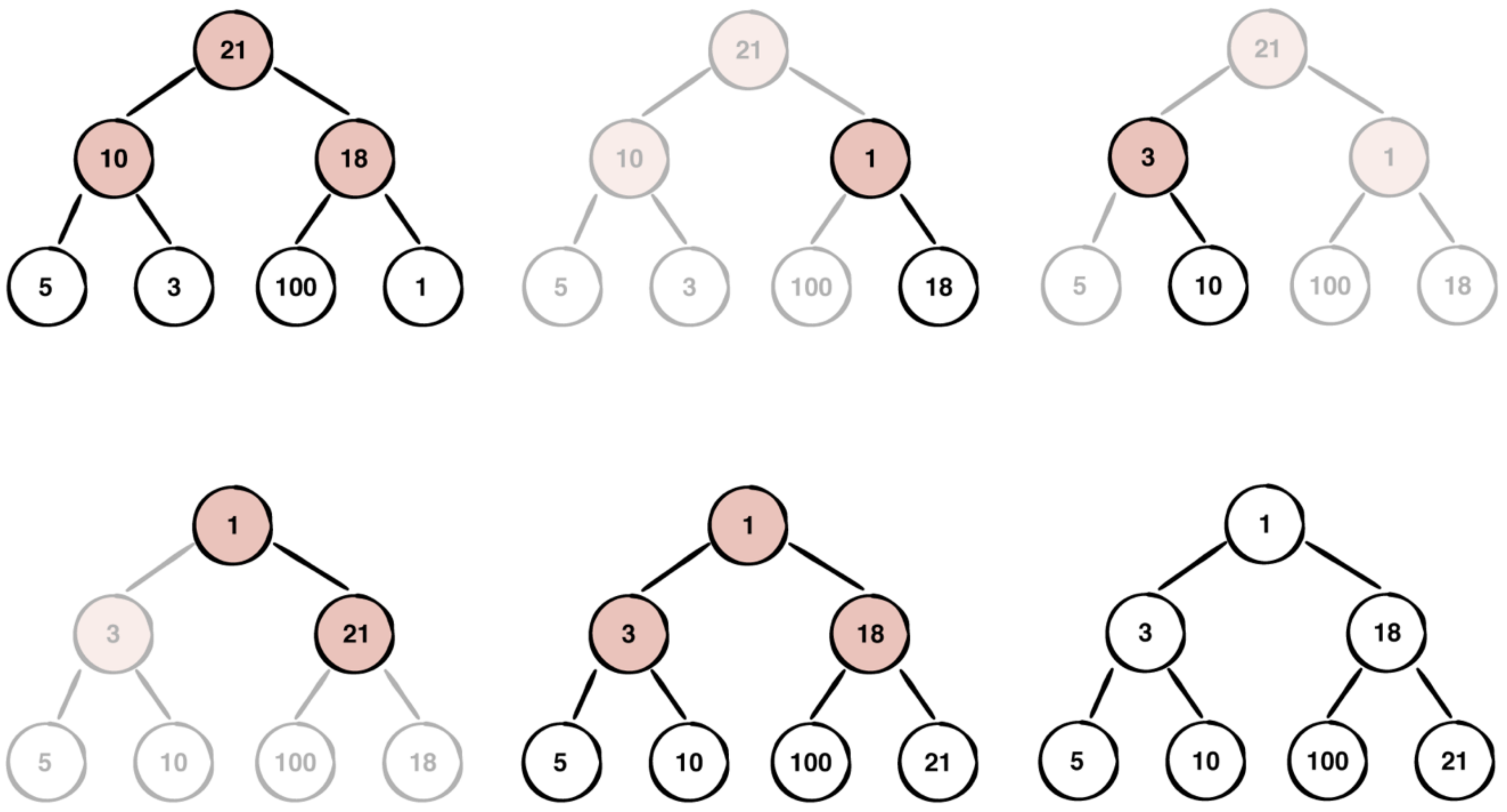
Every element insertion and removal from the heap takes *O*(log *n*). Keep in mind that you're also doing this `n` times. The overall time complexity is *O*(*n* log *n*).

Challenge 2: The min heap visualization

Given the following array list, visually construct a minheap. Provide a step-by-step diagram of how the minheap is constructed.

```
arrayListOf(21, 10, 18, 5, 3, 100, 1)
```

Solution 2



Challenge 3: Heap merge

Write a method that combines two heaps.

Solution 3

Add this as an additional function for your `AbstractHeap` class after defining the same operation on the `Heap` interface:

```
override fun merge(heap: AbstractHeap<T>) {
  elements.addAll(heap.elements)
  buildHeap()
}

private fun buildHeap() {
  if (!elements.isEmpty()) {
    (count / 2 downTo 0).forEach {
      siftDown(it)
    }
  }
}
```

To merge two heaps, you first combine both arrays which takes *O*(*m*), where `m` is the length of the `heap` you are merging.

Building the heap takes *O*(*n*). Overall the algorithm runs in *O*(*n*).

Challenge 4: Min heap check

Write a function to check if a given array is a minheap.

Solution 4

To check if the given array is a minheap, you only need to go through the parent nodes of the binary heap. To satisfy the minheap, every parent node must be less than or equal to its left and right child node.

Here’s how you can determine if an array is a minheap:

```
override fun isMinHeap(): Boolean {
  if (isEmpty) return true // 1
  (count / 2 - 1 downTo 0).forEach {
    // 2
    val left = leftChildIndex(it) // 3
    val right = rightChildIndex(it)
    if (left < count &&
      compare(elements[left], elements[it]) < 0) { // 4
      return false
    }
    if (right < count
      && compare(elements[right], elements[it]) < 0) { // 5
      return false
    }
  }
  return true // 6
}
```

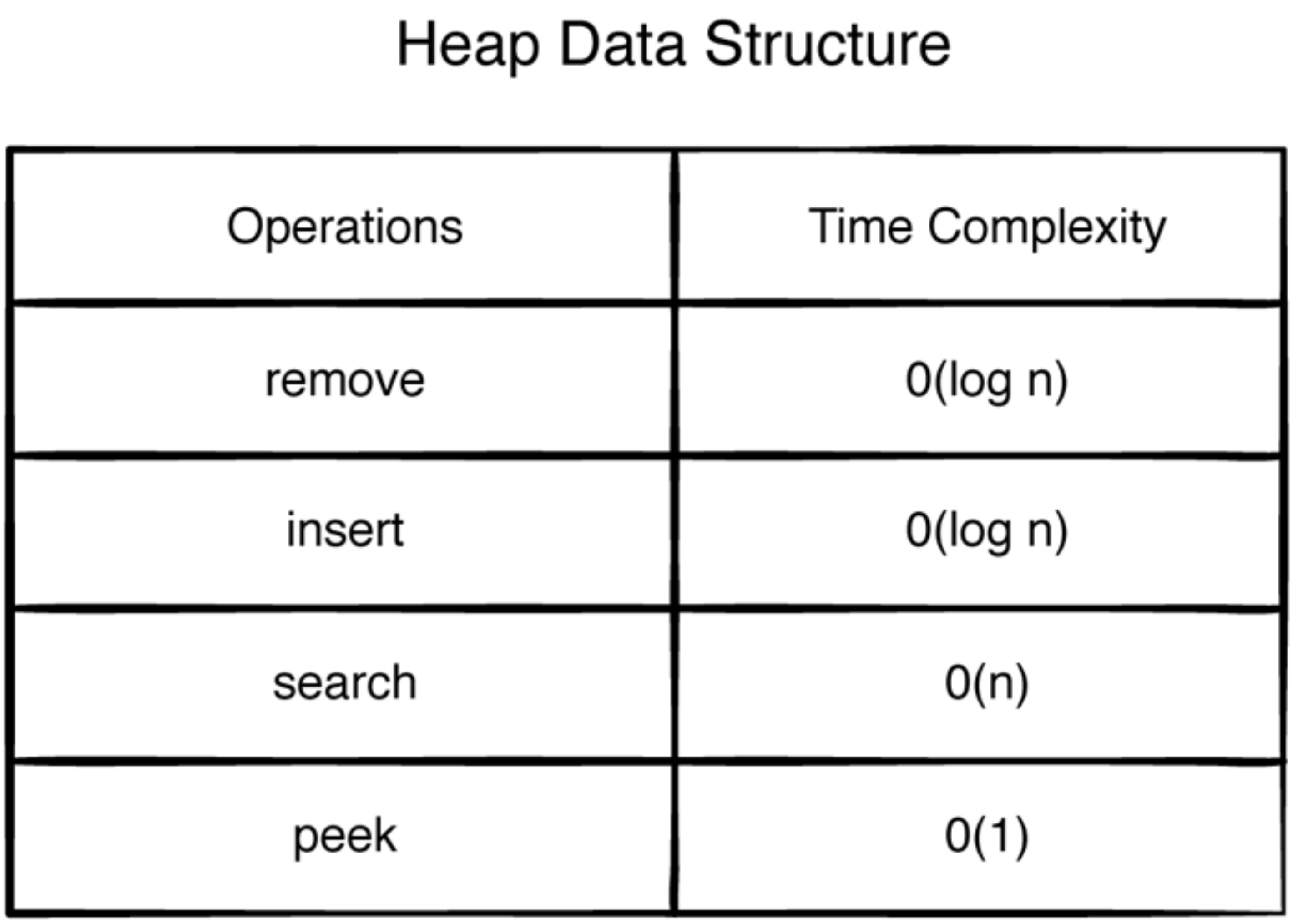
Here's how it works:

1. If the array is empty, it's a minheap.
2. Go through all of the parent nodes in the array in reverse order.
3. Get the left and right child index.
4. Check to see if the left element is less than the parent.
5. Check to see if the right element is less than the parent.
6. If every parent-child relationship satisfies the minheap property, return `true`.

The time complexity of this solution is *O*(*n*). This is because you still have to go through every element in the array.

Key points

- Here's a summary of the algorithmic complexity of the heap operations you implemented in this chapter:


Heap Data Structure

Operations	Time Complexity
remove	0(log n)
insert	0(log n)
search	0(n)
peek	0(1)

Heap operation time complexity

- The heap data structure is good for maintaining the highest or lowest priority element.
- Every time you insert or remove items from the heap, you must check to see if it satisfies the rules of the priority.

Have a technical question? Want to report a bug? You can ask questions and report bugs to the book authors in our official book forum here.