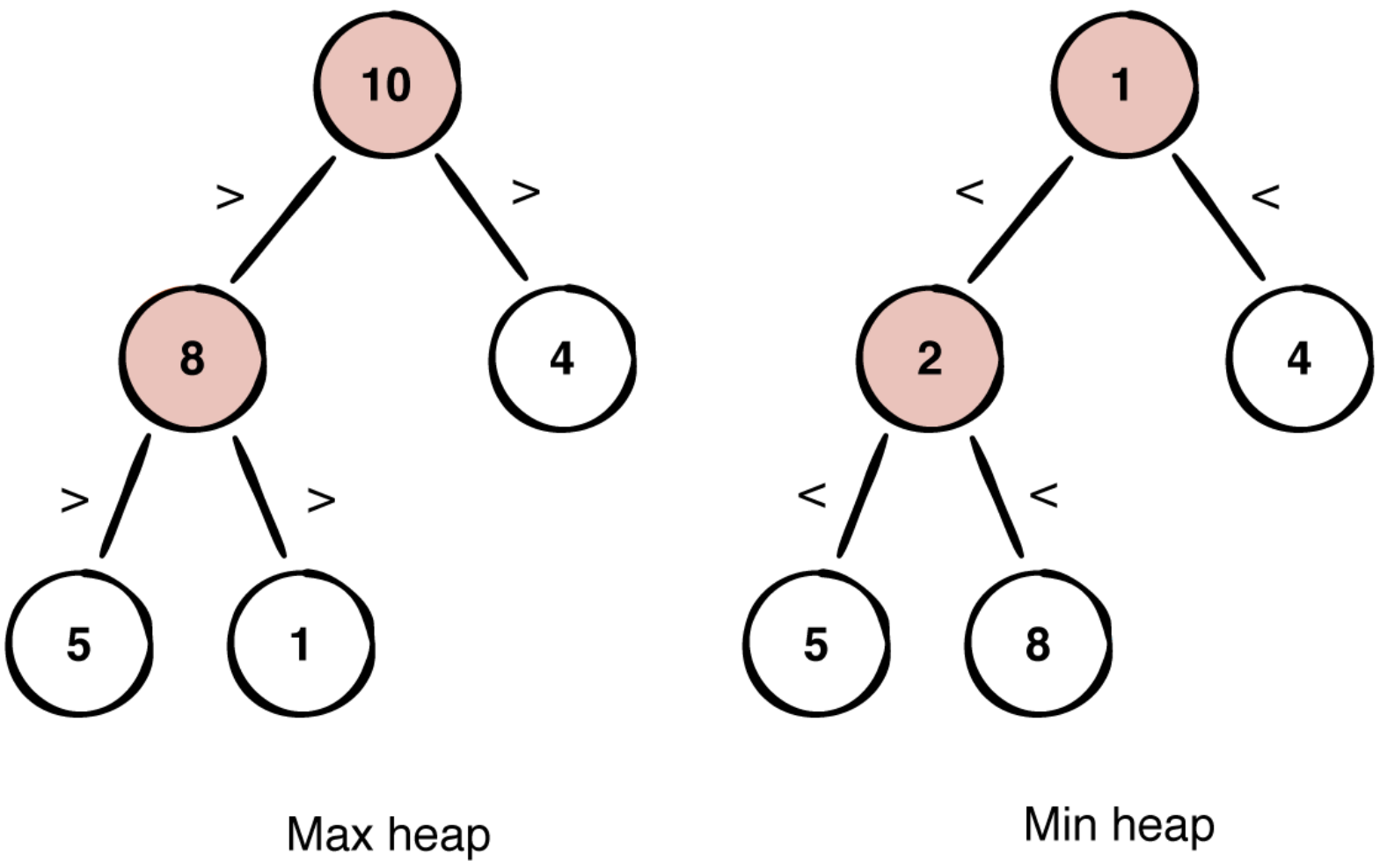
17 Heap Sort Written by Márton Braun

Heap sort is another comparison-based algorithm that sorts an array in ascending order using a heap. This chapter builds on the heap concepts presented in Chapter 12, “Heap Data Structure.”

Heap sort takes advantage of a heap being, by definition, a partially sorted binary tree with the following qualities:

1. In a max heap, all parent nodes are larger than their children.
2. In a min heap, all parent nodes are smaller than their children.

The diagram below shows a heap with parent node values underlined:



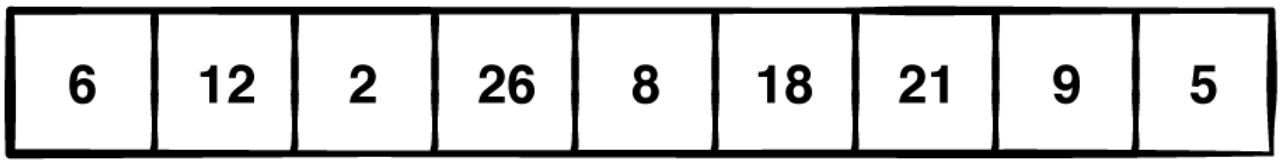
Getting started

Open up the starter project. This project already contains an implementation of a heap. Your goal is to extend `Heap` so it can also sort.

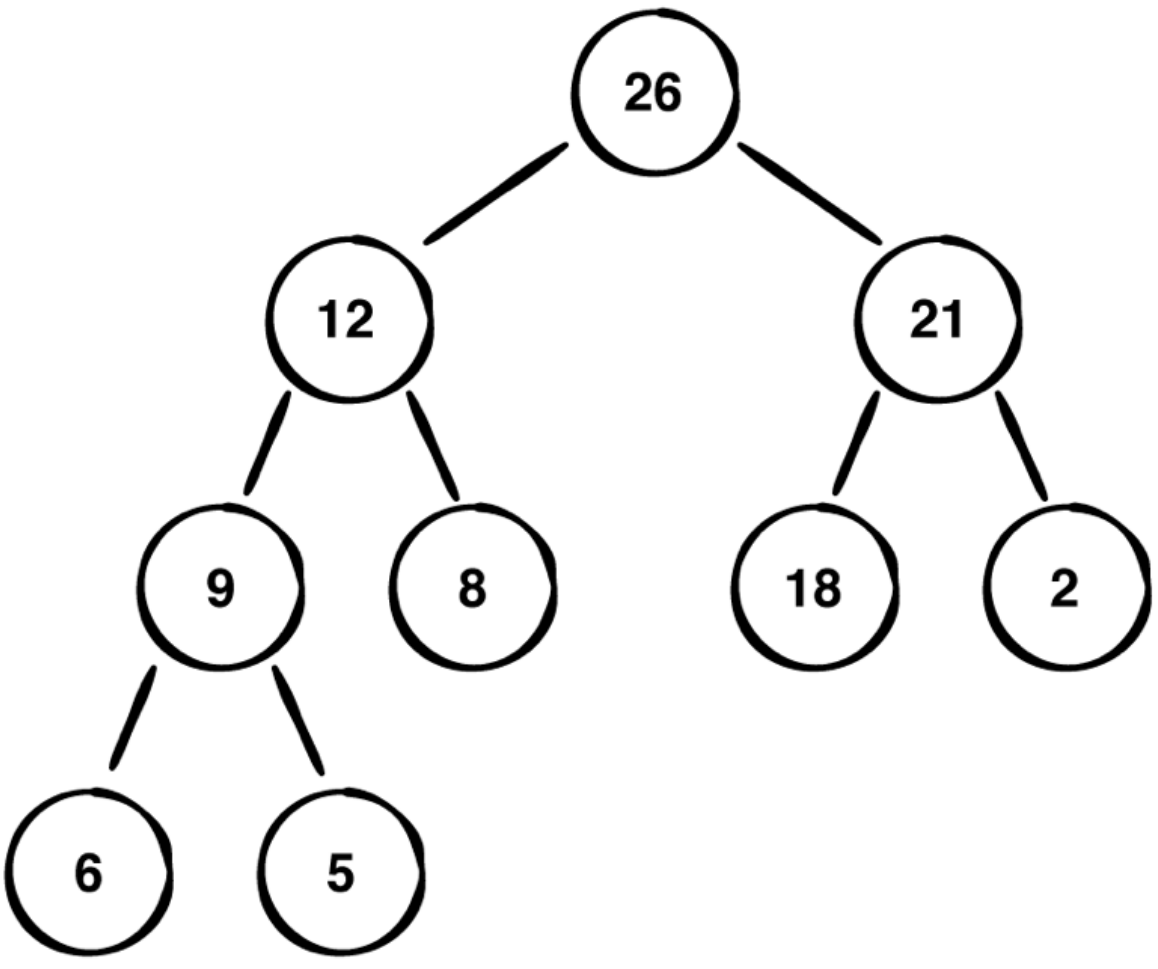
Before you get started, let's look at a visual example of how heap sort works.

Example

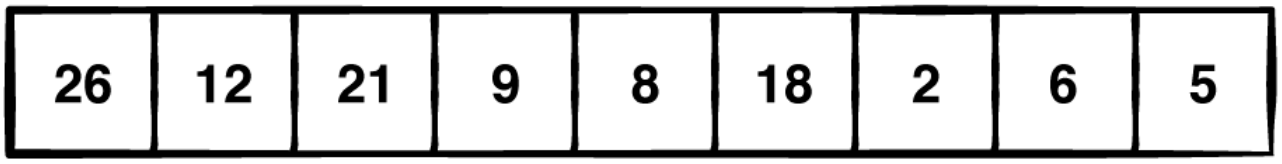
For any given unsorted array, to sort from lowest to highest, heap sort must first convert this array into a heap:



This conversion is done by sifting down all the parent nodes so that they end up in the right spot. The resulting heap is:



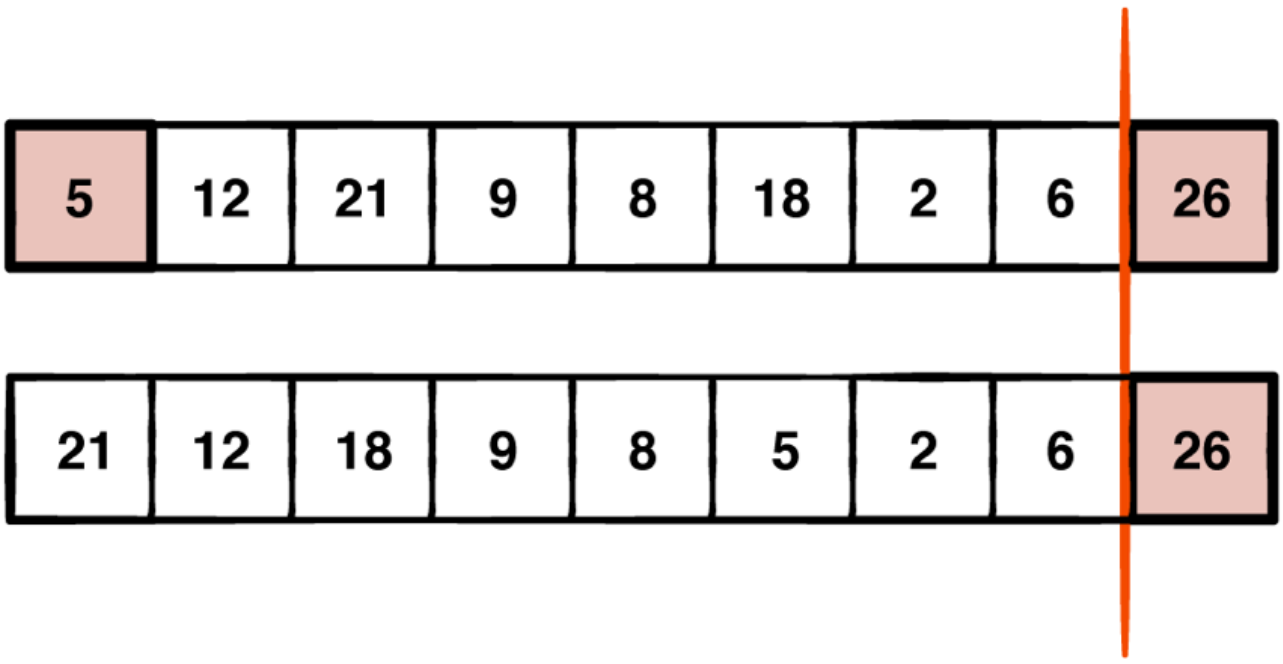
This corresponds with the following array:



Because the time complexity of a single sift-down operation is $O(\log n)$, the total time complexity of building a heap is $O(n \log n)$.

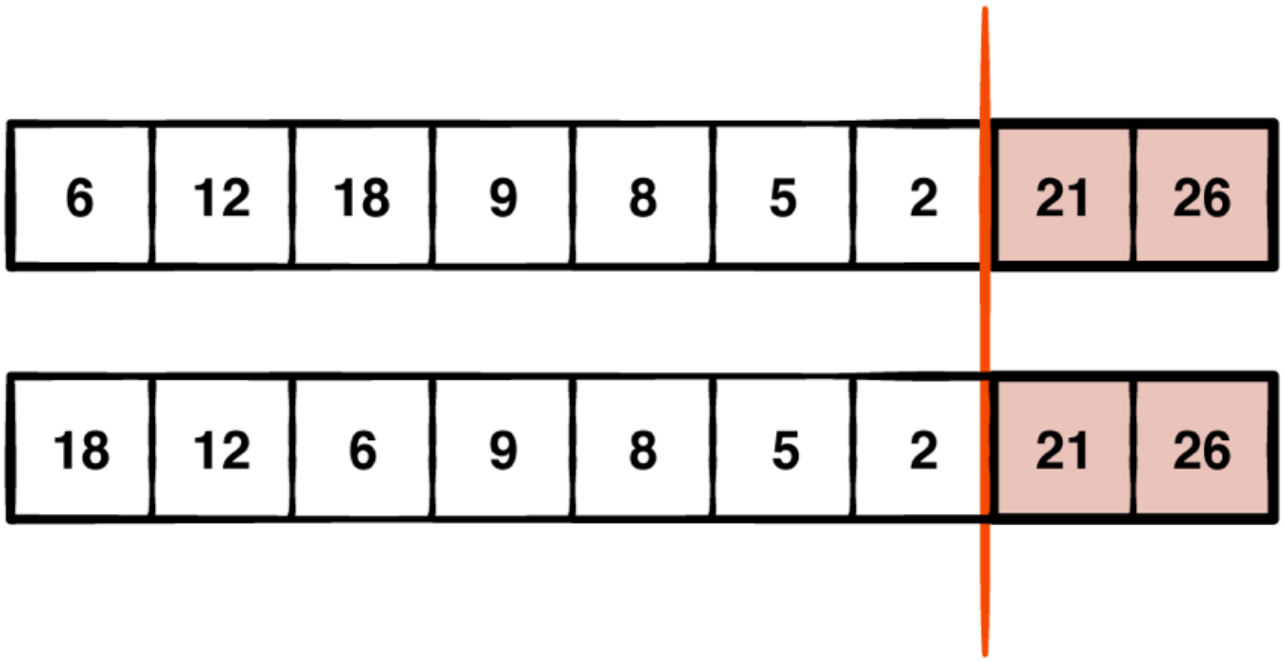
Let's look at how to sort this array in ascending order.

Because the largest element in a max heap is always at the root, you start by swapping the first element at index **0** with the last element at index ***n* - 1**. As a result of this swap, the last element of the array is in the correct spot, but the heap is now invalidated. The next step is, thus, to sift down the new root node **5** until it lands in its correct position.



Note that you exclude the last element of the heap as you no longer consider it part of the heap, but of the sorted array.

As a result of sifting down **5**, the second largest element **21** becomes the new root. You can now repeat the previous steps, swapping **21** with the last element **6**, shrinking the heap and sifting down **6**.



Starting to see a pattern? Heap sort is very straightforward. As you swap the first and last elements, the larger elements make their way to the back of the array in the correct order. You simply repeat the swapping and sifting steps until you reach a heap of size 1.

The array is then fully sorted.

5	8	6	2	9	12	18	21	26

8	5	6	2	9	12	18	21	26

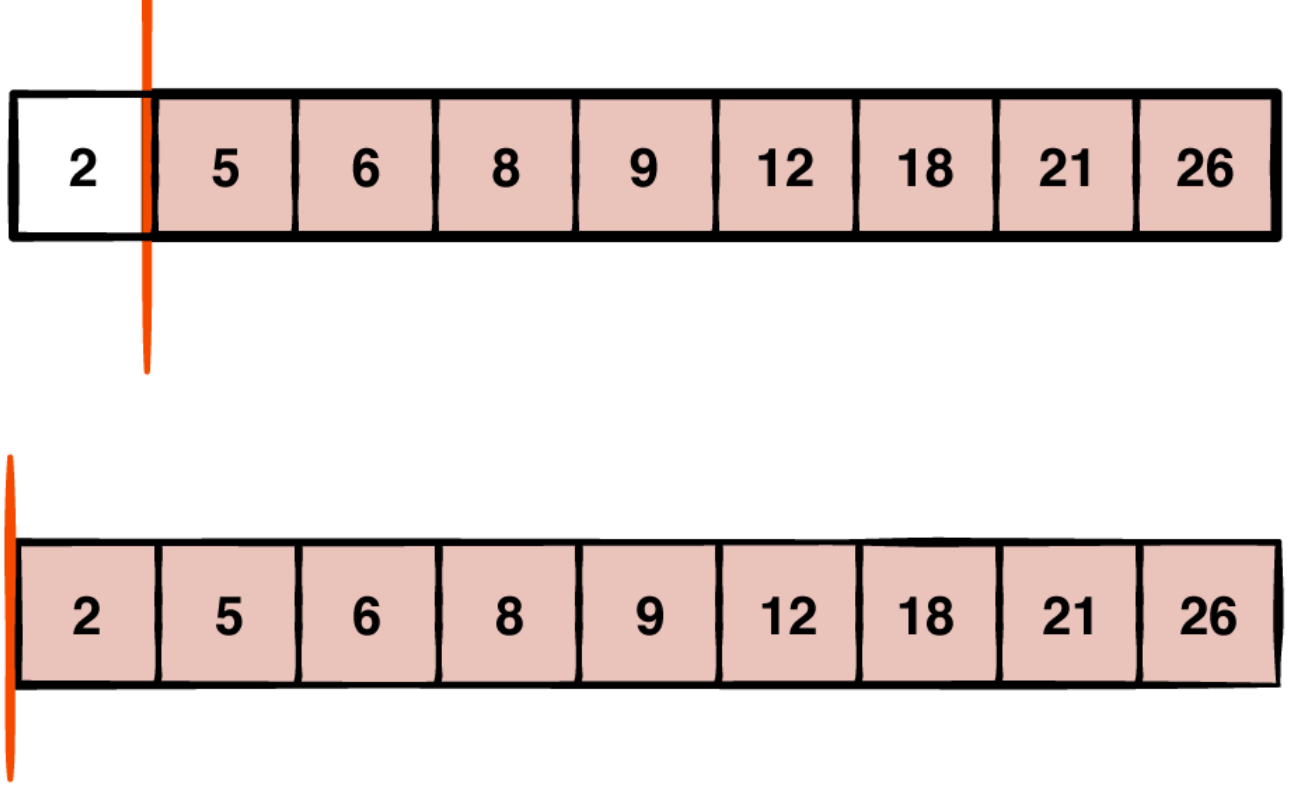
2	5	6	8	9	12	18	21	26

6	5	2	8	9	12	18	21	26

2	5	6	8	9	12	18	21	26

5	2	6	8	9	12	18	21	26

2	5	6	8	9	12	18	21	26



> **Note**: This sorting process is very similar to selection sort from **Chapter 14**.

Implementation

Next, you'll implement this sorting algorithm. The actual implementation is simple. You'll be performing the sort on an `Array` and reusing the algorithms already implemented in `Heap` in the form of extension functions. You'll reorder the elements using `heapify()`. After that, the heavy lifting will be done by a `siftDown()` method.

First, create a new file in the **heapsort** package, called **HeapSort.kt**. You might remember that the `siftDown()` method in `Heap` needs to determine the index of the left and right children of an element at a given index. You will start by copying the functions that do just that, as top level functions in this new file:

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

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

After that, you will copy the `siftDown()` method from `Heap`, and transform

it into an extension function for `Array<T>`. Since this extension function can work with all kinds of `T` elements, you'll also need to add a `Comparator<T>` to the parameters of the function. The transformed `siftDown()` function will look like this:

```
fun <T : Any> Array<T>.siftDown(
  index: Int,
  upTo: Int,
  comparator: Comparator<T>,
) {
  var parent = index
  while (true) {
    val left = leftChildIndex(parent)
    val right = rightChildIndex(parent)
    var candidate = parent
    if (left < upTo &&
      comparator.compare(this[left], this[candidate]) > 0
    ) {
      candidate = left
    }
    if (right < upTo &&
      comparator.compare(this[right], this[candidate]) > 0
    ) {
      candidate = right
    }
    if (candidate == parent) {
      return
    }
    this.swapAt(parent, candidate)
    parent = candidate
  }
}
```

The differences between this function and the method that the `Heap` has is that you operate on `this` array instead of `elements`, and that your `compare()` calls are addressed to the `comparator` you get as a parameter. The algorithm itself remains the same, so if you can't wrap your head

around this, you can take a look at the **Heap Data Structure** chapter again, which explains this function in detail.

Next, you'll need a `heapify()` function. As with `siftDown()`, it will be an extension function very similar to the one in `Heap`. This one will also have a `Comparator<T>` parameter, as it will have to call `siftDown()`. Copy this into **HeapSort.kt**:

```
fun <T : Any> Array<T>.heapify(comparator: Comparator<T>) {
  if (this.isNotEmpty()) {
    (this.size / 2 downTo 0).forEach {
      this.siftDown(it, this.size, comparator)
    }
  }
}
```

The final step is to implement the actual sorting. This is simple enough, here's how:

```
fun <T : Any> Array<T>.heapSort(comparator: Comparator<T>) {
  this.heapify(comparator) // 1
  for (index in this.indices.reversed()) { // 2
    this.swapAt(0, index) // 3
    siftDown(0, index, comparator) // 4
  }
}
```

Here's what's going on:

1. You reorder the elements so that the array looks like a `Heap`.
2. You loop through the array, starting from the last element.
3. You swap the first element and the last element. This moves the largest unsorted element to its correct spot.
4. Because the heap is now invalid, you must sift down the new root node. As a result, the next largest element will become the new root.

Note that, in order to support heap sort, you've added an additional parameter `upTo` to the `siftDown()` method. This way, the sift down only uses the unsorted part of the array, which shrinks with every iteration of the loop.

Finally, give your new method a try. Add this to the `main()` function in **Main.kt**:

```
"Heap sort" example {
  val array = arrayOf(6, 12, 2, 26, 8, 18, 21, 9, 5)
  array.heapSort(ascending)
  print(array.joinToString())
}
```

This should print:

```
---Example of Heap sort---
2, 5, 6, 8, 9, 12, 18, 21, 26
```

Performance

Even though you get the benefit of in-memory sorting, the performance of heap sort is *O*(*n* log *n*) for its best, worse and average cases. This is because you have to traverse the whole array once and, every time you swap elements, you must perform a sift down, which is an *O*(log *n*) operation.

Heap sort is also not a stable sort because it depends on how the elements are laid out and put into the heap. If you were heap sorting a deck of cards by their rank, for example, you might see their suite change order with respect to the original deck.

Challenges

Challenge 1: Fewest comparisons

When performing a heap sort in ascending order, which of these starting arrays requires the fewest comparisons?

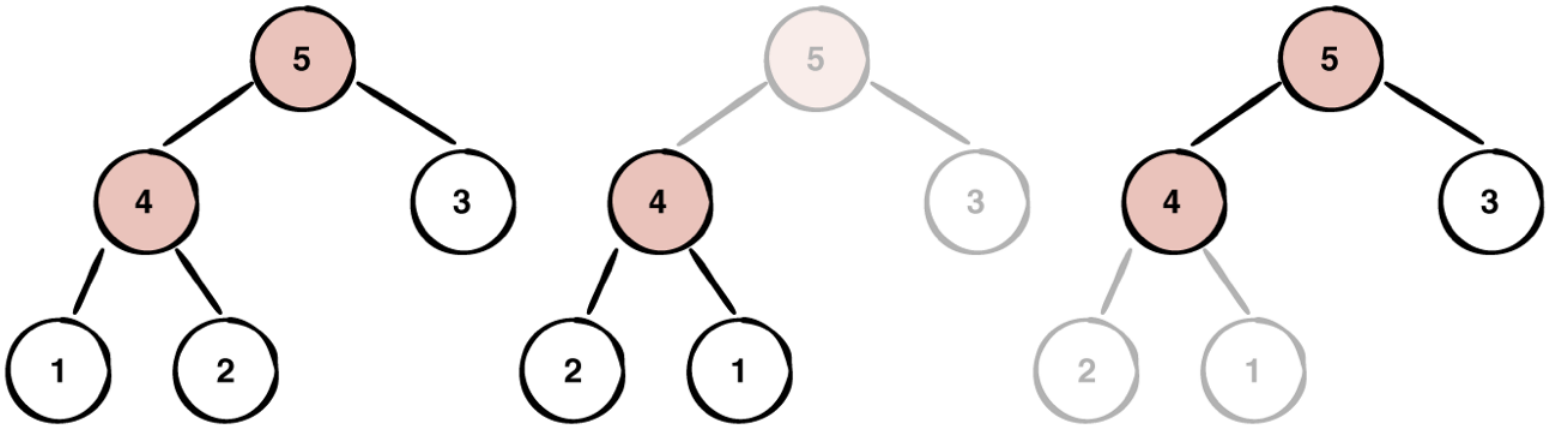
- `[1,2,3,4,5]`
- `[5,4,3,2,1]`

Solution 1

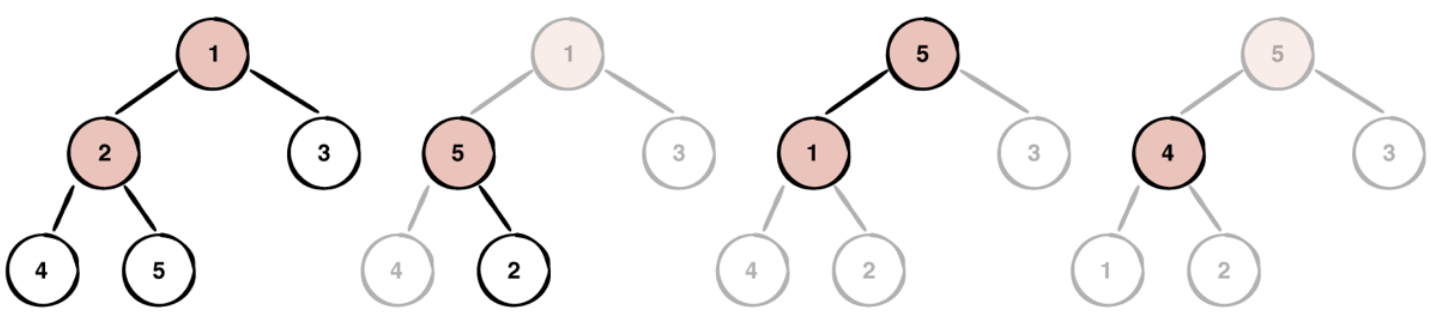
When sorting elements in ascending order using heap sort, you first need a max heap. What you really need to look at is the number of comparisons that happen when constructing the max heap.

`[5,4,3,2,1]` will yield the fewest number of comparisons, since it's already a max heap and no swaps take place.

When building a max heap, you only look at the parent nodes. In this case there are two parent nodes, with two comparisons.



`[1,2,3,4,5]` will yield the most number of comparisons. There are two parent nodes, but you have to perform three comparisons:



Challenge 2: Descending sort

The current example of heap sort sorts the elements in **ascending** order. How would you sort in **descending** order?

Solution 2

This is a simple one as well. You just need to swap the `ascending` comparator with a descending one when you create the `SortingHeap`. Looking at how `ascending` is defined, you can easily come up with a `descending`:

```
val descending = Comparator { first: Int, second: Int ->
    when {
      first < second -> 1
      first > second -> -1
      else -> 0
    }
}
```

If you haven't already noticed, you just need to change the signs for -1 and 1. That's it! Now you can create another example in **Main.kt** to see how it works:

```
"Heap sort descending" example {
  val array = arrayOf(6, 12, 2, 26, 8, 18, 21, 9, 5)
  array.heapSort(descending)
  print(array.joinToString())
}
```

The result is a descending sorted list:

```
---Example of Heap sort descending---
26, 21, 18, 12, 9, 8, 6, 5, 2
```

Key points

- Heap sort leverages the max-heap data structure to sort elements in an array.
- Heap sort sorts its elements by following a simple pattern: swap the first and last element, perform a `sift-down`, decrease the array size by one; then repeat.
- The performance of heap sort is $O(n \log n)$ for its best, worse and average cases.

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.