13 Priority Queues Written by Irina Galata

Queues are lists that maintain the order of elements using *first in, first out* (FIFO) ordering. A **priority queue** is another version of a queue. However, instead of using FIFO ordering, elements are dequeued in priority order.

A priority queue can have either a:

- **Max-priority**: The element at the front is always the largest.
- **Min-priority**: The element at the front is always the smallest.

A priority queue is especially useful when you need to identify the maximum or minimum value within a list of elements.

In this chapter, you'll learn the benefits of a priority queue and build one by leveraging the existing queue and heap data structures that you studied in previous chapters.

Applications

Some useful applications of a priority queue include:

- **Dijkstra's algorithm**: Uses a priority queue to calculate the minimum cost.
- **A* pathfinding algorithm**: Uses a priority queue to track the unexplored routes that will produce the path with the shortest length.
- **Heap sort**: Many heap sorts use a priority queue.
- **Huffman coding**: Useful for building a compression tree. A min-priority queue is used to repeatedly find two nodes with the smallest frequency that don't yet have a parent node.

Priority queues have many more applications and practical uses; the list above represents only a handful.

Common operations

In Chapter 5, "Queues", you established the following interface for queues:

```
interface Queue<T: Any> {

  fun enqueue(element: T): Boolean

  fun dequeue(): T?

  val count: Int
    get

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

  fun peek(): T?
}
```

A priority queue has the same operations as a normal queue, so only the implementation will be different.

The priority queue will implement the `Queue` interface and the common operations:

- **`enqueue`**: Inserts an element into the queue. Returns `true` if the operation is successful.
- **`dequeue`**: Removes the element with the highest priority and returns it. Returns `null` if the queue is empty.
- **`count`**: Property for the number of items in the queue.
- **`isEmpty`**: Checks if the queue is empty. The implementation just checks if the **count** property is 0.
- **`peek`**: Returns the element with the highest priority without removing it. Returns `null` if the queue is empty.

You're ready to look at different ways to implement a priority queue.

Implementation

You can create a priority queue in the following ways:

1. **Sorted array**: This is useful to obtain the maximum or minimum value of an element in $O(1)$ time. However, insertion is slow and requires $O(n)$ because you have to search the right position for every element you insert.
2. **Balanced binary search tree**: This is useful in creating a double-ended priority queue, which features getting both the minimum and maximum value in $O(\log n)$ time. Insertion is better than a sorted array, also in $O(\log n)$.
3. **Heap**: This is a natural choice for a priority queue. A heap is more efficient than a sorted array because a heap only needs to be partially sorted. All heap operations are $O(\log n)$ except extracting the min value from a min priority heap is a lightning-fast $O(1)$. Likewise, extracting the max value from a max priority heap is also $O(1)$.

Next, you'll look at how to use a heap to create a priority queue.

To get started, open the starter project. Inside, you'll notice the following files:

1. **Heap.kt**: The heap data structure (from the previous chapter) that you'll use to implement the priority queue.
2. **Queue.kt**: Contains the interface that defines a queue.

Add the following abstract class:

```
// 1
abstract class AbstractPriorityQueue<T: Any> : Queue<T> {

  // 2
  abstract val heap: Heap<T>
    get

  // more to come ...
}
```

Here's a closer look at the code:

1. `AbstractPriorityQueue` implements the `Queue` interface and is generic in the type `T`. It's an abstract class because you want to manage comparison using either `Comparable<T>` objects or an external `Comparator<T>` implementation.
2. You're going to use a `Heap<T>`, so you need an abstract property that the specific implementation will define.

To implement the `Queue` interface, add the following to `AbstractPriorityQueue`:

```
  // 1
  override fun enqueue(element: T): Boolean {
    heap.insert(element)
    return true
  }
```

```
  // 2
  override fun dequeue() = heap.remove()

  // 3
  override val count: Int
    get() = heap.count

  // 4
  override fun peek() = heap.peek()
```

The heap is a perfect candidate for a priority queue. To implement the operations of a priority queue, you need to call various methods of a heap.

1. By calling `enqueue()`, you add the element into the heap using `insert()`, which guarantees to arrange data internally so that the one with the highest priority is ready to extract. The overall complexity of `enqueue()` is the same as `insert()`: $O(\log n)$.

2. By calling `dequeue()`, you remove the root element from the heap using `remove()`. The Heap guarantees to get the one with the highest priority. The overall complexity of `dequeue()` is the same as `remove()`: $O(\log n)$.

3. `count` uses the same property of the heap.

4. `peek()` delegates to the same method of the heap.

Using Comparable objects

`AbstractPriorityQueue<T>` implements the `Queue<T>` interface delegating to a `Heap<T>`. You can implement this using either `Comparable<T>` objects or a `Comparator<T>`. In this example, you'll use the former.

Add the following code to **PriorityQueue.kt**.

```
class ComparablePriorityQueueImpl<T : Comparable<T>> :
  AbstractPriorityQueue<T>() {

  override val heap = ComparableHeapImpl<T>()
}
```

Here, you implement `heap` using a `ComparableHeapImpl<T>` object. The `ComparablePriorityQueueImpl<T>` needs an object that implements the `Comparable<T>` interface.

To test this implementation, add the following code to **Main.kt**:

```
"max priority queue" example {
  // 1
  val priorityQueue = ComparablePriorityQueueImpl<Int>()
  // 2
  arrayListOf(1, 12, 3, 4, 1, 6, 8, 7).forEach {
    priorityQueue.enqueue(it)
  }
  // 3
  while (!priorityQueue.isEmpty) {
    println(priorityQueue.dequeue())
  }
}
```

In this example, you:

1. Create a `ComparablePriorityQueueImpl<Int>` using `Int` as generic type value which is `Comparable<Int>`.
2. Enqueue the value from an unsorted array into the priority queue.
3. Dequeue all of the values from the priority queue.

When you run the code, notice the elements are removed largest to smallest. The following is printed to the console:

```
---Example of max priority queue---
```

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

Using Comparator objects

Providing different `Comparator<T>` interface implementations allows you to choose the priority criteria.

Add the following code to **PriorityQueue.kt**.

```
class ComparatorPriorityQueueImpl<T: Any>(
  private val comparator: Comparator<T>
) : AbstractPriorityQueue<T>() {

  override val heap = ComparatorHeapImpl(comparator)
}
```

Here, the only difference is the value provided to `heap`, which is now a `ComparatorHeapImpl<T>` and needs a `Comparator<T>` that you provide as a constructor parameter.

To test this implementation, add the following code to `main()` inside **Main.kt**:

```
"min priority queue" example {
  // 1
  val stringLengthComparator = Comparator<String> { o1, o2 ->
    val length1 = o1?.length ?: -1
    val length2 = o2?.length ?: -1
    length1 - length2
  }
```

```
  // 2
  val priorityQueue = ComparatorPriorityQueueImpl(stringLengthComparator)
  // 3
  arrayListOf("one", "two", "three", "four", "five", "six", "seven", "eight
    priorityQueue.enqueue(it)
  }
  // 4
  while (!priorityQueue.isEmpty) {
    println(priorityQueue.dequeue())
  }
}
```

In this example, you:

1. Create a `Comparator<String>` implementation that compares `String` based on the length from the longest to the shortest.
2. Create a `ComparatorPriorityQueueImpl` using the previous comparator in the constructor.
3. Enqueue value from an unsorted array as `String` into the priority queue.
4. Dequeue all the values from the priority queue.

When you run the code, you'll see this output where the `String` objects are sorted from the longest to the shortest.

```
---Example of min priority queue---
three
eight
seven
nine
four
five
one
two
six
```

Challenges

Challenge 1: Constructing ArrayList priority queues

You learned to use a heap to construct a priority queue by implementing the `Queue` interface. Now, construct a priority queue using an `ArrayList`:

```
interface Queue<T: Any> {

  fun enqueue(element: T): Boolean

  fun dequeue(): T?

  val count: Int
    get

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

  fun peek(): T?
}
```

Solution 1

Recall that a priority queue dequeues elements in priority order. It could either be a min or max priority queue. To make an array-based priority queue, you need to implement the `Queue` interface. Instead of using a heap, you can use an array list.

First, add the following code to **PriorityQueueArray.kt**:

```
// 1
abstract class AbstractPriorityQueueArrayList<T: Any> : Queue<T> {

  // 2
  protected val elements = ArrayList<T>()

  // 3
  abstract fun sort()
```

```
  // more to come ...
}
```

Here, you:

1. Define the `AbstractPriorityQueueArrayList<T>` abstract class implementing the `Queue<T>` interface.
2. Define the `elements` property of type `ArrayList<T>` as protected so it can be accessed by the classes extending this.
3. The `sort` abstract function is the one you're going to implement in different ways depending on the usage of `Comparable<T>` objects or a `Comparator<T>`.

With this code, some of the `Queue<T>` operations come for free, so add the following code:

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

override fun peek() = elements.firstOrNull()
```

Here, you're assuming that the `ArrayList<T>` is always sorted, and if it's not empty, it always contains the element with the highest priority in position 0. This assumption allows you to implement the `dequeue` operation using this code:

```
override fun dequeue() =
  if (isEmpty) null else elements.removeAt(0)
```

It's important to know how the `dequeue` operation is *O*(n) because the removal of an item in position 0 requires the shift of all of the other elements. A possible optimization, which you can try as an exercise, is to put the element with the highest priority in the last position so that you don't have to shift any elements but instead reduce the size by 1.

Next, add the `enqueue` method. This is the one responsible for the sorting:

```
override fun enqueue(element: T): Boolean {
  // 1
  elements.add(element)
  // 2
  sort()
  // 3
  return true
}
```

To `enqueue` an element into an array-based priority queue, this code does the following:

1. Appends the element in the `ArrayList`.
2. Sorts the elements into the `ArrayList` using the `sort` function.
3. Returns `true` because the element was inserted with success.

The overall time complexity here is the complexity of the `sort` implementation, because the `add` operation of the `ArrayList` is *O*(*1*).

Before implementing `sort()`, add this code so you can print the priority queue in a nice format:

```
override fun toString() = elements.toString()
```

You can now provide different realizations for the `AbstractPriorityQueueArrayList<T>` class and the `sort` operation.

To manage `Comparable<T>` objects, add the following code:

```
class ComparablePriorityQueueArrayList<T : Comparable<T>> : AbstractPriorit
  override fun sort() {
    Collections.sort(elements)
```

```
  }
}
```

Here, you implement `sort()` using the same method of the `Collections` class. The complexity, in this case, is *O*(*n log n*); it's the same if you want to use a `Comparator<T>`, which you can do using the following code:

```
class ComparatorPriorityQueueArrayList<T: Any>(
  private val comparator: Comparator<T>
) : AbstractPriorityQueueArrayList<T>() {
  override fun sort() {
    Collections.sort(elements, comparator)
  }
}
```

Can you do better? Sure! If you always insert the new item in the right position, you have to shift all of the other elements — and this can be done in *O*(*n*). You can now write this implementation for `Comparable<T>` objects:

```
class CustomPriorityQueueArrayList<T : Comparable<T>> : AbstractPriorityQue
  override fun sort() {
    var index = count - 2
    while (index >= 0 &&
      elements[index + 1].compareTo(elements[index]) > 0) {
      swap(index, index + 1)
      index--;
    }
  }

  private fun swap(i: Int, j: Int) {
    val tmp = elements[i]
    elements[i] = elements[j]
    elements[j] = tmp
  }
}
```

This is an $O(n)$ operation since you have to shift the existing elements to the left by one until you find the right position.

Congratulations, you now have an array-based priority queue.

To test the priority queue, add the following code to `main()`:

```
"max priority array list based queue" example {
  val priorityQueue = CustomPriorityQueueArrayList<Int>()
  arrayListOf(1, 12, 3, 4, 1, 6, 8, 7).forEach {
    priorityQueue.enqueue(it)
  }
  priorityQueue.enqueue(5)
  priorityQueue.enqueue(0)
  priorityQueue.enqueue(10)
  while (!priorityQueue.isEmpty) {
    println(priorityQueue.dequeue())
  }
}
```

Challenge 2: Sorting

Your favorite concert was sold out. Fortunately, there's a waitlist for people who still want to go. However, the ticket sales will first prioritize someone with a military background, followed by seniority.

Write a `sort` function that returns the list of people on the waitlist by the appropriate priority. `Person` is provided below and should be put inside **Person.kt**:

```
data class Person(
  val name: String,
  val age: Int,
  val isMilitary: Boolean)
```

Solution 2

Given a list of people on the waitlist, you would like to prioritize the people in the following order:

1. Military background.
2. Seniority, by age.

The best solution for this problem is to put the previous logic into a `Comparator<Person>` implementation and then use the proper priority queue implementation. In this way, you can give `Person` objects different priority providing different `Comparator<Person>` implementations.

Add this code to **Person.kt**:

```
object MilitaryPersonComparator : Comparator<Person> {
  override fun compare(o1: Person, o2: Person): Int {
    if (o1.isMilitary && !o2.isMilitary) {
      return 1
    } else if (!o1.isMilitary && o2.isMilitary) {
      return -1
    } else if (o1.isMilitary && o2.isMilitary) {
      return o1.age.compareTo(o2.age)
    }
    return 0
  }
}
```

To test your priority sort function, try a sample data set by adding the following:

```
  "concert line" example {
    val p1 = Person("Josh", 21, true)
    val p2 = Person("Jake", 22, true)
    val p3 = Person("Clay", 28, false)
    val p4 = Person("Cindy", 28, false)
    val p5 = Person("Sabrina", 30, false)
    val priorityQueue = ComparatorPriorityQueueImpl(MilitaryPersonComparato
    arrayListOf(p1, p2, p3, p4, p5).forEach {
```

```
      priorityQueue.enqueue(it)
    }
    while (!priorityQueue.isEmpty) {
      println(priorityQueue.dequeue())
    }
  }
```

Running the previous code, you'll get this output:

```
---Example of concert line---
Jake
Josh
Cindy
Clay
Sabrina
```

Key points

- A priority queue is often used to find the element in priority order.
- The `AbstractPriorityQueue<T>` implementation creates a layer of abstraction by focusing on key operations of a `queue` and leaving out additional functionality provided by the heap data structure.
- This makes the priority queue's intent clear and concise. Its only job is to enqueue and dequeue elements, nothing else.
- The `AbstractPriorityQueue<T>` implementation is another good example of **Composition over (implementation) inheritance**.

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.