# 9 Priority Queues

This chapter discusses a collection data type that is similar to but generalizes the queue data type—whereas a queue implements a first-come first-serve policy, a priority queue has a notion of the priority of each member item and the item with the highest priority is served first. To motivate the priority queue data type, consider the following standard job interview question:

> *We are presented with a huge list of numbers that arrive one by one, and we want to find the smallest* $k$ *numbers. The input list is gigantic and cannot be stored entirely in memory. However,* $k$ *is small relative to the length of the entire list* $n$ *(i.e.,* $k \ll n$*). What are we to do?*

**Priority Queue.** Like a special/fast lane, items (e.g., jobs) in a priority queue are serviced according to how "important" they are, not necessarily first-come first-serve. This has far-ranging applications.

A natural idea is to maintain a list of the smallest $k$ numbers we have seen so far and update that list as we observe more elements. To carry out this idea, observe that a new number will replace a number in the existing list if it is among the smallest $k$ numbers—and the number that will be removed from the list is the largest number in the list.

Translating this idea into code is straightforward. In an example Java implementation below, the method `receive` will be called every time a number arrives and the variable `bottom` keeps the smallest $k$ numbers so far.

**Code 9.1**: Find the smallest k numbers from a long list.

```
1  public class SmallestK {
2      List<Integer> bottom;   // keep the smallest k numbers
3      int k;
4      SmallestK(int k) { bottom = new ArrayList<>(); this.k=k; }
5
6      void receive(int num) {
7          bottom.add(num);
8          while (bottom.size() > k) {
9              int largest = Collections.max(bottom);
10             bottom.remove(largest);
11         }
12     }
13
14     List<Integer> bottomK() { return new ArrayList<>(bottom); }
15 }
```

**Example.** Consider $k = 3$ and the input numbers `20 4 9 11 10 8 7 6 3 1`. Over time, the list maintained (`bottom` in the code) looks as follows:

| New # | `bottom` |
|---|---|
| 20 | [**20**] |
| 4 | [20, **4**] |
| 9 | [20, 4, **9**] |
| 11 | [~~20~~, 4, 9, **11**] |
| 10 | [4, 9, ~~11~~, **10**] |
| 8 | [4, 9, ~~10~~, **8**] |
| 7 | [4, ~~9~~, 8, **7**] |
| 6 | [4, ~~8~~, 7, **6**] |
| 3 | [4, ~~7~~, 6, **3**] |
| 1 | [4, ~~6~~, 3, **1**] |

The more involved question is, what is the running time and how much space does it need? A moment's thought reveals that after each arrival (i.e., call to `receive`), `bottom` is kept at length at most $k$. Inside `receive`, the list can temporarily grow to length at most $k+1$ but will be shrunk back to length at most $k$ before the method terminates. Both `Collections.max` and `bottom.remove(.)` have to each look through the list, costing $O(k)$. So per arrival, we spend $O(k)$ time. Furthermore, the space requirement is also $O(k)$. This means for an input sequence of length $n$, this costs us $O(nk)$ time.

Our goal in this chapter is to develop a data type that makes it possible to quickly find and remove the maximum item in a collection—without harming the speed of adding an item to the collection too much.

## 9.1 A Priority Queue Data Type

A *priority queue* stores a collection of elements where each element has an associated priority that decides the ordering within the queue. We model the priorities as a function $p(e)$ that returns the priority of an element $e$. This function is mainly for exposition purposes and is often never materialized or stored anywhere in the implementation. Priorities determine the relative positions of the elements in the queue. Below are three typical scenarios:

- The priority is the element itself, so $p(e) = e$.
- The priority is easily computable from the element, so $p(e)$ takes practically no time to compute from $e$.
- The priority is stored as part of the element: for example, whereas the actual element is `e`, we store inside our data structure a pair `(priority, e)`, so the priority $p(e)$ is `priority` and is retrieved from the storage.

**Operations.** In addition to basic collection operations (e.g., `add`, `isEmpty`, `size`), a priority queue data type supports the following operations:

- `findMax()` — return without removing the element with the highest priority.
- `removeMax()` — remove and return the element with the highest priority.

Code 9.2 shows a minimal Java interface for the priority queue data type as discussed. Elements of such a priority queue are of a generic type `T`. By using Java's `Comparable` interface, the users can specify how the elements will be compared according to their priorities. For the purpose of this chapter, if multiple elements have the same highest priority, these operations can return any one of them in a consistent manner. While this variant is designed for efficiently extracting the maximum, variants that extract the minimum are prevalent and can be defined symmetrically.

It is worth noting that the priority queue data type is different from the first-in first-out (FIFO) queue studied previously. Unlike in a standard queue, a priority queue removes the highest-priority item first. Indeed, many common scenarios require that the "client" with the highest priority be helped first (e.g., an emergency-room waiting line, plane taking off, packet prioritization),

**Code 9.2**: A minimal priority queue interface in Java.

```java
interface PriorityQueue<T extends Comparable<? super T>> {
    // is this empty?
    boolean isEmpty();

    // add elt to the priority queue
    void add(T elt);

    // return the maximum value
    T findMax();

    // remove the maximum value
    void removeMax();

    // how many entries?
    int size();
}
```

regardless of the arrival order. Thus, the first-in first-out (FIFO) queue we studied previously does not capture this setting. In this sense, a priority queue is closely related to a sorted sequence; however, a priority queue is more dynamic—new elements are added over time (interleaved with the removal operation) and the max-priority element is defined for that instant.

**Applications.** Aside from the motivating example at the start of this chapter, the priority queue data type has found many applications, including

- sorting — we first insert all the elements into a priority queue and begin to successively remove elements from the queue (they will appear in descending order).
- event simulation and job scheduling — we keep a priority queue ordered by event time or job priority; the "jobs" can then be processed in chronological order or in a way that the important jobs do not starve).
- graph exploration and game playing AI — we keep a priority queue full of possible next moves ordered by how likely they will turn out to be a good move and use the priority queue to help pick the best next move.

In many of these applications, the priority queue data structure will have to handle a large number of entries, calling for their operations to be supported efficiently. We will now explore a few implementation options.

## 9.2 Basic Implementations

### Implementation I: Unordered Sequence

We start out with an extremely simple design. In the first implementation, our underlying data structure will be a sequence data type. In Java, we can choose

among a number of options, for example, `ArrayList` and `LinkedList`. We will store the elements in the collection unordered. Therefore, it is straightforward to add an element to the collection. Each new element can be appended to the back of the sequence, a quick operation for both the array list and linked list; however, finding the maximum and removing the maximum, though easy to implement, will take some time. An example Java implementation for the `Integer` element type is below:

**Code 9.3**: A priority queue kept as an unsorted sequence

```
1  import java.util.*;
2
3  public class UnorderedPQ implements PriorityQueue<Integer> {
4      List<Integer> entries;
5      UnorderedPQ() { entries = new ArrayList<>(); }
6
7      // is this empty?
8      public boolean isEmpty() { return 0==entries.size(); }
9
10     // add elt to the priority queue
11     public void add(Integer elt) { entries.add(elt); }
12
13     // return the maximum value
14     public Integer findMax() {
15         return Collections.max(entries);
16     }
17
18     // delete the maximum value
19     public void removeMax() {
20         int maxValue = Collections.max(entries);
21         entries.remove(maxValue);
22     }
23
24     public int size() { return entries.size(); }
25 }
```

The choice of an `ArrayList` means that appending takes $O(1)$ time, so `add` is an $O(1)$-time operation. But both `findMax` and `removeMax` are implemented via `Collections.max`, which scans the sequence for the maximum element and thus requires $O(n)$ time, where $n$ is the size of the collection. Notice that any correct algorithm will have to carry out this scan because the sequence is arbitrarily ordered. Perhaps, keeping the sequence ordered (i.e., sorted) can help. We will explore this idea next.

### Implementation II: Ordered Sequence

The previous implementation requires $O(n)$ to find the maximum element. We will attempt to make this operation faster. If the location of the maximum were known, we would not have to scan the whole sequence. In the second

implementation, we will experiment with keeping the sequence sorted. With this arrangement, finding the maximum and/or deleting it is easy because we know the maximum element is at the back of the sequence, so reading and popping the back will do. Adding a new element, however, will be more complicated as keeping the sequence sorted means finding the right spot for the new element and inserting it there.

In terms of running time, `findMax` and `removeMax` only take $O(1)$ time now. But if we keep the sequence as an `ArrayList` as before, the running time of `add` is inevitably $O(n)$ in the worst case: the new element might need to be positioned at the front of the sequence, causing all existing elements to shift. It is instructive to convince yourself that changing the underlying storage to a `LinkedList` does not improve the worst-case time bounds either.

### Running Time At A Glance

The table below shows the running times for the two implementations we just discussed and the running times we anticipate for our third implementation. Let $n$ be the number of elements in the priority queue at that moment.

| **Operations** | Unordered List | Ordered List | Binary Heap |
|---|---|---|---|
| `add` | $O(1)$ | $O(n)$ | $O(\log n)$ |
| `findMax` | $O(n)$ | $O(1)$ | $O(1)$ |
| `removeMax` | $O(n)$ | $O(1)$ | $O(\log n)$ |

## 9.3 The Binary Heap Structure

In the previous two implementations, we either keep the collection fully sorted at all times or keep it arbitrarily ordered. But these are the two extremes that either do too much or too little. We strive for a middle ground: *keep the collection partially sorted*—that is, sufficiently sorted to answer our question but without doing too much work.

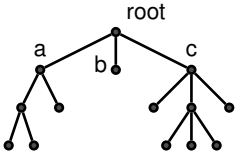
### Trees, Very Briefly

The collection data structures we have seen so far share a common property: they are linear—there is a sense of an absolute position on a line from left to right. Often, this confines our thinking and limits what we can achieve.

It is possible, however, to break this pattern. One of the most important nonlinear structures is the tree structure. We will briefly discuss it to get a feel for what trees are and discuss it in depth in subsequent chapters.

Instead of organizing data linearly, a tree is a hierarchical structure with a few notable properties. Each item in a tree is generically called a node. There is a special top element known as the root. Every node has zero or more children. When a node does not have any children, it is called a leaf. In computer science, we often draw trees with its root node at the top and their descendants in subsequent layers below that.

**An Example Tree.** The tree below has the root at the top.



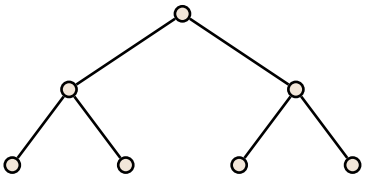

The children of the root are a, b, and c.

For now, we are most interested in binary trees. These are tree structures where each node can have at most *two* children. We will see how to take advantage of such a structure to obtain a fast implementation for the priority queue data type.

## Binary Heap Invariants

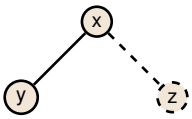
The basic idea is to keep the items sorted enough so that the maximum item can be easily discovered but at the same time, it must not be too rigid so that each add operation only has to move a few things around.

To accomplish this, we are keeping a binary tree of a special kind. Remember that a binary tree is a tree where each parent can have at most 2 children. Below, we give an example of a binary tree. This is a *perfect binary tree*—a binary tree where every node except the leaves has two children and the leaves are all at the same level.



However, a binary tree is not always perfect, but in general, one hopes to get close to the shape of the perfect tree.

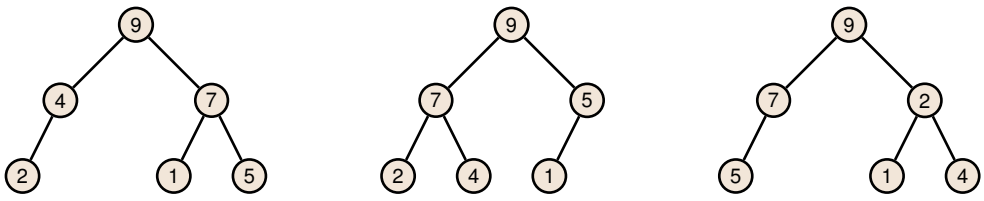
**Heap Ordering.** If $x$, $y$ and $z$ are the priorities at the respective nodes, then the invariant says that $x \geqslant y$ and, if existed, $x \geqslant z$.




**Definition 9.1** (Binary Heap Tree)**.** A *binary heap tree* is a binary tree that maintains the following two invariants:

- **(I1)** The priority of an element is greater than or equal to the priorities of its children. (Among the children any order is fine.)
- **(I2)** The tree is *complete*—full at all levels except potentially at bottom level (the leaves) where it must be left-aligned.

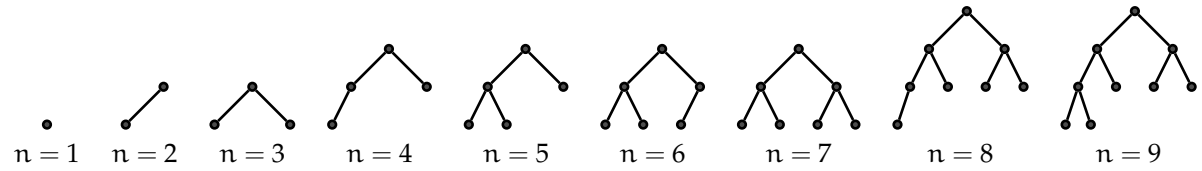
Let us examine these invariants in turn: The first invariant guarantees, among other things, that the root of the tree (the top node) always has the max element in the collection. As we will soon see, it further guarantees that adding an item or deleting the max will never cost more than the number of levels the heap tree has. The following trees demonstrate the first invariant (or the lack thereof); they do not necessarily respect the second invariant.



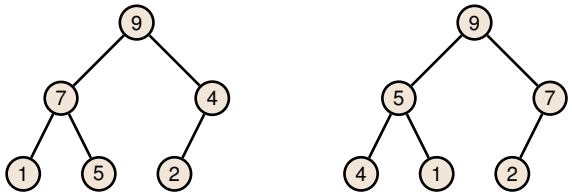

All three figures have the same set of elements. The left and the middle figures show examples of trees that respect the first invariant, but only the middle

also satisfies the second invariant. The right figure violates the first invariant because although 2 is bigger than 1, it is smaller than 4. Notice that as we made an observation earlier, the maximum element is necessarily at the top.

The second invariant controls the shape of the tree and effectively the height. By indicating that the tree must be full at all levels except the last, we know that the shape of the tree is completely determined by the number of elements—and not at all by what the data items are. For example, the following figure shows the shapes for $n = 1, 2, 3, \ldots, 9$.




Below are two examples of binary heap trees (i.e, they satisfy both the invariants) on the set of elements $\{1, 2, 4, 5, 7, 9\}$:




## 9.4 Operations On The Binary Heap

To satisfy the priority queue interface, we aim to support the following main operations: `findMax`, `add`, and `removeMax`. Throughout, the binary heap is a maximum-oriented heap—that is, the root has the largest element. We will discuss these operations in turn.

### `findMax`: Finding The Maximum Element

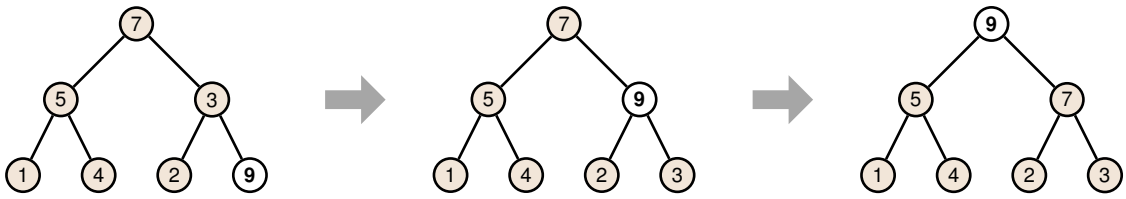
By the binary heap invariant, the maximum element is located at the root of the binary heap tree. This can be shown inductively. Thus, `findMax` can simply return the element at the top of the tree (aka. the root). This encourages a data storage format where the root is easily accessible.

### `add`: Inserting Into A Heap

We have established that the shape of a heap tree is completely determined by the number of nodes. Therefore, there is no question what shape the new tree is going to take. Specifically, because the tree is filled level by level, the new tree takes the shape of the old tree plus a new node attached to the bottom-right end. However, we cannot simply put the new item there since

it will violate the heap ordering invariant. The more involved question is, therefore, *how we can add the new item and maintain the heap ordering invariant?*

**Idea:** We will put the new element at the bottom-right node anyway and fix what is broken. To understand how we might fix the tree, we will look at an example below (the labels inside the nodes are their priorities).




On the left-most panel, we add 9 to an existing heap at the bottom-right position. Placing that 9 there violates the heap ordering property because 9 is larger 3, which at the moment is the parent of 9. We can easily fix this: swap 9 and 3. The resulting tree is shown in the middle figure. Still, the heap ordering property remains violated—9 is bigger than 7, which is 9's parent. Once again, we swap them, resulting in the tree on the right. At this point, we have fully restored the heap ordering invariant.

In general, notice that we can only violate the heap ordering invariant where the new element is, and as we exchange the new item and its current parent, it "swims" up the tree, potentially creating another location where the invariant is violated. However, because the tree is untouched anywhere else, the violation is contained locally. We summarize this process as follows.

**Promotion in a binary heap:** As the priority of a node becomes larger than its parent's priority, we can fix this in a few simple steps, as outlined by the pseudocode below:

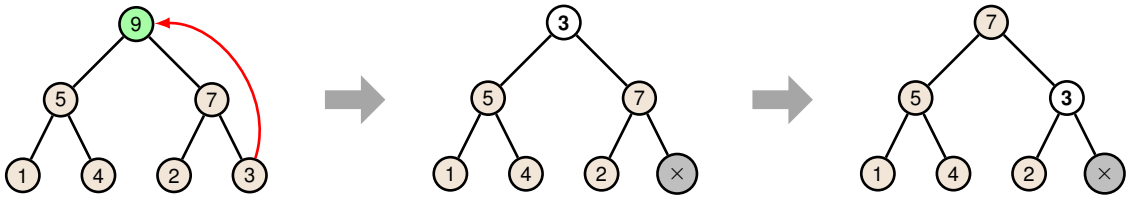
```
def swim(k):
    while not is_root(k) and p(parent_of(k)) < p(k):
        swap(parent_of(k), k)
        k = parent_of(k)
```

### `removeMax`: Deleting The Maximum Element

We know already that the max element lives at the top of the tree (i.e., the root). To delete the maximum element, we will remove the item at the root, but then, we will also need to fix up the tree—if we simply remove the root, what is left is not really a tree. To fix this, we can take hints from the heap invariants: because the shape of the tree is completely determined by its size, we know what the tree must look like after we are done—exactly like before except with the bottom-right node removed.

**Idea:** Take the node at the bottom-right, place it in place of the root (which we wish to delete), and fix what is broken.

Similar to the insertion case, we check if the node we place at the top violates the heap ordering—if it does, we swap it with the larger of its two children, causing this node to "sink" down the tree. Heap-ordering violation




may still take place, and if it does, it will follow where we sent the new root down. But this eventually has to stop because it will sink to the bottom of the tree or stop if the violation has all been eliminated. We summarize this process as follows.

**Demotion in a binary heap:** As the priority of a node becomes smaller than one or both of its children's priorities, we can fix this in a few simple steps, as outlined by the pseudocode below:

```
def sink(k):
    while not is_leaf(k):
        max_child = the child of k having a larger priority
        if p(k) >= p(max_child): break
        swap(k, max_child)
        k = max_child
```

### Quick Note on Running Time

We cannot yet talk about the actual running time because we have not specified how the tree is to be represented. But it is worth pointing out that for `add` and `removeMax`, the cost, in the worst case, is proportional to the number of levels that the tree has (i.e., we have to go the whole length of the path from top to bottom or vice versa). Hence, if the heap tree is $d$-level deep, `add` and `removeMax` will each cost $O(d)$ time.

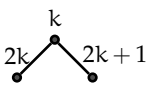
## 9.5 Storing The Binary Heap Tree In An Array

How can we economically and efficiently store the binary heap tree? It turns out the binary heap tree invariants give us an elegant way of mapping the tree's nodes into locations in an array.

To define this mapping, we will label nodes in level order. That is to say, the root at the top of the tree is associated with location 1, and we recursively define the mapping as follows:

> *If a node is associated with location* $k$*, its left child is at location* $2k$ *and its right child is at* $2k+1$.

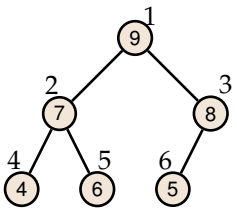
**Recursive Label Mapping.** The left and right children of a node at location $k$ are at locations $2k$ and $2k+1$, respectively.




This relationship allows easy navigation between nodes as indicated by the following code:

```
int parentOf(int k) { return k/2; }
int leftOf(int k) { return 2*k; }
int rightOf(int k) { return 2*k + 1;}
```

This means the following tree shape is mapped onto an array using the number next to each node, and hence the keys are stored in the array as shown on the right:




| *index* | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| entries | ∅ | 9 | 7 | 8 | 4 | 6 | 5 |

**Tree Depth.** The deepest node is at location $n$ and the root is at location 1. Now the parent of $n$ is at location $\lfloor n/2 \rfloor$, whose parent is at location $\lfloor \lfloor n/2 \rfloor /2 \rfloor$. Repeating this argument gives that the depth of the tree is at most $\log_2(n)$.

One thing falls out nicely from this discussion. Using the parent function and noticing the node with the largest index has index $n$, we can find the depth of any binary heap tree on $n$ nodes.

**Lemma 9.2.** A heap tree with $n$ nodes can be at most $\log_2(n)$ deep.

## Implementation in Java

We define a class `BinaryHeap` that has a generic type parameter `T`; however, this type parameter needs to be constrained so that elements of this type can always be compared, like so:

```
public class BinaryHeap<T extends Comparable<? super T>>
    implements PriorityQueue<T>
```

Instead of implementing a resizable array ourselves, we resort to Java's built-in `ArrayList` to store our binary heap tree's elements. Hence, we store our elements and initialize it as follows:

```
private List<T> entries; // a resizable array for elements

public BinaryHeap() {
    entries=new ArrayList<>();
    entries.add(null); // dummy value
}
```

It helps to define a number of utility methods to help with navigation and simplifying code:

```
public int size() { return entries.size()-1; }
public boolean isEmpty() { return 0==size(); }

private void swp(int i, int j) {
    Collections.swap(entries, i, j);
}
private int compare(int i, int j) {
    return entries.get(i).compareTo(entries.get(j));
}
```

**`findMax`.** With this setup in place, finding the maximum element is simple:

```
public T findMax() { return entries.get(1); }
```

**add and removeMax.** The main workhorse for add is the swim maneuver:

```
private void swim(int l) {
    while (l > 1 && compare(parentOf(l), l)<0) {
        swp(parentOf(l), l);
        l = parentOf(l);
    }
}
```

With it, the add operation consists simply of appending a new element and invoking swim, like so:

```
public void add(T e) {
    entries.add(e);
    swim(this.size());
}
```

For removeMax, the main workhorse is the sink maneuver:

```
private int maxIndex(int l) {
    int maxDex = leftOf(l), n = this.size();
    if (rightOf(l) <= n &&
        compare(maxDex, rightOf(l)) < 0) { maxDex = rightOf(l); }
    return maxDex;
}

private void sink(int l) {
    int n = this.size();
    while (leftOf(l) <= n) { // not yet a leaf
        int maxDex = maxIndex(l);
        if (compare(l, maxDex) >= 0) { break; }
        swp(l, maxDex);
        l = maxDex;
    }
}
```

With the heavy lifting done by sink, removeMax is straightforward:

```
public void removeMax() {
    T lastElt = entries.remove(this.size());
    if (!isEmpty()) {
        entries.set(1, lastElt);
        sink(1);
    }
}
```

Altogether, this is a rudimentary implementation of the binary heap data structure, where findMax takes $O(1)$ time, and both add and removeMax require $O(\log n)$ time. The data structure uses $O(n)$ space.

## Java's Built-In `PriorityQueue`

Java has a built-in PriorityQueue implementation. The class is called PriorityQueue, which is part of java.util.PriorityQueue. Unlike ours,

Java's implementation uses a minimum-oriented binary heap. See the documentation for details. For most users and applications, this is the implementation to use. It is convenient and robust, and plays nicely with other Java features. Only rarely will we need to write our own priority queue.

## 9.6 Application: Heap Sort

We have seen several sorting algorithms so far. One direct application of the `PriorityQueue` is sorting. After a quick moment of thought, we can see that if we manage to add all the elements that we wish to sort to a priority queue, removing the maximum the first time will yield the largest element. Doing so the second time will yield the second largest element, and so on. This leads to the following code (we omit the template declaration for type `T` for improved readability):

**Code 9.4**: Basic Heap Sort.

```
1 void heapSort(T[] a) {
2     PriorityQueue<T> pq = new BinaryHeap<>();
3     for (T elt: a) { pq.add(elt); }
4     for (int k=a.length-1;k>=0;k--) {
5         a[k] = pq.findMax();
6         pq.removeMax();
7     }
8 }
```

In this code, the first `for` loop adds all the elements to the priority queue. For an input array of length $n$, this means calling `add` a total of $n$ times, requiring $O(n \log n)$ time overall. The other `for` loop pulls out the current maximum one by one. Because there are $n$ elements, this loop also runs for $n$ iterations. Thus, this loop takes $O(n \log n)$ time. In total, this algorithm runs in $O(n \log n)$ time, which matches the running time of the best comparison-based sorting algorithms we have looked at earlier.

Yet we can, as it turns out, make the first loop faster: although this will not improve the overall asymptotic running time, it is possible in $O(n)$ time to build a binary heap tree with $n$ elements starting from empty and this does translate to improved performance in practice.

### Building The Heap Faster

From an empty binary heap tree, the current method builds the heap by adding the elements one by one, requiring a `swim` operation, which takes $O(\log n)$, per addition. The improved `buildHeap` algorithm will put in all the elements at once and repair the ordering afterwards. To reflect this, we are adding a constructor that takes an array of elements as shown in Code 9.5.

*How can we ensure that the heap is properly ordered?* The trick is to run `sink` on every possible node in the binary heap tree from the bottom. It is important

**Code 9.5**: Alternative binary heap constructor

```
1 public BinaryHeap(T[] initArray) {
2     this(); // initialize the heap as usual
3     for (T elt: initArray) { entries.add(elt); }
4     buildHeap();
5 }
```

to start from the bottom and work our way back up to the root because correctness crucially depends on being able to guarantee that before `sink` is called on the node at location $k$, every node beneath it already respects the heap ordering. Hence, we have the following code:

```
private void buildHeap() {
    int n = this.size();
    for (int k=n/2;k>=1;k--) { sink(k); }
}
```

Notice that location $\lfloor n/2 \rfloor$ is the bottom-most node with at least one child; the `sink` operation has no effects on any locations beyond that.

This construction helps improve the running time because each `sink` operation pushes its value down towards a leaf beneath it, so about 50% of the nodes are one step away from the leaves and cost 1 "push" (compared to about $\log n$ steps required in a comparable `add` and its corresponding `swim` operation). The next 25% of the nodes approximately are two steps away from the leaves and cost 2 "pushes," and so on. More precisely, a `sink` called at location $k$ will consider locations $2^0 \cdot k, 2^1 \cdot k, 2^2 \cdot k, 2^3 \cdot k, \ldots$ until $2^t \cdot k \geqslant n$. This means a running time of $O(\log(n/k))$ for that `sink` call. Hence, the total running time of `buildHeap` is

$$\sum_{k=1}^{\lfloor n/2 \rfloor} \log\left(\frac{n}{k}\right) = O(n) \tag{9.1}$$

(Exercise 9.6. will prove this relationship concretely.)

We conclude that `buildHeap` takes $O(n)$ time and hence, our new constructor takes $O(n)$ time overall. This means our heap sort can be upgraded to take $O(n) + O(n \log n) = O(n \log n)$ time.

## Exercises

**Exercise 9.1.** A number means adding that number to the priority queue and a $ means finding and removing the maximum. Give the sequence of values returned by the executing the following sequence:

```
11, 17, 44, $, 34, 14, $, 33, 42, 3, $, 37, $, $, 1,
$, 5, $
```

**Exercise 9.2.** Draw the shape (i.e., without the actual data) of the binary heap tree on $n = 19$ keys.

**Exercise 9.3.** Draw all possible binary heap trees on the set of keys $\{h, e, a, p, i, y\}$. How would each of the tree you have drawn map to an array representation?

**Exercise 9.4.** A student came up with the following idea for a priority queue implementation: we only need to keep the maximum number we have seen so far and update it as new numbers are added. This way `add` will take $O(1)$ time and so is `findMax`. Hence, it is possible to support priority queue operations all in constant time and space. Criticize whether this implementation is feasible.

**Exercise 9.5.** Write out a proof of Lemma 9.2 in detail.

**Exercise 9.6.** Prove Equation (9.1). (*Hint:* There are about $n/4$ values of $k$ between $n/4$ and $n/2$ and for these values of $k$, $\log(n/k) \leqslant \log 4$. Continuing this argument gives

$$\sum_{k=1}^{\lfloor n/2 \rfloor} \log_2(n/k) = O\left(\sum_{\ell=1}^{\log n} \frac{n}{2^\ell} \cdot \ell\right).$$

It should not be too difficult to argue that this summation converges to $O(n)$ even if $\ell \to \infty$. )

**Exercise 9.7.** How would you implement a stack using the priority queue as the underlying data structure? How about a (FIFO) queue?

**Exercise 9.8.** In a maximum-oriented binary heap tree (as discussed in this chapter), where can we find the minimum-priority element? If, in addition to the current operations, we wish to add support for `findMin`, what is the best running time we can obtain using a maximum-oriented binary heap tree?

**Exercise 9.9.** The motivating example at the beginning of the chapter seeks to maintain the smallest $k$ numbers from a long list of numbers. Update the example Java implementation to use the built-in Java's `PriorityQueue` to keep the running time per new number to $O(\log k)$. When you complete this exercise, the overall running time should be improved to $O(n \log k)$.

**Exercise 9.10.** You have $k$ `LinkedList<Integer>` instances, each of which is ordered from small to large, though there may be duplicates. Your goal is to merge these lists to create a combined sorted list. Using a suitable priority queue data structure, you will implement a function

```
LinkedList<Integer> mergeAll(LinkedList<Integer>[] lists)
```

that takes an array of `LinkedList<Integer>`'s and returns a `LinkedList<Integer>` that is the combined list in sorted order (small to large).

Your algorithm must run in $O(N \log k)$, where $N$ is the sum of the list lengths and $k = \texttt{lists.length}$. Keep in mind, accessing (i.e., reading from or writing to) a linked list is cheap at the front and the back of the list; however, accessing it anywhere else is generally expensive. The cost is proportional to how far it is from the end.

## Chapter Notes

The priority queue ADT described here has many possible efficient implementations. The binary heap data structure was invented by J.W.J. Williams [Wil64] to implement a sorting algorithm known as heap sort. The more efficient $O(n)$-time heap-building technique described in this chapter is due to Floyd [Flo64]. There are faster implementations and implementations with different time-complexity tradeoffs. Examples include the binomial heap and Fibonacci heap; see, for example, Cormen et al. [Cor+09] for further detail. Using a different design, priority queues such as the leftist heap can be quickly merged, for example, in logarithmic time. For special types of keys, faster implementations are possible. An example is the van Emde Boas priority queue [Boa77] for integer-only keys.