Sergio E. Garcia Tapia
Algorithms by Sedgewick and Wayne (4th edition) [SW11]
November 2nd, 2024

2.4: Priority Queues

Exercise 1. Suppose that the sequence `P R I O * R * * I * T * Y * * * Q U E * * * U * E` (where a letter means *insert* and an asterisk means *remove the maximum*) is applied to an initially empty priority queue. Give the sequence of letters returned by the *remove the maximum* operations.

Solution.

```
P
P R
P R I
P R I O
P I O // max removed: R
P I O R
P I O // max removed: R
I O // max removed: P
I O I
I I // max removed: O
I I T
I I // max removed: T
I I Y
I I // max removed: Y
I // max removed: I
// max removed: I
Q
Q U
Q U E
Q E // max removed: U
E // max removed: Q
// max removed: E
U
// max removed: U
E
```

At the end, `E` remains on the queue. The sequence letters returned is:

`R R P O T Y I I U Q E U`

Exercise 2. Criticize the following idea: To implement *find the maximum* in constant time, why not use a stack or a queue, but keep track of the maximum value inserted so far, then return that value for *find the maximum*.

Solution. One issue is that this only guarantees that the first *find the maximum* operation can be returned in constant time. Once that items is removed, if the client then asks for the next value, this operation then requires linear time.

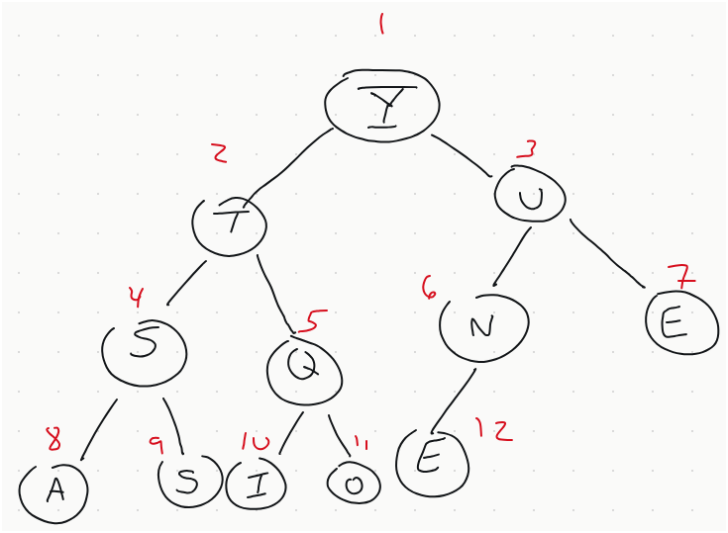

Figure 1: Priority queue built out of keys `E A S Y Q U E S T I O N` for Exercise 2.4.5

Exercise 3. Provide priority-queue implementations that support *insert* and *remove the maximum*, one for each of the following underlying data structures: unordered array, ordered array, unordered linked list, and ordered linked list. Give a table of the worst-case bounds for each operation for each of your four implementations.

Solution. See package `com.segarciat.algs4.ch2.sec4.ex03`. The time complexities for the two main operations are given below for priority queue with n items:

	Insert	Remove the maximum
`UnorderedArrayMaxPQ`	$O(1)$	$O(n)$
`OrderedArrayMaxPQ`	$O(n)$	$O(1)$
`UnorderedListMaxPQ`	$O(1)$	$O(n)$
`OrderedListMaxPQ`	$O(n)$	$O(1)$

Exercise 4. Is an array that is sorted in decreasing order a max-oriented heap?

Solution. If we take entry `a[k]`, then it will certainly be larger than `a[2*k]` and `a[2*k + 1]`. Therefore if we see it as a binary tree, the binary tree is heap-ordered. In short, yes it is.

Exercise 5. Give the heap that results when the keys `E A S Y Q U E S T I O N` are inserted in that ordered into an initially empty max-oriented heap.

Solution. See Figure 1.

The contents of the corresponding array would be:

```
[ null, "Y", "T", "U", "S", "Q", "N", "E", "A", "S", "I", "O", "E" ]
```

Exercise 6. Using the conventions of Exercise 2.4.1, give the sequence of heaps produced when the operations `P R I O * R * * I * T * Y * * * Q U E * * * U * E` are performed on an initially empty max-oriented heap.

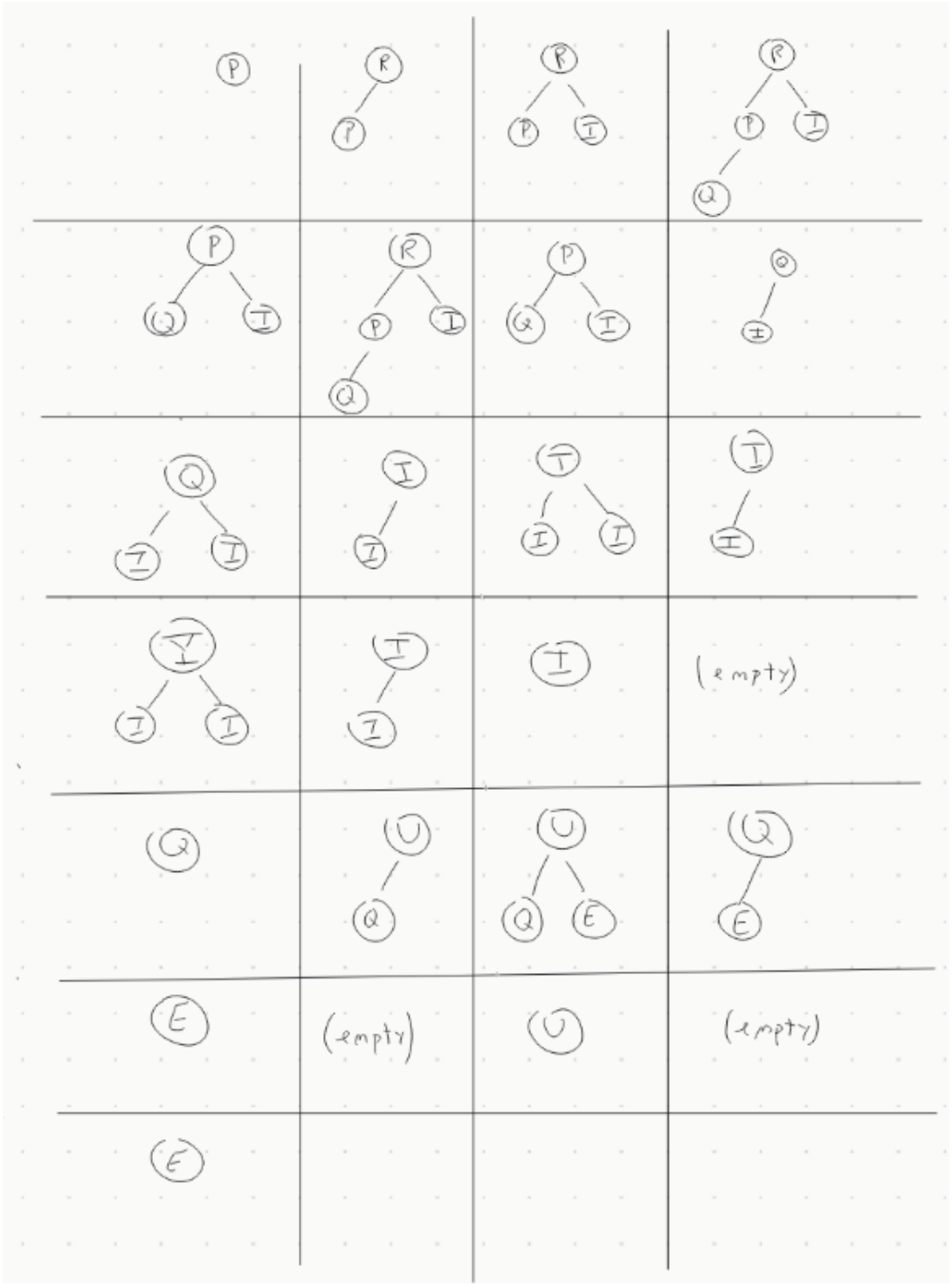

Figure 2: Sequence of priority queue for Exercise 2.4.6

Solution. See Figure 2.

Exercise 7. The largest item in a heap must appear in position `1`, and the second largest must be in position `2` or `3`. Give the list of positions in a heap of size 31 where the kth largest:

(i) can appear, and

(ii) cannot appear,

for $k = 2, 3, 4$ (assuming the values to be distinct).

Solution.

(i) Interpreting k to start at 0, where 0th largest means largest, 1th largest means 2nd largest, now we are considering $k = 2$, which is the third largest.

- $k = 2$ (third largest): Position 1 is taken by the largest. The second largest is at position 2 or position 3. The third largest could be at position 2 or 3 as well (whichever one is not occupied by the second largest). In this case, it would be a child of the largest. Otherwise, it could be a child of the second largest. If the second largest is at position 2 and the third largest was it child, then it could be at positions 4 or 5. If the second largest is at position 3 and the third largest is its child, the it could be at position 6 or 7.

  Altogether, third largest could be in positions $\{2, 3, 4, 5, 6, 7\}$.

- $k = 3$ (fourth largest): This item could be a child of the largest, allowing for it to be at positions 2 or 3. This happens if the third largest is a child of the second largest.

  If the fourth largest is not a child of the largest, it could be a child of the second largest. Hence this would allow for positions 4, 5, 6, 7. Instead, if the fourth largest was a child of the third largest, then we have a few cases. If the 3rd largest was at positions 2 and 3, then we get no new information, since that would suggest positions 4,5,6,7 for the fourth largest, which we already know.

  However, the third largest may also be at positions $4, 5, 6, 7$. If the the fourth largest is a child of the third, then it could be at a position that is a child of those positions. These are $8, 9, 10, 11, 12, 13, 14, 15$.

  Hence, the fourth largest may appear at positions numbered in

  $$\{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$$

- $k = 4$ (fifth largest): The fifth largest could be at positions 2 and 3. Again, this would, for example if the second largest is at position 2, and the third and fourth largest are its children.

  Similarly, the fifth largest could be at positions 4,5,6,7. This could happen, for example, if the second and third largest fill up positions 2 and 3.

  Next, if the second and third largest fill up positions 2 and 3, and if the fourth largest fills up one of the positions 4,5,6,7, then by making the fifth largest a child of the fourth largest, we could have it fill positions $8, 9, 10, 11, 12, 13, 14, 15$.

One last possibility is if the second is a child of the first, the third is a child of the second, and the fourth is a child of the third. If we now make the fifth largest a child of the fourth, then it could take on any integer position between 16 and 31, inclusive.

Hence, the fifth largest could be anywhere between positions 2 and 31, inclusive.

(ii) The places they may not appear are precisely the complement of the locations I found.

Exercise 8. Answer the previous exercise for the kth *smallest* item.

Solution. Consider first where the smallest may be. The smallest cannot have a child. Thus it must be in positions 16 through 31, inclusive. The second smallest must also be at positions 16 through 31 because even though it may have the smallest key as its child, it cannot have any other key as its child. Since the tree has 31 nodes, it is a perfect tree, so every internal node has two children. Since the second smallest cannot have two children, it follows that it cannot be an internal node.

(i)
- $k = 2$ (third smallest): The third smallest may be in any of positions 16 through 31 as before. Since it can have the smallest and second smallest as its children, it could also be at positions 8 through 15. It cannot be at a higher level because that would imply that it is the root of a binary tree with 7 nodes, making it larger than 6 other nodes, and that's not the case.
- $k = 3$ (fourth smallest): By a similar reasoning as $k = 2$, it can only be at positions 8 through 31.
- $k = 4$ (fifth smallest): By a similar reasoning as $k = 2$, it can only be at positions 8 through 31. For example if it was at position 7, then its two children and their children would all be smaller. This is impossible since the fifth smallest can only be larger than four other keys, not six.

(ii) Again just take the complements of the positions previous found.

Exercise 9. Draw all of the different heaps that can be made from the five keys `A B C D E`, then draw all of the different heaps that can be made from the five keys `A A A B B`.

Solution. I made the heaps max-oriented. See Figure 3 and Figure 4.

Exercise 10. Suppose that we wish to avoid wasting one position in a heap-ordered array `pq[]`, putting the largest value in `pq[0]`, its children in `pq[1]` and `pq[2]`, and so forth, proceeding in level order. Where are the parents and children of `pq[k]`?

Solution. The children are at position $2k + 1$ and $2k + 2$. The parent is at $\lfloor \frac{k-1}{2} \rfloor$.

Exercise 11. Suppose that your application will have a huge number of *insert* operations, but only a few *remove the maximum* operations. Which priority-queue implementation do you think would be most effective: heap, unordered array, or ordered array?

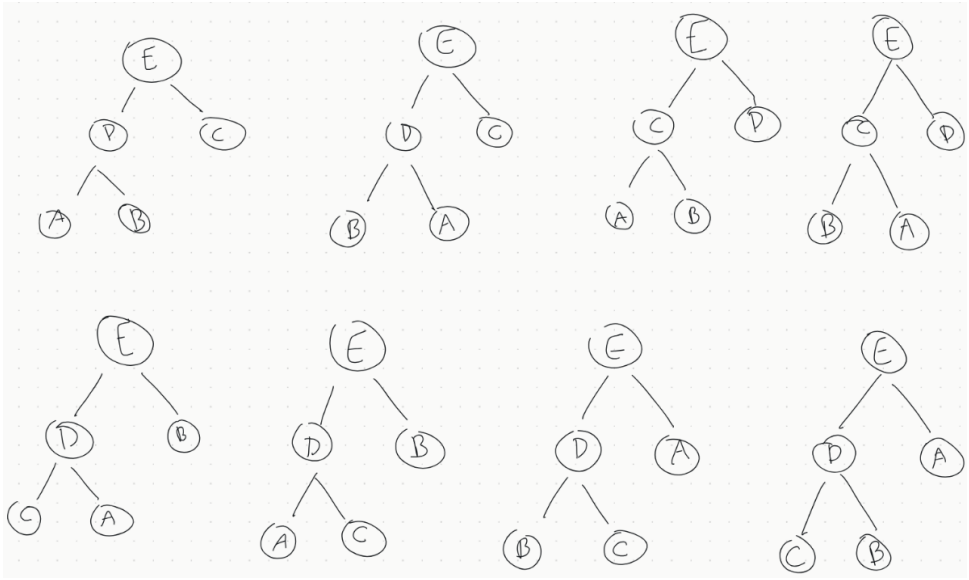

Figure 3: Exercise 2.4.9: All possible max-oriented heap with keys `A, B, C, D, E`

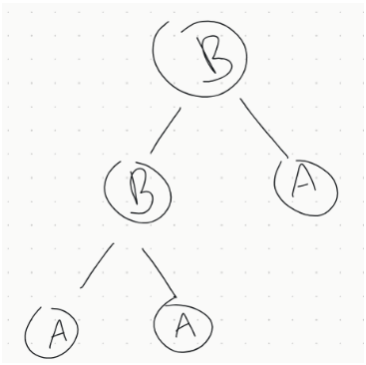

Figure 4: Exercise 2.4.9: All possible max-oriented heap with keys `A, A, A, B, B`

Solution. The worst would be the ordered array implementation because it's expensive to add an item, given that it must maintain the keys in order.

On the one hand, the unordered array implementation seems right for the job because inserts are $O(1)$, even though remove operations are $O(n)$.

Exercise 12. Suppose that your application will have a huge number of *find the maximum* operations, but a relatively small number of *insert* and *remove the maximum* operations. Which priority-queue implementation do you think would be most effective: heap, unordered array, or ordered array?

Solution. The ordered appear implementation requires $O(1)$ for find-the-maximum operations. Even though inserts take $O(n)$, this seems like the best tool for the job.

Exercise 13. Describe a way to avoid the `j < n` test in `sink()`.

Solution. The context in which this test appears is that we are checking to see whether a node is smaller than its largest child. However the largest child could be the right child, and that child could be the last node in the heap. The way the code was presented, not having this test means we could have an `IndexOutOfBoundsException`.

Since this can only happen at the last level, we could eliminate the test altogether, but change the controlling expression of the `while` from `2*k <= n` to `2*k < n`. This way, we guarantee that `2*k + 1 <= n`, so that `less(2k, 2k + 1)` is also valid.

Now if the loop breaks ends because of the `while` controlling expression is `false`, then we know that `2 * k == n`. Thus, we can add an `if`, statement after the loop to check for this condition, and if the key at `k` is smaller than the key at `2 * k` (which equals `n`), then we do the final exchange.

Alternatively, we can check if `n` is even, because only in this case can we have this issue. If so, we can exchange the item at position `k` and the item at position `n`, if `a[k] < a[n]`. Then once again we change the `while` controlling expression to `2*k <= n` to `2*k < n`. Now the test `j < n` can be eliminated.

Exercise 14. What is the minimum number of items that must be exchanged during a *remove the maximum* operation in a heap of size n with no duplicate keys? Give a heap of size 15 for which the minimum is achieved. Answer the same questions for two and three successive *remove the maximum* operations.

Solution. Suppose the heap has a single child at the last level, which can occur if $n = 2^k$ for some integer k. The height of such a tree is k. In a max-oriented binary heap, the root contains the largest key. When a *remove the maximum* operation is issued, the root is exchanged with the last node in the tree, and the new root now sinks through the heap until the correct location for it is found. The number of exchanges is dependent on the value of the key of that last node, but it must be at most $k-1$ because the tree decreases in height in this instance.

Suppose the last node is the $(k+1)$th largest key, where the 0th largest key is the largest key in the heap (located at the root). This could happen, for example, if the second and third largest are children of the root, and all next largest nodes are under the subheap rooted at the third largest. For example, the fourth largest is a child of the third largest, the fifth largest is a child of the fourth, and so on. Since the height of the tree is k, this allows the nth node to be the $(k+1)$th largest. In particular, no node in the

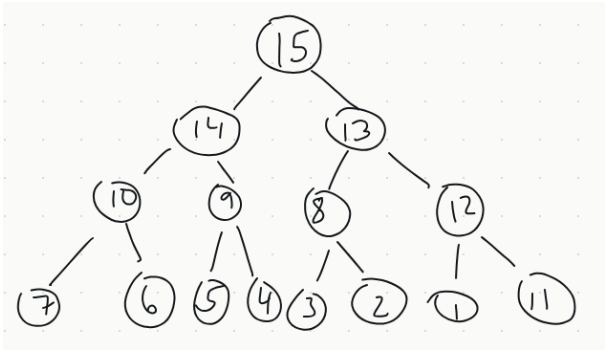

Figure 5: Exercise 14: Heap for which the first *remove the maximum* operation would require 2 exchanges.

subheap rooted at the second largest key is larger than the last node. When the *remove the maximum operation* is issued, the last node becomes the root and then sinks. Since the second largest was a child of the old root, the exchange happens in that direction, so at least 1 exchange happens. However, no more exchange happens, because no node in the subheap that was rooted at the second largest (which is now the largest) was larger than the old $(k+1)$th largest node (now the kth largest node). The conclusion is that only 2 were required; the initial exchange with the old root, and the exchange between the old $(k+1)$th largest and old second largest!

If the tree has size n and height $k = \lfloor n \rfloor$, with $k \neq 2^k$, then a similar argument once again shows that only one exchange is necessary. See Figure 5.

A similar setup leads to a minimum of 6 exchanges when 2 *remove the maximum* operations are issued one after the other. This time, assume $n \neq 2^k$, Suppose that the tree has size n with height $k = \lfloor \lg n \rfloor$, and $n \neq 2^k$. By a similar vein, have the largest at the root, the second and largest as its child. However, this time we make the third largest a child of the second. The other child of the root will be the fourth largest, which in turn is the parent of the fifth largest, and so on. Now the leaf becomes the $(k+2)$th largest instead of the $(k+1)$th as before. Its sibling is the $(k+3)$th largest, and both are children of the kth largest. When the first *remove the max* operation is issued, the $(k+2)$th largest is exchanged with the root, which is then exchanged with the second largest, which is then exchange with the third largest. At that point, no more exchanges occur, because that subheap has the $(k+4)$th largest and beyond. The next *remove the maximum* operation exchanges the what was the second largest before (now at the root) with what was the $(k+3)$th largest before (now the last node). The exchange goes in the direction of what was the third largest before, and then one more exchange occurs because that node has what was the $(k+2)$th largest before as its child. Now the $(k+2)$th largest is a parent of the $(k+3)$th largest. A total of 6 exchanges occurred in this case. See Figure 6.

Exercise 15. Design a linear-time certification algorithm to check whether an array `pq[]` is a min-oriented heap.

Solution. See `com.segarciat.algs4.ch2.sec4.MinPQCertification`.

Exercise 17. Prove that building a minimum-oriented priority queue of size k then doing $n-k$ *replace the minimum* (*insert followed by remove the minimum*) operation leaves the k largest of the n items in the priority queue.

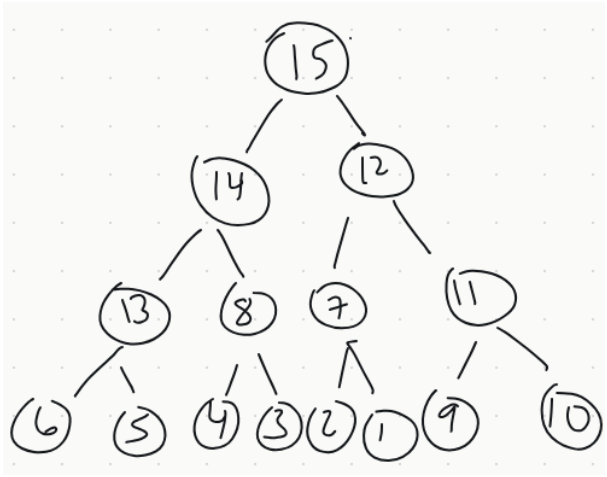

Figure 6: Exercise 14: Heap for which two consecutive *remove the maximum* operations would require 6 exchanges.

Solution.

Proof. Suppose the minimum-oriented priority queue has k elements. Suppose that the k largest keys are $x_1, \ldots, x_k$, where x_1 is the largest, and x_k is the kth largest.

Let x_m be the mth largest, where m is an integer such that $1 \leq m \leq k$. Then there's at most $m-1$ keys larger than x_m, and $m-1 < k$. Suppose that x_m is already on the priority queue, and a *replace the minimum* operation is issued. That is, a new key is added to the queue, so that the queue now has $k+1$ keys. Then x_m is not the minimum, because if it was, then the other k keys would be larger. But that's impossible because at most $m-1 < k$ keys are larger than x_m. Thus, x_m is not eligible for removal, and x_m will remain in the queue. Therefore, if one of the k largest keys is already in the queue, then it remains in the queue. If x_m is not in the queue, but it is inserted, then by the same reasoning, it is not removed from the queue.

Since m is arbitrary with $1 \leq m \leq k$, this argument shows that $x_1, \ldots, x_k$ all remain in the queue after the $n-k$ operations have been issued, because once each goes in the queue, they are never removed. □

Exercise 18. In `MaxPQ`, suppose that a client calls `insert()` with an item that is larger than all items in the queue, and then immediately calls `delMax()`. Assume that there are no duplicate keys. Is the resulting heap identical to the heap as it was before these operations? Answer the same for two `insert()` operations (the first with a key larger than all keys in the queue and the second for a key larger than that one) followed by two `delMax()` operations.

Solution. For a single `insert()` and `delMax()` operation pair, the heap would remain the same. Suppose the root is initially x, and `insert()` is called with a larger key y. Our implementation places y at the end, unconditionally swaps it with its parent each time. Let $a_0, a_1, \ldots, a_k$ be the sequence of items it is swapped with, where $a_k = x$. When the `delMax()` operation is issued, a_0 is swapped with y before y is removed. Next, we `sink()` starting from a_0. We know that x was a child of a_k, and since that was the largest key before y was added, it is the largest key after y is removed. Thus, we swap a_0 and a_k. Next, a_k had a_{k-1} as its child, which is now the child of a_0. Before a_{k-1} was swapped

down a level, it was larger than its two children, and its old child is now the child of a_0 (and hence a_{k-1}'s sibling). Hence, a_{k-1} is the largest among the two, so the `swap()` brings a_{k-1} up a level as a child of a_k. Continuing this way, a_0 arrives back at its old location.

I also think the same happens if two `insert()` operations with larger keys each time are then followed by two `delMax()` operations; the resulting heap would be identical.

Exercise 19. Implement the constructor for `MaxPQ` that takes an array of items as argument using the bottom-up heap construction method described on page 323 in the text.

Solution. See `com.segarciat.algs4.ch2.sec4.ex19.MaxPQ`.

Exercise 22. *Array resizing.* Add array resizing to `MaxPQ`, and prove bounds like those of Proposition Q for array accesses, in an amortized sense.

Solution. I will follow the proof format in Proposition E from Section 1.4:

Proof. For each `insert()` operation that causes the `pq` array to grow, say from size n to $2n$, consider the $n/2 - 1$ operations that most recently caused the priority queue (not the array) to grow to k, for k from size $n/2 + 2$ to n. Each `insert()` takes about $\sim \lg k$ array accesses. The resize from size n to $2n$ takes $4n$ array accesses. If we average these operations, we get

$$\begin{aligned}\frac{4n + \sum_{k=n/2}^{n} \lg k}{\sum_{k=\frac{n}{2}+1}^{n} 1} &\leq \frac{4n + \lg n!}{n/2} \\ &\sim \frac{4n + n \lg n}{n/2} \\ &= 8 + \lg n\end{aligned}$$

Here I approximated $\sum_{k=n/2}^{n} \lg k$ as $n \lg n$ using Stirling's approximation. Therefore the cost of these $n/2 - 1$ `insert()` operations is about $\sim \lg n$. Because the remaining operations are similarly dependent on the tree depth of the binary heap corresponding to the priority queue, the bounds similarly follow. □

Exercise 24. *Priority queue with explicit links.* Implement a priority queue using a heap-ordered binary tree, but use a triply linked structure instead of an array. You will need three links per node: two to traverse down the tree and one to traverse up the tree. Your implementation should guarantee logarithmic running time per operation, even if no maximum priority-queue size is known ahead of time.

Solution. See `com.segarciat.algs4.ch2.sec4.ex24.LinkedMaxPQ`

Exercise 26. *Heap without exchanges.* Because the `exchange()` primitive is used in the `sink()` and `swim()` operations, the items are loaded and stored twice as often as necessary. Give more efficient implementations to avoid this inefficiency, a la insertion sort (see Exercise 2.1.25).

Solution. See `com.segarciat.algs4.ch2.sec4.ex.26.MaxPQ`.

Exercise 27. *Find the minimum.* Add a `min()` method to `MaxPQ`. Your implementation should use constant time and constant extra space.

Solution. Simple. We add a private field `minKey`. On a call to `insert()`, we decide if we should update `minKey` as follows: if the queue is empty, then the key we are inserting is the minimum; otherwise, if the key being added is smaller, than we update `minKey` to it. On a call to `delMax()`, we set `minKey` to `null` if the queue becomes empty, since it would be the last key removed from the queue.

References

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th ed. Addison-Wesley, 2011. ISBN: 9780321573513.