Lab 4: Queues and Priority Queues

November 5, 2016

Contents

Objectives

By the end of this lab, you will:

- understand what a queue is and how it differs from a list.
- be able to implement queues using lists.
- understand priority queues.
- write an efficient implementation of priority queues using heaps.

Note: Much of this lab is adapted from *Programming and Programming Languages (2016): Chapter 13, Halloween Analysis.*

1 What Is a Queue?

We have already seen lists and sets. Now let's consider another fundamental computer science data structure: the **queue**. A queue is a linear, ordered data structure, just like a list; however, the set of operations they offer is different. In a list, the traditional operations follow a last-in, first-out discipline: `.first` returns the element most recently `link`ed. The first element that we link to a list is the one that is hardest to get back, since we must traverse the entire list. In contrast, a queue follows a first-in, first-out discipline. The first element that is added to a queue will be the easiest to get back.

Let's introduce a pair of images that will help you think about the difference between lists and queues. A list can be thought of as a stack of heavy blocks. Adding or removing blocks from the middle or the bottm of the stack is hard, because we have to lift up the blocks on top first. With that in mind, we add and remove blocks only at the top of the stack. So at any point in time, the easiest block to remove is the last one that was added. The last in is the first out. A queue, on the other hand, can be thought of as the line at TA hours: the first people to enter the line are the first to receive help. The first in is the first out.

The two primary operations that we want to support with a queue are `enqueue`, which will add an element to the queue, and `dequeue`, which will take the next element off the queue, giving us both that element and the resulting shorter queue. Let's lay those down more precisely with some Pyret code.

```
data Queue<T>:
    # we'll figure out this definition later
end

fun enqueue<T>(q :: Queue<T>, elt :: T) -> Queue<T>

data Dequeued<T>:
    | none-left
    | elt-and-q(e :: T, r :: Queue<T>)
end

fun dequeue<T>(q :: Queue<T>) -> Dequeued<T>
```

Notice that because we want `dequeue` to give us two different pieces of information, we defined a new datatype, `Dequeued`, with those two parts. `none-left` will be the output of `dequeue` when the input is an empty queue. `elt-and-q` for queues is similar to `link` for lists, in that we can use it to operate on the first-in and rest of the queue.

2 Implementing Queues

2.1 A Naive Implementation

Task: With your lab partner, come up with two different ways to implement the queue operations using a single list to represent a queue. For each of the two ways, give the big-O set for the worst-case runtime for each function in terms of the number of elements in the queue. There is no need to write any code for this problem, but you may do so if it helps you. (Hint: Be careful with the order of the list. Make sure that your implementation is actually a queue, following a first-in, first-out discipline.)

⋆ **Before continuing, call over a TA to check your work so far.**

2.2 A More Efficient Implementation

As you may have guessed, we can do better. Observe that if we represent the queue as a list, with the most recently enqueued element first, then enqueuing is cheap (constant time). In contrast, if we represent the queue as a list in the reverse order, then dequeuing is constant time. That is why it makes sense to use *two* lists: one to which enqueuing takes constant time, and another, in reverse order, from which dequeuing takes constant time. We will call those two lists `enqueue-to` and `dequeue-from`, respectively.

Task: Use this stencil code to fill in the data definition for Queue.

Question: Keeping all of the elements in both `enqueue-to` and `dequeue-from` is not time-efficient. Why?

Answer: If we do that, then we would need to update `enqueue-to` when we dequeue and `dequeue-from` when we enqueue, both of which will always take time linear in the number of elements in the queue. That's worse than using just one list!

So, we've learned that we absolutely can't enqueue elements onto `dequeue-from` or dequeue elements from `enqueue-to` if we want to have an efficient implementation of queues. So, in order to support enqueue and dequeue, we *must* enqueue only onto `enqueue-to` and dequeue only from `dequeue-from`.

Question: What problem are we going to run into if we only enqueue to `enqueue-to` and dequeue from `dequeue-from` without doing anything else?

Answer: When we enqueue elements, they will be put on `enqueue-to`, but when we try to dequeue those same elements, they will still be in `enqueue-to`, so we won't be able to dequeue them from `dequeue-from` as we would like to.

In order to solve this problem, we need to move elements from `enqueue-to` to `dequeue-from` at some point, by calling a function that we'll name `requeue`.

Task: In the same stencil code as before, implement `requeue`.

Task: *Analyze* the worst-case runtime of `requeue` as a function of the number of elements in the queue.

⋆ **Before continuing, call over a TA to check your work so far.**

This seems like bad news, since we can't requeue in constant time if we want to maintain the first-in, first-out order of the queue. However, we're still better off than we were in the naive implementation. In the naive implementation, one of operations took a linear amount of time in the number of elements *every time* we called it. By contrast, if we're careful about when we use requeue when implementing enqueue and dequeue, then when the queue is large, we won't have to call `requeue` very often at all.

Task: Implement `enqueue` and `dequeue` in the same stencil as before. Determine the big-O set of the worst-case runtime of each function. Note that we are not asking for a formal analysis.

Note: You will likely find that your conclusion above is unsatisfying, particularly because the worst case is predictably rare. In lab 6, we will see a form of runtime analysis, called amortized analysis, that is much more informative in this case.

⋆ **Before continuing, call over a TA to check your work so far.**

3 What Is a Priority Queue?

So far, we've seen that lists are first-in, last-out; and queues are first-in, first-out. But what if we want the next element out to be based on an arbitrary ordering of elements? Suppose, for example, that we have a collection of numbers, and we want to be able to easily take out the minimum number at any point in time. Then, we use a **priority queue**.

You can think of a priority queue as a queue, except that each element is associated with a priority number. The purpose of the priority number is that when we "dequeue" a priority queue, we take the element with the highest priority (not necessarily the one that was enqueued first). The main operations that we want a priority queue to support are `insert-with-priority`, which is analogous to `enqueue`, and `get-highest-priority` and `remove-highest-priority`, which are jointly analogous to `dequeue`. They are written in Pyret code below:

```
data PriorityQueue<T>:
    # one of many possible data definitions (more on this later)
end

fun insert-with-priority<T>(elt :: T, p :: Number,
    pq :: PriorityQueue<T>) -> PriorityQueue<T>:
  doc: "Inserts elt into pq with priority p"
end

fun get-highest-priority<T>(pq :: PriorityQueue<T>) -> T:
  doc: "Produces the element in pq with the highest associated prioirty."
end

fun remove-highest-priority<T>(pq :: PriorityQueue<T>) -> PriorityQueue<T>:
  doc: ‘‘‘Produces a Priority Queue like pq but with the element with the
       highest priority removed.‘‘‘
end
```

4 Implementing Priority Queues

Before we continue, let's simplify things a bit. In what follows, we consider only a special case of priority queues (PQs), with the following specifications:

1. The PQ contains only numbers
2. The priority of each number in the PQ is the number itself multiplied by -1 (so the minimum number has the highest priority)

In this special case, we will give the operations above more appropriate names: `insert`, `get-min`, and `remove-min`, respectively. It will be less messy to consider just this special case, and once you understand how to implement it, generalizing it should be fairly easy.

4.1 Sorted List

Our PQ could be a list, sorted in ascending order.

Task: How could we write `insert`, `get-min`, and `remove-min` using a sorted list? Determine the big-O set of the worst-case runtime of each function as a function of the number of elements in the PQ.

Note: Generally, the list would be sorted in descending order of priority.

4.2 Heap

The sorted list grants extremely efficient removal, but it is suboptimal when it comes to insertion. If we would like our PQ to perform sequences of `insert` *and* `remove-min`, we should instead use a balanced binary **heap**, which guarantees logarithmic time in the number of elements for both.

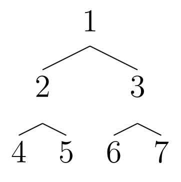
4.2.1 Background

For our purposes, a balanced binary heap is a binary tree that has the following characteristics:

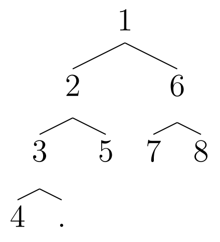
1. The nodes contain numbers.
2. (Order condition) Each node's number is less than or equal to the numbers of its children. This means that the minimum number is at the top of the heap.
3. (Balance condition) Each node's left subtree contains either the same number of nodes or one node more than the right subtree.

Task: For each of the trees below, state whether it satisfies a) the order condition and b) the balance condition. Note that "." is simply a placeholder for the empty node: it indicates that there is nothing there.

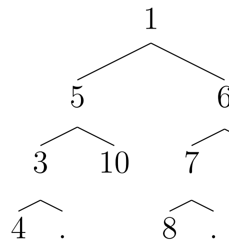
1.



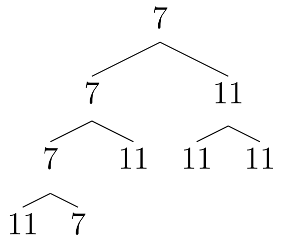
2.



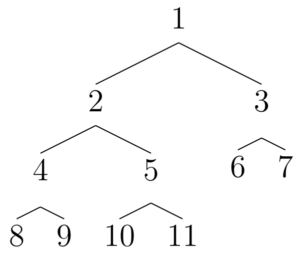
3.



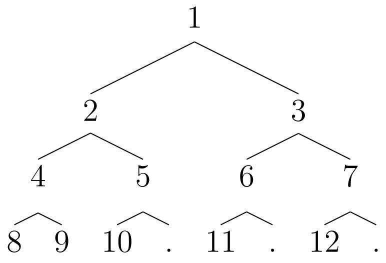
4.



5.



6.



Note: A more general heap definition allows the nodes to contain any comparable values, not necessarily numbers.

Note: These are called min heaps, because the minimum is at the top. As you may have guessed, it is also possible to have analogous max heaps.

Note: The balance condition that you see here is slightly different from other heap balance conditions you may see in other contexts.

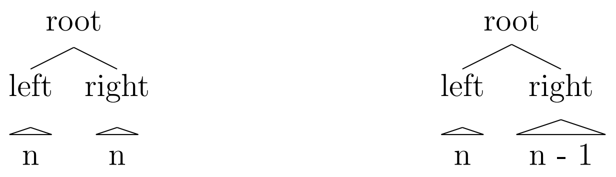
⋆ **Before continuing, call over a TA to check your work so far.**

Here's a stencil code file. We've given you the code for `get-min` and `remove-min`. You're going to implement `insert`.

4.2.2 insert

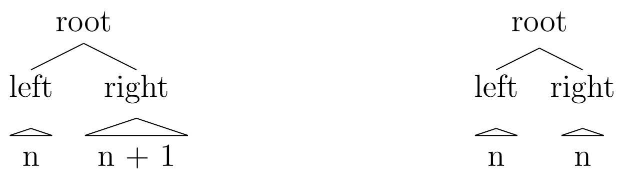
As you might imagine, this is going to be a structurally recursive procedure on the input heap, in which the number to be inserted descends down the heap until it finds its place. The tricky part of this is maintaining the balance invariant in logarithmic time in the number of nodes. Indeed, you may be thinking to yourself, "In order to know whether I should insert into the left or right subheap, I need to know how many nodes are in each so that my insertion doesn't result in an imbalance. And in order to know the number of nodes in each subheap, I have to recur through the entire heap, which will take linear time in the number of nodes..." It is clear from this thought that we can't decide the subheap in which to insert by recurring through them. We must decide *immediately* where to insert. How can we possibly do that, considering that we have such limited information about the input heap?

One of the few things that we do know is vital: the input heap satisfies the balance condition. That means that it takes one of the two forms below:



In words, if n is the number of nodes in the entire left subheap, then the number of nodes in the entire right subheap will be either n or $n - 1$.

Now, consider what this same picture would look like if we insert a number into the right half. We are again in one of two cases:



We're not done yet. The case on the left does not meet the balance condition, and we still don't know which of the two cases we're in. But there's still something *fast* that we can do to resolve the issue.

Task: What single thing can we do that will reestablish balance if we're in the case on the left or maintain balance if we're in the case on the right? Call over a TA to help if you haven't figured it out after a minute of thinking.

Once you have the insight, keep in mind that an imbalance could occur anywhere along the path of right children, so you'll have to rebalance recursively down the right. We'll leave it to you to argue that this doesn't put us over-budget for runtime.

Task: Why did we insert into the right subheap instead of the left?

⋆ **Before continuing, call over a TA to check your work so far.**

Task: Now that you've figured out the hardest part, implement `insert` in the stencil code given. We haven't told you how to do everything, so you'll have to fill in some gaps. Note that we've provided an oracle to check whether your code is correct, but you should still write test cases of your own.

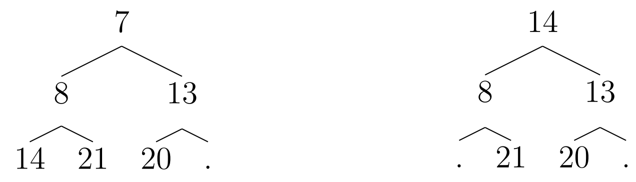
Task: Argue that your implementation of `insert` takes logarithmic time in the number of nodes in the heap in the worst case.

⋆ **Before continuing, call over a TA to check your work so far.**

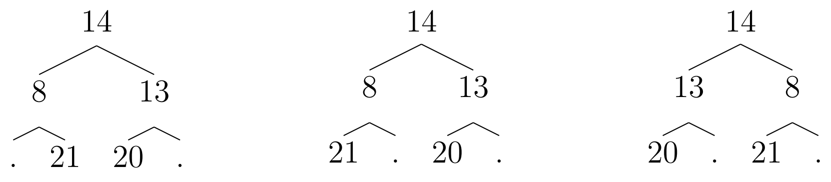
4.2.3 remove-min

We've given you the code of a working solution for `remove-min`. Your goal in this section is to understand it. Here is a description of the procedure.

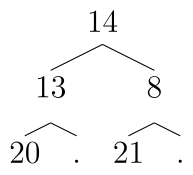
1. Find the leftmost node at the bottom of the heap. Remove it, and store its value (see `amputate-bottom-left` in the solution code). Replace the minimum of the heap, found at the very top, with this value. For example, performing this step on the heap on the left produces the (improper) heap on the right.

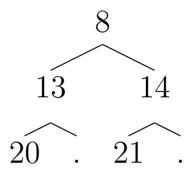


2. Rebalance the heap by swapping the children of all nodes on the leftmost path, effectively starting from the bottom and working your way to the top (see `rebalance` in the solution code). Performing this step on the left heap below would produce the one on the right, with the intermediate step shown in the middle.



3. Reorder the heap by "sifting down" the new value at the top. Recursively swap it with the smaller of the two children until it has reached the bottom of the heap or all of the values below it are larger (see `reorder` in the solution code). In the example that we've been using, we just need to swap that top value, 14, once.





Task: Did it matter that we chose to remove the bottom-left node, as opposed to other nodes? If not, what other node(s) could we have removed? If so, why?

Task: To make sure that you understand the `remove-min` procedure that we supplied, write a few interesting, diverse, implementation-specific test cases for it. The input heaps on which you test should have unique structures.

⋆ **Once a TA signs off on your work, you've finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you've just written.**