(Provisional) Lecture 26: Queues, Amortized Analysis
10:00 AM, Nov 4, 2019

Contents

1 Notes on ADTs 1
   1.1 Invariants .................................................. 1

2 Queues 2
   2.1 A First Attempt: A Single Queue .................. 3
   2.2 A Second Attempt: A Double-list Queue ............ 5
   2.3 Plan B, or is it C or D, by now? .................. 7

3 Amortized Analysis 8
   3.1 Example 1 .................................................. 8
   3.2 Example 2 .................................................. 9

4 Summary 9

Objectives

By the end of this lecture, you will know:

• some of the benefits of using ADTs
• what queues are

By the end of this lecture, you will be able to:

• implement queues efficiently(!) as an abstract data type

1 Notes on ADTs

Abstract Data Types (ADTs) are useful because of the level of abstraction they give us.

Stacks, for example, are generalizable - one can imagine a stack of cards, a stack of plates at the Sharpe Refectory, or a stack of notes. For all of these different kinds of stacks, you add an item (typically) by placing it on top of the stack, and when you remove an item you (typically) remove it from the top of the stack — these actions are inherent to a stack regardless of what actually
comprises it. Once you define an ADT like a Stack, you can use it in a million places, knowing that the same functionality remains across all implementations.

Another important benefit of using ADTs lies in data hiding. Data hiding refers to the idea that people working on different parts of a project shouldn’t necessarily know what other folks are working on. This means that a programmer can focus on solely their code, while not being distracted by the rest of the implementation of a project. Perhaps more importantly, through data hiding, the number of people who are allowed to change specific parts of a project can be limited.

ADTs also allow for something called implementation neutrality. Implementation neutrality refers to the idea that when you use a module that meets the module type `STACK`, you can easily switch it out for another module that also implements the type `STACK`. This can come in handy if you know that one implementation is generally faster than the other for certain procedures that you know you will be using more often.

1.1 Invariants

Let’s take a moment to think back to binary search trees. We made a rule that every value in the left child of a node must be less than the value of the node. A rule like this that is enforced in a data structure is called an invariant, and following such a rule throughout the implementation of a data structure is typically called maintaining an invariant. Invariants implemented by the programmer in a module implementation are often hidden from the user, but because access to the implementation is limited, one can ensure that some invariant is maintained throughout the functions of the ADT. Maintaining invariants in this way can often lead to simpler code, and sometimes to more complex code. So why do we do it? Because maintaining invariants can also lead to code that is provably correct.

We won’t, in CS17, prove any code is correct, but for things like, say, your self-driving car, you’d sure like to be confident that the code is correct, right?

2 Queues

Suppose we’d like to store an ordered collection of data. Sometimes, we’ll want the container for our data to be LIFO (“last in, first out”), which means that the easiest entry to access is the one that was most recently added to the collection. As we saw in the last lecture, the LIFO data structure is called a stack.

Other times, we’ll want it to be FIFO (“first in, first out”) which means that the easiest entry to access is the one that was least recently added to the collection. Think about a line of people waiting for a bus or a bank teller; the first person in line gets helped first. The FIFO data structure is called a queue.

The operations supported by a queue are similar to those supported by a stack, but we use different names for them:

- check if the queue is_empty;
- enqueue a datum onto one end (which, reflecting the idea that a queue is like a line of people waiting for a bus, we’ll call the “back”) of the queue;
• dequeue the datum at the opposite end (say, the front) of the queue; and
• peek at the datum at the opposite end (again, the front) of the queue.

An important application of stacks and queues arises in (tree, or more generally, graph) search. There are two basic search strategies: depth-first, which searches deep before it searches wide, and breadth-first, which searches wide before it searches deep.

Depth-first search is implemented using stacks: all nodes at depth 1 are pushed onto the stack; then one of those nodes is popped, and its successors are pushed onto the top of the stack; then one of those successors is popped, and its successors are pushed onto the top of the stack; and so on. In this way, deeper nodes are visited before wider ones.

In contrast, breadth-first search is implemented using queues: all nodes at depth 1 are enqueued onto the queue; then one of those nodes is dequeued, and all of its successors are enqueued onto the back of the queue; then a node at the front of the queue is dequeued (another one at depth 1), and all of its successors are enqueued onto the back of the queue; and so on. Here, all nodes at depth 1 are visited before any at depth 2, and so on.

2.1 A First Attempt: A Single Queue

Here is the ReasonML signature for queues. It includes the name of the data type, the constructor, and the types of the various operations:

```reason
module type QUEUE = {
  type queue('a);

  let empty: queue('a);
  let is_empty: queue('a) => bool;

  let enqueue: ('a, queue('a)) => queue('a);

  let dequeue: queue('a) => queue('a);
  let peek: queue('a) => 'a;
};
```

As said earlier, enqueue inserts a datum onto one end of the queue and dequeue removes an item from the queue. You could say that these are ‘similar’ to a list’s insert and delete function, or a stack’s push and pop, in the sense that these operations insert and remove an item from those data structures. Some texts use enq and deq as shorthand. Unfortunately, there’s also a data structure called a double-ended queue, which is sometimes abbreviated ‘DEQ’. There are really no good solutions.
Efficiently implementing a stack is trivial; we can just use a list and each operation is $O(1)$. But how can we efficiently implement a queue? As a first attempt, let’s try using lists again. Here is the ReasonML module that uses lists to implement the module type QUEUE:

```reasonml
module ListQueue: QUEUE = {
  type queue('a) =
  | Queue(list('a));

  let empty = Queue([]);

  let is_empty = q => q == empty;

  let enqueue = (datum, Queue(lst)) => Queue([datum, ...lst]);

  let rec init: list('a) => list('a) = (fun
    | [] => []
    | [hd, ...tl] => [hd, ...init(tl)]
    | [] => failwith("init: invalid input (empty list)"):
      list('a) => list('a)
    );

  let rec last: list('a) => 'a = (fun
    | [hd] => hd
    | [], ...tl] => last(tl)
    | [] => failwith("last: invalid input (empty list)"):
      list('a) => 'a
    );

  let dequeue = (Queue(lst)) => Queue(init(lst));

  let peek = (Queue(lst)) => last(lst);
};
```

Hm... to enqueue a datum onto a queue is simple; we just cons the datum onto the list. But to dequeue a datum off the queue (to view the datum at the front of the queue), we have to access the last element of the list, which takes $O(n)$ time, where $n$ is the number of elements in the list. This means if we enqueued and dequeued any number of elements in a queue we would use $k(n)$ operations.

Intuitively, it seems that we should be able to implement operations on a queue in $O(1)$ time, just as we did operations on a stack. In a stack, the datum at the top was the easiest to access. Similarly, for a queue, we must ensure that the datum at the front is the easiest to access.

So how can we achieve this? Let’s give it another go, this time storing the list in reverse:
let is_empty = q => q == empty;

let enqueue = (datum, Queue(lst)) => Queue(lst @ [datum]);

let dequeue = (Queue(lst)) =>
    switch (lst) {
      | [] => failwith("dequeue: invalid input (empty queue)")
      | [_, ...tl] => Queue(tl)
    };

let peek = (Queue(lst)) =>
    switch (lst) {
      | [] => failwith("peek: invalid input (empty queue)")
      | [hd, ..._] => hd
    };

Now dequeue and peek take constant time! That’s good. BUT, enqueueing a datum onto a queue requires appending that datum to the end of the list, which again takes \(O(n)\) time, where \(n\) is the number of elements in the list. That’s not so good.

Idea: What if we used two stacks?

### 2.2 A Second Attempt: A Double-list Queue

Let’s take a step back. Observe that the constant time implementation of enqueue does the right thing, assuming that data is popped off the end of the list. Similarly, the constant time implementations of dequeue and peek also do the right thing, assuming that data is enqueued at the end of the list.

Let’s try modifying our representation so that these operations don’t conflict. The key idea is to represent a queue using two lists, rather than just one:

1. a first list which stores data in service order, so that servicing/dequeuing data is easy; each datum is just popped off. We’ll call this list "front", since it stores data at the front of the queue.

2. a second list which stores data in reverse arrival order, so that storing/enqueuing data is also easy; each datum is just pushed onto the second list. We’ll call this list "back", since it stores data at the back.

Note: The slides used during this lecture also showed how to implement this idea, slightly differently, using two stacks.

For example, following our two-list implementation, the queue that contains the entries 32, 31, 18, and 17, in that order, with 32 at the front (i.e., 32 has been waiting in the queue the longest, and so is next in line to be serviced), can be represented by any of the following combinations of front and back lists (where the left column represents our "front" lists, and those on the right represent their corresponding "back" lists):
Using this idea, let’s have another go at implementing queues using lists:

```ocaml
module DoubleListQueue : QUEUE = {
  type queue('a) =
  | Queue(list('a), list('a));

  let empty = Queue([], []);

  let is_empty = q => q == empty;

  let enqueue = (datum, Queue(f, b)) =>
    Queue(f, [datum, ...b]);

  let dequeue = (Queue(f, b)) =>
    switch (f) {
      | [] => failwith("dequeue: invalid input (empty queue)")
      | [_ , ...tl] => Queue(tl, b)
    };

  let peek = (Queue(f, b)) =>
    switch (f) {
      | [] => failwith("peek: invalid input (empty queue)")
      | [hd, ..._] => hd
    };
};
```

This implementation is basically correct. That is, the various operations interact with the front and back lists correctly. But there’s one major problem. If we only push data on the back list and pop data off the front list, then there won’t ever be any data to dequeue!

One possible fix would be to write dequeue_helper and peek_helper — procedures that are called whenever the front list is empty, and report an error if the back list is also empty, but otherwise “do the right thing.”

In the case of dequeue, the right thing to do would be to produce a queue with a front list that is the tail of the reverse of the given, non-empty back list, together with a new, empty back list.

In the case of peek, the right thing to do would be to produce the last element of the given non-empty, back list, which can be accomplished by reversing the back list, and then producing its head.

```ocaml
let rec dequeue_helper : 'a list => 'a queue = ( fun
  | [] => failwith ("dequeue_helper: invalid input (empty queue)")
  | _ => (dequeue (Queue (List.rev b, [])))
)
and dequeue (Queue (f, b)) = 
  switch (f) {
  | [] => (dequeue_helper (b))
  | [_, ... tl] => Queue (tl, b)};

let rec peek_helper : 'a list -> 'a = 
  | [] => failwith ("peek_helper: invalid input (empty queue)")
  | _ => (peek (Queue (List.rev b, [])))
)

and peek (Queue (f, b)) = 
  switch (f) {
  | [] => (peek_helper (b))
  | [hd, ... _] => hd }

The proposed dequeue_helper has the advantage of resetting the queue, so that the front list is no longer empty, which means that further calls to dequeue will not require further list traversals. Unfortunately, the proposed peek_helper does not have this same advantage. Repeated calls to peek would necessitate repeated list traversals, since peek does not alter the configuration of the queue. Is there another way? One that would avoid these repeated list traversals by peek?

2.3 Plan B, or is it C or D, by now?

One way to ensure that peek operates in constant time is to maintain the following invariant:

The front list is never empty, unless the queue itself is empty.

In other words, if the queue is non-empty, the "front" list must also be non-empty. To maintain this invariant, we need to do two things:

- whenever we dequeue a datum (specifically, off the front list), we have to check that that does not leave us with an empty front list, and if it does, we have to refill it.
- whenever we enqueue a datum on to the queue (specifically, on to the back list), we again have to check that the front list is not empty (i.e., whether the queue itself was empty before this datum was enqueues onto it), and if it is, we have to refill it.

So, here’s how this, our final, implementation looks:

module DoubleListQueue: QUEUE = {
  type queue('a) =
  | Queue([list('a), list('a)));

  let empty = Queue([], []);

  let is_empty = q => q == empty;

  let rec refill = (f: list('a), b: list('a)) : queue('a) =>
    switch (f) {

| [] => Queue(List.rev(b), [])
| _  => refill(List.tl(f), [List.hd(f), ...b])
|

let enqueue = (datum, Queue(f, b)) =>
    refill(f, [datum, ...b]);

let dequeue = (Queue(f, b)) =>
    switch (f) {
        | [] => failwith("empty queue")
        | [_, ...tl] => refill(tl, b)
    };

let peek = (Queue(f, b)) =>
    switch (f) {
        | [] => failwith("empty queue")
        | [hd, ..._] => hd
    };
|

**Question:** How do we reach the base case in the `refill` procedure? That is, how is it that the front list is ever empty if our invariant says that it can’t be?

**Answer:** Our invariant says that the front list cannot be empty *unless the queue is empty*. Hence, we reach this base case whenever we push an item onto an empty queue. We can also reach the base case when we pop an item off a queue in which the length of the front list is 1, leaving the front list empty.

Let’s analyze this code.

- `is_empty` takes constant time
- `push` takes constant time, plus however long `refill` takes
- `pop` takes constant time, plus however long `refill` takes
- `peek` takes constant time

So what’s the runtime of `refill`? If the front list is nonempty, then `refill` takes constant time. But if the front list is empty, then `refill` calls `reverse`, which runs in time $\Theta(n \mapsto n)$, where $n$ is the number of elements in the list.

Uh-oh. Looks like we’re back where we started...or are we?

### 3 Amortized Analysis

Can you believe it? After all this work, it seems as though we’ve still only implemented queues with operations that run in linear time in the worst case, since `enqueue` and `dequeue` may require that we reverse the back list.
In fact, it not only *seems* that way — it actually *is* that way. Our queue operations are in fact worst-case linear-time. But there is still something we can salvage, by looking at the bigger picture of what happens over the course of a long sequence of queue operations, rather than a singular worst-case scenario.

When we carry out any series of operations on a queue, some of the reverse operations are expensive, but others are not, so that there is an upper bound on the *total* expense of all reverse operations. In what follows, we will defend this observation, and then conclude that each individual operation runs in constant *amortized* time.

**Amortized analysis** is a technique by which the run time of each of a sequence of operations is determined by averaging the *aggregate* run time (or a bound on this aggregate) over the sequence.

All operations other than enqueue and dequeue take constant time. So, we only need to show that sequences of enqueue and dequeue operations are amortized constant time. Let’s look at some examples.

### 3.1 Example 1

First, suppose we perform alternating enqueue and dequeue operations.

- Because the front list is empty upon each enqueue, each such operation involves enqueuing an element onto the back list, and then setting the front list equal to the reverse of the back list.
- All this reversing may sound expensive, but since the lists that are reversed are only of length 1, each reverse operation actually only takes constant time.
- Each dequeue is also constant time, since the lists that are supposedly reversed with each dequeue are empty!
- So all the operations in this sequence take (non-amortized) constant time.

But this was not a worst-case sequence, so let’s look at another one.

### 3.2 Example 2

In particular, let’s try $n$ enqueue operations followed by $n$ dequeue operations:

- Upon the first enqueue, the front list is empty, so this enqueue results in a call to refill, which moves the one element in the queue from the back list to the front list, in constant time.
- The other calls to enqueue are also constant, for a total run time of $(n - 1)\Theta(n \mapsto 1)$.
- Then we dequeue. After the first dequeue (constant time), the front list is empty, so refill is called; the call to refill is expensive this time $-\Theta(n \mapsto n)$.
- But then the rest of the calls to dequeue are constant, for a total cost of $(n - 1)\Theta(n \mapsto 1)$.
- Thus, the total run time of the entire sequence is $2(n - 1)\Theta(n \mapsto 1) + \Theta(n \mapsto n) = \Theta(n \mapsto n)$, which averages to $\Theta(n \mapsto n)/2n = \Theta(n \mapsto 1)$ time per operation.
Indeed, for any sequence of $n$ operations, we might consider the total run time of the entire sequence to be $\Theta(n \mapsto n)$. This claim follows from the following observation: after a $\Theta(n \mapsto n)$ refill operation, there is enough fodder on the queue for the next $n$ operations (at least) to run in constant time. Spreading the cost of an $\Theta(n \mapsto n)$ operation across $n$ operations yields amortized constant time.

There's an alternative analysis, which we'll call a "total work analysis": if we start with an empty queue, and do $n$ queue operations on it, then at most $n$ data items are involved. And in any data item's progress through a queue, at most four stack operations happen to it: it gets pushed and popped on the back stack, and pushed and popped on the front stack. That's a total of at most 4 operations, so the largest possible number of basic operations (where we count each stack operation as "basic") is $4n$, so the average number per queue operation is no more than 4, which is a constant.

4 Summary

Ideas

- ADTs are useful for many reasons including: generalizibility, data hiding, and implementation neutrality.

- Maintaining an invariant can be a useful tool when implementing an ADT.

- A queue is similar to a stack, except it implements a FIFO ("first in, first out") data structure. Queues can be implemented in many ways, including as two lists, or as two stacks.

Skills

- You now know how to run a rough amortized analysis as we did for looking at the runtime of procedures in our queue implementation using two lists.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback)