Lecture Notes on Stacks & Queues

15-122: Principles of Imperative Computation
Frank Pfenning, André Platzer, Rob Simmons

Lecture 9
September 24, 2013

1 Introduction

In this lecture we introduce *queues* and *stacks* as data structures, e.g., for managing tasks. They follow similar principles of organizing the data. Each provides simple functions for adding and removing elements. But they differ in terms of the order in which the elements are removed. They can be implemented easily as a library in C0. In this lecture, we will focus on the abstract principles of queues and stacks and defer a detailed implementation to the next lecture.

Relating this to our learning goals, we have

Computational Thinking: We illustrate the power of *abstraction* by considering both the client-side and library-side of the interface to a data structure.

Algorithms and Data Structures: Queues and stacks are important data structures in their own right, but also our first examples of *abstract datatypes.*

Programming: Use and design of interfaces.

2 The Stack Interface

Stacks are data structures that allow us to insert and remove items. They operate like a stack of papers or books on our desk - we add new things to the *top* of the stack to make the stack bigger, and remove items from the top

as well to make the stack smaller. This makes stacks a LIFO (Last In First Out) data structure – the data we have put in last is what we will get out first.

Before we consider the implementation of a data structure it is helpful to consider the interface. We then program against the specified interface. Based on the description above, we require the following functions:

```
/* type elem must be defined by the client */

bool stack_empty(stack S);  /* O(1), check if stack empty */
stack stack_new();          /* O(1), create new empty stack */
void push(stack S, elem e); /* O(1), add item on top of stack */
elem pop(stack S)           /* O(1), remove item from top */
//@requires !stack_empty(S);
  ;
```

We want the creation of a new (empty) stack as well as pushing and popping an item all to be constant-time operations, as indicated by $O(1)$. Furthermore, pop is only possible on non-empty stacks. This is a fundamental aspect of the interface to a stack, that a client can only read data from a non-empty stack. So we include this as a requires contract in the interface.

We are being quite abstract here — we do not write, in this file, what type the elements of the stack have to be. Instead we assume that at the top of the file, or before this file is read, we have already defined a type `elem` for the type of stack elements. We say that the implementation is *generic or polymorphic* in the type of the elements. Unfortunately, neither C nor C0 provide a good way to enforce this in the language and we have to rely on programmer discipline.

In the future, we will sometimes indicate that we have a typedef waiting to be filled in by the client by writing the following:

```
typedef _________ elem;
```

This is not actually valid C0, but the client using this library will be able to fill in the underscores with a valid type to make the stack a stack of this type. In this example, we will assume that the client wrote

```
typedef string elem;
```

The critical point here is that this is a choice that is up to the *user* of the library (the *client*), and it is not a choice that the stack library needs to know or care about.

3 Using the Stack Interface

We play through some simple examples to illustrate the idea of a stack and how to use the interface above. We write a stack as

$$x_1, x_2, \ldots, x_n$$

where x_1 is the *bottom* of the stack and x_n is the *top* of the stack. We *push* elements on the top and also *pop* them from the top.

For example:

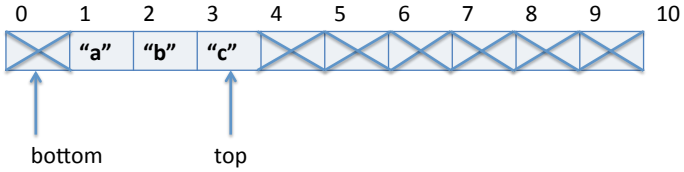
Stack	**Command**	**Other variables**
	`stack S = stack_new();`	
	`push(S, "a");`	
`"a"`	`push(S, "b");`	
`"a","b"`	`string e = pop(S);`	$e =$ `"b"`
`"a"`	`push(S, "c");`	$e =$ `"b"`
`"a","c"`	`e = pop(S);`	$e =$ `"c"`
`"a"`		$e =$ `"c"`

4 One Stack Implementation (With Arrays)

Any programming language is going to come with certain data structures "built-in." Arrays, the only really complex data structure we have used so far in this class, are one example in C0. Other data structures, like stacks and queues, need to be constructed using more primitive language features.

We will get to a more proper implementation of stacks in the next lecture, using linked lists. For this lecture we will implement stacks by using the familiar arrays that we have already been using so far in this class.

The idea is to put all data elements in an array and maintain an integer `top`, which is the index where we read off elements.



To help identify the similarities with the queue implementation, we decide to also remember an integer `bottom`, which is the index of the bottom of the stack. (The bottom will, in fact, remain 0.)

With this design decision, we do not have to handle the bottom of the stack much different than any other element on the stack. The difference is that the data at the bottom of the stack is meaningless and will not be used in our implementation.

There appears to be a very big limitation to this design of stacks: our stack can't contain more than 9 elements, like a pile of books on our desk that cannot grow too high lest it reach the ceiling or fall over. There are multiple solutions to this problem, but for this lecture we will be content to work with stacks that have a limited maximum capacity.

4.1 Structs and data structure invariants

Currently, our picture of a stack includes three different things: an array containing the data, an integer indicating where the top is, and an integer indicating where the bottom is. This is similar to the situation in Homework 1 where we had data (an array of pixels) and two integers, a width and a height.

C0 has a feature that allows us to bundle these things up into a `struct` rather than passing around all the pieces separately. We define:

```
struct stack_header {
  elem[] data;
  int top;
  int bottom;
};
typedef struct stack_header* stack;
```

What this notation means exactly, and especially what the part with `struct stack_header*` is all about, will be explained in the next lecture. (These are pointers and it is crucial to understand them, but we defer this topic for now.) For now, it is sufficient to think of this as providing a notation for bundling aggregate data. When we have a struct `S` of type `stack`, we can refer to the data as `S->data`, the integer representing the top of the stack as `S->top`, and the integer representing the bottom of the stack as `S->bottom`.

When does a struct of this type represent a valid stack? Whenever we define a new data type representation we should first think about the data structure invariants. Making these explicit is important as we think about

and write the pre- and postconditions for functions that implement the interface. Here, it is a simple check of making sure that the `bottom` and `top` indices are in the range of the array and that `bottom` stays at 0, where we expect it to be.

```
bool is_stack(stack S)
{
  if (!(S->bottom == 0)) return false;
  if (!(S->bottom <= S->top)) return false;
  //@assert S->top < \length(S->data);
  return true;
}
```

WARNING: This specification function is missing something very important (a check for `NULL`) – we will return to this next time!

When we write specification functions, we use a style of repeatedly saying

```
  if (!(some invariant of the data structure)) return false;
```

so that we can read off the invariants of the data structure. A specification function like `is_stack` should be safe – it should only ever return `true` or `false` or raise an assertion violation – and if possible it should avoid raising an assertion violation. Assertion violations are sometimes unavoidable because we can only check the length of an array inside of the assertion language.

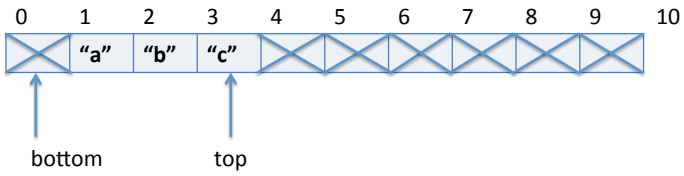
4.2 Checking for emptiness

To check if the stack is empty, we only need to check whether `top` and `bottom` are the same number.

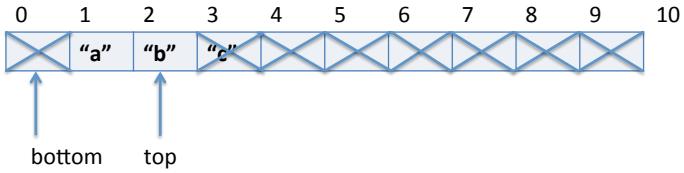
```
bool stack_empty(stack S)
//@requires is_stack(S);
{
  return S->top == S->bottom;
}
```

4.3 Popping from a stack

To pop an element from the stack we just look up the data that is stored at the position indicated by the `top` field of the stack in the array `S->data` of the `data` field of the stack. To indicate that this element has now been removed from the stack, we decrement the `top` field of the stack. We go from



to



The `"c"` can still be present in the array at position 3, but it is now a part of the array that we don't care about, which we indicate by putting an `X` over it. In code, popping looks like this:

```
elem pop(stack S)
//@requires is_stack(S);
//@requires !stack_empty(S);
//@ensures is_stack(S);
{
  elem r = S->data[S->top];
  S->top--;
  return r;
}
```

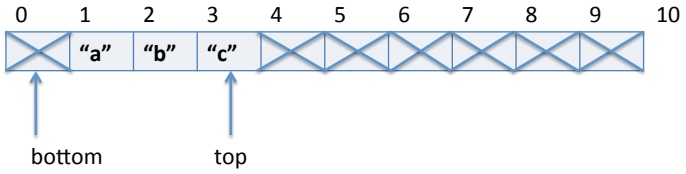
Notice that contracts are cumulative. Since we already indicated

```
//@requires !stack_empty(S);
```

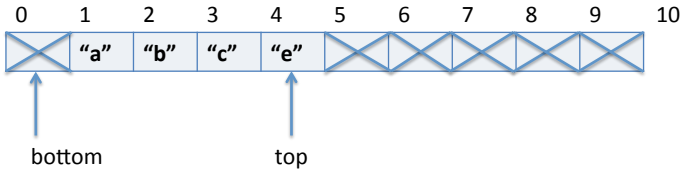
in the interface of `pop`, we would not have to repeat this requires clause in the implementation. We repeat it regardless to emphasize its importance.

4.4 Pushing onto a stack

To push an element onto the stack, we increment the `top` field of the stack to reflect that there are more elements on the stack. And then we put the element `e` into the array `S->data` at position `top`. While this is simple, it is still a good idea to draw a diagram. We go from



to



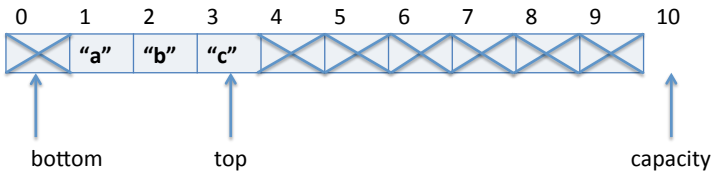
In code:

```
void push(stack S, elem e)
//@requires is_stack(S);
//@ensures is_stack(S);
{
  S->top++;
  S->data[S->top] = e;
}
```

Why is the array access `S->data[S->top]` safe? Is it even safe? At this point, it is important to note that it is not safe if we ever try to push more elements on the stack than we have reserved space for. We fully address this shortcoming of our stack implementation in the next lecture. What we can do right now to address the issue is to redesign the `struct stack_header` by adding a `capacity` field that remembers the length of the array of the `data` field:

```
struct stack_header {
  elem[] data;
  int top;
  int bottom;
  int capacity;           // capacity == \length(data);
};
typedef struct stack_header* stack;
```

Giving us the following updated view of array-based stacks:



The comment that `capacity == \length(data)` is helpful for indicating what the intent of `capacity` is, but it is preferable for us to modify our `is_stack` function to account for the change. (The **WARNING** from before still applies here.)

```
bool is_stack(stack S)
{
  if (!(S->bottom == 0)) return false;
  if (!(S->bottom <= S->top)) return false;
  if (!(S->top < S->capacity)) return false;
  //@assert S->capacity == \length(S->data);
  return true;
}
```

With a `capacity` in hand, we check for sufficient space with an explicit assert statement before we try to access the array or change `top`.

```
void push(stack S, elem e)
//@requires is_stack(S);
//@ensures is_stack(S);
{
  assert(S->top < S->capacity - 1);  // otherwise no space left
  S->top++;
  S->data[S->top] = e;
}
```

This assertion can indeed fail if the client tries to push too many elements on the stack, which is why we use a *hard assert* – an assertion that will run whether or not we compile with `-d`. The alternative would be to expose the capacity of the stack to the user with a `stack_full` function and then add a precondition `//@requires !stack_full(S)` to our `push()` function.

4.5 Creating a new stack

For creating a new stack, we allocate a `struct stack_header` and initialize the `top` and `bottom` numbers to 0.

```
stack stack_new()
//@ensures stack_empty(\result);
//@ensures is_stack(\result);
{
  stack S = alloc(struct stack_header);
  S->bottom = 0;
  S->top = 0;
  S->capacity = 100;               // arbitrary resource bound
  S->data = alloc_array(elem, S->capacity);
  return S;
}
```

As shown above, we also need to allocate an array `data` to store the elements in. At this point, at the latest, we realize a downside of our stack implementation. If we want to implement stacks in arrays in the simple way that we just did, the trouble is that we need to decide its capacity ahead of time. That is, we need to decide how many elements at maximum will ever be allowed in the stack at the same time. Here, we arbitrarily choose the capacity 100, but this gives us a rather poor implementation of stacks in case the client needs to store more data. We will see how to solve this issue with a better implementation of stacks in the next lecture.

This completes the implementation of stacks, which are a very simple and pervasive data structure.

5 Abstraction

An important point about formulating a precise interface to a data structure like a stack is to achieve *abstraction*. This means that as a client of the data structure we can only use the functions in the interface. In particular, we are not permitted to use or even know about details of the implementation of stacks.

Let's consider an example of a client-side program. We would like to examine the element at the top of the stack without removing it from the stack. Such a function would have the declaration

```
string peek(stack S)
//@requires !stack_empty(S);
  ;
```

The first instinct might be to write it as follows:

```
string peek(stack S)
//@requires !stack_empty(S);
{
  return S->data[S->top];
}
```

However, this would be *completely wrong*. Let's recall the interface:

```
bool stack_empty(stack S);    /* O(1), check if stack empty */
stack stack_new();            /* O(1), create new empty stack */
void push(stack S, elem e); /* O(1), add item on top of stack */
elem pop(stack S);          /* O(1), remove item from top */
//@requires !stack_empty(S);
  ;
```

We don't see any `top` field, or any `data` field, so accessing these as a client of the data structure would violate the abstraction. Why is this so wrong? The problem is that if the library implementer decided to improve the code, or perhaps even just rename some of the structures to make it easier to read, then the client code will suddenly break! In fact, we will provide a different implementation of stacks in the next lecture, which would make the above implementation of `peek` break. With the above client-side implementation of `peek`, the stack interface does not serve the purpose it is intended for, namely provide a reliable way to work with a data structure. Interfaces are supposed to separate the implementation of a data structure

in a clean way from its use so that we can change one of the two without affecting the other.

So what can we do? It is possible to implement the peek operation *without violating the abstraction*! Consider how before you read on.

The idea is that we pop the top element off the stack, remember it in a temporary variable, and then push it back onto the stack before we return.

```
string peek(stack S)
//@requires !stack_empty(S);
{
  string x = pop(S);
  push(S, x);
  return x;
}
```

This is clearly less efficient: instead of just looking up the fields of a struct and accessing an element of an array we actually have to pop an element and then push it back onto the stack. However, it is still a constant-time operation ($O(1)$) since both `pop` and `push` are constant-time operations. Nonetheless, we have a possible argument to include a function `peek` in the interface and implement it library-side instead of client-side to save a small constant of time.

If we are actually prepared to extend the interface, then we can go back to our original implementation.

```
string peek(stack S)
//@requires !stack_empty(S);
{
  return S->data[S->top];
}
```

Is this a good implementation? Not quite. First we note that inside the library we should refer to elements as having type `elem`, not `string`. For our running example, this is purely a stylistic matter because these two are synonyms. But, just as it is important that clients respect the library interface, it is important that the library respect the client interface. In this case, that means that the users of a stack can, without changing the library, decide to change the definition of `elem` type in order to store different data in the stack.

Second we note that we are now missing a precondition. In order to even check if the stack is non-empty, we first need to be assured that it is a valid stack. On the client side, all elements of type `stack` come from the library, and any violation of data structure invariants could only be discovered when we hand it back through the library interface to a function implemented in the library. Therefore, the client can assume that values of

type `stack` are valid and we don't have explicit pre- or post-conditions for those. Inside the library, however, we are constantly manipulating the data structure in ways that break and then restore the invariants, so we should check if the stack is indeed valid.

From these two considerations we obtain the following code for *inside the library*:

```
elem peek(stack S)
//@requires is_stack(S);
//@requires !stack_empty(S);
{
  return S->data[S->top];
}
```

6 Computing the Size of a Stack

Let's exercise our data structure once more by developing two implementations of a function that returns the size of a stack: one on the client's side, using only the interface, and one on the library's side, exploiting the data representation. Let's first consider a client-side implementation, using only the interface so far.

```
int stack_size(stack S);
```

Again, we encourage you to consider this problem and program it before you read on.

First we reassure ourselves that it will not be a simple operation. We do not have access to the array (in fact, as the client, we cannot know that there is an array), so the only thing we can do is pop all the elements off the stack. This can be accomplished with a while-loop that finishes as soon as the stack is empty.

```
int stack_size(stack S) {
  int count = 0;
  while (!stack_empty(S)) {
    pop(S);
    count++;
  }
  return count;
}
```

However, this function has a *big* problem: in order to compute the size we have to destroy the stack! Clearly, there may be situations where we would like to know the number of elements in a stack without deleting all of its elements. Fortunately, we can use the idea from the `peek` function in amplified form: we maintain a new *temporary stack* T to hold the elements we pop from S. Once we are done counting, we push them back onto S to repair the damage.

```
int stack_size(stack S) {
  stack T = stack_new();
  int count = 0;
  while (!stack_empty(S)) {
    push(T, pop(S));
    count++;
  }
  while (!stack_empty(T)) {
    push(S, pop(T));
  }
  return count;
}
```

The complexity of this function is clearly $O(n)$, where n is the number of elements in the stack S, since we traverse each while loop n times, and perform a constant number of operations in the body of both loops. For that, we need to know that `push` and `pop` are constant time ($O(1)$).

What about a library-side implementation of `stack_size`? This can be done more efficiently.

```
int stack_size(stack S)
//@requires is_stack(S);
{
  return S->top - S->bottom;
}
```

7 The Queue Interface

A *queue* is a data structure where we add elements at the back and remove elements from the front. In that way a queue is like "waiting in line": the first one to be added to the queue will be the first one to be removed from the queue. This is also called a FIFO (First In First Out) data structure. Queues are common in many applications. For example, when we read a book from a file as in Assignment 2, it would be natural to store the words in a queue so that when we are finished reading the file the words are in the order they appear in the book. Another common example are buffers for network communication that temporarily store packets of data arriving on a network port. Generally speaking, we want to process them in the order that they arrive.

Here is our interface:

```
/* type elem must be defined */

bool queue_empty(queue Q); /* O(1), check if queue is empty */
queue queue_new();         /* O(1), create new empty queue */
void enq(queue Q, elem s); /* O(1), add item at back */
elem deq(queue Q)          /* O(1), remove item from front */
//@requires !queue_empty(Q);
  ;
```

Dequeuing is only possible on non-empty queues, which we indicate by a requires contract in the interface.

We can write out this interface without committing to an implementation of queues. In particular, the type `queue` remains *abstract* in the sense that we have not given its definition. This is important so that different implementations of the functions in this interface can choose different representations. Clients of this data structure should not care about the internals of the implementation. In fact, they should not be allowed to access them at all and operate on queues only through the functions in this interface. Some languages with strong module systems enforce such abstraction

rigorously. In C, it is mostly a matter of adhering to conventions.

8 Using the Queue Interface

We play through some simple examples to illustrate the idea of a queue and how to use the interface above. We write a queue as

$$x_1, x_2, \ldots, x_n$$

where x_1 is the *front* of the queue and x_n is the *back* of the queue. We *enqueue* elements in the back and *dequeue* them from the front.

For example:

Queue	**Command**	**Other variables**
	`queue Q = queue_new();`	
	`enq(Q, "a");`	
`"a"`	`enq(Q, "b");`	
`"a","b"`	`string s = deq(Q);`	$s =$ `"a"`
`"b"`	`enq(Q, "c");`	$s =$ `"a"`
`"b","c"`	`s = deq(Q);`	$s =$ `"b"`
`"c"`		$s =$ `"b"`

9 Copying a Queue Using Its Interface

Suppose we have a queue Q and want to obtain a copy of it. That is, we want to create a new queue C and implement an algorithm that will make sure that Q and C have the same elements and in the same order. How can we do that? Before you read on, see if you can figure it out for yourself.

The first thing to note is that

```
queue C = Q;
```

will not have the effect of copying the queue Q into a new queue C. Just as for the case of arrays, this assignment makes C and Q aliases, so if we change one of the two, for example enqueue an element into C, then the other queue will have changed as well. Just as for the case of arrays, we need to implement a function for copying the data.

The queue interface provides functions that allow us to dequeue data from the queue, which we can do as long as the queue is not empty. So we create a new queue C. Then we read all data from queue Q and put it into the new queue C.

```
queue C = queue_new();
while (!queue_empty(Q)) {
  enq(C, deq(Q));
}
//@assert queue_empty(Q);
```

Now the new queue C will contain all data that was previously in Q, so C is a copy of what used to be in Q. But there is a problem with this approach. Before you read on, can you find out which problem?

Queue C now is a copy of what used to be in Q before we started copying. But our copying process was destructive! By dequeueing all elements from Q to put them into C, Q has now become empty. In fact, our assertion at the end of the above loop even indicated `queue_empty(Q)`. So what we need to do is put all data back into Q when we are done copying it all into C. But where do we get it from? We could read it from the copy C to put it back into Q, but, after that, the copy C would be empty, so we are back to where we started from. Can you figure out how to copy all data into C and make sure that it also ends up in Q? Before you read on, try to find a solution for yourself.

We could try to enqueue all data that we have read from Q back into Q before putting it into C.

```
queue C = queue_new();
while (!queue_empty(Q)) {
  string s = deq(Q);
  enq(Q, s);
  enq(C, s);
}
//@assert queue_empty(Q);
```

But there is something very fundamentally wrong with this idea. Can you figure it out?

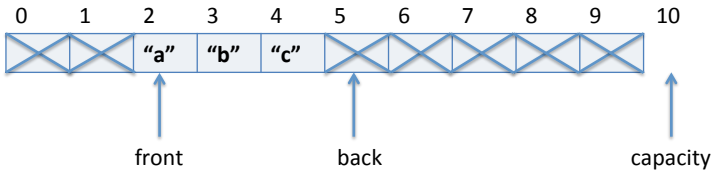
The problem with the above attempt is that the loop will never terminate unless Q is empty to begin with. For every element that the loop body dequeues from Q, it enqueues one element back into Q. That way, Q will always have the same number of elements and will never become empty. Therefore, we must go back to our original strategy and first read all elements from Q. But instead of putting them into C, we will put them into a third queue T for temporary storage. Then we will read all elements from the temporary storage T and enqueue them into both the copy C *and* back into the original queue Q. At the end of this process, the temporary queue T will be empty, which is fine, because we will not need it any longer. But both the copy C and the original queue Q will be replenished with all the elements that Q had originally. And C will be a copy of Q.

```
queue queue_copy(queue Q) {
  queue T = queue_new();
  while (!queue_empty(Q)) {
    enq(T, deq(Q));
  }
  //@assert queue_empty(Q);
  queue C = queue_new();
  while (!queue_empty(T)) {
    string s = deq(T);
    enq(Q, s);
    enq(C, s);
  }
  //@assert queue_empty(T);
  return C;
}
```

For example, when `queue_copy` returns, neither C nor Q will be empty. Except if Q was empty to begin with, in which case both C and Q will still be empty in the end.

10 The Queue Implementation

In this lecture, we implement the queue using an array, similar to how we have implemented stacks in this lecture.



A queue is implemented as a struct with a `front` and `back` field. The `front` field is the index of the front of the queue, the `back` field is the index of the back of the queue. We need both so that we can dequeue (at the front) and enqueue (back).

In the stack, we did not use anything outside the range $(bottom, top]$, and for queues, we do not use anything outside the range $[front, back)$. Again, we mark this in diagrams with an `X`.

The above picture yields the following definition, where we will again remember the capacity of the queue, i.e., the length of the array stored in the `data` field.

```
struct queue_header {
  elem[] data;
  int front;
  int back;
  int capacity;
};
typedef struct queue_header* queue;
```

When does a struct of this type represent a valid queue? In fact, whenever we define a new data type representation we should first think about the data structure invariants. Making these explicit is important as we think about and write the pre- and postconditions for functions that implement the interface.

What we need here is simply that the `front` and `back` are within the array bounds for array `data` and that the capacity is not too small. The `back` of the queue is not used (marked `X`) but in the array, so we decide to

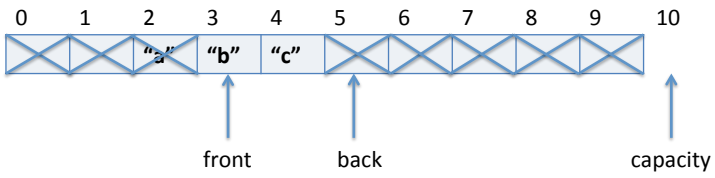
require that the capacity of a queue be at least 2 to make sure we can store at least one element. (The **WARNING** about NULL still applies here.)

```
bool is_queue(queue Q)
{
  if (Q->capacity < 2) return false;
  if (Q->front < 0 || Q->front >= Q->capacity) return false;
  if (Q->back < 0 || Q->back >= Q->capacity) return false;
  //@assert Q->capacity == \length(Q->data);
  return true;
}
```

To check if the queue is empty we just compare its front and back. If they are equal, the queue is empty; otherwise it is not. We require that we are being passed a valid queue. Generally, when working with a data structure, we should always require and ensure that its invariants are satisifed in the pre- and post-conditions of the functions that manipulate it. Inside the function, we will generally temporarily violate the invariants.

```
bool queue_empty(queue Q)
//@requires is_queue(Q);
{
  return Q->front == Q->back;
}
```

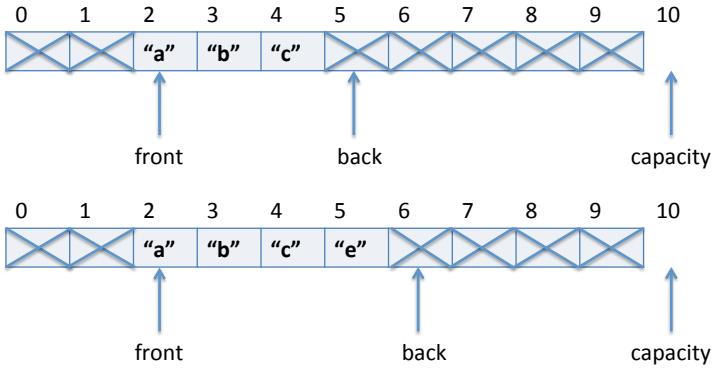
To dequeue an element, we only need to increment the field `front`, which represents the index in `data` of the front of the queue. To emphasize that we never use portions of the array outside the `front` to `back` range, we first save the dequeued element in a temporary variable so we can return it later. In diagrams:



And in code:

```
elem deq(queue Q)
//@requires is_queue(Q);
//@requires !queue_empty(Q);
//@ensures is_queue(Q);
{
  elem e = Q->data[Q->front];
  Q->front++;
  return e;
}
```

To enqueue something, that is, add a new item to the back of the queue, we just write the data (here: a string) into the extra element at the back, and increment `back`. You should draw yourself a diagram before you write this kind of code. Here is a before-and-after diagram for inserting `"e"`:



In code:

```
void enq(queue Q, string s)
//@requires is_queue(Q);
//@ensures is_queue(Q);
//@ensures !queue_empty(Q);
{
  assert(Q->back < Q->capacity-1);  // otherwise out of resources
  Q->data[Q->back] = e;
  Q->back++;
}
```

To obtain a new empty queue, we allocate a queue header struct and initialize both `front` and `back` to 0, the first element of the array. We do not initialize the elements in the array because its contents are irrelevant until some data is put in. It is good practice to always initialize memory if we care about its contents, even if it happens to be the same as the default value placed there.

```
queue queue_new()
//@ensures is_queue(\result);
//@ensures queue_empty(\result);
{
  queue Q = alloc(struct queue_header);
  Q->front = 0;
  Q->back = 0;
  Q->capacity = 100;
  Q->data = alloc_array(elem, Q->capacity);
  return Q;
}
```

Observe that, unlike the queue implementation, the queue interface only uses a single contract: that `deq` requires a non-empty queue to work. The queue implementation has several additional implementation contracts. All queue implementation functions use `is_queue(Q)` in their requires and ensures contract. The only exception is the `queue_new` implementation, which ensures the analogue `is_queue(\result)` instead. These `is_queue` contracts do not appear in the queue interface because `is_queue` itself does not appear in the interface, because it is an *internal data structure invariant*. If the client obeys the interface abstraction, all he can do with queues is create them via `queue_new` and then pass them to the various queue operations in the interface. At no point in this process does the client have the opportunity to tamper with the queue data structure to make it fail `is_queue`, unless the client violates the interface.

But there are other additional contracts in the queue implementation, which we want to use to check our implementation, and they still are not part of the interface. For example. we could have included the following additional contracts in the interface

```
queue queue_new()          /* O(1), create new empty queue */
//@ensures queue_empty(\result);
  ;
void enq(queue Q, elem s)  /* O(1), add item at back */
```

```
//@ensures !queue_empty(Q);
  ;
```

Those contracts need to hold for all queue implementations. Why did we decide not to include them? The reason is that there are not many situations in which this knowledge about queues is useful, because we rarely want to dequeue an element right after enqueueing it. This is in contrast to the `//@requires !queue_empty(Q)` contract of `deq`, which is critical for the client to know about, because he can only dequeue elements from non-empty queues and has to check for non-emptyness before calling `deq`.

Similar observations hold for our rationale for designing the stack interface.

11 Bounded versus Unbounded Stacks & Queues

Both the queue and the stack implementation that we have seen so far have a fundamental limitation. They are of bounded capacity. However large we allocate their `data` arrays, there is a way of enqueuing elements into the queue or pushing elements onto the stack that requires more space than the array has had in the first place. And if that happens, the `enq` or `push` operations will fail an assertion because of a resource bound that the client has no way of knowing about. This is bad, because the client would have to expect that any of his `enq` or `push` operations might fail, because he does not know about the capacity of the queue and has no way of influencing this.

One way of solving this problem would be to add operations into the interface that make it possible to check whether a queue is full.

```
bool queue_full(queue Q);
```

Then we change the precondition of `enq` to require that elements can only be enqueued if the queue is not full

```
void enq(queue Q, elem s)
//@requires !queue_full(Q)
....
```

Similarly, we could add an operation to the interface of stacks to check whether the stack is full

```
bool stack_full(stack S);
```

And require that pushing is only possible if the stack is not full

```
void push(stack S, elem s)
//@requires !stack_full(S)
....
```

The advantage of this design is that the client now has a way of checking whether there still is space in the stack/queue. The downside, however, is that the client still does not have a way of increasing the capacity if he wants to store more data in it.

In the next lecture, we will see a better implementation of stacks and queues that does not have any of those capacity bounds. That implementation uses pointers and linked lists.

Exercises

Exercise 1 *Can you implement a version of stack that does not use the* `bottom` *field in the* `struct stack_header`*?*

Exercise 2 *Consider what would happen if we* `pop` *an element from the empty stack when contracts are not checked? When does an error arise?*

Exercise 3 *Our queue implementation wastes a lot of space unnecessarily. After enqueuing and dequeueing a number of elements, the* `back` *may reach the capacity limit. If the* `front` *has moved on, then there is a lot of space wasted in the beginning of the* `data` *array. How can you change the implementation to reuse this storage for enqueuing elements? How do you need to change the implementation of* `enq` *and* `deq` *for that purpose?*

Exercise 4 *Our queue design always "wasted" one element that we marked* `X`*. Can we save this memory and implement the queue without extra elements? What are the tradeoffs and alternatives when implementing a queue?*

Exercise 5 *The stack implementation using arrays may run out of space if its capacity is exceeded. Can you think of a way of implementing unbounded stacks stored in an array?*