Stacks and Queues

An Introduction to Data Structures

CSE/IT 122

NMT Department of Computer Science and Engineering

> "Bad programmers worry about the code. Good programmers worry about data structures and their relationships."
>
> — Linus Torvalds

> "Smart data structures and dumb code works a lot better than the other way around."
>
> — Eric S. Raymond

---

1 Introduction

Before we look at how to implement *stacks* and *queues* in C, we should first introduce them as **data structures**. Specifically these are data structures for managing tasks. They follow similar principles of organizing the data, both provide functionality for adding and removing elements, but differ in how they handle removal. So, without further ado, lets jump in!

2 Stacks

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 also `remove` items from the *top* to make the stack smaller. This makes stacks a LIFO (Last In First Out) data structure - the data we have added most recently will pop out first.

Before we consider the implementation to a data structure, we should first consider its *interface* - or the methods and variables it requires. Based on our previous description, we are going to require the following functions:

```
1 /* type elem must be defined */
2
3 bool is_empty(stack S);      /* O(1), check if stack empty */
4 stack new_stack();           /* O(1), create empty new stack */
5 void push(elem e, stack S); /* O(1), add item on top of stack */
6 elem pop(stack S);           /* O(1), remove item from top of stack */
7 /* requires !is_empty(S); */
```

We want the creation of a new stack, as well as pushing and popping an item to all be constant-time operations, as indicated by the O(1) in the comments. Furthermore, `pop()` should only be possible on a non-empty stack. This feature is *fundamental* to the interface of a stack - that a user can **only** pop from a non-empty stack.

We are being pretty abstract here - we are not writing what type the elements of the stack have to be. Instead we are just going to assume that at the top of the file, or before this file is read, we have already defined the type of `elem`. We say that this implementation is **generic** or **polymorphic** in the type of elements that it can handle. The critical point here is that the *type* of the element is up to the client - or whoever is actually *using* our stack structure.

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

2.1 Using the Stack Interface

Let's play through a simple example to illustrate the idea of a stack and how to use the interface we created above. We write a stack as

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

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

For example:

Stack	Command	Other variables
	`stack S = new_stack()`	
	`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"`

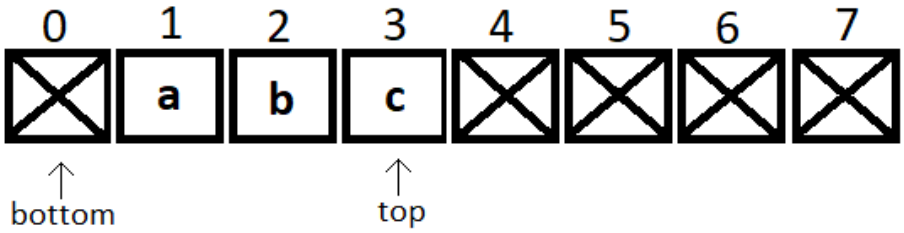
2.2 Stack implementation using arrays

Any programming language is going to come with certain data structures *built in*. Arrays are an example of this in C. But other data structures, like *stacks*, need to be built in to the language using existing language features.

We will get to a more proper implementation of stacks in the next lecture, but for now we will implement a stack in C using arrays, since we should already be *very* familiar with the way they work.

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

can also keep track of the `bottom` of the array, which will be 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.



This implementation of stacks works pretty well, but it seems to have a glaring limitation - our stack can't get any bigger than the array in which we store it, like a pile of books on our desk that cannot grow taller than the ceiling. There are multiple solutions to this problem, but for this lecture, we can be content to work with stacks that have a limited maximum capacity.

2.3 Structs and data structure invariants

Currently our picture of a stack includes two different things: an array containing the struct data and an integer indicating where the `top` is. C has a feature that will allow us to bundle these things up and pass them around together instead of separatly. That structure is called a **struct**.

For example:

```
1 struct stack_header{
2     string[] data;
3     int top;
4     int bottom;
5 };
6 typedef struct stack_header* stack;
```

What this notation means exactly, and especially what the part with the `struct stack_header*` is all about, will be explained in the next lecture. *Hint: These are pointers and it is crucial to understand them, but we are going to 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 post-conditions 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.

So, with this information we could write a function to see if our data structure is indeed a stack as we have designed it. For example:

```
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 - but we will return to this next time.

When we write specification functions, we want to 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. Assertion violations are sometimes unavoidable because we can only check the length of an array inside of the assertion language.

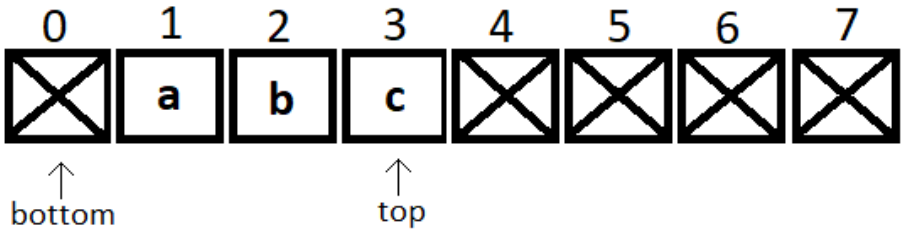
2.4 Checking for emptiness

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

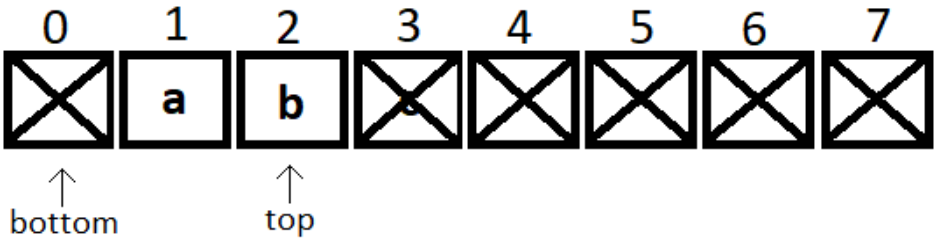
```
bool is_empty(stack S){
    /* requires is_stack(S) */
    return S->top == S->bottom
}
```

2.5 Popping from a stack

To pop and element form 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

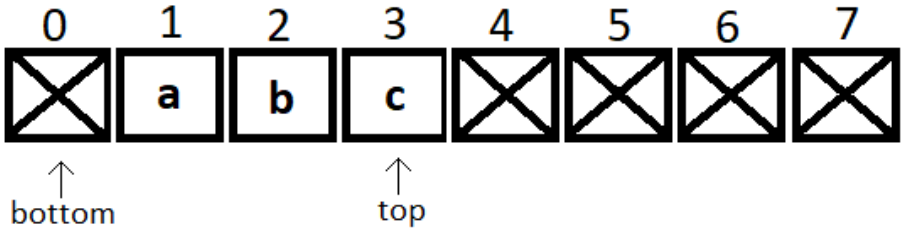


The *c* can still be part of the array at position 3, but it is now a part of the array that we don't care about, which we indicate by putting an *X* over it. In code, popping looks like this:

```
string pop(stack S){
    /* requires is_stack(S) */
    /* requires !is_empty(S); */
    /* ensures is_stack(s); */
    string r = S->data[S->top];
    S->top--;
    return r;
}
```

2.6 Pushing onto a stack

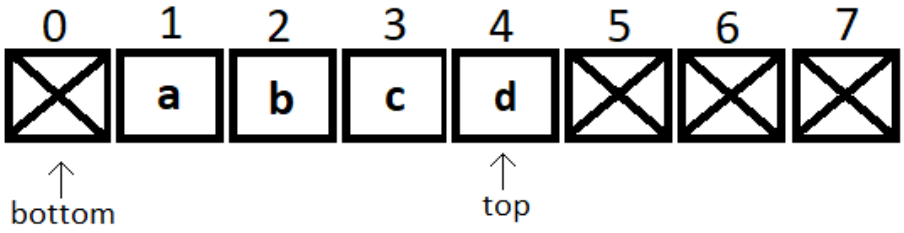
To push an element onto the stack, we increment the `top` field of the stack to reflect that there are more elements on the stack. And then we put the element *d* at the position `top` into the array `S->data`. While this is a pretty simple operation, it is always a good idea to draw a diagram. We go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



to

In code, our push function might look like this:

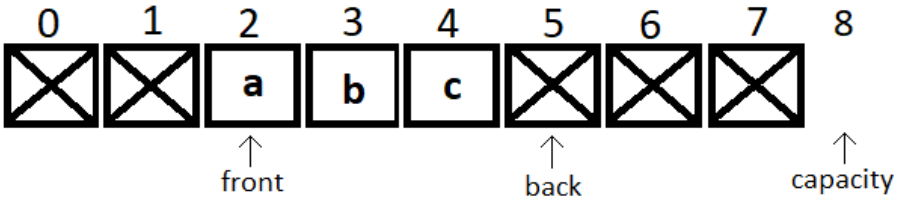
```
void push(stack S, elem e){
    /* requires is_stack(S) */
    /* ensures is_stack(S); */
    S->top++
    S->data[S->top] = e;
    /* we can assume that a "d" was entered and passed as e */
}
```



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 when we remake it with pointers. 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{
    string[] data;
    int top;
    int bottom;
    int capacity;       /* capacity = \length(data);
};
typedef struct struct_header* stack;
```

Giving us an updated picture of our 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 can now 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;
    /* we can assume that a "d" was entered and passed as 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 thay 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.

2.7 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 new_stack(){
    /* 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 = 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, we will need to determine its capacity **ahead** of time. That is, we need to know how many elements at maximum will ever be allowed into 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 using pointers soon!

2.8 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 arrays directly (in fact, we cannot know how it is implemented), so the only thing we can do is `pop` all the elements off the stack. This can be accomplished with a prototypical `while` loop that finishes as soon the stack is empty.

```
int stack_size(stack S){
    int count = 0;
    while(!is_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. Instead, we can maintain a new *temporary stack* T to hold the elements we pop from S. Once we are done counting, we push the elements back onto `S` to repair the damage.

```
int stack_size(stack S){
    int count = 0;
    while(!is_empty(S)){
        push(T, pop(S));
        count++;
    }
    while(!is_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 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 $O(1)$. (Which we do know, from our original interface)

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

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

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

3 Abstraction

An important point about formulating a precise interface to a data structure like a stack is to achieve *abstraction*. This means thay as a client of the data structure we an 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 of the top of the stack without removing it from the stack. Such a function would have the declaration:

```
string peek(stack S){
    /* requires !is_empty(S) */
    . . .
}
```

The first instinct might be to write it as follows:

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

However, this would be **completely wrong**! Let's recall our interface:

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

bool is_empty(stack S);      /* O(1), check if stack empty */
stack new_stack();           /* O(1), create empty new stack */
void push(elem e, stack S); /* O(1), add item on top of stack */
elem pop(stack S);           /* O(1), remove item from top of stack */
/* requires !is_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 using stacks later which would make the above implementation of `peek` break. With the above client-side implementation of `break`, 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 !is_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, this is still a constant-time operation $O(1)$ because 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 !is_empty(S) */
    return S->data[S->top];
}
```

Is this a good implementation? Not quite. First we note that inside teh 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 the 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- adn post-conditions for those. Inside the library, however, we are constantly maniupulating 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*:

```
rlrm peek(stack S){
    /* requires is_stack(S) */
    /* requires !is_empty(S) */
    return S->data[S->top];
}
```

4 Queues

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 to 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, 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 communications 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 *queue* interface:

```
1 /* type elem must be defined */
2
3 bool is_empty(queue Q);                  /* O(1), check if queue is empty */
4 queue new_queue();                       /* O(1), create new empty queue */
5 void enq(queue Q, elem s);               /* O(1), add item at back */
6 elem deq(queue Q);                       /* O(1), remove item from front */
7 /* requires !is_empty(Q) */
```

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

We can write out this interface without commiting 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 abstractions rigorously. In C, it is mostly a matter of adhering to conventions.

4.1 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 in 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 = new_queue()`	
	`enq(Q, "a");`	
"a"	`enq(Q, "b");`	
"a", "b"	`string s = deq(Q);`	`s = "b"`
"b"	`enq(Q, "c");`	`s = "b"`
"b", "c"	`s = deq(Q);`	`s = "b"`
"c"		`s = "c"`

4.2 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 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 array, this assignment makes `C` and `Q` aliased, 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 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 the data from queue `Q` and put it into the new queue `C`.

```
queue C = new_queue();
while(!is_empty(Q)){
    enq(C, deq(Q));
}
/* assert is_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, determine the problem that arises with this implementation.

Queue `C` is now a copy of what used to be in queue `Q` before we started copying. But our copying process was **destructive**! By dequeueing all elements in `Q` to put them into `C`, `Q` has now become empty! In fact, our assertion at the end of the above loop even indicated `is_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 then the copy `C` would be empty. Can you figure out how to copy all the data into `C` and make sure that it also ends up in `Q`? Before you press on, try and determine the 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 = new_queue();
while(!is_empty(Q)){
    string s = deq(Q);
    enq(Q, s);
    enq(C, s);
}
/* assert is_empty(Q) */
```

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

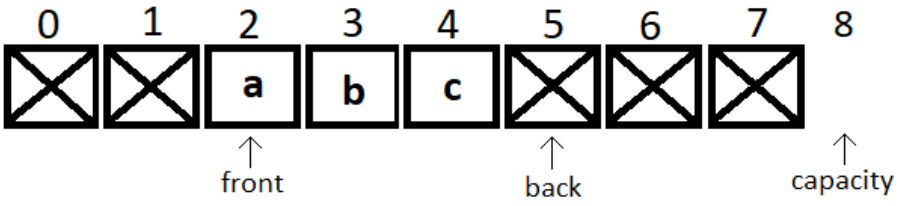
The problem with the above attempt is that the loop will never terminate unless `Q` is empty to begin with. For every element that the loop body dequeues from `Q`, it enqueues one element back into `Q`. That way, `Q` will always have the same number of elements and will never become empty. Therefore, we must go back to our original strategy and first read all elements of `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 the original `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 contain all the elements that `Q` had originally.

```
queue queue_copy(queue Q){
    queue T = new_queue();
    while(! is_empty(Q)){
        enq(T, deq(Q));
    }
    /* assert is_empty(Q) */
    queue C = new_queue();
    while(!is_empty(T)){
        string s = deq(T);
        enq(Q, s);
        enq(C, s);
    }
    /* assert is_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.

4.3 The queue implementation

Here, we will implement the queue using an array, similar to how we implemented stacks earlier.



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

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 the data structure invariants. Making these explicit is important as we think about and write the pre- and post-conditions 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) 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;
}
```

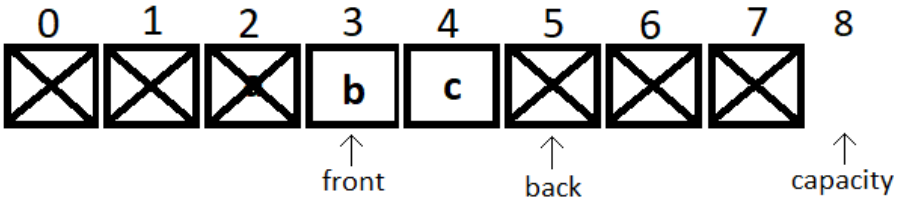
To check if the queue is empty we just compare its front and back. If they are equal, the queue is empty; otherwise it is not. We require that we are being passed a valid queue. Generally, when working with a data structure, we should always require and ensure that its invariants are satisfied in the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

```
bool is_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, we see the following:

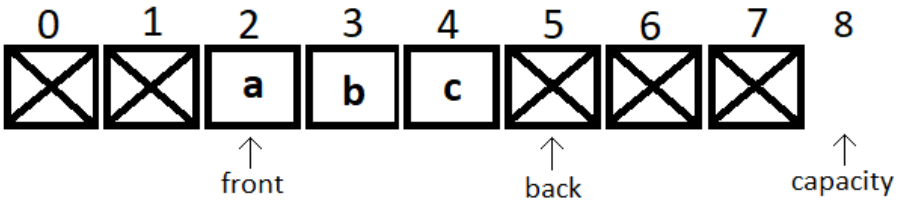
And in code, we get the following:

```
elem deq(queue Q){
    /* requires is_queue(Q) */
    /* requires !is_empty(Q) */
    /* ensures is_queue(Q) */
    elem e = Q->data[Q->front];
```

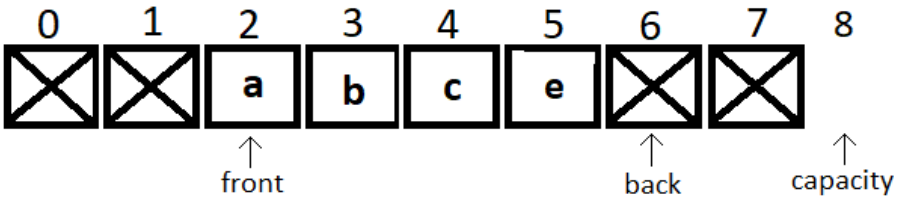



```
6       Q->front++;
7       return e;
8   }
```

To enqueue something, that is, add a new item to the back of the queue, we just write the data 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*. We go from



to



Now that we have our visualization, let's look at the code:

```
1   void enq(queue Q, elem e){
2       /* requires is_queue(Q) */
3       /* ensures is_queue(Q)  */
4       /* ensures !is_empty(Q) */
5       assert(Q->back < Q->capacity - 1);  /* otherwise out of resources */
6       Q->data[Q->back] = e;
7       Q->back++;
8   }
```

To obtain a new empty queue, we allocate a list struct and initialize both `front` and `back` to zero, 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 new_queue(){
    /* ensures is_queue(\result) */
    /* ensures is_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 statements. The only exception is the `new_queue` 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 `new_queue` 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 new_queue();              /* O(1), create an empty queue */
/* ensures is_empty(\result) */

void enq(queue Q, elem s);      /* O(1), add item at back */
/* ensures !is_empty(Q) */
```

Those contracts need to hold for all queue implementations. Why did we decide not to include them? The reason is there are not many situations in which this knowledge about queues is useful, because we rarely dequeue an element right after enqueueing it. This is in contrast to the `/* requires !is_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-emptiness before calling `deq`.

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

5 Bounded versus unbounded stacks and queues

Both the queue and the stack implementations 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 enqueueing elements into the queue or pushing elements onto the stack that requires more space

than the array has in the first place. And if that happens the `enq` or `push` 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 or stack 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 has a way of checking whether there still is space in the stack or 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.

6 Exercises

1. Can you implement a version of stack that does not use `bottom` field in the `struct stack_header`?
2. Consider what would happen if we `pop` an element from the empty stack when contracts are not checked? When does an error arise?
3. The check in `is_queue` required the capacity to be at least 1 to have space for the unused data at `back`. Can we actually enqueue data into a queue with `capacity == 1`. If not, what do we need to change in `is_queue` to make sure the queue can hold at least one element before it runs out of space?
4. Our queue implementation wastes a lot of space unnecessarily. After enqueueing 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 enqueueing elements? How do you need to change the implementation of `enq` and `deq` for that purpose?

5. 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?

6. 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?