Lecture Notes on
Stacks & Queues

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

Lecture 10
September 25, 2014

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 an *abstract data type* in C0, like the abstract `arr` type of arrays that we discussed in the previous lectures). Today we will not talk about the implementation of arrays; we will implement them in the next lecture.

Relating this to our learning goals, we have

Computational Thinking: We illustrate the power of *abstraction* by considering new data structures from the client side.

Algorithms and Data Structures: Queues and stacks are two important data structure to understand.

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:

```
bool stack_empty(stack S);    /* O(1), check if stack empty */
stack stack_new();            /* O(1), create new empty stack */
void push(stack S, string e); /* O(1), add item on top of stack */
string 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.

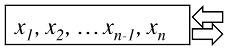
One thing to observe is that there's nothing special about the `string` type here. It would be nice to have a data structure that was *generic*, and able to work with strings, integers, arrays, and so on, but we will discuss that possibility later.

3 Using the Stack Interface

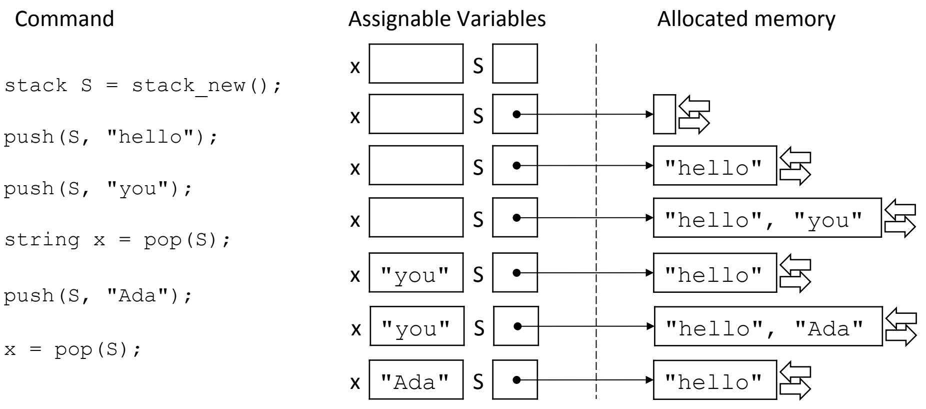
We play through some simple examples to illustrate the idea of a stack and how to use the interface above. We write a stack as

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

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. If we're feeling artistic, we can draw stacks with arrows to emphasize that we're pushing and popping from the top:



Here is a more complex example, showing the effect of several steps on the state of assignable variables and allocated memory, where the stack data structure resides:



```
stack S = stack_new();
push(S, "hello");
push(S, "you");
string x = pop(S);
push(S, "Ada");
x = pop(S);
```

Remember that we think of the assignable `S` like a pointer or an array: it is not literally an arrow but a number representing the address of the in-memory representation of the stack.

4 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);
  ;
```

If we knew how stacks were implemented, we might be able to implement, as clients of the stack data structure, something like this:

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

However, *this would be completely wrong*. As clients of the stack data structure, we only know about the functions provided by the interface. However, it is possible to implement the peek operation correctly *without violating the abstraction*!

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;
}
```

Depending on the implementation of stacks, this might be less efficient than a library-side implementation of peek. However, as long as `push` and `pop` are still a constant-time operations, `peek` will still be constant time ($O(1)$).

5 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);
```

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 ($O(1)$).

A library-side implementation of `stack_size` can be done in $O(1)$, but we won't consider that today.

6 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, string s); /* O(1), add item at back */
string 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.

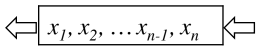
Again, we can write out this interface without committing to an implementation of queues. In particular, the type `queue` remains *abstract* in the sense that we have not given its definition. This is important so that different implementations of the functions in this interface can choose different 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.

7 Using the Queue Interface

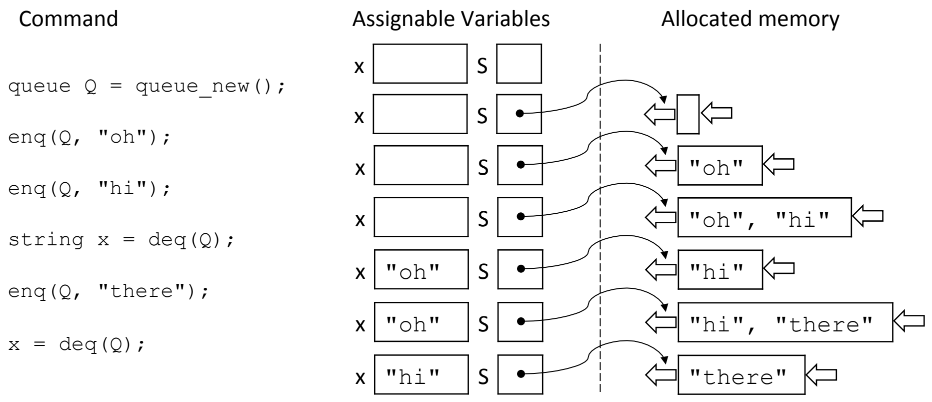
We play through some simple examples to illustrate the idea of a queue and how to use the interface above. We write a queue as

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

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. If we want to emphasize this, we can draw queues like this:



Here's a trace of the queues in action:



8 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 stacks, 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 stacks, 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.