Lecture Notes on Data Structure Implementation

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

Lecture 11
October 1, 2013

1 Introduction

In the last lecture we talked about using pointers and structs to implement *linked lists*. In this lecture, we'll talk about using linked lists as an implementation of the stacks and queues we introduced a week ago. The linked list implementation of stacks and queues allows us to handle lists of any length.

Relating this to our learning goals, we have

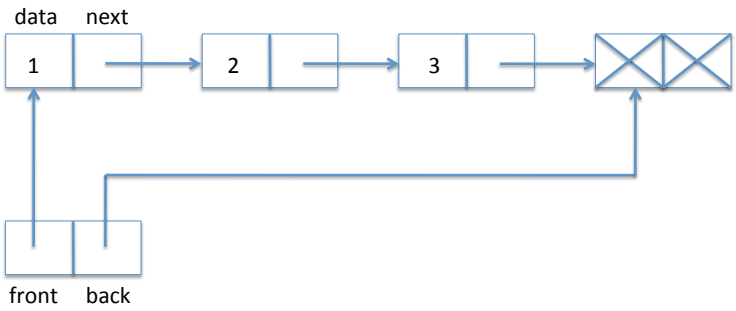
Computational Thinking: We emphasize the importance of *abstraction* by producing a second implementation of the stacks and queues we introduced in the notes for lecture 9.

Algorithms and Data Structures: We utilize *linked lists*, and also discuss an algorithm for circularity checking.

Programming: We will again see *structs* and *pointers*.

2 Queues with Linked Lists

Here is a picture of the queue data structure the way we envision implementing it, where we have elements 1, 2, and 3 in the queue.



A queue is implemented as a struct with a `front` and `back` field. The `front` field points to the front of the queue, the `back` field points to the back of the queue. We need these two pointers so we can efficiently access both ends of the queue, which is necessary since dequeue (front) and enqueue (back) access different ends of the list.

In the array implementation of queues, we kept the back as one greater than the index of the last element in the array. In the linked-list implementation of queues, we use a similar strategy, making sure the `back` pointer points to one element past the end of the queue. Unlike arrays, there must be something in memory for the pointer to refer to, so there is always one extra element at the end of the queue which does not have valid data or `next` pointer. We have indicated this in the diagram by writing `X`.

The above picture yields the following definition.

```
struct queue_header {
  list* front;
  list* back;
};
typedef struct queue_header* queue;
```

We call this a *header* because it doesn't hold any elements of the queue, just pointers to the linked list that really holds them. The type definition allows us to use `queue` as a type that represents a *pointer to a queue header*. We define it this way so we can hide the true implementation of queues from the client and just call it an element of type `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 if we follow `front` and then move down the linked list we eventually arrive at `back`. We call this a *list segment*. We also want both `front` and `back` not to be `NULL` so it conforms to the picture, with one element already allocated even if the queue is empty.

```
bool is_queue(queue Q) {
  if (Q == NULL) return false;
  if (Q->front == NULL) return false;
  if (Q->back == NULL) return false;
  if (!is_segment(Q->front, Q->back)) return false;
  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 queue_empty(queue Q)
//@requires is_queue(Q);
{
  return Q->front == Q->back;
}
```

To obtain a new empty queue, we just allocate a list struct and point both front and back of the new queue to this struct. We do not initialize the list element because its contents are irrelevant, according to our representation. 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);
  list* p = alloc(struct list_node);
  Q->front = p;
  Q->back = p;
  return Q;
}
```

Let's take one of these lines apart. Why does

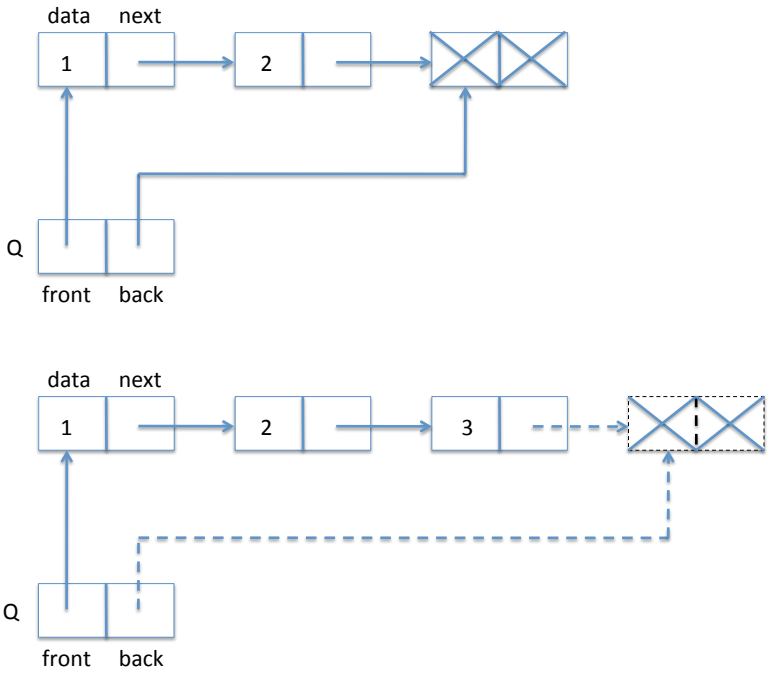
```
queue Q = alloc(struct queue_header);
```

make sense? According to the definition of `alloc`, we might expect

```
struct queue_header* Q = alloc(struct queue_header);
```

since allocation returns the address of what we allocated. Fortunately, we defined `queue` to be a short-hand for `struct queue_header*` so all is well.

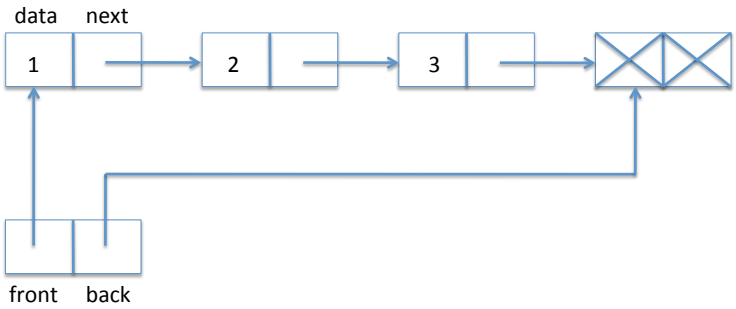
To enqueue something, that is, add a new item to the back of the queue, we just write the data (here: a string) into the extra element at the back, create a new back element, and make sure the pointers updated correctly. You should draw yourself a diagram before you write this kind of code. Here is a before-and-after diagram for inserting `"3"` into a list. The new or updated items are dashed in the second diagram.

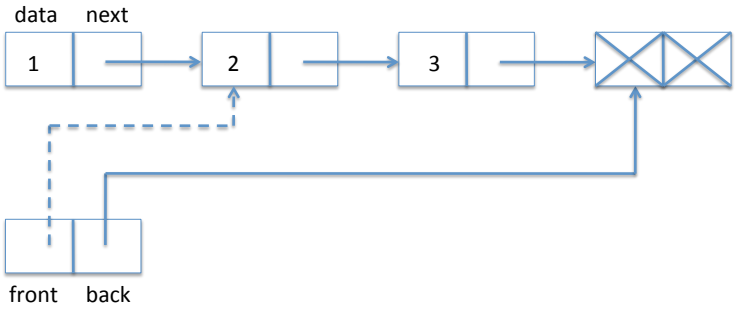


Another important point to keep in mind as you are writing code that manipulates pointers is to make sure you perform the operations in the right order, if necessary saving information in temporary variables.

```
void enq(queue Q, string s)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{
  list* p = alloc(struct list);
  Q->back->data = s;
  Q->back->next = p;
  Q->back = p;
}
```

Finally, we have the dequeue operation. For that, we only need to change the front pointer, but first we have to save the dequeued element in a temporary variable so we can return it later. In diagrams:





And in code:

```
string deq(queue Q)
//@requires is_queue(Q);
//@requires !queue_empty(Q);
//@ensures is_queue(Q);
{
  string s = Q->front->data;
  Q->front = Q->front->next;
  return s;
}
```

Let's verify that the our pointer dereferencing operations are safe. We have

```
  Q->front->data
```

which entails two pointer dereference. We know `is_queue(Q)` from the precondition of the function. Recall:

```
bool is_queue(queue Q) {
  if (Q == NULL) return false;
  if (Q->front == NULL) return false;
  if (Q->back == NULL) return false;
  if (!is_segment(Q->front, Q->back)) return false;
  return true;
}
```

We see that `Q->front` is okay, because by the first test we know that `Q != NULL` is the precondition holds. By the second test we see that both `Q->front` and `Q->back` are not null, and we can therefore dereference them.

We also make the assignment `Q->front = Q->front->next`. Why does this preserve the invariant? Because we know that the queue is not empty (second precondition of `deq`) and therefore `Q->front != Q->back`. Because `Q->front` to `Q->back` is a valid non-empty segment, `Q->front->next` cannot be null.

An interesting point about the dequeue operation is that we do not explicitly deallocate the first element. If the interface is respected there cannot be another pointer to the item at the front of the queue, so it becomes *unreachable*: no operation of the remainder of the running programming could ever refer to it. This means that the garbage collector of the C0 runtime system will recycle this list item when it runs short of space.