Lecture Notes on Lists and Queues

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 7
September 14, 2010

1 Introduction

In this lecture we first review *amortized analysis* with another example, this time studying binary counters. Then we will introduce *queues* as a data structure and *linked lists* that underly their implementation. In order to implement them we need *recursive types*, which are quite common in the implementation of data structures.

2 Amortized Analysis of Binary Counters

A binary counter is a sequence of bits, representing a number in binary form, that can be incremented. We do not presume a fixed precision, but allow the counter to be extended in case it overflows. To simplify the analysis we assume we start at 0 and count upwards.

We are interested in measuring the cost of incrementing the counter in terms of the number of bits we have to flip. For example, incrementing $0_{[2]}$ to $1_{[2]}$ requires 1 bit flip, incrementing $1_{[2]}$ to $10_{[2]}$ requires 2: we complement the lowest bit, and then we add another digit on the left, which we consider the same as flipping a (hypothetical) 0 to 1.

What is the cost of a single increment? It looks mostly like constant time, but not quite. For example, when the number is $11111111_{[2]}$ then the increment will take 9 bit flips, which is equal to the number of bits we have (8), plus 1. In general, after we incremented $n = 2^k$ times, we will have $k + 1$ bits in our representation. So the maximal number of bit flips during a single increment is $O(k) = O(\log(n))$.

But what is the cost of incrementing a counter n times? Naively multiplying we might think the cost is $O(n * \log(n))$, but we might suspect it could be less, like $O(n)$, if we analyze the situation more carefully. A perfect example where we can try an amortized analysis!

Let's do some counting of bit flips, starting at 0, to see if we can form a conjecture. Since we are working with binary numbers throughout we omit the base $_{[2]}$.

number	op	flips
0	inc	1
1	inc	2
10	inc	1
11	inc	3
100	inc	1
101	inc	2
110	inc	1
111	inc	4
1000	inc	1
1001		

Looking at the structure we see at on each increment, there is exactly one flip from 0 to 1, and some variable number of flips from 1 to 0. Let's separate these out.

number	op	$0 \to 1$	$1 \to 0$
0	inc	1	0
1	inc	1	1
10	inc	1	0
11	inc	1	2
100	inc	1	0
101	inc	1	1
110	inc	1	0
111	inc	1	3
1000	inc	1	0
1001			

The question is how to account for the $1 \to 0$ flips. The idea behind amortized analysis here is to set aside a constant number of tokens on each increment, so that we have enough saved up to perform the $1 \to 0$ flips. Recall that each token represents an operation we *could have* performed and stayed within constant time for each of the n increments, but did not. Therefore, we can perform these operations later without violating the $O(n)$ bound for the total number of n increments.

The crucial insight here is that on each increment we create exactly one new 1 that was not in the previous number, namely the $0 \rightarrow 1$ flip. So, if we put aside one token for every increment, we should have one token for every digit 1 in the number. That's enough to pay for the flips $1 \rightarrow 0$, because the number of these flips on each increment is bounded by the number of 1's.

Let's run through this again and count the number of tokens we we have. We save one on each operation

tokens before inc	number	op	$0 \rightarrow 1$	$1 \rightarrow 0$	tokens saved	tokens spent
0	0	inc	1	0	$+1$	0
1	1	inc	1	1	$+1$	-1
1	10	inc	1	0	$+1$	0
2	11	inc	1	2	$+1$	-2
1	100	inc	1	0	$+1$	0
2	101	inc	1	1	$+1$	-1
2	110	inc	1	0	$+1$	0
3	111	inc	1	3	$+1$	-3
1	1000	inc	1	0	$+1$	0
2	1001					

From this table we form the conjecture that if we save a token on every increment, and we spend a token for every $1 \rightarrow 0$ flip during an increment, then we always have exactly as many tokens as 1's. Since number of 1's can never become negative, the number of tokens never becomes negative.

To prove this, we note two things. First, when we start we have no tokens saved and also no 1's in the number, so the invariant holds.

Assume after some number of steps we have k 1's. If the rightmost bit is a 0, we flip it to become a 1, which is the only operation we perform. We also add one token, which accounts for the additional 1 in the number. If the rightmost r bits are 1's, we flip them all to 0's (costing r tokens) and the next 0 to a 1 (gaining one token). So we spend r tokens and gained one, so we have $k - r + 1$. But this is also the number of 1's, because we started with k, performed r flips from 1 to 0 and one from 0 to 1.

Taking these two together, we know the invariant that the number of tokens equals the number of 1s is true at the beginning and preserved by every increment. It therefore must always be true.

The total number of operations for n increments is the sum of the $0 \rightarrow 1$ flips (which is n) and $1 \rightarrow 0$ flips. The latter is bounded by the number of tokens, of which we put aside n over n operations. It is therefore $O(n)$ and the total number of bit flips is $O(2 * n) = O(n)$.

3 Linked Lists

Linked lists are a common alternative to arrays in the implementation of data structures. Each item in a linked list contains a data element of some type and a *pointer* to the next item in the list. It is easy to insert and delete elements in a linked lists, which is not a natural operation on arrays. On the other hand access to an element in the middle of the list is usually $O(n)$, where n is the length of the list.

An item in a linked list consists of a struct containing the data element and a pointer to another linked list. This gives rise to the following definition:

```
struct list {
  string data;
  struct list* next;
};
typedef struct list* list;
```

This definition is an example of a *recursive type*. A struct of this type contains a pointer to another struct of the same type, and so on. We usually use the special element of type `t*`, namely `NULL`, to indicate that we have reached the end of the list. Sometimes (as will be the case for queues introduced next), we can avoid the explicit use of `NULL` and obtain more elegant code. The type definition is there to create the type name `list`, which stands for a pointer to a `struct list`.

There are some restriction on recursive types. For example, a declaration such as

```
struct infinite {
  int x;
  struct infinite next;
}
```

would be rejected by the C0 compiler because it would require an infinite amount of space. The general rule is that a struct can be recursive, but the recursion must occur beneath a pointer or array type, whose values are addresses. This allows a finite representation for values of the struct type.

We don't introduce any general operations on lists; let's wait and see what we need where they are used. Linked lists as we use them here are a *concrete type* which means we do *not* construct an interface and a layer of abstraction around them. When we use them, we know about and exploit their precise internal structure. This is contrast to *abstract types* such as unbounded arrays or queues or stacks (see next lecture) whose implementation is hidden behind an interface, exporting only certain operations. This limits what clients can do, but it allows the author of a library to improve its implementation without having to worry about breaking client code. Concrete types are cast into concrete once and for all.

4 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 in a book as we discussed in last lecture, it would be natural to store the 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.

Before we consider the implementation to 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:

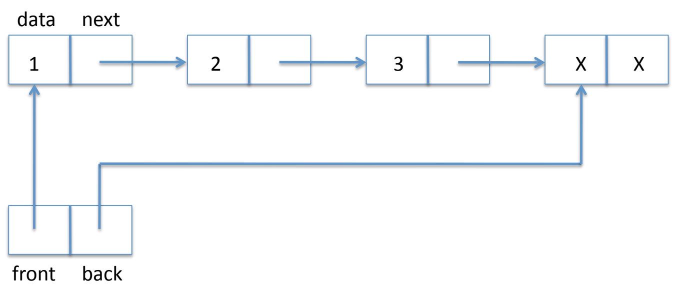
```
typedef struct queue* queue;

bool is_empty(queue Q);          /* O(1), check is queue is empty */
queue q_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 */
```

We can write out this interface without committing to an implementation of queues. After the type definition we know only that a queue will be implemented as a pointer to a `struct queue`.

5 The Queue Implementation

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. In arrays, we often work with the length which is one greater than the index of the last element in the array. In 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 {
  list front;
  list back;
};
```

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) {
  return Q->front != NULL && Q->back != NULL
    && is_segment(Q->front, Q->back);
}
```

Next, the code for checking whether two pointers delineate a list segment. When both `start` and `end` are `NULL`, we consider it a valid list segment, even though this will never come up for queues. It is a common code pattern for working with linked lists and similar data representation to have a pointer variable, here called p, that is updated to the next item in the list on each iteration until we hit the end of the list.

```
bool is_segment(list start, list end) {
  list p = start;
  while (p != end) {
    if (p == NULL) return false;
    p = p->next;
  }
  return true;
}
```

Here we stop in two situations: if $p = null$, then we cannot come up against end any more because we have reached the end of the list and we return `false`. The other situation is if we find end, in which case we return `true` since we have a valid list segment.

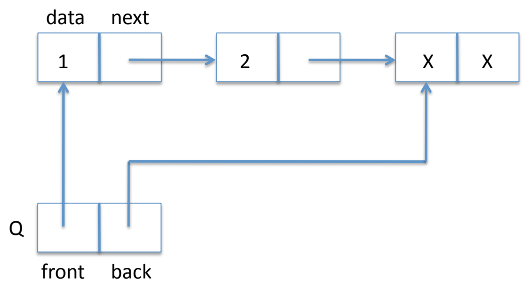
To check if the queue is empty, we just compare its front and back. If they are equal, the queue is empty; otherwise it is not. We require that we are being passed a valid queue. Generally, when working with a data structure, we should always require and ensure that its invariants are satisifed in the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

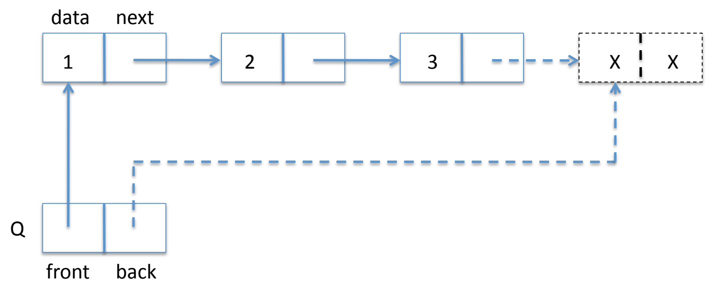
```
bool is_empty(queue Q)
//@requires is_queue(Q);
{
  return Q->front == Q->back;
}
```

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.

```
queue q_new()
//@ensures is_queue(\result);
//@ensures is_empty(\result);
{
  queue Q = alloc(struct queue);
  list l = alloc(struct list);
  Q->front = l;
  Q->back = l;
  return Q;
}
```

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.



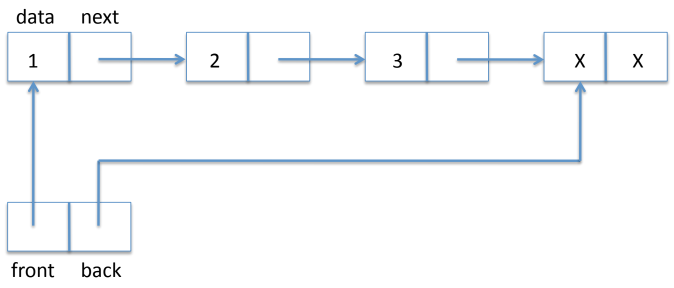


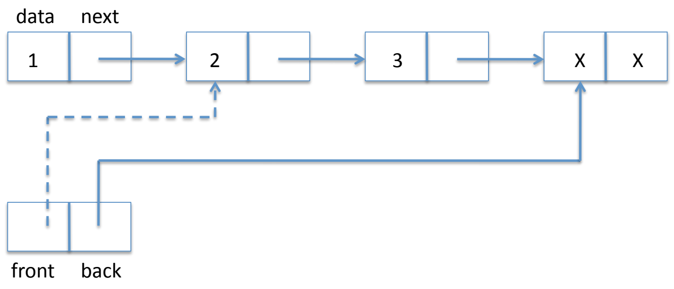
Another important point to keep in mind as you are writing code that manipulates pointers is to make sure you perform the operations in the right order, if necessary saving information in temporary variables.

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

The invariant marked *internal* is one that only makes sense in the context of the present implementation, but would not be meaningful to a client.

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 !is_empty(Q);
```

```
//@ensures is_queue(Q);
//@ensures string_equal(\old(Q->front->data), \result); /* internal */
{ assert(!is_empty(Q), "cannot deq from empty queue");
  { string s = Q->front->data;
    Q->front = Q->front->next;
    return s;
  }
}
```

We included an explicit `assert` statement with an error message so that if `deq` is called with an empty queue we can issue an appropriate error message. Unlike the precondition of the function, this will *always* be checked, which is good practice when writing library code that might be called incorrectly from the outside.

The

We do not always check whether the given queue is valid for two reasons. First, it takes $O(n)$ time when there are n elements in the queue, so dequeuing and enqueuing would no longer be constant time. Second, if the client respects the interface and manipulates the data structure only through the given interface, then it should not be possible to construct an invalid queue. On the other hand, it is perfectly possible to construct an empty queue and mistakenly hand it to the `deq` function, so we check this condition explicitly.

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.