Lecture Notes on Queues

15-122: Principles of Imperative Computation
Frank Pfenning and Jamie Morgenstern

Lecture 9
February 14, 2012

1 Introduction

In this lecture we 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 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 list, 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_node {
  string data;
  struct list_node* next;
};
typedef struct list_node* 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.

Lists can be formed for an arbitrary element type and work the same way, whether the element type is `string` or `int` or whatever. In order to reflect this in our implementation, we define an alias `elem` for `string` as follows

```
typedef string elem;
```

Then we refer to `elem` in the implementation. So we modify the above definition of the list data type to refer to `elem` instead of `string`.

```
struct list_node {
  elem data;
  struct list* next;
};
typedef struct list_node* list;
```

As a side remark, note that more complex programming languages provide mechanisms to understand `elem` in a way that makes it possible to implement generic data structures.

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 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.

3 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 exactly 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_node* queue;

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 */
```

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`.

4 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_node {
  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) {
  if (Q == NULL) return false;
  if (Q->front == NULL || Q->back == NULL) return false;
  return 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. This function may not terminate if the list contains a cycle. We will address this issue in the next lecture; for now we assume all lists are acyclic.

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 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.

```
queue queue_new()
//@ensures is_queue(\result);
//@ensures queue_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 of type `elem` (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, elem s)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{
  list l = alloc(struct list);
  Q->back->data = s;
  Q->back->next = l;
  Q->back = l;
}
```

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:

```
elem deq(queue Q)
//@requires is_queue(Q);
//@requires !queue_empty(Q);
//@ensures is_queue(Q);
{
  assert(!queue_empty(Q));
  elem 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.

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.

Exercises

Exercise 1 *What happens when we swap the order of the lines in the* `enq` *function and why?*

Exercise 2 *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?*

5 Stack Interface

Stacks are similar to queues in that we can insert and remove items. But we remove them from the same end that we add them, which makes stacks a LIFO (Last In First Out) data structure.

Here is our interface

```
/* type elem must be defined */
typedef struct stack_node* stack;

bool stack_empty(stack S);      /* O(1) */
stack stack_new();              /* O(1) */
void push(stack S, elem e);     /* O(1) */
elem pop(stack S);              /* O(1) */
```

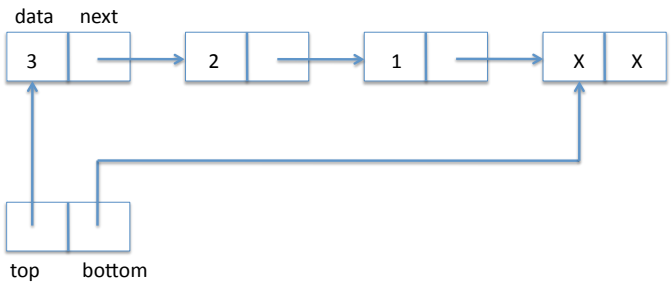
We want the creation of a new (empty) stack as well as pushing and popping an item all to be constant-time operations.

We are being slightly more abstract here than in the case of queues in that 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.

6 Stack Implementation

The idea is to reuse linked lists. We follow the basic structure of our queue implementation, except that we read off elements from the same end that we write them to. We call the pointer to this end `top`. Since we do not perform operations on the other side of the stack, we do not necessarily need a

pointer to the other end. For structural reasons, and in order to identify the similarities with the queue implementation, we still decide to remember a pointer `bottom` to the bottom of the stack. 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. A typical stack then has the following form:



Note that the end of the linked list (i.e., the pointer `bottom->next`) is marked with the special `NULL` pointer that cannot be dereferenced. We define:

```
struct list_node {
  elem data;
  struct list_node* next;
};
typedef struct list_node* list;


struct stack_node {
  list top;
  list bottom;
};

typedef struct stack_node* stack;
```

To test if some structure is a valid stack, we only need to check that the list starting at `top` ends in `bottom`, which is the same as checking that this is a list segment (as introduced in the last lecture).

```
bool is_stack (stack S) {
  if (S == NULL) return false;
```

```
  if (S->top == NULL || S->bottom == NULL) return false;
  return is_segment(S->top, S->bottom);
}
```

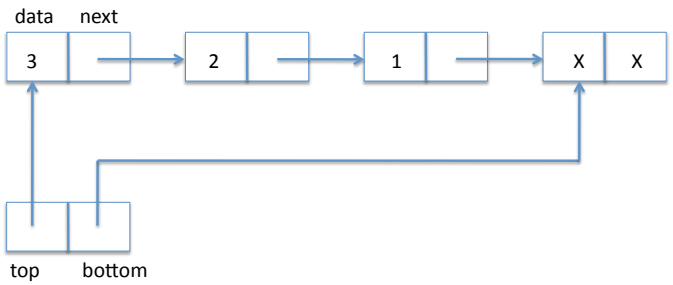
To check if the stack is empty, we only need to check whether `top` and `bottom` point to the same element.

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

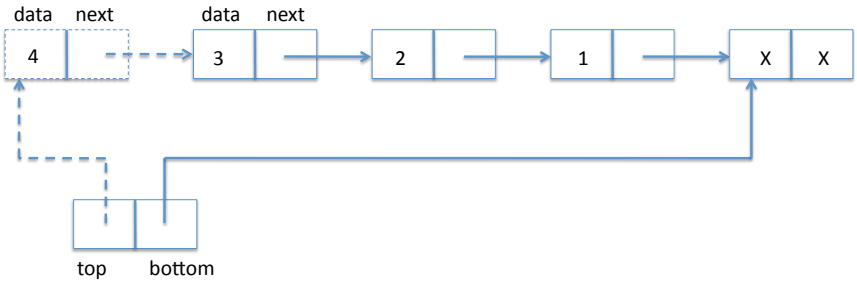
For creating a new stack, we allocate a list element for `top` and `bottom` to point to.

```
stack stack_new()
//@ensures is_stack(\result);
//@ensures stack_empty(\result);
{
  stack S = alloc(struct stack);
  list l = alloc(struct list); /* does not need to be initialized! */
  S->top = l;
  S->bottom = l;
  return S;
}
```

To push an element onto the stack, we create a new list item, set its data field and then its next field to the current top of the stack. Finally, we need to update the `top` field of the stack to point to the new list item. While this is simple, it is still a good idea to draw a diagram. We go from



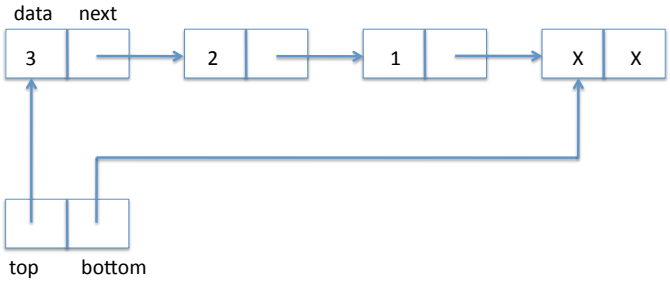
to



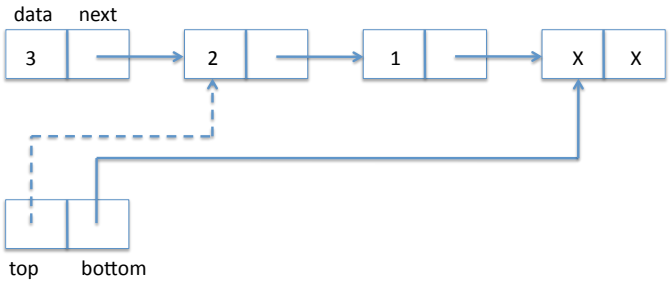
In code:

```
void push(stack S, elem e)
//@requires is_stack(S);
//@ensures is_stack(S);
{
  list l = alloc(struct list);
  l->data = e;
  l->next = S->top;
  S->top = l;
}
```

Finally, to pop an element from the stack we just have to move the top pointer to follow the next pointer from the top of the stack. As in the case of dequeuing an element from the previous lecture, the list item that previously constituted the top of the stack will no longer be accessible and be garbage collected as needed by the runtime system. We go from



to



In code:

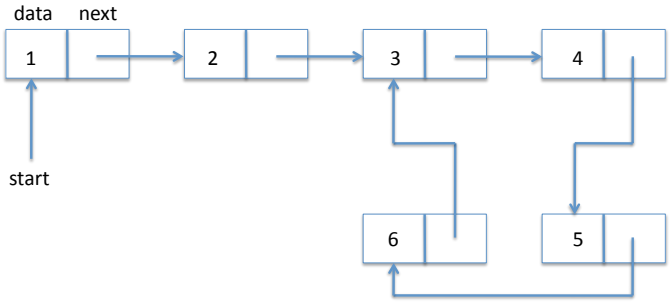
```
elem pop(stack S)
//@requires is_stack(S);
//@requires !stack_empty(S);
//@ensures is_stack(S);
{
  assert(!stack_empty(S));
  elem e = S->top->data;
  S->top = S->top->next;
  return e;
}
```

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

7 Detecting Circularity

Checking whether a stack or a queue satisify their data structure invariant raises an interesting question: what if, somehow, we created a list that

contains a cycle, such as



In that case, following next pointers until we reach `NULL` actually never terminates. The program for checking a segment will get into an infinite loop.

In general, contracts should terminate and have no effects. It is marginally acceptable if a contract diverges, because it will not incorrectly claim that the contract it satisfied, but it would clearly be better if it explicitly rejected a circular list. But how do we check that? Before you read on, you should seriously think about the problem, like our class did in lecture.

Here is the original `is_segment` predicate.

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

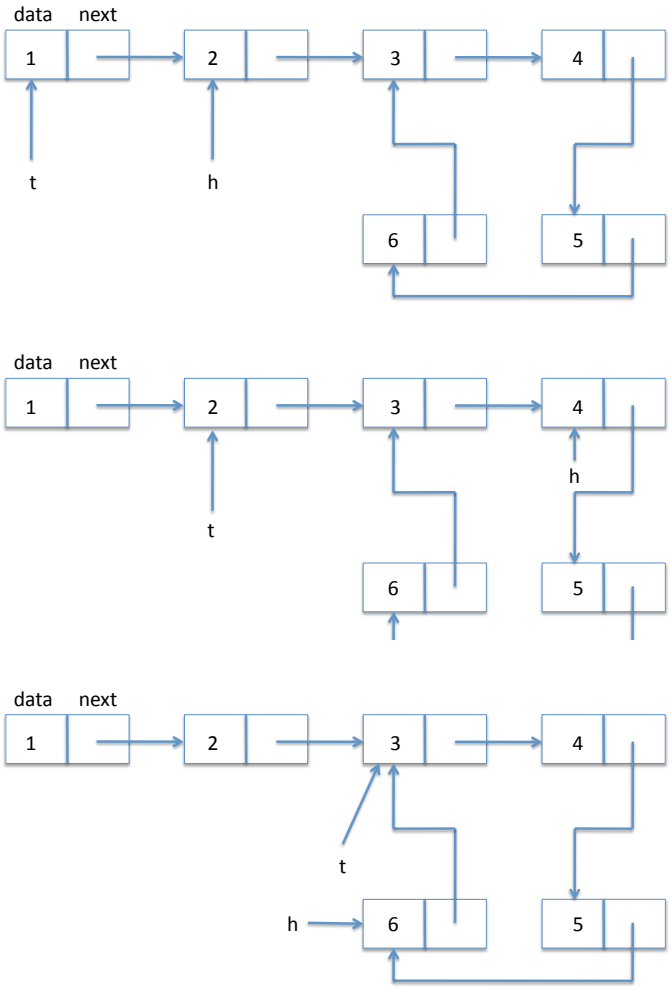
One the simplest solutions proposed in class keeps a copy of the `start` pointer. Then when we advance p we run through an auxiliary loop to check if the next element is already in the list. The code would be something like

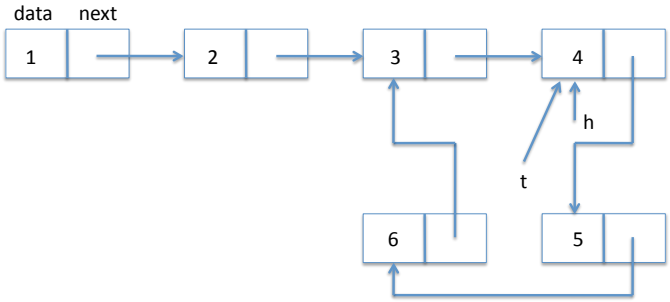
```
bool is_segment(list start, list end)
{ list p = start;
  while (p != end) {
    if (p == NULL) return false;
    { list q = start;
      while (q != p) {
        if (q == p->next) return false; /* circular */
        q = q->next;
      }
    }
    p = p->next;
 }
 return true;
}
```

Unfortunately this solution requires $O(n^2)$ time for a list with n elements, whether it is circular or not.

Again, consider if you can find a better solution before reading on.

The idea for a more efficient solution is to create *two* pointers, let's name them t and h. t traverses the list like the pointer p before, in single steps. h, on the other hand, skips two elements ahead for every step taken by p. If the slower one t ever gets into a loop, the other pointer h will overtake it from behind. And this is the only way that this is possible. Let's try it on our list. We show the state of t and h on every iteration.





In code:

```
bool is_circular(list l)
{ if (l == NULL) return false;
  list t = l;       // tortoise
  list h = l->next; // hare
  while (t != h)
    { if (h == NULL || h->next == NULL) return false;
      t = t->next;
      h = h->next->next;
    }
  return true;
}
```

This algorithm has been called the *tortoise and the hare* and is due to Floyd. We have chosen t to stand for *tortoise* and h to stand for *hare*.

A few points about this code: in the condition inside the loop we exploit the short-circuiting evaluation of the logical or '`||`' so we only follow the next pointer for h when we know it is not `NULL`. Guarding against trying to dereference a `NULL` pointer is an extremely important consideration when writing pointer manipulation code such as this. The access to `h->next` and `h->next->next` is guarded by the `NULL` checks in the if statement. But what about the dereference of `t` in `t->next`? Before you turn the page: can you figure it out?

One solution would be to add another if statement checking whether `t==NULL`. That is unnecessarily inefficient, though, because the tortoise t, being slower than the hare h, will never follow pointers that the hare has not followed already successfully. In particular, they cannot be `NULL`. How do we represent this information? The loop invariant `t != NULL` may come to mind, but it is hard to prove that it actually is a loop invariant, because, for all we know so far, `t->next` may be `NULL` even if `t` is not.

The crucial loop invariant that is missing is the information that the tortoise will be able to travel to the current position of the hare by following next pointers. Of course, the hare will have moved on then, but at least there is a chain of next pointers from the current position of the tortoise to the current position of the hare. This is represented by the following loop invariant in `is_circular`:

```
bool is_circular(list l)
{ if (l == NULL) return false;
  list t = l;       // tortoise
  list h = l->next; // hare
  while (t != h)
    //@loop_invariant is_segment(t, h);
    { if (h == NULL || h->next == NULL) return false;
      t = t->next;
      h = h->next->next;
    }
  return true;
}
```

As an exercise, you should prove this loop invariant. How would this invariant imply that t is not NULL? The key insight is that the loop invariant ensures that there is a linked list segment from t to h, and the loop condition ensures $t \neq h$. Thus, if there is a link segment from t to a different h, the access `t->next` must work. We could specify this formally by enriching the contract of `is_segment`, which is what you should do as an exercise.

This algorithm has complexity $O(n)$. An easy way to see this was suggested by a student in class: when there is no loop, the hare will stumble over `NULL` after $O(n)$ steps. If there is a loop, then consider the point when the tortoise enters the loop. At this point, the hare must already be somewhere in the loop. Now for every step the tortoise takes in the loop the hare takes two, so on every iteration it comes one closer. The hare will catch the tortoise after at most half the size of the loop. Therefore the overall complexity of $O(n)$: the tortoise will not complete a full trip around the loop.

Exercises

Exercise 3 *Extend the stack interface to return the element on the top of the stack, without changing the stack, but requiring that the stack be nonempty.*

Exercise 4 *Stacks are usually implemented with just one pointer, to the top of the stack. Rewrite the implementation in this style, dispensing with the* `bottom` *pointer, terminating the list with* `NULL` *instead.*

Exercise 5 *Prove the loop invariant of the tortoise and hare algorithm. Prove why all pointer accesses are going to be valid (no dereference of* `NULL`*).*

Exercise 6 *Because the hare is faster than the tortoise, the tortoise and hare algorithm finds the end of a (non-cyclic) list faster, i.e., with less iterations than the* `is_segment` *algorithm. Can we make the algorithm go faster by making the hare speed up or? Or, better yet, can we replace the hare by a fox that moves at speed 3 instead of 2? Prove the invariants for this algorithm and either prove or disprove whether it works correctly. In either case (prove or disprove), identify all steps in our correctness argument that are different for the fox compared to the hare.*