Lecture Notes on Linked Lists

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

Lecture 11
September 30, 2014

1 Introduction

In this lecture we discuss the use of *linked lists* to implement the stack and queue interfaces that were introduced in the last lecture. The linked list implementation of stacks and queues allows us to handle lists of any length.

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 are not natural operations on arrays, since arrays have a fixed size. 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. In C0 we have to commit to the type of element that is stored in the linked list. We will refer to this data as having type `elem`, with the expectation that there will be a type definition elsewhere telling C0 what `elem` is supposed to be. Keeping this in mind ensures that none of the code actually depends on what type is chosen. These considerations give rise to the following definition:

```
struct list_node {
  elem 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 our use of linked lists in stacks and queues), 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 `struct list_node`, so that a pointer to a list node will be `list*`.

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

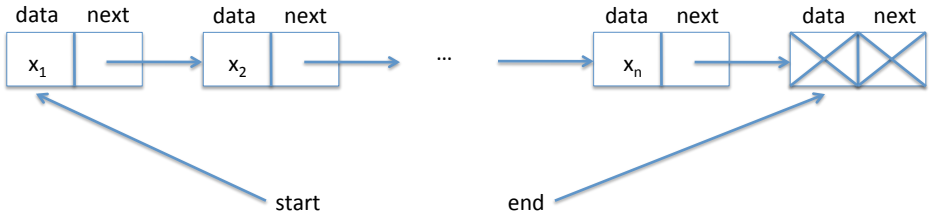
```
struct infinite {
  int x;
  struct infinite next;
}
```

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.

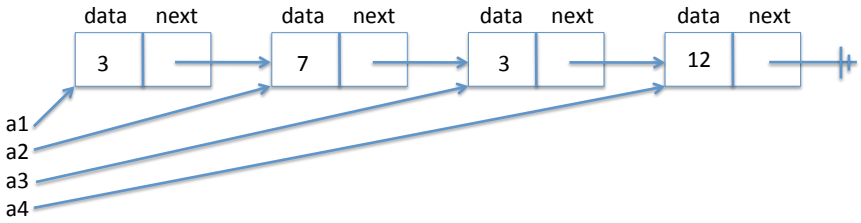
We don't introduce any general operations on lists; let's wait and see what we need where they are used. Linked lists as we use them here are a *concrete type* which means we do *not* construct an interface and a layer of abstraction around them. When we use them we know about and exploit their precise internal structure. This is contrast to *abstract types* such as 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 List segments

A lot of the operations we'll perform in the next few lectures are on *segments* of lists: a series of nodes starting at *start* and ending at *end*.



This is the familiar structure of an "inclusive-lower, exclusive-upper" bound: we want to talk about the data in a series of nodes, ignoring the data in the last node. That means that, for any non-NULL list node pointer `l`, a segment from l to l is empty (contains no data). Consider the following structure:



According to our definition of segments, the data in the segment from *a1* to *a4* is the sequence $3, 7, 3$, the data in the segment from *a2* to *a3* contains the sequence 7, and the data in the segment from *a1* to *a1* is the empty sequence. Note that if we compare the pointers *a1* and *a3* C0 will tell us they are *not equal* – even though they contain the same data they are different locations in memory.

Given an inclusive beginning point *start* and an exclusive ending point *end*, how can we check whether we have a segment from *start* to *end*? The simple idea is to follow *next* pointers forward from *start* until we reach *end*. If we reach NULL instead of *end* then we know that we missed our desired endpoint, so that we do not have a segment. (We also have to make sure that we say that we do not have a segment if either *start* or *end* is NULL, as that is not allowed by our definition of segments above.) We can implement this simple idea in all sorts of ways:

Recursively

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

For loop

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

While loop

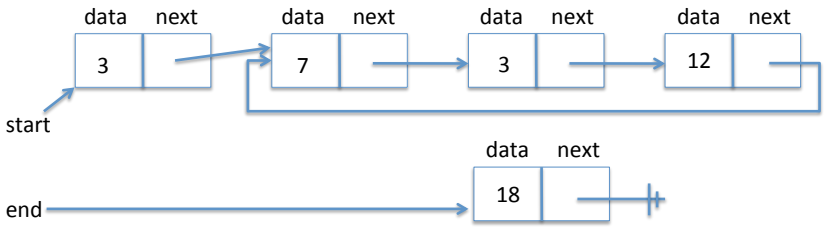
```
bool is_segment(list* start, list* end) {
  list* l = start;
  while (l != NULL) {
    if (l == end) return true;
    l = l->next;
  }
  return false;
}
```

However, every one of these implementations of `is_segment` has the same problem: if given a circular linked-list structure, the specification function `is_segment` may not terminate.

It's quite possible to create structures like this, intentionally or unintentionally. Here's how we could create the above structure in Coin:

```
--> list* start = alloc(list);
--> start->data = 3;
--> start->next = alloc(list);
--> start->next->data = 7;
--> start->next->next = alloc(list);
--> start->next->next->data = 3;
--> start->next->next->next = alloc(list);
--> start->next->next->next->data = 12;
--> start->next->next->next->next = start->next;
--> list* end = alloc(list);
--> end->data = 18;
--> end->next = NULL;
--> is_segment(start, end);
```

and this is what it would look like:



While it is not strictly necessary, *whenever possible, our specification functions should return* `true` *or* `false` *rather than not terminating or raising an assertion violation*. We do treat it as strictly necessary that our specification functions should always be safe – they should never divide by zero, access an array out of bounds, or dereference a null pointer. We will see how to address this problem in our next lecture.

4 Checking for Circularity

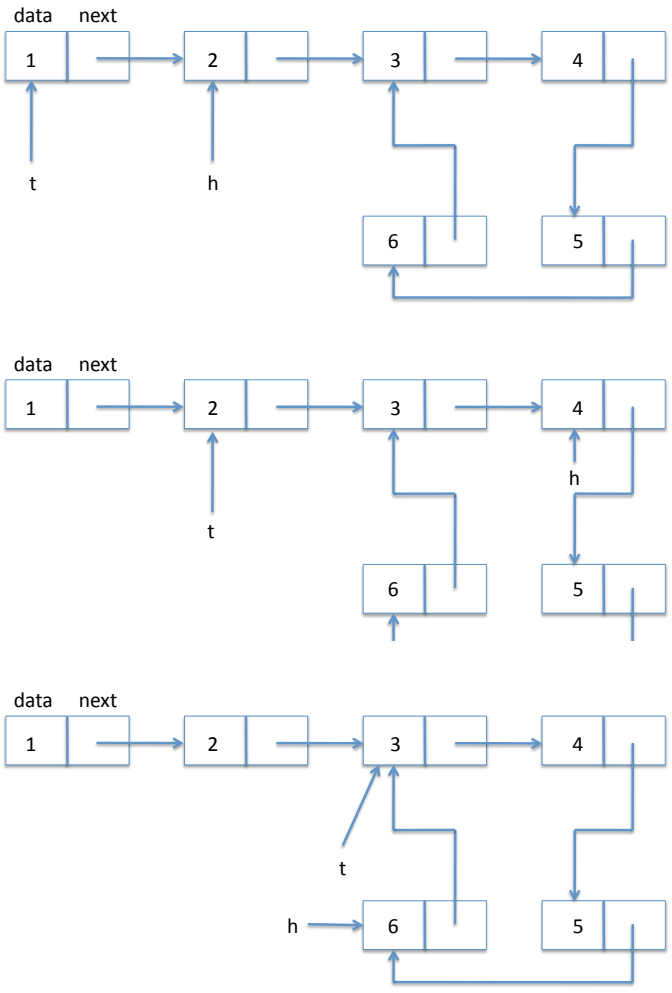
In order to make sure the `is_segment` function correctly handles the case of cyclic loops, let's write a function to detect whether a list segment is cyclic. We can call this function before we call `is_segment`, and then be confident that `is_segment` will always terminate.

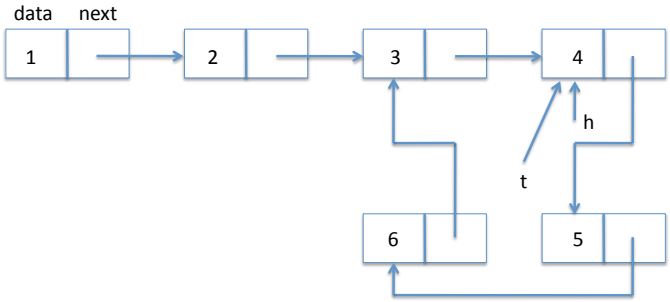
One of the simplest solutions proposed in class keeps a copy of the `start` pointer. Then when we advance p we run through an auxiliary loop to check if the next element is already in the list. The code would be something like this:

```
bool is_acyclic(list* start, list* end) {
  for (list* p = start; p != end; p = p->next)
  //@loop_invariant is_segment(start, p);
  {
    if (p == NULL) return true;

    for (list* q = start; q != p; q = q->next)
    //@loop_invariant is_segment(start, q);
    //@loop_invariant is_segment(q, p);
    {
      if (q == p->next) return false; /* circular */
    }
  }
  return true;
}
```

This solution requires $O(n^2)$ time for a list with n elements, whether it is circular or not. This doesn't really matter, because we're only using `is_acyclic` as a specification function, but there is an $O(n)$ solution. See if you can find it before reading on.

The idea for a more efficient solution was suggested in class. Create *two* pointers, a fast and a slow one. Let's name them h for *hare* and t for *tortoise*. The slow pointer t traverses the list in single steps. Fast h, on the other hand, skips two elements ahead for every step taken by t. If the faster h starts out ahead of t and ever reaches the slow t, then it must have gone in a cycle. Let's try it on our list. We show the state of t and h on every iteration.





In code:

```
bool is_acyclic(list* start, list* end) {
  if (start == NULL) return true;
  list* h = start->next;          // hare
  list* t = start;                // tortoise
  while (h != t) {
    if (h == NULL || h->next == NULL) return true;
    h = h->next->next;
    //@assert t != NULL; // hare is faster and hits NULL quicker
    t = t->next;
  }
  //@assert h == t;
  return false;
}
```

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? One way would be to rely on our operational reasoning and insert an assert:

```
//@assert t != NULL; // hare is faster and hits NULL quicker
```

But as the comment indicates, it is hard to justify in logical reasoning why this assert never fails. Can we achieve the same logically? Yes, but while you think about how, we will first analyze the complexity of the algorithm and resolve another mystery.

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 $O(n/2)$ 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. In particular, whenever the algorithm returns `true`, it's because the hare caught the tortoise, which is an obvious cycle. Yet, since, in the cyclic case, the distance between the hare and the tortoise strictly decreases in each iteration, the algorithm will correctly detect all cycles.

Now, where is the mystery? When inspecting the `is_acyclic` function, we are baffled why it never uses `end`. Indeed, `is_acyclic(start, end)` correctly checks whether the `NULL`-terminated list beginning at `start` is cyclic, but ignores the value of `end` entirely. Hence, `is_segment(start, end)`, which calls `is_acyclic(start, end)`, will categorically return `false` on any cyclic list even if the segment from `start` to `end` would still have been acyclic.

This is actually fine for our intended use case in stacks and queues, because we do not want them to be cyclic ever. But let's fix the issue for more general uses. Can you figure out how?

The idea is to let the hare watch out for `end` and simply treat `end` as yet another reason to stop iterating, just like `NULL`. If the hare passes by `end`, then the list segment from `start` to `end` cannot have been cyclic, because he would, otherwise, have found `end` in his first time around the cycle already. Note that the same argument would not quite work when, instead, tortoise checks for `end`, because tortoise might have just found `end` before the hare went around the cycle already.

```
bool is_acyclic(list* start, list* end) {
  if (start == NULL) return true;
  list* h = start->next;         // hare
  list* t = start;               // tortoise
  while (h != t) {
    if (h == NULL || h->next == NULL) return true;
    if (h == end || h->next == end) return true;
    h = h->next->next;
    //@assert t != NULL; // hare is faster and hits NULL quicker
    t = t->next;
  }
  //@assert h == t;
  return false;
}
```

This algorithm is a variation of what has been called the *tortoise and the hare* and is due to Floyd 1967.

5 Tortoise is Never NULL

Let's get back to whether we can establish why the following assertion holds by logical reasoning.

```
//@assert t != NULL; // hare is faster and hits NULL quicker
```

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[1], but at least

[1] Isn't that Zeno paradoxical?

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_acyclic`:

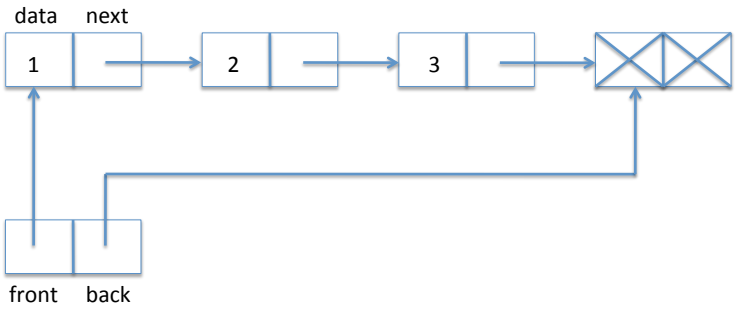
```
bool is_acyclic(list* start, list* end) {
  if (start == NULL) return true;
  if (start->next == NULL) return true;
  list* h = start->next;         // hare
  list* t = start;               // tortoise

  while (h != t)
  //@loop_invariant is_segment(t, h);
  {
    if (h->next == NULL || h->next->next == NULL) return true;
    if (h == end || h->next == end) return true;
    h = h->next->next;
    t = t->next;
  }

  //@assert h == t;
  return false;
}
```

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.

Watch out for one subtle issue, though. Now the implementations and contracts of `is_acyclic` and `is_segment` are mutually recursive. That means, with contracts enabled (`cc0 -d`), some calls to `is_segment` will never terminate. This can be fixed by introducing a copy of `is_segment` distinguishing cyclic from noncyclic segments. The key insight is from the complexity analysis. The hare and the tortoise will never be farther apart than the size of the cycle.

6 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; the `is_segment` function we already wrote enforces this.

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

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.

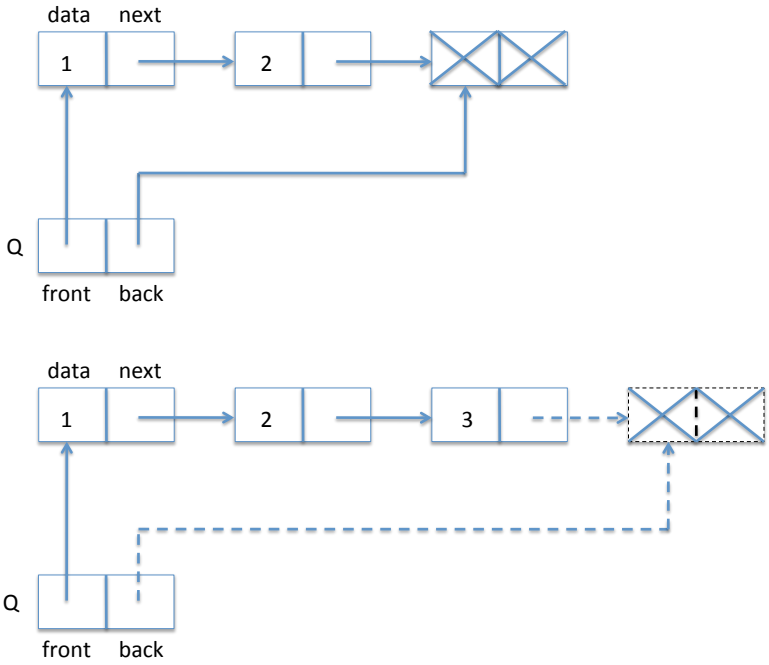
```
bool queue_empty(queue Q)
//@requires is_queue(Q);
{
  return Q->front == Q->back;
}
```

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;
}
```

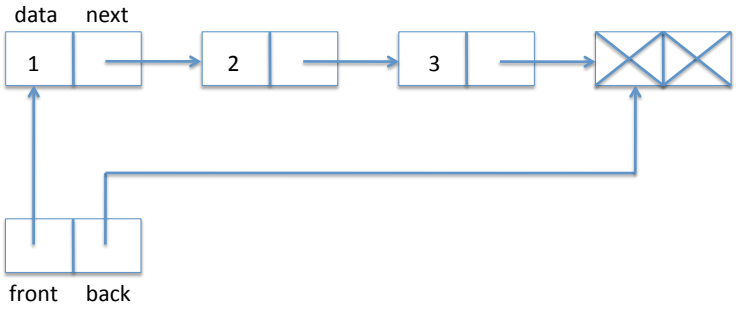
To enqueue something, that is, add a new item to the back of the queue, we just write the data (here: a string) into the extra element at the back, create a new back element, and make sure the pointers updated correctly. You should draw yourself a diagram before you write this kind of code. Here is a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

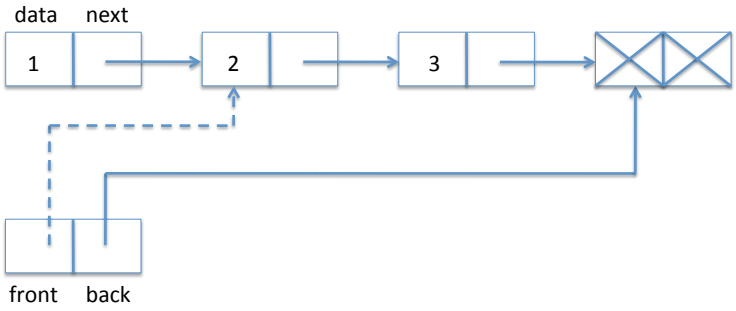


In code:

```
void enq(queue Q, string s)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{
  list* p = alloc(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) {
  return Q != NULL && is_segment(Q->front, Q->back);
}
```

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.

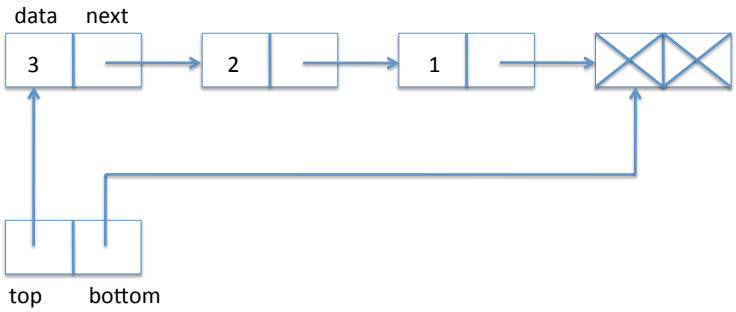
We also make the assignment `Q->front = Q->front->next`. Why does this preserve the invariant? Because we know that the queue is not empty (second precondition of `deq`) and therefore `Q->front != Q->back`. Because `Q->front` to `Q->back` is a valid non-empty segment, `Q->front->next` cannot be null.

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.

7 Stacks with Linked Lists

For the implementation of stacks, we can reuse linked lists and 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:



Here, 3 is the element at the top of the stack.

We define:

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

struct stack_header {
  list* top;
  list* bottom;
};
typedef struct stack_header* stack;
```

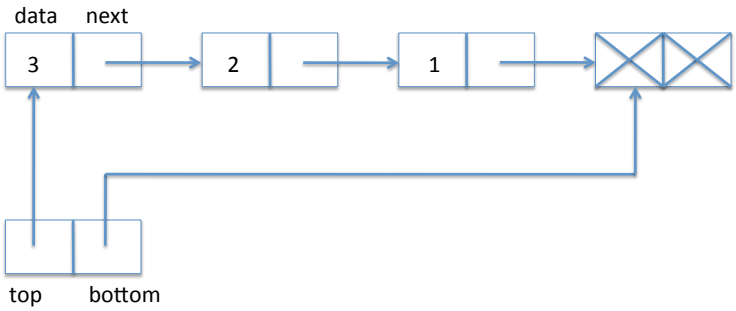
To test if some structure is a valid stack, we only need to check that the list starting at `top` ends in `bottom`; this is almost identical to the data structure invariant for queues:

```
bool is_stack(stack S) {
  return S != NULL && is_segment(S->top, S->bottom);
}
```

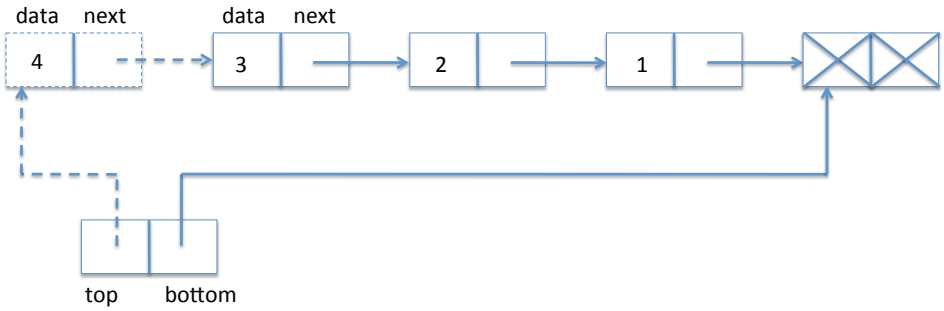
Popping from a stack requires taking an item from the front of the linked list, which is much like dequeuing.

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

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 – the opposite end of the linked list from the queue. 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* p = alloc(struct list_node);
  p->data = e;
  p->next = S->top;
  S->top = p;
}
```

This completes the implementation of stacks.

Exercises

Exercise 1 *Consider what would happen if we* `pop` *an element from the empty stack when contracts are not checked in the linked list implementation? When does an error arise?*

Exercise 2 *Stacks are usually implemented with just one pointer in the header, to the top of the stack. Rewrite the implementation in this style, dispensing with the* `bottom` *pointer, terminating the list with* `NULL` *instead.*