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

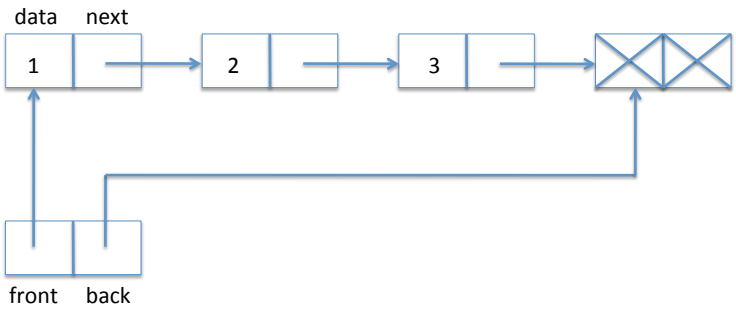
Computational Thinking: We emphasize the importance of *abstraction* by producing a second implementation of the stacks and queues we introduced in the notes for lecture 9.

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

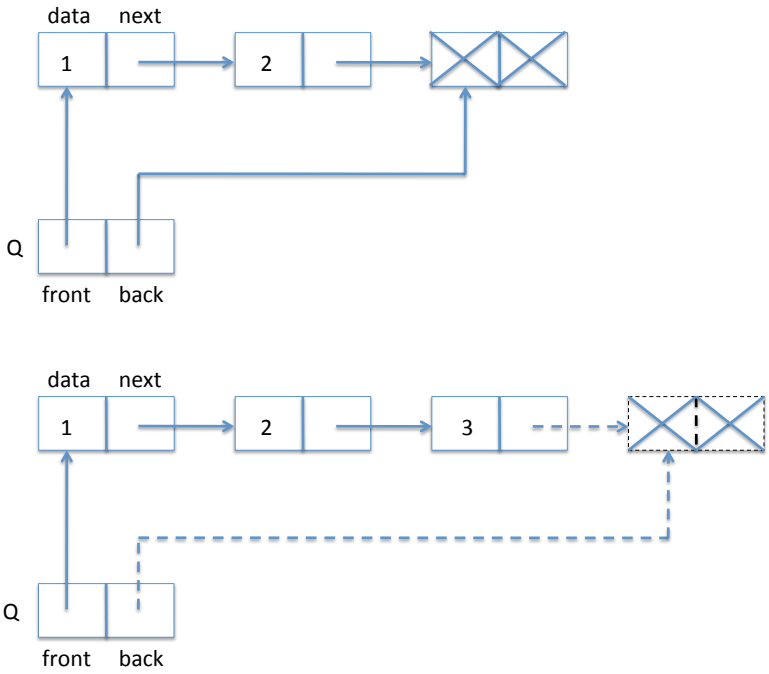
```
queue Q = alloc(struct queue_header);
```

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.

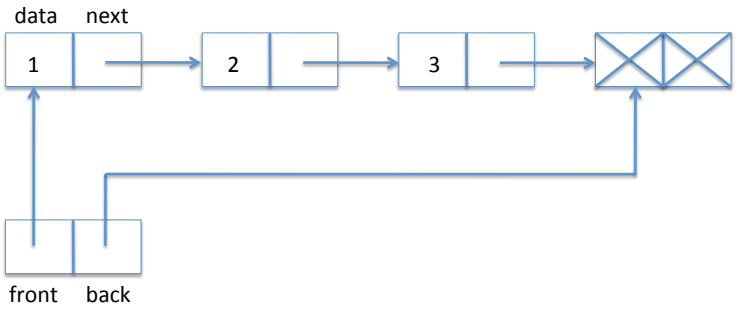
To enqueue something, that is, add a new item to the back of the queue, we just write the data (here: a string) into the extra element at the back, create a new back element, and make sure the pointers updated correctly. You should draw yourself a diagram before you write this kind of code. Here is a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

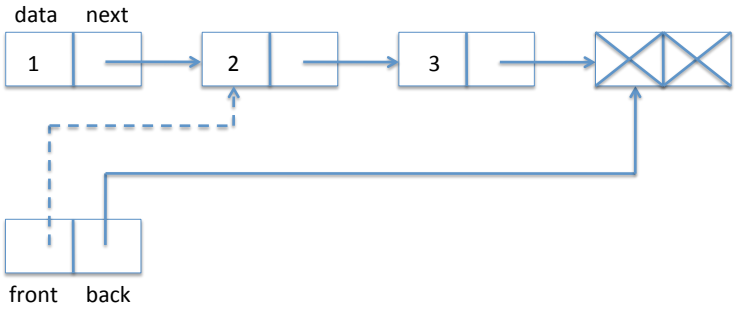


Another important point to keep in mind as you are writing code that manipulates pointers is to make sure you perform the operations in the right order, if necessary saving information in temporary variables.

```
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;
}
```

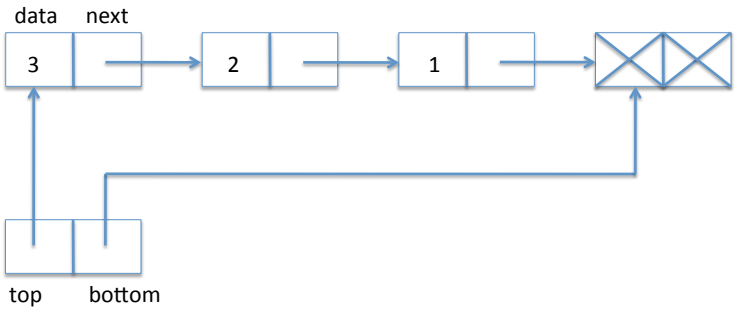
We see that `Q->front` is okay, because by the first test we know that `Q != NULL` is the precondition holds. By the second test we see that both `Q->front` and `Q->back` are not null, and we can therefore dereference them.

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.

3 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

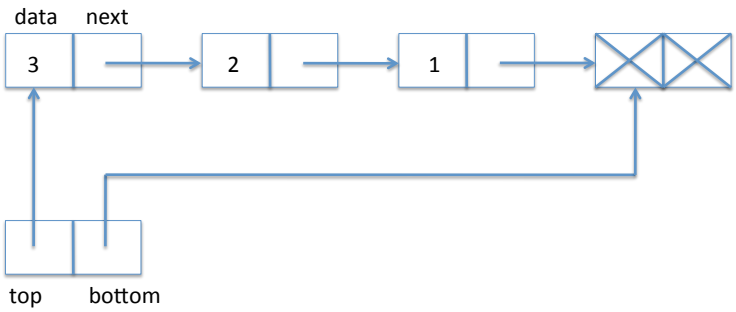
the list starting at `top` ends in `bottom`; this is almost identical to the data structure invariant for queues:

```
bool is_stack(stack S) {
  if (S == 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;
}
```

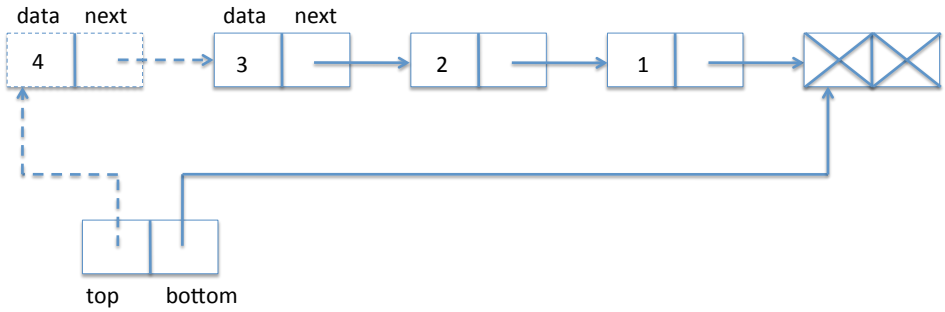
Popping from a stack requires taking an item from the front of the linked list, which is much like dequeuing.

```
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



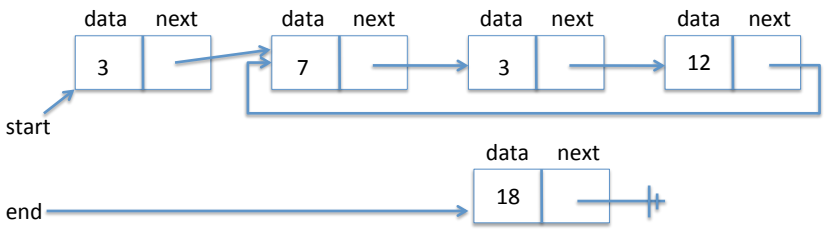
In code:

```
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;
}
```

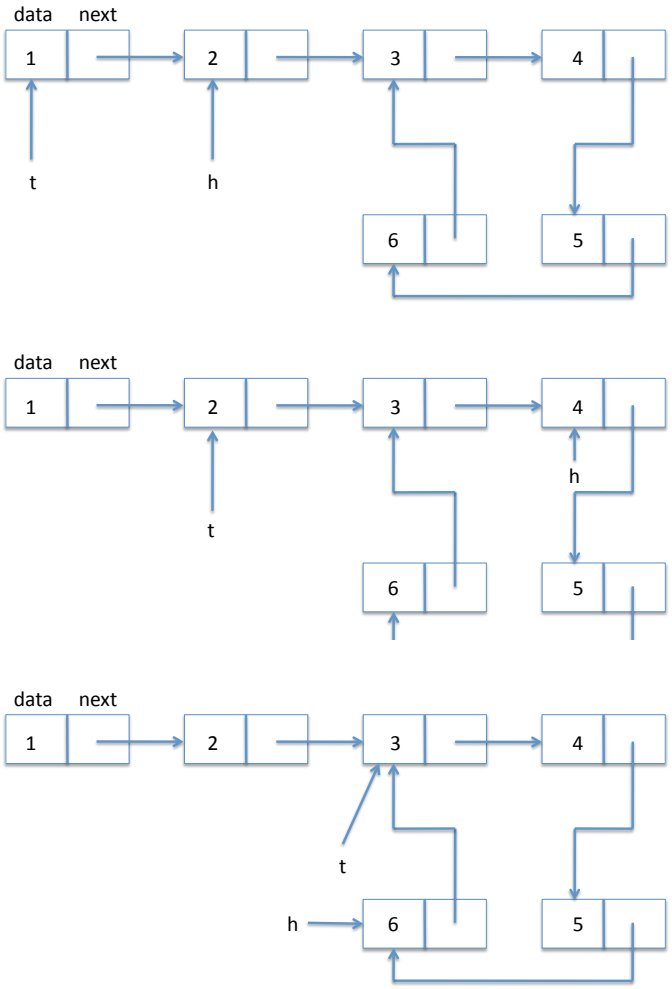
This completes the implementation of stacks, which are a very simple and pervasive data structure.

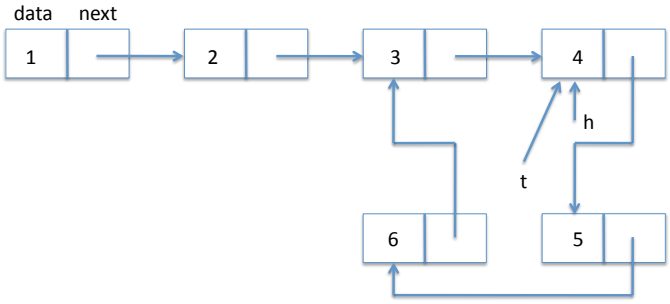
4 Circularity checking

In the last lecture, we talked about how to identify circularity in linked lists, thus avoiding



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 show that this loop invariant is true initially and implies safety. 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.

However, this loop invariant is *not* preserved by every iteration of the loop! Why not? (Try to work out the answer before going to the next page.)

The definition of `is_segment` we have presented does not allow either endpoint to be `NULL`, but there is no reason that `h->next->next`, which will be `h` on the next iteration of the loop, shouldn't be null. The segment that we have is actually from `t` to whatever pointed to `h`.

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

Because we have a loop invariant that `hprev` and `h` are equal, we can instead bring `h` inside the loop rather than tracking it separately.

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

Another option we could have taken would have been to change our definition of `is_segment` to allow for `NULL` to appear in valid segments. You can do this as an exercise.

It is now possible to prove that this loop invariant is preserved. How would this invariant imply that t is not NULL? How does it imply that the loop guard is safe?

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

5 Data in Memory

Ultimately, "all" data resides in memory. In fact, part of the data may also be kept in fast registers directly on the CPU. You will learn about registers in detail in 15-213 and can learn about their use in programming languages in 15-411. For the purposes of today's lecture, it is sufficient to pretend all data would sit in memory and ignore registers for the time being. This simplifies the principles without losing too much precision.

The data in memory is addressed by memory addresses that fit to the addressing of the CPU in your computer. We will just pretend 32bit addresses, because those are shorter to write down. All addresses are positive, so the lowest address is 0x00000000 and the highest address $2^{32} - 1 =$ `0xFFFFFFFF`. All data (with the caveat about registers) sits in memory at some address. One important question about all data in memory is how big it is, so that the compiler can make sure program data is stored without accidental overlapping regions.

The basic memory layout looks as follows:

```
OS AREA
============
System stack   (local variables and function calls)
============
unused
============
System heap    (data allocated here... alloc or alloc_array)
```

```
============
.text (read only)  (program instructions sit in memory)
============
OS AREA
```

One consequence of this memory layout is that the stack grows towards the heap, and the heap usually grows towards the stack. The reason that the stack is called a stack is because it operates somewhat like the principle of the stack data structure. Your program can put new data on the top of the stack. It can also pop elements of the stack if this data is no longer necessary. Unlike the stack abstraction, it may appear as if your program internally also modifies data that is on the stack, even if it is not quite at the top of it. However, the only data on the stack that the program modifies is in the top range of the stack (perhaps the top 512 bytes or so, depending on the function that runs), even if it is not just the top word of the stack.

Programs cannot access memory cells that belong to the operating system. If they try, programs get an "exception" like a segmentation fault. Where can that happen in C0? C0 takes great care to ensure that it never gives you any pointers to uninitialized or random or garbage data in memory, *except*, of course, the `NULL` pointer. `NULL` is a special pointer to the memory address 0, which belongs to the operating system. Any access by a user-land program by dereferencing a NULL pointer causes a segfault.

If, however, you are writing a program that will be running as part of the operating system, your program has no protection against `NULL` pointer dereferencing anymore, because memory address 0 is a valid address for the operating system, even if not for regular programs. When writing code for operating systems, you, thus, need to have mastered the art of protecting against illegal `NULL` dereferences. This is exactly one of the things that contracts, loop invariants, and assertions prepare you for.

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.*