Lecture 10
Linked Lists

15-122: Principles of Imperative Computation (Spring 2018)
Frank Pfenning, Rob Simmons, André Platzer, Iliano Cervesato

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 work lists of any length.

This fits as follows with respect to our learning goals:

Computational Thinking: We discover that arrays contain implicit information, namely the indices of elements, which an be made explicit as the addresses of the nodes of a linked list. We also encounter the notion of trade-off, as arrays and linked lists have different advantages and drawbacks and yet achieve similar purposes.

Algorithms and Data Structures: We explore linked lists, a data structure used pervasively in Computer Science, and examine some basic algorithms about them.

Programming: We see that programming algorithms for linked lists can be tricky, which exposes once more the power of stating and checking invariant. We use linked lists to implement stacks and queues.

1 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:

```
1 struct list_node {
2   elem data;
3   struct list_node* next;
4 };
5 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*`. We could also have written these two statements in the other order, to make better use of the type definition:

```
1 typedef struct list_node list;
2 struct list_node {
3   elem data;
4   list* next;
5 };
```

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

```
1 struct infinite {
2   int x;
3   struct infinite next;
4 }
```

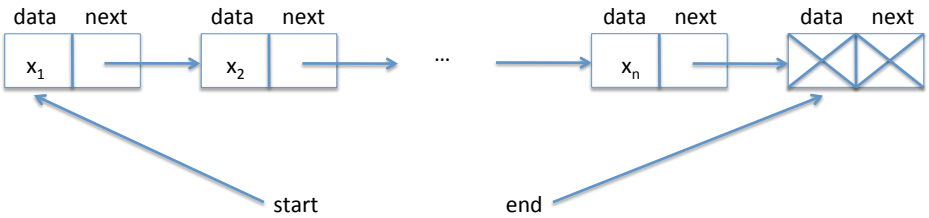
would be rejected by the C0 compiler because it would require an infinite amount of space. The general rule is that a struct can be recursive, but the recursion must occur beneath a pointer or array type, whose values are addresses. This allows a finite representation for values of the struct type.

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

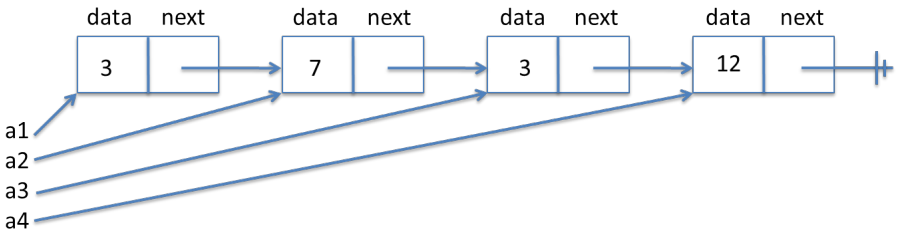
abstraction around them. When we use them we know about and exploit their precise internal structure. This is in contrast to *abstract types* such as queues or stacks whose implementation is hidden behind an interface, exporting only certain operations. This limits what clients can do, but it allows 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.

2 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);
}
```

Using a `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;
}
```

Using a `for` loop:

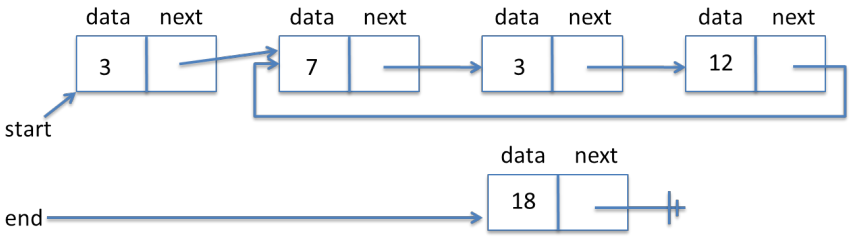
```
bool is_segment(list* start, list* end) {
  for (list* p = start; p != NULL; p = p->next) {
    if (p == end) return true;
  }
  return false;
}
```

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 a circular linked list 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:

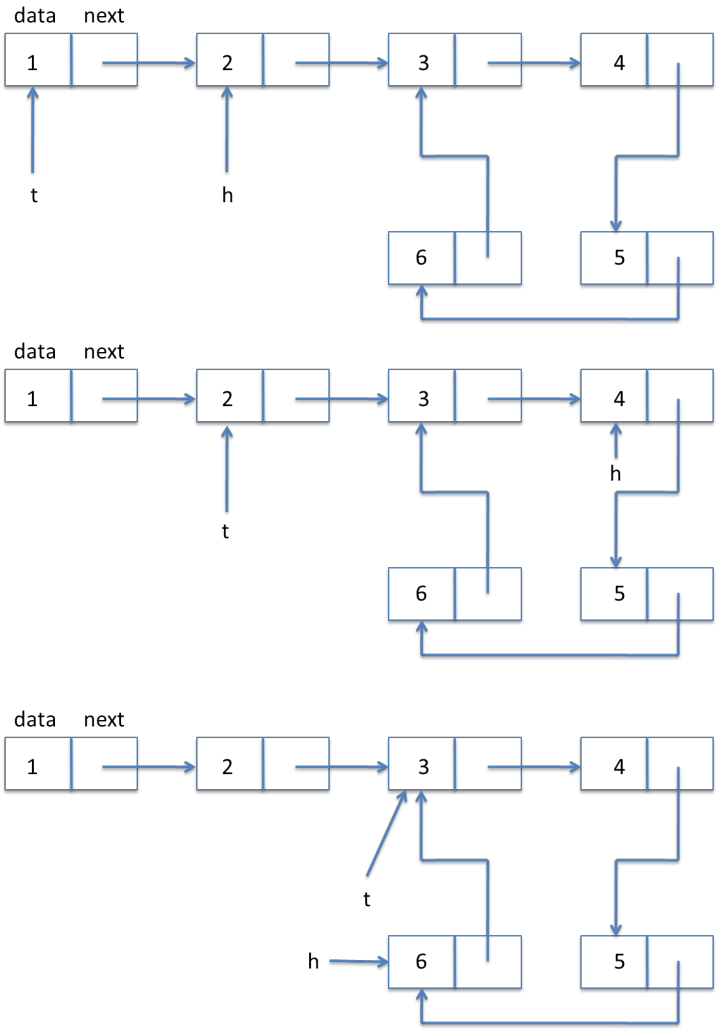


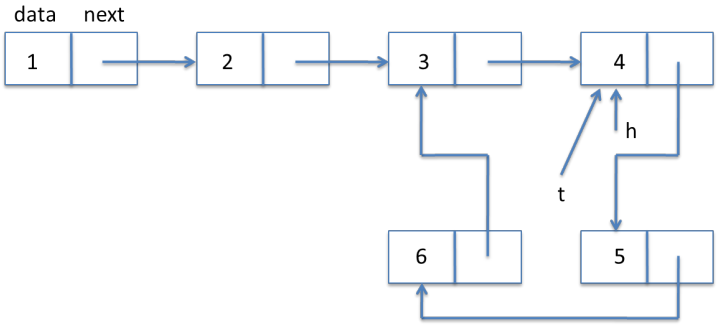
Whenever possible, our specification functions should return `true` or `false` rather than not terminating or raising an assertion violation. We do treat it as strictly necessary that our specification functions should always be safe — they should never divide by zero, access an array out of bounds, or dereference a null pointer.

3 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.

Our cycle detection function makes use of 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:

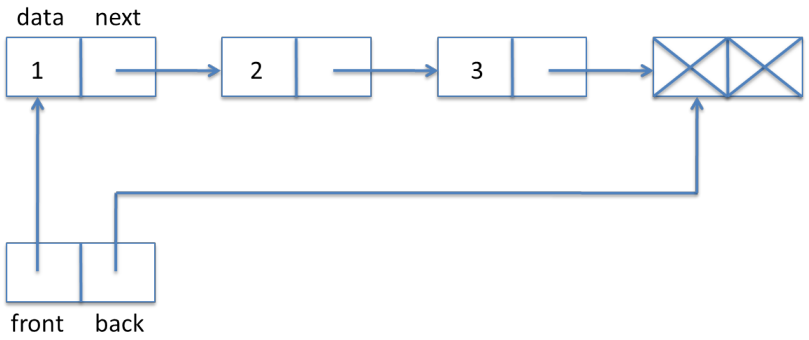
```
1  bool is_acyclic(list* start) {
2    if (start == NULL) return true;
3    list* h = start->next;          // hare
4    list* t = start;                // tortoise
5    while (h != t) {
6      if (h == NULL || h->next == NULL) return true;
7      h = h->next->next;
8      //@assert t != NULL; // faster hare hits NULL quicker
9      t = t->next;
10   }
11   //@assert h == t;
12   return false;
13 }
```

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.

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

4 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.

It is convenient to have the `back` pointer point to one element past the end of the queue. Therefore, there is always one extra element at the end of the queue which does not have valid data or `next` pointer. We call it the *dummy node* and we have indicated it in the diagram by writing `X`.

The above picture yields the following definition.

```
1 typedef struct queue_header queue;
2 struct queue_header {
3   list* front;
4   list* back;
5 };
```

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_t` 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_t`.

```
7 typedef queue* queue_t;
```

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 called 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.

```
9  bool is_queue(queue* Q) {
10   return Q != NULL
11       && is_acyclic(Q->front)
12       && is_segment(Q->front, Q->back);
13 }
```

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.

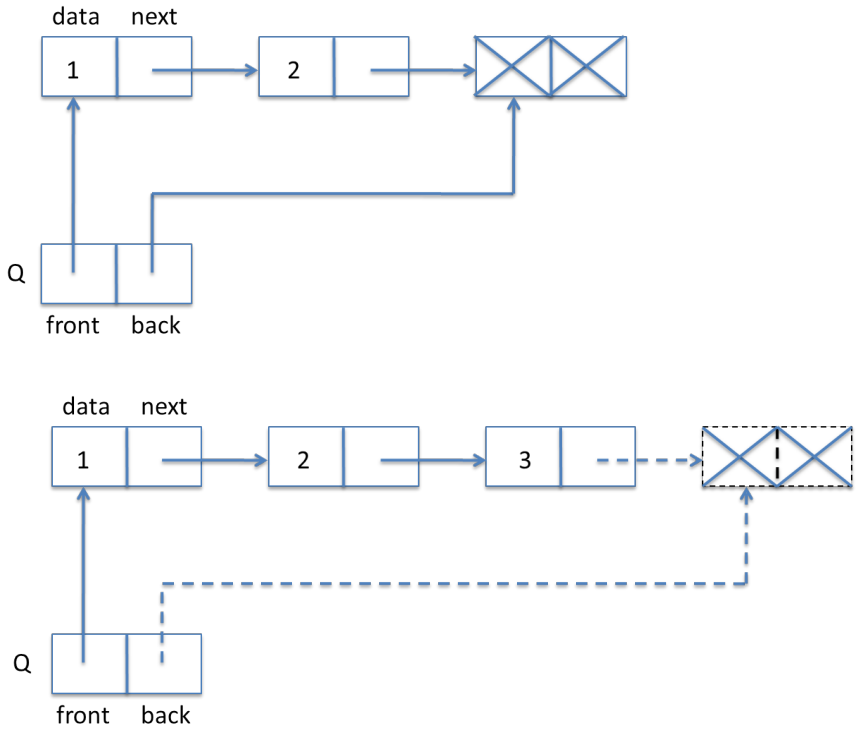
```
15 bool queue_empty(queue* Q)
16 //@requires is_queue(Q);
17 {
18   return Q->front == Q->back;
19 }
```

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. Said this, 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.

```
21 queue* queue_new()
22 //@ensures is_queue(\result);
23 //@ensures queue_empty(\result);
24 {
25   queue* Q = alloc(queue);    // Create header
26   list* dummy = alloc(list);  // Create dummy node
27   Q->front = dummy;           // Point front
28   Q->back = dummy;            //   and back to dummy node
29   return Q;
30 }
```

To enqueue something, that is, add a new item to the back of the queue, we just write the data into the extra element at the back, create a new back

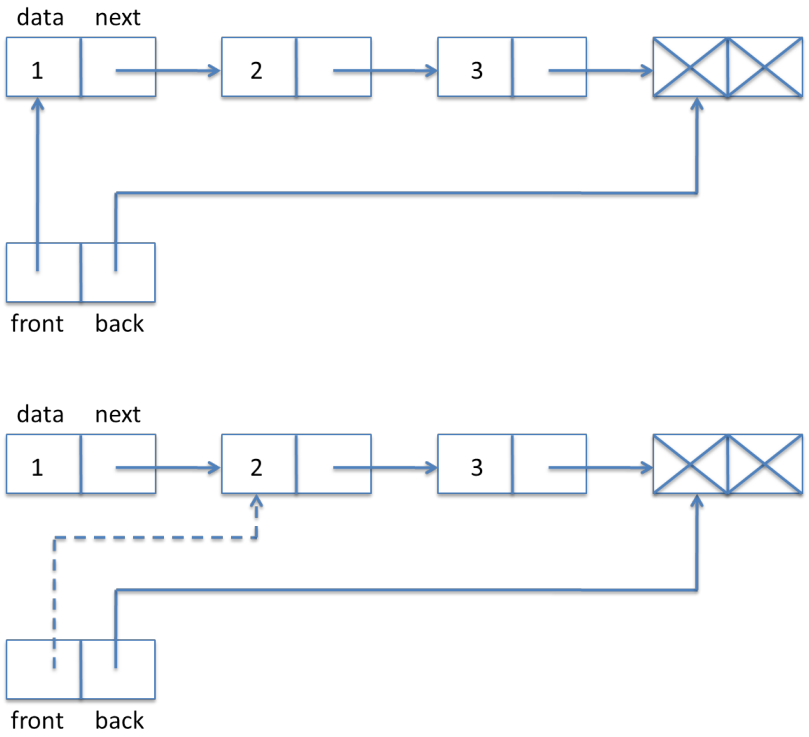
element, and make sure the pointers are updated correctly. You should draw yourself a diagram before you write this kind of code. Here is a before-and-after diagram for inserting 3 into a list. The new or updated items are dashed in the second diagram.



In code:

```
32 void enq(queue* Q, elem x
33 //@requires is_queue(Q);
34 //@ensures is_queue(Q);
35 {
36   list* new_dummy = alloc(list);  // Create a new dummy node
37   Q->back->data = x;              // Store x in old dummy node
38   Q->back->next = new_dummy;
39   Q->back = new_dummy;
40 }
```

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:

```
42 elem deq(queue* Q)
43 //@requires is_queue(Q);
44 //@requires !queue_empty(Q);
45 //@ensures is_queue(Q);
46 {
47   elem x = Q->front->data;
48   Q->front = Q->front->next;
49   return x;
50 }
```

Let's verify that 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:

```
9  bool is_queue(queue Q) {
10   return Q != NULL
11       && is_acyclic(Q->front)
12       && is_segment(Q->front, Q->back);
13 }
```

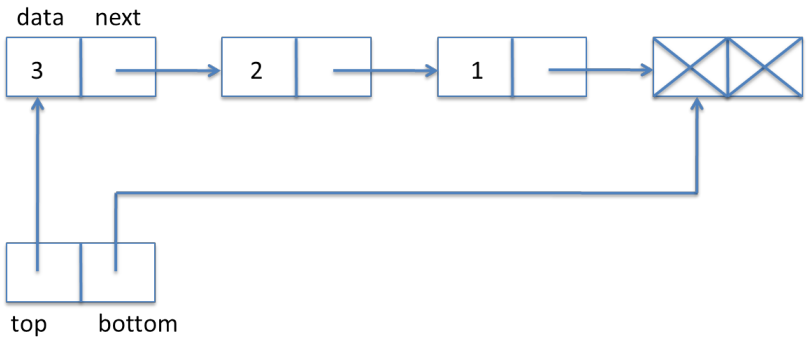
We see that `Q->front` is okay, because by the first test we know that `Q != NULL` is the precondition holds. By the second test we see that both `Q->front` and `Q->back` are not null, and we can therefore dereference them.

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.

5 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, the validation function `is_stack`, internal to the library implementation, and the client operations `stack_empty` and `stack_new` are implemented identically to what we saw for queues. The `bottom` pointer of the stack is otherwise unused. A typical stack then has the following form:



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

We define:

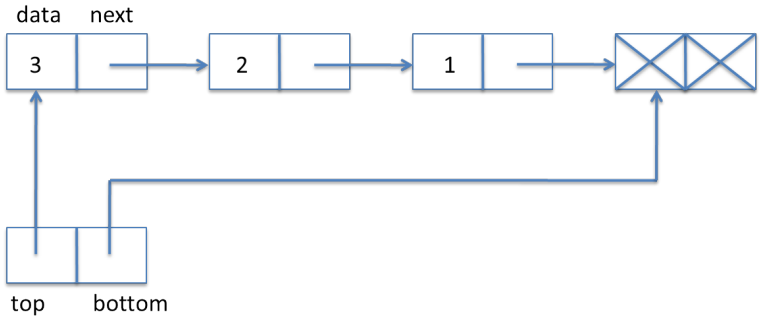
```
1  typedef struct stack_header stack;
2  struct stack_header {
3    list* top;
4    list* bottom;
5  };
6
7  bool is_stack(stack* S) {
8    return S != NULL
9        && is_acyclic(S->top)
10       && is_segment(S->top, S->bottom);
11 }
```

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

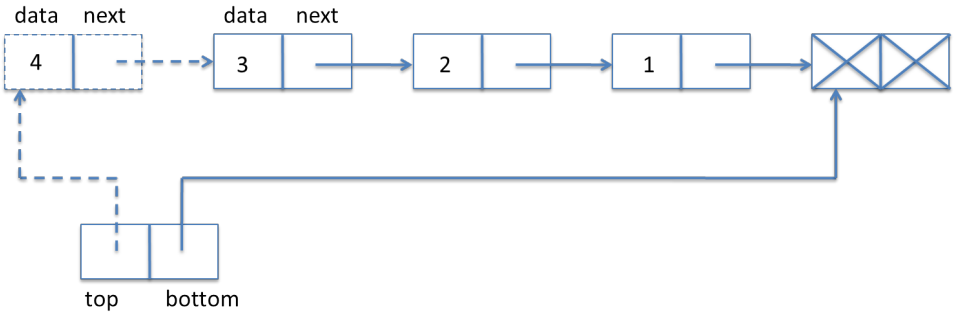
```
30 elem pop(stack* S)
31 //@requires is_stack(S);
32 //@requires !stack_empty(S);
33 //@ensures is_stack(S);
34 {
35   elem x = S->top->data;
36   S->top = S->top->next;
37   return x;
38 }
```

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:

```
40 void push(stack* S, elem x)
41 //@requires is_stack(S);
42 //@ensures is_stack(S);
43 {
44   list* p = alloc(list);  // Allocate a new top node
45   p->data = x;
46   p->next = S->top;
47   S->top = p;
48 }
```

The client-side type `stack_t` is defined as a pointer to a `stack_header`:

```
50 typedef stack* stack_t;
```

This completes the implementation of stacks.

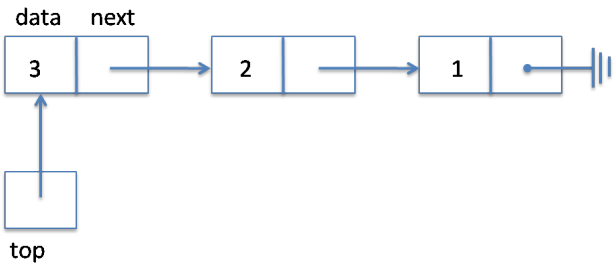
6 Sharing

We observed in the last section that the `bottom` pointer of a `stack_header` structure is unused other than for checking that a stack is empty. This suggests a simpler representation, where we take the empty stack to be `NULL` and do without the `bottom` pointer. This yields the following declarations

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

bool is_stack(stack* S) {
  return S != NULL && is_acyclic(S->top);
}
```

and pictorial representation of a stack:

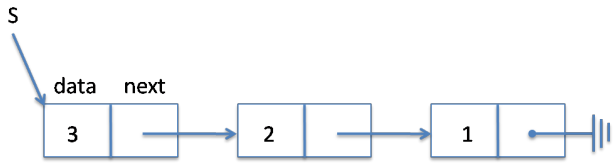


But, then, why have a header at all? Can't we define the stack simply to be the linked list pointed by `top` instead?

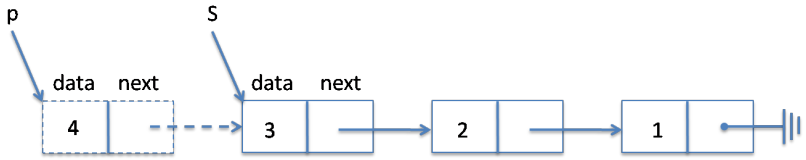
Eliminating the header would lead to a redesign of the interface and therefore to changes in the code that the client writes. Specifically,

1. `NULL` is now a valid stack — it represents the empty stack. Therefore, we would have to remove all those `NULL` checks from the interface. (Alternatively, we can bring back the dummy node, but this time with a mandatory `NULL` pointer in the `next` field.)

2. More dramatically, we need to change the type of `push` and `pop`. Consider performing the operation `push(S, 4)` where `S` contains the address of the stack from the caller's perspective:



This call would result in the following stack:



where `p` is a pointer to the newly allocated list node. Note that the stack has not changed from the point of view of the caller! In fact, from the caller's standpoint, `S` still points to the node containing 3. The only way for the caller to access the updated stack is that the pointer `p` be given back to it. Thus, `push` must now return the updated stack. Therefore, we need to change its prototype to

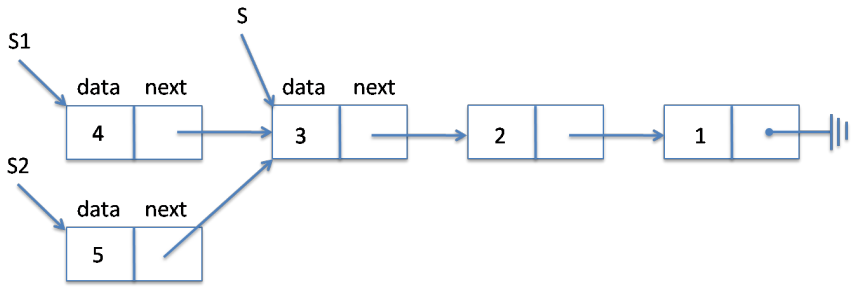
```
stack_t push(stack_t S, elem x);
```

The same holds for `pop`, with a twist: `pop` already returns the value at the top of the stack. It now needs to return both this value and the updated stack.

With such header-less stacks, the client has the illusion that `push` and `pop` produces a new stack each time they are invoked. However, the underlying linked lists share many of the same elements. Consider performing the following operations on the stack `S` above:

```
stack_t S1 = push(S, 4);
stack_t S2 = push(S, 5);
```

This yields the following memory layout:



All three stacks share nodes 3, 2 and 1. Observe furthermore that the second call to `push` operated on `S`, which remained unchanged after the first call. At this point, a `pop` on `S` would result in a fourth stack, say `S3`, which points to node 2.

Sharing is an efficient approach to maintaining multiple versions of a data structure as a sequence of operations is performed on them. Sharing is not without its perils, however. As an exercise, consider an implementation of queues such that `enq` and `deq` return to their caller a pair of pointers to the front and back of the underlying linked list (maybe packaged in a **`struct`**). A carefully chosen series of `enq` and `deq` operations will break the queue (or more precisely its representation invariant).

Exercises

Exercise 1. *The tortoise-and-hare implementation of circularity checking we gave has an assertion,* `t != NULL`*, which we can't prove with the given loop invariants. What loop invariants would allow us to prove that assertion correct? Can we write loop invariants that allow us to prove, when the loop exits, that we have found a cycle?*

Exercise 2. *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 3. *Complete the implementations of stack as defined at the beginning of Section 6, dispensing with the* `bottom` *pointer, terminating the list with* `NULL` *instead.*

Exercise 4. *Consider an implementation of queues as linked list such that* `enq` *and* `deq` *return to their caller a new header to the front and back of the underlying linked list each time they are called. Engineer a series of* `enq` *and* `deq` *operations*

that, starting from a valid queue, will result in a data structure that does not satisfy the representation invariant of queues (i.e., result in a broken queue).

Exercise 5. *Here's a simple idea to check that a linked list is acyclic: first, we keep 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) {
  for (list* p = start; p != NULL; 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 code has however an issue. Can you find it?