Lecture 10
Linked Lists

15-122: Principles of Imperative Computation (Fall 2023)
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.

Additional Resources

- Review slides (`https://cs.cmu.edu/~15122/handouts/slides/review/10-linkedlist.pdf`)
- OLI modules (`https://cs.cmu.edu/~15122/handouts/oli/oli-10.shtml`)
- Code for this lecture (`https://cs.cmu.edu/~15122/handouts/code/10-linkedlist.tgz`)

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:

```
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*`. We could also have written these two statements in the other order, to make better use of the type definition:

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

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

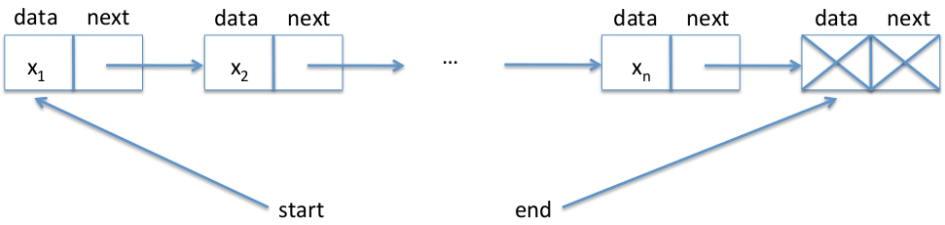
```
struct infinite {
  int x;
  struct infinite next;
}
```

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.

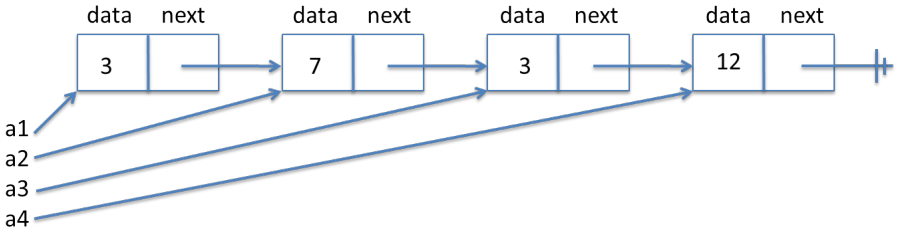
We don't introduce any general operations on lists; let's wait and see what we need where they are used. Linked lists as we use them here are a *concrete type* which means we do *not* construct an interface and a layer of abstraction around them. When we use them we know about and exploit their precise internal structure. This is 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 point to locations that contain the same data, *a1* and *a3* point to 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:

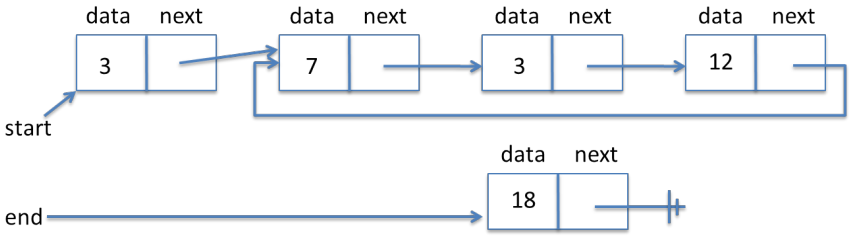
```
bool is_segment(list* start, list* end) {
  for (list* p = start; p != NULL; p = p->next) {
    if (p == end) return true;
  }
  return false;
}
```

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:

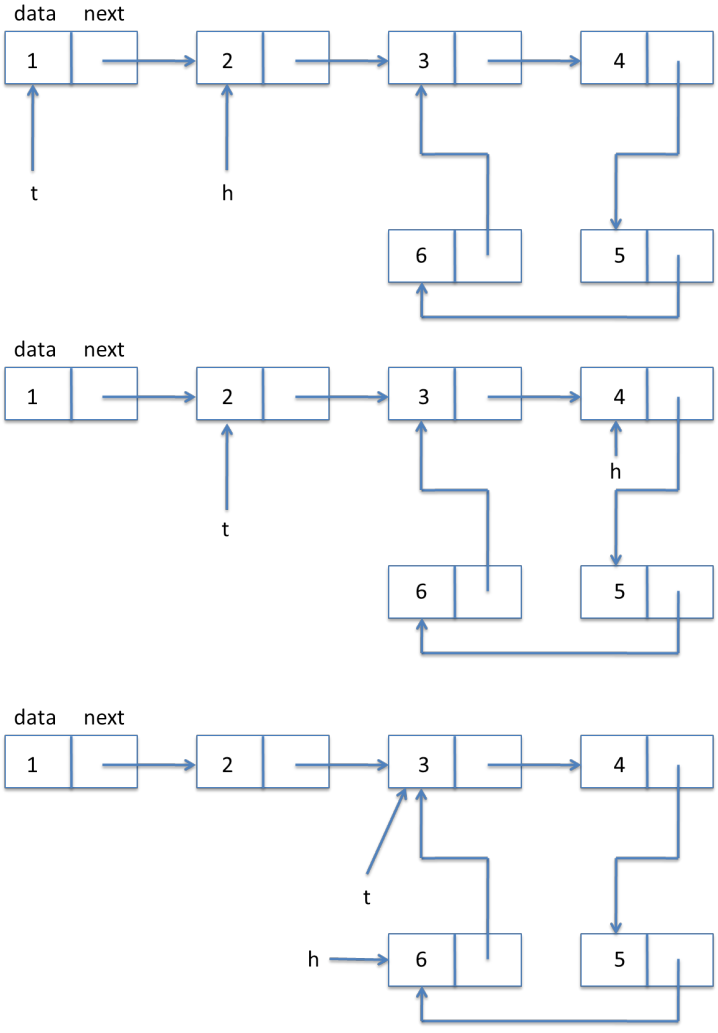


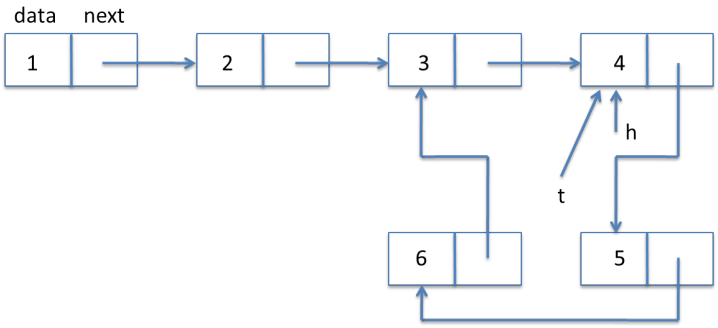
Whenever possible, our specification functions should return `true` or `false` rather than not terminating or raising an assertion violation. We do treat it as strictly necessary that our specification functions should always be safe — they should never divide by zero, access an array out of bounds,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

3 Checking for Circularity

In order to make sure the `is_segment` function correctly handle 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:

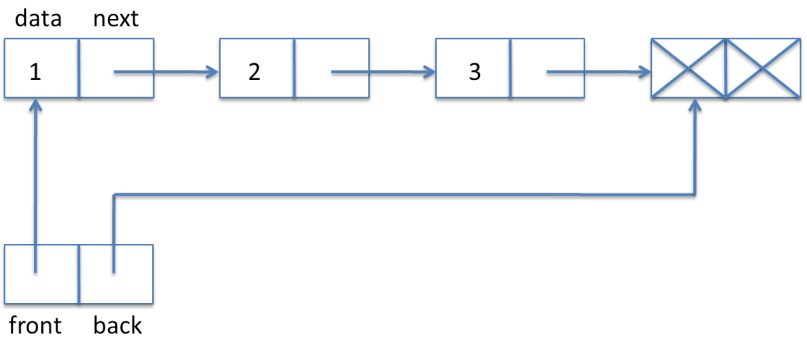
```
bool is_acyclic(list* start) {
  if (start == NULL) return true;
  list* h = start->next;          // hare
  list* t = start;                // tortoise
  while (h != t) {
    if (h == NULL || h->next == NULL) return true;
    h = h->next->next;
    //@assert t != NULL; // faster hare hits NULL quicker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

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.

```
typedef struct queue_header queue;
struct queue_header {
  list* front;
  list* back;
};
```

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`.

```
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 pic-

ture, 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_acyclic(Q->front)
      && is_segment(Q->front, Q->back);
}
```

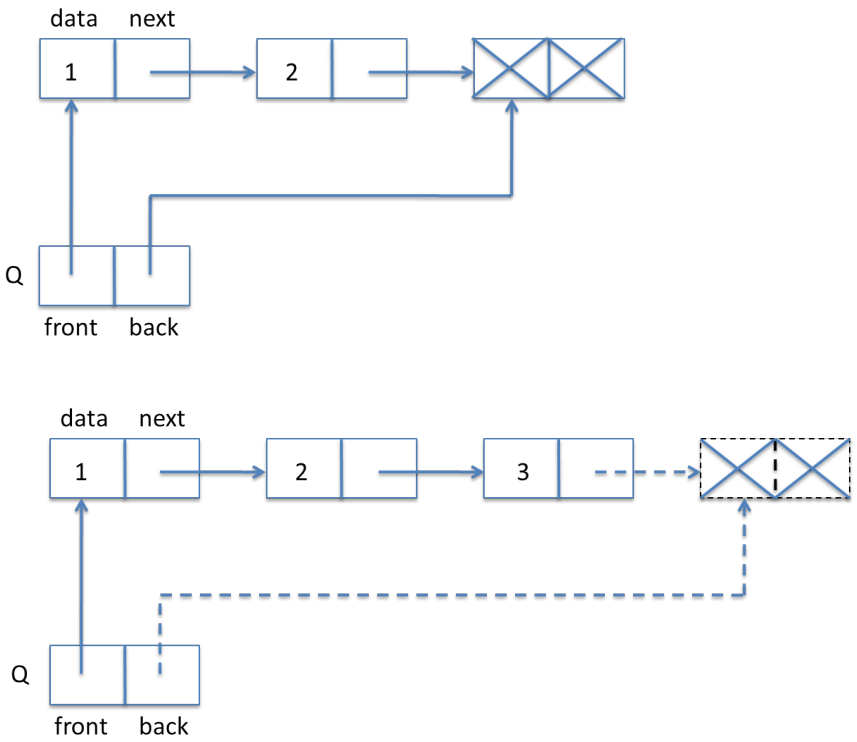
To check if the queue is empty we just compare its front and back. If they are equal, the queue is empty; otherwise it is not. We require that we are being passed a valid queue. Generally, when working with a data structure, we should always require and ensure that its invariants are satisfied in the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

```
queue* queue_new()
//@ensures is_queue(\result);
//@ensures queue_empty(\result);
{
  queue* Q = alloc(queue);    // Create header
  list* dummy = alloc(list);  // Create dummy node
  Q->front = dummy;           // Point front
  Q->back = dummy;            //   and back to dummy node
  return Q;
}
```

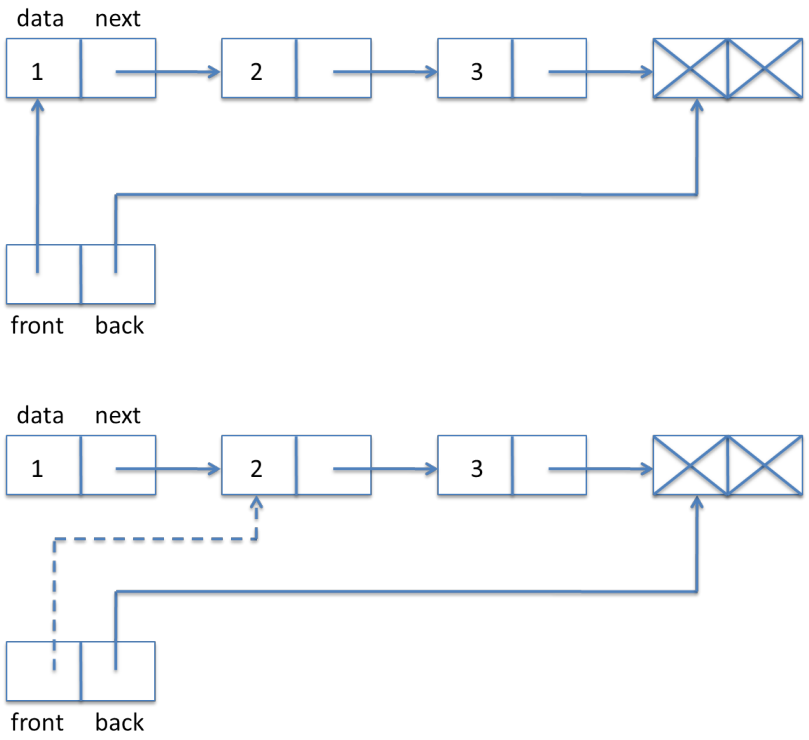
To enqueue something, that is, add a new item to the back of the queue, we just write the data into the extra element at the back, create a new back element, and make sure the pointers are updated correctly. You should draw yourself a diagram before you write this kind of code. Here is a before-and-after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



In code:

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

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);
{
  elem x = Q->front->data;
  Q->front = Q->front->next;
  return x;
}
```

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:

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

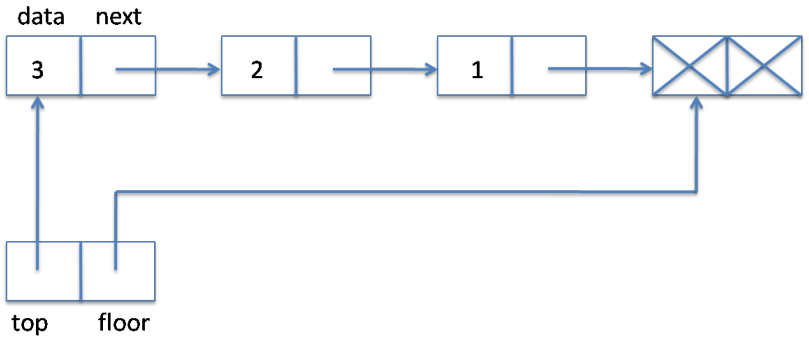
We see that `Q->front` is okay, because by the first test we know that `Q != NULL` is the precondition holds. By the third test we see that both `Q->front` and `Q->back` are not null, and we can therefore dereference them.

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 `floor` to a dummy node right after the last element (or *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 `floor` 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.

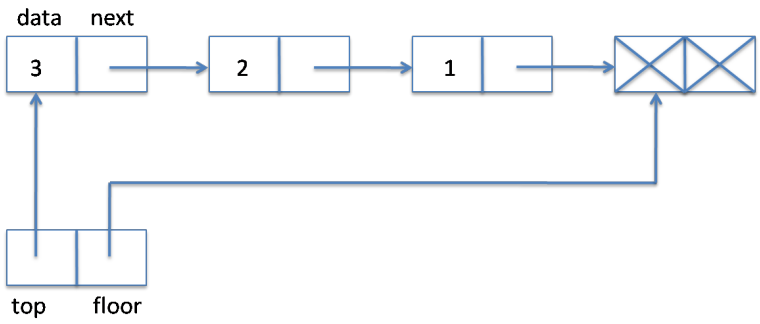
We define:

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

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

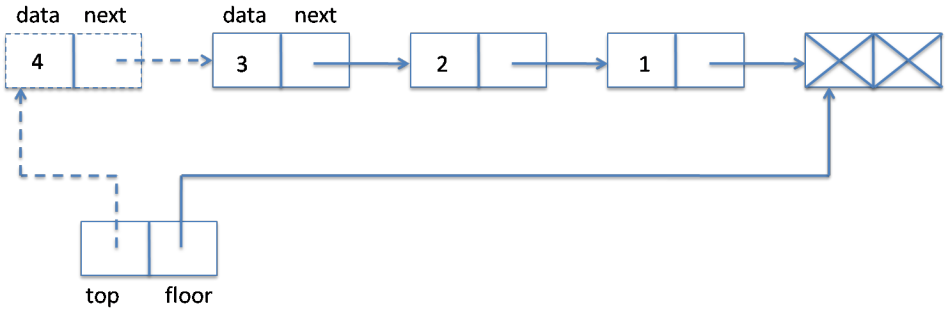
Popping from a stack requires taking an item from the front of the linked list, which is much like dequeuing.

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

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 x)
//@requires is_stack(S);
//@ensures is_stack(S);
{
  list* p = alloc(list);  // Allocate a new top node
  p->data = x;
  p->next = S->top;
  S->top = p;
}
```

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

```
typedef stack* stack_t;
```

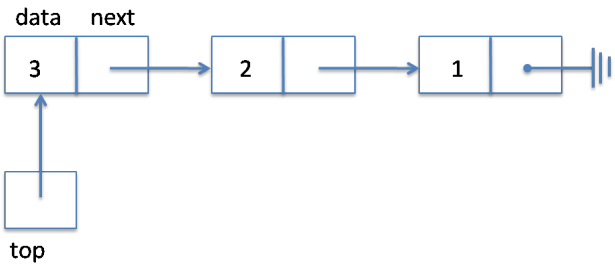
This completes the implementation of stacks.

6 Sharing

We observed in the last section that the `floor` 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 `floor` 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:

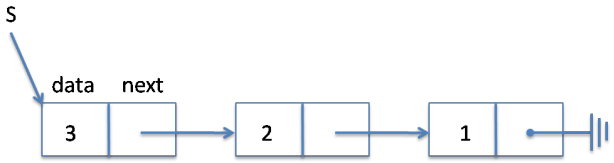


But, then, why have a header at all? Can't we define the stack simply to be the linked list pointed by `top` instead?

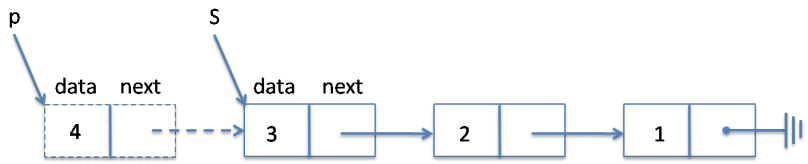
Eliminating the header would lead to a redesign of the interface and therefore to changes in the code that the client writes. Specifically,

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

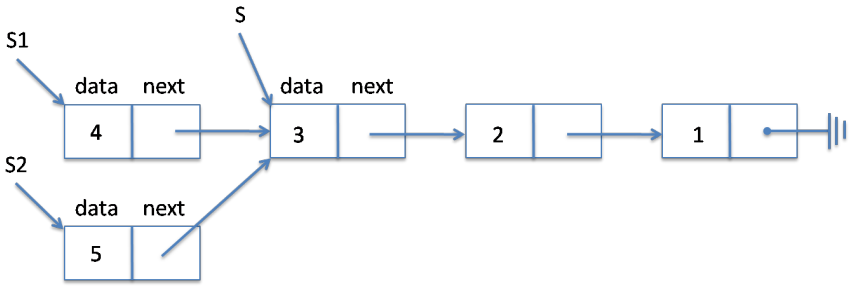
```
stack_t push(stack_t S, elem x);
```

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).

7 Exercises

Exercise 1 (sample solution on page 22)**.** *Define the function*

```
bool is_sum(list* start, list* end, int sum);
```

that checks that the sum of all nodes in the segment from `start` *to* `end` *is equal to* `sum`*. You may assume that the data contained in each node is an integer. How should it behave if the segment is empty?*

Exercise 2 (sample solution on page 22)**.** *Define the function*

```
int lseg_len(list* start, list* end)
/*@requires is_segment(start, end); @*/ ;
```

that returns the number of elements in the list segment [`start`,`end`)*.*

Exercise 3 (sample solution on page 22)**.** *Define the function*

```
elem ith(list* l, int i)
/*@requires i >= 0; @*/ ;
```

that returns the data in `i`*-th elements in list* `l` *(counting from 0). If there are fewer than* `i` *elements before encountering a* `NULL` *pointer, an assertion should fail.*

What is the asymptotic complexity of calling `ith(l,i)` *on a list* `l` *with* n *elements, assuming* $0 <= i < n$*?*

Exercise 4 (sample solution on page 23)**.** *Define the following specification functions on list segments with integer elements:*

```
bool is_in_lseg(int x, list* start, list* end)
/*@requires is_segment(start, end); @*/ ;

bool is_sorted_lseg(list* start, list* end)
/*@requires is_segment(start, end); @*/ ;
```

The first returns `true` *if* `x` *occurs in the list segment* [`start`,`end`) *and* `false` *otherwise. The second returns* `true` *if the input list segment is sorted in ascending order.*

Exercise 5 (sample solution on page 24)**.** *The function* `ith(l,i)` *you defined in an earlier exercise works just like an array access* `A[i]`*, except that it does so on a linked list. Using it and other functions you wrote for previous exercises, implement a version of binary search that operates on list segments. For simplicity, you may assume that the type* `elem` *of data elements has been defined to be* `int`*. Here's the function prototype.*

```
int lseg_binsearch(int x, list* start, list* end)
//@requires is_segment(start, end);
//@requires is_sorted_lseg(l, start, end);
/*@ensures (\result == -1 || !is_in_lseg(x, start, end))
        || (0 <= \result && \result < lseg_len(start, end) &&
            ith(start, \result) == x);
@*/ ;
```

What is the asymptotic complexity of calling `lseg_binsearch(x,start,end)` *on a list segment* [`start`, `end`) *with* n *elements?*

Exercise 6 (sample solution on page 24)**.** *Recall the tortoise-and-hare implementation of circularity checking:*

```
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; // hare is faster and hits NULL quicker
9      t = t->next;
10   }
11   //@assert h == t;
12   return false;
13 }
```

We cannot proved the assertion `t != NULL` *on line 8 with the given loop invariants. Why? What loop invariants would allow us to prove that this assertion holds? Can we write loop invariants that allow us to prove, when the loop exits, that we have found a cycle?*

Exercise 7 (sample solution on page 26)**.** *Here's a simple idea to check that a linked list is acyclic: first, we make a copy* `p` *of the* `start` *pointer. Then when we advance* `p` *we run through an auxiliary loop to check if its next element is already in the list. The code would be something like this:*

```
bool bad_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?

Exercise 8 (sample solution on page 26)**.** *In this chapter, we validated a segment from nodes* `start` *to* `end` *by first calling* `is_acyclic(start)` *to make sure there is no cycle starting at* `start`*, and then by calling* `is_segment(start,end)` *to make sure* `start` *is connected to* `end`*. There is one situation however where this approach does not return the expected result. What is this situation? Once you have identified it, write a specification function* `is_definite_segment` *that is immune from this issue, thereby returning* `true` *on all valid segments and* `false` *on all invalid segments.*

Exercise 9 (sample solution on page 27)**.** *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 10 (sample solution on page 27)**.** *Complete the implementations of stack as defined at the beginning of Section 6, dispensing with the* `floor` *pointer, terminating the list with* `NULL` *instead.*

Exercise 11 (sample solution on page 28)**.** *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 12 (sample solution on page 29)**.** *We say "on the* i^{th} *iteration of our naive* `is_segment` *loop, we know that we can get from* `start` *to* `p` *by following exactly* i *pointers." Write a function*

```
is_reachable_in(list* start, list* end, int numsteps)
/*@requires numsteps >= 0; @*/ ;
```

This function should return `true` *if we can get from* `start` *to* `end` *in exactly* `numsteps` *steps. Use this function as a loop invariant for* `is_segment`*.*

Exercise 13 (sample solution on page 30)**.** *What happens when we swap the order of the last two lines in the* `enq` *function and why? For reference, here's our original code:*

```
void enq(queue* Q, string s)
//@requires is_queue(Q);
//@ensures is_queue(Q) && !queue_empty(Q);
{
  list* l = alloc(list);
  Q->back->data = s;
  Q->back->next = l;
  Q->back = l;
}
```

Exercise 14. *Write an interface and implementation of a* double-ended queue *where we can add and remove elements at both ends. Make sure that all operations you specify can be implemented in constant time.*

Sample Solutions

Solution of exercise 1
The sum of all the elements in a list (or array) segment would be defined recursively as the first element plus the sum of the rest of the segment. Then, it is natural to define the sum of an empty segment to be zero. Thus, `is_sum(start, end, n)` on an empty segment would return `true` exactly when `n == 0`.

```
bool is_sum(list* start, list* end, int sum)
//@requires is_segment(start, end);
{
  list* l = start;
  int n = 0;
  while (l != end) {
    n += l->data;
    l = l->next;
  }
  return n == sum;
}
```

Solution of exercise 2
The function `lseg_len` is defined as follows:

```
int lseg_len(list* start, list* end)
//@requires is_segment(start, end);
{
  int n = 0;
  for (list* p = start; p != end; p = p->next)
  //@loop_invariant p != NULL;
  {
    n++;
  }
  return n;
}
```

Solution of exercise 3
For a change, we will assume the elements of the list have type `elem`, which we could have defined as anything. We implement the function `ith` as follows:

```
elem ith(list* l, int i)
//@requires i >= 0;
{
  for (list* p = l; p != NULL; p = p->next) {
    if (i == 0) return p->data;
    i--;
  }
  assert(false);   // i is greater than the length of l
  return p->data;  // possibly unsafe but unreachable
}
```

For an n-element list `l`, the asymptotic complexity of `ith(l,i)` is $O(n)$.

Solution of exercise 4
Sample implementations of `is_in_lseg` and `is_sorted_lseg` are as follows:

```
bool is_in_lseg(int x, list* start, list* end)
//@requires is_segment(start, end);
{
  for (list* p = start; p != end; p = p->next)
  //@loop_invariant p != NULL;
  {
    if (p->data == x) return true;
  }
  return false;
}

bool is_sorted_lseg(list* start, list* end)
//@requires is_segment(start, end);
{
  if (start == end)  // empty list segment
    return true;

  int x = start->data;
  for (list* p = start->next; p != end; p = p->next)
  //@loop_invariant p != NULL;
  {
    if (x > p->data) return false;
    x = p->data;
  }
  return true;
}
```

Solution of exercise 5

Here is the code for `lseg_binsearch`. It differs from the code for binary search on arrays in that we use `ith(start,i)` in all places where our original code used `A[i]`, and for the use of the adaptations of some of the `arrayutil` specification functions.

```
int lseg_binsearch(int x, list* start, list* end)
//@requires is_sorted_lseg(start, end);
/*@ensures (\result == -1 || !is_in_lseg(x, start, end))
        || (0 <= \result && \result < lseg_len(start, end) &&
            ith(start, \result) == x); @*/
{
  int lo = 0;
  int hi = lseg_len(start, end);

  while (lo < hi)
  //@loop_invariant 0 <= lo && lo <= hi && hi <= lseg_len(start, end);
  {
    int mid = lo + (hi - lo)/2;
    //@assert lo <= mid && mid < hi;

    int mid_data = ith(start, mid);
    if (mid_data == x) return mid;
    if (mid_data < x) {
      lo = mid+1;
    } else { //@assert mid_data > x;
      hi = mid;
    }
  }
  //@assert !is_in_lseg(x,start,end);
  return -1;
}
```

The complexity of `lseg_binsearch` on a list segment of length n is $O(n \log n)$: it makes $\log n$ accesses to the list (just like binary search on arrays) but each access now costs $O(n)$.

Solution of exercise 6

We cannot prove the assertion `t != NULL` because we do not have anything to point to about the value of `t`.

One natural candidate loop invariant is `t != NULL`. Proving that this candidate loop invariant holds initially is immediate by lines 2 and 4. We stumble on preservation however: knowing that `t != NULL` at the start of

an arbitrary iteration of the loop does not allow us to conclude anything about the value of `t->next`, which is the value of `t` at the end of this iteration.

The crucial loop invariant we are missing is the information that the tortoise `t` will be able to travel to the current position of the hare `h` 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 adding the loop invariant `is_segment(t,h)` in `is_acyclic`:

```
bool is_acyclic(list* start) {
  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;
    h = h->next->next;
    t = t->next;
  }
  //@assert h == t;
  return false;
}
```

Proving the validity of our added loop invariant is routine at this point, and we leave it as an extra exercise.

Note that the version of `is_segment` we wrote in this chapter does not have `is_acyclic` as a precondition. Had we provided such a precondition, these two specification functions would be *mutually recursive*, which would greatly complicate proving the validity of our added loop invariant, or make the proof impossible if this could trigger an infinite recursion. Fortunately, an infinite recursion cannot happen. The key insight comes from complexity analysis: the hare and the tortoise will never be farther apart than the size of the cycle. This is not a point-to proof however.

The added loop invariant implies that `t` is not `NULL` since, whenever `is_segment(start,end)` is `true`, neither `start` nor `end` can be `NULL`.

At the end of the loop, we know that `h == t` by the loop guard. This is not sufficient to prove that there the node they point to is part of a cycle: any empty segment has its start pointer equal to its end pointer whether there is cycle or not. For a cycle to be present, we need to know that we can reach

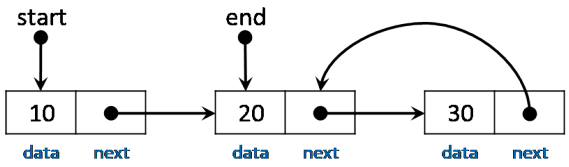
`t` from `t->next`. But observe that we set up the loop so that `t->next == h` and each iteration sets these pointers further apart. This suggests adding `is_segment(t->next,h)` as an additional loop invariant. Together with the loop guard, it allows us to prove that we can reach `t` from `t->next`.

Solution of exercise 7
The code does not work when the input-list is a self-loop, as in the following example:

```
int main() {
  list* a = alloc(list);
  a->next = a;                          // self loop
  assert(bad_is_acyclic(a));
  return 0;
}
```

As the first execution of the outer loop is executed, the loop guard of the inner loop is immediately `false`. Since `p->next` is equal to `p`, the outer loop runs for ever.

Solution of exercise 8
Consider a list segment that contains a cycle past its end pointer. Here's an example:



The call `is_acyclic(start)` will reject this list right away although there *is* a valid segment from `start` to `end`.

We can fix this issue modifying `is_acyclic` to check if `end` has been reached. To do so, we add `end` as an additional argument to our function (renamed `is_definite_segment`) and return `true` if `start` is `end` or if the hare passes or reaches `end`. We also need to modify the `NULL` cases to return `false` instead of `true` as they do not identify the desire segment.

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

This enhanced version of `is_acyclic` subsumes our original `is_segment`: we do not need it anymore.

Solution of exercise 9

In the implementation of `pop`, we return the data element in node pointed to by the `top` field of the stack header. Assuming this empty stack is valid, the node that `top` points to is the same that the `floor` field points to, which is a dummy node. This `pop` would have two problems:

1. The element returned is the value of the dummy node, which is unspecified. The stack being empty, it is certainly a value that is not contained in the stack.

2. In addition to returning the data element at the top of the stack, `pop` sets `top` to `top->next`. This pointer too is unspecified. One of two things can happen as execute this instruction:

   - `top->next` is `NULL`, which entails that the next call to `pop` will dereference the `NULL` pointer, thereby aborting the program.
   - `top->next` is not `NULL`, in which case the next `pop` will produce another element that is not in the stack.

   In both cases, the stack would be invalid. Were contracts enabled, this would be caught by the call to `is_segment` within `is_stack`.

Solution of exercise 10

We only display the changes with respect to the implementation of stacks shown earlier in this chapter

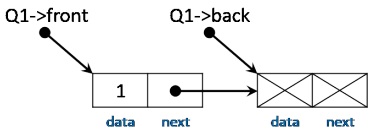
```
typedef struct stack_header stack;
struct stack_header {
  list* top;
};

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

stack* stack_new()
//@ensures is_stack(\result);
{
  stack* S = alloc(stack);
  return S;
}

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

A first change of note is in the specification function `is_stack`: we do not use `is_segment` since we are not using list segments in this implementation (there is no end-of-segment to check). We need however to check that the underlying list is `NULL`-terminated. This equivalent to requiring that it does not contain a cycle. In fact, a linked list is acyclic if and only if it is `NULL`-terminated.

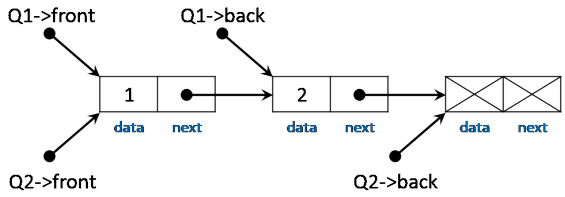
The remaining changes have to do with the end-of-list terminator. Since `NULL` represents the empty (`NULL`-terminated) list, we check that a stack is empty by simply testing whether the underlying list is `NULL`. The function `stack_new` may at first appear like it is missing some instructions: recall however that the default pointer value is `NULL`. Therefore, the `top` field of the returned stack `S` has implicitly been initialized to the empty linked list.

Solution of exercise 11

Let's start with the following 1-element queue `Q1`, with `1` as its data:

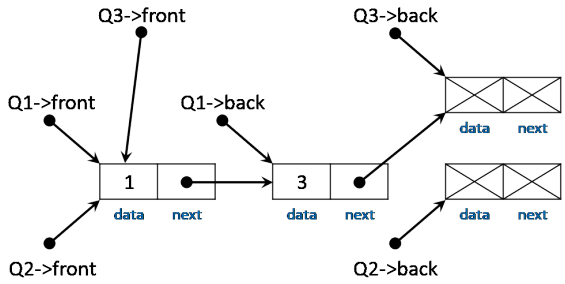


Next, let's enqueue `2` into `Q1`. We get back the queue `Q2` which point to the front and back of the updated underlying list segment. Note that the front and back pointers of `Q1` still point to it's original list segment



So far so good: we have two queues in memory, which share two nodes.

Next, let's enqueue `3` into `Q1`. We write this value into the `data` field of the dummy node of `Q1`, which happens to be the node containing the value of `Q2` we just added. We also point the `next` field of that node to a new dummy node. The resulting memory layout is as follows:



But note that this last step destroyed `Q2`: the start and end nodes of `Q2` are not connected by a valid segment any more.

Sharing works wonderfully for stacks, but not for queues.

Solution of exercise 12

The code for `is_reachable_in` can be written in many ways. Here is one:

```
bool is_reachable_in(list *start, list* end, int numsteps)
//@requires numsteps >= 0;
{
  for (list *l = start; l != end; l = l->next) {
    if (l == NULL) return false;
    numsteps--;
  }
  if (numsteps == 0) return true;
  return false;
}
```

Having written this function, we can write an iterative variant of `is_segment` that increments a counter at each iteration of the main loop. We can then provide a loop invariant that uses `is_reachable_in` with the start node, the current node and this counter as arguments. The code is as follows:

```
bool is_segmentC(list* start, list* end) {
  int numsteps = 0;
  for (list* p = start; p != NULL; p = p->next)
  //@loop_invariant is_reachable_in(start, p, numsteps);
  {
    if (p == end) return true;
    numsteps++;
  }
  return false;
}
```

Solution of exercise 13

The first two lines of `enq` create a new dummy node and store the element to enqueue into the old dummy node. In the last two lines of `enq`, we assign the pointers so that the old dummy node points to the new dummy node and reassign the new `back` pointer of the queue to this new dummy node. If we swap the order of the last two lines, we first assign `Q->back` to the new dummy node. However, by doing so, we lost access to the old dummy node, where the element we just enqueued is stored, and cannot reassign its `next` pointer to the new dummy node (without traversing the entire queue).