1 Introduction

In this lecture we introduce queues as a data structure and linked lists that underly their implementation. In order to implement them we need recursive types, which are quite common in the implementation of data structures.

Relating this to our learning goals, we have

Computational Thinking: We will encounter abstraction in a central way.

Algorithms and Data Structures: We are looking at queues and (briefly) stacks as well as linked lists.

Programming: We will see structs and pointers, and the use of recursion in the definition of structs.

2 The Queue Interface

A queue is a data structure where we add elements at the back and remove elements from the front. In that way a queue is like “waiting in line”: the first one to be added to the queue will be the first one to be removed from the queue. This is also called a FIFO (First In First Out) data structure. Queues are common in many applications. For example, when we read a book from a file as in Assignment 2, it would be natural to store the the words in a queue so that when we are finished reading the file the words are in the order they appear in the book. Another common example are
Queues

buffers for network communication that temporarily store packets of data arriving on a network port. Generally speaking, we want to process them in the order that they arrive.

Before we consider the implementation to a data structure it is helpful to consider the interface. We then program against the specified interface. Based on the description above, we require the following functions:

```c
bool queue_empty(queue Q); /* O(1), check if queue is empty */
queue queue_new(); /* O(1), create new empty queue */
void enq(queue Q, string s); /* O(1), add item at back */
string deq(queue Q); /* O(1), remove item from front */
```

We can write out this interface without committing to an implementation of queues. In particular, the type `queue` remains abstract in the sense that we have not given its definition. This is important so that different implementations of the functions in this interface can choose different representations. Clients of this data structure should not care about the internals of the implementation. In fact, they should not be allowed to access them at all and operate on queues only through the functions in this interface. Some languages with strong module systems enforce such abstraction rigorously. In C, it is mostly a matter of adhering to conventions.

### 3 Using the Queue Interface

We play through some simple examples to illustrate the idea of a queue and how to use the interface above. We write a queue as

\[ x_1, x_2, \ldots, x_n \]

where \( x_1 \) is the front of the queue and \( x_n \) is the back of the queue. We enqueue elements in the back and dequeue them from the front.

For example:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Command</th>
<th>Other variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &quot;a&quot; )</td>
<td><code>queue Q = queue_new();</code></td>
<td><code>string s = deq(Q);</code></td>
</tr>
<tr>
<td>( &quot;b&quot; )</td>
<td><code>enq(Q, &quot;a&quot;);</code></td>
<td><code>s = &quot;a&quot;</code></td>
</tr>
<tr>
<td>( &quot;a&quot;, &quot;b&quot; )</td>
<td><code>enq(Q, &quot;b&quot;);</code></td>
<td><code>s = &quot;a&quot;</code></td>
</tr>
<tr>
<td>( &quot;b&quot;, &quot;c&quot; )</td>
<td><code>string s = deq(Q);</code></td>
<td><code>s = &quot;b&quot;</code></td>
</tr>
<tr>
<td>( &quot;c&quot; )</td>
<td><code>enq(Q, &quot;c&quot;);</code></td>
<td><code>s = &quot;b&quot;</code></td>
</tr>
</tbody>
</table>
4 Structs and Pointers

A *struct* is just an aggregate type, consisting of several data elements stored together of potentially different types. Compare this to arrays, which is an aggregate of elements of the same type. Structs must be explicitly declared. For example, a struct representing a point in two-dimensional space (such as the location of a pixel in an image), could be declared as

```c
struct point {
    int x;
    int y;
    bool visited;
};
```

Here `x`, `y`, and `visited` are not variables, but *fields* of the struct. The declaration expresses that every point has `x` and `y` fields, intended to represent its coordinates, and a `visited` field that might be used in a program, say, to computed shortest paths between points.

Structs do not necessarily fit into a machine word because they can have arbitrarily many components, so they must be allocated on the heap (in memory, just like arrays). This is true if they happen to be small enough to fit into a word in order to maintain a uniform and simple language.

How, then, do we manipulate structs? We use the same solution as for arrays: we manipulate them via their address in memory. Instead of `alloc_array` we call `alloc` which returns a *pointer* to the struct that has been allocated in memory. Let’s look at an example in coin.

```c
% coin coord.c0
Coin 0.3.0 'Nickel'(r103, Mon Aug 27 15:30:29 EDT 2012)
Type '#help' for help or '#quit' to exit.
--> struct point* p = alloc(struct point);
p is 0xFF4FFF80 (struct point*)
--> p->x;
0 (int)
--> p->y;
```

We can access the fields of a structs, for reading or writing, through the notation `p -> f` where `p` is a pointer to a struct, and `f` is the name of a field in that struct. Continuing above, let’s see what the default values are in the allocated memory.

```c
--> p->x;
0 (int)
--> p->y;
```
0 (int)
--> p->visited;
false (bool)
-->

We can write to the fields of a struct by using the arrow notation on the left-hand side of an assignment.

--> p->x = 438;
(*p).x is 438 (int)
--> p->x;
438 (int)
-->

The notation (*p).f is a longer form of p->f. First, *p follows the pointer to arrive at the struct in memory, then .f selects the field f. We will rarely use this dot-notation (*p).f in this course, preferring the arrow-notation p->f.

5 Pointers

As we have seen in the previous section, a pointer is needed to refer to a struct that has been allocated on the heap. It can also be used more generally to refer to an element of arbitrary type that has been allocated on the heap. For example:

% coin
Coin 0.3.0 'Nickel'(r103, Mon Aug 27 15:30:29 EDT 2012)
Type '#help' for help or '#quit' to exit.
--> int* p = alloc(int);
p is 0xFF4FFF80 (int*)
--> *p = 3;
*(p) is 3 (int)
--> *p;
3 (int)
-->

In this case we refer to the value using the notation *p, either to read (when we use it inside an expression) or to write (if we use it on the left-hand side of an assignment).

So we would be tempted to say that a pointer value is simply an address. This is not quite correct: there is also a special value NULL. Its main
feature is that NULL is not a valid address, so we cannot dereference it to obtain stored data. For example:

```c
% coin
Coin 0.3.0 'Nickel'(r103, Mon Aug 27 15:30:29 EDT 2012)
Type '#help' for help or '#quit' to exit.
--> int*p = NULL;
p is NULL (int*)
--> *p;
Error: null pointer was accessed
Last position: <stdio>:1.1-1.3
```

To rephrase, we say that a pointer value is an address, of which there are two kinds. A valid address is one that has been allocated explicitly with alloc, while NULL is an invalid address. In C, there are opportunities to create many other invalid addresses, as we will discuss in another lecture.

Attempting to dereference the null pointer is a safety violation in the same class as trying to access an array with an out-of-bounds index. In C0, you will reliably get an error message, but in C the result is undefined and will not necessarily lead to an error. Therefore:

> Whenever you dereference a pointer p, either as *p or p->f, you must have a reason to know that p cannot be NULL.

In many cases this may require function preconditions or loop invariants, just as for array accesses.

## 6 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 choose strings, for purposes of illustration. Interestingly, none of the code actually depends on what type is chosen, but this genericity is not yet explicit. These considerations give rise to the following definition:
struct list_node {
    string data;
    struct list_node* next;
};
typedef struct list_node list;

This definition is an example of a recursive type. A struct of this type contains a pointer to another struct of the same type, and so on. We usually use the special element of type t*, namely NULL, to indicate that we have reached the end of the list. Sometimes (as will be the case for queues introduced next), we can avoid the explicit use of NULL and obtain more elegant code. The type definition is there to create the type name list, which stands for 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

```c
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.
7 The Queue Implementation

Here is a picture of the queue data structure the way we envision implementing it, where we have elements 1, 2, and 3 in the queue.

A queue is implemented as a struct with a front and back field. The front field points to the front of the queue, the back field points to the back of the queue. 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 arrays, we often work with the length which is one greater than the index of the last element in the array. In queues, we use a similar strategy, making sure the back pointer points to one element past the end of the queue. Unlike arrays, there must be something in memory for the pointer to refer to, so there is always one extra element at the end of the queue which does not have valid data or next pointer. We have indicated this in the diagram by writing X.

The above picture yields the following definition.

```c
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.

```cpp
bool is_queue(queue Q) {
    if (Q == NULL) return false;
    if (Q->front == NULL || Q->back == NULL) return false;
    return is_segment(Q->front, Q->back);
}
```

Next, the code for checking whether two pointers delineate a list segment. When both start and end are NULL, we consider it a valid list segment, even though this will never come up for queues. It is a common code pattern for working with linked lists and similar data representation to have a pointer variable, here called p, that is updated to the next item in the list on each iteration until we hit the end of the list.

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

Here we stop in two situations: if p = NULL, then we cannot come up against end any more because we have reached the end of the list and we return false. The other situation is if we find end, in which case we return true since we have a valid list segment. This function may not terminate if the list contains a cycle. We will address this issue in the next lecture; for now we assume all lists are acyclic.

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

```c
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:

![Diagram of dequeue operation]

And in code:

```c
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

\[
Q->\text{front}->\text{data}
\]

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 || Q->back == NULL) return false;
    return 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.
Exercises

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

Exercise 2  Our queue design always “wasted” one element that we marked X. Can we save this memory and implement the queue without extra elements? What are the tradeoffs and alternatives when implementing a queue?