Week 02 Lab Exercise

ADTs, Analysis of Algorithms

Objectives

- To re-acquaint you with the Queue ADT
- To understand the difference between interface and implementation in ADTs
- To manipulate an array-based data structure
- To implement core queue operations
- To analyse the time complexity of some algorithms
- To benchmark different implementations of an ADT

Admin

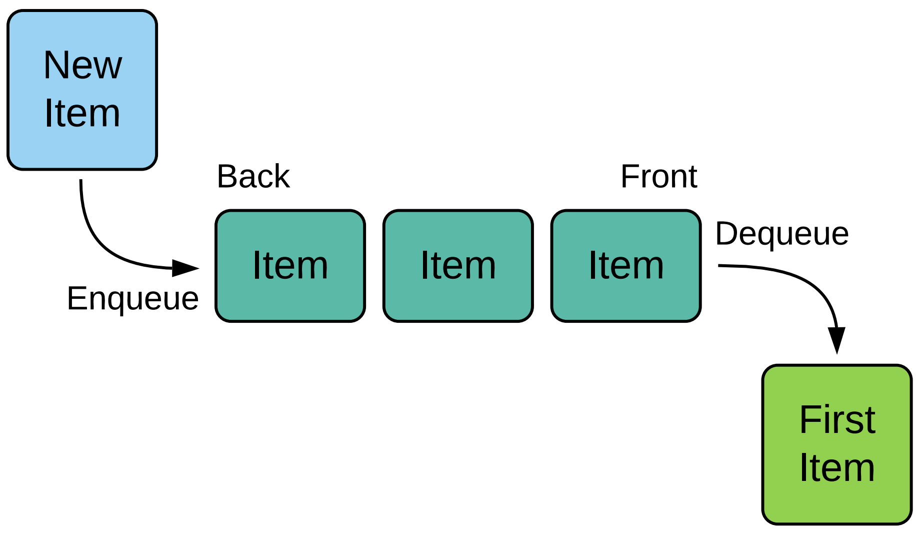
Marks	5 (see Assessment section for more details)
Demo	in the Week 02 or Week 03 lab
Submit	see the Submission section
Deadline	submit by 5pm Sunday of Week 02
Late penalty	0.5 marks per day

Background

At the end of COMP1511, you had a brief look at the Queue ADT. Since the Queue is an important data structure that we'll be using in some tree and graph algorithms, it would be useful to have a look at it again in more detail. We'll also use it as an opportunity to perform some time complexity analysis.

Queues and ADTs

You should be reasonably familiar with the concept of a queue in the real world. The way it works in computing is exactly the same! A queue is a linear data structure, with a front and a back. Items enter the queue by joining the back of the queue, and items are removed one at a time from the front of the queue. The operations of adding an item to the back and removing an item from the front are known as enqueuing and dequeuing respectively.



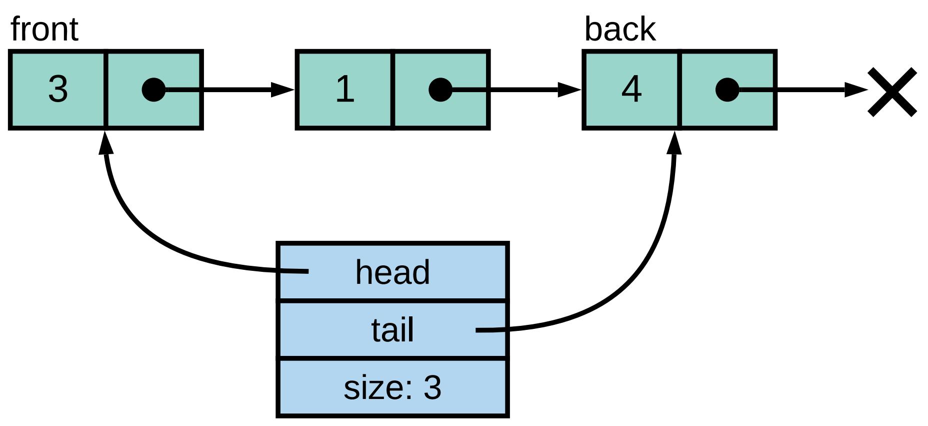
An important thing to emaphasise is that the Queue is an **abstract data type (ADT)**. This means that users of the Queue know *what* operations can be performed on the Queue (e.g., enqueue, dequeue), but they do not need to know *how* the Queue is implemented. The operations that can be performed on an ADT are collectively called the interface to the ADT, and are declared in a header (*.h) file, while the implementation details go in the corresponding *.c file. The implementation (the *.c file) is not directly #included in the program, which means that the user of the ADT does not have access to the internal representation of the ADT (i.e., what fields does it contain and what concrete data structures does it use) and does not know how the functions declared in the header file are implemented.

Queue Implementations

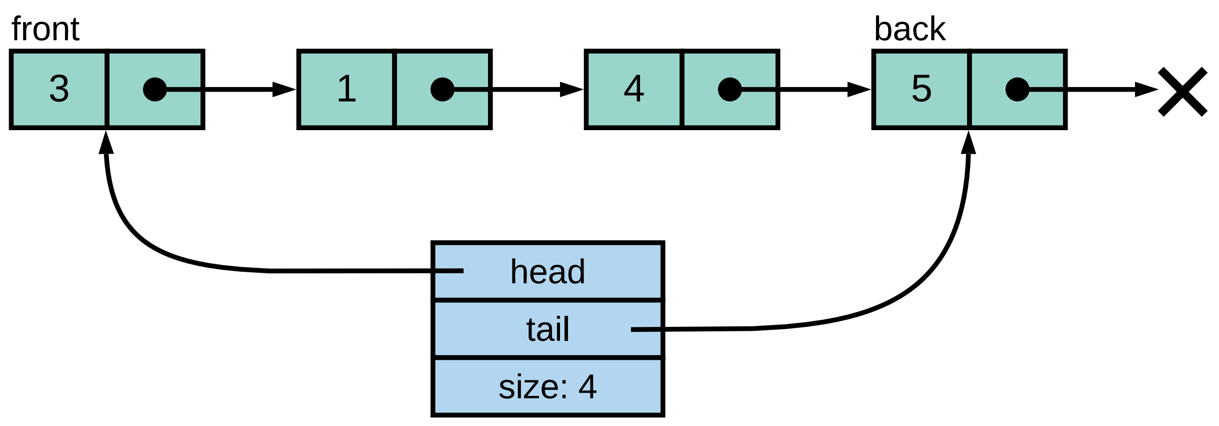
Since the Queue is an ADT, it could be implemented in several different ways, for example, using a linked list or using an array. The choice of implementation determines the time and space complexities of the operations. Here are the different queue implementations we will explore:

List Queue

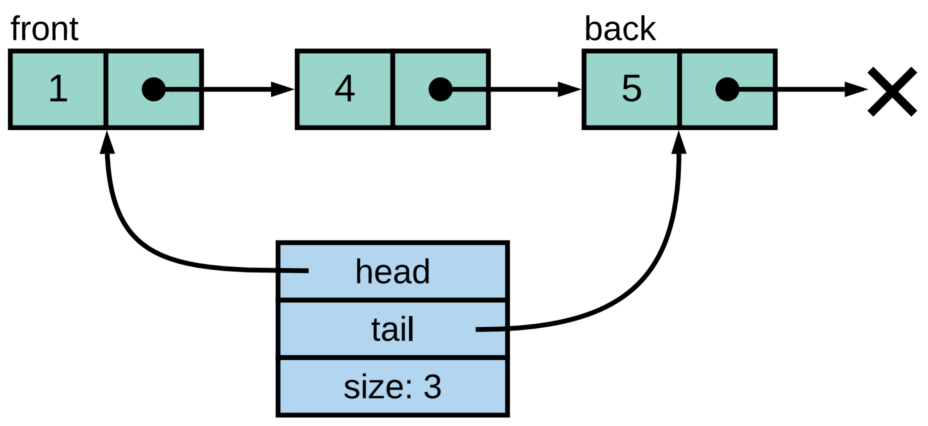
In this implementation, the items in the queue are stored in a linked list. To enqueue an item, the item is added to the end of the list, and to dequeue an item, the item is removed from the beginning of the list. To enable both operations to be efficient, the queue contains pointers to the first and last nodes in the list. Here are some diagrams demonstrating the enqueue and dequeue operations on the list queue:



After enqueuing 5:

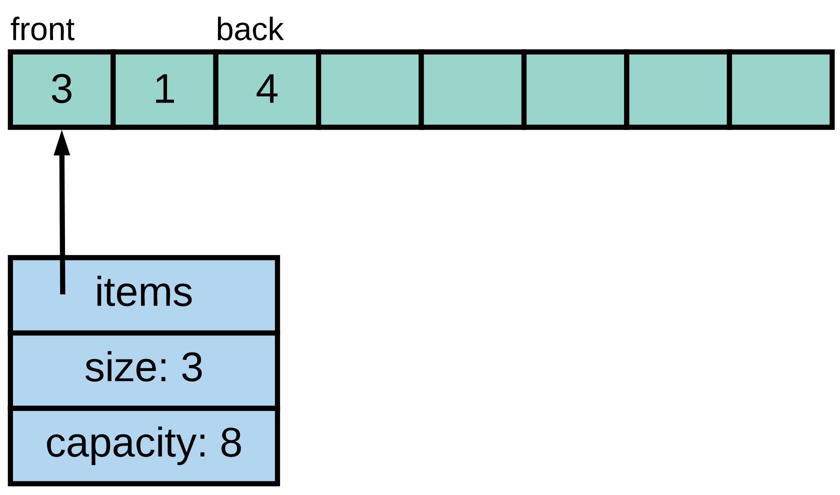


After dequeuing 3:

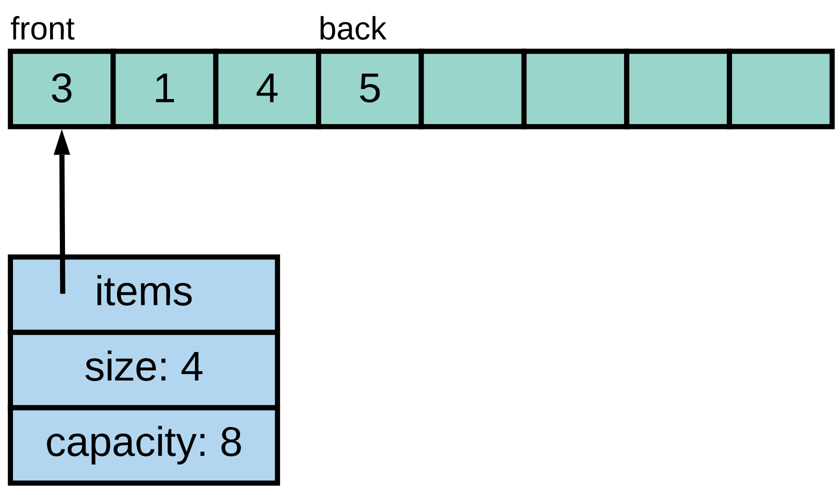


Array Queue

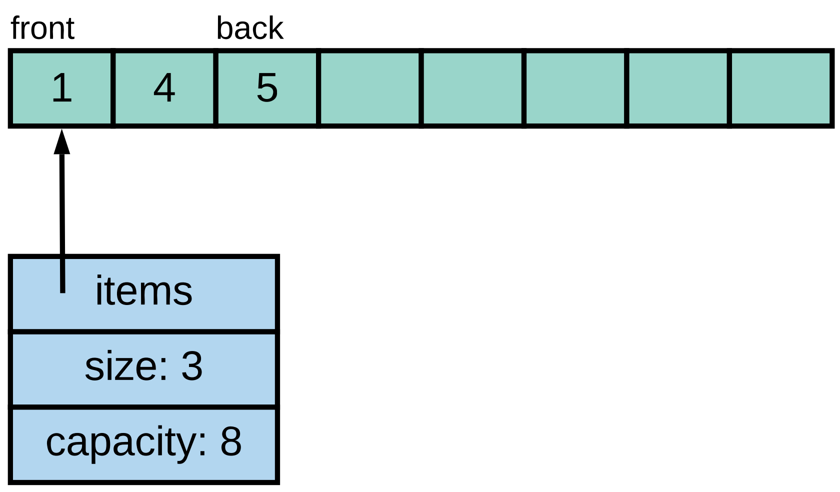
In this implementation, the items in the queue are stored in an array. To enqueue an item, the item is simply placed in the next available slot in the array, and to dequeue an item, the item at index 0 is removed and the rest of the items are shifted down. Since the implementation uses an array, the array may become full and if more items need to be inserted, the array will need to be expanded. Also, since the array will usually not be full, we will need to keep track of both the number of items (i.e., the size of the queue) and the size of the array (i.e., the capacity). Here are some diagrams demonstrating the enqueue and dequeue operations on the array queue:



After enqueuing 5:

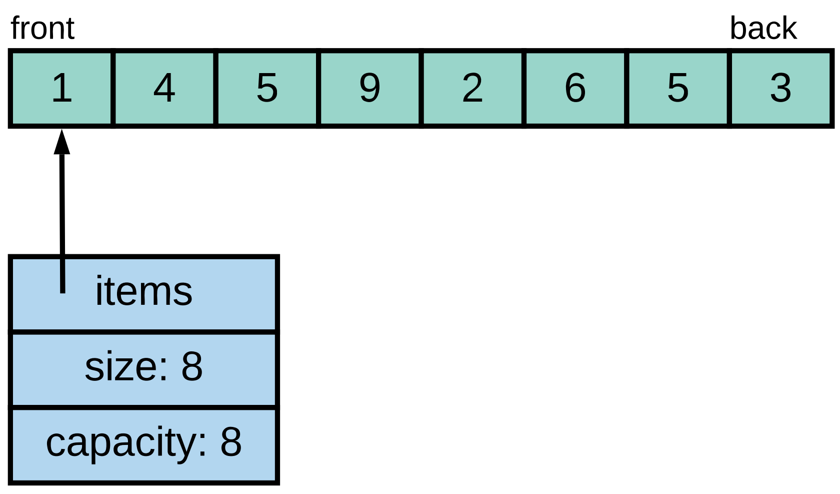


After dequeuing 3:



Note: In these diagrams, when an array slot is empty, that simply means that the value in that slot is irrelevant.

The following diagrams show the behaviour of the array queue when the queue becomes full and more items need to be inserted:



After enqueuing 5:

To make room for the new item, the array is resized to double its original size. Note that the capacity field is updated to reflect the new size of the array.

Now take some time to compare the linked list queue and array queue. Consider the advantages and disadvantages of each.

Circular Array Queue

This implementation is similar to the array queue, except that when we dequeue, we don't shuffle the rest of the items down - we simply leave them and allow the back of the queue to wrap around the end of the array. If enough enqueues and dequeues are performed, the queue may circle around the array several times. Since the front of the queue is not necessarily at index 0 anymore, another variable is needed to keep track of the index containing the item at the front of the queue. The following diagrams demonstrate the behaviour of the queue:

After enqueuing 5:

After dequeuing 3:

When enough enqueues and dequeues are performed, the back of the queue will wrap around the end of the array, as shown in the following diagrams:

After enqueuing 3:

Note that when the array becomes full, we will still need to resize the array if another item needs to be enqueued. One of the tasks in this lab will involve working out what needs to be done during a resizing.

Setting Up

Create a directory for this lab, change into it, and run the following command:

```
$ unzip /web/cs2521/21T1/labs/week02/downloads/lab.zip
```

If you're working at home, download `lab.zip` by clicking on the above link and then run the `unzip` command on the downloaded file.

If you've done the above correctly, you should now have the following files:

`Makefile`	a set of dependencies used to control compilation
`Queue.h`	interface to the Queue ADT
`ListQueue.c`	implementation of the Queue ADT using a linked list (complete)
`ArrayQueue.c`	implementation of the Queue ADT using an array (complete)
`CircularArrayQueue.c`	implementation of the Queue ADT using a circular array (incomplete)
`testQueue.c`	tests for the Queue ADT
`answers.txt`	a template for you to fill in your answers for Task 2
`benchmarkQueue.c`	benchmark test for the Queue ADT (incomplete)

Before you start using these programs, it's worth having a quick look at the code, especially the Queue ADT interface and its various implementations. Are there any constructs that you don't understand? Try to work them out with your lab partner, or ask your tutor.

Once you've understood the programs, run the command:

```
$ make test
```

This will leave three executable files in your working directory (along with some `.o` files):

`testListQueue`, `testArrayQueue`, `testCircularArrayQueue`
These executables test each of the queue implementations. All of the tests come from the same source file `testQueue.c`. To run them, all you need to do is enter the name of the executable (preceded by a dot and slash) and then press enter. Note that `testCircularArrayQueue` currently fails as the `CircularArrayQueue` implementation is incomplete.

Task 1

Complete the `CircularArrayQueue` implementation by implementing the enqueuing and dequeuing functions in `CircularArrayQueue.c`.

To get an idea of how these functions should behave, you can refer to the diagrams above. Keep in mind that your enqueuing function must be able to handle the situation where the array containing the items is full and needs to be expanded.

HINT:

A good rule of thumb is to double the size of the array each time it needs to be expanded. You may want to look into *realloc(3)* for expanding the array.

Once you think that you have got the functions working, recompile and run the tests for `CircularArrayQueue`:

```
$ make test
...
$ ./testCircularArrayQueue
```

These tests are by no means comprehensive - passing the tests does not guarantee that your implementation is correct. When we mark your lab, our tests will cover more edge cases. We recommend that you add a few more tests to `testQueue.c` to cover some additional cases, especially those that exercise the array-resizing part of your implementation.

Task 2

Now that all the Queue implementations are working, it's time to do some time complexity analysis!

Open `answers.txt`. This file contains three tables - one for each of the Queue implementations we discussed. To fill in the tables, you will need to examine the time complexities of the enqueuing and dequeuing operations in each implementation. There are three "kinds" of time complexity to determine: (1) the *best* case, (2) the *worst* case, and (3) the *average* case. To explain what these mean, we will refer to the following function that checks whether a given element exists in an unordered array of size n:

```
bool arrayContains(int A[], int n, int elem) {
    for (int i = 0; i < n; i++) {
        if (A[i] == elem) {
            return true;
        }
    }
    return false;
}
```

best case
The best case time complexity occurs when the input is such that the operation can complete as quickly as possible. For the above example, the best case would be where the element being searched for is the first element of the array, because we would return `true` straight away without needing to examine any of the other elements. In this situation, since we only needed to examine one item, regardless of the size of the array, the best case time complexity is $O(1)$.
Important: Saying that the best case time complexity of an operation is $O(1)$ because the input size could be small is invalid reasoning, as you cannot choose the size of the input - the size of the input is arbitrary, i.e., n. So if you claim that the best case time complexity of a function is $O(1)$, your reasoning must be based on something other than input size.

worst case
The worst case time complexity occurs when the input is such that the operation takes as long as possible to complete. For the above example, the worst case would be where the element being searched for is either the last element in the array, or not in the array at all. In these situations, since we need to examine all n items, the worst case time complexity is $O(n)$.

average case
The average case time complexity is the time complexity of the operation when averaged out over many executions on different inputs. For the above example, if the array contains the element being searched for, then on average, half ($n/2$) of the array elements would be examined, and hence the time complexity would be $O(n)$. Meanwhile, if the array does not contain the element being searched for, then we would end up examining all n array elements, and so the time complexity would also be $O(n)$. Since for both of these types of inputs, the time complexity is $O(n)$, the overall average case time complexity is $O(n)$.

Make sure you understand your answers, as your tutor could ask you to explain any of them.

NOTE:

You can assume that *malloc* and *free* are $O(1)$. However, you can't make the same assumption about *realloc* - once you understand what *realloc* does, this should be obvious. See the [documentation for *realloc(3)*].

HINT:

The average case time complexity for enqueueng in the array queue and circular array queue may be difficult to determine due to the occasional resizings. As a hint, consider the following: How often do resizings occur? Do they occur more or less often as more items are enqueued?

Task 3

In the previous task, you may have found that the `CircularArrayQueue` is straight up better than the `ArrayQueue`. However, time complexities *can* be deceiving. For example, if Algorithm A has a smaller time complexity than Algorithm B, that doesn't necessarily mean Algorithm A will *always* perform faster than Algorithm B - that simply means Algorithm A will *eventually* perform faster than Algorithm B when the input size is large enough, but this threshold input size may be unrealistically large such that Algorithm B is actually faster in most cases. Hence, it is usually a good idea to get some concrete evidence to prove that one data structure or algorithm is better than another before you start using it.

This is where *benchmarking* comes in! From the always reliable Wikipedia:

> In computing, a benchmark is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it.

In this task, your goal is to benchmark the `ArrayQueue` and `CircularArrayQueue` to prove that your implementation of the `CircularArrayQueue` from Task 1 is more efficient than the `ArrayQueue`. To do this, there is a stub file `benchmarkQueue.c` from which you can create a Queue and call the standard Queue operations. Try to come up with a sequence of operations which will clearly set the two implementations apart (and hopefully confirm `CircularArrayQueue` as the better one).

> NOTE:
>
> A good benchmark does not have to be very complex. It is perfectly possible to write a good benchmark using fewer than 10 lines of code.

Once you think you have written a good benchmark, run the command:

```
$ make benchmark
...
```

Two executables will be produced - one called `benchmarkArrayQueue` which benchmarks the `ArrayQueue`, and another which benchmarks the `CircularArrayQueue`. Run and time both of these programs with the time command:

```
$ time ./benchmarkArrayQueue

real    xmx.xxxs
user    xmx.xxxs
sys     xmx.xxxs
$ time ./benchmarkCircularArrayQueue

real    xmx.xxxs
user    xmx.xxxs
sys     xmx.xxxs
```

If there is a clear difference in times between the two benchmarks, congratulations! You've successfully used a benchmark to compare two programs. If you are curious, you could also benchmark the `ListQueue` implementation by adding instructions to produce `benchmarkListQueue` to the `Makefile`.

Submission

You need to submit three files: `CircularArrayQueue.c`, `answers.txt` and `benchmarkQueue.c`. You can submit these via the command line using `give` using the command:

```
$ give cs2521 lab02 CircularArrayQueue.c answers.txt benchmarkQueue.c
```

or you can submit them via WebCMS. After submitting them (either in this lab class or the next) show your tutor, who'll give you feedback on your code, ask you about your results, and award a mark.

> NOTE:
>
> If you did not complete all the tasks, you must still submit *all* the required files. This could mean creating an empty file if a required file did not exist before.

Assessment

To receive a mark for this lab, you *must* demonstrate your lab to your tutor during your Week 02 or Week 03 lab session. You will be marked based on the following criteria:

Code correctness (2 marks)
This is the correctness of your code for Part 1. The correctness of your code will be determined by tests that we will run against your submission.

Complexity analysis (2 marks)
This is how accurate you were with the time complexities that you obtained in Part 2 and whether you can explain your answers. You must be prepared to explain any of the time complexities to your tutor.

Benchmarking (1 mark)

This is whether you can explain and justify your benchmarking program for Part 3 - what did you do and why did you expect it to reveal a difference between the `ArrayQueue` and `CircularArrayQueue` implementations?
