Computer Laboratory 11

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Essential information

- This assignment is due Monday Dec 4th at noon and will be turned in on gradescope
- The assignment is intended to be done in pairs, although you're allowed to work on your own if you want.
- You can start the assignment before your lab period, although I recommend not getting too far – the lab TAs often have useful hints, and I don't want you wasting too much time struggling with something that the TAs might announce in-class.
- For more rules see the lab rules document on canvas.

2 Introduction

In this lab we will be building another classic data structure – the queue. The queue is another simple data structure, and one that is well suited to implementations with linked data structures. While an efficient queue *is possible* with an array-based data structure, it is much harder than the linked-chain approach. Through writing this we will:

- Practice implementing a linked-chain data structure
- Take the basic linked-chain manipulations we've seen in lecture, and apply them in a novel way.
- Learn a bit about the Queue Abstract Data Type.

While conceptually tractable, LinkedChain datastructures can be a menace to debug. There are many bugs you might face in the design of this code. The reason these bugs are tricky to debug is that an operation can, by all appearances, succeed, but actually have set the datastructure up for future failure. Each operation is responsible not only for it's own behavior, but to make sure every variable is updated appropriately to allow other operations to run successfully. For example, if an operation accidentally creates a "loop" in the linked chain, future operations may infinitely loop (this can even confuse the debugger!) Or a remove might leave the queue "half empty" with size zero and null front pointer, but non-null rear pointer. Alternatively, a null could be introduced into the code in a place where it doesn't belong, leading to crashes due to a NullPointerException.

Because the debugging of this datastructure can get challenging, the entire lab will be simply writing this data structure. While debugging remember this advice:

- While a failed test is an indication that your code is wrong, a passing test is never a promise that your code is correct. Never *assume* a function isn't part of an issue until you rule it out with debugging strategies such as adding print statements or using the interactive debugger.
- The line of code that is wrong may not be directly related the error you get. If your program crashes on one line of code it is always possible that something wrong happened earlier in the program, but that it only became an issue at this line.
- If you are having a bug in your code it means you are not understanding your code correctly. All bugs trace back to disagreements between you and the machine about what you wanted it to do. You can only begin debugging by accepting that the computer is right, and you are wrong, about what the code you typed means.
- The first step of debugging is gathering more information. Don't aim to fix the bug with every change, but instead start by trying to better understand what your code is doing. Once you know what your code does – and why that isn't correct, fixing the code is often much easier!
- Don't forget to "undo" changes that didn't help solve a problem. As a community I would estimate that programmers have made almost as many bugs as they have fixed while debugging code. By "undoing" changes that don't help solve a problem you can make sure you don't make things harder as you go!

3 Files

This lab will involve the following **provided** files.

- `CacheBlockQueueNodeTest.java`
- `CacheBlockQueueTest.java`

This lab also involves the following files **you will create** in the student package:

- `CacheBlockQueueNode.java`
- `CacheBlockQueue.java`

4 Instructions

Before beginning you should:

1. Setup an IntelliJ project
2. Download the provided files and place them in the src folder. You will likely need to comment code, out before you can test anything.
3. Start with the CacheBlockQueueNode class, the move onto the CacheBlockQueue itself.

5 Theory: What is a Queue

In programming theory, we say that a Queue is any data structure that supports the following operations:

- enqueue(elem) add something to the back of the queue.
- front() return the current front of the queue without modifying the queue
- dequeue() return the current front of the queue, removing it from the queue
- isEmpty() check if the queue is empty.

While these names are unfamiliar, the behaviors should familiar. You are probably quite familiar with queues such as "wait lists" to get into a class, and "lines" at the store. These "storage" systems have two operations: enqueue (put something into the line) and dequeue (remove the thing from line that's been waiting the longest). Another way you can think of this is like a list where we only add things to the end of the list (enqueue) or remove things from the beginning of the list (dequeue). By adding at the end, and removing from the beginning, we keep data stored in the order it was added, with elements closer to the front/beginning of the queue being in the queue longer than elements towards the end/back of the queue. Another term for this is FIFO (First In, First out) because whatever was added to the queue first will be the removed from the queue first.

Queues are much easier to represent using a linked chain approach than an array approach. A straightforward array solution will have either enqueue or dequeue run in time $O(n)$ due to the need to either remove, or add, at the front of the array (depending on how you lay-out the queue in the array). While this can be resolved with advanced trickery, this is not the focus of the lab.

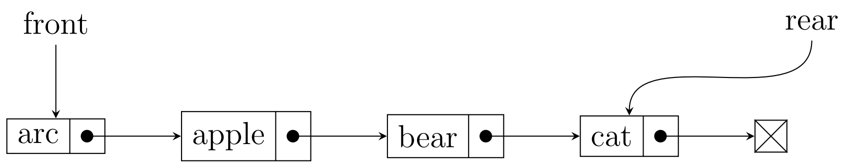
In this lab we will be storing a Queue using a linked chain approach. The approach here is nearly identical to how we represent a List using a linked chain approach. To be efficient we track both a front and end node of the linked chain. The front node will always point to the front of the queue (the place we remove data from) The rear node will always point to the last element of the queue (the place we add data to).

To understand why we structure the queue this way, remember: We can add or remove from the front of the linked chain in $O(1)$ time (we saw this in the LinkedStack example in lecture). At the rear of the chain, however, we can only add in $O(1)$ time. Removing from the rear of a linked chain structure takes $O(n)$ time. Therefore, by putting things in the order we get $O(1)$ performance for all core operations. (This perhaps is the intuitive way to do it when you think of the names, but it was worth exploring why this is also the right way to do it).

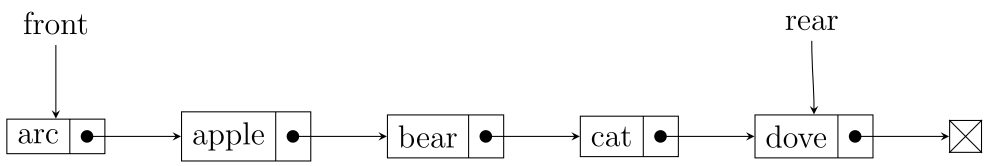
5.1 Normal Queue Example

To help make this basic representation a bit more clear, here are some visualizations. Remember – we will not be using this **exact** representation in this assignment (if we did you could write 95% of the lab with lecture code!). Instead you will be asked to make a small modification to this representation (which we will describe shortly.) That said, this general representation serves as the backbone for what we will do, so make sure what we present here makes sense before you get started.

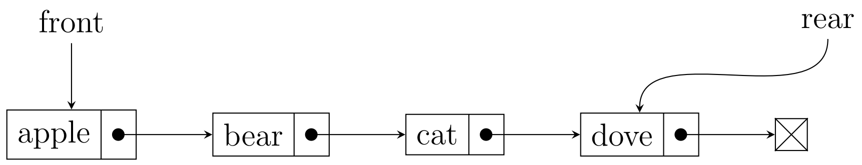
If the current state of a queue was `[arc, apple, bear, cat]` (so "arc" is at the front of the queue, followed by "apple", and "bear" with "cat" at the end of the queue) We might represent this with our linked chain (and two variables) like so:



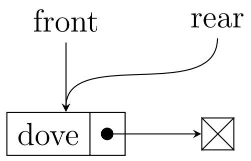
At this point, if we were to run `enqueue(dove)` the queue would conceptually be updated to `[arc, apple, bear, cat, dove]`



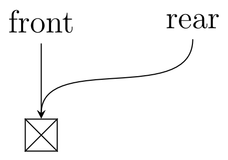
If we were to then `dequeue` we would expect "arc" to be returned, and the new state of the linked chain to be:



If we were to dequeue several more times we might have the single-element queue: `[dove]`



Note how, when there is only 1 element in the queue, front and rear point to the same element. Now, if we were to dequeue one more time, we would have an empty queue,

represented by pointing front and rear to null. 

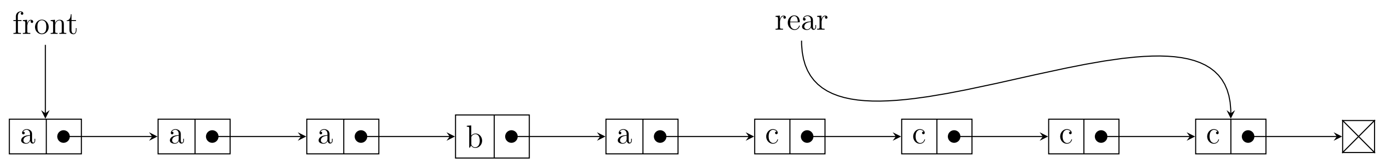
6 CacheBlock representation

Queues are often used in computers to store tasks that are waiting to be completed. This tends to happen when multiple programs are communicating, or where the computer is waiting for tasks to be completed by other devices. A prime example of this is the “print queue” – the software waiting list for documents that have been sent to the printer, but not printed. Even the fastest printer in the world cannot keep up with the speed at which computers generate documents to print.

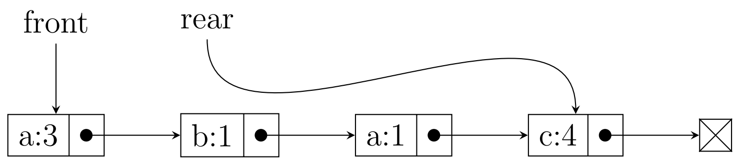
The example of a print queue highlights an interesting inefficiency with our previously described approach to a queue: what if someone needs 500 copies of the same document. Our current description of a linked chain based queue would need to store 500 nodes, each with the same document in the queue. This seems rather inefficient.

Consider the series of items enqueued into the queue. We sat that any number of items repeated is a “cache block” So if we were to enqueue: `a a a b a c c c c`, we would say that we enqueued a cache block of 3 a’s, a cache block of 1 b, a cache block of 1 a, and a cache block of 4 c’s Note that we still call one element on it’s own one cache block. When data often comes in large blocks of repeated data, then a “CacheBlock” representation can be more efficient.

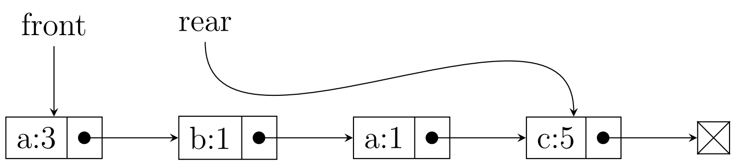
A “CacheBlock” data representation stores data with a count of repetitions, instead of storing the repeated data directly. For example, for the queue:
`a -> a -> a -> b -> a -> c -> c -> c -> c` instead of storing ten nodes:



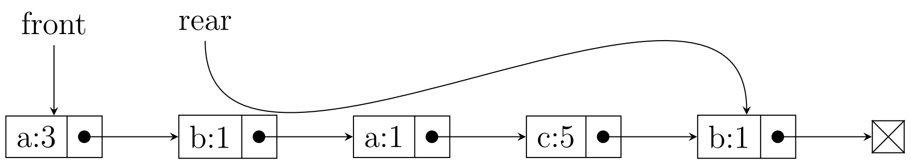
We might store 4 nodes, one for each cache block:
`a:3 -> b:1 -> a:1 -> c:4` (where `a:3` means “a repeated 3 times”)



If we were to add another c to the end, instead of making a new node we can simply increase the repeat count (`a:3 -> b:1 -> a:1 -> c:5`).



If we were to add something other than c to the end, however, we would need a new node: (adding b to the end: `a:3 -> b:1 -> a:1 -> c:5 -> b:1`).



A few further notes:

- Like before – an empty queue would be represented by two null nodes.
- When a node's count reaches 0 it should be removed from the linked chain.
- It's easy to miss this above – but notice that you can have multiple nodes for one piece of data so long as they are not part of one cache block. (so they have to be separated by at least one other piece of data.)

In an array data structure, a cache block representation often means adding another array to track the cache block counts. In a Linked chain data structure, this means modifying our node variables to track another variable: the repeat count. In either case, making this change also tends to change each interaction with our datastructure to account for the possibility that we can change the repeat count of an existing cache block of data instead of having to add/remove new node. **This will become the real challenge:** Anyone could cobble together a working queue from code we've done in lecture – but modifying the code we've already seen to account for cache blocks will require deeper understanding of how the code relates to the linked chain structure.

7 Formal Requirements

You are required to write the two following classes:

1. CacheBlockQueueNode
2. CacheBlockQueue

7.1 CacheBlockQueueNode

This class should be pretty straightforward to you. If you are having trouble you may wish to reference the LinearNode class from class notes.

The CacheBlockQueueNode must have:

- a type parameter. This represents the type of data stored in the CacheBlock Queue Node. You can call this type parameter whatever you wish, but I will refer to it with the single-letter name T in this document.
- three private variables, a T variable `data`, an int `count`, and a variable to track the `next` CacheBlockQueueNode in the linked chain.
- A constructor that takes initial values for data and next, setting count to initially be 1.
- the following set and get methods, which do what you might expect (get and set the related private variables):
  - `public T getData()`
  - `public int getCount()`
  - `public CacheBlockQueueNode<T> getNext()`
  - `public void setData(T)`
  - `public void setCount(int)`
  - `public void setNext(CacheBlockQueueNode<T>)`
- all methods on this class run in $O(1)$ time.

7.2 CacheBlockQueue

The CacheBlockQueue must have:

- every method of this class run in $O(1)$ (unless otherwise noted).
- a type parameter. This represents the type of data stored in the CacheBlock Queue. You can call this type parameter whatever you wish, but I will refer to it with the single-letter name T in this document.
- private variables. You are allowed to choose which private variables you need, however, you will find these three necessary to get $O(1)$ performance.
  - a CacheBlockQueueNode for the front of the queue
  - a CacheBlockQueueNode for the rear of the queue
  - a size variable counting the size of the data structure.
- A default(0 argument) constructor should be available. This should initialize the private variables correctly to represent an empty linked queue.
- A method `public void enqueue(T)` which adds an element to the queue.
  - You should take care here to have a special case for adding to an empty queue.
  - You should take care here to have different cases for when the new element is equal to the current end of queue, from when it is not. If the data being enqueued is

equal to the current end of queue, the count on the related CacheBlockQueueNode needs to be updated, but no new nodes are needed, otherwise a new node will be needed.

- A method `public T front()` which returns the current front of the queue without removing it. If the queue is empty return null
- A method `public T dequeue()` which removes the current front of the queue (returning the value removed). If the queue is empty return null and make no change to the private variables of the datastrucutre.
  - You will need a special case to remove the last element from the queue (creating an empty data structure)
  - You should take care here to have different cases for when the item removed was the last of a cache block or when the item removed was not the last of the cache block. If the item removed was not the last of the cache block, then the count variable in the related node needs to be updated, but no nodes should be removed from the linked chain. Otherwise, if this was the last of the cache block a node will need to be removed.
- a method `public int frontOfLineRepeatCount()` which returns the size of the cache block at the front of the queue. Another way to think about this function – it returns how many times you need to call `dequeue()` before a new value would be returned. If the queue is empty return 0.
- a method `public int getSize()` returns the size of the queue. Note this will not equal the number of nodes in the linked chain which stores the queue. Instead, this would be the sum of repeat counts in all nodes in the queue. (HINT – actually computing such a sum will not meet the runtime requirement. Think about if you can track this as you go!)
- a method `public boolean isEmpty()` which returns true if the queue is empty.
- an overridden `public String toString()`. This should run in $O(n)$ time. The formatting for this is demonstrated in the tests. You may find it useful to implement a toString method on the `CacheBlockQueueNode` class to make this easier. This is allowed, but is not required.

8 Submission

For this lab you need to turn in two files:

- `CacheBlockQueueNode.java`
- `CacheBlockQueue.java`

You can submit other files if you wish, but we do not promise to look at them during grading. Your submission must be entered into gradescope before the beginning of your next lab, although it can be up to 24 hours late for a 10% deduction.

9 Grading

This assignment will be partially manually graded, and partially automatically graded.