CMPT 115: Principles of Computer Science

Queueing Simulation

Department of Computer Science
University of Saskatchewan

March 14 - 18, 2016

Laboratory 8 Overview

Today's Goal

- To implement circular-array-based queues and run a simple simulation. (Files to get started can be found on Moodle: `Queue.h`, `Queue.cc`).

Exercises (to hand in with Assignment 8)

- Your C++ code for Lab 8: `Queue.h`, `Queue.cc`, `testQueue.cc`.

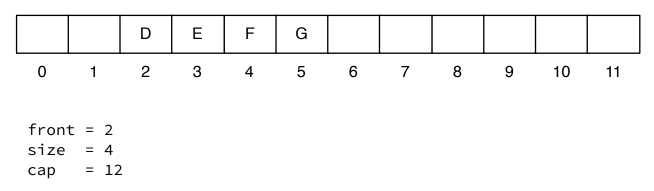
Node-based queues and simple array-based queues

- A queue is a linear organization with a FIFO behaviour.
- A node-based queue can be very efficient, with O(1) behaviour for enqueue and dequeue operations.
- Enqueue is O(1), if we know where the last data is.
- Main problem: a simple array-based dequeue operation would copy data elements towards the front of the queue, which is O(n).
- Solution: don't copy anything!

Array-based queues without copying or shuffling

- Let's keep track of three pieces of information:
  1. The capacity of the queue (never changes)
  2. The number of elements in the queue (changes as data is added or deleted)
  3. The index of the front of the queue.
- Enqueue: store the new data at index `front + size`, then increase `size`
- Dequeue: add 1 to the index stored in `front`

Array-based queues without copying or shuffling

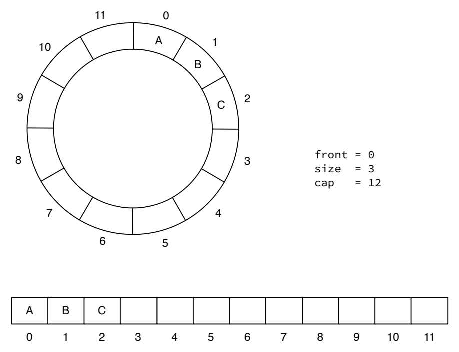


0	1	2	3	4	5	6	7	8	9	10	11
		D	E	F	G						

```
front = 2
size  = 4
cap   = 12
```

- The front of the queue is given by the index `front`.
- The end of the queue is given by `front + size`.
- As data is enqueued and dequeued, the ends of the queue drift towards the far end of the array.
- This works as long as we don't run out of room!

Circular queues

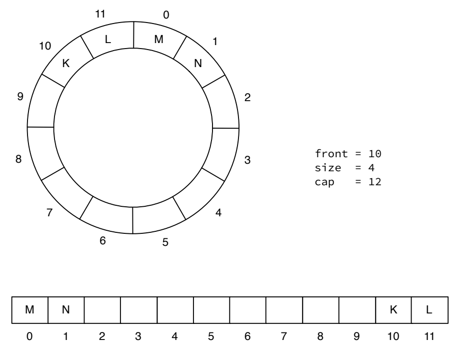


```
front = 0
size  = 3
cap   = 12
```

0	1	2	3	4	5	6	7	8	9	10	11
A	B	C									

- A circular array is a regular array, but we use it *as if* the back was joined to the front, like a circle.
- The modulus operator % is crucial for this idea!

Circular queues

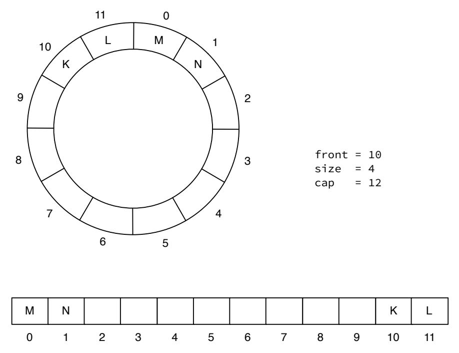


```
front = 10
size  = 4
cap   = 12
```

M	N									K	L
0	1	2	3	4	5	6	7	8	9	10	11

- A circular array is “circular” because the elements wrap around to the beginning of the array if we run out of space at the end.
- As long as the number of elements is below capacity, we can keep enqueueing and dequeueing data.

Circular queues



```
front = 10
size  = 4
cap   = 12
```

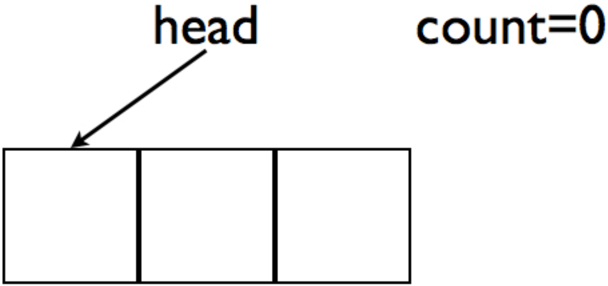
M	N									K	L
0	1	2	3	4	5	6	7	8	9	10	11

- The modulus operator % is crucial for this idea!
- In the above example, an enqueue operation would put a new element in index `(front + size) % capacity`.

Demonstration

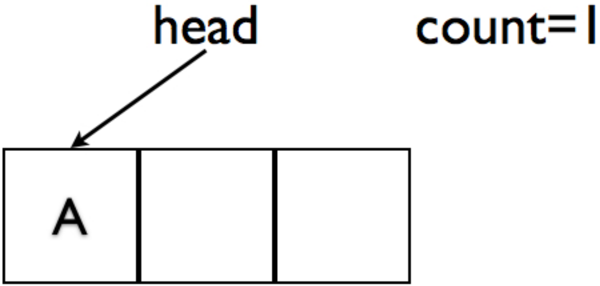
- The next few slides show an example of a queue with capacity 3. Very small to show the circularity.
- The front of the queue is called head.
- Watch the excitement in Step 7 as Element D gets enqueued in index 0, filling up the queue!

Data structure intuition - Step 1



```
CreateQueue(3)
```

Step 2



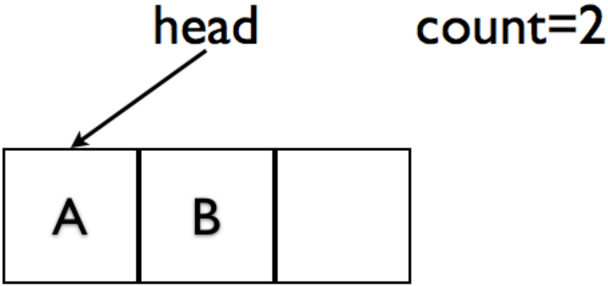
head

count=1

A

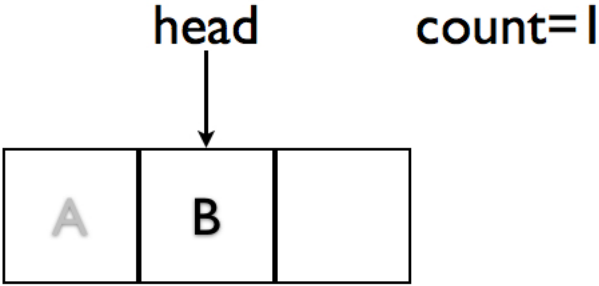
EnQueue(A)

Step 3



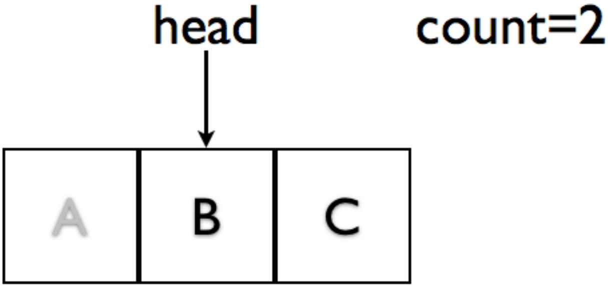
`EnQueue(B)`

Step 4



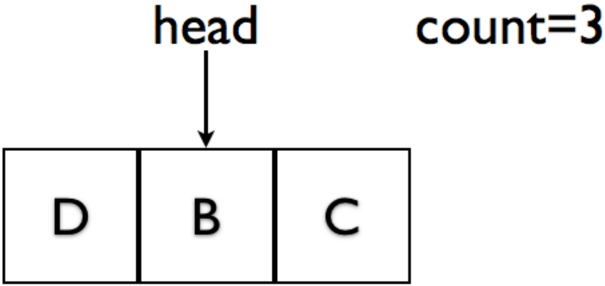
`DeQueue(x)`⇒`x=A`

Step 5



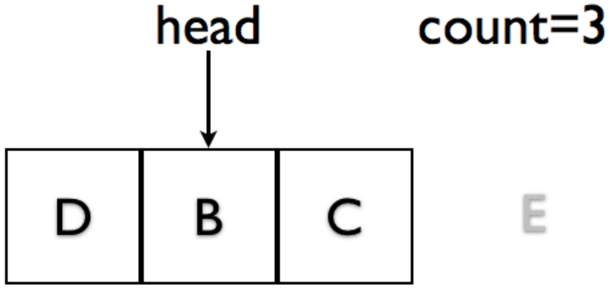
EnQueue(C)

Step 6



```
EnQueue(D)
```

Step 7



head

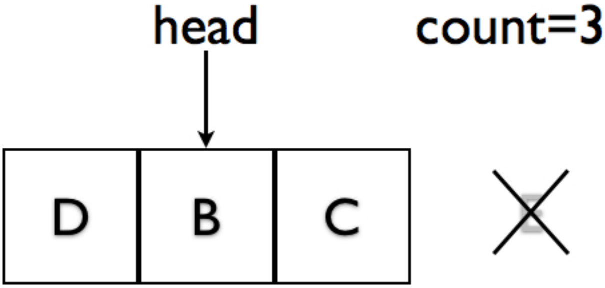
count=3

D B C

E

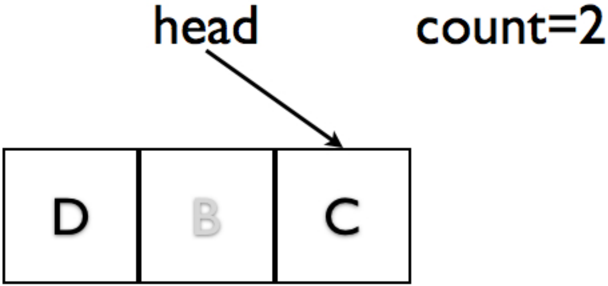
`EnQueue(E)`

Step 8



EnQueue(E)⇒false

Step 9



head

count=2

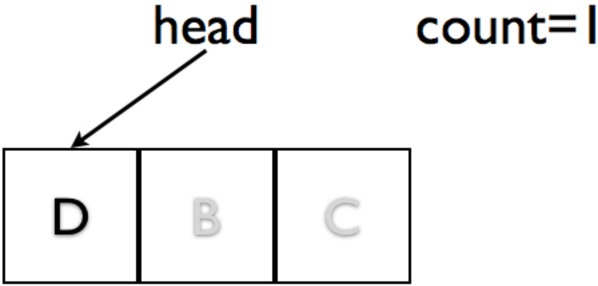
D

B

C

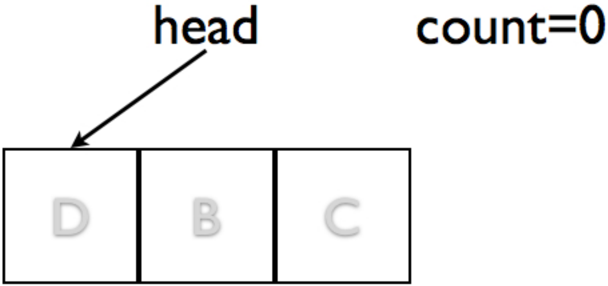
DeQueue(x)⇒x=B

Step 10



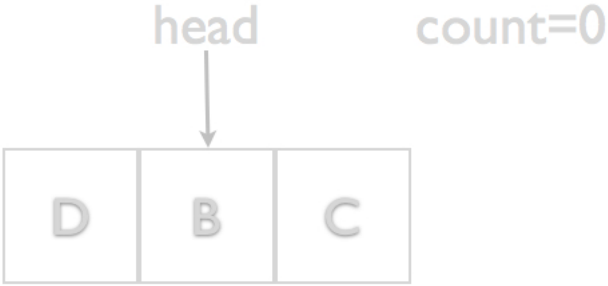
DeQueue(x)⇒x=C

Step 11



DeQueue(x)⇒x=D

Step 12



`DestroyQueue()`

Queue data Structure

What should the data structure look like?

Queue data Structure

What should the data structure look like?
No, that's a stupid question.

Queue data Structure

```
struct Queue {
  Element *elts;          // the elements
  int head;               // index of front element
  int numElements;        // number of elements in queue
  int capacity;           // how many elements can be stored
};
```

You'll find this and the operation prototypes in Queue.h

createQueue Interface

```
// Algorithm createQueue(cap)
// Pre: cap :: integer, the capacity of the new Queue
// Post: allocates space for the Queue
// Return: a reference to the new Queue
Queue *createQueue(int);
```

Activity Implement this operation in Queue.cc

destroyQueue Interface

```
// Algorithm destroyQueue()
// Post: deallocates space used by the Queue
void destroyQueue(Queue *);
```

ACTIVITY Implement this operation in Queue.cc

enqueue Interface

```
// Algorithm enqueue(q,e)
// Pre: q :: reference to a Queue
//      e :: Element
// Post: Stores e in q
// Return: true if successful,
//         false if queue is already full
bool enqueue(Queue *, Element);
```

ACTIVITY Implement this operation in Queue.cc

dequeue Interface

```
// Algorithm dequeue(q,e)
// Pre: q :: reference to Queue
//      e :: reference to Element
// Post: copies data to *e, and removes it from queue
// Return: true if successful,
//         false if queue is already empty
bool dequeue(Queue *, Element *);
```

ACTIVITY Implement this operation in Queue.cc

queueSize Interface

```
// Algorithm queueSize(q)
// Pre: q :: reference to a Queue
// Return: the number of elements in the queue
int queueSize(Queue *);
```

Activity Implement this operation in Queue.cc

queueEmpty Interface

```
// Algorithm queueEmpty(q)
// Pre: q :: reference to a Queue
// Return: true if the queue is empty, false otherwise
bool queueEmpty(Queue *);
```

ACTIVITY Implement this operation in Queue.cc

queueFull Interface

```
// Algorithm queueFull(q)
// Pre: q :: reference to a Queue
// Return: true if the queue is full, false otherwise
bool queueEmpty(Queue *);
```

ACTIVITY Implement this operation in Queue.cc

Activity: A simple queuing simulation

- The goal is to model real queue, like at a coffee shop.
- Every person “takes a number" and when a new position is available a “next number is called". The number is incremented by one every time one is taken.
- The user will control who arrives and who leaves, but your queue will make sure that the people in the queue are served in a FIFO order, i.e., fairly.

ACTIVITY: A simple queuing simulation

- Build the following program to test out the circular queue.
- Create a queue of size 10, which is the maximum size of the store/lineup.
- Continually ask the user to "enter t to take a number, c to call a number or q to quit".
  - If they enter 't', it should add the next number to the queue and display the number to the screen.
  - If they enter 'c', it should remove the last number from the queue and display it to the screen.
  - If the queue is empty, it should not be possible to leave the lineup.
  - If the queue is full, it should not be possible to take a number.

A demo of how the program should behave

```
Enter t to take a number, or c to call next number, or q to quit:
t
You have number 1.
Enter t to take a number, or c to call next number, or q to quit:
t
You have number 2.
Enter t to take a number, or c to call next number, or q to quit:
c
The number 1 should leave the line
Enter t to take a number, or c to call next number, or q to quit:
t
You have number 3.
Enter t to take a number, or c to call next number, or q to quit:
c
The number 2 should leave the line
Enter t to take a number, or c to call next number, or q to quit:
q
```

Activity Implement this program in queueTest.cc