Computer Science 331
Queues

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #11

Outline

1. Definition
2. Applications
3. Implementations
   - Array-Based Implementation (Circular Queues)
   - List-Based Implementation
4. Generalizations
   - Double Ended Queues
   - Priority Queues
5. Queues in Java

The Queue ADT

A **queue** is a collection of objects that can be accessed in “first-in, first-out” order: The only element that is visible and that can be removed is the oldest remaining element.

Attributes:

- *size* : The number of elements on the queue; $size \geq 0$ at all times.
- *front* : The first element of the queue. This refers to `null`, a special value, if the queue is empty (that is, if $size = 0$)
- *rear:* The position in the queue where the next element is to be inserted, or a `null` value when the queue is empty.

Definition of the Queue ADT (cont.)

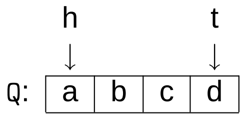
Operations: (Java interface names: “offer,” “remove,” “poll”)

- `Queue()`: Constructor; creates an empty queue
- `enqueue(T element)`: Inserts an element at the *rear* of the queue
- `dequeue()`: Removes and returns the element at the *front*
- `peek()`: Returns the element at the *front* of the queue without removing it (leaving the queue unchanged)
- `size()`: Returns the number of elements on the queue
- `isEmpty()`: Reports whether the queue is empty

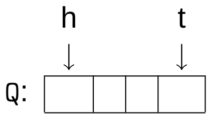
Note: Operations `dequeue` and `peek` each have the **pre-condition** that the queue is nonempty and thrown an *NoSuchElementException* exception if this condition is not satisfied when they are called.

Implementation Using an Array

Initial Queue



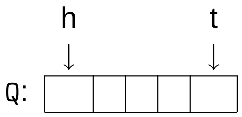
Effect of `Q.peek()`



Output:

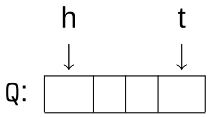
Implementation Using an Array

Effect of `Q.enqueue(e)`



Output:

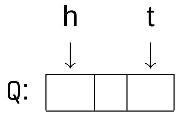
Effect of `Q.dequeue()`



Output:

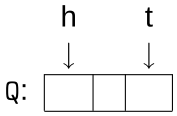
Implementation Using an Array

Effect of `Q.dequeue()`



Output:

Effect of `Q.peek()`



Output:

Variation: Bounded Queues

These queues are created to have a maximum *capacity* (possibly user-defined — so that two constructors are needed)

- If the capacity would be exceeded when a new element is enqueued then an `enqueue` operation throws a *FullQueueException* exception and leaves the queue unchanged
- Additional operations included a `capacity()` operation that returns the capacity of the queue as well as an `isFull()` test

Types of Applications

Scheduling:

- Examples: *Print Queues* and *File Servers* — In each case requests are served on a first-come first-served basis, so that a queue can be used to store the requests

Simulation:

- Example: *Modelling traffic* in order to determine optimal traffic lighting (to maximize car throughput)
- *Discrete Event Simulation* is used to provide empirical estimates
- Queues are used to store information about simulated cars waiting at an intersection

Checking for Palindromes

Palindrome: Word or phrase whose letters are the same backwards as forwards.

Examples:

Madam, I'm Adam.
Delia saw I was ailed.

See `http://www.palindromelist.com` for lots of examples.

Exercise: Design an algorithm that uses both a stack *and* a queue to decide whether a string is a palindrome in linear time.

Straightforward Array-Based Representation

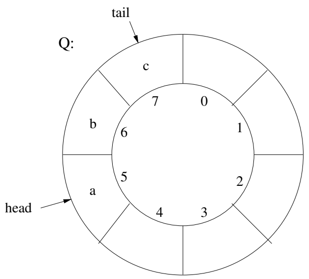
Doesn't work well! Problems:

- If we try to keep the *head* element at position 0 then we must shift the entire contents of the array over, every time there is a `dequeue` operation
- On the other hand, if we try to keep the *rear* element at position 0 then we must shift the entire contents of the array over, every time there is an `enqueue` operation

Operations are too expensive, either way!

A "Circular" Array

Solution: Allow *both* the position of the head and rear element to move around, as needed.

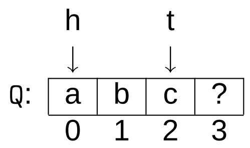


		t				h		
		↓				↓		
Q:	d	e	?	?	?	a	b	c
	0	1	2	3	4	5	6	7

`head=5, tail=1, size=5`

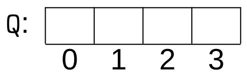
Example with Queue Operations

Initial Queue


h
t
Q: a b c ?
0 1 2 3

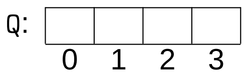
head = 0
tail = 2
size = 3

Q.enqueue(d)


Q:
0 1 2 3

head =
tail =
size =

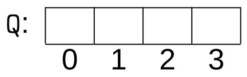
Q.dequeue()


Q:
0 1 2 3

head =
tail =
size =

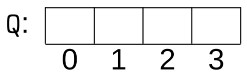
Example with Queue Operations (cont.)

Q.enqueue(e)


Q:
0 1 2 3

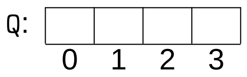
head =
tail =
size =

Q.dequeue()


Q:
0 1 2 3

head =
tail =
size =

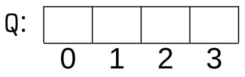
Q.dequeue()


Q:
0 1 2 3

head =
tail =
size =

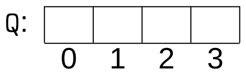
Example with Queue Operations (cont.)

Q.dequeue()


Q:
0 1 2 3

head =
tail =
size =

Q.dequeue()


Q:
0 1 2 3

head =
tail =
size =

Implementation of Queue Operations

```
public class CircularArrayQueue<T> {
  private T[] queue;
  private int head;
  private int tail;
  private int size;

  public CircularArrayQueue()
    {

  public boolean isEmpty()
    {

  public T peek() {
    if (isEmpty())  throw new NoSuchElementException;
    return queue[head];
  }
```

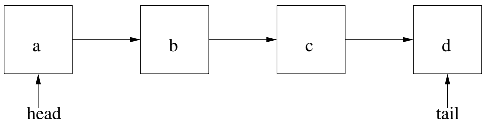
Implementation of Queue Operations (cont.)

```
public T dequeue() {
  if (isEmpty())  throw new NoSuchElementException;
  T x = queue[head];

  return x;
}
public enqueue(T x) {
  if () {
    T [] queueNew = (T[]) new Object[2*queue.length];
    for (int i=0; i<queue.length-1; ++i)
      queueNew[i] = queue[(head+i) % queue.length];
    head = 0;  tail = queue.length-1;  queue = queueNew;
  }
  else

  queue[tail] = x; ++size;
}
```

Implementation Using a Linked List

Singly-linked list representation:

- head points to first element, tail points to last element



Operations:

- dequeue: delete first element of list
- enqueue(x): insert at tail of list

Why not have the tail point to the first element and the head point to the last?

Implementation Using a Linked List, Example

Effect of dequeue()

Pseudocode:

-

Cost:

Effect of enqueue(x)

Pseudocode:

-
-
-

Implementation of Queue Operations

```
public class LinkedListQueue<T> {
  private class QueueNode<T> { similar to StackNode }

  private QueueNode<T> head, tail;
  private int size;

  public LinkedListQueue() {
    {

  public boolean isEmpty()  {

  public T peek() {
    if (isEmpty())  throw new NoSuchElementException();
    return head.value;
  }
```

Implementation of Queue Operations (cont.)

```
public void enqueue(T x) {
  QueueNode<T> newNode = new QueueNode<T>(x,null);
  if (isEmpty())

  else

  tail = newNode;  ++size;
}

public T dequeue() {
  if (isEmpty())  throw new NoSuchElementException();
  T x = head.value; head = head.next;
  if (head == null)

  --size; return x;
}
```

Comparison of Array and List-Based Implementations

Array-based:

- all operations almost always $\Theta(1)$
- `enqueue` is $\Theta(n)$ in the worst case (resizing the array)
- good for bounded queues (and stacks) where worst case doesn't occur

List-based:

- all operations $\Theta(1)$ in worst case
- extra storage requirement (one reference per item)
- good for large queues (and stacks) without a good upper bound on size (resizing is expensive)

Choice of implementation to use depends on the application.

Double Ended Queue — "Dequeue"

A "double ended queue (dequeue)" allows both operations on both ends:

Operations:

- `addFront(x)`: Insert item `x` onto front
- `removeFront()`: Remove and report value of front item
- `addRear(x)`: Append item `x` onto back
- `removeRear()`: Remove and report value of rear item

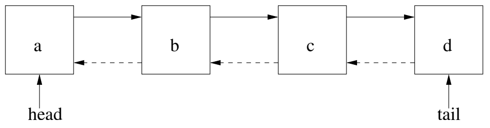
Operations `removeFront` and `removeRear` should throw exceptions if called when the dequeue is empty.

Implementations

Circular array implementation — similar to that of a regular queue.

- `addFront`, `addRear` cost $\Theta(n)$ in worst-case (due to resizing the array), $\Theta(1)$ otherwise
- all other operations $\Theta(1)$

A *doubly-linked list* can also be used:



- All operations in time $\Theta(1)$ (exercise)
- Without a `previous` pointer, removeRear is $\Theta(n)$

Priority Queues

A **priority queue** associates a *priority* as well as a *value* with each element that is inserted.

The *element with smallest priority* is removed, instead of the oldest element, when an element is to be deleted.

Priority Queues will be considered again we discuss algorithms for **sorting**.

Also applicable for **data compression** (eg. Huffman encoding).

Queues in Java

Java Collections Framework:

- includes a more general “Queue” interface and numerous classes that implement this
- **Warning:** The term “queue” is used in Java is used to describe a *much* larger set of structures than is standard.

Queues in the Textbook:

- Chapter 7 of the textbook includes additional details along with two implementations — one that is an adaption of a `List` and another that is an array-based implementation, built “from scratch”