Computer Science 331

Queues[1]

As part of the SAGES Teaching Scholar Program

Parthasarathi Das

Department of Computer Science
University of Calgary

Lecture #14

[1] Adapted from Dr. Michael Jacobson's lecture slides.

Outline

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

Learning Outcomes

By the end of today's session, you will be able to -

- understand what queues are, their various types and some applications of queues.
- implement queues using arrays and linked lists
- apply this ADT suitably to solve problems

Introduction to Queues

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

A Queue ADT

Queue Interface:

```
public interface Queue<T> {
  public boolean add(T x);
  public T remove();
  public T element();
  public boolean isEmpty();
}
```

Queue Invariant:

- The object that is visible (and that would be removed next) is the oldest object that remains in the queue.

A Queue ADT: Methods

1. `boolean add (T e):`
   - *Precondition:*
     a) Class Invariant.
   - *Postcondition:*
     a) The item `e` is added at the rear of the queue.
     b) Value returned is `true`.

A Queue ADT: Methods

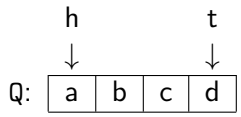
2. `T remove():`
   - *Precondition:*
     a) Class Invariant.
     b) Queue is nonempty.
   - *Postcondition:*
     a) Item at the front of the queue is removed.
     b) The removed value is returned as output.
   - *Exception:* A `NoSuchElementException` is thrown if the queue is empty when this method is called

A Queue ADT: Methods

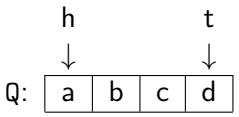
3. `T element():`
   - *Precondition:*
     a) Class Invariant
     b) Queue is nonempty
   - *Postcondition:*
     a) Queue is unchanged
     b) The element at the front of the queue is returned as output
   - *Exception:* A `NoSuchElementException` is thrown if the queue is empty when this method is called

Implementation Using an Array

Initial Queue



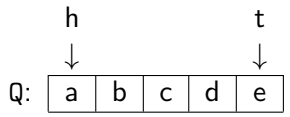
Effect of `Q.element()`



Output: a

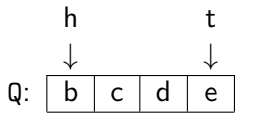
Implementation Using an Array

Effect of `Q.add(e)`



Output: no output

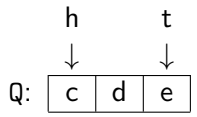
Effect of `Q.remove()`



Output: a

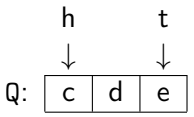
Implementation Using an Array

Effect of `Q.remove()`



Output: b

Effect of `Q.element()`



Output: c

Variation: Bounded Queues

These queues are created to have a maximum *capacity* (possibly user-defined — in which case, two constructors are needed).

Like bounded stacks, bounded queues can be implemented more simply (and efficiently) than their unbounded counterparts.

If a bounded queue is already full, and either `add` or `offer` is called, then the queue is not changed. The failure to add another item is indicated differently in each case:

- The method "`add`" throws an `IllegalStateException`.
- The method "`offer`" returns the value `false` instead of `true`.

"Six" Operations, Reconsidered

In addition to the queue operations described above, there are three more which perform the same functions but handle error reporting differently.

1. Throwing an exception
   a) `add`: Insertion of new element at rear
   b) `remove`: Removal of front element
   c) `element`: Report front element without removal
2. Unusual output (`false` or `null`)
   a) `offer`: Insertion of new element at rear
   b) `poll`: Removal of front element
   c) `peek`: Report front element without removal

At this point one can see that the six methods provide three different operations, using two approaches to report error conditions:

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:

- *Modelling traffic* in order to determine optimal traffic lighting (to maximize car throughput). Queues are used to store information about simulated cars waiting at an intersection. Driverless cars?

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

| A Santa dog lived as a devil God at NASA | Malayalam |

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

Straightforward Array-Based Representation

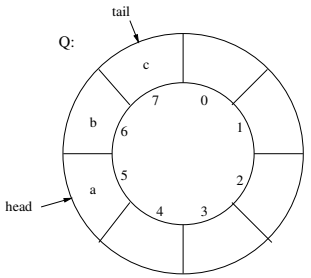
Doesn't work well! Problems:

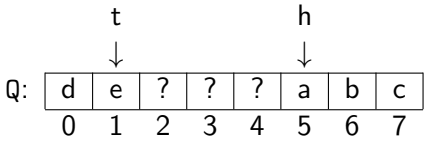
- If we try to keep the *head* element at position 0 then we must shift the entire contents of the array over, every time there is a `remove` operation
- On the other hand, if we try to keep the *rear* element at position 0 then we must shift the entire contents of the array over, every time there is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

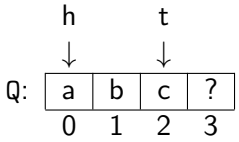




`head=5, tail=1, size=5`

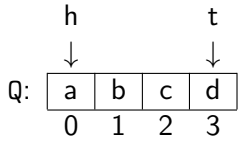
Example with Queue Operations

Initial Queue



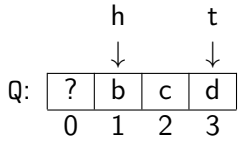

```
head = 0
tail = 2
size = 3
```

Q.add(d)




```
head = 0
tail = 3
size = 4
```

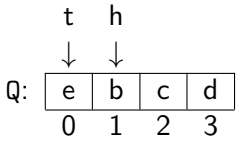
Q.remove()




```
head = 1
tail = 3
size = 3
```

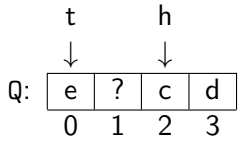

Example with Queue Operations (cont.)

Q.add(e)



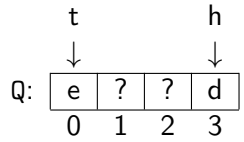

```
head = 1
tail = 0
size = 4
```

Q.remove()




```
head = 2
tail = 0
size = 3
```

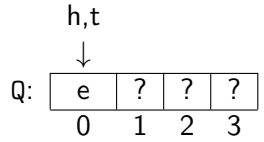
Q.remove()




```
head = 3
tail = 0
size = 2
```

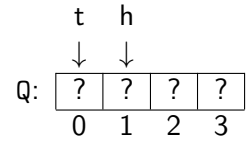

Example with Queue Operations (cont.)

Q.remove()




```
head = 0
tail = 0
size = 1
```

Q.remove()




```
head = 1
tail = 0
size = 0
```


Implementation of Queue Operations

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

  public CircularArrayQueue()
    { tail= -1; head = size = 0; queue = (T[]) new Object[8]; }

  public boolean isEmpty()
    { return (size == 0); }

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

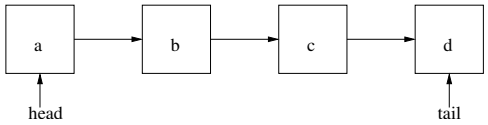
Implementation of Queue Operations (cont.)

```
public T remove() {
  if (isEmpty())  throw new NoSuchElementException;
  T x = queue[head];   queue[head] = null;
  head = (head+1) % queue.length;  --size;
  return x;
}
public add(T x) {
  if (size == queue.length) {
    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
    tail = (tail + 1) % queue.length;
  queue[tail] = x; ++size;
}
```


Implementation Using a Linked List

Singly-linked list representation:

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



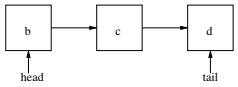
Operations:

- `remove`: delete first element of list
- `add(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

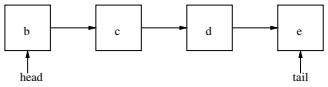
Effect of `remove()`



Pseudocode:

- `head = head.next;`

Effect of `add(x)`



Pseudocode:

- create new list node
- `tail.next = new;`
- `tail = new;`

Cost: $\Theta(1)$ (independent of queue size)

Implementation of Queue Operations

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

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

  public LinkedListQueue() {
    { size = 0; head = tail = (QueueNode<T>) null; }

  public boolean isEmpty()  { return (head == null); }

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

Implementation of Queue Operations (cont.)

```
public void add(T x) {
  QueueNode<T> newNode = new QueueNode<T>(x,null);
  if (isEmpty())
    head = newNode;
  else
    tail.next = newNode;
  tail = newNode;  ++size;
}

public T remove() {
  if (isEmpty())  throw new NoSuchElementException();
  T x = head.value; head = head.next;
  if (head == null)
    tail == null;
  --size; return x;
}
```


Comparison of Array and List-Based Implementations

Array-based:

- all operations almost always $\Theta(1)$ (amortized cost)
- add 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)

Double Ended Queue — "dequeue"

A "double ended queue (dequeue or deque, pronounced *deck*)" allows both operations on both ends:

Operations:

- `addFirst(x)`: Insert item `x` onto front
- `addLast(x)`: Append item `x` onto back
- `isEmpty()`: Return `True` if the deque is empty
- `removeFirst()`: Remove and report value of front item
- `removeLast()`: Remove and report value of rear item
- `getFirst()`: Report value of front item
- `getLast()`: Report value of rear item

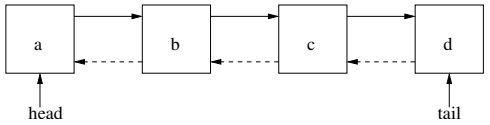
Operations `removeFirst` and `removeLast` should throw exceptions and `getFirst()` and `getLast()` should return `Null` if called when the dequeue is empty.

Implementations

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

- `addFirst`, `addLast` 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, removeLast is $\Theta(n)$

Applications: ?

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 when we discuss

- algorithms for sorting
- graph algorithms

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

A Complication

Complication: There are multiple data types that resemble the "simple queue" that are described in these notes but that also differ from it in significant ways.

The Java Collections Framework does include a `Queue<E>` interface — but this is implemented (potentially, somewhat confusingly) by classes providing several of the ADTs described here!

A Complication

Solution, for our Purposes: Java's `LinkedList<E>` class implements the `Queue<E>` interface and provides a "simple queue" when it does so.

The statement

```
Queue<String> names = new LinkedList<String>();
```

creates a new `Queue` reference, "`names`," that stores information to `String` objects. While the actual object referenced by `names` is of type `LinkedList<String>`, only the `Queue` methods can be applied to it (because, again, `names` is a `Queue` reference).

What This Provides: A way to use the Java Collections Framework to obtain an efficient and reliable implementation of a "simple queue"

Queues in the Textbook

Introduction to Algorithms

- by Cormen, Lieserson, Rivest, and Stein
- Section 10.1

Data Structures: Abstraction and Design Using Java

- by Elliot B. Koffman and Paul A. T. Wolfgang
- Chapter 4