CS121 Data Structures A, C
Queues

Varduhi Yeghiazaryan
vyeghiazaryan@aua.am

Fall 2021

Queues

A **queue** is a collection of objects that are inserted and removed according to the **first-in, first-out (FIFO)** principle

A user can only access or remove the element that has been in the queue the longest, at the ‘front’ of the queue

We say that elements enter a queue at the back and are removed from the front

Queues are a simplest and fundamental data structure. They are among the most important, as they are used in many applications

The Queue Abstract Data Type

Formally, a queue is an ADT that supports the following methods:

enqueue(e):	Adds element e to the back of the queue
dequeue():	Removes and returns the first element from the queue (or `null` if the queue is empty)
first():	Returns the first element of the queue, without removing it (or `null` if the queue is empty)
size():	Returns the number of elements in the queue
isEmpty():	Returns a boolean indicating whether the queue is empty

By convention, elements added to the queue can have arbitrary type and a newly created queue is empty

Example

Method	**Return Value**	first $\leftarrow Q \leftarrow$ last
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
size( )	2	(5, 3)
dequeue( )	5	(3)
isEmpty( )	false	(3)
dequeue( )	3	( )
isEmpty( )	true	( )
dequeue( )	null	( )
enqueue(7)	–	(7)
enqueue(9)	–	(7, 9)
first( )	7	(7, 9)
enqueue(4)	–	(7, 9, 4)

The Queue Application Programming Interface (API)

In Java, we define an **interface** corresponding to our Queue ADT

```java
public interface Queue<E> {
  /** Returns the number of elements in the queue. */
  int size();
  /** Tests whether the queue is empty. */
  boolean isEmpty();
  /** Inserts an element at the rear of the queue. */
  void enqueue(E e);
  /** Returns, but does not remove, the first element of the queue (null if empty). */
  E first();
  /** Removes and returns the first element of the queue (null if empty). */
  E dequeue();
}
```

The `java.util.Queue` Interface in Java

Java provides a type of queue interface, `java.util.Queue`, which has functionality similar to the traditional queue ADT

But the documentation for the `java.util.Queue` interface does not insist that it support only the FIFO principle

Our Queue ADT	**Interface java.util.Queue**	
	throws exceptions	returns special value
enqueue(e)	add(e)	offer(e)
dequeue()	remove()	poll()
first()	element()	peek()
size()	size()	
isEmpty()	isEmpty()	

In C++, the Standard Template Library (STL) provides an implementation of a queue

Array-Based Queue

A simple way of implementing the Queue ADT uses an array

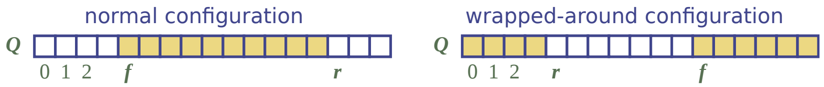
We use an array of size N in a *circular way*, i.e. contents of the queue “wrap around” the end of the array

Two variables keep track of the front and size

data:	a reference to the underlying array
f:	index of the front element
sz:	number of stored elements

We use the modulo operator (remainder of division)

The array storing the queue elements may become full



Array-Based Queue Implementation

```java
/** Implementation of the queue ADT using a fixed−length array. */
public class ArrayQueue<E> implements Queue<E> {
  public static final int CAPACITY = 1000;  // default array capacity

  // instance variables
  private E[] data;                        // generic array used for storage
  private int f = 0;                       // index of the front element
  private int sz = 0;                      // current number of elements

  // constructors
  public ArrayQueue() {this(CAPACITY);} // constructs queue with default capacity
  public ArrayQueue(int capacity) {     // constructs queue with given capacity
    data = (E[]) new Object[capacity];  // safe cast; compiler may give warning
  }

  // methods
  /** Returns the number of elements in the queue. */
  public int size() { return sz; }

  /** Tests whether the queue is empty. */
  public boolean isEmpty() { return (sz == 0); }
```

Array-Based Queue Implementation (cont'd)

```

  /** Inserts an element at the rear of the queue. */
  public void enqueue(E e) throws IllegalStateException {
    if (sz == data.length) throw new IllegalStateException("Queue is full");
    int avail = (f + sz) % data.length;     // use modular arithmetic
    data[avail] = e;
    sz++;
  }

  /** Returns, but does not remove, the first element of the queue (null if empty). */
  public E first() {
    if (isEmpty()) return null;
    return data[f];
  }

  /** Removes and returns the first element of the queue (null if empty). */
  public E dequeue() {
    if (isEmpty()) return null;
    E answer = data[f];
    data[f] = null;                         // dereference to help garbage collection
    f = (f + 1) % data.length;
    sz--;
    return answer;
  }
}
```

Array-Based Queue: Analysis

Drawback: fixed-capacity array, limiting the ultimate queue size

If the application needs much less space than the reserved capacity, memory is wasted

If we try to enqueue an element into a full queue, the implementation throws an exception and refuses to store the new element

Analysis: each method executes a constant number of statements involving arithmetic operations, comparisons, and assignments, or calls to `size` and `isEmpty`, which both run in constant time

Method	**Running Time**
size	$O(1)$
isEmpty	$O(1)$
first	$O(1)$
enqueue	$O(1)$
dequeue	$O(1)$

Space usage: $O(N)$, where N is the size of the array, independent from the number $n \leq N$ of elements in the queue

Linked-List-Based Queue

A second simple way of implementing the Queue ADT uses a singly or circularly linked list

We add elements (to the rear of the queue) at the back of the list

Thus all methods execute in constant time

The linked-list approach has memory usage proportional to the number of actual elements currently in the queue

No arbitrary capacity limits

Using the **adapter** design pattern, we modify an existing class so that its methods match those of a related, but different, class or interface

Linked-List-Based Queue Implementation

```
/** Realization of a FIFO queue as an adaptation of a SinglyLinkedList. */
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list = new SinglyLinkedList<>();    // an empty  list
  public LinkedQueue() { }                      // new queue relies on the initially empty list
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void enqueue(E element) { list.addLast(element); }
  public E first() { return list.first(); }
  public E dequeue() { return list.removeFirst(); }
}
```

Each method of our `LinkedQueue` adaptation also runs in $O(1)$ worst-case time

In practice, the linked-list methods are more expensive than the array-based methods

A Circular Queue

Reminder: a circularly linked list class supports all behaviours of a singly linked list, and an additional `rotate()` method that efficiently moves the first element to the end of the list

We can generalize the `Queue` interface to define a new `CircularQueue` interface:

```
public interface CircularQueue<E> extends Queue<E> {
  /**
   * Rotates the front element of the queue to the back of the queue.
   * This does nothing if the queue is empty.
   */
  void rotate();
}
```

It is easily implemented by adapting the `CircularlyLinkedList` class to produce a new `LinkedCircularQueue` class

Note that a call to `Q.rotate()` is implemented more efficiently than the combination of calls `Q.enqueue(Q.dequeue())`

Double-Ended Queues

A **double-ended queue**, or **deque**, is a queue-like data structure that supports insertion and deletion at both the front and the back of the queue

The deque abstract data type is more general than both the stack and the queue ADTs

This is useful in some applications

The Deque Abstract Data Type

addFirst(e):	Inserts element e at the front
addLast(e):	Inserts element e at the back
removeFirst():	Removes and returns the first element (or `null` if the deque is empty)
removeLast():	Removes and returns the last element (or `null` if the deque is empty)
first():	Returns the first element, without removing it (or `null` if the deque is empty)
last():	Returns the last element, without removing it (or `null` if the deque is empty)
size():	Returns the number of elements in the deque
isEmpty():	Returns a boolean indicating whether the deque is empty

Example

Method	**Return Value**	*D*
addLast(5)	–	(5)
addFirst(3)	–	(3, 5)
addFirst(7)	–	(7, 3, 5)
first( )	7	(7, 3, 5)
removeLast( )	5	(7, 3)
size( )	2	(7, 3)
removeLast( )	3	(7)
removeFirst( )	7	( )
addFirst(6)	–	(6)
last( )	6	(6)
addFirst(8)	–	(8, 6)
isEmpty( )	false	(8, 6)
last( )	6	(8, 6)

The Deque Application Programming Interface (API)

In Java, we define an **interface** corresponding to our Deque ADT

```
/**
 * Interface for a double-ended queue: a collection of elements that can be inserted
 * and removed at both ends; this interface is a simplified version of java.util.Deque.
 */
public interface Deque<E> {
  /** Returns the number of elements in the deque. */
  int size();
  /** Tests whether the deque is empty. */
  boolean isEmpty();
  /** Returns, but does not remove, the first element of the deque (null if empty). */
  E first();
  /** Returns, but does not remove, the last element of the deque (null if empty). */
  E last();
  /** Inserts an element at the front of the deque. */
  void addFirst(E e);
  /** Inserts an element at the back of the deque. */
  void addLast(E e);
  /** Removes and returns the first element of the deque (null if empty). */
  E removeFirst();
  /** Removes and returns the last element of the deque (null if empty). */
  E removeLast();
}
```

Deque Implementations

The deque ADT can be efficiently implemented using either an array or a linked list for storing elements

If using an array, it can be treated in circular fashion

If using a linked list, a doubly linked list is most appropriate

Every method from the Deque ADT will have $O(1)$ running time

Summary

Reading

Section 6.2 Queues

Section 6.3 Double-Ended Queues

Questions?