QUEUES (6.2)

EECS 2011

Queues: FIFO

- Insertions and removals follow the Fist-In First-Out rule:
  - Insertions: at the rear of the queue
  - Removals: at the front of the queue

- Applications, examples:
  - Waiting lists
  - Access to shared resources (e.g., printer)
  - Multiprogramming (UNIX)

Queue ADT

- Data stored: arbitrary objects
- Operations:
  - ***enqueue***(object): inserts an element at the end of the queue
  - object ***dequeue***(): removes and returns the element at the front of the queue
  - object ***first***(): returns the element at the front without removing it
- Execution of ***dequeue***() or ***first***() on an empty queue

  → throws *EmptyQueueException*
- Another useful operation:
  - **boolean *isEmpty***(): returns true if the queue is empty; false otherwise.

Queue Operations

- ***enqueue***(object)
- object ***dequeue***()
- object ***first***()
- **boolean *isEmpty***()
- **int *size***(): returns the number of elements in the queue

- Any others? Depending on implementation and/or applications

```
public interface Queue {
public int size();
public boolean isEmpty();
public Object first()
  throws
  EmptyQueueException;
public Object dequeue()
  throws
  EmptyQueueException;
public void enqueue (Object
  obj);
}
```

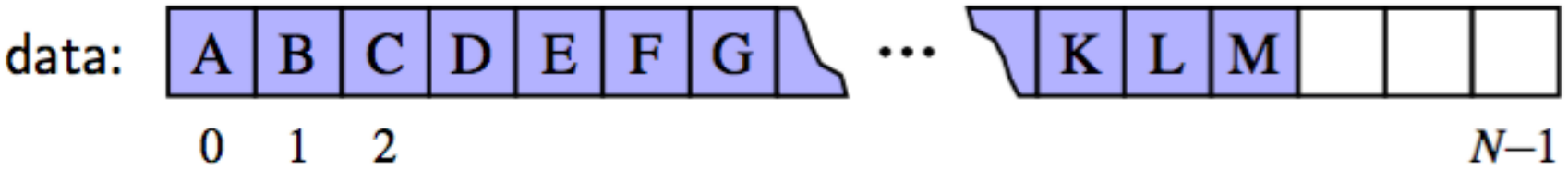
Queue Example

Operation	*Output*	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
first()	*7*	(7)
dequeue()	*7*	()
dequeue()	*"error"*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)

Array-based Implementation

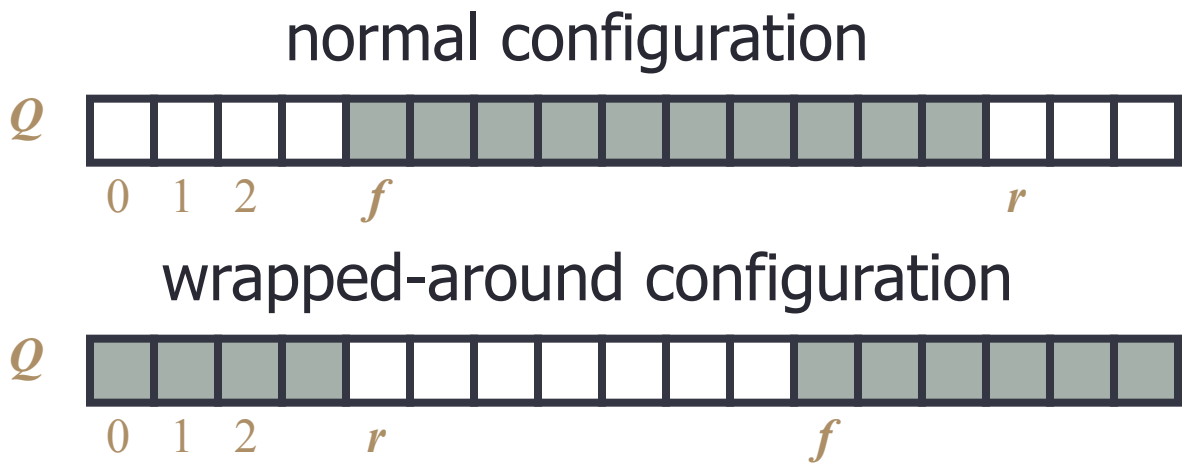
- An array Q of maximum size N
- We need to decide where the front and rear are.
- How to enqueue, dequeue?
- Running time of enqueue?
- Running time of dequeue?

Enqueue and Dequeue



Array-based Queue

- Use an array of size N in a circular fashion
- Two variables keep track of the front and size
  - f index of the front element
  - sz number of stored elements
- When the queue has fewer than N elements, array location $r = (f + sz) \bmod N$ is the first empty slot past the rear of the queue

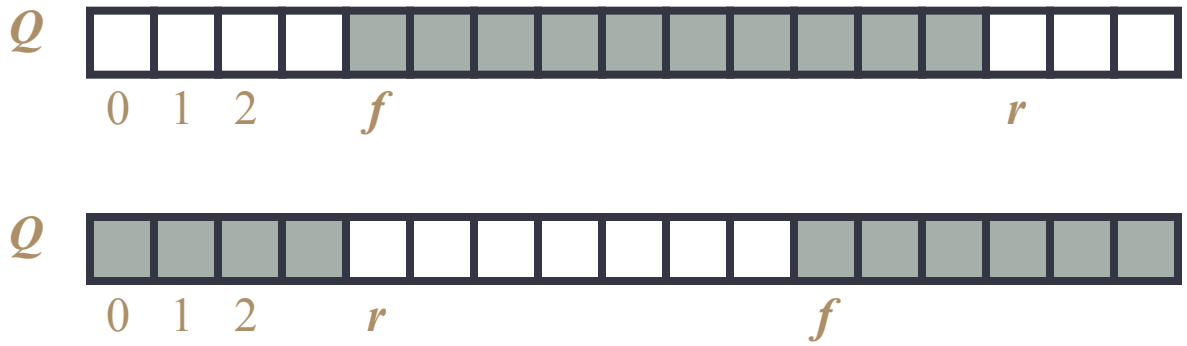


Queue Operations

- We use the modulo operator (remainder of division)

Algorithm *size*()
  return *sz*

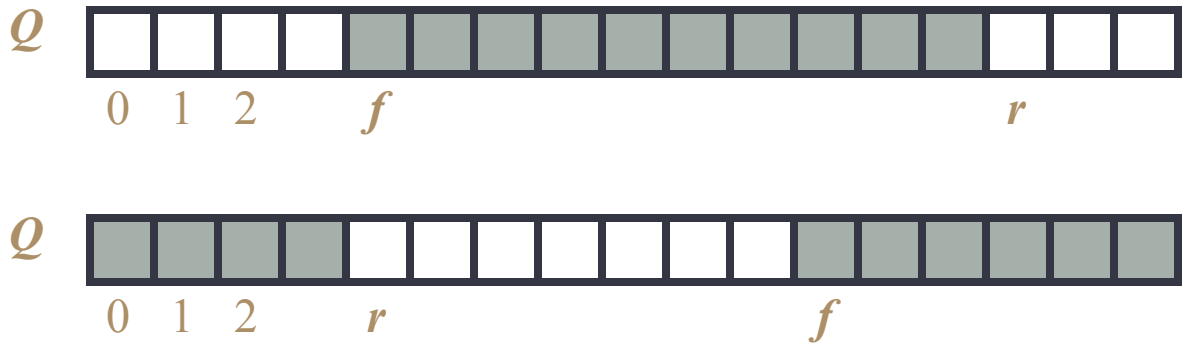
Algorithm *isEmpty*()
  return (*sz* == 0)



Queue Operations: *enqueue*

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

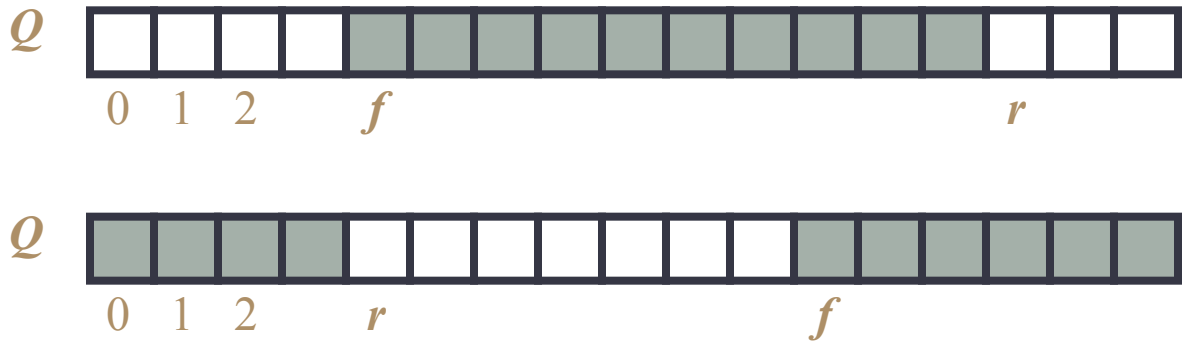
Algorithm *enqueue*(e)
  if *size*$() = N - 1$ **then**
    throw *IllegalStateException*
  else
    $r \leftarrow (f + sz) \bmod N$
    $Q[r] \leftarrow e$
    $sz \leftarrow (sz + 1)$



Queue Operations: *dequeue*

- Note that operation dequeue returns null if the queue is empty

```
Algorithm dequeue()
  if isEmpty() then
    return null
  else
    e ← Q[f]
    f ← (f + 1) mod N
    sz ← (sz − 1)
    return e
```



Analysis of Circular Array Implementation

Performance

- Each operation runs in $O(1)$ time

Limitation

- The maximum size N of the queue is fixed
- How to determine N?
- Alternatives?
  - Extendable arrays
  - Linked lists (singly or doubly linked?)

Singly or Doubly Linked?

- Singly linked list

```
public static class Node
   {
     private Object data;
     private Node  next;
   }
```

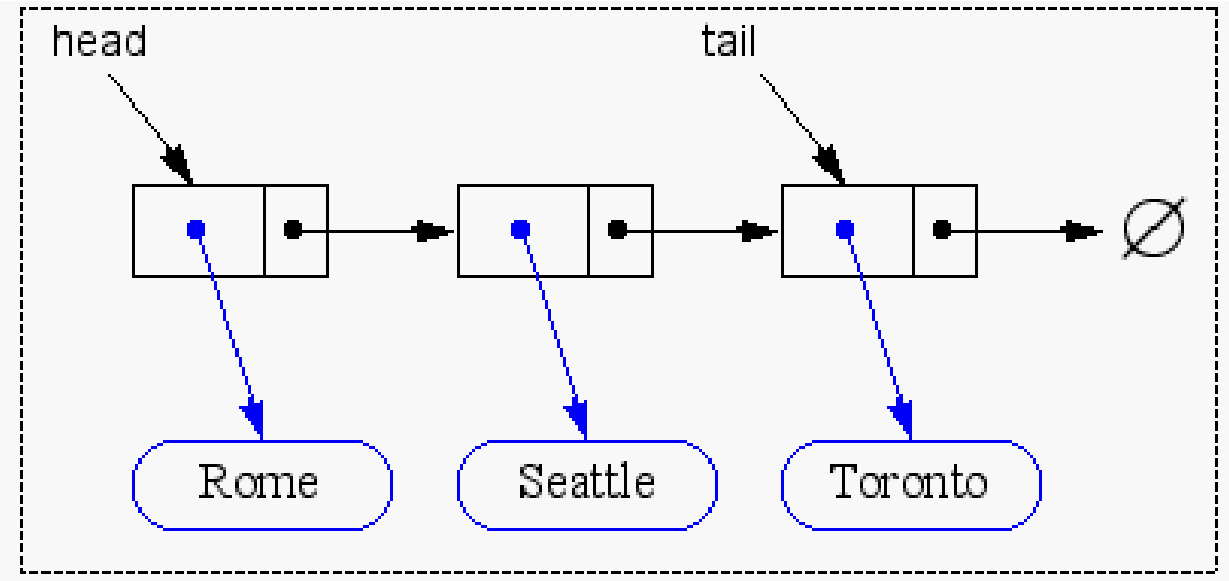
- Needs less space.
- Simpler code in some cases.
- Insertion at tail takes O(n).

- Doubly linked list

```
public static class DNode
   {
     private Object data;
     private Node  prev;
     private Node next;
   }
```

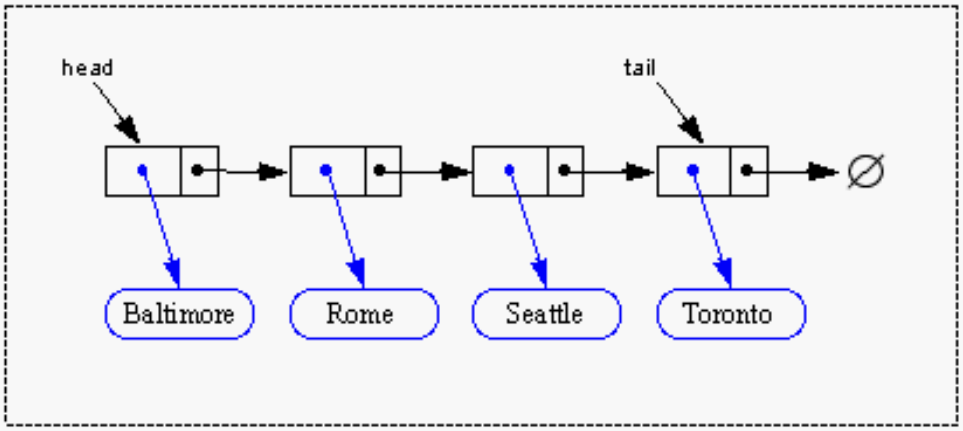
- Uses more space; more code
- Better running time in many cases; all O(1) except searching.

Implementing a Queue with a Singly Linked List

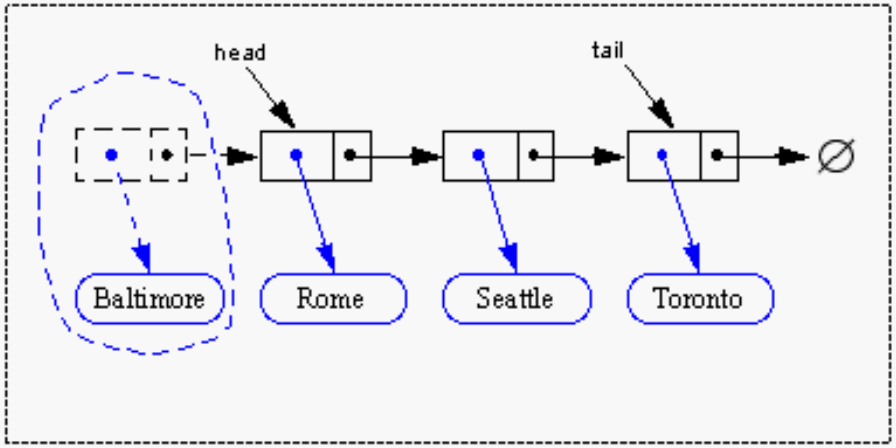


- Head of the list = front of the queue (enqueue)
- Tail of the list = rear of the queue (dequeue)
- *Is this efficient?*

dequeue(): Removing at the Head



- advance head reference

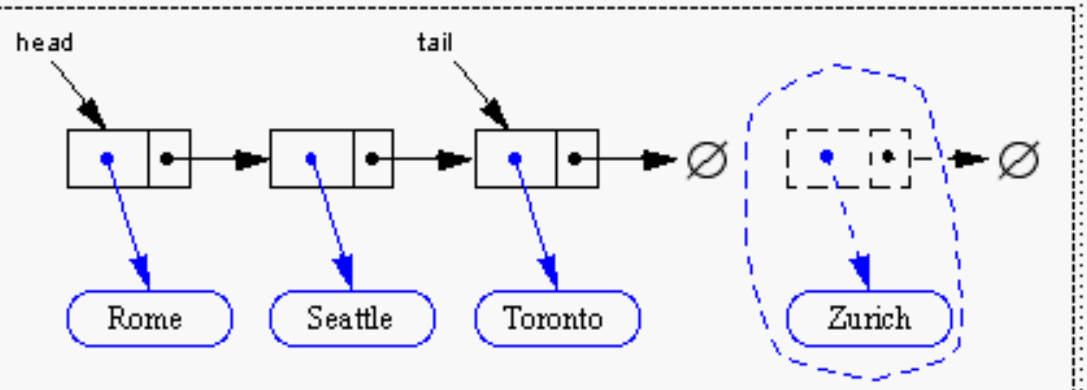


- inserting at the head is just as easy

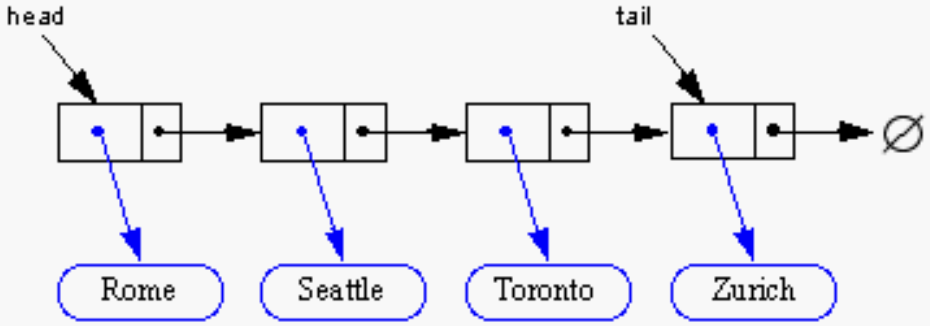
Running time = ?

enqueue(): Inserting at the Tail

- create a new node



- chain it and move the tail reference



- how about removing at the tail?

Running time = ?

Method *enqueue*() in Java

```
public void enqueue(Object obj) {
  Node node = new Node();
  node.setElement(obj);
  node.setNext(null);      // node will be new tail node
  if (size == 0)
    head = node;           // special case of a previously empty queue
  else
    tail.setNext(node);    // add node at the tail of the list
  tail = node;             // update the reference to the tail node
  size++;
}
```

Method *dequeue*() in Java

```
public Object dequeue() throws QueueEmptyException {
  Object obj;
  if (size == 0)
    throw new QueueEmptyException("Queue is empty.");
  obj = head.getElement();
  head = head.getNext();
  size—;
  if (size == 0)
    tail = null;     // the queue is now empty
  return obj;
}
```

Analysis of Implementation with Singly-Linked Lists

- Each methods runs in $O(1)$ time
- Note: Removing at the tail of a singly-linked list requires $\theta(n)$ time. Avoid this!

Comparison with array-based implementation:

- No upper bound on the size of the queue (subject to memory availability)
- More space used per element (*next* pointer)
- Implementation is more complicated (pointer manipulations)
- Method calls consume time (*setNext*, *getNext*, etc.)

Homework

- Study Code Fragment 6.10 (array-based implementation)
- Study Code Fragment 6.11 (implementing a queue using the SinglyLinkedList ADT)

Next time …

- Double-ended Queues (DEQues) (6.3)
- Extendable Arrays (7.2.3)