Queues

cse2011

section 5.2 of textbook

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)
  - …

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 ***front***(): returns the element at the front without removing it

- Execution of ***dequeue***() or ***front***() 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 ***front***()
- **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 front()
  throws   EmptyQueueException;
public Object dequeue()
  throws
     EmptyQueueException;
public void enqueue (Object
     obj);
}
```

Queue Example

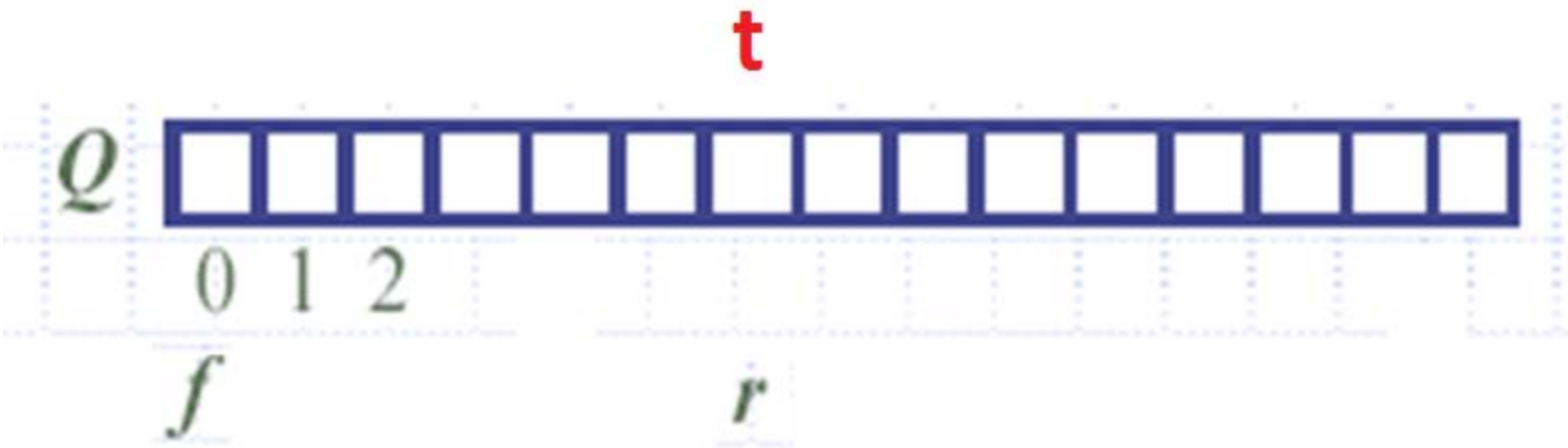
Operation	*Output*	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
front()	*7*	(7)
dequeue()	*7*	()
dequeue()	*"error"*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()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

Array-based Implementation
Not Efficient

- Front is fixed as the first element.
- We only keep one index, **t**, to keep track of the rear.
- enqueue is **O(?).**
- dequeue is **O(?).**

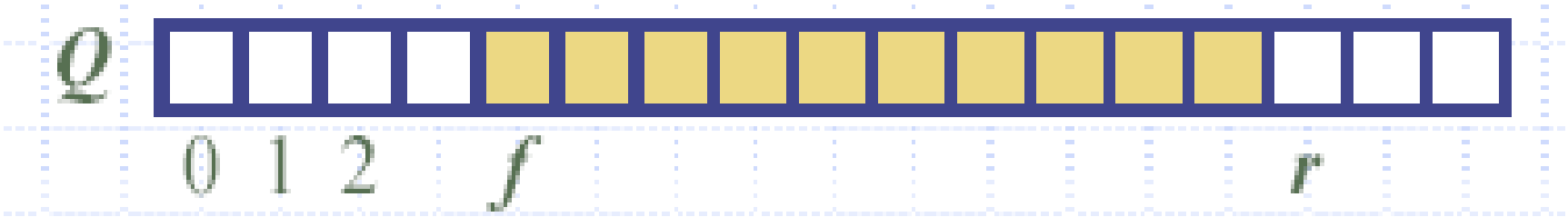


Array-based Implementation

- An array ***Q*** of maximum size ***N***
- Need to keep track the **front** and **rear** of the queue:

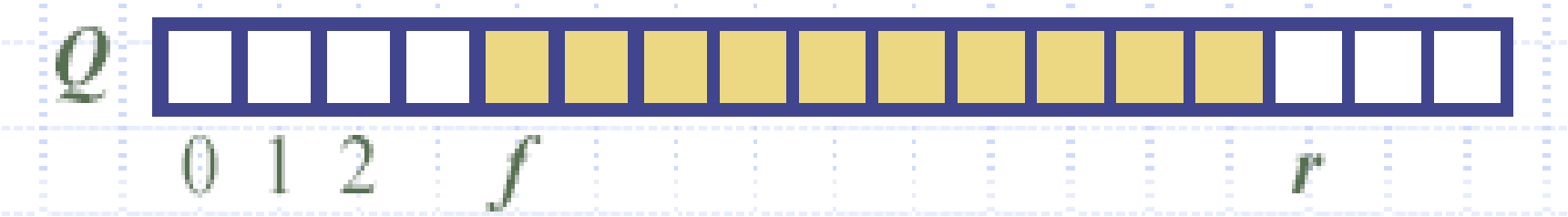
  f: index of the front object

  r: index immediately past the rear element
- Note: *Q*[*r*] is empty (does not store any object)



Array-based Implementation

- Front element: $Q[f]$
- Rear element: $Q[r - 1]$
- Queue is empty: $f = r$
- Queue size: $r - f$
- How to dequeue?
- How to enqueue?



Dequeue() and Enqueue()

Algorithm *dequeue*():

if (*isEmpty*())

 throw *QueueEmptyException;*

temp = *Q*[*f*];

f = *f* + 1;

return *temp*;

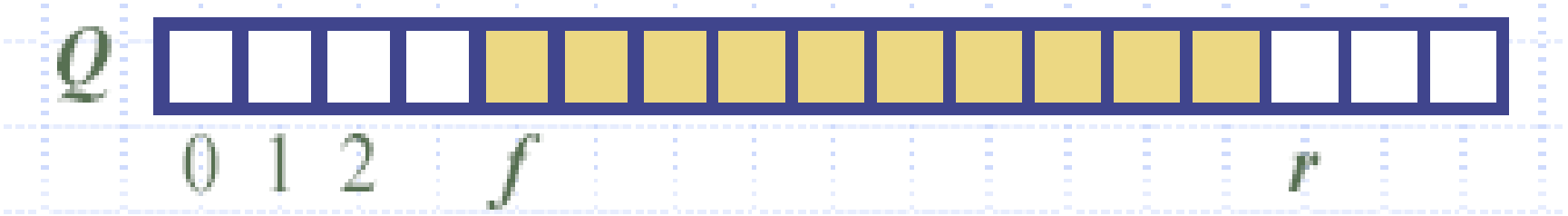
Algorithm *enqueue*(object):

if (*r* == *N*)

 throw *QueueFullException;*

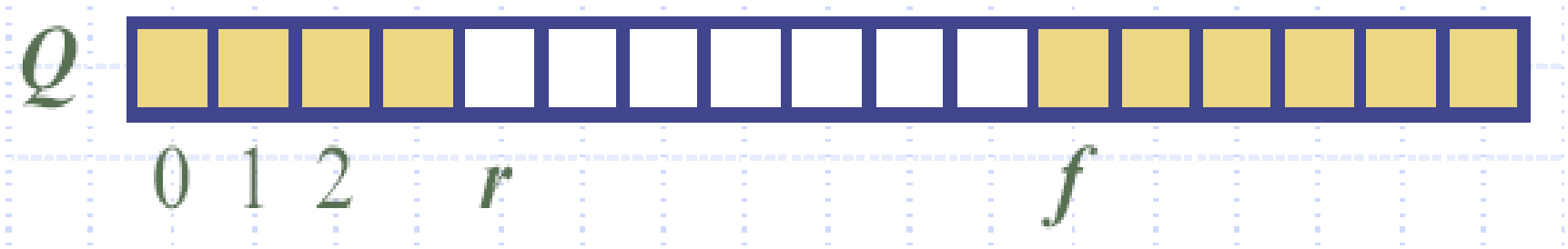
Q[*r*] = object;

r = *r* + 1;



It is assumed that **f ≤ r**. What is the **problem**?

Circular Array Implementation



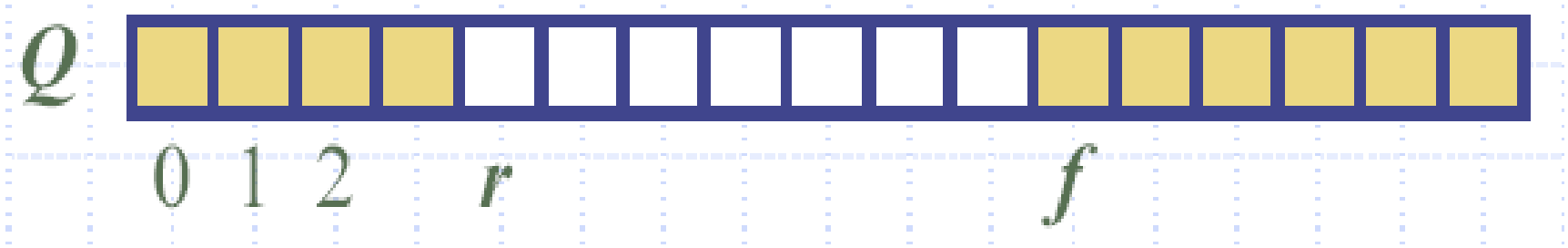
- Analogy:
  A snake chases its tail
- Front element: $Q[f]$
  Rear element: $Q[r - 1]$

- Incrementing f, r
  $f = (f + 1) \bmod N$
  $r = (r + 1) \bmod N$
  mod: Java operator "%"

  It is the remainder after an integral division.

f ≤ r is **not** a valid assumption anymore.

Circular Array Implementation (2)



- Queue size =
  $(N - f + r) \bmod N$
  → verify this
- Queue is empty: $f = r$
- When r reaches and overlaps with f, the queue is full: $r = f$

- To **distinguish** between **empty** and **full** states, we impose a **constraint**: Q can hold at most $N - 1$ objects (one cell is wasted). So r never overlaps with f, except when the queue is empty.

Pseudo-code

Algorithm *enqueue*(object):

if (*size*() == *N* – *1*)

  throw *QueueFullException;*

Q[*r*] = object;

r = (*r* + 1) mod *N*;

Algorithm *dequeue*():

if (*isEmpty*())

 throw *QueueEmptyException;*

temp = *Q*[*f*];

f = (*f* + 1) mod *N*;

return *temp*;

Pseudo-code (2)

Algorithm *front*():
if (*isEmpty*())
 throw *QueueEmptyException;*
return *Q*[*f*];

Algorithm *isEmpty*():
 return (*f* = *r*);

Algorithm *size*():
 return ((*N* – *f* + *r*) mod *N*);

Homework: Remove the constraint "*Q* can hold at most *N* – 1 objects". That is, *Q* can store up to *N* objects. Implement the Queue ADT using a circular array.

Note: there is no corresponding built-in Java class for queue ADT

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;
   }
```

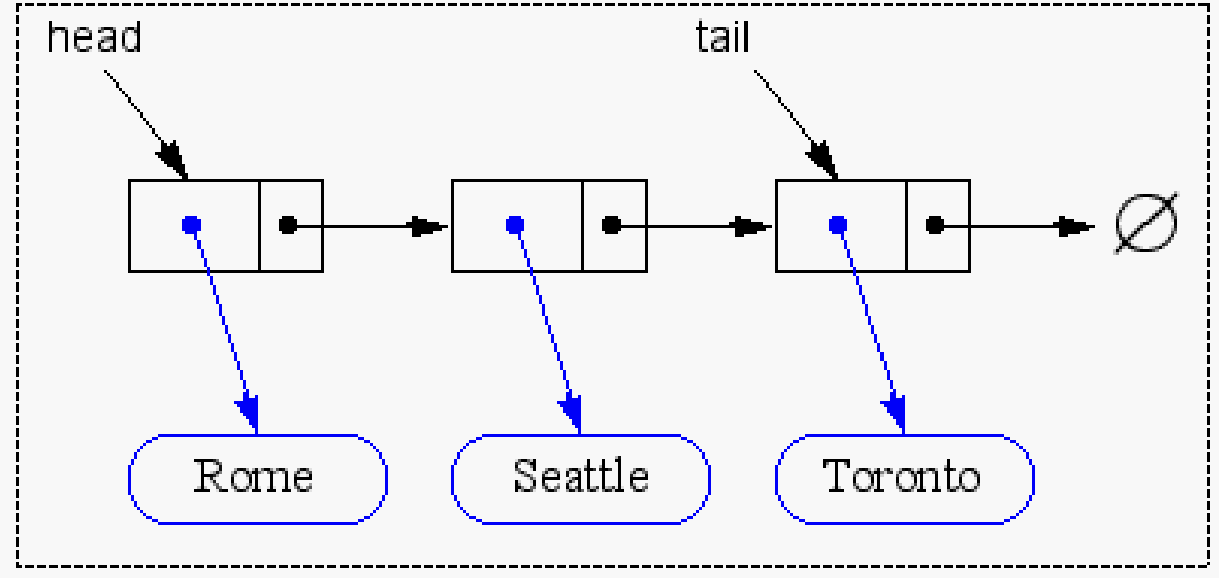
- Needs less space.
- Simpler code in some cases.
- Insertion at tail takes **O(n)**.

- Doubly linked list

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

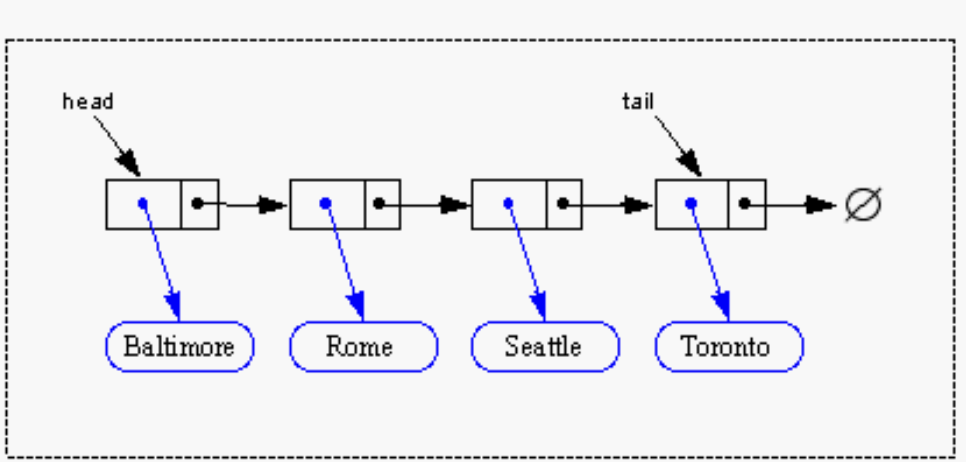
- Better running time in many cases (discussed before).

Implementing a Queue with a Singly Linked List

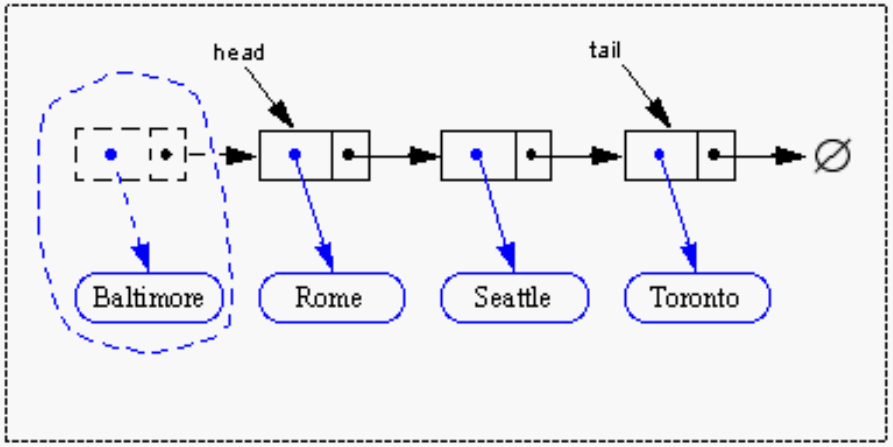


- Head of the list = rear of the queue (enqueue)
- Tail of the list = front of the queue (dequeue)
- *Is this efficient?*

dequeue(): Removing at the Head



- advance head reference

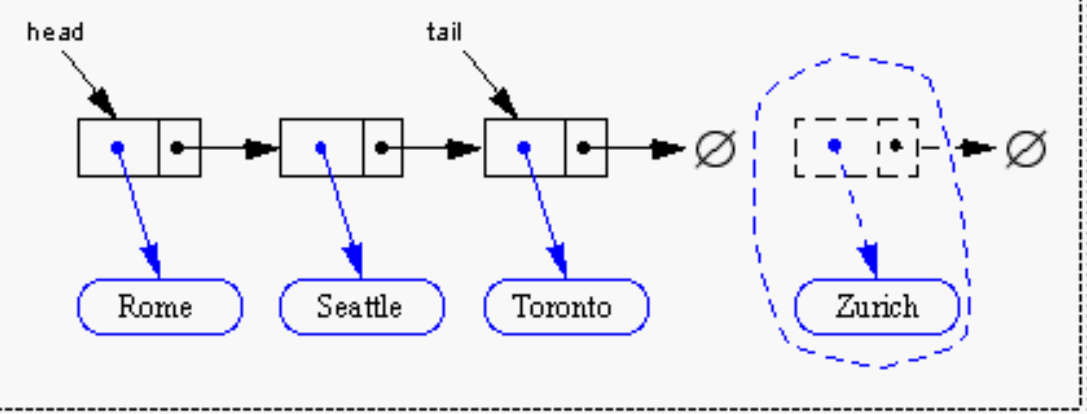


- inserting at the head is just as easy

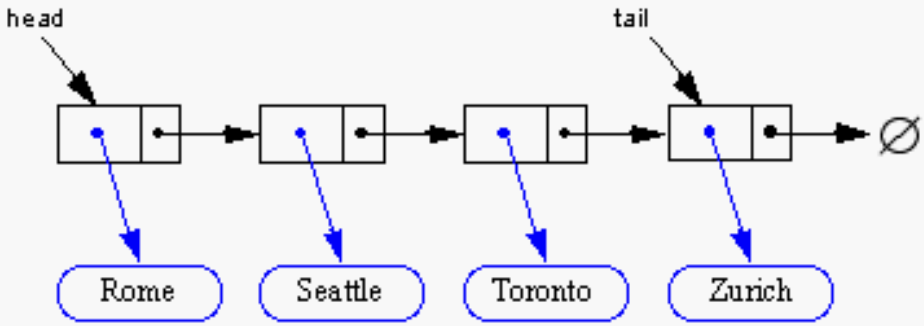
Running time = ?

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 $O(n)$ 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

- In the doubly linked list implementation, we use two dummy nodes, *header* and *trailer*, to make coding simple.
- Why don't we use two dummy nodes in the singly linked list implementation?
- *Hint*: Implement the queue operations using a singly linked list with two dummy nodes *header* and *trailer*.