Queues

CSE 2011

Fall 2009

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 ***front***(): returns the element at the front <u>without removing</u> 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

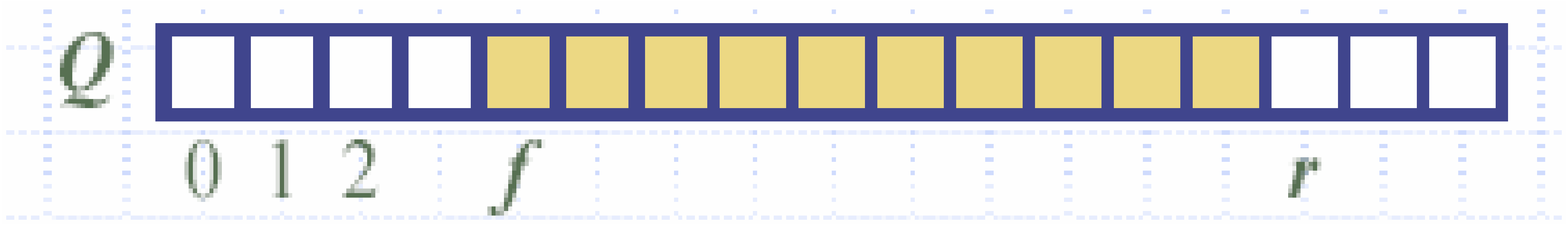
Operation	***Output***	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
front()	*7*	(7)
dequeue()	*7*	()
dequeue()	*"error"*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()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

Array-based Implementation

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

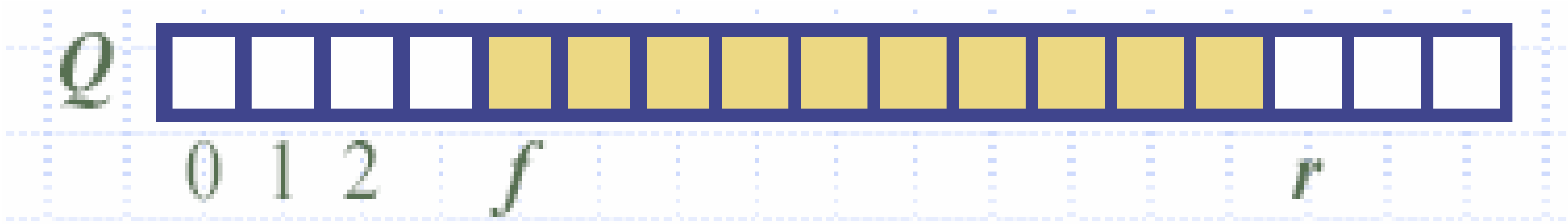
  f: index of the front object

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



Array-based Implementation

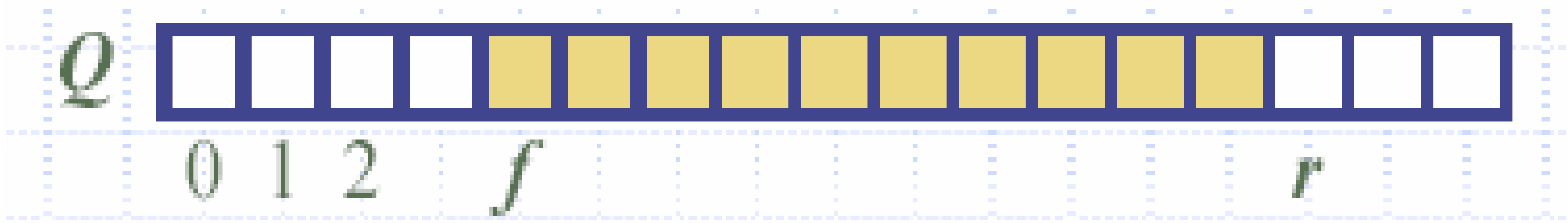
- Front element: *Q*[*f*]
- Rear element: *Q*[*r – 1*]
- Queue is empty: *f* = *r*
- Queue size*: r – f*



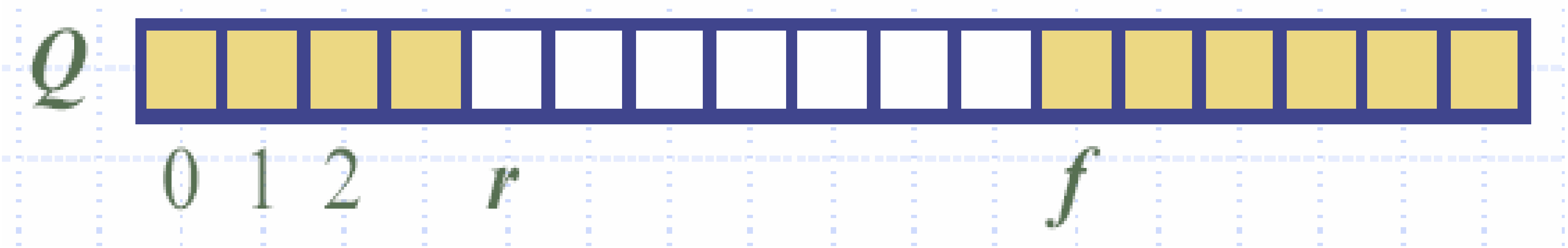
Dequeue() and Enqueue()

```
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;
```



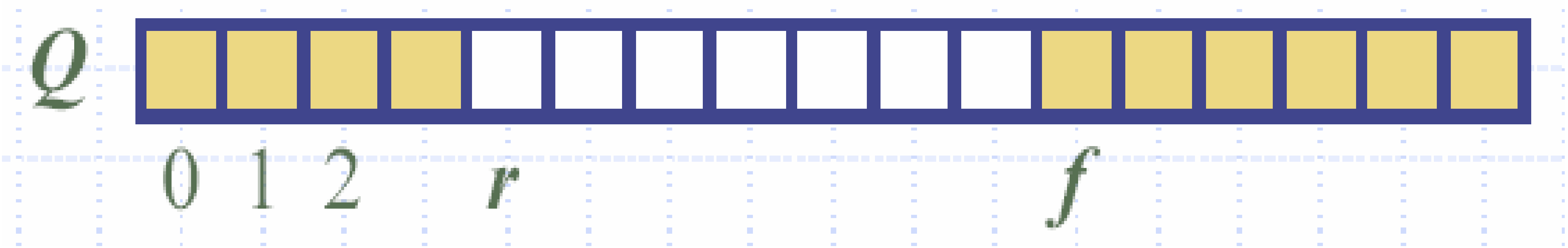
Circular Array Implementation



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

- Incrementing *f*, *r*
  f = (*f* + 1) mod *N*
  r = (*r* + 1) mod *N*
  mod: Java operator “%”

Circular Array Implementation



- Queue size =
  (*N* – *f* + *r*) mod *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;
$r = (r + 1) \bmod N$;
```

```
Algorithm dequeue():
if (isEmpty())
    throw QueueEmptyException;
temp = Q[f];
$f = (f + 1) \bmod N$;
return temp;
```

Pseudo-code

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

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

Algorithm size():
    return ($(N - f + r) \bmod N$);
```

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

```
private static class Node<AnyType>
   {
      public AnyType data;
      public Node<AnyType>   next;
   }
```

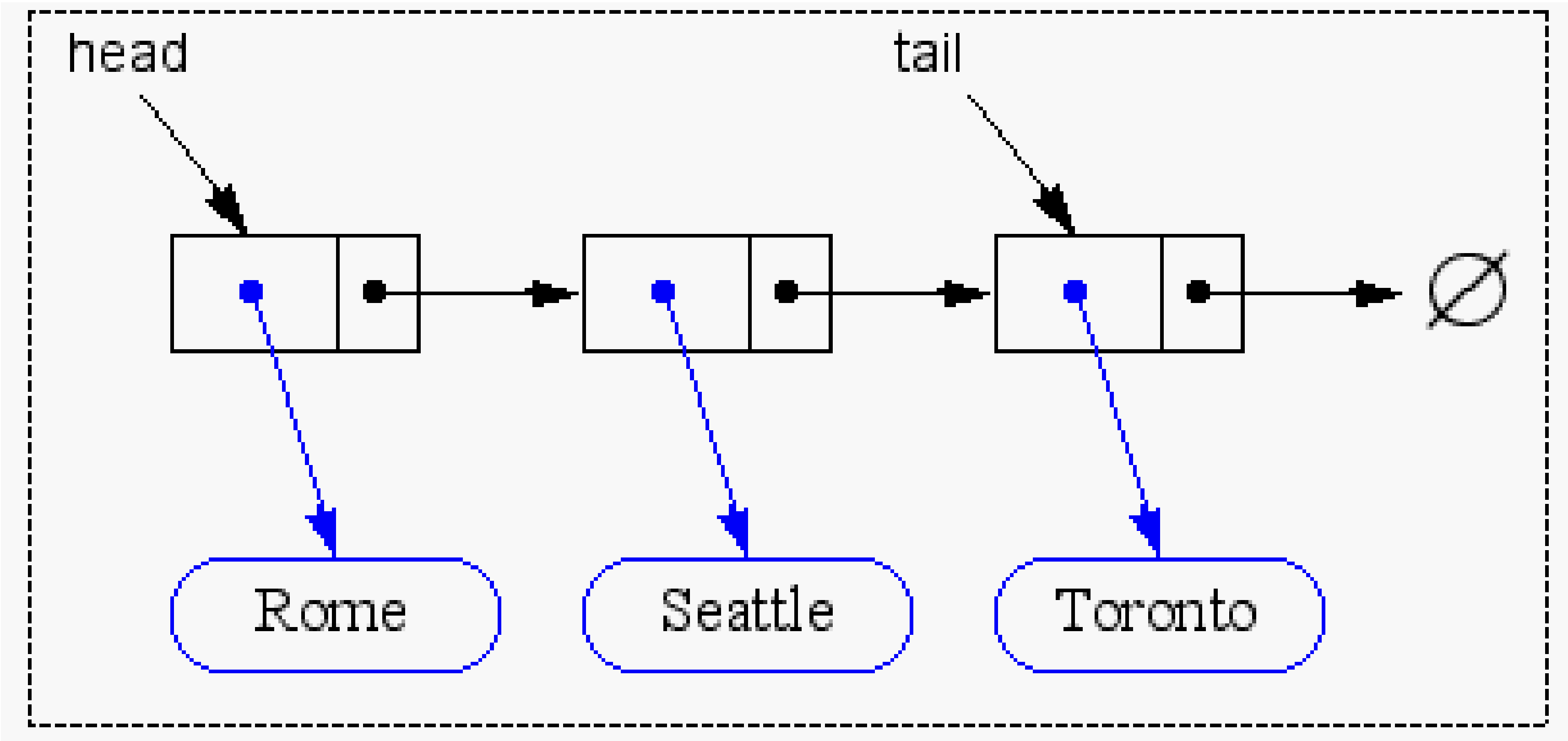
- Needs less space.
- Simpler code in some cases.
- Insertion at tail takes O(n).

- Doubly linked list

```
private static class DNode<AnyType>
   {
      public AnyType data;
      public Node<AnyType>   prev;
      public Node<AnyType>   next;
   }
```

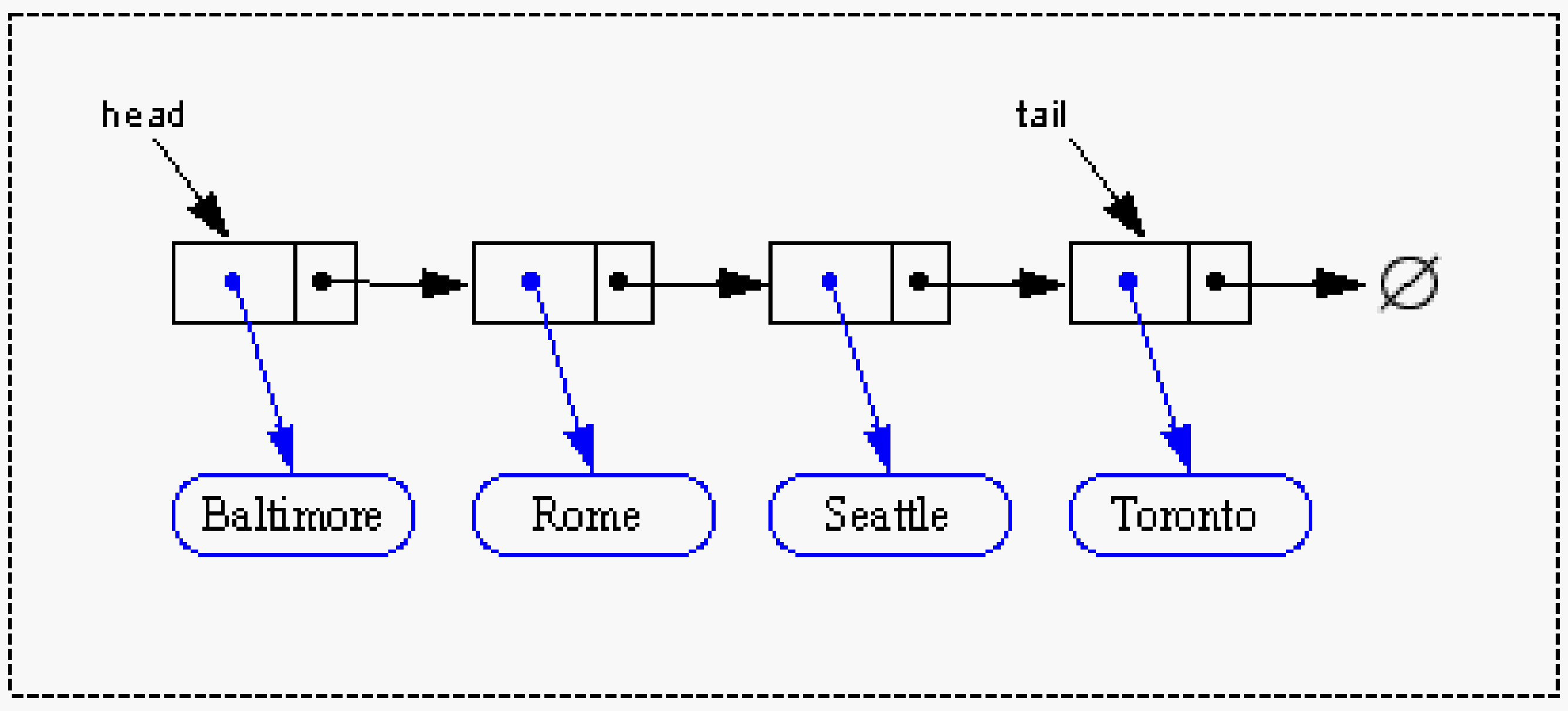
- Better running time in many cases (discussed before).

Implementing a Queue with a Singly Linked List

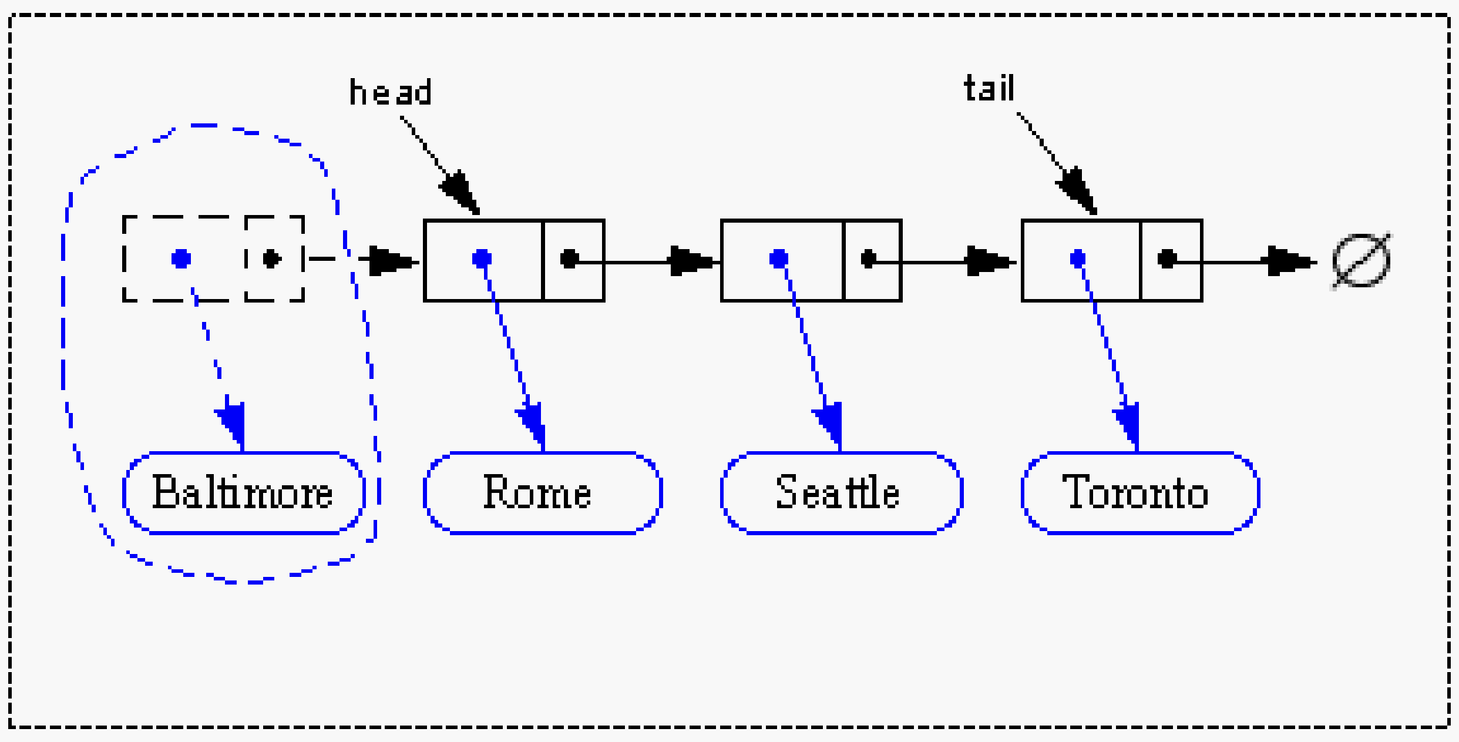


- Head of the list = front of the queue (enqueue)
- Tail of the list = rear of the queue (dequeue)
- *Is this efficient?*

dequeue(): Removing at the Head



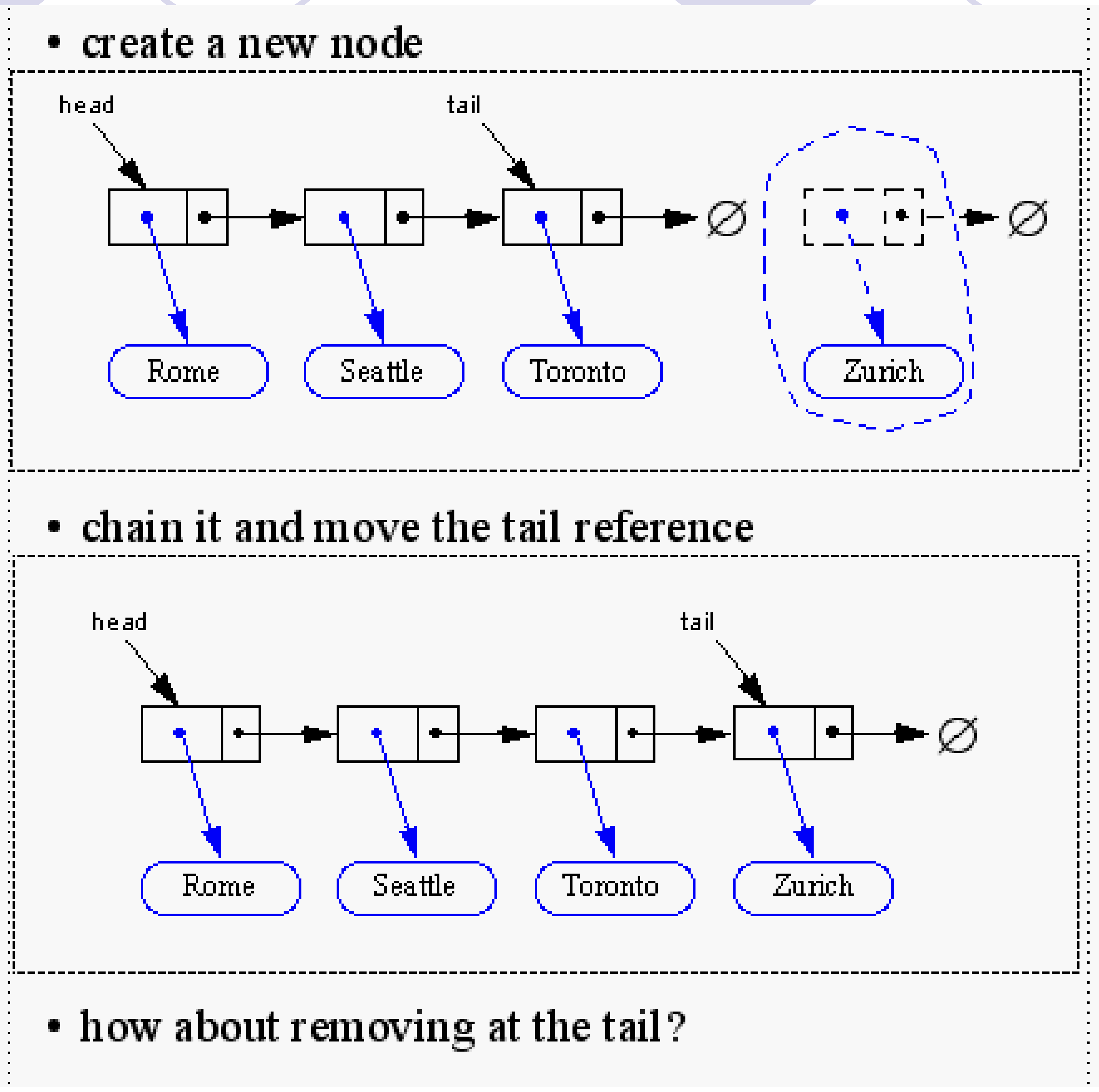
- advance head reference



Running time = ?

- inserting at the head is just as easy

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

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.)

Next time …

- Double-ended Queues (Deques) (5.3)