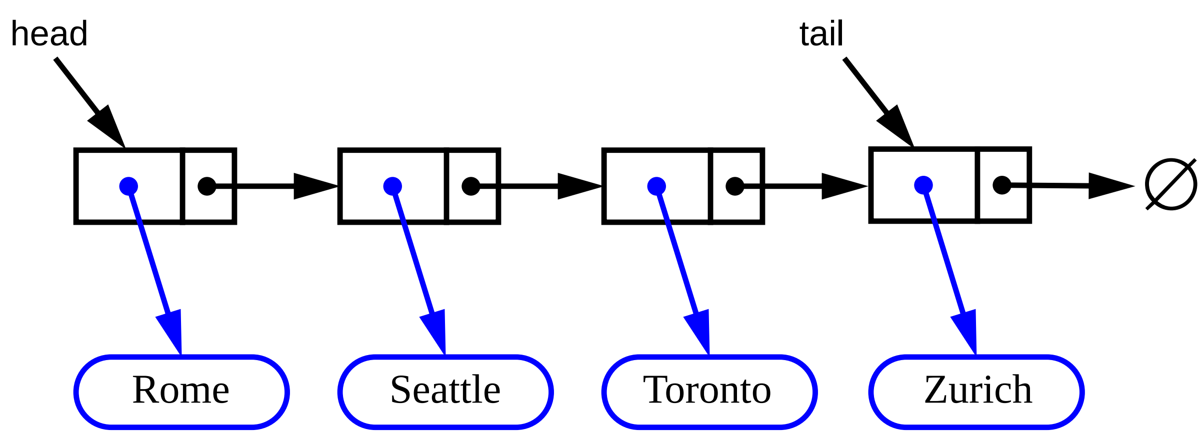
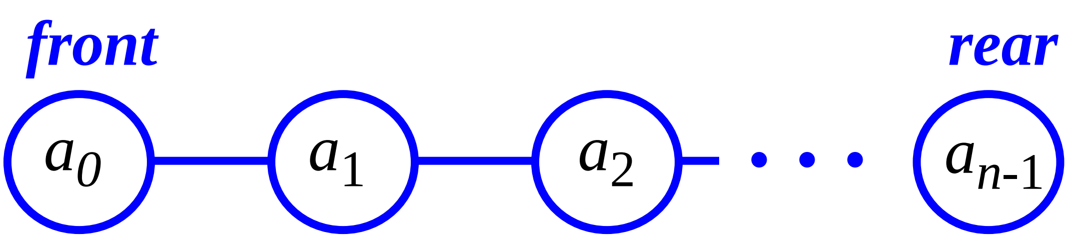
QUEUES AND LINKED LISTS

- Queues
- Linked Lists
- Double-Ended Queues



Queues

- A queue differs from a stack in that its insertion and removal routines follows the first-in-first-out (FIFO) principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the *rear* (enqueued) and removed from the *front* (dequeued)

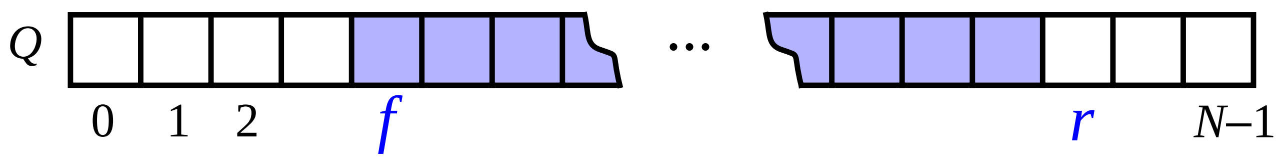


The Queue Abstract Data Type

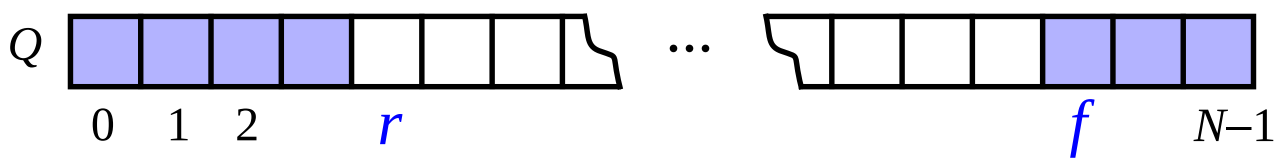
- The queue has two fundamental methods:
  - enqueue(*o*): Insert object *o* at the rear of the queue
  - dequeue(): Remove the object from the front of the queue and return it; an error occurs if the queue is empty
- These support methods should also be defined:
  - size(): Return the number of objects in the queue
  - isEmpty(): Return a boolean value that indicates whether the queue is empty
  - front(): Return, but do not remove, the front object in the queue; an error occurs if the queue is empty

An Array-Based Queue

- Create a queue using an array in a circular fashion
- A maximum size N is specified, e.g. $N = 1{,}000$.
- The queue consists of an N-element array Q and two integer variables:
  - f, index of the front element
  - r, index of the element after the rear one
- "normal configuration"



- "wrapped around" configuration



- What does f=r mean?
- How do we compute the number of elements in the queue from f and r?

An Array-Based Queue (contd.)

- Pseudo-Code (contd.)

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

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

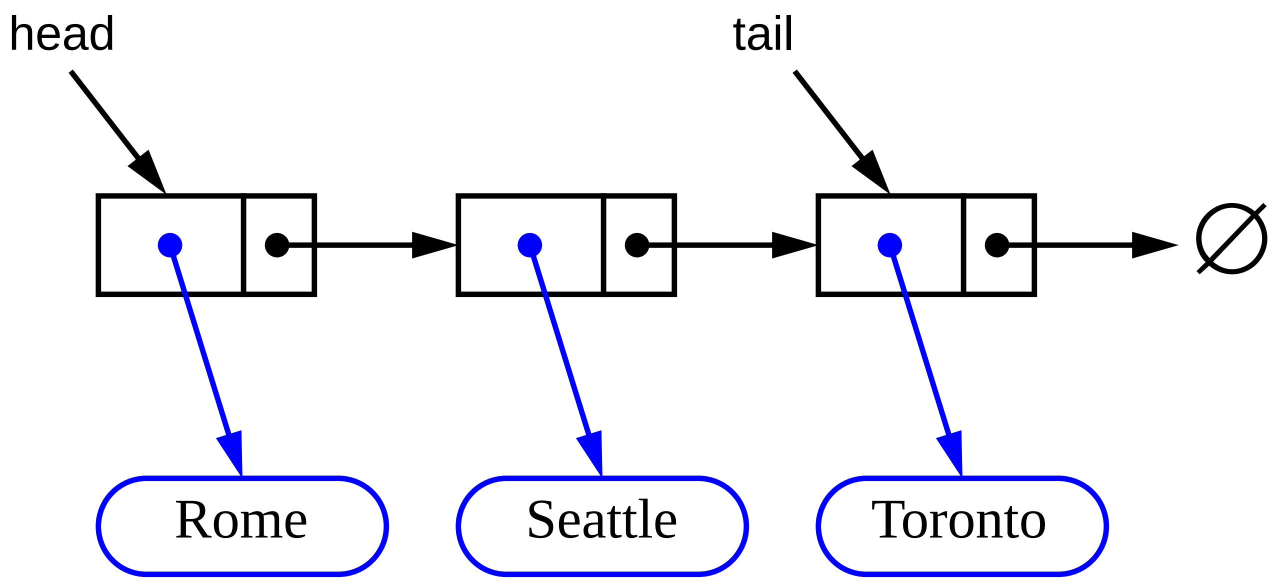
Algorithm front():
  if isEmpty() **then**
    throw a QueueEmptyException
  return $Q[f]$

Algorithm dequeue():
  if isEmpty() **then**
    throw a QueueEmptyException
  $temp \leftarrow Q[f]$
  $Q[f] \leftarrow$ **null**
  $f \leftarrow (f + 1) \bmod N$
  return *temp*

Algorithm enqueue(o):
  if size $= N - 1$ **then**
    throw a QueueFullException
  $Q[r] \leftarrow o$

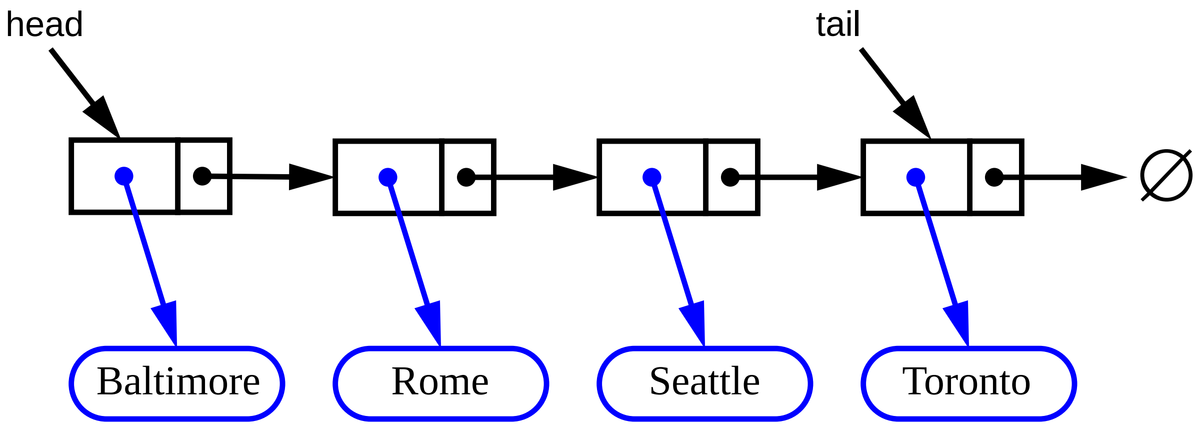
Implementing a Queue with a Singly Linked List

- nodes connected in a chain by links

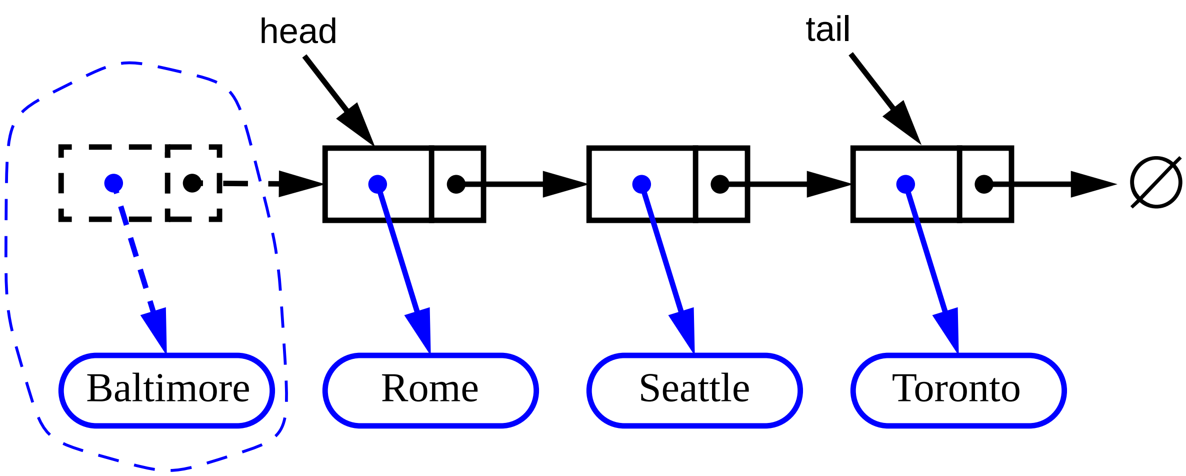


- the head of the list is the front of the queue, the tail of the list is the rear of the queue
- why not the opposite?

Removing at the Head



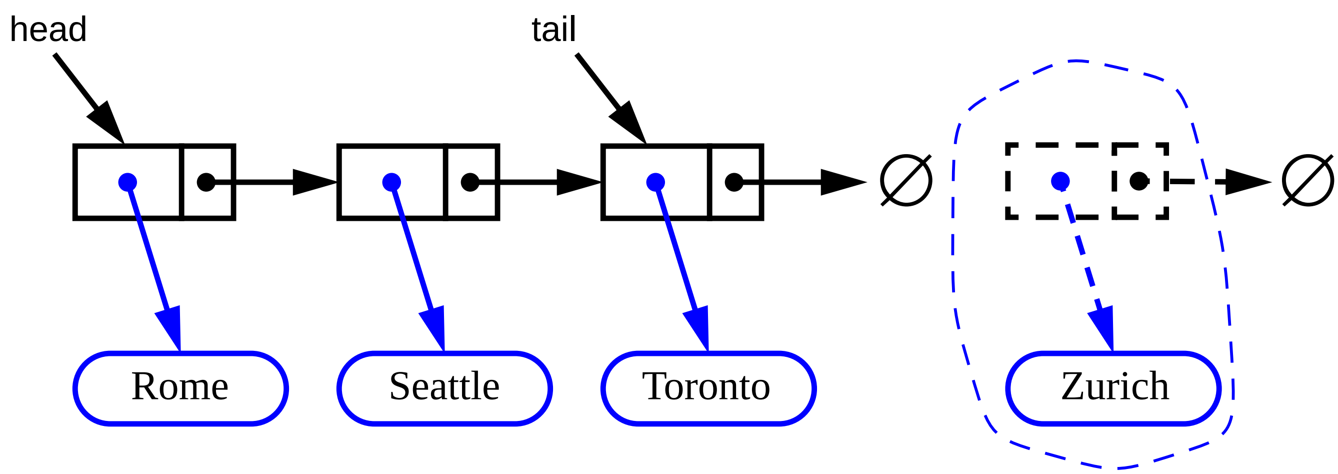
- advance head reference



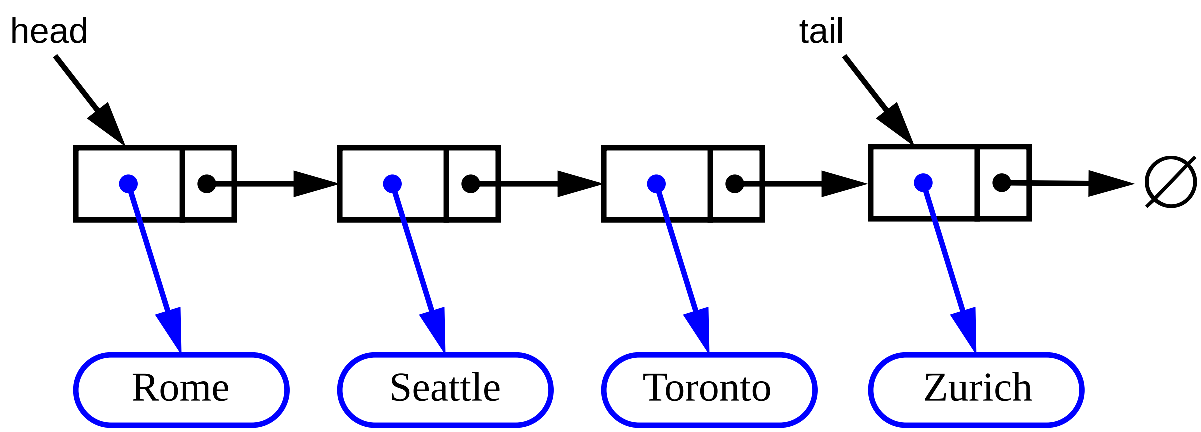
- inserting at the head is just as easy

Inserting at the Tail

- create a new node



- chain it and move the tail reference



- how about removing at the tail?

Double-Ended Queues

- A double-ended queue, or deque, supports insertion and deletion from the front and back.
- The Deque Abstract Data Type
  - insertFirst(*e*): Insert e at the deginning of deque.
  - insertLast(*e*): Insert e at end of deque
  - removeFirst(): Removes and returns first element
  - removeLast(): Removes and returns last element
- Additionally supported methods include:
  - first()
  - last()
  - size()
  - isEmpty()

Implementing Stacks and Queues with Deques

- Stacks with Deques:

Stack Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
top()	last()
push(e)	insertLast(e)
pop()	removeLast()

- Queues with Deques:

Queue Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue()	insertLast(e)
dequeue()	removeFirst()

The Adaptor Pattern

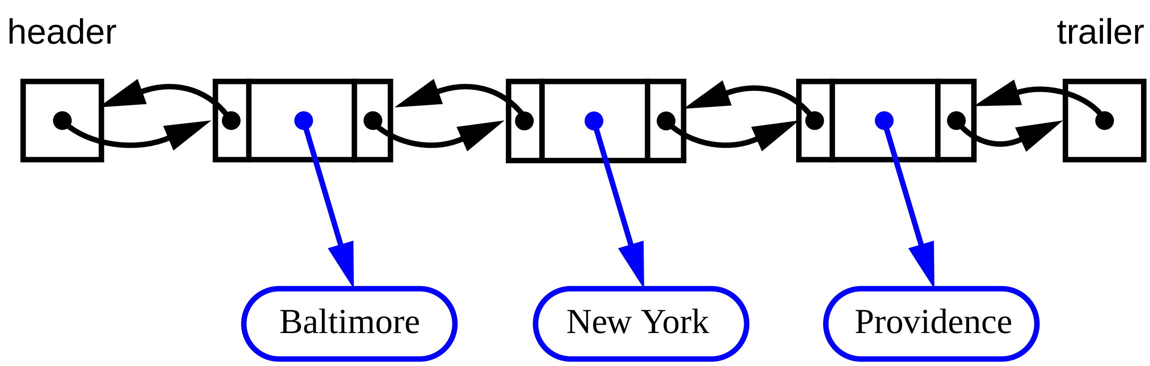
- Using a deque to implement a stack or queue is an example of the adaptor pattern. Adaptor patterns implement a class by using methods of another class
- In general, adaptor classes specialize general classes
- Two such applications:
  - Specialize a general class by changing some methods.

    Ex: implementing a stack with a deque.
  - Specialize the types of objects used by a general class.

    Ex: Defining an IntegerArrayStack class that adapts ArrayStack to only store integers.

Implementing Deques with Doubly Linked Lists

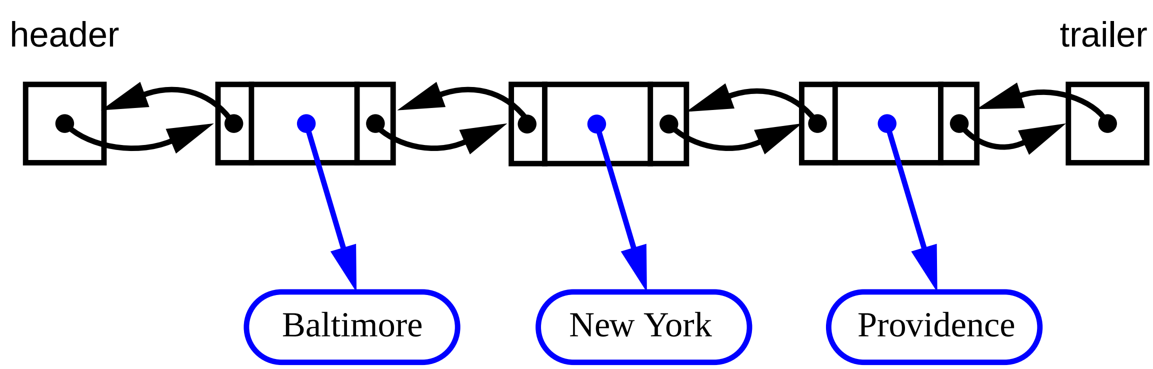
- Deletions at the tail of a singly linked list cannot be done in constant time.
- To implement a deque, we use a doubly linked list. with special header and trailer nodes.



- A node of a doubly linked list has a next and a prev link. It supports the following methods:
  - setElement(Object e)
  - setNext(Object newNext)
  - setPrev(Object newPrev)
  - getElement()
  - getNext()
  - getPrev()
- By using a doubly linked list, all the methods of a deque run in O(1) time.

Implementing Deques with Doubly Linked Lists (cont.)

- When implementing a doubly linked lists, we add two special nodes to the ends of the lists: the header and trailer nodes.
  - The header node goes before the first list element. It has a valid next link but a null prev link.
  - The trailer node goes after the last element. It has a valid prev reference but a null next reference.
- The header and trailer nodes are sentinel or "dummy" nodes because they do not store elements.
- Here's a diagram of our doubly linked list:



Implementing Deques with Doubly Linked Lists (cont.)

- Here's a visualization of the code for removeLast().