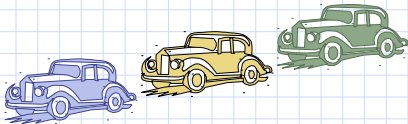


Queues



© 2013 Goodrich, Tamassia, Goldwasser

Queues

1

The Queue ADT

▪ The **Queue** ADT stores arbitrary objects

▪ Insertions and deletions follow the first-in first-out scheme

▪ Insertions are at the rear of the queue and removals are at the front of the queue

▪ Main queue 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

▪ Auxiliary queue operations:

- object **first**(): returns the element at the front without removing it
- integer **len**(): returns the number of elements stored
- boolean **is\_empty**(): indicates whether no elements are stored

▪ Exceptions

- Attempting the execution of dequeue or front on an empty queue throws an **EmptyQueueException**

© 2013 Goodrich, Tamassia, Goldwasser

Queues

2

Example

| Operation    | Return Value | first ← Q ← last |
|--------------|--------------|------------------|
| Q.enqueue(5) | –            | [5]              |
| Q.enqueue(3) | –            | [5, 3]           |
| len(Q)       | 2            | [5, 3]           |
| Q.dequeue()  | 5            | [3]              |
| Q.is_empty() | False        | [3]              |
| Q.dequeue()  | 3            | []               |
| Q.is_empty() | True         | []               |
| Q.dequeue()  | “error”      | []               |
| Q.enqueue(7) | –            | [7]              |
| Q.enqueue(9) | –            | [7, 9]           |
| Q.first()    | 7            | [7, 9]           |
| Q.enqueue(4) | –            | [7, 9, 4]        |
| len(Q)       | 3            | [7, 9, 4]        |
| Q.dequeue()  | 7            | [9, 4]           |

© 2013 Goodrich, Tamassia, Goldwasser

Queues

3

Applications of Queues

▪ Direct applications

- Waiting lists, bureaucracy
- Access to shared resources (e.g., printer)
- Multiprogramming

▪ Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

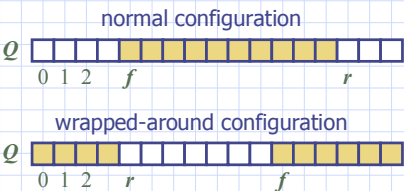
© 2013 Goodrich, Tamassia, Goldwasser

Queues

4

## Array-based Queue

- Use an array of size  $N$  in a circular fashion
- Two variables keep track of the front and rear
  - $f$ : index of the front element
  - $r$ : index immediately past the rear element
- Array location  $r$  is kept empty

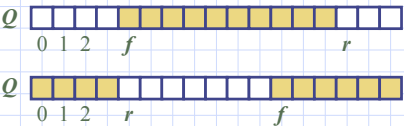


## Queue Operations

- We use the modulo operator (remainder of division)

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

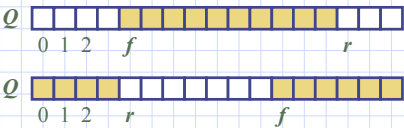
**Algorithm *isEmpty()***  
**return**  $(f = r)$



## Queue Operations (cont.)

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

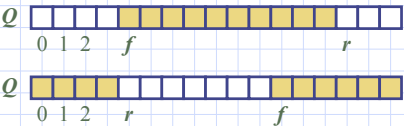
**Algorithm *enqueue(o)***  
**if**  $size() = N - 1$  **then**  
    **throw** *FullQueueException*  
**else**  
     $Q[r] \leftarrow o$   
     $r \leftarrow (r + 1) \bmod N$



## Queue Operations (cont.)

- Operation `dequeue` throws an exception if the queue is empty
- This exception is specified in the queue ADT

**Algorithm *dequeue()***  
**if**  $isEmpty()$  **then**  
    **throw** *EmptyQueueException*  
**else**  
     $o \leftarrow Q[f]$   
     $f \leftarrow (f + 1) \bmod N$   
    **return**  $o$



# Queue in Python

- Use the following three instance variables:
  - `_data`: is a reference to a list instance with a fixed capacity.
  - `_size`: is an integer representing the current number of elements stored in the queue (as opposed to the length of the data list).
  - `_front`: is an integer that represents the index within `data` of the first element of the queue (assuming the queue is not empty).

# Queue in Python, Beginning

```
1 class ArrayQueue:
2     """FIFO queue implementation using a Python list as underlying storage."""
3     DEFAULT_CAPACITY = 10 # moderate capacity for all new queues
4
5     def __init__(self):
6         """Create an empty queue."""
7         self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
8         self._size = 0
9         self._front = 0
10
11    def __len__(self):
12        """Return the number of elements in the queue."""
13        return self._size
14
15    def is_empty(self):
16        """Return True if the queue is empty."""
17        return self._size == 0
18
19    def first(self):
20        """Return (but do not remove) the element at the front of the queue.
21        Raise Empty exception if the queue is empty.
22        """
23        if self.is_empty():
24            raise Empty('Queue is empty')
25        return self._data[self._front]
26
27    def dequeue(self):
28        """Remove and return the first element of the queue (i.e., FIFO).
29        Raise Empty exception if the queue is empty.
30        """
31        if self.is_empty():
32            raise Empty('Queue is empty')
33        answer = self._data[self._front]
34        self._data[self._front] = None # help garbage collection
35        self._front = (self._front + 1) % len(self._data)
36        self._size -= 1
37        return answer
```

# Queue in Python, Continued

```
40 def enqueue(self, e):
41     """Add an element to the back of queue."""
42     if self._size == len(self._data):
43         self._resize(2 * len(self._data)) # double the array size
44     avail = (self._front + self._size) % len(self._data)
45     self._data[avail] = e
46     self._size += 1
47
48 def _resize(self, cap):
49     """Resize to a new list of capacity >= len(self)."""
50     # we assume cap >= len(self)
51     old = self._data
52     self._data = [None] * cap # keep track of existing list
53     # allocate list with new capacity
54     walk = self._front
55     for k in range(self._size):
56         self._data[k] = old[walk] # only consider existing elements
57         # intentionally shift indices
58         walk = (1 + walk) % len(old) # use old size as modulus
59     self._front = 0 # front has been realigned
```

# Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue `Q` by repeatedly performing the following steps:

- `e = Q.dequeue()`
- Service element `e`
- `Q.enqueue(e)`

