# Queues

# Introduction to Queues

- A queue is a **waiting line**

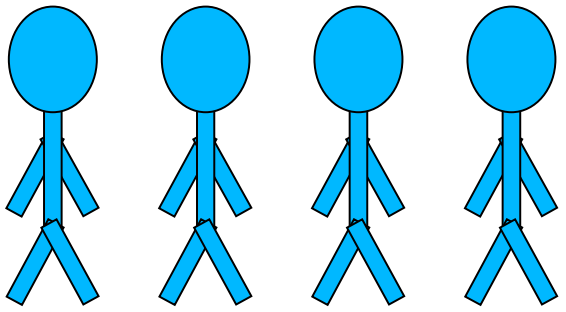
- It's in daily life:
  - A line of persons waiting to check out at a supermarket
  - A line of persons waiting to purchase a ticket for a film
  - A line of planes waiting to take off at an airport
  - A line of vehicles at a toll booth

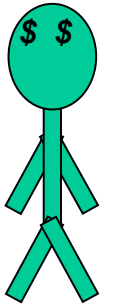
# Introduction to Queues

- Difference between **Stack** and **Queues**:
  - Stack exhibits last-in-first-out (LIFO)
  - Queue exhibits first-in-first-out (FIFO)

# The Queue Operations

A queue is like a line of people waiting for a bank teller. The queue has a **front** and a **rear**.



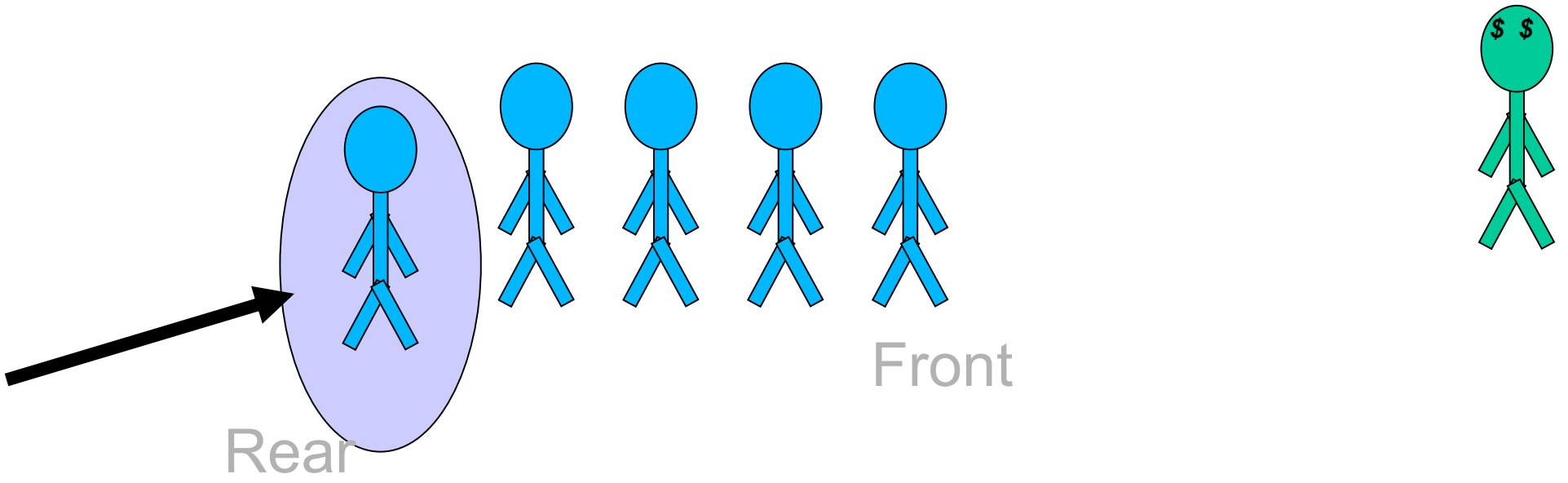



Front

Rear

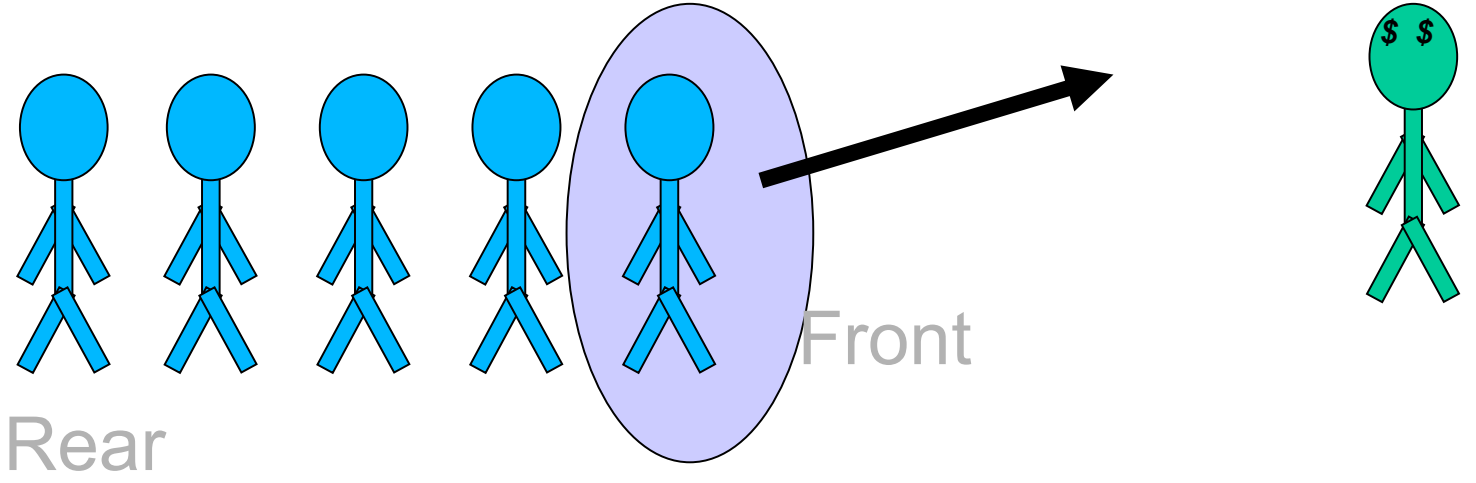
# The Queue Operations

New people must enter the queue at the rear. The C++ queue class calls this a **push**, although it is usually called an **enqueue** operation.



# The Queue Operations

When an item is taken from the queue, it always comes from the front. The C++ queue calls this a **pop**, although it is usually called a **dequeue** operation.



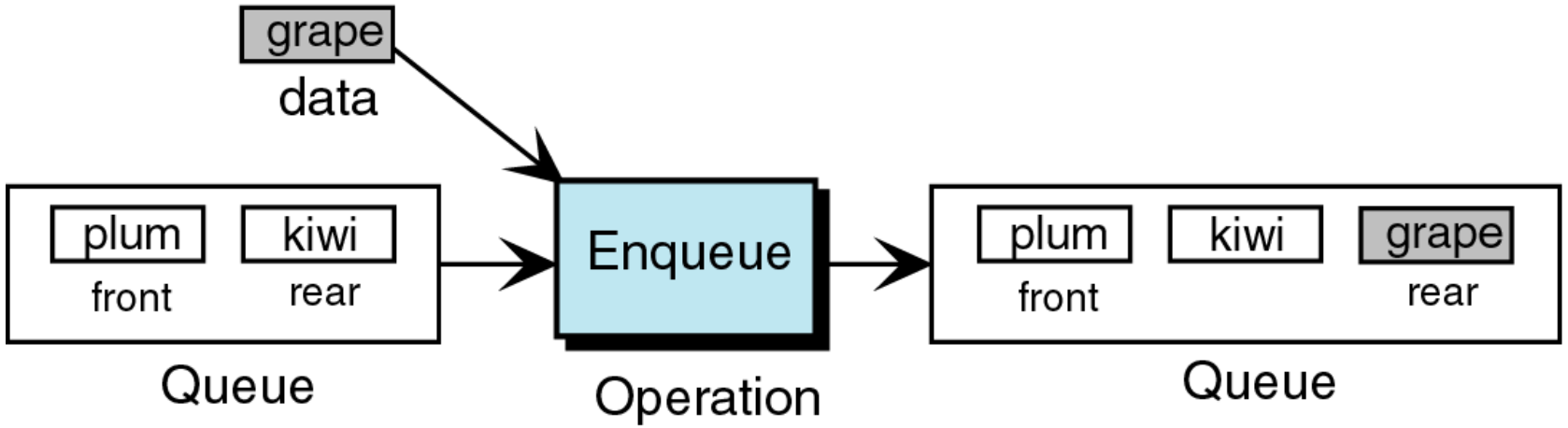
# •Queue Operations

•There are four basic queue operations.

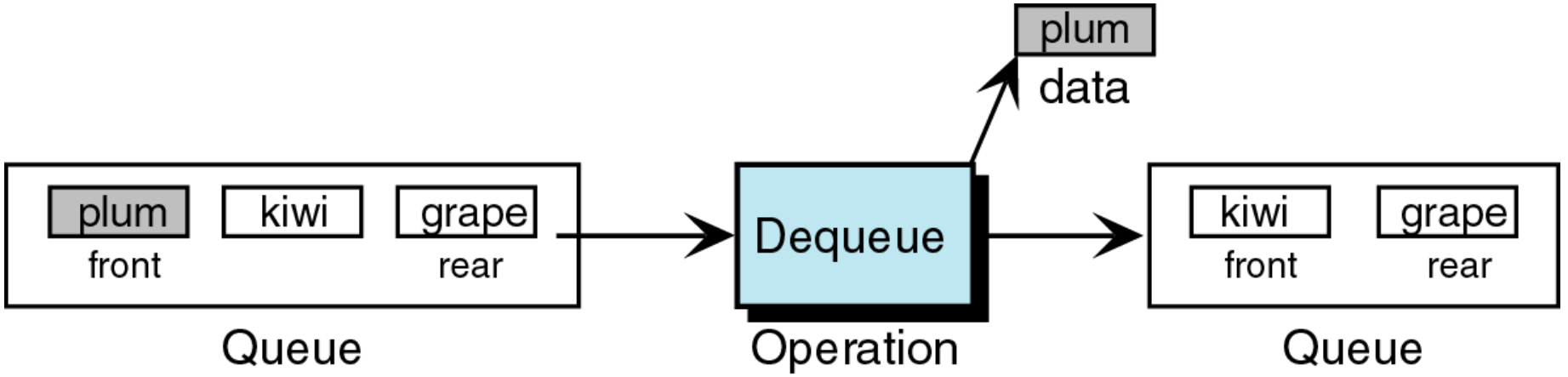
•Data can be inserted at the rear and processed from the front.

1. Enqueue ; inserts an element at the rear of the queue.
2. Dequeue ; deletes an element at the front of the queue.
3. Queue Front; examines the element at the front of the queue.
4. Queue Rear; examines the element at the rear of the queue.

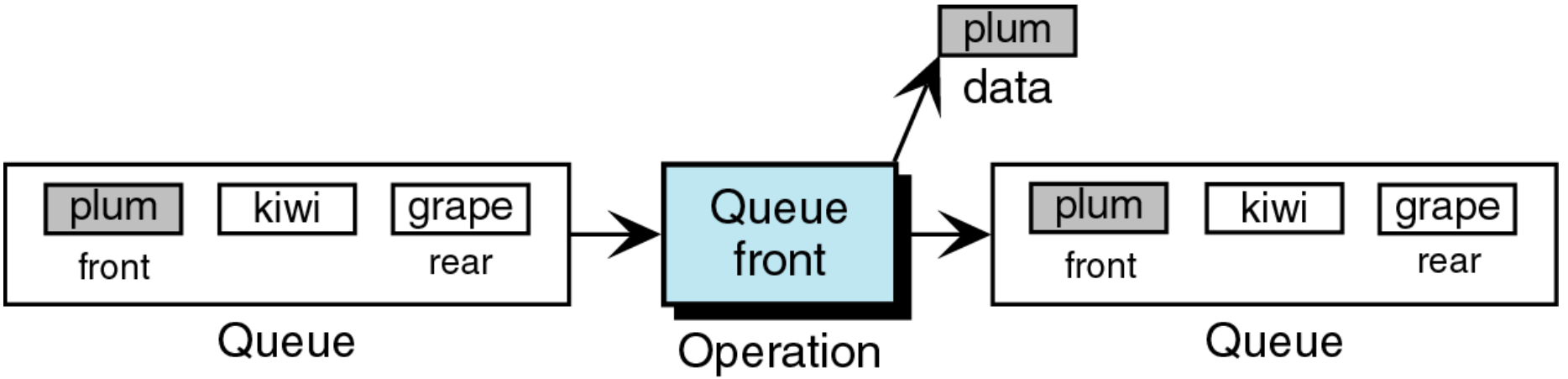
# •Queue Operations



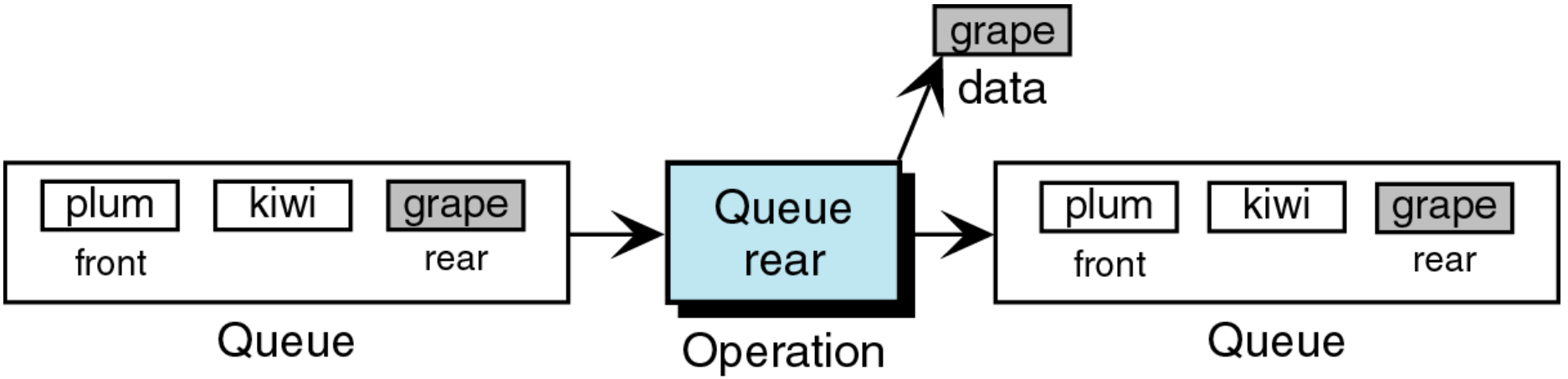
# •Queue Operations



# •Queue Operations



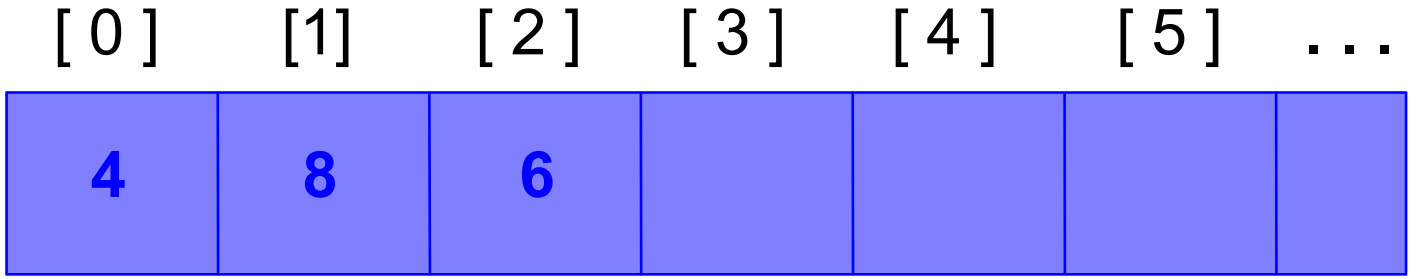
# •Queue Operations




•If there are no data in the queue, then the queue is in an
•**underflow state**.

# Array Implementation

A queue can be implemented with an array, as shown here. For example, this queue contains the integers 4 (at the front), 8 and 6 (at the rear).

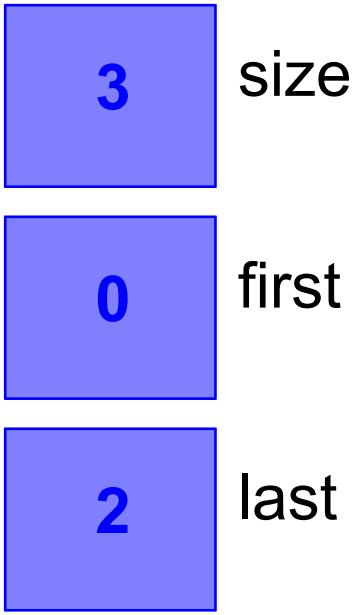


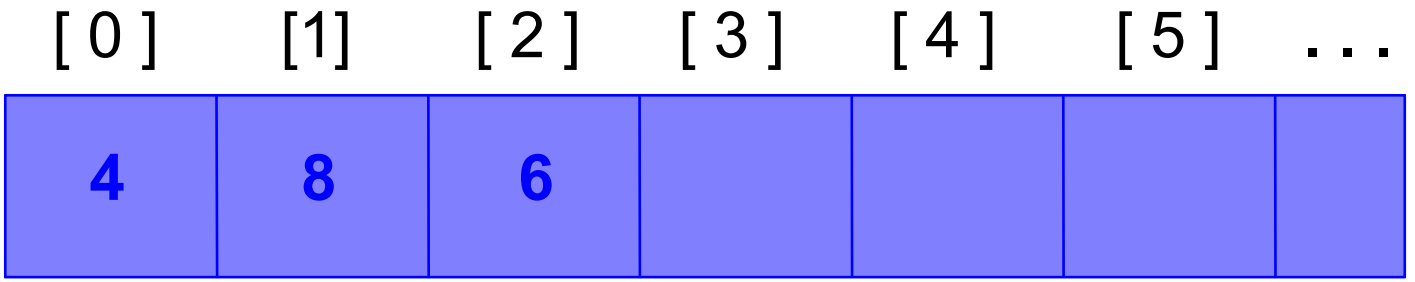

An array of integers to implement a queue of integers

We don't care what's in this part of the array.

# Array Implementation

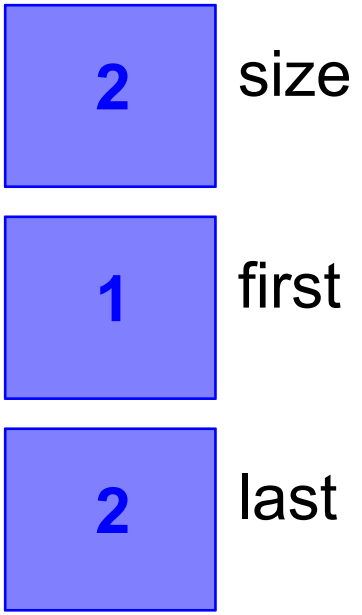
The easiest implementation also keeps track of the number of items in the queue and the index of the first element (at the front of the queue), the last element (at the rear).

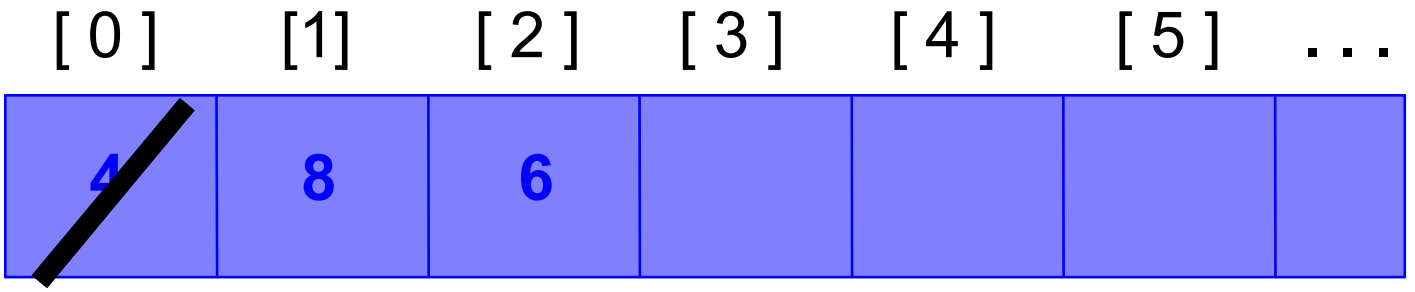


# A Dequeue Operation

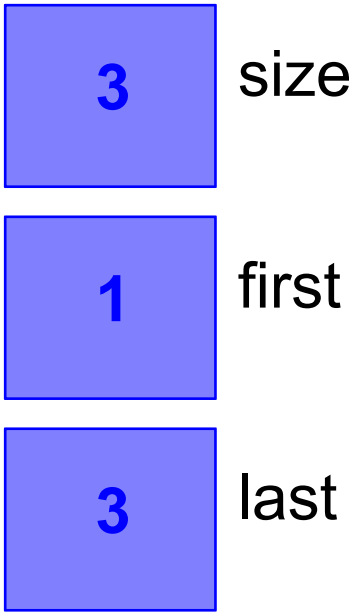
When an element leaves the queue, size is decremented, and first changes, too.

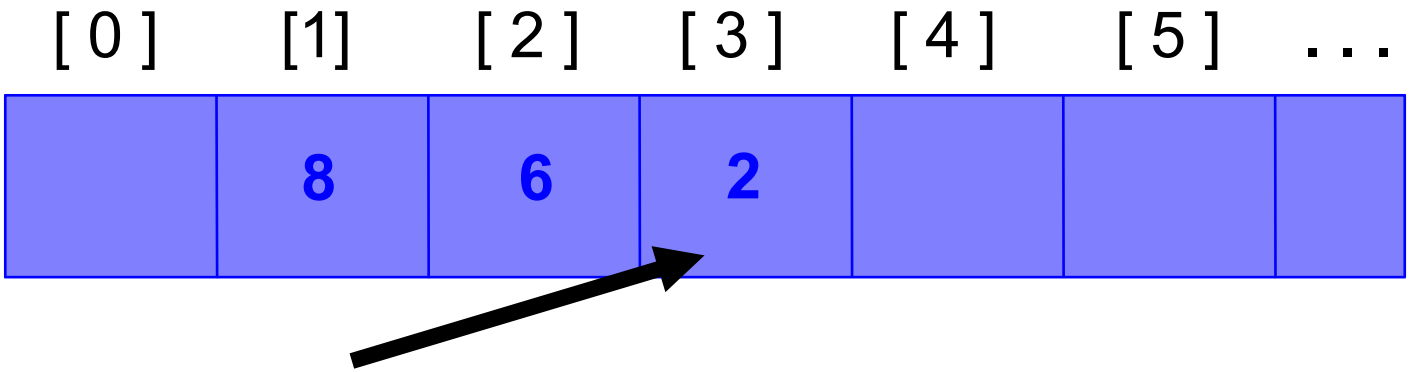


# An Enqueue Operation

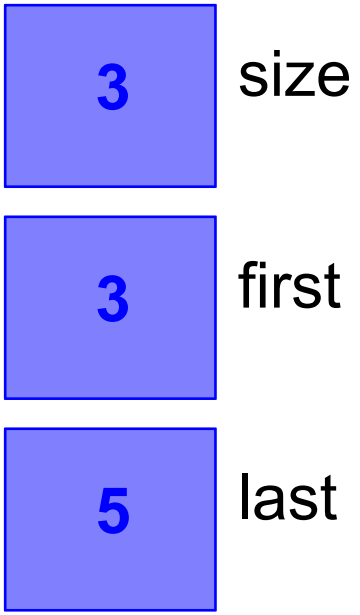
When an element enters the queue, size is incremented, and last changes, too.

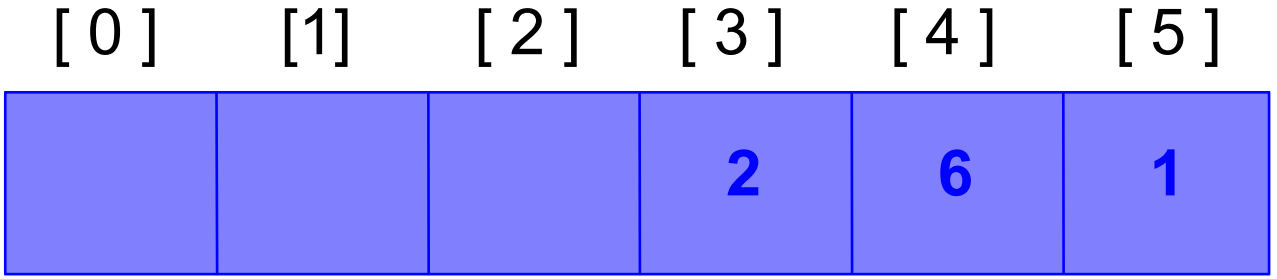


# At the End of the Array

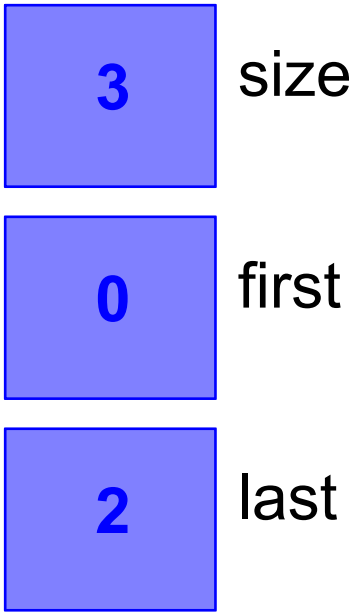
There is special behavior at the end of the array. For example, suppose we want to add a new element to this queue, where the last index is [5]:

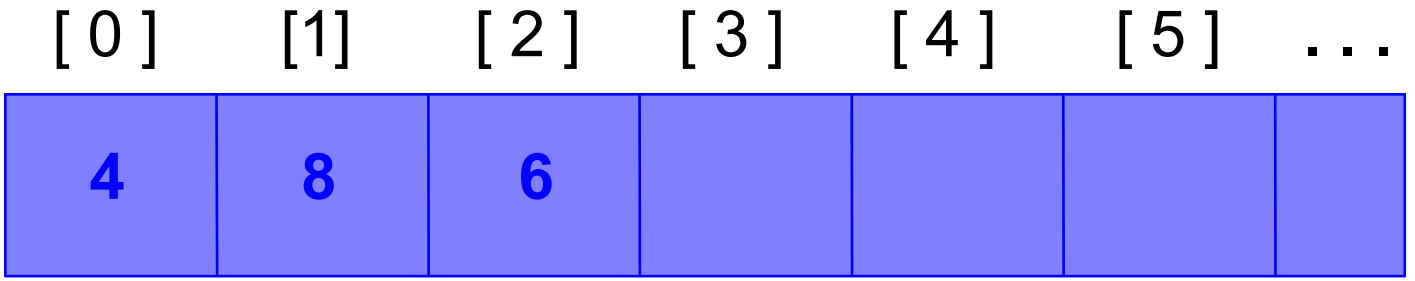


# Array Implementation

- Easy to implement
- But it has a limited capacity with a fixed array
- Or you must use a dynamic array for an unbounded capacity
- Special behavior is needed when the rear reaches the end of the array.



# Queue Full and Empty

Queue Full:

Rear=n (Maxsize of an array)

Queue Empty:

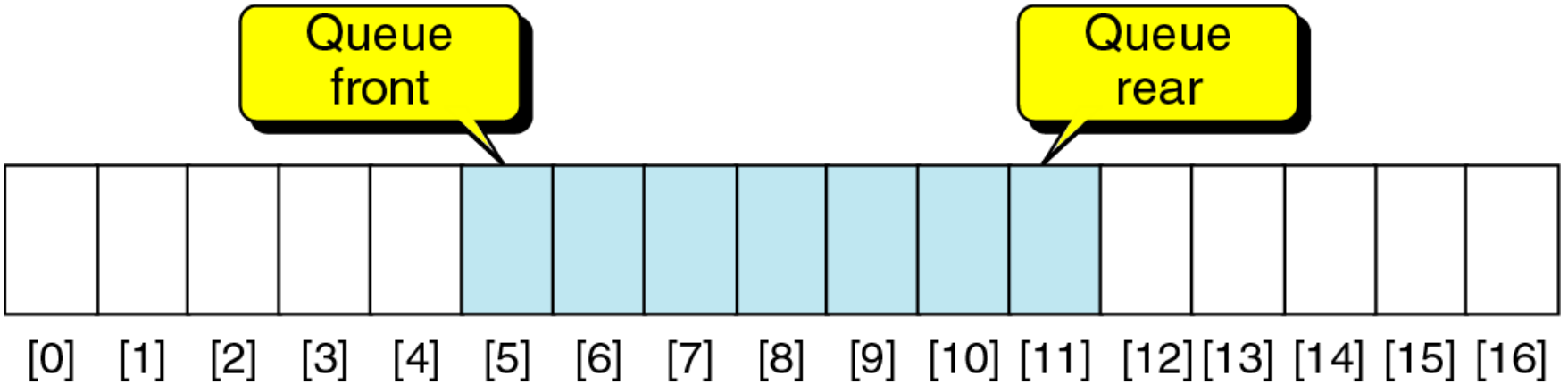
Rear < Front

# Circular Queue

- Problems
  - We quickly "walk off the end" of the array
- Possible solutions
  - Shift array elements
  - Use a circular queue

# Circular Queue

- Use of Linear Array to implement a queue.




- Waste of memory: The deleted elements can not be re-used.

- Solution: to use circular queue.

# Circular Queue Full and Empty

Unfortunately, it is difficult under this representation to determine when the queue is empty.

The condition **rear < front** is no longer a valid test for the empty queue.

One way to solve this problem is to establish the convention that the value of front is the array index **immediately preceding** the first element of the queue rather than the index of the first element itself.

Thus, one element of the array is to be scarified and allow a queue to grow only as large as one less than the size of the array

# Circular Queue Full and Empty

Queue Full:

(rear +1) % n == front

Queue Empty:

front==rear

# Priority Queues

A Special form of queue from which items are removed according to their designated priority and not the order in which they entered.

Two kinds of priority queues:

- Min priority queue.
- Max priority queue.

# Max Priority Queue

- Collection of elements.
- Each element has a priority or key.
- Supports following operations:
  - empty
  - size
  - insert an element into the priority queue (push)
  - get element with max priority (top)
  - remove element with max priority (pop)

# Applications

- Sorting
- use element key as priority
- insert elements to be sorted into a priority queue
- remove/pop elements in priority order
  - if a min priority queue is used, elements are extracted in ascending order of priority (or key)
  - if a max priority queue is used, elements are extracted in descending order of priority (or key)

# Josephus Problem

A set of players are present in a circle.

Counting from a given player, every ‘nth’ player is considered out and eliminated from the game ‘out’

Counting starts again from the next person after the removed player, and the next ‘nth’ player is removed.

The game continues until only one player remains, who is the winner

# Algorithm for Josephus Problem

1. Obtain the initial player list
2. Go to starting player. Start count of 1.
3. Increment count, go to next player. If player-list end is reached, go to list beginning.
4. If count < n, go back to step 3
5. If count = n, remove player from list. Set count = 1 from next player. Go back to Step 3.
6. If next player is same as current player, declare winner.

**Implementation**

- 2D Arrays to hold player names
- Circular lists

**Circular lists** are an easier implementation for Step 3
Step 5: easier with doubly linked circular lists
workaround: eliminate nth player when count = n-1

End of Chapter