Queues

Saikrishna Arcot
M. Hudachek-Buswell

August 18, 2018

Queue

- A queue is a data structure where insertions and deletions follow the FIFO (first-in first-out) scheme. In other words, the first object you added into the queue is the first object removed from the queue.

Queue

- A queue is a data structure where insertions and deletions follow the FIFO (first-in first-out) scheme. In other words, the first object you added into the queue is the first object removed from the queue.
- Insertions are at the rear of the queue and removals are at the front of the queue. You do not add, remove or access from any other point in the queue.

Queue

- A queue is a data structure where insertions and deletions follow the FIFO (first-in first-out) scheme. In other words, the first object you added into the queue is the first object removed from the queue.
- Insertions are at the rear of the queue and removals are at the front of the queue. You do not add, remove or access from any other point in the queue.
- Like stacks, queues are an Abstract Data Type (ADT) with more than one implementation.

Queue

- Main queue operations:

Queue

- Main queue operations:
  - `enqueue(object)`: inserts an element at the end of the queue.

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.

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.
- Auxillary queue operations:

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.
- Auxillary queue operations:
  - `object first()`: returns the element at the front without removing it.

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.
- Auxillary queue operations:
  - `object first()`: returns the element at the front without removing it.
  - `integer size()`: returns the number of elements stored.

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.
- Auxillary queue operations:
  - `object first()`: returns the element at the front without removing it.
  - `integer size()`: returns the number of elements stored.
  - `boolean isEmpty()`: indicates whether no elements are stored.

Example

Operation	Return value	Queue
`enqueue(5)`	-	(5)
`enqueue(3)`	-	(5, 3)
`dequeue()`	5	(3)
`enqueue(7)`	-	(3, 7)
`dequeue()`	3	(7)
`first()`	7	(7)
`dequeue()`	7	()
`dequeue()`	`null`	()
`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-backed Queue

- Use an array of size `N` in a circular fashion for the queue.

Array-backed Queue

- Use an array of size `N` in a circular fashion for the queue.
- Two variables keep track of the front and size:

Array-backed Queue

- Use an array of size `N` in a circular fashion for the queue.
- Two variables keep track of the front and size:
  - `front`: index of the front element

Array-backed Queue

- Use an array of size `N` in a circular fashion for the queue.
- Two variables keep track of the front and size:
  - `front`: index of the front element
  - `size`: number of stored elements

Array-backed Queue

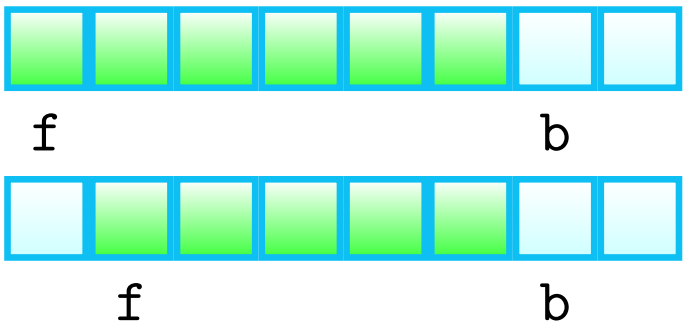
- Use an array of size `N` in a circular fashion for the queue.
- Two variables keep track of the front and size:
  - `front`: index of the front element
  - `size`: number of stored elements
- When the queue has fewer than `N` elements, array location $back = (front + size) \bmod N$ is the first empty slot past the rear of the queue.

Array-backed Queue

Example of a queue implemented with an array using a "wraparound" technique.

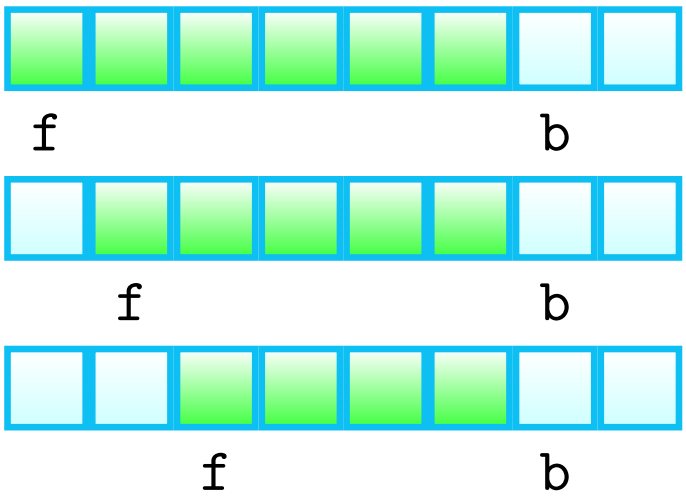
Array-backed Queue

Example of a queue implemented with an array using a "wraparound" technique.



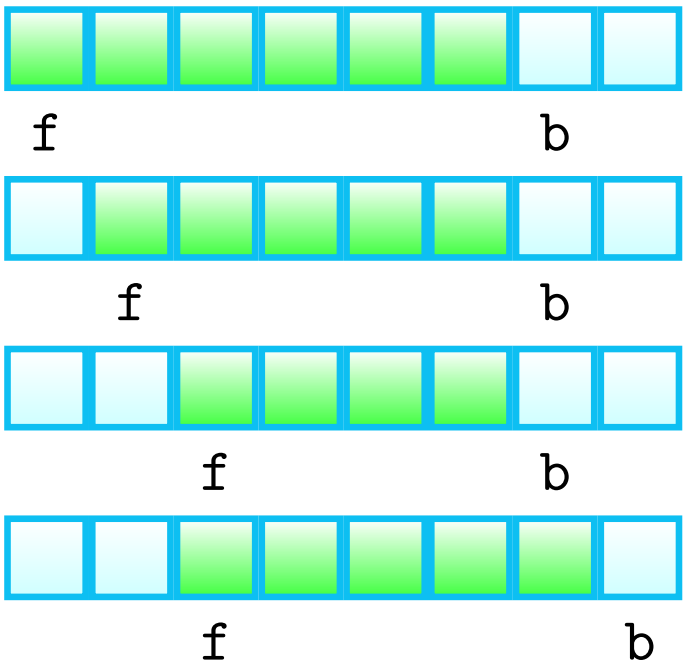
Array-backed Queue

Example of a queue implemented with an array using a "wraparound" technique.



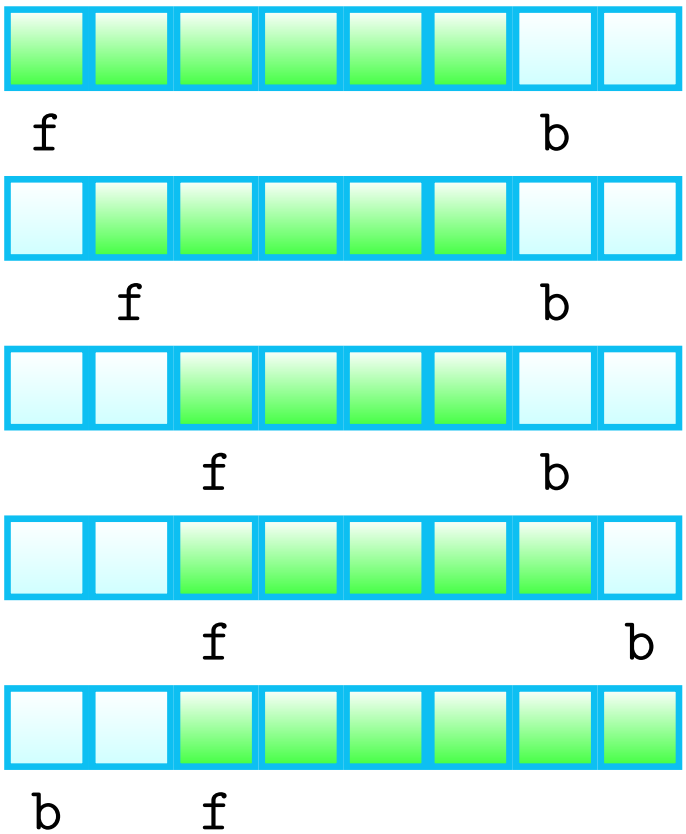
Array-backed Queue

Example of a queue implemented with an array using a "wraparound" technique.



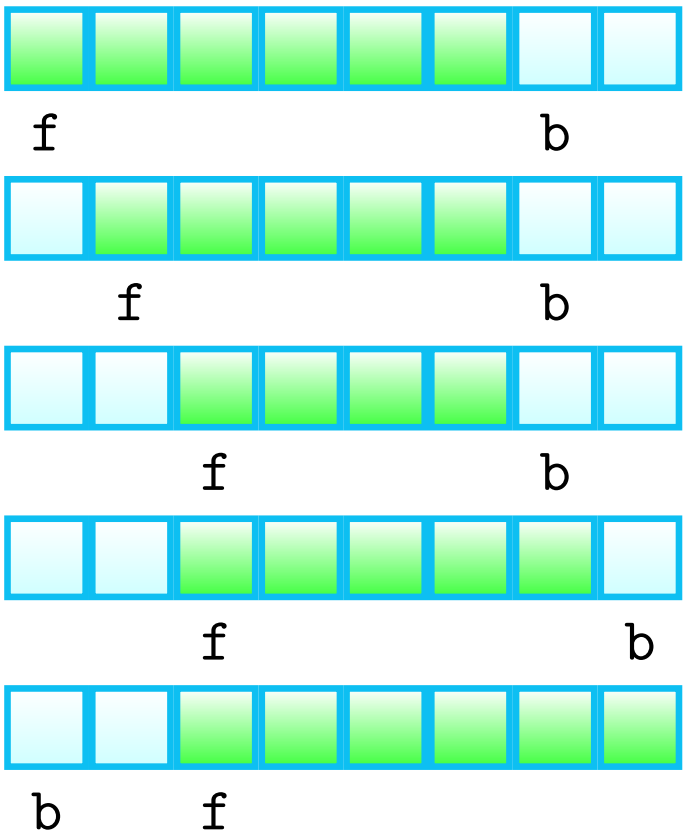
Array-backed Queue

Example of a queue implemented with an array using a "wraparound" technique.



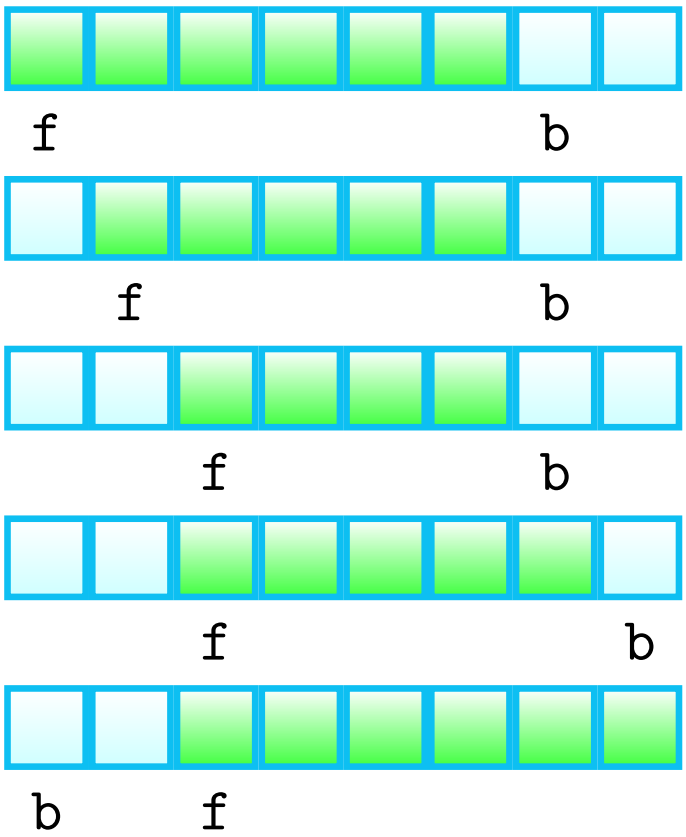
Array-backed Queue

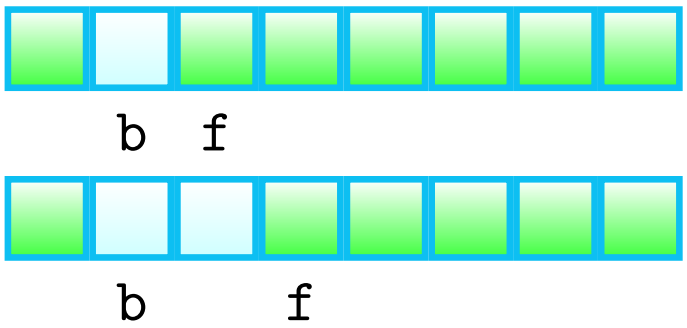
Example of a queue implemented with an array using a "wraparound" technique.



Array-backed Queue

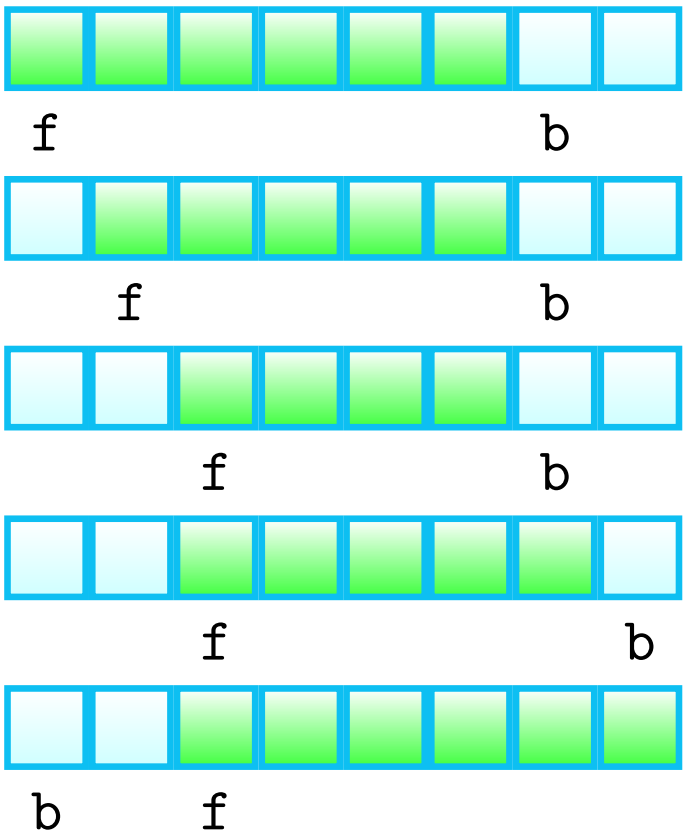
Example of a queue implemented with an array using a "wraparound" technique.

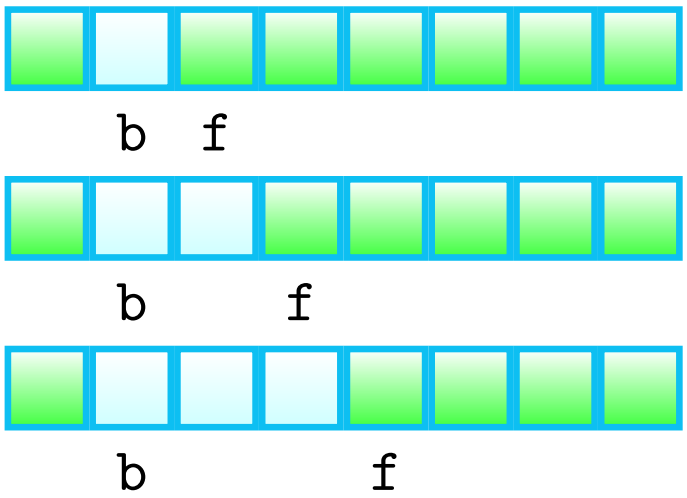




Array-backed Queue

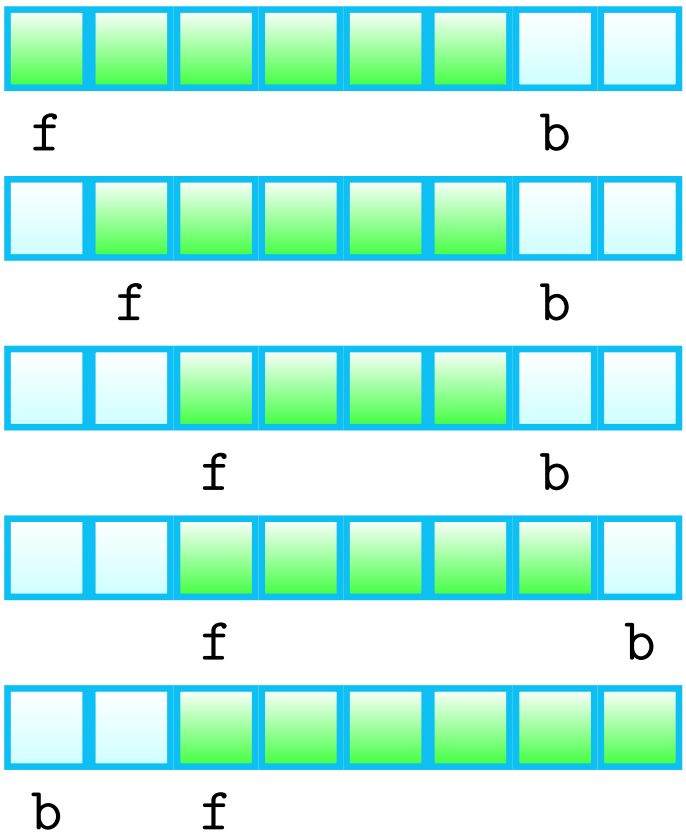
Example of a queue implemented with an array using a "wraparound" technique.

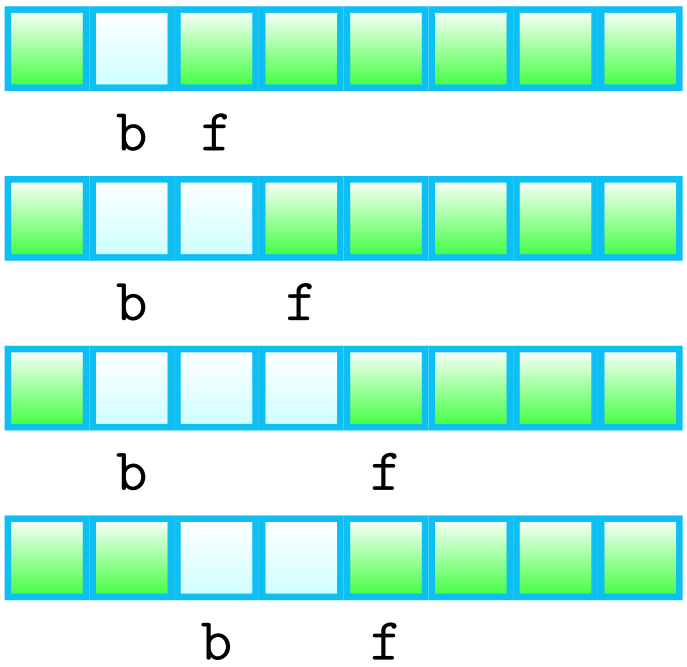




Array-backed Queue

Example of a queue implemented with an array using a "wraparound" technique.





Array-backed Queue

procedure ENQUEUE(o)
    $back \leftarrow (front + size) \bmod N$
    $arr[back] \leftarrow o$
    $size \leftarrow size + 1$
end procedure

Keep in mind that you need to consider the case when the queue is full and how to handle it.

Array-backed Queue

procedure DEQUEUE
    $item \leftarrow arr[f]$
    $front \leftarrow (front + 1) \bmod N$
    $size \leftarrow size - 1$
    return *item*
end procedure

Notice that the cell in the array, $arr[f]$ is not set to *null* after it is dequeued. Consider the case, when the queue is empty and you attempt to dequeue.

Linked List-backed Queue

- A linked list could also be used as the backing data structure of a queue.

Linked List-backed Queue

- A linked list could also be used as the backing data structure of a queue.
- Elements would be added and removed to opposite ends of the list. The most efficient way to do this would depend on the list.

Performance

- Performance

Performance

- Performance
  - Let n be the number of elements in the queue.

Performance

- Performance
  - Let n be the number of elements in the queue.
  - The space used is $O(n)$.

Performance

- Performance
  - Let n be the number of elements in the queue.
  - The space used is $O(n)$.
  - Each operations runs in time $O(1)$.

Limitations

- Limitations

Limitations

- Limitations
  - For an array-backed queue, the initial maximum size of the stack must be defined *a priori*.

Limitations

- Limitations
  - For an array-backed queue, the initial maximum size of the stack must be defined *a priori*.
  - For an array-backed queue, if the backing array is to resized when it becomes full, this would then be an $O(n)$ operation (but the `enqueue()` operation itself would be considered amortized $O(1)$).

Limitations

- Limitations
  - For an array-backed queue, the initial maximum size of the stack must be defined *a priori*.
  - For an array-backed queue, if the backing array is to resized when it becomes full, this would then be an $O(n)$ operation (but the `enqueue()` operation itself would be considered amortized $O(1)$).
    - During the process of resizing, the queue would be "unwound", so that the front of the queue would be in index 0 of the resized array.

Limitations

- Limitations
  - For an array-backed queue, the initial maximum size of the stack must be defined *a priori*.
  - For an array-backed queue, if the backing array is to resized when it becomes full, this would then be an $O(n)$ operation (but the `enqueue()` operation itself would be considered amortized $O(1)$).
    - During the process of resizing, the queue would be "unwound", so that the front of the queue would be in index 0 of the resized array.
  - Linked list-backed queue do not have either of the above two limitations.

Applications of Queues

- Direct applications

Applications of Queues

- Direct applications
  - Waiting lists

Applications of Queues

- Direct applications
  - Waiting lists
  - Access to shared resources (e.g. printer)

Applications of Queues

- Direct applications
  - Waiting lists
  - Access to shared resources (e.g. printer)
  - Multithreading

Applications of Queues

- Direct applications
  - Waiting lists
  - Access to shared resources (e.g. printer)
  - Multithreading
- Indirect applications

Applications of Queues

- Direct applications
  - Waiting lists
  - Access to shared resources (e.g. printer)
  - Multithreading
- Indirect applications
  - Auxillary data structure for algorithms

Applications of Queues

- Direct applications
  - Waiting lists
  - Access to shared resources (e.g. printer)
  - Multithreading
- Indirect applications
  - Auxillary data structure for algorithms
  - Component of other data structures

Round Robin Schedulers

- When you have multiple processes running on your computer, the CPU needs a way of determining in what order to work on each process. A round-robin scheduler is one simple way of doing that.

Round Robin Schedulers

- When you have multiple processes running on your computer, the CPU needs a way of determining in what order to work on each process. A round-robin scheduler is one simple way of doing that.
- We can implement a round robin scheduler using a queue `processes` by repeatedly performing the following steps:

Round Robin Schedulers

- When you have multiple processes running on your computer, the CPU needs a way of determining in what order to work on each process. A round-robin scheduler is one simple way of doing that.
- We can implement a round robin scheduler using a queue `processes` by repeatedly performing the following steps:
  1. `process = processes.dequeue()`

Round Robin Schedulers

- When you have multiple processes running on your computer, the CPU needs a way of determining in what order to work on each process. A round-robin scheduler is one simple way of doing that.
- We can implement a round robin scheduler using a queue `processes` by repeatedly performing the following steps:
  1. `process = processes.dequeue()`
  2. Do work on `process`

Round Robin Schedulers

- When you have multiple processes running on your computer, the CPU needs a way of determining in what order to work on each process. A round-robin scheduler is one simple way of doing that.
- We can implement a round robin scheduler using a queue `processes` by repeatedly performing the following steps:
  1. `process = processes.dequeue()`
  2. Do work on `process`
  3. `processes.enqueue(process)`