**The Queue ADT**

* Queues are known as FIFO (first in, first out) lists.
* Like stacks, **queues** are lists.
  + A **stack** is a list with the restriction that insertions and deletions can be performed at the **end of the list**, called the **top**.
  + With a **queue**, however, **insertion is done at one front**, whereas **deletion is performed at the end**.

## **Queue Model**

* The two most basic operations on a queue are
  + **enqueue** - insert an element at the end of the list (rear)
  + **dequeue** - deletes (and returns) the element at the start of the list (front)
* The following figure shows the abstract model of a queue.

Application

Description automatically generated with low confidence

* As with stacks, both an array and linked list implementation of a queue gives **O(1) runtimes for every operation**.

# **Array Implementation of Queues**

**Queue** **Variables**

1. **theArray** – an internal array to hold objects
2. **rear** – a pointer to the end of the queue
3. **size** – the number of elements in the queue

**Queue Operations**

1. enqueue (push) - Pushes an item onto the top of the stack.
2. dequeue (pop) - Removes and returns the object at the top of the stack.
3. isEmpty - Tests if this stack is empty.
4. peek - returns the object at the top of this stack without removing it.

**Queue Operations**

1. enqueue (push) - O(1)
2. dequeue (pop) - O(N)
3. isEmpty - O(1)
4. peek - O(1)

**Enqueue**

To enqueue an element x:

1. set theArray[rear] = x,
2. increment rear,
3. increment size,

**// Basic idea only!**

**enqueue(x)**

**{**

**theArray[back] = x;**

**rear++;**

**size++;**

**}**

**Dequeue**

To dequeue an element x:

1. save front value theArray[front] to temp variable,
2. shift all elements left one index,
3. decrement rear,
4. decrement size,
5. return theArray[front]

**// Basic idea only!**

**dequeue()**

**{**

**T x = theArray[front];**

**for(int i = 0; i < rear; i++)**

**queue[i] = queue[i+1];**

**queue[rear] = null;**

**rear--;**

**size--;**

**return x;**

**}**

What’s missing here?

What can be improved?

What is the runtime?

# **Circular Array Implementation of Queues**

**Queue** **Variables**

1. **theArray** – an internal array to hold objects
2. **front** – a pointer to the front of the queue
3. **back** – a pointer to the end of the queue
4. **size** – the number of elements in the queue

**Queue Operations**

1. enqueue (push) - Pushes an item onto the top of the stack.
2. dequeue (pop) - Removes and returns the object at the top of the stack.
3. isEmpty - Tests if this stack is empty.
4. peek - returns the object at the top of this stack without removing it.

**Queue Operations**

1. enqueue (push) - O(1)
2. dequeue (pop) - O(1)
3. isEmpty - O(1)
4. peek - O(1)

**Enqueue Improved**

What if ***queue*** is full?

To enqueue an element x:

1. check if queue is full,
2. set theArray[back] = x,
3. increment back,
4. increment size,

**public void enqueue (T x)**

**{**

**if (size() == queue.length) expandCapacity();**

**theArray[back] = x;**

**back = (back+1) % theArray.length;**

**size++;**

**}**

**Dequeue Improved**

What if ***queue*** is empty?

To dequeue an element x:

1. check if queue is empty,
2. save front value theArray[front] to temp variable,
3. set old theArray[front] value to null
4. increment front,
5. decrement currentSize,
6. return theArray[front]

**public T dequeue() throws EmptyCollectionException**

**{**

**if (isEmpty()) throw new EmptyCollectionException ("queue is empty");**

**T x = theArray[front];**

**theArray[front] = null; // set old value to null**

**front = (front+1) % theArray.length;**

**count--;**

**return result;**

**}**

**Example**

Here is how **theArray** would look after **7 enqueues** and **3 dequeues**:

Table

Description automatically generated

**Problem**

* There is one potential problem with this implementation.
* After 10 **enqueues**, the queue ***appears*** to be full, since **back** is now at the last array index, and the next **enqueue** would be in a nonexistent position (out of bounds index).
* However, there might only be a few elements in the queue because several elements may have already been **dequeued**.
* Therefore, **back** is at the end of the array (because there has been a total of 10 **enqueues**), yet the entire array is not filled (because there have been several **dequeues**).
* We can’t simply increase the size of the array, because we would otherwise be wasting space, and continuing to create a bigger and bigger hole at the front of the array.
* In addition, Queues, like Stacks, frequently stay small even in the presence of a lot of operations. We can take advantage of this fact.

**Wraparound**

* Therefore, whenever **front** or **back** gets to the end of the array, it is **wrapped around to the beginning**.
* If incrementing either **back** or **front** causes it to go past the array, the value is reset to the **first position** in the array.

**front = (front+1) % theArray.length;**

**back = (back+1) % theArray.length;**

* This is known as a **circular array** implementation.
* The following figures show the queue during some operations.

Table

Description automatically generated

Diagram

Description automatically generated

**Checking if The Queue is Empty**

* Some programmers use different ways of representing the **front** and **back** of a queue.
* For instance, some do not use an entry to keep track of the **size**, because they rely on the base case that when the queue is empty, back = front-1.
* The size is computed implicitly by comparing back and front.
* This is a very tricky way to go, because there are some special cases, so be very careful if you need to modify code written this way.
* If the **size** is not maintained as an explicit data field, then the queue is full when there are **theArray.length-1** elements, since only theArray.length different sizes can be differentiated, and one of these is 0.
* Pick any style you like and make sure that all your routines are consistent. Since there are a few options for implementation, it is probably worth a comment or two in the code, if you don’t use the currentSize field.

# **Linked List Implementation of Queues**

**Queue** **Variables**

1. **theArray** – an internal array to hold objects
2. **front** – a pointer to the front of the queue
3. **back** – a pointer to the end of the queue
4. **size** – the number of elements in the queue

**Queue Operations**

1. enqueue (push) - Pushes an item onto the top of the stack.
2. dequeue (pop) - Removes and returns the object at the top of the stack.
3. isEmpty - Tests if this stack is empty.
4. peek - returns the object at the top of this stack without removing it.

**Queue Operations**

1. enqueue (push) - O(1)
2. dequeue (pop) - O(1)
3. isEmpty - O(1)
4. peek - O(1)

# **Array vs. Linked List Implementation of Queues**

Array:

* May waste unneeded space or run out of space
* Space constants per element/object is small
* Operations are simple/fast

Operations not in Queue ADT

* Constant-time O(1) “access to kth element”
* For operation “insertAtPosition” must shift all later elements

List:

* Always just enough space (we do not overestimate or waste unnecessary space or run out of space)
* Much higher space constant per element/object
* Operations are simple/fast

Operations not in Queue ADT

* No constant-time “access to kth element”. O(i)
* For operation “insertAtPosition” must traverse all earlier elements