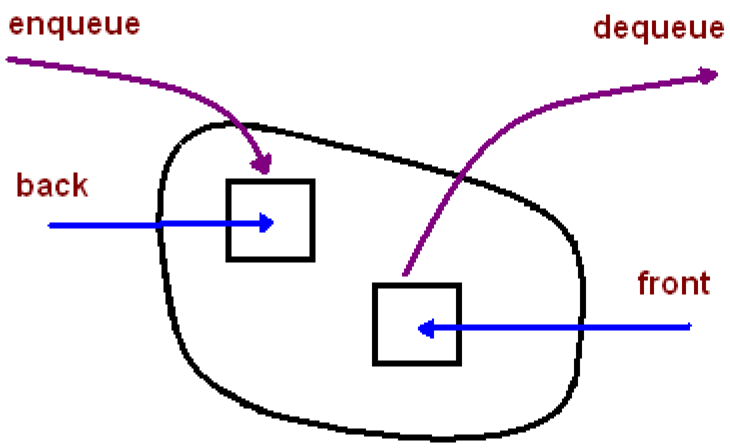
Queues

Table of contents

1. Introduction
2. Queue applications
3. Queue implementation
4. Array based implementation
5. Linked list based implementation

Introduction

After the stack, the next interesting ADT is the queue. Just like a stack, a queue is also a restricted ordered sequence, with the difference that we can only add to one end — the **back** or **rear**, and remove from the other end — the **front** of the queue. You stand in a queue to get on a bus, where the earlier you arrive, the earlier you get on the bus; you stand in a queue to get service at a bank, where the teller services the front of the queue next. You are not, strictly speaking, allowed to add to the middle of the queue, nor are you allowed to remove an item from the middle of the queue. The only item that can be taken out of the queue is the one that was added the **earliest**; a queue is thus a "first in, first out" (FIFO) data structure. The following figure shows the FIFO nature of a queue.



Three basic queue operations are:

- **`enqueue(obj)`**: adds **`obj`** to the back of the queue ("overflow" error if the queue has fixed capacity, and is full)
- **`dequeue`**: removes and returns the item at the front of the queue ("underflow" error if the queue is empty)
- **`peek`**: returns the item that is at the front of the queue, but does not remove it ("underflow" error if the queue is empty)

Back to Table of contents

Queue applications

- **Palindrome**: A simple application is to check whether a string (or any sequence) is a palindrome or not. A *palindrome* is a sequence of symbols that reads the same backward or forward. Examples include:
    - radar
    - 31413
    - Able was I ere I saw Elba
    - GCATTACG
    - A man, a plan, a canal - Panama

  (Sometimes we ignore case. Sometimes we ignore spaces. Sometimes we ignore non-alphabetic characters.)

  How can we tell if a sequence of characters is a palindrome? We can use a stack and a queue. For each character in the sequence, we push it on the stack and place it on the back of the queue. When we have finished reading the sequence, we pop the stack and remove the front item from the queue, checking to see if the characters match. If so, we repeat until the stack and queue are empty (in which case the sequence is a palindrome) or until the popped and dequeued characters differ (in which case the sequence is not a palindrome).

  As an example, consider the word "radar":

```
|r|<-- top of stack
|a|
|d|                   -----
|a|                   radar
|r|                   -----
---                   ^    ^
                   back    front
```

  The algorithm may be written as the following:

```
begin with an empty queue, an empty stack, and an input string;
for each character in the input string, do:
   if the character is a letter
      then enqueue it into the queue;
           push it onto the stack;
end for;
while the queue is not empty, do:
   c1 = dequeue the queue;
   c2 = pop the queue;
   if c1 <> c2
      then return false;
end while;
// All the characters must have matched, so must be a palindrome
return true;
```

Note that you need to skip over the *non-letters*, and ignore the case as well. You can use the static methods `Character.isLetter` and `Character.toLowerCase` do accomplish that quite trivially.

- **Finding your way out of a maze**: Remember how we used a stack to find our way out of a stack (using a technique known as **backtracking**)? We can also use a queue, which changes the sequence of steps we take, but the end result is the same.

  Here is the algorithm to do just that.

```
Create an empty queue that holds the places you need to visit;
Put the starting point in this queue;
While the queue is not empty, do:
   Remove a place from the queue;
   If it is the finish point
       then we are done;
   ElseIf you have already visited this square
       then ignore it;
   Else
       mark the place as "visited";
       add all the neighbors of this place to your queue:
end for;
// If you eventually reach an empty queue and have not found the exit,
// there is no solution.
```

Back to Table of contents

Queue implementation

In the standard library of classes, the data type queue is an *adapter* class, meaning that a queue is built on top of other data structures. The underlying structure for a queue could be an array, a vector, an ArrayList, a linked list, or any other sequence (`Collection` class in JDK). Regardless of the type of the underlying data structure, a `Queue` must implement the same functionality. This is achieved by providing an interface:

```
public interface Queue {
    // The number of items in the queue
    int size();
    // Returns true if the queue is empty
    boolean isEmpty();
    // Adds the new item on the back of the queue, throwing the
    // QueueOverflowException if the queue is at maximum capacity. It
    // does not throw an exception for an "unbounded" queue, which
    // dynamically adjusts capacity as needed.
    void enqueue(Object o) throws QueueOverflowException;
    // Removes the item in the front of the queue, throwing the
    // QueueUnderflowException if the queue is empty.
    Object dequeue() throws QueueUnderflowException;
    // Peeks at the item in the front of the queue, throwing
    // QueueUnderflowException if the queue is empty.
    Object peek() throws QueueUnderflowException;
    // Returns a textual representation of items in the queue, in the
    // format "[ x y z ]", where x and z are items in the front and
    // back of the queue respectively.
    String toString();
    // Returns an array with items in the queue, with the item in the
    // front of the queue in the first slot, and back in the last slot.
    Object[] toArray();
    // Searches for the given item in the queue, returning the
    // offset from the front of the queue if item is found, or -1
    // otherwise.
    int search(Object o);
}
```

One requirement of a `Queue` implementation is that the `enqueue` and `dequeue` operations run in *constant time*, that is, the time taken for queue operation is independent of how big or small the queue is.

Back to Table of contents

Array based implementation

In an array-based implementation we add the new item being enqueued at the end of the array (constant cost), and consequently dequeue the item in the front the queue from the beginning of the array (linear cost, since we have to shift all the item to fill the hole). Or, we add the new item being enqueued in the beginning of the array (linear cost for shifting), and consequently dequeue the item in the front the queue from the end of the array (constant cost). Neither is a *win-win* situation, so we have to accept the loss in either enqueue or dequeue.

Let's choose the 1st option, in which the front item is always at index `0`, and where we shift after each `dequeue` operation. In this scheme, the front of the queue is always at index `0`, and rear one is at `size - 1`, where `size` denotes the number of items in the queue. A partial implementation is shown below.

```
public class LinearArrayQueue implements Queue {
    private Object[] data;      // The array container
    private int      size;      // The number of items in the queue

    // Default initial capacity, which may then dynamically change
    private static final int DEF_INIT_CAPACITY = 100;

    public LinearArrayQueue() {
        data = new Object[DEF_INIT_CAPACITY];
        size = 0;
    }
}
```

The following show a queue through a sequence of queue operations using a circular array. Note that we don't need to keep a separate `rear` variable since it's equal to `size - 1`.

```
       0    1    2    3    4     5
     ---------------------------------------
     |   |    |    |    |   |    |   |   |           size = 0
     ---------------------------------------
     (front, back undefined for an empty queue)

Let's enqueue the values 5 7 9 15 -4 and 17. The result is:

       0    1    2    3    4     5
     ---------------------------------------
     | 5 |  7 |  9 |  15 | -4 |  17 |   |   |          size = 6
     ---------------------------------------
       ^-- front                  ^-- rear

Dequeuing an item results in the following:

       0    1    2    3    4     5
     ---------------------------------------
     |  7 |  9 |  15 | -4 |  17 |   |   |   |          size = 5
     ---------------------------------------
       ^-- front             ^-- rear

Dequeuing again:

       0    1    2    3    4     5
     ---------------------------------------
     | 9 | 15 | -4 |  17 |    |     |   |   |          size = 4
     ---------------------------------------
       ^-- front      ^-- rear
```

`And so on.`

If we choose not to shift, but we may seem to have run out of space at the end, even if multiple dequeue operations have created space in the beginning of the array. Again, we have to shift the items to create space at the end, where we lose again.

The solution is to use a **circular** array instead of a **linear** array. See the notes on circular array for details on how it works, and why it provides a *win-win* situation when implementing a queue.

For a queue using a circular array as the container, the only complications are that we have to keep track of the index of the front item (which is now not necessarily at index `0`), and we have to **wrap** around the underlying array container when we get to the end of it. If **front** is the index of the front item, the index of the rear item is given by `(front + size - 1) % capacity`, where `capacity` is the length of the built-in array where we keep the actual queue. As in the case with the **linear** array, we add new items to the **rear** of the circular array, and remove old items from the **front** of the circular array. The index of the next available slot is then `(front + size) % capacity`.

In a *bounded* or fixed-size queue abstraction, the capacity stays unchanged, therefore when *rear* reaches *capacity*, the queue object throws an exception.

In an *unbounded* or dynamic queue abstraction when *rear* reaches *capacity*, we double up the queue size.

The following shows a partial linear array-based implementation of an *unbounded* queue.

```
public class ArrayQueue implements Queue {
    private Object[] data;      // The array container
    private int      front;     // The index of the front item
    private int      size;      // The number of items in the queue

    // Default initial capacity, which may then dynamically change
    private static final int DEF_INIT_CAPACITY = 100;

    public ArrayQueue() {
        data = new Object[DEF_INIT_CAPACITY];
        front = 0;
        size = 0;
    }

    // The rest of the implementation goes here...
}
```

The following show a queue through a sequence of queue operations using a circular array. Note that we don't need to keep a separate `rear` variable since it's equal to `(front + size - 1) % data.length`. And the next available position is simply `(front + size) % data.length`.

```
       0    1    2   3   4   5    6    7   8   9
```

```
     ----------------------------------------------
     |    |    |    |     |    |     |   |   |    |      size = 0
     ----------------------------------------------
     (front, rear undefined for an empty queue)

If we enqueue the elements 10, 20, 30, 40 and 50, we get:

       0    1    2   3   4   5    6    7   8   9
     ----------------------------------------------
     | 10 | 20 | 30 | 40  | 50 |     |   |   |    |      size = 5
     ----------------------------------------------
       ^-- front             ^-- rear

Now if we dequeue one element, we get:

       0    1    2   3   4   5    6    7   8   9
     ----------------------------------------------
     |    | 20 | 30 | 40  | 50 |     |   |   |    |      size = 4
     ----------------------------------------------
             ^-- front      ^-- rear

Now if we dequeue all four remaining elements, we get:

       0    1    2   3   4   5    6    7   8   9
     ----------------------------------------------
     |    |    |    |     |    |     |   |   |    |      size = 0
     ----------------------------------------------
     (front, rear undefined for an empty queue)

Now let's enqueue elements 5, 7, 9, 15, -4 and 17.

       0    1    2   3   4   5    6    7   8   9
     ----------------------------------------------
     | -4 | 17 |    |     |    |  5  | 7 | 9 | 15 |      size = 6
     ----------------------------------------------
            ^--rear               ^--front
```

To iterate over the items, we start at the front element and go to the rear, but unlike in the case of linear array, we may have to "wrap around" when we get to the end. It's often easiest to use two different indices to iterate through a cyclic array. For example:

```
int j = front;
for (int i = 0; i < size; i++) {
    visit(data[j]);
    j = (j + 1) % data.length;
}
```

Note how we're using `i` to count the number of times we need to iterate, and `j` to actually index into the underlying array. The variable `data` is a reference to the underlying array, which means that capacity is `data.length`, and the variable variable `size ≤ data.length` is the number of items in the queue.

If we wanted to iterate backwards (ie., from the rear end of the queue to the front), then:

```
int j = (front + size - 1) % data.length;  // index of item in back
for (int i = size-1; i >= 0; i--) {
    visit(data[j]);
    j--;
    if (j == -1)
        j = data.length - 1;
}
```

Note that we can't use modulus operation when iterating backwards, but must explicitly set the index to the point to the last slot in the array container once we "fall off" the front.

Resizing a cyclic array means that we have to do the following:

1. Allocate a new array of the desired capacity
2. Copy the elements from the old cyclic array container to the new one
3. Reset the front index to `0`.

Here's an example: let's start with the following queue:

```
  0    1    2   3   4   5    6    7   8   9
----------------------------------------------
| -4 | 17 |    |     |    |  5  | 7 | 9 | 15 |      size = 6
----------------------------------------------  capacity = 10
       ^--rear             ^--front
```

```
Now let's resize it to have a capacity of 11.

Here's the WRONG final answer (WHY?):
```

```
  0    1    2   3   4   5    6    7   8   9   10
--------------------------------------------------
| -4 | 17 |    |     |    |  5  | 7 | 9 | 15 |   |      size = 6
--------------------------------------------------  capacity = 11
       ^--rear             ^--front
```

```
Here's one possible correct solution:
```

```
  0    1    2   3   4   5    6    7   8   9   10
--------------------------------------------------
| 5  | 7  | 9  | 15  | -4 |  17 |   |   |    |   |      size = 6
--------------------------------------------------  capacity = 11
  ^-- front                 ^--rear
```

is not really that tricky - it's actually quite easy if you use two different indices. One index goes from `front` to `front + size - 1`, and other one from `0` to `size - 1`. Again, see the notes on Circular Array available from Moodle course site to see how it all works.

The moral of the story is this: we use a circular array to avoid shifting items when we dequeue. The cost is that we have to maintain the position or index of the item in the front of the queue; give the number of items, we can then compute the position of the other items, and the position of the next available slot.

Back to Table of contents

Linked list based implementation

Linked List-based implementation provides the best (from the efficiency point of view) dynamic queue implementation.

The following shows a partial head-referenced singly-linked based implementation of an *unbounded* queue. In an singly-linked list-based implementation we add the new item being added at the end of the array, using a tail reference to append at constant cost (why?), and consequently remove the front item from the beginning of the list, again at constant cost.

```
public class ListQueue implements Queue {
    private Node head;          // Reference to the front of the queue
    private int  size;          // The number of items in the queue

    public ListQueue() {
        head = null;
        size = 0;
    }

    // The rest of the implementation goes here...
}
```

Back to Table of contents