**Applications of Queue Data Structure**

[Queue](http://en.wikipedia.org/wiki/Queue_%28data_structure%29)is used when things don’t have to be processed immediatly, but have to be processed in **F**irst **I**n **F**irst **O**ut order like [Breadth First Search](http://en.wikipedia.org/wiki/Breadth-first_search). This property of Queue makes it also useful in following kind of scenarios.

**1)** When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.  
**2)**When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

# Find the largest multiple of 3

Given an array of non-negative integers. Find the largest multiple of 3 that can be formed from array elements.

For example, if the input array is {8, 1, 9}, the output should be “9 8 1″, and if the input array is {8, 1, 7, 6, 0}, output should be “8 7 6 0″

As this is the array containing single digit, we can maintain count in the array of 10 (in order to sort).

Let say r=(sum of all the elements)%3

* If r=0, dump all the elements in the descending order.
* If r=1, decrease count to 1 for elements having mod 3 is 1. (e.g. 1,4,7). If count is 0 for all these elements then decrease count to 2 of elements having mod 3 is 2. (e.g. 2,5,8). If it was not possible then ans is not possible. Else dump all the elements in the decreasing order with count.
* If r=2, decrease count to 1 for elements having mod 3 is 1. (e.g. 2,5,8). If count is 0 for all these elements then decrease count to 1 of elements having mod 3 is 2. (e.g. 1,4,7). If it was not possible then ans is not possible. Elese dump all the elements in the decreasing order with count.

# Implement Queue using Stacks

**Method 1 (By making enQueue operation costly)**

This method makes sure that oldest entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

enQueue(q, x)

1) While stack1 is not empty, push everything from satck1 to stack2.

2) Push x to stack1 (assuming size of stacks is unlimited).

3) Push everything back to stack1.

dnQueue(q)

1) If stack1 is empty then error

2) Pop an item from stack1 and return it

**Method 2 (By making deQueue operation costly)**

In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

enQueue(q, x)

1) Push x to stack1 (assuming size of stacks is unlimited).

deQueue(q)

1) If both stacks are empty then error.

2) If stack2 is empty

While stack1 is not empty, push everything from satck1 to stack2.

3) Pop the element from stack2 and return it.

**TC** : O(1) (enqueue is just pushing so O(1) and dequeue is one time popping from stack1, other calls will take O(1) for popping from stack2 so amortized O(1))

**Queue can also be implemented using one user stack and one Function Call Stack.**

Below is modified Method 2 where recursion (or Function Call Stack) is used to implement queue using only one user defined stack.

enQueue(x)

1) Push x to stack1.

deQueue:

1) If stack1 is empty then error.

2) If stack1 has only one element then return it.

3) Recursively pop everything from the stack1, store the popped item

in a variable res, push the res back to stack1 and return res

# An Interesting Method to Generate Binary Numbers from 1 to n

Given a number n, write a function that generates and prints all binary numbers with decimal values from 1 to n.

Examples:

Input: n = 2

Output: 1, 10

Input: n = 5

Output: 1, 10, 11, 100, 101

A simple method is to run a loop from 1 to n, call decimal to binary inside the loop.

|  |
| --- |
| 1) Create an empty queue of strings |
| 2) Enqueue the first binary number “1” to queue. |
| 3) Now run a loop for generating and printing n binary numbers. |
| ……a) Dequeue and Print the front of queue. |
| ……b) Append “0” at the end of front item and enqueue it. |
| ……c) Append “1” at the end of front item and enqueue it. |

# Minimum time required to rot all oranges

Given a matrix of dimension m\*n where each cell in the matrix can have values 0, 1 or 2 which has the following meaning:

0: Empty cell

1: Cells have fresh oranges

2: Cells have rotten oranges

So we have to determine what is the minimum time required so that all the oranges become rotten. A rotten orange at index [i,j] can rot other fresh orange at indexes [i-1,j], [i+1,j], [i,j-1], [i,j+1] (up, down, left and right). If it is impossible to rot every orange then simply return -1.

Examples:

Input: arr[][C] = { {2, 1, 0, 2, 1},

{1, 0, 1, 2, 1},

{1, 0, 0, 2, 1}};

Output:

All oranges can become rotten in 2 time frames.

Input: arr[][C] = { {2, 1, 0, 2, 1},

{0, 0, 1, 2, 1},

{1, 0, 0, 2, 1}};

Output:

All oranges cannot be rotten.

The idea is to user Breadth First Search. Below is algorithm.

1) Create an empty Q.

2) Find all rotten oranges and enqueue them to Q. Also enqueue

a delimiter to indicate beginning of next time frame.

3) While Q is not empty do following

3.a) While delimiter in Q is not reached

(i) Dequeue an orange from queue, rot all adjacent oranges.

While rotting the adjacents, make sure that time frame

is incremented only once. And time frame is not icnremented

if there are no adjacent oranges.

3.b) Dequeue the old delimiter and enqueue a new delimiter. The

oranges rotten in previous time frame lie between the two

delimiters.