# Lab Tutorial for Week 4: Queue Implementation

## Contents

1. Tutorial w4a: Array-based Queue
2. Tutorial w4b: Linked-based Queue (optional)
3. Tutorial w4c: Queue Application (optional)

## Tutorial w4a: Array-based Queue

### A queue is a FIFO (First In First Out) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the REAR and removed from the other end called the FRONT.

### insert (enqueue) operation adds an element to the REAR of the queue

### delete (dequeue) operation removes the element from the FRONT of the queue.

### peek operation returns the value of the FRONT element of the queeu.

### Queues can be easily represented using arrays.

### Practice:

1. Open the starter file (queue\_array.cpp) in your IDE.
2. This file contains the array implementation of Queue, but the implementation of insert (enqueue) and delete (dequeue) operation are missing.

class Queue

{

private:

int queue[MAX];

int front = -1

int rear = -1;

public:

void insert(int);

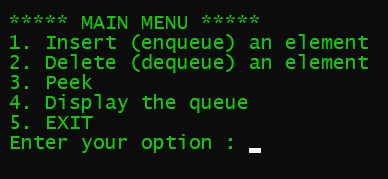
int delete\_element();

int peek();

void display();

};

1. Your task is to complete these two operations. (*see the hints at the back pages of this tutorial*)
2. Save the completed source file and make sure it’s compiled. When you run it, you will have the following prompt:



1. Test your Queue with the following scenario:
   1. Insert data into the Queue until the Queue is full. What would happen with your program?
   2. Delete the data in the Queue until the Queue is empty. What would happen with your program?

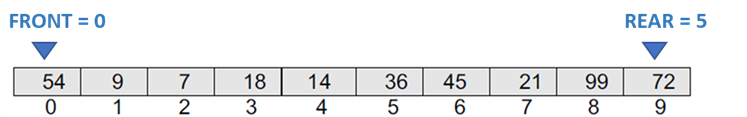
### Questions:

1. What are the strengths and weaknesses of implementing Queue using array?
2. In which situation array implementation of Queue is preferred?
3. How do you think about alleviating the problem of inserting a new element in a full queue?

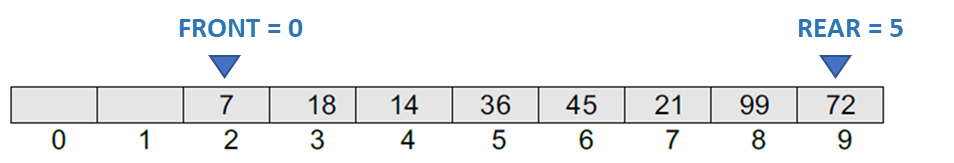
### Discussions:

Using normal (linear) array to implement a queue may have drawbacks in the following scenarios.

Scenario 1: What is the queue is full? Of course, if you want to insert another value, it will not be possible. There is no empty space where the value can be inserted.



Scenario 2: Consider a scenario in which two successive deletions are made, then suppose we want to insert a new element in the queue.



Notice that even though there is space available, the overflow condition still exists because the condition rear = MAX – 1 still holds true.

How would you resolve this problem?

*HINTS: using a circular buffer or circular array.*

## Tutorial w4b: Linked-based Queue (OPTIONAL)

### In the tutorial 4a, you might have noticed the major drawbacks of array-based queue.

### In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element. The START pointer of the linked list is used as FRONT. Here, we will also use another pointer called REAR, which will store the address of the last element in the queue.

### All insertions will be done at the rear end and all the deletions will be done at the front end. If FRONT = REAR = NULL, then it indicates that the queue is empty..

### Practice:

1. Open the starter file (linked\_queue.cpp) in your IDE.
2. This file contains the linked implementation of Queue, but the implementation of insert (enqueue) and delete (dequeue) operation are missing.

struct node

{

int data;

node\* next;

};

struct queue

{

node\* front;

node\* rear;

};

class Queue

{

public:

queue\* q;

Queue();

void create\_queue();

queue\* insert(int);

queue\* delete\_element();

queue\* display();

int peek();

};

1. Notice that we we declared the q data member (the variable) of the Queue class public to have a convenient access instead of using a getter method.
2. Your task is to complete these two operations. (*see the hints at the back pages of this tutorial*)
3. Save the completed source file and make sure it’s compiled.
4. Test your Queue with the following scenario:
   1. Insert data into the Queue until the Queue is full. What would happen with your program?
   2. Delete the data in the Queue until the Queue is empty. What would happen with your program?

### Questions:

1. What are the strengths and weaknesses of implementing Queue using linked-list?
2. In which situation array implementation of Queue is preferred than linked-list one?

## Tutorial w4c: Queue Application (Optional)

### In this section we will practice how to apply queue in typical problems where queue can be easily applied for a simple and efficient solution, for example:

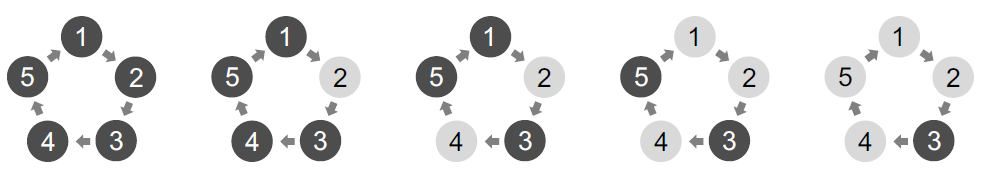
* Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
* Queues are used to transfer data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., pipes, file IO, sockets.
* Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
* Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.
* Queues are used in operating system for handling interrupts. When programming a real-time system that can be interrupted, for example, by a mouse click, it is necessary to process the interrupts immediately, before proceeding with the current job. If the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriate data structure.

Let’s use our Queue to solve a well-known problem called Josephus Problem.

In Josephus problem, *n* people stand in a circle waiting to be executed. The counting starts at some point in the circle and proceeds in a specific direction around the circle. In each step, a certain number of people are skipped and the next person is executed (or eliminated). The elimination of people makes the circle smaller and smaller. At the last step, only one person remains who is declared the ‘winner’.

Therefore, if there are *n* number of people and a number *k* which indicates that *k*–1 people are skipped and *k–th* person in the circle is eliminated, then the problem is to choose a position in the initial circle so that the given person becomes the winner.

For example, if there are 5 (n) people and every second (k) person is eliminated, then first the person at position 2 is eliminated followed by the person at position 4 followed by person at position 1 and finally the person at position 5 is eliminated. Therefore, the person at position 3 becomes the winner.



Try the same process with n = 7 and k =3. You will find that person at position 4 is the winner. The elimination goes in the sequence of 3, 6, 2, 7, 5 and 1.

struct node

{

int player\_id;

struct node \*next;

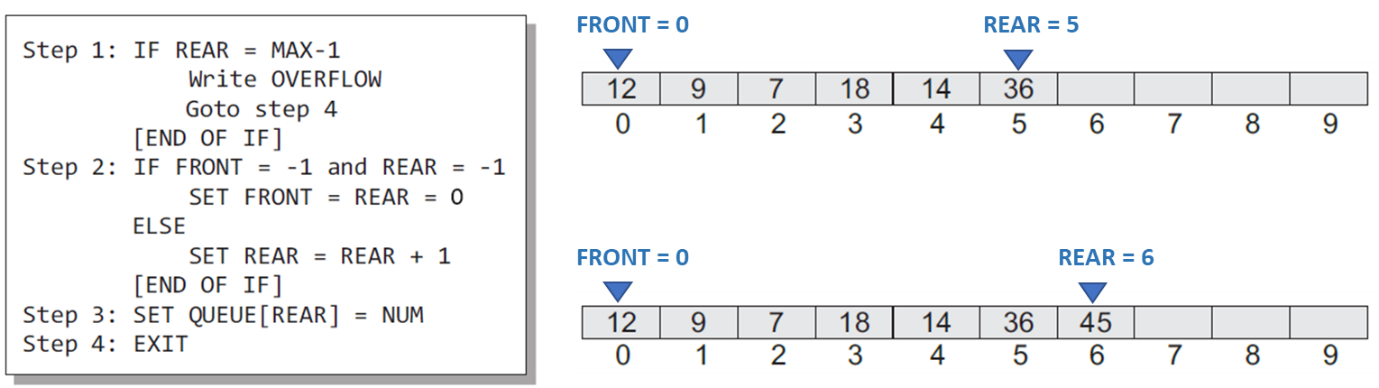
};

---end of Tutorial Week 4---

# Hints for Array Implementation of Queue

1. **Insert (enqueue) Operation**

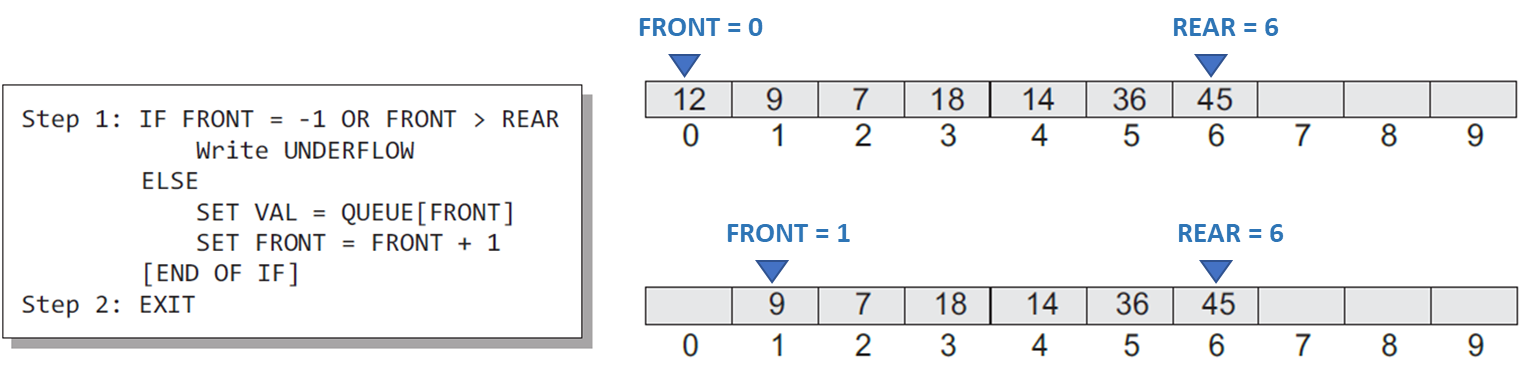
Suppose we want to add another element with value 45, then REAR would be incremented by 1 and the value would be stored at the position pointed by REAR.



Every time a new element has to be added, we repeat the same procedure. However, before inserting an element in a queue, we must check for overflow conditions. An overflow will occur when we try to insert an element into a queue that is already full. When REAR = MAX – 1, where MAX is the size of the queue, we have an overflow condition.

1. **Delete (dequeue) Operation**

If we want to delete an element from the queue, then the value of FRONT will be incremented. Deletions are done from only this end of the queue.

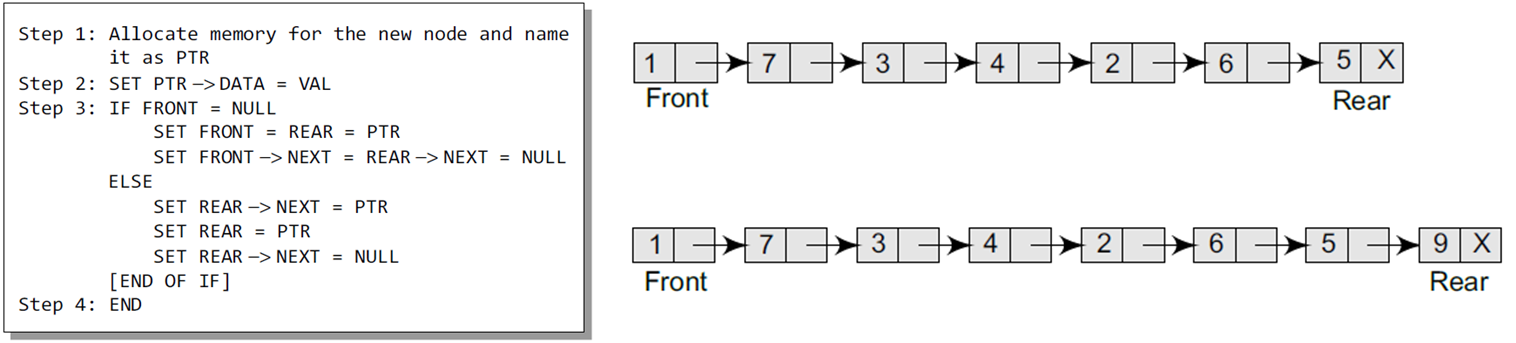


Before deleting an element from a queue, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a queue that is already empty. If FRONT = –1 and REAR = –1, it means there is no element in the queue.

# Hints for Linked Implementation of Queue

1. **Insert (enqueue) Operation**

The new element is added as the last element of the queue.

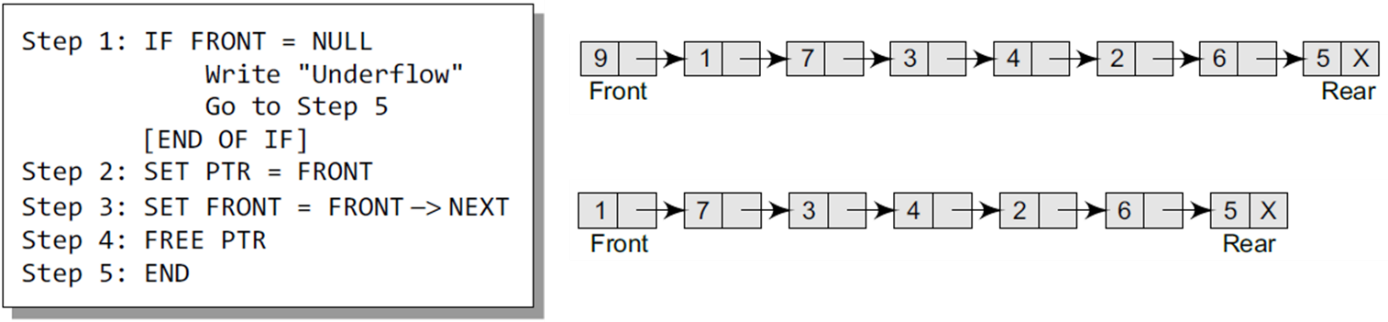


To insert an element with value 9, we first check if FRONT=NULL. If the condition holds, then the queue is empty. So, we allocate memory for a new node, store the value in its data part and NULL in its next part. The new node will then be called both FRONT and rear.

However, if FRONT != NULL, then we will insert the new node at the rear end of the linked queue and name this new node as rear.

1. **Delete (dequeue) Operation**

Before deleting the value, we must first check if FRONT=NULL because if this is the case, then the queue is empty and no more deletions can be done. If an attempt is made to delete a value from a queue that is already empty, an underflow message is printed.



To delete an element, we first check if FRONT=NULL. If the condition is false, then we delete the first node pointed by FRONT. The FRONT will now point to the second element of the linked queue. You also need to free the memory occupied by PTR to avoid memory leakage.