L7-8: Queues

SESSION	Sessions 7-8
CLASS MODE	LECTURE PRACTICAL
SLIDES	
Completed	☐
Date	@October 28, 2022

Queue

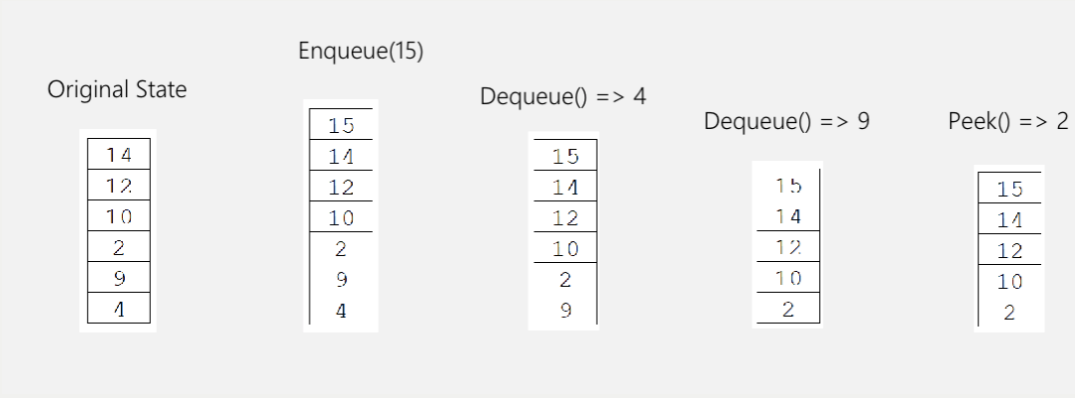
→Like queue in a coffeeshop

Queue ADT (Abstract Data Type)

Description

- Elements are added at one side (the end/rear/tail)
- Elements are taken at the other side (the beginning/front/head)
- Therefore, it is a FIFO (First In, First Out) data structure
- ADT (Abstract Data Type):
  - enqueue
  - dequeue
  - peek/first

▼ Queue Operations visual representation:



Creating a Queue class

💡 →Notes for Günther's way of making a Queue

▼ Starting point

```
public class Queue {

    public void Enqueue(int number) {

    }
    public int Dequeue() {

    }
    public int Peek() {

    }
}
```

This is the Queue "contract", this means it has been previously decided that a queue class HAS TO have all these functionalities (it is up to you HOW you implement it):

→ **Enqueue()** method: receives an integer as input and appends it to the queue.

→ **Dequeue()** method: eliminates and returns the bottom value in the queue.

→ **Peek()** method: return the bottom value in the queue.

How to create a Queue class:

1. We need an internal data structure → ARRAY
    a. We choose an array because it is our way of storing data, we choose how to manipulate this data in whichever way we want. In this case, we will manipulate it as a Queue class (Enqueuing a value at the end of the queue, Dequeuing and Peeking the front value of the queue).
2. We need at least two variables for the RearPosition (or TopPosition) and FrontPosition (or BottomPosition).
    a. This is for us to keep track of the indexes of the values we want to peek (the top and bottom, rear and front).

▼ Visual representation of the things we need

```
public class Queue {

    private int size;
    private int[] intArr;

    private int front = -1;
    private int rear = -1;

}
```

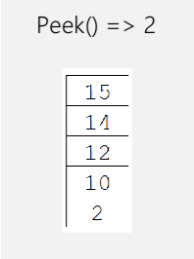
💡 As you can see the initial value of the TopPosition (rear) and BottomPosition (front) is -1, this is because we need to initialise it, and this number means our stack is empty (array indices start at 0). As soon as we introduce one value (one integer), these values will go in the first index of our array, index 0, so we will increase the value of the TopPosition and BottomPosition to 0.

Peek()

💡 **Peek() method:** this method returns the first value in our queue (FIFO). Careful, it doesn't print it, it returns it, this means that if we do $variable = Stack.Peek()$ the value that the function Peek() returns will be stored in variable.

```
//Peek() method: returns the Bottom value of our stack
    public int Peek(){
        if ((BottomPosition < 0) || (BottomPosition > TopPosition)){
            Console.WriteLine("Cannot Dequeue, the queue is empty");
            return 0;
        }
        else{
            int numerito = QueueArray[BottomPosition];
            return numerito;
        }
    }
```

▼ **Visual representation of Peek()**

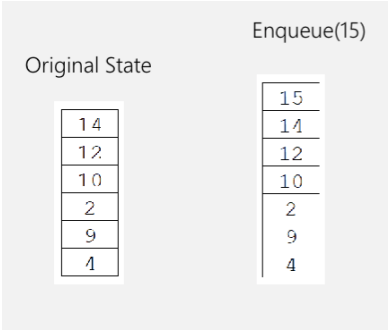


Enqueue()

💡 **Enqueue() method:** this method adds a new value to the stack, in oder words, appends a new value to the stack. We will do this by increasing the TopPosition (rear) by one and assigning that value to the index(TopPosition) of out array.

```
//Enqueue() method: receives integer 'number', increases variable 'TopPosition' by one and assigns number to that element in the array
    public void Enqueue(int number){
        if (TopPosition == Size-1){
            Console.WriteLine("Cannot Enqueue, the queue is full");
            return;
        }
        else{
            if ((BottomPosition==-1)&&(TopPosition==-1)){
                BottomPosition++;
                TopPosition++;
            }
            else{
                TopPosition++;
            }
            QueueArray[TopPosition] = number;
        }
    }
```

▼ **Visual representation of push**

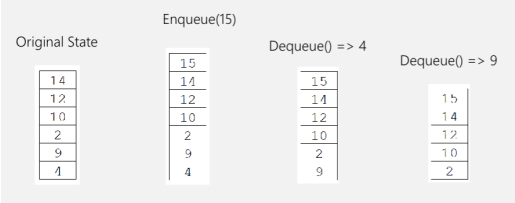


Dequeue()

💡 **Dequeue() method:** this method returns the bottom most value (BottomPosition or front)and removes it. As you can see it does the same function as peek, but it increases BottomPosition by one, **in this way it is like we removed the value from the stack!**
→ Same thing as Peek() we have to be careful because it does not print the value. It returns it. It is important to understand that returning the value means that after the function is executed, the function has a value. For example, if you multiply 2*2, the output is 4, so if you execute Pop() or Peek(), the output will be the top value.

```
//Dequeue() method: eliminates the bottom value in the stack, increases the 'BottomPosition' by one and returns
    //the integer that was eliminated
    public int Dequeue(){
        if (BottomPosition < 0 || BottomPosition > TopPosition){
            Console.WriteLine("Cannot Dequeue, the queue is empty");
            return 0;
        }
        else{
            int numerito = QueueArray[BottomPosition];
            if (BottomPosition==TopPosition){
                BottomPosition = -1;
                TopPosition = -1;
            }
            else{
                BottomPosition++;
            }
            return numerito;
        }
    }
```

▼ **Visual representation of pop**



Refining the Stack Class (Advanced Stack)

💡 Now that we have a first version of the class, we will modify it to make it **more sophisticated**.

🌶️ I have been lazy and the Sophisticated implementation is incorporated in the previous chunks of code. But I will address these parts in this section 🤗.

Stack Overflow → lifeguard Enqueue() method

💡 **Stack Overflow (or Queue Overflow in this case):** An error condition that occurs when there is no room in the stack for a new item.

💡 We look at this problem whenever we are enqueuing, and we solve it by checking if the Queue is full, in other words, if the $TopPosition = Size - 1$. If it is full, in order to enqueue (append a value) we do not allow to append another value, preventing the stack overflow problem.
→ Instead we could call the DoubleTheInternalArray() method which changes the size of the array and allows us to append.

Stack Underflow → lifeguard Peek() and Dequeue() methods.

💡 **Stack Underflow (or Queue Underflow in this case):** An error condition that occurs when an item is called for from the stack, but the stack is empty.

💡 We look at this problem whenever we are dequeuing or peeking, and we solve it by checking if the BottomPosition is **-1** (this means the Queue is empty), if it is **-1** we do not allow the code to Dequeue() or Peek().

Constructors

💡 Another incorporation in the Queue is the creation of **"CONSTRUCTORS"**. We have two of them, and their job is to initialise the necessary values.
→If you DO NOT input a value whenever you are calling the Class Queue from Main, it executes the first Constructor and sets the size of the array to a predetermined value (in this case size=10)
→If you DO pass an integer as an input from Main, this constructor will generate an array with this 'newSize', so we will have a Queue of the wanted size.

```
//Queue Constructor: If you DO NOT pass an input this constructor only generates the array for the predetermined 'Size'
 public Queue(){
     Size = 10;
     QueueArray = new int[Size];
 }

 //Queue Constructor: If you DO pass an integer as an input, this constructor will generate an array with this 'newSize'
 public Queue(int newSize){
     Size = newSize;
     QueueArray = new int[Size];
 }
```

🌶️ **CAREFUL!** Now you need a variable to store the size of the array (Size) → We declare it at the top of the code.

```
//We declare the internal variables we will use throughout the class
    private int Size;
    private int[] QueueArray;
    private int TopPosition = -1;
    private int BottomPosition = -1;
```

My codes VisualStudioCode

▼ Queue Implementation

```
internal class Queue
{
     //We declare the internal variables we will use throughout the class
    private int Size;
    private int[] QueueArray;
    private int TopPosition = -1;
    private int BottomPosition = -1;

     //Queue Constructor: If you DO NOT pass an input this constructor only generates the array for the predetermined 'Size'
    public Queue(){
        Size = 10;
        QueueArray = new int[Size];
    }

    //Queue Constructor: If you DO pass an integer as an input, this constructor will generate an array with this 'newSize'
    public Queue(int newSize){
        Size = newSize;
        QueueArray = new int[Size];
    }

    //Enqueue() method: receives integer 'number', increases variable 'TopPosition' by one and assigns number to that element in t
    public void Enqueue(int number){
```

```
        if (TopPosition == Size-1){
            Console.WriteLine("Cannot Enqueue, the queue is full");
            return;
        }
        else{
            if ((BottomPosition==-1)&&(TopPosition==-1)){
                BottomPosition++;
                TopPosition++;
            }
            else{
                TopPosition++;
            }
            QueueArray[TopPosition] = number;
        }
    }

    //Dequeue() method: eliminates the bottom value in the stack, increases the 'BottomPosition' by one and returns
    //the integer that was eliminated
    public int Dequeue(){
        if (BottomPosition < 0 || BottomPosition > TopPosition){
            Console.WriteLine("Cannot Dequeue, the queue is empty");
            return 0;
        }
        else{
            int numerito = QueueArray[BottomPosition];
            if (BottomPosition==TopPosition){
                BottomPosition = -1;
                TopPosition = -1;
            }
            else{
                BottomPosition++;
            }
            return numerito;
        }
    }

    //Peek() method: returns the Bottom value of our stack
    public int Peek(){
        if ((BottomPosition < 0) || (BottomPosition > TopPosition)){
            Console.WriteLine("Cannot Dequeue, the queue is empty");
            return 0;
        }
        else{
            int numerito = QueueArray[BottomPosition];
            return numerito;
        }
    }

    //PrintStack() method: prints the whole stack
    public void PrintQueue(){
        Console.WriteLine("The number of elements in the stack is: " + (TopPosition + 1));
        Console.WriteLine("The TopPosition of the Queue is in the index " + TopPosition + " of the array.");
        Console.WriteLine("The BottomPosition of the Queue is in the index " + BottomPosition + " of the array.");
        for(int i =BottomPosition; i <= TopPosition; i++){
            Console.WriteLine("position " + (i-BottomPosition) + ": " + QueueArray[i]);
        }
    }

    //ReorganiseQueue() method: I don't think I use it, but it rearranges the array so that all values are at the beginning of the
    private void ReorganiseQueue(){
        int j = 0;
        for(int i = BottomPosition; i<=TopPosition; i++){
            QueueArray[j] = QueueArray[i];
        }
        Console.WriteLine("The queue was reorganised correctly");
    }

    //DoubleTheInternalArray() method: This method doubles the size of the 'StackArray' so that we do not overflow the stack
    private void DoubleTheInternalArray(){
        int newSize = Size*2;
        int[] newQueueArray = new int[newSize];

        for(int i=0; i<=TopPosition; i++){
            newQueueArray[i] = QueueArray[i];
        }
        QueueArray = newQueueArray;
        Size = newSize;
        Console.WriteLine("Resized the queue");
        Console.WriteLine("New size; " + newSize);
    }
}
```