Cases Where $T(n) \in \Theta\ (1)$

```
public Customer first()
{
    return queue[front]; O(1) Operation
}

public int getSize()
{
    return size; O(1) Operation
}

public boolean isEmpty()
{
    return size == 0; O(1) Operation
}

public boolean isFull()
{
    return size == capacity; O(1) Operation
}

public CustomerQueue(boolean usesGrowthMethod)  //Initialize

{
    queue = new Customer[capacity]; O(1) Operation
    this.usesGrowthMethod = usesGrowthMethod; O(1) Operation
}
```

Enqueue

Best Case: $\boldsymbol{\Omega}(\mathbf{1})$

Average Case: Since $\Omega(1) \supset \Omega(\mathrm{n}), (\mathrm{T}(\mathrm{n}) \in \Omega(\mathrm{n}) \wedge \mathrm{T}(\mathrm{n}) \in \mathrm{O}(\mathrm{n})) \rightarrow \boldsymbol{\Theta}(\boldsymbol{n})$

Worst Case: $\mathbf{O}(\boldsymbol{n})$ (Due to Enforcing Minimum More Often than Not)

```
public void enqueue(Customer customer)
{
    int allocated = 0; O(1) Operation
    Customer temp = customer; O(1) Operation

    try
    {
        if (isFull())
        {
            if (size == 80)
            {
                throw new QueueOverflowException(); O(1) Operation
            }
            else
            {
                dynamicallyResize(); O(n) Operation
            }
        }

        allocated = (front + size) % capacity; O(1) Operation
        queue[allocated] = temp; O(1) Operation
        size++; O(1) Operation

        if (!isEmpty())
        {
            if (customer.getId() < first().getId())
            {
                restoreMin(); O(n) Operation
            }
        }
    }
    catch (QueueOverflowException e)
    {
        System.out.println(e.getMessage()); O(1) Operation
    }
}
```

Dequeue

Best Case: $\boldsymbol{\Omega(1)}$

Average Case: Since $\Omega(1) \supset \Omega(n), (T(n) \in \Omega(n) \wedge T(n) \in O(n)) \rightarrow \boldsymbol{\Theta(n)}$

Worst Case: $\boldsymbol{O(n)}$ (Due to Enforcing Minimum More Often than Not)

```
public Customer dequeue(boolean enforceMinimum)

{
    Customer first = queue[front]; O(1) Operation
    try
    {
        if (isEmpty())
        {
            throw new QueueUnderflowException(); O(1) Operation
        }
        else
        {
            first = queue[front]; O(1) Operation
            queue[front] = null; O(1) Operation
            front = (front + 1) % capacity; O(1) Operation
            size--; O(1) Operation
            //When resizing the queue this does not need to occur.
            if (enforceMinimum)
            {
                restoreMin(); O(n) Operation
            }
        }
    }
    catch (QueueUnderflowException e)
    {
        System.out.println(e.getMessage()); O(1) Operation
    }
    return first; O(1) Operation
}
```