Zakho Technical College

Department of Computer Information System

Data Structures and Algorithms

3. Analysis (Queue and Circular Queue)

Lecturers:

Sipan M. Hameed

Ahmed Jamil

www.sipan.dev

2024-2025

1. Table of Contents

3. C# Standard Queue and Circular Queue

Let's describe detailed explanation of **C# Queues** and **Circular Queues** with **code examples** and **time/space complexity analysis** for all operations:

3.1 Built-in Queue<T> in C#

The Queue<T> class in System.Collections.Generic implements a **dynamic FIFO (First-In-First-Out)** data structure using a circular buffer internally. It automatically resizes when full.

3.2 Example Code:

```
using System;

using System.Collections.Generic;


Queue<int> queue = new Queue<int>();
```

// Enqueue elements (O(1) amortized time)

```
queue.Enqueue(10); // Add 10

queue.Enqueue(20); // Add 20

queue.Enqueue(30); // Add 30
```

// Dequeue front element (O(1) time)

```
int removedItem = queue.Dequeue(); // Removes 10
```

// Peek at front element (O(1) time)

```
int frontItem = queue.Peek(); // Returns 20
```

// Check count (O(1) time)

```
Console.WriteLine(queue.Count); // Output: 2
```

// Check if an element exists (O(n) time)

```
bool contains20 = queue.Contains(20); // Returns true
```

3.3 Time/Space Complexity:

Operation	Time Complexity	Space Complexity	Description
Enqueue()	**O(1) amortized**	O(n) (dynamic growth)	Adds an item to the rear.
Dequeue()	**O(1)**	O(n)	Removes the front item.
Peek()	**O(1)**	O(n)	Returns the front item.
Contains()	**O(n)**	-	Checks for element existence.
Count	**O(1)**	-	Returns current element count.

3.4 Key Notes:

- Enqueue is **amortized O(1)** because the internal array occasionally doubles in size (an O(n) operation), but this cost is spread over many operations.
- Space grows linearly with the number of elements (O(n)).

3.5 Custom Circular Queue (Fixed-Size)

A **Circular Queue** uses a fixed-size array to reuse space efficiently. When the rear index reaches the end, it wraps to the start using modulo arithmetic.

3.6 Example Code:

```
using System;
namespace ConsoleApp2
{
    public class CircularQueue<T>
    {
        private readonly T[] _elements;
        private int _front;
        private int _rear;
        private int _count;
        private readonly int _capacity;

        public CircularQueue(int capacity)
        {
            _capacity = capacity;
            _elements = new T[capacity];
            _front = _rear = _count = 0;
        }

        // Enqueue: O(1) time
        public void Enqueue(T item)
        {
            if (IsFull)
                throw new InvalidOperationException("Queue is full.");

            _elements[_rear] = item;
            _rear = (_rear + 1) % _capacity; // Wrap around
            _count++;
        }

        // Dequeue: O(1) time
        public T Dequeue()
        {
            if (IsEmpty)
                throw new InvalidOperationException("Queue is empty.");

            T item = _elements[_front];
            _front = (_front + 1) % _capacity; // Wrap around
            _count--;
            return item;
        }
```

```
// Peek: O(1) time
        public T Peek() => IsEmpty ? throw new
InvalidOperationException("Queue is empty.") : _elements[_front];

        public bool IsEmpty => _count == 0; // O(1)
        public bool IsFull => _count == _capacity; // O(1)
        public int Count => _count; // O(1)
    }
    class Program
    {
        static void Main(string[] args)
        {
            var cq = new CircularQueue<string>(3);
            cq.Enqueue("A"); // O(1)
            cq.Enqueue("B"); // O(1)
            cq.Enqueue("C"); // O(1)
            Console.WriteLine(cq.Dequeue()); // "A" (O(1))
            cq.Enqueue("D"); // Success (O(1))
            Console.WriteLine(cq.Peek()); // "B" (O(1))

        }
    }
}
```

3.7 Time/Space Complexity:

Operation	Time Complexity	Space Complexity	Description
Enqueue()	**O(1)**	O(k) (fixed preallocation)	Adds to the rear.
Dequeue()	**O(1)**	O(k)	Removes the front item.
Peek()	**O(1)**	O(k)	Returns front item.
IsEmpty	**O(1)**	-	Checks if empty.
IsFull	**O(1)**	-	Checks if full.
Count	**O(1)**	-	Returns current element count.

3.8 Key Notes:

- Fixed space (O(k)) where k is the capacity.
- Uses modulo operations (%) to wrap indices.
- No resizing overhead: All operations are strictly **O(1)**.

3.9 Key Differences

Feature	Queue<T>	Circular Queue
Resizing	Yes (dynamic growth)	No (fixed size)
Enqueue Time	O(1) amortized	O(1)
Space Overhead	O(n) (dynamic)	O(k) (fixed)
Use Case	General-purpose FIFO	Fixed-size buffers

3.10 When to Use Which?

1. **Queue<T>**:
    - Use for **dynamic data** where the maximum size is unknown (e.g., task scheduling, request handling).
    - Preferred for most general-purpose FIFO needs.
2. **Circular Queue**:
    - Use for **fixed-size buffers** (e.g., streaming data, hardware buffers, caching).
    - Ideal when memory constraints are strict or resizing is undesirable.

3.11 Trade-offs Summary

- **Dynamic Queue**: Flexible but has occasional O(n) resizing costs.
- **Circular Queue**: Predictable O(1) operations but fixed capacity.

Choose based on your application's requirements for memory flexibility vs. performance guarantees.