Introduction to Stacks and Queues

Specialized Collections in C#

Scott Tremaine
Software Developer and Educator

Breakpoint Coding Tutorials

Contents

Understanding Stacks

A stack is an abstract data type (ADT) that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the most recently added element. The order in which elements come off a stack gives rise to its alternative name, LIFO (Last In, First Out).

Key Characteristics of a Stack

- The defining feature of a stack is its LIFO structure, meaning the last element added (pushed) to the stack will be the first one to be removed (popped). This is similar to a stack of plates; you add (push) plates to the top and remove (pop) the topmost plate first.
- Elements are added and removed from the top of the stack only, referred to as the "top" of the stack. Unlike arrays or linked lists, stacks do not allow access to elements at arbitrary positions.
- The size of a stack can grow or shrink dynamically as elements are pushed or popped, respectively. This makes stacks very flexible in terms of memory usage.

Common Uses of Stacks

Stacks are used in a variety of scenarios where the LIFO order is important. Examples include:

- **Function Call Management:** The call stack in most programming languages is a stack that keeps track of function calls, managing the return addresses and local variables.
- **Expression Evaluation:** Stacks are used to evaluate arithmetic expressions, particularly in converting infix expressions to postfix expressions and then evaluating them.
- **Undo Mechanism:** Many applications, like text editors, use stacks to implement the undo feature. Each action is pushed onto the stack, and an undo operation pops the last action.

LIFO (Last In, First Out) Principle

The LIFO principle is the cornerstone of the stack data structure.

- When you add an element to a stack, it is placed on top of all previously added elements. This means the newest element is always at the top of the stack.
- When you remove an element from a stack, you take the topmost element off, which is the most recently added element. This ensures that the last element added is the first one to be removed.

Think of a stack of books. If you add books one by one to a pile (push), the last book you placed on the pile will be the first one you take off when you start removing books (pop). This last book in, first book out sequence exemplifies the LIFO principle. The LIFO principle affects how algorithms and processes are designed.

For example, recursive algorithms utilize the call stack to keep track of previous function calls, and each return from a recursive call follows the LIFO order, ensuring correct execution flow.

Importance of Understanding Stacks

Understanding stacks and the LIFO principle is fundamental for several reasons:

- Many algorithms, especially those involving recursion, depth-first search (DFS) in graph theory, and expression parsing, rely on stacks. Grasping the LIFO concept is crucial for implementing and optimizing these algorithms.
- Stacks help in managing memory efficiently, particularly in programming languages that use the stack for function call management. This ensures that local variables and return addresses are stored and accessed in a structured manner.
- Many complex problems can be broken down into simpler sub-problems that can be managed using stacks. Understanding how to utilize stacks effectively can simplify problem-solving approaches and lead to more efficient solutions.

Implementing a Stack

Let's explore how to use the built-in `Stack<T>` class in C# and perform basic stack operations: Push, Pop, Peek, and IsEmpty.

Creating a Stack

First, you need to include the necessary namespace and create an instance of the `Stack<T>` class.

```
using System;
using System.Collections.Generic;

public class StackExample
{
    public static void Main()
    {
        Stack<int> stack = new Stack<int>();
    }
}
```

Push Operation

The `Push` method adds an element to the top of the stack.

```
stack.Push(10);
stack.Push(20);
stack.Push(30);
```

Pop Operation

The `Pop` method removes and returns the top element of the stack. If the stack is empty, it throws an `InvalidOperationException`.

```
int poppedElement = stack.Pop();
Console.WriteLine($"Popped element: {poppedElement}");
```

Peek Operation

The `Peek` method returns the top element without removing it from the stack. If the stack is empty, it throws an `InvalidOperationException`.

```
int topElement = stack.Peek();
Console.WriteLine($"Top element: {topElement}");
```

IsEmpty Operation

The `IsEmpty` method checks whether the stack is empty. In the built-in `Stack<T>`, this can be achieved using the `Count` property.

```
bool isEmpty = stack.Count == 0;
Console.WriteLine($"Is stack empty: {isEmpty}");
```

Complete Example

Here's a complete example demonstrating all these operations using the built-in `Stack<T>` class in C#:

```
using System;
using System.Collections.Generic;

public class StackExample
{
    public static void Main()
    {
        // Create a stack instance
        Stack<int> stack = new Stack<int>();

        // Push elements onto the stack
        stack.Push(10);
        stack.Push(20);
        stack.Push(30);

        // Peek the top element
        Console.WriteLine($"Top element: {stack.Peek()}");

        // Pop an element from the stack
```

```
        Console.WriteLine($"Popped element: {stack.Pop()}");

        // Check the top element again
        Console.WriteLine($"Top element after pop: {stack.Peek()}");

        // Check if the stack is empty
        Console.WriteLine($"Is stack empty: {stack.Count == 0}");

        // Pop remaining elements
        stack.Pop();
        stack.Pop();

        // Verify the stack is empty
        Console.WriteLine($"Is stack empty after popping all elements:
    {stack.Count == 0}");
     }
}
```

Practical Stack Examples

Let's explore two practical applications of stacks: balancing parentheses and implementing undo functionality in applications. These examples will demonstrate how stacks can be effectively utilized to solve real-world problems.

Balancing Parentheses

Balancing parentheses is a common problem in computer science, especially in the context of compilers and interpreters. The goal is to ensure that every opening parenthesis has a corresponding closing parenthesis in the correct order.

Problem Statement

Given a string containing parentheses, determine if the parentheses are balanced. A string is considered balanced if:

- Every opening parenthesis has a corresponding closing parenthesis.
- Parentheses are correctly nested.

Example

- Input: `"(())"` - Balanced
- Input: `"(()"` - Not balanced
- Input: `"()()"` - Balanced

Algorithm

1. Initialize an empty stack.
2. Traverse each character in the string.

3. If the character is an opening parenthesis `(`, push it onto the stack.
4. If the character is a closing parenthesis `)`, check the stack:
    - If the stack is empty, return false (unbalanced).
    - Pop the top element from the stack. If it does not match the corresponding opening parenthesis, return false.
5. After traversal, if the stack is not empty, return false (unbalanced).
6. If the stack is empty, return true (balanced).

Implementation

```
using System;
using System.Collections.Generic;

public class ParenthesesBalancer
{
    public static bool AreParenthesesBalanced(string input)
    {
        Stack<char> stack = new Stack<char>();

        foreach (char c in input)
        {
            if (c == '(')
            {
                stack.Push(c);
            }
            else if (c == ')')
            {
                if (stack.Count == 0)
                {
                    return false;
                }
                stack.Pop();
            }
        }

        return stack.Count == 0;
    }

    public static void Main()
    {
        string input = "(())";
        bool result = AreParenthesesBalanced(input);
        Console.WriteLine($"Are the parentheses in \"{input}\" balanced
   ? {result}");

        input = "(()";
        result = AreParenthesesBalanced(input);
        Console.WriteLine($"Are the parentheses in \"{input}\" balanced
   ? {result}");

        input = "()()";
        result = AreParenthesesBalanced(input);
```

```
        Console.WriteLine($"Are the parentheses in \"{input}\" balanced
    ? {result}");
    }
}
```

Undo Functionality in Applications

The undo functionality is a common feature in many applications, such as text editors and graphics software. It allows users to revert the last action performed. Stacks are ideal for implementing this functionality because they follow the LIFO principle, which ensures that the most recent action is undone first.

Problem Statement

Implement an undo feature that allows reverting the last performed action.

Algorithm

1. Use a stack to keep track of actions performed.
2. When an action is performed, push it onto the stack.
3. When an undo operation is requested, pop the top action from the stack and revert it.

Example

- Actions: Type `"A"`, Type `"B"`, Type `"C"`
- Undo: Remove `"C"`
- Undo: Remove `"B"`

Implementation

```
using System;
using System.Collections.Generic;

public class UndoFunctionality
{
    private Stack<string> actionStack = new Stack<string>();
    private string currentText = "";

    public void PerformAction(string action)
    {
        actionStack.Push(action);
        currentText += action;
        Console.WriteLine($"Performed action: {action}, Current text: {
    currentText}");
    }

    public void UndoAction()
    {
```

```
        if (actionStack.Count == 0)
        {
            Console.WriteLine("No actions to undo");
            return;
        }

        string lastAction = actionStack.Pop();
        currentText = currentText.Substring(0, currentText.Length -
    lastAction.Length);
        Console.WriteLine($"Undone action: {lastAction}, Current text:
    {currentText}");
    }

    public static void Main()
    {
        UndoFunctionality undoFeature = new UndoFunctionality();

        undoFeature.PerformAction("A");
        undoFeature.PerformAction("B");
        undoFeature.PerformAction("C");

        undoFeature.UndoAction();
        undoFeature.UndoAction();
    }
}
```

Explanation of the Examples

Balancing Parentheses

Stack Usage: The stack is used to store opening parentheses as they are encountered. When a closing parenthesis is found, the stack is checked to ensure there is a corresponding opening parenthesis. **Edge Cases:** The algorithm handles cases where there are unmatched opening or closing parentheses by checking the stack's state after traversal.

Undo Functionality

Stack Usage: The stack records each action performed. The `PerformAction` method adds actions to the stack, while the `UndoAction` method removes the most recent action. **Edge Cases:** The algorithm checks if the stack is empty before attempting to undo an action, ensuring that no errors occur when there are no actions to undo.

Understanding Queues

A queue is an abstract data type (ADT) that serves as a collection of elements, with two primary operations: enqueue, which adds an element to the end of the collection, and dequeue, which removes the element from the front of the collection. The order in which elements come out of a queue gives rise to its alternative name, FIFO (First In, First Out).

Key Characteristics of a Queue

- The defining feature of a queue is its FIFO structure, meaning the first element added (enqueued) to the queue will be the first one to be removed (dequeued). This is similar to a line of people waiting at a service counter; the first person in line is the first to be served.
- Queues have two access points: the front and the rear. Elements are added at the rear and removed from the front. Unlike stacks, queues do not allow access to elements at arbitrary positions.
- The size of a queue can grow or shrink dynamically as elements are enqueued or dequeued, respectively. This makes queues very flexible in terms of memory usage.

Examples of Queue Usage

- **Task Scheduling:** Queues are used to manage tasks in operating systems, ensuring that processes are executed in the order they arrive.
- **Print Spooling:** Print jobs are managed in a queue to ensure that documents are printed in the order they are submitted.
- **Breadth-First Search (BFS):** Queues are essential for BFS algorithms in graph traversal and pathfinding.

FIFO (First In, First Out) Principle

The FIFO principle is the cornerstone of the queue data structure. To understand this principle better, consider the following aspects:

- When you add an element to a queue, it is placed at the end of the queue, behind all previously added elements. This ensures that the newest element waits behind the elements that were added before it.
- When you remove an element from a queue, you take the frontmost element, which is the oldest element that has been in the queue the longest. This ensures that elements are processed in the order they were added.
- Think of a queue at a ticket counter. People join the queue at the end and are served at the front. The first person to join the queue is the first person to be served, and this order is maintained for everyone in the queue.

Impact on Algorithms and Processes

The FIFO principle affects how algorithms and processes are designed:

- Task scheduling in operating systems uses queues to ensure that tasks are processed in the order they arrive, preventing newer tasks from jumping ahead of older ones.

Importance of Understanding Queues

Understanding queues and the FIFO principle is fundamental for several reasons:

- Many algorithms, especially those involving level-order traversal in trees, breadth-first search in graphs, and certain scheduling algorithms, rely on queues. Grasping the FIFO concept is crucial for implementing and optimizing these algorithms.
- Queues help in managing processes efficiently, particularly in systems that require orderly processing of tasks. This ensures that resources are allocated fairly and that processes are handled in a predictable manner.
- Many complex problems can be simplified using queues. Understanding how to utilize queues effectively can streamline problem-solving approaches and lead to more efficient solutions.

Using the Built-In Queue<T> Class

To start, you need to include the necessary namespace and create an instance of the Queue<T> class. The Queue<T> class is a generic class, so you can specify the type of elements it will store.

Creating a Queue

```
using System;
using System.Collections.Generic;

public class QueueExample
{
    public static void Main()
    {
        Queue<int> queue = new Queue<int>();
    }
}
```

Adding Elements to the Queue

The Enqueue method adds an element to the end of the queue.

```
queue.Enqueue(10);
queue.Enqueue(20);
queue.Enqueue(30);
Console.WriteLine("Elements added to the queue.");
```

Removing Elements from the Queue

The Dequeue method removes and returns the front element of the queue. If the queue is empty, it throws an InvalidOperationException.

```
int dequeuedElement = queue.Dequeue();
Console.WriteLine($"Dequeued element: {dequeuedElement}");
```

Peeking at the Front Element

The Peek method returns the front element without removing it from the queue. If the queue is empty, it throws an InvalidOperationException.

```
int frontElement = queue.Peek();
Console.WriteLine($"Front element: {frontElement}");
```

Checking if the Queue is Empty

The IsEmpty method checks whether the queue is empty. In the built-in Queue<T>, this can be achieved using the Count property.

```
bool isEmpty = queue.Count == 0;
Console.WriteLine($"Is queue empty: {isEmpty}");
```

Complete Example

Here's a complete example demonstrating all these operations using the built-in Queue<T> class in C#:

```
using System;
using System.Collections.Generic;

public class QueueExample
{
    public static void Main()
    {
        // Create a queue instance
        Queue<int> queue = new Queue<int>();

        // Enqueue elements into the queue
        queue.Enqueue(10);
        queue.Enqueue(20);
        queue.Enqueue(30);
        Console.WriteLine("Elements added to the queue: 10, 20, 30");

        // Peek the front element
        int frontElement = queue.Peek();
        Console.WriteLine($"Front element: {frontElement}");

        // Dequeue an element from the queue
        int dequeuedElement = queue.Dequeue();
        Console.WriteLine($"Dequeued element: {dequeuedElement}");

        // Check the front element again
        frontElement = queue.Peek();
        Console.WriteLine($"Front element after dequeue: {frontElement}
   ");

        // Check if the queue is empty
        bool isEmpty = queue.Count == 0;
        Console.WriteLine($"Is queue empty: {isEmpty}");

        // Dequeue remaining elements
        queue.Dequeue();
```

```
        queue.Dequeue();

        // Verify the queue is empty
        isEmpty = queue.Count == 0;
        Console.WriteLine($"Is queue empty after dequeuing all elements
    : {isEmpty}");
    }
}
```

Detailed Explanation of Each Operation

Enqueue

The `Enqueue` method adds an element to the end of the queue. This ensures that elements are added in the order they arrive, maintaining the FIFO order.

```
queue.Enqueue(10);
queue.Enqueue(20);
queue.Enqueue(30);
```

In this example, the elements 10, 20, and 30 are added to the queue in that order.

Dequeue

The `Dequeue` method removes and returns the front element of the queue. This operation ensures that elements are removed in the order they were added.

```
int dequeuedElement = queue.Dequeue();
Console.WriteLine($"Dequeued element: {dequeuedElement}");
```

In this example, the front element (10) is removed from the queue and its value is printed.

Peek

The `Peek` method returns the front element without removing it from the queue. This allows you to see the front element while keeping the queue intact.

```
int frontElement = queue.Peek();
Console.WriteLine($"Front element: {frontElement}");
```

In this example, the front element (20) is retrieved and its value is printed without removing it from the queue.

IsEmpty

The `IsEmpty` method checks whether the queue is empty by comparing the `Count` property to 0.

```
bool isEmpty = queue.Count == 0;
Console.WriteLine($"Is queue empty: {isEmpty}");
```

In this example, the `isEmpty` variable will be true if the queue is empty and false otherwise.

Practical Queue Examples

Let's explore practical applications of queues by implementing two real-world scenarios: a printer queue simulation and an order processing system. These examples will help illustrate how queues can be used to manage tasks efficiently and ensure orderly processing of elements.

Printer Queue Simulation

A printer queue is a classic example of a queue in action. In this simulation, we will model a printer that processes print jobs in the order they are received. Each print job will be added to the queue, and the printer will process them one by one.

Problem Statement

Simulate a printer queue where print jobs arrive and are processed in the order they are received.

Solution

We will use a queue to manage the print jobs. Each print job will be represented as a simple class with attributes like job name and number of pages. The queue will handle the addition of new print jobs and the processing of these jobs in a FIFO manner.

Create a class to represent a print job with necessary attributes

```
using System;
using System.Collections.Generic;

public class PrintJob
{
    public string JobName { get; set; }
    public int NumberOfPages { get; set; }

    public PrintJob(string jobName, int numberOfPages)
    {
        JobName = jobName;
        NumberOfPages = numberOfPages;
    }

    public override string ToString()
    {
        return $"{JobName} ({NumberOfPages} pages)";
    }
}
```

Use a Queue to manage the print jobs

```
public class PrinterQueue
{
    private Queue<PrintJob> queue;
```

```
    public PrinterQueue()
    {
        queue = new Queue<PrintJob>();
    }

    public void AddJob(PrintJob job)
    {
        queue.Enqueue(job);
        Console.WriteLine($"Added: {job}");
    }

    public void ProcessJobs()
    {
        while (queue.Count > 0)
        {
            PrintJob job = queue.Dequeue();
            Console.WriteLine($"Processing: {job}");
            // Simulate the time taken to print the job
            System.Threading.Thread.Sleep(job.NumberOfPages * 100);
        }
        Console.WriteLine("All jobs processed.");
    }

    public bool IsEmpty()
    {
        return queue.Count == 0;
    }
}
```

Add some print jobs to the queue and process them

```
public class Program
{
    public static void Main()
    {
        PrinterQueue printerQueue = new PrinterQueue();

        // Add some print jobs to the queue
        printerQueue.AddJob(new PrintJob("Job1", 5));
        printerQueue.AddJob(new PrintJob("Job2", 3));
        printerQueue.AddJob(new PrintJob("Job3", 10));

        // Process the print jobs
        printerQueue.ProcessJobs();
    }
}
```

Explanation

- We define a `PrintJob` class to represent each print job.
- The `PrinterQueue` class manages a queue of `PrintJob` objects using the built-in `Queue<T>` class.

- The `AddJob` method enqueues new print jobs, and the `ProcessJobs` method dequeues and processes each job in FIFO order.
- The `Main` method adds some sample print jobs to the queue and processes them.

Order Processing System

An order processing system is another practical example where queues are used to manage tasks. In this system, customer orders are received and processed in the order they are placed.

Problem Statement

Simulate an order processing system where customer orders are processed in the order they are received.

Solution

We will use a queue to manage customer orders. Each order will be represented as a class with attributes like order ID, customer name, and order details. The queue will handle the addition of new orders and the processing of these orders in a FIFO manner.

Create a class to represent a customer order with necessary attributes

```
using System;
using System.Collections.Generic;

public class Order
{
    public int OrderID { get; set; }
    public string CustomerName { get; set; }
    public string OrderDetails { get; set; }

    public Order(int orderId, string customerName, string orderDetails)
    {
        OrderID = orderId;
        CustomerName = customerName;
        OrderDetails = orderDetails;
    }

    public override string ToString()
    {
        return $"OrderID: {OrderID}, Customer: {CustomerName}, Details:
    {OrderDetails}";
    }
}
```

Use a Queue to manage the customer orders

```
public class OrderQueue
{
    private Queue<Order> queue;
```

```
    public OrderQueue()
    {
        queue = new Queue<Order>();
    }

    public void AddOrder(Order order)
    {
        queue.Enqueue(order);
        Console.WriteLine($"Added: {order}");
    }

    public void ProcessOrders()
    {
        while (queue.Count > 0)
        {
            Order order = queue.Dequeue();
            Console.WriteLine($"Processing: {order}");
            // Simulate the time taken to process the order
            System.Threading.Thread.Sleep(500);
        }
        Console.WriteLine("All orders processed.");
    }

    public bool IsEmpty()
    {
        return queue.Count == 0;
    }
}
```

Add some orders to the queue and process them

```
public class Program
{
    public static void Main()
    {
        OrderQueue orderQueue = new OrderQueue();

        // Add some orders to the queue
        orderQueue.AddOrder(new Order(1, "Alice", "2x T-shirts, 1x
   Jeans"));
        orderQueue.AddOrder(new Order(2, "Bob", "1x Laptop, 1x Mouse"))
   ;
        orderQueue.AddOrder(new Order(3, "Charlie", "3x Books"));

        // Process the orders
        orderQueue.ProcessOrders();
    }
}
```

Explanation

- We define an `Order` class to represent each customer order.
- The `OrderQueue` class manages a queue of `Order` objects using the built-in `Queue<T>` class.

- The `AddOrder` method enqueues new orders, and the `ProcessOrders` method dequeues and processes each order in FIFO order.
- The `Main` method adds some sample orders to the queue and processes them.

Comparing Stacks and Queues

Differences and Similarities

Stacks and queues are both fundamental data structures that provide ways to store and manage collections of elements. While they share some similarities, they also have distinct differences that make each suitable for specific scenarios.

Differences

Order of Operations:

- **Stack:** Operates on a LIFO (Last In, First Out) principle. The last element added (pushed) to the stack is the first one to be removed (popped).
- **Queue:** Operates on a FIFO (First In, First Out) principle. The first element added (enqueued) to the queue is the first one to be removed (dequeued).

Access Points:

- **Stack:** Elements are added and removed from the same end, referred to as the "top" of the stack.
- **Queue:** Elements are added at the "rear" and removed from the "front," having two distinct access points.

Use Cases:

- **Stack:** Ideal for scenarios requiring backtracking or reverse-order processing, such as undo mechanisms, recursive algorithms, and syntax parsing.
- **Queue:** Suitable for orderly processing where the order of arrival needs to be preserved, such as task scheduling, print spooling, and breadth-first search algorithms.

Similarities

- Both stacks and queues are abstract data types that define a specific way of organizing and accessing data, independent of underlying implementations.
- Both structures can dynamically grow and shrink as elements are added or removed, offering flexibility in memory usage.
- Both data structures support basic operations to add, remove, and inspect elements, although the methods and the order of these operations differ.

Use Cases Comparison

Stacks

- The call stack in programming languages tracks active subroutines, managing function calls and returns in a LIFO order.
- Used in converting and evaluating mathematical expressions (infix, postfix).
- Many applications use stacks to implement undo features, storing previous states to revert changes.

Queues

- Operating systems use queues to manage process scheduling, ensuring fair CPU time allocation.
- Print jobs are queued to be processed in the order they were received, ensuring orderly printing.
- Queues are essential for BFS algorithms in graph traversal, exploring nodes level by level.

Choosing the Right Data Structure

Choosing between a stack and a queue depends on the specific requirements of the problem at hand. Several factors should be considered:

Order of Processing

- If the problem requires processing elements in reverse order of their arrival (LIFO), a stack is the appropriate choice.
- If the problem requires processing elements in the same order as their arrival (FIFO), a queue is the better option.

Access Pattern

- Stacks are suitable for problems that involve recursive processing or backtracking.
- Queues are ideal for problems that require level-order traversal or orderly task processing.