Введение

Стек — это линейная структура данных, которая следует определенному порядку выполнения операций. Порядок может быть LIFO (Last In First Out) или FILO (First In Last Out). LIFO подразумевает, что элемент, который вставлен последним, выводится первым, а FILO подразумевает, что элемент, который вставлен первым, выводится последним.

Он ведет себя как стопка тарелок, где последняя добавленная тарелка — первая из тех, которые нужно удалить. Подумайте об этом так:

Помещение элемента в стек похоже на добавление новой тарелки сверху.

Выталкивание элемента удаляет верхнюю тарелку из стека.

1. Описание алгоритма

Стек (от англ. *stack* — стопка) — структура данных, представляющая из себя упорядоченный набор элементов, в которой добавление новых элементов и удаление существующих производится с одного конца, называемого вершиной стека. Притом первым из стека удаляется элемент, который был помещен туда последним, то есть в стеке реализуется стратегия «последним вошел — первым вышел» (last-in, first-out — LIFO). Примером стека в реальной жизни может являться стопка тарелок: когда мы хотим вытащить тарелку, мы должны снять все тарелки выше. Вернемся к описанию операций стека:

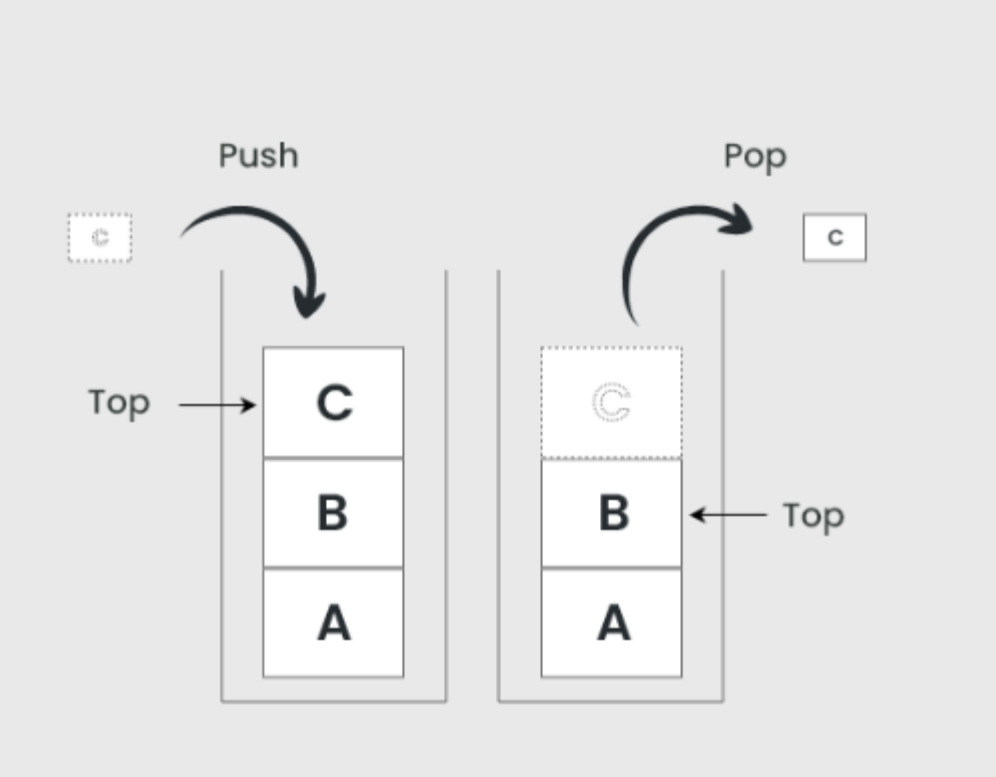
- emptyempty — проверка стека на наличие в нем элементов,
- pushpush (запись в стек) — операция вставки нового элемента,
- poppop (снятие со стека) — операция удаления нового элемента.

Реализации

Для стека с n

элементами требуется O(n)

памяти, так как она нужна лишь для хранения самих элементов.



Основные операции стека:

Для выполнения манипуляций в стеке нам предоставлены определенные операции для Stack, в том числе:

• push() для вставки элемента в стек

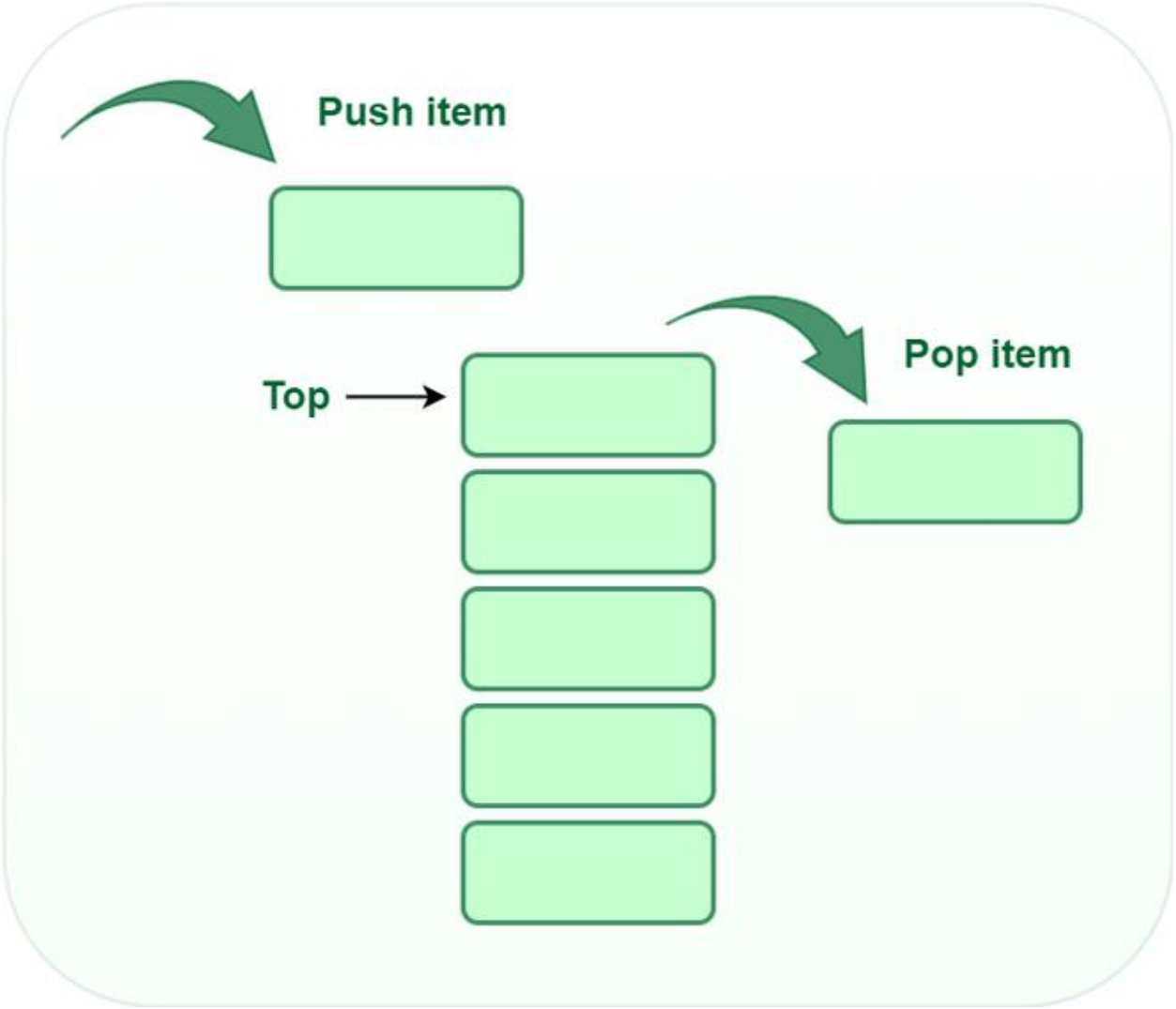
• pop() для удаления элемента из стека

• top() Возвращает верхний элемент стека.

• isEmpty() возвращает true, если стек пуст, в противном случае false.

• size() возвращает размер стека.

В этой статье мы рассмотрим, как выполнять эти операции в Stack.



Псевдокод основных операций:

Операция Push добавляет элемент в стек.

Если стек заполнен, то говорят, что это состояние переполнения.

```
using System;
using System.Collections.Generic;
class Program {
   static void Main()
   {
      Stack<int> s = new Stack<int>(); // Creating a stack
                                       // of integers
```

```
        s.Push(1); // Pushing 1 to the stack top
        s.Push(2); // Pushing 2 to the stack top
        s.Push(3); // Pushing 3 to the stack top
        s.Push(4); // Pushing 4 to the stack top
        s.Push(5); // Pushing 5 to the stack top

        // Printing the stack
        while (s.Count > 0) {
            Console.Write(
                s.Peek()
                + " "); // Peek() gets the top element
                        // without removing it
            s.Pop(); // Pop() removes the top element
        }

        // The above loop prints "5 4 3 2 1"
    }
}
```

Вывод

5 4 3 2 1

Операция извлечения из стека:

Операция извлечения используется для удаления элемента из стека.

Элементы извлекаются в обратном порядке, в котором они были помещены. Если стек пуст, то говорят, что это состояние Underflow.

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


class Program {
   static void Main()
   {
      // Creating a stack of integers
      Stack<int> s = new Stack<int>();


      // Pushing elements onto the stack
      s.Push(1); // This pushes 1 to the stack top
      s.Push(2); // This pushes 2 to the stack top
      s.Push(3); // This pushes 3 to the stack top
      s.Push(4); // This pushes 4 to the stack top
      s.Push(5); // This pushes 5 to the stack top


      // Removing elements from the stack using Pop function
      while (s.Count > 0) {
         Console.Write(s.Peek() + " "); // Displaying the top element without removing it
         s.Pop(); // Removes the top element from the stack
      }
   }
}
```

Вывод

5 4 3 2 1

Операция Top в стеке:

Операция Top используется для возврата верхнего элемента стека.

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

class Program
{
   static int TopElement(Stack<int> s)
   {
      return s.Peek();
   }

   static void Main()
   {
      Stack<int> s = new Stack<int>(); // creating a stack of integers

      s.Push(1); // This pushes 1 to the stack top
      Console.WriteLine(TopElement(s)); // Prints 1 since 1 is present at the stack top

      s.Push(2); // This pushes 2 to the stack top
      Console.WriteLine(TopElement(s)); // Prints 2 since 2 is present at the stack top

      s.Push(3); // This pushes 3 to the stack top
      Console.WriteLine(TopElement(s)); // Prints 3 since 3 is present at the stack top
   }
}
```

Вывод

```
1
2
3
```

Операция isEmpty в стеке:

операция isEmpty — это логическая операция, которая используется для определения того, пуст ли стек.

Эта операция вернет true, если стек пуст, в противном случае false.

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

class Program
{
    // Function to check if a stack is empty
    static bool IsEmpty(Stack<int> s)
    {
        return s.Count == 0;
    }

    static void Main()
    {
        Stack<int> s = new Stack<int>();

        // Check if the stack is empty
        if (IsEmpty(s))
        {
            Console.WriteLine("Stack is empty.");
        }
        else
        {
            Console.WriteLine("Stack is not empty.");
        }

        // Push a value (1) onto the stack
        s.Push(1);

        // Check if the stack is empty after pushing a value
        if (IsEmpty(s))
        {
```

```
            Console.WriteLine("Stack is empty.");
        }
        else
        {
            Console.WriteLine("Stack is not empty.");
        }
    }
}
```

Вывод

Stack is empty.

Stack is not empty.

Операция size() в стеке:

Операция size в стеке используется для возврата количества элементов, присутствующих внутри стека.

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

public class Program
{
   public static void Main(string[] args)
   {
      Stack<int> s = new Stack<int>(); // creating a stack of integers

      Console.WriteLine(s.Count); // Prints 0 since the stack is empty

      s.Push(1); // This pushes 1 to the stack top
      s.Push(2); // This pushes 2 to the stack top
      Console.WriteLine(s.Count); // Prints 2 since the stack contains two elements

      s.Push(3); // This pushes 3 to the stack top
      Console.WriteLine(s.Count); // Prints 3 since the stack contains three elements
   }
}
//This code is contribiuted by Kishan.
```

Вывод

0 2 3

2. Алгоритмический анализ

Анализ стека фокусируется на эффективности его базовых операций с точки зрения времени и использования памяти.

Доказательство корректности:

Корректность стека интуитивно понятна и напрямую вытекает из его определения принципа LIFO. Каждая операция четко определена:

PUSH всегда добавляет элемент в точно определенное место (сверху).

POP всегда удаляет элемент из точно такого же места (сверху).

PEEK просто обеспечивает доступ к элементу наверху, не изменяя состояние стека. Таким образом, элементы всегда обрабатываются в обратном порядке их поступления, обеспечивая принцип LIFO. Если базовая реализация (например, использование массива или динамического списка) делает операции добавления/удаления эффективными, стек будет функционировать правильно.

Временная сложность:

- **PUSH (O(1))**: Добавление элемента наверх стека (в конец списка/массива) занимает постоянное время. Это означает, что время выполнения операции PUSH не зависит от количества элементов, уже находящихся в стеке.
- **POP (O(1))**: Удаление элемента с верха стека (с конца списка/массива) также занимает постоянное время.
- **PEEK (O(1))**: Доступ к элементу наверху стека (для достижения последнего элемента в списке/массиве) занимает постоянное время.
- **IS_EMPTY (O(1))**: Проверка на пустое место (проверка длины списка/массива) занимает постоянное время.
- **SIZE (O(1))**: Получение размера стека также является постоянной операцией.

Таким образом, все основные операции со стеком имеют постоянную временную сложность (O(1)), что делает стек очень быстрой и эффективной структурой данных для ситуаций, когда элементы необходимо быстро добавлять и удалять с одного конца стека.

Пространственная сложность:

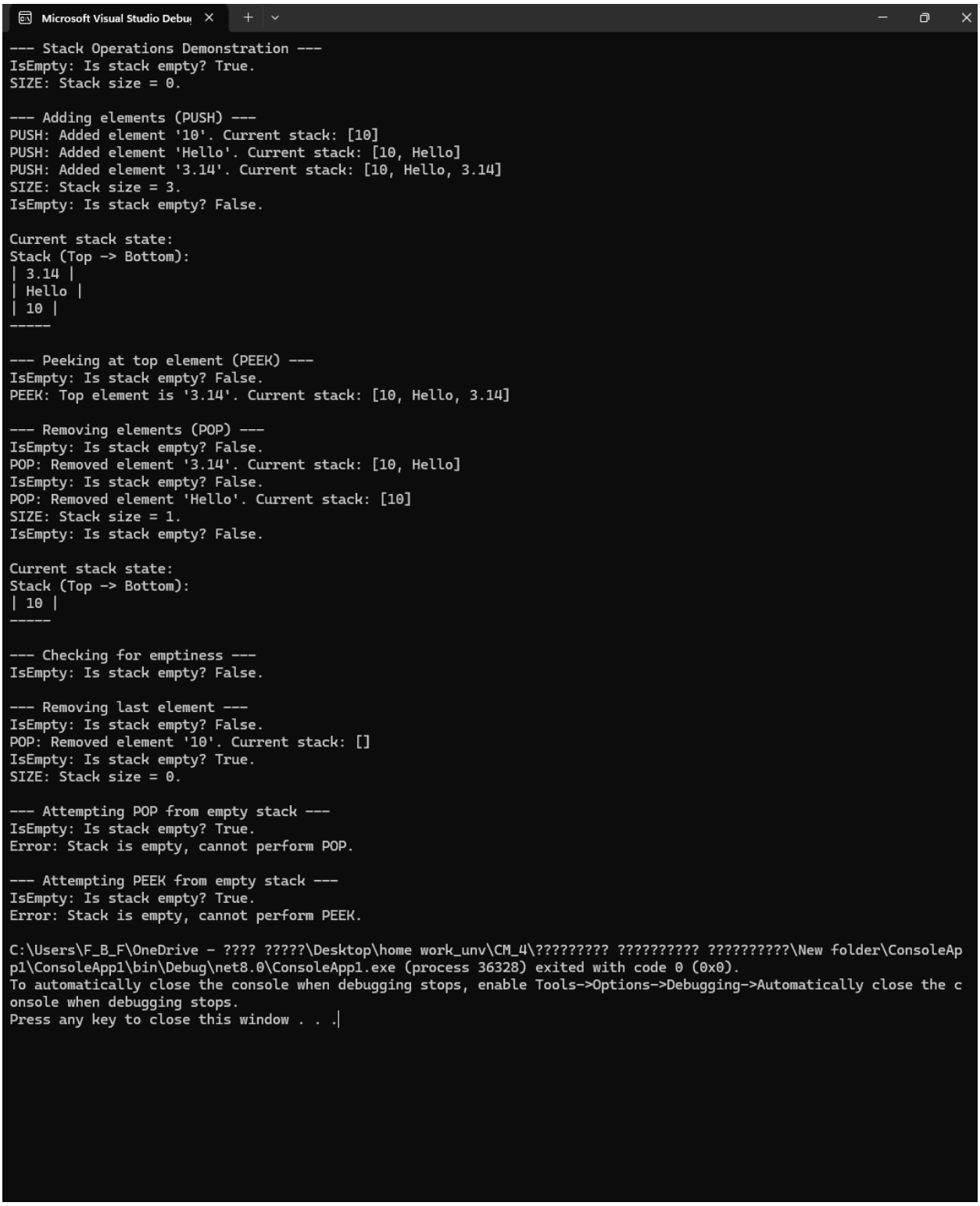
- **O(N)**: Пространственная сложность стека прямо пропорциональна количеству элементов (N), которые он хранит. Каждый элемент требует определенного объема памяти, поэтому для хранения N элементов требуется память, эквивалентная N элементам. Дополнительные затраты на хранение указателя или самой базовой структуры данных обычно незначительны и не зависят

Реализация

на **C#**, используя встроенный класс List<T> (список), который предоставляет методы, аналогичные операциям стека (Add для PUSH и RemoveAt для POP с

использованием индекса последнего элемента). В .NET также есть встроенный класс Stack<T>, но для демонстрации базовой реализации мы создадим свой.


```
using System;
using System.Collections.Generic;
using System.Linq; // For Reverse() and ToList() methods

public class MyStack<T>
{
    private List<T> items;

    /// <summary>
    /// Constructor for MyStack class.
    /// Initializes an empty list to store stack elements.
    /// </summary>
    public MyStack()
    {
        items = new List<T>();
    }

    /// <summary>
    /// Checks if the stack is empty.
    /// Returns true if the stack is empty, otherwise false.
    /// </summary>
    public bool IsEmpty()
    {
        bool isEmpty = items.Count == 0;
        Console.WriteLine($"IsEmpty: Is stack empty? {isEmpty}.");
        return isEmpty;
    }

    /// <summary>
    /// Adds an element to the top of the stack.
    /// </summary>
    /// <param name="item">The element to add.</param>
    public void Push(T item)
    {
        items.Add(item);
        Console.WriteLine($"PUSH: Added element '{item}'. Current stack: [{string.Join(", ", items)}]");
    }

    /// <summary>
    /// Removes and returns the element from the top of the stack.
    /// Throws an InvalidOperationException if the stack is empty.
    /// </summary>
    /// <returns>The removed element.</returns>
    public T Pop()
    {
        if (IsEmpty())
        {
            throw new InvalidOperationException("Error: Stack is empty, cannot perform POP.");
        }
        T poppedItem = items[items.Count - 1]; // Get the last element
        items.RemoveAt(items.Count - 1);       // Remove it
        Console.WriteLine($"POP: Removed element '{poppedItem}'. Current stack: [{string.Join(", ", items)}]");
        return poppedItem;
    }

    /// <summary>
    /// Returns the element at the top of the stack without removing it.
    /// Throws an InvalidOperationException if the stack is empty.
    /// </summary>
    /// <returns>The element at the top of the stack.</returns>
    public T Peek()
    {
        if (IsEmpty())
        {
            throw new InvalidOperationException("Error: Stack is empty, cannot perform PEEK.");
        }
        T topItem = items[items.Count - 1]; // Get the last element
        Console.WriteLine($"PEEK: Top element is '{topItem}'. Current stack: [{string.Join(", ", items)}]");
        return topItem;
    }

    /// <summary>
    /// Returns the current number of elements in the stack.
    /// </summary>
    /// <returns>The number of elements in the stack.</returns>
    public int Size()
    {
        int currentSize = items.Count;
        Console.WriteLine($"SIZE: Stack size = {currentSize}.");
        return currentSize;
    }

    /// <summary>
    /// Returns a string representation of the stack for convenient output.
    /// Elements are displayed in LIFO order (top to bottom).
    /// </summary>
    public override string ToString()
    {
        if (IsEmpty())
        {
            return "Stack: [] (empty)";
        }
        // Create a string representation that mimics a vertical stack
        // Use Reverse() to display in LIFO order
        List<string> reversedItems = items.Select(item => item.ToString()).ToList();
        reversedItems.Reverse();

        string stackRepresentation = "Stack (Top -> Bottom):\n";
        foreach (string itemStr in reversedItems)
        {
            stackRepresentation += $"| {itemStr} |\n";
        }
        stackRepresentation += "'-----'";
        return stackRepresentation;
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("--- Stack Operations Demonstration ---");
        MyStack<object> myStack = new MyStack<object>(); // Use object to demonstrate different types

        myStack.IsEmpty(); // Is stack empty? True
        myStack.Size();    // Stack size: 0

        Console.WriteLine("\n--- Adding elements (PUSH) ---");
        myStack.Push(10);
        myStack.Push("Hello");
        myStack.Push(3.14);
        myStack.Size();    // Stack size: 3

        Console.WriteLine($"\nCurrent stack state:\n{myStack}");

        Console.WriteLine("\n--- Peeking at top element (PEEK) ---");
        myStack.Peek();    // Top element: 3.14

        Console.WriteLine("\n--- Removing elements (POP) ---");
        myStack.Pop();     // Removed element: 3.14
        myStack.Pop();     // Removed element: Hello
        myStack.Size();    // Stack size: 1

        Console.WriteLine($"\nCurrent stack state:\n{myStack}");

        Console.WriteLine("\n--- Checking for emptiness ---");
        myStack.IsEmpty(); // Is stack empty? False

        Console.WriteLine("\n--- Removing last element ---");
        myStack.Pop();     // Removed element: 10
        myStack.IsEmpty(); // Is stack empty? True
        myStack.Size();    // Stack size: 0

        Console.WriteLine("\n--- Attempting POP from empty stack ---");
        try
        {
            myStack.Pop();
        }
        catch (InvalidOperationException e)
        {
            Console.WriteLine(e.Message); // Error: Stack is empty, cannot perform POP.
        }

        Console.WriteLine("\n--- Attempting PEEK from empty stack ---");
        try
        {
            myStack.Peek();
        }
        catch (InvalidOperationException e)
        {
            Console.WriteLine(e.Message); // Error: Stack is empty, cannot perform PEEK.
        }
    }
}
```



Описание реализации на C#:

- **public class MyStack<T>**: Определяет универсальный класс MyStack, который может хранить элементы любого типа T. Это позволяет

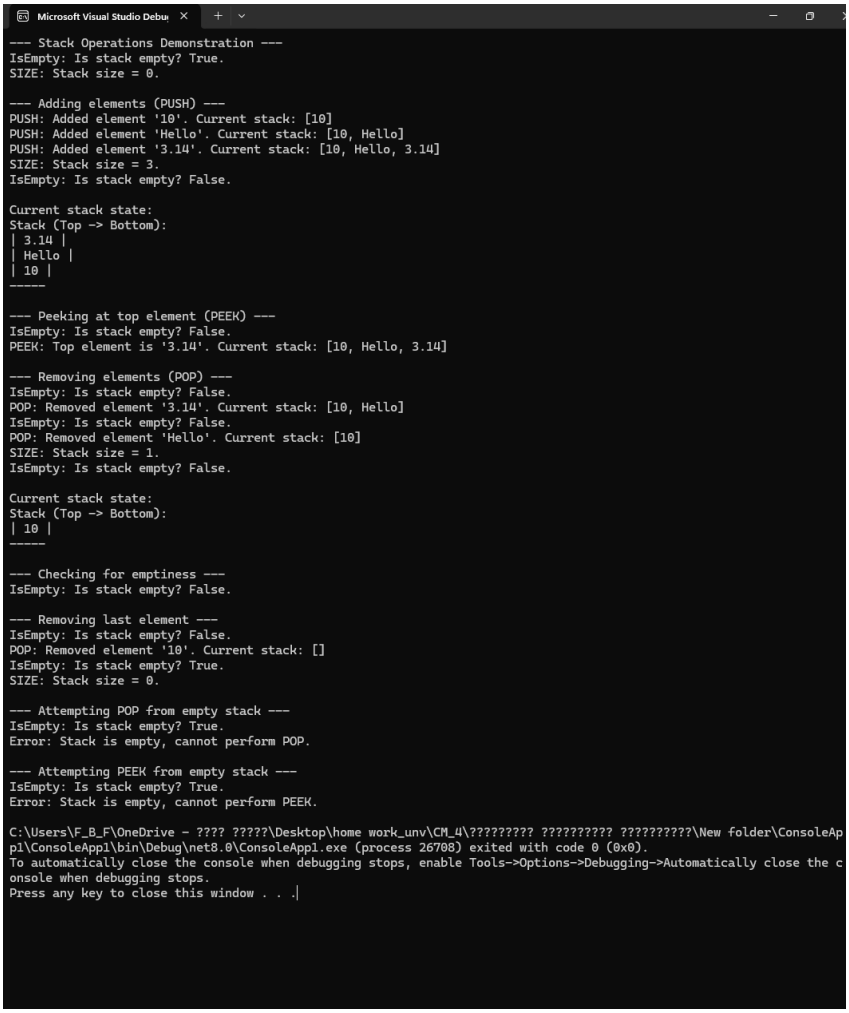
использовать стек для целых чисел, строк, пользовательских объектов и т.д.

- **private List<T> items;**: Внутри класса используется приватный экземпляр System.Collections.Generic.List<T>. Этот список служит базовым хранилищем для элементов стека. В C# List<T> эффективно добавляет и удаляет элементы с конца, что идеально подходит для реализации стека.
- **public MyStack()**: Конструктор класса. При создании нового объекта MyStack он инициализирует items как новый пустой список.
- **public bool IsEmpty()**: Проверяет, содержит ли items какие-либо элементы, используя свойство Count. Возвращает true, если список пуст, указывая на пустой стек. Включает вывод в консоль для отслеживания состояния.
- **public void Push(T item)**: Метод добавляет item в конец списка items с помощью метода Add(). Конец списка List<T> выступает в роли вершины стека. После добавления выводится сообщение о выполненной операции на английском языке.
- **public T Pop()**: Метод удаляет и возвращает элемент с вершины стека.
    - Сначала он проверяет, не пуст ли стек, используя IsEmpty(). Если стек пуст, он выбрасывает InvalidOperationException с английским сообщением, предотвращая ошибку доступа к пустому списку.
    - Если стек не пуст, он получает последний элемент (items[items.Count - 1]) и затем удаляет его с помощью RemoveAt(items.Count - 1).
    - Возвращает удаленный элемент и выводит сообщение в консоль на английском языке.
- **public T Peek()**: Метод возвращает элемент с вершины стека без его удаления. Аналогично Pop(), он проверяет на пустоту и выбрасывает исключение при необходимости с английским сообщением. Возвращает последний элемент списка (items[items.Count - 1]). Включает вывод в консоль на английском языке.

- **public int Size()**: Возвращает количество элементов в списке items (свойство Count), что соответствует текущему размеру стека. Выводит текущий размер на английском языке.
- **public override string ToString()**: Переопределенный метод, который предоставляет удобное строковое представление объекта MyStack. Он форматирует вывод так, чтобы стек был показан вертикально, с верхним элементом сверху, имитируя реальную стопку. Использует Linq's Reverse() для отображения элементов в порядке LIFO. Сообщение "Stack: [] (empty)" также на английском.
- **public class Program и public static void Main(string[] args)**: Это точка входа в консольное приложение C#. Здесь создается экземпляр MyStack<object> (используется object для демонстрации, что стек может хранить разные типы данных) и последовательно выполняются все основные операции стека для демонстрации его работы. Также предусмотрена обработка исключений при попытке извлечь или просмотреть элементы из пустого стека, с английскими сообщениями.

Визуализация работы алгоритма


File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help
Search
ConsoleApp1
Debug
Any CPU
ConsoleApp1
Program.cs
ConsoleApp1
ConsoleVisualizer
MainVisualizer(string[] args)



Пояснения к коду консольной визуализации:

- ConsoleVisualizer класс: Содержит метод DisplayStack, который использует ваш MyStack класс'a ToString() метод для вывода текущего состояния стека в консоль.
- DisplayStack<T>(MyStack<T> stack, string actionDescription): Принимает экземпляр стека и описание действия, затем выводит его в консоль с задержкой, чтобы вы могли пошагово отслеживать изменения.
- System.Threading.Thread.Sleep(1000);: Добавляет паузу в 1 секунду, чтобы изменения были более заметными.
- Main метод: Демонстрирует последовательность операций, вызывая DisplayStack после каждой, чтобы показать изменение стека.

Примеры выполнения

Для демонстрации работы стека и анализа его поведения, рассмотрим следующий сценарий с последовательностью операций.

Используемый код: Реализация MyStack<object> из раздела 3. Входные данные (последовательность операций):

1. Создать пустой стек.
2. Выполнить IsEmpty()
3. Выполнить Push(5)
4. Выполнить Push(15)
5. Выполнить Push(7)
6. Выполнить Peek()
7. Выполнить Pop()
8. Выполнить Peek()
9. Выполнить Push(20)

№	Операция	Состояние Стека (вершина -> основание)	Вывод в консоль / Результат	Комментарий
1	MyStack<object> myStack = new MyStack<object	[]		Инициализация пустого стека.
2	myStack.IsEmpty()	[]	IsEmpty: Стек пуст? True.	Стек действительно пуст.
3	myStack.Push(5)	[5]	PUSH: Добавлен элемент '5'. Текущий стек: [5]	Элемент '5' добавлен на вершину.
4	myStack.Push(15)	[15, 5]	PUSH: Добавлен элемент '15'. Текущий стек: [5, 15]	Элемент '15' добавлен поверх '5'.
5	myStack.Push(7)	[7, 15, 5]	PUSH: Добавлен элемент '7'. Текущий стек: [5, 15, 7]	Элемент '7' добавлен поверх '15'. Теперь '7' - вершина.
6	myStack.Peek()	[7, 15, 5]	PEEK: Верхний элемент '7'. Текущий стек: [5, 15, 7]	Просмотрен верхний элемент. Стек не изменился.
7	myStack.Pop()	[15, 5]	POP: Удален элемент '7'. Текущий стек: [5, 15]	Элемент '7' (последний добавленный) удален. '15' стал вершиной.
8	myStack.Peek()	[15, 5]	PEEK: Верхний элемент '15'. Текущий стек: [5, 15]	Просмотрен новый верхний элемент.
9	myStack.Push(20)	[20, 15, 5]	PUSH: Добавлен элемент '20'. Текущий стек: [5, 15, 20]	Элемент '20' добавлен.
10	myStack.Size()	[20, 15, 5]	SIZE: Размер стека = 3.	Текущий размер стека.
11	myStack.Pop()	[15, 5]	POP: Удален элемент '20'. Текущий стек: [5, 15]	Элемент '20' удален.
12	myStack.Pop()	[5]	POP: Удален элемент '15'. Текущий стек: [5]	Элемент '15' удален.
13	myStack.IsEmpty()	[5]	IsEmpty: Стек пуст? False.	Стек содержит один элемент.
14	myStack.Pop()	[]	POP: Удален элемент '5'. Текущий стек: [] Ошибка: Стек пуст, нее	Сначала удаляется '5', затем при следующей попытке pop возникает оши

Заключение

В рамках данной курсовой работы был детально рассмотрен алгоритм "Стек" – одна из фундаментальных линейных структур данных, оперирующая по принципу LIFO (Last In, First Out). Были подробно описаны его основные операции: PUSH, POP, PEEK, IS_EMPTY и SIZE

Анализ алгоритма показал, что все ключевые операции стека выполняются за **константное время (O(1))**, что делает его чрезвычайно эффективным и быстрым для решения задач, требующих операций добавления и удаления с одного конца.

Пространственная сложность стека прямо пропорциональна количеству хранимых элементов (O(N)).

Для практической демонстрации принципов работы стека была представлена его реализация на языке программирования **C#** с использованием стандартного List<T>

Ссылки

https://github.com/FATHEY12352/Coursework

https://neerc.ifmo.ru/wiki/index.php?title=%D0%A1%D1%82%D0%B5%D0%BA

https://www.sciencedirect.com/topics/computer-science/stack-algorithm

https://www.geeksforgeeks.org/dsa/stack-data-structure/