САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе № 4
по курсу «Алгоритмы и структуры данных»
Тема: Стек, очередь, связанный список.
Вариант 7

Выполнила:
Заботкина М.А.
К3139

Проверил:
Афанасьев А.В.

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

Задача №1. Стек

Реализуйте работу стека. Для каждой операции изъятия элемента выведите ее результат. На вход программе подаются строки, содержащие команды. Каждая строка содержит одну команду. Команда — это либо "+ N", либо "–". Команда "+ N"означает добавление в стек числа N, по модулю не превышающего 10^9 . Команда "–"означает изъятие элемента из стека. Гарантируется, что не происходит извлечения из пустого стека. Гарантируется, что размер стека в процессе выполнения команд не превысит 10^6 элементов.

• Формат входного файла (input.txt). В первой строке входного файла содержится M ($1 \le M \le 10^6$) – число команд. Каждая последующая строка исходного файла содержит ровно одну команду.

• Формат выходного файла (output.txt). Выведите числа, которые удаляются из стека с помощью команды "–", по одному в каждой строке. Числа нужно выводить в том порядке, в котором они были извлечены из стека. Гарантируется, что изъятий из пустого стека не производится.

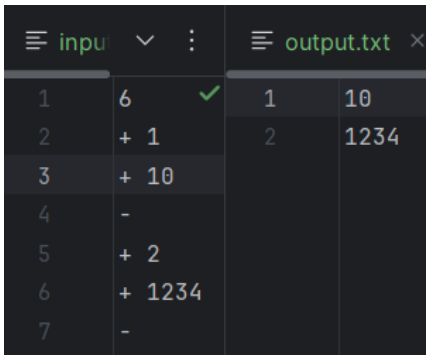
• Ограничение по времени. 2 сек.

• Ограничение по памяти. 256 мб.

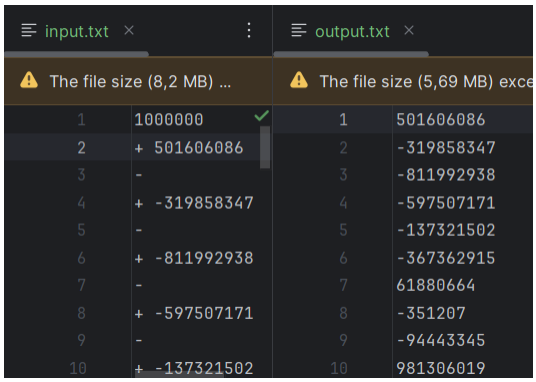
```
def stacks(commands):
    stack = []
    result = []

    for command in commands:
        if command[0] == '+':
            stack.append(int(command.split()[1]))
        elif command == '-':
            result.append(stack.pop())

    return result
```

Функция stacks предназначена для обработки списка команд, управляющих стеком. Она принимает список строк, где каждая строка представляет команду. Команды бывают двух типов: добавление элемента в стек ('+ N', где N — число) и удаление верхнего элемента ('-'). Функция создаёт пустой стек и список result для сохранения удалённых элементов. При обработке команды добавления число извлекается из строки и помещается в стек, а при обработке команды удаления верхний элемент извлекается из стека и добавляется в result. После выполнения всех команд возвращается список result, содержащий элементы в порядке их удаления из стека.

Результат работы кода на примерах из текста задачи:



Результат работы кода на максимальных и минимальных значениях:



	Время выполнения, с	Затраты памяти, Mb
Нижняя граница диапазона значений входных данных из текста задачи	0.0010004043579101562	0.0001373291015625
Пример из задачи	0.0010309219360351562	0.00024890899658203125
Верхняя граница диапазона значений входных данных из текста задачи	0.9925742149353027	17.334403038024902

Вывод по задаче: в рамках данной задачи была реализована работа стека, поддерживающего операции добавления и удаления элементов. Реализация обеспечила корректное выполнение всех команд, поступающих во входном файле, включая обработку команд извлечения с сохранением порядка элементов.

Задача №4. Скобочная последовательность. Версия 2

Определение правильной скобочной последовательности такое же, как и в задаче 3, но теперь у нас больше набор скобок: []{}().

Нужно написать функцию для проверки наличия ошибок при использовании разных типов скобок в текстовом редакторе типа LaTeX. Для удобства, текстовый редактор должен не только информировать о наличии ошибки в использовании скобок, но также указать точное место в коде (тексте) с ошибочной скобочкой.

В первую очередь объявляется ошибка при наличии первой несовпадающей закрывающей скобки, перед которой отсутствует открывающая скобка, или которая не соответствует открывающей, например, ()[} - здесь ошибка укажет на }.

Во вторую очередь, если описанной выше ошибки не было найдено, нужно указать на первую несовпадающую открывающую скобку, у которой отсутствует закрывающая, например, (в ([]. Если не найдено ни одной из указанный выше ошибок, нужно сообщить, что использование скобок корректно.

Помимо скобок, код может содержать большие и маленькие латинские буквы, цифры и знаки препинания. Формально, все скобки в коде (тексте) должны быть разделены на пары совпадающих скобок, так что в каждой паре открывающая скобка идет перед закрывающей скобкой, а для любых двух пар скобок одна из них вложена внутри другой, как в (foo[bar]) или они разделены, как в f(a,b)-g[c]. Скобка [соответствует скобке], соответствует и (соответствует) .

• Формат входного файла (input.txt). Входные данные содержат одну строку S, состоящую из больших и маленьких латинских букв, цифр, знаков препинания и скобок из набора [] (). Длина строки S – 1 ≤ S leq10^5 .

• Формат выходного файла (output.txt). Если в строке S скобки используются правильно, выведите «Success» (без кавычек). В противном случае выведите отсчитываемый от 1 индекс первой несовпадающей закрывающей скобки, а если нет несовпадающих закрывающих скобок, выведите отсчитываемый от 1 индекс первой открывающей скобки, не имеющей закрывающей.

• Ограничение по времени. 5 сек.

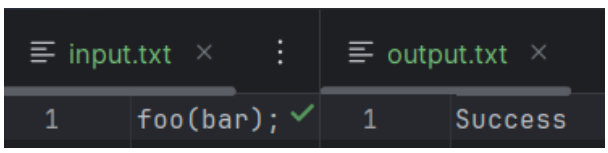
• Ограничение по памяти. 256 мб.

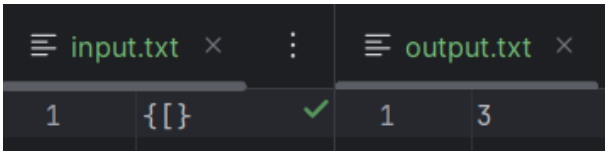
```
def brackets(s):
    stack = []
    pairs = {')': '(', ']': '[', '}': '{'}
```

```
    for i, element in enumerate(s):
        if element in '([{':
            stack.append((element, i + 1))
        elif element in ')]}':
            if not stack:
                return i + 1
            top, index = stack.pop()
            if top != pairs[element]:
                return i + 1
    if stack:
        return stack[-1][1]

    return "Success"
```

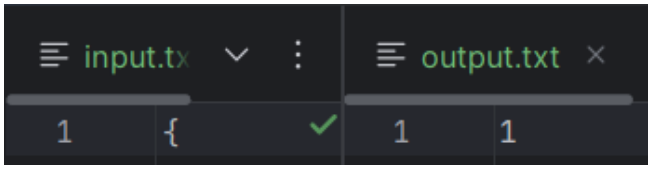
Функция brackets проверяет строку s на правильность расстановки скобок. Она использует стек для отслеживания открывающих скобок и их позиций, а также словарь соответствий, связывающий закрывающие скобки с открывающими. Если встречается открывающая скобка, она добавляется в стек вместе с её позицией. При встрече закрывающей скобки проверяется, пуст ли стек. Если стек пуст, это означает, что у закрывающей скобки нет пары, и возвращается её позиция. Если стек не пуст, извлекается верхний элемент, и проверяется, соответствует ли он текущей закрывающей скобке. Если соответствие нарушено, возвращается позиция ошибки. После обработки строки, если в стеке остаются незакрытые скобки, функция возвращает позицию последней из них. Если все скобки правильно расставлены, возвращается строка "Success".

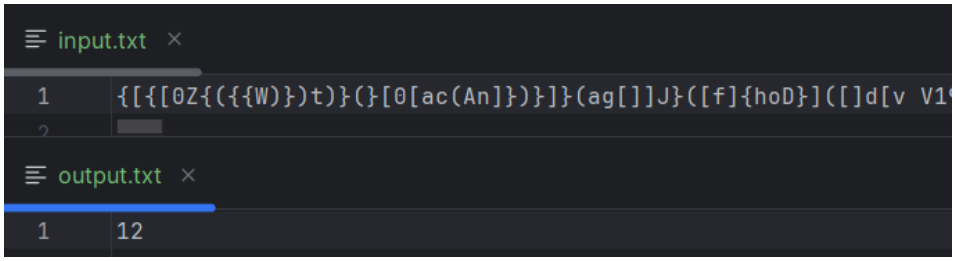
Результат работы кода на примерах из текста задачи:





Результат работы кода на максимальных и минимальных значениях:





	Время выполнения, с	Затраты памяти, Mb
Нижняя граница диапазона значений входных данных из текста задачи	0.0009148120880126953	0.00019073486328125
Пример из задачи	0.0010004043579101562	0.00019073486328125
Верхняя граница диапазона значений входных данных из текста задачи	0.0010082721710205078	0.00019073486328125

Вывод по задаче: реализован алгоритм для проверки корректности расстановки скобок. Алгоритм корректно определяет наличие ошибок и их местоположение, включая обработку различных типов скобок. Проверка основана на использовании стека, что обеспечивает эффективную работу.

Задача №5. Стек с максимумом

Стек - это абстрактный тип данных, поддерживающий операции Push() и Pop(). Нетрудно реализовать его таким образом, чтобы обе эти операции работали за константное время. В этой задаче ваша цель - реализовать стек, который также поддерживает поиск максимального значения и гарантирует, что все операции по-прежнему работают за константное время.

Реализуйте стек, поддерживающий операции Push(), Pop() и Max().

• Формат входного файла (input.txt). В первой строке входного файла содержится n (1 ≤ n ≤ 400000) – число команд. Последующие n строк исходного файла содержит ровно одну команду: push V, pop или max. 0 ≤ V ≤ 10^5.

• Формат выходного файла (output.txt). Для каждого запроса max выведите (в отдельной строке) максимальное значение стека.

• Ограничение по времени. 5 сек.

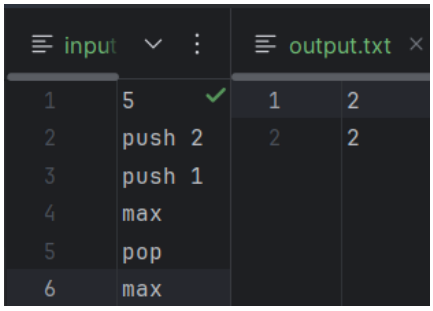
• Ограничение по памяти. 512 мб.

```
def stack_max(commands):
    stack = []
    max_stack = []
    result = []

    for command in commands:
        if command[0] == 'push':
```

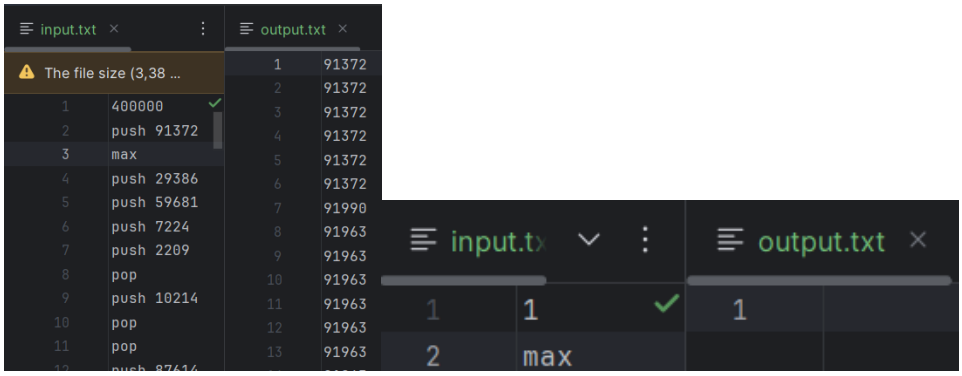
```
            value = int(command[1])
            stack.append(value)
            if not max_stack or value >= max_stack[-1]:
                max_stack.append(value)
        elif command[0] == 'pop':
            if stack:
                value = stack.pop()
                if value == max_stack[-1]:
                    max_stack.pop()
        elif command[0] == 'max':
            if max_stack:
                result.append(str(max_stack[-1]))

    return result
```

Функция stack_max обрабатывает команды для работы со стеком и позволяет эффективно отслеживать максимальный элемент. Она использует два стека: основной stack, в котором хранятся добавляемые элементы, и дополнительный max_stack, который отслеживает текущий максимум. При выполнении команды push x элемент x добавляется в основной стек, а если стек максимумов пуст или элемент больше либо равен текущему максимуму, он также добавляется в max_stack. При выполнении команды pop верхний элемент удаляется из основного стека, и если он равен текущему максимуму, то удаляется и из стека максимумов. Команда max добавляет в список result текущее максимальное значение, которое всегда находится на вершине стека максимумов.

Результат работы кода на примерах из текста задачи:



Результат работы кода на минимальных и максимальных значениях:



	Время выполнения, с	Затраты памяти, Mb
Нижняя граница диапазона значений входных данных из текста задачи	0.0009999275207519531	4.57763671875e-05
Пример из задачи	0.0010876655578613281	0.000263214111328125
Верхняя граница диапазона значений входных данных из текста задачи	0.46149396896362305	9.534236907958984

Вывод по задаче: реализован стек, поддерживающий операции добавления, удаления и поиска максимального элемента за константное время. Использование дополнительного стека для отслеживания текущего максимума обеспечило необходимую производительность.

Задача №7. Максимум в движущейся последовательности

Задан массив из n целых чисел - a1, ..., an и число m < n, нужно найти максимум среди последовательности ("окна") {ai , ..., ai+m−1} для каждого значения 1 ≤ i ≤ n − m + 1. Простой алгоритм решения этой задачи за O(nm) сканирует каждое "окно"отдельно.

Ваша цель - алгоритм за O(n).

• Формат входного файла (input.txt). В первой строке содержится целое число n (1 ≤ n ≤ 10^5) – количество чисел в исходном массиве, вторая строка содержит n целых чисел a_1, ..., a_n этого массива, разделенных пробелом (0 ≤ a_i ≤ 10^5). В третьей строке - целое число m - ширина "окна"(1 ≤ m ≤ n).

• Формат выходного файла (output.txt). Нужно вывести max a_i , ..., a_i+m−1 для каждого 1 ≤ i ≤ n − m + 1.

• Ограничение по времени. 5 сек.

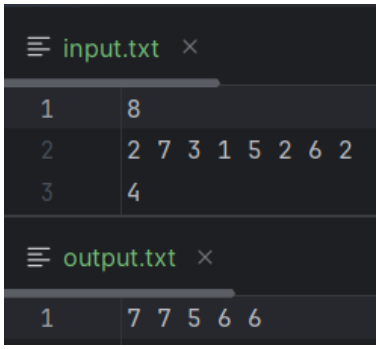
• Ограничение по памяти. 512 мб.

```
def find_max(n, arr, m):
    dequeue = []
    result = []

    for i in range(n):
        if dequeue and dequeue[0] < i - m + 1:
            dequeue.pop(0)
        while dequeue and arr[dequeue[-1]] < arr[i]:
            dequeue.pop()
        dequeue.append(i)
```

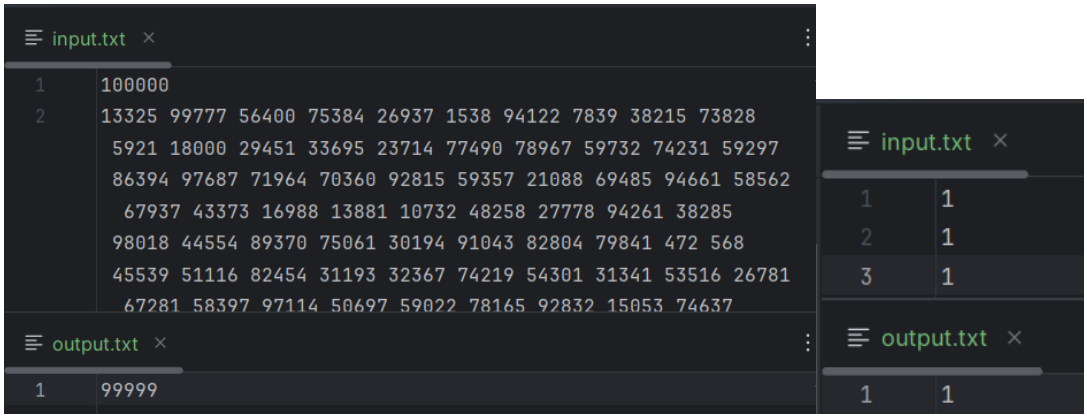
```
        if i >= m - 1:
            result.append(arr[dequeue[0]])

    return result
```

Код работает с деком для эффективного нахождения максимумов в каждом окне длины m массива. Переменная dequeue используется для хранения индексов элементов массива arr, которые могут быть максимумами в текущем окне. На каждом шаге проверяется, есть ли в dequeue индексы элементов, вышедших за пределы текущего окна (индексы меньше, чем i - m + 1), и такие индексы удаляются из начала dequeue. Затем из конца dequeue удаляются индексы элементов, значения которых в массиве arr меньше текущего элемента arr[i], так как они больше не могут быть максимумами в текущем окне. После этого текущий индекс i добавляется в конец dequeue. Если текущее положение в массиве соответствует концу окна (индекс i не меньше m - 1), в список результатов result добавляется значение массива arr по индексу, который находится в начале dequeue, так как этот индекс соответствует максимальному элементу текущего окна.

Результат работы кода на примерах из текста задачи:



Результат работы кода на максимальных и минимальных значениях:



	Время выполнения, с	Затраты памяти, Mb
Нижняя граница диапазона значений входных данных из текста задачи	0.0009646415710449219	9.1552734375e-05
Пример из задачи	0.0009865760803222656	0.0001220703125
Верхняя граница диапазона значений входных данных из текста задачи	0.28409242630004883	0.001129150390625

Вывод по задаче: реализован алгоритм поиска максимума в каждом окне фиксированного размера в массиве. Алгоритм основан на использовании дека, что позволило достичь сложности O(n). Реализация продемонстрировала высокую эффективность на больших объемах данных.

Дополнительные задачи

Задача №8. Постфиксная запись

*В постфиксной записи (или обратной польской записи) операция записывается после двух операндов. Например, сумма двух чисел A и B записывается как A B +. Запись B C + D * обозначает привычное нам (B + C) * D, а запись A B C + D * + означает A + (B + C) * D. Достоинство постфиксной записи в том, что она не требует скобок и дополнительных соглашений о приоритете операторов для своего чтения. Дано выражение в обратной польской записи. Определите его значение.*

*• Формат входного файла (input.txt). В первой строке входного файла дано число N ($1 \le n \le 10^6$) – число элементов выражения. Во второй строке содержится выражение в постфиксной записи, состоящее из N элементов. В выражении могут содержаться неотрицательные однозначные числа и операции +, -, *. Каждые два соседних элемента выражения разделены ровно одним пробелом.*

• Формат выходного файла (output.txt). Необходимо вывести значение записанного выражения. Гарантируется, что результат выражения, а также результаты всех промежуточных вычислений, по модулю будут меньше, чем 2^{31}.

• Ограничение по времени. 2 сек.

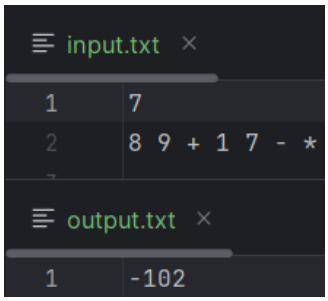
• Ограничение по памяти. 256 мб

```
def postfix(data):
    stack = []

    for element in data:
        if element.isdigit() or (element[0] == '-' and len(element) > 1):
            stack.append(int(element))
        else:
            b = stack.pop()
            a = stack.pop()
            if element == '+':
                stack.append(a + b)
            elif element == '-':
                stack.append(a - b)
            elif element == '*':
                stack.append(a * b)

    return stack[0]
```

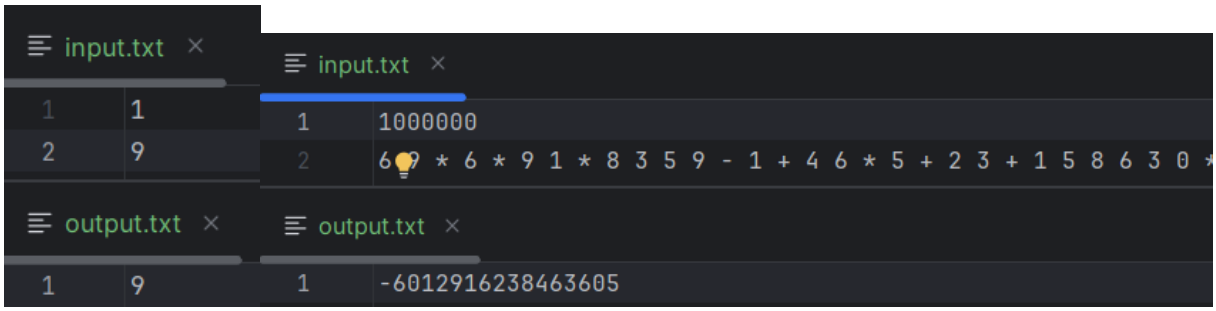
Для вычислений используется стек stack, который временно хранит операнды и промежуточные результаты. Функция проходит по каждому элементу входного списка data. Если элемент является числом, определяемым с помощью метода isdigit или проверки на знак - для

отрицательных чисел, оно преобразуется в целое число и добавляется в стек. Если элемент — это оператор (+, -, *), из стека извлекаются два последних значения: сначала второй операнд b, затем первый операнд a. Выполняется соответствующая операция над a и b, после чего результат добавляется обратно в стек. В конце, после обработки всех элементов, в стеке остаётся одно значение — результат вычисления, который возвращается функцией.

Результат работы кода на примерах:



Результат работы кода на максимальных и минимальных значениях:



	Время выполнения, с	Затраты памяти, Mb
Нижняя граница диапазона значений входных данных из текста задачи	0.0010058879852294922	7.62939453125e-05
Пример из задачи	0.0008885860443115234	0.0001373291015625
Верхняя граница диапазона значений входных данных из текста задачи	0.668757438659668	0.017482757568359375

Вывод по задаче: реализован алгоритм вычисления выражений в обратной польской записи с использованием стека. Решение корректно

обрабатывает арифметические операции и возвращает правильный результат.

Задача №13. Реализация стека, очереди и связанных списков

1. Реализуйте стек на основе связного списка с функциями isEmpty, push, pop и вывода данных.

2. Реализуйте очередь на основе связного списка функциями Enqueue, Dequeue с проверкой на переполнение и опустошения очереди.

Очередь

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Queue:
    def __init__(self, capacity=None):
        self.head = None
        self.tail = None
        self.capacity = capacity
        self.size = 0

    def is_empty(self):
        return self.head is None

    def is_full(self):
        return self.capacity is not None and self.size >= self.capacity

    def enqueue(self, data):
        if self.is_full():
            print("Queue is full")
            return
        new_node = Node(data)
        if self.is_empty():
            self.head = self.tail = new_node
        else:
            self.tail.next = new_node
            self.tail = new_node
        self.size += 1
        print(f"Added: {data}")

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Queue is empty!")
        removed_data = self.head.data
        self.head = self.head.next
        if self.head is None:
            self.tail = None
        self.size -= 1
        return removed_data

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" ")
            current = current.next
        print("None")
```

Этот код реализует очередь с помощью связного списка. Очередь работает по принципу FIFO (первым пришёл — первым вышел). Класс Node представляет узел очереди, который содержит данные и ссылку на следующий узел. Класс Queue управляет самой очередью, используя два указателя: head (первый элемент) и tail (последний элемент), а также переменную size, которая отслеживает количество элементов в очереди. Класс Queue имеет несколько методов. Метод is_empty проверяет, пуста ли очередь, возвращая True, если head равен None. Метод is_full проверяет, достигла ли очередь своей ёмкости, если она была указана при инициализации. Метод enqueue добавляет элемент в конец очереди. Если очередь заполнена, выводится сообщение "Queue is full". Если очередь пуста, новый узел становится и первым, и последним элементом. В противном случае элемент добавляется в конец очереди, а указатель tail обновляется. Метод dequeue удаляет элемент из начала очереди. Если очередь пуста, выбрасывается ошибка. Удаляется первый узел, и указатель head перемещается на следующий элемент. Если после этого очередь становится пустой, tail становится равным None. Метод возвращает данные удалённого элемента. Метод display выводит все элементы очереди от первого до последнего, разделяя их пробелами, а после всех элементов выводится None.

Стек

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Stack:
    def __init__(self):
        self.top = None

    def is_empty(self):
        return self.top is None

    def push(self, data):
        new_node = Node(data)
        new_node.next = self.top
        self.top = new_node

    def pop(self):
        if self.is_empty():
            return "Stack is empty"
        data = self.top.data
        self.top = self.top.next
        return data

    def display(self):
        current = self.top
        while current:
```

```
            print(current.data, end=" ")
            current = current.next
        print("None")
```

Этот код реализует стек с помощью связного списка, где элементы добавляются и удаляются по принципу LIFO (последним пришёл — первым вышел). Класс Node представляет узел в связном списке, который хранит данные и ссылку на следующий узел, инициализируя её как None. Класс Stack реализует сам стек с рядом методов. Метод __init__ инициализирует стек, устанавливая указатель top на None, что означает, что стек пуст. Метод is_empty проверяет, пуст ли стек, возвращая True, если top равен None. Метод push добавляет новый элемент в стек, создавая новый узел, который ссылается на текущий верхний элемент стека, а затем обновляет top, чтобы новый узел стал верхним. Метод pop удаляет верхний элемент из стека и возвращает его значение, если стек не пуст. Если стек пуст, возвращается сообщение "Stack is empty". Метод display выводит все элементы стека от верхнего к нижнему. Для этого он перебирает все узлы, начиная с верхнего элемента, и выводит их данные, пока не достигнет конца списка. После этого выводится None, чтобы показать конец стека.

Вывод по задаче: реализация стека и очереди через связные списки продемонстрировала гибкость и эффективность этих структур данных. Стек был реализован с поддержкой операций добавления, удаления и отображения элементов, работающих по принципу LIFO, а очередь — с реализацией механизма FIFO, включая контроль переполнения и пустоты. Использование связного списка обеспечило динамическое управление памятью без ограничения размера структуры, что позволило стабильно обрабатывать данные различного объема. Задача продемонстрировала практическую ценность этих подходов, углубив понимание динамических структур данных и их роли в построении сложных алгоритмов.

Вывод

Лабораторная работа была посвящена изучению и реализации базовых структур данных: стека, очереди и связного списка. Выполнение задач позволило освоить их принципы работы, включая управление элементами, использование памяти и применение для решения различных алгоритмических проблем. Реализация этих структур продемонстрировала их эффективность при обработке данных, проверке корректности последовательностей и выполнении вычислений. Полученные навыки закладывают прочную основу для дальнейшего изучения алгоритмов, оптимизации их работы и применения в сложных вычислительных системах.