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 1

Выполнил:
Бен Шамех Абделазиз
Группа: K3239

Проверила:
Ромакина Оксана Михайловна

Санкт-Петербург
2025 г.

Содержание

1 Задача 1. Стек

1.1 Условие

Реализуйте работу стека. Для каждой операции извлечения элемента («-») выведите результат. Элементы добавляются командой «+ N».

1.2 Листинг кода

```python
import sys

def stack_processing():
    input_data = sys.stdin.read().split()
    if not input_data: return
    iterator = iter(input_data)
    next(iterator) #
    stack = []
    result = []
    try:
        for comm in iterator:
            if comm == '+':
                stack.append(next(iterator))
            elif comm == '-':
                result.append(stack.pop())
    except StopIteration: pass
    sys.stdout.write('\n'.join(result))
```

1.3 Текстовое объяснение

Используем стандартный список Python как стек. Операция `append` добавляет элемент на вершину (конец списка), а `pop` удаляет и возвращает его. Оба метода работают за амортизированное время $O(1)$. Буферизованный ввод через `stdin.read` критически важен для обработки 10^6 команд.

1.4 Результаты выполнения

Пример: Ввод: `6, + 1, + 10, -, + 2, + 1234, -` → Вывод: `10, 1234`.

Сценарий	**Время (сек)**	**Память (Mb)**
Нижняя граница	0.00003 sec	4.10 Mb
Пример из задачи	0.00005 sec	4.10 Mb
Верхняя граница (n=10^6)	0.31201 sec	38.50 Mb

2 Задача 2. Очередь

2.1 Условие

Реализуйте работу очереди. Для каждой операции извлечения («-») выведите ее результат.

2.2 Листинг кода

```python
class Queue:
    def __init__(self, n: int):
        self.queue = [None] * n
        self.head = self.tail = self.count_el = 0; self.n = n

    def append(self, item):
        if self.count_el < self.n:
            self.queue[self.tail] = item
            self.tail = (self.tail + 1) % self.n
            self.count_el += 1

    def pop(self):
        if self.count_el > 0:
            temp_per = self.queue[self.head]
            self.head = (self.head + 1) % self.n
            self.count_el -= 1
            return temp_per
```

2.3 Текстовое объяснение

Создаем класс циклической очереди на базе статического массива. Добавление и удаление элементов происходит за счет перемещения указателей `head` и `tail`. Использование оператора `%` обеспечивает «зацикливание» указателей при достижении конца массива.

2.4 Результаты выполнения

Сценарий	**Время (сек)**	**Память (Mb)**
Нижняя граница	0.00046 sec	14.70 Mb
Пример из задачи	0.00099 sec	14.89 Mb
Верхняя граница	0.75439 sec	34.30 Mb

3 Задача 3. Скобочная последовательность. Версия 1

3.1 Условие

Проверить, является ли последовательность скобок вида «()» и «[]» правильной (ПСП).

3.2 Листинг кода

```python
def valid_brackets(s):
    stack = []
    for ch in s:
        if ch in "([":
            stack.append(ch)
        elif ch in ")]":
            if not stack: return False
            top = stack.pop()
            if (ch == ")" and top != "(") or (ch == "]" and top != "["):
                return False
    return len(stack) == 0
```

3.3 Текстовое объяснение

Для проверки используем стек. Открывающие скобки пушатся в стек. При встрече закрывающей скобки проверяем вершину стека: если типы не соответствуют или стек пуст — это ошибка. В конце работы стек должен быть пустым.

3.4 Результаты выполнения

Пример: `([)]` → `NO`; `([])` → `YES`.

Сценарий	**Время (сек)**	**Память (Mb)**
Пример из задачи	0.0061 sec	14.89 Mb
Верхняя граница	0.67889 sec	34.50 Mb

4 Задача 6. Очередь с минимумом

4.1 Условие

Реализовать очередь, поддерживающую стандартные операции и запрос минимального элемента за $O(1)$.

4.2 Листинг кода

```python
from collections import deque

class QueueMin:
    def __init__(self, n: int):
        self.queue = [None] * n
        self.min_deque = deque() #

        self.head = self.tail = self.count_el = 0; self.n = n

    def append(self, item):
        self.queue[self.tail] = item
        self.tail = (self.tail + 1) % self.n
        self.count_el += 1
        while self.min_deque and self.min_deque[-1] > item:
            self.min_deque.pop()
        self.min_deque.append(item)

    def pop(self):
        val = self.queue[self.head]
        if self.min_deque and val == self.min_deque[0]:
            self.min_deque.popleft()
        self.head = (self.head + 1) % self.n
        self.count_el -= 1
        return val
```

4.3 Текстовое объяснение

Для поиска минимума за константное время используется вспомогательный монотонный дек. При добавлении элемента в очередь мы удаляем из конца дека все элементы, которые больше текущего. Таким образом, в начале дека всегда находится минимальный элемент текущей очереди.

4.4 Результаты выполнения

Сценарий	Время (сек)	Память (Mb)
Пример из задачи	0.00627 sec	15.23 Mb
Верхняя граница	0.65656 sec	35.43 Mb

5 Задача 7. Максимум в движущейся последовательности

5.1 Условие

Для массива a и ширины окна m найти максимум в каждом окне за $O(n)$.

5.2 Листинг кода

```python
from collections import deque

def max_slide_window(arr, w_len):
    ans_arr = []
    deq = deque()
    for ind, item in enumerate(arr):
        while deq and arr[deq[-1]] <= item:
            deq.pop()
        deq.append(ind)
        if deq[0] == ind - w_len:
            deq.popleft()
        if ind + 1 >= w_len:
            ans_arr.append(arr[deq[0]])
    return ans_arr
```

5.3 Текстовое объяснение

Алгоритм использует монотонный дек для хранения индексов. Мы поддерживаем дек таким образом, чтобы значения по индексам были убывающими. Индекс текущего максимума окна всегда находится в `deq[0]`. При каждом шаге мы удаляем из дека индексы, вышедшие за пределы окна. Сложность алгоритма — $O(n)$.

5.4 Результаты выполнения

Сценарий	**Время (сек)**	**Память (Mb)**
Пример из задачи	0.00643 sec	14.80 Mb
Верхняя граница	0.70423 sec	32.34 Mb

6 Задача 10. Очередь в пекарню

6.1 Условие

Моделирование работы пекарни с учетом времени прихода и степени нетерпения покупателей.

6.2 Текстовое объяснение

Анализируя время прихода и терпение покупателей, мы моделируем обслуживание в очереди. Если покупатель видит, что перед ним больше людей, чем он готов ждать, он уходит сразу. Очередь обрабатывается по принципу FIFO с шагом в 10 минут на человека.

6.3 Результаты выполнения

Сценарий	**Время (сек)**	**Память (Mb)**
Пример из задачи	0.0087 sec	15.19 Mb
Верхняя граница	0.69952 sec	31.48 Mb

7 Задача 13. Реализация через связанные списки

7.1 Листинг кода (Стек)

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

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

    def push(self, data):
        temp_node = Node(data)
        temp_node.next = self.last
        self.last = temp_node

    def pop(self):
        if self.last:
            res = self.last.data
            self.last = self.last.next
            return res
        raise IndexError("Empty stack")
```

7.2 Текстовое объяснение

Реализован стек на базе односвязного списка. В отличие от реализации на массиве, здесь не требуется перевыделение памяти, а каждая операция `push` и `pop` занимает строго константное время $O(1)$.

7.3 Результаты выполнения

Сценарий	**Время (сек)**	**Память (Mb)**
Пример из задачи	0.00275 sec	14.91 Mb
Верхняя граница	0.55322 sec	30.11 Mb

8 Вывод

1. В ходе лабораторной работы были реализованы фундаментальные структуры данных: стек и очередь (как на базе массивов, так и через связанные списки).

2. Применение монотонных деков позволило решить задачи поиска экстремумов (минимум в очереди, максимум в окне) за амортизированное время $O(1)$ для каждого элемента, что дает общую линейную сложность $O(n)$.

3. Практически подтверждено, что выбор правильной структуры данных и алгоритма критически влияет на возможность обработки больших объемов данных в рамках временных лимитов.