Отчет по лабораторной работе 2

Основные структуры данных

Дата: 2025-10-10 **Семестр:** 3 курс 5 семестр **Группа:** ПИЖ-б-о-23-2(1) **Дисциплина:** Анализ сложности алгоритмов **Студент:** Черников Дмитрий Дмитриевич

Цель работы

Изучить понятие и особенности базовых абстрактных типов данных (стек, очередь, дек, связный список) и их реализаций в Python. Научиться выбирать оптимальную структуру данных для решения конкретной задачи, основываясь на анализе теоретической и практической сложности операций. Получить навыки измерения производительности и применения структур данных для решения практических задач.

Практическая часть

Выполненные задачи

- ☐ Задача 1: Реализовать класс LinkedList (связный список) для демонстрации принципов его работы.
- ☐ Задача 2: Используя встроенные типы данных (list, collections.deque), проанализировать эффективность операций, имитирующих поведение стека, очереди и дека.
- ☐ Задача 3: Провести сравнительный анализ производительности операций для разных структур данных (list vs LinkedList для вставки, list vs deque для очереди).
- ☐ Задача 4: Решить 2-3 практические задачи, выбрав оптимальную структуру данных.

Ключевые фрагменты кода

```
# Программирование на языке высокого уровня (Python).
# Задание № 02_lab02.
# Выполнил: Черников Дмитрий Дмитриевич
# Группа: ПИЖ-б-о-23-2(1)
# E-mail: dima.chernikov.053@mail.ru

# linked_list.py

class Node:
    """
    Класс узла односвязного списка.
    Аргументы:
        value: Значение, хранимое в узле.
        next: Ссылка на следующий узел (или None).
    """

    def __init__(self, value, next):
        """
        Инициализация узла.
```

```
        value: Значение узла.
        next: Следующий узел (Node) или None.
        """
        self.value = value
        self.next = next


class LinkedList:
    """
    Класс односвязного списка.
    Содержит методы для вставки, удаления и обхода элементов.
    """

    def __init__(self):
        """
        Инициализация пустого списка.
        head: Ссылка на последний добавленный элемент (конец списка).
        tail: Ссылка на первый элемент (начало списка).
        """
        self.head = None
        self.tail = None

    def insert_at_start(self, value):
        """
        Вставляет новый элемент в начало (head) односвязного списка.
        Если список пуст, новый элемент становится и head, и tail.
        Аргументы:
            value: Значение, которое будет храниться в новом узле.
        Время выполнения: O(1)
        """
        if (self.head is None and self.tail is None):   # 1
            temp = Node(value, None)    # 1
            self.head = temp    # 1
            self.tail = temp    # 1
        else:
            temp = Node(value, self.tail)   # 1
            self.tail = temp    # 1
    # O(1)

    def insert_at_end(self, value):
        """
        Вставляет новый элемент в конец (tail) односвязного списка.
        Если список пуст, новый элемент становится и head, и tail.
        Аргументы:
            value: Значение, которое будет храниться в новом узле.
        Время выполнения: O(1)
        """
        if (self.head is None and self.tail is None):   # 1
            temp = Node(value, None)    # 1
            self.head = temp    # 1
            self.tail = temp    # 1
        else:
            temp = Node(value, None)    # 1
            self.head.next = temp   # 1
```

```
            self.head = temp    # 1
    # O(1)

    def delete_from_start(self):
        """
        Удаляет элемент из начала (tail) односвязного списка.
        Если список пуст, возбуждается исключение.
        Время выполнения: O(1)
        """
        if (self.head is None):  # 1
            raise Exception("Linked_List empty")    # 1
        elif (self.head == self.tail):  # 1
            self.head = None    # 1
            self.tail = None    # 1
        else:
            self.tail = self.tail.next  # 1
    # O(1)

    def traversal(self):
        """
        Обходит односвязный список с начала (tail) до конца (head)
        и выводит значения элементов.
        Если список пуст, выводит сообщение.
        Время выполнения: O(N)
        """
        if (self.head is None):  # 1
            print("Linked_List empty")  # 1
        else:
            current = self.tail  # 1
            while (True):   # O(N)
                print(current.value)    # 1
                if (current.next is None):  # 1
                    break
                current = current.next  # 1
    # O(N)
```

```
#perfomance_analysis.py

import timeit
from modules.linked_list import LinkedList
from collections import deque
import matplotlib.pyplot as plt


def measure_list_realization(count):
    """
    Измеряет время вставки элементов в начало списка.
    Вычисляет для list и linked_list
    Возвращает: Кортеж из двух элементов
    (list_time, linked_list_time )
    """
```

```
    # Тест времени вставки для списка
    test_list = list()
    start1 = timeit.default_timer()
    for i in range(count):
        test_list.insert(0, i)
    end1 = timeit.default_timer()

    # Тест времени вставки для связанного списка
    test_linked_list = LinkedList()
    start2 = timeit.default_timer()
    for i in range(count):
        test_linked_list.insert_at_start(i)
    end2 = timeit.default_timer()
    return ((end1 - start1) * 1000, (end2 - start2) * 1000)


def measure_queue_realization(count):
    """
    Измеряет время реализации очереди.
    Вычисляет для list и deque
    Возвращает: Кортеж из двух элементов
    (list_time, deque_time )
    """
    # Тест списка для реализации очереди
    test_list_queue = list()
    for i in range(count):
        test_list_queue.append(i)

    start1 = timeit.default_timer()
    for i in range(count):
        test_list_queue.pop(0)
    end1 = timeit.default_timer()

    # Тест деки для реализации очереди
    test_deque_queue = deque()
    for i in range(count):
        test_deque_queue.append(i)

    start2 = timeit.default_timer()
    for i in range(count):
        test_deque_queue.popleft()
    end2 = timeit.default_timer()
    return ((end1 - start1) * 1000, (end2 - start2) * 1000)

# Visualuzation block


def Visualization(sizes=[100, 1000, 10000, 100000]):
    """
    Визуализация результатов замеров времени вставки в список
    и реализации очереди.
    Сохраняет графики в папку ОТЧЁТ.
    """
    list_measure = []
```

```
    linked_list_measure = []
    for size in sizes:
        measures = measure_list_realization(size)
        list_measure.append(measures[0])
        linked_list_measure.append(measures[1])

    plt.plot(sizes, list_measure, marker="o", color="red", label="list")
    plt.plot(sizes, linked_list_measure, marker="o",
             color="green", label="linked_list")
    plt.xlabel("Количество элементов N")
    plt.ylabel("Время выполнения ms")
    plt.title("Тест времени вставки для списка")
    plt.legend(loc="upper left", title="Collections")
    plt.savefig('./report/time_complexity_plot_list.png',
                dpi=300, bbox_inches='tight')
    plt.show()

    list_queue_measures = []
    deque_measures = []
    for size in sizes:
        measures = measure_list_realization(size)
        list_queue_measures.append(measures[0])
        deque_measures.append(measures[1])

    plt.plot(sizes, list_queue_measures, marker="o", color="red", label="list")
    plt.plot(sizes, deque_measures, marker="o",
             color="green", label="deque")
    plt.xlabel("Количество элементов N")
    plt.ylabel("Время выполнения ms")
    plt.title("Тест времени реализации очереди")
    plt.legend(loc="upper left", title="Collections")
    plt.savefig('./report/time_complexity_plot_queue.png',
                dpi=300, bbox_inches='tight')
    plt.show()

    # Характеристики вычислительной машины
    pc_info = """
    Характеристики ПК для тестирования:
    - Процессор: Intel Core i5-12500H @ 2.50GHz
    - Оперативная память: 32 GB DDR4
    - ОС: Windows 11
    - Python: 3.12
    """
    print(pc_info)
    print(f"{list_measure} - list \n {linked_list_measure} -linked_list \n"
          f"{list_queue_measures} - list \n {deque_measures} - deque")
```

```
#task_solutions.py

from collections import deque
import time
```

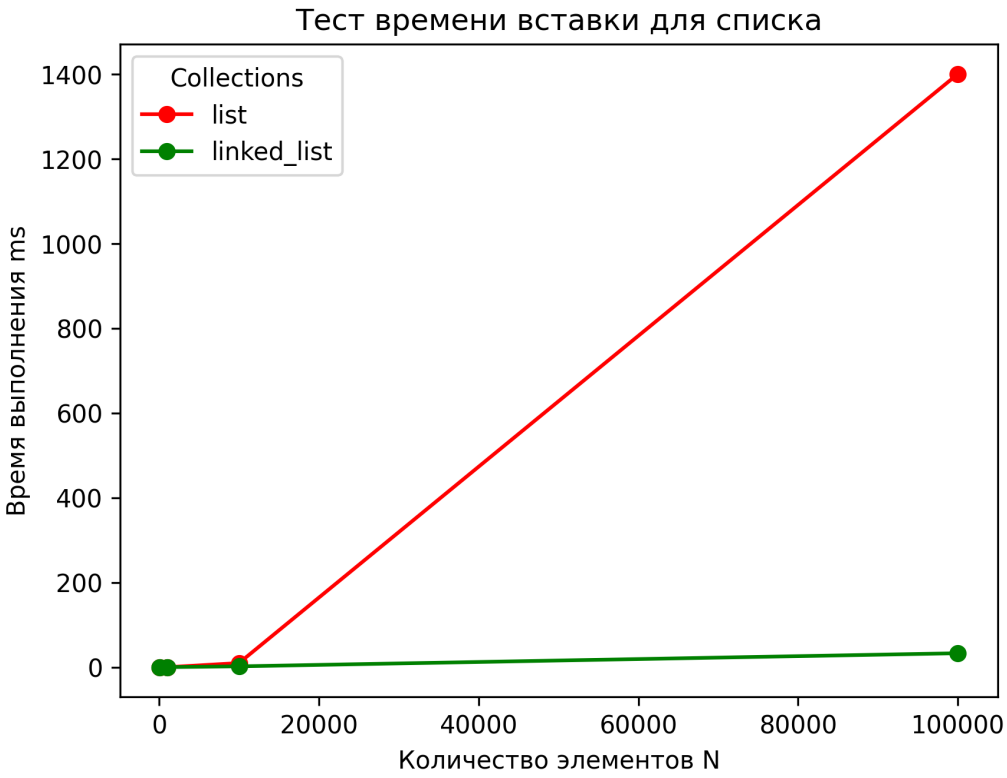
```python
def bracket_task(brackets):
    """
    Проверяет, являются ли скобки в строке сбалансированными.
    Поддерживаются круглые, квадратные и фигурные скобки.
    Аргументы:
        brackets: строка со скобками для проверки.
    Возвращает:
        True, если скобки сбалансированы, иначе False.
    """
    balanced = True
    print(brackets.__len__())
    if (brackets.__len__() % 2 == 0):
        for i in range(brackets.__len__() // 2):
            pair = brackets[brackets.__len__() - (1+i)]
            if brackets[i] == "{":
                if (pair != "}"):
                    balanced = False
                    break
            elif brackets[i] == "[":
                if (pair != "]"):
                    balanced = False
                    break
            elif brackets[i] == "(":
                if (pair != ")"):
                    balanced = False
                    break
            else:
                balanced = False
                break
    else:
        balanced = False
    return balanced


def printing_task(orders):
    """
    Моделирует процесс печати документов из очереди.
    Каждый заказ печатается с задержкой в 2 секунды.
    Аргументы:
        orders: итерируемый объект с названиями документов для печати.
    """
    deq = deque(orders)
    print("Начало печати")
    while deq.__len__() != 0:
        time.sleep(2)
        print(f"{deq.popleft()} напечатано")
    print("Конец печати")


def palindrome_task(palindrom):
    """
    Проверяет, является ли переданная последовательность палиндромом.
```

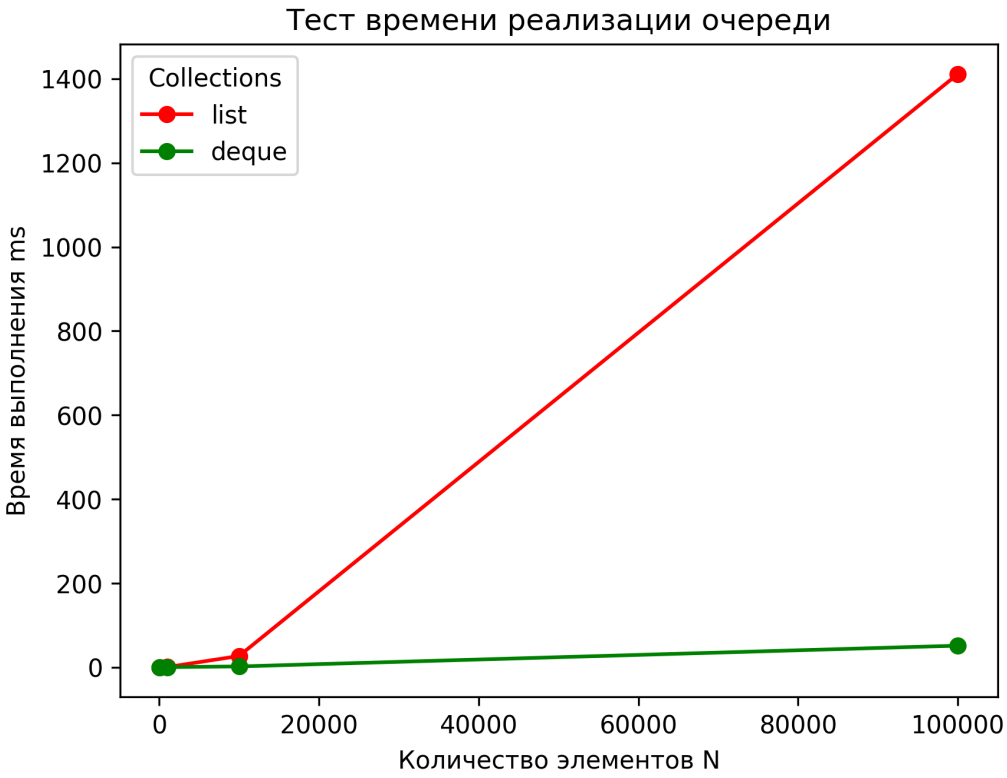
```
    Аргументы:
        palindrom: строка или последовательность для проверки.
    Возвращает:
        True, если последовательность палиндром, иначе False.
    """
    deq = deque(palindrom)
    is_palindrom = True
    for i in range(deq.__len__() // 2):
        if (deq[i] != deq[deq.__len__() - (1+i)]):
            is_palindrom = False
            break
    return is_palindrom
```

```
# main.py
import modules.perfomance_analysis as pa
import modules.task_solutions as ts

if __name__ == "__main__":
    # Performance analysis block
    sizes = [100, 1000, 10000, 100000]
    pa.Visualization(sizes)

    # bracket task
    print(ts.bracket_task("{[()]}"))

    # printing task
    orders = {"Отчёт по продажам", "Дипломная работа", "Рецепт пирога"}
    ts.printing_task(orders)

    # palindrome task
    print(ts.palindrome_task("12332"))
```

```
    Характеристики ПК для тестирования:
    - Процессор: Intel Core i5-12500H @ 2.50GHz
    - Оперативная память: 32 GB DDR4
    - ОС: Windows 11
    - Python: 3.12

[0.01379998866468668, 0.18279999494552612, 9.937399998307228, 1400.2086999826133]
- list
 [0.027600006433203816, 0.3482000029180199, 2.2190000163391232, 33.13450000132434]
-linked_list
[0.06570000550709665, 0.49249999574385583, 26.865099993301556, 1411.3209999923129]
- list
 [0.04439998883754015, 0.37910000537522137, 2.0296000002417713, 51.44469998776913]
- deque
6
```

```
True
Начало печати
Дипломная работа напечатано
Отчёт по продажам напечатано
Рецепт пирога напечатано
Конец печати
False
```





Ответы на контрольные вопросы

1. Отличие динамического массива (list) от связного списка по сложности операций

- **Динамический массив (`list` в Python)** хранит элементы в **непрерывной области памяти**.
    - Вставка в начало требует **сдвига всех элементов**, поэтому имеет сложность **O(n)**.
    - Доступ по индексу выполняется за **O(1)**, так как элемент можно найти по адресу.
- **Связный список** хранит элементы в **узлах**, связанных ссылками.
    - Вставка в начало — просто изменение одной ссылки, сложность **O(1)**.
    - Доступ по индексу требует последовательного обхода, сложность **O(n)**.

---

2. Принцип работы стека и очереди с примерами

- **Стек (LIFO — Last In, First Out)**: последний добавленный элемент извлекается первым.
  Примеры использования:
    1. Реализация механизма *undo/redo* в редакторах.
    2. Обход дерева в глубину (DFS).
- **Очередь (FIFO — First In, First Out)**: первый добавленный элемент извлекается первым.
  Примеры использования:
    1. Планирование задач в операционной системе.
    2. Обработка запросов в принтере или веб-сервере.

---

3. Почему `list.pop(0)` — O(n), а `deque.popleft()` — O(1)

- В `list` элементы хранятся подряд в памяти. При удалении первого элемента все остальные **сдвигаются на одну позицию**, что требует **O(n)** времени.
- В `deque` элементы хранятся в **двухсторонней очереди**, где есть ссылки на начало и конец. Удаление первого элемента лишь изменяет ссылку, без сдвига, поэтому выполняется за **O(1)**.

---

4. Какая структура данных подходит для системы "отмены действий" (undo)

Наилучший выбор — **стек (LIFO)**.
Каждое новое действие помещается на вершину стека. При выполнении "Отмены" извлекается последнее действие, которое было выполнено последним — это идеально соответствует принципу LIFO.
Для функции *повтора (redo)* можно использовать второй стек.

---

5. Почему вставка в начало списка медленнее, чем в связный список

- У **списка (list)** вставка в начало требует **сдвига всех элементов вправо**, что даёт сложность **O(n)**.

- У **связного списка** вставка в начало — это просто добавление нового узла и изменение одной ссылки (**O(1)**).

Поэтому при вставке 1000 элементов в начало список тратит значительно больше времени, чем связный список, что и подтверждает теоретическую асимптотику.