Отчет по лабораторной работе 2

Основные структуры данных. Анализ и применение

Дата: 2025-10-03
Семестр: 3 курс 1 полугодие - 5 семестр
Группа: ПИЖ-б-о-23-2
Дисциплина: Анализ сложности алгоритмов
Студент: Мальцев Виталий Игоревич

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.

Задание:

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

```
# 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 linked_list import LinkedList
from collections import deque
import matplotlib.pyplot as plt


def measure_list_realization(count):
    # 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):
    # 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


sizes = [100, 1000, 10000, 100000]
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.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.show()


#Характеристики вычислительной машины
pc_info = """
Характеристики ПК для тестирования:
- Процессор: Intel Core i3-12100F
- Оперативная память: 16 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


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


# print(bracket_task("{[()]}"))


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")


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


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


print(palindrome_task("12332"))
```

<image src="./time_complexity_plot_list.png">
<image src="./time_complexity_plot_queue.png">

Контрольные вопросы

1. В чем ключевое отличие динамического массива (list в Python) от связанного списка с точки зрения сложности операций вставки в начало и доступа по индексу?

Динамический массив (list в Python):

- **Вставка в начало:** O(n) — требует сдвига всех элементов вправо.
- **Доступ по индексу:** O(1) — элементы хранятся в непрерывной памяти, доступ по адресу вычисляется за константное время.

Связанный список:

- **Вставка в начало:** O(1) — достаточно изменить указатель на голову списка.
- **Доступ по индексу:** O(n) — необходимо пройти по цепочке указателей от начала до нужного элемента.

Ключевое отличие:
Динамический массив оптимизирован для быстрого доступа по индексу, но медленен при вставке/удалении в начале. Связанный список, наоборот, эффективен для вставки/удаления в начале, но медленен при произвольном доступе.

---

2. Объясните принцип работы стека (LIFO) и очереди (FIFO). Приведите по два примера их практического использования.

Стек (LIFO — Last In, First Out)

- **Принцип:** Последний добавленный элемент извлекается первым.
- **Операции:** push (добавить), pop (извлечь).

Примеры использования:

1. **Обратный отсчёт в браузере:** кнопка «Назад» использует стек истории страниц.
2. **Вызов функций в программе:** стек вызовов (call stack) хранит адреса возврата

и локальные переменные.

Очередь (FIFO — First In, First Out)

- **Принцип:** Первый добавленный элемент извлекается первым.
- **Операции:** enqueue (добавить), dequeue (извлечь).

Примеры использования:

1. **Печать документов:** задания отправляются в очередь, печатаются в порядке поступления.
2. **Обработка запросов веб-сервера:** запросы обрабатываются в порядке их получения.

---

3. Почему операция удаления первого элемента из списка (list) в Python имеет сложность O(n), а из дека (deque) — O(1)?

- **Python list (динамический массив):**
  При удалении первого элемента (`del lst[0]` или `lst.pop(0)`) все остальные элементы должны быть сдвинуты на одну позицию влево. Это требует перемещения $n-1$ элементов → **O(n)**.

- **collections.deque (двусторонняя очередь):**
  Реализована как двусвязный список (или кольцевой буфер). Удаление с любого конца не требует сдвига элементов — просто обновляются указатели на начало/конец → **O(1)**.

> **Важно:** deque оптимизирован для операций на обоих концах, в то время как list — для доступа по индексу и операций в конце.

---

4. Какую структуру данных вы бы выбрали для реализации системы отмены действий (undo) в текстовом редакторе? Обоснуйте свой выбор.

Выбор: Стек (LIFO)

Обоснование:

- Система отмены действий работает по принципу «последнее действие — первое отменяется». Это идеально соответствует LIFO.
- При каждом изменении текста (ввод символа, удаление, форматирование) текущее состояние (или команда) помещается в стек.
- При нажатии Ctrl+Z — последнее действие извлекается из стека и откатывается.
- Реализация простая и эффективная: O(1) на операцию push/pop.

Альтернатива — история в виде списка: возможна, но менее естественна и требует управления текущей позицией. Стек — каноническое решение.

5. Замеры показали, что вставка 1000 элементов в начало списка заняла значительно больше времени, чем вставка в начало вашей реализации связанного списка. Объясните результаты с точки зрения асимптотической сложности.

Результат объясняется разницей в асимптотической сложности операций:

- **Вставка в начало Python list:** O(n) на каждую вставку.
  Для 1000 вставок:
    - 1-я вставка: сдвиг 0 элементов → O(1)
    - 2-я вставка: сдвиг 1 элемента → O(1)
    - ...
    - 1000-я вставка: сдвиг 999 элементов → O(n)
      Итого: сумма сдвигов = 0 + 1 + 2 + ... + 999 ≈ $n^2/2$ → **$O(n^2)$**.
- **Вставка в начало связанного списка:** O(1) на каждую вставку.
  Для 1000 вставок:
    - Каждая вставка требует только изменения указателя головы → O(1)
      Итого: 1000 × O(1) = **O(n)**.

Вывод:
Асимптотически вставка в начало динамического массива — $O(n^2)$, а в связанном списке — O(n). Поэтому при увеличении количества операций (n=1000) разница во времени становится критической. Это подтверждает теоретические оценки сложности.