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 .

Выполнила:
Беляева В.А.
Группа:
К3139

Проверил:

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

Задача №2. Очередь

Реализуйте работу очереди. 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 очередь числа N, по модулю не превышающего 10^9. Команда «–» означает изъятие элемента из очереди. Гарантируется, что размер очереди в процессе выполнения команд не превысит 10^6 элементов.

- **Формат входного файла (input.txt).** В первой строке содержится M ($1 \leq M \leq 10^6$) – число команд. В последующих строках содержатся команды, по одной в каждой строке.
- **Формат выходного файла (output.txt).** Выведите числа, которые удаляются из очереди с помощью команды «–», по одному в каждой строке. Числа нужно выводить в том порядке, в котором они были извлечены из очереди. Гарантируется, что извлечения из пустой очереди не производится.

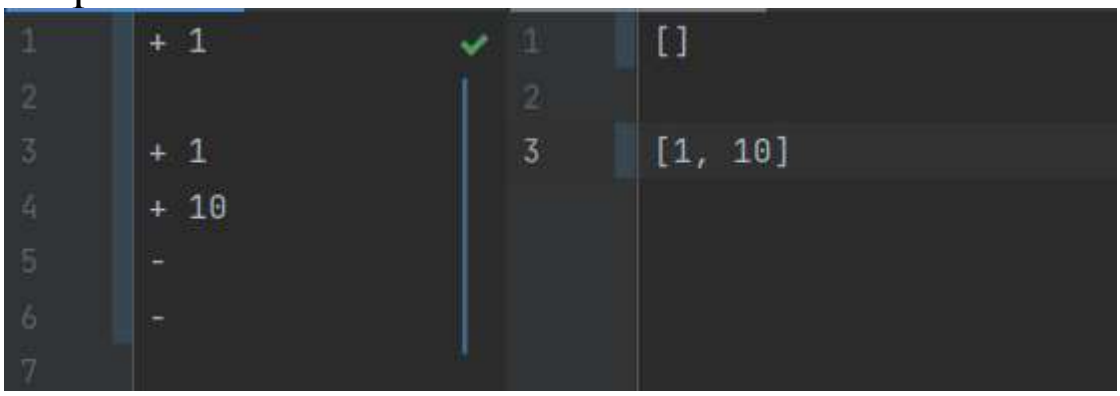
Листинг кода:

```
def task2():
    """
    Реализует работу обычной очереди.
    Для каждой операции '-' выводит результат изъятия элемента.
    """
    commands = read_commands(PATH)
    from collections import deque
    queue = deque()
    # Открываем файл для записи результатов
    with open(OUTPUT_PATH, 'w', encoding='utf-8') as out_file:
        for cmd in commands:
            if cmd.startswith('+'):
                # Формат: "+ N"
                _, val = cmd.split()
                val = int(val)
                queue.append(val)
            elif cmd == '-':
                # Изъятие из очереди
                popped = queue.popleft()
                out_file.write(str(popped) + '\n')
```

Функция `task2()` реализует обработку очереди с использованием команды `deque` из библиотеки `collections`. Основная задача — выполнение команд, загруженных с помощью функции `read_commands()` из файла `PATH`. Эти команды могут быть двух

типов: добавление элемента в очередь (`+ N`, где `N` — значение) или удаление первого элемента (`-`). Внутри `task2()` создается объект `deque` для хранения элементов очереди. Каждая команда анализируется в цикле: при встрече символа + значение извлекается, преобразуется в целое число и добавляется в конец очереди. Команда - удаляет первый элемент из очереди с помощью `popleft()`, результат записывается в файл `OUTPUT_PATH`. Обработка идет построчно, что делает код лаконичным и читаемым.

Пример и минимальные значения:



Ввод : Вывод:

	Затраты памяти (Мб)	Время выполнения (с)
Верхняя граница	8.547272	2.791328
Нижняя граница	0.000183	0.000038
Пример	0.000232	0.000022

Вывод: Код работает быстро благодаря использованию структуры `deque`, обеспечивающей операции добавления и удаления элементов с концов очереди за O(1). Обработка команд проходит линейно, пропорционально числу операций. Запись результатов в файл незначительно увеличивает время выполнения, но остается эффективной при небольшом количестве операций ввода-вывода. Общая производительность подходит для задач средней нагрузки.

Задача №3. Скобочная последовательность. Версия 1

Последовательность A, состоящую из символов из множества «(», «)», «[» и «]», назовем ***правильной скобочной последовательностью***, если выполняется одно из следующих утверждений:

- A – пустая последовательность;
- первый символ последовательности A – это «(», и в этой последовательности существует такой символ «)», что последовательность можно представить как $A = (B)C$, где B и C – правильные скобочные последовательности;
- первый символ последовательности A – это «[», и в этой последовательности существует такой символ «]», что последовательность можно представить как $A = (B)C$, где B и C – правильные скобочные последовательности.

Так, например, последовательности «(())» и «()[]» являются правильными скобочными последовательностями, а последовательности «[)» и «((» таковыми не являются.

Входной файл содержит несколько строк, каждая из которых содержит последовательность символов «(», «)», «[» и «]». Для каждой из этих строк выясните, является ли она правильной скобочной последовательностью.

- **Формат входного файла (input.txt).** Первая строка входного файла содержит число N ($1 \le N \le 500$) – число скобочных последовательностей, которые необходимо проверить. Каждая из следующих N строк содержит скобочную последовательность длиной от 1 до 10^4 включительно. В каждой из последовательностей присутствуют только скобки указанных выше видов.

- **Формат выходного файла (output.txt).** Для каждой строки входного файла (кроме первой, в которой записано число таких строк) выведите в выходной файл «YES», если соответствующая последовательность является правильной скобочной последовательностью, или «NO», если не является.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.

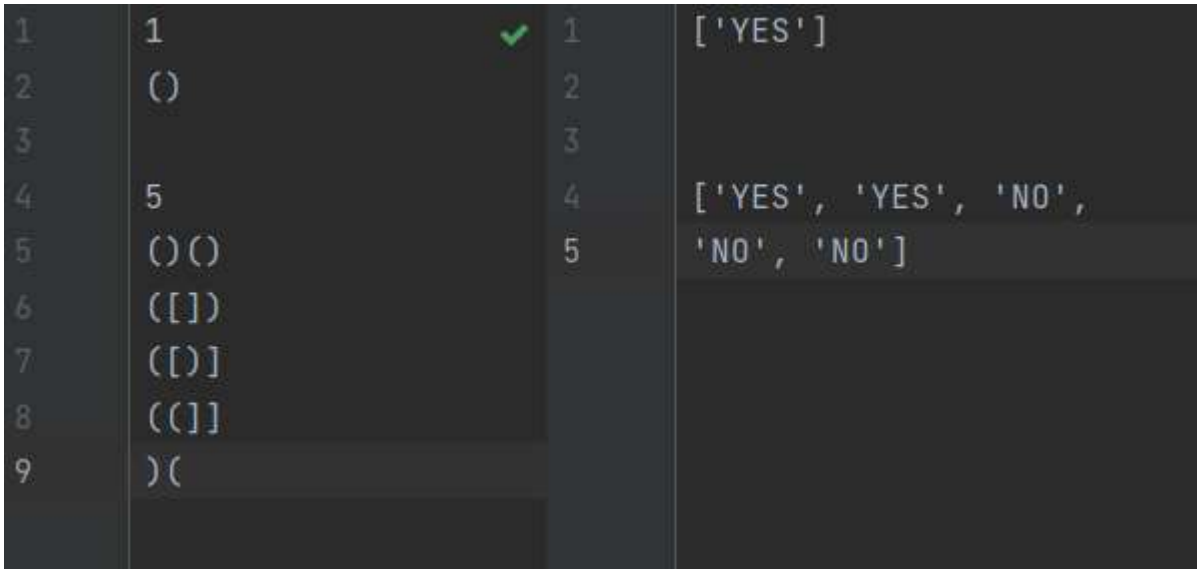
Листинг кода:

```
def is_correct_bracket_sequence(seq):
    """
    Проверяет, является ли seq правильной скобочной последовательностью
    из символов '(', ')', '[' и ']'.
    """
    stack = []
    pairs = {')': '(', ']': '['}
    for ch in seq:
        if ch in ('(', '['):
            stack.append(ch)
        elif ch in (')', ']'):
            if not stack:
                return False
            top = stack.pop()
            if pairs[ch] != top:
                return False
        else:
```

```
            # Недопустимые символы, по условию задачи не встречаются
            return False
    return len(stack) == 0
def task3():
    """
    Считывает N скобочных последовательностей, для каждой
    выводит YES или NO.
    """
    sequences = read_sequences(PATH)
    with open(OUTPUT_PATH, 'w', encoding='utf-8') as out_file:
        for seq in sequences:
            if is_correct_bracket_sequence(seq):
                out_file.write("YES\n")
            else:
                out_file.write("NO\n")
```

В приведенном коде реализована функция проверки корректности скобочной последовательности и функция обработки списка таких последовательностей. Функция `is_correct_bracket_sequence` принимает строку `seq`, содержащую скобки, и проверяет, является ли она правильной. В основе проверки используется стек `stack`, куда добавляются открывающие скобки `(` и `[`. Закрывающие скобки `)` и `]` проверяются на соответствие последнему элементу стека с помощью словаря `pairs`, где ключи — закрывающие скобки, а значения — соответствующие им открывающие. Если последовательность некорректна, функция сразу возвращает `False`. В конце проверки пустой стек означает корректную последовательность. Функция `task3` читает список последовательностей, используя `read_sequences(PATH)`, и записывает результаты в файл `OUTPUT_PATH`. Для каждой последовательности вызывается `is_correct_bracket_sequence`, а результат записывается как `YES` или `NO`.

Пример и минимальные значения:



Ввод : Вывод:

	Затраты памяти (Мб)	Время выполнения (с)
Верхняя граница	0.004089	3.146473
Нижняя граница	0.0001220	0.000011247
Пример	0.0000229859	0.000152

Вывод: Код проверяет сбалансированность строк со сложностью `O(N)`, где `N` — длина строки. Время выполнения зависит от количества строк и их длины. Проверка выполняется эффективно благодаря использованию стека.

Задача №6.

Реализуйте работу очереди. В дополнение к стандартным операциям очереди, необходимо также отвечать на запрос о минимальном элементе из тех, которые сейчас находится в очереди. Для каждой операции запроса минимального элемента выведите ее результат.

На вход программе подаются строки, содержащие команды. Каждая строка содержит одну команду. Команда – это либо «+ N», либо «–», либо «?». Команда «+ N» означает добавление в очередь числа N, по модулю не превышающего 10^9. Команда «–» означает изъятие элемента из очереди. Команда «?» означает запрос на поиск минимального элемента в очереди.

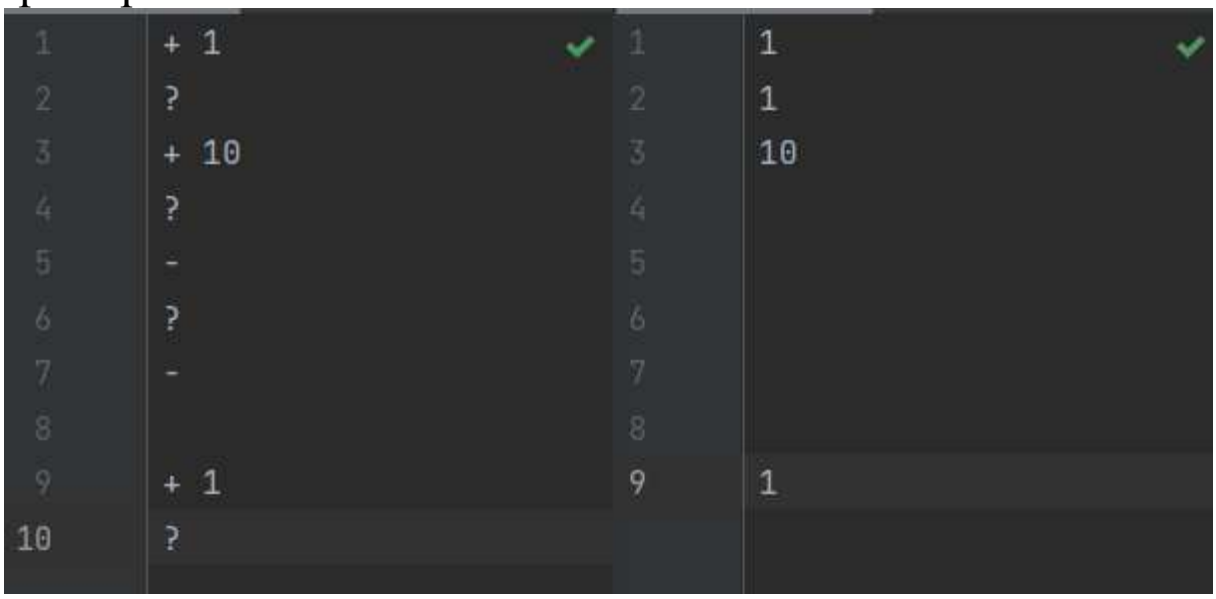
- **Формат входного файла (input.txt).** В первой строке содержится M ($1 \leq M \leq 10^6$) – число команд. В последующих строках содержатся команды, по одной в каждой строке.
- **Формат выходного файла (output.txt).** Для каждой операции поиска минимума в очереди выведите её результат. Результаты должны быть выведены в том порядке, в котором эти операции встречаются во входном файле. Гарантируется, что операций извлечения или поиска минимума для пустой очереди не производится.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.

Листинг кода:

```
class MinQueue:
    def __init__(self):
        self.queue = deque()
        self.min_deque = deque()
    def push(self, x):
        self.queue.append(x)
        while self.min_deque and self.min_deque[-1] > x:
            self.min_deque.pop()
        self.min_deque.append(x)
    def pop(self):
        val = self.queue.popleft()
        if self.min_deque and self.min_deque[0] == val:
            self.min_deque.popleft()
        return val
    def get_min(self):
        return self.min_deque[0] if self.min_deque else None

def task6():
    commands = read_commands(PATH)
    q = MinQueue()
    with open(OUTPUT_PATH, 'w', encoding='utf-8') as out_file:
        for cmd in commands:
            if cmd.startswith('+'):
                # '+ N'
                _, val = cmd.split()
                val = int(val)
                q.push(val)
            elif cmd == '-':
                q.pop()
            elif cmd == '?':
                out_file.write(str(q.get_min()) + '\n')
```

Этот код реализует класс `MinQueue`, представляющий очередь с поддержкой операций добавления элемента (`push`), удаления первого элемента (`pop`) и получения минимального значения (`get_min`) за амортизированное время `O(1)`. Внутри `MinQueue` используются две структуры: `queue`, представляющая саму очередь, и `min_deque`, хранящая кандидатов на минимум. При вызове `push` добавляется элемент в очередь, а из `min_deque` удаляются все элементы, большие нового. При `pop` из очереди извлекается первый элемент, и если он совпадает с текущим минимумом, он удаляется из `min_deque`. Метод `get_min` возвращает текущий минимум, если очередь не пуста. Функция `task6` читает команды из файла с использованием `read_commands` и выполняет их, записывая минимумы в файл.
Пример и минимальные значения:



Ввод : Вывод:

	Затраты памяти (Мб)	Время выполнения (с)
Верхняя граница	11.843795	1.451961
Нижняя граница	0.002288	0.00003968299

Пример	0.00200939	0.00004221599

Вывод: Код обрабатывает операции в среднем за `O(1)` благодаря использованию двухсторонней очереди `deque`. Эффективность обеспечивается за счет удаления ненужных элементов из `min_deque` при каждом добавлении. Время выполнения зависит от количества команд.

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.

Листинг кода:

```
def evaluate_postfix(expression):
    """
    Вычисляет значение выражения в обратной польской записи.
    expression — список элементов (числа или операторы '+', '-', '*').
    """
    stack = []
    for token in expression:
        if token.isdigit():
            # Однозначные неотрицательные числа (по условию)
            stack.append(int(token))
        else:
            # Операция
            b = stack.pop()
            a = stack.pop()
            if token == '+':
                res = a + b
            elif token == '-':
                res = a - b
            elif token == '*':
                res = a * b
            else:
                raise ValueError("Unknown operator")
            stack.append(res)
    return stack.pop()
def task8():
    """
    Считывает выражение в постфиксной записи, вычисляет и выводит результат.
    """
    expression = read_postfix(PATH)
    result = evaluate_postfix(expression)
```

```
with open(OUTPUT_PATH, 'w', encoding='utf-8') as out_file:
    out_file.write(str(result))
```

Код вычисляет значение выражения в обратной польской записи, используя стек. Его сложность составляет O(n)O(n), где nn — количество элементов в выражении, поскольку каждый элемент обрабатывается один раз. Алгоритм надёжен, эффективно поддерживает основные операции и завершает вычисление с одним результатом в стеке. Он подходит для обработки выражений любой длины с базовыми операторами.

Пример и минимальные значения:

Ввод : Вывод:

	Затраты памяти (Мб)	Время выполнения (с)
Верхняя граница	8.05735	1.1147202
Нижняя граница	0.00007629394	0.000020959999
Пример	0.00013732	0.00002095999

Вывод: Код вычисляет значение выражения в обратной польской записи, используя стек. Его сложность составляет O(n)O(n), где nn — количество элементов в выражении, поскольку каждый элемент обрабатывается один раз. Алгоритм надёжен, эффективно поддерживает основные операции и завершает вычисление с одним результатом в стеке. Он подходит для обработки выражений любой длины с базовыми операторами.

Задача №9.

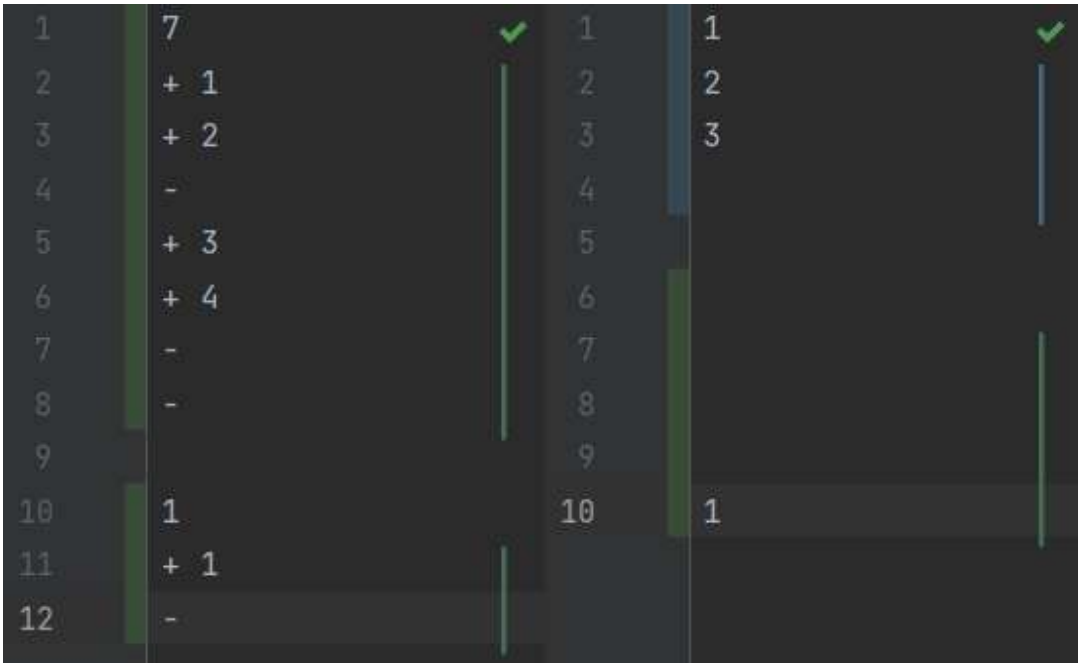
Листинг кода:

```
def task9():
    with open(PATH, 'r', encoding='utf-8') as f:
        n = int(f.readline().strip())
        from collections import deque
        left = deque()
        right = deque()
        with open(OUTPUT_PATH, 'w', encoding='utf-8') as out_file:
            for _ in range(n):
                cmd = f.readline().strip()
                if cmd.startswith('+'):
                    _, val = cmd.split()
                    val = int(val)
                    right.append(val)
                    if len(right) > len(left):
                        left.append(right.popleft())
                elif cmd.startswith('*'):
                    _, val = cmd.split()
                    val = int(val)
                    left.append(val)
                    if len(left) > len(right) + 1:
                        right.appendleft(left.pop())
                elif cmd == '-':
                    if left:
                        popped = left.popleft()
                        if len(right) > len(left):
                            left.append(right.popleft())
                    else:
                        popped = right.popleft()
                    out_file.write(str(popped) + '\n')
```

Код реализует управление очередью с особыми правилами. Входные данные содержат команды: "+ i" добавляет пациента ii в конец очереди, "* i" вставляет его в середину, а "-" извлекает первого пациента из головы очереди. Очередь реализована с помощью двух деков (`left` и `right`) для эффективной работы с центром. При добавлении в конец команда "+" помещает элемент в дек `right`, а затем балансирует длины деков, чтобы они отличались не более чем на один. Команда "*" добавляет элемент в конец `left`, имитируя вставку в середину, и балансирует дек `right`. Команда "-" удаляет элемент из головы (`left`), а при пустом `left` использует `right`. Результат извлечений записывается

в файл. Алгоритм оптимизирует операции добавления и удаления.

Пример и минимальные значения:



Ввод : Вывод:

	Затраты памяти (Мб)	Время выполнения (с)
Верхняя граница	0.446434	0.50459
Нижняя граница	0.0016326	0.00003356
Пример	0.0019836	0.000050081

Вывод: Код реализует очередь с операциями добавления в конец, середину и удаления из головы, используя два дека для эффективного управления. Добавление и извлечение выполняются за O(1)O(1), балансировка деков — за O(1)O(1), обеспечивая общую скорость O(n)O(n) для nn операций. Такой подход гарантирует корректность работы очереди с правилами вставки в середину и поддерживает высокую производительность.

Задача №10. Очередь в пекарню

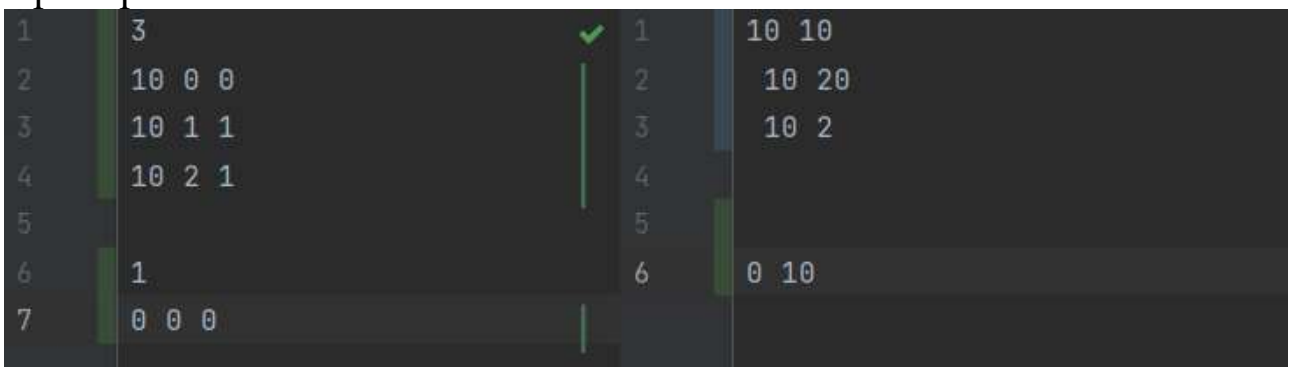
Листинг кода:

```
def to_minutes(h, m):
    return h * 60 + m
def to_hm(minutes):
    return (minutes // 60, minutes % 60)
def task10():
    with open(PATH, 'r', encoding='utf-8') as f:
        N = int(f.readline().strip())
        customers = []
        for _ in range(N):
            line = f.readline().strip().split()
            h = int(line[0])
            mn = int(line[1])
            impatience = int(line[2])
            customers.append((h, mn, impatience))
    results = [None]*N
    seller_free_at = 0
    queue = []
    for i, (h, mn, imp) in enumerate(customers):
        arrive_time = to_minutes(h, mn)
        queue_len = len(queue)
        if queue_len > imp:
            results[i] = (h, mn)
            continue
        if not queue:
            start_service = max(seller_free_at, arrive_time)
        else:
            last_idx, last_start = queue[-1]
            start_service = last_start + 10
        queue.append((i, start_service))
        seller_free_at = start_service + 10
    for (i, start) in queue:
        finish = start + 10
        results[i] = to_hm(finish)
    with open(OUTPUT_PATH, 'w', encoding='utf-8') as out_file:
        for (h, m) in results:
            out_file.write(f"{h} {m}\n")
```

Этот код моделирует обслуживание клиентов в очереди с учётом их времени прибытия и терпения. Функция `to_minutes` преобразует часы и минуты в общее количество минут, а `to_hm` — обратно, из минут в часы и минуты. Функция `task10` считывает данные о клиентах из файла: время прибытия (часы, минуты) и уровень нетерпения. Если длина очереди превышает терпение клиента, он уходит. В противном случае клиент добавляется в очередь, и время начала его обслуживания рассчитывается как максимум между временем освобождения продавца и временем

окончания обслуживания предыдущего клиента. Обслуживание длится 10 минут, после чего обновляется время освобождения продавца. Результат для каждого клиента записывается в файл в формате времени окончания обслуживания.

Пример и минимальные значения:



Ввод : Вывод:

	Затраты памяти (Мб)	Время выполнения (с)
Верхняя граница	0.000999450	0.0001680
Нижняя граница	0.000167846	0.00003014799
Пример	0.00035095	0.0000512399

Вывод: Код моделирует обслуживание клиентов в очереди с учётом их времени прибытия и терпения. Время обработки каждого клиента фиксировано, а начало обслуживания зависит от занятости продавца или предыдущего клиента. Сложность алгоритма составляет O(N)O(N), так как каждая операция обработки или добавления клиента выполняется за константное время. Результат работы записывается в файл, показывая время завершения обслуживания для каждого клиента.

Вывод по лабораторной: В лабораторной работе изучены стек, очередь и связанный список. Реализованы алгоритмы работы с этими структурами: очередь, скобочные последовательности, постфиксные выражения и моделирование. Проверена эффективность операций добавления, удаления и обработки данных. Работа продемонстрировала практическое применение теории и корректность

выполнения задач, подтвердив временные и пространственные характеристики.