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 6

Выполнил:
Данилова А.В.
К3141 (номер группы)

Проверила:
Артамонова В.Е.

Санкт-Петербург
2024 г.

Содержание отчета

Задачи по варианту

2 задача. 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.

Листинг кода:

```
from pathlib import Path
from lab_4.utils.utils import work
from collections import deque


def realize_queue(n: int, operations) -> list:
    my_queue = deque()
    deleted = []
    for f in operations:
        if len(f) == 2:
            my_queue.append(f[1])
        else:
            deleted.append(my_queue.popleft())
    return deleted


if __name__ == "__main__":
    work(Path(__file__), realize_queue)
```

Импортируется класс deque из модуля collections. Создается объект класса очередь. Цикл for проходится по операциям и добавляет удаленные элементы в массив, финальное значение которого будет возвращено искомой функцией.

Результат работы кода на примерах из текста задачи:

```
LAB NUMBER: 4
            TASK NUMBER: 2
                    INPUT DATA: (4, ['+', 1], ['+', 10], ['-'], ['-'])
                    OUTPUT DATA: [1, 10]
```

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения
Нижняя граница диапазона значений входных данных из текста задачи	3.5999983083456755e-06
Верхняя граница диапазона значений входных данных из текста задачи	0.10797050001565367

Вывод по задаче:
Очередь эффективно и лёгко управляет большим объёмом данных, а также обладает высокой скоростью передачи этих данных.

3 задача. 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.

Листинг кода:

```
from collections import deque
from pathlib import Path
from lab_4.utils.utils import work


def isValid(s: str) -> bool:
    stack = deque()
    brackets = {
        "(": ")",
        "[": "]"
    }
    for br in s:
        if br in brackets:
            stack.append(br)
        else:
            if stack and brackets[stack.pop()] == br:
                continue
            else:
                return False
    return not stack


def bracket_sequence(n, data):
    ans = []
    for seq in data:
        if isValid(seq[0]):
            ans.append('YES')
        else:
            ans.append('NO')
    return ans


if __name__ == "__main__":
    work(Path(__file__), bracket_sequence)
```

Создаётся объект deque и базовый словарь с скобками. Цикл for проходится по входной строке s, если скобка открывающая, то она добавляется в стек, если же она закрывающая, то её пара – одна из открывающих удаляется из стека. Если удалять нечего или удаляется скобка другого типа, то исходная последовательность неправильная.

Результат работы кода на примерах из текста задачи:

```
LAB NUMBER: 4
            TASK NUMBER: 3
                    INPUT DATA: (5, ['()()'], ['([])'], ['([])'], ['(()]'], [')('])
                    OUTPUT DATA: ['YES', 'YES', 'NO', 'NO', 'NO']
```

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения
Нижняя граница диапазона значений входных данных из текста задачи	8.500006515532732e-06
Верхняя граница диапазона значений входных данных из текста задачи	0.00021389999892562628

Вывод по задаче:
С помощью структуры данных стек можно и, не сильно много думая, решить большое количество задач за линейное время.

5 задача. 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().

Листинг кода:

```
from lab_4.utils.utils import work
from pathlib import Path
from collections import deque


class Stack:
    def __init__(self):
        self.stack_ = deque()

    def push(self, item):
        if self.stack_:
            new_max = max(item, self.stack_[-1][1])
        else:
            new_max = item
        self.stack_.append((item, new_max))

    def pop(self):
        if self.stack_:
            return self.stack_.pop()[0]

    def get_max(self):
        if self.stack_:
            return self.stack_[-1][1]
```

```
def main(n, operations):
    my_queue = Stack()
    max_values = []
    for f in operations:
        match f[0]:
            case 'push':
                my_queue.push(f[1])
            case 'pop':
                my_queue.pop()
            case 'max':
                max_values.append(my_queue.get_max())
    return max_values


if __name__ == "__main__":
    work(Path(__file__), main)
```

Класс Stack:
__init__: Инициализирует пустой стек с помощью deque().
push(self, item): Добавляет элемент item в стек. В отличие от обычного стека, он также отслеживает максимальное значение. Он хранит в стеке кортежи (item, new_max), где new_max - максимальное значение среди всех элементов в стеке после добавления item.
pop(self): Удаляет и возвращает верхний элемент стека.
get_max(self): Возвращает максимальное значение среди элементов в стеке.

Результат работы кода на примерах из текста задачи:

```
LAB NUMBER: 4
           TASK NUMBER: 5
                  INPUT DATA: (6, ['push', 2], ['push', 1], ['max'], ['pop'], ['pop'], ['max'])
                  OUTPUT DATA: [2, None]
```

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения
Нижняя граница диапазона значений входных данных из текста задачи	1.0899995686486363e-05
Верхняя граница диапазона значений входных данных из текста задачи	0.005258599994704127

Вывод по задаче:
Используя, стек можно найти за константное время максимум.

12 задача. Строй новобранцев

В этой задаче n новобранцев, пронумерованных от 1 до n, разделены на два множества: строй и толпа. Вначале строй состоит из новобранца номер 1, все остальные составляют толпу. В любой момент времени строй стоит в один ряд по прямой. Товарищ сержант может использовать четыре команды. Вот они.

• "I, встать в строй слева от J."Эта команда заставляет новобранца номер I, находящегося в толпе, встать слева от новобранца номер J, находящегося встрою.

• "I, встать в строй справа от J."Эта команда действует аналогично предыдущей, за исключением того, что I встает справа от J.

• "I, выйти из строя."Эта команда заставляет выйти из строя новобранца номер I. После этого он присоединяется ктолпе.

• "I, назвать соседей."Эта команда заставляет глубоко задуматься новобранца номер I, стоящего в строю, и назвать номера своих соседей по строю, сначала левого, потом правого. Если кто-то из них отсутствует (новобранец находится на краю ряда), то вместо соответствующего номера он должен назвать 0.

Известно, что ни в каком случае строй не остается пустым. Иногда строй становится слишком большим, и товарищ сержант уже не может проверять сам, правильно ли отвечает новобранец. Поэтому он попросил вас написать программу, которая помогает ему в нелегком деле обучения молодежи и выдает правильные ответы для его командю

Листинг кода:

```
from lab_4.utils.utils import work
from pathlib import Path


class InLine:
    def __init__(self):
        self.left = 0
        self.right = 0


class Formation:
    def __init__(self, n):
        self.a = [InLine() for _ in range(n + 1)]

    def go_left(self, i, j):
        self.a[i].left = self.a[j].left
        if self.a[i].left != 0:
            self.a[self.a[i].left].right = i
```

```
        self.a[j].left = i
        self.a[i].right = j

    def go_right(self, i, j):
        self.a[i].right = self.a[j].right
        if self.a[i].right != 0:
            self.a[self.a[i].right].left = i
        self.a[j].right = i
        self.a[i].left = j

    def go_out(self, i):
        if self.a[i].left != 0:
            self.a[self.a[i].left].right = self.a[i].right
        if self.a[i].right != 0:
            self.a[self.a[i].right].left = self.a[i].left
        self.a[i].left = 0
        self.a[i].right = 0

    def tell_neighbours(self, i):
        return f"{self.a[i].left} {self.a[i].right}"



def main(n, m, operations):
    formation = Formation(n)
    ans = []
    for action in operations:
        if action[0] == "left":
            i, j = int(action[1]), int(action[2])
            formation.go_left(i, j)
        elif action[0] == "right":
            i, j = int(action[1]), int(action[2])
            formation.go_right(i, j)
        elif action[0] == "leave":
            i = int(action[1])
            formation.go_out(i)
        elif action[0] == "name":
            i = int(action[1])
            ans += [formation.tell_neighbours(i)]

    return ans


if __name__ == "__main__":
    work(Path(__file__), main)
    main(5, 1, ['right', 2, 1])
```

1. Класс InLine: Это внутренний класс, представляющий одного солдата. Он хранит ссылки на левого и правого соседей. Значение 0 указывает на отсутствие соседа.
2. Класс Formation:

__init__(self, n): Создает строй из n + 1 элементов (т.к индексы с 0). Каждый элемент инициализируется как InLine с пустыми ссылками на

go_left(self, i, j): Перемещает элемент i влево от элемента j. Он обновляет ссылки left и right для элементов i и j, а также для соседей i (если они есть)

go_right(self, i, j): Аналогично go_left, но перемещает элемент i вправо от элемента j.

go_out(self, i): Удаляет элемент i из строя, обновляя ссылки его соседей.
tell_neighbours(self, i): Возвращает номера левого и правого соседей i-го солдата.

Результат работы кода на примерах из текста задачи:

```
LAB NUMBER: 4
            TASK NUMBER: 12
                    INPUT DATA: (3, 4, ['left', 2, 1], ['right', 3, 1], ['leave', 1], ['name', 2])
                    OUTPUT DATA: ['0 3']
```

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения
Нижняя граница диапазона значений входных данных из текста задачи	8.500006515532732e-06
Верхняя граница диапазона значений входных данных из текста задачи	0.0938717000244651

Вывод по задаче:
Решение этой задачи моделирует двусвязный список, представляющий собой линейную формацию - строй солдат, что показывает универсальность данной структуры данных.

Дополнительные задачи

9 задача. Поликлиника

Очередь в поликлинике работает по сложным правилам. Обычные пациенты при посещении должны вставать в конец очереди. Пациенты, которым "только справку забрать встают ровно в ее середину, причем при нечетной длине очереди они встают сразу за центром. Напишите программу, которая отслеживает порядок пациентов в очереди.

Листинг кода:

```
from lab_4.utils.utils import work
from pathlib import Path


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


class Clinic:
    def __init__(self):
        self.head = None
        self.last = None

    def new_patient(self, data):
        if self.head is None:
            self.head = Node(data)
            self.last = self.head
        else:
            self.last.next = Node(data)
            self.last.next.prev = self.last
            self.last = self.last.next

    def find_middle(self):
        _l = self.head
        r = self.last
        while _l != r and _l.next != r:
            _l = _l.next
            r = r.prev
        return _l

    def get_certificate(self, data):
        if self.head is None:
            self.head = Node(data)
            self.last = self.head
            return
        elif self.head == self.last:
            self.new_patient(data)
            return
        mid = self.find_middle()
        next_to_mid = mid.next
        mid.next = Node(data)
        mid.next.prev = mid
        mid.next.next = next_to_mid
        next_to_mid.prev = mid.next
```

```
    def go_doctor(self):
        if self.head is None:
            return None
        elif self.head == self.last:
            patient = self.head
            self.head = self.last = None
            return patient.data

        patient = self.head
        self.head = self.head.next
        self.head.prev = None
        return patient.data

    def __repr__(self):
        queue = ''
        temp = self.head
        while temp is not None:
            queue += f' {temp.data}'
            temp = temp.next
        return queue


def main(n, operations):
    ans = []
    q = Clinic()
    for action in operations:
        match action[0]:
            case '+':
                q.new_patient(action[1])
            case '':
                q.get_certificate(action[1])
            case '-':
                ans += [q.go_doctor()]
    return ans


if __name__ == "__main__":
    work(Path(__file__), main)
```

1. Класс Node: Представляет собой узел двусвязного списка. Хранит данные пациента (data) и ссылки на предыдущий (prev) и следующий (next) узлы.

2. Класс Clinic: Реализует двусвязный список, представляющий очередь пациентов.

__init__: Инициализирует пустую очередь (head = None, last = None).

new_patient(self, data): Добавляет нового пациента (data) в конец очереди.

find_middle: Находит средний узел в списке. Использует два указателя, _l и r, которые движутся с разных концов списка к середине.

get_certificate(self, data): Добавляет пациента (data) перед средним элементом очереди. Это симулирует выдачу сертификата — пациент получает внеочередной прием.

go_doctor(): Удаляет и возвращает первого пациента из очереди (симулирует прием у врача).

__repr__: Возвращает строковое представление очереди.

Результат работы кода на примерах из текста задачи:

```
LAB NUMBER: 4
           TASK NUMBER: 9
                  INPUT DATA: (7, ['+', 1], ['+', 2], ['-'], ['+', 3], ['+', 4], ['-'], ['-'])
                  OUTPUT DATA: [1, 2, 3]
```

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения
Нижняя граница диапазона значений входных данных из текста задачи	6.570000550709665e-05
Верхняя граница диапазона значений входных данных из текста задачи	13.273511599982157

Вывод по задаче:
На примере из этой задачи стало понятно, что двусвязный список является эффективным решением задач на добавление и удаление элементов.

13 задача⋆. Реализация стека, очереди и связанных списков

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

Листинг кода:

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


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

    def isEmpty(self):
        if self.start:
            return False
        return True
```

```
    def push(self, element):
        newP = Node(element)
        if self.start is None:
            self.start = self.top = newP
            return
        newP.prev = self.top
        self.top.next = newP
        self.top = newP

    def pop(self):
        if self.isEmpty():
            return None
        popped_value = self.top
        if self.start == self.top:
            self.top = self.start = None
            return popped_value.val

        self.top = self.top.prev
        if self.top is not None:
            self.top.next = None
        return popped_value.val

    def printstack(self):
        if self.isEmpty():
            print('List is Empty')
            return
        curr = self.start
        print("Stack is :  ", end='')
        while curr is not None:
            print(curr.val, end=' ')
            curr = curr.next
        print()


if __name__ == "__main__":
    pass
    work(Path(__file__), main)
```

__init__: инициализирует пустой стек. self.start указывает на первый узел (нижний элемент стека), а self.top – на последний узел (верхний элемент стека).

isEmpty(): проверяет, пустой ли стек. Возвращает True, если стек пустой, и False в противном случае.

push(self, element): добавляет элемент element на вершину стека. Создается новый узел newP, и он вставляется в начало списка. Если стек был пустым, self.start и self.top указывают на newP.

pop(self): удаляет и возвращает элемент с вершины стека. Если стек пустой, возвращается None. Если в стеке только один элемент, self.start и self.top устанавливаются в None. В противном случае, self.top перемещается к предыдущему узлу.

printstack(): выводит содержимое стека на консоль.

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения
Нижняя граница диапазона значений входных данных из текста задачи	3.65770010230550665e-05
Верхняя граница диапазона значений входных данных из текста задачи	1.73497243199982157

Вывод по задаче:

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

Листинг кода:

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


class Queue:
    def __init__(self):
        self.head = None
        self.last = None

    def isEmpty(self):
        if self.head:
            return False
        return True

    def enqueue(self, data):
        if self.last is None:
            self.head = Node(data)
            self.last = self.head
        else:
            self.last.next = Node(data)
            self.last.next.prev = self.last
            self.last = self.last.next

    def dequeue(self):
        if self.head is None:
            return None
        else:
            temp = self.head.data
            if self.last != self.head:
                self.head = self.head.next
            else:
                self.head = self.last = None
            return temp
```

```
    def size(self):
        temp = self.head
        count = 0
        while temp is not None:
            count = count + 1
            temp = temp.next
        return count

    def __repr__(self):
        queue = ''
        temp = self.head
        while temp is not None:
            queue += f'{temp.data} '
            temp = temp.next
        return queue


if __name__ == "__main__":
    pass
```

__init__: инициализирует пустую очередь. self.head указывает на первый узел (голова очереди), а self.last – на последний узел (хвост очереди).
isEmpty(): проверяет, пуста ли очередь. Возвращает True, если очередь пуста, и False в противном случае.
enqueue(self, data): добавляет элемент data в конец очереди. Создается новый узел, и он добавляется после self.last. Если очередь была пустой, то self.head и self.last указывают на новый узел.
dequeue(self): удаляет и возвращает элемент из начала очереди. Если очередь пуста, возвращается None. Если в очереди только один элемент, то self.head и self.last устанавливаются в None.
size(): возвращает количество элементов в очереди. Перебирает все узлы и считает их.
__repr__: возвращает строковое представление очереди (все элементы через пробел).

Результат работы кода на максимальных и минимальных значениях:

	Время выполнения
Нижняя граница диапазона значений входных данных из текста задачи	4.570000550709665e-05
Верхняя граница диапазона значений входных данных из текста задачи	1.883511599982157

Вывод по задаче:
Использование двусвязного списка позволяет эффективно добавлять и удалять элементы с обоих концов (хотя для стека обычно используется только один конец).

Вывод

Задачи показали, что правильный выбор структуры данных играет ключевую роль в эффективности алгоритмов. Понимание особенностей стеков, очередей и связных списков позволяет разрабатывать эффективные и масштабируемые решения для различных задач обработки данных.