**МИНОБРНАУКИ РОССИИ**
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**
**Кафедра МО ЭВМ**

# ОТЧЕТ

**по лабораторной работе №2**

**по дисциплине «Информатика»**

**Тема: Алгоритмы и структуры данных в Python**

Студент гр. 3341 Лодыгин И.А.

Преподаватель Иванов Д.В.

Санкт-Петербург

2024

**Цель работы**

Целью работы является освоение работы с алгоритмами и структурами данных в Python.

**Задание**

1 вариант.

В данной лабораторной работе Вам предстоит реализовать связный однонаправленный список. Для этого необходимо реализовать 2 зависимых класса:

Node

Класс, который описывает элемент списка.

Он должен иметь 2 поля:

- data # Данные элемента списка, приватное поле.
- next # Ссылка на следующий элемент списка.

И следующие методы:

- __init__(self, data, next) - конструктор, у которого значения по умолчанию для аргумента next равно None.
- get_data(self) - метод возвращает значение поля data (это необходимо, потому что в идеале пользователь класса не должен трогать поля класса Node).
- __str__(self) - перегрузка стандартного метода __str__, который преобразует объект в строковое представление. Для данной лабораторной необходимо реализовать следующий формат перевода объекта класса Node в строку:

data: <node_data>, next: <node_next>”,

где <node_data> - это значение поля data объекта Node, <node_next> - это значение поля next объекта, на который мы ссылаемся, если он есть, иначе None.

Пример того, как должен выглядеть результат реализации __str__ см. ниже.

Пример того, как должен выглядеть вывод объекта:

```
node = Node(1)
print(node) # data: 1, next: None
node.next = Node(2, None)
print(node) # data: 1, next: 2
```

Linked List

Класс, который описывает связный однонаправленный список.

Он должен иметь 2 поля:

- head # Данные первого элемента списка.
- length # Количество элементов в списке.

И следующие методы:

- __init__(self, head) - конструктор, у которого значения по умолчанию для аргумента head равно None.

Если значение переменной head равна None, метод должен создавать пустой список.

Если значение head не равно None, необходимо создать список из одного элемента.

- __len__(self) - перегрузка метода __len__, он должен возвращать длину списка (этот стандартный метод, например, используется в функции len).
- append(self, element) - добавление элемента в конец списка. Метод должен создать объект класса Node, у которого значение поля data будет равно element и добавить этот объект в конец списка.
- __str__(self) - перегрузка стандартного метода __str__, который преобразует объект в строковое представление. Для данной лабораторной необходимо реализовать следующий формат перевода объекта класса однонаправленного списка в строку:

Если список пустой, то строковое представление:

LinkedList[]

Если не пустой, то формат представления следующий:

LinkedList[length = <len>, [data:<first_node>.data, next: <first_node>.data; data:<second_node>.data, next:<second_node>.data; … ; data:<last_node>.data, next: <last_node>.data], где <len> - длина связного списка, <first_node>, <second_node>, <third_node>, … , <last_node> - элементы однонаправленного списка.

Пример того, как должен выглядеть результат реализации см. ниже.

- pop(self) - удаление последнего элемента. Метод должен выбрасывать исключение IndexError с сообщением "LinkedList is empty!", если список пустой.
- clear(self) - очищение списка.
- delete_on_end(self, n) - удаление n-того элемента с конца списка. Метод должен выбрасывать исключение KeyError, с сообщением "Element doesn't exist!", если количество элементов меньше n.

Пример того, как должно выглядеть взаимодействие с Вашим связным списком:

```
linked_list = LinkedList()
print(linked_list) # LinkedList[]
print(len(linked_list)) # 0
linked_list.append(10)
print(linked_list) # LinkedList[length = 1, [data: 10, next: None]]
print(len(linked_list)) # 1
linked_list.append(20)
print(linked_list) # LinkedList[length = 2, [data: 10, next:20; data: 20, next: None]]
print(len(linked_list)) # 2
linked_list.pop()
print(linked_list)
print(linked_list) # LinkedList[length = 1, [data: 10, next: None]]
print(len(linked_list)) # 1
```

Вам не требуется реализовывать создание экземпляров ваших классов и вызов методов, это сделает проверяющая система.

В отчете вам требуется:

Указать, что такое связный список. Основные отличия связного списка от массива.

Указать сложность каждого метода.

Описать возможную реализацию бинарного поиска в связном списке. Чем отличается реализация алгоритма бинарного поиска для связного списка и для классического списка Python?

**Основные теоретические положения**

В Python существует множество алгоритмов и структур данных, которые могут быть использованы для эффективной обработки информации. Одним из основных принципов при работе с алгоритмами и структурами данных в Python является стремление к оптимальной производительности и минимальному потреблению ресурсов.

Основные теоретические положения об алгоритмах и структурах данных в Python включают в себя понятия такие как сложность времени и пространства, рекурсивные алгоритмы, сортировка и поиск, работа с массивами и списками, работа с деревьями и графами, а также различные методы оптимизации.

Python предоставляет широкий выбор встроенных алгоритмов и структур данных, таких как списки, словари, множества, кортежи, очереди, стеки, деревья и графы. Кроме того, существует возможность реализации собственных алгоритмов и структур данных с использованием встроенных инструментов языка.

Важно учитывать особенности реализации конкретных алгоритмов и структур данных в Python, чтобы добиться оптимальной производительности и эффективности работы программы.

**Выполнение работы**

В ходе выполнения работы были реализованы два класса: Node для представления элемента связного списка и LinkedList для описания самого списка. Класс Node имеет приватное поле __data для хранения данных элемента и ссылку next на следующий элемент. Он также реализует методы для получения данных (get_data) и для строкового представления элемента (__str__).

Класс LinkedList содержит поле head, указывающее на начало списка, и поле length, отражающее текущую длину списка. В конструкторе инициализируется список с заданным начальным элементом, если он передан, либо пустой список. Методы append добавляет элемент в конец списка, увеличивая length, pop удаляет последний элемент списка, а delete_on_end удаляет элемент на позиции n с конца списка.

Строковое представление списка (__str__) включает информацию о длине списка и о его элементах. Если список пуст, выводится LinkedList[]. Если есть элементы, формируется строка, содержащая данные и ссылки на следующие элементы.

Теперь ответим на вопросы:

Что такое связный список?

Связный список — это структура данных, состоящая из узлов, каждый из которых содержит как собственные данные, так и ссылку на следующий узел в последовательности. Это отличается от массива тем, что память для элементов списка выделяется динамически по мере добавления новых элементов, в отличие от массива, который использует непрерывную область памяти.

Основные отличия связного списка от массива?

В связном списке элементы располагаются в различных областях памяти, а для доступа к элементам используются указатели, в то время как в массиве элементы располагаются последовательно в непрерывной области памяти.

Размер связного списка может изменяться динамически, в то время как размер массива обычно фиксирован.

Вставка и удаление элементов в связном списке происходят эффективно в любом месте списка, в то время как в массиве операции вставки и удаления могут быть дорогими из-за необходимости сдвига других элементов.

Сложность каждого метода:

append: O(n) в худшем случае, где n — длина списка, так как при добавлении элемента нужно пройти весь список до его конца.

pop: O(n) в худшем случае, так как требуется пройти весь список до предпоследнего элемента.

delete_on_end: O(n), так как требуется пройти список до нужного элемента, который нужно удалить.

__len__: O(1), так как просто возвращает значение length.

__str__: O(n), так как требуется пройти весь список для формирования строкового представления.

Реализация бинарного поиска в связном списке:

Бинарный поиск требует доступа к элементам по индексу, что неэффективно для связного списка из-за того, что элементы хранятся не последовательно в памяти. Однако, если в списке узлы имеют какую-то сортировку, можно использовать модифицированный бинарный поиск. Например, можно начать с середины списка, сравнивать значения и двигаться вправо или влево, в зависимости от результатов сравнения, пока не будет найден нужный элемент или не будет понятно, что элемента в списке нет. Это отличается от бинарного поиска в массиве, где доступ осуществляется по индексу и проверяется условие на основе среднего элемента.

**Тестирование**

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

| № п/п | Входные данные | Выходные данные |
|---|---|---|
| 1. | linked_list = LinkedList()<br>print(linked_list) # LinkedList[]<br>print(len(linked_list)) # 0<br>linked_list.append(10)<br>print(linked_list) # LinkedList[length = 1, [data: 10, next: None]]<br>print(len(linked_list)) # 1<br>linked_list.append(20)<br>print(linked_list) # LinkedList[length = 2, [data: 10, next:20; data: 20, next: None]]<br>print(len(linked_list)) # 2<br>linked_list.pop()<br>print(linked_list)<br>print(linked_list) # LinkedList[length = 1, [data: 10, next: None]]<br>print(len(linked_list)) # 1 | LinkedList[]<br>0<br>LinkedList[length = 1, [data: 10, next: None]]<br>1<br>LinkedList[length = 2, [data: 10, next: 20; data: 20, next: None]]<br>2<br>LinkedList[length = 1, [data: 10, next: None]]<br>LinkedList[length = 1, [data: 10, next: None]]<br>1 |

**Выводы**

Была освоена работа с алгоритмами и структурами данными в Python, а также реализована программа с их использованием.

# ПРИЛОЖЕНИЕ А
# ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

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

	def get_data(self):
		return self.__data

	def __str__(self):
		if(self.next == None): return f"data: {self.__data}, next: None"
		return f"data: {self.__data}, next: {self.next.get_data()}"

	def set_data(self, data):
		self.__data = data

class LinkedList:
	def __init__(self, head=None):
		self.head = head
		self.length = 1 if head else 0

	def __len__(self):
		return self.length

	def append(self, element):
		if self.head is None:
			self.head = Node(element)
		else:
			current = self.head
			while current.next:
				current = current.next
			current.next = Node(element)
		self.length += 1

	def __str__(self):
		if self.length == 0:
			return "LinkedList[]"
		result = f"LinkedList[length = {self.length}, ["
		current = self.head
		while current:
			result += f"data: {current.get_data()}, next: {current.next.get_data() if current.next else None}; "
			current = current.next
		result = result[:-2]
		result += "]]"
		return result

	def pop(self):
		if self.head is None:
```

```
            raise IndexError("LinkedList is empty!")
        if self.head.next is None:
            self.head = None
        else:
            current = self.head
            while current.next.next:
                current = current.next
            current.next = None
        self.length -= 1

    def delete_on_end(self, n):
        if n > self.length or n <= 0:
            raise KeyError("Element doesn't exist!")
        else:
            current = self.head
            prev = None
            for _ in range(self.length - n):
                prev = current
                current = current.next
            if current.next is None:
                prev.next = None
            else:
                current.set_data(current.next.get_data())
                current.next = current.next.next
        self.length -= 1

    def clear(self):
        self.head = None
        self.length = 0
```