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

# ОТЧЕТ

**по лабораторной работе №2**

**по дисциплине «Информационные технологии»**

**ТЕМА: АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ В PYTHON**

Студент гр. 3341 _____ Рябов М.Л.

Преподаватель _____ Иванов Д.В.

Санкт-Петербург

2024

**Цель работы**

Изучить алгоритмы и структуры данных, реализовать программу на Python, содержащую в себе связный однонаправленный список, реализованный классами Node и LinkedList.

**Задание**

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).
- change_data(self, new_data) - метод меняет значение поля data объекта 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.
- change_on_start(self, n, new_data) - изменение поля data n-того элемента с НАЧАЛА списка на new_data. 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**

Линейный односвязный список – это структура данных, представляющая собой последовательность узлов, каждый из которых хранит какие-то полезные данные и указатель на следующий элемент. Важно, что в памяти элементы не находятся последовательно, в отличие от массивов.

Узел (Node): Узел представляет элемент списка и содержит две важные информации: данные и ссылку на следующий узел в списке.

Голова списка (Head): Голова списка - это первый элемент списка. Она содержит ссылку на начало списка.

Хвост списка (Tail): Хвост списка - это последний элемент списка. У хвоста нет ссылки на следующий элемент, поэтому его указатель обычно равен None.

Пустой список: Если список не содержит ни одного элемента, то его голова (head) указывает на None, что означает пустой список.

Операции над списком: Основные операции включают добавление элемента в начало списка (prepend), добавление элемента в конец списка (append), удаление элемента из списка (pop), поиск элемента в списке и т.д.

Сложность операций: Вставка и удаление элемента в начале списка имеют сложность O(1), так как требуется только изменение ссылок. Вставка и удаление элемента в конце списка имеют сложность O(n), так как приходится пройти весь список до последнего элемента.

Итерирование: Для прохода по всем элементам списка используется итерирование. Мы начинаем с головы списка и переходим от узла к узлу, пока не достигнем конца списка (пока ссылка не станет None).

**Выполнение работы**

1. Класс Node представляет узел односвязного списка. Каждый узел содержит две основные части: данные data и ссылку на следующий узел next. При создании объекта узла с помощью конструктора __init__, передаются данные и, опционально, ссылка на следующий узел. Метод get_data() позволяет получить данные узла, а метод change_data() используется для изменения этих данных. Метод __str__ возвращает строковое представление узла, включая данные и данные следующего узла, если таковой имеется.

2. Класс LinkedList представляет собой сам линейный список. В конструкторе __init__ инициализируется список с указанием головного узла head, который по умолчанию равен None. Метод __len__ возвращает текущую длину списка. Метод append добавляет новый узел с данными в конец списка. Метод pop удаляет последний узел из списка. Метод change_on_start изменяет данные узла на определенной позиции в списке. Метод clear очищает список, удаляя все узлы.

1) Основные отличия связного списка от массива:

1. Память: В массиве элементы хранятся последовательно в памяти, что обеспечивает быстрый доступ к элементам по индексу. В связном списке элементы могут храниться в разных областях памяти, и доступ к элементам осуществляется последовательно от начала списка.

2. Вставка и удаление: В связном списке операции вставки и удаления элементов выполняются быстрее, чем в массиве, так как не требуется перемещать все элементы после вставляемого или удаляемого элемента. В массиве при вставке или удалении элемента может потребоваться перемещение всех последующих элементов.

3. Динамический размер: Размер связного списка может меняться динамически, в отличие от массива, который обычно имеет фиксированный размер.

2) Сложность каждого использованного метода:

1. Метод __len__: Возвращает текущую длину списка. Сложность этого метода - O(1), так как просто возвращает значение атрибута self.length.

2. Метод append: Добавляет новый элемент в конец списка. Сложность этого метода - O(n), где n - длина списка, так как требуется проход от головы списка до последнего элемента.

3. Метод __str__: Возвращает строковое представление списка. Сложность этого метода - O(n), где n - длина списка, так как нужно пройти по всем элементам списка для формирования строки.

4. Метод pop: Удаляет последний элемент списка. Сложность этого метода - O(n), где n - длина списка, так как требуется проход к предпоследнему элементу для обновления ссылки на последний элемент.

5. Метод change_on_start: Изменяет данные узла на определенной позиции в списке. Сложность этого метода - O(n), где n - позиция узла, так как нужно выполнить проход к указанной позиции в списке.

6. Метод clear: Очищает список, удаляя все элементы. Сложность этого метода - O(n), где n - длина списка, так как нужно удалить каждый элемент списка.

Сложность методов get_data, change_data и __str__ в классе Node - O(1), так как они выполняются за постоянное время без зависимости от размера списка.

3) Бинарный поиск - это эффективный алгоритм поиска элемента в отсортированном массиве или списке. Он работает путем разделения массива на две части и последующего сравнения искомого элемента с элементом в середине массива. Если элемент найден, поиск завершается. В противном случае поиск продолжается в подмассиве, который может содержать искомый элемент.

Однако в связном списке бинарный поиск не так просто реализовать из-за его структуры. Основные причины, по которым реализация бинарного поиска для связного списка отличается от классического списка Python, включают:

1. Отсутствие прямого доступа к элементам по индексу: В связном списке нет прямого доступа к элементам по индексу, поэтому невозможно быстро получить элемент в середине списка, что необходимо для бинарного поиска. Вместо этого, необходимо последовательно переходить к нужным узлам списка.

2. Линейное время доступа к среднему элементу: Для выполнения бинарного поиска необходимо получить средний элемент списка. В классическом массиве это может быть сделано за O(1) времени, так как можно использовать индексацию. Однако в связном списке необходимо выполнить линейный проход от начала списка к середине, что требует O(n) времени в худшем случае.

3. Неэффективность поиска по сравнению с линейным поиском: В связном списке бинарный поиск не всегда более эффективен, чем линейный поиск из-за необходимости многократных линейных проходов по списку для получения среднего элемента.

Тем не менее, в некоторых случаях, когда связный список реализует интерфейс поиска по индексу или имеет специальные свойства, такие как упорядоченность элементов, можно реализовать некоторые варианты бинарного поиска. Например, можно использовать модифицированный бинарный поиск, который использует средний элемент как опорный элемент, а затем выполняет поиск справа или слева от опорного элемента, в зависимости от значения. Однако это будет менее эффективно, чем в случае с массивом.

**Тестирование**

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

| № п/п | Входные данные | Выходные данные | Комментарии |
|---|---|---|---|
| 1. | linked_list = LinkedList()<br>print(linked_list)<br>print(len(linked_list))<br>linked_list.append(10)<br>print(linked_list)<br>print(len(linked_list))<br>linked_list.append(20)<br>print(linked_list)<br>print(len(linked_list))<br>linked_list.pop()<br>print(linked_list)<br>print(linked_list)<br>print(len(linked_list)) | 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 | Проверка с emoevm |
| 2. | test_list = LinkedList()<br>try:<br>test_list.pop()<br>except IndexError as e:<br>print("Попытка вызвать метод pop на пустом списка: ", e)<br>test_list.append(5)<br>try:<br>test_list.change_on_start(1, 10)<br>except KeyError as e:<br>print("Попытка изменить элемент в начале списка", e)<br>try: | Попытка вызвать метод pop на пустом списка: LinkedList is empty!<br>Попытка изменить элемент в середине списка: "Element doesn't exist!"<br>Попытка изменить элемент за пределами списка: "Element doesn't exist!"<br>Попытка вызвать метод pop на пустом списке после очистки: LinkedList is empty! | Этот код попробует выполнить операции с пустым списком, изменить элементы списка в разных позициях, а затем очистить список и снова попытаться выполнить операцию, чтобы проверить, что методы обрабатывают граничные случаи корректно. |

| | | | |
|---|---|---|---|
| | test_list.change_on_start(2, 20)<br>except KeyError as e:<br>print("Попытка изменить элемент в середине списка:", e)<br>try:<br>test_list.change_on_start(5, 50)<br>except KeyError as e:<br>print("Попытка изменить элемент за пределами списка:", e)<br>test_list.clear()<br>try:<br>test_list.pop()<br>except IndexError as e:<br>print("Попытка вызвать метод pop на пустом списке после очистки:", e) | | |

**Выводы**

В рамках данной задачи был реализован связанный однонаправленный список через два класса: Node и LinkedList.

## 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 is None:
                  return f"data: {self.get_data()}, next: None"
              else:
                                  return  f"data:  {self.get_data()},  next:
{self.next.get_data()}"


      class LinkedList:

          def __init__(self, head=None):
              self.head = head
              self.length = 1 if head is not None else 0

          def __len__(self):
              return self.length

          def append(self, element):
              self.length += 1
              if self.length == 1:
                  self.head = Node(element)
              else:
                  current = self.head
                  while current.next is not None:
                      current = current.next
                  current.next = Node(element)

          def __str__(self):
              if self.length == 0:
                  return "LinkedList[]"
              else:
                  s = f"LinkedList[length = {self.length}, ["
                  current = self.head
                  while current.next is not None:
                      s += f"{current.__str__()}; "
                      current = current.next
                  s += current.__str__()
                  s += "]]"
              return s
```

```
def pop(self):
    if self.length == 0:
        raise IndexError("LinkedList is empty!")
    elif self.length == 1:
        self.head = None
        self.length -= 1
    else:
        current, del_elem = self.head, self.head
        while del_elem.next is not None:
            del_elem = del_elem.next
        while current.next != del_elem:
            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!")
    elif n == self.length:
        self.head = self.head.next
        self.length -= 1
    else:
        current = self.head
        current_n = self.length
        while (current_n - 1) != n:
            current_n -= 1
            current = current.next
        current.next = current.next.next
        self.length -= 1

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