

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

ОТЧЕТ
по лабораторной работе №2
по дисциплине «Информационный технологии»
Тема: Алгоритмы и структуры данных в Python

Студент гр. 3344		Тукалкин. В.А.
Преподаватель		Иванов Д.В.

Санкт-Петербург
2024

Цель работы

Ознакомиться с алгоритмами и структурами данных в языке программирования Python.

Задание.

Вариант 2.

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:

- 1) `data` # Данные элемента списка, приватное поле.
- 2) `next` # Ссылка на следующий элемент списка.

И следующие методы:

- 1) `__init__(self, data, next)` - конструктор, у которого значения по умолчанию для аргумента `next` равно `None`.
- 2) `get_data(self)` - метод возвращает значение поля `data` (это необходимо, потому что в идеале пользователь класса не должен трогать поля класса `Node`).
- 3) `__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`.

Linked List

Класс, который описывает связный однонаправленный список.

Он должен иметь 2 поля:

- 1) `head` # Данные первого элемента списка.

2) `length` # Количество элементов в списке.

И следующие методы:

- 1) `__init__(self, head)` - конструктор, у которого значения по умолчанию для аргумента `head` равно `None`. Если значение переменной `head` равно `None`, метод должен создавать пустой список. Если значение `head` не равно `None`, необходимо создать список из одного элемента.
- 2) `__len__(self)` - перегрузка метода `__len__`, он должен возвращать длину списка (этот стандартный метод, например, используется в функции `len`).
- 3) `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.
- 4) `__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.
- 5) `pop(self)` - удаление последнего элемента. Метод должен выбрасывать исключение `IndexError` с сообщением `"LinkedList is empty!"`, если список пустой.
- 6) `clear(self)` - очищение списка.

- 7) `delete_on_start(self, n)` - удаление `n`-того элемента с НАЧАЛА списка.
- Метод должен выбрасывать исключение `KeyError`, с сообщением "Element doesn't exist!", если количество элементов меньше `n`.

Выполнение работы

1) Связный список — базовая динамическая структура данных в информатике, состоящая из узлов, содержащих данные и ссылки («связки») на следующий и/или предыдущий узел списка. Основные отличия связного списка от массива:

- Хранение данных: массив хранит элементы в последовательном порядке в непрерывном участке памяти, где каждый элемент имеет свой индекс»; связный список хранит элементы в произвольном порядке в виде последовательности узлов, каждый из которых содержит данные и указатель на следующий узел;
- Доступ к элементам: доступ к элементам массива осуществляется по индексу элемента; доступ к элементам связного списка осуществляется путем прохода по указателям на узлы, начиная с начального узла;
- Производительность операций: вставка в начало (массив — $O(n)$, список — $O(1)$), доступ по индексу ($O(1)$, $O(n)$), удаление из начала ($O(n)$, $O(1)$), длина ($O(1)$, $O(n)$).

2) Сложность методов:

Класс Node:

- `__init__` — $O(1)$;
- `get_data` — $O(1)$;
- `__str__` — $O(1)$.

Класс LinkedList:

- `__init__` — $O(n)$;
- `__len__` — $O(1)$;
- `append` — $O(n)$;
- `__str__` — $O(n)$;
- `pop` — $O(n)$;

- `delete_on_start` – $O(n)$;
- `clear` – $O(1)$.

3) Алгоритм бинарного поиска неэффективен при использовании в связных списках, так как в них нет возможности индексации. Поиск центрального элемента, необходимого для работы алгоритма, потребует времени $O(n)$. Поэтому проход по каждому элементу списка будет значительно быстрее и эффективнее, чем применение бинарного поиска.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	<pre>node = Node(1) print(node) node.next = Node(2, 1 None) print(node) print(node.get_data()) l_1 = LinkedList() print(l_1) print(len(l_1)) l_1.append(10) l_1.append(20) l_1.append(30) l_1.append(40) print(l_1) print(len(l_1)) l_1.delete_on_start(2) print(l_1)</pre>	<pre>data: 1, next: None data: 1, next: 2 1 LinkedList[] 0 LinkedList[length = 4, [data: 10, next: 20; data: 20, next: 30; data: 30, next: 40; data: 40, next: None]] 4 LinkedList[length = 3, [data: 10, next: 30; data: 30, next: 40; data: 40, next: None]]</pre>	Верный ответ

Выводы

Были изучены алгоритмы и структуры данных в, на примере программы, выполняющей с операции с связным однонаправленным списком в Python.

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

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

    def __len__(self):
        return self.length

    def append(self, element):
        tmp=Node(element)
        if self.length==0:
            self.head=tmp
            self.length=1
            return self.head
        NextEl=self.head
        while NextEl.next!=None: NextEl=NextEl.next
        NextEl.next=tmp
        self.length+=1

    def __str__(self):
        if self.length==0:
            return "LinkedList[]"
        arr=[]
        NextEl=self.head
        arr.append(f"data: {NextEl.data}, next: {None if
NextEl.next==None else NextEl.next.get_data()}")
        while NextEl.next!=None:
            NextEl=NextEl.next
            arr.append(f"data: {NextEl.get_data()}, next: {None if
NextEl.next==None else NextEl.next.get_data()}")
        return f"LinkedList[length = {self.length}, [{';
'.join(arr)}]]"

    def pop(self):
        if self.head==None: raise IndexError("LinkedList is
empty!")
```

```

else:
    NextEl = self.head
    prev = NextEl
    while NextEl.next!=None:
        prev=NextEl
        NextEl=NextEl.next
    prev.next=None
    self.length-=1

def delete_on_start(self, n):
    if n>self.length or n<=0:
        raise KeyError("Element doesn't exist!")
    NextEl=self.head
    count=1
    prev=NextEl
    while count!=n:
        prev=NextEl
        NextEl=NextEl.next
        count+=1
    if n==1 and self.head.next!=None:
        self.head=self.head.next
    prev.next=NextEl.next
    self.length-=1

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

```