

Министерство цифрового развития связи и массовых коммуникаций РФ

Государственное бюджетное образовательное учреждение высшего  
образования

Ордена Трудового Красного Знамени

«Московский технический университет связи и информатики»

Кафедра «Математическая кибернетика и информационные технологии»

дисциплина «Структуры и алгоритмы обработки данных»

Отчет по лабораторной работе №2

«Методы поиска»

Подготовил: студент группы

БВТ1903 Саввин Д.И.

Проверил: Кутейников И.А.

Москва

2021

## Оглавление.

1. ЦЕЛЬ РАБОТЫ .....	3
2. ВЫПОЛНЕНИЕ .....	4
3. ВЫВОД.....	17

## 1. ЦЕЛЬ РАБОТЫ

Реализовать методы поиска в соответствии с заданием. Организовать генерацию начального набора случайных данных. Для всех вариантов добавить реализацию добавления, поиска и удаления элементов. Оценить время работы каждого алгоритма поиска и сравнить его со временем работы стандартной функции поиска, используемой в выбранном языке программирования.

Задание№1:

- Бинарный поиск;
- Бинарное дерево;
- Фибоначчиев;
- Интерполяционный;

Задание№2:

- Простое рехэширование;
- Рехэширование с помощью псевдослучайных чисел;
- Метод цепочек;

Задание№3:

-Расставить на стандартной 64-клеточной шахматной доске 8 ферзей так, чтобы ни один из них не находился под боем другого». Подразумевается, что ферзь бьёт все клетки, расположенные по вертикалям, горизонталям и обеим диагоналям.

## 2. ВЫПОЛНЕНИЕ

### Задание №1.

Ниже представлена функция бинарного поиска, параметры `array` и `value` – список и значение элемента который мы хотим найти. При удачном поиске функция возвращает индекс искомого элемента, а при неудачном -1.

```
# Binary_Search
def binarySearch(array, value):
    lowBound = 0
    upBound = len(array) - 1
    while lowBound <= upBound:
        center = (lowBound + upBound) // 2
        if array[center] == value:
            return center
        elif array[center] > value:
            upBound = center - 1
        elif array[center] < value:
            lowBound = center + 1
    return -1
```

На рисунках 1 и 2 изображены результаты работы функции `binarySearch()`, с случайно сгенерированными параметрами.

Сложность алгоритма –  $O(\log(n))$

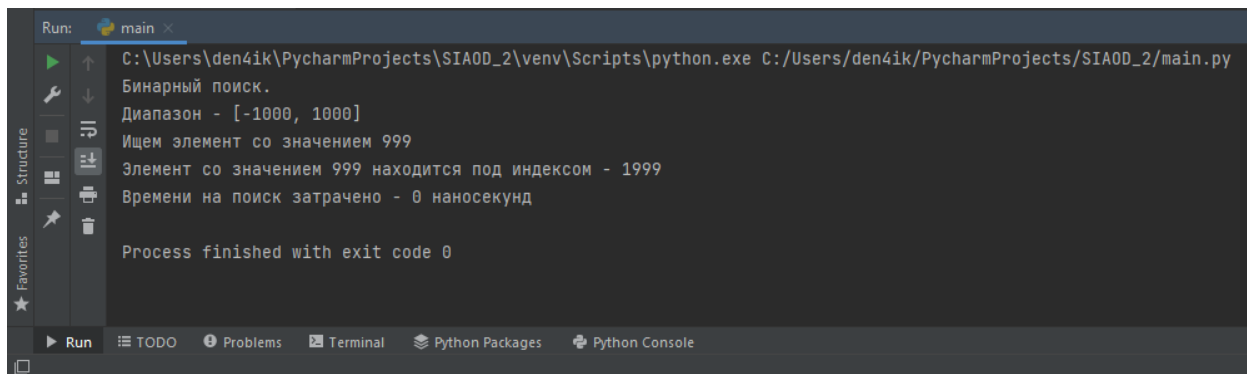


Рис. 1 – Результат работы `binarySearch()`.

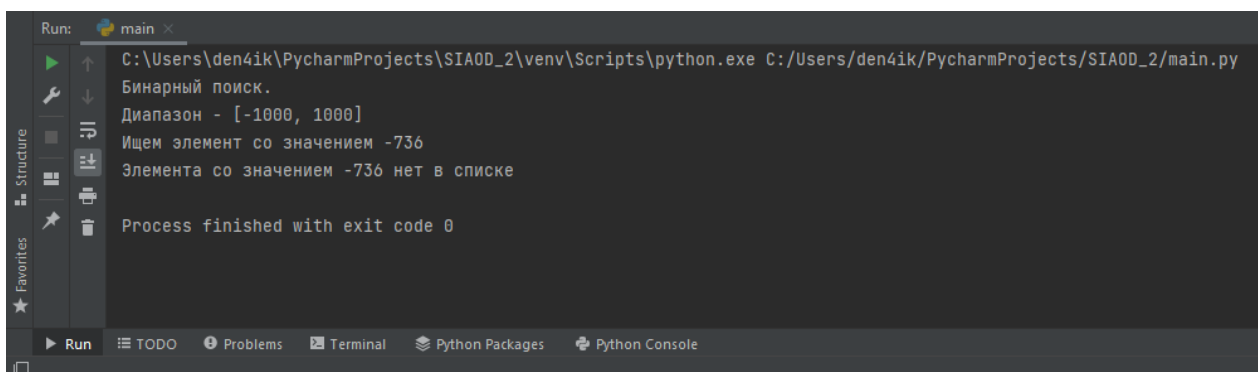


Рис. 2 – Результат работы `binarySearch()`.

Ниже представлен код классов `BinarTreeNode` и `BinarTree`, где `BinarTreeNode` – является классом узла бинарного дерева, а `BinarTree` – описывает само бинарное дерево.

#### `BinarTreeNode.py`

```
class BinarTreeNode:

    #Конструктор
    def __init__(self, value):
        self.value = value
        self.rod = None
        self.left = None
        self.right = None

    def setValue(self, value):
        self.value = value

    def setRod(self, node):
        self.rod = node

    def setLeft(self, node):
        self.left = node

    def setRight(self, node):
        self.right = node

    def getValue(self):
        return self.value

    def getRod(self):
        return self.rod

    def getLeft(self):
        return self.left

    def getRight(self):
        return self.right
```

## BinarTree.py

```
from BinarTreeNode import BinarTreeNode

class BinarTree:

    #Конструктор
    def __init__(self, node):
        self.root = node

    #Метод добавления значения в дерево
    def add(self, value):
        node = self.root
        while True:
            if value < node.value:
                if node.left:
                    node = node.left
                else:
                    node.left = BinarTreeNode(value)
                    node.left.rod = node
                    break
            elif value > node.value:
                if node.right:
                    node = node.right
                else:
                    node.right = BinarTreeNode(value)
                    node.right.rod = node
                    break
            elif value == node.value:
                break

    #Метод поиска в дереве по значению, возвращает поколение
    def find(self, value):
        pokoleniye = 0
        node = self.root
        while True:
            if node.value == value:
                return pokoleniye
                break
            if node.value > value:
                if node.left:
                    node = node.left
                    pokoleniye += 1
                else:
                    return -1
            if node.value < value:
                if node.right:
                    node = node.right
                    pokoleniye += 1
                else:
                    return -1

    #Метод удаления элемента по значению
    def remove(self, value):
        node = self.root
        arr = [self.root]
        while True:
            if node.value > value:
                if node.left:
                    node = node.left
                else:
                    return -1
            elif node.value < value:
                if node.right:
```

```

        node = node.right
    else:
        return -1
    elif node.value == value:
        array = self.copyToArray([], self.root)
        array.remove(value)
        newself = BinarTree(BinarTreeNode(array[0]))
        for i in range(1, len(array)):
            newself.add(array[i])
        break
    self.root = newself.root

#Вспомогательный метод записывающий все элементы дерева в массив
def copyToArray(self, array, node):
    arr = array
    if node:
        arr.append(node.value)
        self.copyToArray(arr, node.left)
        self.copyToArray(arr, node.right)
    return arr

```

Ниже представлены рисунки с результатами прохода по дереву методом pre\_order, а также добавления, поиска(возвращает инструкцию пути к элементу если он есть), и удаления элементов из дерева.

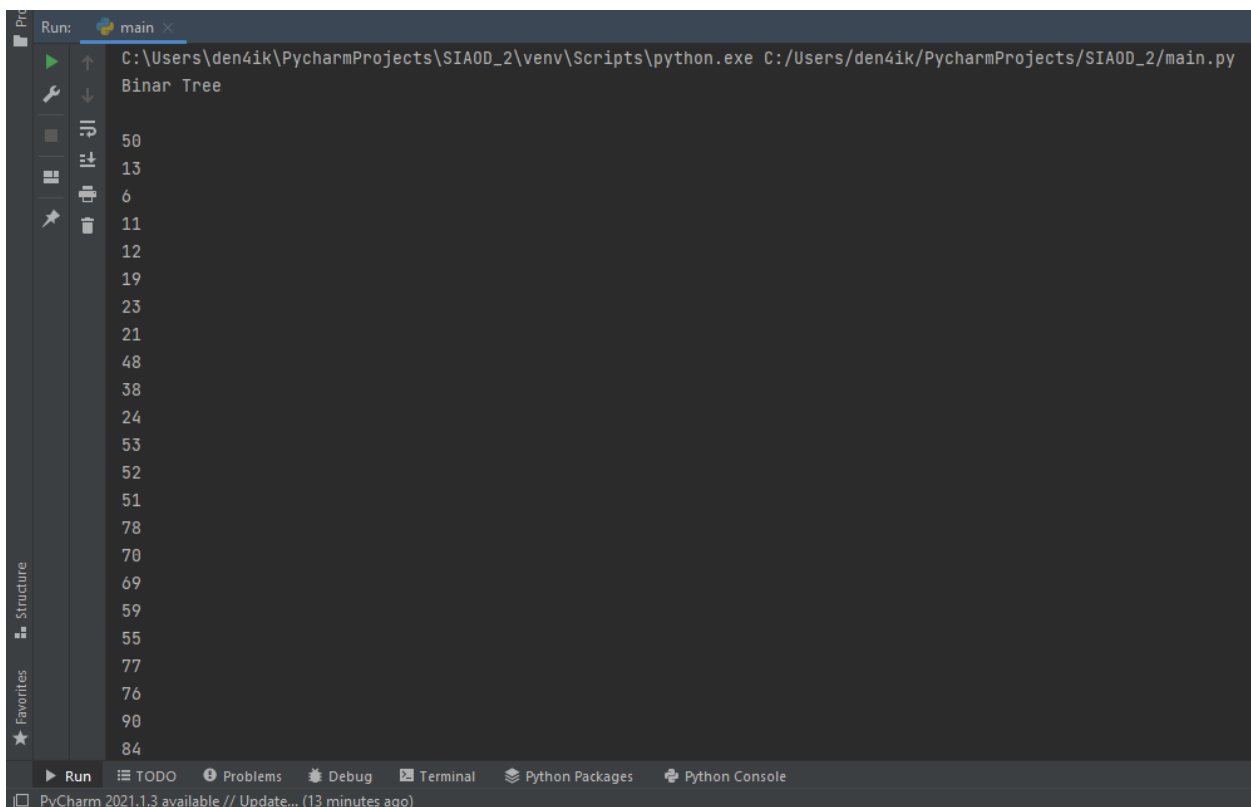


Рис. 1 – Результат работы дерева.

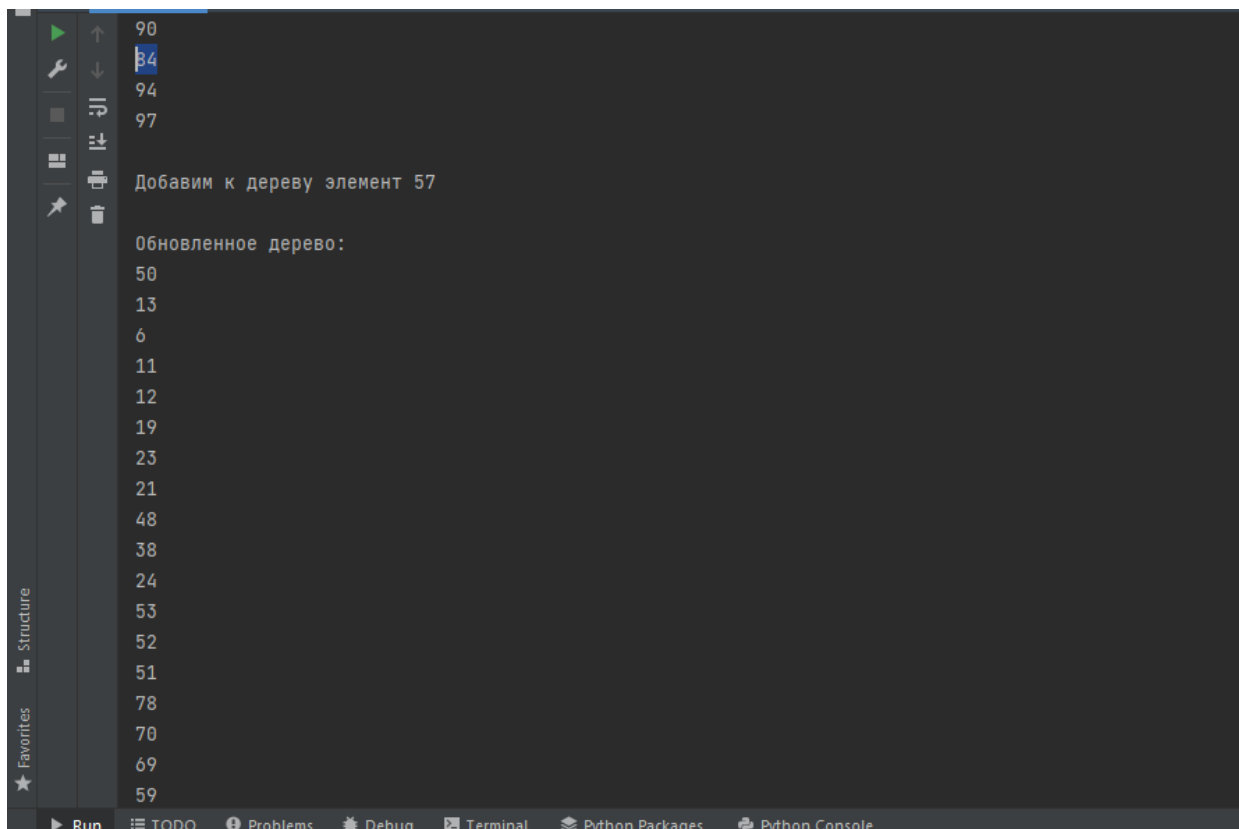


Рис. 2 – Результат работы дерева.

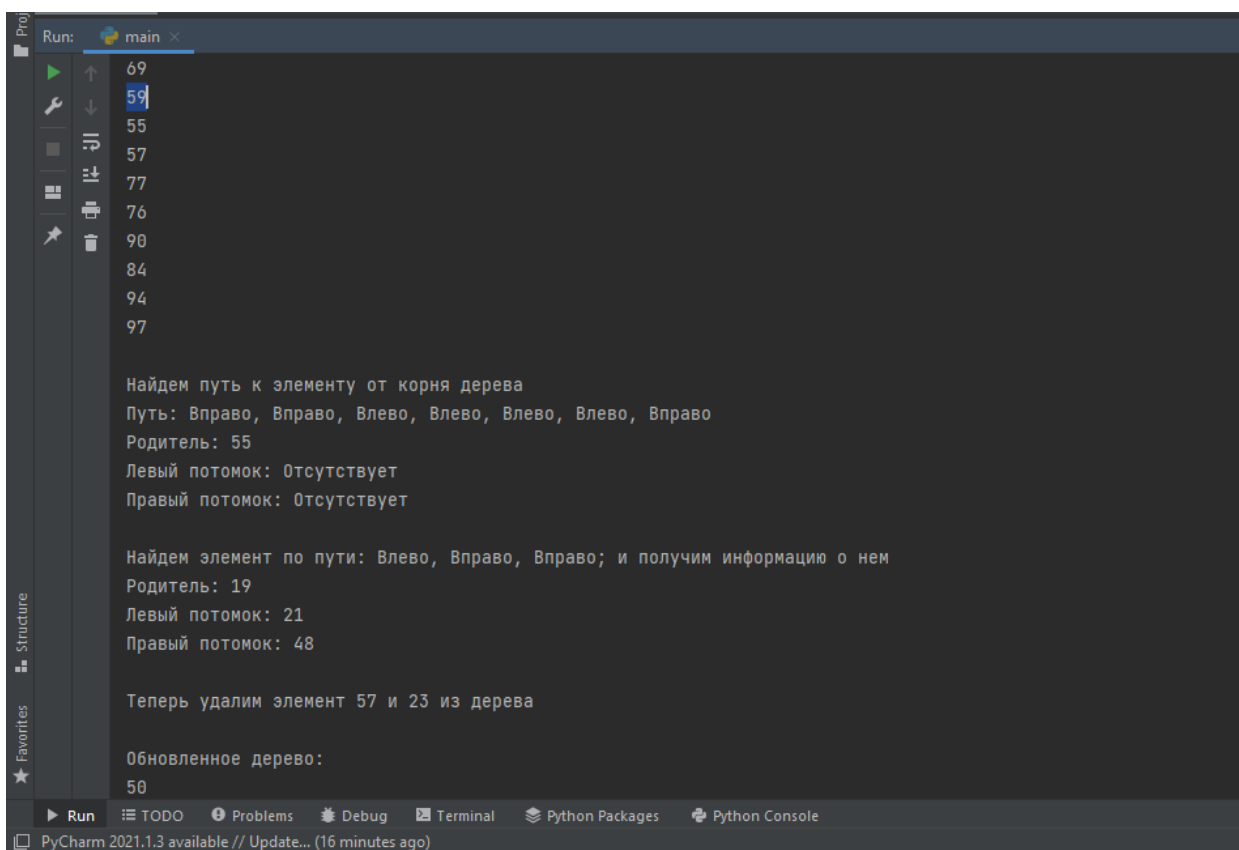


Рис. 3 – Результат работы дерева.



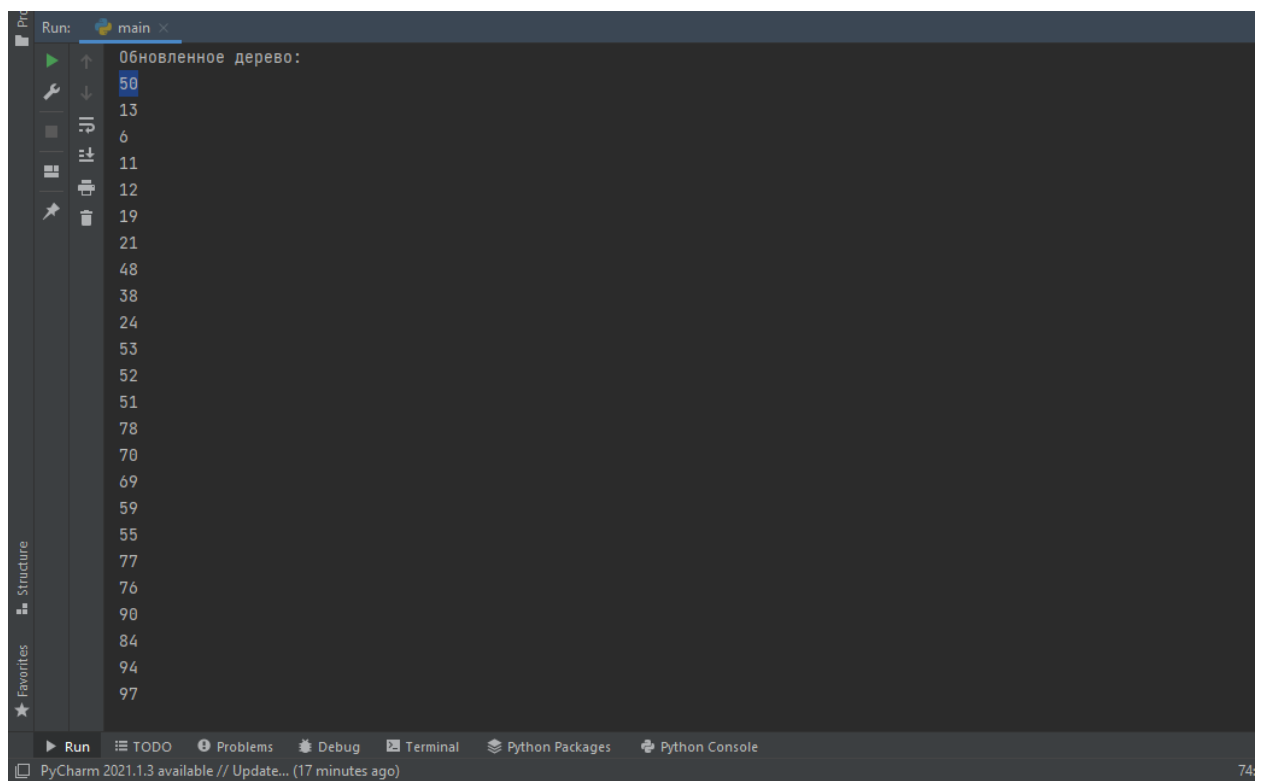


Рис. 4 – Результат работы дерева.

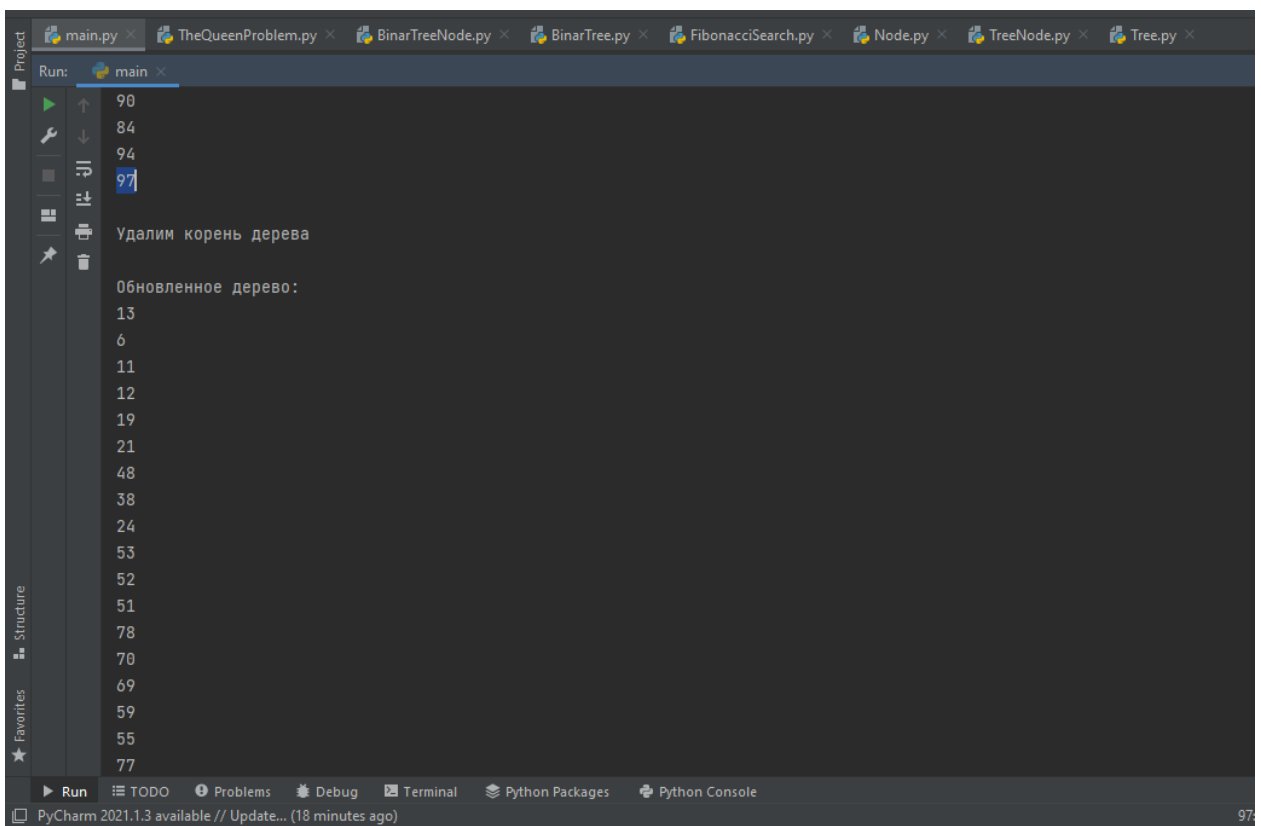


Рис. 5 – Результат работы дерева.

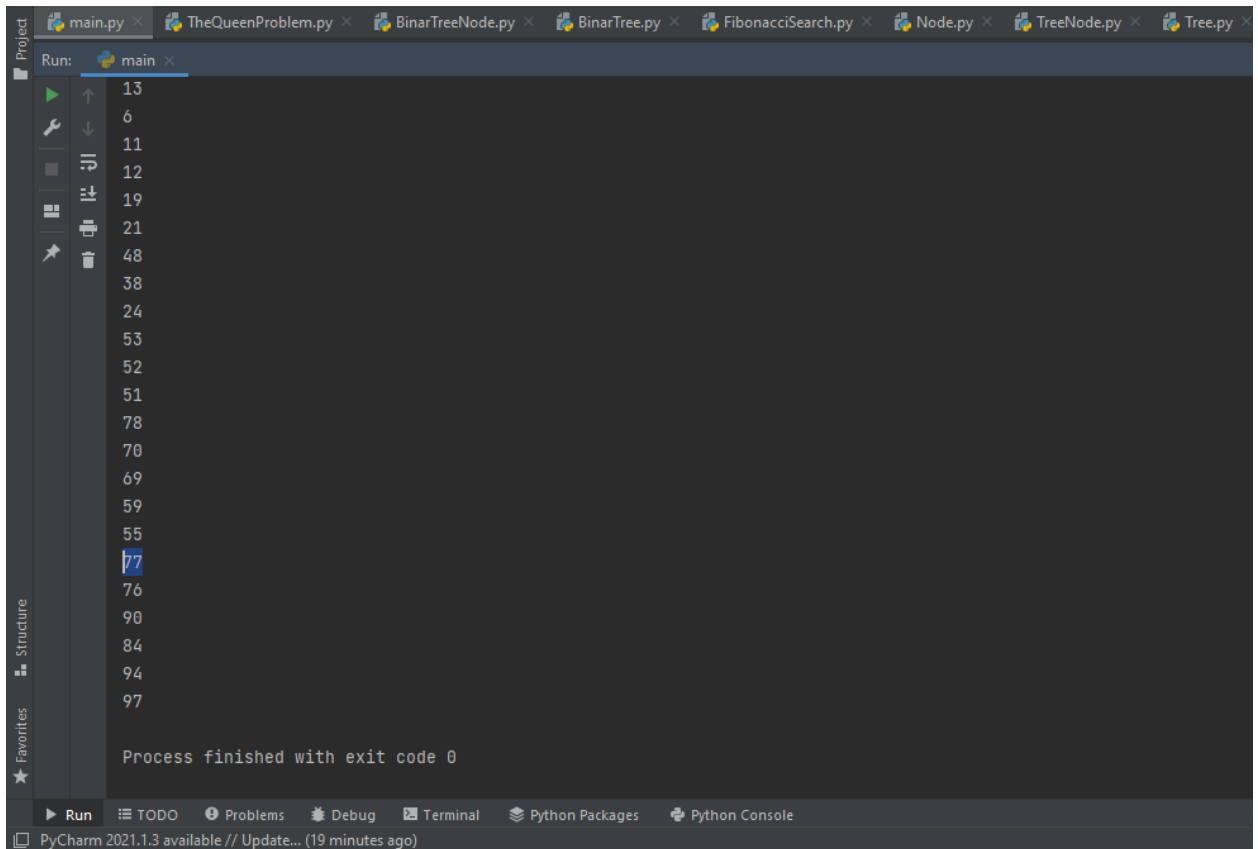


Рис. 6 – Результат работы дерева.

Сложность алгоритма в общем случае –  $O(h)$ , где  $h$  зависит от высоты.

Ниже представлен код класса `FibonacciSearch.py`, который описывает Фибоначчиев метод поиска.

```
class FibonacciSearch:
    def __init__(self):
        self.i = 0
        self.q = 0
        self.p = 0
        self.stop = False

    #FibonacciNum
    def F(self, val):
        a = val
        if val > 1:
            a = 0
            arr = [0, 1]
            for i in range(2, val + 1):
                a = arr[i - 2] + arr[i - 1]
                arr.append(a)
            return a

    def startInit(self, array):
        self.stop = False
        k = 0
        n = len(array)
        while (self.F(k+1) < n):
            k += 1
        m = self.F(k+1) - (n+1)
```

```

        self.i = self.F(k)
        self.q = self.F(k-2)
        self.p = self.F(k-1)

    def upIndex(self):
        if self.p == 1:
            self.stop = True
        self.i = self.i + self.q
        self.p = self.p - self.q
        self.q = self.q - self.p

    def downIndex(self):
        if self.q == 0:
            self.stop = True
        self.i = self.i - self.q
        temp = self.q
        self.q = self.p - self.q
        self.p = temp

    def search(self, array, value):
        self.startInit(array)
        result_index = -1
        while not self.stop:
            if self.i < 0:
                self.upIndex()
            elif self.i >= len(array):
                self.downIndex()
            elif array[self.i] == value:
                result_index = self.i
                break
            elif value < array[self.i]:
                self.downIndex()
            elif value > array[self.i]:
                self.upIndex()
        return result_index

```

Результат работы Фибоначчиева поиска изображен на рисунке 7.

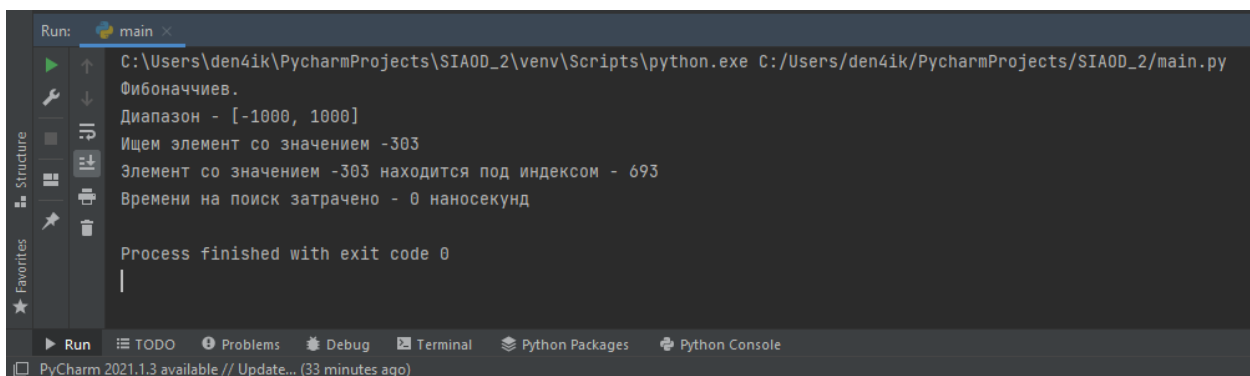


Рис. 7 – Результат работы Фибоначчиева метода поиска.

Сложность алгоритма –  $O(2^n)$

Ниже представлен код функции `interpolation_search()`, отвечающий за интерполяционный поиск указанного элемента в указанном списке.

```
# Interpolation_Search
def interpolation_search(lst, val):
    low = 0
    high = len(lst) - 1
    search_res = False
    index = -1

    while (low <= high) and (val >= lst[low]) and (val <= lst[high]) and not search_res:
        middle = low + int(((high - low) / (lst[high] - lst[low])) * (val - lst[low]))
        guess = lst[middle]
        if guess == val:
            search_res = True
            index = middle
        if guess < val:
            low = middle + 1
        if guess > val:
            high = middle - 1
    return index
```

Результат работы функции `interpolation_search()` изображен на рисунке ниже.

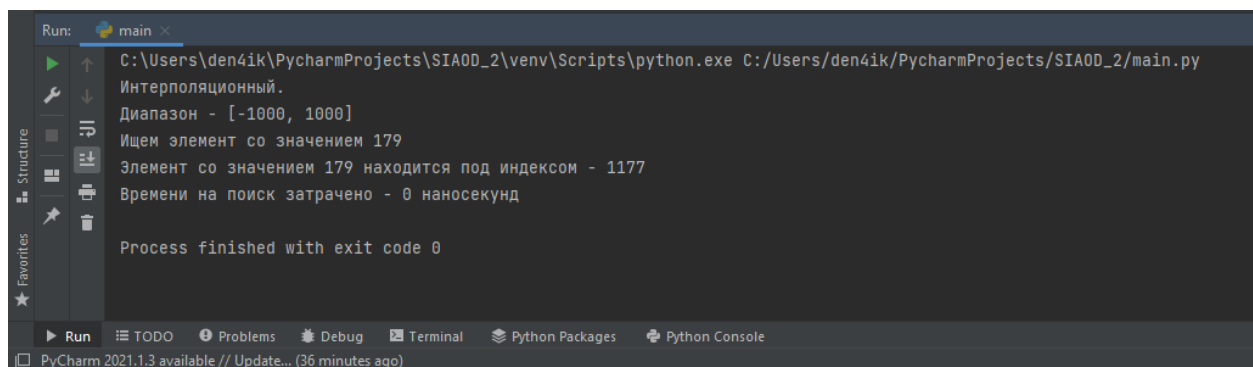


Рис. 8 – Результат работы `interpolation_search()`.

Сложность алгоритма –  $O(\log(\log(n)))$ .

Результат работы стандартной функции поиска Python представлен на рисунке 8.1.

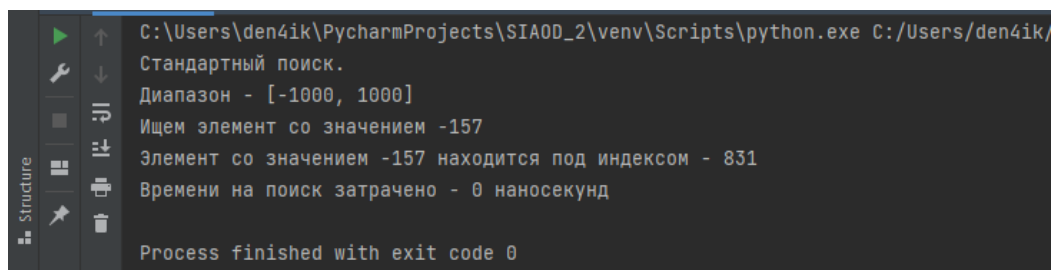


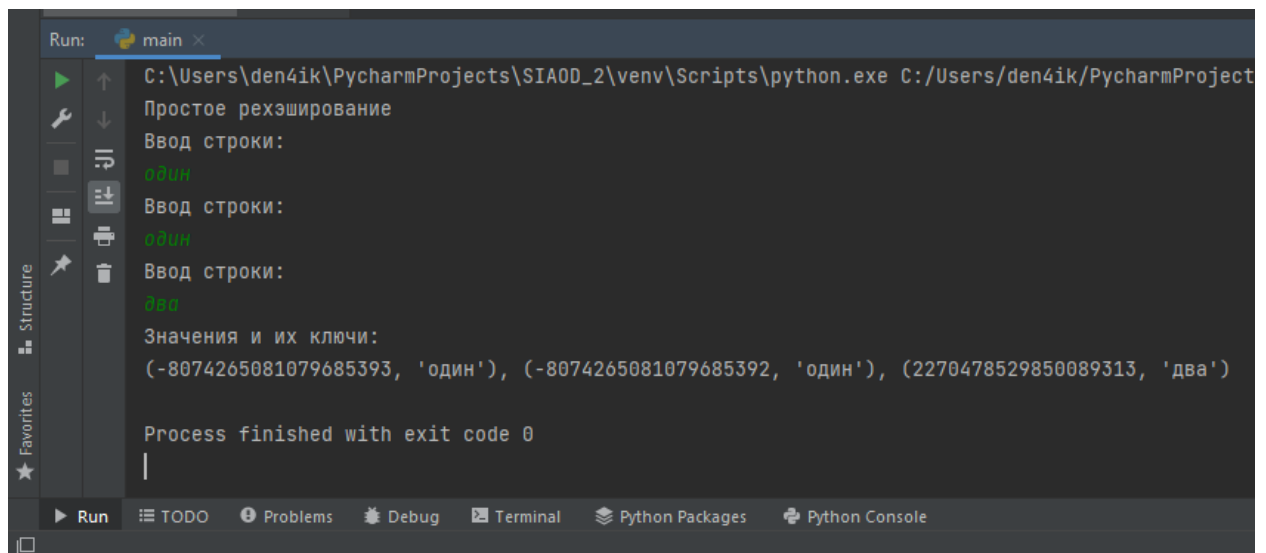
Рис. 8.1 – Результат использования стандартной функции поиска.

## Задание №2.

Ниже представлен код функции простого рехэширования `simple_re_hash()` и результат ее работы.

```
# Простое рехэширование
def simple_re_hash(sl, value):
    temp = str(value)
    i = 1
    while True:
        if hash(temp) not in sl.keys():
            sl[hash(temp)] = value
            break
        else:
            while hash(temp) + i in sl.keys():
                i += 1
            sl[hash(temp) + i] = value
            break
```

Рис. 9 – Функция `simple_re_hash()`.



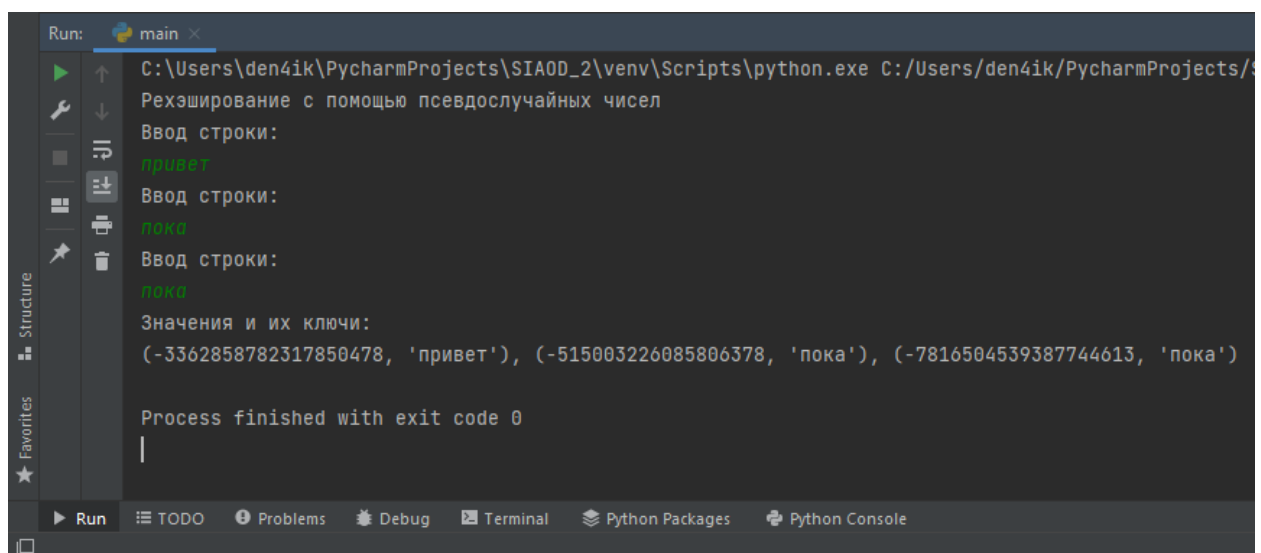
```
Run: main x
C:\Users\den4ik\PycharmProjects\SIA00_2\venv\Scripts\python.exe C:/Users/den4ik/PycharmProject
Простое рехэширование
Ввод строки:
один
Ввод строки:
один
Ввод строки:
два
Значения и их ключи:
(-8074265081079685393, 'один'), (-8074265081079685392, 'один'), (2270478529850089313, 'два')
Process finished with exit code 0
```

Рис. 10 – Результат работы простого рехэширования.

Ниже представлен код рехэширования с помощью псевдослучайных чисел, а также результат работы данной функции.

```
# Рехэширование с помощью псевдослучайных чисел
def rand_re_hash(sl, value):
    temp = value
    while True:
        if hash(temp) not in sl.keys():
            sl[hash(temp)] = value
            break
        else:
            temp += str(randint(0, 1000))
```

Рис. 11 – Функция rand\_re\_hash().



```
Run: main x
C:\Users\den4ik\PycharmProjects\SIA00_2\venv\Scripts\python.exe C:/Users/den4ik/PycharmProjects/SIA00_2/main.py
Рехэширование с помощью псевдослучайных чисел
Ввод строки:
привет
Ввод строки:
пока
Ввод строки:
пока
Значения и их ключи:
(-3362858782317850478, 'привет'), (-515003226085806378, 'пока'), (-7816504539387744613, 'пока')
Process finished with exit code 0
```

Рис.12 – Результат работы rand\_re\_hash().

Ниже представлен код и результат работы функции chain\_method().

```
# Метод цепочек
def chain_method(sl, value):
    temp = value
    if hash(temp) in sl.keys():
        if isinstance(sl[hash(temp)], deque):
            sl[hash(temp)].append(value)
        else:
            a = sl[hash(temp)]
            sl[hash(temp)] = deque([a, value])
    else:
        sl[hash(value)] = value
```

Рис.13 – Функция chain\_method().

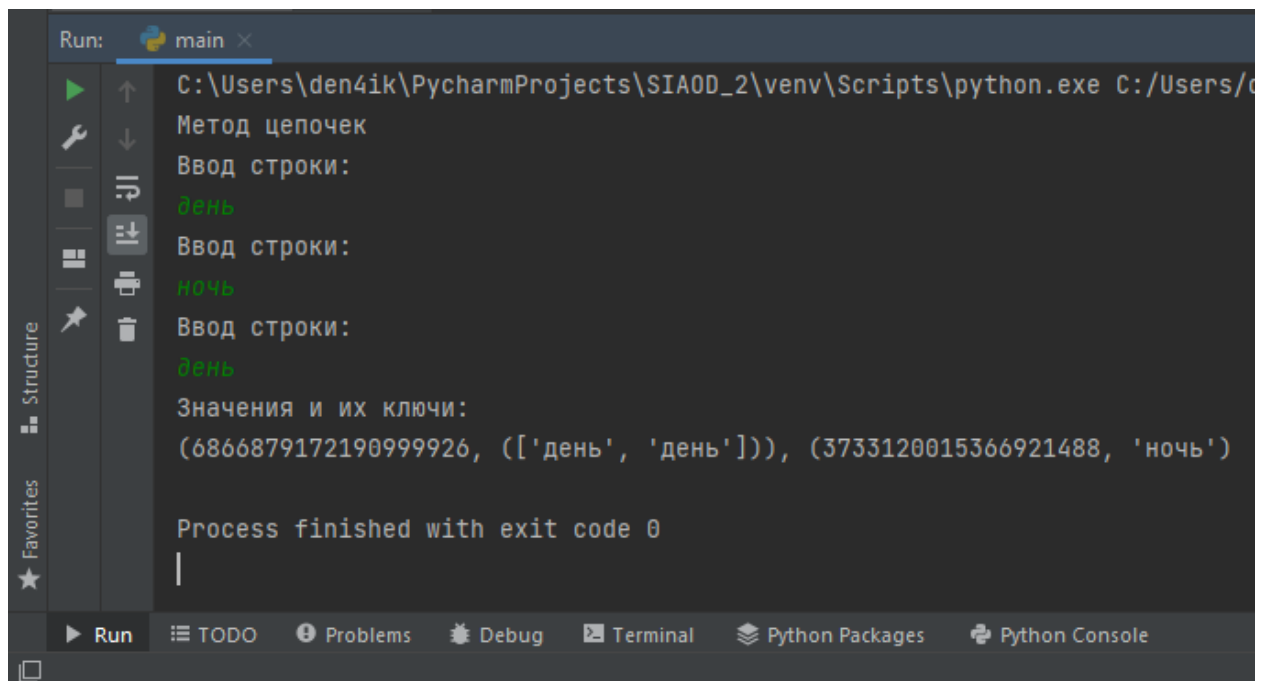


Рис.14 – Результат работы chain\_method().

### Задание №3.

Ниже представлен код класса TheQueenProblem, описывающего задачу о 8 ферзях.

```
import numpy as np

class TheQueenProblem:
    # Конструктор
    def __init__(self):
        self.field = np.zeros((8,8))
        for i in range(0,8):
            for j in range(0,8):
                self.field[i][j] = 0

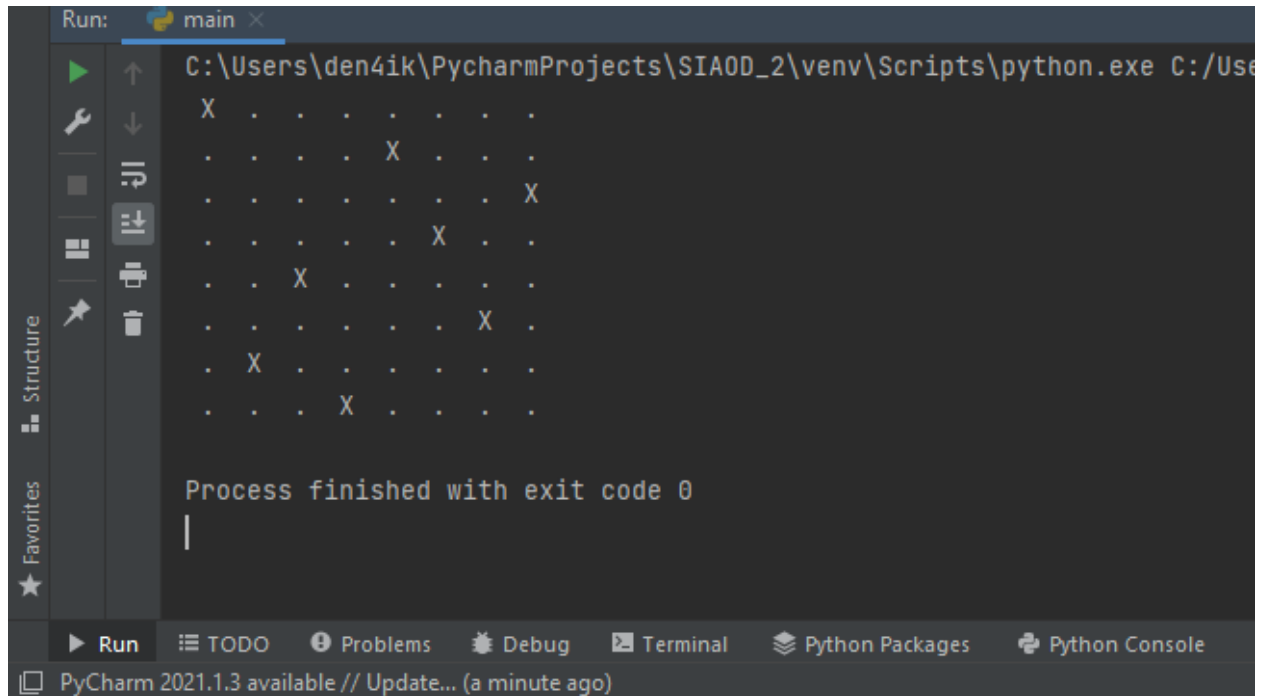
    # Метод пытающийся вставить ферзь на уровень i
    def try_queen(self, i):
        result = False
        for j in range(0,8):
            if self.field[i][j] == 0:
                self.set_queen(i, j)
                if i == 7:
                    result = True
                else:
                    result = self.try_queen(i+1)
                    if result == False:
                        self.reset_queen(i, j)
            if result == True:
                break
        return result

    # Метод устанавливающий ферзь на координаты i,j
    def set_queen(self, i, j):
        for x in range(0,8):
            self.field[x][j] += 1
            self.field[i][x] += 1
            foo = j - i + x
            if (foo >= 0) and (foo < 8):
                self.field[x][foo] += 1
            foo = j + i - x
            if (foo >= 0) and (foo < 8):
                self.field[x][foo] += 1
        self.field[i][j] = -1

    # Метод удаляющий ферзь с координат i,j
    def reset_queen(self, i, j):
        for x in range(0,8):
            self.field[x][j] -= 1
            self.field[i][x] -= 1
            foo = j - i + x
            if (foo >= 0) and (foo < 8):
                self.field[x][foo] -= 1
            foo = j + i - x
            if (foo >= 0) and (foo < 8):
                self.field[x][foo] -= 1
        self.field[i][j] = 0
```



На рисунке 15 представлен результат решения задачи с помощью методов класса TheQueenProblem.



```
Run: main x
C:\Users\den4ik\PycharmProjects\SIA0D_2\venv\Scripts\python.exe C:/Use
X . . . . .
. . . . X . .
. . . . . . X
. . . . X . .
. . X . . . .
. . . . . X .
. X . . . . .
. . . X . . .

Process finished with exit code 0
```

Рис. 15 – Результат решения задачи.

### 3. ВЫВОД

В ходе данной лабораторной работы были получены различные навыки поиска, хеширования, а также использование рекурсивных алгоритмов. Сравнили время работы наших алгоритмов с стандартными средствами Python.