

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**"МИРЭА - Российский технологический университет"**

**РТУ МИРЭА**

|  |
| --- |
| **Институт информационных технологий (ИТ)** |
| **Кафедра математического обеспечения и стандартизации информационных технологий (МОСИТ)** |

**ОТЧЁТ ПО ПРАКТИЧЕСКОМУ ЗАДАНИЮ №4**

**по дисциплине «Структуры и алгоритмы обработки данных»**

Тема: **«**Сбалансированные деревья поиска (СДП) и их применение для поиска данных в файле»

|  |  |  |  |
| --- | --- | --- | --- |
| Отчет представлен к рассмотрению:  Студентка группы ИНБО-01-20 | «06» ноября 2021 г. |  | Тульцова А.Д. |
| (подпись) | | | |
| Преподаватель | «06» ноября 2021 г. |  | Сорокин А.В. |
|  | (подпись) | | |

Москва, 2021 г.

CОДЕРЖАНИЕ

[Цель работы 3](#_Toc86419941)

[Задание 1 3](#_Toc86419942)

[Задание 2 8](#_Toc86419943)

[Задание 3 13](#_Toc86419944)

[Вывод 15](#_Toc86419945)

[Список информационных источников 16](#_Toc86419946)

# Цель работы

Получение навыков в разработке и реализации алгоритмов управления бинарным деревом поиска и сбалансированными бинарными деревьями поиска (красно-чёрными деревьями), в применении файловых потоков прямого доступа к данным файла, в применении сбалансированного дерева поиска для прямого доступа к записям файла.

# Задание 1

**Постановка задачи.**

Разработать приложение, которое использует бинарное дерево поиска (БДП) для поиска записи с ключом в файле.

**Вариант 11.**

|  |
| --- |
| **Структура элемента множества**  **(ключ – подчеркнутое поле)** |
| Железнодорожная справка:  номер поезда, пункт отправления, пункт назначения, время отправления |

**Дано:**

Класс «Бинарное дерево поиска». Методы: включение элемента в дерево, поиск ключа в дереве, удаление ключа из дерева, отображение дерева.

Класс управления файлом. Методы: создание двоичного файла записей фиксированной длины из заранее подготовленных данных в текстовом файле; поиск записи в файле с использованием БДП.

**Результат.**

Приложение, выполняющее операции: включение элемента в дерево, поиск ключа в дереве, удаление ключа из дерева, отображение дерева, создание двоичного файла записей фиксированной длины из заранее подготовленных данных в текстовом файле, поиск записи в файле с использованием БДП.

**Код приложения.**

Код БДП:

compares = 0  
  
  
class BSTree:  
 def \_\_init\_\_(self):  
 self.start = None  
  
 def add(self, key, value) -> None:  
 if self.start is None:  
 self.start = Node(key, value)  
 else:  
 self.start.add(key, value)  
  
 def \_\_str\_\_(self) -> str:  
 if self.start is not None:  
 sa = []  
 self.start.add\_to\_line(sa, 0)  
 return '\n'.join(sa)  
 else:  
 return 'Empty'  
  
 def remove(self, key) -> None:  
 if self.start is not None:  
 self.start = self.start.remove(key)  
  
 def get(self, key) -> int:  
 global compares  
 compares = 0  
 if self.start is None:  
 return -1  
 else:  
 return self.start.get(key)  
  
  
class Node:  
 def \_\_init\_\_(self, key, value=1):  
 self.key = key  
 self.value = value  
 self.left = None  
 self.right = None  
  
 def add(self, key, value=1) -> None:  
 if key > self.key:  
 if self.right is None:  
 self.right = Node(key, value)  
 else:  
 self.right.add(key, value)  
 elif key == self.key:  
 self.value += value  
 else:  
 if self.left is None:  
 self.left = Node(key, value)  
 else:  
 self.left.add(key, value)  
  
 def add\_to\_line(self, sa, depth) -> None:  
 if self.right is not None:  
 self.right.add\_to\_line(sa, depth + 1)  
 sa.append(' ' \* depth + f'{self.key}|{self.value}')  
 if self.left is not None:  
 self.left.add\_to\_line(sa, depth + 1)  
  
 def remove(self, key):  
 if key > self.key:  
 if self.right is not None:  
 self.right = self.right.remove(key)  
 return self  
 elif key < self.key:  
 if self.left is not None:  
 self.left = self.left.remove(key)  
 return self  
 else:  
 if self.left is None and self.right is None:  
 return None  
 elif self.right is None:  
 return self.left  
 elif self.left is None:  
 return self.right  
 else:  
 self.key, self.value = self.left.find\_max()  
 self.left = self.left.remove(self.key)  
 return self  
  
 def find\_max(self) -> (str, int):  
 if self.right is None:  
 return self.key, self.value  
 else:  
 return self.right.find\_max()  
  
 def get(self, key):  
 global compares  
 compares += 1  
 if self.key == key:  
 return self.value  
 if key > self.key and self.right is not None:  
 return self.right.get(key)  
 if key < self.key and self.left is not None:  
 return self.left.get(key)  
 return -1

Класс обработки файла:

from typing import Dict  
  
  
class FileHandler:  
 def \_\_init\_\_(self, file\_name: str, data: Dict[str, int]):  
 self.file\_name = file\_name  
 self.data = data  
 self.size = 0  
 self.length = 0  
 with open(self.file\_name, 'w'):  
 pass  
  
 def add(self, \*\*kwargs) -> str:  
 res = []  
 for key in self.data.keys():  
 s = str(kwargs[key]).ljust(self.data[key], ' ')  
 res.append(s)  
 res = ''.join(map(str, res))  
 with open(self.file\_name, 'ab') as f:  
 f.write(res.encode())  
 self.length = len(res.encode())  
 self.size += 1  
 return self.size - 1  
  
 def get(self, n: int) -> str:  
 pos = self.length \* n  
 with open(self.file\_name, 'rb') as f:  
 f.seek(pos)  
 res = f.read(self.length).decode()  
 return res  
  
 def remove(self, n: int) -> None:  
 pass

Класс объединения работы с файлом и деревом:

class Combining:  
 def \_\_init\_\_(self, combining, file\_handler):  
 self.combining = combining  
 self.file\_handler = file\_handler  
  
 def get(self, key) -> str:  
 from time import perf\_counter # Импорт perf\_counter из библиотеки time.  
 a = perf\_counter() # Время до получения записи.  
 n = self.combining.get(key)  
 b = perf\_counter() # Время после получения записи.  
 print(f"Потраченное время на получение записи: {b - a:0.7f} с.") # Потраченное время на получение записи.  
 if n != -1:  
 return self.file\_handler.get(n)  
 else:  
 return 'None'  
  
 def add(self, key, \*\*kwargs) -> None:  
 value = self.file\_handler.add(key=key, \*\*kwargs)  
 self.combining.add(key, value)  
  
 def remove(self, key) -> None:  
 value = self.combining.get(key)  
 self.combining.remove(key)  
 self.file\_handler.remove(value)

Код тестирования:

from BSTree import BSTree as Tree  
import BSTree  
from FileHandler import FileHandler  
from Combining import Combining  
  
  
def test():  
 # Генератор случайных чисел для заполнения файла.  
 import random # Импорт библиотеки random.  
 numbers = list(range(1103030000, 1103040000)) # Список с номерами поездов (от 1103030000 до 1103040000).  
 random.shuffle(numbers) # Перемешивание списка.  
 f = open('data.txt', 'w') # Открытие файла "data.txt" на запись.  
 i = 0 # Для первой записи в номер\_поезда будет записан первый элемент из списка numbers.  
 for x in range(10000): # Повторять 10000 раз:  
 # Запись в строку "номер\_поезда;место\_отправления;место\_прибытия;время\_отправления".  
 f.write(str(numbers[i]) + ';' + 'Place' + str(random.randint(1, 5000)) + ';' +  
 'Place' + str(random.randint(5001, 9999)) + ';' + 'Time' + str(random.randint(1, 9999)) + "\n")  
 i += 1 # Для следующей записи в номер\_поезда будет записан следующий элемент из списка numbers.  
 f.close() # Закрытие файла "data.txt".  
  
 # Создание объекта для работы с файлом с четырьмя полями.  
 fw = FileHandler('data.bin', {'key': 11, 'place1': 10, 'place2': 10, 'time1': 9})  
 tree = Tree() # Создание дерева.  
 comb = Combining(tree, fw) # Объединение работы файла и дерева.  
 fill\_comb(comb, 'data.txt') # Заполнение из файла.  
 print(comb.combining) # Вывод дерева.  
 # Получение данных по ключу.  
 print("Получение первой записи:")  
 print(comb.get(str(numbers[0]))) # Первая запись из файла.  
 print(f'Число произведённых сравнений при получении первой записи: {BSTree.compares}')  
 print() # Вывод пустой строки для визуального разделения выходных данных.  
 print("Получение последней записи:")  
 print(comb.get(str(numbers[9999]))) # Последняя запись из файла.  
 print(f'Число произведённых сравнений при получении последней записи: {BSTree.compares}')  
 print()  
 print("Получение записи, расположенной в середине файла:")  
 print(comb.get(str(numbers[4999]))) # Запись, расположенная в середине файла.  
 print(f'Число произведённых сравнений при получении записи, расположенной в середине файла: {BSTree.compares}')  
 print()  
  
def fill\_comb(comb: 'Combining', path: str):  
 with open(path, 'r') as f:  
 for line in f:  
 line = line.split(';')  
 comb.add(key=line[0], place1=line[1], place2=line[2], time1=line[3])  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 test()

**Тестирование программы.**

Было запущено тестирование, в ходе которого в файл с данными было записано 10000 элементов (Рисунок 1).

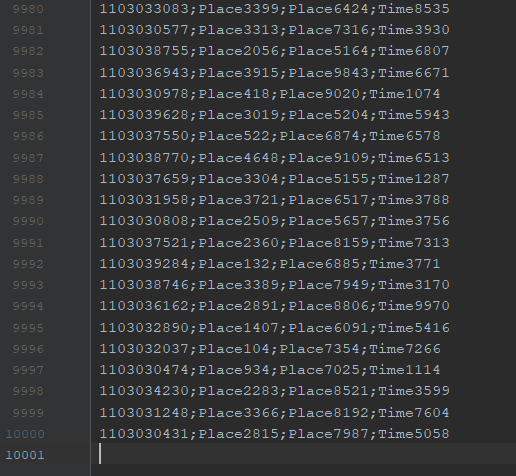


Рисунок 1 – Фрагмент файла из 10000 записей.

На основе файла было создано бинарное дерево поиска из 10000 узлов (Рисунок 2). Каждый узел дерева содержит в качестве ключа номер поезда, а в качестве информационного поля – номер строки в файле, который является ссылкой на информацию по ключу: пункт отправления, пункт назначения, время отправления.



Рисунок 2 – Фрагмент БДП из 10000 узлов.

Далее по бинарному дереву поиска была получена первая запись файла, последняя запись файла и запись, расположенная в середине файла. Для каждой полученной записи было найдено потраченное время на её получение, а также количество произведённых сравнений во время получения записи (Рисунок 3).

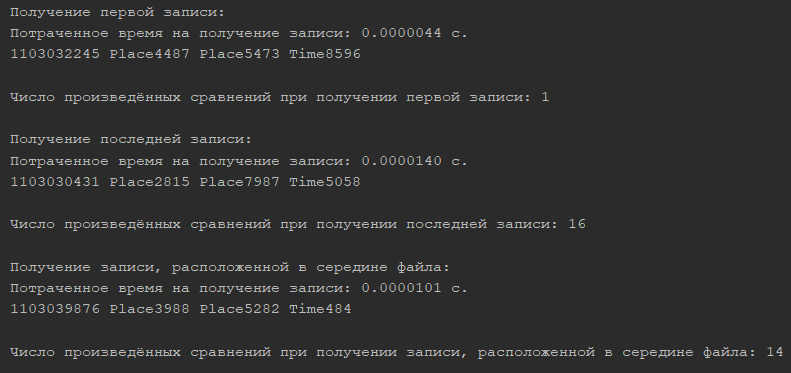


Рисунок 3 – Полученные данные, время поиска и количество сравнений.

# Задание 2

**Постановка задачи.**

Разработать приложение, которое использует сбалансированное дерево поиска, предложенное в варианте, для доступа к записям файла.

**Вариант 11.**

|  |  |
| --- | --- |
| **Сбалансированное дерево поиска (СДП)** | **Структура элемента множества (ключ – подчеркнутое поле)** |
| Красно-чёрное | Железнодорожная справка:  номер поезда, пункт отправления, пункт назначения, время отправления |

**Дано:**

Класс «Сбалансированное дерево поиска». Методы: включение элемента в дерево, поиск ключа в дереве, удаление ключа из дерева, отображение дерева.

Класс управления файлом. Методы: создание двоичного файла записей фиксированной длины из заранее подготовленных данных в текстовом файле; поиск записи в файле с использованием СДП.

**Результат.**

Приложение, выполняющее операции: включение элемента в дерево, поиск ключа в дереве, удаление ключа из дерева, отображение дерева, создание двоичного файла записей фиксированной длины из заранее подготовленных данных в текстовом файле, поиск записи в файле с использованием СДП, нахождение количества выполненных поворотов дерева.

**Код приложения.**

Код СДП (красно-чёрного дерева):

RED = 'КРАСНЫЙ'  
BLACK = 'ЧЁРНЫЙ'  
compares = 0  
turns = 0  
  
  
class RBNode:  
 def \_\_init\_\_(self, key=None, value=1, parent=None, color=RED):  
 self.key = key  
 self.value = value  
 self.color = color  
 self.parent = parent  
 self.left = None  
 self.right = None  
  
 def paint(self, color):  
 self.color = color  
  
 def isLeftChild(self):  
 # Имеет родительский узел и является левым потомком.  
 return self.parent and self is self.parent.left  
  
 def isRightChild(self):  
 # Имеет родительский узел и является правильным потомком.  
 return self.parent and self is self.parent.right  
  
 def sibling(self):  
 if self.isLeftChild(): # Если это левый дочерний элемент, то вернуться к правому дочернему элементу.  
 return self.parent.right  
 if self.isRightChild(): # Если это правый дочерний элемент, то вернуться к левому дочернему элементу.  
 return self.parent.left  
 return None # Ни левый, ни правый – отсутствие дочернего узла.  
  
 def uncle(self):  
 if self.parent is None:  
 return None  
 return self.parent.sibling()  
  
 def add(self, key=None, value=1, parent=None, color=RED) -> None:  
 if key > self.key:  
 if self.right is None:  
 self.right = RBNode(key, value, parent, color)  
 else:  
 self.right.add(key, value, parent, color)  
 elif key == self.key:  
 self.value += value  
 else:  
 if self.left is None:  
 self.left = RBNode(key, value, parent, color)  
 else:  
 self.left.add(key, value, parent, color)  
  
 def add\_to\_line(self, sa, depth) -> None:  
 if self.right is not None:  
 self.right.add\_to\_line(sa, depth + 1)  
 sa.append(' ' \* depth + f'–{self.color}–{self.key}|{self.value}')  
 if self.left is not None:  
 self.left.add\_to\_line(sa, depth + 1)  
  
 def remove(self, key):  
 if key > self.key:  
 if self.right is not None:  
 self.right = self.right.remove(key)  
 return self  
 elif key < self.key:  
 if self.left is not None:  
 self.left = self.left.remove(key)  
 return self  
 else:  
 if self.left is None and self.right is None:  
 return None  
 elif self.right is None:  
 return self.left  
 elif self.left is None:  
 return self.right  
 else:  
 self.key, self.value = self.left.find\_max()  
 self.left = self.left.remove(self.key)  
 return self  
  
 def find\_max(self) -> (str, int):  
 if self.right is None:  
 return self.key, self.value  
 else:  
 return self.right.find\_max()  
  
 def get(self, key):  
 global compares  
 compares += 1  
 if self.key == key:  
 return self.value  
 if key > self.key and self.right is not None:  
 return self.right.get(key)  
 if key < self.key and self.left is not None:  
 return self.left.get(key)  
 return -1  
  
  
class RBTree:  
 def \_\_init\_\_(self):  
 self.start = None  
 self.size = 0  
  
 def \_\_str\_\_(self) -> str:  
 if self.start is not None:  
 sa = []  
 self.start.add\_to\_line(sa, 0)  
 return '\n'.join(sa)  
 else:  
 return 'Empty'  
  
 def isRed(self, node):  
 # Текущий узел существует и красный.  
 return node and node.color == RED  
  
 def isBlack(self, node):  
 # Текущий узел не существует (внешний узел - чёрный по умолчанию) или цвет чёрный.  
 return node is None or node.color == BLACK  
  
 def predecessor(self, node):  
 if node is None:  
 return None  
 if node.left: # Самый большой узел левого поддерева - предшественник.  
 p = node.left  
 while p.right:  
 p = p.right  
 return p  
 # Нет левого поддерева => если это правое поддерево родительского - родительский является предшественником.  
 while node.parent and node is node.parent.left:  
 node = node.parent  
 return node.parent  
  
 def successor(self, node):  
 if node is None:  
 return None  
 if node.right: # Наименьший узел правого поддерева является преемником.  
 s = node.right  
 while s.left:  
 s = s.left  
 return s  
 # Правого поддерева нет => если это левое поддерево родительского - родительский является преемником.  
 while node.parent and node is node.parent.right:  
 node = node.parent  
 return node.parent  
  
 def add(self, key, value):  
 if self.start is None:  
 self.start = RBNode(key, value)  
 self.size += 1  
 self.\_insert(self.start)  
 return  
 parent = self.start  
 node = self.start  
 flag = 0  
 while node:  
 parent = node  
 if key > node.key:  
 node = node.right  
 flag = 0  
 elif key < node.key:  
 node = node.left  
 flag = 1  
 else:  
 node.key = key  
 return  
 new = RBNode(key=key, value=value, parent=parent)  
 if flag == 0:  
 parent.right = new  
 else:  
 parent.left = new  
 self.size += 1  
 self.\_insert(new)  
  
 def \_insert(self, node):  
 parent = node.parent  
 # Добавить корневой узел или переполнить корневой узел.  
 if parent is None:  
 node.paint(BLACK)  
 return  
 # Чёрный родительский узел => вставить напрямую без обработки.  
 if self.isBlack(parent):  
 return  
 # Случай, когда родительский узел - красный.  
 grand = parent.parent  
 grand.paint(RED) # Дед всегда будет красным, независимо от цвета дяди.  
 uncle = parent.sibling()  
 # Переполнение узла.  
 if self.isRed(uncle): # Если красный дядя:  
 parent.paint(BLACK)  
 uncle.paint(BLACK)  
 self.\_insert(grand) # Узел переполняется, дед - как вновь вставленный узел.  
 return  
 if parent.isLeftChild():  
 if node.isLeftChild():  
 parent.paint(BLACK)  
 else:  
 node.paint(BLACK)  
 self.LeftRotate(parent)  
 self.RightRotate(grand)  
 else: # Если чёрный дядя:  
 if node.isLeftChild():  
 node.paint(BLACK)  
 self.RightRotate(parent)  
 else:  
 parent.paint(BLACK)  
 self.LeftRotate(grand)  
  
 def get(self, key) -> int:  
 global compares  
 compares = 0  
 if self.start is None:  
 return -1  
 else:  
 return self.start.get(key)  
  
 def \_search(self, subtree, key):  
 if subtree is None:  
 return None  
 elif key < subtree.key:  
 return self.\_search(subtree.left, key)  
 elif key > subtree.key:  
 return self.\_search(subtree.right, key)  
 else:  
 return subtree.key  
  
 def remove(self, key):  
 # Фактически, удаленный узел - это конечный узел (т.е. последний уровень B-дерева).  
 node = self.\_search(self.start, key)  
 if node is None:  
 return  
 self.size -= 1  
 if node.left and node.right: # Узел со степенью 2: нахождение его преемника.  
 s = self.successor(node)  
 node.key = s.key  
 node = s  
 replacement = node.left if node.left else node.right # Элемент, используемый для замены.  
 if replacement: # Узел степени 1.  
 replacement.parent = node.parent  
 if node.parent is None: # Корневой узел.  
 self.start = replacement  
 elif node.parent.left is node: # Левое поддерево родительского узла.  
 node.parent.left = replacement  
 else: # Правое поддерево родительского узла.  
 node.parent.right = replacement  
 self.\_remove(replacement)  
 node.left = node.right = node.parent = None  
 elif node.parent is None: # Корневой узел.  
 self.start = None  
 self.\_remove(node)  
 else: # Листовой узел.  
 if node is node.parent.left:  
 node.parent.left = None  
 else:  
 node.parent.right = None  
 self.\_remove(node)  
 node.parent = None  
  
 def \_remove(self, node):  
 if self.isRed(node): # Если заменяемый узел - красный, то покрасить его в чёрный.  
 node.paint(BLACK)  
 return  
 parent = node.parent  
 if parent is None:  
 return  
 # Альтернативный узел - чёрный.  
 left = node.isLeftChild() or parent.left is None  
 sibling = parent.right if left else parent.left  
 if left: # Родственный узел - справа.  
 if self.isRed(sibling): # Если красный брат:  
 sibling.paint(BLACK)  
 parent.paint(RED)  
 self.LeftRotate(parent)  
 sibling = parent.right  
 if self.isBlack(sibling.left) and self.isBlack(sibling.right):  
 parentBlack = self.isBlack(parent)  
 parent.paint(BLACK)  
 sibling.paint(RED)  
 if parentBlack: # Если родительский элемент также чёрный, то это вызовет потерю родительского значения.  
 if parent.isLeftChild():  
 self.\_remove(parent) # Рассматривать родителя как удалённый узел.  
 else: # У родственного узла есть хотя бы один красный узел.  
 if sibling.right.isBlack():  
 self.RightRotate(sibling)  
 sibling = parent.right  
 sibling.color = parent.color  
 parent.paint(BLACK)  
 sibling.right.paint(BLACK)  
 self.LeftRotate(parent)  
 else: # Родственный узел слева, полностью симметричен верхней стороне.  
 if self.isRed(sibling):  
 sibling.paint(BLACK)  
 parent.paint(RED)  
 self.RightRotate(parent)  
 sibling = parent.left  
 if self.isBlack(sibling.left) and self.isBlack(sibling.right):  
 parentBlack = parent.isBlack()  
 parent.paint(BLACK)  
 sibling.paint(RED)  
 if parentBlack:  
 if parent.isLeftChild():  
 self.\_remove(parent)  
 else:  
 if self.isBlack(sibling.left):  
 self.LeftRotate(sibling)  
 sibling = parent.left  
 sibling.color = parent.color  
 parent.paint(BLACK)  
 sibling.left.color = BLACK  
 self.RightRotate(parent)  
  
 def LeftRotate(self, grand):  
 global turns  
 turns += 1  
 parent = grand.right  
 child = parent.left  
 grand.right = child  
 parent.left = grand  
 self.\_rotate(grand, parent, child)  
  
 def RightRotate(self, grand):  
 global turns  
 turns += 1  
 parent = grand.left  
 child = parent.right  
 grand.left = child  
 parent.right = grand  
 self.\_rotate(grand, parent, child)  
  
 def \_rotate(self, grand, parent, child):  
 # Сохранение соответствующего отношения наведения после поворота.  
 if grand.isLeftChild():  
 grand.parent.left = parent  
 elif grand.isRightChild():  
 grand.parent.right = parent  
 else:  
 self.start = parent  
 if child: # Указание родительского элемента ребёнка на деда.  
 child.parent = grand  
 parent.parent = grand.parent # Направление родительского элемента на главного родителя.  
 grand.parent = parent

Код тестирования:

from RBTree import RBTree as Tree  
import RBTree  
from FileHandler import FileHandler  
from Combining import Combining

def test():  
 # Генератор случайных чисел для заполнения файла.  
 import random # Импорт библиотеки random.  
 numbers = list(range(1103030000, 1103040000)) # Список с номерами поездов (от 1103030000 до 1103040000).  
 random.shuffle(numbers) # Перемешивание списка.  
 f = open('data.txt', 'w') # Открытие файла "data.txt" на запись.  
 i = 0 # Для первой записи в номер\_поезда будет записан первый элемент из списка numbers.  
 for x in range(10000): # Повторять 10000 раз:  
 # Запись в строку "номер\_поезда;место\_отправления;место\_прибытия;время\_отправления".  
 f.write(str(numbers[i]) + ';' + 'Place' + str(random.randint(1, 5000)) + ';' +  
 'Place' + str(random.randint(5001, 9999)) + ';' + 'Time' + str(random.randint(1, 9999)) + "\n")  
 i += 1 # Для следующей записи в номер\_поезда будет записан следующий элемент из списка numbers.  
 f.close() # Закрытие файла "data.txt".  
 # Создание объекта для работы с файлом с четырьмя полями.  
 fw = FileHandler('data.bin', {'key': 11, 'place1': 10, 'place2': 10, 'time1': 9})  
 tree = Tree() # Создание дерева.  
 comb = Combining(tree, fw) # Объединение работы файла и дерева.  
 fill\_comb(comb, 'data.txt') # Заполнение из файла.  
 print(comb.combining) # Вывод дерева.  
 # Получение данных по ключу.  
 print("Получение первой записи:")  
 print(comb.get(str(numbers[0]))) # Первая запись из файла.  
 print(f'Число произведённых сравнений при получении первой записи: {RBTree.compares}')  
 print() # Вывод пустой строки для визуального разделения выходных данных.  
 print("Получение последней записи:")  
 print(comb.get(str(numbers[9999]))) # Последняя запись из файла.  
 print(f'Число произведённых сравнений при получении последней записи: {RBTree.compares}')  
 print()  
 print("Получение записи, расположенной в середине файла:")  
 print(comb.get(str(numbers[4999]))) # Запись, расположенная в середине файла.  
 print(f'Число произведённых сравнений при получении записи, расположенной в середине файла: {RBTree.compares}')  
 print()  
 # Получение количества произведённых поворотов.  
 print(f'Общее число произведённых поворотов: {RBTree.turns}')  
  
def fill\_comb(comb: 'Combining', path: str):  
 with open(path, 'r') as f:  
 for line in f:  
 line = line.split(';')  
 comb.add(key=line[0], place1=line[1], place2=line[2], time1=line[3])  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 test()

**Тестирование программы.**

Было запущено тестирование, в ходе которого в файл с данными было записано 10000 элементов (Рисунок 4).

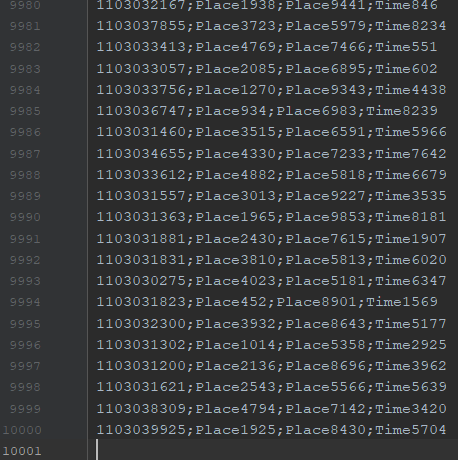


Рисунок 4 – Фрагмент файла из 10000 записей.

На основе файла было создано СДП: красно-чёрное дерево (Рисунок 5). Каждый узел дерева содержит информацию о своём цвете (узел – красный или чёрный), ключ – номер поезда, а также информационное поле – номер строки в файле, который является ссылкой на информацию по ключу: пункт отправления, пункт назначения, время отправления.

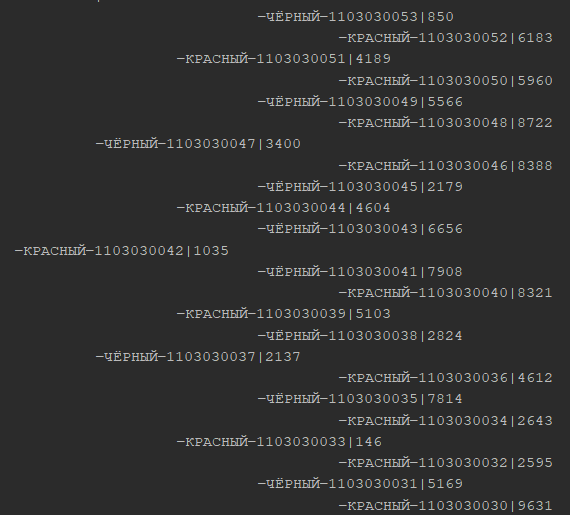


Рисунок 5 – Фрагмент СДП, построенного по файлу из 10000 записей.

Далее по дереву была получена первая запись файла, последняя запись файла и запись, расположенная в середине файла. Для каждой полученной записи было найдено потраченное время на её получение, а также количество произведённых сравнений во время получения записи (Рисунок 6).

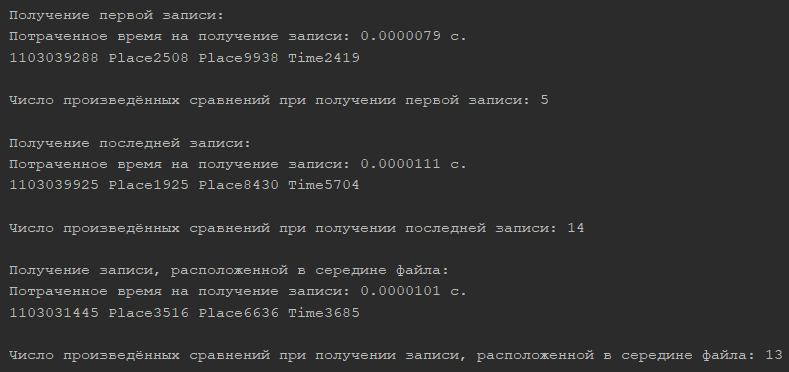


Рисунок 6 – Полученные данные, время поиска и количество сравнений.

После этого было получено общее количество произведённых поворотов и выведено на экран (Рисунок 7).



Рисунок 7 – Количество произведённых поворотов.

# Задание 3

**Постановка задачи.**

Выполнить анализ алгоритма поиска записи с заданным ключом при применении структур данных:

* хеш-таблица;
* бинарное дерево поиска;
* СДП (красно-чёрное дерево).

**Код приложения.**

Код тестирования:

from BSTree import BSTree as BST  
import BSTree  
from RBTree import RBTree as RBT  
import RBTree  
from HashTable import HashTable as HT  
import HashTable  
from FileHandler import FileHandler  
from Combining import Combining  
  
  
def test():  
 # Заполнение файла 1000 записями.  
 print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")  
 print("Запущено тестирование программы для случая, когда в файле 1000 записей.")  
  
 # Генератор случайных чисел для заполнения файла.  
 import random # Импорт библиотеки random.  
 numbers = list(range(110303000, 110304000)) # Список с номерами поездов (от 110303000 до 110304000).  
 random.shuffle(numbers) # Перемешивание списка.  
 f = open('data.txt', 'w') # Открытие файла "data.txt" на запись.  
 i = 0 # Для первой записи в номер\_поезда будет записан первый элемент из списка numbers.  
 for x in range(1000): # Повторять 1000 раз:  
 # Запись в строку "номер\_поезда;место\_отправления;место\_прибытия;время\_отправления".  
 f.write(str(numbers[i]) + ';' + 'Place' + str(random.randint(1, 500)) + ';' +  
 'Place' + str(random.randint(501, 999)) + ';' + 'Time' + str(random.randint(1, 999)) + "\n")  
 i += 1 # Для следующей записи в номер\_поезда будет записан следующий элемент из списка numbers.  
 f.close() # Закрытие файла "data.txt".  
  
 # Создание объекта для работы с файлом с четырьмя полями.  
 fw = FileHandler('data.bin', {'key': 11, 'place1': 10, 'place2': 10, 'time1': 9})  
 bs = BST() # Создание БДП.  
 rb = RBT() # Создание СДП.  
 ht = HT() # Создание хеш-таблицы.  
 bs\_comb = Combining(bs, fw) # Объединение работы файла и БДП.  
 rb\_comb = Combining(rb, fw) # Объединение работы файла и СДП.  
 ht\_comb = Combining(ht, fw) # Объединение работы файла и хеш-таблицы.  
 fill\_comb(bs\_comb, 'data.txt') # Заполнение БДП из файла.  
 fill\_comb(rb\_comb, 'data.txt') # Заполнение СДП из файла.  
 fill\_comb(ht\_comb, 'data.txt') # Заполнение хеш-таблицы из файла.  
  
 # Получение первой записи файла по ключу.  
 print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_(000)")  
 print("Получение первой записи с помощью бинарного дерева поиска:")  
 print()  
 print(bs\_comb.get(str(numbers[0])))  
 print(f'Число произведённых сравнений при получении первой записи: {BSTree.compares}')  
 print("-----------------------------------------------------------------------------------------")  
 print("Получение первой записи с помощью СДП (красно-чёрного дерева):")  
 print()  
 print(rb\_comb.get(str(numbers[0])))  
 print(f'Число произведённых сравнений при получении первой записи: {RBTree.compares}')  
 print("-----------------------------------------------------------------------------------------")  
 print("Получение первой записи с помощью хеш-таблицы:")  
 print()  
 print(ht\_comb.get(str(numbers[0])))  
 print(f'Число произведённых сравнений при получении первой записи: {HashTable.compares}')  
  
 # Получение последней записи файла по ключу.  
 print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_(999)")  
 print("Получение последней записи с помощью бинарного дерева поиска:")  
 print()  
 print(bs\_comb.get(str(numbers[999])))  
 print(f'Число произведённых сравнений при получении последней записи: {BSTree.compares}')  
 print("-----------------------------------------------------------------------------------------")  
 print("Получение последней записи с помощью СДП (красно-чёрного дерева):")  
 print()  
 print(rb\_comb.get(str(numbers[999])))  
 print(f'Число произведённых сравнений при получении последней записи: {RBTree.compares}')  
 print("-----------------------------------------------------------------------------------------")  
 print("Получение последней записи с помощью хеш-таблицы:")  
 print()  
 print(ht\_comb.get(str(numbers[999])))  
 print(f'Число произведённых сравнений при получении последней записи: {HashTable.compares}')  
  
 # Получение по ключу записи, расположенной в середине файла.  
 print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_(499)")  
 print("Получение записи, расположенной в середине файла, с помощью бинарного дерева поиска:")  
 print()  
 print(bs\_comb.get(str(numbers[499])))  
 print(f'Число произведённых сравнений при получении записи, расположенной в середине файла: {BSTree.compares}')  
 print("-----------------------------------------------------------------------------------------")  
 print("Получение записи, расположенной в середине файла, с помощью СДП (красно-чёрного дерева):")  
 print()  
 print(rb\_comb.get(str(numbers[499])))  
 print(f'Число произведённых сравнений при получении записи, расположенной в середине файла: {RBTree.compares}')  
 print("-----------------------------------------------------------------------------------------")  
 print("Получение записи, расположенной в середине файла, с помощью хеш-таблицы:")  
 print()  
 print(ht\_comb.get(str(numbers[499])))  
 print(f'Число произведённых сравнений при получении записи, расположенной в середине файла: {HashTable.compares}')  
  
 # Заполнение файла 10000 записями.  
 print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")  
 print("Запущено тестирование программы для случая, когда в файле 10000 записей.")  
 # Генератор случайных чисел для заполнения файла.  
 import random # Импорт библиотеки random.  
 numbers = list(range(1103030000, 1103040000)) # Список с номерами поездов (от 1103030000 до 1103040000).  
 random.shuffle(numbers) # Перемешивание списка.  
 f = open('data.txt', 'w') # Открытие файла "data.txt" на запись.  
 i = 0 # Для первой записи в номер\_поезда будет записан первый элемент из списка numbers.  
 for x in range(10000): # Повторять 10000 раз:  
 # Запись в строку "номер\_поезда;место\_отправления;место\_прибытия;время\_отправления".  
 f.write(str(numbers[i]) + ';' + 'Place' + str(random.randint(1, 5000)) + ';' +  
 'Place' + str(random.randint(5001, 9999)) + ';' + 'Time' + str(random.randint(1, 9999)) + "\n")  
 i += 1 # Для следующей записи в номер\_поезда будет записан следующий элемент из списка numbers.  
 f.close() # Закрытие файла "data.txt".  
  
 # Создание объекта для работы с файлом с четырьмя полями.  
 fw = FileHandler('data.bin', {'key': 11, 'place1': 10, 'place2': 10, 'time1': 9})  
 bs = BST() # Создание БДП.  
 rb = RBT() # Создание СДП.  
 ht = HT() # Создание хеш-таблицы.  
 bs\_comb = Combining(bs, fw) # Объединение работы файла и БДП.  
 rb\_comb = Combining(rb, fw) # Объединение работы файла и СДП.  
 ht\_comb = Combining(ht, fw) # Объединение работы файла и хеш-таблицы.  
 fill\_comb(bs\_comb, 'data.txt') # Заполнение БДП из файла.  
 fill\_comb(rb\_comb, 'data.txt') # Заполнение СДП из файла.  
 fill\_comb(ht\_comb, 'data.txt') # Заполнение хеш-таблицы из файла.  
  
 # Получение первой записи файла по ключу.  
 print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_(0000)")  
 print("Получение первой записи с помощью бинарного дерева поиска:")  
 print()  
 print(bs\_comb.get(str(numbers[0])))  
 print(f'Число произведённых сравнений при получении первой записи: {BSTree.compares}')  
 print("-----------------------------------------------------------------------------------------")  
 print("Получение первой записи с помощью СДП (красно-чёрного дерева):")  
 print()  
 print(rb\_comb.get(str(numbers[0])))  
 print(f'Число произведённых сравнений при получении первой записи: {RBTree.compares}')  
 print("-----------------------------------------------------------------------------------------")  
 print("Получение первой записи с помощью хеш-таблицы:")  
 print()  
 print(ht\_comb.get(str(numbers[0])))  
 print(f'Число произведённых сравнений при получении первой записи: {HashTable.compares}')  
  
 # Получение последней записи файла по ключу.  
 print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_(9999)")  
 print("Получение последней записи с помощью бинарного дерева поиска:")  
 print()  
 print(bs\_comb.get(str(numbers[9999])))  
 print(f'Число произведённых сравнений при получении последней записи: {BSTree.compares}')  
 print("-----------------------------------------------------------------------------------------")  
 print("Получение последней записи с помощью СДП (красно-чёрного дерева):")  
 print()  
 print(rb\_comb.get(str(numbers[9999])))  
 print(f'Число произведённых сравнений при получении последней записи: {RBTree.compares}')  
 print("-----------------------------------------------------------------------------------------")  
 print("Получение последней записи с помощью хеш-таблицы:")  
 print()  
 print(ht\_comb.get(str(numbers[9999])))  
 print(f'Число произведённых сравнений при получении последней записи: {HashTable.compares}')  
  
 # Получение по ключу записи, расположенной в середине файла.  
 print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_(4999)")  
 print("Получение записи, расположенной в середине файла, с помощью бинарного дерева поиска:")  
 print()  
 print(bs\_comb.get(str(numbers[4999])))  
 print(f'Число произведённых сравнений при получении записи, расположенной в середине файла: {BSTree.compares}')  
 print("-----------------------------------------------------------------------------------------")  
 print("Получение записи, расположенной в середине файла, с помощью СДП (красно-чёрного дерева):")  
 print()  
 print(rb\_comb.get(str(numbers[4999])))  
 print(f'Число произведённых сравнений при получении записи, расположенной в середине файла: {RBTree.compares}')  
 print("-----------------------------------------------------------------------------------------")  
 print("Получение записи, расположенной в середине файла, с помощью хеш-таблицы:")  
 print()  
 print(ht\_comb.get(str(numbers[4999])))  
 print(f'Число произведённых сравнений при получении записи, расположенной в середине файла: {HashTable.compares}')  
  
  
def fill\_comb(comb: 'Combining', path: str):  
 with open(path, 'r') as f:  
 for line in f:  
 line = line.split(';')  
 comb.add(key=line[0], place1=line[1], place2=line[2], time1=line[3])  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 test()

**Тестирование программы.**

Было запущено тестирование, в ходе которого в файл с данными было записано 1000 элементов (Рисунок 8).

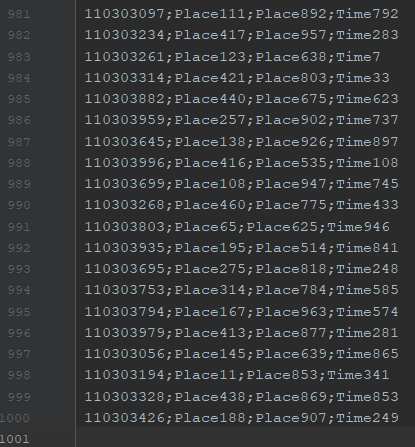


Рисунок 8 – Фрагмент файла из 1000 записей.

На основе файла были созданы следующие структуры данных: бинарное дерево поиска, сбалансированное дерево поиска: красно-чёрное дерево и хеш-таблица с цепным хешированием.

Далее была получена первая запись файла: сначала с помощью БДП, затем с помощью СДП и после этого с помощью хеш-таблицы. Для каждого алгоритма получения записи было найдено потраченное время на её получение, а также количество произведённых сравнений во время получения записи (Рисунок 9).

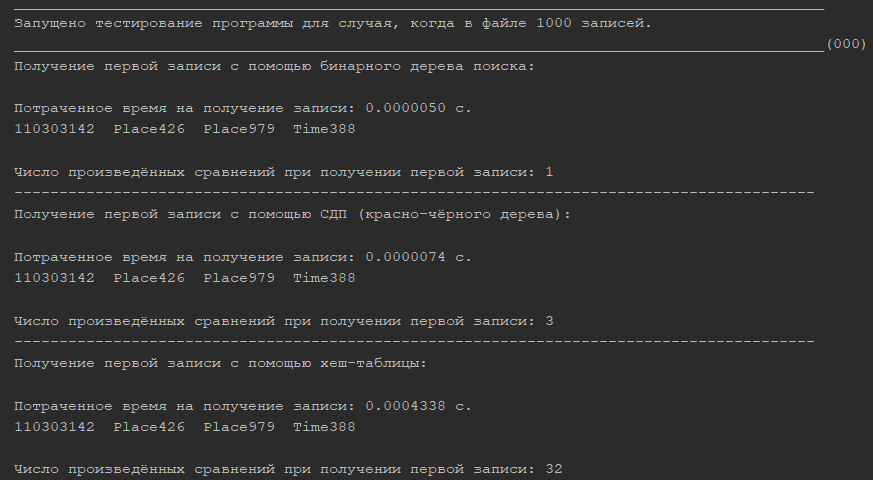


Рисунок 9 – Полученные данные, время поиска и количество сравнений для первой записи.

Подобным образом были получены последняя запись и запись, расположенная в середине файла, а также соответствующая информация по работе алгоритмов поиска. (Рисунок 10, 11).

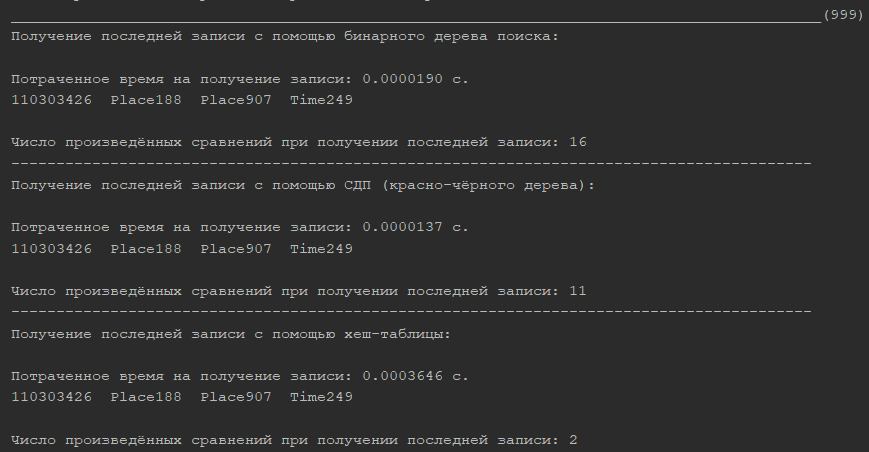


Рисунок 10 – Полученные данные, время поиска и количество сравнений для последней записи.

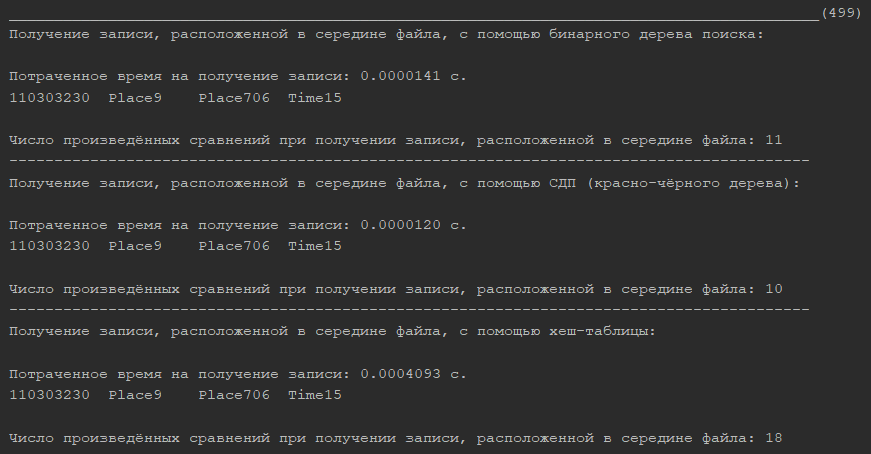


Рисунок 11 – Полученные данные, время поиска и количество сравнений для записи, расположенной в середине файла.

После этого был обновлён файл с данными: в него было записано теперь 10000 элементов (Рисунок 12).

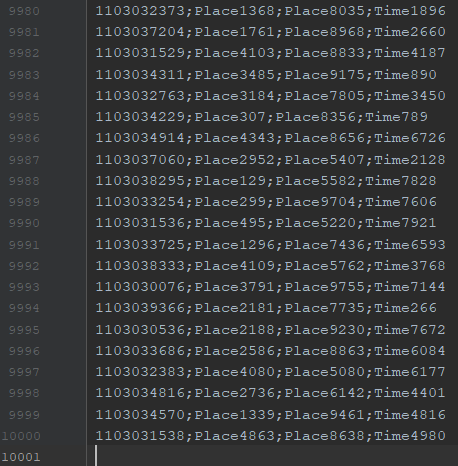


Рисунок 12 – Фрагмент файла из 10000 записей.

На основе файла были созданы структуры данных: бинарное дерево поиска, сбалансированное дерево поиска: красно-чёрное дерево и хеш-таблица с цепным хешированием.

Далее была получена первая запись файла: сначала с помощью БДП, затем с помощью СДП и после этого с помощью хеш-таблицы. Для каждого алгоритма получения записи было найдено потраченное время на её получение, а также количество произведённых сравнений во время получения записи (Рисунок 13).

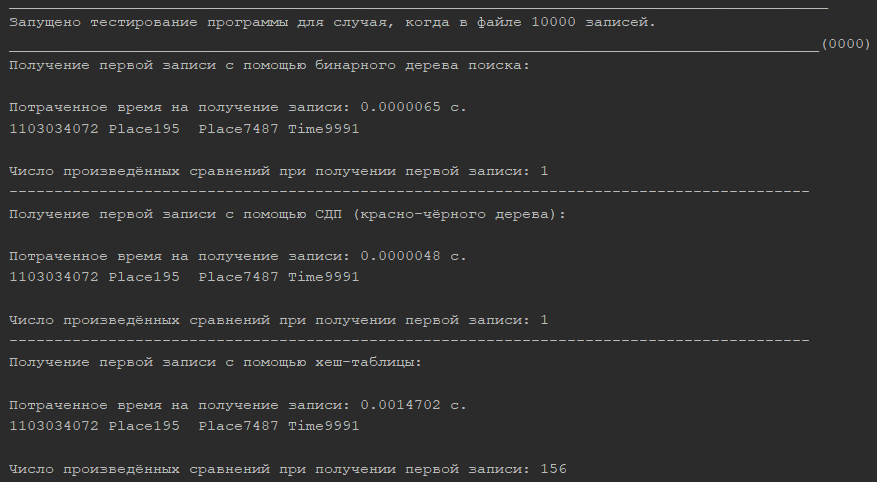


Рисунок 13 – Полученные данные, время поиска и количество сравнений для первой записи.

Подобным образом были получены последняя запись и запись, расположенная в середине файла, а также соответствующая информация по работе алгоритмов поиска. (Рисунок 14, 15).

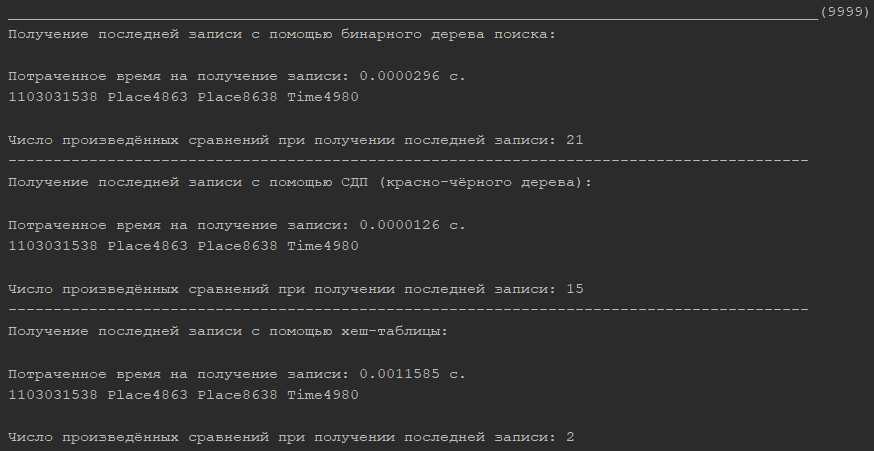


Рисунок 14 – Полученные данные, время поиска и количество сравнений для последней записи.

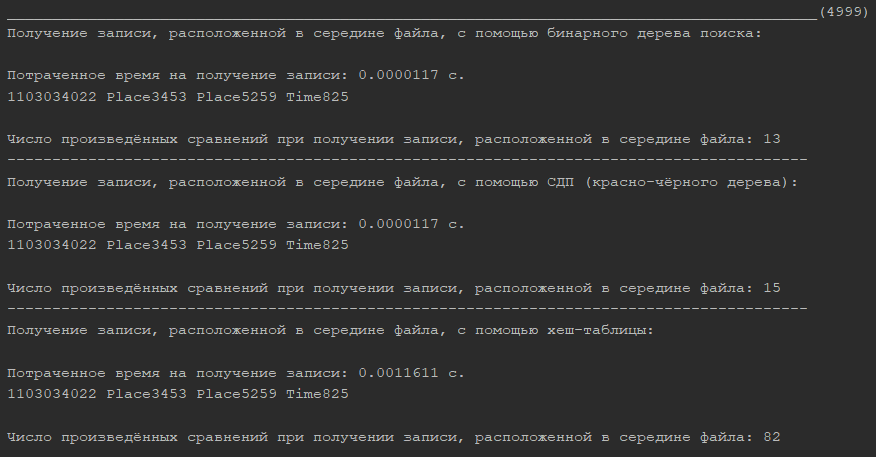


Рисунок 15 – Полученные данные, время поиска и количество сравнений для записи, расположенной в середине файла.

**Анализ результатов.**

Получение первой записи из файла:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов, загруженных в структуру в момент выполнения поиска | Вид поисковой структуры | Емкостная сложность: объем памяти для структуры | Количество выполненных сравнений | Время на поиск ключа в структуре, мс |
| 1000 | БДП | n | 1 | 0.0050 |
| Красно-чёрное | n | 3 | 0.0074 |
| Хэш-таблица | n | 32 | 0.4338 |
| 10000 | БДП | n | 1 | 0.0065 |
| Красно-чёрное | n | 1 | 0.0048 |
| Хэш-таблица | n | 156 | 1.4702 |

Получение последней записи из файла:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов, загруженных в структуру в момент выполнения поиска | Вид поисковой структуры | Емкостная сложность: объем памяти для структуры | Количество выполненных сравнений | Время на поиск ключа в структуре, мс |
| 1000 | БДП | n | 16 | 0.0190 |
| Красно-чёрное | n | 11 | 0.0137 |
| Хэш-таблица | n | 2 | 0.3646 |
| 10000 | БДП | n | 21 | 0.0296 |
| Красно-чёрное | n | 15 | 0.0126 |
| Хэш-таблица | n | 2 | 1.1585 |

Получение записи, расположенной в середине файла:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество элементов, загруженных в структуру в момент выполнения поиска | Вид поисковой структуры | Емкостная сложность: объем памяти для структуры | Количество выполненных сравнений | Время на поиск ключа в структуре, мс |
| 1000 | БДП | n | 11 | 0.0141 |
| Красно-чёрное | n | 10 | 0.0120 |
| Хэш-таблица | n | 18 | 0.4093 |
| 10000 | БДП | n | 13 | 0.0117 |
| Красно-чёрное | n | 15 | 0.0117 |
| Хэш-таблица | n | 82 | 1.1611 |

По результатам видно, что почти во всех случаях наиболее быстрое получение данных осуществлялось с помощью сбалансированного дерева поиска: красно-чёрного дерева. В общем случае, чуть менее быстрым был поиск данных посредством бинарного дерева поиска. Получение данных с помощью хеш-таблицы происходило намного медленнее во всех случаях.

# Вывод

В ходе работы были приобретены практические навыки по использованию и построению бинарного дерева поиска и сбалансированного дерева поиска.

# Список информационных источников

1. Лекции по дисциплине «Структуры и алгоритмы обработки данных» /

Л. А. Скворцова, МИРЭА – Российский технологический университет, 2021.