

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:  Студент группы ИНБО-01-20 | «07» ноября 2021 г. |  | Салов В.Д. |
| (подпись) | | | |
| Преподаватель | «07» ноября 2021 г. |  | Сорокин А.В. |
|  | (подпись) | | |

Москва, 2021 г.

CОДЕРЖАНИЕ

[Цель работы 3](#_Toc87154033)

[Задание 1 3](#_Toc87154034)

[Задание 2 10](#_Toc87154035)

[Задание 3 20](#_Toc87154036)

[Вывод 30](#_Toc87154037)

[Список информационных источников 31](#_Toc87154038)

# 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 9.**

|  |
| --- |
| **Структура элемента множества**  **(ключ – подчеркнутое поле)** |
| Страховой полис:  номер, компания, фамилия владельца |

**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 BST:  
 def \_\_init\_\_(self):  
 self.start = None  
  
 def add(self, key, value) -> None:  
 if self.start is None:  
 self.start = NodeOfBST(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 NodeOfBST:  
 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 = NodeOfBST(key, value)  
 else:  
 self.right.add(key, value)  
 elif key == self.key:  
 self.value += value  
 else:  
 if self.left is None:  
 self.left = NodeOfBST(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}>({depth})')  
 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  
 a = perf\_counter()  
 n = self.combining.get(key)  
 b = perf\_counter()  
 print(f"Затраченное время на получение записи: {b - a:0.7f} с.")  
 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 BS\_Tree import BST  
import BS\_Tree  
from F\_Handler import FileHandler  
from FS\_Combining import Combining  
  
  
def test():  
 # Заполнение файла 10000 записями.  
 import random  
 numbers = list(range(2401030000, 2401040000)) # Список с номерами страховых полисов.  
 random.shuffle(numbers) # Перемешивание списка.  
 f = open('Data\_Records.txt', 'w') # Открытие файла "Data\_Records.txt" на запись.  
 i = 0 # Для первой записи в номер\_полиса будет записан первый элемент из списка numbers.  
 for x in range(10000): # Повторять 10000 раз:  
 # Запись в строку: "номер\_полиса;компания;фамилия\_владельца".  
 f.write(str(numbers[i]) + ';' + 'Company' + str(random.randint(1, 9999)) + ';'  
 + 'Soname' + str(random.randint(1, 9999)) + "\n")  
 i += 1 # Для следующей записи в номер\_полиса будет записан следующий элемент из списка numbers.  
 f.close() # Закрытие файла "Data\_Records.txt".  
 # Создание объекта для работы с файлом с тремя полями.  
 fw = FileHandler('Data\_Records.bin', {'key': 11, 'company': 12, 'soname': 11})  
 tree = BST() # Создание дерева.  
 comb = Combining(tree, fw) # Объединение работы файла и дерева.  
 fill\_comb(comb, 'Data\_Records.txt') # Заполнение из файла.  
 print(comb.combining) # Вывод дерева.  
 print()  
 # Получение данных по ключу.  
 print(f'Первая запись из файла: {comb.get(str(numbers[0]))}')  
 print(f'Количество произведённых сравнений при получении первой записи: {BS\_Tree.compares}')  
 print()  
 print(f'Последняя запись из файла: {comb.get(str(numbers[9999]))}')  
 print(f'Количество произведённых сравнений при получении последней записи: {BS\_Tree.compares}')  
 print()  
 print(f'Запись в середине файла: {comb.get(str(numbers[4999]))}')  
 print(f'Количество произведённых сравнений при получении записи в середине файла: {BS\_Tree.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], company=line[1], soname=line[2])  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 test()

**Тестирование программы.**

После запуска тестирования в файл с данными было записано 10000 элементов (Рисунок 1).

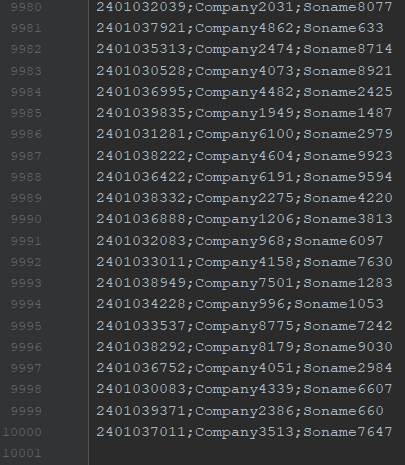


Рисунок 1 – Фрагмент файла из 10000 записей.

После этого было создано бинарное дерево поиска из 10000 узлов (Рисунок 2). Каждый узел дерева содержит в качестве ключа номер страхового полиса, а в качестве информационного поля – номер строки в файле, который является ссылкой на информацию по ключу: компания, фамилия владельца. Также для удобства при выводе каждого узла справа от него отображается (в скобках) его уровень глубины в дереве.

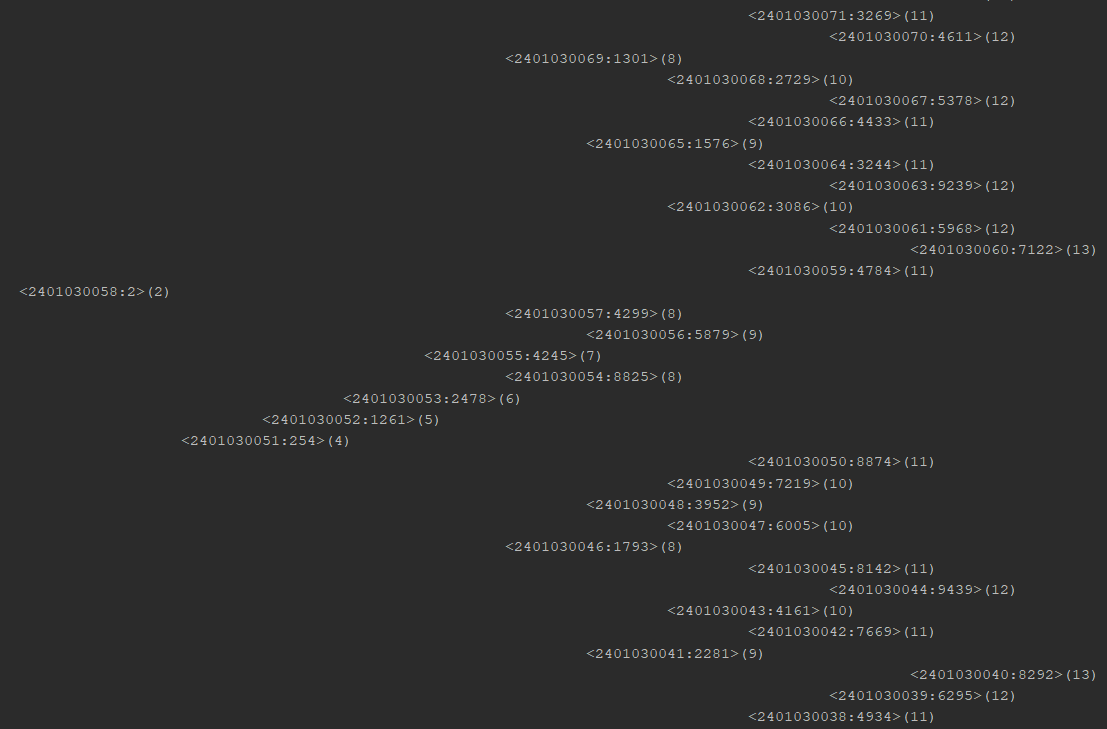


Рисунок 2 – Фрагмент БДП из 10000 узлов.

Далее по бинарному дереву поиска были получены следующие записи: первая запись файла, последняя запись файла и запись в середине файла. Для каждой полученной записи было найдено затраченное время на её получение, а также выведено количество произведённых сравнений во время получения записи (Рисунок 3).

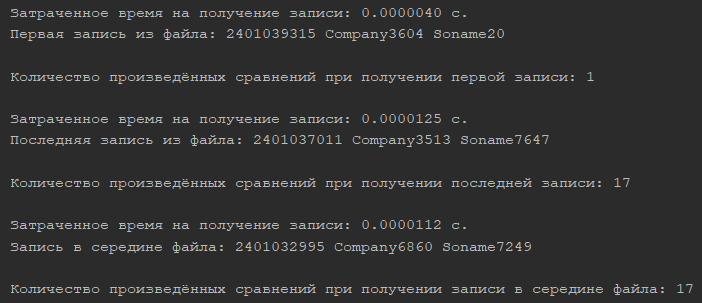


Рисунок 3 – Полученные данные, время поиска и количество сравнений.

# 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 | Страховой полис:  номер, компания, фамилия владельца |

**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 = 'R'  
BLACK = 'B'  
compares = 0  
turns = 0  
  
  
class RBT:  
 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):  
 return node and node.color == RED  
  
 def isBlack(self, node):  
 return node is None or node.color == BLACK  
  
 def predecessor(self, node):  
 if node is None:  
 return None  
 if node.left:  
 p = node.left  
 while p.right:  
 p = p.right  
 return p  
 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:  
 s = node.right  
 while s.left:  
 s = s.left  
 return s  
 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 = NodeOfRBT(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 = NodeOfRBT(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  
 if parent is None:  
 node.paint(BLACK)  
 return  
 if self.isBlack(parent):  
 return  
 grand = parent.parent  
 grand.paint(RED)  
 uncle = parent.sibling()  
 if self.isRed(uncle):  
 parent.paint(BLACK)  
 uncle.paint(BLACK)  
 self.\_insert(grand)  
 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):  
 node = self.\_search(self.start, key)  
 if node is None:  
 return  
 self.size -= 1  
 if node.left and node.right:  
 s = self.successor(node)  
 node.key = s.key  
 node = s  
 replacement = node.left if node.left else node.right  
 if replacement:  
 replacement.parent = node.parent  
 if node.parent is None:  
 self.start = replacement  
 elif node.parent.left is node:  
 node.parent.left = replacement  
 else:  
 node.parent.right = replacement  
 self.\_remove(replacement)  
 node.left = node.right = node.parent = None  
 elif node.parent is None:  
 self.start = None  
 self.\_remove(node)  
 else:  
 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):  
 node.paint(BLACK)  
 return  
 parent = node.parent  
 if parent is None:  
 return  
 left = node.isLeftChild() or parent.left is None  
 sibling = parent.right if left else parent.left  
 if left:  
 if self.isRed(sibling):  
 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:  
 if parent.isLeftChild():  
 self.\_remove(parent)  
 else:  
 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:  
 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):  
 if grand.isLeftChild():  
 grand.parent.left = parent  
 elif grand.isRightChild():  
 grand.parent.right = parent  
 else:  
 self.start = parent  
 if child:  
 child.parent = grand  
 parent.parent = grand.parent  
 grand.parent = parent  
  
  
class NodeOfRBT:  
 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):  
 return self.parent and self is self.parent.left  
  
 def isRightChild(self):  
 return self.parent and self is self.parent.right  
  
 def sibling(self):  
 if self.isLeftChild():  
 return self.parent.right  
 if self.isRightChild():  
 return self.parent.left  
 return None  
  
 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 = NodeOfRBT(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 = NodeOfRBT(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}>({depth})')  
 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

Код тестирования:

from RB\_Tree import RBT  
import RB\_Tree  
from F\_Handler import FileHandler  
from FS\_Combining import Combining  
  
  
def test():  
 # Заполнение файла 10000 записями.  
 import random  
 numbers = list(range(2401030000, 2401040000)) # Список с номерами страховых полисов.  
 random.shuffle(numbers) # Перемешивание списка.  
 f = open('Data\_Records.txt', 'w') # Открытие файла "Data\_Records.txt" на запись.  
 i = 0 # Для первой записи в номер\_полиса будет записан первый элемент из списка numbers.  
 for x in range(10000): # Повторять 10000 раз:  
 # Запись в строку: "номер\_полиса;компания;фамилия\_владельца".  
 f.write(str(numbers[i]) + ';' + 'Company' + str(random.randint(1, 9999)) + ';'  
 + 'Soname' + str(random.randint(1, 9999)) + "\n")  
 i += 1 # Для следующей записи в номер\_полиса будет записан следующий элемент из списка numbers.  
 f.close() # Закрытие файла "Data\_Records.txt".  
 # Создание объекта для работы с файлом с тремя полями.  
 fw = FileHandler('Data\_Records.bin', {'key': 11, 'company': 12, 'soname': 11})  
 tree = RBT() # Создание дерева.  
 comb = Combining(tree, fw) # Объединение работы файла и дерева.  
 fill\_comb(comb, 'Data\_Records.txt') # Заполнение из файла.  
 print(comb.combining) # Вывод дерева.  
 print()  
 # Количество выполненных поворотов.  
 print(f'Количество выполненных поворотов: {RB\_Tree.turns}')  
 print()  
 # Получение данных по ключу.  
 print(f'Первая запись из файла: {comb.get(str(numbers[0]))}')  
 print(f'Количество сравнений при получении первой записи: {RB\_Tree.compares}')  
 print()  
 print(f'Последняя запись из файла: {comb.get(str(numbers[9999]))}')  
 print(f'Количество сравнений при получении последней записи: {RB\_Tree.compares}')  
 print()  
 print(f'Запись в середине файла: {comb.get(str(numbers[4999]))}')  
 print(f'Количество сравнений при получении записи в середине файла: {RB\_Tree.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], company=line[1], soname=line[2])  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 test()

**Тестирование программы.**

После запуска тестирования в файл с данными было записано 10000 элементов (Рисунок 4).

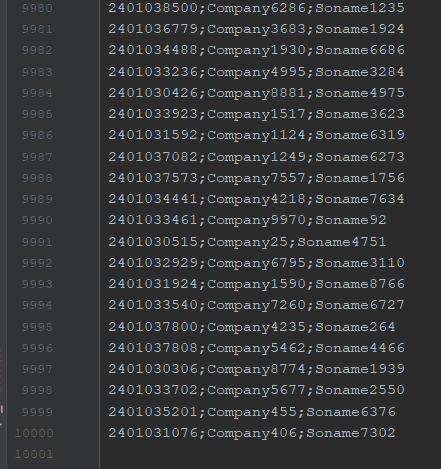


Рисунок 4 – Фрагмент файла из 10000 записей.

После этого было создано СДП: красно-чёрное дерево (Рисунок 5). Каждый узел дерева содержит информацию о своём цвете (узел – красный или чёрный), ключ – номер страхового полиса, а также информационное поле – номер строки в файле, который является ссылкой на информацию по ключу: компания, фамилия владельца. Также для удобства при выводе каждого узла справа от него отображается (в скобках) его уровень глубины в дереве.

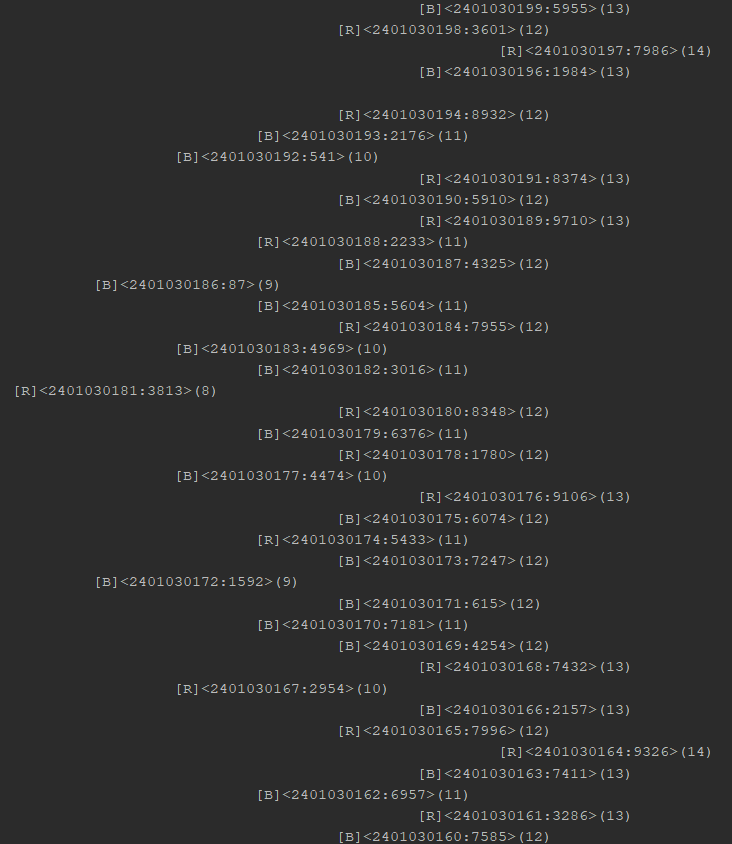


Рисунок 5 – Фрагмент СДП, построенного по файлу из 10000 записей.

Было получено количество произведённых поворотов (Рисунок 6).



Рисунок 6 – Количество произведённых поворотов.

После этого по дереву были получены следующие записи: первая запись файла, последняя запись файла и запись в середине файла. Для каждой полученной записи было найдено потраченное время на её получение, а также выведено количество произведённых сравнений во время получения записи (Рисунок 7).

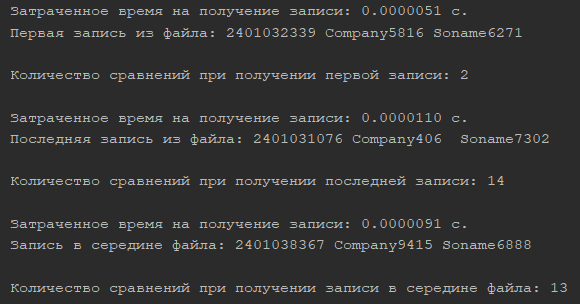


Рисунок 7 – Полученные данные, время поиска и количество сравнений.

# 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 BS\_Tree import BST  
import BS\_Tree  
from RB\_Tree import RBT  
import RB\_Tree  
from Hash\_Table import HT  
import Hash\_Table  
from F\_Handler import FileHandler  
from FS\_Combining import Combining  
  
  
def test():  
 print("--- Заполнение файла 1000 записями. ---")  
 print()  
 import random  
 numbers = list(range(240103000, 240104000)) # Список с номерами страховых полисов.  
 random.shuffle(numbers) # Перемешивание списка.  
 f = open('Data\_Records.txt', 'w') # Открытие файла "Data\_Records.txt" на запись.  
 i = 0 # Для первой записи в номер\_полиса будет записан первый элемент из списка numbers.  
 for x in range(1000): # Повторять 1000 раз:  
 # Запись в строку: "номер\_полиса;компания;фамилия\_владельца".  
 f.write(str(numbers[i]) + ';' + 'Company' + str(random.randint(1, 999)) + ';'  
 + 'Soname' + str(random.randint(1, 999)) + "\n")  
 i += 1 # Для следующей записи в номер\_полиса будет записан следующий элемент из списка numbers.  
 f.close() # Закрытие файла "Data\_Records.txt".  
 # Создание объекта для работы с файлом с тремя полями.  
 fw = FileHandler('Data\_Records.bin', {'key': 11, 'company': 12, 'soname': 11})  
 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\_Records.txt') # Заполнение БДП из файла.  
 fill\_comb(rb\_comb, 'Data\_Records.txt') # Заполнение СДП из файла.  
 fill\_comb(ht\_comb, 'Data\_Records.txt') # Заполнение хеш-таблицы из файла.  
 # print(bs\_comb.combining) # Вывод БДП.  
 # print(rb\_comb.combining) # Вывод СДП.  
 # HT.print() # Вывод хеш-таблицы.  
  
 # Получение первой записи файла по ключу.  
 print(">>> Первая запись <<<")  
 print()  
 print("С помощью БДП:")  
 bs\_comb.get(str(numbers[0]))  
 print(f'Количество сравнений: {BS\_Tree.compares}')  
 print()  
 print("С помощью СДП:")  
 rb\_comb.get(str(numbers[0]))  
 print(f'Количество сравнений: {RB\_Tree.compares}')  
 print()  
 print("С помощью хеш-таблицы:")  
 ht\_comb.get(str(numbers[0]))  
 print(f'Количество сравнений: {Hash\_Table.compares}')  
 print()  
  
 # Получение последней записи файла по ключу.  
 print(">>> Последняя запись <<<")  
 print()  
 print("С помощью БДП:")  
 bs\_comb.get(str(numbers[999]))  
 print(f'Количество сравнений: {BS\_Tree.compares}')  
 print()  
 print("С помощью СДП:")  
 rb\_comb.get(str(numbers[999]))  
 print(f'Количество сравнений: {RB\_Tree.compares}')  
 print()  
 print("С помощью хеш-таблицы:")  
 ht\_comb.get(str(numbers[999]))  
 print(f'Количество сравнений: {Hash\_Table.compares}')  
 print()  
  
 # Получение по ключу записи, расположенной в середине файла.  
 print(">>> Запись в середине файла <<<")  
 print()  
 print("С помощью БДП:")  
 bs\_comb.get(str(numbers[499]))  
 print(f'Количество сравнений: {BS\_Tree.compares}')  
 print()  
 print("С помощью СДП:")  
 rb\_comb.get(str(numbers[499]))  
 print(f'Количество сравнений: {RB\_Tree.compares}')  
 print()  
 print("С помощью хеш-таблицы:")  
 ht\_comb.get(str(numbers[499]))  
 print(f'Количество сравнений: {Hash\_Table.compares}')  
 print()  
  
 print("--- Заполнение файла 10000 записями. ---")  
 print()  
 import random  
 numbers = list(range(2401030000, 2401040000)) # Список с номерами страховых полисов.  
 random.shuffle(numbers) # Перемешивание списка.  
 f = open('Data\_Records.txt', 'w') # Открытие файла "Data\_Records.txt" на запись.  
 i = 0 # Для первой записи в номер\_полиса будет записан первый элемент из списка numbers.  
 for x in range(10000): # Повторять 10000 раз:  
 # Запись в строку: "номер\_полиса;компания;фамилия\_владельца".  
 f.write(str(numbers[i]) + ';' + 'Company' + str(random.randint(1, 9999)) + ';'  
 + 'Soname' + str(random.randint(1, 9999)) + "\n")  
 i += 1 # Для следующей записи в номер\_полиса будет записан следующий элемент из списка numbers.  
 f.close() # Закрытие файла "Data\_Records.txt".  
 # Создание объекта для работы с файлом с тремя полями.  
 fw = FileHandler('Data\_Records.bin', {'key': 11, 'company': 12, 'soname': 11})  
 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\_Records.txt') # Заполнение БДП из файла.  
 fill\_comb(rb\_comb, 'Data\_Records.txt') # Заполнение СДП из файла.  
 fill\_comb(ht\_comb, 'Data\_Records.txt') # Заполнение хеш-таблицы из файла.  
 # print(bs\_comb.combining) # Вывод БДП.  
 # print(rb\_comb.combining) # Вывод СДП.  
 # HT.print() # Вывод хеш-таблицы.  
  
 # Получение первой записи файла по ключу.  
 print(">>> Первая запись <<<")  
 print()  
 print("С помощью БДП:")  
 bs\_comb.get(str(numbers[0]))  
 print(f'Количество сравнений: {BS\_Tree.compares}')  
 print()  
 print("С помощью СДП:")  
 rb\_comb.get(str(numbers[0]))  
 print(f'Количество сравнений: {RB\_Tree.compares}')  
 print()  
 print("С помощью хеш-таблицы:")  
 ht\_comb.get(str(numbers[0]))  
 print(f'Количество сравнений: {Hash\_Table.compares}')  
 print()  
  
 # Получение последней записи файла по ключу.  
 print(">>> Последняя запись <<<")  
 print()  
 print("С помощью БДП:")  
 bs\_comb.get(str(numbers[9999]))  
 print(f'Количество сравнений: {BS\_Tree.compares}')  
 print()  
 print("С помощью СДП:")  
 rb\_comb.get(str(numbers[9999]))  
 print(f'Количество сравнений: {RB\_Tree.compares}')  
 print()  
 print("С помощью хеш-таблицы:")  
 ht\_comb.get(str(numbers[9999]))  
 print(f'Количество сравнений: {Hash\_Table.compares}')  
 print()  
  
 # Получение по ключу записи, расположенной в середине файла.  
 print(">>> Запись в середине файла <<<")  
 print()  
 print("С помощью БДП:")  
 bs\_comb.get(str(numbers[4999]))  
 print(f'Количество сравнений: {BS\_Tree.compares}')  
 print()  
 print("С помощью СДП:")  
 rb\_comb.get(str(numbers[4999]))  
 print(f'Количество сравнений: {RB\_Tree.compares}')  
 print()  
 print("С помощью хеш-таблицы:")  
 ht\_comb.get(str(numbers[4999]))  
 print(f'Количество сравнений: {Hash\_Table.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], company=line[1], soname=line[2])  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 test()

**Тестирование программы.**

После запуска тестирования в файл с данными было записано 1000 элементов (Рисунок 8).

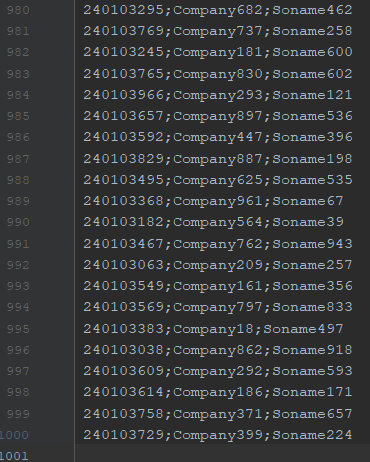


Рисунок 8 – Фрагмент файла из 1000 записей.

После этого были созданы следующие структуры данных: бинарное дерево поиска, сбалансированное дерево поиска (красно-чёрное дерево) и хеш-таблица (с цепным типом хеширования).

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).

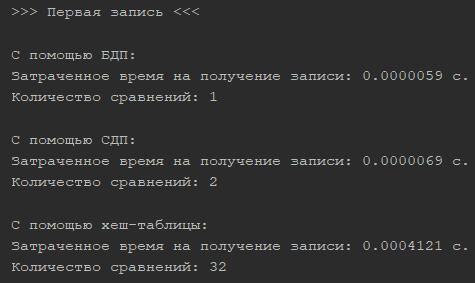


Рисунок 9 – Полученные данные, время поиска и количество сравнений для первой записи.

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

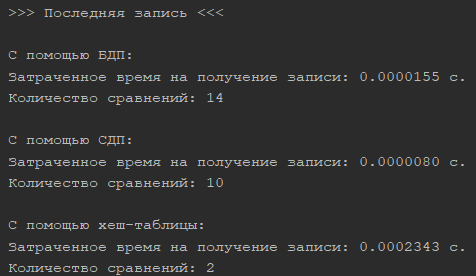


Рисунок 10 – Полученные данные, время поиска и количество сравнений для последней записи.

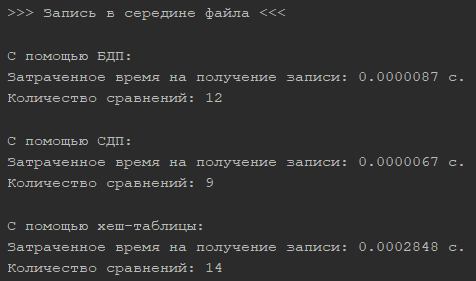


Рисунок 11 – Полученные данные, время поиска и количество сравнений для записи в середине файла.

Тестирование продолжилось: был файл с данными было записано уже 10000 элементов (Рисунок 12).

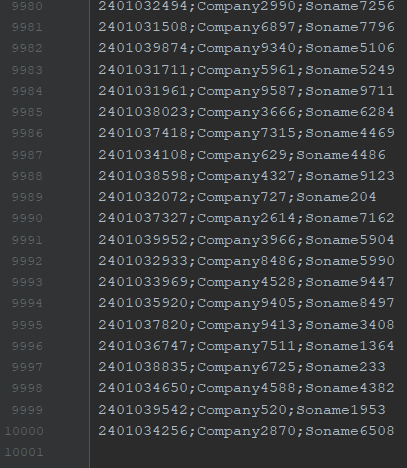


Рисунок 12 – Фрагмент файла из 10000 записей.

Были созданы структуры данных: бинарное дерево поиска, сбалансированное дерево поиска (красно-чёрное дерево) и хеш-таблица (с цепным типом хеширования).

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 выведено количество произведённых сравнений во время получения записи (Рисунок 13).

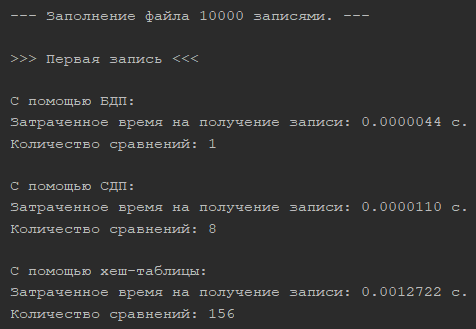


Рисунок 13 – Полученные данные, время поиска и количество сравнений для первой записи.

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

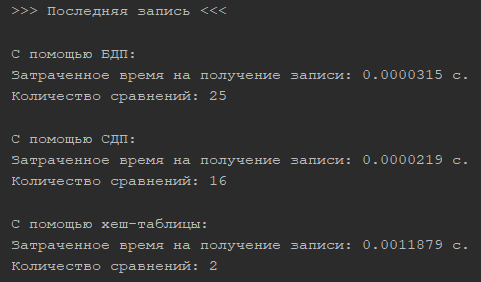


Рисунок 14 – Полученные данные, время поиска и количество сравнений для последней записи.

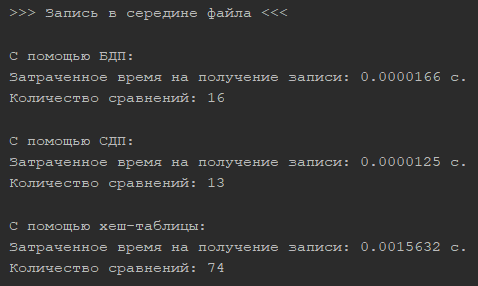


Рисунок 15 – Полученные данные, время поиска и количество сравнений для записи в середине файла.

**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 | 0.0059 |
| Красно-чёрное | n | 2 | 0.0069 |
| Хэш-таблица | n | 32 | 0.4121 |
| 10000 | БДП | n | 1 | 0.0044 |
| Красно-чёрное | n | 8 | 0.0110 |
| Хэш-таблица | n | 156 | 1.2722 |

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 | 14 | 0.0155 |
| Красно-чёрное | n | 10 | 0.0080 |
| Хэш-таблица | n | 2 | 0.2343 |
| 10000 | БДП | n | 25 | 0.0315 |
| Красно-чёрное | n | 16 | 0.0219 |
| Хэш-таблица | n | 2 | 1.1879 |

Получение записи 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 | 12 | 0.0087 |
| Красно-чёрное | n | 9 | 0.0067 |
| Хэш-таблица | n | 14 | 0.2848 |
| 10000 | БДП | n | 16 | 0.0166 |
| Красно-чёрное | n | 13 | 0.0125 |
| Хэш-таблица | n | 74 | 1.5632 |

Исходя из полученных результатов, можно сделать вывод, что в общем случае наиболее быстрое получение данных производилось с помощью сбалансированного дерева поиска: красно-чёрного дерева. Поиск данных посредством бинарного дерева поиска уступает по скорости предыдущему способу за исключением случая поиска первой записи файла. Получение данных с помощью хеш-таблицы во всех случаях происходило медленнее остальных двух способов поиска данных.

# 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.