



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.

(подпись)

Салов В.Д.

Преподаватель

«07» ноября 2021 г.

(подпись)

Сорокин А.В.

Москва, 2021 г.

СОДЕРЖАНИЕ

Цель работы	3
Задание 1	3
Задание 2	10
Задание 3	20
Вывод.....	30
Список информационных источников	31

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: <u>номер</u> , 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)) + ';'
                + 'Sname' + 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).

9980	2401032039;Company2031;Sname8077
9981	2401037921;Company4862;Sname633
9982	2401035313;Company2474;Sname8714
9983	2401030528;Company4073;Sname8921
9984	2401036995;Company4482;Sname2425
9985	2401039835;Company1949;Sname1487
9986	2401031281;Company6100;Sname2979
9987	2401038222;Company4604;Sname9923
9988	2401036422;Company6191;Sname9594
9989	2401038332;Company2275;Sname4220
9990	2401036888;Company1206;Sname3813
9991	2401032083;Company968;Sname6097
9992	2401033011;Company4158;Sname7630
9993	2401038949;Company7501;Sname1283
9994	2401034228;Company996;Sname1053
9995	2401033537;Company8775;Sname7242
9996	2401038292;Company8179;Sname9030
9997	2401036752;Company4051;Sname2984
9998	2401030083;Company4339;Sname6607
9999	2401039371;Company2386;Sname660
10000	2401037011;Company3513;Sname7647
10001	

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.

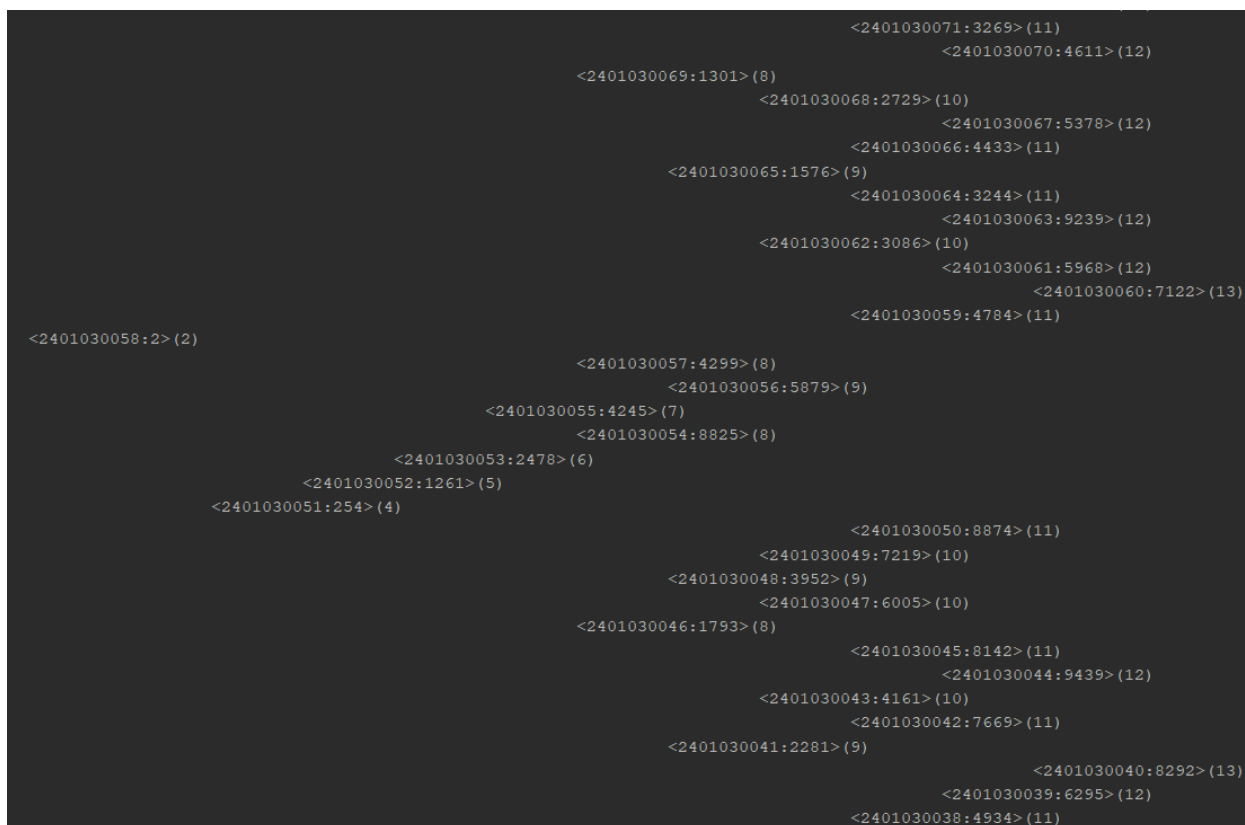


Рисунок 2 – Фрагмент БДП из 10000 узлов.

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).

```

Затраченное время на получение записи: 0.0000040 с.
Первая запись из файла: 2401039315 Company3604 Soname20

Количество произведённых сравнений при получении первой записи: 1

Затраченное время на получение записи: 0.0000125 с.
Последняя запись из файла: 2401037011 Company3513 Soname7647

Количество произведённых сравнений при получении последней записи: 17

Затраченное время на получение записи: 0.0000112 с.
Запись в середине файла: 2401032995 Company6860 Soname7249

Количество произведённых сравнений при получении записи в середине файла: 17

```

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: <u>номер</u> , 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)) + ';'
                + 'Sname' + 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).

9980	2401038500;Company6286;Sname1235
9981	2401036779;Company3683;Sname1924
9982	2401034488;Company1930;Sname6686
9983	2401033236;Company4995;Sname3284
9984	2401030426;Company8881;Sname4975
9985	2401033923;Company1517;Sname3623
9986	2401031592;Company1124;Sname6319
9987	2401037082;Company1249;Sname6273
9988	2401037573;Company7557;Sname1756
9989	2401034441;Company4218;Sname7634
9990	2401033461;Company9970;Sname92
9991	2401030515;Company25;Sname4751
9992	2401032929;Company6795;Sname3110
9993	2401031924;Company1590;Sname8766
9994	2401033540;Company7260;Sname6727
9995	2401037800;Company4235;Sname264
9996	2401037808;Company5462;Sname4466
9997	2401030306;Company8774;Sname1939
9998	2401033702;Company5677;Sname2550
9999	2401035201;Company455;Sname6376
10000	2401031076;Company406;Sname7302
10001	

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.

```

[R]<2401030194:8932>(12)
[B]<2401030193:2176>(11)
[B]<2401030192:541>(10)
[R]<2401030191:8374>(13)
[B]<2401030190:5910>(12)
[R]<2401030189:9710>(13)
[R]<2401030188:2233>(11)
[B]<2401030187:4325>(12)
[B]<2401030186:87>(9)
[B]<2401030185:5604>(11)
[R]<2401030184:7955>(12)
[B]<2401030183:4969>(10)
[B]<2401030182:3016>(11)
[R]<2401030181:3813>(8)
[R]<2401030180:8348>(12)
[B]<2401030179:6376>(11)
[R]<2401030178:1780>(12)
[B]<2401030177:4474>(10)
[R]<2401030176:9106>(13)
[B]<2401030175:6074>(12)
[R]<2401030174:5433>(11)
[B]<2401030173:7247>(12)
[B]<2401030172:1592>(9)
[B]<2401030171:615>(12)
[B]<2401030170:7181>(11)
[B]<2401030169:4254>(12)
[R]<2401030168:7432>(13)
[R]<2401030167:2954>(10)
[B]<2401030166:2157>(13)
[R]<2401030165:7996>(12)

```

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

Было получено количество произведённых поворотов (Рисунок 6).

Количество выполненных поворотов: 5844

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).

```

Затраченное время на получение записи: 0.0000051 с.
Первая запись из файла: 2401032339 Company5816 Soname6271

Количество сравнений при получении первой записи: 2

Затраченное время на получение записи: 0.0000110 с.
Последняя запись из файла: 2401031076 Company406 Soname7302

Количество сравнений при получении последней записи: 14

Затраченное время на получение записи: 0.0000091 с.
Запись в середине файла: 2401038367 Company9415 Soname6888

Количество сравнений при получении записи в середине файла: 13

```

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()

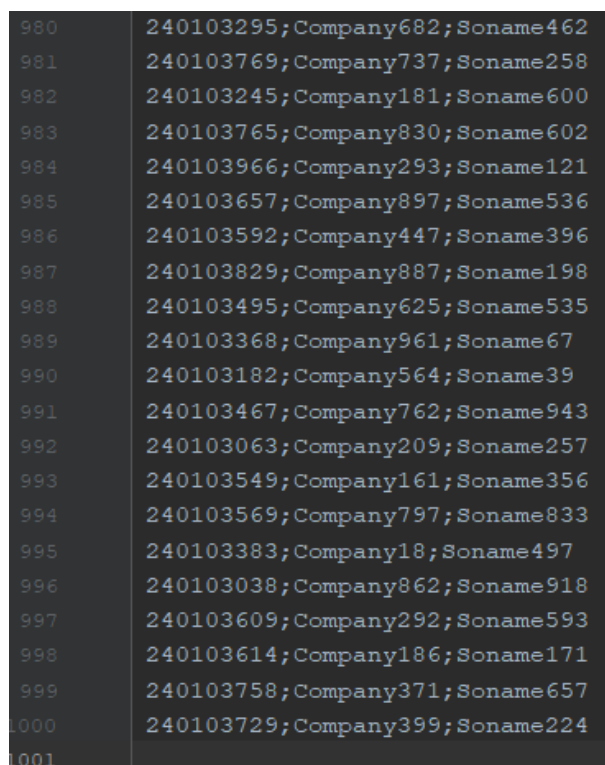
def fill_comb(comb: 'Combining', path: str):
    with open(path, 'r') as f:
        for line in f:
            line = line.split(';')
            comb.add(key=line[0], company=line[1], soname=line[2])

if __name__ == '__main__':
    test()

```

Тестирование программы.

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



980	240103295;Company682;Sname462
981	240103769;Company737;Sname258
982	240103245;Company181;Sname600
983	240103765;Company830;Sname602
984	240103966;Company293;Sname121
985	240103657;Company897;Sname536
986	240103592;Company447;Sname396
987	240103829;Company887;Sname198
988	240103495;Company625;Sname535
989	240103368;Company961;Sname67
990	240103182;Company564;Sname39
991	240103467;Company762;Sname943
992	240103063;Company209;Sname257
993	240103549;Company161;Sname356
994	240103569;Company797;Sname833
995	240103383;Company18;Sname497
996	240103038;Company862;Sname918
997	240103609;Company292;Sname593
998	240103614;Company186;Sname171
999	240103758;Company371;Sname657
1000	240103729;Company399;Sname224
1001	

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).


```
>>> Первая запись <<<

С помощью БДП:
Затраченное время на получение записи: 0.0000059 с.
Количество сравнений: 1

С помощью СДП:
Затраченное время на получение записи: 0.0000069 с.
Количество сравнений: 2

С помощью хеш-таблицы:
Затраченное время на получение записи: 0.0004121 с.
Количество сравнений: 32
```

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.

```
>>> Последняя запись <<<

С помощью БДП:
Затраченное время на получение записи: 0.0000155 с.
Количество сравнений: 14

С помощью СДП:
Затраченное время на получение записи: 0.0000080 с.
Количество сравнений: 10

С помощью хеш-таблицы:
Затраченное время на получение записи: 0.0002343 с.
Количество сравнений: 2
```

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

```
>>> Запись в середине файла <<<

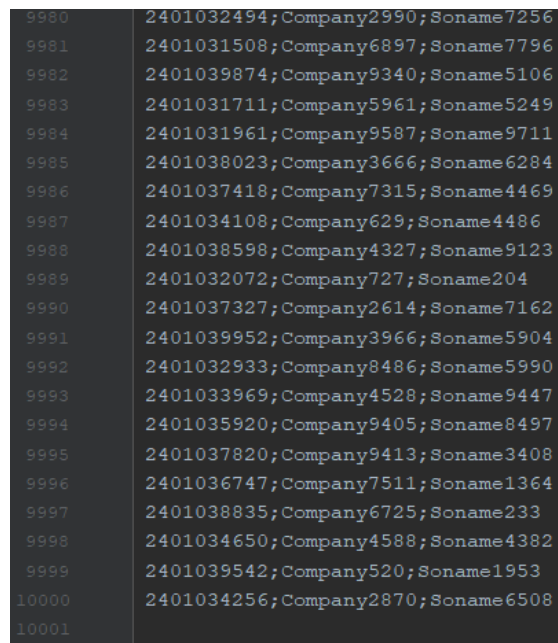
С помощью БДП:
Затраченное время на получение записи: 0.0000087 с.
Количество сравнений: 12

С помощью СДП:
Затраченное время на получение записи: 0.0000067 с.
Количество сравнений: 9

С помощью хеш-таблицы:
Затраченное время на получение записи: 0.0002848 с.
Количество сравнений: 14
```

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).

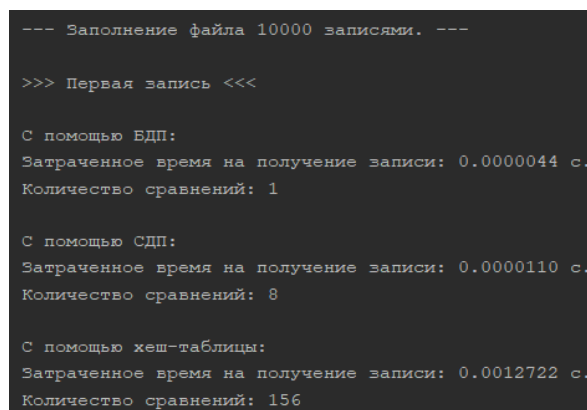


9980	2401032494;Company2990;Soname7256
9981	2401031508;Company6897;Soname7796
9982	2401039874;Company9340;Soname5106
9983	2401031711;Company5961;Soname5249
9984	2401031961;Company9587;Soname9711
9985	2401038023;Company3666;Soname6284
9986	2401037418;Company7315;Soname4469
9987	2401034108;Company629;Soname4486
9988	2401038598;Company4327;Soname9123
9989	2401032072;Company727;Soname204
9990	2401037327;Company2614;Soname7162
9991	2401039952;Company3966;Soname5904
9992	2401032933;Company8486;Soname5990
9993	2401033969;Company4528;Soname9447
9994	2401035920;Company9405;Soname8497
9995	2401037820;Company9413;Soname3408
9996	2401036747;Company7511;Soname1364
9997	2401038835;Company6725;Soname233
9998	2401034650;Company4588;Soname4382
9999	2401039542;Company520;Soname1953
10000	2401034256;Company2870;Soname6508
10001	

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).



```
--- Заполнение файла 10000 записями. ---

>>> Первая запись <<<

С помощью БДП:
Затраченное время на получение записи: 0.0000044 с.
Количество сравнений: 1

С помощью СДП:
Затраченное время на получение записи: 0.0000110 с.
Количество сравнений: 8

С помощью хеш-таблицы:
Затраченное время на получение записи: 0.0012722 с.
Количество сравнений: 156
```

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.

```
>>> Последняя запись <<<

С помощью БДП:
Затраченное время на получение записи: 0.0000315 с.
Количество сравнений: 25

С помощью СДП:
Затраченное время на получение записи: 0.0000219 с.
Количество сравнений: 16

С помощью кеш-таблицы:
Затраченное время на получение записи: 0.0011879 с.
Количество сравнений: 2
```

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

```
>>> Запись в середине файла <<<

С помощью БДП:
Затраченное время на получение записи: 0.0000166 с.
Количество сравнений: 16

С помощью СДП:
Затраченное время на получение записи: 0.0000125 с.
Количество сравнений: 13

С помощью кеш-таблицы:
Затраченное время на получение записи: 0.0015632 с.
Количество сравнений: 74
```

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 хэш-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.