Министерство цифрового развития, связи и массовых коммуникаций РФ ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования «МОСКОВСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ СВЯЗИ И ИНФОРМАТИКИ»

Кафедра «Информатика»

Дисциплина: Информационные технологии и программирование

Лабораторная работа №5

По теме: «Разработка алгоритмов формирования и обработки

двоичных деревьев поиска, авл-деревьев и в-деревьев»

Вариант №22  
Выполнил: студент гр. БЭИ2202 Сериков Д. Н.

Проверил: зав. кафедры «Информатика» Халабия Р. Ф.

Москва, 2024 г.

1. **Постановка задачи**

Дано узлов n = 40 значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от -190 до 190, Х= 15, для В+ дерева m =5.

Согласно варианту написать программу выполняющие следующие действия. Вывод деревьев выполнить в графическом виде.

1. Построить бинарное дерево.
2. Найти сумму элементов бинарного дерева.
3. Найти произведение элементов кратных 3
4. Найти вершины, у которых количество потомков в левом поддереве не равно количеству потомков в правом поддереве.
5. Найти вершины, для которых высота левого поддерева не равна высоте правого поддерева.
6. Определить число вхождений элемента x в бинарное дерево.
7. Найти максимальный элемент бинарного дерева и количество повторений максимального элемента в данном дереве.
8. Определить, есть ли в бинарном дереве хотя бы два одинаковых элемента.
9. Определить максимальное количество одинаковых элементов бинарного дерева.
10. Определить, является ли бинарное дерево симметричным.
11. Определить, является ли бинарное дерево деревом поиска.
12. Вывести все листья дерева поиска в порядке возрастания.
13. Сформировать два идеально сбалансированных дерева из отрицательных инеотрицательных элементов дерева.
14. Вывести на экран все пути, ведущие от корня к листьям бинарного дерева, у которых суммарный вес элементов минимальный.
15. Найти последний номер из всех уровней бинарного дерева, на которых есть положительные элементы.
16. На каждом уровне бинарного дерева найти максимальный элемент.
17. На каждом уровне дерева найти количество внутренних вершин и количество листьев.
18. Найти суммы элементов всех нечетных уровней.
19. Найти минимальный и максимальный пути между листьями бинарного дерева.
20. Удалить из бинарного дерева наименьшее количество вершин таким образом, чтобы полученное дерево было строго бинарным.
21. Вывести элементы дерева на экран используя следующие обходы дерева:
22. инфиксным обходом
23. постфиксным обходом
24. префиксным обходом
25. Вставить новые элементы в дерево так чтобы оно стало деревом B+.
26. Преобразовать B+дерево в красно-чёрное дерево.
27. **Алгоритм решения задачи**

**Метод createNode:**

Шаг 1: Создать новый узел.

Шаг 2: Установить ключ узла равным переданному значению.

Шаг 3: Инициализировать левого и правого потомков как пустые.

Шаг 4: Вернуть указатель на созданный узел.

**Метод addNode:**

Шаг 1: Если корень пуст, вызвать метод создания узла с ключом и вернуть результат.

Шаг 2: Если ключ меньше текущего узла, добавить узел в левое поддерево рекурсивно.

Шаг 3: Если ключ больше или равен текущему узлу, добавить узел в правое поддерево рекурсивно.

Шаг 4: Вернуть текущий узел.

**Метод buildTree:**

Шаг 1: Установить начальное значение для генерации случайных чисел.

Шаг 2: Для каждого из n узлов:

1. Сгенерировать случайный ключ в диапазоне [minKey, maxKey].

2. Вызвать метод добавления узла с текущим ключом.

Шаг 3: Завершить построение дерева.

**Метод sumElements:**

Шаг 1: Если текущий узел пуст, вернуть 0.

Шаг 2: Вычислить сумму текущего узла, суммы левого поддерева и суммы правого поддерева рекурсивно.

Шаг 3: Вернуть вычисленную сумму.

**Метод productOfMultiplesOfThree:**

Шаг 1: Если текущий узел пуст, вернуть 1.

Шаг 2: Если ключ текущего узла кратен 3, учесть его в произведении.

Шаг 3: Рекурсивно вычислить произведение левого и правого поддеревьев.

Шаг 4: Вернуть итоговое произведение.

**Метод countDescendants:**

Шаг 1: Если текущий узел пуст, вернуть 0.

Шаг 2: Рекурсивно вычислить количество потомков для левого и правого поддеревьев.

Шаг 3: Суммировать 1 (текущий узел) с количеством потомков поддеревьев.

Шаг 4: Вернуть результат.

**Метод findNodesWithUnequalDescendants:**

Шаг 1: Если текущий узел пуст, завершить метод.

Шаг 2: Вычислить количество потомков левого и правого поддеревьев.

Шаг 3: Если количество потомков различается, добавить ключ текущего узла в результат.

Шаг 4: Рекурсивно обработать левое и правое поддеревья.

**Метод calculateHeight:**

Шаг 1: Если текущий узел пуст, вернуть 0.

Шаг 2: Рекурсивно вычислить высоту левого и правого поддеревьев.

Шаг 3: Вернуть 1 плюс максимум из высот поддеревьев.

**Метод findNodesWithUnequalHeights:**

Шаг 1: Если текущий узел пуст, завершить метод.

Шаг 2: Вычислить высоты левого и правого поддеревьев.

Шаг 3: Если высоты различаются, добавить ключ текущего узла в результат.

Шаг 4: Рекурсивно обработать левое и правое поддеревья.

**Метод countOccurrences:**

Шаг 1: Если текущий узел пуст, вернуть 0.

Шаг 2: Проверить, равен ли ключ текущего узла заданному значению.

Шаг 3: Рекурсивно вычислить количество вхождений в левом и правом поддеревьях.

Шаг 4: Суммировать результаты.

**Метод findMaxElement:**

Шаг 1: Если текущий узел пуст, завершить метод.

Шаг 2: Если ключ текущего узла больше максимального, обновить максимальный ключ и обнулить счетчик.

Шаг 3: Если ключ равен максимальному, увеличить счетчик.

Шаг 4: Рекурсивно обработать левое и правое поддеревья.

**Метод isSymmetric:**

Шаг 1: Если оба поддерева пусты, вернуть true.

Шаг 2: Если только одно из поддеревьев пусто, вернуть false.

Шаг 3: Сравнить ключи текущих узлов и рекурсивно проверить симметрию:

1. Левого поддерева первого узла и правого поддерева второго узла.

2. Правого поддерева первого узла и левого поддерева второго узла.

**Метод isBST:**

Шаг 1: Если текущий узел пуст, вернуть true.

Шаг 2: Если ключ текущего узла меньше минимального допустимого или больше максимального допустимого, вернуть false.

Шаг 3: Рекурсивно проверить левое поддерево с обновленным максимальным значением.

Шаг 4: Рекурсивно проверить правое поддерево с обновленным минимальным значением.

Шаг 5: Вернуть true, если обе проверки прошли.

**Метод collectLeaves:**

Шаг 1: Если текущий узел пуст, завершить метод.

Шаг 2: Если узел является листом (нет потомков), добавить его ключ в список листьев.

Шаг 3: Рекурсивно обработать левое и правое поддеревья.

**Метод findpaths:**

Шаг 1: Если текущий узел пуст, завершить метод.

Шаг 2: Добавить ключ текущего узла в текущий путь.

Шаг 3: Если узел является листом, добавить текущий путь в список путей.

Шаг 4: Рекурсивно обработать левое и правое поддеревья.

Шаг 5: Удалить текущий узел из пути (возврат на уровень выше).

**Метод findMinWeightPath:**

Шаг 1: Найти все пути от корня до листьев.

Шаг 2: Для каждого пути вычислить сумму ключей.

Шаг 3: Найти путь с минимальной суммой ключей.

Шаг 4: Вернуть путь с минимальным весом.

**Метод findLastLevelWithPositive:**

Шаг 1: Инициализировать очередь узлов с уровнем корня равным 0.

Шаг 2: Для каждого узла из очереди:

1. Если ключ узла положительный, обновить последний уровень с положительными числами.

2. Добавить левого и правого потомков в очередь с увеличением уровня.

Шаг 3: Вернуть последний уровень с положительными числами.

**Метод findMaxOnEachLevel:**

Шаг 1: Инициализировать очередь узлов с уровнями.

Шаг 2: Для каждого узла из очереди:

1. Если уровень больше размера списка максимумов, добавить ключ как максимум для нового уровня.

2. Иначе обновить максимум текущего уровня.

3. Добавить потомков узла в очередь с увеличением уровня.

Шаг 3: Вернуть список максимумов.

**Метод findNodesAndLeavesOnEachLevel:**

Шаг 1: Инициализировать очередь узлов с уровнями.

Шаг 2: Для каждого узла из очереди:

1. Если уровень больше размеров списков, добавить новые элементы для уровня.

2. Если узел является листом, увеличить счетчик листьев текущего уровня.

3. Иначе увеличить счетчик внутренних узлов.

4. Добавить потомков в очередь с увеличением уровня.

Шаг 3: Вернуть списки внутренних узлов и листьев.

**Метод sumOfOddLevels:**

Шаг 1: Инициализировать очередь узлов с уровнями.

Шаг 2: Для каждого узла из очереди:

1. Если уровень нечетный, добавить ключ узла к сумме.

2. Добавить потомков узла в очередь с увеличением уровня.

Шаг 3: Вернуть итоговую сумму.

**Метод findMinMaxPaths:**

Шаг 1: Если текущий узел пуст, завершить метод.

Шаг 2: Если узел является листом:

1. Обновить минимальную длину пути.

2. Обновить максимальную длину пути.

Шаг 3: Рекурсивно обработать левое и правое поддеревья с увеличением текущей длины.

**Метод collectElements:**

Шаг 1: Если текущий узел пуст, завершить метод.

Шаг 2: Рекурсивно собрать элементы из левого поддерева.

Шаг 3: Добавить ключ текущего узла в список.

Шаг 4: Рекурсивно собрать элементы из правого поддерева.

**Метод buildBalancedTree:**

Шаг 1: Если начальный индекс больше конечного, вернуть null.

Шаг 2: Найти середину массива элементов.

Шаг 3: Создать узел с элементом из середины.

Шаг 4: Рекурсивно построить левое и правое поддеревья.

Шаг 5: Вернуть корень дерева.

**Метод createBalancedTrees:**

Шаг 1: Собрать все элементы дерева.

Шаг 2: Разделить элементы на отрицательные и неотрицательные списки.

Шаг 3: Отсортировать оба списка.

Шаг 4: Построить сбалансированное дерево для каждого списка.

Шаг 5: Вернуть оба дерева.

**Метод transformToStrictlyBinary:**

Шаг 1: Если текущий узел пуст, вернуть null.

Шаг 2: Рекурсивно преобразовать левое и правое поддеревья.

Шаг 3: Если у узла два потомка или нет потомков, вернуть узел.

Шаг 4: Если у узла один потомок, удалить текущий узел, вернуть его потомка.

**Метод copyTree:**

Шаг 1: Если текущий узел пуст, вернуть null.

Шаг 2: Создать новый узел с ключом текущего узла.

Шаг 3: Рекурсивно скопировать левое и правое поддеревья.

Шаг 4: Вернуть указатель на новый узел.

**Метод inOrderTraversal:**

Шаг 1: Если текущий узел пуст, завершить выполнение метода.

Шаг 2: Рекурсивно вызвать inOrderTraversal для левого поддерева.

Шаг 3: Добавить ключ текущего узла в выходной поток.

Шаг 4: Рекурсивно вызвать inOrderTraversal для правого поддерева.

**Метод postOrderTraversal:**

Шаг 1: Если текущий узел пуст, завершить выполнение метода.

Шаг 2: Рекурсивно вызвать postOrderTraversal для левого поддерева.

Шаг 3: Рекурсивно вызвать postOrderTraversal для правого поддерева.

Шаг 4: Добавить ключ текущего узла в выходной поток.

**Метод preOrderTraversal:**

Шаг 1: Если текущий узел пуст, завершить выполнение метода.

Шаг 2: Добавить ключ текущего узла в выходной поток.

Шаг 3: Рекурсивно вызвать preOrderTraversal для левого поддерева.

Шаг 4: Рекурсивно вызвать preOrderTraversal для правого поддерева.

**Метод splitBPlusNode:**

Шаг 1: Получить максимальное количество ключей в узле на основе параметра t.

Шаг 2: Найти средний индекс ключей.

Шаг 3: Создать новый узел и настроить его свойства.

Шаг 4: Если текущий узел является листом:

Шаг 4.1: Перенести правую часть ключей и указателей в новый узел.

Шаг 4.2: Настроить связи между соседними листовыми узлами.

Шаг 4.3: Получить первый ключ нового узла и обновить родителя.

Шаг 5: Если текущий узел не является листом:

Шаг 5.1: Перенести правую часть ключей и дочерних указателей в новый узел.

Шаг 5.2: Обновить родителя для перемещённых дочерних узлов.  
Шаг 6: Если текущий узел является корнем:

Шаг 6.1: Создать новый корень и обновить его ключи и дочерние указатели.

Шаг 7: Если родительский узел переполнен, рекурсивно вызвать splitBPlusNode для родителя.

**Метод findLeaf:**

Шаг 1: Установить текущий узел на корень дерева.

Шаг 2: Если дерево пустое, вернуть null.

Шаг 3: Пока текущий узел не является листовым:

Шаг 3.1: Найти подходящий ребёнок для перехода по ключу.

Шаг 3.2: Перейти к выбранному ребёнку.

Шаг 4: Вернуть текущий узел.

**Метод insertKey:**

Шаг 1: Если дерево пустое, создать новый корневой узел.

Шаг 2: Найти листовой узел для вставки с помощью findLeaf.

Шаг 3: Вставить ключ и значение в листовой узел в нужной позиции.

Шаг 4: Если узел переполнен, вызвать splitBPlusNode для листового узла.

**Метод insertBinaryTreeKeysToBPlus:**

Шаг 1: Если текущий узел пуст, завершить выполнение метода.

Шаг 2: Вставить ключ текущего узла в B+ дерево.

Шаг 3: Рекурсивно вызвать insertBinaryTreeKeysToBPlus для левого поддерева.

Шаг 4: Рекурсивно вызвать insertBinaryTreeKeysToBPlus для правого поддерева.

**Метод initializeRedBlackTree:**

Шаг 1: Создать NIL-узел с чёрным цветом.

Шаг 2: Установить NIL-узел как корень дерева.

**Метод leftRotate:**

Шаг 1: Сохранить правого потомка текущего узла.

Шаг 2: Переместить левого потомка правого узла на место правого потомка текущего узла.

Шаг 3: Обновить родителя нового узла.

Шаг 4: Обновить связи между узлами для завершения левого поворота.

**Метод rightRotate:**

Шаг 1: Сохранить левого потомка текущего узла.

Шаг 2: Переместить правого потомка левого узла на место левого потомка текущего узла.

Шаг 3: Обновить родителя нового узла.

Шаг 4: Обновить связи между узлами для завершения правого поворота.

**Метод fixInsert:**

Шаг 1: Пока текущий узел и его родитель имеют красный цвет:

Шаг 1.1: Если родитель текущего узла является левым потомком:

Шаг 1.1.1: Выполнить операции с "дядей" узла в зависимости от его цвета.

Шаг 1.1.2: Выполнить вращение для корректировки дерева.

Шаг 1.2: Иначе выполнить аналогичные операции для правого потомка.

Шаг 2: Установить корень дерева чёрным.

**Метод inorderTraversal для красно-чёрного дерева:**

Шаг 1: Если текущий узел равен NIL, завершить выполнение метода.

Шаг 2: Рекурсивно вызвать inorderTraversal для левого поддерева.

Шаг 3: Напечатать значение текущего узла.

Шаг 4: Рекурсивно вызвать inorderTraversal для правого поддерева.

**Метод search:**

Шаг 1: Если текущий узел равен NIL или его данные совпадают с искомыми, вернуть текущий узел.

Шаг 2: Если значение меньше данных текущего узла, рекурсивно вызвать search для левого потомка.

Шаг 3: Иначе рекурсивно вызвать search для правого потомка.

**Метод searchRedBlackTree:**

Шаг 1: Вызвать метод search, передав в него корень красно-чёрного дерева, NIL и искомое значение.

Шаг 2: Вернуть результат вызова метода search.

**Метод insertBPlusTreeKeysToRBT:**

Шаг 1: Если текущий узел пуст, завершить выполнение метода.

Шаг 2: Если текущий узел является листовым:

Шаг 2.1: Для каждого ключа в текущем узле вызвать метод insert для красно-чёрного дерева.

Шаг 3: Если текущий узел не является листовым:

Шаг 3.1: Для каждого ребёнка текущего узла рекурсивно вызвать метод insertBPlusTreeKeysToRBT.

**Метод** **drawNodeRedBlackTree:**

Шаг 1: Если текущий узел равен nullptr или NIL, завершить выполнение метода.

Шаг 2: Если у текущего узла есть левый потомок:

Шаг 2.1: Нарисовать линию от текущего узла к левому потомку.

Шаг 2.2: Рекурсивно вызвать drawNodeRedBlackTree для левого потомка.

Шаг 3: Если у текущего узла есть правый потомок:

Шаг 3.1: Нарисовать линию от текущего узла к правому потомку.

Шаг 3.2: Рекурсивно вызвать drawNodeRedBlackTree для правого потомка.

Шаг 4: Создать и настроить круг для отображения текущего узла.

Шаг 5: Настроить текстовое отображение значения узла и нарисовать его.

**Метод** **drawRedBlackTree:**

Шаг 1: Загрузить шрифт для отображения узлов.

Шаг 2: Вызвать метод drawNodeRedBlackTree, передав параметры корня дерева, его NIL и начальные координаты.

**Метод drawRedBlackTreeWindow:**

Шаг 1: Создать окно для отображения красно-чёрного дерева.

Шаг 2: Настроить кнопку для выхода.

Шаг 3: Инициализировать красно-чёрное дерево.

Шаг 4: Вызвать insertBPlusTreeKeysToRBT, чтобы вставить ключи из дерева B+ в красно-чёрное дерево.

Шаг 5: В цикле событий окна:

Шаг 5.1: Проверить закрытие окна.

Шаг 5.2: Обработать нажатие кнопки "Exit".

Шаг 6: Очистить окно, нарисовать дерево и кнопку, обновить окно.

**Метод drawTree:**

Шаг 1: Если текущий узел пуст, завершить выполнение метода.

Шаг 2: Создать круг для узла и настроить его параметры.

Шаг 3: Если у узла есть левый потомок:

Шаг 3.1: Нарисовать линию к левому потомку.

Шаг 3.2: Рекурсивно вызвать drawTree для левого потомка.

Шаг 4: Если у узла есть правый потомок:

Шаг 4.1: Нарисовать линию к правому потомку.

Шаг 4.2: Рекурсивно вызвать drawTree для правого потомка.

Шаг 5: Нарисовать текущий узел и его значение.

**Метод drawBalanceTreeWindow:**

Шаг 1: Создать два новых дерева: для отрицательных и неотрицательных элементов.

Шаг 2: Заполнить деревья с помощью createBalancedTrees.

Шаг 3: Создать окно для отображения сбалансированных деревьев.

Шаг 4: Настроить кнопку для перехода к следующему окну.

Шаг 5: В цикле событий окна:

Шаг 5.1: Проверить закрытие окна.

Шаг 5.2: Обработать нажатие кнопки "Next".

Шаг 6: Очистить окно, нарисовать оба дерева, кнопку и обновить окно.

**Метод drawRemoveMinVertexWindow:**

Шаг 1: Создать копию дерева и преобразовать её в строго бинарное дерево.

Шаг 2: Создать окно для отображения строго бинарного дерева.

Шаг 3: Настроить кнопку для перехода к следующему окну.

Шаг 4: В цикле событий окна:

Шаг 4.1: Проверить закрытие окна.

Шаг 4.2: Обработать нажатие кнопки "Next".

Шаг 5: Очистить окно, нарисовать дерево, кнопку и обновить окно.

1. **Листинг программы**

|  |
| --- |
| #include <SFML/Graphics.hpp>  #include <iostream>  #include <cstdlib>  #include <ctime>  #include <set>  #include <map>  #include <queue>  #include <limits>  #include <numeric>  #include <algorithm>  #include <vector>  #include <sstream>  **struct** Node {  **int** key;  Node\* left;  Node\* right;  };  Node\* **createNode**(**int** key) {  Node\* newNode = **new** Node;  newNode->key = key;  newNode->left = nullptr;  newNode->right = nullptr;  **return** newNode;  }  Node\* **addNode**(Node\* root, **int** key) {  **if** (!root)  **return** createNode(key);  **if** (key < root->key)  root->left = addNode(root->left, key);  **else**  root->right = addNode(root->right, key);  **return** root;  }  **void** **buildTree**(Node\*& root, **int** n, **int** minKey, **int** maxKey) {  srand(**static\_cast**<**unsigned**>(time(**0**)));  **for** (**int** i = **0**; i < n; ++i) {  **int** key = minKey + rand() % (maxKey - minKey + **1**);  root = addNode(root, key);  }  }  **int** **sumElements**(Node\* node) {  **if** (!node)  **return** **0**;  **return** node->key + sumElements(node->left) + sumElements(node->right);  }  **int64\_t** **productOfMultiplesOfThree**(Node\* node) {  **if** (!node)  **return** **1**;  **int64\_t** product = (node->key % **3** == **0**) ? node->key : **1**;  **return** product \* productOfMultiplesOfThree(node->left) \* productOfMultiplesOfThree(node->right);  }  **int** **countDescendants**(Node\* node) {  **if** (!node) **return** **0**;  **return** **1** + countDescendants(node->left) + countDescendants(node->right);  }  **void** **findNodesWithUnequalDescendants**(Node\* node, std::vector<**int**>& result) {  **if** (!node) **return**;  **int** leftCount = countDescendants(node->left);  **int** rightCount = countDescendants(node->right);  **if** (leftCount != rightCount) {  result.push\_back(node->key);  }  findNodesWithUnequalDescendants(node->left, result);  findNodesWithUnequalDescendants(node->right, result);  }  **int** **calculateHeight**(Node\* node) {  **if** (!node) **return** **0**;  **return** **1** + std::max(calculateHeight(node->left), calculateHeight(node->right));  }  **void** **findNodesWithUnequalHeights**(Node\* node, std::vector<**int**>& result) {  **if** (!node) **return**;  **int** leftHeight = calculateHeight(node->left);  **int** rightHeight = calculateHeight(node->right);  **if** (leftHeight != rightHeight) {  result.push\_back(node->key);  }  findNodesWithUnequalHeights(node->left, result);  findNodesWithUnequalHeights(node->right, result);  }  **int** **countOccurrences**(Node\* node, **int** x) {  **if** (!node) **return** **0**;  **int** count = (node->key == x) ? **1** : **0**;  **return** count + countOccurrences(node->left, x) + countOccurrences(node->right, x);  }  **void** **findMaxElement**(Node\* node, **int**& maxElement, **int**& count) {  **if** (!node) **return**;  **if** (node->key > maxElement) {  maxElement = node->key;  count = **1**;  }  **else** **if** (node->key == maxElement) {  count++;  }  findMaxElement(node->left, maxElement, count);  findMaxElement(node->right, maxElement, count);  }  **bool** **hasDuplicates**(Node\* node, std::set<**int**>& seen) {  **if** (!node) **return** false;  **if** (seen.count(node->key)) **return** true;  seen.insert(node->key);  **return** hasDuplicates(node->left, seen) || hasDuplicates(node->right, seen);  }  **void** **countFrequencies**(Node\* node, std::map<**int**, **int**>& freq) {  **if** (!node) **return**;  freq[node->key]++;  countFrequencies(node->left, freq);  countFrequencies(node->right, freq);  }  **bool** **isMirror**(Node\* left, Node\* right) {  **if** (!left && !right) **return** true;  **if** (!left || !right) **return** false;  **return** left->key == right->key &&  isMirror(left->left, right->right) &&  isMirror(left->right, right->left);  }  **bool** **isSymmetric**(Node\* root) {  **if** (!root) **return** true;  **return** isMirror(root->left, root->right);  }  **bool** **isBST**(Node\* node, **int** minVal = INT\_MIN, **int** maxVal = INT\_MAX) {  **if** (!node) **return** true;  **if** (node->key <= minVal || node->key >= maxVal) **return** false;  **return** isBST(node->left, minVal, node->key) &&  isBST(node->right, node->key, maxVal);  }  **void** **collectLeaves**(Node\* node, std::vector<**int**>& leaves) {  **if** (!node) **return**;  **if** (!node->left && !node->right) {  leaves.push\_back(node->key);  }  collectLeaves(node->left, leaves);  collectLeaves(node->right, leaves);  }  **void** **findPaths**(Node\* node, std::vector<**int**>& currentPath, std::vector<std::vector<**int**>>& paths) {  **if** (!node) **return**;  currentPath.push\_back(node->key);  **if** (!node->left && !node->right) {  paths.push\_back(currentPath);  }  **else** {  findPaths(node->left, currentPath, paths);  findPaths(node->right, currentPath, paths);  }  currentPath.pop\_back();  }  std::vector<**int**> findMinWeightPath(Node\* root) {  **if** (!root) **return** std::vector<**int**>();  std::vector<std::vector<**int**>> paths;  std::vector<**int**> currentPath;  findPaths(root, currentPath, paths);  std::vector<**int**> minWeightPath;  **int** minWeight = std::numeric\_limits<**int**>::max();  **for** (**const** **auto**& path : paths) {  **int** weight = std::accumulate(path.begin(), path.end(), **0**);  **if** (weight < minWeight) {  minWeight = weight;  minWeightPath = path;  }  }  **return** minWeightPath;  }  **int** findLastLevelWithPositive(Node\* root) {  **if** (!root) **return** -**1**;  std::queue<std::pair<Node\*, **int**>> q;  q.push({ root, **0** });  **int** lastPositiveLevel = -**1**;  **while** (!q.empty()) {  Node\* current = q.front().first;  **int** level = q.front().second;  q.pop();  **if** (current->key > **0**) lastPositiveLevel = level;  **if** (current->left) q.push({ current->left, level + **1** });  **if** (current->right) q.push({ current->right, level + **1** });  }  **return** lastPositiveLevel;  }  **void** findMaxOnEachLevel(Node\* root, std::vector<**int**>& maxElements) {  **if** (!root) **return**;  std::queue<std::pair<Node\*, **int**>> q;  q.push({ root, **0** });  **while** (!q.empty()) {  Node\* current = q.front().first;  **int** level = q.front().second;  q.pop();  **if** (level >= maxElements.size()) {  maxElements.push\_back(current->key);  }  **else** {  maxElements[level] = std::max(maxElements[level], current->key);  }  **if** (current->left) q.push({ current->left, level + **1** });  **if** (current->right) q.push({ current->right, level + **1** });  }  }  **void** findNodesAndLeavesOnEachLevel(Node\* root, std::vector<**int**>& internalCounts, std::vector<**int**>& leafCounts) {  **if** (!root) **return**;  std::queue<std::pair<Node\*, **int**>> q;  q.push({ root, **0** });  **while** (!q.empty()) {  Node\* current = q.front().first;  **int** level = q.front().second;  q.pop();  **if** (level >= internalCounts.size()) {  internalCounts.push\_back(**0**);  leafCounts.push\_back(**0**);  }  **if** (!current->left && !current->right) {  leafCounts[level]++;  }  **else** {  internalCounts[level]++;  }  **if** (current->left) q.push({ current->left, level + **1** });  **if** (current->right) q.push({ current->right, level + **1** });  }  }  **int** sumOfOddLevels(Node\* root) {  **if** (!root) **return** **0**;  std::queue<std::pair<Node\*, **int**>> q;  q.push({ root, **0** });  **int** sum = **0**;  **while** (!q.empty()) {  Node\* current = q.front().first;  **int** level = q.front().second;  q.pop();  **if** (level % **2** == **1**) {  sum += current->key;  }  **if** (current->left) q.push({ current->left, level + **1** });  **if** (current->right) q.push({ current->right, level + **1** });  }  **return** sum;  }  **void** findMinMaxPaths(Node\* node, **int** currentLength, **int**& minPath, **int**& maxPath) {  **if** (!node) **return**;  **if** (!node->left && !node->right) {  minPath = std::min(minPath, currentLength);  maxPath = std::max(maxPath, currentLength);  }  **else** {  findMinMaxPaths(node->left, currentLength + **1**, minPath, maxPath);  findMinMaxPaths(node->right, currentLength + **1**, minPath, maxPath);  }  }  **void** collectElements(Node\* root, std::vector<**int**>& elements) {  **if** (!root) **return**;  collectElements(root->left, elements);  elements.push\_back(root->key);  collectElements(root->right, elements);  }  Node\* buildBalancedTree(**const** std::vector<**int**>& elements, **int** start, **int** end) {  **if** (start > end) **return** nullptr;  **int** mid = start + (end - start) / **2**;  Node\* root = createNode(elements[mid]);  root->left = buildBalancedTree(elements, start, mid - **1**);  root->right = buildBalancedTree(elements, mid + **1**, end);  **return** root;  }  **void** createBalancedTrees(Node\* root, Node\*& negativeTree, Node\*& nonNegativeTree) {  std::vector<**int**> allElements;  std::vector<**int**> negativeElements;  std::vector<**int**> nonNegativeElements;  collectElements(root, allElements);  **for** (**int** element : allElements) {  **if** (element < **0**)  negativeElements.push\_back(element);  **else**  nonNegativeElements.push\_back(element);  }  std::sort(negativeElements.begin(), negativeElements.end());  std::sort(nonNegativeElements.begin(), nonNegativeElements.end());  negativeTree = buildBalancedTree(negativeElements, **0**, negativeElements.size() - **1**);  nonNegativeTree = buildBalancedTree(nonNegativeElements, **0**, nonNegativeElements.size() - **1**);  }  Node\* removeNode(Node\* node) {  **if** (!node) **return** nullptr;  **if** (!node->left && !node->right) {  **delete** node;  **return** nullptr;  }  **if** (!node->left) {  Node\* temp = node->right;  **delete** node;  **return** temp;  }  **if** (!node->right) {  Node\* temp = node->left;  **delete** node;  **return** temp;  }  **return** node;  }  Node\* transformToStrictlyBinary(Node\* root) {  **if** (!root) **return** nullptr;  root->left = transformToStrictlyBinary(root->left);  root->right = transformToStrictlyBinary(root->right);  **if** ((root->left && root->right) || (!root->left && !root->right)) {  **return** root;  }  **return** removeNode(root);  }  Node\* copyTree(Node\* root) {  **if** (!root) **return** nullptr;  Node\* newRoot = createNode(root->key);  newRoot->left = copyTree(root->left);  newRoot->right = copyTree(root->right);  **return** newRoot;  }  **void** inOrderTraversal(Node\* root, std::wostringstream& oss) {  **if** (!root) **return**;  inOrderTraversal(root->left, oss);  oss << root->key << " ";  inOrderTraversal(root->right, oss);  }  **void** postOrderTraversal(Node\* root, std::wostringstream& oss) {  **if** (!root) **return**;  postOrderTraversal(root->left, oss);  postOrderTraversal(root->right, oss);  oss << root->key << " ";  }  **void** preOrderTraversal(Node\* root, std::wostringstream& oss) {  **if** (!root) **return**;  oss << root->key << " ";  preOrderTraversal(root->left, oss);  preOrderTraversal(root->right, oss);  }  **struct** BPlusNode {  std::vector<**int**> keys;  std::vector<**void**\*> pointers;  std::vector<BPlusNode\*> children;  BPlusNode\* parent = nullptr;  BPlusNode\* left = nullptr;  BPlusNode\* right = nullptr;  **bool** isLeaf = false;  **int** keyCount = **0**;  };  **struct** BPlusTree {  BPlusNode\* root = nullptr;  **int** t;  };  **void** **splitBPlusNode**(BPlusTree& tree, BPlusNode\* bPlusNode) {  **int** t = tree.t;  **int** maxKeys = **2** \* t - **1**;  **int** midIndex = t - **1**;  BPlusNode\* newNode = **new** BPlusNode();  newNode->isLeaf = bPlusNode->isLeaf;  newNode->parent = bPlusNode->parent;  **if** (bPlusNode->isLeaf) {  newNode->keys.assign(bPlusNode->keys.begin() + t, bPlusNode->keys.end());  newNode->pointers.assign(bPlusNode->pointers.begin() + t, bPlusNode->pointers.end());  bPlusNode->keys.resize(t);  bPlusNode->pointers.resize(t);  newNode->right = bPlusNode->right;  **if** (newNode->right) {  newNode->right->left = newNode;  }  bPlusNode->right = newNode;  newNode->left = bPlusNode;  **int** midKey = newNode->keys[**0**];  **if** (bPlusNode == tree.root) {  tree.root = **new** BPlusNode();  tree.root->keys.push\_back(midKey);  tree.root->children.push\_back(bPlusNode);  tree.root->children.push\_back(newNode);  tree.root->isLeaf = false;  bPlusNode->parent = tree.root;  newNode->parent = tree.root;  }  **else** {  BPlusNode\* parentNode = bPlusNode->parent;  **auto** it = std::upper\_bound(parentNode->keys.begin(), parentNode->keys.end(), midKey);  **int** pos = it - parentNode->keys.begin();  parentNode->keys.insert(it, midKey);  parentNode->children.insert(parentNode->children.begin() + pos + **1**, newNode);  newNode->parent = parentNode;  **if** (parentNode->keys.size() > maxKeys) {  splitBPlusNode(tree, parentNode);  }  }  }  **else** {  **int** midKey = bPlusNode->keys[midIndex];  newNode->keys.assign(bPlusNode->keys.begin() + midIndex + **1**, bPlusNode->keys.end());  newNode->children.assign(bPlusNode->children.begin() + midIndex + **1**, bPlusNode->children.end());  **for** (**auto** child : newNode->children) {  **if** (child) {  child->parent = newNode;  }  }  bPlusNode->keys.resize(midIndex);  bPlusNode->children.resize(midIndex + **1**);  **if** (bPlusNode == tree.root) {  tree.root = **new** BPlusNode();  tree.root->keys.push\_back(midKey);  tree.root->children.push\_back(bPlusNode);  tree.root->children.push\_back(newNode);  tree.root->isLeaf = false;  bPlusNode->parent = tree.root;  newNode->parent = tree.root;  }  **else** {  BPlusNode\* parentNode = bPlusNode->parent;  **auto** it = std::upper\_bound(parentNode->keys.begin(), parentNode->keys.end(), midKey);  **int** pos = it - parentNode->keys.begin();  parentNode->keys.insert(it, midKey);  parentNode->children.insert(parentNode->children.begin() + pos + **1**, newNode);  newNode->parent = parentNode;  **if** (parentNode->keys.size() > maxKeys) {  splitBPlusNode(tree, parentNode);  }  }  }  }  BPlusNode\* **findLeaf**(BPlusTree& tree, **int** key) {  BPlusNode\* current = tree.root;  **if** (!current) {  **return** nullptr;  }  **while** (!current->isLeaf) {  **int** i = **0**;  **while** (i < current->keyCount && key >= current->keys[i]) {  ++i;  }  **if** (i < current->children.size()) {  current = current->children[i];  }  **else** {  **return** nullptr;  }  }  **return** current;  }  **void** **insertKey**(BPlusTree& tree, **int** key, **void**\* value) {  **if** (tree.root == nullptr) {  tree.root = **new** BPlusNode();  tree.root->isLeaf = true;  }  BPlusNode\* leaf = findLeaf(tree, key);  **if** (!leaf) **return**;  **auto** it = std::upper\_bound(leaf->keys.begin(), leaf->keys.end(), key);  **int** pos = it - leaf->keys.begin();  leaf->keys.insert(it, key);  leaf->pointers.insert(leaf->pointers.begin() + pos, value);  leaf->keyCount = leaf->keys.size();  **int** maxKeys = **2** \* tree.t - **1**;  **if** (leaf->keys.size() > maxKeys) {  splitBPlusNode(tree, leaf);  }  }  **void** **insertBinaryTreeKeysToBPlus**(Node\* root, BPlusTree& tree) {  **if** (root == nullptr) **return**;  insertKey(tree, root->key, nullptr);  insertBinaryTreeKeysToBPlus(root->left, tree);  insertBinaryTreeKeysToBPlus(root->right, tree);  }  **void** **drawNodeBPlusTree**(sf::RenderWindow& window, sf::Font& font, BPlusNode\* node, **float** x, **float** y, **float** offsetX, **float** offsetY) {  sf::RectangleShape rect;  rect.setSize(sf::Vector2f(**70** \* node->keys.size(), **40**));  rect.setOutlineThickness(**2**);  rect.setOutlineColor(sf::Color::Green);  rect.setFillColor(sf::Color::White);  rect.setPosition(x - rect.getSize().x / **2**, y);  window.draw(rect);  **for** (**size\_t** i = **0**; i < node->keys.size(); i++) {  sf::Text text(std::to\_string(node->keys[i]), font, **16**);  text.setFillColor(sf::Color::Black);  text.setPosition(x - rect.getSize().x / **2** + i \* **70** + **5**, y + **5**);  window.draw(text);  }  **if** (!node->isLeaf) {  **for** (**size\_t** i = **0**; i < node->children.size(); i++) {  **if** (node->children[i]) {  **float** childX = x - offsetX / **2** + i \* (offsetX / (node->children.size() - **1**));  sf::Vertex line[] = {  sf::Vertex(sf::Vector2f(x, y + **40**), sf::Color::Green),  sf::Vertex(sf::Vector2f(childX, y + offsetY), sf::Color::Green)  };  window.draw(line, **2**, sf::Lines);  drawNodeBPlusTree(window, font, node->children[i], childX, y + offsetY, offsetX / **2**, offsetY);  }  }  }  }  **void** **drawBPlusTree**(sf::RenderWindow& window, BPlusNode\* root, **float** x, **float** y, **float** offsetX, **float** offsetY) {  **if** (!root) **return**;  sf::Font font;  **if** (!font.loadFromFile("C:/Windows/Fonts/times.ttf")) {  std::cerr << "Ошибка загрузки шрифта" << std::endl;  **return**;  }  drawNodeBPlusTree(window, font, root, x, y, offsetX, offsetY);  }  **struct** RedBlackNode {  **int** data;  std::string color;  RedBlackNode\* left;  RedBlackNode\* right;  RedBlackNode\* parent;  RedBlackNode(**int** data = **0**)  : data(data), color("RED"), left(nullptr), right(nullptr), parent(nullptr) {  }  };  **struct** RedBlackTree {  RedBlackNode\* root;  RedBlackNode\* NIL;  };  **void** **initializeRedBlackTree**(RedBlackTree& rbt) {  rbt.NIL = **new** RedBlackNode();  rbt.NIL->color = "BLACK";  rbt.NIL->left = rbt.NIL->right = rbt.NIL;  rbt.root = rbt.NIL;  }  **void** **leftRotate**(RedBlackTree& rbt, RedBlackNode\* x) {  RedBlackNode\* y = x->right;  x->right = y->left;  **if** (y->left != rbt.NIL) {  y->left->parent = x;  }  y->parent = x->parent;  **if** (x->parent == nullptr) {  rbt.root = y;  }  **else** **if** (x == x->parent->left) {  x->parent->left = y;  }  **else** {  x->parent->right = y;  }  y->left = x;  x->parent = y;  }  **void** **rightRotate**(RedBlackTree& rbt, RedBlackNode\* x) {  RedBlackNode\* y = x->left;  x->left = y->right;  **if** (y->right != rbt.NIL) {  y->right->parent = x;  }  y->parent = x->parent;  **if** (x->parent == nullptr) {  rbt.root = y;  }  **else** **if** (x == x->parent->right) {  x->parent->right = y;  }  **else** {  x->parent->left = y;  }  y->right = x;  x->parent = y;  }  **void** **fixInsert**(RedBlackTree& rbt, RedBlackNode\* k) {  **while** (k->parent && k->parent->color == "RED") {  **if** (k->parent == k->parent->parent->left) {  RedBlackNode\* uncle = k->parent->parent->right;  **if** (uncle->color == "RED") {  k->parent->color = "BLACK";  uncle->color = "BLACK";  k->parent->parent->color = "RED";  k = k->parent->parent;  }  **else** {  **if** (k == k->parent->right) {  k = k->parent;  leftRotate(rbt, k);  }  k->parent->color = "BLACK";  k->parent->parent->color = "RED";  rightRotate(rbt, k->parent->parent);  }  }  **else** {  RedBlackNode\* uncle = k->parent->parent->left;  **if** (uncle->color == "RED") {  k->parent->color = "BLACK";  uncle->color = "BLACK";  k->parent->parent->color = "RED";  k = k->parent->parent;  }  **else** {  **if** (k == k->parent->left) {  k = k->parent;  rightRotate(rbt, k);  }  k->parent->color = "BLACK";  k->parent->parent->color = "RED";  leftRotate(rbt, k->parent->parent);  }  }  }  rbt.root->color = "BLACK";  }  **void** **insert**(RedBlackTree& rbt, **int** data) {  RedBlackNode\* newNode = **new** RedBlackNode(data);  newNode->left = rbt.NIL;  newNode->right = rbt.NIL;  RedBlackNode\* parent = nullptr;  RedBlackNode\* current = rbt.root;  **while** (current != rbt.NIL) {  parent = current;  **if** (newNode->data < current->data) {  current = current->left;  }  **else** {  current = current->right;  }  }  newNode->parent = parent;  **if** (parent == nullptr) {  rbt.root = newNode;  }  **else** **if** (newNode->data < parent->data) {  parent->left = newNode;  }  **else** {  parent->right = newNode;  }  **if** (newNode->parent == nullptr) {  newNode->color = "BLACK";  **return**;  }  **if** (newNode->parent->parent == nullptr) {  **return**;  }  fixInsert(rbt, newNode);  }  **void** **inorderHelper**(RedBlackNode\* node, RedBlackNode\* NIL) {  **if** (node != NIL) {  inorderHelper(node->left, NIL);  std::cout << node->data << " ";  inorderHelper(node->right, NIL);  }  }  **void** **inorderTraversal**(**const** RedBlackTree& rbt) {  inorderHelper(rbt.root, rbt.NIL);  }  RedBlackNode\* **search**(RedBlackNode\* node, RedBlackNode\* NIL, **int** data) {  **if** (node == NIL || data == node->data) {  **return** node;  }  **if** (data < node->data) {  **return** search(node->left, NIL, data);  }  **return** search(node->right, NIL, data);  }  RedBlackNode\* **searchRedBlackTree**(**const** RedBlackTree& rbt, **int** data) {  **return** search(rbt.root, rbt.NIL, data);  }  **void** **insertBPlusTreeKeysToRBT**(BPlusNode\* node, RedBlackTree& rbt) {  **if** (!node) **return**;  **if** (node->isLeaf) {  **for** (**int** key : node->keys) {  insert(rbt, key);  }  **return**;  }  **for** (BPlusNode\* child : node->children) {  insertBPlusTreeKeysToRBT(child, rbt);  }  }  **void** **drawNodeRedBlackTree**(sf::RenderWindow& window, sf::Font& font, RedBlackNode\* node, RedBlackNode\* NIL, **float** x, **float** y, **float** offsetX, **float** offsetY) {  **if** (node == nullptr || node == NIL) {  **return**;  }  **if** (node->left != NIL) {  sf::Vertex line[] = {  sf::Vertex(sf::Vector2f(x, y), sf::Color::Black),  sf::Vertex(sf::Vector2f(x - offsetX, y + offsetY), sf::Color::Black)  };  window.draw(line, **2**, sf::Lines);  drawNodeRedBlackTree(window, font, node->left, NIL, x - offsetX, y + offsetY, offsetX / **2**, offsetY);  }  **if** (node->right != NIL) {  sf::Vertex line[] = {  sf::Vertex(sf::Vector2f(x, y), sf::Color::Black),  sf::Vertex(sf::Vector2f(x + offsetX, y + offsetY), sf::Color::Black)  };  window.draw(line, **2**, sf::Lines);  drawNodeRedBlackTree(window, font, node->right, NIL, x + offsetX, y + offsetY, offsetX / **2**, offsetY);  }  sf::CircleShape circle(**30**);  circle.setFillColor(node->color == "RED" ? sf::Color::Red : sf::Color::Black);  circle.setOutlineThickness(**2**);  circle.setOutlineColor(sf::Color::White);  circle.setPosition(x - **30**, y - **30**);  // Draw the node value as text  sf::Text text;  text.setFont(font);  text.setString(std::to\_string(node->data));  text.setCharacterSize(**16**);  text.setFillColor(sf::Color::White);  text.setPosition(x - **15**, y - **15**);  window.draw(circle);  window.draw(text);  }  **void** **drawRedBlackTree**(sf::RenderWindow& window, **const** RedBlackTree& rbt, **float** x, **float** y, **float** offsetX, **float** offsetY) {  sf::Font font;  **if** (!font.loadFromFile("C:/Windows/Fonts/times.ttf")) {  std::cerr << "Error loading font!" << std::endl;  **return**;  }  drawNodeRedBlackTree(window, font, rbt.root, rbt.NIL, x, y, offsetX, offsetY);  }  **void** **drawRedBlackTreeWindow**(BPlusNode\* root) {  sf::RenderWindow RedBlackTreeWindow(sf::VideoMode(**1920**, **1080**), L"Лабораторная работа №5 - Красно-чёрное дерево");  sf::Font font;  **if** (!font.loadFromFile("C:/Windows/Fonts/times.ttf")) {  std::cerr << "Ошибка загрузки шрифта" << std::endl;  **return**;  }  // Создаем кнопку  sf::RectangleShape button2(sf::Vector2f(**200**, **100**));  button2.setPosition(**1700**, **900**);  button2.setFillColor(sf::Color::Red);  sf::Text buttonText2("Exit", font, **30**);  buttonText2.setFillColor(sf::Color::Black);  buttonText2.setPosition(**1770**, **930**);  RedBlackTree rbt;  initializeRedBlackTree(rbt);  insertBPlusTreeKeysToRBT(root, rbt);  **while** (RedBlackTreeWindow.isOpen()) {  sf::Event event;  **while** (RedBlackTreeWindow.pollEvent(event)) {  **if** (event.type == sf::Event::Closed)  RedBlackTreeWindow.close();  // Обработка клика на кнопку  **if** (event.type == sf::Event::MouseButtonPressed && event.mouseButton.button == sf::Mouse::Left) {  sf::Vector2i mousePos = sf::Mouse::getPosition(RedBlackTreeWindow);  **if** (button2.getGlobalBounds().contains(**static\_cast**<sf::Vector2f>(mousePos))) {  // Переход на следующее окно  RedBlackTreeWindow.close();  exit(**0**);  }  }  }  RedBlackTreeWindow.clear(sf::Color(**128**, **200**, **255**, **128**));  drawRedBlackTree(RedBlackTreeWindow, rbt, **800**, **50**, **400**, **100**);  RedBlackTreeWindow.draw(button2);  RedBlackTreeWindow.draw(buttonText2);  RedBlackTreeWindow.display();  }  }  **void** **drawBPlusTreeWindow**(Node\* root) {  sf::RenderWindow bTreeWindow(sf::VideoMode(**1920**, **1080**), L"Лабораторная работа №5 - Дерево B+");  sf::Font font;  **if** (!font.loadFromFile("C:/Windows/Fonts/times.ttf")) {  std::cerr << "Ошибка загрузки шрифта" << std::endl;  **return**;  }  // Создаем кнопку  sf::RectangleShape button2(sf::Vector2f(**200**, **100**));  button2.setPosition(**1700**, **900**);  button2.setFillColor(sf::Color::Green);  sf::Text buttonText2("Next", font, **30**);  buttonText2.setFillColor(sf::Color::White);  buttonText2.setPosition(**1770**, **930**);  BPlusTree tree;  tree.t = **8**;  insertBinaryTreeKeysToBPlus(root, tree);  **while** (bTreeWindow.isOpen()) {  sf::Event event;  **while** (bTreeWindow.pollEvent(event)) {  **if** (event.type == sf::Event::Closed)  bTreeWindow.close();  **if** (event.type == sf::Event::KeyPressed && event.key.code == sf::Keyboard::Enter) {  drawRedBlackTreeWindow(tree.root);  }  // Обработка клика на кнопку  **if** (event.type == sf::Event::MouseButtonPressed && event.mouseButton.button == sf::Mouse::Left) {  sf::Vector2i mousePos = sf::Mouse::getPosition(bTreeWindow);  **if** (button2.getGlobalBounds().contains(**static\_cast**<sf::Vector2f>(mousePos))) {  // Переход на следующее окно  drawRedBlackTreeWindow(tree.root);  }  }  }  bTreeWindow.clear(sf::Color(**128**, **200**, **255**, **128**));  drawBPlusTree(bTreeWindow, tree.root, **900**, **50**, **400**, **100**);  bTreeWindow.draw(button2);  bTreeWindow.draw(buttonText2);  bTreeWindow.display();  }  **return**;  }  **void** **drawTree**(sf::RenderWindow& window, Node\* node, **float** x, **float** y, **float** xOffset) {  **if** (!node) **return**;  sf::Color lightBlue(**206**, **225**, **242**);  sf::CircleShape circle(**15**);  circle.setFillColor(lightBlue);  circle.setPosition(x, y);  sf::Font font;  **if** (!font.loadFromFile("C:/Windows/Fonts/times.ttf")) {  std::cerr << "Ошибка загрузки шрифта" << std::endl;  **return**;  }  sf::Color blackBlue(**37**, **53**, **66**);  sf::Text text;  text.setFont(font);  text.setString(std::to\_string(node->key));  text.setCharacterSize(**12**);  text.setFillColor(blackBlue);  text.setPosition(x + **5**, y + **5**);  **if** (node->left) {  sf::Vertex line[] = {  sf::Vertex(sf::Vector2f(x + **15**, y + **15**), blackBlue),  sf::Vertex(sf::Vector2f(x - xOffset + **10**, y + **65**), blackBlue)  };  window.draw(line, **2**, sf::Lines);  drawTree(window, node->left, x - xOffset, y + **50**, xOffset / **1.728**);  }  **if** (node->right) {  sf::Vertex line[] = {  sf::Vertex(sf::Vector2f(x + **15**, y + **15**), blackBlue),  sf::Vertex(sf::Vector2f(x + xOffset + **25**, y + **70**), blackBlue),  };  window.draw(line, **2**, sf::Lines);  drawTree(window, node->right, x + xOffset, y + **50**, xOffset / **1.728**);  }  window.draw(circle);  window.draw(text);  }  **void** **drawRemoveMinVertexWindow**(Node\* root) {  sf::RenderWindow removeMinVertexWindow(sf::VideoMode(**1920**, **1080**), L"Лабораторная работа №5 - Строго бинарное дерево с наименьшим удаленными вершинами");  sf::Font font;  **if** (!font.loadFromFile("C:/Windows/Fonts/times.ttf")) {  std::cerr << "Ошибка загрузки шрифта" << std::endl;  **return**;  }  // Создаем кнопку  sf::RectangleShape button2(sf::Vector2f(**200**, **100**));  button2.setPosition(**1700**, **900**); // Позиция кнопки  button2.setFillColor(sf::Color::Green);  sf::Text buttonText2("Next", font, **30**);  buttonText2.setFillColor(sf::Color::White);  buttonText2.setPosition(**1770**, **930**); // Позиция текста (по центру кнопки)  Node\* transformedTree = copyTree(root);  transformedTree = transformToStrictlyBinary(transformedTree);  transformToStrictlyBinary(transformedTree);  **while** (removeMinVertexWindow.isOpen()) {  sf::Event event;  **while** (removeMinVertexWindow.pollEvent(event)) {  **if** (event.type == sf::Event::Closed)  removeMinVertexWindow.close();  **if** (event.type == sf::Event::KeyPressed && event.key.code == sf::Keyboard::Enter) {  drawBPlusTreeWindow(root);  }  // Обработка клика на кнопку  **if** (event.type == sf::Event::MouseButtonPressed && event.mouseButton.button == sf::Mouse::Left) {  sf::Vector2i mousePos = sf::Mouse::getPosition(removeMinVertexWindow);  **if** (button2.getGlobalBounds().contains(**static\_cast**<sf::Vector2f>(mousePos))) {  // Переход на следующее окно  drawBPlusTreeWindow(root);  }  }  }  removeMinVertexWindow.clear(sf::Color(**128**, **200**, **255**, **128**));  drawTree(removeMinVertexWindow, transformedTree, **900**, **50**, **400**);  removeMinVertexWindow.draw(button2);  removeMinVertexWindow.draw(buttonText2);  removeMinVertexWindow.display();  }  **return**;  }  **void** **drawBalanceTreeWindow**(Node\* root) {  sf::RenderWindow balanceTreeWindow(sf::VideoMode(**1920**, **1080**), L"Лабораторная работа №5 - Сбалансированные деревья с положительными и отрицательными элементами");  Node\* negativeTree = nullptr;  Node\* nonNegativeTree = nullptr;  createBalancedTrees(root, negativeTree, nonNegativeTree);  sf::Font font;  **if** (!font.loadFromFile("C:/Windows/Fonts/times.ttf")) {  std::cerr << "Ошибка загрузки шрифта" << std::endl;  **return**;  }  // Создаем кнопку  sf::RectangleShape button2(sf::Vector2f(**200**, **100**));  button2.setPosition(**1700**, **900**); // Позиция кнопки  button2.setFillColor(sf::Color::Green);  sf::Text buttonText2("Next", font, **30**);  buttonText2.setFillColor(sf::Color::White);  buttonText2.setPosition(**1770**, **930**); // Позиция текста (по центру кнопки)  **while** (balanceTreeWindow.isOpen()) {  sf::Event event;  **while** (balanceTreeWindow.pollEvent(event)) {  **if** (event.type == sf::Event::Closed)  balanceTreeWindow.close();  **if** (event.type == sf::Event::KeyPressed && event.key.code == sf::Keyboard::Enter) {  drawRemoveMinVertexWindow(root);  }  // Обработка клика на кнопку  **if** (event.type == sf::Event::MouseButtonPressed && event.mouseButton.button == sf::Mouse::Left) {  sf::Vector2i mousePos = sf::Mouse::getPosition(balanceTreeWindow);  **if** (button2.getGlobalBounds().contains(**static\_cast**<sf::Vector2f>(mousePos))) {  // Переход на следующее окно  drawRemoveMinVertexWindow(root);  }  }  }  balanceTreeWindow.clear(sf::Color(**128**, **200**, **255**, **128**));  drawTree(balanceTreeWindow, negativeTree, **600**, **50**, **200**);  drawTree(balanceTreeWindow, nonNegativeTree, **600**, **400**, **200**);  balanceTreeWindow.draw(button2);  balanceTreeWindow.draw(buttonText2);  balanceTreeWindow.display();  }  **return**;  }  **void** **drawResultsWindow**(Node\* root) {  sf::RenderWindow resultsWindow(sf::VideoMode(**1920**, **1080**), L"Лабораторная работа №5 - Окно результатов");  **int** sum = sumElements(root);  **int64\_t** product = productOfMultiplesOfThree(root);  std::vector<**int**> nodesWithUnequalDescendants;  findNodesWithUnequalDescendants(root, nodesWithUnequalDescendants);  std::vector<**int**> nodesWithUnequalHeights;  findNodesWithUnequalHeights(root, nodesWithUnequalHeights);  **int** x = **15**;  **int** occurrencesOfX = countOccurrences(root, x);  **int** maxElement = INT\_MIN, maxCount = **0**;  findMaxElement(root, maxElement, maxCount);  std::set<**int**> seen;  **bool** duplicatesExist = hasDuplicates(root, seen);  std::map<**int**, **int**> frequencies;  countFrequencies(root, frequencies);  **int** maxFrequency = **0**;  **for** (**const** **auto**& pair : frequencies) {  maxFrequency = std::max(maxFrequency, pair.second);  }  **bool** symmetric = isSymmetric(root);  **bool** bst = isBST(root);  std::vector<**int**> leaves;  collectLeaves(root, leaves);  std::sort(leaves.begin(), leaves.end());  std::vector<**int**> minWeightPath = findMinWeightPath(root);  **int** lastPositiveLevel = findLastLevelWithPositive(root);  std::vector<**int**> maxOnLevels;  findMaxOnEachLevel(root, maxOnLevels);  std::vector<**int**> internalCounts, leafCounts;  findNodesAndLeavesOnEachLevel(root, internalCounts, leafCounts);  **int** oddLevelSum = sumOfOddLevels(root);  **int** minPath = INT\_MAX, maxPath = INT\_MIN;  findMinMaxPaths(root, **0**, minPath, maxPath);  std::wostringstream inOrder, postOrder, preOrder;  inOrderTraversal(root, inOrder);  postOrderTraversal(root, postOrder);  preOrderTraversal(root, preOrder);  sf::Font font;  **if** (!font.loadFromFile("C:/Windows/Fonts/times.ttf")) {  std::cerr << "Ошибка загрузки шрифта" << std::endl;  **return**;  }  // Создаем кнопку  sf::RectangleShape button2(sf::Vector2f(**200**, **100**));  button2.setPosition(**1700**, **900**); // Позиция кнопки  button2.setFillColor(sf::Color::Green);  sf::Text buttonText2("Next", font, **30**);  buttonText2.setFillColor(sf::Color::White);  buttonText2.setPosition(**1770**, **930**); // Позиция текста (по центру кнопки)  **int** yOffset = **10**;  **const** **int** lineSpacing = **30**;  sf::Text sumText;  sumText.setFont(font);  sumText.setString(L"2) Сумма всех элементов дерева: " + std::to\_wstring(sum));  sumText.setCharacterSize(**18**);  sumText.setFillColor(sf::Color::Black);  sumText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  sf::Text productText;  productText.setFont(font);  productText.setString(L"3) Произведение элементов, кратных 3: " + std::to\_wstring(product));  productText.setCharacterSize(**18**);  productText.setFillColor(sf::Color::Black);  productText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  std::wstring unequalDescendantsInfo = L"4) Вершины с неравным числом потомков: ";  **for** (**int** key : nodesWithUnequalDescendants) {  unequalDescendantsInfo += std::to\_wstring(key) + L" ";  }  sf::Text unequalDescendantsText;  unequalDescendantsText.setFont(font);  unequalDescendantsText.setString(unequalDescendantsInfo);  unequalDescendantsText.setCharacterSize(**18**);  unequalDescendantsText.setFillColor(sf::Color::Black);  unequalDescendantsText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  std::wstring unequalHeightsInfo = L"5) Вершины с неравной высотой поддеревьев: ";  **for** (**int** key : nodesWithUnequalHeights) {  unequalHeightsInfo += std::to\_wstring(key) + L" ";  }  sf::Text unequalHeightsText;  unequalHeightsText.setFont(font);  unequalHeightsText.setString(unequalHeightsInfo);  unequalHeightsText.setCharacterSize(**18**);  unequalHeightsText.setFillColor(sf::Color::Black);  unequalHeightsText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  sf::Text occurrencesText;  occurrencesText.setFont(font);  occurrencesText.setString(L"6) Количество вхождений элемента " + std::to\_wstring(x) + L": " + std::to\_wstring(occurrencesOfX));  occurrencesText.setCharacterSize(**18**);  occurrencesText.setFillColor(sf::Color::Black);  occurrencesText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  sf::Text maxElementText;  maxElementText.setFont(font);  maxElementText.setString(L"7) Максимальный элемент: " + std::to\_wstring(maxElement) +  L", количество: " + std::to\_wstring(maxCount));  maxElementText.setCharacterSize(**18**);  maxElementText.setFillColor(sf::Color::Black);  maxElementText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  sf::Text duplicatesText;  duplicatesText.setFont(font);  duplicatesText.setString(L"8) Есть ли дубликаты: " + std::wstring(duplicatesExist ? L"Да" : L"Нет"));  duplicatesText.setCharacterSize(**18**);  duplicatesText.setFillColor(sf::Color::Black);  duplicatesText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  sf::Text maxFreqText;  maxFreqText.setFont(font);  maxFreqText.setString(L"9) Максимальное количество одинаковых элементов: " + std::to\_wstring(maxFrequency));  maxFreqText.setCharacterSize(**18**);  maxFreqText.setFillColor(sf::Color::Black);  maxFreqText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  sf::Text symmetricText;  symmetricText.setFont(font);  symmetricText.setString(L"10) Симметрично ли дерево: " + std::wstring(symmetric ? L"Да" : L"Нет"));  symmetricText.setCharacterSize(**18**);  symmetricText.setFillColor(sf::Color::Black);  symmetricText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  sf::Text bstText;  bstText.setFont(font);  bstText.setString(L"11) Является ли деревом поиска: " + std::wstring(bst ? L"Да" : L"Нет"));  bstText.setCharacterSize(**18**);  bstText.setFillColor(sf::Color::Black);  bstText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  std::wstring leavesInfo = L"12) Листья дерева поиска: ";  **for** (**int** leaf : leaves) {  leavesInfo += std::to\_wstring(leaf) + L" ";  }  sf::Text leavesText;  leavesText.setFont(font);  leavesText.setString(leavesInfo);  leavesText.setCharacterSize(**18**);  leavesText.setFillColor(sf::Color::Black);  leavesText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  std::wstring minPathStr = L"14) Минимальный путь от корня к листьям (вес = ";  **int** minWeight = std::accumulate(minWeightPath.begin(), minWeightPath.end(), **0**);  minPathStr += std::to\_wstring(minWeight) + L"): ";  **for** (**int** val : minWeightPath) {  minPathStr += std::to\_wstring(val) + L" ";  }  sf::Text minPathText;  minPathText.setFont(font);  minPathText.setString(minPathStr);  minPathText.setCharacterSize(**18**);  minPathText.setFillColor(sf::Color::Black);  minPathText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  sf::Text lastPositiveText;  lastPositiveText.setFont(font);  lastPositiveText.setString(L"15) Последний уровень с положительными элементами: " + std::to\_wstring(lastPositiveLevel));  lastPositiveText.setCharacterSize(**18**);  lastPositiveText.setFillColor(sf::Color::Black);  lastPositiveText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  std::wstring maxOnLevelsStr = L"16) Максимальные элементы на уровнях: ";  **for** (**size\_t** i = **0**; i < maxOnLevels.size(); ++i) {  maxOnLevelsStr += L"Уровень " + std::to\_wstring(i) + L": " + std::to\_wstring(maxOnLevels[i]) + L"; ";  }  sf::Text maxOnLevelsText;  maxOnLevelsText.setFont(font);  maxOnLevelsText.setString(maxOnLevelsStr);  maxOnLevelsText.setCharacterSize(**18**);  maxOnLevelsText.setFillColor(sf::Color::Black);  maxOnLevelsText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  sf::Text oddLevelSumText;  oddLevelSumText.setFont(font);  oddLevelSumText.setString(L"18) Сумма элементов на нечетных уровнях: " + std::to\_wstring(oddLevelSum));  oddLevelSumText.setCharacterSize(**18**);  oddLevelSumText.setFillColor(sf::Color::Black);  oddLevelSumText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  sf::Text minMaxPathsText;  minMaxPathsText.setFont(font);  minMaxPathsText.setString(L"19) Минимальный путь между листьями: " + std::to\_wstring(minPath) +  L" Максимальный путь между листьями: " + std::to\_wstring(maxPath));  minMaxPathsText.setCharacterSize(**18**);  minMaxPathsText.setFillColor(sf::Color::Black);  minMaxPathsText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  sf::Text inOrderText;  inOrderText.setFont(font);  inOrderText.setString(L"21) Инфиксный обход: " + inOrder.str());  inOrderText.setCharacterSize(**18**);  inOrderText.setFillColor(sf::Color::Black);  inOrderText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  sf::Text postOrderText;  postOrderText.setFont(font);  postOrderText.setString(L"21) Постфиксный обход: " + postOrder.str());  postOrderText.setCharacterSize(**18**);  postOrderText.setFillColor(sf::Color::Black);  postOrderText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  sf::Text preOrderText;  preOrderText.setFont(font);  preOrderText.setString(L"21) Префиксный обход: " + preOrder.str());  preOrderText.setCharacterSize(**18**);  preOrderText.setFillColor(sf::Color::Black);  preOrderText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  std::wstring nodesAndLeavesStr = L"17) Количество внутренних узлов и листьев на уровнях:**\n**";  **for** (**size\_t** i = **0**; i < internalCounts.size(); ++i) {  nodesAndLeavesStr += L"Уровень " + std::to\_wstring(i) + L": Внутренние = " +  std::to\_wstring(internalCounts[i]) + L", Листья = " + std::to\_wstring(leafCounts[i]) + L"**\n**";  }  sf::Text nodesAndLeavesText;  nodesAndLeavesText.setFont(font);  nodesAndLeavesText.setString(nodesAndLeavesStr);  nodesAndLeavesText.setCharacterSize(**18**);  nodesAndLeavesText.setFillColor(sf::Color::Black);  nodesAndLeavesText.setPosition(**10**, yOffset);  yOffset += lineSpacing;  **while** (resultsWindow.isOpen()) {  sf::Event event;  **while** (resultsWindow.pollEvent(event)) {  **if** (event.type == sf::Event::Closed)  resultsWindow.close();  **if** (event.type == sf::Event::KeyPressed && event.key.code == sf::Keyboard::Escape) {  resultsWindow.close();  }  **if** (event.type == sf::Event::KeyPressed && event.key.code == sf::Keyboard::Enter) {  drawBalanceTreeWindow(root);  }  // Обработка клика на кнопку  **if** (event.type == sf::Event::MouseButtonPressed && event.mouseButton.button == sf::Mouse::Left) {  sf::Vector2i mousePos = sf::Mouse::getPosition(resultsWindow);  **if** (button2.getGlobalBounds().contains(**static\_cast**<sf::Vector2f>(mousePos))) {  // Переход на следующее окно  drawBalanceTreeWindow(root);  }  }  }  resultsWindow.clear(sf::Color(**128**, **200**, **255**, **128**));  resultsWindow.draw(sumText);  resultsWindow.draw(productText);  resultsWindow.draw(occurrencesText);  resultsWindow.draw(unequalDescendantsText);  resultsWindow.draw(unequalHeightsText);  resultsWindow.draw(maxElementText);  resultsWindow.draw(duplicatesText);  resultsWindow.draw(maxFreqText);  resultsWindow.draw(symmetricText);  resultsWindow.draw(bstText);  resultsWindow.draw(leavesText);  resultsWindow.draw(minPathText);  resultsWindow.draw(lastPositiveText);  resultsWindow.draw(oddLevelSumText);  resultsWindow.draw(minMaxPathsText);  resultsWindow.draw(maxOnLevelsText);  resultsWindow.draw(inOrderText);  resultsWindow.draw(postOrderText);  resultsWindow.draw(preOrderText);  resultsWindow.draw(nodesAndLeavesText);  resultsWindow.draw(button2);  resultsWindow.draw(buttonText2);  resultsWindow.display();  }  }  **int** **main**() {  setlocale(LC\_ALL, "Russian");  sf::RenderWindow window(sf::VideoMode(**1920**, **1080**), L"Лабораторная работа №5 - Исходное дерево");  Node\* root = nullptr;  **const** **int** nodeCount = **23**;  **const** **int** minKey = -**80**;  **const** **int** maxKey = **160**;  buildTree(root, nodeCount, minKey, maxKey);  sf::Font font;  **if** (!font.loadFromFile("C:/Windows/Fonts/times.ttf")) {  std::cerr << "Ошибка загрузки шрифта" << std::endl;  **return** -**1**;  }  // Создаем кнопку  sf::RectangleShape button(sf::Vector2f(**200**, **100**));  button.setPosition(**1700**, **900**); // Позиция кнопки  button.setFillColor(sf::Color::Green);  // Текст для кнопки  **if** (!font.loadFromFile("C:/Windows/Fonts/times.ttf")) {  std::cerr << "Ошибка загрузки шрифта" << std::endl;  **return** -**1**;  }  sf::Text buttonText("Next", font, **30**);  buttonText.setFillColor(sf::Color::White);  buttonText.setPosition(**1770**, **930**); // Позиция текста (по центру кнопки)  **while** (window.isOpen()) {  sf::Event event;  **while** (window.pollEvent(event)) {  **if** (event.type == sf::Event::Closed)  window.close();  **if** (event.type == sf::Event::KeyPressed && event.key.code == sf::Keyboard::Enter) {  drawResultsWindow(root);  }  // Обработка клика на кнопку  **if** (event.type == sf::Event::MouseButtonPressed && event.mouseButton.button == sf::Mouse::Left) {  sf::Vector2i mousePos = sf::Mouse::getPosition(window);  **if** (button.getGlobalBounds().contains(**static\_cast**<sf::Vector2f>(mousePos))) {  // Переход на следующее окно  drawResultsWindow(root);  }  }  }  window.clear(sf::Color (**128**, **200**, **255**, **128**)); // RGBA (с прозрачностью));  drawTree(window, root, **900**, **50**, **400**);  window.draw(button);  window.draw(buttonText);  window.display();  }  **return** **0**;  } |

1. **Результат работы программы**

Результат работы программы представлен на рисунках 1-5.

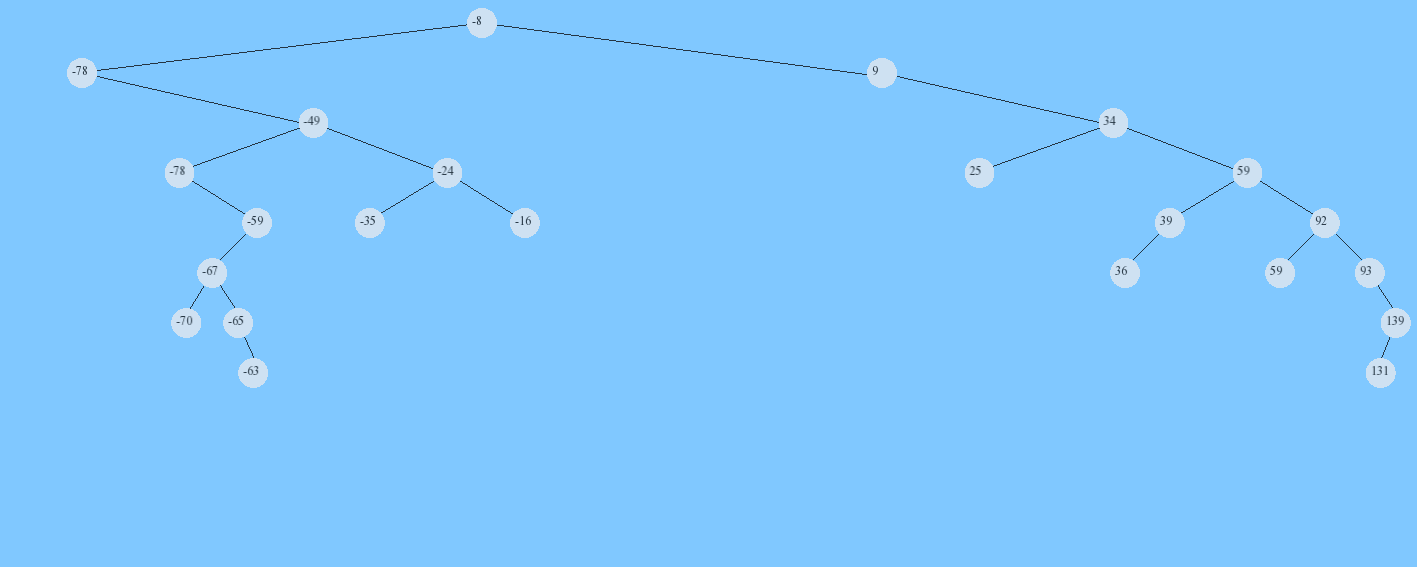


Рисунок 1 – Исходное дерево

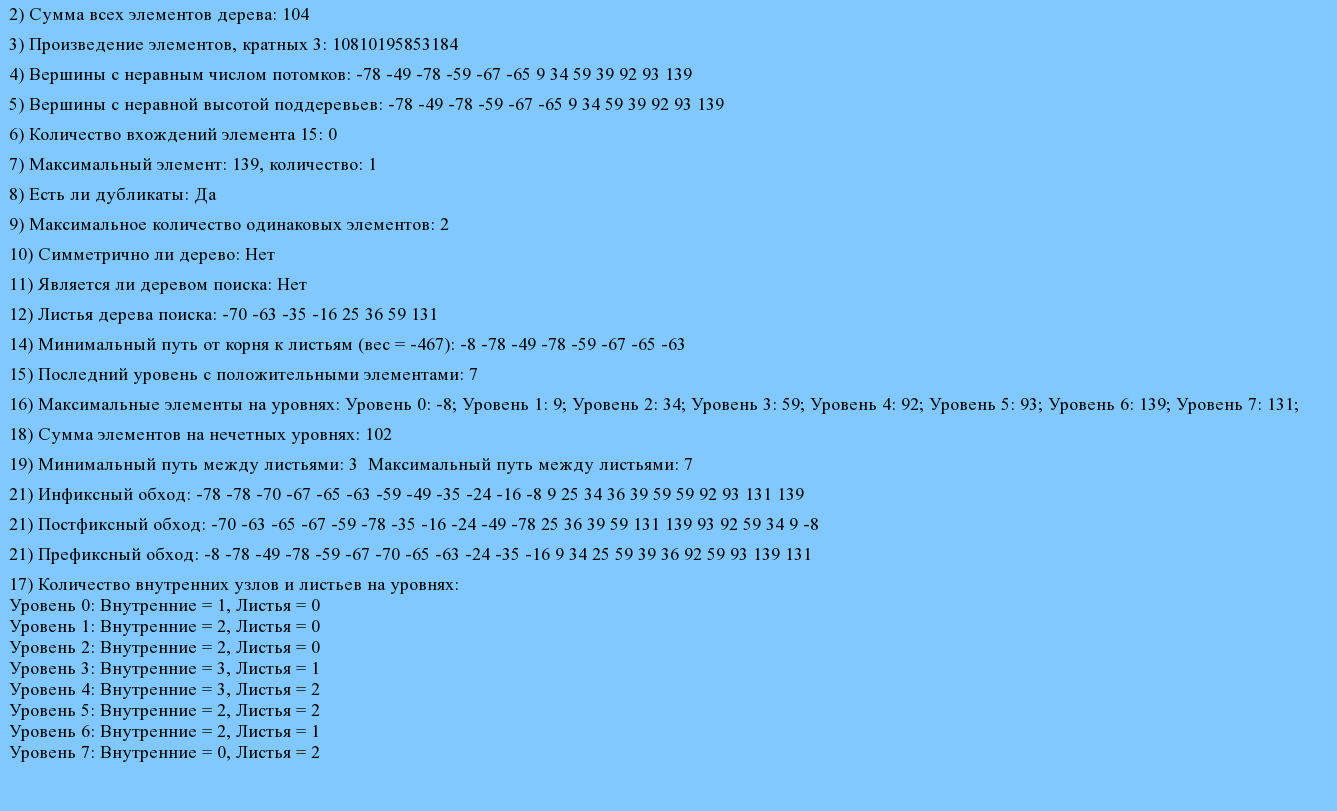


Рисунок 2 – Окно результатов

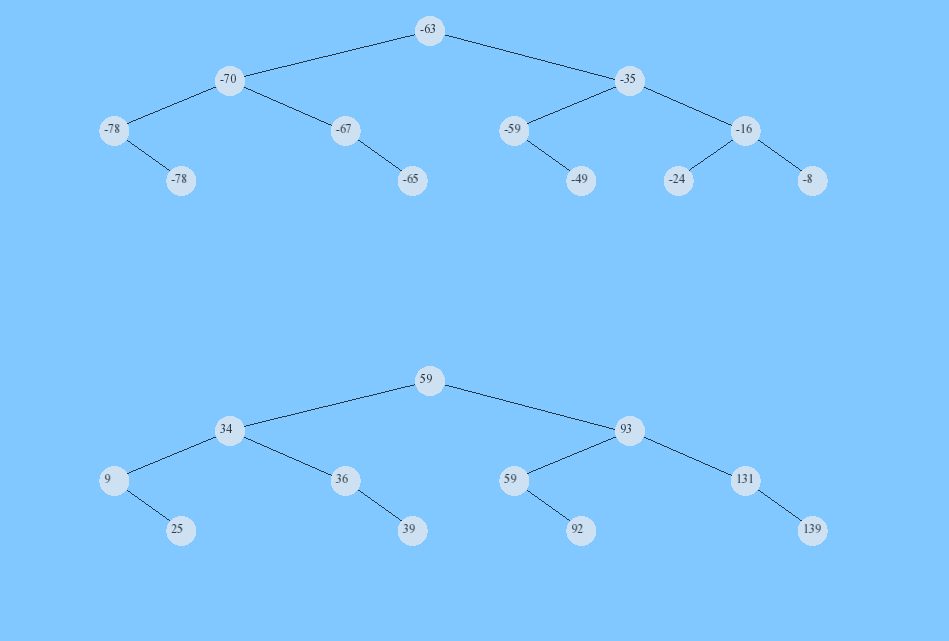


Рисунок 3 – Сбалансированные деревья с положительными и отрицательными элементами

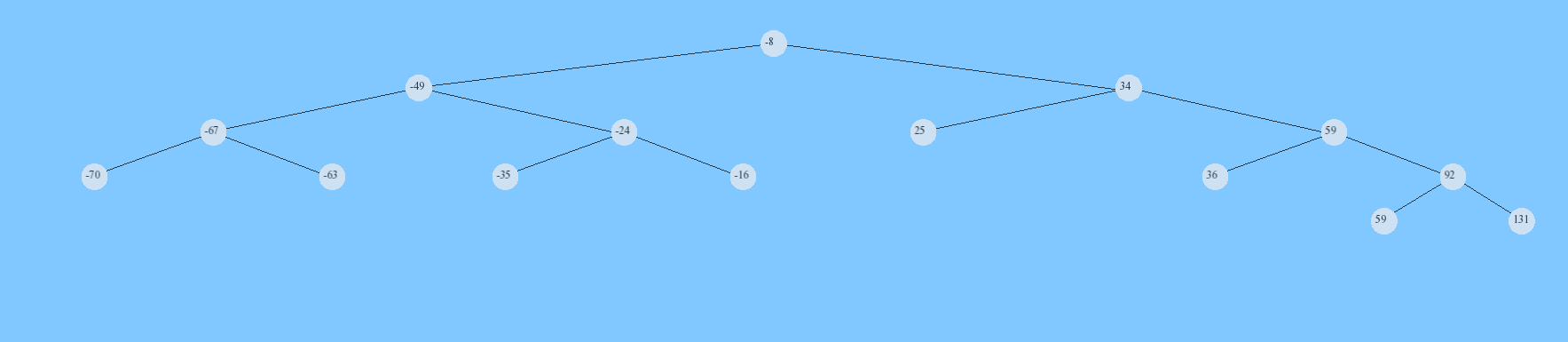


Рисунок 4 – Строго бинарное дерево с наименьшими удаленными вршинами

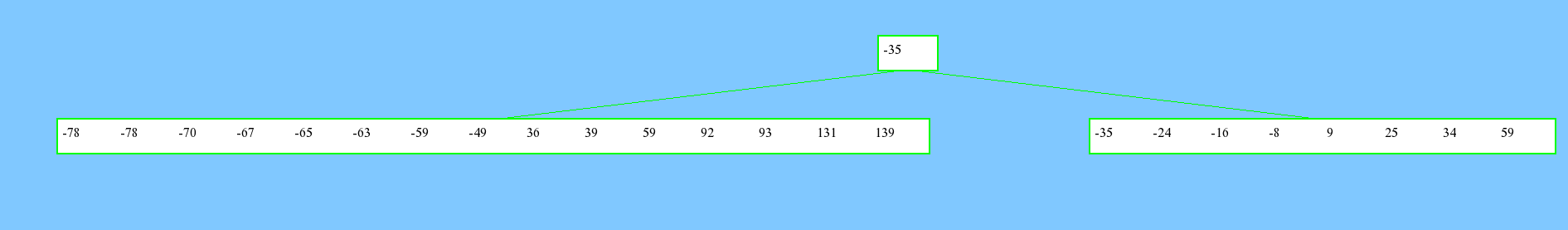


Рисунок 5 – Дерево В+

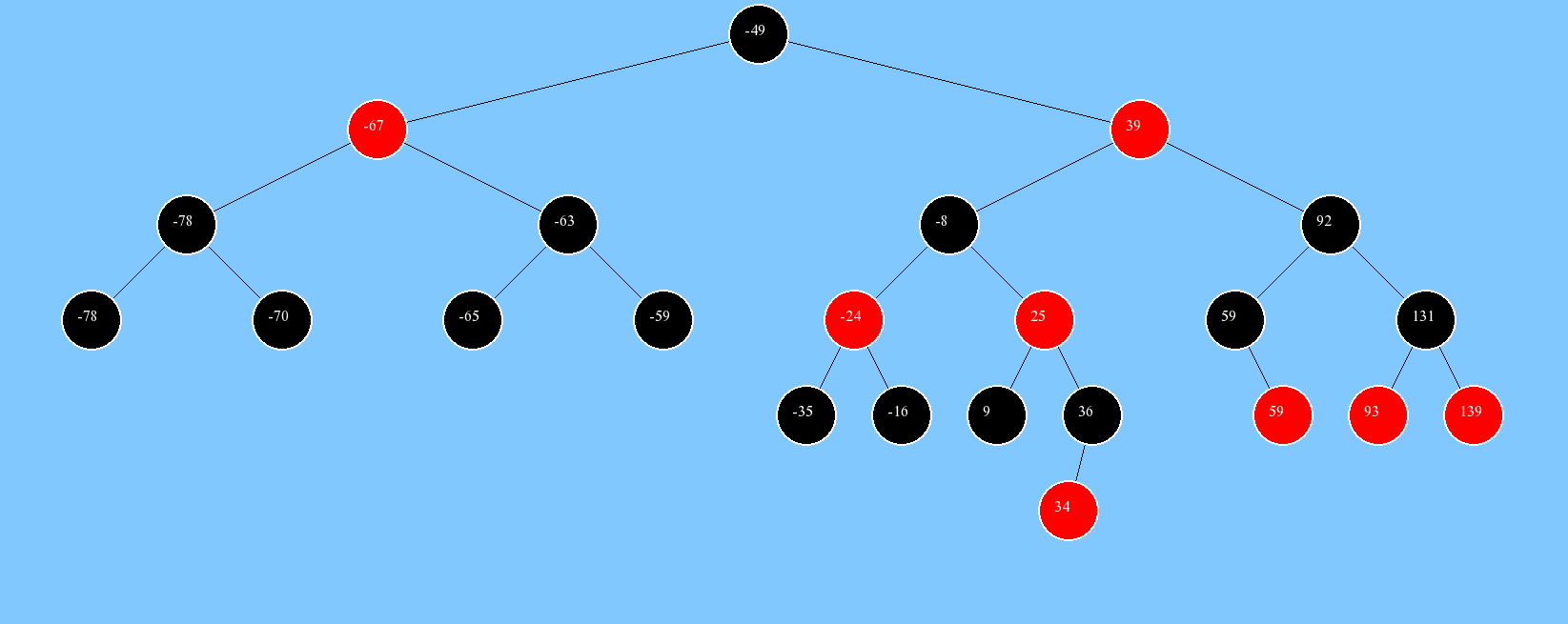


Рисунок 6 – Красно-черное дерево

1. **Вывод**

В ходе проделанной работы были изучены понятия, формирования, особенности алгоритмизации и программирования задач, в которых используются деревья.