**Структуры алгоритмов.**

**Сравнение двоичного дерева поиска и красно-черного дерева.**

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**10.05.03. КП20 101801107 02**

Листов 27

**Содержание:**

1. Тестируемые структуры………………………………………………………………3

1.1 Двоичное дерево поиска (BTS) ……………………………………………3

1.2 Двоичное дерево поиска (AVL) ……………………………………………3

1.3 Красно-черное дерево (RBT) ………………………………………………3

2. Описание основных операций с BTS и RBT, AVL …………………………………5

2.1 Вставка элемента …………………………...………………………………5

2.2 Удаление элемента ………………………...………………………………9

2.3 Поиск элемента …………………………...…………………………………12

3. Описание структуры программного комплекса…..……………………………....…13

4. Описание структур данных…………………………………………...………………16

5. Описание методики проведения экспериментального исследования……......……22

6. Описание и анализ результатов проведённого исследования……………….…..…21

7. Выводы по результатам проведенного анализа…………………………….….……26

8. Список литературы………………….………………………………………………....27

## 1. Тестируемые структуры

**Двоичное дерево поиска (BTS)**

**Двоичное дерево поиска** (англ. *binary search tree*) —  это простой алгоритм сортировки. Суть его заключается в том что, на каждом шаге алгоритма мы берем один из элементов массива, находим позицию для вставки и вставляем. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

Пример сортировки слиянием. Сначала делим список на кусочки (по 1 элементу), затем сравниваем каждый элемент с соседним, сортируем и объединяем. В итоге, все элементы отсортированы и объединены вместе.

**АВЛ – дерево поиска (AVL)**

**АВЛ – дерево поиска -** сбалансированное по высоте двоичное дерево поиска, в котором у любой вершины высота левого и правого поддеревьев различаются не более чем на 1. Основная идея: eсли вставка или удаление элемента приводит к нарушению сбалансированности дерева, то выполняется его балансировка. Так же вводится понятие фактор баланса: разность высот его левого и правого поддеревьев.

**Красно-черное дерево (RBT)**

**Красно-черное дерево** (англ. *red-black tree*) —  это сбалансированное двоичное дерево поиска. Сбалансированность такого дерева достигается за счёт введения дополнительного атрибута узла дерева – «цвета». Атрибут может принимать одно из двух значений – «черный», «красный».

При этом все листья дерева являются фиктивными и не содержат данных, но относятся к дереву и являются чёрными. Цвет имеет свои свойства:

* Каждый узел промаркирован красным или чёрным цветом
* Корень и конечные узлы (листья) дерева — чёрные
* У красного узла родительский узел — чёрный
* Все простые пути из любого узла x до листьев содержат одинаковое количество чёрных узлов
* Чёрный узел может иметь чёрного родителя

**2. ОПИСАНИЕ АЛГОРИТМОВ РЕШЕНИЯ ЗАДАЧИ**

**2.1 Вставка элемента:**

**Примечание:** данные добавляются в двоичное дерево с помощью уникальных ключей, поэтому добавление дубликатов, было исключено.

**Для двоичного дерева, он же BST:**

Шаг 1. Если дерево пусто, заменить его на дерево с одним корневым узлом и остановиться.

Шаг 2. Иначе сравнить ключ новой вершины K с ключом   
корневого узла X.

* Если K>X, рекурсивно добавить новую вершину в правое поддерево.
* Если K<X, рекурсивно добавить новую вершину в левое поддерево.
* Если K=X, заменить вершину текущего узла новым значением.

**Для двоичного дерева с балансировкой (AVL):**

Шаг 1. Обход дерева с поиском, пока не убедимся, что ключа в дереве нет.

Шаг 2. Включения новой вершины в дерево и определения результирующих показателей балансировки.

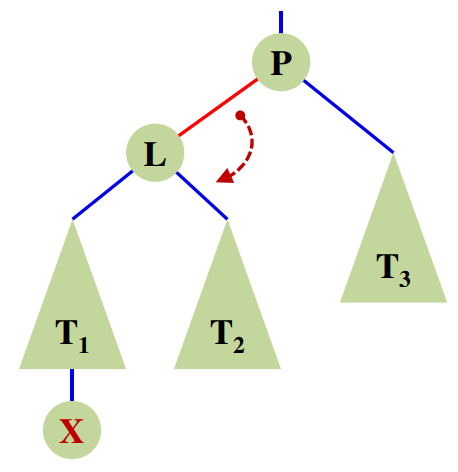
Шаг 3. Необходимо обновить коэффициенты сбалансированности родительских узлов.

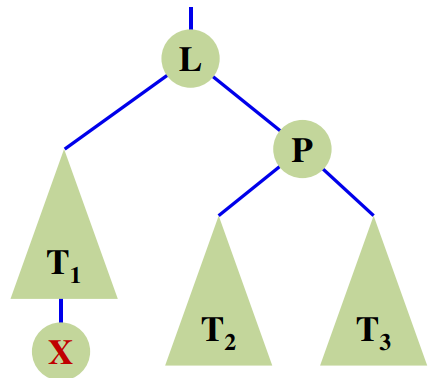
Шаг 1. Находим коэффициенты родительских узлов.

Шаг 2. Если любой родительский узел принял значение -2 или 2, то необходимо выполнить балансировку поддерева путем поворота.

**Правый одиночный поворот** выполняется, когда в левое поддерево добавляется новый элемент.

Поворачиваем ребро, связывающее корень и его левый дочерний узел, вправо.





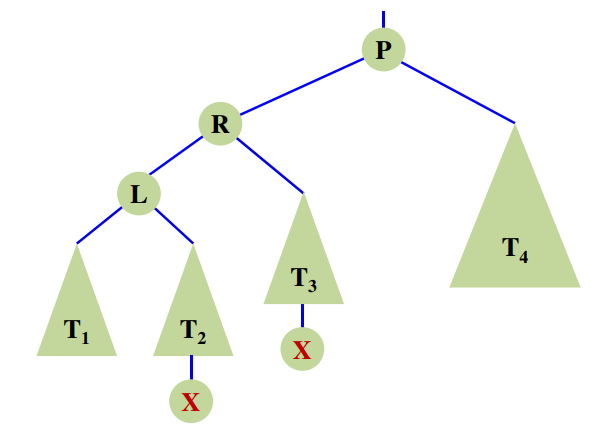
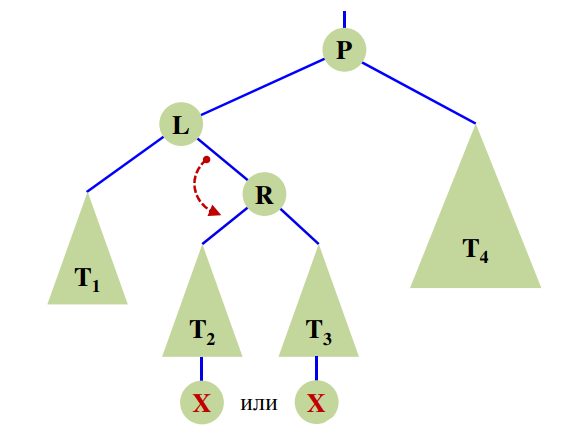
**Левый одиночный поворот** выполняется, когда в правое поддерево добавляется новый элемент.

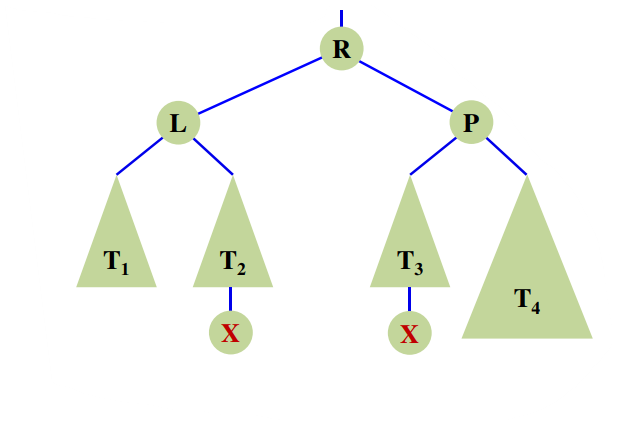
Поворачиваем ребро, связывающее корень и его левый дочерний узел, влево.

**Двойной лево-правый поворот** после добавления элемента

в правое поддерево левого дочернего узла дерева.

Шаг 1. Выполняется Левый поворот левого поддерева.



Шаг 2. Выполняется Правый поворот нового поддерева с новой вершиной. 

**Двойной право-левый поворот** после добавления элемента в правое поддерево левого дочернего узла дерева.

Шаг 1. Выполняется Правый поворот правого поддерева.

Шаг 2. Выполняется Левый поворот нового поддерева с новой вершиной.

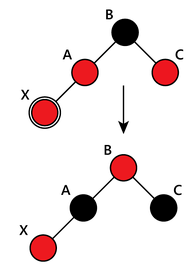
**Для красно-черного дерева:**

Шаг 1. Идём от корня, пока указатель на следующего сына не станет ***null***. Вставляем вместо него новый элемент с двумя листами потомками и красным цветом. Далее выполняем балансировку:

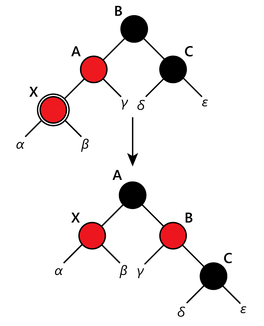
1. Если отец нового элемента черный, то завершаем алгоритм.

2. Если отец нового элемента красный, то достаточно рассмотреть два случая:

2.1. "Дядя" этого узла тоже красный. Перекрашиваем "отца" и "дядю" в чёрный цвет, а "деда" - в красный. При этом "дед" может нарушать свойство дерева ("прадед" может быть красным). Далее рекурсивно пытаемся восстановить свойства дерева, продвигаясь к предкам.



2.2 "Дядя" чёрный. Если выполнить только перекрашивание, то может нарушиться постоянство чёрной высоты дерева по всем ветвям. Поэтому выполняем поворот. Если добавляемый узел был правым потомком, то необходимо сначала выполнить левое вращение, которое сделает его левым потомком.



**2.2 Удаление элемента:**

**Для двоичного дерева, он же BST:**

Шаг 1. Если дерево пусто, остановиться;

Шаг 2. Иначе сравнить текущий ключ K с ключом X корневого узла n.

* Если K>X, рекурсивно удалить K из правого поддерева;
* Если K<X, рекурсивно удалить K из левого поддерева;
* Если K=X, то необходимо рассмотреть три случая.
  + Если обоих детей нет, то удаляем текущий узел и обнуляем ссылку на него у родительского узла;
  + Если одного из детей нет, то значения полей ребёнка m ставим вместо соответствующих значений корневого узла, затирая его старые значения, и освобождаем память, занимаемую узлом m;
  + Если оба ребёнка присутствуют, то
    - Если левый узел m правого поддерева отсутствует   
      (n->right->left)
      * Копируем из правого узла в удаляемый поля K, V и ссылку на правый узел правого потомка.
  + Иначе
    - Возьмём самый левый узел m, правого поддерева   
      n->right;
    - Скопируем данные (кроме ссылок на дочерние элементы) из m в n;
    - Рекурсивно удалим узел m.

**Для двоичного дерева с балансировкой (AVL):**

Шаг 1. Если вершина — лист, то удалим её и вызовем балансировку всех её предков в порядке от родителя к корню.

Шаг 2. Иначе найдём самую близкую по значению вершину в поддереве наибольшей высоты (правом или левом) и переместим её на место удаляемой вершины, при этом вызвав процедуру её удаления.

**Для красно-черного дерева:**

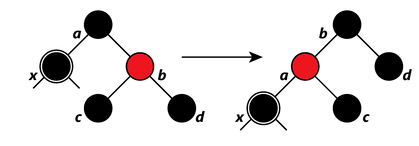
При удалении вершины могут возникнуть три случая в зависимости от количества её детей:

* Если у вершины нет детей, то изменяем указатель на неё у родителя на nгl.
* Если у неё только один ребёнок, то делаем у родителя ссылку на него вместо этой вершины.
* Если же имеются оба ребёнка, то находим вершину со следующим значением ключа. У такой вершины нет левого ребёнка (так как такая вершина находится в правом поддереве исходной вершины, и она самая левая в нем, иначе бы мы взяли ее левого ребенка.

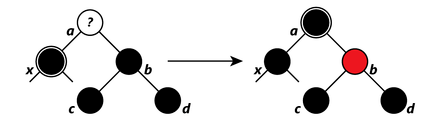
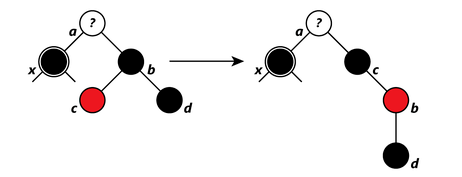
**Проверка балансировки:**

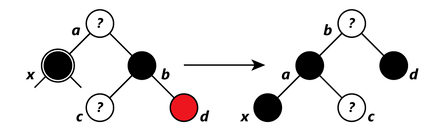
Потомок текущей вершины:

* Если брат это потомка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в чёрный, а отца — в красный цвет, сохраняя таким образом черную высоту дерева.



Если брат текущей вершины был чёрным, то получаем три случая:

1. Оба ребёнка у брата чёрные. Красим брата в красный цвет и рассматриваем далее отца вершины. Делаем его черным.
2. Если у брата правый ребёнок чёрный, а левый красный, то перекрашиваем брата и его левого сына и делаем вращение. 
3. Если у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца — в чёрный, делаем вращение. Выходим из алгоритма.

****

**2.3 Поиск элемента:**

Алгоритм поиск для каждого дерева аналогичен:

Шаг 1. Если дерево пусто, сообщить, что узел не найден, и остановиться.

Шаг 2. Иначе сравнить K со значением ключа корневого узла X.

* Если K=X, выдать ссылку на этот узел и остановиться.
* Если K>X, рекурсивно искать ключ K в правом поддереве.
* Если K<X, рекурсивно искать ключ K в левом поддереве.

**3. ОПИСАНИЕ СТРУКТУРЫ ПРОГРАММНОГО КОМПЛЕКСА**

|  |  |
| --- | --- |
| **Form1.cs** | |
| Add(object sender, EventArgs e)  void Remove(object sender, EventArgs e)  void Search(object sender, EventArgs e)  void Clear\_Click(object sender, EventArgs e)  void Clear()  void Plan(List<string> commands, List<int> nodes)  void GetPlan(object sender, EventArgs e) | |
| **BST.cs** |
| BST()  BSTNode Find(int val)  void Insert(int i)  void Remove(int key)  BSTNode GetSuccessor |
| **AVL.cs** |
| public AVL()  void Add(int key)  AVLNode RecursiveInsert(AVLNode current, AVLNode n)  AVLNode BalanceTree(AVLNode current)  int BalanceFactor(AVLNode current)  void Remove(int target)  AVLNode Delete(AVLNode current, int target)  void Find(int data)  AVLNode Find(int target, AVLNode current)  int GetHeight(AVLNode current)  int max(int l, int r)  AVLNode RotateLL(AVLNode parent)  AVLNode RotateRR(AVLNode parent)  AVLNode RotateLR(AVLNode parent)  AVLNode RotateRL(AVLNode parent) |
| **RBT.cs** |
| RBTNode NullBlackNode{ get; set; }  RBT()  RBTNode createNode(int newKey)  void MakeRoot(RBTNode node)  void MakeLeftChild(RBTNode parent, RBTNode child)  void MakeRightChild(RBTNode parent, RBTNode child)  void ReplaceParentWithChild(RBTNode parent, RBTNode child)  RBTNode Find(RBTNode node, int key)  RBTNode Find(int key)  void BalanceTreeAfterInsert(RBTNode newNode)  UnbalancedInsert(RBTNode node, RBTNode newNode)  RBTNode UnbalancedInsert(int newValue)  UnbalancedDelete(RBTNode node, out RBTNode nodeDeleted, out RBTNode nodeReplacedDeleted)  void BalanceTreeAfterDelete(RBTNode node)  void RemoveLeft(RBTNode parent)  void RemoveRight(RBTNode parent) |
| **TreeDraw.cs** |
| public TreeDrawNode(int key, NodeColor tColor)  TreeDraw(RBT tree)  TreeDraw(AVL tree)  TreeDraw(BST tree)  TreeDrawNode GetChild(RBT.RBTNode node, TreeDrawNode currentnode)  TreeDrawNode GetChild(AVL.AVLNode node, TreeDrawNode currentnode)  TreeDrawNode GetChild(BST.BSTNode node, TreeDrawNode currentnode)  Image Draw()  List<PointF> GetListPoint(float x1, float x2, float y1, float y2, float w, float h)  Image Draw(TreeDrawNode current, out int center) |

## 4. ОПИСАНИЕ СТРУКТУР ДАННЫХ

**Form1.cs**

|  |  |
| --- | --- |
| **Описание метода** | **Функция метода** |
| Add(object sender, EventArgs e) | Добавляет одну заданную вершину ко всем деревьям (вызов функции добавления в каждом дереве ) |
| void Remove(object sender, EventArgs e) | Удаляет одну заданную вершину ко всем деревьям (вызов функции удаления в каждом дереве ) |
| void Search(object sender, EventArgs e) | Производит поиск заданной вершины в каждом дереве (вызов функции поиска в каждом дереве ) |
| void Clear\_Click(object sender, EventArgs e) | Вызов очистки данных |
| void Clear() | Очистка данных |
| void Plan(List<string> commands, List<int> nodes) | Выполнение загруженного плана |
| void GetPlan(object sender, EventArgs e) | Загрузка плана |

**BST.cs**

|  |  |
| --- | --- |
| **Описание метода** | **Функция метода** |
| BST() | Создает двоичное дерево |
| BSTNode Find(int val) | Поиск вершины в дереве. Возвращает найденную вершину. |
| void Insert(int i) | Вставка вершины по значению ключа. |
| void Remove(int key) | Удаление вершины |
| BSTNode GetSuccessor (BSTNode node) | Получение преемника, он же потомок, текущей вершины. Возвращает вершину. |

**AVL.cs**

|  |  |
| --- | --- |
| **Описание метода** | **Функция метода** |
| public AVL() | Создает двоичное дерево с балансировкой. |
| void Add(int key) | Вставка вершины по значению ключа. Если корня не существует, создает его, иначе вызывает рекурсивную вставку. |
| AVLNode RecursiveInsert(AVLNode current, AVLNode n) | Производит рекурсивную вставку заданной вершины. Возвращает добавленную вершину. |
| AVLNode BalanceTree(AVLNode current) | Балансировка дерева с помощью фактора баланса. В зависимости от фактора, вызывает функции вращения. |
| int BalanceFactor(AVLNode current) | Вычисляет фактор баланса. |
| void Remove(int target) | Удаление вершины. Вызов основной функции удаления, при завершении которой корню присваиваются новые значения потомков и самого корня. |
| AVLNode Delete(AVLNode current, int target) | Основная функция удаления. Возвращает новый корень дерева с новыми значениями. |
| void Find(int data)  AVLNode Find(int target, AVLNode current) | Функции поиска |
| int GetHeight(AVLNode current) | Вычисляет высоту дерева |
| AVLNode RotateLL(AVLNode parent) | Одиночный левый поворот. |
| AVLNode RotateRR(AVLNode parent) | Одиночный правый поворот. |
| AVLNode RotateLR(AVLNode parent) | Двойной лево-правый поворот. |
| AVLNode RotateRL(AVLNode parent) | Двойной право-левый поворот. |

**RBT.cs**

|  |  |
| --- | --- |
| **Описание метода** | **Функция метода** |
| RBT() | Создает красно-черное дерево. |
| RBTNode createNode(int newKey) | Создание вершины, присваивание ей базисных значений: цвет, родителя. |
| void MakeRoot(RBTNode node) | Создание корня. |
| void кeplaceParentWithChild(RBTNode parent, RBTNode child) | Изменение поддерева с потомками. Используется при вставке. |
| RBTNode Find(RBTNode node, int key)  RBTNode Find(int key) | Функции поиска:  1) Вызывается из второй, откуда получает вершину для поиска из ключа.  2) Используется как промежуточный метод для передачи данных. |
| void BalanceTreeAfterInsert(RBTNode newNode) | Балансировка дерева, после вставки. |
| UnbalancedDelete(RBTNode node, out RBTNode nodeDeleted, out RBTNode nodeReplacedDeleted) | Удаление «без баланса», после выполнения вызывается функция балансировки. |
| RBTNode UnbalancedInsert(int newValue) | Вставка «без баланса», после выполнения вызывается функция балансировки. |
| void BalanceTreeAfterDelete(RBTNode node) | Балансировка дерева, после удаления. |

**TreeDraw.cs**

|  |  |
| --- | --- |
| **Описание метода** | **Функция метода** |
| public TreeDrawNode(int key, NodeColor tColor) | Создание вершин и копий, деревьев. |
| TreeDraw(RBT tree)  TreeDraw(AVL tree)  TreeDraw(BST tree) | Функции для копирования деревьев. Для каждого дерева создаётся копия, по которой позже создаётся холст. |
| TreeDrawNode GetChild(BST.BSTNode node, TreeDrawNode currentnode)  TreeDrawNode GetChild(RBT.RBTNode node, TreeDrawNode currentnode)  TreeDrawNode GetChild(AVL.AVLNode node, TreeDrawNode currentnode) | Функции для копирования вершин деревьев. |
| Image Draw() | Вызов функции рисования текущего дерева (копии). |
| List<PointF> GetListPoint(float x1, float x2, float y1, float y2, float w, float h) | Вычисляет массив точек для построения |
| Image Draw(TreeDrawNode current, out int center) | Функция рисования: создаёт холст, перебирает вершины, учитывает цвет. |

## 5. ОПИСАНИЕ МЕТОДИКИ ПРОВЕДЕНИЯ ЭКСПЕРИМЕНТАЛЬНОГО ИССЛЕДОВАНИЯ

Экспериментальное исследование будет состоять в том, что мы будем оценивать время работы каждого алгоритма операций вставки, удаление, поиска и построенные визуальные деревья двоичного дерева поиска с разными исходными данными.

**6. ОПИСАНИЕ И АНАЛИЗ РЕЗУЛЬТАТОВ ПРОВЕДЁННОГО ИССЛЕДОВАНИЯ**

Для эксперимента мы сначала будем загружать план из 7 команд:

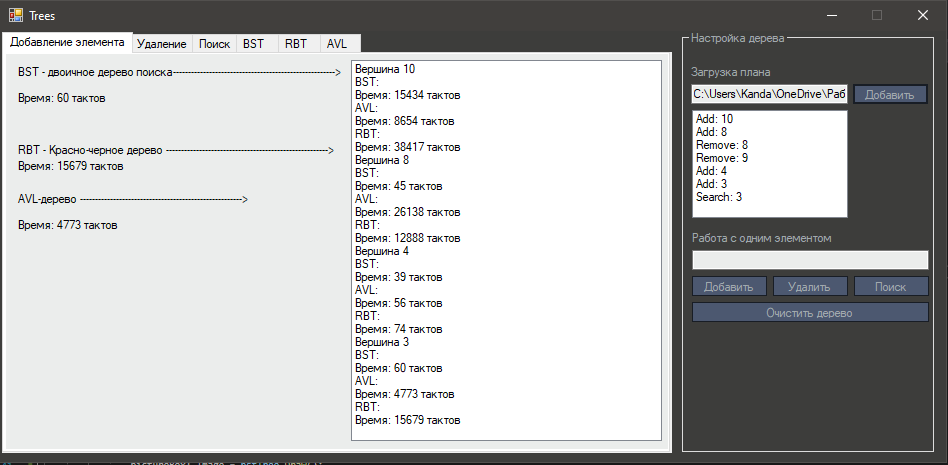


Рисунок 1 – План на 7 команд

Как мы можем заметить, что вставка элементов примерно одинакова во некоторых случаях. AVL и красно-черное дерево иногда проигрывают двоичному

Возьмем план на 52 команды:

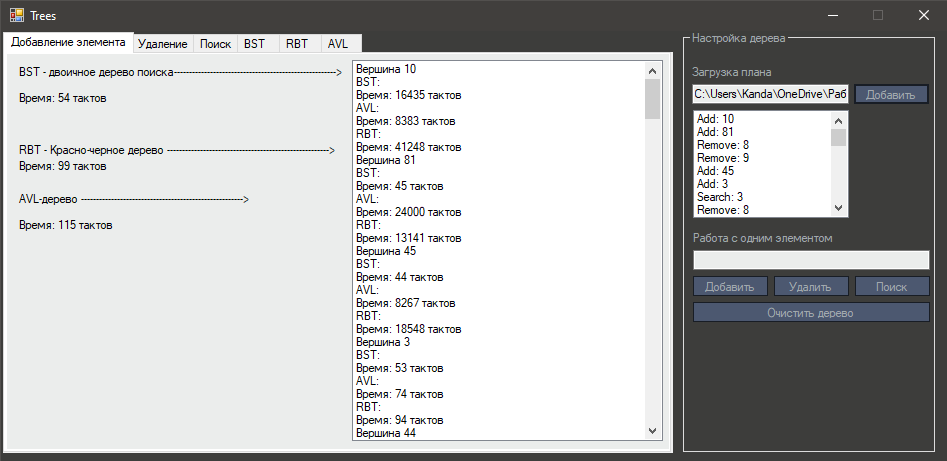


Рисунок 2 – План на 52 команды

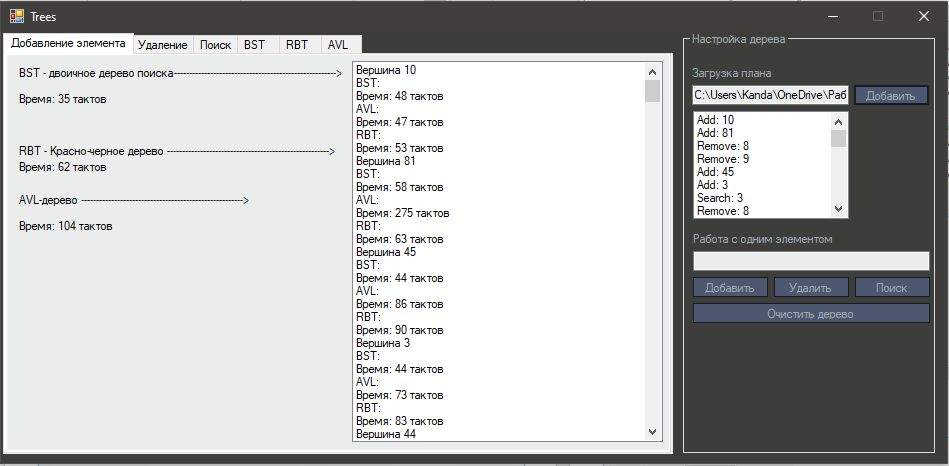


Рисунок 3 – План на 100 команд – Добавление

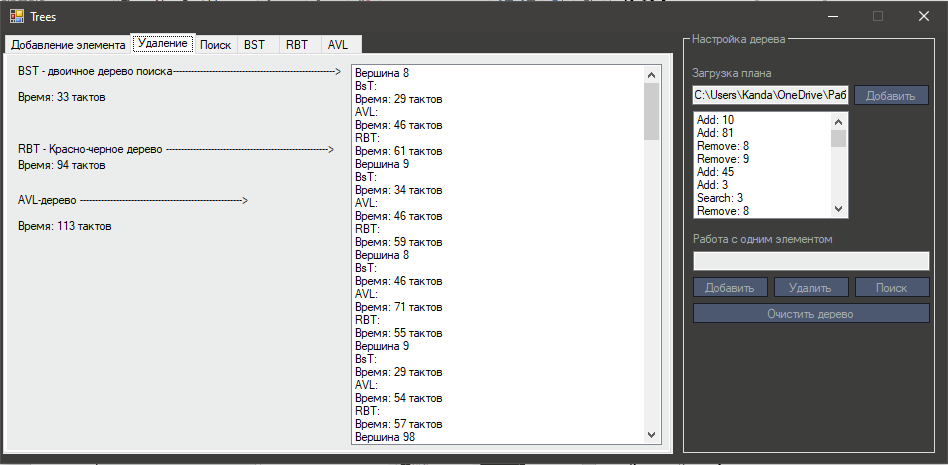


Рисунок 4 – План на 100 команд – Удаление

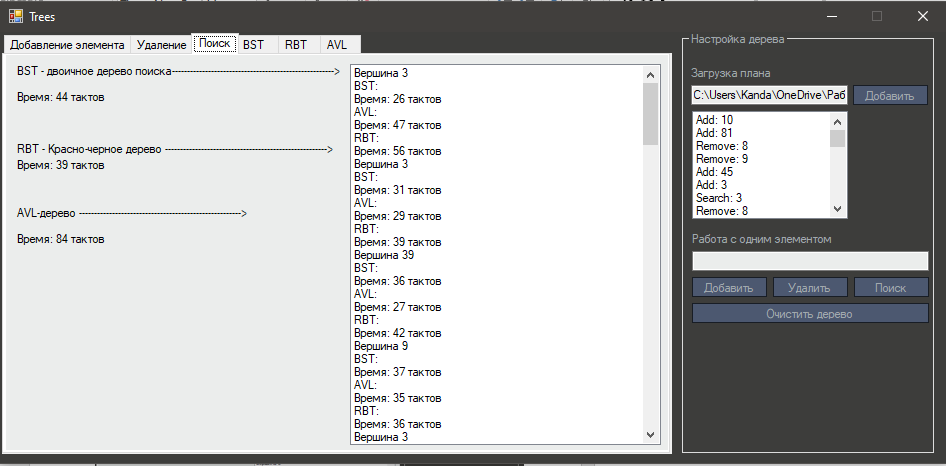


Рисунок 5 – План на 100 команд – Поиск

Далее оценим каждую структуру с помощью построенных деревьев.

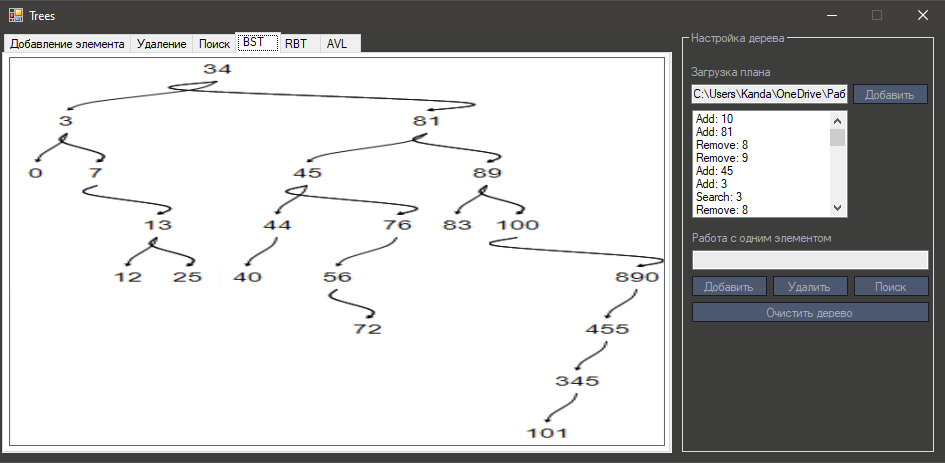


Рисунок 5 – Двоичное дерево поиска.

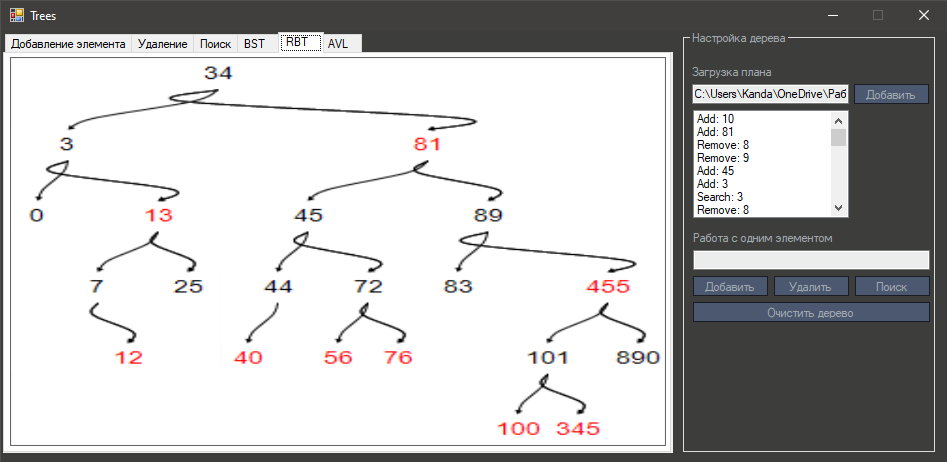


Рисунок 6 – Красно-черное дерево

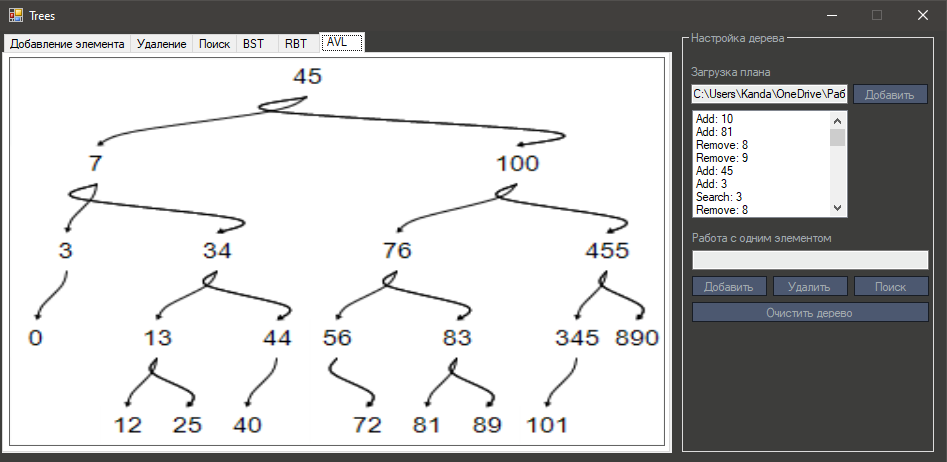


Рисунок 7 – AVL – дерево

Таблица с экспериментальными данными выглядит следующим образом

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tree(n) | Добавление t | Удаление t | Поиск t | Кол-во уровней |
|  |  |  |  |  |

Где Tree– тип структуры (n - кол-во команд), t– общее время каждой операции.

Проведем замеры времени с разным количеством элементов и запишем результат в виде таблицы.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tree(n) | Добавление t | Удаление t | Поиск t | Кол-во уровней |
| BST(7) | 33787 | 14927 | 306 | 3 |
| RBT(7) | 78558 | 20191 | 381 | 3 |
| AVL(7) | 43129 | 22003 | 447 | 3 |
| BST(52) | 1386 | 403 | 663 | 6 |
| RBT(52) | 2064 | 402 | 534 | 5 |
| AVL(52) | 2077 | 728 | 905 | 5 |
| BST(100) | 47578 | 57460 | 732 | 8 |
| RBT(100) | 310385 | 66972 | 3556 | 6 |
| AVL(100) | 87207 | 21300 | 11032 | 5 |

Как мы можем заметить, что двоичный поиск без балансирвки оказывается иногда быстрее, однако уровни такого дерева будут расти, что может сказаться на памяти.

**7. ВЫВОДЫ ПО РЕЗУЛЬТАТАМ ПРОВЕДЕННОГО АНАЛИЗА**

В ходе исследования выяснилось, что двоичные деревья имеют свои достоинства и недостатки.

По полученным результатам можно сделать вывод, что наиболее оптимальным деревом является AVL. В отличии от красно-черного дерева, балансировка AVL дерева лучше. У красно-черного может возникать пере балансировка. Однако поиск по такому дереву гораздо быстрее и эффективнее. По уровням красно-черное дерево и AVL практически не отличаются. Однако с большим количеством элементов очевидно, что AVL более сбалансирован.

## 8. Список литературы.

1. Двоичное дерево поиска [Электронный ресурс] – Режим доступа https://ru.wikipedia.org/wiki/Двоичное\_дерево\_поиска

2. Реализации алгоритмов/АВЛ-дерево [Электронный ресурс] – Режим доступа: https://ru.wikibooks.org/wiki/Реализации\_алгоритмов/АВЛ-дерево

3. Курносов Михаил Георгиевич Лекция 3: АВЛ-деревья (AVL tree) [Электронный ресурс] – Режим доступа: <http://www.mkurnosov.net/teaching/uploads/DSA/dsa-fall-lecture3.pdf>

4. Red-Black Tree Operation [Электронный ресурс] – Режим доступа: <https://www.cs.auckland.ac.nz/software/AlgAnim/red_black_op.html>