Министерство науки и высшего образования Российской Федерации

Новосибирский Государственный технический университет

Кафедра автоматизированных систем управления



**Отчет по лабораторной работе №3**

**Вариант №6.1**

**по дисциплине «Архитектура средств вычислительной техники»**

**«Деревья поиска»**

Выполнили

студенты группы АВТ-813:

Кинчаров Данил

Пайхаев Алексей

Чернаков Кирилл

Преподаватель:

Ландовский Владимир Владимирович,

к.т.н., доцент кафедры АСУ

г. Новосибирск

2020 г.

**Содержание**

[Цель работы: 3](#_Toc53608536)

[Задание: 3](#_Toc53608537)

[Cхемы структур данных: 4](#_Toc53608538)

[Блок-схемы алгоритмов: 5](#_Toc53608539)

[Текст программы: 13](#_Toc53608540)

[Описание работы программы с интерфейсом: 15](#_Toc53608541)

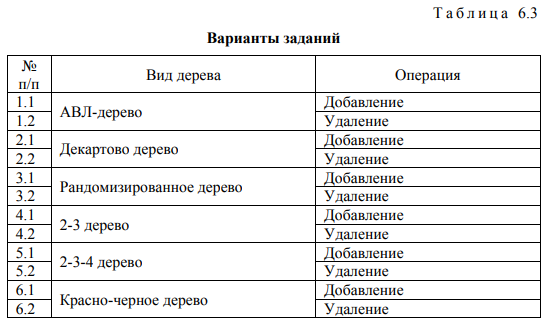
[Вывод: 16](#_Toc53608542)

# Цель работы:

Познакомиться со способами балансировки двоичных деревьев поиска и другими видами сбалансированных деревьев, приобрести навыки их реализации.

# Задание:

Для заданного в табл. 6.3 вида дерева разработать программу, позволяющую оценить число выполняемых операций при вставке (удалении) n элементов. Построить графики изменения среднего числа операций на вставку (удаление) одного элемента в зависимости от n. Каждый из вариантов деревьев, как и в предыдущей работе, разрабатывает коллектив из двух подгрупп.

****

# Схемы структур данных:

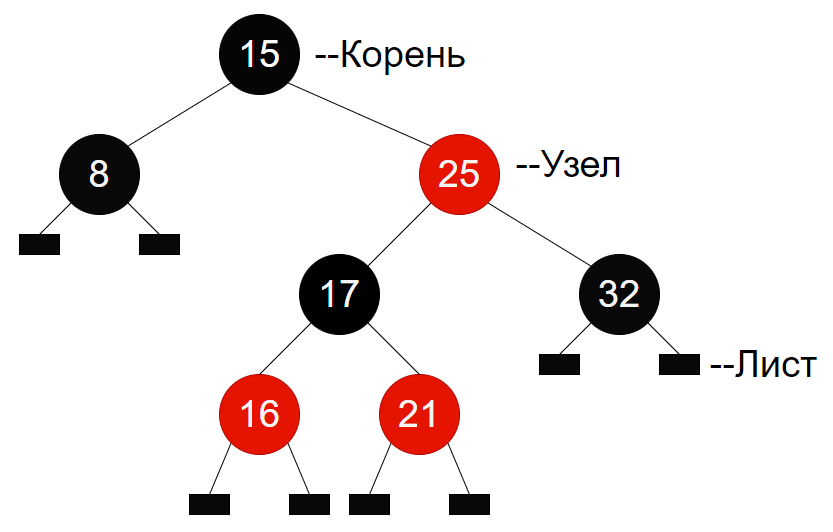


Рисунок 1 – Схема красно-черного дерева.

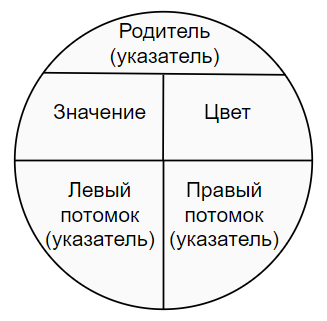
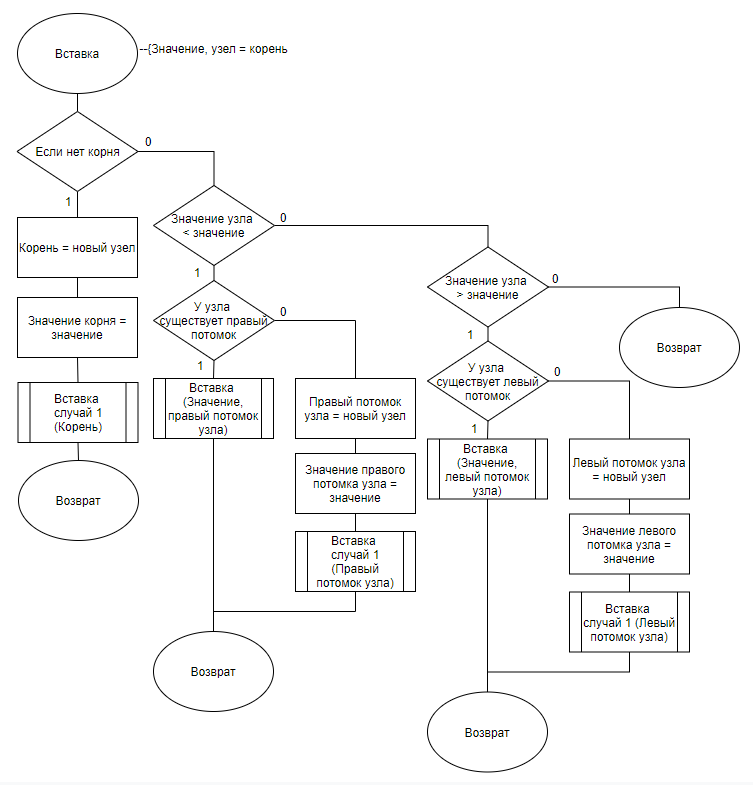
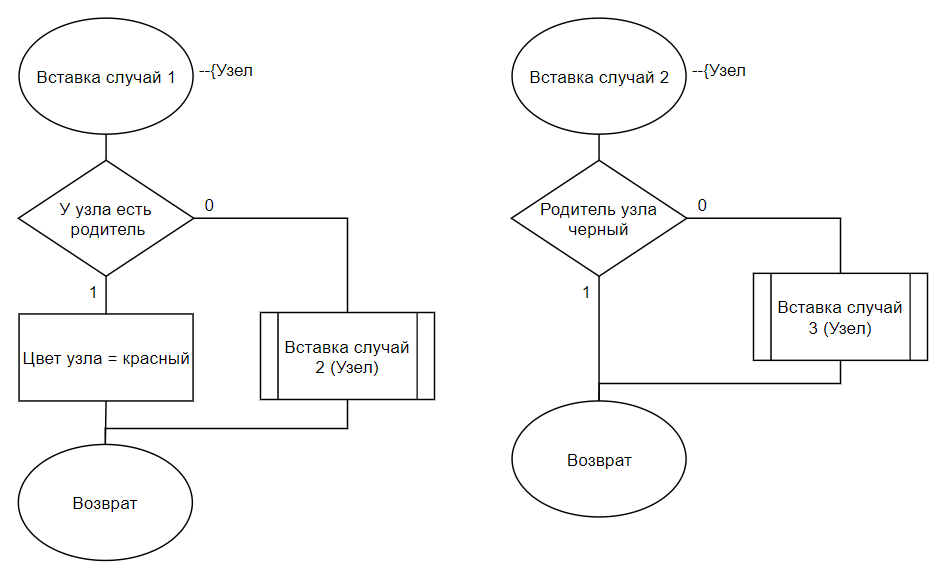
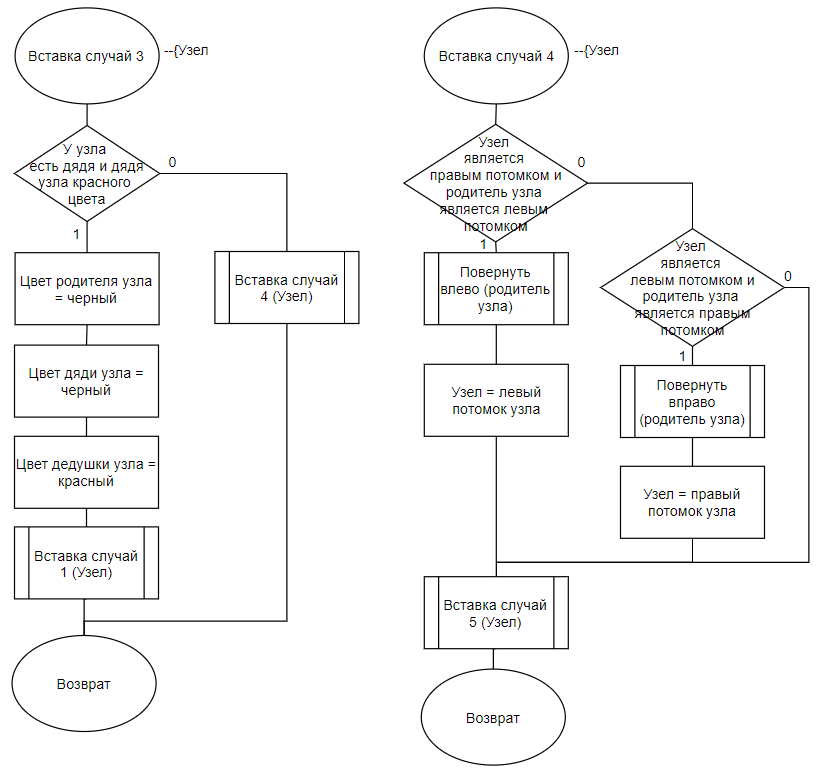


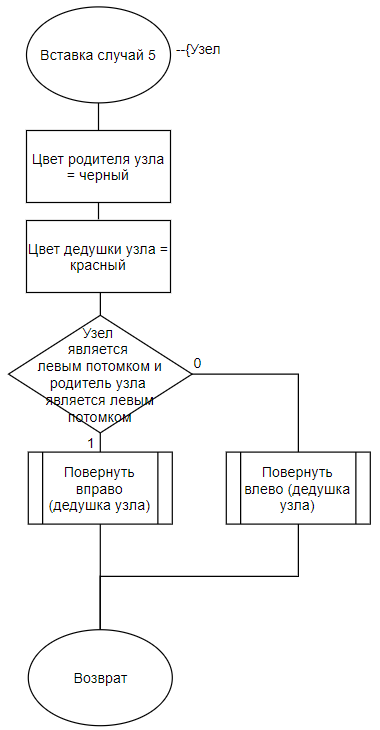
Рисунок 2 – Структура узла.

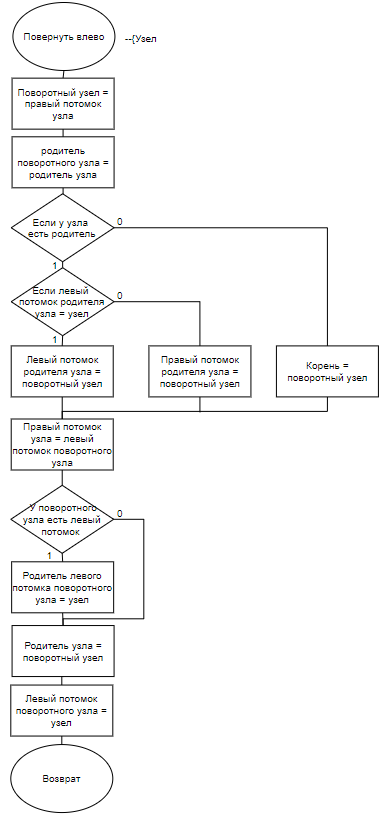
# Блок-схемы алгоритмов:

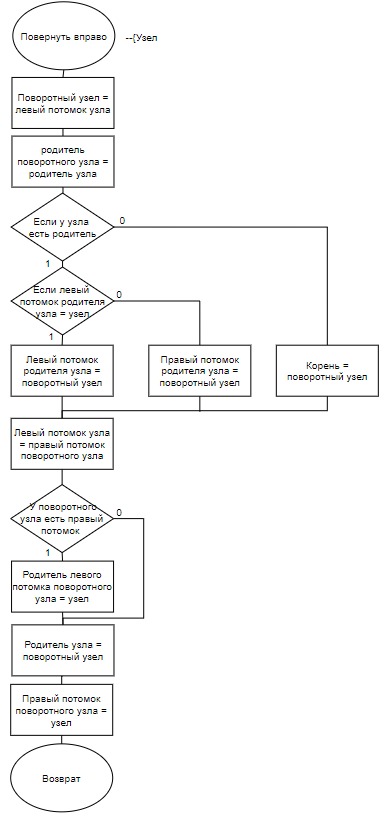








****

****

# Результаты экспериментов

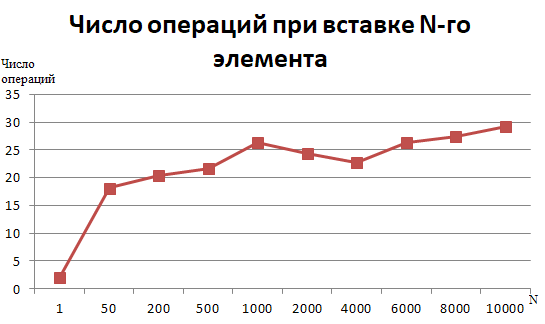
****

Рис. 3 - график изменения среднего числа операций на вставку одного элемента в зависимости от N. (Средние значения по результатам 100 экспериментов)

# Текст программы:

class RedBlackTree {

  constructor() {

    this.root = null

    this.counter = 0

  }

  getGrandParent( node ) {

    if ( node !== null && node.parent !== null ) {

      return node.parent.parent

    } else {

      return null

    }

  }

  getUncle( node ) {

    let grandParent = this.getGrandParent( node )

    this.counter++

    if ( grandParent == null ) {

      return null

    }

    else if ( node.parent === grandParent.left ) {

      return grandParent.right

    } else {

      return grandParent.left

    }

  }

  rotateLeft( node ) {

    let pivot = node.right

    pivot.parent = node.parent

    this.counter += 2

    if ( node.parent !== null ) {

      if ( node.parent.left === node ) {

        node.parent.left = pivot

        this.counter++

      } else {

        node.parent.right = pivot

        this.counter++

      }

    } else {

      this.root = pivot

      this.counter++

    }

    node.right = pivot.left

    if ( pivot.left !== null ) {

      pivot.left.parent = node

      this.counter++

    }

    node.parent = pivot

    pivot.left = node

    this.counter += 3

  }

  rotateRight( node ) {

    let pivot = node.left

    pivot.parent = node.parent

    this.counter += 2

    if ( node.parent !== null ) {

      if ( node.parent.left === node ) {

        node.parent.left = pivot

        this.counter++

      } else {

        node.parent.right = pivot

        this.counter++

      }

    } else {

      this.root = pivot

      this.counter++

    }

    node.left = pivot.right

    if ( pivot.right !== null ) {

      pivot.right.parent = node

      this.counter++

    }

    node.parent = pivot

    pivot.right = node

    this.counter += 3

  }

  resetCounter() {

    this.counter = 0

  }

  insert( value, node = this.root ) {

    if ( this.root === null ) {

      this.root = {

        value,

        left: null,

        right: null,

        parent: null,

        colorRed: true

      }

      this.insertCase1( this.root )

      this.counter++

      return this.counter

    } else if ( node.value < value ) {

        if ( node.right !== null ) {

          this.insert( value, node.right )

          this.counter++

        } else {

          node.right = {

            value,

            left: null,

            right: null,

            parent: node,

            colorRed: true

          }

          this.insertCase1( node.right )

          this.counter++

        }

        return this.counter

    } else if ( node.value > value ) {

      if ( node.left !== null ) {

        this.insert( value, node.left )

        this.counter++

      } else {

        node.left = {

          value,

          left: null,

          right: null,

          parent: node,

          colorRed: true

        }

        this.insertCase1( node.left )

        this.counter++

      }

      return this.counter

    } else {

        return this.counter

    }

  }

  insertCase1( node ) {

    if ( node.parent === null ) {

      node.colorRed = false

      this.counter++

    } else {

      this.insertCase2( node )

      this.counter++

    }

  }

  insertCase2( node ) {

    if ( !node.parent.colorRed ) {

      return

    } else {

      this.insertCase3( node )

      this.counter++

    }

  }

  insertCase3( node ) {

    let uncle = this.getUncle( node )

    let grandParent = this.getGrandParent( node )

    this.counter += 2

    if ( uncle !== null && uncle.colorRed ) {

      node.parent.colorRed = false

      uncle.colorRed = false

      grandParent.colorRed = true

      this.insertCase1( grandParent )

      this.counter += 4

    } else {

      this.insertCase4( node )

      this.counter++

    }

  }

  insertCase4( node ) {

    let grandParent = this.getGrandParent( node )

    this.counter++

    if ( node === node.parent.right && node.parent === grandParent.left ) {

      this.rotateLeft( node.parent )

      node = node.left

      this.counter += 2

    } else if ( node === node.parent.left && node.parent === grandParent.right ) {

      this.rotateRight( node.parent )

      node = node.right

      this.counter += 2

    }

    this.insertCase5( node )

    this.counter++

  }

  insertCase5( node ) {

    let grandParent = this.getGrandParent( node )

    node.parent.colorRed = false

    grandParent.colorRed = true

    this.counter += 3

    if ( node === node.parent.left && node.parent === grandParent.left ) {

      this.rotateRight( grandParent )

      this.counter++

    } else {

      this.rotateLeft( grandParent )

      this.counter++

    }

  }

}

let N = 10000

let results = new Array( N ).fill( 0 )

for( let j = 0; j < 100; j++ ) {

  let tree = new RedBlackTree()

  for( let i = 0; i < N; i++ ) {

    tree.insert( Math.random() \* 100 )

    results[ i ] += tree.counter

    tree.resetCounter()

  }

}

for( let i = 0; i < N; i++ ) {

  results[ i ] /= 100

}

# Вывод:

В ходе лабораторной работы была изучена балансировка двоичного красно-черного дерева поиска и реализован алгоритм вставки узла в дерево.

Преимущество красно-черных деревьев в том, что при вставке выполняется не более  O(1) вращений. Ещё важно, что примерно половина вставок произойдут задаром. Процедуру балансировки красно-черного дерева практически всегда можно выполнять параллельно с процедурами поиска, так как алгоритм поиска не зависит от атрибута цвета узлов.

Сбалансированность этих деревьев хуже, чем у [АВЛ](https://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%92%D0%9B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE), но работа по поддержанию сбалансированности в красно-чёрных деревьях обычно эффективнее. Для балансировки красно-чёрного дерева производится минимальная работа по сравнению с АВЛ-деревьями.