МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования

**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**

АНАЛИЗ КРАСНО-ЧЁРНОГО ДЕРЕВА

ОТЧЁТ

студента 2 курса 251 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Толстова Роберта Сергеевича

Проверено:

доцент, к. ф.-м. н. _____ М. И. Сафрончик

Саратов 2025

СОДЕРЖАНИЕ

1 Анализ красно-чёрного дерева

1.1 Описание структуры, процедур и функций на C++ в заголовках

```
#pragma once
#include <iomanip>
#include <iostream>

enum Color { BLACK, RED };

struct RBT {
  Color color;
  int value;
  RBT *right;
  RBT *left;
  RBT *parent;
};

RBT *createRoot(int value);
RBT *createNode(int value);

void leftTurn(RBT *&root, RBT *node);
void rightTurn(RBT *&root, RBT *node);

RBT *findGrandParent(RBT *node);
RBT *findUncle(RBT *node);
RBT *findSibling(RBT *node);

void insert(RBT *&root, RBT *parent, int value);
void remove(RBT *&root, RBT *parent, int value);

RBT *find(RBT *root, int value);
```

```
void maxHeight(RBT *root, short &max, short depth = 1);
void print(RBT *x);
```

1.2 Имплементация методов на C++

```
#include "RBT.h"
#include <iomanip>
#include <iostream>

using namespace std;

namespace {
void insertCase1(RBT *&root, RBT *node);
void insertCase2(RBT *&root, RBT *node);
void insertCase3(RBT *&root, RBT *node);
void insertCase4(RBT *&root, RBT *node);
void insertCase5(RBT *&root, RBT *node);

void deleteCase1(RBT *&root, RBT *node);
void deleteCase2(RBT *&root, RBT *node);
void deleteCase3(RBT *&root, RBT *node);
void deleteCase4(RBT *&root, RBT *node);
void deleteCase5(RBT *&root, RBT *node);
void deleteCase6(RBT *&root, RBT *node);

void replaceNode(RBT *&root, RBT *node);
void removeOne(RBT *&root, RBT *node);
} // namespace

RBT *createRoot(int value) {
  RBT *node = new RBT;
  node->parent = node->left = node->right = nullptr;
  node->color = BLACK;
```

```
  node->value = value;
  return node;
}

RBT *createNode(int value) {
  RBT *node = new RBT;
  node->value = value;
  node->color = RED;
  node->left = node->right = node->parent = nullptr;
  return node;
}

void leftTurn(RBT *&root, RBT *currentNode) {
  RBT *rightChild = currentNode->right;
  currentNode->right = rightChild->left;

  if (rightChild->left != nullptr) {
    rightChild->left->parent = currentNode;
  }
  rightChild->parent = currentNode->parent;

  if (currentNode->parent) {
    if (currentNode == currentNode->parent->left) {
      currentNode->parent->left = rightChild;
    } else {
      currentNode->parent->right = rightChild;
    }
  }

  rightChild->left = currentNode;
  currentNode->parent = rightChild;
```

```
  if (!rightChild->parent) {
    root = rightChild;
    root->color = BLACK;
  }
}

void rightTurn(RBT *&root, RBT *currentNode) {
  RBT *leftChild = currentNode->left;
  currentNode->left = leftChild->right;

  if (leftChild->right != nullptr) {
    leftChild->right->parent = currentNode;
  }
  leftChild->parent = currentNode->parent;

  if (currentNode->parent) {
    if (currentNode == currentNode->parent->left) {
      currentNode->parent->left = leftChild;
    } else {
      currentNode->parent->right = leftChild;
    }
  }

  leftChild->right = currentNode;
  currentNode->parent = leftChild;

  if (!leftChild->parent) {
    root = leftChild;
    root->color = BLACK;
  }
```

```
}

RBT *findGrandParent(RBT *node) {
  return (node && node->parent) ? node->parent->parent : nullptr;
}

RBT *findUncle(RBT *node) {
  RBT *grandParent = findGrandParent(node);
  if (!grandParent)
    return nullptr;
  return (grandParent->left == node->parent) ? grandParent->right
                                             : grandParent->left;
}

RBT *findSibling(RBT *node) {
  if (!node || !node->parent)
    return nullptr;
  return (node->parent->left == node) ? node->parent->right
                                      : node->parent->left;
}

namespace {
void insertCase1(RBT *&root, RBT *node) {
  if (!node->parent) {
    node->color = BLACK;
  } else {
    insertCase2(root, node);
  }
}

void insertCase2(RBT *&root, RBT *node) {
```

```
  if (node->parent->color == BLACK)
    return;
  insertCase3(root, node);
}

void insertCase3(RBT *&root, RBT *node) {
  RBT *uncle = findUncle(node);
  RBT *grandParent = findGrandParent(node);

  if (uncle && uncle->color == RED && node->parent->color == RED) {
    uncle->color = BLACK;
    node->parent->color = BLACK;
    grandParent->color = RED;
    insertCase1(root, grandParent);
  } else {
    insertCase4(root, node);
  }
}

void insertCase4(RBT *&root, RBT *node) {
  RBT *grandParent = findGrandParent(node);

  if (node == node->parent->right && grandParent->left == node->parent) {
    leftTurn(root, node->parent);
    node = node->left;
  } else if (node == node->parent->left && grandParent->right == node->parent) {
    rightTurn(root, node->parent);
    node = node->right;
  }
```

```
  insertCase5(root, node);
}

void insertCase5(RBT *&root, RBT *node) {
  RBT *grandParent = findGrandParent(node);
  grandParent->color = RED;
  node->parent->color = BLACK;

  if (node == node->parent->left && grandParent->left == node-
>parent) {
    rightTurn(root, grandParent);
  } else {
    leftTurn(root, grandParent);
  }
}
} // namespace

void insert(RBT *&root, RBT *parent, int value) {
  if (value < parent->value) {
    if (!parent->left) {
      parent->left = createNode(value);
      parent->left->parent = parent;
      insertCase1(root, parent->left);
    } else {
      insert(root, parent->left, value);
    }
  } else if (value > parent->value) {
    if (!parent->right) {
      parent->right = createNode(value);
      parent->right->parent = parent;
      insertCase1(root, parent->right);
```

```
    } else {
      insert(root, parent->right, value);
    }
  }
}

namespace {
void deleteCase1(RBT *&root, RBT *node) {
  if (!node->parent) {
    root = (node->left) ? node->left : node->right;
    if (root)
      root->color = BLACK;
  } else {
    deleteCase2(root, node);
  }
}

void deleteCase2(RBT *&root, RBT *node) {
  RBT *sibling = findSibling(node);

  if (sibling && sibling->color == RED) {
    node->parent->color = RED;
    sibling->color = BLACK;

    if (node == node->parent->left) {
      leftTurn(root, node->parent);
    } else {
      rightTurn(root, node->parent);
    }
  }
  deleteCase3(root, node);
```

```
}

void deleteCase3(RBT *&root, RBT *node) {
  RBT *sibling = findSibling(node);

  if (sibling && node->parent->color == BLACK && sibling->color ==
BLACK &&
      (!sibling->left || sibling->left->color == BLACK) &&
      (!sibling->right || sibling->right->color == BLACK)) {
    sibling->color = RED;
    deleteCase1(root, node->parent);
  } else {
    deleteCase4(root, node);
  }
}

void deleteCase4(RBT *&root, RBT *node) {
  RBT *sibling = findSibling(node);

  if (sibling && node->parent->color == RED && sibling->color ==
BLACK &&
      (!sibling->left || sibling->left->color == BLACK) &&
      (!sibling->right || sibling->right->color == BLACK)) {
    sibling->color = RED;
    node->parent->color = BLACK;
  } else {
    deleteCase5(root, node);
  }
}

void deleteCase5(RBT *&root, RBT *node) {
```

```
  RBT *sibling = findSibling(node);

  if (sibling && sibling->color == BLACK) {
    if (node == node->parent->left && sibling->left &&
        sibling->left->color == RED &&
        (!sibling->right || sibling->right->color == BLACK)) {
      sibling->color = RED;
      sibling->left->color = BLACK;
      rightTurn(root, sibling);
    } else if (node == node->parent->right && sibling->right &&
               sibling->right->color == RED &&
               (!sibling->left || sibling->left->color == BLACK)) {
      sibling->color = RED;
      sibling->right->color = BLACK;
      leftTurn(root, sibling);
    }
  }
  deleteCase6(root, node);
}

void deleteCase6(RBT *&root, RBT *node) {
  RBT *sibling = findSibling(node);

  if (sibling) {
    sibling->color = node->parent->color;
    node->parent->color = BLACK;

    if (node == node->parent->left) {
      sibling->right->color = BLACK;
      leftTurn(root, node->parent);
    } else {
```

```
      sibling->left->color = BLACK;
      rightTurn(root, node->parent);
    }
  }
}
} // namespace

namespace {
void replaceNode(RBT *&root, RBT *node) {
  RBT *child = (node->left) ? node->left : node->right;
  child->parent = node->parent;

  if (node->parent) {
    if (node == node->parent->left) {
      node->parent->left = child;
    } else {
      node->parent->right = child;
    }
  }
}

void removeOne(RBT *&root, RBT *node) {
  if (node->left && node->right) {
    RBT *successor = node->left;
    while (successor->right)
      successor = successor->right;
    swap(node->value, successor->value);
    node = successor;
  }

  if (node->left || node->right) {
```

```
    RBT *child = (node->left) ? node->left : node->right;
    replaceNode(root, node);

    if (node->color == BLACK) {
      if (child->color == RED) {
        child->color = BLACK;
      } else {
        deleteCase1(root, node);
      }
    }
  } else {
    if (node->color == BLACK) {
      deleteCase1(root, node);
    }

    if (node->parent) {
      if (node == node->parent->left) {
        node->parent->left = nullptr;
      } else {
        node->parent->right = nullptr;
      }
    }
  }
  delete node;
}
} // namespace

void remove(RBT *&root, RBT *parent, int value) {
  RBT *node = find(root, value);
  if (node)
    removeOne(root, node);
```

```
}

RBT *find(RBT *root, int value) {
  if (!root || root->value == value)
    return root;
  return (value < root->value) ? find(root->left, value)
                               : find(root->right, value);
}

void maxHeight(RBT *root, short &maxDepth, short currentDepth) {
  if (currentDepth > maxDepth)
    maxDepth = currentDepth;
  if (root->left)
    maxHeight(root->left, maxDepth, currentDepth + 1);
  if (root->right)
    maxHeight(root->right, maxDepth, currentDepth + 1);
}

namespace {
string formatNode(RBT *node) {
  if (!node)
    return "    ";
  string val = to_string(node->value);
  return (node->color == RED ? "R" : "B") + val + string(3 -
val.size(), ' ');
}

void printHelper(RBT ***nodes, RBT *current, short depth = 0, short
index = 0) {
  nodes[depth][index] = current;
  if (current->left) {
```

```
    printHelper(nodes, current->left, depth + 1, 2 * index);
  }
  if (current->right) {
    printHelper(nodes, current->right, depth + 1, 2 * index + 1);
  }
}
} // namespace

void print(RBT *root) {
  if (!root)
    return;

  short maxDepth = 1;
  maxHeight(root, maxDepth);
  const short width = 1 << (maxDepth - 1);

  RBT ***nodes = new RBT **[maxDepth];
  for (short i = 0; i < maxDepth; ++i) {
    nodes[i] = new RBT *[width];
    fill_n(nodes[i], width, nullptr);
  }

  printHelper(nodes, root, 0, 0);

  for (short i = 0; i < maxDepth; ++i) {
    short spacing = (1 << (maxDepth - i - 1)) * 4;
    for (short j = 0; j < (1 << i); ++j) {
      cout << setw(j == 0 ? spacing / 2 : spacing) <<
formatNode(nodes[i][j]);
    }
    cout << endl;
```

```
  }

  for (short i = 0; i < maxDepth; ++i)
    delete[] nodes[i];
  delete[] nodes;
}
```

В лучшем случае никакие свойства дерева нарушены не были или произошло перекрашивание за константное время. Временная сложность таких операций составляет O(log n). В худшем случае требуется не более 2 поворотов для восстановления баланса за константное время. Временная сложность составит также O(log n).

Итак, получается, что средний случай тоже O(log n).

В этом и заключается уникальное и отличительное свойство КЧД от обычного бинарного дерева поискам – способность сохранять логарифмическую сложность для всех операций независимо от входных данных