Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

# К У Р С О В А Я  Р А Б О Т А

## Реализация алгоритма удаления узла в красно-чёрном дереве

по дисциплине «Объектно-ориентированное программировани»

Выполнила студентка группы 3331506/80401 Е. А. Барымова
Руководитель Е. М. Кузнецова

«__»_____2021 г

Санкт-Петербург
2021

# Содержание

# 1 Введение

Красно-чёрное дерево – это самобалансирующееся двоичное дерево поиска, которое используется для организации сравнимых данных, таких как фрагменты текста или числа.

Красно-чёрные деревья применяют:

- Большинство самобалансирующихся библиотечных функций двоичных деревьев поиска, таких как map и set в C ++ (OR TreeSet и TreeMap в Java).
- Для реализации планирования ЦП Linux.
- В алгоритме кластеризации K-средних для уменьшения временной сложности.
- В MySQL для индексов таблиц.

# 2 Описание алгоритма

В отличие от обычного двоичного дерева красно-чёрное дерево имеет атрибут цвета и несколько связанных с этим принципов построения:

1. Узел может быть либо красным, либо чёрным и имеет двух потомков;
2. Корень — чёрный.
3. Все листья — чёрные.
4. Оба потомка каждого красного узла — чёрные.
5. Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число чёрных узлов.

Благодаря этому дерево является сбалансированным, но и из-за этого же при удалении и вставке узлов необходимо следить за выполнением свойств.

Если представить все варианты действий в зависимости от положения удаляемого узла в дереве, то можно получить блок-схему, представленную на рисунке 1. В этой схеме также обозначено, какие функции отвечают за те или иные решения. В блок-схеме используются обозначения такие же, как и в программе:

*delNode* – удаляемый узел;

*sibling* – «брат/сестра» – узел, у которого тот же родитель, что и у удаляемого узла;

*parent* – родитель *delNode* и *sibling* (но иногда речь идёт о родителе переданного в функцию узла);

*successor* – «преемник» – узел, который может занять место *delNode*.

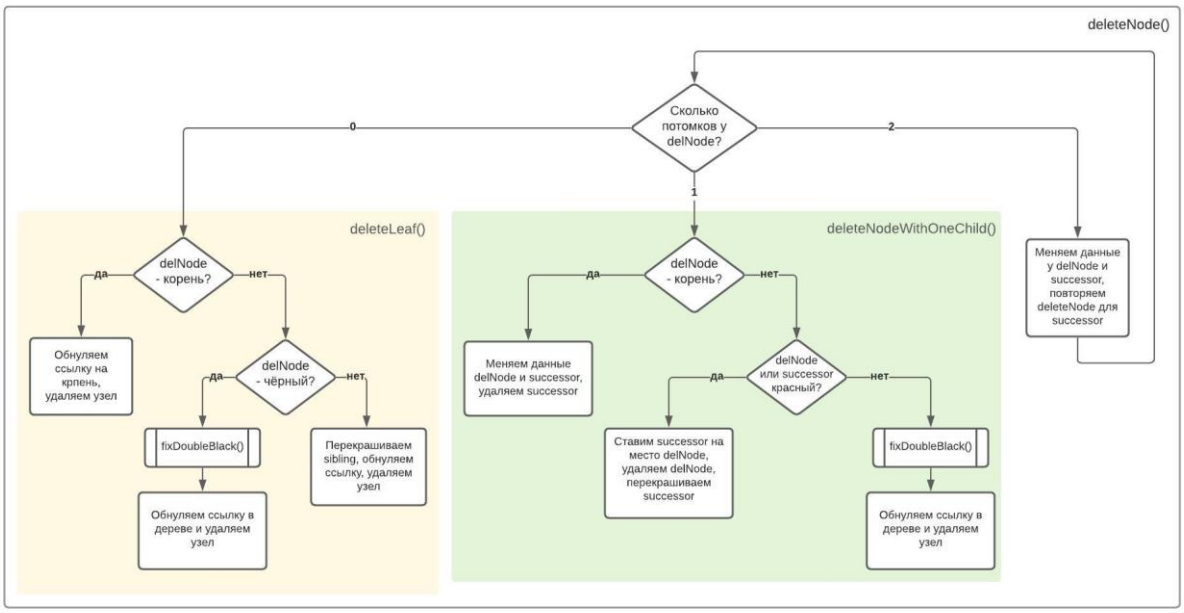


Рисунок 1 – Блок-схема удаления узла из красно-чёрного дерева

В блок-схеме на рисунке 1 часть рассуждений представлена в виде отдельной процедуры fixDoubleBlack(), которая представлена полностью на рисунке 2. В блок-схеме функции fixDoubleBlack() используются две функции, перестраивающие дерево – leftRotate(), rightRotate().

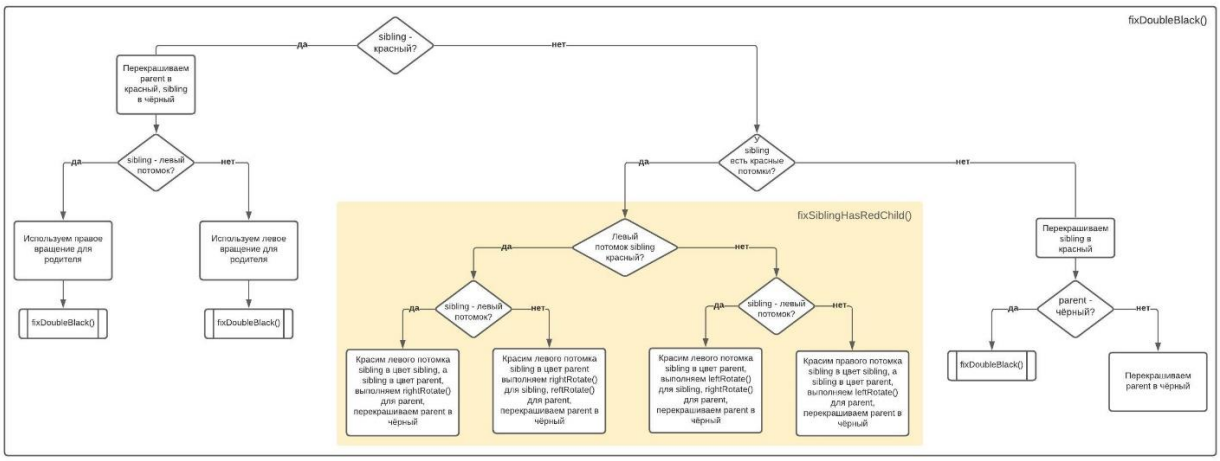


Рисунок 2 – Блок-схема функции fixDoubleBlack()

## 3. Сложность алгоритма

Сложность алгоритма удаления узла из красно-чёрного дерева равна высоте дерева. Поскольку красно-чёрное дерево является сбалансированным, его высота может быть вычислена от количества элементов в дереве - $\boldsymbol{O(\log(n))}$.

# 4. Исследование алгоритма

Результаты измерения зависимости скорости работы алгоритма от количества узлов в красно-чёрном дереве представлены в таблице 1.

Таблица 1 – Время выполнения алгоритма

| Кол-во эл-тов, шт. | 10 | 20 | 50 | 100 | 200 |
|---|---|---|---|---|---|
| Время выполнения, мс | 942 | 955 | 978 | 1005 | 1054 |

Количество элементов выбиралось таким, чтобы график был в полулогарифмическом масштабе. Во всех измерениях есть фиксированная составляющая времени – время на сборку проекта. Однако если посмотреть на график, представленный на рисунке 3, то можно увидеть, что график соответствует логарифмической зависимости.

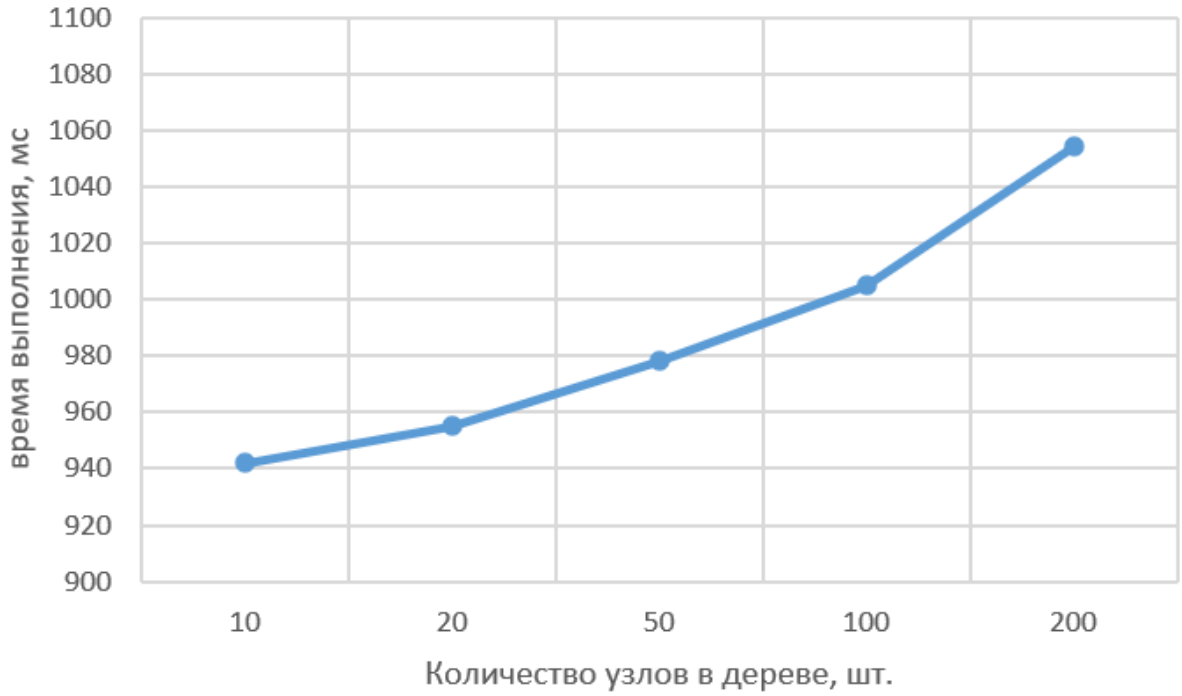


Рисунок 3 – Исследование скорости

# 5. Список литературы


1. Томас Кормен, Алгоритмы: построение и анализ, 3-е изд. : Пер. с англ. – М. : ООО «И. Д. Вильямс», 2013. – 1328 с.

2. Рафгарден Тим, Совершенный алгоритм. Графовые алгоритмы и структуры данных. – СПб.: Питер, 2019. – 256 с.

# 6. Приложение. Код

Файл *“RedBlackTree.h”*.

```h
#ifndef UNTITLED2_REDBLACKTREE_H
#define UNTITLED2_REDBLACKTREE_H

enum COLOR { RED, BLACK };

class Node {
public:
    int data;
    int key;
    COLOR color;
    Node *left, *right, *parent;
public:
    Node();
    Node(int data, int key);
    ~Node();
};

class RBTree {
public:
    Node *root;
private:
    bool isLeft(Node* node);  // Является ли узел левым ребёнокм своего родителя
    bool checkDoubleBlack(Node* node1, Node* node2);  // Функция проверяет условие "двойного чёрного"
    Node* searchSuccessor(Node* node);  // Функция поиска узла, который займёт место удаляемого
    Node* searchSibling(Node* node);  // Функция находит брата узла
    void swapData(Node* node1, Node* node2);  // Функция переставляет местами данные
    void leftRotate(Node* node);  // Левое вращение - меняет положение узлов
    void rightRotate(Node* node);  // Правое вращение - меняет положение узлов
    void fixDoubleBlack(Node* node);  // Функция перестраивающая дерево при "двойном чёрном"
    void fixSiblingHasRedChild(Node* sibling);  // Подфункция функции fixDoubleBlack для случая SiblingIsBlack
    void deleteLeaf(Node* delNode);   // Функция удаления листа
    void deleteNodeWithOneChild(Node* delNode);  // Функция удаления узла с одним потомком
public:
    void deleteNode(Node* node);  // Удаление заданного узла
    void del(int key);  // Удаления узла по ключу
    Node* search(int key);  // Поиск узла по ключу
public:
    RBTree();
    ~RBTree();
};


#endif //UNTITLED2_REDBLACKTREE_H
```

Файл *“RedBlackTree.cpp”*.

```cpp
#include "RedBlackTree.h"
#include <iostream>

Node::Node() {
    left = nullptr;
    right = nullptr;
    color = RED;
}

```

```
Node::Node(int data, int key) {
    parent = nullptr;
    left = nullptr;
    right = nullptr;
    color = RED;
    this->key = key;
    this->data = data;
}

Node::~Node() {
    delete left;
    delete right;
}

RBTree::RBTree() {
    root = nullptr;
}

RBTree::~RBTree() {
    delete root;
}

// Является ли узел левым ребёнокм своего родителя
bool RBTree::isLeft(Node *node) {
    return node->parent->left == node;
}

// Функция проверяет условие "двойного чёрного"
bool RBTree::checkDoubleBlack(Node* node1, Node* node2) {
    return (node1 == nullptr or node1->color == BLACK) and
           (node1 == nullptr or node2->color == BLACK);
}

// Функция поиска узла, который займёт место удаляемого
Node* RBTree::searchSuccessor(Node *node) {
    // Если у удаляемого узла два потомка,
    // то его заменит узел с наименьшим ключом в правом поддереве
    if ((node->left != nullptr) and (node->right != nullptr)) {
        node = node->right;
        while (node->left != nullptr) {
            node = node->left;
        }
    }
```

```
53
54          // Если удаляемый узел - лист,
55          // то возвращаем пустой указатель
56          if ((node->left == nullptr) and (node->right == nullptr)) {
57              return nullptr;
58          }
59
60          // Если у удаляемого узла есть один потомок,
61          // то возвращаем его
62          if (node->left != nullptr) {
63              return node->left;
64          } else {
65              return node->right;
66          }
67      }
68
69      // Функция находит брата узла
70      Node* RBTree::searchSibling(Node *node) {
71          if (node->parent == nullptr) {
72              return nullptr;
73          }
74
75          if (isLeft(node)) {
76              return node->parent->right;
77          } else {
78              return node->parent->left;
79          }
80      }
81
82      // Функция переставляет местами значения узлов
83      void RBTree::swapData(Node* delNode, Node* successor) {
84          int temp = delNode->data;
85          delNode->data = successor->data;
86          successor->data = temp;
87      }
88
89      // Левое вращение - меняет положение узлов
90      void RBTree::leftRotate(Node* node) {
91          Node *pivot = node->right;  // Правый ребёнок node встанет на место node
92          Node* parent = node->parent;
93
94          // Переставляем node и pivot
95          if (node == root) {
96              root = pivot;
97          } else {
98              if (isLeft(node)) {
99                  parent->left = pivot;
```

```
100         } else {
101             parent->right = pivot;
102         }
103     }
104     pivot->parent = parent;
105     pivot->left = node;
106 
107     // Левоый ребёнок pivot становится правым ребёнком node
108     node->right = pivot->left;
109     if (pivot->left != nullptr) {
110         pivot->left->parent = node;
111     }
112 }
113 
114 // Правое вращение - меняет положение узлов
115 void RBTree::rightRotate(Node* node) {
116     Node *pivot = node->left;  // Левый ребёнок node встанет на место node
117     Node* parent = node->parent;
118 
119     // Переставляем node и pivot
120     if (node == root) {
121         root = pivot;
122     } else {
123         if (isLeft(node)) {
124             parent->left = pivot;
125         } else {
126             parent->right = pivot;
127         }
128     }
129     pivot->parent = parent;
130     pivot->right = node;
131 
132     // Правый ребёнок pivot становится левым ребёнком node
133     node->left = pivot->right;
134     if (pivot->right != nullptr) {
135         pivot->right->parent = node;
136     }
137 }
138 
139 // Подфункция функции fixDoubleBlack
140 void RBTree::fixSiblingHasRedChild(Node* sibling) {
141     Node* parent = sibling->parent;
142     if (sibling->left != nullptr and sibling->left->color == RED) {
143         if (isLeft(sibling)) {
144             // left left
145             sibling->left->color = sibling->color;
146             sibling->color = parent->color;
147             rightRotate(parent);
```

```
148         } else {
149             // right left
150             sibling->left->color = parent->color;
151             rightRotate(sibling);
152             leftRotate(parent);
153         }
154     } else {
155         if (isLeft(sibling)) {
156             // left right
157             sibling->right->color = parent->color;
158             leftRotate(sibling);
159             rightRotate(parent);
160         } else {
161             // right right
162             sibling->right->color = sibling->color;
163             sibling->color = parent->color;
164             leftRotate(parent);
165         }
166     }
167     parent->color = BLACK;
168 }
169
170 // Функция перестраивающая дерево при "двойном чёрном"
171 void RBTree::fixDoubleBlack(Node *BBNode) {
172     // Если достигли корня - задача выполнена
173     if (BBNode == root) {
174         return;
175     }
176
177     Node* sibling = searchSibling(BBNode);
178     Node* parent = BBNode->parent;
179
180     // Если нет брата, "двойной чёрный" перемещается на родителя
181     if (sibling == nullptr) {
182         fixDoubleBlack(parent);
183         return;
184     }
185
186     // Если есть брат
187     if (sibling->color == RED) {  // sibling - красный
188         parent->color = RED;
189         sibling->color = BLACK;
190         if (isLeft(sibling)) {
191             rightRotate(parent);
192         } else {
```

```
            leftRotate(parent);
        }
        fixDoubleBlack(BBNode);
    } else {    // sibling - чёрный
        if (checkDoubleBlack(sibling->left, sibling->right)) {  // Оба потомка sibling чёрные
            sibling->color = RED;
            if (parent->color == BLACK)
                fixDoubleBlack(parent);
            else
                parent->color = BLACK;
        } else {  // У sibling есть красные потомки
            fixSiblingHasRedChild(sibling);
        }
    }
}

// Функция удаления листа
void RBTree::deleteLeaf(Node* delNode) {
    Node* parent = delNode->parent;
    Node* successor = searchSuccessor(delNode);  // Находим узел, который займёт место delNode
    bool isDoubleBlack = checkDoubleBlack(delNode, successor);

    if (delNode == root) {  // Если delNode является корнем дерева
        root = nullptr;
    } else {
        if (isDoubleBlack) {      // Если delNode - чёрный
            fixDoubleBlack(delNode);
        } else {                  // Если delNode - красный
            searchSibling(delNode)->color = RED;
        }

        // Обнуляем ссылки в дереве на удаляемый узел
        if (parent->left == delNode) {
            parent->left = nullptr;
        } else {
            parent->right = nullptr;
        }
    }
    delete delNode;
}

// Функция удаления узла с одним потомком
void RBTree::deleteNodeWithOneChild(Node* delNode) {
    Node* parent = delNode->parent;
    Node* successor = searchSuccessor(delNode);  // Находим узел, который займёт место delNode
    bool isDoubleBlack = checkDoubleBlack(delNode, successor);

    if (delNode == root) {  // Если delNode является корнем дерева
        delNode->data = successor->data;
        delNode->left = delNode->right = nullptr;
        delete successor;
    } else {  // Если delNode не корень
        // Меняем ссылки
        if (isLeft(delNode)) {
            parent->left = successor;
        } else {
            parent->right = successor;
        }
        successor->parent = parent;

```

```
253         // Удаляем узел
254         delete delNode;
255 
256         if (isDoubleBlack) {  // delNode и successor чёрные
257             fixDoubleBlack(successor);
258         } else {              // Или delNode, или successor красный
259             // Делаем новый узел чёрным
260             successor->color = BLACK;
261         }
262     }
263 }
264 
265 // Ищет узел по заданному ключу
266 Node* RBTree::search(int key) {
267     Node *temp = RBTree::root;
268     while (temp != nullptr) {
269         if (key < temp->key) {  // Если искомое меньше найденного
270             if (temp->left == nullptr) {  // Если нет левого потомка,
271                 break;                    // temp - ближайший по знач ключа
272             } else {
273                 temp = temp->left;        // проверяем левого потомка найденного узла
274             }
275         } else if (key == temp->key) {  //Значения равны - нашли нужный ключ
276             break;
277         } else {  // Если искомое больше найденного
278             if (temp->right == nullptr) {  // Если нет правого потомка,
279                 break;                     // temp - ближайший по знач ключа
280             } else {
281                 temp = temp->right;  // проверяем правого потомка найденного узла
282             }
283         }
284     }
285     return temp;
286 }
287 
288 // Удаляет переданный узел
289 void RBTree::deleteNode(Node *delNode) {
290     Node* successor = searchSuccessor(delNode);  // Находим узел, который займёт место delNode
291 
292     if (successor == nullptr) {  // Если delNode - лист
293         deleteLeaf(delNode);
294     } else if (delNode->left == nullptr or delNode->right == nullptr) {  // Если у delNode один потомок
295         deleteNodeWithOneChild(delNode);
296     } else {  // Если у delNode два потомка
297         swapData(delNode, successor);
298         deleteNode(successor);
299     }
300 }
301 
302 void RBTree::del(int key) {
303     if (RBTree::root == nullptr) {  // Деревое пустое, нечего удалять
304         return;
305     }
306 
307     // Ищем узел с таким же или ближайшим ключом
308     Node *node = search(key);
309     if (node->key != key) {
310         std::cout << "Нет узлов с значением: " << key << std::endl;
311         return;
312     }
313 
314     // Передаём узел в функцию удаления узла
315     deleteNode(node);
316 }
```