

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

КУРСОВАЯ РАБОТА

Реализация алгоритма удаления узла в красно-чёрном дереве
по дисциплине «Объектно-ориентированное программирование»

Выполнила студентка группы 3331506/80401

Е.А. Барымова

Руководитель

М.С. Ананьевский

«__»_____2021 г

Санкт-Петербург

2021

Содержание

1. Введение	3
2. Описание алгоритма	4
3. Сложность алгоритма	6
4. Исследование алгоритма	7
5. Список литературы	8
Приложение. Код	9

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 — как правило чёрный. Это правило слабо влияет на работоспособность модели, так как цвет корня всегда можно изменить с красного на чёрный;
3. Все листья, не содержащие данных — 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*.

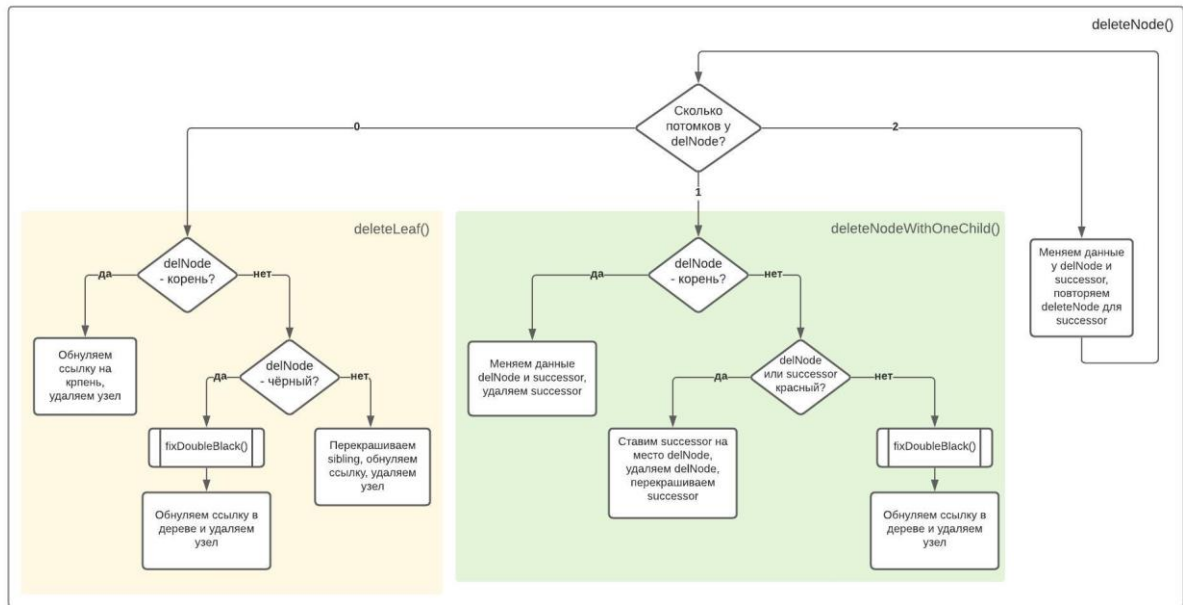


Рисунок 1 – Блок-схема удаления узла из красно-чёрного дерева

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()`.

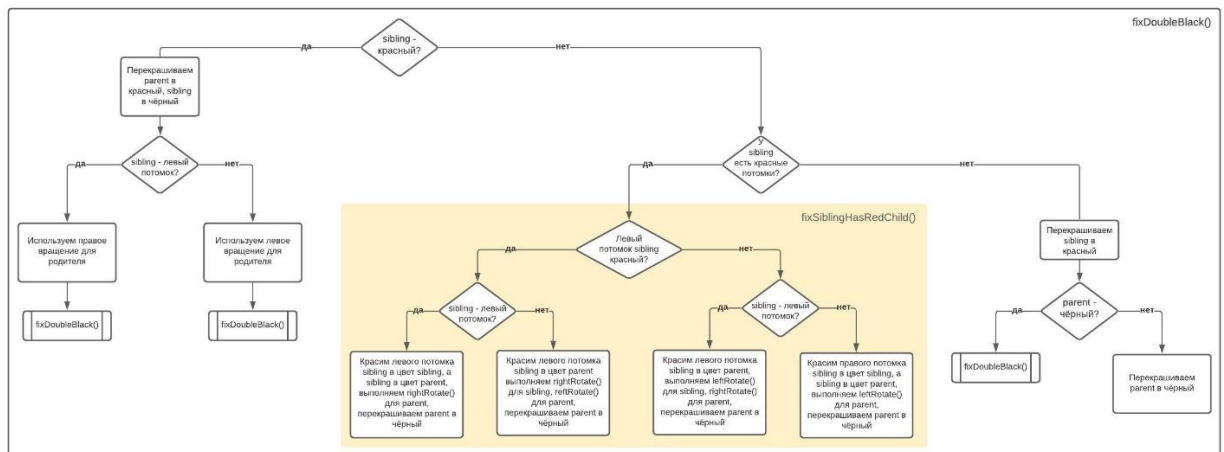


Рисунок 2 – Блок-схема функции `fixDoubleBlack()`

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 - $O(\log(n))$.

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.

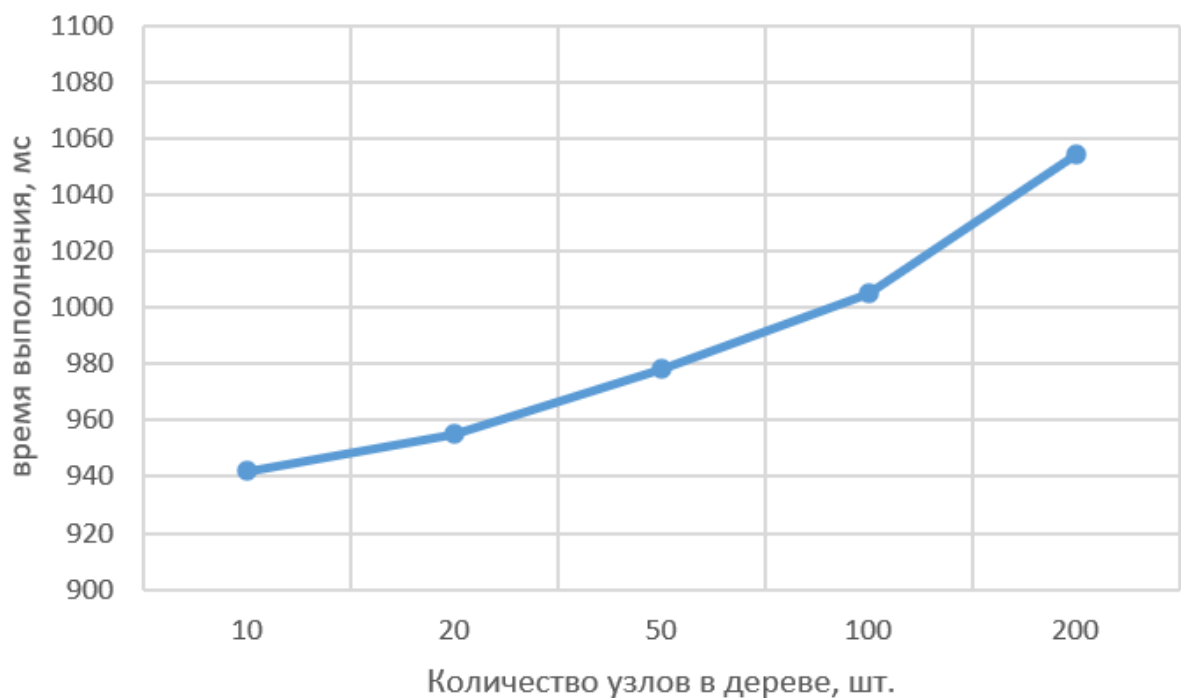


Рисунок 3 – Исследование скорости

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”.

```
1  #ifndef UNTITLED2_REDBLACKTREE_H
2  #define UNTITLED2_REDBLACKTREE_H
3
4  enum COLOR { RED, BLACK };
5
6  class Node {
7  public:
8      int data;
9      int key;
10     COLOR color;
11     Node *left, *right, *parent;
12
13  public:
14     Node();
15     Node(int data, int key);
16     ~Node();
17 };
18
19 class RBTTree {
20 public:
21     Node *root;
22 private:
23     bool isLeft(Node* node); // Является ли узел левым ребёнком своего родителя
24     bool checkDoubleBlack(Node* node1, Node* node2); // Функция проверяет условие "двойного чёрного"
25     Node* searchSuccessor(Node* node); // Функция поиска узла, который займёт место удаляемого
26     Node* searchSibling(Node* node); // Функция находит брата узла
27     void swapData(Node* node1, Node* node2); // Функция переставляет местами данные
28     void leftRotate(Node* node); // Левое вращение - меняет положение узлов
29     void rightRotate(Node* node); // Правое вращение - меняет положение узлов
30     void fixDoubleBlack(Node* node); // Функция перестраивающая дерево при "двойном чёрном"
31     void fixSiblingHasRedChild(Node* sibling); // Подфункция функции fixDoubleBlack для случая SiblingIsBlack
32     void deleteLeaf(Node* delNode); // Функция удаления листа
33     void deleteNodeWithOneChild(Node* delNode); // Функция удаления узла с одним потомком
34 public:
35     void deleteNode(Node* node); // Удаление заданного узла
36     void del(int key); // Удаления узла по ключу
37     Node* search(int key); // Поиск узла по ключу
38
39 public:
40     RBTTree();
41     ~RBTTree();
42 };
43 #endif // UNTITLED2_REDBLACKTREE_H
```

Файл “RedBlackTree.cpp”.

```
1  #include "RedBlackTree.h"
2  #include <iostream>
3
4  Node::Node() {
5      left = nullptr;
6      right = nullptr;
7      color = RED;
8  }
9
```

```

10  ➡ Node::Node(int data, int key) {
11      parent = nullptr;
12      left = nullptr;
13      right = nullptr;
14      color = RED;
15      this->key = key;
16      this->data = data;
17  }
18
19  ➡ Node::~~Node() {
20      delete left;
21      delete right;
22  }
23
24  ➡ RBTREE::RBTREE() {
25      root = nullptr;
26  }
27
28  ➡ RBTREE::~~RBTREE() {
29      delete root;
30  }
31
32      // Является ли узел левым ребёнком своего родителя
33  ➡ bool RBTREE::isLeft(Node *node) {
34      return node->parent->left == node;
35  }
36
37      // Функция проверяет условие "двойного чёрного"
38  ➡ bool RBTREE::checkDoubleBlack(Node* node1, Node* node2) {
39      return (node1 == nullptr or node1->color == BLACK) and
40             (node1 == nullptr or node2->color == BLACK);
41  }
42
43      // Функция поиска узла, который займёт место удаляемого
44  ➡ Node* RBTREE::searchSuccessor(Node *node) {
45      // Если у удаляемого узла два потомка,
46      // то его заменит узел с наименьшим ключом в правом поддереве
47      if ((node->left != nullptr) and (node->right != nullptr)) {
48          node = node->right;
49          while (node->left != nullptr) {
50              node = node->left;
51          }
52      }

```

```

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     // Левый 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 {

```

```

193         leftRotate(parent);
194     }
195     fixDoubleBlack(BBNode);
196 } else { // sibling - чёрный
197     if (checkDoubleBlack(sibling->left, sibling->right)) { // Оба потомка sibling чёрные
198         sibling->color = RED;
199         if (parent->color == BLACK)
200             fixDoubleBlack(parent);
201         else
202             parent->color = BLACK;
203     } else { // У sibling есть красные потомки
204         fixSiblingHasRedChild(sibling);
205     }
206 }
207 }
208
209 // Функция удаления листа
210 void RBTREE::deleteLeaf(Node* delNode) {
211     Node* parent = delNode->parent;
212     Node* successor = searchSuccessor(delNode); // Находим узел, который займёт место delNode
213     bool isDoubleBlack = checkDoubleBlack(delNode, successor);
214
215     if (delNode == root) { // Если delNode является корнем дерева
216         root = nullptr;
217     } else {
218         if (isDoubleBlack) { // Если delNode - чёрный
219             fixDoubleBlack(delNode);
220         } else { // Если delNode - красный
221             searchSibling(delNode->color = RED;
222         }
223
224         // Обнуляем ссылки в дереве на удаляемый узел
225         if (parent->left == delNode) {
226             parent->left = nullptr;
227         } else {
228             parent->right = nullptr;
229         }
230     }
231     delete delNode;
232 }
233
234 // Функция удаления узла с одним потомком
235 void RBTREE::deleteNodeWithOneChild(Node* delNode) {
236     Node* parent = delNode->parent;
237     Node* successor = searchSuccessor(delNode); // Находим узел, который займёт место delNode
238     bool isDoubleBlack = checkDoubleBlack(delNode, successor);
239
240     if (delNode == root) { // Если delNode является корнем дерева
241         delNode->data = successor->data;
242         delNode->left = delNode->right = nullptr;
243         delete successor;
244     } else { // Если delNode не корень
245         // Меняем ссылки
246         if (isLeft(delNode)) {
247             parent->left = successor;
248         } else {
249             parent->right = successor;
250         }
251         successor->parent = parent;
252     }

```

```

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 }

```