

Санкт-Петербургский Политехнический университет Петра Великого

Институт машиностроения, материалов и транспорта

Кафедра «Мехатроника и роботостроение (при ЦНИИ РТК)»

Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Тема: Красно-черное дерево. Добавление

Разработал:

ст. гр. 3331506/80401 Калясова С. С.

Преподаватель

Кузнецова Е. М.

Санкт-Петербург

2021 г

Содержание

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

Введение

Красно-чёрное дерево - двоичное дерево поиска, в котором баланс осуществляется на основе "цвета" узла дерева, который принимает только два значения: "красный" и "чёрный". Такие деревья являются наиболее часто используемыми сбалансированными деревьями двоичного поиска, и они гарантированно поддерживают вставку, удаление и поиск.

Красно-чёрное дерево используется для организации сравнимых данных, таких как фрагменты текста или числа.

Применение:

- Ява: `java.util.TreeMap`, `java.util.TreeSet`
- С ++ STL (в большинстве реализаций): `map`, `multimap`, `multiset`
- Ядро Linux: полностью честный планировщик, `linux / rbtree.h`

Описание алгоритма

Красно-черное дерево является "достаточно сбалансированным". Т.е. путь от вершины дерева до любого листа дерева занимает примерно одинаковое время. Длины поддеревьев могут отличаться больше, чем на единицу, но никогда не больше, чем в два раза.

Для красно-черного дерева должны выполняться следующие свойства:

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.

При этом для удобства, листьями красно-чёрного дерева считаются фиктивные «нулевые» узлы, не содержащие данных. Пример данного дерева представлен на рисунке 1.

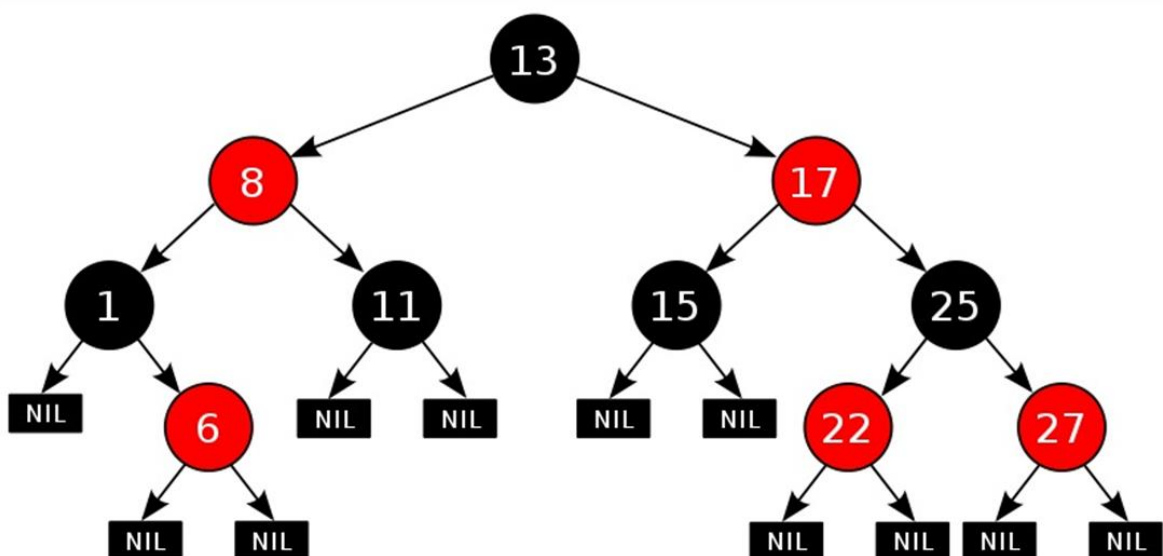


Рисунок 1 — Красно-черное дерево

При добавлении в дерево новый узел окрашивается в «красный» цвет. Далее проверяются правила, приведённые выше; если они нарушаются, дерево балансируется.

Количество черных узлов на ветви от корня до листа называется черной высотой дерева.

Сложность алгоритма

Красно-чёрные деревья имеют вычислительную сложность поиска, одинаковую с АВЛ-деревьями, а именно $O(\ln N)$. Причём поддержание структуры в «оптимальной форме» требует на порядок меньших усилий. Для данной структуры требуется память, кратная количеству элементов, следовательно, пространственная сложность алгоритма линейна — $O(N)$, как и в случае бинарного дерева поиска.

Исследование алгоритма

Если посчитать время заполнения красно-черного дерева различным количеством данных, то получим результаты, приведенные в таблице 1.

Таблица 1- Время выполнения алгоритма

Число элементов, шт.	10	10^2	$5 \cdot 10^2$	10^3	$2 \cdot 10^3$	$5 \cdot 10^3$	$7 \cdot 10^3$	10^4
Время выполнения, мс.	24	323	546	1190	2266	4502	7397	11970

На рисунке 2 приведен график времени выполнения алгоритма в зависимости от количества данных.

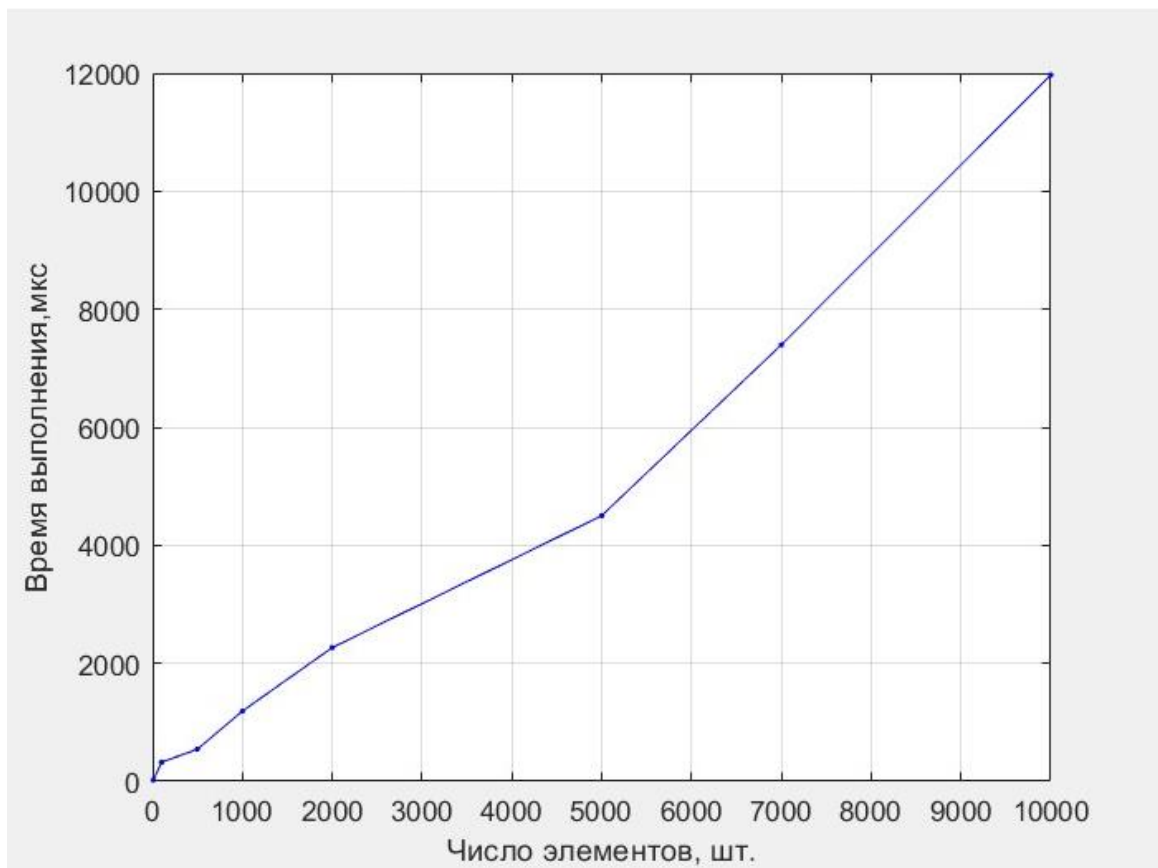


Рисунок 2 – Численный анализ

По графику видно, что в моей реализации алгоритма при больших значениях сложность приближается к линейной. Это происходит из-за рекурсивных вызовов родителя и дедушки в алгоритме.

Список литературы

1. Роберт Седжвик, Алгоритмы на C++. Фундаментальные алгоритмы и структуры данных / пер. с англ. (Algorithms in C++) — М.: Вильямс. — 2011 г. — 1056 с
2. Томас Кормен, Алгоритмы: построение и анализ / пер. с англ. (Introduction to Algorithms) — А.: Шеня. — 2002 г. — 955 с

Приложение 1. Код программы

```
1  #include <iostream>
2  #include <string>
3  #include <chrono>
4
5
6  class Node {
7  public:
8      int key;
9      std::string data;
10     bool red;
11
12     public:
13         Node *left;
14         Node *right;
15
16     public:
17         Node();
18         Node(int key, std::string data);
19         ~Node();
20 };
21
22 class Tree {
23 private:
24     Node *root;
25     Node *findNode(Node* root, int key);
26     Node *findParent(Node * root, int key);
27     static Node *findUncle(Node *root, int childKey, Node* parent, Node* grParent);
28     void recurseAdding (Node* child, Node* root);
29     void rotateRight (Node* node);
30     void rotateLeft (Node* node);
31     void outputTree (Node *root, int totalSpace);
32
33 public:
34     void printTree();
35     bool add(const int key, std::string data);
36     std::string find(const int key);
37
38 public:
39     Tree();
40     ~Tree();
41 };
42
43 Node::Node() {
44     left = nullptr;
45     right = nullptr;
46     red = true;
47 }
48
49 Node::Node(int key, std::string data) {
50     this->key = key;
51     this->data = data;
52     left = nullptr;
53     right = nullptr;
54     red = true;
55 }
```

```

54 Node::~Node() {
55     delete left;
56     delete right;
57 }
58
59 Tree::Tree() {
60     root = nullptr;
61 }
62
63 Tree::~Tree() {
64     delete root;
65 }
66
67 void Tree::outputTree(Node *root, int level) {
68     if(root)
69     {
70         outputTree(root->left, level: level + 1);
71         for(int i = 0; i < level; i++) std::cout << "  ";
72         if (root->red == true)
73             printf( format: "r");
74         if (root->red == false)
75             printf( format: "b");
76         std::cout << root->key << std::endl;
77         outputTree(root->right, level: level + 1);
78     }
79 }
80
81
82 void Tree::printTree() {
83     outputTree(root, level: 0);
84 }
85
86 void Tree::rotateRight (Node *node) {
87     Node *grgrParent = findParent(root, node->key);
88     Node *parent = node->left;
89     if (grgrParent != nullptr){
90         if (grgrParent->right == node)
91             grgrParent->right = parent;
92         else
93             grgrParent->left = parent;
94     }
95     node->left = parent->right;
96     parent->right = node;
97 }
98
99 void Tree::rotateLeft(Node *node) {
100     Node *grgrParent = findParent(root, node->key);
101     Node *parent = node->right;
102     if (grgrParent != nullptr){
103         if (grgrParent->right == node)
104             grgrParent->right = parent;
105         else
106             grgrParent->left = parent;
107     }

```

```

108     node->right = parent->left;
109     parent->left = node;
110 }
111
112 Node* Tree::findNode(Node *root, int key) {
113     if (root == nullptr) return root;
114     if (root->key == key) return root;
115     if (root->key > key) return findNode(root->left, key);
116     return findNode(root->right, key);
117 }
118
119 Node* Tree::findParent(Node *root, int key){
120     if (root->key < key) {
121         if (root->right == nullptr)
122             return root;
123         if (root->right->key == key)
124             return root;
125         return findParent(root->right, key);
126     }
127
128     if (root->key > key) {
129         if (root->left == nullptr)
130             return root;
131         if (root->left->key == key)
132             return root;
133         return findParent(root->left, key);
134     }
135     return nullptr;
136 }
137
138 Node* Tree::findUncle(Node *root, int childKey, Node* parent, Node *grParent) {
139     if (parent == nullptr)
140         return nullptr;
141     if (grParent == nullptr)
142         return nullptr;
143     if (grParent->left == parent)
144         return grParent->right;
145     else
146         return grParent->left;
147 }
148
149 std::string Tree::find(const int key){
150     Node* temp = root;
151     temp = findNode(temp, key);
152     if (temp == nullptr)
153         return "nothing find";
154     return temp->data;
155 }
156
157 void Tree::recurseAdding(Node* child, Node* root) {
158     root->red = false;
159     child->red = true;
160     Node* parent = findParent(root, child->key);
161
162     if (parent == nullptr){
163         child->red = false;

```

```

164         return;
165     }
166
167     if (!(parent->red)) {
168         return;
169     }
170
171     Node* grParent = findParent(root, parent->key);
172     Node* uncle = findUncle(root, child->key, parent, grParent);
173
174     if (parent->red) {
175         if ((uncle != nullptr) && uncle->red) {
176             parent->red = false;
177             uncle->red = false;
178             grParent->red = true;
179             return recurseAdding(grParent, root);
180         }
181
182         if ((uncle == nullptr) || !(uncle->red)) {
183             if (grParent->left == parent){
184                 if (parent->right == child){
185                     rotateLeft(parent);
186                     child = grParent->left->left;
187                     parent = findParent(root, child->key);
188                     grParent = findParent(root, parent->key);
189                 }
190                 parent->red = false;
191                 grParent->red = true;
192                 rotateRight(grParent);
193                 root->red = false;
194                 return;
195             }
196
197             if (grParent->right == parent){
198                 if (parent->left == child){
199                     rotateRight(parent);
200                     child = grParent->right->right;
201                     parent = findParent(root, child->key);
202                     grParent = findParent(root, parent->key);
203                 }
204                 parent->red = false;
205                 grParent->red = true;
206                 rotateLeft(grParent);
207                 root->red = false;
208                 return;
209             }
210         }
211     }
212 }
213
214 bool Tree::add(const int key, std::string data) {
215     Node* child = new Node (key,data);
216
217     if (findNode(root, key) != nullptr)
218         return false;
219

```

```

220     if (root == nullptr){
221         child->red = false;
222         root = child;
223         return true;
224     }
225
226     Node* parent = findParent(root, key);
227
228     if (key <= parent->key)
229         parent->left = child;
230     else
231         parent->right = child;
232
233     recurseAdding(child, root);
234     return true;
235 }
236
237 int main() {
238     Tree testTree;
239     testTree.add( key: 20, data: "1");
240     testTree.add( key: 86, data: "2");
241     testTree.add( key: 35, data: "3");
242     testTree.add( key: 13, data: "4");
243     testTree.add( key: 62, data: "5");
244     testTree.add( key: 100, data: "6");
245     testTree.add( key: 48, data: "7");
246     testTree.add( key: -15, data: "8");
247     testTree.add( key: 0, data: "9");
248     testTree.add( key: 5, data: "10");
249     testTree.printTree();
250 }

```