ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Студент гр.3331506/90401: Гаврилов Д.В.

Преподаватель: Ананьевский А. С.

Санкт-Петербург

2022

Оглавление

Введение

Красно-чёрное дерево(кчд) — один из видов самобалансирующихся двоичных деревьев поиска, в котором ноды окрашиваются в красный или в черный цвет. КЧД является потомком 2-3 дерева, мы просто заменяем 3-ноды на две 2-ноды и пометим их цветом, в нашем случае красным. Так же красное дерево обладает 4 свойствами.

Свойства:

1. *Две красные ноды не могут идти подряд*
2. *Корень дерева всегда черный*
3. *Все null-ноды (ноды, которые не имеют потомков) – черные*
4. *Высота дерева измеряется только по черным нодам и называется "черной высотой"*

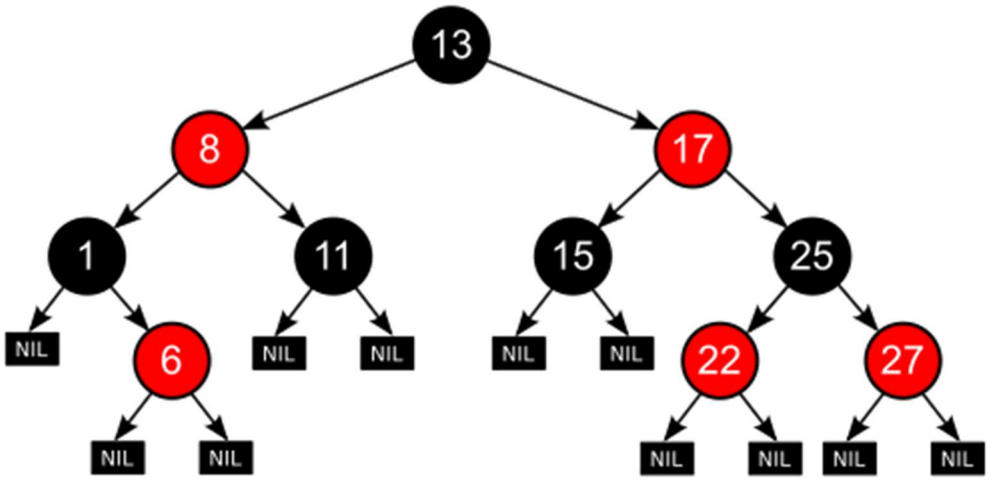


Рисунок 1 – Красно черное дерево

Описание алгоритма

Вставка

Вставка в красно-черном дереве элементы вставляются в позиции NULL-листьев. Вставленный узел всегда окрашивается в красный цвет. Далее идет процедура проверки сохранения свойств красно-черного дерева 1-4.

Если нарушаются какие-то из правил у нас есть три метода которые помогут нам решить эту проблему. Это операции поворота, левый и правый, и третья операция это перекрашивание узлов.

Так же нас есть 3 метода восстановления КЧД, то есть его балансировки, теперь опишу в каких ситуациях использовать какие методы

1. *Родительский узел текущего узла имеет красный цвет, а другой дочерний узел (узел дяди) узла-дедушки текущего узла также имеет красный цвет.*
    a. Установите «родительский узел» в черный цвет.
    b. Установите «Узел дяди» на черный.
    c. Установите «дедушкин узел» на «красный».
    d. Установите для «дедушкины узлы» значение «текущий узел» (красный узел), то есть продолжайте работать на «текущем узле» впоследствии.
2. *Родительский узел текущего узла красный, дядя - черный, а текущий узел - правый дочерний узел своего родительского узла.*
    a. Используйте «родительский узел» в качестве «нового текущего узла»
    b. Поверните налево с «новым текущим узлом» в качестве точки опоры.
3. *Родительский узел текущего узла красный, дядя - черный, а текущий узел - левый потомок его родительского узла.*
    a. Установите «родительский узел» на «черный».
    b. Установите «дедушкин узел» на «красный»

c. Поверните направо с «дедушкиным узлом» в качестве точки опоры.

Удаление

Для удаления необходимо выполнить следующие операции, сначала удалить узел как в двоичном дереве поиска, а потом нужно восстановить свойства дерева до красно-черного.

1 этап – удаление как для двоичного дерева

Есть 3 случая

1. Узел не имеет детей – просто удаляем
2. У узла есть только один сын – он встает на место удаленного узла
3. У узла есть 2 наследника

Дальше нужно восстановить красно черное дерево, так как при удалении у нас могло нарушиться 1, 2, 4 свойство, рассмотрим возможные ситуации

1. *x - это «черный + черный» узел, а родственный узел у x красный. (В это время родительский узел x и дочерние узлы дочерних узлов x оба являются черными узлами)..*
    a. Установите родной узел x на «черный».
    b. Установите родительский узел x на «красный».
    c. Установите «дедушкин узел» на «красный».
    d. После поворота налево сбросьте узел-брат x.
2. *x - это «черный + черный» узел, одноуровневый узел x черный, а оба дочерних узла x - черный.*
    a. Установите родственный узел x на «красный».
    b. Установите «родительский узел x» на «новый узел x».

3. *x - это «черный + черный» узел, одноуровневый узел x черный, левый дочерний узел брата x красный, а правый черный. (На самом деле, здесь необходимо различать, является ли x левым или правым поддеревом родительского узла. Если x является левым поддеревом, оно будет обработано в соответствии с приведенными здесь правилами. Если x является правым поддеревом, оно изменится слева направо и справа налево)*

    a. Установите левый дочерний узел узла «x брат» на «черный»

    b. становите узел «x брат» на «красный»

    c. Правое вращение узла сестры x.

    d. Повернув направо, сбросьте родной узел x

4. *x является узлом "черный + черный", одноуровневый узел x черный, правый дочерний узел брата x красный, а левый дочерний узел брата x любого цвета. (Там же.)*

    a. Назначьте цвет родительского узла x узлу-брату x.

    b. Установите x родительский узел на «черный».

    c. Установите правый подраздел узла «x брат» на «черный».

    d. Левостороннее вращение родительского узла x.

    e. Установите «x» в «корневой узел».

Исследование алгоритма

У КЧД есть свойство связанное с глубиной дерева, по условию оно не может быть больше 1, значит что при минимальном количестве красных нодов глубина равна кол-ву черных, а если максимальное то в два раза больше.

Значит что путь от корня до листа не может различаться не более чем в двое.

$$\boldsymbol{h \leq 2log(N+1)}$$

Получаем что сложность поиска в этом дереве $\boldsymbol{O(logN)}$.

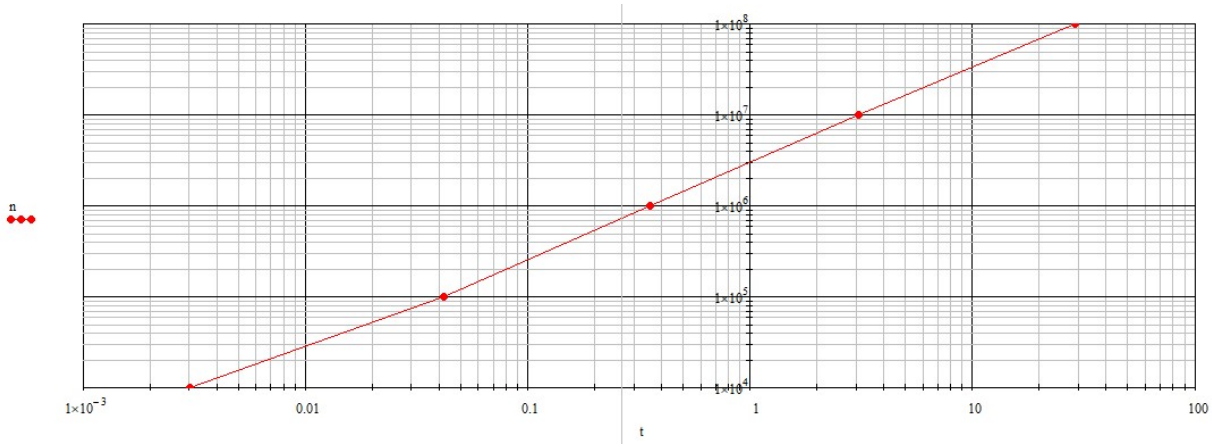
Так же проведем замер времени вставки в красно-черное дерево, для этого напишем алгоритм, который будет создавать каждый раз КЧД увеличивая узлы в 10 раз. Полученные данные отобразим на графике, где обе оси имеют логарифмический масштаб чтобы были видны все полученные значения. Как видно по графику – время растет линейно.

```
    uint64_t startTime, endTime;
    int degree = 2;
    //srand(time(NULL));
    int key;
    for(uint64_t j = 100; j <= 100000000; j*=10)
    {
        RedBlackTree tree;
        startTime = clock();
        for(int i = 0; i < j; i++){
            key = rand();
            tree.insert(key);
        }

        endTime = clock();
        cout <<"10^"<< degree << ": "<< (double)(endTime - startTime)/CLK_TCK << "seconds.\n";
        degree ++;
        tree.deleteAll();
    }
    return 0;
```

```
main
redblackTree
C:\Users\dimag\CLionProjects\redblackTree\cmake-build-debug\redblackTree.exe
10^2: 0seconds.
10^3: 0seconds.
10^4: 0.003seconds.
10^5: 0.042seconds.
10^6: 0.353seconds.
10^7: 3.046seconds.
10^8: 28.813seconds.

Process finished with exit code 0
```



Заключение

В работе был рассмотрен пример ассоциативного массива – красно черное дерево. Например, данное дерево используется в STL для контейнера map.

Анализ дерева показал, что оно обеспечивает логорафмическую сложность поиска, что важно что оно такое как и в среднем так и в худшем случае, а так же линейную сложность при создании, и как следствие при балансировки.

Список литературы

- Т. Кормен, Ч. Лейсерзон, Р. Риверст, К. Штайн - Алгоритмы. Построение и анализ (2013)
- Понимаем красно-черное дерево часть 1 и 2, Хабр URL: https://habr.com/ru/post/555404/ , https://habr.com/ru/post/557328/
- Красно черное дерево, Википедия URL: https://ru.wikipedia.org/wiki/Красно-черное_дерево

Приложение

```
 1 #include <iostream>
 2 #include <string>
 3 #include <time.h>
 4 #include "rbtree.h"
 5 using namespace std;
 6
 7     void RedBlackTree::initializeNULLNode(NodePtr
   node, NodePtr parent) {
 8         node->data = 0;
 9         node->parent = parent;
10         node->left = nullptr;
11         node->right = nullptr;
12         node->color = Black;
13     }
14
15     NodePtr RedBlackTree::searchTreeHelper(NodePtr
   node, int key) {
16         if (node == TNULL || key == node->data)
   return node;
17
18         if (key < node->data) return searchTreeHelper
   (node->left, key);
19
20         return searchTreeHelper(node->right, key);
21     }
22
23     void RedBlackTree::deleteFix(NodePtr x) {
24         NodePtr s;
25         while (x != root && x->color == Black) {
26             if (x == x->parent->left) {
27                 s = x->parent->right;
28                 if (s->color == Black) {
29                     s->color = Black;
30                     x->parent->color = Red;
31                     leftRotate(x->parent);
32                     s = x->parent->right;
33                 }
34
35                 if (s->left->color == Black && s->
   right->color == Black) {
36                     s->color = Red;
```

```
37                      x = x->parent;
38                  }
39                  else {  // случай 1
40                      if (s->right->color == Black) {
41                          s->left->color = Black;
42                          s->color = Red;
43                          rightRotate(s);
44                          s = x->parent->right;
45                      }
46
47                      s->color = x->parent->color;
48                      x->parent->color = Black;
49                      s->right->color = Black;
50                      leftRotate(x->parent);
51                      x = root;
52                  }
53
54              }
55              else {
56                  s = x->parent->left;
57                  if (s->color == Red) {
58                      s->color = Black;
59                      x->parent->color = Red;
60                      rightRotate(x->parent);
61                      s = x->parent->left;
62                  }
63
64                  if (s->right->color == Black) {
65                      s->color = Red;
66                      x = x->parent;
67                  }
68                  else {
69                      if (s->left->color == Black) {
70                          s->right->color = Black;
71                          s->color = Red;
72                          leftRotate(s);
73                          s = x->parent->left;
74                      }
75
76                      s->color = x->parent->color;
77                      x->parent->color = Black;
```

```
 78                      s->left->color = Black;
 79                      rightRotate(x->parent);
 80                      x = root;
 81                  }
 82              }
 83          }
 84          x->color = Black;
 85      }
 86
 87      void RedBlackTree::rbTransplant(NodePtr u,
    NodePtr v) {
 88          if (u->parent == nullptr) {
 89              root = v;
 90          }
 91          else if (u == u->parent->left) {
 92              u->parent->left = v;
 93          }
 94          else {
 95              u->parent->right = v;
 96          }
 97          v->parent = u->parent;
 98      }
 99
100      void RedBlackTree::deleteNodeHelper(NodePtr node
    , int key) {
101          NodePtr z = TNULL;
102          NodePtr x, y;
103          while (node != TNULL) { // поиск нужного
    узла
104              if (node->data == key) {
105                  z = node;
106              }
107
108              if (node->data <= key) {
109                  node = node->right;
110              }
111              else {
112                  node = node->left;
113              }
114          }
115
```

```
116         if (z == TNULL) { // не нашел нужный узел
117             return;
118         }
119 
120         y = z;
121         int y_original_color = y->color;
122         if (z->left == TNULL) {
123             x = z->right;
124             rbTransplant(z, z->right);
125         }
126         else if (z->right == TNULL) {
127             x = z->left;
128             rbTransplant(z, z->left);
129         }
130         else {
131             y = minimum(z->right);
132             y_original_color = y->color;
133             x = y->right;
134             if (y->parent == z) {
135                 x->parent = y;
136             }
137             else {
138                 rbTransplant(y, y->right);
139                 y->right = z->right;
140                 y->right->parent = y;
141             }
142 
143             rbTransplant(z, y);
144             y->left = z->left;
145             y->left->parent = y;
146             y->color = z->color;
147         }
148         delete z;
149         if (y_original_color == 0) {
150             deleteFix(x);
151         }
152     }
153 
154     void RedBlackTree::insertFix(NodePtr k) {
155         NodePtr u;
156         while (k->parent->color == Red) {
```

```
157             if (k->parent == k->parent->parent->
    right) {
158                 u = k->parent->parent->left;
159                 if (u->color == Red) { // case 1
160                     u->color = Black;
161                     k->parent->color = Black;
162                     k->parent->parent->color = Red;
163                     k = k->parent->parent;
164                 }
165                 else {
166                     if (k == k->parent->left) {
167                         k = k->parent;
168                         rightRotate(k);
169                     }
170                     k->parent->color = Black; //
    случай 2
171                     k->parent->parent->color = Red;
172                     leftRotate(k->parent->parent);
173                 }
174             }
175             else {
176                 u = k->parent->parent->right;
177 
178                 if (u->color == Red) {
179                     u->color = Black;
180                     k->parent->color = Black;
181                     k->parent->parent->color = Red;
182                     k = k->parent->parent;
183                 }
184                 else {
185                     if (k == k->parent->right) {
186                         k = k->parent;
187                         leftRotate(k);
188                     }
189                     k->parent->color = Black;
190                     k->parent->parent->color = Red;
191                     rightRotate(k->parent->parent);
192                 }
193             }
194             if (k == root) {
195                 break;
```

```
196             }
197         }
198         root->color = Black;
199     }
200 
201     void RedBlackTree::printHelper(NodePtr root,
    string indent, bool last) {
202         if (root != TNULL) {
203             cout << indent;
204             if (last) {
205                 cout << " R----";
206                 indent += "   ";
207             }
208             else {
209                 cout << " L----";
210                 indent += " |  ";
211             }
212 
213             string sColor = root->color ? "R" : "B";
214             cout << root->data << "(" << sColor <<
    ")" << endl;
215             printHelper(root->left, indent, false);
216             printHelper(root->right, indent, true);
217         }
218     }
219 
220     RedBlackTree::RedBlackTree() {
221         TNULL = new Node;
222         TNULL->color = Black;
223         TNULL->left = nullptr;
224         TNULL->right = nullptr;
225         root = TNULL;
226     }
227 
228     void RedBlackTree::leftRotate(NodePtr x) {
229         NodePtr y = x->right;
230         x->right = y->left;
231         if (y->left != TNULL) {
232             y->left->parent = x;
233         }
234         y->parent = x->parent;
```

```
235         if (x->parent == nullptr) {
236             this->root = y;
237         }
238         else if (x == x->parent->left) {
239             x->parent->left = y;
240         }
241         else {
242             x->parent->right = y;
243         }
244         y->left = x;
245         x->parent = y;
246     }
247 
248     void RedBlackTree::rightRotate(NodePtr x) {
249         NodePtr y = x->left;
250         x->left = y->right;
251         if (y->right != TNULL) {
252             y->right->parent = x;
253         }
254         y->parent = x->parent;
255         if (x->parent == nullptr) {
256             this->root = y;
257         }
258         else if (x == x->parent->right) {
259             x->parent->right = y;
260         }
261         else {
262             x->parent->left = y;
263         }
264         y->right = x;
265         x->parent = y;
266     }
267 
268     void RedBlackTree::insert(int key) {
269         NodePtr node = new Node;
270         node->parent = nullptr;
271         node->data = key;
272         node->left = TNULL;
273         node->right = TNULL;
274         node->color = Red;
275 
```

```
276         NodePtr y = nullptr;
277         NodePtr x = this->root;
278 
279         while (x != TNULL) {
280             y = x;
281             if (node->data == x->data)
282             {
283                 return;
284             }
285             else if (node->data < x->data) {
286                 x = x->left;
287             }
288             else {
289                 x = x->right;
290             }
291         }
292 
293         node->parent = y;
294 
295         if (y == nullptr) {
296             root = node;
297         }
298         else if (node->data < y->data) {
299             y->left = node;
300         }
301         else {
302             y->right = node;
303         }
304 
305         if (node->parent == nullptr) {
306             node->color = Black;
307             return;
308         }
309 
310         if (node->parent->parent == nullptr) {
    return; }
311 
312         insertFix(node);
313     }
314 
315     void RedBlackTree::printTree() {
```

```
316         if (root) {
317             printHelper(this->root, "", true);
318         }
319     }
320 
321 
322     NodePtr RedBlackTree::minimum(NodePtr node) {
323         while (node->left != TNULL) {
324             node = node->left;
325         }
326         return node;
327     }
328 
329 int main() {
330     uint64_t startTime, endTime;
331     int degree = 2;
332     //srand(time(NULL));
333     int key;
334     for(uint64_t j = 100; j <= 100000000; j*=10)
335     {
336         RedBlackTree tree;
337         startTime = clock();
338         for(int i = 0; i < 100000000; i++){
339             key = rand();
340             tree.insert(key);
341         }
342 
343         endTime = clock();
344         cout <<"10^"<< degree << ": "<< (double)(
    endTime - startTime)/CLK_TCK << "seconds.\n";
345         degree ++;
346         tree.deleteAll();
347     }
348     return 0;
349 }
```

```
 1 #pragma once
 2
 3 using namespace std;
 4
 5 enum Color{
 6     Black,
 7     Red
 8 };
 9
10 struct Node {
11     int data;
12     Node* parent;
13     Node* left;
14     Node* right;
15     Color color;
16 };
17
18 typedef Node* NodePtr;
19
20 class RedBlackTree {
21 private:
22     NodePtr root;
23     NodePtr TNULL;
24
25     void initializeNULLNode(NodePtr node, NodePtr
   parent);
26
27     NodePtr searchTreeHelper(NodePtr node, int key);
28
29     void deleteFix(NodePtr x);
30
31     void rbTransplant(NodePtr u, NodePtr v);
32
33     void deleteNodeHelper(NodePtr node, int key);
34
35     void insertFix(NodePtr k);
36
37     void printHelper(NodePtr root, string indent,
   bool last) ;
38
39     void leftRotate(NodePtr x);
```

```
40
41      void rightRotate(NodePtr x);
42
43  public:
44      RedBlackTree();
45
46      NodePtr getTNULL() { return TNULL; }
47
48      NodePtr minimum(NodePtr node);
49
50      void deleteAll(){ root = TNULL; }
51
52      NodePtr searchTree(int k) { return
    searchTreeHelper(this->root, k); }
53
54      void insert(int key);
55
56      NodePtr getRoot() { return this->root; }
57
58      void deleteNode(int data) { deleteNodeHelper(this
    ->root, data); }
59
60      void printTree();
61  };
62
```

```
cmake_minimum_required(VERSION 3.20)
project(redblackTree)

set(CMAKE_CXX_STANDARD 14)

add_executable(redblackTree main.cpp rbtree.h)

```