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
Карасева Вадима Дмитриевича

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

Саратов 2025

СОДЕРЖАНИЕ

1 Пример программы

```
#include <iostream>
using namespace std;

struct tree // Структура дерева
{
        int inf;
        tree* left;
        tree* right;
        tree* parent;
        bool color; // True - черный, False - красный
};


// Повороты
void left_rotate(tree*& tr, tree* x) // Левый
{
        tree* y = x->right; // Y - правый ребенок X
        x->right = y->left; // Левый ребенок Y становится правым ребенком X
        if (y->left) // Если левый ребенок Y есть
                y->left->parent = x; // Родителем левого ребенка становится X
        y->parent = x->parent; // Родитель для Y становится родитель X
        if (x->parent)
        {
                if (x == x->parent->left) // Если X левый ребенок
                        x->parent->left = y; // Y становится левым ребенком
                         ↪ для родителя X
                else
                        x->parent->right = y;// Становится правым ребенком
                         ↪ соответственно
        }
        y->left = x; // X становится левым ребенком Y
        x->parent = y; // Y становится родитем X
        if (!y->parent) // Если у Y нет родителя
        {
                tr = y;
                y->color = true; // Y становится корнем и окрашивается в
                 ↪ черный цвет
        }
}

```

```
38 void right_rotate(tree*& tr, tree* x) // правый поворот
39 {
40         tree* y = x->left;
41         x->left = y->right;
42         if (y->right)
43                 y->right->parent = x;
44         y->parent = x->parent;
45         if (x->parent)
46         {
47                 if (x == x->parent->left)
48                         x->parent->left = y;
49                 else
50                         x->parent->right = y;
51         }
52         y->right = x;
53         x->parent = y;
54         if (!y->parent)
55         {
56                 tr = y;
57                 y->color = true;
58         }
59 }
60
61 // создание узлов
62 tree* node(tree* p, int x) // красный узел
63 {
64         tree* n = new tree;
65         n->inf = x;
66         n->parent = p;
67         n->right = NULL;
68         n->left = NULL;
69         n->color = false;
70         return n;
71 }
72
73 tree* root(int x) // черный узел
74 {
75         tree* n = new tree;
76         n->inf = x;
77         n->parent = NULL;
78         n->right = NULL;
```

```
79          n->left = NULL;
80          n->color = true;
81          return n;
82  }
83
84  tree* grandparent(tree* x) // дедушка
85  {
86          if (x && x->parent)
87                  return x->parent->parent;
88          else
89                  return NULL;
90  }
91
92  tree* uncle(tree* x) // дядя
93  {
94          tree* g = grandparent(x);
95          if (!g)
96                  return NULL;
97
98          if (x->parent == g->left)
99                  return g->right;
100         else
101                 return g->left;
102 }
103
104
105 tree* brother(tree* x) // брат
106 {
107         if (x && x->parent)
108         {
109                 if (x->parent->left == x)
110                         return x->parent->right;
111                 else
112                         return x->parent->left;
113         }
114         else
115                 return NULL;
116 }
117
118 void preorder_color(tree* tr) // прямой обход с цветами (К-Л-П)
119 {
```

```
120         if (tr)
121         {
122                 cout << tr->inf; // корень
123                 if (tr->color == true)
124                         cout << " - b  ";
125                 else
126                         cout << " - r  ";
127                 preorder_color(tr->left); // левый
128                 preorder_color(tr->right); // правый
129         }
130 }
131
132 // вставка
133 void insert_case5(tree*& tr, tree* x) // Родитель красный, дядя, дед, брат и
    ↪  дети черные
134 {
135         tree* G = grandparent(x); // Дед
136         x->parent->color = true; // Родитель становится черным
137         G->color = false; // Дед становится красным
138         if ((x == x->parent->left) && (x->parent = G->left)) // Если X - левый
          ↪  ребенок и родитель - левый ребенок, то правый поворот, иначе левый
139                 right_rotate(tr, G);
140         else
141                 left_rotate(tr, G);
142 }
143
144 void insert_case4(tree*& tr, tree* x) // родитель красный, дядя черный
145 {
146         tree* G = grandparent(x); // Дед
147         if ((x == x->parent->right) && (x->parent == G->left)) // Если X -
          ↪  правый ребенок и родитель - левый ребенок
148         {
149                 left_rotate(tr, x->parent); // Левый поворот
150                 x = x->left; // Переходим к левому ребенку
151         }
152         else
153         {
154                 if ((x == x->parent->left) && (x->parent == G->right)) //
                  ↪  Иначе правый поворот
155                 {
156                         right_rotate(tr, x->parent);
```

```
157                         x = x->right;
158                 }
159         }
160         insert_case5(tr, x);
161 }
162
163 void insert_case3(tree*& tr, tree* x) // если родитель, Дядя и X - красные
164 {
165         tree* U = uncle(x); // дядя
166         tree* G = grandparent(x); // дедушка
167         if (U && U->color == false && x->parent->color == false)  // если дядя
        ↪  и родитель красные
168         {
169                 x->parent->color = true; // родитель становится черным
170                 U->color = true; // дядя становится черным
171                 G->color = false; // дед - красным
172
173                 if (!G->parent) // если у деда нет родителя - становится
                 ↪  черным
174                         G->color = true;
175                 else
176                 {
177                         if (grandparent(G)) // если у деда есть дед
178                                 insert_case3(tr, G);
179                         else
180                                 return;
181                 }
182         }
183         else
184                 insert_case4(tr, x);
185 }
186
187 void insert_case2(tree*& tr, tree* x) // если родитель x - красный, то
    ↪  переходим к 3 случаю
188 {
189         if (x->parent->color == false)
190                 insert_case3(tr, x);
191         else
192                 return;
193 }
194
```

```
195 void insert_case1(tree*& tr, tree* x) // если узел - корень
196 {
197         if (!x->parent)
198                 x->color = true; // становится черным
199         else
200                 insert_case2(tr, x);
201 }
202
203 void insert(tree*& tr, tree* prev, int x) // Prev - предыдущий узел
204 {
205         if ((x < prev->inf) && (!prev->left)) // Если X меньше предыдущего и у
        ↪  него нет левого ребенка
206         {
207                 prev->left = node(prev, x); // Становится левым ребенком prev
208                 insert_case1(tr, prev->left);
209         }
210         else
211         {
212                 if ((x > prev->inf) && (!prev->right)) // Если X больше
                ↪  предыдущего и у него нет правого ребенка
213                 {
214                         prev->right = node(prev, x); // Становится правым
                        ↪  ребенком prev
215                         insert_case1(tr, prev->right);
216                 }
217                 else
218                 {
219                         if ((x < prev->inf) && (prev->left)) // Если
                        ↪  меньше(больше) и есть левый(правый) ребенок, идем
                        ↪  дальше по дереву
220                             insert(tr, prev->left, x);
221                         else
222                         {
223                                 if ((x > prev->inf) && (prev->right))
224                                 {
225                                         insert(tr, prev->right, x);
226                                 }
227                         }
228                 }
229         }
230 }
```

```
231
232 // удаление
233 void del_element6(tree*& tr, tree* x) // брат, близкий племянник - черные,
    ↪ дальний племянник - красный, а его дети - черные
234 {
235         tree* B = brother(x);
236
237         B->color = x->parent->color; // брат становится того же цвета, что и
            ↪ родитель
238         x->parent->color = true; // родитель становится черным
239
240         if (x == x->parent->left)  // если X - левый ребенок
241         {
242                 B->right->color = true; // правый племянник становится черным
243                 left_rotate(tr, x->parent); // левый поворот
244         }
245         else
246         {
247                 B->left->color = true; // левый племянник становится черным
248                 right_rotate(tr, x->parent); // правый поворот
249         }
250 }
251
252 void del_element5(tree*& tr, tree* x) // брат, дальний племянник - черные,
    ↪ близкий племянник - красный, а его дети - черные
253 {
254         tree* B = brother(x); // брат
255
256         bool col_l; // цвет левого племянника
257         bool col_r; // цвет правого племянника
258
259         if (!B->left)
260                 col_l = true;
261         else
262                 col_l = B->left->color;
263
264         if (!B->right)
265                 col_r = true;
266         else
267                 col_r = B->right->color;
268
```

```
269         if (x == x->parent->left && col_l == false && col_r == true) // если X
            ↪  - левый ребенок, левый племянник - красный, а правый - черный
270         {
271                 B->color = false; // брат становится красным
272                 B->left->color = true; // левый племянник становится черным
273                 right_rotate(tr, B); // правый поворот
274         }
275         else
276         {
277                 if (x == x->parent->right && col_l == true && col_r == false)
                    ↪  // Если X - правый ребенок, левый племянник - черный, а
                    ↪  правый - красный
278                 {
279                         B->color = false; // брат становится красным
280                         B->right->color = true; // правый племянник становится
                            ↪  черным
281                         left_rotate(tr, B); // левый поворот
282                 }
283         }
284         del_element6(tr, x);
285 }
286
287 void del_element4(tree*& tr, tree* x) // X, брат, племянники - черные,
    ↪  родитель красный
288 {
289         tree* B = brother(x); // брат
290
291         bool col_l; // цвет л племянника
292         bool col_r; // цвет п племянника
293
294         if (!B->left)
295                 col_l = true;
296         else
297                 col_l = B->left->color;
298
299         if (!B->right)
300                 col_r = true;
301         else
302                 col_r = B->right->color;
303
```

```
304         if (x->parent->color == false && B->color == true && col_l == true &&
            ↪  col_r == true) // Если родитель - красный, а брат и племянники -
            ↪  черные
305         {
306                 B->color = false; // брат становится красным
307                 x->parent->color = true; // родитель становится черным
308         }
309         else
310                 del_element5(tr, x);
311 }
312
313 void del_element3(tree*& tr, tree* x) // X, родитель, брат, племянники -
    ↪  черные
314 {
315         tree* B = brother(x); // брат
316
317         bool col_l; // цвет л племянника
318         bool col_r; // цвет п племянника
319
320         if (!B->left)
321                 col_l = true;
322         else
323                 col_l = B->left->color;
324
325         if (!B->right)
326                 col_r = true;
327         else
328                 col_r = B->right->color;
329
330         if (x->parent->color == true && B->color == true && col_l == true &&
            ↪  col_r == true) // Если родитель, брат и племянники - черные
331         {
332                 B->color = false; // брат становится красным
333
334                 if (!x->parent) // если X - корень
335                 {
336                         if (x->left)
337                                 x->left->color = true; // если есть левый
                                    ↪  ребенок - он черный
338                         else
339                         {
```

```
340                                 tr = x;
341                                 x->right->color = true;
342                         }
343                 }
344                 else
345                 {
346                         tree* B = brother(x);
347                         if (B->color == false)
348                         {
349                                 x->parent->color = false;
350                                 B->color = true;
351                                 if (x == x->parent->left)
352                                         left_rotate(tr, x->parent);
353                                 else
354                                         right_rotate(tr, x->parent);
355                         }
356                         del_element3(tr, x);
357                 }
358         }
359         else
360                 del_element4(tr, x);
361 }
362
363 void del_element2(tree*& tr, tree* x) // x - черный, родитель черный, брат
    ↪ красный
364 {
365         tree* B = brother(x);
366         if (B->color == false)
367         {
368                 x->parent->color = false;
369                 B->color = true;
370                 if (x == x->parent->left)
371                         left_rotate(tr, x->parent);
372                 else
373                         right_rotate(tr, x->parent);
374         }
375         del_element3(tr, x);
376 }
377
378 void del_element1(tree*& tr, tree* x) // x - корень дерева, одна ветка
379 {
```

```
380         if (!x->parent)
381         {
382                 if (x->left)
383                         x->left->color = true;
384                 else {
385                         tr = x;
386                         x->right->color = true;
387                 }
388         }
389         else {
390                 del_element2(tr, x);
391         }
392 }
393
394 void replace(tree*& tr, tree* x) // случай когда удаляем x с одним ребенком
395 {
396         if (x->left)
397         {
398                 if (x->parent)
399                 {
400                         tree* ch = x->left;
401                         ch->parent = x->parent;
402                         if (x == x->parent->left)
403                                 x->parent->left = ch;
404                         else
405                                 x->parent->right = ch;
406                 }
407         }
408         else
409         {
410                 if (x->parent)
411                 {
412                         tree* ch = x->right;
413                         ch->parent = x->parent;
414                         if (x == x->parent->left)
415                                 x->parent->left = ch;
416                         else
417                                 x->parent->right = ch;
418                 }
419         }
420 }
```

```
421
422 tree* Max_par(tree* tr)
423 {
424         if (!tr->right)
425                 return tr; // нет правого ребенка
426         else
427                 return Max_par(tr->right); // идем по правой ветке до конца
428 }
429
430 void delete_one(tree*& tr, tree* x) // удаление
431 {
432         tree* buf = NULL;
433
434         if (x->right && x->left) // есть два ребенка
435         {
436
437                 buf = Max_par(x->left); // предыдущий по значению элемент
438
439                 int tpm = x->inf;
440                 x->inf = buf->inf; // меняем местами с предыдущим по значению
                    ↪ элементом
441                 buf->inf = tpm;
442
443                 x = buf;
444         }
445         if (x->right || x->left) // есть один ребенок
446         {
447                 tree* ch = NULL;
448                 if (x->left && !x->right)
449                         ch = x->left;
450                 if (x->right && !x->left)
451                         ch = x->right;
452
453                 replace(tr, x);
454                 if (x->color == true) // если удаляемый элемент был красным то
                    ↪ тот, что на его месте перекрашиваем в красный (если нужно)
455                 {
456                         if (ch->color == false)
457                                 ch->color = true;
458                         else
459                                 del_element1(tr, x);
```

```
460                 }
461         }
462         else // нет детей
463         {
464                 if (!x->right && !x->left)
465                 {
466                         if (x->color == true) {
467                                 del_element1(tr, x);
468                         }
469                         else
470                         {
471                                 if (x == x->parent->left)
472                                         x->parent->left = NULL;
473                                 else
474                                         x->parent->right = NULL;
475                         }
476                 }
477         }
478
479         if (!x->left && !x->right)
480         {
481                 if (x == x->parent->left)
482                         x->parent->left = NULL;
483                 else
484                 {
485                         if (x == x->parent->right)
486                                 x->parent->right = NULL;
487                 }
488         }
489
490         delete x;
491 }
492
493 tree* find(tree* tr, int x) // поиск
494 {
495         if (!tr || x == tr->inf) // нашли или дошли до конца ветки
496                 return tr;
497         if (x < tr->inf)
498                 return find(tr->left, x); // ищем по левой ветке
499         else
500                 return find(tr->right, x); // ищем по правой ветке
```

```
501 }
502
503 int main() {
504         setlocale(LC_ALL, "Russian");
505
506         tree* tr = NULL;
507
508         cout << "Введите количество элементов: \n";
509         int n; cin >> n;
510
511         cout << "Введите элементы: \n";
512         int x; cin >> x;
513         tr = root(x);
514         for (int i = 0; i < n - 1; ++i) {
515                 cin >> x;
516                 insert(tr, tr, x);
517         }
518
519         cout << "Прямой обход: ";
520         preorder_color(tr);
521         cout << endl;
522
523         for (int i = 0; i < 1; i++) {
524                 cout << "Введите элемент для удаления: ";
525                 cin >> x;
526                 delete_one(tr, find(tr, x));
527                 cout << endl;
528
529                 cout << "Прямой обход: ";
530                 preorder_color(tr);
531                 cout << endl;
532         }
533
534
535         for (int i = 0; i < 1; i++) {
536                 cout << "Введите элемент для удаления: ";
537                 cin >> x;
538                 delete_one(tr, find(tr, x));
539                 cout << endl;
540
541                 cout << "Прямой обход: ";
```

```
542                 preorder_color(tr);
543                 cout << endl;
544         }
545
546
547
548         // Ввод нового элемента
549         cout << "Введите элемент для вставки: ";
550         cin >> x;
551         insert(tr, tr, x); // Вставляем введенный элемент
552
553         // Вывод дерева после вставки
554         cout << "Прямой обход после вставки нового элемента: ";
555         preorder_color(tr);
556         cout << endl;
557
558         return 0;
559 }
```

2 Поиск в красно-черном дереве

Пусть у красно-чёрного дерева будет высота h. Так как у красной вершины чёрные дети (по свойству 3), количество красных вершин не больше $h/2$. Тогда чёрных вершин не меньше, чем $h/2-1$. Для количества внутренних вершин в дереве выполняется неравенство $N \leq h/2 - 1$. Прологарифмировав неравенство, имеем: $\log(N+1) \geq h/2 \rightarrow 2\log(N+1) \geq h \rightarrow h \leq 2\log(N+1)$.

Во всех случаях операция поиска займёт не более $O(logN)$ времени, так как красно-чёрное дерево является сбалансированным.

3 Вставка в красно-черном дереве

3.1 Лучший случай

Никакие свойства не были нарушены либо произошло просто перекрашивание, занимающее константное время. Сложность — $O(logN)$.

3.2 Худший случай

Красно-чёрным деревьям требуется не более 2 поворотов для восстановления баланса, каждый из которых занимает константное время. Так что в худшем случае при вставке будет 2 оборота, и временная сложность составит $O(logN)$.

3.3 Средний случай

Средний случай является средним значением всех возможных случаев, следовательно, временная сложность вставки в этом случае также составит $O(logN)$. Таким образом, временная сложность для всех случаев равна $O(logN)$.

4 Удаление в красно-черном дереве

4.1 Лучший случай

В лучшем случае вращений нет, и может происходить разве что перекрашивание за константное время, потому что мы просто обращаемся к областям памяти. Временная сложность составит $O(logN)$.

4.2 Худший случай

Красно-чёрным деревьям требуется не более 3 поворотов во время удаления. Так что в худшем случае при удалении будет 3 поворота и временная сложность составит $O(logN)$.

4.3 Средний случай

Средний случай является средним значением всех возможных случаев, следовательно, временная сложность вставки в этом случае также составит $O(logN)$. Таким образом, временная сложность для всех случаев равна $O(logN)$.

5 Обходы дерева

Красно-чёрное дерево, также, как и дерево бинарного поиска имеет 3 основных обхода: прямой, обратный и симметричный. Их разница заключается в том, в каком порядке мы обращаемся к элементам. Каждый из них будет иметь временную сложность $O(N)$, так как процедура вызывается ровно два раза для каждого узла дерева.

6 Расход памяти

Расход памяти в красно-чёрном дереве происходит так же, как в двоичном дереве поиска, и определяется общим количеством узлов. Поэтому получаем O(N), потому что нам не нужно дополнительное пространство для хранения повторяющихся структур данных. Данный вывод следует из того, что каждый узел имеет три указателя: левый ребёнок, правый ребёнок и родитель. Каждый узел занимает O(1) памяти. Для отслеживания цвета каждого узла требуется только один бит информации на каждый узел. Во многих случаях дополнительный бит данных может храниться без дополнительных затрат памяти. В результате сложность по памяти будет равна O(N), где N – количество узлов в дереве.

Вращение и перекрашивание происходят за время $O(1)$. Временная сложность всех функций равна $O(logN)$, потому что дерево всегда сбалансированное.