Санкт-Петербургский политехнический университет Петра Великого

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Тема: RedBlackTree

Студент гр. 3331506/00401 Лещёва А. В.

Преподаватель Ананьевский М. С.

Санкт-Петербург

2023

Оглавление

Введение

При разработке программного обеспечения применяются различные структуры хранения данных. От способа хранения информации и алгоритмов её поиска зависит производительность разрабатываемого программного продукта.

В 1978 году Гюиба (Guibas) и Седжвик (Sedgewick) изобрели концепцию красно-черного дерева, удовлетворяющего такому умеренно нестрогому требованию. Красно-черные деревья - это структуры данных, используемые для реализации карт преобразования данных в библиотеке стандартных шаблонов C++ (C++ Standard Template Library).

Красно-черные деревья были разработаны с целью исправить несбалансированность двоичных деревьев. Для этого двоичному дереву добавляется алгоритм, балансирующий его. Каждый-элемент в красно-черном дереве имеет не только ключ для сравнения элементов, но и свойство – цвет. Цветов всего два – красный и черный. Алгоритм красно-черного дерева при добавлении или удалении элемента проверяет не нарушился ли порядок цветов элементов и проводит изменения связей элементов и цветов при необходимости. Благодаря этому усложнению красно-черное дерево всегда сбалансировано, и время поиска в нем элемента пропорционально:

$$\log_2(n),$$

где n – общее количество элементов в дереве.

Зависимость времени поиска от количества вершин представлена на рисунке 1.

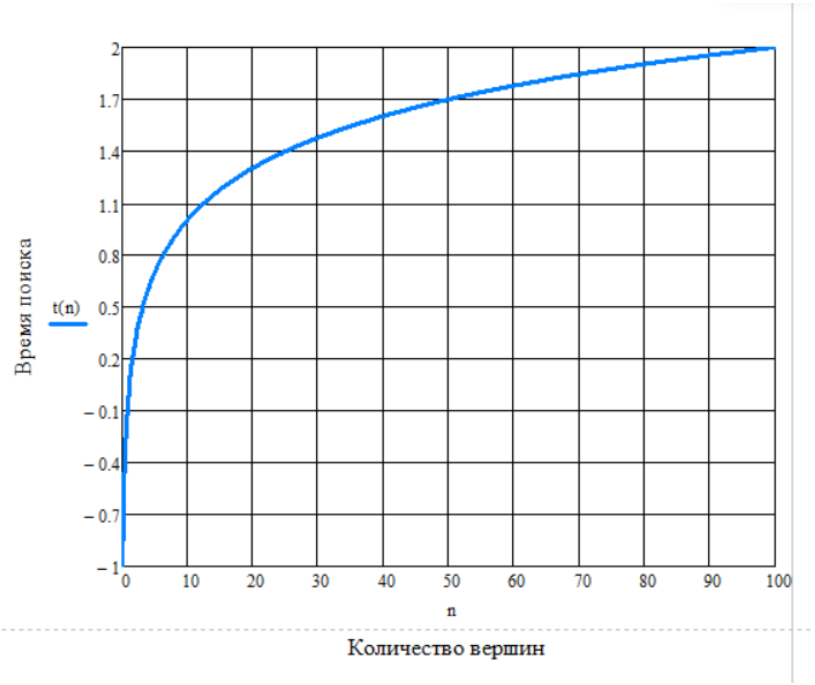

Рисунок 1 – Время поиска.

За это приходится расплачиваться дополнительными затратами памяти на хранение цветов элементов и увеличенным временем работы алгоритма вставки и удаления элемента[1]

Реализация красно-черных деревьев на императивных языках достаточно сложна, так как требует большого количества манипуляций с указателями. Считается, что производительность императивных реализаций обычно выше функциональных, однако функциональные реализации красно-черных деревьев намного более компактны, идеи, используемые для реализации алгоритмов просты и понятны, в результате чего и производительность таких реализаций заметно повышается.

Самое главное преимущество красно-черных деревьев в том, что при вставке выполняется не более O(1) вращений.

Операции над красно-черными деревьями имеют логарифмическую сложность, что достигается балансировкой дерева.[2] Большая часть операции с деревом двоичного поиска займет время[3, стр.117]:

$$O(h),$$

где h-высота дерева двоичного поиска.

Зависимость времени операций от высоты дерева представлена на рисунке 2

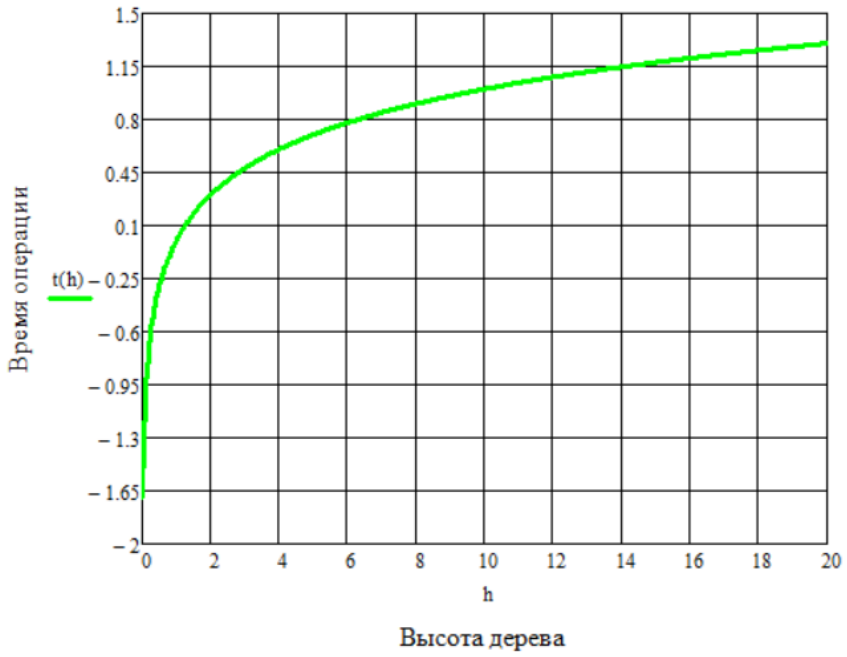

Рисунок 2 – Время операций

В таблице 1 представлена сложность операций RedBlackTree:

Таблица 1 – Сложность операций

Операция	Сложность операции
Вставка	O(log n)
Удаление	O(log n)
Поиск	O(log n)

Такие деревья имеют большое практическое значение, так как их использование сокращает машинное время, требуемое на выполнение различных алгоритмов.[2]

Экспериментально была найдена зависимость времени операции (вставки) от количества вершин. На рисунке 3 представлена временная зависимость от вставки 1, 11, 21, 31, …,601 вершины.

Количество вершин	1	11	21	31	41	51	61	71	81	91	101	201	301	401	501	601
Время операции	0,011	0,028	0,144	0,245	0,525	0,837	1,05	0,634	1,05	0,609	0,804	1,301	7,859	2,77	4,122	3,234

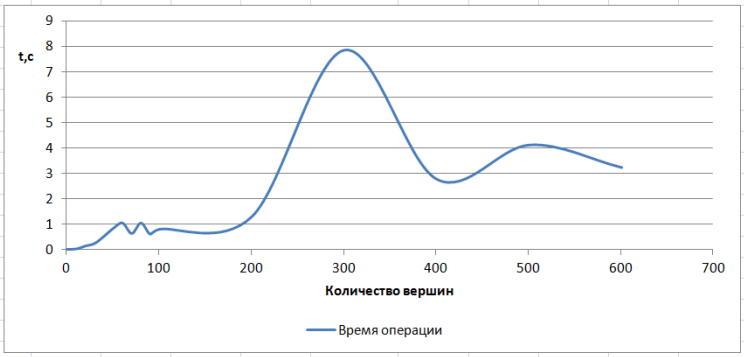

Рисунок 3 – Экспериментальная зависимость.

Экспериментальная зависимость времени поиска от количества вершин представлена на рисунке 4.

Вершины	1	101	201	301	401	501	601
Время поиска	0,002	0,012	0,015	0,004	0,134	0,019	0,12

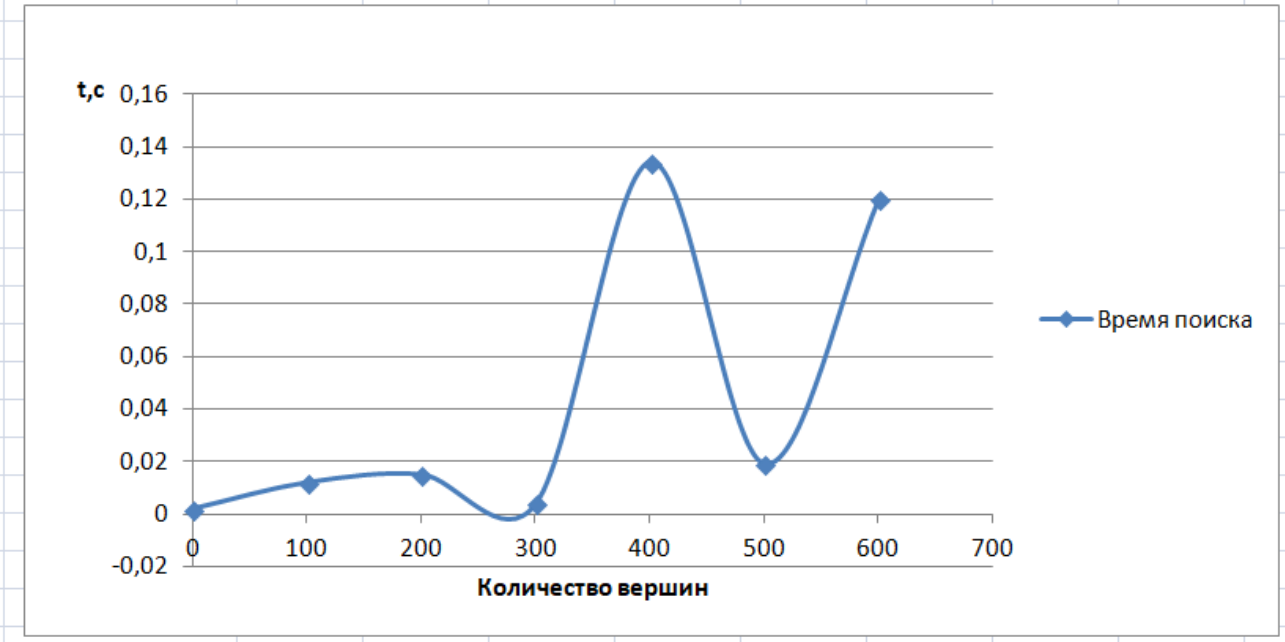

Рисунок 5 – Экспериментальная зависимость времени поиска.

2. RedBlackTree.

Красно-чёрное дерево (англ. *red-black tree*) — двоичное дерево поиска, в котором баланс осуществляется на основе "цвета" узла дерева, который принимает только два значения: "красный" (англ. *red*) и "чёрный" (англ. *black*).

При этом все листья дерева являются фиктивными и не содержат данных, но относятся к дереву и являются чёрными.

Для экономии памяти фиктивные листья можно сделать одним общим фиктивным листом.

Пример Красно-чёрного дерева приведен на рисунке 1.

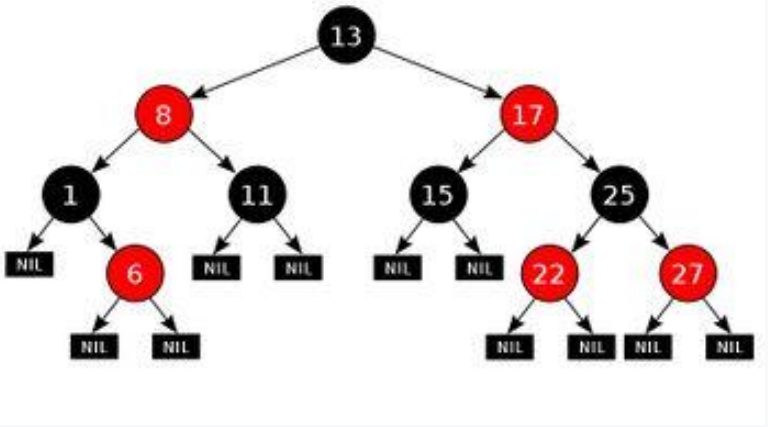

Рисунок 1 - Пример красно-чёрного дерева.

2.1 Свойства RedBlackTree.

Существует пять правил, позволяющих красно-чёрному дереву самобалансироваться[4, стр.134]:

1. Все узлы могут иметь лишь один из двух цветов. Красный, либо чёрный
2. Корень дерева – чёрный.
3. Все листья – чёрные
4. У красного узла потомки могут быть только чёрного цвета.
5. Пути от узла к его листьям должны содержать одинаковое количество черных узлов (это черная высота).

Следствием из этих правил является утверждение, что самый короткий путь от узла до листа не более чем в два раза короче самого длинного пути. Чтобы доказать это утверждение достаточно рассмотреть правила 4 и 5.

2.2 Глубина RedBlackTree.

Пусть количество чёрных узлов в коротком пути равно B и соответствует общему количеству узлов в пути. Самый длинный путь формируется включением красных узлов в самый короткий путь. Правило 4 не позволяет включать красные узлы подряд. Потому самый длинный путь равен 2B. Это свойство и позволяет реализовать эффективный алгоритм поиска по дереву. Отсутствие необходимости балансировать дерево обуславливается более сложным алгоритмом вставки элемента, по сравнению с обычным двоичным деревом поиска.

2.3 Достоинства RedBlackTree.

1. Самое главное преимущество красно-черных деревьев в том, что при вставке выполняется не более O(1) вращений. Это важно, например, в алгоритме построения динамической выпуклой оболочки. Ещё важно, что примерно половина вставок и удалений произойдут задаром.
2. Процедуру балансировки практически всегда можно выполнять параллельно с процедурами поиска, так как алгоритм поиска не зависит от атрибута цвета узлов.
3. Сбалансированность этих деревьев хуже, чем у АВЛ, но работа по поддержанию сбалансированности в красно-чёрных деревьях обычно эффективнее. Для балансировки красно-чёрного дерева производится минимальная работа по сравнению с АВЛ-деревьями.

3. Основные операции

3.1 Правый и левый повороты.

В красно чёрном дереве для балансировки используются правый и левый поворот

Алгоритм реализации правого поворота представлен на рисунке 3, левого на рисунке 4.

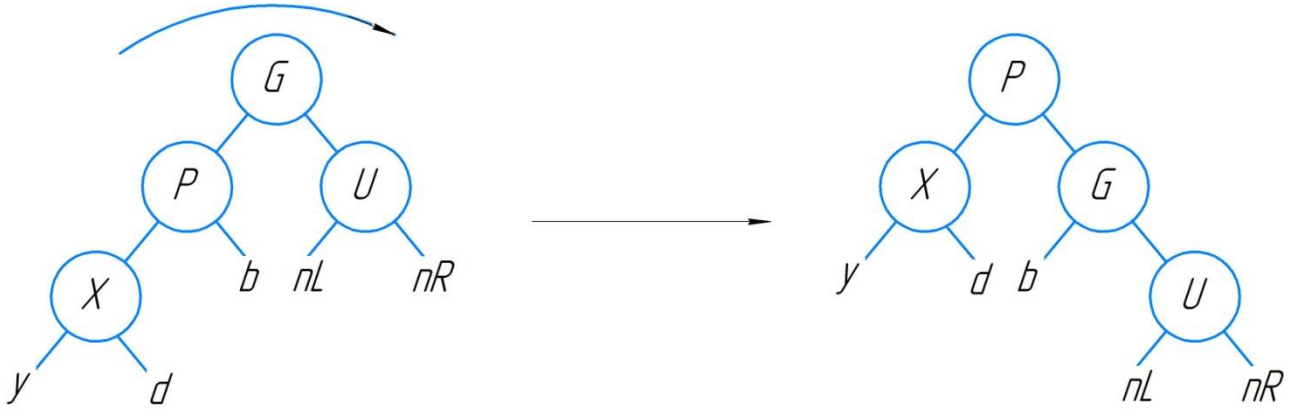

Рисунок 3 – Правый поворот

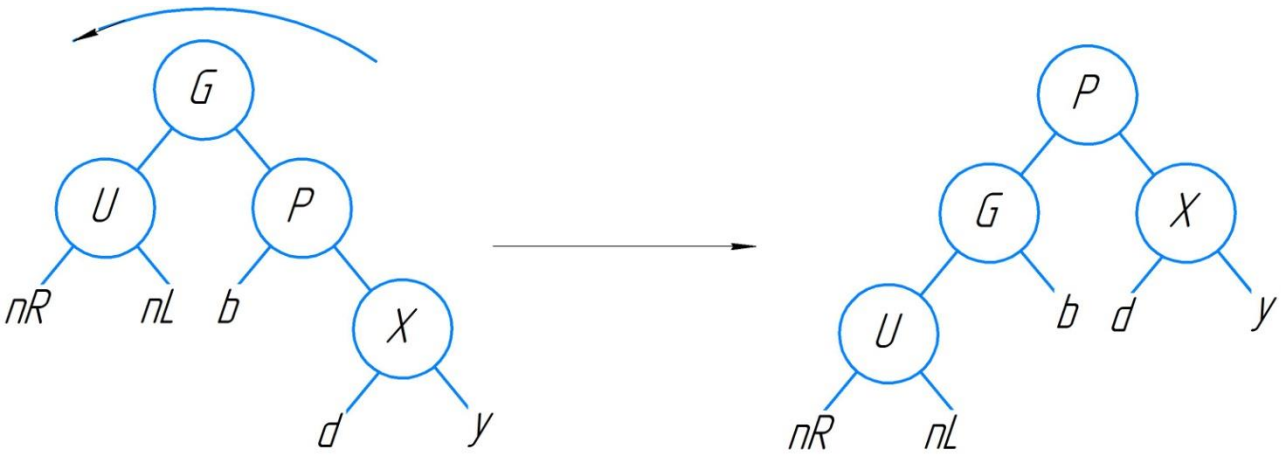

Рисунок 4 –Левый поворот.

3.2 Вставка

Каждый элемент вставляется вместо листа, поэтому для выбора места вставки идём от корня до тех пор, пока указатель на следующего сына не станет null (то есть этот сын — лист). Вставляем вместо него новый элемент с нулевыми потомками и красным цветом. Красный цвет нового узла нужен для того, чтобы не нарушить свойство глубины, но нужно проверить балансировку на случай, если у нас получилось два красный узла подряд.

3.3 Балансировка после вставки

Балансировка начнет работать, если у нас после вставки получилось два красных узла подряд.

Балансировка делится на два случая: дядя нового узла черный или красный.

Введем новые обозначения, которые будут использоваться в дальнейшем

X – новый узел.

P – родитель нового узла.

G – дедушка нового узла (родитель P)

U – дядя нового узла (брат P и ребенок G)

nL и nR – левый и правый племянники соответственно.

1. Дядя красный.

Дядя нового узла тоже красный. Тогда, чтобы сохранить свойства , просто перекрашиваем отца и дядю в чёрный цвет, а дедушку — в красный. В таком случае черная высота в этом поддереве одинакова для всех листьев и у всех красных вершин отцы черные. Проверяем, не нарушена ли балансировка. Для дедушки, ведь у него тоже мог быть красный родитель. Если в результате этих перекрашиваний мы дойдём до корня, то в нём в любом случае ставим чёрный цвет, чтобы дерево удовлетворяло свойству. Случай представлен на рисунке 5.

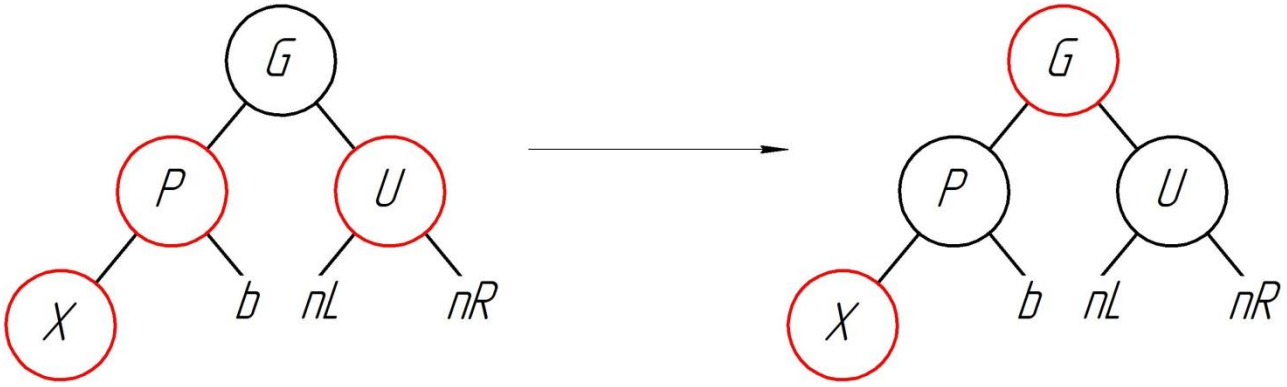

Рисунок 5 – Дядя красный.

2. Дядя чёрный

Дядя чёрный. Если выполнить только перекрашивание, то может нарушиться постоянство чёрной высоты дерева по всем ветвям. Поэтому выполняем поворот. Здесь имеет значения дядя правый или левый потомок дедушки, если дядя правый, то поворот правый, если левый, то левый. Эти случаи представлены на рисунках 6 и 7.

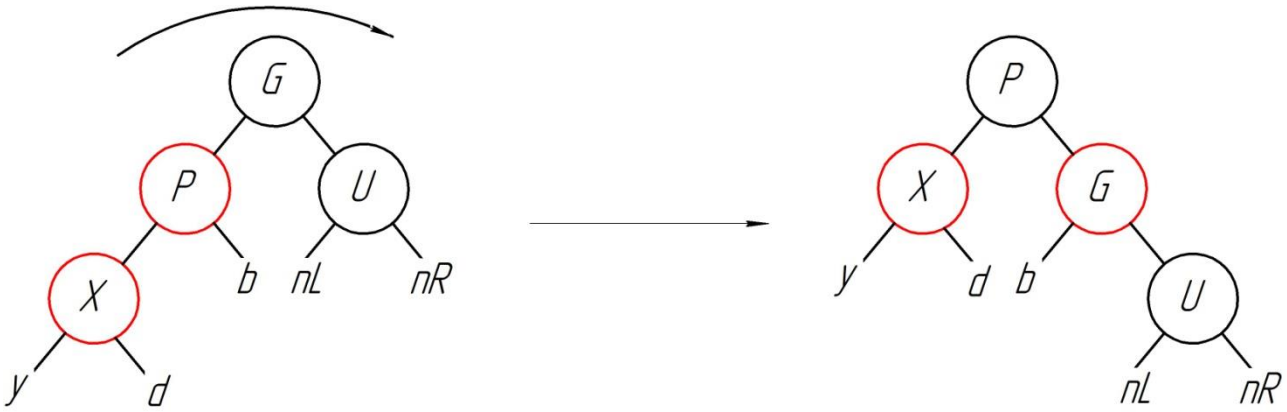

Рисунок 6 – Дядя правый ребенок – правый поворот.

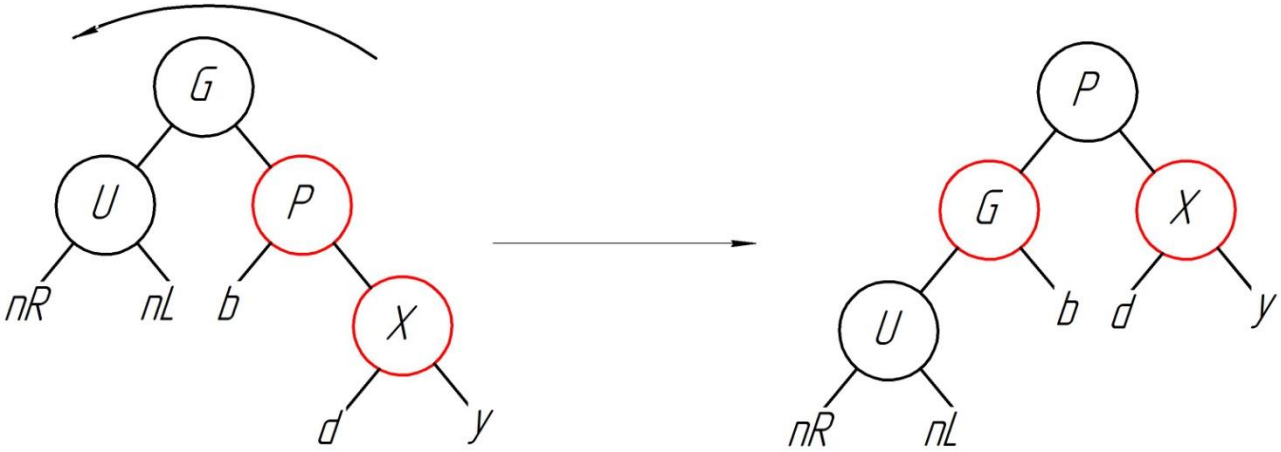

Рисунок 7– Дядя левый ребенок – левый поворот.

Также стоит отдельно рассмотреть случай, который представлен на рисунке 8.

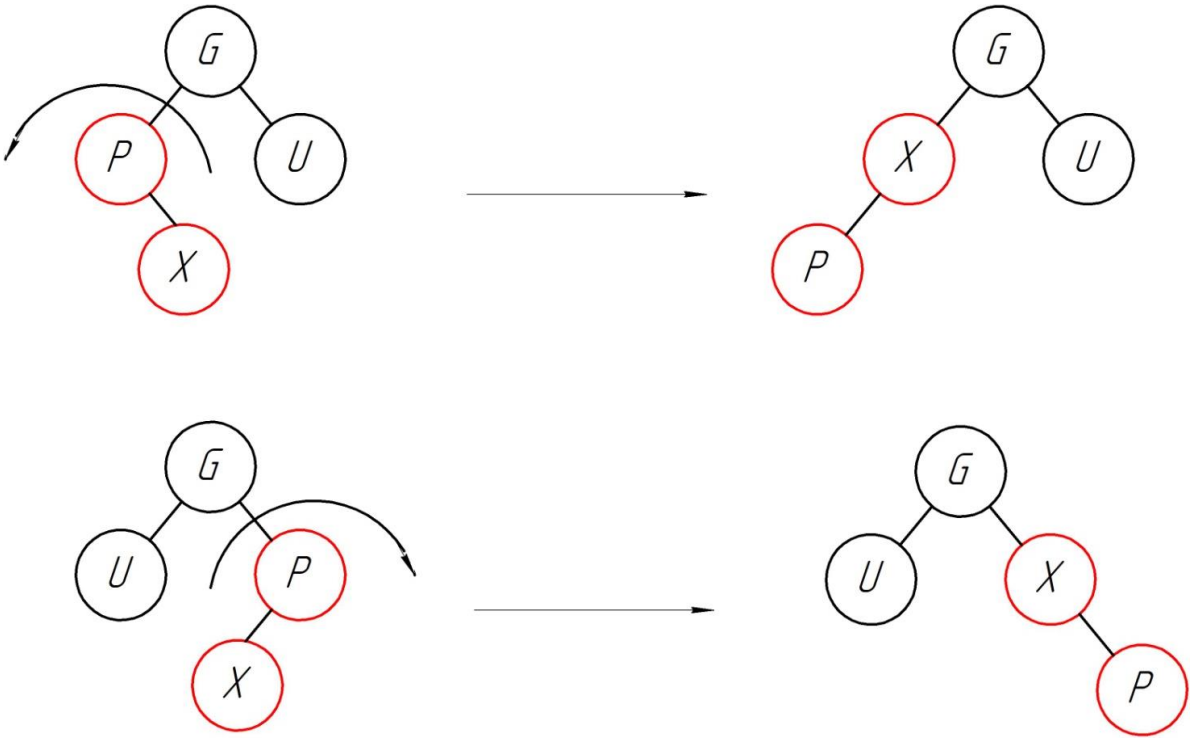

Рисунок 8 - Дополнительный поворот.

Чтобы сделать поворот, как на рисунках 6 и 7, нужно привести их к определенному виду. Если у нас случай, который представлен на рисунке 8, то необходимо сделать дополнительно левый, либо правый поворот.

После поворотов у нас P (который стал дедушкой) черный, значит два подряд красных не будет. Прекращаем балансировку.

3.4 Удаление

При удалении вершины могут возникнуть три случая в зависимости от количества её детей:

- ♣ Если у вершины нет детей, то просто удаляем вершину, деля её null
- ♣ Если у неё только один ребёнок, то делаем у родителя ссылку на него вместо этой вершины.
- ♣ Если же имеются оба ребёнка, то находим минимальную вершину. Для этого, от удаляемого узла нам нужно сделать шаг вправо и до конца влево. Этот случай представлен на рисунке 9.

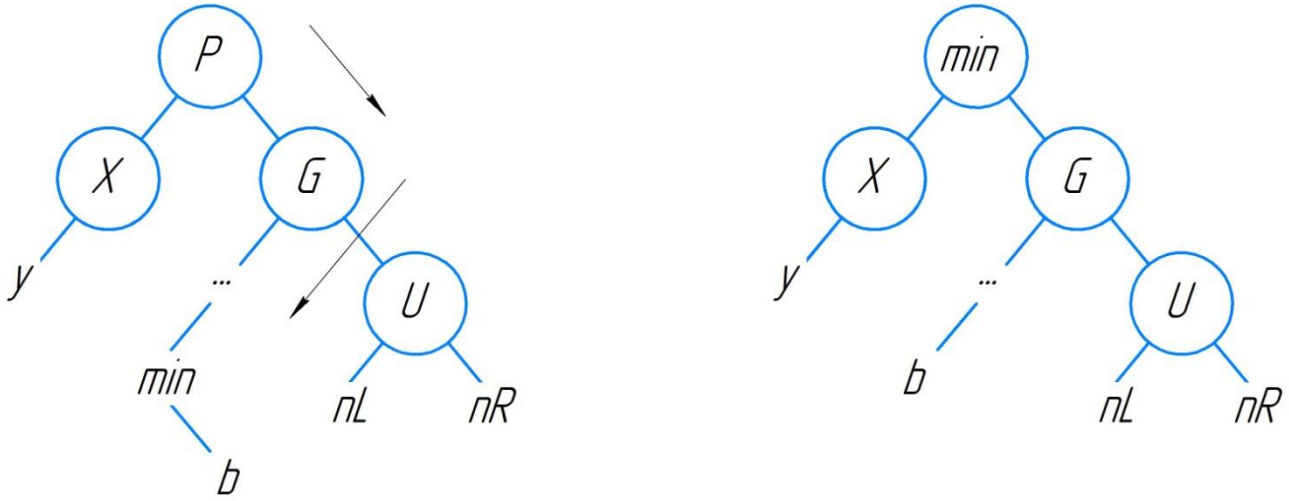

Рисунок 9 – Есть два ребенка.

3.5 Балансировка после удаления

При удалении красной вершины никакие свойства не будут нарушены. При удалении черной вершины нарушится черная глубина. Для такого случая нужна балансировка

Введем обозначения:

X – удаляемый узел (для демонстрации, где был, сейчас это null).

P – родитель X узла.

G – дедушка X узла (родитель P)

U – дядя X узла (брат P и ребенок G)

nL и nR – левый и правый племянники соответственно.

Будут рассмотрены случая, когда дядя – правый ребенок. В противном случае, всё делается аналогично.

1. Дядя красный.

Необходимо сделать дядю черным. Для этого используем перекрашивание, как на рисунке 10, правый поворот и переходим к случаю 2.

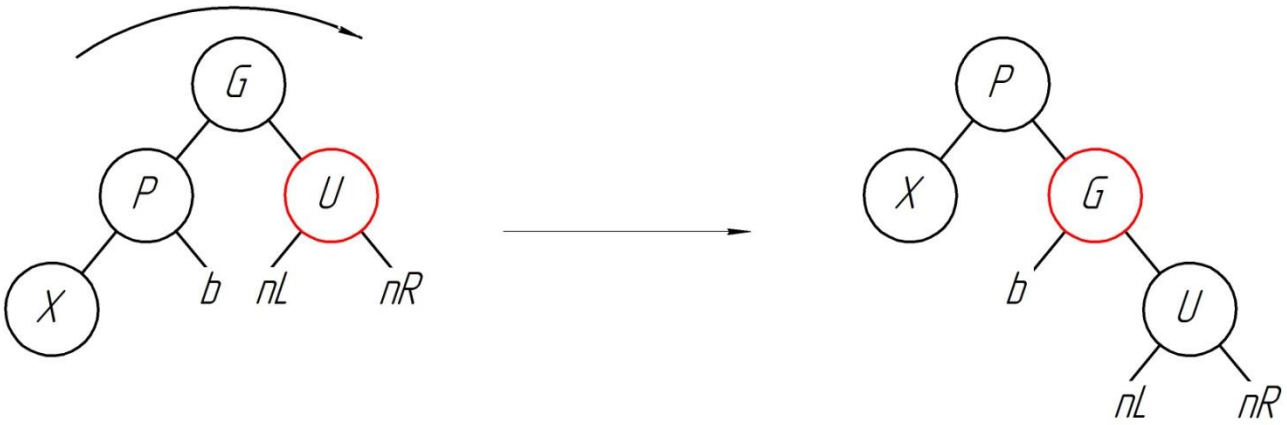

Рисунок 10 – Дядя красный

2. Дядя чёрный

Либо сразу, либо после случая 1.

Добавим обозначение:

S – брат

T и U – дети брата.

Этот случай делится на три случая

2.1 Оба ребенка брата чёрные.

Делаем перекрашивание, как на рисунке 11 и рекурсивно смотрим отца.

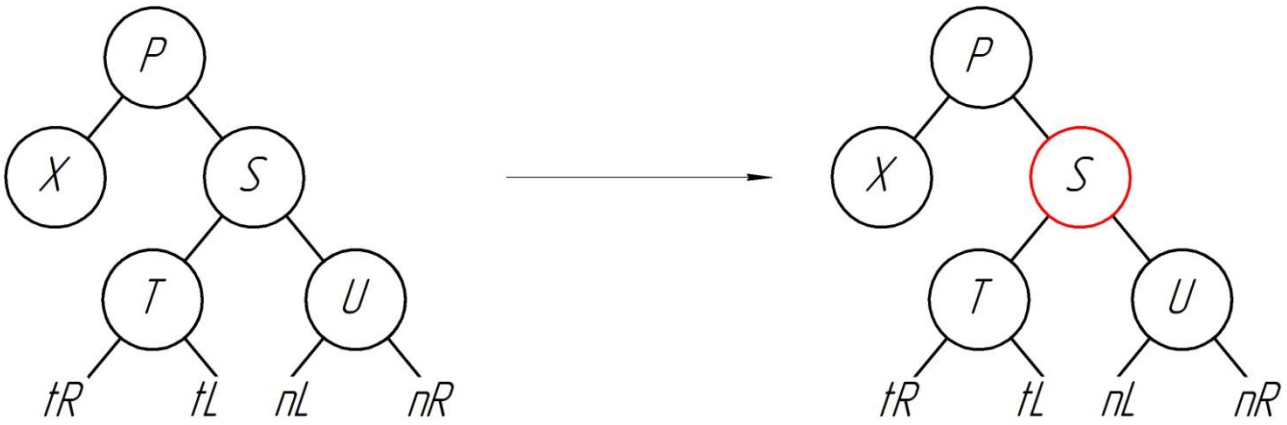

Рисунок 11 – Дети брата чёрные

2.2 У брата: правый ребенок – чёрный, левый – красный.

Перекрашиваем, как на рисунке 12 и делаем правый поворот относительно оси TS

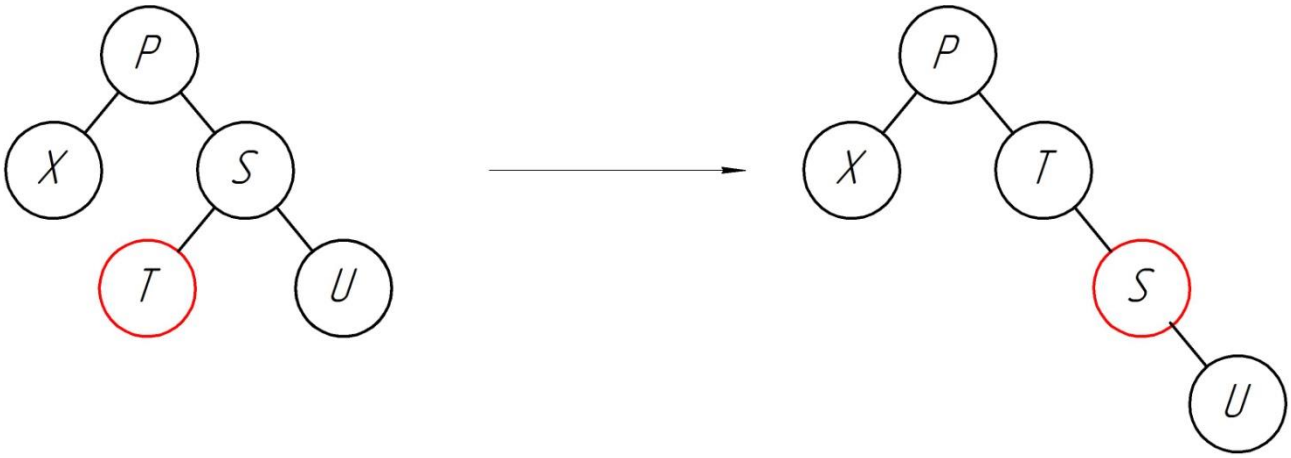

Рисунок 12 – Случай 2.2

Теперь у нас получается, что у брата точно правый ребенок – красный и мы переходим к случаю 2.3.

2.3 У брата правый ребенок – красный, левый не важно.

Делаем перекрашивание, как на рисунке 13 и левый поворот относительно PS.

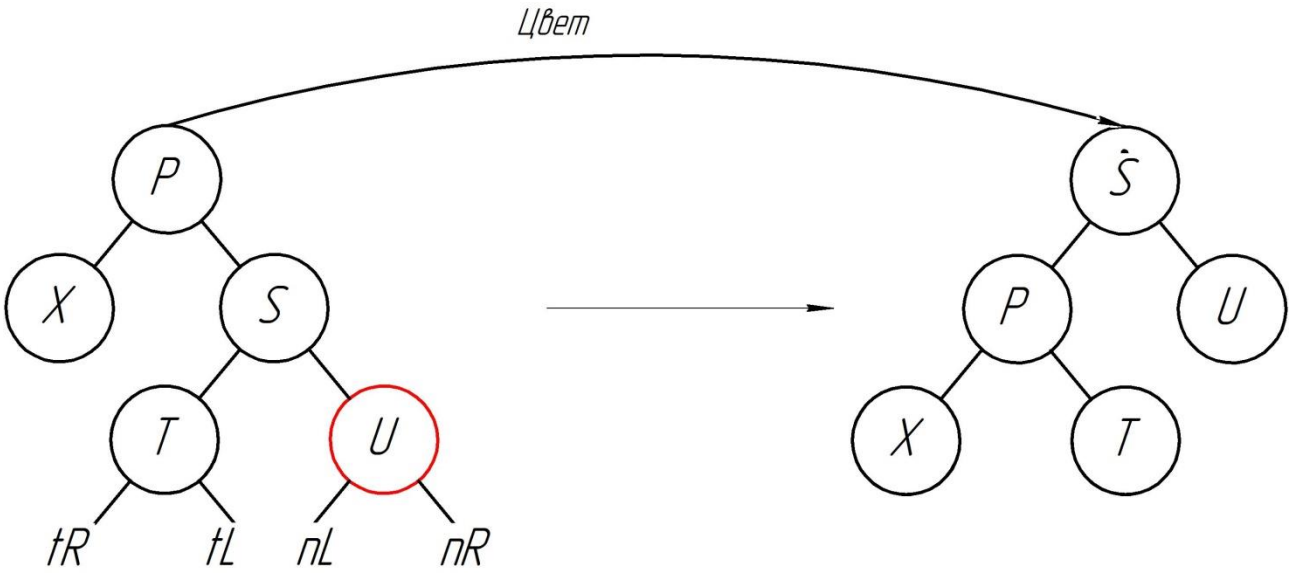

Рисунок 13 – Случай 2.3

Балансировка закончена.

3.6 Поиск

Поиск производится по средствам создания новой переменной и прохождения всего дерева, сравнивая данные каждой встречаемой на пути вершины и выходом при нахождении.

4. Заключение.

В настоящее время для решения многих типов задач, например, таких как классификация, оптимизация и прогнозирование, используются деревья принятия решения. Данный метод заключается в построении структуры дерева, узлами которого являются условные операции и результаты задачи, а ветвями – вероятности перехода к узлу или булевы переменные [5].

Ассоциативные массивы (словари) в большинстве библиотек реализованы именно с помощью красно черных деревьев.

Красно-чёрное дерево используется для организации сравнимых данных, таких как фрагменты текста или числа. Листовые узлы красно-чёрных деревьев не содержат данных, благодаря чему не требуют выделения памяти — достаточно записать в узле-предке в качестве указателя на потомка нулевой указатель.

Красно-чёрные деревья используются в драйвере ext4, подсистеме управление памятью при упорядочивании диапазонов выделения виртуальных адресов, планировщике запросов ввода-вывода, драйверах некоторых физических устройств и даже планировщике процессов ядра Linux. [2]

Приложение

1. Код программы RedBlackTree

https://github.com/soomrack/MR2022/tree/main/Anya_Leshchyova/RedBlackTree

```
#include <iostream>
#include <stack>

struct Node {
    int data;
```

```
    int key;
    int color;
    Node *left;
    Node *right;
};

typedef Node *NodePtr;
typedef NodePtr *NodePtrPtr;

class Exception {
    int kod_mistake;
public:
    Exception(int kod);
};


Exception::Exception(int kod) {
    kod_mistake = kod;
}

Exception stack_full = 1;
Exception stack_empty = 2;


class Stack {
    NodePtrPtr *stack;
    int sp;  //stack point
    int capacity; // размер стека
public:
    Stack();
    ~Stack();
    void push(NodePtrPtr);
    void pop();
    NodePtrPtr peek();   //посмотреть самый верхний элемент ( без извлечения )
    inline bool isEmpty();
    inline bool isFull();
    void print();
    void clear();
};

Stack::Stack(){
    capacity = 10;
    stack = new NodePtrPtr[capacity];
    sp = -1;
}

Stack::~Stack() {
    delete[] stack;
}


void Stack::push(NodePtrPtr item) {
    if (isFull()) throw stack_full;
    stack[++sp] = item;
}


void Stack::pop() {
```

```
    if (isEmpty()) throw stack_empty;
    sp--;
}


NodePtrPtr Stack::peek() {
    if (!isEmpty()) {
        return stack[sp];
    }
    else {
        throw stack_empty;
    }
}


inline bool Stack::isEmpty() {
    return sp == -1;
}


inline bool Stack::isFull() {
    return sp == capacity - 1;
}


void Stack::print() {
    for (int number = sp; number >= 0; number--){
        std::cout << (*(stack[number]))->data<< " ";
    }
    std::cout <<  " \n";
}


void Stack::clear(){
    while (isEmpty() != true){
        pop();
    }
}


class RedBlackTree{
public:
    RedBlackTree();
    bool search(int data) ;
    void del(int data);
    void print();
    void insert(int data);

private:
    NodePtr root;
    NodePtr TNULL;
    int hash ();
    int count;  // для hash функции
    Stack stack_parent_ptr;
    NodePtr search(NodePtr node, int data);
    void print(NodePtr root, std::string indent, bool last);
    void del(NodePtr node, int data);
    void insert_fix (NodePtrPtr parent_node_ptr,NodePtrPtr node_ptr  ) ;
```

```
    void right_rotate (NodePtrPtr node_ptr, NodePtrPtr parent_node_ptr,
NodePtrPtr parent_parent_node_ptr);
    void left_rotate (NodePtrPtr node_ptr, NodePtrPtr parent_node_ptr,
NodePtrPtr parent_parent_node_ptr);
    NodePtrPtr min(NodePtrPtr node);  // node =! TNULL
    NodePtrPtr parent_min(NodePtrPtr  node);  // node =! TNULL
    void del_fix(NodePtr node_to_del, int data);
};


RedBlackTree:: RedBlackTree(){
    TNULL = new Node;
    TNULL->left = nullptr;
    TNULL->right = nullptr;
    root = TNULL;
    TNULL->color = 0;
    TNULL->key = 0;
    count = -1;
}


int RedBlackTree:: hash (){
    count ++;
    return count + 1;
}


void    RedBlackTree::insert_fix(NodePtrPtr    parent_node_ptr,NodePtrPtr
node_ptr  ) {
    NodePtr count = (*parent_node_ptr);
    while (count->color == 1) {
        stack_parent_ptr.pop();
        if ((*stack_parent_ptr.peek())->left == *parent_node_ptr) {
            //  MY PARENT IS LEFT
            if ((*stack_parent_ptr.peek())->right->color == 1) {
                //  MY UNCLE IS RED
                //             G(B)                          G(R)
                //            /    \                        /   \
                //          P(R)    U(R)      ->          P(B)  U(B)
                //          |                              |
                //         X(R)                           X(R)
                (*stack_parent_ptr.peek())->color = 1;
                (*stack_parent_ptr.peek())->right->color = 0;
                (*parent_node_ptr)->color = 0;
                if       ((*stack_parent_ptr.peek())       !=       root)
stack_parent_ptr.pop();
            } else {
                //  MY UNCLE IS BLACK
                if ((*parent_node_ptr)->right == *node_ptr) {
                    //             G(B)                           G(B)
                    //            /    \                         /   \
                    //          P(R)    U(B)      ->           X(R)  G(R)
                    //            \                           /
                    //           X(R)                       P(R)
                    left_rotate(node_ptr, node_ptr, parent_node_ptr);
                }
                //            G(B)                           P(B)
                //           /    \                         /   \
```

```
                //          P(R)    U(B)      ->          X(R)  G(R)
                //         /                                    \
                //        X(R)                                   X(B)
                (*stack_parent_ptr.peek())->color = 1;
                (*parent_node_ptr)->color = 0;
                right_rotate(node_ptr,                    parent_node_ptr,
stack_parent_ptr.peek());
                break;
            }
        } else {
            //  MY PARENT IS RIGHT
            if ((*stack_parent_ptr.peek())->left->color == 1) {
                //  MY UNCLE IS RED
                //             G(B)                          G(R)
                //            /    \                        /   \
                //          U(R)    P(R)      ->          P(B)  U(B)
                //                    |                           |
                //                   X(R)                        X(R)
                (*stack_parent_ptr.peek())->color = 1;
                (*stack_parent_ptr.peek())->left->color = 0;
                (*parent_node_ptr)->color = 0;
                if       ((*stack_parent_ptr.peek())       !=       root)
stack_parent_ptr.pop();
            } else {
                //  MY UNKLE IS BLACK
                if ((*parent_node_ptr)->left == *node_ptr) {
                    //             G(B)                          G(R)
                    //            /    \                        /   \
                    //          U(R)    P(R)      ->          P(B)   X(B)
                    //                    /                           \
                    //                  X(R)                           U(R)
                    right_rotate(node_ptr, node_ptr, parent_node_ptr);
                }
                //             G(B)                          P(B)
                //            /    \                        /   \
                //          U(R)    P(R)      ->          G(R)   X(R)
                //                    \                   /
                //                     X(R)             U(B)
                (*stack_parent_ptr.peek())->color = 1;
                (*parent_node_ptr)->color = 0;
                left_rotate(node_ptr,                    parent_node_ptr,
stack_parent_ptr.peek());
                break;
            }
        }
        root->color = 0;
        node_ptr = parent_node_ptr;
        parent_node_ptr = stack_parent_ptr.peek();
        count = *stack_parent_ptr.peek();
    }
}


void RedBlackTree:: insert(int key) {
    NodePtr node = new Node;
    node->data = key;
    node->left = TNULL;
    node->right = TNULL;
```

```
    node->color = 1;
    node->key = hash();
    NodePtrPtr parent_new_node_ptr = &(this->root);
    if (root == TNULL){
        root = node;
        root->color = 0;
        return;
    }
    do {
        stack_parent_ptr.push(&(*parent_new_node_ptr));
        if (((*parent_new_node_ptr)->data > key)){
            if( (*parent_new_node_ptr)->left == TNULL) break;
            parent_new_node_ptr = &((*parent_new_node_ptr)->left);
        } else {
            if( (*parent_new_node_ptr)->right == TNULL) break;
            parent_new_node_ptr = &((*parent_new_node_ptr)->right);
        }
    } while ((*parent_new_node_ptr) != TNULL );
    ((*parent_new_node_ptr)->data > key) ? &((*parent_new_node_ptr)->left
= node) :
    &((*parent_new_node_ptr)->right = node);
    NodePtrPtr  new_node_ptr = &node;
    insert_fix(parent_new_node_ptr, new_node_ptr);
    stack_parent_ptr.clear();
    return;
}


void  RedBlackTree::  right_rotate  (NodePtrPtr  node_ptr,  NodePtrPtr
parent_node_ptr, NodePtrPtr parent_parent_node_ptr){
    NodePtrPtr child = node_ptr; //сохраним ссылки для рекурсии
    if (*parent_parent_node_ptr == nullptr){
        const NodePtr buffer = (*node_ptr)->right; // =b
        root = (*node_ptr); // x=y -> y is root
        root->right = (*parent_node_ptr);
        (*parent_node_ptr)->left = buffer;
        return;
    }
    const NodePtr buffer = (*parent_node_ptr)->right; // = c
    const NodePtr ded = (*parent_parent_node_ptr); // = x
    (*parent_parent_node_ptr) = *parent_node_ptr; // x= y
    (*parent_parent_node_ptr)->right = ded;
    ded->left= buffer;
}


void  RedBlackTree::  left_rotate  (NodePtrPtr  node_ptr,  NodePtrPtr
parent_node_ptr, NodePtrPtr parent_parent_node_ptr) {
    NodePtrPtr child = node_ptr;
    if ((*parent_parent_node_ptr) == nullptr){
        const NodePtr buffer = (*node_ptr)->left; // = a
        this->root = *node_ptr; // x = y
        root->left = *parent_node_ptr; // a = x
        root->left->right = buffer; // +a
        return;
    }
    const NodePtr buffer = (*parent_node_ptr)->left;
    const NodePtr ded = (*parent_parent_node_ptr);
```

```
    *parent_parent_node_ptr = *parent_node_ptr;
    (*parent_parent_node_ptr)->left = ded;
    ded->right = buffer;
    parent_node_ptr = child;
}


NodePtrPtr RedBlackTree:: min(NodePtrPtr node) {
    while ((*node)->left != TNULL){
        node = &((*node)->left);
    }
    return node;
}


NodePtrPtr RedBlackTree:: parent_min(NodePtrPtr   node) {
    while ((*node)->left->left != TNULL)
        node = &((*node)->left);
    return node;
}


void RedBlackTree:: del( int key) {
    NodePtrPtr node_to_del_ptr = &(this->root);
    while ((*node_to_del_ptr) != TNULL){
        if ((*node_to_del_ptr)->data == key) break;
        node_to_del_ptr    =    ((*node_to_del_ptr)->data    >    key)    ?
&((*node_to_del_ptr)->left) : &((*node_to_del_ptr)->right);
    }
    if ((*node_to_del_ptr) == TNULL) return;
    int color_node_to_del = (*node_to_del_ptr)->color;
    if ((*node_to_del_ptr)->right == TNULL) {
        if ((*node_to_del_ptr)->left == TNULL){
            (*node_to_del_ptr)->color = color_node_to_del;
            *node_to_del_ptr = (*node_to_del_ptr)->left;
            if(color_node_to_del   ==   0)   del_fix(   (*node_to_del_ptr),
key);
            stack_parent_ptr.clear();
            return;
        }
        (*node_to_del_ptr)->color = color_node_to_del;
        *node_to_del_ptr = (*node_to_del_ptr)->left;
        return;
    }
    if ((*node_to_del_ptr)->left == TNULL) {
        *node_to_del_ptr = (*node_to_del_ptr)->right;
        (*node_to_del_ptr)->color = color_node_to_del;
        return;
    }
    if ((*node_to_del_ptr)->right->left == TNULL){  //ОСОБЫЙ СЛУЧАЙ
        //           | ->A
        //         N1
        //    E<-/    \->B
        //     N2      N3
        //         R<-/   \
        //         TNULL    N4
        (*node_to_del_ptr)->right->left = (*node_to_del_ptr)->left; // R
= E
```

```
        (*node_to_del_ptr) = (*node_to_del_ptr)->right; //A = B
        (*node_to_del_ptr)->color = color_node_to_del;
        (*node_to_del_ptr)->color = color_node_to_del;
        if(color_node_to_del == 0) del_fix( (*node_to_del_ptr),  key);
        stack_parent_ptr.clear();
        return;
    }
    NodePtrPtr minimum = (min(&((*node_to_del_ptr)->right)));
    int color_minimum = (*minimum)->color;
    int minimum_data = (*minimum)->data;
    (*minimum)->right->color = color_minimum;
    //          | ->A                                                  A = R
    //         N1                                                    D = E
    //    E<-/    \->B                                               T = B
    //     N2      N3                                                R = T
    //         R<-/   \
    //          MIN    N4
    //      D <-/ \->T
    //              N6
    const NodePtr buffer = *node_to_del_ptr; // = A
    const NodePtr buffer_min = (*minimum)->right; // = T
    const  NodePtrPtr  parent_min_ptr  =  parent_min(&((*node_to_del_ptr)-
>right)); // = B
    (*node_to_del_ptr) = *minimum; // A = R
    (*minimum)->left = (buffer)->left; // D = E
    (*minimum)->right = (buffer)->right; // T = B
    (*parent_min_ptr)->left = buffer_min ; //R = T
    (*node_to_del_ptr)->color = color_node_to_del;
    if(color_node_to_del   ==   0)   del_fix(   (*parent_min_ptr)->left,
minimum_data);
    stack_parent_ptr.clear();
}


void RedBlackTree::del_fix(NodePtr node_to_del, int key) {
    NodePtrPtr  node = {&this->root};
    while (*node != TNULL){
        if (*node == node_to_del)break;
        node  =  ((*node)->data  >  key)  ?  &((*node)->left)  :  &((*node)-
>right);
        stack_parent_ptr.push(node);
    }
    if (*node != TNULL) stack_parent_ptr.push(node);
    NodePtrPtr parent = node;
    while       ((*stack_parent_ptr.peek())       !=       root       &&
(*stack_parent_ptr.peek())->color == 0){
        if ((*stack_parent_ptr.peek()) == TNULL) stack_parent_ptr.pop();
        if ( (*stack_parent_ptr.peek())->left == *parent){
            // I AM LEFT
            if ((*stack_parent_ptr.peek())->right->color == 1){
                // MY BROTHER IS RED
                //             G(B)                          P(B)
                //            /    \                        /   \
                //          P(B)    U(R)      ->         X(B)  G(R)
                //          |                                    |
                //         X(B)                                  U(B)
                //#4
                (*stack_parent_ptr.peek())->right->color = 0;
```

```
                (*stack_parent_ptr.peek())->color = 1;
                left_rotate(node,           &((*stack_parent_ptr.peek())-
>left),stack_parent_ptr.peek());
            }
            if ((*stack_parent_ptr.peek())->right->right->color == 0 and
(*stack_parent_ptr.peek())->right->left->color == 0) {
                //  BOTH NEPHEWS ARE BLACK
                //             G(?)                          G(B)
                //            /    \                        /   \
                //          X(B)    U(R)      ->          X(B)   U(R)
                //                    /  \                          /  \
|
                //               n1(B) n2(B)                  n1(B) n2(B)
                //#5
                (*stack_parent_ptr.peek())->right->color = 1;
                (*stack_parent_ptr.peek())->color = 0;
                parent = stack_parent_ptr.peek();
                stack_parent_ptr.pop();
            } else {
                if  (  (*stack_parent_ptr.peek())->right->right->color  ==
0){
                    //  RIGHT NEPHEWS is BLACK
                    //             G(?)                          G(B)
                    //            /    \                        /   \
                    //          X(B)    U(B)      ->          X(B)   n1(B)
                    //                    /  \                          \
                    //                                       n1(R)   n2(B)
U(R)
                    //
\
                    //
n2(B)
                    (*stack_parent_ptr.peek())->right->left->color = 0;
                    (*stack_parent_ptr.peek())->right->color = 1;
                    right_rotate(node,&((*stack_parent_ptr.peek())-
>right->right),&((*stack_parent_ptr.peek())->right) );
                }
                //  RIGHT NEPHEWS is RED
                //             G(?)                          U(color-G)
                //            /    \                        /   \
                //          X(B)    U(B)      ->          G(B)   n2(B)
                //                    /  \                /  \
                //               n1(?) n2(R)          X(B)  n2(B)
                (*stack_parent_ptr.peek())->right->color
=(*stack_parent_ptr.peek())->color;
                (*stack_parent_ptr.peek())->right->right->color = 0;
                (*stack_parent_ptr.peek())->color = 0;
                left_rotate(node,&((*stack_parent_ptr.peek())-
>right),stack_parent_ptr.peek() );
                return;
            }
        } if ( (*stack_parent_ptr.peek())->right== *parent) {
            //  I AM RIGHT
            if ((*stack_parent_ptr.peek())->left->color == 1){
                //  MY BROTHER IS RED
                (*stack_parent_ptr.peek())->left->color = 0;
                (*stack_parent_ptr.peek())->color = 1;
```

```
                right_rotate(node,          &((*stack_parent_ptr.peek())-
>right),stack_parent_ptr.peek());
            }
            if((*stack_parent_ptr.peek())->left->left->color  ==  0  and
(*stack_parent_ptr.peek())->left->right->color == 0 ){
                //  BOTH NEPHEWS ARE BLACK
                (*stack_parent_ptr.peek())->left->color = 1;
                (*stack_parent_ptr.peek())->color = 0;
                parent = stack_parent_ptr.peek();
                stack_parent_ptr.pop();
            } else {
                if ((*stack_parent_ptr.peek())->left->left->color == 0) {
                    //  LEFT NEPHEWS is BLACK
                    (*stack_parent_ptr.peek())->left->right->color = 0;
                    (*stack_parent_ptr.peek())->left->color = 1;
                    left_rotate(node,       &((*stack_parent_ptr.peek())-
>right->right), &((*stack_parent_ptr.peek())->right));
                }
                //  LEFT NEPHEWS is RED
                (*stack_parent_ptr.peek())->left->color                 =
(*stack_parent_ptr.peek())->color;
                (*stack_parent_ptr.peek())->color = 0;
                (*stack_parent_ptr.peek())->left->left->color = 0;
                right_rotate(node,   &((*stack_parent_ptr.peek())->left),
stack_parent_ptr.peek());
                break;
            }
        }
    }
}


bool RedBlackTree::search(int data_) {
    NodePtr root_help = this->root;
    while (root_help != TNULL){
        if (root_help->data == data_) return true;
        root_help  =  (root_help->data  >  data_)  ?  root_help->left  :
root_help->right;
    }
    return false;
}


void RedBlackTree:: print(NodePtr root, std::string indent, bool last) {
    if (root != TNULL) {
        std::cout << indent;
        if (last) {
            std::cout << "R----";
            indent += "   ";
        } else {
            std::cout << "L----";
            indent += "|  ";
        }
        std::string sColor = root->color ? "RED" : "BLACK";
        std::cout << root->data << "(" << sColor << ")" << "\n";
        print(root->left, indent, false);
        print(root->right, indent, true);
    }
```

```
}


void RedBlackTree:: print() {
    if (root) {
        print(this->root, "", true);
    }
}


int main() {
    return 0;
}
```

Литература

1. Хайков, Д. В. Сравнение структур хранения данных «двоичное дерево», «красно-черное дерево» и «B-дерево» / Д. В. Хайков, Р. С. Пипенко // Новое слово в науке и образовании : Материалы Международной (заочной) научно-практической конференции, Минск, 26 сентября 2022 года. – Нефтекамск: Научно-издательский центр "Мир науки" (ИП Вострецов Александр Ильич), 2022. – С. 7-10. – EDN AFWGLD

2. Гордиенко, А. П. Функциональная реализация красно-черного дерева / А. П. Гордиенко, О. В. Амелина // Информационные системы и технологии 2015 : Материалы III Международной научно-технической интернет-конференции, Орел, 01 апреля – 31 2015 года / ФГБОУ ВПО "Государственный университет-учебно-научно-производственный комплекс". – Орел: Общество с ограниченной ответственностью "Стерх", 2015. – С. 76. – EDN UMCGRH..

3. Чуприна, Ю. А. Визуализация красно-черных деревьев / Ю. А. Чуприна, С. Д. к. Халилова // . – 2021. – № 4(34). – С. 116-124. – EDN OAVAUW.

4. Филоненко, И. Н. Применение красно-черных двоичных деревьев поиска в современных информационных системах / И. Н. Филоненко, М. В. Чеботарев // Вестник Коломенского института (филиала) Московского политехнического университета. Серия: Естественные и технические науки. – 2017. – № 10. – С. 131-140. – EDN YUDWMH.

5. Логачёв О. А., Сальников А. А., Ященко В. В. Булевы функции в теории кодирования и криптологии. М. : МЦНМО, 2004. 470 с