Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

ИНСТИТУТ КОСМИЧЕСКИХ И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

ОТЧЕТ
по дисциплине Алгоритмы и структуры данных
Практическая работа №6 — Деревья

Преподаватель _____ Матковский И. В.
подпись, дата инициалы, фамилия

Студент КИ19-07Б, 031941597 _____ Горбацевич А. А.
номер группы, зачётной книжки подпись, дата инициалы, фамилия

Красноярск 2020

Содержание

1. Задание на работу

1.1 Разработать для решения поставленной задачи алгоритм; реализовать полученный алгоритм с использованием обычных, красно-черных и АВЛ-деревьев . Оценить сложность полученных алгоритмов.

2. Задание на вариант

2.1 Слить два дерева в одно.

3. Исходный код программы

```
// dsaa_04.cpp
// Горбацевич Андрей
#include <iostream>
#include <chrono>


inline void time_passed(std::chrono::high_resolution_clock::time_point start, double& holder) {
    auto stop = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);
    holder = duration.count();
}

#define BLACK 0u
#define RED 1u

class RBTree {
public:
    struct Node {
        Node *parent = nullptr;
        Node *left = nullptr;
        Node *right = nullptr;

        unsigned int color : 1 = BLACK;
        int val = -1;
    };

private:
    Node *root;

    static Node* parent(Node *n) {
        return n->parent;
    }

    static Node* sibling(Node *n) {
        auto p = RBTree::parent(n);

        if (p != nullptr) {
            return n == p->left? p->right : p->left;
        }
        return nullptr;
    }

    static Node* parent_sibling(Node *n) {
        return RBTree::sibling(RBTree::parent(n));
    }

    static void rotate_left(Node *n) {
        auto nm = n->right;
        auto p = RBTree::parent(n);

        n->right = nm->left;
        nm->left = n;
        n->parent = nm;
        if (n->right != nullptr) {
```

```
        n->right->parent = n;
    }

    if (p != nullptr) {
        if (n == p->left) {
            p->left = nm;
        }
        else if (n == p->right) {
            p->right = nm;
        }
    }

    nm->parent = p;
}

static void rotate_right(Node *n) {
    auto nm = n->left;
    auto p = RBTree::parent(n);

    n->left = nm->right;
    nm->right = n;
    n->parent = nm;
    if (n->left != nullptr) {
        n->left->parent = n;
    }

    if (p != nullptr) {
        if (n == p->left) {
            p->left = nm;
        }
        else if (n == p->right) {
            p->right = nm;
        }
    }

    nm->parent = p;
}

void _insert_rec(Node *sub_root, Node *n) {
    if (sub_root != nullptr) {
        if (n->val < sub_root->val) {
            if (sub_root->left != nullptr) {
                this->_insert_rec(sub_root->left, n);
                return;
            }
            else {
                sub_root->left = n;
            }
        }
        else {
            if (sub_root->right != nullptr) {
                this->_insert_rec(sub_root->right, n);
                return;
            }
            else {
                sub_root->right = n;
            }
        }
    }
```

```
    n->parent = sub_root;
    n->left = n->right = nullptr;
    n->color = RED;
}

void _fix_tree(Node *n) {
    if (n->color != BLACK && RBTree::parent(n) == nullptr) {
        // at the top, need to repaint
        n->color = BLACK;
    }
    else if (RBTree::parent(n)->color == BLACK) {
        // ok
        return;
    }
    else if (RBTree::parent_sibling(n) != nullptr && RBTree::parent_sibling(n)->color == RED) {
        // if parent and it's sibling red, then we can repaint them black and recursively repaint grandparent
        RBTree::parent(n)->color = BLACK;
        RBTree::parent_sibling(n)->color = BLACK;
        RBTree::parent(RBTree::parent(n))->color = RED;
        _fix_tree(RBTree::parent(RBTree::parent(n)));
    }
    else {
        // it may happen what parent.color != parent_sibling.color, then we can rotate twice to fix that and retain
        // structure
        auto p = RBTree::parent(n);
        auto gp = RBTree::parent(p);

        if (n == p->right && p == gp->left) {
            RBTree::rotate_left(p);
            n = n->left;
        } else if (n == p->left && p == gp->right) {
            RBTree::rotate_right(p);
            n = n->right;
        }

        // rotating second time, thus getting everything in order
        p = RBTree::parent(n);
        gp = RBTree::parent(p);

        if (n == p->left) {
            RBTree::rotate_right(gp);
        }
        else {
            RBTree::rotate_left(gp);
        }
        p->color = BLACK;
        gp->color = RED;
    }
}

void _print_tree(Node *n) {
    if (n == nullptr) {
        return;
    }

    printf("    %d\n", n->val);
```

```
        if (n->left != nullptr) {
            printf("    %d -> %d\n", n->val, n->left->val);
            this->_print_tree(n->left);
        }

        if (n->right != nullptr) {
            printf("    %d -> %d\n", n->val, n->right->val);
            this->_print_tree(n->right);
        }
    }

public:
    bool contains(int value) {
        auto cn = this->root;

        while (cn != nullptr) {
            if (cn->val == value) {
                return true;
            }
            else if (cn->val > value) {
                cn = cn->left;
            }
            else if (cn->val < value) {
                cn = cn->right;
            }
        }

        return false;
    }

    void insert(Node *n) {
        if (this->contains(n->val)) {
            delete n;
            return;
        }

        this->_insert_rec(this->root, n);

        this->_fix_tree(n);

        this->root = n;
        while (RBTree::parent(this->root) != nullptr) {
            root = RBTree::parent(root);
        }
    }

    void insert(int val) {
        auto nn = new Node {
                nullptr, nullptr, nullptr, BLACK, val
        };

        this->insert(nn);
    }

    void merge(Node *n) {  // l-n-r (in-order) traversal
        if (n == nullptr) {
            return;
```

```
        }

        if (n->left != nullptr) {
            this->merge(n->left);
        }

        this->insert(n->val);

        if (n->right != nullptr) {
            this->merge(n->right);
        }
    }

    void merge(RBTree &tree) {
        this->merge(tree.root);
    }

    void print_tree(const std::string& prefix) {
        printf("digraph %s {\n", prefix.c_str());
        this->_print_tree(this->root);
        printf("}\n");
    }
};

class AVLTree {
public:
    struct Node {
        Node *left = nullptr;
        Node *right = nullptr;

        int val = -1;
        int height = 1;
    };

    enum balance_type {
        LEFT_SKEWD = -1,
        BALANCED = 0,
        RIGHT_SKEWD = 1
    };

private:
    Node *root;

    static int height(Node *n) {
        if (n == nullptr) {
            return 0;
        }

        return n->height;
    }

    static balance_type balance(Node *n) {
        if (n == nullptr) {
            return balance_type::BALANCED;
        }

        int diff = AVLTree::height(n->left) - AVLTree::height(n->right);
        return diff == 2? balance_type::LEFT_SKEWD : (diff == -2? balance_type::RIGHT_SKEWD :
```

```
balance_type::BALANCED);
    };

    static Node* small_left_rotation(Node *n) {
        Node *rst = n->right;
        Node *lrst = rst->left;

        // Perform rotation
        rst->left = n;
        n->right = lrst;

        // Update heights
        n->height = std::max(AVLTree::height(n->left), AVLTree::height(n->right)) + 1;
        rst->height = std::max(AVLTree::height(rst->left), AVLTree::height(rst->right)) + 1;

        // Return new root
        return rst;
    }

    static Node* small_right_rotation(Node *n) {
        Node *lst = n->left;
        Node *rlst = lst->right;

        // Perform rotation
        lst->right = n;
        n->left = rlst;

        // Update heights
        n->height = std::max(AVLTree::height(n->left), AVLTree::height(n->right)) + 1;
        lst->height = std::max(AVLTree::height(lst->left), AVLTree::height(lst->right)) + 1;

        // Return new root
        return lst;
    }

    Node* _internal_insert(Node *sub_root, Node *n) {
        if (sub_root == nullptr) {
            return n;
        }

        if (sub_root->val > n->val) {
            sub_root->left = this->_internal_insert(sub_root->left, n);
        }
        else {
            sub_root->right = this->_internal_insert(sub_root->right, n);
        }

        sub_root->height = std::max(AVLTree::height(sub_root->left), AVLTree::height(sub_root->right)) + 1;

        balance_type nb = AVLTree::balance(sub_root);

        if (nb == balance_type::LEFT_SKEWD) {
            if (n->val < sub_root->left->val) {  // l-l
                return AVLTree::small_right_rotation(sub_root);
            }
            else if (n->val > sub_root->left->val) {  // l-r
                sub_root->left = AVLTree::small_left_rotation(sub_root->left);
                return AVLTree::small_right_rotation(sub_root);
```

```
            }
        }
        else if (nb == balance_type::RIGHT_SKEWD) {
            if (n->val > sub_root->right->val) {  // r-r
                return AVLTree::small_left_rotation(sub_root);
            }
            else if (n->val < sub_root->right->val) {  // r-l
                sub_root->right = AVLTree::small_right_rotation(sub_root->right);
                return AVLTree::small_left_rotation(sub_root);
            }
        }

        return sub_root;
    }

    void _print_tree(Node *n) {
        if (n == nullptr) {
            return;
        }

        printf("    %d\n", n->val);

        if (n->left != nullptr) {
            printf("    %d -> %d\n", n->val, n->left->val);
            this->_print_tree(n->left);
        }

        if (n->right != nullptr) {
            printf("    %d -> %d\n", n->val, n->right->val);
            this->_print_tree(n->right);
        }
    }

public:
    bool contains(int value) {
        auto cn = this->root;

        while (cn != nullptr) {
            if (cn->val == value) {
                return true;
            }
            else if (cn->val > value) {
                cn = cn->left;
            }
            else if (cn->val < value) {
                cn = cn->right;
            }
        }

        return false;
    }

    void insert(Node *node) {
        if (this->contains(node->val)) {
            delete node;
            return;
        }
```

```
        this->root = AVLTree::_internal_insert(this->root, node);
    }

    void insert(int val) {
        auto nn = new Node {
            nullptr, nullptr, val, 1
        };

        this->insert(nn);
    }

    void merge(Node *n) {  // l-n-r (in-order) traversal
        if (n == nullptr) {
            return;
        }

        if (n->left != nullptr) {
            this->merge(n->left);
        }

        this->insert(n->val);

        if (n->right != nullptr) {
            this->merge(n->right);
        }
    }

    void merge(AVLTree &tree) {
        this->merge(tree.root);
    }

    void print_tree(const std::string& prefix) {
        printf("digraph %s {\n", prefix.c_str());
        this->_print_tree(this->root);
        printf("}\n");
    }
};

#define DATASET 5

int main() {
#if DATASET == 1
    auto ftv = {5, 32, 8, 4, 21};
    auto stv = {14, 48, 88, 13, 33, 37};
#elif DATASET == 2
    auto ftv = {9123, 4409, 8243, 3504, 5432, 8943};
    auto stv = {4686, 232, 8780, 7792, 248, 632, 4122};
#elif DATASET == 3
    auto ftv = {9,8,7,6};
    auto stv = {1,2,3,4,5,6,7};
#else
    auto ftv = {1,2,3,4,5,6,7};
    auto stv = {8,9,10,11,12,13,14,15};
#endif
    RBTree rbt1{};
    RBTree rbt2{};
    RBTree rbt3{};
```

```
AVLTree avlt1{};
AVLTree avlt2{};
AVLTree avlt3{};

{
   printf("==========RED-BLACK TREE SECTION==========\n");
   printf("==========1. INSERTION====================\n");
   double et1, et2 = et1 = 0;
   {
      auto start = std::chrono::high_resolution_clock::now();
      for (int d : ftv) {
         rbt1.insert(d);
      }
      time_passed(start, et1);
   }
   {
      auto start = std::chrono::high_resolution_clock::now();
      for (int d : stv) {
         rbt2.insert(d);
      }
      time_passed(start, et2);
   }
   printf("%.0f %.0f\n", et1, et2);
   printf("==========2. MERGING======================\n");
   double et = 0;
   {
      auto start = std::chrono::high_resolution_clock::now();
      rbt3.merge(rbt1);
      rbt3.merge(rbt2);
      time_passed(start, et);
   }
   printf("%.0f\n", et);
   printf("====================END====================\n");
}

printf("\n\n");

{
   printf("==========AVL TREE SECTION================\n");
   printf("==========1. INSERTION====================\n");
   double et1, et2 = et1 = 0;
   {
      auto start = std::chrono::high_resolution_clock::now();
      for (int d : ftv) {
         avlt1.insert(d);
      }
      time_passed(start, et1);
   }
   {
      auto start = std::chrono::high_resolution_clock::now();
      for (int d : stv) {
         avlt2.insert(d);
      }
      time_passed(start, et2);
   }
   printf("%.0f %.0f\n", et1, et2);
   printf("==========2. MERGING======================\n");
   double et = 0;
   {
      auto start = std::chrono::high_resolution_clock::now();
```

```
            avlt3.merge(avlt1);
            avlt3.merge(avlt2);
            time_passed(start, et);
        }
        printf("%.0f\n", et);
        printf("====================END====================\n");
    }

    printf("==========PRINTING TREES==================\n");
    // all output done in dot-graph-notation
    rbt3.print_tree("rb3");
    avlt3.print_tree("avl3");
    printf("==========END PRINTING TREES==============\n");


    return 0;
}
```

4. Теоретические оценки временной сложности алгоритмов

4.1 Вставка: O(log n) (оба дерева);

4.2 Объединение: O(n log n) (оба дерева)

5. Экспериментальные оценки временной и пространственной сложности программы

Размер входного набора данных	Вставка, RBTree, микросекунды	Объединение, RBTree, микросекунды	Вставка, AVLTree, микросекунды	Объединение, AVLTree, микросекунды
5 и 6	0	0	0	0
6 и 7	0	0	0	0
4 и 7	0	0	0	0
40 и 42	0	0	0	0
80 и 80	0	0	0	0

Приложение А
Результаты работы программы

```
C:\Users\Admin\CLionProjects\instp_02d\cmake-build-debug\instp_02d.exe
==========RED-BLACK TREE SECTION==========
==========1. INSERTION====================
0 0
==========2. MERGING======================
0
==================END=====================


==========AVL TREE SECTION================
==========1. INSERTION====================
0 0
==========2. MERGING======================
0
==================END=====================
==========PRINTING TREES==================
[rb:21 {bg:slategray}]->[rb:5 {bg:crimson}]
[rb:5 {bg:crimson}]->[rb:4 {bg:slategray}]
[rb:4 {bg:slategray}]->[rb:4:_null_l {bg:gray}]
[rb:4 {bg:slategray}]
[rb:4 {bg:slategray}]->[rb:4:_null_r {bg:gray}]
[rb:5 {bg:crimson}]
[rb:5 {bg:crimson}]->[rb:13 {bg:slategray}]
[rb:13 {bg:slategray}]->[rb:8 {bg:crimson}]
[rb:8 {bg:crimson}]->[rb:8:_null_l {bg:gray}]
[rb:8 {bg:crimson}]
[rb:8 {bg:crimson}]->[rb:8:_null_r {bg:gray}]
[rb:13 {bg:slategray}]
[rb:13 {bg:slategray}]->[rb:14 {bg:crimson}]
[rb:14 {bg:crimson}]->[rb:14:_null_l {bg:gray}]
[rb:14 {bg:crimson}]
[rb:14 {bg:crimson}]->[rb:14:_null_r {bg:gray}]
[rb:21 {bg:slategray}]
[rb:21 {bg:slategray}]->[rb:33 {bg:crimson}]
[rb:33 {bg:crimson}]->[rb:32 {bg:slategray}]
[rb:32 {bg:slategray}]->[rb:32:_null_l {bg:gray}]
[rb:32 {bg:slategray}]
[rb:32 {bg:slategray}]->[rb:32:_null_r {bg:gray}]
[rb:33 {bg:crimson}]
[rb:33 {bg:crimson}]->[rb:48 {bg:slategray}]
[rb:48 {bg:slategray}]->[rb:37 {bg:crimson}]
[rb:37 {bg:crimson}]->[rb:37:_null_l {bg:gray}]
[rb:37 {bg:crimson}]
[rb:37 {bg:crimson}]->[rb:37:_null_r {bg:gray}]
[rb:48 {bg:slategray}]
[rb:48 {bg:slategray}]->[rb:88 {bg:crimson}]
[rb:88 {bg:crimson}]->[rb:88:_null_l {bg:gray}]
[rb:88 {bg:crimson}]
[rb:88 {bg:crimson}]->[rb:88:_null_r {bg:gray}]
[avl:14 {bg:orange}]
[avl:14 {bg:orange}]->[avl:8 {bg:orange}]
[avl:8 {bg:orange}]
[avl:8 {bg:orange}]->[avl:5 {bg:orange}]
[avl:5 {bg:orange}]
[avl:5 {bg:orange}]->[avl:4 {bg:orange}]
[avl:4 {bg:orange}]
[avl:4 {bg:orange}]->[avl:4:_null_l {bg:gray}]
[avl:4 {bg:orange}]->[avl:4:_null_r {bg:gray}]
[avl:5 {bg:orange}]->[avl:5:_null_r {bg:gray}]
[avl:8 {bg:orange}]->[avl:13 {bg:orange}]
[avl:13 {bg:orange}]
[avl:13 {bg:orange}]->[avl:13:_null_l {bg:gray}]
[avl:13 {bg:orange}]->[avl:13:_null_r {bg:gray}]
[avl:14 {bg:orange}]->[avl:37 {bg:orange}]
[avl:37 {bg:orange}]
[avl:37 {bg:orange}]->[avl:32 {bg:orange}]
[avl:32 {bg:orange}]
[avl:32 {bg:orange}]->[avl:21 {bg:orange}]
[avl:21 {bg:orange}]
[avl:21 {bg:orange}]->[avl:21:_null_l {bg:gray}]
[avl:21 {bg:orange}]->[avl:21:_null_r {bg:gray}]
[avl:32 {bg:orange}]->[avl:33 {bg:orange}]
[avl:33 {bg:orange}]
[avl:33 {bg:orange}]->[avl:33:_null_l {bg:gray}]
[avl:33 {bg:orange}]->[avl:33:_null_r {bg:gray}]
[avl:37 {bg:orange}]->[avl:48 {bg:orange}]
[avl:48 {bg:orange}]
[avl:48 {bg:orange}]->[avl:48:_null_l {bg:gray}]
[avl:48 {bg:orange}]->[avl:88 {bg:orange}]
[avl:88 {bg:orange}]
[avl:88 {bg:orange}]->[avl:88:_null_l {bg:gray}]
[avl:88 {bg:orange}]->[avl:88:_null_r {bg:gray}]
==========END PRINTING TREES==============

Process finished with exit code 0
```

Рисунок 1: результат работы программы

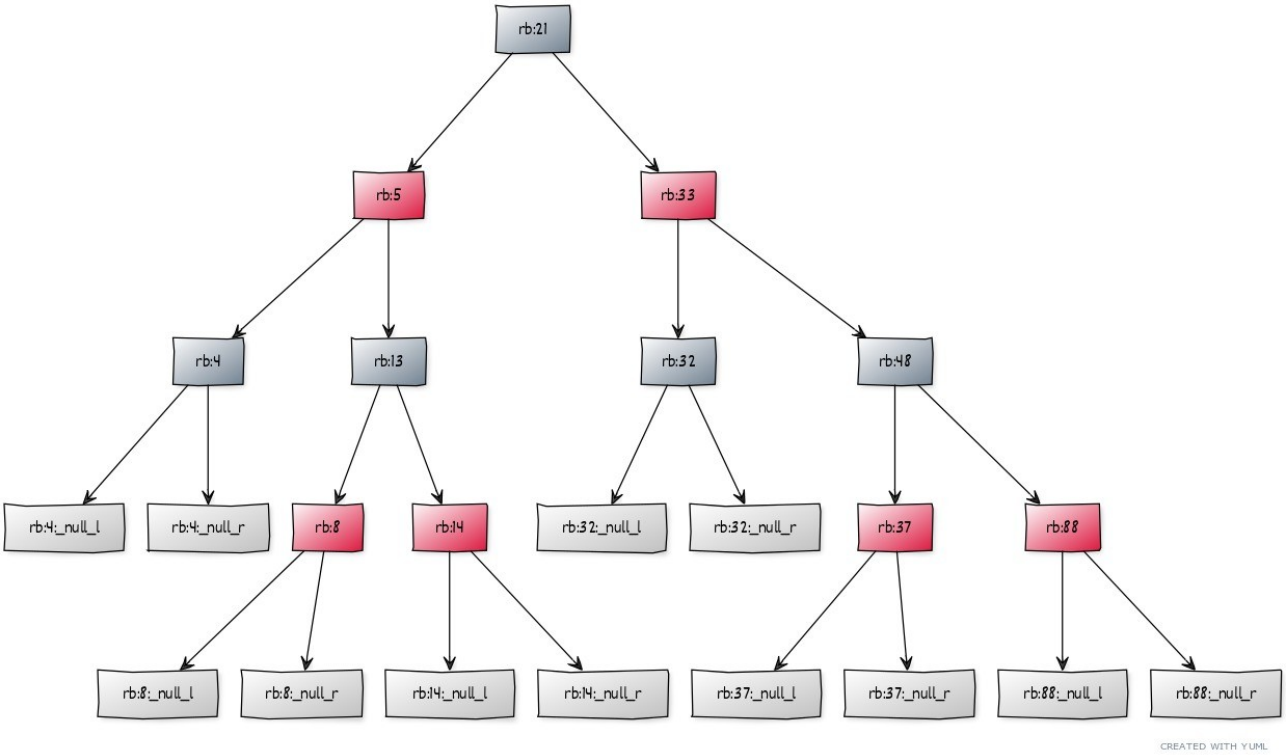

Рисунок 2: RBTree из входных данных

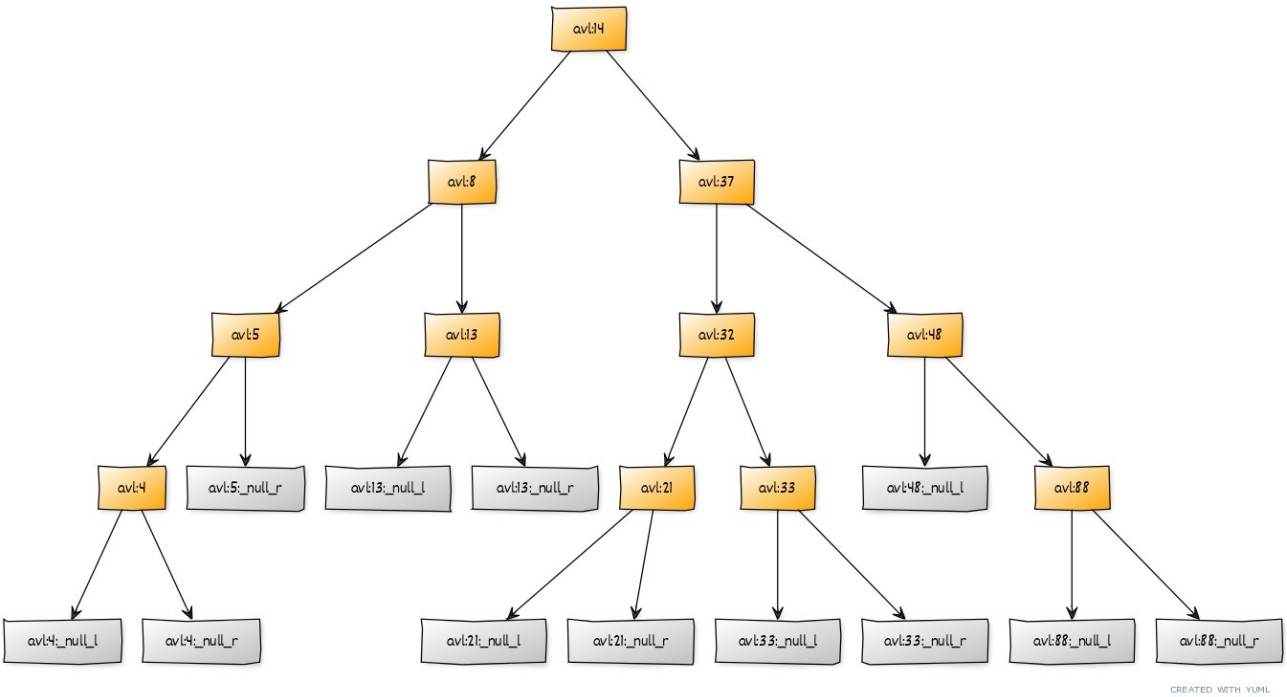

Рисунок 3: AVLTree из входных данных