МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева»

(Самарский университет)

Институт естественных и математических наук

Механико-математический факультет

Кафедра информатики и вычислительной математики

Направление подготовки

02.03.03 Математическое обеспечение и администрирование информационных систем

Направленность (профиль) «Разработка и администрирование информационных систем»

Курсовая работа

по дисциплине «Структуры и алгоритмы компьютерной обработки данных»

**Структура данных “Splay-дерево”**

|  |
| --- |
| Выполнил студент  курса 2 группы 4245-020303D  Кравчук Владислав Валерьевич  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Научный руководитель  к.ф.-м.н., доцент  Русакова М.С.  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Работа защищена  «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2020 г.  Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  зав. кафедрой ИиВМ  д.ф.-м.н., профессор  Степанов А.Н. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

Самара 2025

Содержание

[Введение 3](#_Toc199703855)

[1 Описание и анализ структур данных 4](#_Toc199703856)

[1.1 Термины и определения 4](#_Toc199703857)

[1.2 Описание структуры Splay-дерева 4](#_Toc199703858)

[1.3 Свойства Splay-дерева 5](#_Toc199703859)

[1.4 Области практического применения Splay-дерева 6](#_Toc199703860)

[2 Обзор алгоритмов работы со Splay-деревом 7](#_Toc199703861)

[2.1 Алгоритм добавления элемента в Splay-дерево 7](#_Toc199703862)

[2.2 Алгоритм поиска элемента в Splay-дереве 7](#_Toc199703863)

[2.3 Алгоритм удаления элемента из Splay-дерева 7](#_Toc199703864)

[2.4 Алгоритм поднятия узла в корень в Splay-дереве 8](#_Toc199703865)

[2.5 Оценка сложности алгоритма добавления, поиска и удаления элемента в Splay-дереве. 9](#_Toc199703866)

[3 Использование Splay-дерева при работе с автоматизированной информационной системой на железнодорожном вокзале 10](#_Toc199703867)

[3.1 Постановка задачи 10](#_Toc199703868)

[3.2 Алгоритм решения задачи 10](#_Toc199703869)

[3.3 Программная реализация алгоритма 11](#_Toc199703870)

[3.3.1 Реализация алгоритма построения Splay-дерева 11](#_Toc199703871)

[3.3.2 Проверка работоспособности программной реализации 12](#_Toc199703872)

[Заключение 14](#_Toc199703873)

[Список использованных источников 15](#_Toc199703874)

[ПРИЛОЖЕНИЕ А — программная реализация Splay-дерева 15](#_Toc199703875)

# Введение

В настоящее время с целью организации и систематизации данных разрабатываются разные алгоритмы и структуры данных, которые позволяют упростить процесс поиска данных. Одним из самых популярных и простых является бинарный поиск, который представляет собой алгоритм поиска элемента в отсортированном массиве данных.

С целью эффективного хранения и поиска данных, реализации алгоритмов сортировки и управления иерархическими структурами разработано бинарное дерево поиска. В зависимости от способа организации данных и алгоритма балансировки существует несколько видов двоичных деревьев: двоичные деревья поиска (BST), АВЛ-деревья, красно-черные деревья, Splay-деревья и так далее. Структурой данных, позволяющей быстрее находить те данные, которые использовались недавно, является разработанное в 1985 году Дэниелом Слейтером и Робертом Тарьяном Splay-дерево.

Целью работы является изучение алгоритмов обработки Splay-деревьев и особенностей их применения при решении практических задач.

Для достижения поставленной цели необходимо решить следующие задачи, которые обеспечивают освоение компетенции ОПК-2:

– изучить структуру данных Splay-дерева, ее особенности и свойства;

– провести анализ, оценку трудоемкости и асимптотической сложности алгоритмов работы со Splay-деревьями;

– провести проектирование, разработку и тестирование программного продукта, позволяющего осуществлять проверку того, является ли Splay-дерево эффективным при работе с автоматизированной информационной системой на железнодорожном вокзале.

# Описание и анализ структур данных

## Термины и определения

При описании структуры Splay-дерева используются следующие понятия и определения:

Бинарное дерево поиска — это иерархическая структура данных, в которой каждый узел имеет значение и ссылки на узлы левых и правых поддеревьев.

Поддерево — часть древообразной структуры данных, которая может быть представлена в виде отдельного дерева.

Корневой узел — узел на самом верхнем уровне, на котором начинается выполнение большинства операций над деревом (не имеющий предков).

Родительский узел — это узел, имеющий потомков.

Дочерний узел — который сам является потомком.

Листовой узел — это узел, у которого нет потомков: его правое и левое поддерево равно null.

## Описание структуры Splay-дерева

Splay-дерево является самобалансирующимся бинарным деревом поиска, при этом не является перманентно сбалансированным и на отдельных запросах может работать даже линейное время. Splay-дереву не нужно хранить дополнительную информацию (например, высота в АВЛ-дереве), что делает его эффективным по памяти. После каждого запроса Splay-дерево меняет свою структуру, что позволяет наиболее эффективно обрабатывать часто повторяющиеся запросы. [Асимптотическая](https://ru.wikipedia.org/wiki/%D0%90%D0%BC%D0%BE%D1%80%D1%82%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D1%8B%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7) сложность в большинстве случаев составляет .

Основным действием, которое производится над структурой, является операция Splay. Эта операция перемещает вершину с ключом x в корень дерева путем различных поворотов.

## Свойства Splay-дерева

Splay-дерево — это самобалансирующаяся структура данных, в которой последний ключ, к которому осуществлялся доступ, всегда помещается в корень. Дерево не обеспечивает балансировку (левое и правое поддеревья могу быть разной высоты). Отсутствие дополнительной информации в каждом узле обеспечивает наименьшие затраты памяти. Поднятие используемого угла в корень обеспечивает быстрый доступ при повторных обращениях. Поднятие узла в корень происходит с помощью «поворотов».

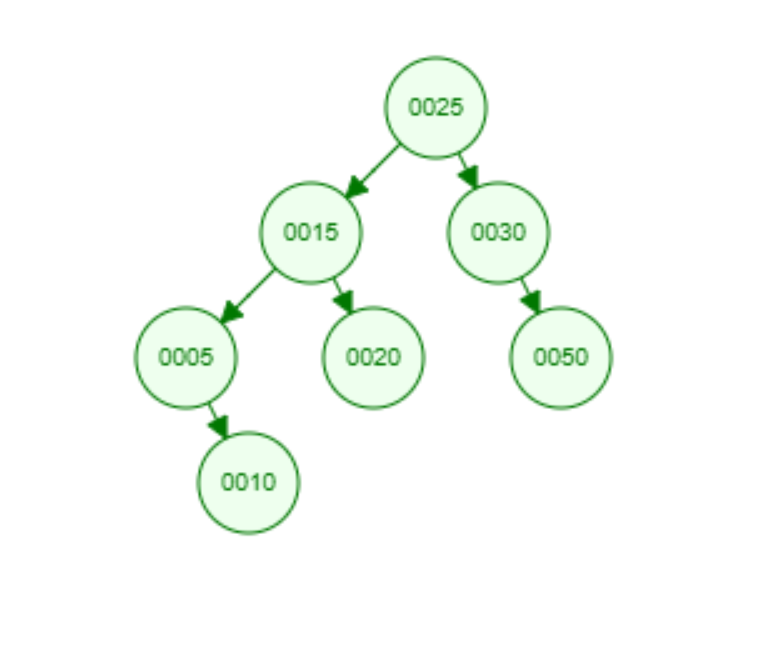
Каждый узел дерева содержит:

– ссылку на левое поддерево;

– ссылку на правое поддерево;

– полезную информацию.

На рисунке 1 представлен фрагмент Splay-дерева:



*Рисунок 1 – Splay-дерево*

Splay-дерево является бинарным деревом поиска, поэтому для него выполняются правила: ключ левого поддерева меньше, а ключ правого поддерева больше родительского узла.

## Области практического применения Splay-дерева

Splay-дерево — это разновидность саморегулирующегося бинарного дерева поиска, которое автоматически перемещает часто используемые узлы ближе к корню, обеспечивая быстрый доступ к ним, поэтому его используют при кэшировании данных, маршрутизации в сетях и в алгоритмах сжатия данных. Как правило, время доступа к узлу имеет асимптотическую сложность , но при обращении к ранее использующемуся узлу асимптотическая сложность принимает константное значение . При этом, Splay-дерево не использует дополнительные данные для его построения, что обеспечивает экономию памяти.

Однако использование Splay-дерева избегают в тех случаях, когда требуется строгое соблюдение времени выполнения, так как в худшем случае асимптотическая сложность равна .

# Обзор алгоритмов работы со Splay-деревом

## Алгоритм добавления элемента в Splay-дерево

Алгоритм вставки элемента в Splay-дерево представлен в приложении А.

Вставка узла в Splay-дерево с n узлами может быть выполнена за время . Вставка нового элемента происходит по правилам бинарного дерева поиска. После размещения узла он поднимается в корень.

## Алгоритм поиска элемента в Splay-дереве

Алгоритм поиска элемента в Splay-дереве представлен в приложении А.

При поиске элемента происходит сравнение его ключа с узлом дерева. Если ключи равны, то это искомый элемент. Если ключ искомого элемента меньше, происходит переход в левое поддерево. Если ключ искомого элемента больше, происходит переход в правое поддерево.

## Алгоритм удаления элемента из Splay-дерева

Алгоритм удаления элемента из Splay-дерева представлен в приложении А.

При удалении элемента из Splay-дерева узел поднимается в корень и удаляется по правилам бинарного дерева поиска.

## Алгоритм поднятия узла в корень в Splay-дереве

Алгоритм поднятия узла в корень в Splay-дерева представлен в приложении А.

В Splay-дереве используется 6 видов поворотов:

- малый левый поворот (zig)

- малый правый поворот (zag)

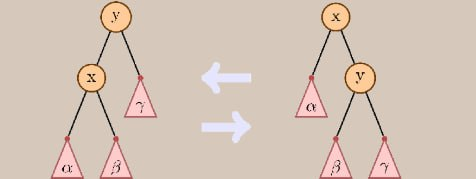
- большой левый поворот (zig-zig)

- большой правый поворот (zag-zag)

- левый-правый поворот (zig-zag)

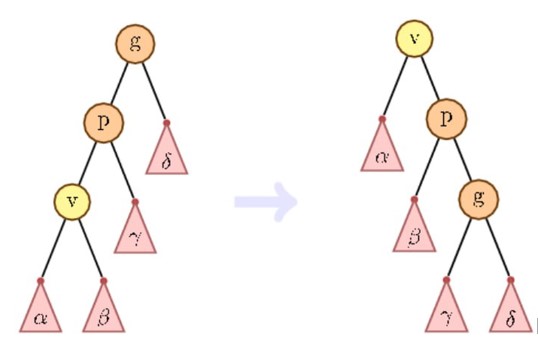
- правый-левый поворот (zag-zig)

Пример малого поворота в Splay-дереве представлен на рисунке 2:



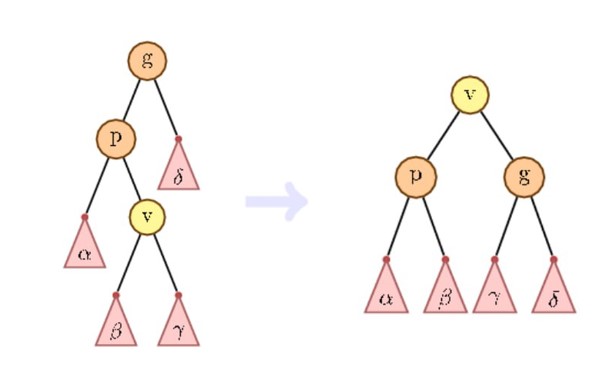
*Рисунок 2 – Малый правый поворот в Splay-дереве*

Пример большого правого поворота в Splay-дереве представлен на рисунке 3:



*Рисунок 3 – Большой правый поворот в Splay-дереве*

Пример левого-правого поворота в Splay-дереве представлен на рисунке 4:



*Рисунок 4 – Левый-правый поворот в Splay-дереве*

## Оценка сложности алгоритма добавления, поиска и удаления элемента в Splay-дереве

Операции добавления, поиска и удаления имеют одинаковую асимптотическую сложность. В лучшем случае это , в среднем — , в худшем — .

# Использование Splay-дерева при работе с автоматизированной информационной системой на железнодорожном вокзале

## Постановка задачи

Имеется автоматизированная информационная система на железнодорожном вокзале, которая содержит сведения об отправлении поездов дальнего следования и организована в виде Splay-дерева. Для каждого поезда указываются номер, станция назначения, время отправления. Разработать программу, которая обеспечит:

– первоначальный ввод данных в систему и формирование бинарного дерева;

– добавление и удаление информации о поездах;

– поиск информации о поезде по его номеру;

– вывод всего дерева.

## Алгоритм решения задачи

На вход подается файл с информацией о поездах: номер поезда, следующая станция и время отправления. Ключом каждого узла является номер поезда. Далее совершаются такие действия, как добавление и удаление информации о поездах, а также поиск информации о поезде по его номеру. Каждое действие сопровождается поднятием в корень узла, над которым оно выполняется. Таким образом узлы, к которым несколько раз происходит обращение, находятся в корне дерева, и их поиск занимает минимальное кол-во времени и имеет асимптотическую сложность .

Вывод информации о всех поездах осуществляется полным перебором дерева. Таким образом, асимптотическая сложность этого действия .

Добавление узла происходит по правилам бинарного дерева поиска. После размещения узла в дерево он поднимается в корень.

Поиск происходит по центральному перебору узлов (родительский узел, левое поддерево, правое поддерево). Найденный узел поднимается в корень. Средним значением асимптотической сложности является , в лучшем случае — , в худшем случае — .

Удаление происходит поиском удаляемого узла и поднятием его в корень с последующим удалением.

## Программная реализация алгоритма

Для программной реализации решения рассматриваемой задачи был выбран язык программирования C++ и среда разработки Visual Studio Code.

Среда разработки Visual Studio Code предоставляет широкий спектр возможностей для разработки программного обеспечения и высокую производительность.

Язык С++ — мощный универсальный язык программирования, обеспечивающий высокую производительность и контроль над системными ресурсами. Область применения включает его использование для разработки программного обеспечения, приложений реального времени, создания операционных систем и драйверов.

### Реализация алгоритма построения Splay-дерева

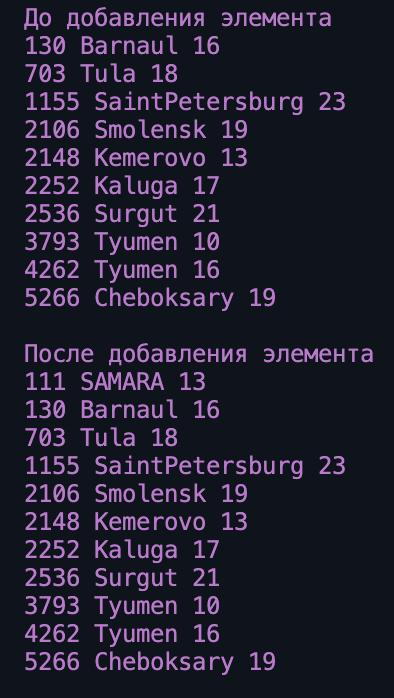
Структура Splay-дерева не отличается от структуры дерева бинарного поиска. Каждый узел хранит в себе полезную информацию и ссылки на левое и правое поддеревья. Поскольку для решения практической задачи требуется обеспечить хранение информации о каждом поезде, то для этого используется класс “Train”.

Все операции выполнены итерационным подходом, обеспечивающим большую скорость и меньшие затраты памяти по сравнению с рекурсивным подходом.

### Проверка работоспособности программной реализации

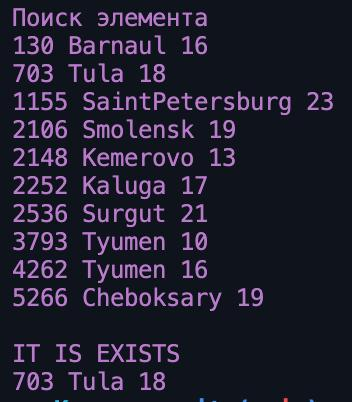
Тестирование программы проходило с использованием малого и большого количества данных. На небольшом количестве элементов были проверены все случаи операций добавления, поиска и удаления, балансирования и других операций. Далее была проведена проверка структуры на большом количестве элементов.

Проверка операции добавления осуществлялась при помощи добавления в исходное дерево элемента с номером поезда 111, с названием следующей станции SAMARA, со временем отправления 13. Результат представлен на рисунке 5.



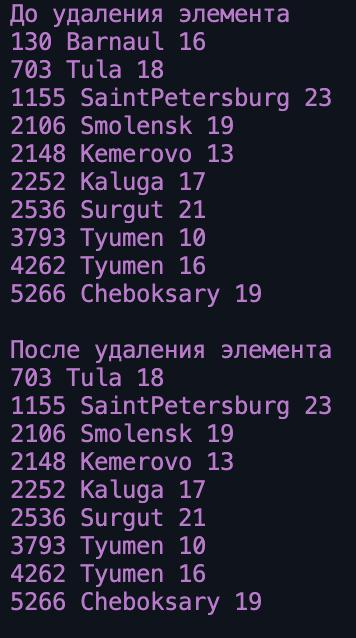
*Рисунок 5 – Результат проверки операции добавления*

Проверка операции поиска осуществлялась при помощи поиска в исходном дереве элемента под номером 703. Результат представлен на рисунке 6.



*Рисунок 6 – Результат проверки операции поиска*

Проверка операции удаления осуществлялась при помощи удаления в исходном дереве элемента с номером поезда 130. Результат представлен на рисунке 7.



*Рисунок 7 – Результат проверки операции удаления*

Анализируя результаты, можно сделать вывод, что программа работает корректно и Splay-дерево эффективно при работе с автоматизированной информационной системой на железнодорожном вокзале.

Код тестируемой программы представлен в приложении.

# Заключение

В процессе выполнения курсовой работы было сделано следующее:

1) Изучена структура данных Splay-дерева, ее особенности и свойства;

2) Проведен анализ алгоритма добавления элемента в Splay-дерево и его балансировки;

3) Проведен анализ алгоритма поиска элемента в Splay-дереве и его балансировки;

4) Проведен анализ алгоритма удаления элемента из Splay-дерева и его балансировки;

5) Проведена оценка трудоемкости и асимптотической сложности алгоритмов работы со Splay-деревьями;

6) Спроектирован, разработан и протестирован программный продукт, позволяющий осуществлять проверку того, является ли Splay-дерево эффективным при работе с автоматизированной информационной системой на железнодорожном вокзале.

Таким образом, в ходе выполнения курсовой работы были сформированы систематические знания способов применения современного математического аппарата и структур, используемых для хранения компьютерных данных в основных алгоритмах их обработки, сформированы умения применять математические методы, структуры и алгоритмы обработки компьютерных данных при проектировании и разработке программных продуктов, а также приобретены навыки применения математического аппарата, алгоритмов и различных структур данных при решении конкретных задач, что свидетельствует о том, что компетенция ОПК-2 освоена.

# Список использованных источников

1. Кнут, Д. Э. Искусство программирования [Текст]/Д. Э. Кнут – М.: Вильямс, 2002. – Т. 1. – 720 с.

2. Splay-дерево [Электронный ресурс]: Викиконспекты. – URL: https://neerc.ifmo.ru/wiki/index.php?title=Splay-дерево (дата обращения: 10.05.2025).

3. Splay-деревья [Электронный ресурс]: Коллективный блог программистов «Хабр». – URL: <https://habr.com/ru/company/JetBrains-education/blog/210296/> (дата обращения: 10.05.2025).

# ПРИЛОЖЕНИЕ А — программная реализация Splay-дерева

#include <clocale>

#include <fstream>

#include <iomanip>

#include <iostream>

#include <sstream>

#include <string>

using namespace std;

// класс, описывающий информацию поездов

class Train {

private:

int number; // номер поезда

string next\_station; // следущая станция

int departure\_time; // время отправления

public:

Train(int n, string s, int t) {

number = n;

next\_station = s;

departure\_time = t;

}

int get\_number() { return number; }

string get\_next\_station() { return next\_station; }

int get\_departure\_time() { return departure\_time; }

};

// Структура узла дерева

struct Node {

Train express;

Node\* Parent;

Node\* R\_Node;

Node\* L\_Node;

Node(Train& train) : express(train) {

R\_Node = nullptr;

L\_Node = nullptr;

Parent = nullptr;

}

};

class Splay\_Tree {

private:

Node\* root;

public:

Splay\_Tree() {

root = nullptr;

}

Node\* get\_root() { return root; }

// Проверка на пустоту

bool Is\_Empty() {

return root == nullptr;

}

// Добавление нового элента

void Add\_Knot(Node k) {

Node\* adder\_node = new Node(k);

Node\* current\_node = root;

if (Is\_Empty()) {

root = adder\_node;

} else {

bool flag = false;

while (not flag) {

if (adder\_node->express.get\_number() > current\_node->express.get\_number()) {

// cout << "ALARM\_ADD\n";

if (current\_node->R\_Node == nullptr) {

current\_node->R\_Node = adder\_node;

adder\_node->Parent = current\_node;

flag = true;

} else {

current\_node = current\_node->R\_Node;

}

} else {

if (current\_node->L\_Node == nullptr) {

current\_node->L\_Node = adder\_node;

adder\_node->Parent = current\_node;

flag = true;

} else {

current\_node = current\_node->L\_Node;

}

}

}

}

Move\_Knot\_to\_Root(adder\_node);

}

// Вывод всех элементов дерева

void Print\_All() {

if (root == nullptr) {

cout << "Дерево пустое\n";

return;

}

Node\* current\_node = root;

stack<Node\*> Train\_stack;

while (current\_node != nullptr || !Train\_stack.empty()) {

while (current\_node != 0) {

Train\_stack.push(current\_node);

current\_node = current\_node->L\_Node;

}

current\_node = Train\_stack.top();

Train\_stack.pop();

cout << current\_node->express.get\_number() << " " << current\_node->express.get\_next\_station()

<< " " << current\_node->express.get\_departure\_time() << "\n";

current\_node = current\_node->R\_Node;

}

}

// Поднятия узла в корень

void Move\_Knot\_to\_Root(Node\* c) {

Node\* current\_node = c;

while (current\_node != root) {

Node\* father = current\_node->Parent;

Node\* grand\_father;

if (current\_node->Parent == nullptr) {

grand\_father = nullptr;

root = current\_node;

break;

} else {

grand\_father = father->Parent;

}

if (grand\_father == nullptr) { // малый поворот

if (father->L\_Node == current\_node) {

father->L\_Node = current\_node->R\_Node;

if (current\_node->R\_Node != nullptr)

current\_node->R\_Node->Parent = father;

current\_node->R\_Node = father;

father->Parent = current\_node;

current\_node->Parent = nullptr;

} else {

father->R\_Node = current\_node->L\_Node;

if (current\_node->L\_Node != nullptr)

current\_node->L\_Node->Parent = father;

current\_node->L\_Node = father;

father->Parent = current\_node;

current\_node->Parent = nullptr;

}

} else {

// zig-zig левый

if (grand\_father->L\_Node == father && father->L\_Node == current\_node) {

grand\_father->L\_Node = father->R\_Node;

if (father->R\_Node != nullptr)

father->R\_Node->Parent = grand\_father;

father->L\_Node = current\_node->R\_Node;

if (current\_node->R\_Node != nullptr)

current\_node->R\_Node->Parent = father;

father->R\_Node = grand\_father;

current\_node->Parent = grand\_father->Parent;

if (current\_node->Parent != nullptr) {

if (current\_node->Parent->L\_Node == grand\_father)

current\_node->Parent->L\_Node = current\_node;

else

current\_node->Parent->R\_Node = current\_node;

}

grand\_father->Parent = father;

father->Parent = current\_node;

current\_node->R\_Node = father;

}

// zig-zig правый

else if (grand\_father->R\_Node == father && father->R\_Node == current\_node) {

grand\_father->R\_Node = father->L\_Node;

if (father->L\_Node != nullptr)

father->L\_Node->Parent = grand\_father;

father->R\_Node = current\_node->L\_Node;

if (current\_node->L\_Node != nullptr)

current\_node->L\_Node->Parent = father;

father->L\_Node = grand\_father;

current\_node->Parent = grand\_father->Parent;

if (current\_node->Parent != nullptr) {

if (current\_node->Parent->L\_Node == grand\_father)

current\_node->Parent->L\_Node = current\_node;

else

current\_node->Parent->R\_Node = current\_node;

}

grand\_father->Parent = father;

father->Parent = current\_node;

current\_node->L\_Node = father;

}

// zig-zag левый

else if (grand\_father->L\_Node == father && father->R\_Node == current\_node) {

grand\_father->L\_Node = current\_node->R\_Node;

if (current\_node->R\_Node != nullptr)

current\_node->R\_Node->Parent = grand\_father;

father->R\_Node = current\_node->L\_Node;

if (current\_node->L\_Node != nullptr)

current\_node->L\_Node->Parent = father;

current\_node->Parent = grand\_father->Parent;

if (current\_node->Parent != nullptr) {

if (current\_node->Parent->L\_Node == grand\_father)

current\_node->Parent->L\_Node = current\_node;

else

current\_node->Parent->R\_Node = current\_node;

}

current\_node->L\_Node = father;

current\_node->R\_Node = grand\_father;

father->Parent = current\_node;

grand\_father->Parent = current\_node;

}

// zig-zag правый

else if (grand\_father->R\_Node == father && father->L\_Node == current\_node) {

grand\_father->R\_Node = current\_node->L\_Node;

if (current\_node->L\_Node != nullptr)

current\_node->L\_Node->Parent = grand\_father;

father->L\_Node = current\_node->R\_Node;

if (current\_node->R\_Node != nullptr)

current\_node->R\_Node->Parent = father;

current\_node->Parent = grand\_father->Parent;

if (current\_node->Parent != nullptr) {

if (current\_node->Parent->L\_Node == grand\_father)

current\_node->Parent->L\_Node = current\_node;

else

current\_node->Parent->R\_Node = current\_node;

}

current\_node->L\_Node = grand\_father;

current\_node->R\_Node = father;

father->Parent = current\_node;

grand\_father->Parent = current\_node;

}

}

}

}

// Поиск узла

Node\* Find\_Train(int number) {

Node\* current\_node = root;

bool flag = false;

while (not flag) {

// cout << "!@#";

// cout << current\_node->express.get\_number() << "\n";

if (current\_node->express.get\_number() == number) {

Move\_Knot\_to\_Root(current\_node);

flag = true;

return current\_node;

} else if (current\_node->express.get\_number() < number) {

current\_node = current\_node->R\_Node;

} else if (current\_node->express.get\_number() > number) {

current\_node = current\_node->L\_Node;

}

if (current\_node == nullptr) {

flag = false;

return nullptr;

}

}

return 0;

}

// Нахождение максимального ключа

Node\* Find\_Max(Node\* c) {

Node\* current\_node = c;

while (current\_node->R\_Node != nullptr)

current\_node = current\_node->R\_Node;

return current\_node;

}

// Нахождение минимального ключа

Node\* Find\_Min(Node\* c) {

Node\* current\_node = c;

while (current\_node->L\_Node != nullptr)

current\_node = current\_node->L\_Node;

return current\_node;

}

// Поиск по значению

Node\* Find\_By\_Station(string station) {

Node\* current\_root = root;

stack<Node\*> Train\_stack;

bool flag = false;

while (current\_root != nullptr) {

// cout << current\_root->express.get\_number() << " " << current\_root->express.get\_next\_station()

// << " " << current\_root->express.get\_departure\_time() << "\n";

if (current\_root->express.get\_next\_station() == station) {

flag = true;

return current\_root;

}

if (current\_root->L\_Node != nullptr) {

if (current\_root->R\_Node != nullptr) {

Train\_stack.push(current\_root->R\_Node);

}

current\_root = current\_root->L\_Node;

} else if (current\_root->R\_Node != nullptr) {

current\_root = current\_root->R\_Node;

} else {

if (not Train\_stack.empty()) {

current\_root = Train\_stack.top();

Train\_stack.pop();

} else {

current\_root = nullptr;

}

}

}

if (not flag)

return nullptr;

return 0;

}

// Удаления узла

void Pop\_Knot(int number) {

Node\* deleted\_knot = Find\_Train(number);

if (deleted\_knot == nullptr)

return;

Move\_Knot\_to\_Root(deleted\_knot);

// Node\* current\_node = deleted\_knot;

if (deleted\_knot->L\_Node == nullptr && deleted\_knot->R\_Node == nullptr) {

root = nullptr;

delete deleted\_knot;

} else if (deleted\_knot->L\_Node != nullptr && deleted\_knot->R\_Node == nullptr) {

Node\* current\_node = deleted\_knot->L\_Node;

root = current\_node;

current\_node->Parent = nullptr;

delete deleted\_knot;

} else if (deleted\_knot->L\_Node == nullptr && deleted\_knot->R\_Node != nullptr) {

Node\* current\_node = deleted\_knot->R\_Node;

root = current\_node;

current\_node->Parent = nullptr;

delete deleted\_knot;

} else if (deleted\_knot->L\_Node != nullptr && deleted\_knot->R\_Node != nullptr) {

cout << deleted\_knot->L\_Node->express.get\_number() << " " << deleted\_knot->R\_Node->express.get\_number() << "\n";

Node\* current\_node = deleted\_knot->L\_Node;

current\_node = Find\_Max(current\_node);

if (current\_node->Parent->R\_Node == current\_node) {

current\_node->Parent->R\_Node = current\_node->L\_Node;

if (current\_node->L\_Node != nullptr)

current\_node->L\_Node->Parent = current\_node->Parent;

current\_node->L\_Node = deleted\_knot->L\_Node;

deleted\_knot->L\_Node->Parent = current\_node;

current\_node->R\_Node = deleted\_knot->R\_Node;

if (deleted\_knot->R\_Node != nullptr)

deleted\_knot->R\_Node->Parent = current\_node;

} else {

current\_node->R\_Node = deleted\_knot->R\_Node;

if (deleted\_knot->R\_Node != nullptr)

deleted\_knot->R\_Node->Parent = current\_node;

}

root = current\_node;

current\_node->Parent = nullptr;

delete deleted\_knot;

}

}

void Clear\_Tree() {

if (root == nullptr)

return;

stack<Node\*> nodeStack;

Node\* current = root;

Node\* lastVisited = nullptr;

while (current != nullptr || !nodeStack.empty()) {

while (current != nullptr) {

nodeStack.push(current);

current = current->L\_Node;

}

current = nodeStack.top();

// Если правый узел существует и еще не был посещен, идем в него

if (current->R\_Node != nullptr && current->R\_Node != lastVisited) {

current = current->R\_Node;

} else {

nodeStack.pop();

delete current;

lastVisited = current;

current = nullptr;

}

}

root = nullptr;

}

};

int main() {

setlocale(LC\_ALL, "Russian");

Splay\_Tree trains;

ifstream input\_file("input\_file.txt");

if (!input\_file.is\_open()) {

cout << "Не удалось открыть файл!" << endl;

return 1;

}

int n;

string s;

int t;

string line;

while (getline(input\_file, line)) {

// cout << "Читаем строку: " << line << "\n";

istringstream liner(line);

char comma;

if (liner >> n >> comma >> s >> t) {

s.pop\_back();

Train trtrtr(n, s, t);

trains.Add\_Knot(trtrtr);

// cout << "Добавлен поезд: " << trtrtr.get\_number() << endl;

} else {

cout << "Ошибка разбора строки: " << line << "\n";

}

}

// Проверка Add\_Knot

Train test\_train1(111, "SAMARA", 13);

trains.Add\_Knot(test\_train1);

// Проверка Print\_All

trains.Print\_All();

cout << "\n";

// Проверка Pop\_Knot

trains.Pop\_Knot(130);

// trains.Find\_Train(130);

cout << "-----------\n";

trains.Print\_All();

cout << "\n";

// Проверка Find\_Train

Node\* ft = trains.Find\_Train(111);

if (ft != nullptr)

cout << "IT IS EXISTS\n"

<< ft->express.get\_number() << " "

<< ft->express.get\_next\_station() << " " << ft->express.get\_departure\_time() << "\n";

else

cout << "IT IS NOT EXISTS\n";

// Проверка поиска по станции

Node\* st = trains.Find\_By\_Station("Kemerovo");

if (st != nullptr) {

cout << st->express.get\_number() << "\n";

}

// Проверка очистки всего дерева

trains.Clear\_Tree();

trains.Print\_All();

}