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
Факультет механико-математический
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

В настоящее время с целью организации и систематизации данных разрабатываются разные алгоритмы и структуры данных, которые позволяют упростить процесс поиска 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 1983 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 приложения, использующего Splay-дерево для поиска информации о поездах на железнодорожном вокзале.

1 Описание и анализ структур данных

1.1 Термины и определения

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

Дерево — структура данных, элементы которой, называемые вершинами (узлами), связаны отношениями подчиненности, когда одному элементу может быть подчинено несколько, но при этом сам он может быть подчинен только одному [1].

Бинарное дерево поиска — 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 родительских узлов).

Родительский узел — узел, которому подчиняются дочерние узлы.

Дочерний узел — узел, который подчиняется одному родительскому узлу.

Листовой узел — узел, не имеющий дочерних элементов.

1.2 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 [2]. 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 сложность в большинстве случаев составляет $O(\log n)$.

Основным действием, которое производится над структурой, является операция Splay. Эта операция перемещает вершину с ключом x в корень дерева путем различных поворотов [3].

1.3 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:

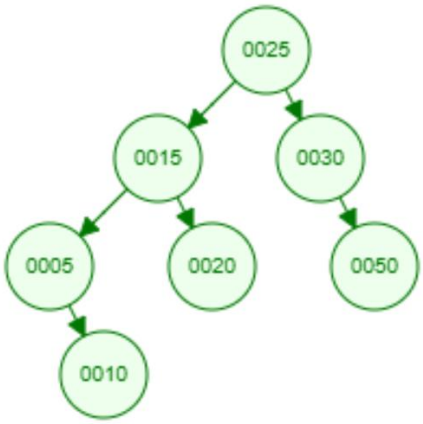

Рисунок 1 – Splay-дерево

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

1.4 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 [4]. Как правило, время доступа к узлу имеет асимптотическую сложность $O(\log n)$, но при обращении к ранее использующемуся узлу асимптотическая сложность принимает константное значение $O(1)$. 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 $O(n)$.

2 Обзор алгоритмов работы со Splay-деревом

2.1 Алгоритм добавления элемента в Splay-дерево

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

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

2.2 Алгоритм поиска элемента в Splay-дереве

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

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.

2.3 Алгоритм удаления элемента из Splay-дерева

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

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

2.4 Алгоритм поднятия узла в корень в Splay-дереве

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

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:

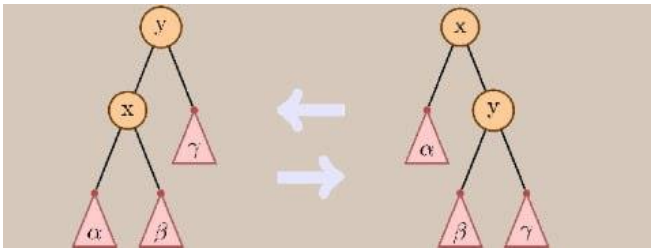

Рисунок 2 – Малый правый поворот в Splay-дереве

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

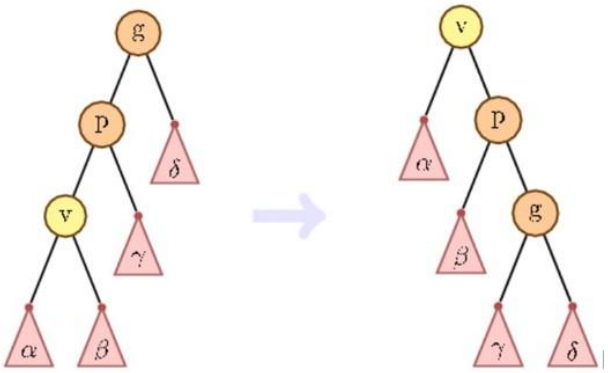

Рисунок 3 – Большой правый поворот в Splay-дереве

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

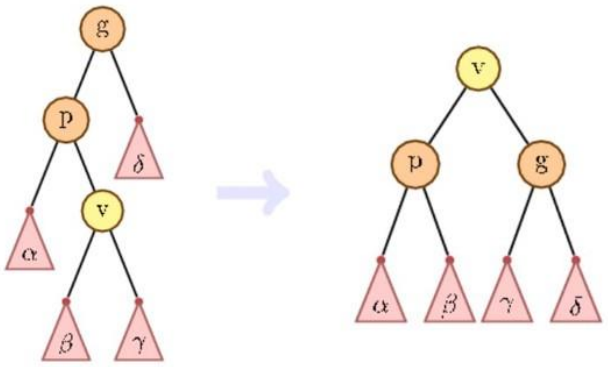

Рисунок 4 – Левый-правый поворот в Splay-дереве

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

Операции добавления, поиска и удаления имеют одинаковую асимптотическую сложность. В лучшем случае это $O(1)$, в среднем — $O(\log n)$, в худшем — $O(n)$.

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

3.1 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 Splay-дерева;

– добавление и удаление информации о поездах;

– поиск информации о поезде по его номеру;

– вывод всего дерева.

3.2 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 $O(1)$.

Вывод информации о всех поездах осуществляется полным перебором дерева. Таким образом, асимптотическая сложность этого действия $O(n)$.

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 $O(\log n)$, в лучшем случае — $O(1)$, в худшем случае — $O(n)$.

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

3.3 Программная реализация алгоритма

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

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

Язык C++ — 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.

3.3.1 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”.

Метод добавления (Add_Node) принимает на вход один параметр — объект типа Node. Он в свою очередь содержит указатели на родительский узел (Parent), указатель на левый дочерний узел (L_Node) и указатель на

правый дочерний узел (R_Node), информацию о поезде (Train_express), которая включает номер поезда (number), следующую станцию (next_station) и время отправления (departure_time).

Метод поиска (Find_Train) принимает на вход один параметр — номер поезда, который является ключом искомого узла. Происходит стандартный поиск бинарного дерева поиска. Поиск начинается с корня дерева. Ключ искомого узла сравнивается с текущим узлом. Если ключ меньше, то осуществляется переход к левому поддереву текущего узла. Если ключ больше, то осуществляется переход к правому поддереву текущего узла. Если ключи равны, то текущий узел и есть искомый. Происходит поднятие найденного ключа в корень. Найденный узел возвращается.

Метод удаления (Pop_Node) принимает на вход один параметр — номер поезда, который является ключом удаляемого узла. Происходит стандартный поиск бинарного дерева поиска. Поиск начинается с корня дерева. Ключ искомого узла сравнивается с текущим узлом. Если ключ меньше, то осуществляется переход к левому поддереву текущего узла. Если ключ больше, то осуществляется переход к правому поддереву текущего узла. Если ключи равны, то текущий узел и есть искомый. Происходит поднятие найденного ключа в корень и его удаление.

Существует три возможных случая удаления.

Первый случай — удаляемый узел без дочерних узлов. В этом случае освобождается память этого узла и указатель на корень (root) становится равным nullptr.

Второй случай — удаляемый узел содержит один дочерний узел. Дочерний узел удаляемого узла становится корнем, указатель на родителя дочернего узла (Parent) становится равным nullptr, освобождается память удаляемого узла.

Третий случай — удаляемый узел содержит два дочерних узла. Находим самый правый (максимальный) узел в левом поддереве. Если этот узел непосредственный левый дочерний узел, то правый указатель родителя

максимального узла становится равным левому указателю максимального узла. Указатель родителя левого узла максимального узла становится равным указателю родителя максимального узла. Заменяем левый и правый указатели максимального узла на левый и правый указатели удаляемого узла. Максимальный узел в левом поддереве становится корнем. Происходит очищение удаляемого узла. Иначе правый указатель дочернего узла максимального узла становится равным указателю правого узла удаляемого узла. Указатель родителя правого узла удаляемого узла становится равным указателю максимального узла левого поддерева. Максимальный узел становится корнем. Происходит очищение удаляемого узла.

В методе поднятия узла в корень (Move_Node_to_Root) 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).

Операции добавления, поиска и удаления имеют одинаковую асимптотическую сложность. В лучшем случае это $O(1)$, в среднем — $O(\log n)$, в худшем — $O(n)$.

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

3.3.2 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.

```
До добавления
130 Barnaul 16
703 Tula 18
1155 SaintPetersburg 23
2106 Smolensk 19
2148 Kemerovo 13
2252 Kaluga 17
2536 Surgut 21
3793 Tyumen 10
4262 Tyumen 16
5266 Cheboksary 19

После добавления
111 SAMARA 13
130 Barnaul 16
703 Tula 18
1155 SaintPetersburg 23
2106 Smolensk 19
2148 Kemerovo 13
2252 Kaluga 17
2536 Surgut 21
3793 Tyumen 10
4262 Tyumen 16
5266 Cheboksary 19
```

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.

```
Поиск
130 Barnaul 16
703 Tula 18
1155 SaintPetersburg 23
2106 Smolensk 19
2148 Kemerovo 13
2252 Kaluga 17
2536 Surgut 21
3793 Tyumen 10
4262 Tyumen 16
5266 Cheboksary 19

IT IS EXISTS
703 Tula 18
```

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.

```
До удаления
130 Barnaul 16
703 Tula 18
1155 SaintPetersburg 23
2106 Smolensk 19
2148 Kemerovo 13
2252 Kaluga 17
2536 Surgut 21
3793 Tyumen 10
4262 Tyumen 16
5266 Cheboksary 19

После удаления
130 Barnaul 16
1155 SaintPetersburg 23
2106 Smolensk 19
2148 Kemerovo 13
2252 Kaluga 17
2536 Surgut 21
3793 Tyumen 10
4262 Tyumen 16
5266 Cheboksary 19
```

Рисунок 7 – Результат проверки операции удаления

Анализируя результаты, можно сделать вывод, что приложение работает корректно и Splay-дерево эффективно при поиске информации о поездах на железнодорожном вокзале.

Код тестируемой программы представлен в приложении Д.

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;

3) Проведен анализ алгоритма поиска элемента в Splay-дерева;

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

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

6) Спроектировано, разработано и протестировано приложение, использующее Splay-дерево для поиска информации о поездах 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-деревья [Электронный ресурс]: Коллективный блог программистов «Хабр». – URL: https://habr.com/ru/company/JetBrains-education/blog/210296/ (дата обращения: 10.05.2025).

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

4. Splay-дерево. Поиск. [Электронный ресурс]: Коллективный блог программистов «Хабр». – URL: https://habr.com/ru/companies/otus/articles/535316/

ПРИЛОЖЕНИЕ А — программная реализация алгоритма добавления элемента в Splay-дерево

```
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()) {
                    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);
    }
```

ПРИЛОЖЕНИЕ Б — программная реализация алгоритма поиска в Splay-дереве

```
// 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;
}
```

ПРИЛОЖЕНИЕ В — программная реализация алгоритма удаления элемента из Splay-дерева

```
// Удаление 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;
    }
}
```

ПРИЛОЖЕНИЕ Г — программная реализация алгоритма поднятия узла в корень в Splay-дереве

```
// 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;
            }
        }
    }
}
```

ПРИЛОЖЕНИЕ Д — листинг кода

```
#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();

}
```