МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра «Прикладная математика и информатика»
(наименование)

01.03.02 Прикладная математика и информатика
(код и наименование направления подготовки, специальности)

Компьютерные технологии и математическое моделирование
(направленность (профиль) / специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему Сравнительный анализ сбалансированных деревьев поиска

Студент Д.О. Панин (И.О. Фамилия) _____ (личная подпись)

Руководитель канд. пед. наук, доцент, О. М. Гущина
(ученая степень, звание, И.О. Фамилия)

Консультант М. В. Дайнеко
(ученая степень, звание, И.О. Фамилия)

Тольятти 2021

Аннотация

Тема бакалаврской работы: «Сравнительный анализ сбалансированных деревьев поиска». Актуальность этой работы в том, что при решении различных практических задач, появляется потребность в поддерживании множества ключей, с возможностью быстрого поиска, добавления или удаления ключа. Одним из примеров таких задач является задача поиска перевода слова в словаре. Выбор наиболее оптимальной структуры данных для решения подобных задач имеет большое значение.

Объект исследования: процесс поиска и модификации множества ключей.

Предмет исследования: декартово дерево, рандомизированное бинарное дерево поиска, splay-дерево, АВЛ-дерево, B-дерево.

Цель работы: определение сильных и слабых сторон структур для их оптимального использования при решении практических задач.

Во введении рассказывается о задаче поиска во множестве и его модификации, структурах, использующихся при решении этой задачи и практических задачах, для решения которых эти структуры используются.

В первой главе работы рассматривается структура декартова дерева, рандомизированного бинарного дерева поиска, splay-дерева, АВЛ-дерева, B-дерева, алгоритмы поиска и модификации этих структур, приведена оценка сложности данных алгоритмов.

Во второй главе проведено тестирование рассматриваемых в работе структур на различных задачах, на основе результатов тестирования совершен сравнительный анализ.

Бакалаврская работа выполнена на 47 страницах, состоит из введения, двух глав, заключения, содержит 31 рисунок и 1 таблицу.

Abstract

The topic of the present graduation work is *Comparative analysis of the balanced search trees*. The relevance of this work is that, when solving various practical problems, there is a need to maintain a variety of keys, with the ability to quickly find, add or delete a key. One of the examples of these tasks is to find a translation of a word in a dictionary. The choice of the most optimal data structure for solving such problems is of great importance.

The graduation work consists of an introduction, two chapters, a conclusion, 31 figures and 1 table.

The object of the research is the process of searching and modifying a number of keys.

The subject of the research is the Cartesian tree, randomized binary search tree, splay-tree, AVL-tree, B-tree.

The purpose of the investigation is to determine the strengths and weaknesses of the structures for their optimal use when solving practical problems.

The introduction dwells on the problem of searching among a number of the keys and its modifications, as well as the structures used to solve this problem and the practical problems these structures are used to find solutions to.

The first chapter of the work considers the structure of a Cartesian tree, a randomized binary search tree, a splay-tree, an AVL-tree, a B-tree, as well as reveals the algorithms for searching and modifying these structures. This chapter also assesses the complexity of these algorithms.

In the second chapter, the structures reviewed in the work are tested on various tasks. This chapter also carries out a comparative analysis based on the test results obtained.

Оглавление

Введение

Одной из часто встречающихся задач в программировании является построение множества, элементами которого являются значения из очень большого диапазона, поиска в нем и его модификации. Так как диапазон велик, невозможно для каждого из возможных значений хранить в памяти информацию о том, находится ли оно во множестве.

Основной операцией во множестве является поиск значения по ключу. Можно хранить все элементы, находящиеся во множестве в списке или массиве, и просто перебирать их, пока не найдется тот, который нужен, либо будет достигнут конец списка, однако такой способ может оказаться слишком медленным, если размер множества большой, ведь он тратит O(n) на поиск (n – размер множества). Возможным решением для ускорения может стать двоичный поиск, для его применения необходимо, чтобы элементы множества находились в памяти в отсортированном порядке, тогда на каждой итерации поиска отрезок, на котором может находиться искомый ключ будет уменьшаться в 2 раза, значит время поиска составит O(logn) [2].

Однако данные во множестве чаще всего не являются статическими, ключи добавляются и удаляются, структура также должна поддерживать эти операции. Чтобы добавить или удалить ключ в последовательный отсортированный блок памяти, необходимо сдвинуть все ключи, находящиеся после добавляемого/удаляемого ключа в памяти, на одну позицию, что может занять O(n) времени, сделав модификацию множества тяжелой операцией.

Выбирая между хранением ключей в случайном порядке и хранением в отсортированном, мы, по сути, выбираем между вариантом с быстрой (O(logn)) вставкой/удалением, но медленным (O(n)) поиском и вариантом с медленной (O(n)) вставкой/удалением, но быстрым (O(logn)) поиском.

Для решения задачи быстрой вставки/удаления и поиска ключа были созданы деревья поиска. Такие структуры позволяют достигнуть оценки O(logn) для всех необходимых операций и занимают O(n) памяти. Было

изобретено множество различных сбалансированных деревьев поиска. В 1989 году Сидель Раймундом (Seidel Raimund) и Цецилия Арагон (Aragon Cecilia) создали декартово дерево, которое также называют дерамидой. Советские ученые Адельсон-Вельский и Ландис в 1962 году изобрели АВЛ-дерево, бинарное дерево, баланс в котором поддерживается за счет того, что разница высот левого и правого поддерева для каждой из вершин не превышает 1. Это свойство АВЛ-дерева делает его высоту примерно равной O(logn). В 1970 году Р. Бейер и Э. МакКрейт создали B-дерево, сильно ветвящееся, сбалансированное дерево, которое может содержать несколько ключей в каждой из вершин, тем самым позволяющее уменьшить количество операций чтения с диска. Благодаря этой особенности B-деревья часто используются при построении индексов в базах данных [20]. Сбалансированные деревья поиска используют для решения следующих задач:

1. Так называемая «словарная проблема». Примером такого множества на практике может являться англо-русский словарь, в котором ключ – это слово на английском, а значение это перевод данного слова на русский, также хороший пример - это телефонная книга, где ключом является ФИО человека а значением – его телефонный номер. В программировании словари обычно называются ассоциативными массивами, и они уже реализованы в стандартных библиотеках (map в C++, TreeMap в Java).

2. Структура данных rope, представляющая из себя сбалансированное двоичное дерево. Эта структура для хранения больших строк, которая позволяет совершать многие операции вроде конкатенации, взятия подстроки на строках длины n с асимптотикой порядка O(logn). Эффективность достигается за счет того, что строка хранится в памяти не как массив из символов, а разбита на небольшие части, которые хранятся в узлах дерева rope.

3. Индексы в базах данных основаны на деревьях поиска, одним из наиболее часто используемых является B-дерево. Так как элементы в деревьях расположены в отсортированном порядке, они позволяют не только искать

ключи с конкретными значениями, но и, например, искать наименьший ключ со значением больше заданного, или вычислять количество ключей из заданного диапазона.

Данная работа посвящена теоретическому и практическому сравнительному анализу сбалансированных деревьев поиска.

Объект исследования – процесс модификации и поиска во множестве.

Предметом исследования являются декартово дерево, рандомизированное дерево поиска, АВЛ-дерево, splay-дерево, B-дерево.

Целью бакалаврской работы является сравнение эффективности рассматриваемых деревьев поиска на различных задачах, выявление сильных и слабых сторон структур при использовании на практике.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Рассмотреть задачи, решаемые с помощью деревьев поиска.
2. Рассмотреть различные деревья поиска.
3. Провести теоретический сравнительный анализ рассматриваемых структур на предмет эффективности.
4. Реализовать рассматриваемые деревья поиска на языке программирования C++.
5. Провести генерирование данных для тестирования деревьев на предмет производительности.
6. Провести тестирование на сгенерированных данных.
7. Осуществить сравнительный анализ деревьев поиска на сгенерированных данных.

Бакалаврская работа состоит из введения, двух глав и заключения.

В первой главе дано описание различных сбалансированных деревьев поиска, приведена асимптотическая оценка их работы.

Вторая глава посвящена тестированию рассмотренных структур на различных задачах и сравнительному анализу на основании результатов тестирования.

Глава 1 Деревья поиска

1.1 Декартово дерево

Декартово дерево – это структура, которая представляет из себя объединение бинарного дерева поиска и бинарной кучи, поэтому его также называют treap (tree + heap) или дерамида (дерево + пирамида). Дерамиды впервые были изобретены Сиделем Раймундом (Seidel Raimund) и Цецилия Арагоном (Aragon Cecilia) в 1989 г [18].

Бинарное дерево поиска можно описать следующим образом: это дерево, в котором каждая вершина имеет не более 2 потомков, в каждой из вершин записано по ключу (обычно это число), для каждой вершины ключ в ней больше любого из ключей в вершинах ее левого поддерева, но меньше любого из ключей во всех вершинах ее правого поддерева. Для одного набора ключей можно построить множество разных бинарных деревьев поиска, некоторые из вариантов для набора ключей (1, 2, 3, 4) изображены на рисунке 1.

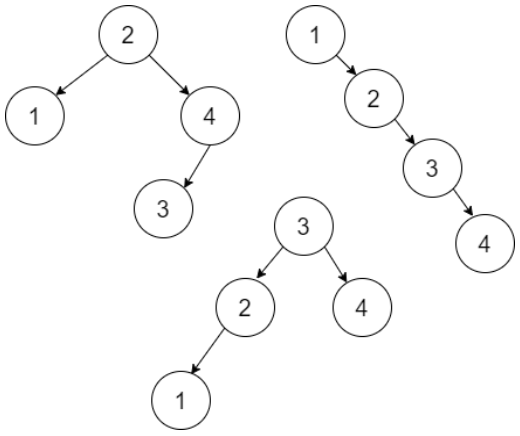

Рисунок 1 – Варианты построения бинарного дерева поиска для набора ключей (1, 2, 3, 4)

Бинарная куча или пирамида, также как и бинарное дерево, является деревом, в котором у каждой из вершин число потомков не превосходит 2. В

любой вершине кучи записан ключ, для ключей выполняется следующее правило: значение ключа в вершине больше значений ключей в ее потомках. Для одного набора ключей можно построить множество разных бинарных деревьев поиска, некоторые из вариантов для набора ключей (1, 2, 3, 4) изображены на рисунке 2.

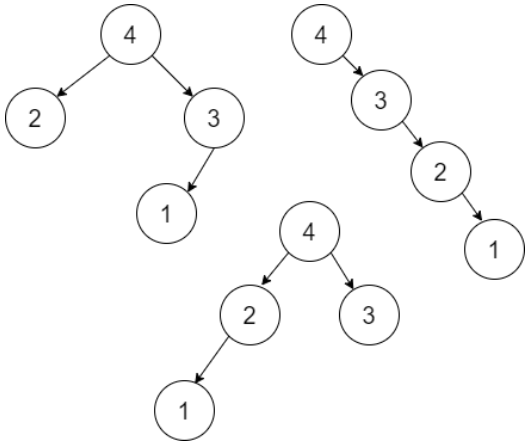

Рисунок 2 – Варианты построения бинарной кучи для набора ключей (1, 2, 3, 4)

Декартово дерево представляет из себя бинарное дерево (у каждой вершины не более 2 потомков), в вершинах которого содержатся пары (x, y). Оно является двоичным деревом поиска по x (для всех вершин в левом поддереве x меньше, а для всех в правом больше) и бинарной пирамидой по y (приоритет в предке всегда больше приоритета в вершине). Значение x обычно называют ключом, а y – приоритетом.

Дерево называют декартовым, потому что его вершины можно представить в виде точек на прямоугольной декартовой системе координат на плоскости. Ребра между вершинами дерева можно представить в виде отрезков, соединяющих точки, соответствующие этим вершинам. Благодаря ограничениям на значения ключей и приоритетов, ребра дерева на плоскости никогда не будут пересекаться на плоскости, а касаться только в вершинах

дерева. На рисунке 3 представлен пример корректного декартова дерева (слева) и его отображение на плоскости (справа):

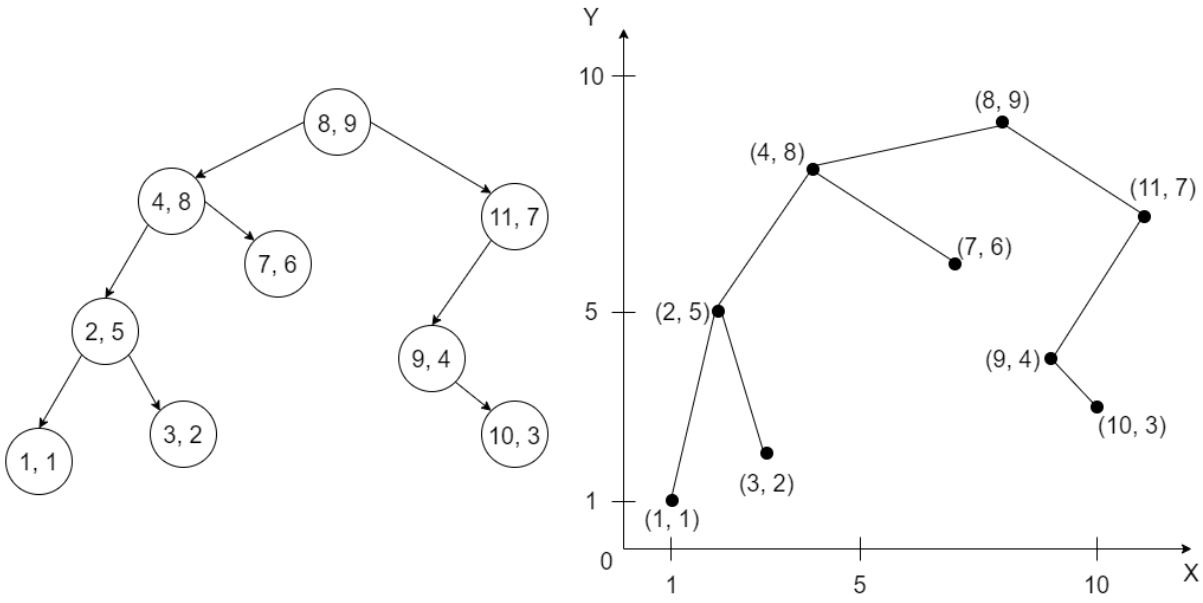

Рисунок 3 – Декартово дерево и его отображение на плоскости

Для любого набора пар (ключ, приоритет) можно построить лишь одно корректное декартово дерево. Алгоритм построения следующий: корнем дерева становится вершина с наибольшим приоритетом, все вершины с приоритетами меньше чем в корне будут в левом поддереве, остальные в правом. Таким же образом рекурсивно строятся левое и правое поддеревья, их корни соединяются с корнем дерева.

Главным свойством декартова дерева, позволяющим совершать запросы и модификации этой структуры за O(logn) является то, что, если приоритеты являются случайными, какими бы не были значения ключей, высота дерева не будет превышать $4\log_2 n$ с вероятностью близкой к 100% [2].

Поиск ключа в декартовом дереве осуществляется так же, как и в бинарном дереве, спуском от корневой вершины, если искомый ключ меньше ключа в вершине, спускаемся в левого потомка, если больше – в правого. Если дошли до листа, а по пути так и не встретился нужный ключ, значит его в

дереве нет. Длина пути, по которому осуществляется проход, при поиске не превышает высоту дерева, значит асимптотика поиска O(logn).

Для вставки и удаления узлов в дереве нужно уметь разрезать его по ключу (операция split) и соединять 2 дерева в одно (операция merge). Рассмотрим эти операции подробно.

Операция split разделяет декартово дерево на 2 корректных декартовых дерева по ключу x таким образом, что в левом дереве все ключи меньше x, а в правом – больше. Сначала определим, где окажется корень дерева. Если ключ в нем меньше x, то корень вместе со всем своим левым поддеревом (так как в нем в этом случае все ключи также меньше x) окажется в левой части результата. Следующим шагом нужно определить какая часть правого поддерева корня останется на месте, а какая часть пойдет в правую часть результата. Это задача по сути идентична изначальной, нужно просто разрезать правое поддерево по x, левое из получившихся деревьев подвесить справа к корню, который уже находится в левом дереве, а правая часть станет правым деревом. Случай, когда ключ в корне больше x решается аналогично. Задача является рекурсивной, количество итераций рекурсии не превысит высоту дерева, так как на каждом шаге высота дерева, которое нужно разрезать уменьшается как минимум на 1, следовательно асимптотика split составляет O(logn) [8].

Формально, split по ключу x работает следующим образом:

1. Если ключ в вершине больше x, отделить от нее левое поддерево и выполнить с ним split по x, корень правого из получившихся деревьев сделать левым сыном вершины.
2. Если ключ в вершине меньше x, отделить от нее правое поддерево и выполнить с ним split по x, корень левого из получившихся деревьев сделать правым сыном вершины.

На рисунке 4 изображено действие split по ключу 6:

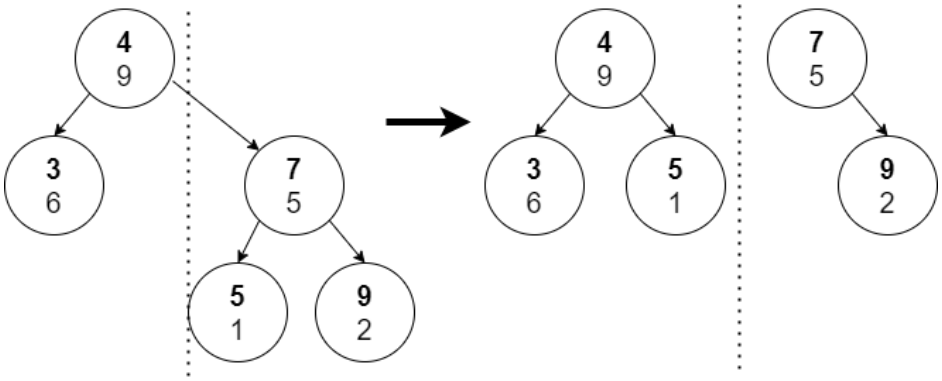

Рисунок 4 – Split декартова дерева по ключу 6

Операция merge строит декартово дерево по 2 корректным декартовым деревьям при условии, что любой ключ в левом дереве больше любого в правом. Очевидно, что корнем получившегося в результате слияния дерева должна стать вершина с наибольшим приоритетом из всех вершин, что есть в левом и правом деревьях. Если приоритет в корне левого дерева больше приоритета в корне правого, Корень левого дерева станет корнем итогового дерева, его левое поддерево останется на месте, а правое поддерево левого дерева и правое дерево нужно будет слить в одно дерево и подвесить справа от корня, снова рекурсия. Случай, когда приоритет в корне правого дерева больше, решается аналогично. На каждой итерации получившееся после слияния дерево подвешивается на уровень ниже к итоговому дереву, значит асимптотика merge – O(logn) [9].

Алгоритм операции merge выглядит следующим образом:

1. Если приоритет в корне левого дерева больше приоритета в корне правого, следует отделить правое поддерево левого дерева от его корня, выполнить merge для этого поддерева и правого дерева, а корень получившегося в результате дерева сделать правым потомком корня левого дерева.

2. Если приоритет в корне правого дерева больше приоритета в корне левого, следует отделить левое поддерево правого дерева от его корня,

выполнить merge для этого поддерева и левого дерева, а корень получившегося в результате дерева сделать левым потомком корня правого дерева.

На рисунке 5 изображены этапы действия merge:

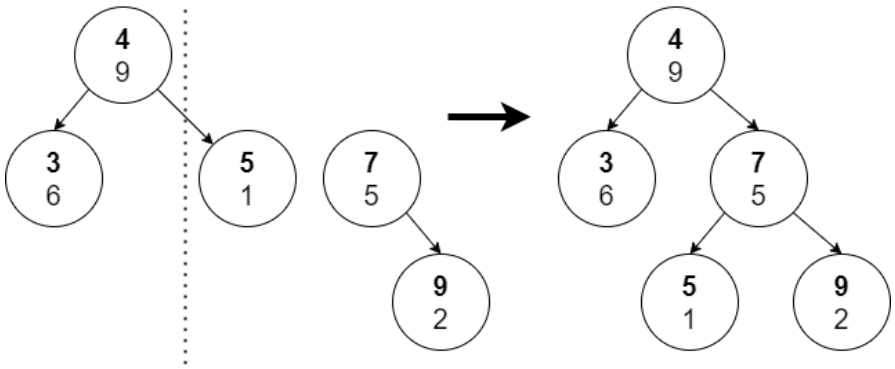

Рисунок 5 – Действие merge

Добавление ключа в декартово дерево осуществляется так: сначала совершается split по этому ключу, затем merge левого дерева и дерева, состоящего из одной вершины, которой и является вставляемый элемент, и в конце merge получившегося дерева с правым. Так как split и merge работают за O(logn), то и асимптотика вставки такая же.

Чтобы удалить ключ, нужно совершить split по нему таким образом, чтобы все ключи меньше или равные данному оказались в левом дереве, остальные – в правом. Далее следует совершить спуск от корня в левом дереве, каждый раз переходя в правого потомка, пока он есть, таким образом в конце оказавшись в вершине с наибольшим ключом в этом дереве, которую и следует удалить. Если это корень, удалить его и заменить левое дерево на его левое поддерево, иначе удалить эту вершину, подвесив ее левое поддерево справа к ее предку. После этого нужно соединить деревья обратно с помощью merge. Так как спуск в дереве, split и merge работают за O(logn), то и асимптотика удаления такая же.

Таким образом, высота декартова дерева равна O(logn) за счет того, что приоритеты в вершинах являются рандомными числами, что позволяет быстро

осуществлять поиск путем спуска из корня, совершать split/merge при добавлении и удалении ключа.

1.2 Рандомизированное бинарное дерево поиска

Рандомизированное бинарное дерево поиска – это бинарное дерево поиска, удовлетворяющее следующим требованиям:

1. Если в дереве n вершин, каждая из них может быть его корнем с вероятностью 1/n.
2. Любое его поддерево также является рандомизированным бинарным деревом поиска.

Эти свойства делают математическое ожидание высоты рандомизированного дерева равным O(logn) [15].

Поиск в такой структуре осуществляется так же, как и в стандартном бинарном дереве поиска, начиная с корня, необходимо спускаться в левого потомка, если искомый ключ меньше ключа в вершине, иначе – в правого, и так, пока не найдется этот ключ. Работает за O(logn).

Вставка в рандомизированное дерево осуществляется двумя способами, рассмотрим их подробнее. В первом варианте вставка ключа происходит так же, как и его поиск, только в случае, когда потомок, в которого следует переходить, отсутствует, на его место и вставляется этот ключ. Очевидно, что асимптотика этой операции такая же, как и у поиска, O(logn). На рисунке 6 изображена вставка ключа 5 в дерево, путь, по которому следует пройти, отмечен пунктиром.

Вторым вариантом вставки является вставка в корень. Сначала вершина добавляется в дерево в качестве листа, как это делается в первом варианте вставки, а после этого меняется местами со своим предком, пока не окажется корнем. Просто поменять местами ключи в вершине и ее предке нельзя, так как это может привести к нарушению главного свойства бинарного дерева –

все ключи в левом поддереве меньше ключа в каждой из вершин, а в правом больше [17].

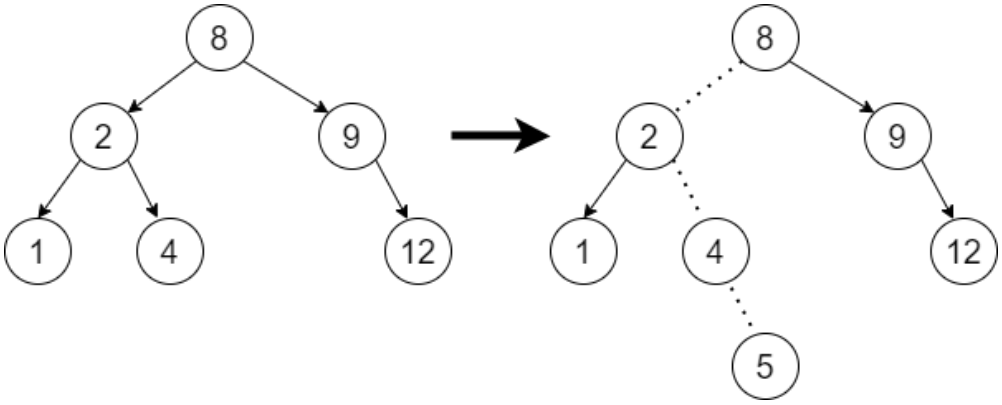

Рисунок 6 – Вставка ключа в рандомизированное бинарное дерево поиска

Если вершина является правым потомком для своего корня, необходимо совершить трансформацию, которую обычно называют левый поворот. На рисунке 7 изображены изменения, произошедшие после поворота. V – вершина, для поднятия которой совершается поворот, B и C – ее левое и правое поддеревья, соответственно. P является предком V, A – левое поддерево предка.

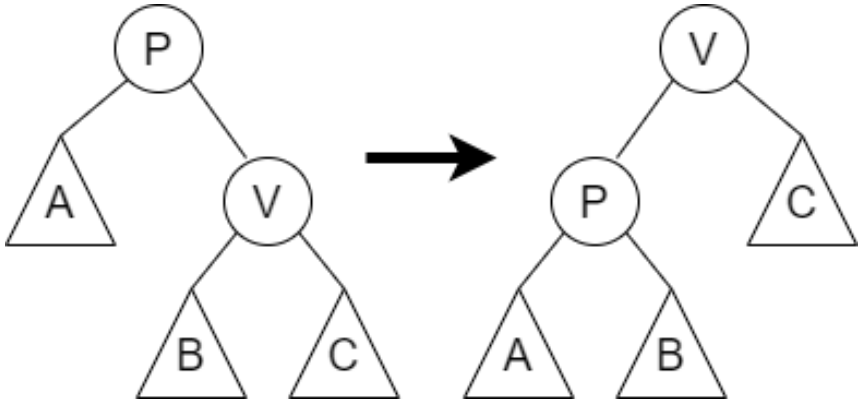

Рисунок 7 – Левый поворот

В случае, когда вершина является левым потомком, выполняется правый поворот. На рисунке 8 изображены изменения, произошедшие после поворота. V – вершина, для поднятия которой совершается поворот, A и B – ее левое и правое поддеревья, соответственно. P является предком V, C – правое поддерево предка.

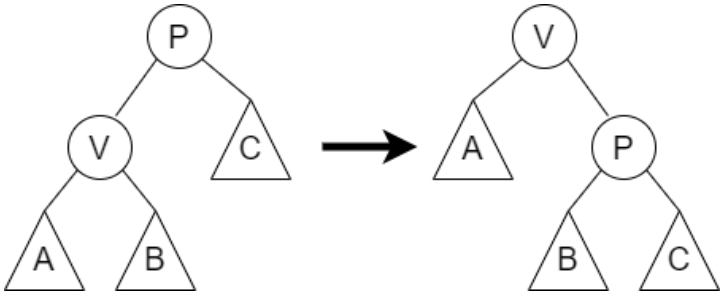

Рисунок 8 – Правый поворот

На рисунке 9 приведен пример вставки вершины с ключом 5 в рандомизированное дерево с последующим подъемом этой вершины в корень с помощью последовательности левых и правых поворотов:

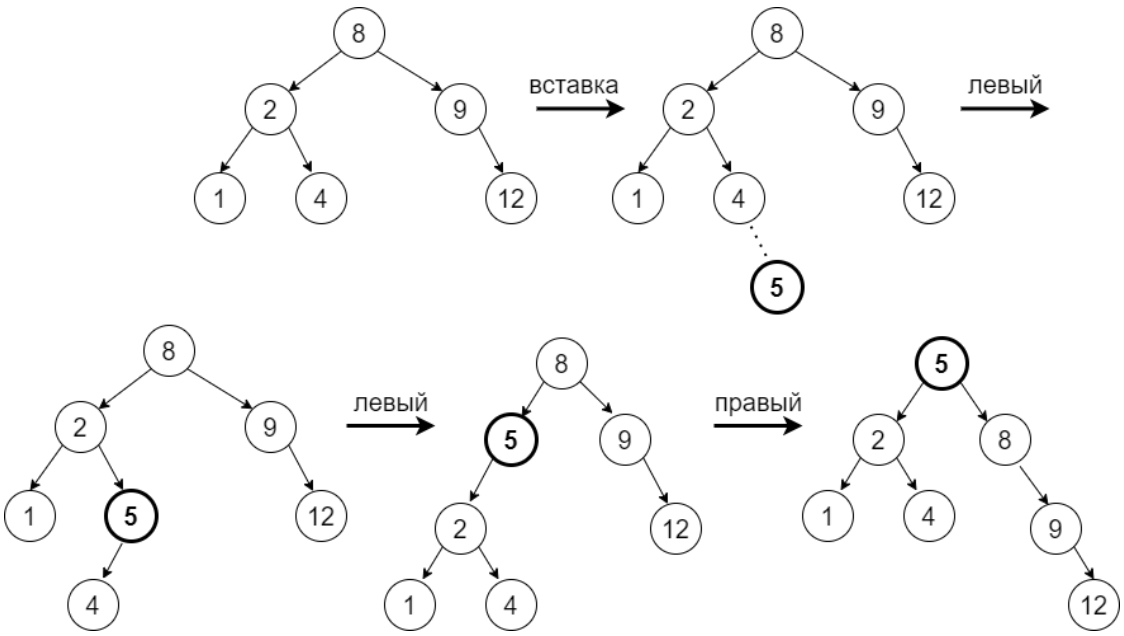

Рисунок 9 – Вставка вершины с ключом 5 в корень

Вставка вершины в качестве листа работает за O(logn), при подъеме ее в корень совершается O(logn) правых/левых поворотов с асимптотикой O(1), следовательно асимптотика вставки в корень составляет O(logn) [11].

Для того, чтобы после вставки вершины дерево оставалось рандомизированным, нужно чтобы вероятность того, что вершина является корнем дерева, для любой из вершин была 1/n, где n – кол-во вершин в дереве. Значит вероятность того, что вставляемая вершина станет корнем должна быть 1/(n + 1), где n – кол-во вершин в дереве перед вставкой. Для этого можно сгенерировать случайное целое число в диапазоне [0, n], если оно равно 0 – выполнить вставку в корень, иначе обычную.

Чтобы удалить ключ из дерева, нужно найти вершину с этим ключом, совершив спуск из корня. Если у найденной вершины нет потомков, можно просто удалить ее. Если у вершины есть только один потомок, следует удалить ее, и подвесить этого потомка к предку удаленной вершины на ее место, либо сделать корнем, если был удален корень. В случае, когда у вершины, которую нужно будет удалить, есть оба потомка, следует объединить поддеревья с корнями в этих потомках в одно дерево, а корень получившегося дерева займет место удаленной вершины.

Объединение деревьев происходит следующим образом: корнем нового дерева становится корень одного из объединяемых деревьев, нужно определить, какой именно. Пусть размеры кол-во вершин в левом и правом поддеревьях равно n_1 и n_2 соответственно. Тогда корнем нового дерева становится корень левого с вероятностью $n_1 / (n_1 + n_2)$, а корень правого с вероятностью $n_2 / (n_1 + n_2)$. Чтобы определить, какой же именно из корней выбрать, можно сгенерировать случайное число в диапазоне $[0, n_1 + n_2 – 1]$, если оно окажется меньше n_1, сделать корнем корень левого дерева, иначе – правого.

Определившись с корнем нового дерева, нужно выполнить следующие действия:

1. Если корнем становится корень левого дерева, левым поддеревом итогового дерева нужно сделать левое поддерево левого дерева, а правым поддеревом – результат объединения правого поддерева левого дерева и правого дерева.

2. Если корнем становится корень правого дерева, правым поддеревом итогового дерева нужно сделать правое поддерево правого дерева, а левым поддеревом – результат объединения левого поддерева правого дерева и левого дерева.

На рисунке 10 изображены этапы объединения двух рандомизированных бинарных деревьев:

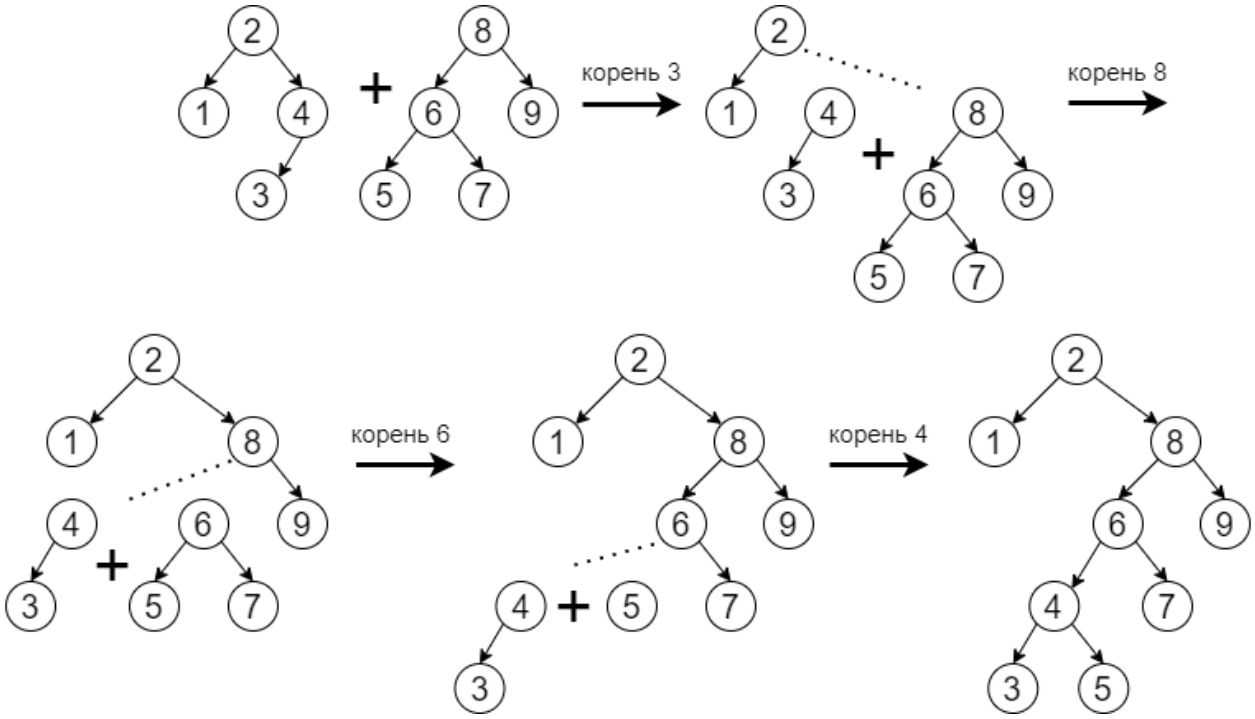

Рисунок 10 – Объединение двух рандомизированных бинарных деревьев

На каждой итерации объединения высота одного из деревьев уменьшается как минимум на 1. Так как их изначальные высоты не могут превышать logn, асимптотика объединения O(logn). Удаление вершины

состоит из спуска в нее из корня и объединения ее поддеревьев, следовательно его асимптотика тоже O(logn) [10].

Таким образом, высота рандомизированного дерева поиска равна O(logn) за счет того, что каждая из его n вершин может быть его корнем с вероятностью 1/n, что позволяет быстро осуществлять поиск путем спуска из корня, перемещение вершины в корень при вставке в корень и объединение двух рандомизированных деревьев при удалении ключа.

1.3 Splay-дерево

Расширяющееся (англ. splay) дерево является самобалансирующимся бинарным деревом поиска, которое поддерживает необходимый баланс ветвления, позволяющий достигать выполнения поиска, вставки и удаления ключа за логарифмическое время. Структура была предложена в середине восьмидесятых Робертом Тарьяном и Даниелем Слейтором [19].

Основным действием, производящимся над структурой, является операция splay [16]. Эта операция перемещает вершину с ключом x в корень дерева путем различных поворотов. Рассмотрим подробнее виды поворотов.

Поворот zig: совершается только в случае, когда предком вершины с ключом является корень дерева. На рисунке 11 изображены 2 случая, первый – когда x является левым потомком для своего предка p, второй – когда правым:

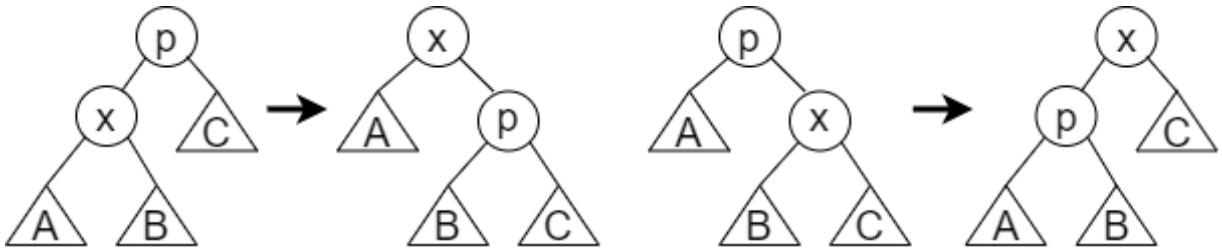

Рисунок 11 – Поворот zig

Поворот zig-zig: пусть предок продвигаемой в корень вершины x называется p, а предок p – p2. Данный вид поворота совершается, когда вершины x и p являются либо обе левыми потомками, либо обе правыми. Zig-zig можно представить как последовательность из 2 zig: сначала zig для p и p2, затем zig для x и p. На рисунке 12 изображены оба случая применения данного поворота:

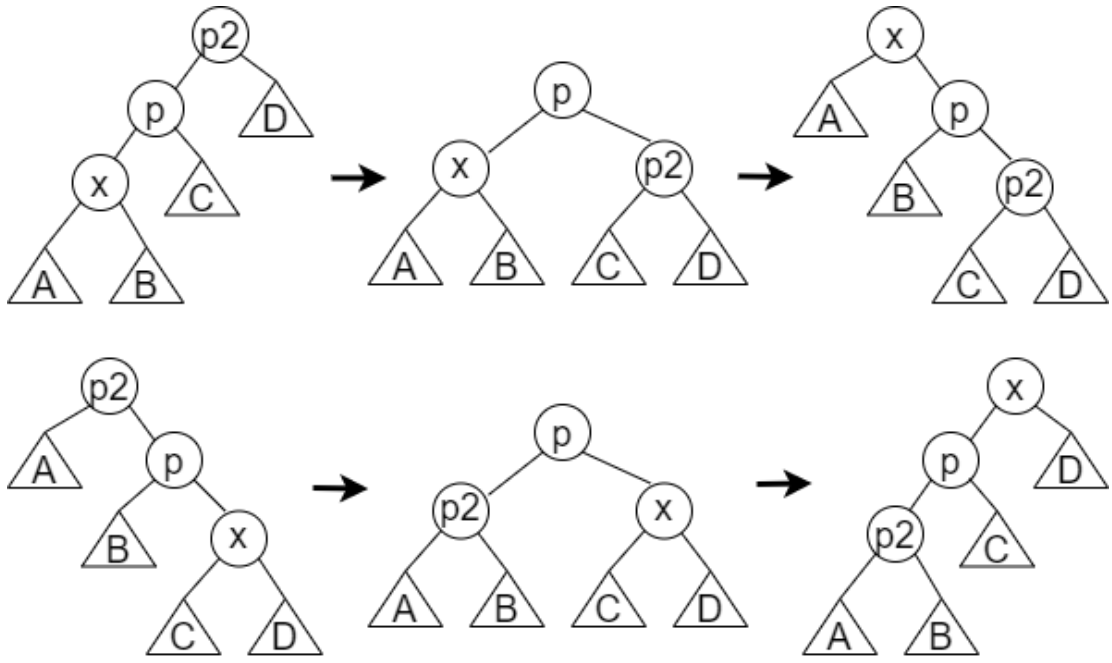

Рисунок 12 – Поворот zig-zig

Поворот zig-zag: пусть предок продвигаемой в корень вершины x называется p, а предок p – p2. Данный вид поворота совершается, когда вершина x является правым потомком для p, а вершина p – левым потомком для p2, либо наоборот, x является правым потомком для p, а сама вершина p – левым потомком p2. Zig-zag можно представить как последовательность из 2 zig: сначала zig для x и p, затем zig для x и p2. На рисунке 13 изображены оба случая применения данного поворота:

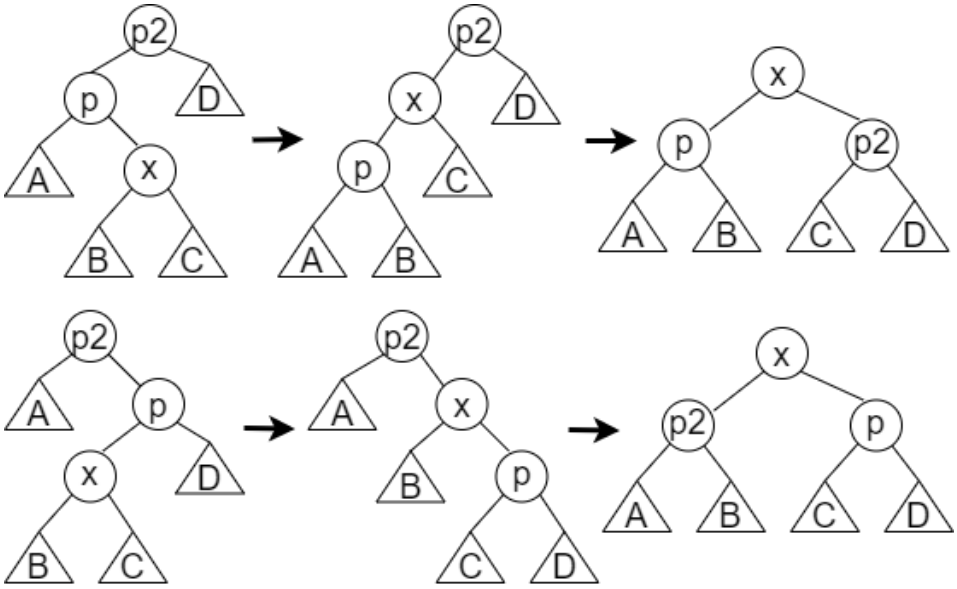

Рисунок 13 – Поворот zig-zag

Поиск вершины в splay-дереве происходит так же, как и в обычном бинарном дереве, путем спуска от корня до искомой вершины. Единственное отличие в том, что если вершина нашлась, она перемещается в корень дерева посредством операции splay.

При вставке, новая вершина становится корнем дерева. Если до этого в нем уже были вершины, нужно совершить попытку найти вершину со вставляемым ключом, в какой-то момент при спуске из корня не окажется потомка для перехода. Необходимо сделать splay для вершины, на которой закончился спуск, переместив ее в корень. Если ключ в этой вершине меньше ключа в добавляемой, необходимо отделить правое поддерево от корня и подвесить эти 2 дерева к новой вершине, если же ключ в корне больше добавляемого ключа, следует отделить левое поддерево от корня, затем подвесить получившиеся в результате деревья к новой вершине [13].

На рисунке 14 приведен пример вставки ключа 4 в splay-дерево. Сначала проводится поиск этого ключа в дереве, который останавливается в вершине 3, так как у нее нет правого сына, в которого нужно переходить при поиске большего ключа. Затем 3 перемещается в корень с помощью splay, правое

поддерево отсекается. В итоге имеются 2 дерева, в одном все ключи меньше 4, в другом – больше, они подвешиваются к новой вершине 4.

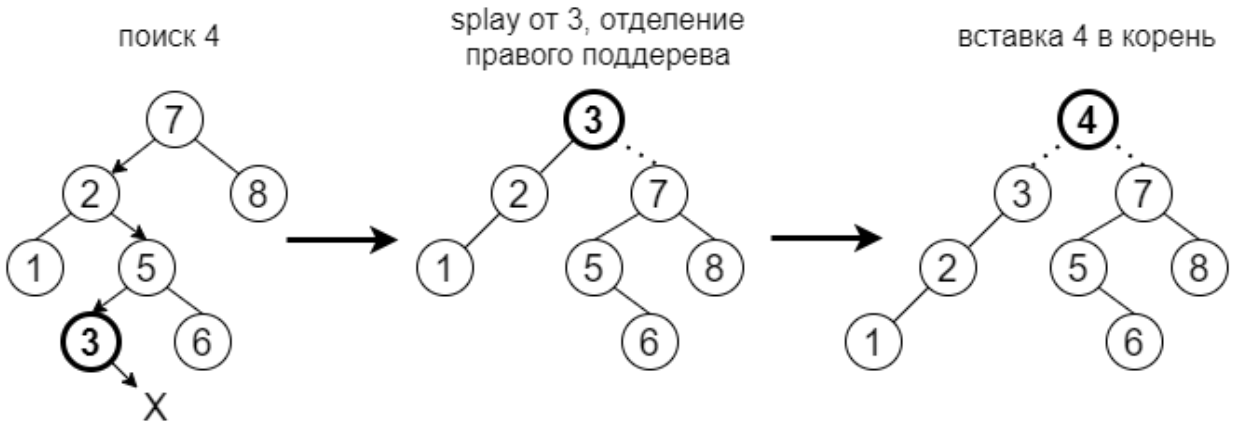

Рисунок 14 – Вставка ключа 4 в splay-дерево

Чтобы удалить вершину из дерева, нужно найти ее и переместить в корень, используя операцию splay. После удаления корня необходимо соединить его левое и правое поддеревья в корректное splay-дерево. Если левого поддерева нет, корнем становится корень правого поддерева, иначе в левом поддереве ищется вершина с наибольшим ключом (для этого достаточно совершить спуск по дереву из корня, каждый раз спускаясь в правого потомка, пока он есть) и перемещается в его корень с помощью splay. После этого у корня левого поддерева не будет правого потомка, поэтому на его место можно будет подвесить правое поддерево, таким образом соединив их [14].

На рисунке 15 представлен пример удаления ключа 5 из splay-дерева. Сначала находится положение ключа в дереве, после чего выполняется splay от него, делая удаляемую вершину корнем. Корень удаляется, в левом поддереве ищется вершина с наибольшим ключом (4), совершается splay от нее, после чего она становится корнем левого поддерева, у которого нет правого потомка, так как в этом поддереве нет вершин с ключами больше. После этого правое поддерево подвешивается к корню левого.

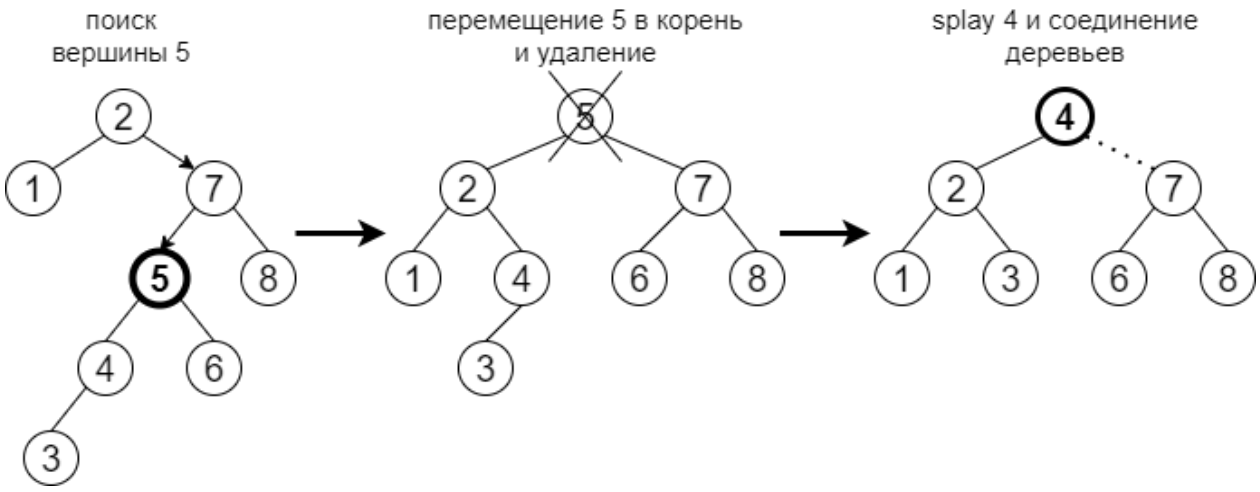

Рисунок 15 – Удаление ключа 5 из splay-дерева

Доказано, что выбор между zig, zig-zig и zig-zag во время splay по описанным выше принципам, позволяет осуществлять поиск вставку и удаление ключей в структуре за O(logn).

1.4 АВЛ-дерево

АВЛ-дерево является сбалансированным двоичным деревом поиска, изобретенным советскими учеными Георгием Максимовичем Адельсон-Вельским и Евгением Михайловичем Ландисом в 1962 году. Аббревиатура АВЛ, как несложно догадаться, возникла из первых букв фамилий создателей [1].

Баланс в этой структуре гарантирует следующее правило: для каждой вершины дерева высоты левого и правого поддеревьев отличаются не более, чем на 1. Доказано, что это достаточно для того, чтобы высота дерева всегда была порядка O(logn) [3].

Поиск ключа в АВЛ-дереве осуществляется так же, как и в обычном двоичном дереве поиска, ведь оно им и является. Работает за O(logn), так как высота дерева не превышает logn.

Разница высот правого и левого поддеревьев для любой вершины АВЛ-дерева не превышает 1 за счет использования операции, называемой балансировкой вершины. Эта операция применяется в случае, когда разность высот левого и правого поддеревьев этой вершины становится равной 2. Балансировка перестраивает поддерево с корнем в вершине так, что разности высот поддеревьев для любой из вершин поддерева не будут превышать 1, следовательно поддерево снова станет АВЛ-деревом [4].

Пусть разность высот левого и правого поддеревьев вершины v равна f(v). Рассмотрим случай, когда f(v) = -2. Для достижения баланса, можно совершить операцию, называемую малым левым вращением. На рисунке 16 изображена трансформация поддерева с корнем в v при малом левом вращении, u – правый потомок v, A – левое поддерево v, B и C – левое и правое поддеревья вершины u соответственно:

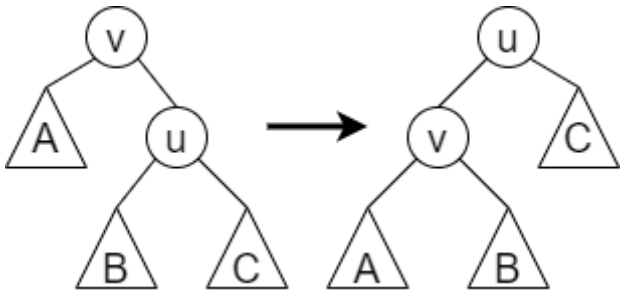

Рисунок 16 – Малое левое вращение в АВЛ-дереве

Приведет ли это действие к балансу, зависит от разности высот поддеревьев B и C. Так как поддерево с корнем в u является сбалансированным, есть 3 варианта значения f(u), рассмотрим их:

1. f(u) = -1, после вращения f(v) = 0, f(u) = 0, высота поддерева уменьшается на 1, баланс достигнут.

2. f(u) = 0, после вращения f(v) = -1, f(u) = 1, высота поддерева не изменилась, баланс достигнут.

3. f(u) = 1, после вращения f(v) = -1, f(u) = 2, высота поддерева не изменилась, баланс не достигнут.

В случае, когда до поворота f(u) = 1, баланса с его помощью достичь не удается, поэтому в такой ситуации применяется большое левое вращение [7]. На рисунке 17 представлены изменения, происходящие с поддеревом с корнем в v в результате большого левого вращения. Вершина u – правый потомок v, вершина g – левый потомок u, A – левое поддерево v, D – правое поддерево u, B и C – левое и правое поддеревья вершины g соответственно.

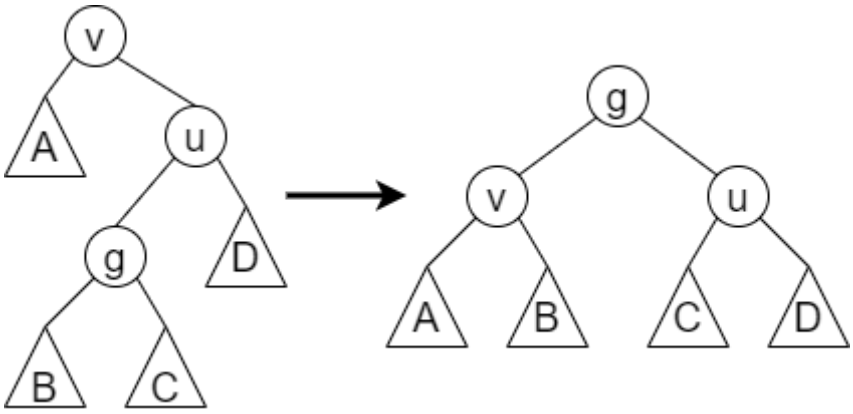

Рисунок 17 - Большое левое вращение в АВЛ-дереве

Значение f(g) (разность высот поддеревьев B и C) перед вращением может быть равно -1, 0 или 1. Проверим, приведет ли большое левое вращение к балансу во всем поддереве в каждом из этих случаев:

1. f(g) = -1, после вращения f(v) = 0, f(u) = -1, f(g) = 0, высота поддерева уменьшается на 1, баланс достигнут.

2. f(g) = 0, после вращения f(v) = 0, f(u) = 0, f(g) = 0, высота поддерева уменьшается на 1, баланс достигнут.

3. f(g) = 1, после вращения f(v) = 1, f(u) = 0, f(g) = 0, высота поддерева уменьшается на 1, баланс достигнут.

Во всех случаях, когда f(u) = 1, большое левое вращение позволяет достичь баланса, следовательно при балансировке вершины v в случае, когда

f(v) = -2, нужно совершать малое левое вращение, если f(u) = -1 или f(u) = 0, и большое левое вращение в случае f(u) = 1. Ситуация, когда f(v) = 2 является полностью симметричной, нужно выполнить малое или большое правое вращение, в зависимости от разности высот левого и правого поддеревьев левого потомка v.

Вставка ключа в АВЛ-дерево осуществляется таким же способом, как и в двоичном дереве, с помощью спуска от корня, каждый раз выбирая переход в левого потомка, если ключ меньше и в правого, если больше. Когда нужного потомка не окажется, добавим на это место вершину. Разница высот левого и правого поддерева для некоторых вершин на пути, который будет пройден при вставке, может стать равной 2 или -2, поэтому нужно пройти по этому пути от добавленной вершины обратно к корню, и каждый раз, когда разность высот поддеревьев для текущей вершины равна 2 или -2, совершать балансировку в этой вершине. Так как после любого из 4 видов вращений высота поддерева, в корне которого совершалась балансировка, либо останется неизменной, либо уменьшается на 1, модуль разности поддеревьев для всех вершин никогда не превысит 2. После того, как будет проведена балансировка на всех нуждающихся в ней вершинах на пути от добавленной вершины до корня, дерево снова станет корректным сбалансированным АВЛ-деревом.

На рисунке 18 изображен пример вставки ключа 3 в АВЛ-дерево. После добавления его в качестве листа дерева, нарушился баланс в вершинах 5 и 7 (разность поддеревьев стала равна 2). При подъеме из 3 в корень, первой на пути встретится вершина 5, в ней будет совершена балансировка путем большого правого вращения, после этого баланс в вершине 7 также восстановится, так как высота ее левого поддерева уменьшится на 1.

Удаление ключа в АВЛ-дереве происходит следующим образом:

1. Если вершина с этим ключом – лист, она удаляется.

2. Если у вершины есть только левый потомок, удалить ее и подвесить ее левое поддерево к ее предку.

3. Если у вершины v есть оба потомка, следует найти вершину u с минимальным ключом в ее правом поддереве и удалить ее, предварительно переместив значение ключа из u в v. У удаленной вершины u может быть правое поддерево, если оно есть, следует подвесить его к предку u на ее место [6].

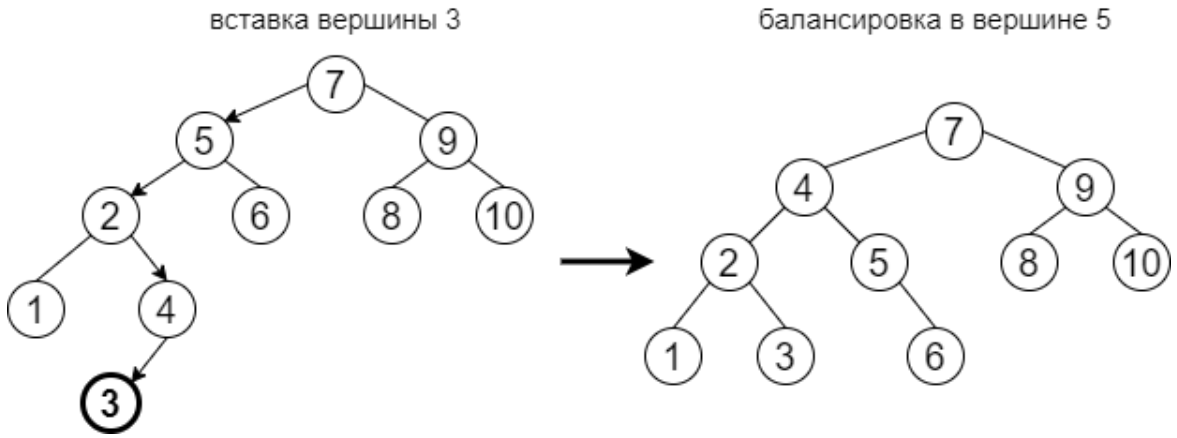

Рисунок 18 – Вставка ключа 3 в АВЛ-дерево

После этого необходимо пройти по пути от удаленной вершины к корню и восстановить баланс так же, как это делается при вставке.

Рассмотрим процесс удаления ключа на конкретном примере. На рисунке 19 изображено удаление ключа 2 из АВЛ-дерева. У вершины с ключом 2 есть правое поддерево, в нем ищется наименьший ключ – 3. В вершине с ключом 2, ключ меняется на 3, а вершина с ключом 3 удаляется, ее правый потомок, вершина 4, подвешивается к вершине с измененным ключом в качестве правого потомка. После этих действий разность высот поддеревьев для вершины 5 становится равной -2, что приводит к нарушению баланса. Для его восстановления проводится балансировка в вершине 5, совершается большое левое вращение.

Асимптотика добавления/удаления вершины в АВЛ-дерево составляет O(logn), так как любое вращение выполняется за O(1), а их количество не превышает высоту дерева, которая, как известно, приблизительно равна logn.

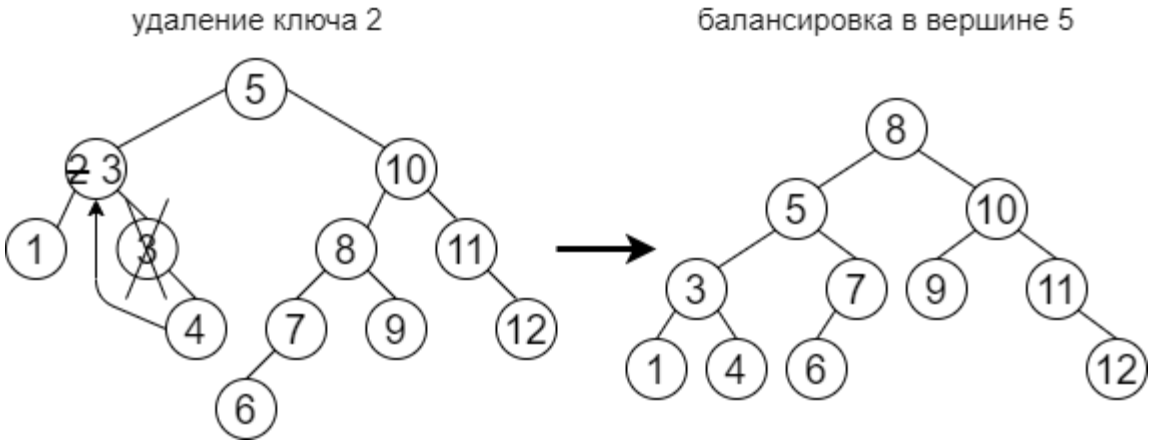

Рисунок 19 – Большое левое вращение

Таким образом, высота АВЛ-дерева равна O(logn) за счет того, что высоты поддеревьев любой из его вершин отличаются не более, чем на 1, что позволяет быстро осуществлять поиск путем спуска из корня и балансировку при вставке и удалении ключа.

1.5 B-дерево

B-дерево это сильноветвящееся сбалансированное дерево поиска, позволяющее производить быстрый поиск, добавление и удаление элементов во множество, изобретенное Р. Бейером и Э. МакКрейтом в 1970 году [5].

B-дерево является идеально сбалансированным, так как все листья в нем находятся на одинаковом расстоянии от корня. Его структура зависит от параметра t (t >= 2), который обычно называют минимальной степенью дерева, и определяется следующими правилами:

1. Любой узел, кроме корня должен содержать от t – 1 до 2t – 1 ключей.
2. Корень дерева должен содержать от 1 до 2t – 1 ключей, либо 0 ключей в случае, если дерево пустое.
3. Ключи во всех узлах упорядочены по возрастанию, у каждого узла (кроме листьев) с ключами k_1, … k_n ровно n+1 потомок, и значения ключей во

всем поддереве для каждого из них должны лежать в отрезке $[k_{i-1}, k_i]$, $k_0 = -\infty$, $k_{n+1} = +\infty$.

Пример B-дерева с параметром t=2 представлен на рисунке 20:

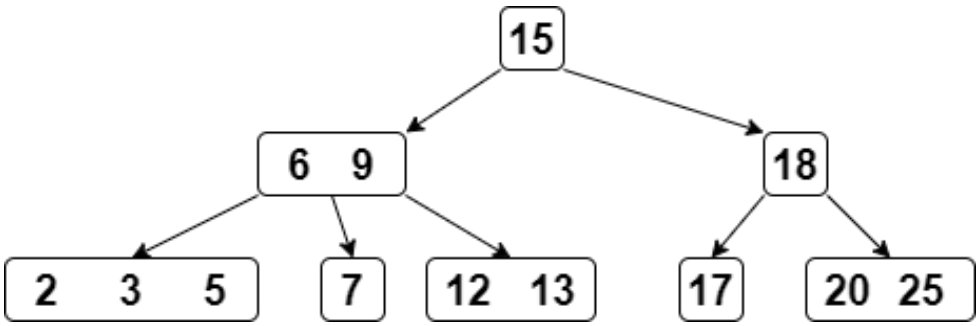

Рисунок 20 – Пример B-дерева с t=2

Поиск ключа в этой структуре осуществляется с помощью спуска из корня, на каждом шаге ключ ищется среди ключей в самой вершине, если его там нет, происходит спуск в потомка с соответствующим искомому ключу диапазоном значений, если достигнут лист, а ключ не найден, его в дереве нет. Поиск ключа в вершине занимает O(t), а кол-во рассмотренных вершин равно высоте дерева, которая составляет $\log_t n$, следовательно время поиска $O(t * \log_t n)$ [4].

Для того, чтобы вставить ключ в дерево, необходимо спуститься из корня в соответствующий лист, как будто проходит поиск этого ключа, затем добавить ключ в этот лист так, чтобы все ключи там по-прежнему находились в порядке возрастания. Однако может оказаться, что в листе уже было $2t - 1$ ключа перед вставкой, значит после их будет $2t$, что превышает лимит. Поэтому нужно организовать спуск в дереве таким образом, чтобы количество вершин в листе стало меньше $2t - 1$. Это делается так: каждый раз, когда во время спуска в текущей вершине $2t - 1$ ключа, необходимо разделить ее на две способом, изображенным на рисунке 21:

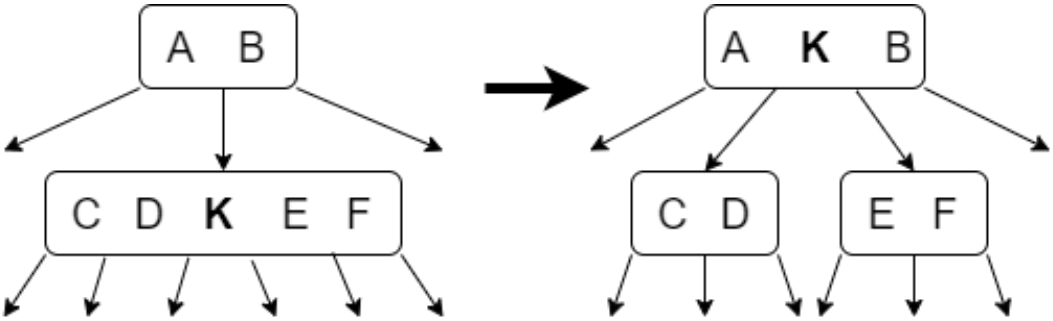

Рисунок 21 – Разделение вершины в B-дереве на 2 части

Ключ K, который находится в середине вершины, перемещается в ее предка, а сама вершина делится на 2 части, размер каждой из получившихся частей станет t – 1, в то же время размер предка не превысит 2t – 1 после добавления в него, так как до этого он был меньше. Если это корень, то он также разделится на 2 вершины, которые станут потомками нового корня содержащего один ключ K. Деление вершины на 2 и перемещение ключа в предка занимает O(t), значит асимптотика вставки $O(t * \log_t n)$ [12].

На рисунке 22 показаны этапы вставки ключа 4 в B-дерево с t = 2. В корне находятся 3 ключа: 2, 7 и 9. Раз кол-во ключей в вершине равно 2t – 1, необходимо разделить ее, ключ 7 из середины переместится в новый корень, в получившихся при разделении вершинах будет по 1 ключу, из корня будет совершен переход в вершину с ключом 2. У этой вершины 2 потомка, следует перейти в правого, так как 4 > 2. В этой вершине тоже 3 ключа, поэтому центральный ключ 5 переносится в предка, ключ 4 добавляется в лист с ключом 3.

Удаление ключа из листа возможно в случае, когда в нем t или больше ключей, либо это корень, который в этом случае является единственной вершиной в B-дереве, поэтому во время спуска из корня нужно добавлять ключ в вершину, в которую осуществляется переход, если там лишь t – 1 ключ. В случае, когда у вершины с t – 1 ключами, в которую осуществляется переход

есть сосед слева или справа, в котором хотя бы t ключей, можно совершить перемещение ключа из соседа в предка, а из предка в нужную вершину.

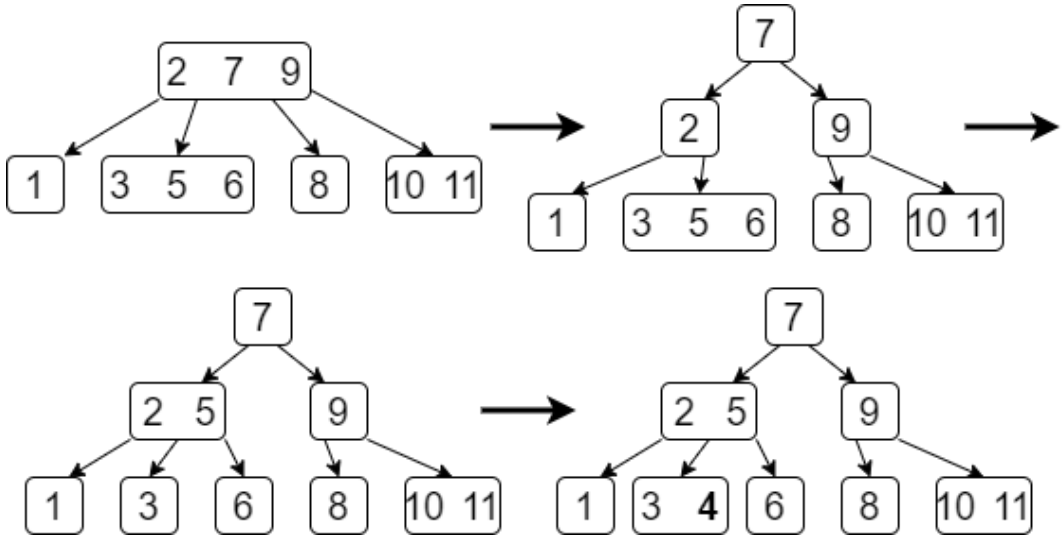

Рисунок 22 – Вставка ключа 4 в B-дерево

На рисунке 23 изображено перемещение ключа K из левого соседа в предка на место ключа K2, который в свою очередь перемещается в вершину с t – 1 ключом:

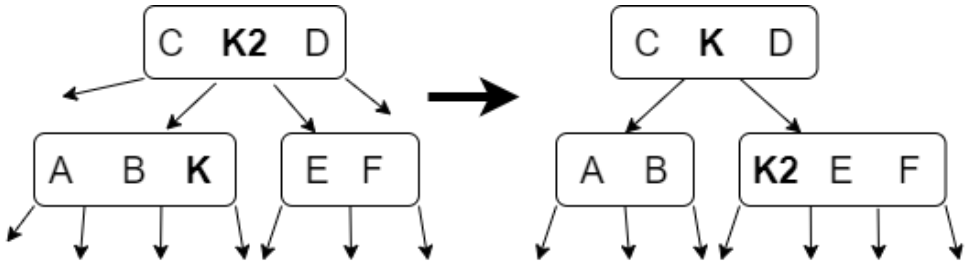

Рисунок 23 – Перемещение ключа из левого соседа в B-дереве

Подобным образом осуществляется и перемещение из правого соседа. Если количество ключей во всех соседях равно t – 1, вариант с перемещением ключа невозможен, поэтому нужно выполнить слияние вершины с соседом

(хотя бы один сосед у вершины есть всегда, так как t >= 2), соединив их с помощью ключа K из предка, как показано на рисунке 24:

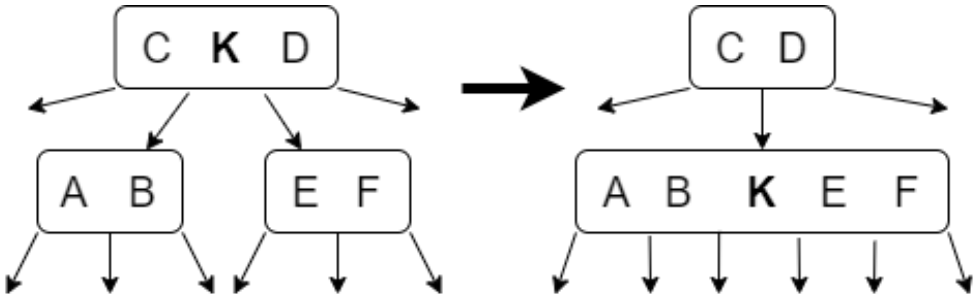

Рисунок 24 – Соединение вершин в B-дереве с помощью ключа из предка

Количество ключей в полученной вершине составит (t – 1) * 2 + 1 = 2t – 1, а в предке станет на 1 вершину меньше. Если предок не является корнем, в нем как минимум t вершин до слияния, и их не станет меньше t – 1, а из корня можно всегда удалить вершину, не нарушив ограничений. В том случае, когда это будет последняя вершина в корне, новым корнем станет вершина, получившаяся в результате слияния.

Если вершина, которую следует удалить, находится во внутреннем узле, необходимо проверить количество ключей в потомке слева от нее, и, если их t или больше, найти и удалить в поддереве этого потомка вершину с наибольшим ключом (она всегда в листе), а значение в текущей вершить заменить на значение в удаленной. Аналогично можно провести замену на самый маленький ключ в поддереве правого потомка. Если оба потомка имеют по t – 1 ключу, следует склеить их с помощью вершины, которую нужно удалить, как это показано на рисунке 9, а затем продолжить процесс, спустившись в созданную в результате объединения вершину.

Можно убедиться, что все описанные выше случаи при удалении работают за $O(t * \log_t n)$.

Рассмотрим подробно процесс удаления ключа 10 из B-дерева с t = 2 на рисунке 25. Из корня нужно совершить переход в правого потомка, но в связи

с тем, что там t – 1 ключ, это сделать нельзя. В левом соседе этой вершины 2 ключа, что больше t – 1, значит можно совершить перемещение правого ключа 5 в корень, а ключ 8 из корня переместить в вершину, после чего туда можно переходить. Ключ 10, который нужно удалить находится во внутреннем узле, значит просто удалить его оттуда невозможно. В потомке слева от ключа 10 количество ключей равно t – 1, значит заменить его на ключ оттуда нельзя. В потомке справа 3 ключа, следовательно можно взять наименьший ключ из него (11) и заменить ключ 10 на него.

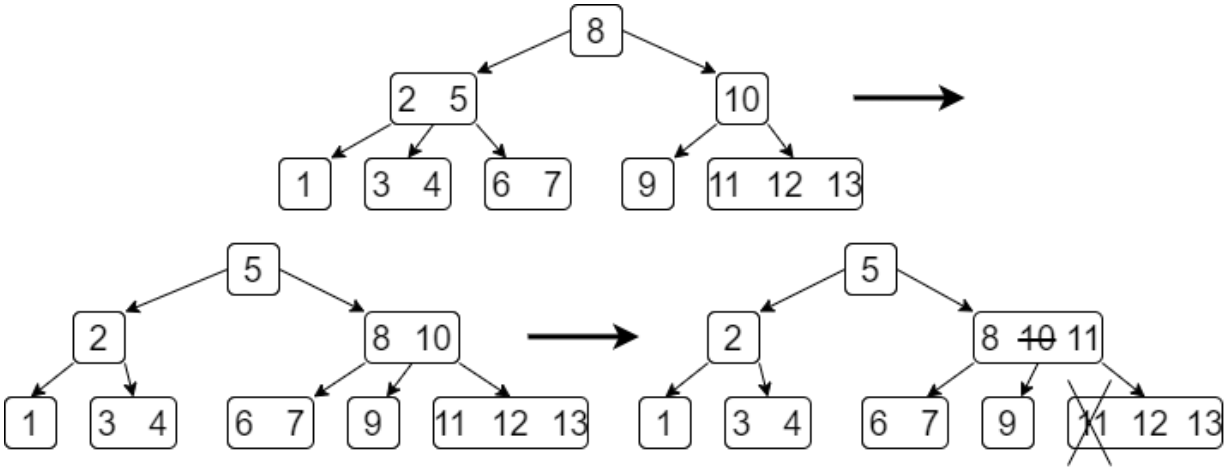

Рисунок 25 – Удаление ключа 10 из B-дерева

Таким образом, высота B-дерева равна $O(\log_t n)$ за счет того, что количество ключей во всех его вершинах кроме корня должно быть в диапазоне от t – 1 до 2t – 1 включительно, что позволяет быстро осуществлять поиск путем спуска из корня, а также вставку и удаление ключей.

Глава 2 Тестирование деревьев поиска

Как правило, асимптотической оценки сложности алгоритма для определения его реальной эффективности на практике недостаточно, поэтому прибегают к созданию различных тестов и запуску алгоритма на них для объективной оценки.

2.1 Поиск оптимального параметра t для B-дерева

Производительность и память, занимаемая структурой для декартова дерева, рандомизированного бинарного дерева поиска, splay-дерева и АВЛ-дерева зависит лишь от размера самого дерева и операций, производимых над ним. В отличие от них у B-дерева есть параметр t, определяющий диапазон возможных значений количества ключей в вершине дерева. Асимптотика поиска, вставки и удаления в B-дереве $O(t * \log_t n)$, однако на практике не всегда выгодно выбирать именно такой t при количестве вершин в дереве n, чтобы минимизировать значение $t * \log_t n$. Чтобы выяснить оптимальные t для различных размеров деревьев n, на которых будут проводиться дальнейшие тестирования, необходимо поставить такую задачу, чтобы по ходу ее выполнения размер структуры был порядка n, и в то же время над деревом выполнялись все возможные операции: вставка, удаление, поиск.

Для тестирования сначала построим дерево из n вершин, а затем будем с одинаковой вероятностью совершать над ним одну из трех операций. Для нахождения оптимального t, измерим время, затраченное на выполнение операций над B-деревом (не учитывая время построения). Так как вероятность того, что текущая операция является вставкой ключа равна вероятности того, что она является удалением, размер дерева не будет сильно изменяться по ходу выполнения действий над ним.

Алгоритм тестирования будет выглядеть следующим образом:

1. Построить B-дерево из n вершин. Для этого создадим массив keys из n элементов с индексами [0, n-1] и установим keys[i] = i. Затем перемешаем числа в этом массиве и в цикле по i от 0 до n-1 добавим ключ keys[i] в структуру.

2. Засечь время старта тестирования.

3. m раз выполнить одно из трех действий: вставку, добавление или удаление ключа. Чтобы определить какую операцию выполнить, сгенерировать целое число от 0 до 2, если это 0 – добавить ключ x, если такого в структуре нет, 1 – удалить ключ x, если такой есть в дереве, 2 – выполнить поиск ключа x. Значение x – это случайное число в диапазоне [0, n-1].

4. Вычислить текущее время, вычесть из него время старта, получив время, затраченное на модификации дерева и поиск в нем.

Возьмем количество операций $m = 10^5$ и для $n = 10^3$, $5*10^3$, 10^4, $5*10^4$, 10^5, $5*10^5$, 10^6, проведем вычисления используя параметр t = 10, 20, … , 90, 100. Для каждой пары (n, t), сделаем 10 запусков и вычислим среднее значение по ним.

На рисунке 26 представлен график, составленный по результатам описанного алгоритма тестирования. На графике изображена зависимость времени, затраченного на выполнение m запросов к B-дереву (ось y) от его параметра t (ось x).

Из полученных результатов видно, что с ростом размера дерева растет и время выполнения модификаций и поиска. Для каждого из рассматриваемых n, график, как и следовало ожидать, похож на график функции $y = x * \log_x n$, сначала с увеличением t производительность постоянно улучшается, затем в какой-то точке ситуация меняется, и производительность начинает постоянно ухудшаться.

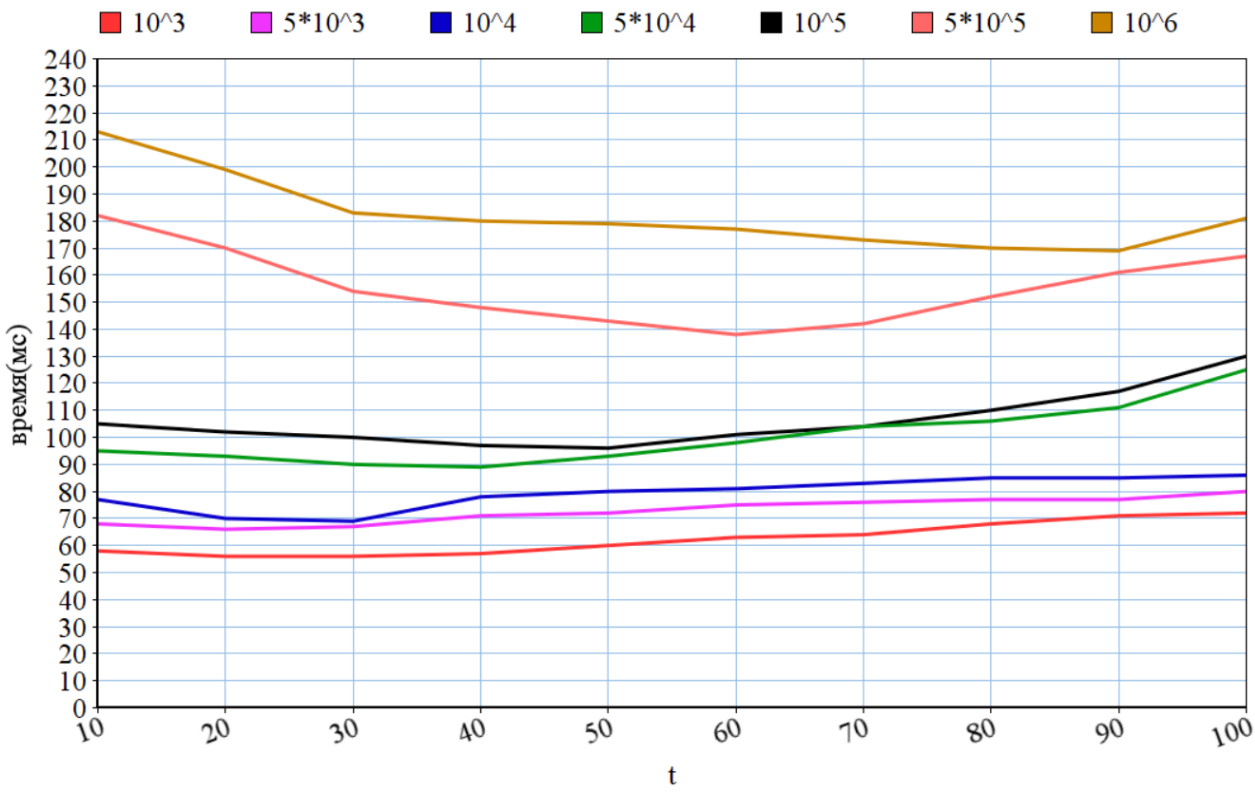

Рисунок 26 – Результаты тестирования для различных пар (n, t)

В таблице 1 представлено оптимальное значение t для каждого из n на котором проводилось тестирование. Именно эти t будут использоваться при сравнении эффективности B-дерева из n вершин с любым из остальных рассматриваемых здесь деревьев такого же размера.

Таблица 1 – Оптимальные значения t для B-дерева из n вершин

N	10^3	5*10^3	10^4	5*10^4	10^5	5*10^5	10^6
T	20	20	30	40	50	60	90

После того, как оптимальное значение t было вычислено для различных n опытным путем, можно производить сравнения B-деревьев с остальными рассматриваемыми в работе структурами.

2.2 Тестирование на задаче построения дерева

Сравним время построения декартова дерева, рандомизированного бинарного дерева поиска, splay-дерева, B-дерева и АВЛ-дерева из набора ключей. Для этого будем вставлять ключи из диапазона [0, n-1] в случайном порядке. Проведем тестирование для n = 10^3, 5*10^3, 10^4, 5*10^4, 10^5, 5*10^5, 10^6, чтобы получить наиболее точное значение, сделаем несколько запусков с различным порядком вставки ключей, возьмем среднее значение.

На рисунке 27 представлен график зависимости времени построения дерева из n вершин от значения n:

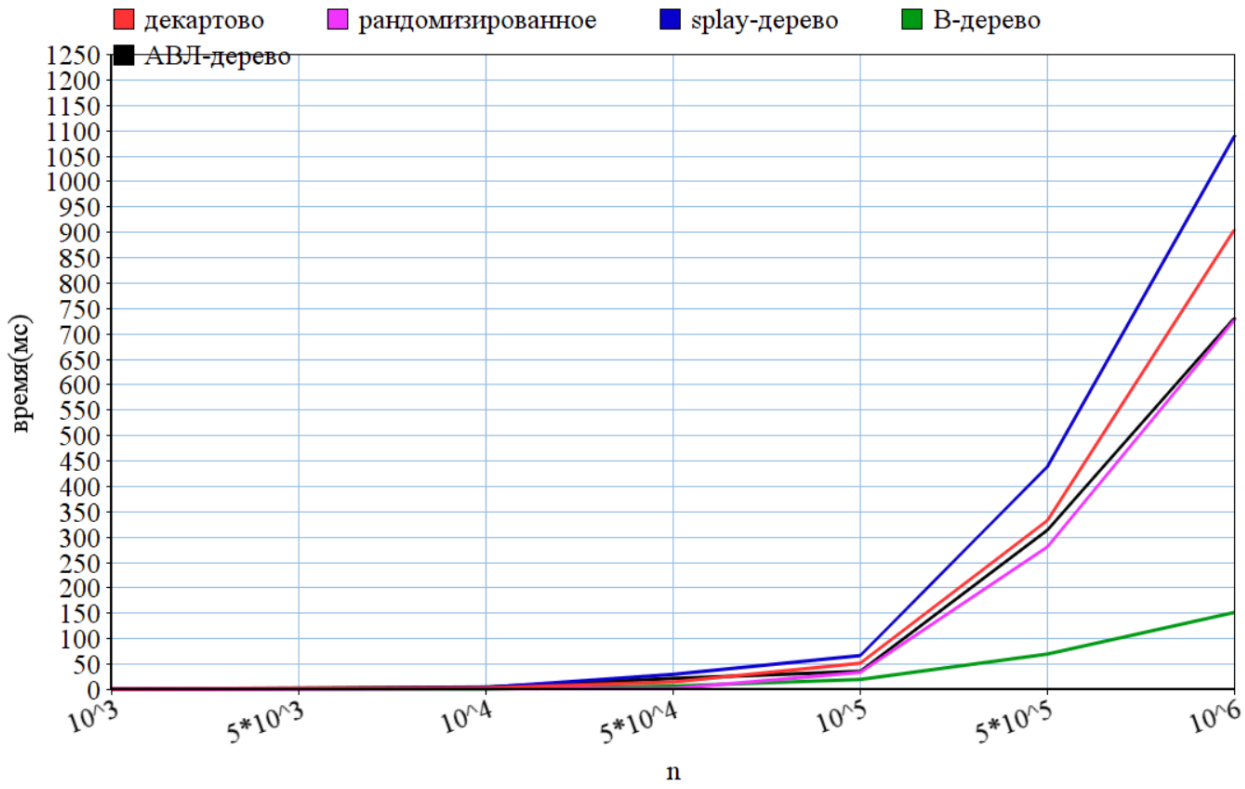

Рисунок 27 – Время построения дерева из n вершин

Для всех деревьев наблюдается линейная зависимость времени построения дерева из набора ключей от его размера. Самой эффективной структурой по результатам тестирования оказалось B-дерево, зачастую при

добавлении нового ключа структура дерева не меняется, он просто вставляется в лист, в котором еще есть свободное место. Следующим по скорости стало рандомизированное дерево, вставка ключа в него с вероятностью n/(n + 1) является достаточно быстрой операцией, осуществляется спуск до нужной позиции и ключ вставляется в качестве листа. Лишь с небольшой вероятностью 1/(n + 1) приходится совершать вставку в корень и перестраивать дерево. Похожие результаты показало АВЛ-дерево, в нем также часто новый ключ просто вставляется в лист без дальнейшей перестройки структуры, в связи с тем, что балансировка не требуется. В декартовом дереве вставка осуществляется немного медленнее, чем в рандомизированном и АВЛ деревьях, для ее осуществления всегда нужно совершать split дерева по вставляемому ключу, а затем сливать его с получившимися деревьями с помощью merge, а это достаточно затратные по времени операции. Медленнее всех осуществляется построение splay-дерева, там новый ключ всегда вставляется в корень, а его левым и правым поддеревьями становятся части дерева, разделенного операцией splay, которая оказалось достаточно затратной по времени.

Одним из важнейших факторов для любой структуры данных является размер памяти, необходимый для ее хранения. Проведем тестирование и измерим количество памяти, потребляемой рассматриваемыми в работе деревьями. Его результаты представлены на рисунке 28 в виде графика, отображающего зависимость размера занимаемой памяти от количества вершин в дереве.

Наименее затратной по памяти структурой является B-дерево, в каждой его вершине хранится массив из t ключей и массив из t+1 указателей на потомков, также там хранится информация о том, сколько ключей содержит вершина в текущий момент и является ли она листом. Таким образом, B-дерево из n вершин занимает порядка 2n памяти. Следующими по оптимальному потреблению памяти являются декартово, рандомизированное и splay деревья, они показали приблизительно одинаковые результаты при тестировании, их

графики наложились друг на друга. В этих деревьях на каждый ключ приходится по отдельной вершине, в которой хранится значение ключа и указатели на левого и правого потомков. В вершине также хранится дополнительная информация, для декартова дерева это приоритет, для рандомизированного это размер поддерева с корнем в данной вершине, для splay дерева – указатель на предка, необходимый для осуществления вращений при операции splay. Эти деревья требуют порядка 4n памяти для хранения n ключей. Наихудший результат показало АВЛ-дерево, в нем, так же, как и во всех остальных структурах, кроме B-дерева, каждый ключ хранится в отдельной вершине, дополнительно к ключу и ссылкам на потомков приходится хранить высоту поддерева с корнем в этой вершине и разницу высот ее левого и правого поддеревьев для проведения балансировки, все это занимает 5n памяти.

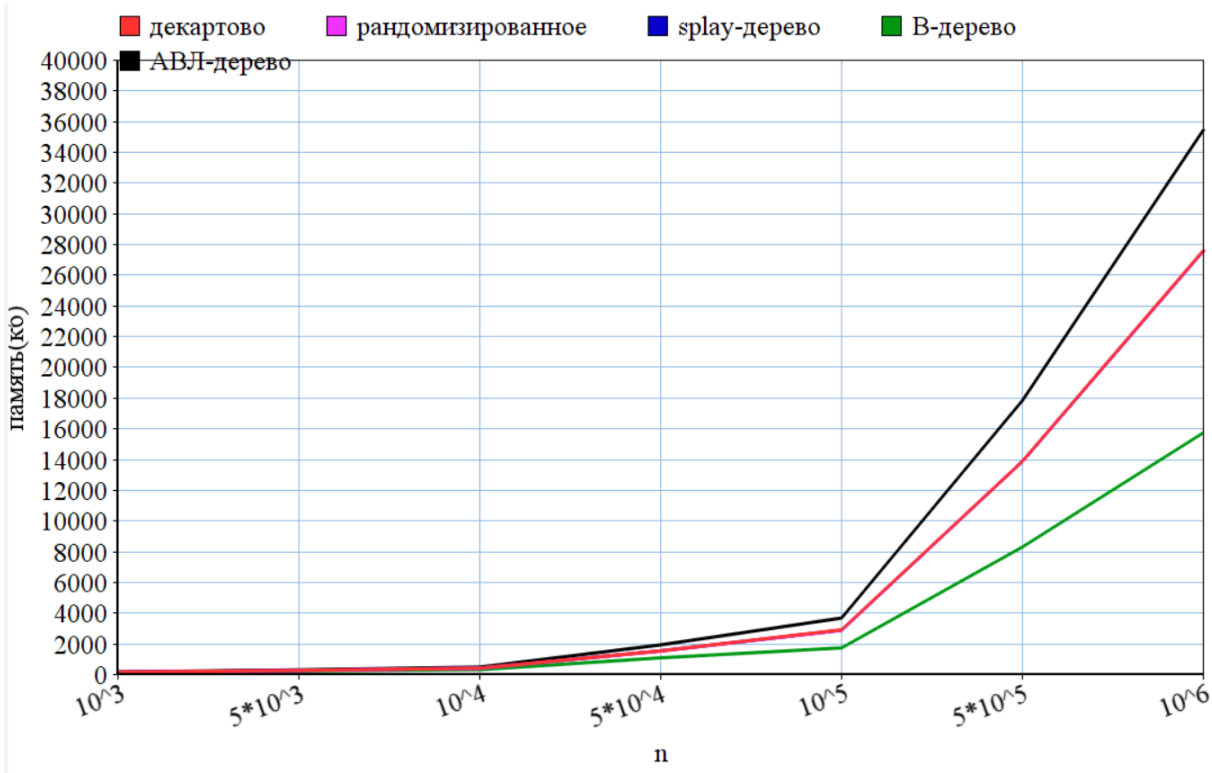

Рисунок 28 – Память, занимаемая деревом из n вершин

Тестирование показало, что с возрастанием n B-дерево строится из набора ключей заметно быстрее, чем остальные структуры, а также занимает в 2 - 2.5 раз меньше памяти.

2.3 Тестирование на задаче поиска ключа

Основной операцией для дерева поиска является проверка наличия в нем определенного ключа. Чтобы сравнить эффективность рассматриваемых деревьев поиска, проведем тестирование по следующему алгоритму для $n = 10^3, 5*10^3, 10^4, 5*10^4, 10^5, 5*10^5, 10^6$:

1. Построить дерево из n вершин. Для этого создадим массив keys из n элементов с индексами [0, n*2-1] и установим keys[i] = i. Затем перемешаем числа в этом массиве и в цикле по i от 0 до n-1 добавим первые n ключей keys[i] в структуру.
2. Засечь время старта тестирования.
3. 10^6 раз выполнить поиск ключа x. Значение x – это случайное число в диапазоне [0, n*2-1], таким образом ключ будет находиться/не находиться с вероятностью 50%.
4. Вычислить текущее время, вычесть из него время старта, получив время, затраченное на поиск в дереве.

На рисунке 29 изображен график зависимости времени поиска ключа в дереве от количества ключей в нем.

Быстрее всего поиск осуществляется в B-дереве, это обусловлено тем, что при правильном выборе параметра t, достигается идеальный баланс между высотой дерева и количеством ключей в его вершине. В Декартовом, рандомизированном и АВЛ деревьях время поиска зависит высоты дерева, как показало тестирование, в них поиск происходит медленнее, чем в B-дереве, самый лучший результат из этих трех деревьев у АВЛ-дерева, оно показывает схожие с B-деревом результаты при $n<10^5$, однако при больших размерах дерева, начинает уступать, самое медленное из этой тройки – декартово

дерево. Медленнее всех поиск ключа происходит в splay-дереве, так как после нахождения ключа нужно переместить его в корень с помощью действия splay.

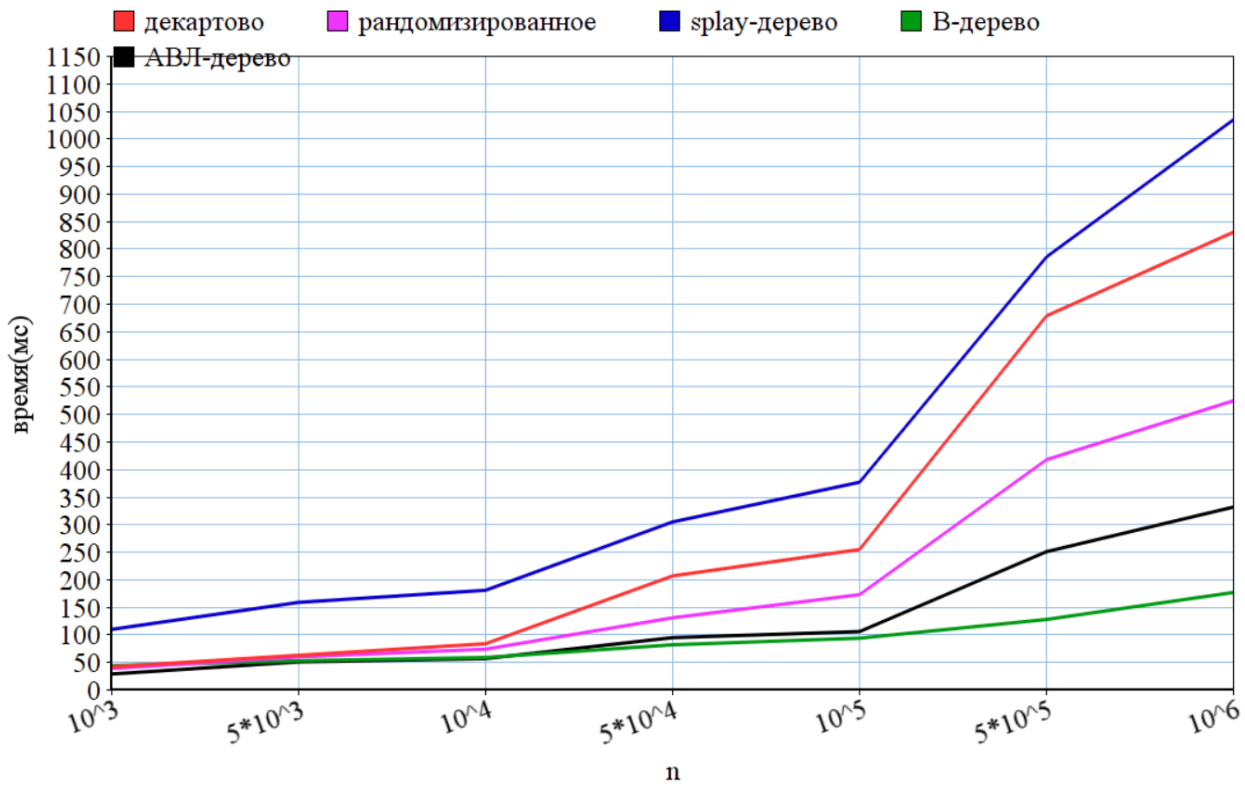

Рисунок 294 – Время на поиск ключа в дереве с n ключами

Тестирование показало, что среди всех рассматриваемых в работе деревьев поиска, в B-дереве поиск происходит быстрее, чем в остальных структурах, выигрыш в производительности в сравнении с самой медленной, splay-деревом, составляет 5-6 раз.

2.4 Тестирование на всех операциях

Проведем тестирование рассматриваемых структур, используя все три операции, которые можно совершить над ними: поиск, вставка, удаление ключа. Для тестирования используем алгоритм, который применялся при поиске оптимального параметра t для B-дерева, где над деревом содержащим

n ключей с одинаковой вероятностью совершается одна из 3 операций, и так 10^6 раз. Итоги тестирования представлены на рисунке 30 в виде графика, изображающего зависимость времени выполнения всех операций от количества ключей в структуре.

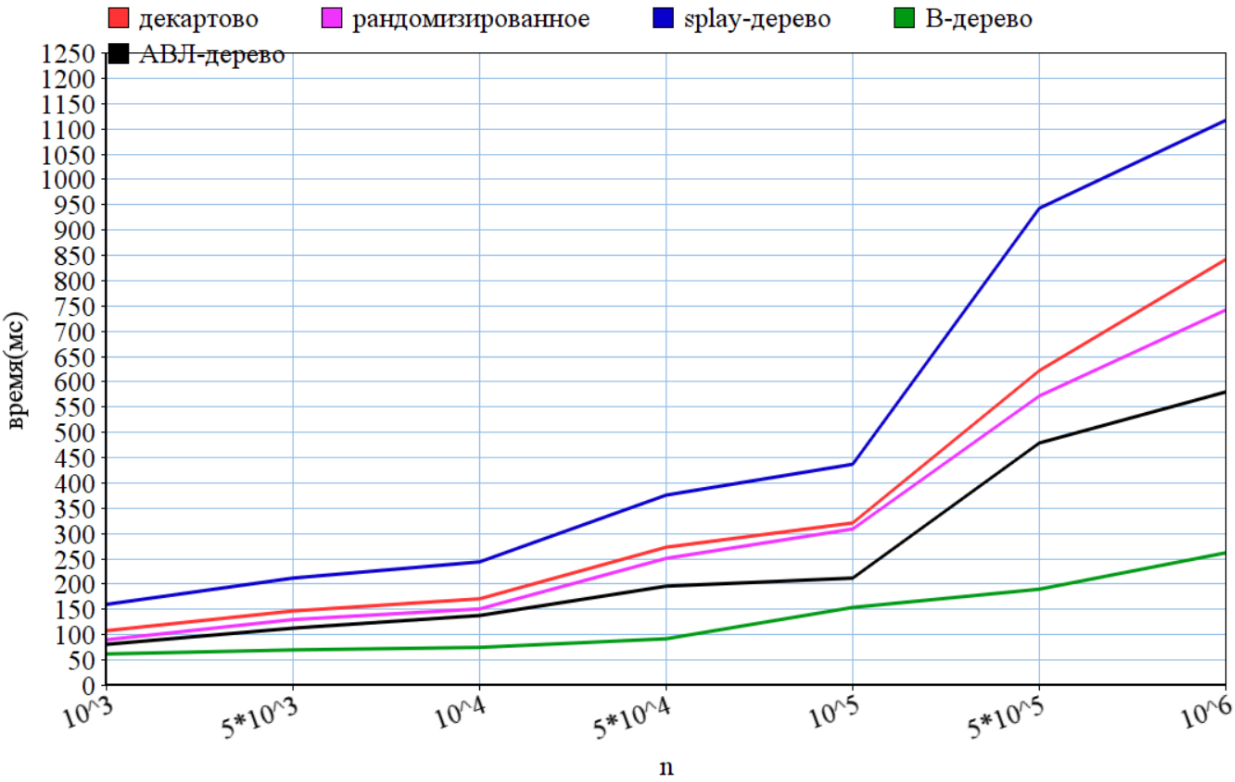

Рисунок 30 – Результаты тестирования на всех операциях

Тестирование показало, что среди всех рассматриваемых в работе деревьев поиска, B-дерево в среднем наиболее производительное, чем остальные структуры, выигрыш в производительности в сравнении с самой медленной, splay-деревом, составляет 4 раза.

2.5 Выводы по тестированию

По результатам тестирования на различных задачах наилучшие результаты показало B-дерево. Если примерно известно количество ключей в

наборе, над которыми будут производиться операции поиска/вставки/удаления ключа, и это количество не будет сильно изменяться со временем, можно ориентируясь на него, выбрать оптимальный параметр t, что позволит достичь максимальной производительности. B-дерево потребляет меньше памяти, чем остальные деревья за счет того, что одна вершина содержит несколько ключей, которые хранятся в массиве. Минусом данной структуры является сложность ее реализации, при удалении ключа возникает множество различных случаев, каждый из которых необходимо корректно обрабатывать.

Следующим по эффективности оказалось АВЛ-дерево, оно достаточно быстро строится из набора ключей, при вставке вершины не требуется разделять дерево на части, достаточно лишь спуститься и добавить лист, в редких случаях совершая балансировку. Недостатком является большое потребление памяти по сравнению с остальными рассматриваемыми структурами.

Рандомизированное и декартово деревья показали немного худшие результаты, чем АВЛ. Одной из основных причин является то, что вычисление случайного числа является довольно затратной по времени операцией. В декартовом дереве случайное число нужно сгенерировать, чтобы установить приоритет для новой вершины, а в рандомизированном случайное число необходимо для того, чтобы решить совершать обычную вставку, или вставку в корень, также оно нужно для определения корня дерева при слиянии. Большим плюсом данных структур является лаконичная реализация.

Наименее эффективной из всех рассмотренных структур оказалось splay-дерево. Его единственным плюсом является то, что доступ к данным, поиск по которым осуществлялся недавно, будет быстрее, так как за счет выполнения splay на каждом запросе на поиск, они ближе к корню.

На рисунке 31 изображена диаграмма с данными всех проведенных тестирований для n порядка 10^6:

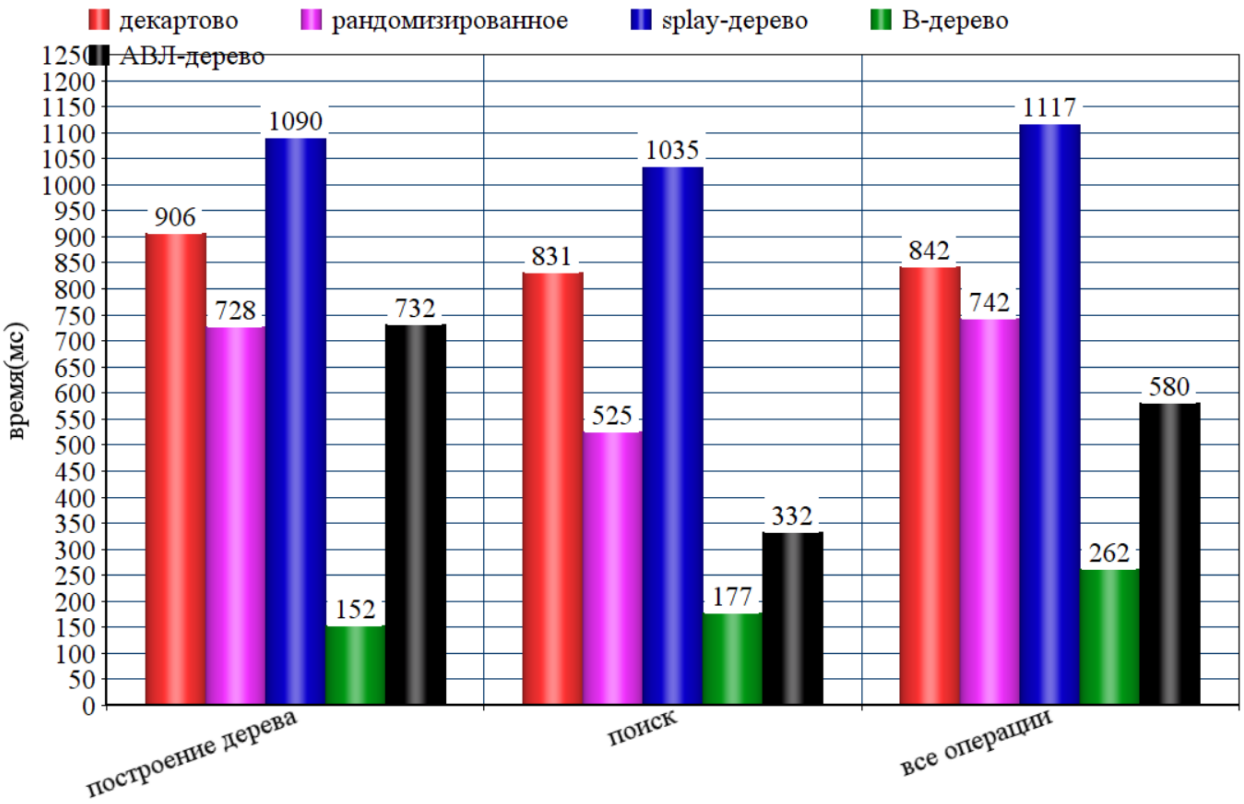

Рисунок 31 – Итоговые результаты тестирования

Тестирование показало, что B-дерево является наиболее эффективным и занимает меньше памяти, чем остальные деревья, рассматриваемые в этой работе. Однако нельзя утверждать, что использование других структур не имеет смысла, так как у них есть различные возможности, которых в B-дереве нет, например, декартово и рандомизированное деревья позволяют слияние двух деревьев и разделение дерева по ключу за O(logn), splay-дерево позволяет совершать поиск ключа, который недавно вставили/искали за O(1).

Заключение

В ходе выполнения бакалаврской работы была рассмотрена задача поиска ключа во множестве и модификации этого множества путем добавления/удаления ключей. На протяжении многих лет для решения этой задачи создавались различные структуры данных, большая часть из которых является сбалансированными деревьями поиска. У деревьев поиска есть множество применений на практике, на их основе реализована структура данных ассоциативный массив в различных языках программирования, деревья используются при построении индексов в базах данных. Именно поэтому решение этой задачи очень важно.

Был рассмотрен принцип работы таких структур, как декартово дерево, рандомизированное бинарное дерево поиска, splay-дерево, АВЛ-дерево, B-дерево, позволяющих достаточно эффективно решать задачу поиска во множестве и его модификации, дана асимптотическая оценка сложности операции, производимых в этих деревьях.

После этого, был разработан алгоритм генерации данных для тестирования, затем на полученных данных было проведено тестирование.

Исходя из результатов тестирования был проведен сравнительный анализ рассматриваемых деревьев поиска на различных задачах. На практике, наиболее эффективным показало себя B-дерево. При выборе оптимального значения параметра t данная структура превосходит остальные в производительности.

Однако тестирование также показало, что структуры данных могут по-разному показывать себя на различных задачах относительно друг друга, поэтому при выборе структуры для решения конкретной задачи, всегда нужно проводить тестирование на данных, максимально приближенных к этой задаче, и по результатам этого тестирования совершать выбор.

Список используемой литературы

1. Адельсон-Вельский Г. М., Ландис Е. М. Один алгоритм организации информации // Доклады АН СССР. – 1962. – Т. 146, № 2. – С. 263–266. URL: http://www.mathnet.ru/links/20762a931307aef02c107c3d8f6244b8/dan26964.pdf
2. Алгоритмы. Построение и анализ. Томас Кормен [и др.] – М.: «Вильямс», 2019. – 1328 с.
3. Вирт Н. Алгоритмы и структуры данных. – ДМК Пресс, 2011. – 272 с.
4. Кнут Д. Искусство программирования. Том 3. Сортировка и поиск. – Диалектика-Вильямс, 2019. – 832 с.
5. Левитин А. В. Алгоритмы. Введение в разработку и анализ. – М.: Вильямс, 2006. – 576 с.
6. АВЛ-дерево [Электронный ресурс]. - Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=АВЛ-дерево
7. АВЛ-деревья [Электронный ресурс]. - Режим доступа: https://habr.com/ru/post/150732/
8. Декартово дерево [Электронный ресурс]. - Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=Декартово_дерево
9. Декартово дерево: Часть 1. Описание, операции, применения [Электронный ресурс]. - Режим доступа: https://habr.com/ru/post/101818/
10. Рандомизированное бинарное дерево поиска [Электронный ресурс]. - Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=Рандомизированное_бинарное_дерево_поиска
11. Рандомизированные деревья поиска [Электронный ресурс]. - Режим доступа: https://habr.com/ru/post/145388/
12. B-дерево [Электронный ресурс]. - Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=B-дерево

13. Splay-дерево [Электронный ресурс]. - Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=Splay-дерево

14. Splay-деревья [Электронный ресурс]. - Режим доступа: https://habr.com/ru/company/JetBrains-education/blog/210296/

15. Bruce Reed. The height of a random binary search tree, 2003. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.152.1289&rep=rep1&type=pdf

16. Daniel Sleator, Robert Tarjan. Self-Adjusting Binary Search Trees, 1985. URL: http://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf

17. Martinez Conrado, Roura Salvador. Randomized binary search trees, 1997. URL: https://www.researchgate.net/publication/220432012_Randomized_Binary_Search_Trees

18. Raimund Seidel, Cecilia R. Aragon. Randomized Search Trees, 1996. URL: https://faculty.washington.edu/aragon/pubs/rst96.pdf

19. Robert Tarjan. Data Structures and Networks Algorithms, 1987. URL: https://doc.lagout.org/Others/Data%20Structures/Data%20Structures%20and%20Network%20Algorithms%20%5BTarjan%201987-01-01%5D.pdf

20. R. Bayer, E. McCreight. Organization and Maintenance of Large Ordered Indexes, 1970. URL: https://infolab.usc.edu/csci585/Spring2010/den_ar/indexing.pdf