МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: «Исследование абстрактных структур данных. АВЛ-дерево vs RB-дерево»

Студент гр. 0304 _____ Максименко Е.М.

Преподаватель _____ Берленко Т.А.

Санкт-Петербург
2021

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Максименко Е.М,

Группа 0304

Тема работы: «Исследование абстрактных структур данных. АВЛ-дерево vs RB-дерево»

Исходные данные:

АВЛ-дерево vs RB-дерево. **Исследование**. "Исследование" — реализация требуемых структур данных/алгоритмов; генерация входных данных (вид входных данных определяется студентом); использование входных данных для измерения количественных характеристик структур данных, алгоритмов, действий; сравнение экспериментальных результатов с теоретическими. Вывод промежуточных данных не является строго обязательным, но должна быть возможность убедиться в корректности алгоритмов.

Содержание пояснительной записки:

«Содержание», «Введение», «Реализация структур данных», «Тестирование корректности реализаций структур данных», «Исследование структур данных», «Заключение», «Список использованных источников»

Предполагаемый объем пояснительной записки:

Не менее 15 страниц.

Дата выдачи задания: 26.10.2021

Дата сдачи реферата:

Дата защиты реферата:

Студент _____ Максименко Е.М,

Преподаватель _____ Берленко Т.А.

АННОТАЦИЯ

В данной работе были исследованы такие структуры данных, как АВЛ-дерево и RB-дерево. Данные структуры данных реализованы на языке C++. В работе была проведена проверка корректности реализаций и замеры фактического времени работы некоторых операций в данных структурах данных.

СОДЕРЖАНИЕ

1. ВВЕДЕНИЕ

1.1. Цель курсовой работы и исходные условия

Исходные условия:

АВЛ-дерево vs RB-дерево. **Исследование**.

"Исследование" **—** реализация требуемых структур данных/алгоритмов; генерация входных данных (вид входных данных определяется студентом); использование входных данных для измерения количественных характеристик структур данных, алгоритмов, действий; сравнение экспериментальных результатов с теоретическими. Вывод промежуточных данных не является строго обязательным, но должна быть возможность убедиться в корректности алгоритмов.

Целью данной работы является реализация данных структур данных и тестирование их характеристик производительности (потребляемое процессорное время) в зависимости от количества элементов в дереве.

1.2 Задачи

Для достижения поставленной цели требуется решить следующие задачи:

- Реализовать логику требуемых структур данных (АВЛ-дерева и RB-дерева)
- Реализовать набор тестов для проверки корректности работы реализации структур данных
- Реализовать модуль для исследования характеристик производительности структур данных

2. РЕАЛИЗАЦИЯ СТРУКТУР ДАННЫХ

2.1. АВЛ-дерево

АВЛ-дерево является сбалансированным по высоте двоичным деревом поиска, для которого выполняется свойство: для каждой его вершины высота ее поддеревьев различается не более, чем на 1.

АВЛ-дерево поддерживает операции поиска, вставки и удаления элементов, которые и были реализованы в коде.

Классы АВЛ-дерева и узла АВЛ-дерева

Для реализации АВЛ-дерева был создан класс узла этого дерева — *AVLNode*, который хранит в себе указатели на своих детей, а также данные и высоту дерева, корнем которого данный узел является. Класс имеет набор геттеров и сеттеров, а также имеет метод для получения фактора балансировки дерева, корнем которого является данный узел. Фактор балансировки — разница высот правого и левого поддеревьев узла.

Был создан класс самого АВЛ-дерева *AVLTree*. В данном классе имеется единственное поле — ссылка на корневой узел. Были созданы основные методы для работы с данной структурой данных, такие как методы поиска, вставки и удаления узла. Были также созданы вспомогательные методы, такие как методы восстановления баланса.

Поиск в АВЛ-дереве

Поиск узла в AVL-дереве основан на свойствах бинарных деревьев поиска: элементы с меньшим по значению ключом располагаются в левом поддереве узла, элементы с большим по значению ключом — в правом поддереве узла. Поэтому для поиска был использован цикл, в котором происходит сравнение искомого ключа с ключом узла: если они равны, то цикл останавливается и возвращается указатель на данный узел, если искомый ключ меньше, происходит поиск в левом поддереве узла, если искомый ключ больше — в правом поддереве. Если был достигнут листовой элемент и узел не был найден, поиск вернет *nullptr*. Так как узлы, которые посещает функция

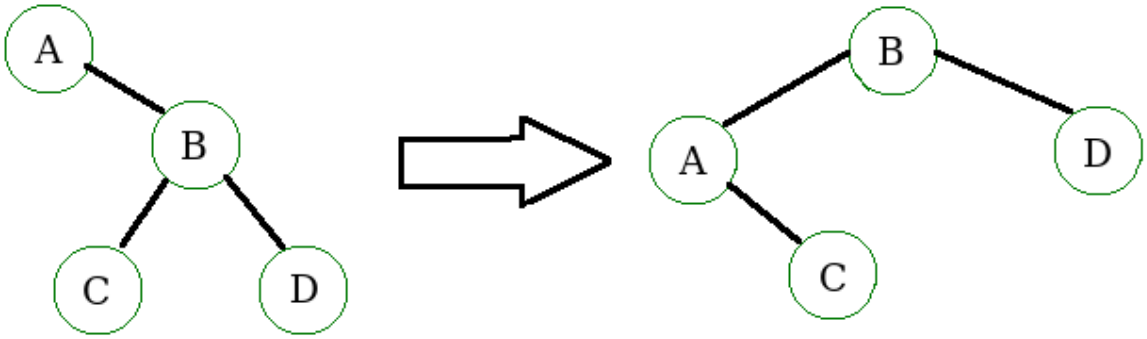
образуют нисходящий путь от корня, время работы операции составляет O(h), где h — высота дерева, т. е. O(log n).

Методы балансировки АВЛ-дерева

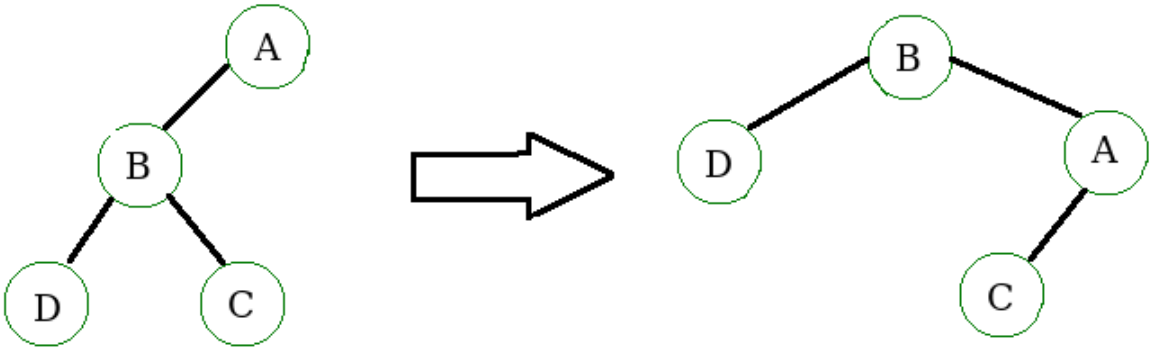
АВЛ-дерево имеет свойство, связанное с разницей высот поддеревьев каждого узла, для его поддержания используются методы балансировки. Существует 4 вида балансировки АВЛ-деревьев:

- Малый левый поворот
- Малый правый поворот
- Большой левый поворот
- Большой правый поворот

Алгоритм левого поворота осуществляется в три шага: текущий узел A заменяется его правым потомком B, узел C, являющийся левым потомком узла B, становится правым потомком узла A, сам узел A становится левым потомком узла B.



Алгоритм правого поворота осуществляется также в три шага: текущий узел A заменяется его левым потомком B, узел C, являющийся правым потомком узла B, становится левым потомком узла A, сам узел A становится правым потомком узла B.



Большой левый поворот по своей сути является объединением малых: сперва выполняется малый правый поворот относительно правого ребенка узла, потом — малый левый поворот относительно самого узла. Большой правый поворот является зеркальным к большому левому: сперва выполняется малый левый поворот относительно левого ребенка узла, потом — малый правый поворот относительно самого узла.

Для различных случаев разбалансировки используются различные виды балансировки. Так, если фактор баланса для узла составляет 2, то будет использоваться один из видов левого поворота, если он составляет -2 — один из видов правого поворота. Рассмотрим лишь случай с фактором баланса, равным двум, для -2 ситуация будет симметричной. Если высота правого поддерева больше высоты левого поддерева, то выполняется малый левый поворот, иначе — большой левый поворот.

Вставка элемента в АВЛ-дерево

В данной реализации АВЛ-дерева будет использоваться рекурсивный алгоритм вставки. Он заключается в том, чтобы найти позицию для вставки нового узла путем сравнения искомого ключа и ключа текущего узла. Если искомый ключ меньше ключа текущего узла, то будет вызван метод вставки, которому будет передан в качестве текущего узла левый потомок узла, с

которым происходило сравнение на предыдущем шаге, если больше — правый потомок. Если попытаться вставить в дерево узел с уже существующим ключом, будет сгенерировано исключение, так как ключи в дереве должны быть уникальными. Когда будет достигнут листовой узел, будет создан новый АВЛ-узел с переданным ключом и будет произведена балансировка дерева. Рекурсия используется для того, чтобы восстановить свойства дерева во всех узлах, по которым был произведен проход в процессе поиска места для вставки, так как при вставке фактор баланса в этих узлах изменяется. Так как в процессе вставки рассматривается не более O(h) вершин дерева, причем для каждой не более одного раза запускается балансировка, то сложность данной операции составит O(log n).

Удаление элемента в АВЛ-дереве

В данной реализации АВЛ-дерева используется рекурсивный алгоритм удаления узла. Поиск вершины осуществляется так же, как и в алгоритме вставки с использованием рекурсии. Если элемент не найден в дереве, удаление не происходит. Если удаляемый элемент является листовым, данный узел просто удаляется. Если у вершины есть лишь один потомок, то данный потомок занимает позицию удаленного узла. Если у вершины есть два потомка, то необходимо найти ближайший по значению к данному узлу элемент. Выполняется это переходом в правое поддерево удаляемого узла и поиском самого левого узла в данном поддереве. После этого переместим данный узел на место удаляемого, при этом вызвав процедуру его удаления. После того, как элемент был удален, необходимо провести балансировку дерева. Рекурсия используется для того, чтобы восстановить свойства дерева во всех узлах, по которым был произведен проход в процессе поиска удаляемого, так как при удалении фактор баланса в этих узлах изменяется. Так как удаление включает операцию поиска, требующую не более O(log n) операций, а также восстановление свойств только в тех вершинах, по которым был совершен проход в процессе поиска, общая сложность операции составит O(log n).

2.2. RB-дерево

RB-дерево (красно-черное дерево) является самобалансирующимся бинарным деревом поиска. Сбалансированность достигается за счет введения такой характеристики, как цвет узла. Цвет узла может быть одним из двух: красным либо черным (отсюда и берется название структуры данных). Для красно-черных деревьев существуют определенные свойства, которые должны быть выполнены, чтобы дерево было корректным:

1. Узел может быть либо красным, либо черным, при этом имеет двух потомков.
2. Корень имеет черный цвет.
3. Все листовые узлы, не содержащие данных, — черные.
4. Оба потомка красного узла — черные.
5. Любой простой путь из любого узла X до листьев содержит одинаковое количество черных узлов.

RB-дерево поддерживает операции поиска, вставки и удаления элементов, которые и были реализованы в коде.

Классы RB-дерева и узла RB-дерева

Для реализации RB-дерева был создан класс узла этого дерева — *RBNode*, который хранит в себе указатели на своих детей, указатель на родительский узел, а также данные и цвет данного узла. Класс имеет набор геттеров и сеттеров для обеспечения доступа к полям.

Был создан класс самого RB-дерева *RBTree*. В данном классе имеется единственное поле — ссылка на корневой узел. Были созданы основные методы для работы с данной структурой данных, такие как методы поиска, вставки и удаления узла. Также были созданы вспомогательные методы, такие как восстановление свойств дерева после вставки и удаления.

Поиск в RB-дереве

Поиск узла в RB-дереве такой же, как и в АВЛ-дереве, и основан на свойствах бинарных деревьев поиска: элементы с меньшим по значению ключом располагаются в левом поддереве узла, элементы с большим по

значению ключом — в правом поддереве узла. Поэтому для поиска был использован цикл, в котором происходит сравнение искомого ключа с ключом узла: если они равны, то цикл останавливается и возвращается указатель на данный узел, если искомый ключ меньше, происходит поиск в левом поддереве узла, если искомый ключ больше — в правом поддереве. Если был достигнут листовой элемент и узел не был найден, поиск вернет *nullptr*. Так как узлы, которые посещает функция образуют нисходящий путь от корня, время работы операции составляет O(h), где h — высота дерева, т. е. O(log n).

Методы балансировки RB-дерева

Для балансировки RB-дерева в определенных случаях могут использовать левый и правый повороты, а также перекраска узлов. Алгоритм левого поворота в RB-дереве следующий: берется правый потомок B текущего узла A, родителем узла B становится узел, который прежде был родителем A, причем у родителя сохраняется указатель на узел B (вместо A), после этого левый потомок узла B становится правым потомком узла A, родителем узла A становится узел B, а сам узел A становится левым потомком узла B. Алгоритм правого поворота является зеркальным к алгоритму левого поворота.

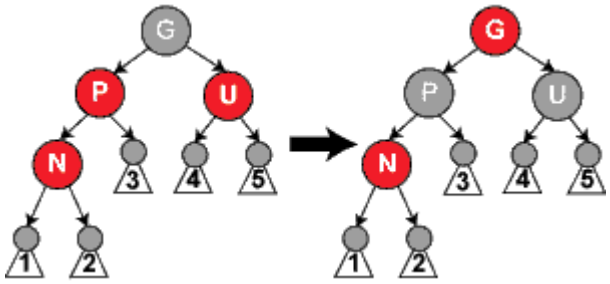
Вставка элемента в RB-дерево

Вставка нового элемента производится вместо листового узла, поэтому в цикле производится поиск места для вставки нового узла в соответствии с критерием бинарного дерева поиска. Если элемент с вставляемым ключом существует, будет выброшено исключение. Когда место было найдено, создается новый узел с необходимым ключом, по умолчанию его цвет красный. После вставки необходимо восстановить свойства RB-дерева.

В первом методе восстановления баланса проверяется, является ли вставленный узел корневым, если условие выполнено — узел окрашивается в черный цвет, иначе вызывается второй метод восстановления баланса.

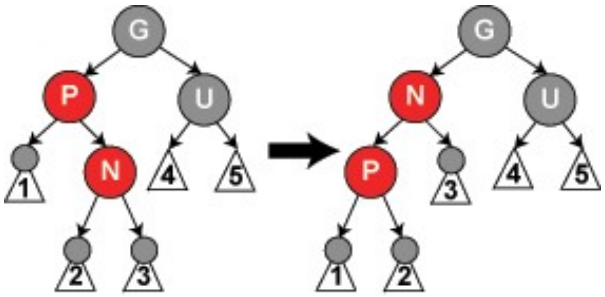
Во втором методе проверяется цвет родителя вставленного элемента: если родитель черный, то никакие свойства не нарушены, иначе вызывается третий метод восстановления баланса.

В третьем методе проверяется цвет дяди вставленного элемента: если дядя был красным, необходимо выполнить перекраску родителя и дяди элемента в черный цвет, а цвет



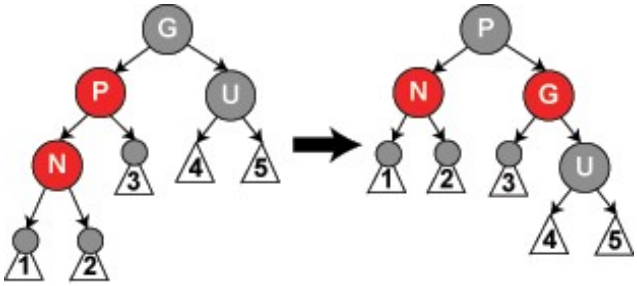
дедушки элемента нужно изменить на красный, после чего необходимо снова вызвать первый метод восстановления баланса, однако уже для дедушки данного узла. Если же дядя элемента был черным, вызывается четвертый метод восстановления баланса.

В четвертом методе проверяется, находятся ли вставленный узел относительно своего родителя и родитель относительно дедушки на разных сторонах. Если это так, будет осуществлен поворот в сторону,



противоположную расположению узла относительно родителя. При этом далее будет рассматриваться не сам вставленный узел, а его потомок, находящийся на противоположной стороне от расположения узла относительно его родителя, так как при повороте поменялись роли родительского и вставленного узлов. Далее вызывается пятый метод восстановления баланса.

В пятом методе родитель рассматриваемого узла перекрашивается в черный цвет, его дедушка перекрашивается в красный цвет, после чего осуществляется



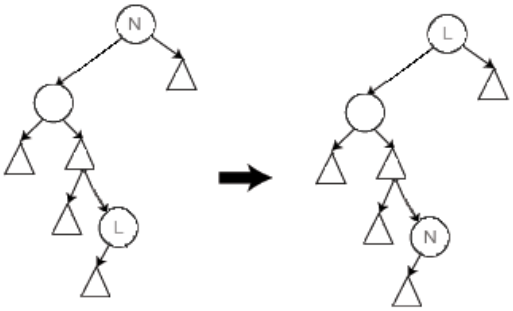
поворот в сторону, противоположную расположению узла относительно его родителя. После данных операций дерево становится сбалансированным.

Поиск места для вставки требует количество операций, равное h, где h — высота дерева. Также после добавления элемента требуется некоторое

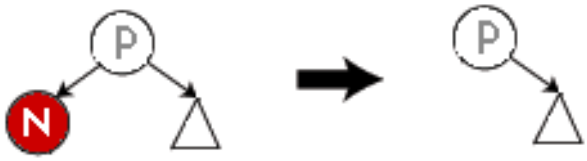
количество процедур перекраски, равное h, а также не более двух процедур поворота. Следовательно, сложность алгоритма вставки O(log n).

Удаление элемента в RB-дереве

При удалении элемента в RB-дереве сперва вызывается его поиск с помощью алгоритма, описанного выше. Если удаляемый элемент был найден, алгоритм продолжится, иначе — процедура удаления будет окончена. Далее при удалении может возникнуть несколько случаев, которые обрабатываются отдельно.



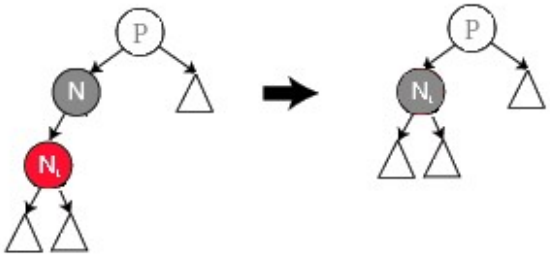
Если у удаляемого узла было два потомка, то вызывается первый метод балансирования дерева после удаления. Выполняется поиск ближайшего по ключу узла. Выполняется это переходом в левое поддерево удаляемого узла и поиском самого правого узла в данном поддереве. После этого ключи у удаляемого и найденного узлов обмениваются. Найденный узел был крайним правым элементом в левом поддереве удаляемого узла, значит его правый потомок нулевой. Далее будет происходить процедура удаления найденного узла.



Второй случай подразумевает, что удаляемый узел имеет красный цвет. Если это так, то он обязан иметь двух нулевых потомков, так как если бы он имел одного потомка, то было бы нарушено свойство про черную высоту, а если двух, то выполнился бы первый метод удаления. Так как потомков у узла нет и он был красного цвета, у родителя указатель на удаляемый узел заменяется на *nullptr*, при этом никакие свойства не будут нарушены.

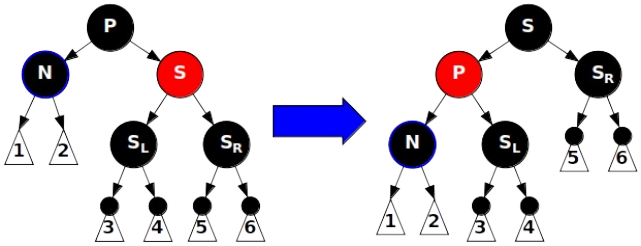
Третий случай балансировки вызывается при условии, что удаляемый узел имеет красного потомка, второй потомок обязан быть нулевым. Условие

про второго потомка выполнится, так как, если бы потомков было двое, был бы вызван первый метод, по результатам которого у рассматриваемого узла становится не более одного потомка. В этом случае ребенок рассматриваемого узла занимает его место (удаляемого узла) и перекрашивается в черный цвет.

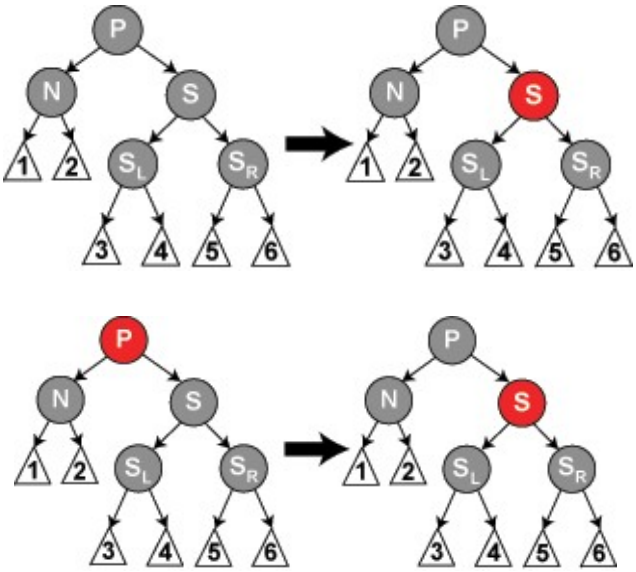


В четвертом методе узел является черным и имеет двух нулевых потомков. При простом его удалении будет нарушено свойство про черную высоту, поэтому необходимо балансировать дерево. Четвертый метод балансировки после удаления в свою очередь разбивается еще на четыре. Вызов проверки всех 4х случаев происходит последовательно.

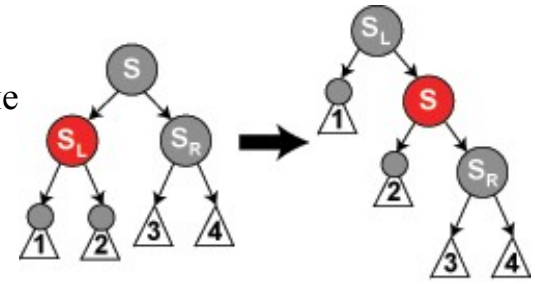
Случай 4.1 отрабатывает, если цвет брата узла красный. Тогда для балансировки осуществляется поворот относительно родителя узла в сторону, совпадающую с расположением узла относительно родителя. После поворота брат окрашивается в черный, а родитель — в красный.



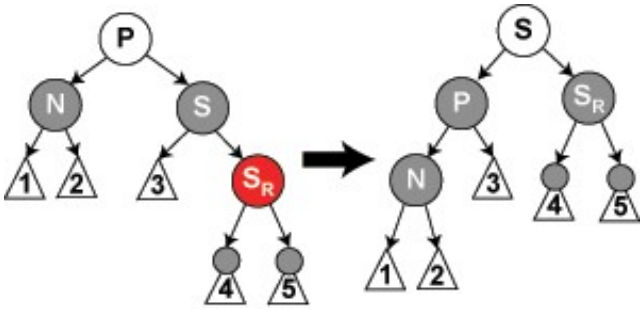
Случай 4.2 отрабатывает, если брат и оба его потомка черные (либо нулевые). Если родитель узла черный, то брат окрашивается в красный, после чего четвертый случай балансировки удаления вызывается уже для родителя узла. Если родитель узла красный, то цвет брата становится красным, а цвет родителя становится черным.



Случай 4.3 отрабатывает, если брат узла является черным, потомок брата, находящийся на стороне, совпадающей с расположением узла относительно родителя, является красным, а второй потомок — черный (либо нулевой). Если данное условие выполнено, то брат окрашивается в красный цвет, его красный ребенок меняет цвет на черный, после чего вызывается поворот относительно брата в сторону, противоположную расположению узла относительно родителя.



Случай 4.4 отрабатывает, если брат узла является черным, а его потомок, находящийся на стороне, противоположной расположению удаляемого узла относительно родителя, красный. Цвет второго потомка значения не имеет. При выполнении данного условия цвет брата меняется на цвет родителя, цвет родителя становится черным, а цвет красного потомка брата меняется на черный. После перекрашивания вызывается метод поворота относительно родителя удаляемого узла в сторону, совпадающую с расположением узла относительно родителя.



Удаление элемента в RB-дереве требует его поиска, что занимает O(log n) времени. При восстановлении свойств лишь в 4 методе имеется рекурсия, с помощью которой осуществляется подъем по дереву и балансировка при необходимости. Так как операции поворотов и перекрашивания выполняются за O(1), восстановление свойств дерева в худшем случае требует O(log n) времени. Итоговая сложность удаления элемента — O(log n).

3. ТЕСТИРОВАНИЕ КОРРЕКТНОСТИ РЕАЛИЗАЦИИ СТРУКТУР ДАННЫХ

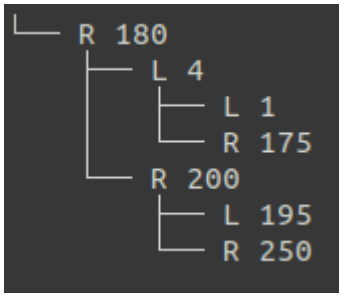
Для тестирования операций с АВЛ-деревом и RB-деревом используется сервис https://www.cs.csubak.edu/~msarr/visualizations/RedBlack.html (для RB-деревьев) и https://www.cs.usfca.edu/~galles/visualization/AVLtree.html (для АВЛ-деревьев). Вывод программы после проведения операций будет сравниваться с результатами этих же операций с этим же деревом на сервисе.

3.1. АВЛ-дерево

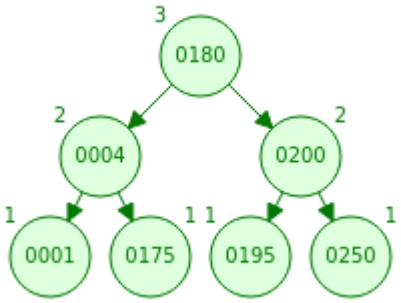
Вставка элементов в АВЛ-дерево

Последовательно в дерево были вставлены ключи 180, 175, 200, 1, 195, 250, 4.

Результат работы программы:



Результат на сервисе:

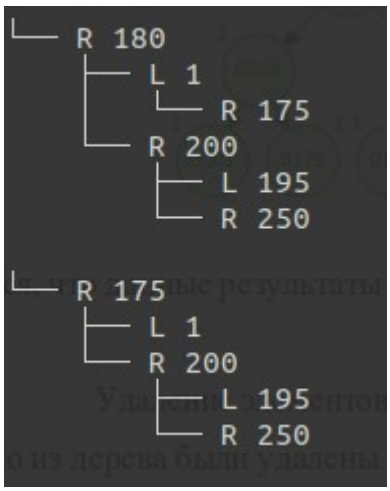


Можно убедиться, что данные результаты совпадают, следовательно программа тест проходит.

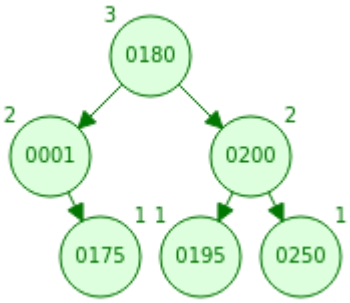
Удаление элементов из АВЛ-дерева

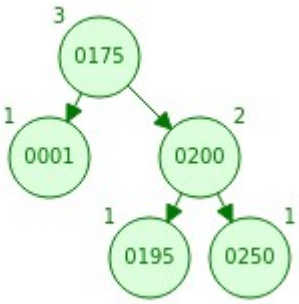
Последовательно из дерева были удалены ключи 4, 180.

Результат работы программы:



Результат на сервисе:





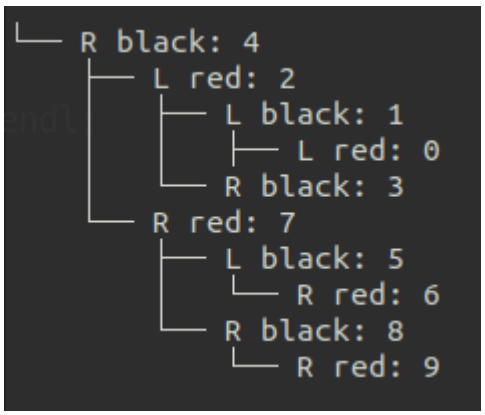
Можно убедиться, что данные результаты совпадают, следовательно программа тест проходит.

3.2. RB-дерево

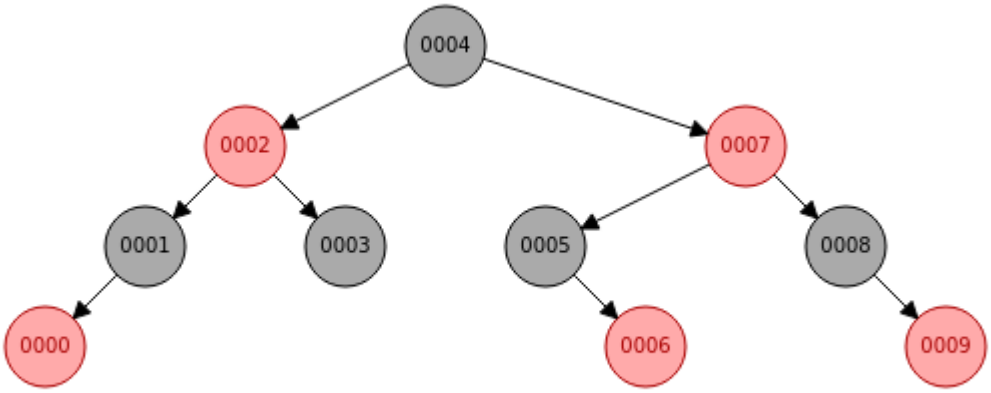
Вставка элементов в RB-дерево

Последовательно в дерево были вставлены ключи 5, 3, 4, 1, 7, 8, 2, 6, 0, 9.

Результат работы программы:



Результат на сервисе:

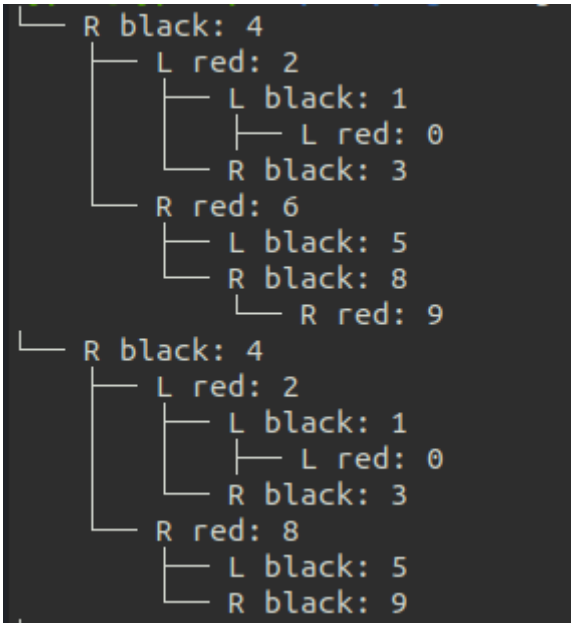


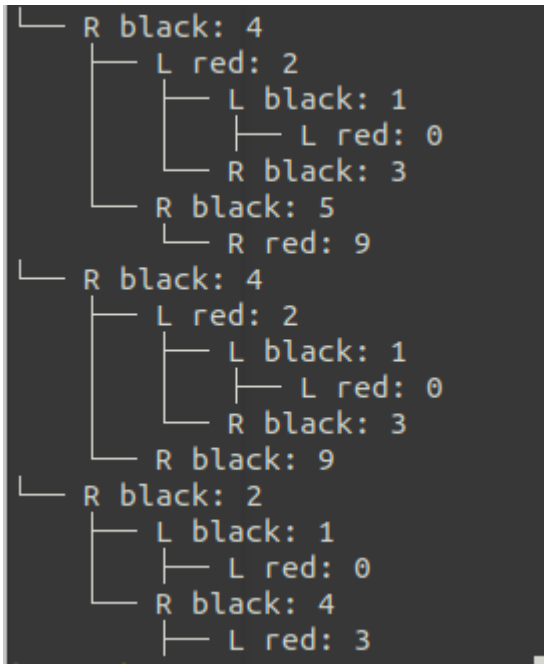
Можно убедиться, что данные результаты совпадают, следовательно программа тест проходит.

Удаление элементов из RB-дерева

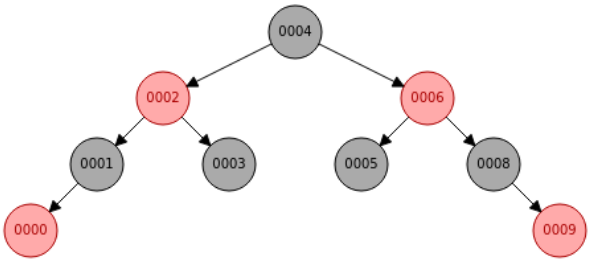
Последовательно из дерева были удалены ключи 7, 6, 8, 5, 9.

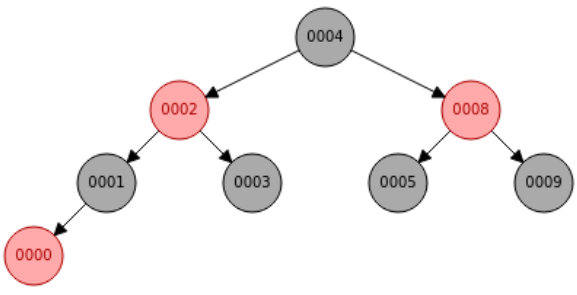
Результат работы программы:

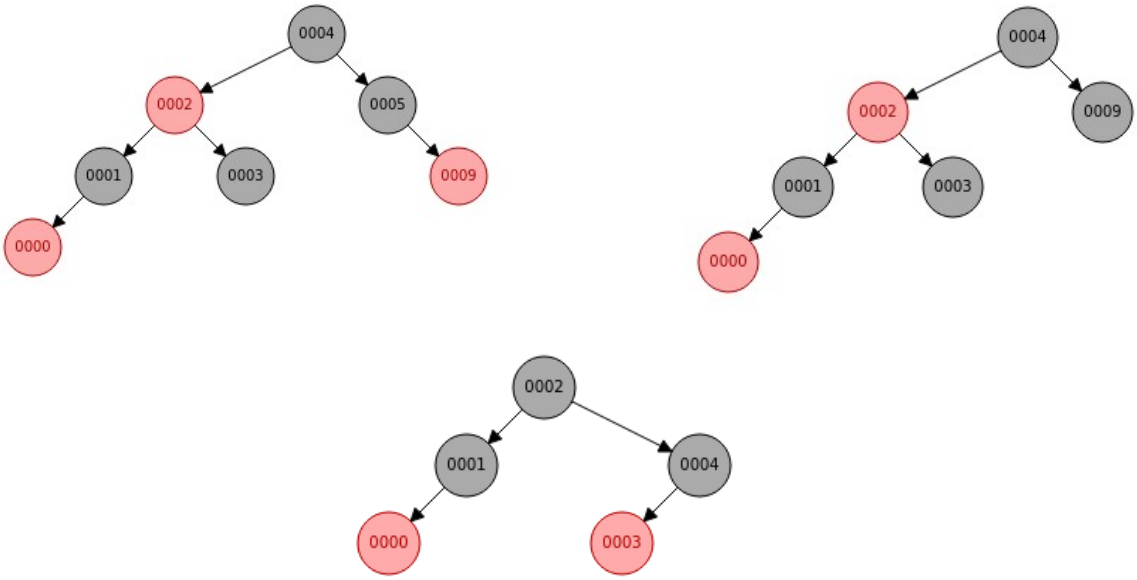




Результат на сервисе:







Можно убедиться, что данные результаты совпадают, следовательно программа тест проходит.

4. ИССЛЕДОВАНИЕ СТРУКТУР ДАННЫХ

Для исследования характеристик производительности операций в АВЛ-деревьях и RB-деревьях был написан файл *benchmark.cpp*, в котором производится генерация данных, которые будут вставляться и удаляться из структур данных, чтобы на их примере можно было оценить практическое среднее время операций. Также были написаны функции, которые занимаются измерением времени, затраченного той или иной структурой данных на определенную операцию.

Для замера времени работы операций вставки и удаления в АВЛ-дереве и RB-дереве с помощью функции был сгенерирован набор случайных уникальных ключей. Данный набор ключей будет последовательно вставляться в оба дерева, после чего элементы из этого же набора, только в другом, случайном, порядке будут удаляться из обоих деревьев. Набор ключей состоит из ~100000 различных элементов, находящихся в диапазоне от -500000 до 500000. Замеры времени операции происходят для каждого тысячного элемента набора, причем замеры производятся 20 раз для минимизации погрешности измерений.

4.1 АВЛ-дерево

Вставка элементов в АВЛ-дерево

Теоретически, операция вставки в АВЛ-дерево выполняется за O(log n) операций, где n — количество элементов в дереве. Во время исследования был проведен замер времени вставки случайного элемента в зависимости от количества элементов в дереве. Результаты см. на рис. 1.

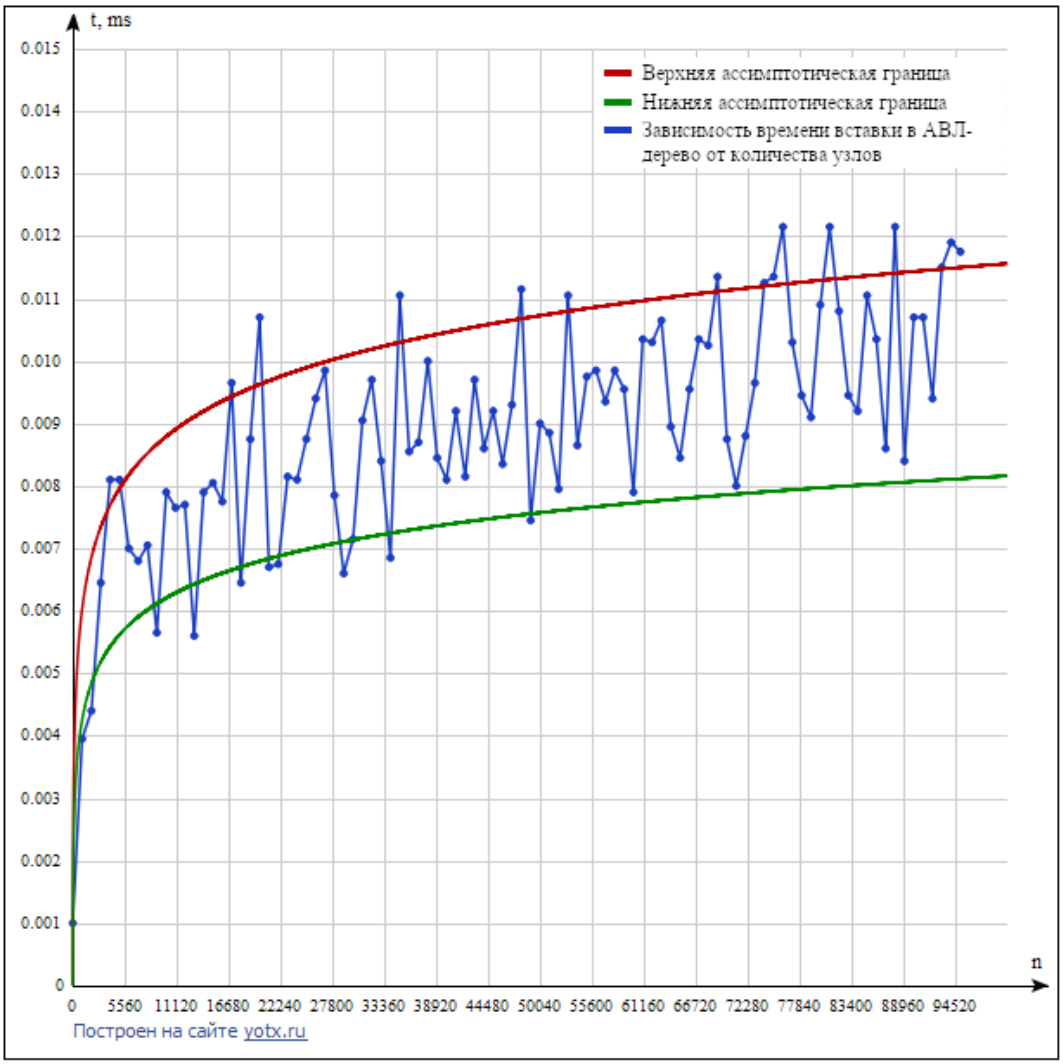


Рисунок 1 — Время работы (в мс) операции вставки в АВЛ-дерево в зависимости от количества вставленных узлов

По данному графику можно проследить логарифмическую зависимость времени вставки от количества элементов в дереве. Практический результат подтверждает теоретическую оценку времени вставки в АВЛ-дерево.

Удаление элементов из АВЛ-дерева

Для проведения исследования времени работы операции удаления из АВЛ-дерева использовалось дерево, полученное после вставки всего набора ключей.

Теоретически, операция удаления из АВЛ-дерева выполняется за O(log n) операций, где n — количество элементов в дереве. Во время исследования был проведен замер времени вставки случайного элемента в зависимости от количества элементов в дереве. Результаты см. на рис. 2.

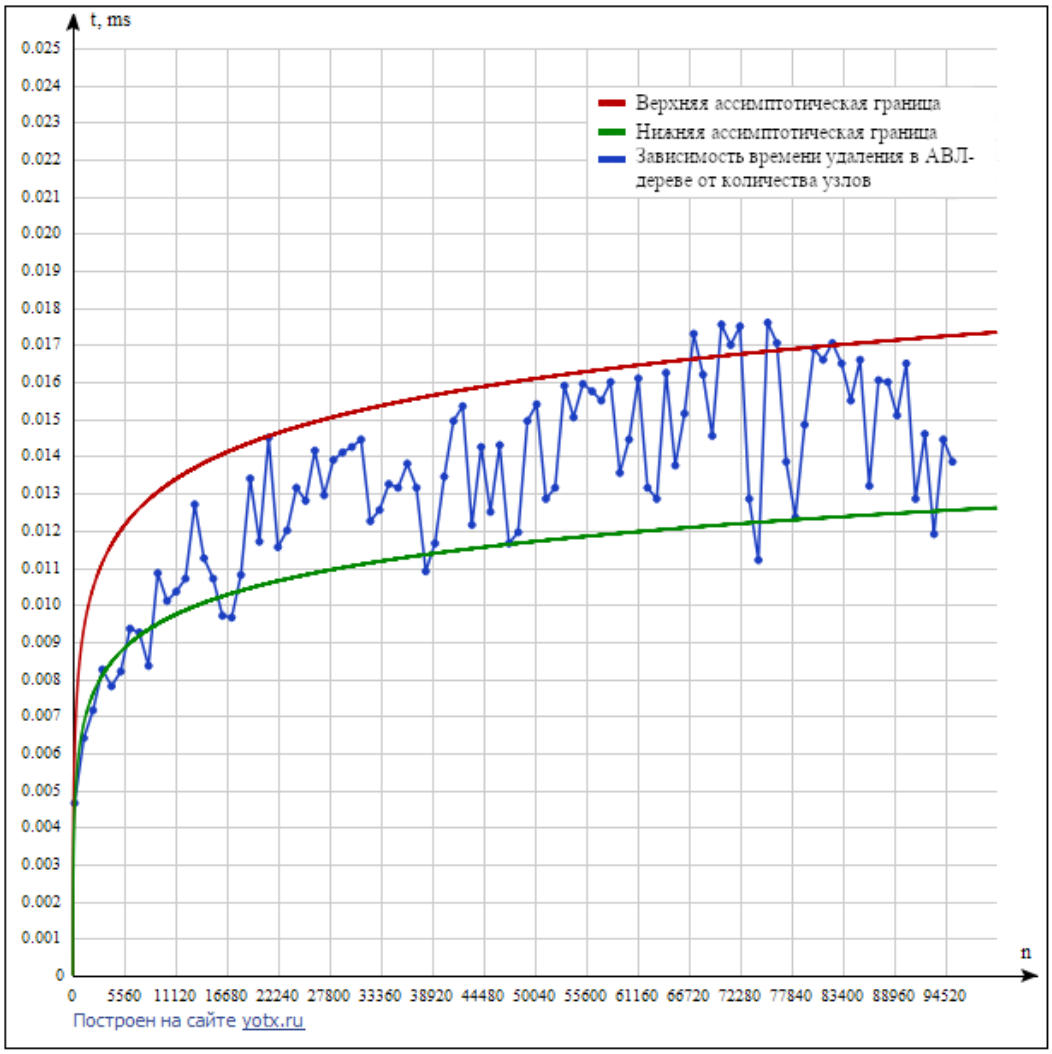


Рисунок 1 — Время работы (в мс) операции удаления из АВЛ-дерева в зависимости от количества вставленных узлов

По данному графику можно проследить логарифмическую зависимость времени удаления от количества элементов в дереве. Практический результат подтверждает теоретическую оценку времени удаления из АВЛ-дерева.

4.2 RB-дерево

Вставка элементов в RB-дерево

Теоретически, операция вставки в RB-дерево выполняется за O(log n) операций, где n — количество элементов в дереве. Во время исследования был проведен замер времени вставки случайного элемента в зависимости от количества элементов в дереве. Набор ключей для вставки совпадает с набором ключей для вставки в АВЛ-дерево. Результаты см. на рис. 3.

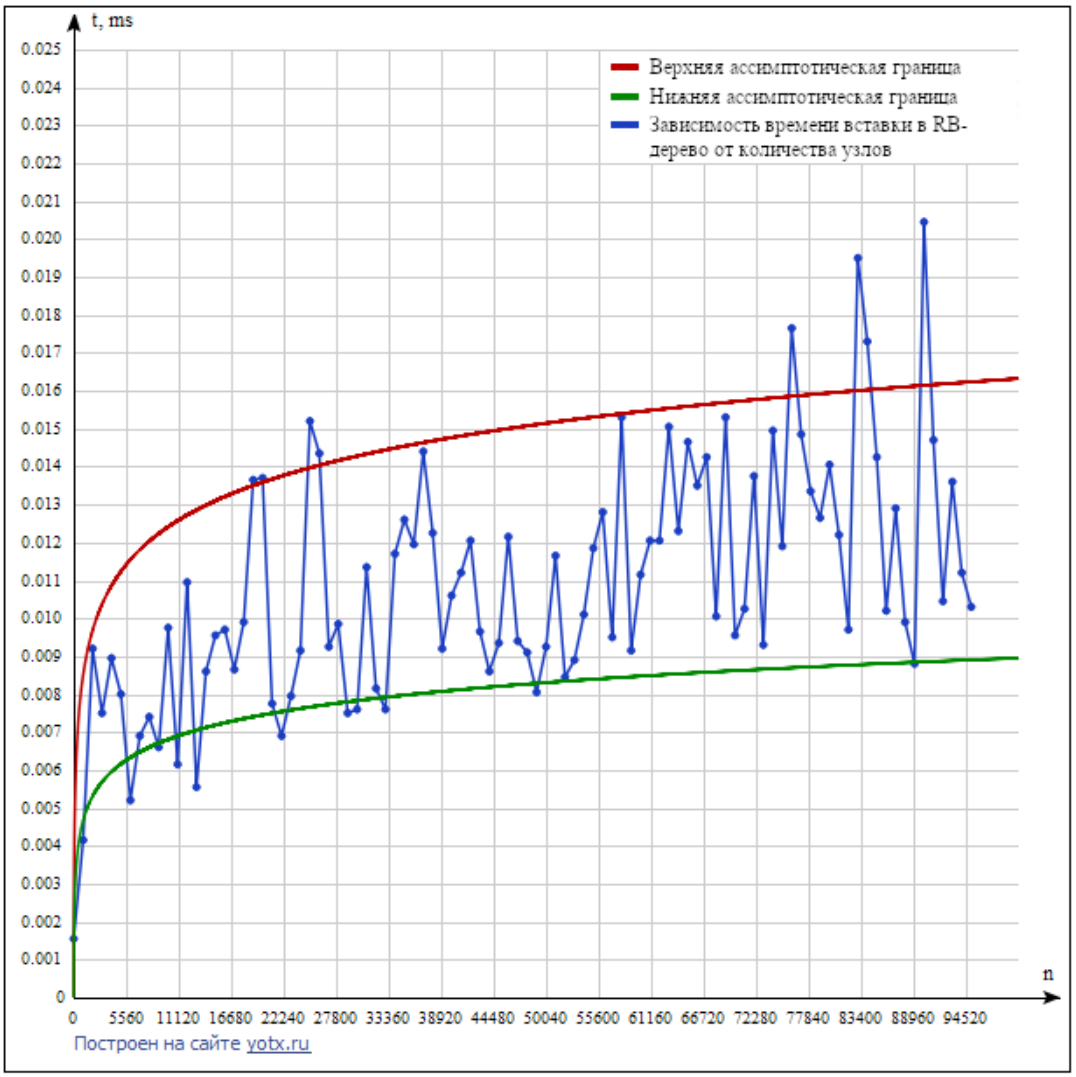


Рисунок 3 — Время работы (в мс) операции вставки в RB-дерево в зависимости от количества вставленных узлов

На данном графике, так же как и на графике вставки в АВЛ-дерево, можно проследить логарифмическую зависимость времени вставки от количества элементов в дереве. Практический результат подтверждает теоретическую оценку времени вставки в RB-дерево.

Удаление элементов из АВЛ-дерева

Для проведения исследования времени работы операции удаления из RB-дерева использовалось дерево, полученное после вставки всего набора ключей.

Теоретически, операция удаления из RB-дерева выполняется за O(log n) операций, где n — количество элементов в дереве. Во время исследования был проведен замер времени вставки случайного элемента в зависимости от количества элементов в дереве. Результаты см. на рис. 4.

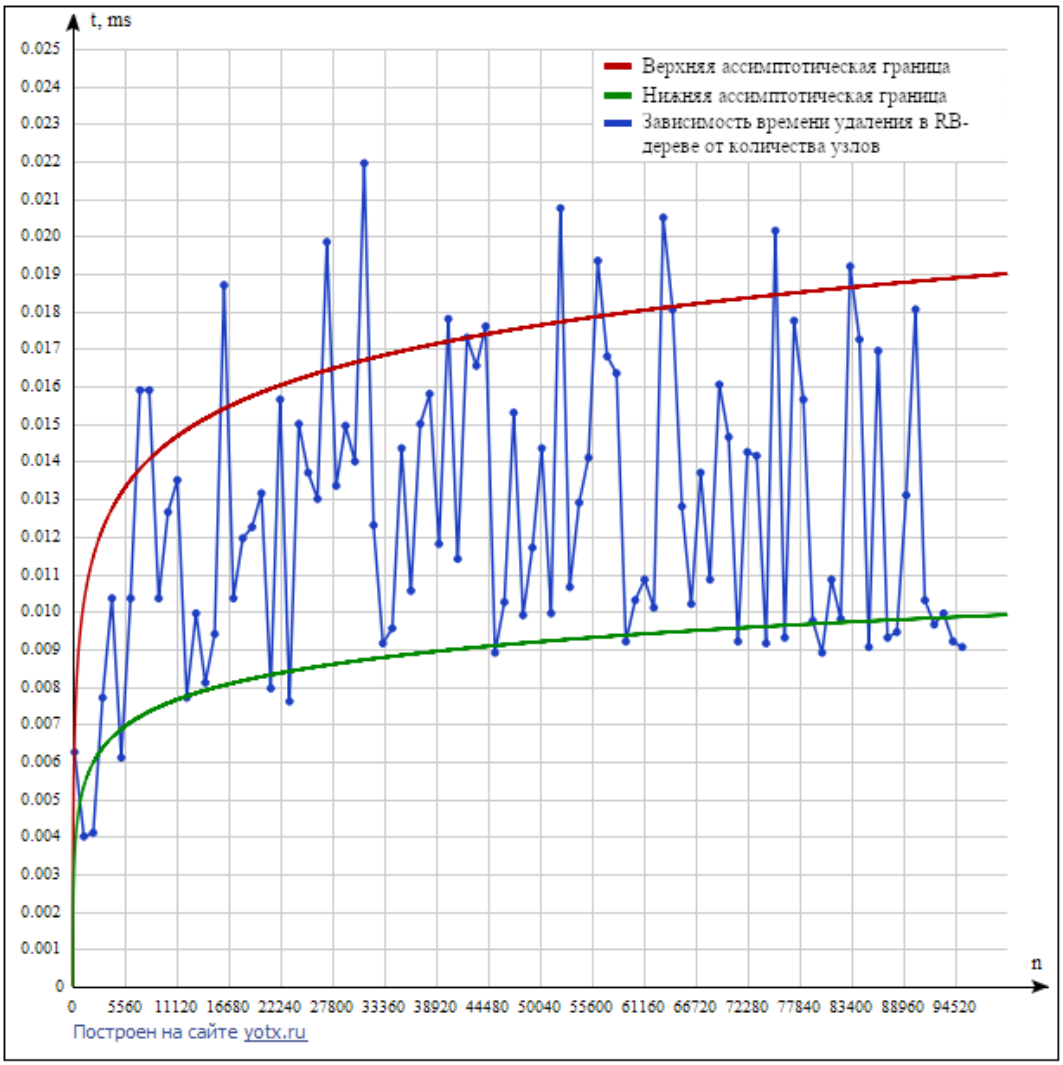


Рисунок 4 — Время работы (в мс) операции удаления из RB-дерева в зависимости от количества вставленных узлов

На данном графике так же можно проследить логарифмическую зависимость времени работы алгоритма удаления от количества узлов. Практический результат подтверждает теоретическую оценку времени удаления из RB-дерева.

4.3. Сравнение структур данных

Было проведено сравнение времени работы алгоритмов вставки и удаления в АВЛ-деревьях и в RB-деревьях. Теоретическая оценка количества операций для выполнения обеих операций в обеих структурах данных совпадает и равна O(log n).

По результатам вставки одних и тех же данных в обе структуры данных и замера времени операции в зависимости от количества элементов были

построены графики времени вставки для АВЛ-дерева (синий график) и времени вставки для RB-дерева (красный график). Результаты см. на рис. 5.

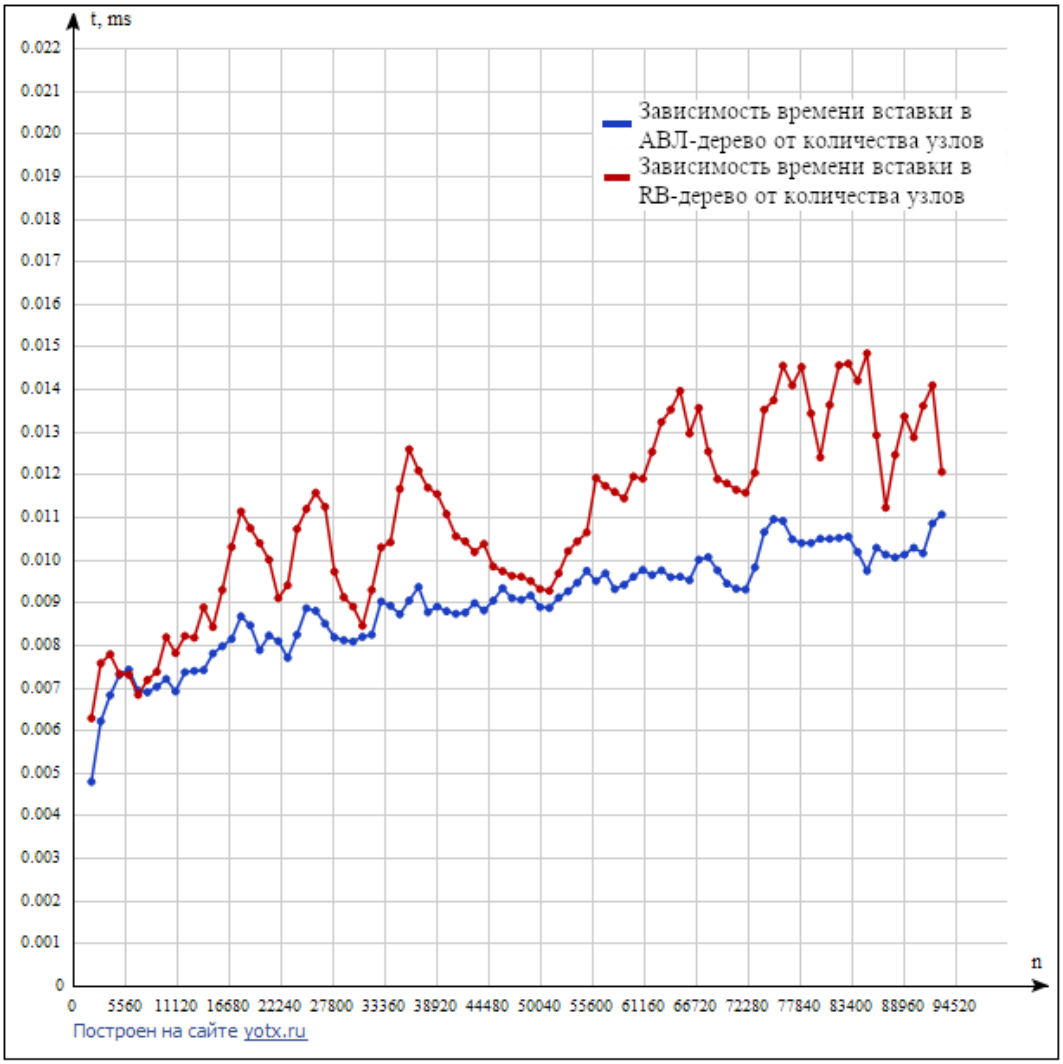


Рисунок 5 — сравнение времени вставки в АВЛ-дерево (синий) и в RB-дерево (красный) в зависимости от количества элементов

По графику видно, что времени на вставку элемента в RB-дерево в среднем затрачивается больше, чем на вставку элемента в AVL-дерево.

По результатам удаления одних и тех же данных из обеих структур данных и замера времени операции в зависимости от количества элементов были построены графики времени удаления для АВЛ-дерева (синий график) и времени удаления для RB-дерева (красный график). Результаты см. на рис. 6.

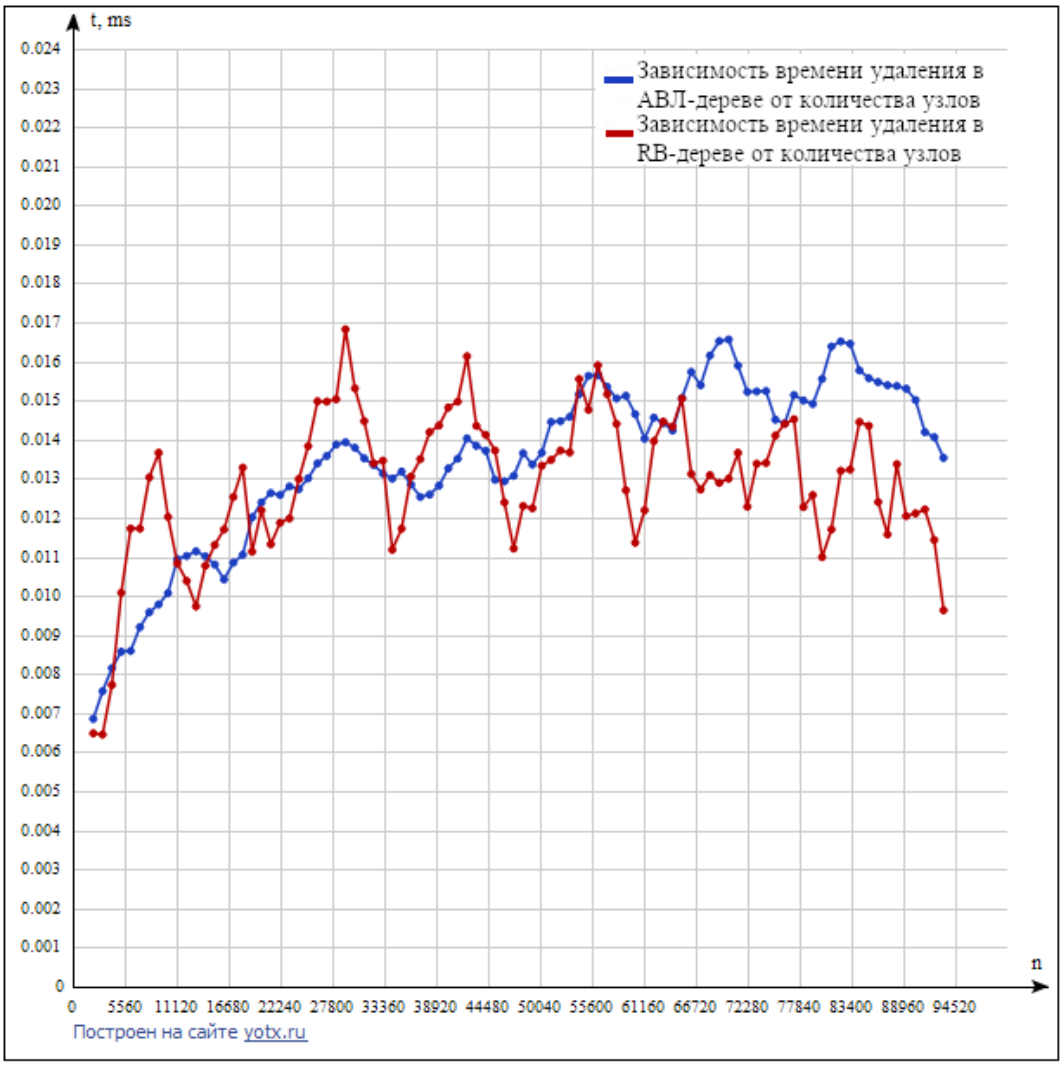


Рисунок 6 — сравнение времени удаления из АВЛ-дерева (синий) и из RB-дерева (красный) в зависимости от количества элементов

По графику видно, что времени на удаление элемента из RB-дерева в среднем затрачивается меньше, чем на удаление элемента из AVL-дерева, хоть и время операции удаления в RB-дереве менее стабильно, чем время операции удаления в АВЛ-дереве.

ЗАКЛЮЧЕНИЕ

В ходе курсовой работы были реализованы на языке C++ такие структуры данных, как АВЛ-дерево и RB-дерево. Была протестирована корректности работы реализаций данных структур данных, а также было исследовано время работы операций вставки и удаления для данных структур. Также было проведено сравнение времени работы данных операций для АВЛ-дерева и RB-дерева.

Исследование подтвердило теоретические оценки количества операций для вставки и удаления элементов в обеих структурах данных — O(log n). По результатам сравнения затраченного времени для вставки элемента в каждую из структур данных было выявлено, что вставка элемента в среднем работает быстрее для АВЛ-дерева, удаление — для RB-дерева.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Онлайн-справочник по языку программированиия C++ // cppreference.com. URL: https://en.cppreference.com/w/

2. Статья об АВЛ-деревьях на Википедии // wikipedia.org. URL: https://ru.wikipedia.org/wiki/АВЛ-дерево

3. Пример реализации АВЛ-дерева на языке Java // javascopes.com. URL: https://javascopes.com/java-avl-trees-d25bb070/

4. Статья о Красно-чёрных деревьях на Википедии // wikipedia.org. URL: https://ru.wikipedia.org/wiki/Красно-чёрное_дерево

5. Пример реализации RB-дерева на языке C++ // coders-hub.com. URL: https://www.coders-hub.com/2015/07/red-black-tree-rb-tree-using-c.html

6. Статья о Красно-чёрных деревьях // russianblogs.com. URL: https://russianblogs.com/article/38531121187/