МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра №806 «Вычислительная математика и программирование»

Курсовая работа

по курсу «Фундаментальные алгоритмы»

Разработка алгоритмов системы хранения и управления данными на основе динамических структур данных

Выполнил: Иванченко М.Д.

Группа: 8О-210Б

Преподаватель: А.М. Романенков

Москва, 2025

Содержание

Введение

Современные вычислительные задачи предъявляют высокие требования не только к скорости алгоритмов, но и к качеству их программной реализации. Разработка высокопроизводительных структур данных и алгоритмов, способных корректно и устойчиво обрабатывать большие объёмы информации — включая длинные целые числа, сложные иерархии узлов и исключительные ситуации — остаётся насущной в криптографии, финансовом анализе, научных расчётах и многих других областях, где точность критична.

В данной работе была поставлена цель: создать на C++20 библиотеку фундаментальных алгоритмов и структур данных, разработанную по SOLID принципам, с гарантированным управлением ресурсами и полной защитой от ошибок времени выполнения.

Полученная библиотека обладает практической ценностью: её можно использовать как готовый строительный блок для сложных вычислительных систем, а также как наглядный пример применения современных принципов проектирования и сопровождения программного кода.

Теоретическая часть

Современные вычислительные системы предъявляют высокие требования к эффективности, надежности и масштабируемости алгоритмов и структур данных. Теоретическая основа таких решений опирается на фундаментальные принципы компьютерных наук, включая оптимизацию операций, управление ресурсами и проектирование абстракций, устойчивых к изменениям. В данной работе рассматривается ряд ключевых задач, направленных на демонстрацию применения классических алгоритмов и паттернов проектирования в контексте языка C++ (стандарт C++20 и выше).

Клиентский логгер

Понятие и назначение

Клиентский логгер — это программный компонент, который непрерывно фиксирует события, происходящие во время работы приложения. Он необходим по четырем основным причинам:

1. Помощь разработчику в воспроизведении ошибок;
2. Эксплуатация система с контролем задержек и пропускной способности;
3. Обеспечение аудита действий пользователей;
4. После критического сбоя именно журналы позволяют восстановить последовательность событий.

Способы логирования

Существуют разные каналы вывода журналов. Самый простой путь — писать сообщения в стандартные потоки stdout и stderr, что удобно на этапе разработки. Для долговременного хранения используют файлы на диске с поддержкой ротации и архивов. В распределённых сервисах сообщения чаще всего отправляются по сети — через syslog, HTTP или gRPC — в централизованное хранилище. Низкоуровневый код, например драйверы или real-time-компоненты, иногда пишет в кольцевой буфер оперативной памяти ради минимальной задержки, а системное ПО может интегрироваться с journald или Windows Event Log. Выбор канала определяется компромиссами между быстродействием, устойчивостью и удобством последующего анализа.

Уровни серьезности

Чтобы не утонуть в потоке сообщений, каждое из них помечают «весомостью». Распространенная шкала такова:

— trace: самые подробные сообщения, полезные лишь разработчику;
— debug: отладочная информация;
— info: нормальное рабочее состояние;
— warning: нештатная, но ещё не фатальная ситуация;
— error: ошибка, после которой система может продолжить работу;
— fatal: критический сбой, требующий немедленного внимания.

Каждому каналу вывода назначают собственный минимальный уровень. Например, файл принимает всё, начиная с trace, а консоль ограничивается warning и выше.

Структурированное логирование

Структурированное логирование — это подход к записи сообщений журнала в виде организованных данных с четко заданной структурой, таких как JSON, XML или других форматов, которые легко парсить и обрабатывать машинными средствами. В отличие от традиционных текстовых логов, которые требуют ручного разбора и сложных регулярных выражений, структурированные логи содержат явно обозначенные поля, такие как временные метки, уровни серьезности, тип события, идентификаторы пользователя или транзакции. Благодаря этому значительно упрощается автоматический поиск и фильтрация событий, а также интеграция с системами мониторинга и аналитики.

Использование структурированных логов позволяет эффективно решать задачи быстрого поиска и диагностики ошибок в распределенных системах, поскольку легко формировать сложные аналитические запросы и агрегировать события по разным критериям. Например, при возникновении сбоя можно сразу отфильтровать логи по конкретному идентификатору запроса или категории ошибки.

Бинарное дерево поиска

Понятие о бинарном дереве поиска, структура

Бинарное дерево поиска — фундаментальная структура данных, изобретенная в 1960 году Эндрю Дональд Бутом. Это структура данных, состоящая из узлов, где каждый узел содержит пару ключ-значение, а также ссылки на два поддерева: левое и правое. Основное свойство бинарного дерева поиска заключается в том, что для каждого узла его левое поддерево содержит только элементы, меньшие, чем ключ самого узла, а правое поддерево — только элементы, большие, чем ключ узла. Такое упорядочивание позволяет эффективно выполнять операции поиска, вставки и удаления.

Операции над бинарным деревом

Операция удаления элемента

Операция поиска в бинарном дереве поиска заключается в том, чтобы пройти по дереву, начиная с корня, и на каждом шаге выбирать одно из поддеревьев в зависимости от того, меньше или больше искомый ключ, чем текущий. Алгоритм поиска работает следующим образом:

1. Начинаем с корня дерева.
2. Если ключ текущего узла равен искомому, то поиск завершён.
3. Если ключ текущего узла больше искомого, продолжаем поиск в левом поддереве.
4. Если ключ текущего узла меньше искомого, продолжаем поиск в правом поддереве.
5. Если поддерево пусто, значит ключ не найден.

Асимптотическая сложность поиска:

- худший случай: $O(n)$ (при вырождении дерева в линейный список)
- средний случай: $O(log\ n)$
- лучший случай: $O(1)$ (искомый ключ - корень)

Операция вставки элемента

Процесс вставки нового элемента в бинарное дерево поиска аналогичен поиску, с тем исключением, что вместо того, чтобы просто искать, мы создаем новый узел и вставляем его в подходящее место в дереве.

1. Начинаем с корня дерева;
2. Если ключ нового элемента меньше ключа текущего узла, то переходим в левое поддерево. Если ключ больше — в правое поддерево;
3. Повторяем шаги 2 и 3, пока не дойдём до пустого поддерева;
4. Вставляем новый элемент в это пустое поддерево.

Асимптотическая сложность поиска:

- худший случай: $O(n)$ (при вырождении дерева в линейный список)
- средний случай: $O(log\ n)$
- лучший случай: $O(n)$ (дерево пустое)

AVL-дерево

Понятие об AVL-дереве, структура

AVL-дерево (от англ. Adelson-Velsky and Landis tree) — это самобалансирующееся бинарное дерево поиска, в котором для каждого узла поддерживается условие баланса, то есть разница между высотами левого и правого поддеревьев любого узла не может быть больше 1. Это позволяет поддерживать логарифмическую высоту дерева, что в свою очередь гарантирует эффективные операции поиска, вставки и удаления.

В структуре узла AVL-дерева, помимо уже указанных выше элементов, также находится высота данного узла. Высота определяется как максимум из высоты правого поддерева и левого поддерева с прибавлением единицы. Высота пустого поддерева равна 0.

Операции в AVL-дереве

Операция поиска

Операция поиска в AVL-дереве аналогична таковому в бинарном дереве поиска.

Асимптотическая сложность:

- худший случай: $O(log\ n)$
- средний случай: $O(log\ n)$
- лучший случай: $O(1)$ (искомый элемент в корне)

Операция вставки

Процесс вставки нового элемента в AVL-дерево аналогичен вставке в обычное бинарное дерево поиска, но после вставки элемента необходимо проверить и, при необходимости, восстановить баланс дерева с помощью поворотов.

1. Вставка элемента происходит как в обычном бинарном дереве поиска: начиная с корня, мы находим подходящее место для нового элемента;
2. Проверяем баланс родителей до корня и при необходимости выполняем балансировку.

Операция балансировки

Пусть a — узел, для которого выполняется балансировка.

Случай 1. Малый левый поворот поворот используется тогда, когда разница высот левого поддерева (поддерево L) и правого поддерева (поддерева b) равна 2 и высота левого поддерева правого поддерева (поддерева C) меньше или равно высоту правого поддерева правого поддерева (поддерева R).

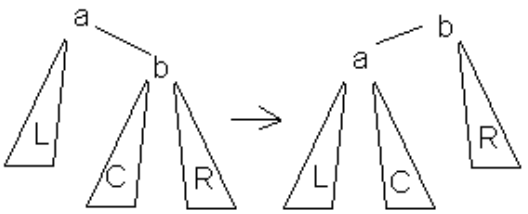

Рисунок 1. Балансировка AVL-дерева малым левым поворотом

Случай 2. Большой левый поворот используется тогда, когда разница высот левого поддерева (поддерева L) и правого поддерева (поддерева b) равна 2 и высота левого поддерева правого поддерева (поддерева C) больше высоты правого поддерева правого поддерева (поддерева R).

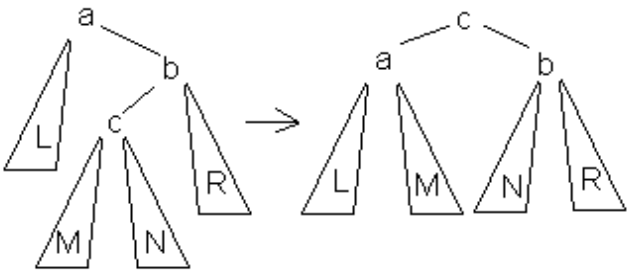

Рисунок 2. Балансировка AVL-дерева большим левым поворотом

Случай 3. Малый правый поворот используется тогда, когда разница высот правого поддерева (поддерева R) и левого поддерева (поддерева b) равна 2 и высота правого поддерева левого поддерева (поддерева C) меньше или равно высоте левого поддерева левого поддерева (поддерева L).

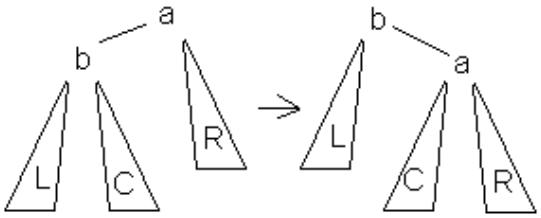

Рисунок 3. Балансировка AVL дерева малым правым поворотом

Случай 4. Большой правый поворот используется тогда, когда разница высот правого поддерева (поддерева R) и левого поддерева (поддерева b) равна 2 и высота правого поддерева левого поддерева (поддерева C) строго больше высоты левого поддерева левого поддерева (поддерева L).

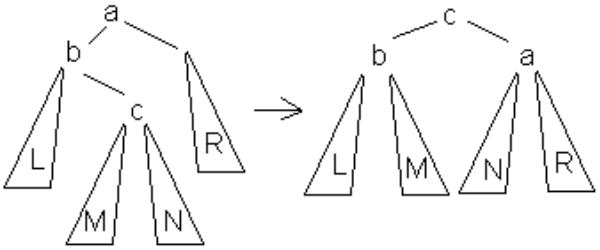

Рисунок 4. Балансировка AVL-дерева большим правым поворотом

Операция удаления из AVL-дерева

1. Сначала удаляем элемент как в обычном бинарном дереве поиска;
2. После удаления проверяем баланс элементов до корня и при необходимости выполняем балансировку.

Splay-дерево

Понятие о splay-дереве

Splay-дерево — это самобалансирующееся бинарное дерево поиска, в котором элементы не всегда находятся в строго сбалансированном состоянии, как в AVL-деревьях или красно-чёрных деревьях. Вместо этого, Splay-дерево использует операцию $Splay$, которая «подтягивает» недавно используемый элемент к корню дерева. Это достигается с помощью серии поворотов, что позволяет улучшить эффективность операций, которые часто обращаются к одному и тому же элементу.

Операции над splay-деревом

Операция $Splay$

Основная задача операции $Splay$ — перемещение узла в корень дерева при помощи серии различных поворотов. Повороты выполняются по следующему алгоритму до тех пор, пока узел не будет перемещен в корень:

1. Если родитель узла является корнем, то выполняется малый поворот относительно его родителя (правый, если узел является левым ребенком и левый, если узел является правым ребенком).
2. Если узел и его родитель оба являются правыми сыновьями или левым сыновьями, то выполняется двойной левый или двойной правый повороты относительно дедушки (родителя родителя).
3. Если узел является правым сыном, а его родитель левым сыном дедушки, то выполняется большой правый поворот относительно дедушки.
4. Если узел является левым сыном, а его родитель правым сыном дедушки, то выполняется большой левый поворот относительно дедушки.

Асимптотическая сложность:

— худший случай: $O(n)$ (если дерево сильно вырождено)
— средний случай: $O(log\ n)$
— лучший случай: $O(1)$ (искомый элемент в корне)

Операция $Merge$ (слияние)

Пусть операция выполняется для деревьев $T1$ и $T2$, в которых все ключи $T1$ меньше ключей в $T2$.

1. Делаем $Splay$ для максимального элемента $T1$, тогда у корня $T1$ не будет правого ребенка.
2. После этого делаем $T2$ правым ребенком $T1$.

Асимптотическая сложность:

— худший случай: $O(n)$ (если дерево сильно вырождено)
— средний случай: $O(log\ n)$
— лучший случай: $O(1)$

Операция $Split$ (разделение)

Разделим дерево по значению x:

1. Найдем наименьший элемент, больший или равный x;
2. Делаем для него $Splay$;
3. Отсоединяем от корня левого ребёнка и возвращаем два получившихся дерева.

Асимптотическая сложность:

— худший случай: $O(n)$ (если дерево сильно вырождено)
— средний случай: $O(log\ n)$
— лучший случай: $O(1)$

Операция поиска

Операция поиска аналогична поиску в бинарном дереве поиска. После нахождения элемента для него выполняется $Splay$. Асимптотическая сложность такая же, как и в бинарном дереве поиска.

Операция добавления элемента

1. Запускаем $Split$ от добавляемого элемента;
2. Подвешиваем получившиеся деревья за элемент к добавлению.

Асимптотическая сложность:

- худший случай: $O(n)$ (если дерево сильно вырождено)
- средний случай: $O(log\ n)$
- лучший случай: $O(1)$

Операция удаления элемента

1. Находим элемент в дереве;
2. Делаем $Splay$ для него;
3. Делаем текущим деревом $Merge$ его детей;
4. Подвешиваем получившиеся деревья за элемент к добавлению.

Асимптотическая сложность:

- худший случай: $O(n)$ (если дерево сильно вырождено)
- средний случай: $O(log\ n)$
- лучший случай: $O(1)$

Красно-чёрное дерево

Понятие о красно-черном дереве, структура

Красно-чёрное дерево (от англ. Red-Black tree) — это самобалансирующееся бинарное дерево поиска, каждый узел которого помечен дополнительным битом для обозначения цвета узла (красный или чёрный) [1]. Эти цвета используются для обеспечения приблизительной сбалансированности дерева, что гарантирует выполнение операций поиска, вставки и удаления за логарифмическое время.

Структура красно-чёрного дерева определяется следующими свойствами:

1. Каждый узел окрашен либо в красный, либо в чёрный цвет.
2. Корень дерева и листья (NIL-узлы, ∅) всегда чёрные.
3. Оба дочерних узла каждого красного узла обязательно черные (запрещено два красных узла подряд).
4. Любой простой путь от произвольного узла до листового узла содержит одинаковое количество черных узлов.

Эти свойства гарантируют, что самый длинный путь от корня до листьев не более чем вдвое длиннее самого короткого пути, что обеспечивает сбалансированность дерева.

Операции в красно-черном дереве

Операция поиска

Операция поиска в красно-черном дереве аналогична поиску в обычном бинарном дереве поиска. Асимптотическая сложность:

- худший случай: $O(log\ n)$
- средний случай: $O(log\ n)$
- лучший случай: $O(1)$

Операция вставки

Вставка нового элемента в красно-чёрное дерево начинается с обычной процедуры вставки в бинарное дерево поиска. Новый узел z всегда окрашивается в красный. Затем выполняется процедура восстановления свойств, которая рассматривает три возможные конфигурации (для каждой есть зеркальный вариант «влево/вправо»).

Случай 1. Родитель z и дядя z (“брат отца”) красные:

Перекрашиваем родителя и дядю в чёрный, дедушку — в красный; затем продолжить проверку, поднявшись к дедушке. Чёрная высота поддеревьев выравнивается, но красный дедушка может нарушать свойство 4, поэтому цикл продолжается.

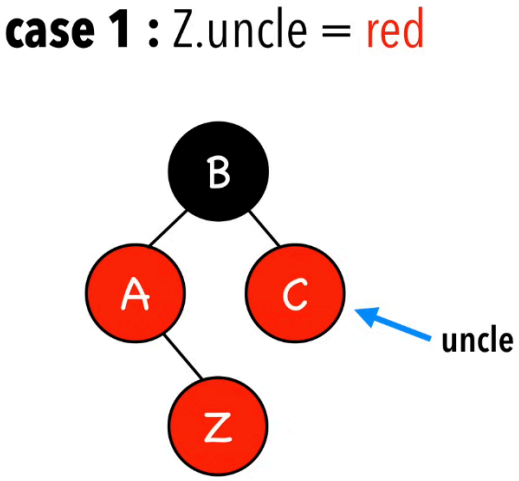

Рисунок 5. Ситуация 1 до восстановления свойств КЧД

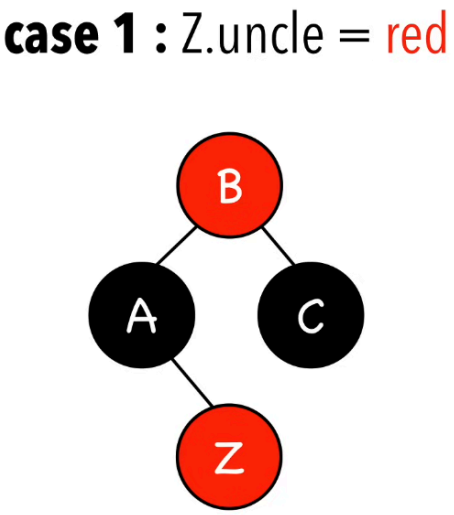

Рисунок 6. Ситуация 1 после восстановления свойств КЧД

(корень не покрашен в чёрный, потому что это поддерево)

Случай 2. Дядя z чёрный (или ∅), а z образует “треугольник” с родителем (например, родитель — левый ребёнок дедушки, а z — правый ребёнок родителя):

Выполняем малый поворот (левый либо правый) вокруг родителя. После поворота z и его родитель меняются местами, что приводит к случаю 3.

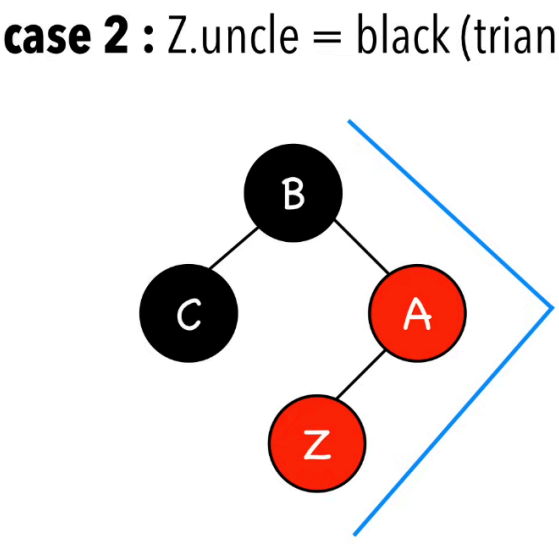

Рисунок 7. Ситуация 2 до восстановления свойств КЧД

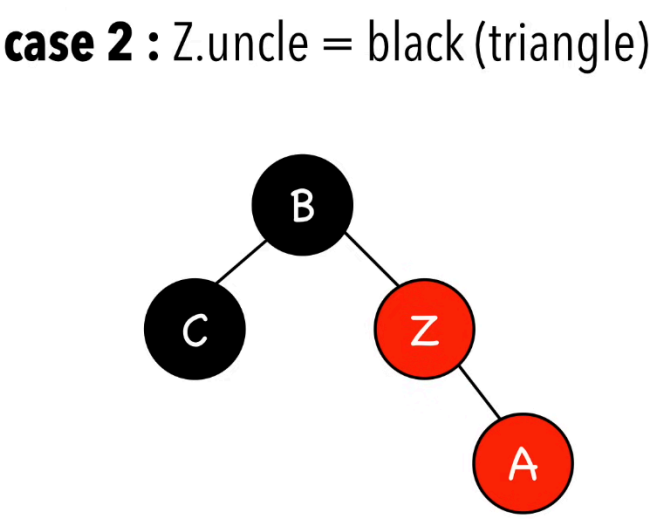

Рисунок 8. Ситуация 2 после восстановления свойств КЧД

(выполнен правый поворот относительно A)

Случай 3. Дядя z чёрный, а z и его родитель лежат на одной прямой (оба левые или оба правые потомки:

Перекрашиваем родителя z в чёрный, дедушку — в красный; затем выполняем малый поворот вокруг дедушки, направленный к z. В результате все свойства красно-чёрного дерева восстановлены, цикл завершается.

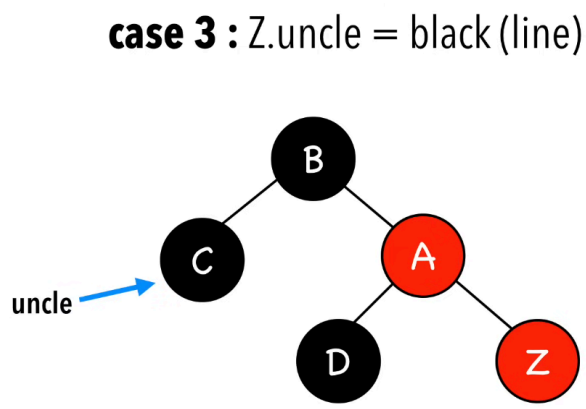

Рисунок 9. Ситуация 3 до восстановления свойств КЧД

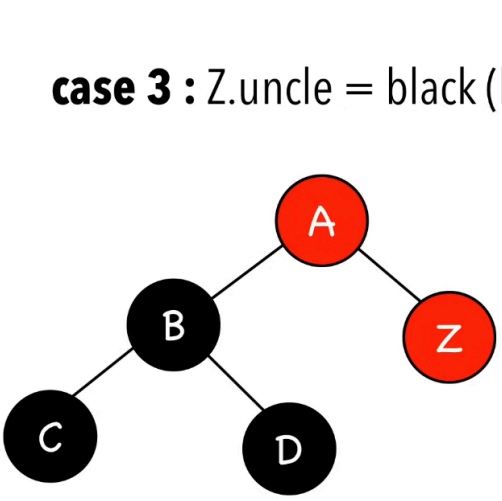

Рисунок 10. Ситуация 3 после левого поворота вокруг дедушки z

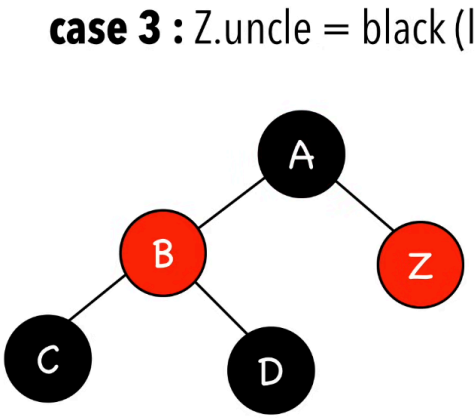

Рисунок 11. Ситуация 3 после восстановления свойств КЧД

Если в начале цикла z оказался корнем, то после выхода из цикла корень перекрашивается в черный, что гарантирует выполнение свойства 2.

Операция удаления

Удаляем узел так же, как в обычном дереве поиска; после фактического удаления вызывается процедура восстановления для узла x, который занял освободившееся место. Она заключается в следующем:

Если удаленный (или перемещенный) узел был чёрным, у x может появиться «двойная чернота». Алгоритм поднимает ее вверх, пока не восстановятся свойства дерева. Существуют четыре конфигурации (у каждой есть зеркальное отражение).

Случай 1. Брат w узла x красный:

Перекрашиваем w в чёрный, родителя x — в красный. Выполняем поворот вокруг родителя в сторону x. После этого новый брат становится чёрным, и алгоритм переходит к случаям 2-4.

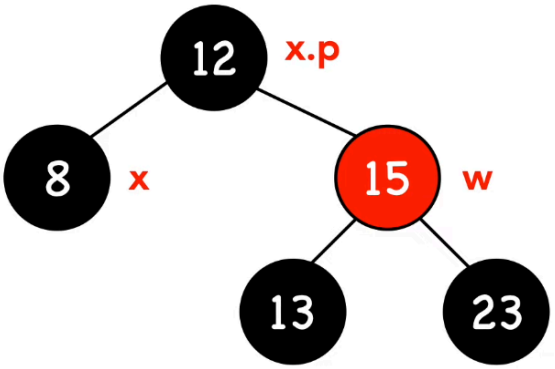

Рисунок 12. Ситуация 1 до восстановления свойств КЧД

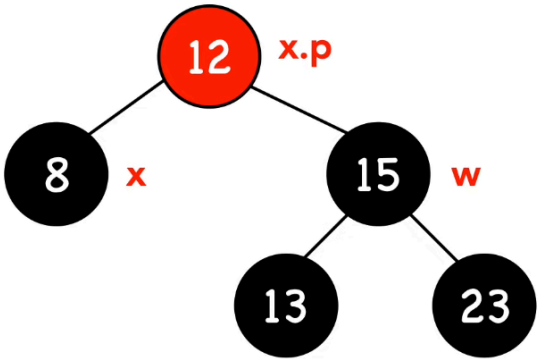

Рисунок 13. Узлы w и родитель x перекрашены

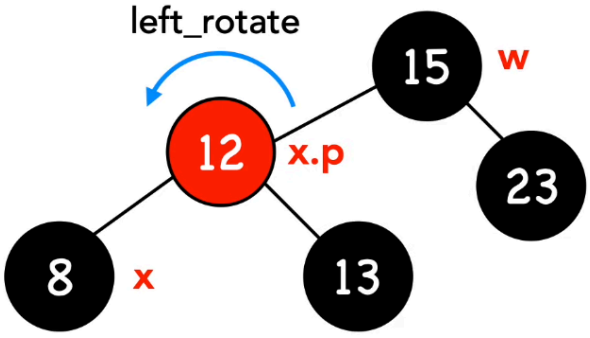

Рисунок 14. Выполнен левый поворот относительно родителя x

Случай 2. Брат w черный, и оба его потомка черные:

Перекрашиваем w в красный; затем поднимаем «двойную черноту» на родителя ($x \leftarrow x_p$) и повторяем цикл.

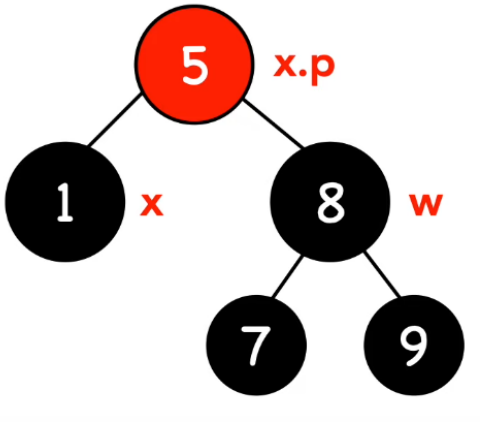

Рисунок 15. Ситуация 2 до восстановления свойств КЧД

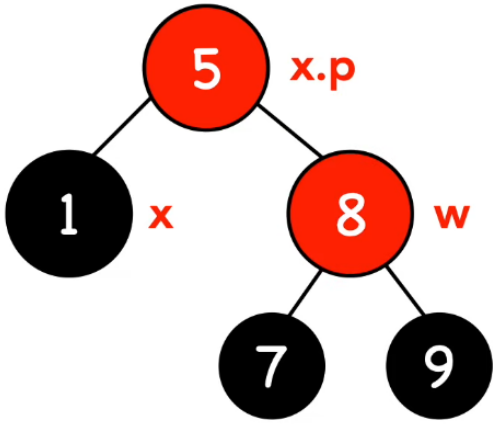

Рисунок 16. Перекрасили брата x, переходим к родителю

Случай 3. Брат w черный, его «дальний» потомок чёрный, а «ближний» — красный (например, x — левый ребёнок; тогда ближний — левый потомок w, дальний — правый:

Перекрашиваем ближнего красного потомка в чёрный, w — в красный. Выполняем поворот вокруг w в сторону, противоположную x. После поворота получаем случай 4.

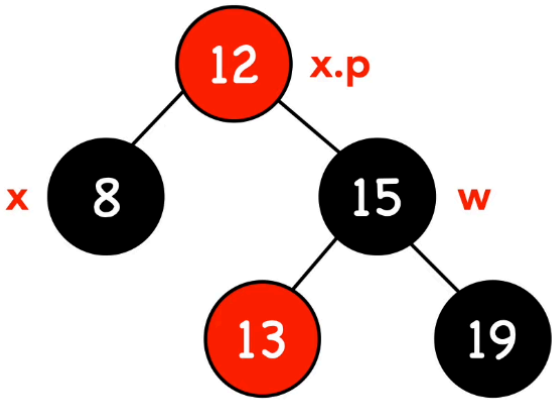

Рисунок 17. Ситуация 3 до восстановления свойств КЧД

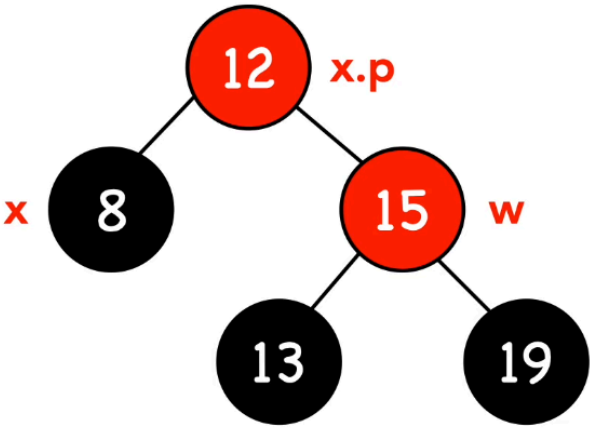

Рисунок 18. Перекрасили левое поддерево брата x и его самого

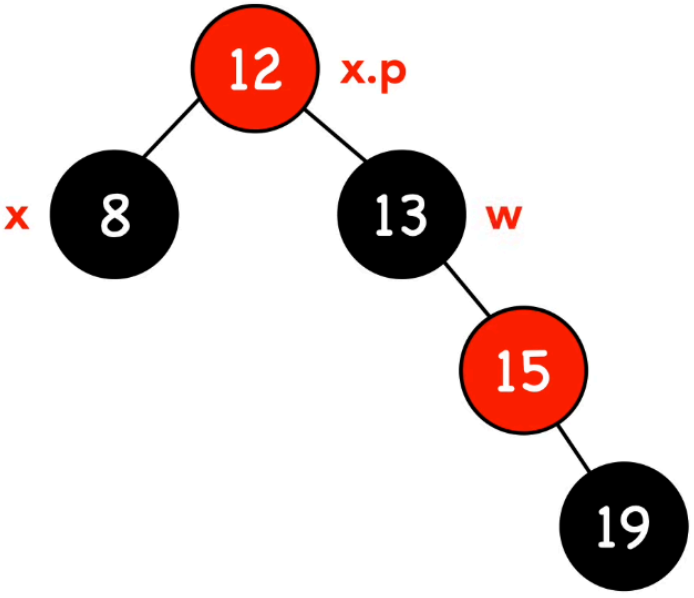

Рисунок 19. Выполнили правый поворот относительно w и перешли к родителю

Случай 4. Брат w чёрный, его «дальний» потомок красный:

Присваиваем $w_c = x_{p_c}$; родителя x_p перекрашиваем в чёрный, дальнего красного потомка w — в чёрный. Выполняем поворот вокруг родителя в сторону x. Затем делаем x корнем (что снимает «двойную черноту») и выходим из цикла.

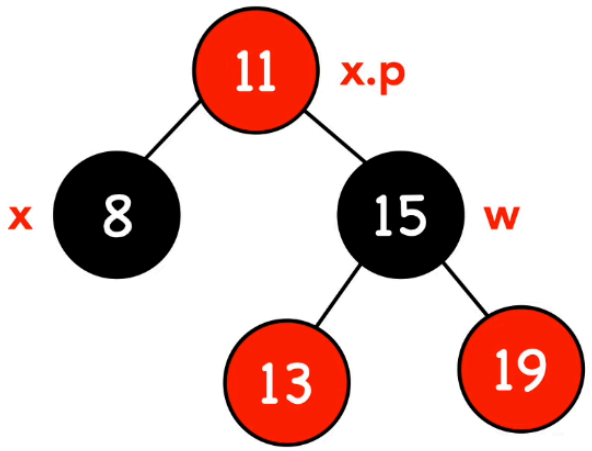

Рисунок 20. Ситуация 4 до исправления свойств КЧД

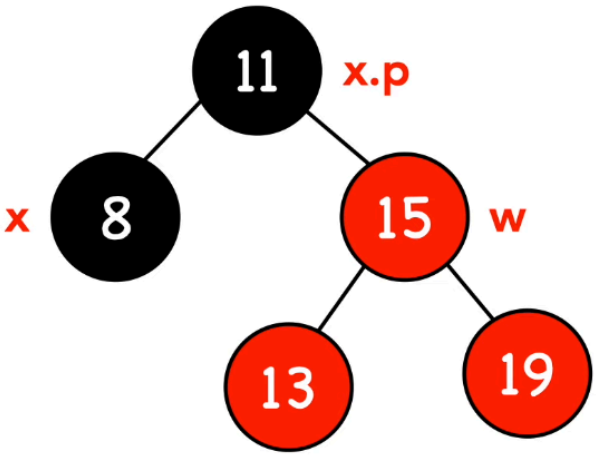

Рисунок 21. Перекрасили узлы

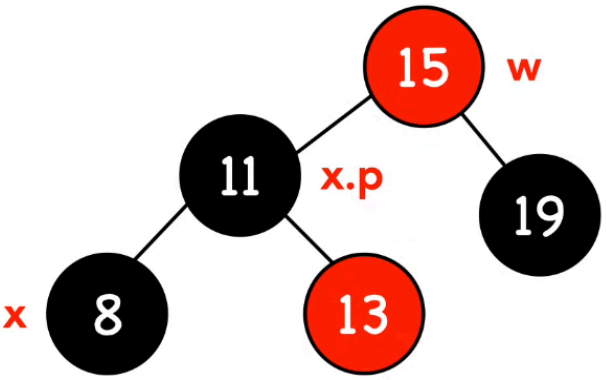

Рисунок 22. Выполнили левый поворот относительно x_p

По завершении цикла узел x перекрашивается в черный, окончательно восстанавливая все свойства дерева.

Балансировка гарантирует сохранение логарифмической высоты дерева.

Асимптотическая сложность операций вставки и удаления:

- худший случай: $O(log\ n)$
- средний случай: $O(log\ n)$
- лучший случай: $O(log\ n)$

Таким образом, красно-чёрные деревья — это мощный инструмент, позволяющий эффективно выполнять основные операции с набором данных за гарантированное логарифмическое время, что делает их чрезвычайно полезными в практических приложениях, где требуется быстрый поиск и модификация данных.

Класс длинного целого числа

Сложения длинных целых чисел

1. Оба числа приводятся к одному знаку и одинаковой длине;
2. Цифры складываются справа налево (от младших к старшим) с учетом переноса;
3. Если сумма цифр превышает основание системы счисления излишек сохраняется как перенос для следующего разряда;
4. После обработки всех разрядов оставшийся перенос добавляется в старший разряд.

Вычитание длинных целых чисел

1. Если знаки чисел разные, вычитание заменяется сложением с инвертированным знаком второго числа;
2. Оба числа преобразуются к положительным для упрощения логики;
3. Если уменьшаемое меньше вычитаемого, их меняют местами, а результат получает отрицательный знак;
4. Обработка цифр справа налево с учетом займа;
5. Если текущая цифра уменьшаемого меньше цифры вычитаемого, занимается единица из старшего разряда;
6. Удаление ведущих нулей и установка корректного знака.

Сдвиг влево для длинных целых чисел

1. Свиг разбивается на две части: сдвиг целых цифр и сдвиг внутри цифр;
2. В начало массива цифр добавляется количество нулевых блоков, равное сдвигу целых цифр;
3. Каждый блок сдвигается влево на сдвиг внутри цифр битов;
4. Переполнение (старшие биты) переносится в следующий блок;
5. Если после сдвига остается перенос, он добавляется как новый старший блок.

Отношения порядка на множестве длинных целых чисел

1. Сравнить знаки чисел, если они отличаются, то больше положительное;
2. Если знаки одинаковые, число с большим количеством цифр больше (для положительных) или меньше (для отрицательных);
3. Если длины одинаковые, цифры сравниваются начиная со старшего разряда. Для положительных чисел: первая цифра, которая больше, определяет большее число. Для отрицательных чисел: первая цифра, которая меньше, определяет меньшее число.

Вставка в поток в системе счисления с основанием 10

1. Если число равно нулю, возвращается строка "0";
2. Переводим число в десятичную систему счисления и запоминаем в строку;
3. Если исходное число отрицательное, в конец строки добавляется символ “–”;
4. Поскольку цифры собирались от младшего разряда к старшему, строка переворачивается;
5. Отправляем строку в поток.

Умножение длинных целых чисел согласно алгоритму умножения чисел в столбик

1. Создается временная структура для хранения промежуточных значений, размер которой равен сумме количества цифр в множимом и множителе. Все элементы инициализируются нулями;
2. Для каждой цифры множителя, начиная с самой младшей:

a. Текущая цифра множителя соответствует определенному сдвигу в результирующей структуре (чем старше цифра множителя, тем больше сдвиг);
b. Для каждой цифры множимого:
    1. Умножаем текущую цифру множимого на текущую цифру множителя;
    2. Добавляем к полученному произведению значение, уже находящееся в соответствующей позиции результата, и перенос с предыдущего шага;
    3. Вычисляем остаток от деления этой суммы на основание системы счисления. Этот остаток записывается в текущую позицию результата;
    4. Целая часть от деления становится новым переносом.
c. Добавление переноса: После обработки всех цифр множимого оставшийся перенос записывается в следующую позицию результата.

3. Финальная обработка:
    a. Исключаются нули в старших разрядах результата, чтобы число не содержало лишних символов;
    b. Если исходные числа имеют разные знаки, результат становится отрицательным.

Сложность этого алгоритма $O(n^2)$, где n – максимальная из двух длин перемножаемых чисел.

Умножения длинных целых чисел согласно алгоритму Карацубы умножения чисел

Алгоритм Карацубы — это метод умножения больших чисел, основанный на стратегии «разделяй и властвуй». Вычислительная сложность алгоритма $O(n^{log_2 3})$. Алгоритм рекурсивный, поэтому должен быть базовый случай, при котором мы будем использовать другой алгоритм умножения.

Шаги алгоритма:

1. Если длина чисел меньше порогового значения для базового случая, то используем алгоритм умножения в столбик;
2. Множители X и Y разбиваются на 2 части:
    a. $X = A * 10^k + B$;

b. $Y = C * 10^{k} + D$, где $k = [n / 2]$, n – длина числа.

3. Вычисляются 3 произведения, каждое из произведений вычисляется рекурсивно этим же способом:
   a. $P_1 = A * C$;
   b. $P_2 = B * D$;
   c. $P_3 = (A + B) * (C + D)$.
4. Вычисляется промежуточная сумма: $P_4 = P_3 - P_1 - P_2$;
5. Итоговое произведение: $X * Y = P_1 * 10^{2k} + P_4 * 10^{k} + P_2$.

Умножения длинных целых чисел согласно алгоритму Шёнхаге-Штрассена умножения чисел

Многочлен степени $(n - 1)$ можно однозначно задать не только своими коэффициентами, но и значениями в n различных точках.

При прямом перемножении многочленов, заданных своими коэффициентами, нужно потратить $O(n^2)$ операций. Но если многочлены заданы своими значениями в $2n$ различных точках, то их можно перемножить за $O(n)$: значение многочлена-произведения $A(x) * B(x)$ в точке x_i просто становится равным $A(x_i) * B(x_i)$.

Основная идея алгоритма заключается в том, что если мы посчитаем значения в каких-то различных $(n + m)$ точках для обоих многочленов A и B то, попарно перемножив их, мы за $O(n + M)$ операций можем получить значения в тех же точках для многочлена $A(x) * B(x)$, и с их помощью интерполяцией получить коэффициенты многочлена-произведения и решить задачу.

Для любого натурального n есть ровно n комплексных «корней из единицы», то есть чисел ω_k, таких что:

$$\omega_k^{\,n} = 1$$

Дискретным преобразованием Фурье собственно и называется вычисление значений многочлена в комплексных корнях из единицы:

$$y_j = \sum_{k=0}^{n-1} x_k \omega_1^{jk}$$

Обратным дискретным преобразованием Фурье называется, обратная операция — интерполяция коэффициентов x_i по значениям y_j.

Обратное ДПФ можно вычислить по формуле:

$$x_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k \omega_{n-1}^{jk}$$

Схема Кули-Тьюки

Представим многочлен в виде $P(x) = A(x^2) + x * B(x^2)$, где $A(x)$ состоит из коэффициентов при четных степенях x, а $B(x)$ — из коэффициентов при нечетных.

Пусть $n = 2 * k$, тогда заметим, что для любого целого числа t выполняется:

$$\omega^{2t} = \omega^{2(t \bmod k)}$$

Зная это, исходную формулу для значения многочлена в точке ω^t можно записать так:

$$P(\omega^t) = A(\omega^{2t}) + \omega^t * B(\omega^{2t}) = A(\omega^{2(t \bmod k)}) + \omega^t * B(\omega^{2(t \bmod k)})$$

Различных корней вида ω^{2t}, значения в которых нам потребуются для пересчета, будет в два раза меньше, а также в обоих многочленах будет в два раза меньших коэффициентов — значит, мы только что успешно разбили нашу задачу на две, каждая из которых в два раза меньше.

При использовании комплексных чисел в алгоритме мы встретимся с потерей точности. Нам от комплексных чисел на самом деле нужно было только одно свойство: что у единицы есть n «корней». На самом деле, помимо комплексных чисел, есть и другие алгебраические объекты, обладающие таким свойством — например, элементы кольца вычетов по модулю.

Для вычисления произведения двух чисел согласно алгоритму Шёнхаге-Штрассена без потери точности, нужно использовать Number-theoretic transform, которое реализует FTT в модулярной арифметике.

Итоговый алгоритм для перемножения полиномов A и B:

1. $NA = NTT(A)$;

2. $NB = NTT(B)$;
3. Почленно перемножить элементы NA и NB;
4. Применяем к результату INTT.

Целочисленное деление длинных целых чисел согласно алгоритму деления чисел в столбик

1. Проверка тривиальных случаев:
    a. Если делитель равен нулю → ошибка (деление на ноль);
    b. Если делимое равно нулю → результат 0;
    c. Если делимое по модулю меньше делителя → результат 0.
2. Итеративное определение цифр частного:
    a. Для каждого старшего разряда делимого:
        1. К текущему остатку «переносится» очередной разряд делимого;
        2. Находится максимальная цифра q, такая что: q * делитель ≤ текущий остаток;
        3. Из остатка вычитается значение (q * делитель);
        4. Цифра q добавляется в соответствующий разряд результата.
3. Оптимизация результата:
    a. Удаление ведущих нулей из частного;
    b. Если остаток отрицательный → корректировка частного и остатка.

Для каждой итерации цикла для поиска q используется алгоритм бинарного поиска для оптимизации.

Вычислительная сложность: $O(n^2 \log_2 BASE)$, так как сам алгоритм работает за $O(n \log_2 BASE)$ и вычитание внутри длинного целого работает за $O(n)$.

Класс дробь

На основе класса длинное целое, можно реализовать класс дробь, с помощью которого можно вычислять тригонометрические, и дробные функции.

Арифметические операции

Все арифметические операции реализуются по правилам работы с дробями, а также после каждой операции необходимо вызвать функцию, которая будет сокращать дробь.

Сложение:

$$\frac{a}{b} + \frac{c}{d} = \frac{a * d + c * b}{b * d}$$

Вычитание:

$$\frac{a}{b} - \frac{c}{d} = \frac{a * d - c * b}{b * d}$$

Умножение:

$$\frac{a}{b} * \frac{c}{d} = \frac{a * \mathrm{c}}{b * d}$$

Деление:

$$\frac{a}{b} \div \frac{c}{d} = \frac{a * d}{b * c}$$

Тригонометрические функции

Тригонометрические функции вычисляются через разложение в ряд Тейлора или через связь с другими функциями.

Синус:

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ \ldots$$

Арксинус:

$$arcsin(x) = x + (\frac{1}{2})\frac{x^3}{3} + (\frac{1*3}{2*4})\frac{z^7}{7} + \ \ldots$$

Косинус:

$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ \ldots$$

Арккосинус:

$$arccos(x) = \frac{\pi}{2} - arcsin(x)$$

Тангенс:

$$tg(x) = \frac{sin(x)}{cos(x)}$$

Арктангенс:

Если $|x| \leq 1$:

$$arctg(x) = x - \frac{x^3}{3} + \frac{x^5}{5} + \ ...$$

Если $|x| \geq 1$:

$$arctg(x) = \frac{\pi}{2} - arctg(\frac{1}{x})$$

Котангенс:

$$ctg(x) = \frac{cos(x)}{sin(x)}$$

Арккотангенс:

$$arcctg(x) = \frac{\pi}{2} - arctg(x)$$

Секанс:

$$sec(x) = \frac{1}{cos(x)}$$

Арксеканс:

$$arcsec(x) = arccos(\frac{1}{x})$$

Косеканс:

$$cosec(x) = \frac{1}{sin(x)}$$

Арккосеканс:

$$arccosec(x) = arcsin(\frac{1}{x})$$

Возведение в степень

Используем алгоритм быстрого (бинарного) возведения в степень.

Вычисления корня натуральной степени из дроби

Численный метод для нахождения корня уравнения, который является модификацией метода Ньютона. Он также известен как метод касательных второго порядка. Вычисления происходят итеративно, пока допустимая точность не будет достигнута:

$$x_{k+1} = \frac{(n-1)*x_k + \frac{A}{x_k^{n-1}}}{n}$$

Логарифмы

Натуральный логарифм:

Используется разложение для $ln(\frac{1+y}{1-y})$, где $y = \frac{x-1}{x+1}$:

$$ln(x) = 2 * (y + \frac{y^3}{3} + \frac{y^5}{5} + \dots)$$

Логарифм по основанию 2:

$$log_2(x) = \frac{ln(x)}{ln(2)}$$

Десятичный логарифм:

$$lg(x) = \frac{ln(x)}{ln(10)}$$

Практическая часть

Клиентский логгер

Клиентский логгер реализован как производный класс от абстрактного *logger*. Внутри он хранит ассоциативную структуру, сопоставляющую каждому уровню важности пару: флаг вывода в консоль и список файловых потоков. Потоки обёрнуты в маленький вспомогательный класс *refcounted_stream*, который ведёт глобальную таблицу открытых файлов: при каждом новом логгере счётчик ссылок увеличивается, а когда он достигает нуля, файл автоматически закрывается, что надёжно защищает от повторных открытий и утечек файловых дескрипторов (см. приложение А, листинги А.2, А.4).

Метод *log* (см. приложение А, листинг А.1) сначала ищет по уровню важности набор назначенных потоков; если такой уровень не сконфигурирован, выполнение завершается мгновенно. Далее вызывается *make_format*, где пользовательский шаблон проходит посимвольный разбор: флаг “%d” подменяется текущей датой, “%t” — временем, “%s” — строковым представлением важности, “%m” — самим сообщением (см. приложение А, листинг А.3). Символы, не совпавшие с флагами, выводятся как есть. Получившаяся строка записывается в консоль, если она включена, и дублируется во все открытые файлы соответствующего уровня.

Конструктор логгера получает от строителя уже сформированную карту потоков и строку формата. Он лишь проходит по каждому уровню и вызывает *open* у всех *refcounted_stream*, чтобы гарантировать, что реальные *ofstream* уже готовы к работе. Сам класс не содержит статического состояния, поэтому все операторы копирования и перемещения реализованы простым переносом карт и строки формата.

client_logger_builder инкапсулирует поэтапную сборку этой карты. Метод *add_file_stream* добавляет путь к файлу в список потоков нужной важности, а *add_console_stream* выставляет булевый признак вывода на экран. При необходимости конфигурацию можно описать в JSON-файле, а *transform_with_configuration* разберёт его: читается объект по заданному пути, при наличии поля format задаётся формат, а для каждой секции с именем важности вызывается вспомогательный *parse_severity*, который разносит флаги *console* и массив *paths* по внутренним структурам.

Пока логгер работает, открытые файлы остаются общими для всех его копий. Когда очередная копия уничтожается, её refcounted_stream декрементирует

счетчик; как только он достигнет нуля, поток закрывается и запись в глобальной таблице удаляется. Такой подход избавляет приложение от дублирующего вывода в один и тот же файл и упрощает управление ресурсами.

Таким образом, клиентский логгер сочетает в себе паттерн “Строитель” для удобной конфигурации, формальный синтаксис пользовательского формата и безопасное управление файлами через счетчик ссылок.

Бинарное дерево поиска

Основные компоненты реализации

Структура узла дерева

Узел хранит данные (пару ключ-значение) и указатели на родителя и поддеревья. Используется “variadic template” для универсального конструирования.

Основной класс бинарного дерева поиска:

Листинг #1. Фрагмент класса бинарного дерева поиска

```
template<typename tkey, typename tvalue, compator<tkey> compare = std::less<tkey>, typename tag = __detail::BST_TAG>
class binary_search_tree : private compare {
	// ...
protected:
	node *_root;
	logger *_logger;
	size_t _size;
	pp_allocator<value_type> _allocator;
};
```

Класс является шаблонным со следующими параметрами:

- *tkey*: тип ключа
- *tvalue*: тип значения
- *compare*: компаратор для сравнения ключей (по умолчанию *std::less*)
- *tag*: тэг для специализации поведения (используется для наследования)

Класс бинарного дерева поиска и структура его узла поддерживают семантику перемещения. Помимо этого, все основные операции также

поддерживают ее, и могут работать как с *lvalue* объектами, так и с *rvalue* объектами.

Итераторы

Реализованы 3 типа итераторов для разных порядков обхода:

1. Префиксный (*prefix_iterator*) - обход в порядке “корень-левый-правый”;
2. Инфиксный (*infix_iterator*) - обход в отсортированном порядке (по умолчанию);
3. Постфиксный (*postfix_iterator*) - обход в порядке “левый-правый-корень”.

Для каждого типа итератора реализованы:

- обычный итератор,
- константный итератор,
- reverse итератор,
- константный reverse итератор.

Итераторы являются STL-совместимыми, они удовлетворяют концепту *std::bidirectional_iterator*. Помимо указателя на узел, итераторы также имеют поле *_backup*. Оно используется для того, чтобы при необходимости можно было вернуться к предыдущей посещенной ноде, например, если будет вызвана операция декремента для *end()*.

Вспомогательный класс:

Листинг #2. Вспомогательный класс bst_impl

```
namespace __detail {

     template<typename tkey, typename tvalue, typename compare, typename tag>

    class bst_impl {

    public:

        template<class ...Args>

        static node *create_node(binary_search_tree &cont, Args &&...args);

        static void delete_node(binary_search_tree &cont, node *n);

        static void erase(binary_search_tree &cont, node **node_ptr);

        static void swap(binary_search_tree &lhs, binary_search_tree &rhs) noexcept;
```

```
    };
}
```

Основным вспомогательным классом является *bst::impl*. В нем содержатся функции, необходимые для реализации паттерна проектирования “шаблонный метод”. Эти функции в дальнейшем переопределяются для реализации дополнительного функционала в классах-наследниках, таких как классы красно-черного дерева, AVL-дерева, splay дерева. Функции *post_insert* и *post_search*, как понятно из названия, вызываются после операций вставки и поиска. Функция *delete_node* отвечает за освобождение узлов, которые могут иметь иную структуру, чем в классе бинарного дерева поиска. Функция *erase* отвечает непосредственно за удаление элемента из дерева.

Основные операции

Вставка элементов

Вставка элементов реализована в соответствии с алгоритмом. Сначала происходит поиск элемента в дереве. Далее выделяется память для узла при помощи функции *__detail::bst_impl<tkey, tvalue, compare, tag>::create_node()*. После этого обновляются указатели у родителя нового узла.

Листинг #3. Цикл поиска места вставки нового узла

```
while (current != nullptr) {
   if (compare_keys(value.first, current->data.first)) {
       prev = current;
       current = current->left_subtree;
   } else {
       prev = current;
       current = current->right_subtree;
   }
}
```

Удаление элементов

Как было описано в теоретической части, рассматриваются три случая: удаление узла без детей, с одним ребенком и с двумя детьми. В данной реализации узел с двумя детьми заменяется самым правым ребенком левого поддерева.

Поиск элементов

Поиск элементов реализован в соответствии с изложенным ранее алгоритмом. Функция поиска возвращает инфиксный итератор, указывающий на искомую ноду, либо итератор *end()*, если искомая нода не найдена.

Повороты

При реализации поворотов двойной левый поворот был реализован при помощи вызова двух малых левых поворотов, а большой левый поворот был реализован при помощи последовательных правого и далее левого поворотов. Аналогично с правым двойным поворотом и правым большим поворотом.

AVL-дерево

AVL-дерево поддерживает сбалансированность с помощью использования данных о высоте поддеревьев, что позволяет эффективно выполнять операции поиска, вставки и удаления с логарифмической сложностью.

Основные классы и структуры

Класс AVL_tree

Класс AVL_tree является основным элементом реализации. Он наследует от базового класса *binary_search_tree* и реализует операции вставки, удаления и балансировки, специфичные для AVL-дерева.

Данный класс публично наследуется от класса бинарного дерева поиска. Для использования паттерна “шаблонный метод” у класса AVL-дерева используется в качестве шаблонного параметра тэг *AVL_TAG*, для которого определены соответствующие функции в *bst_impl*.

Листинг #4. Структура узла

```
struct node final : public parent::node {
   size_t height;
   void recalculate_height() noexcept;
   /*
    * Returns positive if right subtree is bigger
    */
```

```
    short get_balance() const noexcept;


    template<class ...Args>
    node(parent::node *par, Args &&... args);
    ~node() noexcept override = default;
};
```

Каждый узел хранит высоту поддерева, что позволяет вычислять баланс дерева. Методы *recalculate_height* и *get_balance* служат для обновления высоты узла и определения его баланса (разница между высотами правого и левого поддеревьев).

Вставка

Процесс вставки в AVL-дерево начинается с обычной вставки элемента в бинарное дерево поиска. Однако, после добавления нового узла необходимо провести несколько дополнительных шагов для поддержания сбалансированности дерева:

1. Стандартная вставка, как в бинарном дереве поиска;
2. Пересчёт высот. После вставки нового узла необходимо пересчитать высоты всех узлов от вставленного до корня дерева, для этого используется методы узла *recalculate_height*. Высота каждого узла обновляется в зависимости от высот его левого и правого поддеревьев;
3. Балансировка дерева. После того как высоты узлов обновлены, необходимо выполнить операцию балансировки.

Удаление

1. Поиск элемента для удаления: Как и в обычном бинарном дереве поиска, мы начинаем с корня и ищем элемент. После того как нужный элемент найден, начинаем процесс его удаления;
2. Удаляем узел как в бинарном дереве поиска;
3. Пересчёт высот: После удаления элемента и возможной замены узлов, необходимо пересчитать высоты всех узлов от места удаления до корня;
4. Ребалансировка: После удаления элемента и пересчёта высот вызывается функция ребалансировки (*rebalance_to_root*).

Ребалансировка

Для этого используется метод *rebalance* (см. приложение Б, листинг Б.1).

1. Проверка баланса: После изменения структуры дерева мы начинаем проверку баланса с узла, который был затронут операцией (вставка или удаление). Баланс узла определяется как разница высот правого и левого поддеревьев. Если эта разница больше 1 или меньше -1, то узел нарушает баланс, и его необходимо откорректировать;
2. Выбор типа поворота:
    a. Если баланс равен 2 (правое поддерево "перевешивает"), то проверяется баланс правого поддерева:
        1. Если правое поддерево правого узла более высокое (или сбалансированное), выполняется правый поворот;
        2. Если левое поддерево правого узла более высокое, выполняется двойной поворот: сначала левый поворот на правом поддереве, затем правый поворот на родительском узле.
    b. Если баланс равен -2 (левое поддерево "перевешивает"), то проверяется баланс левого поддерева:
        1. Если левое поддерево левого узла более высокое (или сбалансированное), выполняется левый поворот;
        2. Если правое поддерево левого узла более высокое, выполняется двойной поворот: сначала правый поворот на левом поддереве, затем левый поворот на родительском узле.
3. Обновление высот: После выполнения поворота или последовательных поворотов, необходимо пересчитать высоты всех затронутых узлов.

Обработка всех этих случаев при балансировке выглядит следующим образом:

Листинг #5. Фрагмент функции rebalance, касающийся обработки случая, когда левое поддерево больше правого

```
} else if (n->get_balance() == -2) {

   auto hl = left->left_subtree

             ? static_cast<typename AVL_tree<tkey, tvalue, compare>::node
*>(left->left_subtree)->height : 0;

   auto hr = left->right_subtree

             ? static_cast<typename AVL_tree<tkey, tvalue, compare>::node
*>(left->right_subtree)->height : 0;
```

```
   if (hr <= hl) {

       binary_search_tree<tkey, tvalue, compare,
__detail::AVL_TAG>::small_right_rotation(to_balance);

   } else {
```

Итераторы

Реализованы итераторы, аналогичные итераторам бинарного дерева поиска. Фактически итераторы AVL-дерева являются оберткой вокруг обычных итераторов бинарного дерева.

Splay-дерево

Рассматриваемая реализация Splay-дерева использует стандартные операции бинарного дерева поиска, но с добавлением специфической операции *splay*, которая перемещает недавно использованные элементы в корень дерева. Это дерево не поддерживает строгую балансировку, как, например, AVL-дерево, но эффективно обслуживает запросы, которые часто касаются одного и того же элемента, помещая его в корень для ускорения будущих операций.

Основные классы и структуры

Класс splay_tree

Класс *splay_tree* наследует от шаблонного класса *binary_search_tree* и реализует дополнительные операции, характерные для splay-дерева. В отличие от стандартных бинарных деревьев поиска, операция splay применяется после каждой операции вставки, удаления и поиска для перемещения соответствующего узла в корень.

Реализация использует шаблоны для обеспечения универсальности, позволяя работать с любыми типами ключей и значений. Это даёт возможность создавать деревья для разных типов данных без необходимости дублирования кода.

Конструкторы

Класс имеет несколько конструкторов для инициализации дерева из различных источников данных:

— конструктор по умолчанию с возможностью указания компаратора, аллокатора и логгера;

— конструкторы для диапазонов (с использованием итераторов или стандартных диапазонов C++20);
— конструктор для инициализации дерева из списка пар ключ-значение.

Листинг #6. Конструктор перемещения для splay_tree

```
template<typename tkey, typename tvalue, compator<tkey> compare>

splay_tree<tkey, tvalue, compare>::splay_tree(splay_tree &&other) noexcept
{

	parent::_root = other._root;

	parent::_logger = other._logger;

	parent::_allocator = other._allocator;

	parent::_size = other._size;

	other._root = nullptr;

	other._size = 0;

}
```

Методы

— *insert*: Осуществляет вставку элемента в дерево, а затем выполняет операцию splay для перемещения вставленного элемента в корень.
— *splay*: Основная операция Splay-дерева, которая перемещает элемент в корень дерева. Включает различные виды поворотов для восстановления структуры дерева.
— *merge*: Сливает два поддерева в одно дерево, используя splay-операцию для перемещения элементов в корень.
— *split*: Разделяет дерево на два поддерева по указанному ключу.

Операции с деревом

— Конструкторы и операторы копирования/перемещения: Поддерживаются стандартные операции копирования и перемещения для удобства работы с объектами класса *splay_tree*. В случае копирования элементы вставляются заново, а при перемещении происходит переназначение указателей на корень и другие важные структуры;
— Деструктор: Освобождает ресурсы, связанные с деревом.

Вспомогательные функции и реализации

create_node и *delete_node*

Эти методы используются для создания и удаления узлов в дереве. Они работают с аллокатором, который выделяет или освобождает память для узлов. Это позволяет эффективно управлять памятью при работе с динамическими данными.

post_insert и *post_search*

Эти методы вызываются после операции вставки или поиска и выполняют операцию *splay*. Например, после вставки нового элемента метод *post_insert* вызывает *splay*, чтобы переместить только что вставленный узел в корень.

erase

Этот метод используется для удаления узла. Он сначала вызывает операцию *splay*, чтобы переместить удаляемый узел в корень, а затем выполняет операцию слияния двух поддеревьев (левого и правого) для удаления узла из дерева.

Особенности реализации Splay-дерева

Операция *Splay*

Алгоритм изложен в теоретической части. Для поворотов дерева используются функции родительского класса бинарного дерева поиска (см. приложение В, листинг В.1).

Метод *Merge*

Сливает два дерева в одно, сначала выполняя операцию *splay* на правом поддереве первого дерева (или на самом правом элементе), затем соединяет оба поддерева. Это позволяет эффективно объединять два дерева в одно без необходимости их полного перераспределения.

Метод *Split*

Делит дерево на два поддерева, одно из которых будет содержать все элементы, меньшие заданного ключа, а другое — все элементы, большие или равные этому ключу. После выполнения операции *splay* элемент с заданным ключом оказывается в корне дерева, и мы просто разделяем дерево на два поддерева.

Реализация данной операции представлена ниже:

Листинг #7. Реализация операции Split

```
template<typename tkey, typename tvalue, compator<tkey> compare>

std::pair<typename binary_search_tree<tkey, tvalue, compare,
__detail::SPL_TAG>::node *, typename binary_search_tree<tkey, tvalue,
compare, __detail::SPL_TAG>::node *>

splay_tree<tkey, tvalue, compare>::split(const tkey key) {

     using node = binary_search_tree<tkey, tvalue, compare,
__detail::SPL_TAG>::node;

     auto it = parent::lower_bound(key);

     node *n = it.get_node();

     splay(n);

     node *left_child = parent::_root->left_subtree;

     parent::_root->left_subtree = nullptr;

     left_child->parent = nullptr;

     return {parent::_root, left_child};

}
```

Красно-чёрное дерево

Класс красно-чёрного дерева наследуется от бинарного дерева поиска и делегирует большинство своих методов в родителя. Структура узла была дополнена цветом. Непосредственная реализация находится в пространстве имён *__detail*.

Функция *create_node* создаёт узел в динамической области памяти с помощью аллокатора, переданному дереву. Созданный узел всегда имеет красный цвет. Функция *delete_node* деаллоцирует выделенный узел.

После вставки нового узла в красно-чёрное дерево его свойства могут быть нарушены. Функция *post_insert* исправляет это (см. приложение Г, листинг Г.1).

Пока вставленный узел не корень и его родитель красный, берутся указатели на родителя и дедушку, определяется «дядя» и в зависимости от его цвета либо перекрашиваются три узла и поднимается текущий узел к дедушке, либо при необходимости сначала выполняется «сдвиг» с малым поворотом (левым или правым), а затем ещё один поворот в противоположную сторону с перекраской родителя в чёрный, а дедушки в красный.

Как только узел становится корнем или его родитель уже чёрный, цикл завершается, и последний шаг гарантирует, что корень всегда остаётся чёрным. Таким образом, свойства дерева восстановлены.

Функция `erase` начинается с того, что переданный узел приводится к типу узла красно-чёрного дерева и проверяется на валидность — если он равен *nullptr*, выбрасывается исключение (см. приложение Г, листинг Г.2).

Если у удаляемого узла отсутствует один из потомков, он заменяется существующим потомком, а указатель *x* сохраняет ссылку на поддерево. Если оба потомка присутствуют, находится крайний правый узел в левом поддереве (преемник), его цвет сохраняется, а *x* устанавливается на его левое поддерево. После этого выполняется замена: если преемник является непосредственным потомком удаляемого узла, изменяются только родительские ссылки, иначе сначала производится *transplant* преемника, затем к нему присоединяется его левое поддерево, потом *transplant* исходного узла, и, наконец, к преемнику подсоединяется правое поддерево.

Далее узел удаляется из памяти, размер дерева уменьшается, и если удаленный узел был чёрным, начинается цикл восстановления красно-чёрных свойств. Пока *x* не является корнем и остаётся чёрным, вычисляются его родитель и брат, после чего в зависимости от положения *x* (левый или правый ребёнок) и цвета брата последовательно рассматриваются четыре случая:

— Если брат красный, он перекрашивается в черный, а родитель — в красный. Выполняется малый поворот в сторону *x*, указатель на брата обновляется;
— Если оба потомка брата черные, то брат перекрашивается в красный, а *x* поднимается к родителю;
— В остальных ситуациях при неодинаковой окраске потомков брата сначала поворачивается брат, затем родитель, и узлы перекрашиваются.

Когда все случаи учтены и свойства восстановлены, *x* (если он не *nullptr*) окрашивается в чёрный, а указатель *n* обновляется на новое текущее место (*x* или корень).

Класс длинного целого числа

Структура представления длинного целого числа

В классе *big_int* хранится *bool* переменная знак числа(1, если + и 0, если -) и *std::vector* коэффициентов типа *unsigned int*, который также включает в себя аллокатор, который передается объекту класса в конструкторе.

Порядок хранения цифр в числе — *little endian*. Число представлено в системе счисления с основанием $2^{8\times sizeof(unsigned\ int)}$ (см. приложение Д, листинг Д.1).

Конструкторы

Класс имеет несколько конструкторов для инициализации длинного целого числа из различных источников данных (см. приложение Д, листинг Д.2):

— инициализация числа из строки, которая представляет число, представленное в системе счисления *radix*;
— инициализация из вектора чисел типа *unsigned int* и знака *bool*;
— инициализация нулем числа, получая на вход только аллокатор.

Сложение длинных чисел

Выравнивание разрядов: *max_size* вычисляется как максимум между длиной текущего числа (*_digits.size()*) и длиной other._*digits* + *shift* (сдвиг влево для *other*).

Размер *_digits* увеличивается до *max_size*, недостающие разряды заполняются нулями. Сложение поразрядно:

В цикле обрабатываются все разряды от младшего к старшему (*little-endian*). На каждом шаге:

— Суммируются: текущий разряд (*_digits*[i]), разряд *other* (с учетом сдвига *i* - *shift*) и перенос (*carry*);
— Новый разряд: *_digits*[*i*] = *sum* % *BASE* (остаток от деления на основание системы);
— Перенос: *carry* = *sum* / *BASE*.

Если после обработки всех разрядов остался перенос (*carry* > 0), он добавляется в старший разряд (см. приложение Д, листинг Д.3).

Разность длинных чисел

Реализован алгоритм вычитания целых чисел в соответствии с теоретической частью. Цикл по разрядам (little-endian: младшие разряды в начале массива):

Начальное значение difference равно займу (*borrow*) с предыдущего шага.

К difference добавляется текущий разряд *abs_this* и вычитается разряд *abs_other*. Если *difference* < 0, то добавляется *BASE* (заём из старшего разряда), устанавливается *borrow* = -1 для следующего разряда.

Результат сохраняется в *result*[*i*] (см. приложение Д, листинг Д.4).

Умножение длинных целых чисел согласно алгоритму умножения чисел в столбик

Метод *multiply_assign* с правилом *trivial* реализует классический алгоритм умножения длинных целых чисел "в столбик" (см. приложение Е, листинг Е.1).

Основные этапы работы:

Если хотя бы один из множителей равен нулю, результат обнуляется.

Создается объект result, размер массива *_digits* которого равен сумме длин множителей. Внешний цикл перебирает разряды текущего числа (*this->_digits[i]*). Внутренний цикл:

1. Перебирает разряды *other* или продолжает, пока есть перенос (*carry*);
2. Вычисляет произведение:
   $prod = \text{разряд}_{this} * \text{разряд}_{other} + \text{значение\_в\_}result + \text{перенос};$
3. Сохраняет остаток от деления на $BASE$ в *result._digits[i + j]*;
4. Обновляет перенос для следующего разряда.

Умножения длинных целых чисел согласно алгоритму Карацубы умножения чисел

В *big_int* для длинных чисел умножение переключается с квадратичного алгоритма на Карацубу, когда решающая функция *decide_mult* видит, что количество разрядов второго множителя превышает 32; тогда *operator**= передаёт управление *multiply_karatsuba* (см. приложение Ж, листинг Ж.1). Эта свободная функция объявлена friend, чтобы обращаться к приватным вектору *_digits* и признаку знака, но при этом остаётся чистой: она не меняет аргументы, а строит новый *big_int* и отдаёт его вверх по стеку.

Первая строчка *multiply_karatsuba* делает быстрый выход: если хотя бы один операнд короче девяти машинных слов, рекурсия невыгодна, и функция просто делегирует тривиальному умножению в один проход по разрядам. Число 8 выбрано эмпирически, чтобы не тратить время на нарезку и аллокации, когда выигрыша всё равно не будет.

Далее определяется max_size — это половина большей из двух длин, округленная вверх. Каждое число режется на две части: векторы low получают первые max_size элементов, а $high$ — оставшиеся. Срезы собираются через *assign*, затем *optimise* удаляет возможные ведущие нули и поправляет знак. Таким образом, a представляется как $h_1 * BASE^{max\_size} + l_1$, а b — как $h_2 * BASE^{max\_size} + l_2$.

Функция рекурсивно вычисляет три произведения: $z_0 = l_1 * l_2$, $z_2 = h_1 * h_2$ и $z_1 = (l_1 + h_1)(l_2 + h_2)$. После возврата z_1 избыточен, поэтому из него вычитаются z_0 и z_2 — получается скобка $(h_1 + l_1)(h_2 + l_2) - h_1h_2 - l_1l_2$, ровно как в классической формуле Карацубы.

Чтобы проставить степени основания, используются сдвиги влево: *operator*<<= перемножает число на $BASE^k$, где k равно количеству «слотов» разрядов. z_2 двигается на $2 * max\_size$, z_1 — на max_size. Это эквивалентно умножению на $BASE^{2*max\_size}$ и $BASE^{max\_size}$ соответственно, но быстрее и без промежуточных копий.

Последний шаг складывает z_2, z_1 и z_0, выставляет знак результата равенство знаков множителей и вызывает *optimise* для удаления ведущих нулей. Итоговый алгоритм занимает $O(n^{log_2 3})$ операций над разрядами вместо $O(n^2)$. Тривиальный случай обрабатывается начальной проверкой, а памяти выделяется ровно столько, сколько нужно, поскольку каждая ветвь рекурсии строит свои low и $high$ без резервирования «про запас». Благодаря этому умножение больших *big_int* получается ощутимо быстрее школьного» способа, особенно когда числа занимают сотни и тысячи машинных слов.

Умножения длинных целых чисел согласно алгоритму Шёнхаге-Штрассена умножения чисел

Реализован алгоритм Шёнхаге-Штрассена умножения чисел согласно теоретической части.

Для того, чтобы вычисления происходили с приемлемой точностью было выбрано два модуля:

$$MOD1 = 119 * 2^{23} + 1 = 998244353$$

$$MOD2 = 504 * 2^{22} + 1 = 2113929217$$

Оба модуля — простые числа и удовлетворяют всем условиям для использования в NTT. Также для обоих корней были предпосчитаны первообразные корни девятнадцатой степени из единицы и обратные к ним с помощью формулы Эйлера для обратного в кольце вычетов по модулю. Девятнадцатая степень корней обеспечивает точность вычислений для чисел меньших 250 000 знаков длинной.

Для оптимизации вычислений при первом вызове функций предподсчитываются массивы PWS и $IPWS$ для каждого модуля, числа в которых являются разными степенями корней из единицы и обратные к ним.

Для каждого модуля просчитывается полный цикл алгоритма:

1. Вычисляется NTT для обоих векторов;
2. Почленно перемножаем элементы векторов, полученных после NTT;
3. Вызываем $INTT$ для результата.

После вычислений по обоим модулям используется китайская теорема об остатках для двух модулей, чтобы восстановить точность вычислений. Для каждого индекса вызывается функция CRT которая восстанавливает точность вычислений.

В конце результат восстанавливается их вектора коэффициентов с помощью алгоритма Горнера (см. приложение З, листинг З.1).

Целочисленное деление в столбик

В функции обрабатывается исключительная ситуация деления на 0, в таком случае выбрасывается исключение типа std::logic_error. Как уже было сказано выше, каждая цифра искомого частного определяется итеративно при помощи бинарного

поиска. В качестве левой границы бинарного поиска взят 0, в качестве правой границы, которая должна быть недостижима, взято основание системы счисления. Деление выполняется вместе с присваиванием, то есть частное будет находиться в делимом. Помимо этого, функция также возвращает ссылку на результат. (см. приложение И, листинг И.1).

Класс дробь

Класс *fraction* инкапсулирует обыкновенную дробь с произвольной точностью: числитель и знаменатель хранятся в *big_int*, а значит ограничения появляются только у памяти. Основная инвариантная идея проста: любая дробь хранится в приведённом виде со знаком, вынесенным в знаменатель, и с взаимно-простыми числителем и знаменателем. За это отвечает приватный метод *optimise* (см. приложение К, листинг К.2). Он запрещает нулевой знаменатель, обнуляет знаменатель, если числитель равен нулю, переносит знак в знаменатель, вычисляет НОД с помощью собственной *gcd* и делит обе части на него, тем самым обеспечивая каноническое представление.

Создание объекта возможно двумя путями (см. приложение К, листинг К.1). Шаблонный конструктор с “perfect forwarding” принимает любые типы, конвертируемые к *big_int*, и тут же вызывает *optimise*. Вторая точка входа — конструктор, принимающий аллокатор для *big_int*; он позволяет строить нулевую дробь и использовать пользовательскую стратегию выделения памяти. По умолчанию знаменатель ставится равным единице, чтобы ноль выглядел как 0/1 .

Арифметика реализована школьными методами: при сложении или вычитании числители приводятся к общему знаменателю, после чего вызывается *optimise*; при умножении и делении числители и знаменатели перемножаются «как есть», затем опять вызывается *optimise*. Чтобы снизить накладные расходы на копирование, каждая базовая операция реализована в двух видах: компаунд-оператор (“+=”, “−=”, “*=”, “/=”) изменяет сам объект и возвращает его по ссылке, а отдельный «чистый» вариант выполняет те же действия, но создает копию, не затрагивая исходный экземпляр. Унарный минус копирует объект, меняет знак знаменателя и снова оптимизирует, поэтому -(-3/4) гарантированно даст 3/4, а не -3/-4. Равенство проверяется простым сравнением числителей и знаменателей, так как канонический вид уже обеспечен. Для частичного упорядочения используется “<=>”: знак сравнивается отдельно, затем дроби сопоставляются через перекрестное умножение с учетом возможного отрицательного знаменателя, что устраняет риск переполнения при конструировании промежуточных дробей .

Для ввода и вывода задействованы операторы потоков. *operator*<< просто печатает результат *to_string*; последняя выводит знак, числитель и модуль знаменателя. operator>> читает строку, валидирует её регулярным выражением (допустимы форматы “a” и “a/b” со знаком), конструирует дробь и сразу оптимизирует её, так что любые лишние нули или минусы исчезают ещё на этапе чтения.

Математический блок покрывает целый ряд функций. Тригонометрические *sin* и *cos* вычисляются по степенным рядам Тейлора: члены ряда суммируются, пока их модули превышают *epsilon*. *tg* и *ctg* реализованы через отношение синуса и косинуса, так что деление на ноль проверяется автоматически. Обратные тригонометрические функции строятся по классическим степенным рядам (*arcsin*) или через тождества (*arccos* через π/2 – *arcsin*) (см. приложение К, листинг К.3).

Для *ln* использована формула 1/2 *ln*((1+x)/(1-x)) с разложением в ряд, которая сходится быстрее, если *x* близок к единицы. Булевы проверки исключают недопустимые аргументы — отрицательные числа для логарифма, |x| > 1 для arcsin и arccos, нули для ctg и так далее (см. приложение К, листинг К.4).

Корень *n*–й степени считается методом Ньютона: считается приближение, пока разность соседних шагов не станет меньше *epsilon*; для чётной степени заранее блокируется отрицательный радиканд. *pow* работает через быстрое возведение в степень, экономя число умножений (см. приложение К, листинг К.5).

Каждая из этих функций использует только уже готовые операции над дробями, так что никакой повторной оптимизации числителей/знаменателей внутри не требуется — результат каждого элементарного действия уже приведён к канону той же *optimise*.

Внешние свободные функции *abs* и *gcd* дополняют класс: первая перегружается для *big_int* и *fraction*, вторая реализует классический алгоритм Евклида для целых *big_int*. Они объявлены в том же .cpp-файле, поэтому доступны как inlined-кандидаты внутри тяжёлых циклов математических функций.

Таким образом, *fraction* даёт полностью самодостаточный тип произвольной рациональной точности с полной арифметикой, богатым набором элементарных и специальных функций и строгим контролем над каноническим представлением, что позволяет безопасно использовать его в остальной части проекта без риска накопления неупрощённых дробей или пропуска знака.

Вывод

В ходе работы разработана компактная, но насыщенная библиотека алгоритмов и структур данных, ориентированная на современные требования к производительности и устойчивости. В неё входит семейство поисковых деревьев — от бинарного до AVL-, красно-чёрного и Splay-деревьев. Для высокоточной арифметики реализован длинный целочисленный тип, автоматически переключающийся между «школьным» $O(n^2)$ умножением, Карацубой и Шёнхаге-Штрассеном при росте размера операндов. На его основе построена рациональная математика: дроби всегда переводятся в канонический вид, поддерживают полный набор операций и расширяются элементарными функциями — от синуса до логарифма. Библиотеку дополняет гибкий механизм логирования. С помощью «Строителя» формируется конфигурация: определяется формат сообщений и задаются каналы вывода. Логгер ведёт запись, делит открытые файлы между своими копиями через счётчик ссылок и тем самым исключает повторное открытие одних и тех же дескрипторов.

Код написан под стандарт C++20. Везде используются семантика перемещения, RAII и умные указатели, а входные данные проверяются до начала вычислений. Архитектура разделяет хранение данных, алгоритмы и конфигурацию, опираясь на принципы SOLID, что облегчает дальнейшее расширение.

Практическая ценность проявляется в двух направлениях. Во-первых, библиотека готова к применению там, где требуются надёжные структуры данных и длинная арифметика — в криптографии, финансовой аналитике, научных расчётах. Во-вторых, проект служит демонстрацией того, как фундаментальные алгоритмы можно реализовать «по промышленным стандартам»: с тестами, строгим управлением ресурсами и декларативной настройкой.

Список использованных источников

1. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. Алгоритмы: построение и анализ, 3-е издание = Introduction to Algorithms, Third Edition. — М.: «Вильямс», 2013. — 1328 с. — ISBN 978-5-8459-1794-2.

2. Быстрое преобразование Фурье [Электронный ресурс] // Algorithmica. — URL: https://ru.algorithmica.org/cs/algebra/fft/ (дата обращения: 16.05.2025).

Приложения

Приложение А Реализация клиентского логгера

Листинг А.1 – Реализация метода логирования

```
logger& client_logger::log(
	const std::string &text,
	logger::severity severity) &
{
	auto streams_iter = _output_streams.find(severity);

	if (streams_iter == _output_streams.end()) {
		return *this;
	}
	std::string formatted_text = make_format(text, severity);

	auto& streams = streams_iter->second;

	// If console stream is enabled, print to console
	if (streams.second) {
	std::cout << formatted_text << std::endl;
	}

	// Print to all file streams
	for (auto& file_stream : streams.first) {
		auto stream_ptr = file_stream._stream.second;
		if (stream_ptr != nullptr) {
			*stream_ptr << formatted_text << std::endl;
		}
	}
```

```
    return *this;
}
```

Листинг А.2 – Реализация функции форматирования

```
std::string client_logger::make_format(const std::string &message, severity
sev) const
{
    std::ostringstream msg;

    bool in_flag = false;

    for (auto& c : _format) {
        if (c == '%') {
            in_flag = true;
            continue;
        } else if (!in_flag) {
            msg << c;
            continue;
        }

        client_logger::flag flag = client_logger::char_to_flag(c);

        switch (flag) {
            case client_logger::flag::DATE:
                msg << logger::current_date_to_string();
                break;
            case client_logger::flag::TIME:
                msg << logger::current_time_to_string();
                break;
            case client_logger::flag::SEVERITY:
```

```
                    msg << logger::severity_to_string(sev);
                    break;
               case client_logger::flag::MESSAGE:
                    msg << message;
                    break;
               case client_logger::flag::NO_FLAG:
                    msg << c;
                    break;
          }

          in_flag = false;
     }

     return msg.str();
}
```

Листинг А.2 – Конструктор потока со счётчиком ссылок

```
client_logger::refcounted_stream::refcounted_stream(const std::string
&path)
{
     auto opened_stream = _global_streams.find(path);

     if (opened_stream == _global_streams.end()) {
          // Register empty stream
          _global_streams[path] = {1, std::ofstream()};
     } else {
          // Increment refcount
          opened_stream->second.first++;
     }

     _stream = {path, nullptr};
```

```
}
```

Листинг А.3 – Метод *open* у потока со счётчиком ссылок

```
void client_logger::refcounted_stream::open()
{
	if (_stream.second != nullptr) {
		return;
	}

	auto& stream = _global_streams[_stream.first].second;

	if (stream.is_open()) {
		_stream.second = &stream;
		return;
	}

	stream.open(_stream.first);

	if (!stream.is_open()) {
		throw std::runtime_error("Failed to open file: " +
_stream.first);
	}

	_stream.second = &stream;
}
```

Листинг А.4 – Деструктор потока со счётчиком ссылок

```
client_logger::refcounted_stream::~refcounted_stream()
{
	auto& stream = _global_streams[_stream.first];
	if (--stream.first == 0) {
```

```
            stream.second.close();
            _global_streams.erase(_stream.first);
        }
    }
```

Приложение Б Реализация AVL-дерева

Листинг Б.1 – Код функции *rebalance*

```
template<typename tkey, typename tvalue, compator<tkey> compare>

binary_search_tree<tkey, tvalue, compare, __detail::AVL_TAG>::node *

AVL_tree<tkey, tvalue, compare>::rebalance(parent::node *to_balance) {

    auto *n = static_cast<typename AVL_tree<tkey, tvalue, compare>::node
*>(to_balance);

   auto *left = static_cast<typename AVL_tree<tkey, tvalue, compare>::node
*>(n->left_subtree);

   auto *right = static_cast<typename AVL_tree<tkey, tvalue, compare>::node
*>(n->right_subtree);

   if (n->get_balance() == 2) {

       auto hl = right->left_subtree

                              ? static_cast<typename AVL_tree<tkey, tvalue,
compare>::node *>(right->left_subtree)->height : 0;

       auto hr = right->right_subtree

                              ? static_cast<typename AVL_tree<tkey, tvalue,
compare>::node *>(right->right_subtree)->height : 0;

       if (hl <= hr) {

                                binary_search_tree<tkey, tvalue, compare,
__detail::AVL_TAG>::small_left_rotation(to_balance);

       } else {

                                binary_search_tree<tkey, tvalue, compare,
__detail::AVL_TAG>::big_left_rotation(to_balance);

       }

   } else if (n->get_balance() == -2) {

       auto hl = left->left_subtree

                              ? static_cast<typename AVL_tree<tkey, tvalue,
compare>::node *>(left->left_subtree)->height : 0;

       auto hr = left->right_subtree

                              ? static_cast<typename AVL_tree<tkey, tvalue,
compare>::node *>(left->right_subtree)->height : 0;

       if (hr <= hl) {
```

```
                                binary_search_tree<tkey,   tvalue,   compare,
__detail::AVL_TAG>::small_right_rotation(to_balance);

       } else {

                                binary_search_tree<tkey,   tvalue,   compare,
__detail::AVL_TAG>::big_right_rotation(to_balance);

       }

   }

   n->recalculate_height();

   if (n->parent) {

       if (n->parent->left_subtree) {

                static_cast<typename AVL_tree<tkey, tvalue, compare>::node
*>(n->parent->left_subtree)->recalculate_height();

       }

       if (n->parent->right_subtree) {

                static_cast<typename AVL_tree<tkey, tvalue, compare>::node
*>(n->parent->right_subtree)->recalculate_height();

       }

   }

   return to_balance;

}
```

Приложение В Реализация splay-дерева

Листинг В.1 - Реализация операции splay.

```
template<typename tkey, typename tvalue, compator<tkey> compare>

void splay_tree<tkey, tvalue, compare>::splay(binary_search_tree<tkey,
tvalue, compare, __detail::SPL_TAG>::node *x) {

   using node = binary_search_tree<tkey, tvalue, compare,
__detail::SPL_TAG>::node;

   if (x == nullptr) {

       return;

   }


   while (x->parent != nullptr) {

       node *p = x ? x->parent : nullptr;

       node *g = p ? p->parent : nullptr;

       if (g == nullptr) {

           if (x == p->left_subtree) {

               parent::small_right_rotation(p);

           } else {

               parent::small_left_rotation(p);

           }

           if (p->parent == nullptr) {

               parent::_root = p;

           }

       } else {

           bool x_is_left = x == p->left_subtree;

           bool p_is_left = p == g->left_subtree;

           if (x_is_left == p_is_left) {

               if (x_is_left) {

                   parent::double_right_rotation(g);

               } else {
```

```
                    parent::double_left_rotation(g);
                }
            } else {
                if (!x_is_left && p_is_left) {
                    parent::big_right_rotation(g);
                } else {
                    parent::big_left_rotation(g);
                }
            }
        }
    }
}
```

Приложение Г Реализация красно-черного дерева

Листинг Г.1 – Функция исправления свойств после вставки

```
template<typename tkey, typename tvalue, typename compare>
void bst_impl<tkey, tvalue, compare, RB_TAG>::post_insert(
	binary_search_tree<tkey, tvalue, compare, RB_TAG>& cont,
	typename binary_search_tree<tkey, tvalue, compare, RB_TAG>::node** n)
{
	// ... usings & helpers ...
	auto* z = static_cast<rnode*>(*n);
	while (z != static_cast<rnode*>(cont._root)
		&& z->get_parent()->color == rb::node_color::RED)
	{
		rnode* parent = z->get_parent();
		rnode* grand = z->get_grandparent();
		if (parent == grand->get_left())
		{
			rnode* uncle = grand->get_right();
			// 1. "y is red"
			if (uncle && uncle->color == rb::node_color::RED)
			{
				parent->color = rb::node_color::BLACK;
				uncle->color = rb::node_color::BLACK;
				grand->color = rb::node_color::RED;
				z = grand;
			}
			else
			{
				// 2. "y is black, z is a right child"
				if (z == parent->get_right())
```

```
            {
                  z = parent;
                  bst::small_left_rotation(link_to(parent));
                  parent = z->get_parent();
            }
            // 3. "y is black, z is a left child"
            parent->color = rb::node_color::BLACK;
            grand->color = rb::node_color::RED;
            bst::small_right_rotation(link_to(grand));
      }
}
else /* parent == grand->get_right() */
{
      rnode* uncle = grand->get_left();
      // 1. "y is red"
      if (uncle && uncle->color == rb::node_color::RED)
      {
            parent->color = rb::node_color::BLACK;
            uncle->color = rb::node_color::BLACK;
            grand->color = rb::node_color::RED;
            z = grand;
      }
      else
      {
            // 2. "y is black, z is a left child"
            if (z == parent->get_left())
            {
                  z = parent;
                  bst::small_right_rotation(link_to(parent));
                  parent = z->get_parent();
```

```
                    }
                    // 3. "y is black, z is a right child"
                    parent->color = rb::node_color::BLACK;
                    grand->color = rb::node_color::RED;
                    bst::small_left_rotation(link_to(grand));
                }
            }
        }
        static_cast<rnode*>(cont._root)->color = rb::node_color::BLACK;
}
```

Листинг Г.2 – Функция удаления узла

```
template<typename tkey, typename tvalue, typename compare>
void bst_impl<tkey, tvalue, compare, RB_TAG>::erase(
        binary_search_tree<tkey, tvalue, compare, RB_TAG>& cont,
        typename binary_search_tree<tkey, tvalue, compare, RB_TAG>::node** n)
{
        // ... usings & helpers ...
        rnode* z = static_cast<rnode*>(*n);
        if (!z) throw std::out_of_range("invalid iterator");

        rnode* y = z; // Удаляемая нода
        typename rb::node_color oc = y->color; // Цвет удаляемой ноды
        rnode* x = nullptr; // Поддерево заменяющей ноды

        if (!z->get_left() || !z->get_right())
        {
                x = z->get_left() ? z->get_left() : z->get_right();
                transplant(z, x);
        }
```

```
else
{
      y = maximum(z->get_left());
      oc = y->color;
      x = y->get_left();

      if (y->parent == z)
      {
            if (x) x->parent = y;
      }
      else
      {
            transplant(y, x);
            y->left_subtree = z->left_subtree;
            y->left_subtree->parent = y;
      }

      transplant(z, y);
      y->right_subtree = z->right_subtree;
      y->right_subtree->parent = y;
      y->color = z->color;
}

delete_node(cont, z);
--cont._size;

if (oc == rb::node_color::BLACK)
{
      while (x != static_cast<rnode*>(cont._root)
            && color(x) == rb::node_color::BLACK)
```

```
{
	rnode* xp = x ? x->get_parent() : nullptr;

	if (xp == nullptr) // В корне
	{
		if (x) x->color = rb::node_color::BLACK;
		break;
	}

	rnode* w = x->get_sibling();

	// Если нет родственника, обрабатываем этот случай как №2.
	if (w == nullptr)
	{
		x = xp;
		continue;
	}

	if (x == xp->get_left())
	{
		// 1. "w" is red"
		if (color(w) == rb::node_color::RED)
		{
			w->color = rb::node_color::BLACK;
			xp->color = rb::node_color::RED;
			bst::small_left_rotation(link_to(xp));
			w = xp->get_right();
		}

		// 2. "w is black and both of w's children are black"
```

```
                if (child_is_black(w, false) && child_is_black(w,
true))
                {
                     if (w) w->color = rb::node_color::RED;
                     x = xp;
                     continue;
                }

                // 3. "w is black, w's left child is red, and w's
right child is black"
                if (!child_is_black(w, false) && child_is_black(w,
true))
                {
                     w->get_left()->color = rb::node_color::BLACK;
                     w->color = rb::node_color::RED;
                     bst::small_right_rotation(link_to(w));
                     w = xp->get_right();
                }

                // 4. "w is black, w's left child is black, and w's
right child is red"
                w->color = xp->color;
                xp->color = rb::node_color::BLACK;
                if (rnode* wr = w->get_right())
                wr->color = rb::node_color::BLACK;
                bst::small_left_rotation(link_to(xp));
                break;
           }
           else /* x == xp->get_right() */
           {
                // 1. "w" is red"
                if (color(w) == rb::node_color::RED)
```

```
                {
                        w->color = rb::node_color::BLACK;
                        xp->color = rb::node_color::RED;
                        bst::small_right_rotation(link_to(xp));
                        w = xp->get_left();
                }

                // 2. "w is black and both of w's children are black"
                if (child_is_black(w, false) && child_is_black(w,
true))
                {
                        if (w) w->color = rb::node_color::RED;
                        x = xp;
                        continue;
                }

                // 3. "w is black, w's left child is black, and w's
right child is red"
                if (!child_is_black(w, true) && child_is_black(w,
false))
                {
                        w->get_right()->color = rb::node_color::BLACK;
                        w->color = rb::node_color::RED;
                        bst::small_left_rotation(link_to(xp));
                        w = xp->get_left();
                }

                // 4. "w is black, w's left child is red, and w's
right child is black"
                w->color = xp->color;
                xp->color = rb::node_color::BLACK;
                w->get_left()->color = rb::node_color::BLACK;
```

```
                    bst::small_right_rotation(link_to(xp));
                    break;
                }
            }

            if (x) x->color = rb::node_color::BLACK;
        }

        *n = (x ? x : static_cast<rnode*>(cont._root));
    }
```

Приложение Д Реализация класса big_int

Листинг Д.1 – Класс big_int

```
class big_int {
   bool _sign; // 1 +  0 -
   std::vector<unsigned int, pp_allocator<unsigned int> > _digits;
// ...
```

Листинг Д.2 – Конструкторы класса big_int

```
big_int::big_int(const std::vector<unsigned int, pp_allocator<unsigned int>
> &digits, bool sign) : _sign(sign),
    _digits(digits) {
    _digits.push_back(0);
    optimise();
}


big_int::big_int(std::vector<unsigned int, pp_allocator<unsigned int> >
&&digits, bool sign) noexcept : _sign(sign),
    _digits(std::move(digits)) {
    _digits.push_back(0);
    optimise();
}


big_int::big_int(const std::string &num, unsigned int radix,
pp_allocator<unsigned int> allocator) : _sign(true),
    _digits(allocator) {
    if (radix < 2 || radix > 36) {
        throw std::invalid_argument("Radix must be between 2 and 36");
    }
    if (num.empty()) {
        _digits.push_back(0);
        return;
```

```
}
std::string_view num_view(num);
size_t start = 0;
if (num_view[0] == '-') {
    _sign = false;
    start = 1;
} else if (num_view[0] == '+') {
    start = 1;
}
while (start < num_view.size() - 1 && num_view[start] == '0') {
    ++start;
}
num_view = num_view.substr(start);

if (num_view.empty() || num_view == "0") {
    _digits.push_back(0);
    _sign = true;
    return;
}
for (char c: num_view) {
    // string view only for using this range based for
    if (!is_valid_digit(c, radix)) {
        throw std::invalid_argument(
            std::string("Invalid character '") + c +
            "' for radix " + std::to_string(radix));
    }
    const unsigned digit = char_to_value(c);
    unsigned carry = digit;

    for (auto &d: _digits) {
```

```
            const auto product = static_cast<unsigned long long int>(d) *
radix + carry;

            d = static_cast<unsigned int>(product & UINT_MAX);

            carry = static_cast<unsigned int>(product >> (8 *
sizeof(unsigned int)));

        }


        if (carry > 0) {

            _digits.push_back(carry);

        }

    }

}


big_int::big_int(pp_allocator<unsigned int> allocator) : _sign(true),
_digits(allocator) {

    _digits.push_back(0);

};
```

Листинг Д.3 – Сложение big_int

```
big_int &big_int::plus_assign(const big_int &other, size_t shift) & {

    if (other._digits.size() == 1 && other._digits[0] == 0) return *this;

    if (_sign == other._sign) {

        size_t max_size = std::max(_digits.size(), other._digits.size() +
shift);

        _digits.resize(max_size, 0);


        unsigned long long carry = 0;

        for (size_t i = 0; i < max_size; ++i) {

            unsigned long long sum = carry;

            if (i < _digits.size()) {

                sum += _digits[i];

            }
```

```
            if (i >= shift && (i - shift) < other._digits.size()) {
                sum += other._digits[i - shift];
            }

            _digits[i] = static_cast<unsigned int>(sum % BASE);
            carry = sum / BASE;
        }

        if (carry > 0) {
            _digits.push_back(static_cast<unsigned int>(carry));
        }
    } else {
        big_int temp(other);
        temp._sign = _sign;
        minus_assign(temp, shift);
    }
    optimise();
    if (this->is_zero()) {
        _sign = true;
    }
    return *this;
}
```

Листинг Д.4 – Вычитание big_int

```
big_int &big_int::minus_assign(const big_int &other, size_t shift) & {
    if (other.is_zero()) return *this;

    if (_sign != other._sign) {
        big_int temp(other);
```

```
        temp._sign = _sign;
        return plus_assign(temp, shift);
    }

    big_int abs_this(*this);
    abs_this._sign = true;
    big_int abs_other(other);
    abs_other._sign = true;
    abs_other <<= shift;

    bool result_sign = _sign;
    if ((abs_this <=> abs_other) == std::strong_ordering::less) {
        result_sign = !result_sign;
        std::swap(abs_this._digits, abs_other._digits);
    }

    const size_t max_size = std::max(abs_this._digits.size(),
abs_other._digits.size());
    std::vector<unsigned int, pp_allocator<unsigned int> > result(max_size,
0, _digits.get_allocator());
    long long borrow = 0;

    for (size_t i = 0; i < max_size; ++i) {
        long long difference = borrow;
        if (i < abs_this._digits.size()) {
            difference += abs_this._digits[i];
        }

        if (i < abs_other._digits.size()) {
            difference -= abs_other._digits[i];
        }
```

```
        if (difference < 0) {
            difference += static_cast<long long>(BASE);
            borrow = -1;
        } else {
            borrow = 0;
        }
        result[i] = static_cast<unsigned int>(difference);
    }
    _digits = std::move(result);
    _sign = result_sign;
    optimise();
    if (this->is_zero()) {
        _sign = true;
    }
    return *this;
}
```

Приложение Е Реализация умножения в столбик

Листинг Е.1 – Умножение в столбик

```
big_int &big_int::multiply_assign(const big_int &other,
big_int::multiplication_rule rule) & {

    if (other.is_zero() || this->is_zero()) {

        _digits.clear();

        _digits.push_back(0);

        optimise();

        return *this;

    }


    if (rule == multiplication_rule::trivial) {

        big_int result(_digits.get_allocator());

        result._digits.resize(_digits.size() + other._digits.size(), 0);

        for (size_t i = 0; i < _digits.size(); ++i) {

            unsigned long long carry = 0;

            for (size_t j = 0; j < other._digits.size() || carry; ++j) {

                unsigned long long prod = result._digits[i + j] + carry;

                if (j < other._digits.size()) {

                    prod += static_cast<unsigned long long>(_digits[i]) *
other._digits[j];

                }


                result._digits[i + j] = static_cast<unsigned int>(prod %
BASE);

                carry = prod / BASE;

            }

        }


        _sign = (_sign == other._sign);
```

```
        _digits = std::move(result._digits);
        optimise();
        return *this;
    }
    // ...
}
```

Приложение Ж Реализация умножения алгоритмом Карацубы

Листинг Ж.1 – Умножение алгоритмом Карацубы

```
big_int multiply_karatsuba(const big_int &a, const big_int &b) {
	if (a._digits.size() <= 8 || b._digits.size() <= 8) {
		big_int result = a;
		result.multiply_assign(b,
big_int::multiplication_rule::trivial);
	return result;
	}
	const size_t max_size = (std::max(a._digits.size(), b._digits.size())
+ 1) / 2;


	big_int low1(a._digits.get_allocator()),
high1(a._digits.get_allocator());
	low1._digits.assign(
		a._digits.begin(),
		a._digits.begin() + std::min(max_size, a._digits.size())
	);
	if (a._digits.size() > max_size) {
		high1._digits.assign(a._digits.begin() + max_size,
a._digits.end());
	}
	low1.optimise();
	high1.optimise();
	// Sawing the first number


	big_int low2(b._digits.get_allocator()),
high2(b._digits.get_allocator());
	low2._digits.assign(
		b._digits.begin(),
		b._digits.begin() + std::min(max_size, b._digits.size())
```

```
    );
    if (b._digits.size() > max_size) {
        high2._digits.assign(b._digits.begin() + max_size,
b._digits.end());
    }
    low2.optimise();
    high2.optimise();
    // Sawing the second number

    big_int z0 = multiply_karatsuba(low1, low2);
    big_int z2 = multiply_karatsuba(high1, high2);

    big_int sum1 = low1 + high1;
    big_int sum2 = low2 + high2;
    big_int z1 = multiply_karatsuba(sum1, sum2);
    z1 -= z0;
    z1 -= z2;

    z2 <<= (max_size * 2); // this operation is equivalent to  *base^2
    z1 <<= max_size; // this operation is equivalent to  *base

    big_int result = z2 + z1 + z0;
    // (h1*Base + l1)(h2*Base +l2) = h1*h2*Base^2 + ((h1 + l1)(h2 + l2) -
h1*h2 - l1*l2)*Base + l1*l2
    result._sign = (a._sign == b._sign);
    result.optimise();
    return result;
}
```

Приложение З Реализация умножения Шёнхаге-Штрассена

Листинг 3.1 – Умножение алгоритмом Шёнхаге-Штрассена

```
using ll = long long;


// For all ai in vector Big_int number ai <= MOD, so I need to translate
vector to this base

constexpr ll base = 998244353 - 1;


/* MOD must have a structure (k * 2^m + 1), and it should be a prime
number.

 * MAXN <= 2^m

 * since I use two modules, I need to take MAXN <= min(2^m1, 2^m2) where m1
and m2 they belong to different modules

 * MAXN defines the maximum length of the numbers to be multiplied

 * MAXN >= N + M - 1, where N, M length of the numbers */

constexpr ll MAXN = (1 << 19);


/* W - primitive root of the MAXN degree of 1: x^k = 1 % MOD

 * IW - inverse modulo MOD to W

 * INV2 - inverse modulo MOD to 2 */


// MOD1 = 119 * 2^23 + 1

constexpr ll MOD1 = 998244353, W1 = 805775211, IW1 = 46809892, INV21 =
499122177;

//MOD2 = 504 * 2^22 + 1

constexpr ll MOD2 = 2113929217, W2 = 1838344356, IW2 = 130030948, INV22 =
1056964609;


/* pws[n] = W^(MAXN/n)

 * ipws[n] = IW^(MAXN/n)

 * They are used in the NTT algorithm to optimize calculations. */
```

```
void init(ll *pws1, ll *ipws1, ll *pws2, ll *ipws2) {
    pws1[MAXN] = W1;
    ipws1[MAXN] = IW1;
    for (int i = MAXN / 2; i >= 1; i /= 2) {
        pws1[i] = (pws1[i * 2] * pws1[i * 2]) % MOD1;
        ipws1[i] = (ipws1[i * 2] * ipws1[i * 2]) % MOD1;
    }

    pws2[MAXN] = W2;
    ipws2[MAXN] = IW2;
    for (int i = MAXN / 2; i >= 1; i /= 2) {
        pws2[i] = (pws2[i * 2] * pws2[i * 2]) % MOD2;
        ipws2[i] = (ipws2[i * 2] * ipws2[i * 2]) % MOD2;
    }
}



// Cooley-Tukey algorithm to find NTT
void fft(std::vector<ll> &a, std::vector<ll> &ans, int l, int cl, int step,
int n, bool inv, ll MOD, ll *pws, ll *ipws,
         ll INV2) {
    if (n == 1) {
        ans[l] = a[cl];
        return;
    }
    fft(a, ans, l, cl, step * 2, n / 2, inv, MOD, pws, ipws, INV2); // for
even
    fft(a, ans, l + n / 2, cl + step, step * 2, n / 2, inv, MOD, pws, ipws,
INV2); // for uneven
    ll cw = 1; // Current multiplier
    const ll gw = (inv ? ipws[n] : pws[n]); // Main multiplier for the
current level
```

```
    for (int i = l; i < l + n / 2; i++) {
        const ll u = ans[i];
        const ll v = (cw * ans[i + n / 2]) % MOD;
        ans[i] = (u + v) % MOD;
        ans[i + n / 2] = (u - v) % MOD;
        if (ans[i + n / 2] < 0) ans[i + n / 2] += MOD;
        if (inv) {
            ans[i] = (ans[i] * INV2) % MOD;
            ans[i + n / 2] = (ans[i + n / 2] * INV2) % MOD;
        }
        cw = (cw * gw) % MOD;
    }
}


template<ll MOD, ll * pws, ll * ipws, ll INV2>
std::vector<ll> poly_multiply(std::vector<ll>& a, std::vector<ll>& b, const
size_t n) {
    std::vector<ll> a_fft(n), b_fft(n);
    fft(a, a_fft, 0, 0, 1, n, false, MOD, pws, ipws, INV2);
    fft(b, b_fft, 0, 0, 1, n, false, MOD, pws, ipws, INV2);


    std::vector<ll> product_fft(n);
    for (size_t i = 0; i < n; ++i) {
        product_fft[i] = (a_fft[i] * b_fft[i]) % MOD;
    }


    std::vector<ll> product(n);
    fft(product_fft, product, 0, 0, 1, n, true, MOD, pws, ipws, INV2);
    return product;
}
```

```
// Normalization is necessary, because after CRT, the numbers may be
greater than base.
void normalize(std::vector<ll>& product) {
    ll carry = 0;
    for (auto &x: product) {
        x += carry;
        carry = x / base;
        x %= base;
    }
    while (carry) {
        product.push_back(carry % base);
        carry /= base;
    }
}

// Chinese remainder theorem for 2 modules it reduces to solving a system
of 2 equations
ll crt(const ll a1, const ll a2) {
    constexpr ll M = MOD1 * MOD2;
    constexpr ll y1 = 210156705; // mod2^{-1} mod mod1
    constexpr ll y2 = 1668891489; // mod1^{-1} mod mod2

    const ll term1 = (static_cast<__int128_t>(a1) * MOD2 % M) * y1 % M;
    const ll term2 = (static_cast<__int128_t>(a2) * MOD1 % M) * y2 % M;
    return (term1 + term2) % M;
}

big_int multiply_schonhage_strassen(big_int first,big_int second) {
    static ll pws1[MAXN + 1], ipws1[MAXN + 1];
    static ll pws2[MAXN + 1], ipws2[MAXN + 1];
```

```
    static bool initialized = false;
    if (!initialized) {
        init(pws1, ipws1, pws2, ipws2);
        initialized = true;
    }

    std::vector<ll> normalize_first_digits;
    std::vector<ll> normalize_second_digits;
    while (first != 0) {
        big_int div_res = first % base;
        normalize_first_digits.push_back(div_res._digits[0]);
        first /= base;
    }
    while (second != 0) {
        big_int div_res = second % base;
        normalize_second_digits.push_back(div_res._digits[0]);
        second /= base;
    }

    // For Cooley-Tukey algorithm, vectors must have length power of 2
    size_t n = 1;
    while (n < normalize_first_digits.size() +
normalize_second_digits.size()) n <<= 1;
    normalize_first_digits.resize(n, 0);
    normalize_second_digits.resize(n, 0);

    const auto res1 = poly_multiply<MOD1, pws1, ipws1,
INV21>(normalize_first_digits, normalize_second_digits, n);
    const auto res2 = poly_multiply<MOD2, pws2, ipws2,
INV22>(normalize_first_digits, normalize_second_digits, n);
```

```
    std::vector<ll> combined(res1.size());
    for (size_t i = 0; i < combined.size(); ++i) {
        combined[i] = crt(res1[i], res2[i]);
    }
    normalize(combined);

    // Collect the result from the coefficients obtained from the CRT
    big_int result = 0;
    big_int pow = 1;
    for (const long long i : combined) {
        big_int tmp = i;
        result += tmp * pow;
        pow *= base;
    }
    return result;
}
```

Приложение И Реализация целочисленного деления

Листинг И.1 – Реализация целочисленного деления

```
big_int &big_int::divide_assign(const big_int &other,
big_int::division_rule rule) & {

    if (this->is_zero()) return *this;

    if (other.is_zero()) throw std::logic_error("Division by zero");


    big_int abs_this(*this);

    abs_this._sign = true;

    big_int abs_other(other);

    abs_other._sign = true;

    if (abs_this < abs_other) {

        _digits.clear();

        _digits.push_back(0);

        optimise();

        return *this;

    }


    std::vector<unsigned int, pp_allocator<unsigned int> >
quotient(_digits.size(), 0, _digits.get_allocator());

    big_int remain(0, _digits.get_allocator());

    for (int i = static_cast<int>(_digits.size()) - 1; i >= 0; i--) {

        remain._digits.insert(remain._digits.begin(), _digits[i]);

        remain.optimise();


        unsigned long long left = 0, q = 0, right = BASE;

        while (left <= right) {

            const unsigned long long mid = left + (right - left) / 2;

            big_int temp = abs_other * big_int(static_cast<long long>(mid),
_digits.get_allocator());
```

```
            if (remain >= temp) {
                q = mid;
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        } // Binsearch for find the whole result

        if (q > 0) {
            const big_int temp = abs_other * big_int(static_cast<long
long>(q), _digits.get_allocator());
            remain -= temp;
        }
        quotient[i] = static_cast<unsigned int>(q);
    }

    _sign = (_sign == other._sign);
    _digits = std::move(quotient);
    optimise();
    return *this;
}
```

Приложение К Реализация класса дроби

Листинг К.1 – Конструктор класса дроби

```
fraction::fraction(const pp_allocator<big_int::value_type> allocator)
      : _numerator(0, allocator), _denominator(1, allocator) {
}
```

Листинг К.2 – Метод *optimise*

```
void fraction::optimise() {
      if (_denominator == 0) {
            throw std::invalid_argument("Denominator cannot be zero");
      }
      if (_numerator == 0) {
            _denominator = 1;
            return;
      }
      if (_numerator < 0) {
            _numerator = abs(_numerator);
            _denominator = 0_bi - _denominator;
      }
      const big_int divisor = gcd(abs(_numerator), abs(_denominator));
      _numerator /= divisor;
      _denominator /= divisor;
}
```

Листинг И.3 – Операции ввода/вывода

```
std::ostream &operator<<(std::ostream &stream, fraction const &obj) {
      stream << obj.to_string();
      return stream;
}

std::istream &operator>>(std::istream &stream, fraction &obj) {
```

```
	std::string input;
	stream >> input;
	std::regex fraction_regex(R"(^([-+]?[0-9]+)(?:/([-+]?[0-9]+))?$)");
	std::smatch match;
	if (std::regex_match(input, match, fraction_regex)) {
		big_int numerator(match[1].str(), 10);
		big_int denominator(1);
		if (match[2].matched) {
			denominator = big_int(match[2].str(), 10);
		}
		obj = fraction(numerator, denominator);
	} else {
		throw std::invalid_argument("Invalid fraction format");
	}
	obj.optimise();
	return stream;
}

std::string fraction::to_string() const {
	std::stringstream ss;
	if (_denominator < 0) {
		ss << "-";
	}
	ss << _numerator << "/" << abs(_denominator);
	return ss.str();
}
```

Листинг К.3 – Тригонометрические функции

```
fraction fraction::sin(fraction const &epsilon) const {
	fraction x = *this;
	fraction result(0, 1);
```

```
	fraction term = x;
	int n = 1;
	while (abs(term) > epsilon) {
		result += term;
		term = term * (-x * x);
		term /= fraction(2 * n * (2 * n + 1), 1);
		n++;
	}
	return result;
}

fraction fraction::arcsin(fraction const &epsilon) const {
	fraction x = *this;
	if (x > fraction(1, 1) || x < fraction(-1, 1)) {
		throw std::domain_error("arcsin is undefined for |x| > 1");
	}
	fraction result(0, 1);
	fraction term = x;
	big_int n = 1;
	fraction x_squared = x * x;

	while (abs(term) > epsilon) {
		result += term;
		big_int numerator = (2_bi * n - 1) * (2_bi * n - 1);
		big_int denominator = 2_bi * n * (2_bi * n + 1);
		fraction coeff(numerator, denominator);
		term = term * x_squared * coeff;
		n += 1;
	}
	return result;
```

```
}

fraction fraction::cos(fraction const &epsilon) const {
      fraction x = *this;
      fraction result(1, 1);
      fraction term(1, 1);
      int n = 1;
      while (true) {
            term = term * (-x * x);
            big_int denominator = (2 * n - 1) * (2 * n);
            term /= fraction(denominator, 1);
            if (abs(term) <= epsilon) {
                  break;
            }
            result += term;
            n++;
      }
      return result;
}

fraction fraction::arccos(fraction const &epsilon) const {
      if (*this > fraction(1, 1) || *this < fraction(-1, 1)) {
            throw std::domain_error("arccos is undefined for |x| > 1");
      }
      fraction half(1, 2);
      fraction pi_over_2 = half.arcsin(epsilon) * fraction{3 , 1};
      fraction arcsin_val = this->arcsin(epsilon);
      return pi_over_2 - arcsin_val;
}
```

```
fraction fraction::tg(fraction const &epsilon) const {
	fraction cosine = this->cos(epsilon * epsilon);
	if (cosine._numerator == 0) {
		throw std::domain_error("Tangent undefined");
	}
	return this->sin(epsilon * epsilon) / cosine;
}

fraction fraction::arctg(fraction const &epsilon) const {
	if (_denominator < 0) {
		return -(-*this).arctg(epsilon);
	}
	if (*this > fraction(1, 1)) {
		return fraction(1, 2) - (fraction(1, 1) / *this).arctg(epsilon);
	}
	fraction result(0, 1);
	fraction term = *this;
	int n = 1;
	int count = 0;
	while (abs(term) > epsilon) {
		result += fraction((count % 2 == 0) ? 1 : -1, n) * term;
		n += 2;
		count++;
		term *= *this * *this;
	}
	return result;
}
```

Листинг К.4 – Логарифмические функции

```
fraction fraction::ln(fraction const &epsilon) const {
	if (*this <= fraction(0, 1)) {
```

```
            throw std::domain_error("Natural logarithm of non-positive
number");
      }

      fraction y = (*this - fraction(1, 1)) / (*this + fraction(1, 1));
      fraction y_squared = y * y;
      fraction term = y;
      fraction sum = term;
      int denominator = 1;

      while (true) {
            term *= y_squared;
            denominator += 2;
            fraction delta = term / fraction(denominator, 1);

            if (abs(delta) <= epsilon) {
                  break;
            }
            sum += delta;
      }

      return sum * fraction(2, 1);
}

fraction fraction::log2(fraction const &epsilon) const {
      if (*this <= fraction(0, 1)) {
            throw std::domain_error("Logarithm of non-positive number is
undefined");
      }
      fraction ln_x = this->ln(epsilon);
      fraction ln_2 = fraction(2, 1).ln(epsilon);
```

```
	return ln_x / ln_2;
}


fraction fraction::lg(fraction const &epsilon) const {
	if (*this <= fraction(0, 1)) {
		throw std::domain_error("Base-10 logarithm of non-positive
number is undefined");
	}

	fraction ln_x = this->ln(epsilon);
	fraction ln_10 = fraction(10, 1).ln(epsilon);
	return ln_x / ln_10;
}
```

Листинг К.5 – Квадратный корень через алгоритм Ньютона-Рафсона

```
fraction fraction::root(size_t degree, fraction const &epsilon) const {
	if (degree == 0) {
		throw std::invalid_argument("Degree cannot be zero");
	}
	if (degree == 1) {
		return *this;
	}
	if (_numerator < 0 && degree % 2 == 0) {
		throw std::domain_error("Even root of negative number is not
real");
	}
	fraction x = *this;
	fraction guess = *this / fraction(degree, 1);
	fraction prev_guess;
	do {
		prev_guess = guess;
		fraction power = guess.pow(degree - 1);
```

```
            if (power._numerator == 0) {
                  throw std::runtime_error("Division by zero in root
      calculation");
            }
            guess = (fraction(degree - 1, 1) * guess + *this / power) /
      fraction(degree, 1);
      } while ((guess - prev_guess > epsilon) || (prev_guess - guess >
epsilon));
      if (_numerator < 0 && degree % 2 == 1) {
            guess = -guess;
      }
      return guess;
}
```

Приложение Л Репозиторий с исходным кодом

mai-fa-sp-4-sem [Электронный ресурс]. // github — URL: https://github.com/oryce/mai-fa-sp-4-sem (дата обращения: 16.05.2025).