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»

Реализация прикладных программ на двоичных деревьях и арифметических алгоритмов

Выполнил: Колесник Д.С.

Группа: 8О-213Б

Преподаватель: А.М. Романенков

Москва, 2025

Содержание

Введение

Современные вычислительные задачи предъявляют всё более высокие требования к различным аспектам программных решений: это касается не только скорости выполнения алгоритмов, но и качества их реализации, включая такие аспекты, как надёжность, устойчивость к ошибкам и способность эффективно масштабироваться при увеличении объёмов данных. Разработка высокопроизводительных структур данных и алгоритмов, которые способны корректно, стабильно и без потери точности обрабатывать значительные объёмы информации, становится критически важной задачей. Это особенно актуально в контексте работы с длинными целыми числами, сложными иерархическими структурами узлов, а также в условиях возникновения исключительных ситуаций, которые могут повлиять на ход вычислений. Такие требования характерны для множества прикладных областей: криптографии, где точность и безопасность вычислений играют ключевую роль, финансового анализа, где ошибки в расчётах могут привести к значительным убыткам, научных исследований, где необходима высокая точность при обработке больших массивов данных, а также многих других сфер, где даже малейшие отклонения в результатах недопустимы.

В рамках данной работы была поставлена амбициозная цель: разработать на языке программирования C++20 полноценную библиотеку, которая включала бы в себя набор фундаментальных алгоритмов и структур данных, спроектированных с учётом современных стандартов качества программного обеспечения. Основное внимание уделялось строгому следованию принципам SOLID, которые обеспечивают модульность, гибкость и удобство поддержки кода. Кроме того, важным приоритетом стало гарантированное управление ресурсами — библиотека должна эффективно использовать память и другие ресурсы, избегая утечек и обеспечивая оптимальную производительность. Ещё одним ключевым аспектом стало обеспечение полной защиты от ошибок времени выполнения: это включает в себя проверку входных данных, обработку исключительных ситуаций и предотвращение состояний, которые могут привести к сбоям программы. Такой подход позволяет сделать библиотеку надёжным инструментом даже в самых требовательных условиях эксплуатации.

Итогом работы стала библиотека, которая обладает значительной практической ценностью и может быть применена в самых разных сценариях. Во-первых, она представляет собой готовый строительный блок, который можно интегрировать в сложные вычислительные системы, требующие высокой производительности и точности — например, в системах для криптографических

вычислений, анализа финансовых данных или моделирования научных процессов. Во-вторых, библиотека служит наглядным примером того, как современные принципы проектирования программного обеспечения могут быть применены на практике. Она демонстрирует, как правильно организовать архитектуру проекта, как использовать передовые возможности языка C++20 (такие как семантика перемещения, RAII и умные указатели), а также как обеспечить удобство сопровождения кода в долгосрочной перспективе. Таким образом, библиотека не только решает конкретные задачи, но и может быть использована в образовательных целях, показывая разработчикам, как эффективно сочетать фундаментальные алгоритмы с современными подходами к разработке программного обеспечения.

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 представляет собой программный модуль, предназначенный для постоянной регистрации событий, происходящих в процессе работы приложения. Он играет ключевую роль по нескольким направлениям:

1. Диагностика и отладка — логгер помогает разработчикам воспроизводить и анализировать ошибки, фиксируя контекст их возникновения;
2. Мониторинг производительности — позволяет отслеживать параметры, такие как задержки и пропускная способность, что критично при эксплуатации системы в реальных условиях;
3. Аудит пользовательской активности — ведет журнал действий пользователей, обеспечивая прозрачность и подотчетность, особенно в системах с повышенными требованиями к безопасности;
4. Анализ после сбоев — в случае критических отказов именно лог-файлы позволяют восстановить последовательность событий и выявить первопричину инцидента.

Кроме того, логгирование важно не только в момент возникновения ошибки, но и как средство для предиктивной аналитики: накопленные данные могут использоваться для выявления аномалий, прогнозирования сбоев и оптимизации поведения приложения.

Способы логирования

Существует множество каналов для вывода логов, каждый из которых подходит для разных этапов жизненного цикла приложения и сценариев эксплуатации.

Самый простой и часто используемый способ — вывод сообщений в стандартные потоки stdout и stderr. Это удобно на этапе разработки и отладки, особенно при запуске приложения в интерактивной среде или контейнерах вроде Docker, где стандартные потоки легко интегрируются с внешними системами логирования.

Для долговременного хранения логов чаще применяются файлы на диске, нередко с поддержкой ротации (удаление или архивирование старых записей) и архивирования — это помогает избежать переполнения хранилища и упрощает управление логами.

В распределенных системах и микросервисной архитектуре сообщения, как правило, отправляются по сети в централизованные хранилища. Это может происходить через такие протоколы, как syslog, HTTP, gRPC или специализированные агенты (например, Fluentd, Logstash). Такой подход позволяет агрегировать логи с множества узлов, обеспечивая их централизованный сбор, поиск и анализ.

Для низкоуровневого или real-time-кода (например, драйверов устройств или компонентов ОС с жёсткими ограничениями по времени отклика) логирование часто реализуется через кольцевой буфер в оперативной памяти, что минимизирует задержки и не зависит от состояния внешней системы ввода-вывода.

Также важную роль играют системные службы логирования, такие как journald (в системах на базе systemd) или Windows Event Log. Они обеспечивают глубокую интеграцию с ОС и поддерживают расширенные функции вроде подписей, фильтрации и разграничения прав доступа.

В конечном счёте, выбор канала логирования — это компромисс между производительностью, надежностью хранения, удобством анализа и возможностями интеграции с существующей инфраструктурой. Например, в критичных к задержке системах предпочтение отдается памяти, в корпоративной среде — централизованным лог-сервисам с отказоустойчивостью и удобным интерфейсом для поиска и оповещений.

Уровни серьезности

Чтобы эффективно управлять потоком логов и не перегружать систему или разработчика избыточной информацией, каждому сообщению присваивается уровень важности (severity level). Это помогает фильтровать сообщения в зависимости от контекста и целей анализа.

Наиболее распространенная шкала уровней логирования выглядит следующим образом:

- trace — максимально подробные сообщения, фиксируют буквально каждый шаг исполнения. Используются для глубокой диагностики и профилирования, чаще всего только на стадии разработки.
- debug — отладочная информация: переменные, промежуточные вычисления, ветвления. Полезна при локализации ошибок.
- info — сообщения об обычном ходе работы приложения: запуск, остановка, успешные операции. Применяется для общего мониторинга.
- warning — указывает на аномалию или потенциальную проблему, которая не мешает работе, но требует внимания (например, превышение лимита, истекающий токен).
- error — ошибка, которая привела к сбою в каком-то компоненте, но система в целом остаётся работоспособной. Такие события важно отслеживать и анализировать.
- fatal (или critical) — критическая ошибка, после которой приложение не может продолжать работу. Такие сообщения, как правило, инициируют аварийное завершение или автоматическое восстановление.

Каждому каналу вывода логов (консоль, файл, сеть и т.д.) можно задать свой порог логирования. Это значит, что:

- Консоль может выводить только сообщения уровня warning и выше, чтобы не засорять экран незначительной информацией;
- Файловый лог — собирать всё, включая trace, для последующего анализа;
- Сетевые агенты — ограничиваться info и выше, чтобы снизить нагрузку на канал передачи данных.

Такой подход обеспечивает гибкость и масштабируемость логирования, позволяя подстроить поведение под конкретные цели: от разработки и тестирования до эксплуатации и инцидент-менеджмента.

Дополнительно, некоторые системы позволяют определять кастомные уровни, а также разделять логи по категориям (тегам или подсистемам) — это даёт ещё более точный контроль над тем, что и куда логируется.

Структурированное логирование

Структурированное логирование — это метод записи логов в виде формализованных и однозначно интерпретируемых данных, например, в формате JSON, XML или других машинно-читаемых представлений. В отличие от обычных текстовых сообщений, которые часто сложно анализировать автоматически, структурированные логи включают четко определенные поля: временные метки, уровень важности, идентификаторы сессий или пользователей, тип события и другие метаданные.

Такой подход позволяет без дополнительных преобразований передавать логи в системы мониторинга, хранилища данных и инструменты анализа. Это особенно важно в масштабируемых и распределенных архитектурах, где быстрое выявление проблемы зависит от возможности задать точный запрос по нужным параметрам.

Например, при сбое можно мгновенно найти все связанные события по request_id, отследить путь запроса через микросервисы, сопоставить его с активностью пользователя и выявить источник ошибки. Кроме того, структурированные логи легко индексируются и визуализируются, что делает их удобным инструментом не только для отладки, но и для построения дашбордов и алертов в реальном времени.

Бинарное дерево поиска

Понятие о бинарном дереве поиска, структура

Бинарное дерево поиска — это базовая структура данных, предложенная в 1960 году Эндрю Дональдом Бутом. Оно представляет собой иерархическую систему узлов, где каждый узел содержит ключ и ассоциированное с ним значение, а также две ссылки на дочерние поддеревья: левое и правое. Ключевое свойство этой структуры заключается в следующем: все ключи в левом поддереве меньше ключа текущего узла, а все ключи в правом поддереве больше.

Благодаря такому порядку элементы в дереве организуются логически, что позволяет выполнять основные операции — такие как поиск, вставка и удаление — за логарифмическое время в среднем случае. Это делает бинарное дерево поиска удобным решением для задач, где требуется частый доступ к данным по ключу с приемлемой скоростью. Однако при неудачном порядке вставки, например в случае уже отсортированных данных, дерево может выродиться в список, и тогда эффективность операций падает до линейной. Чтобы этого избежать, используют самобалансирующиеся варианты, такие как AVL-деревья или красно-черные деревья.

Операции над бинарным деревом

Операция удаления элемента

Операция удаления элемента в бинарном дереве поиска немного сложнее, чем поиск или вставка, так как требует сохранения свойства упорядоченности дерева. В зависимости от количества дочерних узлов у удаляемого узла различают три случая:

1. Если у узла нет потомков (лист) — он просто удаляется, и ссылка на него обнуляется в родительском узле.
2. Если у узла один потомок — он удаляется, а его единственный потомок подставляется на его место, то есть ссылка родительского узла перенаправляется на дочерний узел удаляемого.
3. Если у узла два потомка — это самый сложный случай. Тогда нужно найти либо максимальный элемент в левом поддереве (предшественник), либо минимальный элемент в правом поддереве (преемник), скопировать его значение в удаляемый узел, а затем рекурсивно удалить этот элемент (предшественник или преемник) из соответствующего поддерева. Это сохраняет структуру дерева и его инвариант порядка.

Асимптотическая сложность удаления:

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

Операция поиска 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 во многом повторяет алгоритм поиска места, куда этот элемент должен быть помещён. Алгоритм начинается с корня и на каждом шаге спускается либо в левое, либо в правое поддерево в зависимости от значения ключа. Когда достигается пустое место, создается новый узел, и он добавляется как потомок соответствующего узла.

Алгоритм выглядит так:

1. Начинаем с корня дерева.
2. Сравниваем ключ нового элемента с ключом текущего узла.
3. Если новый ключ меньше — идем в левое поддерево, если больше — в правое.
4. Повторяем шаги, пока не наткнемся на пустое место.
5. В этом месте создаём новый узел и добавляем его в дерево.

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 — это разновидность бинарного дерева поиска с автоматическим балансированием, названная в честь его изобретателей — Г. М. Адельсона-Вельского и Е. М. Ландиса. Основное отличие от обычного бинарного дерева поиска заключается в том, что для каждого узла поддерживается баланс, выраженный как разница высот левого и правого поддеревьев, которая не может превышать 1. Это ограничение позволяет сохранять дерево примерно сбалансированным и, следовательно, высота дерева всегда остаётся порядка логарифма от количества узлов.

Каждый узел AVL-дерева, помимо ключа, значения и ссылок на потомков, содержит поле с высотой — это целое число, определяемое как максимум высот левого и правого поддерева плюс один. Для пустого поддерева высота считается равной 0. Поддержание этой высоты необходимо для быстрой оценки баланса и принятия решения о необходимости поворотов (ротаций) при вставке или удалении узлов, что обеспечивает гарантированную эффективность основных операций — поиска, вставки и удаления.

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 (left rotation) применяется, когда баланс в узле нарушен в сторону правого поддерева: разница высот между левым и правым поддеревьями равна 2, то есть правое поддерево значительно выше. При этом важное условие — высота левого поддерева правого потомка (поддерево C) меньше либо равна высоте правого поддерева правого потомка (поддерево R).

В такой ситуации, чтобы восстановить баланс, выполняется левый поворот вокруг текущего узла. Это означает, что правый дочерний узел становится новым корнем этого поддерева, а исходный узел становится левым потомком нового корня. Поддерево C переходит в правое поддерево исходного узла. В результате высота поддерева снижается, и баланс восстанавливается.

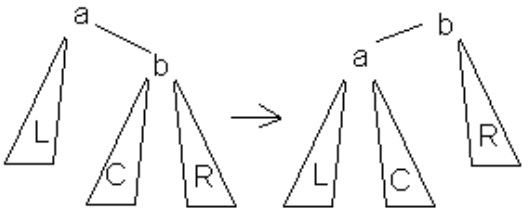

Рисунок 1. Балансировка AVL-дерева малым левым поворотом

Случай 2. Большой левый поворот (или двойной левый поворот) применяется в ситуации, когда баланс узла нарушен в пользу правого поддерева — разница высот между левым и правым поддеревьями равна 2, но при этом высота левого поддерева правого потомка (поддерево C) больше высоты правого поддерева правого потомка (поддерево R).

В этом случае простой левый поворот не исправит дисбаланс, потому что высокий левый ребёнок правого поддерева создаёт дополнительный перекос. Для восстановления баланса сначала выполняется правый поворот вокруг правого дочернего узла (то есть вокруг поддерева C), а затем — левый поворот вокруг исходного узла.

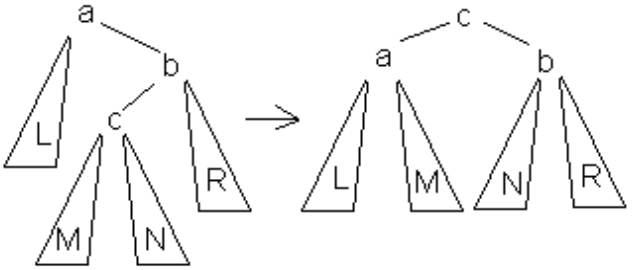

Рисунок 2. Балансировка AVL-дерева большим левым поворотом

Случай 3 — малый правый поворот применяется, когда баланс узла нарушен в пользу левого поддерева, то есть разница высот между правым и левым поддеревьями равна 2, и при этом высота правого поддерева левого потомка (поддерево C) меньше или равна высоте левого поддерева левого потомка (поддерево L).

В такой ситуации для восстановления баланса выполняется правый поворот вокруг текущего узла. Это означает, что левый дочерний узел становится новым корнем поддерева, а исходный узел становится правым потомком нового корня. Поддерево C при этом переходит в левое поддерево исходного узла.

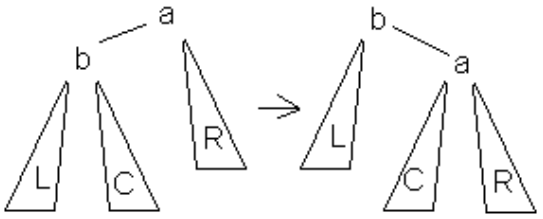

Рисунок 3. Балансировка AVL дерева малым правым поворотом

Случай 4. Большой правый поворот (двойной правый поворот) применяется, когда баланс узла нарушен в пользу левого поддерева — разница высот между правым и левым поддеревьями равна 2, и при этом высота правого поддерева левого потомка (поддерево C) строго больше высоты левого поддерева левого потомка (поддерево L).

В такой ситуации простой правый поворот не восстанавливает баланс, так как высокий правый ребёнок левого поддерева создаёт дополнительный перекос.

Чтобы исправить это, сначала выполняется левый поворот вокруг левого дочернего узла (поддерева C), а затем правый поворот вокруг исходного узла.

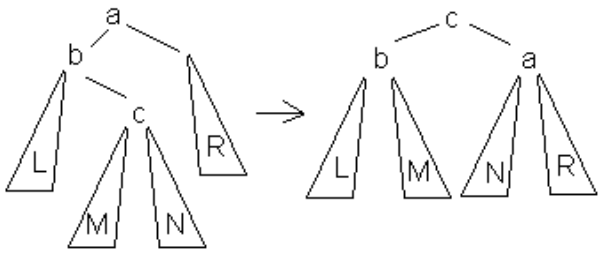

Рисунок 4. Балансировка AVL-дерева большим правым поворотом

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.

Класс длинного целого числа

Сложения длинных целых чисел

1. Числа представляются в виде массивов цифр или блоков фиксированной длины, обычно с младшими разрядами в начале массива;
2. Складываются соответствующие цифры с учётом переноса из предыдущего разряда;
3. При сумме цифр вычисляются новый разряд (остаток от деления на основание) и перенос (целая часть от деления);
4. Если после обработки всех разрядов остался перенос, он добавляется как новый старший разряд;
5. Сложение выполняется за время O(n), где n — количество цифр в большем числе;

Вычитание длинных целых чисел

1. Если числа имеют разные знаки, операция вычитания превращается в сложение с обратным знаком второго числа;
2. Для упрощения алгоритма оба числа сначала приводятся к положительным значениям;

3. Если уменьшаемое оказывается меньше вычитаемого, их меняют местами, при этом результату присваивается отрицательный знак;
4. Вычитание выполняется поразрядно, начиная с младших цифр, учитывая возможный займ из старшего разряда;
5. Когда цифра уменьшаемого меньше соответствующей цифры вычитаемого, происходит займ единицы из следующего 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 $O(n^{log_2 3})$.

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: $X * Y = P_1 * 10^{2k} + P_4 * 10^k + P_2$.

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

Арифметические операции

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:

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

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:

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

Арксинус:

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

Косинус:

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

Арккосинус:

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

Тангенс:

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

Арктангенс:

Если $|x| \leq 1$:

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

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 возведения в степень. Если степень равна 0, возвращается дробь 1/1. В противном случае используется метод бинарного разложения степени для минимизации числа умножений. Создаются две дроби: base (копия исходной дроби) и result (изначально 1/1). Пока степень degree больше 0, алгоритм проверяет младший бит степени: если он 1, result умножается на base. Затем base умножается на себя (возводится в квадрат), а степень делится на 2 (сдвиг вправо). Этот процесс повторяется, пока степень не станет 0.

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 построен как класс-наследник абстрактного logger. Внутри он хранит ассоциативную структуру, где для каждого уровня важности связаны два параметра: флаг, отвечающий за вывод сообщений на консоль, и набор файловых потоков. Файловые потоки обёрнуты в вспомогательный класс refcounted_stream, который ведёт учёт открытых файлов через глобальную таблицу ссылок. При создании нового экземпляра логгера счётчик ссылок на соответствующий файл увеличивается, а при удалении — уменьшается. Когда счётчик достигает нуля, файл автоматически закрывается. Это гарантирует эффективное управление ресурсами, предотвращая повторные открытия одних и тех же файлов и исключая утечки дескрипторов.

Метод log сначала ищет в настройках потоков, назначенных для указанного уровня важности; если для этого уровня ничего не настроено, метод сразу завершает выполнение.

Листинг #1. Функция log

```
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;
}
```

Затем вызывается функция make_format, которая разбирает пользовательский шаблон по символам: "%d" заменяется на текущую дату, "%t" — на время, "%s" — на строковое имя уровня важности, "%m" — на само сообщение. Все символы, не являющиеся флагами, остаются без изменений. Полученная строка выводится в консоль, если вывод для неё активирован, и одновременно записывается во все файлы, открытые для данного уровня важности.

Листинг #2. Функция log

```
std::string client_logger::make_format(const std::string &message, severity sev) const
{
    try {
    std::string result;
    std::time_t now = std::time(nullptr);

    std::tm* tm = nullptr;
    std::tm tm_buf{};
    #ifdef _WIN32
    gmtime_s(&tm_buf, &now);
    tm = &tm_buf;
    #else
    tm = std::gmtime(&now);
    if (!tm) {
```

```
        throw std::runtime_error("Failed to convert time to GMT");
    }
    #endif
    char buffer[80];
    for (size_t i = 0; i < _format.size(); ++i)
    {
        if (_format[i] == '%' && i + 1 < _format.size())
        {
            flag f = char_to_flag(_format[++i]);
            switch (f)
            {
            case flag::DATE:
            std::strftime(buffer, sizeof(buffer), "%Y-%m-%d", tm);
            result += buffer;
            break;
            case flag::TIME:
            std::strftime(buffer, sizeof(buffer), "%H:%M:%S", tm);
            result += buffer;
            break;
            case flag::SEVERITY:
            result += severity_to_string(sev);
            break;
            case flag::MESSAGE:
            result += message;
            break;
            default:
            result += _format[i];
            break;
            }
        }
```

```
            else
            {
                  result += _format[i];
            }
      }
      return result;
      } catch (const std::exception& e) {
      std::cerr << "Error formatting log message: " << e.what() << std::endl;
      return message;
      }
}
```

При создании логгера конструктор получает от строителя уже готовую структуру с потоками и шаблон формата. Внутри он проходит по каждому уровню важности и вызывает метод open у всех refcounted_stream, чтобы убедиться, что файлы действительно открыты и готовы к записи. Сам класс не держит никакого глобального состояния, поэтому операции копирования и перемещения просто переносят внутренние данные — карту потоков и строку формата — без дополнительной обработки.

client_logger_builder отвечает за поэтапное формирование карты потоков. Метод add_file_stream добавляет указанный путь к файлу в список потоков для определённого уровня важности, а add_console_stream устанавливает флаг, разрешающий вывод сообщений в консоль. Конфигурацию при желании можно задать через JSON-файл — метод transform_with_configuration прочитает объект по указанному пути, при наличии поля format задаст шаблон формата, а для каждой секции, соответствующей уровню важности, вызовет вспомогательную функцию parse_severity. Она распределит значения флага console и массива путей paths по внутренним структурам билдера.

Пока логгер используется, открытые файлы разделяются между всеми его копиями. Когда одна из копий уничтожается, счётчик ссылок в refcounted_stream уменьшается, и при достижении нуля соответствующий файловый поток закрывается, а запись в общей таблице удаляется. Это предотвращает дублирование записи в один файл и облегчает контроль за ресурсами.

Таким образом, клиентский логгер сочетает удобный механизм построения конфигурации через паттерн “Строитель”, строгий разбор пользовательского формата и надёжное управление файлами благодаря подсчёту 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.

Листинг #3. Структура node бинарного дерева поиска

```
struct node
{
public:
	value_type data;
	node* parent;
	node* left_subtree;
	node* right_subtree;

	template<class ...Args>
	explicit node(node* parent, Args&& ...args);
	virtual ~node() =default;
};
```

Основной класс бинарного дерева поиска:

Листинг #4. 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 соответствуют стандарту STL и реализуют концепцию std::bidirectional_iterator. Помимо указателя на текущий узел, они содержат дополнительное поле _backup. Это поле служит для сохранения предыдущего посещённого узла, что позволяет при необходимости вернуться назад, например, при вызове операции декремента от итератора end().

Вспомогательный класс:

Листинг #5. 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;

    };

}
```

Вся базовая логика работы с деревом инкапсулирована в классе bst::impl, который служит фундаментом для построения более сложных структур. Он определяет набор методов, формирующих скелет алгоритмов, а конкретные детали могут задаваться в наследуемых классах. Такой подход соответствует шаблону проектирования "template method", позволяя, например, AVL-дереву внедрять собственные правила балансировки. После вставки или поиска могут быть вызваны специальные хуки — post_insert и post_search — чтобы выполнить дополнительные действия. Метод delete_node предусмотрен для корректного освобождения узлов, если их внутренняя структура отличается от классической. Удаление элементов реализовано через функцию erase, в которой сосредоточена основная логика удаления.

Основные операции

Вставка элементов

Вставка элемента выполняется согласно стандартному алгоритму. Сначала происходит поиск подходящего места в дереве для нового элемента. Затем с помощью функции __detail::bst_impl<tkey, tvalue, compare, tag>::create_node() выделяется память и создаётся новый узел. После этого обновляются указатели родительского узла, чтобы корректно связать его с новым элементом.

Листинг #6. Цикл поиска места вставки нового узла

```
if (parent == nullptr) {_root = new_node; }
```

```
else if (compare_keys(new_node->data.first, parent->data.first))
{ parent->left_subtree = new_node; }
else { parent->right_subtree = new_node; }
      ++_size;
      if (_logger)
      {
      _logger->log("Inserted or assigned new node with move semantics", logger::severity::debug);
      }
      __detail::bst_impl<tkey, tvalue, compare, tag>::post_insert(*this, &new_node);
      return infix_iterator(new_node);
```

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.

Листинг #7. Функция поиска элемента find

```
      node* current = _root;
      while (current != nullptr) {
      if (compare::operator()(key, current->data.first)) {
            current = current->left_subtree;
      }
      else if (compare::operator()(current->data.first, key)) {
            current = current->right_subtree;
      }
      else {
            return infix_iterator(current);
```

```
        }
        }
        return end();
}
```

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 сохраняет баланс за счёт отслеживания высот поддеревьев каждого узла. Это позволяет своевременно выполнять повороты при нарушении баланса и поддерживать высоту дерева в пределах O(log n), что обеспечивает быструю работу основных операций — поиска, вставки и удаления.

Основные классы и структуры

Класс AVL_tree

Класс AVL_tree служит центральным компонентом реализации самобалансирующегося дерева. Он расширяет функциональность стандартного бинарного дерева поиска, публично наследуясь от соответствующего базового класса. Все ключевые операции — такие как вставка, удаление и восстановление баланса — переопределены с учётом особенностей AVL-структуры. Чтобы связать реализацию с шаблонным механизмом паттерна "template method", класс параметризуется специальным тегом AVL_TAG. Этот тег используется в bst_impl для выбора нужных версий функций, отвечающих за балансировку и корректную обработку операций в контексте AVL-дерева.

Листинг #8. Структура узла

```
struct node final : public parent::node {
   size_t height;
   void recalculate_height() noexcept;
   short get_balance() const noexcept;
```

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

По сравнению с бинарным деревом поиска, в каждом узле AVL-дерева дополнительно сохраняется информация о высоте, что необходимо для контроля сбалансированности структуры. Метод recalculate_height пересчитывает высоту текущего узла на основе значений его поддеревьев, а get_balance возвращает баланс-фактор — разницу между высотой правого и левого поддеревьев. Эти данные используются для определения необходимости поворотов и выбора подходящего типа балансировки.

Вставка

Вставка элемента в AVL-дерево начинается так же, как и в обычном бинарном дереве поиска — элемент помещается на своё место в соответствии с порядком ключей. После этого запускается процесс восстановления баланса:

1. Сначала происходит подъём вверх от вставленного узла к корню, на каждом шаге вызывается метод recalculate_height для корректного обновления высоты текущего узла;
2. После пересчёта высот определяется баланс-фактор каждого из этих узлов. Если для какого-либо из них модуль баланса превышает 1, запускается соответствующий тип поворота (малый или большой, левый или правый) — в зависимости от того, где был добавлен новый элемент и как это повлияло на структуру дерева.

Удаление

Удаление элемента из AVL-дерева начинается с поиска нужного узла, как и в обычном бинарном дереве поиска — обход дерева от корня с выбором левого или правого поддерева в зависимости от ключа.

Когда нужный узел найден, он удаляется по стандартным правилам бинарного дерева поиска: если у узла два потомка — его значение заменяется на минимум из правого поддерева, затем удаляется этот минимальный узел.

После удаления требуется обновить высоты всех узлов на пути от места удаления до корня. Для этого на каждом шаге вызывается метод пересчёта высоты текущего узла.

Завершается процесс вызовом процедуры ребалансировки — начиная с ближайшего к удалению узла и далее вверх к корню. Если для какого-либо узла баланс нарушен, выполняются соответствующие повороты для восстановления баланса дерева.

Ребалансировка

После вставки или удаления узла происходит непосредственная проверка сбалансированности дерева, начиная с родительского узла модифицированного элемента и далее вверх по дереву до корня. Баланс определяется как разность высот правого и левого поддеревьев. Если модуль этой разности превышает 1, требуется корректировка структуры.

Балансировка выполняется по мере продвижения вверх по дереву. На каждом шаге определяется тип поворота:

1. Правосторонний дисбаланс (баланс > 1):
    a. Если правое поддерево правого потомка "тяжелее" (баланс ≥ 0), выполняется одиночный левый поворот.
    b. Если левое поддерево правого потомка "тяжелее" (баланс < 0), сначала выполняется правый поворот над правым поддеревом, затем — левый поворот над текущим узлом (двойной поворот).
2. Левосторонний дисбаланс (баланс < -1):
    a. Если левое поддерево левого потомка "тяжелее" (баланс ≤ 0), выполняется одиночный правый поворот.
    b. Если правое поддерево левого потомка "тяжелее" (баланс > 0), сначала выполняется левый поворот над левым поддеревом, затем — правый поворот над текущим узлом (двойной поворот).

Обработка всех этих случаев при балансировке выглядит следующим образом:

Листинг #9. Фрагмент post_insert для правостороннего дисбаланса

```
 if (balance > 1) {

                auto *right = static_cast<avl_node
*>(current->right_subtree);

                if (right && right->get_balance() < 0) {
```

```
                node_type *&right_ref = (right->parent->left_subtree ==
right)
                      ? right->parent->left_subtree
                      : right->parent->right_subtree;
                cont.small_right_rotation(right_ref);
                right->recalculate_height();
                }
                cont.small_left_rotation(subtree_ref);
                static_cast<avl_node *>(current)->recalculate_height();
                if (current->parent) {
                static_cast<avl_node
*>(current->parent)->recalculate_height();
                }
                current = static_cast<avl_node *>(subtree_ref);
          } else if (balance < 0) {
```

Итераторы

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

Класс длинного целого числа

Структура представления длинного целого числа

Класс big_int используется для представления длинных целых чисел и содержит два ключевых элемента. Первый — это булева переменная, которая отвечает за знак числа: значение true означает положительное число, false — отрицательное. Второй элемент — это вектор типа std::vector, в котором хранятся коэффициенты числа в виде последовательности беззнаковых целых чисел. Эти коэффициенты формируют само значение числа в некотором фиксированном основании в зависимости от реализации. Кроме того, вектор использует аллокатор, который передаётся объекту big_int через конструктор.

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

Листинг #10. Начало класса big_int

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

Конструкторы

Класс имеет несколько конструкторов для инициализации длинного целого числа из различных источников данных

- 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 определенный аллокатор.
- иные конструкторы

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.

Листинг #11. Функция для сложения в 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(_digits);
    return *this;
}
```

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*]

Листинг #12. Функция для сложения в big_int

```
big_int& big_int::minus_assign(const big_int& other, size_t shift) & {
    if (other._digits.size() == 1 && other._digits[0] == 0) { 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;
    auto cmp = (abs_this <=> abs_other);
    bool result_sign = _sign;
    if (cmp == std::strong_ordering::less) {
        result_sign = !result_sign;
```

```
            std::vector<unsigned int, pp_allocator<unsigned int>>
temp_digits = abs_other._digits;

            abs_other._digits = abs_this._digits;

            abs_this._digits = temp_digits;

      }

      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 diff = borrow;

            if (i < abs_this._digits.size()) {

                  diff += abs_this._digits[i]; }

            if (i < abs_other._digits.size()) {

                  diff -= abs_other._digits[i];

            }

            if (diff < 0) {

                  diff += BASE;

                  borrow = -1;

            } else {

                  borrow = 0;

            }

            result[i] = static_cast<unsigned int>(diff);

      }

      _digits = std::move(result);

      _sign = result_sign;

      optimise(_digits);

      if (is_zero(_digits)) {_sign = true; }

      return *this;

}
```

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".

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.

Листинг #13. Часть функции multiply_assign для тривиального умножения

```
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 product = result._digits[i + j] +
carry;
               if (j < other._digits.size()) {
                    product += static_cast<unsigned long
long>(_digits[i]) * other._digits[j];
               }
               result._digits[i + j] = static_cast<unsigned int>(product
% BASE);
               carry = product / BASE;
          }
     }
```

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

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*. 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.

Операция умножения с использованием алгоритма Карацубы реализована следующим образом: сначала определяется максимальный размер векторов цифр обоих операндов как n, а m вычисляется как n/2 с округлением вверх. Если n меньше или равно 4 (выбрано эмпирически), применяется тривиальное правило умножения. В противном случае цифры обоих операндов разделяются на нижнюю и верхнюю половины (a_low, a_high, b_low, b_high). Если размер вектора цифр меньше или равен m, нижняя половина заполняется текущими цифрами, а верхняя дополняется нулем; иначе нижняя половина берет первые m цифр, а верхняя — оставшиеся.

Далее выполняется оптимизация всех четырех векторов (a_low, a_high, b_low, b_high) для удаления ведущих нулей. Затем вычисляются три произведения: z0 как произведение нижних половин a_low и b_low, z2 как произведение верхних половин a_high и b_high, и z1 как произведение сумм (a_low + a_high) и (b_low + b_high) с последующим вычитанием z0 и z2. Для всех этих умножений используется тривиальное правило, так как предполагается, что размеры операндов уже достаточно малы.

Результат формируется в векторе result размером 2*n, изначально заполненном нулями. Сначала в result записываются цифры z0, начиная с позиции 0. Затем добавляются цифры z1, начиная с позиции m, с учетом переносов: каждая цифра z1 складывается с текущим значением в result[i+m], и если сумма

превышает BASE, перенос распространяется на следующие позиции. Аналогично добавляются цифры z2, начиная с позиции 2*m, также с учетом переносов.

Знак результата определяется как совпадение знаков операндов (_sign == other._sign). Итоговый вектор цифр присваивается _digits, после чего выполняется оптимизация для удаления ведущих нулей. Метод возвращает текущий объект.

Листинг #14. Часть функции multiply_assign для алгоритма Карацубы

```
if (rule == multiplication_rule::Karatsuba) {
	size_t n = std::max(_digits.size(), other._digits.size());
	size_t m = n / 2 + (n % 2);
	if (n <= 4) {
		return multiply_assign(other, multiplication_rule::trivial);
	}
	std::vector<unsigned int, pp_allocator<unsigned int>>
a_high(_digits.get_allocator());
	std::vector<unsigned int, pp_allocator<unsigned int>>
a_low(_digits.get_allocator());
	std::vector<unsigned int, pp_allocator<unsigned int>>
b_high(other._digits.get_allocator());
	std::vector<unsigned int, pp_allocator<unsigned int>>
b_low(other._digits.get_allocator());
	if (_digits.size() <= m) {
		a_low = _digits;
		a_high.push_back(0);
	} else {
		a_low.assign(_digits.begin(), _digits.begin() + m);
		a_high.assign(_digits.begin() + m, _digits.end());
	}
	if (other._digits.size() <= m) {
		b_low = other._digits;
		b_high.push_back(0);
	} else {
		b_low.assign(other._digits.begin(), other._digits.begin() + m);
```

```
            b_high.assign(other._digits.begin() + m, other._digits.end());
      }
      optimise(a_low);
      optimise(a_high);
      optimise(b_low);
      optimise(b_high);
      big_int z0(a_low, true);
      z0.multiply_assign(big_int(b_low, true),
multiplication_rule::trivial);
      big_int z2(a_high, true);
      z2.multiply_assign(big_int(b_high, true),
multiplication_rule::trivial);
      big_int a_sum(a_low, true);
      a_sum += big_int(a_high, true);
      big_int b_sum(b_low, true);
      b_sum += big_int(b_high, true);
      big_int z1(a_sum);
      z1.multiply_assign(b_sum, multiplication_rule::trivial);
      z1 -= z0;
      z1 -= z2;
      std::vector<unsigned int, pp_allocator<unsigned int>>
result(_digits.get_allocator());
      result.resize(n * 2, 0);
      for (size_t i = 0; i < z0._digits.size(); ++i) {
            result[i] = z0._digits[i];
      }
      for (size_t i = 0; i < z1._digits.size(); ++i) {
            unsigned long long temp = static_cast<unsigned long
long>(result[i + m]) + z1._digits[i];
            result[i + m] = static_cast<unsigned int>(temp % BASE);
            if (temp >= BASE) {
                  unsigned long long carry = temp / BASE;
```

```
                size_t j = i + m + 1;
                while (carry > 0 && j < result.size()) {
                unsigned long long next = static_cast<unsigned long
long>(result[j]) + carry;
                result[j] = static_cast<unsigned int>(next % BASE);
                carry = next / BASE;
                j++;
                }
                if (carry > 0) {
                result.push_back(static_cast<unsigned int>(carry));
                }
          }
     }
     for (size_t i = 0; i < z2._digits.size(); ++i) {
          unsigned long long temp = static_cast<unsigned long
long>(result[i + 2*m]) + z2._digits[i];
          result[i + 2*m] = static_cast<unsigned int>(temp % BASE);
          if (temp >= BASE) {
                unsigned long long carry = temp / BASE;
                size_t j = i + 2*m + 1;
                while (carry > 0 && j < result.size()) {
                unsigned long long next = static_cast<unsigned long
long>(result[j]) + carry;
                result[j] = static_cast<unsigned int>(next % BASE);
                carry = next / BASE;
                j++;
                }
                if (carry > 0) {
                result.push_back(static_cast<unsigned int>(carry));
                }
          }
```

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

Целочисленное деление в столбик

Операция деления с использованием алгоритма реализована так: проверяется деление на ноль и обнуление делимого. Берется модуль обоих чисел (abs_this, abs_other). Если abs_this меньше abs_other, результат — ноль. Иначе формируется вектор частного размером (_digits.size() - other._digits.size() + 1) и остаток remainder. Цифры делимого добавляются в remainder справа налево. Для каждой позиции выполняется бинарный поиск q, где remainder >= abs_other * q, с вычитанием temp. Частное записывается в quotient, остаток обновляется. Знак результата — совпадение знаков операндов, оптимизируется и возвращается.

Листинг #15. Функция divide_assign для деления

```
big_int& big_int::divide_assign(const big_int& other, division_rule rule) &
{
    if (is_zero(other._digits)) {
        throw std::logic_error("Division by zero");
    }
    if (is_zero(_digits)) {
        return *this;
    }
    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);
```

```
            _sign = true;
            return *this;
      }
      std::vector<unsigned int, pp_allocator<unsigned int>>
quotient(_digits.size() - other._digits.size() + 1, 0,
_digits.get_allocator());
      big_int remainder(_digits.get_allocator());
      remainder._digits.clear();
      remainder._digits.push_back(0);
      for (int i = static_cast<int>(_digits.size()) - 1; i >= 0; --i)  {
            remainder._digits.insert(remainder._digits.begin(), _digits[i]);
            optimise(remainder._digits);

            if (remainder < abs_other){
                  if (i < static_cast<int>(quotient.size()))
                        quotient[i] = 0;
                  continue;
            }

            unsigned long long left = 0, right = BASE - 1;
            unsigned long long q = 0;
            while (left <= right) {
                  unsigned long long mid = left + (right - left) / 2;
                  big_int temp(abs_other);
                  temp *= big_int(static_cast<long long>(mid),
_digits.get_allocator());
                  if (remainder >= temp) {
                        q = mid;
                        left = mid + 1;
                  } else {
                        right = mid - 1;
```

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

        _digits = std::move(quotient);
        _sign = (_sign == other._sign);
        optimise(_digits);
        if (is_zero(_digits)) {
            _sign = true;
        }

        return *this;
}
```

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*.

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.

Листинг #16. 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;
}
```

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*)

Листинг #17. 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;
}
```

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.

Листинг #18. 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;
```

```
}
```

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.

Листинг #19. Квадратный корень с помощью алгоритма Ньютона

```
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;
}
```

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*.

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 создания была разработана лаконичная, но мощная библиотека, объединяющая алгоритмы и структуры данных, ориентированные на скорость работы и устойчивость к ошибкам в современных задачах. В неё вошли различные поисковые деревья — от простых бинарных до сбалансированных AVL-структур. Для работы с высокой точностью реализован тип для больших целых чисел, который динамически выбирает метод умножения: от традиционного $O(n^2)$ до Карацубы, адаптируясь к объёму данных.

На этом фундаменте построена система рациональных чисел: дроби нормализуются в единую форму, поддерживают весь спектр операций и расширяются базовыми функциями — от тригонометрических до логарифмических. Дополнением служит гибкий инструмент логирования, настраиваемый через паттерн "Строитель": пользователь определяет стиль записей и пути вывода. Логгер эффективно управляет файлами, используя счётчик ссылок для совместного доступа, что исключает дублирование открытых дескрипторов.

Практическая значимость проявляется в двух направлениях. С одной стороны, библиотека готова к использованию в областях, где важны надёжные структуры и точные вычисления, — например, в криптографии, анализе финансов или научных экспериментах. С другой — проект служит примером реализации

базовых алгоритмов на профессиональном уровне: с тестовым покрытием, тщательным контролем ресурсов и настройкой через декларативный подход.

Список использованных источников

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

2. 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.Кнут, Д. Э. Искусство программирования, том 2: Полученные алгоритмы / Д. Э. Кнут. — М.: Вильямс, 2007. — 784 с. — ISBN 978-5-8459-0086-9.

3. Гольдберг, Д. Что каждый программист должен знать о числах с плавающей запятой [Электронный ресурс] // ACM Computing Surveys. — URL: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html (дата обращения: 22.05.2025).

4. Седжвик, Р. Алгоритмы на C++, части 1-4: основы, структуры данных, сортировка, поиск / Р. Седжвик. — 4-е изд. — М.: Диалектика, 2017. — 960 с. — ISBN 978-5-8459-2055-3.

5. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт. — 2-е изд. — СПб.: Невский Диалект, 2005. — 352 с. — ISBN 978-5-94057-128-5.

Приложение А Репозиторий с исходным кодом

FIIT_FA_SP [Электронный ресурс]. // github — URL: https://github.com/DmitriyKolesnikM8O/FIIT_FA_SP (дата обращения: 21.05.2025).