

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего
профессионального образования
«Нижегородский государственный университет им. Н.И. Лобачевского»

Институт информационных технологий, математики и механики

ИССЛЕДОВАТЕЛЬСКАЯ РАБОТА

**«Реализация и исследование структур поиска и изменения данных с
логарифмической сложностью»**

Выполнили: студенты группы 381808-1

Оганян Роберт Владимирович

Рухович Игорь Владимирович

Тронин Дмитрий Валерьевич

Научный руководитель:

Лебедев И. Г.

Нижний Новгород
2020

Оглавление

Введение	4
Постановка задачи	7
Руководство пользователя	8
Добавление нового алгоритма.....	8
Добавление нового теста	8
Добавление нового теста скорости (benchmark).....	9
Руководство программиста.....	12
Структура программы	12
Структура наследования	14
Описание алгоритмов.....	16
Общий принцип работы основных методов	16
Повороты:.....	18
AVL дерево	21
Идея:	21
Основные методы:	22
Сложность:	24
Красно-черное дерево	25
Идея:	25
Основные методы:	26
Сложность:	30
Декартово дерево.....	32
Идея.....	32
Основные операции.....	32
Сложность:	35
Сплей дерево	36
Идея:	36
Основные методы:	40
Сложность:	41
Список с пропусками	42
Идея.....	42
Основные методы:	43
Сложность:	45
Эксперименты.....	46
Выводы из экспериментов:.....	60
Общие выводы:	62

Литература	64
Приложение.....	65
Приложение А.....	65
Приложение В.....	65

Введение

В наши дни информационные технологии занимают важное место не только в специализированных, но и в повседневных сферах жизни. Компьютеры применяются в менеджменте, учебе, бизнесе и многих других сферах деятельности человека.

Разработкой программного обеспечения занимается такая отрасль науки, как программирование. Она становится все более актуальной и приобретает все большее и большее значение в последнее время, ведь с каждым днем компьютер становится все более необходимым и незаменимым.

Таким образом, новые информационные технологии очень актуальны в наше время и нуждаются в большем внимании для последующей разработки и совершенствования. Большое значение имеет также и программирование, одно из фундаментальных разделов информатики, и потому не остающееся в стороне.

Программирование содержит целый ряд важных внутренних задач. Одной из наиболее важных задач для программирования является задача эффективного хранения и использования данных. В большинстве современных информационных систем объемы используемых программ и программных систем могут измеряться не только десятками килобайтов, но и сотнями мегабайтов. Более того, с ростом объема программы удельная стоимость ее создания может нелинейно возрастать. Поэтому к поиску данных в таких информационных системах применяются особые требования, такие как:

- Точность нахождения информации
- Максимальная скорость
- Прогнозируемость времени поиска
- Простые алгоритмы перебора не соответствуют данным требованиям из-за чего разработчики используют алгоритмы поиска с использованием двоичных деревьев.

Бинарное (двоичное) дерево – это динамическая структура данных, представляющее собой дерево, в котором каждая вершина имеет не более двух потомков.

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде.

Добавление элементов в дерево бинарного поиска может сделать его крайне несбалансированным, а иногда даже привести к его вырождению в длинное вытянутое дерево, подобное связному списку.

Проблема этого вырождения заключается не в том, что дерево перестает корректно функционировать (элементы продолжают храниться в отсортированном порядке), а в том, что в данном случае эффективность древовидной структуры ставится под удар. Для идеально сбалансированного дерева (в котором все родительские узлы имеют по два дочерних узла, а все листья размещаются на одном уровне, плюс-минус один) время поиска, время вставки и время удаления соответствуют $O(\log n)$. В вырожденном же дереве его высота будет $O(n)$, и доступ данных значительно замедлится. Иначе говоря, если для выполнения основной операции в дереве с 1000 узлов требуется время, равное t , для ее выполнения в дереве с 1 000 000 узлов потребуется время равное всего лишь $2t$. С другой стороны, время выполнения базовых операций в вырожденном дереве пропорционально $O(n)$, и, следовательно, для выполнения этой же операции в дереве с 1 000 000 узлов потребовалось бы время, равное $1000t$.

Чтобы этого не допустить, разработчики придумали целый ряд различных структур данных (в особенности деревья), самобалансирующиеся таким образом, чтобы основные операции работали за логарифм времени.

Основным преимуществом двоичного дерева поиска перед другими структурами данных является возможная высокая эффективность реализации основанных на нём алгоритмов поиска и сортировки. Двоичное дерево поиска применяется для построения более абстрактных структур, таких, как, например, множества (set), мультимножества (multiset), ассоциативные массивы (map, multimap).

Цели исследовательской работы:

- Систематизация, углубление и применение знаний по программированию в среде C++.
- Реализация следующих структур данных, работающих за логарифмическую сложность:
 - AVL tree
 - Splay tree
 - Cartesian tree (Декартово дерево)
 - Red-Black tree
 - Skip-list
- Реализация системы тестирования этих структур данных
- Сравнение эффективности реализованных структур данных и анализ полученных результатов

Постановка задачи

Требуется реализовать библиотеку BST, содержащая реализацию пяти структур данных с логарифмическим временем поиска, добавления и удаления элементов. Библиотека должна включать в себя следующие структуры данных:

- AVL tree (AVL дерево)
- Splay tree (Сплэй дерево)
- Cartesian tree (Декартово дерево)
- Red-Black tree (Красно-черное дерево)
- Skip-list (Список с пропусками)

Также требуется разработать интерфейс, позволяющий при создании объекта выбрать алгоритм работы. Написанная структура должна позволять помимо уже имеющихся использовать любые другие реализованные пользователем алгоритмы, соответствующие некоторым требованиям.

К библиотеке нужно реализовать универсальные тесты и провести эксперименты на производительность разных функций реализованных алгоритмов на множестве различных наборов данных. Полученные результаты нужно отобразить графически, провести анализ и сделать выводы.

Руководство пользователя

Добавление нового алгоритма

Чтобы добавить свою реализацию бинарного дерева в библиотеку необходимо:

- В папке trees завести свой .h файл с исходным кодом алгоритма.
- В файле CMakeLists.txt добавить к библиотеке TREES свой .h файл.
- Исходный код алгоритма должен представлять собой класс, который:
- Имеет какой-нибудь конструктор
- Является наследником класса `ITree<T>`
- Определяет все абстрактные методы класса `ITree<T>` (описаны в `abstract_tree.h`)
- Имеет класс, реализующий итерацию, наследник класса `ITree<T>::ITreeItImpl`
- Определяет все абстрактные методы класса `ITree<T>::ITreeItImpl`

Ожидаемое поведение всех функций описано в `abstract_tree.h`.

Теперь можно использовать свою реализацию бинарного дерева. Для создания экземпляра класса необходимо инстанцировать объект своего класса с произвольным типом T, получить указатель на этот объект и привести его к указателю на `ITree<T>`. Рекомендуется делать это с помощью `std::shared_ptr`. Пример:

```
std::shared_ptr<ITree<T>> myInstance = std::make_shared<MyClass<T>>(args),
```

где MyClass – название вашего класса, args – аргументы конструктору.

Такая запись не требует особых операций по очистке памяти после использования.

Добавление нового теста

Чтобы добавить свой тест для одного/всех имеющихся алгоритмов необходимо:

- В файле `full_test_set.h` добавить функцию вида `void MyTest(ImplType)` с любым именем, принимающую на вход объект `ImplType`, который указывает, какой алгоритм стоит тестировать сейчас.

- Использовать синтаксис библиотеки Catch2 для проверок. Например, REQUIRE(condition) требует соблюдения condition для успешного прохождения, REQUIRE_NOTHROW(expression) требует, чтобы выполняемая операция не генерировала ошибку, а REQUIRE_THROWS_AS(expression, exType) – чтобы выполняемое выражение генерировало ошибку типа exType. Подробное описание всех методов можно найти на официальном сайте Catch2.
- Использовать функции MakeTree и MakeCopyAssignment для создания экземпляра класса.
- Добавить свой тест в тестирующую среду – в конструкторе класса TestFramework выполнить tests_.emplace("test_name", MyTest), где первый аргумент это название теста, а второй – сама функция теста.

Чтобы тестировать свой алгоритм необходимо:

- В файле full_test_set.h расширить enum class ImplType названием своего алгоритма.
- В этом же файле добавить по аналогии с уже имеющимися алгоритмами свой в функции MakeTree и MakeCopyAssignment.
- Добавить свой алгоритм в тестирующую среду – в конструкторе класса TestFramework в файле test_framework.cpp выполнить types_.emplace("type_name", ImplType::MyClass), где первый аргумент это название алгоритма, а второй – сама функция теста.
- Скомпилировать и запустить файл run_all_tests и увидеть в stdout результаты прохождения каждого теста.

Добавление нового теста скорости (benchmark).

Чтобы добавить свой тест скорости для одного/всех имеющихся алгоритмов необходимо:

- В файле benchmarks.h добавить функцию вида double MyTest(ImplType, std::mt19937&, uint64_t) с любым именем, принимающую на вход объект ImplType, который указывает, какой алгоритм стоит тестировать сейчас, ГПСЧ

в виде Вихря Мерсенна и целое положительное число, означающее количество операций, скорость которых нужно измерить сейчас. Функция возвращает число типа double – время работы теста в миллисекундах.

- Использовать функции MakeTree для создания экземпляра класса.
- Добавить свой тест в тестирующую среду – в конструкторе класса BenchFramework выполнить `benchmarks_.emplace("test_name", MyTest)`, где первый аргумент это название теста, а второй – сама функция теста.

Чтобы тестировать свой алгоритм необходимо:

- В файле benchmarks.h расширить enum class ImplType названием своего алгоритма.
- В этом же файле добавить по аналогии с уже имеющимися алгоритмами свой в функцию MakeTree.
- Добавить свой алгоритм в тестирующую среду – в конструкторе класса BenchFramework в файле bench_framework.cpp выполнить `types_.emplace("type_name", ImplType::MyClass)`, где первый аргумент это название алгоритма, а второй – сама функция теста.
- Скомпилировать и запустить файл run_benchmarks и увидеть в сгенерированных файлах с данными время выполнения каждого теста на каждом алгоритме.

Пример применения библиотеки для решения прикладных задач
Вася решил начать изучать английский язык. Для этого он загрузил на компьютер книгу на английском и решил выучить из неё все слова. Вася разработал план на Q дней для изучения новых английских слов. Если бы все слова были записаны в лексикографическом порядке, то в i -й день Вася хотел бы выучить все слова, начиная с x_i и до y_i . Помимо этого Вася может найти в интернете новые слова и добавить их в план. Вася поделился с вами своим планом, помогите ему выбрать нужные слова на каждый день.

В первой строке записано единственное число Q , $1 \leq Q \leq 10^6$ – количество дней, в течение которых Вася будет учить английский язык. Следующие $2Q$ строк выглядят так:

Строка с номером $2k - 1$ содержит новые слова, которые Вася нашёл в интернете в k -й день и решил добавить в план и заканчивается ‘.’.

Строка с номером $2k$ содержит 2 слова x_i и y_i , x_i лексикографически меньше y_i .

Нумерация строк начинается с 1. Все слова состоят из строчных латинских букв. Суммарное количество слов за все дни не превышает 10^7 , суммарное количество слов, которые выучит Вася не превышает 10^6 .

Выведите Q строк, таких, что i -я строка содержит слова, которые Вася будет учить в i -й день.

```
tronindmitr@DESKTOP-Q1HPRSD:/mnt/f/Проект/trees/cmake-build-release-wsl$ ./task
3
apple banana .
apple apple
apple
grape orange .
banana grape
banana grape
cherry watermelon .
cherry watermelon
cherry grape orange watermelon
```

Рисунок 1.1. Скриншот работы программы

Как видим, программа успешно компилируется и работает корректно.

Исходный код решения задачи находится в Приложении А.

Руководство программиста

Структура программы

Структура проекта выглядит следующим образом:

benchmarks	It is possible to choose the max number of threads now. All tests have been ...	13 hours ago
catch	Added Catch.hpp. Now it works	3 months ago
experiments	It is possible to choose the max number of threads now. All tests have been ...	13 hours ago
tests	Removed redundant files. Signed almost all functions	23 hours ago
trees	Abstract tree functions are named	22 hours ago
.clang-format	moved files from /linter to root, pipeline works now	4 months ago
.gitignore	Sply tree rewritten but still doesnt work	5 days ago
.gitlab-ci.yml	Trees & tests refactoring	1 month ago
CMakeLists.txt	Removed redundant files. Signed almost all functions	23 hours ago
README.md	Add README.md	20 hours ago

Рисунок 2.1. Скриншот GitLab со всеми файлами проекта.

Папка benchmarks содержит тестирование по времени для написанных структур данных

bench_framework.cpp	It is possible to choose the max number of threads now. All tests have...	13 hours ago
benchmarks.h	Removed redundant files. Signed almost all functions	23 hours ago
run_benchmarks.cpp	It is possible to choose the max number of threads now. All tests have...	13 hours ago

Рисунок 2.2. Скриншот содержимого папки benchmarks.

Папка experiments содержит результаты benchmarks, Experiments.ipynb содержит графики, построенные по результатам (по файлам с расширение csv).

..		
multithreaded	It is possible to choose the max number of threads now. ...	13 hours ago
!_converging_int_series_erase_after_increasi...	It is possible to choose the max number of threads now. ...	13 hours ago
!_converging_int_series_erase_after_random...	It is possible to choose the max number of threads now. ...	13 hours ago
!_converging_int_series_insert_bench.csv	It is possible to choose the max number of threads now. ...	13 hours ago
!_decreasing_int_series_erase_after_increasi...	It is possible to choose the max number of threads now. ...	13 hours ago
!_decreasing_int_series_erase_after_random...	It is possible to choose the max number of threads now. ...	13 hours ago
!_decreasing_int_series_insert_bench.csv	It is possible to choose the max number of threads now. ...	13 hours ago
!_diverging_int_series_erase_after_increasin...	It is possible to choose the max number of threads now. ...	13 hours ago
!_diverging_int_series_erase_after_random...	It is possible to choose the max number of threads now. ...	13 hours ago
!_diverging_int_series_insert_bench.csv	It is possible to choose the max number of threads now. ...	13 hours ago

Рисунок 2.3.1 Скриншот папки experiments с результатами тестирования на время.

!_random_int_series_erase_after_increasing...	It is possible to choose the max number of threads now. ...	13 hours ago
!_random_int_series_erase_after_random_s...	It is possible to choose the max number of threads now. ...	13 hours ago
!_random_sparse_int_series_insert_bench.csv	It is possible to choose the max number of threads now. ...	13 hours ago
!_random_sparse_strings_insert_bench.csv	It is possible to choose the max number of threads now. ...	13 hours ago
!_random_strings_erase_after_random_inse...	It is possible to choose the max number of threads now. ...	13 hours ago
Experiments.ipynb	It is possible to choose the max number of threads now. ...	13 hours ago
some_text.txt	Ended up with benchmarks. Number of threads is limited...	1 day ago

Рисунок 2.3.2 Скриншот папки experiments с результатами тестирования на время.

Папка catch содержит библиотеку catch

"		
h++ catch.hpp	Added Catch.hpp. Now it works	3 months ago

Рисунок 2.4 Скриншот папки catch.

Папка tests содержит тестирование на правильность реализованных структур данных. Тесты подписаны.

h full_test_set.h	Removed redundant files. Signed almost all functions	23 hours ago
C++ run_all_tests.cpp	Removed redundant files. Signed almost all functions	23 hours ago
C++ run_tests_separately.cpp	Removed redundant files. Signed almost all functions	23 hours ago
C++ test_framework.cpp	Removed redundant files. Signed almost all functions	23 hours ago

Рисунок 2.5 Скриншот папки tests с тестами на корректную работу.

Папка trees содержит реализованные структуры данных

h abstract_tree.h	Abstract tree functions are named	22 hours ago
h avl_tree.h	v15 Removed a lot of Recursives, Changed skiplist	1 week ago
h cartesian_tree.h	Find and LowerBound are non-recursive now (in Cartesian)	1 week ago
h rb_tree.h	v15 Removed a lot of Recursives, Changed skiplist	1 week ago
h skip_list.h	Resolved Skip Lists errors	6 days ago
h splay_tree.h	Added stdlib++ set as a tree	5 days ago
h stdlib_set.h	Added stdlib++ set as a tree	5 days ago

Рисунок 2.6 Скриншот папки trees с реализованными структурами данных.

Структура наследования

Реализованные структуры данных (avl_tree.h, cartesian_tree.h, rb_tree.h, skip_list.h, splay_tree.h) наследуются от родительского класса abstract_tree.h следующим образом:

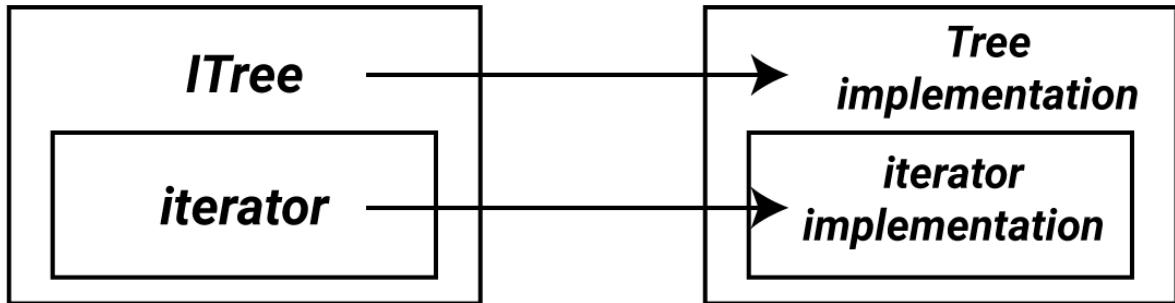


Рисунок 2.7 Схема наследования от родительского класса abstract_tree.h.

Все методы родительского класса abstract_tree.h переопределяются в дочерних классах, так что достаточно будет описать их в родительском классе.

Основные Методы abstract_tree.h:

- virtual std::shared_ptr<ITreeItImpl> Clone() = 0; - Возвращает копию хранимого итератора
- virtual void Increment() = 0; - Двигает итератор на следующий элемент
- virtual void Decrement() = 0; - Двигает итератор на предыдущий элемент
- virtual const T Dereferencing() const = 0; - Разыменование итератора, возвращает значение элемента, на который указывает итератор
- virtual const T* Arrow() const = 0; - Возвращает указатель на значение элемента, на который указывает итератор
- virtual bool IsEqual(std::shared_ptr<ITreeItImpl> other) const = 0; - Проверяет равны ли 2 итератора между собой
- virtual std::shared_ptr<ITreeItImpl> Begin() const = 0; - Возвращает указатель на итератор на начало, на самый левый элемент в дереве
- virtual std::shared_ptr<ITreeItImpl> End() const = 0; - Возвращает указатель на итератор на конец, на элемент, следующий после последнего элемента дерева

- `[[nodiscard]] virtual size_t Size() const = 0;` - Возвращает размер, количество элементов в дереве
- `[[nodiscard]] virtual bool Empty() const = 0;` - Возвращает пустое ли дерево
- `virtual std::shared_ptr<ITreeItImpl> Find(const T& value) const = 0;` - Поиск элемента в дереве. Возвращает указатель на итератор на найденный элемент, либо возвращает указатель на итератор на следующий после последнего элемент.
- `virtual std::shared_ptr<ITreeItImpl> LowerBound(const T& value) const = 0;`
 - Поиск нижнего предела для элемента в дереве. Возвращает указатель на итератор на элемент, который следовал бы в дереве после `value`.
- `virtual void Insert(const T& value) = 0;` - Вставка элемента в дерево
- `virtual void Erase(const T& value) = 0;` - Удаление элемента из дерева
- `virtual void Clear() = 0;` - Очистка дерева, освобождение памяти и удаления всех элементов дерева.

Описание алгоритмов

Общий принцип работы основных методов

В основе бинарных поисковых деревьев лежат такие операции, как:

- Поиск (Find)
- Вставка (Insert)
- Удаление (Erase)
- Слияние (Merge)
- Разделение (Split)

Поиск:

Поиск в бинарном дереве происходит по следующим правилам:

- Если дерево пусто, то сообщить, что ключ не найден.
- Иначе сравнить требуемое значение со значением в корне
 - Если значение в вершине больше, то повторно запустить поиск в левом поддереве
 - Если значение в вершине меньше, то повторно запустить поиск в правом поддереве
 - Иначе текущая вершина – искомая

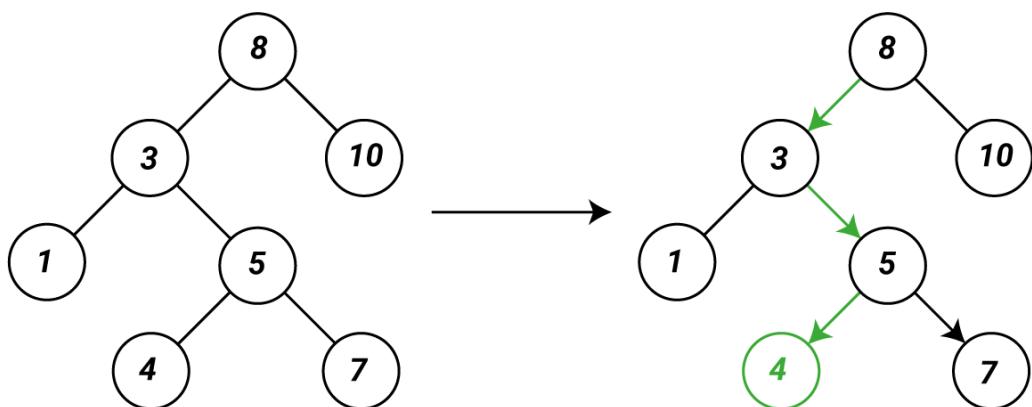


Рисунок 3.1. Иллюстрация работы метода поиска.

Вставка:

Вставка работает аналогично поиску, только при нахождении пустого потомка подвешивается новая вершина и, возможно, выполняется балансировка.

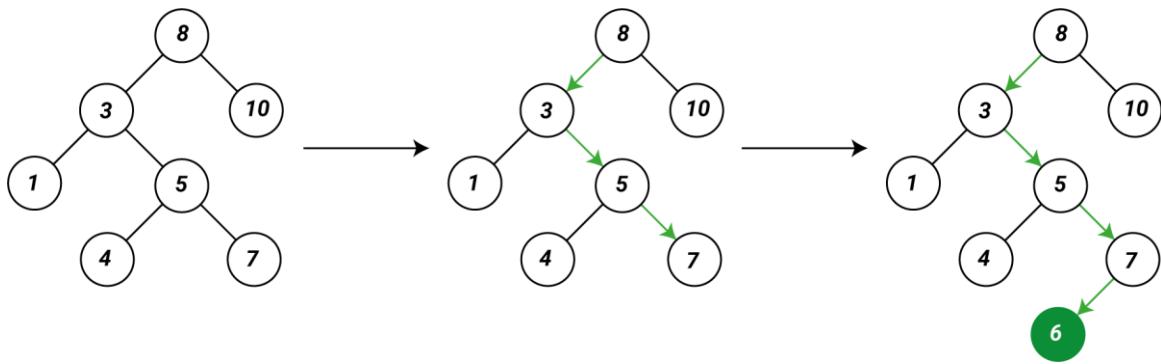


Рисунок 3.2. Иллюстрация работы метода добавления новой вершины.

Удаление:

Находим удаляемую вершину с помощью `find`.

- Если удаляемая вершина – лист (не имеет потомков), то просто меняем указатель у родительской вершины, который указывал на удаляемую вершину, на нулевой указатель.
- Если у удаляемой вершины 1 потомок, то меняем указатель у родительской вершины, который указывал на удаляемую вершину, на этого единственного потомка.
- Если у удаляемой вершины есть 2 потомка, то ищем следующий элемент и ставим его на место удаляемого.

Далее, в зависимости от реализации может быть выполнена балансировка.

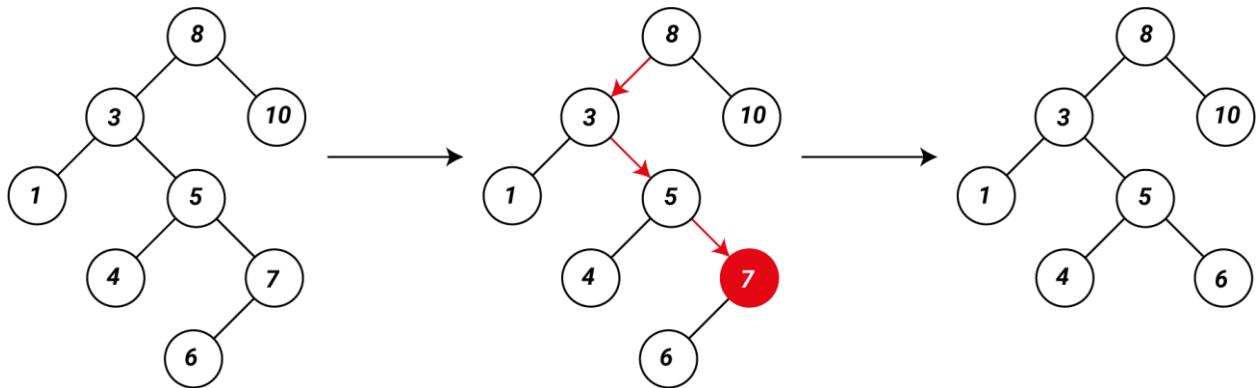


Рисунок 3.3. Иллюстрация работы метода удаления вершины.

Разделение:

$\text{Split}(k)$ делит дерево по ключу k , возвращая левое поддерево T_1 , правое поддерево T_2 и сам ключ k , если он имелся в исходном дереве. При этом для всех ключей выполняется $T_1 < k < T_2$.

Алгоритм работы:

Начинаем из корня дерева.

Если k больше корня, то выполняется операция Split для правого поддерева (T_2), которая возвращает поддеревья T_2^L и T_2^R . Тогда результат исходной операции Split это поддерево T_2^R в качестве правого и вся остальная часть дерева кроме вершины k в качестве левого поддерева.

Аналогично алгоритм выполняется, если k меньше корня.

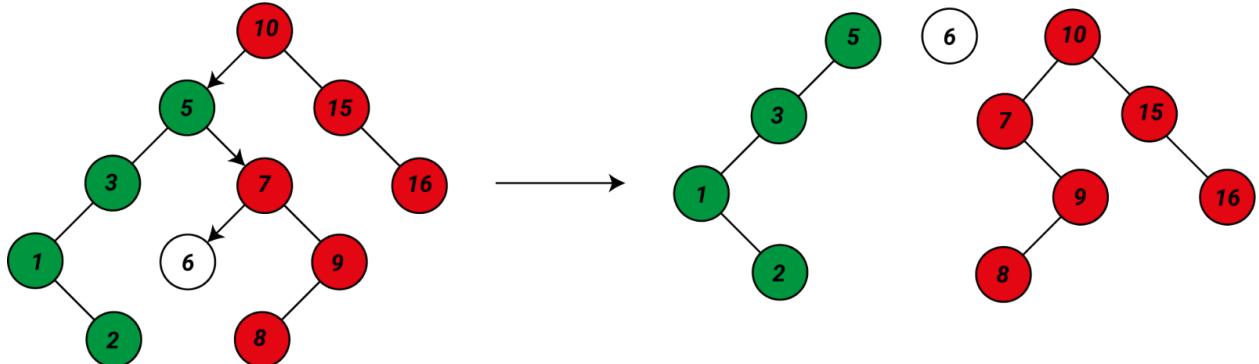


Рисунок 3.4. Иллюстрация работы метода разделения.

Слияние:

Принимает на вход 2 сбалансированных дерева, таких что в первом все ключи меньше всех ключей во втором, и возвращает одно сбалансированное дерево, состоящее из всех ключей данных деревьев.

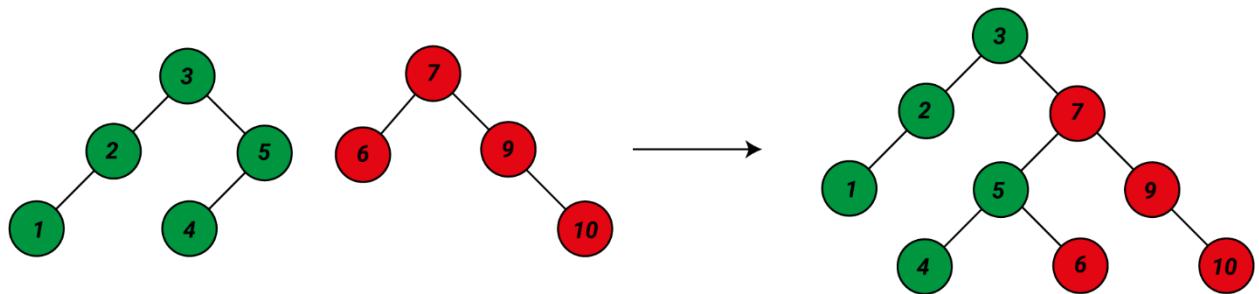


Рисунок 3.5. Иллюстрация работы метода слияния.

Повороты:

В основе всех балансировок лежат левое вращение, правое вращение и их комбинации. Рассмотрим следующий рисунок. На этом рисунке треугольники представляют дочерние деревья, которые могут содержать любое количество узлов

(хоть ноль или больше узлов) – их количество не влияет на работу алгоритма поворота.

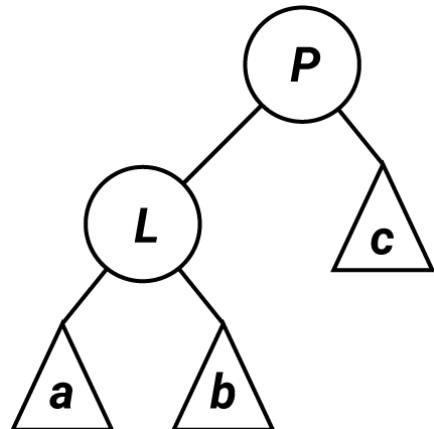


Рисунок 3.6. Дерево до поворота.

Рассмотрим узел “L”. Из определения бинарного дерева поиска, он больше всех узлов в его левом дочернем дереве и меньше всех узлов в правом дочернем дереве. Также так как он является левым дочерним узлом, его родительский узел больше всех узлов в его правом дочернем дереве (в поддереве ‘b’). Следовательно, если повернуть левый дочерний узел в позицию его родительского узла так, чтобы его правое дочернее дерево стало новым левым дочерним деревом родительского узла, получившееся бинарное дерево не перестает удовлетворять свойствам бинарного дерева поиска.

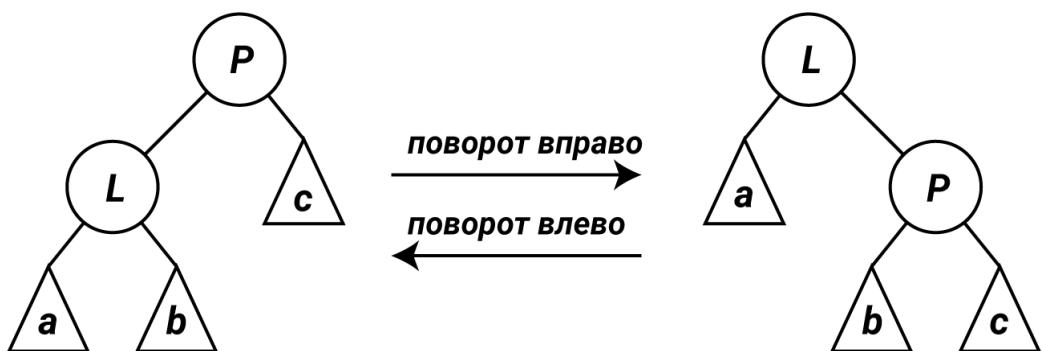


Рисунок 3.7. Иллюстрация поворота.

Для исходного дерева можно было бы записать следующее неравенство: $(a < L < b) < P < c$. Для нового дерева имеем: $a < L < (b < P < c)$. Операция $<$

коммуникативная, поэтому неравенство остаётся справедливым и при удалении круглых скобок.

Рассмотренную операцию называют *right rotation* (поворот вправо). При этом ранг левого дочернего узла L повышается, а ранг родительского узла P понижается. То есть, узел L перемещается на один уровень вверх, а узел P – на один уровень вниз. Такой поворот называется поворотом вокруг узла P .

Аналогично существует симметричный поворот *left rotation* (поворот влево). При повороте влево узел P перемещается на один уровень вверх, а узел L – на один уровень вниз.

АВЛ дерево

АВЛ — аббревиатура, образованная первыми буквами фамилий создателей (советских учёных) Георгия Максимовича Адельсон-Вельского и Евгения Михайловича Ландиса.

Идея:

В АВЛ дереве для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

Лемма:

Пусть m_h — минимальное число вершин в АВЛ дереве высоты h , тогда $m_h = F_{h+2} - 1$, где F_h - h -ое число Фибоначчи.

Доказательство:

Пусть m_h - минимальное число вершин в АВЛ дереве высоты h . Тогда легко заметить, что $m_{h+2} = m_{h+1} + m_h + 1$. Докажем равенство $m_h = F_{p+2} - 1$ по индукции.

База индукции $m_1 = F_3 - 1$ — верно, $m_1 = 1$, $F_3 = 2$.

Допустим, что $m_h = F_{h+2} - 1$ верно.

Тогда

$$m_{h+1} = m_h + m_{h-1} + 1 = F_{h+2} - 1 + F_{h+1} - 1 + 1 = F_{h+3} - 1$$

Значит, $m_h = F_{h+2} - 1$ — доказано.

Теорема:

Высота АВЛ дерева с n ключами имеет высоту $h = O(\log N)$.

Доказательство:

Пусть $h(x)$ — высота поддерева с корнем в x .

$$F_h = \Omega(\varphi^h), \text{ где } \varphi = \frac{\sqrt{5}+1}{2}$$

Таким образом по лемме:

$$n \geq \varphi^h$$

Логарифмируя по φ получаем:

$$n \geq h$$

Таким образом получаем, что высота АВЛ дерева - $O(\log n)$.

Основные методы:

Поиск:

Поиск работает так же, как описано в общем случае. Очевидно, что в худшем случае он пройдет высоту дерева, а значит по доказанному выше, сложность этого метода $O(\log n)$.

Вставка:

Вставка работает так же, как описано в общем случае. Затем для каждой вершины на пути добавленной вершины до корня вызывается функция балансировки.

Удаление:

Удаление работает так же, как описано в общем случае. Затем от родителя удаленной вершины и до корня для каждой вершины вызывается функция балансировки.

Функция балансировки:

Первым делом пересчитывается балансировочный фактор – разность высот левого и правого поддеревьев, который вычисляется на основе высот вершин.

Рассмотрим ситуацию, когда балансировочный фактор равен 2 (или -2 ведь эти ситуации аналогичны в точности до симметрии).

Пусть q — правый дочерний узел узла p , а s — левый дочерний узел узла q и $h(a) - h(q) = 2$.

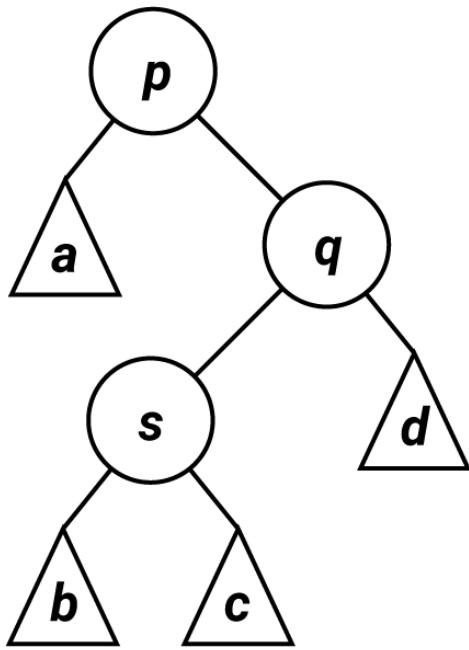


Рисунок 4.1. Иллюстрация дисбаланса.

Легко заметить, что для исправления дисбаланса в узле p достаточно выполнить поворот влево вокруг вершины p , либо большой поворот так же вокруг p .

Если высота левого поддерева узла q меньше либо равна высоте его правого поддерева: $h(s) \leq h(d)$, то выполняется простой поворот.

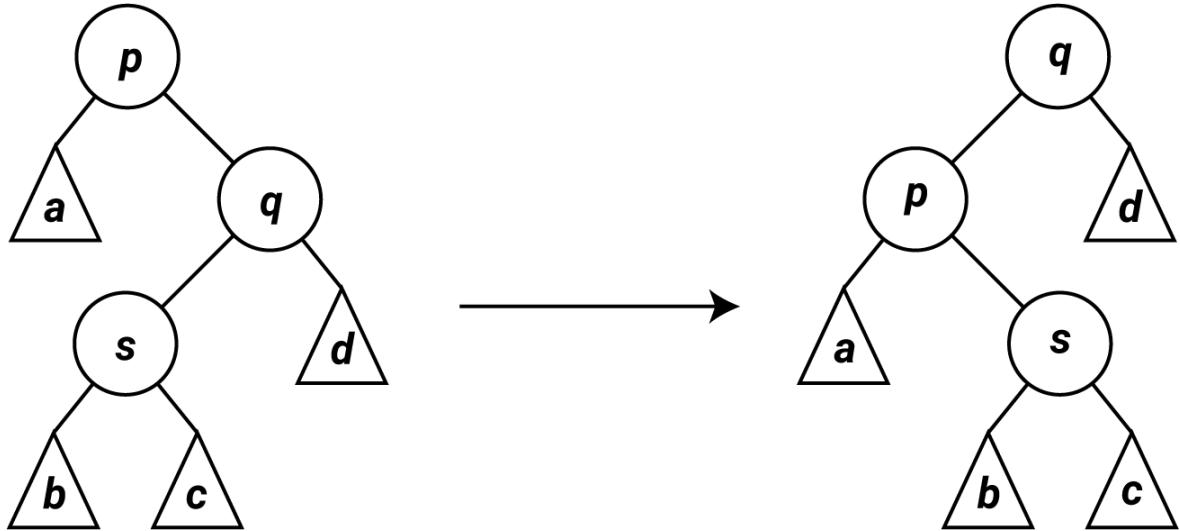


Рисунок 4.2. Иллюстрация работы функции балансировки АВЛ дерева ($h(\square) \leq h(\square)$).

Иначе ($h(s) > h(d)$) применяется большой поворот, который к двум простым — сначала правый поворот вокруг q и затем левый вокруг p .

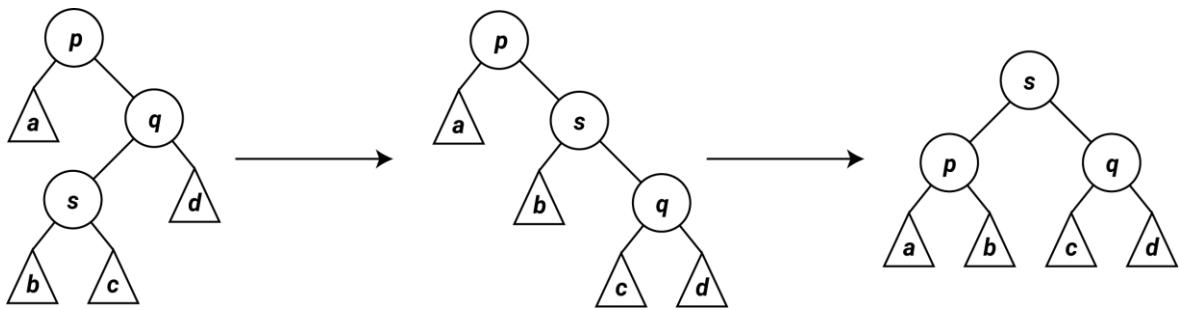


Рисунок 4.3 Иллюстрация работы функции балансировки АВЛ дерева ($h(\square) > h(\square)$).

Таким образом в дереве устраняются все дисбалансы.

Сложность:

Добавление:

Так как в процессе добавления вершины мы рассматриваем не более, чем $O(h)$ вершин дерева, и для каждой запускаем балансировку не более одного раза, то суммарное количество операций при включении новой вершины в дерево составляет $O(\log n)$ операций.

Удаление:

Так как на удаление вершины и балансировку суммарно тратится, как и ранее, $O(h)$ операций. Таким образом, требуемое количество действий — $O(\log n)$.

Название операции	В среднем	В худшем случае
Поиск	$O(\log n)$	$O(\log n)$
Удаление	$O(\log n)$	$O(\log n)$
Вставка	$O(\log n)$	$O(\log n)$

Рисунок 4.4 Таблица асимптотики для AVL дерева.

Красно-чёрное дерево

Изобретателем красно-чёрного дерева считают немца Рудольфа Байера. Название «красно-чёрное дерево» структура данных получила в статье Л. Гимбаса и Р. Седжвика (1978). По словам Гимбаса, они использовали ручки двух цветов. По словам Седжвика, красный цвет лучше всех смотрелся на лазерном принтере.

Красно-чёрные деревья являются одними из наиболее активно используемых на практике самобалансирующихся деревьев поиска. В частности, контейнеры `set` и `map` в большинстве реализаций библиотеки STL языка C++, класс `TreeMap` языка Java, так же, как и многие другие реализации ассоциативного массива в различных библиотеках, основаны на красно-чёрных деревьях.

Идея:

Красно-чёрное дерево – это двоичное дерево поиска вершины которого разделены на красные и черные. При чем должны соблюдаться определенные требования, чтобы дерево можно было называть сбалансированным.

Свойства красно-чёрного дерева:

- Каждый лист – черный
- Не может подряд идти 2 красные вершины
- Все простые пути из любого узла x до листьев содержат одинаковое количество чёрных узлов (совпадает их черная высота)

Высота красно-чёрного дерева:

Теорема:

Красно-чёрное дерево с внутренними вершинами (т. е. не считая фиктивных листьев) имеет высоту не более $2\lg(n + 1)$.

Доказательство:

Докажем, что дерево с корнем в x содержит по меньшей мере $2^{bh(x)} - 1$ внутренних вершин. Докажем это с помощью индукции от листьев к корню. Для листьев черная высота равна 0, и значит поддерево действительно содержит по меньшей мере $2^{bh(x)} - 1 = 2^0 - 1 = 0$ внутренних вершин. Пусть теперь вершина x

не является листом и имеет черную высоту k . Оба ее ребенка имеют черную высоту не меньше $k - 1$ (k – если красный, $k - 1$ – если черный). По предположению индукции левое и правое поддеревья вершины x содержит не менее $2^{k-1} - 1 + 2^{k-1} - 1 + 1 = 2^k - 1$ внутренних вершин.

Теперь обозначим высоту дерева за h . Из-за того, что 2 красные вершины не могут идти подряд, то по крайней мере половина всех вершин на пути от корня к листу, не считая корень – черные вершины. Следовательно черная высота дерева не меньше $h/2$. Тогда:

$$n \geq 2^{\frac{h}{2}} - 1$$

$$n + 1 \geq 2^{\frac{n}{2}}$$

$$\lg \lg (n + 1) \geq h/2$$

$$h \leq 2\lg(n + 1)$$

Что и требовалось доказать.

Основные методы:

Поиск:

Поиск работает так же, как описано в общем случае. Очевидно, что в худшем случае он пройдет высоту дерева, а значит по доказанному выше, сложность этого метода $O(\log n)$.

Добавление:

Добавление вершины работает так же, как описано в общем случае. Отметим, что добавляемая вершина всегда красная. Затем вызывается функция балансировки.

В ней в точности до симметрии рассматриваются несколько возможных вариантов:

Случай 1:

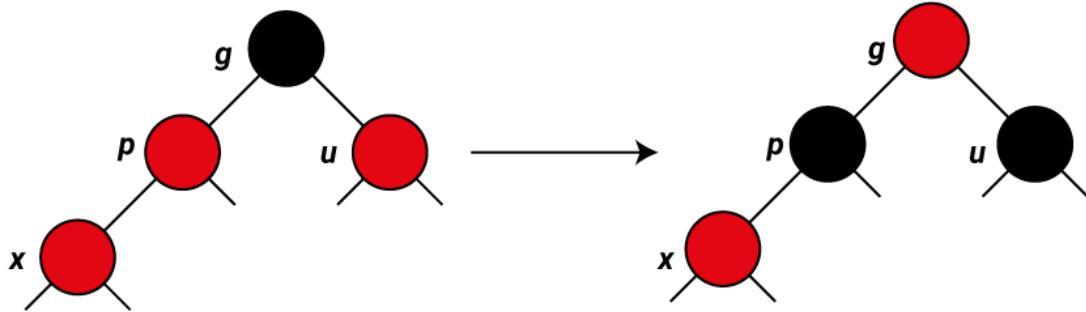


Рисунок 5.1.1. Иллюстрация первого случая функции балансировки после добавления вершины.

Отец (вершина p) и дядя (вершина u) вершины красные. Тогда мы можем спустить цвет с дедушки (вершина g) вниз. Черная высота останется неизменной. Однако возможно нарушение 2 правила для дедушки, поэтому повторно запускаем функцию балансировки для этой вершины.

Случай 2:

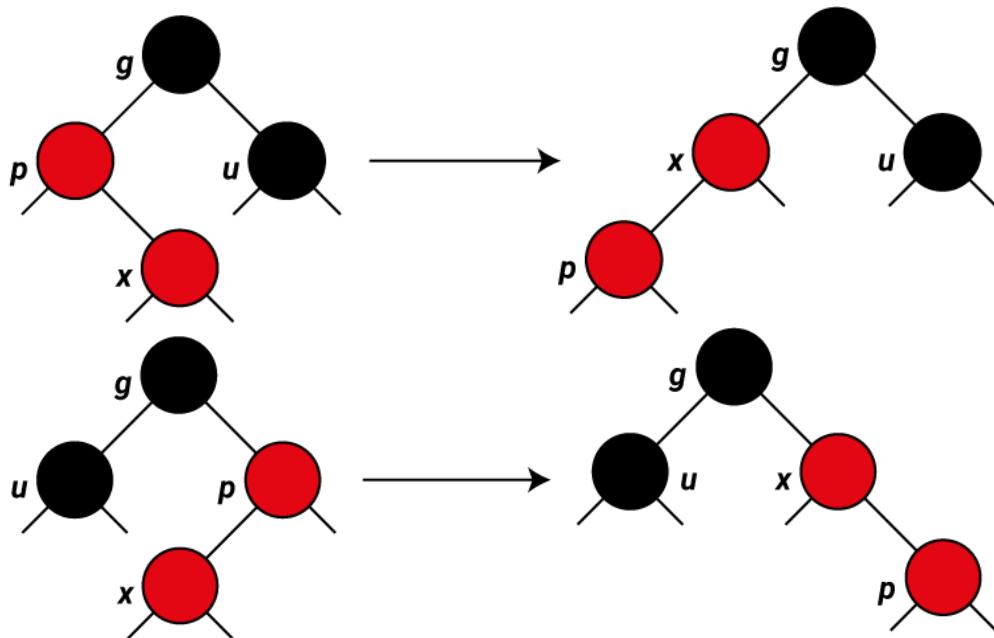


Рисунок 5.1.2. Иллюстрация второго случая функции балансировки после добавления вершины.

Начальная ситуация изображена на картинке (рис. 5.1.2):

Сделаем малый поворот для добавленной вершины. С помощью этого мы перейдем к начальным условиям для случая 3 (где будем рассматривать родителя добавленной вершины).

Случай 3:

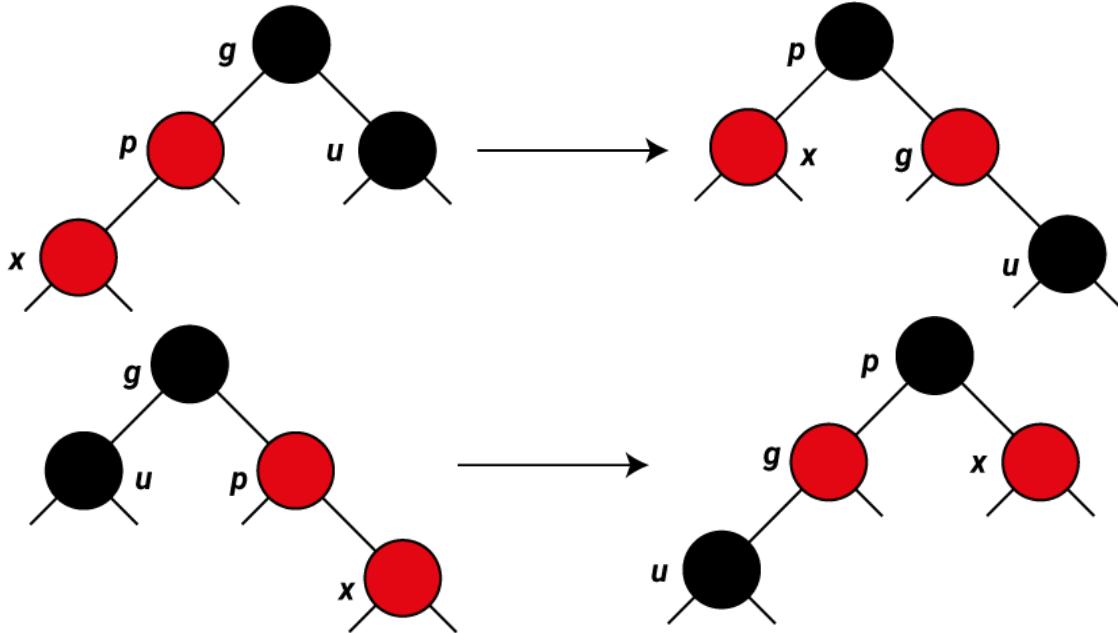


Рисунок 5.1.3. Иллюстрация третьего случая функции балансировки после добавления вершины.

Начальная ситуация: черный дядя (вершина u) и красный пapa (вершина p). Здесь нам понадобится совершить большой поворот вокруг деда (вершина g), а затем перекрасить g в красный, а p в черный.

Убедимся, что черная высота после поворота сохранена.

Удаление:

Удаление работает так же как описано в общем случае. Затем, если удаленная вершина была черной, запускается функция балансировки, в которой рассматривается ребенок удаленной вершины:

Шаг 1:

Если брат (вершина b) этого ребёнка красный, то делаем вращение вокруг ребра между отцом (вершина p) и братом, тогда брат становится родителем отца. Красим b в чёрный, а p — в красный цвет, сохраняя таким образом черную высоту дерева. Хотя все пути по-прежнему содержат одинаковое количество чёрных узлов, сейчас x имеет чёрного брата и красного отца. Таким образом, мы можем перейти к следующему шагу.

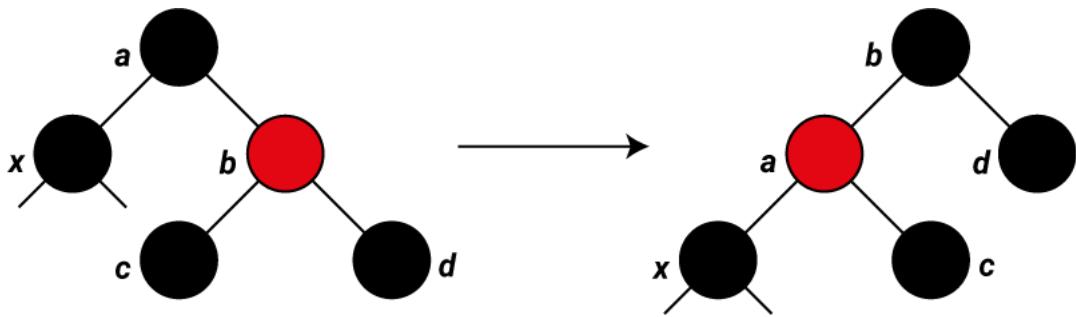


Рисунок 5.2.1. Иллюстрация первого шага функции балансировки после удаления вершины.

Шаг 2:

Если брат (вершина b) текущей вершины был чёрным, то имеем три случая:

Случай 1:

Оба ребёнка (вершины c и d) у брата чёрные. Красим брата (вершину b) в красный цвет, а отца (вершину a) в чёрный. Этим действием мы не изменим количество чёрных узлов на путях, проходящих через b, но добавим один к числу чёрных узлов на путях, проходящих через x. Тем самым восстановим влияние удаленного чёрного узла. Значит после удаления и последующей балансировки чёрная высота будет одинаковой. Далее рассматриваем вершину a.

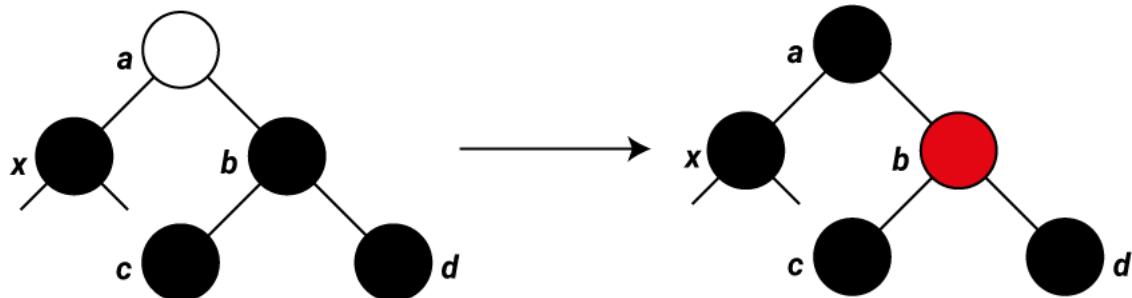


Рисунок 5.2.2. Иллюстрация первого случая второго шага функции балансировки после удаления вершины.

Случай 2:

Если у брата (вершина b) правый ребёнок (вершина d) чёрный, а левый (вершина c) — красный, то перекрашиваем брата в красный, а его левого сына в чёрный, и делаем вращение. Все пути так же содержат одинаковое количество чёрных узлов, но теперь у x есть чёрный брат с красным правым потомком, и мы переходим к следующему случаю. Ни x, ни его отец (вершина a) не влияют на эту трансформацию. Далее рассматриваем исходную вершину (вершину x).

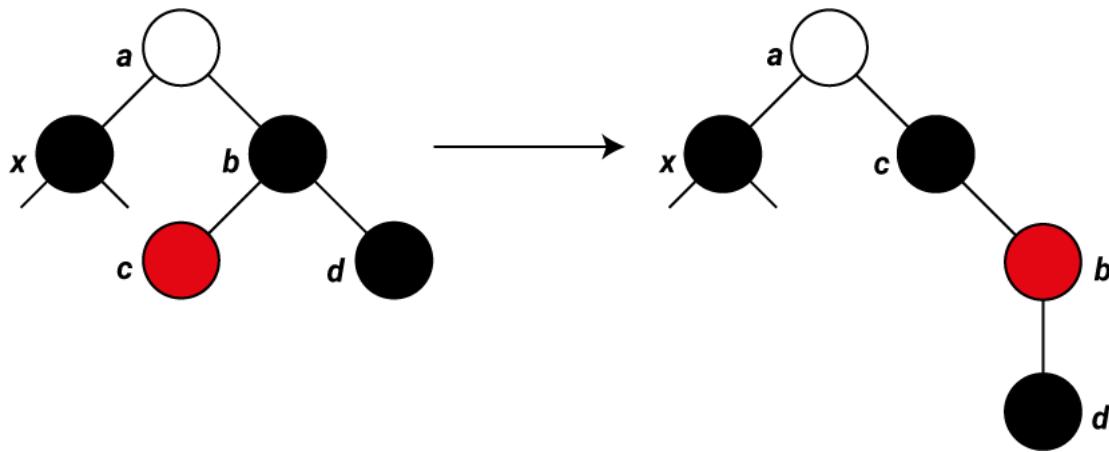


Рисунок 5.2.3. Иллюстрация второго случая второго шага функции балансировки после удаления вершины.

Случай 3:

Если у брата (вершина b) правый ребёнок (вершина d) красный, то перекрашиваем брата в цвет отца, его ребёнка и отца — в чёрный, делаем вращение. Поддерево по-прежнему имеет тот же цвет корня, поэтому свойство 3 и 4 не нарушаются. Но у x теперь появился дополнительный чёрный предок: либо a стал чёрным, или он и был чёрным и b был добавлен в качестве чёрного дедушки. Таким образом после балансировки мы вернули влияние удаленной чёрной вершины. Выходим из алгоритма.

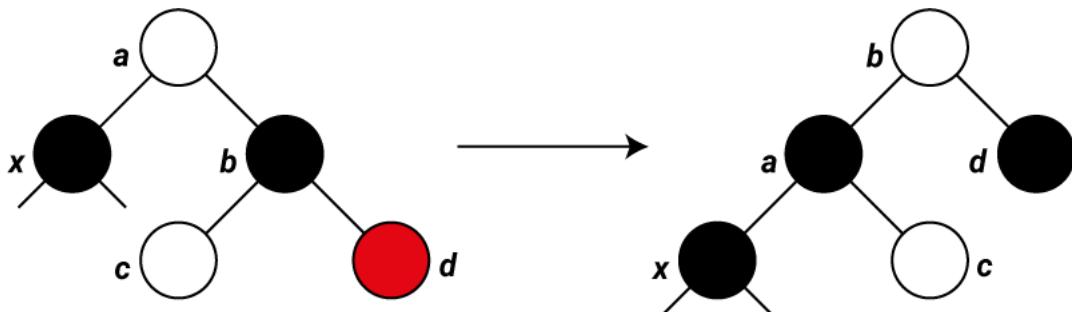


Рисунок 5.2.4. Иллюстрация третьего случая второго шага функции балансировки после удаления вершины.

Продолжаем тот же алгоритм, пока текущая вершина чёрная и мы не дошли до корня дерева. Из рассмотренных случаев ясно, что при удалении выполняется не более трёх вращений.

Сложность:

Добавление:

Из рассмотренных случаев ясно, что цикл продолжается при первом случае. Значит в худшем случае функция балансировки после добавления дойдет до корня, а в остальных случаях цикл прекращается. Значит, сложность добавления $O(\log n)$. Заметим, что происходит не более 2-х вращений при любой операции вставки.

Удаление:

Так же как и с добавлением, в худшем случае мы дойдем до корня, значит сложность удаления - $O(\log n)$.

Название операции	В среднем	В худшем случае
Поиск	$O(\log n)$	$O(\log n)$
Удаление	$O(\log n)$	$O(\log n)$
Вставка	$O(\log n)$	$O(\log n)$

Рисунок 5.3 Таблица асимптотики для Red-black дерева

Декартово дерево

Декартово дерево или дуча (дерево + куча), дерамида (дерево + пирамида), курево (куча + дерево) – это структура данных, объединяющая в себе бинарное дерево поиска и бинарную кучу. Идея декартова дерева была предложена в 1996 году Раймундом Зайделем и Сесилией Родригес Арагон.

Идея

Декартово дерево представляет собой бинарное дерево, в узлах которого хранятся пары (x, y) , где x – это ключ, а y – приоритет. Оно является двоичным деревом поиска по x и бинарной кучей по y . Предполагая, что все x и все y являются различными, получаем, что если некоторый элемент дерева содержит (x_0, y_0) , то у всех элементов в левом поддереве $x < x_0$, у всех элементов в правом поддереве $x > x_0$, а также для обоих поддеревьев верно, что: $y < y_0$.

Основные операции

Split:

Эта операция устроена совершенно обычно. $\text{Split}(k)$ делит дерево по ключу k , возвращая левое поддерево T_1 , правое поддерево T_2 и сам ключ k , если он имелся в исходном дереве. При этом для всех ключей выполняется $T_1 < k < T_2$.

Из алгоритма, приведённого в описании операции следует, что она выполняется за линейное от высоты дерева время $O(h)$.

Merge:

Имея два дерева T_1 и T_2 нужно получить объединённое дерево T . У T должен быть корень, а в декартовом дереве корень имеет наивысший приоритет. Выберем вершину с наивысшим приоритетом среди T_1 и T_2 . Поскольку для них выполнены требования декартового дерева, вершина с наибольшим приоритетом это один из корней данных поддеревьев. Пусть корень T_1 имеет больший приоритет. Тогда левое поддерево T это левое поддерево T_1 , а справа нужно подвесить объединение правого поддерева T_1 и дерева T_2 .

Таким образом, сложность операции Merge также становится $O(h)$.

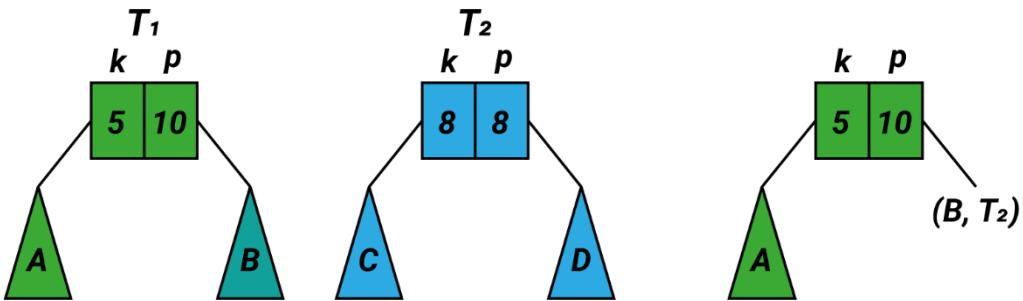


Рисунок 6.1 Иллюстрация метода merge

Find:

Поиск в декартовом дереве аналогичен операции поиска в любом другом бинарном дереве и имеет сложность $O(h)$.

Insert:

В простейшем виде операция вставки ключа k производится в 3 действия: Сначала мы выполняем Split по k и получаем два поддерева T_1 и T_2 . Затем мы выполняем слияние для T_1 и k , как дерева из одной вершины. После этого полученное дерево мы сливаем с поддеревом T_2 .

Сложность такой операции эквивалента сложности операций Split и Merge – $O(h)$. Нужно упомянуть, что ключ для новой вершины мы получаем от пользователя, а вот её приоритет нужно придумать самостоятельно. Будем генерировать его случайно и равномерно. Почему – станет ясно из доказательства сложности.

Erase:

Эту операцию можно реализовать как разбиение дерева по ключу k и последующее слияние левого и правого поддеревьев.

Сложность такого алгоритма составляет $O(h)$.

Высота декартова дерева со случайными приоритетами:

Будем считать, что все выбранные приоритеты у попарно различны.

Для начала введём несколько обозначений:

x_k — вершина с k -ым по величине ключом;

индикаторная величина $A_{i,j} = 1$, если x_i предок x_j и 0 в противоположном случае.

$d(v)$ — глубина вершины v ;

В силу обозначений глубину вершины можно записать как количество предков:

$$d(x_k) = \sum_{i=1}^n A_{i,k}.$$

Теперь можно выразить математическое ожидание глубины конкретной вершины:

$E(d(x_k)) = \sum_{i=1}^n \Pr[A_{i,k} = 1]$, поскольку математическое ожидание линейно и для индикаторной величины равно вероятности того, что индикатор стал единицей.

Для подсчёта средней глубины вершин нам нужно сосчитать вероятность того, что вершина x_i является предком вершины x_k , то есть $\Pr[A_{i,k} = 1]$.

Введём новое обозначение:

$X_{i,k} = X_{k,i}$ — множество ключей $\{x_i, \dots, x_k\}$, мощность которого равна $|k - i| + 1$.

Лемма:

$\forall i \neq k, x_i$ — предок $x_k \Leftrightarrow x_i$ имеет наибольший приоритет среди $X_{i,k}$.

Доказательство:

Если x_i является корнем, то оно является предком x_k и по определению имеет максимальный приоритет среди всех вершин, следовательно, и среди $X_{i,k}$.

С другой стороны, если x_k — корень, то x_i — не предок x_k , и x_k имеет максимальный приоритет в декартовом дереве; следовательно, x_i не имеет наибольший приоритет среди $X_{i,k}$.

Теперь предположим, что какая-то другая вершина x_m — корень. Тогда, если x_i и x_k лежат в разных поддеревьях, то $i < m < k$ или $i > m > k$, следовательно, x_m содержится в $X_{i,k}$. В этом случае x_i — не предок x_k , и наибольший приоритет среди $X_{i,k}$ имеет вершина с номером m .

Наконец, если x_i и x_k лежат в одном поддереве, то доказательство применяется по индукции: пустое декартово дерево есть тривиальная база, а рассматриваемое поддерево является меньшим декартовым деревом.

Так как распределение приоритетов равномерное, каждая вершина среди $X_{i,k}$ может иметь максимальный приоритет, мы немедленно приходим к следующему равенству:

$\Pr[A_{i,k} = 1] = 0$, если $k = i$, и $\frac{1}{|k-i|+1}$ иначе. Тогда:

$$E(d(x_k)) = \sum_{i=1}^n \Pr[A_{i,k} = 1] = \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^n \frac{1}{i-k+1} \leq \ln(k) + \ln(n-k) + 2.$$

Мы использовали неравенство $\sum_{i=1}^n \frac{1}{i} \leq \ln(n) + 1$.

Поскольку $\log(n) = O(\ln(n))$, итоговая средняя высота декартова дерева составляет $O(\log(n))$.

Сложность:

Название операции	В среднем	В худшем случае
Поиск	$O(\log n)$	$O(n)$
Удаление	$O(\log n)$	$O(n)$
Вставка	$O(\log n)$	$O(n)$

Рисунок 6.2 Таблица асимптотики для Cartesian дерева

Сплей дерево

Splay дерево придумали Роберт Тарьян и Даниель Слейтор в 1983 году.

Идея:

В данной структуре данных ключевой особенностью является балансирующая функция Splay, вызывающаяся после каждой операции, перестраивающая дерево таким образом, что вершина, к которой вызван Splay, встает в корень.

Splay состоит из 3 (учитывая симметрию 6) операций вращения вершин:

Zig (+ Zag)

Zig-Zig (+ Zag-Zag)

Zig-Zag (+ Zag-Zig)

Обозначим вершину, которую хотим переместить в корень за x , её родителя — p , а родителя p (если существует) — g .

Zig: выполняется, когда p является корнем. Дерево поворачивается по ребру между x и p . Существует лишь для разбора крайнего случая и выполняется только один раз в конце, когда изначальная глубина x была нечётна.

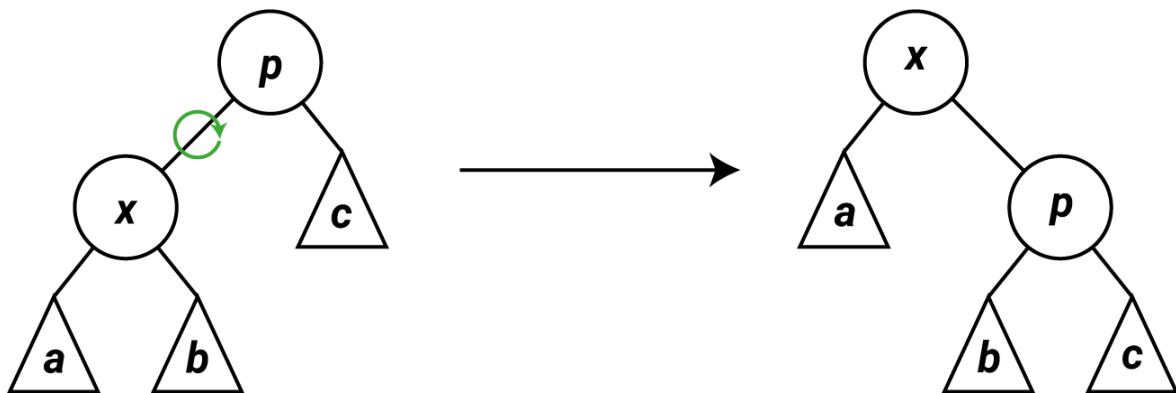


Рисунок 6.1 Иллюстрация поворота zig.

Zig-Zig: выполняется, когда и x , и p являются левыми (или правыми) сыновьями. Дерево поворачивается по ребру между g и p , а потом — по ребру между p и x .

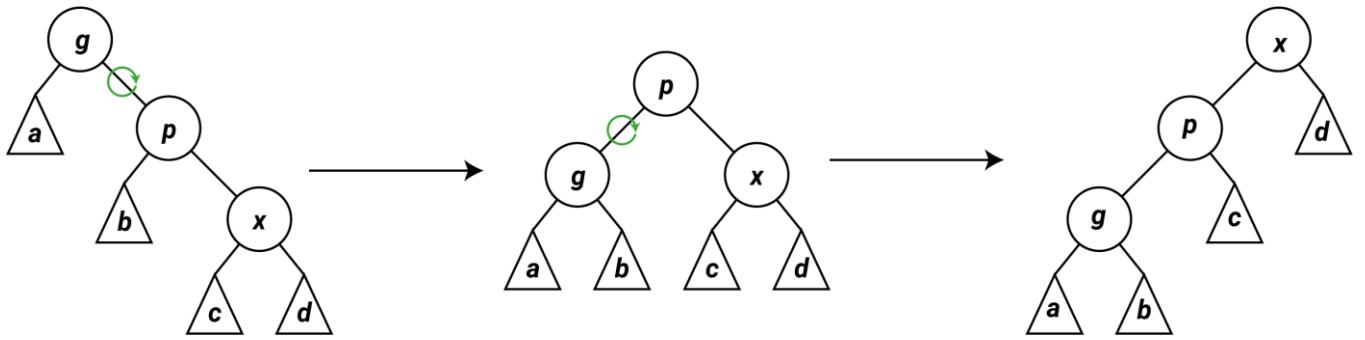


Рисунок 6.2 Иллюстрация поворота zig-zig.

Zig-Zag: выполняется, когда x является правым сыном, а p — левым (или наоборот). Дерево поворачивается по ребру между p и x , а затем — по ребру между x и g .

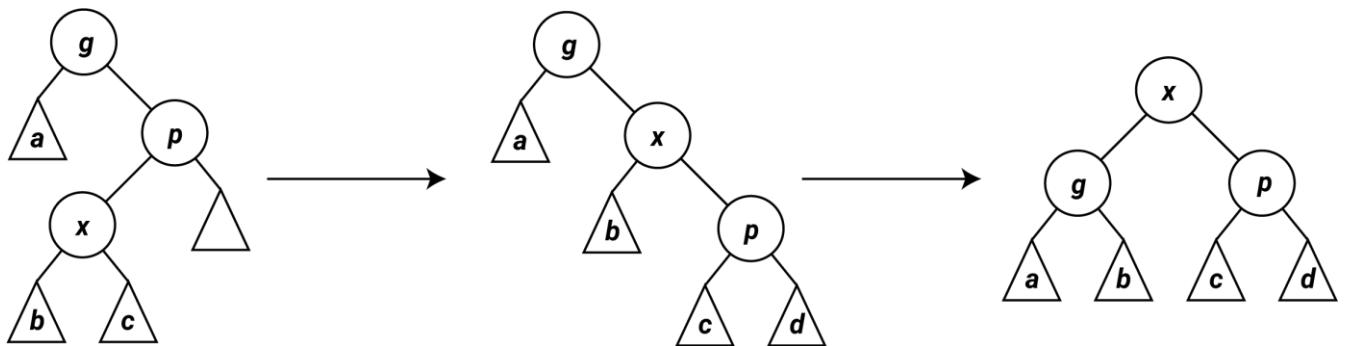


Рисунок 6.3 Иллюстрация поворота zig-zag..

Амортизационный анализ сплай-дерева проводится с помощью метода потенциалов. Потенциалом рассматриваемого дерева назовем сумму рангов его вершин. Ранг вершины x — это величина, обозначаемая $r(x)$ и равная $C(x)$, где $C(x)$ — количество вершин в поддереве с корнем в x .

Лемма:

Амортизированное время операции splay вершины x в дереве с корнем t не превосходит $3r(t) - 3r(x) + 1$.

Доказательство:

Проанализируем каждый шаг операции splay. Пусть r и r' — ранги вершин после шага и до него соответственно, p — предок вершины x , а g — предок p (если есть).

Разберём случаи в зависимости от типа шага:

Zig:

Поскольку выполнен один поворот, то амортизированное время выполнения шага

$$T = 1 + r'(x) + r'(p) - r(x) - r(p)$$

(поскольку только у вершин x и p меняется ранг).

Ранг вершины p уменьшился, поэтому $T \leq 1 + r'(x) - r(x)$.

Ранг вершины x увеличился, поэтому $r'(x) - r(x) \geq 0$.

Следовательно, $T \leq 1 + 3r'(x) - 3r(x)$.

Zig-zig:

Выполнено два поворота, амортизированное время выполнения шага

$$T = 2 + r'(x) + r'(p) + r'(g) - r(p) - r(x) - r(g).$$

Поскольку после поворотов поддерево с корнем в x будет содержать все вершины, которые были в поддереве с корнем в g (и только их), поэтому $r'(x) = r(g)$.

Используя это равенство, получаем:

$T = 2 + r'(p) + r'(g) - r(x) - r(p) \leq 2 + r'(p) + r'(g) - 2r(x)$, поскольку $r(x) \leq r(p)$.

Далее, так как $r'(p) \leq r'(x)$, получаем, что

$$T \leq 2 + r'(x) + r'(g) - 2r(x).$$

Мы утверждаем, что эта сумма не превосходит $3(r'(x) - r(x))$, то есть, что $r(x) + r'(g) - 2r'(x) \leq -2$. Преобразуем полученное выражение следующим образом:

$$(r(x) - r'(x)) + (r'(g) - r'(x)) = \frac{c(x)}{c'(x)} + \frac{c(g)}{c'(x)}.$$

Из рисунка видно, что

$C'(g) + C(x) \leq C'(x)$, значит, сумма выражений под логарифмами не превосходит единицы. Далее, рассмотрим сумму логарифмов $a + b = ab$. При $a + b \leq 1$ произведение ab по неравенству между средними не превышает $\frac{1}{4}$. А поскольку логарифм — функция возрастающая, то $ab \leq -2$, что и является требуемым неравенством.

Zig-zag:

Выполнено два поворота, амортизированное время выполнения шага

$$T = 2 + r'(x) + r'(p) + r'(g) - r(p) - r(x) - r(g).$$

Поскольку $r'(x) = r(g)$, то

$$T = 2 + r'(p) + r'(g) - r(p) - r(x).$$

Далее, так как $r(x) \leq r(p)$,

$$\text{то } T \leq 2 + r'(p) + r'(g) - 2r(x).$$

Мы утверждаем, что эта сумма не превосходит $2(r'(x) - r(x))$, то есть, что $r'(p) + r'(g) - 2r'(x) \leq -2$.

Но, поскольку $r'(p) + r'(g) - 2r'(x) = \frac{C(p)}{C(x)} + \frac{C(g)}{C(x)} \leq -2$ - аналогично доказанному ранее, что и требовалось доказать.

Итого, получаем, что амортизированное время шага zig-zag не превосходит $2(r'(x) - r(x)) \leq 3(r'(x) - r(x))$.

Поскольку за время выполнения операции splay выполняется не более одного шага типа zig, то суммарное время не будет превосходить $3r(t) - 3r(x) + 1$, поскольку утроенные ранги промежуточных вершин сокращаются (входят в сумму как с плюсом, так и с минусом).

Тогда суммарное время работы splay:

$T_{\text{splay}} \leq 3 * \log_2 N - 3 * \log_2 C(x) + 1 = O(\log_2 N)$, где N — число элементов в дереве.

Что и требовалось доказать.

Основные методы:

Find:

Операция поиска работает так же, как и было описано в общем случае за исключением того, что при нахождении элемента вызывается балансирующая функция Splay к найденной вершине, а при не нахождении элемента Splay вызывается к последней просмотренной вершине.

Insert:

Операция вставки работает так же, как и было описано в общем случае за исключением того, что при успешной вставке элемента вызывается балансирующая функция Splay к вставленной вершине, а если этот элемент уже был в дереве, то также вызываем к нему Splay.

Erase:

Операция удаления работает следующим образом: сначала пытаемся найти данную вершину. Если не находим ее, то вызываем балансирующую функцию Splay к последней просмотренной вершине. Если же находим вершину, то сначала запускаем Splay к ней. Она встает в корень дерева, и мы вызываем Merge (слияние) к ее сыновьям.

Merge:

Слияние принимает на вход 2 дерева: left и right. Причем все вершины в дереве right должны быть больше, чем в left. Далее для минимального элемента дерева right производится Splay, после чего в дереве left ищем место для вставки корня right и сливаем деревья.

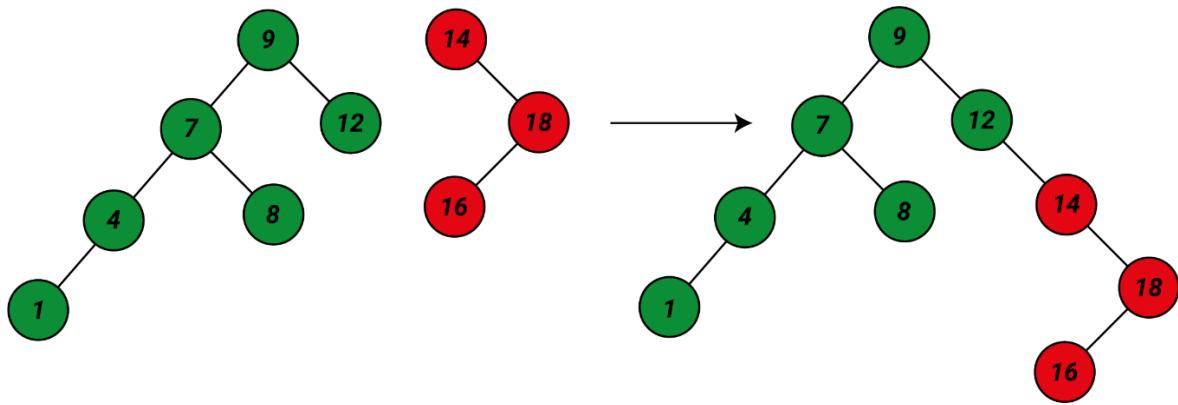


Рисунок 6.4 Иллюстрация работы метода «слияние».

Сложность:

Исходя из леммы об амортизированной сложности операции Splay, доказанной ранее, можем сделать вывод, что так как все методы состоят из поиска и балансирующей функции splay, то сложность всех основных методов будет в среднем случае будут $O(\log n)$, а в худшем *Amortized*($O(\log n)$).

Название операции	В среднем	В худшем случае
Поиск	$O(\log n)$	Amortized($O(\log n)$)
Удаление	$O(\log n)$	Amortized($O(\log n)$)
Вставка	$O(\log n)$	Amortized($O(\log n)$)

Рисунок 6.5 Таблица асимптотики для Splay дерева.

Список с пропусками

Список с пропусками был разработан в 1989 году Уильямом Пью из Университета Мэриленда. Список с пропусками рассматривается как альтернатива самобалансирующимся бинарным деревьям поиска.

Идея

Список с пропусками состоит из нескольких уровней, на каждом из которых находится отсортированный связный список. На самом нижнем (первом) уровне располагаются все элементы. Дальше около половины элементов в таком же порядке располагаются на втором, почти четверть — на третьем и так далее, но при этом известно, что если элемент расположен на уровне i , то он также расположен на всех уровнях, номера которых меньше i .

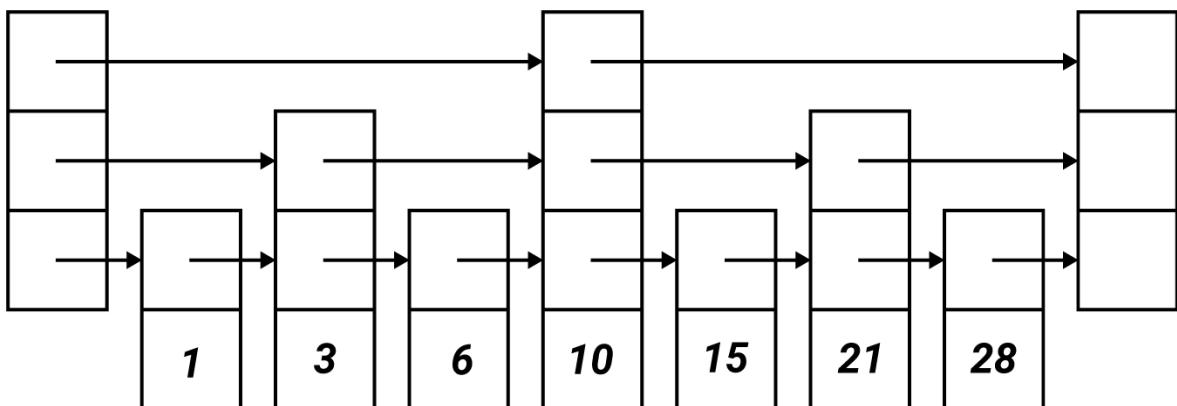


Рисунок 7.1 Структура данных “Список с пропусками”

Лемма:

Ожидаемая глубина списка с пропусками с n вершинами в среднем $\log n + 2$.

Доказательство:

Для каждого r из $(1, 2, \dots, \infty)$ определим индикатор вероятности рандома:

$I_r = 0$, если L_r пустой;

$I_r = 1$, если L_r не пустой.

При условии, что L_r — количество вершин на r уровне.

Тогда высота списка с пропусками определяется как: $\sum_{r=1}^{\infty} I_r$

Заметим, что I_r никогда не бывает больше, чем $|L_r|$, получаем $E[I_r] \leq E[|L_r|] = \frac{n}{2^r}$.

Таким образом имеем:

$$\begin{aligned} E[h] &= E\left[\sum_{r=1}^{\infty} I_r\right] = \sum_{r=1}^{\infty} E[I_r] = \sum_{r=1}^{\log n} E[I_r] + \sum_{r=\lceil \log n \rceil + 1}^{\infty} E[I_r] \leq \sum_{r=1}^{\log n} 1 + \sum_{r=\lceil \log n \rceil + 1}^{\infty} \frac{n}{2^r} \\ &\leq \log n + \sum_{r=0}^{\infty} \frac{1}{2^r} = \log n + 2 \end{aligned}$$

Основные методы:

Поиск:

Поиск элемента начинается с самого верхнего уровня. Пока значение в следующей ячейке меньше, переходим к следующему элементу. Когда встречаем значение больше, либо равно, то перемещаемся на один уровень ниже. Продолжаем так до тех пор, пока не окажемся на первом уровне. Если нужная вершина есть в списке, то она находится справа.

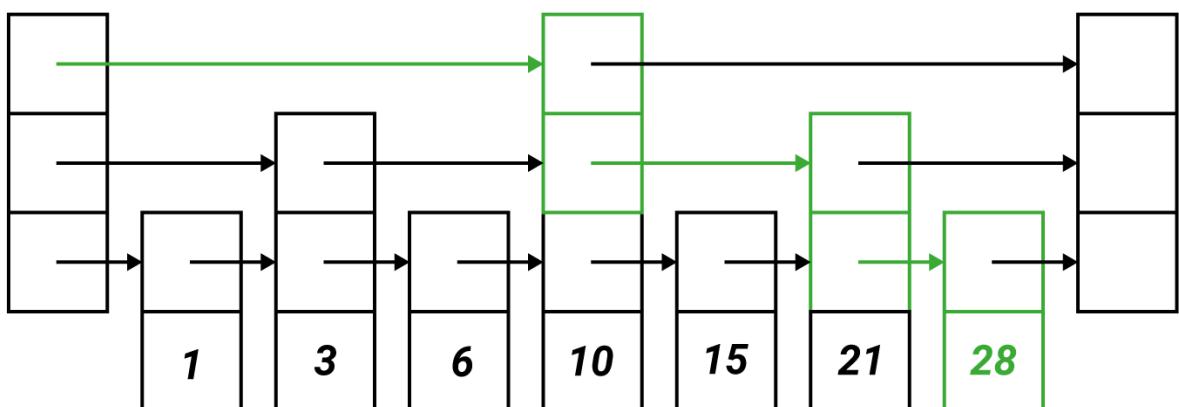


Рисунок 7.2 Метод поиска элемента в списке с пропусками

Вставка:

Сначала происходит поиск места вставки элемента. Поиск начинается с самого верхнего уровня. Пока значение в следующей ячейке меньше, переходим к следующему элементу. Когда встречаем значение больше, либо равно, то перемещаемся на один уровень ниже. Продолжаем так до тех пор, пока не окажемся

на первом уровне. Если такая вершина уже существует в списке, то ничего не делаем. Иначе вставляем справа нужную вершину.

Далее с определенной вероятностью $1/n$ (наиболее распространенный вариант $n = 2$) мы строим вершину на уровне выше и соединяем ее с левой и правой вершиной. Для простоты в случае $n = 2$ будем говорить, что мы “бросаем монетку”. Так вот, если, бросая монетку, нам выпадает “орел”, то мы строим новый уровень. Повторяем так до тех пор, пока нам не выпадет “решка”.

Стоит обговорить случай, когда вставка нового элемента увеличивает число уровней. Тогда необходимо создать еще один отсортированный список, в котором будет всего один текущий элемент. В некоторых реализациях принято, что вставка каждого нового элемента увеличивает число уровней не более чем на один.

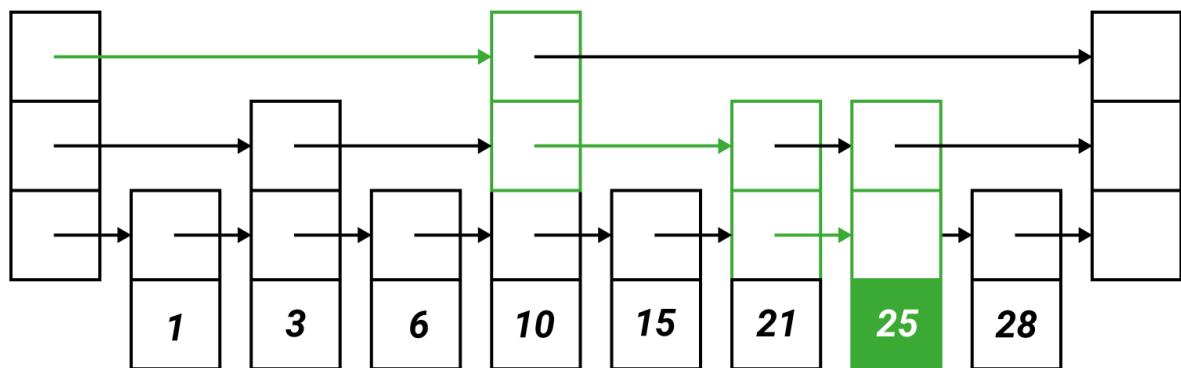


Рисунок 7.3 Метод вставки элемента в списке с пропусками

Удаление:

Сначала происходит поиск места вставки элемента, но ищем мы уже не нижний уровень, а верхний. Поиск начинается с самого верхнего уровня. Пока значение в следующей ячейке меньше, переходим к следующему элементу. Если значение в следующей ячейке больше, то: спускаемся на уровень ниже и повторяем поиск. Если же нижнего уровня нет и нужной вершины нет справа, то заканчиваем операцию удаления. Если же значение в вершине справа равняется нужной, то удаляем ее и, спускаясь вниз, удаляем все вершины, пока не дойдем до самого нижнего уровня. Если же в процессе удаления у нас сверху образовался пустой уровень без вершин, то мы удаляем его.

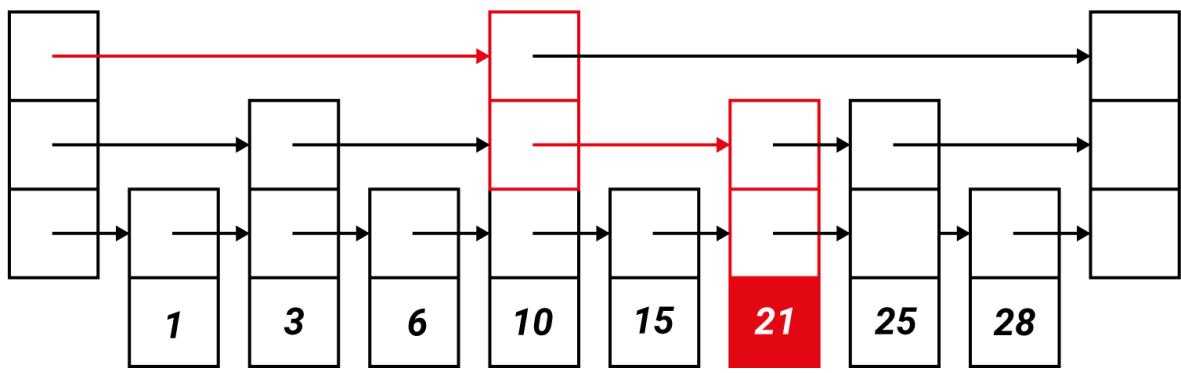


Рисунок 7.4 Метод удаления элемента в списке с пропусками

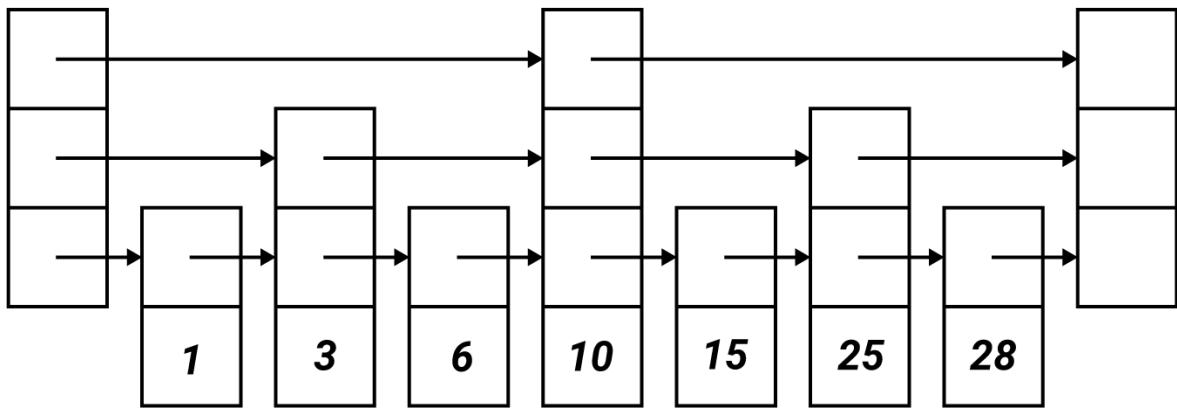


Рисунок 7.4 Результат удаления элемента в списке с пропусками

Сложность:

Так как ожидаемое время выполнения операции поиска пропорционально ожидаемой глубине списка с пропусками – $O(h)$, то стоимость всех остальных методов, состоящих из поиска и построении/удалении вершин в среднем будет $O(\log n)$.

Аналогично, в худшем случае если при построении какой-либо вершины ее глубина будет n или же список с пропусками будет вырожденным, то и время выполнения всех операций будет $O(n)$.

Название операции	В среднем	В худшем случае
Поиск	$O(\log n)$	$O(n)$
Удаление	$O(\log n)$	$O(n)$
Вставка	$O(\log n)$	$O(n)$

Рисунок 7.5 Таблица асимптотики в списке с пропусками

Эксперименты

Мы провели тесты производительности для наших алгоритмов на разных наборах данных. Для сравнения мы также измерили время работы на тех же данных алгоритма std::set, реализованного в стандартной библиотеке C++ (компилятор gcc 7.4.0, C++17). Пусть мы хотим проверить время работы той или иной операции на n элементах, тогда введём обозначения для следующих числовых последовательностей:

$$\text{Inc (increasing)} = 0, 1, 2, \dots, n$$

$$\text{Dec (decreasing)} = n, n - 1, \dots, 0$$

$$\text{Conv (converging)} = n, 0, n - 1, 1, \dots, \left\lfloor \frac{n}{2} \right\rfloor$$

$$\text{Div (diverging)} = \left\lfloor \frac{n}{2} \right\rfloor, \left\lfloor \frac{n}{2} \right\rfloor + 1, \left\lfloor \frac{n}{2} \right\rfloor - 1, \dots, n, 0$$

RSp (random sparse) = $x_1, x_2, \dots, x_n : x_i \in [-2^{31}, 2^{31} - 1]$, x_i распределены равномерно

RDn (random dense) = $x_1, x_2, \dots, x_n : x_i \in \left[0, \frac{n}{k}\right], k = \text{const} = 5$ в нашем эксперименте, x_i равномерно распределены по интервалу.

Для всех экспериментов мы брали n из логарифмического распределения по такому правилу:

Минимальное $n = 1$

Максимальное $n = 10^6$

n принимает 35 различных значений.

Таким образом мы брали n из ряда $1, q, q^2, \dots, q^{34} = 10^6$, где $q = \sqrt[34]{10^6}$

На графиках по оси Ох отложены значения n , а по оси Оу время в миллисекундах для выполнения теста при выбранном значении n . При каждом значении n все ответы были разделены на $\ln(n)$ для удобства анализа графиков. Таким образом прямая линия на графике соответствует функции, эквивалентной $f(n) = n \cdot \ln(n)$. Вогнутый график говорит о том, что функция является $o(n \cdot \ln(n))$, выпуклый – функция растёт быстрее, чем $O(n \cdot \ln(n))$.

Каждый тест был выполнен по 5 раз и время работы алгоритмов было усреднено во избежание неточностей. Яркая цветная линия на графике отображает среднее время выполнения теста по 5 попыткам, а полупрозрачная фигура того же цвета вокруг показывает стандартное отклонение, взятое за квадратный корень из дисперсии времени выполнения ($\sqrt{\frac{1}{5} t_1^2 + \dots + t_5^2}$).

Все эксперименты проводились на компьютере с процессором Intel Core i5-8265U с тактовой частотой 1.6ГГц, поднимающейся до 3.9ГГц, 8ГБ оперативной памяти. Несмотря на возможность многопоточного запуска тестов, использовалось только одно ядро, во избежание влияния тестов на друг друга.

Графики были построены по полученным данным в среде Jupyter Notebook на языке Python 3 с помощью библиотек numpy, matplotlib и seaborn. Все данные, а также исходный код построений доступен в приложении и git-репозитории.

Операция вставки:

Для всех 6 видов числовых последовательностей были проведены измерения времени вставки всех элементов из них:

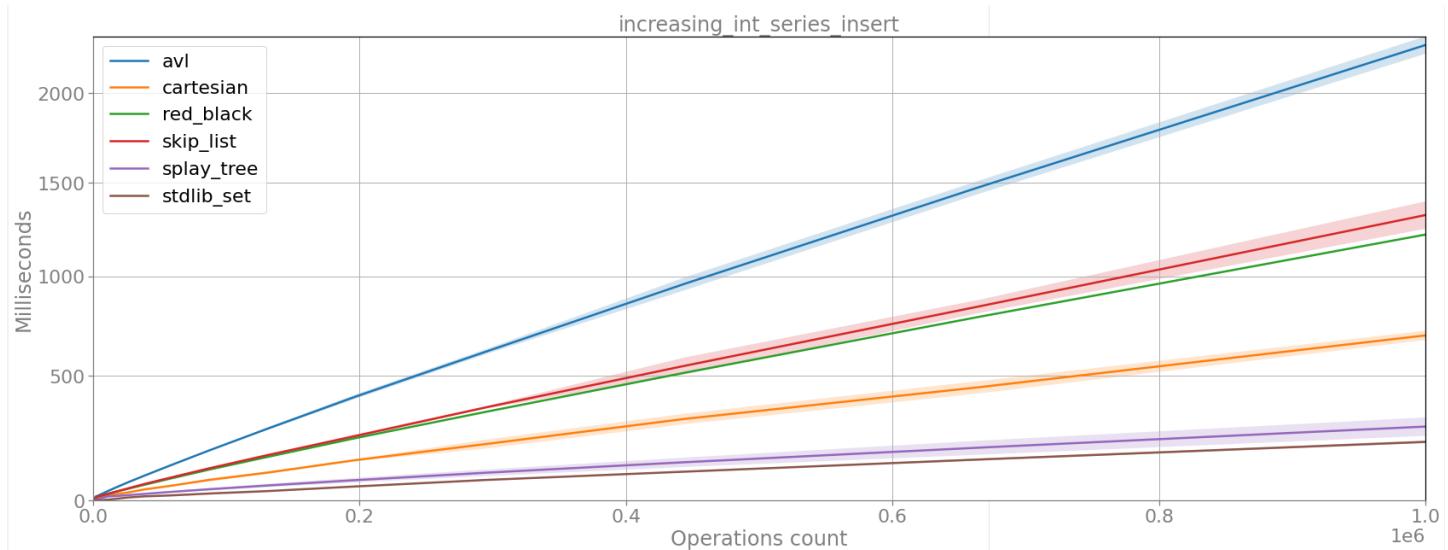


Рисунок 8.1.1. График зависимости времени работы операции вставки от количества элементов последовательности Inc

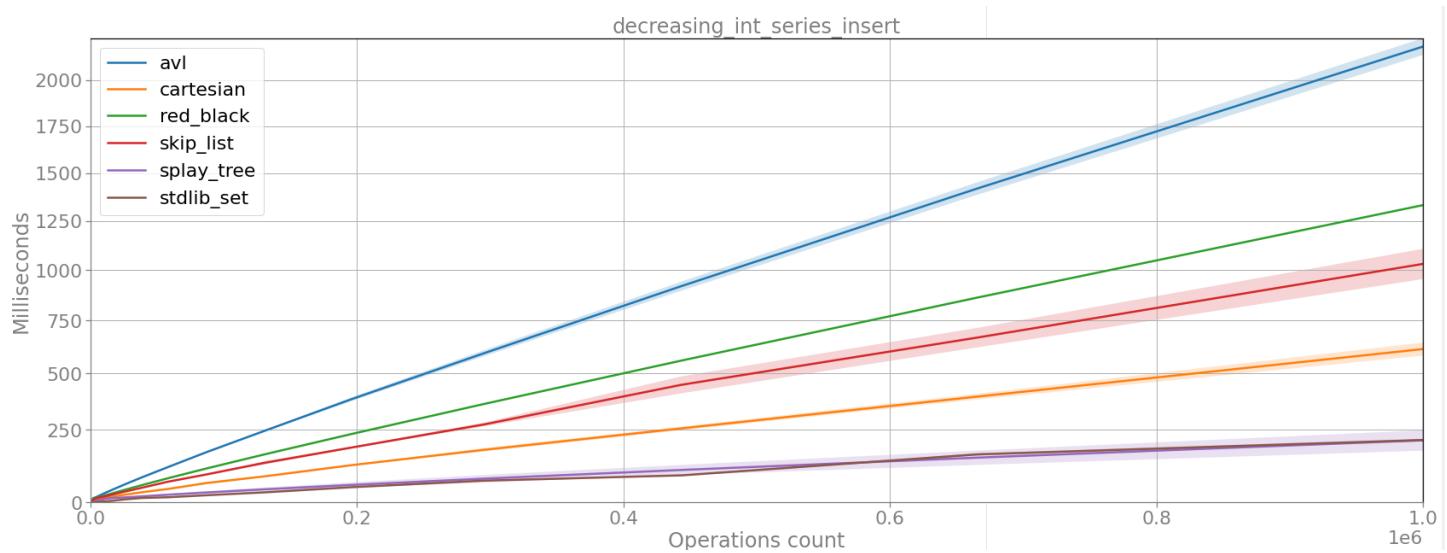


Рисунок 8.1.2. График зависимости времени работы операции вставки от количества элементов последовательности Dec

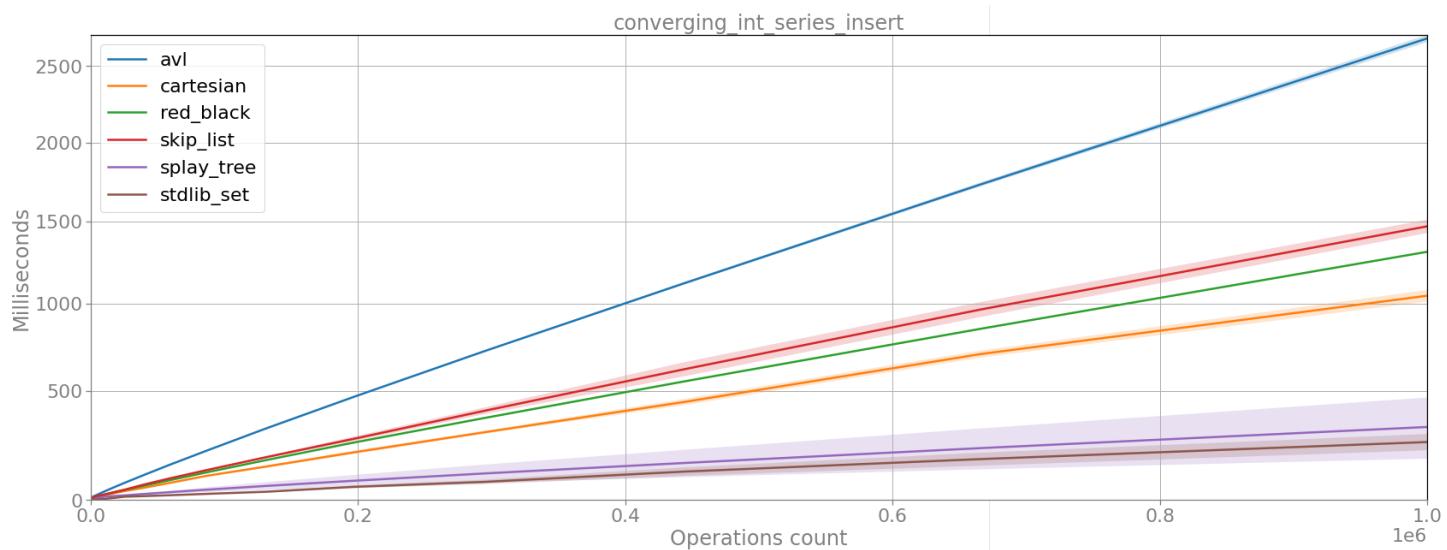


Рисунок 8.1.3. График зависимости времени работы операции вставки от количества элементов последовательности Conv.

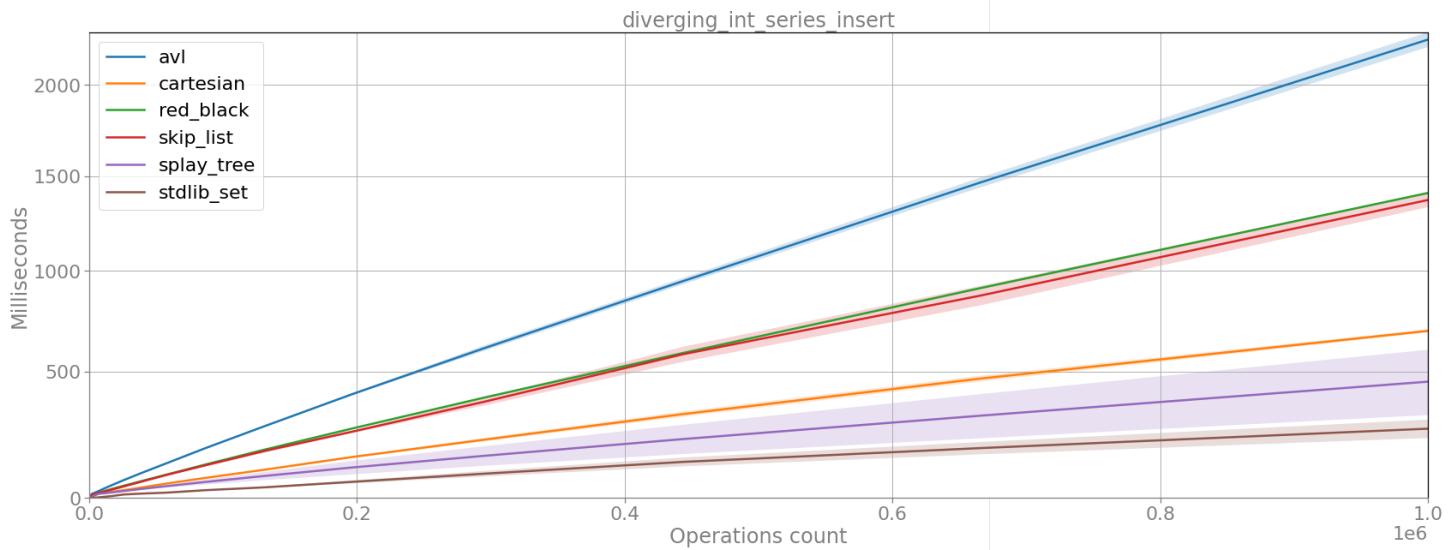


Рисунок 8.1.4. График зависимости времени работы операции вставки от количества элементов последовательности Div

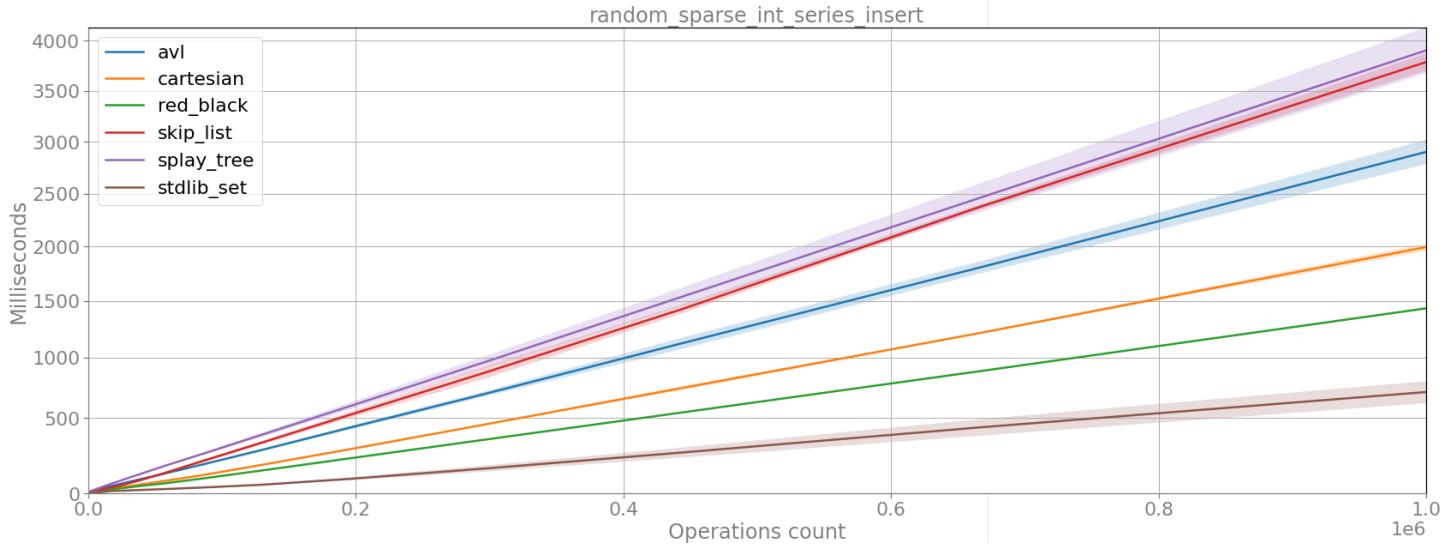


Рисунок 8.1.4. График зависимости времени работы операции вставки от количества элементов последовательности RSp

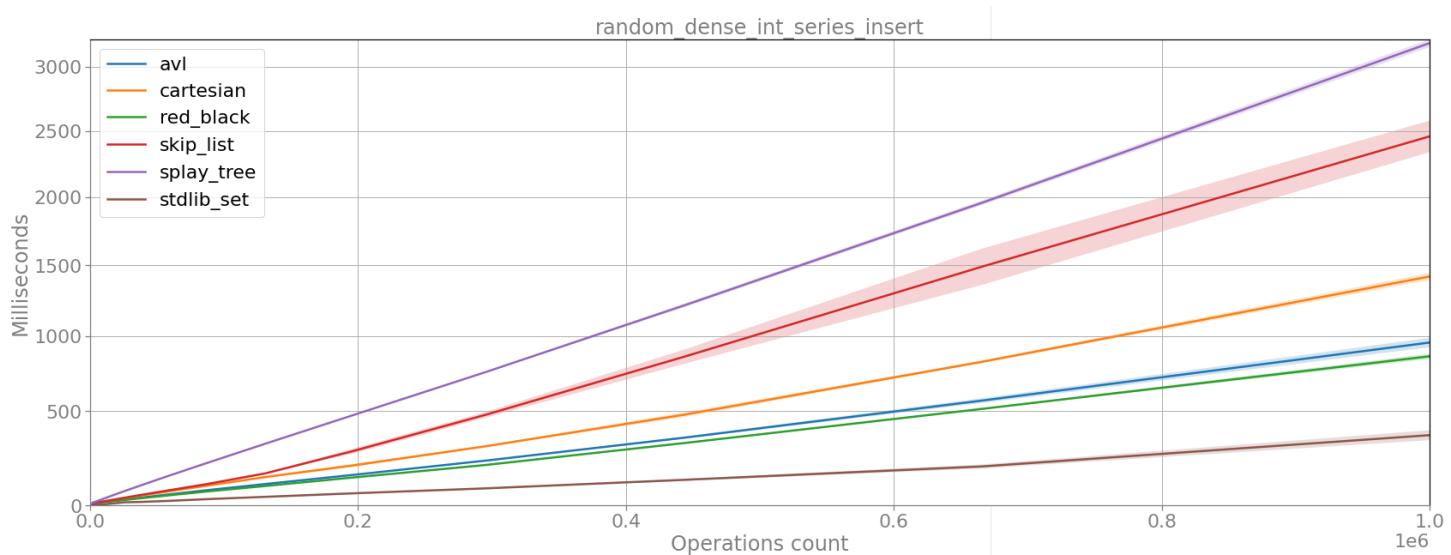


Рисунок 8.1.4. График зависимости времени работы операции вставки от количества элементов последовательности RDn

Также мы проверили время работы алгоритмов на строковых данных. Мы брали исходный текст размером 1КБ, в конец которого добавлялись числа как в операциях RSp и RDn. Текстовые данные отличаются намного большим временем, требуемым для сравнения двух элементов:

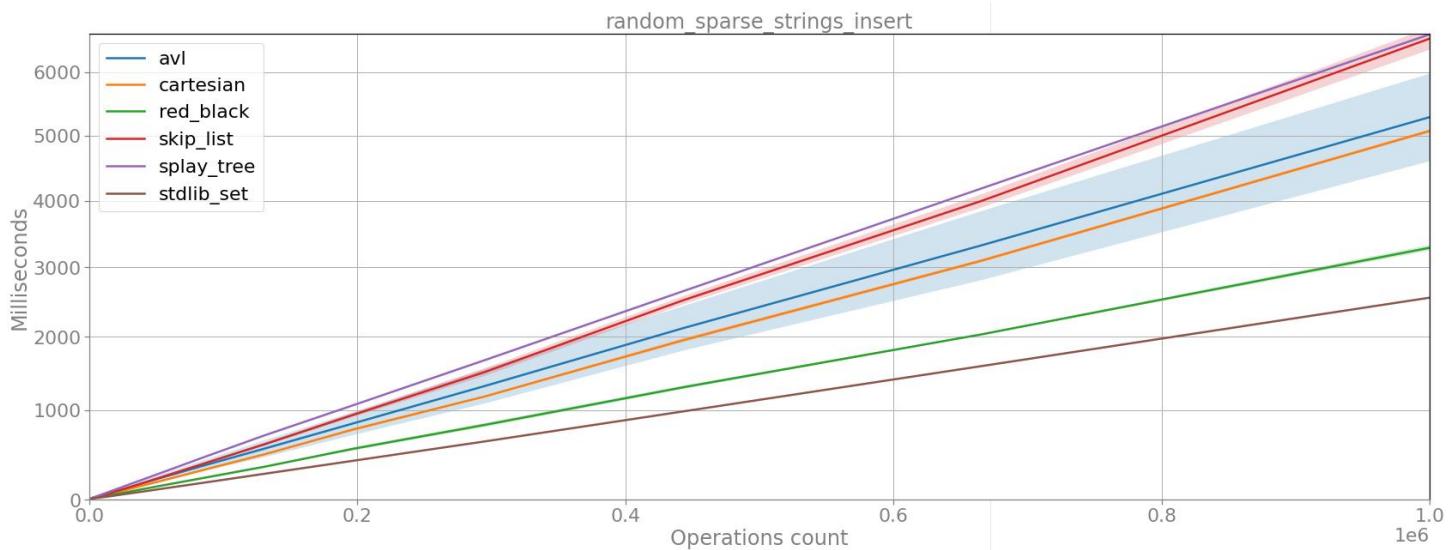


Рисунок 8.1.6. График зависимости времени работы операции вставки от количества элементов последовательности строк, сформированной RSp

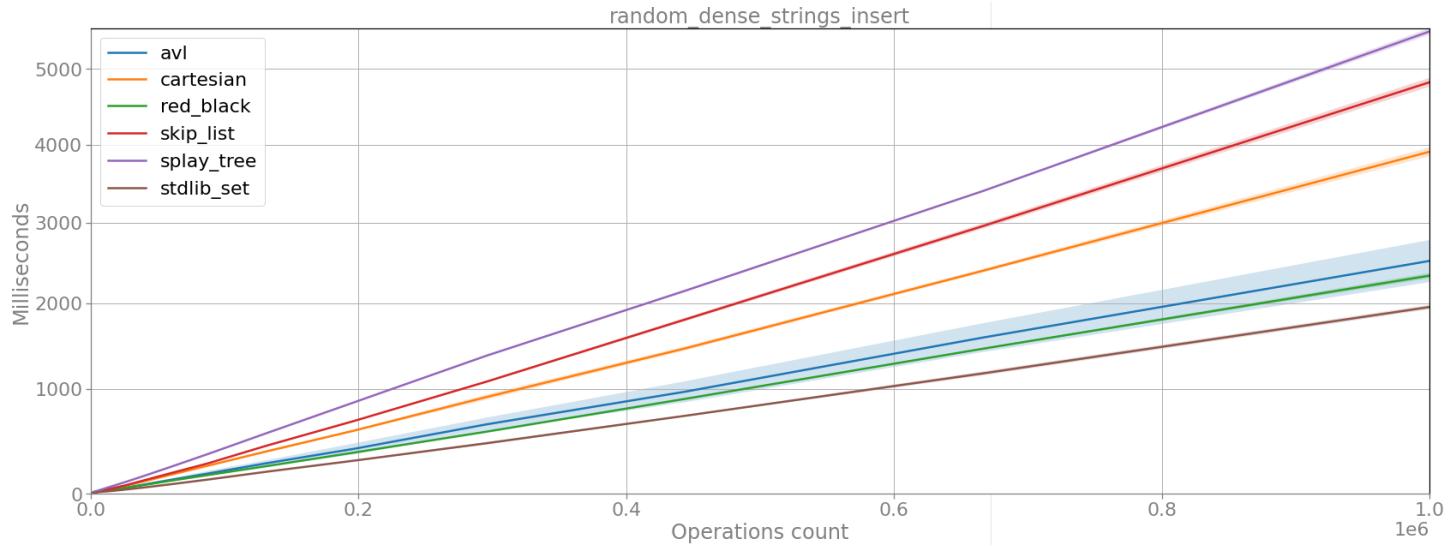


Рисунок 8.1.7. График зависимости времени работы операции вставки от количества элементов последовательности строк, сформированной RDn

Операция удаления:

После вставки мы решили сравнить время работы операций удаления элементов в алгоритмах. Для сравнения удалений мы взяли два принципиально разных случая начальных заполнений структур – последовательностями Inc и RSp.

Мы удаляли элементы от наименьшего к наибольшему:

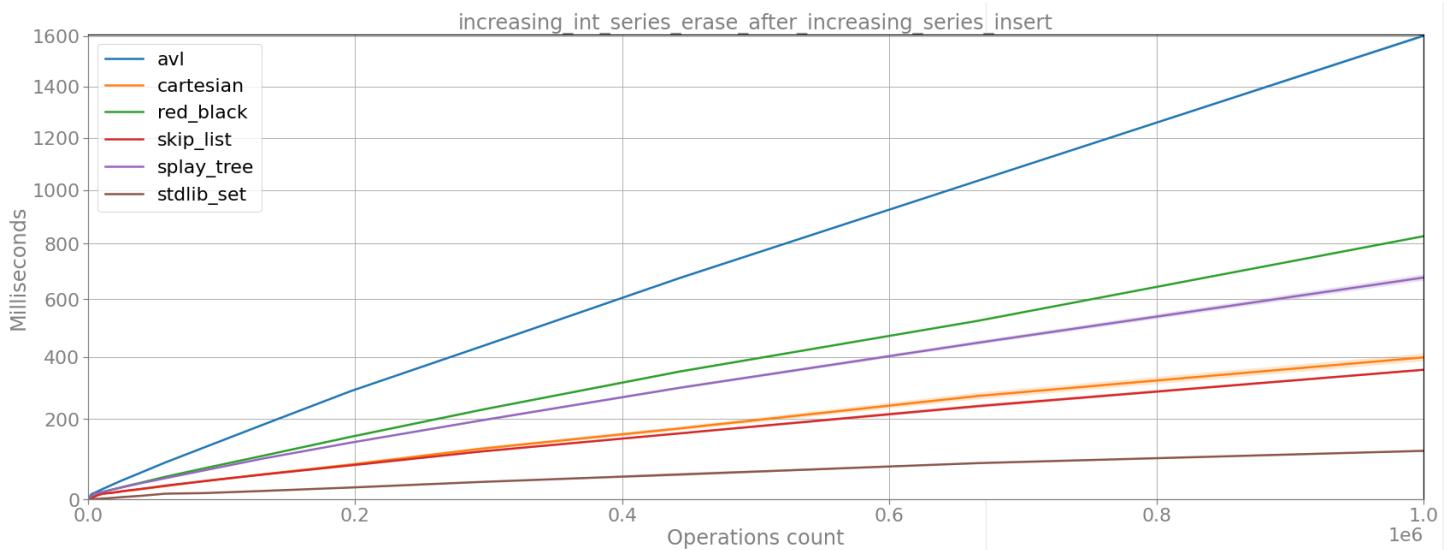


Рисунок 8.2.1. График зависимости времени работы операции удаления от количества элементов последовательности Inc. Вариант 1.

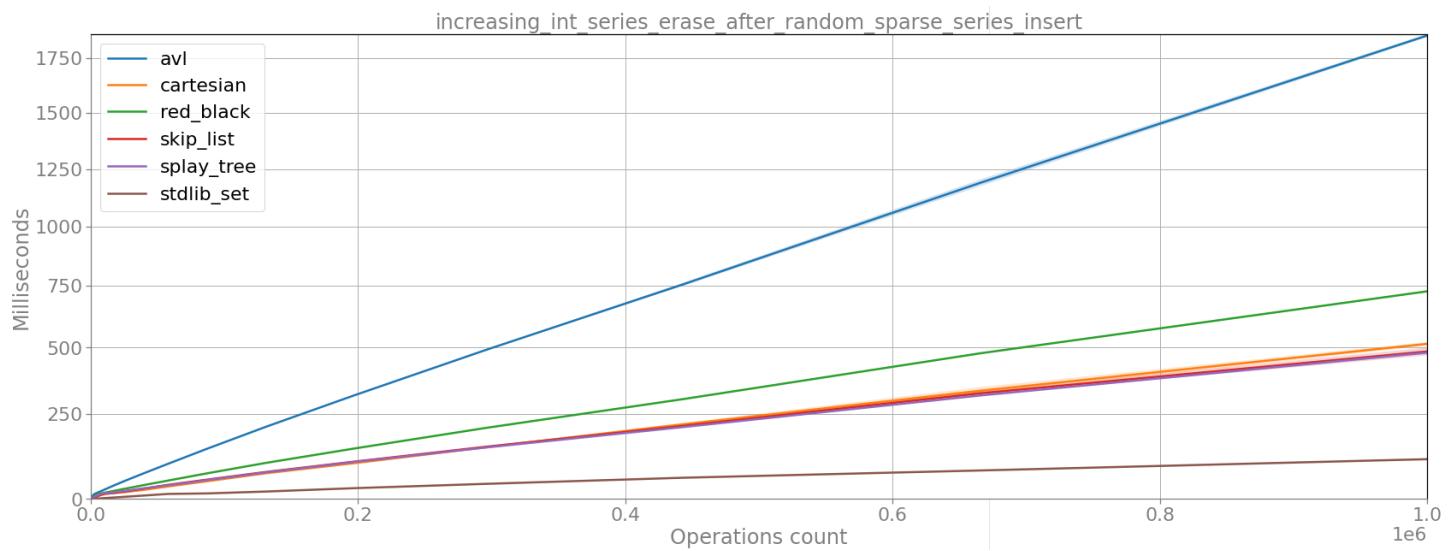


Рисунок 8.2.2. График зависимости времени работы операции удаления от количества элементов последовательности Inc. Вариант 2.

Наоборот, от наибольшего к наименьшему:

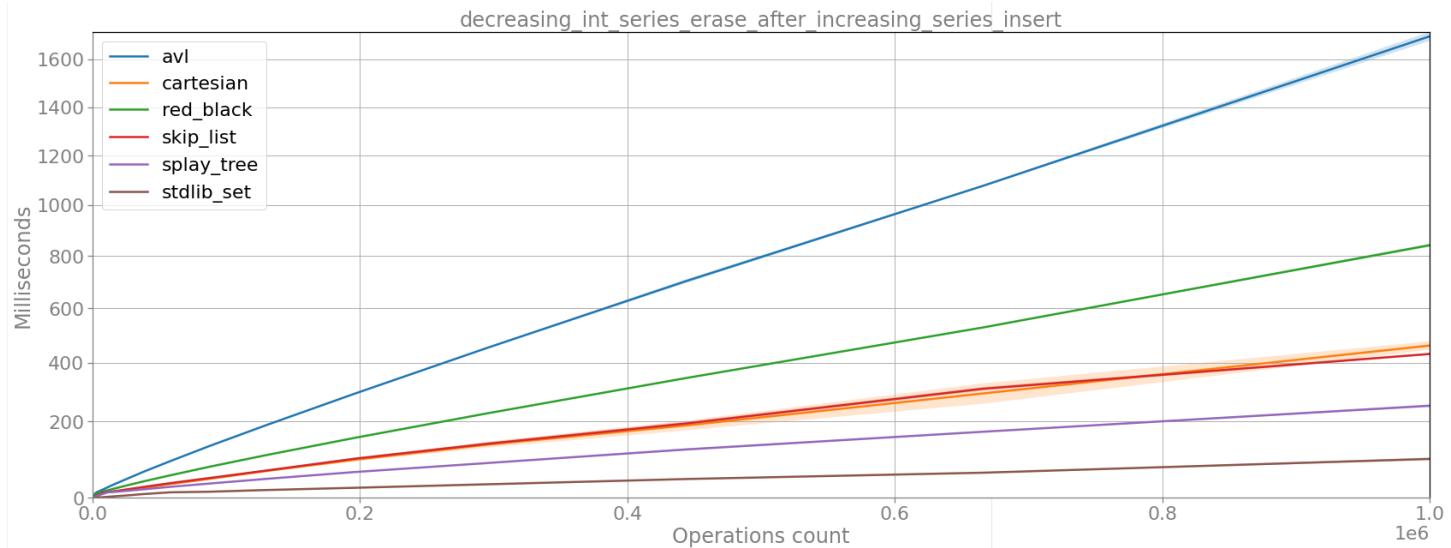


Рисунок 8.2.3. График зависимости времени работы операции удаления от количества элементов последовательности Dec. Вариант 1.

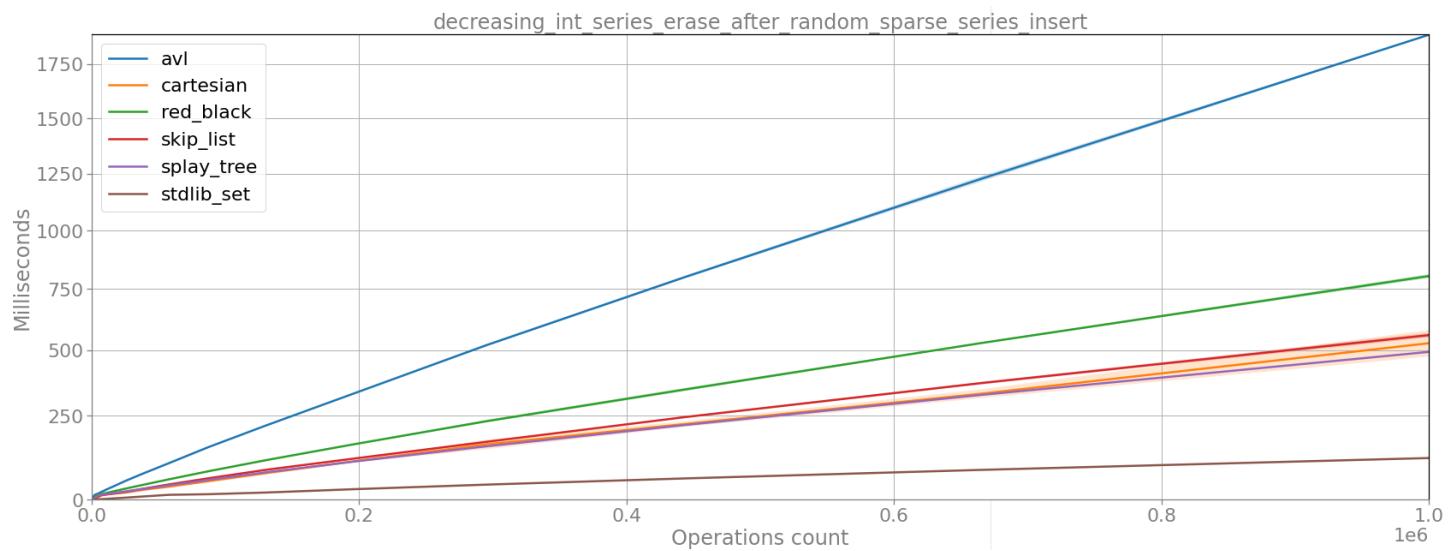


Рисунок 8.2.4. График зависимости времени работы операции удаления от количества элементов последовательности Dec. Вариант 2.

Мы также удаляли элементы в порядке Conv:

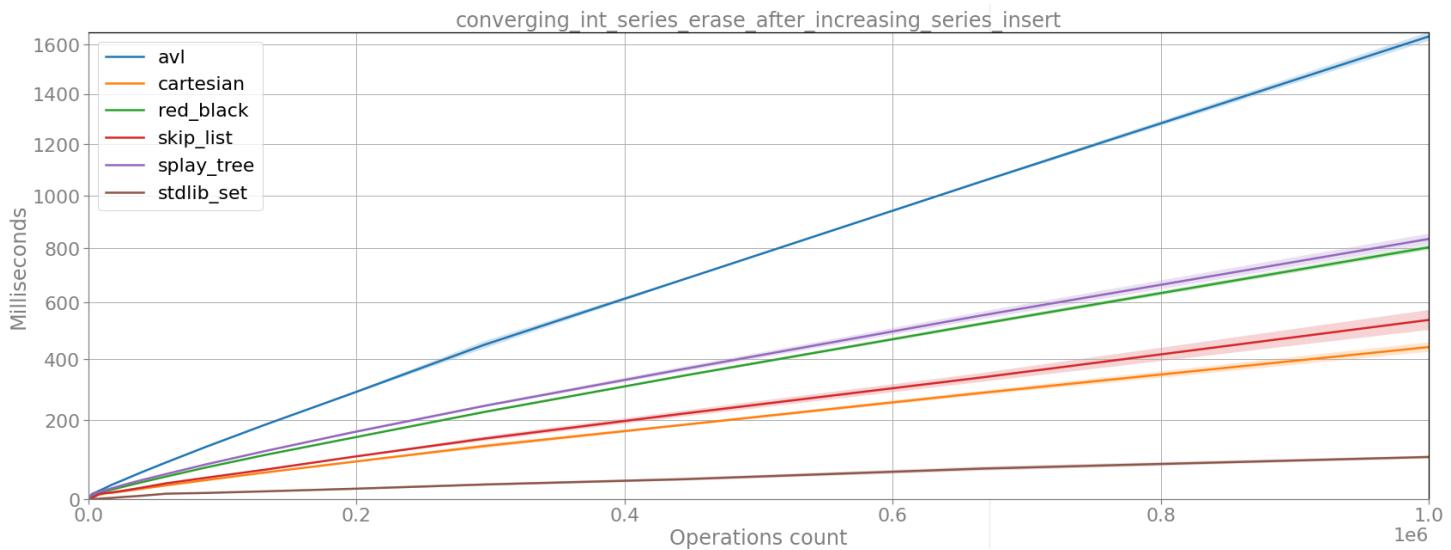


Рисунок 8.2.5. График зависимости времени работы операции удаления от количества элементов последовательности Conv. Вариант 1.

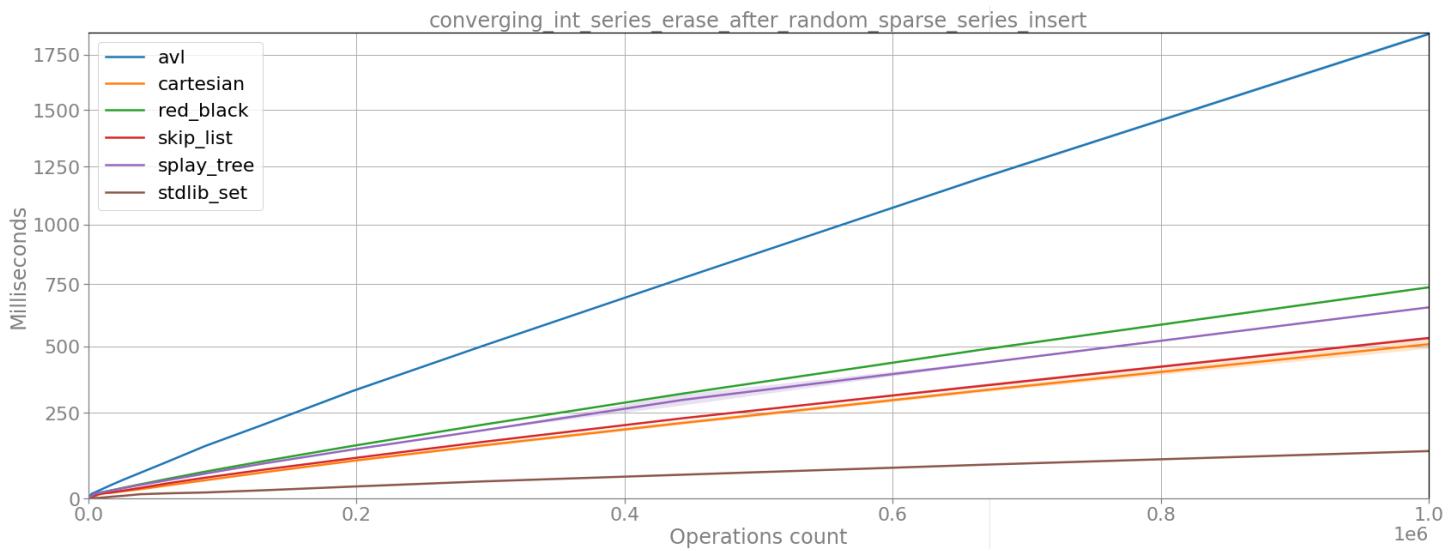


Рисунок 8.2.6. График зависимости времени работы операции удаления от количества элементов последовательности Conv. Вариант 2.

И в порядке Div:

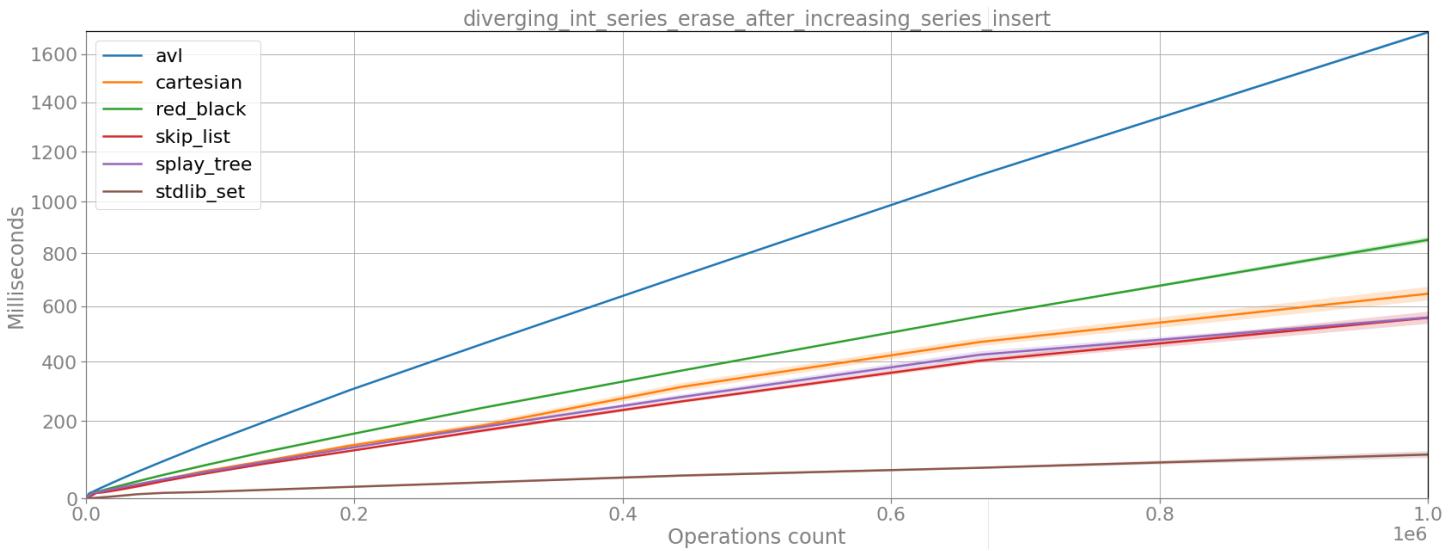


Рисунок 8.2.7. График зависимости времени работы операции удаления от количества элементов последовательности Div. Вариант 1.

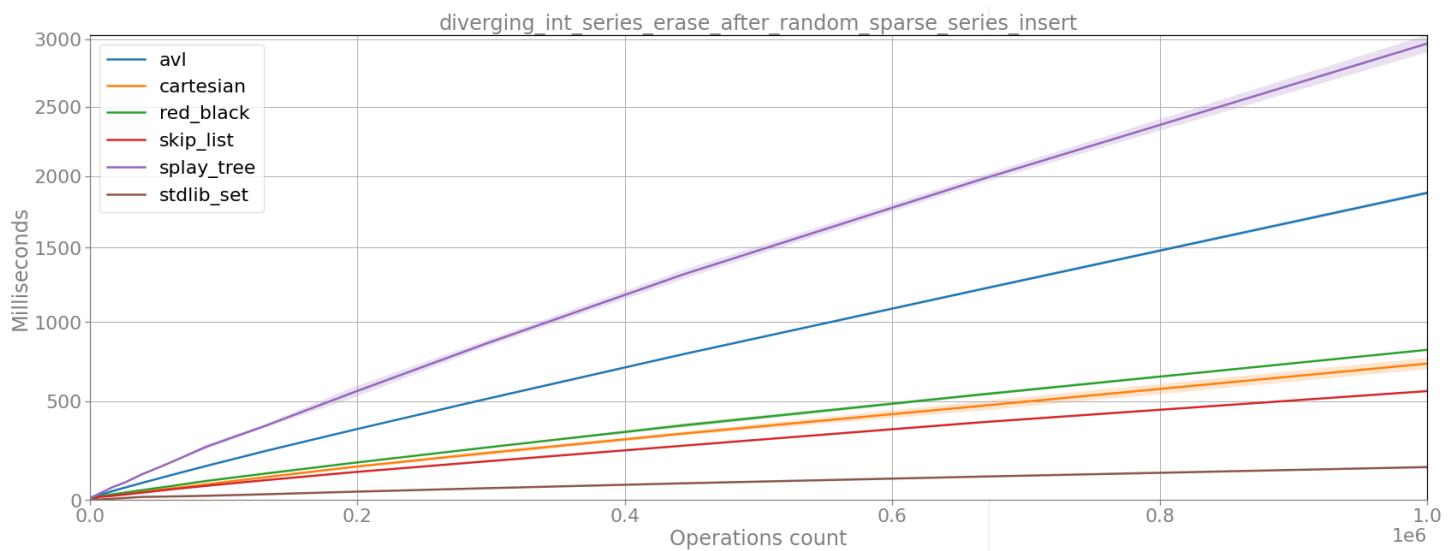


Рисунок 8.2.8. График зависимости времени работы операции удаления от количества элементов последовательности Div. Вариант 2.

Также мы пробовали удалять из структур несуществующие элементы (но близкие к тем, что записаны в структурах):

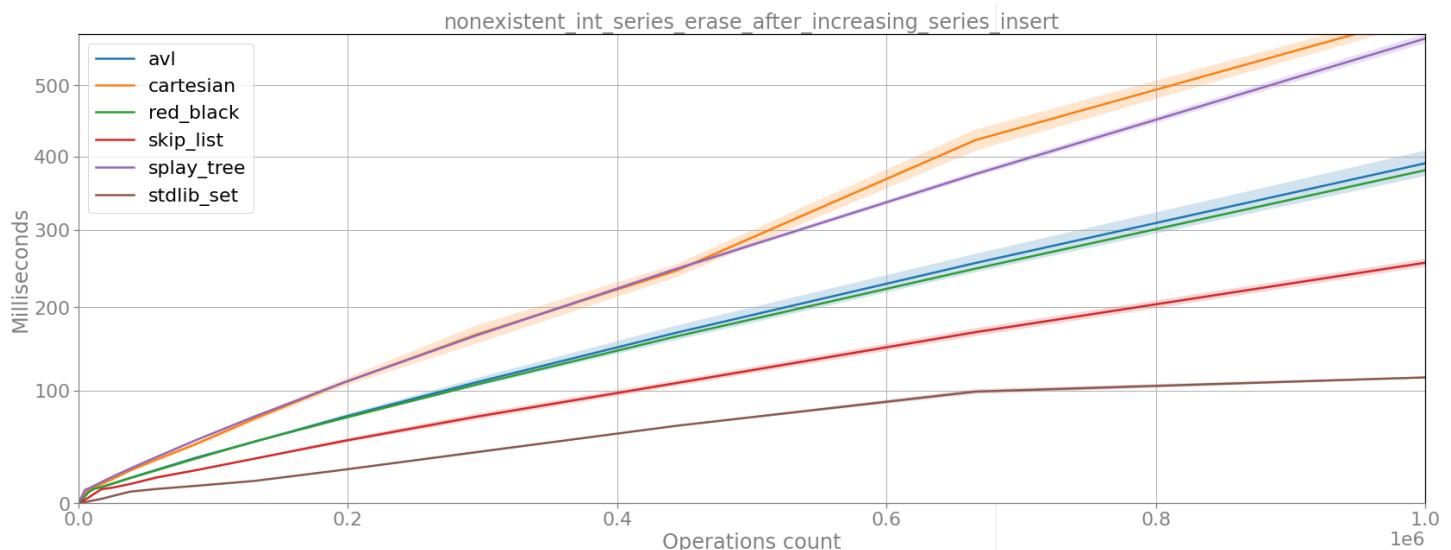


Рисунок 8.2.9. График зависимости времени работы операции удаления несуществующих элементов от количества элементов последовательности. Вариант 1.

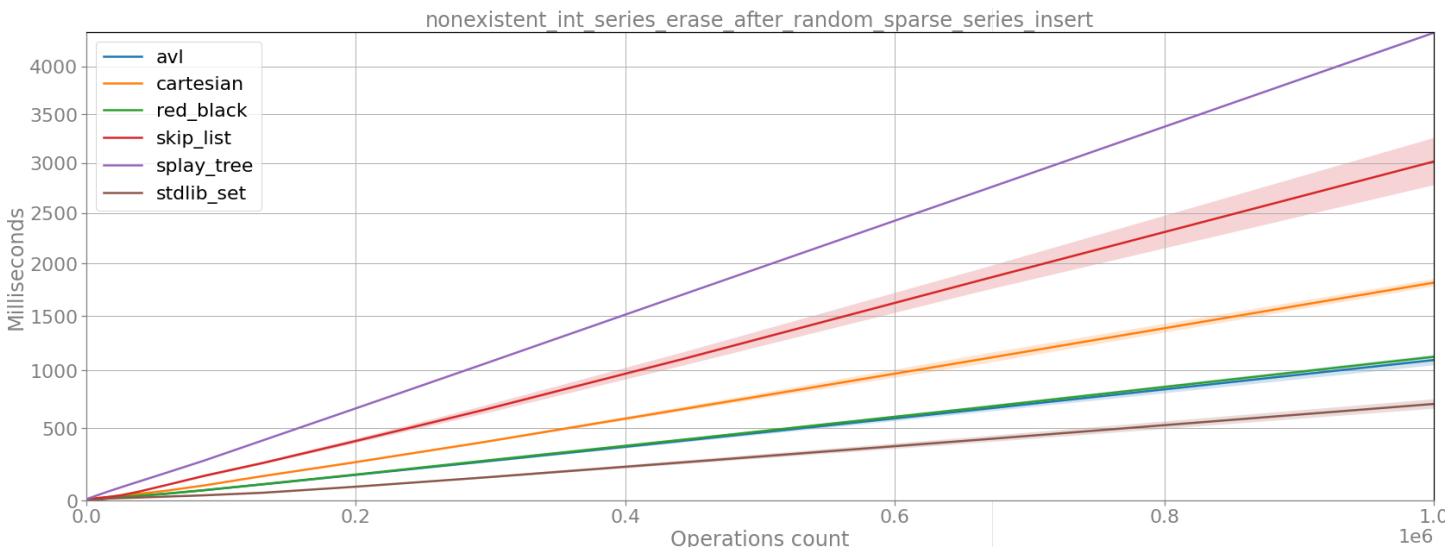


Рисунок 8.2.10. График зависимости времени работы операции удаления несуществующих элементов от количества элементов последовательности. Вариант 2.

И удаляли элементы в случайном порядке, так, как это обычно происходит в таких структурах:

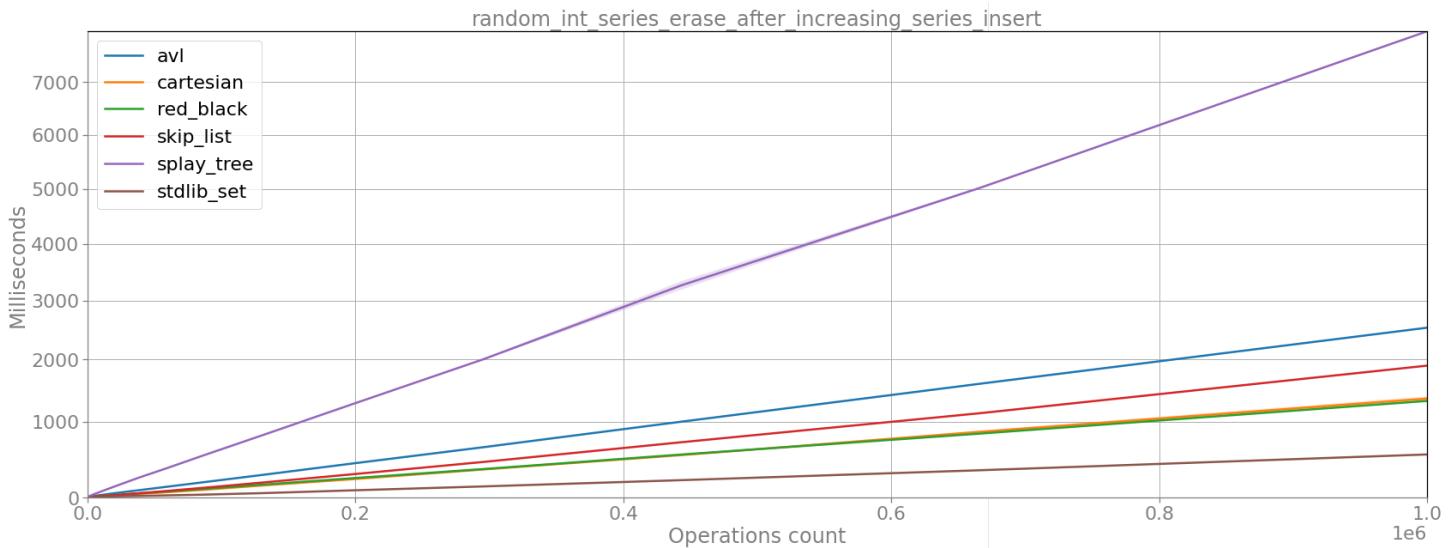


Рисунок 8.2.11. График зависимости времени работы операции удаления случайных элементов от количества элементов последовательности. Вариант 1.

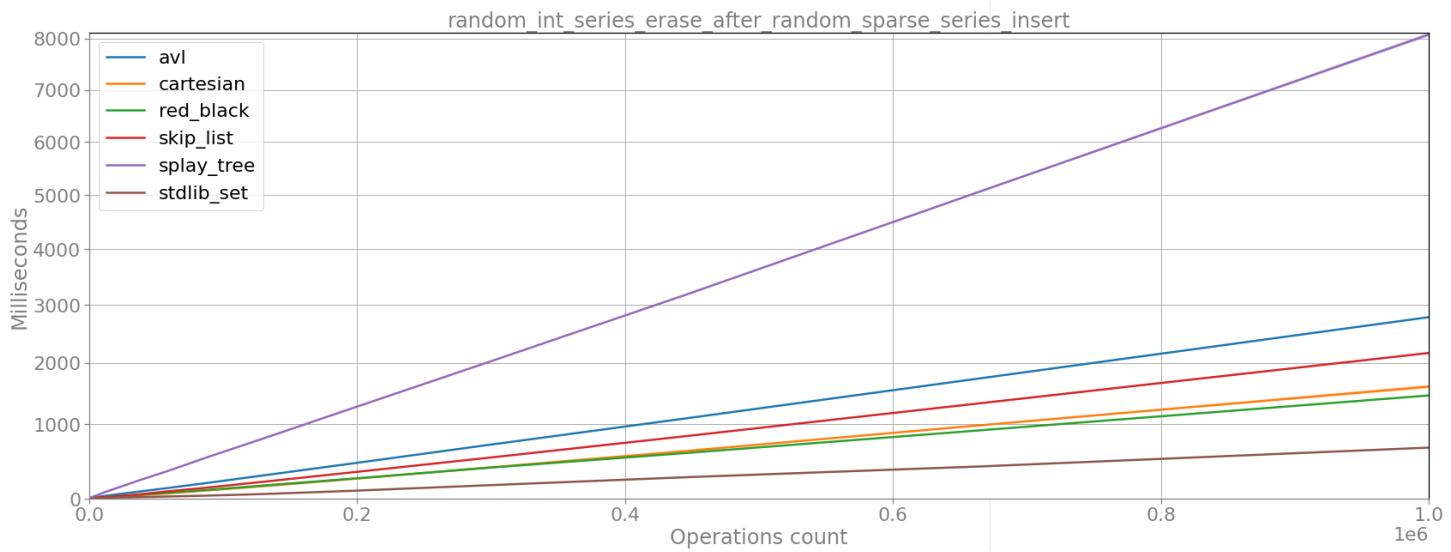


Рисунок 8.2.12. График зависимости времени работы операции удаления случайных элементов от количества элементов последовательности. Вариант 2.

После этого мы проверили операцию удаления на строковых типах, вставив в структуры близкие случайные строки (мы добавляли в конец одинаковой строки по числу из RSp, как в примере выше):

Мы удаляли строки в случайному порядке:

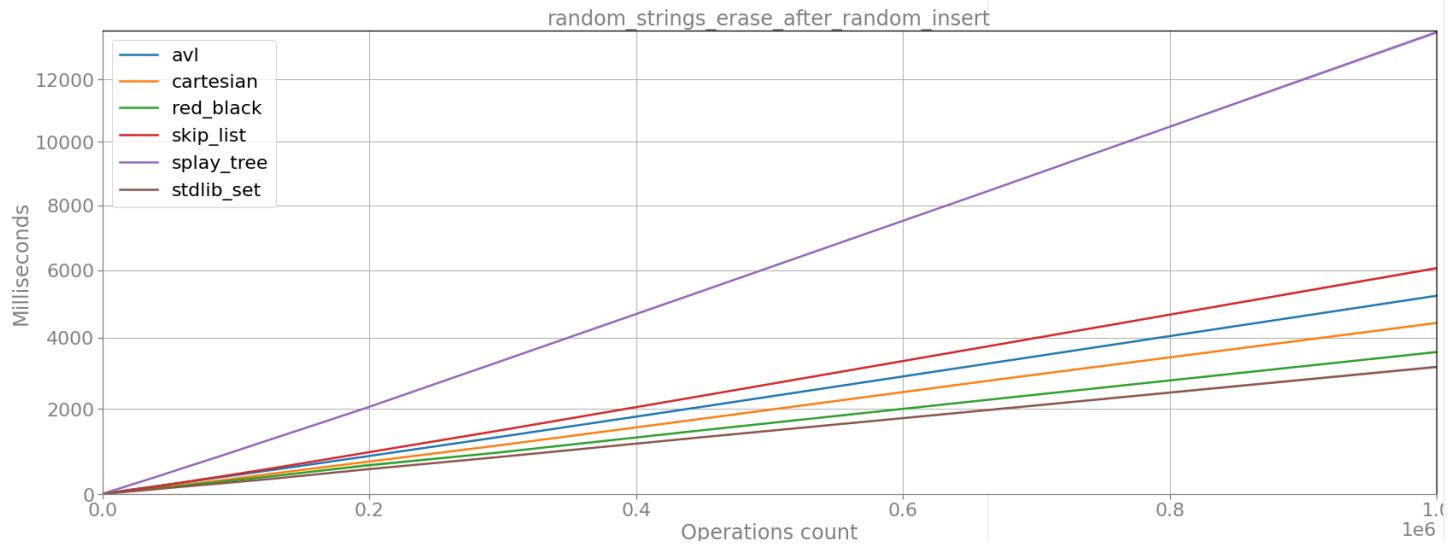


Рисунок 8.2.13. График зависимости времени работы операции удаления случайных строк от количества элементов последовательности.

А также удаляли несуществующие строки:

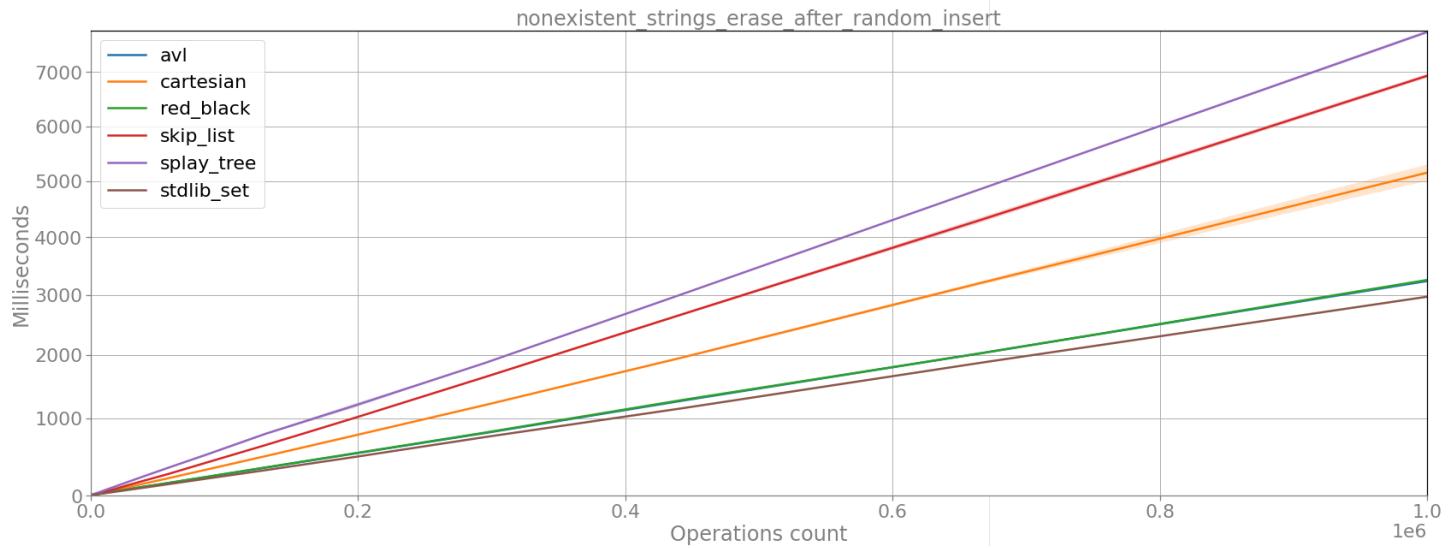


Рисунок 8.2.14. График зависимости времени работы операции удаления несуществующих строк от количества элементов последовательности.

После этого мы решили попробовать чередовать вставку элементов с удалением, чтобы максимально приблизить поведение теста к реальности:

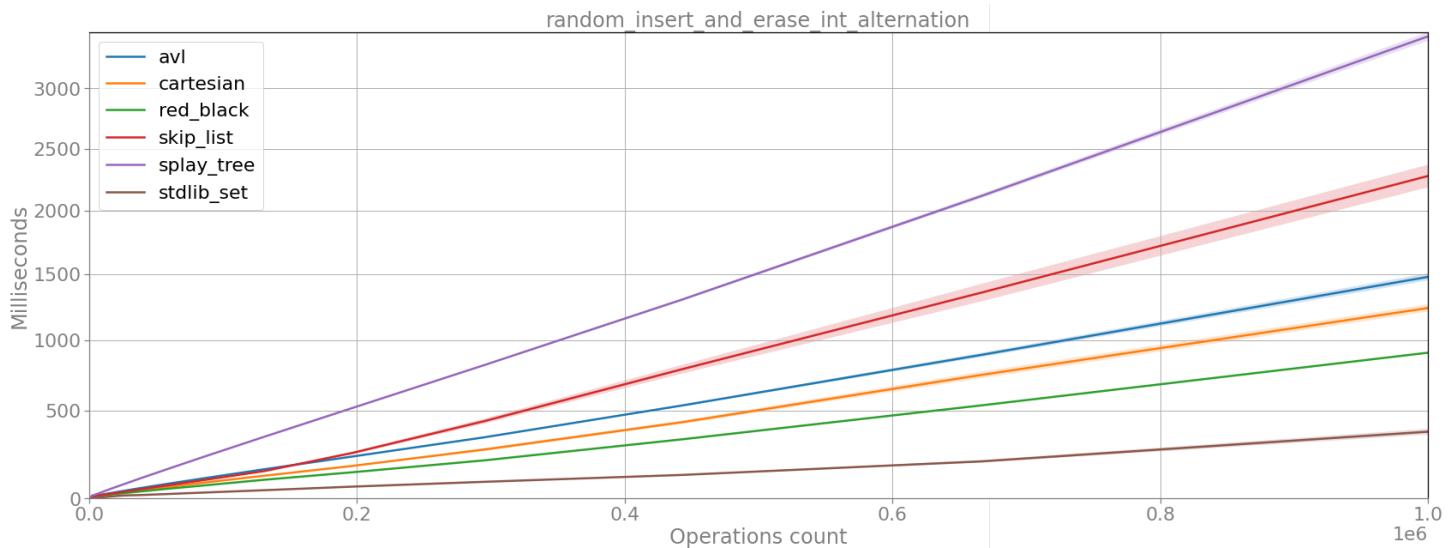


Рисунок 8.2.15. График зависимости времени работы операции удаления элементов вместе с чередованием с операцией вставки от количества элементов последовательности.

И, наконец, мы проверили время работы операции поиска элемента:

Мы искали элементы в случайном порядке:

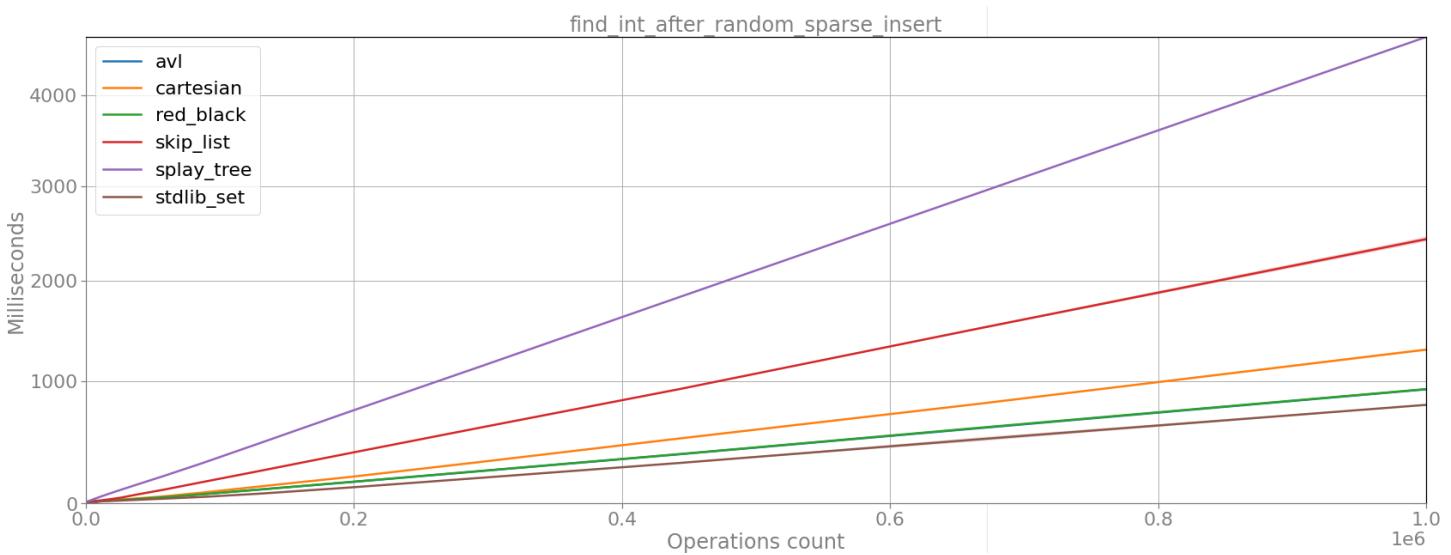


Рисунок 8.3.1. График зависимости времени работы операции поиска случайных элементов от количества элементов последовательности.

А также те, которых на самом деле нет в структурах:

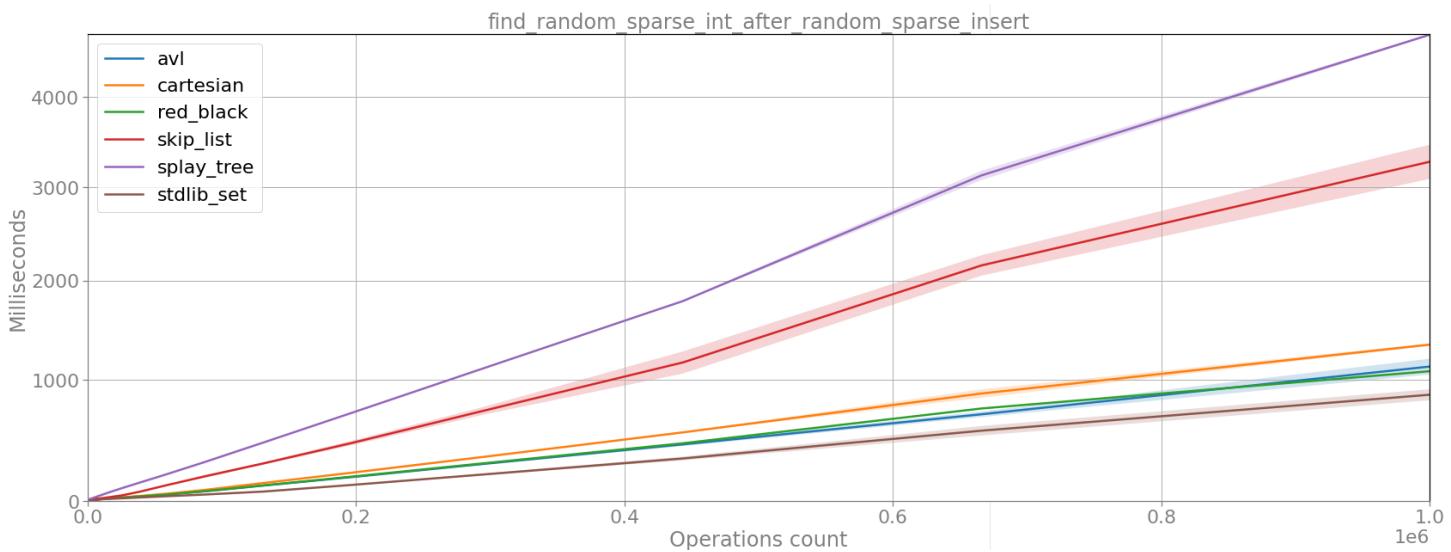


Рисунок 8.3.2. График зависимости времени работы операции поиска несуществующих элементов от количества элементов последовательности.

Также проверили время работы операции `lower_bound` – нахождение элемента, равного искомому или следующего за ним. В отличие от `find`, она всегда возвращает какой-то результат:

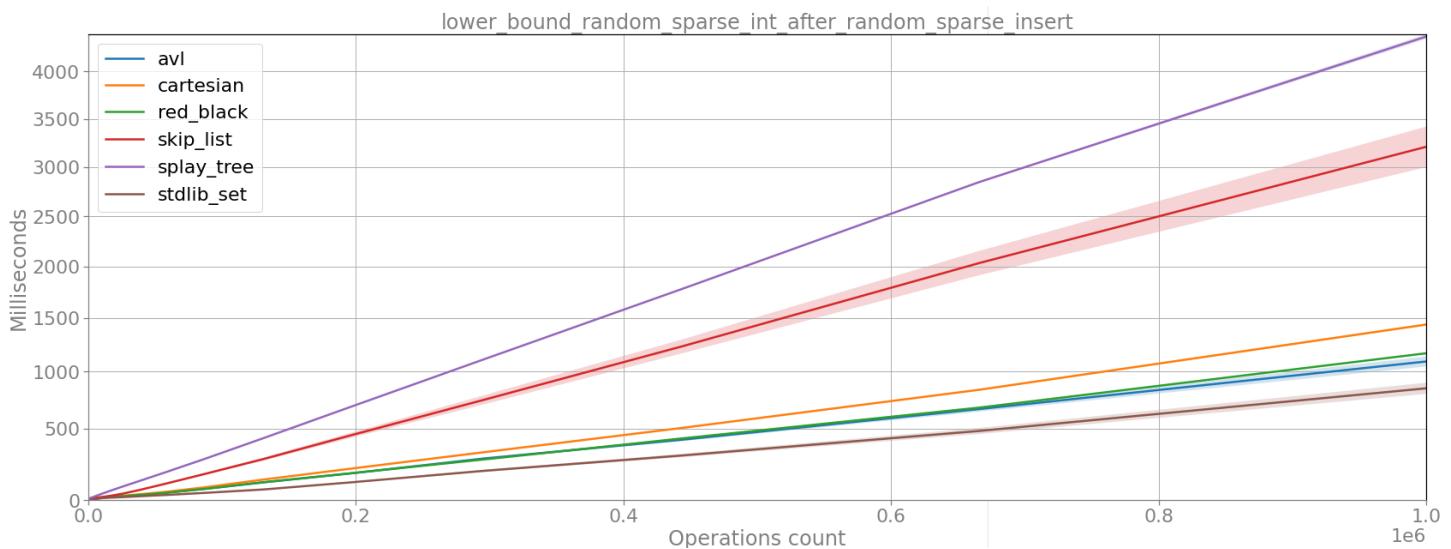


Рисунок 8.3.3. График зависимости времени работы операции `lower_bound` случайных элементов от количества элементов последовательности.

Очень важно уточнить, что учитывалось только **непосредственное время работы конкретных операций**. Например, время работы операции удаления включает в себя **только время работы операции удаления**, и не учитывается время вставки.

Выводы из экспериментов:

В выводах мы не учитываем время работы алгоритма `std::set`. Несмотря на то, что он и работает на алгоритме Red-Black tree, который реализован у нас, его реализация намного сложнее наших структур. Например, память для него выделяется массивом, а не по одному элементу, и используется повторно после освобождения. В нём также использован более легковесный способ хранения элементов, основанный на наследовании и не использованы умные указатели, сильно упрощающие жизнь простому программисту, но дающие, хоть и очень небольшие, но заметные накладные расходы.

На вставке элементов в закономерных последовательностях быстрее всех с большим отрывом работает Splay-дерево. Хуже всех на этих данных себя показывает AVL-дерево. Декартово дерево также стремится в лидеры.

На вставке элементов в случайном порядке хуже всех работают рандомизированные структуры данных – Splay-дерево и список с пропусками.

Декартово дерево, как рандомизированный алгоритм, также не в лидерах. Лучше всех на этих данных работает красно-черное дерево.

На вставке строковых данных, где на время работы сильно влияет количество сравнений, проводимое в структуре, с ещё большим отрывом от остальных побеждает красно-черное дерево.

AVL-дерево работает сильно дольше остальных практически во всех операциях удаления. Возможно в его реализации есть какая-то ошибка, которую мы не успели заметить.

При удалении элементов в закономерных последовательностях лидерство делят Декартово дерево и список с пропусками. Иногда Splay-дерево также пробивается в лидеры.

С удалением несуществующих элементов лучше всех справляются красно-черное и AVL-дерево.

На одном из рисунков можно заметить странные изгибы всех графиков. Скорее всего операционная система немного подпортила результаты, выполняя свои задачи во время тестирования, как бы мы ни старались этого избежать.

При удалении случайных элементов Splay-дерево работает практически в 4 раза медленнее остальных алгоритмов. В лидерах красно-черное и Декартово деревья.

В тесте с чередованием вставок и удалений, наиболее похожим на реальные условия, лучше всех себя ведёт красно-черное дерево. С небольшим отрывом от него идут AVL и Декартово.

В операции поиска и lower_bound красно-черное и AVL деревья практически совпадают графиками и занимают лидерство. Декартово дерево идёт с небольшим отрывом, тогда как остальные структуры данных сильно отстают по времени.

На одном из рисунков мы снова видим необычный изгиб всех графиков – снова списываем это на ОС.

Общие выводы:

Красно-чёрное дерево работает лучше всех рассмотренных нами алгоритмов практически на всех тестах. Отдадим ему заслуженное первое место в нашем эксперименте.

Декартово дерево – весьма неплохой, но рандомизированный алгоритм, который порой работает не так быстро, как хотелось бы. Впрочем, он уверенно занимает второе место.

Список с пропусками – очень простой в написании рандомизированный алгоритм. Иногда он работает быстрее лидеров, но иногда и отстает. Он получает почётное третье место в нашем эксперименте.

AVL-дерево – очень хороший алгоритм, подающий большие надежды и в ряде случаев работающий наравне с красно-чёрным деревом. Тем не менее, может быть из-за не лучшей реализации, у него есть и слабые стороны. Он занимает четвёртое место.

Splay-дерево – ещё один простой в написании алгоритм, имеющий очень неплохую оценку сложности, но из-за большой константы в этой оценке отстающий по скорости. Он оказался на последнем месте в нашем эксперименте.

Мы реализовали библиотеку, содержащую реализации пяти структур данных с логарифмическим временем поиска, добавления и удаления элементов. А именно:

- AVL tree (AVL дерево)
- Splay tree (Сплэй дерево)
- Cartesian tree (Декартово дерево)
- Red-Black tree (Красно-чёрное дерево)
- Skip-list (Список с пропусками)

Мы также разработали интерфейс, позволяющий при создании объекта выбрать алгоритм работы. Написанная структура должна позволяет помимо уже имеющихся использовать любые другие реализованные пользователем алгоритмы, соответствующие некоторым требованиям.

Мы реализовали универсальные тесты и провели эксперименты на производительность наших алгоритмов на множестве различных задач. Полученные результаты наглядно изображены на множестве графиков.

Мы предоставили информацию, необходимую для использования и модификации нашей библиотеки любым пользователем интернета.

Наша библиотека находится в свободном доступе на сайте GitLab.com.

https://gitlab.com/igor_rukhovich/trees

В ходе данной работы мы разобрались в устройстве таких структур данных как АВЛ, сплей, декартово, красно-черное деревья, а также список с пропусками. Мы на практике нашли их сильные и слабые стороны. Также, работая над проектом, мы получили множество практических навыков. Например, мы получили опыт работы с новыми возможностями языка C++ 17-го стандарта, на более профессиональном уровне познакомились с системой контроля версий, системой CI/CD, библиотекой тестирования Catch2. Что немаловажно, мы получили неоценимый опыт командной работы. Все это пригодится в ходе нашей дальнейшей профессиональной деятельности.

Литература

1. Вики Конспекты университета ИТМО. - Режим доступа:
<https://neerc.ifmo.ru/wiki> - Загл. с экрана.
2. Сообщество IT специалистов. - Режим доступа: <https://neerc.ifmo.ru/wiki> - Загл. с экрана.
3. Википедия - свободная библиотека. - Режим доступа:
<https://ru.wikipedia.org/wiki/> - Загл. с экрана.
4. Кнут Д., Искусство программирования, том 3. Сортировка и поиск 2-е изд. М.:Издательский дом «Вильямс», 2007. 824 с.
5. Вирт Н., Алгоритмы и структуры данных. СПб.: Невский Диалект, 2008. 352 с
6. Кормен Т.Х., Алгоритмы: построение и анализ / Лейзерсон Ч.И., Ривест Р.Л., Штайн К. – 2-е изд. – М.: Вильямс, 2005. – 1296 с

Приложение

Приложение A

```
#include <iostream>
#include <string>

#include "abstract_tree.h"
#include "rb_tree.h"

int main() {
    uint64_t q;
    std::cin >> q;
    std::shared_ptr<ITree<std::string>> tree =
std::make_shared<RBTree<std::string>>();

    for(uint64_t i = 0; i < q; ++i) {
        std::string words;
        std::cin >> words;
        while ( words != ".") {
            tree->insert(words);
            std::cin >> words;
        }
        std::string cur_word;
        std::cin >> cur_word;
        auto left_it = tree->find(cur_word);
        std::cin >> cur_word;
        auto right_it = tree->find(cur_word);
        right_it++;
        while(left_it != right_it){
            std::cout << *left_it << ' ';
            ++left_it;
        }
        std::cout << '\n';
    }
    return 0;
}
```

Приложение B

```
Trees/abstract_tree.h

#pragma once
#include <cassert>
#include <memory>

/***
 * Comparison between two values in tree node.
 * 'std::nullopt' is always larger than any argument.
 * @tparam T Value type
 * @param lhs LHS comparison argument
 * @param rhs RHS comparison argument
 */
```

```

* @return True if LHS < RHS
*/
template <class T>
bool operator<(const std::optional<T>& lhs, const std::optional<T>& rhs) {
    return (lhs && (!rhs || *lhs < *rhs));
}

/***
 * Abstract class that works as BST with chosen tree algorithm inside
 * @tparam T Tree value type
 */
template <class T>
class ITree {
protected:
    ITree() = default;

    /**
     * Abstract class that is needed for iterators implementation
     */
    class ITreeItImpl {
public:
    /**
     * Virtual destructor is needed for every abstract class
     */
    virtual ~ITreeItImpl() = default;

    /**
     * Implements iterator copying
     * @return Copy of the stored iterator
     */
    virtual std::shared_ptr<ITreeItImpl> Clone() const = 0;

    /**
     * Implements moving the iterator to the next element
     */
    virtual void Increment() = 0;

    /**
     * Implements moving the iterator to the previous element
     */
    virtual void Decrement() = 0;

    /**
     * Implements getting a value from the iterator
     * @return Value that the iterator points to
     */
    virtual const T Dereferencing() const = 0;

    /**
     * Implements getting a pointer to a value from the iterator
     * @return Pointer to a value that the iterator points to
     */
}

```

```

    virtual const T* Arrow() const = 0;

    /**
     * Implements iterator comparison
     * @param other Iterator to compare with
     * @return True if iterators are equal
     */
    virtual bool IsEqual(std::shared_ptr<ITreeItImpl> other) const
= 0;
};

/***
 * @return Pointer to an iterator implementation to the leftmost
element in the tree
*/
virtual std::shared_ptr<ITreeItImpl> Begin() const = 0;

/***
 * @return Pointer to an iterator implementation
 * to the next to the rightmost element in the tree
*/
virtual std::shared_ptr<ITreeItImpl> End() const = 0;

/***
 * @return Number of elements in the tree
*/
[[nodiscard]] virtual size_t Size() const = 0;

/***
 * @return True if there is no elements in the tree
*/
[[nodiscard]] virtual bool Empty() const = 0;

/***
 * Method for finding an element in the tree
 * @param value Element to find
 * @return Pointer to an iterator implementation to an element if
it is found or
 * pointer to an iterator implementation to the next to the
rightmost element in the tree
*/
virtual std::shared_ptr<ITreeItImpl> Find(const T& value) const =
0;

/***
 * Method for finding an element which would be the next to the
given element in the tree
 * @param value Element to find
 * @return Pointer to an iterator implementation to an element
 * which would be the next to the given element in the tree
*/
virtual std::shared_ptr<ITreeItImpl> LowerBound(const T& value)
const = 0;

```

```

/**
 * Method for inserting an element to the tree
 * @param value Element to insert
 */
virtual void Insert(const T& value) = 0;

/**
 * Method for deleting an element from the tree
 * @param value Element to delete
 */
virtual void Erase(const T& value) = 0;

/**
 * Method for deleting all the elements from the tree
 */
virtual void Clear() = 0;

public:
    /**
     * Virtual destructor is needed for every class with virtual
     * methods
     */
    virtual ~ITree() = default;

    /**
     * ITree iterator class.
     * Satisfies C++ LegacyBidirectionalIterator named requirement
     * Uses Pimpl pattern
     */
    class iterator {
public:
    iterator() = delete;
    explicit iterator(std::shared_ptr<ITreeItImpl> pimpl) :
        pimpl_(pimpl) {
    }
    iterator(const iterator& other) : pimpl_(other.pimpl_) {
    }
    ~iterator() = default;

    iterator& operator++() {
        pimpl_->Increment();
        return *this;
    }
    iterator operator++(int) {
        iterator cpy(pimpl_->Clone());
        pimpl_->Increment();
        return cpy;
    }
    iterator& operator--() {
        pimpl_->Decrement();
        return *this;
    }
}

```

```

iterator operator--(int) {
    iterator cpy(pimpl_->Clone());
    pimpl_->Decrement();
    return cpy;
}

const T operator*() const {
    return pimpl_->Dereferencing();
}
const T* operator->() const {
    return pimpl_->Arrow();
}

bool operator==(const iterator& other) const {
    return pimpl_->IsEqual(other.pimpl_);
}
bool operator!=(const iterator& other) const {
    return !pimpl_->IsEqual(other.pimpl_);
}

private:
    /**
     * pointer to the specific implementation
     */
    std::shared_ptr<ITreeItImpl> pimpl_;
};

/** @return Iterator to the leftmost element in the tree */
iterator begin() const {
    return iterator(Begin());
}

/** @return Iterator to the next to the rightmost element in the tree */
iterator end() const {
    return iterator(End());
}

/** @return Number of elements in the tree */
[[nodiscard]] size_t size() const {
    return Size();
}

/** @return True if there is no elements in the tree */
[[nodiscard]] bool empty() const {

```

```

        return Empty();
    }

/***
 * Method for finding an element in the tree
 * @param value Value to find
 * @return Iterator to an element if it is found or
 * iterator to the next to the rightmost element in the tree
 */
iterator find(const T& value) const {
    return iterator(Find(value));
}

/***
 * Method for finding an element which would be the next to the
given element in the tree
 * @param value Value to find
 * @return Iterator to an element which would be the next to the
given element in the tree
 */
iterator lower_bound(const T& value) const {
    return iterator(LowerBound(value));
}

/***
 * Method for inserting an element to the tree
 * @param value Value to insert
 * It may already be stored in the tree
 */
void insert(const T& value) {
    Insert(value);
}

/***
 * Method for deleting an element from the tree
 * @param value Value to delete
 * It might not be stored in the tree
 */
void erase(const T& value) {
    Erase(value);
}

/***
 * Method for deleting all the elements from the tree
 */
void clear() {
    Clear();
}
};

trees/avl_tree.h
#endif
#endif

```

#pragma once

```

#include <initializer_list>
#include <memory>

template <class T>
class ITree;

template <class T>
class AVLTree : public ITree<T> {
private:
    typedef typename ITree<T>::ITreeItImpl BaseImpl;

public:
    struct Node {
        Node() {
            left_ = nullptr;
            right_ = nullptr;
            parent_ = std::weak_ptr<Node>();
            value_ = std::nullopt;
            height_ = 1;
        }

        explicit Node(const T &value) : value_(value) {
            left_ = nullptr;
            right_ = nullptr;
            parent_ = std::weak_ptr<Node>();
            height_ = 1;
        }

        Node(const Node &other) : value_(other.value_) {
            left_ = other.left_;
            right_ = other.right_;
            parent_ = other.parent_;
            height_ = other.height_;
        }

        std::shared_ptr<Node> left_;
        std::shared_ptr<Node> right_;
        std::weak_ptr<Node> parent_;
        std::optional<T> value_;
        uint8_t height_;
    };
};

AVLTree() {
    end_ = std::make_shared<Node>();
    begin_ = end_;
    root_ = end_;
    size_ = 0;
}

template <class InitIterator>
AVLTree(InitIterator begin, InitIterator end) : AVLTree() {
    for (InitIterator cur(begin); cur != end; ++cur) {
        Insert(*cur);
    }
}

```

```

        }
    }
AVLTree(std::initializer_list<T> list) : AVLTree() {
    for (const T &value : list) {
        Insert(value);
    }
}

AVLTree(const AVLTree &other) : AVLTree() {
    for (const T &value : other) {
        Insert(value);
    }
}
AVLTree(AVLTree &&other) noexcept : AVLTree() {
    std::swap(root_, other.root_);
    std::swap(begin_, other.begin_);
    std::swap(end_, other.end_);
    std::swap(size_, other.size_);
}
AVLTree(std::shared_ptr<ITree<T>> other) :
AVLTree(*dynamic_cast<AVLTree<T>*>(other.get())))
{
AVLTree &operator=(const AVLTree &other) {
    if (root_ == other.root_) {
        return *this;
    }
    end_ = std::make_shared<Node>();
    root_ = end_;
    begin_ = end_;
    size_ = 0;
    for (const T &value : other) {
        Insert(value);
    }
    return *this;
}
AVLTree &operator=(AVLTree &&other) noexcept {
    if (root_ == other.root_) {
        return *this;
    }
    std::swap(root_, other.root_);
    std::swap(begin_, other.begin_);
    std::swap(end_, other.end_);
    std::swap(size_, other.size_);
    return *this;
}
~AVLTree() override {
    root_ = nullptr;
    begin_ = nullptr;
    end_ = nullptr;
    size_ = 0;
}

```

```

[[nodiscard]] size_t Size() const override {
    return size_;
}
[[nodiscard]] bool Empty() const override {
    return !static_cast<bool>(size_);
}

std::shared_ptr<BaseImpl> Find(const T &value) const override {
    return FindImpl(root_, value);
}
std::shared_ptr<BaseImpl> LowerBound(const T &value) const
override {
    std::optional val(value);
    return LowerBoundImpl(root_, val);
}

void Insert(const T &value) override {
    std::shared_ptr<Node> new_node =
std::make_shared<Node>(value);
    if (InsertImplementation(new_node)) {
        ++size_;
    }
}
void Erase(const T &value) override {
    auto nodeInTree =
std::static_pointer_cast<AVLTreeItImpl>(Find(value));
    if (nodeInTree->IsEqual(End())) {
        return;
    }

    EraseImplementation(nodeInTree->GetPointer());
    --size_;
}

void Clear() override {
    root_ = std::shared_ptr<Node>();
    begin_ = root_;
    end_ = root_;
    size_ = 0;
}

private:
    std::shared_ptr<Node> begin_;
    std::shared_ptr<Node> end_;
    std::shared_ptr<Node> root_;
    size_t size_;

/*
 * -----ITERATOR IMPLEMENTATION-----
 */
class AVLTreeItImpl : public BaseImpl {

```

```

public:
    AVLTreeItImpl() = delete;
    explicit AVLTreeItImpl(std::shared_ptr<Node> pointer) :
it_(pointer) {
}
AVLTreeItImpl(const AVLTreeItImpl &other) : it_(other.it_) {

std::shared_ptr<BaseImpl> Clone() const override {
    return std::make_shared<AVLTreeItImpl>(*this);
}
void Increment() override {
    if (!it_->value_) {
        throw std::runtime_error("Index out of range while
increasing");
    }
    if (it_->right_) {
        it_ = it_->right_;
        while (it_->left_) {
            it_ = it_->left_;
        }
    } else {
        auto parent = it_->parent_.lock();
        while (parent && parent->right_ == it_) {
            it_ = parent;
            parent = it_->parent_.lock();
        }
        it_ = parent;
    }
}
void Decrement() override {
    if (it_->left_) {
        it_ = it_->left_;
        while (it_->right_) {
            it_ = it_->right_;
        }
    } else {
        auto parent = it_->parent_.lock();
        while (parent && parent->left_ == it_) {
            it_ = parent;
            parent = it_->parent_.lock();
        }
        if (parent) {
            it_ = parent;
        } else {
            throw std::runtime_error("Index out of range while
decreasing");
        }
    }
}

const T Dereferencing() const override {
    if (it_ && !it_->value_) {

```

```

        throw std::runtime_error("Index out of range on
operator*");
    }
    return it_->value_.value();
}
const T *Arrow() const override {
    if (it_ && !it_->value_) {
        throw std::runtime_error("Index out of range on
operator->");
    }
    return &it_->value_.value();
}

bool IsEqual(std::shared_ptr<BaseImpl> other) const override {
    auto casted =
std::dynamic_pointer_cast<AVLTreeItImpl>(other);
    if (!casted) {
        return false;
    }
    return it_ == casted->it_;
}
std::shared_ptr<Node> GetPointer() {
    return it_;
}

private:
    std::shared_ptr<Node> it_;
};

std::shared_ptr<BaseImpl> Begin() const override {
    return std::make_shared<AVLTreeItImpl>(begin_);
}
std::shared_ptr<BaseImpl> End() const override {
    return std::make_shared<AVLTreeItImpl>(end_);
}

/* -----
 * -----PRIVATE FUNCTIONS-----
 * -----
 */
std::shared_ptr<BaseImpl> FindImpl(std::shared_ptr<Node> from,
                                    const std::optional<T>
&value) const {
    while (from) {
        if (value < from->value_) {
            from = from->left_;
        } else if (from->value_ < value) {
            from = from->right_;
        } else {
            return std::make_shared<AVLTreeItImpl>(from);
        }
    }
}

```

```

        return End();
    }

    static std::shared_ptr<BaseImpl>
LowerBoundImpl(std::shared_ptr<Node> from,
                           const
std::optional<T> &value) {
    while (true) {
        if (value < from->value_) {
            if (from->left_) {
                from = from->left_;
            } else {
                return std::make_shared<AVLTreeItImpl>(from);
            }
        } else if (from->value_ < value) {
            if (from->right_) {
                from = from->right_;
            } else {
                auto impl = std::make_shared<AVLTreeItImpl>(from);
                impl->Increment();
                return impl;
            }
        } else {
            return std::make_shared<AVLTreeItImpl>(from);
        }
    }
}

void LeftRotate(std::shared_ptr<Node> from) {
    auto right_node = from->right_;

    from->right_ = right_node->left_;
    if (right_node->left_) {
        right_node->left_->parent_ = from;
    }

    right_node->left_ = from;
    right_node->parent_ = from->parent_;

    auto parent = from->parent_.lock();
    if (parent) {
        if (parent->right_ == from) {
            parent->right_ = right_node;
        } else {
            parent->left_ = right_node;
        }
    } else {
        root_ = right_node;
    }

    from->parent_ = right_node;
    RecalcHeight(from);
}

```

```

        RecalcHeight(right_node);
    }
void RightRotate(std::shared_ptr<Node> from) {
    auto left_node = from->left_;

    from->left_ = left_node->right_;
    if (left_node->right_) {
        left_node->right_->parent_ = from;
    }

    left_node->right_ = from;
    left_node->parent_ = from->parent_;

    auto parent = from->parent_.lock();
    if (parent) {
        if (parent->right_ == from) {
            parent->right_ = left_node;
        } else {
            parent->left_ = left_node;
        }
    } else {
        root_ = left_node;
    }

    from->parent_ = left_node;
    RecalcHeight(from);
    RecalcHeight(left_node);
}

int AVLBalanceFactor(std::shared_ptr<Node> node) const {
    uint8_t hl, hr;

    if (!node->left_) {
        hl = 0;
    } else {
        hl = node->left_->height_;
    }

    if (!node->right_) {
        hr = 0;
    } else {
        hr = node->right_->height_;
    }
    return hr - hl;
}
void AVLFixBalance(std::shared_ptr<Node> node) {
    RecalcHeight(node);
    if (AVLBalanceFactor(node) == 2) {
        if (node->right_ && AVLBalanceFactor(node->right_) < 0) {
            RightRotate(node->right_);
        }
        LeftRotate(node);
    }
}

```

```

    }

    if (AVLBalanceFactor(node) == -2) {
        if (node->left_ && AVLBalanceFactor(node->left_) > 0) {
            LeftRotate(node->left_);
        }
        RightRotate(node);
        return;
    }
}

void RecalcHeight(std::shared_ptr<Node> node) {
    if (!node) {
        return;
    }

    uint8_t hl, hr;

    if (!node->left_) {
        hl = 0;
    } else {
        hl = node->left_->height_;
    }

    if (!node->right_) {
        hr = 0;
    } else {
        hr = node->right_->height_;
    }

    if (hl > hr) {
        node->height_ = hl + 1;
    } else {
        node->height_ = hr + 1;
    }
}

bool InsertImplementation(const std::shared_ptr<Node> &new_node) {
    if (!(root_)) {
        root_ = new_node;
        RecalcBegin();
        return true;
    }

    auto cur_node = root_;
    auto next_node = cur_node;
    while (next_node) {
        cur_node = next_node;
        if (new_node->value_ < cur_node->value_) {
            next_node = cur_node->left_;
        } else if (cur_node->value_ < new_node->value_) {
            next_node = cur_node->right_;
        } else {

```

```

        return false;
    }
}

if (new_node->value_ < cur_node->value_) {
    cur_node->left_ = new_node;
} else {
    cur_node->right_ = new_node;
}
new_node->parent_ = cur_node;

while (cur_node) {
    AVLFixBalance(cur_node);
    cur_node = cur_node->parent_.lock();
}
RecalcBegin();
return true;
}
void EraseImplementation(std::shared_ptr<Node> delete_node) {
    auto parent = delete_node->parent_.lock();
    std::shared_ptr<Node> child_node;

    // Node doesn't have children
    if (!delete_node->right_ && !delete_node->left_) {
        if (parent) {
            if (parent->left_ == delete_node) {
                parent->left_ = nullptr;
            } else {
                parent->right_ = nullptr;
            }
        } else {
            root_ = nullptr;
        }
        // Node has only 1 child
    } else if ((delete_node->right_ && !delete_node->left_) ||
               (!delete_node->right_ && delete_node->left_)) {

        child_node = delete_node->right_ ? delete_node->right_ :
delete_node->left_;
        if (!parent) {
            root_ = child_node;

        } else {
            if (parent->left_ == delete_node) {
                parent->left_ = child_node;
            } else {
                parent->right_ = child_node;
            }
            child_node->parent_ = parent;
        }
    } else {
        std::shared_ptr<Node> swap_node = delete_node->right_;
        while (swap_node->left_) {

```

```

        swap_node = swap_node->left_;
    }
    parent = swap_node->parent_.lock();
    if (swap_node == delete_node->right_) {
        SwapWithChild(delete_node, swap_node);
        parent = swap_node;
    } else {
        SwapWithOffspring(delete_node, swap_node);
    }
}

while (parent) {
    AVLFixBalance(parent);
    parent = parent->parent_.lock();
}

RecalcBegin();
}

// Set begin_ after modification
void RecalcBegin() {
    auto node = root_;
    while (node->left_) {
        node = node->left_;
    }
    begin_ = node;
}

// When swap node is child
void SwapWithChild(std::shared_ptr<Node> from_node,
std::shared_ptr<Node> swap_node) {
    auto parent = from_node->parent_.lock();
    if (parent) {
        if (parent->left_ == from_node) {
            parent->left_ = swap_node;
        } else {
            parent->right_ = swap_node;
        }
    } else {
        root_ = swap_node;
    }
    swap_node->parent_ = parent;

    swap_node->left_ = from_node->left_;
    if (from_node->left_) {
        from_node->left_->parent_ = swap_node;
    }
}
// When swap node is not child
void SwapWithOffspring(std::shared_ptr<Node> from_node,
std::shared_ptr<Node> swap_node) {
    auto from_parent = from_node->parent_.lock();
    if (from_parent) {

```

```

        if (from_parent->left_ == from_node) {
            from_parent->left_ = swap_node;
        } else {
            from_parent->right_ = swap_node;
        }
    } else {
        root_ = swap_node;
    }

    auto swap_parent = swap_node->parent_.lock();
    swap_parent->left_ = swap_node->right_;
    if (swap_node->right_) {
        swap_node->right_->parent_ = swap_parent;
    }

    swap_node->parent_ = from_node->parent_;
    swap_node->right_ = from_node->right_;
    from_node->right_->parent_ = swap_node;

    swap_node->left_ = from_node->left_;
    if (from_node->left_) {
        from_node->left_->parent_ = swap_node;
    }
}
};

Trees/cartesian_tree.h

```

```

#pragma once
#include <initializer_list>
#include <exception>
#include <memory>
#include <random>
#include <optional>

template <class T>
class ITree;

template <class T>
class CartesianTree : public ITree<T> {
private:
    typedef typename ITree<T>::ITreeItImpl BaseImpl;

    class Random {
public:
    static uint32_t Next() {
        static Random rand = Random();
        return rand.dist_(rand.gen_);
    }
};

private:
    Random() {

```

```

        std::random_device device;
        gen_ = std::mt19937(device());
        dist_ =
            std::uniform_int_distribution<uint32_t>(1,
std::numeric_limits<uint32_t>::max());
    }

    std::mt19937 gen_;
    std::uniform_int_distribution<uint32_t> dist_;
};

public:
    struct Node {
        Node() {
            left_ = nullptr;
            right_ = nullptr;
            parent_ = std::weak_ptr<Node>();
            priority_ = Random::Next();
            value_ = std::nullopt;
        }
        explicit Node(const T& value) : value_(value) {
            left_ = nullptr;
            right_ = nullptr;
            parent_ = std::weak_ptr<Node>();
            priority_ = Random::Next();
        }
        Node(const Node& other) : value_(other.value_) {
            left_ = other.left_;
            right_ = other.right_;
            parent_ = other.parent_;
            priority_ = other.priority_;
        }
        std::shared_ptr<Node> left_;
        std::shared_ptr<Node> right_;
        std::weak_ptr<Node> parent_;
        uint32_t priority_;
        std::optional<T> value_;
    };
}

CartesianTree() {
    root_ = std::make_shared<Node>();
    begin_ = root_;
    end_ = root_;
    size_ = 0;
}

template <class InitIterator>
CartesianTree(InitIterator begin, InitIterator end) :
CartesianTree() {
    for (InitIterator cur(begin); cur != end; ++cur) {
        Insert(*cur);
    }
}

```

```

}

CartesianTree(std::initializer_list<T> list) : CartesianTree() {
    for (const T& value : list) {
        Insert(value);
    }
}

CartesianTree(const CartesianTree& other) : CartesianTree() {
    for (const T& value : other) {
        Insert(value);
    }
}

CartesianTree(CartesianTree&& other) noexcept : CartesianTree() {
    std::swap(root_, other.root_);
    std::swap(begin_, other.begin_);
    std::swap(end_, other.end_);
    std::swap(size_, other.size_);
}

CartesianTree(std::shared_ptr<ITree<T>> other)
    : CartesianTree(*dynamic_cast<CartesianTree<T>*>(other.get())))
{
}

CartesianTree& operator=(const CartesianTree& other) {
    if (root_ == other.root_) {
        return *this;
    }
    root_ = std::make_shared<Node>();
    begin_ = root_;
    end_ = root_;
    size_ = 0;
    for (const T& value : other) {
        Insert(value);
    }
    return *this;
}

CartesianTree& operator=(CartesianTree&& other) noexcept {
    if (root_ == other.root_) {
        return *this;
    }
    std::swap(root_, other.root_);
    std::swap(begin_, other.begin_);
    std::swap(end_, other.end_);
    std::swap(size_, other.size_);
    return *this;
}

~CartesianTree() override {
    root_ = begin_ = end_ = nullptr;
}

[[nodiscard]] size_t Size() const override {
    return size_;
}

```

```

[[nodiscard]] bool Empty() const override {
    return !size_;
}

std::shared_ptr<BaseImpl> Find(const T& value) const override {
    std::optional<T> val(value);
    auto from = root_;
    while (from) {
        if (val < from->value_) {
            from = from->left_;
        } else if (from->value_ < val) {
            from = from->right_;
        } else {
            return std::make_shared<CartesianTreeItImpl>(from);
        }
    }
    return End();
}

std::shared_ptr<BaseImpl> LowerBound(const T& value) const
override {
    std::optional<T> val(value);
    auto from = root_;
    while (true) {
        if (val < from->value_) {
            if (from->left_) {
                from = from->left_;
            } else {
                return
std::make_shared<CartesianTreeItImpl>(from);
            }
        } else if (from->value_ < val) {
            if (from->right_) {
                from = from->right_;
            } else {
                auto impl =
std::make_shared<CartesianTreeItImpl>(from);
                impl->Increment();
                return impl;
            }
        } else {
            return std::make_shared<CartesianTreeItImpl>(from);
        }
    }
}

void Insert(const T& value) override {
    std::shared_ptr<Node> new_node =
std::make_shared<Node>(value);
    if (InsertRecursive(root_, new_node)) {
        ++size_;
    }
    RecalcBegin();
}

```

```

void Erase(const T& value) override {
    std::optional<T> val(value);
    if (EraseRecursive(root_, val)) {
        --size_;
    }
    RecalcBegin();
}

void Clear() override {
    root_ = std::make_shared<Node>();
    begin_ = root_;
    end_ = root_;
    size_ = 0;
}

private:
    std::shared_ptr<Node> begin_;
    std::shared_ptr<Node> end_;
    std::shared_ptr<Node> root_;
    size_t size_{};

/* -----
 * -----ITERATOR IMPLEMENTATION-----
 * -----
 */

```

class CartesianTreeItImpl : public BaseImpl {

public:

- CartesianTreeItImpl() = delete;
- explicit CartesianTreeItImpl(std::shared_ptr<Node> pointer) : it_(pointer) {}
- CartesianTreeItImpl(const CartesianTreeItImpl& other) : it_(other.it_) {}

std::shared_ptr<BaseImpl> Clone() const override {
 return std::make_shared<CartesianTreeItImpl>(*this);
}

void Increment() override {
 if (!it_->value_) {
 throw std::runtime_error("Index out of range while
increasing");
 }
 if (it_->right_) {
 it_ = it_->right_;
 while (it_->left_) {
 it_ = it_->left_;
 }
 } else {
 auto parent = it_->parent_.lock();
 while (parent->right_ == it_) {
 it_ = parent;
 }

```

                parent = it_->parent_.lock();
            }
            it_ = parent;
        }
    }
void Decrement() override {
    if (it_->left_) {
        it_ = it_->left_;
        while (it_->right_) {
            it_ = it_->right_;
        }
    } else {
        auto parent = it_->parent_.lock();
        while (parent && parent->left_ == it_) {
            it_ = parent;
            parent = it_->parent_.lock();
        }
        if (parent) {
            it_ = parent;
        } else {
            throw std::runtime_error("Index out of range while
decreasing");
        }
    }
}
const T Dereferencing() const override {
    if (!it_->value_) {
        throw std::runtime_error("Index out of range on
operator*");
    }
    return *(it_->value_);
}
const T* Arrow() const override {
    if (!it_->value_) {
        throw std::runtime_error("Index out of range on
operator->");
    }
    return &(*it_->value_);
}
bool IsEqual(std::shared_ptr<BaseImpl> other) const override {
    auto casted =
std::dynamic_pointer_cast<CartesianTreeItImpl>(other);
    if (!casted) {
        return false;
    }
    return it_ == casted->it_;
}

private:
    std::shared_ptr<Node> it_;
};

std::shared_ptr<BaseImpl> Begin() const override {

```

```

        return std::make_shared<CartesianTreeItImpl>(begin_);
    }
std::shared_ptr<BaseImpl> End() const override {
    return std::make_shared<CartesianTreeItImpl>(end_);
}

/*
 * -----PRIVATE FUNCTIONS-----
 */
static std::shared_ptr<Node> Merge(std::shared_ptr<Node> lhs,
std::shared_ptr<Node> rhs) {
    if (!lhs) {
        return rhs;
    } else if (!rhs) {
        return lhs;
    } else if (lhs->priority_ < rhs->priority_) {
        lhs->right_ = Merge(lhs->right_, rhs);
        if (lhs->right_) {
            lhs->right_->parent_ = lhs;
        }
        return lhs;
    } else {
        rhs->left_ = Merge(lhs, rhs->left_);
        if (rhs->left_) {
            rhs->left_->parent_ = rhs;
        }
        return rhs;
    }
}

static void Split(std::shared_ptr<Node> root, const
std::optional<T>& value,
                  std::shared_ptr<Node>& left_sub,
std::shared_ptr<Node>& right_sub) {
    std::shared_ptr<Node> new_subtree = nullptr;
    if (value < root->value_) {
        if (!root->left_) {
            left_sub = nullptr;
        } else {
            Split(root->left_, value, left_sub, new_subtree);
            root->left_ = new_subtree;
            if (new_subtree) {
                new_subtree->parent_ = root;
            }
            if (left_sub) {
                left_sub->parent_ = std::weak_ptr<Node>();
            }
        }
        right_sub = root;
    } else {
        if (!root->right_) {

```

```

        right_sub = nullptr;
    } else {
        Split(root->right_, value, new_subtree, right_sub);
        root->right_ = new_subtree;
        if (new_subtree) {
            new_subtree->parent_ = root;
        }
        if (right_sub) {
            right_sub->parent_ = std::weak_ptr<Node>();
        }
    }
    left_sub = root;
}
}

static bool InsertRecursive(std::shared_ptr<Node>& from,
std::shared_ptr<Node> new_node) {
    if (!from) {
        from = new_node;
        return true;
    } else if (from->priority_ >= new_node->priority_) {
        std::shared_ptr<Node> left_sub = nullptr, right_sub =
nullptr;
        Split(from, new_node->value_, left_sub, right_sub);
        if (left_sub) {
            std::shared_ptr<Node> max_v = left_sub;
            while (max_v->right_) {
                max_v = max_v->right_;
            }
            if (max_v->value_ < new_node->value_) {
                from = Merge(Merge(left_sub, new_node),
right_sub);
                return true;
            } else if (new_node->value_ < max_v->value_) {
                throw std::runtime_error("Error in function
Split()");
            } else {
                from = Merge(left_sub, right_sub);
                return false;
            }
        } else {
            new_node->right_ = right_sub;
            if (right_sub) {
                right_sub->parent_ = new_node;
            }
            from = new_node;
            return true;
        }
    } else if ((from->value_ < new_node->value_) || (new_node-
>value_ < from->value_)) {
        bool result;
        if (new_node->value_ < from->value_) {
            result = InsertRecursive(from->left_, new_node);
        }
    }
}

```

```

        if (from->left_) {
            from->left_->parent_ = from;
        }
    } else {
        result = InsertRecursive(from->right_, new_node);
        if (from->right_) {
            from->right_->parent_ = from;
        }
    }
    return result;
}
return false;
}

static bool EraseRecursive(std::shared_ptr<Node>& from, const
std::optional<T>& value) {
    bool result;
    if (!from) {
        return false;
    } else if (value < from->value_) {
        result = EraseRecursive(from->left_, value);
        if (from->left_) {
            from->left_->parent_ = from;
        }
    } else if (from->value_ < value) {
        result = EraseRecursive(from->right_, value);
        if (from->right_) {
            from->right_->parent_ = from;
        }
    } else {
        from = Merge(from->left_, from->right_);
        return true;
    }
    return result;
}

void RecalcBegin() {
    std::shared_ptr<Node> cur_node = root_;
    while (cur_node->left_) {
        cur_node = cur_node->left_;
    }
    begin_ = cur_node;
}
};

Trees/rb_tree.h

```

```

#pragma once
#include <algorithm>
#include <exception>
#include <initializer_list>
#include <memory>
#include <optional>

```

```

template <class T>
class ITree;

template <class T>
class RBTree : public ITree<T> {
private:
    typedef typename ITree<T>::ITreeItImpl BaseImpl;

public:
    struct Node {
        Node() {
            left_ = nullptr;
            right_ = nullptr;
            parent_ = std::weak_ptr<Node>();
            value_ = std::nullopt;
            is_red_ = true;
        }

        explicit Node(const T& value) : value_(value) {
            left_ = nullptr;
            right_ = nullptr;
            parent_ = std::weak_ptr<Node>();
            is_red_ = true;
        }

        Node(const Node& other) : value_(other.value_) {
            left_ = other.left_;
            right_ = other.right_;
            parent_ = other.parent_;
            is_red_ = other.is_red_;
        }

        std::shared_ptr<Node> left_;
        std::shared_ptr<Node> right_;
        std::weak_ptr<Node> parent_;
        std::optional<T> value_;
        bool is_red_;
    };
};

RBTree() {
    end_ = std::make_shared<Node>();
    begin_ = end_;
    root_ = end_;
    size_ = 0;
}

template <class InitIterator>
RBTree(InitIterator begin, InitIterator end) : RBTree() {
    for (InitIterator cur(begin); cur != end; ++cur) {
        Insert(*cur);
    }
}

RBTree(std::initializer_list<T> list) : RBTree() {
}

```

```

        for (const T& value : list) {
            Insert(value);
        }
    }

RBTree(const RBTree& other) : RBTree() {
    for (const T& value : other) {
        Insert(value);
    }
}

RBTree(RBTree&& other) noexcept : RBTree() {
    std::swap(root_, other.root_);
    std::swap(begin_, other.begin_);
    std::swap(end_, other.end_);
    std::swap(size_, other.size_);
}

RBTree(std::shared_ptr<ITree<T>> other) :
RBTree(*dynamic_cast<RBTree<T>*>(other.get())))
{
}

RBTree& operator=(const RBTree& other) {
    if (root_ == other.root_) {
        return *this;
    }
    end_ = std::make_shared<Node>();
    root_ = end_;
    begin_ = end_;
    size_ = 0;
    for (const T& value : other) {
        Insert(value);
    }
    return *this;
}

RBTree& operator=(RBTree&& other) noexcept {
    if (root_ == other.root_) {
        return *this;
    }
    std::swap(root_, other.root_);
    std::swap(begin_, other.begin_);
    std::swap(end_, other.end_);
    std::swap(size_, other.size_);
    return *this;
}

~RBTree() override {
    root_ = nullptr;
    begin_ = nullptr;
    end_ = nullptr;
    size_ = 0;
}

[[nodiscard]] size_t Size() const override {
    return size_;
}

```

```

[[nodiscard]] bool Empty() const override {
    return !static_cast<bool>(size_);
}

std::shared_ptr<BaseImpl> Find(const T& value) const override {
    std::optional<T> val(value);
    return FindImpl(root_, val);
}

std::shared_ptr<BaseImpl> LowerBound(const T& value) const
override {
    std::optional<T> val(value);
    return LowerBoundImpl(root_, val);
}

void Insert(const T& value) override {
    std::shared_ptr<Node> new_node =
std::make_shared<Node>(value);
    if (InsertImplementation(new_node)) {
        ++size_;
    }
}
void Erase(const T& value) override {
    auto nodeInTree =
std::static_pointer_cast<RBTreeItImpl>(Find(value));
    if (nodeInTree->IsEqual(End())) {
        return;
    }

    EraseImplementation(nodeInTree->GetPointer());
    --size_;
}
void Clear() override {
    root_ = std::shared_ptr<Node>();
    begin_ = root_;
    end_ = root_;
    size_ = 0;
}

void CheckRB() const {
    std::vector<int> blackHeight;
    CheckRBRecursive(root_, blackHeight, 0);

    blackHeight.erase(std::unique(blackHeight.begin(),
blackHeight.end()), blackHeight.end());
    if (blackHeight.size() != 1) {
        throw std::runtime_error("Black height is different");
    }
}

private:
    std::shared_ptr<Node> begin_;

```

```

std::shared_ptr<Node> end_;
std::shared_ptr<Node> root_;
size_t size_;

/* -----
 * -----ITERATOR IMPLEMENTATION-----
 * -----
 */

class RBTreeItImpl : public BaseImpl {
public:
    RBTreeItImpl() = delete;
    explicit RBTreeItImpl(std::shared_ptr<Node> pointer) : it_(pointer) {
    }
    RBTreeItImpl(const RBTreeItImpl& other) : it_(other.it_) {
    }

    std::shared_ptr<BaseImpl> Clone() const override {
        return std::make_shared<RBTreeItImpl>(*this);
    }

    void Increment() override {
        if (!it_->value_) {
            throw std::runtime_error("Index out of range while
increasing");
        }
        if (it_->right_) {
            it_ = it_->right_;
            while (it_->left_) {
                it_ = it_->left_;
            }
        } else {
            auto parent = it_->parent_.lock();
            while (parent && parent->right_ == it_) {
                it_ = parent;
                parent = it_->parent_.lock();
            }
            it_ = parent;
        }
    }
    void Decrement() override {
        if (it_->left_) {
            it_ = it_->left_;
            while (it_->right_) {
                it_ = it_->right_;
            }
        } else {
            auto parent = it_->parent_.lock();
            while (parent && parent->left_ == it_) {
                it_ = parent;
                parent = it_->parent_.lock();
            }
        }
    }
}

```

```

        if (parent) {
            it_ = parent;
        } else {
            throw std::runtime_error("Index out of range while
decreasing");
        }
    }

const T Dereferencing() const override {
    if (it_ && !it_->value_) {
        throw std::runtime_error("Index out of range on
operator*");
    }
    return it_->value_.value();
}

const T* Arrow() const override {
    if (it_ && !it_->value_) {
        throw std::runtime_error("Index out of range on
operator->");
    }
    return &it_->value_.value();
}

bool IsEqual(std::shared_ptr<BaseImpl> other) const override {
    auto casted =
std::dynamic_pointer_cast<RBTreeItImpl>(other);
    if (!casted) {
        return false;
    }
    return it_ == casted->it_;
}

std::shared_ptr<Node> GetPointer() const {
    return it_;
}

private:
    std::shared_ptr<Node> it_;
};

std::shared_ptr<BaseImpl> Begin() const override {
    return std::make_shared<RBTreeItImpl>(begin_);
}
std::shared_ptr<BaseImpl> End() const override {
    return std::make_shared<RBTreeItImpl>(end_);
}

/*
* -----
* -----PRIVATE FUNCTIONS-----
* -----
*/

```

```

    std::shared_ptr<BaseImpl> FindImpl(std::shared_ptr<Node> from,
                                         const std::optional<T>&
value) const {

    while (from) {
        if (value < from->value_) {
            from = from->left_;
        } else if (from->value_ < value) {
            from = from->right_;
        } else {
            return std::make_shared<RBTreeItImpl>(from);
        }
    }
    return End();
};

static std::shared_ptr<BaseImpl>
LowerBoundImpl(std::shared_ptr<Node> from,
                           const
std::optional<T>& value) {
    while (true) {
        if (value < from->value_) {
            if (from->left_) {
                from = from->left_;
            } else {
                return std::make_shared<RBTreeItImpl>(from);
            }
        } else if (from->value_ < value) {
            if (from->right_) {
                from = from->right_;
            } else {
                auto impl = std::make_shared<RBTreeItImpl>(from);
                impl->Increment();
                return impl;
            }
        } else {
            return std::make_shared<RBTreeItImpl>(from);
        }
    }
}

void CheckRBRecursive(std::shared_ptr<Node> from,
std::vector<int>& blackHeight, int bh) const {
    auto parent = from->parent_.lock();
    if (parent) {
        if (parent->is_red_ && from->is_red_) {
            throw std::runtime_error("Two red nodes in a row");
        }
    }
    if (!from->is_red_) {
        ++bh;
    }
    if (!from->left_) {
        blackHeight.push_back(bh);
    } else {

```

```

        CheckRBRecursive(from->left_, blackHeight, bh);
    }
    if (!from->right_) {
        blackHeight.push_back(bh);
    } else {
        CheckRBRecursive(from->right_, blackHeight, bh);
    }
}

bool InsertImplementation(const std::shared_ptr<Node>& new_node) {
    if (!root_) {
        root_ = new_node;
        RecalcBegin();
        return true;
    }

    auto cur_node = root_;
    auto next_node = cur_node;
    while (next_node) {
        cur_node = next_node;
        if (new_node->value_ < cur_node->value_) {
            next_node = cur_node->left_;

        } else if (cur_node->value_ < new_node->value_) {
            next_node = cur_node->right_;
        } else {
            RecalcBegin();
            return false;
        }
    }

    if (new_node->value_ < cur_node->value_) {
        cur_node->left_ = new_node;
    } else {
        cur_node->right_ = new_node;
    }
    new_node->parent_ = cur_node;
    RBFixBalanceAfterInsert(new_node);

    RecalcBegin();
    return true;
}
void RBFixBalanceAfterInsert(std::shared_ptr<Node> from) {
    // Node doesn't have a parent
    if (from->parent_.expired()) {
        return;
    }

    auto parent = from->parent_.lock();

    // Everything is okey
    if (parent->is_red_ == false) {
        return;
    }
}

```

```

}

// Node doesn't have a grandparent
if (parent->parent_.expired()) {
    parent->is_red_ = false;
    return;
}

auto grandparent = parent->parent_.lock();

if (grandparent->right_ == parent) {
    auto uncle = grandparent->left_;
    // Uncle is red
    if (uncle && uncle->is_red_) {
        parent->is_red_ = false;
        uncle->is_red_ = false;
        grandparent->is_red_ = true;
        RBFixBalanceAfterInsert(grandparent);
        return;
    }
    // Node is left
    if (parent->left_ == from) {
        if (grandparent->right_ == parent) {
            grandparent->right_ = from;
        } else {
            grandparent->left_ = from;
        }
        parent->parent_ = from;
        parent->left_ = from->right_;
        if (from->right_) {
            from->right_->parent_ = parent;
        }
        from->right_ = parent;
        from->parent_ = grandparent;
        RBFixBalanceAfterInsert(parent);
        return;
    } else {
        // Big rotation
        if (!grandparent->parent_.expired()) {
            auto grandgrandparent = grandparent-
>parent_.lock();
            if (grandgrandparent->right_ == grandparent) {
                grandgrandparent->right_ = parent;
            } else {
                grandgrandparent->left_ = parent;
            }
            parent->parent_ = grandgrandparent;
        } else {
            root_ = parent;
            parent->parent_ = std::weak_ptr<Node>();
        }
        grandparent->right_ = parent->left_;
        if (parent->left_) {

```

```

        parent->left_->parent_ = grandparent;
    }
    parent->left_ = grandparent;
    grandparent->parent_ = parent;

    parent->is_red_ = false;
    grandparent->is_red_ = true;
    return;
}
// Node is right
} else {
    auto uncle = grandparent->right_;
    // Uncle is red
    if (uncle && uncle->is_red_) {
        parent->is_red_ = false;
        uncle->is_red_ = false;
        grandparent->is_red_ = true;

        RBFixBalanceAfterInsert(grandparent);
        return;
    }
    // Node is right
    if (parent->right_ == from) {
        if (grandparent->right_ == parent) {
            grandparent->right_ = from;
        } else {
            grandparent->left_ = from;
        }
        parent->parent_ = from;
        parent->right_ = from->left_;
        if (from->left_) {
            from->left_->parent_ = parent;
        }
        from->left_ = parent;
        from->parent_ = grandparent;
        RBFixBalanceAfterInsert(parent);
        return;
    } else {
        if (!grandparent->parent_.expired()) {
            auto ggrandparent = grandparent->parent_.lock();
            if (ggrandparent->left_ == grandparent) {
                ggrandparent->left_ = parent;
            } else {
                ggrandparent->right_ = parent;
            }
            parent->parent_ = ggrandparent;
        } else {
            root_ = parent;
            parent->parent_ = std::weak_ptr<Node>();
        }
        grandparent->left_ = parent->right_;
        if (parent->right_) {
            parent->right_->parent_ = grandparent;
        }
    }
}

```

```

        }
        parent->right_ = grandparent;
        grandparent->parent_ = parent;

        parent->is_red_ = false;
        grandparent->is_red_ = true;
        return;
    }
}

void EraseImplementation(std::shared_ptr<Node> delete_node) {
    auto parent = delete_node->parent_.lock();

    // Node doesn't have children
    if (!delete_node->right_ && !delete_node->left_) {
        RBFixBalanceAfterErase(delete_node);
        if (parent) {
            if (parent->left_ == delete_node) {
                parent->left_ = nullptr;
            } else {
                parent->right_ = nullptr;
            }
        } else {
            root_ = nullptr;
        }
    } else if ((delete_node->right_ && !delete_node->left_) ||
               (!delete_node->right_ && delete_node->left_)) {

        auto child_node = delete_node->right_ ? delete_node-
>right_ : delete_node->left_;
        if (!parent) {
            root_ = child_node;
            child_node->parent_ = std::weak_ptr<Node>();
        } else {
            if (parent->left_ == delete_node) {
                parent->left_ = child_node;
            } else {
                parent->right_ = child_node;
            }
            child_node->parent_ = parent;
        }

        if (!delete_node->is_red_ && child_node->is_red_) {
            child_node->is_red_ = false;
        } else if (!delete_node->is_red_ && !child_node-
>is_red_) {

            RBFixBalanceAfterErase(child_node);
        }
    } else {
        auto swap_node = delete_node->right_;

```

```

        while (swap_node->left_) {
            swap_node = swap_node->left_;
        }

        if (delete_node->right_ == swap_node) {
            SwapWithChild(delete_node, swap_node);
        } else {
            SwapWithOffspring(delete_node, swap_node);
        }
        EraseImplementation(delete_node);
        return;
    }

    RecalcBegin();
}

void RBFixBalanceAfterErase(std::shared_ptr<Node> from) {
    while (!from->is_red_ && root_ != from) {
        auto parent = from->parent_.lock();
        if (!parent) {
            return;
        }
        auto sibling = (parent->left_ == from) ? parent->right_ :
parent->left_;
        if (!sibling) {
            return;
        }
        auto sibling_right = sibling->right_;
        auto sibling_left = sibling->left_;
        if (parent->left_ == from) {
            // Brother is red;
            if (sibling->is_red_) {
                sibling->is_red_ = false;
                parent->is_red_ = true;
                LeftRotate(parent);
                sibling = (parent->left_ == from) ? parent->right_ :
parent->left_;
                sibling_right = sibling->right_;
                sibling_left = sibling->left_;
            }
            // Brother and children are black
            if ((!sibling_left || !sibling_left->is_red_) &&
                (!sibling_right || !sibling_right->is_red_)) {
                sibling->is_red_ = true;
                from = parent;
            } else {
                if (!sibling_right || !sibling_right->is_red_) {
                    sibling->is_red_ = true;
                    sibling_left->is_red_ = false;
                    RightRotate(sibling);
                    sibling = (parent->left_ == from) ? parent->right_ :
parent->left_;
                    if (sibling->right_) {

```

```

                sibling_right = sibling->right_;
            } else {
                sibling_right = nullptr;
            }
            if (sibling->left_) {
                sibling_left = sibling->left_;
            } else {
                sibling_left = nullptr;
            }
        }
        sibling->is_red_ = parent->is_red_;
        parent->is_red_ = false;
        if (sibling_right) {
            sibling->right_->is_red_ = false;
        }
        LeftRotate(parent);
        from = root_;
    }
} else {
    // Brother is red;
    if (sibling->is_red_) {
        sibling->is_red_ = false;
        parent->is_red_ = true;
        RightRotate(parent);
        sibling = (parent->left_ == from) ? parent->right_
: parent->left_;
        sibling_right = sibling->right_;
        sibling_left = sibling->left_;
    }
    // Brother is black
    if ((!sibling_left || !sibling_left->is_red_) &&
        (!sibling_right || !sibling_right->is_red_)) {
        sibling->is_red_ = true;
        from = parent;
    } else {
        if (!sibling_left || !sibling_left->is_red_) {
            sibling->is_red_ = true;
            sibling_right->is_red_ = false;
            LeftRotate(sibling);
            sibling = (parent->left_ == from) ? parent-
right_ : parent->left_;
        }
        if (sibling) {
            sibling_right = sibling->right_;
        } else {
            sibling_right = nullptr;
        }
        if (sibling) {
            sibling_left = sibling->left_;
        } else {
            sibling_left = nullptr;
        }
    }
}

```

```

        sibling->is_red_ = parent->is_red_;
        parent->is_red_ = false;
        if (sibling_left) {
            sibling_left->is_red_ = false;
        }
        RightRotate(parent);
        from = root_;
    }
}
from->is_red_ = false;
root_->is_red_ = false;
}

void SwapWithChild(std::shared_ptr<Node> from_node,
std::shared_ptr<Node> swap_node) {
    auto parent = from_node->parent_.lock();
    if (parent) {
        if (parent->left_ == from_node) {
            parent->left_ = swap_node;
        } else {
            parent->right_ = swap_node;
        }
    } else {
        root_ = swap_node;
    }
    swap_node->parent_ = parent;

    from_node->right_ = swap_node->right_;
    if (swap_node->right_) {
        swap_node->right_->parent_ = from_node;
    }

    swap_node->right_ = from_node;
    from_node->parent_ = swap_node;

    swap_node->left_ = from_node->left_;
    if (from_node->left_) {
        from_node->left_->parent_ = swap_node;
    }

    from_node->left_ = nullptr;
    std::swap(swap_node->is_red_, from_node->is_red_);
}
void SwapWithOffspring(std::shared_ptr<Node> from_node,
std::shared_ptr<Node> swap_node) {
    auto from_parent = from_node->parent_.lock();
    if (from_parent) {
        if (from_parent->left_ == from_node) {
            from_parent->left_ = swap_node;
        } else {
            from_parent->right_ = swap_node;
        }
    }
}

```

```

        }
    } else {
        root_ = swap_node;
    }

    auto swap_parent = swap_node->parent_.lock();
    if (swap_parent->right_ == swap_node) {
        swap_parent->right_ = from_node;
    } else {
        swap_parent->left_ = from_node;
    }

    auto tmp_parent = from_node->parent_;
    from_node->parent_ = swap_node->parent_;
    swap_node->parent_ = tmp_parent;

    auto tmp_right = swap_node->right_;
    swap_node->right_ = from_node->right_;
    from_node->right_->parent_ = swap_node;
    from_node->right_ = tmp_right;
    if (from_node->right_) {
        from_node->right_->parent_ = from_node;
    }

    swap_node->left_ = from_node->left_;
    if (from_node->left_) {
        from_node->left_->parent_ = swap_node;
    }

    from_node->left_ = nullptr;
    std::swap(swap_node->is_red_, from_node->is_red_);
}

void LeftRotate(std::shared_ptr<Node>& from) {
    auto parent = from->parent_.lock();
    auto right_node = from->right_;
    std::shared_ptr<Node> next_node = nullptr;
    if (right_node) {
        next_node = right_node->left_;
    }

    if (parent) {
        if (parent->left_ == from) {
            parent->left_ = right_node;
        } else {
            parent->right_ = right_node;
        }
    } else {
        root_ = right_node;
    }

    if (right_node) {

```

```

        right_node->parent_ = parent;
        right_node->left_ = from;
    }

    if (next_node) {
        next_node->parent_ = from;
    }
    from->parent_ = right_node;
    from->right_ = next_node;
}
void RightRotate(std::shared_ptr<Node>& from) {
    auto parent = from->parent_.lock();
    auto left_node = from->left_;
    std::shared_ptr<Node> prev_node = nullptr;

    if (left_node) {
        prev_node = left_node->right_;
    }

    if (parent) {
        if (parent->right_ == from) {
            parent->right_ = left_node;
        } else {
            parent->left_ = left_node;
        }
    } else {
        root_ = left_node;
    }

    if (left_node) {
        left_node->parent_ = parent;
        left_node->right_ = from;
    }

    if (prev_node) {
        prev_node->parent_ = from;
    }
    from->parent_ = left_node;
    from->left_ = prev_node;
}
// Set begin_ after modification
void RecalcBegin() {
    auto node = root_;
    while (node->left_) {
        node = node->left_;
    }
    begin_ = node;
}
};

Trees/skip_list.h
#endif

```

Trees/skip_list.h

#pragma once

```

#include <exception>
#include <initializer_list>
#include <iostream>
#include <memory>

template <class T>
class ITree;

template <class T>
class SkipList : public ITree<T> {
private:
    typedef typename ITree<T>::ITreeItImpl BaseImpl;

    class Optional {
private:
    std::shared_ptr<T> value_;
    char info_;

public:
    Optional() = delete;
    Optional(Optional&&) = delete;

    Optional(char newinfo) {
        info_ = newinfo;
    }

    Optional(const T& value) {
        value_ = std::make_shared<T>(const_cast<T&>(value));
        info_ = 'v';
    }

    Optional(const Optional& other) {
        value_ = other.value_;
        info_ = other.info_;
    }

    ~Optional() {
        value_ = nullptr;
    }

    std::shared_ptr<T> GetValue() const {
        return this->value_;
    }

    char GetInfo() const {
        return this->info_;
    }

    bool operator<(const Optional& other) const {
        if (this->info_ == 'v') {
            if (other.info_ == 'v') {
                return *(this->value_) < *(other.value_);
            } else {

```

```

                return other.info_ != 'b';
            }
        } else if (this->info_ == 'b') {
            return other.info_ != 'b';
        } else {
            return false;
        }
    }
};

class Random {
public:
    static uint32_t Next(uint32_t to = 1u) {
        static Random rand = Random();
        std::uniform_int_distribution<uint32_t> dist(0, to);
        return dist(rand.gen_);
    }

private:
    Random() {
        std::random_device device;
        gen_ = std::mt19937(device());
    }

    std::mt19937 gen_;
};

public:
    struct Node {
        Node() = delete;

        explicit Node(char value_info) : value_(value_info) {
            left_ = std::weak_ptr<Node>();
            down_ = nullptr;
            right_ = nullptr;
        }

        explicit Node(const T& value) : value_(value) {
            left_ = std::weak_ptr<Node>();
            down_ = nullptr;
            right_ = nullptr;
        }

        explicit Node(const Optional& value_info) : value_(value_info)
        {
            left_ = std::weak_ptr<Node>();
            down_ = nullptr;
            right_ = nullptr;
        }

        Node(const Node& other) : value_(other.value_) {
            left_ = other.left_;
            down_ = other.down_;
            right_ = other.right_;
        }
    }
};

```

```

    }

~Node() {
    left_ = right_ = down_ = nullptr;
}

std::shared_ptr<Node> down_;
std::weak_ptr<Node> left_;
std::shared_ptr<Node> right_;
Optional value_;
};

SkipList() {
    head_bot = std::make_shared<Node>('b');
    end_bot = std::make_shared<Node>('e');
    head_bot->right_ = end_bot;
    end_bot->left_ = head_bot;
    head_top = head_bot;
    end_top = end_bot;
    size_ = 0;
}

template <class InitIterator>
SkipList(InitIterator begin, InitIterator end) : SkipList() {
    for (InitIterator cur(begin); cur != end; ++cur) {
        Insert(*cur);
    }
}

SkipList(std::initializer_list<T> list) : SkipList() {
    for (const T& value : list) {
        Insert(value);
    }
}

SkipList(const SkipList& other) : SkipList() {
    for (const T& value : other) {
        Insert(value);
    }
}

SkipList(SkipList&& other) noexcept {
    std::swap(head_top, other.head_top);
    std::swap(end_top, other.end_top);
    std::swap(head_bot, other.head_bot);
    std::swap(end_bot, other.end_bot);
    std::swap(size_, other.size_);
}

SkipList(std::shared_ptr<ITree<T>> other) :
SkipList(*dynamic_cast<SkipList<T>*>(other.get())))
{
}

SkipList& operator=(const SkipList& other) {
    if (head_top == other.head_top) {

```

```

        return *this;
    }
    head_bot = std::make_shared<Node>('b');
    end_bot = std::make_shared<Node>('e');
    head_bot->right_ = end_bot;
    end_bot->left_ = head_bot;
    head_top = head_bot;
    end_top = end_bot;
    size_ = 0;
    for (const T& value : other) {
        Insert(value);
    }
    return *this;
}

SkipList& operator=(SkipList&& other) noexcept {
    if (head_top == other.head_top) {
        return *this;
    }
    std::swap(head_top, other.head_top);
    std::swap(end_top, other.end_top);
    std::swap(head_bot, other.head_bot);
    std::swap(end_bot, other.end_bot);
    std::swap(size_, other.size_);
    return *this;
}

~SkipList() override {
    head_top = head_bot = end_top = end_bot = nullptr;
    size_ = 0;
}

[[nodiscard]] size_t Size() const override {
    return size_;
}

[[nodiscard]] bool Empty() const override {
    return !size_;
}

std::shared_ptr<BaseImpl> Find(const T& value) const override {
    Optional val(value);
    return FindImpl(head_top, val);
}

void Erase(const T& value) override {
    Optional val(value);
    if (EraseImpl(head_top, val)) {
        --size_;
    }
}

```

```

        std::shared_ptr<BaseImpl> LowerBound(const T& value) const
override {
    Optional val(value);
    return LowerBoundImpl(head_top, val);
}

void Insert(const T& value) override {
    std::shared_ptr<Node> new_node =
std::make_shared<Node>(value);
    if (InsertImpl(head_top, new_node)) {
        ++size_;
    }
}

void Clear() override {
    head_bot = std::make_shared<Node>('b');
    end_bot = std::make_shared<Node>('e');
    head_bot->right_ = end_bot;
    end_bot->left_ = head_bot;
    head_top = head_bot;
    end_top = end_bot;
    size_ = 0;
}

private:
    std::shared_ptr<Node> head_top;
    std::shared_ptr<Node> end_top;
    std::shared_ptr<Node> head_bot;
    std::shared_ptr<Node> end_bot;
    size_t size_;
/* -----
 * -----ITERATOR IMPLEMENTATION-----
 * -----
 */
}

class SkipListItImpl : public BaseImpl {
private:
    std::shared_ptr<Node> it_;

public:
    SkipListItImpl() = delete;

    explicit SkipListItImpl(std::shared_ptr<Node> ptr) : it_(ptr)
{
}

SkipListItImpl(const SkipListItImpl& other) : it_(other.it_) {

std::shared_ptr<BaseImpl> Clone() const override {
    return std::make_shared<SkipListItImpl>(*this);
}

```

```

    void Increment() override {
        if (!it_->right_) {
            throw std::runtime_error("Index out of range while
increasing");
        }
        it_ = it_->right_;
    }

    void Decrement() override {
        it_ = it_->left_.lock();
        if (!it_->left_.lock()) {
            throw std::runtime_error("Index out of range while
decreasing");
        }
    }

    const T Dereferencing() const override {
        if (it_->value_.GetInfo() != 'v') {
            throw std::runtime_error("Index out of range on
operator*");
        }
        return *(it_->value_.GetValue());
    }

    const T* Arrow() const override {
        if (it_->value_.GetInfo() != 'v') {
            throw std::runtime_error("Index out of range on
operator->");
        }
        return &(*it_->value_.GetValue());
    }

    bool IsEqual(std::shared_ptr<BaseImpl> other) const override {
        auto casted =
std::dynamic_pointer_cast<SkipListItImpl>(other);
        if (!casted) {
            return false;
        }
        return it_ == casted->it_;
    }
};

std::shared_ptr<BaseImpl> Begin() const override {
    return std::make_shared<SkipListItImpl>(head_bot->right_);
}

std::shared_ptr<BaseImpl> End() const override {
    return std::make_shared<SkipListItImpl>(end_bot);
}

std::shared_ptr<BaseImpl> FindImpl(std::shared_ptr<Node> from,
const Optional& value) const {
    while (true) {

```

```

        if (!from->right_) {
            return End();
        }
        if (from->right_->value_ < value) {
            from = from->right_;
        } else if (!from->down_) {
            if (value < from->right_->value_) {
                return End();
            }
            return std::make_shared<SkipListItImpl>(from->right_);
        } else {
            from = from->down_;
        }
    }
}

bool EraseImpl(std::shared_ptr<Node>& from, const Optional& value)
{
    if (!from->right_) {
        return false;
    }
    if (from->right_->value_ < value) {
        return EraseImpl(from->right_, value);
    }
    if (value < from->right_->value_) {
        if (!from->down_) {
            return false;
        } else {
            return EraseImpl(from->down_, value);
        }
    }
    auto cur_node = from->right_;
    auto next_node = cur_node->right_;
    next_node->left_ = from;
    from->right_ = next_node;
    if (from == head_top && from->right_ == end_top) {
        if (head_top->down_) {
            head_top = head_top->down_;
            end_top = end_top->down_;
        }
    }
    while (cur_node->down_) {
        cur_node = cur_node->down_;
        auto prev_from = cur_node->left_.lock();
        next_node = cur_node->right_;
        prev_from->right_ = next_node;
        next_node->left_ = prev_from;
        if (prev_from == head_top && prev_from->right_ == end_top)
    {
        if (head_top->down_) {
            head_top = head_top->down_;
            end_top = end_top->down_;
        }
    }
}

```

```

        }
    }
    return true;
}

static std::shared_ptr<BaseImpl>
LowerBoundImpl(std::shared_ptr<Node> from,
                           const Optional&
value) {
    while (true) {
        if (from->right_->value_ < value) {
            from = from->right_;
        } else if (!from->down_) {
            return std::make_shared<SkipListItImpl>(from->right_);
        } else {
            from = from->down_;
        }
    }
}

bool InsertImpl(std::shared_ptr<Node> from, std::shared_ptr<Node>
new_node) {

    std::vector<std::shared_ptr<Node>> node_path;
    while (true) {
        if (!from->right_) {
            return false;
        } else if (from->right_->value_ < new_node->value_) {
            from = from->right_;
        } else {
            if (from->down_) {
                node_path.push_back(from);
                from = from->down_;
            } else {
                if (new_node->value_ < from->right_->value_) {
                    new_node->left_ = from;
                    new_node->right_ = from->right_;
                    from->right_->left_ = new_node;
                    from->right_ = new_node;
                    BuildLvl(node_path, new_node);
                    return true;
                }
                return false;
            }
        }
    }
}

void BuildLvl(std::vector<std::shared_ptr<Node>> node_path,
std::shared_ptr<Node> from) {
    while (!Random::Next()) {
        std::shared_ptr<Node> up_node;
        up_node = std::make_shared<Node>(from->value_);

```

```

        up_node->down_ = from;
        if (node_path.size() != 0) {
            auto prev = node_path.back();
            up_node->left_ = prev;
            up_node->right_ = prev->right_;
            prev->right_->left_ = up_node;
            prev->right_ = up_node;
            node_path.pop_back();
        } else {
            std::shared_ptr<Node> new_head, new_end, tmp_head,
tmp_end;
            tmp_head = head_top;
            tmp_end = end_top;
            new_head = std::make_shared<Node>('b');
            new_end = std::make_shared<Node>('e');
            new_head->down_ = tmp_head;
            new_head->right_ = up_node;
            up_node->left_ = new_head;
            up_node->right_ = new_end;
            new_end->down_ = tmp_end;
            new_end->left_ = up_node;
            head_top = new_head;
            end_top = new_end;
            break;
        }
        from = up_node;
    }
    return;
}
};

Trees/splay_tree.h

```

```

#pragma once
#include <exception>
#include <initializer_list>
#include <iostream>
#include <memory>
#include <optional>
#include <queue>

template <class T>
class ITree;

template <class T>
class SplayTree : public ITree<T> {
private:
    typedef typename ITree<T>::ITreeItImpl BaseImpl;

public:
    struct Node {
        Node() {
            parent_ = left_ = right_ = nullptr;

```

```

        value_ = std::nullopt;
    }

explicit Node(const T& value) : value_(value) {
    parent_ = left_ = right_ = nullptr;
}

Node(const Node& other) : value_(other.value_) {
    left_ = other.left_;
    parent_ = other.parent_;
    right_ = other.right_;
}

~Node() {
    left_ = nullptr;
    right_ = nullptr;
}

std::shared_ptr<Node> left_;
std::shared_ptr<Node> right_;
std::weak_ptr<Node> parent_;
std::optional<T> value_;
};

SplayTree() {
    begin_ = end_ = root_ = std::make_shared<Node>();
    size_ = 0;
}

template <class InitIterator>
SplayTree(InitIterator begin, InitIterator end) : SplayTree() {
    for (InitIterator cur(begin); cur != end; ++cur) {
        Insert(*cur);
    }
}

SplayTree(std::initializer_list<T> list) : SplayTree() {
    for (const T& value : list) {
        Insert(value);
    }
}

SplayTree(const SplayTree& other) : SplayTree() {
    for (const T& value : other) {
        Insert(value);
    }
}

SplayTree(SplayTree&& other) noexcept : SplayTree() {
    std::swap(root_, other.root_);
    std::swap(begin_, other.begin_);
    std::swap(end_, other.end_);
    std::swap(size_, other.size_);
}

```

```

}

SplayTree(std::shared_ptr

```

```

        return !size_;
    }

std::shared_ptr<BaseImpl> Find(const T& value) const override {
    std::optional<T> val(value);
    std::shared_ptr<Node> cur_node = root_;
    while (cur_node) {
        if (val < cur_node->value_) {
            if (cur_node->left_) {
                cur_node = cur_node->left_;
            } else {
                const_cast<SplayTree<T>*>(this)->Splay(cur_node);
                return End();
            }
        } else if (cur_node->value_ < val) {
            if (cur_node->right_) {
                cur_node = cur_node->right_;
            } else {
                const_cast<SplayTree<T>*>(this)->Splay(cur_node);
                return End();
            }
        } else {
            const_cast<SplayTree<T>*>(this)->Splay(cur_node);
            return std::make_shared<SplayTreeItImpl>(root_);
        }
    }
    return End();
}

std::shared_ptr<BaseImpl> LowerBound(const T& value) const
override {
    std::optional<T> val(value);
    std::shared_ptr<Node> cur = root_;
    while (true) {
        if (val < cur->value_) {
            if (cur->left_) {
                cur = cur->left_;
            } else {
                const_cast<SplayTree<T>*>(this)->Splay(cur);
                return std::make_shared<SplayTreeItImpl>(root_);
            }
        } else if (cur->value_ < val) {
            if (cur->right_) {
                cur = cur->right_;
            } else {
                auto it = std::make_shared<SplayTreeItImpl>(cur);
                it->Increment();
                const_cast<SplayTree<T>*>(this)->Splay(it-
>GetPointer());
                return it;
            }
        } else {
            const_cast<SplayTree<T>*>(this)->Splay(cur);
        }
    }
}

```

```

        return std::make_shared<SplayTreeItImpl>(root_);
    }
}

void Insert(const T& value) override {
    std::shared_ptr<Node> new_node =
std::make_shared<Node>(value);
    std::shared_ptr<Node> cur = root_;
    while (true) {
        if (new_node->value_ < cur->value_) {
            if (cur->left_) {
                cur = cur->left_;
            } else {
                new_node->parent_ = cur;
                cur->left_ = new_node;
                if (begin_ == cur) {
                    begin_ = new_node;
                }
                cur = cur->left_;
                break;
            }
        } else if (cur->value_ < new_node->value_) {
            if (cur->right_) {
                cur = cur->right_;
            } else {
                new_node->parent_ = cur;
                cur->right_ = new_node;
                cur = cur->right_;
                break;
            }
        } else {
            Splay(cur);
            return;
        }
    }
    Splay(cur);
    ++size_;
}

void Erase(const T& value) override {
    std::optional<T> val(value);
    auto cur_node = root_;
    while (true) {
        if (val < cur_node->value_) {
            if (cur_node->left_) {
                cur_node = cur_node->left_;
            } else {
                Splay(cur_node);
                return;
            }
        } else if (cur_node->value_ < val) {
            if (cur_node->right_) {

```

```

        cur_node = cur_node->right_;
    } else {
        Splay(cur_node);
        return;
    }
} else {
    break;
}
}
Splay(cur_node);
if (root_ == begin_) {
    auto it =
std::dynamic_pointer_cast<SplayTreeItImpl>(Begin());
    it->Increment();
    begin_ = it->GetPointer();
}
--size_;
std::shared_ptr<Node> left_sub = root_->left_, right_sub =
root_->right_;
root_ = right_sub;
cur_node = root_;
while (left_sub) {
    right_sub = right_sub->left_;
    cur_node->left_ = left_sub;
    left_sub->parent_ = cur_node;
    cur_node = cur_node->left_;
    if (right_sub) {
        left_sub = left_sub->right_;
        cur_node->right_ = right_sub;
        right_sub->parent_ = cur_node;
        cur_node = cur_node->right_;
    } else {
        break;
    }
}
}

void Clear() override {
    size_ = 0;
    std::queue<std::shared_ptr<Node>> nodes;
    nodes.emplace(root_);
    begin_ = end_ = root_ = nullptr;
    while (!nodes.empty()) {
        std::shared_ptr<Node> cur = nodes.front();
        nodes.pop();
        if (cur->left_) {
            nodes.emplace(cur->left_);
        }
        if (cur->right_) {
            nodes.emplace(cur->right_);
        }
        cur->left_ = nullptr;
        cur->right_ = nullptr;
    }
}
}

```

```

    }
    begin_ = end_ = root_ = std::make_shared<Node>();
}

private:
    std::shared_ptr<Node> begin_;
    std::shared_ptr<Node> end_;
    std::shared_ptr<Node> root_;
    size_t size_;

/* -----
 * -----ITERATOR IMPLEMENTATION-----
 * -----
 */

class SplayTreeItImpl : public BaseImpl {
private:
    std::shared_ptr<Node> it_;

public:
    SplayTreeItImpl() = delete;

    explicit SplayTreeItImpl(std::shared_ptr<Node> other) : it_(other) {
    }

    SplayTreeItImpl(const SplayTreeItImpl& other) : it_(other.it_)

    std::shared_ptr<Node> GetPointer() {
        return it_;
    }

    std::shared_ptr<BaseImpl> Clone() const override {
        return std::make_shared<SplayTreeItImpl>(*this);
    }

    void Increment() override {
        if (!it_->value_) {
            throw std::runtime_error("Index out of range while
increasing");
        }
        if (it_->right_) {
            it_ = it_->right_;
            while (it_->left_) {
                it_ = it_->left_;
            }
        } else {
            while (it_->parent_.lock()->right_ == it_) {
                it_ = it_->parent_.lock();
            }
            it_ = it_->parent_.lock();
        }
    }
}

```

```

        }

    }

void Decrement() override {
    if (it_->left_) {
        it_ = it_->left_;
        while (it_->right_) {
            it_ = it_->right_;
        }
    } else {
        auto parent = it_->parent_.lock();
        while (parent && parent->left_ == it_) {
            it_ = parent;
            parent = it_->parent_.lock();
        }
        if (parent) {
            it_ = parent;
        } else {
            throw std::runtime_error("Index out of range while
decreasing");
        }
    }
}

const T Dereferencing() const override {
    if (!it_->value_) {
        throw std::runtime_error("Index out of range on
operator*");
    }
    return *(it_->value_);
}

const T* Arrow() const override {
    if (!it_->value_) {
        throw std::runtime_error("Index out of range on
operator->");
    }
    return &(*it_->value_);
}

bool IsEqual(std::shared_ptr<BaseImpl> other) const override {
    auto casted =
std::dynamic_pointer_cast<SplayTreeItImpl>(other);
    if (!casted) {
        return false;
    }
    return it_ == casted->it_;
}
};

std::shared_ptr<BaseImpl> Begin() const override {
    return std::make_shared<SplayTreeItImpl>(begin_);
}

```

```

        std::shared_ptr<BaseImpl> End() const override {
            return std::make_shared<SplayTreeItImpl>(end_);
        }

        void Splay(std::shared_ptr<Node> from) {
            std::shared_ptr<Node> parent = from->parent_.lock();
            while (parent) {
                std::shared_ptr<Node> grandparent = parent-
>parent_.lock();
                if (!grandparent) {
                    if (parent->right_ == from) {
                        Zag(from, parent);
                    } else {
                        Zig(from, parent);
                    }
                    break;
                }
                if (grandparent->right_ == parent) {
                    if (parent->right_ == from) {
                        ZagZag(from, parent, grandparent);
                    } else {
                        ZagZig(from, parent, grandparent);
                    }
                } else {
                    if (parent->right_ == from) {
                        ZigZag(from, parent, grandparent);
                    } else {
                        ZigZig(from, parent, grandparent);
                    }
                }
                parent = from->parent_.lock();
            }
            root_ = from;
        }

        static void Zig(std::shared_ptr<Node> x, std::shared_ptr<Node> y)
{
            std::shared_ptr<Node> hanger = y->parent_.lock();
            bool left_child = hanger && hanger->left_ == y;
            y->left_ = x->right_;
            if (y->left_) {
                y->left_->parent_ = y;
            }
            x->right_ = y;
            y->parent_ = x;
            SetHanger(x, hanger, left_child);
}

        static void Zag(std::shared_ptr<Node> x, std::shared_ptr<Node> y)
{
            std::shared_ptr<Node> hanger = y->parent_.lock();
            bool left_child = hanger && hanger->left_ == y;

```

```

y->right_ = x->left_;
if (y->right_) {
    y->right_->parent_ = y;
}
x->left_ = y;
y->parent_ = x;
SetHanger(x, hanger, left_child);
}

static void ZigZig(std::shared_ptr<Node> x, std::shared_ptr<Node>
y, std::shared_ptr<Node> z) {
    std::shared_ptr<Node> hanger = z->parent_.lock();
    bool left_child = hanger && hanger->left_ == z;
    z->left_ = y->right_;
    if (z->left_) {
        z->left_->parent_ = z;
    }
    z->parent_ = y;
    y->right_ = z;
    y->left_ = x->right_;
    if (y->left_) {
        y->left_->parent_ = y;
    }
    y->parent_ = x;
    x->right_ = y;
    SetHanger(x, hanger, left_child);
}

static void ZagZag(std::shared_ptr<Node> x, std::shared_ptr<Node>
y, std::shared_ptr<Node> z) {
    std::shared_ptr<Node> hanger = z->parent_.lock();
    bool left_child = hanger && hanger->left_ == z;
    z->right_ = y->left_;
    if (z->right_) {
        z->right_->parent_ = z;
    }
    z->parent_ = y;
    y->left_ = z;
    y->right_ = x->left_;
    if (y->right_) {
        y->right_->parent_ = y;
    }
    y->parent_ = x;
    x->left_ = y;
    SetHanger(x, hanger, left_child);
}

static void ZigZag(std::shared_ptr<Node> x, std::shared_ptr<Node>
y, std::shared_ptr<Node> z) {
    std::shared_ptr<Node> hanger = z->parent_.lock();
    bool left_child = hanger && hanger->left_ == z;
    z->left_ = x->right_;
    if (z->left_) {

```

```

        z->left_->parent_ = z;
    }
    z->parent_ = x;
    x->right_ = z;
    y->right_ = x->left_;
    if (y->right_) {
        y->right_->parent_ = y;
    }
    y->parent_ = x;
    x->left_ = y;
    SetHanger(x, hanger, left_child);
}

static void ZagZig(std::shared_ptr<Node> x, std::shared_ptr<Node>
y, std::shared_ptr<Node> z) {
    std::shared_ptr<Node> hanger = z->parent_.lock();
    bool left_child = hanger && hanger->left_ == z;
    z->right_ = x->left_;
    if (z->right_) {
        z->right_->parent_ = z;
    }
    z->parent_ = x;
    x->left_ = z;
    y->left_ = x->right_;
    if (y->left_) {
        y->left_->parent_ = y;
    }
    y->parent_ = x;
    x->right_ = y;
    SetHanger(x, hanger, left_child);
}

static void SetHanger(std::shared_ptr<Node> x,
std::shared_ptr<Node> hanger, bool left_child) {
    x->parent_ = hanger;
    if (hanger) {
        if (left_child) {
            hanger->left_ = x;
        } else {
            hanger->right_ = x;
        }
    }
}
};

Trees/stdlib_set.h

```

```

#pragma once
#include <exception>
#include <initializer_list>
#include <iostream>
#include <memory>
#include <optional>

```

```

#include <queue>
#include <set>

template <class T>
class ITree;

template <class T>
class StdlibSet : public ITree<T> {
private:
    typedef typename ITree<T>::ITreeItImpl BaseImpl;

public:
    StdlibSet() : set_() {
    }

    template <class InitIterator>
    StdlibSet(InitIterator begin, InitIterator end) : set_(begin, end)
    {
    }

    StdlibSet(std::initializer_list<T> list) : set_(list) {
    }

    StdlibSet(const StdlibSet& other) : set_(other.set_) {
    }

    StdlibSet(StdlibSet&& other) noexcept : set_(other.set_) {
    }

    StdlibSet(std::shared_ptr<ITree<T>> other)
        : StdlibSet(*dynamic_cast<StdlibSet<T>*>(other.get())))
    {
    }

    StdlibSet& operator=(const StdlibSet& other) {
        set_ = other.set_;
        return *this;
    }

    StdlibSet& operator=(StdlibSet&& other) noexcept {
        set_ = std::move(other.set_);
        return *this;
    }

    [[nodiscard]] size_t Size() const override {
        return set_.size();
    }

    [[nodiscard]] bool Empty() const override {
        return set_.empty();
    }

    std::shared_ptr<BaseImpl> Find(const T& value) const override {
        return std::make_shared<StdlibSetItImpl>(set_.find(value));
    }
}

```

```

}

    std::shared_ptr<BaseImpl> LowerBound(const T& value) const
override {
    return
std::make_shared<StdlibSetItImpl>(set_.lower_bound(value));
}

void Insert(const T& value) override {
    set_.insert(value);
}

void Erase(const T& value) override {
    set_.erase(value);
}

void Clear() override {
    set_.clear();
}

private:
    std::set<T> set_;

/*
* -----
* -----ITERATOR IMPLEMENTATION-----
* -----
*/
}

class StdlibSetItImpl : public BaseImpl {
private:
    typename std::set<T>::iterator it_;

public:
    StdlibSetItImpl() = delete;

    StdlibSetItImpl(const StdlibSetItImpl& other) : it_(other.it_)

    StdlibSetItImpl(typename std::set<T>::iterator other) :
it_(other) {
}

    std::shared_ptr<BaseImpl> Clone() const override {
        return std::make_shared<StdlibSetItImpl>(*this);
    }

    void Increment() override {
        ++it_;
    }

    void Decrement() override {
        --it_;
    }
}

```

```

    }

    const T Dereferencing() const override {
        return *it_;
    }

    const T* Arrow() const override {
        return &(*it_);
    }

    bool IsEqual(std::shared_ptr<BaseImpl> other) const override {
        auto casted =
std::dynamic_pointer_cast<StdlibSetItImpl>(other);
        if (!casted) {
            return false;
        }
        return it_ == casted->it_;
    }
};

std::shared_ptr<BaseImpl> Begin() const override {
    return std::make_shared<StdlibSetItImpl>(set_.begin());
}

std::shared_ptr<BaseImpl> End() const override {
    return std::make_shared<StdlibSetItImpl>(set_.end());
}
};

```

Tests/full_test_set.h

```

#pragma once
#include <exception>
#include <iostream>
#include <random>
#include <set>
#include <string>
#include <utility>
#include <vector>

#include "../trees/abstract_tree.h"
#include "../trees/avl_tree.h"
#include "../trees/cartesian_tree.h"
#include "../trees/rb_tree.h"
#include "../trees/skip_list.h"
#include "../trees/splay_tree.h"
#include "../trees/stdlib_set.h"

#define RELEASE_BUILD

/// All types of trees
enum class ImplType { kAVL, kCartesian, kRB, kSkipList, kSplay, kSet
};
```

```

/**
 * Makes a tree of given type and returns a shared pointer on it
 * @tparam T Tree value type
 * @tparam Types Types of constructor parameters
 * @param type Type of tree to make
 * @param params Parameters for tree constructor
 * @return Shared pointer on a new tree
 */
template <class T, class... Types>
std::shared_ptr<ITree<T>> MakeTree(ImplType type, Types... params) {
    if (type == ImplType::kAVL) {
        return std::make_shared<AVLTree<T>>(params...);
    } else if (type == ImplType::kCartesian) {
        return std::make_shared<CartesianTree<T>>(params...);
    } else if (type == ImplType::kB) {
        return std::make_shared<RBTree<T>>(params...);
    } else if (type == ImplType::kSkipList) {
        return std::make_shared<SkipList<T>>(params...);
    } else if (type == ImplType::kSplay) {
        return std::make_shared<SplayTree<T>>(params...);
    } else if (type == ImplType::kSet) {
        return std::make_shared<StdlibSet<T>>(params...);
    } else {
        throw std::runtime_error("Impossible behaviour");
    }
}

/**
 * Replacement for ITree operator=(const ITree& other)
 *
 * Makes a copy of given in @param rhs tree and assigns it to @param lhs
 * @tparam T Tree value type
 * @param type Type of tree to copy
 * @param lhs Used to return a copy
 * @param rhs Given tree
 */
template <class T>
void MakeCopyAssignment(ImplType type, std::shared_ptr<ITree<T>>& lhs,
                      std::shared_ptr<ITree<T>> rhs) {
    if (type == ImplType::kAVL) {
        *dynamic_cast<AVLTree<T>*>(lhs.get()) =
*dynamic_cast<AVLTree<T>*>(rhs.get());
    } else if (type == ImplType::kCartesian) {
        *dynamic_cast<CartesianTree<T>*>(lhs.get()) =
*dynamic_cast<CartesianTree<T>*>(rhs.get());
    } else if (type == ImplType::kB) {
        *dynamic_cast<RBTree<T>*>(lhs.get()) =
*dynamic_cast<RBTree<T>*>(rhs.get());
    } else if (type == ImplType::kSkipList) {
        *dynamic_cast<SkipList<T>*>(lhs.get()) =
*dynamic_cast<SkipList<T>*>(rhs.get());
    }
}

```

```

    } else if (type == ImplType::kSplay) {
        *dynamic_cast<SplayTree<T>>(lhs.get()) =
*dynamic_cast<SplayTree<T>>(rhs.get());
    } else if (type == ImplType::kSet) {
        *dynamic_cast<StdlibSet<T>>(lhs.get()) =
*dynamic_cast<StdlibSet<T>>(rhs.get());
    } else {
        throw std::runtime_error("Impossible behaviour");
    }
}

/***
 * Checks if std::set<T> and ITree<T> had the same elements
 * @tparam T Tree value type
 * @param set Std::set for comparison
 * @param tree ITree object for comparison
 * @return True if elements in both structures are same
 */
template <class T>
bool operator==(std::set<T> set, std::shared_ptr<ITree<T>> tree) {
    if (set.size() != tree->size()) {
        return false;
    }
    auto tree_it = tree->begin();
    for (const T& elem : set) {
        REQUIRE_NO_THROW(*tree_it);
        if (elem != *tree_it) {
            return false;
        }
        REQUIRE_NO_THROW(++tree_it);
    }
    REQUIRE(tree_it == tree->end());
    return true;
}

/***
 * Checks if std::set<T> and ITree<T> gave the same answers on find()
and lower_bound() queries.
 * @tparam T Tree value type
 * @param set Std::set for comparison
 * @param tree ITree object for comparison
 * @param value Value for find() and lower_bound() queries
 */
template <class T>
void CheckFindAndLB(const std::set<T>& set, std::shared_ptr<ITree<T>>
tree, const T& value) {
    auto set_it = set.find(value);
    if (set_it == set.end()) {
        REQUIRE(tree->find(value) == tree->end());
    } else {
        auto it = tree->find(value);
        REQUIRE(*it == *set_it);
        if (set_it != set.begin()) {

```

```

        REQUIRE(it != tree->begin());
        REQUIRE(*(--it) == *(--set_it));
        ++it, ++set_it;
    }
    ++set_it, ++it;
    if (set_it != set.end()) {
        REQUIRE(it != tree->end());
        REQUIRE(*it == *set_it);
    }
}

set_it = set.lower_bound(value);
if (set_it == set.end()) {
    auto it = tree->lower_bound(value);
    REQUIRE(it == tree->end());
    if (set_it != set.begin()) {
        REQUIRE(it != tree->begin());
        REQUIRE(*(--set_it) == *(--it));
    }
} else {
    auto it = tree->lower_bound(value);
    REQUIRE(*it == *set_it);
    if (set_it != set.begin()) {
        REQUIRE(it != tree->begin());
        REQUIRE(*(--set_it) == *(--it));
        ++set_it, ++it;
    }
    ++set_it, ++it;
    if (set_it != set.end()) {
        REQUIRE(it != tree->end());
        REQUIRE(*set_it == *it);
    }
}
}

/***
 * Random number generator functor.
 * Uses singleton pattern.
 */
class Random {
public:
    /**
     * Generates random number in the interval [from, to]
     * @tparam T Integral type parameter
     * @param from Start of the interval
     * @param to End of the interval
     * @return Number in the interval [from, to]
     */
    template <class T>
    static uint32_t Next(const T& from, const T& to) {
        static Random rand = Random();
        std::uniform_int_distribution<T> dist(from, to);
        return dist(rand.gen_);
    }
}

```

```

    }

private:
    /**
     * If the build is DEBUG, it generates a predictable identical
     sequence.
     */
    Random() {
#ifndef RELEASE_BUILD
        /// GOOD INIT
        std::random_device device;
        gen_ = std::mt19937(device());
#else
        /// BAD INIT
        gen_ = std::mt19937(0u);
#endif
    }

    std::mt19937 gen_;
};

/***
 * All functions below are same
 *
 * Test
 * @param type Type of tree to check
 */
void SomeTest(ImplType type) {
    auto tree = MakeTree<int>(type);
    tree->insert(1);
    REQUIRE(*tree->begin() == 1);
}

void EmptinessTest(ImplType type) {
    auto tree = MakeTree<int>(type);
    REQUIRE(tree->size() == 0);
    REQUIRE(tree->empty());
    REQUIRE_NO_THROW(tree->clear());
    REQUIRE(tree->size() == 0);
    REQUIRE_NO_THROW(tree->erase(5));
    REQUIRE(tree->empty());
}

void EmptyIteratorsTest(ImplType type) {
    auto tree = MakeTree<int>(type);
    REQUIRE(tree->begin() == tree->end());
    REQUIRE(tree->find(10) == tree->end());
    REQUIRE(tree->lower_bound(0) == tree->end());
    {
        auto it = tree->begin();
        REQUIRE_THROWS_AS(--it, std::exception);
    }
}

```

```

        auto it = tree->begin();
        REQUIRE_THROWS_AS(++it, std::exception);
    }
    {
        auto it = tree->end();
        REQUIRE_THROWS_AS(it--, std::exception);
    }
    {
        auto it = tree->begin();
        REQUIRE_THROWS_AS(it++, std::exception);
    }
    {
        auto it = tree->end();
        REQUIRE_THROWS_AS(++it, std::exception);
    }
    REQUIRE(tree->empty());
}

void EmptyCopyingTest(ImplType type) {
    auto tree1 = MakeTree<int>(type);
    auto tree2 = MakeTree<int, std::shared_ptr<ITree<int>>>(type,
tree1);
    REQUIRE(tree1->empty());
    REQUIRE(tree2->empty());
    REQUIRE(tree1->begin() != tree2->begin());
    REQUIRE(tree1->end() != tree2->end());
    auto tree3 = MakeTree<int, std::shared_ptr<ITree<int>>>(type,
tree1);
    MakeCopyAssignment(type, tree3, tree2);
    REQUIRE(tree2->empty());
    REQUIRE(tree3->empty());
    REQUIRE(tree3->begin() != tree2->begin());
    REQUIRE(tree3->end() != tree2->end());
    REQUIRE_NOTHROW(MakeCopyAssignment(type, tree3, tree3));
}

void FewElementsTest(ImplType type) {
    std::vector<int> fill = {1, 0};
    auto tree = MakeTree<int>(type);
    std::set<int> set;
    for (int& value : fill) {
        tree->insert(value);
        set.insert(value);
        REQUIRE(!tree->empty());
    }
    REQUIRE(set == tree);
    set.clear();
    REQUIRE_NOTHROW(tree->clear());
    REQUIRE(set == tree);
    REQUIRE(tree->empty());
}

void FewElementsIteratorTest(ImplType type) {

```

```

{
    std::vector<int> fill = {3, 4, 2, 5, 1};
    std::set<int> set(fill.begin(), fill.end());
    auto tree = MakeTree<int>(type, fill.begin(), fill.end());
    REQUIRE(set == tree);
    REQUIRE(tree->find(10) == tree->end());
    REQUIRE(tree->lower_bound(0) == tree->begin());
}
{
    std::vector<int> fill = {3, 4, 2, 5, 1};
    std::set<int> set(fill.begin(), fill.end());
    auto tree = MakeTree<int>(type, fill.begin(), fill.end());
    auto it = tree->end();
    REQUIRE_THROWS_AS(*it, std::exception);
    REQUIRE_THROWS_AS(it++, std::exception);
    it = tree->begin();
    REQUIRE_THROWS_AS(--it, std::exception);
}
{
    std::vector<std::pair<std::string, int>> fill = {
        {"one", 1}, {"two", 2}, {"three", 3}, {"four", 4}};
    std::set<std::pair<std::string, int>> set(fill.begin(),
fill.end());
    auto tree = MakeTree<std::pair<std::string, int>>(type,
fill.begin(), fill.end());
    auto it = tree->begin();
    REQUIRE(it->first == "four");
    REQUIRE(it->second == 4);
    ++it, ++it;
    REQUIRE(it->first == "three");
    REQUIRE(it->second == 3);
    it = tree->begin();
    REQUIRE_THROWS_AS(it--, std::exception);
    it = tree->end();
    REQUIRE_THROWS_AS(*it, std::exception);
    REQUIRE_THROWS_AS(it->first, std::exception);
    REQUIRE_THROWS_AS(it->second, std::exception);
    REQUIRE_THROWS_AS(++it, std::exception);
}
{
    auto tree = MakeTree<std::pair<int, int>,
std::initializer_list<std::pair<int, int>>>(
        type, {{0, 1},
                {-5, 0},
                {3, 11},
                {std::numeric_limits<int>::max(),
std::numeric_limits<int>::min()}});
    REQUIRE(tree->begin()->first == -5);
    REQUIRE((--tree->end())->second ==
std::numeric_limits<int>::min());
    tree->clear();
    tree->insert(std::make_pair(1, 1));
    tree->insert(std::make_pair(-1, 1));
}

```

```

    REQUIRE(tree->size() == 2);
    REQUIRE(tree->begin()->first == -1);
}
}

void FewElementsCopyingTest(ImplType type) {
{
    std::set<int> fill = {123, 532, 635, 13, 256, 986};
    auto tree = MakeTree<int>(type, fill.begin(), fill.end());
    auto tree2 = MakeTree<int>(type, tree);
    REQUIRE(tree2->size() == tree->size());
    tree2->erase(532);
    REQUIRE(tree2->size() == 5);
    REQUIRE(tree->size() == 6);
    tree->insert(1);
    tree2->insert(100);
    REQUIRE(tree2->size() == 6);
    REQUIRE(tree->size() == 7);
    REQUIRE(tree->find(1) != tree->end());
    REQUIRE(*tree2->lower_bound(99) == 100);
    tree->clear();
    REQUIRE(tree2->size() == 6);
}
{
    std::set<int> fill = {123, 532, 635, 13, 256, 986};
    auto tree = MakeTree<int>(type, fill.begin(), fill.end());
    auto tree2 = MakeTree<int>(type);
    MakeCopyAssignment(type, tree2, tree);
    REQUIRE(tree2->size() == tree->size());
    tree2->erase(532);
    REQUIRE(tree2->size() == 5);
    REQUIRE(tree->size() == 6);
    tree->insert(1);
    tree2->insert(100);
    REQUIRE(tree2->size() == 6);
    REQUIRE(tree->size() == 7);
    REQUIRE(tree->find(1) != tree->end());
    REQUIRE(*tree2->lower_bound(99) == 100);
    tree->clear();
    REQUIRE(tree2->size() == 6);
}
{
    auto tree = MakeTree<int>(type);
    auto tree2 = MakeTree<int>(type);
    MakeCopyAssignment(type, tree2, tree);
    REQUIRE(tree->empty());
    REQUIRE(tree2->empty());
    tree->insert(10);
    REQUIRE(tree2->empty());
    tree->erase(10);
    REQUIRE(tree2->empty());
    tree2->insert(15);
    tree2->insert(20);
}

```

```

        auto tree3 = MakeTree<int>(type);
        MakeCopyAssignment(type, tree3, tree2);
        tree2->clear();
        REQUIRE(tree3->size() == 2);
    }
}

std::vector<int> fill = {3, 3, -1, 6, 0, 0, 17, -5, 4, 2};
std::set<int> set(fill.begin(), fill.end());
auto tree1 = MakeTree<int>(type, fill.begin(), fill.end());
auto tree2 = MakeTree<int>(type);
MakeCopyAssignment(type, tree2, tree1);
tree2->insert(5);
tree2->insert(18);
tree2->insert(-2);
auto tree1_it = tree1->begin(), tree2_it = tree2->begin();
auto it = set.begin();
while (tree1_it != tree1->end() || tree2_it != tree2->end() || it != set.end()) {
    if (*tree2_it == 5 || *tree2_it == 18 || *tree2_it == -2)
    {
        ++tree2_it;
        continue;
    }
    if (tree1_it == tree1->end() || tree2_it == tree2->end() || it == set.end())
    {
        REQUIRE(tree1_it == tree1->end());
        REQUIRE(tree2_it == tree2->end());
        REQUIRE(it == set.end());
    } else {
        REQUIRE(*tree1_it == *tree2_it);
        REQUIRE(*tree1_it == *it);
        ++tree1_it, ++tree2_it, ++it;
    }
}
}
}
}

```

```

/**
 * @class for testing.
 * It helps to check if the memory is released properly after removing
each object.
 * It also has only `less` operator for comparison.
 */
class StrangeInt {
public:
    static int counter;

    StrangeInt() : value_() {
        ++counter;
    }
    StrangeInt(int value) : value_(value) {
        ++counter;
    }
}

```

```

}

StrangeInt(const StrangeInt& other) : value_(other.value_) {
    ++counter;
}

StrangeInt(StrangeInt&& other) noexcept : value_(other.value_) {
    ++counter;
}

bool operator<(const StrangeInt& other) const {
    return value_ < other.value_;
}

static void init() {
    counter = 0;
}

~StrangeInt() {
    --counter;
}

private:
    int value_;
};

int StrangeInt::counter;

void StrangeTest(ImplType type) {
{
    int count = StrangeInt::counter;
    auto tree = MakeTree<StrangeInt>(type);
    tree->insert(2);
    tree->insert(42);
    tree->clear();
    REQUIRE(count == StrangeInt::counter);
}
{
    int count = StrangeInt::counter;
    std::set<int> fill = {123, 532, 635, 13, 256, 986};
    auto tree = MakeTree<StrangeInt>(type, fill.begin(),
fill.end());
    for (auto& value : *tree) {
        tree->erase(value);
    }
    REQUIRE(count == StrangeInt::counter);
}
int count = StrangeInt::counter;
{
    std::set<int> fill = {123, 532, 635, 13, 256, 986};
    auto tree = MakeTree<StrangeInt>(type, fill.begin(),
fill.end());
}
REQUIRE(count == StrangeInt::counter);
}

void StrangeCopyTest(ImplType type) {

```

```

{
    int count = StrangeInt::counter;
    std::set<int> fill = {123, 532, 635, 13, 256, 986};
    auto tree = MakeTree<StrangeInt>(type, fill.begin(),
fill.end());
    auto tree2 = MakeTree<StrangeInt>(type, tree);
    tree2->insert(1000);
    auto tree3 = MakeTree<StrangeInt>(type);
    MakeCopyAssignment(type, tree3, tree2);
    tree3->erase(1000);
    REQUIRE(tree3->size() == tree->size());
    REQUIRE(tree->size() + 1 == tree2->size());
    tree->clear();
    tree2->clear();
    tree3->clear();
    REQUIRE(count == StrangeInt::counter);
}
int count = StrangeInt::counter;
{
    std::set<int> fill = {123, 532, 635, 13, 256, 986};
    auto tree = MakeTree<StrangeInt>(type, fill.begin(),
fill.end());
    auto tree2 = MakeTree<StrangeInt>(type, tree);
    tree2->insert(1000);
    auto tree3 = MakeTree<StrangeInt>(type);
    MakeCopyAssignment(type, tree3, tree2);
    tree3->erase(1000);
    REQUIRE(tree3->size() == tree->size());
    REQUIRE(tree->size() + 1 == tree2->size());
}
REQUIRE(count == StrangeInt::counter);
}

void FindAndLBTest(ImplType type) {
    for (int count = 0; count < 100; ++count) {
        std::vector<int> fill;
        for (int i = 0; i < 10; ++i) {
            fill.emplace_back(Random::Next(-10, 10));
        }
        std::set<int> set(fill.begin(), fill.end());
        auto tree = MakeTree<int>(type, fill.begin(), fill.end());
        for (int i = 0; i < 40; ++i) {
            CheckFindAndLB<int>(set, tree, Random::Next(-10, 10));
        }
    }
}

void InsertAndEraseTest(ImplType type) {
    for (int count = 0; count < 100; ++count) {
        std::vector<int> fill;
        for (int i = 0; i < 10; ++i) {
            fill.emplace_back(Random::Next(-10, 10));
        }
    }
}

```

```

        std::set<int> set(fill.begin(), fill.end());
        auto tree = MakeTree<int>(type);
        for (const int& value : fill) {
            tree->insert(value);
        }
        for (int i = 0; i < 10; ++i) {
            int value = Random::Next(-10, 10);
            if (Random::Next(0, 1)) {
                set.insert(value);
                tree->insert(value);
            } else {
                set.erase(value);
                tree->erase(value);
            }
            CheckFindAndLB(set, tree, value);
        }
        auto it = set.begin();
        while (!set.empty()) {
            if (Random::Next(0, 5)) {
                ++it;
            } else {
                int value = *it;
                tree->erase(value);
                it = set.erase(it);
                CheckFindAndLB(set, tree, value);
            }
            if (it == set.end()) {
                it = set.begin();
            }
        }
    }
}

void RBBlackHeightTest(ImplType type) {
    if (type != ImplType::kRB) {
        std::cout << "Test is only designed for RB trees. ";
        return;
    }
    for (int count = 0; count < 100; ++count) {
        std::vector<int> fill;
        for (int i = 0; i < 10; ++i) {
            fill.emplace_back(Random::Next(-100, 100));
        }
        std::set<int> set(fill.begin(), fill.end());
        auto tree = std::make_shared<RBTree<int>>();
        for (const int& value : fill) {
            tree->insert(value);
            REQUIRE_NO_THROW(tree->CheckRB());
        }

        for (int i = 0; i < 10; ++i) {
            int value = Random::Next(-100, 100);
            if (Random::Next(0, 1)) {

```

```

        set.insert(value);
        tree->insert(value);
    } else {
        set.erase(value);
        tree->erase(value);
    }
    REQUIRE_NO_THROW(tree->CheckRB());
}
auto it = set.begin();
while (!set.empty()) {
    if (Random::Next(0, 5)) {
        ++it;
    } else {
        int value = *it;
        tree->erase(value);
        it = set.erase(it);
        REQUIRE_NO_THROW(tree->CheckRB());
    }
    if (it == set.end()) {
        it = set.begin();
    }
}
}
}
}

```

Tests/run_all_tests.cpp

```

#include <string>

#include "test_framework.cpp"

/***
 * Runs all tests for all trees in the project
 */
TEST_CASE("Test All") {
    TestFramework framework;
    framework.RunAllForAll();
}

```

Tests/run_tests_separately.h

```

#include <string>

#include "test_framework.cpp"

/***
 * All functions below are same
 *
 * Runs all tests for chosen structure
 */
TEST_CASE("Test AVL") {
    TestFramework framework;
    framework.RunAll(Substr("AVL"));
}

```

```

}

TEST_CASE("Test Cartesian") {
    TestFramework framework;
    framework.RunAll(Substr("Cartesian"));
}

TEST_CASE("Test Red-Black") {
    TestFramework framework;
    framework.RunAll(Substr("Red-Black"));
}

TEST_CASE("Test Skip list") {
    TestFramework framework;
    framework.RunAll(Substr("Skip_list"));
}

TEST_CASE("Test Splay") {
    TestFramework framework;
    framework.RunAll(Substr("Splay"));
}

```

Tests/test_framework.cpp

```

#define CATCH_CONFIG_MAIN
#include "../catch/catch.hpp"
#include <algorithm>
#include <functional>
#include <iostream>
#include <map>
#include <string>
#include <vector>

#include "full_test_set.h"

using std::cout;

/**
 * Framework for testing ITree based structures
 */
class TestFramework {
public:
    TestFramework() {
#ifdef RELEASE_BUILD
        cout << "Test framework started at release build\n\n";
#else
        cout << "Test framework started at debug build\n\n";
#endif
        /// All types of trees are listed below.
        types_.emplace("AVL tree", ImplType::kAVL);
        types_.emplace("Cartesian tree", ImplType::kCartesian);
        types_.emplace("Red-Black tree", ImplType::kRB);
        types_.emplace("Skip list", ImplType::kSkipList);
    }
};

```

```

types_.emplace("Splay tree", ImplType::kSplay);

/**
 * All tests are listed below.
 * We use '!' for useful and essential tests,
 * '%' for simple and demonstrative tests.
 */
tests_.emplace("%_simple_test", SomeTest);
tests_.emplace("%_rb_only_black_height_test",
RBBlackHeightTest);
tests_.emplace("!_emptiness_test", EmptinessTest);
tests_.emplace("!_empty_iterators_test", EmptyIteratorsTest);
tests_.emplace("!_empty_copying_test", EmptyCopyingTest);
tests_.emplace("!_few_elements_test", FewElementsTest);
tests_.emplace("!_few_elements_iterator_test",
FewElementsIteratorTest);
tests_.emplace("!_few_elements_copying_test",
FewElementsCopyingTest);
tests_.emplace("!_strange_test", StrangeTest);
tests_.emplace("!_strange_copy_test", StrangeCopyTest);
tests_.emplace("!_find_and_lower_bound_test", FindAndLBTTest);
tests_.emplace("!_insert_and_erase_test", InsertAndEraseTest);
}

/**
 * This function runs given test with tree types satisfying @param
tree_predicate
 * @tparam TreePredicate Tree predicate functor type
 * @param test_name Name of the test to run
 * @param tree_predicate Functor for choosing trees
 */
template <class TreePredicate>
void RunTest(const std::string &test_name, TreePredicate
tree_predicate) {
    auto it = tests_.find(test_name);
    if (it == tests_.end()) {
        return;
    }
    for (auto &type : types_) {
        if (tree_predicate(type.first)) {
            cout << "Running " << it->first << " on " <<
type.first << ":" ;
            try {
                it->second(type.second);
                cout << "Success!\n\n";
            } catch (std::exception &ex) {
                cout << "Failure: " << ex.what() << "\n\n";
            } catch (...) {
                cout << "Failure: Unknown exception. PLEASE THROW
\"std::exception\" BASED "
                        "EXCEPTIONS\n\n";
            }
        }
    }
}

```

```

        }

    }

/***
 * This function generalizes 'RunTest()' to use all available
trees
 * @param test_name Name of the test to run
 */
void RunTestForAll(const std::string &test_name) {
    RunTest(test_name, Every());
}

/***
 * This function runs tests satisfying @param test_predicate
 * with tree types satisfying @param tree_predicate
 * @tparam TestPredicate Test predicate functor type
 * @tparam TreePredicate Tree predicate functor type
 * @param test_predicate Functor for choosing tests
 * @param tree_predicate Functor for choosing trees
 */
template <class TestPredicate, class TreePredicate>
void RunTests(TestPredicate test_predicate, TreePredicate
tree_predicate) {
    std::unordered_map<std::string, ImplType> trees_for_tests;
    for (auto &type : types_) {
        if (tree_predicate(type.first)) {
            trees_for_tests.insert(type);
        }
    }
    for (auto &test : tests_) {
        if (test_predicate(test.first)) {
            cout << "Running " << test.first << " on some
trees:\n";
            for (auto &tree : trees_for_tests) {
                cout << tree.first << ": ";
                try {
                    test.second(tree.second);
                    cout << "Success!\n";
                } catch (std::exception &ex) {
                    cout << "Failure: " << ex.what() << "\n";
                } catch (...) {
                    cout << "Failure: Unknown exception. PLEASE
THROW \"std::exception\" BASED "
                        "EXCEPTIONS\n";
                }
            }
            cout << "Test passed!\n\n";
        }
    }
}
***/


```

```

    * This function generalizes 'RunTests()' to use all available
trees
    * @tparam TestPredicate Test predicate functor type
    * @param test_predicate Functor for choosing tests
    */
template <class TestPredicate>
void RunTestsForAll(TestPredicate test_predicate) {
    RunTests(test_predicate, Every());
}

/***
    * This function generalizes 'RunTests()' to check all available
tests
    * @tparam TreePredicate Tree predicate functor type
    * @param tree_predicate Functor for choosing trees
    */
template <class TreePredicate>
void RunAll(TreePredicate tree_predicate) {
    RunTests(Every(), tree_predicate);
}

/***
    * This function generalizes 'RunTests()' to check all available
tests
    * on all available trees
    */
void RunAllForAll() {
    RunTests(Every(), Every());
}

/***
    * This function returns a 'std::vector' of tests satisfying
@param test_predicate
    * @tparam TestPredicate Test predicate functor type
    * @param test_predicate Functor for choosing tests
    * @return 'std::vector' of test names satisfying @param
test_predicate
    */
template <class TestPredicate>
std::vector<std::string> ShowTests(TestPredicate test_predicate)
const {
    std::vector<std::string> result;
    for (const auto &test : tests_) {
        if (test_predicate(test.first)) {
            result.emplace_back(test.first);
        }
    }
    std::sort(result.begin(), result.end());
    return result;
}

/***
    * This function returns all available tests

```

```

 * @return 'std::vector' of all available test names
 */
[[nodiscard]] std::vector<std::string> ShowAllTests() const {
    std::vector<std::string> result;
    for (const auto &test : tests_) {
        result.emplace_back(test.first);
    }
    std::sort(result.begin(), result.end());
    return result;
}

private:
    /// Types of trees (name + type)
    std::map<std::string, ImplType> types_;
    /// Tests (name + function)
    std::map<std::string, std::function<void(ImplType)>> tests_;

    /// Functor for every object in a list
    class Every {
public:
    /**
     * Functor, which always returns true
     * @param arg String to compare with
     * @return True
     */
    bool operator()(const std::string &arg) {
        return true;
    }
};

/// Functor for objects, which has a substring in their name matching
/// to @param str_
class Substr {
public:
    /**
     * Constructor for the functor
     * @param str Substring, which we are going to find
     */
    explicit Substr(const char *str) : str_(str) {

    }

    /**
     * Functor
     * @param arg String to compare with
     * @return True if 'str_' is contained in @param arg
     */
    bool operator()(const std::string &arg) {
        return arg.find(str_) != std::string::npos;
    }
};

private:
    std::string str_;

```

```

};

/// Functor for objects, which name matches to the given string
class FullMatch {
public:
    /**
     * Constructor for the functor
     * @param str String, which we are going to compare with
     */
    explicit FullMatch(const char *str) : str_(str) {}

    /**
     * Functor
     * @param arg String to compare with
     * @return True if 'str_' is matched with @param arg
     */
    bool operator()(const std::string &arg) {
        return arg == str_;
    }

private:
    std::string str_;
};

```

Benchmarks/bench_framework.cpp

```

#include <functional>
#include <iostream>
#include <map>
#include <mutex>
#include <queue>
#include <string>
#include <thread>
#include <vector>

#include "benchmarks.h"

using std::cout;

/// stdout access blocking
std::mutex stdout_mutex_;

/**
 * Framework for speed testing ITree based structures
 */
class BenchFramework {
public:
    BenchFramework() {
        /// All types of trees are listed below.
        types_.emplace("AVL_tree", ImplType::kAVL);
        types_.emplace("Cartesian_tree", ImplType::kCartesian);
        types_.emplace("Red-Black_tree", ImplType::kRB);
    }
}

```

```

types_.emplace("Skip_list", ImplType::kSkipList);
types_.emplace("Splay_tree", ImplType::kSplay);
types_.emplace("Stdlib_set", ImplType::kSet);

/**
 * All benchmarks are listed below.
 * To determine it's ours, we put '!' in the beginning
 */
benchmarks_.emplace("!_increasing_int_series_insert_bench",
IncreasingIntSeriesInsert);
benchmarks_.emplace("!_decreasing_int_series_insert_bench",
DecreasingIntSeriesInsert);
benchmarks_.emplace("!_converging_int_series_insert_bench",
ConvergingIntSeriesInsert);
benchmarks_.emplace("!_diverging_int_series_insert_bench",
DivergingIntSeriesInsert);
benchmarks_.emplace("!_random_sparse_int_series_insert_bench",
RandomSparseIntSeriesInsert);
benchmarks_.emplace("!_random_dense_int_series_insert_bench",
RandomDenseIntSeriesInsert);
benchmarks_.emplace("!_random_sparse_strings_insert_bench",
RandomSparseStringsInsert);
benchmarks_.emplace("!_random_dense_strings_insert_bench",
RandomDenseStringsInsert);

benchmarks_.emplace("!_increasing_int_series_erase_after_increasing_se
ries_insert_bench",
IncreasingIntSeriesEraseAfterIncreasingSeriesInsert);

benchmarks_.emplace("!_decreasing_int_series_erase_after_increasing_se
ries_insert_bench",
DecreasingIntSeriesEraseAfterIncreasingSeriesInsert);

benchmarks_.emplace("!_converging_int_series_erase_after_increasing_se
ries_insert_bench",
ConvergingIntSeriesEraseAfterIncreasingSeriesInsert);

benchmarks_.emplace("!_diverging_int_series_erase_after_increasing_ser
ies_insert_bench",
DivergingIntSeriesEraseAfterIncreasingSeriesInsert);

benchmarks_.emplace("!_nonexistent_int_series_erase_after_increasing_s
eries_insert_bench",
NonexistentIntSeriesEraseAfterIncreasingSeriesInsert);

benchmarks_.emplace("!_random_int_series_erase_after_increasing_series
_insert_bench",

```

```

RandomIntSeriesEraseAfterIncreasingSeriesInsert);

benchmarks_.emplace("!_increasing_int_series_erase_after_random_sparse_
_series_insert_bench",
IncreasingIntSeriesEraseAfterRandomSparseSeriesInsert);

benchmarks_.emplace("!_decreasing_int_series_erase_after_random_sparse_
_series_insert_bench",
DecreasingIntSeriesEraseAfterRandomSparseSeriesInsert);

benchmarks_.emplace("!_converging_int_series_erase_after_random_sparse_
_series_insert_bench",
ConvergingIntSeriesEraseAfterRandomSparseSeriesInsert);

benchmarks_.emplace("!_diverging_int_series_erase_after_random_sparse_
series_insert_bench",
DivergingIntSeriesEraseAfterRandomSparseSeriesInsert);
    benchmarks_.emplace(
"!_nonexistent_int_series_erase_after_random_sparse_series_insert_benc
h",
NonexistentIntSeriesEraseAfterRandomSparseSeriesInsert);

benchmarks_.emplace("!_random_int_series_erase_after_random_sparse_ser
ies_insert_bench",
RandomIntSeriesEraseAfterRandomSparseSeriesInsert);

benchmarks_.emplace("!_random_strings_erase_after_random_insert_bench"
,
RandomStringsEraseAfterRandomInsert);

benchmarks_.emplace("!_nonexistent_strings_erase_after_random_insert_b
ench",
NonexistentStringsEraseAfterRandomInsert);

benchmarks_.emplace("!_random_insert_and_erase_int_alternation_bench",
RandomInsertAndEraseIntAlternation);

benchmarks_.emplace("!_find_int_after_random_sparse_insert_bench",
FindIntAfterRandomSparseInsert);

benchmarks_.emplace("!_find_random_sparse_int_after_random_sparse_inse
rt_bench",

```

```

FindRandomSparseIntAfterRandomSparseInsert);

benchmarks_.emplace("!_lower_bound_random_sparse_int_after_random_sparse_insert_bench",

LowerBoundRandomSparseIntAfterRandomSparseInsert);
}

/***
 * This structure specifies how to test our trees
 * begin and end are the boundaries of the value interval
 * if log_scale is set, then step specifies how many values (in
log scale) will be tested
 * if not, then step is just step, which is added to the current
value
 * num_folds specifies the number of identical tests for averaging
 */
struct Range {
    Range() = delete;
    Range(uint64_t begin, uint64_t end, uint64_t step = 1, bool
log_scale = false,
          uint64_t num_folds = 5) {
        begin_ = begin;
        end_ = end;
        step_ = step;
        log_scale_ = log_scale;
        num_folds_ = num_folds;
    }

    uint64_t begin_;
    uint64_t end_;
    uint64_t step_;
    uint64_t num_folds_;
    bool log_scale_;
};

private:
/***
 * This function runs given benchmark with given range and tree
types
 * Results are written to the file, the name of which matches the
name of the bench
 * @param bench Benchmark to run
 * @param path Path to the results folder
 * @param range Range for benchmarking
 * @param types Tree types for benchmarking
 * @param gen Mersenne Twister generator
 */
static void RunBench(
    std::pair<const std::string, std::function<double(ImplType,
std::mt19937 &, uint64_t)>>
        &bench,

```

```

    const std::string &path, const Range &range, const
    std::map<std::string, ImplType> &types,
    std::mt19937 gen) {
    auto begin = std::chrono::high_resolution_clock::now();
    std::ofstream out(path + bench.first + ".csv");
    out.precision(3);
    out << "op_count";
    for (auto &type : types) {
        for (uint64_t i = 0; i < range.num_folds_; ++i) {
            out << ", " << type.first << "_split_" << i;
        }
    }
    try {
        if (range.log_scale_) {
            assert(range.step_ > 1);
            long double step =
                std::pow((long double)(range.end_) / range.begin_,
1.01 / (range.step_ - 1));
            long double cur_approx = range.begin_;
            uint64_t prev = 0;
            for (uint64_t i = 1; i < range.step_; ++i) {
                uint64_t cur = std::floor(cur_approx);
                if (cur == prev) {
                    cur_approx *= step;
                    continue;
                }
                out << '\n' << std::to_string(cur);
                for (auto &type : types) {
                    for (uint64_t fold = 0; fold <
range.num_folds_; ++fold) {
                        out << ", " << std::fixed <<
bench.second(type.second, gen, cur);
                    }
                }
                cur_approx *= step;
                prev = cur;
            }
            out << '\n' << std::to_string(range.end_);
            for (auto &type : types) {
                for (uint64_t fold = 0; fold < range.num_folds_;
++fold) {
                    out << ", " << std::fixed <<
bench.second(type.second, gen, range.end_);
                }
            }
        } else {
            for (uint64_t i = range.begin_; i <= range.end_; i +=
range.step_) {
                out << '\n' << std::to_string(i);
                for (auto &type : types) {
                    for (uint64_t fold = 0; fold <
range.num_folds_; ++fold) {

```

```

                                out << ", " << std::fixed <<
bench.second(type.second, gen, i);
}
}
}
} catch (std::exception &ex) {
    std::lock_guard<std::mutex> lockGuard(stdout_mutex_);
    cout << bench.first << "\tfailure: " << ex.what() << '\n';
} catch (...) {
    std::lock_guard<std::mutex> lockGuard(stdout_mutex_);
    cout << bench.first
        << "\tfailure: Unknown exception. PLEASE THROW
\"std::exception\" BASED "
        "EXCEPTIONS\n";
}
out.close();
auto end = std::chrono::high_resolution_clock::now();
double time =
std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
            nanoMultiplier;
std::lock_guard<std::mutex> lockGuard(stdout_mutex_);
cout << bench.first << ":\tOK. Time spent: " << time <<
"ms\n";
}

public:
/** 
 * This function runs all benchmarks, which satisfy the given
predicate.
 * It takes common range and path to the folder with results.
 * @tparam BenchPredicate Template parameter functor
 * @param path Path to the results folder
 * @param range Range for benchmarking
 * @param bench_predicate Functor for benchmarking
 * @param max_thread_count Maximum number of threads running at
the same time
 */
template <class BenchPredicate>
void RunBenchmarks(const std::string &path, const Range &range,
                    BenchPredicate bench_predicate, uint64_t
max_thread_count) {
    if(max_thread_count < 1){
        cout << "Wrong thread number\n";
        return;
    }
    std::random_device device;
    std::queue<std::thread> threads;
    auto it = benchmarks_.begin();
    while (it != benchmarks_.end() && threads.size() <
max_thread_count - 1) {
        if (bench_predicate(it->first)) {

```

```

        std::cout << "Running " << it->first << '\n';
        std::cout_mutex_.unlock();
        threads.emplace(RunBench, std::ref(*it),
std::ref(path), std::ref(range),
                           std::ref(types_),
std::mt19937(device()));
                           ++it;
}
}

while (it != benchmarks_.end()) {
    if (bench_predicate(it->first)) {
        std::cout << "Running " << it->first << '\n';
        std::cout_mutex_.unlock();
        threads.emplace(RunBench, std::ref(*it),
std::ref(path), std::ref(range),
                           std::ref(types_),
std::mt19937(device()));
                           ++it;
        threads.front().join();
        threads.pop();
    }
}
while (!threads.empty()) {
    threads.front().join();
    threads.pop();
}
}

/***
 * This function generalizes 'RunBenchmarks()' to use all
available benchmarks
 * @param path Path to the results folder
 * @param range Range for benchmarking
 * @param max_thread_count Maximum number of threads running at
the same time
 */
void RunAllBenchmarks(const std::string &path, const Range &range,
uint64_t max_thread_count) {
    RunBenchmarks(path, range, Every(), max_thread_count);
}

private:
    // Types of trees (name + type)
    std::map<std::string, ImplType> types_;
    // Benchmarks (name + function)
    std::map<std::string, std::function<double(ImplType, std::mt19937
&, uint64_t)>> benchmarks_;

    // Functor for every object in a list
    class Every {
public:

```

```

    /**
     * Functor, which always returns true
     * @param arg String to compare with
     * @return True
     */
    bool operator()(const std::string &arg) {
        return true;
    }
};

};

/// Functor for objects, which has a substring in their name matching
// to @param str_
class Substr {
public:
    /**
     * Constructor for the functor
     * @param str Substring, which we are going to find
     */
    explicit Substr(const char *str) : str_(str) {

    }

    /**
     * Functor
     * @param arg String to compare with
     * @return True if 'str_' is contained in @param arg
     */
    bool operator()(const std::string &arg) {
        return arg.find(str_) != std::string::npos;
    }

private:
    std::string str_;
};

/// Functor for objects, which name matches to the given string
class FullMatch {
public:
    /**
     * Constructor for the functor
     * @param str String, which we are going to compare with
     */
    explicit FullMatch(const char *str) : str_(str) {

    }

    /**
     * Functor
     * @param arg String to compare with
     * @return true if 'str_' is matched with @param arg
     */
    bool operator()(const std::string &arg) {
        return arg == str_;
    }
}

```

```

private:
    std::string str_;
};

Benchmarks/benchmarks.h

#pragma once
#include <algorithm>
#include <chrono>
#include <fstream>
#include <iostream>
#include <random>
#include <vector>

#include "../trees/abstract_tree.h"
#include "../trees/avl_tree.h"
#include "../trees/cartesian_tree.h"
#include "../trees/rb_tree.h"
#include "../trees/skip_list.h"
#include "../trees/splay_tree.h"
#include "../trees/stdlib_set.h"

/// Nanoseconds to milliseconds
#define nanoMultiplier 1e-6

/// All types of trees
enum class ImplType { kAVL, kCartesian, kRB, kSkipList, kSplay, kSet
};

/***
 * Makes a tree of given type and returns a shared pointer on it
 * @tparam T Tree value type
 * @tparam Types Types of constructor parameters
 * @param type Type of tree to make
 * @param params Parameters for tree constructor
 * @return shared pointer on a new tree
 */
template <class T, class... Types>
std::shared_ptr<ITree<T>> MakeTree(ImplType type, Types... params) {
    if (type == ImplType::kAVL) {
        return std::make_shared<AVLTree<T>>(params...);
    } else if (type == ImplType::kCartesian) {
        return std::make_shared<CartesianTree<T>>(params...);
    } else if (type == ImplType::kRB) {
        return std::make_shared<RBTree<T>>(params...);
    } else if (type == ImplType::kSkipList) {
        return std::make_shared<SkipList<T>>(params...);
    } else if (type == ImplType::kSplay) {
        return std::make_shared<SplayTree<T>>(params...);
    } else if (type == ImplType::kSet) {
        return std::make_shared<StdlibSet<T>>(params...);
    } else {

```

```

        throw std::runtime_error("Impossible behaviour");
    }
}

/***
 * All functions below are same
 *
 * Benchmark
 * @param type Type of tree to check
 * @param gen Mersenne Twister generator
 * @param op_count Number of operations to do
 * @return Operating time in milliseconds
 */
double IncreasingIntSeriesInsert(ImplType type, std::mt19937& gen,
uint64_t op_count) {
    auto tree = MakeTree<int>(type);
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->insert(i);
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
        nanoMultiplier;
}

double DecreasingIntSeriesInsert(ImplType type, std::mt19937& gen,
uint64_t op_count) {
    auto tree = MakeTree<int>(type);
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->insert(-i);
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
        nanoMultiplier;
}

double ConvergingIntSeriesInsert(ImplType type, std::mt19937& gen,
uint64_t op_count) {
    auto tree = MakeTree<int>(type);
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i<op_count>> 1u; ++i) {
        tree->insert(i);
        tree->insert(op_count - i - 1);
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
        nanoMultiplier;
}

```

```

double DivergingIntSeriesInsert(ImplType type, std::mt19937& gen,
uint64_t op_count) {
    auto tree = MakeTree<int>(type);
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = op_count >> 1u; i < op_count; ++i) {
        tree->insert(i);
        tree->insert(op_count - i - 1);
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
            nanoMultiplier;
}

double RandomSparseIntSeriesInsert(ImplType type, std::mt19937& gen,
uint64_t op_count) {
    auto tree = MakeTree<int>(type);
    std::uniform_int_distribution
dist(std::numeric_limits<int>::min(),
     std::numeric_limits<int>::max());
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->insert(dist(gen));
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
            nanoMultiplier;
}

double RandomDenseIntSeriesInsert(ImplType type, std::mt19937& gen,
uint64_t op_count) {
    auto tree = MakeTree<int>(type);
    std::uniform_int_distribution dist(0ul, op_count / 5);
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->insert(dist(gen));
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
            nanoMultiplier;
}

double RandomSparseStringsInsert(ImplType type, std::mt19937& gen,
uint64_t op_count) {
    auto tree = MakeTree<std::string>(type);
    std::uniform_int_distribution
dist(std::numeric_limits<int>::min(),
     std::numeric_limits<int>::max());
    std::ifstream fin("../experiments/some_text.txt");

```

```

    std::string text;
    getline(fin, text);
    fin.close();
    std::vector<std::string> elements;
    for (uint64_t i = 0; i < op_count; ++i) {
        elements.emplace_back(text + std::to_string(dist(gen)));
    }
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->insert(elements[i]);
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count() *
        nanoMultiplier;
}

double RandomDenseStringsInsert(ImplType type, std::mt19937& gen,
    uint64_t op_count) {
    auto tree = MakeTree<std::string>(type);
    std::uniform_int_distribution dist(0ul, op_count / 5);
    std::ifstream fin("../experiments/some_text.txt");
    std::string text;
    getline(fin, text);
    fin.close();
    std::vector<std::string> elements;
    for (uint64_t i = 0; i < op_count; ++i) {
        elements.emplace_back(text + std::to_string(dist(gen)));
    }
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->insert(elements[i]);
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count() *
        nanoMultiplier;
}

double IncreasingIntSeriesEraseAfterIncreasingSeriesInsert(ImplType
    type, std::mt19937& gen,
    uint64_t
    op_count) {
    auto tree = MakeTree<int>(type);
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->insert(i);
    }
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->erase(i);
    }
    auto end = std::chrono::high_resolution_clock::now();

```

```

        return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
            nanoMultiplier;
    }

double DecreasingIntSeriesEraseAfterIncreasingSeriesInsert(ImplType
type, std::mt19937& gen,
                                                        uint64_t
op_count) {
    auto tree = MakeTree<int>(type);
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->insert(i);
    }
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = op_count - 1; i > 0; --i) {
        tree->erase(i);
    }
    tree->erase(0);
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
            nanoMultiplier;
}

double ConvergingIntSeriesEraseAfterIncreasingSeriesInsert(ImplType
type, std::mt19937& gen,
                                                        uint64_t
op_count) {
    auto tree = MakeTree<int>(type);
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->insert(i);
    }
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i<op_count>> 1u; ++i) {
        tree->erase(i);
        tree->erase(op_count - i - 1);
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
            nanoMultiplier;
}

double DivergingIntSeriesEraseAfterIncreasingSeriesInsert(ImplType
type, std::mt19937& gen,
                                                        uint64_t
op_count) {
    auto tree = MakeTree<int>(type);
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->insert(i);
    }
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = op_count >> 1u; i < op_count; ++i) {

```

```

        tree->erase(i);
        tree->erase(op_count - i - 1);
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
            nanoMultiplier;
}

double NonexistentIntSeriesEraseAfterIncreasingSeriesInsert(ImplType
type, std::mt19937& gen,
                                         uint64_t
op_count) {
    auto tree = MakeTree<int>(type);
    for (uint64_t i = 0; i < op_count << 1u; i += 2) {
        tree->insert(i);
    }
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 1; i < op_count << 1u; i += 2) {
        tree->erase(i);
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
            nanoMultiplier;
}

double RandomIntSeriesEraseAfterIncreasingSeriesInsert(ImplType type,
std::mt19937& gen,
                                         uint64_t
op_count) {
    auto tree = MakeTree<int>(type);
    std::vector<int> elements;
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->insert(i);
        elements.emplace_back(i);
    }
    std::shuffle(elements.begin(), elements.end(), gen);
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->erase(elements[i]);
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
            nanoMultiplier;
}

double IncreasingIntSeriesEraseAfterRandomSparseSeriesInsert(ImplType
type, std::mt19937& gen,
                                         uint64_t
op_count) {
    auto tree = MakeTree<int>(type);

```

```

    std::uniform_int_distribution
dist(std::numeric_limits<int>::min(),
     std::numeric_limits<int>::max());
    std::vector<int> elements;
    for (uint64_t i = 0; i < op_count; ++i) {
        elements.emplace_back(dist(gen));
        tree->insert(elements.back());
    }
    std::sort(elements.begin(), elements.end());
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->erase(elements[i]);
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
           nanoMultiplier;
}

double DecreasingIntSeriesEraseAfterRandomSparseSeriesInsert(ImplType
type, std::mt19937& gen,
                                                               uint64_t
op_count) {
    auto tree = MakeTree<int>(type);
    std::uniform_int_distribution
dist(std::numeric_limits<int>::min(),
     std::numeric_limits<int>::max());
    std::vector<int> elements;
    for (uint64_t i = 0; i < op_count; ++i) {
        elements.emplace_back(dist(gen));
        tree->insert(elements.back());
    }
    std::sort(elements.rbegin(), elements.rend());
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->erase(elements[i]);
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
           nanoMultiplier;
}

double ConvergingIntSeriesEraseAfterRandomSparseSeriesInsert(ImplType
type, std::mt19937& gen,
                                                               uint64_t
op_count) {
    auto tree = MakeTree<int>(type);
    std::uniform_int_distribution
dist(std::numeric_limits<int>::min(),

```

```

std::numeric_limits<int>::max());
    std::vector<int> elements;
    for (uint64_t i = 0; i < op_count; ++i) {
        elements.emplace_back(dist(gen));
        tree->insert(elements.back());
    }
    std::sort(elements.begin(), elements.end());
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i < op_count >> 1u; ++i) {
        tree->erase(elements[i]);
        tree->erase(elements[op_count - i - 1]);
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count() *
        nanoMultiplier;
}

double DivergingIntSeriesEraseAfterRandomSparseSeriesInsert(ImplType
type, std::mt19937& gen,
                                                        uint64_t
op_count) {
    auto tree = MakeTree<int>(type);
    std::uniform_int_distribution
dist(std::numeric_limits<int>::min(),
     std::numeric_limits<int>::max());
    std::vector<int> elements;
    for (uint64_t i = 0; i < op_count; ++i) {
        elements.emplace_back(dist(gen));
        tree->insert(elements.back());
    }
    std::sort(elements.begin(), elements.end());
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = op_count >> 1u; i < op_count; ++i) {
        tree->erase(elements[i]);
        tree->erase(elements[op_count - i - 1]);
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count() *
        nanoMultiplier;
}

double NonexistentIntSeriesEraseAfterRandomSparseSeriesInsert(ImplType
type, std::mt19937& gen,
                                                        uint64_t
op_count) {
    auto tree = MakeTree<int>(type);
    std::uniform_int_distribution
dist(std::numeric_limits<int>::min()),

```

```

std::numeric_limits<int>::max());
for (uint64_t i = 0; i < op_count; ++i) {
    tree->insert(dist(gen));
}
auto begin = std::chrono::high_resolution_clock::now();
for (uint64_t i = 0; i < op_count; ++i) {
    tree->erase(dist(gen));
}
auto end = std::chrono::high_resolution_clock::now();
return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
        nanoMultiplier;
}

double RandomIntSeriesEraseAfterRandomSparseSeriesInsert(ImplType
type, std::mt19937& gen,
                                                       uint64_t
op_count) {
    auto tree = MakeTree<int>(type);
    std::uniform_int_distribution
dist(std::numeric_limits<int>::min(),
     std::numeric_limits<int>::max());
    std::vector<int> elements;
    for (uint64_t i = 0; i < op_count; ++i) {
        elements.emplace_back(dist(gen));
        tree->insert(elements.back());
    }
    std::shuffle(elements.begin(), elements.end(), gen);
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->erase(elements[i]);
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
        nanoMultiplier;
}

double RandomStringsEraseAfterRandomInsert(ImplType type,
std::mt19937& gen, uint64_t op_count) {
    auto tree = MakeTree<std::string>(type);
    std::uniform_int_distribution
dist(std::numeric_limits<int>::min(),
     std::numeric_limits<int>::max());
    std::ifstream fin("../experiments/some_text.txt");
    std::string text;
    getline(fin, text);
    fin.close();
    std::vector<std::string> elements;
    for (uint64_t i = 0; i < op_count; ++i) {

```

```

elements.emplace_back(text + std::to_string(dist(gen)));
tree->insert(elements.back());
}
std::shuffle(elements.begin(), elements.end(), gen);
auto begin = std::chrono::high_resolution_clock::now();
for (uint64_t i = 0; i < op_count; ++i) {
    tree->erase(elements[i]);
}
auto end = std::chrono::high_resolution_clock::now();
return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
    nanoMultiplier;
}

double NonexistentStringsEraseAfterRandomInsert(ImplType type,
std::mt19937& gen,
                                         uint64_t op_count) {
    auto tree = MakeTree<std::string>(type);
    std::uniform_int_distribution
dist(std::numeric_limits<int>::min(),
     std::numeric_limits<int>::max());
    std::ifstream fin("../experiments/some_text.txt");
    std::string text;
    getline(fin, text);
    fin.close();
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->insert(text + std::to_string(dist(gen)));
    }
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->erase(text + std::to_string(dist(gen)));
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
    nanoMultiplier;
}

double RandomInsertAndEraseIntAlternation(ImplType type, std::mt19937&
gen, uint64_t op_count) {
    if (op_count < 10) {
        return 0.0;
    }
    uint64_t step = op_count / 10;
    auto tree = MakeTree<int>(type);
    std::uniform_int_distribution dist(0, static_cast<int>(3 * step));
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i < step << 1u; ++i) {
        tree->insert(dist(gen));
    }
    for (uint64_t i = 0; i < step; ++i) {
        tree->erase(dist(gen));
    }
}

```

```

    }
    for (uint64_t i = 0; i < step << 1u; ++i) {
        tree->insert(dist(gen));
    }
    for (uint64_t i = 0; i < step << 1u; ++i) {
        tree->erase(dist(gen));
    }
    for (uint64_t i = 0; i < step; ++i) {
        tree->insert(dist(gen));
    }
    for (uint64_t i = 0; i < step << 1u; ++i) {
        tree->erase(dist(gen));
    }
    auto end = std::chrono::high_resolution_clock::now();
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
        nanoMultiplier;
}

double FindIntAfterRandomSparseInsert(ImplType type, std::mt19937&
gen, uint64_t op_count) {
    auto tree = MakeTree<int>(type);
    std::uniform_int_distribution
dist(std::numeric_limits<int>::min(),
     std::numeric_limits<int>::max());
    std::vector<int> elements;
    for (uint64_t i = 0; i < op_count; ++i) {
        elements.emplace_back(dist(gen));
        tree->insert(elements.back());
    }
    std::shuffle(elements.begin(), elements.end(), gen);
    // We use counter, so that compiler doesn't apply optimizations
    int counter = 0;
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i < op_count; ++i) {
        auto it = tree->find(elements[i]);
        if (it != tree->end()) {
            counter += *it;
        }
    }
    auto end = std::chrono::high_resolution_clock::now();
    elements.emplace_back(counter);
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count() *
        nanoMultiplier;
}

double FindRandomSparseIntAfterRandomSparseInsert(ImplType type,
std::mt19937& gen,
                                         uint64_t op_count) {
    auto tree = MakeTree<int>(type);

```

```

    std::uniform_int_distribution
dist(std::numeric_limits<int>::min(),
     std::numeric_limits<int>::max());
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->insert(dist(gen));
    }
    // We use counter, so that compiler doesn't apply optimizations
    int counter = 0;
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i < op_count; ++i) {
        auto it = tree->find(dist(gen));
        if (it != tree->end()) {
            counter += *it;
        }
    }
    auto end = std::chrono::high_resolution_clock::now();
    std::vector<int> useless(1, counter);
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count() *
        nanoMultiplier;
}

double LowerBoundRandomSparseIntAfterRandomSparseInsert(ImplType type,
std::mt19937& gen,
                                         uint64_t
op_count) {
    auto tree = MakeTree<int>(type);
    std::uniform_int_distribution
dist(std::numeric_limits<int>::min(),
     std::numeric_limits<int>::max());
    for (uint64_t i = 0; i < op_count; ++i) {
        tree->insert(dist(gen));
    }
    // We use counter, so that compiler doesn't apply optimizations
    int counter = 0;
    auto begin = std::chrono::high_resolution_clock::now();
    for (uint64_t i = 0; i < op_count; ++i) {
        auto it = tree->lower_bound(dist(gen));
        if (it != tree->end()) {
            counter += *it;
        }
    }
    auto end = std::chrono::high_resolution_clock::now();
    std::vector<int> useless(1, counter);
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count() *
        nanoMultiplier;
}

```

Benchmarks/run_benchmarks.cpp

```
#include "bench_framework.cpp"
/**
 * Here you can run the benchmarks
 * @return 0
 */
int main() {
    BenchFramework framework;
    framework.RunAllBenchmarks("../experiments/",
BenchFramework::Range(1, 1'000'000, 35, true), 1);
    // framework.RunAllBenchmarks("../experiments/",
BenchFramework::Range(0, 1'000'000, 200'000));
    // framework.RunAllBenchmarks("../experiments/",
BenchFramework::Range(1, 100'000, 10, true));
    return 0;
}
```

Experiments/Experiments.ipynb

```
%matplotlib inline
import seaborn as sns
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.special import lambertw
pd.plotting.register_matplotlib_converters()
import warnings
warnings.filterwarnings('ignore')
tick_color = "gray"

e = 2.71828182846

def forward(x):
    return np.array([el/np.log(el) if el > e else el for el in x])

def inverse(x):
    return np.array([el*np.log(el) if el > e else el for el in x])

code = r"""
data = pd.read_csv("!_" + title + "_bench.csv",
index_col=0)
data.columns = data.columns.str.strip().str.lower().str.replace(' ', '_').str.replace('(', '').str.replace(')', '')
cur_data = data.loc[:, "avl_tree_split_0":"avl_tree_split_4"]
avl_mean = cur_data.mean(axis="columns")
avl_std = cur_data.std(axis="columns")
cur_data = data.loc[:, "cartesian_tree_split_0":"cartesian_tree_split_4"]
cartesian_mean = cur_data.mean(axis="columns")
cartesian_std = cur_data.std(axis="columns")
cur_data = data.loc[:, "red-black_tree_split_0":"red-
black_tree_split_4"]
red_black_mean = cur_data.mean(axis="columns")
red_black_std = cur_data.std(axis="columns")
```

```

cur_data = data.loc[:, "skip_list_split_0":"skip_list_split_4"]
skip_list_mean = cur_data.mean(axis="columns")
skip_list_std = cur_data.std(axis="columns")
cur_data = data.loc[:, "splay_tree_split_0":"splay_tree_split_4"]
splay_mean = cur_data.mean(axis="columns")
splay_std = cur_data.std(axis="columns")
cur_data = data.loc[:, "stdlib_set_split_0":"stdlib_set_split_4"]
set_mean = cur_data.mean(axis="columns")
set_std = cur_data.std(axis="columns")

plt.figure(figsize=(28, 10))
plt.ylim(0, max(avl_mean.iloc[-1] + avl_std.iloc[-1],
splay_mean.iloc[-1] + splay_std.iloc[-1]))
plt.xlim(0, 1000000)
sns.set_context("notebook", font_scale=2, rc={"lines.linewidth": 2.5})
ax = sns.lineplot(x=data.index, y=avl_mean, label="avl")
sns.lineplot(x=data.index, y=cartesian_mean, label="cartesian")
sns.lineplot(x=data.index, y=red_black_mean, label="red_black")
sns.lineplot(x=data.index, y=skip_list_mean, label="skip_list")
sns.lineplot(x=data.index, y=splay_mean, label="splay_tree")
sns.lineplot(x=data.index, y=set_mean, label="stdlib_set")
ax.fill_between(x=data.index, alpha=0.2, y1=avl_mean - avl_std,
y2=avl_mean + avl_std)
ax.fill_between(x=data.index, alpha=0.2, y1=cartesian_mean -
cartesian_std, y2=cartesian_mean + cartesian_std)
ax.fill_between(x=data.index, alpha=0.2, y1=red_black_mean -
red_black_std, y2=red_black_mean + red_black_std)
ax.fill_between(x=data.index, alpha=0.2, y1=skip_list_mean -
skip_list_std, y2=skip_list_mean + skip_list_std)
ax.fill_between(x=data.index, alpha=0.2, y1=splay_mean - splay_std,
y2=splay_mean + splay_std)
ax.fill_between(x=data.index, alpha=0.2, y1=set_mean - set_std,
y2=set_mean + set_std)
ax.spines["bottom"].set_color(tick_color)
ax.spines["left"].set_color(tick_color)
ax.grid()
ax.set_title(title, color=tick_color)
ax.set_xlabel("Operations count", color=tick_color)
ax.set_ylabel("Milliseconds", color=tick_color)
ax.tick_params(axis="both", colors=tick_color)
ax.set_yscale('function', functions=(forward, inverse))"""

title = "increasing_int_series_insert"
exec(code)

```