

# Задания к работе №5 по фундаментальным алгоритмам.

Все задания реализуются на языке программирования C++ (стандарт C++14 и выше).

Реализованные в заданиях приложения не должны завершаться аварийно; все возникающие исключительные ситуации должны быть перехвачены и обработаны.

Во всех заданиях запрещено пользоваться функциями, позволяющими завершить выполнение приложения из произвольной точки выполнения.

Во всех заданиях при реализации необходимо разделять контексты работы с данными (поиск, сортировка, добавление/удаление, модификация и т. п.) и отправка данных в поток вывода / выгрузка данных из потока ввода.

Во всех заданиях все вводимые (с консоли, файла, командной строки) пользователем данные должны (если не сказано обратное) быть подвергнуты валидации в соответствии с типом валидируемых данных.

Во всех заданиях необходимо контролировать ситуации с невозможностью [пере]выделения памяти; во всех заданиях необходимо корректно освобождать всю выделенную динамическую память.

Все ошибки, связанные с операциями открытия файла, должны быть обработаны; все открытые файлы должны быть закрыты.

Реализованные компоненты должны зависеть от абстракций (см. задание 0), а не от конкретных реализаций абстракций. Для реализованных компонентов должны быть переопределены следующие механизмы классов C++: конструктор копирования, деструктор, оператор присваивания, конструктор перемещения, присваивание перемещением.

0. Опишите контракты взаимодействия и связанные типы для следующих компонент:

- Логгер:

- *severity* - ICS (in-class scope) перечисление с возможными значениями (в порядке увеличения жёсткости): *trace*, *debug*, *information*, *warning*, *error*, *critical*; nested по отношению к интерфейсу *logger*; описывает уровень жёсткости логгирования;
- *logger* - интерфейс, предоставляющий метод записи лога с заданным уровнем жёсткости

*virtual logger \*logger::log(const std::string& target, severity level) const = 0;*

- Аллокатор:

- *memory* - интерфейс, предоставляющий методы для
  - выделения динамической памяти размера *size* байтов из заданного контекста (задекорирован оператором +=)  
*virtual void \*memory::allocate(size\_t target\_size) const = 0;*
  - освобождения ранее выделенной из заданного контекста динамической памяти (задекорирован оператором -=):

*virtual void memory::deallocate(void const \* const target\_to\_dealloc) const = 0;*

- Длинное целое число:

- *bigint* - интерфейс, предоставляющий функционал работы с числом:
  - добавление числа (задекорированы операторами += и + соответственно)  
*virtual bigint \*bigint::add(bigint const \* const summand) = 0;*  
*bigint \*bigint::sum(bigint const \* const summand) const;*

- вычитание числа (задекорированы операторами -= и - соответственно)  
*virtual bigint \*bigint::subtract(bigint const \* const subtrahend) = 0;*  
*bigint \*bigint::subtraction(bigint const \* const subtrahend) const;*

- отношение порядка на двух числах (задекорированы операторами <, >, <=, >= соответственно)

*virtual bool bigint::lower\_than(bigint const \* const other) const = 0;*

*virtual bool bigint::greater\_than(bigint const \* const other) const = 0;*

*virtual bool bigint::lower\_than\_or\_equal\_to(bigint const \* const other) const = 0;*

*virtual bool bigint::greater\_than\_or\_equal\_to(bigint const \* const other) const = 0;*

- отношение эквивалентности на двух числах (задекорированы операторами ==, != соответственно)

*virtual bool bigint::equals(bigint const \* const other) const = 0;*

*virtual bool bigint::not\_equals(bigint const \* const other) const = 0;*

- Функционал для умножения длинных целых чисел:

- *bigint\_multiplication* - интерфейс, предоставляющий метод умножения двух чисел:

*virtual bigint \*bigint\_multiplication::multiply(bigint const \* const left\_multiplier,  
bigint const \* const right\_multiplier) const = 0;*

- Функционал для целочисленного деления длинных целых чисел:
  - *bigint\_dividion* - интерфейс, предоставляющий метод целочисленного деления двух чисел:  

$$\text{virtual bigint* bigint\_division::divide}(\text{bigint const * const dividend, bigint const * const divider, bigint\_multiplication const * const multiplication\_impl}) \text{ const} = 0;$$
- Ассоциативный контейнер:
  - *associative\_container* - родовой интерфейс (кастомизируемые параметры родowego интерфейса: тип ключа *tkey*, тип значения *tvalue*, тип компаратора на ключах *tkey\_comparer*), предоставляющий функционал для работы с ассоциативным контейнером:
    - поиск значения по ключу (задекорированный перегруженным индексатором)  

$$\text{bool associative\_container::find}(\text{key\_value\_pair* target\_key\_and\_result\_value});$$
    - вставка значения по ключу (задекорированный перегруженным оператором +=)  

$$\text{void associative\_container::insert}(\text{const tkey\& key, const tvalue\& value});$$
    - удаление значения по ключу (задекорированный перегруженным оператором -=)  

$$\text{void associative\_container::remove}(\text{const tkey\& key, tvalue *removed\_value});$$
  - *key\_value\_pair* - nested по отношению к интерфейсу *associative\_container* структура, хранящая в себе объекты типа ключа и значения.
- Вероятностный тест простоты:
  - *probabilistic\_primality\_test* - интерфейс, описывающий вероятностный тест для определения простоты числа и предоставляющий метод:  

$$\text{virtual bool probabilistic\_primality\_test::is\_probably\_prime}(\text{bigint const * const value, double epsilon}) \text{ const} = 0;$$

Для каждого типа абстракции и связанных с ними типов предусмотрите отдельное пространство имён.

Для интерфейса длинного числа самостоятельно продумайте абстракции для итераторов.

1. Реализуйте логгер на основе контракта *logger* из задания 0. Ваша реализация логгера должна конфигурироваться двумя способами:

- на основе конфигурационного файла (структуру файла продумайте самостоятельно);
- на основе порождающего паттерна проектирования *builder*.

Конфигурирование объекта логгера позволяет задавать потоки вывода (файловые, консольный) и минимальные *severity* заданных потоков вывода для конкретного объекта логгера (пример: если минимальный *severity* == *warning*, то через данный объект логгера в данный поток вывода будут выводиться логи только с *severity* == *warning*, *error*, *critical*). При повторной настройке уже открытого для объекта логгера потока необходимо обновить уже настроенный *severity*.

Учтите, что один и тот же поток вывода может использоваться одновременно разными объектами логгеров (и *severity* этого потока вывода для разных логгеров также может различаться). При разрушении последнего объекта логгера, связанного с заданным файловым потоком вывода, этот файловый поток вывода должен быть закрыт.

Структура лога:

*[timestamp][severity] message*

где

*timestamp* - текущие <дата по григорианскому календарю>/<время> в формате

*dd/MM/yyyy hh:mm:ss*

*severity* - строковое представление значения жёсткости лога

*message* - передаваемое логгеру сообщение.

Реализуйте и продемонстрируйте работу приложения, в рамках которого

- различными способами конфигурируются несколько объектов логгера (при этом хотя бы два логгера связаны с одним файловым потоком вывода);
- для записи логов в файл/на консоль используются сконфигурированные объекты логгеров;
- при разрушении объектов логгеров закрываются связанные с ними потоки вывода.

2. Реализуйте аллокатор на основе контракта *memory* из задания 0. Выделение и освобождение динамической памяти реализуйте посредством операторов *new/new[]* и *delete/delete[]* соответственно. Продемонстрируйте работу аллокатора, разместив в нём объекты различных типов (числа, строки, объекты собственных типов данных). Предусмотрите логгирование (на основе реализации логгера из задания 1) вызовов интерфейсных методов (на уровне Вашей реализации аллокатора) с указанием:

- текстового описания действия;
- адреса места в памяти (относительно адреса со значением 0 на уровне глобальной кучи), для которого происходит выделение/освобождение;
- неслужебного содержимого освобождаемого участка перед освобождением в виде коллекции значений байт (в диапазоне [0..255]).

3. Реализуйте аллокатор на основе контракта *memory* из задания 0. Выделение памяти реализуйте при помощи методов (с возможностью конфигурации конкретной реализации через конструктор) первого подходящего, лучшего подходящего, худшего подходящего, а освобождение памяти - при помощи метода освобождения в рассортированном списке. Служебные данные для работы аллокатора размещайте в доверенной ему области памяти (размер доверенной области памяти задаётся на уровне конструктора объекта аллокатора и доверенная память при этом запрашивается из аллокатора, передаваемого как параметр по умолчанию конструктору (если аллокатор не передан, память запрашивается из глобальной кучи)). Продемонстрируйте работу аллокатора, разместив в нём объекты различных типов (числа, строки, объекты собственных типов данных). Предусмотрите логгирование (на основе реализации логгера из задания 1) вызовов интерфейсных методов (на уровне Вашей реализации аллокатора) с указанием:

- текстового описания действия;
- адреса места в памяти (относительно адреса начала доверенной аллокатору области памяти), для которого происходит выделение/освобождение;
- неслужебного содержимого освобождаемого участка перед освобождением в виде коллекции значений байт (в диапазоне [0..255]).

4. Реализуйте аллокатор на основе контракта *memory* из задания 0. Выделение памяти реализуйте при помощи методов (с возможностью конфигурации конкретной реализации через конструктор) первого подходящего, лучшего подходящего, худшего подходящего, а освобождение памяти - при помощи метода освобождения с дескрипторами границ. Служебные данные для работы аллокатора размещайте в доверенной ему области памяти (размер доверенной области памяти задаётся на уровне конструктора объекта аллокатора и доверенная память при этом запрашивается из аллокатора, передаваемого как параметр по умолчанию конструктору (если аллокатор не передан, память запрашивается из глобальной кучи)). Продемонстрируйте работу аллокатора, разместив в нём объекты различных типов данных (числа, строки, объекты собственных типов данных, etc.); память под аллокатор запросите из реализации аллокатора из задания 3. Предусмотрите логгирование (на основе реализации логгера из задания 1) вызовов интерфейсных методов (на уровне Вашей реализации аллокатора) с указанием:

- текстового описания действия;
- адреса места в памяти (относительно адреса начала доверенной аллокатору области памяти), для которого происходит выделение/освобождение;
- неслужебного содержимого освобождаемого участка перед освобождением в виде коллекции значений байт (в диапазоне [0..255]).

5. Реализуйте аллокатор на основе контракта *memory* из задания 0. Выделение и освобождение памяти реализуйте при помощи алгоритмов системы двойников. Служебные данные для работы аллокатора размещайте в доверенной ему области памяти (размер доверенной области памяти задаётся на уровне конструктора объекта аллокатора и доверенная память при этом запрашивается из аллокатора, передаваемого как параметр по умолчанию конструктору (если аллокатор не передан, память запрашивается из глобальной кучи)). Продемонстрируйте работу аллокатора, разместив в нём объекты различных типов данных (числа, строки, объекты собственных типов данных, etc.), а также объекты логгеров из задания 1. Предусмотрите логгирование (на основе реализации логгера из задания 1) вызовов интерфейсных методов (на уровне Вашей реализации аллокатора) с указанием:

- текстового описания действия;
- адреса места в памяти (относительно адреса начала доверенной аллокатору области памяти), для которого происходит выделение/освобождение;
- неслужебного содержимого освобождаемого участка перед освобождением в виде коллекции значений байт (в диапазоне [0..255]).



6. Реализуйте класс длинного целого числа на основе контракта *bigint* из задания 0. Распределение памяти под вложенные в объект длинного целого числа данные организуйте через аллокатор, подаваемый объекту через конструктор. Данными объекта числа является информация о знаке числа (хранится в значении типа *int*) и массив цифр в системе счисления с основанием  $2^{8 \times \text{sizeof}(\text{int})}$ . Оптимизируйте расход памяти для чисел в диапазоне

$$[-2^{8 \times \text{sizeof}(\text{int})-1} \dots 2^{8 \times \text{sizeof}(\text{int})-1} - 1]$$

за счёт хранения значения в знаковом поле. Для класса реализуйте перегруженные операторы вставки в поток строкового представления числа в системе счисления с основанием 10 и выгрузки из потока строкового представления числа в системе счисления с основанием 10, а также конструктор от строкового представления числа в системе счисления с основанием 10. Продемонстрируйте работу реализованного класса.

7. Реализуйте контракт *bigint\_multiplication* из задания 0 согласно алгоритму умножения чисел в столбик и алгоритму Карацубы умножения чисел. Продемонстрируйте работу реализованных классов, сравнив время, затраченное на вычисление значения выражения

$$\sum_{i=1}^{10000} i!$$

Вывод данных в файловый поток вывода организуйте при помощи реализации логгера из задания 1. Для каждого метода используйте отдельный объект логгера; вывод данных организуйте в один файл (с указанием найденного значения и использованной для нахождения результата реализации умножения чисел).

8. Реализуйте контракт *bigint\_multiplication* из задания 0 согласно алгоритму Шёнхаге-Штрассена умножения чисел. Продемонстрируйте работу реализованного класса, вычислив и выведя в файловый поток вывода значение выражения

10000

$$\sum_{i=1} i!!$$

и время, затраченное на вычисление. Вывод данных в поток вывода организуйте при помощи реализации логгера из задания 1.

9. Реализуйте контракт *bigint\_division* из задания 0 согласно алгоритму Ньютона деления чисел. Также на основе реализованного контракта реализуйте функцию вычисления значения НОД при помощи расширенного алгоритма Евклида. Продемонстрируйте работу реализованного класса, вычислив и выведя в файловый поток значения двух псевдослучайных (распределение равномерное) чисел из диапазона  $[0 \dots 2^{16384} - 1]$ , значения их НОД, найденные коэффициенты соотношения Безу, а также время, затраченное на нахождение НОД. Вывод данных в поток вывода организуйте при помощи реализации логгера из задания 1.

10. Реализуйте контракт *bigint\_division* из задания 0 согласно алгоритму Бурникеля-Циглера деления чисел. Также на основе реализованного контракта реализуйте функцию вычисления значения НОД при помощи бинарного алгоритма Евклида. Продемонстрируйте работу реализованного класса, вычислив и выведя в файловый поток значения двух псевдослучайных (распределение равномерное) чисел из диапазона  $[0 \dots 2^{16384} - 1]$ , значения их НОД, а также время, затраченное на нахождение НОД. Вывод данных в поток вывода организуйте при помощи реализации логгера из задания 1.

11. Реализуйте родовой класс бинарного дерева поиска на основе контракта *associative\_container* из задания 0. Распределение вложенных в объект дерева данных организуйте через аллокатор, подаваемый объекту через конструктор. В узлах дерева запрещено хранение указателя на родительский узел. Операции CRD реализуйте на основе поведенческого паттерна проектирования “шаблонный метод”, предусмотрев хуки для выполнения в подклассах BST дополнительных операций до/после рекурсивного вызова относительно текущего узла дерева (параметры хуков: стек указателей на элементы дерева, формирующих путь до текущего узла; адрес указателя на текущий узел; параметры декорирующего интерфейсного метода). Реализуйте итераторы для обхода (префиксного, инфиксного, постфиксного) дерева с возвратом из итератора ключа, значения, глубины (относительно корня; глубина корня дерева равна нулю) обходимого узла; для каждого узла дерева, в порядке, определяемом правилом обхода. Также реализуйте защищённые методы малого левого и малого правого поворотов.

Продемонстрируйте работу реализованного функционала.

12. На основе реализованного класса из задания 12 реализуйте родовой класс AVL-дерева. Для реализации пронаследуйте тип узла дерева с добавлением в тип-наследник необходимой информации и сконфигурируйте хуки шаблонных методов.
- Продемонстрируйте работу реализованного класса, заполнив дерево псевдослучайными данными в количестве 10000 элементов и выведя получившееся дерево в файловый поток вывода при помощи логгера, реализованного в задании 1.

13. На основе реализованного класса из задания 12 реализуйте родовой класс красно-чёрного дерева. Для реализации пронаследуйте тип узла дерева с добавлением в тип-наследник необходимой информации и сконфигурируйте хуки шаблонных методов.
- Продемонстрируйте работу реализованного класса, заполнив дерево псевдослучайными данными в количестве 10000 элементов и выведя получившееся дерево в файловый поток вывода при помощи логгера, реализованного в задании 1.



14. На основе реализованного класса из задания 12 реализуйте родовой класс косого дерева. Для реализации пронаследуйте тип узла дерева с добавлением в тип-наследник необходимой информации и сконфигурируйте хуки шаблонных методов.
- Продемонстрируйте работу реализованного класса, заполнив дерево псевдослучайными данными в количестве 10000 элементов и выведя получившееся дерево в файловый поток вывода при помощи логгера, реализованного в задании 1.

15. Реализуйте родовой класс *B*-дерева на основе контракта *associative\_container* из задания 0. Распределение вложенных в объект дерева данных организуйте через аллокатор, подаваемый объекту через конструктор. В узлах дерева запрещено хранение указателя на родительский узел. Реализуйте итераторы для обхода (префиксного, инфиксного, постфиксного) дерева с возвратом из итератора ключа, значения, глубины (относительно корня; глубина корня дерева равна нулю) обходимого узла; для каждого узла дерева, в порядке, определяемом правилом обхода.
- Продемонстрируйте работу реализованного класса, заполнив дерево псевдослучайными данными в количестве 10000 элементов и выведя получившееся дерево в файловый поток вывода при помощи логгера, реализованного в задании 1.

16. Реализуйте родовой класс  $B^+$ -дерева на основе контракта *associative\_container* из задания 0. Распределение вложенных в объект дерева данных организуйте через аллокатор, подаваемый объекту через конструктор. В узлах дерева запрещено хранение указателя на родительский узел. Реализуйте итераторы для обхода (префиксного, инфиксного, постфиксного) дерева с возвратом из итератора ключа, значения, глубины (относительно корня; глубина корня дерева равна нулю) обходимого узла; для каждого узла дерева, в порядке, определяемом правилом обхода.
- Продемонстрируйте работу реализованного класса, заполнив дерево псевдослучайными данными в количестве 10000 элементов и выведя получившееся дерево в файловый поток вывода при помощи логгера, реализованного в задании 1.

17. Реализуйте класс отношения (*relation*), представляющий собой систему индексирования данных произвольного типа. Класс поддерживает следующий функционал:

- хранение коллекции данных (ключи/значения) в виде списка;
- хранение коллекции индексов (ассоциативных контейнеров с различными объектами-распределителями памяти), содержащих адреса объектов узлов с ключами и коллекциями значений, в контейнере `std::map`, в виде значений по строковым ключам следующего вида:

`<identifier_1>,<identifier_2>,...,<identifier_n>`

где идентификаторы имеют максимальную длину 32 символа, могут начинаться с символа буквы и содержать символы цифр и букв (в любом регистре, регистрозависимость имеет место быть); идентификаторы не могут повторяться, порядок идентификаторов произволен, количество идентификаторов произвольно; у двух различных объектов индексов в отношении ключи должны различаться с точностью до порядка следования идентификаторов;

- вставка нового индекса с указанием типа индекса (на базе какого контейнера из реализованных) в виде объекта перечисления, компаратора по ключам, а также ключа индекса;
- удаление индекса по его ключу;
- вставка данных (в список и все индексы) по ключу;
- удаление данных (из списка и всех индексов) по ключу;
- поиск данных по ключу данных и ключу индекса (если ключа индекса не существует, необходимо сгенерировать исключительную ситуацию типа `relation_exception`, производного от `std::exception` и вложенного по отношению к классу отношения).

Методы вставки/удаления индексов/данных, а также генерация ключа индекса перед поиском в отношении должны быть реализованы с применением `fluent`-цепочки вызовов (см. паттерн `builder`). В качестве используемых ассоциативных контейнеров используйте реализации классов из заданий 12-17. Учтите, что необходимо обеспечить хранение данных в коллекции индексов, даже если они равны по компаратору любого (или нескольких) индекса (ов).

Продемонстрируйте работу с реализованным классом.

18. На основе реализованного в заданиях 6-11 функционала реализуйте класс дроби (с кастомным объектом-распределителем памяти), хранящей в себе числитель и знаменатель (знак должен храниться в знаменателе дроби). В произвольный момент времени числитель и знаменатель любого объекта дроби должны быть взаимно просты между собой. Для класса реализуйте методы и задекорируйте их операторами для сложения, вычитания, умножения, деления дробей; реализуйте методы для:

- возведения дроби в целую неотрицательную степень;
- извлечения корня  $n$ -й степени с заданной точностью (в виде дроби);
- вычисления тригонометрических функций ( $\sin$ ,  $\cos$ ,  $\text{tg}$ ,  $\text{ctg}$ ,  $\text{sec}$ ,  $\text{cosec}$ ,  $\arcsin$ ,  $\arccos$ ,  $\text{arctg}$ ,  $\text{arcctg}$ ,  $\text{arcsec}$ ,  $\text{arccosec}$ ) с заданной точностью (в виде дроби).

Продемонстрируйте работу реализованного функционала.

19. На основе реализованного в задании 19 класса дроби реализуйте класс комплексного числа с кастомным объектом-распределителем памяти (вещественная и мнимая части должны являться рациональными числами, репрезентируемыми объектами дробей). Для класса реализуйте методы и задекорируйте их операторами для сложения, вычитания, умножения, деления комплексных чисел; реализуйте методы вычисления аргумента, модуля, корня  $n$ -й степени.

Продемонстрируйте работу реализованного функционала.

20. При помощи класса дроби из задания 18 реализуйте методы вычисления значений  $e$  и  $\pi$  с точностью 50000 знаков после запятой. Результатами работы методов должны являться вычисленные значения и количество затраченных на нахождение итераций.

21. На основе контракта *probabilistic\_primality\_test* реализуйте базовый класс вероятностного теста простоты, предоставляющий закрытый метод для конвертации значения минимальной требуемой вероятности простоты в число итераций теста, а также реализацию контракта на основе набора защищённых чистых виртуальных функций для:

- определения вероятности простоты одной итерации теста;
- выполнения одной итерации теста.

Пронаследуйте реализованный класс вероятностного теста простоты для реализации тестов простоты Ферма, Соловея-Штрассена, Миллера-Рабина.

На основе реализованных тестов простоты реализуйте метод генерации псевдослучайного простого (с заданной вероятностью простоты) числа заданной битовой длины и продемонстрируйте его работу.