Оглавление

Паттерны

1. Назовите основные характеристики шаблонов.	Имя - все шаблоны имеют уникальное имя, служащее для их идентификации; Назначение данного шаблона; Задача, которую шаблон позволяет решить; Способ решения, предлагаемый в шаблоне для решения задачи в том контексте, где этот шаблон был найден; Участники - сущности, принимающие участие в решении задачи; Следствия от использования шаблона как результат действий, выполняемых в шаблоне; Реализация - возможный вариант реализации шаблона.
2. Назовите три основные группы паттернов.	Порождающие паттерны беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей. Структурные паттерны показывают различные способы построения связей между объектами. Поведенческие паттерны заботятся об эффективной коммуникации между объектами.

3. Расскажите про паттерн Одиночка (Singleton).	**Порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.**
	Конструктор помечается как private, а для создания нового объекта Singleton использует специальный метод getInstance(). Он либо создаёт объект, либо отдаёт существующий объект, если он уже был создан.
	private static Singleton instance;
	public **static** Singleton getInstance() {
	if (instance == null) {
	instance = new Singleton();
	}
	return instance;
	}
	+: можно не создавать множество объектов для ресурсоемких задач, а пользоваться одним
	- : нарушает принцип единой ответственности, так как его могут использовать множество объектов

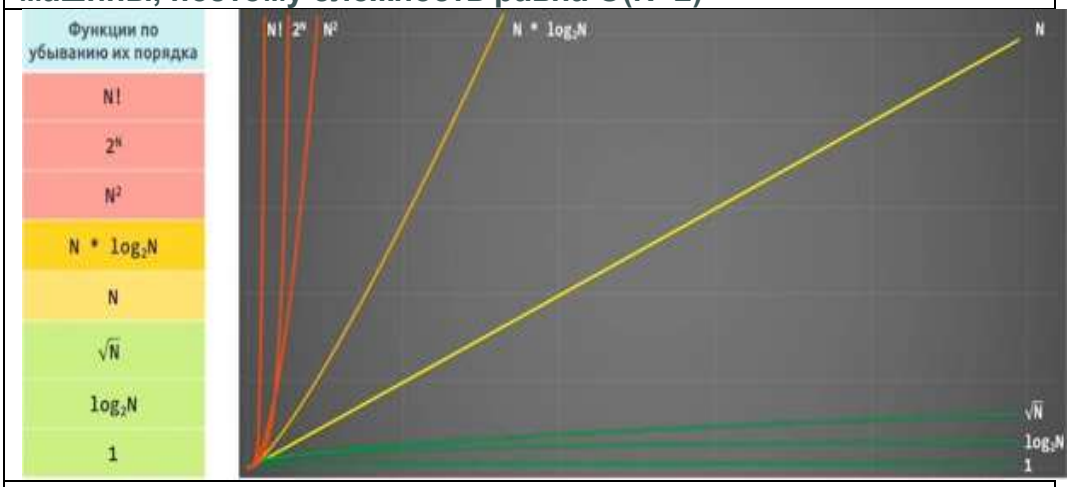
	Почему считается антипаттерном? -Нельзя тестировать с помощью mock, но можно использовать powerMock. -Нарушает принцип единой ответственности -Нарушает Open/Close принцип, его нельзя расширить
	Можно ли его синхронизировать без synchronized у метода? -Можно сделать его Enum (eager). Это статический final класс с константами. JVM загружает final и static классы на этапе компиляции, а значит несколько потоков не могут создать несколько инстансов. -С помощью double checked locking (lazy). Synchronized внутри метода:

	private static volatile Singleton instance;
	public static Singleton getInstance() {
	Singleton localInstance = instance;
	if (localInstance == null) {
	// first check
	synchronized (Singleton.class) {
	localInstance = instance;
	if (localInstance == null) {
	// second check
	instance = localInstance = new Singleton();
	}
	}
	}
	return localInstance;
	}

4. Расскажите про паттерн Строитель (Builder).	**Порождающий паттерн, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений одного объекта.** Паттерн предлагает вынести конструирование объекта за пределы его собственного класса, поручив это дело отдельным объектам, называемым строителями. Процесс конструирования объекта разбить на отдельные шаги (например, построитьСтены, вставитьДвери). Чтобы создать объект, вам нужно поочерёдно вызывать методы строителя. Причём не нужно запускать все шаги, а только те, что нужны для производства объекта определённой конфигурации. Можно пойти дальше и выделить вызовы методов строителя в отдельный класс, называемый **директором**. В этом случае директор будет задавать порядок шагов строительства, а строитель — выполнять их. +: Позволяет использовать один и тот же код для создания различных объектов. Изолирует сложный код сборки объектов от его основной бизнес-логики. - : Усложняет код программы из-за введения дополнительных классов.
5. Расскажите про паттерн Фабричный метод (Factory Method).	Фабричный метод (Factory Method) - это паттерн, который определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах. То есть паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам.
6. Расскажите про паттерн Абстрактная фабрика (Abstract Factory).	**Порождающий паттерн проектирования, который представляет собой интерфейс для создания других классов, не привязываясь к конкретным классам создаваемых объектов.** Абстрактная фабрика предлагает выделить общие интерфейсы для отдельных продуктов, составляющих семейства. Так, все вариации кресел получат общий интерфейс Кресло, все диваны реализуют интерфейс Диван и так далее. Далее вы создаёте абстрактную фабрику — общий интерфейс, который содержит **фабричные методы** создания всех продуктов семейства (например, создатьКресло, создатьДиван и создатьСтолик). Эти операции должны возвращать абстрактные типы продуктов, представленные интерфейсами, которые мы выделили ранее — Кресла, Диваны и Столики. +: гарантированно будет создаваться тип одного семейства - : Усложняет код программы из-за введения множества дополнительных классов.
7. Расскажите про паттерн Прототип (Prototype).	**Порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.** Паттерн поручает создание копий самим копируемым объектам. Он вводит общий интерфейс с методом clone для всех объектов, поддерживающих клонирование. Реализация этого метода в разных классах очень схожа. Метод создаёт новый объект текущего класса и копирует в него значения всех полей собственного объекта. +: Позволяет клонировать объекты, не привязываясь к их конкретным классам. - : Сложно клонировать составные объекты, имеющие ссылки на другие объекты.
8. Расскажите про паттерн Адаптер (Adapter).	**Структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.** Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту. При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о наличии первого. +: Отделяет и скрывает от клиента подробности преобразования различных интерфейсов. - : Усложняет код программы из-за введения дополнительных классов.

9. Расскажите про паттерн Декоратор (Decorator).	**Структурный паттерн проектирования, который позволяет добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».** Целевой объект помещается в другой объект-обёртку, который запускает базовое поведение обёрнутого объекта, а затем добавляет к результату что-то своё. Оба объекта имеют общий интерфейс, поэтому для пользователя нет никакой разницы, с каким объектом работать — чистым или обёрнутым. Вы можете использовать несколько разных обёрток одновременно — результат будет иметь объединённое поведение всех обёрток сразу. Адаптер не менят состояния объекта, а декоратор может менять. +: Большая гибкость, чем у наследования. - : Труднее конфигурировать многократно обёрнутые объекты.
10. Расскажите про паттерн Заместитель (Proxy).	**Структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заменители, которые перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.** Заместитель предлагает создать новый класс-дублёр, имеющий тот же интерфейс, что и оригинальный служебный объект. При получении запроса от клиента объект-заместитель сам бы создавал экземпляр служебного объекта, выполняя промежуточную логику, которая выполнялась бы до (или после) вызовов этих же методов в настоящем объекте. +: Позволяет контролировать сервисный объект незаметно для клиента. - : Увеличивает время отклика от сервиса.
11. Расскажите про паттерн Итератор (Iterator).	**Поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.** Идея состоит в том, чтобы вынести поведение обхода коллекции из самой коллекции в отдельный класс. Детали: Создается итератор и интерфейс, который возвращает итератор. В классе, в котором надо будет вызывать итератор, имплементируем интерфейс, возвращающий итератор, а сам итератор делаем там нестатическим вложенным классом, так как он нигде использоваться больше не будет.
12. Расскажите про паттерн Шаблонный метод (Template Method).	**Поведенческий паттерн проектирования, который пошагово определяет алгоритм и позволяет наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.** Паттерн предлагает разбить алгоритм на последовательность шагов, описать эти шаги в отдельных методах и вызывать их в одном шаблонном методе друг за другом. Для описания шагов используется абстрактный класс. Общие шаги можно будет описать прямо в абстрактном класе. Это позволит подклассам переопределять некоторые шаги алгоритма, оставляя без изменений его структуру и остальные шаги, которые для этого подкласса не так важны.
13. Расскажите про паттерн Цепочка обязанностей (Chain of Responsibility).	Поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи. Базируется на том, чтобы превратить каждую проверку в отдельный класс с единственным методом выполнения. Данные запроса, над которым происходит проверка, будут передаваться в метод как аргументы. Каждый из методов будет иметь ссылку на следующий метод-обработчик, что образует цепь. Таким образом, при получении запроса обработчик сможет не только сам что-то с ним сделать, но и передать обработку следующему объекту в цепочке. Может и не передавать, если проверка в одном из методов не прошла, например.
14. Какие паттерны используются в Spring	**Singleton** - Bean scopes **Factory** - Bean Factory classes **Prototype** - Bean scopes **Adapter** - Spring Web and Spring MVC **Proxy** - Spring Aspect Oriented Programming support **Template Method** - JdbcTemplate, HibernateTemplate etc **Front Controller** - Spring MVC DispatcherServlet **DAO** - Spring Data Access Object support

Framework?	**Dependency Injection**
15. Какие паттерны используются в Hibernate?	**Domain Model** – объектная модель предметной области, включающая в себя как поведение так и данные. **Data Mapper** – слой мапперов (Mappers), который передает данные между объектами и базой данных, сохраняя их независимыми друг от друга и себя. **Proxy** — применяется для ленивой загрузки. **Factory** — используется в SessionFactory
16. Шаблоны GRASP: Low Coupling (низкая связанность) и High Cohesion (высокая сплоченность)	**Low Coupling** - части системы, которые изменяются вместе, должны находиться близко друг к другу. **High Cohesion** - если возвести Low Coupling в абсолют, то можно прийти к тому, чтобы разместить всю функциональность в одном единственном классе. В таком случае связей не будет вообще, но что-то тут явно не так, ведь в этот класс попадет совершенно несвязанная между собой бизнес-логика. Принцип High Cohesion говорит следующее: части системы, которые изменяются параллельно, должны иметь как можно меньше зависимостей друг на друга. Low Coupling и High Cohesion представляют из себя два связанных между собой паттерна, рассматривать которые имеет смысл только вместе. Их суть: система должна состоять из слабо связанных классов, которые содержат связанную бизнес-логику. Соблюдение этих принципов позволяет удобно переиспользовать созданные классы, не теряя понимания о их зоне ответственности.
17. Расскажите про паттерн Saga	Сага — это механизм, обеспечивающий согласованность данных в микросервисах без применения распределенных транзакций. Для каждой системной команды, которой надо обновлять данные в нескольких сервисах, создается некоторая сага. Сага представляет из себя некоторый «чек-лист», состоящий из последовательных локальных ACID-транзакций, каждая из которых обновляет данные в одном сервисе. Для обработки сбоев применяется компенсирующая транзакция. Такие транзакции выполняются в случае сбоя на всех сервисах, на которых локальные транзакции выполнились успешно. Типов транзакций в саге четыре: **Компенсирующая** — отменяет изменение, сделанное локальной транзакцией. **Компенсируемая** — это транзакция, которую необходимо компенсировать (отменить) в случае, если последующие транзакции завершаются неудачей. **Поворотная** — транзакция, опеределяющая успешность всей саги. Если она выполняется успешно, то сага гарантированно дойдет до конца. **Повторяемая** — идет после поворотной и гарантированно завершается успехом.
Алгоритмы	
18. Что такое Big O? Как происходит оценка асимптотической сложности алгоритмов?	Big O (O большое / символ Ландау) - математическое обозначение порядка функции для сравнения асимптотического поведения функций. Асимптотика - характер изменения функции при стремлении ее аргумента к определённой точке. Любой алгоритм состоит из неделимых операций процессора(шагов), поэтому нужно измерять время в операциях процессора, вместо секунд. DTIME - количество шагов(операций процессора), необходимых, чтобы алгоритм завершился.
	Временная сложность обычно оценивается путём подсчёта числа элементарных операций, осуществляемых алгоритмом. Время исполнения одной такой операции при этом берётся константой, то есть асимптотически оценивается как O(1). Сложность алгоритма состоит из двух факторов: временная сложность и сложность по памяти. **Временная сложность -** функция, представляющая

	зависимость количество операций процессора, необходимых, чтобы алгоритм звершился, от размера входных данных. Все неделимые операции языка(операции сравнения, арифметические, логические, инициализации и возврата) считаются выполняемыми за 1 операцию процесора, эта погрешность считается приемлемой. При росте N, слагаемые с меньшей скорость роста всё меньше влияют на значение функции. Поэтому, вне зависимости от констант при слагаемых, **слагаемое с большей скорость роста определяет значение функии. Данное слагаемое называют порядком функции.** Пример: T(N) **= 5 * N^2** + 999 * N... Где (5 * N^2) и (9999 * N) являются слагаемыми функции. **Константы(5 и 999) не указываются в рамках нотации Big O, так как не показывают абсолютную сложность алгоритма, так как могут изменяться в зависимости от машины, поэтому сложность равна O(N^2)**
	
	В порядке возрастания сложности: 1. O(1) - константная, чтение по индексу из массива 2. O(log(n)) - логарифмическая, бинарный поиск в отсортированном массиве 3. O(√n) - сублинейная 4. O(n) - линейная, перебор массива в цикле, два цикла подряд, линейный поиск наименьшего или наибольшего элемента в неотсортированном массиве 5. O(n*log(n)) - квазилинейная, сортировка слиянием, сортировка кучей 6. O(n^2) - полиномиальная(квадратичная), вложенный цикл, перебор двумерного массива, сортировка пузырьком, сортировка вставками 7. O(2^n) - экспоненциальная, алгоритмы разложения на множители целых чисел 8. O(n!) - факториальная, решение задачи коммивояжёра полным перебором **Алгоритм считается приемлемым, если сложность не превышает O(n*log(n)), иначе говнокод.**

19. Что такое рекурсия? Сравните преимущества и недостатки итеративных и рекурсивных алгоритмов. С примерами.	Рекурсия - способ отображения какого-либо процесса внутри самого этого процесса, то есть ситуация, когда процесс является частью самого себя. Рекурсия состоит из базового случая и шага рекурсии. Базовый случай представляет собой самую простую задачу, которая решается за одну итерацию, например, if(n == 0) return 1. В базовом случае обязательно присутствует условие выхода из рекурсии; Смысл рекурсии в движении от исходной задачи к базовому случаю, пошагово уменьшая размер исходной задачи на каждом шаге рекурсии. После того, как будет найден базовй случай, срабатывает условие выхода из рекурсии, и стек рекурсивных вызовов разворачивается в обратном порядке, пересчитывая результат исходной задачи, который основан на результате, найденном в базовом случае. Так работает рекурсивное вычисление факториала: **int factorial(int n) {** **if(n == 0) return 1;** // базовый случай с условием выхода **else return n * factorial(n - 1);** // шаг рекурсии (рекурсивный вызов) **}** Или даже так: **return (n==0) ? 1 : n * factorial(n-1);** **Рекурсия имеет линейную сложность O(n);**
	Циклы дают лучшую производительность, чем рекурсивные вызовы, поскольку вызовы методов потребляют больше ресурсов, чем исполнение обычных операторов. Циклы гарантируют отсутствие переполнения стека, т.к. не требуется выделения доп. памяти. В случае рекурсии стек вызовов разрастается, и его необходимо просматривать для получения конечного ответа. При использовании головной рекурсии также необходимо принимать во внимание размер стека. Если уровней вложенности много или изменятся, то предпочительна рекурсия. Если их несколько, то лучше цикл.

20. Что такое жадные алгоритмы? Приведите пример.	Жадные алгоритмы являются одной из 3х техник создания алгоритмов, вместе с принципом "Разделяй и властвуй" и динамическим программированием. **Жадный алгоритм - это алгоритм, который на каждом шагу совершает локально оптимальные решения, т.е. максимально возможное из допустимых, не учитывая предыдущие или следующие шаги. Последовательность этих локально оптимальных решений приводит (не всегда) к глобально оптимальному решению.** **Т.е. задача рабивается на подзадачи, в каждой подзадаче делается оптимальное решение и, в итоге, вся задача решается оптимально. При этом важно является ли каждое локальное решение безопасным шагом. Безопасный шаг - приводящий к оптимальному решению.** К примеру, алгоритм Дейкстры нахождения кратчайшего пути в графе вполне себе жадный, потому что мы на каждом шагу ищем вершину с наименьшим весом, в которой мы еще не бывали, после чего обновляем значения других вершин. При этом можно доказать, что кратчайшие пути, найденные в вершинах, являются оптимальными.
21. Расскажите про пузырьковую сортировку.	Будем идти по массиву слева направо. Если текущий элемент больше следующего, меняем их местами. Делаем так, пока массив не будет отсортирован. Асимптотика в худшем и среднем случае – O(n^2), в лучшем случае – O(n) - массив уже отсортирован.
22. Расскажите про быструю сортировку.	Выберем некоторый опорный элемент(pivot). После этого перекинем все элементы, меньшие его, налево, а большие – направо. Для этого используются дополнительные переменные - значения слева и справа, которые сравниваются с pivot. Рекурсивно вызовемся от каждой из частей, где будет выбран новый pivot. В итоге получим отсортированный массив, так как каждый элемент меньше опорного стоял раньше каждого большего опорного. Асимптотика: O(n*log(n)) в среднем и лучшем случае. Наихудшая оценка O(n^2) достигается при неудачном выборе опорного элемента.
23. Расскажите про сортировку слиянием.	Основана на парадигме «разделяй и властвуй». Будем делись массив пополам, пока не получим множество массивов из одного элемента. После чего выполним процедуру слияния: поддерживаем два указателя, один на текущий элемент первой части, второй – на текущий элемент второй части. Из этих двух элементов выбираем минимальный, вставляем в ответ и сдвигаем указатель, соответствующий минимуму. Так сделаем слияния массивов из 1го элемента в массивы по 2 элемента, затем из 2х в 4 и т.д. Слияние работает за O(n), уровней всего log(n), поэтому **асимптотика O(n*log(n))**.
24. Расскажите про бинарное дерево.	Бинарное дерево - иерархическая структура данных, в которой каждый узел может иметь двух потомков. Как правило, первый называется родительским узлом, а наследники называются левым и правым нодами/узлами. Каждый узел в дереве задаёт поддерево, корнем которого он является. Оба поддерева — левое и правое — тоже являются бинарными деревьями. Ноды, которые не имеют

	потомков, называются листьями дерева. У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X. У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X. Этим достигается упорядоченная структура данных, то есть всегда отсортированная. Поиск в лучшем случае - O(log(n)), худшем - O(n) - при вырождении в связанный список.
25. Расскажите про красно-черное дерево.	Усовершенствованная версия бинарного дерева. Каждый узел в к/ч дереве имеет дополнительное поле - цвет. К/ч дерево отвечает следующим требованиям: 1) Узел либо красный, либо черный. 2) Корень - черный. 3) Все листья - черные и не хранят данных. 4) Оба потомка каждого красного узла - черные. 5) Любой простой путь от узла-предка до листового узла-потомка содержит одинаковое число черных узлов. Если не одинаковое, то происходит переворот. При добавлении постоянно увеличивающихся/уменьшающихся чисел в бинарное дерево, оно вырождается в связанный список и теряет свои преимущества. Тогда как к/ч дерево может потребовать до двух поворотов для поддержки сбалансированности, чтобы избежать вырождения. При операциях удаления в бинарном дереве для удаляемого узла надо найти замену. К/ч дерево сделает тоже самое, но потребует до трёх поворотов для поддержки сбалансированности. В этом и состоит преимущство. Сложность поиска, вставки и удаления - O(log(n))
26. Расскажите про линейный и бинарный поиск.	Линейный поиск - сложность O(n), так как все элементы проверяются по очереди. Бинарный поиск - O(log(n)). Массив должен быть отсортирован. Происходит поиск индекса в массиве, содержащего искомое значение. 1) Берем значение из середины массива и сраваем с искомым. Индекс середины считается по формуле mid = (high + low) / 2 low - индекс начала левого подмассива, high - индекс конца правого подмассива. 2) Если значение в середине больше искомого, то рассматриваем левый подмассив и high = middle - 1 3) Если меньше, то правый и low = middle + 1 4) Повторяем, пока mid не страновится равен искомому элементу или подмассив не станет пустым.

	public static int binarySearch(int[] a, int key) {
	int low = 0;
	int high = a.length - 1;
	while (low <= high) {
	int mid = (low + high)/2;
	if (key > a[mid]) {
	low = mid + 1;
	} else if (key < a[mid]) {
	high = mid - 1;
	} else return mid;

```
}
return -1;
}
```

27. Расскажите про очередь и стек.

Stack это область хранения данных, находящееся в общей оперативной памяти (RAM). Всякий раз, когда вызывается метод, в памяти стека создается новый блок-фрейм, который содержит локальные переменные метода и ссылки на другие объекты в методе. Как только метод заканчивает работу, блок также перестает использоваться, тем самым предоставляя доступ для следующего метода. Размер стековой памяти намного меньше объема памяти в куче. Стек в Java работает по схеме LIFO

Queue - это очередь, которая обычно (но необязательно) строится по принципу FIFO (First-In-First-Out) - соответственно извлечение элемента осуществляется с начала очереди, вставка элемента - в конец очереди.

Хотя этот принцип нарушает, к примеру PriorityQueue, использующая «natural ordering» или переданный Comparator при вставке нового элемента.

Deque (Double Ended Queue) расширяет Queue и согласно документации это линейная коллекция, поддерживающая вставку/извлечение элементов с обоих концов. Помимо этого реализации интерфейса Deque могут строится по принципу FIFO, либо LIFO.

Реализации и Deque, и Queue обычно не переопределяют методы equals() и hashCode(), вместо этого используются унаследованные методы класса Object, основанные на сравнении ссылок.

28. Сравните сложность вставки, удаления, поиска и доступа по индексу в ArrayList и LinkedList.

	Временная сложность							
	Среднее				*Худшее*			
	Индекс	Поиск	Вставка	Удаление	Индекс	Поиск	Вставка	Удаление
ArrayList	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Vector	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
LinkedList	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Hashtable	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
HashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeMap	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))
HashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeSet	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))