Урок 1

Общие сведения об алгоритмах и структурах данных

Введение в алгоритмы и структуры данных.

Введение

В этом курсе для работы со структурами мы разработаем алгоритмы, связанные с поиском, удалением, вставкой, выводом, прохождением и сортировкой данных. Рассмотрим, как рассчитывается эффективность алгоритма. Узнаем, что такое О-большое.

Программируя на Java, не стоит слишком задумываться о реализации структур данных. Java — это не просто язык программирования, а огромная платформа, которая включает множество библиотек, в том числе реализации структур данных. Но понимать работу этих структур необходимо, чтобы правильно их выбирать при решении задач.

Коротко о том, что будем изучать. В первом уроке — общие сведения об алгоритмах и структурах данных. Краткие сведения об ООП, так как этот подход к программированию мы будем использовать. Рассмотрим примитивные и ссылочные типы данных. Научимся вычислять эффективность алгоритма с помощью О-синтаксиса.

Второй урок посвящен массивам. Рассмотрим алгоритмы поиска, вставки и удаления элемента. Узнаем про сдвиги в массивах. Изучим примитивные сортировки методом пузырька, вставок, выбора.

В третьем уроке перейдем к ADT — абстрактным структурам данных, о которых поговорим чуть позже. Начнем со стека и очереди. Посмотрим, где они могут использоваться и как организуются.

В четвертом уроке поговорим о связанных списках. Рассмотрим различия между списками и массивами для хранения элементов данных. Реализуем очередь и стек на основе связанного списка.

Пятый урок посвящен интересному подходу в программировании — рекурсии. В этом уроке мы не будем рассматривать новую структуру данных, а попробуем изменить некоторые алгоритмы, которые раньше решались через циклы. Также рекурсия часто используется в деревьях.

Им посвящен шестой урок: изучим двоичные деревья. Рассмотрим простейшую древовидную структуру — несбалансированное дерево двоичного поиска. Разберем вставку, удаление и обход таких деревьев.

Седьмой урок — о графах. Рассмотрим задачи поиска в глубину и ширину.

В восьмом уроке поговорим о хеш-таблицах. Это интересная структура, в которой вставка, поиск и удаление происходит за время O(1).

Понятие о структурах данных и алгоритмах

Под структурой данных понимается способ хранения данных в памяти компьютера. Это могут быть массивы, связанные списки, деревья, стеки, очереди и т.д.

Прежде чем говорить о структурах данных, рассмотрим, как работает память компьютера. Разберем это на примере коробок, в которые перекладывают вещи во время переезда.

Представим, что в коробку можно положить только одну вещь. Перевезти необходимо восемь вещей, так что для этого понадобится восемь коробок. В принципе, можно сказать, что так работает память компьютера, только вместо коробок у памяти — ячейки. Чтобы понимать, в какую коробку мы положили конкретную вещь, ее можно подписать. Так же и у памяти компьютера — у каждой ячейки есть адрес.

Книги

Адрес: ***fe001aab***

Зубная паста

Адрес: ***ee4543aa***

Каждый раз, когда мы хотим что-то сохранить в памяти компьютера, запрашиваем у него место, а он выдает адрес, где можно сохранить данные. Так какие же данные можно сохранить в памяти компьютера?

Ссылочные и примитивные типы данных

В Java типы данных можно разделить на два вида: примитивные и ссылочные. Разница — в способе хранения. Рассмотрим простой пример, в котором объявляются две переменные:

```
int a;
Person p1;
```

В первой строке объявляется примитивная переменная типа **int**. Когда этой переменной будет передано значение, оно будет храниться в области памяти, которая связана с именем **a**.

Во второй строке переменная **p1** является ссылочной. Область памяти здесь ассоциируется с **p1** и не содержит данных, а хранит только адрес памяти, в которой располагается объект типа **Person**.

Пока области памяти **p1** не будет присвоен объект, в ней будет храниться ссылка на специальный объект **null**. Таким же образом область памяти «**a**» не содержит значения, пока оно не присвоено. Если в программе обратиться к переменной «**a**» до присвоения ей значения, компилятор выдаст ошибку.

```
public static void main(String[] args) {
      int a;
      Person p1;

      System.out.println(a);
}

Exception in thread "main" java.lang.RuntimeException: Uncompilable source code
- variable a might not have been initialized
      at com.structalgorithm.lesson1.Lesson1App.main(Lesson1App.java:21)
```

Компилятор выдает ошибку о том, что переменная «**a**» не инициализирована.

В отличие от C++, в Java для работы со ссылочными типами данных не используются указатели, что делает код намного понятнее. Но только визуально: указатели скрыты от программиста, но они есть, и все ссылочные типы их используют.

Что касается оператора присваивания: когда речь идет о ссылочных типах, копируется только адрес.

```
p1 = p2;
```

Теперь имена **p1** и **p2** относятся к одному объекту, то есть ссылаются на него.

Так как ссылочные типы данных — это объекты, они должны быть созданы с помощью специального оператора **new**. Он возвращает ссылку на объект.

```
Person p1 = new Person();
```

Возвращает ссылку на адрес, а не указатель, как в C++, который содержит адрес в памяти, где хранится объект. Поэтому программист не знает фактический адрес **p1**.

Как же освободить память, которая была выделена под объект, если мы не знаем, где расположены данные? Это не нужно, так как Java периодически просматривает все блоки памяти, которые были выделены с помощью оператора **new**, и проверяет, есть ли на них использующиеся ссылки. Если ссылок нет, то память чистится — выполняется уборка мусора. В Java реализован специальный помощник, занимающийся этим автоматически — сборщик мусора.

Рассмотрим, как передаются аргументы ссылочных и примитивных типов данных в методы.

```
public static void main(String[] args) {
	int a = 5;
	changeValue(a);
	System.out.println(a);
}

public static void changeValue(int a){
	a = 10;
}
```

В этом примере создается переменная примитивного типа данных **int**. Когда аргументом передается переменная примитивного типа, создается ее копия. Поэтому переменная **a**, объявленная в методе **main**, и переменная **a**, переданная в метод **changeValue,** — это две разные переменные. Выполнив листинг выше, получим следующий ответ:

```
--- exec-maven-plugin:1.2.1:exec (default-cli) @ lesson2 ---

5

------------------------------------------------------------------------

BUILD SUCCESS

------------------------------------------------------------------------

Total time: 0.582s

Finished at: Sun Nov 12 12:19:48 MSK 2017

Final Memory: 5M/119M
```

Сделаем то же самое для переменной ссылочного типа данных:

```
public static void main(String[] args) {
	Person p1 = new Person("Artem");
	changeName(p1);
	System.out.println(p1.name);
}

public static void changeName(Person p1){
	p1.name = "Dmitriy";
}
```

Создадим объект **person**, в котором есть одно поле типа **String**, и присвоим ему значение **Artem**. Передадим переменную **p1** в метод **changeName**. Запустим программу и получим следующий результат:

```
--- exec-maven-plugin:1.2.1:exec (default-cli) @ lesson2 ---

Dmitriy

------------------------------------------------------------------------

BUILD SUCCESS

------------------------------------------------------------------------

Total time: 0.582s

Finished at: Sun Nov 12 12:19:48 MSK 2017

Final Memory: 5M/119M
```

В результате выполнения программы имя объекта **p1** изменилось. В качестве аргумента было передано не значение **p1**, а ссылка на адрес памяти, где хранятся данные **p1**.

Поговорим о равенстве. Для примитивных типов, где сравниваются значения, все выглядит просто:

```
public static void main(String[] args) {
	int a = 5;
	int b = 5;

	if (a == b) {
		System.out.println("a и b равны");
	} else {
		System.out.println("a и b не равны");
	}
}
```

Когда сравниваются два примитивных типа, сопоставляются их значения. Поэтому при сравнении значений 5 и 5 будет выведена надпись «a и b равны».

Если речь о ссылках на объекты — такое сравнение вернет ложный результат:

```
public static void main(String[] args) {
	Person a = new Person("Artem");
	Person b = new Person("Artem");

	if (a == b) {
		System.out.println("a и b равны");
	} else {
		System.out.println("a и b не равны");
	}
}
```

Разберемся, почему так происходит. Ранее было сказано, что оператор **new** возвращает ссылку на адрес в памяти, где физически будет расположен объект. Так как мы дважды вызываем оператор **new**, вернутся две разные ссылки. Сравнивая их, мы и получаем **false**.

Для сравнения ссылочных типов данных в Java используют метод **equals**.

```
public static void main(String[] args) {
	Person a = new Person("Artem");
	Person b = new Person("Artem");

	if (a.equals(b)){
		System.out.println("a и b равны");
	} else {
		System.out.println("a и b не равны");
	}
}
```

Метод **equals** необходимо переопределить — в данном случае в классе **Person**.

```
@Override
public boolean equals(Object obj) {
	if (obj == this) {
		return true;
	}
	if (!(obj instanceof Person)) {
		return false;
	}
	Person p = (Person) obj;

	return p.name == this.name;
}
```

В таблице перечислены все примитивные типы данных, которые есть в Java. Те, что отсутствуют здесь, являются ссылочными.

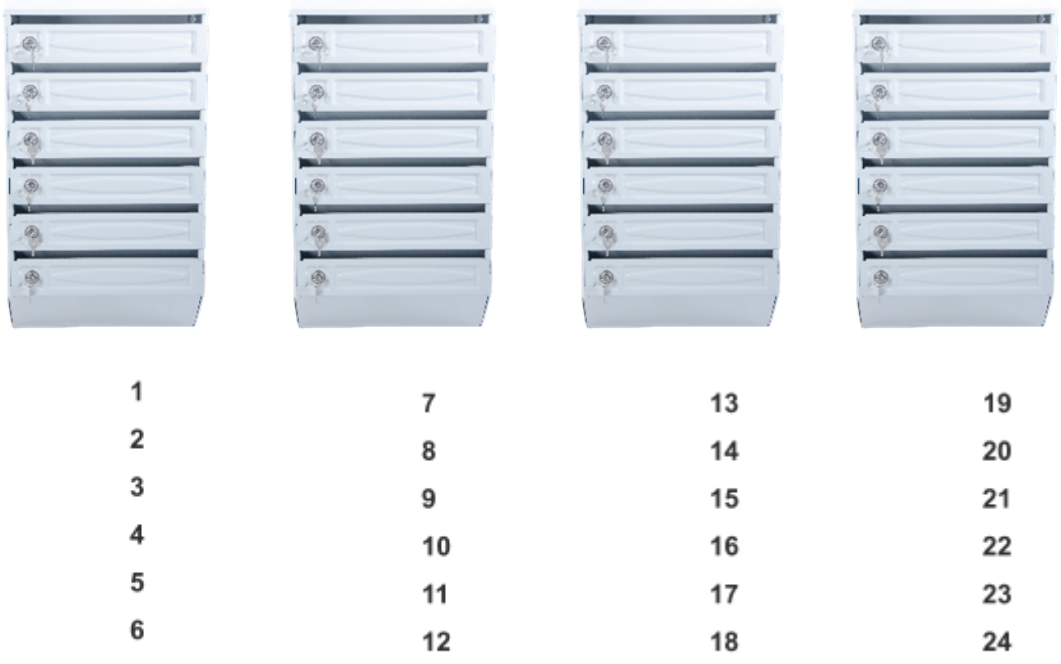
Наименование	Размер в битах	Диапазон значений
int	32	от -2 147 483 648 до 2 147 483 647
float	32	от 10^{-38} до 10^{38} с 7 значащими цифрами

double	64	от 10^{-308} до 10^{308} с 15 значащими цифрами
Byte	8	от -128 до 127
Short	16	от -32 768 до 32 767
Long	64	от -9 223 372 036 854 775 808 до +9 223 372 036 854 775 807
Char	16	от `u0000` до `uFFFF`
Boolean	1	True, false

Немного про массивы

Чтобы хранить множество данных, можно использовать массивы.

Пример: почтальон приносит почту в многоквартирный дом. У него полная сумка корреспонденции для жильцов. Если он положит ее всю в одну коробку, жильцам будет сложно найти свою почту. То ли дело — использовать для каждой квартиры почтовые ящики и разложить письма и газеты в них.



Почтовые ящики пронумерованы аналогично квартирам. Теперь жильцы легко найдут свою почту. Для этого необходимо открыть ящик со своим номером. Так же работают массивы. Для них в памяти выделяется место для хранения определенного количества элементов, и к каждому из этих элементов можно обратиться по индексу. Подробно рассмотрим массивы в следующем уроке.

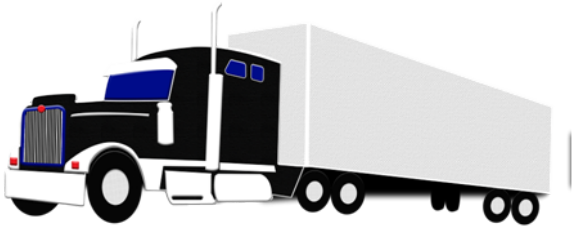
Все остальные структуры — связанные списки, стеки, очереди, деревья, графы, хеш-таблицы — относятся к абстрактным типам данных.

Абстрактный тип данных

Типы данных — это элементы данных, обладающие определенными характеристиками и операциями, которые можно совершать над этими данными. Например, примитивный тип **int**: если говорить про элемент данных, то это целое число в диапазоне от -2 147 483 648 до 2 174 483 647. Над этим числом можно проводить операции сложения, вычитания, деления, умножения и другие. Такие операции являются неотъемлемой частью его смыслового содержания. Чтобы понять смысл типа данных, нужно посмотреть, какие операции над ним можно выполнять.

С появлением ООП стало возможным создавать свои типы данных. В качестве каркаса стали выступать классы. Они содержат поля, являющиеся характеристиками, и методы, которые являются операциями над данными. Например, класс для представления даты и времени или дробных чисел.

Абстракция — это отделение логики работы какой-либо структуры от конкретной реализации. Это модель поведения объекта. Например, какие общие характеристики можно выделить у грузовых, легковых машин и автобусов?



К общим характеристикам машин можно отнести:

- количество колес;
- объем двигателя;
- массу;
- количество посадочных мест.

К общим операциям можно отнести:

- завести двигатель;
- остановить двигатель;
- нажать на педаль газа;
- нажать на педаль тормоза;
- и так далее.

Когда описаны основные характеристики для машин, можно сказать, что создана абстракция, которая их характеризует. Теперь на ее основе можно сделать конкретные классы с реализацией для автобусов, грузовых и легковых машин.

В объектно-ориентированном программировании абстрактным типом данных называется структура, которая не зависит от конкретной реализации, а содержит характеристики (поля класса) и операции над ними (методы).

В данном курсе кроме массивов мы будем создавать структуры данных, которые относятся к абстрактным типам. Для простоты не будем использовать интерфейсы и модификаторы доступа **private**.

Вернемся к примеру массива, где почтальон разносил почту. Можно ли заменить массив для хранения набора данных?

Приведем еще пример. Представьте, что на работе есть список дел для целого отдела. Сотруднику Иванову дали на месяц несколько заданий, которые он должен выполнить в определенном порядке.

Январь 2018

1 Ремонт компьютера	2 Установка антивируса	3 Почистить компьютер	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28

Представим, что этот календарь — память компьютера. Дни, которые закрашены градиентом, заняты другими специалистами. Что будет, если хранить эти данные в массиве? Перед сотрудником Ивановым поставили новую задачу. Он должен установить операционную систему на компьютер пользователя. Как добавить эту задачу в календарь, если для массива выделяется конкретная область, а каждый элемент вставляется справа от предыдущего? В нашем случае четвертая ячейка занята, и записать новую задачу не представляется возможным.

Но календарь можно представить в виде не таблицы, а списка дел.

Теперь можем добавлять сколько угодно дел, если будем использовать список. Но как отделить в памяти компьютера дела Иванова от дел Петрова? Сделаем список связанным и для каждого

элемента проставим ссылку на следующий. Тогда они могут располагаться в памяти где угодно, так как перелинкованы.

Структура такого типа данных будет обладать следующими характеристиками:

- Данные (в нашем случае — задача);
- Ссылка в памяти компьютера на следующий элемент.

Теперь память компьютера будет выглядеть так:

Январь 2018

1 Ремонт компьютера	2 Установка антивируса	3 Почистить компьютер	4	5	6	7
8	9	10	11	12 Установить ОС	13	14
15 Настроить 1С	16	17	18	19 Установить ПО	20	21
22	23	24 Заменить блок питания	25	26	27	28

Обозначим еще несколько структур данных, о которых подробнее поговорим в следующих уроках.

Представим, что мы разработали свою социальную сеть. Она растет с каждым днем: уже зарегистрировано более десяти миллионов человек. При регистрации есть условие: каждый новый пользователь должен иметь уникальный логин. То есть программа должна сопоставить логин регистрирующегося пользователя с логинами уже зарегистрированных, а при совпадении вывести сообщение об ошибке.

Попробуем использовать массив для хранения информации о логинах:

1 Viktor	2 Main	3 Worker	4	5	6	7
8	9	10	11	12 Builder	13	14
15 Boiler	16	17	18	19 Warrior	20	21
22	23	24 Zeus	25	26	…	10000000 philip

Новый пользователь хочет зарегистрироваться под логином **Philip**.

Программа начинает перебирать все элементы массива:

Первый элемент — Viktor != Philip

Второй элемент — Main != Philip

Третий элемент — Worker != Philip

….

Элемент десять миллионов — Philip == Philip

Проделав десять миллионов операций, система выдает ошибку, так как пользователь с таким логином существует.

Чтобы упростить поиск и уменьшить количество операций, можно использовать абстрактный тип данных — двоичное дерево.

Коснемся темы алгоритмов. Они обеспечивают выполнение поиска, вставки, удаления и других задач со структурами данных.

Как правило, конкретная структура данных выбирается в зависимости от задачи. В таблице — основные характеристики структур данных:

Структура	Достоинства	Недостатки
Массив	Быстрая вставка, быстрый доступ	Медленный поиск, медленное удаление, фиксированный размер
Упорядоченный массив	Поиск выполняется быстрее, чем в обычном массиве	Медленная вставка и удаление, фиксированный размер
Стек	Быстрый доступ в порядке «последним пришел — первым вышел»	Медленный доступ к другим элементам
Очередь	Быстрый доступ в порядке «первым пришел — первым вышел»	Медленный доступ к другим элементам
Связанный список	Быстрая вставка, быстрое удаление	Медленный поиск
Двоичное дерево	Быстрый поиск, вставка, удаление	Сложный алгоритм удаления
Граф	Моделирование реальных ситуаций	Некоторые алгоритмы сложны и медленны
Хеш-таблица	Очень быстрый доступ (если ключ известен)	Медленный доступ (если ключ неизвестен), память используется неэффективно

Как определить эффективность структуры данных

Определить точное время выполнения конкретного алгоритма и структуры данных очень сложно. Существует множество вычислительных устройств, которые обладают разной вычислительной мощностью. Например, алгоритмы поиска можно выполнять как на мобильных устройствах, так и на суперкомпьютерах, которые используют методы параллельной обработки данных. Время, затраченное на поиск, будет разное. Как оценивать эффективность?

Вернемся к примеру с поиском и сравнением логинов пользователя для регистрации в социальной сети.

В первом случае мы использовали линейный поиск, во втором (который касается деревьев) — бинарный. О поисках поговорим во втором уроке, а сейчас поверим на слово и посмотрим на рисунок ниже:

Если список пользователей состоит из 100 элементов, на поиск может потребоваться 100 операций сравнения. Если из 4000000000 элементов — 4000000000 операций. Такое время называется линейным.

Если использовать бинарный поиск (в отсортированном массиве), то на обнаружение пользователя в списке из 100 элементов может потребоваться 7 операций сравнения, а на поиск логина в списке из 4000000000 элементов — всего 32 операции. Это логарифмическое время.

На рисунке линейное и логарифмическое время представлено в специальной нотации — О-большом.

Используя О-большое, мы абстрагируемся от аппаратного обеспечения — будь то суперкомпьютеры или домашние ПК. Не учитываем время, затрачиваемое на выполнение одной операции, а считаем количество операций, которые необходимо произвести для получения результата.

Записывая эффективность через О-большое, можно сказать, что «**O**» — это функция, которая учитывает время выполнения алгоритма на разных аппаратных платформах, а «**n**» — это количество операций.

Но почему мы говорим, что линейный поиск выполняется за **O(n)**? Ведь алгоритм, который ищет уже зарегистрированных пользователей, может найти необходимое значение в первом элементе. Тогда время выполнения будет **O(1)**.

Потому что О-большое определяет время выполнения в худшем случае — когда необходимое значение находится в конце списка.

Распространенные разновидности О-большого, которые часто будут встречаться в курсе:

- **O(log n)** — бинарный поиск;
- **O(n)** — линейный поиск;
- **O(n* log n)** — быстрая сортировка;
- **O(n^2)** — медленный алгоритм сортировки (пузырьковая);
- **O(n!) —** очень медленный алгоритм. Например, задача о коммивояжере.

Практическое задание

1. Прочитать в книге *Grokking Algorithms: An illustrated guide for programmers and other curious people* про алгоритмы и О-большое.

Дополнительные материалы

1. Тут и тут можно ознакомится с абстрактными типами данных.
2. Grokking Algorithms: An illustrated guide for programmers and other curious people.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Лафоре Р. Структуры данных и алгоритмы Java. Классика Computers Science. 2-е изд. —СПб.: Питер, 2013. — 121-178 сс.