GeekBrains

Лекция 2. Структуры данных. Массивы. Алгоритмы массивов.

Оглавление

Цель лекции:

- Узнать, что такое структуры данных
- Вспомнить, что такое массивы
- Разобрать различные операции с массивами на основе различных алгоритмов

План лекции:

- Структуры данных. Массивы
- Простые алгоритмы сортировки массивов: пузырьком, выбором, вставками
- Поиск в неотсортированном массиве (перебор), отсортированном массиве (бинарный поиск). Оценка сложности.
- Продвинутые алгоритмы сортировки: Быстрая сортировка, пирамидальная сортировка

Структуры данных

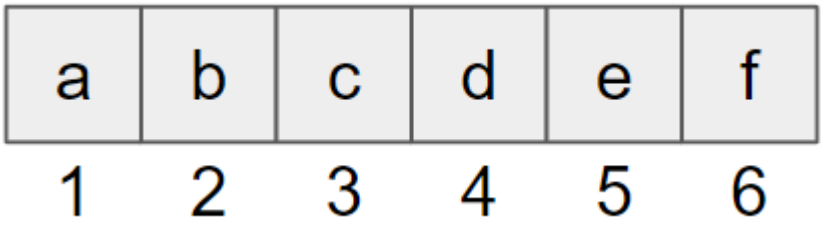
Структурами данных называют некоторый контейнер с данными, обладающий специфическим внутренним устройством (макетом) и логикой хранения. Различные макеты могут быть эффективны для некоторых операций и неэффективны для других.

Каждый разработчик в начале своего обучения, при знакомстве с языком программирования, начинает с операций над простейшими типами данных – числами и строками. Операции над такими данными простые, логичные и полностью покрывают потребности в решении базовых задач. Но со временем сложность задач растет и у разработчика появляется необходимость в объединении этих простых структур в более сложные для упрощения процесса разработки и логики решения задач. Все современные языки программирования имеют из коробки работать с различными типами данных, которые можно применять по мере необходимости. При этом нельзя сказать, что без них совсем нельзя обойтись – под всеми этими структурами всегда лежат самые обычные числа и строки – но вот написание кода без их использования для сколько-нибудь сложного проекта становится непосильной задачей. Всегда стоит помнить, что для каждой задачи существуют свои наиболее подходящие инструменты для решения. Базово, вы можете написать систему любой сложности на C или Assembler, но затраченные ресурсы на создание и поддержку такой системы будут просто огромными в сравнении с системами, написанными на современных языках программирования и манипулирующими различными сложными структурами данных. Для разработчика очень важно понимать внутреннюю структуру различных структур данных для эффективного применения их в процессе решения поставленных задач. У всех структур данных есть как сильные, так и слабые стороны и это обязательно нужно учитывать, когда речь идет про выбор подходящего пути решения. О них поговорим далее.

Массивы

Самой распространенной и простой для понимания структурой данных является массив.

Массив - это контейнер, хранящий данные идентифицируемые по индексу. К любому элементу массива всегда можно обратиться по его индексу и достать или заменить его.
В зависимости от языка программирования, массив может иметь как статичную длину (Java), так и динамическую (JavaScript), но основной особенность работы с массивами является скорость доступа к элементам – т.к. каждый из них имеет один конкретный индекс, программа всегда знает где в памяти лежит конкретный индекс за счет чего скорость доступа к ним составляет **O(1)**. Таким образом, если вы знаете индекс требуемого элемента, независимо от того, сколько всего элементов лежит в массиве, процесс его получения будет занимать константное одинаковое время.



Наиболее популярными задачи в отношении массивов можно назвать поиск по массиву и сортировку. Выполнять эти операции можно с помощью различных алгоритмов, некоторые из них мы сегодня разберем.

Простые алгоритмы сортировки

Сортировка пузырьком

Один из наиболее простых алгоритмов сортировки массива – пузырьковая сортировка. Нагляднее всего работу этого алгоритма можно продемонстрировать в вертикальном массиве – наиболее легкие элементы стремятся вверх, словно пузырьки воздуха в жидкости.
<-Слайд с анимацией ->
Базовый алгоритм предполагает, что каждый элемент необходимо сравнить с соседним и, если правый элемент меньше левого, то их меняют местами. Алгоритм повторяется до тех пор, пока все элементы в массиве не выстроятся в нужном порядке.

```java
public static void sort(int[] array) {
        boolean needSort;
        do {
            needSort = false;
            for (int i = 0; i < array.length - 1; i++) {
                if (array[i] > array[i + 1]) {
                    int temp = array[i];
                    array[i] = array[i + 1];
                    array[i + 1] = temp;
                    needSort = true;
                }
            }
        } while (needSort);
    }
```

Как можно понять из примера, количество шагов для сортировки будет отличаться в зависимости от того, в каком порядке изначально стоят элементы массива. Если массив уже отсортирован, то достаточно пройти по массиву 1 раз, чтобы убедиться в этом. При этом никаких операций перестановки элементов выполнено не будет. В такой ситуации сложность алгоритма будет **O(n)**. В тоже время, если самый

маленький элемент находится в самом конце, то количество проходов по массиву размером **n** будет так же достигать **n**, что дает нам сложность **O(n^2)**. Этот пример явно иллюстрирует, почему при оценке алгоритма не вычисляют реальное количество необходимых шагов, для получения результата. Даже один и тот же алгоритм может давать различное количество шагов для входных данных одного размера. Это подводит нас к еще одной особенности оценки сложности алгоритма. Один и тот же алгоритм может давать различные результаты для разных входящих данных, поэтому для оценки некоторых алгоритмов используются понятия максимальной (предельной) сложности и ожидаемой сложности.

Максимальная сложность – количество шагов для обработки наиболее неблагоприятного состояния входных данных.

Ожидаемая сложность – это вариант, который будет релевантен для большей части возможных кейсов.

Для некоторых алгоритмов ожидаемая сложность будет совпадать с максимальной, в каких-то – нет. В приведенном выше алгоритме пузырьковой сортировки максимальной сложностью вычисления будет **O(n^2)** и эта же сложность является ожидаемой для этого алгоритма – для данного алгоритма очень небольшое количество кейсов дает сложность ниже указанной, а значит в большинстве случаев она будет стремиться именно к максимальной.

Сортировка выбором

Так же очень простой алгоритм сортировки, который предполагает поиск наименьшего (или наибольшего) значения правее от сравниваемого элемента. В случае, если такой элемент найден – происходит перестановка с начальным элементом.

<-Вставить анимацию сортировки->

Данный алгоритм очень похож на пузырьковую сортировку, за тем исключением, что для его записи удобнее использовать не цикл while, а 2 циклов for, вложенные друг в друга.

```java
public static void sort(int[] array) {
        for (int i = 0; i < array.length; i++) {
            int minPosition = i;
            for (int j = i + 1; j < array.length; j++) {
                if (array[j] < array[minPosition]) {
                    minPosition = j;
                }
            }
            if (minPosition != i) {
                int temp = array[i];
                array[i] = array[minPosition];
                array[minPosition] = temp;
            }
        }
    }
```

Это наглядно приводит нас так же к сложности **O(n^2)**. В данном примере это несколько нагляднее – внутри одного алгоритма со сложностью **O(n)** вызывается еще один алгоритм **O(n)**, что по правилам перемножения даст как раз сложность **O(n^2)**. По факту, подобный подход к сортировке уменьшает количество реальных операций перестановки, в сравнении с сортировкой пузырьком, но общее количество сравнений будет точно таким же.

Сортировка вставками

Так же, как нечто среднее, между сортировкой пузырьком и выбором, можно выделить сортировку вставками. Принцип работы у нее точно такой же, как у предыдущей, только после сравнения двух элементов мы не запоминаем индекс наименьшего (наибольшего) из элементов, а сразу производим перестановку.
<-анимация алгоритма->

```java
public static void sort(int[] array){
        for (int i = 0; i < array.length; i++) {
            for (int j = i + 1; j < array.length; j++) {
                if (array[j] < array[i]) {
                    int temp = array[i];
                    array[i] = array[j];
                    array[j] = temp;
                }
            }
        }
    }
```

Точно так же, как и обе предыдущие сортировки, сложность данного алгоритма равна **O(n²)**. Как мы видим, все 3 рассмотренных алгоритма совершенно идентичны с точки зрения сложности. А также мы можем сказать, что использовать алгоритмы подобной сложности нельзя на данных большого размера, т.к. скорость выполнения алгоритма будет очень медленной. Попробуйте взять любой из рассмотренных алгоритмов и отсортировать массив, в котором 1 000 000 элементов. В среднем, пузырьковая сортировка может занять до получаса, в зависимости от мощности вашего компьютера, что явно не подходит для использования в массивах подобного размера.

Алгоритмы поиска

Поиск по массиву также относится к одной из самых базовых операций, с которыми можно столкнуться. В целом, код поиска знаком большинству начинающих разработчиков, т.к. именно такие алгоритмы идут в виде учебных задач при знакомстве с этой структурой данных. Самый просто в реализации и понимании способ – это перебор всего массива до тех пор, пока не встретится искомый элемент.

```java
    public static int findIndex(int value, int[] array) {
        for (int i = 0; i < array.length; i++) {
            if (array[i] == value) {
                return i;
            }
        }
        return -1;
    }
```

Как несложно понять, благодаря использованию цикла for, мы получаем классическую сложность обхода массива - **O(n)**. Это неплохая сложность, но при работе с большими массивами и при регулярном поиске может стать проблемой. Можно ли найти что-то в массиве быстрее? В массиве, где данные расположены в случайном порядке, создать какой-либо универсальный алгоритм, который бы выдавал устойчивый результат, превышающий эффективность классического перебора не представляется реальным. Да, возможно какой-либо конкретный случай может содержать определенные закономерности, которые можно использовать для этого, но универсальным такой алгоритм не станет. В таком

вопросе гораздо эффективнее можно ориентироваться по отсортированному массиву данных. В нем сложность поиска можно существенно снизить применив, например, алгоритм бинарного поиска.

💡 **Бинарный поиск** - тип поискового **алгоритма**, который последовательно делит пополам заранее отсортированный массив данных, чтобы обнаружить нужный элемент. Другие его названия — **двоичный поиск**, **метод** половинного деления, дихотомия. Принцип работы **алгоритма бинарного поиска**. Основная последовательность действий **алгоритма** выглядит так: Сортируем массив данных. Делим его пополам и находим середину.

Этот алгоритм использует сортировку массива для пропуска большей части данных при поиске. Бинарный поиск начинается с середины массива, где сразу получает данные в какой части массива может находится искомый элемент – если центральный элемент массива меньше искомого – значит искомый в правой части массива. Если больше – значит в левой. Далее применяется аналогичная проверка для выбранной половины данных, снова через сравнение центрального элемента отрезка.
<-анимация алгоритма->
Таким образом, при поиске элемента количество операций сравнения будет существенно меньше, чем в операции поиска перебором. Более того, т.к. мы оперирует центральным элементом отрезка, чтобы сделать всего на 1 шаг больше нам необходимо увеличить количество элементов самого массива вдвое, чтобы центральный элемент массива оказался центральным для правой или левой части массива. Это яркий пример логарифмической сложности алгоритма O(log n). Таким образом, один раз потратив время на сортировку данных мы можем во много раз сократить временные затраты на многократный поиск в дальнейшем.
Благодаря особенности данного алгоритма его очень просто записать с помощью рекурсии – операция сравнения абсолютно идентичная для любого из вложенных шагов.

```
public static int search(int value, int[] array, int min, int max) {
        int midpoint;
        if (max < min) {
            return -1;
        } else {
            midpoint = (max - min) / 2 + min;
        }

        if (array[midpoint] < value) {
            return search(value, array, midpoint + 1, max);
        } else {
            if (array[midpoint] > value) {
                return search(value, array, min, midpoint - 1);
            } else {
                return midpoint;
            }
        }
    }
```

Как мы видим, функция принимает в себя границы массива, в пределах которого необходимо осуществлять поиск. Таким образом, каждый рекурсивный шаг уменьшает диапазон, в котором необходимо произвести поиск. При этом сам алгоритм не меняется.

Сложность этого алгоритма – **O(log n)**. И вложенная логика, которая требует сравнить элемент с искомым и определить куда двигаться дальше – вправо или влево. Данный алгоритм имеет сложность **O(1)**, т.к. эта операция никак не зависит от размера массива и, по сути, оперирует всегда с 1 значением. Перемножение сложностей дает нам **O(1 * log n)** => **O(log n)**. Подход, используемый бинарным поиском, разделяющий объем данных пополам на каждом шаге, называется «разделяй и властвуй». Он позволяет создавать алгоритмы со сложностями **O(m * log n)**, где **m** может быть как константой (непосредственно бинарный поиск), так и непосредственно **n**. Зачастую такие алгоритмы сложнее, но гораздо выгоднее с точки зрения производительности, т.к. дают сложность ощутимо меньше, чем **O(n^2)**. Это подводит нас к алгоритмам сортировки, основанным на подобном принципе.

Продвинутые алгоритмы сортировки

Быстрая сортировка (quicksort)

Чаще всего, когда мы используем сортировку, уже реализованную в штатных средствах языка программирования или библиотеки, под капотом мы встретим именно быструю сортировку. Суть быстрой сортировки – разделить массив на 2 части таким образом, чтобы справа все числа были больше, чем слева, при этом их порядок относительно друг друга не важен. Это позволит не сравнивать элементы справа с элементами слева больше 1 раза, как раз для достижения их разделения на 2 части. И далее этот же подход будет применяться для каждой из получившихся частей, равно как предусматривает принцип «разделяй и властвуй». При этом, в отличии от бинарного поиска, количество операций в момент разделения не константное, а линейное – необходимо сравнить все элементы правой и левой части с неким эталоном и при необходимо – поменять их местами. В данном алгоритме такой элемент называется пивотом.

> 💡 **Пивот** - от английского поворот (pivot).

При этом нет единого эталонного алгоритма выбора пивота. Это может быть абсолютно любой элемент массива – средний, крайний, случайный и т.д.
<-анимация квиксорта->
Берутся 2 маркера, стоящие в 2 крайних позициях и начинаются двигаться друг к другу, сравнивая элементы с пивотом. Как только левый маркер находит элемент больше пивота, он останавливается до тех пор, пока правый маркер не встретит элемент меньше пивота. Это сигнал к тому, что данные элементы необходимо поменять местами, чтобы соблюсти условие – все элементы левее пивота меньше либо равны ему, а справа – больше либо равны. Маркеры продолжаются двигаться друг к другу, пока не пересекутся. Точка пересечения в итоге будет тот самый пивот, который при этом может сменить свое положение. После этого алгоритм рекурсивно запустится для левой и правой части относительно пивота и будет продолжаться до тех пор, пока весь массив не будет отсортирован.

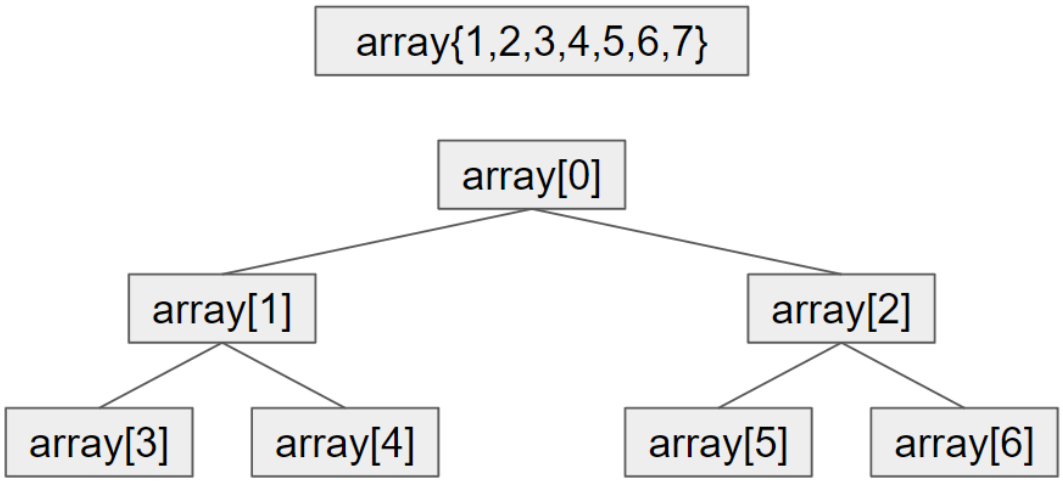
```java
public static void sort(int[] array, int startPosition, int endPosition) {
        int leftPosition = startPosition;
        int rightPosition = endPosition;
        int pivot = array[(startPosition + endPosition) / 2];
        do {
            while (array[leftPosition] < pivot) {
                leftPosition++;
            }
            while (array[rightPosition] > pivot) {
                rightPosition--;
            }
            if (leftPosition <= rightPosition) {
                if (leftPosition < rightPosition) {
                    int temp = array[leftPosition];
                    array[leftPosition] = array[rightPosition];
                    array[rightPosition] = temp;
                }
                leftPosition++;
                rightPosition--;
            }

        } while (leftPosition <= rightPosition);

        if (leftPosition < endPosition) {
            sort(array, leftPosition, endPosition);
        }
        if (startPosition < rightPosition) {
            sort(array, startPosition, rightPosition);
        }
    }
```

Если сравнивать данный алгоритм с теми, что были рассмотрены ранее, можно увидеть, что количество сравнений элементов друг с другом существенно снижено. Мы так же имеет два вложенных алгоритма. В лучшем из вариантов мы каждый раз будет делить массив ровно пополам (плюс-минус 1 элемент), что дает максимальную глубину рекурсии log n. И проводить сравнение элементов друг с другом в пределах получившихся массивов (**O(n)**), тем самым получим сложность **O(n log n)**. В худшем – пивот всегда будет оказываться крайним элементом при разбиении коллекции (пивот является максимальным или минимальным значением рассматриваемого массива), а значит максимальная глубина рекурсии будет равна n-1, что даст сложность O(n^2). При этом, эффективной сложностью данной сортировки принято считать именно **O(n log n)**, т.к. шанс попадания пивота крайним элементом, особенно на большом объеме данных, очень невелик. Это легко проверить, запустив сортировку того же массива в 1 000 000 элементов, на котором экспериментировали с пузырьковой сортировкой. Процесс займет не больше нескольких секунд.

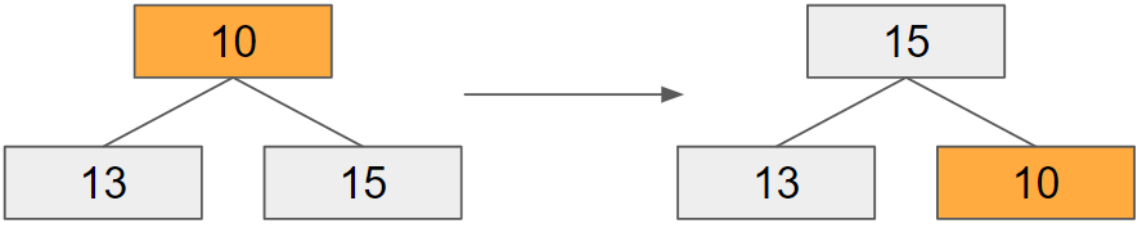
Сортировка кучей (пирамидальная)

Особенность данной сортировки в использовании дополнительной структуры данных называемой бинарной кучей (пирамидой).

Бинарная куча представляет из себя древовидную структуру, когда у каждого объекта может быть до 2 детей. При этом строится из массива она предельно просто – первый элемент массива является корнем, 2 и 3 его детьми, 4 и 5 детьми элемента 2 и т.д. пока в массиве остаются элементы.



Благодаря такому подходу к составлению бинарной кучи она получает следующее свойство: если принять элемент с индексом *i* за родителя, то индексы его дочерних элементов будут *2 * i + 1* и *2 * i + 2*.
Таким образом, несмотря на то, что сортировка использует для своего алгоритма бинарную кучу, в реальности строить никакую отдельную структуру данных не нужно, т.к. мы в любой момент можем определить детей для каждого из элементов и проводить их сравнение или обмен.
Общая идея сортировки пирамидой заключается в том, что сравнение элементов происходит не между всеми элементами массива, а только в пределах построенной пирамидальной структуры, т.е. родителя и его детей. Такая операция называется ***«просеиванием»***, когда интересующий нас узел кучи сравнивается со своими двумя детьми и меняется местами с тем, что больше родителя. Если оба ребенка больше родителя – обмен происходит с наибольшим из детей.



Дальше необходимо определить алгоритм, в каком порядке необходимо проводить операции просеивания. Для этого весь процесс пирамидальной сортировки делится на 2 этапа.
Первый этап – это подготовка кучи. Определяем правую часть кучи по формуле **n/2-1**, где n – длина массива. Начиная с указанного индекса, мы начинаем операции просеивания в цикле до тех пор, пока не придем к началу массива. В результате этой операции самый большой элемент нашего массива окажется в индексе 0, что является вершиной пирамиды.
Второй этап – начинается с того, что первый и последний элемент массива меняется местами, тем самым наибольший элемент оказывается в конце массива, а текущая вершина (индекс 0) начинает операцию просеивания по пирамиде с

размером n-1, в результате чего снова наибольший элемент займет 0 индекс нашего массива. Меняем его местами с предпоследним элементом массива (последний мы уже определи) и повторяем операцию. Это происходит до тех пор, пока все элементы массива не займут свое место, а размер пирамиды для просеивания не уменьшится до 0.
<-Вставить анимацию->

```
    public static void sort(int[] array) {
        // Построение кучи (перегруппируем массив)
        for (int i = array.length / 2 - 1; i >= 0; i--)
            heapify(array, array.length, i);

        // Один за другим извлекаем элементы из кучи
        for (int i = array.length - 1; i >= 0; i--) {
            // Перемещаем текущий корень в конец
            int temp = array[0];
            array[0] = array[i];
            array[i] = temp;

            // Вызываем процедуру heapify на уменьшенной куче
            heapify(array, i, 0);
        }
    }

    private static void heapify(int[] array, int heapSize, int rootIndex) {
        int largest = rootIndex; // Инициализируем наибольший элемент как корень
        int leftChild = 2 * rootIndex + 1; // левый = 2*rootIndex + 1
        int rightChild = 2 * rootIndex + 2; // правый = 2*rootIndex + 2

        // Если левый дочерний элемент больше корня
        if (leftChild < heapSize && array[leftChild] > array[largest])
            largest = leftChild;

        // Если правый дочерний элемент больше, чем самый большой элемент на данный момент
        if (rightChild < heapSize && array[rightChild] > array[largest])
            largest = rightChild;
        // Если самый большой элемент не корень
        if (largest != rootIndex) {
            int temp = array[rootIndex];
            array[rootIndex] = array[largest];
            array[largest] = temp;

            // Рекурсивно преобразуем в двоичную кучу затронутое поддерево
            heapify(array, heapSize, largest);
        }
    }
```

В данном случае мы имеем обратную, в отличии от быстрой сортировки, зависимость на сложность алгоритма – внешний цикл содержит **O(n)** шагов, а вложенный работает по уже знакомому нам принципу **O(log n)**, т.к. количество операций соответствует обходу вложенных элементов, что суммарно дает нам сложность **O(n log n)**, как и у быстрой сортировки.

Итоги

Сегодня мы познакомились с понятием структуры данных, рассмотрели 2 из них – массив и бинарную кучу. Разобрали несколько алгоритмов сортировки и поиска и определили их сложность. Оценка сложности тех или иных решений критично важна для решения рабочих задач. Умение определять сложность решения и понимать его сильные и слабые стороны один из ключевых навыков любого разработчика.

Дополнительные материалы

https://prog-cpp.ru/sort-pyramid/
https://habr.com/ru/company/otus/blog/460087/
https://habr.com/ru/post/415935/