



Урок 2

Массивы и сортировка

Работа с массивами и способы их сортировки.

[Введение](#)

[Создание массива](#)

[Обращение к элементам массива](#)

[Инициализация](#)

[Вывод элементов массива](#)

[Удаление элемента массива](#)

[Линейный и двоичный поиск](#)

[Сложность алгоритмов](#)

[Сортировка](#)

[Пузырьковая сортировка](#)

[Сортировка методом выбора](#)

[Сортировка методом вставки](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Введение

Массив — одна из популярных структур данных, поддерживается многими языками программирования. Как правило, изучение структур данных начинается с массивов, так как более сложные структуры содержат их.

Массив в Java — это структура данных, которая содержит элементы одного типа, расположенные друг за другом в памяти компьютера. Массив в Java является ссылочным типом данных, а доступ к его элементам осуществляется по индексу. Размерность массива, или его длина, определяется как количество входящих в него элементов. В Java индексация массива начинается с индекса 0.



Элементы массива

В этом уроке рассмотрим работу с массивами в Java: их создание, вставку и поиск элементов. Изучим одну из разновидностей — упорядоченный массив, а также разберемся со способами сортировки элементов.

Создание массива

В Java массив является ссылочным типом данных, то есть относится к объектам. Поэтому для его создания используют оператор **new**.

```
public class Main {  
    public static void main(String[] args) {  
        int[] myArr;           // Определение ссылки на массив  
        myArr = new int[10];   // Создание массива и сохранение ссылки на него  
    }  
}
```

В переменной **myArr** хранится ссылка на блок памяти, в которой содержатся данные массива. В большинстве случаев определение ссылки и создание массива объединяются в одно действие. Это позволяет сократить код программы.

```
public class Main {
    public static void main(String[] args) {
        int[] myArr = new int[10]; // Создание массива и сохранение ссылки на
        него
    }
}
```

Использование оператора `[]` говорит о том, что тип переменной принадлежит объекту массива. Есть альтернативный синтаксис создания массива: оператор `[]` ставится после имени переменной, а не после типа данных.

```
public class Main {
    public static void main(String[] args) {
        int myArr[] = new int[10]; // Создание массива и сохранение ссылки на
        него
    }
}
```

Для определения размера массива используется **метод** `length`.

```
public class Main {
    public static void main(String[] args) {
        int[] myArr = myArr = new int[10];
        int len = myArr.length;
    }
}
```

Массивы в Java имеют фиксированный размер, и изменить его после создания массива нельзя.

Обращение к элементам массива

Для обращения к элементу массива после ссылки в квадратных скобках указывается его индекс. Такой синтаксис используется во многих языках программирования.

```
public class Main {
    public static void main(String[] args) {
        int[] myArr = myArr = new int[10];
        int x = myArr[2];
    }
}
```

В Java индексация массива начинается с 0. Если выйти за пределы размерности массива, будет сгенерировано исключение **`IndexOutOfBoundsException`**.

Инициализация

Инициализация массива совпадает с инициализацией примитивных и ссылочных типов данных. Например, для массива типа **`int`** значение элементов по умолчанию будет равным 0. Если массив имеет ссылочный тип элементов, по умолчанию его элементы заполняются специальными объектами **`null`**. Для примера рассмотрим создание и инициализацию двух массивов, один из которых содержит ссылочные элементы, а второй — элементы примитивного типа данных.

```
public class Main {
    public static void main(String[] args) {
```

```

        int[] myArr1 = new int[10];
        People[] myArr2 = new People[10];
        System.out.println(myArr1[1]);
        System.out.println(myArr2[1]);
    }
}

class People {
    String name;
}

```

Первый вывод в консоль даст результат 0, а второй **null**.

Объявление, создание и заполнение массива можно объединить в одну команду, используя следующий синтаксис:

```

public class Main {
    public static void main(String[] args) {
        int[] myArr1 = {1, 2, 3, 4, 5, 2};
        System.out.println(myArr1[1]);
    }
}

```

В таком случае размер массива определяется количеством значений, указанных в фигурных скобках.

Вывод элементов массива

Для вывода в консоль всех элементов массива используется оператор **for** и метод **println**.

```

public class Main {
    public static void main(String[] args) {
        int[] myArr1 = {1, 2, 3, 4, 5, 2};
        for (int i = 0; i < myArr1.length; i++) {
            System.out.println(myArr1[i]);
        }
    }
}

```

Удаление элемента массива

Так как размерность массива изменить нельзя, то и элемент массива явно удалить невозможно. Можно сдвинуть массив на одно значение влево. Допустим, необходимо удалить элемент массива со значением 3. Первое, что следует сделать, — найти его перебором в цикле и сохранить его индекс. Дальше в новом цикле, начиная с найденного индекса, присвоить новые значения остальным элементам массива.

```

public class Main {
    public static void main(String[] args) {
        int[] myArr1 = {1, 2, 3, 4, 5, 2};
        int i;
        int len = myArr1.length;
        int search = 3;

        System.out.println("Выводим массив");
        for (int j = 0; j < len; j++) {
            System.out.println(myArr1[j]);
        }
    }
}

```

```

    }

    System.out.println("Поиск искомого элемента");
    for (i = 0; i < len; i++) {
        if (myArr1[i] == search) {
            break;
        }
    }

    System.out.println("Сдвиг всех элементов на 1 шаг влево");
    for (int j = i; j < len - 1; j++) {
        myArr1[j] = myArr1[j + 1];
    }
    len--;

    System.out.println("Вывод массива с удаленным элементом");
    for (int j = 0; j < len; j++) {
        System.out.println(myArr1[j]);
    }
}
}

```

При запуске программы создается массив **myArr1**, который содержит 6 элементов типа **int**. Объявляется переменная **i**, в которую будет помещен индекс искомого значения. Объявляется и инициализируется переменная **len**, содержащая количество элементов массива. В переменной **search** указывается искомое значение, которое будет удалено. После вывода элементов массива в консоль, циклом пробегаясь по нему, ищем необходимый элемент. Если он найден, останавливаем цикл. Таким образом в переменной **i** будет находиться позиция искомого элемента. Далее в новом цикле осуществляем сдвиг влево на один элемент и устанавливаем значение переменной **len** на единицу меньше.

Результат работы программы:

```

Выводим массив
1
2
3
4
5
2
Поиск искомого элемента
Сдвиг всех элементов на 1 шаг влево
Вывод массива с удаленным элементом
1
2
3
5
2

```

В программировании считается плохим тоном писать программу в одном файле. Отделим логику, которая касается работы с массивом, от основной программы. Создадим класс, который будет содержать методы создания массива, поиска элемента и удаления найденного элемента.

```
class MyArr{
    private int[] arr;
    private int size;

    public MyArr(int size){
        this.size = size;
        this.arr = new int[size];
    }

    public int getElement(int index){
        return this.arr[index];
    }

    public void setElement(int index, int elem){
        this.arr[index] = elem;
    }

    public int[] deleteElement(int elem){
        int i=0;
        for(i=0;i<this.size;i++){
            if (this.arr[i] == elem) break;
        }

        for (int j=i;j<this.size-1; j++){
            this.arr[j] = this.arr[j+1];
        }
        this.size--;

        return this.arr;
    }

    public int getSize(){
        return this.size;
    }
}

public class Main {
    public static void main(String[] args) {

        MyArr arr = new MyArr(10);
        arr.setElement(0, 5);
        arr.setElement(1, 5);
        arr.setElement(2, 5);
        arr.setElement(3, 5);
        arr.setElement(4, 5);
        arr.setElement(5, 5);
        arr.setElement(6, 1);
        arr.setElement(7, 5);
        arr.setElement(8, 5);
        arr.setElement(9, 5);

        System.out.println("Выводим массив");
        for(int j=0; j<arr.getSize(); j++){
            System.out.println(arr.getElement(j));
        }
    }
}
```

```

    }

    arr.deleteElement(1);

    System.out.println("Выводим новый массив");
    for(int j=0; j<arr.getSize(); j++){
        System.out.println(arr.getElement(j));
    }
}
}

```

Таким образом мы отделили структуру данных от остального кода программы. Теперь в методе **main** используем методы, реализованные в классе **MyArr**. Но и такая структура класса не идеальна. По сути, методы **setElement** и **getElement** делают то же, что и оператор `[]`. Также пользователь применяет операторы цикла для вывода информации о массиве. Сделаем небольшой рефакторинг и уменьшим количества кода в методе **main**. В классе **MyArr** создадим три метода: **insert**, **delete** и **display**.


```

class MyArr {
    private int[] arr;
    private int size;

    public MyArr(int size){
        this.size = 0;
        this.arr = new int[size];
    }

    public void display(){
        for(int i = 0; i < this.size; i++) {
            System.out.println(this.arr[i]);
        }
    }

    public void delete(int value){
        int i = 0;
        for(i = 0; i < this.size; i++) {
            if (this.arr[i] == value) break;
        }

        for (int j = i; j < this.size - 1; j++) {
            this.arr[j] = this.arr[j + 1];
        }
        this.size--;
    }

    public void insert(int value){
        this.arr[this.size] = value;
        this.size++;
    }
}

public class Main {
    public static void main(String[] args) {
        MyArr arr = new MyArr(10);
        arr.insert(5);
        arr.insert(1);
        arr.insert(2);
        arr.insert(5);
        arr.insert(4);
        arr.insert(5);
        arr.insert(6);
        arr.insert(5);
        arr.insert(8);
        arr.insert(9);

        System.out.println("Выводим массив");
        arr.display();
        arr.delete(1);
        System.out.println("Выводим новый массив");
        arr.display();
    }
}

```

В примере поиск удаляемого элемента находится внутри метода **delete()**. Алгоритм поиска выглядит не идеально, так как не учитывает тот факт, что искомый элемент может быть не найден.

Линейный и двоичный поиск

Рассмотрим два вида поиска: линейный и двоичный. Создадим метод **find**, который будет выполнять линейный поиск. Он осуществляется простым сравнением очередного элемента в массиве с искомым значением. Если значения совпадают, поиск считается завершенным.

Пример реализации линейного поиска в классе **MyArr**:

```
public boolean find(int value) {  
    for (int i = 0; i < this.size; i++) {  
        if (this.arr[i] == value) {  
            return true;  
        }  
    }  
    return false;  
}
```

Метод **find** осуществляет поочередное сравнение всех элементов массива с искомым значением и, если элемент найден, возвращает логическое значение **true**. Если был достигнут конец массива и элемент не был обнаружен, возвращается логическое значение **false**.

Добавим в реализацию метода **main** проверку на существование удаляемого элемента:

```
public static void main(String[] args) {
    MyArr arr = new MyArr(10);
    arr.insert(5);
    arr.insert(1);
    arr.insert(2);
    arr.insert(5);
    arr.insert(4);
    arr.insert(5);
    arr.insert(6);
    arr.insert(5);
    arr.insert(8);
    arr.insert(9);

    int search = 8;

    System.out.println("Выводим исходный массив");
    arr.display();

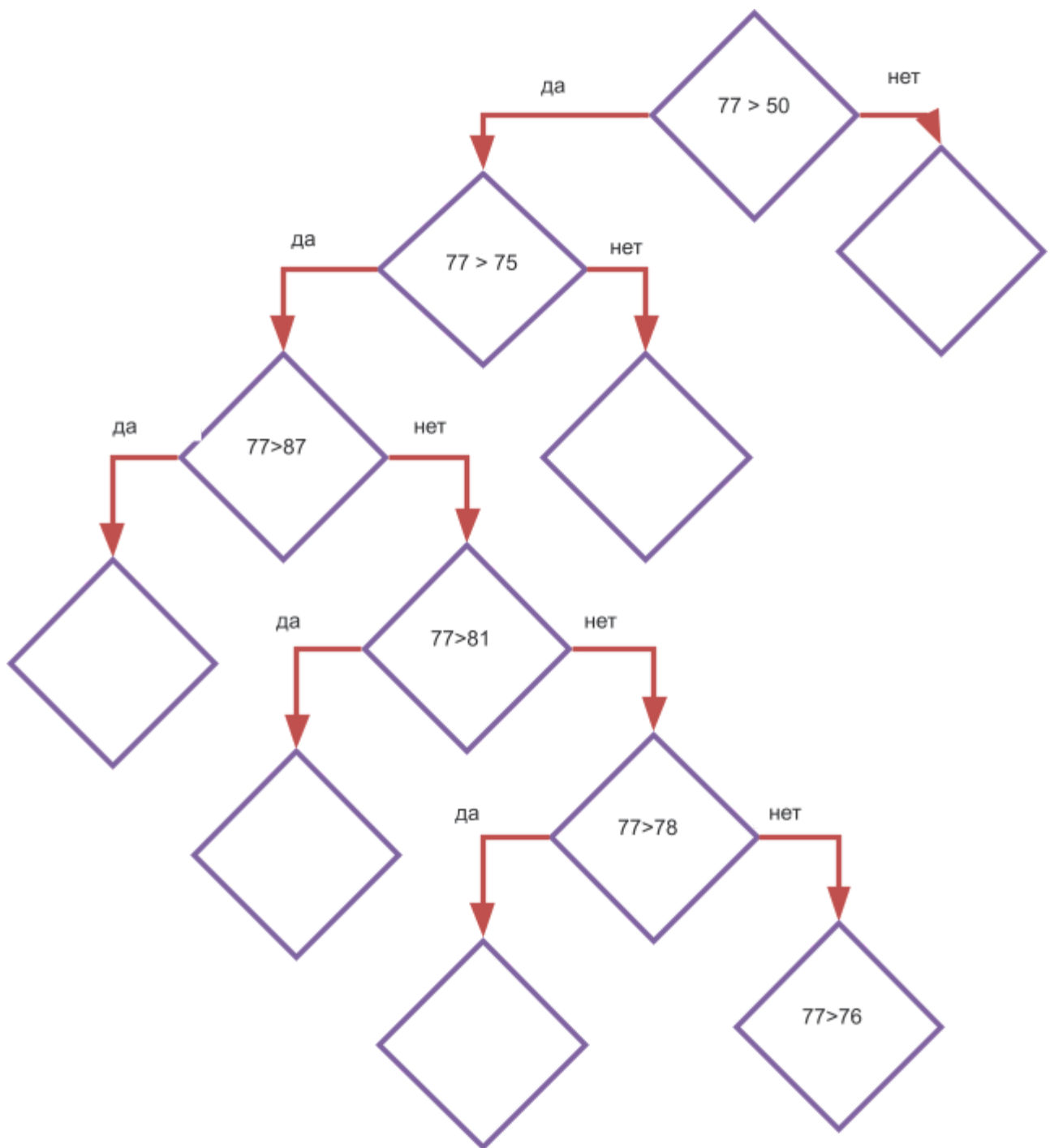
    if (arr.find(search)) {
        arr.delete(search);
        System.err.println("Элемент " + search + " удален");
        System.out.println("Выводим новый массив");
        arr.display();
    } else {
        System.out.println("Не удалось найти элемент " + search);
    }
}
```

Теперь если искомый элемент не будет найден, программа сообщит об этом.

Двоичный поиск — это алгоритм поиска элемента в отсортированном массиве. Его можно сравнить с игрой «Угадай число», когда один участник загадывает число от 1 до 100, а другой пытается его назвать. Чтобы сделать это за меньшее число шагов, следует начинать с 50. Если загаданное число больше — оно находится в диапазоне 51–100, и поиск искомого значения сократится вдвое.

Представим, что мы хотим найти число 77. На первом шаге разделим диапазон значений пополам и сравним число 77 с 50. Число 77 больше 50, поэтому продолжим поиск от 51 до 100. Каждое следующее предположение будет сокращать диапазон искомого значения вдвое. Следующее сравнение — с 75. Так как наше число больше, поиск продолжится от 76 до 100. И так далее, пока не найдется число 77.

Если бы мы использовали линейный поиск, для нахождения необходимого элемента потребовалось бы 77 шагов.



Добавим к классу **MyArr** метод **binaryFind**, который выполняет двоичный поиск элемента массива.

Метод **binaryFind**:

```
public boolean binaryFind(int value){
    int low = 0;
    int high = this.size-1;
    int mid;
    while(low <= high){
        mid = (low + high) / 2;
        if (value == this.arr[mid]) {
            return true;
        }
        else {
            if (value < this.arr[mid]){
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }
    }
    return false;
}
```

Сначала определяем переменные, которые хранят индексы верхней и нижней границы поиска. В переменной **mid** будет граница поиска, которая делит наш диапазон пополам. Поиск будет проходить внутри цикла **while**, пока нижняя граница поиска будет меньше верхней. В цикле проверяем соответствие переменной **mid** искомому значению. Если значение не найдено, сравниваем его с переменной **mid**. В случае, когда искомое значение меньше, присваиваем верхней границе значение, которое находится в переменной **mid**. В ином случае присваиваем значение переменной **mid** нижней границе. После этого переходим на новую итерацию цикла — до тех пор, пока значение не будет найдено или переменная **low** будет больше, чем **high**.

Так как массив отсортирован по возрастанию, необходимо модернизировать алгоритм метода вставки элемента. Вставлять значения в конец массива не получится — теперь надо сравнивать вставляемый элемент со всеми в массиве и вставлять его на нужное место, а остальные сдвигать вправо.

Код обновленного метода **insert**:

```
public void insert(int value){
    int i;
    for(i=0;i<this.size;i++){
        if (this.arr[i]>value)
            break;
    }
    for(int j=this.size;j>i;j--){
        this.arr[j] = this.arr[j-1];
    }
    this.arr[i] = value;
    this.size++;
}
```

Полный код класса **MyArr** и класса **Main**:

```
public class MyArr {
    private int[] arr;
    private int size;

    public MyArr(int size) {
        this.size = 0;
        this.arr = new int[size];
    }

    public boolean binaryFind(int value){
        int low = 0;
        int high = this.size - 1;
        int mid;
        while (low <= high) {
            mid = (low + high) / 2;
            if (value == this.arr[mid]) {
                return true;
            } else {
                if (value < this.arr[mid]) {
                    high = mid - 1;
                } else {
                    low = mid + 1;
                }
            }
        }
        return false;
    }

    public boolean find(int value) {
        int i;
        for (i = 0; i < this.size; i++) {
            if (this.arr[i] == value) {
                return true;
            }
        }
        return false;
    }

    public void display() {
        for (int i = 0; i < this.size; i++) {
            System.out.println(this.arr[i]);
        }
    }

    public void delete(int value){
        int i = 0;
        for(i = 0; i < this.size; i++) {
            if (this.arr[i] == value) {
                break;
            }
        }
    }
}
```

```

        for (int j = i; j < this.size - 1; j++){
            this.arr[j] = this.arr[j + 1];
        }
        this.size--;
    }

    public void insert(int value){
        int i;
        for(i = 0; i < this.size; i++) {
            if (this.arr[i]>value) {
                break;
            }
        }
        for (int j = this.size; j > i; j--){
            this.arr[j] = this.arr[j-1];
        }
        this.arr[i] = value;
        this.size++;
    }
}

public class Main {
    public static void main(String[] args) {
        MyArr arr = new MyArr(10);
        arr.insert(-10);
        arr.insert(45);
        arr.insert(26);
        arr.insert(20);
        arr.insert(25);
        arr.insert(40);
        arr.insert(75);
        arr.insert(80);
        arr.insert(82);
        arr.insert(91);

        int search = -10;

        System.out.println(arr.binaryFind(search));
    }
}

```

В результате работы программы метод **binarySearch** вернет логический результат **true** или **false**.

Формула, по которой можно посчитать количество шагов при двоичном поиске:

$$r = 2^s$$

r — размер диапазона, **s** — количество шагов.

Представим, что нужно проводить поиск в диапазоне от 1 до 100. Если **s** = 6, то $2^6 = 64$, и этого недостаточно, чтоб покрыть весь диапазон поиска. А если **s** = 7, то этого хватает с избытком. Поиск искомого значения в диапазоне от 1 до 100 пройдет максимум за 7 шагов.

Разберемся с объектами, ведь до сих пор наш массив содержал примитивные типы данных **int**. Рассмотрим работу с массивом объектов на примере класса **Person**, который содержит информацию об имени и возрасте человека. Для правильной работы класса **myArr** необходимо скорректировать методы. Изменим тип массива с **int** на **Person**. Поиск и удаление теперь осуществляются по ключевому полю **search**, которое является объектным типом **String**. Поэтому для сравнения по этому полю необходимо использовать метод **equals()** вместо **==**. В методе **insert** теперь происходит вставка объекта **Person** вместо переменной типа **int**.

```
class Person{
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

class MyArr{
    private Person[] arr;
    private int size;

    public MyArr(int size){
        this.size = 0;
        this.arr = new Person[size];
    }

    public boolean find(String search){
        int i;
        for(i=0;i<this.size;i++){
            if (this.arr[i].getName().equals(search)) break;
        }
        if (i==this.size)
            return false;
        else
            return true;
    }

    public void display(){
        for(int i=0;i<this.size;i++){
            System.out.println(this.arr[i].getName()+"
"+this.arr[i].getAge());
        }
    }

    public void delete(String search){
        int i=0;
        for(i=0;i<this.size;i++){
            if (this.arr[i].getName().equals(search)) break;
        }
    }
}
```



```

        for (int j=i; j<this.size-1; j++){
            this.arr[j] = this.arr[j+1];
        }
        this.size--;
    }

    public void insert(String name, int age){
        this.arr[this.size] = new Person(name, age);
        this.size++;
    }
}

public class Main {

    public static void main(String[] args) {
        int size = 100;
        MyArr arr = new MyArr(size);
        arr.insert("Vasya", 10);
        arr.insert("Igor", 15);
        arr.display();

        arr.delete("Igor");

        arr.display();
    }
}

```

Сложность алгоритмов

Сложность алгоритма вставки не зависит от количества элементов массива, так как новый элемент всегда вставляется в конец списка. Представим, что для вставки требуется время **k**, которое для любого значения будет одинаковым. Тогда алгоритм вставки будет рассчитываться по такой формуле:

$$T = k$$

T — время вставки в неупорядоченный массив.

Сложность линейного поиска зависит от того, когда встретится искомый элемент. В худшем случае время поиска можно рассчитать по формуле:

$$T = k \times N$$

N — количество элементов массива.

Как и для линейного поиска, можно записать формулу сложности двоичного поиска:

$$T = k \times \log_2 N$$

Время, потраченное на двоичный поиск, будет пропорционально логарифму **N** с основанием 2.

Так как во всех формулах используется константа **k**, которая зависит от различных условий — мощности процессора, эффективности кода, сгенерированного компилятором и других, — ее можно

опустить и использовать O-синтаксис (Order of..). Он не дает конкретных цифр, а передает общий характер зависимости времени выполнения от количества элементов.

Таким образом, формулы можно привести к виду:

- Линейный поиск — $O(N)$;
- Двоичный поиск — $O(\log_2 N)$;
- Вставка в неупорядоченный массив — $O(1)$, то есть константа.

Можно сказать, что массивы поддерживают все необходимые операции для работы с данными. Вставка в неупорядоченный массив происходит быстро, зато поиск — медленно. Другой недостаток массива в том, что он имеет фиксированную длину. Если указать в начале программы меньший размер, чем требуется, это приведет к ошибкам, а если больший — к нецелевому использованию памяти.

Сортировка

Пузырьковая сортировка

Один из самых простых способов сортировки элементов массива. Рассмотрим механизм пузырьковой сортировки. Чтобы отсортировать числа по возрастанию, с самого левого элемента массива сравниваются два значения. Если левый элемент больше правого, они меняются местами. После чего происходит один шаг вправо, и действия повторяются снова, и так до конца массива. При первом проходе выполняется **N-1** действий. Когда определено самое большое число, расположившееся в итоге конце массива, сортировка идет дальше. Алгоритм начинается с начала и вновь сравнивает элементы. На этом этапе количество действий будет равно **N-2**. Сортировка продолжается, пока не будет отсортирован весь массив.

Создадим в классе **MyArr** метод **sortBubble**, который реализует пузырьковую сортировку. Также добавим закрытый метод **change**, который будет вызываться из метода **sortBubble** и менять элементы массива местами.

Ниже представлены два реализованных метода, которые необходимо вставить в класс **MyArr**:

```
public void sortBubble() {
    int out, in;
    for (out = this.size - 1; out >= 1; out--) {
        for (in = 0; in < out; in++) {
            if (this.arr[in] > this.arr[in + 1]) {
                change(in, in + 1);
            }
        }
    }
}

private void change(int a, int b) {
    int tmp = this.arr[a];
    this.arr[a] = this.arr[b];
    this.arr[b] = tmp;
}
```

Внешний цикл по переменной **out** начинается с конца массива и с каждой итерацией уменьшается на единицу. Это экономит время и позволяет не сравнивать уже отсортированные значения. Внутренний

цикл по **in** сравнивает между собой значения массива и вызывает метод **change**, в котором происходит смена позиций элементов.

Сложность пузырьковой сортировки очень высокая и достигает **O(N²)**. Это происходит за счет использования двух циклов: внешнего и внутреннего.

Сортировка методом выбора

В сортировке методом выбора перебор элементов начинается с крайнего левого. В отличие от пузырьковой сортировки, здесь добавляется дополнительный элемент — маркер. В начале сортировки крайний левый элемент помечается маркером и считается минимальным. Далее каждое следующее значение сравнивается с маркером. Если оно меньше, чем значение в маркере, необходимо переписать последнее на обнаруженное минимальное. Когда проход по массиву будет завершен, в маркере будет минимальный элемент.

Теперь необходимо переставить значение, которое находится в маркере, на первое место в массиве. Первый шаг сделан: один элемент отсортирован. Для сортировки одного элемента было проведено **N-1** сравнений и одна перестановка. С каждой следующей итерацией будет отсортировано по одному элементу.

Создадим в классе **MyArr** метод **sortSelect**, который реализует сортировку методом выбора:

```
public void sortSelect() {
    int out, in, mark;
    for (out=0; out<this.size; out++) {
        mark = out;
        for (in = out+1; in<this.size; in++) {
            if (this.arr[in] < this.arr[mark]) {
                mark = in;
            }
        }
        change(out, mark);
    }
}
```

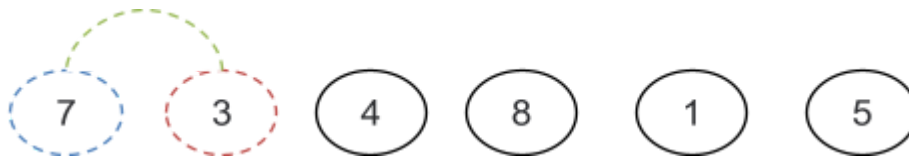
Перебор элементов внешнего цикла начинается с нулевого индекса. Внутренний цикл начинается с элемента, соответствующего текущему значению элемента внешнего цикла, и движется вправо с каждой итерацией. На каждой внутренней итерации цикла элементы сравниваются, и если элемент внутреннего цикла меньше маркера **mark**, происходит операция присваивания **mark = in**.

При применении алгоритма метода выбора количество перестановок равно **O(N)**, а количество сравнений **O(N²)**.

Сортировка методом вставки

Сортировка методом вставки — лучшая среди элементарных алгоритмов сортировки.

Суть алгоритма — в сравнении каждого элемента массива со всеми остальными, которые находятся слева от него. Представим, что есть массив, который состоит из 6 элементов: {7,3,4,8,1,5}. Отсортируем его методом вставки. Сортировка начинается со второго элемента, так как у первого нет соседей слева.



Сравниваем 3 и 7. Три меньше семи. Меняем их местами.



На следующем шаге берем цифру четыре и сравниваем ее сначала с семеркой.



Четыре меньше семи. Меняем их местами. На следующем шаге сравниваем четверку с тройкой.



Так как четверка больше чем тройка, менять их местами не будем. То же самое нужно сделать с остальными элементами массива.

Реализация сортировки методом вставки:

```
public void sortInsert(){
    int in, out;
    for(out = 1; out < this.size; out++){
        int temp = this.arr[out];
        in = out;
        while(in > 0 && this.arr[in-1] >= temp){
            this.arr[in] = this.arr[in-1];
            --in;
        }
        this.arr[in] = temp;
    }
}
```

Так как первый элемент массива с индексом 0 сравнивать с самим собой смысла нет, внешний цикл начинается со второго элемента массива. Во внутреннем цикле **while** счетчик **in** начинает с позиции **out** и движется влево и завершается, когда **temp** станет меньше элемента массива или когда дальнейшее смещение станет невозможным. При каждом проходе по циклу **while** следующий отсортированный элемент сдвигается на одну позицию влево.

Отсортируем объекты, используя сортировку методом вставки. Ниже представлен код, реализующий ее. В приведенном примере создан класс, в котором есть методы удаления и добавления элементов массива, а также метод **sortInsertObj**, который сортирует объекты типа **Person**.

```

class Person{
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

class MyArr{
    private Person[] arr;
    private int size;

    public MyArr(int size){
        this.size = 0;
        this.arr = new Person[size];
    }

    public boolean find(String search){
        int i;
        for(i=0;i<this.size;i++){
            if (this.arr[i].getName().equals(search)) break;
        }
        if (i==this.size)
            return false;
        else
            return true;
    }

    public void display(){
        for(int i=0;i<this.size;i++){
            System.out.println(this.arr[i].getName()+"
"+this.arr[i].getAge());
        }
    }

    public void delete(String search){
        int i=0;
        for(i=0;i<this.size;i++){
            if (this.arr[i].getName().equals(search)) break;
        }

        for (int j=i;j<this.size-1; j++){
            this.arr[j] = this.arr[j+1];
        }
        this.size--;
    }
}

```

```

    }

    public void insert(String name, int age){
        this.arr[this.size] = new Person(name, age);
        this.size++;
    }

    public void sortInsertObj(){
        int in, out;
        for(out = 1; out < this.size; out++){
            Person temp = this.arr[out];
            in = out;
            while(in > 0 && this.arr[in-1].getName().compareTo(temp.getName())
> 0){
                this.arr[in] = this.arr[in-1];
                --in;
            }
            this.arr[in] = temp;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        int size = 100;
        MyArr arr = new MyArr(size);
        arr.insert("Vasya", 10);
        arr.insert("Igor", 15);
        arr.insert("Viktor", 15);
        arr.display();

        arr.sortInsertObj();

        arr.display();
    }
}

```

Практическое задание

1. Создать массив большого размера (миллион элементов).
2. Написать методы удаления, добавления, поиска элемента массива.
3. Заполнить массив случайными числами.
4. Написать методы, реализующие рассмотренные виды сортировок, и проверить скорость выполнения каждой.

Дополнительные материалы

1. [Двоичный поиск.](#)
2. [Сортировка методом вставки.](#)

3. [Сортировка методом выбора.](#)
4. [Пузырьковая сортировка.](#)

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Лафорте Р. Структуры данных и алгоритмы в Java. Классика Computers Science. 2-е изд. —СПб.: Питер, 2013. — 46-119 сс.