Урок 2

# Массивы и сортировка

Работа с массивами и способов их сортировки.

# Введение

Массивы - одни из самых популярных структур данных поддерживаются многими языками программирования. Как правило, изучение структур данных начинается с массивов, так как более сложные структуры используют его для своей реализации.

Массивы в Java – это структура данных, которая содержит элементы одного типа, которые расположены друг за другом в памяти компьютера. Массив в Java являются ссылочным типом данных, а доступ к его элементам осуществляется по индексу. Размерность массива или, другими словами, его длина это количество его элементов. В Java индексация массива начинается с индекса ноль.


В уроке будет рассмотрена работа с массивами в Java, в частности, создание массива, вставка и поиск его элементов. Рассмотрим, одну из разновидностей массива, который называется упорядоченным, а также разберемся со способами сортировки элементов массива.

## Создание массива

Начнем с самого простого – это создание массива. Как было сказано выше, в Java массив является ссылочным типом данных, т.е. относится к объектам. Поэтому для его создания необходимо использовать 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 содержаться данные массива.

В большинстве случаев определение ссылки и создание массива объединяются в одной 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. Также есть альтернативный синтаксис создания массива, когда 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 нельзя.

## Обращение к элементам массива

Для обращения к элементам массива, после ссылки в квадратных скобках указывается индекс требуемого элемента. Такой синтаксис для обращения к элементам массива 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, то 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 же массив имеет ссылочный тип элементов, то 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 указанным 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 нельзя. Можно сдвинуть массив на одно значение влево. Допустим, нам необходимо удалить элемент массива со значением 3. Первое, что необходимо сделать, это 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 = 4;

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

        for(i=0;i<len;i++){
            if (myArr1[i] == search) break;
        }
        for (int j = i; j<len-1; j++){
            myArr1[j] = myArr1[j+1];
        }
        len--;

        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 массиву, ищем необходимый элемент и, если он найден, останавливаем цикл. Таким образом, переменная 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 будет следующим.

```
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. Давайте разделим 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 мы используем методы, реализованные в классе MyArr. Но и такая структура созданного класса является не идеальной и низкоуровневой. По сути методы setElement и getElement делают тоже самое, что и оператор []. Также пользователь использует 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){
        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 класса 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 осуществлять линейный поиск. Линейный поиск осуществляется простым сравнением очередного элемент в массиве с искомым значением и если значения совпадают, то поиск считается завершенным.

Ниже представлен пример реализации линейного поиска в классе MyArr.

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

}
```

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 найден, то возвращается логическое значение false.

Добавим в реализацию метода main проверку на существование удаляемого элемента.

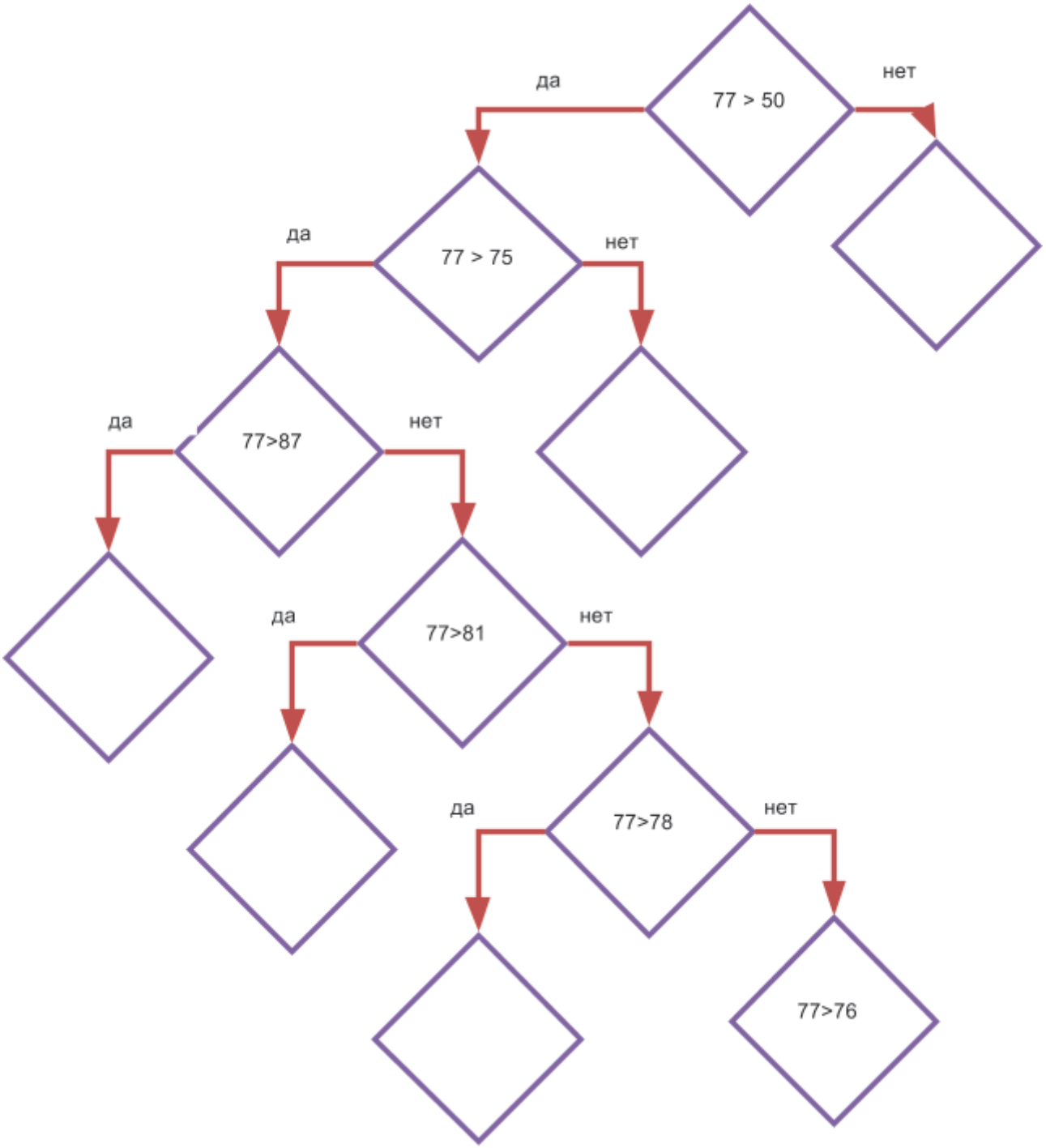
```
public static void main(String[] args) {

        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 массив");
        arr.display();

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

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

}
```

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

Двоичный поиск – это алгоритм поиска элемента в отсортированном массиве. Двоичный поиск можно сравнить с игрой «угадай число», когда один игрок загадывает число от 1 до 100, а другой пытается его угадать. Для того чтобы угадать число за меньшее число шагов, следует всегда начинать с 50. Если загаданное число больше, то оно находится в диапазоне 51-100 и поиск искомого значения сократиться вдвое.

Представим, что мы хотим найти число 77. На первом шаге мы разделим диапазон значений пополам и сравним число 77 с 50. Число 77 больше 50, поэтому искомое значение будем искать в диапазоне от 51 до 100. Каждое следующее предположение будет сокращать диапазон искомого значения вдвое. Следующее сравнение нашего числа с 77 с числом 75. Так как наше число больше, поиск будет осуществляться в диапазоне от 76 до 100 и так далее пока не найдется число.

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




Добавим к нашему классу MyArr метод binaryFind, который осуществляет двоичный поиск элемента массива.

Метод binaryFind выглядит следующим образом.

```
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;
                        } else {
                                low = mid+1;
                        }
                }
         }
         return false;

}
```

Сначала определяем переменные, которые хранят индексы верхней и нижней границы поиска. В переменной mid будет граница нашего поиска, которая будет делить 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, иначе присваиваем значение переменой mid нижней границе. После чего переходим на новую итерацию цикла, до тех пор, пока значение не будет найдено или переменная low будет больше чем high.

Так как массив является отсортированным по возрастанию, появляется необходимость модернизировать алгоритм метода вставки элемента. Как раньше вставлять значения в конец массива не получится, теперь необходимо сравнивать вставляемый элемент со всеми, которые есть в массиве, и вставлять его на нужное место, а остальные сдвигать вправо. Ниже представлен 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 представлен ниже.

```
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;
                 } 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) 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]);
        }
    }

    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.

Как же определить скорость работы двоичного поиск и количество шагов, которое потребуется ему для нахождения искомого значения?

Формула, по которой можно высчитать количество шагов при двоичном поиске, выглядит следующим образом.

$$r = 2^s$$

r - размер диапазона, s - количество шагов.

Представим, что нам нужно проводить поиск в диапазоне от 1 до 100. Если s будет равна 6, то $6^2 = 64$ и этого не достаточно, чтоб покрыть весь диапазон поиска. А если s = 7, то этого будет достаточно с избытком. Т.е. поиск искомого значения в диапазоне от 1 до 100 будет произведен максимум за 7 шагов.

Давайте попробуем разобраться 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 внести некоторые изменения в методы. Изменим тип массива с int на Person. Поиск и удаление теперь осуществляется 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

Рассмотрим сложность алгоритмов вставки, линейного и двоичного поиска. Сложность алгоритма вставки не зависит от количества элементов массива, так как новый элемент всегда вставляется в конец списка. Представим, что для вставки требуется некоторое K время, которое для вставки любого значения будет одинаковым. Тогда формула для расчета алгоритма вставки будет рассчитываться как:

$$T = k$$

Где T – время вставки в неупорядоченный массив.

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

$$T = k \times N$$

Где N – количество элементов массива.

Как и для линейного поиска можно составить формулу и для двоичного поиска:

$$T = k \times log_2 N$$

Как было сказано ранее, 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, таких как мощность процессора, эффективности кода, который сгенерировал компилятор и т.д., ее можно опустить. И использовать в формулах «O-синтаксис».

Таким образом, наши формулы могут быть приведены к виду:

Линейный поиск – O(N);

Двоичный поиск – $O(\log_2 N)$;

Вставка в неупорядоченный массив – O(1).

Таким образом, можно сказать, что массивы поддерживают все необходимые операции для работы с данными. Вставка в неупорядоченный массив происходит быстро, зато поиск осуществляется медленно. Другой недостаток массивов заключается в том, что он имеет фиксированную длину. Если указать в начале программы меньший размер массива, чем требуется, то это приведет к ошибкам в работе программы, а если выше, то приведет к нецелевому использованию памяти.

# Сортировка

Рассматривая двоичный поиск, было сказано, что его применяют к отсортированному массиву. Рассмотрим основные виды сортировок, которые применяются над массивами.

## Пузырьковая сортировка

Один из самых простых способов сортировки элементов массива – пузырьковая сортировка. Пузырьковая сортировка выполняется следующим образом. Представим, что нам необходимо отсортировать числа по возрастанию. Начиная с самого левого элемента массива, сравниваются два значения. Если правый элемент больше левого, то они меняются местами. После чего происходит один шаг в право и действия повторяются снова, пока не будет достигнут конец массива. Но на этом сортировка не заканчивается. Сейчас мы только определили самое большое число, которое будет стоять в конце массива. При первом проходе выполняется N-1 действий. После установки первого числа в конец массива, алгоритм начинается с начала и вновь сравнивает между собой элементы. На этом шаге количество действий будет равно N-2 и так далее, пока не будет отсортирован весь массив.

Давайте в классе MyArr создадим метод sortBubble, который реализует пузырьковую сортировку. Также создадим 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 позволит сократить время и 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 большая и достигает $O(N^2)$. 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 в сортировке методом выбора добавляется дополнительный элемент, который называется маркер. В начале сортировки крайний левый элемент помечается маркером и считается самым минимальным. Далее каждое следующее значение сравнивается с маркером. Если значение следующего элемента будет меньше, чем значения в маркере, то необходимо переписать значения маркера на значения сравниваемого с ним элемента. Когда проход по массиву будет завершен, то в маркере будет находиться минимальный элемент. Теперь необходимо переставить значение, которое находится в маркере на первое место в массиве. Первый шаг закончен, и 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);
        }

}
```

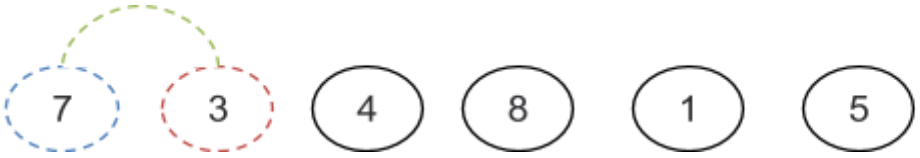
Перебор элементов внешнего цикла начинается с нулевого индекса. Внутренний цикл начинается с элемента соответствующего текущему значению элемента внешнего цикла и двигается вправо с каждой итерацией. На каждой внутренней итерации цикла элементы сравниваются и если элемент внутреннего цикла меньше маркера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

Используя алгоритм методом выбора, количество перестановок становится равной O(N), а количество сравнений $O(N^2)$.

## Сортировка методом вставки

Сортировка методом вставки является лучшей среди элементарных алгоритмов сортировки, которые были рассмотрены в данном уроке.

Суть алгоритма заключается в сравнивание каждого элемента массива со всеми остальными элементами, которые находятся справа от него. Представим, что есть массив, который состоит из 6 следующих элементов {7,3,4,8,1,5}. Давайте отсортируем его методом вставки. Сортировка начинается со второго элемента, так как у первого элемента нет соседей справа.




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, то менять их местами не будем.

Тоже самое нужно будет сделать со всеми остальными элементами массива

Ниже представлена 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 ноль сравнивать справа смысла нет. Внешний цикл начинается со второго элемента массива. Во внутреннем цикле while счетчик in начинает с позиции out и двигается влево — либо пока temp не станет меньше элемента массива, либо когда дальнейшее смещение станет невозможным. При каждом проходе по циклу while следующий отсортированный элемент сдвигается на одну позицию вправо.

Давайте попробуем отсортировать объекты, используя сортировку методом вставки. Ниже представлен код, реализующий эту сортировку. 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();
    }

}
```

# Домашнее 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.