Лабораторная работа №3

Выполнил работу

Гапанюк Антон Андреевич

Группа: 6204-010302D

Самара, 2025 г.

Цель работы: дополнить пакет для работы с функциями одной переменной, заданными в табличной форме, добавив классы исключений, новый класс функций и базовый интерфейс.

Ход работы

Задание 1

Изучим классы исключений:

1. **java.lang.Exception** – базовый класс всех исключений. Родитель всех «проверяемых» исключений.
2. **java.lang.IndexOutOfBoundsException** – выход за границы. Возникает при обращении к несуществующему индексу в массиве, списке и т. д.
3. **java.lang.ArrayIndexOutOfBoundsException** - частный случай. Возникает при обращении к массиву с некорректным индексом
4. **java.lang.IllegalArgumentException** - неверный аргумент. Возникает, когда методу передают некорректный аргумент.
5. **java.lang.IllegalStateException** - неверное состояние. Возникает, когда объект находится в состоянии, не позволяющем выполнить операцию.

Задание 2

Создадим классы исключений

FunctionPointIndexOutOfBoundsException – исключение выхода за границы набора точек при обращении к ним по номеру, наследует от класса **IndexOutOfBoundsException**;

InappropriateFunctionPointException – исключение, выбрасываемое при попытке добавления или изменения точки функции несоответствующим образом, наследует от класса **Exception**.

```java
public class FunctionPointIndexOutOfBoundsException extends IndexOutOfBoundsException {
    // Конструктор по умолчанию
    public FunctionPointIndexOutOfBoundsException(){
        super(); // super() - вызов конструктора родителя
    }
    // Конструктор с сообщением
    public FunctionPointIndexOutOfBoundsException(String message){
        super(message);
    }
    // Конструктор с индексом
    public FunctionPointIndexOutOfBoundsException(int index){
        super("Индекс: [" + index + "] выходит за границы");
    }
    // Конструктор с индексом и размером массива
    public FunctionPointIndexOutOfBoundsException(int index, int size){
        super("Индекс: [" + index + "] выходит за границы массива размером " + size);
    }
}
```

Рис. 1 – класс **FunctionPointIndexOutOfBoundsException**

```java
public class InappropriateFunctionPointException extends Exception{
    // Конструктор по умолчанию
    public InappropriateFunctionPointException(){
        super();
    }
    // Конструктор с сообщением
    public InappropriateFunctionPointException(String message){
        super(message);
    }
    // Конструктор с сообщением и причиной
    public InappropriateFunctionPointException(String message, Throwable cause){
        super(message, cause); // Throwable - суперкласс всех ошибок и исключениий в Java
    }
}
```

Рис. 2 – класс **InappropriateFunctionPointException**

Задание 3

Добавим исключения **IllegalArgumentException** в конструкторы **TabulatedFunction** для проверки области определения и количества точек.

```java
// Проверка на область определения
if (leftX >= rightX){
    throw new IllegalArgumentException("Левая граница (" + leftX + ") должна быть меньше правой (" + rightX + ")");
}
// Проверка на кол-во точек
if (pointsCount < 2){
    throw new IllegalArgumentException(s:"Кол-во точек должно быть не меньше двух");
}
```

Рис. 3 – Проверки в конструкторе **TabulatedFunction**

Во втором конструкторе во второй проверке заменим **pointsCount** на **values.length**.

В методы **getPoint()**, **setPoint()**, **getPointX()**, **setPointX()**, **getPointY()**, **setPointY()** и **deletePoint()** добавим исключение **FunctionPointIndexOutOfBoundsException**. Оно будет проверять выходит ли номер, переданный в метод, за границы набора точек. Это обеспечит корректность обращений к точкам функции.

```
// Проверка на номер, выходящий за границы набора точек
if (index < 0 || index >= points.length){
    throw new FunctionPointIndexOutOfBoundsException(index, points.length);
}
```

Рис. 4 – Проверка на номер

Добавим исключения **InappropriateFunctionPointException** в методы **setPoint()** и **setPointX()**. Они будут вызываться в случае, если координата **x** задаваемой точки лежит вне интервала, определяемого значениями соседних точек функции.

```
// Проверяем, лежит ли координата x вне интервала
        if (index > 0 && x <= points[index - 1].getX()){
            throw  new InappropriateFunctionPointException("X координата (" + x + ")
должна быть больше предыдущей точки (" + points[index - 1].getX() + ")");
        }
        if (index < points.length - 1 && x >= points[index + 1].getX()){
             throw  new InappropriateFunctionPointException("X координата (" + x + ")
должна быть меньше следующей точки (" + points[index - 1].getX() + ")");
          }
```

Добавим исключение **InappropriateFunctionPointException** в метод **addPoint()**. Оно будет вызываться в случае, если абсцисса точки из набора точек функции совпадает с абсциссой добавляемой точки.

```
// Проверяем на дубликат
        if (insert_index < points.length && point.getX() ==
points[insert_index].getX())
            throw  new InappropriateFunctionPointException("Точка X с координатой ("
+ point.getX() + ") уже существует");
```

В метод **deletePoin**t добавим исключение, проверяющее, что после удаления останется минимум 2 точки.

```
if (points.length < 3){
            throw new IllegalStateException("Невозможно удалить точку: количество
точек не может быть меньше двух");
        }
```

Задание 4 (код в примечании документа)

Реализуем двусвязный циклический список. Он будет состоять из двух классов: **LinkedListTabulatedFunction** и **FunctionNode**. **FunctionNode** будет отвечать за узлы списка, а **LinkedListTabulatedFunction** за весь список. Класс **LinkedListTabulatedFunction** будет совмещать в себе две функциии: будет описывать связный список и работу с ним и будет описывать работы с табулированной функцией и ее точками.

Реализуем первую функцию. Опишем класс **FunctionNode**, класс **LinkedListTabulatedFunction**.

Реализуем методы:

- **FunctionNode getNodeByIndex(int index)** - возвращающий ссылку на объект элемента списка по его номеру
- **FunctionNode addNodeToTail()**, добавляющий новый элемент в конец списка и возвращающий ссылку на объект этого элемента.
- **FunctionNode addNodeByIndex(int index)**, добавляющий новый элемент в указанную позицию списка и возвращающий ссылку на объект этого элемента.
- **FunctionNode deleteNodeByIndex(int index)**, удаляющий элемент списка по номеру и возвращающий ссылку на объект удаленного элемента.

Задание 5 (код в примечании документа)

Реализуем вторую функцию класса **LinkedListTabulatedFunction**. Создадим конструкторы и методы, аналогичные конструкторам и методам класса **TabulatedFunction**. Отличием этих методов будет то, что мы будем теперь работать с узлами списка и вызывать существующие геттеры и сеттеры (принцип инкапсуляции). Учтем исключения и не забудем про оптимизацию. Оптимизировать будем благодаря полям **lastIndex** и **lastAccessed.**

Задание 6

Переименуем класс **TabulatedFunction** на **ArrayTabulatedFunction**. Создадим интерфейс **TabulatedFunction**, содержащий объявления общих методов классов **ArrayTabulatedFunction** и **LinkedListTabulatedFunction**.

Оба класса будут реализовать созданный интерфейс. Теперь работы с функциями заключена в типе интерфейса, а в классах заключена только реализация этой работы.

```
public interface TabulatedFunction {
    // Методы получения границ области определения
    double getLeftDomainBorder();
    double getRightDomainBorder();

    // Метод получения значения функции
    double getFunctionValue(double x);

    // Методы работы с точками
    int getPointsCount();
    FunctionPoint getPoint(int index);
    void setPoint(int index, FunctionPoint point) throws InappropriateFunctionPointException;
    double getPointX(int index);
    void setPointX(int index, double x) throws InappropriateFunctionPointException;
    double getPointY(int index);
    void setPointY(int index, double y);

    // Методы модификации точек
    void deletePoint(int index);
    void addPoint(FunctionPoint point) throws InappropriateFunctionPointException;
}
```

Рис. 5 – Интерфейс **TabulatedFunction**

Задание 7

Проведем тесты, сначала убедимся, что базовые функции работают, затем проверим исключения. По указанию преподавателя, изменим знак сравнения «==» на сравнение с машинным эпсилоном.

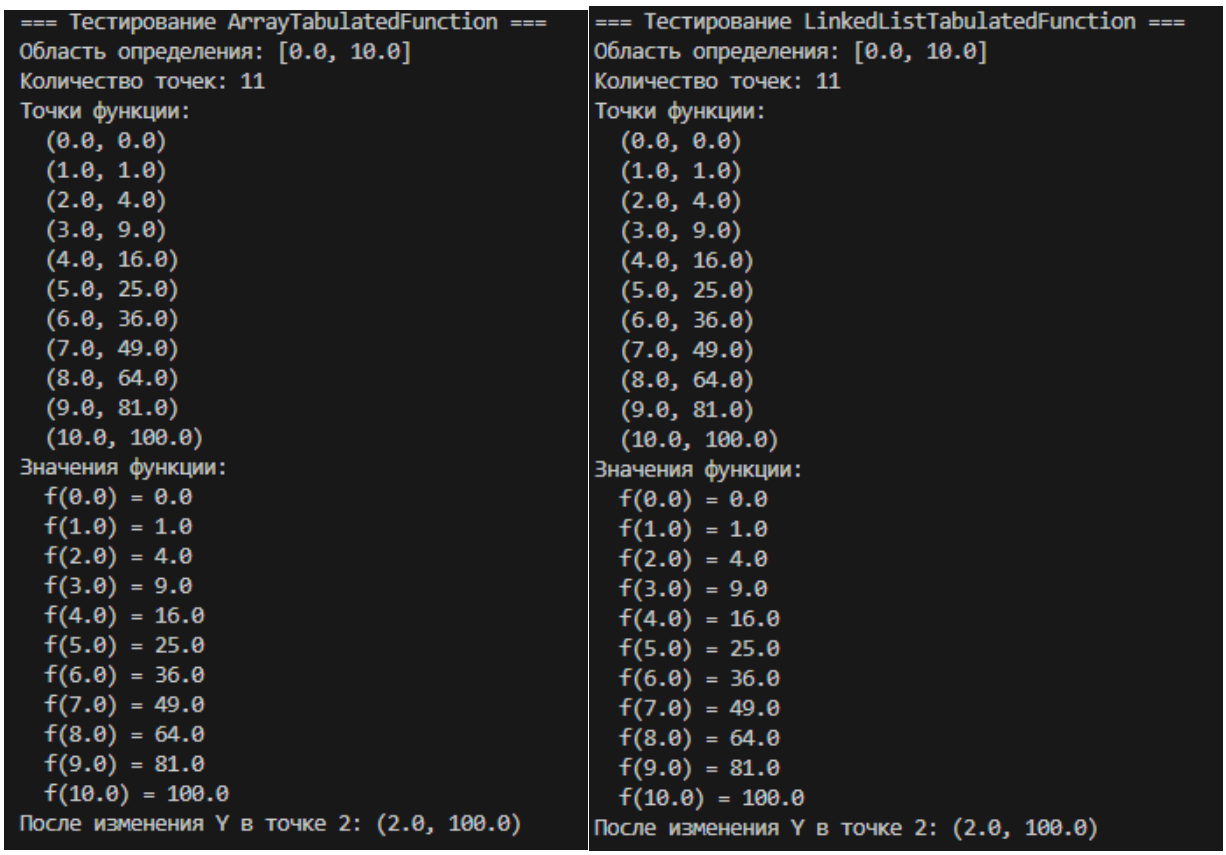
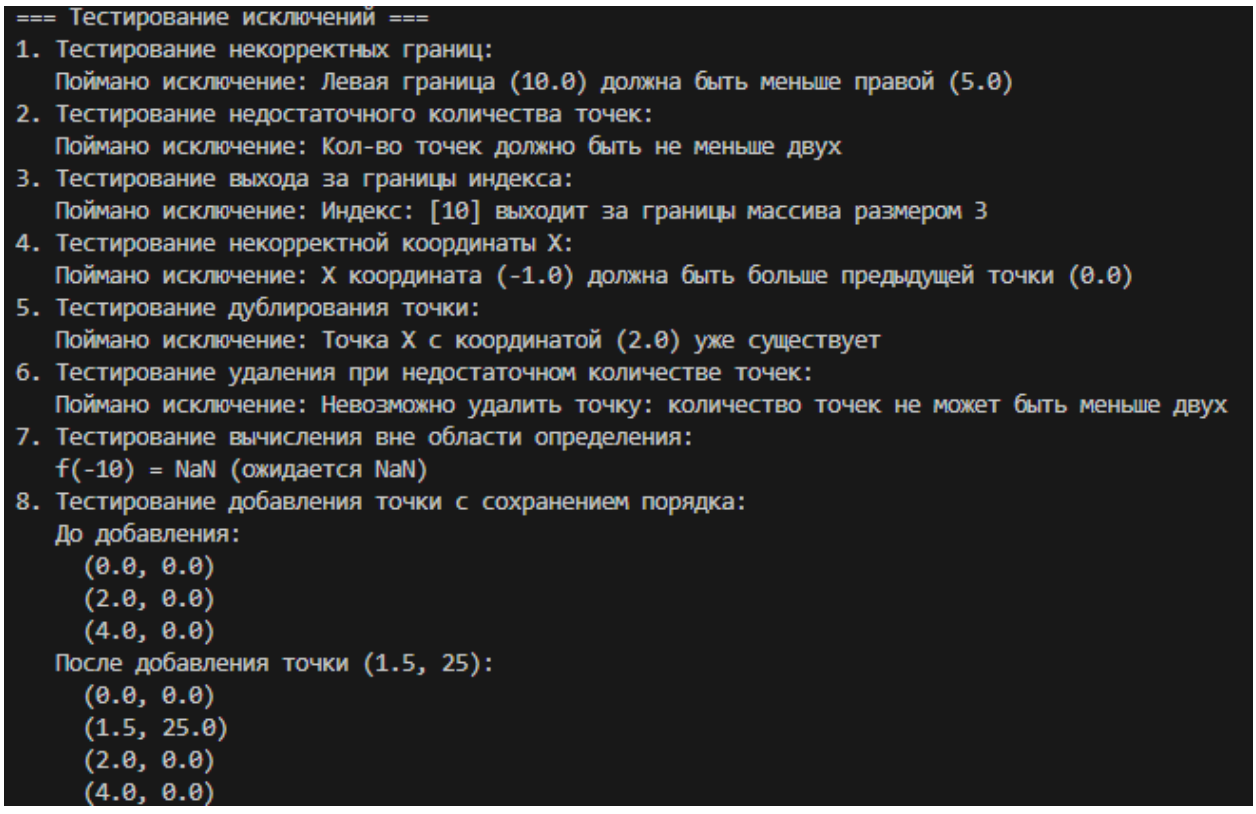


Рис 6-7 Базовые тесты



Из тестов видно, что код рабочий, все тесты выполняются корректно.

Примечание (код LinkedListTabulatedFunction)

```
public class LinkedListTabulatedFunction implements TabulatedFunction{

    // ========= Внутренний класс для узла списка =========

    private class FunctionNode {
        private FunctionPoint point; // точка

        // Ссылки на соседние узлы
        private FunctionNode prev;
        private FunctionNode next;



        // Конструкторы
        public FunctionNode(FunctionPoint point){
            this.point = point;
            this.prev = null;
            this.next = null;
        }

        public FunctionNode(FunctionPoint point, FunctionNode prev, FunctionNode
next){
            this.point = point;
            this.prev = prev;
            this.next = next;
        }

        // Геттеры и сеттеры
        public FunctionPoint getPoint(){
            return point;
        }

        public void setPoint(FunctionPoint point){
            this.point = point;
        }

        public FunctionNode getPrev(){
            return prev;
        }
```

```
        public FunctionNode getNext(){
            return next;
        }

        public void setPrev(FunctionNode prev){
            this.prev = prev;
        }

        public void setNext(FunctionNode next){
            this.next = next;
        }
    }

    // ========= Основной класс LinkedListTabulatedFunction ==========

    private FunctionNode head; // Голова списка (не хранит данные)
    private int pointsCount; // Кол-во точек
    private FunctionNode lastAccessed; // Последний доступный узел (для оптимизации)
    private int lastIndex; // Индекс последнего доступа

    // Конструкторы
    public LinkedListTabulatedFunction(double leftX, double rightX, int pointsCount){
        // Проверка на область определения
        if (leftX >= rightX){
            throw new IllegalArgumentException("Левая граница (" + leftX + ") должна
быть меньше правой (" + rightX + ")");
        }
        // Проверка на кол-во точек
        if (pointsCount < 2){
            throw new IllegalArgumentException("Кол-во точек должно быть не меньше
двух");
        }
        initializeList(leftX, rightX, pointsCount, null);
    }

    public LinkedListTabulatedFunction(double leftX, double rightX, double[] values){
        // Проверка на область определения
        if (leftX >= rightX){
```

```
            throw new IllegalArgumentException("Левая граница (" + leftX + ") должна
быть меньше правой (" + rightX + ")");
        }
        // Проверка на кол-во точек
        if (values.length < 2){
            throw new IllegalArgumentException("Кол-во точек должно быть не меньше
двух");
        }

        initializeList(leftX, rightX, values.length, values);

    }

    // Инициализация списка
    public void initializeList(double leftX, double rightX, int count, double[]
values){
        // Создаем голову двухсвязного циклического списка
        head = new FunctionNode(null);
        head.setNext(head);
        head.setPrev(head);

        pointsCount = 0;
        lastAccessed = head;
        lastIndex = -1;

        double step = ((rightX - leftX)/(count - 1));
        for(int i = 0; i < count; ++i){
            double x = leftX + i * step;
            double y = (values == null) ? 0 : values[i];
            addNodeToTail().setPoint(new FunctionPoint(x, y)); // Добавляем элементы
в конец списка
        }
    }

    // Метод добавления узла в конец списка
    private FunctionNode addNodeToTail(){
        FunctionNode newNode = new FunctionNode(null);

        // Вставляем перед головой
        FunctionNode tail = head.getPrev();
```

```
        // Связываем элементы списка
        tail.setNext(newNode);
        newNode.setNext(head);
        newNode.setPrev(tail);
        head.setPrev(newNode);

        ++pointsCount;
        lastAccessed = newNode;
        lastIndex = pointsCount - 1;

        return newNode;
    }

    // Оптимизированный доступ к узлу по индексу
    private FunctionNode getNodeByIndex(int index){
        // Проверка на номер, выходящий за границы набора точек
        if (index < 0 || index >= pointsCount){
            throw new FunctionPointIndexOutOfBoundsException(index, pointsCount);
        }

        // Оптимизация: начинаем с последнего доступного узла
        FunctionNode current;
        int startIndex;

        if (lastIndex != -1 && Math.abs(index - lastIndex) < Math.abs(index)){
            // Начинаем с последнего доступного узла
            current = lastAccessed;
            startIndex = lastIndex;
        }
        else {
            // Начинаем с головы
            current = head.getNext();
            startIndex = 0;
        }

        // Двигаемся к нужному узлу
        if (index >= startIndex){
            // Двигаемся вперед
            for (int i = startIndex; i < index; ++i){
```

```
            current = current.getNext();
        }
    }
    else {
        // Двигаемся назад
        for (int i = startIndex; i > index; --i){
            current = current.getPrev();
        }
    }

    // Сохраняем для след. вызова
    lastAccessed = current;
    lastIndex = index;

    return current;
}

private FunctionNode addNodeByIndex(int index){
    // Проверка на номер, выходящий за границы набора точек
    if (index < 0 || index >= pointsCount){
        throw new FunctionPointIndexOutOfBoundsException(index, pointsCount);
    }

    // Особый случай добавления в конец
    if (index == pointsCount){
        return addNodeToTail();
    }

    FunctionNode newNode = new FunctionNode(null);

    // Находим узел, перед которым вставляем
    FunctionNode nextNode = getNodeByIndex(index);
    FunctionNode prevNode = nextNode.getPrev();

    // Связываем элементы
    newNode.setNext(nextNode);
    newNode.setPrev(prevNode);
    nextNode.setPrev(newNode);
    prevNode.setNext(newNode);
```

```
        ++pointsCount;
        lastAccessed = newNode;
        lastIndex = index;

        return newNode;
    }

    private FunctionNode deleteNodeByIndex(int index){
        // Проверка на номер, выходящий за границы набора точек
        if (index < 0 || index >= pointsCount){
            throw new FunctionPointIndexOutOfBoundsException(index, pointsCount);
        }

        // Проверяем, что после удаления останется минимум 2 точки
        if (pointsCount < 3){
            throw new IllegalStateException("Невозможно удалить точку: количество
точек не может быть меньше двух");
        }

        FunctionNode delNode = getNodeByIndex(index);
        FunctionNode prevNode = delNode.getPrev();
        FunctionNode nextNode = delNode.getNext();

        // Удаляем ссылки на удаляемый узел
        prevNode.setNext(nextNode);
        nextNode.setPrev(prevNode);

        // Очищаем ссылки удаляемого узла
        delNode.setPrev(null);
        delNode.setNext(null);

        --pointsCount;

        // Обновляем lastAccesed
        if (lastIndex == index){
            lastAccessed = (index < pointsCount) ? nextNode : head.getNext();
            lastIndex = (index < pointsCount) ? index : 0;
        }
        else if (lastIndex > index){
            --lastIndex;
```

```
        }

        return delNode;
    }


    // ========== Методы TabulatedFunction ==========

    // Методы получения крайних значений области определения
    public double getLeftDomainBorder(){
        return head.getNext().getPoint().getX();
    }

    public double getRightDomainBorder(){
        return head.getPrev().getPoint().getX();
    }

    public double getFunctionValue(double x) {
        if (x < getLeftDomainBorder() || x > getRightDomainBorder()) {
            return Double.NaN;
        }

        // Простой поиск с начала
        FunctionNode current = head.getNext();
        while (current != head) {
            FunctionNode next = current.getNext();
            if (next == head) {
                // Последний элемент
                if (x == current.getPoint().getX()) {
                    return current.getPoint().getY();
                }
                break;
            }

            double x1 = current.getPoint().getX();
            double x2 = next.getPoint().getX();

            if (Math.abs(x - x1) < 1e-10)  return current.getPoint().getY();
           if (Math.abs(x - x2) < 1e-10)  return next.getPoint().getY();
            if (x >= x1 && x <= x2) {
```

```
                double y1 = current.getPoint().getY();
                double y2 = next.getPoint().getY();
                return linearInterpolation(x, x1, x2, y1, y2);
            }

            current = next;
        }

        return Double.NaN;
}

    // Линейная интерполяция
    private double linearInterpolation(double x, double x1, double x2, double y1,
double y2){
        return y1 + (y2 - y1)/(x2 - x1)*(x - x1); // уравнение прямой по двум точкам
    }

    // Метод, возвращающий кол-во точек
    public int getPointsCount(){
        return pointsCount;
    }

    // Метод получения точки по индексу (возвращает копию)
    public FunctionPoint getPoint(int index){
        return new FunctionPoint(getNodeByIndex(index).getPoint());
    }

    // Функция замены указанной точки на переданную
    public void setPoint(int index, FunctionPoint point) throws
InappropriateFunctionPointException {

        FunctionNode newNode = getNodeByIndex(index);

        // Проверяем, лежит ли координата x вне интервала
        if (index > 0 && point.getX() <= getNodeByIndex(index -
1).getPoint().getX()){
            throw  new InappropriateFunctionPointException("X координата (" +
point.getX() + ") должна быть больше предыдущей точки (" + getPointX(index - 1) +
")");
        }
```

```
        if (index < pointsCount - 1 && point.getX() >= getNodeByIndex(index +
1).getPoint().getX()){
            throw new InappropriateFunctionPointException("X координата (" +
point.getX() + ") должна быть меньше следующей точки (" + getPointX(index + 1) +
")");
        }
        newNode.setPoint(new FunctionPoint(point));
    }

    // Метод возвращения абсциссы указанной точки
    public double getPointX(int index){
        return getNodeByIndex(index).getPoint().getX();
    }

    // Метод установки нового значения абсциссы у конкретной точки
    public void setPointX(int index, double x) throws
InappropriateFunctionPointException {
        FunctionPoint point = getPoint(index);
        point.setX(x);
        setPoint(index, point);
    }

    // Метод получения Y по индексу
    public double getPointY(int index){
        return getNodeByIndex(index).getPoint().getY();
    }

    // Метод установки Y по индексу
    public void setPointY(int index, double y){
        getNodeByIndex(index).getPoint().setY(y);
    }

    // Метод удаления точки по индексу
    public void deletePoint(int index) {
        deleteNodeByIndex(index);
    }

    public void addPoint(FunctionPoint point) throws
InappropriateFunctionPointException {
        int insertIndex = 0;
```

```
        // Оптимизациия: начинаем поиск с lastAccessed если он есть
        if (lastIndex != -1 && lastAccessed != head){
            double lastX = getPointX(lastIndex);
            if (point.getX() > lastX){
                // Ищем вперед от lastIndex
                insertIndex = lastIndex + 1;
                while (insertIndex < pointsCount && point.getX() >
getPointX(insertIndex)){
                    ++insertIndex;
                }
            }
            else {
                // Ищем назад от lastIndex
                insertIndex = lastIndex;
                while (insertIndex > 0 && point.getX() < getPointX(insertIndex - 1))
{
                insertIndex--;
                }
            }
        } else {
        // Обычный поиск с начала
        while (insertIndex < pointsCount && point.getX() > getPointX(insertIndex)) {
            insertIndex++;
            }
        }

        // Проверяем на дубликат
        if (insertIndex < pointsCount && Math.abs(point.getX() -
getPointX(insertIndex)) < 1e-10) {
            throw new InappropriateFunctionPointException("Точка с координатой X = "
+ point.getX() + " уже существует");
        }

        // Вставляем новую точку
        FunctionNode newNode = addNodeByIndex(insertIndex);
        newNode.setPoint(new FunctionPoint(point));
    }

}
```