МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение

высшего образования

«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

**Лабораторная работа № 3**

*по дисциплине «Алгоритмы и структуры данных»*

Выполнил студент

группы ПИбд-12

Нгуен Т. М.

Проверил доцент кафедры

«Информационные системы»

Кулешов А.В.

Ульяновск, 2025

### Задачи

### Первое задание Цель

Целью данной программы является:

* построение **бинарного дерева поиска** (Binary Search Tree),
* добавление узлов,
* выполнение **симметричного обхода** дерева (inorder traversal), при котором значения выводятся в **возрастающем порядке**.

**Описание реализации**

**Класс Node**

Представляет один узел дерева:

* int val — значение узла;
* Node left, right — ссылки на левого и правого потомков.

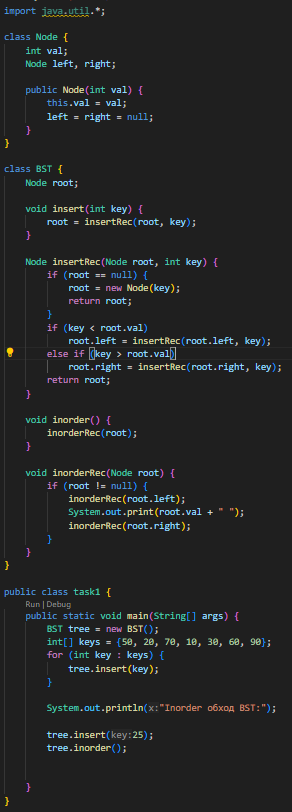
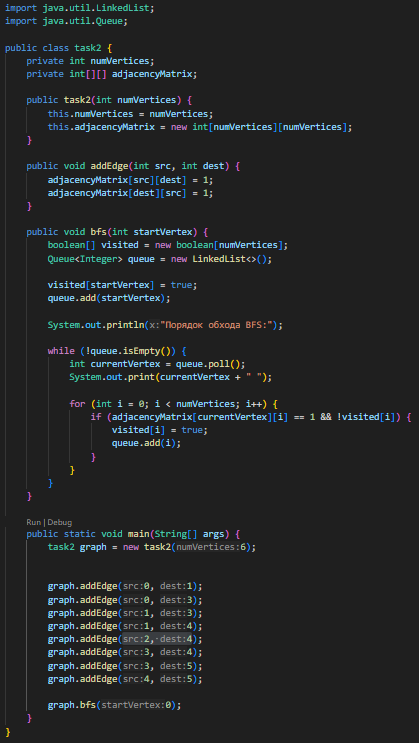
**Класс BST**

Содержит функциональность бинарного дерева поиска:

* insert(int key) — метод для вставки значения в дерево;
* insertRec(Node root, int key) — рекурсивная вставка с соблюдением свойства BST:
  + значения меньше корня идут влево,
  + больше — вправо.
* inorder() — запускает симметричный (inorder) обход;
* inorderRec(Node root) — рекурсивный обход:
  + Обход левого поддерева
  + Вывод значения текущего узла
  + Обход правого поддерева

**main() метод:**

1. Создаёт дерево.
2. Вставляет массив значений: {50, 20, 70, 10, 30, 60, 90}.
3. Добавляет значение 25.
4. Выполняет **симметричный обход** дерева, выводя значения в **отсортированном порядке**.

  
  
  
  
  
  
  
  
  
Вторая задача  
  
В этой задаче выполнен обход всех узел деревы, а чтобы обойти все узлы, нужно было начинать с корня и далее по его соседям, также реализованная проверка на то, чтобы мы не посещали узел несколько раз, все соседи я помещал в стек, чтобы удобно было вытягивать их оттуда и при обходе всех узлов, код завершал работу.

**Класс task2**

Представляет неориентированный граф:

* int numVertices — количество вершин в графе.
* int[][] adjacencyMatrix — **матрица смежности**, в которой 1 означает наличие ребра между вершинами.

**Метод addEdge(int src, int dest)**

Добавляет **ребро** между вершинами src и dest. Так как граф **неориентированный**, ребро устанавливается в обоих направлениях:

adjacencyMatrix[src][dest] = 1;

adjacencyMatrix[dest][src] = 1;

**Метод bfs(int startVertex)**

Выполняет **обход в ширину**:

1. Создаёт массив visited[] для отслеживания посещённых вершин.
2. Использует **очередь (Queue)** для управления порядком обхода.
3. Обходит граф, начиная с startVertex, последовательно добавляя в очередь все смежные непосещённые вершины.

**Метод main()**

1. Создаёт граф из 6 вершин.
2. Добавляет рёбра между ними.
3. Запускает bfs(0), начиная обход с вершины 0.

Третье задание

**Класс Edge**

Представляет ребро графа:

* to — вершина, в которую ведет ребро.
* weight — вес ребра.

**Метод dijkstra(Map<String, List<Edge>> graph, String start)**

Реализует алгоритм Дейкстры:

* distance — отображение из вершины в текущее известное кратчайшее расстояние от стартовой вершины.
* queue — очередь вершин, ожидающих обработки.

Логика:

1. Все расстояния инициализируются как бесконечность (Integer.MAX\_VALUE), кроме начальной вершины (0).
2. В цикле извлекается вершина из очереди.
3. Для каждой смежной вершины проверяется, не даст ли текущее ребро более короткий путь.
4. Если найден более короткий путь — обновляется расстояние и вершина снова добавляется в очередь.

**Метод main()**

* Определяет граф с четырьмя вершинами: **A, B, C, D**.
* Устанавливает связи и веса рёбер.
* Запускает dijkstra(graph, "A") — поиск кратчайших путей из вершины A.

