Algorithms With Java

Week 1 Day 1: Поиск в графах

Академия Ковалевского

Содержание

1. Теория

Поиск в графах (обход графов)

Два наиболее популярных алгоритма обхода графов: в глубину (depth first) и в ширину (breadth first). Обходы деревьев нужны для того, чтобы оптимально быстро найти необходимый элемент в дереве.

Алгоритмы в глубину имеют три типа обходов:

- pre-order (сначала проверяются родители),
- in-order (сначала проверяем всех детей),
- post-order (порядок такой, в каком дерево строилось).

Поиск в глубину на википедии.

Алгоритм поиска в ширину — сначала по рут ноде, потом по всем ее детям, потом по всем детям детей и так до самого последнего. Поиск в ширину на википедии.

Интересная статья, есть перевод на Хабр — поиск в глубину и поиск в ширину простыми словами.

2. Практическая работа

GraphHelper2

Класс **GraphHelper2**, в котором нужно реализовать два метода:

1. Метод includesDepthFirstSearch(), который производит поиск элемента по дереву в глубину.
2. Метод includesBreathFirstSearch(), который производит поиск элемента по дереву в ширину.

```
package academy.kovalevskyi.algorithms.week1.day1;

import academy.kovalevskyi.algorithms.week1.day0.GraphBinaryNode;
import java.util.LinkedList;
import java.util.Queue;

public class GraphHelper2 {

  public static boolean includesDepthFirstSearch(GraphBinaryNode<?> root,
Object value) {
    // TO DO
  }

  public static boolean includesBreathFirstSearch(GraphBinaryNode<?> root,
Object value) {
    // TO DO
  }
}
```

XO

Данная задача — это еще одна задача с реального интервью в большие компании. Эта задача уже не задается на самих интервью, она была рассчитана на mid+ девелоперов. На задачу уделялось 45 минут, однако в силу объемности задачи не ожидалось, что интервьюер закончит ее полностью за 45 минут. У вас есть сутки и задача должна быть закончено полностью.

При реализации следующих трёх классов вам необходимо **применить теорию графов (думайте над этой задачей как над проблемой решаемой графами)** для генерации всех конечных состояний игры крестики-нолики.

Конечное состояние игры — это такое состояние поля игры, которое:

- достижимо по правилам игры,
- имеет победителя или
- ничья

Пример корректного финального состояния:

X	O	X
O	X	O
		X

Пример некорректного финального состояния:

X	O	X
O	X	O
	O	X

В отличии от прошлого примера этот пример недостижим по правилам игры, так как “O” не может походить после того как уже выиграл “X”.

Для упрощения предположим, что первым всегда ходит “X”, поле всегда 3 на 3 (не нужно пытаться сделать решение для общего случая).

XoHelper

Класс **XoHelper** является основным классом, генерирующим все конечные состояния при помощи метода generateEndStates().

```
package academy.kovalevskyi.algorithms.week1.day1;

import java.util.HashSet;
import java.util.Set;
import java.util.stream.IntStream;

public class XoHelper {
  public static Set<XoBoard> generateEndStates() {
    // TO DO
  }
}
```

XoFigure

Класс **XoFigure** служит для хранения фигурок игры.

```
package academy.kovalevskyi.algorithms.week1.day1;

public enum XoFigure {
  figureX, figureO
}
```

XoBoard

Класс **XoBoard**, в котором реализовать:

1. Конструктор XoBoard() — конструктор, который создает пустое поле, также нужно создать конструктор копирования, который создает полную копию поля предыдущей доски.
2. Метод getFigure() — возвращает фигурку установленную в данной координате (или null).
3. Метод setFigure() — устанавливает фигуру в заданную координату.
4. Метод hasWinner() — возвращает фигурку победителя, если на доске есть победитель (или null).
5. Метод tie() — возвращает true, если на доске ничья.

6. Метод equals() — стандартный.
7. Метод hashCode() — стандартный.

```
package academy.kovalevskyi.algorithms.week1.day1;

import java.util.Arrays;
import java.util.stream.IntStream;

public class XoBoard {
  public XoBoard() {}

  public XoBoard(XoBoard copy) {
    // TO DO
  }

  public XoFigure getFigure(int x, int y) {
    // TO DO
  }

  public void setFigure(int x, int y, XoFigure figure) {
    // TO DO
  }

  public XoFigure hasWinner() {
    // TO DO
  }

  public boolean tie() {
    // TO DO
  }

  @Override
  public boolean equals(Object o) {
    // TO DO
  }

  @Override
  public int hashCode() {
    // TO DO
  }

  // дополнительный метод toString() может пригодится в этом классе, но он не
обязателен

}
```