Algorithms With Java

Week 1 Day 0: Binary tree

Академия Ковалевского

Содержание

1. Теория

Бинарные деревья и Графы

Тема графов одна из САМЫХ важных в программировании, так как практически каждая задача так или иначе может быть представлена графами, посему этим днем мы начинаем погружаться в увлекательнейшею пучину добра и радости.

Дерево — структура данных (graph, граф), представляющая собой древовидную структуру в виде набора связанных узлов.

Хорошая статья о древовидных структурах данных, а это перевод статьи на русский.

Двоичное дерево состоит из узлов (вершин) — записей вида (left, right, value):

- value — некоторые данные, привязанные к узлу,
- left и right — ссылки на узлы, являющиеся детьми данного узла — левый и правый сыновья соответственно.

Описание бинарного дерева на википедии, пригодится для понимания сегодняшних задач.

Перед тем как приступать к практическому заданию — вспоминайте, что такое java record.

2. Практическая работа

GraphBinaryNode

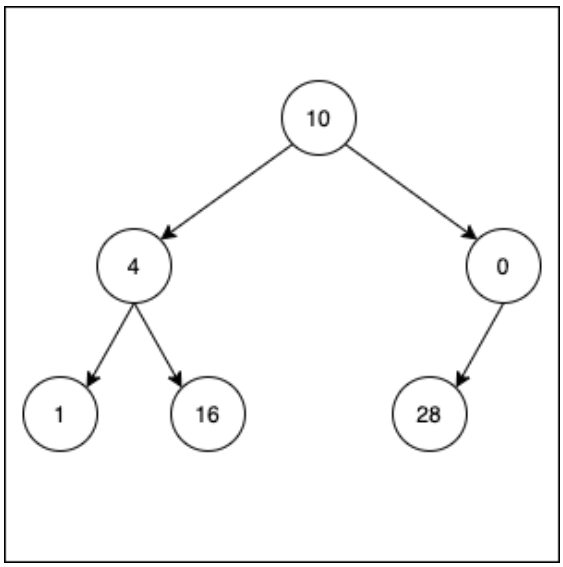
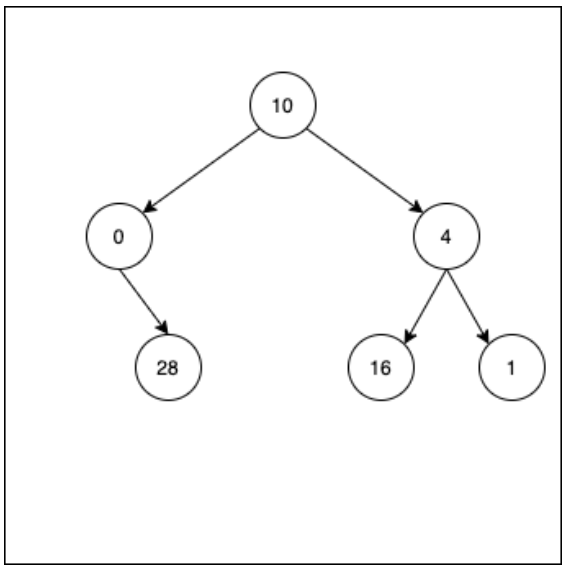
Это java record. Он хранит в себе:

- Ссылки на ветки:
  - Левую ноду,
  - Правую ноду.
- Некое значение (value) любого типа.

GraphHelper

Класс **GraphHelper**, в котором необходимо реализовать два метода:

1. Метод equals(), который проверит — равны ли графы.
2. Метод invertGraph(), который инвертирует входящий граф. Пример инвертирования:

Original	Inverted
	

Сигнатура GraphHelper:

```
package academy.kovalevskyi.algorithms.week1.day0;

public class GraphHelper {
  public static boolean equals(GraphBinaryNode<?> left, GraphBinaryNode<?> right) {
    //  TO DO
  }

  public static <T> GraphBinaryNode<T> invertGraph(GraphBinaryNode<T> root) {
    // TO DO
  }
}
```

IntGraphHelper

Теперь мы перейдем к задачам с бинарными графами, у которых каждая нода хранит число. Это позволит нам создать наше первое “отсортированное дерево” и написать свой первый простой алгоритм поиска по дереву.

Сигнатура IntGraphHelper:

```
package academy.kovalevskyi.algorithms.week1.day0;

public class IntGraphHelper {
  public static GraphBinaryNode<Integer> createNode(Integer value) {
    // TODO
  }

  public static GraphBinaryNode<Integer> addNode(GraphBinaryNode<Integer> root, Integer
value) {
    // TODO
  }

  public static boolean contains(GraphBinaryNode<Integer> root, Integer value) {
   // TODO
  }
}
```

Класс **IntGraphHelper**, в котором нужно реализовать:

1. Метод createNode() — создает новый элемент (node) с интовым значением и пустыми потомками.
2. Метод addNode() — рекурсивный метод, который создает новый граф,

добавляя новый элемент в новосозданный граф. **Важно**:

- Этот метод создает новый граф, а не меняет текущий (т.е. все изменения immutable)
- Этот метод добавляет новое значение в отсортированном виде! То есть на каждой node он будет сравнивать значение текущей node с новой node и принимать решение в левую или правую часть графа добавить новую node.

3. Метод contains() — рекурсивный метод, который проходит по графу и проверяет наличие заданного числа в графе. **Важно**:
   - Подумайте — какая средняя, худшая и лучшая сложность у данного решения? Приведите пример, в котором у графа будет самая худшая сложность.
   - Почему именно такая худшая сложность?
   - Можно ли как-то “перераспределить” элементы графа из придуманного примера, чтобы улучшить худшую сложность?

OptimaFinder

Эта задача ранее использовалась на интервью в нескольких больших компаниях (ныне уже не используется). Звучит она так:

Представьте, что есть некая программа. Неизвестно, что она делает, такой себе черный ящик. На вход черного ящика можно подать *Double*, который говорит программе, сколько виртуальных CPU можно использовать. Для этой задачи, предположим, что можно использовать любой кусок CPU, например, 0.0002, то есть любой *Double*. И хоть в реальной жизни редко выделяешь CPU меньше чем 0.5, для текущей задачи, предположим, что CPU можно задать любым числом типа *Double*. На выходе наш черный ящик выдает *Double*-число, которое говорит, сколько секунд у черного ящика заняло, чтобы закончить задачу.

Точно известно, что чем больше CPU давать черному ящику, тем меньше времени у него займет выполнение задачи. Пример:

- 2.0 => 2.0
- 3.0 => 1.9
- 4.5 => 1.2
- 6.9 => 0.8

Однако, мы точно знаем, что есть некоторое количество CPU, после которого черный ящик не сможет ускорять свою работу. Есть некоторый предел

распараллеливания, по достижению которого скорость работы начнет замедляться.

- 2.0 => 2.0
- 3.0 => 1.9
- 4.5 => 1.2
- 6.9 => 0.8
- 7.1 => 0.7
- **7.2 => 0.6**
- 7.3 => 0.7

Как видно из данного примера, у нашего черного ящика оптимум находится, когда он использует где-то 7.2 CPU. Важно: точный размер CPU может быть любым *Double*, и может быть даже не вычислим, в данном примере мы можем сказать, что количество CPU для оптимум равно 7.2 +/- 0.1. Или, иными словами, с точностью 0.1 ответ 7.2.

Задача — реализовать программу, которая для любого черного ящика будет находить оптимальное количество CPU (с заданной точностью), при которой время выполнения работы черным ящиком будем минимально.

Сигнатура OptimaFinder:

```
package academy.kovalevskyi.algorithms.week1.day0;

import java.util.function.Function;

public class OptimaFinder {

 public static double findOptima(
     Function<Double, Double> f, double start, double end, double precision)
{
   // TODO
   }
 }
}
```

На вход передается:

- **f** — это наша функция, которая представляет черный ящик и на вход получает количество CPU, а на выходе дает время в секундах. **Важно**:
    - Эта функция возвращает значение мгновенно
    - Сложность функции O(1)

- **start**/**end** — минимально и максимально допустимое значение CPU, ни при каких условиях нельзя вызывать функцию f, передавая туда значение за пределами этого диапазона!
- **precision** — точность ответа. То есть если верный ответ 1.2, а точность 0.1, то верным будет считаться любой ответ от 1.1 до 1.3 включительно.

Финальный ответ должен иметь логарифмическую сложность.

Если не знаете как начать:

- Подумайте почему эта задача находится в дне, где есть задача по поиску по отсортированному графу (где на каждой ноде нужно принять решение смотреть влево или вправо)
- Не получается с *double* — начните делать с *int*
- Не получается с *int* — начните делать методом полного перебора интов