

Intelligent Systems

Beka Bekerí

Enrique Valverde Soriano

Raquel Del Castillo Pérez

Task 1.

First of all, we have decided to create two classes in order to represent the nodes and the edges, this classes contain all the information about each type of the graph elements.

In order to store these elements, we have chosen to use the data structure *list* having two lists, one is used to store node objects and the other is used to store edge objects.

The first method `BelongNode` takes the ID of an OSM node and checks by using the node list previously created if the ID is in the list of nodes meaning that the node belongs to our graph and returning true or false if the ID is not found.

The method `positionNode` takes the ID of an OSM node and uses an if statement in order to check if the node is in the list of nodes and if the node exists it prints the coordinates that are stored in that node object.

The last but not the least, `adjacentNode` takes the ID of an OSM node and checks if the node exists and if that happens it checks in the list of edges which of those edges has the node as the source node, if that condition is satisfied it adds the edge to the list of adjacent edges of that node and prints the results.

Task 2.

For this task we have been asked to implement the main structure of the program, focusing on the class `TreeNode` in order to test different ordered data structures that will contain objects of the class `TreeNode` ordered by a random variable named `f`, the objective of the test is to determine which data structure will handle better the problem, for this purpose a stress test has been implemented using different sizes for the data structures in order to obtain the data structure with the biggest size possible, it is important also to consider that another criteria to determine if the data structure is adequate are the access time and if the data structure have an implicit method for ordering.

The data structures that we are going to use in the test are the `SortedSet`, `PriorityQueue` and `LinkedList`, `SortedSet` and `PriorityQueue` have implicit methods for ordering (comparators) that made this two structures an important option, for the `LinkedList` there is no implicit method for comparing the inserted elements so, it is needed the implementation of a method.

In the following image, the results of the three tests are shown:

```
Time LinkedList 100 nodes: 0 ms
Time LinkedList 1000 nodes: 0 ms
Time LinkedList 10000 nodes: 0 ms
Time LinkedList 100000 nodes: 13 ms
Time LinkedList 200000 nodes: 17 ms
Time LinkedList 500000 nodes: 74 ms
Time LinkedList 1000000 nodes: 199 ms
Time LinkedList 2000000 nodes: 1100 ms
Time LinkedList 5000000 nodes: 4061 ms
Time LinkedList 10000000 nodes: 5390 ms

Time SortedSet 100 nodes: 0 ms
Time SortedSet 1000 nodes: 2 ms
Time SortedSet 10000 nodes: 12 ms
Time SortedSet 100000 nodes: 67 ms
Time SortedSet 200000 nodes: 119 ms
Time SortedSet 500000 nodes: 400 ms
Time SortedSet 1000000 nodes: 1001 ms
Time SortedSet 2000000 nodes: 2018 ms
Time SortedSet 5000000 nodes: 5575 ms
Time SortedSet 10000000 nodes: 11595 ms
Time SortedSet 20000000 nodes: 29020 ms
Time SortedSet 50000000 nodes: 66731 ms

Time PriorityQueue 100 nodes: 0 ms
Time PriorityQueue 1000 nodes: 1 ms
Time PriorityQueue 10000 nodes: 1 ms
Time PriorityQueue 100000 nodes: 18 ms
Time PriorityQueue 200000 nodes: 13 ms
Time PriorityQueue 500000 nodes: 26 ms
Time PriorityQueue 1000000 nodes: 51 ms
Time PriorityQueue 2000000 nodes: 389 ms
Time PriorityQueue 5000000 nodes: 263 ms
Time PriorityQueue 10000000 nodes: 1061 ms
Time PriorityQueue 20000000 nodes: 4050 ms
Time PriorityQueue 50000000 nodes: 20637 ms
```

As we can see, the fastest one is the LinkedList, remember that here, we are comparing the insertion times, it is important to mention that as the PriorityQueue and the SortedSet have implicit methods, the insertion needs a little more time because of the comparisons that are made.

Looking at the image is easy to say that the linked list is the best option, but, there is a problem with this kind of structure, first of all, the size of it is not so big compared with the other data structures we are comparing and there is also a problem with the access times that is a big problem due to the fact that we will need to retrieve data from the data structure very often.

```
Time LinkedList 100 nodes: 3 ms
Time LinkedList 1000 nodes: 1 ms
Time LinkedList 10000 nodes: 0 ms
Time LinkedList 100000 nodes: 33 ms
Time LinkedList 200000 nodes: 21 ms
Time LinkedList 500000 nodes: 90 ms
Time LinkedList 1000000 nodes: 258 ms
Time LinkedList 2000000 nodes: 1721 ms
Time LinkedList 5000000 nodes: 5631 ms
Time LinkedList 10000000 nodes: 2619 ms
Time LinkedList 20000000 nodes: 17672 ms
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
    at Code.Main.main(Main.java:33)
```

This image shows the memory exception that comes with the use of LinkedList, it has very fast insertion times but the size problem is an important one and also the need of using a method in order to sort the elements that are contained in this data structure.

So, after all tests and a little bit of deliberation we have decided that we are going to use PriorityQueue in order to store the elements of the final program, PriorityQueue uses an implicit method for sorting, has very fast access times and a big enough size in order to store all elements in bigger problems.

The **code for this task can be found in this repository:**

<https://github.com/BekaBekeri/SSInteligentes>