Programming Paradigms – Ruby Homework

Elisa Marengo Pietro Galliani

1st Semester 2020/21

In this exercise, we want to implement a family tree, where each node represents a member of the family and can have children. Each child is in turn a member of the family and can have children.

We also want to perform some operation on the elements of the family, like adding members, order them according to the year of birth, print them, or add a spouse.

Your task is to implement such a tree by implementing the properties and methods described in the following of this document.

Class Person. Implement the class *Person* which represents a person and must have the following properties

- a field `name`.
- a field `surname`.
- a field `year_birth`.
- a field `married_with` to store the name of the spouse.
- a field `children` that is an array of objects of type Person. The elements of the array represent the children of a person.
- a field `parent` that stores the reference to the parent node. (Note that the requirement is to store the reference to the parent, not a property like the name or suchlike). For the root, the value of the parent is `nil`.
- a class variable `num_people` to count how many objects of the class are created.

Methods of the class. The class implements the following methods

1. The constructor only takes four parameters: *i)* the name, *ii)* the surname, *iii)* the year of birth, and *iv)* the name of the spouse.

   The constructor will initialize the value of `parent` to nil. The value of the parent will be properly set in the method `add_child` (described in the following), i.e., when the node is added as child of another node.

The constructor is incrementing the value of `num_people` when a new member is created.

2. Implement a method `add_child` that adds a child taken as parameter, to the person the method is invoked on. The method also takes care of properly setting the parent for the child node.

   Additionally, the method should be implemented in such a way that the array of children of a node is ordered in non-decreasing order w.r.t. to the year of birth. Therefore, when adding a child it must be inserted in the right position.

3. Implement a method `add_spouse` that adds a spouse to the current person (i.e., the node on which the method is invoked).

   The method takes the name of the spouse as a parameter and an array of children which is not nil if the person already has children. In this case, the children are added to the children of the person the method is invoked on. Also in this case, we want the final array of children to be ordered in non-decreasing order w.r.t. the year of birth.

4. Redefine the method `to_s()` to print the information of a node in a suitable way. Note that `puts obj` uses `obj.to_s()` to convert the object to a string. Therefore, by redefining `to_s()` it is then possible to use just `puts obj` to obtained the desired readable output.

   The method `to_s()` returns the string to be printed, does not print it inside the method (you can use `+` for string concatenation).

   The information to be printed should be as follow:

   - Print the information on the current node including the name and surname of the parent if this is not `nil`. Also for the spouse, it should print the value for `married_with` only if not nil.
   - The print should look something like this for the root

     ```
     name:  tom rossi
     year:  1900
     married_with:  rose
     number of children:  2
     ```

     And the other nodes:

     ```
     name:  mary rossi child of tom rossi
     year:  1932
     married_with:  fred
     number of children:  1
     ```

5. A method `traverse_bfs` which traverses the tree implementing a recursive **breadth-first-search**. While traversing the tree, the `traverse_bfs` prints the information of the node using the puts. The puts will use the method `to_s()` that you redefined.

6. Write another version of the `traverse_bfs` called `each` which takes a *code block* and executes the code block on each element of the tree following a breath-first-search strategy.

7. Import the module `Enumerable` and implement the missing method that is needed to sort the elements of the tree. Implement the method so that the elements are compared (and ordered) according to `year_birth`

8. Write a method that returns the number of elements that have been created `num_people`.

Test file. Write a test file that tests the methods that you implemented.

Create a family tree with some members and children. The family tree should be created in such a way to properly test the methods that you implemented. For instance, a tree with two levels is not adequate to test the breath-first-search. Also, adding children in an ascending orders would not test that your insertion of children in the right position works. The completeness of the tests will be part of the evaluation. Moreover, the correctness of your submission will not be evaluated based only on the tests that you submit, but additional tests may be performed.

Additionally, your tests should include the following

1. Test your `each` method using a code block for printing the elements of the tree.

2. Test your `each` method using a code block to search for a member in your tree. The member is searched by name. For instance, we may want to search for a member whose name is `mary`.

   In the code block, to stop the search once the first element with name `mary` is found you can use `break element` to stop the loop and return the variable `element`.

3. Test the method `sort` (provided by the `Enumerable` module) by printing the resulting tree. For instance, if `root` is the root element of the tree, use the command `puts root.sort`

Submission: You will have to submit two separate files:

1. One for the class Person described above.

2. One for the tests.

Deadline: 29 October 2020, 7 AM (= 1 hours before the lesson)