Trees

Table of Contents

1. Session 2 (lab): Trees (I)

1.1. Binary Trees

The goal of this exercise is to create your first linked binary tree, and get to know its methods: insertion, extraction, traverse the tree (pre-order, in-order, post-order) as well as comparison of trees and searches.

If you recall the recursive definition of a tree, a binary tree is empty or consists of a root element, with a node containing **exactly** two children trees (left subtree and right subtree).

Following this definition, you need two classes to define a binary tree:

1. The class defining the binary tree datatype itself.
2. The class defining its nodes.

The first one represents the tree and defines the public API. The second class allows the implementation through links or references among trees, and must not be part of the public API.

Section 1. BTree Interface

The `BTree` interface that will be used in this assignment is shown below.

```
public interface BTree<E> {
    static final int LEFT = 0;
    static final int RIGHT = 1;

    boolean isEmpty();

    E getInfo() throws BTreeException;
    BTree<E> getLeft() throws BTreeException;
    BTree<E> getRight() throws BTreeException;

    void insert(BTree<E> tree, int side) throws BTreeException;
    BTree<E> extract(int side) throws BTreeException;

    String toStringPreOrder();
    String toStringInOrder();
    String toStringPostOrder();
    String toString(); // pre-order

    int size();
    int height();

    boolean equals(BTree<E> tree);
    boolean find(BTree<E> tree);
}
```

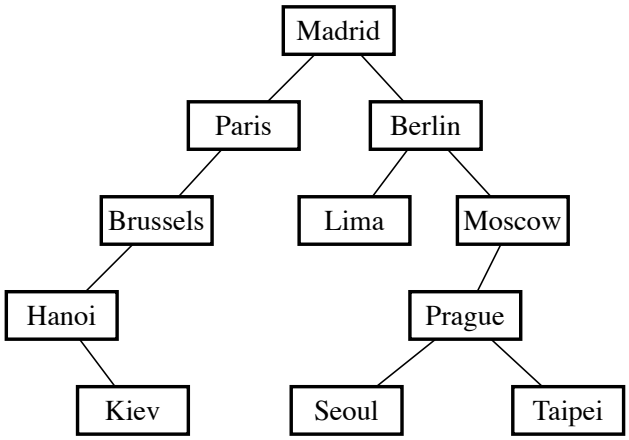
Add comments to the interface explaining what the attributes are for. Explain also what each method does and what it is useful for, the input arguments and the return values. Explain as well under which conditions exceptions are thrown.

Section 2. Use Example

Suppose you work for the development department of a travel agency. The agency offers several flights to different cities of the world. In each city, the client can choose between two different destinations. However, because of weather issues, some of these flights are closed.

Suppose also that your department already has an implementation of the interface described in the previous section, called `LBTree` (LinkedBinaryTree). You are asked to create an application to manage the flight offers explained before.

Answer (in a piece of paper) the following questions regarding the applications. Suppose that you have a flights tree like the following one:



1. What is the size of the tree which root is Madrid?
2. What is the elements list of the tree which root is Madrid (in pre-order)?
3. What is the size of the tree which root is Moscow?
4. What is the elements list of the tree which root is Moscow (in post-order)?

Once you have answered the questions, keep on reading.

Write (once again, in a piece of paper) the main method of this application. This main method creates the tree shown in the picture above, and prints in the standard output the answers to the previous questions.

You can suppose that the elements stored in the tree are `String`s, which contain the name of the cities. You can also suppose that the LBTree class has two constructors: `LBTree(E info)` and `LBTree()`.

Pay special attention to the management of the exceptions, an also on how to make the tree insertion, so that the result is the one in the figure.

You can see an example of the execution of the application below:

```
Warning: SPOILERS BELLOW!
```

```
; java Cities
Madrid tree size = 11
Madrid tree = Madrid Paris Brussels Hanoi Kiev Berlin Lima Moscow Prague Seoul Taipei
Moscow tree size = 4
Moscow tree = Seoul Taipei Prague Moscow
```

Section 3. The LBTree and LBNode classes

Write the `LBTree<E>` class, which implements the `BTree` interface. Code also the `LBNode<E>` class needed by `LBTree<E>` class, as well as the `BTreeException` class.

Start with the `BTreeException` class. This class extends from `Exception`, and it simply has a constructor with an only input argument, of String type.

Next, write the `LBNode<E>` class starting from the following template:

```
  class LBNode<E> {

    /* your attributes here */

    LBNode(E info, BTree<E> left, BTree<E> right) {
        /* your code here */
    }

    E getInfo() {
        /* your code here */
    }

    void setInfo(E info) {
        /* your code here */
    }

    BTree<E> getLeft() {
        /* your code here */
    }

    void setLeft(BTree<E> left) {
        /* your code here */
    }

    BTree<E> getRight() {
        /* your code here */
    }

    void setRight(BTree<E> right) {
        /* your code here */
    }
}
```

Finally, write the `LBTree<E>` class, which must have two constructors:

1. `LBTree()`: creates an empty tree, i.e., a tree which root is `null`.
2. `LBTree(E info)`: creates a tree of size 1, with a node that stores `info`, a two empty subtrees.

You can test your solution with the LBTreeTest.java class, which performs thorough tests with every method of the `LBTree` class.

In order to understand the errors shown by this class, you need to understand first how the testing code works.

Next, an execution example of this class is shown, supposing that there already exists a working implementation of `LBTree`:

```
; java LBTreeTest
testing isEmpty: OK
testing getInfo: OK
testing insert: OK
testing getLeft: OK
testing getRight: OK
testing extract: OK
testing toStringPreOrder: OK
testing toStringInOrder: OK
testing toStringPostOrder: OK
testing toString: OK
testing size: OK
testing height: OK
testing equals: OK
testing find: OK
```

Section 4. Compatibility with other binary trees

Let's go back to the travel agency example. Suppose that another department of your business uses a different implementation of `BTree`, called `ABTree`. In this implementation, the nodes (called `ABNode`) use an array to store the subtrees, instead of using references like in our implementation.

Therefore, their trees root is a reference to an `ABNode`, which is besides called "wurzel" (instead of "root"), since this department is in Germany.

Answer the following questions:

1. Could you insert a new tree of type `ABTree` with some new cities, which comes from the other deparment, to the left of Brussels in your `LBTree` that started in Madrid?
2. If you could, could you search for a mixed subtree (with `LBNode` and `ABNode` nodes) in the tree resulting from the previous insertion?

2. Homework

2.1. N-ary trees and file systems

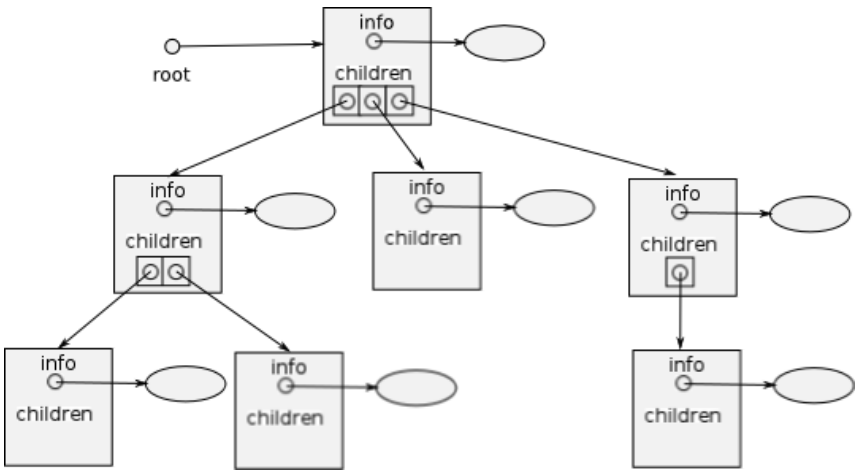
Objective

To study another kind of more generic trees than binary ones, by proposing an application example: file systems. The students are supposed to learn how to generalize the operations with binary trees in order to apply them to the case of N-ary trees.

N-ary trees and file systems

Up to now we have been working with binary trees, but they are not the only kind of trees we can find. Sometimes we need more flexible trees that allow us to have, for each node, N children nodes, where N has not to be exactly two and can be a different value for each node. This data structure is known as N-ary tree, and it is shown in Figure 1. As you can see, each tree node contains a reference to the information stored in it (`info`), as well as a set of references to the children nodes (`children`). In order to access every tree node we just need a reference to the root node (`root`), as in the case of binary trees.

Figure 1. Graphical representation of a N-ary tree



In this exercise we are going to see an example in which N-ary trees are necessary: file systems. Let's suppose that we have the following file system (also known as directory tree):

```
C:
  |_Program Files
    |_Eclipse
    |_Java
  |_My Documents
    |_Images
    |_Music
    |_Videos
  |_ProgSis
```

```
|_Project
  |_Module1
  |_Module2
```

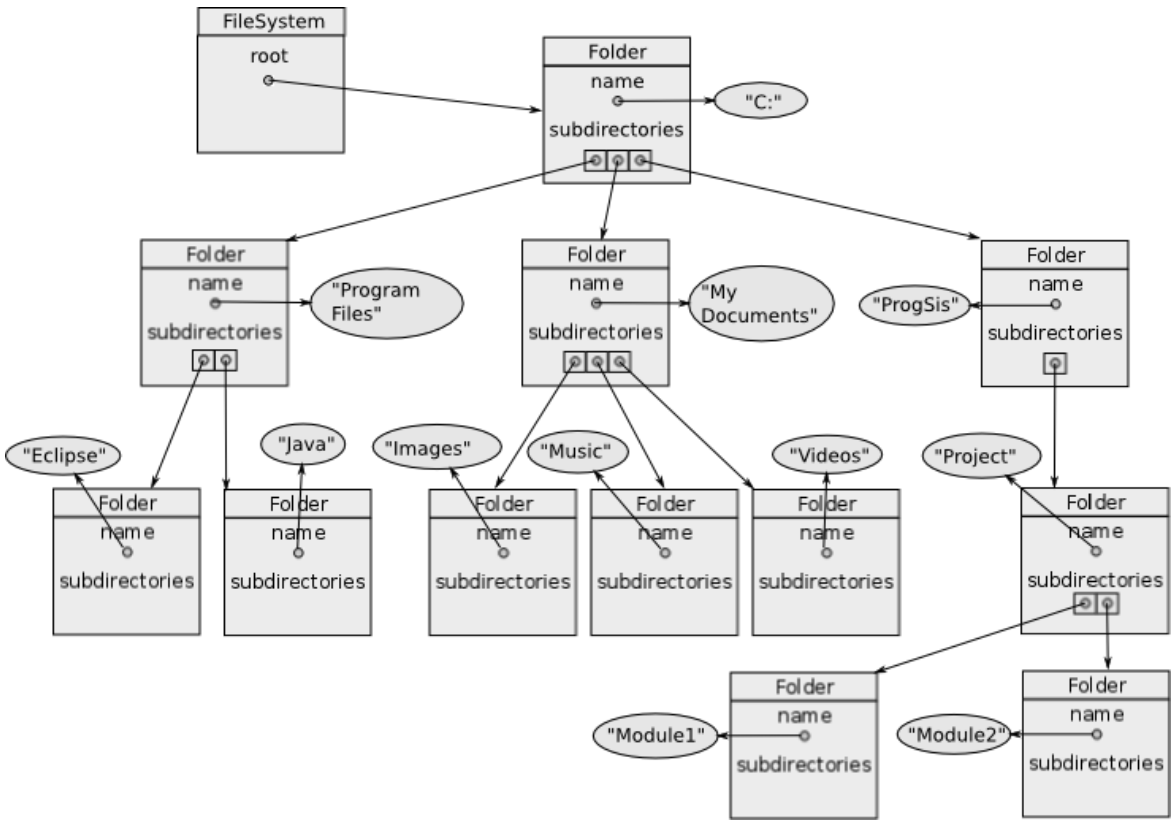
As its name suggests, every directory or folder (in our case we are going to simplify the problem by ignoring the files) is stored following a tree structure: there is a root folder (C:) which contains many folders, each one of the containing many more and so on. In order to create and manage a file system, we are going to take the generic data structure shown in Figure 1, and we are going to make it specific to the case we are studying. The nodes in the image will be our folders. Each folder will be represented by an object of the `Folder` class. This class has two attributes:

- `name` is an attribute of type `String` which stores the folder's name
- `subdirectories` is an attribute of type `ArrayList` which stores the subdirectories (objects of type `Folder`) the folder contains.

In order to represent the file system, we will use the `FileSystem` class, which plays the role of tree since it is in charge of storing the reference to the root folder (`root` attribute of type `Folder`).

Figure 2 represents the sample file system shown before by using the Java objects we will be dealing with during the exercise.

Figure 2. Graphical representation of a file system modeled with Java objects



Exercise

In order to make the exercise easier, we provide part of the `Folder` class, as well as the structure of the `FileSystem` class. Both files can be downloaded from the following links:

- Folder.java
- FileSystem.java

First of all you will practice the use of `ArrayList` objects (if you do not know how to use them, check the API). In order to do that you have to implement the following methods of `Folder` class:

1. `public Folder addSubdirectory(String folderName)` : adds a new folder, which name is `folderName` , to the set of subdirectories and return a reference to the new folder.
2. `public void printSubdirectories()` : prints in the screen the name of every subdirectory contained by the folder, with the following format: subdirectory1 subdirectory2 ... subdirectoryN. Only direct subdirectories (children nodes) will be printed, ignoring the subdirectories that can be contained in each one of them.

Before we move on, make sure that the methods you just implemented work as expected. In order to do that, create a `FileSystemTest.java` class and check if they behave as supposed to.

Once we have clear how `Folder` objects behave and how to traverse an object of `ArrayList` class, we can start dealing with the file system itself. You have to implement the following methods of the `FileSystem` class (besides making the necessary tests in order to check the proper working of each one with the `FileSystemTest` class you created earlier):

1. `public Folder searchFolder(Folder root, String folderName)` : search the folder which name is `folderName` in the whole tree. If the folder exists, it returns a reference to it, or null otherwise. *(Hint: this method is recursive since, for each foder, you have to search in its subdirectories, in the subdirectories of these ones and so on until you find the folder or you have checked every node in the tree. It might be useful to use an auxiliary method* `private Folder searchFolder(Folder root, String folderName)` *, where* `root` *is the root folder of each subtree over which the search is going to be applied).*

2. `public Folder addNewFolder(String parentFolderName, String newFolderName)`: creates a new folder which name is `newFolderName`, and adds it to the subdirectories set of the folder with name `parentFolderName`. If there is already a folder in the tree with the name `newFolderName` or if the folder `parentFolderName` does not exist, then the method does nothing and null is returned. Otherwise, the new folder is created and added, and a reference to it returned.

3. `public void printFileSystem()` : prints in screen the file system structure with the following format:

```
C:
 |_Program Files
   |_Eclipse
   |_Java
 |_My Documentos
   |_Images
   |_Music
   |_Videos
 |_ProgSis
   |_Project
     |_Module1
     |_Module2
```

(Hint: this method is also recursive. Moreover, the number of white spaces before each name has much to do with the tree level where the folder is stored. It might be useful to use an auxiliary method `private void printFileSystem(Folder root, int level)` *with which the recursive calls will be made.*