UNIVERSITÉ DU LUXEMBOURG

Binary Tree

Programming Fundamentals 2

Goals

- ✯ Manipulation and implementation of a binary tree data structure.
- ✯ Breadth-First Search (BFS)
- ✯ Depth-First Search (DFS)
- ✯ **Relevant videos**:
  - Binary Tree Data Structure.

Deliverables

1. The code on your Github repository generated by clicking here: `https://classroom.github.com/a/p98fwRqg`
2. **Reviewer**: Maria Hartmann (mhart01 on Github).
3. **Automated testing** using JUnit and unit testing.

Exercise 1 – From scratch

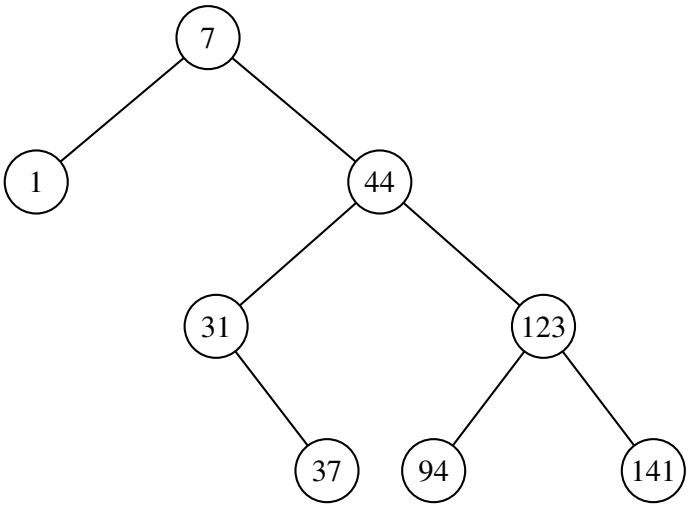
In this exercise, you are asked to implement a binary search tree. It means that the tree must store different entries which are pairs key/value. The data must be organised so that for any node of the graph with a key `k`, any node contained in the left subtree has a key lower than `k` (keys must be comparable), and any node contained in the right subtree has a key greater than `k`. Your implementation must consider any type for the keys and values. The only restriction is that it should not be allowed to add in the table a pair key/value with a key `null`. The reason is that keys must be compared between themselves for searching elements. A key `null` would not thus be comparable with other keys.

1. In file `BinarySearchTree.java`, implement a binary search tree from the template provided. The three main methods for a hash table are the following.
   - `public V get(K key)` Return the value associated with the given key, and `null` if the given key is not present in the tree.
   - `public V put(K key, V value)` Associate the given value with the given key in the tree. If the key is already present in the tree, this method changes the associated value and returns the former value; otherwise a new pair key/value is added in the tree and this method returns `null`.
   - `public V remove(K key)` Remove the pair key/value associated with the given key if such an entry is already in the tree. This method returns the value associated with the given key if the latter exists in the tree, and `null` otherwise.

For methods `put` and `remove`, you may modify the structure of the tree. It is very important to keep the most important property of a binary search tree, i.e. any key in the left subtree must be lower and any key in the right subtree must be greater.

2. Test your code with a file `BinarySearchTreeTest.java` where you can implement your unit tests.

Exercise 2 – Visualisation



keys (`Integer`)	values (`String`)
1	"Bulbasaur"
7	"Squirtle"
31	"Nidoqueen"
37	"Vulpix"
44	"Gloom"
94	"Gengar"
123	"Scyther"
141	"Kabutops"

Figure 1: Example of binary tree shown with keys on the left and mapping between keys and values on the right

1. Implement the method `public String toString()` in class `BinaryTree` which returns a string description of the content of the tree. The format of the string should make pairs key/value appear in the order obtained by exploring the tree with a Depth-First Search (DFS). For instance, if you consider the tree represented in Figure 1, the output should be:

```
7:  Squirtle
   |_ 1:  Bulbasaur
      |_ null
      |_ null
   |_ 44:  Gloom
      |_ 31:  Nidoqueen
         |_ null
         |_ 37:  Vulpix
            |_ null
            |_ null
      |_ 123:  Scyther
         |_ 94:  Gengar
            |_ null
            |_ null
         |_ 141:  Kabutops
            |_ null
            |_ null
```

2. Add a method `public List<K> keys()` which returns a list of every key stored in the tree (you can use the list data structure provided by Java). The keys must appear in the list in the same order as if you traverse the tree with a Breadth-First Search (BFS). So with the example shown in Figure 1, the list should be: `[7, 1, 44, 31, 123, 37, 94, 141]`