Trees

Table of Contents

1. Session 4 (lab): Trees (II)

1.1. Binary Search Trees

In the previous lab session you were working with binary trees and their basic operations: creation, insertion, extraction, search and traversing the tree. This we will go a step further and study a specific type of binary tree: binary search trees.

Section 1. Skeleton of a binary search tree

If you recall from theory class, a binary search tree is a sorted binary tree (it has two subtrees, left and right). This means that the nodes of the tree follow certain order. If we want to sort the nodes, we will need a key that serves to indicate which node comes first and which one later.

The nodes of a binary search node are sorted in such way that, for each node, the keys of the nodes in its left subtree are lower (or equal), whilst the keys of the nodes in the right subtree are greater (or equal).

Bearing these concepts in mind, and using the concepts you learnt in the previous session with binary trees, we will implement a binary search tree.

More precisely, we will write an implementation of the `BSTree<E>` (Binary Search Node) interface shown below. For this implementation we will use linked nodes, as in the previous session. Therefore, our binary search tree will be represented by the `LBSTree<E>` class.

```
public interface BSTree<E> {

    boolean isEmpty();

    E getInfo();
    Comparable getKey();
    BSTree<E> getLeft();
    BSTree<E> getRight();

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

    void insert(Comparable key, E info);

    BSTree<E> search(Comparable key);
}
```

Before to start implementing the `LBSTree<E>` class, we need to define the class representing the nodes the tree is made of. Complete the following skeleton for the `LBSNode<E>` class, which will be in charge of representing the tree nodes.

```
public class LBSNode<E> {

    private E info;
        private Comparable key;
        private BSTree<E> right;
        private BSTree<E> left;
    public LBSNode(Comparable key, E info, BSTree<E> left, BSTree<E> right;) {
        /* your code here */
    }

    /** Get and set methods */

    public String toString() {
        if(info != null) {
            return info.toString();
        } else {
            return "";
        }
    }
}
```

Pay special attention to the `key` attribute. Notice that its type is `Comparable`. Take a look at this interface API. Which method does any `key` need to implement in order to correctly implement the interface? Take a look at `String` and `Integer` APIs. Do they implement the `Comparable` interface? Could they be used as `key`?

Next, we are going to actually implement the binary search tree. Observer carefully the `BSTree<E>` interface, and answer the following questions:

- Which methods are new or need to have a different implementation with respect to the binary tree case? Why?
- Notice that, in this case, we will not use exception (in order to simplify the implementation). What will the methods do when the tree does not have the requested information? Think about the answer for each of the related methods: `getInfo()`, `getKey()`, `getRight()`, `getLeft()` y `search(Comparable key)`.

Start implementing the `LBSTree<E>` class:

- Define its attribute(s).
- Define two constructors. The first one will create a tree of size 1 (with a node inside), which content (information and key) will be passed as arguments of the constructor. The second constructor will create an empty tree, and therefore will not receive any argument.
- Implement the methods defined by the interface, except `insert` and `search` methods, that will be the target of the next section.

Section 2. Inserting, extracting and searching in a binary search tree

So far, we have the skeleton of our `LBSTree<E>` class. In this section we will complete it, writing the two most important methods: `insert` y `search`.

- Implement the `void insert(Comparable key, E info)` method. It will insert the new information, stored in `info`, in the corresponding place of the binary search tree, taking into account the `key`. Remember that if `key` is lower than the current node's key, you will insert the node in its left subtree; if `key` is greater, you will insert the node in its right subtree; and if it is equal, you will overwrite the information in the node with the new `info`. This process is repeated for each node you traverse along the tree, until you reach a node which subtree where you want to add the new information is empty. Therefore, you will need to implement the method using recursion.
- Implement the `BSTree<E> search(Comparable key)` method. It will return the subtree which key is equal to `key`. If the key is not found, `null` must be returned. Take advantage of the binary search tree properties in order to search in the most efficient way.

Finally, check your implementation with the `LBSTreeTest` class.

Section 3. Using a binary search tree

Now that we have already implemented our binary search tree, we will see an example of use. In this case, we will create a phone agenda.

The key with which we will sort the agenda will be the name and surnames of each contact. The sorting will work in such a way that first, the first surname will be compared, if they are equal, the second surname will be compared, and if they are also equal, we will consider the name. In order to sort the names, we need a class that will be called `Name`. This class must implement the `Comparable` interface. Recall the `Comparable` interface API. Which method must the `Name` class implement in order to comply with that interface?

Complete the following skeleton so that the object compared (`this`) is alphabetically lower than the argument received by the comparison method, you will return -1; if they are the same, 0; and if it is greater, 1. Remember that a name is lower than other one if the first surname is lower (is alphabetically previous) than the first surname of the other name; if both first surnames are the same, we will compare the second surname; and if the second surnames are equal, we will compare the names.

```
class Name implements Comparable<Name> {

    private String name;
    private String firstSurname;
    private String secondSurname;

    public Name(String name, String firstSurname, String secondSurname) {
        this.name = name;
        this.firstSurname = firstSurname;
        this.secondSurname = secondSurname;
    }

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getFirstSurname() {
        return this.firstSurname;
    }

    public void setFirstsurame(String firstSurname) {
        this.firstSurname = firstSurname;
    }

    public String getSecondSurname() {
        return this.secondSurname;
    }

    public void setSecondSurname(String secondSurname) {
        this.secondSurname = secondSurname;
    }

    public String toString() {
        return this.firstSurname+" "+this.secondSurname+", "+this.name;
    }

    @Override
    public int compareTo(Name anotherName){
        /** Your code here */
    }
}
```

We already have the sorting key to be used in our agenda. Now, we need to store all the data of each contact of our agenda, for which we will use the `Contact` class. Complete the following skeleton with the needed code in the constructor, and adding the get and set methods for every attribute.

```
class Contact {

    private Name name;
    private int phoneNumber;
    private String postalAddress;

    public Contact(String name, String firstSurname, String secondSurname,
                    int phoneNumber, String postalAdress) {
        /** Your code here */
    }

    /** Get and set methods for every attribute */

    public String toString() {
        String contactStr = this.name.toString();
        contactStr = contactStr+"\nPhone: "+this.phoneNumber;
        contactStr = contactStr+"\nAddress: "+this.postalAddress;
        return contactStr;
    }
}
```

Finally, we will implemente our phone agenda, `PhoneAgenda` using a binary search tree, so that searching for a contact will be fast and efficient. Implement this class following these guidelines:

- Which attributes does `PhoneAgenda` need?
- Define and implement a constructor with no parameters, in which you will initialize the attributes you previously defined.
- Create a `void insert(String name, String firstSurname, String secondSurname, int phone, String address)` method. It will insert a new contact in the agenda. Remember that you have available a `Contact` class, where the contact's information is sotred, and a `Name` class, that will be the sorting key of the agenda.
- Implement a `Contact search(String name, String firstSurname, String secondSurname)` method. It will return an object of

`Contact` class containing the data about the contact being searched, or `null` if that person is not in the agenda.

- Implement a `void print()` method. It will print in the standard output the information of every contact in the agenda, sorted alphabetically. Before writing a single line of code, think which one of the three tree traversal methods (preorder, inorder, postorder) you must use.