Data Structures and Algorithms
25 May 2018
2017-2018

Student:

Problem 1 (2 points) – Suppose that BSTree is a class that implements a binary search tree. You can use its methods (you do not have to implement them!).

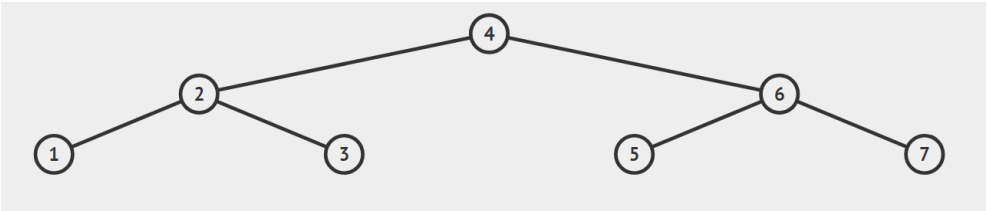
```
public class BSTree  {
        BSNode root;
        …
        public void insert(int key) {
        …
        }
        ….
}
```

Write a static method that takes a sorted (ascending) input of integers (you do not need to check that the array is sorted) and returns a balanced BSTree.

Some help:

- You do not have to obtain the balance factor of a node.
- You do not have to balance the tree. In other words, you do not have to apply any rotation or move nodes from a subtree to the other.
- Just think how to take advantage of the assumption that the array is already sorted.
- You must use divide and conquer approach.
- You can use the insert method of BSTree.
- You can write an auxiliary method that takes more input parameters (for example, indexes and a tree) if you need.

For example:
A = [1, 2, 3, 4, 5, 6, 7]
tree =



Solution:

```
public static BSTree arrayToBST(int[] data) {

      BSTree tree=new BSTree();
      if (data!=null && data.lenght>0)
            arrayToBST (data, 0, data.length - 1, tree);
      return tree;
}
```

```
private static void arrayToBST(int[] data, int start, int end,
BSTree tree) {
          if (start > end) return;
          int mid = (end + start) / 2;
          tree.insert(data[mid]);
          arrayToBST(data, start, pos – 1, tree);
          arrayToBST(data, pos + 1, end, tree);
}
```

Problem 2 (1 points) –
A) (0.5) Suppose that BSNode is a class that implements a binary search node. Write a non-static method, **equals**, which takes a BSNode object as input parameter and checks if this is equal to the invoking object. The method returns true if they are equal, and false otherwise. We consider that two nodes are equal when their attributes have the same value and their left and right subtrees are also equal ("equal" does not mean the same node).

```
public class BSNode {
        Integer key;
        String elem;
        BSTNode parent;
        BSTNode left;
        BSTNode right;
}

public boolean equals(BSNode obj) {
….
}
```

B) (0.25) Suppose that BSTree is a class that implements a binary search tree. Write a non-static method, **equals**, which takes a BSTree object as input parameter, and checks if this tree is equal to the invoking tree. You must use the equals method of BSNode to implement your solution.

```
public boolean equals(BSTree tree) {
…
}
```

C) (0.25) What is the time complexity of these methods in Big-O notation?. Explain.

Solution:

```
(a)
public boolean equals(BSTNode node){
     return equals(this,node);
}

public static boolean equals(BSTNode n1, BSTNode n2){
      if (n1 == null && n2 == null) return true;
```

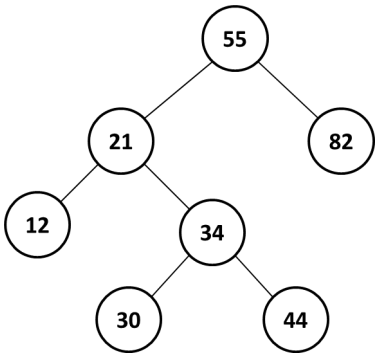
```
        if (n1 != null && n2 != null) {
                  return (n1.key == n2.key &&
             n1.elem.compareTo(n2.elem)==0
                            && equals(n1.left, n2.left)
                            && equals(n1.right, n2.right));

      }
      //otherwise
      return false;
}
```

Note: compareTo allows you to compare nulls. Equals throws an exception when one or both objects are null.

```
(b)
public boolean equals(BSTree tree){
      return equals(this,tree);
}

public static boolean equals(BSTree t1, BSTree t2){
       if (t1 == null && t2 == null) return true;

       if (t1 != null && t2 != null) {
            if (t1.root!= null) return t1.root.equals(t2.root);
            if (t2.root!= null) return t2.root.equals(t1.root);
      }

      //otherwise
      return false;
}
```

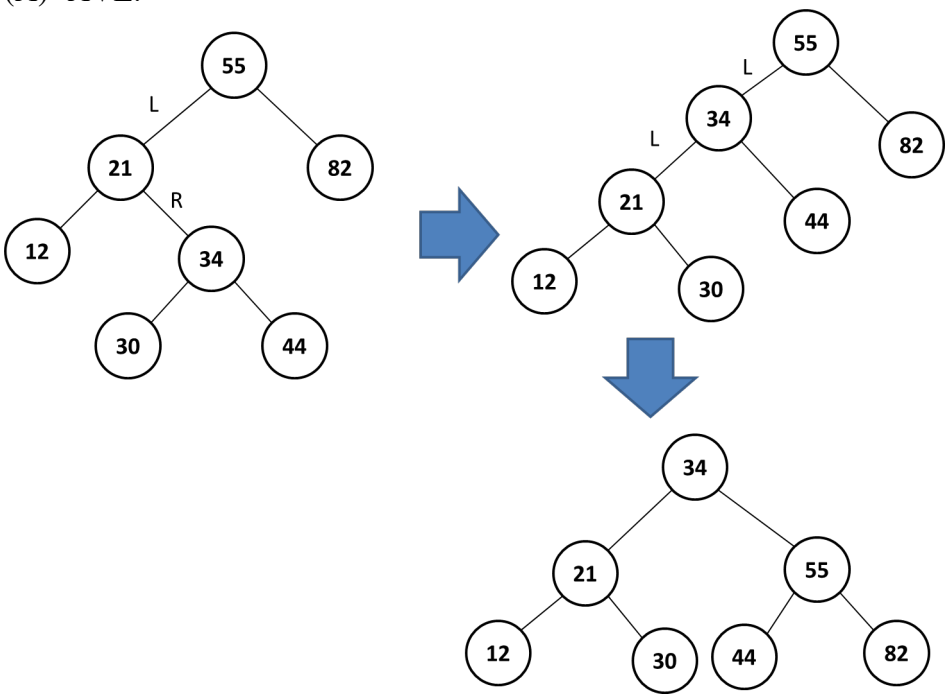
(c) Suppose that n is the size of the tree (or the subtree that hangs from the invoking node).
equals for BSNode has complexity O(n), because it has to traverse all nodes than hang from the invoking node.
Equals for BSTree also compexity O(n), because it has to traverse all the nodest than hang from the invoking tree.

Problem 4 (1 point):
A) (0.5) Balance the following binary search tree so that it satisfies the AVL property (height balance).
B) (0.5) Draw its size-balanced tree.



Solution:

(A)- AVL:



(B)- Size-balancing:
The resulting tree is the same obtained with the AVL rotations

Problem 3 (2 points).
A univeristy needs to manage all prerequisites among subjects in a degree. In this way, when a student wants to enrol in a given subject (for example, "Algorithms and Data Structures (ADS)"), the university must verify that he/she has completed all the required subjects for ADS (which are "Calculus" and "Programming"). Suppose that the number of subjects is 20.

1) (0.25) What data structure(s) is the most appropiate to represent the subjects and the requirement among them?. Explain. Are the relationships between subjects symmetrical?.
2) (0.25) Please, write the attributes and a constructor method (which takes an array with the name of the subjects as input parameter). In the constructor, no relation is defined yet.

3) (0.5) Write a non-static method, **allows()**, that takes the name of a subject as input parameter, and returns a list with all subjects that have as requirement the input subject. Suppose that the relationships among the subjects have already been stored in your structure. For example, ADS is required for the following subjects: *"Heuristics and optimization"*, *"Object-oriented programming"*, *"Knowledge Engineering"*, "Operating systems design", "*Formal Languages* and *Automata Theory*".
4) (0.5) Write a non-static method, **getRequiredSubjectsFor**, that takes the name of a subject as input parameter, and returns the list of subjects that a student should have passed to be able to enroll in the input subject. For example, to enrol to ADS, the student should have passed the following subjects: *"Calculus"*, *"Programming"*.
5) (0.5) Write a non-static method, **notAllowed**, that takes a list with all subjects passed by a student, and returns a list with all subjects in which the student can not enrol yet.

Note: In this problem, you can use Java classes such as LinkedList<String> or ArrayList<String> to implement your solution. If you prefer, you can use our classes Slist, DList, SQueue or Stack (you do not need to implement them, only you must remember the name of their methods, and must know how to use them).

Solution:

(a) A graph is an ideal data structure to represent the subjects and the relationships between them. The graph must be unweighted (the relationships have no weight) and directed (since the relationships are not symmetric). Regarding the implementation, since the maximum number of subjects is 20, using a matrix can be a good option, because it would use only 400 memory locations to store the possible relationships. It would also be a very good option if the graph is dense, that is, if there are many relationships between the subjects. On the other hand, the use of a list of adjacencies would also be an optimal option both in spatial complexity (you only use the necessary memory) and in time (most operations would have linear complexity).

(b)

```
public class RequirementGraph  {
	public String[] subjects;
	public int num;
	ArrayList<Integer>[] lstAdjacents;

	public RequirementGraph(String[] subjects) {
			this.subjects=subjects;
			this.num=subjects.length;
			//we must initialize each adjacent list
			lstAdjacents=new ArrayList[num];
			for (int i=0; i<num; i++) {
				lstAdjacents[i]=new ArrayList();
			}
	}
```

(c)

```
//returns all possible subjects that have as requirement the
```

```
input subject
	public ArrayList<String> allows(String subject) {
		int index=getIndex(subject);
		if (index==-1) return null;
		ArrayList<String> result=new ArrayList<String>();
		for (int i=0; i<lstAdjacents[index].size();i++) {
			int adj=lstAdjacents[index].get(i);
			result.add(getNameSubject(adj));
		}
		return result;
	}
```

Note: You can suppose that the methods **getIndex** and **getNameSubject**.

(d)

```
public ArrayList<String> getRequiredSubjectsFor(String subject) {
	int index=getIndex(subject);
	if (index==-1) return null;

	ArrayList<String> result=new ArrayList<String>();
	for (int i=0; i<num;i++) {
		for (int j=0; j<lstAdjacents[i].size();j++) {
			int adj=lstAdjacents[i].get(j);
			if (index==adj) result.add(getNameSubject(i));
		}
	}
	return result;
}
```

(e)

```
public ArrayList<String> notAllowed(ArrayList<String> lst) {
	if (lst==null) return null;
	ArrayList<String> notAllow=new ArrayList<String>();
	//traverse all subjects
	for (int i=0; i<subjects.length; i++) {
		String subject=subjects[i];
		//if the subject is already in the list, we do nothing
		//otherwise:
		if (!lst.contains(subject)) {
			//this subject has not been passed yet
			//we gets all required subject for this
			ArrayList<String>  req=getRequiredSubjectsFor(subject);
			boolean allowed=true;
			for (int j=0; j<req.size()&&allowed;j++) {
				String name=required.get(i);
				//if some of the required subjects is not in the list,
				//then this subject is not allowed. We must break the loop
				if (!lst.contains(name)) allowed=false;
			}
			if (!allowed) notAllow.add(subject);
			}
		}
	return notAllow;
}
```