CO518 Assignment 2 Due Date: 06/12/2019 Total Marks: 100

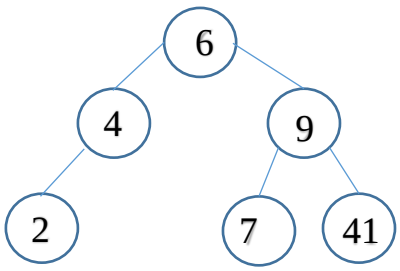
Task: implement binary search trees without any links. The way it works is as follows:

All tree nodes are stored in an array, the Comparable [] elems array in the skeleton implementation. Every index position in the array corresponds to a subtree. We keep a second array int[] sizes in which we store the sizes of those subtrees. A size of 0 means that the corresponding subtree is empty. It is also empty if the index is out of bound of the array.

The way indices are viewed as (sub-) trees works (or: should work) as follows:

- The entire tree (the root) corresponds to index 0.
- If a tree corresponds to position i then:
  - its A value is stored in elems[i], the provided private getKey(i) method will retrieve it as a value of type A
  - the size of that (sub-) tree is stored in sizes[i]
  - the left subtree of tree i corresponds to index position 2*i+1
  - the right subtree of tree i corresponds to index position 2*i+2

For example, consider this tree:



This would be represented as follows: elems = { 6, 4, 9, 2, null, 7, 41}, and sizes = {6, 2, 3, 1, 0, 1, 1}, though there might be additional null values at the end of the elems array and, correspondingly, additional 0 values at the end of the sizes array.

Your task is to realize the following operations, completing the provided skeletal code. Of course, the methods can call each other, and you can provide additional methods if it helps. The implementations should be efficient – generally half the marks are for correctness, the other for efficiency. Apart from grow() and insert(), all operations here can be realized in a run time proportional to the tree's depth d, and d=O(log(n)) on balanced trees of size n.

1. int **findIndex**(A x), private method. This is returning the index position of value x in the array (if it is there), or where we would put it (if it is not there, assuming the array were large enough). In our sample tree, findIndex(7) would return 5, findIndex(3) would return 8. [16 marks]
2. boolean **contains**(A x), public method. This return true iff x is in the tree. [8 marks]
3. A **get**(int i), public method. This should return the i-th value of the tree, in comparison order. In particular get(0) is the smallest and get(sizes[0]-1) the largest element. Return null if the index is out of bounds. Hint: you need to exploit the sizes array for an efficient implementation. [16 marks]

4. boolean **insert**(A x), public method. This should insert the value x into the tree, but only change the tree if the value was not there already. It should return true if and only if the tree was changed. Notice that if the tree changes the sizes array is affected and needs to be updated in the right places. If there is not enough room to insert the element x use the grow() method to make room. [20 marks]
5. void **grow**(), private method. When trying to insert values we may run out of index room, because the arrays are too small. So we want to replace them with bigger ones and copy the data across in the right way. For half marks (10), simply double the sizes of the arrays and copy everything across to the same place. For full marks (20) restructure the tree so that the new tree is balanced in the following sense: the sizes of left/right subtrees for every node can differ by at most 1. The simple version of grow() can be implemented as an O(n) operation, the other requires O(n*d). [20 marks].
6. boolean **delete**(A x), public method. Delete the element x from the tree. Return true iff the tree was changed. When the value x is deleted at position i it should be replaced with either the largest element of the left subtree of i, or the smallest from the right subtree of i (always pick the subtree with more elements), unless both of these are empty. For example, if we delete the root value (6) of our example, it should be replaced with the 7, removing it from its old position. Again, notice that the sizes array would be affected in various places, in the example it goes down by 1 in positions 0, 2 and 5. [20 marks]

Notice that the methods labelled here as "`private`" have in the provided skeletal code the access modifier "`protected`". Reason: this enables white-box testing. Do not change these access modifiers.
For the submission just submit your updated LinklessTrees.java file – no .jar files, .zip files etc.