

CSCI B505 Spring 20: Programming assignment 4

Due online: Friday, April 10, 11:59pm EST

Submit your work online using Canvas. You can use Python, Java or C++. You can (but not required to) use files `tree_array.py` and `TreeArray.java` as a starter file. Ask AI's if you have any questions.

What to submit

For this assignment you will be submitting a source code. Please, follow good coding practices: use indentation, write comments, etc. The code should include the following:

- Implementation of methods as described below.
- Running time for all methods.
- An exhaustive set of tests.

Motivation

Python's list and Java's ArrayList are arrays with a variable size: you can insert an element at an arbitrary position and remove an element from the array. However, insertion/removal can take linear time in the worst case. Can we do them faster?

There is an interesting solution: implementing a dynamic array using a Binary Tree. It supports the same operations as a variable-size array; however, when implemented using balanced Binary Tree, it requires $O(\log n)$ time for all operations. For simplicity, we ask you to implement this data structure using an **unbalanced** Binary Tree.

Main idea

Consider an in-order (left-to-right) traversal x_0, \dots, x_{n-1} of a tree. We index our elements in this order: i.e. the index 0 corresponds to element x_0 , index 1 corresponds to element x_1 , etc. Consider an example in Figure 1 for explanation.

We never store indices explicitly (because their update would again take linear time). How can we find the element with index i in the tree? To do this, for each node, we also store its size: the number of nodes in the corresponding subtree. I.e. a node should store five values: data (a corresponding element), size, parent, and left and right children.

Let u be the current node. let ℓ and r be its left and right children respectively and assume that we want to find an element with index i in this subtree. Then there are three cases (if $\ell = null$ then $\ell.size = 0$):

- $i < \ell.size$. Then we should find an element with index i in ℓ .
- $i = \ell.size$. Then u is the required element.
- $i > \ell.size$. Then we should find an element with index $i - \ell.size - 1$ in r .

Check Lab 8 slides for some examples.


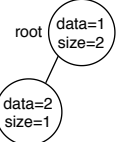
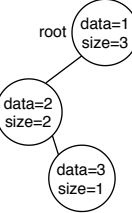
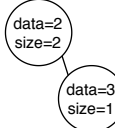
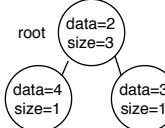
	Initially	Insert(0,1)	Insert(0,2)	Insert(1,3)	remove(2)	insert(0,4)
Array representation	[]	[1]	[2, 1]	[2, 3, 1]	[2, 3]	[4, 2, 3]
Tree representation	root = null					

Figure 1: Example

Implementation

You should implement the described data structure using an **unbalanced** Binary Tree. You can assume that all keys are integers. Your data structure must support the operations described below. If an index is out of bounds, throw *IndexError* for Python or *ArrayIndexOutOfBoundsException* for java.

- **size()** – returns the number of nodes in the tree.
- **find(i)** – this is an auxiliary function which returns a node at index i . Implementation is explained in the previous section.
- **get(i)** – returns an element at index i . Use *find(i)* and return the value in the node.
- **set(i, x)** – returns an element at index i . Use *find(i)* and change the data in the node.
- **remove(i)** – removes an element at index i . Use *find(i)* and remove the node from the tree. The node removal algorithm is identical to removal from a Binary Search Tree.
- **insert(i, key)** – inserts *key* into the tree at position i , i.e. between $(i - 1)$ -th and i -th elements. If $i = 0$, the element is inserted in the beginning, and if $i = \text{size}()$, the element is inserted in the end. If $i = 0$, then you should create a new leftmost node in the tree. Otherwise, let u be a node with index $i - 1$. You want to create an element which would be the next one in in-order. It suffices to create a new leftmost node in its right subtree (the idea is somewhat similar to tree removal).
- **inorder()** – returns a Python's *list* or Java's *List* of elements of the tree in in-order (left-to-right). I.e. it returns an array representation of the tree. Must work in linear time.

Size recalculation

Each node stores the size of its subtree (as a field in class Node). When the tree doesn't change, sizes also don't change, and we can access them in constant time. However, *insert* and *remove* modify the tree, and therefore sizes of some nodes could change. Therefore, we have to recalculate sizes of such nodes.

The good news are that the only nodes which could be affected are nodes on the path from the root to the point where the change happened. To do this, you should start with the lowest affected node: for insertion it's the an inserted node, and for removal it's the parent of the node which was finally removed. After that, you should use *parent* links to reach the root (function **fix_sizes**). For each visited node, you should recalculate its size (using function **Node.fix_size**). It's not allowed to recalculate sizes using a function which recursively traverses the entire tree, since it'll take linear time.

Running times

In a comment in the beginning of your source file, show worst-case time complexity for all methods (you don't have to prove it).

Tests

Please check correctness of your data structure with an exhaustive set of tests.