Binary Search Tree

Introduction

As requested by our professor at the subject Basic Data Structure 2 (EDB2), at Federal University of Rio Grande do Norte (UFRN), we shall implement a binary search tree, which is described here. Not only that, we must also add some useful methods, besides the standards ones, which are:

- `nthElement()` - Returns the nth element of a tree (indexing by 1) from in-order visitation method.
- `position()` - Returns the element of a given position (relative to in-order method & indexed by 1).
- `median()` - Returns the element that has the median of the tree.
- `isFull()` - Returns a `boolean` value telling if the binary tree is a full tree.
- `isComplete()` - Returns a `boolean` value telling if the binary tree is a complete tree.
- `toString()` - Returns a string containing the visitation sequence by level.

Instructions

By default, this program will recieve two files:

- Keys file A file that will generate the initial tree, all separated by spaces, i.e.:

```
12 30 50 10 11 5 30 90
```

- Commands file A file containing commands, one on each line, some of them needs args, you can put them by a single space, i.e.:

```
CHEIA
ENESIMO 10
INSIRA 22
IMPRIMA
```

Supported commands

In this version, we will support the following commands:

- `ENESIMO N` - Will return the nth element on the in-order representation of the binary search tree.
- `POSICAO N` - Will return the position that the element is on a in-order representation of the binary search tree.
- `MEDIANA` - Will return the median of an in-order representation of a binary search tree.
- `CHEIA` - Will tell if the tree is a full tree (All empty nodes are on the `last` level).
- `COMPLETA` - Will tell if the tree is a complete tree (All empty nodes are on the `last` or `last-1` levels).
- `IMPRIMA` - Will return the binary search tree represented by level.
- `REMOVA N` - Will remove key `N` from the binary search tree.
- `INSIRA N` - Will insert key `N` into the binary search tree.

Compilation

1. In order to compile, some dependencies are required:

- `make`
- `gcc` || `clang`

2. Clone the repository onto your computer and run on terminal:

```
make
```

3. Run the program with:

```
./bintree [data-file] [command-file]
# ex: ./bintree test/1.tree test/1.cmd
```

Report

Approaches Followed

To implement a usable Binary Search Tree, we adopted the ideia of a object Tree, which is a composition of numerous Nodes (another object implemented) and containing information of it's root and it's number of nodes. Now about the Nodes. Our object Node has information over it's key (same as content), the number os Nodes on his left and right, and pointer connecting it to it's left-below, right-below, and above Nodes. With such information, we were able to completly implement the standard and new methods. For example, thanks to knowing the amount of Nodes to the left and right of a given Node, we were able to implement `nthElement` , `position` and `median` methods, with complexity `O(h)` , where **h** is Tree's height.

To assist onto implementation, we cretead several auxiliary private methods, so we could, for example, be able to determine Tree's height, or update informations on Nodes by every insertion or remotion, or even calculate distance over one Node to Tree's root Node. Inside most methods, this auxiliary ones were called to facilitate the general implementation and complexity.

Asymptotic Complexity Analysis

We shall briefly explain all methods complexities, besides the standard binary search tree functions. Observe that this isn't a deep justification over complexity. Also consider **h** as **Tree's Height**. In a balanced binary tree, **`h == log(n)`** .

- `nthElement()` : `O(h)` . We have inside each node two variables holding number of nodes in left and right sub-trees. Then, to discover the nth element we just compare if a given `node.l_subtrees` is bigger/smaller than and then we dig down into the correct direction, getting a height based asymptotic complexity (`O(h)`).
- `position()` : `O(h)` . Same principle as the nthElement (since it has an internal call inside for the comparissons).
- `median()` : `O(h)` . We call the `nthElement()` function on `tree.size+1/2` (if `tree.size` it's an odd number) or the smaller element between the `tree.size/2` and `tree.size+1/2` (if `tree.size` is an even number). If it derivates mostly from a `O(h)` function, then it is also a `O(h)` .

- `isFull()`: `O(n)`. It depends on `maxHeight()` function, wich digs down onto both left and right Nodes.
- `isComplete()`: `O(h*n)`. Same ideia of `toString()` was used, so that we could determine the number of Nodes on it's last level, but while checking each Node, a function to determine Node's distance to Tree root is executed, `distRoot()`,which has `O(h)` complexity. After that, it's complexity is determinated by `maxHeight()` function. Even though multiple `O(n)` methods are called, this procedure still classifies as `O(n)`.
- `toString()`: `O(n)`. It's impossible print all elements without going through all elements.

Authorship

Programs developed by *Daniel Guerra* and *Felipe Ramos*, on 2018.2