Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Department Informatik
Markus Püschel
Peter Widmayer
Thomas Tschager
Tobias Pröger
Tomáš Gavenčiak

1st December 2016

Algorithmen & Datenstrukturen Exercise Sheet P11 AS 16

Hand-in: Before Thursday, 8th December 2016 10:00 via the online judge (source code only).

Exercise P11.1 *Binary trees.*

A binary search tree is a rooted tree where every node X has a *key* value $\text{key}(X)$ and may have a *left child* $\text{left}(X)$ and/or a *right child* $\text{right}(X)$. Let $S(X)$ be the set of all the keys in the subtree under X (including $\text{key}(X)$). The main rule of search trees states that all the values in $SN(\text{left}(X))$ are smaller than $\text{key}(X)$, and all the values in $S(\text{right}(X))$ are larger than $\text{key}(X)$. In the exercises, all the keys are different for simplification. A parent $\text{parent}(X)$ is a node such that X is its left or right child.

In the lecture, you have seen AVL trees, which are *balanced* binary trees, e.g. their algorithm not only maintains the search tree rule, but also makes sure that the tree is not very deep. In this exercise you will implement several operations for binary search trees *without* rebalancing. In very deep unbalanced trees, the operations may take a long time, but in our case, the data ensures that the depth is never too large, namely never more than 50. With this assumption, we recommend using recursive functions to implement the operations.

As usual, we provide a program skeleton template as an Eclipse project archive on the lecture website. The template implements all the input and output and provides working class `TreeNode`, so you only need to implement the missing functions, and modify it lightly for subtask B. See the notes below for some details of the design of class `TreeNode`.

Task A The first task is to implement the functions `insert`, `contains` and `delete`. See below for examples.

`TreeNode.insert(val)` inserts the value *val* as a new *leaf node*[1] without modifying the rest of the tree[2]. You may assume that *val* is not present in the tree before the call.

`TreeNode.contains(val)` checks if *val* is present in the tree without modifying it.

`TreeNode.delete(val)` removes the value *val* from the tree in the following way. Let X be the node with $\text{key}(X) = val$. If node X is a leaf, just remove X from its parent. If X has exactly one child Y (left or right), replace X by Y in $\text{parent}(X)$ (so leaving X out of the tree). If X has both children, let Y be the node with closest *larger* key than X. Note that such Y is always in $S(\text{right}(X))$. Now delete Y from the tree (recursively by deleting $\text{key}(Y)$ from subtree under X) and then set $\text{key}(X) = \text{key}(Y)$ (moving key from Y to X, so only the original key of X got deleted). You may assume that the deleted element does exists in the tree.

[1] *Leaf* is a node without any children.
[2] Note there is always an unique position where the new key should go.

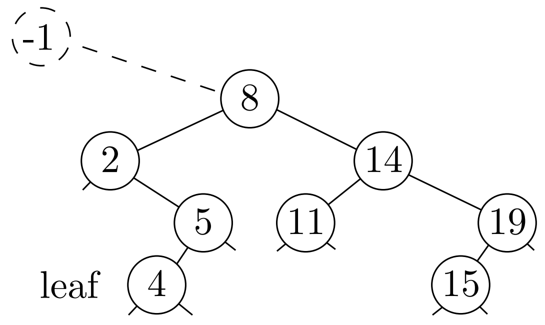
Task B Modify class `TreeNode` to also remember the size of the subtree under the node, e.g. remember size$(X) = |S(X)|$. Modify `insert` and `delete` to properly update these values. For that observe that when you add or remove a node, only the sizes on the path "up" to the root need to be updated, and only by +1 or -1.

Then implement function `TreeNode.findByRank(r)` that returns r-th smallest key of the tree, counting from 0. (So `findByRank(0)` returns the smallest element.)

The way to find r-th element in a subtree under X is the following: If $k < \text{size}(\text{left}(X))$ then the desired element is also r-th element under left(X), so search recursively there. If $k = \text{size}(\text{left}(X))$ then the desired element is key(X). And if $k > \text{size}(\text{left}(X))$ then the desired element is $(k - \text{size}(\text{left}(X)) - 1)$-th element under right$(X)$.

Example

Consider inserting elements in the following order to an empty tree: 8, 2, 14, 5, 11, 19, 15, 4. The resulting tree (with a special element -1 as the root – see Notes below) is the following, and corresponds to bracket notation `(2 ((4) 5)) 8 ((11) 14 ((15) 19))`. For effects of delete and find by rank, see the input examples below.



In/output The input consists of several commands. Every command is a letter and optionally a number, all separated by spaces or newlines. (The judge inputs generally contain one command per line, but the template reads any combination of spaces and newlines.) Some of the commands print a one-line response to the output. The program starts with an empty tree.

`I val` inserts value *val* into the tree. `C val` checks if *val* is in the tree, printing `YES` or `NO`. `D val` deletes *val* from the tree. `P` prints the tree in bracket notation (see example below; this is implemented by the template). `R rank` prints *rank*-th element of the tree (counting elements from 0). `X` stops the program.

All the values are integers between 0 and 10 000 000, there at most 100 000 elements in the tree and at most 200 000 operations. The depth of the tree is below 50.

Grading You will get 1 bonus point for every 100 judge points, rounded down, with maximum of 200 judge points. The program should perform every operation (except `P`) in time $\mathcal{O}(d)$ where d is the depth of the tree. Submit your `Main.java` at `https://judge.inf.ethz.ch/team/websubmit.php?cid=18985&problem=DA_P11.1`, enroll password is "`quicksort`".

`test1` and `judge1` (30 points) only contain inserts, prints and membersip test, `test2` and `judge2` (70 points) also contain deletes and `test3` and `judge3` (100 points) contain all the operation types.

Example

Input (more commands on a line) and output (of the last command)

```
I 8 I 2 I 14 I 5 I 11 P        | (2 (5)) 8 ((11) 14)
C 5                            | YES
I 19 I 15 C 4                  | NO
I 4 C 4                        | YES
C 3                            | NO
P                              | (2 ((4) 5)) 8 ((11) 14 ((15) 19))
D 2 D 5 C 4                    | YES
P                              | (4) 8 ((11) 14 ((15) 19))
D 4 D 14 D 11 C 15             | YES
P                              | 8 (15 (19))
I 2 I 5 P                      | (2 (5)) 8 (15 (19))
R 1                            | 5
R 3                            | 15
R 4                            | 19
X
```

Notes The tree is implemented by just one class `TreeNode`. Any missing left and right children should be `null`, as well as missing parent of the root. In practice, you would probably also implement class like `BinaryTree` that would represent the tree and hide the nodes, but for our simple exercise that is not necessary.

Our trees also always have a root node with "special" key `-1` (that is always smaller than any other keys in our data), and all the operations to be performed on the tree are applied to the root. This actually simplifies many of the operations, as many of them would need special cases for empty tree or modifying the root of the tree (e.g. root has no parents), while this special root is never modified, searched for etc.

Reminder: Do not forget to properly update the `parent` node of every node that you modify.

As usual, the project archive also contains more test data for your local testing. You can use the provided `Judge.java` program to run your `Main.java` on your computer on all the provided tests – just open and run `Judge.java` in the project as you would run `Main.java`. This does not change the way that your `Main.java` works and is just an extra tool. Also, the local test data are of course different and generally much smaller than the data that are used in the online judge.