Treecheck - Protocol

Ormir Gjurgjej, Nitika Kumar

May 10, 2016

1 Rekursive Funktionen

1.1 push

In order to add a new node to the tree, the push function needs to be executed.
The value of the node that the user wants to insert is given over to the function as a parameter (int k). First of all, the keys of the tree (available nodes) are compared to the parameter (k) starting with the root. If the current key is smaller than k and there is no node on the right hand side of key (as nullptr is a pointer pointing to null meaning there is no node available), a new node is then inserted with the value of k and so the balance is incremented (bal++).
Otherwise if the key is smaller than k and there is already another node on the right hand side of key, the push function is called again and "pushes" k onto the new key which is the node hanging on the right hand side of the previous key (recursion).
However, if this is not the case either we move on to the next if condition and check whether key is greater than k and there is no node on the left hand side of key. If this condition is true, the new node with the value of k is added to the left of key. In addition to that, the balance variable decrements (ball–) as the parent node of k has only one new node hanging on one side and needs another one on the right to balance it out.
If there is a node on the left hand side of key, the function push is called again in order to turn this node into the new key to check all conditions again and compare them to k (recursion). Again, the variable bal is decremented.
The very last else statement tells the user that k is already available in the

tree and will not be added as it is a duplicate.

```
1  void Node::push(int k) {
2      if (key < k && right == nullptr) {
3          right = new Node(k);
4          bal++;
5      } else if (key < k && right != nullptr) {
6          right->push(k);
7          bal++;
8      } else if (key > k && left == nullptr) {
9          left = new Node(k);
10         bal--;
11     } else if (key > k && left != nullptr) {
12         left->push(k);
13         bal--;
14     } else {
15         std::cout << "Duplicate key " << k << std::endl;
16     }
17 }
18
```

Listing 1: Push function

1.2 balance

The purpose of the balance function is to show the balance of the current AVL tree.
Here, the function looks at each node and checks whether or not there is something on the left or right hand side of the node. It first makes its way down to the left of the tree until it reaches the very last node. Afterwards it goes up the nodes again and checks for some on the right hand side until it has finished checking the entire tree. (recursion)
Eventually, the variable bal decides whether or not the tree is indeed an AVL tree - as in well balanced. If bal is smaller than 2 and greater than -2, the function returns true. However, if the condition is not true, it is not a well balanced tree and the user sees the corresponding output on the screen.

```
1 void Node::balance() const {
2     if (left != nullptr) {
3         left->balance();
4     }
5
```

```
6       if (right != nullptr) {
7           right->balance();
8       }
9
10      std::cout << "bal(" << key << ") = " << bal;
11
12      if (bal < 2 && bal > -2) {
13          std::cout << std::endl;
14      } else {
15          std::cout << " (AVL Violation!)" << std::endl;
16      }
17  }
```

Listing 2: Balance function

2 Estimation

The estimation of AVL Trees (for the average as well as worst cases):

Access	Search	Insertion	Deletion
O(log(n))	O(log(n))	O(log(n))	O(log(n))

Table 1: Big O Notation